External Memory Generalized Suffix and LCP Arrays Construction

Felipe A. Louza[1]   Guilherme P. Telles[2]   Cristina D. A. Ciferri[1]

[1]Institute of Mathematics and Computer Science
University of São Paulo, SP, Brazil

[2]Institute of Computing
University of Campinas, SP, Brazil

CPM 2013
Bad Herrenalb, Germany

Introduction

Suffix and LCP arrays

- Provide an efficent data structure to solve many string problems
- Several algorithms have been proposed to construct suffix and LCP arrays in external memory *e.g.* [Ferragina et al., 2012, Bingmann et al., 2013]

Indexing string sets

- To use those algorithms it would be necessary to concatenate all strings into a single one $\mathcal{T} = T_1\$_1 T_2\$_2 \ldots T_k\$_k$ with different end-markers $\$_i$
- BWT and LCP array for string sets in external memory [Bauer et al., 2012]
- Generalized suffix and LCP arrays for sets of large strings (*e.g.* genomic data)

These algorithms are aimed to index single strings

Introduction

Suffix and LCP arrays

- Provide an efficent data structure to solve many string problems
- Several algorithms have been proposed to construct suffix and LCP arrays in external memory *e.g.* [Ferragina et al., 2012, Bingmann et al., 2013]

Indexing string sets

- To use those algorithms it would be necessary to concatenate all strings into a single one $\mathcal{T} = T_1\$_1 T_2\$_2 \ldots T_k\$_k$ with different end-markers $\$_i$
- BWT and LCP array for string sets in external memory [Bauer et al., 2012]
- Generalized suffix and LCP arrays for sets of large strings (*e.g.* genomic data)

This approach limits the number indexed strings
For example, using 1 byte for each character, k is limited by $256 - |\Sigma|$

Introduction

Suffix and LCP arrays

- Provide an efficent data structure to solve many string problems
- Several algorithms have been proposed to construct suffix and LCP arrays in external memory *e.g.* [Ferragina et al., 2012, Bingmann et al., 2013]

Indexing string sets

- To use those algorithms it would be necessary to concatenate all strings into a single one $\mathcal{T} = T_1\$_1 T_2\$_2 \ldots T_k\$_k$ with different end-markers $\$_i$
- BWT and LCP array for string sets in external memory [Bauer et al., 2012]
- Generalized suffix and LCP arrays for sets of large strings (*e.g.* genomic data)

This algorithm aims at indexing large sets of small strings with fixed size
It is common in NGS data

Introduction

Suffix and LCP arrays

- Provide an efficent data structure to solve many string problems
- Several algorithms have been proposed to construct suffix and LCP arrays in external memory *e.g.* [Ferragina et al., 2012, Bingmann et al., 2013]

Indexing string sets

- To use those algorithms it would be necessary to concatenate all strings into a single one $\mathcal{T} = T_1\$_1 T_2\$_2 \ldots T_k\$_k$ with different end-markers $\$_i$
- BWT and LCP array for string sets in external memory [Bauer et al., 2012]
- Generalized suffix and LCP arrays for sets of large strings (*e.g.* genomic data)

Contribution

We introduce eGSA, an external memory algorithm to construct both generalized suffix and LCP arrays

Introduction

Let $T = T[1]T[2]\dots T[n-1]\,\$$ be a string of length n, $T[i] \in \Sigma$ and $\$ \notin \Sigma$

- $T[i,j] = T[i]\dots T[j]$, $1 \leq i \leq j \leq n$ is a substring of T
- A suffix of T is a substring $T[k,n]$
- α-suffix: a suffix starting with $\alpha \in \Sigma$

Generalized Suffix Array (GSA) and LCP Array

- Let $\mathcal{T} = \{T_1, \dots, T_k\}$ be a set of k strings of lengths $n_1, \dots, n_k$
- The GSA of $\mathcal{T}$ is an array of integers (i,j) that specifies the order of all suffixes $T_i[j, n_i]$

An additional order relation is defined for the tail suffixes $T_i[n_i, n_i] = \$$ as $T_i[n_i, n_i] < T_j[n_j, n_j]$ if $i < j$

- The LCP array of $\mathcal{T}$ is an array containing the length of the longest common prefix (lcp) of every pair of consecutive suffixes in GSA, and $LCP[1] = 0$

Introduction

Let $T = T[1]T[2]\dots T[n-1]\,\$$ be a string of length n, $T[i] \in \Sigma$ and $\$ \notin \Sigma$

- $T[i,j] = T[i]\dots T[j]$, $1 \leq i \leq j \leq n$ is a substring of T
- A suffix of T is a substring $T[k,n]$
- α-suffix: a suffix starting with $\alpha \in \Sigma$

Generalized Suffix Array (GSA) and LCP Array

- Let $\mathcal{T} = \{T_1, \dots, T_k\}$ be a set of k strings of lengths $n_1, \dots, n_k$
- The GSA of $\mathcal{T}$ is an array of integers (i,j) that specifies the order of all suffixes $T_i[j, n_i]$

An additional order relation is defined for the tail suffixes $T_i[n_i, n_i] = \$$ as $T_i[n_i, n_i] < T_j[n_j, n_j]$ if $i < j$

- The LCP array of $\mathcal{T}$ is an array containing the length of the longest common prefix (lcp) of every pair of consecutive suffixes in GSA, and $LCP[1] = 0$

Introduction

Let $T = T[1]T[2]\dots T[n-1]\,\$$ be a string of length n, $T[i] \in \Sigma$ and $\$ \notin \Sigma$

- $T[i,j] = T[i]\dots T[j]$, $1 \leq i \leq j \leq n$ is a substring of T
- A suffix of T is a substring $T[k,n]$
- α-suffix: a suffix starting with $\alpha \in \Sigma$

Generalized Suffix Array (GSA) and LCP Array

- Let $\mathcal{T} = \{T_1, \dots, T_k\}$ be a set of k strings of lengths $n_1, \dots, n_k$
- The GSA of $\mathcal{T}$ is an array of integers (i,j) that specifies the order of all suffixes $T_i[j, n_i]$

An additional order relation is defined for the tail suffixes $T_i[n_i, n_i] = \$$ as $T_i[n_i, n_i] < T_j[n_j, n_j]$ if $i < j$

- The LCP array of $\mathcal{T}$ is an array containing the length of the longest common prefix (lcp) of every pair of consecutive suffixes in GSA, and $LCP[1] = 0$

eGSA

eGSA: External Memory Generalized Suffix and LCP Arrays Construction Algorithm

- Based on the 2PMMS [Garcia-Molina et al., 1999]
- Input: a set of k strings $\mathcal{T} = \{T_1, \ldots, T_k\}$ with lengths $n_i, \ldots, n_k$
- Output: generalized suffix and lcp array for $\mathcal{T}$

In a glance, eGSA works as follows:

- **Phase 1:** For each $T_i \in \mathcal{T}$, internal memory sorting $\rightarrow SA_i$, LCP_i, and write them in external memory
- **Phase 2:** Merge the previous computed arrays obtaining GSA and LCP

