Maintaining Longest Common Extensions in Dyn-FO

Accepted for Symposium on Combinatorial Pattern Matching (CPM) 2026

Daniel Albert

March 3, 2026

Logic and Computing: Databases, Automata, Complexity
Chair 1: Logic in Computer Science
Faculty of Computer Science
Technische Universität Dortmund

Dyn-FO

Dyn-FO [Patnaik and Immerman 1997]

- Dynamic algorithm where updates and queries are expressible in FO-logic

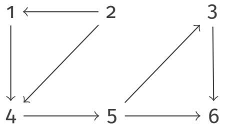
Dyn-FO

Dyn-FO [Patnaik and Immerman 1997]

- Dynamic algorithm where updates and queries are expressible in FO-logic

REACH in acyclical graphs [Patnaik and Immerman 1997, Theorem 4.2]

Query(a, b)	Is b reachable from a in G?



- Query$(1, 6)$ = true

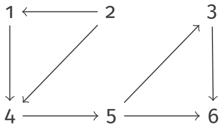
Dyn-FO

Dyn-FO [Patnaik and Immerman 1997]

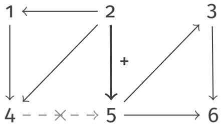
- Dynamic algorithm where updates and queries are expressible in FO-logic

REACH in acyclical graphs [Patnaik and Immerman 1997, Theorem 4.2]

Data	
Ins(a, b)	Insert edge (a, b) into G. (assume G remains acyclical)
Del(a, b)	Delete edge (a, b) from G.
Query(a, b)	Is b reachable from a in G?



- Query$(1, 6) =$ true
- Del$(4, 5)$
- Ins$(2, 5)$



- Query$(1, 6) =$ false

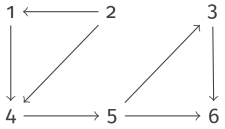
Dyn-FO

Dyn-FO [Patnaik and Immerman 1997]

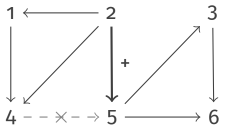
- Dynamic algorithm where updates and queries are expressible in FO-logic

REACH in acyclical graphs [Patnaik and Immerman 1997, Theorem 4.2]

Data	$R(x, y)$: node y is reachable from node x in G
Ins(a, b)	Insert edge (a, b) into G. (assume G remains acyclical) $R'(x, y) \equiv$
Del(a, b)	Delete edge (a, b) from G. $R'(x, y) \equiv$
Query(a, b)	Is b reachable from a in G? $R(a, b)$.



- Query$(1, 6)$ = true
- Del$(4, 5)$
- Ins$(2, 5)$



- Query$(1, 6)$ = false

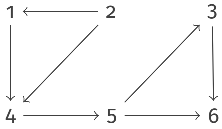
Dyn-FO

Dyn-FO [Patnaik and Immerman 1997]

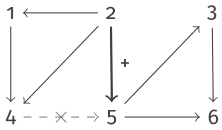
- Dynamic algorithm where updates and queries are expressible in FO-logic

REACH in acyclical graphs [Patnaik and Immerman 1997, Theorem 4.2]

Data	$R(x, y):$ node y is reachable from node x in G
Ins(a, b)	Insert edge (a, b) into G. (assume G remains acyclical) $R'(x, y) \equiv R(x, y) \vee (R(x, a) \wedge R(b, y))$
Del(a, b)	Delete edge (a, b) from G. $R'(x, y) \equiv$
Query(a, b)	Is b reachable from a in G? $R(a, b)$.



- Query$(1, 6)$ = true
- Del$(4, 5)$
- Ins$(2, 5)$



- Query$(1, 6)$ = false

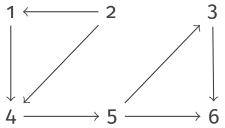
Dyn-FO

Dyn-FO [Patnaik and Immerman 1997]

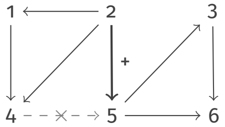
- Dynamic algorithm where updates and queries are expressible in FO-logic

REACH in acyclical graphs [Patnaik and Immerman 1997, Theorem 4.2]

Data	$R(x,y)$:	node y is reachable from node x in G
Ins(a,b)	Insert edge (a,b) into G. (assume G remains acyclical)	
	$R'(x,y) \equiv$	$R(x,y) \vee (R(x,a) \wedge R(b,y))$
Del(a,b)	Delete edge (a,b) from G.	
	$R'(x,y) \equiv$	$R(x,y) \wedge \big[\neg R(x,a) \vee \neg R(b,y) \vee \exists u,v\big(E(u,v) \wedge R(x,u) \wedge R(u,a) \wedge R(v,y) \wedge \neg R(v,a) \wedge (v \neq b \vee u \neq b)\big)\big]$
Query(a,b)	Is b reachable from a in G?	
	$R(a,b)$.	



- Query$(1,6)$ = true
- Del$(4,5)$
- Ins$(2,5)$



- Query$(1,6)$ = false

Dyn-FO Continuation

Successor in $D \subseteq [1, n]$

Ins(a)	Insert a into D.
Del(a)	Remove a from D.
Query(a, b)	Is b the successor of a in D?

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, 25, 29, 31\}$

- Query$(23, 28) =$ false
- Query$(23, 25) =$ true
- Ins(28)
- Del(25)

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, \cancel{25}, \underline{28}, 29, 31\}$

- Query$(23, 28) =$ true
- Query$(23, 25) =$ false

Dyn-FO Continuation

Successor in $D \subseteq [1, n]$

Data	$D(x): \quad x \in D$
Ins(a)	Insert a into D.
	$D'(x) \equiv \quad D(x) \lor a = x$
Del(a)	Remove a from D.
	$D'(x) \equiv \quad D(x) \land a \neq x$
Query(a, b)	Is b the successor of a in D?

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, 25, 29, 31\}$

- Query$(23, 28)$ = false
- Query$(23, 25)$ = true
- Ins(28)
- Del(25)

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, \cancel{25}, \underline{28}, 29, 31\}$

- Query$(23, 28)$ = true
- Query$(23, 25)$ = false

Dyn-FO Continuation

Successor in $D \subseteq [1, n]$

Data	$D(x): \quad x \in D$
Ins(a)	Insert a into D. $D'(x) \equiv \quad D(x) \vee a = x$
Del(a)	Remove a from D. $D'(x) \equiv \quad D(x) \wedge a \neq x$
Query(a, b)	Is b the successor of a in D? $D(b) \wedge b > a \wedge \neg\exists v\big[D(v) \wedge v > a \wedge b > v\big]$

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, 25, 29, 31\}$

- Query$(23, 28)$ = false
- Query$(23, 25)$ = true
- Ins(28)
- Del(25)

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, \cancel{25}, \underline{28}, 29, 31\}$

- Query$(23, 28)$ = true
- Query$(23, 25)$ = false

Dyn-FO Continuation

Successor in $D \subseteq [1, n]$

Data	$D(x):$	$x \in D$
Ins(a)	Insert a into D.	
	$D'(x) \equiv$	$D(x) \vee a = x$
Del(a)	Remove a from D.	
	$D'(x) \equiv$	$D(x) \wedge a \neq x$
Query(a, b)	Is b the successor of a in D?	
	$D(b) \wedge b > a \wedge \neg\exists v\left[D(v) \wedge v > a \wedge b > v\right]$	

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, 25, 29, 31\}$

- Query$(23, 28)$ = false
- Query$(23, 25)$ = true
- Ins(28)
- Del(25)

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, \not{25}, \underline{28}, 29, 31\}$

- Query$(23, 28)$ = true
- Query$(23, 25)$ = false

Observation

- Algorithm barely uses auxiliary information and updates.
  - We can probably do better.
  - What does *better* mean?

Dyn-FO Continuation

Successor in $D \subseteq [1, n]$

Data	$D(x): \quad x \in D$
Ins(a)	Insert a into D. $D'(x) \equiv \quad D(x) \lor a = x$
Del(a)	Remove a from D. $D'(x) \equiv \quad D(x) \land a \neq x$
Query(a, b)	Is b the successor of a in D? $D(b) \land b > a \land \neg\exists v \big[D(v) \land v > a \land b > v\big]$

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, 25, 29, 31\}$

- Query$(23, 28)$ = false
- Query$(23, 25)$ = true
- Ins(28)
- Del(25)

$D = \{3, 6, 7, 10, 11, 14, 16, 18, 20, 22, \cancel{25}, \underline{28}, 29, 31\}$

- Query$(23, 28)$ = true
- Query$(23, 25)$ = false

Observation

- Algorithm barely uses auxiliary information and updates.
- We can probably do better.
- What does *better* mean?

  - **Work-Sensitive Dyn-FO**

Work-Sensitive Dyn-FO

Follows from Immerman 1999, Corollary 5.10

Problem $\mathcal{P}$ is in Dyn-FO

$\Leftrightarrow$

There is a dynamic parallel constant-time algorithm for $\mathcal{P}$ on a common CRCW PRAM with polynomial work.

