Enhanced and Extended Suffix Arrays

Adrian Regenfuß

Technische Universität München

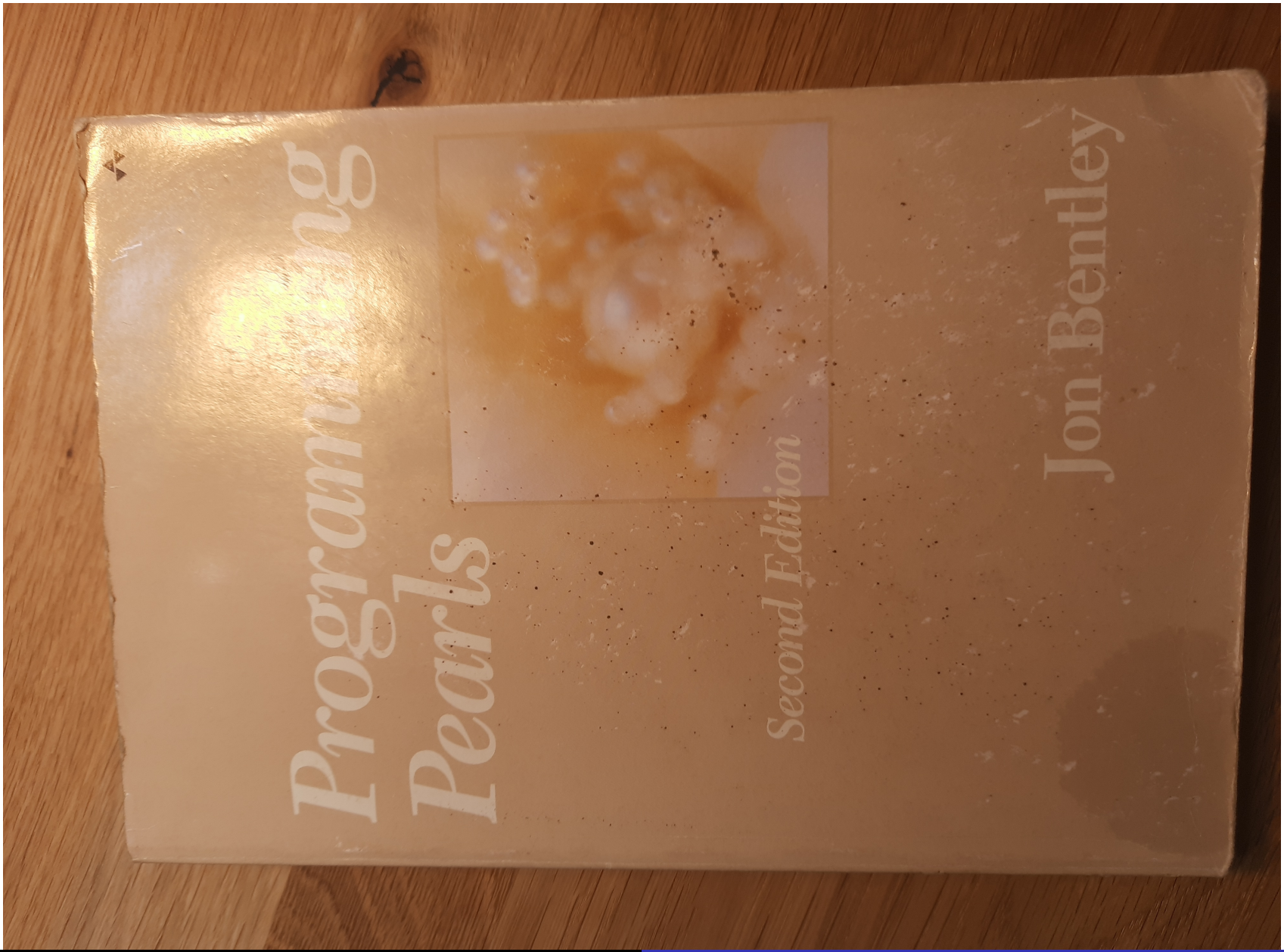
Sommersemester 2020

Übersicht

4 years ago

- Me in Nepal
- No computer with me, only pen, paper and Programming Pearls

My old, damaged copy of Programming Pearls



Thinking a lot about Algorithms

Problems back then

- Thinking about searching, sorting, collision detection, string matching
- Specifically: What is the longest repeated substring of a string?
- Or: what is the longest supermaximal repeat of a string?
- Devising very complex algorithms

Discovering Suffix Arrays

- On the flight back reading the 15th chapter of Programming Pearls
- Describes this exact problem, solved using suffix arrays

Übersicht

Structure of the Presentation

What, How, Why, in that order.

