LZD Factorization: Simple and Practical Online Grammar Compression with Variable-to-Fixed Encoding

Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, Masayuki Takeda
Kyushu University

Overview

- We propose a novel online grammar compression algorithm called LZ Double, which is based on LZ78
- With slight modification to LZ78, LZ Double achieves better compression ratio
- Moreover, compared to previous online grammar compression algorithms,
  - Compression ratios of LZ Double are better
  - Compression speed of LZ Double is competitive

LZ78 Factorization [Ziv and Lempel 1978]

Definition

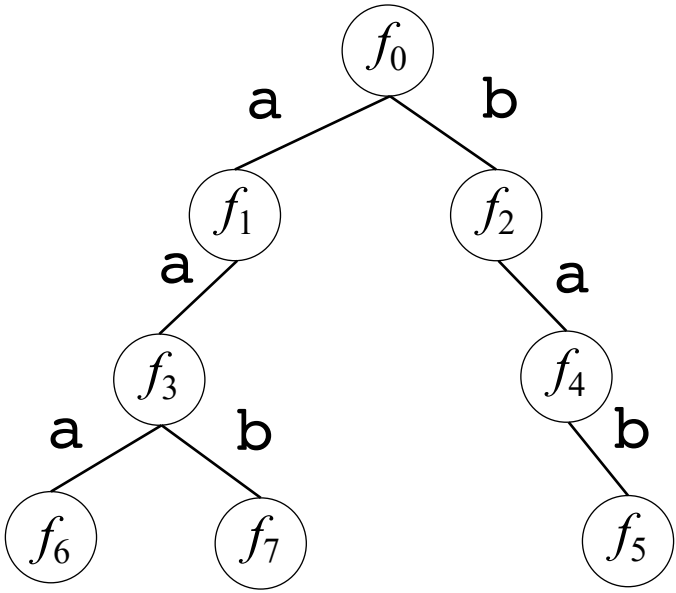
Let $f_0 = \varepsilon$, LZ78 factorization of a string T is $f_0 f_1 \dots f_m$ such that, for f_j starting at $i = 1 + |f_0 f_1 \dots f_{j-1}|$, f_j is $f_k\, c$ $(0 \leqq k < j \leqq m)$, where

- $f_k \in \{f_0 f_1 \dots f_{j-1}\}$ is the longest previous factor (LPF) that occurs at i
- c is a following character $T[i + |f_k|]$

$T =$ a|b|aa|ba|bab|aaa|aab|··

f_1 f_2 f_3 f_4 f_5 f_6 f_7

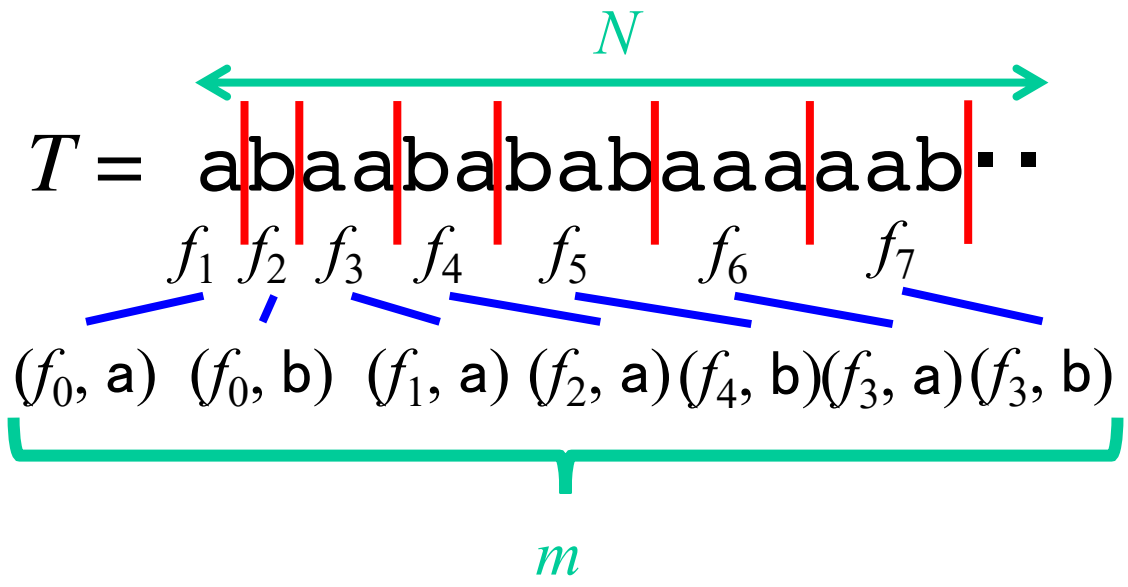
$(f_0, \mathtt{a})$ $(f_0, \mathtt{b})$ $(f_1, \mathtt{a})$ $(f_2, \mathtt{a})$ $(f_4, \mathtt{b})$ $(f_3, \mathtt{a})$ $(f_3, \mathtt{b})$