Work-Sensitive Dyn-FO

Follows from Immerman 1999, Corollary 5.10

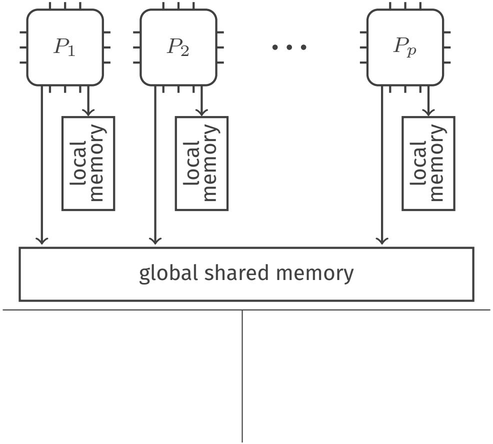
Problem $\mathcal{P}$ is in Dyn-FO

$\Leftrightarrow$

There is a dynamic parallel constant-time algorithm for $\mathcal{P}$ on a common CRCW PRAM with polynomial work.

common CRCW PRAM

PRAM parallel machine with multiple processors, synchronized steps, shared memory



Work-Sensitive Dyn-FO

Follows from Immerman 1999, Corollary 5.10

Problem $\mathcal{P}$ is in Dyn-FO

$\Leftrightarrow$

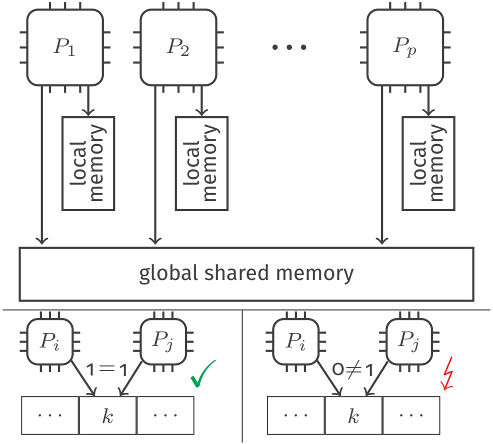
There is a dynamic parallel constant-time algorithm for $\mathcal{P}$ on a common CRCW PRAM with polynomial work.

common CRCW PRAM

PRAM parallel machine with multiple processors, synchronized steps, shared memory

CRCW concurrent read and write allowed

common concurrent write only with same value



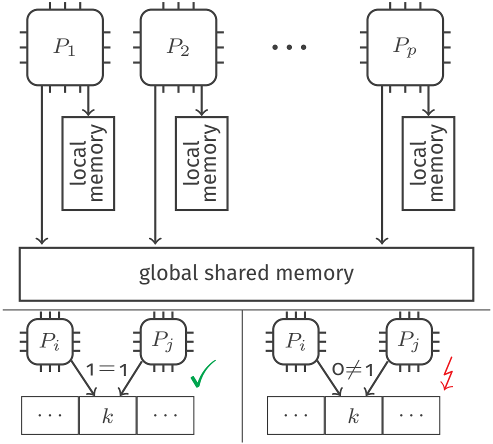
Work-Sensitive Dyn-FO

Follows from Immerman 1999, Corollary 5.10

Problem $\mathcal{P}$ is in Dyn-FO

$\Leftrightarrow$

There is a dynamic parallel constant-time algorithm for $\mathcal{P}$ on a common CRCW PRAM with polynomial work.



common CRCW PRAM

PRAM	parallel machine with multiple processors, synchronized steps, shared memory
CRCW	concurrent read and write allowed
common	concurrent write only with same value

number of processors $\times$ runtime $=$ work

▶ ***better* = less work**

Longest Common Extension (LCE)

Definition

Given: String $S[1, n]$, indices i, j

Longest Common Extension (LCE) $=$

$$\begin{array}{cccccccccccccccccccccc} & & & & & & & & & i=9 & & & & & & j=16 & & & & \\ & & & & & & & & & \downarrow & & & & & & \downarrow & & & & \\ S = & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{a} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{a} \end{array}$$

Longest Common Extension (LCE)

Definition

Given: String $S[1, n]$, indices i, j

Longest Common Extension (LCE) $= \begin{cases} \text{Longest Common Prefix (LCP) of } S[i, n] \text{ and } S[j, n] \\ \end{cases}$

$i = 9$ ↓ $\quad j = 16$ ↓

$S =$ a b a b c a a b | b a b c a | a b | b a b c a |

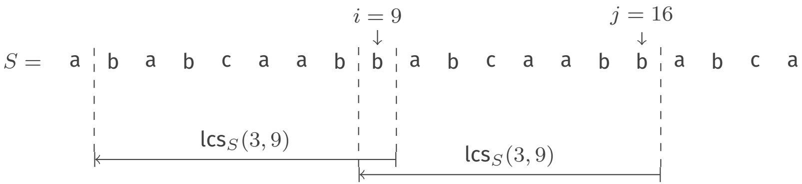
$\mathrm{lcp}_S(3, 9)$ $\quad$ $\mathrm{lcp}_S(3, 9)$

Longest Common Extension (LCE)

Definition

Given: String $S[1,n]$, indices i, j

Longest Common Extension (LCE) $= \begin{cases} \text{Longest Common Prefix (LCP) of } S[i,n] \text{ and } S[j,n] \\ \text{Longest Common Suffix (LCS) of } S[1,i] \text{ and } S[1,j] \end{cases}$

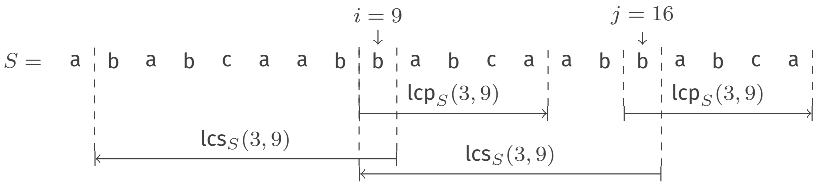


Longest Common Extension (LCE)

Definition

Given: String $S[1,n]$, indices i, j

Longest Common Extension (LCE) $=$
- Longest Common Prefix (LCP) of $S[i,n]$ and $S[j,n]$
- Longest Common Suffix (LCS) of $S[1,i]$ and $S[1,j]$



Observation

- LCP and LCS are symmetrical
  - From now only LCP

Dynamic LCP

init($\{a, b, c\}$) $S = \varepsilon$

Problem

init(Σ)

- Initialization $S = \varepsilon$
- Alphabet Σ

Dynamic LCP

$\mathsf{init}(\{a, b, c\})$ $S = \varepsilon$

$\mathsf{ins}(1, \mathsf{a})$ $S = \boxed{\mathbf{a}}$

$\mathsf{ins}(1, \mathsf{c})$ $S = \boxed{\mathbf{c}}$ a

⋮

$\mathsf{ins}(4, \mathsf{b})$ $S =$ a b a $\boxed{\mathbf{b}}$ c a a b b a b c a a b b a b c a

Problem

$\mathsf{init}(\Sigma)$

- Initialization $S = \varepsilon$
- Alphabet Σ

$\mathsf{ins}(i, \sigma)$

- Insert σ at i

Dynamic LCP

init$(\{a, b, c\})$	$S = \varepsilon$
ins(1, a)	$S =$ **a**
ins(1, c)	$S =$ **c** a
⋮	
ins(4, b)	$S =$ a b a **b** c a a b b a b c a a b b a b c a
lcp(9, 16) = 5	$S =$ a b a b c a a b [**b** a b c a] a b [**b** a b c a]

Problem

init(Σ)

- Initialization $S = \varepsilon$
- Alphabet Σ

ins(i, σ)

- Insert σ at i

lcp(i, j)

- Compute lcp$_S(i, j)$

Dynamic LCP

init$(\{a, b, c\})$	$S = \varepsilon$
ins$(1, \text{a})$	$S =$ **a**
ins$(1, \text{c})$	$S =$ **c** a
⋮	
ins$(4, \text{b})$	$S =$ a b a **b** c a a b b a b c a a b b a b c a
lcp$(9, 16) = 5$	$S =$ a b a b c a a b [**b** a b c a] a b [**b** a b c a]
del(3)	$S =$ a b ‖ b c a a b b a b c a a b b a b c a
lcp$(1, 13) = 3$	$S =$ [**a** b b] c a a b b a b c a [**a** b b] a b c a

Problem

init(Σ)
- Initialization $S = \varepsilon$
- Alphabet Σ

ins(i, σ)
- Insert σ at i

del(i)
- Delete position i

lcp(i, j)
- Compute lcp$_S(i, j)$

LCP Algorithms

Computing LCP from scratch

sequential naïve in $\mathcal{O}(n)$ time

parallel constant time with $\mathcal{O}(n)$ work (slightly more complicated)

LCP Algorithms

Computing LCP from scratch

sequential naïve in $\mathcal{O}(n)$ time

parallel constant time with $\mathcal{O}(n)$ work (slightly more complicated)

- both clearly optimal
- from scratch not very interesting

LCP Algorithms

Computing LCP from scratch

sequential	naïve in $\mathcal{O}(n)$ time
parallel	constant time with $\mathcal{O}(n)$ work (slightly more complicated)

- both clearly optimal
- from scratch not very interesting

Static Algorithms

- Given S, build data structure $\mathcal{D}$
- Given i, j and $\mathcal{D}$, compute LCP in $\mathcal{O}(1)$

LCP Algorithms

Computing LCP from scratch

sequential naïve in $\mathcal{O}(n)$ time

parallel constant time with $\mathcal{O}(n)$ work (slightly more complicated)

- both clearly optimal
- from scratch not very interesting

Static Algorithms

- Given S, build data structure $\mathcal{D}$
- Given i, j and $\mathcal{D}$, compute LCP in $\mathcal{O}(1)$

- Farach-Colton, Ferragina, and Muthukrishnan 2000: Sequential, Time $\mathcal{O}(n)$
- Shun 2014: Parallel, Time $\mathcal{O}(\log^2 n)$, Work $\mathcal{O}(n)$

LCP Algorithms

Computing LCP from scratch

sequential naïve in $\mathcal{O}(n)$ time

parallel constant time with $\mathcal{O}(n)$ work (slightly more complicated)

- both clearly optimal
- from scratch not very interesting

Static Algorithms

- Given S, build data structure $\mathcal{D}$
- Given i, j and $\mathcal{D}$, compute LCP in $\mathcal{O}(1)$

- Farach-Colton, Ferragina, and Muthukrishnan 2000: Sequential, Time $\mathcal{O}(n)$
- Shun 2014: Parallel, Time $\mathcal{O}(\log^2 n)$, Work $\mathcal{O}(n)$