- Description of data structures
- Clarification of terminology
- Description of algorithms
- Discussion of advantages/disadvantages

Übersicht

Contents of an Enhanced Suffix Array

- Suffix Array suftab
- LCP Array lcptab
- Burrows-Wheeler Transformation Array bwttab
- Inverse Suffix Array suftab^{-1}

Suffix Array suftab

Take string over finite alphatbet Σ, e.g. "cag"
Add character "$"(is greater than any other character in Σ)
Sort suffixes
Positions of sorted suffixes in the string are entries of suftab

Example for suftab

For "cag"the suffixes are ("cag$", "ag$", "g$", "$").
The sorted suffixes: ("ag$", "cag$", "g$", "$")
The suffix array then is $[1, 0, 2, 3]$

LCP Array lcptab

LCP Array is the array of the lengths for the prefixes of neighbouring suffixes in the suffix array

More formal for a string S:

$\text{lcptab}[i] = k \Leftrightarrow S[\text{suftab}[i]..\text{suftab}[i] + k] = S[\text{suftab}[i + 1]..\text{suftab}[i - 1] + k]$

Value is 0 for the first suffix (the string itself).

LCP Array Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Burrows-Wheeler Transformation Array bwttab

Informally, bwttab contains the character before the suffix referenced in suftab.

bwttab$[i] = c \Leftrightarrow S[suftab[i] - 1] = c$

Empty (or placeholder character) for the suffix that is also S.

With BWT array, the Enhanced Suffix Array contains the whole string.

LCP Array with BWT Array Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Inverse Suffix Array suftab^{-1}

Only sometimes part of the Enhanced Suffix Array. Is, as the name suggests, the inverse of the suffix array.
$S[\text{suftab}^{-1}[\text{suftab}[i]]] = S[i]$.
Example "cag":
$\text{suftab} = [1, 0, 2, 3]$, $\text{suftab}^{-1} = [0, 1, 3, 2]$

LCP-Intervals

ℓ-interval: Roughly an interval in the LCP array where the LCP value is always $\geq \ell$, and which is surrounded by LCP values $< \ell$.

More formally: The interval $[i..j]$ is an LCP-interval of value ℓ (short $\ell - [i..j]$) iff

- lcptab$[i] < \ell$
- lcptab$[k] \geq \ell$ for all $k : i + 1 \leq k \leq j$
- lcptab$[k] = \ell$ for at least one $k : i + 1 \leq k \leq j$
- lcptab$[j + 1] < \ell$

Example for an ℓ-Interval

$1 - [4..6]$ is an LCP interval, $2 - [4..5]$ is also an interval (note that they are embedded in each other!