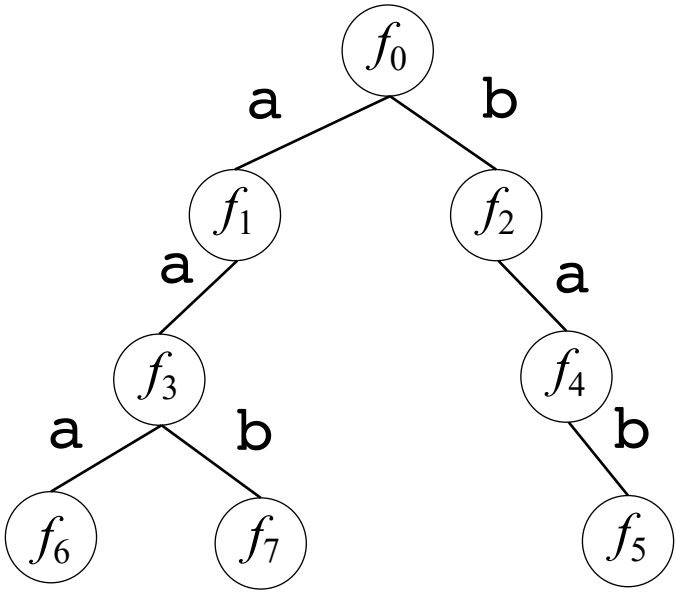


LZ78 Factorization [Ziv and Lempel 1978]

Theorem

For a string T of length N over an alphabet of size σ, LZ78 factorization can be computed in online manner in $O(N \log \sigma)$ time and $O(m)$ space





LZ78 Factorization [Ziv and Lempel 1978]

- **Good:** simple, and easy to implement
- **Bad:** low compression ratio
  - New factor can be at most 1 character longer than the longest previous factors

LZ78: $f_j = f_k\, c$

we modify this

Idea of LZ Double

In the LZD factorization, the new factor is the concatenation of two previous longest factors

- **Good:** still simple, easy to implement, AND better compression ratio
  - New factor can be twice as long as the longest previous factors

LZ78: $f_j = f_k\, c$

LZ Double: $f_j = f_{l(j)} f_{r(j)}$

concatenation of two LPFs

Formal Definition of LZ Double Factorization

Definition

Let $f_0 = \varepsilon$, LZD factorization of T is $f_0 f_1 \ldots f_m$ such that, for $i = 1 + |f_0 f_1 \ldots f_{j-1}|$, f_j is $f_{l(j)} f_{r(j)}$, where

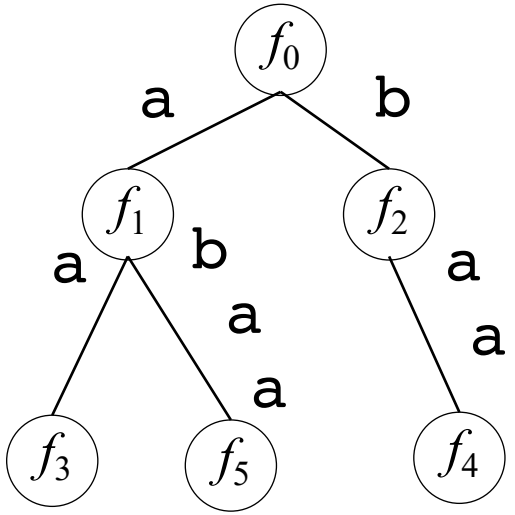
$f_{l(j)} \in \{f_1 \ldots f_{j-1}\} \cup \Sigma$ is LPF that occurs at i

$f_{r(i)} \in \{f_0 f_1 \ldots f_{j-1}\}$ is LPF that occurs at $i + |f_{l(i)}|$

$T =$ a | b | aa | baa | abaa | ababaabb···

f_1 f_2 f_3 f_4 f_5

(a, f_0) (b, f_0) (f_1, f_1) (f_2, f_3) (f_1, f_4)



LZ Double Factorization

Definition

Let $f_0 = \varepsilon$, LZD factorization of T is $f_0 f_1 \ldots f_m$ such that, for $i = 1 + |f_0 f_1 \ldots f_{j-1}|$, f_j is $f_{l(j)} f_{r(j)}$, where

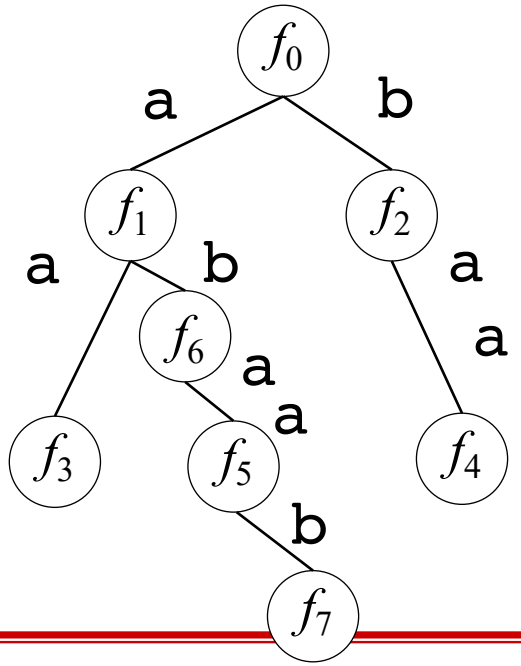
$f_{l(j)} \in \{f_1 \ldots f_{j-1}\} \cup \Sigma$ is LPF that occurs at i

$f_{r(i)} = \{f_0 f_1 \ldots f_{j-1}\}$ is LPF that occurs at $i + |f_{l(i)}|$

$T =$ a | b | a a | b a a | a b a a | a b | a b a a b | b ···

f_1 f_2 f_3 f_4 f_5 f_6 f_7

(a, f_0) (b, f_0) (f_1, f_1) (f_2, f_3) (f_1, f_4) (f_1, f_2) (f_5, f_2)



