The surprising algebra of string comparison

Alexander Tiskin and Boris Zolotov

Department of Mathematics and Computer Science, St Petersburg University

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Introduction

Overview

Longest common subsequence under string concatenation

$lcs(\text{“RUMPLESTILTSKIN”}, \text{“STEAK”}) = 3$
$lcs(\text{“RUMPLESTILTSKIN”}, \text{“STILTON”}) = 6$

$\rightsquigarrow lcs(\text{“RUMPLESTILTSKIN”}, \text{“STEAK”} + \text{“STILTON”}) = ?$

Standard approach: dynamic programming

Divide-and-conquer as an alternative?

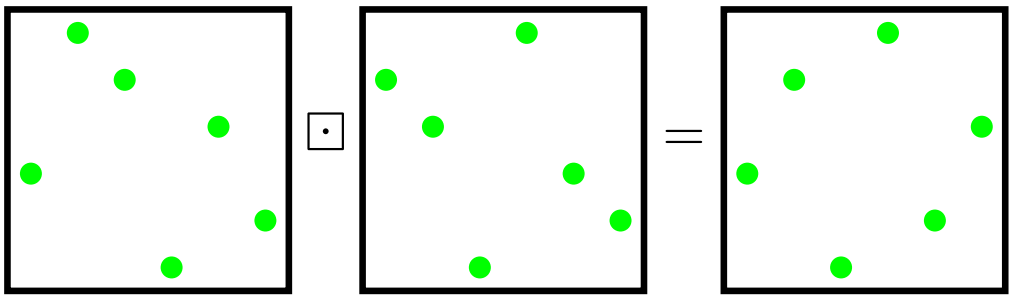
Introduction

Overview

Unit-Monge matrices under distance (a.k.a. tropical) multiplication

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 1 & 1 & 2 & 3 & 4 \\ 0 & 1 & 1 & 1 & 2 & 2 & 3 \\ 0 & 0 & 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \odot \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 3 & 4 & 5 \\ 0 & 0 & 1 & 2 & 2 & 3 & 4 \\ 0 & 0 & 0 & 1 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 1 & 2 & 3 & 3 & 4 & 5 \\ 0 & 1 & 1 & 2 & 2 & 3 & 4 \\ 0 & 1 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

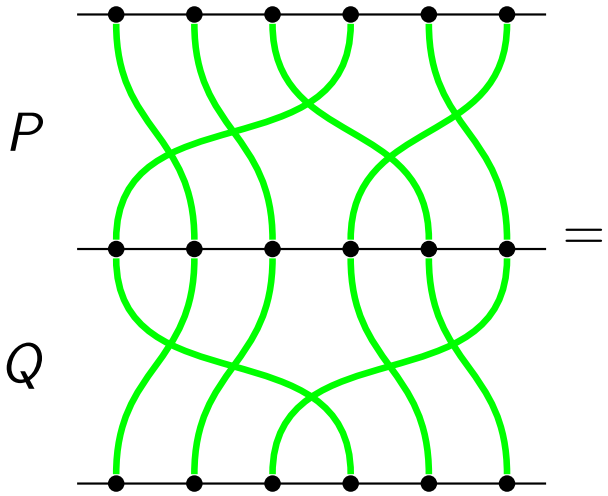
Permutation matrices under sticky multiplication



Introduction

Overview

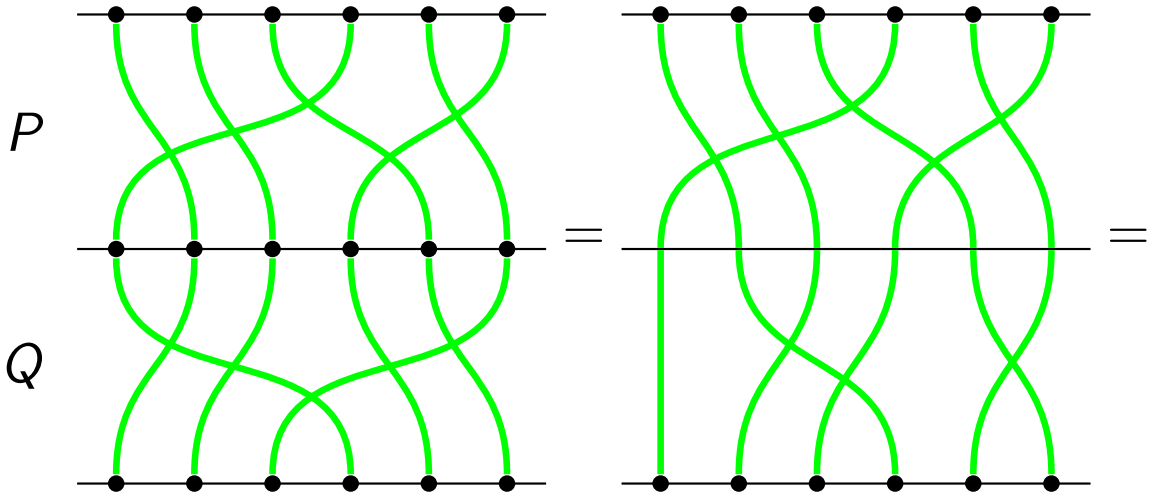
Sticky braids (a.k.a. Hecke words)



Introduction

Overview

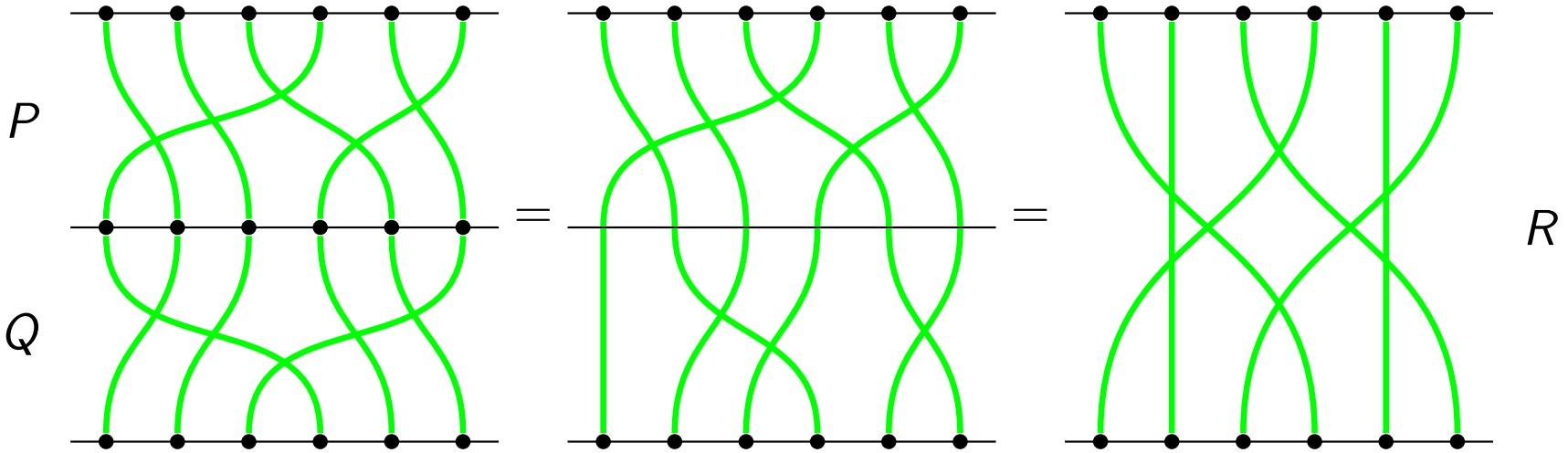
Sticky braids (a.k.a. Hecke words)



Introduction

Overview

Sticky braids (a.k.a. Hecke words)



Introduction

Overview

Striking connection between these seemingly unrelated structures:

- behaviour of LCS length under string concatenation
- distance multiplication of unit-Monge matrices = sticky multiplication of permutation matrices
- multiplication of sticky braids

These structures:

- are isomorphic monoids with a deep algebraic meaning
- admit a fast multiplication algorithm
- have far-reaching algorithmic applications
- connected to computational geometry, combinatorics, statistical mechanics...
- have applications in software, data storage, bioinformatics...

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Unit-Monge matrices and sticky braids

Monge matrices

Dominance-sum matrix (a.k.a. distribution matrix) of matrix D:
$D^{\Sigma}[i,j] = \sum_{\hat{\imath}>i,\hat{\jmath}<j} D\langle\hat{\imath},\hat{\jmath}\rangle$

Cross-difference matrix (a.k.a. density matrix) of matrix E:
$E^{\square}\langle\hat{\imath},\hat{\jmath}\rangle = E[\hat{\imath}^{-},\hat{\jmath}^{+}] - E[\hat{\imath}^{-},\hat{\jmath}^{-}] - E[\hat{\imath}^{+},\hat{\jmath}^{+}] + E[\hat{\imath}^{+},\hat{\jmath}^{-}]$

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^{\square} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

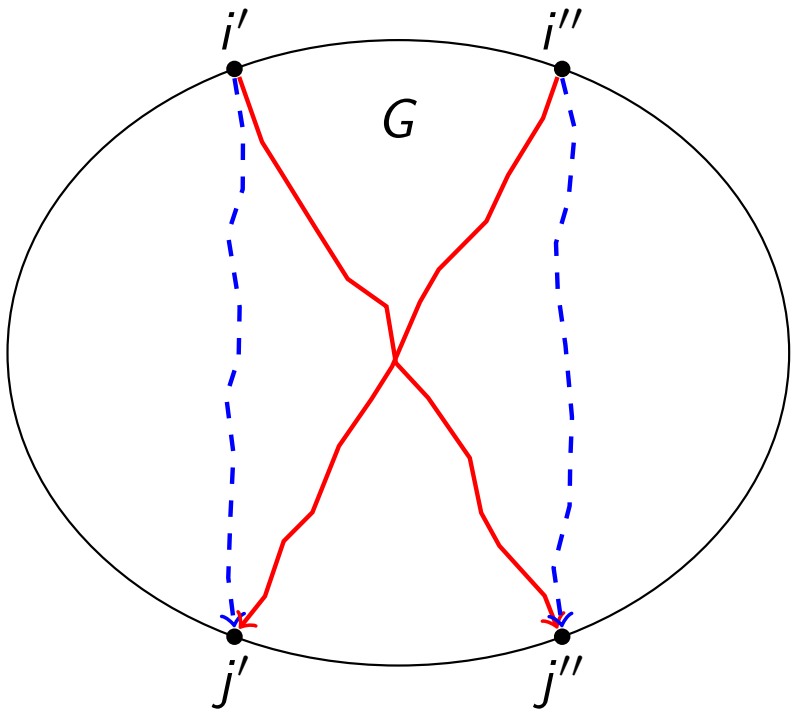
$(D^{\Sigma})^{\square} = D$ for all D

Matrix E is simple, if $(E^{\square})^{\Sigma} = E$: only zeros in left column and bottom row

Matrix E is Monge, if $E^{\square}$ is nonnegative

Intuition: boundary-to-boundary distances in a (weighted) planar graph

G. Monge (1746–1818)



$$E[i',j'] + E[i'',j''] \le E[i',j''] + E[i'',j']$$

Unit-Monge matrices and sticky braids

Unit-Monge matrices

Matrix E is unit-Monge, if $E^{\square}$ is a permutation matrix

Intuition: boundary-to-boundary distances in a grid-like graph (in particular, the LCS/alignment grid for a pair of strings)

Matrix E is unit-Monge, if $E^{\square}$ is a permutation matrix

Intuition: boundary-to-boundary distances in a grid-like graph (in particular, the LCS/alignment grid for a pair of strings)

Simple unit-Monge matrix (a.k.a. "rank function"): P^{Σ}, where P is a permutation matrix

P used as implicit representation of P^{Σ}

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Tropical semiring (a.k.a. (min, +)-semiring, distance semiring)

- addition $\oplus$ given by min
- multiplication $\odot$ given by $+$

Matrix tropical multiplication

$A \odot B = C \qquad C[i,k] = \bigoplus_j \big(A[i,j] \odot B[j,k]\big) = \min_j \big(A[i,j] + B[j,k]\big)$

Intuition: shortest path distances in weighted graphs

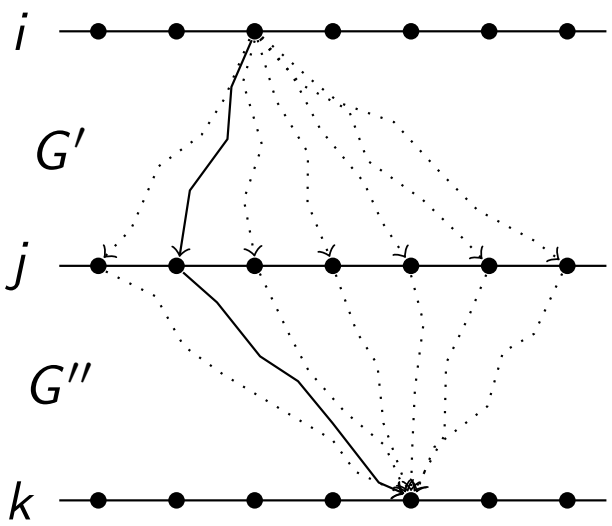
Unit-Monge matrices and sticky braids

Sticky multiplication

Matrix classes closed under $\odot$-multiplication (for given n):

- general (integer, real) matrices $\sim$ general weighted graphs
- Monge matrices $\sim$ planar weighted graphs
- simple unit-Monge matrices $\sim$ grid-like graphs

Intuition: gluing distances in a composition of graphs



Unit-Monge matrices and sticky braids

Sticky multiplication

Recall: permutation matrices = implicit simple unit-Monge matrices

Matrix sticky multiplication (implicit tropical multiplication)

$P \boxdot Q = R$ iff $P^\Sigma \odot Q^\Sigma = R^\Sigma$

Unit-Monge monoid $\mathbb{H}_n$

- permutation matrices under $\boxdot$
- simple unit-Monge matrices under $\odot$

Isomorphic to the Hecke monoid

Multiplication in $\mathbb{H}_n$

Given permutation matrices P, Q, obtain $P \boxdot Q = R$

Multiplication in $\mathbb{H}_n$

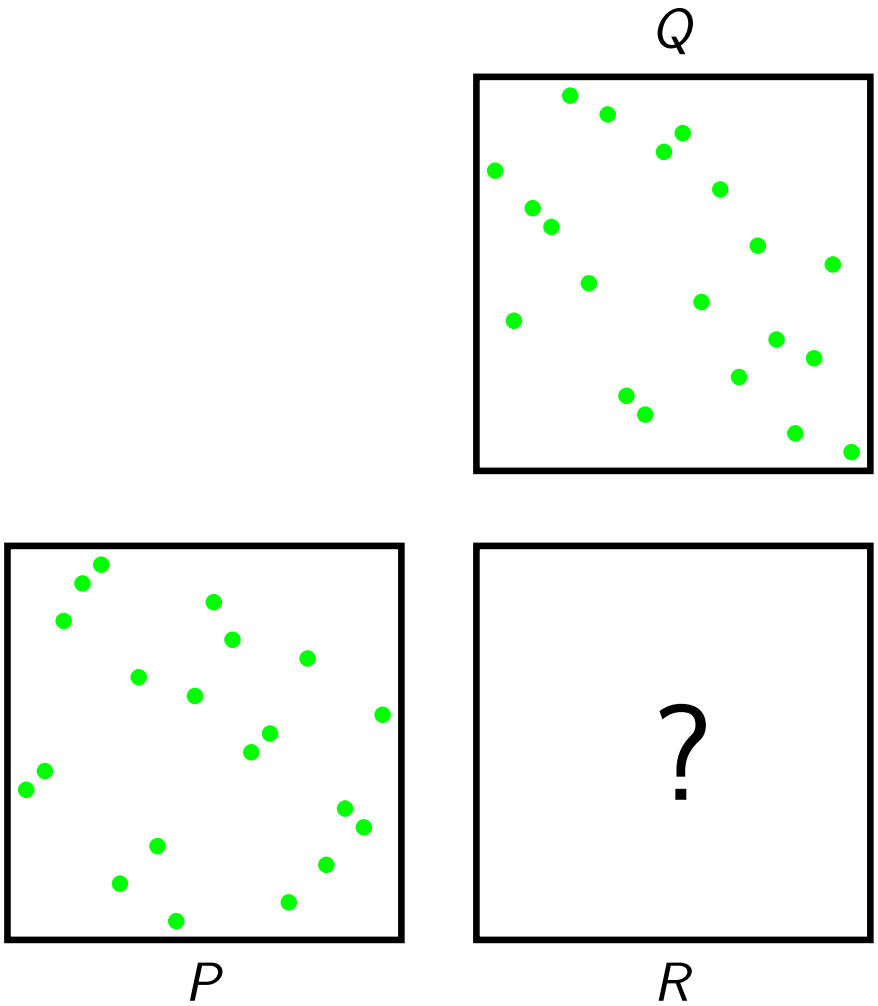
Given permutation matrices P, Q, obtain $P \boxdot Q = R$

Tropical multiplication and multiplication in $\mathbb{H}_n$: running time

type	time	
general $\odot$	$O(n^3)$	standard
	$O\left(\frac{n^3(\log\log n)^3}{\log^2 n}\right)$	[Chan: 2007]
Monge $\odot$	$O(n^2)$	via [Aggarwal+: 1987]
permutation $\boxdot$	$O(n^{1.5})$	[T: 2006]
	$O(n \log n)$	[T: 2010]

Unit-Monge matrices and sticky braids

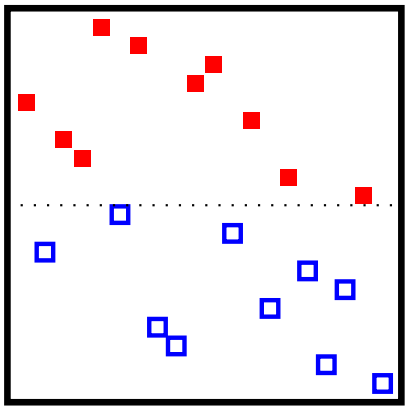
Sticky multiplication

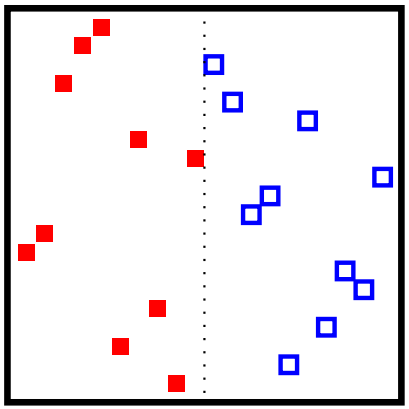


Unit-Monge matrices and sticky braids

Sticky multiplication

Q_{lo}, Q_{hi}



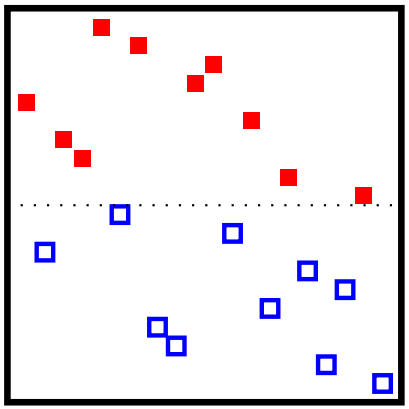


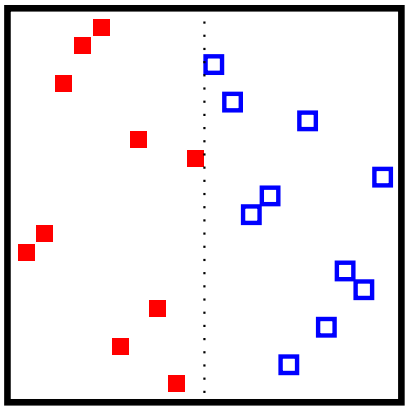
P_{lo}, P_{hi}

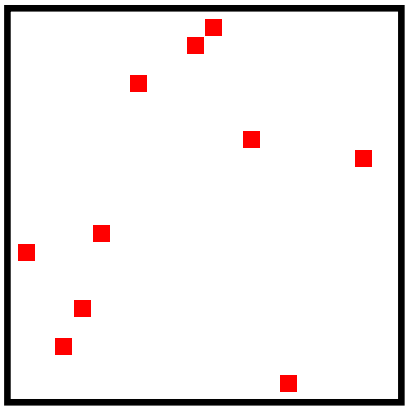
Unit-Monge matrices and sticky braids

Sticky multiplication

Q_{lo}, Q_{hi}



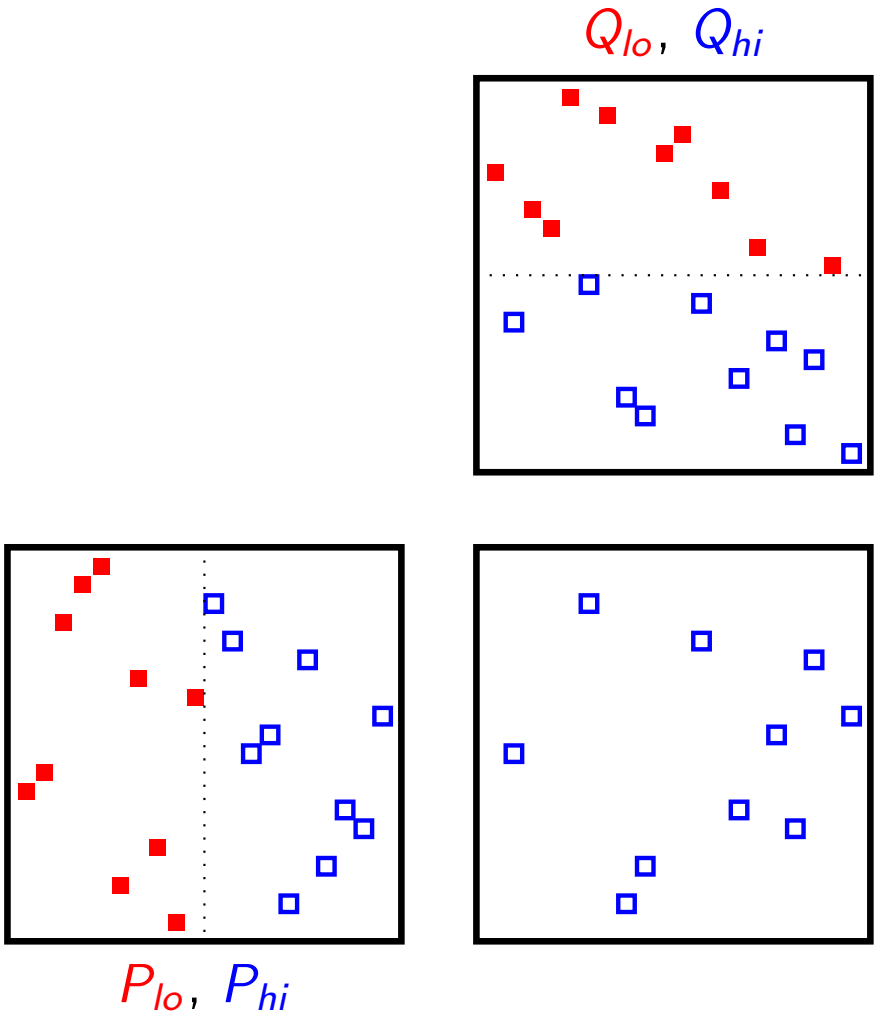




P_{lo}, P_{hi}

Unit-Monge matrices and sticky braids

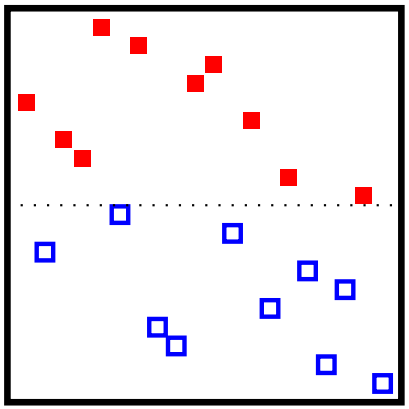
Sticky multiplication

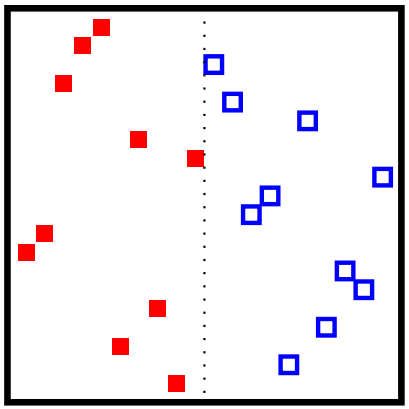


Unit-Monge matrices and sticky braids

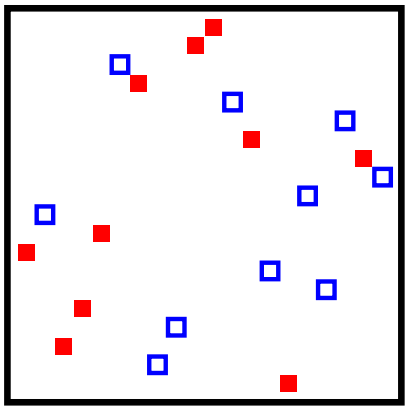
Sticky multiplication

Q_{lo}, Q_{hi}





P_{lo}, P_{hi}

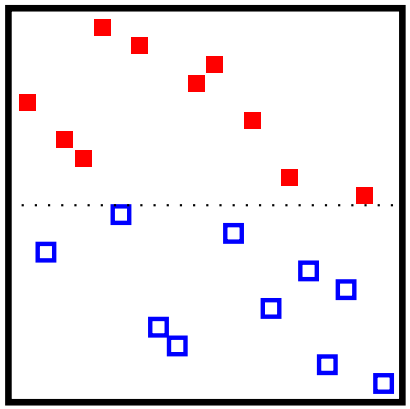


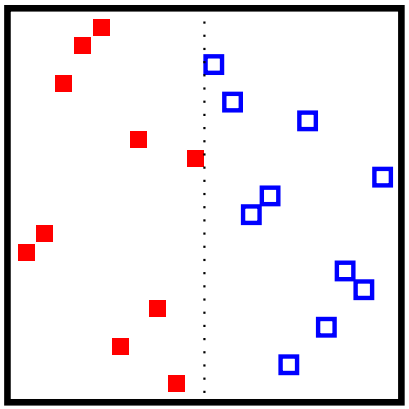
$R_{lo} + R_{hi}$

Unit-Monge matrices and sticky braids

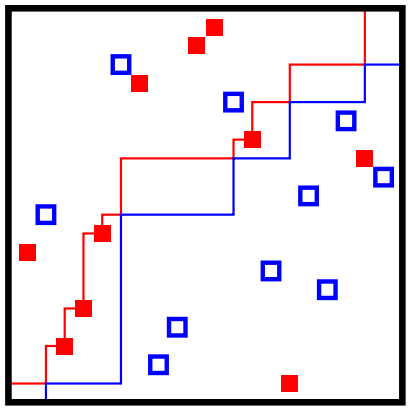
Sticky multiplication

Q_{lo}, Q_{hi}





P_{lo}, P_{hi}

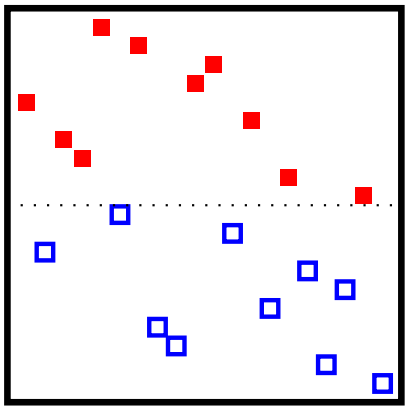


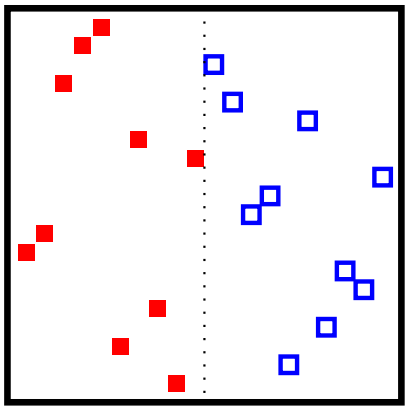
$R_{lo} + R_{hi}$

Unit-Monge matrices and sticky braids

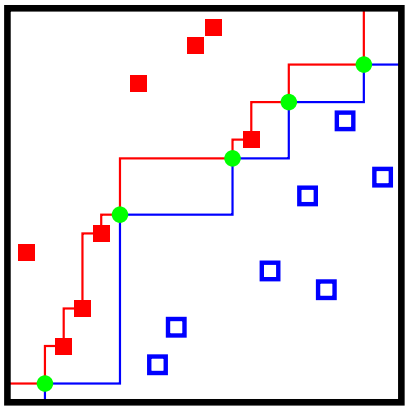
Sticky multiplication

Q_{lo}, Q_{hi}





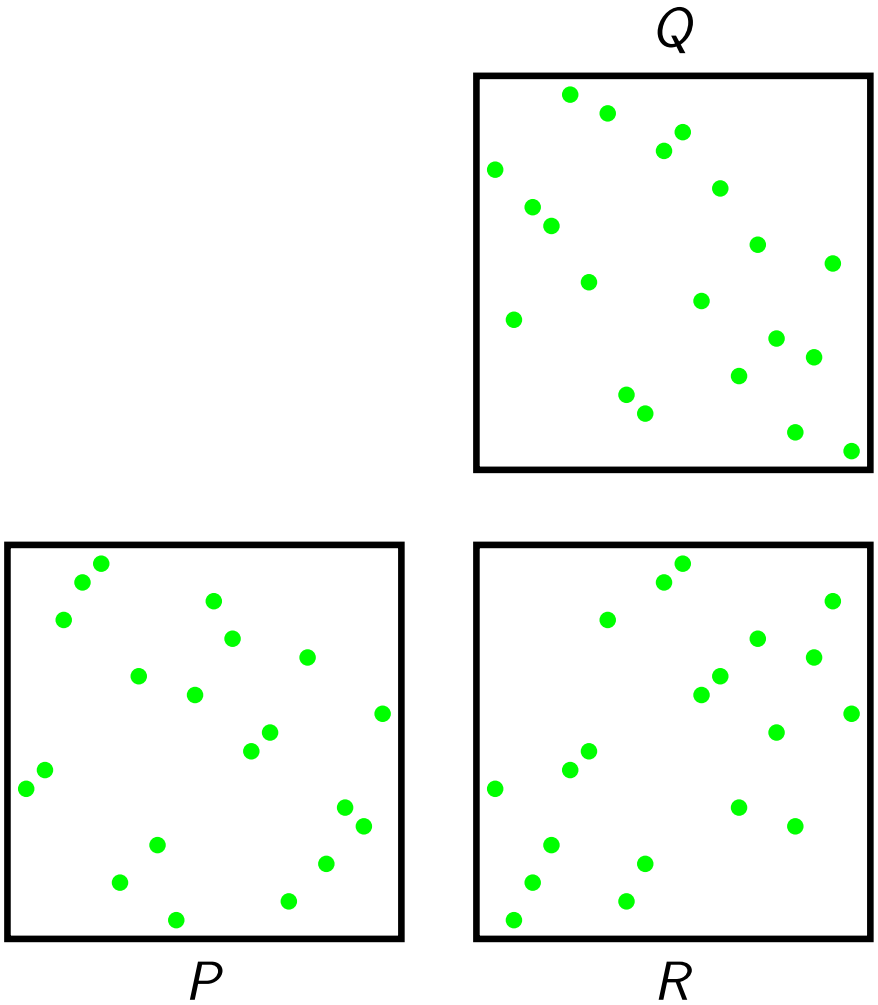
P_{lo}, P_{hi}



R

Unit-Monge matrices and sticky braids

Sticky multiplication



Multiplication in $\mathbb{H}_n$: Steady Ant algorithm

$P \boxdot Q = R \qquad R^\Sigma(i,k) = \min_j\big(P^\Sigma(i,j) + Q^\Sigma(j,k)\big)$

Divide: split range of $j \rightsquigarrow$ two recursive subproblems on $n/2$-matrices

$P_{lo} \boxdot Q_{lo} = R_{lo} \qquad P_{hi} \boxdot Q_{hi} = R_{hi}$

Each subproblem determines good nonzeros, remaining in main problem's solution

Conquer: trace border path through range of i, k (bottom-left to top-right of R), separating good nonzeros of one subproblem from the other

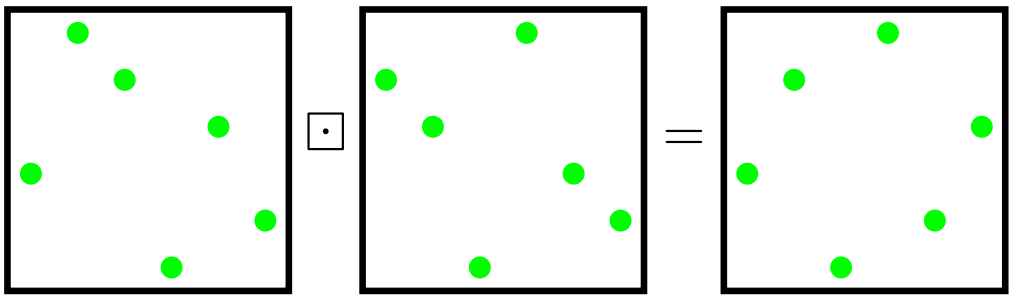
Border path invariant: balance condition on bad nonzeros
$|\{\text{nonzeros of } R_{hi} \text{ above-left}\}| = |\{\text{nonzeros of } R_{lo} \text{ below-right}\}|$

Step through border path: can maintain invariant in time $O(1)$ per step

Keep all good nonzeros; replace bad nonzeros by fresh nonzeros on border path

Conquer time $O(n)$ Overall time $O(n \log n)$

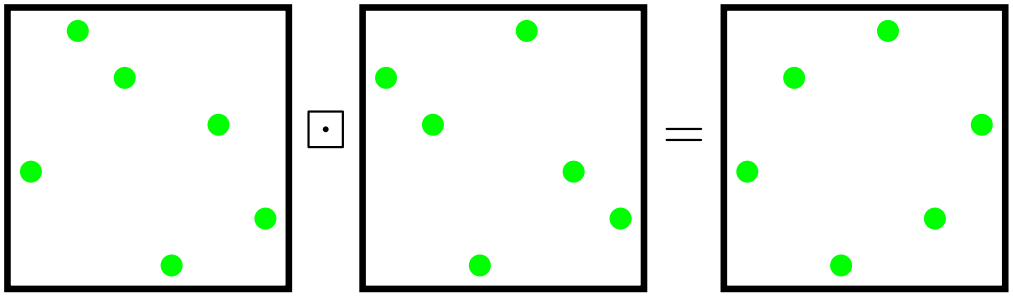
$P \boxdot Q = R$: permutation matrices; sticky braids

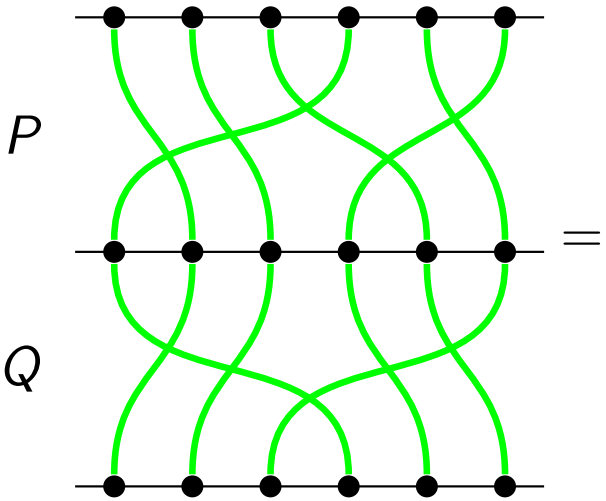


Unit-Monge matrices and sticky braids

Sticky braids

$P \boxdot Q = R$: permutation matrices; sticky braids

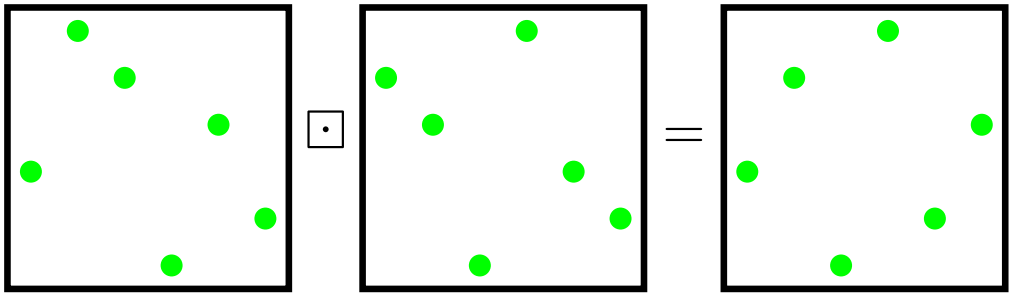


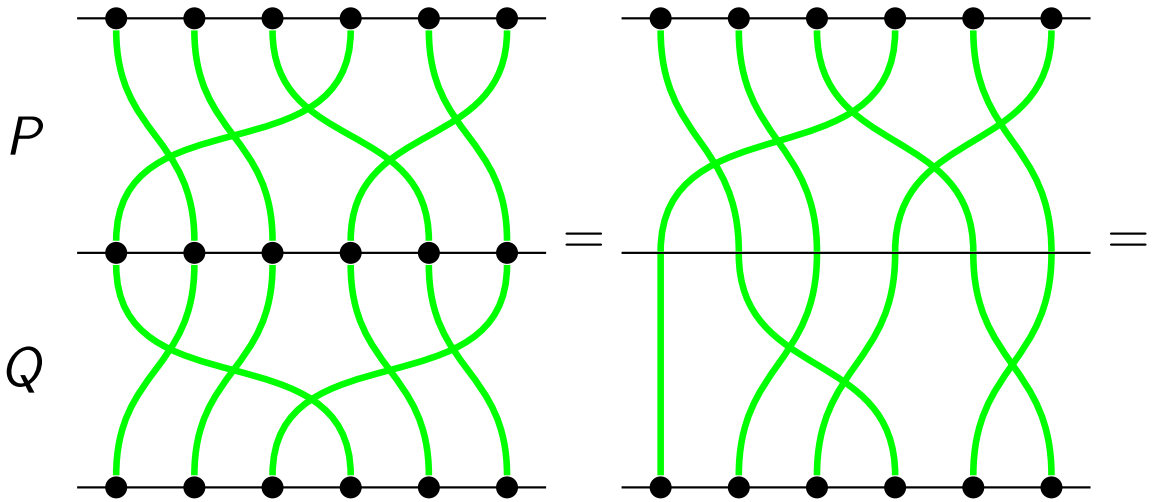


Unit-Monge matrices and sticky braids

Sticky braids

$P \boxdot Q = R$: permutation matrices; sticky braids

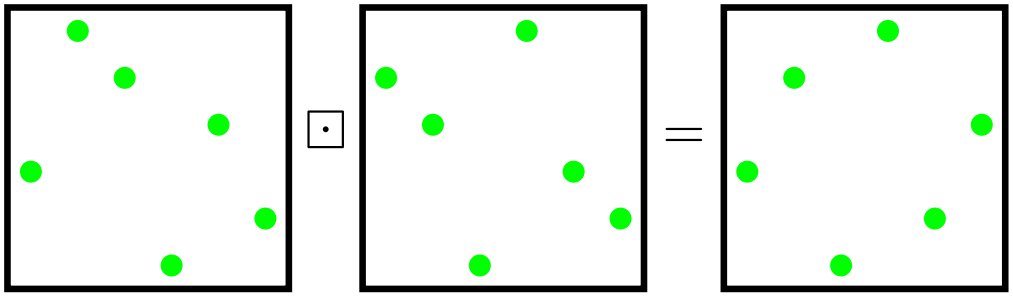


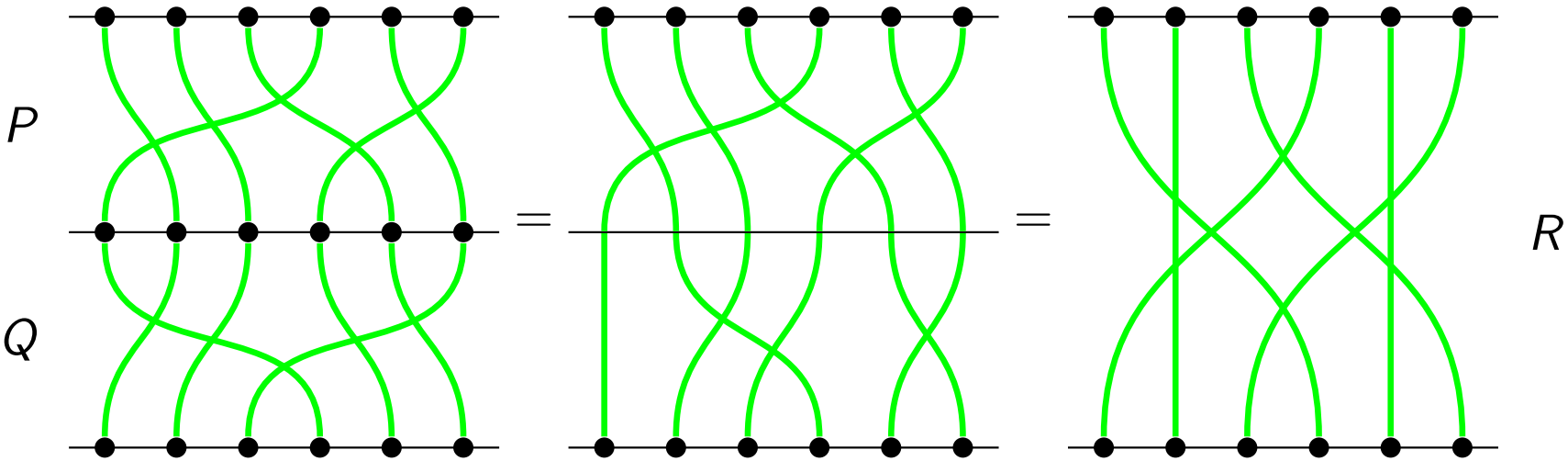


Unit-Monge matrices and sticky braids

Sticky braids

$P \boxdot Q = R$: permutation matrices; sticky braids

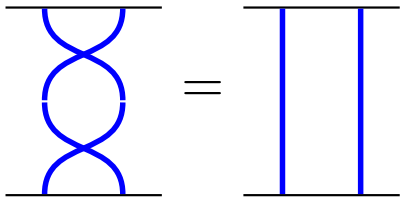




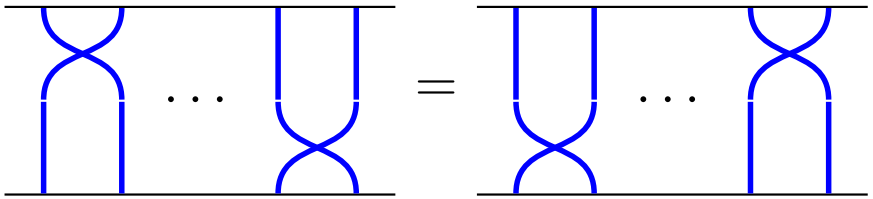
Symmetric group $\mathbb{S}_n$ in Coxeter presentation

$n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary transpositions)

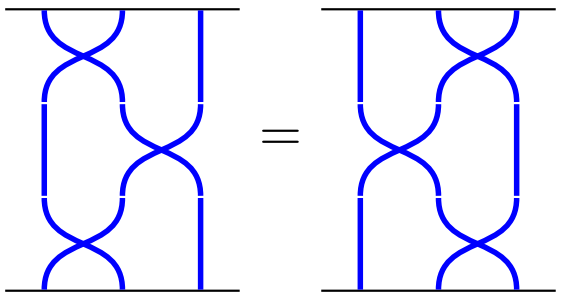
Involution:
$g_i^2 = 1 \quad$ for all i



Far commutativity:
$g_i g_j = g_j g_i \quad j - i > 1$



Braid relations:
$g_i g_j g_i = g_j g_i g_j \quad j - i = 1$

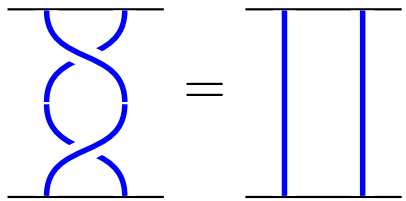


$|\mathbb{S}_n| = n!$

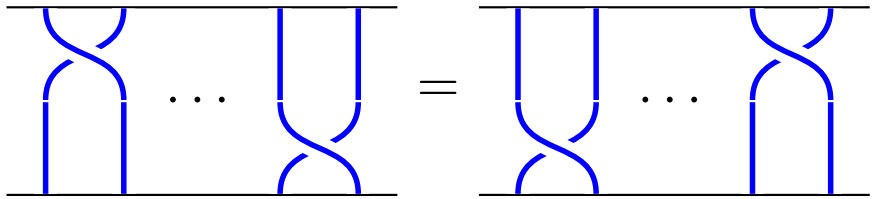
Classical braid group $\mathbb{B}_n$

$n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary transpositions)

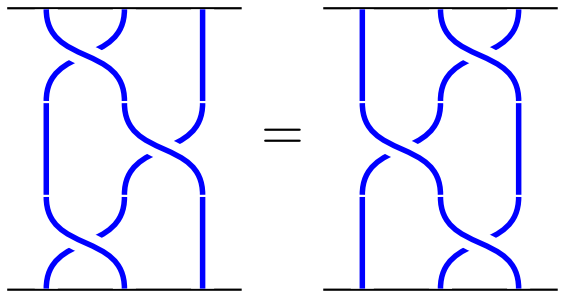
Inversion:
$g_i g_i^{-1} = 1 \quad$ for all i



Far commutativity:
$g_i g_j = g_j g_i \quad j - i > 1$



Braid relations:
$g_i g_j g_i = g_j g_i g_j \quad j - i = 1$

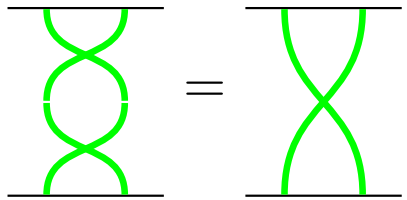


$|\mathbb{B}_n| = \infty$ canonical projection $\mathbb{B}_n \to \mathbb{S}_n$

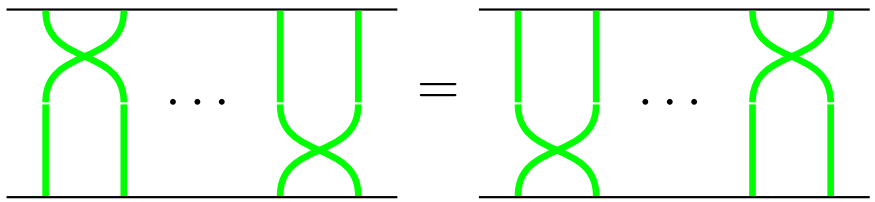
Sticky braid (Hecke) monoid $\mathbb{H}_n$ in Coxeter presentation

$n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary transpositions)

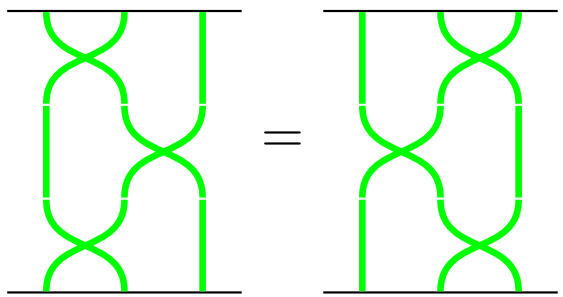
Idempotence:
$g_i^2 = g_i \quad$ for all i



Far commutativity:
$g_i g_j = g_j g_i \quad j - i > 1$



Braid relations:
$g_i g_j g_i = g_j g_i g_j \quad j - i = 1$



$|\mathbb{H}_n| = n! \qquad$ canonical bijection $\mathbb{H}_n \leftrightarrow \mathbb{S}_n$

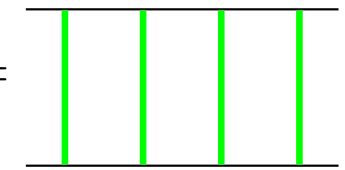
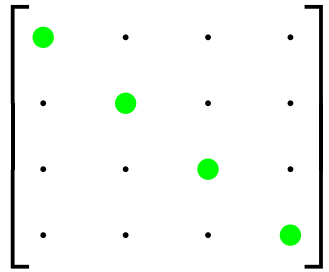
Unit-Monge matrices and sticky braids

Sticky braids

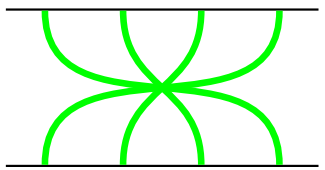
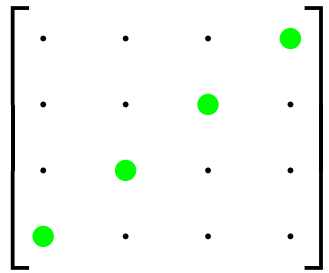
Special elements in $\mathbb{H}_n$

(Denote P^{R} = counterclockwise rotation of P)

Identity I: $I \boxdot x = x$

$I =$  $=$ 

Zero I^{R}: $I^{\mathrm{R}} \boxdot x = I^{\mathrm{R}}$

$I^{\mathrm{R}} =$  $=$ 

Zero divisors: e.g. $P^{\mathrm{R}} \boxdot P = P \boxdot P^{\mathrm{RRR}} = I^{\mathrm{R}}$ for all P

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Fundamentals of string comparison

LCS problem

a, b: strings of length m, n

Longest common subsequence (LCS) score:

- length of longest string that is a subsequence of both a and b
- in computational biology, unweighted alignment
- in ergodic theory, used to define the Feldman–Katok metric
- in software engineering, the diff tool

$lcs(\text{“BAABCBCA”}, \text{“CABCABA”}) = length(\text{“ABCBA”}) = 5$

LCS problem

LCS score for a vs b

LCS problem

LCS score for a vs b

LCS: running time

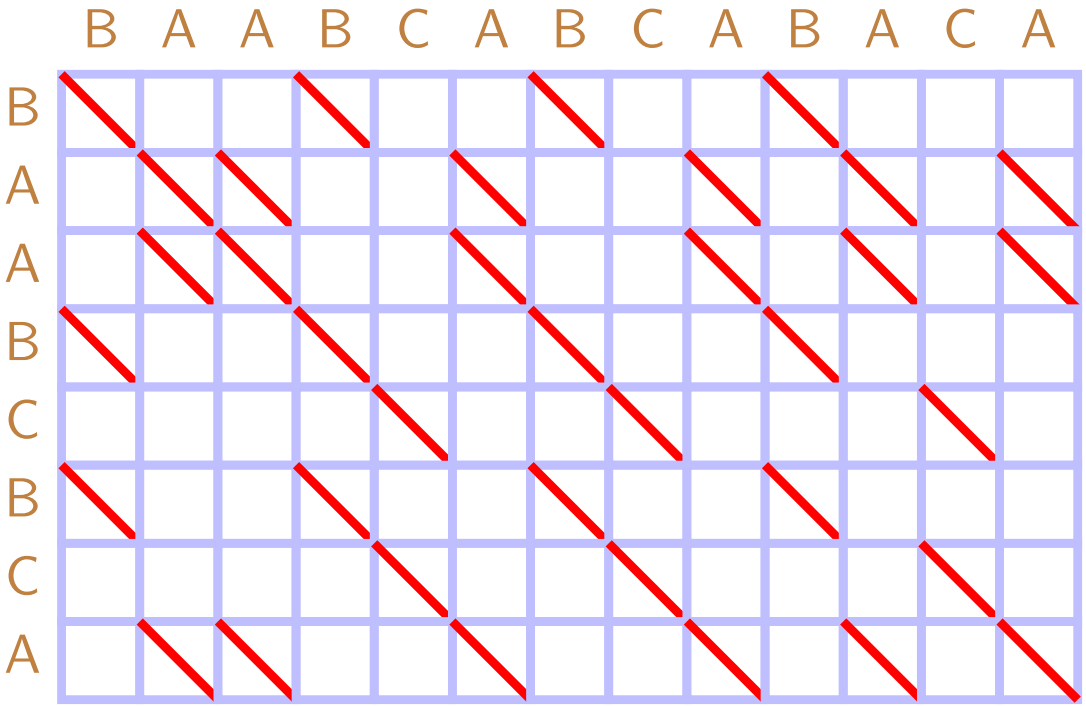
$O(mn)$		[Wagner, Fischer: 1974]
$O\left(\frac{mn}{(\log n)^c}\right)$		[Masek, Paterson: 1980] [Crochemore+: 2003] [Paterson, Dančík: 1994] [Bille, Farach-Colton: 2008]
No $O\left((mn)^{1-\epsilon}\right)$	$\epsilon > 0$; assuming SETH	[Abboud+: 2015] [Backurs, Indyk: 2015]

Polylog's exponent c depends on alphabet size and computation model

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

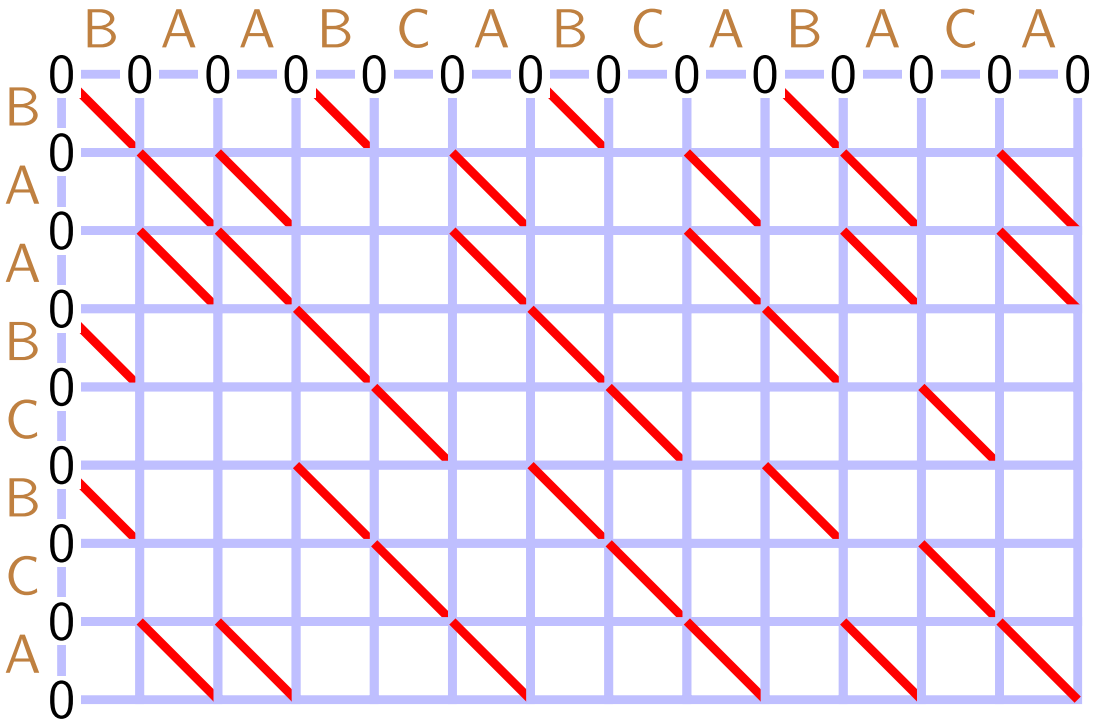
$$lcs(a, \emptyset) = 0$$
$$lcs(\emptyset, b) = 0$$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

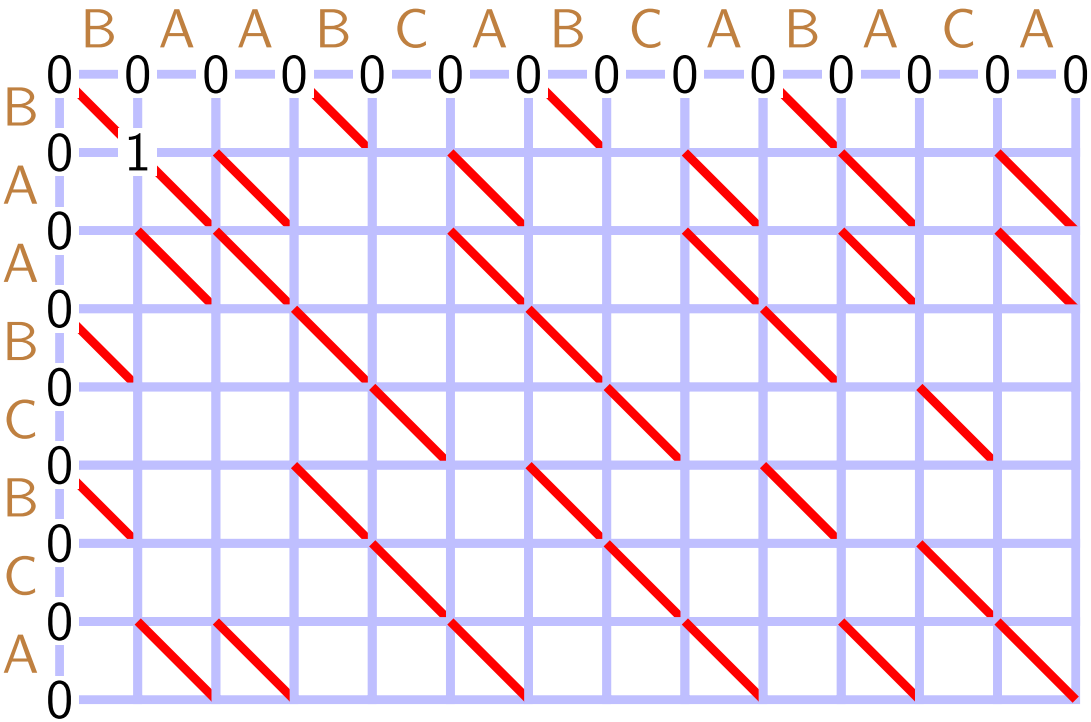
$$lcs(a, \emptyset) = 0$$
$$lcs(\emptyset, b) = 0$$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

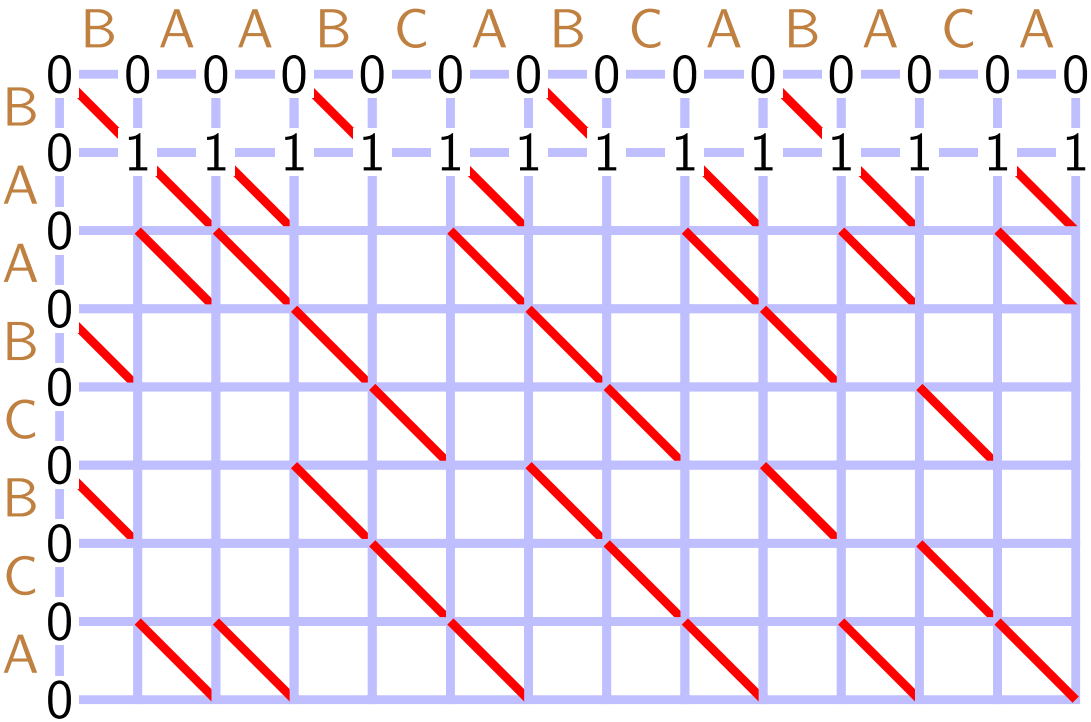
$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

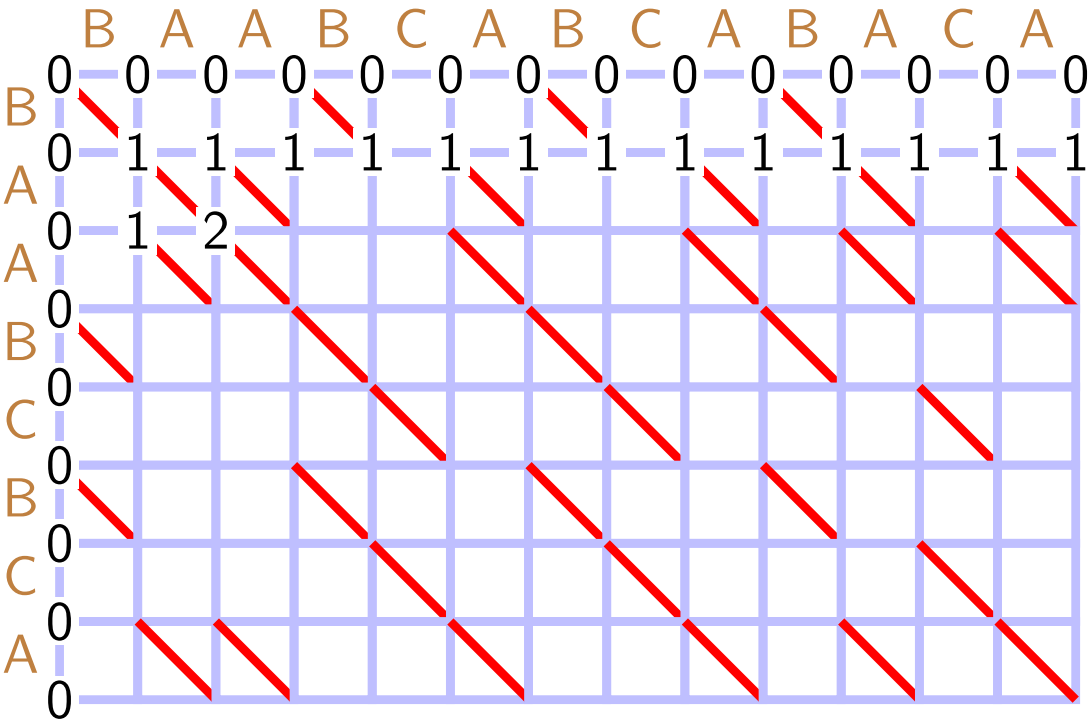
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

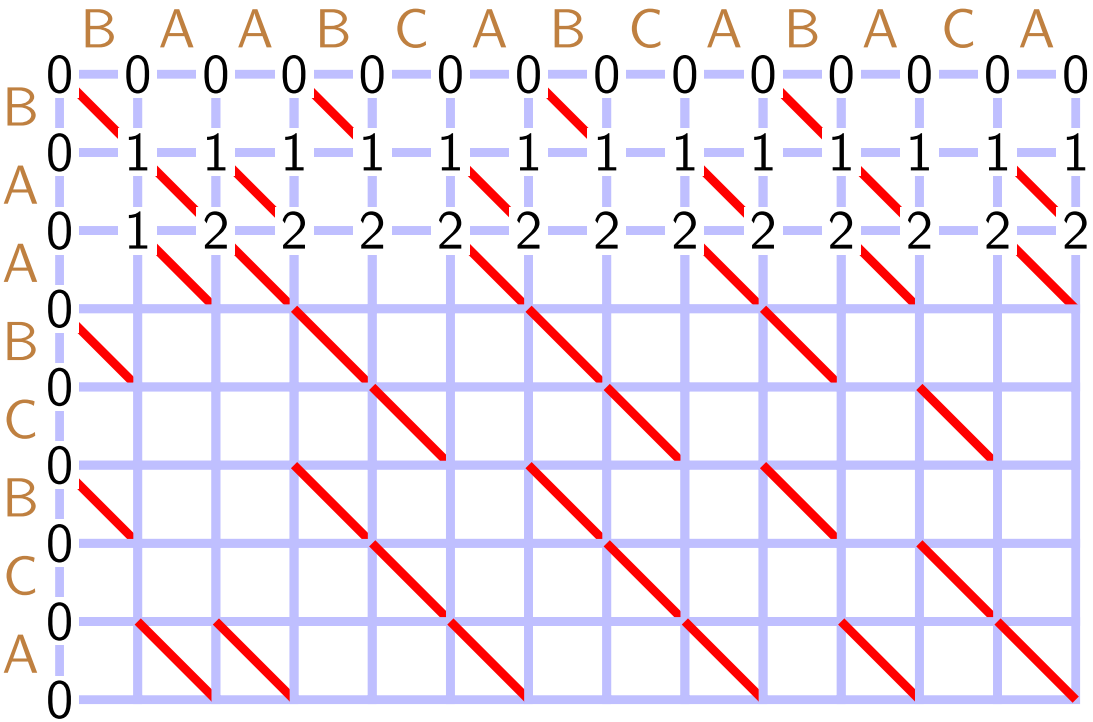
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

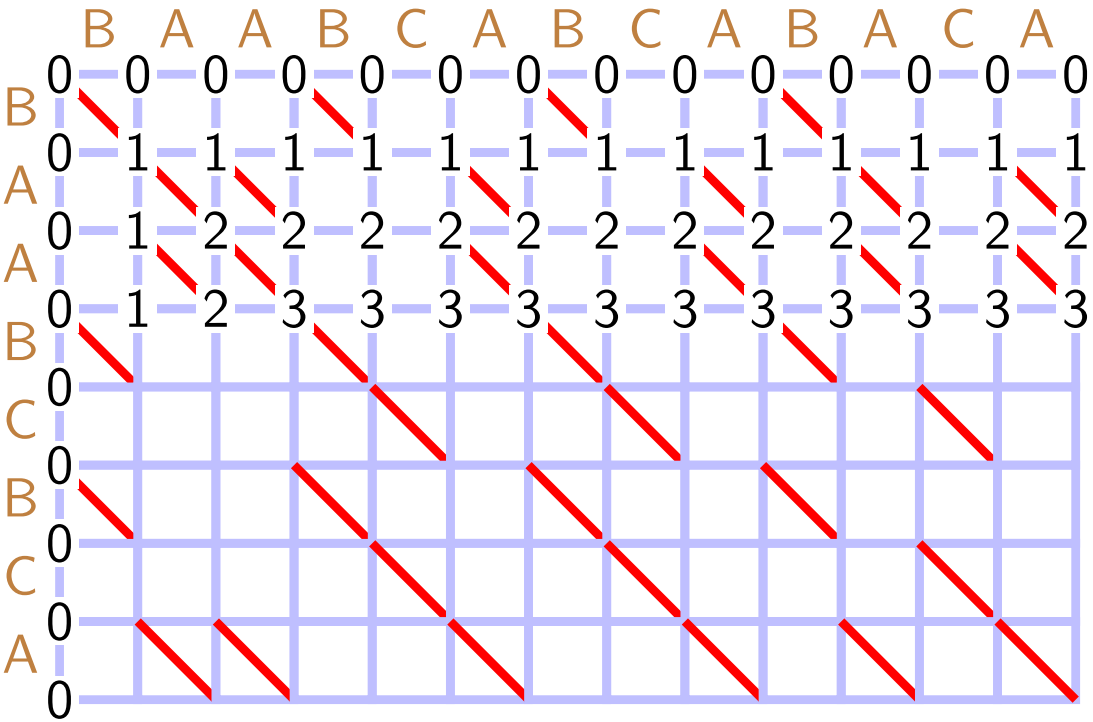
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

