max planck institut
informatik

Suffix arrays in linear time

Algorithms on Strings

Paweł Gawrychowski

May 29, 2013

Outline

Constructing the suffix array

LCP

Recall that suffix array is simply the lexicographically sorted list of all suffixes of a given word w.

$w = \texttt{mississippi}$

$SA[1] =$	11	$=$	i
$SA[2] =$	8	$=$	ippi
$SA[3] =$	5	$=$	issippi
$SA[4] =$	2	$=$	ississippi
$SA[5] =$	1	$=$	mississippi
$SA[6] =$	10	$=$	pi
$SA[7] =$	9	$=$	ppi
$SA[8] =$	7	$=$	sippi
$SA[9] =$	4	$=$	sissippi
$SA[10] =$	6	$=$	ssippi
$SA[11] =$	3	$=$	ssissippi

So far we did not care about the size of the alphabet that much. For the reason that should be clear after the next hour, today we will to work with a word w where each letter is an integer from $\{1, 2, \ldots, |w|\}$.

The goal...

is to develop an $\mathcal{O}(n)$ time algorithm, where $n = |w|$.

Let's start with an $\mathcal{O}(n^2)$ time algorithm.

Radix sort

A sequence of n pairs from $\{1, \ldots, n\} \times \{1, \ldots, n\}$ can be sorted in $\mathcal{O}(n)$ time.

$\mathcal{O}(n^2)$ algorithm

For each $i = n, n-1, n-2, \ldots, 1$ construct a sorted list L_i containing all $w[i..n]$, $w[i+1..n]$, $w[i+2..n]$, ... $w[n..n]$.

Assume that we have the list L_{i+1}, and want to construct the list for i. For each $j = i, i+1, \ldots, n$ construct a pair $(w[j], \mathrm{nr}_{i+1}[j+1])$, where $\mathrm{nr}_i(j)$ is the position of $w[j..n]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now let's try to improve the complexity to $\mathcal{O}(n \log n)$.

When you think about the previous solution, some fragments of the word will be compared again... and again...

We apply the *doubling* paradigm. To make the description easier, append n additional null characters to w, so it's actually a word of length $2n$.

$\mathcal{O}(n \log n)$ algorithm

For each $i = 0, 1, 2, \ldots, \log n$ construct a sorted list L_i containing all $w[1..1 + 2^i - 1]$, $w[2..2 + 2^i - 1]$, ..., $w[n..n + 2^i - 1]$.

Assume that we have the list L_i, and want to construct the list for $i + 1$. For each $j = 1, 2, \ldots, n$ construct a pair $(\mathrm{nr}_i(j), \mathrm{nr}_i(j + 2^i))$, where $\mathrm{nr}_i(j)$ is the position of $w[j..j + 2^i - 1]$ on the list L_i. Then sort all pairs, and notice that their order determines L_{i+1}.

Now we are ready for the **real thing**.

Kärkkäinen and Sanders 2003

Suffix array can be constructed in $\mathcal{O}(n)$ time.

The idea is a recursive one. We will try to design the algorithm so that its running time can be expressed as $T(n) = T(\alpha n) + \mathcal{O}(n)$, where α is *some* constant less than 1.

Notice that the recursion solves to $T(n) = \mathcal{O}(n)$.

We partition all suffixes into three groups.

$$
\begin{aligned}
S_0 &= \{w[3..n], w[6..n], w[9..n], \ldots\} \\
S_1 &= \{w[1..n], w[4..n], w[7..n], \ldots\} \\
S_2 &= \{w[2..n], w[5..n], w[8..n], \ldots\}
\end{aligned}
$$

S_r are all suffixes that start at positions of the form $3k + r$.

The goal is to sort all suffixes. We could start with something simpler: sorting (separately) each S_r.

First trick

Say that we want to sort only S_1. We could split w into blocks of length 3, treat each block as a single letter, and recursively solve a smaller problem of size $\frac{n}{3}$.

$w[1]$	$w[2]$	$w[3]$	$w[4]$	$w[5]$	$w[6]$	$\cdots$	$w[n\text{-}2]$	$w[n\text{-}1]$	$w[n]$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

$(w[1], w[2], w[3])$	$(w[4], w[5], w[6])$	$\cdots$	$(w[n\text{-}2], w[n\text{-}1], w[n])$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

$\mathrm{nr}(1)$	$\mathrm{nr}(4)$	$\cdots$	$\mathrm{nr}(n-2)$

We must be careful here: we promised that the input word of length $|w|$ will contain only letters $\{1, 2, \ldots, |w|\}$, and here we create triples of letters.

Alphabet renaming

Create a sorted list of all triples $(w[i], w[i+1], w[i+2])$ and then compute the number $\mathrm{nr}(i)$ of each triples there. This can be done in $\mathcal{O}(n)$ time using radix sort.

We append two null characters to w, so that the expression $(w[i], w[i+1], w[i+2])$ always makes sense.

OK, so we can sort S_0. Similarly, we can sort S_1 and S_2, but we cannot afford to sort all of them!

Second trick

Assuming that we have already sorted $S_1 \cup S_2$, we can sort $S_0 \cup S_1 \cup S_2$ in $\mathcal{O}(n)$ time. For this we represent each suffix as a pair:

- $w[3k..n]$ becomes $(w[3k], w[3k+1..n])$,
- $w[3k+1..n]$ becomes $(w[3k+1], w[3k+2..n])$,
- $w[3k+2..n]$ becomes $(w[3k+2], w[3(k+1)..n])$.

The order on pairs is clearly the same as order on suffixes.