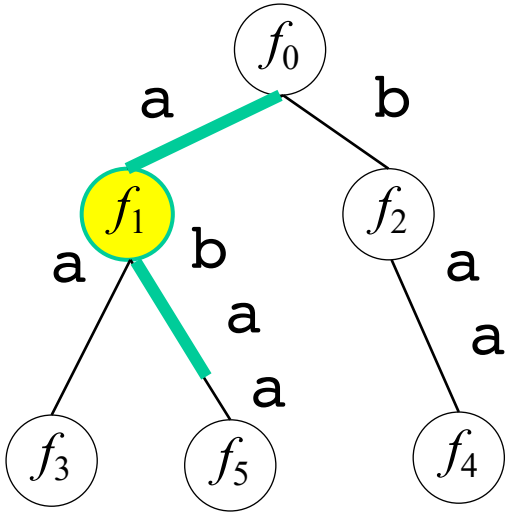
Naive LZD Factorization Algorithm

- Stores all previous factors in a Patricia tree, and marks their corresponding nodes
- Traverses the tree with suffix $T[i..N]$, and then the deepest marked node in this path is LPF
- Inserts a new factor, and marks corresponding node

$T =$ abaababaabaababaababb···

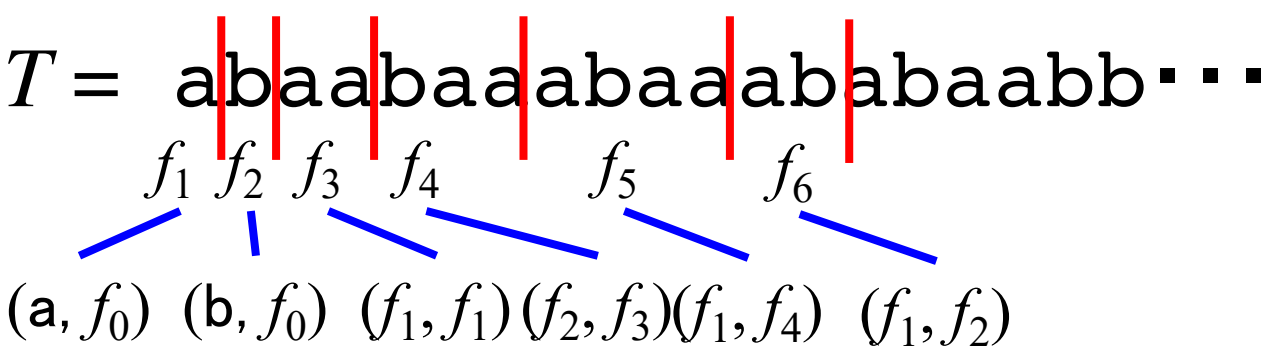
f_1 f_2 f_3 f_4 f_5

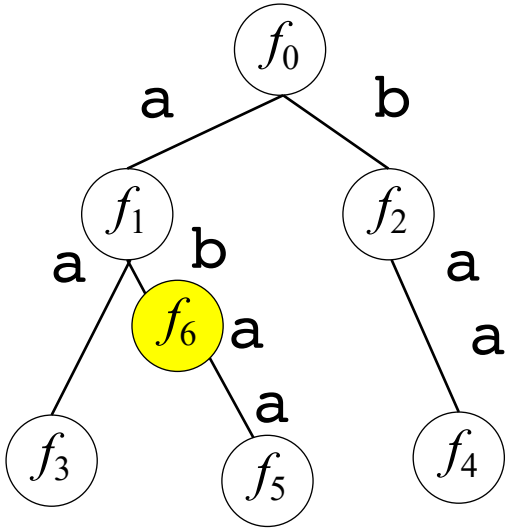
(a, f_0) (b, f_0) (f_1, f_1) (f_2, f_3) (f_1, f_4)



Naive LZD Factorization Algorithm

- Stores all previous factors in a Patricia tree, and marks their corresponding nodes
- Traverses the tree with suffix $T[i..N]$, and then the deepest marked node in this path is LPF
- Inserts a new factor, and marks corresponding node

