Space-Efficient Construction Algorithm for Circular Suffix Tree

Wing-Kai Hon, Tsung-Han Ku, Rahul Shah and Sharma Thankachan

CPM2013

Outline

- Preliminaries and Motivation
- Circular Suffix Tree
- Our Indexes and the Construction Algorithms

Preliminaries

- Let P be a pattern of length $|P|$.
- P^{∞} denote the string of infinite number of times of P
- $Q = P[i..|P|]P[1..i\text{-}1]$ be the i-th cyclic shift of P.

Ex:
Let $P = abcd \Rightarrow P^{\infty} = abcdabcdabcdabcd\ldots\ldots\ldots$

Cyclic shifts of P:
abcd, bcda, cdab, dabc

Circular Dictionary Matching

Given a set $D=\{P_1,\ldots,P_d\}$ of d patterns, the circular dictionary matching problem is to index D such that for any online query text T, we can quickly locate the occurrences of any pattern of D and all of its shifts within T.

In 2009, Hon et al. proposed a solution for the circular dictionary matching problem by using succinct circular suffix tree.

Ordering

Let P and Q be two patterns. We say P is circularly larger than Q, or simply P larger than Q, if and only if P^∞ is lexicographically larger than Q^∞. The notions of "smaller than", or "equal to", are defined analogously.

Example:
$P = abaab$, $Q = ab$ ➔ $P < Q$

Preliminaries

Lemma 1:

Let P and Q be two strings, and let m denote the maximum of $|P|$ and $|Q|$. Then, P is circular smaller than Q if and only if $P^{\infty}[1..2m]$ is lexicographically smaller than $Q^{\infty}[1..2m]$. In other words, to compare the "circular" lexicographical order of P and Q, we only need to compare P^{∞} and Q^{∞} directly up to length $2m$.

Circular Suffix Tree

- The circular suffix tree ST_{circ} for D is a compact trie storing all circular suffixes of all P^{∞}_{j}. In addition, the children of a node are arranged according to the lexicographical order of the labels in the incident edges.

- Consequently, the i-th leftmost leaf in the circular suffix tree will correspond to the i-th lexicographically smallest circular suffix, which is indexed by $SA_{circ}[i]$.

Our Index

Succinct Circular Suffix Tree:

(1) The circular suffix array: for representing the leaf labels and the edge labels.

(2) An *Hgt* array: for representing the length of the edge labels

(3) A string of balanced parentheses: for encoding the tree structure.

Circular Suffix Array

The circular suffix array $SA_{circ}[1..n]$ for D is an arrangement of all circular suffixes of all P^{∞}_{j} according to the lexicographical order.

Lemma 2: (Hon et al. 2012)
The circular suffix array can be stored succinctly in $O(n \log \sigma)$ bits, and can be constructed in $O(n \log n)$ time using $O(n \log \sigma)$ bits working space. The functions SA_{circ} and $SA^{-1}{}_{circ}$ can both be evaluated in $O(\log n)$ time.

Hgt Array

In *Hgt*[1..|*P*|] array, *Hgt*[*i*] stores the length of the longest common prefix between suffix S_i and its predecessor suffix S_{i-1}

Index	Suffix	SA	*Hgt*
1	*abc*	5	-1
2	*abcdabc*	1	3
3	*bc*	6	0
4	*bcdabc*	2	2
5	*c*	7	0
6	*cdabc*	3	1
7	*dabc*	4	0

An example of *Hgt* array for *P* = *abcdabc*

Kasai et al.'s algorithm

- Given the suffix array and inverse suffix array of a string *S*.
- Kasai et al.'s algorithm can compute the *Hgt* array of *S* in linear time.