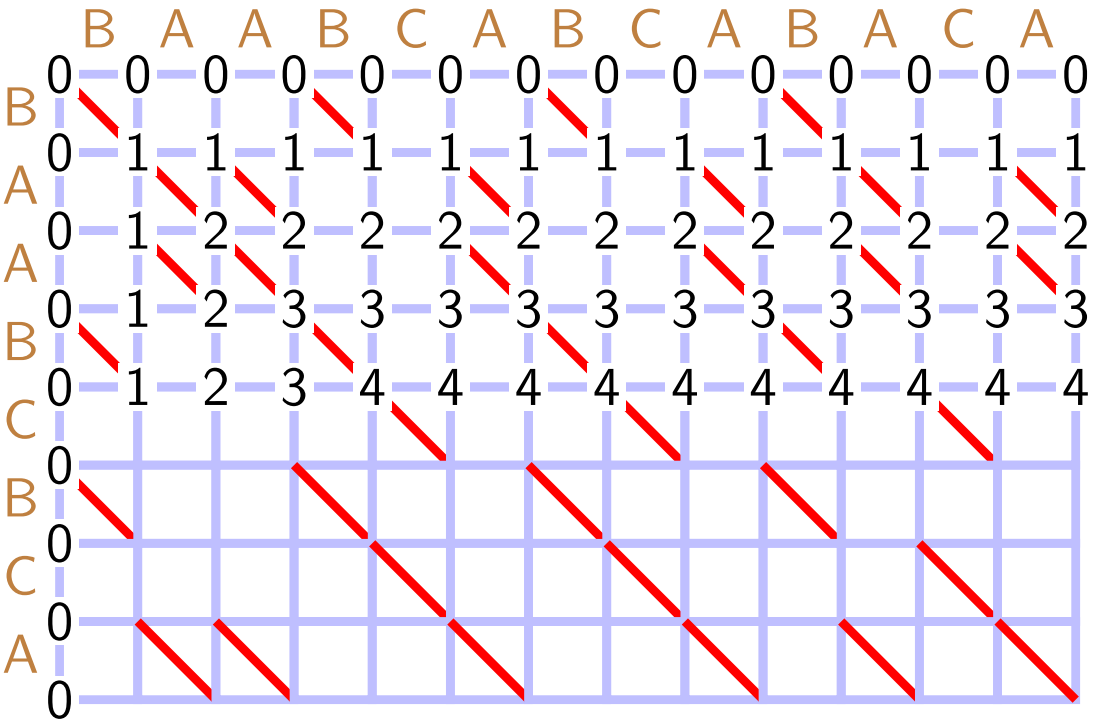
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

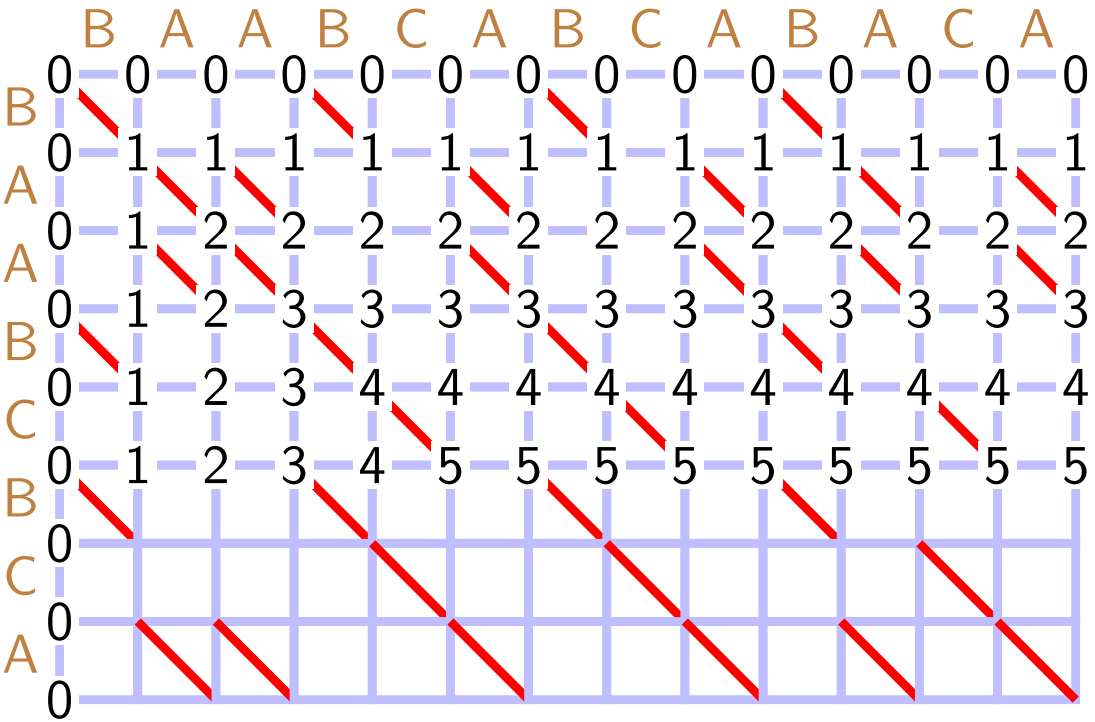
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

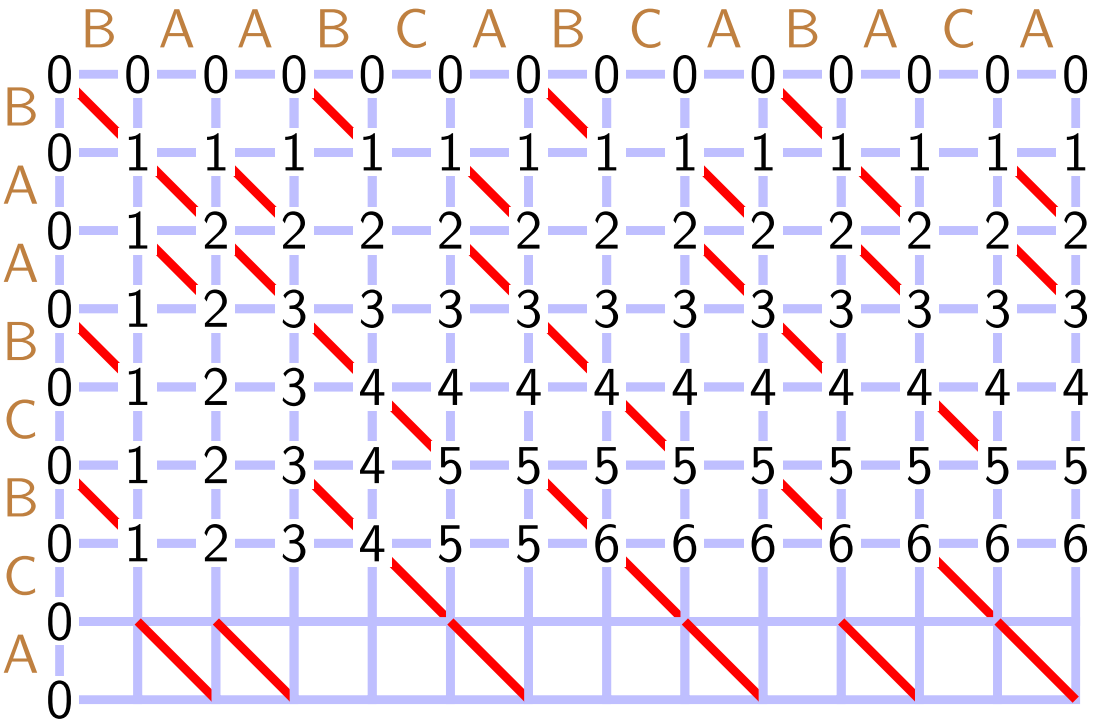
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

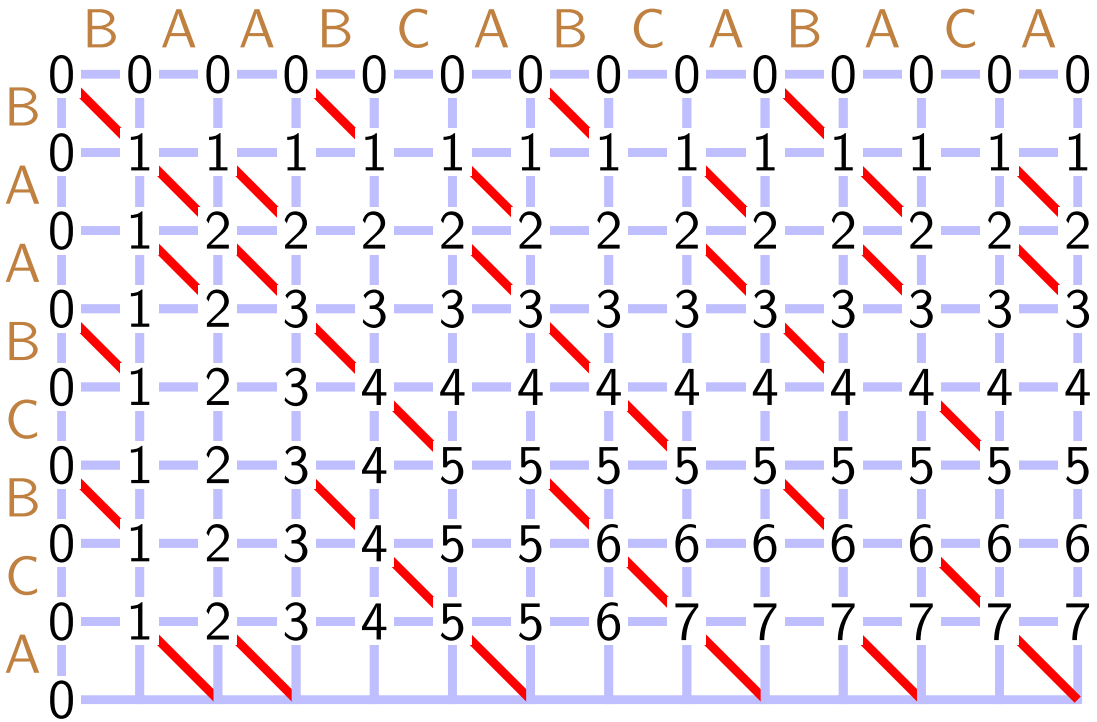
$$lcs(a, \emptyset) = 0$$
$$lcs(\emptyset, b) = 0$$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

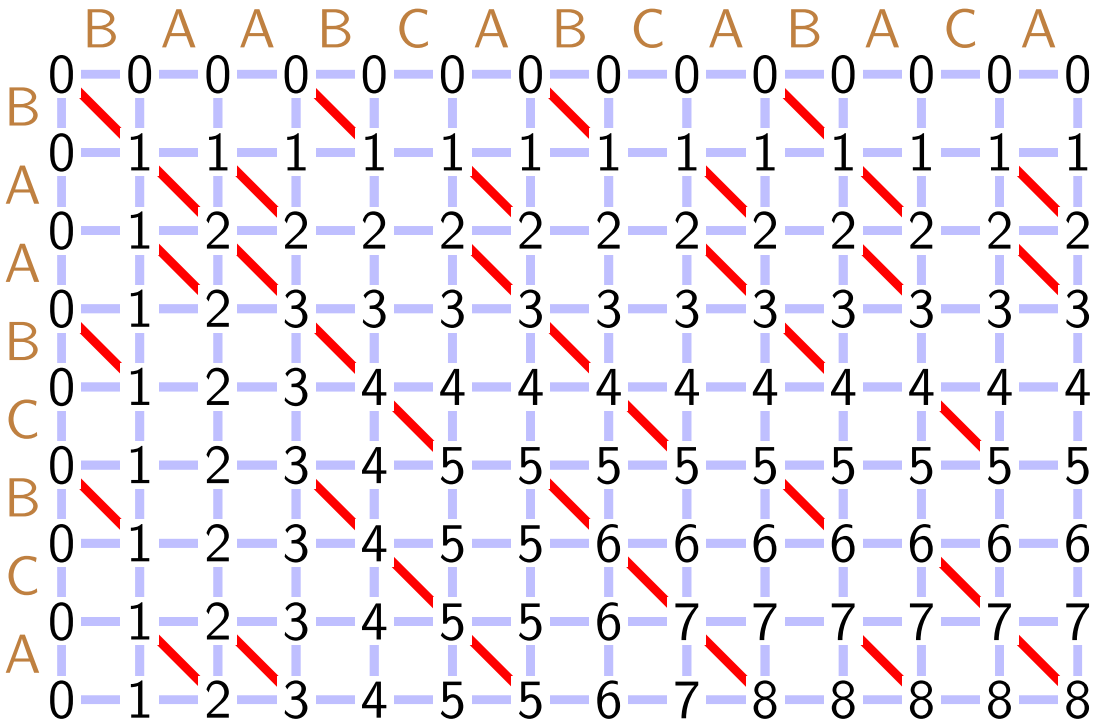
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS computation by classical dynamic programming (DP)



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

blue $= 0$ (skip)
red $= 1$ (match)

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

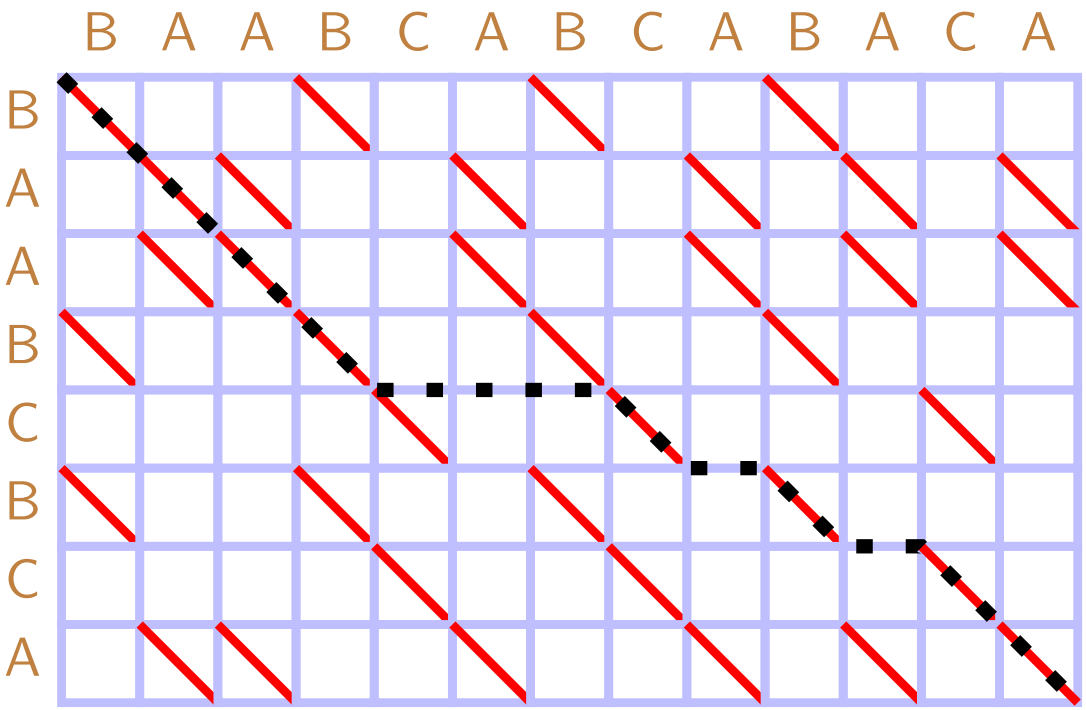
$lcs(\emptyset, b) = 0$

$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

Fundamentals of string comparison

LCS problem

LCS as a maximum path in the LCS grid



$a =$ "BAABCBCA"
$b =$ "BAABCABCABACA"

$blue = 0$ (skip)
$red = 1$ (match)

$lcs(a, b) = 8$

LCS = highest-score path top-left $\rightsquigarrow$ bottom-right

Fundamentals of string comparison

LCS problem

LCS: classical dynamic programming (DP)

Iterate over cells in any $\ll$-compatible order

Active cell update: time $O(1)$

Overall time $O(mn)$

Fundamentals of string comparison

LCS problem

'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'

L. Carroll, *Alice in Wonderland*

Fundamentals of string comparison

LCS problem

'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'

L. Carroll, *Alice in Wonderland*

Dynamic programming: begins at empty strings, proceeds by appending characters, then stops

What about strings that are

- dynamic (prepending/deleting characters)
- compressed (concatenation, taking substrings)
- required to be aligned locally/in parallel

Fundamentals of string comparison

LCS problem

Running DP from both ends: $\times 2$ parallelism, but still not good enough

Is dynamic programming strictly necessary to solve sequence alignment problems?

> Eppstein+, *Efficient algorithms for sequence analysis*, 1991

Semi-local LCS (SLCS) problem

LCS scores for a vs b:

- string-substring (whole a vs every substring of b)
- prefix-suffix (every prefix of a vs every suffix of b)
- suffix-prefix (every suffix of a vs every prefix of b)
- substring-string (every substring of a vs whole b)

Output scores can be represented implicitly

Semi-local LCS (SLCS) problem

LCS scores for a vs b:

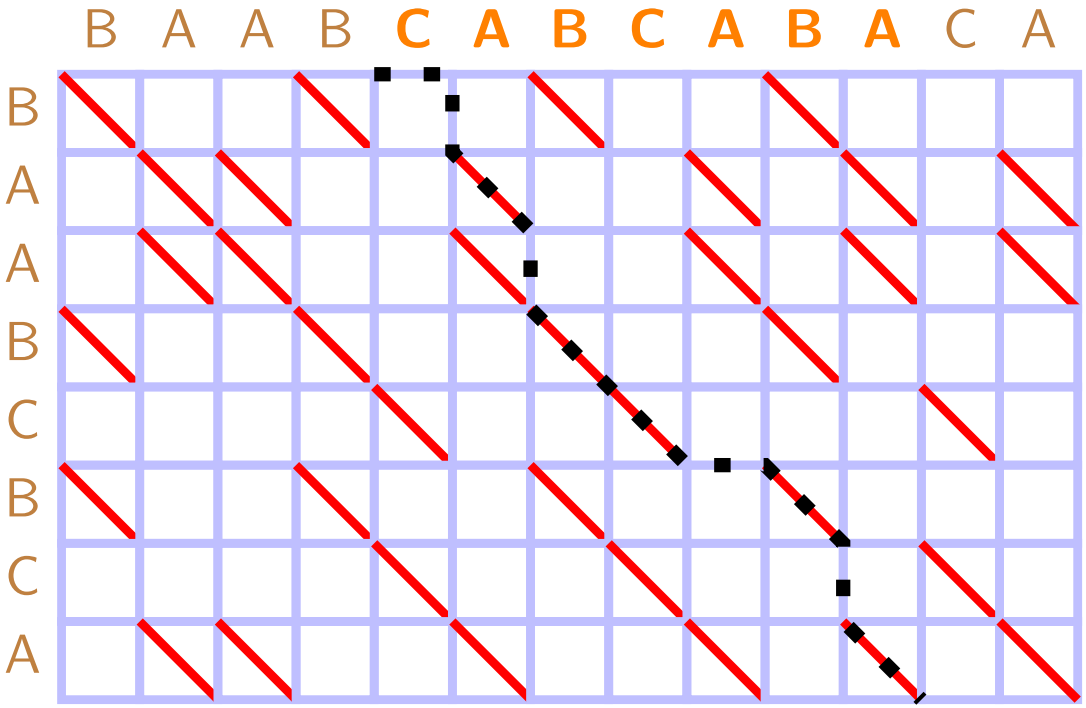
- string-substring (whole a vs every substring of b)
- prefix-suffix (every prefix of a vs every suffix of b)
- suffix-prefix (every suffix of a vs every prefix of b)
- substring-string (every substring of a vs whole b)

Output scores can be represented implicitly

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS as maximum paths in the LCS grid



$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

$blue = 0$ (skip)

$red = 1$ (match)

$lcs(a, b\langle 4 : 11\rangle) = 5$

String-substring LCS: all highest-score top-to-bottom paths

SLCS: all highest-score boundary-to-boundary paths

SLCS: output representation and running time

size	query time		
$O(n^2)$	$O(1)$	string-substring	trivial
$O(m^{1/2}n)$	$O(\log n)$	string-substring	[Alves+: 2003]
$O(n)$	$O(n)$	string-substring	[Alves+: 2005]
$O(n \log n)$	$O(\log^2 n)$		[T: 2006]
...or any 2D orthogonal range counting data structure			

running time		
$O(mn^2) = O(n \cdot mn)$	string-substring	repeated DP
$O(mn)$	string-substring	[Schmidt: 1998; Alves+: 2005]
$O(mn)$		[T: 2006]
$O\left(\frac{mn}{(\log n)^{O(1)}}\right)$		[T: 2006–07]

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS matrix H and SLCS kernel P

$H[i,j]$: max number of matched characters for a vs substring $b\langle i:j\rangle$

$j-i-H[i,j]$: min number of unmatched characters

Properties of matrix $j-i-H[i,j]$:

- simple unit-Monge
- therefore, $= P^{\Sigma}$, where $P = -H^{\square}$ is a permutation matrix

P is the SLCS kernel, giving an implicit representation of H

Range tree for P: memory $O(n \log n)$, query time $O(\log^2 n)$

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS matrix H and SLCS kernel P (only string-substring component shown)

0	1	2	3	4	5	6	6	7	8	8	8	8	8
-1	0	1	2	3	4	5	5	6	7	7	7	7	7
-2	-1	0	1	2	3	4	4	5	6	6	6	6	7
-3	-2	-1	0	1	2	3	3	4	5	5	6	6	7
-4	-3	-2	-1	0	1	2	2	3	4	4	(5)	5	6
-5	-4	-3	-2	-1	0	1	2	3	4	4	5	5	6
-6	-5	-4	-3	-2	-1	0	1	2	3	3	4	4	5
-7	-6	-5	-4	-3	-2	-1	0	1	2	2	3	3	4
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	3	4
-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

$$H[i,j] = \begin{cases} lcs(a, b\langle i : j\rangle) & i \le j \\ j - i & i \ge j \end{cases}$$

$H[0,13] = lcs(a,b) = 8$

$H[4,11] = lcs(a, b\langle 4 : 11\rangle) = 5$

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS matrix H and SLCS kernel P (only string-substring component shown)

0	1	2	3	4	5	6	6	7	8	8	8	8	8
-1	0	1	2	3	4	5	5	6	7	7	7	7	7
-2	-1	0	1	2	3	4	4	5	6	6	6	6	7
-3	-2	-1	0	1	2	3	3	4	5	5	6	6	7
-4	-3	-2	-1	0	1	2	2	3	4	4	(5)	5	6
-5	-4	-3	-2	-1	0	1	2	3	4	4	5	5	6
-6	-5	-4	-3	-2	-1	0	1	2	3	3	4	4	5
-7	-6	-5	-4	-3	-2	-1	0	1	2	2	3	3	4
-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	3	4
-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4
-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3
-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1
-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

$$H[i,j] = \begin{cases} lcs(a, b\langle i : j\rangle) & i \le j \\ j - i & i \ge j \end{cases}$$

blue | *red*: diff in $H = 0 \mid 1$

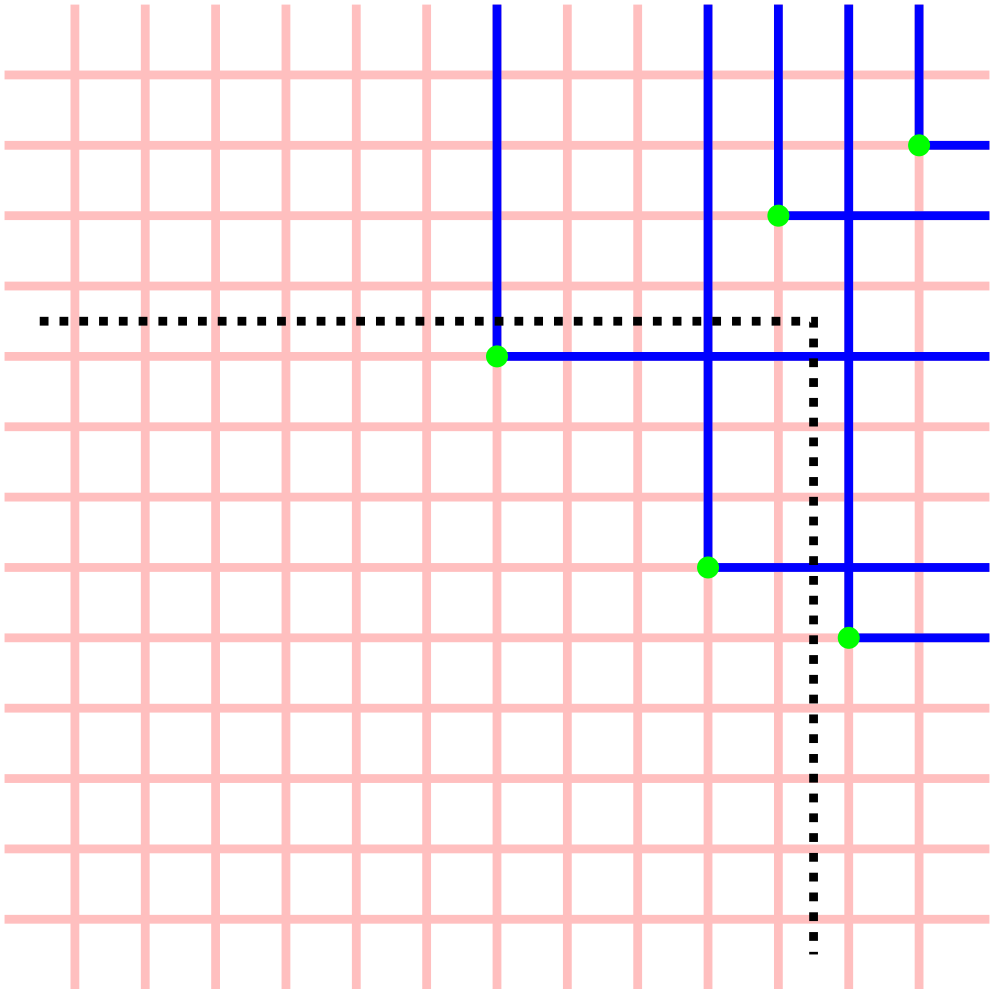
green: $P\langle i,j\rangle = 1$

$$H[i,j] = j - i - P^{\Sigma}[i,j]$$

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS matrix H and SLCS kernel P (only string-substring component shown)



$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

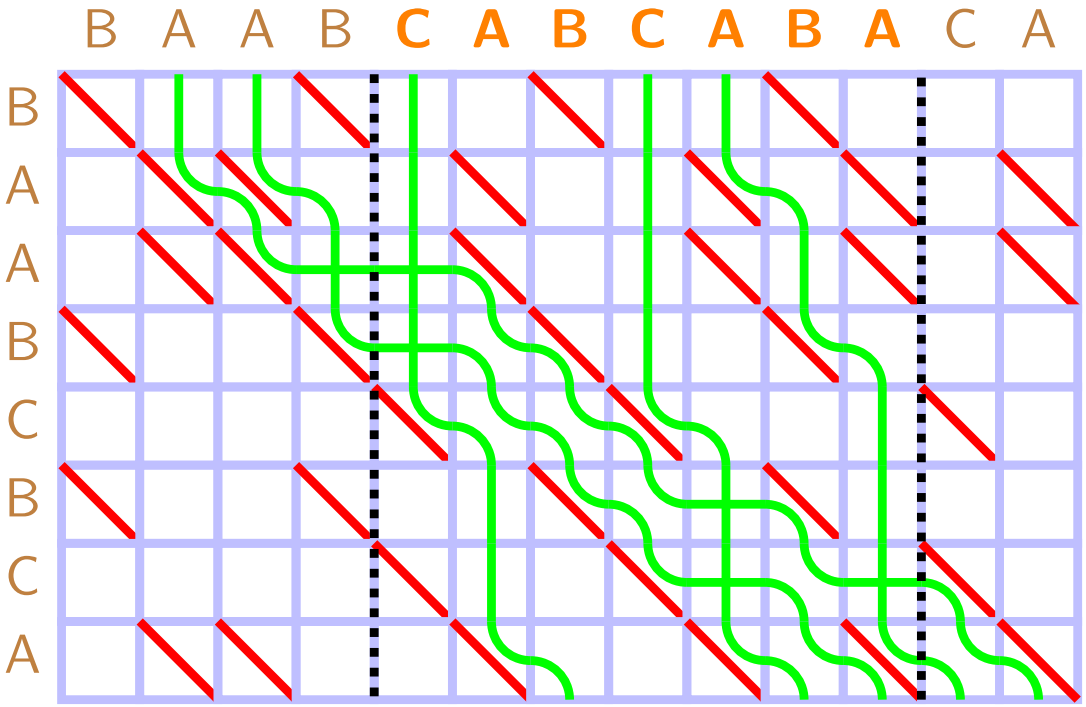
$$H[i,j] = \begin{cases} lcs(a, b\langle i : j\rangle) & i \le j \\ j - i & i \ge j \end{cases}$$

$H[4,11] = 11 - 4 - P^{\Sigma}[i,j] =$
$11 - 4 - 2 = 5$

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS kernel as (reduced) embedded sticky braid in LCS grid



$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

$H[4, 11] = 11 - 4 - P^{\Sigma}[i, j] =$
$11 - 4 - 2 = 5$

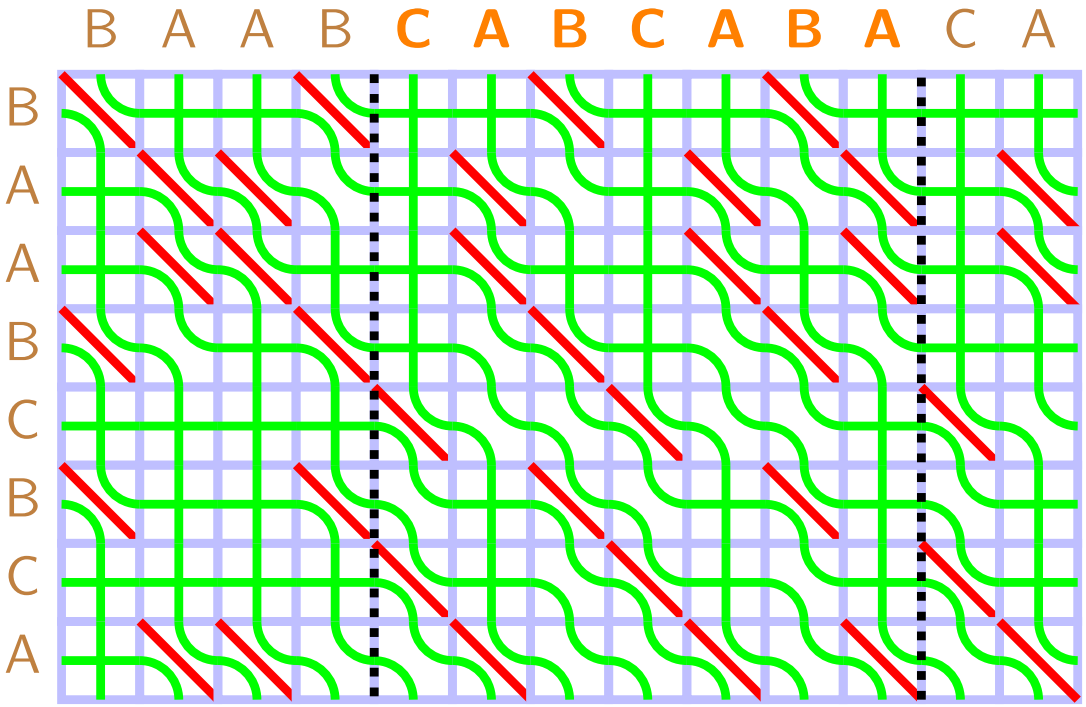
String-substring LCS: $P\langle i, j\rangle = 1$ iff strand i *(top)* $\rightsquigarrow j$ *(bottom)*

Each strand is a unit obstruction to LCS, if crossed left-to-right

Fundamentals of string comparison

Semi-local LCS (SLCS) problem

SLCS kernel as (reduced) embedded sticky braid in LCS grid



$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

$H[4, 11] = 11 - 4 - P^{\Sigma}[i, j] =$
$11 - 4 - 2 = 5$

SLCS: $P\langle i, j\rangle = 1$ iff strand i *(top/left)* $\rightsquigarrow$ j *(bottom/right)*

SLCS kernel: sticky braid with no particular embedding

Fundamentals of string comparison

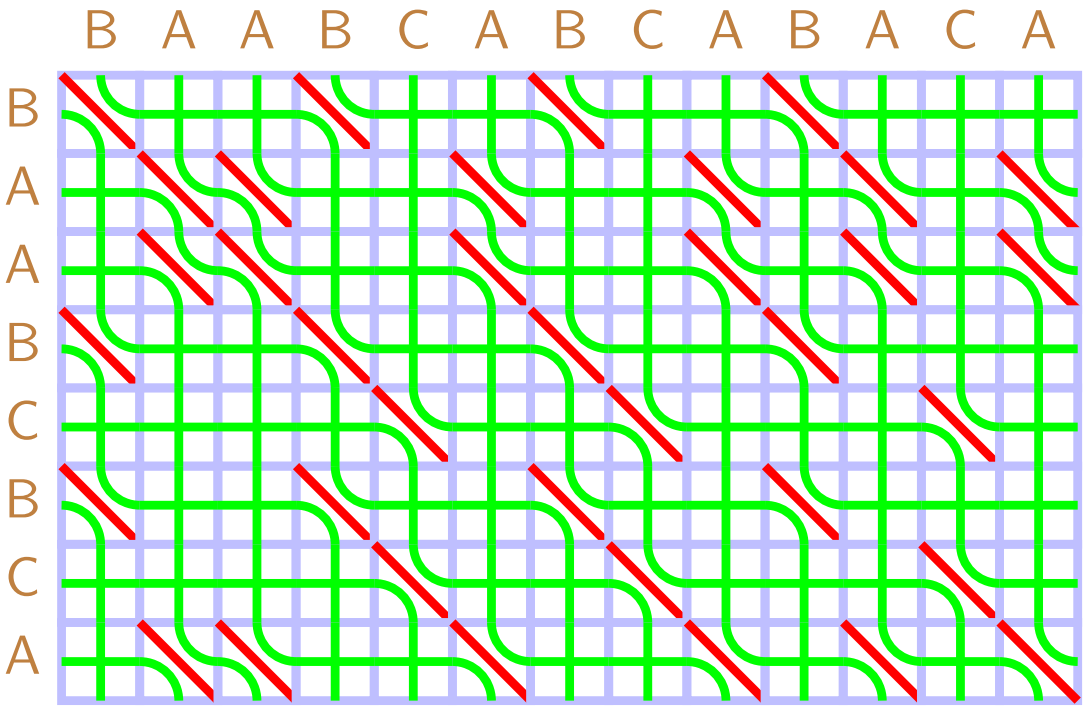
Semi-local LCS (SLCS) problem

Sticky braid: a highly symmetric object (Hecke word)

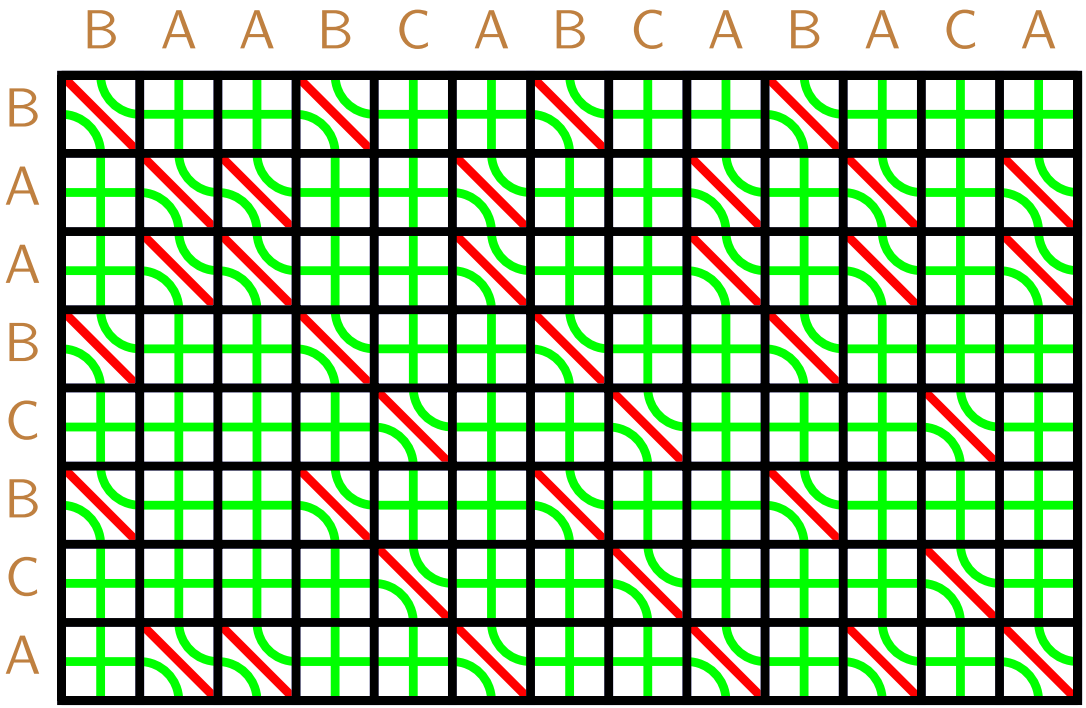
Can be built by assembling subbraids: divide-and-conquer

Flexible approach to local alignment, compressed approximate matching, parallel computation...

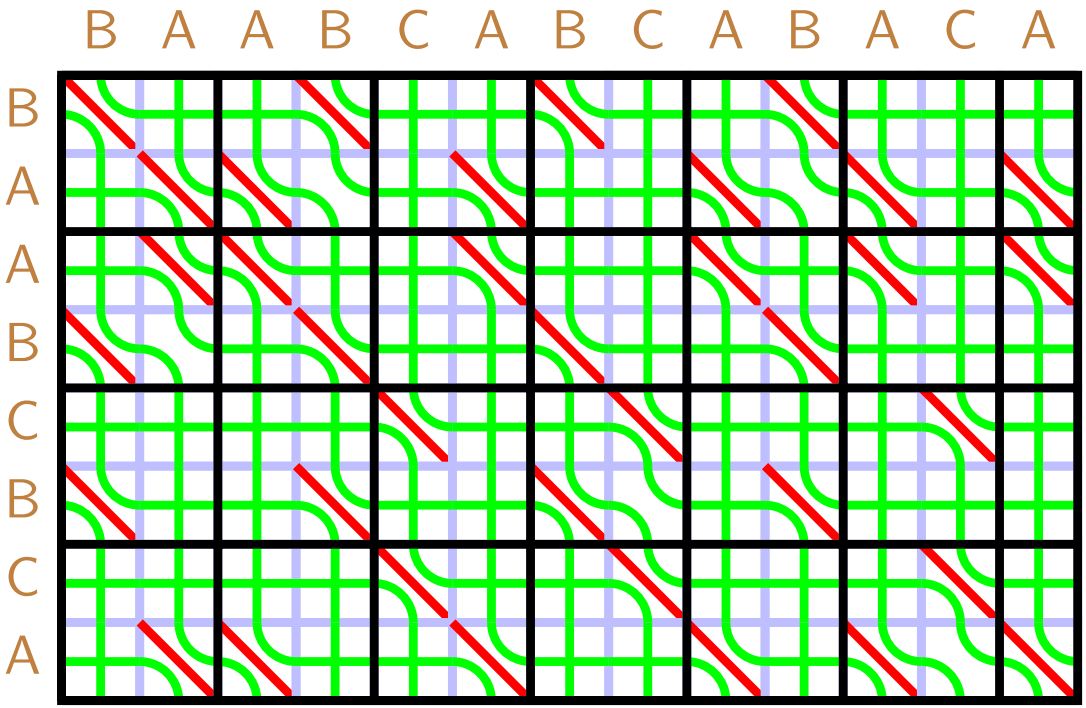
SLCS by recursive combing



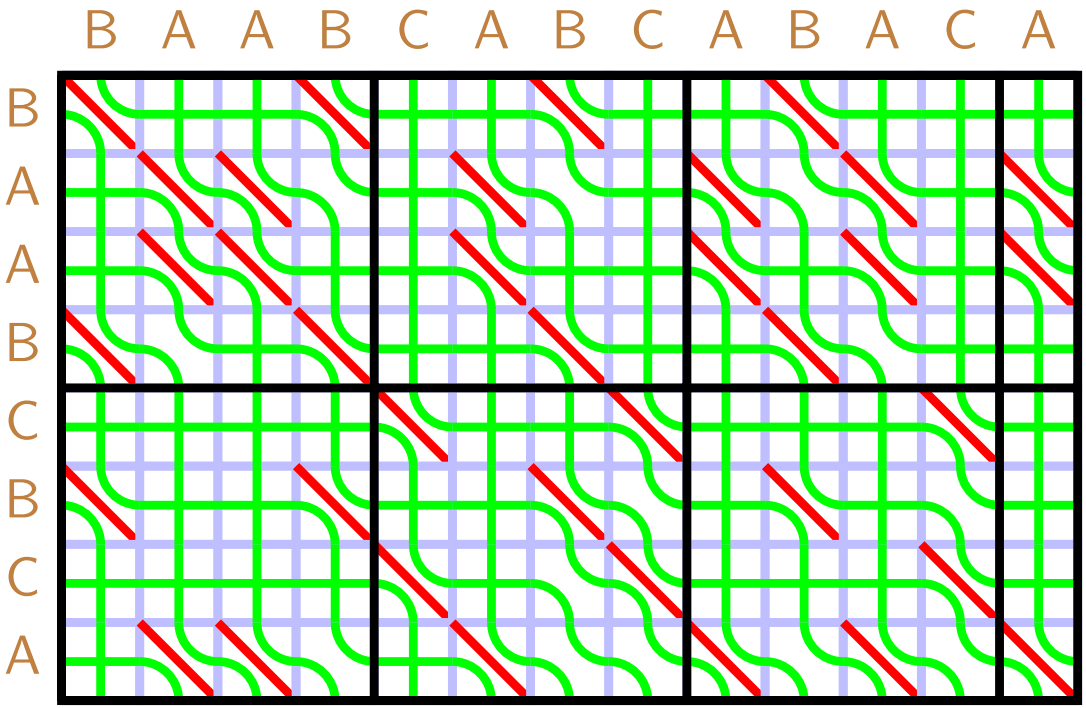
SLCS by recursive combing



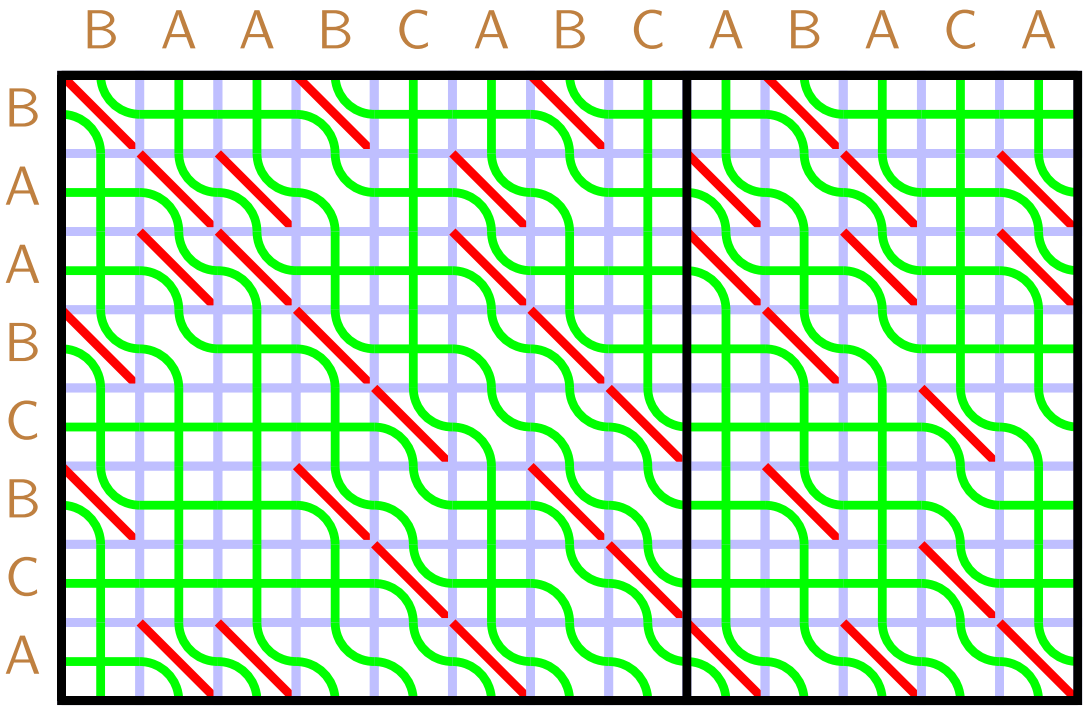
SLCS by recursive combing



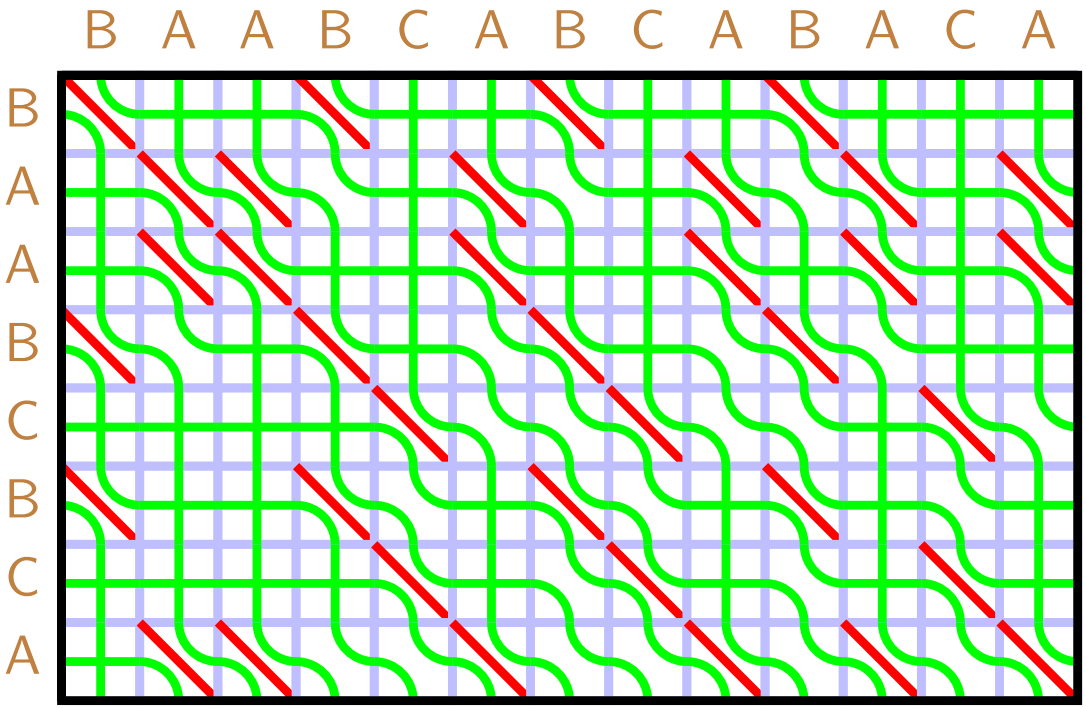
SLCS by recursive combing



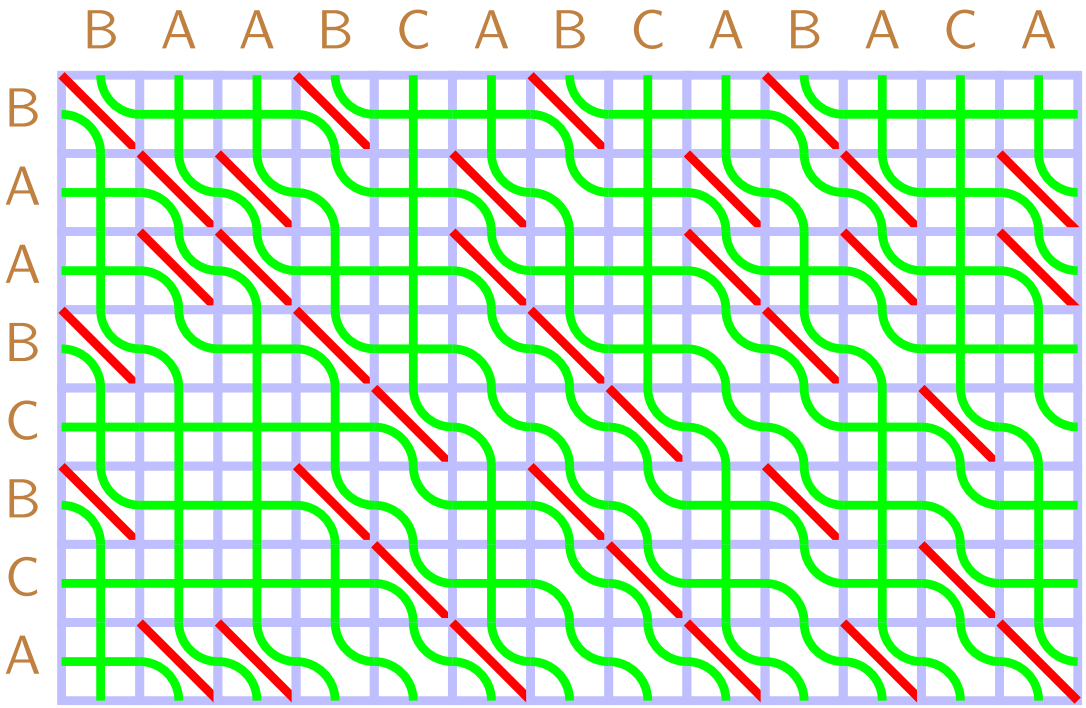
SLCS by recursive combing



SLCS by recursive combing



SLCS by recursive combing



SLCS: recursive combing

Initialise as saturated braid: mismatch cell = crossing

Recursion on LCS grid

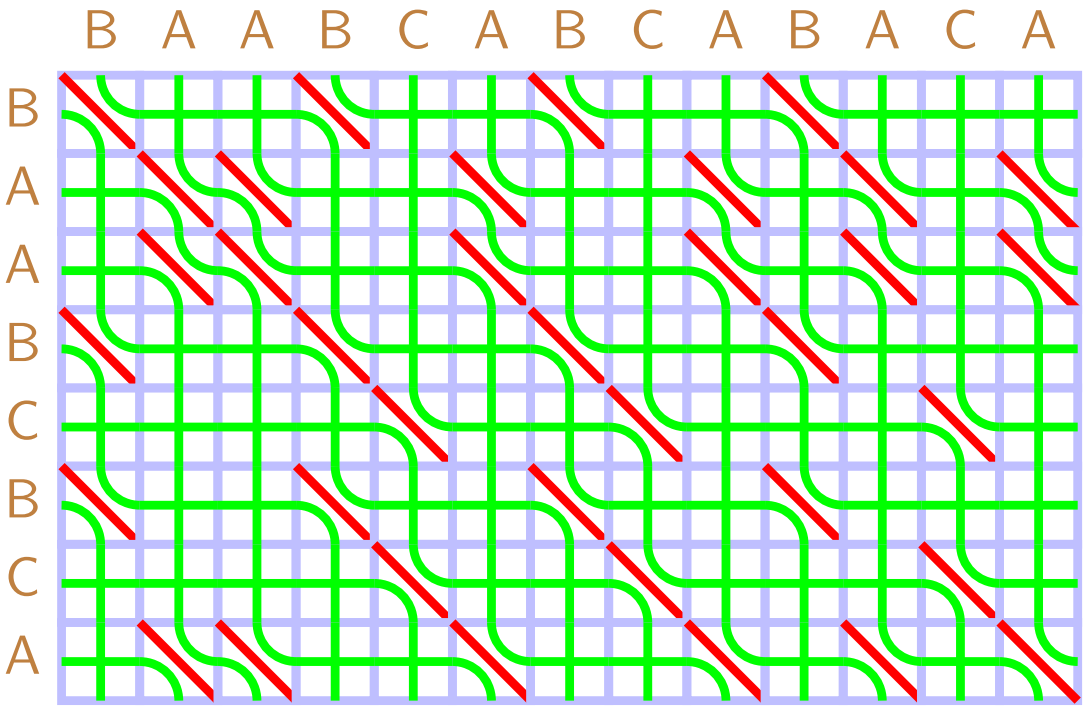
- divide: partition either a or b
- obtain subproblem SLCS kernels recursively
- conquer: SLCS kernel composition by sticky multiplication

