Engineering augmented suffix sorting algorithms

Felipe A. Louza

Advisor: Guilherme P. Telles
Co-advisor: Simon Gog (KIT/Germany)

Institute of Computing (IC)
UNICAMP, Brazil

July 27, 2017

Outline

1. Introduction

2. Burrows-Wheeler transform and LCP array construction in constant space

3. Optimal suffix sorting and LCP array construction for constant alphabets

4. Inducing enhanced suffix arrays for string collections

5. Contributions

6. References

Introduction

Suffix sorting:

- Is the problem of lexicographically ordering all suffixes of a string T of length n.
- Is a fundamental problem in string processing related to:
  - Suffix array (SA) construction [MM93, GBYS92].
  - Burrows-Wheeler transform (BWT) [BW94].

	1	2	3	4	5	6	7
T =	b	a	n	a	n	a	$

all suffixes		sorted suffixes
banana$		$
anana$		a$
nana$	$\xrightarrow{\textit{sort}}$	ana$
ana$		anana$
na$		banana$
a$		na$
$		nana$

We assume that T always ends with $T[n] = \$$, called *sentinel*, which is not present elsewhere in T and precedes every symbol.

Introduction

Suffix sorting:

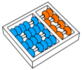
- Is the problem of lexicographically ordering all suffixes of a string T of length n.
- Is a fundamental problem in string processing related to:
  - Suffix array (SA) construction [MM93, GBYS92].
  - Burrows-Wheeler transform (BWT) [BW94].

	1	2	3	4	5	6	7
T =	b	a	n	a	n	a	\$

all suffixes		sorted suffixes		i	SA		
banana\$		\$		1	7		
anana\$		a\$		2	6		
nana\$	$\xrightarrow{\text{sort}}$	ana\$	$\xrightarrow[\text{array}]{\text{suffix}}$	3	4	$\xrightarrow{\text{BWT}}$	annb\$aa
ana\$		anana\$		4	2		
na\$		banana\$		5	1		
a\$		na\$		6	5		
\$		nana\$		7	3		

We assume that T always ends with $T[n] = \$$, called *sentinel*, which is not present elsewhere in T and precedes every symbol.

Introduction



LCP-array:

- SA and BWT are commonly accompanied by the longest common prefix (LCP) array.
- Together, they are the basis of important full-text indexes.

Figure by D. Kempa.

Introduction

LCP-array:

- SA and BWT are commonly accompanied by the longest common prefix (LCP) array.
- Together, they are the basis of important full-text indexes.

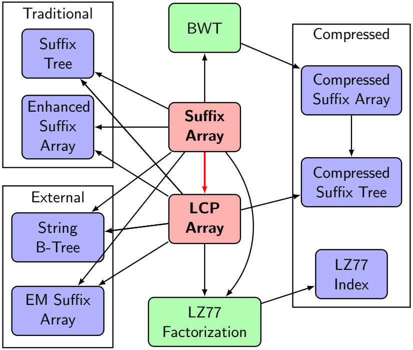


Figure by D. Kempa.

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].
- In 2013, Nong [Non13] presented SACA-K, the first optimal algorithm.

Remark:

- This problem may be considered essentialy solved [Kär16].

Recent advances:

- Alternatives for external memory and parallel architectures*.
- Compute the LCP array and other structures simultaneously during suffix sorting.

Augmented Suffix Sorting

Our contributions:

1. BWT in-place and LCP array construction.
2. SA and LCP array construction in optimal time/space.
3. Augmented suffix sorting for string collections.

*One key to this approaches is the use of fast sequential algorithms.

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].
- In 2013, Nong [Non13] presented SACA-K, the first optimal algorithm.

Remark:

- This problem may be considered essentialy solved [Kär16].

Recent advances:

- Alternatives for external memory and parallel architectures*.
- Compute the LCP array and other structures simultaneously during suffix sorting.

Augmented Suffix Sorting

Our contributions:

1. BWT in-place and LCP array construction.
2. SA and LCP array construction in optimal time/space.
3. Augmented suffix sorting for string collections.

*One key to this approaches is the use of fast sequential algorithms.

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].
- In 2013, Nong [Non13] presented SACA-K, the first optimal algorithm.

Remark:

- This problem may be considered essentialy solved [Kär16].

Recent advances:

- Alternatives for external memory and parallel architectures*.
- Compute the LCP array and other structures simultaneously during suffix sorting.

Augmented Suffix Sorting

Our contributions:

1. BWT in-place and LCP array construction.
2. SA and LCP array construction in optimal time/space.
3. Augmented suffix sorting for string collections.

*One key to this approaches is the use of fast sequential algorithms.

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].
- In 2013, Nong [Non13] presented SACA-K, the first optimal algorithm.

Remark:

- This problem may be considered essentialy solved [Kär16].

Recent advances:

- Alternatives for external memory and parallel architectures*.
- Compute the LCP array and other structures simultaneously during suffix sorting.

Augmented Suffix Sorting

Our contributions:

1. BWT in-place and LCP array construction.
2. SA and LCP array construction in optimal time/space.
3. Augmented suffix sorting for string collections.

* One key to this approaches is the use of fast sequential algorithms.

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].
- In 2013, Nong [Non13] presented SACA-K, the first optimal algorithm.

Remark:

- This problem may be considered essentialy solved [Kär16].

Recent advances:

- Alternatives for external memory and parallel architectures*.
- Compute the LCP array and other structures simultaneously during suffix sorting.

Augmented Suffix Sorting

Our contributions:

1. BWT in-place and LCP array construction.
2. SA and LCP array construction in optimal time/space.
3. Augmented suffix sorting for string collections.

* One key to this approaches is the use of fast sequential algorithms.

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].
- In 2013, Nong [Non13] presented SACA-K, the first optimal algorithm.

Remark:

- This problem may be considered essentialy solved [Kär16].

Recent advances:

- Alternatives for external memory and parallel architectures*.
- Compute the LCP array and other structures simultaneously during suffix sorting.

Augmented Suffix Sorting

Our contributions:

1. BWT in-place and LCP array construction.
2. SA and LCP array construction in optimal time/space.
3. Augmented suffix sorting for string collections.

* One key to this approaches is the use of fast sequential algorithms.

Notations

Strings:

- Let T be a string of length n, $T = T[1, n]$, over a ordered alphabet of size σ.

Alphabet:

- constant: has size $\sigma = O(1)$.
- integer: has size $\sigma = n^{O(1)}$.
- unbounded: otherwise.

T =

1	2	3	4	5	6	7
b	a	n	a	n	a	$

- $T[i]$ is the i-th symbol of T.
- $T[i, j]$ is the substring including symbols from $T[i]$ to $T[j]$, $i \leq j$.
- $T[1, i]$ is a prefix and a $T[i, n]$ is a suffix of T.

Space:

- A string $T[1, n]$ is stored in $n \log \sigma$ bits.
  - 1 byte: ASCII.
- A permutation of integers in $[1, n]$ is stored using $n \log n$ bits.
  - 4 bytes if $n < 2^{31}$, 8 bytes otherwise.

Workspace:

- Is the extra space needed in addition to the space used by the input and output.

Notations

Strings:

- Let T be a string of length n, $T = T[1, n]$, over a ordered alphabet of size σ.

Alphabet:

- constant: has size $\sigma = O(1)$.
- integer: has size $\sigma = n^{O(1)}$.
- unbounded: otherwise.

	1	2	3	4	5	6	7
T =	b	a	n	a	n	a	$

- $T[i]$ is the i-th symbol of T.
- $T[i, j]$ is the substring including symbols from $T[i]$ to $T[j]$, $i \leq j$.
- $T[1, i]$ is a prefix and a $T[i, n]$ is a suffix of T.

Space:

- A string $T[1, n]$ is stored in $n \log \sigma$ bits.
  - 1 byte: ASCII.
- A permutation of integers in $[1, n]$ is stored using $n \log n$ bits.
  - 4 bytes if $n < 2^{31}$, 8 bytes otherwise.

Workspace:

- Is the extra space needed in addition to the space used by the input and output.

Notations

Strings:

- Let T be a string of length n, $T = T[1, n]$, over a ordered alphabet of size σ.

Alphabet:

- constant: has size $\sigma = O(1)$.
- integer: has size $\sigma = n^{O(1)}$.
- unbounded: otherwise.

	1	2	3	4	5	6	7
T =	b	a	n	a	n	a	$

- $T[i]$ is the i-th symbol of T.
- $T[i, j]$ is the substring including symbols from $T[i]$ to $T[j]$, $i \leq j$.
- $T[1, i]$ is a prefix and a $T[i, n]$ is a suffix of T.

Space:

- A string $T[1, n]$ is stored in $n \log \sigma$ bits.
  - 1 byte: ASCII.
- A permutation of integers in $[1, n]$ is stored using $n \log n$ bits.
  - 4 bytes if $n < 2^{31}$, 8 bytes otherwise.

Workspace:

- Is the extra space needed in addition to the space used by the input and output.

Notations

Strings:

- Let T be a string of length n, $T = T[1, n]$, over a ordered alphabet of size σ.

Alphabet:

- constant: has size $\sigma = O(1)$.
- integer: has size $\sigma = n^{O(1)}$.
- unbounded: otherwise.

	1	2	3	4	5	6	7
T =	b	a	n	a	n	a	$

- $T[i]$ is the i-th symbol of T.
- $T[i, j]$ is the substring including symbols from $T[i]$ to $T[j]$, $i \leq j$.
- $T[1, i]$ is a prefix and a $T[i, n]$ is a suffix of T.

Space:

- A string $T[1, n]$ is stored in $n \log \sigma$ bits.
  - 1 byte: ASCII.
- A permutation of integers in $[1, n]$ is stored using $n \log n$ bits.
  - 4 bytes if $n < 2^{31}$, 8 bytes otherwise.

Workspace:

- Is the extra space needed in addition to the space used by the input and output.

Notations

SA and LCP array:

- SA: is an array of integers in the range $[1, n]$ that gives the lexicographic order of all suffixes.
- LCP array: stores the length of the longest common prefix (lcp) of two consecutive suffixes.
- The arrays can be partitioned into σ buckets, one for each symbol in the alphabet.

T =

1	2	3	4	5	6	7
b	a	n	a	n	a	$

i	SA	LCP	sorted suffixes $T[\mathrm{SA}[i], n]$
1	7	0	$
2	6	0	a$
3	4	1	ana$
4	2	3	anana$
5	1	0	banana$
6	5	0	na$
7	3	2	nana$

- The range minimum query (rmq) *w.r.t* LCP:
  - $\mathrm{rmq}(i, j) = \min_{i<k\leq j}\{\mathrm{LCP}[k]\}$.
  - Given T and its LCP array we have:

$$\mathrm{lcp}(T[\mathrm{SA}[i], n], T[\mathrm{SA}[j], n]) = \mathrm{rmq}(i, j)$$

Notations

BWT:

- A reversible transformation that produces a permutation of T which tends to group the occurrences of a symbol in runs [BW94].
- The BWT can be obtained sorting all the n circular shifts of T, and taking the last column.
- Can be defined in terms of SA:

$$\mathrm{BWT}[i] = \begin{cases} T[SA[i]-1] & \text{if } SA[i]-1 > 0 \\ \$ & \text{otherwise.} \end{cases}$$

$$\text{banana\$} \xrightarrow[\textit{shifts}]{\textit{circular}} \begin{array}{c} \hline \text{banana\$} \\ \text{\$banana} \\ \text{a\$banan} \\ \text{na\$bana} \\ \text{ana\$ban} \\ \text{nana\$ba} \\ \text{anana\$b} \\ \hline \end{array} \xrightarrow{\textit{sort}} \begin{array}{c} \text{F} \quad\quad \text{L} \\ \hline \text{\$banana} \\ \text{a\$banan} \\ \text{ana\$ban} \\ \text{anana\$b} \\ \text{banana\$} \\ \text{na\$bana} \\ \text{nana\$ba} \\ \hline \end{array} \xrightarrow[\textit{column}]{\textit{last}} \text{annb\$aa}$$

Notations

BWT:

- A reversible transformation that produces a permutation of T which tends to group the occurrences of a symbol in runs [BW94].
- The BWT can be obtained sorting all the n circular shifts of T, and taking the last column.
- Can be defined in terms of SA:

$$\text{BWT}[i] = \begin{cases} T[SA[i]-1] & \text{if } SA[i]-1 > 0 \\ \$ & \text{otherwise.} \end{cases}$$

banana\$ $\xrightarrow[\textit{shifts}]{\textit{circular}}$

banana\$
\$banana
a\$banan
na\$bana
ana\$ban
nana\$ba
anana\$b

$\xrightarrow{\textit{sort}}$

F L
\$banana
a\$banan
ana\$ban
anana\$b
banana\$
na\$bana
nana\$ba

$\xrightarrow[\textit{column}]{\textit{last}}$ annb\$aa

i	SA	L	F
1	7	a	\$
2	6	n	a\$
3	4	n	ana\$
4	2	b	anana\$
5	1	\$	banana\$
6	5	a	na\$
7	3	a	nana\$

Notations

BWT:

- A reversible transformation that produces a permutation of T which tends to group the occurrences of a symbol in runs [BW94].
- The BWT can be obtained sorting all the n circular shifts of T, and taking the last column.
- Can be defined in terms of SA:

$$\mathrm{BWT}[i] = \begin{cases} T[SA[i]-1] & \text{if } SA[i]-1>0 \\ \$ & \text{otherwise.} \end{cases}$$

banana$ $\xrightarrow[\textit{shifts}]{\textit{circular}}$

banana$
$banana
a$banan
na$bana
ana$ban
nana$ba
anana$b

$\xrightarrow{\textit{sort}}$

F L
$banana
a$banan
ana$ban
anana$b
banana$
na$bana
nana$ba

$\xrightarrow[\textit{column}]{\textit{last}}$ annb$aa

i	SA	L	F
1	7	a	$
2	6	n	a$
3	4	n	ana$
4	2	b	anana$
5	1	$	banana$
6	5	a	na$
7	3	a	nana$

- LF-mapping:
  - The i-th symbol α in column L corresponds to the i-th symbol α in column F.

Outline

BWT and LCP construction in constant space

BWT:

- Standard construction using SA in $O(n)$ time.
  - Workspace: $O(n \log n)$ bits $\Rightarrow$ to store SA$[1, n]$.
- Direct BWT construction (without SA):
  - The most space-efficient is the $O(n^2)$ time BWT in-place due to Crochemore *et al.* [CGKL15].

BWT in-place and LCP array:

- Our contribution:
  - We extend the BWT in-place [CGKL15] to also compute the LCP array in $O(n^2)$ time using O(1) workspace.