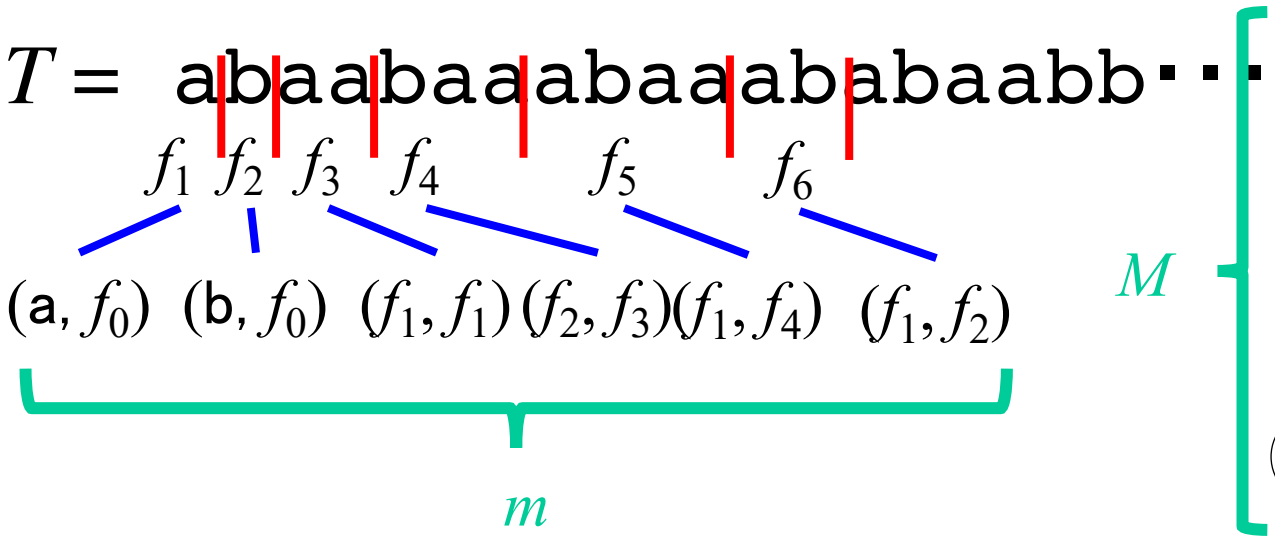


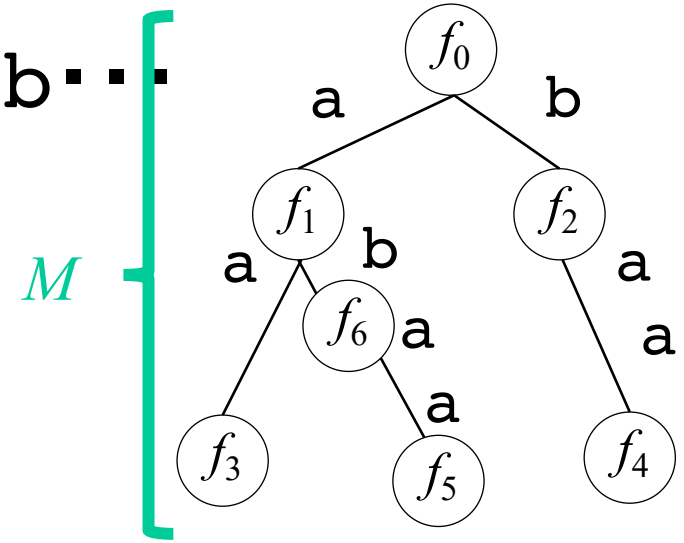


Naive LZD Factorization Algorithm

Theorem

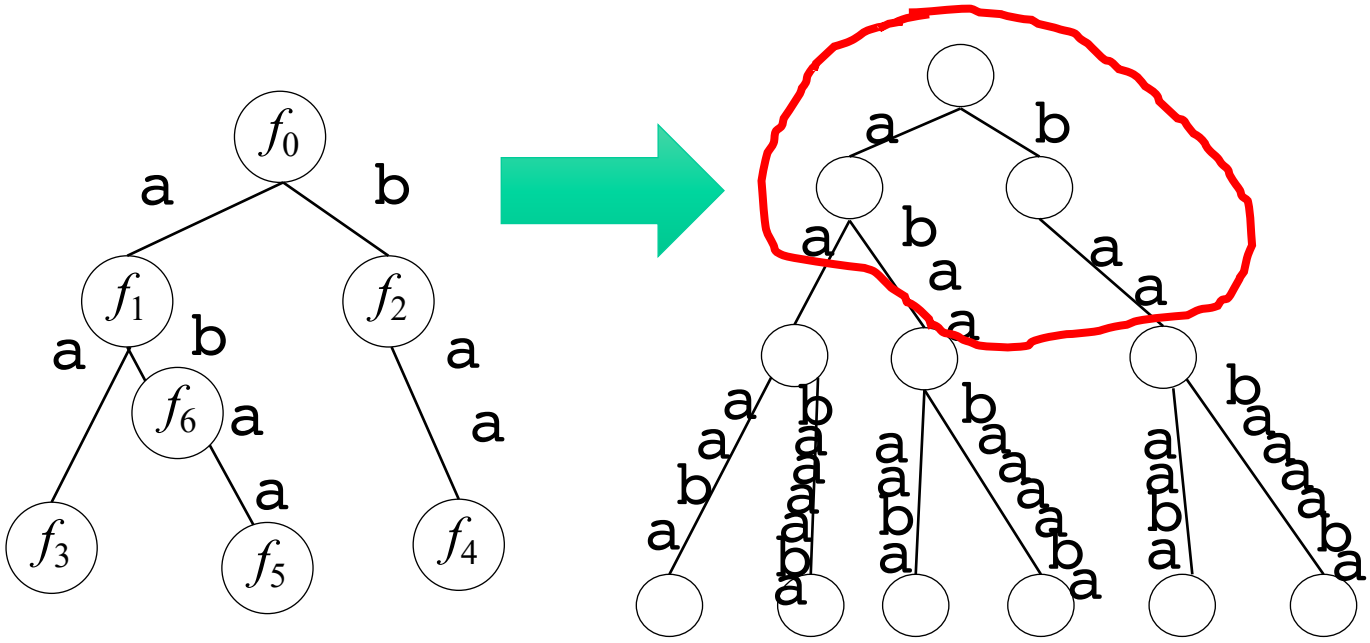
For a string T of length N over an alphabet of size σ, Let m be the number of factors, and M be the maximum length of factors, LZD factorization can be computed in $O(m\,(M + \min(m, M) \log \sigma))$ time and $O(m)$ working space





