On the computation of longest previous non-overlapping factors

Enno Ohlebusch and Pascal Weber

Institute of Theoretical Computer Science
University Ulm

Segovia, 9. October 2019

Motivation: f-factorization

- Variant of the Lempel-Ziv factorization
- Difference: factors must be non overlapping
- Known algorithms compute the LPnF-array

$$\text{LPnF}[i] = \mathit{max}\left\{ \ell \;\middle|\; \begin{array}{l} 0 \le \ell \le n-i \\ S[i..i+\ell-1] \text{ is a substring of } S[0..i-1] \end{array} \right.$$

$$\text{LPF}[i] = \mathit{max}\left\{ \ell \;\middle|\; \begin{array}{l} 0 \le \ell \le n-i \\ S[i..i+\ell-1] \text{ is a substring of } S[0..i-1+\ell-1] \end{array} \right.$$

$$\text{prevOcc}[i] = \text{index of the previous occurence of the factor at position } i$$

Motivation: f-factorization

- Variant of the Lempel-Ziv factorization
- Difference: factors must be non overlapping
- Known algorithms compute the LPnF-array

$$\text{LPnF}[\mathrm{i}] = \mathit{max}\left\{\ell \;\middle|\; \begin{array}{l} 0 \leq \ell \leq \mathrm{n}-\mathrm{i} \\ S[\mathrm{i}..\mathrm{i}+\ell-1] \text{ is a substring of } S[0..\mathrm{i}-1] \end{array}\right.$$

$$\text{LPF}[\mathrm{i}] = \mathit{max}\left\{\ell \;\middle|\; \begin{array}{l} 0 \leq \ell \leq \mathrm{n}-\mathrm{i} \\ S[\mathrm{i}..\mathrm{i}+\ell-1] \text{ is a substring of } S[0..\mathrm{i}-1+\ell-1] \end{array}\right.$$

prevOcc$[\mathrm{i}]$ = index of the previous occurence of the factor at position i

Computing LPnF from LPF

- Right-to-left scan of the LPF-array and its prevOcc-array
- Two different cases:
  1. Factor at position i and its previous occurrence at position $j = \text{prevOcc}[i]$ do not overlap $\rightarrow \text{LPnF}[i] = \text{LPF}[i]$
  2. Otherwise, the current maximum is $\ell = j - i$ but there exist a longer factor if $\text{LPF}[j] > \ell$ at position $k = \text{prevOcc}[j]$
- The second case can be repeated until exhaustion

Computing LPnF from LPF

- Right-to-left scan of the LPF-array and its prevOcc-array
- Two different cases:
  1. Factor at position i and its previous occurrence at position $\mathrm{j} =$ prevOcc$[\mathrm{i}]$ do not overlap $\rightarrow$ LPnF$[\mathrm{i}] =$ LPF$[\mathrm{i}]$
  2. Otherwise, the current maximum is $\ell = \mathrm{j} - \mathrm{i}$ but there exist a longer factor if LPF$[\mathrm{j}] > \ell$ at position $\mathrm{k} =$ prevOcc$[\mathrm{j}]$
- The second case can be repeated until exhaustion

Computing LPnF from LPF

- Right-to-left scan of the LPF-array and its prevOcc-array
- Two different cases:
  1. Factor at position i and its previous occurrence at position $\mathrm{j} =$ prevOcc[i] do not overlap
     $\rightarrow$ LPnF[i] $=$ LPF[i]
  2. Otherwise, the current maximum is $\ell = \mathrm{j} - \mathrm{i}$
     but there exist a longer factor if LPF[j] $> \ell$
     at position $\mathrm{k} =$ prevOcc[j]
- The second case can be repeated until exhaustion

Computing LPnF from LPF

- Right-to-left scan of the LPF-array and its prevOcc-array
- Two different cases:
  1. Factor at position i and its previous occurrence at position $\mathrm{j} =$ prevOcc$[\mathrm{i}]$ do not overlap $\rightarrow$ LPnF$[\mathrm{i}] =$ LPF$[\mathrm{i}]$
  2. Otherwise, the current maximum is $\ell = \mathrm{j} - \mathrm{i}$ but there exist a longer factor if LPF$[\mathrm{j}] > \ell$ at position $\mathrm{k} =$ prevOcc$[\mathrm{j}]$
- The second case can be repeated until exhaustion