Dynamic Algorithms

- Maintain data structure $\mathcal{D}$ for changing S
- Given i, j and $\mathcal{D}$, compute LCP efficiently

LCP Algorithms

Computing LCP from scratch

sequential naïve in $\mathcal{O}(n)$ time

parallel constant time with $\mathcal{O}(n)$ work (slightly more complicated)

- both clearly optimal
- from scratch not very interesting

Static Algorithms

- Given S, build data structure $\mathcal{D}$
- Given i, j and $\mathcal{D}$, compute LCP in $\mathcal{O}(1)$

- Farach-Colton, Ferragina, and Muthukrishnan 2000: Sequential, Time $\mathcal{O}(n)$
- Shun 2014: Parallel, Time $\mathcal{O}(\log^2 n)$, Work $\mathcal{O}(n)$

Dynamic Algorithms

- Maintain data structure $\mathcal{D}$ for changing S
- Given i, j and $\mathcal{D}$, compute LCP efficiently

- Gawrychowski et al. 2018:
  - Related problem, Sequential
  - Time $\mathcal{O}(\log n)$ with high probability

LCP Algorithms

Computing LCP from scratch

sequential naïve in $\mathcal{O}(n)$ time

parallel constant time with $\mathcal{O}(n)$ work (slightly more complicated)

- both clearly optimal
- from scratch not very interesting

Static Algorithms

- Given S, build data structure $\mathcal{D}$
- Given i, j and $\mathcal{D}$, compute LCP in $\mathcal{O}(1)$

- Farach-Colton, Ferragina, and Muthukrishnan 2000: Sequential, Time $\mathcal{O}(n)$
- Shun 2014: Parallel, Time $\mathcal{O}(\log^2 n)$, Work $\mathcal{O}(n)$

Dynamic Algorithms

- Maintain data structure $\mathcal{D}$ for changing S
- Given i, j and $\mathcal{D}$, compute LCP efficiently

- Gawrychowski et al. 2018:
  - Related problem, Sequential
  - Time $\mathcal{O}(\log n)$ with high probability

▶ Work-Sensitive Dyn-FO

For any $\varepsilon > 0$,

$\mathcal{O}(n^\varepsilon)$ work for updates and queries

in parallel constant time on common CRCW PRAM.

Overview

▶ Work-Sensitive Dyn-FO

For any $\varepsilon > 0$, parallel constant time with $\mathcal{O}(n^\varepsilon)$ work for updates and queries on common CRCW PRAM.

Ingredients

Overview

▶ Work-Sensitive Dyn-FO

For any $\varepsilon > 0$, parallel constant time with $\mathcal{O}(n^{\varepsilon})$ work for updates and queries on common CRCW PRAM.

Ingredients

String Synchronizing Sets

- tool from string algorithms
- answering queries easy
- constructing difficult
- maintaining even worse

Overview

▶ Work-Sensitive Dyn-FO

For any $\varepsilon > 0$, parallel constant time with $\mathcal{O}(n^{\varepsilon})$ work for updates and queries on common CRCW PRAM.

Ingredients

String Synchronizing Sets

- tool from string algorithms
- answering queries easy
- constructing difficult
- maintaining even worse

Kempa and Kociumaka 2022

- dynamic string algorithm
- maintains suffix-array
- can maintain string synchronizing sets (with some modifications)
- requires logarithmic time

Overview

▶ Work-Sensitive Dyn-FO

For any $\varepsilon > 0$, parallel constant time with $\mathcal{O}(n^{\varepsilon})$ work for updates and queries on common CRCW PRAM.

Ingredients

String Synchronizing Sets

- tool from string algorithms
- answering queries easy
- constructing difficult
- maintaining even worse

Kempa and Kociumaka 2022

- dynamic string algorithm
- maintains suffix-array
- can maintain string synchronizing sets (with some modifications)
- requires logarithmic time

Muddling

- tool from Dyn-FO algorithms
- from Datta et al. 2019
- constant-time updates compute non-constant-time algorithm
- but introduces outdated information

Overview

▶ Work-Sensitive Dyn-FO

For any $\varepsilon > 0$, parallel constant time with $\mathcal{O}(n^{\varepsilon})$ work for updates and queries on common CRCW PRAM.

Ingredients

String Synchronizing Sets

- tool from string algorithms
- answering queries easy
- constructing difficult
- maintaining even worse

Kempa and Kociumaka 2022

- dynamic string algorithm
- maintains suffix-array
- can maintain string synchronizing sets (with some modifications)
- requires logarithmic time

Muddling

- tool from Dyn-FO algorithms
- from Datta et al. 2019
- constant-time updates compute non-constant-time algorithm
- but introduces outdated information

+ technical details and auxiliary algorithms, skipped here

String Synchronizing Sets

String Synchronizing Set B^τ (e.g. Kociumaka et al. 2024, Section 4)

Set of occurrences (starting positions) of length-τ substrings within a string S with:

$S =$ **a** b **a** b c a a b b **a** b **c** a a b b a b c b

B^4 : 1 2 2 3

1: abab

2: abca

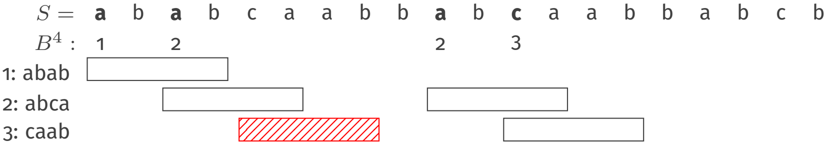
3: caab

String Synchronizing Sets

String Synchronizing Set B^τ (e.g. Kociumaka et al. 2024, Section 4)

Set of occurrences (starting positions) of length-τ substrings within a string S with:

Consistency: Contains either none or all occurrences of a substring ↯



String Synchronizing Sets

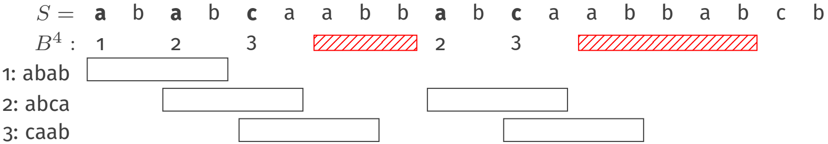
String Synchronizing Set B^τ (e.g. Kociumaka et al. 2024, Section 4)

Set of occurrences (starting positions) of length-τ substrings within a string S with:

Consistency: Contains either none or all occurrences of a substring ✓

Density: There is an occurrence at least every $\frac{1}{2}\tau$ positions.* ↯

*except periodic (repeating) parts



String Synchronizing Sets

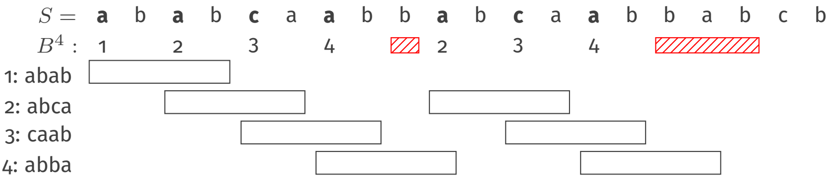
String Synchronizing Set B^τ (e.g. Kociumaka et al. 2024, Section 4)

Set of occurrences (starting positions) of length-τ substrings within a string S with:

Consistency: Contains either none or all occurrences of a substring ✓

Density: There is an occurrence at least every $\frac{1}{2}\tau$ positions.* ↯

*except periodic (repeating) parts



String Synchronizing Sets

String Synchronizing Set B^τ (e.g. Kociumaka et al. 2024, Section 4)

Set of occurrences (starting positions) of length-τ substrings within a string S with:

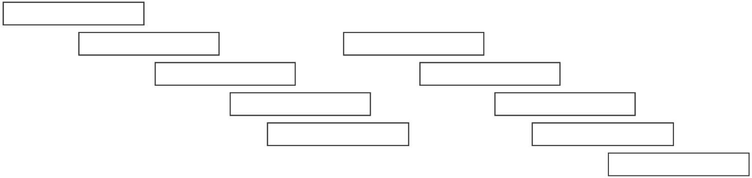
Consistency: Contains either none or all occurrences of a substring ✓

Density: There is an occurrence at least every $\frac{1}{2}\tau$ positions.* ✓ *except periodic (repeating) parts

Sparseness: The total length of all occurrences is $\mathcal{O}(n)$. ¯_(ツ)_/¯

$S =$	**a**	b	**a**	b	**c**	a	**a**	**b**	b	**a**	b	**c**	a	**a**	**b**	b	**a**	b	c	b
B^4 :	1		2		3		4	5		2		3		4	5		6			

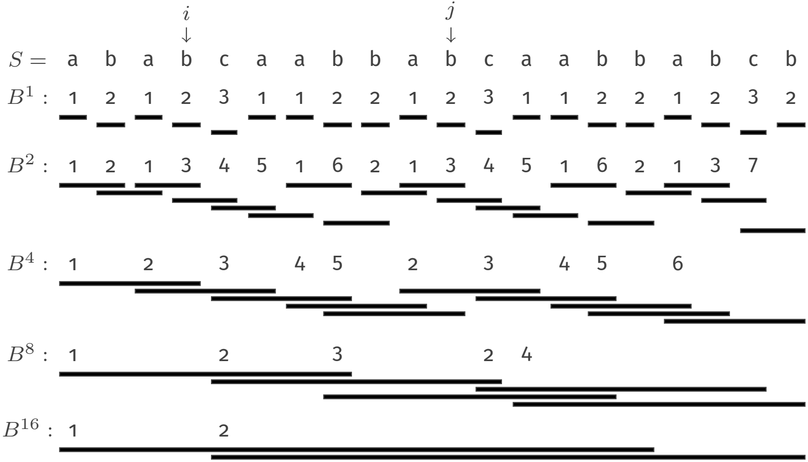
1: abab
2: abca
3: caab
4: abba
5: bbab
6: abcb