Phase 1: Sorting

For each $T_i \in \mathcal{T}$:

1. Construct SA_i and LCP_i using any internal memory algorithm *e.g.* [Fischer, 2011]
2. Compute two auxiliary arrays
3. Write these arrays in external memory

In the case that there is no enough internal memory we can use an external memory algorithm *e.g.* [Bingmann et al., 2013]

Auxiliary arrays:

- $BWT_i[i] = T[SA_i[i] - 1]$ if $SA_i[i] \neq 1$ or $BWT_i[i] = \$$ otherwise
- $PRE_i[i]$ stores the the prefix of $T[SA_i[i] - 1]$

Phase 1: Sorting

For each $T_i \in \mathcal{T}$:

1. Construct SA_i and LCP_i using any internal memory algorithm *e.g.* [Fischer, 2011]
2. Compute two auxiliary arrays
3. Write these arrays in external memory

In the case that there is no enough internal memory we can use an external memory algorithm *e.g.* [Bingmann et al., 2013]

Auxiliary arrays:

- $BWT_i[i] = T[SA_i[i] - 1]$ if $SA_i[i] \neq 1$ or $BWT_i[i] = \$$ otherwise
- $PRE_i[i]$ stores the the prefix of $T[SA_i[i] - 1]$

Phase 1: Sorting

For each $T_i \in \mathcal{T}$:

1. Construct SA_i and LCP_i using any internal memory algorithm *e.g.* [Fischer, 2011]
2. Compute two auxiliary arrays
3. Write these arrays in external memory

In the case that there is no enough internal memory we can use an external memory algorithm *e.g.* [Bingmann et al., 2013]

Auxiliary arrays:

- $BWT_i[i] = T[SA_i[i] - 1]$ if $SA_i[i] \neq 1$ or $BWT_i[i] = \$$ otherwise
- $PRE_i[i]$ stores the the prefix of $T[SA_i[i] - 1]$

Phase 1: Sorting

Prefix Array of T_i, (PRE_i):

- PRE_i contains the prefixes (of length p) of the suffixes in SA_i
- $PRE_i[j] = T_i[SA_i[j], SA_i[j] + p]$ [Barsky et al., 2008]
- $PRE_i[j] = T_i[SA_i[j] + h_j, SA_i[j] + h_j + p]$, where $h_j = min(LCP_i[j], h_{j-1} + p)$ and $h_0 = 0$.

Figure: Example for $T_1 = GATAGA\$$

j	$SA_1[j]$	$LCP_1[j]$	$PRE_1[j]$	$T_1[SA[j], n_1]$
1	6	0	$$	$
2	5	0	A$	A$
3	3	1	AG	AGA$
4	1	1	AT	ATAGA$
5	4	0	GA	GA$
6	0	2	GA	GATAGA$
7	2	0	TA	TAGA$

The probability that $PRE_i[j]$ has the same information of $PRE_i[j+1]$ is large, since the suffixes are sorted in SA_i

Phase 1: Sorting

Prefix Array of T_i, (PRE_i):

- PRE_i contains the prefixes (of length p) of the suffixes in SA_i
- $PRE_i[j] = T_i[SA_i[j], SA_i[j] + p]$ [Barsky et al., 2008]
- $PRE_i[j] = T_i[SA_i[j] + h_j, SA_i[j] + h_j + p]$, where $h_j = min(LCP_i[j], h_{j-1} + p)$ and $h_0 = 0$.

Figure: Example for $T_1 = GATAGA\$$

j	$SA_1[j]$	$LCP_1[j]$	$PRE_1[j]$	$T_1[SA[j], n_1]$
1	6	0	\$\$	\$
2	5	0	A\$	A\$
3	3	1	GA	AGA\$
4	1	1	TA	ATAGA\$
5	4	0	GA	GA\$
6	0	2	TA	GATAGA\$
7	2	0	TA	TAGA\$

We can use the LCP array to compute PRE_i with non-overlapping strings [Sinha et al., 2008]

Phase 2: Merging

Merge the previous computed Suffix and LCP Arrays using:

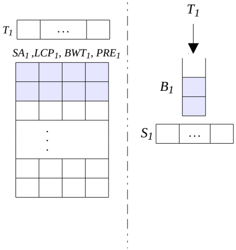
- For each $T_i \in \mathcal{T}$:
  - Partition buffer B_i, which contains blocks of $\langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - String buffer S_i, containing a substring of the suffixes of T_i
- Internal heap, each node represents heading elements (suffixes) of each B_i
- Output buffer $\rightarrow$ GSA and LCP

When the output buffer is full, it is written to external memory

Phase 2: Merging

Merge the previous computed Suffix and LCP Arrays using:

- For each $T_i \in \mathcal{T}$:
  - Partition buffer B_i, which contains blocks of $\langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - String buffer S_i, containing a substring of the suffixes of T_i
- Internal heap, each node represents heading elements (suffixes) of each B_i
- Output buffer $\rightarrow$ *GSA* and LCP

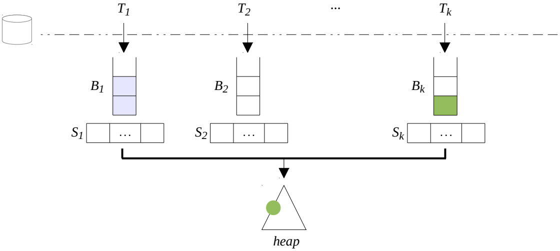


When the output buffer is full, it is written to external memory

Phase 2: Merging

Merge the previous computed Suffix and LCP Arrays using:

- For each $T_i \in \mathcal{T}$:
  - Partition buffer B_i, which contains blocks of $\langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - String buffer S_i, containing a substring of the suffixes of T_i
- Internal heap, each node represents heading elements (suffixes) of each B_i
- Output buffer $\rightarrow$ GSA and LCP

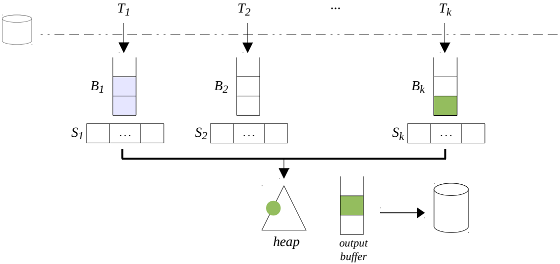


When the output buffer is full, it is written to external memory

Phase 2: Merging

Merge the previous computed Suffix and LCP Arrays using:

- For each $T_i \in \mathcal{T}$:
  - Partition buffer B_i, which contains blocks of $\langle SA_i, LCP_i, BWT_i, PRE_i \rangle$
  - String buffer S_i, containing a substring of the suffixes of T_i
- Internal heap, each node represents heading elements (suffixes) of each B_i
- Output buffer $\rightarrow$ *GSA* and LCP