Example

i	0	1	2	3	4	5	6	7	8	9
S[i]	a	a	a	a	a	a	a	a	a	a
LPF[i]	0	9	8	7	6	5	4	3	2	1
(rm) prevOcc[i]	$\perp$	0	1	2	3	4	5	6	7	8
LPnF[i]										

Example

i	0	1	2	3	4	5	6	7	8	9
S[i]	a	a	a	a	a	a	a	a	a	a
LPF[i]	0	9	8	7	6	5	4	3	2	1
(rm) prevOcc[i]	$\perp$	0	1	2	3	4	5	6	7	8
LPnF[i]										1

Example

i	0	1	2	3	4	5	6	7	8	9
S[i]	a	a	a	a	a	a	a	a	a	a
LPF[i]	0	9	8	7	6	5	4	3	2	1
(rm) prevOcc[i]	$\perp$	0	1	2	3	4	5	6	7	8
LPnF[i]									1	1

Example

i	0	1	2	3	4	5	6	7	8	9
S[i]	a	a	a	a	a	a	a	a	a	a
LPF[i]	0	9	8	7	6	5	4	3	2	1
(rm) prevOcc[i]	$\perp$	0	1	2	3	4	5	6	7	8
LPnF[i]									2	1

Example

i	0	1	2	3	4	5	6	7	8	9
S[i]	a	a	a	a	a	a	a	a	a	a
LPF[i]	0	9	8	7	6	5	4	3	2	1
(rm) prevOcc[i]	$\perp$	0	1	2	3	4	5	6	7	8
(lm) prevOcc[i]	$\perp$	0	0	0	0	0	0	0	0	0
LPnF[i]	0	1	2	3	4	5	4	3	2	1

- Lemma: When using the leftmost prevOcc-array the second case can occur at most once

Compute leftmost prevOcc-array

- Linear-time algorithms which computes the LPF-array with leftmost prevOcc-array are usually slow
- Faster algorithms don't produce a leftmost prevOcc-array
- The leftmost prevOcc-array can easily be obtained:
  - If LPF[i] $> 0,\ \mathrm{j} =$ prevOcc[i], and LPF[j] $\geq$ LPF[i] then k = prevOcc[j] is also starting position of the factor at position i
  - Repeat until the leftmost starting point is found
  - Algorithm not linear in worst-case

Compute leftmost prevOcc-array

- Linear-time algorithms which computes the LPF-array with leftmost prevOcc-array are usually slow
- Faster algorithms don't produce a leftmost prevOcc-array
- The leftmost prevOcc-array can easily be obtained:
  - If LPF$[\mathrm{i}] > 0,\ \mathrm{j} =$ prevOcc$[\mathrm{i}]$, and LPF$[\mathrm{j}] \geq$ LPF$[\mathrm{i}]$ then k = prevOcc$[\mathrm{j}]$ is also starting position of the factor at position i
  - Repeat until the leftmost starting point is found
  - Algorithm not linear in worst-case

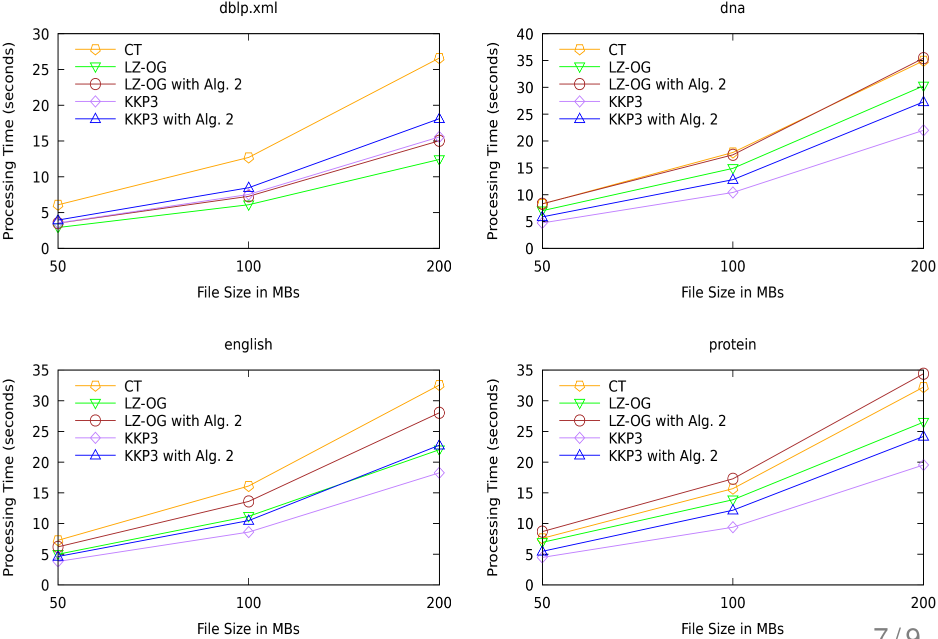
Direct computation of the f-factorization