Answering Queries with String Synchronizing Sets

$S =$	a	b	a	b	c	a	a	b	b	a	b	c	a	a	b	b	a	b	c	b
B^1 :	1	2	1	2	3	1	1	2	2	1	2	3	1	1	2	2	1	2	3	2
B^2 :	1	2	1	3	4	5	1	6	2	1	3	4	5	1	6	2	1	3	7	
B^4 :	1		2		3		4	5		2		3		4	5		6			
B^8 :	1				2			3				2	4							
B^{16} :	1				2															

Answering Queries with String Synchronizing Sets

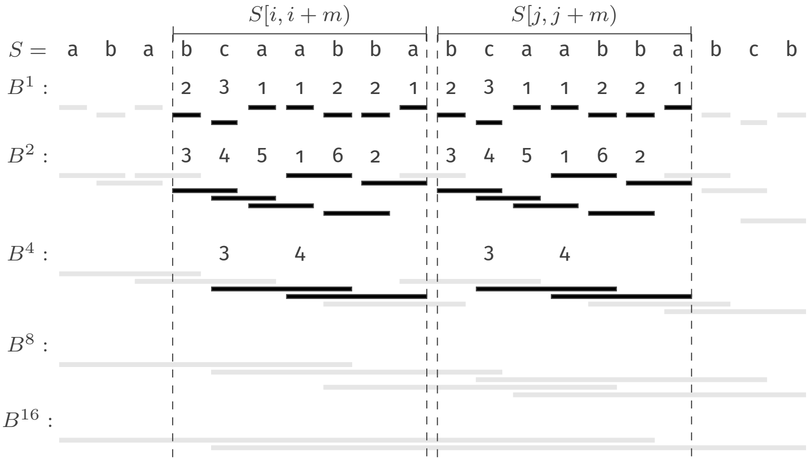


Compute lcp(i, j)

- Search on $m' = \text{lcp}(i, j)$
  - Binary-Search-like
  - $\mathcal{O}(n^{\varepsilon})$ equality queries

"$S[i, i+m) = S[j, j+m)$?"

Answering Queries with String Synchronizing Sets



Compute lcp(i, j)

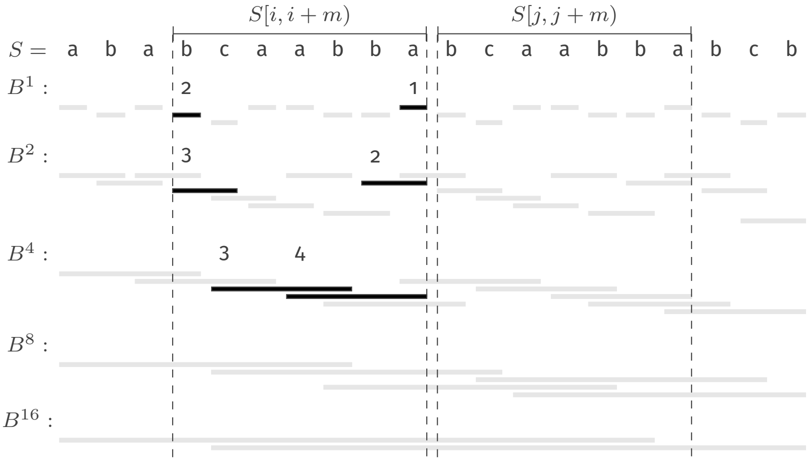
- Search on $m' = \text{lcp}(i, j)$
  - Binary-Search-like
  - $\mathcal{O}(n^{\varepsilon})$ equality queries

"$S[i, i+m) = S[j, j+m)$?"

Equality Query

$$S[i, i+m) = S[j, j+m)$$
$$\Leftrightarrow$$
both have same occurrences

Answering Queries with String Synchronizing Sets



Compute lcp(i, j)

- Search on $m' =$ lcp(i, j)
  - Binary-Search-like
  - $\mathcal{O}(n^\varepsilon)$ equality queries

"$S[i, i+m) = S[j, j+m)$?"

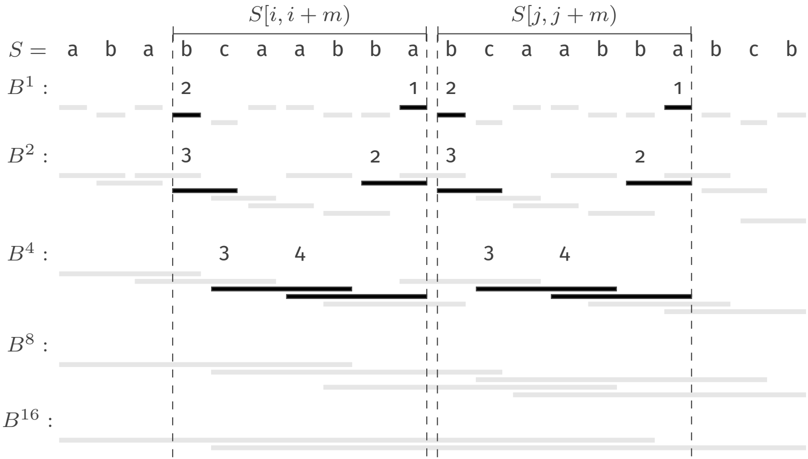
Equality Query

$$S[i, i+m) = S[j, j+m)$$
$$\Leftrightarrow$$

both have same occurrences

1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences.

Answering Queries with String Synchronizing Sets



Compute lcp(i, j)

- Search on $m' =$ lcp(i, j)
  - Binary-Search-like
  - $\mathcal{O}(n^\varepsilon)$ equality queries "$S[i, i+m) = S[j, j+m)$?"

Equality Query

$$S[i, i+m) = S[j, j+m)$$
$$\Leftrightarrow$$
both have same occurrences

1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences.
2. Check if $S[j, j+m)$ has same covering.

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

- maintains the suffix array of a dynamic string
- sequential, $\mathcal{O}(\log^4 n)$ time for queries, $\mathcal{O}(\log^{3+o(1)} n)$ time for updates.
- **maintains dynamic String Synchronizing Sets**

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

- maintains the suffix array of a dynamic string
- sequential, $\mathcal{O}(\log^4 n)$ time for queries, $\mathcal{O}(\log^{3+o(1)} n)$ time for updates.
- **maintains dynamic String Synchronizing Sets**

Sequential Algorithm

- keeps the String Synchronizing Sets and what is used to construct them in memory

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

- maintains the suffix array of a dynamic string
- sequential, $\mathcal{O}(\log^4 n)$ time for queries, $\mathcal{O}(\log^{3+o(1)} n)$ time for updates.
- **maintains dynamic String Synchronizing Sets**

Sequential Algorithm

- keeps the String Synchronizing Sets and what is used to construct them in memory
- updates recalculate parts of this information
  - updates may only affect a small part
- updates String Synchronizing Sets bottom-up
  - uses B^τ to compute $B^{2\tau}$

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

- maintains the suffix array of a dynamic string
- sequential, $\mathcal{O}(\log^4 n)$ time for queries, $\mathcal{O}(\log^{3+o(1)} n)$ time for updates.
- **maintains dynamic String Synchronizing Sets**

Sequential Algorithm

- keeps the String Synchronizing Sets and what is used to construct them in memory
- updates recalculate parts of this information
  - updates may only affect a small part
- updates String Synchronizing Sets bottom-up
  - uses B^τ to compute $B^{2\tau}$

Difficulties for parallel constant-time

- updates have to traverse logarithmic number of levels, but still run in constant time
  - muddling

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

- maintains the suffix array of a dynamic string
- sequential, $\mathcal{O}(\log^4 n)$ time for queries, $\mathcal{O}(\log^{3+o(1)} n)$ time for updates.
- **maintains dynamic String Synchronizing Sets**

Sequential Algorithm

- keeps the String Synchronizing Sets and what is used to construct them in memory
- updates recalculate parts of this information
  - updates may only affect a small part
- updates String Synchronizing Sets bottom-up
  - uses B^τ to compute $B^{2\tau}$

Difficulties for parallel constant-time

- updates have to traverse logarithmic number of levels, but still run in constant time
  - muddling
- need data structures to store sets in constant time and maintain consistent names for occurrences
  - technical, auxiliary algorithms (skipped here)

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.

Time

Level

Set	Record: Delay
B^1	0
B^2	1
B^4	2
B^8	3
B^{16}	4

Delayed Processing

- Sets B^τ are delayed

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.

Delayed Processing

- Sets B^{τ} are delayed
- Changes from an update are not all computed immediately.

Time →

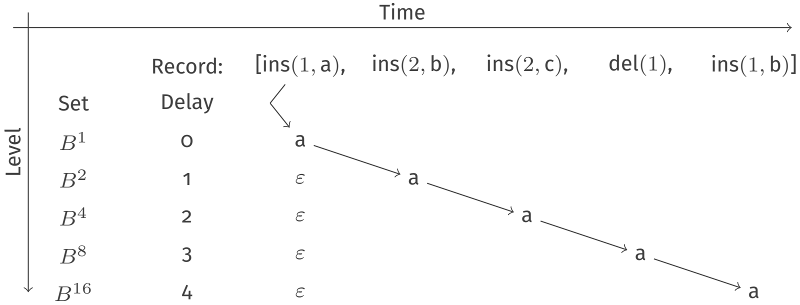
Level ↓

	Record:	[ins$(1, \text{a})$,
Set	Delay	↘
B^1	0	a
B^2	1	ε
B^4	2	ε
B^8	3	ε
B^{16}	4	ε

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.



Delayed Processing

- Sets B^τ are delayed
- Changes from an update are not all computed immediately.
- Spreads computation across following updates.

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.