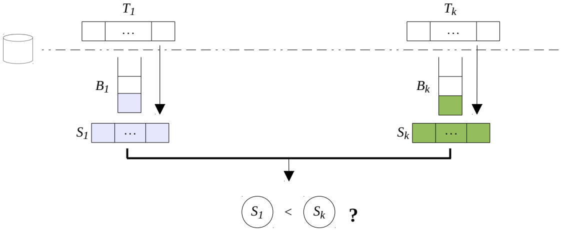
When the output buffer is full, it is written to external memory

Phase 2: Merging

The most sensitive operation is the comparison of elements from each buffer

Naïve approach:

- for each comparison we load into S_i the top suffix of B_i
- It may require too many random disk accesses



Enhanced comparison method:

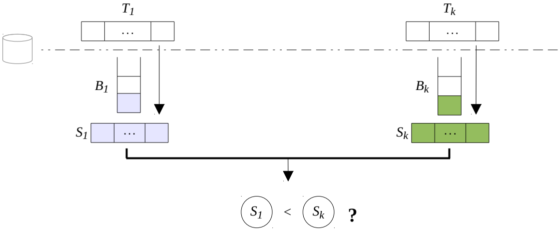
To reduce disk accesses, we propose three strategies: (i) prefix assembling; (ii) *lcp* comparisons; and (iii) inducing suffixes

Phase 2: Merging

The most sensitive operation is the comparison of elements from each buffer

Naïve approach:

- for each comparison we load into S_i the top suffix of B_i
- It may require too many random disk accesses



Enhanced comparison method:

To reduce disk accesses, we propose three strategies: (i) prefix assembling; (ii) *lcp* comparisons; and (iii) inducing suffixes

Phase 2: Merging

(i) Prefix assembling

- PRE_i is used to load the initial prefix of $T_i[SA_i[j], n_i]$ into S_i
- Using LCP_i and PRE_i we can concatenate $(\cdot)$ previous $PRE_i[k]$
  - $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$
  - $h_j = min(LCP_i[j], h_{j-1} + p),\ h_0 = 0$

j	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$T_1[SA[j], n_1]$
...	...	...	...	...	...
5	4	0	A	GA	GA$
...	...	...	...	...	...

S_1					
	G	A	#	A	#

Example:

$j = 5,\ h_5 = 0$
$S_1 = GA\#$

I/O operations → only if the string comparison reaches #

Phase 2: Merging

(i) Prefix assembling

- PRE_i is used to load the initial prefix of $T_i[SA_i[j], n_i]$ into S_i
- Using LCP_i and PRE_i we can concatenate $(\cdot)$ previous $PRE_i[k]$
  - $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$
  - $h_j = min(LCP_i[j], h_{j-1} + p), \; h_0 = 0$

j	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$T_1[SA[j], n_1]$
...	...	...	...	...	...
5	4	0	A	GA	GA$
6	0	2	$	TA	GATA

S_1	G	A	T	A	#

Example:

$j = 6, \; h_6 = min(LCP_i[6], h_5 + p) = min(2, 0 + 2) = 2$
$S_1 = S_1[1, 2] \cdot PRE_1[5] \cdot \# = GA \cdot TA \cdot \#$

I/O operations $\rightarrow$ only if the string comparison reachs $\#$

Phase 2: Merging

(i) Prefix assembling

- PRE_i is used to load the initial prefix of $T_i[SA_i[j], n_i]$ into S_i
- Using LCP_i and PRE_i we can concatenate $(\cdot)$ previous $PRE_i[k]$
  - $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$
  - $h_j = min(LCP_i[j], h_{j-1} + p),\ h_0 = 0$

j	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	$T_1[SA[j], n_1]$
...	...	...	...	...	...
5	4	0	A	GA	GA\$
6	0	2	\$	TA	GATA

S_1	G	A	T	A	#

Example:

$j = 6,\ h_6 = min(LCP_i[6], h_5 + p) = min(2, 0 + 2) = 2$
$S_1 = S_1[1, 2] \cdot PRE_1[5] \cdot \# = GA \cdot TA \cdot \#$

I/O operations $\rightarrow$ only if the string comparison reachs $\#$

Phase 2: Merging

(ii) *LCP* comparisons

- The *lcp* values can be used to speed up suffix comparisons in the heap

Lemma 1:

Let $S_1 < S_2$ and $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ then $lcp(S_2, S_k) \geq l$

We can compare S_2, S_k from l

Phase 2: Merging

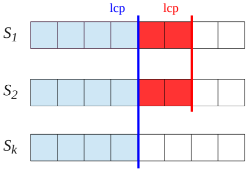
(ii) *LCP* comparisons

- The *lcp* values can be used to speed up suffix comparisons in the heap

Lemma 1:

Let $S_1 < S_2$ and $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ then $lcp(S_2, S_k) \geq l$



We can compare S_2, S_k from l

Phase 2: Merging

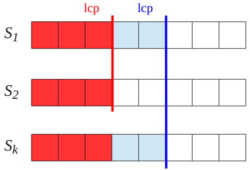
(ii) *LCP* comparisons

- The *lcp* values can be used to speed up suffix comparisons in the heap

Lemma 1:

Let $S_1 < S_2$ and $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ then $lcp(S_2, S_k) \geq l$



We can compare S_2, S_k from l

Phase 2: Merging

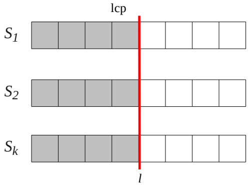
(ii) *LCP* comparisons

- The *lcp* values can be used to speed up suffix comparisons in the heap

Lemma 1:

Let $S_1 < S_2$ and $S_1 < S_k$

- $lcp(S_1, S_2) > lcp(S_1, S_k) \iff S_2 < S_k$
- $lcp(S_1, S_2) < lcp(S_1, S_k) \iff S_2 > S_k$
- $lcp(S_1, S_2) = lcp(S_1, S_k) = l$ then $lcp(S_2, S_k) \geq l$



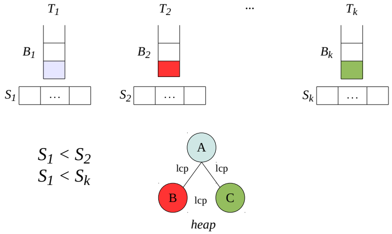
We can compare S_2, S_k from l

Phase 2: Merging

(ii) *LCP* comparisons

Let A, B and C be nodes in the heap storing $B_1[i]$, $B_2[j]$ and $B_k[k]$

- As A is removed from the heap, $B_1[i]$ is moved to the output buffer
- A is replaced by another node D storing $B_1[i+1]$.
- The order of D with respect to its children can be determined by Lemma 1



Example:

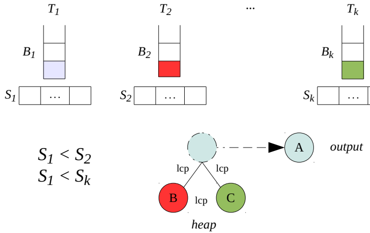
If $lcp(A,D) > lcp(A,B)$ and $lcp(A,D) > lcp(A,C)$ then $D < B$ and $D < C$, D is the next to be removed without string comparisons