Recursion base: $m = n = 1$

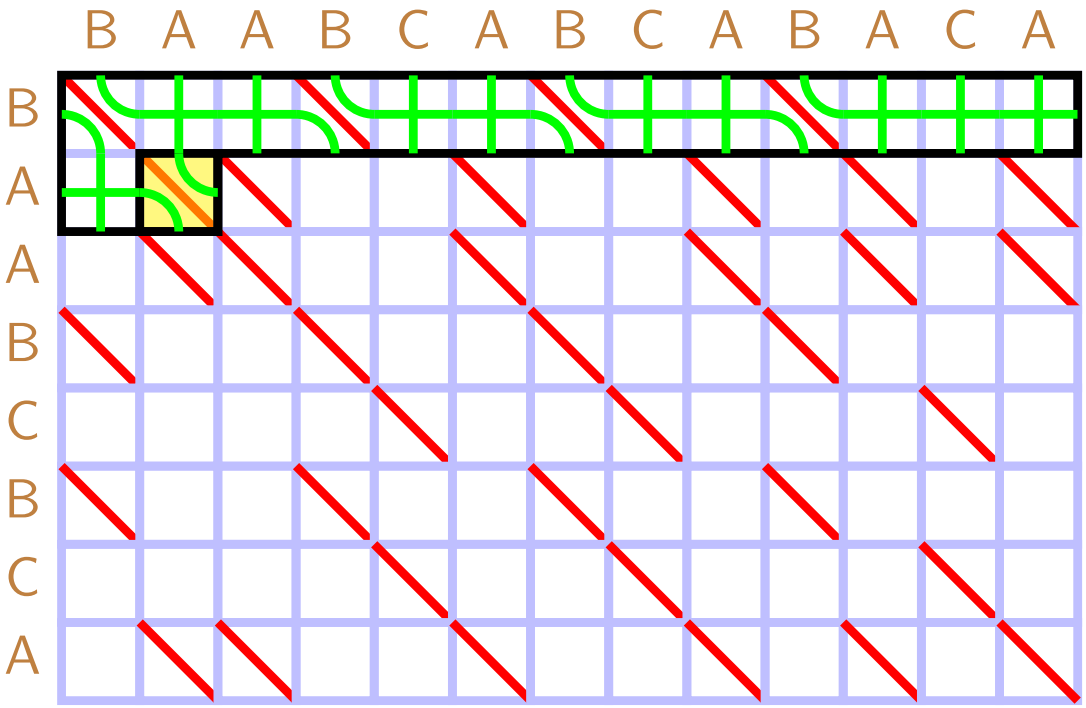
Overall time $O(mn)$

Correctness: by sticky braid relations

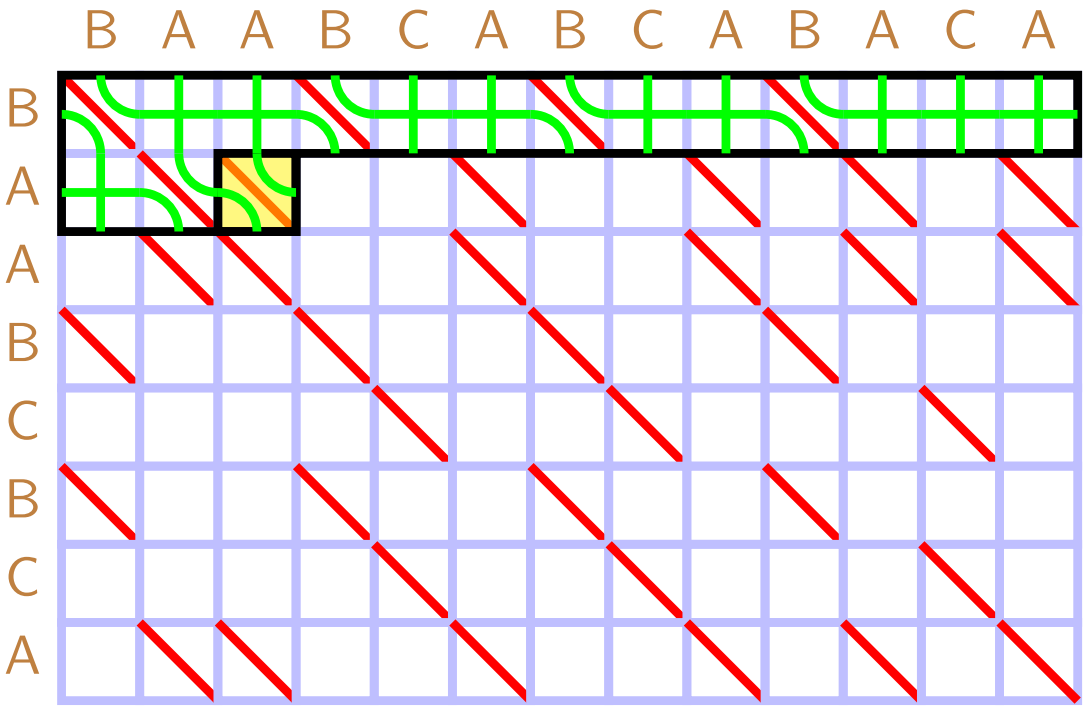
SLCS by iterative combing



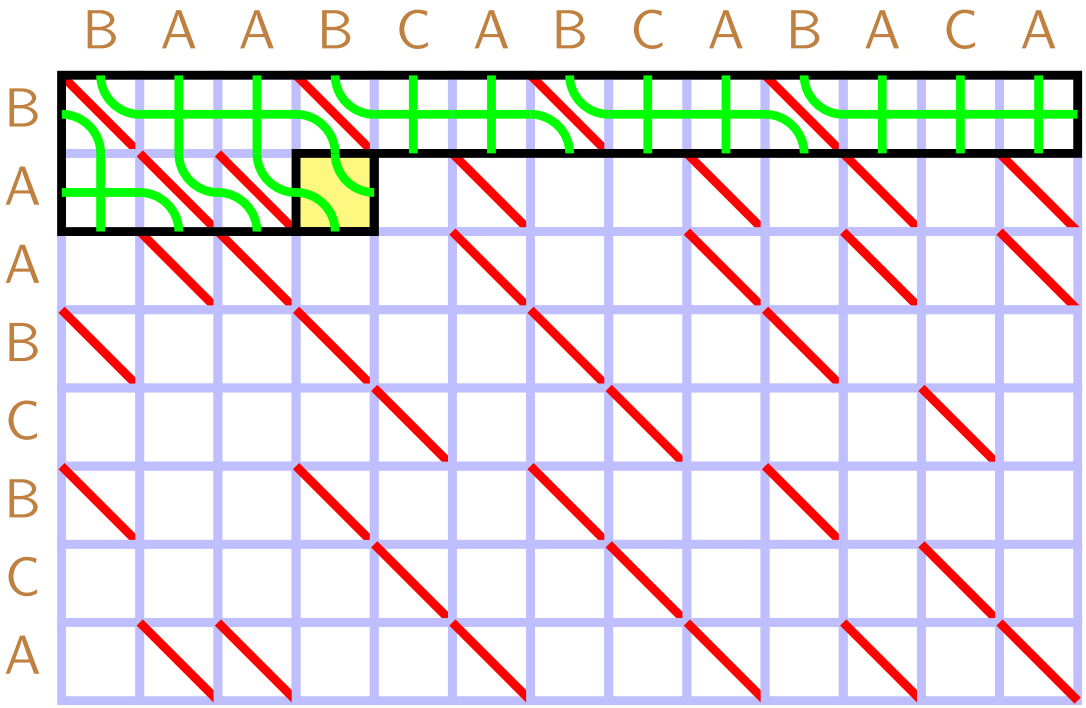
SLCS by iterative combing



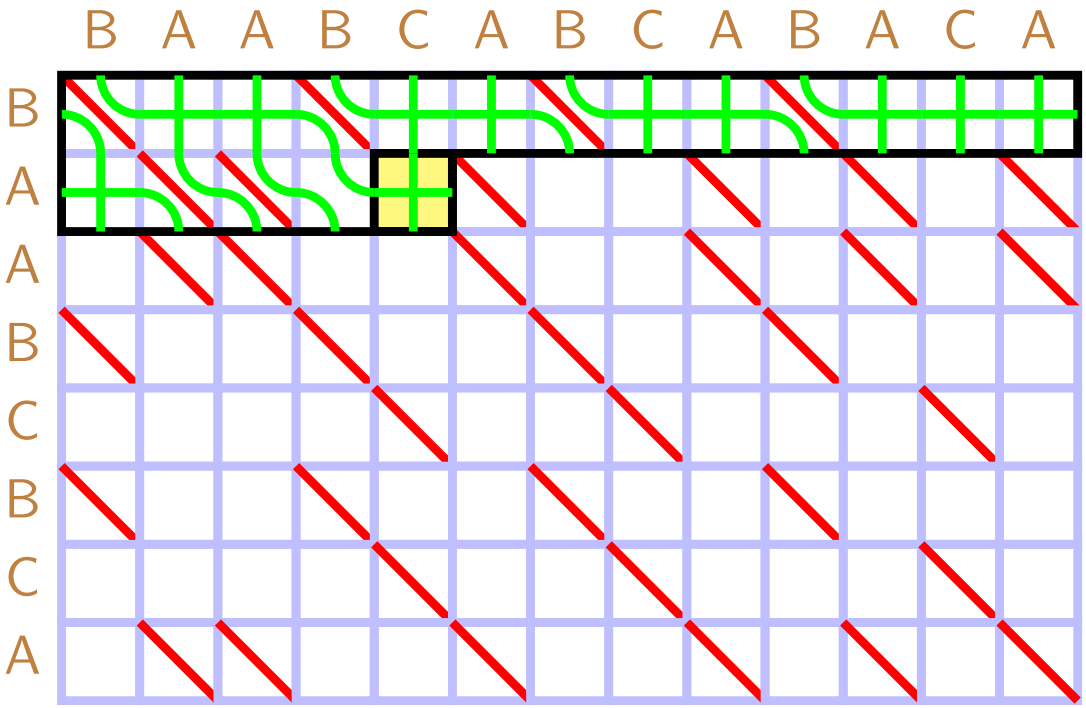
SLCS by iterative combing



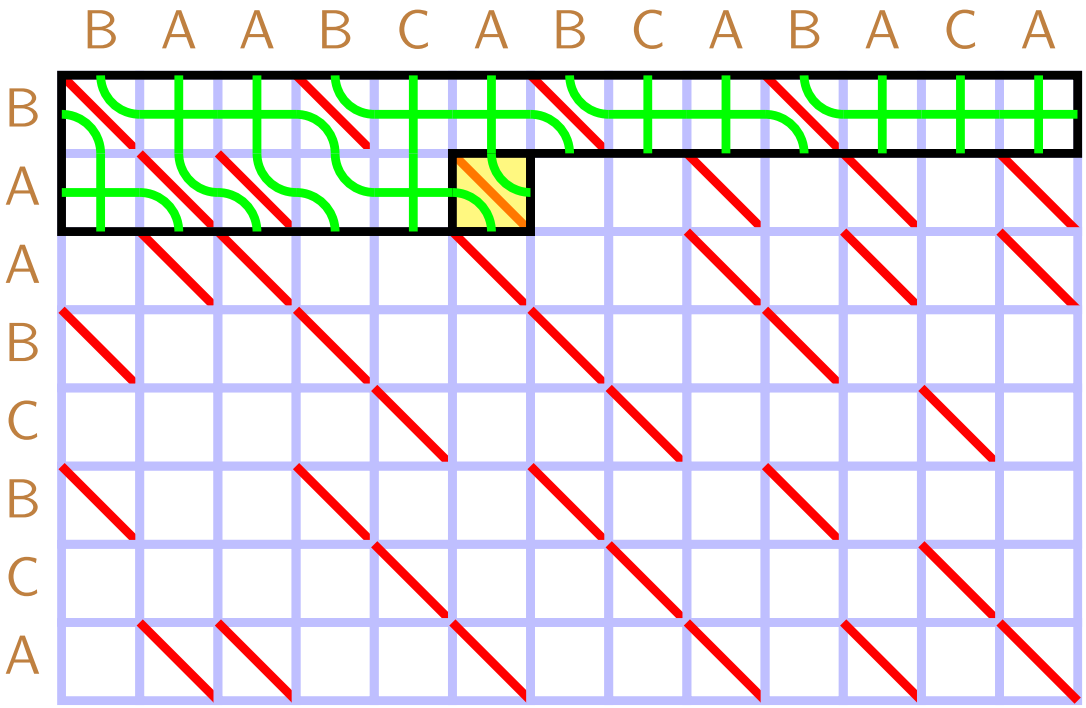
SLCS by iterative combing



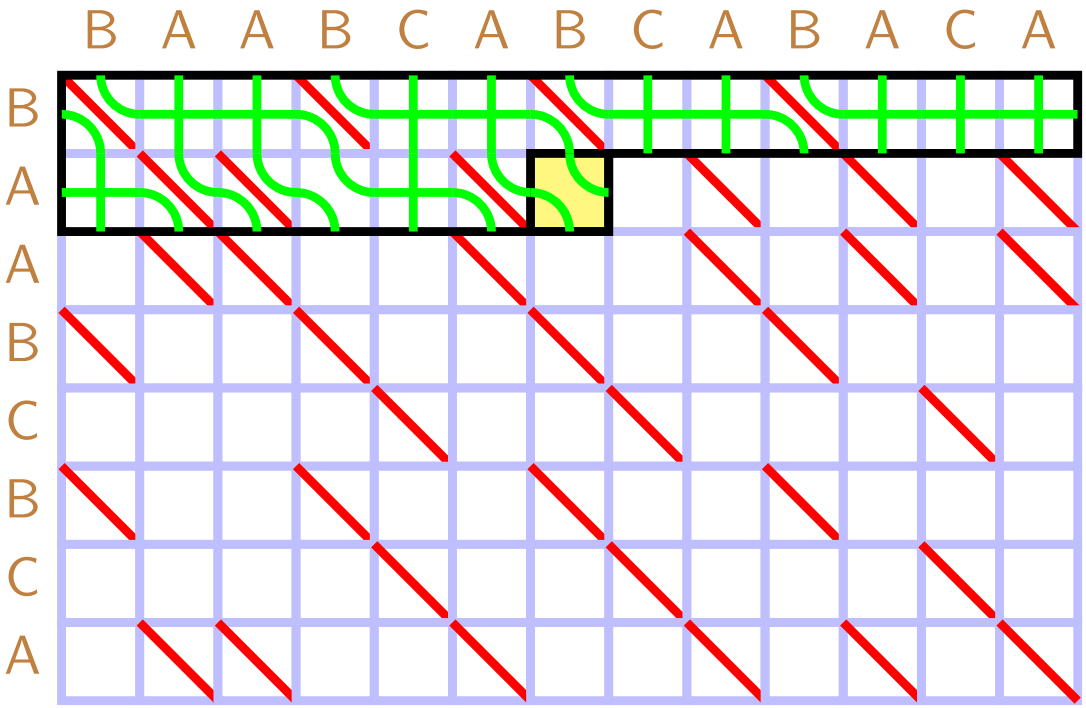
SLCS by iterative combing



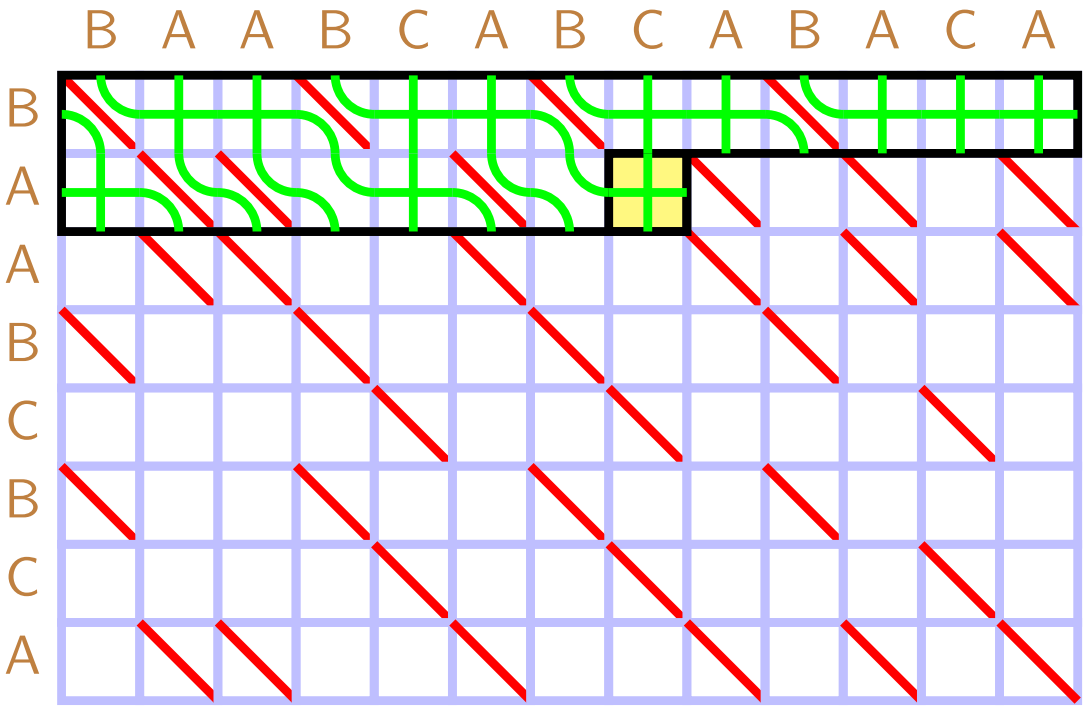
SLCS by iterative combing



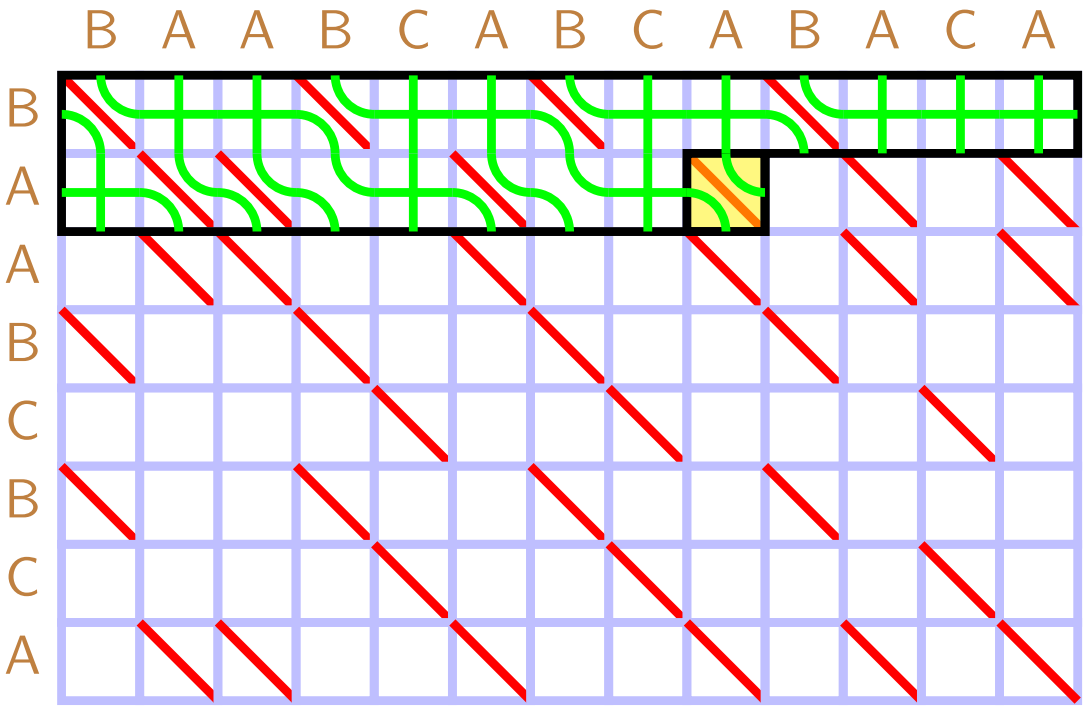
SLCS by iterative combing



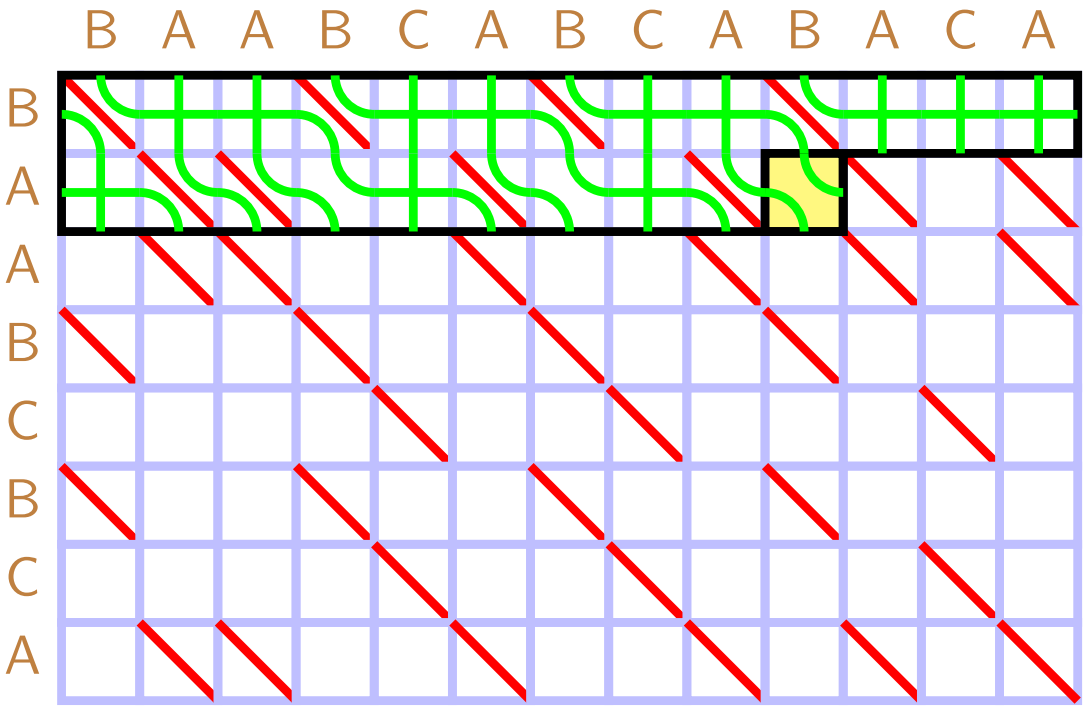
SLCS by iterative combing



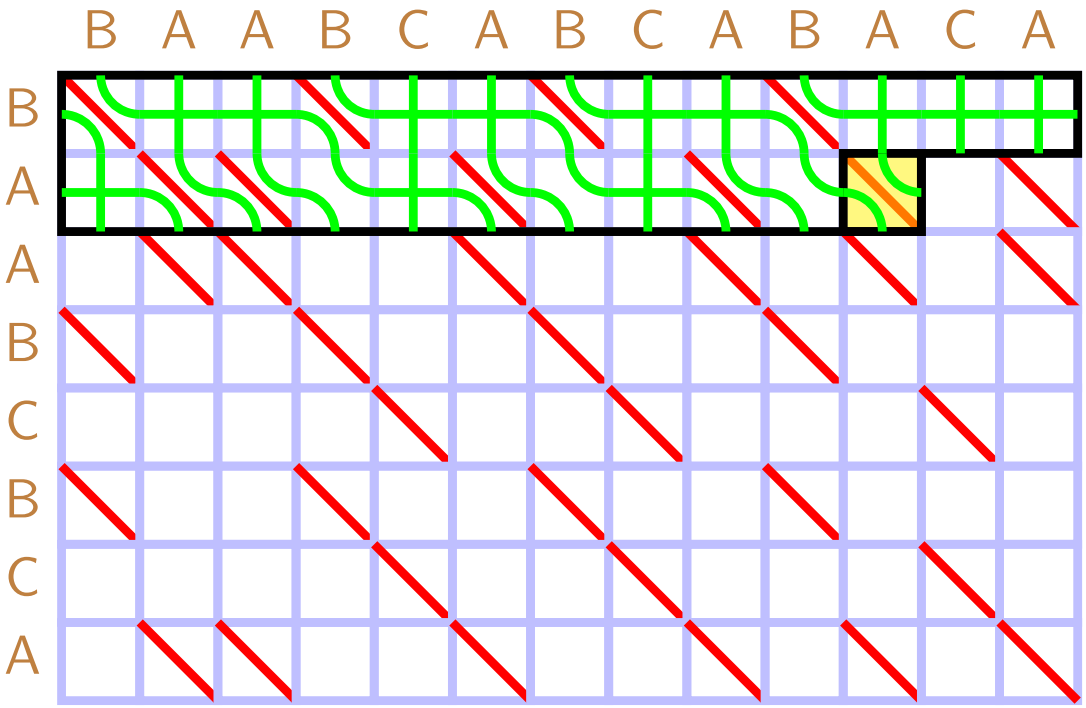
SLCS by iterative combing



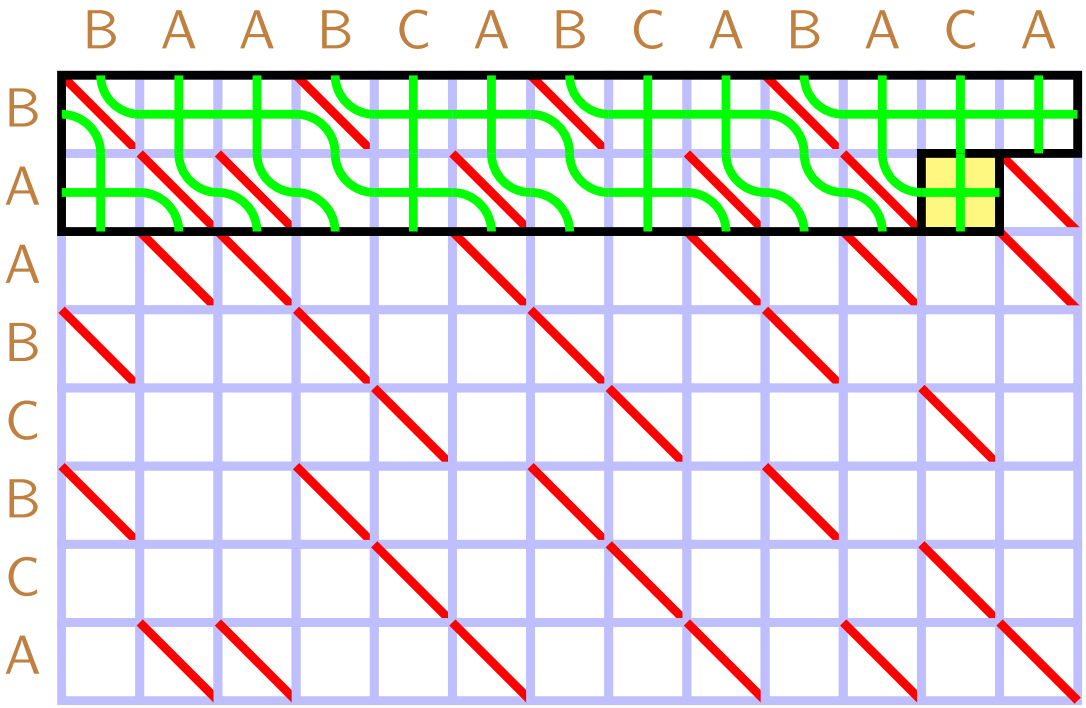
SLCS by iterative combing



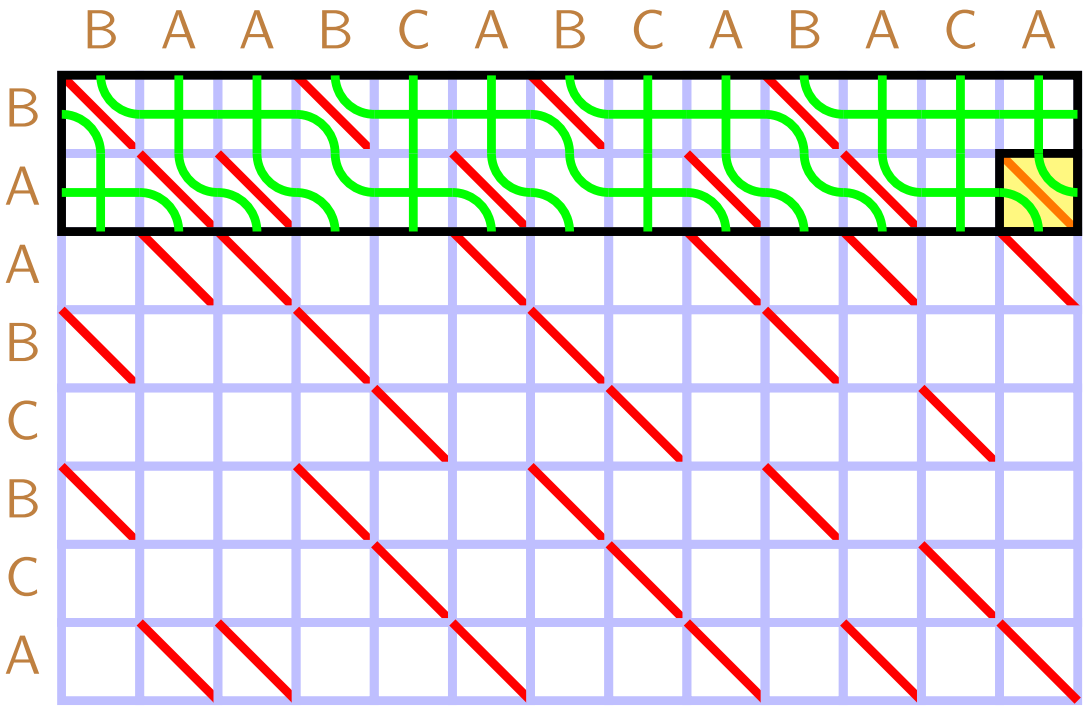
SLCS by iterative combing



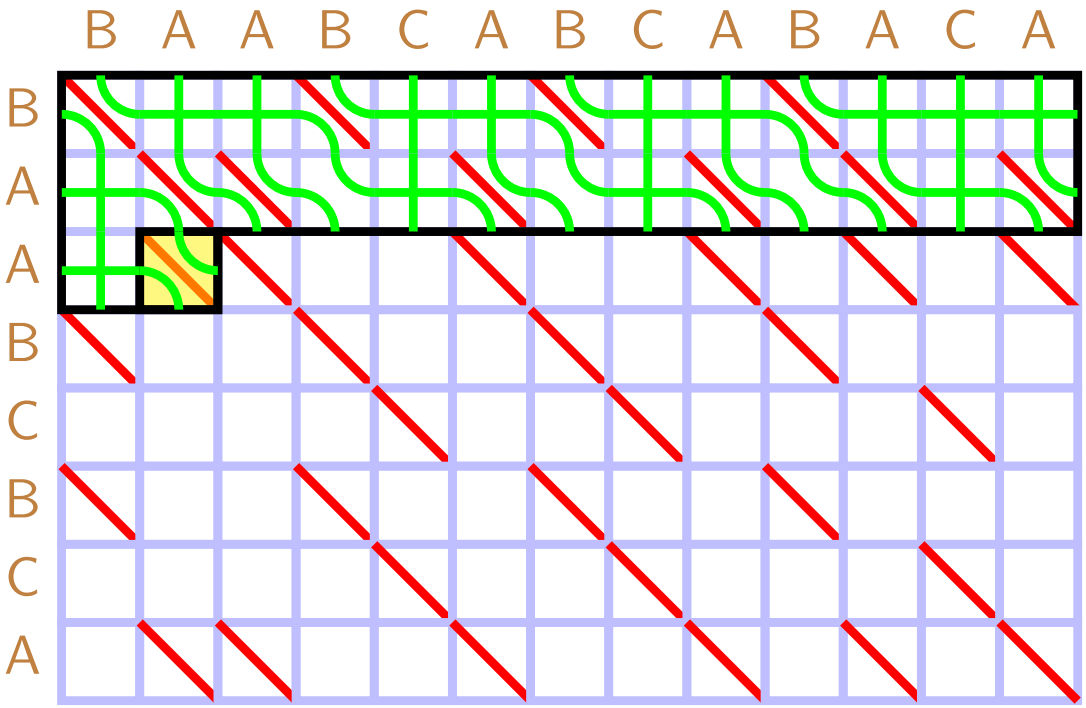
SLCS by iterative combing



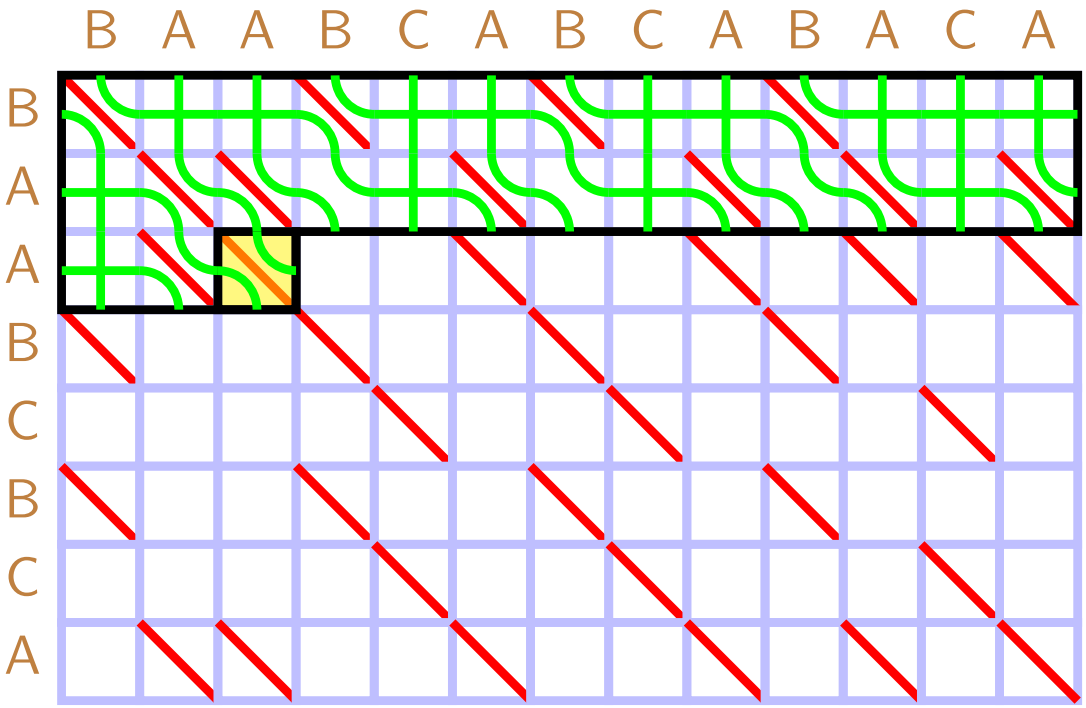
SLCS by iterative combing



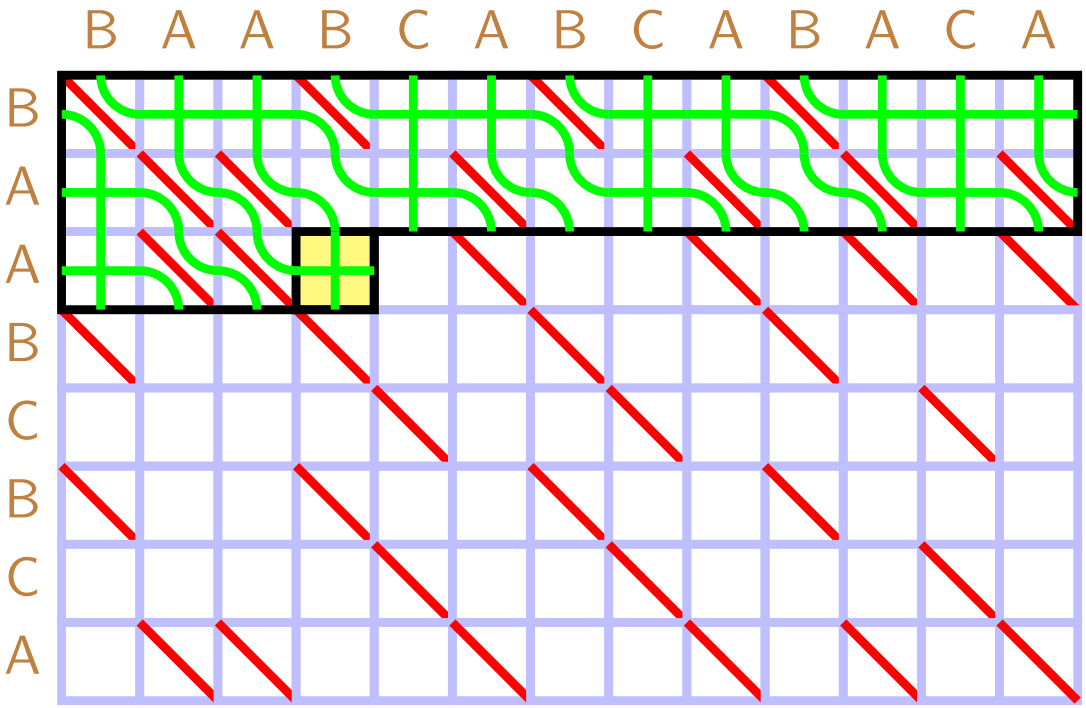
SLCS by iterative combing



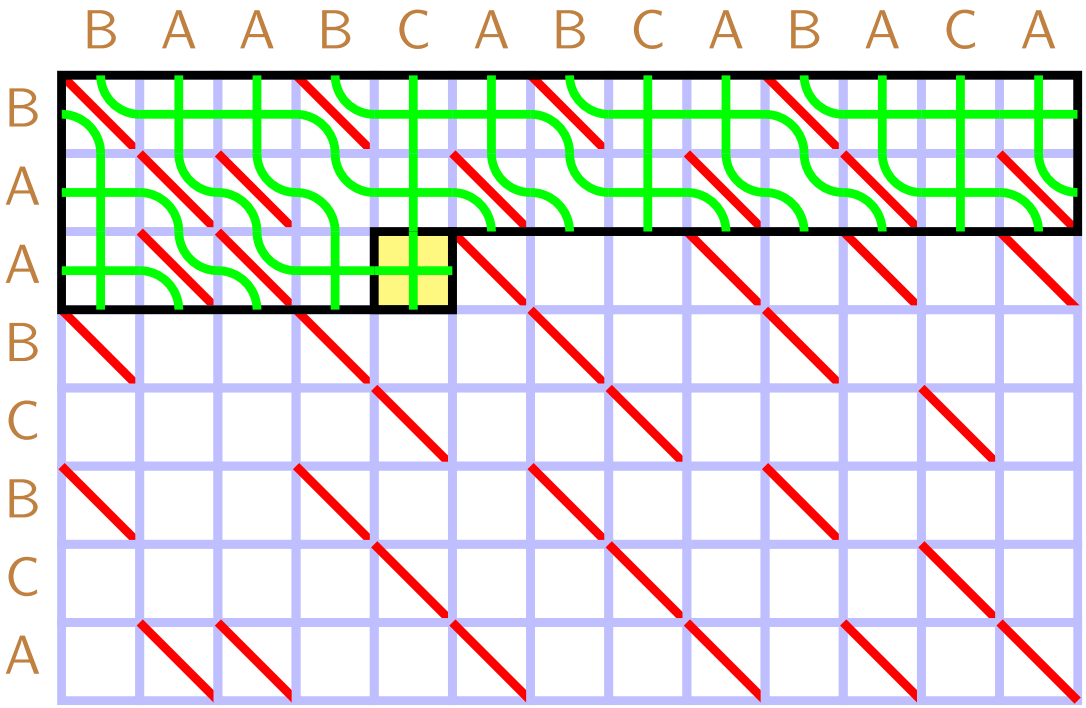
SLCS by iterative combing



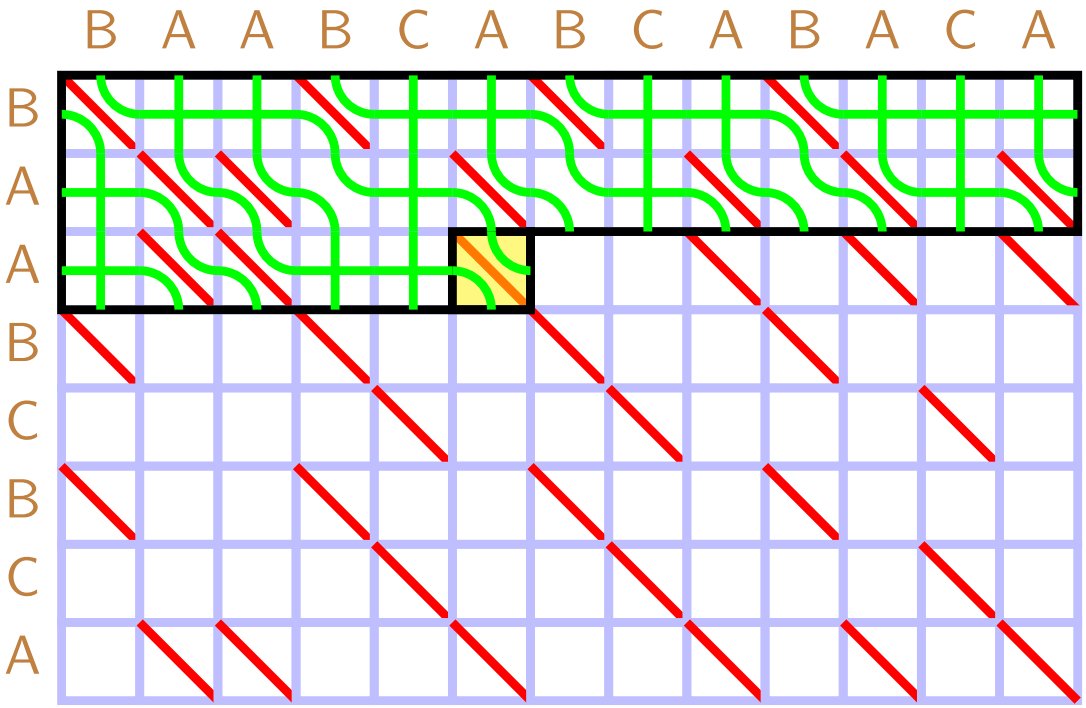
SLCS by iterative combing



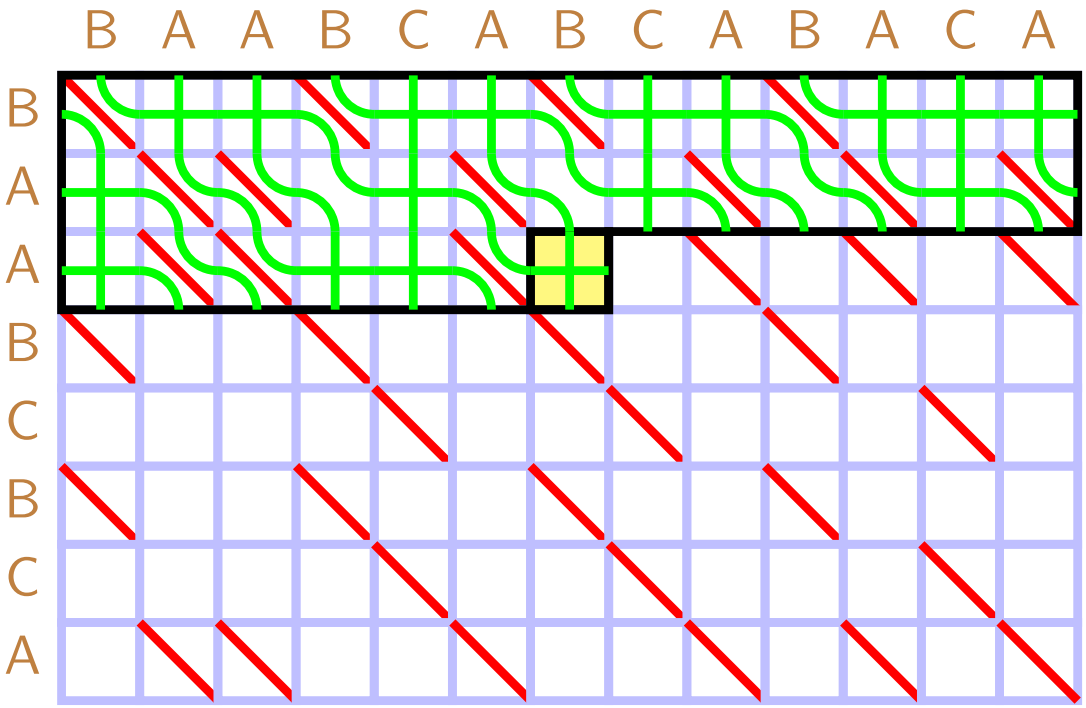
SLCS by iterative combing



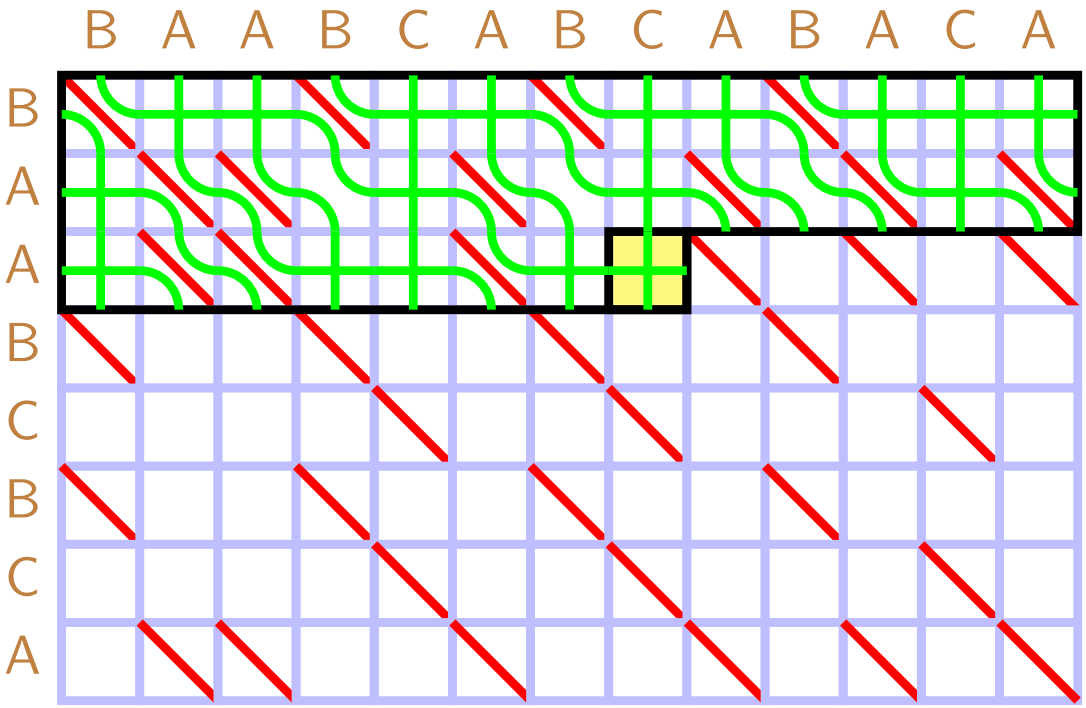
SLCS by iterative combing



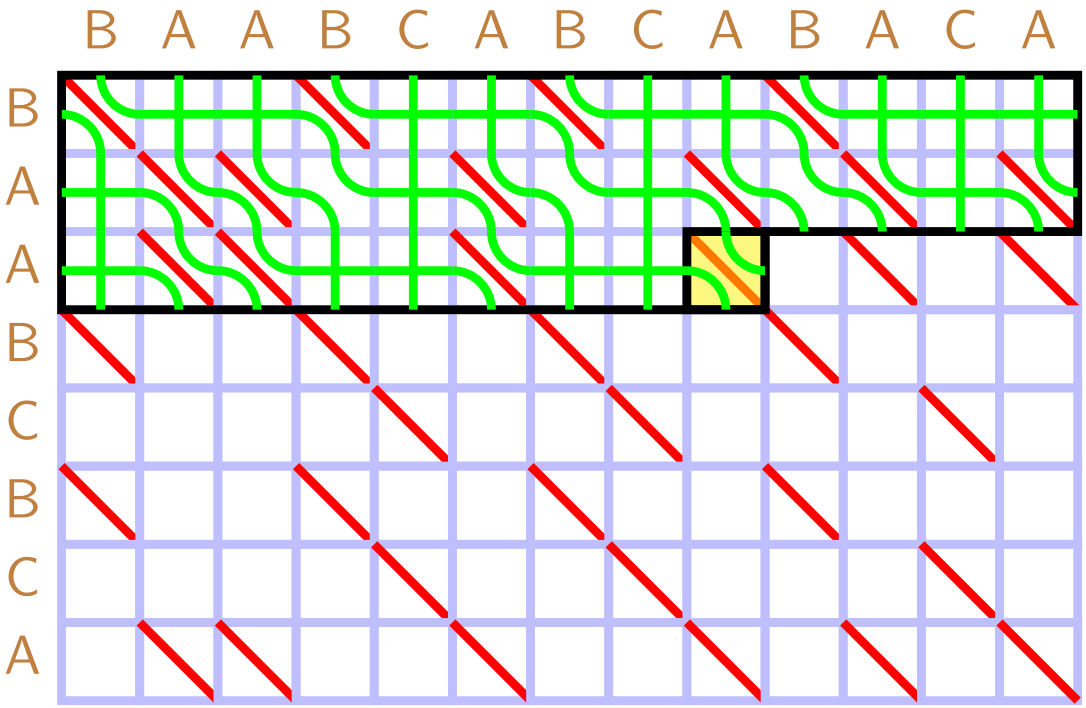
SLCS by iterative combing



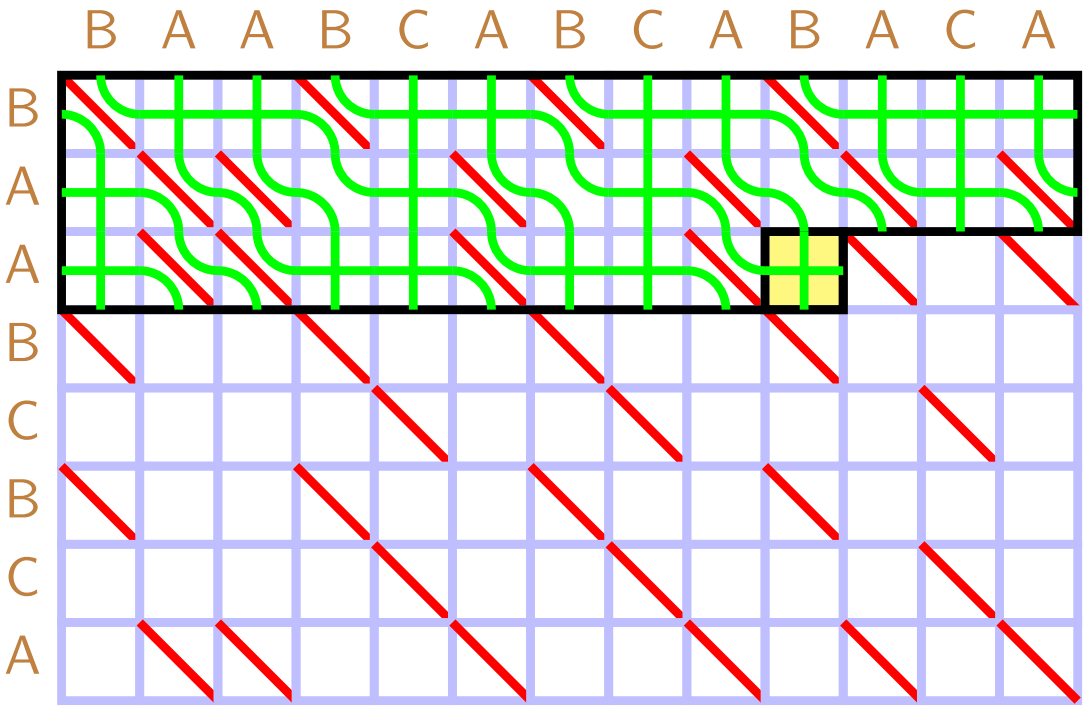
SLCS by iterative combing



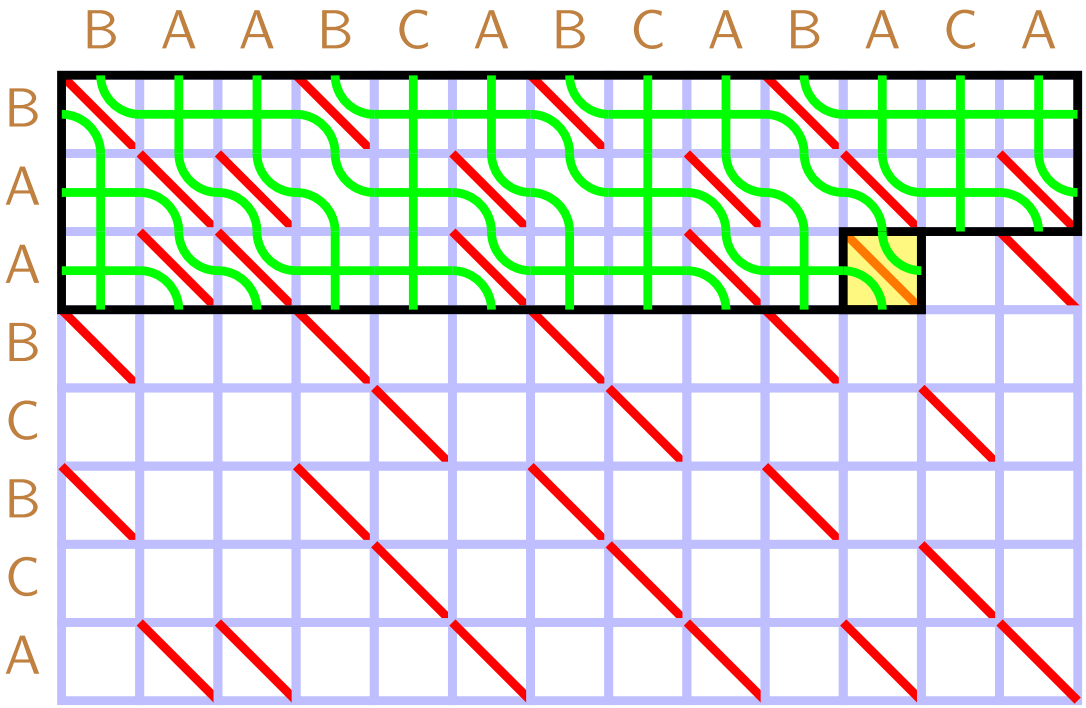
SLCS by iterative combing



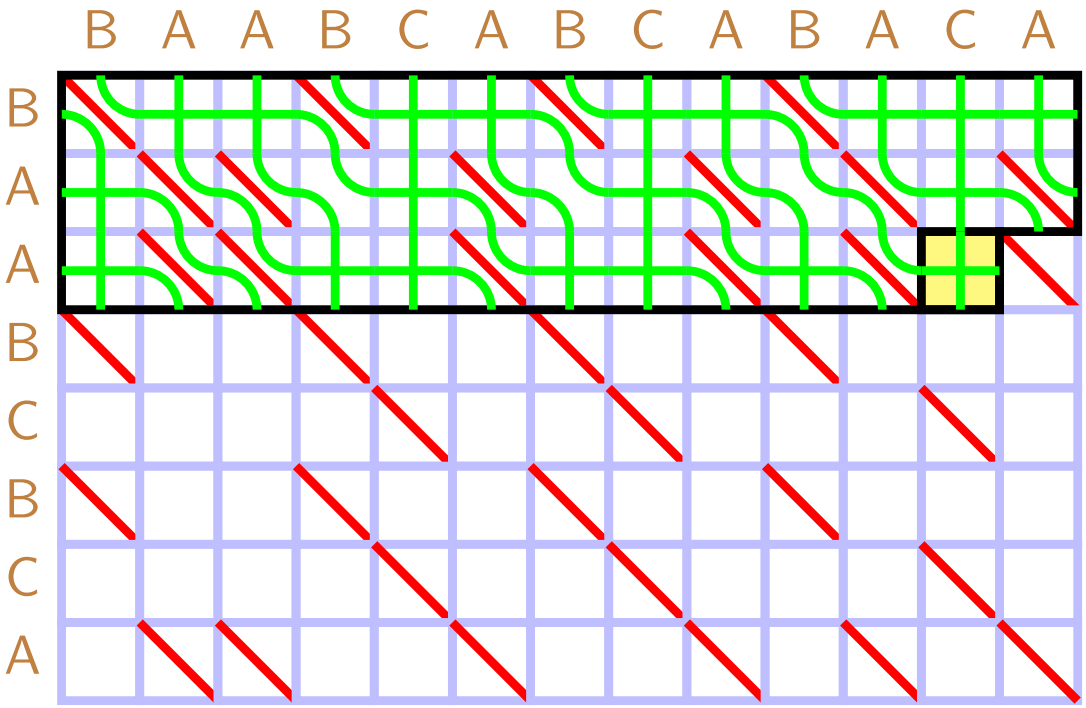
SLCS by iterative combing



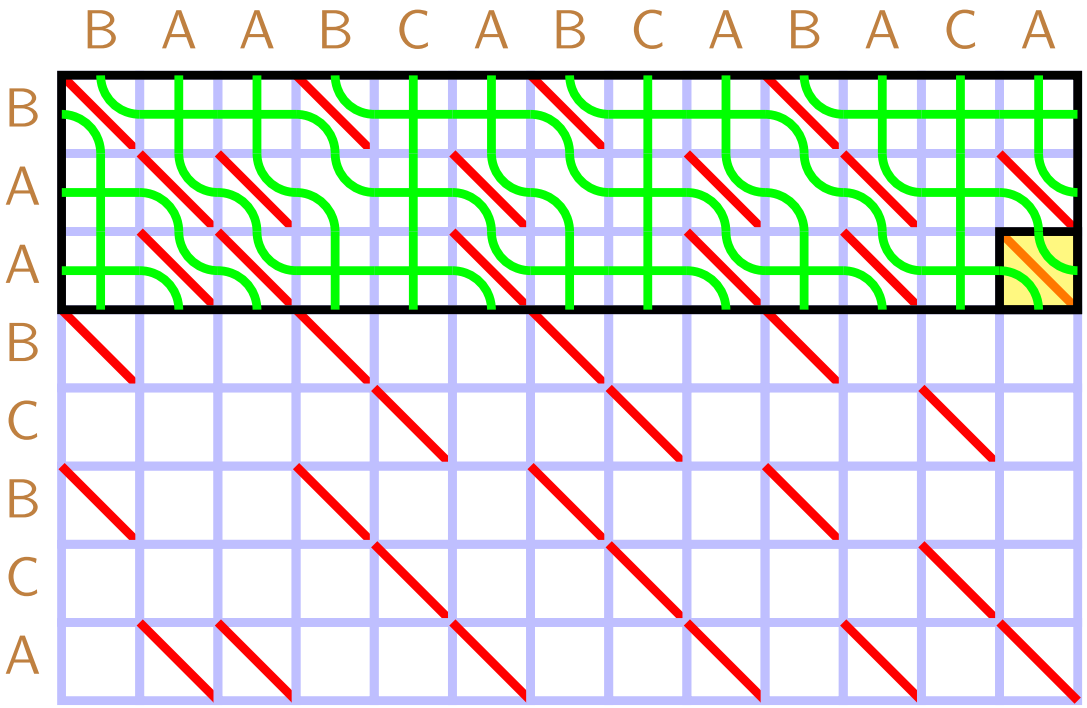
SLCS by iterative combing



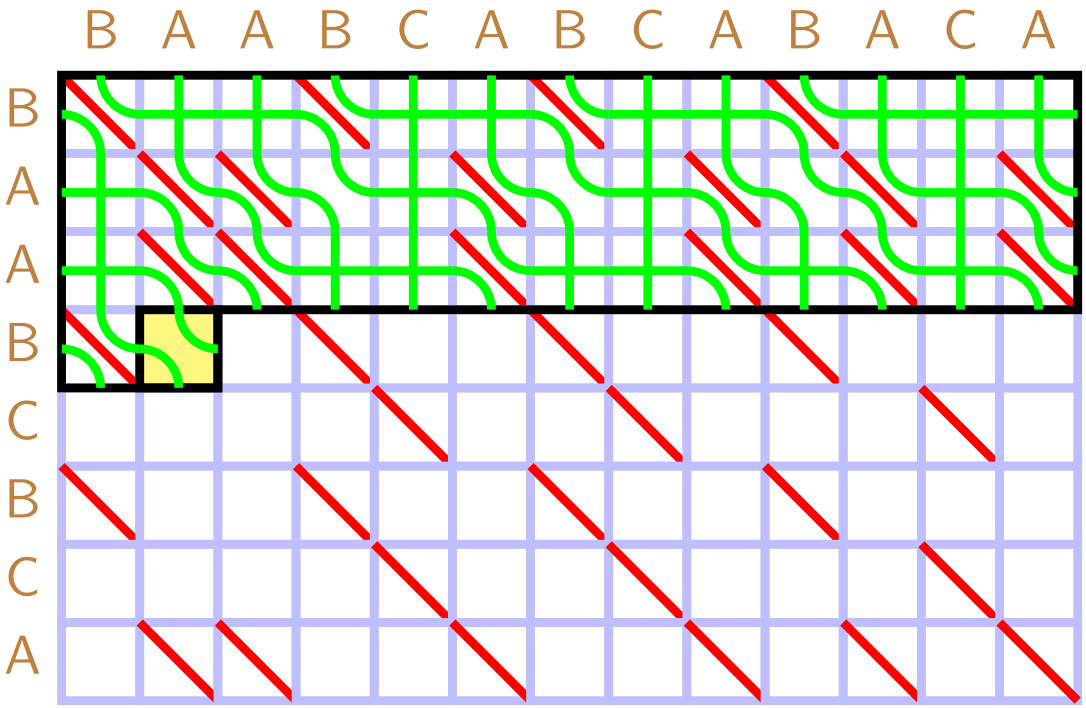
SLCS by iterative combing



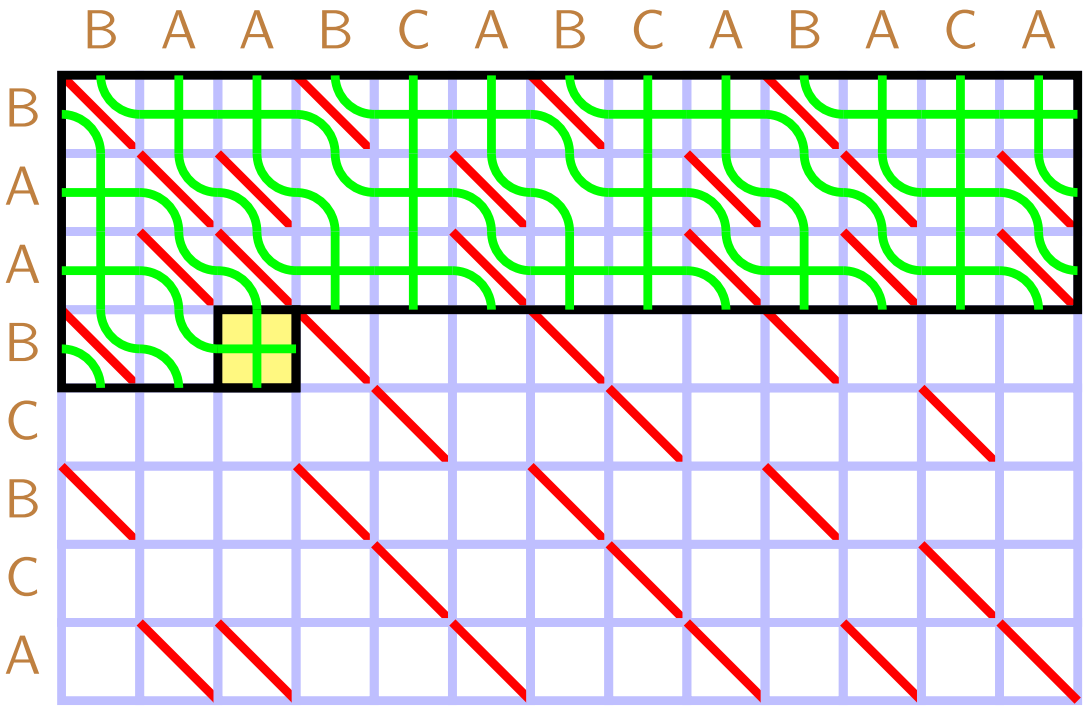
SLCS by iterative combing



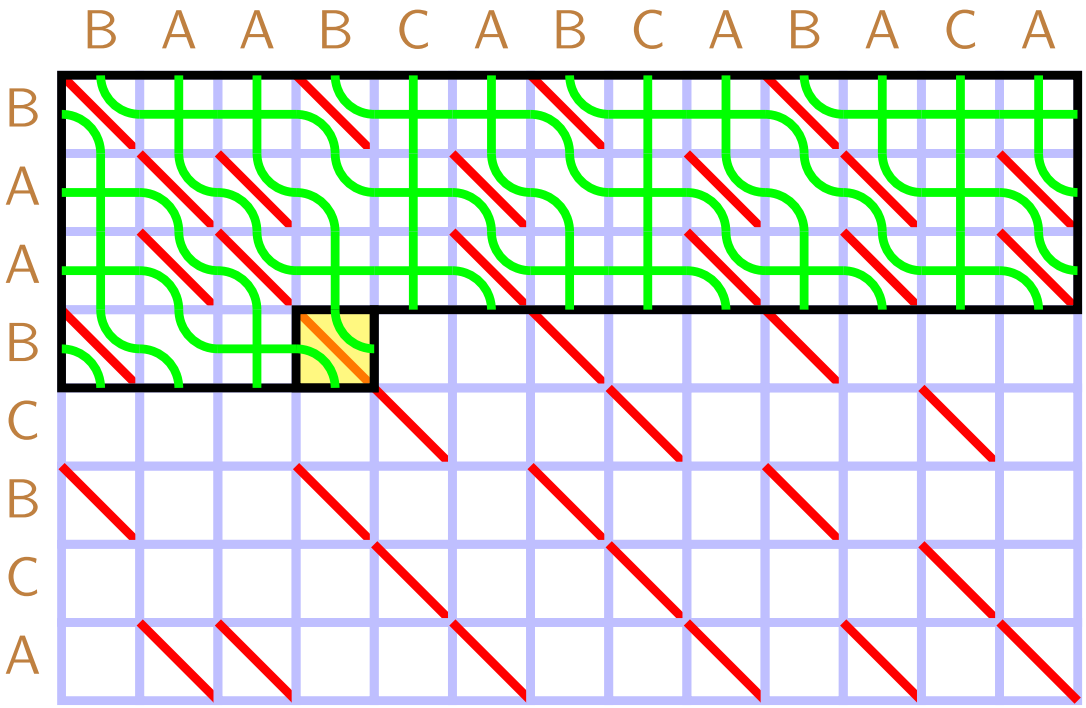
SLCS by iterative combing



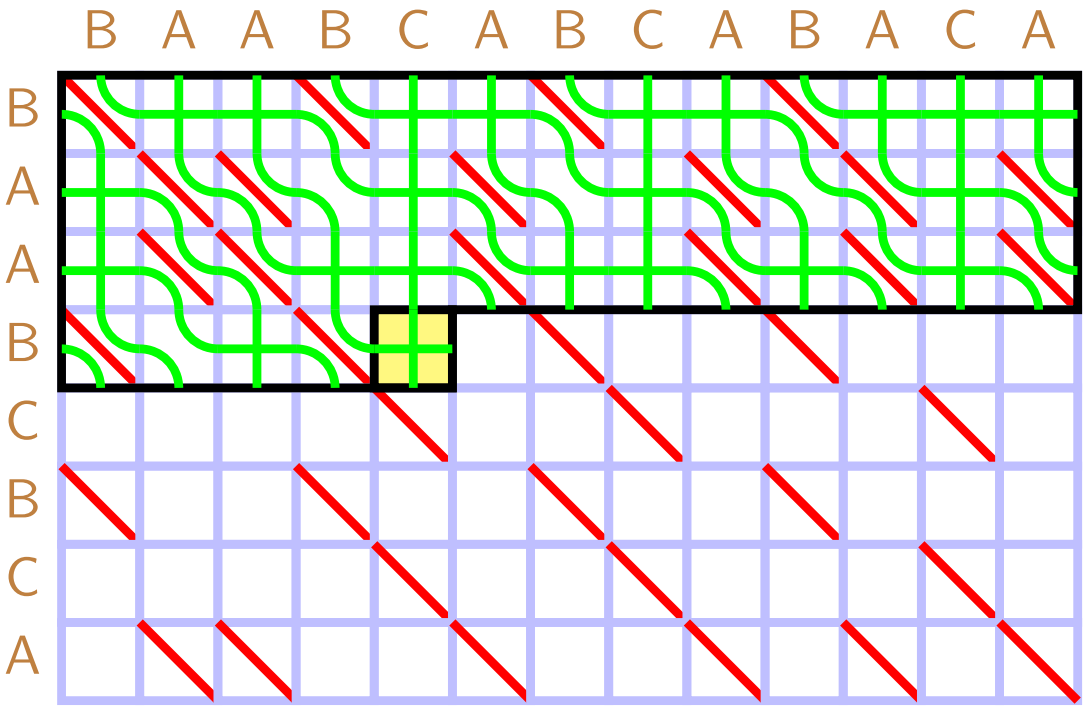
SLCS by iterative combing



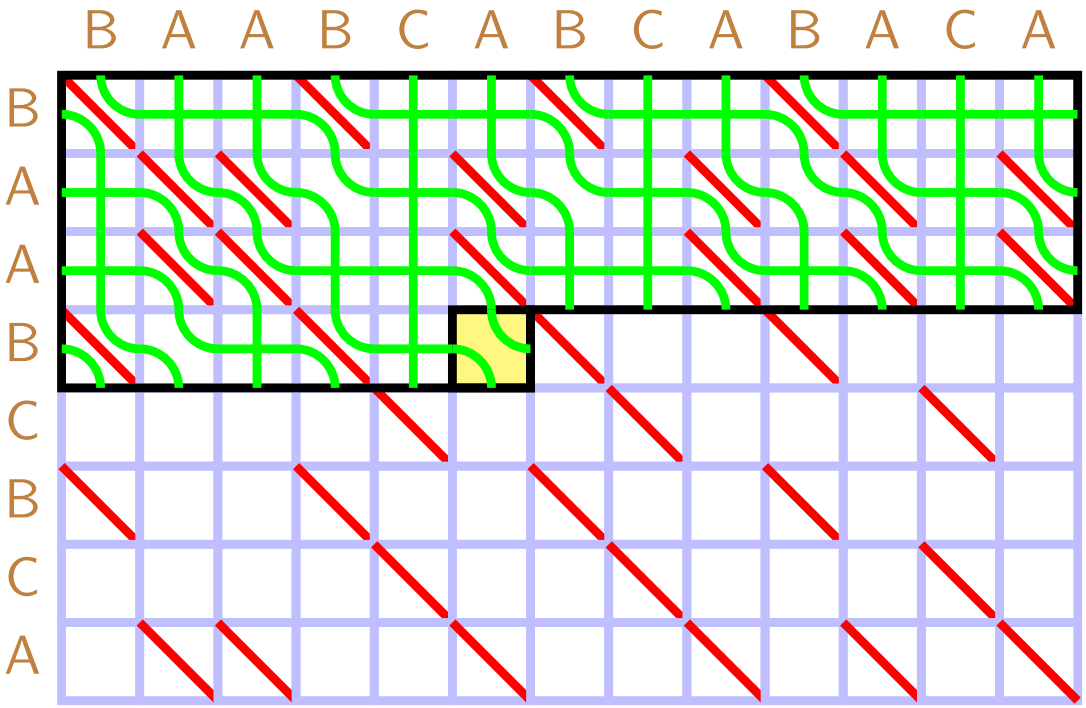
SLCS by iterative combing



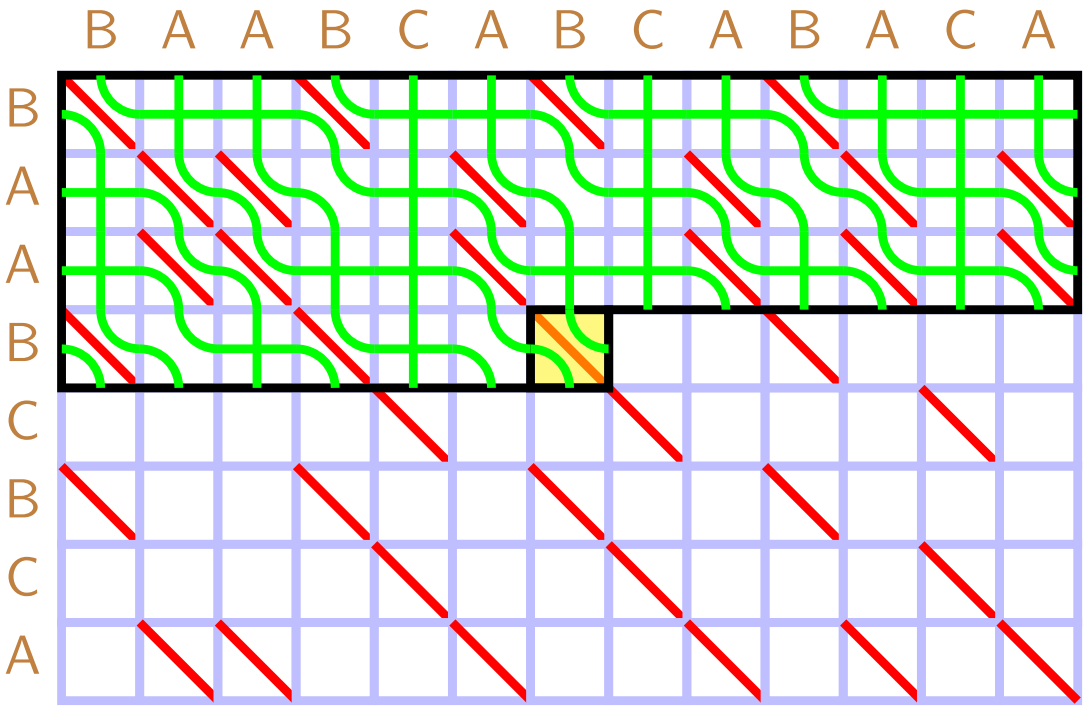
SLCS by iterative combing



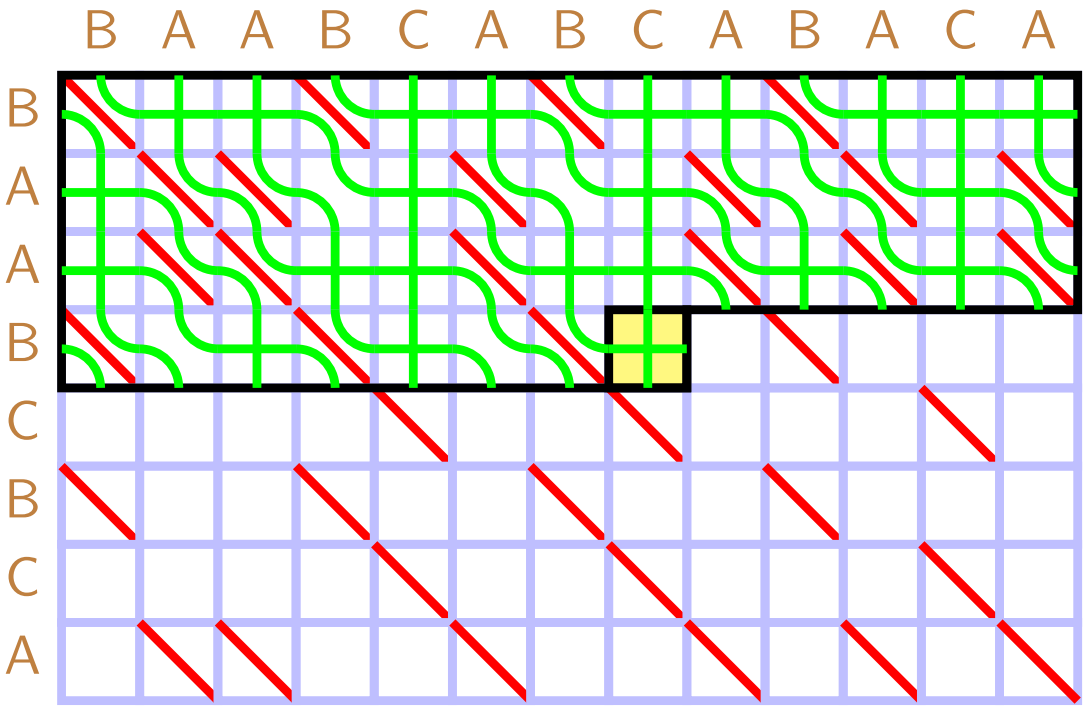
SLCS by iterative combing



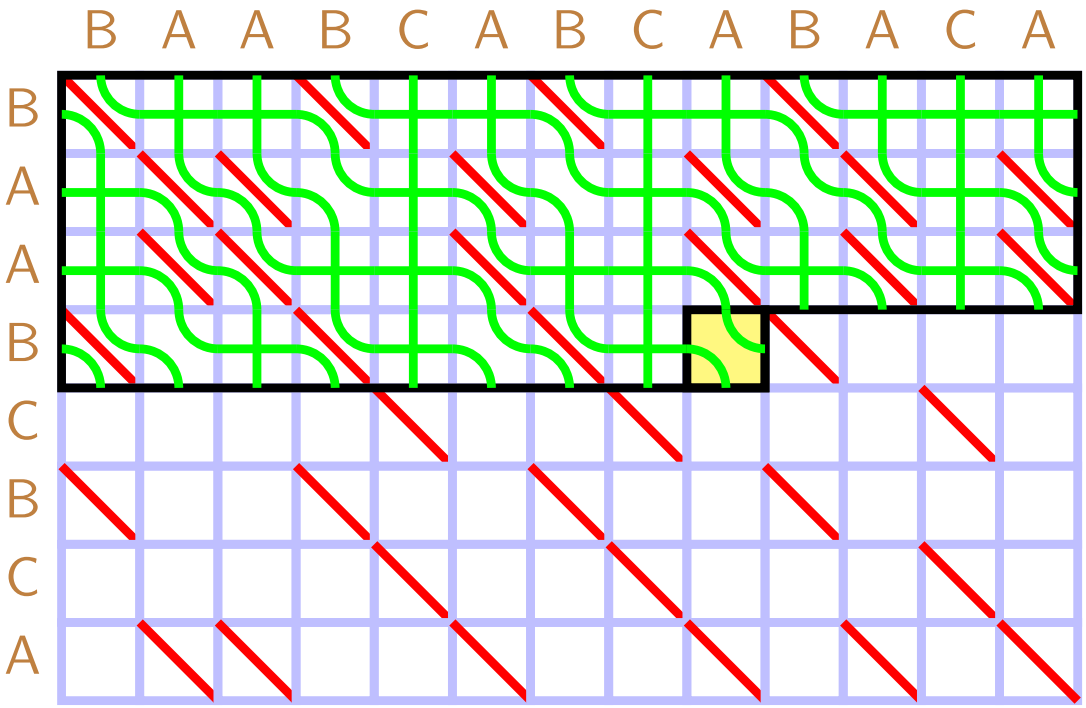
SLCS by iterative combing



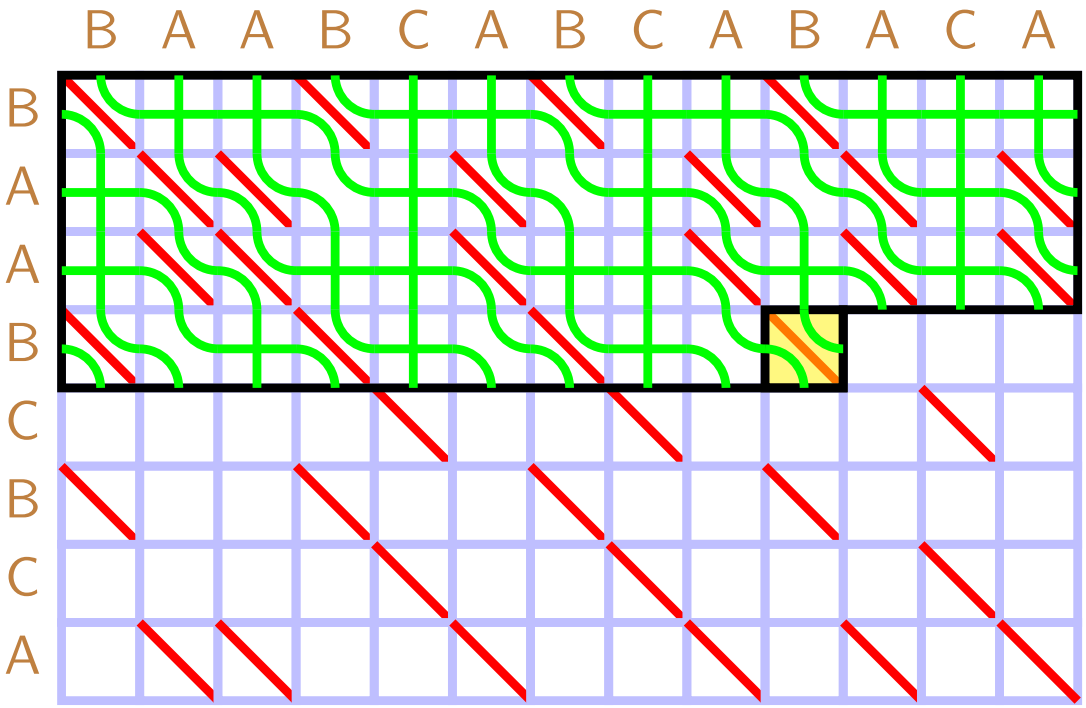
SLCS by iterative combing



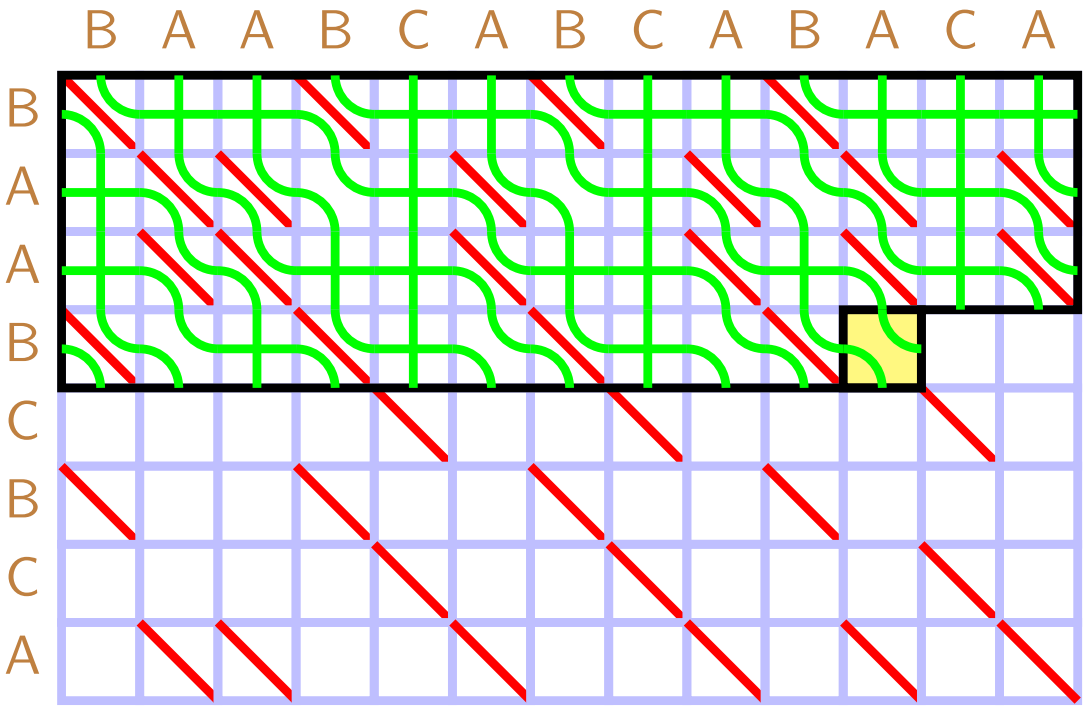
SLCS by iterative combing



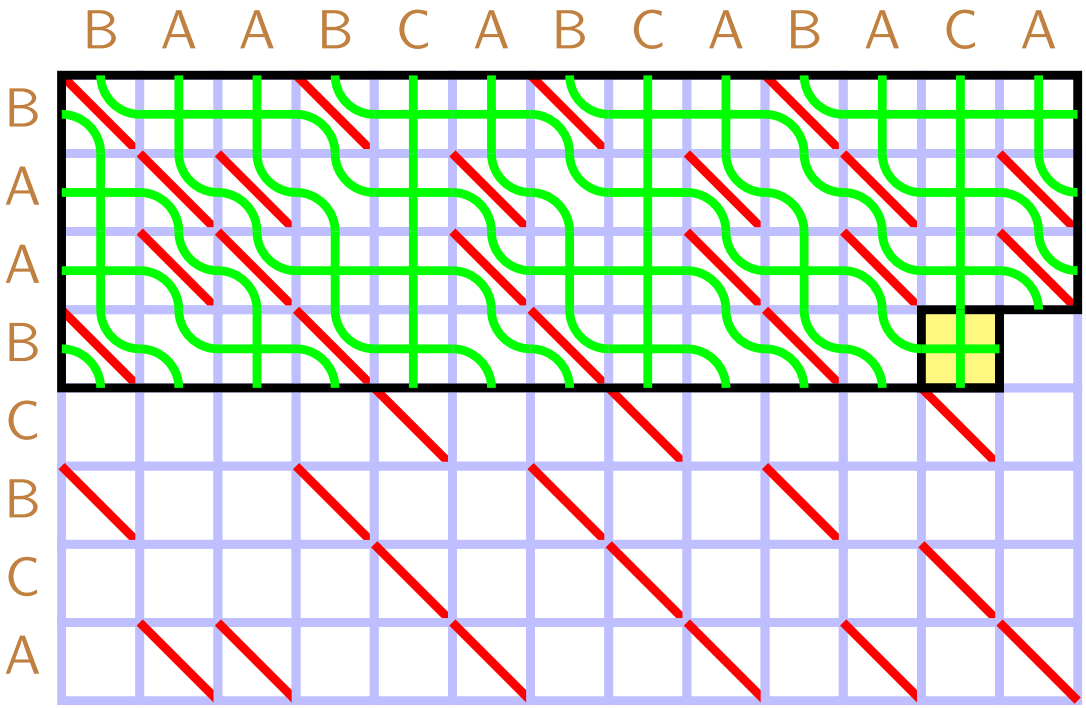
SLCS by iterative combing



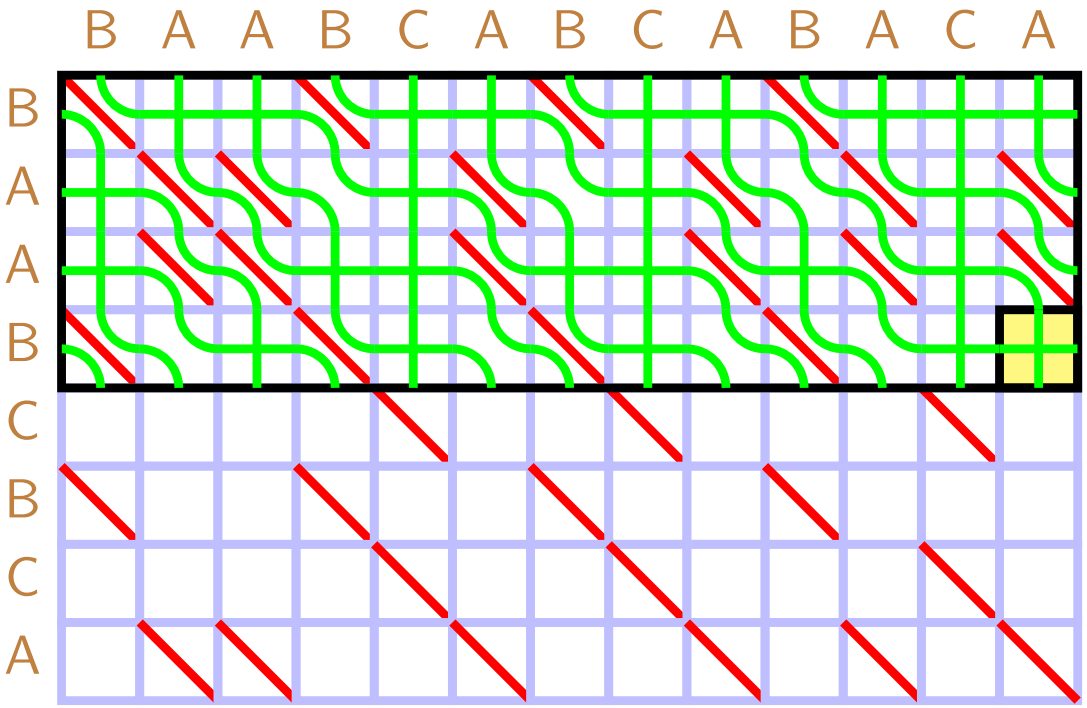
SLCS by iterative combing



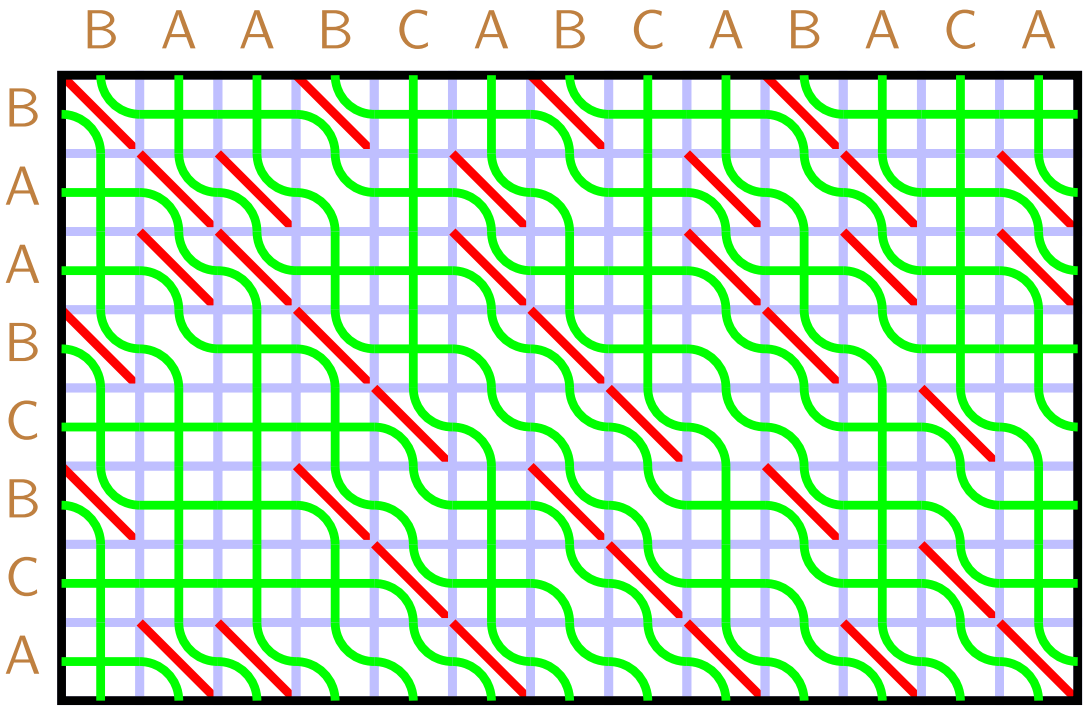
SLCS by iterative combing



SLCS by iterative combing



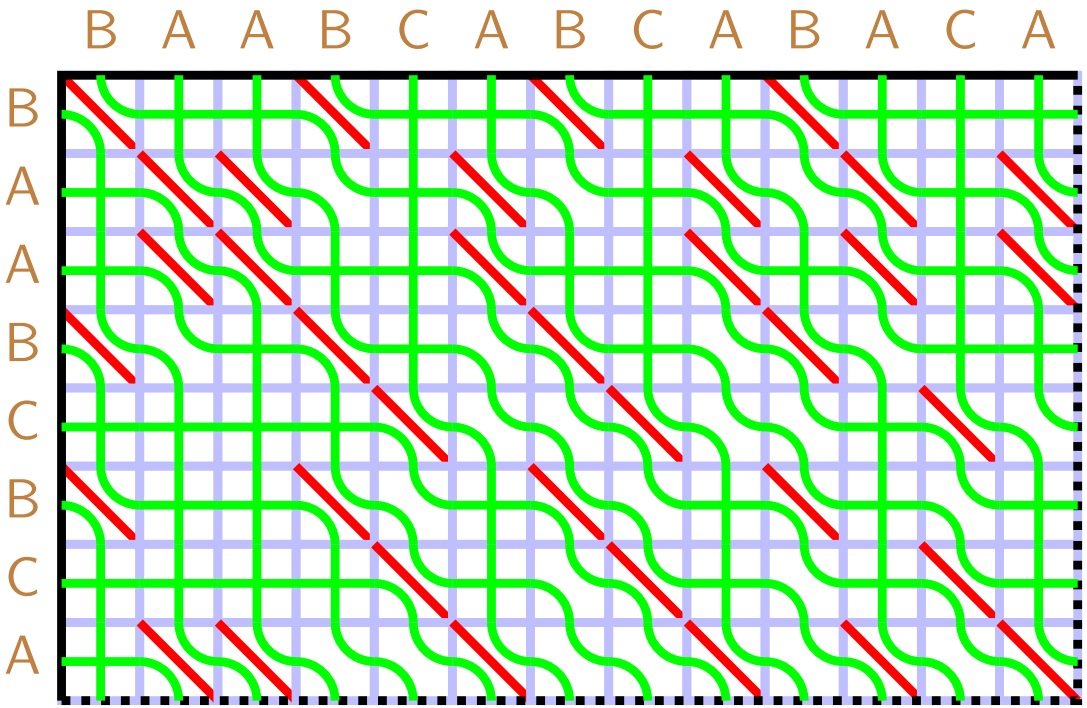
SLCS by iterative combing



Fundamentals of string comparison

Algorithms for SLCS

SLCS by iterative combing



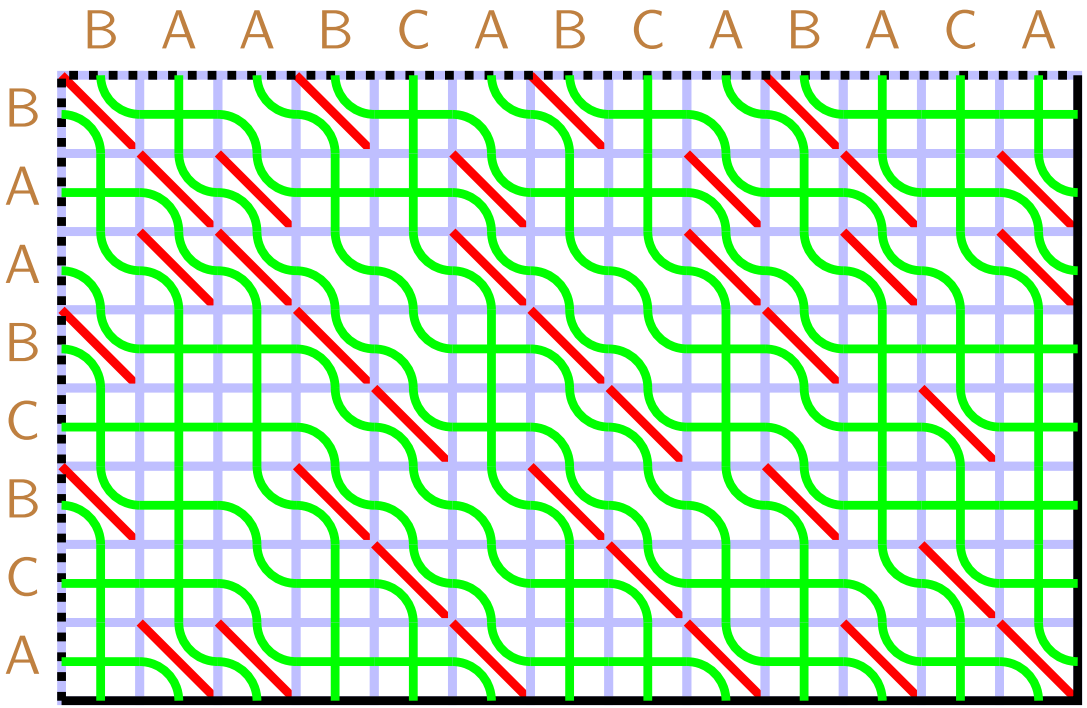
Positively reduced braid: SLCS kernel for every prefix of a vs every prefix of b

Implicit prefix-substring and substring-prefix LCS

Fundamentals of string comparison

Algorithms for SLCS

SLCS by iterative combing (reversed)



Negatively reduced braid: SLCS kernel for every suffix of a vs every suffix of b

Implicit suffix-substring and substring-suffix LCS (dodrans-local LCS)

SLCS: iterative combing

Initialise as saturated braid: mismatch cell = crossing

Iterate over cells in any $\ll$-compatible order

- match cell: skip, keep strands untangled
- mismatch cell: untangle strands if crossed before

Active cell update: time $O(1)$

Overall time $O(mn)$

Correctness: by sticky braid relations

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Rational-weighted string comparison

Alignment and edit distance

The LCS problem is a special case of the alignment problem

Scoring scheme: (match w_+, mismatch w_0, gap w_-)

$w_+ \geq 0 \qquad 2w_- \leq w_0 < w_+ \qquad w_- \leq 0$

$\mathrm{align}(a, b) = w_+ \cdot \#\mathrm{matches} + w_0 \cdot \#\mathrm{mismatches} + w_- \cdot \#\mathrm{gaps}$

LCS score: $(1, 0, 0) \qquad \mathrm{lcs}(a, b) = \#\mathrm{matches}$

Levenshtein score: $\left(1, \frac{1}{2}, 0\right) \qquad \mathrm{lev}(a, b) = \#\mathrm{matches} + \frac{1}{2} \cdot \#\mathrm{mismatches}$

Scoring scheme is

- rational: w_+, w_0, w_- are rational numbers
- regular: $(1, w_0, 0)$, i.e. $w_+ = 1$, $w_- = 0$

Alignment problem

Alignment score for a vs b

Rational-weighted string comparison

Alignment and edit distance

Alignment problem

Alignment score for a vs b

Alignment: running time

$O(mn)$ [Wagner, Fischer: 1974]

$O\left(\frac{mn}{\log n}\right)$ [Crochemore+: 2003]

Semi-local alignment (SA) problem

Analogous to SLCS problem, but replacing LCS scores with alignment scores

Rational-weighted string comparison

Alignment and edit distance

Edit distance: minimum cost to transform a into b by character edits

Substitution (sub) cost $c_0 > 0$
Insertion/deletion (indel) cost $c_- > 0$

Corresponds to scoring scheme $(0, -c_0, -c_-)$

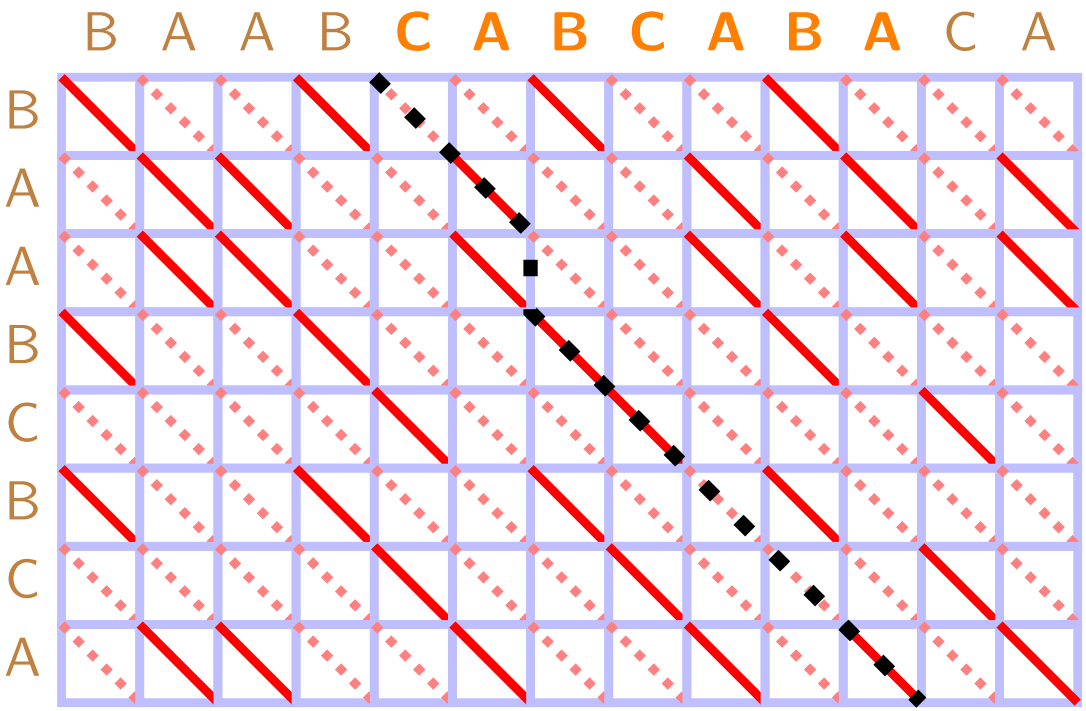
LCS (indel) distance: $c_- = 1$, $c_0 = 2$ $\quad$ $\mathrm{d\_lcs}(a, b) = \#\mathrm{indels}$
scoring scheme $(0, -2, -1) \rightsquigarrow$ regular $(1, 0, 0)$

