Li, Li, Huo - Optimal In-Place Suffix Sorting

Patrick Indri

June 9, 2019

DSSC - Algorithmic Design Exam

Index

- Problem setting;
- Naive solution;
- Preliminary notions;
- Suffix sorting for read-only integer alphabets;
- Additional results and conclusions;
- Auxiliary material.

Problem Setting

Suffix Arrays: a space-saving alternative to suffix trees.

Definition

Given a string $T = T[0, \ldots, n-1]$ where each $T[i] \in \Sigma$ integer alphabet, the *suffix array* SA contains the indices of all suffixes of T which are sorted in lexicographical order.

Problem Setting

Suffix Arrays: a space-saving alternative to suffix trees.

Definition

Given a string $T = T[0, \ldots, n-1]$ where each $T[i] \in \Sigma$ integer alphabet, the *suffix array* SA contains the indices of all suffixes of T which are sorted in lexicographical order.

Example

$T =$ "1120", the suffixes are $\{1120, 220, 20, 0\}$. Since $suf(3) < suf(0) < suf(2) < suf(1)$, then $SA = [3021]$.

Problem Setting

Problem

Construct SA for a given string T.

Main Theorem

There is an in-place linear time algorithm for suffix sorting over integer alphabets, even if the input string T is read only and the size of the alphabet $|\Sigma|$ is $O(n)$.

Naive Solution

- Get all the suffixes and sort them using *Quicksort*, while retaining their original indices. $O(n \log n)$ comparisons for sorting, $O(n)$ to compare suffixes: worst case is $O(n^2 \log n)$.
- Build a suffix tree in $O(n)$ and perform a depth-first traversal on it in $O(n)$.

Preliminary Notions

Notations:

- $suf(i)$ is said to be S-suffix if $suf(i) < suf(i+1)$. Otherwise, it is L-suffix;
- $suf(i)$ is said to be LMS-suffix if $suf(i)$ is S-suffix and $suf(i-1)$ is L-suffix;

Note

Types (S or L, and LMS) can be computed by a linear scan of T.

Preliminary Notions

Notations:

- $suf(i)$ is said to be S-suffix if $suf(i) < suf(i+1)$. Otherwise, it is L-suffix;
- $suf(i)$ is said to be LMS-suffix if $suf(i)$ is S-suffix and $suf(i-1)$ is L-suffix;

Note

Types (S or L, and LMS) can be computed by a linear scan of T.

Example: $T =$ "3122120"

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

Suffix Sorting for Read-only Integer Alphabets

Definitions and assumptions:

- n_S (n_L) denotes the number of S-suffixes (L-suffixes);
- $n_L \leq n_S$;
- n_1 denotes the number of LMS-suffixes;
- $SA[0, \ldots, n-1]$ will store the result;
- Bucket: set of suffixes with the same first character.

Suffix Sorting for Read-only Integer Alphabets

Definitions and assumptions:

- n_S (n_L) denotes the number of S-suffixes (L-suffixes);
- $n_L \leq n_S$;
- n_1 denotes the number of LMS-suffixes;
- $SA[0, \ldots, n-1]$ will store the result;
- Bucket: set of suffixes with the same first character.

Algorithm:

1. Sort all LMS-characters of T;
2. Induced sort all LMS-substrings from sorted LMS-characters;
3. Construct the reduced problem T_1 from sorted LMS-substrings;
4. Sort the LMS-suffixes by recursively solving T_1;
5. Induced sort all suffixes of T from the sorted LMS-suffixes.

1. Sort all *LMS*-characters of T

The *LMS*-characters can be sorted with *Counting Sort*, using $SA[0, \ldots, n/2]$ as the counting array and storing the result in $SA[n - n_1, \ldots, n - 1]$. AUX

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

SA								
						7	1	4

The sorting step takes $O(n)$ time and uses $O(1)$ workspace.

2. Induced sort all *LMS*-substrings from sorted *LMS*-characters

This step is analogous to step 5.

After this step, indices of the ordered *LMS*-substrings are stored in $SA[n - n_1, \ldots, n - 1]$.

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

SA						7	1	4
SA						7	4	1

LMS-substrings are $\{1121, 1120, 0\}$.