Kasai et al.'s algorithm

Compute the *Hgt* array of P = *abcdabc*

Index	Suffix	SA	*Hgt*
1	*abc*	5	
2	*abcdabc*	1	3
3	*bc*	6	
4	*bcdabc*	2	
5	*c*	7	
6	*cdabc*	3	
7	*dabc*	4	

$$h = 3$$

Kasai et al.'s algorithm

Compute the *Hgt* array of $P = abcdabc$

Index	Suffix	SA	*Hgt*
1	*abc*	5	
2	*abcdabc*	1	3
3	*bc*	6	
4	*bcdabc*	2	2
5	*c*	7	
6	*cdabc*	3	
7	*dabc*	4	

$$h = 2$$

Kasai et al.'s algorithm

Compute the *Hgt* array of $P = abcdabc$

Index	Suffix	SA	*Hgt*
1	*abc*	5	
2	*abcdabc*	1	3
3	*bc*	6	
4	*bcdabc*	2	2
5	*c*	7	
6	*cdabc*	3	1
7	*dabc*	4	

$$h = 1$$

Kasai et al.'s algorithm

Compute the *Hgt* array of $P = abcdabc$

Index	Suffix	SA	*Hgt*
1	*abc*	5	
2	*abcdabc*	1	3
3	*bc*	6	
4	*bcdabc*	2	2
5	*c*	7	
6	*cdabc*	3	1
7	*dabc*	4	0

$$h = 0$$

Kasai et al.'s algorithm

Compute the *Hgt* array of $P = abcdabc$

Index	Suffix	SA	*Hgt*
1	*abc*	5	-1
2	*abcdabc*	1	3
3	*bc*	6	
4	*bcdabc*	2	2
5	*c*	7	
6	*cdabc*	3	1
7	*dabc*	4	0

$h = 0$

Kasai et al.'s algorithm

Compute the *Hgt* array of $P = abcdabc$

Index	Suffix	SA	*Hgt*
1	*abc*	5	-1
2	*abcdabc*	1	3
3	*bc*	6	0
4	*bcdabc*	2	2
5	*c*	7	
6	*cdabc*	3	1
7	*dabc*	4	0

$$h = 0$$

Kasai et al.'s algorithm

Compute the *Hgt* array of $P = abcdabc$

Index	Suffix	SA	*Hgt*
1	*abc*	5	-1
2	*abcdabc*	1	3
3	*bc*	6	0
4	*bcdabc*	2	2
5	*c*	7	0
6	*cdabc*	3	1
7	*dabc*	4	0

$$h = 0$$

Hgt Array of Circular Suffixes

- The *Hgt* array can naturally be defined for the circular suffix tree for D.
- We can also extend Kasai et al.'s algorithm to construct the *Hgt* array.
- Nevertheless, when we process the circular suffixes of P^{∞}_j in round j, the total time required will become $O(|P_j|+L_j)$, where L_j denotes the maximum length of LCP between P_j and its predecessor circular suffix.
- In worst case, $L_j = O(n)$, the total running time will become $O(dn)$

Hgt Array of Circular Suffixes

Index	Suffix	SA_{circ}	*Hgt*
1	$(ab)^{\infty}$	(2, 2)	-1
2	$(abbb)^{\infty}$	(1,4)	2
3	$(ba)^{\infty}$	(2,1)	0
4	$(babb)^{\infty}$	(1,3)	3
5	$(bbab)^{\infty}$	(1,2)	1
6	$(bbba)^{\infty}$	(1,1)	2
7	$(b)^{\infty}$	(3,1)	3

An example of *Hgt* array for $D = \{bbba, ba, b\}$

Hgt Array of Circular Suffixes

- *Hgt'* stores the LCP between S_i and its successor S_{i+1}.
- → *Hgt*[*i*] = *Hgt'*[*i*-1]

- In our algorithm, in each round, we compute *Hgt*[*i*] and *Hgt'*[*i*] such that when we compute the LCP between S_j and its predecessor, if *Hgt'*[*j*-1] is computed, we can set *Hgt*[*j*] = *Hgt'*[*j*-1] immediately.

Hgt Array of Circular Suffixes

- Up to now, we are actually assuming the $SA[i]$ and $SA^{-1}[i]$ can be computed in constant time.
- Therefore, our algorithm for computing *Hgt* array is linear.

- Since we adopt Hon et al.'s succinct circular suffix array, the time complexity of our algorithm is $O(n \log n)$.