BWT and LCP construction in constant space

BWT:

- Standard construction using SA in $O(n)$ time.
  - Workspace: $O(n \log n)$ bits $\Rightarrow$ to store SA$[1, n]$.
- Direct BWT construction (without SA):
  - The most space-efficient is the $O(n^2)$ time BWT in-place due to Crochemore *et al.* [CGKL15].

BWT in-place and LCP array:

- Our contribution:
  - We extend the BWT in-place [CGKL15] to also compute the LCP array in $O(n^2)$ time using O(1) workspace.

Related Work

BWT in-place [CGKL15]:

- Overwrites the input string T with the BWT, $n, n-1, \ldots, 1$:
  - At each step i we have BWT of suffix $T[s, n]$, called BWT(T_s), with $s = n - i + 1$
  - The position of \$ in step $i-1$ allows the construction of BWT(T_s).

	1	2	3	4	5	6	7	Step
$T =$	b	a	n	a	n	a	\$	1
	b	a	n	a	n	a	\$	2
	b	a	n	a	a	n	\$	3
	b	a	n	a	n	\$	a	4
	b	a	a	n	n	a	\$	5
	b	a	n	n	\$	a	a	6
BWT $=$	a	n	n	b	\$	a	a	7

Related Work

BWT in-place [CGKL15]:

- Incremental step:
  - Given $\mathrm{BWT}(T_{s+1})$, stored in $T[s+1, n]$:
    1. Replace \$ by $T[s]$.
    2. Find the local rank r of $T[s, n]$.
    3. Insert new suffix and preceding character \$ into $T[r]$.

	s	BWT	suffixes
	1	b	banana$
$s \rightarrow$	2	a	anana$
	3	a	$
	4	n	a$
	5	n	ana$
	6	a	na$
$p \rightarrow$	7	$	nana$

$\mathrm{BWT}(T_{s+1})$

	s	BWT	suffixes
	1	b	banana$
$s \rightarrow$	2	a	anana$
	3	a	$
	4	n	a$
	5	n	ana$
	6	a	na$
$p \rightarrow$	7	a	nana$

Replace \$

	s	BWT	suffixes
	1	b	banana$
	2	a	$
	3	n	a$
	4	n	ana$
$r \rightarrow$	5		a...
	6	a	na$
$p \rightarrow$	7	a	nana$

Find local rank r

	s	BWT	suffixes
	1	b	banana$
	2	a	$
	3	n	a$
	4	n	ana$
$r \rightarrow$	5	$	anana$
	6	a	na$
	7	a	nana$

$\mathrm{BWT}(T_s)$

- Step 2 (LF-mapping):
  - $T[p] \Rightarrow k$-th α in $\mathrm{BWT}(T_s)$ corresponds to k-th α in F.
  - To find r we count: number of $\alpha < T[s]$ in $T[s+1, n]$ and $\alpha = T[s]$ in $T[s+1, r]$.

Related Work

BWT in-place [CGKL15]:

- Analysis (for unbounded alphabets):
  - $O(n^2)$ time: each step i needs $O(n-i)$ time for:
    - Counting, inserting and moving symbols in $T[s, n]$.
  - $O(1)$ workspace:
    - Extra space needed for constant number of variables.

Our contribution

BWT in-place and LCP array:

- Overwrites T with the BWT and computes the LCP array:
  - At each step i we have BWT(T_s) and LCP(T_s) for the suffixes $\{T[s,n],\dots,T[n,n]\}$, with $s = n - i + 1$

	1	2	3	4	5	6	7	Step
$T =$	b	a	n	a	n	a	$	1
							0	

	1	2	3	4	5	6	7	Step
	b	a	n	a	n	a	$	2
						0	0	

	1	2	3	4	5	6	7	Step
	b	a	n	a	a	n	$	3
					0	0	0	

	1	2	3	4	5	6	7	Step
	b	a	n	a	n	$	a	4
				0	0	1	0	

	1	2	3	4	5	6	7	Step
	b	a	a	n	n	a	$	5
			0	0	1	0	2	

	1	2	3	4	5	6	7	Step
	b	a	n	n	$	a	a	6
		0	0	1	3	0	2	

	1	2	3	4	5	6	7	Step
BWT =	a	n	n	b	$	a	a	7
LCP =	0	0	1	3	0	0	2	

Our contribution

BWT in-place and LCP array:

- Incremental step:
  - Given BWT(T_{s+1}) and LCP(T_{s+1}), stored in $T[s+1, n]$ and LCP$[s+1, n]$:
  - Adding $T[s, n]$ to the solution requires evaluating two values of lcp, adjacent to $T[s, n]$.
    1. LCP$[r]$: lcp($T[a, n]$, $T[s, n]$) $\rightarrow$ $T[a, n]$ the largest suffix smaller than $T[s, n]$.
    2. LCP$[r+1]$: lcp($T[s, n]$, $T[b, n]$) $\rightarrow$ $T[b, n]$ the smallest suffix larger than $T[s, n]$.
  - We will show how to compute LCP$[r] = \ell_a =$ lcp($T[a, n]$, $T[s, n]$)*:

	s	LCP	BWT	suffixes
	1	-	b	banana$
	2	0	a	$
	3	0	n	a$
	4	1	n	ana$
$r \rightarrow$	5			anana$
	6	0	a	na$
	7	2	a	nana$

	s	LCP	BWT	suffixes	
	1	-	b	banana$	
	2	0	a	$	
	3	0	n	a$	
	4	1	n	ana$	$T[a, n]$
$r \rightarrow$	5	$\ell_a = ?$	$	anana$	$T[s, n]$
$r+1 \rightarrow$	6	$\ell_b = ?$	a	na$	$T[b, n]$
	7	2	a	nana$	

* lcp($T[s, n]$, $T[b, n]$) may be computed in a similar fashion.

Our contribution

BWT in-place and LCP array:

- Computing $\mathrm{LCP}[r] = \ell_a = \mathrm{lcp}(T[a, n], T[s, n])$:
  - $\mathrm{BWT}(T_{s+1})$ and $\mathrm{LCP}(T_{s+1})$ are sufficient to compute these values.
    1. $\ell_a = \mathrm{lcp}(T[a, n], T[s, n]) = lcp(T[a+1, n], T[s+1, n]) + 1$ if $T[s]$ is equal to the first symbol of $T[a, n]$, otherwise $\mathrm{LCP}[r] = 0$.
    2. We know the position of $T[s+1, n]$ is p from previous step.
    3. We must find the position p_{a+1} of $T[a+1, n]$ in $\mathrm{BWT}(T_{s+1})$.

$$\ell_a = \begin{cases} \mathrm{rmq}(p_{a+1}, p) + 1 & \text{if } T[p_{a+1}] = \mathrm{BWT}[s] \\ 0 & \text{otherwise.} \end{cases}$$

	4	1	n	ana$	$T[a, n]$
$r \rightarrow$	5	$\ell_a = ?$	$	anana$	$T[s, n]$
$p_{a+1} \rightarrow$	6	0	a	na$	$T[a+1, n]$
$p \rightarrow$	7	2	a	nana$	T_{s+1}

Our contribution

BWT in-place and LCP array:

- Computing ℓ_a:
  - To find position p_{a+1} in BWT(T_{s+1}):

    1. $T[a, n]$ has rank r, after the Shift it goes to $r - 1$.
    2. The symbol in BWT[p_{a+1}]=T[a] (the first symbol of $T[a, n]$), that has rank r in BWT(T_{s+1}).

       Question: Where is the symbol with rank r in BWT(T_{s+1}) ??
    3. Property:

       If BWT[p_{a+1}] $= T[s] \Rightarrow p_{a+1} \in [s+1, p)^*$ and p_{a+1} is the largest value in $[s+1, p)$.

       Otherwise, if $p_{a+1} \in [p, n] \Rightarrow$ BWT[p_{a+1}] $< T[s] \Rightarrow \ell_a = 0$.

	s	LCP	BWT	suffixes	
	1	-	b	banana$	
$s \rightarrow$	2	-	a	anana$	
	3	0	a	$	
	4	0	n	a$	
$r \rightarrow$	5	1	n	ana$	$T[a, n]$
$p_{a+1} \rightarrow$	6	0	a	na$	$T[a+1, n]$
$p \rightarrow$	7	2	$	nana$	$T[s+1, n]$

BWT(T_{s+1}) and LCP(T_{s+1})

*Any other symbol equal to $T[s]$ in [p,n] would have a rank $\geq r+1$.

Our contribution

BWT in-place and LCP array:

- Computing ℓ_a:
  - To find position p_{a+1} in BWT(T_{s+1}):
    1. $T[a,n]$ has rank r, after the Shift it goes to $r-1$.
    2. The symbol in BWT[p_{a+1}]=T[a] (the first symbol of $T[a,n]$), that has rank r in BWT(T_{s+1}).
       Question: Where is the symbol with rank r in BWT(T_{s+1}) ??
    3. Property:
       If BWT[p_{a+1}] $= T[s] \Rightarrow p_{a+1} \in [s+1, p)^\star$ and p_{a+1} is the largest value in $[s+1,p)$.
       Otherwise, if $p_{a+1} \in [p,n] \Rightarrow$ BWT[p_{a+1}] $< T[s] \Rightarrow \ell_a = 0$.

	s	LCP	BWT	suffixes	
	1	-	b	banana$	
$s \rightarrow$	2	-	a	anana$	
	3	0	a	$	
	4	0	n	a$	
$r \rightarrow$	5	1	n	ana$	$T[a,n]$
$p_{a+1} \rightarrow$	6	0	a	na$	$T[a+1,n]$
$p \rightarrow$	7	2	$	nana$	$T[s+1,n]$

BWT(T_{s+1}) and LCP(T_{s+1})

*Any other symbol equal to $T[s]$ in [p,n] would have a rank $\geq r+1$.

Our contribution

BWT in-place and LCP array:

- Computing ℓ_a:
  - To find position p_{a+1} in BWT(T_{s+1}):
    1. $T[a, n]$ has rank r, after the Shift it goes to $r - 1$.
    2. The symbol in BWT[p_{a+1}]=T[a] (the first symbol of $T[a, n]$), that has rank r in BWT(T_{s+1}).
       Question: Where is the symbol with rank r in BWT(T_{s+1}) ??
    3. Property:
       If $\text{BWT}[p_{a+1}] = T[s] \Rightarrow p_{a+1} \in [s+1, p)^\star$ and p_{a+1} is the largest value in $[s+1, p)$.
       Otherwise, if $p_{a+1} \in [p, n] \Rightarrow \text{BWT}[p_{a+1}] < T[s] \Rightarrow \ell_a = 0$.

	s	LCP	BWT	suffixes	
	1	-	b	banana$	
$s \rightarrow$	2	-	a	anana$	
	3	0	a	$	
	4	0	n	a$	
$r \rightarrow$	5	1	n	ana$	$T[a, n]$
$p_{a+1} \rightarrow$	6	0	a	na$	$T[a+1, n]$
$p \rightarrow$	7	2	$	nana$	$T[s+1, n]$

BWT(T_{s+1}) and LCP(T_{s+1})

* Any other symbol equal to $T[s]$ in [p,n] would have a rank $\geq r + 1$.

Our contribution

BWT in-place and LCP array:

- Computing ℓ_a:
  - Add: scan backwards BWT(T_{s+1}) from $T[p-1]$ to $T[s+1]$ until we find the first occurrence of BWT$[p_{a+1}] = T[s]$.
    1. If no symbol is found $\Rightarrow \ell_a = 0$
    2. We compute the minimum function for the lcp visited values, obtaining rmq(p_{a+1}, p) as soon as we find $T[p_{a+1}] = T[s]$

	s	LCP	BWT	suffixes	
	1	-	b	banana$	
$s \rightarrow$	2	-	a	anana$	
	3	0	a	$	
	4	0	n	a$	
$r \rightarrow$	5	1	n	ana$	$T[a, n]$
$p_{a+1} \rightarrow$	6	0	a	na$	$T[a+1, n]$
$p \rightarrow$	7	2	$	nana$	$T[s+1, n]$

BWT(T_{s+1}) and LCP(T_{s+1})

	s	LCP	BWT	suffixes	
	1	-	b	banana$	
	2	0	a	$	
	3	0	n	a$	
	4	1	n	ana$	$T[a, n]$
$r \rightarrow$	5	$\ell_a = 3$	$	anana$	$T[s, n]$
	6	$\ell_b = ?$	a	na$	$T[a+1, n]$
$p \rightarrow$	7	2	a	nana$	$T[s+1, n]$

BWT(T_s) and LCP(T_s)

- Computing ℓ_b is symmetric.

Our contribution

BWT in-place and LCP array:

- The analysis remains the same:
  - $O(n^2)$ time:
    - Additional cost: $O(n-i)$ time scan to compute ℓ_a, ℓ_b and to shift LCP.
  - $O(1)$ workspace:
    - Needs only four additional variables to store p_{a+1} and p_{b+1} and the values of ℓ_a and ℓ_b.
- The C code is quite short (45 lines) and clean.

LCP array in compressed representation:

- Our algorithm performs only sequential scans to compute BWT and LCP array.
  - lcp-values can be easily encoded and decoded during such scans using a universal code, such as Elias δ-codes [Eli75].

Tradeoff:

- We provide a theoretical time/space tradeoff for our algorithm when additional memory is allowed.

Our contribution

BWT in-place and LCP array:

- The analysis remains the same:
  - $O(n^2)$ time:
    - Additional cost: $O(n-i)$ time scan to compute ℓ_a, ℓ_b and to shift LCP.
  - $O(1)$ workspace:
    - Needs only four additional variables to store p_{a+1} and p_{b+1} and the values of ℓ_a and ℓ_b.
- The C code is quite short (45 lines) and clean.

LCP array in compressed representation:

- Our algorithm performs only sequential scans to compute BWT and LCP array.
  - lcp-values can be easily encoded and decoded during such scans using a universal code, such as Elias δ-codes [Eli75].

Tradeoff:

- We provide a theoretical time/space tradeoff for our algorithm when additional memory is allowed.

Outline

Optimal suffix sorting and LCP construction

Suffix array:

- Several algorithms to construct SA in $O(n)$ time:
  - SAIS: $O(n)$ time using $O(n \log n)$ bits of workspace [NZC11].
  - SACA-K: $O(n)$ time using $\sigma \log n$ bits of workspace [Non13].

LCP array:

- Can be constructed in $O(n)$ time given $T[1, n]$ and SA (e.g. [KLA+01, Man04, KMP09]).
  - $\Phi-$algorithm by [KMP09]: $O(n)$ time using $n \log n$ bits of workspace.