Phase 2: Merging

(ii) *LCP* comparisons

Let A, B and C be nodes in the heap storing $B_1[i]$, $B_2[j]$ and $B_k[k]$

- As A is removed from the heap, $B_1[i]$ is moved to the output buffer
- A is replaced by another node D storing $B_1[i+1]$.
- The order of D with respect to its children can be determined by Lemma 1



Example:

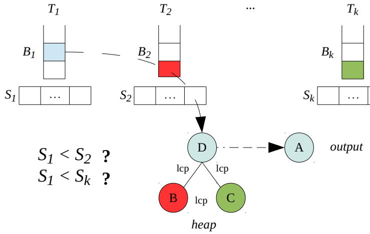
If $lcp(A,D) > lcp(A,B)$ and $lcp(A,D) > lcp(A,C)$ then $D < B$ and $D < C$, D is the next to be removed without string comparisons

Phase 2: Merging

(ii) *LCP* comparisons

Let A, B and C be nodes in the heap storing $B_1[i]$, $B_2[j]$ and $B_k[k]$

- As A is removed from the heap, $B_1[i]$ is moved to the output buffer
- A is replaced by another node D storing $B_1[i+1]$.
- The order of D with respect to its children can be determined by Lemma 1



Example:

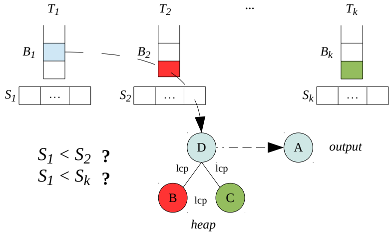
If $lcp(A, D) > lcp(A, B)$ and $lcp(A, D) > lcp(A, C)$ then $D < B$ and $D < C$, D is the next to be removed without string comparisons

Phase 2: Merging

(ii) *LCP* comparisons

Let A, B and C be nodes in the heap storing $B_1[i]$, $B_2[j]$ and $B_k[k]$

- As A is removed from the heap, $B_1[i]$ is moved to the output buffer
- A is replaced by another node D storing $B_1[i+1]$.
- The order of D with respect to its children can be determined by Lemma 1



Example:

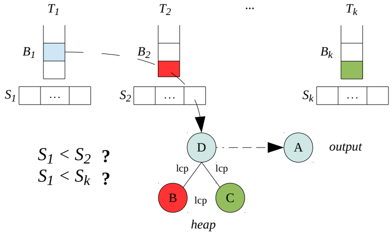
If $lcp(A, D) > lcp(A, B)$ and $lcp(A, D) > lcp(A, C)$ then $D < B$ and $D < C$, D is the next to be removed without string comparisons

Phase 2: Merging

(ii) *LCP* comparisons

Let A, B and C be nodes in the heap storing $B_1[i]$, $B_2[j]$ and $B_k[k]$

- As A is removed from the heap, $B_1[i]$ is moved to the output buffer
- A is replaced by another node D storing $B_1[i+1]$.
- The order of D with respect to its children can be determined by Lemma 1



Example:

If $lcp(A, D) > lcp(A, B)$ and $lcp(A, D) > lcp(A, C)$ then $D < B$ and $D < C$, D is the next to be removed without string comparisons

Phase 2: Merging

(iii) Inducing Suffixes

- We can determine the order of unsorted suffixes from already sorted suffixes

It is used by many suffix sorting algorithms

Lemma 2:

Let *Suff* be the set of all suffixes of $\mathcal{T}$

- If $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ is the lowest element of *Suff*
- then $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ is the lowest β-suffix of $Suff \setminus \{T_i[j, n_i]\}$

Phase 2: Merging

(iii) Inducing Suffixes

- We can determine the order of unsorted suffixes from already sorted suffixes

It is used by many suffix sorting algorithms

Lemma 2:

Let *Suff* be the set of all suffixes of $\mathcal{T}$

- If $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ is the lowest element of *Suff*
- then $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ is the lowest β-suffix of $Suff \setminus \{T_i[j, n_i]\}$

Phase 2: Merging

(iii) Inducing Suffixes

- We can determine the order of unsorted suffixes from already sorted suffixes

It is used by many suffix sorting algorithms

Lemma 2:

Let *Suff* be the set of all suffixes of $\mathcal{T}$

- If $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ is the lowest element of *Suff*
- then $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ is the lowest β-suffix of $Suff \setminus \{T_i[j, n_i]\}$

Induce:

- Remove $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket
- β-bucket: a partition of *SA* that contains only β-suffixes

Note that if $\alpha > \beta$ the suffix $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ was already sorted

Phase 2: Merging

(iii) Inducing Suffixes

Using Lemma 2 to induce suffixes in the merge algorithm:

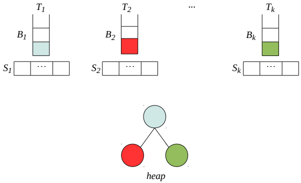
- *Suff* is the set of all unsorted suffixes of $\mathcal{T}$ (remaining in B_i)
- Find the lowest suffix $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ output buffer
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ if $\alpha < \beta$ (using $BWT_i[j]$)
- When the first β-suffix $T_i[j-1, n_i]$ is the lowest in the heap, β-bucket is read from external memory, and induces other suffixes as necessary

Phase 2: Merging

(iii) Inducing Suffixes

Using Lemma 2 to induce suffixes in the merge algorithm:

- *Suff* is the set of all unsorted suffixes of $\mathcal{T}$ (remaining in B_i)
- Find the lowest suffix $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i] \rightarrow$ output buffer
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ if $\alpha < \beta$ (using $BWT_i[j]$)
- When the first β-suffix $T_i[j-1, n_i]$ is the lowest in the heap, β-bucket is read from external memory, and induces other suffixes as necessary

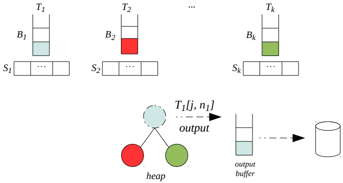


Phase 2: Merging

(iii) Inducing Suffixes

Using Lemma 2 to induce suffixes in the merge algorithm:

- *Suff* is the set of all unsorted suffixes of $\mathcal{T}$ (remaining in B_i)
- Find the lowest suffix $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i]$ → output buffer
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ if $\alpha < \beta$ (using $BWT_i[j]$)
- When the first β-suffix $T_i[j-1, n_i]$ is the lowest in the heap, β-bucket is read from external memory, and induces other suffixes as necessary

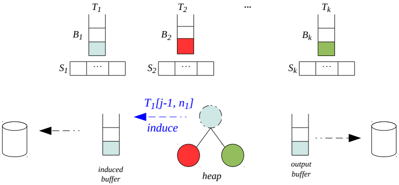


Phase 2: Merging

(iii) Inducing Suffixes

Using Lemma 2 to induce suffixes in the merge algorithm:

- *Suff* is the set of all unsorted suffixes of $\mathcal{T}$ (remaining in B_i)
- Find the lowest suffix $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i]$ → output buffer
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ if $\alpha < \beta$ (using $BWT_i[j]$)
- When the first β-suffix $T_i[j-1, n_i]$ is the lowest in the heap, β-bucket is read from external memory, and induces other suffixes as necessary