Levenshtein (indelsub) distance: $c_- = c_0 = 1$ $\quad$ $\mathrm{d\_lev}(a, b) = \#\mathrm{indels} + \#\mathrm{subs}$
scoring scheme $(0, -1, -1) \rightsquigarrow$ regular $\left(1, \frac{1}{2}, 0\right)$

Rational-weighted string comparison

Alignment and edit distance

SA as maximum paths in the alignment grid



$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

blue $= 0$ (gap)

red (dotted) $= \frac{1}{2}$ (mismatch)

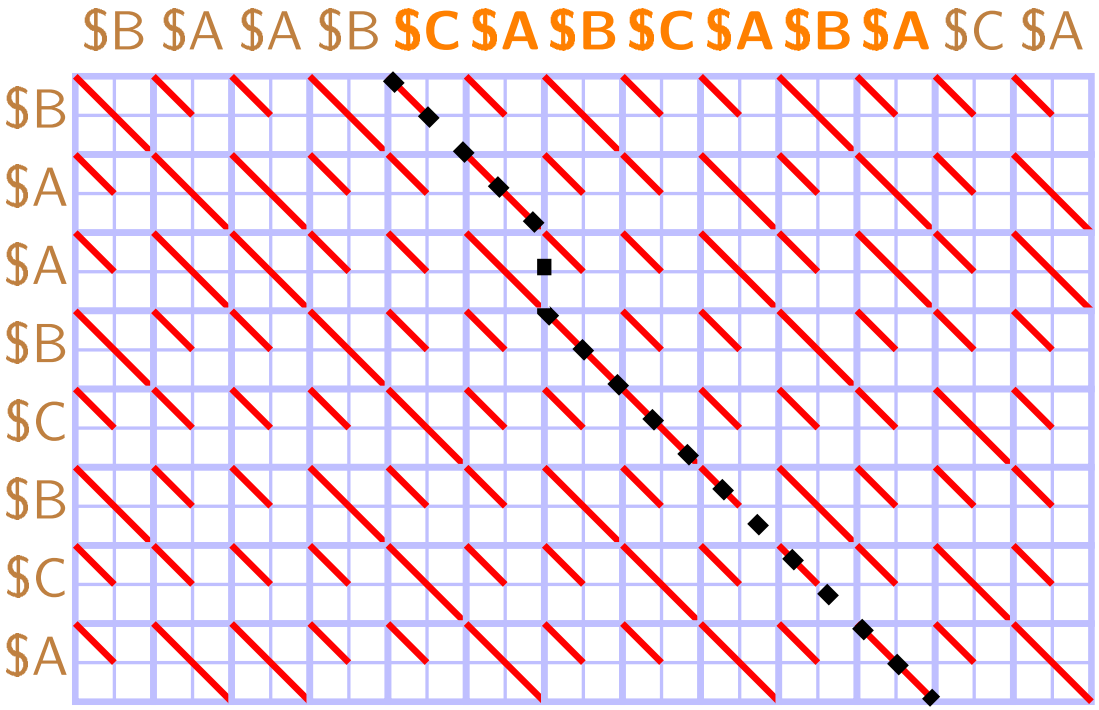
red $= 1$ (match)

$\text{align}(a, b\langle 4 : 11\rangle) = 5.5$

Rational-weighted string comparison

Alignment and edit distance

Equivalent to semi-local LCS for blown-up strings



$a =$ "BAABCBCA"

$b =$ "BAAB**CABCABA**CA"

$blue = 0$ (gap)

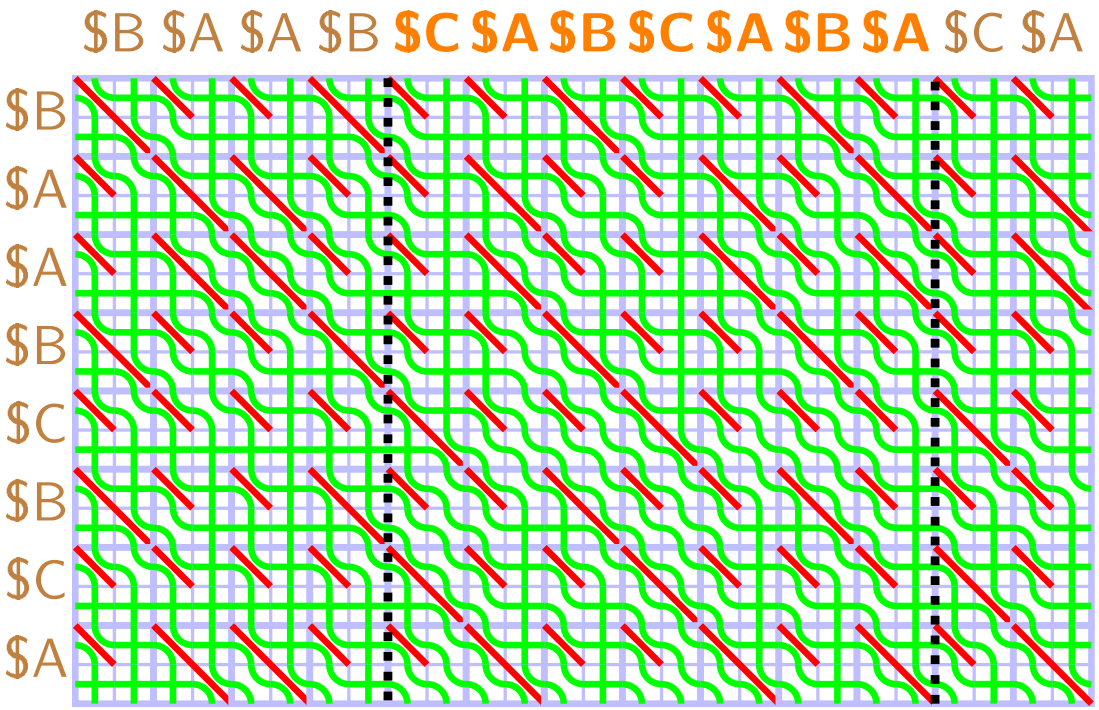
$red = 1$ (match)

$\mathrm{align}(a, b\langle 4 : 11\rangle) =$
$\frac{1}{2} \cdot \mathrm{lcs}(\tilde{a}, \tilde{b}\langle 2 \cdot 4 : 2 \cdot 11\rangle) =$
$\frac{1}{2} \cdot 11 = 5.5$

Rational-weighted string comparison

Alignment and edit distance

Rational-weighted SA via semi-local LCS



a, b, scheme $\left(1, \frac{\mu}{\nu}, 0\right) \rightsquigarrow \tilde{a}$, $\tilde{b}$, scheme $(\nu, \mu, 0)$

Slowdown $\times \nu^2$, can be reduced to $\times \nu$

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Cyclic and periodic string comparison

Affine sticky braids

Affine permutation of order n

- bijection $\langle -\infty : +\infty \rangle \leftrightarrow \langle -\infty : +\infty \rangle$
- invariant under shift by n

Affine permutation matrix of order n

- zero-one matrix over $\langle -\infty : +\infty \rangle^2$
- exactly one nonzero per row/column
- invariant under shift by (n, n)

Affine symmetric group $\tilde{\mathbb{S}}_n$

- multiplication: functional composition; matrix multiplication
- symmetries of a tesselation of $\mathbb{R}^{n-1}$ by simplices

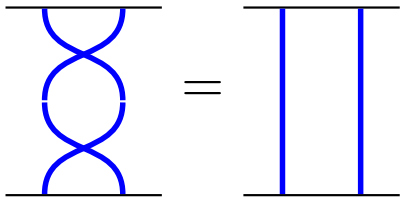
Cyclic and periodic string comparison

Affine sticky braids

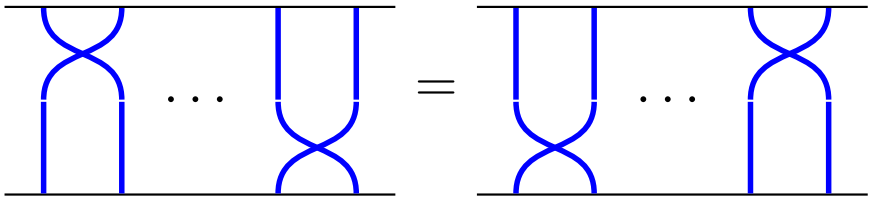
Affine symmetric group $\tilde{\mathbb{S}}_n$ in Coxeter presentation

n generators $\tau_0, \tau_1, \tau_2, \ldots, \tau_{n-1}$ (elementary transpositions)

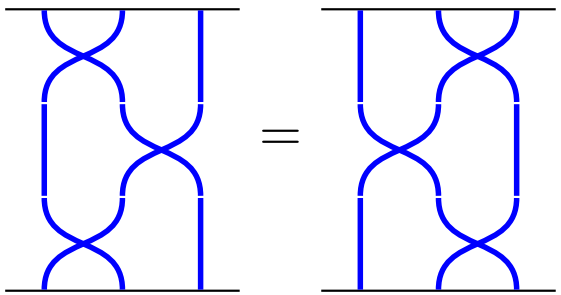
Involution:
$\tau_i^2 = 1 \quad$ for all i



Far commutativity:
$\tau_i \tau_j = \tau_j \tau_i \quad 1 < j - i < n - 1$



Braid relations:
$\tau_i \tau_j \tau_i = \tau_j \tau_i \tau_j \quad j - i \in \{1, n-1\}$



$|\tilde{\mathbb{S}}_n| = \infty$

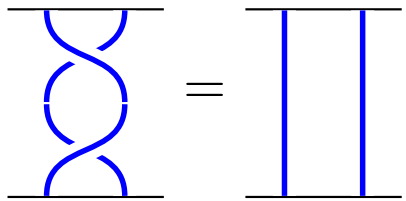
Cyclic and periodic string comparison

Affine sticky braids

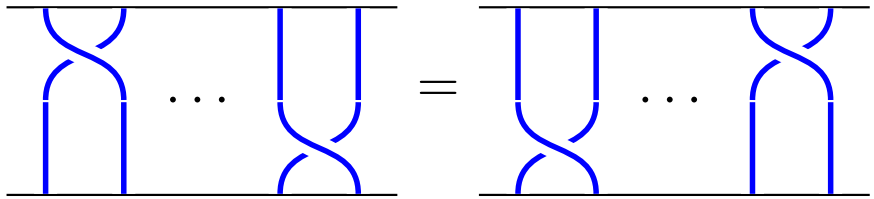
Affine braid group $\tilde{\mathbb{B}}_n$ in Coxeter presentation

n generators $\tau_0, \tau_1, \tau_2, \ldots, \tau_{n-1}$ (elementary transpositions)

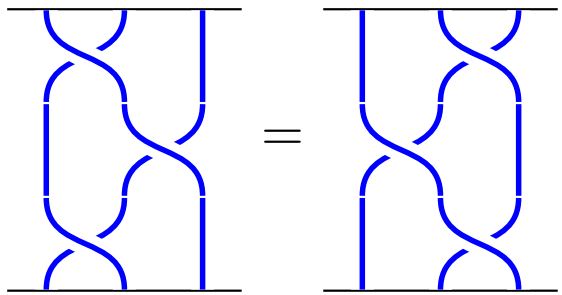
Inversion:
$\tau_i \tau_i^{-1} = 1 \quad$ for all i



Far commutativity:
$\tau_i \tau_j = \tau_j \tau_i \quad 1 < j - i < n - 1$



Braid relations:
$\tau_i \tau_j \tau_i = \tau_j \tau_i \tau_j \quad j - i \in \{1, n-1\}$



$|\tilde{\mathbb{B}}_n| = \infty \qquad$ canonical projection $\tilde{\mathbb{B}}_n \to \tilde{\mathbb{S}}_n$

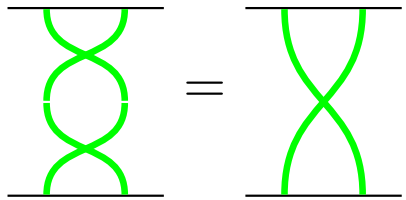
Cyclic and periodic string comparison

Affine sticky braids

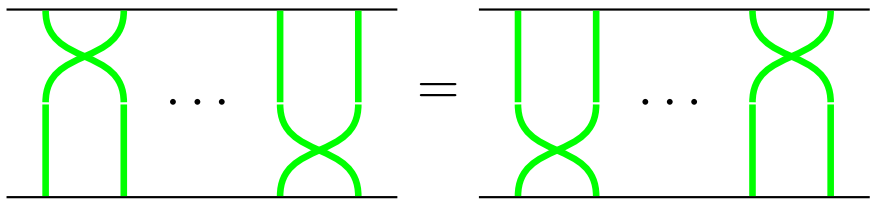
Affine sticky braid (Hecke) monoid $\tilde{\mathbb{H}}_n$ in Coxeter presentation

n generators $\tau_0, \tau_1, \tau_2, \ldots, \tau_{n-1}$ (elementary transpositions)

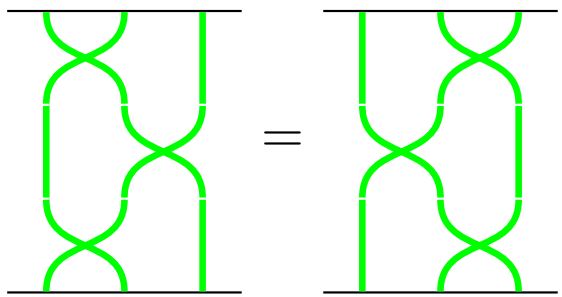
Idempotence:
$\tau_i^2 = \tau_i$ for all i



Far commutativity:
$\tau_i \tau_j = \tau_j \tau_i \quad 1 < j - i < n - 1$



Braid relations:
$\tau_i \tau_j \tau_i = \tau_j \tau_i \tau_j \quad j - i \in \{1, n-1\}$

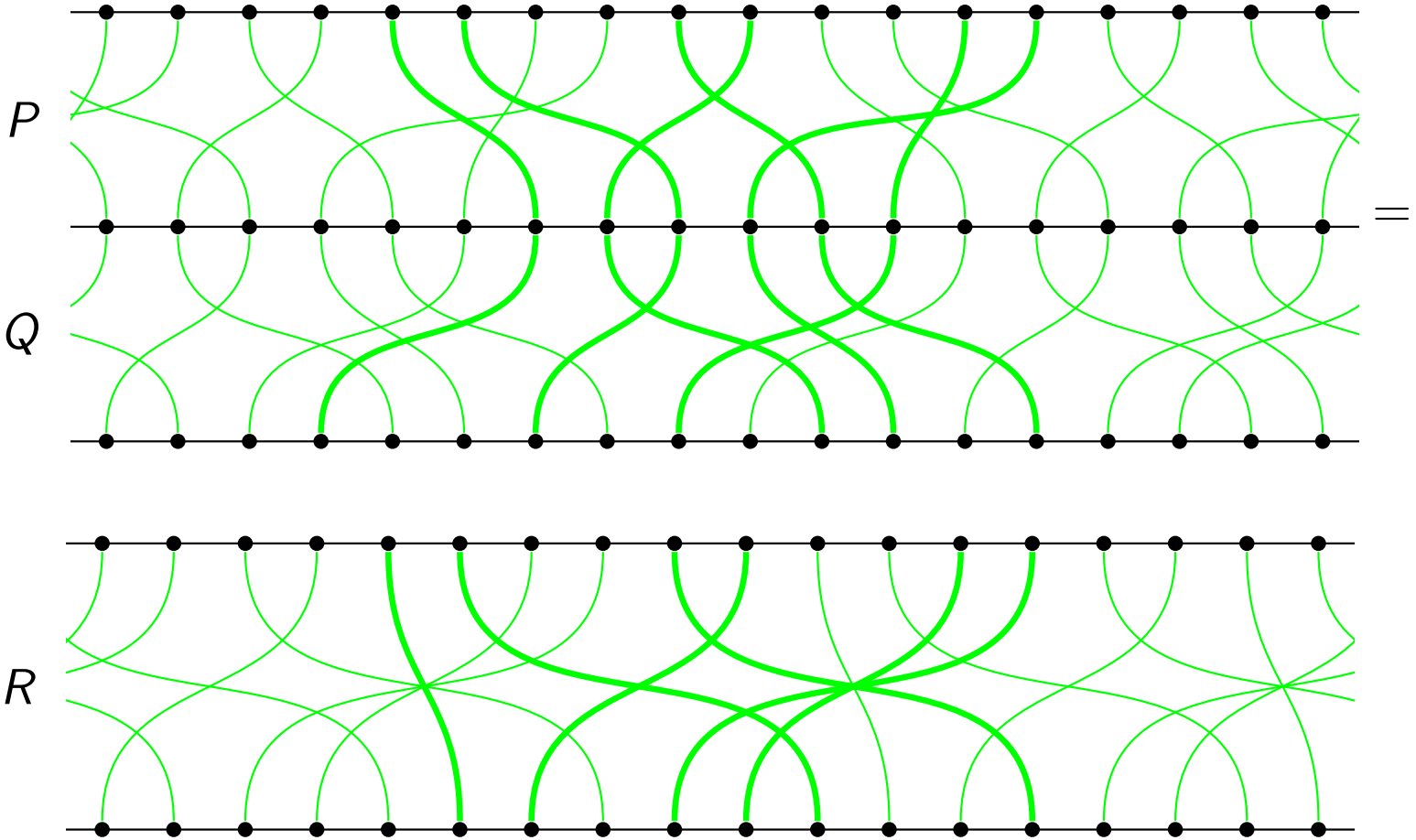


$|\tilde{\mathbb{H}}_n| = \infty$ canonical bijection $\tilde{\mathbb{H}}_n \leftrightarrow \tilde{\mathbb{S}}_n$

Cyclic and periodic string comparison

Affine sticky braids

$P \boxdot Q = R$: affine sticky braids



Multiplication in $\tilde{\mathbb{H}}_n$

Given affine permutation matrices P, Q, obtain $P \boxdot Q = R$

Cyclic and periodic string comparison

Affine sticky braids

Multiplication in $\tilde{\mathbb{H}}_n$

Given affine permutation matrices P, Q, obtain $P \boxdot Q = R$

Multiplication in $\tilde{\mathbb{H}}_n$: running time

$O(n \log n)$ [Gaevoy, Zolotov, T: NEW]

Multiplication in $\tilde{\mathbb{H}}_n$: 3-period algorithm

Extract three (!) consecutive j-periods from P, Q

Perform multiplication in $\mathbb{H}_{3n}$

Replicate middle j-period of the result to obtain R

Cyclic and periodic string comparison

Affine LCS

a, $\bar{b}$: strings of length m, $\bar{n}$ $\quad$ $b = \bar{b}^\infty = \ldots\bar{b}\bar{b}\bar{b}\ldots$: ∞-repeat

Affine SLCS problem

Given a, $\bar{b}$, obtain LCS scores for a vs every substring of $b = \bar{b}^\infty$

Output scores can be represented implicitly

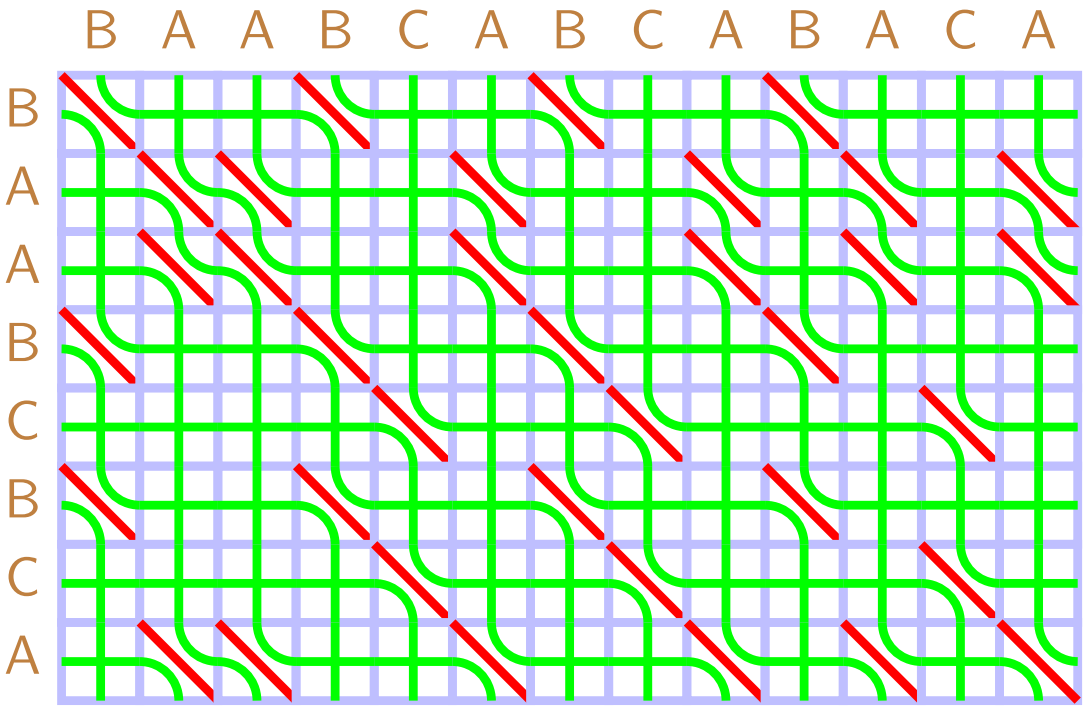
May assume that every character of a occurs in $\bar{b}$ (otherwise delete it)

Only need substrings in b of length $\leq m\bar{n}$ (otherwise LCS score $= m$)

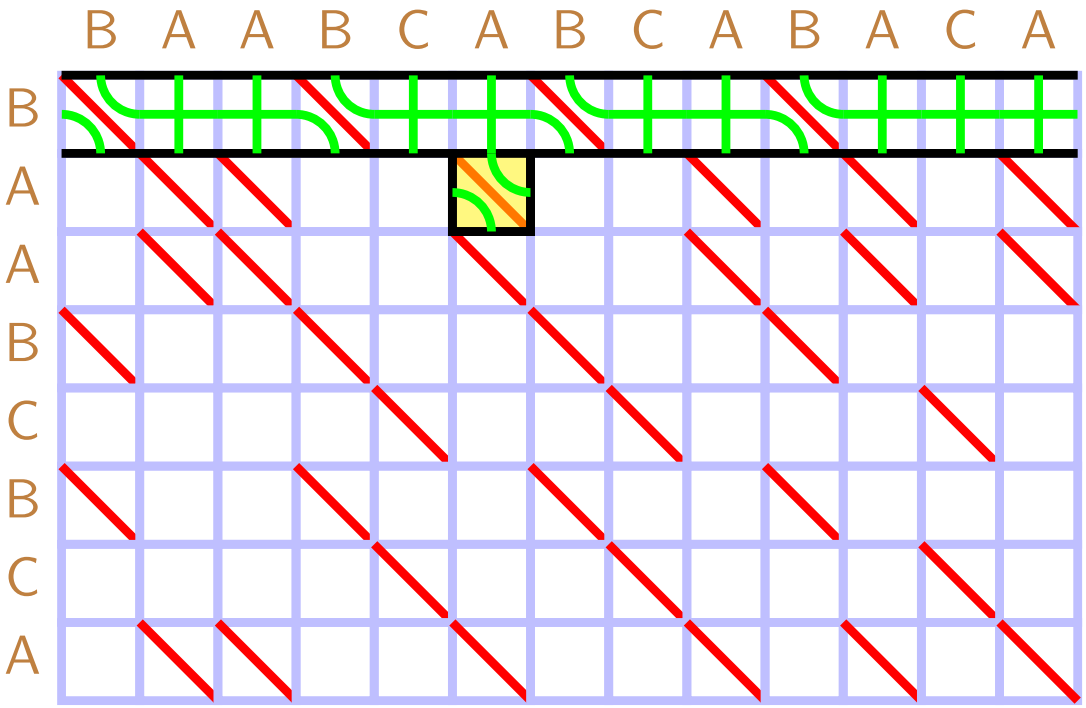
Affine SLCS: running time

$O(mn\bar{n})$	naive
$O(m\bar{n})$	[T: 2009]

