Algorithms for Longest Common Extensions

Jesper Kristensen

DTU Informatics
Technical University of Denmark

August 31, 2011

Contents

The DIRECTCOMP algorithm

Input

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

The DIRECTCOMP algorithm

$$s = \boxed{a} \;\boxed{b}\; \boxed{b}\; \boxed{a}\; \boxed{b}\; \boxed{a}\; \boxed{b}\; \boxed{b}\; \boxed{a}$$

The DIRECTCOMP algorithm

Input

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

The DIRECTCOMP algorithm

$$s = \boxed{a \mid b \mid b \mid a \mid b \mid a \mid b \mid b \mid a}$$

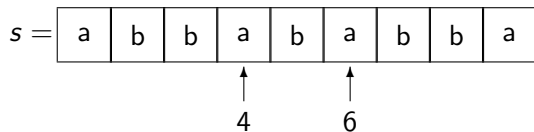$$\uparrow \qquad \uparrow$$
$$4 \qquad 6$$

The DIRECTCOMP algorithm

Input

- $s = abbababba$
- $(i, j) = (4, 6)$
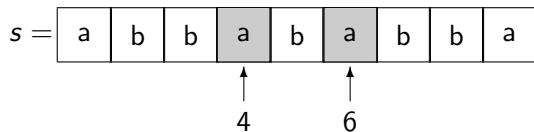
The DIRECTCOMP algorithm

1 match

The DIRECTCOMP algorithm

Input

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

The DIRECTCOMP algorithm

1 match

The DIRECTCOMP algorithm

Input

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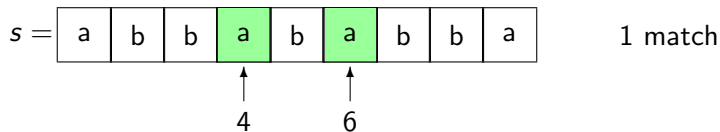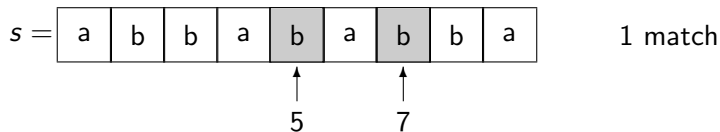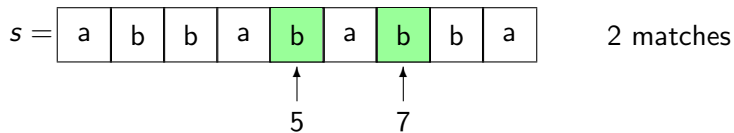- $(i, j) = (4, 6)$

The DIRECTCOMP algorithm

2 matches

The DIRECTCOMP algorithm

Input

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

The DIRECTCOMP algorithm

$$s = \boxed{a} \ \boxed{b} \ \boxed{b} \ \boxed{a} \ \boxed{b} \ \boxed{a} \ \boxed{b} \ \boxed{b} \ \boxed{a}$$

6   8

2 matches

The DIRECTCOMP algorithm

Input

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

The DIRECTCOMP algorithm

2 matches

Result
$LCE_s(4, 6) = 2$

The LCE Problem

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

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

Existing Algorithm: DirectComp

Preprocessing	None
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

Preprocessing	$O(sort(n, \sigma))$
Space	$O(n)$
Query	$O(1)$
Average query	$O(1)$

‾‾‾‾‾‾‾‾‾‾‾‾‾‾

References

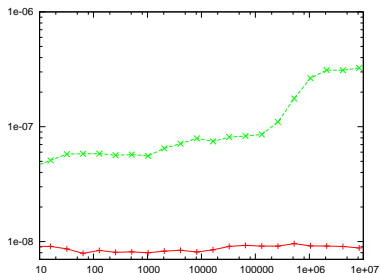J. Fischer, and V. Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *Proc. 17th CPM*, pages 36-48, 2006.

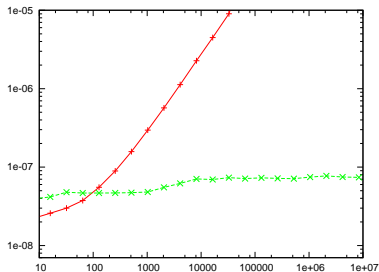D. Harel, R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.*, 13(2):338-355, 1984.