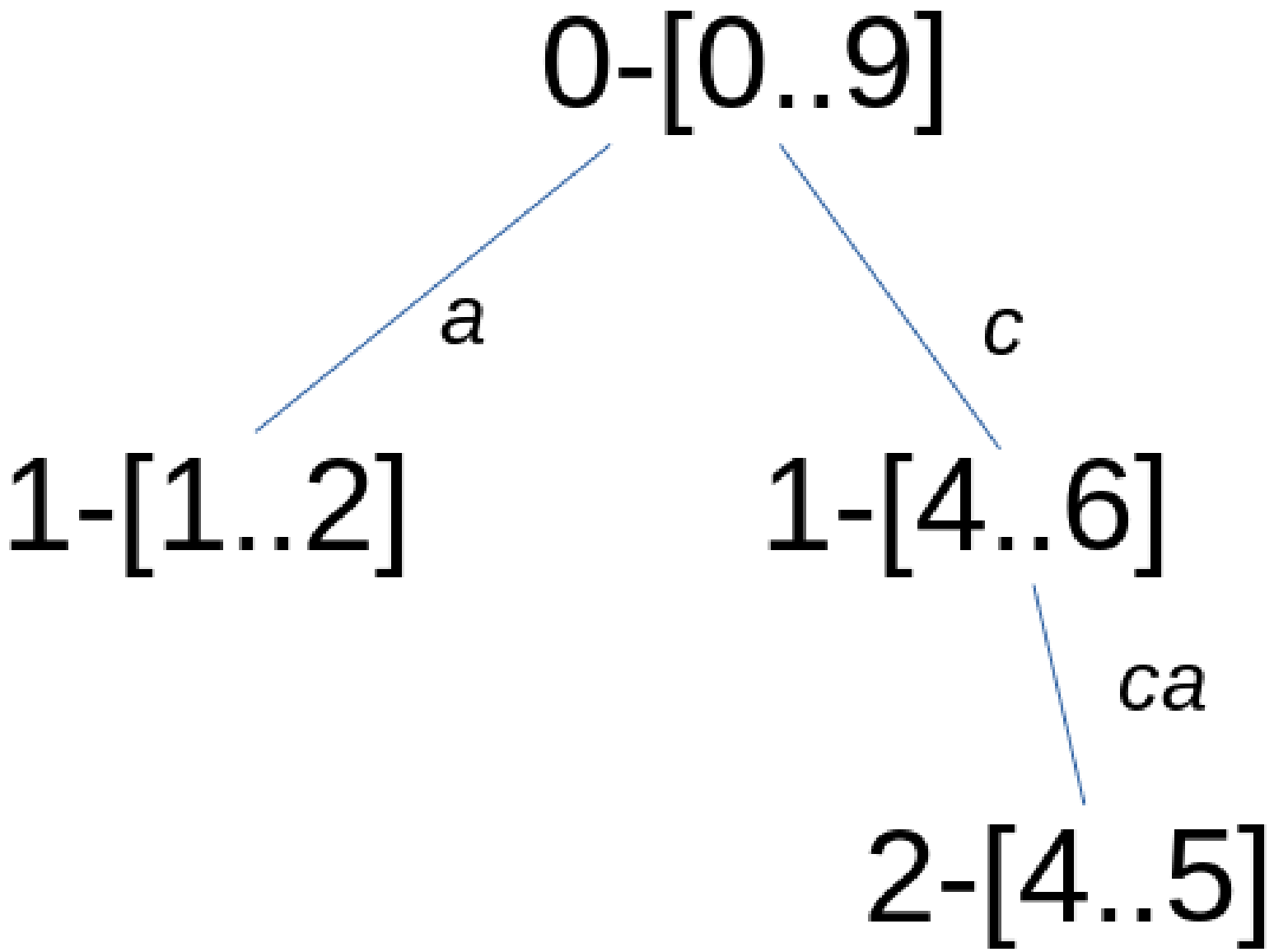
i	suftab	lcptab	SortSuf
0	5	0	acat$
1	1	1	agccacat$
2	7	1	at$
3	4	0	cacat$
4	0	2	cagccacat$
5	6	2	cat$
6	3	1	ccacat$
7	2	0	gccacat$
8	8	0	t$
9	9	0	$

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

LCP-Interval Trees

LCP-intervals are embedded within each other, a 1-interval can contain a 2-interval.
This forms a so-called LCP-interval tree: the root contains the 0-interval, then the 1-intervals in the next level, & so on.
Just the suffix tree without leaves!

LCP-Interval Tree for given string



LCP-Intervals not part of the Enhanced Suffix Array

[1] stresses repeatedly that neither LCP-intervals nor LCP-interval trees are stored, they are merely inferred by LCP array & BWT array.

Übersicht

Enhanced Suffix Array vs. Extended Suffix Array

Terminology a bit jumbled...
We have:

- Suffix Array: Introduced by [9] (but likely used before), only contains suffix array
- Suffix Array with LCP-Array: Very common, also mentioned by [9] and [6]
- Extended Suffix Array: Rarely used, only mentioned by [12], seems to be just suffix array with LCP array.
- Enhanced Suffix Array: Introduced by [1], **focus of this talk**. Contains suffix array, BWT array, LCP array, inverse suffix array

In Short

- There's Suffix Arrays
- There's Suffix Arrays with LCP Information=Extended Suffix Arrays
- There's Enhanced Suffix Arrays (Suffix Arrays with LCP & more)

Übersicht

Many Different String Matching Problems

- Exact String Matching
- Finding Maximal/Supermaximal Repeats
- Finding Longest Common Substring (of ≥ 2 strings)

Many of these can be implemented by traversal of LCP-interval tree.

Exact String Matching (Substring Search)

Find all occurrences of T in S.
Or, more formal: Given strings S and T of lengths n and m, $m \leq n$, return indices $I = \{i_1, \ldots, i_k\}$ so that $\forall i \in I : S[i..i+m] = T$

Using only the Suffix Array

There exists an interval in the suffix array (possibly empty) where T is the prefix of the referenced suffixes.
Use binary search to find the starting position of that interval, then another binary search to find the end position.
Second search uses start position of the interval as left starting point.
Runtime $\mathcal{O}(m + \log n)$ ([9]).