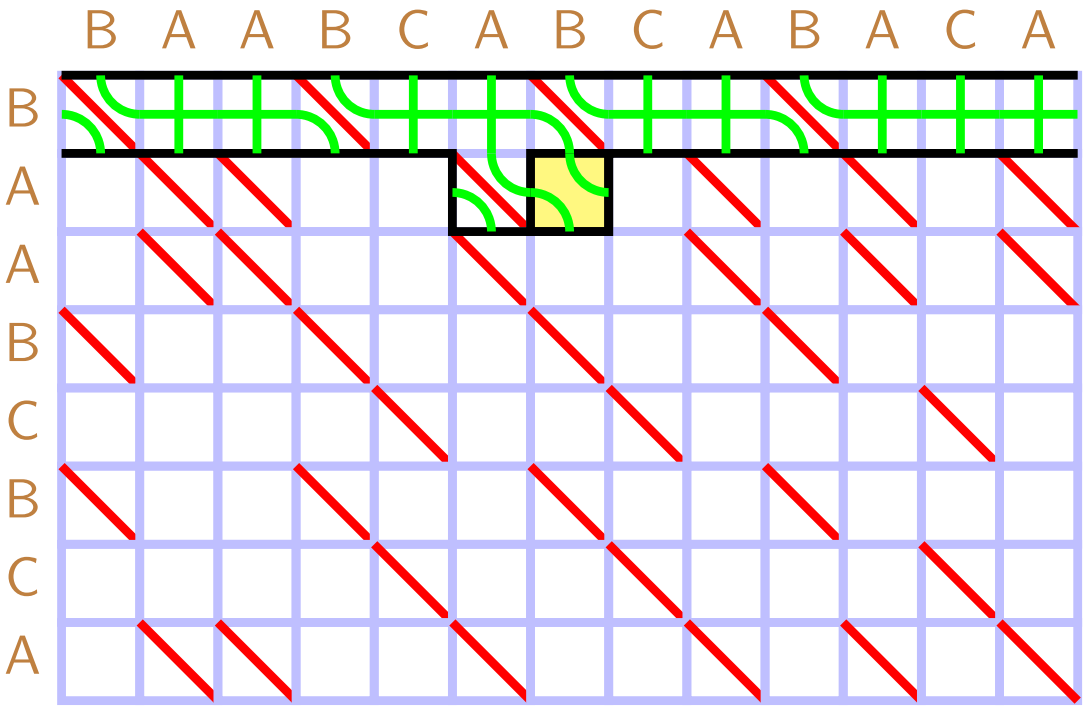
Affine SLCS by wraparound combing



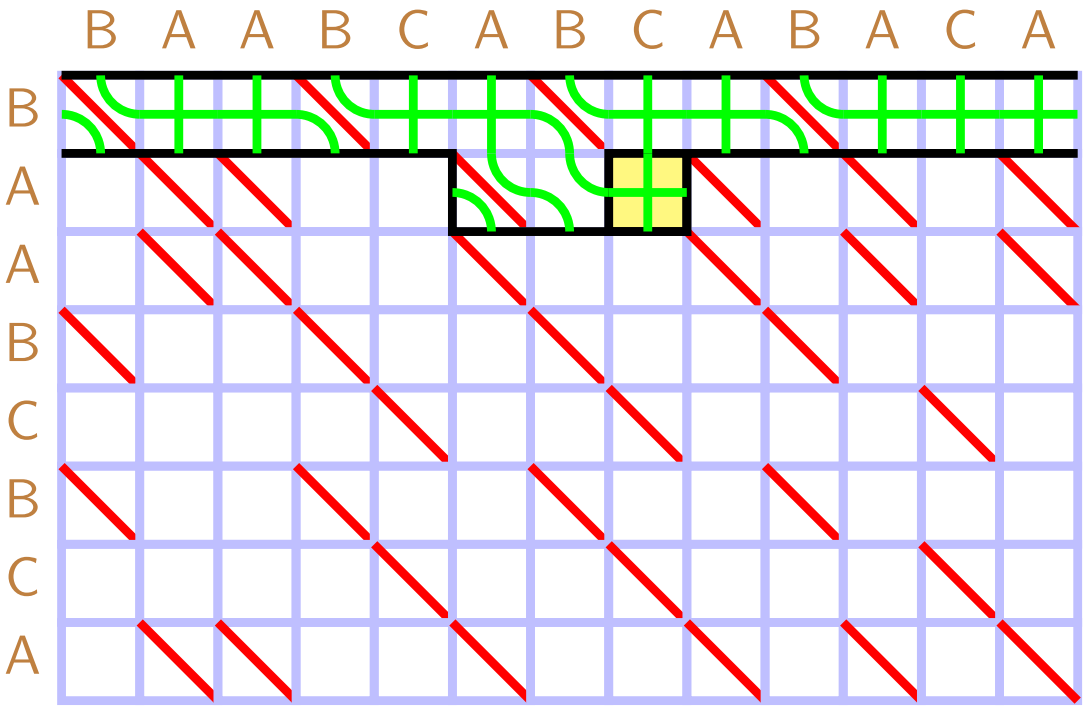
Affine SLCS by wraparound combing



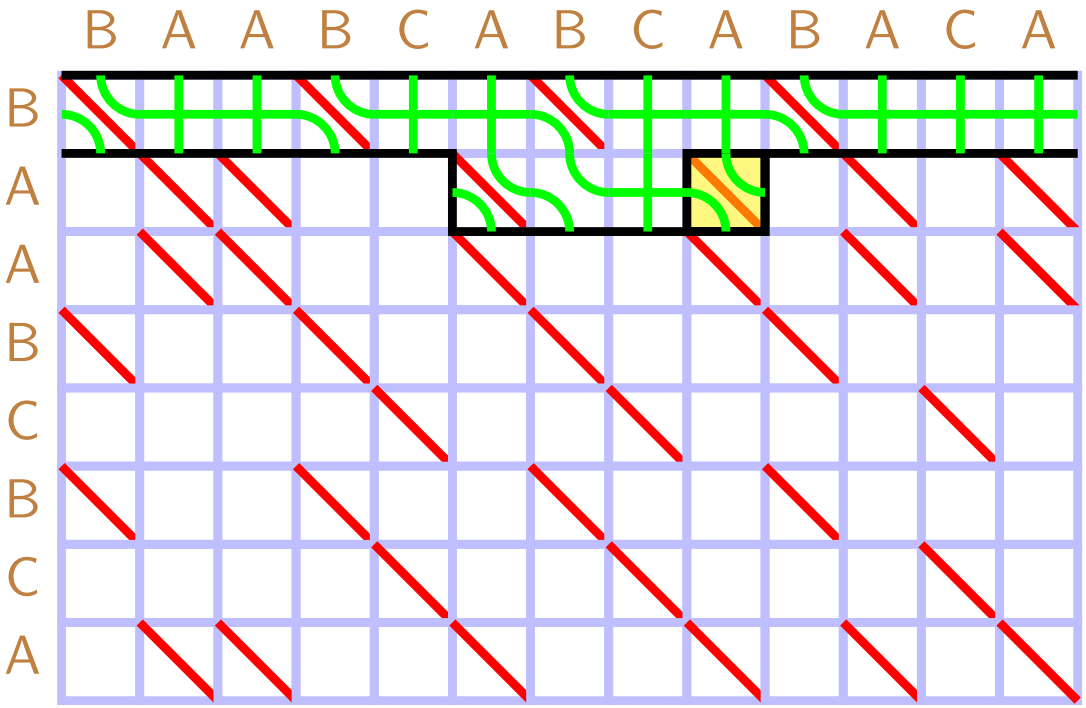
Affine SLCS by wraparound combing



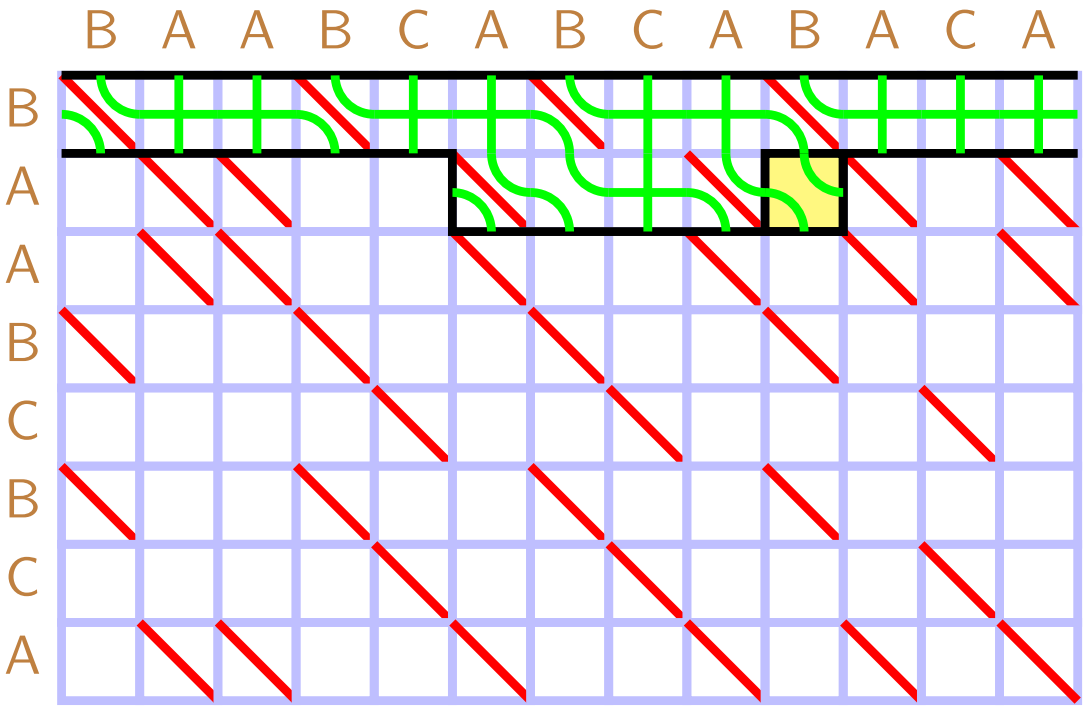
Affine SLCS by wraparound combing



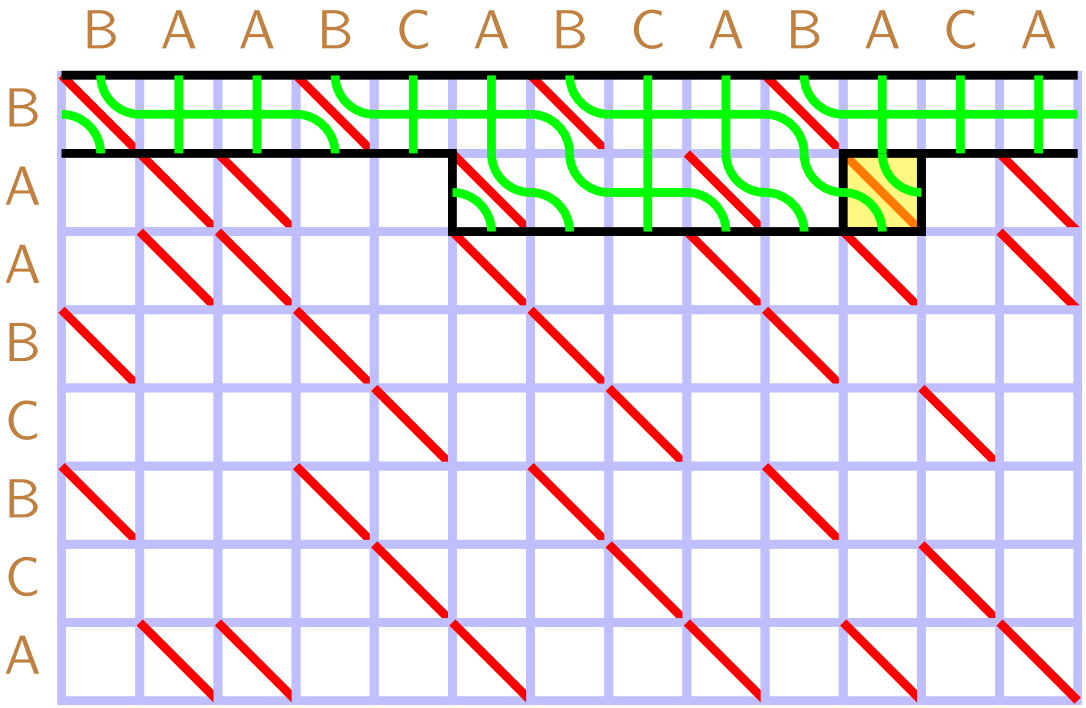
Affine SLCS by wraparound combing



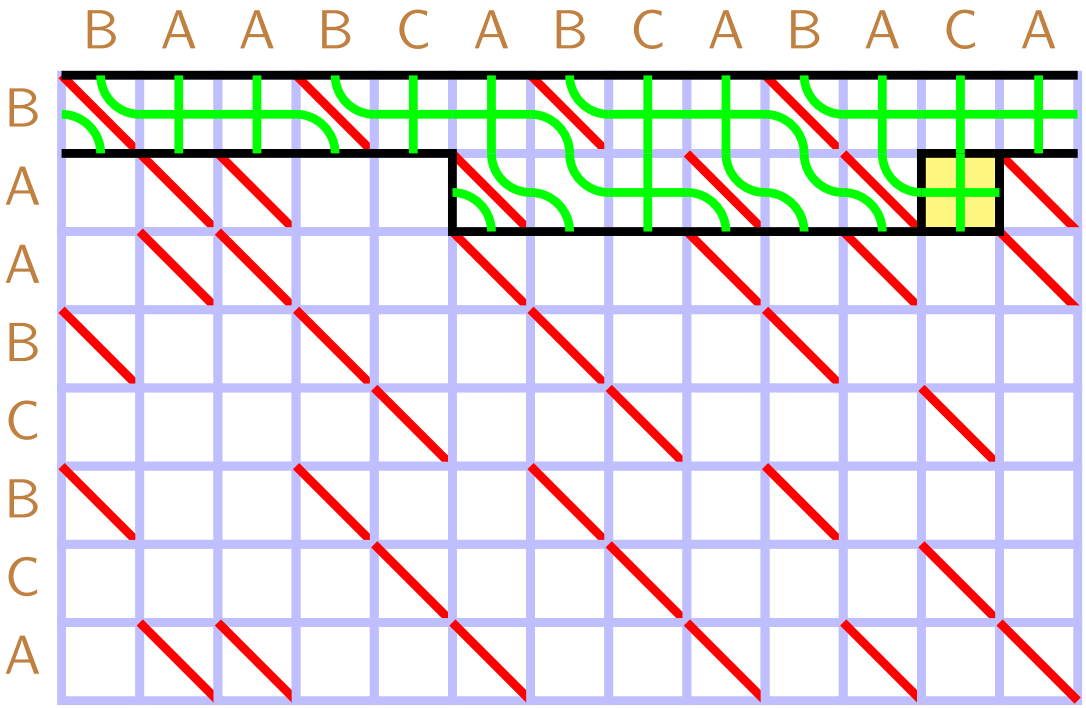
Affine SLCS by wraparound combing



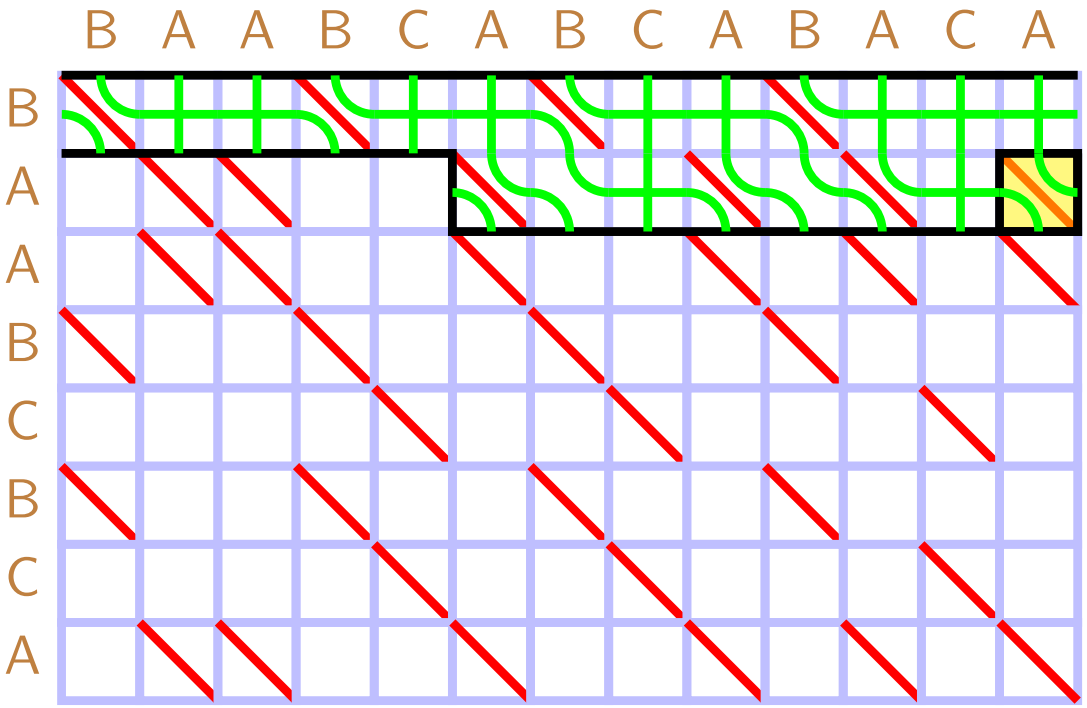
Affine SLCS by wraparound combing



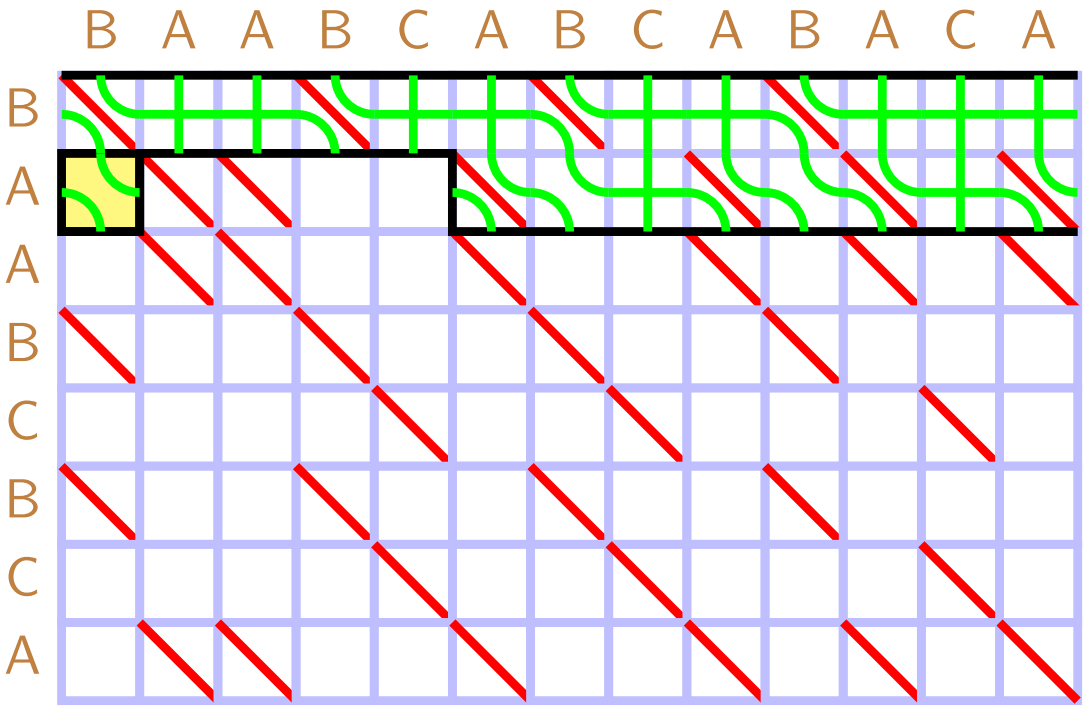
Affine SLCS by wraparound combing



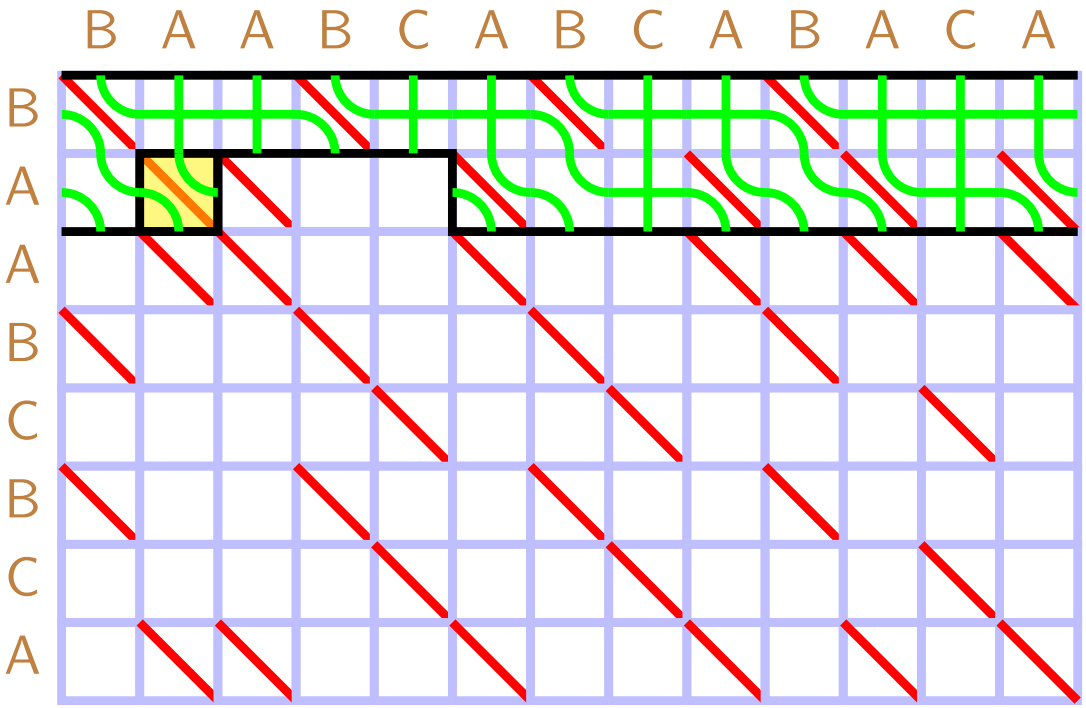
Affine SLCS by wraparound combing



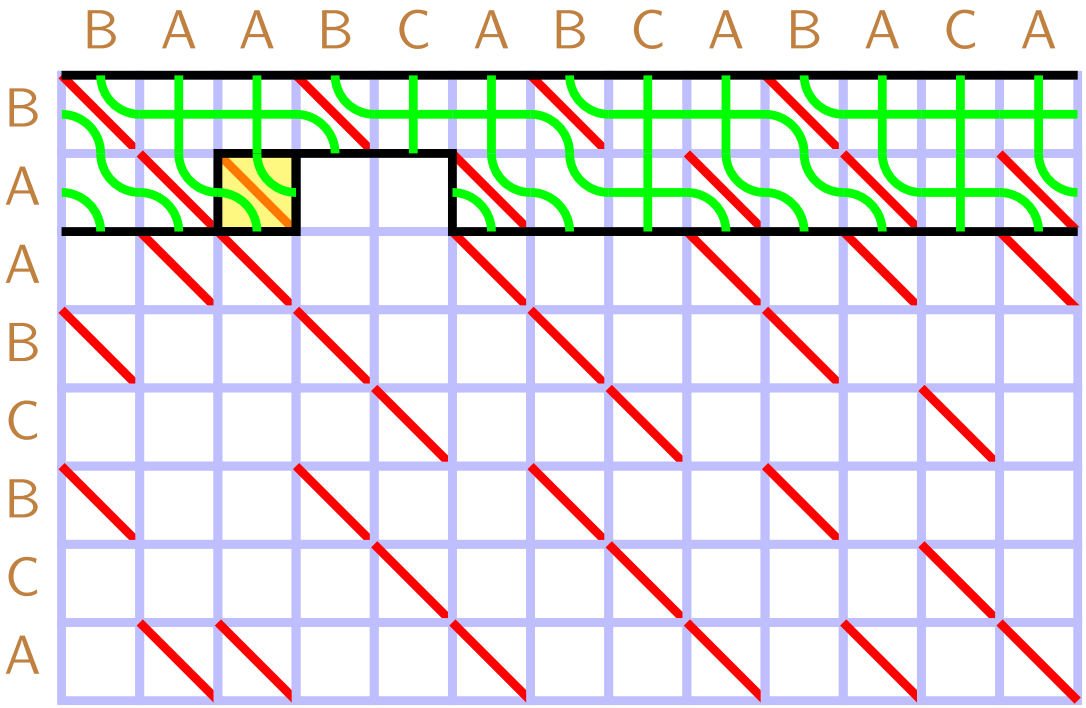
Affine SLCS by wraparound combing



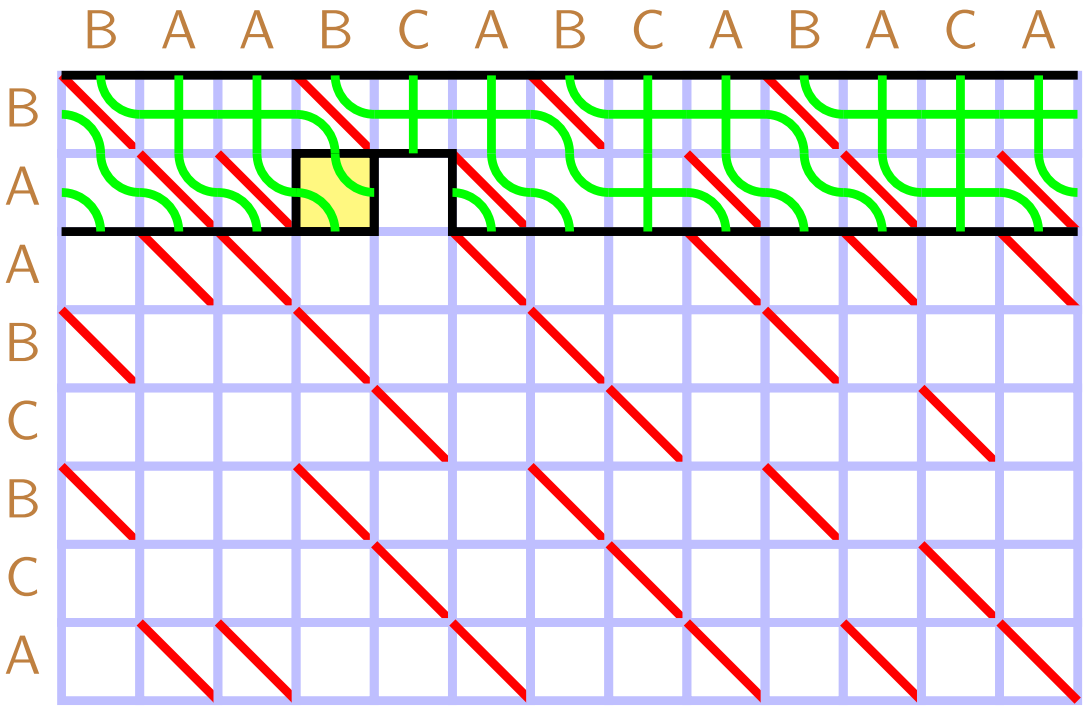
Affine SLCS by wraparound combing



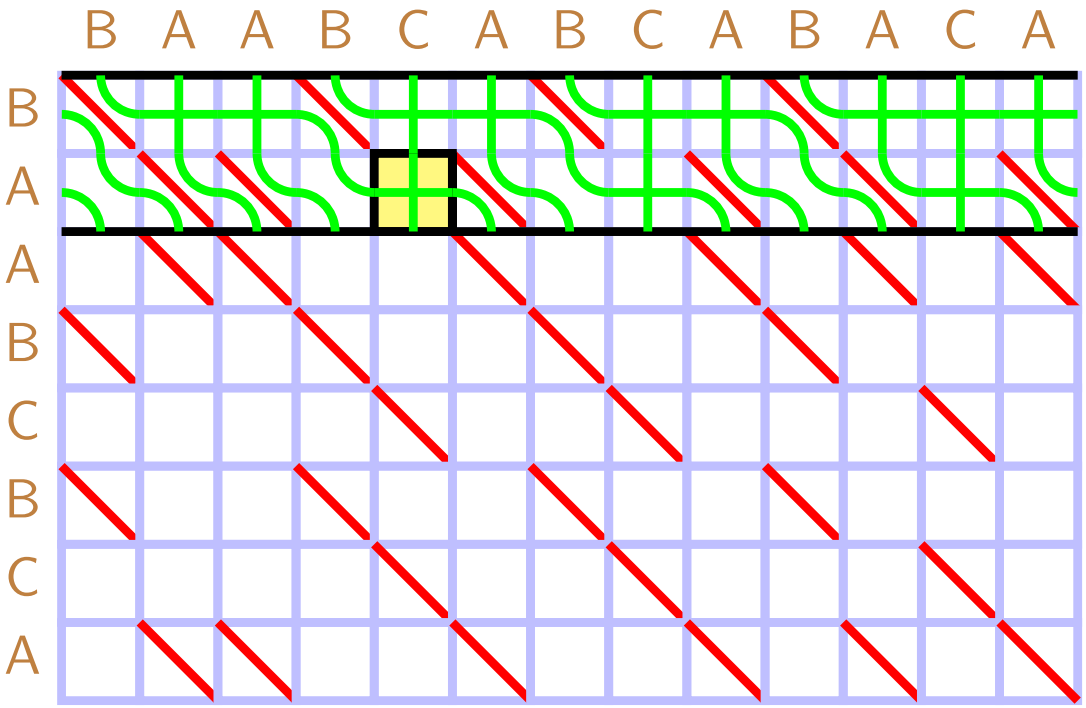
Affine SLCS by wraparound combing



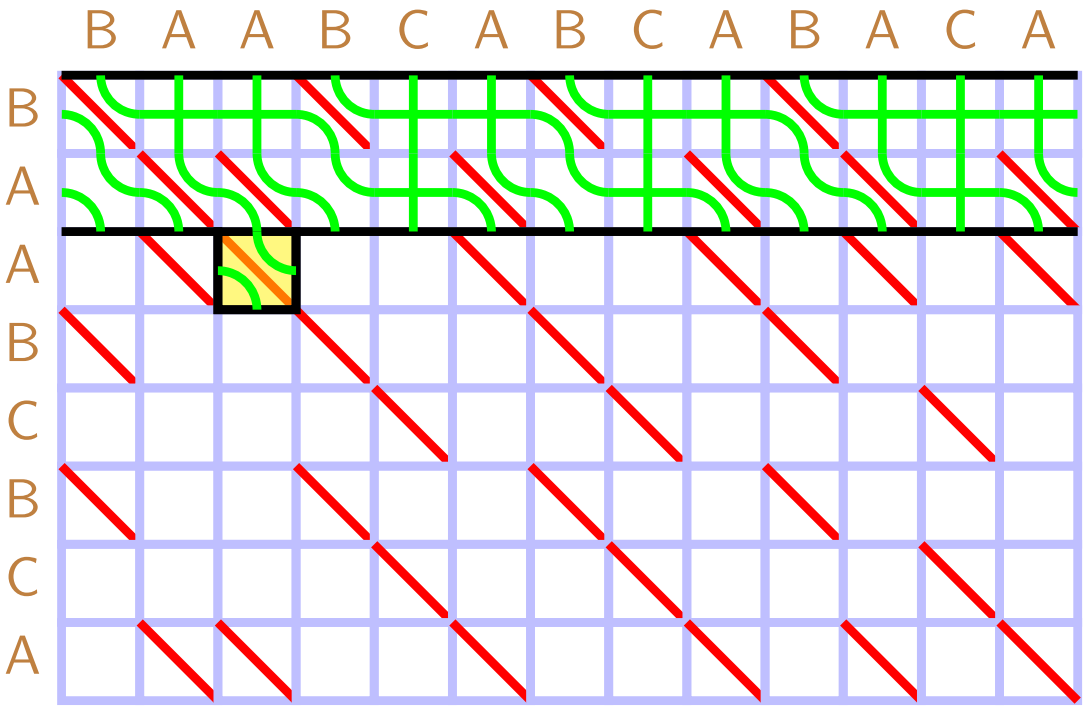
Affine SLCS by wraparound combing



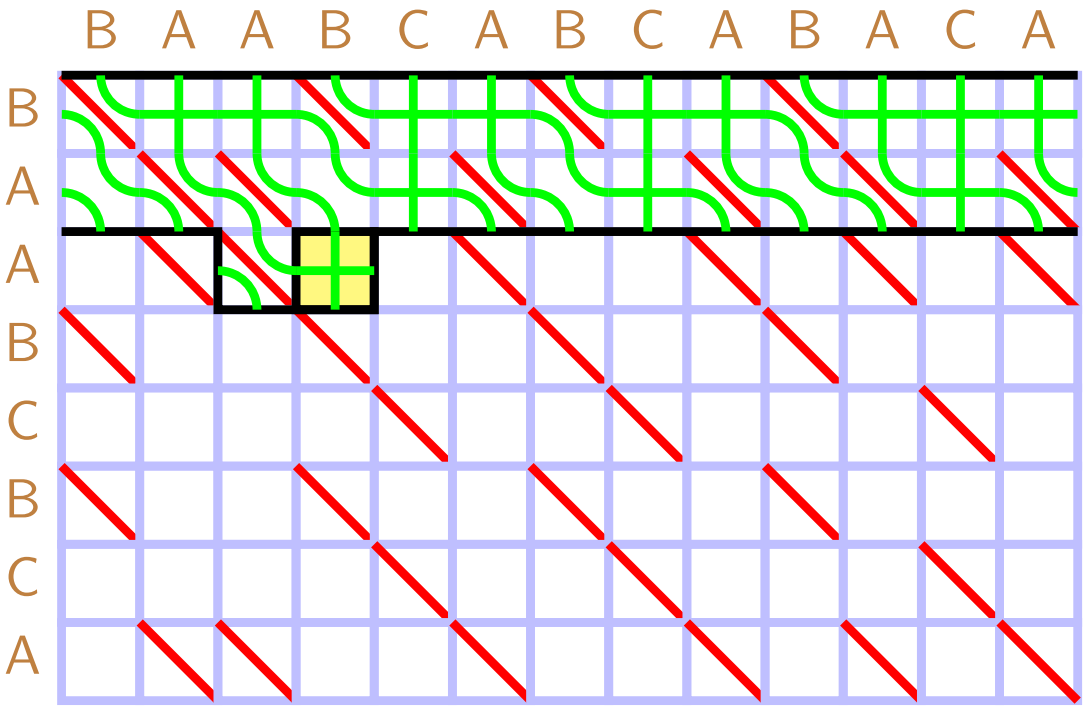
Affine SLCS by wraparound combing



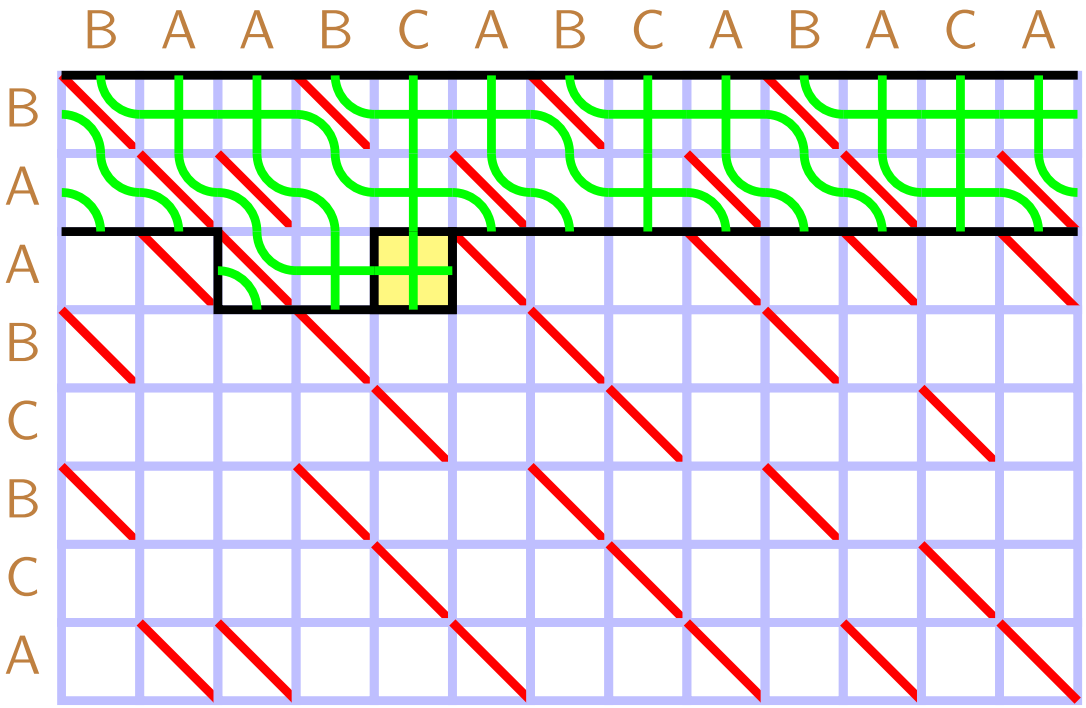
Affine SLCS by wraparound combing



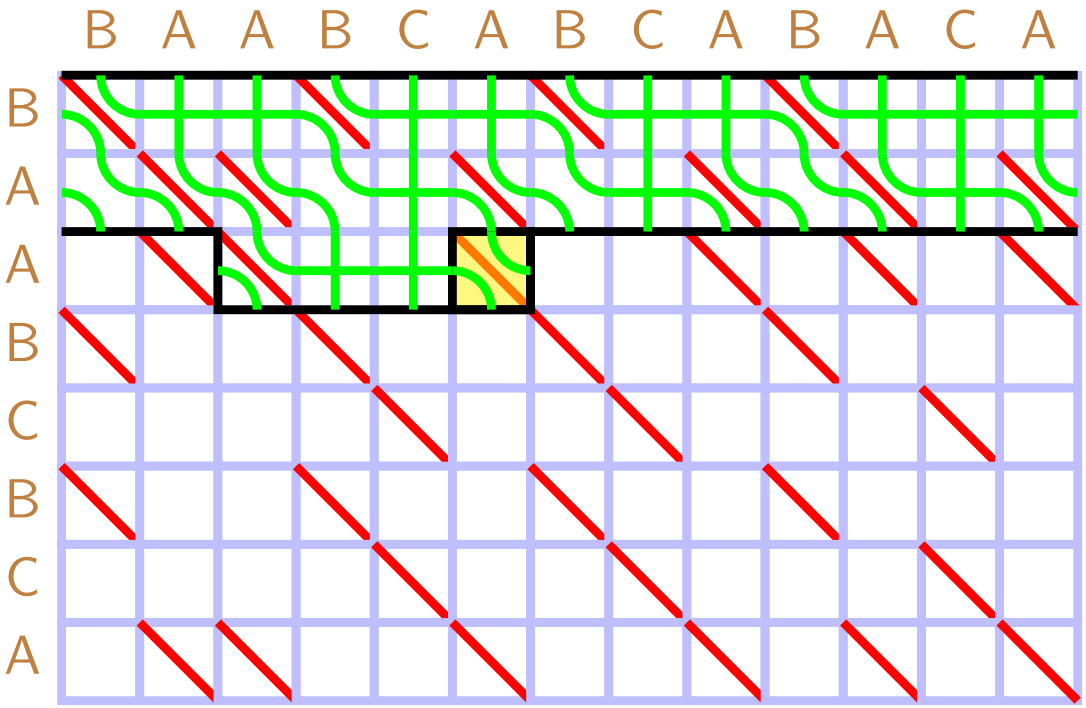
Affine SLCS by wraparound combing



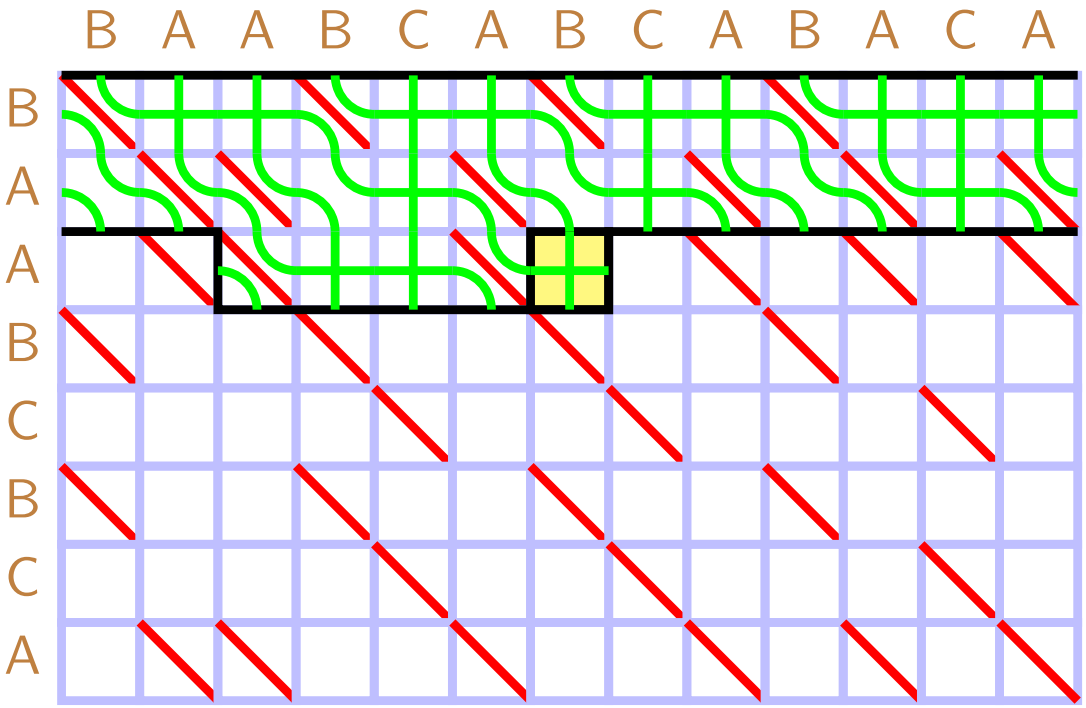
Affine SLCS by wraparound combing



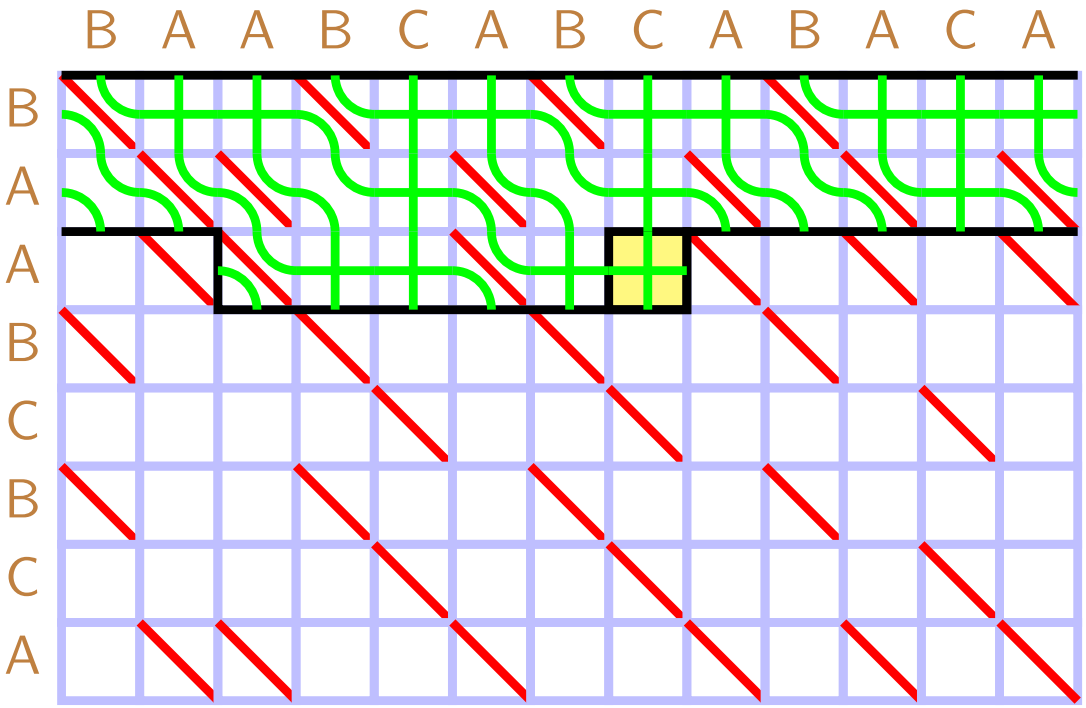
Affine SLCS by wraparound combing



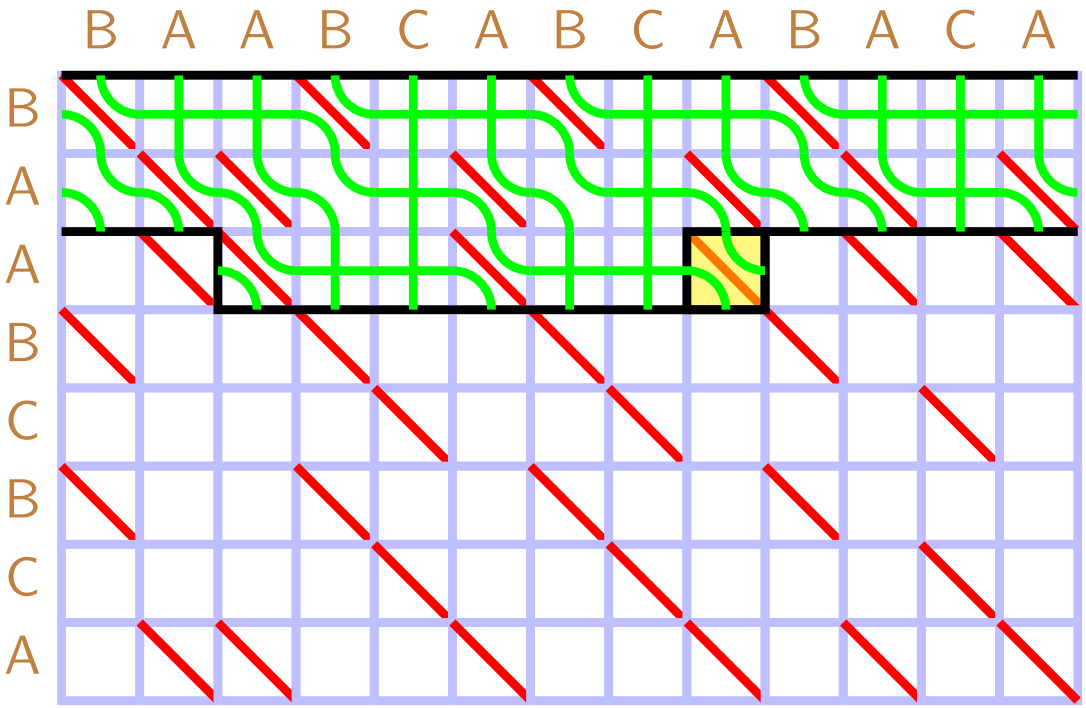
Affine SLCS by wraparound combing



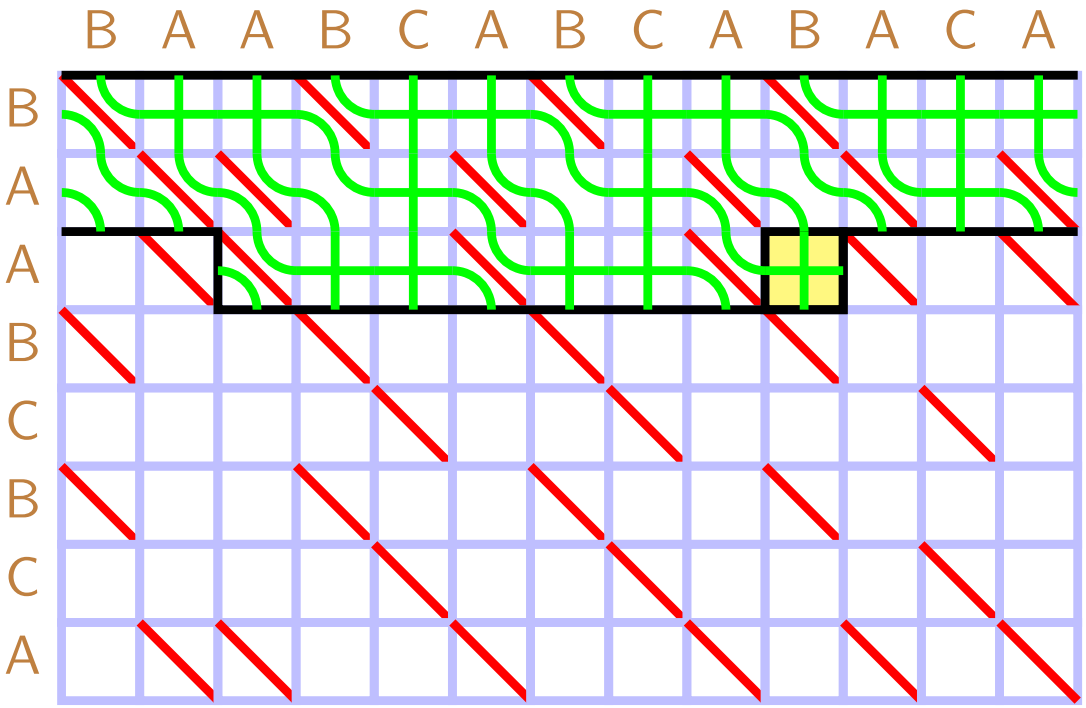
Affine SLCS by wraparound combing



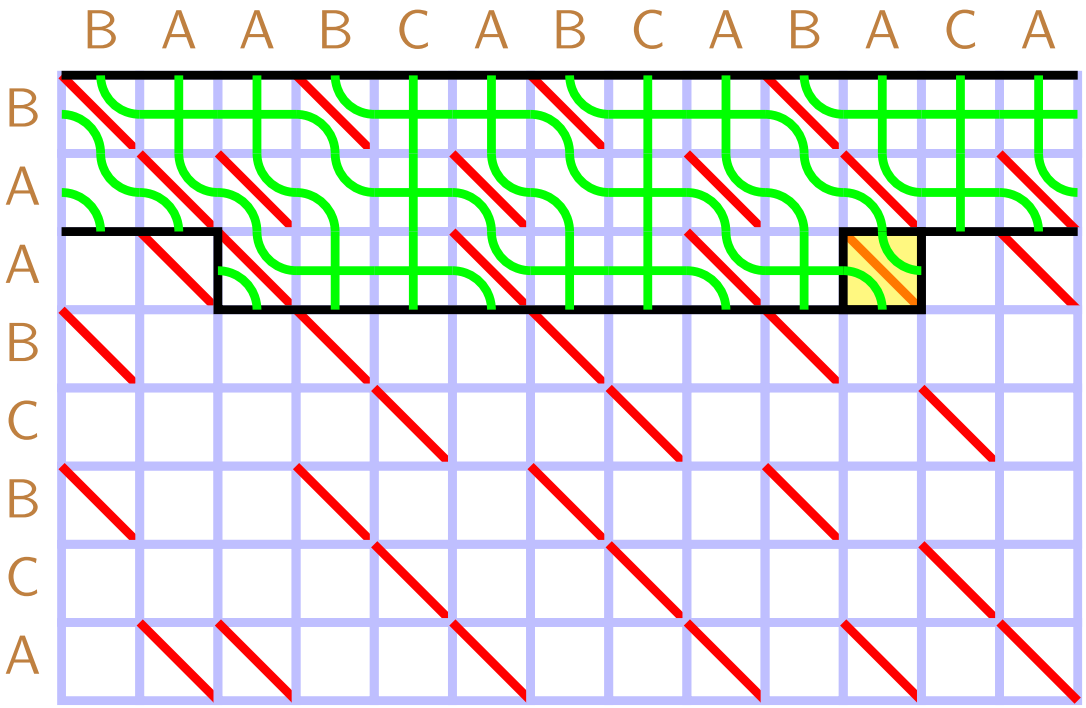
Affine SLCS by wraparound combing



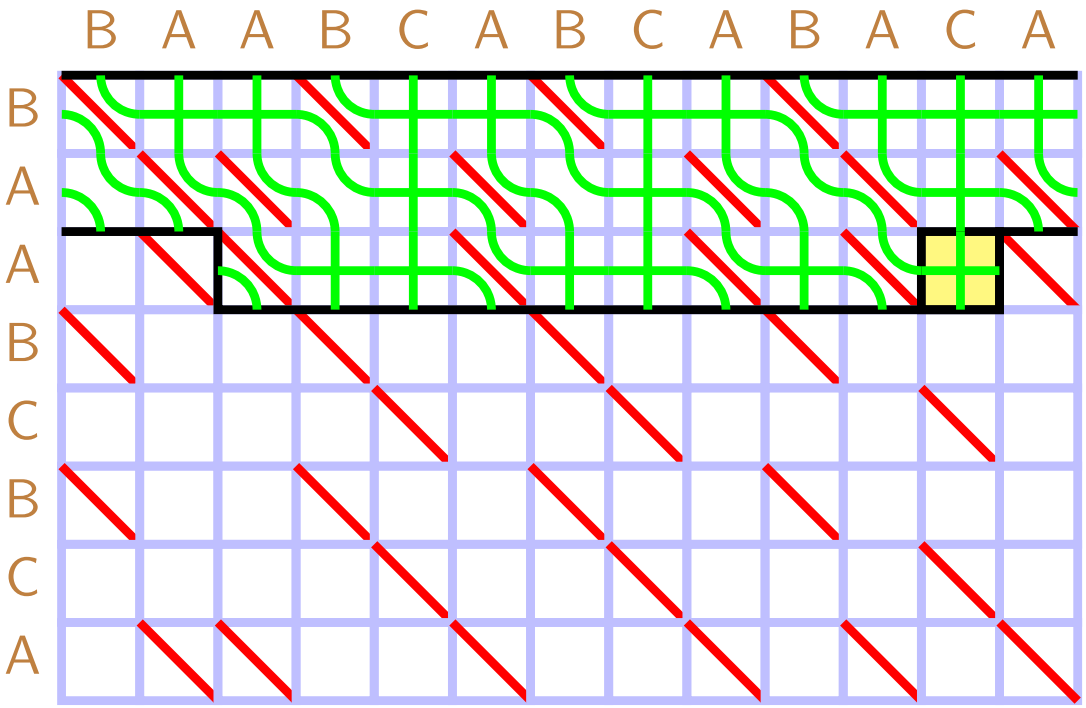
Affine SLCS by wraparound combing



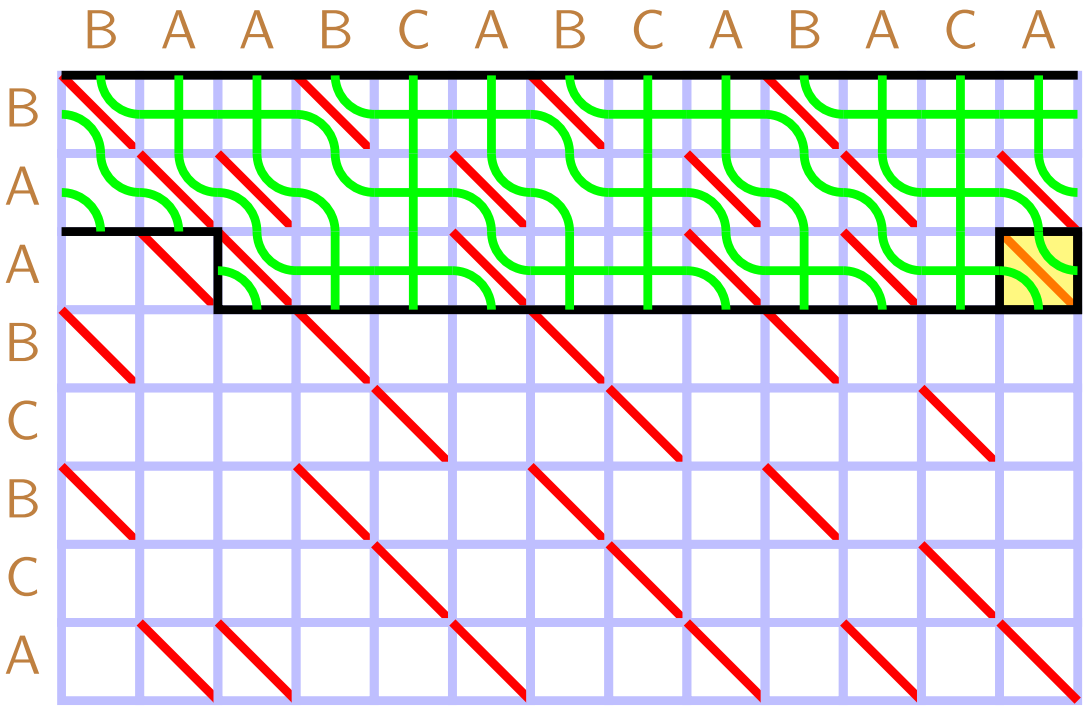
Affine SLCS by wraparound combing



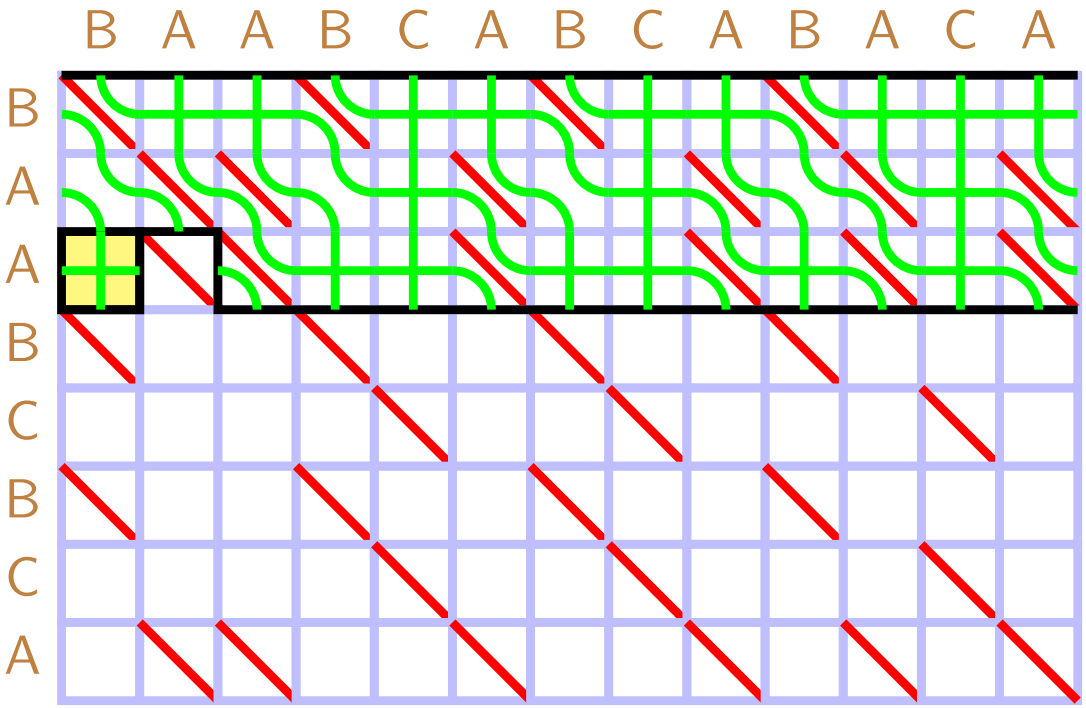
Affine SLCS by wraparound combing



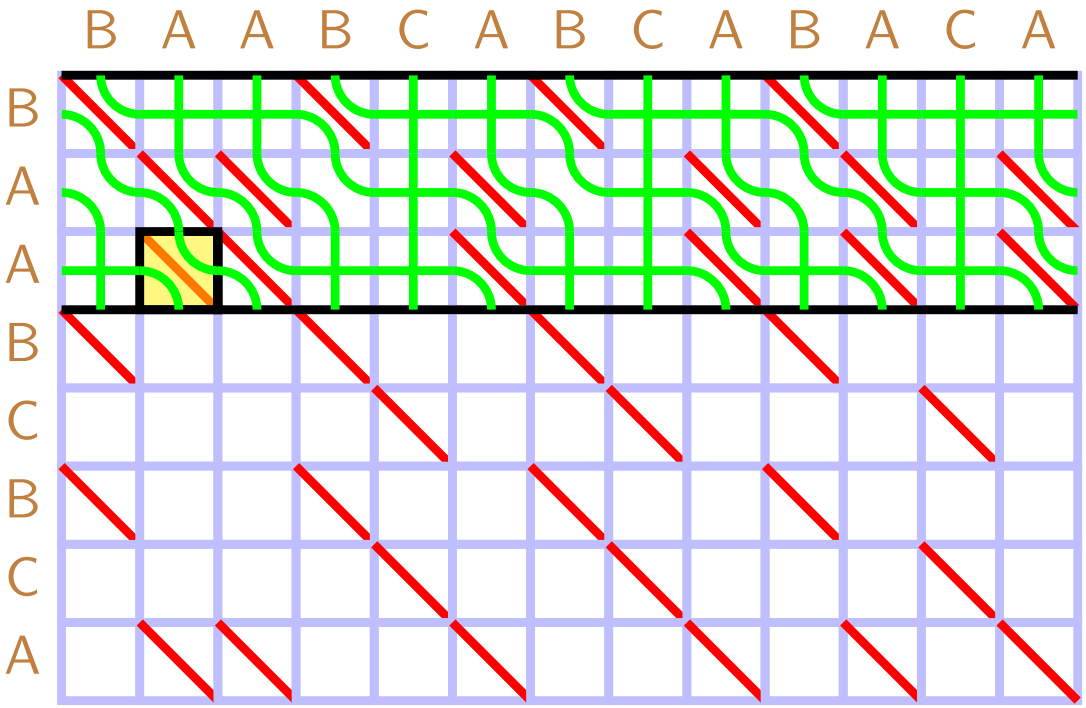
Affine SLCS by wraparound combing



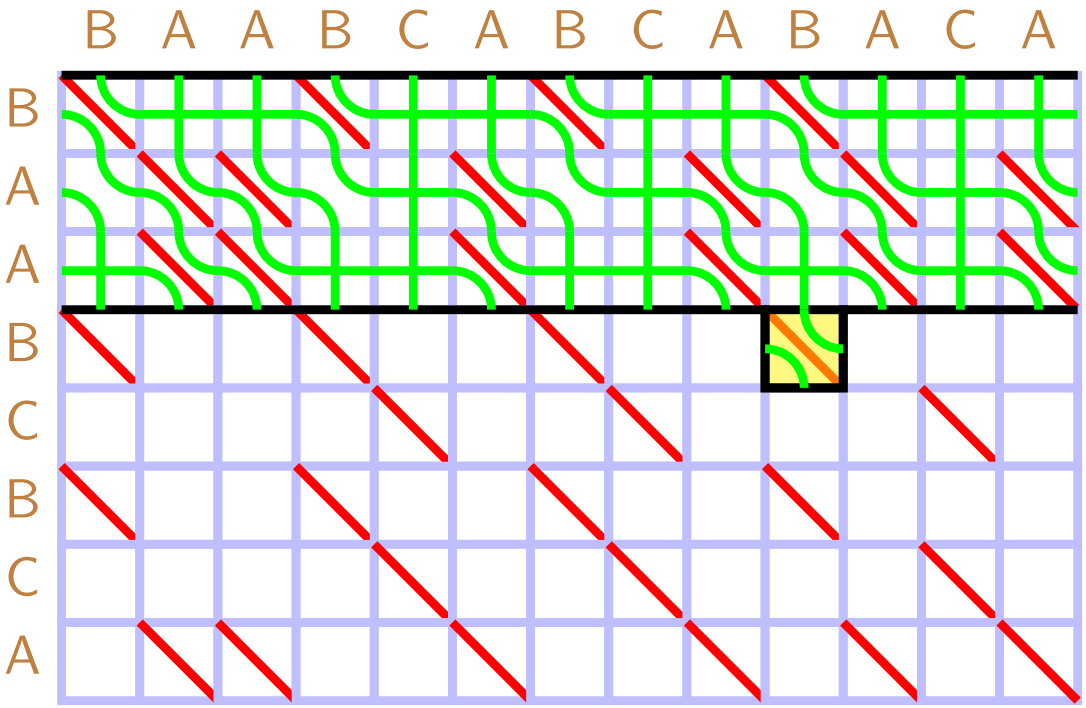
Affine SLCS by wraparound combing



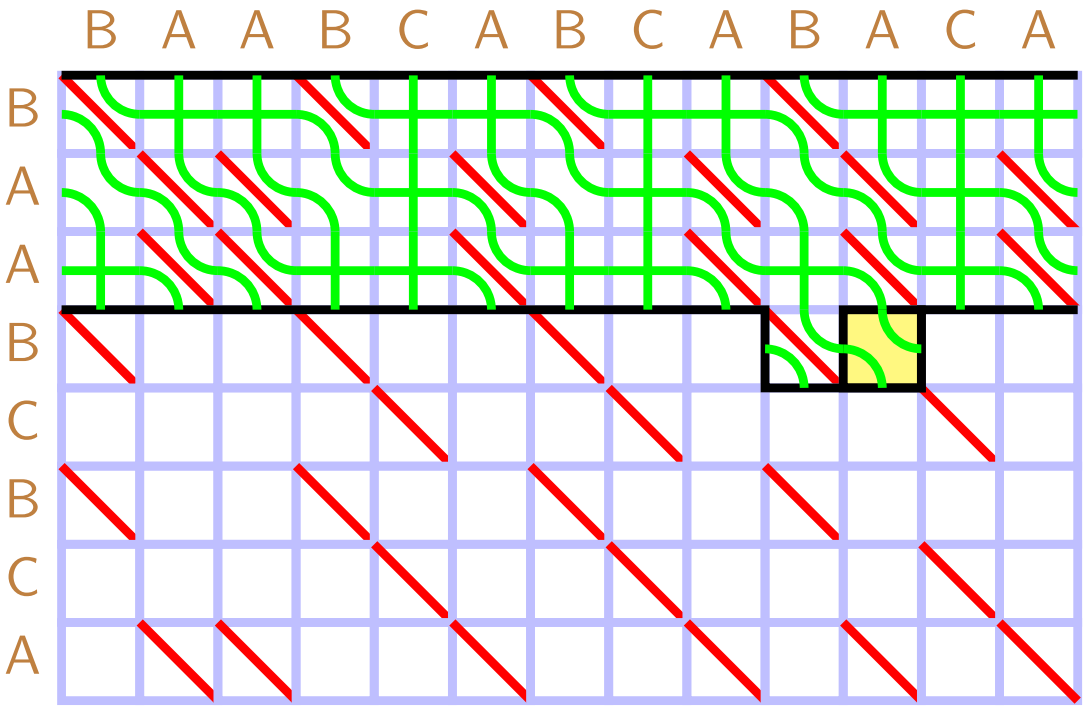
Affine SLCS by wraparound combing



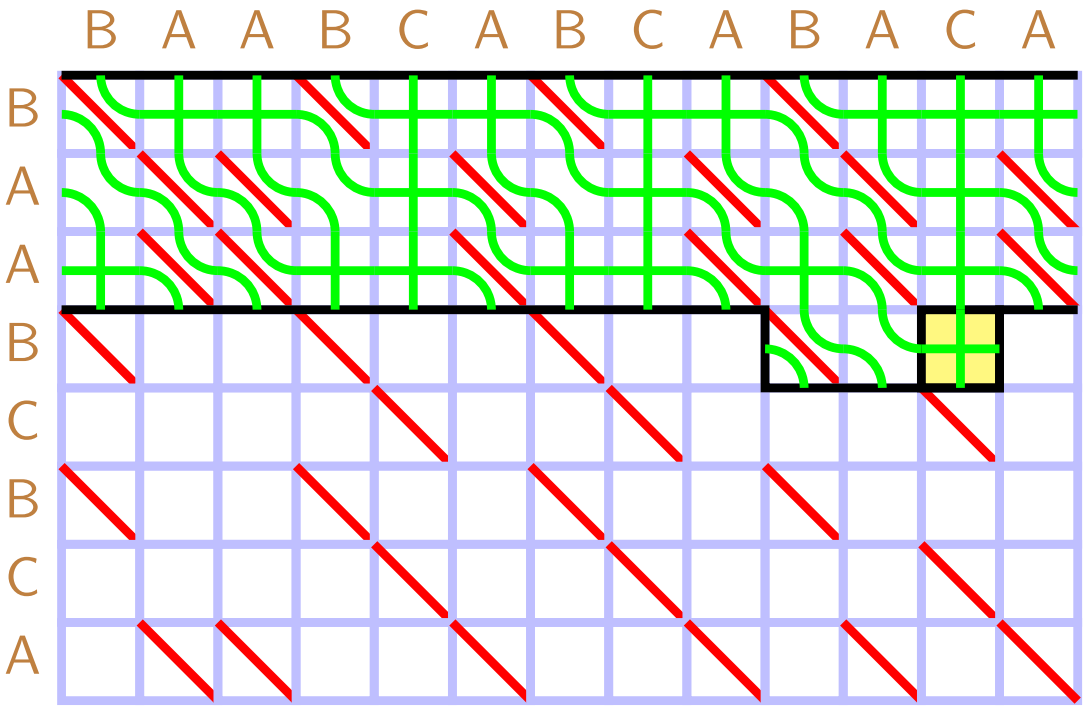
Affine SLCS by wraparound combing



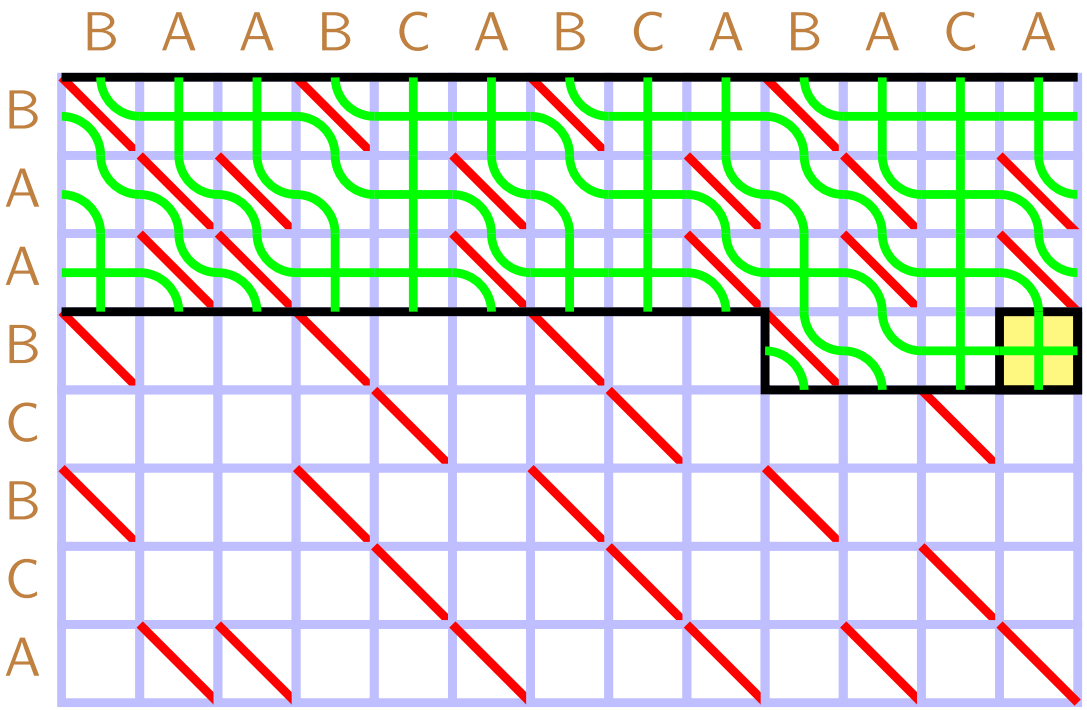
Affine SLCS by wraparound combing



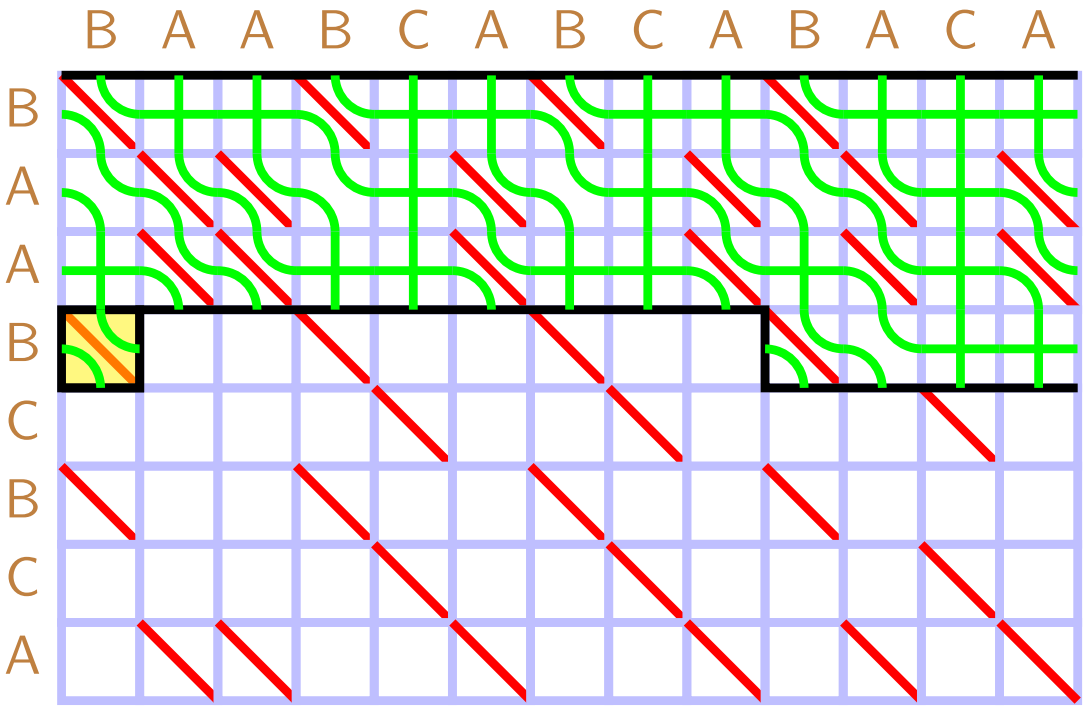
Affine SLCS by wraparound combing



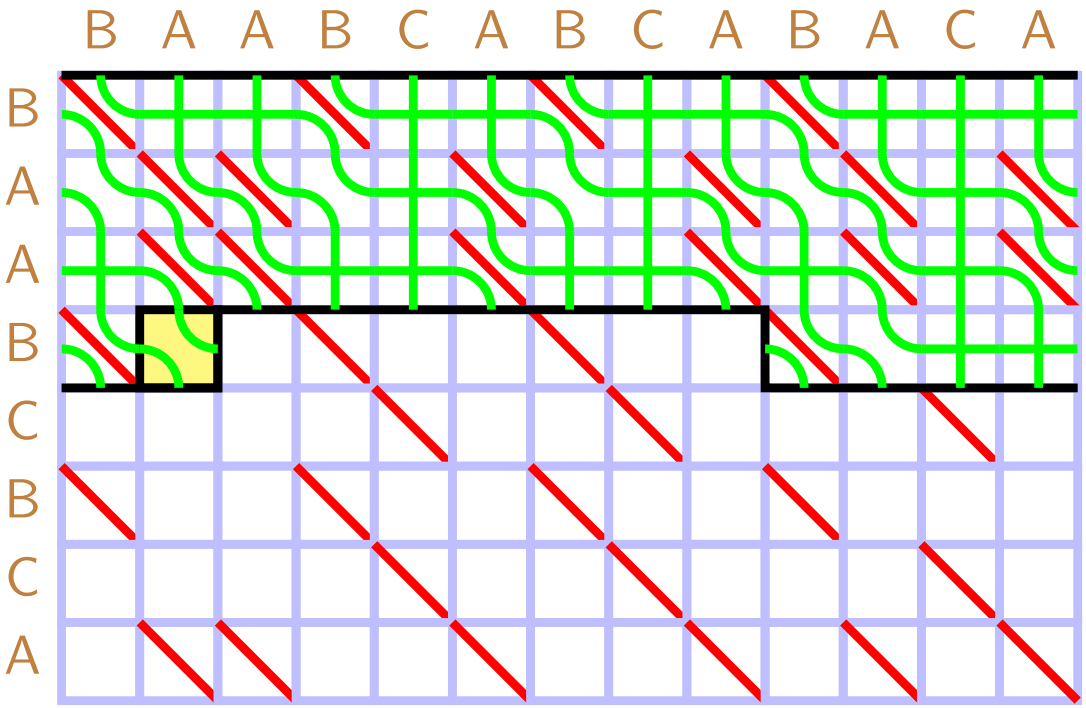
Affine SLCS by wraparound combing



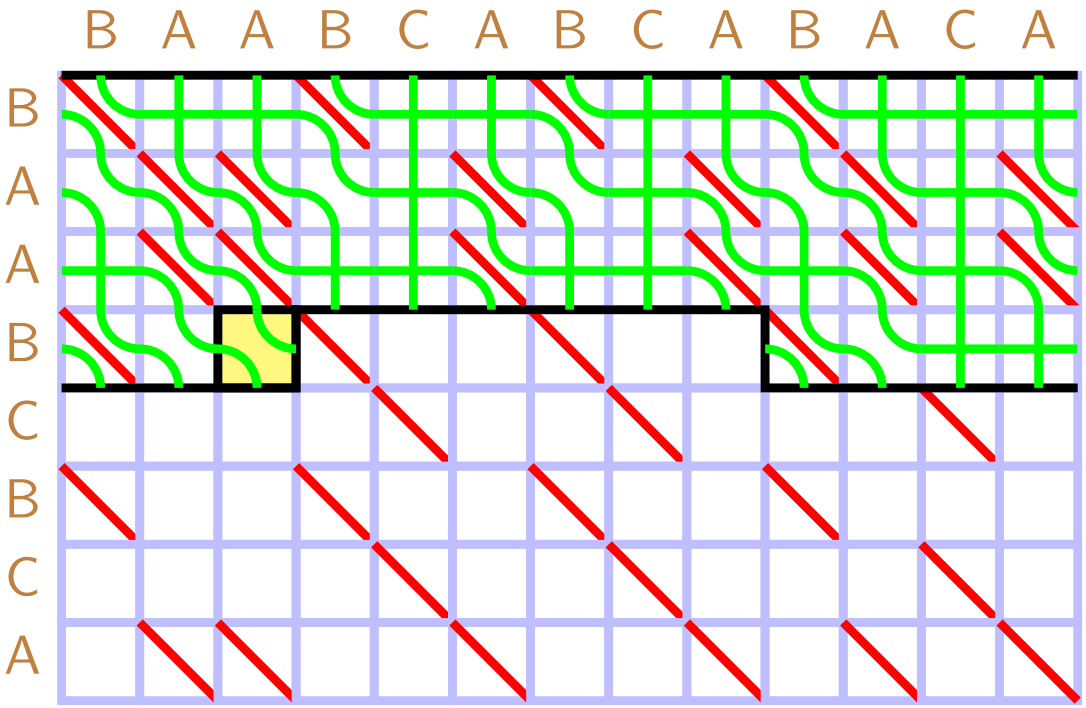
Affine SLCS by wraparound combing



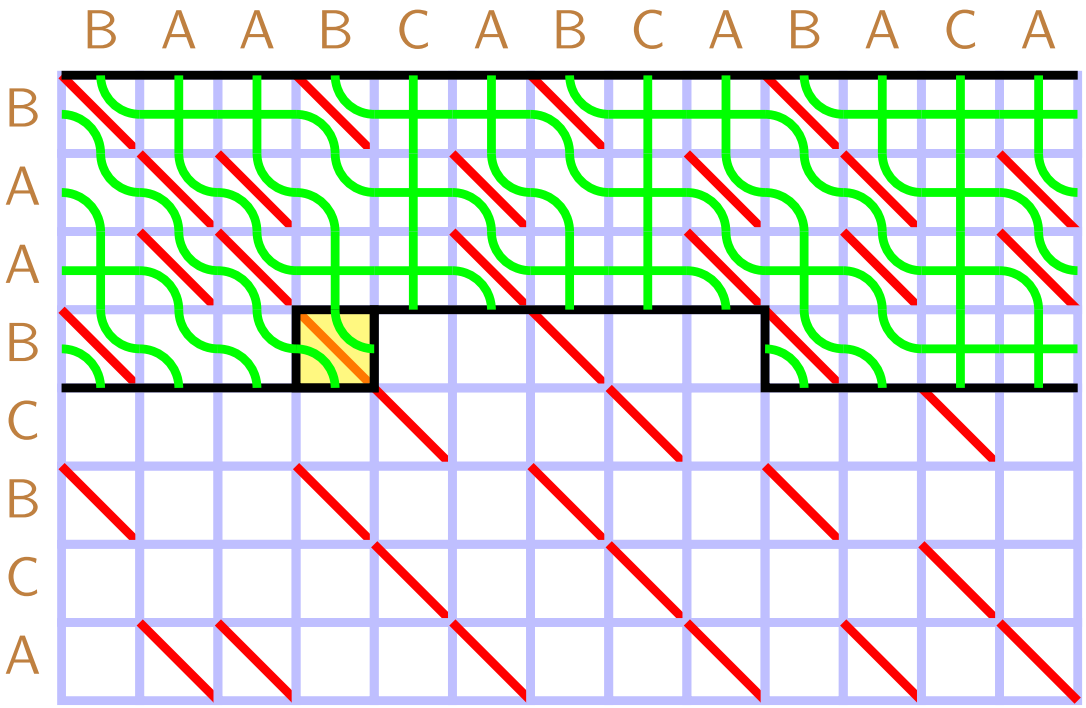
Affine SLCS by wraparound combing



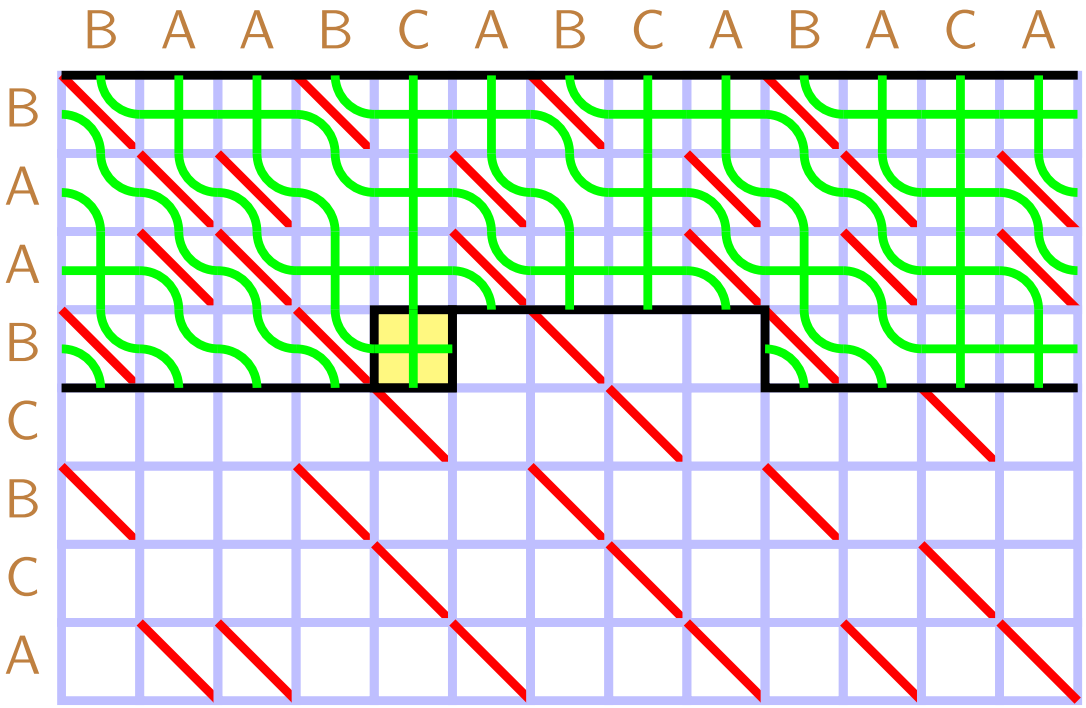
Affine SLCS by wraparound combing



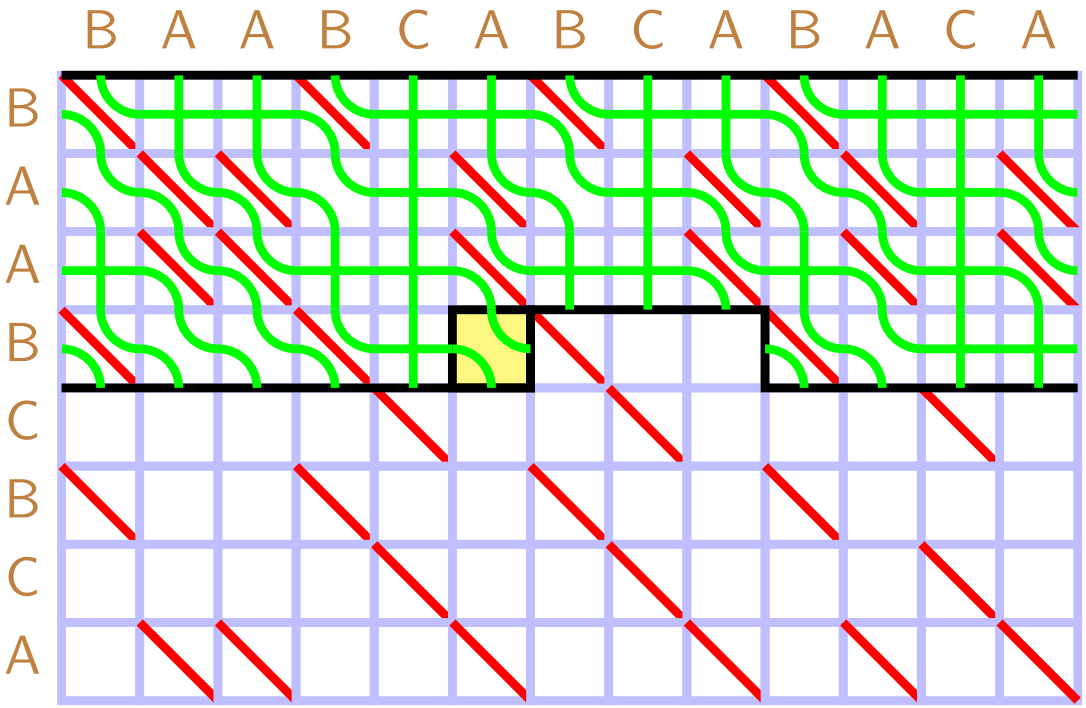
Affine SLCS by wraparound combing



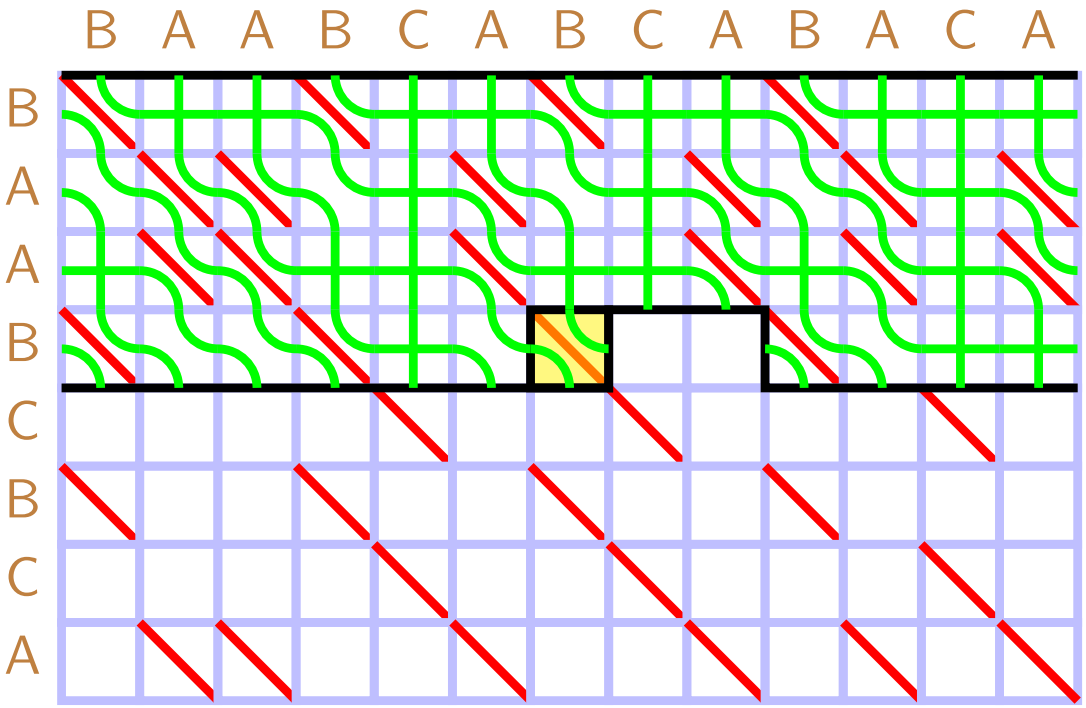
Affine SLCS by wraparound combing



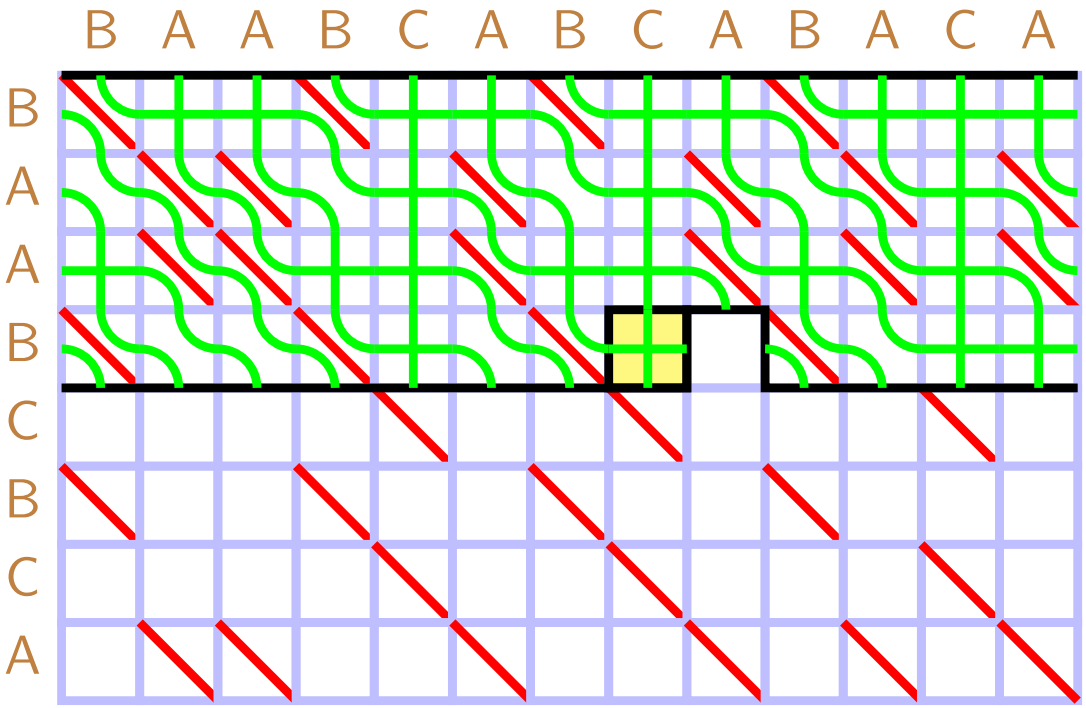
Affine SLCS by wraparound combing



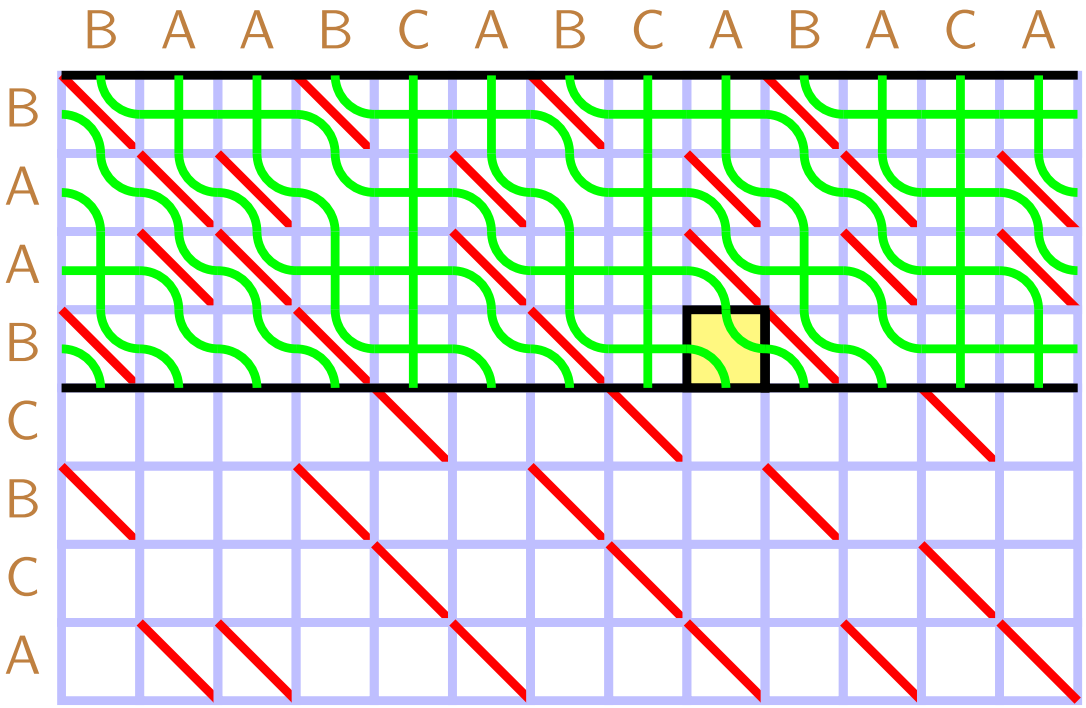
Affine SLCS by wraparound combing



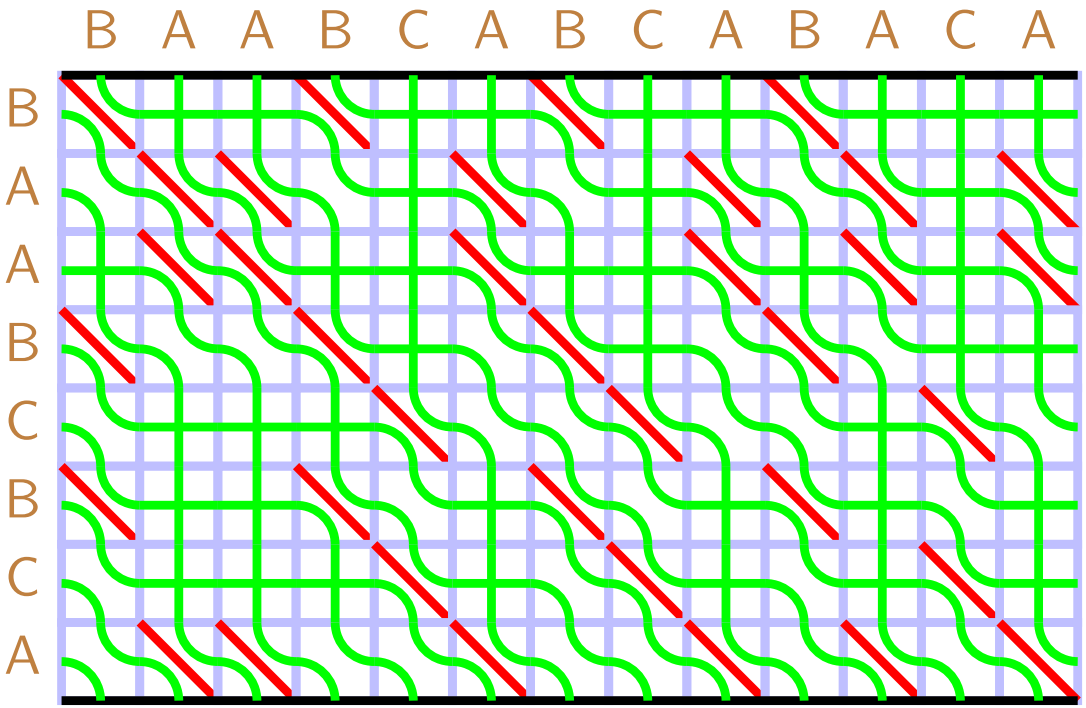
Affine SLCS by wraparound combing



Affine SLCS by wraparound combing



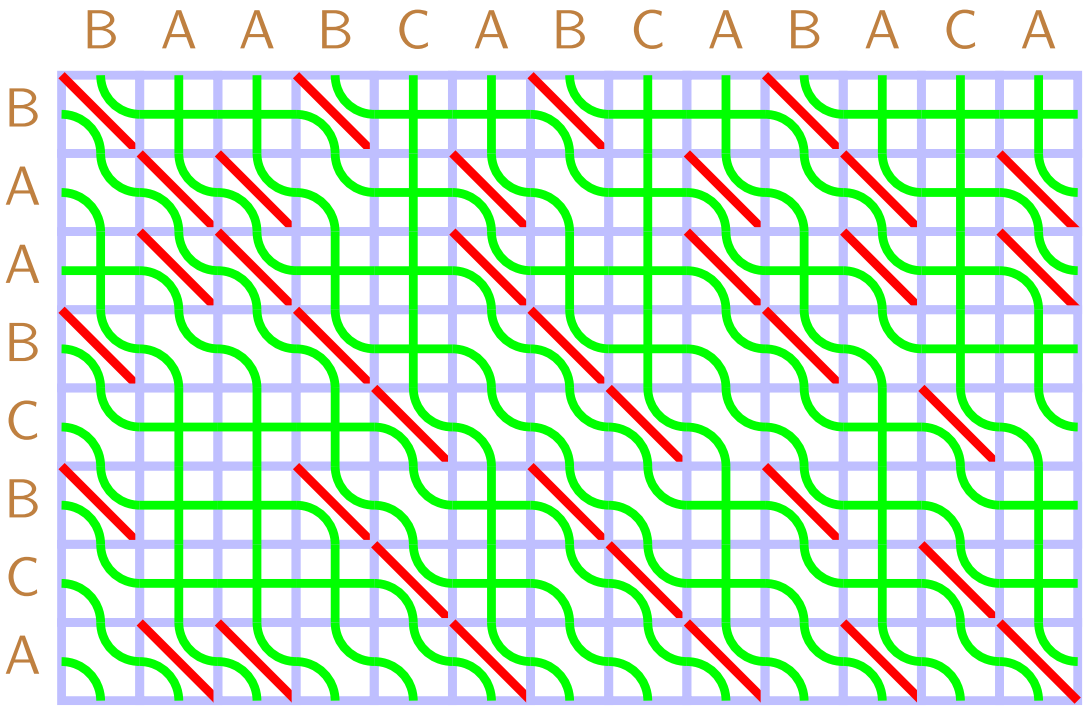
Affine SLCS by wraparound combing



Cyclic and periodic string comparison

Affine LCS

Affine SLCS by wraparound combing



Positively reduced affine braid: SLCS kernel for every prefix of a vs ∞-repeat b

Implicit prefix-substring LCS

Affine SLCS: Wraparound combing

Initialise uncombed sticky braid: mismatch cell = crossing

Iterate over cells in rows: row begins at match, wraps around at boundary

- match cell: skip (keep uncrossed)
- mismatch cell: comb (uncross if strands crossed before, possibly in a different period)