Existing Algorithms: Practical Results

Query times of DIRECTCOMP and LCPRMQ by string length

Average case

Worst case

The $\textsc{Fingerprint}_k$ Algorithm: Data Structure

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

$H_2[i]$	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 $\quad O(k \cdot n^{1/k})$

Average query $\quad O(1)$

The FINGERPRINT$_k$ Algorithm

$$1 \le k \le \lceil \log n \rceil$$

Preprocessing	$O(k \cdot n + \mathit{sort}(n, \sigma))$
Space	$O(k \cdot n)$
Query	$O(k \cdot n^{1/k})$
Average query	$O(1)$

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

Practical Results

Query times of DirectComp, Fingerprint$_2$, Fingerprint$_3$, Fingerprint$_{\lceil \log n \rceil}$ and LcpRmq by string length

Average case
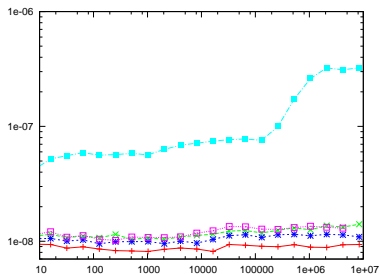
Worst case
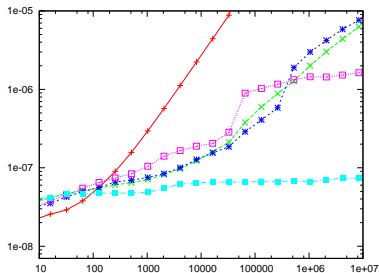
Practical Results

Query times of DIRECTCOMP, FINGERPRINT$_2$, FINGERPRINT$_3$, FINGERPRINT$_{\lceil \log n \rceil}$ and LCPRMQ by string length

Medium LCE size
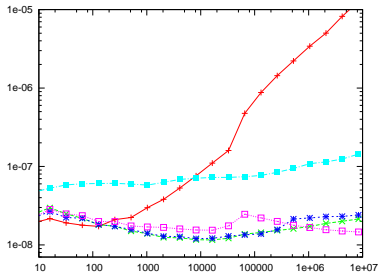
Cache Optimization of Fingerprint_k

- 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:
    $H_\ell[((i-1) \mod t_\ell) \cdot \lceil n/t_\ell \rceil + \lfloor (i-1)/t_\ell \rfloor + 1] = F_{t_\ell}[i]$
  - 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.
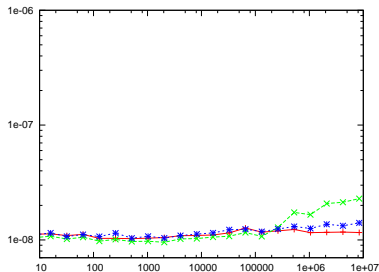
Cache Optimization, Practical Results

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
  - ▶ $((i - 1) \mod t_\ell) \cdot \lceil n/t_\ell \rceil + \lfloor (i - 1)/t_\ell \rfloor + 1$ vs. i
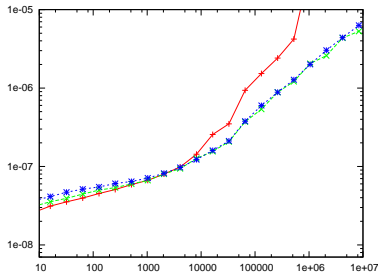
Cache Optimization, Practical Results

Query times of $\textsc{Fingerprint}_2$ 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 $\textsc{Fingerprint}_3$ without cache optimization and
with cache optimization using shift operations vs. multiplication,
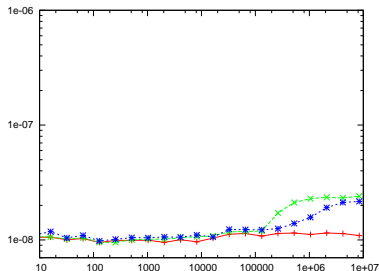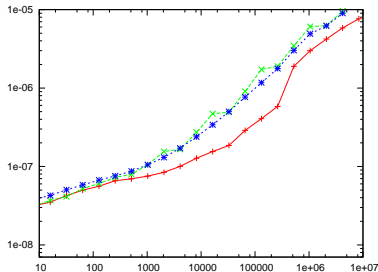division and modulo

Average case

Worst case

Summary

	DIRECT-COMP	LCPRMQ / SUFFIXNCA	FINGERPRINT$_k$
Preprocessing	$O(1)$	$O(sort(n, \sigma))$	$O(k \cdot n + sort(n, \sigma))$
Space	$O(1)$	$O(n)$	$O(k \cdot n)$
Query	$O(n)$	$O(1)$	$O(k \cdot n^{1/k})$
Average query	$O(1)$	$O(1)$	$O(1)$
Query I/O	$O\left(\frac{n}{B}\right)$	$O(1)$	$O\left(k \cdot \left(\frac{n^{1/k}}{B} + 1\right)\right)$

- In practice, the FINGERPRINT$_k$ algorithm is...
  - ...almost as good as DIRECTCOMP and significantly better than LCPRMQ in average case
  - ...significantly better than DIRECTCOMP but worse than LCPRMQ in worst case
- Cache optimization of FINGERPRINT$_k$ improves query times at $k = 2$ and worsens query times at $k \geq 3$

Kommentarer og rettelser til rapporten

- Hvor kommer $r = 0.73n^{0.42}$ fra?

$$r = 0{,}7243n^{0{,}4271}$$