Suffix and LCP arrays:

- SAIS+LCP: Fischer [Fis11] showed how to modify SAIS to also compute the LCP array.
  - $O(n)$ time using $O(n \log n)$ bits of workspace.
- Our contribution:
  - SACA-K+LCP.
  - $O(n)$ time using $\sigma \log n$ bits of workspace.

Optimal suffix sorting and LCP construction

Suffix array:

- Several algorithms to construct SA in $O(n)$ time:
  - SAIS: $O(n)$ time using $O(n \log n)$ bits of workspace [NZC11].
  - SACA-K: $O(n)$ time using $\sigma \log n$ bits of workspace [Non13].

LCP array:

- Can be constructed in $O(n)$ time given $T[1, n]$ and SA (e.g. [KLA+01, Man04, KMP09]).
  - $\Phi-$algorithm by [KMP09]: $O(n)$ time using $n \log n$ bits of workspace.

Suffix and LCP arrays:

- SAIS+LCP: Fischer [Fis11] showed how to modify SAIS to also compute the LCP array.
  - $O(n)$ time using $O(n \log n)$ bits of workspace.
- Our contribution:
  - SACA-K+LCP.
  - $O(n)$ time using $\sigma \log n$ bits of workspace.

Optimal suffix sorting and LCP construction

Suffix array:

- Several algorithms to construct SA in $O(n)$ time:
  - SAIS: $O(n)$ time using $O(n \log n)$ bits of workspace [NZC11].
  - SACA-K: $O(n)$ time using $\sigma \log n$ bits of workspace [Non13].

LCP array:

- Can be constructed in $O(n)$ time given $T[1, n]$ and SA (e.g. [KLA$^+$01, Man04, KMP09]).
  - $\Phi-$algorithm by [KMP09]: $O(n)$ time using $n \log n$ bits of workspace.

Suffix and LCP arrays:

- SAIS+LCP: Fischer [Fis11] showed how to modify SAIS to also compute the LCP array.
  - $O(n)$ time using $O(n \log n)$ bits of workspace.
- Our contribution:
  - SACA-K+LCP.
  - $O(n)$ time using $\sigma \log n$ bits of workspace.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- Induced sorting: is to deduce the order of unsorted suffixes from a set of already sorted.
- The suffixes $T[i,n]$ are classified according to their rank relative to $T[i+1,n]$.

S, L and LMS-types:

- $T[i,n]$ is S-type if $T[i,n] < T[i+1,n]$, otherwise $T[i,n]$ is L-type. The last $T[n,n]$ is S-type.
- $T[i,n]$ is LMS-type if $T[i,n]$ is S-type and $T[i-1,n]$ is L-type. The last $T[n,n]$ is LMS[*].
- Consecutive LMS-suffixes are used to define LMS-substrings.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
type =	L	S	L	S	S	L	S	L	S	S	L	S	L	L	S
		*		*			*		*			*			*
LMS-subs =		a	n	a			a	n	a			a	n	a	\$
				a	a	n	a		a	a	n	a			\$

Key observations:

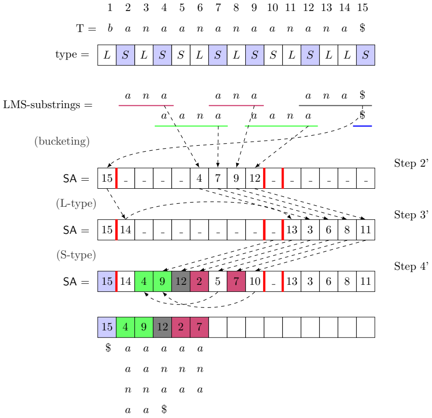
- LMS-suffixes are enough to induce the order of all suffixes of T.
- LMS-substrings can be used to reduced the problem.

[*] The suffix classification can be done in linear time.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

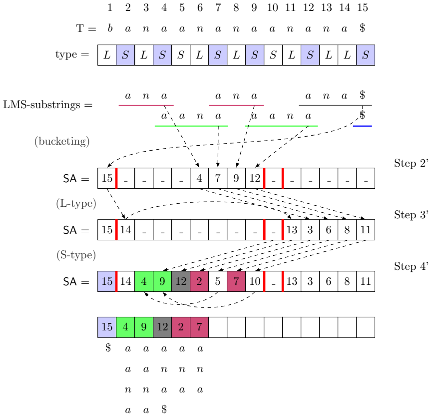
1. Step 1: Sorting the LMS-suffixes:
   - The LMS-substrings are sorted using a modified version of SAIS (bucket-sorting in SA).
     - Step 2': The last symbol of each LMS-substring is added into the end its bucket.
     - The order of the LMS-substrings of size 1 induce the order of L- and S- types in Steps 3' and 4'.



Within each c-bucket, L-type are smaller than S-type suffixes.

SAIS [NZC11] and SACA-K [Non13]:

1. Step 1: Sorting the LMS-suffixes:
   - The LMS-substrings are sorted using a modified version of SAIS (bucket-sorting in SA).
     - Step 2': The last symbol of each LMS-substring is added into the end its bucket.
     - The order of the LMS-substrings of size 1 induce the order of L- and S- types in Steps 3' and 4'.

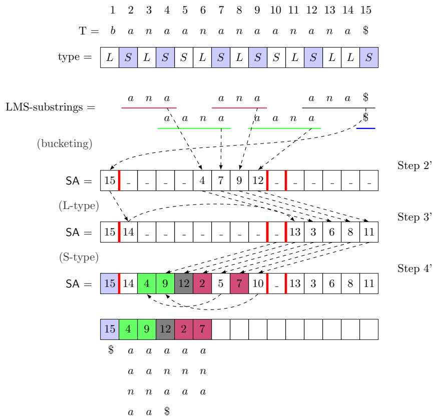


Within each c-bucket, L-type are smaller than S-type suffixes.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

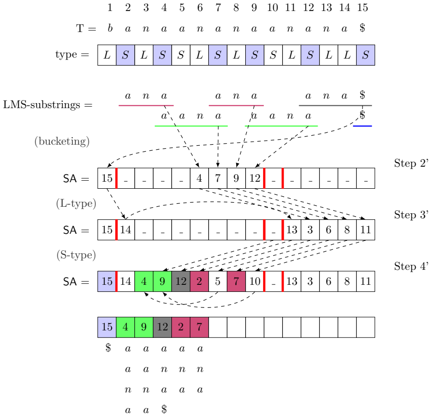
1. Step 1: Sorting the LMS-suffixes:
   - The LMS-substrings are sorted using a modified version of SAIS (bucket-sorting in SA).
     - Step 2': The last symbol of each LMS-substring is added into the end its bucket.
     - The order of the LMS-substrings of size 1 induce the order of L- and S- types in Steps 3' and 4'.



Within each c-bucket, L-type are smaller than S-type suffixes.

SAIS [NZC11] and SACA-K [Non13]:

1. Step 1: Sorting the LMS-suffixes:
   - The LMS-substrings are sorted using a modified version of SAIS (bucket-sorting in SA).
     - Step 2': The last symbol of each LMS-substring is added into the end its bucket.
     - The order of the LMS-substrings of size 1 induce the order of L- and S- types in Steps 3' and 4'.



Within each c-bucket, L-type are smaller than S-type suffixes.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

1. Step 1: Sorting the LMS-suffixes:
   - Each LMS-substring r_i receives a name v_i according to its rank*.
   - A new (shorter) string $T^1 = v_1 v_2 \ldots v_{n^1}$ is created.
     - If all symbols (ranks) of T^1 are unique $\Rightarrow$ all LMS-suffixes are sorted.
     - Otherwise, the problem is solved recursively*. [link]
     - Sorting all suffixes of T^1 is equivalent to sorting all LMS-suffixes of T.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
type =	L	S	L	S	S	L	S	L	S	S	L	S	L	L	S

LMS-substrings =	Names
$a\ n\ a$	4
$a\ a\ n\ a$	2
$a\ n\ a$	4
$a\ a\ n\ a$	2
$a\ n\ a$ \$	3
\$	1

	1	2	3	4	5	6
T^1 =	4	2	4	2	3	1
type =	L	S	L	S	S	S

* Naming of SAIS and SACA-K differs.

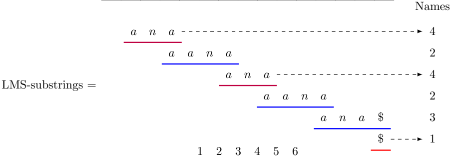
* The alphabet of T^1 is integer, and T^1 is also terminated by a unique smallest *sentinel*.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

1. Step 1: Sorting the LMS-suffixes:
    - Each LMS-substring r_i receives a name v_i according to its rank*.
    - A new (shorter) string $T^1 = v_1 v_2 \dots v_{n^1}$ is created.
        - If all symbols (ranks) of T^1 are unique $\Rightarrow$ all LMS-suffixes are sorted.
        - Otherwise, the problem is solved recursively*. [link]
        - Sorting all suffixes of T^1 is equivalent to sorting all LMS-suffixes of T.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
type =	L	S	L	S	S	L	S	L	S	S	L	S	L	L	S



	1	2	3	4	5	6
T^1 =	4	2	4	2	3	1
type =	L	S	L	S	S	S

* Naming of SAIS and SACA-K differs.

* The alphabet of T^1 is integer, and T^1 is also terminated by a unique smallest *sentinel*.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

1. Step 1: Sorting the LMS-suffixes:
   - Each LMS-substring r_i receives a name v_i according to its rank*.
   - A new (shorter) string $T^1 = v_1 v_2 \ldots v_{n^1}$ is created.
     - If all symbols (ranks) of T^1 are unique $\Rightarrow$ all LMS-suffixes are sorted.
     - Otherwise, the problem is solved recursively*. [link]
     - Sorting all suffixes of T^1 is equivalent to sorting all LMS-suffixes of T.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
type =	L	S	L	S	S	L	S	L	S	S	L	S	L	L	S

LMS-substrings =	Names
$a\ n\ a$	4
$a\ a\ n\ a$	2
$a\ n\ a$	4
$a\ a\ n\ a$	2
$a\ n\ a$ \$	3
\$	1

	1	2	3	4	5	6
T^1 =	4	2	4	2	3	1
type =	L	S	L	S	S	S

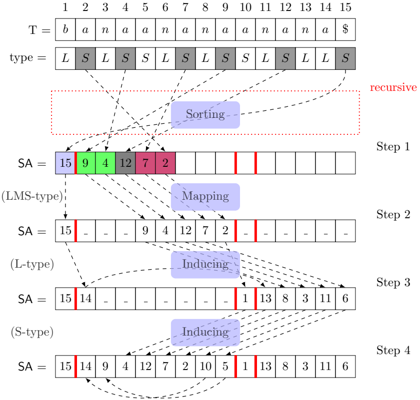
* Naming of SAIS and SACA-K differs.

* The alphabet of T^1 is integer, and T^1 is also terminated by a unique smallest *sentinel*.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- Step 2: LMS-suffixes are mapped to the end of its buckets*.
- Step 3: Scan SA, $1, 2, \ldots, n$, if $T[\mathsf{SA}[i]-1, n]$ is L-type, induce $\mathsf{SA}[i]-1$ into the head of its bucket.
- Step 4: Scan SA, $n, n-1, \ldots, 1$, if $T[\mathsf{SA}[i]-1, n]$ is S-type, induce $\mathsf{SA}[i]-1$ into the end of its bucket.

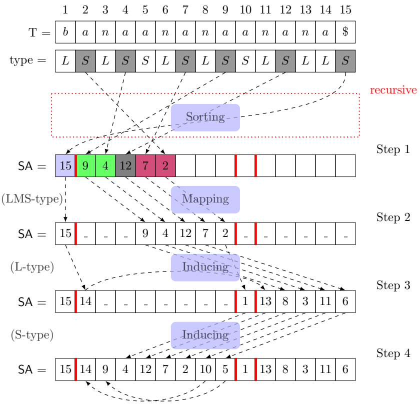


---

*Within each c-bucket, L-type are smaller than S-type suffixes.

SAIS [NZC11] and SACA-K [Non13]:

- Step 2: LMS-suffixes are mapped to the end of its buckets*.
- Step 3: Scan SA, $1, 2, \ldots, n$, if $T[\mathrm{SA}[i]-1, n]$ is L-type, induce $\mathrm{SA}[i]-1$ into the head of its bucket.
- Step 4: Scan SA, $n, n-1, \ldots, 1$, if $T[\mathrm{SA}[i]-1, n]$ is S-type, induce $\mathrm{SA}[i]-1$ into the end of its bucket.

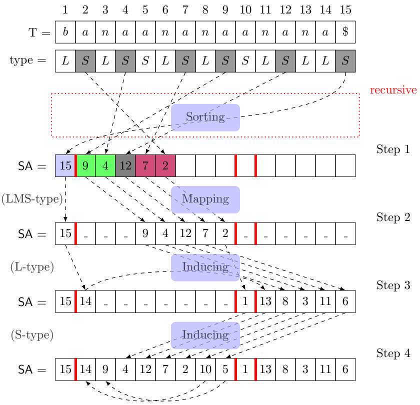


*Within each c-bucket, L-type are smaller than S-type suffixes.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- **Step 2:** LMS-suffixes are mapped to the end of its buckets*.
- **Step 3:** Scan SA, $1, 2, \ldots, n$, if $T[\mathrm{SA}[i]-1, n]$ is L-type, induce $\mathrm{SA}[i]-1$ into the head of its bucket.
- **Step 4:** Scan SA, $n, n-1, \ldots, 1$, if $T[\mathrm{SA}[i]-1, n]$ is S-type, induce $\mathrm{SA}[i]-1$ into the end of its bucket.



*Within each c-bucket, L-type are smaller than S-type suffixes.

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- Time complexity: O(n).
  - Step 1, the reduced problem is at most $n/2$.
  - Steps 2, 3 and 4 may be performed in linear time (scan-based).
- Workspace:
  - The space used by SA suffices for storing both SA^1 and T^1 along all recursive calls.
  - SAIS: $0.5n \log n + n$ bits⋆.
  - SACA-K: $\sigma \log n$ bits⋆.

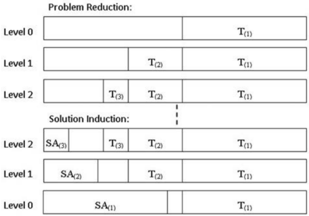
⋆It is dominated by the bucket array and type array

⋆Only for the bucket array at the top recursion level. The type of each $T[i, n]$ on-the-fly in constant time

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- Time complexity: O(n).
  - Step 1, the reduced problem is at most $n/2$.
  - Steps 2, 3 and 4 may be performed in linear time (scan-based).
- Workspace:
  - The space used by SA suffices for storing both SA^1 and T^1 along all recursive calls.
  - SAIS: $0.5n \log n + n$ bits*.
  - SACA-K: $\sigma \log n$ bits*.