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Using the Suffix Array with LCP information

[9] already describes methods of speeding this binary search up using LCP information:
When the upper and lower bound lie within the same ℓ-interval, comparing T and S[suftab[mid]] is can be started at position $T[\ell]$, S[suftab[mid] $+ \ell$].

Generating the LCP Information

For this, [9] proposes pre-computing LCP information for binary-tree-like pairs $((0, n), (0, \frac{n}{2}), (\frac{n}{4}, \frac{3*n}{4})$ etc.), but there are other possible approaches, using e.g. traversal of LCP-interval trees.

Example

T="cac", low $= 3$, high $= 6$.

	i	suftab	lcptab	bwttab	SortSuf
	0	5	0	c	acat$
	1	1	1	c	agccacat$
	2	7	1	c	at$
low →	3	4	0	c	cacat$
mid →	4	0	2		cagccacat$
	5	6	2	a	cat$
high →	6	3	1	g	ccacat$
	7	2	0	a	gccacat$
	8	8	0	a	t$
	9	9	0	t	$

Gusfield's Super-Accelerant

[6, pp. 152] describes a further accelerant to searching using LCP data.
Note that these accelerants don't improve worst-case performance, which is still $\mathcal{O}(m + \log n)$.

Personal Idea

Personal idea: use Interpolation Search ([11]) for searching the suffix array.
Reasons: runs in $\mathcal{O}(\log \log n)$ average case, and genomic data seems uniformly distributed.

traversing the lcp-interval tree

The lcp-interval tree is very useful many kinds of strings processing algorithms.
It is the suffix tree without leaves.
Is *never actually completely constructed*, but *instead generated on the fly from the lcp array*.

Algorithm for traversing the lcp-interval tree

In essence:

- the "level"we're on in the lcp-interval describes the current position in the tree
- we can save child intervals on a stack
- Whenever we "leave"an interval, we
-
  - Pop it from the stack
  - Process it <– Customizable!
  - Add it to the list of child intervals of the top element of the stack
- Whenever we "enter"a new interval, we push it on the stack

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack: (0,0,$\perp$,[])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

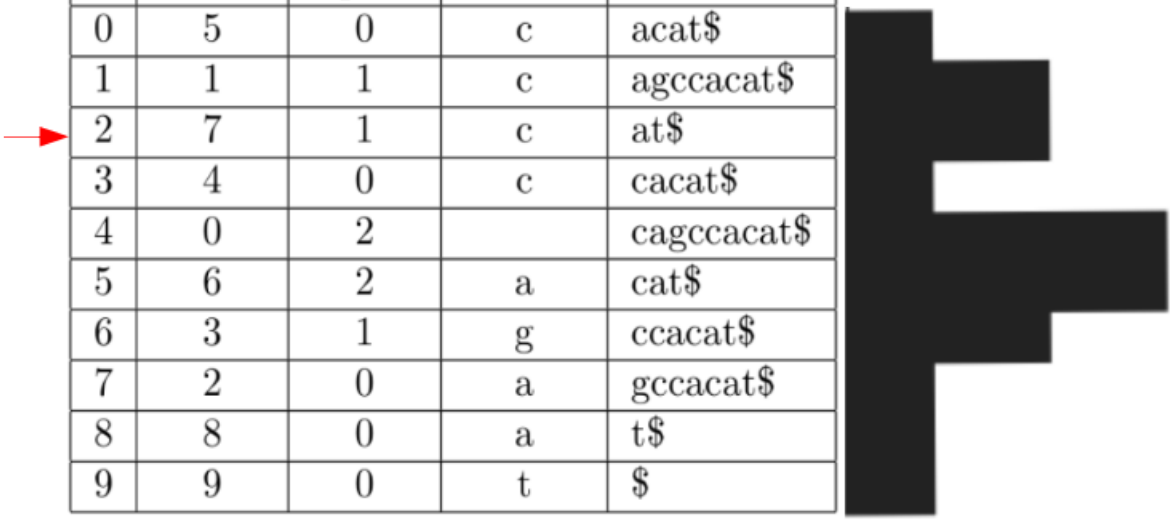
Lastinterval: $\perp$

Stack:

(1,0,$\perp$,[])
(0,0,$\perp$,[])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$



Lastinterval: $\perp$

Stack:

