

Longest Common Extensions via Fingerprinting

Philip Bille Inge Li Gørtz Jesper Kristensen

Technical University of Denmark
DTU Informatics

LATA, March 9, 2012

Contents

Introduction

- The LCE Problem

Existing Results

- The DIRECTCOMP algorithm

- The SUFFIXNCA and LCPRMQ Algorithms

- Practical results

The FINGERPRINT_k Algorithm

- Data Structure

- Query

- Preprocessing

- Practical Results

- Cache Optimization

Summary

The LCE Problem

LCE value $LCE_s(i, j)$ is the length of the longest common prefix of the two suffixes of a string s starting at index i and j

LCE problem Efficiently query multiple LCE values on a static string s and varying pairs (i, j)

Example:

input: $s = \text{abbababba}$, $(i, j) = (4, 6)$

suffix i of s = ababba

suffix j of s = abba

longest common prefix = ab

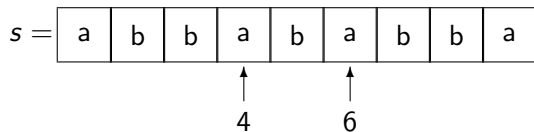
$LCE_s(i, j) = 2$

Existing algorithm: DIRECTCOMP

Input

- ▶ $s = abbababba$
- ▶ $(i, j) = (4, 6)$

The DIRECTCOMP algorithm

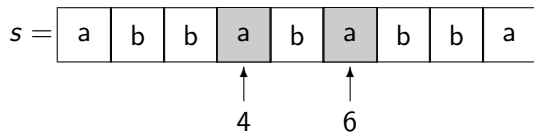


Existing algorithm: DIRECTCOMP

Input

- ▶ $s = abbababba$
- ▶ $(i, j) = (4, 6)$

The DIRECTCOMP algorithm

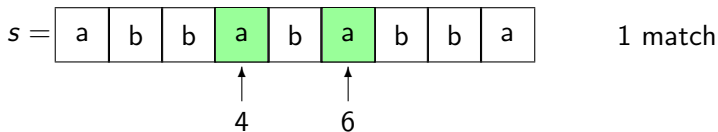


Existing algorithm: DIRECTCOMP

Input

- ▶ $s = \text{abbababba}$
- ▶ $(i, j) = (4, 6)$

The DIRECTCOMP algorithm

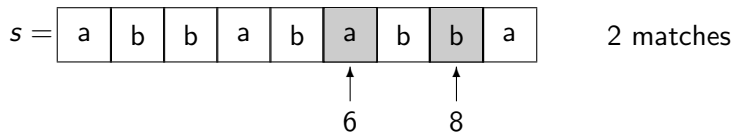


Existing algorithm: DIRECTCOMP

Input

- ▶ $s = \text{abbababba}$
- ▶ $(i, j) = (4, 6)$

The DIRECTCOMP algorithm



Existing Algorithm: DIRECTCOMP

Space	$O(1) + s $
Query	$O(LCE(i, j)) = O(n)$
Average query	$O(1)$

For a string length n and alphabet size σ , the average LCE value over all n^σ strings and n^2 query pairs is $O(1)$.

References

L. Ilie, G. Navarro, and L. Tinta. The longest common extension problem revisited and applications to approximate string searching. *J. Disc. Alg.*, 8(4):418-428, 2010.

Existing Algorithms: SUFFIXNCA and LCPRMQ

Two algorithms with best known bounds:

SUFFIXNCA Nearest common ancestor queries on a suffix tree

LCPRMQ Range minimum queries on a longest common prefix array

Space $O(n)$

Query $O(1)$

Average query $O(1)$

References

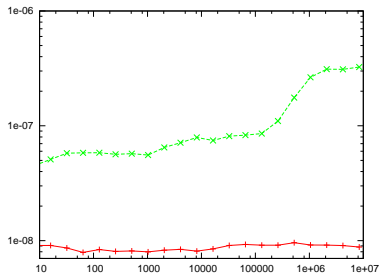
J. Fischer, and V. Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *Proc. 17th CPM*, pages 36-48, 2006.

D. Harel, R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338-355, 1984.

Existing Algorithms: Practical Results

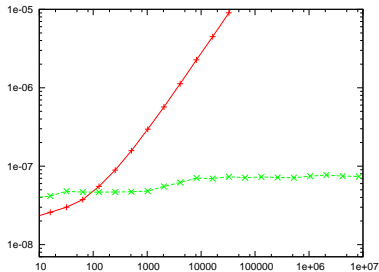
Query times of **DIRECTCOMP** and **LCPRMQ** by string length

Average case



s = random characters
 $\sigma = 10$

Worst case



s = aaaaaa...