Active cell update: time $O(1)$

String-substring LCS score: count strands (with multiplicities, whenever substring covers multiple periods)

Overall time $O(m\bar{n})$

Cyclic and periodic string comparison

Periodic LCS

$b = \bar{b}^k = \bar{b}\bar{b}\ldots\bar{b}$: periodic string, a (weak) form of compression

Periodic LCS problem

Given a, $\bar{b}$, k, obtain LCS score for a vs $b = \bar{b}^k$

We have $n = k\bar{n}$; may assume $k \le m$

Periodic LCS: running time

$O(mk\bar{n})$	naive
$O\big(m(k+\bar{n})\big)$	[Landau, Ziv-Ukelson: 2001]
$O(m\bar{n})$	[T: 2009]

Run affine SLCS; query the score

Periodic approximate matching problem

Maximum alignment score for a vs a substring of $b = \bar{b}^\infty$

- global: only substrings $\bar{b}^k$ of length $k\bar{n}$ across all k
- cyclic: all substrings of length $k\bar{n}$ across all k
- local: all substrings of any length

Periodic approximate matching: running time

$O(m^2\bar{n})$	all	naive
$O(m\bar{n})$	global, local	[Myers, Miller: 1989]
$O(m\bar{n}\log\bar{n})$	cyclic	[Benson: 2005]
$O(m\bar{n})$	all	[T: 2009]

Cyclic and periodic string comparison

Periodic LCS

$\bar{a}$, $\bar{b}$: strings of length $\bar{m}$, $\bar{n}$

Doubly-periodic LCS problem

Given $\bar{a}$, k, $\bar{b}$, l, obtain LCS scores for $a = \bar{a}^k$ vs $b = \bar{b}^l$

Output scores can be represented implicitly

We have $m = k\bar{m}$, $n = l\bar{n}$; $k \le n$, $l \le m$

Doubly-periodic string-substring LCS: running time

$O(mn)$	naive
$O(m\bar{n})$	as single-periodic
$O\big((\bar{m} + \log k \log \bar{n}) \cdot \bar{n}\big)$	[Zolotov, Gaevoy, T: 2025]

Doubly-periodic LCS

Affine SLCS for $\bar{a}$ vs $\bar{b}^\infty$: wraparound combing

Kernel $\mathrm{P}_{\bar{a},\bar{b}^\infty}$

Affine SLCS for $\bar{a}$, $\bar{a}^2$, $\ldots$, $\bar{a}^k = a$ vs b: repeated squaring of $\mathrm{P}_{\bar{a},\bar{b}^\infty}$ in $\tilde{\mathbb{H}}_n$ by 3-period algorithm; $\log k$ iterations

Kernel $\mathrm{P}_{a,\bar{b}^\infty}$

Query $\mathrm{lcs}(a, \bar{b}^l) = \mathrm{lcs}(a, b)$ by dominance counting, respecting strand multiplicities

Overall time $O(\bar{m}\bar{n}) + \log k \cdot O(\bar{n} \log \bar{n}) = O\big((\bar{m} + \log k \log \bar{n}) \cdot \bar{n}\big)$

Cyclic and periodic string comparison

Doubly-periodic LCS

Doubly-periodic LCS: set as Problem L in Petrozavodsk Programming Camp 2023

ICPC (International Collegiate Programming Contest): top annual event for competitive programming

Petrozavodsk Programming Camp: long-standing conference/experimental ground for ICPC problem setters

- high geographical coverage
- stands out by problems' complexity and originality
- used to propose/test new ideas and approaches

Cyclic and periodic string comparison

Doubly-periodic LCS

Initial model solution: standard DP with optimisations

$O(\max(\bar{m}, \bar{n})^3 \log(\bar{m}k + \bar{n}l))$; solving up to $\bar{m} \leq 50$ (time limit 20 s)

Refined in succesive versions using sticky braids

Version with cubic $\boxdot$-mult via generic $\odot$-mult

$O(\bar{m}\bar{n} + \bar{n}^3 \log k)$; solving $\bar{m} \leq 50$ within 1 s; $\bar{m} = 500$ out of time

Version with quadratic $\boxdot$-mult via Monge $\odot$-multiplication (Knuth)

$O(\bar{m}\bar{n} + \bar{n}^2 \log k)$; solving $\bar{m} \leq 500$ within 10 s; $\bar{m} = 1000$ out of time

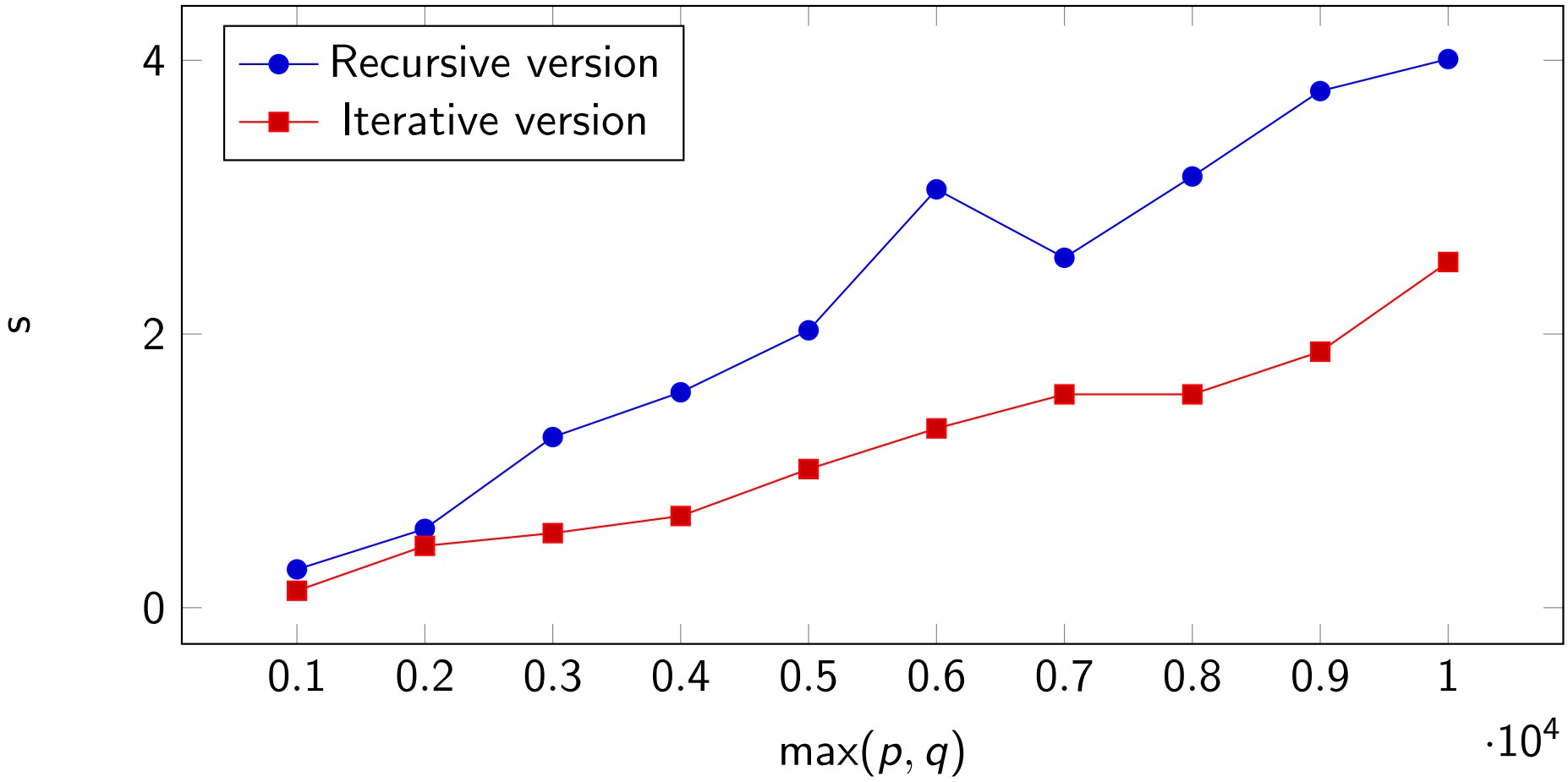
Set as challenge benchmark for contestants

Cyclic and periodic string comparison

Doubly-periodic LCS

Version with quasilinear $\boxdot$-mult (Steady Ant): $O(\bar{m}\bar{n} + \bar{n}\log\bar{n}\log k)$

Version with recursion-free Steady Ant: another speedup by approx $\times 2$



1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

LCS problem on permutation strings (LCSP)

LCS score for a vs b; in each of a, b all characters distinct

$lcs(\text{“CFAEDHGB”}, \text{“DEHCBAFG”}) = 3$

The LCSP problem is equivalent to

- longest increasing subsequence (LIS) in a (permutation) string
- maximum clique in a permutation graph
- maximum crossing-free matching in an embedded bipartite graph

$\mathrm{lis}(\text{“47621385”}) = \mathrm{lcs}(\text{“47621385”}, \text{“12345678”}) = 3$

Sparse string comparison

SLCS on permutation strings

LCSP: running time

$O(n \log n)$		implicit in [Erdös, Szekeres: 1935]
		[Robinson: 1938]
		[Knuth: 1970; Dijkstra: 1980]
$O(n \log \log n)$	unit-RAM	[Chang, Wang: 1992]
		[Bespamyatnikh, Segal: 2000]

LCSP generalisations

- canonical (anti)chain partition for a planar point set
- Robinson–Schensted–Knuth correspondence for Young tableaux

Semi-local LCSP (SLCSP) problem

SLCS scores for a vs b; in each of a, b all characters distinct

Equivalent/similar to

- local longest increasing subsequence (LIS): LIS in every substring of a string
- maximum clique in a circle graph

Sparse string comparison

SLCS on permutation strings

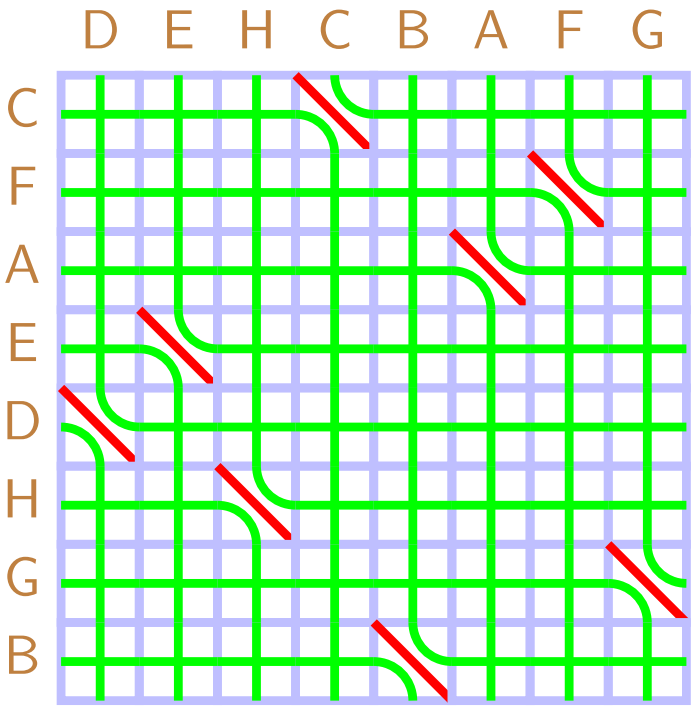
SLCSP: running time

$O(n^2 \log n)$		naive
$O(n^2)$	restricted	[Albert+: 2003; Chen+: 2005]
$O(n^{1.5} \log n)$	randomised, restricted	[Albert+: 2007]
$O(n^{1.5})$		[T: 2006]
$O(n \log^2 n)$		[T: 2010]

Sparse string comparison

SLCS on permutation strings

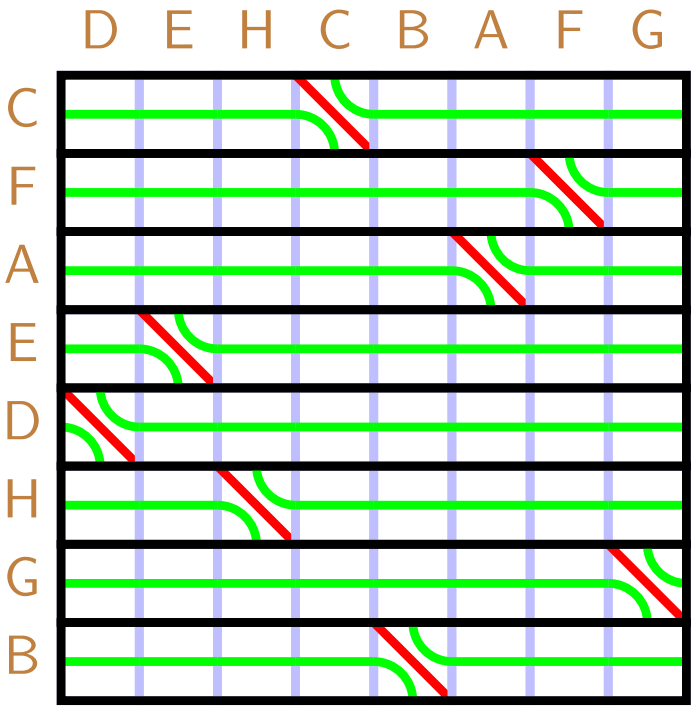
SLCSP by sparse recursive combing



Sparse string comparison

SLCS on permutation strings

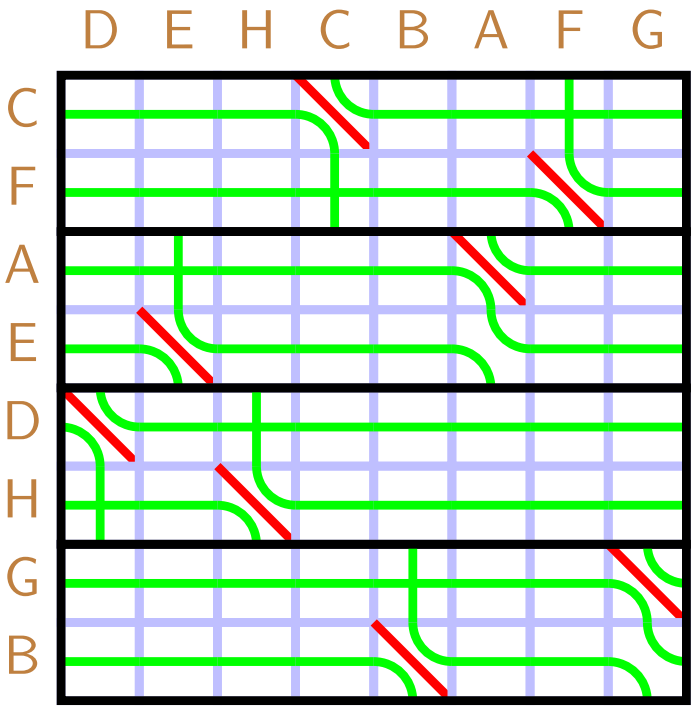
SLCSP by sparse recursive combing



Sparse string comparison

SLCS on permutation strings

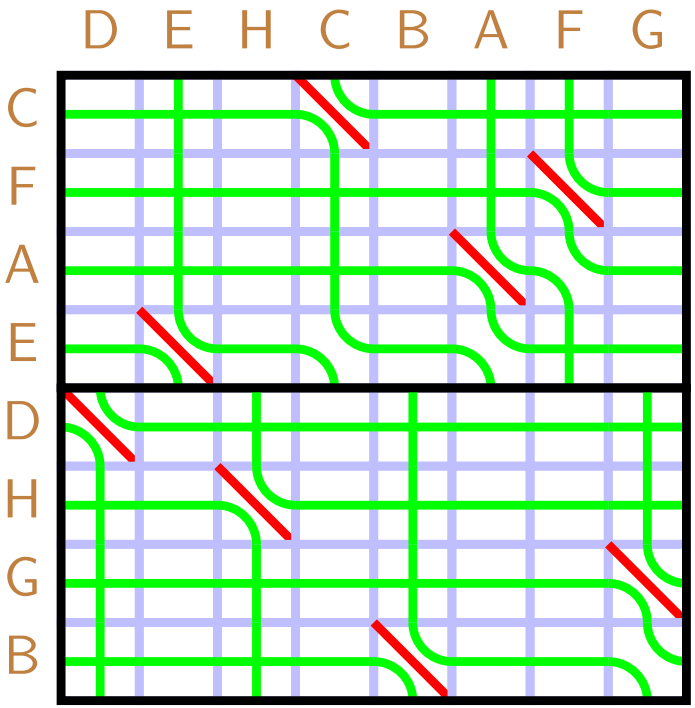
SLCSP by sparse recursive combing



Sparse string comparison

SLCS on permutation strings

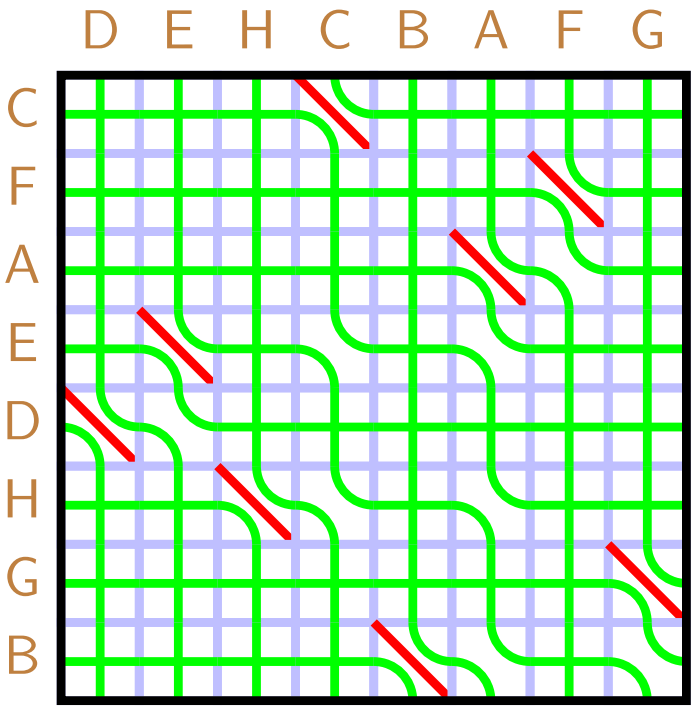
SLCSP by sparse recursive combing



Sparse string comparison

SLCS on permutation strings

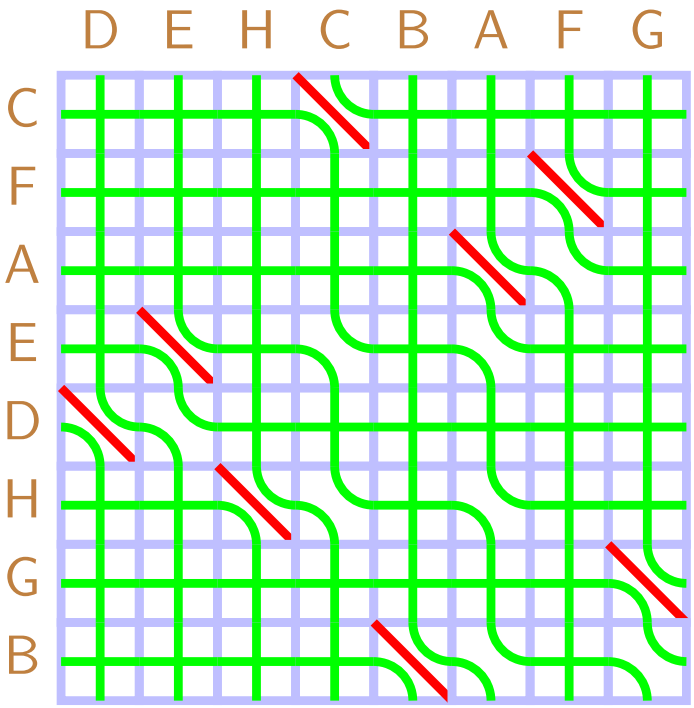
SLCSP by sparse recursive combing



Sparse string comparison

SLCS on permutation strings

SLCSP by sparse recursive combing



Sparse string comparison

SLCS on permutation strings

SLCSP: sparse recursive combing

Divide-and-conquer on the LCS grid

Divide grid (say) horizontally; two subproblems of effective size $n/2$

Conquer: matrix sticky multiplication, time $O(n \log n)$

Overall time $O(n \log^2 n)$

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Compressed string comparison

Grammar compression

SLP-string (also grammar-compressed string): a straight-line program (context-free grammar) generating $t = t_{\bar{n}}$ by $\bar{n}$ assignments of the form

- $t_k = \alpha$, where α is an alphabet character
- $t_k = uv$, where each of u, v is an alphabet character, or t_i for $i < k$

In general, $n = O(2^{\bar{n}})$

Captures various compression types, e.g. LZ78, LZW, Re-Pair

Related to many other compression types, e.g. LZ77, RL-BWT

Simplifying assumption: arithmetic up to n runs in $O(1)$; can be removed by careful index remapping

Compressed string comparison

Grammar compression

Example: Fibonacci string "ABAABABAABAAB"

$t_1 = \mathrm{A} \quad t_2 = t_1\mathrm{B} \quad t_3 = t_2 t_1 \quad t_4 = t_3 t_2 \quad t_5 = t_4 t_3$

$t = t_5 t_4 = ((((\mathrm{AB})\mathrm{A})\mathrm{AB})(\mathrm{AB})\mathrm{A})(((\mathrm{AB})\mathrm{A})\mathrm{AB})$

Compressed string comparison

Grammar compression

Example: LZ78 compression (Lempel-Ziv 1978)

Parsing string into sequence of phrases

Phrase: character or previous phrase + character

$t_1 = \mathrm{A} \quad t_2 = \mathrm{B} \quad t_3 = t_1\mathrm{A} \quad t_4 = t_2\mathrm{A} \quad t_5 = t_4\mathrm{A} \quad t_6 = t_5\mathrm{B}$

$t = t_1 t_2 t_3 t_4 t_5 t_6 =$ "(A)(B)((A)A)((B)A)((BA)A)((BAA)B)"

Compressed string comparison

Grammar compression

Example: BISECTION compression

Recursive splitting and matching

$t_1^1 = \mathrm{AB} \quad t_2^1 = \mathrm{AA} \quad t_3^1 = \mathrm{BA}$

$t_1^2 = t_1^1 t_2^1 \quad t_1^2 = t_2^1 t_3^1$

$t_1^3 = t_1^2 t_2^2 \quad t_2^3 = t_1^2 \mathrm{B}$

$t = t_1^3 t_2^3 = (((\mathrm{AB})(\mathrm{AA}))((\mathrm{BA})(\mathrm{BA})))(((\mathrm{AB})(\mathrm{AA}))\mathrm{B})$

Compressed string comparison

LCS on SLP-strings

Semi-SLP-compressed LCS: running time

$O(m^3\bar{n} + \ldots)$	CFG language	[Myers: 1995]
$O(m^{1.5}\bar{n})$	partial SLCS	[T: 2008]
$O(m^{1.2}\bar{n}^{1.4})$	$\mathbb{R}$ weights	[Hermelin+: 2009]
$O(m \log m \cdot \bar{n})$	partial SLCS	[T: 2010]
$O\big(m \log(m/\bar{n}) \cdot \bar{n}\big)$		[Hermelin+: 2010]
$O\big(m \log^{1/2}(m/\bar{n}) \cdot \bar{n}\big)$		[Gawrychowski: 2012]

SLP-compressed LCS: NP-hard [Lifshits: 2005]

Partial SLCS problem

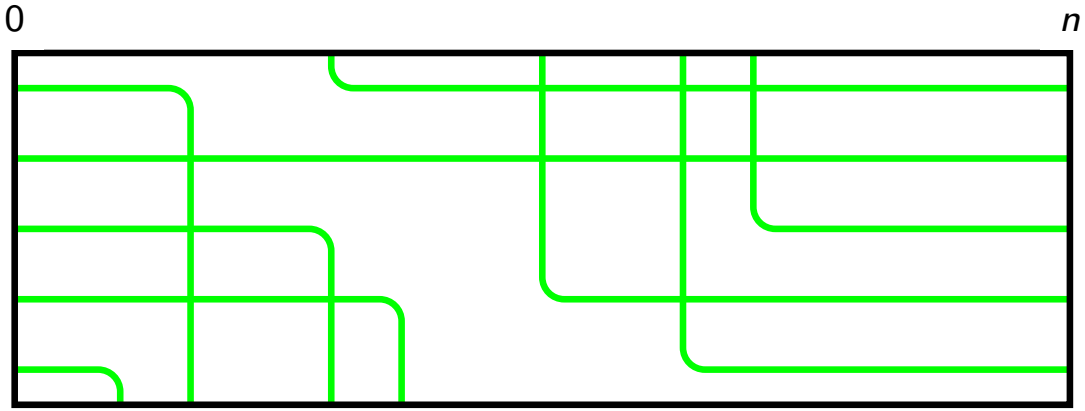
LCS scores for a vs b:

- ~~string-substring~~
- prefix-suffix (every prefix of a vs every suffix of b)
- suffix-prefix (every suffix of a vs every prefix of b)
- substring-string (every substring of a vs whole b)

Output scores represented implicitly in space $poly(m, \bar{n})$ when b an SLP-string

Compressed string comparison

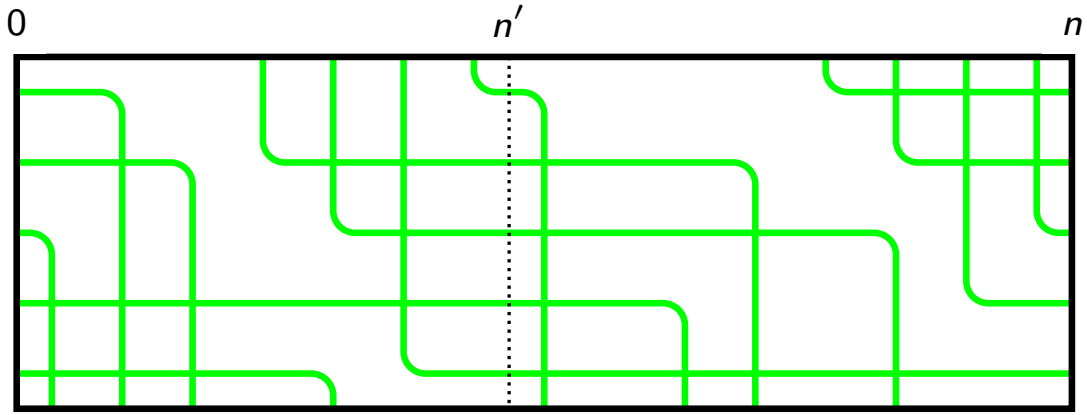
LCS on SLP-strings



Implicit partial SLSC scores: cross-SLCS kernels $P^{\swarrow(0)}_{a,b}$, $P^{\swarrow(n)}_{a,b}$

Compressed string comparison

LCS on SLP-strings



Divide-and-conquer

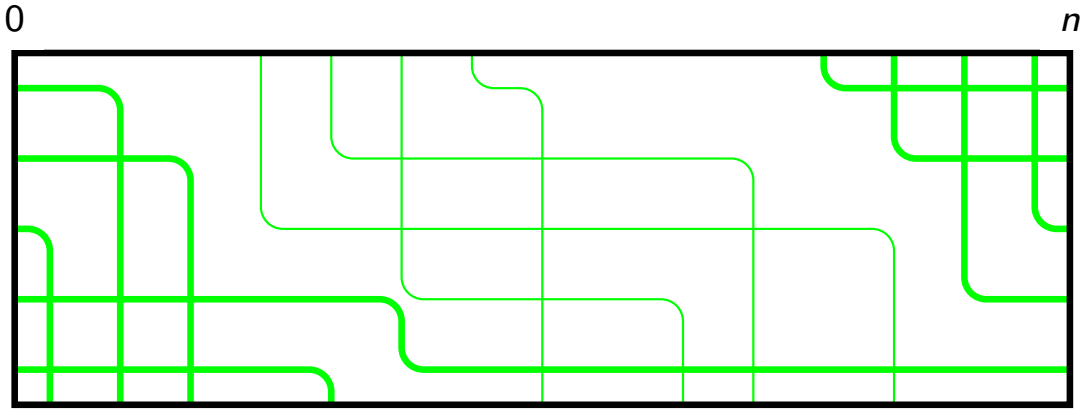
Kernels $\mathrm{P}^{\swarrow(0)}_{a,b'}$, $\mathrm{P}^{\swarrow(n')}_{a,b}$, $\mathrm{P}^{\swarrow(n')}_{a,b''}$, $\mathrm{P}^{\swarrow(n)}_{a,b''}$

$\mathrm{P}^{\swarrow(n')}_{a,b'} \boxdot \mathrm{P}^{\swarrow(n')}_{a,b''} = \mathrm{P}^{\swarrow(n')}_{a,b}$

Obtain $\mathrm{P}^{\swarrow(0)}_{a,b}$, $\mathrm{P}^{\swarrow(n)}_{a,b}$

Compressed string comparison

LCS on SLP-strings



Divide-and-conquer

Kernels $\mathrm{P}^{\swarrow(0)}_{a,b'}$, $\mathrm{P}^{\swarrow(n')}_{a,b}$, $\mathrm{P}^{\swarrow(n')}_{a,b''}$, $\mathrm{P}^{\swarrow(n)}_{a,b''}$

$\mathrm{P}^{\swarrow(n')}_{a,b'} \boxdot \mathrm{P}^{\swarrow(n')}_{a,b''} = \mathrm{P}^{\swarrow(n')}_{a,b}$; obtain $\mathrm{P}^{\swarrow(0)}_{a,b}$, $\mathrm{P}^{\swarrow(n)}_{a,b}$

Semi-SLP-compressed partial SLCS: the algorithm

Iteration over SLP-string b

For every nonterminal string $b_k = uv$, length n_k partial SLCS kernels $\mathrm{P}^{\swarrow(0)}_{a,b_k}$, $\mathrm{P}^{\swarrow(n_k)}_{a,b_k}$: multiplication in $\mathbb{H}_m$, time $O(m \log m)$

$\bar{n}$ iterations, overall time $O(m \log m \cdot \bar{n})$

Indices grow exponentially: index remapping at every iteration

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Local string comparison

Window-substring LCS

w-window: any substring of fixed length w

Window-substring LCS problem

LCS score for every w-window of a vs every substring of b

Local string comparison

Window-substring LCS

w-window: any substring of fixed length w

Window-substring LCS problem

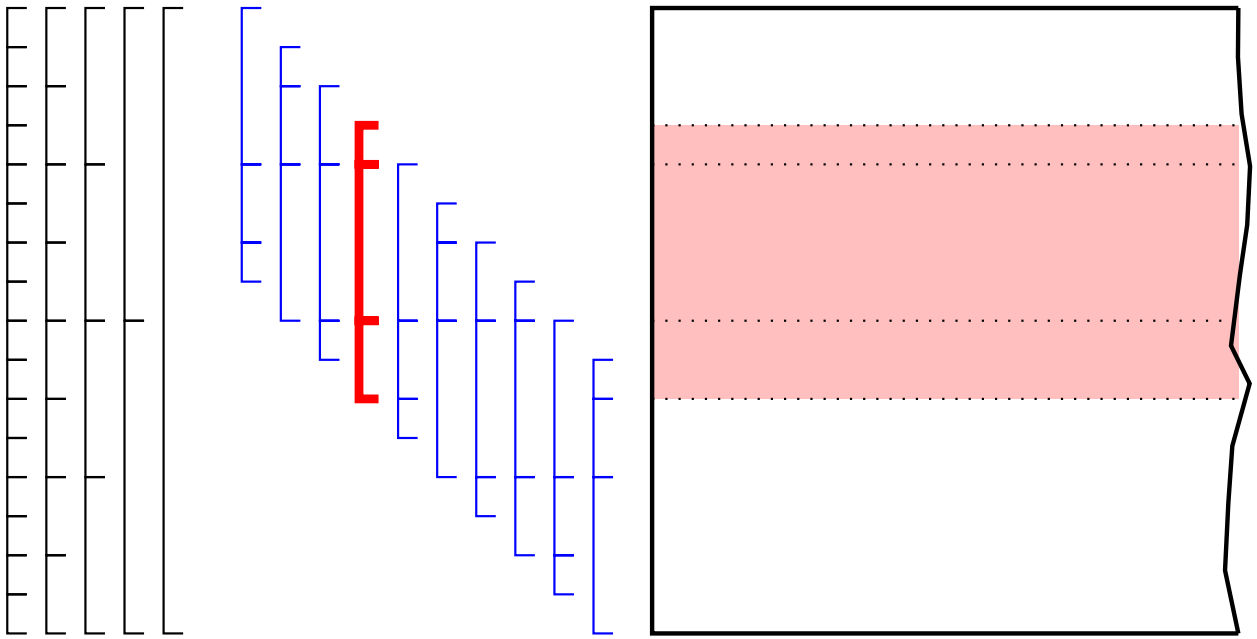
LCS score for every w-window of a vs every substring of b

Window-substring LCS: running time

$O(mn^2w) = mn$ runs $\cdot\, O(nw)$	repeated DP
$O(mnw) = m$ runs $\cdot\, O(nw)$	repeated iterative combing
$O(mn)$	[Krusche, T: 2010]

Local string comparison

Window-substring LCS



Local string comparison

Window-substring LCS

Window-substring LCS: the algorithm

Obtain SLCS kernels for canonical horizontal strips of a vs b

Obtain SLCS kernels for every w-window of a vs b: $\boxdot$-product of canonical kernels

Overall time $O(mn)$

Window-window LCS problem

LCS score for every w-window of a vs every w-window of b

Provides an LCS-scored alignment plot (by analogy with Hamming-scored dot plot), a useful tool for studying evolutionary genome conservation

Running time dominated by window-substring LCS

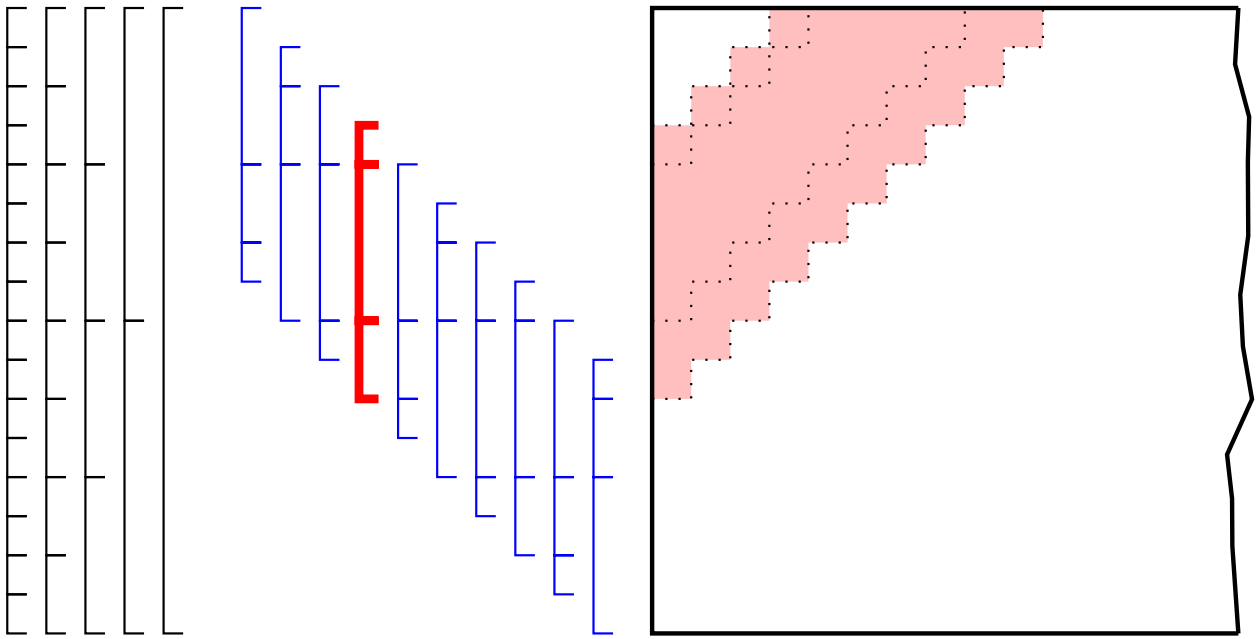
Fixed total length LCS problem

LCS score for every substring a' of a vs every substring b' of b, $|a| + |b| = w$.

Running time: $O(mn)$, same as window-substring LCS

Local string comparison

Window-substring LCS



Fixed total length LCS: the algorithm

Obtain SLCS kernels for canonical antidiagonal strips

Obtain SLCS kernels for antidiagonal strips of width w: $\boxdot$-product of canonical kernels

Provides LCS for every pairs of substrings in a vs b of total length w

Overall time $O(mn)$

Window-window LCS problem

LCS score for every w-window of a vs every w-window of b

Provides an LCS-scored alignment plot (by analogy with Hamming-scored dot plot), a useful tool for studying evolutionary genome conservation

Running time dominated by window-substring LCS

Fragment-substring LCS problem

Given a set of m (possibly overlapping) fragment substrings in a, LCS score for every fragment of a vs every substring of b

Local string comparison

Fragment-substring LCS

Fragment-substring LCS problem

Given a set of m (possibly overlapping) fragment substrings in a, LCS score for every fragment of a vs every substring of b

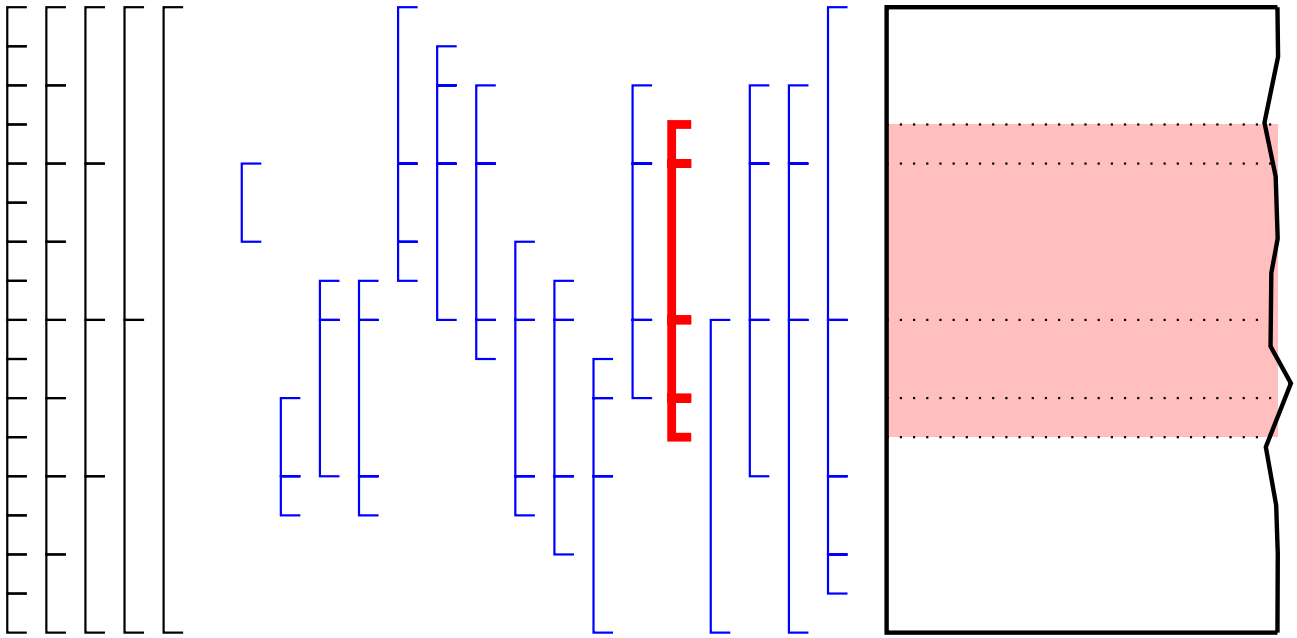
Fragment-substring LCS: running time

$O(m^2 n) = m$ runs $\cdot\, O(mn)$ repeated iterative combing

$O(mn)$ [T: 2008]

Local string comparison

Fragment-substring LCS



Fragment-substring LCS: the algorithm

Obtain SLCS kernels for canonical horizontal strips of a vs b

Obtain SLCS kernels for every fragment of a vs b: $\boxdot$-product of canonical kernels, semigroup range product problem

Overall time $O(mn)$

Local string comparison

Spliced alignment

Spliced alignment problem

Chain of non-overlapping fragments in a, closest to b by alignment score

Describes gene assembly from candidate exons, given a reference gene

Assume $m = n$; let $s =$ total length of fragments

Spliced alignment: running time

$O(ns) = O(n^3)$	[Gelfand+: 1996]
$O(n^{2.5})$	[Kent+: 2006]
$O(n^2 \log^2 n)$	[T: unpublished]
$O(n^2 \log n)$	[Sakai: 2009]

Local string comparison

Local approximate matching

Fix scoring scheme: match $w_+ > 0$, mismatch w_0, gap $w_- < 0$

Local approximate matching (LAM) problem

For every prefix a' of a and every prefix b' of b, maximum alignment score between a suffix of a' and a suffix of b'

In particular, provides maximum alignment score between a substring of a and a substring of b

Degenerates for $w_- = 0$ (in particular, for LCS score)

$h[l, j] = \max_{k,i} score\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$

$\max_{l,j} h[l, j]$

LAM: running time

$O(mn)$ [Smith, Waterman: 1981]

LAM: the algorithm

$$h[l,j] \leftarrow \max \begin{cases} h[l-1,j-1] + score([l-1,j-1] \rightarrow [l,j]) \\ h[l-1,j] + w_- \\ h[l,j-1] + w_- \\ 0 \end{cases}$$

Cell update: time $O(1)$ Overall time $O(mn)$

Local string comparison

Local approximate matching

LAM maximises alignment score across all substring lengths

May not always be the most relevant/robust alignment:

- too short to be significant, prefer lower-scoring but longer alignment
- too long ("shadow" and "mosaic" effects), prefer lower-scoring but shorter alignment

Bounded one-sided LAM (BO-LAM)

Given $w \geq 0$, for every prefix a' of a and every prefix b' of b, maximum alignment score between a suffix a'' of a' and a suffix of b', where $|a''| \geq w$

In particular, provides maximum alignment score between a substring of a of length $\geq w$ and a substring of b

$h[l, j] = \max_{k,i} score\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$

$h^o_{\geq w}[l, j] = \max_{k,i;l-k\geq w} score\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$

$\max_{l,j} h^o_{\geq w}[l, j]$

Bounded symmetric LAM (BS-LAM)

Given $w \geq 0$, for every prefix a' of a and every prefix b' of b, maximum alignment score between a suffix a'' of a' and a suffix b'' of b', where $|a''| + |b''| \geq w$

In particular, provides maximum alignment score between a substring of a and a substring of b, of total length $\geq w$

$h[l, j] = \max_{k,i} score\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$

$h^s_{\geq w}[l, j] = \max_{k,i;l-k+j-i\geq w} score\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$

$\max_{l,j} h^o_{\geq w}[l, j]$

BO- & BS-LAM: running time

$O(mn^2)$	exact	[Arslan, Eğecioğlu: 2004]
$O(mn/\epsilon)$	approx ϵ-relaxed	[Arslan, Eğecioğlu: 2004]
$O(mn)$	exact, rational	[T: 2019]

Approximate ϵ-relaxed BS-LAM: as ordinary BS-LAM, except

- $|a'| + |b'| \geq (1 - \epsilon)w$
- $score(a', b') \geq h^s_{\geq w}[l, j]$

BO-LAM: the algorithm

Window-substring alignment: $H_{a[k:l],b}[i,j]$, $l-k=w$

Implicit unit-Monge matrix searching: $h^o_{=w}[l,j] = \max_{i\in[0:j]} H_{a\langle k:l\rangle,b}[i,j]$

$$h[l,j] \leftarrow \max \begin{cases} h[l-1,j-1] + score([l-1,j-1] \to [l,j]) \\ h[l-1,j] + w_- \\ h[l,j-1] + w_- \\ h^o_{=w}[l,j] \end{cases}$$

Cell update: time $O(1)$ Overall time $O(mn)$

BS-LAM: similar; first stage replaced by fixed total length alignment

Normalised alignment problem [Arslan, Eğecioğlu, Pevzner: 2001]

Normalised alignment score $nscore(a, b) = \frac{score(a,b)}{|a|+|b|}$

Normalised BS-LAM

Given $w \geq 0$, for every prefix a' of a and every prefix b' of b, maximum normalised alignment score between a suffix a'' of a' and a suffix b'' of b', where $|a''| + |b''| \geq w$

In particular, provides maximum normalised alignment score between a substring of a and a substring of b, of total length $\geq w$

$\hbar^s_{\geq w}[l, j] = \max_{k,i;l-k+j-i\geq w} nscore\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$

$\max_{l,j} \hbar^s_{\geq w}[l, j]$

Normalised BS-LAM: running time

$O(mn \log n/\epsilon)$	approx ϵ-relaxed, rational	[Arslan, Eğecioğlu: 2004]
$O(mn \log n)$	exact, rational	[T: NEW]

Approximate ϵ-relaxed normalised BS-LAM: as normalised BS-LAM, except

- $|a'| + |b'| \geq (1 - \epsilon)w$
- $nscore(a', b') \geq \hbar^s_{\geq w}[l, j]$

Normalised BS-LAM: the algorithm

Fractional programming: binary search for optimal alignment score

$\leq \log n$ iterations

Each iteration: BS-LAM with adjusted scoring scheme

Running time: $\log n \cdot O(mn) = O(mn \log n)$

Local LCS oracle

Preprocess a, b to answer LCS score queries for a substring of a vs a substring of b

Local LCS oracle: running time ($m = n$)

preproc	memory	query	
$O(n^2)$	$O(n^2)$	$O(n)$	[Sakai: 2019]
$O(n^2)$	$O(n^2)$	$O(n^{1/2})$	[Sakai: 2022]
$O\left(n^{2+o(1)}\right)$	$O\left(n^{2+o(1)}\right)$	$O\left((\log n)^{O(1)}\right)$	[Charalampopoulos+: 2021]

An Almost Optimal Edit Distance Oracle

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Israel

Paweł Gawrychowski
University of Wrocław, Poland

Shay Mozes
The Interdisciplinary Center Herzliya, Israel

Oren Weimann
University of Haifa, Israel

Abstract

We consider the problem of preprocessing two strings S and T, of lengths m and n, respectively, in order to be able to efficiently answer the following queries: Given positions i, j in S and positions a, b in T, return the optimal alignment score of $S[i\,..\,j]$ and $T[a\,..\,b]$. Let $N = mn$. We present an oracle with preprocessing time $N^{1+o(1)}$ and space $N^{1+o(1)}$ that answers queries in $\log^{2+o(1)} N$ time. In other words, we show that we can efficiently query for the alignment score of every pair of substrings after preprocessing the input for almost the same time it takes to compute just the alignment of S and T. Our oracle uses ideas from our distance oracle for planar graphs [STOC 2019] and exploits the special structure of the alignment graph. Conditioned on popular hardness conjectures, this result is optimal up to subpolynomial factors. Our results apply to both edit distance and longest common subsequence (LCS).

The best previously known oracle with construction time and size $\mathcal{O}(N)$ has slow $\Omega(\sqrt{N})$ query

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

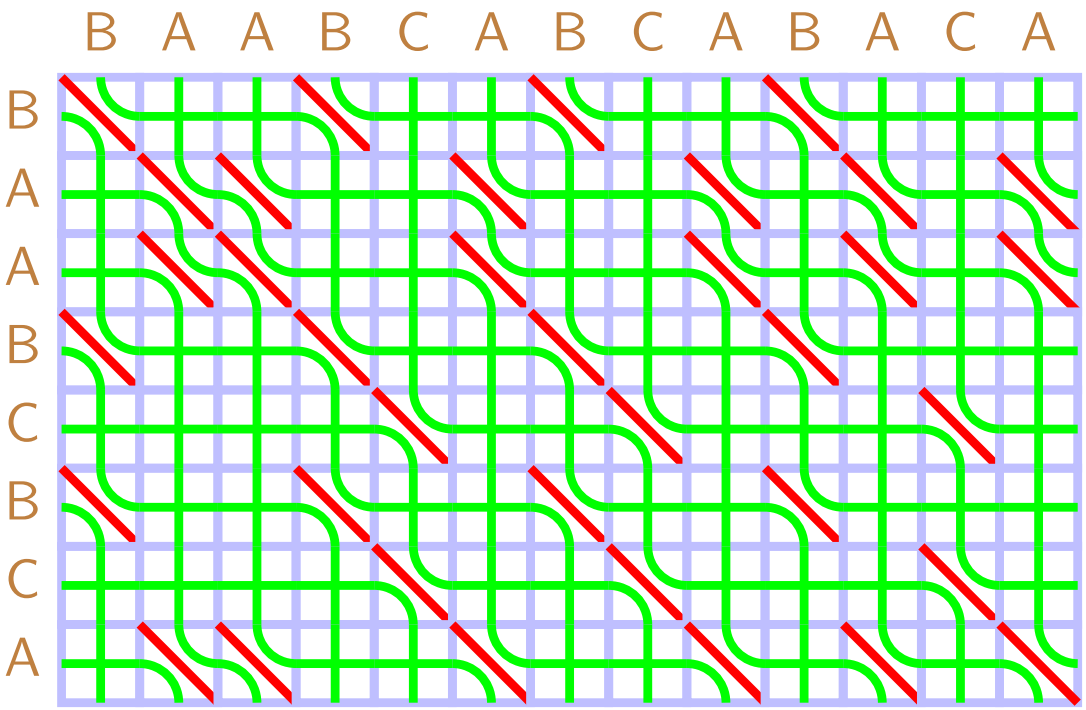
Bit-parallel computation:

- registers: bit vectors of size v
- standard Boolean logic and integer arithmetic

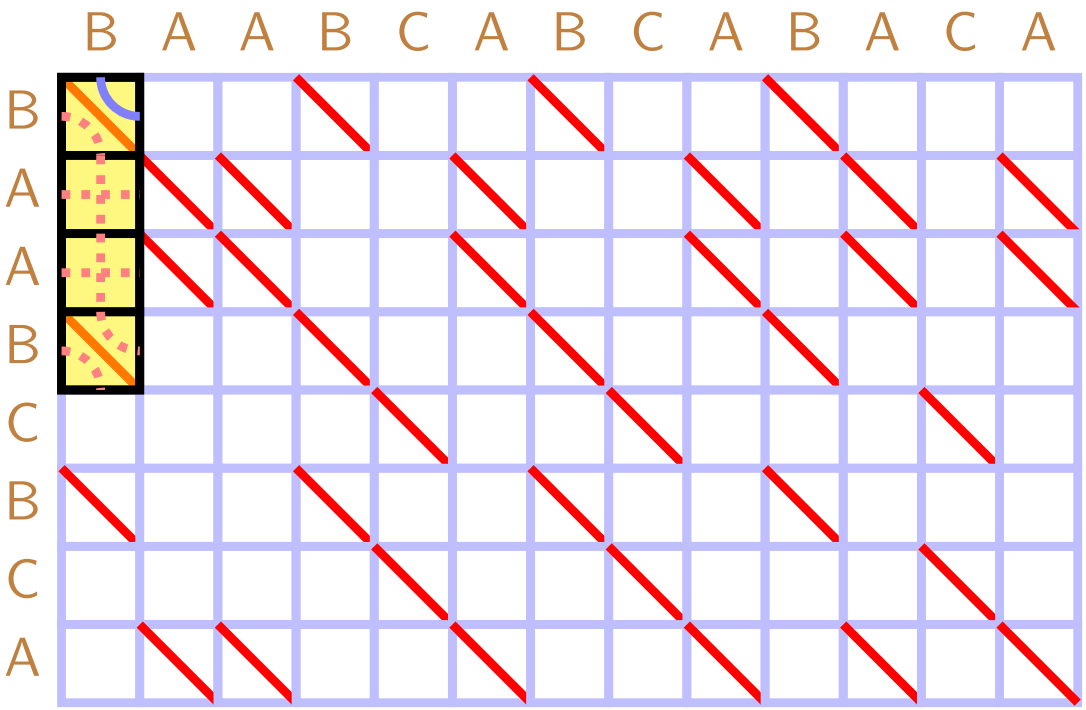
LCS: bit-parallel, running time

$O(\frac{mn}{v})$	tiles	[Allison, Dix: 1986] [Myers: 1999] [Crochemore+: 2001] [Hyyrö: 2004, 2017]
$O(\frac{mn}{v})$	anti-diagonals	[Mishin+: 2021]

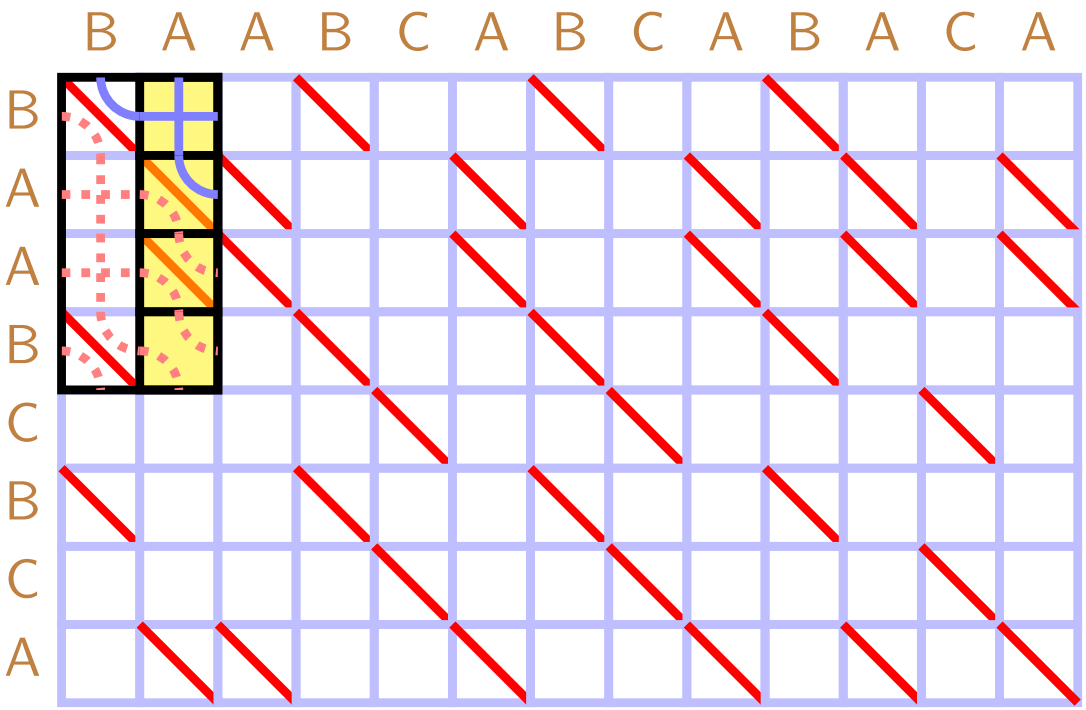
LCS by bit-parallel iterative combing (tiles)



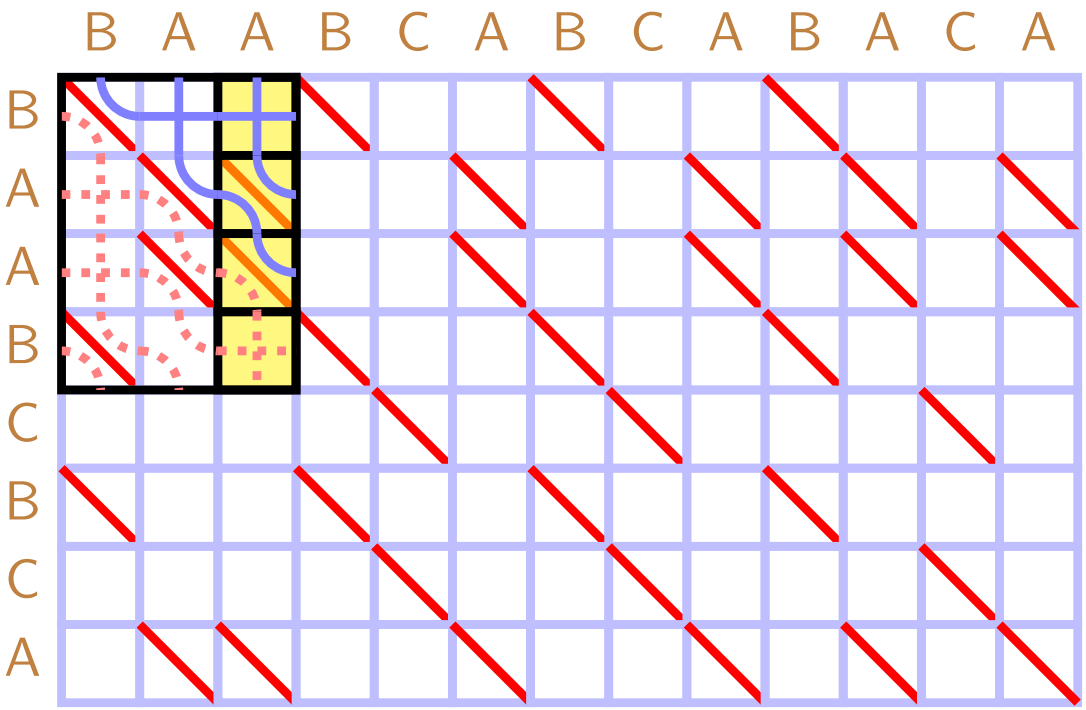
LCS by bit-parallel iterative combing (tiles)



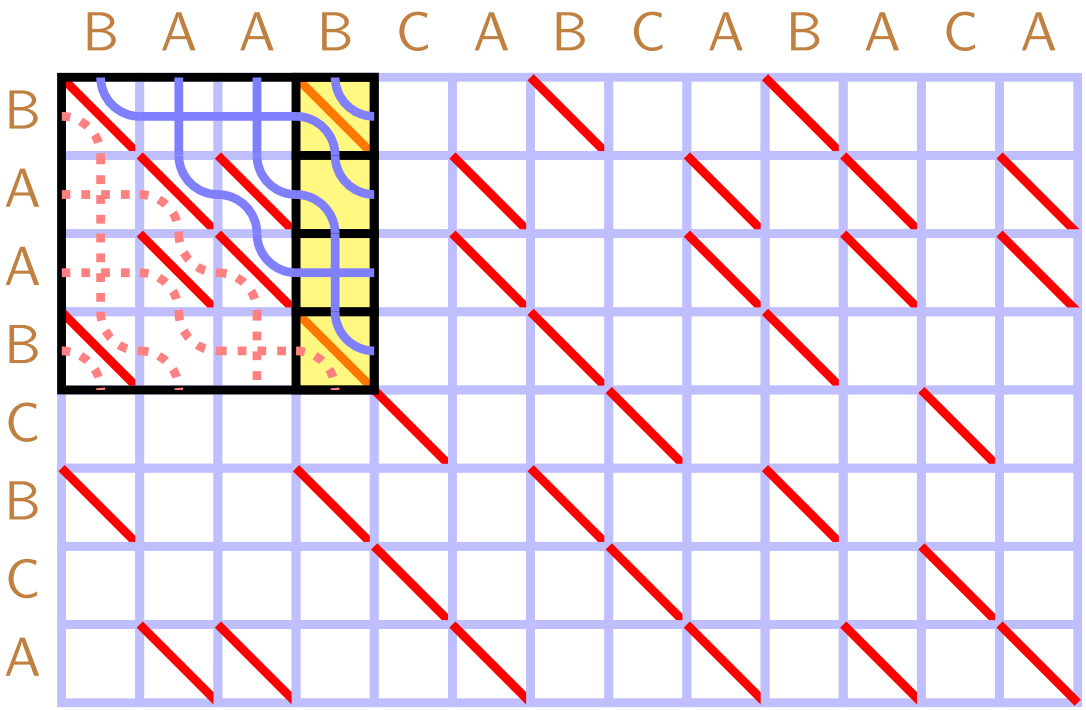
LCS by bit-parallel iterative combing (tiles)



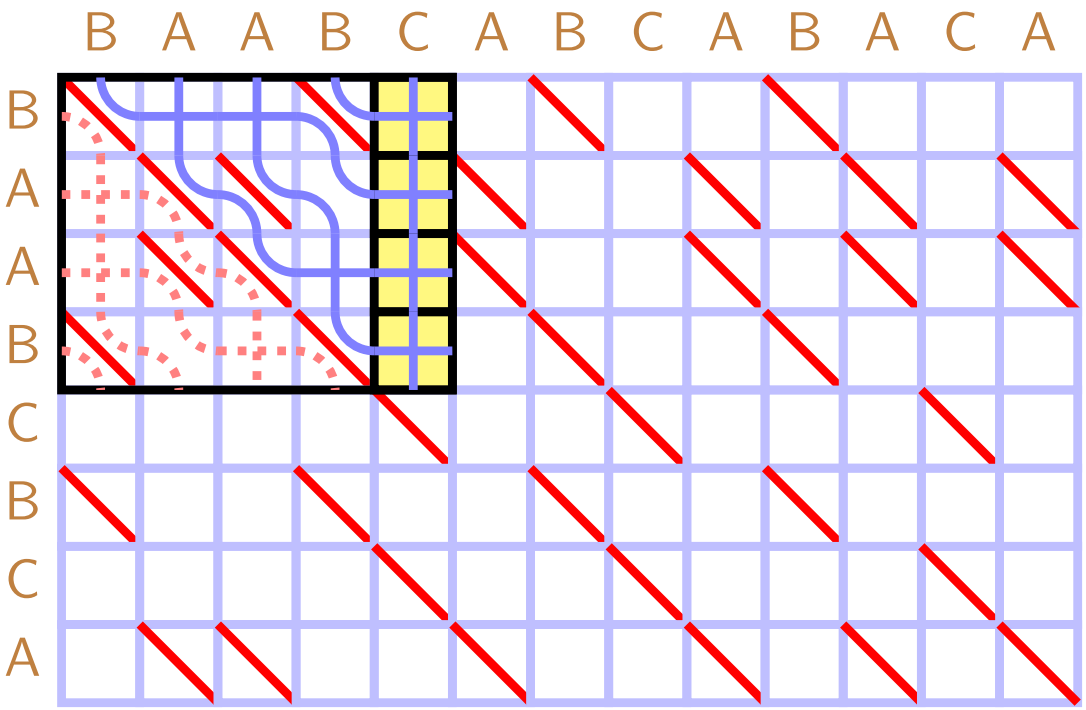
LCS by bit-parallel iterative combing (tiles)



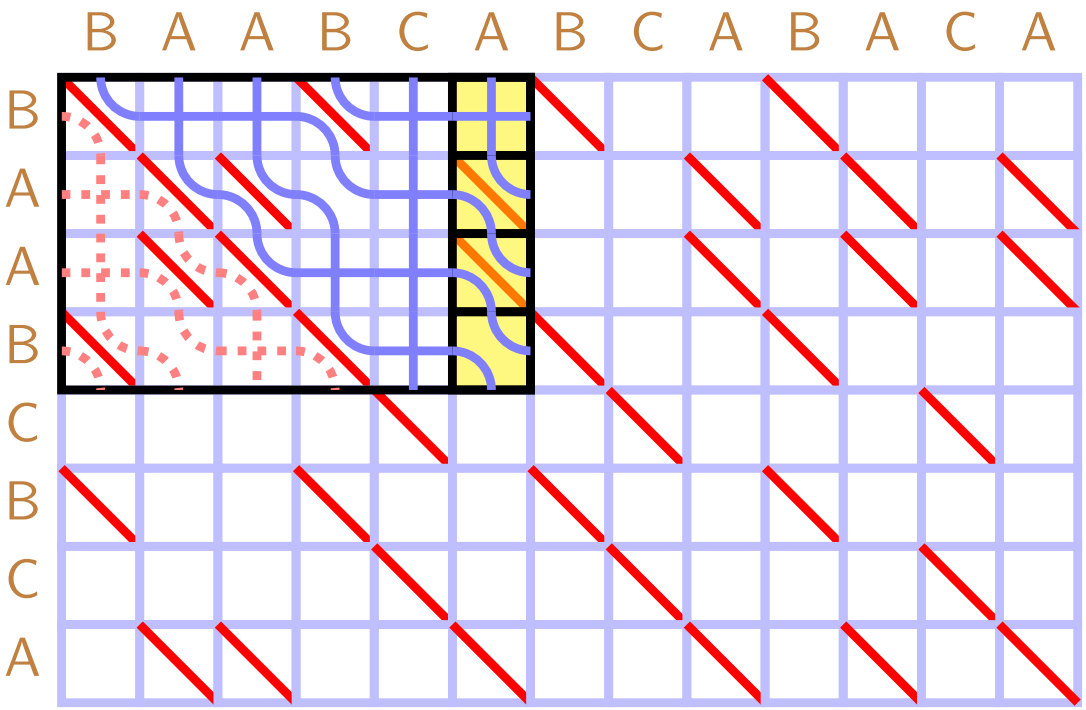
LCS by bit-parallel iterative combing (tiles)