(1,0,$\perp$,[])
(0,0,$\perp$,[])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack:

(1,0,2,[])
(0,0,$\perp$,[])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: (1,0,2,[])

Stack: (0,0,⊥,[])

process(Lastinterval)

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack: (0,0,$\perp$,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack:

(2,3,$\perp$,[])
(0,0,$\perp$,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: ⊥

Stack:

(2,3,⊥,[])
(0,0,⊥,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack:

(2,3,5,[])
(0,0,$\perp$,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: (2,3,5,[])

process(Lastinterval)

Stack: (0,0,⊥,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: ⊥

Stack:

(1,3,6,[(2,3,5,[])])
(0,0,⊥,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: (1,3,6,[(2,3,5,[])

process(Lastinterval)

Stack: (0,0,⊥,[(1,0,2,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack: (0,0,$\perp$,[(1,0,2,[]), (1,3,6,[(2,3,5,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: $\perp$

Stack: (0,0,$\perp$,[(1,0,2,[]), (1,3,6,[(2,3,5,[])])

Example

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Lastinterval: (0,0,**9**,[(1,0,2,[]), (1,3,6,[(2,3,5,[])])]) Stack:

process(Lastinterval)

Processing an Interval

The magic happens in the function process, which could be given as a parameter to the algorithm.
It receives an lcp-interval, containing:

- ℓ-value (length of prefixes)
- Left and right boundary of the interval
- List of children of the interval

Properties of LCP-Interval Tree Traversal

Runtime depends on

- Time for specific data structures (list for children, stack)
- Time complexity for process

In itself, it has a runtime of $\mathcal{O}(n)$.

Maximal Repeats

A maximal repeat is a string ω so that $\omega = S[i_1..j_1] = S[i_2..j_2]$, with $i_1 \neq i_2$, and both $S[i_1 - 1] \neq S[i_2 - 1]$ and $S[j_1 + 1] \neq S[j_2 + 1]$.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Character	x	a	b	c	y	a	b	c	w	a	b	c	y	z

Here, "abc"is a maximal repeat, but "abcy"is a supermaximal repeat.

Computing Maximal Repeats

This algorithm uses the lcp-interval tree traversal, so we start in the function process that receives an lcp-interval.
Runs in $\mathcal{O}(kn + z)$, $k = |\Sigma|$, z number of maximal repeated pairs, $n = |S|$.

Position Sets

Broadly, for every ℓ-interval the algorithm attempts to find the set of all positions in the interval preceded by the character a, $\forall a \in \Sigma$.

$$\mathcal{P} = \begin{cases} \{0 | 0 \in \mathcal{P}_{[i..j]}\} & \text{if } a = \perp \\ \{p \in \mathcal{P}_{[i..j]}\} | p > 0 \text{ and } S[p-1] = a\} & \text{otherwise} \end{cases} \quad (1)$$

Compute Position Sets

Position sets are computed bottom-up, child intervals first.

For all $a, b \in \Sigma$, $a \neq b$, return $((p, p+\ell-1), (p', p'+\ell-1))$, $p < p'$ as maximal repeats. $p \in \mathcal{P}^q_{[i..j]}(a)$, $p' \in \mathcal{P}_{[i'..j']}(b)$.

Combine & Save Position Sets

Combine position set $\mathcal{P}^q_{[i..j]}$ with child position sets $\mathcal{P}_{[i'..j']}$, save them on the stack:

$$\mathcal{P}^{q+1}_{[i..j]}(e) := \mathcal{P}^q_{[i..j]}(e) \cup \mathcal{P}_{[i'..j']}(e) \tag{2}$$

for all $e \in \Sigma \cup \{\perp\}$.

Supermaximal Repeats

A string ω is a supermaximal repeat if it is a maximal repeat that doesn't occur as a proper substring of another maximal repeat.

Finding Supermaximal Repeats

Finding supermaximal repeats is comparatively easy
"A string ω is a supermaximal repeat if[f] [...] there is an ℓ-interval [i..j] such that

- [i..j] is a local maximum in the LCP-table and [i..j] is the ω-interval,
- the characters bwttab[i], bwttab[i+1], ..., bwttab[j] are pairwise distinct."

[1]

Algorithm

Traverse LCP-array linearly, whenever finding a local maximum (and bwttab$[i] \neq$ bwttab$[i+1]$), report it.
Runs in $\mathcal{O}(n)$.