3. Construct the reduced problem T_1

Using the lexicographical order of the *LMS*-substrings, build the reduced problem T_1 and store it in $T[0, \ldots, n_1]$. T_1 can be obtained by a liner scan of *SA*, thus using $O(n)$ time and $O(1)$ workspace.

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

SA						7	4	1
SA	2	1	0			7	4	1

LMS-substrings are $\{1121, 1120, 0\}$.

4. Sort the *LMS*-suffixes by recursively solving T_1

T_1 can be solved iteratively in linear time[1] with no additional workspace. It is stored at the beginning of *SA*. The *LMS*-suffixes are sorted using linear scans and the solution of T_1.

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

SA	2	1	0					
SA	2	1	0			1	4	7
SA	7	4	1			1	4	7
SA						7	4	1

LMS-suffixes are $\{1221120, 1120, 0\}$.

The complexity of this step is $O(n)$ in time and $O(1)$ in workspace.

[1] $\mathcal{T}(n) = \mathcal{T}(n/2) + n = n(1 + 1/2 + 1/4 + 1/8 + \cdots + 1/\log_2 n) \in \Theta(n)$

5. Induced sort all suffixes of T from the sorted LMS-suffixes

It can be demonstrated that sorting the n_L L-suffixes from the sorted LMS-suffixes is symmetrical as sorting the n_S S-suffixes from the sorted L-suffixes. Suppose L-suffixes are sorted.

Index	0	1	2	3	4	5	6	7
T	3	1	2	2	1	1	2	0
Type	L	S	L	L	S	S	L	S
LMS		*			*			*

SA						7	4	1
SA	6	3	2	0				

L-suffixes are $\{31221120, 221120, 21120, 20\}$.

5. Induced sort all suffixes of T from the sorted LMS-suffixes

Pointer Data Structure

Built in linear time, indicates the bucket tails of a S-suffix in constant time. Occupies at most $c_P = cn/\log n$ words, placed at the end of SA. AUX

Interior Counter Trick

Dynamically maintain the RF-pointers (rightmost free pointers) for each bucket. AUX

5. Induced sort all suffixes of T from the sorted LMS-suffixes

Pointer Data Structure

Built in linear time, indicates the bucket tails of a S-suffix in constant time. Occupies at most $c_P = cn/\log n$ words, placed at the end of SA. AUX

Interior Counter Trick

Dynamically maintain the RF-pointers (rightmost free pointers) for each bucket. AUX

The ordering of the S-suffixes proceeds in two steps:

1. Construct a *pointer data structure* $\mathcal{P}$ and, combining it with the *interior counter trick* induce the first $n_S - c_P$ S-suffixes;
2. Use *Binary Search* and the *Interior Counter Trick* on the last c_P S-suffixes.

5. Induced Sort Algorithm

Suppose *L*-indices are already sorted in their buckets.

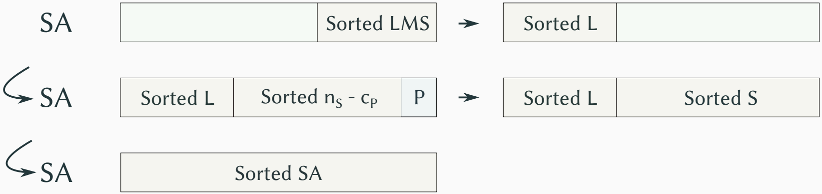
```
def InducedSort(T, SA):
  for i = nL downto 0:
    j = SA[i] - 1 # Indicizes the bucket.
    if T[j] is S-type:
      *RF[j] = j
      RF[j] = next_free_entry
    endif
  endfor
enddef
```

For each query of *RF*, the tail of the bucket is provided in constant time by the pointer data structure, from that the interior counter trick indicates the *RF* entry.

5. Induced sort all suffixes of T from the sorted LMS-suffixes

The first $n_S - c_P$ S-suffixes can be ordered in linear time with no additional workspace. The remaining suffixes can be sorted analogously, using binary search to find the tails of the buckets, since $c_P \log n = O(n)$ and time linearity is preserved.

A stable, in place linear time merging can be used to merge the sorted S- and L-suffixes.



Additional results and conclusion

(Read-only) Integer Alphabets

Considering all the steps, it follows that the algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of a string T over integer alphabets Σ, where T is read-only and $|\Sigma| = O(n)$.

The result trivially holds for non-read-only integer alphabets.