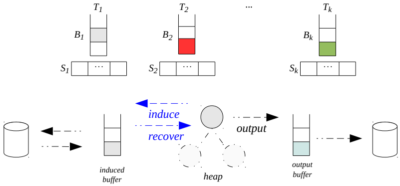


Phase 2: Merging

(iii) Inducing Suffixes

Using Lemma 2 to induce suffixes in the merge algorithm:

- *Suff* is the set of all unsorted suffixes of $\mathcal{T}$ (remaining in B_i)
- Find the lowest suffix $T_1[j, n_1] = \alpha \cdot T_i[j+1, n_i]$ → output buffer
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ if $\alpha < \beta$ (using $BWT_i[j]$)
- When the first β-suffix $T_i[j-1, n_i]$ is the lowest in the heap, β-bucket is read from external memory, and induces other suffixes as necessary



We do not need to compare the induced suffixes
Follow the order in the β-bucket removing elements from B_i

Phase 2: Merging

(iii) Inducing Suffixes

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap

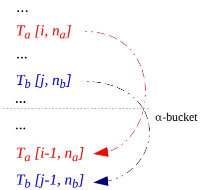
- Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$

Phase 2: Merging

(iii) Inducing Suffixes

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap

- Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$

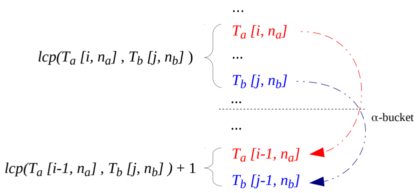


Phase 2: Merging

(iii) Inducing Suffixes

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap

- Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$



Range minimum query on LCP:

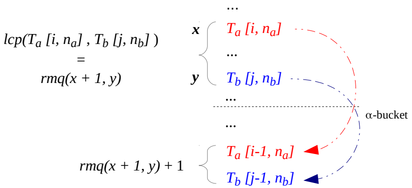
- $rmq(i, j) = \min_{i \leq k \leq j}\{LCP[k]\}$

Phase 2: Merging

(iii) Inducing Suffixes

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap

- Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$



Range minimum query on LCP:

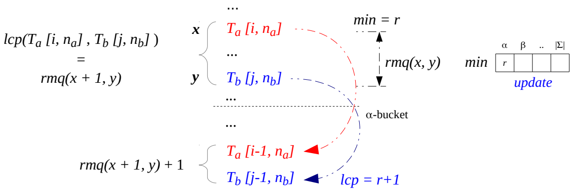
- $LCP(T_a[j, n_a], T_b[j, n_b]) = rmq(x+1, y)$

Phase 2: Merging

(iii) Inducing Suffixes

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap

- Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$



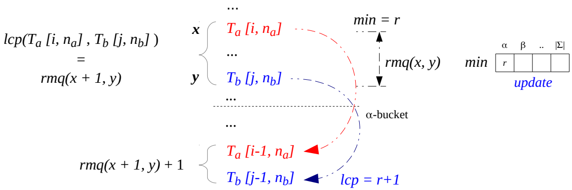
The *rmq* values are computed storing the *min* function for each $\alpha \in \Sigma$ as the *GSA* and LCP are outputed

Phase 2: Merging

(iii) Inducing Suffixes

The *LCP* values of the induced suffixes must also be induced, since they are not calculated when the induced suffixes are not compared in the heap

- Let $T_a[i, n_a]$ be a suffix that induces an α-suffix and let $T_b[j, n_b]$ be the suffix that induces the following α-suffix
- $LCP(T_a[i-1, n_a], T_b[j-1, n_b]) = LCP(T_a[i, n_a], T_b[j, n_b]) + 1.$



When an α-suffix is induced, $min[\alpha] \leftarrow \infty$, and $min[\alpha]$ is computed until the next α-suffix is induced

Performance Evaluation

The performance of eGSA was analyzed through tests with DNA sequences from the genomes:

- (1) Human, (2) Medaka, (3) Zebrafish, (4) Cow, (5) Mouse and (6) Chicken, which were obtained from the Ensembl genome database[1]

Datasets:

Dataset	Genomes	Number of strings	mean LCP	max. LCP	Input size (GB)
1	2	24	19	2,573	0.54
2	6	30	17	5,476	0.92
3	3, 6	56	58	71,314	2.18
4	2, 3, 4	80	44	71,314	4.26
5	1, 4, 5, 6	105	59	168,246	8.50

The mean and max. LCP values provide an approximation of sorting difficulty
Each character in a dataset uses one byte

[1] http://www.ensembl.org/

Performance Evaluation

The performance of eGSA was analyzed through tests with DNA sequences from the genomes:

- (1) Human, (2) Medaka, (3) Zebrafish, (4) Cow, (5) Mouse and (6) Chicken, which were obtained from the Ensembl genome database[1]

Datasets:

Dataset	Genomes	Number of strings	mean LCP	max. LCP	Input size (GB)
1	2	24	19	2,573	0.54
2	6	30	17	5,476	0.92
3	3, 6	56	58	71,314	2.18
4	2, 3, 4	80	44	71,314	4.26
5	1, 4, 5, 6	105	59	168,246	8.50

The mean and max. LCP values provide an approximation of sorting difficulty
Each character in a dataset uses one byte

[1] http://www.ensembl.org/

Performance Evaluation

eGSA was implemented in ANSI/C

- Phase 1: *inducing+sais-lite* algorithm [Fischer, 2011] was used to compute SA_i and LCP_i
- Phase 2: The buffers S_i, B_i, output and induced were set to use 200 KB, 10 MB, 64 MB and 16 MB of internal memory, respectively

The source code is freely available from `http://code.google.com/p/egsa/`

Comparison with eSAIS algorithm [Bingmann et al., 2013]:

- eSAIS is the fastest algorithm to date that computes both suffix and LCP arrays in external memory for a single string
- To index a set of strings, we concatenated all strings in $\mathcal{T}$ into a single one $\mathcal{T} = T_1\$_1 T_2\$_2 \ldots T_k\$_k$, such that $\$_i < \$_j$ if $i < j$ and $\$_i < \alpha$ for each $\alpha \in \Sigma$

The amount of internal memory was restricted to 4 GB for both algorithms

Performance Evaluation

eGSA was implemented in ANSI/C

- Phase 1: *inducing+sais-lite* algorithm [Fischer, 2011] was used to compute SA_i and LCP_i
- Phase 2: The buffers S_i, B_i, output and induced were set to use 200 KB, 10 MB, 64 MB and 16 MB of internal memory, respectively

The source code is freely available from `http://code.google.com/p/egsa/`

Comparison with eSAIS algorithm [Bingmann et al., 2013]:

- eSAIS is the fastest algorithm to date that computes both suffix and LCP arrays in external memory for a single string
- To index a set of strings, we concatenated all strings in $\mathcal{T}$ into a single one $\mathcal{T} = T_1\$_1 T_2\$_2 \ldots T_k\$_k$, such that $\$_i < \$_j$ if $i < j$ and $\$_i < \alpha$ for each $\alpha \in \Sigma$