* It is dominated by the bucket array and type array

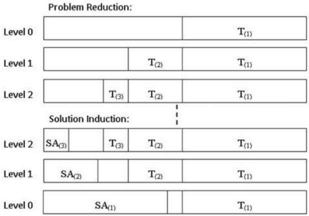
* Only for the bucket array at the top recursion level. The type of each $T[i, n]$ on-the-fly in constant time

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- Time complexity: O(n).
  - Step 1, the reduced problem is at most $n/2$.
  - Steps 2, 3 and 4 may be performed in linear time (scan-based).

- Workspace:
  - The space used by SA suffices for storing both SA^1 and T^1 along all recursive calls.
  - SAIS: $0.5n \log n + n$ bits*.
  - SACA-K: $\sigma \log n$ bits*.



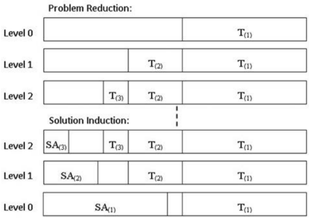
* It is dominated by the bucket array and type array

* Only for the bucket array at the top recursion level. The type of each $T[i, n]$ on-the-fly in constant time

Related Work

SAIS [NZC11] and SACA-K [Non13]:

- Time complexity: O(n).
  - Step 1, the reduced problem is at most $n/2$.
  - Steps 2, 3 and 4 may be performed in linear time (scan-based).
- Workspace:
  - The space used by SA suffices for storing both SA^1 and T^1 along all recursive calls.
  - SAIS: $0.5n \log n + n$ bits*.
  - SACA-K: $\sigma \log n$ bits*.



---

* It is dominated by the bucket array and type array

* Only for the bucket array at the top recursion level. The type of each $T[i, n]$ on-the-fly in constant time

Related Work

SAIS+LCP [Fis11]:

- Key observation: the lcp values of induced suffixes can also be induced.

Modifications:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The lcp-values are "scaled-up" from names in T^1 to name lengths in the LMS-substrings. [link]
- Step 2: the lcp-values are mapped in the LCP-array.
- Steps 3 and 4:
  - Whenever $T[x, n]$ and $T[y, n]$ are induced and placed at adjacent positions $k-1$ and k, LCP$[k]$ can be induced from:

$$\text{lcp}(T[x, n], T[y, n]) = \text{lcp}(T[x+1, n], T[y+1, n]) + 1 = \text{rmq}(i, j) + 1$$

.

The lcp between the last L-suffix and the first S-suffix of each c-bucket by direct comparison (only equal symbols).

Related Work

SAIS+LCP [Fis11]:

- Key observation: the lcp values of induced suffixes can also be induced.

Modifications:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The lcp-values are "scaled-up" from names in T^1 to name lengths in the LMS-substrings. [link]
- Step 2: the lcp-values are mapped in the LCP-array.
- Steps 3 and 4:
  - Whenever $T[x, n]$ and $T[y, n]$ are induced and placed at adjacent positions $k-1$ and k, LCP$[k]$ can be induced from:

$$\mathrm{lcp}(T[x, n], T[y, n]) = \mathrm{lcp}(T[x+1, n], T[y+1, n]) + 1 = \mathrm{rmq}(i, j) + 1$$

.

The lcp between the last L-suffix and the first S-suffix of each c-bucket by direct comparison (only equal symbols).

Related Work

SAIS+LCP [Fis11]:

- Key observation: the lcp values of induced suffixes can also be induced.

Modifications:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The lcp-values are "scaled-up" from names in T^1 to name lengths in the LMS-substrings. [link]
- Step 2: the lcp-values are mapped in the LCP-array.
- Steps 3 and 4:
  - Whenever $T[x, n]$ and $T[y, n]$ are induced and placed at adjacent positions $k-1$ and k, LCP[k] can be induced from:

$$\mathrm{lcp}(T[x,n], T[y,n]) = \mathrm{lcp}(T[x+1,n], T[y+1,n]) + 1 = \mathrm{rmq}(i,j) + 1$$

.

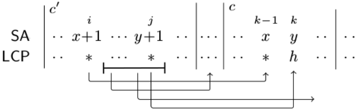


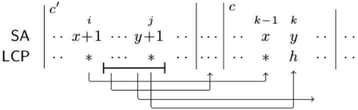
Figure: Inducing the LCP array [Fis11]

The lcp between the last L-suffix and the first S-suffix of each c-bucket by direct comparison (only equal symbols).

Related Work

SAIS+LCP [Fis11]:

- RMQ-alternatives:
  1. Scan the whole interval LCP$[i, j]$ for each rmq $\rightarrow O(n^2)$ time.
  2. Keep an array $C[1, \sigma]$ up-to-date, $C[c]$ stores the minimum LCP between the current suffix and the last induced suffix starting with $c \rightarrow$ in $O(n\sigma)$ time*.
  3. An improved alternative is to use a *semi-dynamic* rmq data structure [FH07] to solve the rmqs in $O(1)$ time using $2n + o(n)$ bits $\rightarrow$ in $O(n)$ time.

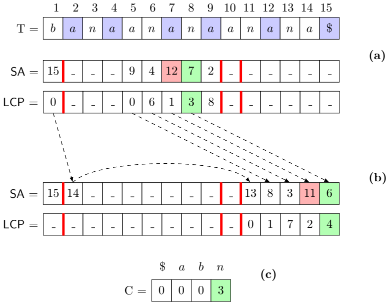


* To keep C up-to-date, at each step an $O(\sigma)$ time procedure is performed to update all values of C.

Related Work

SAIS+LCP [Fis11]:

- RMQ-alternatives:
  1. Scan the whole interval LCP$[i, j]$ for each rmq $\rightarrow O(n^2)$ time.
  2. Keep an array $C[1, \sigma]$ up-to-date, $C[c]$ stores the minimum LCP between the current suffix and the last induced suffix starting with $c \rightarrow$ in $O(n\sigma)$ time*.
  3. An improved alternative is to use a *semi-dynamic* rmq data structure [FH07] to solve the rmqs in $O(1)$ time using $2n + o(n)$ bits $\rightarrow$ in $O(n)$ time.



* To keep C up-to-date, at each step an $O(\sigma)$ time procedure is performed to update all values of C.

Related Work

SAIS+LCP [Fis11]:

- RMQ-alternatives:
  1. Scan the whole interval LCP$[i, j]$ for each rmq $\rightarrow O(n^2)$ time.
  2. Keep an array $C[1, \sigma]$ up-to-date, $C[c]$ stores the minimum LCP between the current suffix and the last induced suffix starting with $c \rightarrow$ in $O(n\sigma)$ time*.
  3. An improved alternative is to use a *semi-dynamic* rmq data structure [FH07] to solve the rmqs in $O(1)$ time using $2n + o(n)$ bits $\rightarrow$ in $O(n)$ time.

* To keep C up-to-date, at each step an $O(\sigma)$ time procedure is performed to update all values of C.

Related Work

SAIS+LCP [Fis11]:

- Time complexity: depends on the rmq alternative*.
  - $O(n)$ time, with the improved alternative.
- Workspace:
  - $1.5n \log n + n + 2n + o(n)$ bits*.
  - The space used by LCP suffices for storing LCP[1] along all recursive calls.

---

* Fischer has implemented an $O(n\sigma)$-time alternative.

* Two arrays of size $n/2$ for rank and size. And the rmq data structure.

Related Work

SAIS+LCP [Fis11]:

- Time complexity: depends on the rmq alternative⋆.
  - $O(n)$ time, with the improved alternative.
- Workspace:
  - $1.5n \log n + n + 2n + o(n)$ bits⋆.
  - The space used by LCP suffices for storing LCP[1] along all recursive calls.

⋆ Fischer has implemented an $O(n\sigma)$-time alternative.

⋆ Two arrays of size $n/2$ for rank and size. And the rmq data structure.

Our contribution

SACA-K+LCP:

- We show how to construct the LCP array during SACA-K maintaining its theoretical bounds.
  - Our algorithm can be viewed as an adaptation of Fischer's algorithm to SACA-K.

Problems:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The procedure that scales up the lcp-values uses additional $O(n \log n)$ bits. [link]
- Step 3 and 4: inducing L- and S-suffixes.
  - The $O(1)$ time rmq-alternative uses additional $2n + o(n)$ bits bits.
  - The $O(\sigma)$ time uses additional $\sigma \log n$ bits.
    - During the recursive calls, the alphabet size σ^1 of T^1 is integer ($\sigma^1 = O(n/2)$).
    - The size of the auxiliary array $C[1, \sigma^1]$ is no longer constant.

Our contribution

SACA-K+LCP:

- We show how to construct the LCP array during SACA-K maintaining its theoretical bounds.
  - Our algorithm can be viewed as an adaptation of Fischer's algorithm to SACA-K.

Problems:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The procedure that scales up the lcp-values uses additional $O(n \log n)$ bits. [link]
- Step 3 and 4: inducing L- and S-suffixes.
  - The $O(1)$ time rmq-alternative uses additional $2n + o(n)$ bits bits.
  - The $O(\sigma)$ time uses additional $\sigma \log n$ bits.
    - During the recursive calls, the alphabet size σ^1 of T^1 is integer ($\sigma^1 = O(n/2)$).
    - The size of the auxiliary array $C[1, \sigma^1]$ is no longer constant.

Our contribution

SACA-K+LCP:

- We show how to construct the LCP array during SACA-K maintaining its theoretical bounds.
  - Our algorithm can be viewed as an adaptation of Fischer's algorithm to SACA-K.

Problems:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The procedure that scales up the lcp-values uses additional $O(n \log n)$ bits. [link]
- Step 3 and 4: inducing L- and S-suffixes.
  - The $O(1)$ time rmq-alternative uses additional $2n + o(n)$ bits bits.
  - The $O(\sigma)$ time uses additional $\sigma \log n$ bits.
    - During the recursive calls, the alphabet size σ^1 of T^1 is integer ($\sigma^1 = O(n/2)$).
    - The size of the auxiliary array $C[1, \sigma^1]$ is no longer constant.

Our contribution

SACA-K+LCP:

- We show how to construct the LCP array during SACA-K maintaining its theoretical bounds.
  - Our algorithm can be viewed as an adaptation of Fischer's algorithm to SACA-K.

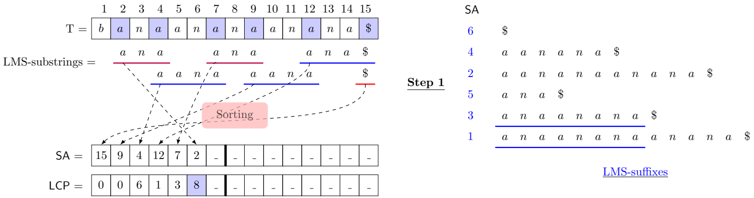
Problems:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The procedure that scales up the lcp-values uses additional $O(n \log n)$ bits. [link]
- Step 3 and 4: inducing L- and S-suffixes.
  - The $O(1)$ time rmq-alternative uses additional $2n + o(n)$ bits bits.
  - The $O(\sigma)$ time uses additional $\sigma \log n$ bits.
    - During the recursive calls, the alphabet size σ^1 of T^1 is integer ($\sigma^1 = O(n/2)$).
    - The size of the auxiliary array $C[1, \sigma^1]$ is no longer constant.

Our contribution

SACA-K+LCP:

- Step 1:
  - We compute LCP of the LMS-suffixes immediately at the top recursion level, just after sorting all LMS-suffixes in Step 1.
- A sparse variant of the Φ-algorithm [KMP09] can be used.
  - linear time.
  - Additional $O(n \log n)$ bits to store $\Phi[1, n/2]$.
- The additional array can be stored in $LCP[1, n]^{\star}$, being subsequently overwritten. [link]

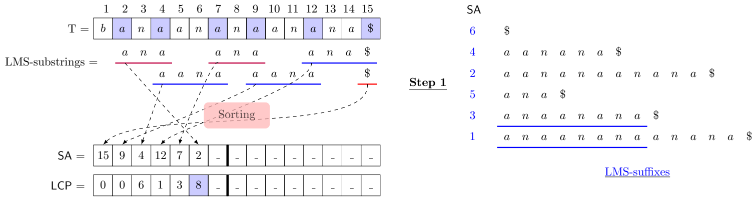


*Fischer observed that in the experimental section [Fis11].

Our contribution

SACA-K+LCP:

- Step 1:
    - We compute LCP of the LMS-suffixes immediately at the top recursion level, just after sorting all LMS-suffixes in Step 1.
    - A sparse variant of the Φ-algorithm [KMP09] can be used.
        - linear time.
        - Additional $O(n \log n)$ bits to store $\Phi[1, n/2]$.
    - The additional array can be stored in $LCP[1, n]^{\star}$, being subsequently overwritten. [link]

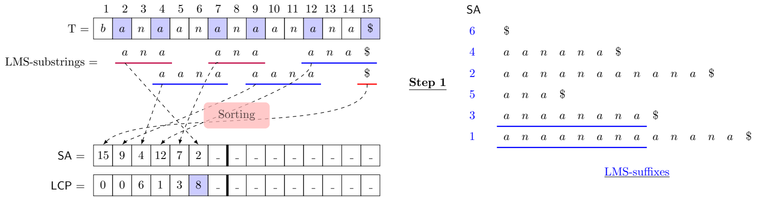


⋆ Fischer observed that in the experimental section [Fis11].

Our contribution

SACA-K+LCP:

- Step 1:
  - We compute LCP of the LMS-suffixes immediately at the top recursion level, just after sorting all LMS-suffixes in Step 1.
  - A sparse variant of the Φ-algorithm [KMP09] can be used.
    - linear time.
    - Additional $O(n \log n)$ bits to store $\Phi[1, n/2]$.
  - The additional array can be stored in $LCP[1, n]^{\star}$, being subsequently overwritten. [link]

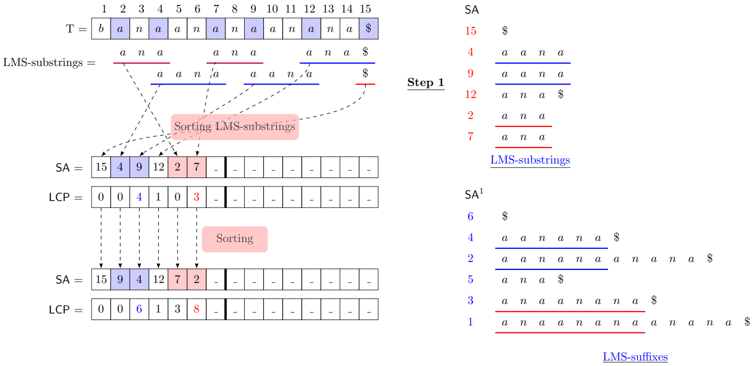


⋆ Fischer observed that in the experimental section [Fis11].