Additional results and conclusion

(Read-only) Integer Alphabets

Considering all the steps, it follows that the algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of a string T over integer alphabets Σ, where T is read-only and $|\Sigma| = O(n)$.

The result trivially holds for non-read-only integer alphabets.

Read Only General Alphabets

For read-only general alphabets (i.e., only comparisons allowed on T) there is an in-place $O(n \log n)$ time algorithm for suffix sorting.

Additional results and conclusion

(Read-only) Integer Alphabets

Considering all the steps, it follows that the algorithm takes $O(n)$ time and $O(1)$ workspace to compute the suffix array of a string T over integer alphabets Σ, where T is read-only and $|\Sigma| = O(n)$.

The result trivially holds for non-read-only integer alphabets.

Read Only General Alphabets

For read-only general alphabets (i.e., only comparisons allowed on T) there is an in-place $O(n \log n)$ time algorithm for suffix sorting.

References

Li, Zhize, Jian Li, and Hongwei Huo. "Optimal in-place suffix sorting." *International Symposium on String Processing and Information Retrieval*. Springer, Cham, 2018.

Auxiliary Material

AUX: Sort all LMS-characters of T

Since $|\Sigma| = O(n)$, assume $\exists d \in \mathbb{N}$ s.t. $|\Sigma| \leq dn$. Divide LMS-characters in $2d$ partitions, where partition i contains elements in $\left[\frac{i|\Sigma|}{2d} + 1, \frac{(i+1)|\Sigma|}{2d}\right]$. Since

$$\frac{|\Sigma|}{2d} \leq \frac{dn}{2d} = \frac{n}{2},$$

$SA[0, \ldots, n/2]$ can be used as a counting array. ↩

AUX: Interior counter trick - 1

Consider a bucket of size m, indexing it as $SA_S\{0, \ldots, m-1\}$. Define the special symbols B_H, B_T, E, R_1 and R_2. `Index(i)` denotes the index of the i-th S-suffix of the bucket. The position of the tail, i.e., `m-1`, is given by the pointer data structure.

```
def InteriorCounterTrick(SAs):
  SAs[0]=BH, SAs[m-2]=E, SAs[m-1]=BT
  # O(m) time.
  if SAs[m-1]=BT and
    (SAs[m-2]=E or SAs[m-SAs[m-2]-3]!=BH):
    for i=1 upto m-3:
      SAs[m-2-i]=Index(i)
      SAs[m-2]++    # Acts as a counter.
    endfor
  endif
```

AUX: Interior counter trick - 2

```
# O(m) time.
if SAs[m-1]=BT and SAs[m-SAs[m-2]-3]=BH:
  shift SAs[1,...,m-3] to SAs[2,...m-2]
  SAs[1]=Index(m-2)
  SAs[m-1]=R2
endif
# O(m) time.
if SAs[m-1]=R2;
  shift SAs[1,...,m-2] to SAs[2,...,m-1]
  SAs[1]=Index(m-1)
  SAs[0]=R1
endif
```

AUX: Interior counter trick - 3

```
    # O(m) time, need to scan from tail
    # backwards to find R1.
    else:
      SAs[0]=Index(m)
enddef
```

The function consists of four steps, each $O(m)$ time, assuming that the tail of a bucket is known. It uses $O(1)$ workspace and, for all the buckets, results in $O(n)$ time. ↩

AUX: Pointer data structure - 1

Assuming $|\Sigma| \leq dn$, divide the S-suffixes of T in $4d$ parts, according to their first character. Let D_i denote the pointer data structure of the i-th part. D_0 can be constructed as follows (analogously for the others). For brevity, $\mathtt{b} = |\Sigma|/4d$.

```
def PointerDataStructure(T, SAs):
  SAs[i]=1 forall i in [1,b]
  for i=n-1 downto 0:
    if T[i] is S-type and in [1,b]: SAs[T[i]]++
  endfor
```

AUX: Pointer data structure - 2

```
  sum=-1
  for i=1 upto b:
    sum+=SAs[i]
    SAs[i]=sum
  endfor
enddef
```

For every S-suffix for which $T[i] \in SA_S[i, |\Sigma|/4d]$, $SA_S[T[i]] - T[i]$ indicates the tail of the bucket $T[i]$: the tail of a bucket can be obtained in constant time. ↩