Hgt Array of Circular Suffixes

- Define $H[j, k] = Hgt[i]$, if the k-th circular suffix of P^{∞}_j is S_i.

➔(1) $Hgt[i] = H[j, k] = H[\mathrm{SA}[i]]$

(2) $H[j, 1] \leqq H[j, 2]+1 \leqq H[j, 3]+2 \leqq \ldots$

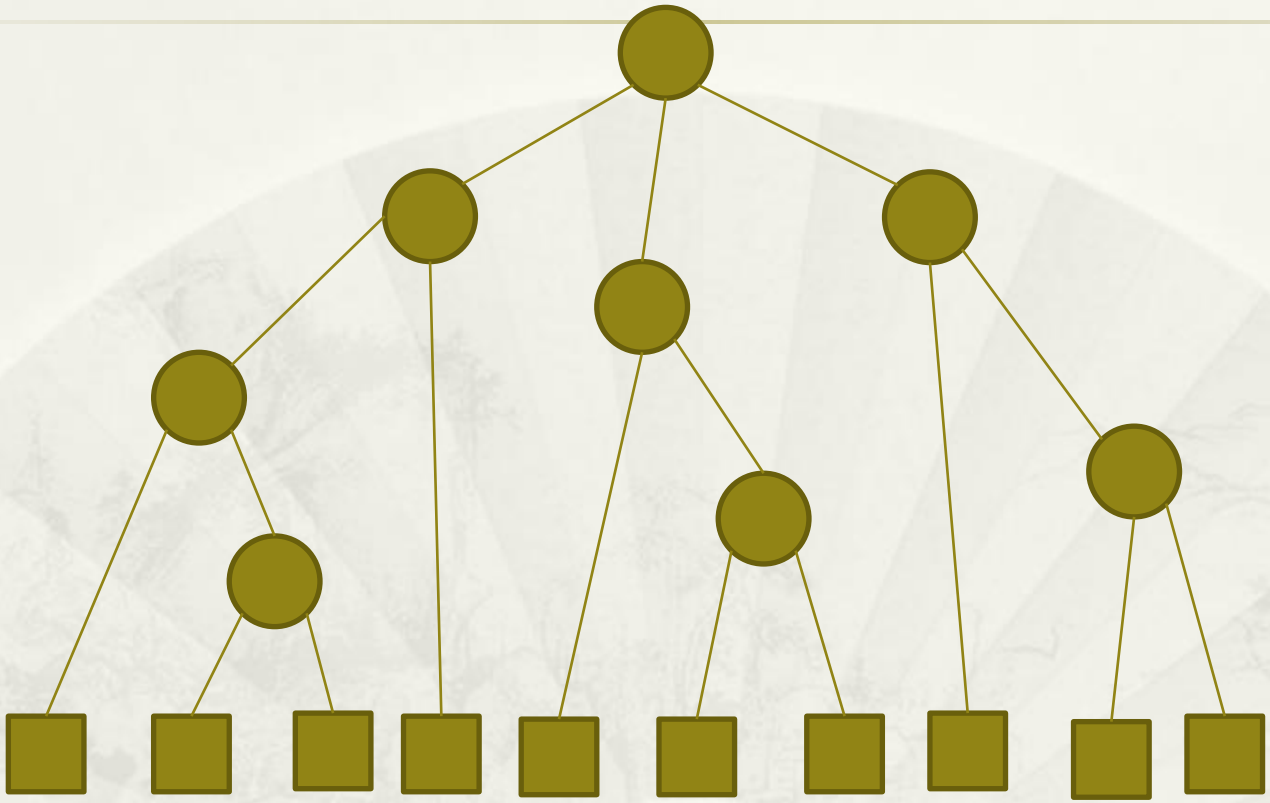
- Since P^{∞}_j is circular, its $(|P_j|+1)$-th circular suffix is the same to itself.

➔$H[j, 1] \leqq H[j, 2]+1 \leqq \ldots \leqq H[j, |P_j|]+|P_j|-1 \leqq H[j, |P_j|+1]+|P_j| = H[j, 1]+|P_j|$

Hgt Array of Circular Suffixes

- In stead of storing *Hgt* array, we store *H* array succinctly (i.e. store the differences).
- Thus, the space complexity of the *H* array is O(d log n+ n) bits.
- Therefore, the Hgt array can be constructed in O(n log n) time using O(n logσ+ d log n) bits of working space.