Time →, Level ↓

Set	Delay	Record: [ins(1, a),	ins(2, b),	ins(2, c),	del(1),	ins(1, b)]
B^1	0	a	ab	acb	ab	bcb
B^2	1	ε	a	ab	acb	ab
B^4	2	ε	ε	a	ab	acb
B^8	3	ε	ε	ε	a	ab
B^{16}	4	ε	ε	ε	ε	a

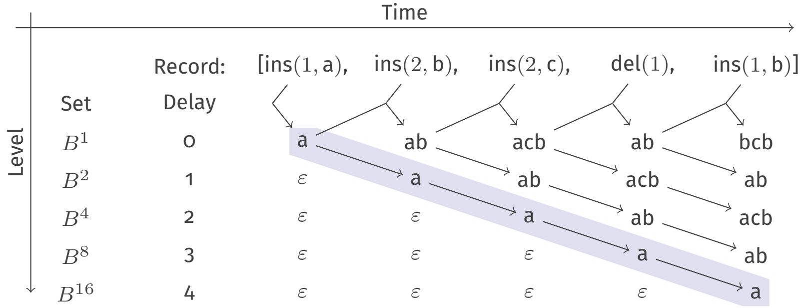
Delayed Processing

- Sets B^τ are delayed
- Changes from an update are not all computed immediately.
- Spreads computation across following updates.
- Computes changes from several updates in parallel.

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.



Delayed Processing

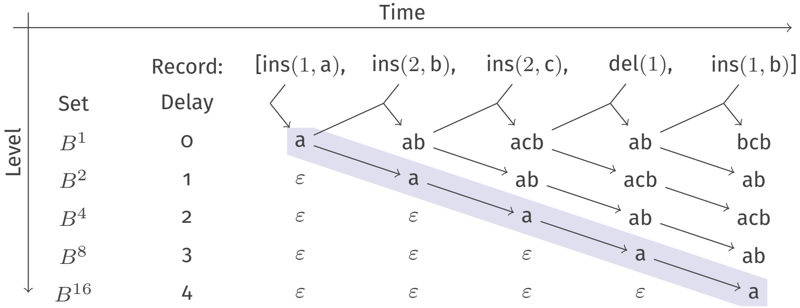
- Sets B^τ are delayed
- Changes from an update are not all computed immediately.
- Spreads computation across following updates.
- Computes changes from several updates in parallel.

$\Rightarrow$ Time available to compute updates grows with number of levels

Achieving Parallel Constant Time

Ideas from the Muddling Lemma [Datta et al. 2019]

- Spread the computation of a time $T(n)$ algorithm across $T(n)$ updates.
- Answer queries with outdated information + a record of recent changes.



Delayed Processing

- Sets B^τ are delayed
- Changes from an update are not all computed immediately.
- Spreads computation across following updates.
- Computes changes from several updates in parallel.

⇒ Time available to compute updates grows with number of levels

Dealing with outdated information

Problem

- Only the topmost String Synchronizing Set B^1 represents the current string.
- The other sets are all delayed.

Set	Delay	Delayed String																					
B^1	0	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a	[a]	b		a
B^2	1	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b	□	a
B^4	2	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	[c]	a		b	c	a
B^8	3	a	b	□	b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a
B^{16}	4	a	b	a	[b]	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a

Record:

Update	Age
ins$(18, \text{a})$	0
del(19)	1
ins$(16, \text{c})$	2
del(3)	3
ins$(4, \text{b})$	4

Dealing with outdated information

Problem

- Only the topmost String Synchronizing Set B^1 represents the current string.
- The other sets are all delayed.

Modified Queries

1. Cut out parts that were recently changed.

Set	Delay	Delayed String																				
B^1	0	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a	a	b		a
B^2	1	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b		a
B^4	2	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b	c	a
B^8	3	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a
B^{16}	4	a	b	a	b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a

Record:

Update	Age
ins$(18, \mathsf{a})$	0
del(19)	1
ins$(16, \mathsf{c})$	2
del(3)	3
ins$(4, \mathsf{b})$	4

Dealing with outdated information

Problem

- Only the topmost String Synchronizing Set B^1 represents the current string.
- The other sets are all delayed.

Modified Queries

1. Cut out parts that were recently changed.
   - Remaining parts are fully up-to-date

Set	Delay	Delayed String																					
B^1	0	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a	a	b		a
B^2	1	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b		a
B^4	2	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b	c	a
B^8	3	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a
B^{16}	4	a	b	a	b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a

Record:

Update	Age
ins$(18, \mathsf{a})$	0
del(19)	1
ins$(16, \mathsf{c})$	2
del(3)	3
ins$(4, \mathsf{b})$	4

Dealing with outdated information

Problem

- Only the topmost String Synchronizing Set B^1 represents the current string.
- The other sets are all delayed.

Modified Queries

1. Cut out parts that were recently changed.
   - Remaining parts are fully up-to-date
2. Answer sub-queries on remaining parts
3. Compare cut-out positions naïvely

Set	Delay	Delayed String																					
B^1	0	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a	a	b		a
B^2	1	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b		a
B^4	2	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b	c	a		b	c	a
B^8	3	a	b		b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a
B^{16}	4	a	b	a	b	c	a	a	b	b	a	b	c	a	a	b	b		a		b	c	a

Record:

Update	Age
ins$(18, \text{a})$	0
del(19)	1
ins$(16, \text{c})$	2
del(3)	3
ins$(4, \text{b})$	4

Summary and Applications

Summary

There is a parallel constant-time algorithm for dynamic LCP on a common CRCW PRAM with $\mathcal{O}(n^{\varepsilon})$ work, for any $\varepsilon > 0$.

- Combines techniques from Dyn-FO (Muddling) and String-Algorithms (String Synchronizing Sets)
- Only for integer alphabets $\Sigma = [1, n]$ (common assumption for string algorithms)

Summary and Applications

Summary

There is a parallel constant-time algorithm for dynamic LCP on a common CRCW PRAM with $\mathcal{O}(n^\varepsilon)$ work, for any $\varepsilon > 0$.

- Combines techniques from Dyn-FO (Muddling) and String-Algorithms (String Synchronizing Sets)
- Only for integer alphabets $\Sigma = [1, n]$ (common assumption for string algorithms)

Squares

$S[i, j]$ is a square if $S[i, j] = uu$

Input	String $S[1, n]$
Set(i, σ)	Set $S' = S[1, i-1]\sigma S[i+1, n]$
Query$()$	What is the longest square in S?

Based on Amir et al. 2019, Theorem 26:

Parallel constant-time algorithm with work $\mathcal{O}(n^\varepsilon)$, for any $\varepsilon > 0$.

Summary and Applications

Summary

There is a parallel constant-time algorithm for dynamic LCP on a common CRCW PRAM with $\mathcal{O}(n^\varepsilon)$ work, for any $\varepsilon > 0$.

- Combines techniques from Dyn-FO (Muddling) and String-Algorithms (String Synchronizing Sets)
- Only for integer alphabets $\Sigma = [1, n]$ (common assumption for string algorithms)

Squares

$S[i, j]$ is a square if $S[i, j] = uu$

Input	String $S[1, n]$
Set(i, σ)	Set $S' = S[1, i-1]\sigma S[i+1, n]$
Query$()$	What is the longest square in S?

Based on Amir et al. 2019, Theorem 26:

Parallel constant-time algorithm with work $\mathcal{O}(n^\varepsilon)$, for any $\varepsilon > 0$.

Dyck Languages D_k

D_k: well-matched parentheses with k types

Input	String $S[1, n]$
Set(i, σ)	Set $S' = S[1, i-1]\sigma S[i+1, n]$
Query$()$	Is $S \in D_k$?

Due to Schmidt et al. 2021, Lemma 6.9:

Parallel constant-time algorithms with work $\mathcal{O}(n^\varepsilon)$, for D_k, $k \in \mathbb{N}$ and any $\varepsilon > 0$.

Bibliography I

Amir, Amihood, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky (2019). "Repetition Detection in a Dynamic String". In: *ESA 2019*. Leibniz International Proceedings in Informatics (LIPIcs) 144, 5:1–5:18.

Datta, Samir, Anish Mukherjee, Thomas Schwentick, Nils Vortmeier, and Thomas Zeume (May 2019). "A Strategy for Dynamic Programs: Start over and Muddle Through". In: *Logical Methods in Computer Science* Volume 15, Issue 2.

Farach-Colton, Martin, Paolo Ferragina, and S. Muthukrishnan (Nov. 2000). "On the Sorting-Complexity of Suffix Tree Construction". In: *J. ACM* 47.6, pp. 987–1011.

Gawrychowski, Paweł, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski (Jan. 2018). "Optimal Dynamic Strings". In: *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Proceedings. Society for Industrial and Applied Mathematics, pp. 1509–1528.

Immerman, Neil (1999). *Descriptive complexity*. New York u.a.: Springer.

Kempa, Dominik and Tomasz Kociumaka (June 2022). "Dynamic Suffix Array with Polylogarithmic Queries and Updates". In: *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 1657–1670.

Kociumaka, Tomasz, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń (Oct. 2024). "Internal Pattern Matching Queries in a Text and Applications". In: *SIAM Journal on Computing* 53.5, pp. 1524–1577.

Bibliography II

Lipták, Zsuzsanna, Francesco Masillo, and Gonzalo Navarro (Aug. 14, 2024). *A Textbook Solution for Dynamic Strings*. URL: http://arxiv.org/abs/2403.13162 (visited on 05/12/2025). Pre-published.

Patnaik, Sushant and Neil Immerman (Oct. 1997). "Dyn-FO: A Parallel, Dynamic Complexity Class". In: *Journal of Computer and System Sciences* 55.2, pp. 199–209.

Schmidt, Jonas, Thomas Schwentick, Till Tantau, Nils Vortmeier, and Thomas Zeume (2021). "Work-Sensitive Dynamic Complexity of Formal Languages". In: *Foundations of Software Science and Computation Structures*. Ed. by Stefan Kiefer and Christine Tasson. Cham: Springer International Publishing, pp. 490–509.