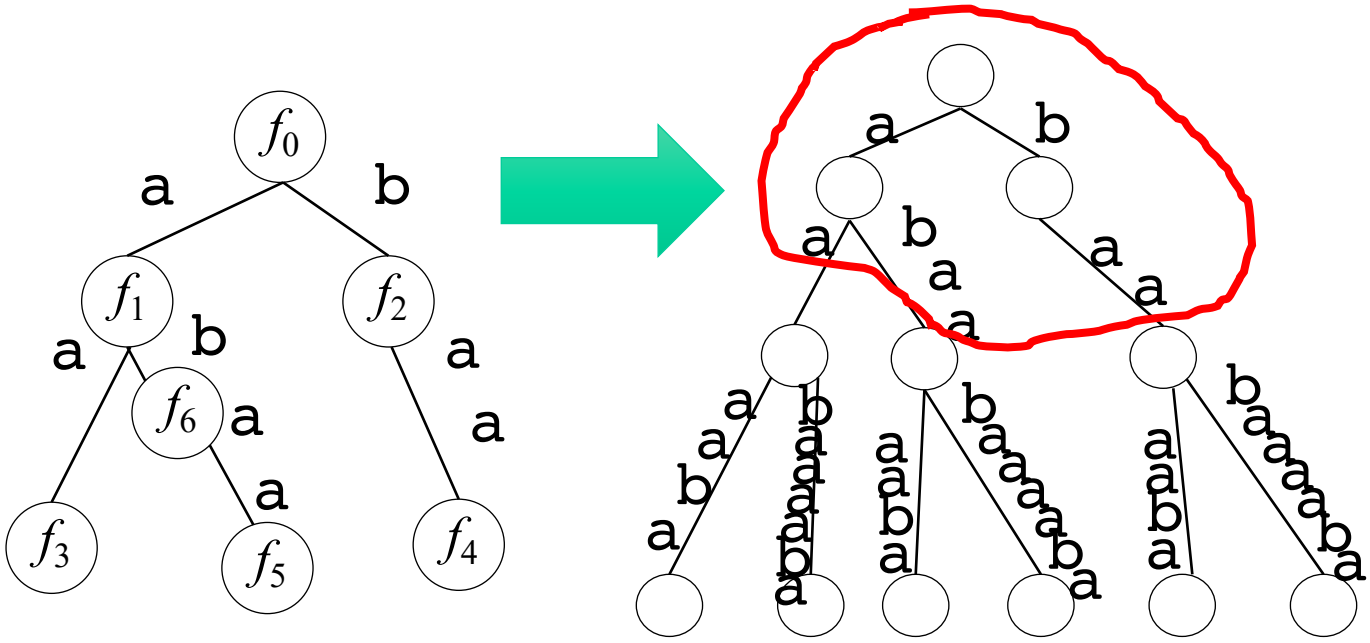
Ideas of $O(N \log \sigma)$ time Algorithm

- It superimposes the Patricia tree for previous factors into Suffix Tree



Ideas of $O(N \log \sigma)$ time Algorithm

- It superimposes the Patricia tree for previous factors into Suffix Tree
- For each position i, the deepest node which represents a substring starting at i can be computed in amortized $O(\log \sigma)$ time using Ukkonen's algorithm



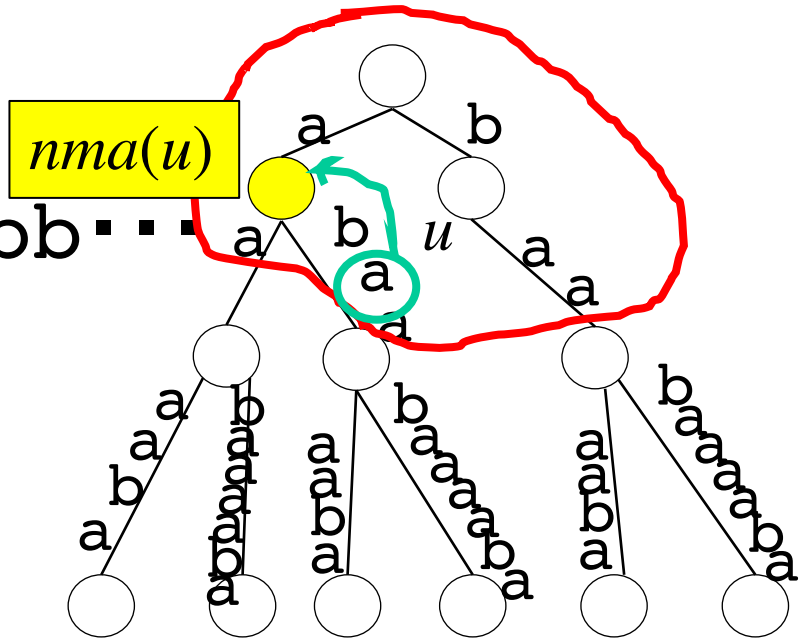
Ideas of $O(N \log \sigma)$ time Algorithm

- For each node u of a growing tree, the nearest marked ancestor of u can be computed in amortized constant time, and an unmarked node can be marked in amortized constant time [Amir+, 1995]

$T =$ abaaabaaabaaababaabb···

f_1 f_2 f_3 f_4 f_5

(a, f_0) (b, f_0) (f_1, f_1) (f_2, f_3) (f_1, f_4)



$O(N \log \sigma)$ LZD Algorithm by Ukkonen's Suffix Tree

Theorem

For a string T of length N over an alphabet of size σ, LZD factorization can be computed in $O(N \log \sigma)$ time and $O(N)$ space