The amount of internal memory was restricted to 4 GB for both algorithms

Performance Evaluation

Experimental results of eGSA and eSAIS execution:

Dataset	μs/input byte		wallclock (sec)		cputime (sec)		efficiency		cputime ratio
	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS/eGSA
1	5.86	**1.72**	3,413	1,005	1,236	687	0.36	0.68	1.80
2	5.97	**1.24**	5,883	1,228	2,110	715	0.36	0.58	2.95
3	6.23	**2.27**	14,596	5,314	4,385	3,349	0.30	0.63	1.31
4	6.41	**2.31**	29,383	10,590	8,542	7,566	0.29	0.71	1.13
5	7.24	**2.79**	66,106	25,502	16,652	13,003	0.25	0.51	1.28

- eGSA have outperformed eSAIS by a factor of 2.5−4.8 in time (columns μs/input byte)
- Then we may conclude that eGSA is an efficient algorithm for generalized suffix and *LCP* arrays construction on external memory

Running time in microseconds per input byte
Efficiency is the proportion of *cputime* by *wallclock*

Performance Evaluation

Experimental results of eGSA and eSAIS execution:

Dataset	μs/input byte		wallclock (sec)		cputime (sec)		efficiency		cputime ratio
	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS/eGSA
1	5.86	**1.72**	3,413	1,005	1,236	687	0.36	0.68	1.80
2	5.97	**1.24**	5,883	1,228	2,110	715	0.36	0.58	2.95
3	6.23	**2.27**	14,596	5,314	4,385	3,349	0.30	0.63	1.31
4	6.41	**2.31**	29,383	10,590	8,542	7,566	0.29	0.71	1.13
5	7.24	**2.79**	66,106	25,502	16,652	13,003	0.25	0.51	1.28

- eGSA have outperformed eSAIS by a factor of 2.5−4.8 in time (columns μs/input byte)
- Then we may conclude that eGSA is an efficient algorithm for generalized suffix and *LCP* arrays construction on external memory

Although the comparison is not totally fair because eSAIS was not designed for multiple strings

Performance Evaluation

Experimental results of eGSA and eSAIS execution:

Dataset	μs/input byte		wallclock (sec)		cputime (sec)		efficiency		cputime ratio
	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS/eGSA
1	5.86	**1.72**	3,413	1,005	1,236	687	0.36	0.68	1.80
2	5.97	**1.24**	5,883	1,228	2,110	715	0.36	0.58	2.95
3	6.23	**2.27**	14,596	5,314	4,385	3,349	0.30	0.63	1.31
4	6.41	**2.31**	29,383	10,590	8,542	7,566	0.29	0.71	1.13
5	7.24	**2.79**	66,106	25,502	16,652	13,003	0.25	0.51	1.28

- eGSA have outperformed eSAIS by a factor of 2.5−4.8 in time (columns μs/input byte)
- Then we may conclude that eGSA is an efficient algorithm for generalized suffix and *LCP* arrays construction on external memory

Although the comparison is not totally fair because eSAIS was not designed for multiple strings

Performance Evaluation

Experimental results of eGSA and eSAIS execution:

Dataset	μs/input byte		wallclock (sec)		cputime (sec)		efficiency		cputime ratio
	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS/eGSA
1	5.86	**1.72**	3,413	1,005	1,236	687	0.36	0.68	1.80
2	5.97	**1.24**	5,883	1,228	2,110	715	0.36	0.58	2.95
3	6.23	**2.27**	14,596	5,314	4,385	3,349	0.30	0.63	1.31
4	6.41	**2.31**	29,383	10,590	8,542	7,566	0.29	0.71	1.13
5	7.24	**2.79**	66,106	25,502	16,652	13,003	0.25	0.51	1.28

- eGSA have outperformed eSAIS by a factor of $2.5-4.8$ in time (columns μs/input byte)
- Then we may conclude that eGSA is an efficient algorithm for generalized suffix and *LCP* arrays construction on external memory

Phase 2 of eGSA used only 1.1 GB of internal memory for dataset 5
The proportion of induced suffixes is 37.4% on the average

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

Ongoing work:

- Constructing a generalized *Burrows-Wheeler transform* of a set of strings
- Considering multiple disks, one for write operations and the others for read operations
- Indexing large sets of small strings (*e.g.* Next-Generation sequencing reads and protein sequences datasets)

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

Ongoing work:

- Constructing a generalized *Burrows-Wheeler transform* of a set of strings
- Considering multiple disks, one for write operations and the others for read operations
- Indexing large sets of small strings (*e.g.* Next-Generation sequencing reads and protein sequences datasets)

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

Ongoing work:

- Constructing a generalized *Burrows-Wheeler transform* of a set of strings
- Considering multiple disks, one for write operations and the others for read operations
- Indexing large sets of small strings (*e.g.* Next-Generation sequencing reads and protein sequences datasets)

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

Ongoing work:

- Constructing a generalized *Burrows-Wheeler transform* of a set of strings
- Considering multiple disks, one for write operations and the others for read operations
- Indexing large sets of small strings (*e.g.* Next-Generation sequencing reads and protein sequences datasets)

Final

Thank you for your attention!

Barsky, M., Stege, U., Thomo, A., and Upton, C. (2008).
A new method for indexing genomes using on-disk suffix trees.
Proc. CIKM, 236(1-2):649.

Bauer, M. J., Cox, A. J., Rosone, G., and Sciortino, M. (2012).
Lightweight LCP Construction for Next-Generation Sequencing Datasets.
In *Proc. WABI*, pages 326–337.

Bingmann, T., Fischer, J., and Osipov, V. (2013).
Inducing suffix and lcp arrays in external memory.
In *Proc. ALENEX*, pages 88–103.

Crochemore, M., Ilie, L., Iliopoulos, C. S., Kubica, M., Rytter, W., and Walen, T. (2013).
Computing the longest previous factor.
European J. of Combinatorics, 34(1):15–26.

Ferragina, P., Gagie, T., and Manzini, G. (2012).
Lightweight data indexing and compression in external memory.
Algorithmica, 63(3):707–730.

Fischer, J. (2011).
Inducing the lcp-array.
In *Proc. Algorithms and Data Structures Symp.*, pages 374–385.

Garcia-Molina, H., Widom, J., and Ullman, J. D. (1999).
Database System Implementation.
Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (2007).
A taxonomy of suffix array construction algorithms.
ACM Computing Surveys, 39(2):1–31.

Sinha, R., Puglisi, S. J., Moffat, A., and Turpin, A. (2008).
Improving suffix array locality for fast pattern matching on disk.
In *Proc. ACM SIGMOD*, pages 661–672.