Shun, Julian (Nov. 2014). "Fast Parallel Computation of Longest Common Prefixes". In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 387–398.

Answering Queries with String Synchronizing Sets (1)

$\Rightarrow$ Compute lcp(i, j) via search on m similar to a binary search, using equality queries "$S[i, i+m) = S[j, j+m)$?"

Answering Queries with String Synchronizing Sets (1)

$\Rightarrow$ Compute lcp(i, j) via search on m similar to a binary search, using equality queries "$S[i, i+m) = S[j, j+m)$?"

Comparing Segments $S[i, i+m)$ and $S[j, j+m)$

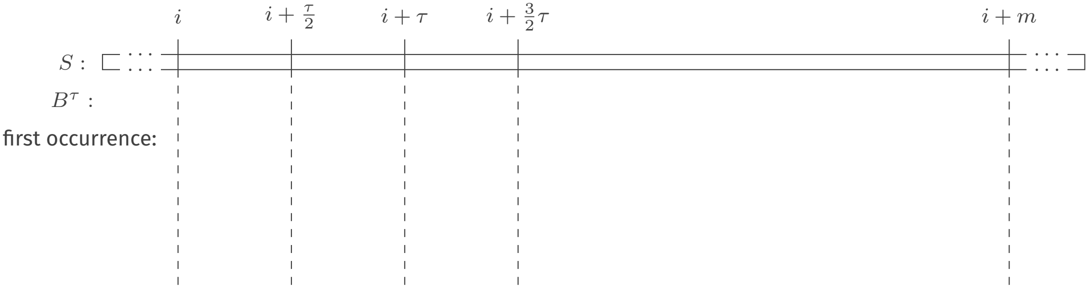
1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences from $B^1, B^2, B^4, \ldots$
2. If $S[i, i+m) = S[j, j+m)$, then $S[j, j+m)$ has the same covering (due to *Consistency*)

Answering Queries with String Synchronizing Sets (1)

⇒ Compute lcp(i, j) via search on m similar to a binary search, using equality queries "$S[i, i+m) = S[j, j+m)$?"

Comparing Segments $S[i, i+m)$ and $S[j, j+m)$

1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences from $B^1, B^2, B^4, \ldots$
2. If $S[i, i+m) = S[j, j+m)$, then $S[j, j+m)$ has the same covering (due to *Consistency*)

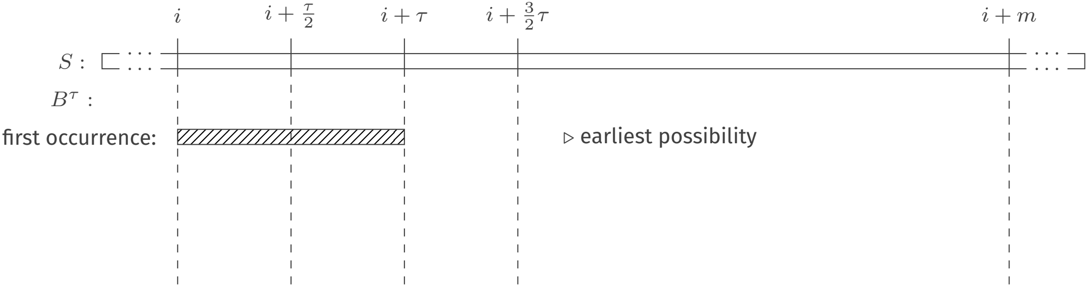


Answering Queries with String Synchronizing Sets (1)

⇒ Compute $\mathrm{lcp}(i, j)$ via search on m similar to a binary search, using equality queries “$S[i, i+m) = S[j, j+m)$?”

Comparing Segments $S[i, i+m)$ and $S[j, j+m)$

1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences from $B^1, B^2, B^4, \ldots$
2. If $S[i, i+m) = S[j, j+m)$, then $S[j, j+m)$ has the same covering (due to *Consistency*)

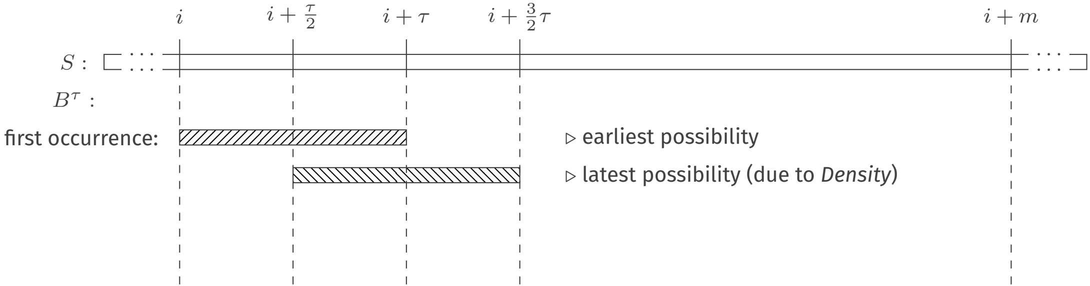


Answering Queries with String Synchronizing Sets (1)

⇒ Compute lcp(i, j) via search on m similar to a binary search, using equality queries "$S[i, i+m) = S[j, j+m)$?"

Comparing Segments $S[i, i+m)$ and $S[j, j+m)$

1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences from $B^1, B^2, B^4, \ldots$
2. If $S[i, i+m) = S[j, j+m)$, then $S[j, j+m)$ has the same covering (due to *Consistency*)

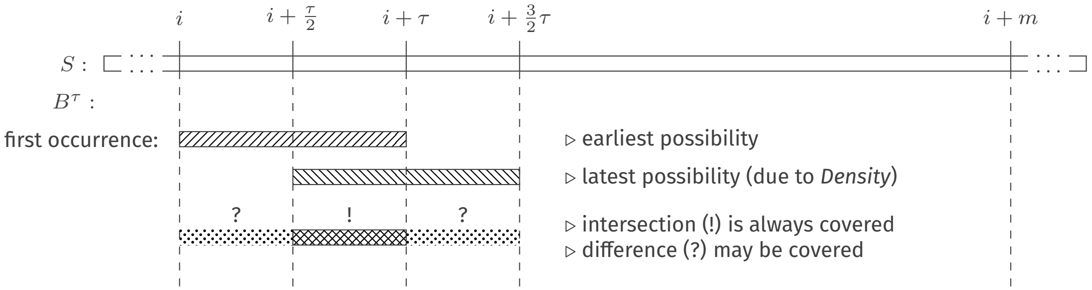


Answering Queries with String Synchronizing Sets (1)

⇒ Compute lcp(i, j) via search on m similar to a binary search, using equality queries "$S[i, i+m) = S[j, j+m)$?"

Comparing Segments $S[i, i+m)$ and $S[j, j+m)$

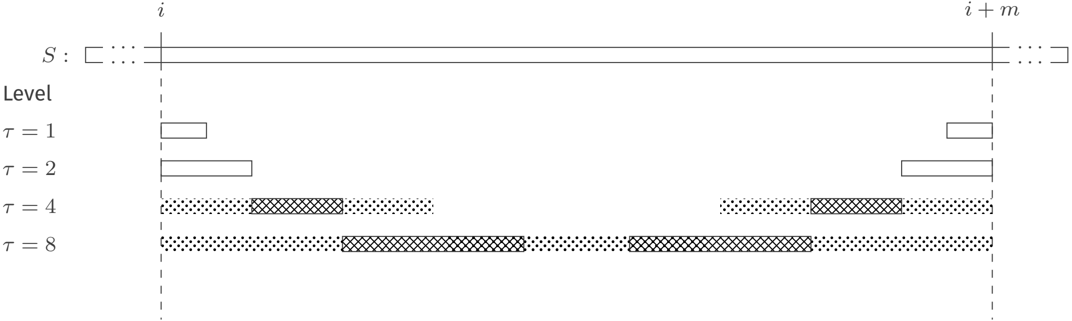
1. Cover $S[i, i+m)$ with few ($\mathcal{O}(\log n)$) occurrences from $B^1, B^2, B^4, \ldots$
2. If $S[i, i+m) = S[j, j+m)$, then $S[j, j+m)$ has the same covering (due to *Consistency*)



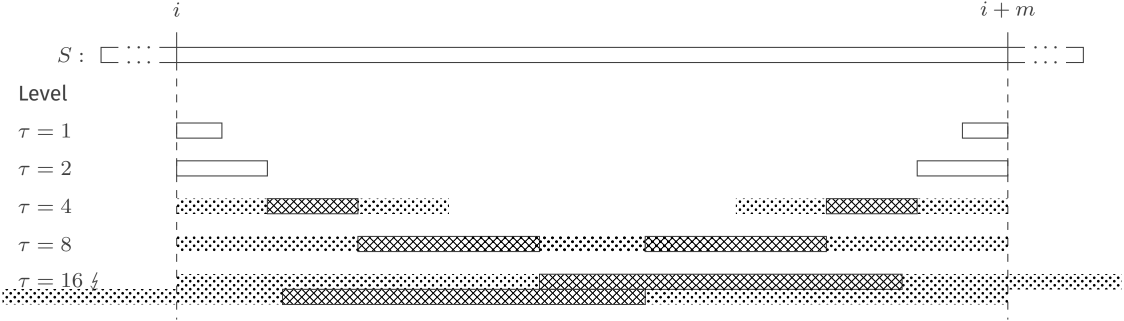
Answering Queries with String Synchronizing Sets (2)

Answering Queries with String Synchronizing Sets (2)