Illustration

i	suftab	lcptab	bwttab	SortSuf
0	5	0	c	acat$
1	1	1	c	agccacat$
2	7	1	c	at$
3	4	0	c	cacat$
4	0	2		cagccacat$
5	6	2	a	cat$
6	3	1	g	ccacat$
7	2	0	a	gccacat$
8	8	0	a	t$
9	9	0	t	$

Finding Maximum Unique Matches

In short, a Maximum Unique Match (MUM) is just a supermaximal repeat in two different strings.
Or, more formally: The MUM u of S_1 and S_2 is a supermaximal repeat of $S_1 \# S_2$ so that $(i_1, j_1) = (i_2, j_2) = u$ and $j_1 < p < i_2$ $(p = |S_1|)$.

Algorithm for finding MUMs

Use the algorithm for finding supermaximal repeats on $S_1\#S_2$.
Filter out non-MUM matches.
Runs in $\mathcal{O}(n)$.
But: This seems very wasteful to me! Every time we want to calculate a MUM, we have to generate the ESA for the whole $S_1\#S_2$! Especially if computing MUMs for many different S_2, this seems horrible! But maybe clever trick for combining ESAs? Haven't seen it yet.

Übersicht

Of the Suffix Array

Obvious $\mathcal{O}(n \log n)$ algorithm: Just sort the suffixes using an $\mathcal{O}(n \log n)$ sorting algorithm (e.g. radix sort, as described in [**?**]).
Possibly use pointers.
Alternative: Use *skew* algorithm described in [7] or the pure-induced sorting algorithm from [10].

Of the Inverse Suffix Array

To be quite honest, I don't really know.
Maybe a trivial algorithm?
[3] describes an algorithm GSACA that also constructs the inverse suffix array, but I haven't had enough time to read & understand it.

Of the LCP Array

Reminder: The LCP-interval tree is the suffix tree without leaves. The suffix tree can be constructed in linear time ([5]), so we can construct first the suffix tree, then remove the leaves which results in the LCP-interval tree, and create the LCP array from that (all three run in $\mathcal{O}(n)$, so in combination they also run in $\mathcal{O}(n)$). Alternatively, one can use the induced sorting algorithm ([4]) to compute both suffix arrays and LCP-arrays.

Of the BWT Array

The BWT array can be constructed in linear time from the suffix array using the naive algorithm.

Übersicht

Motivation

First explain, then convince

- Suffix tree very successful & fast, but uses a lot of memory
- Often used in Genome search/alignment
- Use about 20 bytes per input character ([8])
- Need datastructure with same time complexity as suffix tree, but lower space requirements

Space Requirements

- 10 bytes per input character ($4n$ for the suffix array, n for the LCP array, n for bwttab, $4n$ for suftab^{-1})
- But not all of these have to be in memory all the time! Very algorithm specific (suftab^{-1} only for tandem repeat finding).
- Can be loaded from disk before executing algorithm
- Caveat: works only with strings shorter than 2^{32} bytes, and LCP values < 255 (although there are some workarounds here, described by [2])

Advantages of Enhanced Suffix Arrays

- Construction & algorithms same time complexity as suffix tree for most matching problems
- Better cache coherence (due to linear layout in memory)
- Empirical speedups & easier to implement

Disadvantages of Enhanced Suffix Arrays

Have to be constructed in advance from the string, string can't be changed afterwards (or only with difficulty, [12] describes some methods)
Big alphabets were rarely discussed in the literature, maybe a source of problems? Especially with unicode.
Otherwise seem to be pretty good, it seems.

Implementations

Used in the software Vmatch, see http://www.vmatch.de

Maybe I'll have enough time to implement with interpolation search this summer :-)

Questions

?

Übersicht

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *International Workshop on Algorithms in Bioinformatics*, pages 449–463. Springer, 2002.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

[3] Uwe Baier. *Linear-time suffix sorting*. PhD thesis, Master Thesis, 2015.

[4] Johannes Fischer. Inducing the lcp-array. In *Workshop on Algorithms and Data Structures*, pages 374–385. Springer, 2011.

[5] Robert Giegerich and Stefan Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[6] Dan Gusfield. Algorithms on strings, trees, and sequences. 1997. *Computer Science and Computational Biology. New York: Cambridge University Press*, 1997.

[7] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

[8] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[9] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5): 935–948, 1993.

[10] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 Data Compression Conference*, pages 193–202. IEEE, 2009.

[11] Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log log n search. *Communications of the ACM*, 21 (7):550–553, 1978.

[12] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8(2):241–257, 2010.