LZD factorization with Variable-to-Fixed-Encoding

- LZDVF is a restricted version of LZD that remembers at most 2^L previous factors for each step, where L is the bit-length of codewords representing factors
- Therefore, each factor is the concatenation of two longest previous factors in the dictionary containing at most 2^L previous factors

- We propose two variants of LZDVF
  - LZDVF Prefix : based on least recently used (LRU) strategy
  - LZDVF Count : based on least frequently used (LFU) strategy

LZDVF Prefix

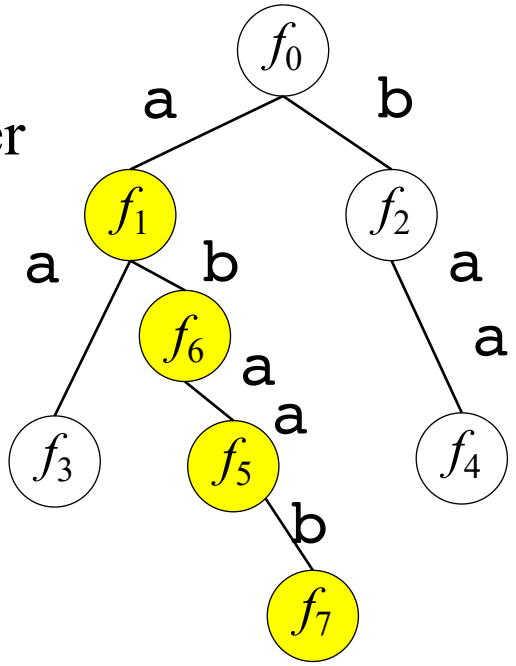
- When a new factor f_i is added, LZDVF Prefix considers that all factors, which are non-empty prefixes of f_i, are used at this step

E.g. when creating a factor $f_7 = (f_5, f_2)$
we consider f_7, f_5, f_6, f_1 are used in this order

$T =$ a | b | aa | ba | aba | aba | abb | ···

f_1 f_2 f_3 f_4 f_5 f_6 f_7

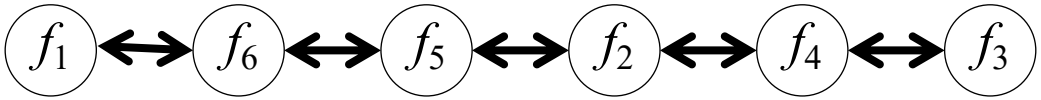
(a, f_0) (b, f_0) (f_1, f_1) (f_2, f_3) (f_1, f_4) (f_1, f_2) (f_5, f_2)



LZDVF Prefix

- Manages factors in a doubly linked list in frequently used order
- Deleting and using a factor can be done in constant time

most recently used — least recently used



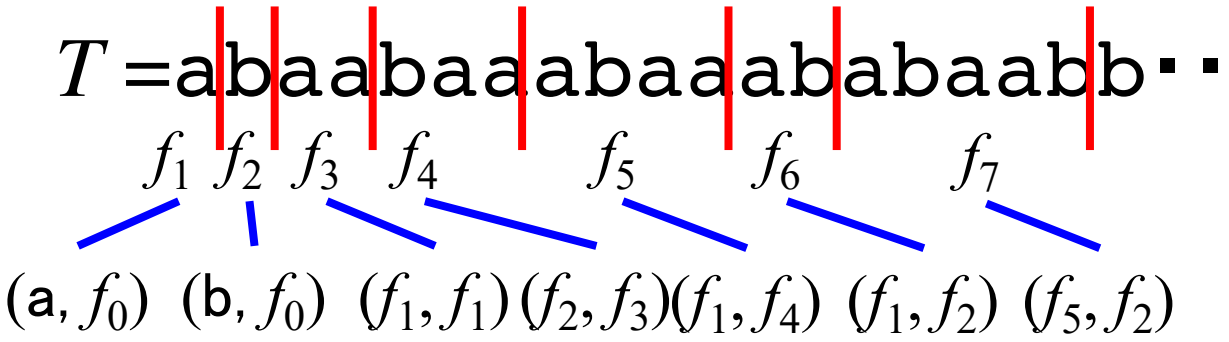
Theorem

Assume $2^L < m$ for simplicity. For a string T of length N over an alphabet of size σ, LZDVF Prefix can be computed in $O(N + m\,(M + \min(M, 2^L) \log \sigma))$ time and $O(2^L)$ working space

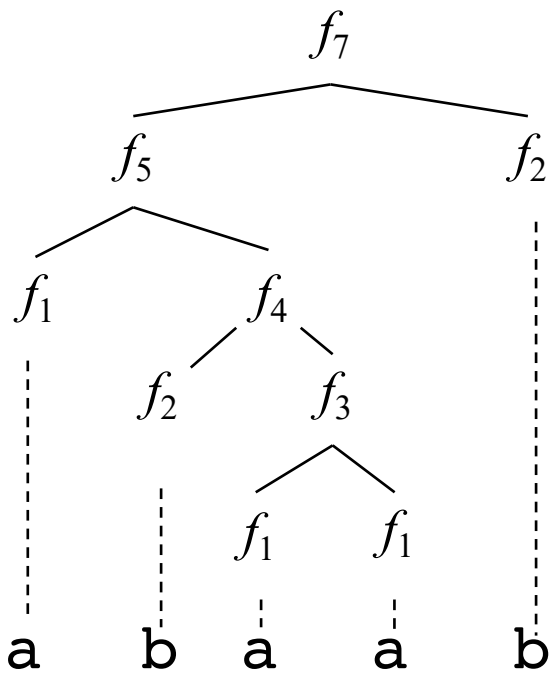
LZDVF Count

- When a new factor f_i is added, LZDVF Count considers that at this step, each factor is used by the number of occurrences in the derivation tree of f_i