Our contribution

SACA-K+LCP:

- Step 1:
  - We augmented this idea by pre-computing LCP of the LMS-suffixes during *naming*⋆.
  - Property: Any two consecutive LMS-suffixes share an lcp larger or equal to the lcp between the LMS-substrings that were in those positions prior to the LMS-suffix sorting.

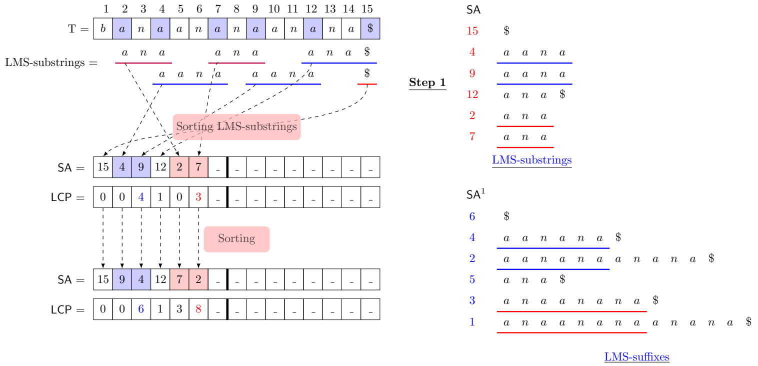


⋆Where each consecutive LMS-substrings is compared to assign its name.

Our contribution

SACA-K+LCP:

- Step 1:
  - We augmented this idea by pre-computing LCP of the LMS-suffixes during *naming*⋆.
  - Property: Any two consecutive LMS-suffixes share an lcp larger or equal to the lcp between the LMS-substrings that were in those positions prior to the LMS-suffix sorting.

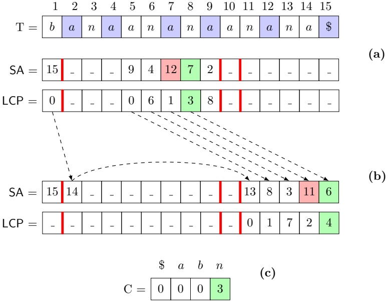


⋆Where each consecutive LMS-substrings is compared to assign its name.

Our contribution

SACA-K+LCP:

- Steps 3 and 4:
  - rmq: $O(n\sigma)$-time alternative:
    - We compute LCP only at the top recursion level $\rightarrow$ $O(n\sigma)$ time.
  - Workspace: Additional $\sigma \log n$ bits to store $C[1, \sigma]$.



Our contribution

SACA-K+LCP:

- Time complexity:
  - $O(n\sigma)$ time.
- Workspace:
  - $O(\sigma \log n)$ bits.

Optimal for string from constant alphabets $\sigma = O(1)$.

Experiments:

- We implemented our algorithm in ANSI C.
- Source code: `https://github.com/felipelouza/sacak-lcp`.
- Experiments with Pizza & Chili datasets.
- We compared: SACA-K+LCP, SAIS+LCP and SACA-K followed by Φ-algorithm.
- Results: [link].
  - SAIS+LCP was the fastest algorithm in all experiments.
  - SACA-K+LCP was the only algorithm that kept the space usage constant: 10KB.

Outline

Inducing enhanced suffix arrays for string collections

String collections:

- Let $\mathcal{T} = T_1, T_2, \ldots, T_d$ be a collection of d strings.
- Sorting all suffixes of $\mathcal{T}$ may be performed by sorting the concatenation of all strings.

Two common approaches to create the concatenated string T^{cat} of total length $(\Sigma_{i=1}^{d} n_i) + 1 = N^{\star}$.

1. $T^{cat} = T_1[1, n_1 - 1] \cdot \$_1 \cdot T_2[1, n_2 - 1] \cdot \$_2 \cdots T_d[1, n_d - 1] \cdot \$_d \cdot \#$
2. $T^{cat} = T_1[1, n_1 - 1] \cdot \$ \ \cdot T_2[1, n_2 - 1] \cdot \$ \ \cdots T_d[1, n_d - 1] \cdot \$ \ \cdot \#$

Drawbacks:

1. Increases the alphabet size of T^{cat} by the number of strings $\sigma^{cat} = O(d)$.
   - Deteriorate the theoretical bounds of many algorithms → SACA-K's workspace would increase to $O(d \log N)$ bits.
2. Do not guarantee the relative order between equal suffixes of T_i and T_j, such that \$ from T_i is smaller than \$ from T_j if and only if $i < j$.
   - lcp-values may exceed the length of the strings.

* $\# < \$ < \$_1 < \$_2 < \ldots < \$_d$ are symbols not in Σ and are smaller than any symbol in the alphabet.

Inducing enhanced suffix arrays for string collections

String collections:

- Let $\mathcal{T} = T_1, T_2, \ldots, T_d$ be a collection of d strings.
- Sorting all suffixes of $\mathcal{T}$ may be performed by sorting the concatenation of all strings.

Two common approaches to create the concatenated string T^{cat} of total length $(\Sigma_{i=1}^{d} n_i) + 1 = N^{\star}$.

1. $T^{cat} = T_1[1, n_1 - 1] \cdot \$_1 \cdot T_2[1, n_2 - 1] \cdot \$_2 \cdots T_d[1, n_d - 1] \cdot \$_d \cdot \#$
2. $T^{cat} = T_1[1, n_1 - 1] \cdot \$ \ \cdot T_2[1, n_2 - 1] \cdot \$ \ \cdots T_d[1, n_d - 1] \cdot \$ \ \cdot \#$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1. T^{cat} =	b	a	n	a	n	a	$\$_1$	a	n	a	b	a	$\$_2$	a	n	a	n	$\$_3$	#
2. T^{cat} =	b	a	n	a	n	a	\$	a	n	a	b	a	\$	a	n	a	n	\$	#

Drawbacks:

1. Increases the alphabet size of T^{cat} by the number of strings $\sigma^{cat} = O(d)$.
   - Deteriorate the theoretical bounds of many algorithms $\rightarrow$ SACA-K's workspace would increase to $O(d \log N)$ bits.
2. Do not guarantee the relative order between equal suffixes of T_i and T_j, such that \$ from T_i is smaller than \$ from T_j if and only if $i < j$.
   - lcp-values may exceed the length of the strings.

$^{\star}\# < \$ < \$_1 < \$_2 < \ldots < \$_d$ are symbols not in Σ and are smaller than any symbol in the alphabet.

Inducing enhanced suffix arrays for string collections

String collections:

- Let $\mathcal{T} = T_1, T_2, \ldots, T_d$ be a collection of d strings.
- Sorting all suffixes of $\mathcal{T}$ may be performed by sorting the concatenation of all strings.

Two common approaches to create the concatenated string T^{cat} of total length $(\Sigma_{i=1}^{d} n_i) + 1 = N^{\star}$.

1. $T^{cat} = T_1[1, n_1 - 1] \cdot \$_1 \cdot T_2[1, n_2 - 1] \cdot \$_2 \cdots T_d[1, n_d - 1] \cdot \$_d \cdot \#$
2. $T^{cat} = T_1[1, n_1 - 1] \cdot \$ \cdot T_2[1, n_2 - 1] \cdot \$ \cdots T_d[1, n_d - 1] \cdot \$ \cdot \#$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1. $T^{cat} =$	b	a	n	a	n	a	$\$_1$	a	n	a	b	a	$\$_2$	a	n	a	n	$\$_3$	$\#$
2. $T^{cat} =$	b	a	n	a	n	a	$\$$	a	n	a	b	a	$\$$	a	n	a	n	$\$$	$\#$

Drawbacks:

1. Increases the alphabet size of T^{cat} by the number of strings $\sigma^{cat} = O(d)$.
   - Deteriorate the theoretical bounds of many algorithms → SACA-K's workspace would increase to $O(d \log N)$ bits.
2. Do not guarantee the relative order between equal suffixes of T_i and T_j, such that \$ from T_i is smaller than \$ from T_j if and only if $i < j$.
   - lcp-values may exceed the length of the strings.

---

* $\# < \$ < \$_1 < \$_2 < \ldots < \$_d$ are symbols not in Σ and are smaller than any symbol in the alphabet.

Inducing enhanced suffix arrays for string collections

String collections:

- Let $\mathcal{T} = T_1, T_2, \ldots, T_d$ be a collection of d strings.
- Sorting all suffixes of $\mathcal{T}$ may be performed by sorting the concatenation of all strings.

Two common approaches to create the concatenated string T^{cat} of total length $(\Sigma_{i=1}^{d} n_i) + 1 = N^{\star}$.

1. $T^{cat} = T_1[1, n_1 - 1] \cdot \$_1 \cdot T_2[1, n_2 - 1] \cdot \$_2 \cdots T_d[1, n_d - 1] \cdot \$_d \cdot \#$
2. $T^{cat} = T_1[1, n_1 - 1] \cdot \$ \cdot T_2[1, n_2 - 1] \cdot \$ \cdots T_d[1, n_d - 1] \cdot \$ \cdot \#$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1. $T^{cat} =$	b	a	n	a	n	a	$\$_1$	a	n	a	b	a	$\$_2$	a	n	a	n	$\$_3$	$\#$
2. $T^{cat} =$	b	a	n	a	n	a	$\$$	a	n	a	b	a	$\$$	a	n	a	n	$\$$	$\#$

Drawbacks:

1. Increases the alphabet size of T^{cat} by the number of strings $\sigma^{cat} = O(d)$.
   - Deteriorate the theoretical bounds of many algorithms → SACA-K's workspace would increase to $O(d \log N)$ bits.
2. Do not guarantee the relative order between equal suffixes of T_i and T_j, such that \$ from T_i is smaller than \$ from T_j if and only if $i < j$.
   - lcp-values may exceed the length of the strings.

* $\# < \$ < \$_1 < \$_2 < \ldots < \$_d$ are symbols not in Σ and are smaller than any symbol in the alphabet.

Inducing enhanced suffix arrays for string collections

Our contribution:

- We show how to modify SAIS [NZC11] and SACA-K [Non13] to sort T^{cat} created by alternative 2 (same separators).
  - Maintaining their theoretical bounds.
  - Respecting the order among all suffixes, $T_i < T_j$ if and only if $i < j^\star$.
  - Improving their practical performance.

- Moreover, we show how to compute during suffix sorting:
  - LCP array (adapting ideas by [Fis11] and [LGT17b]). [link]
  - Document array (DA). [link]

*In other words, we obtain the same results one would get using distinct separators.

Our contribution

gSAIS and gSACA-K

- Key observation:
  1. In T^{cat} every suffix starting with \$ will be a LMS-type suffix, except for the last one.
  2. These $d-1$ LMS-type suffixes will generate a LMS-substring that will be sorted unnecessarily*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$T^{cat} =$	b	a	n	a	n	a	\$	a	n	a	b	a	\$	a	n	a	n	\$	#
type =	L	S	L	S	L	L	S	S	L	S	L	L	S	S	L	S	L	L	S

LMS-substrings = $a\ n\ a$ \$ $a\ n\ a$ \$ $a\ n\ a$ #
$a\ n\ a$ \$ $a\ b\ a$ \$ $a\ n$ \$ #

- To guarantee that a \$ from string T_i will be smaller than a \$ from T_j if and only if $i < j$:
  1. We can use their positions $T^{cat}[i'] = \$ < T^{cat}[j'] = \$$ if and only if $i' < j'$.

*if two suffixes are equal up to their separators \$ then their symbols should not be compared any further

Our contribution

gSAIS and gSACA-K

- Key observation:
  1. In T^{cat} every suffix starting with \$ will be a LMS-type suffix, except for the last one.
  2. These $d-1$ LMS-type suffixes will generate a LMS-substring that will be sorted unnecessarily*.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T^{cat} =	b	a	n	a	n	a	\$	a	n	a	b	a	\$	a	n	a	n	\$	#
type =	L	S	L	S	L	L	S	S	L	S	L	L	S	S	L	S	L	L	S

LMS-substrings = $a\ n\ a$, $a\ n\ a\ \$$, $\$\ a\ n\ a$, $a\ b\ a\ \$$, $\$\ a\ n\ a$, $a\ n\ \$\ \#$, $\#$

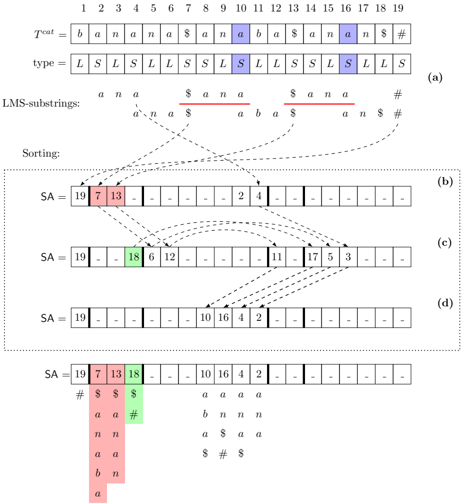
- To guarantee that a \$ from string T_i will be smaller than a \$ from T_j if and only if $i < j$:
  1. We can use their positions $T^{cat}[i'] = \$ < T^{cat}[j'] = \$$ if and only if $i' < j'$.

* if two suffixes are equal up to their separators \$ then their symbols should not be compared any further

Our contribution

gSAIS and gSACA-K

- Step 1: Sorting LMS-substrings.

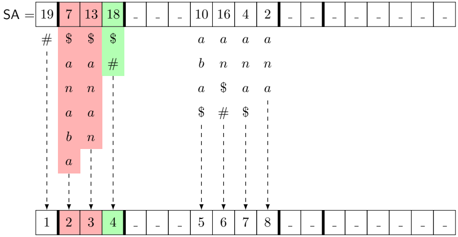


We do not insert the last symbol of the LMS-substrings starting with $ in the bucket-sorting.

Our contribution

gSAIS and gSACA-K

- Naming:
  - Each LMS-substring starting with \$ will receive a different name according to its position in T^{cat}.
  - The reduced string T^1 is created as usual.
- Note:
  - The modifications are necessary only at the top recursion level.
  - T^1 will be exactly the same when applied to T^{cat} using alternative 1.



Our contribution

gSAIS and gSACA-K

- Time complexity:
    - The algorithms remain linear on the length of input, that is $O(N)$.
- Workspace:
    - The algorithms use the same amount of memory of their original versions.
    - In particular, gSACA-K uses $\sigma \log N$ bits, which is optimal for constant alphabets.
- Theoretical improvement:
    - Comparing gSACA-K and SACA-K applied to sort T^{cat} created by alternative 1.
    - The workspace of SACA-K is $(\sigma + d) \log N$ bits.