Why this works?
We already have sorted $S_1 \cup S_2$.
We can easily sort S_0 by replacing each suffix $w[3k..n]$ with the corresponding pair, and then sorting the pairs using radix sort.
Just replace the second element of each pair with its position in the already known sorted sequence of all $S_1 \cup S_2$!
Then we only have to merge two sorted sequences of length $\frac{1}{3}n$ and $\frac{2}{3}n$. This can be done in linear time, assuming that we can compare any two elements in constant time.

So, how to compare any $w[3i..n] \in S_0$ with $w[j..n] \in S_1 \cup S_2$?

- $j = 3k + 1$, then represent $w[3i..n]$ as $(w[3i], w[3i + 1..n])$ and $w[j..n] = w[3k + 1..n]$ as $(w[3k + 1], w[3k + 2..n])$. $w[3i + 1..n]$ can be compared with $w[3k + 2]$ as they both belong to $S_1 \cup S_2$.
- $j = 3k + 2$, then represent $w[3i..n]$ as a triple $(w[3i], w[3i + 1], w[3i + 2..n])$ and $w[j..n] = w[3k + 2..n]$ as $(w[3k + 2], w[3(k + 1), w[3(k + 1) + 1..n])$. $w[3i + 2..n]$ can be compared with $w[3(k + 1) + 1]$ as they both belong to $S_1 \cup S_2$.

Hence any two elements can be compared in constant time.

We are almost done. The only remaining question is how to sort $S_1 \cup S_2$. We know how to sort S_1 and S_2, but we need a stronger observation.

Third trick

The order on all $S_1 \cup S_2$ can be computed by sorting all suffixes of $w' = \mathrm{nr}(1)\mathrm{nr}(4)\mathrm{nr}(7)\ldots\mathrm{nr}(2)\mathrm{nr}(5)\mathrm{nr}(8)\ldots$.

Finally, we get an algorithm with the running time of the form $T(n) = T(\frac{2}{3}n) + \mathcal{O}(n)$, which is $\mathcal{O}(n)$.

Nice! ☺

Recall that our motivation for constructing the suffix array was that using it we can locate an occurrence of any pattern of length m in $\mathcal{O}(m \log n)$ time. Now we are almost ready to speed this up! The suffix array SA alone is not that useful. Usually it is augmented with the inverse suffix array SA^{-1}, where $SA^{-1}[i]$ is the position of $w[i..n]$ in SA, i.e., $SA[SA^{-1}[i]] = i$, and with a longest common prefix structure.

LCP

$\mathrm{lcp}(i, j)$ is the longest common prefix of $w[i..n]$ and $w[j..n]$. $\mathrm{lcp}[i]$ is the longest common prefix of the $(i-1)$-th and i-th suffix in the suffix array, or in other words $\mathrm{lcp}(SA[i-1], SA[i])$, with $\mathrm{lcp}[1]$ not defined.

$W = \texttt{mississippi}$

$SA[1] =$	11	$=$	i
$SA[2] =$	8	$=$	ippi
$SA[3] =$	5	$=$	issippi
$SA[4] =$	2	$=$	ississippi
$SA[5] =$	1	$=$	mississippi
$SA[6] =$	10	$=$	pi
$SA[7] =$	9	$=$	ppi
$SA[8] =$	7	$=$	sippi
$SA[9] =$	4	$=$	sissippi
$SA[10] =$	6	$=$	ssippi
$SA[11] =$	3	$=$	ssissippi

What is $\mathrm{lcp}(8, 2)$?

$W = \texttt{mississippi}$

$SA[1] =$	11	$=$	i			
$SA[2] =$	8	$=$	ippi	lcp[2]	$=$	1
$SA[3] =$	5	$=$	issippi	lcp[3]	$=$	1
$SA[4] =$	2	$=$	ississippi	lcp[4]	$=$	4
$SA[5] =$	1	$=$	mississippi	lcp[5]	$=$	0
$SA[6] =$	10	$=$	pi	lcp[6]	$=$	0
$SA[7] =$	9	$=$	ppi	lcp[7]	$=$	1
$SA[8] =$	7	$=$	sippi	lcp[8]	$=$	0
$SA[9] =$	4	$=$	sissippi	lcp[9]	$=$	2
$SA[10] =$	6	$=$	ssippi	lcp[10]	$=$	1
$SA[11] =$	3	$=$	ssissippi	lcp[11]	$=$	3

What is $\text{lcp}(8, 2)$?

Lemma

$\mathrm{lcp}(i, j)$ is the minimum of all $\mathrm{lcp}[k]$ over $k = SA^{-1}[i] + 1$, $SA^{-1}[i] + 2, \ldots, SA^{-1}[j]$, assuming i is before j in the suffix array.

Proof: see the whiteboard.

So what?

So, assuming that we know $\mathrm{lcp}[i]$, computing any $\mathrm{lcp}(i, j)$ requires just one range minimum query.

RMQ

Given an array $A[1..n]$, preprocess it so that the minimum of any fragment $A[i], A[i+1], \ldots, A[j]$ can be computed efficiently.

During the next lecture, we will see a linear time/space preprocessing allowing answering any query in constant time.

OK, but how to compute the array $\mathrm{lcp}[2], \mathrm{lcp}[3], \ldots, \mathrm{lcp}[n]$?
A trivial solution is to compute each of them separately, resulting in $\mathcal{O}(n^2)$ total complexity. It turns out that a small modification to this naive method improves the time to linear.

Lemma

For any suffix $w[i..n]$, $\mathrm{lcp}[SA^{-1}[i]] - 1 \leq \mathrm{lcp}[SA^{-1}[i+1]]$, assuming $i < n$ and neither $SA^{-1}[i]$ nor $SA^{-1}[i+1]$ are the first element of the suffix array.

Kasai 2006

All lcp can be computed in amortized constant time per entry.

Wait till the tutorial next week for the proof.