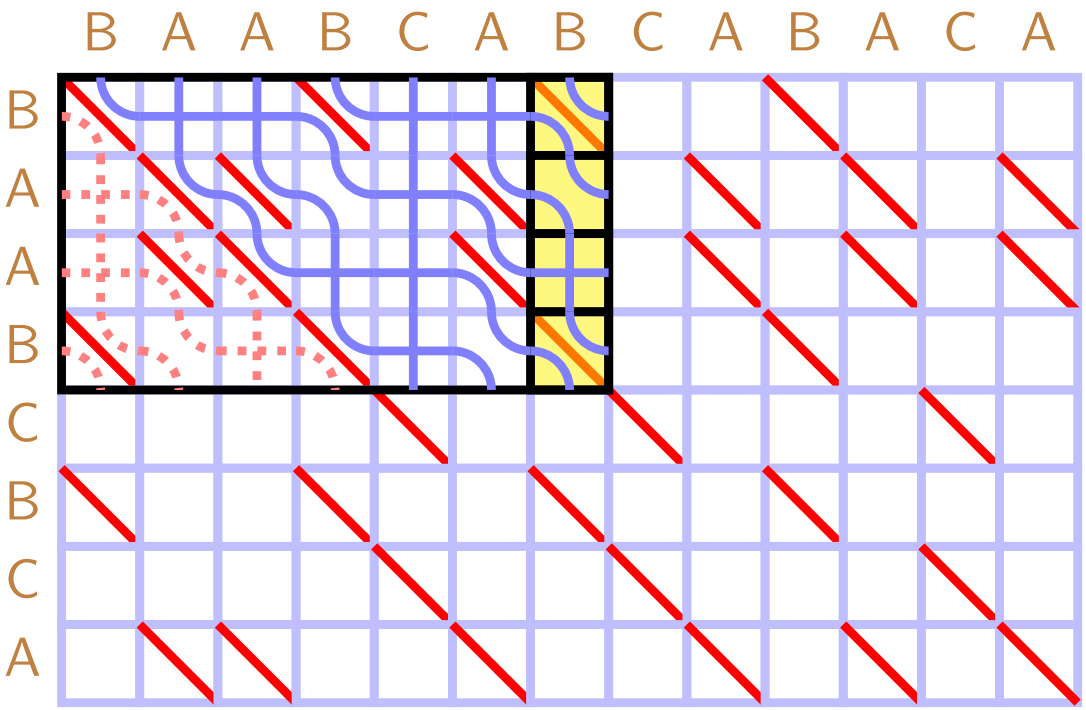
LCS by bit-parallel iterative combing (tiles)



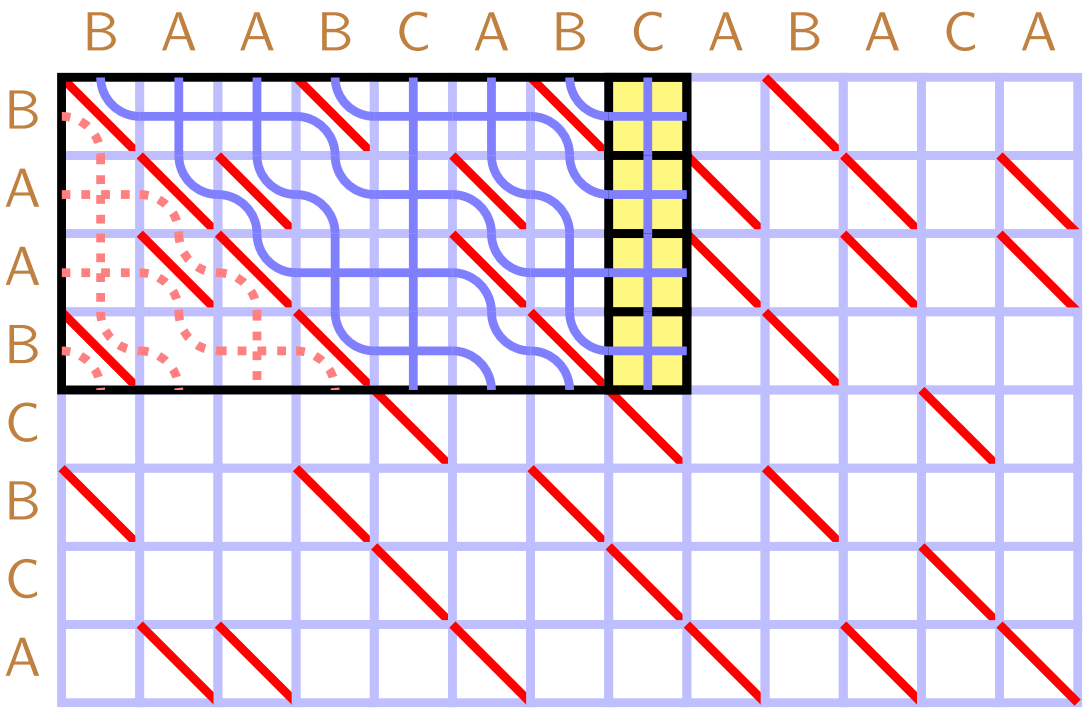
LCS by bit-parallel iterative combing (tiles)



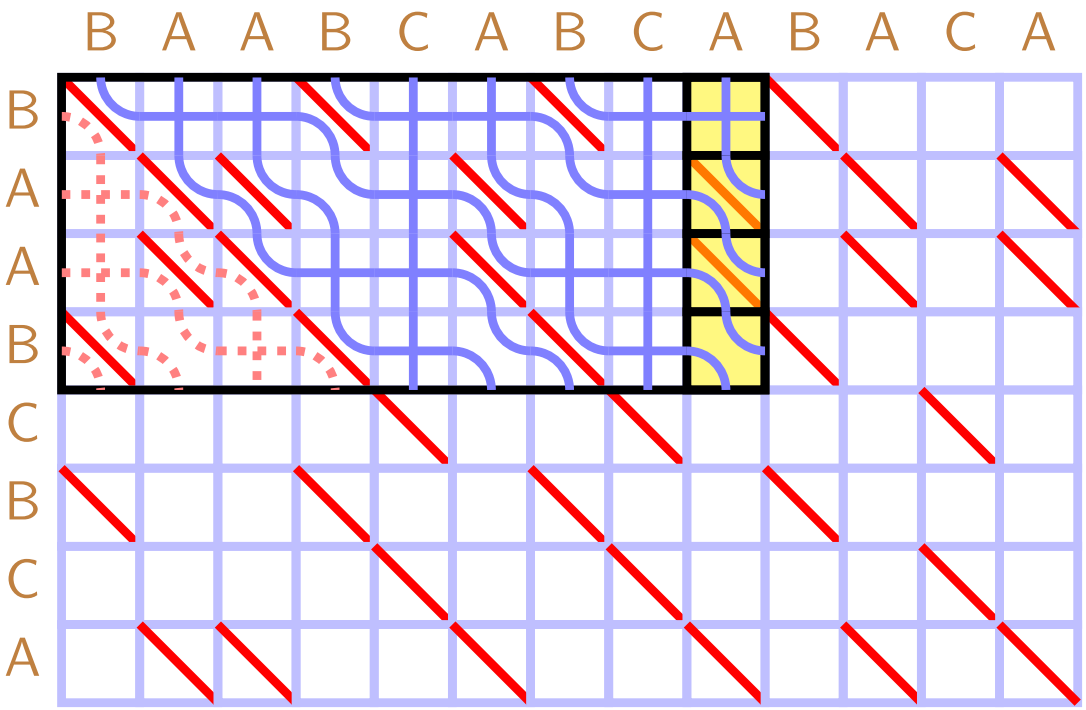
LCS by bit-parallel iterative combing (tiles)



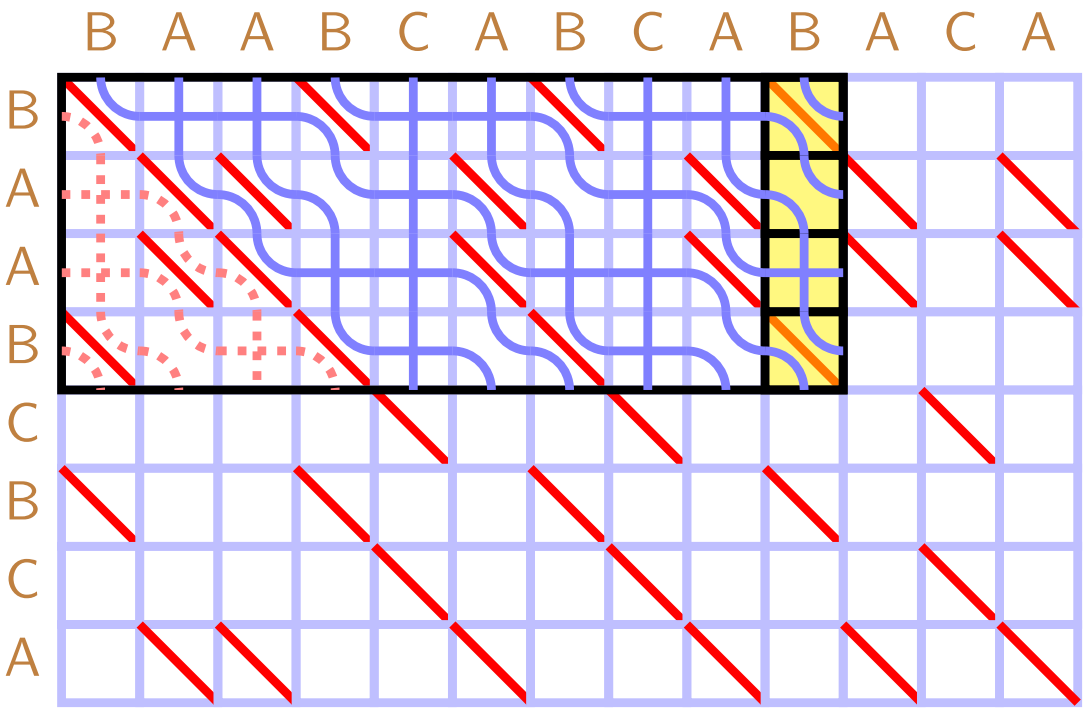
LCS by bit-parallel iterative combing (tiles)



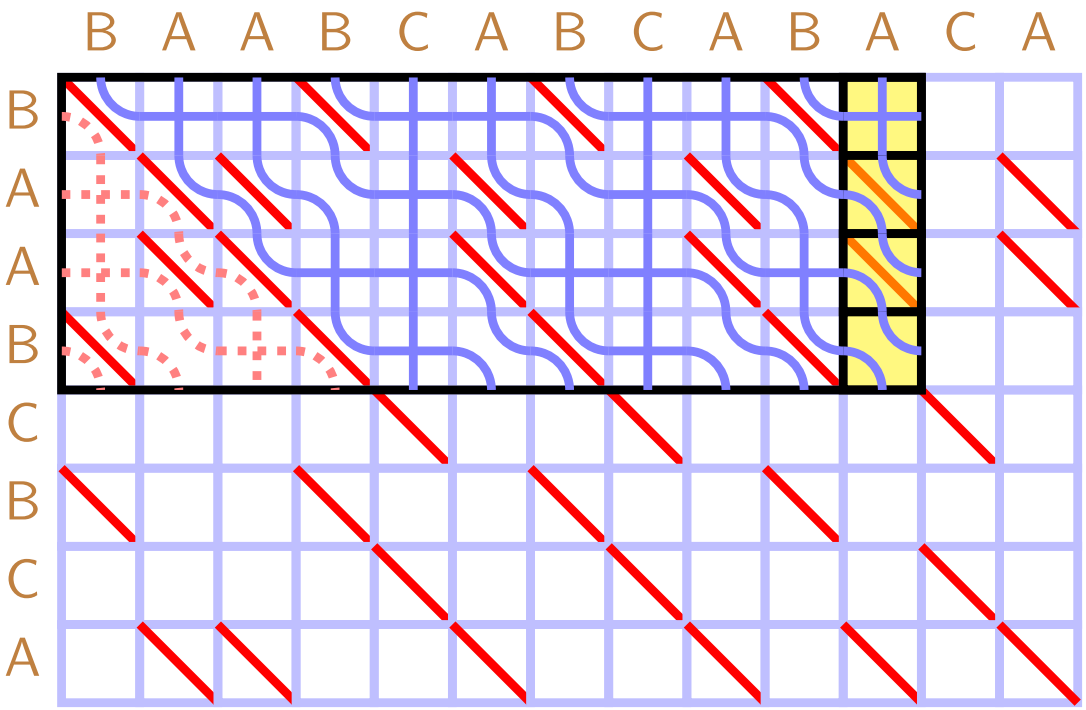
LCS by bit-parallel iterative combing (tiles)



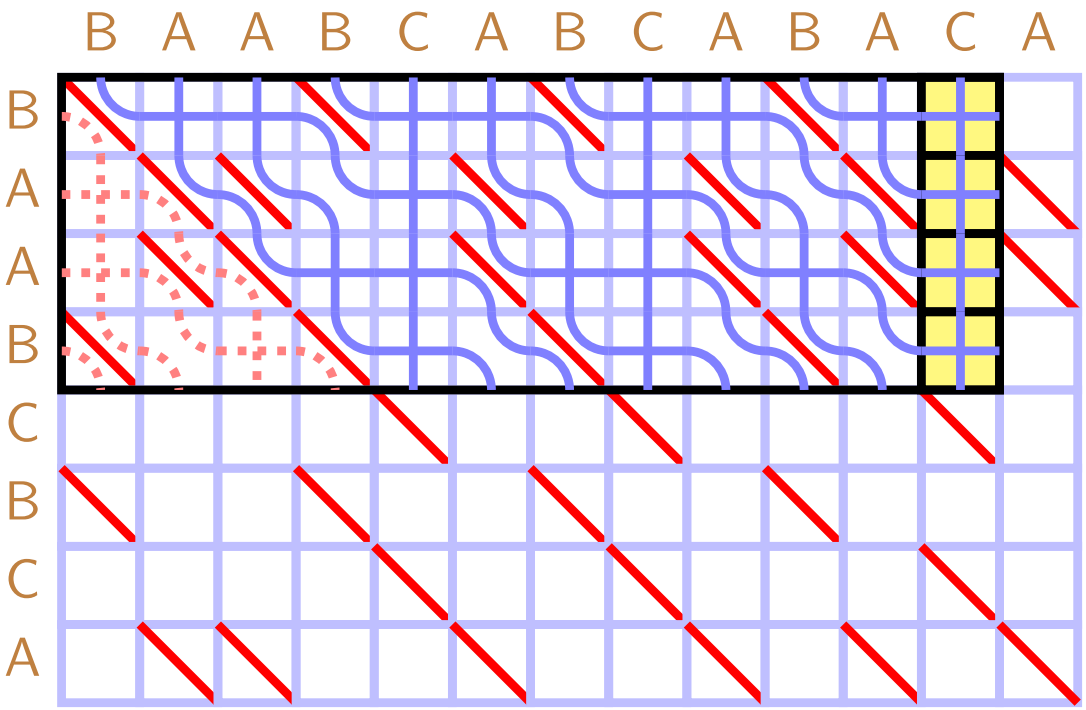
LCS by bit-parallel iterative combing (tiles)



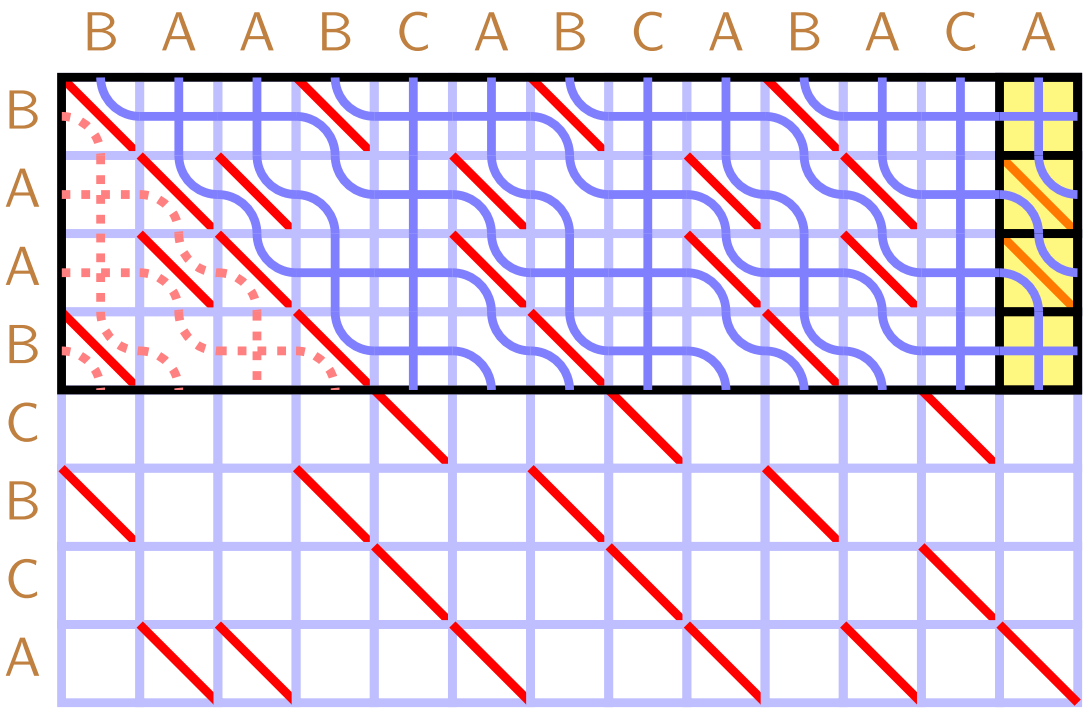
LCS by bit-parallel iterative combing (tiles)



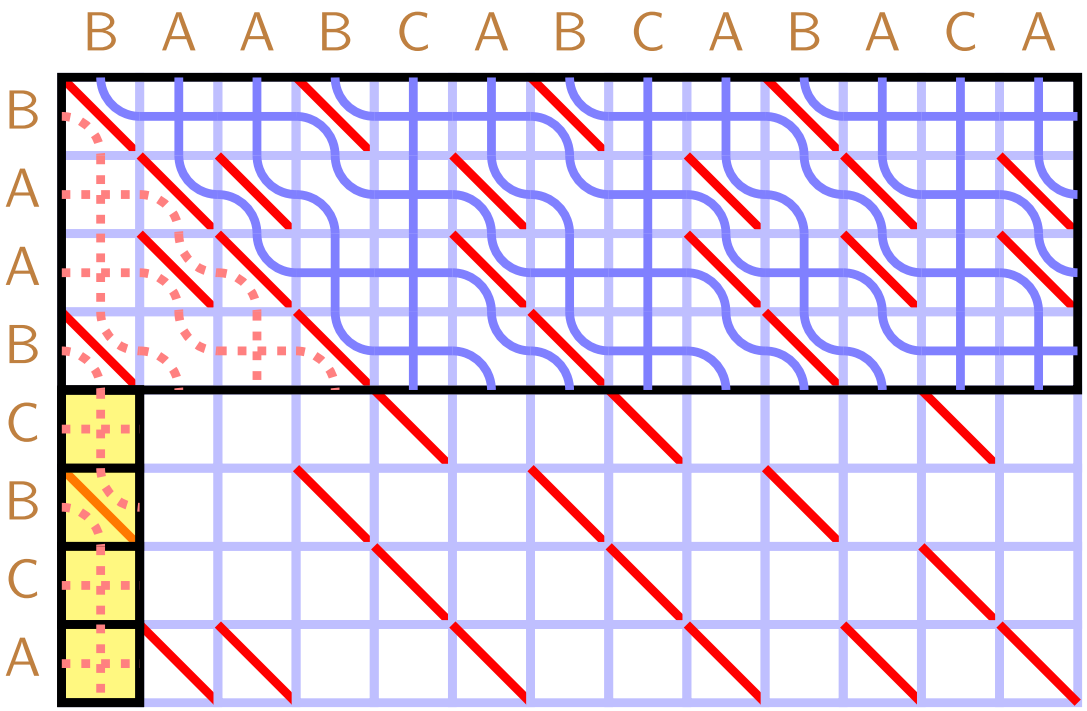
LCS by bit-parallel iterative combing (tiles)



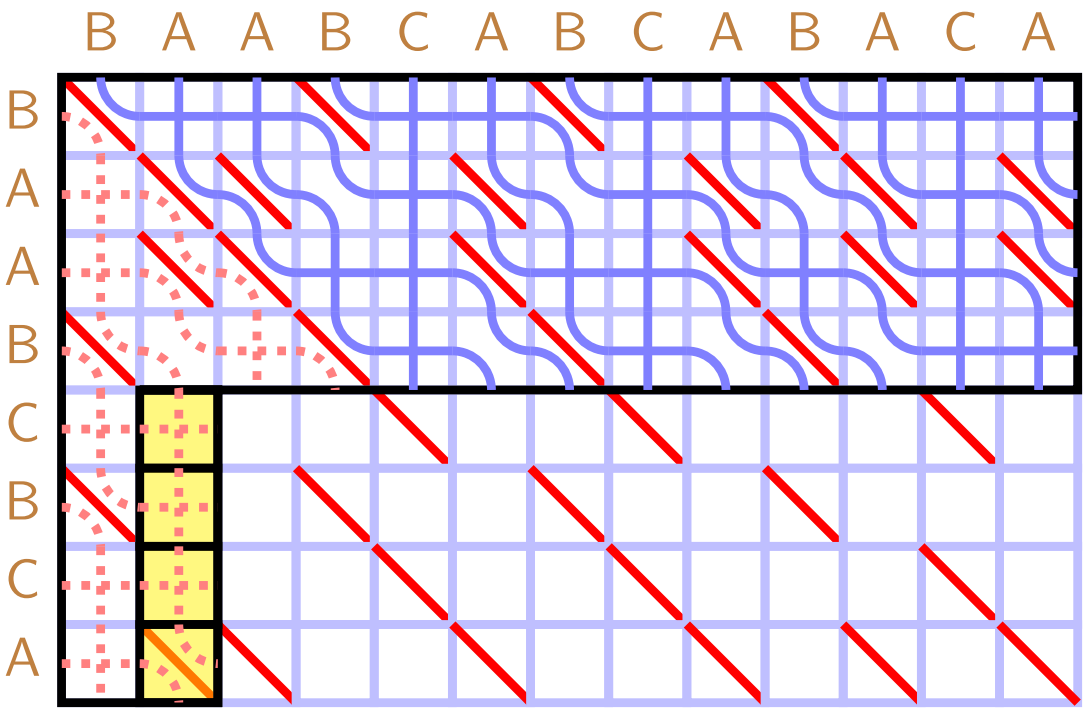
LCS by bit-parallel iterative combing (tiles)



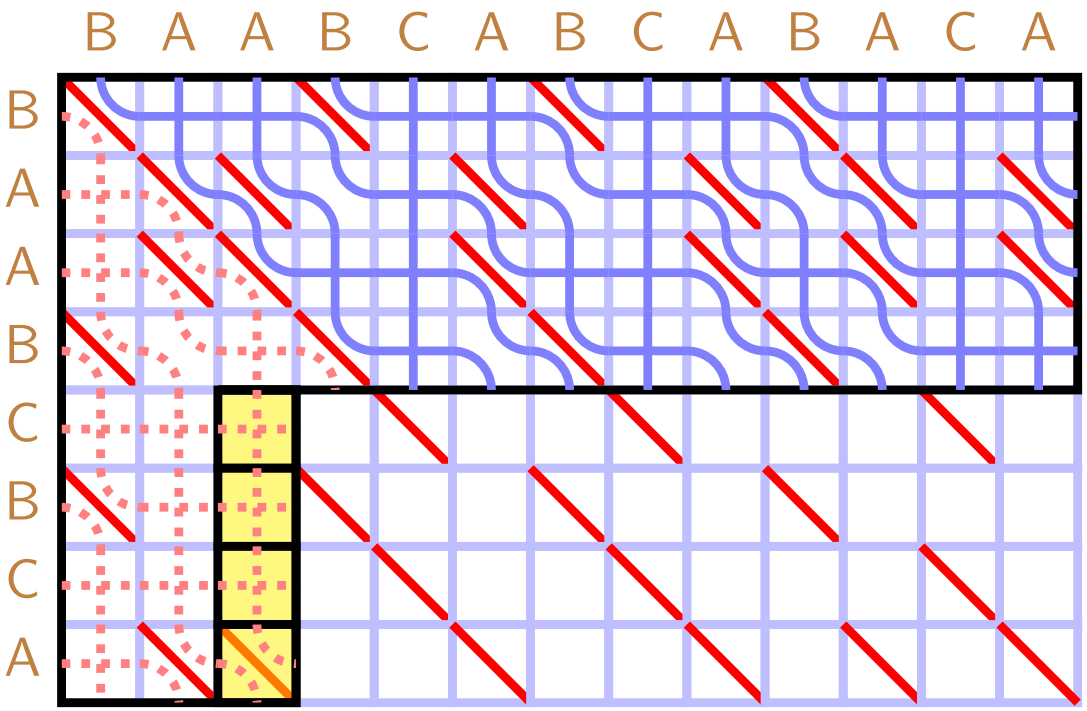
LCS by bit-parallel iterative combing (tiles)



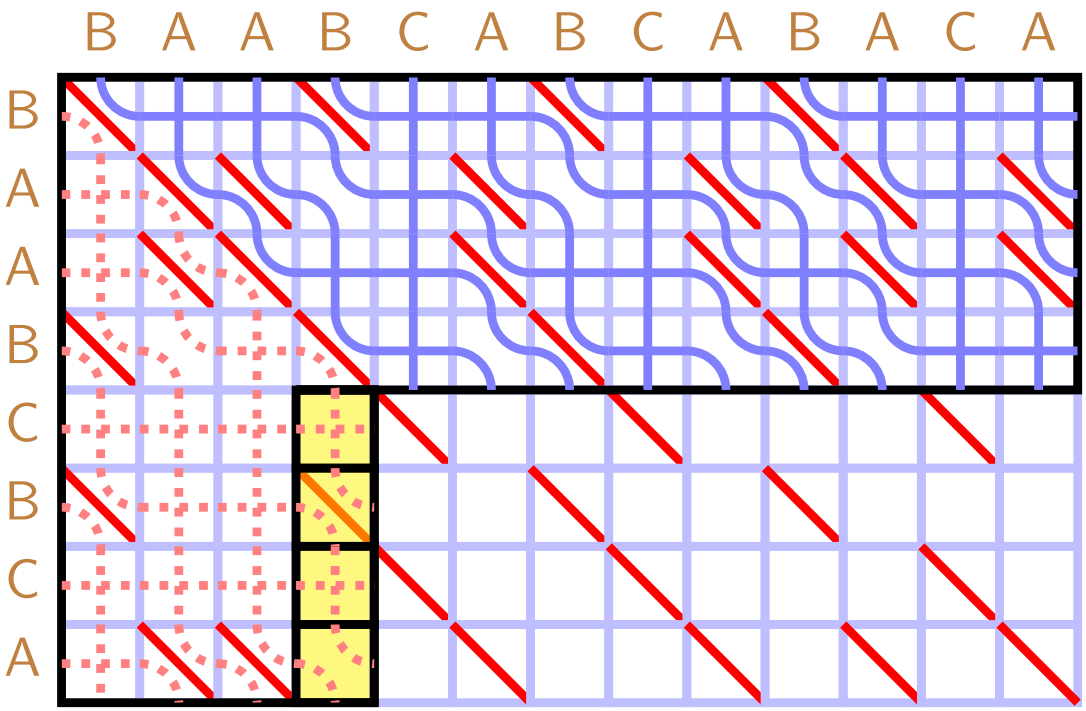
LCS by bit-parallel iterative combing (tiles)



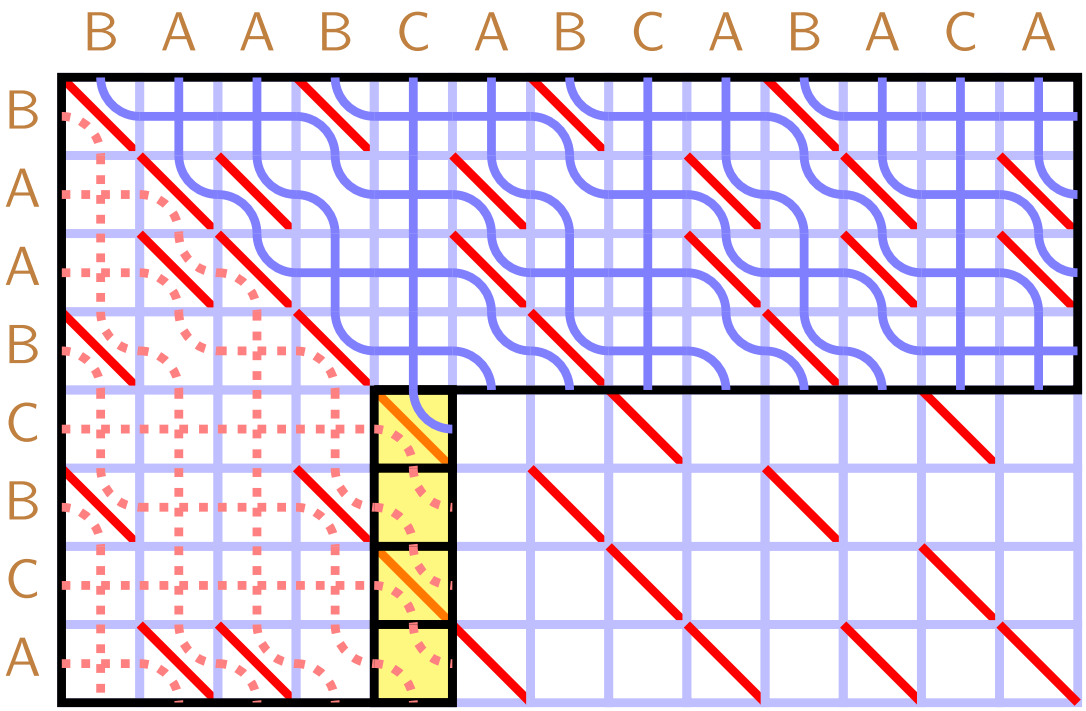
LCS by bit-parallel iterative combing (tiles)



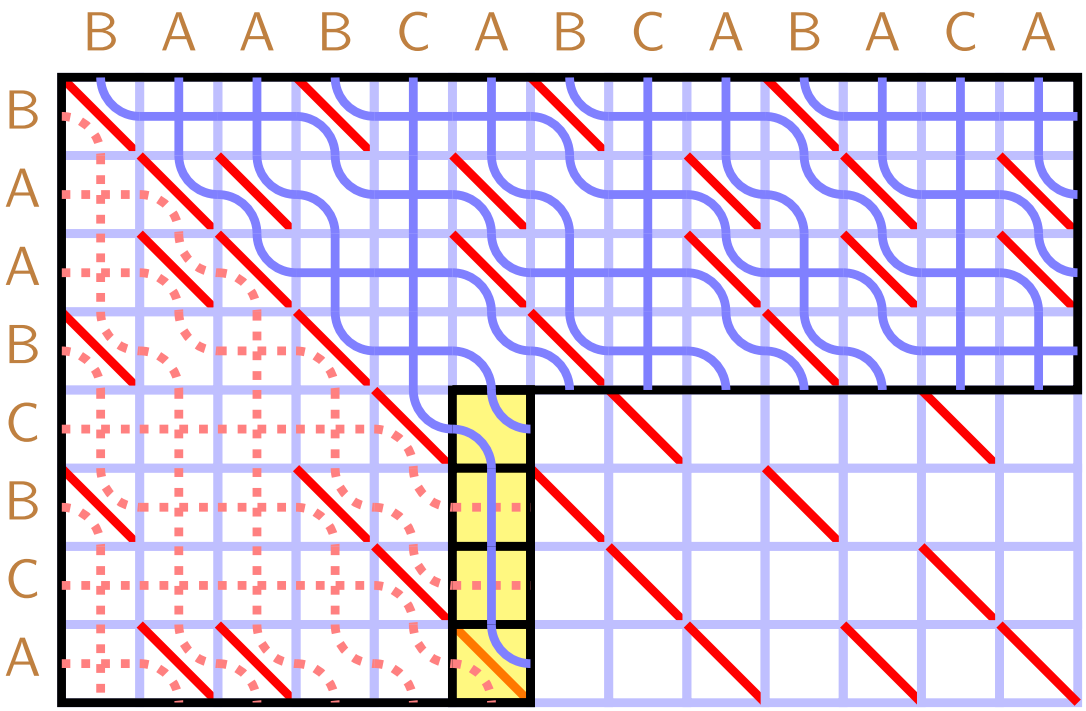
LCS by bit-parallel iterative combing (tiles)



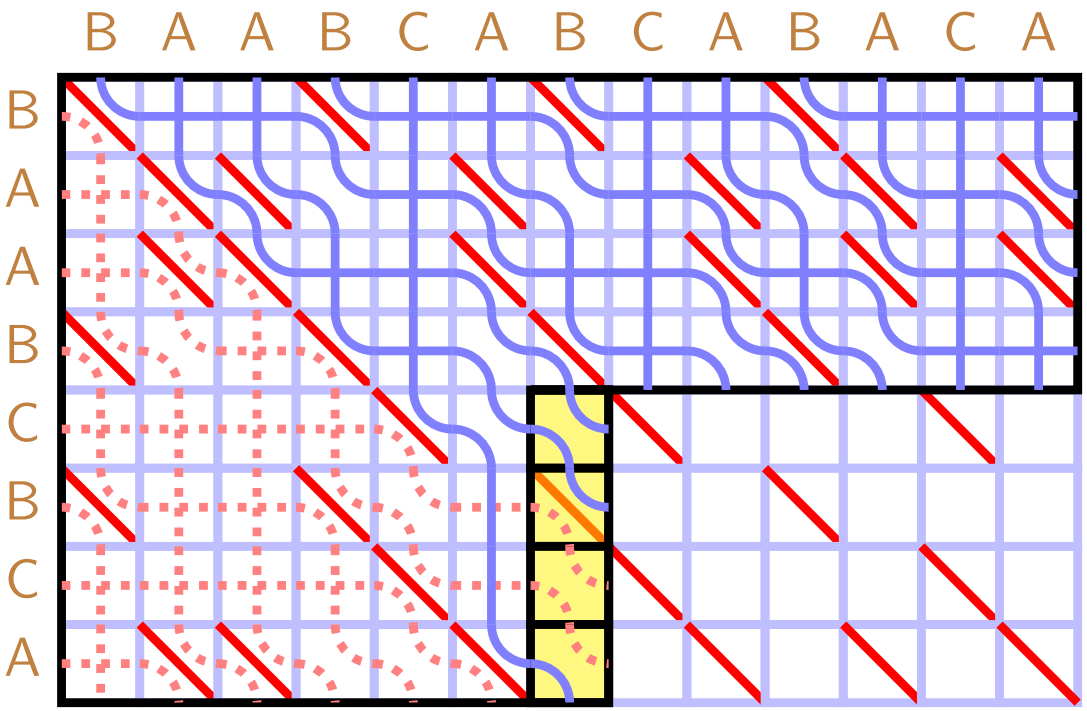
LCS by bit-parallel iterative combing (tiles)



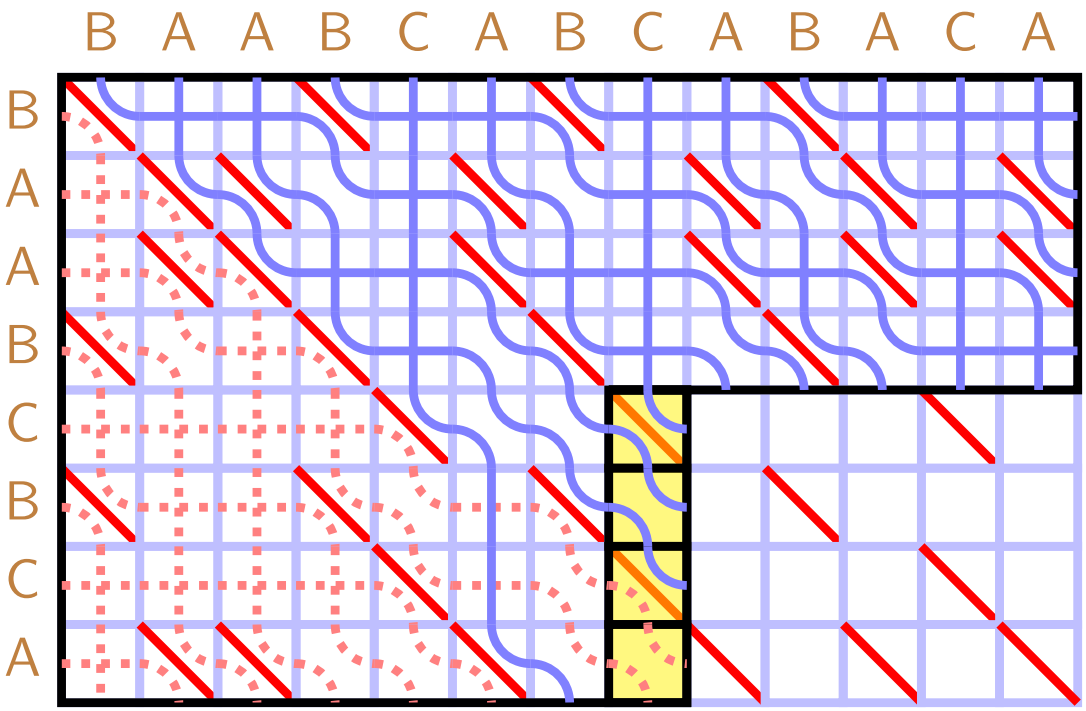
LCS by bit-parallel iterative combing (tiles)



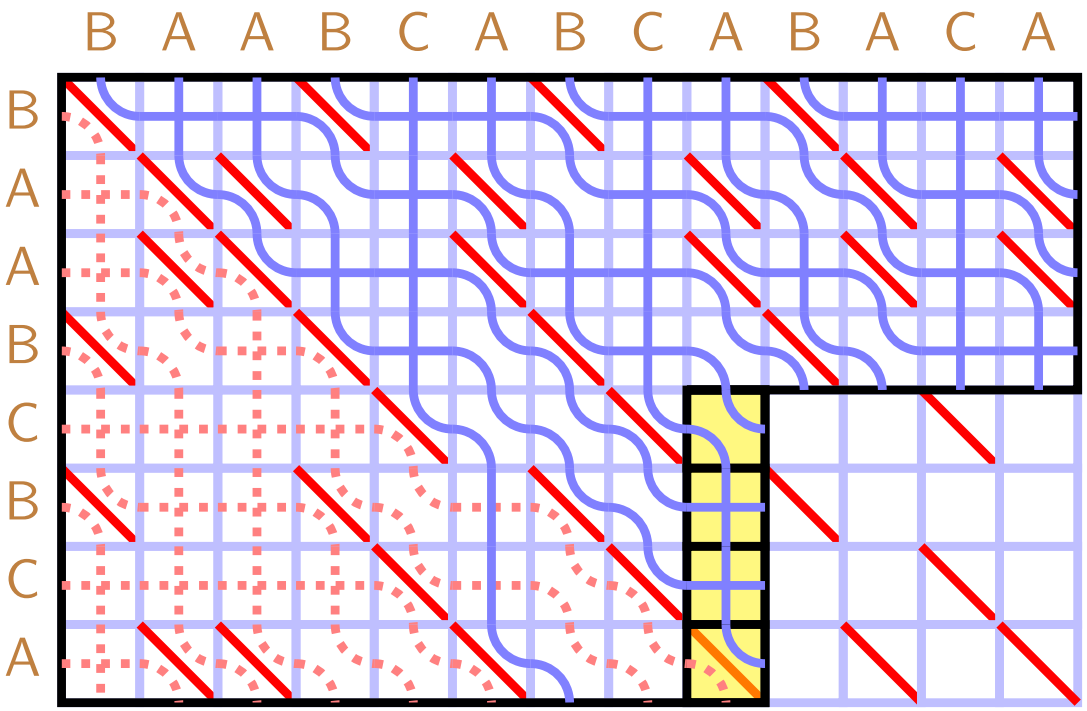
LCS by bit-parallel iterative combing (tiles)



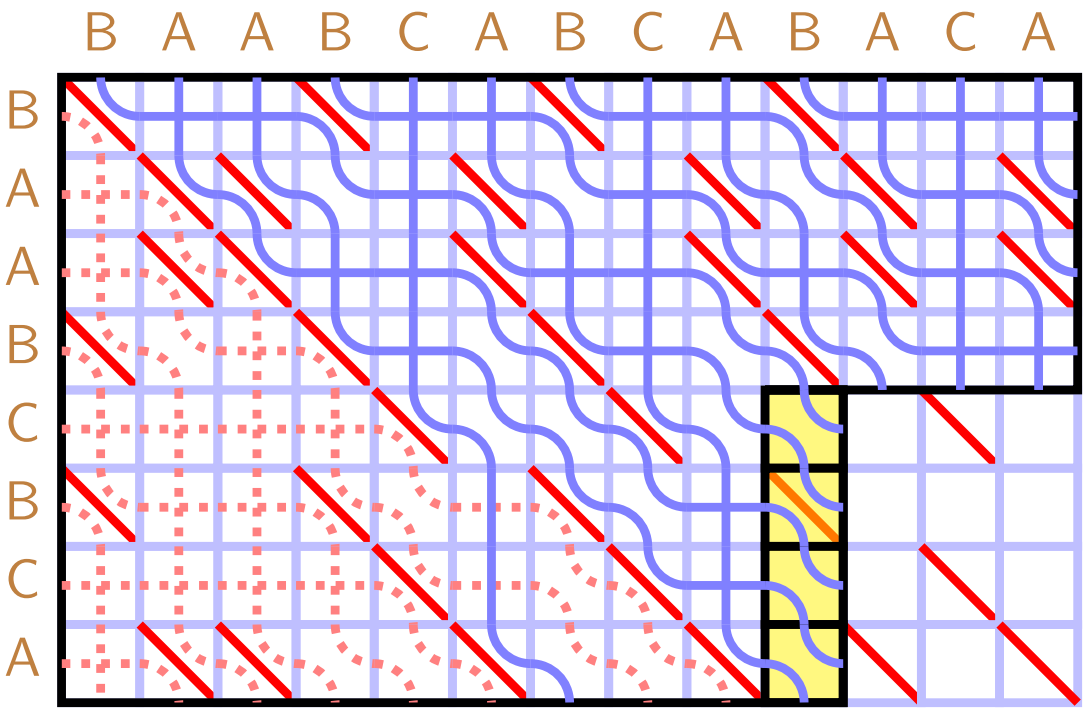
LCS by bit-parallel iterative combing (tiles)



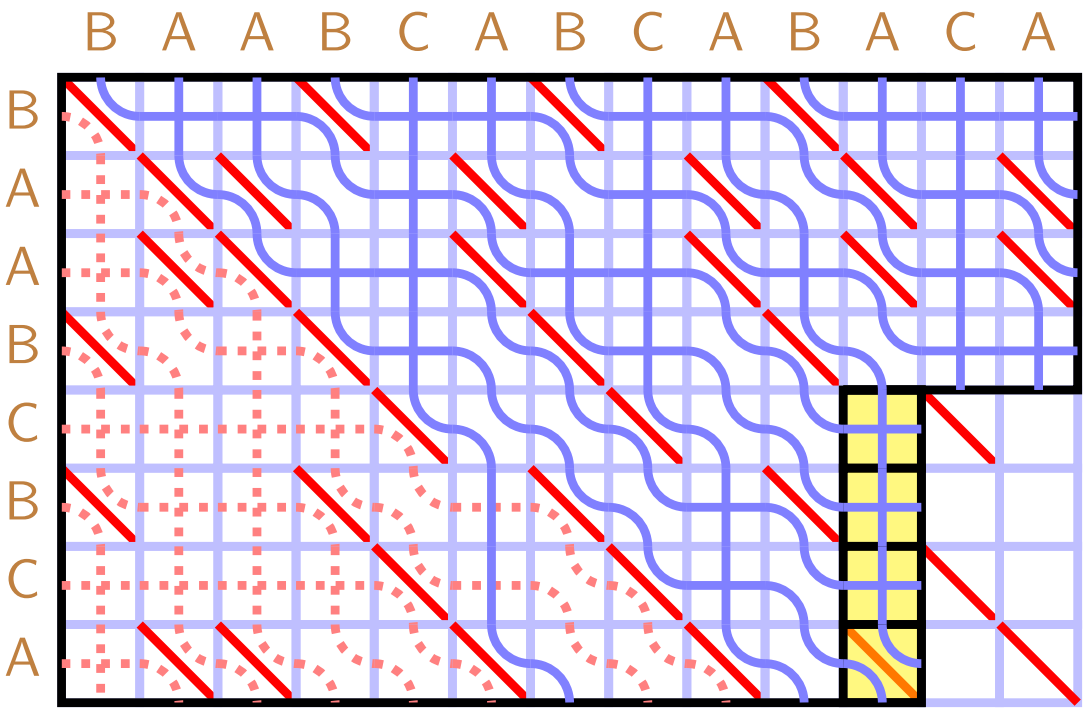
LCS by bit-parallel iterative combing (tiles)



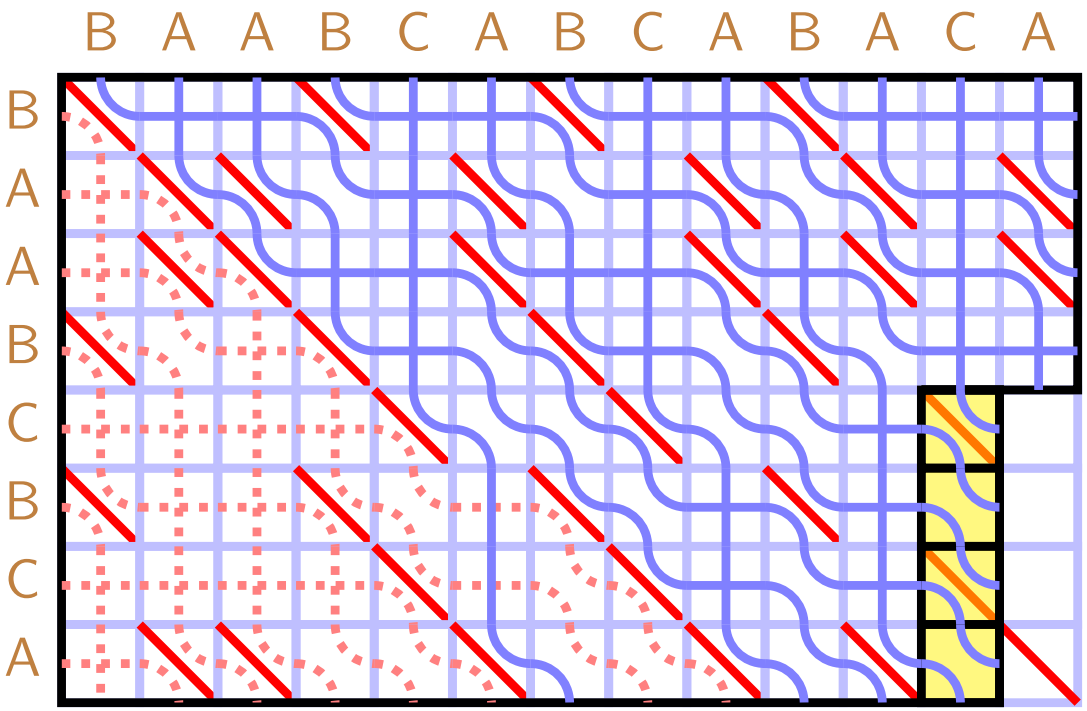
LCS by bit-parallel iterative combing (tiles)



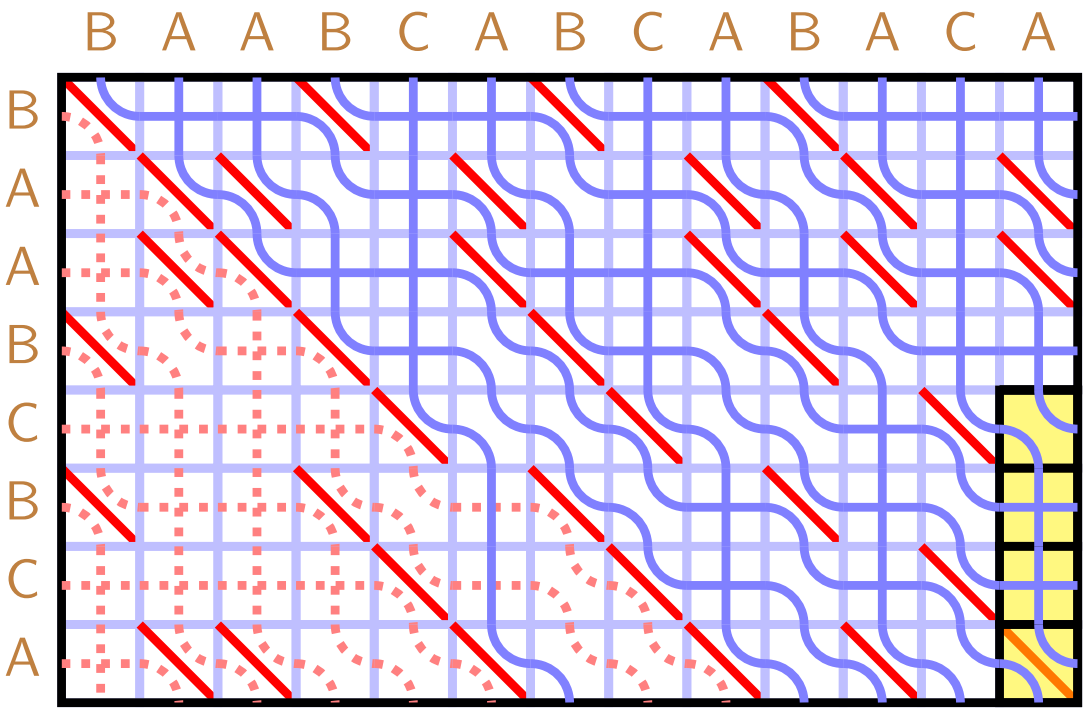
LCS by bit-parallel iterative combing (tiles)



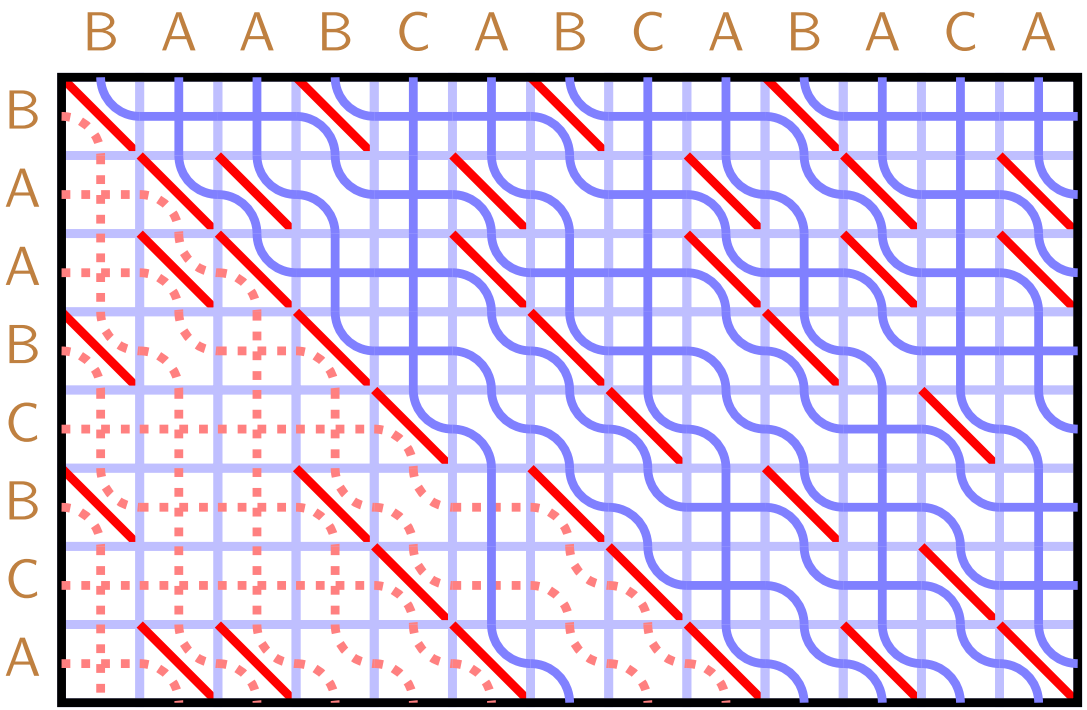
LCS by bit-parallel iterative combing (tiles)



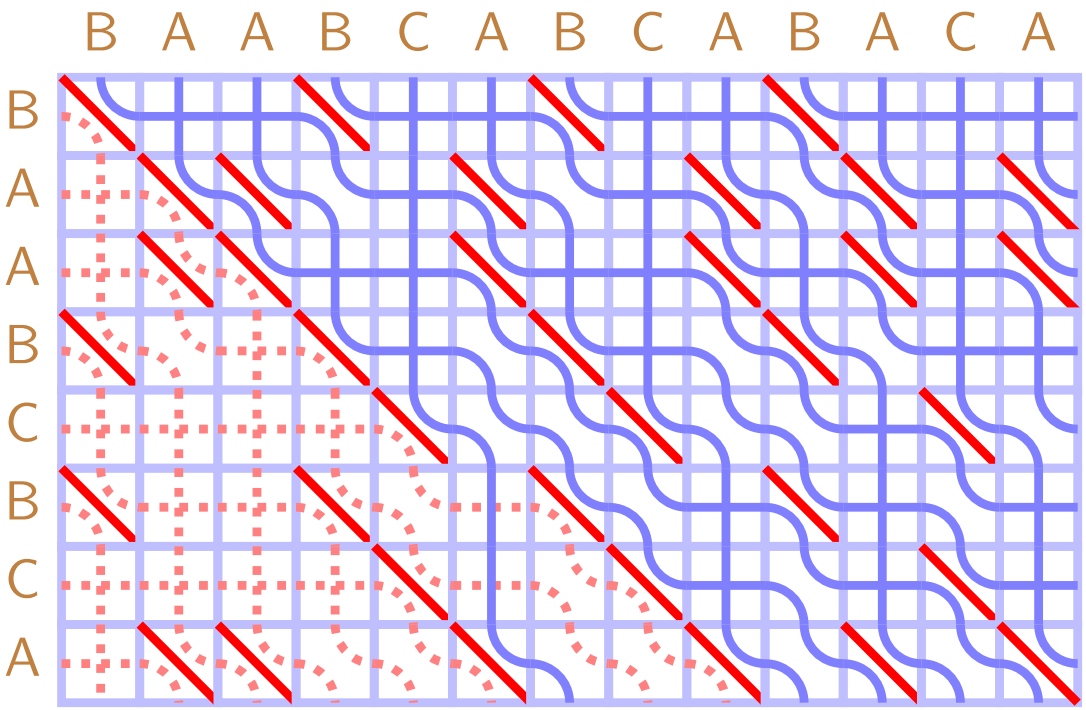
LCS by bit-parallel iterative combing (tiles)



LCS by bit-parallel iterative combing (tiles)



LCS by bit-parallel iterative combing (tiles)



LCS by bit-parallel iterative combing (tiles)

Iterate over cells in vertical (or horizontal) tiles of v cells

Maintain frontier of bits representing strands: v horizontal, one vertical strand into/out of a tile

For each tile:

- get vertical bit from above
- get v horizontal bits from left
- perform combing logic, updating vertical and horizontal bits: vertical bit propagates as integer addition carry

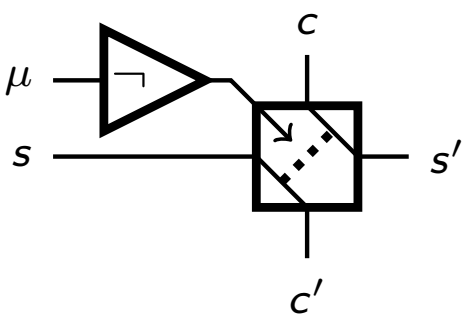
Tile processing: $O(1)$ integer/Boolean operations

Overall time $O\left(\frac{mn}{v}\right)$

LCS by bit-parallel iterative combing (tiles)

μ: match flag $a[i] = b[j]$ s, c: input bits s', c': output bits

Bit combing logic: $(c', s') \leftarrow \texttt{if } \mu \texttt{ then } (s, c) \texttt{ else } (s \wedge c, s \vee c)$

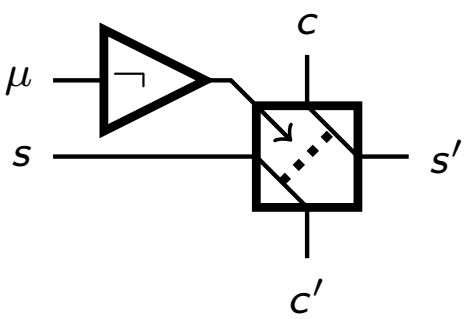


s	0	1	0	1	0	1	0	1
c	0	0	1	1	0	0	1	1
μ	0	0	0	0	1	1	1	1
s'	0	1	1	**1**	0	0	1	1
c'	0	0	0	1	0	1	0	1

LCS by bit-parallel iterative combing (tiles)

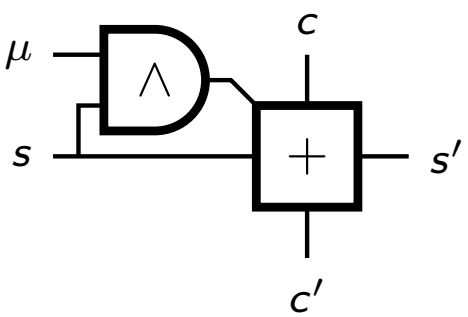
μ: match flag $a[i] = b[j]$ s, c: input bits s', c': output bits

Bit combing logic: $(c', s') \leftarrow \texttt{if } \mu \texttt{ then } (s, c) \texttt{ else } (s \wedge c, s \vee c)$



s	0	1	0	1	0	1	0	1
c	0	0	1	1	0	0	1	1
μ	0	0	0	0	1	1	1	1
s'	0	1	1	**1**	0	0	1	1
c'	0	0	0	1	0	1	0	1

Fast bit combing: $2c' + s' \leftarrow s + (s \wedge \mu) + c$, then single-case correction



s	0	1	0	1	0	1	0	1
c	0	0	1	1	0	0	1	1
μ	0	0	0	0	1	1	1	1
s'	0	1	1	**0**	0	0	1	1
c'	0	0	0	1	0	1	0	1

Parallel and dynamic string comparison

Bit-parallel LCS

LCS by bit-parallel iterative combing (tiles, contd.)

M: match flag vector M, $\neg M$ precomputed

w: machine word size

Fast bit-parallel combing: 4 bit-vector operations

$2^w \cdot c' + S' \leftarrow (S + (S \wedge M) + c) \vee (S \wedge \neg M)$ [Crochemore+: 2001]

Parallel and dynamic string comparison

Bit-parallel LCS

LCS by bit-parallel iterative combing (tiles, contd.)

M: match flag vector M, $\neg M$ precomputed

w: machine word size

Fast bit-parallel combing: 4 bit-vector operations

$2^w \cdot c' + S' \leftarrow (S + (S \wedge M) + c) \vee (S \wedge \neg M)$ [Crochemore+: 2001]

$2^w \cdot c' + S' \leftarrow (S + (S \wedge M) + c) \vee (S - (S \wedge M))$ [Hyyrö: 2004]

Parallel and dynamic string comparison

Bit-parallel LCS

LCS by bit-parallel iterative combing (tiles, contd.)