Experiments

gSAIS, gSACA-K

- All the algorithms were implemented in ANSI C.
- Source code: `https://github.com/felipelouza/gsa-is`.
- Data collections of size up to 16 GB:

collection	σ	$N/2^{30}$	d	N/d	$max(\|T_i\|)$	$mean\_lcp$	max_lcp
pages	205	3.74	1,000	4,019,585	362,724,758	29,595.13	2,912,604
revision	203	0.39	20,433	20,527	2,000,452	31,612.79	1,995,055
influenza	15	0.56	394,217	1,516	2,867	533.83	2,379
wikipedia	208	8.32	3,903,703	2,288	224,488	27.12	61,055
reads	4	2.87	32,621,862	94	101	43.35	101
proteins	25	15.77	50,825,784	333	36,805	91.03	32,882

- We compared gSAIS and gSACA-K with SAIS and SACA-K applied to sort T^{cat}:
  1. SAIS* and SACA-K*: alternative 1 (integer string).
  2. SAIS and SACA-K: alternative 2.

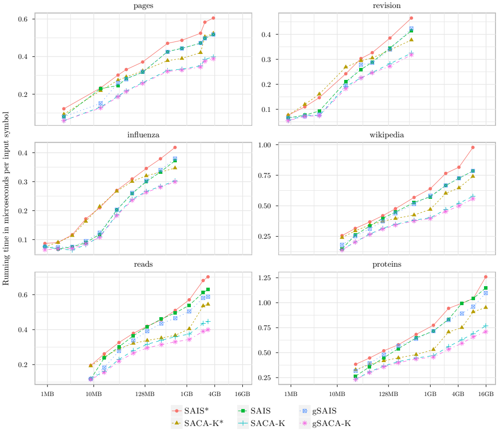
- We also compared gSAIS+LCP, gSACA-K+LCP, gSAIS+DA and gSACA-K+DA. [link]

Columns 7 and 8 show the average and maximum lcp-values computed on the single strings, which provide an approximation for suffix sorting difficulty.

Experiments (SA)

Time (μsec/symbol):

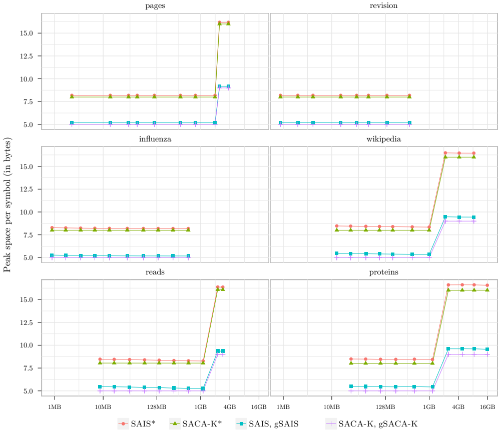
- gSACA-K and SACA-K were the fastest algorithms.
  - gSACA-K was faster when d is large (`proteins` and `reads`), it avoids sorting $d-1$ LMS-substrings.
- Comparing with SACA-K*, the time spent by gSACA-K was 24.3% smaller than on the average.



Experiments (SA)

Peakspace (bytes/symbol):

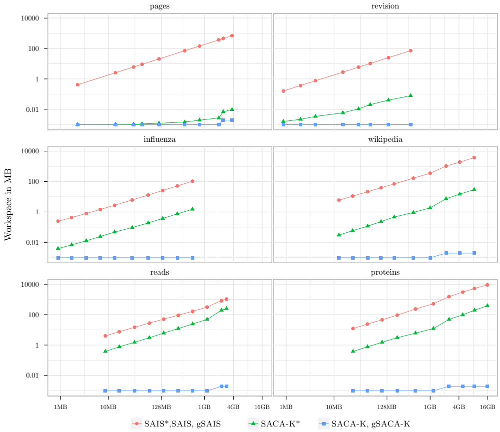
- gSACA-K and SACA-K were the smallest.
  - $5N + O(1)$ bytes when $N < 2^{31}$ and $9N + O(1)$ bytes otherwise.
- Note that when $N > 2^{31}$, the peak memory of all algorithms increases, since they use 64-bits integers.



Experiments (SA)

Workspace (MB):

- SACA-K and gSACA-K: 1 KB when $N < 2^{31}$ and 2 KB otherwise.
  - Optimal for strings from constant alphabets.
- SAIS*, SAIS and gSAIS are $O(N \log N)$ bits, whereas SACA-K* is $O(d \log N)$ bits.



Outline

Contributions

List of publications:

1. **Felipe A. Louza**; Travis Gagie; Guilherme P. Telles. Burrows-Wheeler transform and LCP array construction in constant space. *Journal of Discrete Algorithms*. v. 42: 14-22, 2017.
2. **Felipe A. Louza**; Simon Gog; Guilherme P. Telles. Optimal suffix sorting and LCP array construction for constant alphabets. *Information Processing Letters*, v. 118, 30-34, 2017.
3. **Felipe A. Louza**; Simon Gog; Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, v. 678: 22-39, 2017.
4. **Felipe A. Louza**; Simon Gog; Guilherme P. Telles. Induced suffix sorting for string collections. *In: DCC*, 2016. 43-52.
5. **Felipe A. Louza**; Guilherme P. Telles. Computing the BWT and the LCP array in constant space. *In: IWOCA*, 2015. 312-320.

Other publications:

1. **Felipe A. Louza**; Simon Gog; Leandro Zanotto, Guido Araujo, Guilherme P. Telles. Parallel computation for the all-pairs suffix-prefix problem. *In: SPIRE*, 2016. 122-132.
2. William H. A. Tustumi; Simon Gog; Guilherme P. Telles; **Felipe A. Louza**. An improved algorithm for the all-pairs suffix-prefix problem. *Journal of Discrete Algorithms*, v. 37, 34-43, 2016.

Contributions

List of publications:

1. **Felipe A. Louza**; Travis Gagie; Guilherme P. Telles. Burrows-Wheeler transform and LCP array construction in constant space. *Journal of Discrete Algorithms*. v. 42: 14-22, 2017.
2. **Felipe A. Louza**; Simon Gog; Guilherme P. Telles. Optimal suffix sorting and LCP array construction for constant alphabets. *Information Processing Letters*, v. 118, 30-34, 2017.
3. **Felipe A. Louza**; Simon Gog; Guilherme P. Telles. Inducing enhanced suffix arrays for string collections. *Theoretical Computer Science*, v. 678: 22-39, 2017.
4. **Felipe A. Louza**; Simon Gog; Guilherme P. Telles. Induced suffix sorting for string collections. *In: DCC*, 2016. 43-52.
5. **Felipe A. Louza**; Guilherme P. Telles. Computing the BWT and the LCP array in constant space. *In: IWOCA*, 2015. 312-320.

Other publications:

1. **Felipe A. Louza**; Simon Gog; Leandro Zanotto, Guido Araujo, Guilherme P. Telles. Parallel computation for the all-pairs suffix-prefix problem. *In: SPIRE*, 2016. 122-132.
2. William H. A. Tustumi; Simon Gog; Guilherme P. Telles; **Felipe A. Louza**. An improved algorithm for the all-pairs suffix-prefix problem. *Journal of Discrete Algorithms*, v. 37, 34-43, 2016.

Thank you!

Questions?

Outline

Uwe Baier.
Linear-time suffix sorting - a new approach for suffix array construction.
In *Proc. CPM*, pages 23:1–23:12, 2016.

Timo Bingmann, Johannes Fischer, and Vitaly Osipov.
Inducing suffix and LCP arrays in external memory.
ACM J. Experiment. Algorithmics, 21(2):2.3:1–2.3:27, 2016.

M. Burrows and D.J. Wheeler.
A block-sorting lossless data compression algorithm.
Technical report, Digital SRC Research Report, 1994.

Maxime Crochemore, Roberto Grossi, Juha Kärkkäinen, and Gad M. Landau.
Computing the Burrows-Wheeler transform in place and in small space.
J. Discret. Algorithms, 32:44–52, 2015.

Jasbir Dhaliwal, Simon J. Puglisi, and A. Turpin.
Trends in suffix sorting: A survey of low memory algorithms.
In *Proc. ACSC*, pages 91–98, 2012.

P Elias.
Universal codeword sets and representations of the integers.
IEEE Trans. on Information Theory, 21(2):194–203, March 1975.

Johannes Fischer and Volker Heun.
A new succinct representation of rmq-information and improvements in the enhanced suffix array.
In *Proc. ESCAPE*, pages 459–470, 2007.

Johannes Fischer.
Inducing the LCP-Array.
In *Proc. WADS*, pages 374–385, 2011.

Keisuke Goto and Hideo Bannai.
Space efficient linear time Lempel-Ziv factorization for small alphabets.
In *Proc. DCC*, pages 163–172, 2014.

Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider.
New indices for text: Pat trees and pat arrays.
In *Information Retrieval*, pages 66–82. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

Simon Gog and Enno Ohlebusch.
Fast and lightweight LCP-array construction algorithms.
In *Proc. ALENEX*, pages 25–34, 2011.

Juha Kärkkäinen.
Suffix array construction.
In *Encyclopedia of Algorithms*, pages 2141–2144. Springer, 2016.

Juha Kärkkäinen, Dominik Kempa, Simon J. Puglisi, and Bella Zhukova.
Engineering external memory induced suffix sorting.
In *Proc. ALENEX*, pages 98–108, 2017.

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.

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

Felipe Alves Louza, Travis Gagie, and Guilherme Pimentel Telles.
Burrows-wheeler transform and LCP array construction in constant space.
J. Discret. Algorithms, 42:14–22, 2017.

Felipe Alves Louza, Simon Gog, and Guilherme Pimentel Telles.
Optimal suffix sorting and LCP array construction for constant alphabets.
Inf. Process. Lett., 118:30–34, 2017.

Wei Jun Liu, Ge Nong, Wai Hong Chan, and Yi Wu.
Induced sorting suffixes in external memory with better design and less space.
In *Proc. SPIRE*, pages 83–94, 2015.

Giovanni Manzini.
Two space saving tricks for linear time LCP array computation.
In *Proc. SWAT*, pages 372–383, 2004.

Udi Manber and Eugene W Myers.
Suffix arrays: A new method for on-line string searches.
SIAM J. Comput., 22(5):935–948, 1993.

Ge Nong, Wai Hong Chan, Sheng Qing Hu, and Yi Wu.
Induced sorting suffixes in external memory.
ACM Trans. Inf. Syst., 33(3):12:1–12:15, 2015.

Ge Nong.
Practical linear-time O(1)-workspace suffix sorting for constant alphabets.
ACM Trans. Inform. Syst., 31(3):1–15, 2013.

Ge Nong, Sen Zhang, and Wai Hong Chan.
Two efficient algorithms for linear time suffix array construction.
IEEE Trans. Comput., 60(10):1471–1484, 2011.

Daisuke Okanohara and Kunihiko Sadakane.
A linear-time Burrows-Wheeler transform using induced sorting.
In *Proc. SPIRE*, pages 90–101, 2009.

Simon J. Puglisi, William F. Smyth, and Andrew H. Turpin.
A taxonomy of suffix array construction algorithms.
ACM Comp. Surv., 39(2):1–31, 2007.

Extra slides

Introduction

Suffix array construction algorithms (SACAs):

- Several SACAs have been proposed in the past 20 years [PST07, DPT12].

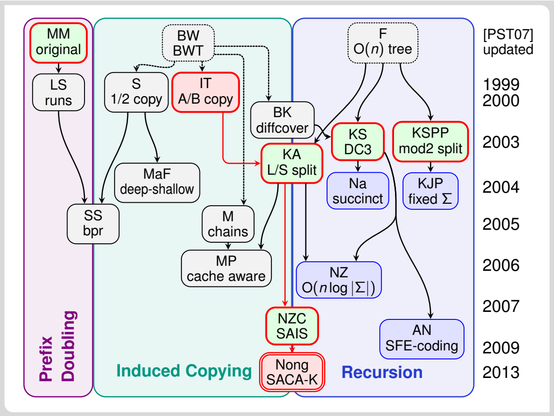


Figure by T. Bingmann.

⋆ MM (1990s), linear time (2003), SAIS (2009) and SACA-K (2013).

Chapter 2

BWT and LCP construction in constant space

BWT in-place:

- Incremental step:
  - Given BWT(T_{s+1}), stored in $T[s+1, n]$:
    1. Find position p of \$.
    2. Find the local rank r of $T[s, n]$.
    3. Replace \$ by $T[s]$.
    4. Insert new suffix and preceding character \$ into $T[r]$.

	s	BWT	suffixes
	1	b	banana\$
$s \rightarrow$	2	a	anana\$
	3	a	\$
	4	n	a\$
	5	n	ana\$
	6	a	na\$
$p \rightarrow$	7	\$	nana\$

BWT(T_{s+1})

	s	BWT	suffixes
	1	b	banana\$
	2	a	\$
	3	n	a\$
	4	n	ana\$
$r \rightarrow$	5		. . .
	6	a	na\$
$p \rightarrow$	7	\$	nana\$

Find local rank r

	s	BWT	suffixes
	1	b	banana\$
	2	a	\$
	3	n	a\$
	4	n	ana\$
	5		a. . .
	6	a	na\$
$p \rightarrow$	7	a	nana\$

Replace \$

	s	BWT	suffixes
	1	b	banana\$
	2	a	\$
	3	n	a\$
	4	n	ana\$
$r \rightarrow$	5	\$	anana\$
	6	a	na\$
	7	a	nana\$

BWT(T_s)

- Step 2, finding r by LF-mapping:
  - $T[s]$ will be placed in $T[p] \Rightarrow k$-th $\alpha \in \Sigma$ in BWT(T_s) corresponds to k-th α in F.
    - number of symbols smaller than $T[s]$ in $T[s+1, n]$.
    - number of symbols equal to $T[s]$ in $T[s+1, r]$.