The FINGERPRINT_k Algorithm: Data Structure

- ▶ For a string $s[1..n]$, the t -length fingerprints $F_t[1..n]$ are natural numbers, such that $F_t[i] = F_t[j]$ if and only if $s[i..i+t-1] = s[j..j+t-1]$.
- ▶ k levels, $1 \leq k \leq \lceil \log n \rceil$
- ▶ For each level, $\ell = 0..k-1$:
 - ▶ $t_\ell = \Theta(n^{\ell/k})$, $t_0 = 1$
 - ▶ $H_\ell = F_{t_\ell}$

$H_2[i]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[i]$	1	(2)	3	4	1	(2)	5	6	5	1	(2)	3	4	1	(2)	5	6	5	6	3	4	6	5	6	7	8	9
$s = H_0[i]$	a	(b b a)			a	(b b a)			b	a	(b b a)			a	(b b a)			b	a	b	a	a	b	a	b	a	\$
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

Space $O(k \cdot n)$

The FINGERPRINT_k Algorithm: Query

1. As long as $H_\ell[i + v] = H_\ell[j + v]$, increment v by t_ℓ , increment ℓ by one, and repeat this step unless and $\ell = k - 1$.
2. As long as $H_\ell[i + v] = H_\ell[j + v]$, increment v by t_ℓ and repeat this step.
3. Stop and return v when $\ell = 0$, otherwise decrement ℓ by one and go to step two.

$H_2[i + v]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[i + v]$	1	2	3	4	1	2	5	6	5	1	2	3	4	1	2	5	6	5	6	3	4	6	5	6	7	8	9
$H_0[i + v]$	a	b	b	a	a	b	b	a	b	a	b	b	a	a	b	b	a	b	a	b	a	a	b	a	b	a	\$
$i + v$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

$H_2[j + v]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[j + v]$	1	2	3	4	1	2	5	6	5	1	2	3	4	1	2	5	6	5	6	3	4	6	5	6	7	8	9
$H_0[j + v]$	a	b	b	a	a	b	b	a	b	a	b	b	a	a	b	b	a	b	a	b	a	a	b	a	b	a	\$
$j + v$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

$$LCE(3, 12) = 9$$

Query $O(k \cdot n^{1/k})$

Average query $O(1)$

The FINGERPRINT_k Algorithm: Preprocessing

- ▶ For each level ℓ
 - ▶ For each t_ℓ -length substring in lexicographically sorted order
 - ▶ If the current substring $s[SA[i] \dots SA[i] + t_\ell - 1]$ is equal to the previous substring, give it the same fingerprint as the previous substring, otherwise give it a new unused fingerprint. The two substrings are equal when $LCP[i] \geq t_\ell$.

Subst.	$H_\ell[i]$	i
a	1	9
aba	2	4
abb	3	6
abb	3	1
ba	5	8
bab	6	3
bab	6	5
bba	8	7
bba	8	2

$s = \text{abbababba}$

For $t_\ell = 3$:

$H_\ell = [3, 8, 6, 2, 6, 3, 8, 5, 1]$

Preprocessing $O(k \cdot n + \text{sort}(n, \sigma))$

The FINGERPRINT_k Algorithm

	$1 \leq k \leq \lceil \log n \rceil$
Space	$O(k \cdot n)$
Query	$O(k \cdot n^{1/k})$
Average query	$O(1)$

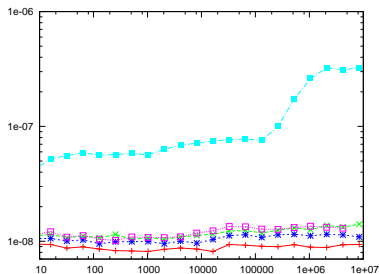
	$k = 1$	$k = 2$	$k = \lceil \log n \rceil$
Space	$O(n)$	$O(n)$	$O(n \log n)$
Query	$O(n)$	$O(\sqrt{n})$	$O(\log n)$
Average query	$O(1)$	$O(1)$	$O(1)$

Space for FINGERPRINT_k is the same as for LCPRMQ when $k = 6$.