E.g. when creating a factor $f_7 = (f_5, f_2)$,
f_5, f_4, f_3 are used once, f_2 is used twice,
and f_1 is used 3 times



The derivation tree of f_7



LZDVF Count

- Manages frequencies for a factor f_i by $Count(f_i)$
- When a new factor f_i is added, $Count(f_j)$ is increased by $vOcc(f_i, f_j)$ for f_j such that $vOcc(f_i, f_j) > 0$, where $vOcc(f_i, f_j)$ is the number of occurrences of f_j in the derivation tree of f_i
- If the dictionary is full, $Count(f_i)$ is decreased one for all factors, and delete factors f_i such that $Count(f_i) = 0$

Theorem

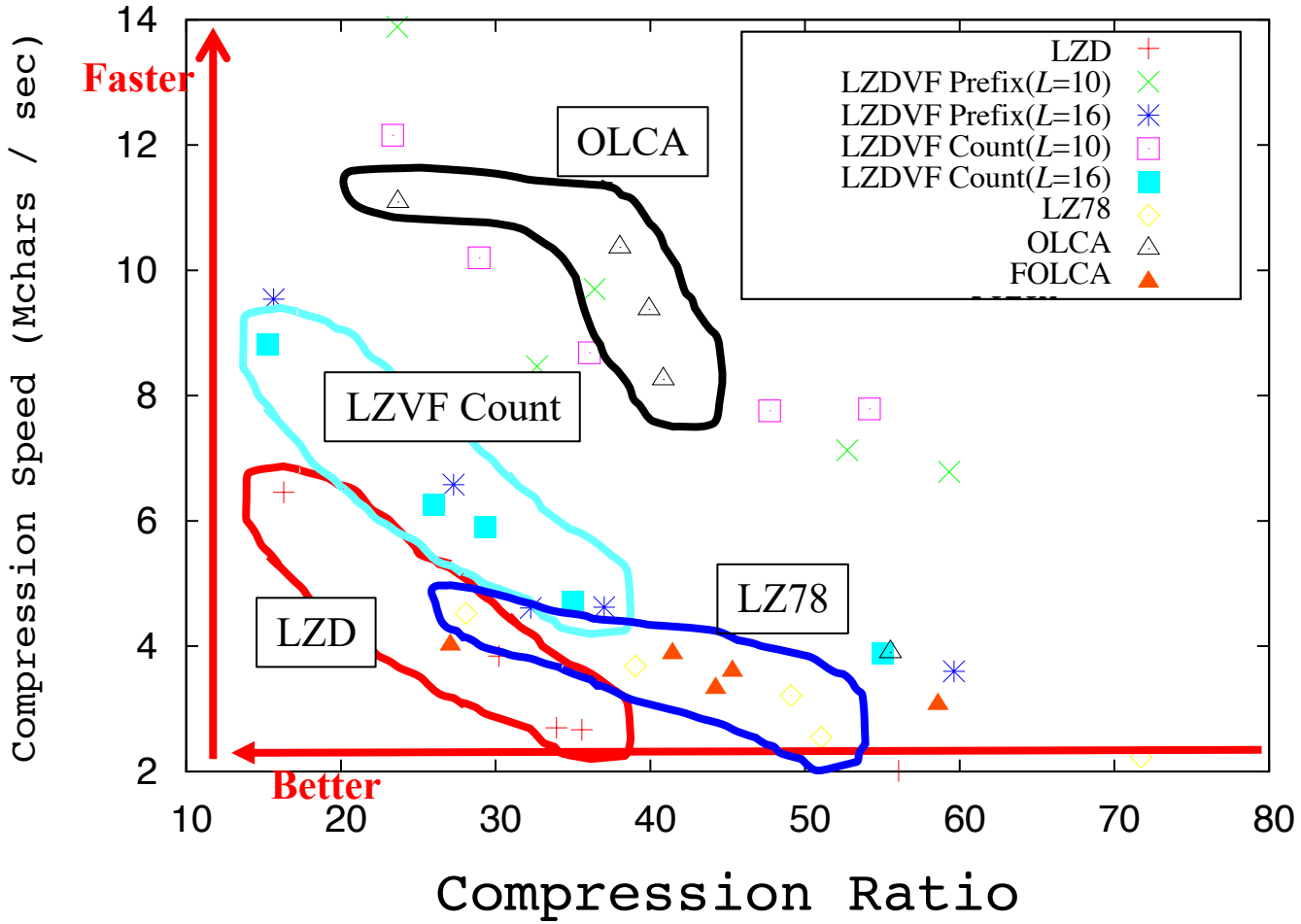
Assume $2^L < m$ for simplicity. For a string T of length N over an alphabet of size σ, LZDVF Prefix can be computed in $O(N + m\,(M + \min(M, 2^L) \log \sigma)$ time and $O(2^L)$ working space

Computational Experiments

- We compared our algorithms LZD (Naive), LZDVF Prefix, Count, and previous online grammar compression algorithms LZ78, FOLCA[Maruyama+, 2013] and OLCA[Maruyama, 2011]
- LZD and LZ78 are once transformed to Straight Line Programs, and encoded in same encoding of OLCA
- Data
  - Pizza Chili corpus http://pizzachili.dcc.uchile.cl/texts.html
    (ENGLISH, DNA, PROTEINS, DBLP, SOURCES of size 200MB)