BWT and LCP construction in constant space

BWT in-place:

- Step 2 (find local position r):
  - $T[s]$ will be placed in $T[p]$.
  - LF-mapping: The i-th symbol $\alpha \in \Sigma$ in L corresponds to the i-th symbol α in F.
  - To determine the position, we need to count:
    - number of symbols smaller than $T[s]$ in $T[s+1,n]$.
    - number of symbols equal to $T[s]$ in $T[s+1,r]$.

	s	BWT	suffixes
	1	b	banana$
$s \rightarrow$	2	a	anana$
	3	a	$
	4	n	a$
	5	n	ana$
	6	a	na$
$p \rightarrow$	7	$	nana$

BWT(T_{s+1})

	s	BWT	suffixes
	1	b	banana$
$s \rightarrow$	2	a	anana$
	3	a	$
	4	n	a$
	5	n	ana$
	6	a	na$
$p \rightarrow$	7	a	nana$

BWT(T_{s+1})

	s	BWT	suffixes
	1	b	banana$
$s \rightarrow$	2	a	anana$
	3	a	$
	4	n	a$
	5	n	ana$
	6	a	na$
$p \rightarrow$	7	$	nana$

$\alpha < T[s]$

	s	BWT	suffixes
	1	b	banana$
$s \rightarrow$	2	a	anana$
	3	a	$
	4	n	a$
$r \rightarrow$	5	n	ana$
	6	a	na$
$p \rightarrow$	7	a	nana$

$\alpha = T[s]$

Chapter 3

Optimal suffix sorting and LCP construction

SAIS and SACA-K:

- Key observations:
  - The order of the LMS-suffixes are enough to induce the order of all suffixes of T
  - The LMS-suffixes can be sorted recursively.

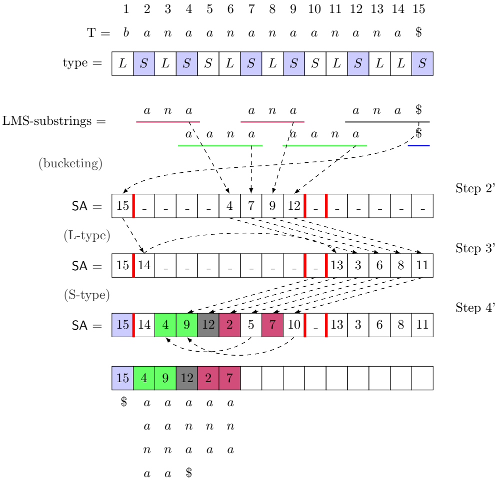
Induced sorting (IS) algorithm:

1. Sort the LMS-type suffixes and store in an auxiliary array SA^1.
2. Scan SA^1 from right to left, and insert each LMS-suffix of T into the tail of its c-bucket.
3. Scan SA from left to right, and for each $T[\mathsf{SA}[i], n]$ if $T[\mathsf{SA}[i]-1, n]$ is L-type then insert $\mathsf{SA}[i]-1$ into the head of its bucket.
4. Scan SA from right to left, and for each $T[\mathsf{SA}[i], n]$ if $T[\mathsf{SA}[i]-1, n]$ is S-type then insert $\mathsf{SA}[i]-1$ into the tail of its bucket.

Optimal suffix sorting and LCP construction

SAIS and SACA-K:

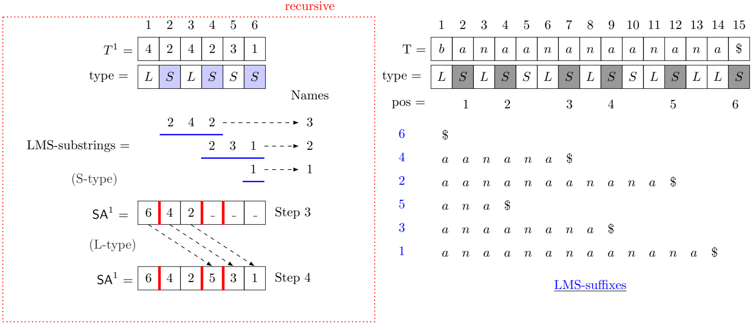
- Sorting LMS-substrings.



Related Work

SAIS and SACA-K:

- Sorting T^1 recursively:
  - The algorithm is recursively applied to sort the suffixes of T^1.
  - The alphabet of T^1 is integer, and T^1 is also terminated by a unique smallest *sentinel*.
- Sorting all suffixes of T^1 is equivalent to sorting all LMS-suffixes of T.



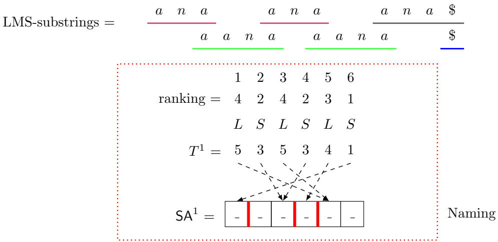
Nong *et al.* observed that the space used by SA suffices for storing both SA^1 and T^1 along all recursive calls.

Nong, 2013:

Removing the bucket array from recursive calls*.

Naming:

- The names are indexes to positions of SA, such that:
  - If T_i is L-type then $T[i] = v_i$ points to the head of its bucket.
  - If T_j is S-type then $T[j] = v_j$ points to the end of its bucket.
- The relative order between all suffixes of T^1 is maintained $\star$.



* In fact, if this problem has not been solved, the workspace of SACA-K would remain $O(n \log n)$ bits.
$\star$ Recall that the alphabet of T^1 is integer, suitable for such scheme.

Nong, 2013:

Nong presented SACA-K, the first linear time sorting algorithm also fast in practice using constant space memory.

- SACA-K's framework is similar to that of SA-IS*, its major improvement is the reduced memory usage.

Key observations:

- The *type* array is no longer necessary.
  - Step 1: *type* is used in to find (and compare) the LMS-substrings.
  - Step 3 and 4: *type* is used to determine the type of $T_{\mathrm{SA}[i]-1}$.

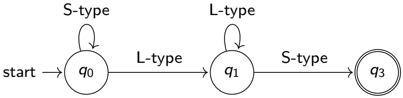


Figure: LMS-substring type pattern recognition, from $T[i]$, $T[i+1]$ to $T[j]$

- The bucket array is only necessary at level 0, where the alphabet of T is constant.

* The naming procedure of SACA-K is different from that in SA-IS.

Related Work

SAIS+LCP:

- Key observation: the lcp values of induced suffixes can also be induced.

Modifications:

- Step 1: the lcp-values of the LMS-suffixes are computed recursively.
  - The lcp-values are "scaled-up" from names in T^1 to name lengths in the LMS-substrings.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T =	b	a	n	a	a	n	a	n	a	a	n	a	n	a	\$
type =	L	S	L	S	S	L	S	L	S	S	L	S	L	L	S
		*		*			*		*			*			*
rank =		4		2			4		2			3			1
size =		3		4			3		4			4			1

$rank$	
1	\$
2	a a n a
3	a n a \$
4	a n a

LMS-substrings

	1	2	3	4	5	6
T^1 =	4	2	4	2	3	1
SA^1 =	6	4	2	5	3	1
	1	2	2	3	4	4
		3	4	1	2	2
		1	2		3	4
			3		1	2
			1			3
						1

	1	2	3	4	5	6	
LCP_{rank} =	0	0	1	3	-	-	
LCP^1 =	0	0	1	0	0	2	
	-	-	-	-	-	7	**(a)** size(4)+size(2)
	-	-	-	-	-	3	**(b)** $\mathsf{lcp}_{rank}(3,4)$
LCP^1 =	0	0	6	1	3	8	**(c)** (a-1)+(b-1)

SA^1	
6	\$
4	a a n a n a \$
2	a a n a n a a n a n a \$
5	a n a \$
3	a n a a n a n a \$
1	a n a a n a n a a n a n a \$

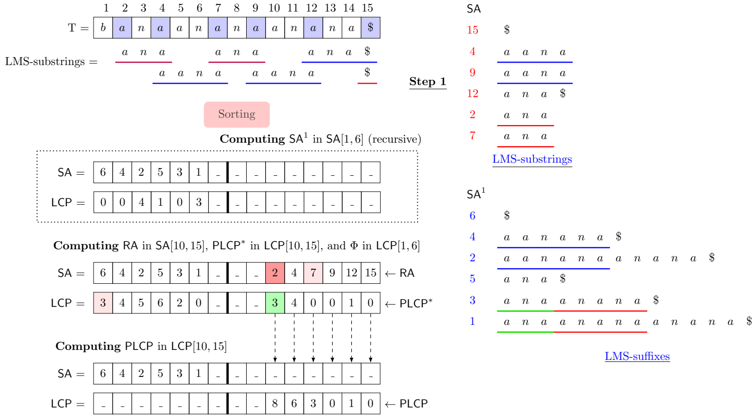
LMS-suffixes

LCP_{rank}, rank, size: additional data structures.

Our contribution

SACA-K+LCP:

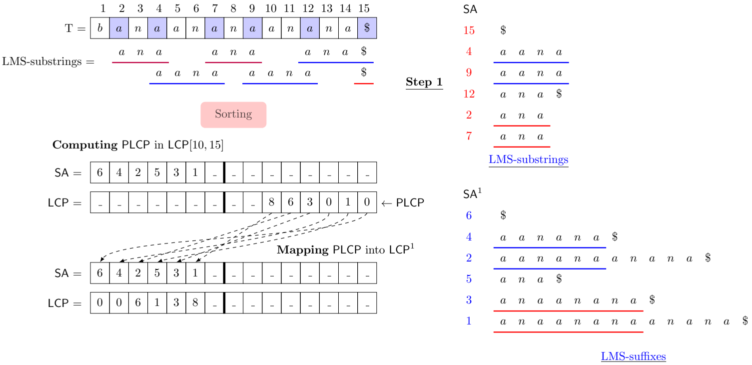
- Step 1:
  - Φ-algorithm first computes the permuted LCP (PLCP) array and then derives LCP.
    - PLCP* is the PLCP pre-computed by lcp-values of LMS-substrings.
    - RA stores the distance between the suffixes being compared and their respective successors (in text order).



Our contribution

SACA-K+LCP:

- Step 2:
  - Mapping:
    - LCP[i] = PLCP[SA[i]].
    - At the end, LCP^1 is computed from PLCP, overwriting positions LCP[$1, n/2$].



Experiments

SACA-K+LCP:

- SAIS+LCP was the fastest algorithm in all experiments.
- SACA-K+LCP was the only algorithm that kept the space usage constant: 10KB.
  - 1KB of SACA-K's workspace added by 9KB used by data structures to solve the rmqs.
- Overhead:
  - SACA-K+LCP vs. SACA-K and Φ-algorithm: similar speed using much less space.

			speed [μs/byte]					workspace [KB]		
dataset	σ	$n/2^{10}$	SACA-K+LCP	SAIS+LCP	SACA-K and Φ	SACA-K	Φ	SACA-K+LCP	SAIS+LCP	SACA-K and Φ
sources	230	205,924	0.26	**0.17**	0.24	0.21	0.03	**10**	16	823,698
xml	97	289,195	0.28	**0.18**	0.26	0.23	0.03	**10**	14	1,156,781
dna	16	394,461	0.38	**0.27**	0.36	0.31	0.05	**10**	13	1,577,843
english.1G	239	1,071,976	0.43	**0.31**	0.42	0.35	0.07	**10**	15	4,287,904
proteins	27	1,156,300	0.41	**0.30**	0.40	0.34	0.06	**10**	13	4,625,201
einstein-de	117	90,584	0.34	**0.18**	0.33	0.30	0.03	**10**	14	362,338
kernel	160	251,916	0.28	**0.16**	0.26	0.23	0.03	**10**	14	1,007,662
fib41	2	261,635	0.34	**0.18**	0.30	0.27	0.03	**10**	13	1,046,540
cere	5	450,475	0.34	**0.20**	0.31	0.28	0.03	**10**	13	1,801,901

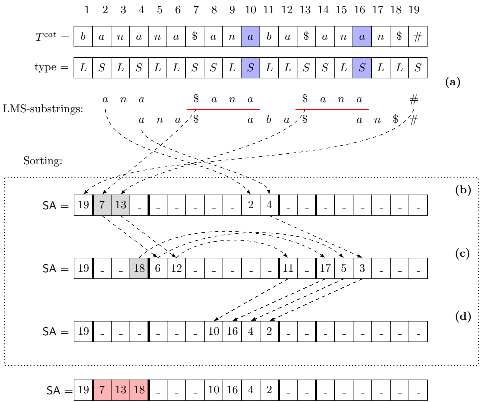
The workspace is the peak space subtracted of the space used by T, SA and LCP ($9n$ bytes). SACA-K's workspace is always 1 KB. Φ's workspace is equal to $4n$ bytes and dominates SACA-K and Φ.

Chapter 4

Our contribution

gSAIS and gSACA-K

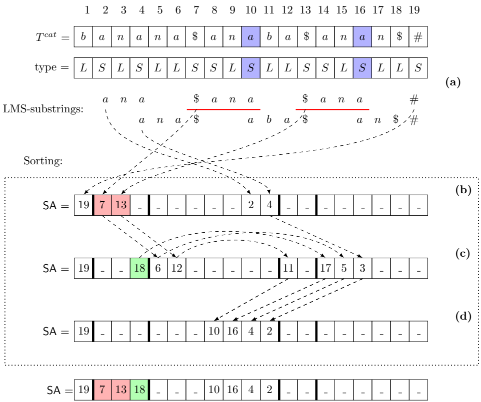
- Step 1 (during the LMS-substring sorting):
  - We do not insert any LMS-suffix $T^{cat}[j, N]$ in its bucket if the next LMS-suffix $T^{cat}[i, N]$ to the left starts with a \$
  - After sorting, we scan $T^{cat}[1, N]$ again inserting the LMS-suffixes directly into the \$-bucket.



Our contribution

gSAIS and gSACA-K

- Step 2' (during the LMS-substring sorting):
  - When the LMS-suffixes are inserted at its bucket, we reserve the last position of the \$-bucket to $T^{cat}[N-1, N]$.
  - Then, we insert the suffix $T^{cat}[N-1, N]$ directly at the tail of its bucket in the end of Step 2.



Our contribution

gSAIS+LCP and gSACA-K+LCP

- We slightly modified the ideas by [Fis11] and [LGT17a].
  1. Our sparse variant of the Φ-algorithm may treat each separator \$ as a distinct symbol*.
  2. We compute directly the lcp-values in the \$-bucket that will be equal to 0.
- Correctness:
  - We do not induce L- or S-type suffixes starting with \$ in Steps 3 and 4.
- Analysis:
  - Our versions, gSAIS+LCP and gSACA-K+LCP, run in $O(N\sigma)$ time.
  - The workspace of gSACA-K+LCP is $4\sigma \log N$ bits.

(a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
SA =	19	7	13	18	_	_	_	10	16	4	2	_	_	_	_	_	_	_	_
LCP =	0	0	0	0	_	_	_	1	1	2	3	_	_	_	_	_	_	_	_

(b)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
SA =	19	7	13	18	6	12	_	10	16	4	2	11	_	17	5	9	_	_	_
LCP =	0	0	0	0	0	1	_	_	1	2	3	0	_	0	1	2	_	_	_

(c)

	#	\$	a	b	n
C =	0	0	1	1	1

* This requires a straightforward modification in the algorithm.