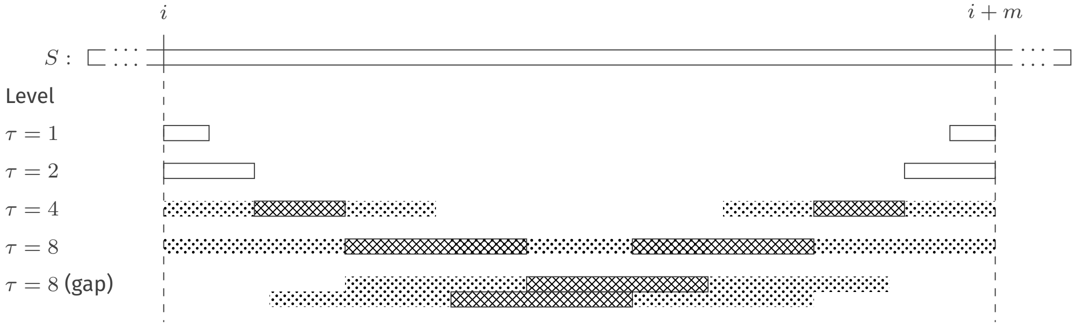
Answering Queries with String Synchronizing Sets (2)



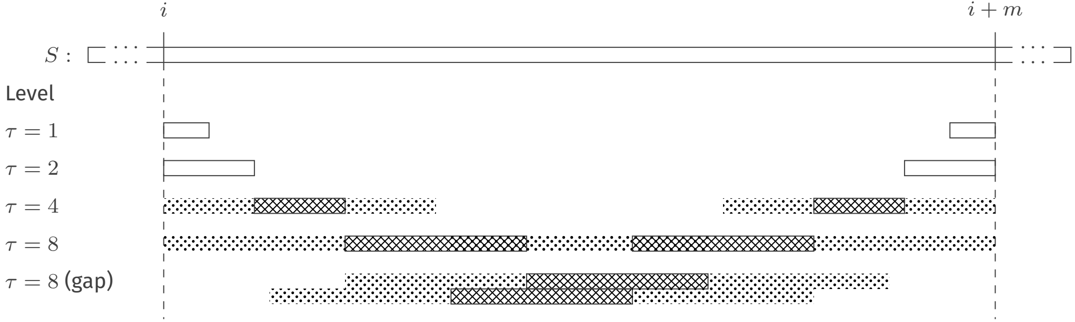
Answering Queries with String Synchronizing Sets (2)



Answering Queries with String Synchronizing Sets (2)



Answering Queries with String Synchronizing Sets (2)

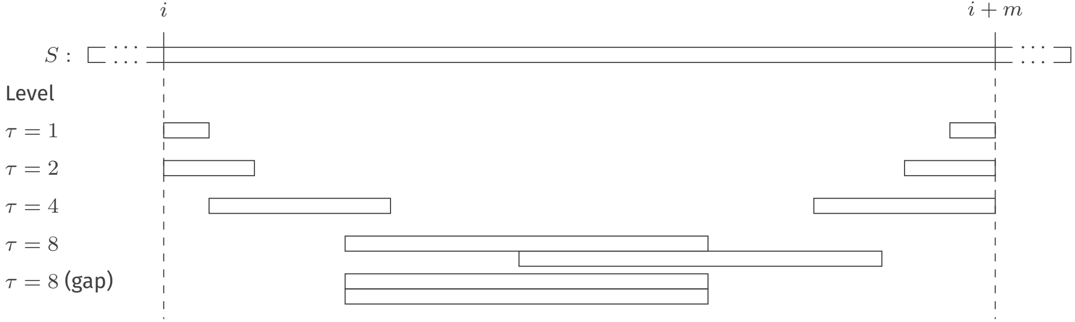


Correctness

- Positions in $S[i, i+m)$ are always covered
- Position outside $S[i, i+m)$ are never covered

$\Rightarrow S[i, j) = S[j, m)$ iff both produce the same covering

Answering Queries with String Synchronizing Sets (2)



Correctness

- Positions in $S[i, i+m)$ are always covered
- Position outside $S[i, i+m)$ are never covered

$\Rightarrow S[i,j) = S[j,m)$ iff both produce the same covering

Towards Constructing String Synchronizing Sets

Consistent Decomposition D

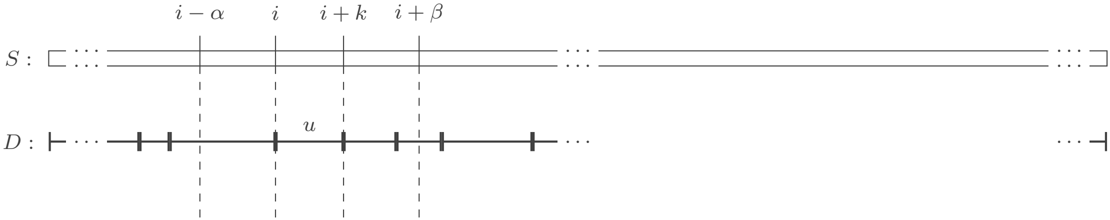
Decompose S into factors $S = u_1 u_2 \dots u_m$ with similar properties to String Synchronizing Sets

Towards Constructing String Synchronizing Sets

Consistent Decomposition D

Decompose S into factors $S = u_1 u_2 \ldots u_m$ with similar properties to String Synchronizing Sets

Consistency: If $S[i, i+k) = u$ is a factor and $S[i-\alpha, i+\beta) = S[j-\alpha, j+\beta]$, then $S[j, j+m) = u$ is a factor

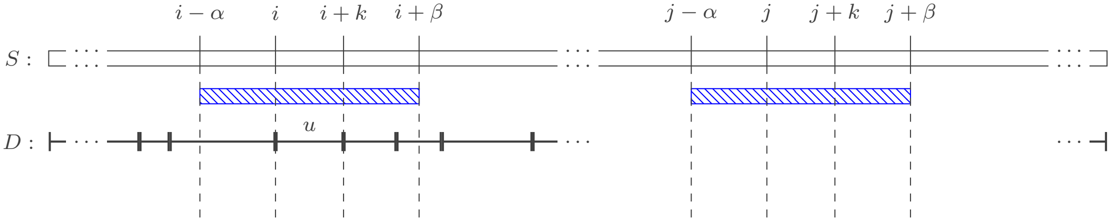


Towards Constructing String Synchronizing Sets

Consistent Decomposition D

Decompose S into factors $S = u_1 u_2 \dots u_m$ with similar properties to String Synchronizing Sets

Consistency: If $S[i, i+k) = u$ is a factor and $S[i-\alpha, i+\beta) = S[j-\alpha, j+\beta]$, then $S[j, j+m) = u$ is a factor

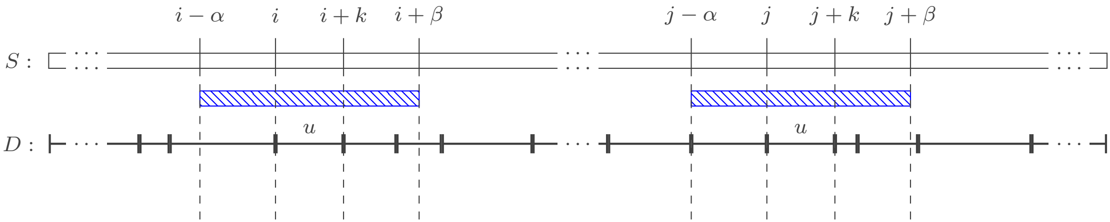


Towards Constructing String Synchronizing Sets

Consistent Decomposition D

Decompose S into factors $S = u_1 u_2 \dots u_m$ with similar properties to String Synchronizing Sets

Consistency: If $S[i, i+k) = u$ is a factor and $S[i-\alpha, i+\beta) = S[j-\alpha, j+\beta]$, then $S[j, j+m) = u$ is a factor



Towards Constructing String Synchronizing Sets

Consistent Decomposition D

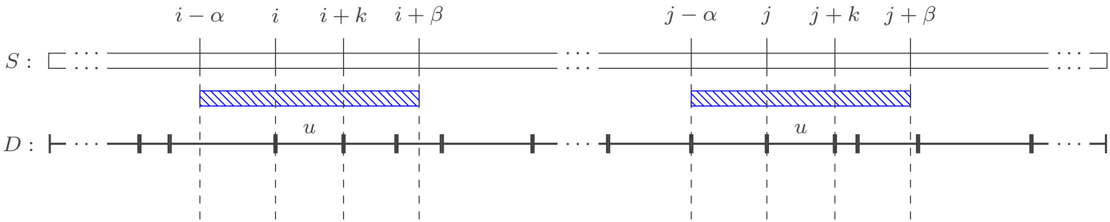
Decompose S into factors $S = u_1 u_2 \dots u_m$ with similar properties to String Synchronizing Sets

Consistency: If $S[i, i+k) = u$ is a factor and $S[i-\alpha, i+\beta) = S[j-\alpha, j+\beta]$, then $S[j, j+m) = u$ is a factor

Density: Max length of factors limited*

Sparseness: Short factors can only exist surrounded by long factors

*except periodic (repeating) parts



Towards Constructing String Synchronizing Sets

Consistent Decomposition D

Decompose S into factors $S = u_1 u_2 \dots u_m$ with similar properties to String Synchronizing Sets

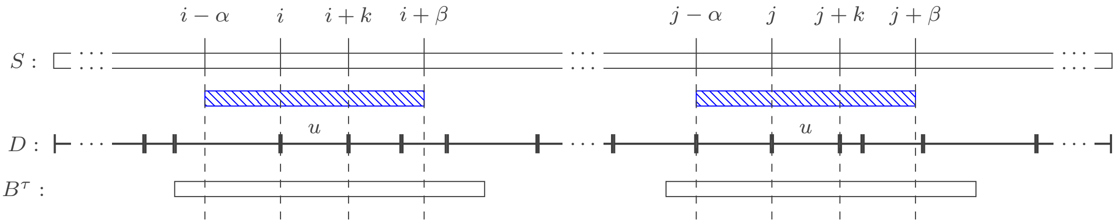
Consistency: If $S[i, i+k) = u$ is a factor and $S[i-\alpha, i+\beta) = S[j-\alpha, j+\beta]$, then $S[j, j+m) = u$ is a factor