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

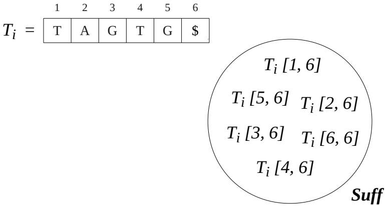
- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

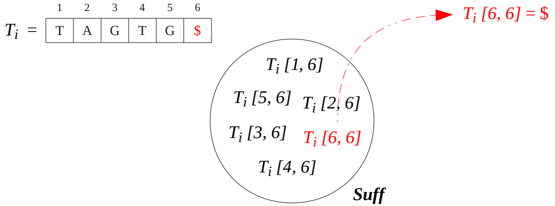


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

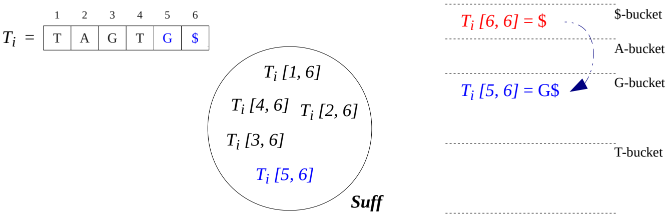


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



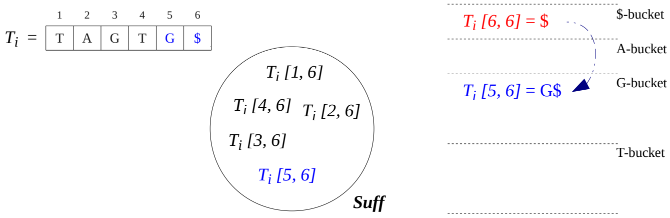
We define β-bucket is a partition of *SA* that contains only suffixes starting with β

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



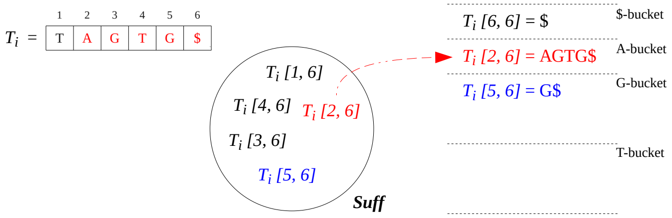
The induced suffixes $T_i[j-1, n_i] = \alpha \cdot T_i[j, n_i]$ cannot be removed from *Suff* because they must induce suffixes $T_i[j-2, n_i]$ as well

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

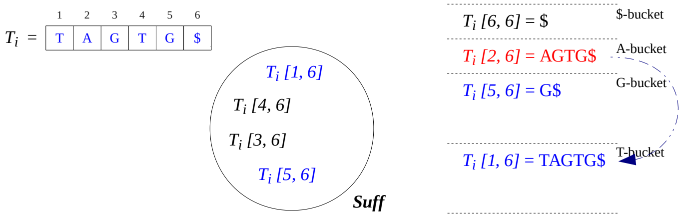


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

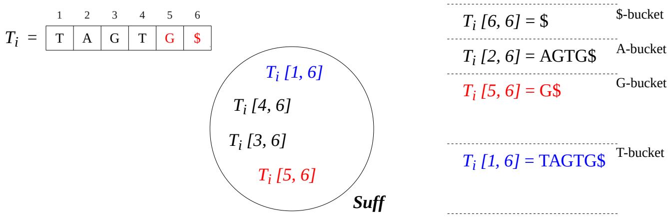


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



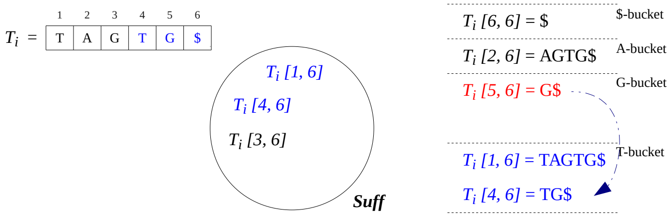
When we reach the β-bucket, as the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from *Suff*

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



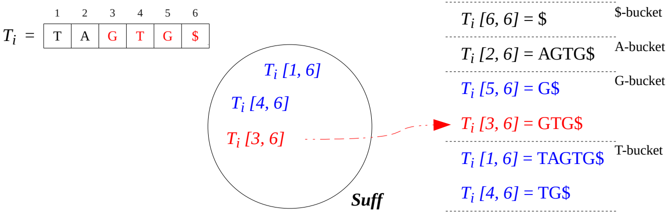
When we reach the β-bucket, as the suffixes $T_i[j-2, n_i]$ are analyzed to be induced, the suffixes $T_i[j-1, n_i]$ are removed from *Suff*

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

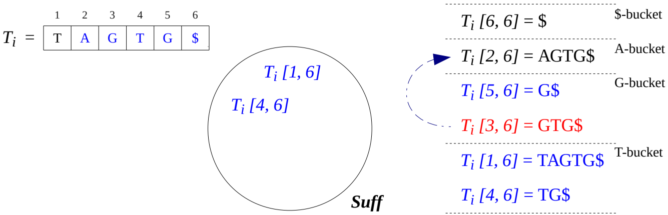


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



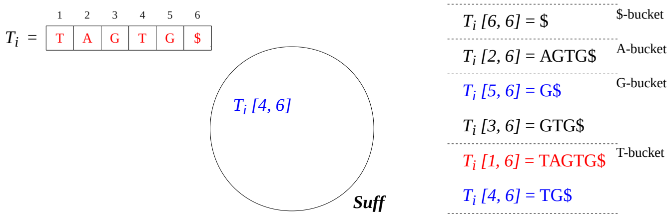
Note that if $\alpha > \beta$ the suffix $T_i[j-1, n_i]$ was already sorted

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

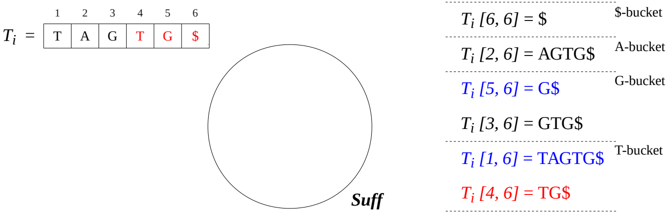


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket

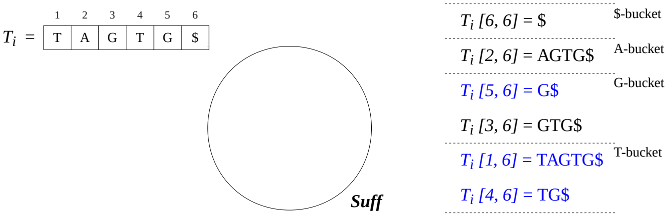


Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



Problem:

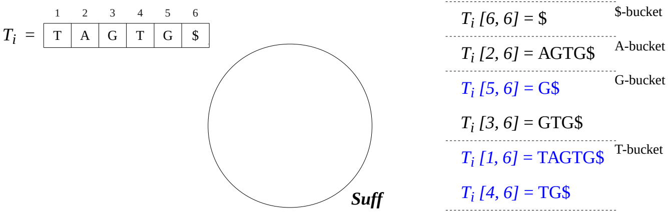
However, this approach is not efficient to sort a single string T_i, since it is always necessary to find the smallest suffix $T_i[j, n_i]$ in *Suff*