Inducing enhanced suffix arrays for string collections

[link]

Document array (DA):

- The suffix array of $T^{cat}[1, N]$ is commonly accompanied by the document array (DA).
  - DA[i] stores the index of the string which suffix $T^{cat}[\mathsf{SA}[i], N]$ came from.

2. $T^{cat} =$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
b	a	n	a	n	a	\$	a	n	a	b	a	\$	a	n	a	n	\$	#

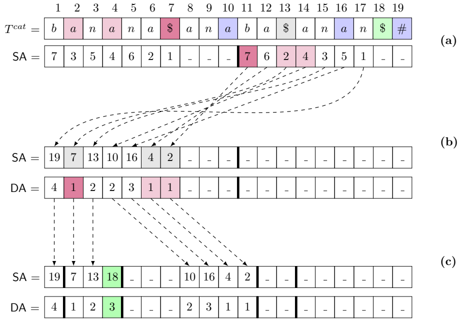
i	SA	LCP	DA	suffixes
1	19	0	4	#
2	18	0	3	\$
3	7	0	1	\$
4	13	0	2	\$
5	6	0	1	a\$
6	12	1	2	a\$
7	10	1	2	aba\$
8	16	1	3	an\$
9	4	2	1	ana\$
10	8	3	2	anaba\$
11	14	3	3	anan\$
12	2	4	1	anana\$
13	11	0	2	ba\$
14	1	2	1	banana\$
15	17	0	3	n\$
16	5	1	1	na\$
17	9	2	2	naba\$
18	15	2	3	nan\$
19	3	3	1	nana\$

DA[1] $= d + 1$ as the suffix $T^{cat}[N, N] = \#$ is always in SA[1].

Our contribution

gSAIS+DA and gSACA-K+DA

- Step 2 (when the LMS-suffixes are mapped back)
  - When scanning $T^{cat}[1, N]$ and ISA[1]:
    - (a) Starting from $i = N, N-1, \ldots, 1$ and $k = d + 1$. If $T^{cat}[i] = \$$ then k is decremented by one.
    - (b) If $T^{cat}[i, N]$ then DA[ISA[1][j]] receives k.
    - (c) At the end, the DA-values are bucket sorted in DA.
- At the end, when $T^{cat}[N-1, N]$ is inserted directly at the tail of its bucket, we also set DA as d.



Our contribution

gSAIS+DA and gSACA-K+DA

- Steps 3 and 4:
  - Whenever a suffix $T^{cat}[i-1, N]$ is induced in position SA[k], DA[k] is induced by the value in DA[ISA[i]].
- Correctness:
  - We do not induce L- or S-type suffixes starting with $ in Steps 3 and 4.
- Analysis:
  - Our versions, gSAIS+DA and gSACA-K+DA, run in $O(N)$ time.
  - The workspace are the same of their original versions.

Experiments

SA and LCP: [link]

- Time: gSACA-K+LCP and gSACA-K combined with Φ were the fastest algorithms.
- Peakspace:
  - gSACA-K+LCP: $9N + O(1)$ bytes when $N < 2^{31}$ and $17N + O(1)$ bytes otherwise.
  - gSACA-K combined with Φ: $13N$ bytes when $N < 2^{31}$ and $25N$ bytes otherwise.
- Workspace:
  - gSACA-K+LCP: 10 KB when $N < 2^{31}$ and 20 KB otherwise.
  - gSACA-K combined with Φ: $O(N \log N)$ bits.

SA and DA: [link]

- Time: gSACA-K+DA and gSACA-K combined with BIT were the fastest algorithms.
- Peakspace:
  - gSACA-K+DA: $9N + O(1)$ bytes when $N < 2^{31}$ and $17N + O(1)$ bytes otherwise.
  - gSACA-K combined with BIT: $9N$ bytes + $O(N)$ bits required by BIT to solve the rank queries.
- Workspace:
  - gSACA-K+DA: 1 KB when $N < 2^{31}$ and 2 KB otherwise.
  - gSACA-K combined with BIT: $N + o(N)$ bits*.

*N bits to store the bitvector $B[1, N]$ + $o(N)$ bits for the rank data structure.

Experiments

SA and LCP: [link]

- Time: gSACA-K+LCP and gSACA-K combined with Φ were the fastest algorithms.
- Peakspace:
  - gSACA-K+LCP: $9N + O(1)$ bytes when $N < 2^{31}$ and $17N + O(1)$ bytes otherwise.
  - gSACA-K combined with Φ: $13N$ bytes when $N < 2^{31}$ and $25N$ bytes otherwise.
- Workspace:
  - gSACA-K+LCP: 10 KB when $N < 2^{31}$ and 20 KB otherwise.
  - gSACA-K combined with Φ: $O(N \log N)$ bits.

SA and DA: [link]

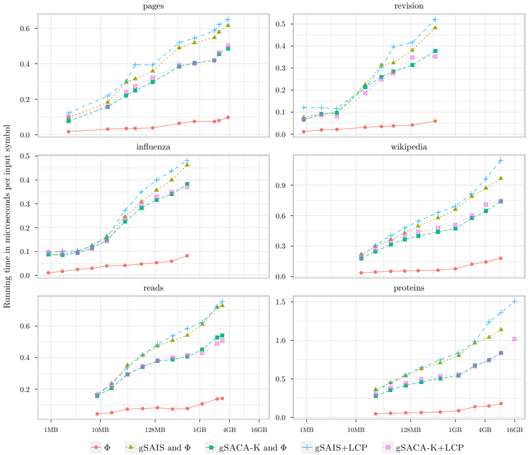
- Time: gSACA-K+DA and gSACA-K combined with BIT were the fastest algorithms.
- Peakspace:
  - gSACA-K+DA: $9N + O(1)$ bytes when $N < 2^{31}$ and $17N + O(1)$ bytes otherwise.
  - gSACA-K combined with BIT: $9N$ bytes + $O(N)$ bits required by BIT to solve the rank queries.
- Workspace:
  - gSACA-K+DA: 1 KB when $N < 2^{31}$ and 2 KB otherwise.
  - gSACA-K combined with BIT: $N + o(N)$ bits*.

*N bits to store the bitvector $B[1, N]$ + $o(N)$ bits for the rank data structure.

Experiments (SA and LCP)

Time:

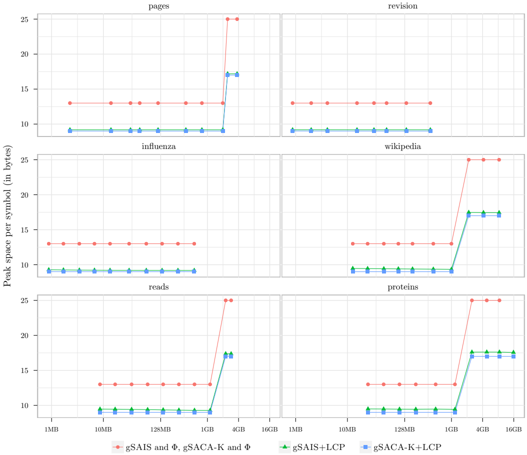
- gSACA-K+LCP and gSACA-K combined with Φ were the fastest algorithms.
- Φ was terminated by the system for `proteins` with 15.77 GB, as it required more than 386 GB of RAM.



Experiments (SA and LCP)

Peakspace:

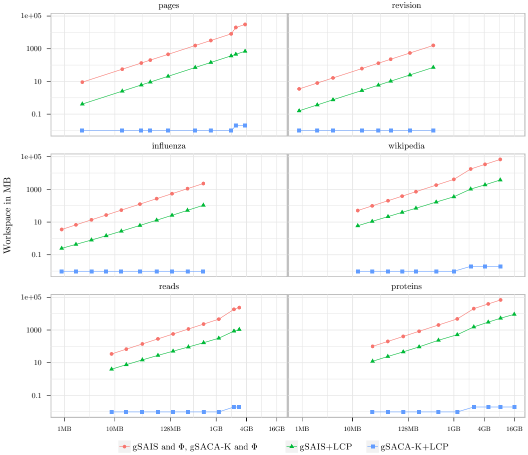
- gSACA-K+LCP: $9N + O(1)$ bytes when $N < 2^{31}$ and $17N + O(1)$ bytes otherwise.
- gSACA-K combined with Φ: $13N$ bytes when $N < 2^{31}$ and $25N$ bytes otherwise.



Experiments (SA and LCP)

Workspace:

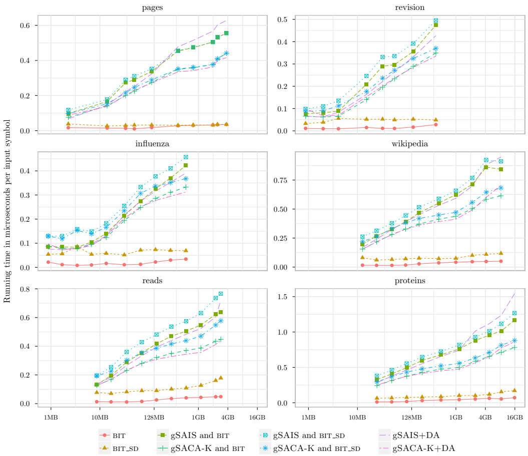
- gSACA-K+LCP: 10 KB when $N < 2^{31}$ and 20 KB otherwise.
- gSAIS+LCP is $O(N)$, whereas gSAIS and gSACA-K combined with Φ are dominated by the workspace of Φ, which uses an additional integer array of size N.



Experiments (SA and DA)

Time:

- gSACA-K+DA and gSACA-K combined with BIT were the fastest algorithms.
- The time added by computing the document array in gSACA-K+DA was 8.3% on the average*.

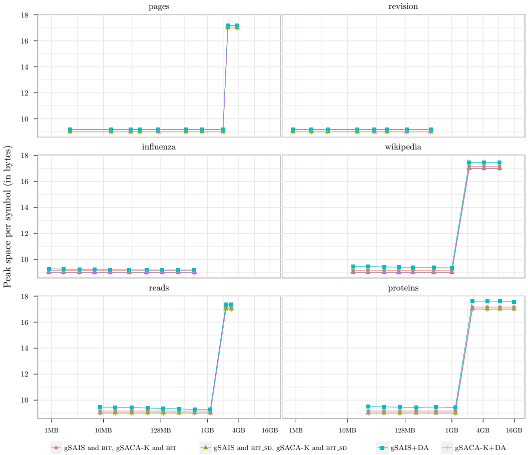


* Easier problem: this time is smaller than the overhead added by the LCP array construction in gSACA-K+LCP.

Experiments (SA and DA)

Peakspace:

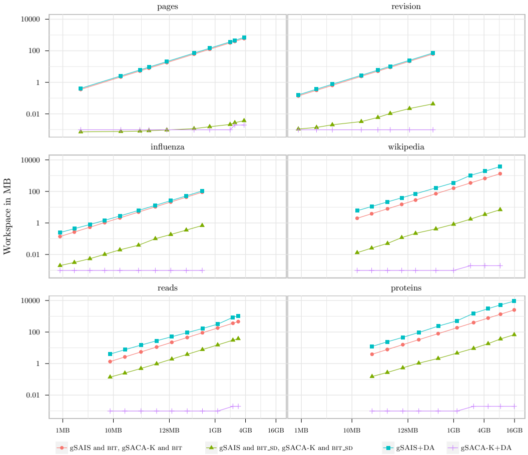
- gSACA-K+DA: $9N + O(1)$ bytes when $N < 2^{31}$ and $17N + O(1)$ bytes otherwise.
- gSACA-K combined with BIT: $9N$ bytes + $O(N)$ bits required by BIT to solve the rank queries.



Experiments (SA and DA)

Workspace:

- gSACA-K+DA: 1 KB when $N < 2^{31}$ and 2 KB otherwise.
- gSAIS+DA is $O(N)$, whereas the combined algorithms are dominated by BIT and BIT_SD.
- gSACA-K combined with BIT: $N + o(N)$ bits*.



*N bits to store the bitvector $B[1, N]$ + $o(N)$ bits for the rank data structure.

Chapter 5

Conclusions and future works

Our contributions:

1. BWT in-place and LCP array in $O(n^2)$-time using $O(1)$-worskpace for unbounded alphabets.
   - Future work: Investigate if it is possible to compute BWT and LCP compressed in only $2n + o(n)$ bits, in quadratic or even $o(n^2)$ time.
2. SA and LCP array in O(n)-time using $O(\sigma \log n)$ bits of worskpace, which is optimal for alphabets of constant size $\sigma = O(1)$.
   - Future work: Investigate whether the recent linear non-recursive SACA [Bai16] can also be adapted to compute the LCP array.
3. Augmented suffix sorting algorithms for string collections in optimal time and space for strings from constant alphabets.
   - Future work: modify algorithms for single strings to handle string collections (e.g. [BFO16, NCHW15, LNCW15, KKPZ17, OS09, GB14]).

Conclusions and future works

Our contributions:

1. BWT in-place and LCP array in $O(n^2)$-time using $O(1)$-worskpace for unbounded alphabets.
   - Future work: Investigate if it is possible to compute BWT and LCP compressed in only $2n + o(n)$ bits, in quadratic or even $o(n^2)$ time.
2. SA and LCP array in O(n)-time using $O(\sigma \log n)$ bits of worskpace, which is optimal for alphabets of constant size $\sigma = O(1)$.
   - Future work: Investigate whether the recent linear non-recursive SACA [Bai16] can also be adapted to compute the LCP array.
3. Augmented suffix sorting algorithms for string collections in optimal time and space for strings from constant alphabets.
   - Future work: modify algorithms for single strings to handle string collections (e.g. [BFO16, NCHW15, LNCW15, KKPZ17, OS09, GB14]).

Conclusions and future works

Our contributions:

1. BWT in-place and LCP array in $O(n^2)$-time using $O(1)$-worskpace for unbounded alphabets.
   - Future work: Investigate if it is possible to compute BWT and LCP compressed in only $2n + o(n)$ bits, in quadratic or even $o(n^2)$ time.
2. SA and LCP array in O(n)-time using $O(\sigma \log n)$ bits of worskpace, which is optimal for alphabets of constant size $\sigma = O(1)$.
   - Future work: Investigate whether the recent linear non-recursive SACA [Bai16] can also be adapted to compute the LCP array.
3. Augmented suffix sorting algorithms for string collections in optimal time and space for strings from constant alphabets.
   - Future work: modify algorithms for single strings to handle string collections (e.g. [BFO16, NCHW15, LNCW15, KKPZ17, OS09, GB14]).