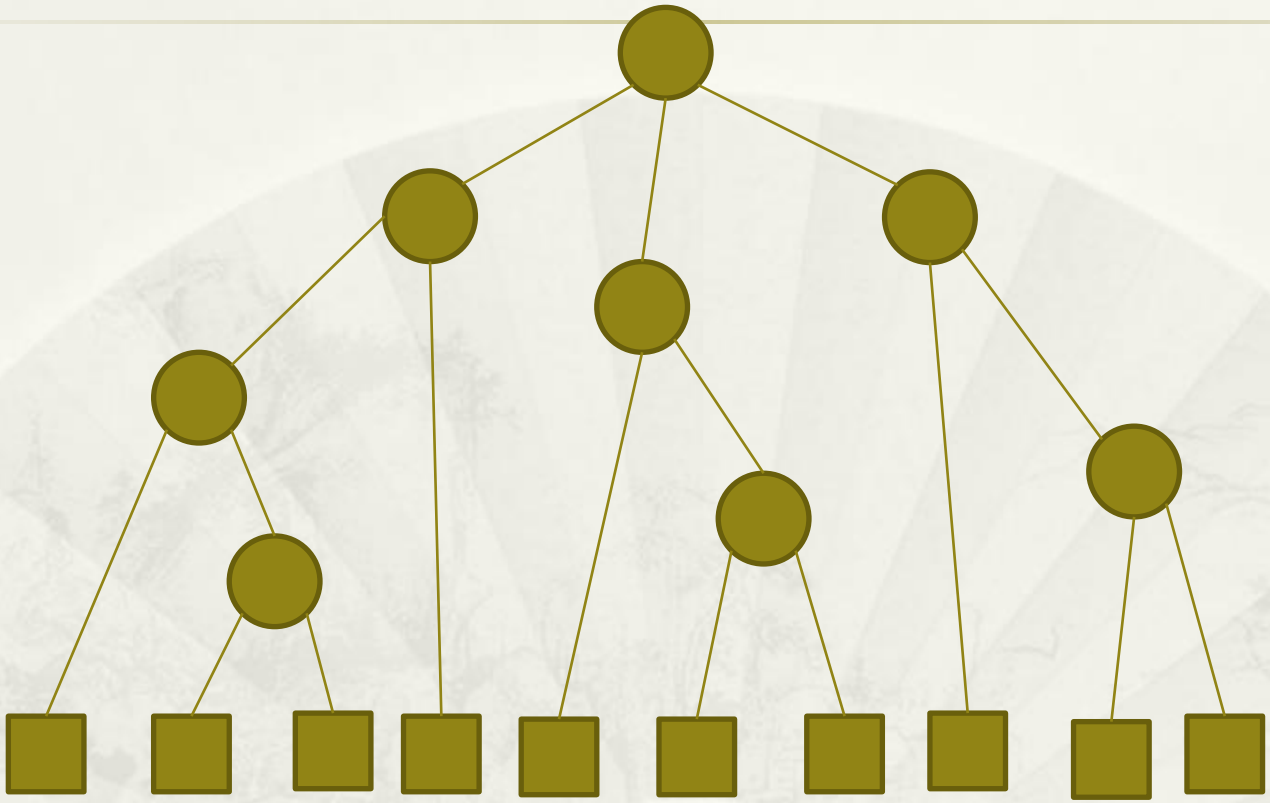
Balanced parentheses representation



The output of the left to right bottom up traversal would be

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

Balanced parentheses representation



The output of the right to left bottom up traversal would be

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

Balanced parentheses representation

The output of the left to right bottom up traversal would be

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

The output of the right to left bottom up traversal would be

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

By using a linear time scanning algorithm, we can compute the balanced parentheses representation of the circular suffix tree.

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

Therefore, we can compute the balanced parentheses representation in O(n log n) time by using O(n) bits working space.

Thank you