Compression Speed and Ratio for pizza chili corpus

- Compression speed of LZD/LZDVFs is slow
- Compression ratio of LZD/LZDVF Count is better than the others



Computational Experiments for Highly-repetitive Large Texts

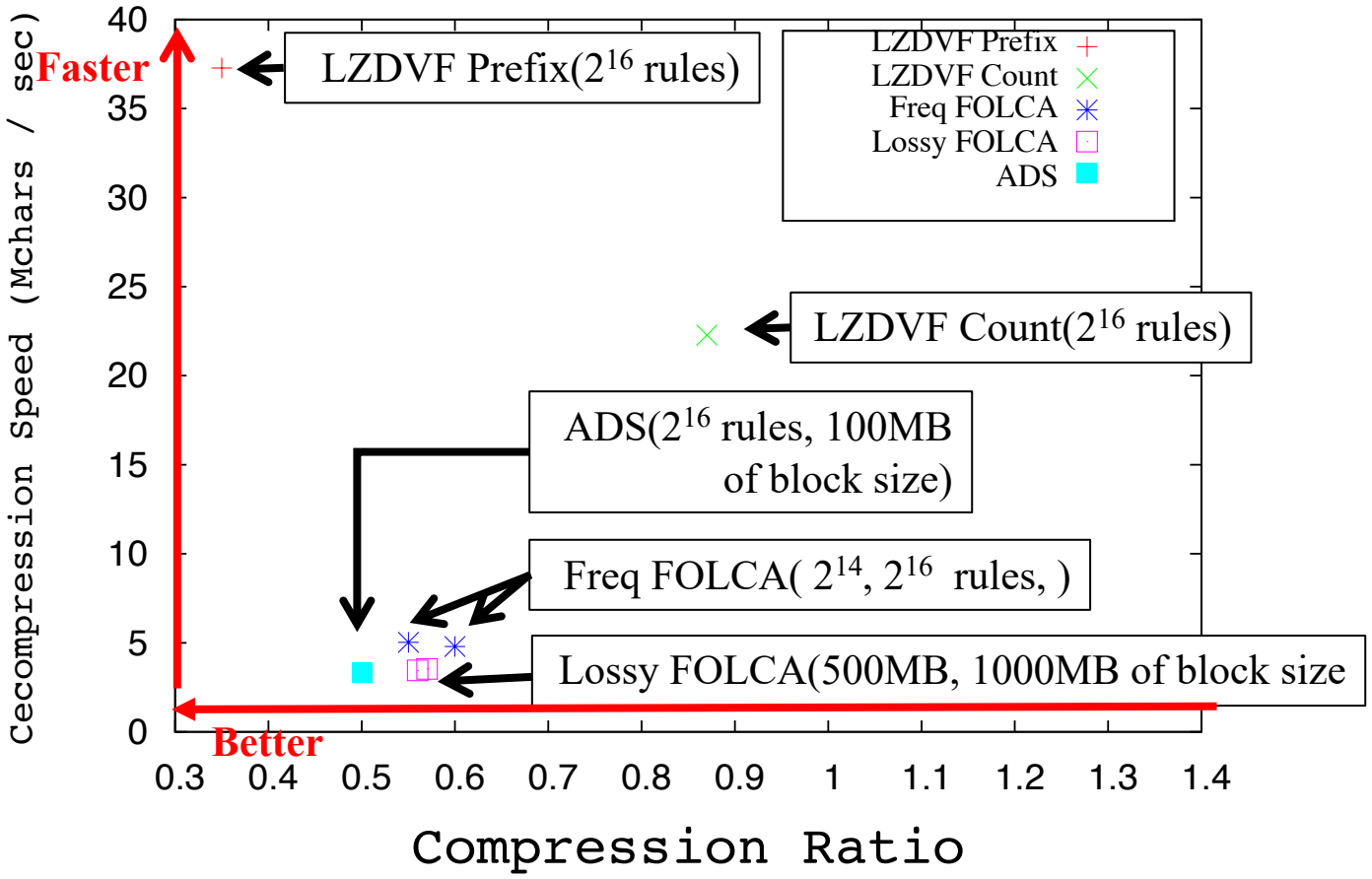
- We compared LZDVFs with
  - Lossy FOLCA
  - Freq OLCA

  } [Maruyama and Tabei, 2014]
  - ADS[Sekine+, 2014]
- For each algorithm except Lossy FOLCA, the parameter of the number of rules locally stored is varied to 2^{12}, 2^{14}, 2^{16}
- For Lossy FOLCA, the parameter of the block size separating the input is varied to 100MB, 500MB, 1000MB,
  and for ADS, it is fixed to 100MB
- Data
  - Highly-repetitive large texts, 10GB of English Wikipedia edit history

http://dumps.wikimedia.org/backup-index.html

Compression Speed for Highly-repetitive Large Texts

- LZDVF Prefix achieves the best compression ratio
- Speed of LZDVF Prefix is almost **8 times faster** than other previous algorithms



Summary

- We proposed LZ78 like online grammar compressions
  - LZ Double
  - LZ Double with Variable-to-Fixed Encoding

Future Work

- Find a worst case instance for Naive algorithm for LZD
- Develop an algorithm to compute LZD using less space
- Analyze approximation ratio of LZD to the smallest grammar