Density: Max length of factors limited*

*except periodic (repeating) parts

Sparseness: Short factors can only exist surrounded by long factors

$\Rightarrow$ Turn each factor in D into an occurrence in B^τ



Computing Consistent Decompositions

Iterative Construction

1. Start with singleton factors

Example

$S =$ a b a b c a a b b a b c a a b b a b c b

$D^1 =$ 

Computing Consistent Decompositions

Iterative Construction

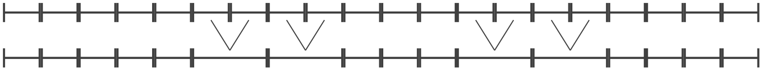
1. Start with singleton factors
2. Repeat until only one factor is left:
   2.1 Merge runs of identical factors

Example

$S =$ a b a b c a a b b a b c a a b b a b c b

$D^1 =$

$D^1_r =$



Computing Consistent Decompositions

Iterative Construction

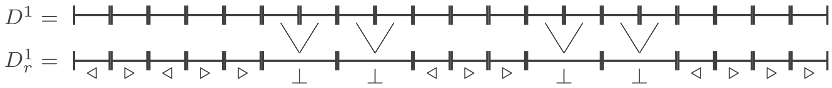
1. Start with singleton factors
2. Repeat until only one factor is left:
   - 2.1 Merge runs of identical factors
   - 2.2 Merge neighboring factors consistently

Consistent Merging

- Partition unique factors into $\triangleleft, \triangleright, \bot$.

Example

$S =$ a b a b c a a b b a b c a a b b a b c b



$\triangleleft = \{\mathrm{a}\}$
$\triangleright = \{\mathrm{b}, \mathrm{c}\}$
$\bot = \{\mathrm{aa}, \mathrm{bb}\}$

Computing Consistent Decompositions

Iterative Construction

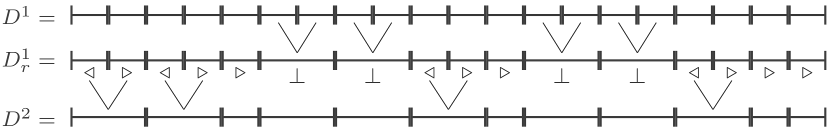
1. Start with singleton factors
2. Repeat until only one factor is left:
   2.1 Merge runs of identical factors
   2.2 Merge neighboring factors consistently

Consistent Merging

- Partition unique factors into $\triangleleft, \triangleright, \bot$.
- Merge $u_i u_{i+1}$ iff $u_i \in \triangleleft, u_{i+1} \in \triangleright$.

Example

$S =$ a b a b c a a b b a b c a a b b a b c b



$\triangleleft = \{\text{a}\}$
$\triangleright = \{\text{b}, \text{c}\}$
$\bot = \{\text{aa}, \text{bb}\}$

Computing Consistent Decompositions

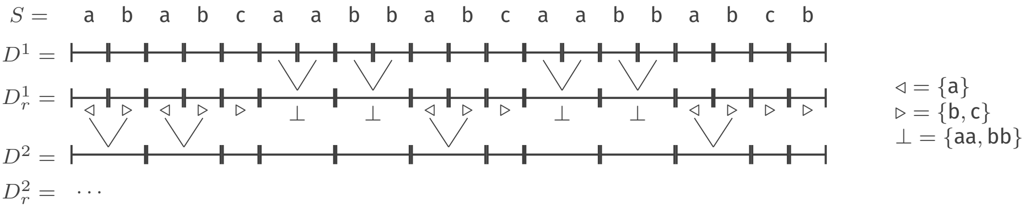
Iterative Construction

1. Start with singleton factors
2. Repeat until only one factor is left:
   - 2.1 Merge runs of identical factors
   - 2.2 Merge neighboring factors consistently

Consistent Merging

- Partition unique factors into $\triangleleft, \triangleright, \perp$.
- Merge $u_i u_{i+1}$ iff $u_i \in \triangleleft, u_{i+1} \in \triangleright$.
- Choose Partition…
  - At random (e.g. Lipták, Masillo, and Navarro 2024)
  - High number of merges (e.g. Kociumaka et al. 2024)

Example



Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts.

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. $\rightarrow$ Local Merging

Local Merging

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. $\rightarrow$ Local Merging

Local Merging

For each k, there are fixed α_k, β_k with:

- During the k-th merging step,

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. → Local Merging

Local Merging

For each k, there are fixed α_k, β_k with:

- During the k-th merging step,
- When deciding if u_i, u_{i+1} should be merged,
  - where j is the position where u_i starts,

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. $\rightarrow$ Local Merging

Local Merging

For each k, there are fixed α_k, β_k with:

- During the k-th merging step,
- When deciding if u_i, u_{i+1} should be merged,
  - where j is the position where u_i starts,
- This decision depends only on $S[j - \alpha_k, j + \beta_k]$.

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. $\rightarrow$ Local Merging
2. The affected parts of the string contain few factors/occurrences. $\rightarrow$ Local Sparseness

Local Merging

For each k, there are fixed α_k, β_k with:

- During the k-th merging step,
- When deciding if u_i, u_{i+1} should be merged,
  - where j is the position where u_i starts,
- This decision depends only on $S[j - \alpha_k, j + \beta_k]$.

Local Sparseness

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. $\rightarrow$ Local Merging
2. The affected parts of the string contain few factors/occurrences. $\rightarrow$ Local Sparseness

Local Merging

For each k, there are fixed α_k, β_k with:

- During the k-th merging step,
- When deciding if u_i, u_{i+1} should be merged,
  - where j is the position where u_i starts,
- This decision depends only on $S[j - \alpha_k, j + \beta_k]$.

Local Sparseness

In the definition of *Sparseness*,

replace "The total length of all occurrences is $\mathcal{O}(n)$"

with "The total length of all occurrences in a length $m > \tau$ segment is $\mathcal{O}(m \log^* m)$"

Maintaining String Synchronizing Sets

Sequential Dynamic Algorithm by Kempa and Kociumaka 2022

Keep decompositions and synchronizing sets in memory. On updates, recalculate affected parts. *Requires:*

1. Updates affect factors/occurrences in a small part of the string. $\rightarrow$ Local Merging
2. The affected parts of the string contain few factors/occurrences. $\rightarrow$ Local Sparseness

Local Merging

For each k, there are fixed α_k, β_k with:

- During the k-th merging step,
- When deciding if u_i, u_{i+1} should be merged,
  - where j is the position where u_i starts,
- This decision depends only on $S[j - \alpha_k, j + \beta_k]$.

Local Sparseness

In the definition of *Sparseness*,

replace "The total length of all occurrences is $\mathcal{O}(n)$"

with "The total length of all occurrences in a length $m > \tau$ segment is $\mathcal{O}(m \log^* m)$"

---

stronger: Sparseness holds within small segments

weaker: More occurrences by $\log^*$-factor

String Synchronizing Sets Hierarchy

$S =$ a b a b c a a b b a b c a a b b a b c b

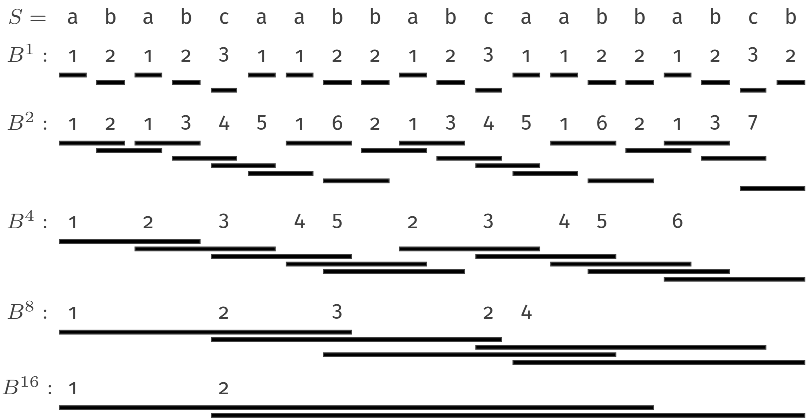
B^4 : 1 2 3 4 5 2 3 4 5 6

String Synchronizing Sets Hierarchy

Hierarchy

$B^1, B^2, B^4, B^8, B^{16}$

B^τ for all
$\tau = 2^i, 0 \leq i \leq \log n$

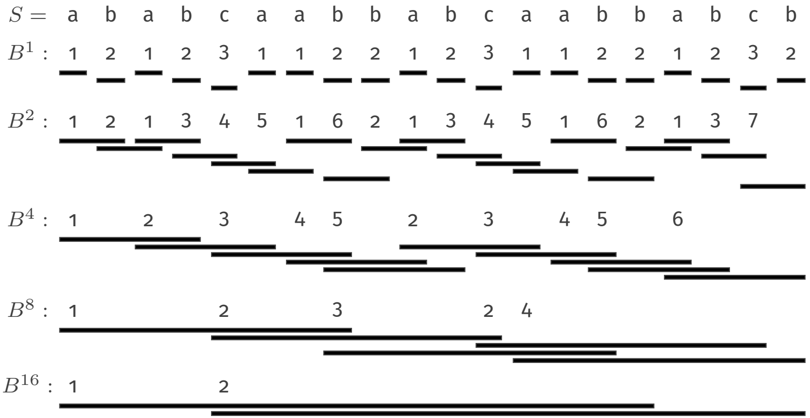


String Synchronizing Sets Hierarchy

Hierarchy

$B^1, B^2, B^4, B^8, B^{16}$

B^τ for all $\tau = 2^i, 0 \leq i \leq \log n$



Observation 1

$B^1 = [1, n]$
$B^2 = [1, n-1]$
due to *Density*