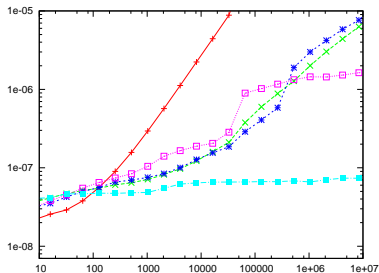
Practical Results

Query times of **DIRECTCOMP**, **FINGERPRINT₂**, **FINGERPRINT₃**, **FINGERPRINT_{⌈log n⌋}** and **LCPRMQ** by string length

Average case



Worst case



Cache Optimization of FINGERPRINT_k

► Original:

- Data structure: $H_\ell[i] = F_{t_\ell}[i]$
- Size: $|H_\ell| = n$
- I/O: $O(k \cdot n^{1/k})$

► Cache optimized:

- Data structure:

$$H_\ell[((i-1) \bmod t_\ell) \cdot \lceil n/t_\ell \rceil + \lfloor (i-1)/t_\ell \rfloor + 1] = F_{t_\ell}[i]$$
- Size: $|H_\ell| = n + t_\ell$
- I/O: $O\left(k \cdot \left(\frac{n^{1/k}}{B} + 1\right)\right)$
 - Best when k is small $\implies n^{1/k}$ is large.

$H_2[i+v]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[i+v]$	1	2	3	4	1	2	5	6	5	1	2	3	4	1	2	5	6	5	6	3	4	6	5	6	7	8	9
$H_0[i+v]$	a	b	b	a	a	b	b	a	b	a	b	b	a	a	b	b	a	b	a	b	a	a	b	a	b	a	\$
$i+v$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

$H_2[j+v]$	1	2	3	4	5	6	7	8	9	1	2	3	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$H_1[j+v]$	1	2	3	4	1	2	5	6	5	1	2	3	4	1	2	5	6	5	6	3	4	6	5	6	7	8	9
$H_0[j+v]$	a	b	b	a	a	b	b	a	b	a	b	b	a	a	b	b	a	b	a	b	a	a	b	a	b	a	\$
$j+v$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27

Cache Optimization, Practical Results

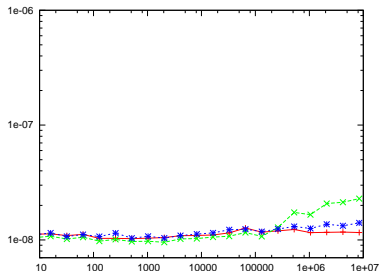
Is I/O optimization good in practice?

- ▶ Pro: better cache efficiency
 - ▶ Best for small k , no change for $k = \lceil \log n \rceil$
- ▶ Con: Calculating memory addresses is more complicated
 - ▶ $((i - 1) \bmod t_\ell) \cdot \lceil n/t_\ell \rceil + \lfloor (i - 1)/t_\ell \rfloor + 1$ vs. i

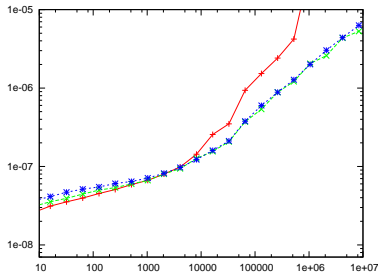
Cache Optimization, Practical Results

Query times of FINGERPRINT₂ without cache optimization and with cache optimization using shift operations vs. multiplication, division and modulo

Average case



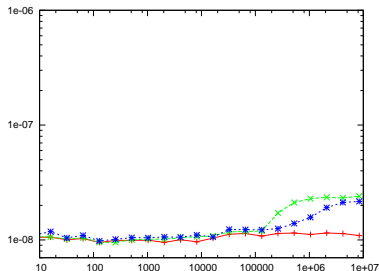
Worst case



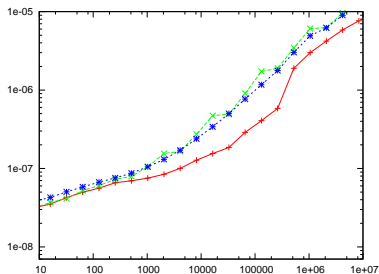
Cache Optimization, Practical Results

Query times of FINGERPRINT₃ **without cache optimization** and with cache optimization using **shift operations** vs. **multiplication, division and modulo**

Average case



Worst case



Summary

	DIRECT- COMP	LCPRMQ / SUFFIXNCA	FINGERPRINT _k
Space	$O(1)$	$O(n)$	$O(k \cdot n)$
Query	$O(n)$	$O(1)$	$O(k \cdot n^{1/k})$
Average query	$O(1)$	$O(1)$	$O(1)$
	fast	slow	fast
Query I/O	$O(\frac{n}{B})$	$O(1)$	$O\left(k \cdot \left(\frac{n^{1/k}}{B} + 1\right)\right)$
Code complexity	very simple	complex	simple

- ▶ Cache optimization of FINGERPRINT_k improves query times at $k = 2$ and worsens query times at $k \geq 3$
- ▶ Space for FINGERPRINT_k is the same as for LCPRMQ when $k = 6$.