- Using
  - the Suffix array
  - the Wavelet tree of the Burrows-Wheeler transform
  - Range maximum queries
- Calculate the factors by backward search on the reversed string
- Run-time $O(n \log |\Sigma|)$
- Implementation uses the `sdsl-lite` library and is publicly available

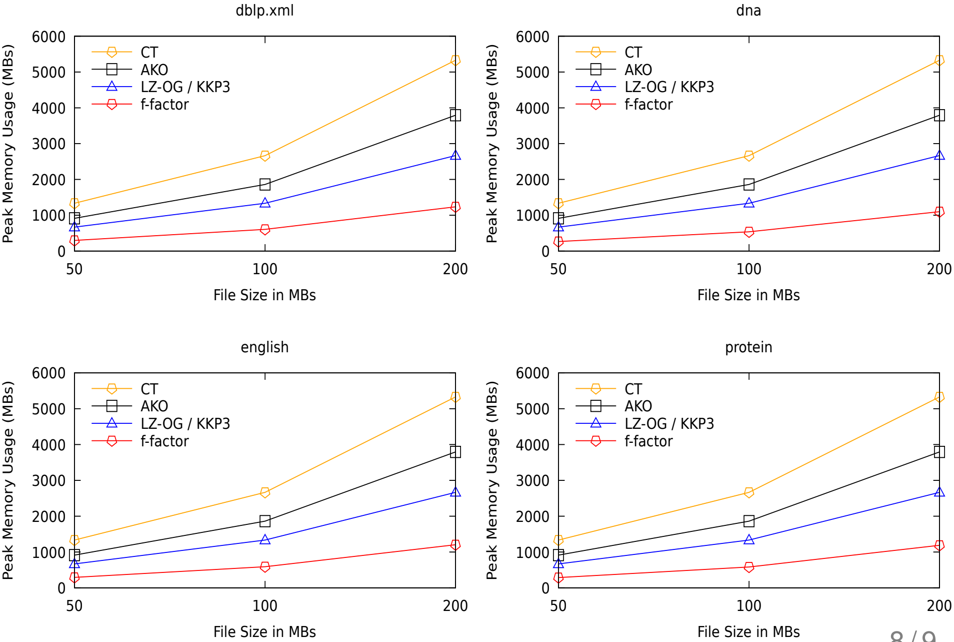
Experimental results

- Existing LPnF construction algorithm (`CT`)
- Linear-time algorithm computing the LPF-array and the leftmost prevOcc-array (`AKO`)
- Two of the fastest algorithms computing the LPF-array and not necessarily a leftmost prevOcc-array (`LZ-OG` & `KKP3`)
  - with and without computing leftmost prevOcc-array before (`Alg.2`)
- Test data files (`dblp.xml`, `dna`, `english`, and `proteins`) are originated from the Pizza & Chili corpus

Experimental results - Processing Time



Experimental results - Peak Memory Usage



Conclusion

- `KKP3` in combination with the featured algorithm outperforms the existing algorithm in terms of run-time and memory usage
- On repetitive strings the leftmost prevOcc-array (`Alg.2`) should be calculated before
- The direct computation of the f-factorization is by an order of magnitude slower but uses much less memory

Conclusion

- `KKP3` in combination with the featured algorithm outperforms the existing algorithm in terms of run-time and memory usage
- On repetitive strings the leftmost prevOcc-array (`Alg.2`) should be calculated before
- The direct computation of the f-factorization is by an order of magnitude slower but uses much less memory

Thank you!