Extra:

(iii) Inducing Suffixes

Lemma 2 can be used to sort the suffixes of T_i as follows:

- *Suff* starts with all suffixes of T_i
- Find the smallest suffix $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ and remove it from *Suff*
- Induce $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ to the first available position in the β-bucket



Merging Sorting:

The smallest suffix is one of those nodes in the heap and can be determined efficiently

Phase 2: Merging

(iii) Inducing Suffixes

Prefix assembling must consider the induced suffixes

Let two consecutive suffixes on SA_i, $SA_i[j] = a$ and $SA_i[j+1] = b$

- If $T_i[a, n_i]$ is induced
  - $T_i[a, n_i]$ will be ignored in the heap comparisons (no assembling)
  - $T_i[b, n_i]$ must start the assembling from the begining

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	suffix
...	...	...	...	...	
j	a	0	A	GC	$GC\ \ldots$
j+1	b	1	$	TA	$GTA\ldots$

S_1						
	#	#	#	#	#	wrong

Solving:

- If a suffix will be induced ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- This *lcp* value is always 1, otherwise it will also be induced

Phase 2: Merging

(iii) Inducing Suffixes

Prefix assembling must consider the induced suffixes

Let two consecutive suffixes on SA_i, $SA_i[j] = a$ and $SA_i[j+1] = b$

- If $T_i[a, n_i]$ is induced
  - $T_i[a, n_i]$ will be ignored in the heap comparisons (no assembling)
  - $T_i[b, n_i]$ must start the assembling from the begining

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	suffix
...	...	...	...	...	
j	a	0	A	GC	$GC\ \ldots$
j+1	b	1	\$	TA	$GTA\ldots$

S_1						
	#	#	#	#	#	wrong

Solving:

- If a suffix will be induced ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- This *lcp* value is always 1, otherwise it will also be induced

Phase 2: Merging

(iii) Inducing Suffixes

Prefix assembling must consider the induced suffixes

Let two consecutive suffixes on SA_i, $SA_i[j] = a$ and $SA_i[j+1] = b$

- If $T_i[a, n_i]$ is induced
  - $T_i[a, n_i]$ will be ignored in the heap comparisons (no assembling)
  - $T_i[b, n_i]$ must start the assembling from the begining

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	suffix
...	...	...	...	...	
j	a	0	A	GC	GC ...
j+1	b	1	\$	TA	GTA...

S_1					
	#	T	A	#	#

wrong

Solving:

- If a suffix will be induced ($T_i[a] > T_i[a+1]$) $\rightarrow$ $LCP[j+1] = 0$
- This *lcp* value is always 1, otherwise it will also be induced

Phase 2: Merging

(iii) Inducing Suffixes

Prefix assembling must consider the induced suffixes

Let two consecutive suffixes on SA_i, $SA_i[j] = a$ and $SA_i[j+1] = b$

- If $T_i[a, n_i]$ is induced
  - $T_i[a, n_i]$ will be ignored in the heap comparisons (no assembling)
  - $T_i[b, n_i]$ must start the assembling from the begining

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	suffix
...	...	...	...	...	
j	a	0	A	GC	$GC\ \ldots$
j+1	b	0	$	GT	$GTA\ldots$

S_1	#	#	#	#	#	wrong

Solving:

- If a suffix will be induced $(T_i[a] > T_i[a+1]) \rightarrow LCP[j+1] = 0$
- This *lcp* value is always 1, otherwise it will also be induced

Phase 2: Merging

(iii) Inducing Suffixes

Prefix assembling must consider the induced suffixes

Let two consecutive suffixes on SA_i, $SA_i[j] = a$ and $SA_i[j+1] = b$

- If $T_i[a, n_i]$ is induced
  - $T_i[a, n_i]$ will be ignored in the heap comparisons (no assembling)
  - $T_i[b, n_i]$ must start the assembling from the begining

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	suffix
...	...	...	...	...	
j	a	0	A	GC	$GC\ \ldots$
j+1	b	0	\$	GT	$GTA\ldots$

S_1	G	T	#	#	#	wrong

Solving:

- If a suffix will be induced $(T_i[a] > T_i[a+1]) \rightarrow LCP[j+1] = 0$
- This *lcp* value is always 1, otherwise it will also be induced

Phase 2: Merging

(iii) Inducing Suffixes

Prefix assembling must consider the induced suffixes

Let two consecutive suffixes on SA_i, $SA_i[j] = a$ and $SA_i[j+1] = b$

- If $T_i[a, n_i]$ is induced
  - $T_i[a, n_i]$ will be ignored in the heap comparisons (no assembling)
  - $T_i[b, n_i]$ must start the assembling from the begining

	$SA_1[j]$	$LCP_1[j]$	BWT_i	$PRE_1[j]$	suffix
...	...	...	...	...	
j	a	0	A	GC	$GC\ \ldots$
j+1	b	0	$	GT	$GTA\ldots$

S_1					
	G	T	#	#	#

wrong

Solving:

- If a suffix will be induced $(T_i[a] > T_i[a+1]) \rightarrow LCP[j+1] = 0$
- This *lcp* value is always 1, otherwise it will also be induced

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

eGSA can be employed to construct:

1. *GSAs* and LCP arrays from *SA* and LCP arrays that have already been computed individually for strings in a dataset
2. The core data structures used by LOF-SA search algorithms [Sinha et al., 2008]
3. Generalized suffix trees in external memory [Barsky et al., 2008]
4. Longest Previous Factor array [Crochemore et al., 2013]

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

eGSA can be employed to construct:

1. *GSAs* and LCP arrays from *SA* and LCP arrays that have already been computed individually for strings in a dataset
2. The core data structures used by LOF-SA search algorithms [Sinha et al., 2008]
3. Generalized suffix trees in external memory [Barsky et al., 2008]
4. Longest Previous Factor array [Crochemore et al., 2013]

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

eGSA can be employed to construct:

1. *GSAs* and LCP arrays from *SA* and LCP arrays that have already been computed individually for strings in a dataset
2. The core data structures used by LOF-SA search algorithms [Sinha et al., 2008]
3. Generalized suffix trees in external memory [Barsky et al., 2008]
4. Longest Previous Factor array [Crochemore et al., 2013]

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

eGSA can be employed to construct:

1. *GSAs* and LCP arrays from *SA* and LCP arrays that have already been computed individually for strings in a dataset
2. The core data structures used by LOF-SA search algorithms [Sinha et al., 2008]
3. Generalized suffix trees in external memory [Barsky et al., 2008]
4. Longest Previous Factor array [Crochemore et al., 2013]

Conclusion

Our contribuition:

eGSA, the first external memory algorithm to construct both generalized suffix and LCP arrays for sets of large strings

eGSA can be employed to construct:

1. *GSAs* and LCP arrays from *SA* and LCP arrays that have already been computed individually for strings in a dataset
2. The core data structures used by LOF-SA search algorithms [Sinha et al., 2008]
3. Generalized suffix trees in external memory [Barsky et al., 2008]
4. Longest Previous Factor array [Crochemore et al., 2013]