M: match flag vector M, $\neg M$ precomputed

w: machine word size

Fast bit-parallel combing: 4 bit-vector operations

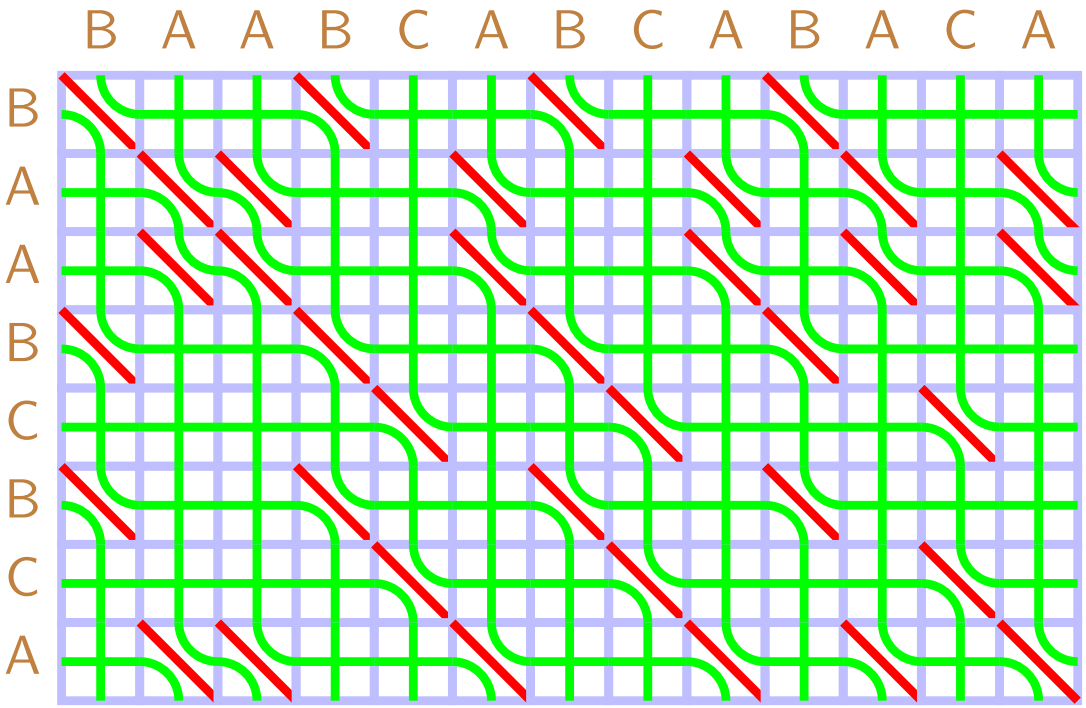
$2^w \cdot c' + S' \leftarrow (S + (S \wedge M) + c) \vee (S \wedge \neg M)$ [Crochemore+: 2001]

$2^w \cdot c' + S' \leftarrow (S + (S \wedge M) + c) \vee (S - (S \wedge M))$ [Hyyrö: 2004]

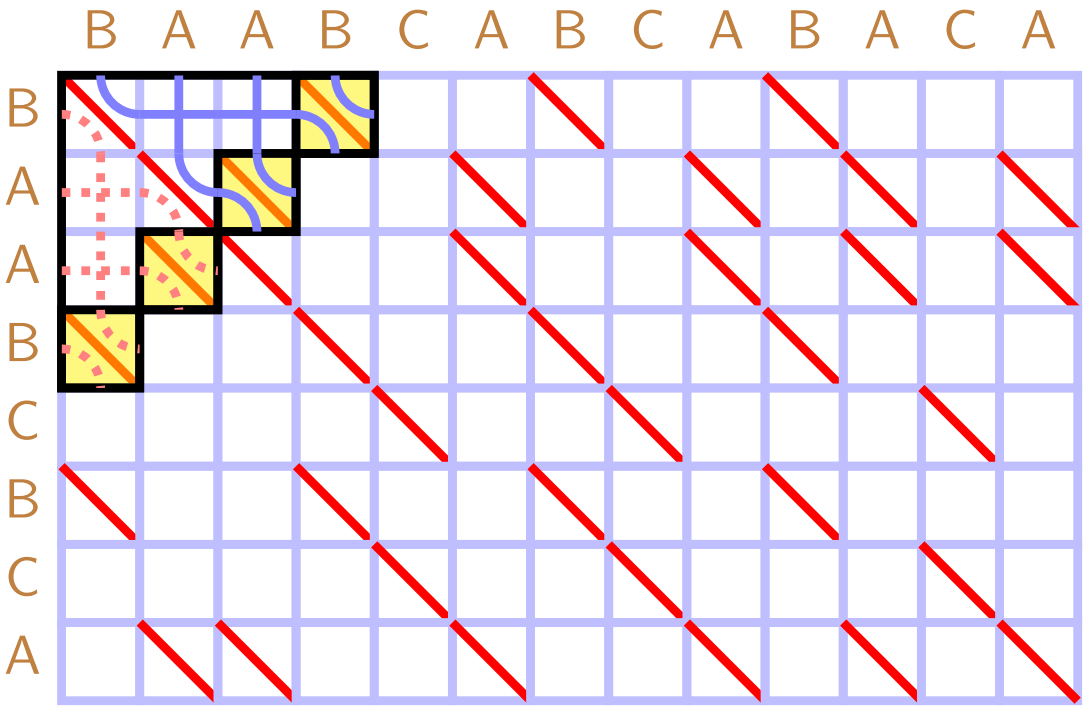
. . . in 3 bit-vector ops: impossible under reasonable assumptions [Hyyrö: 2017]

- vertical tiles; single vector of size v
- arbitrary binary Boolean functions; shifts
- standard arithmetic, including $A + B + carry$

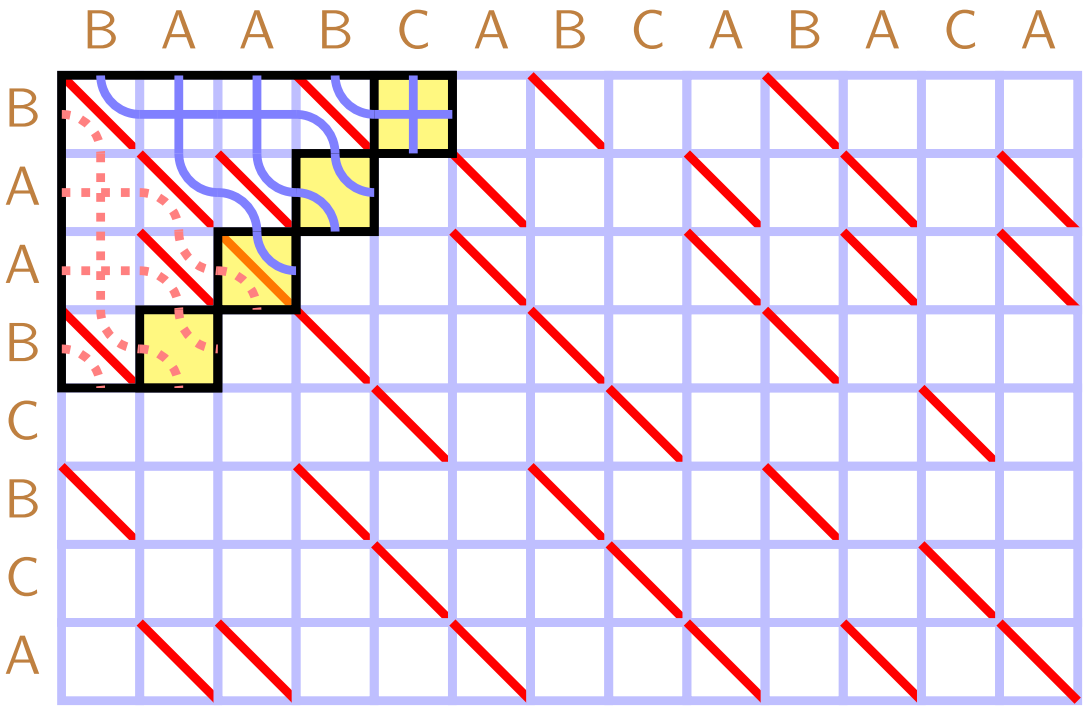
LCS by bit-parallel iterative combing (anti-diagonals)



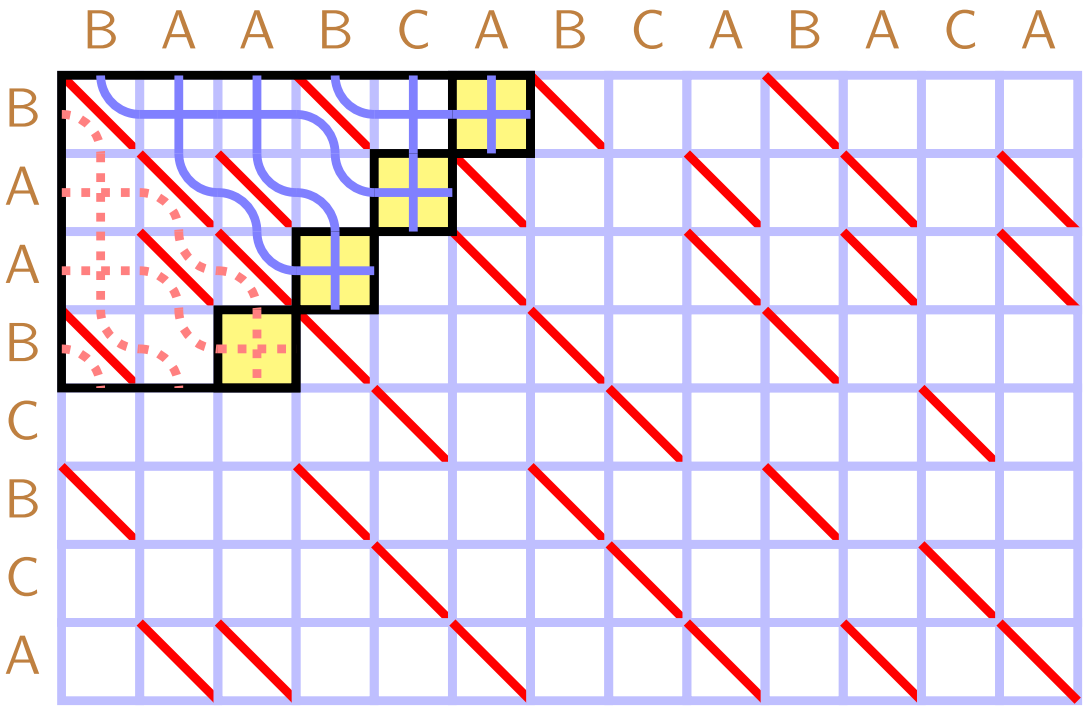
LCS by bit-parallel iterative combing (anti-diagonals)



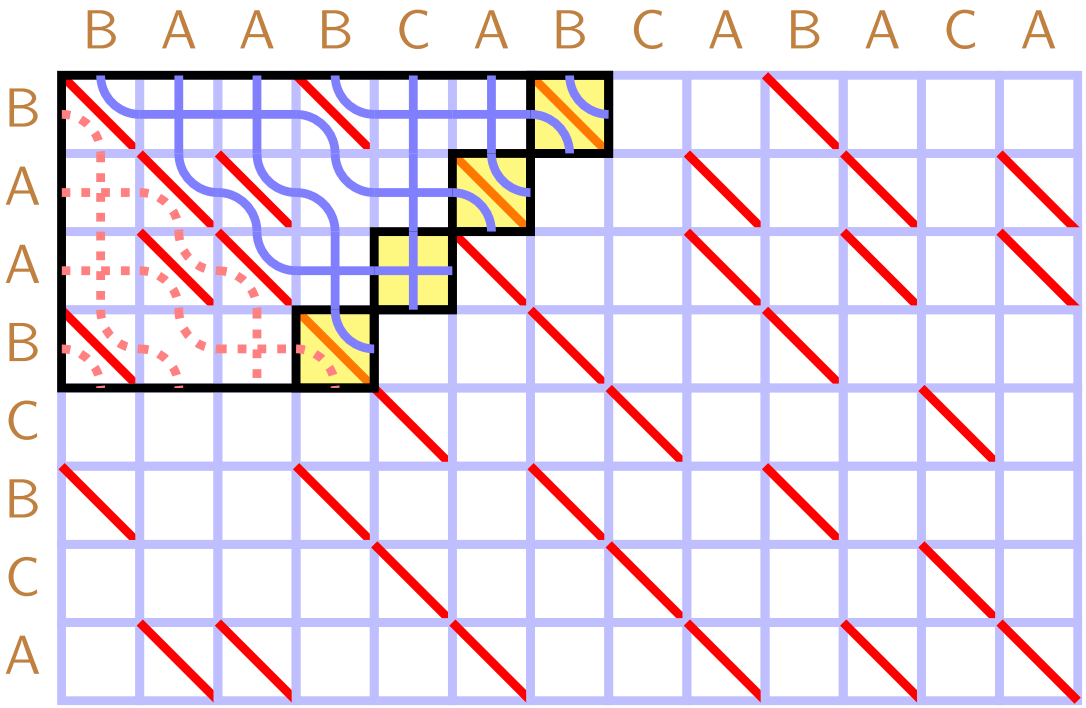
LCS by bit-parallel iterative combing (anti-diagonals)



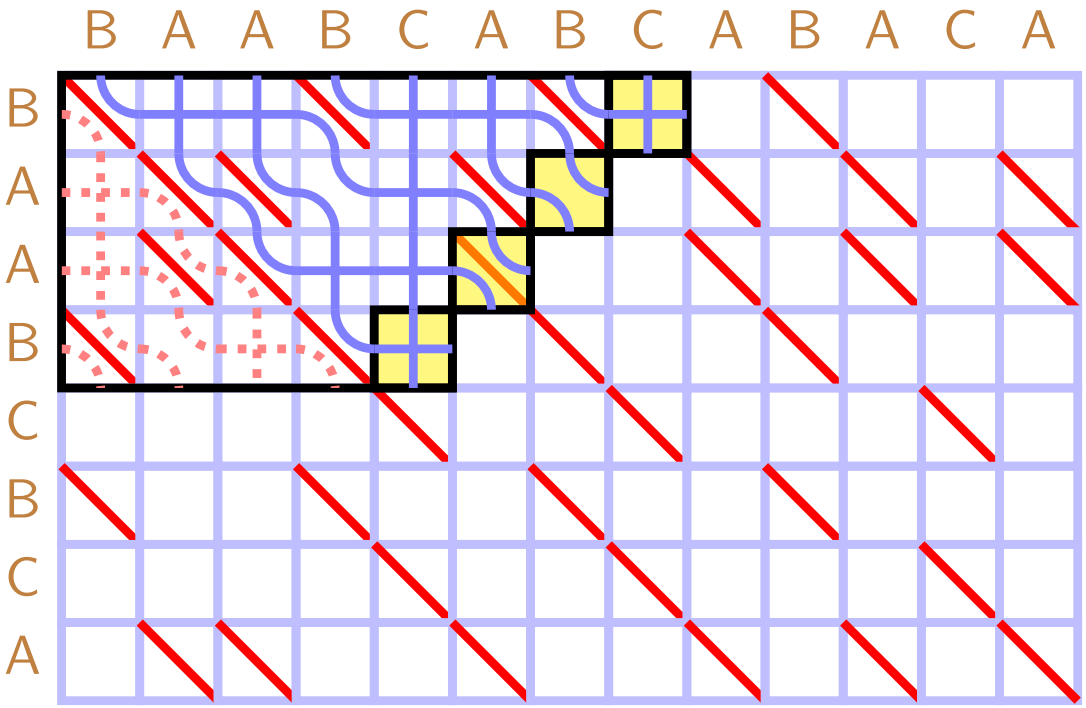
LCS by bit-parallel iterative combing (anti-diagonals)



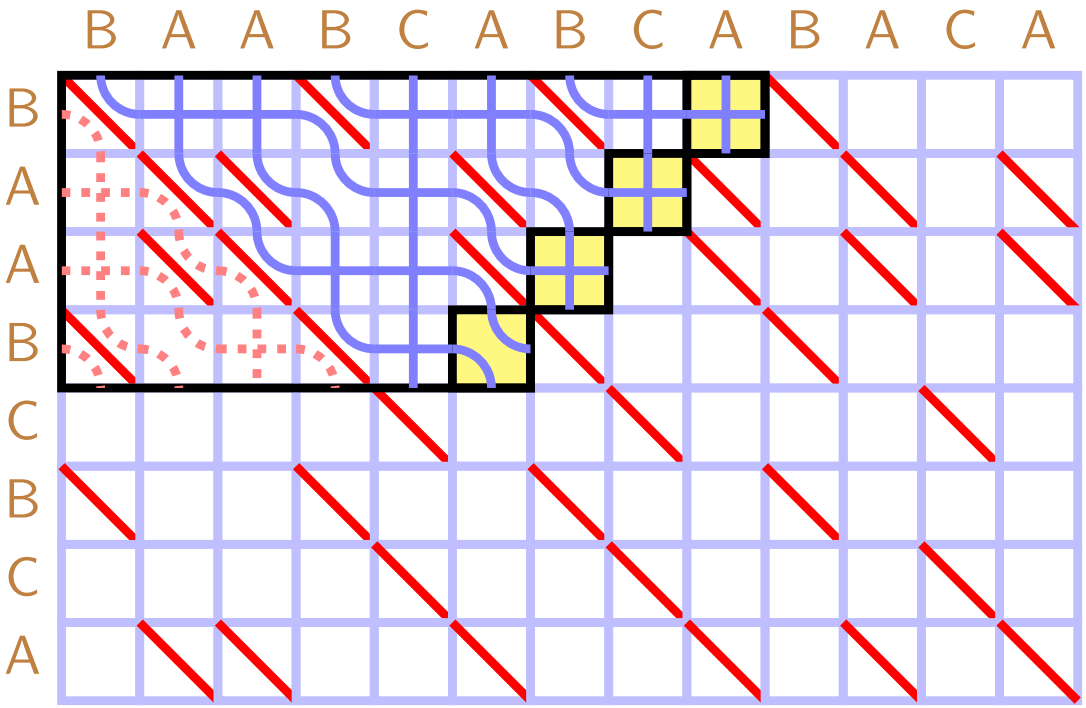
LCS by bit-parallel iterative combing (anti-diagonals)



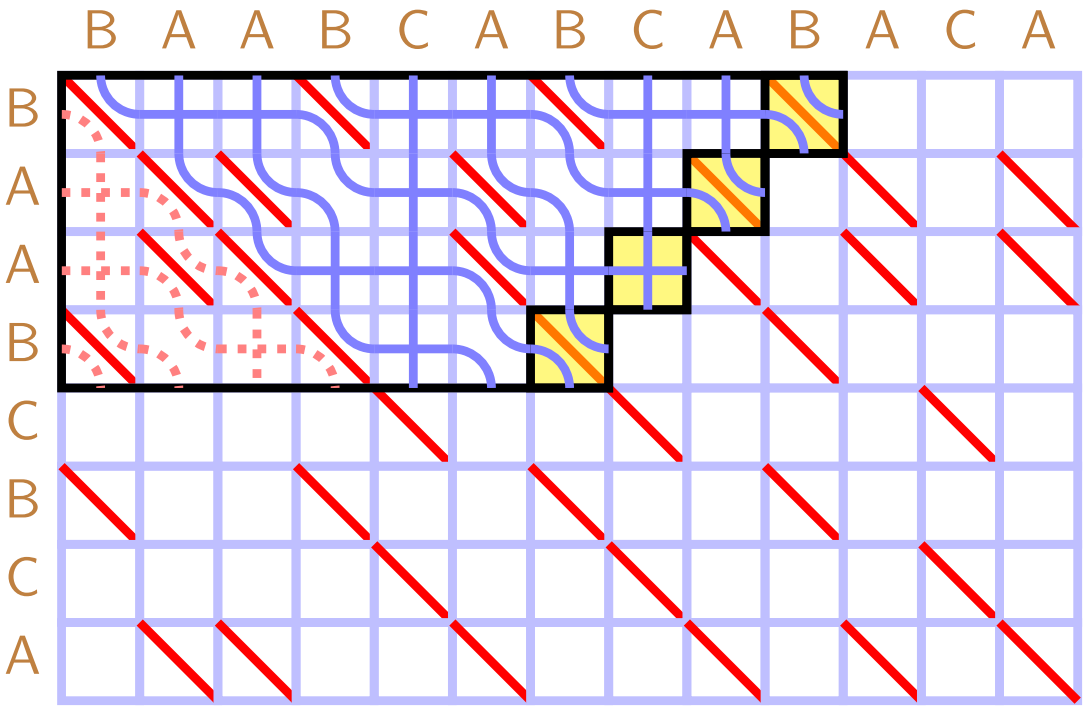
LCS by bit-parallel iterative combing (anti-diagonals)



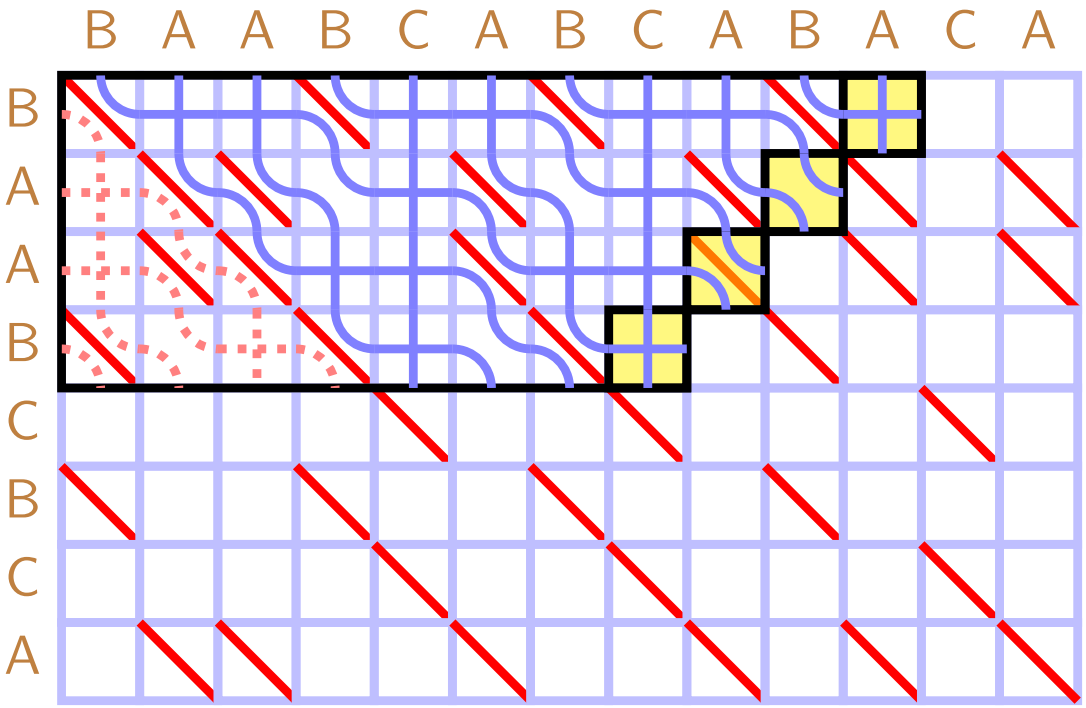
LCS by bit-parallel iterative combing (anti-diagonals)



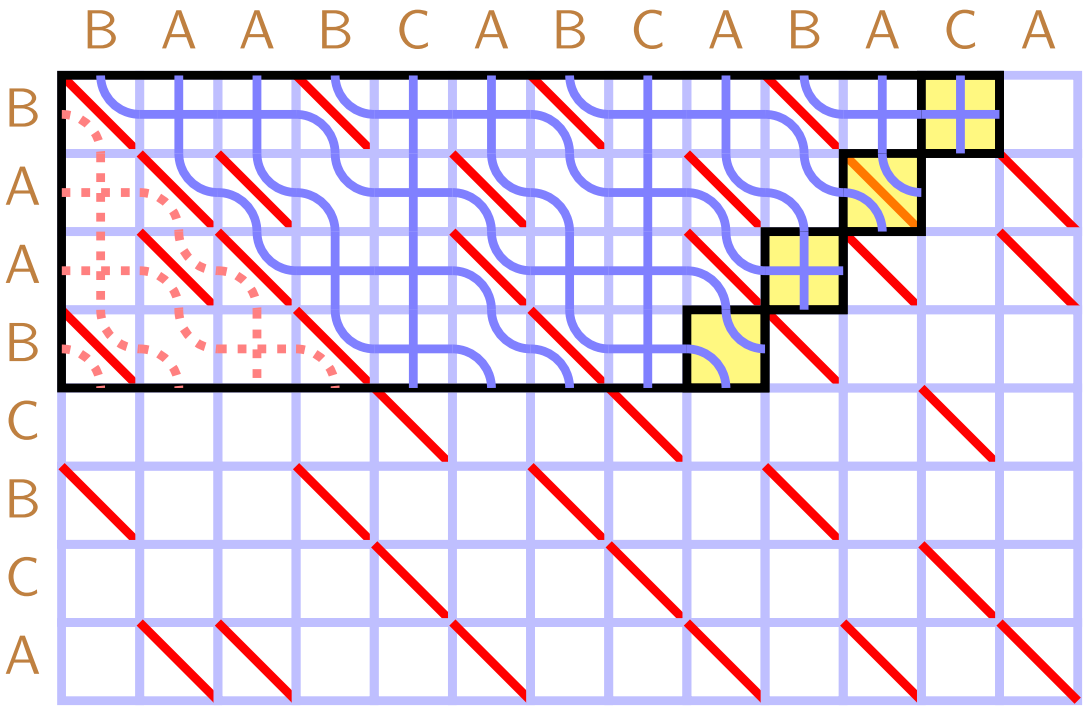
LCS by bit-parallel iterative combing (anti-diagonals)



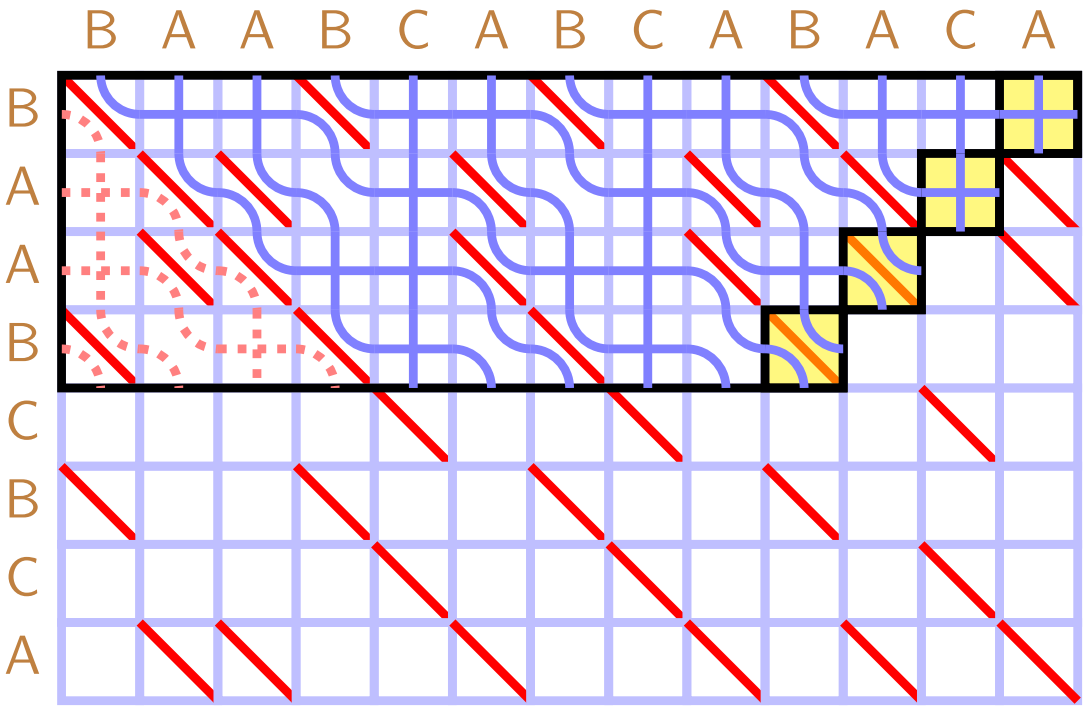
LCS by bit-parallel iterative combing (anti-diagonals)



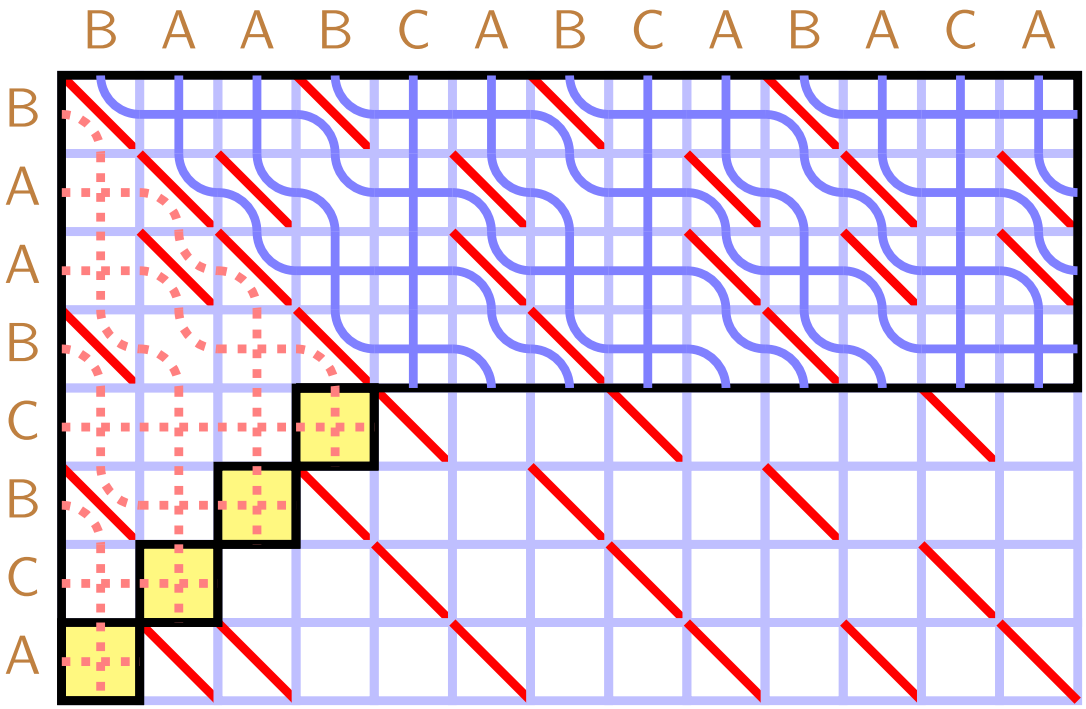
LCS by bit-parallel iterative combing (anti-diagonals)



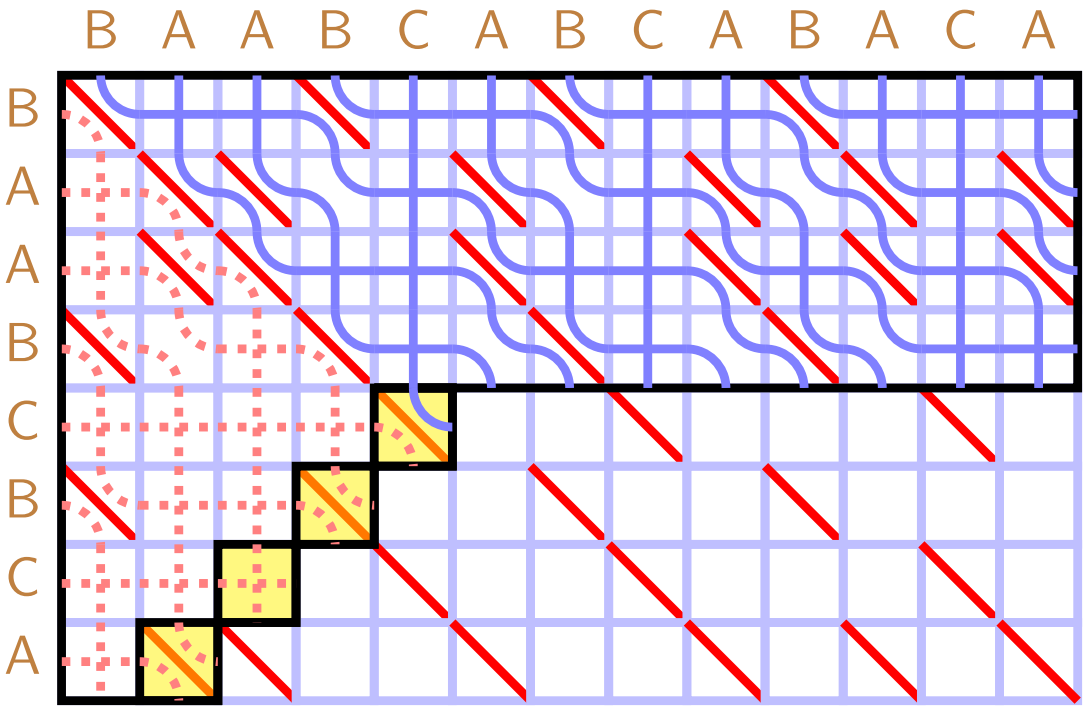
LCS by bit-parallel iterative combing (anti-diagonals)



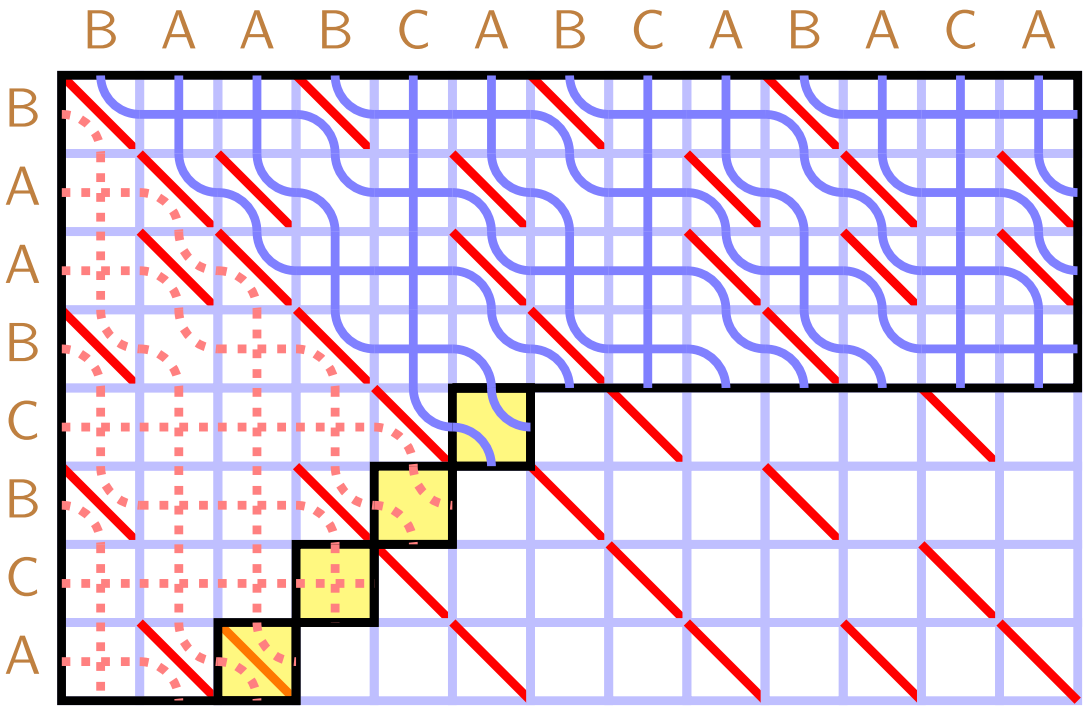
LCS by bit-parallel iterative combing (anti-diagonals)



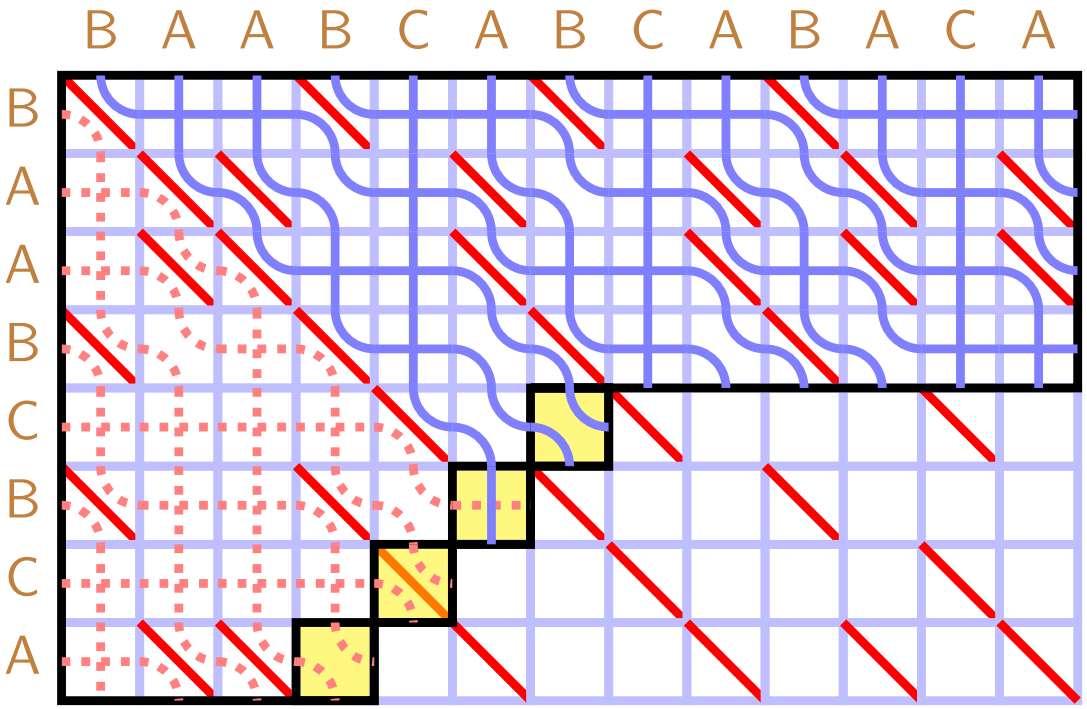
LCS by bit-parallel iterative combing (anti-diagonals)



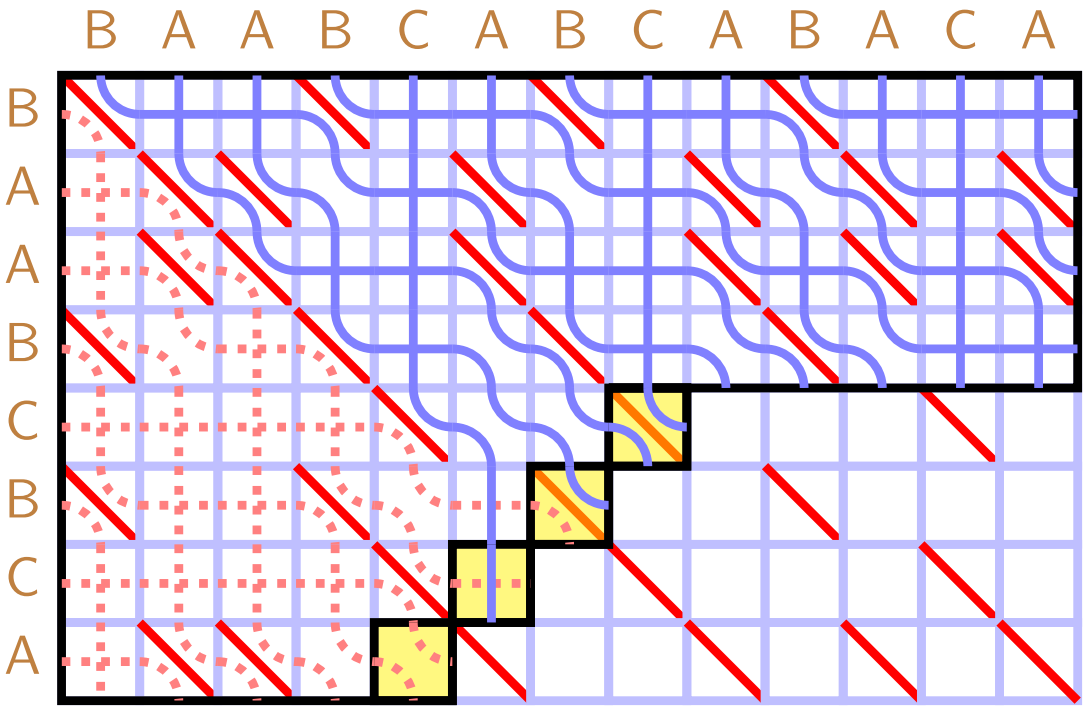
LCS by bit-parallel iterative combing (anti-diagonals)



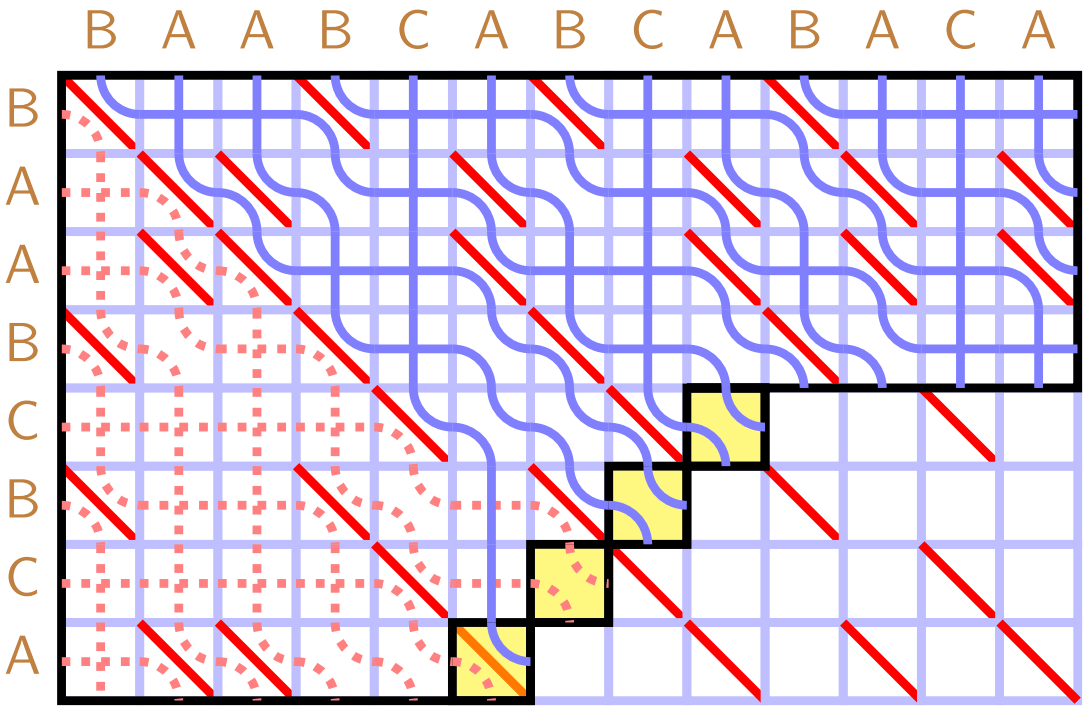
LCS by bit-parallel iterative combing (anti-diagonals)



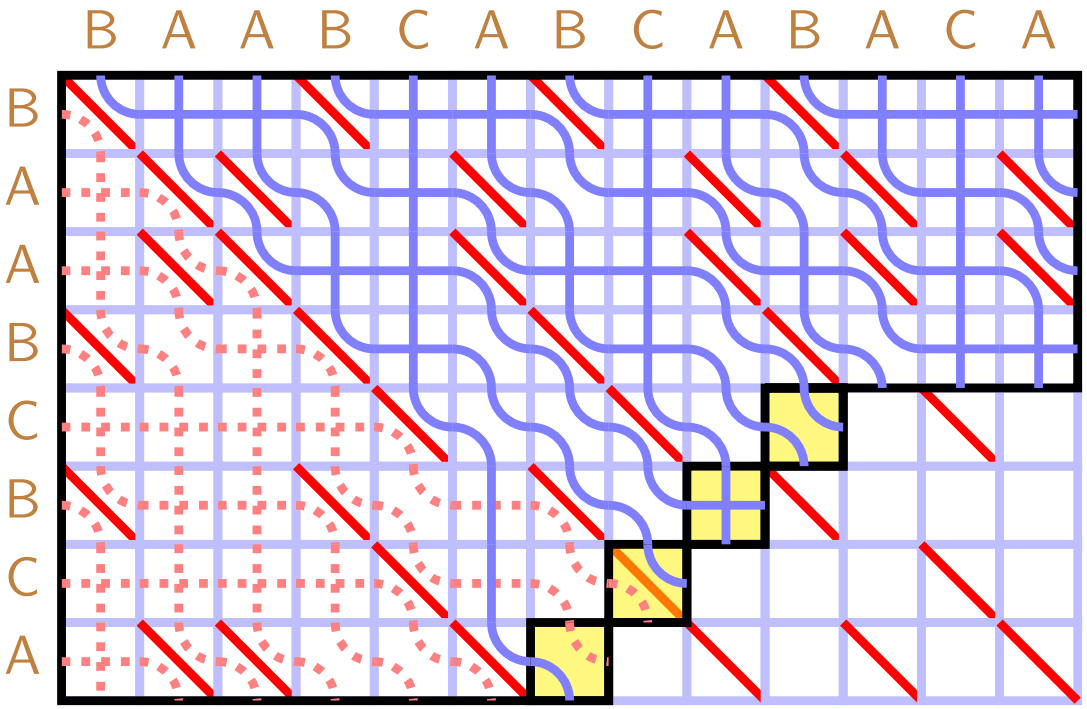
LCS by bit-parallel iterative combing (anti-diagonals)



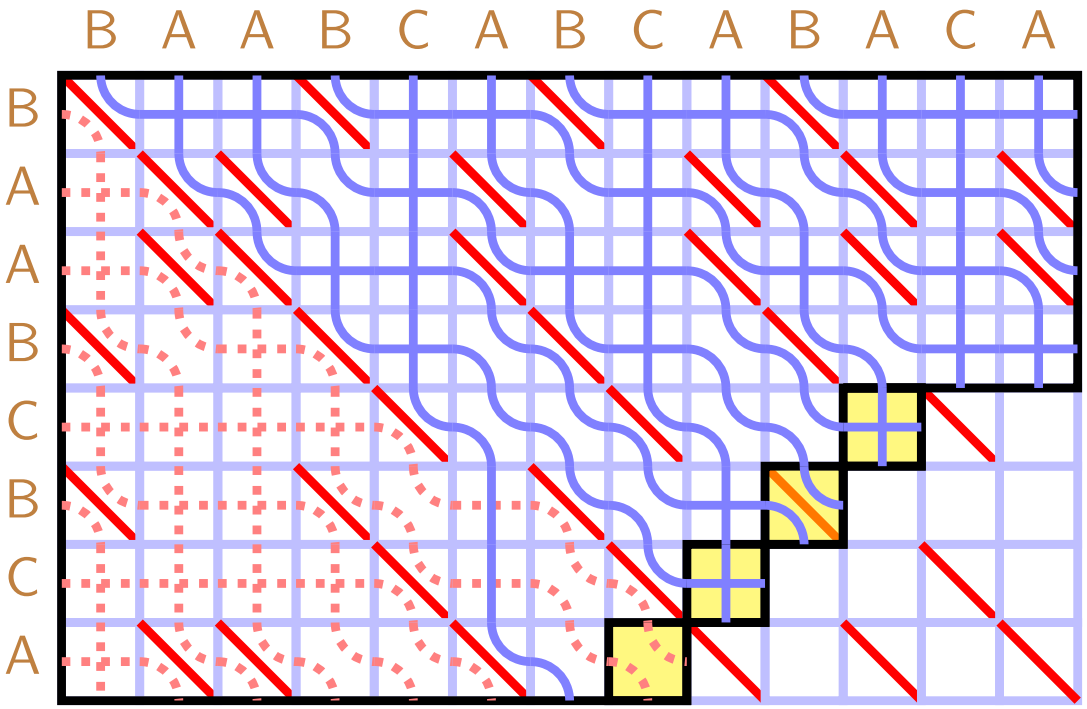
LCS by bit-parallel iterative combing (anti-diagonals)



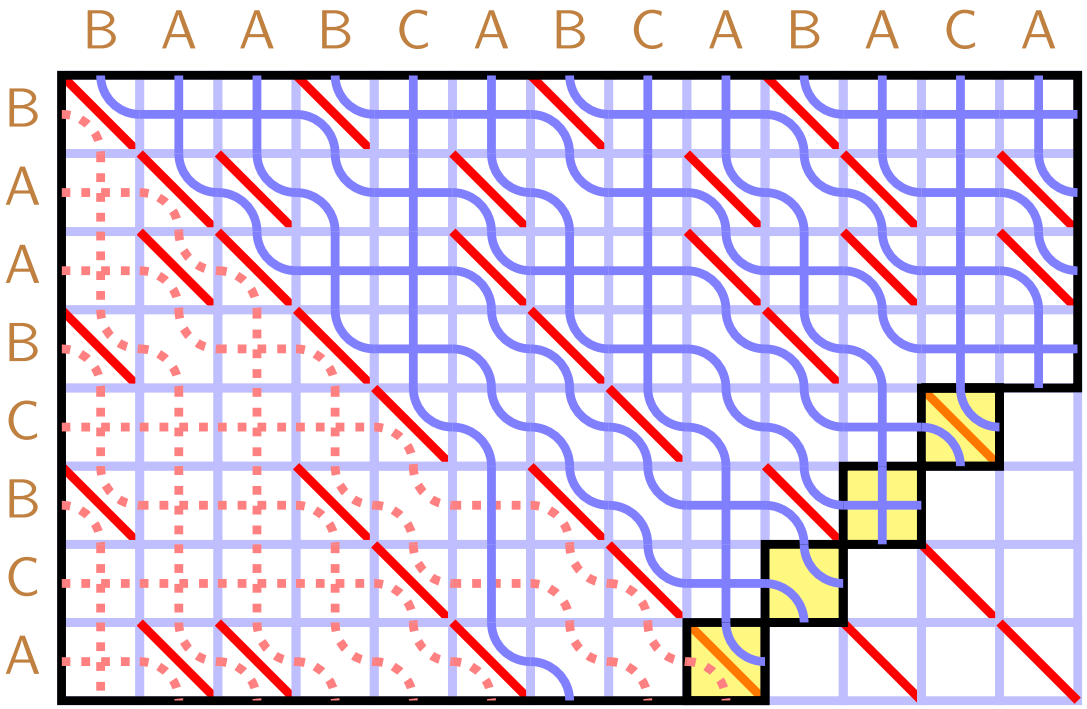
LCS by bit-parallel iterative combing (anti-diagonals)



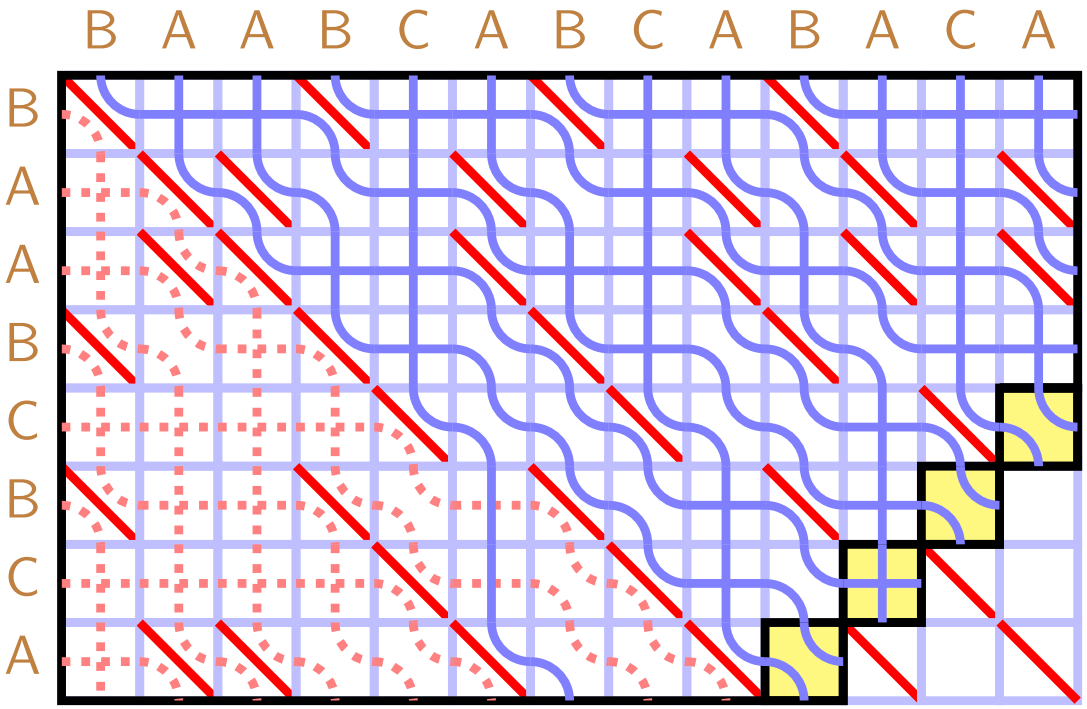
LCS by bit-parallel iterative combing (anti-diagonals)



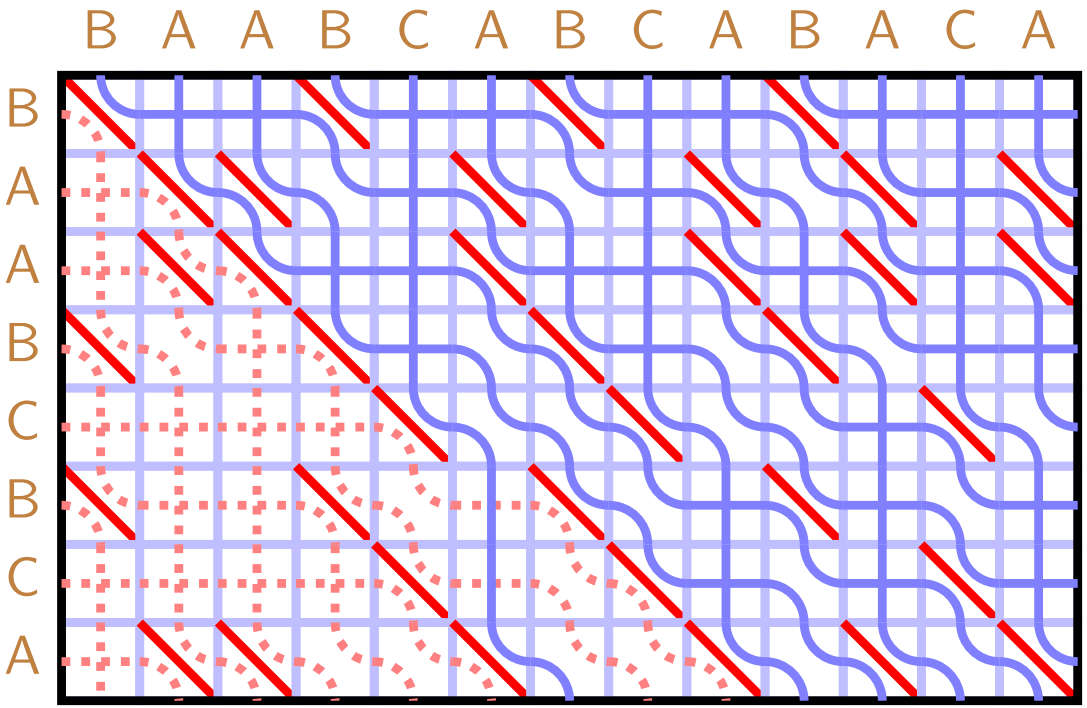
LCS by bit-parallel iterative combing (anti-diagonals)



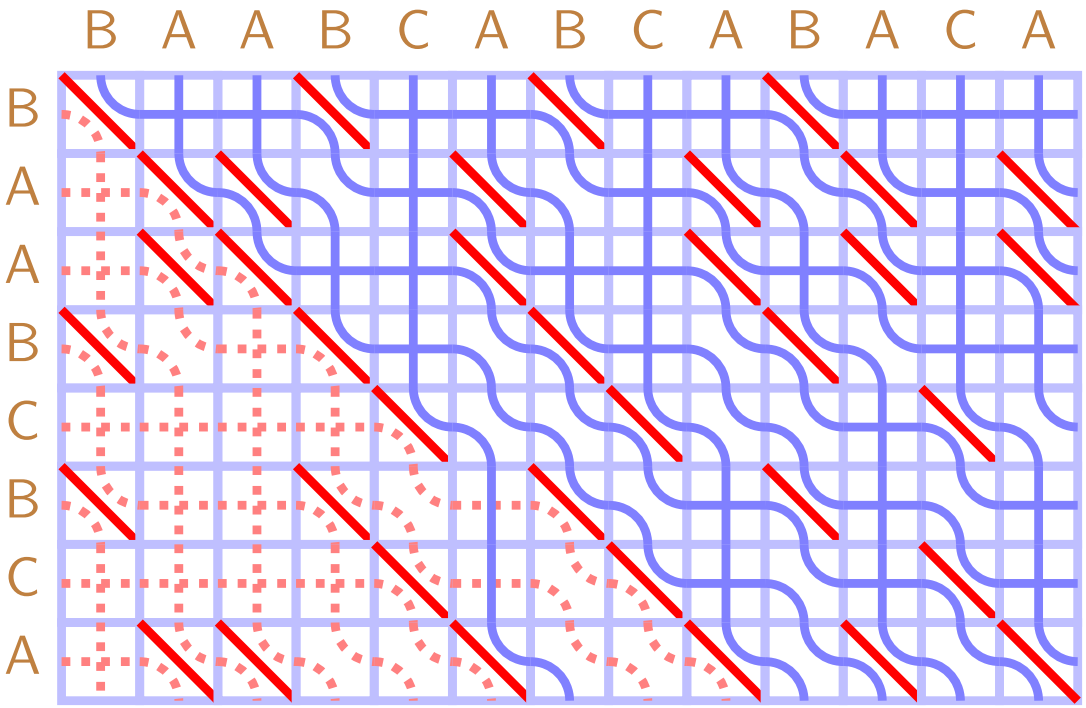
LCS by bit-parallel iterative combing (anti-diagonals)



LCS by bit-parallel iterative combing (anti-diagonals)



LCS by bit-parallel iterative combing (anti-diagonals)



Efficient Parallel Algorithms for String Comparison

Nikita Mishin*
Saint Petersburg State University
Saint Petersburg, Russia
mishinnikitam@protonmail.com

Daniil Berezun*
Saint Petersburg State University
Saint Petersburg, Russia
d.berezun@2009.spbu.ru

Alexander Tiskin
Saint Petersburg State University
Saint Petersburg, Russia
a.tiskin@spbu.ru

ABSTRACT

The longest common subsequence (LCS) problem on a pair of strings is a classical problem in string algorithms. Its extension, the semi-local LCS problem, provides a more detailed comparison of the input strings, without any increase in asymptotic running time. Several semi-local LCS algorithms have been proposed previously; however, to the best of our knowledge, none have yet been implemented. In this paper, we explore a new hybrid approach to the semi-local LCS problem. We also propose a novel bit-parallel LCS algorithm. In the experimental part of the paper, we present an implementation of several existing and new parallel LCS algorithms and evaluate their performance.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; *Algorithm design techniques*; *Dynamic programming*; *Divide and conquer*.

combing) or recursively (*recursive combing*) [23]. However, to our knowledge, none of these algorithms have yet been implemented. In this paper, we explore a hybrid approach, combining iterative and recursive combing. We also propose a novel bit-parallel LCS algorithm, free of integer arithmetic and the associated carry propagation delays that are typical of existing bit-parallel LCS algorithms. Furthermore, we present an implementation of several LCS algorithms, including recursive and iterative combing, as well as their parallel versions using thread-level parallelism, and intra-processor SIMD subword and bit parallelism, with a number of optimizations.

For experimental evaluation of the presented algorithms we use two types of input: randomly generated strings and a real-life dataset of virus genomes. Our experiments show that the running times of our implementations of semi-local LCS algorithms correspond to their theoretical estimations with no extra overheads and are comparable to an implementation of standard LCS. Thus, the

LCS by bit-parallel iterative combing (anti-diagonals, contd.)

Intel's AVX–512 instruction set (since 2017): 32× 512-bit registers

Each register accessible as vector of

- $64 \times$ 8-bit bytes
- $32 \times$ 16-bit words
- $16 \times$ 32-bit dwords
- $8 \times$ 64-bit qwords

LCS by bit-parallel iterative combing (anti-diagonals, contd.)

AVX-512 ternary Boolean logic: `VPTERNLOGD/VPTERNLOGQ`

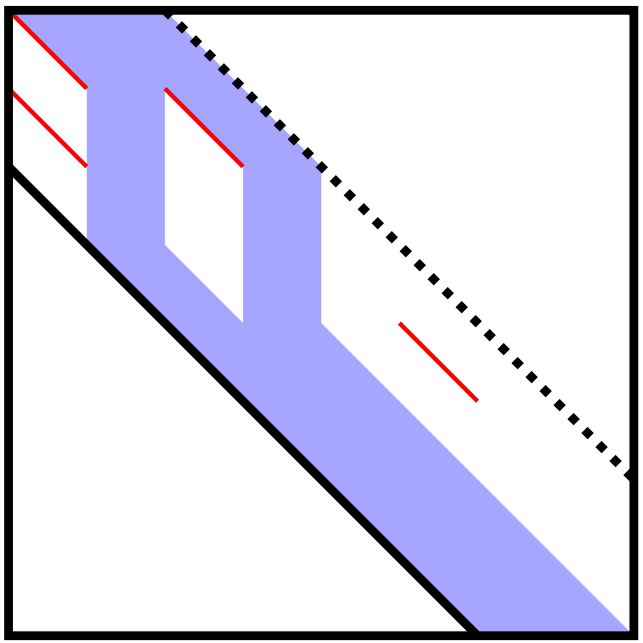
- arbitrary bitwise Boolean function of three bit-vector arguments
- in-place, replacing first argument

> *Takes three bit vectors of 512-bit length as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The [immediate operand] byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the first operand. (Intel® 64 and IA-32 Architectures Software Developer's Manual, May 2019)*

Assuming ternary Boolean functions, combing logic runs in 2 bit-vector ops

High-similarity bit-parallel LCS

$\kappa = d_{LCS}(a, b)$ Assume κ odd, $m = n$



Waterfall within diagonal band of width $\kappa + 1$: time $O\left(\frac{n\kappa}{v}\right)$

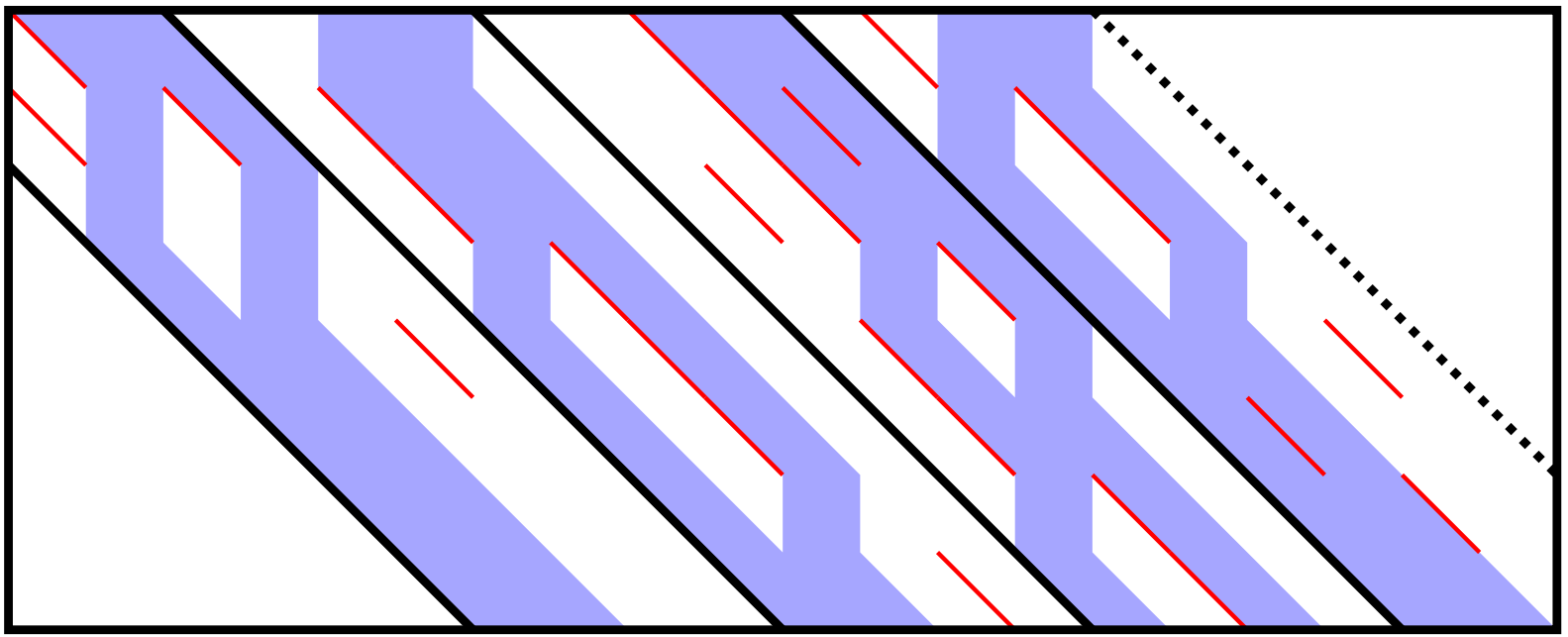
Band waterfall supported from below by separator matches

Parallel and dynamic string comparison

Bit-parallel LCS

High-similarity bit-parallel multi-string LCS: a vs $b_0, \ldots, b_{r-1}$

$\kappa_i = d_{LCS}(a, b_i) \leq \kappa \qquad 0 \leq i < r$



Waterfalls within r diagonal bands of width $\kappa + 1$: time $O\left(\frac{nr\kappa}{v}\right)$

Each band's waterfall supported from below by separator matches

Vector-parallel computation:

- integer vectors of size v
- single instruction, multiple data

Vector-parallel semi-local LCS: running time

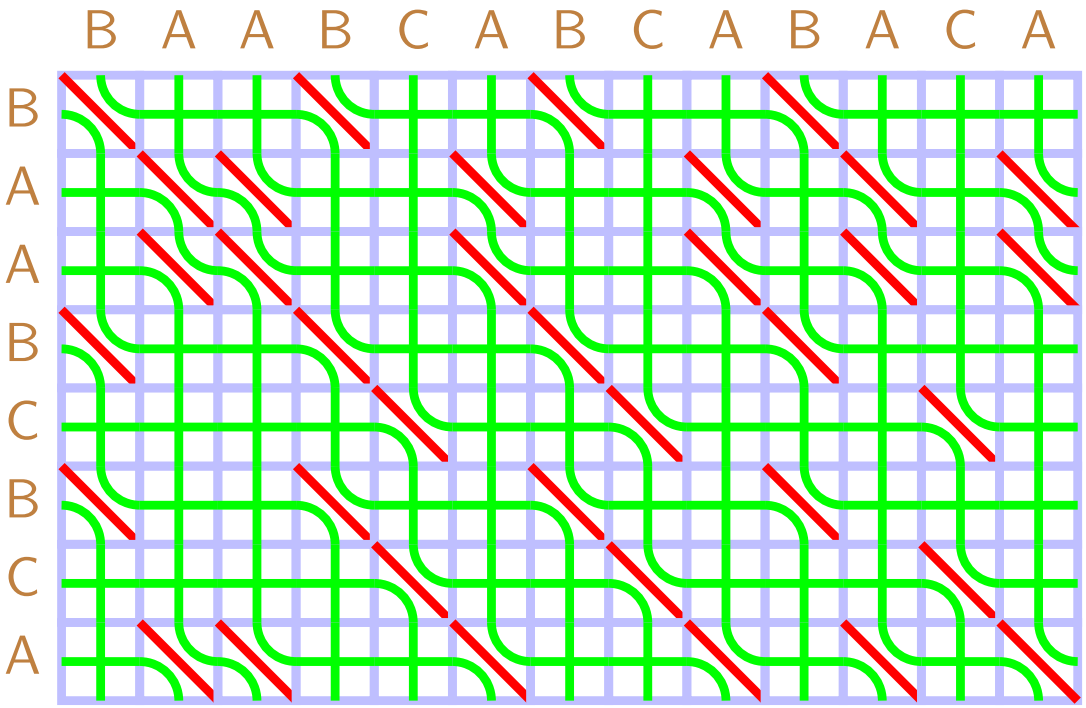
$O\left(\frac{mn}{v}\right)$ [Krusche, T: 2009]

Iterating through LCS grid in vertical/horizontal integer vectors

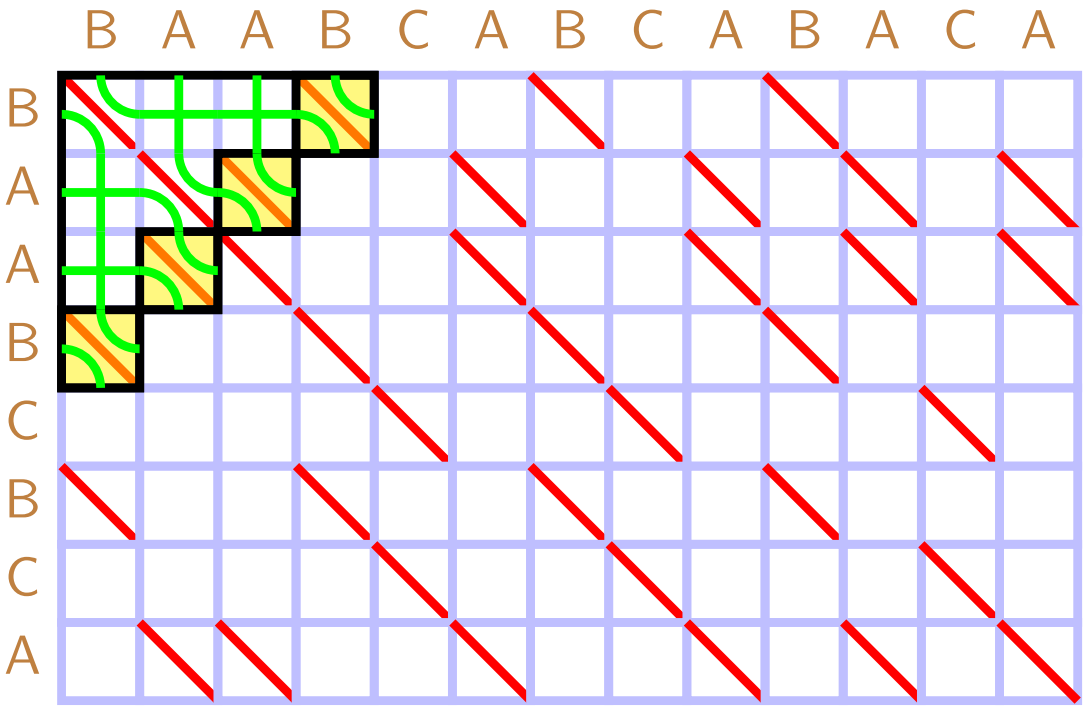
Combing two sticky braid strands in every cell

Vector cells independent

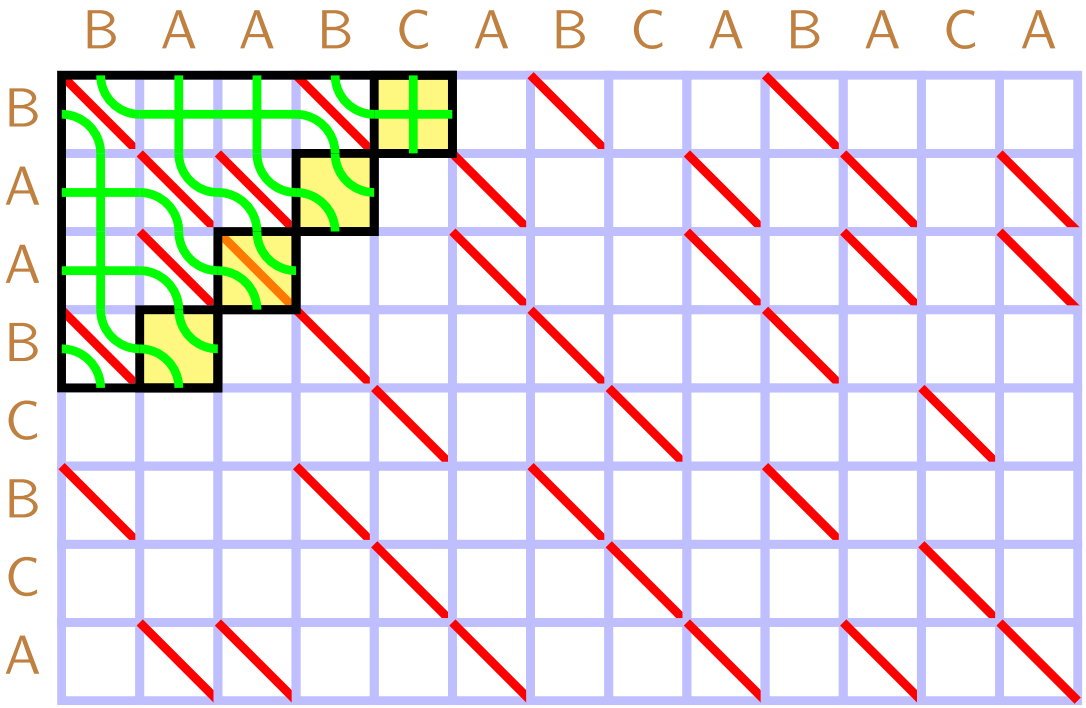
Semi-local LCS by vector-parallel iterative combing



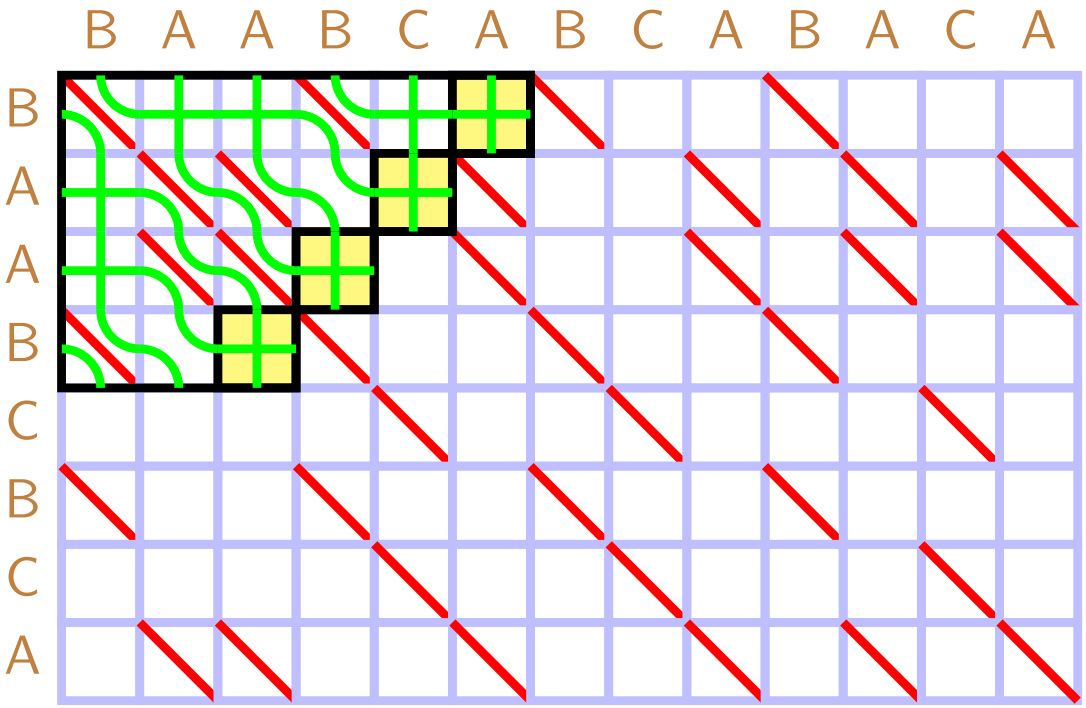
Semi-local LCS by vector-parallel iterative combing



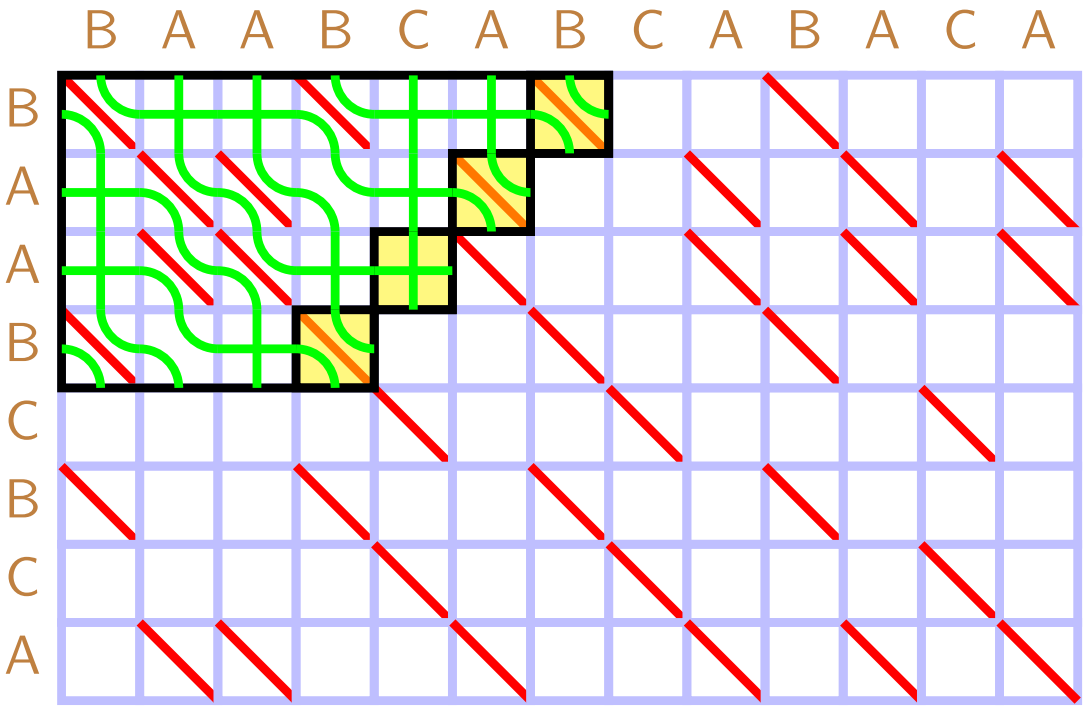
Semi-local LCS by vector-parallel iterative combing



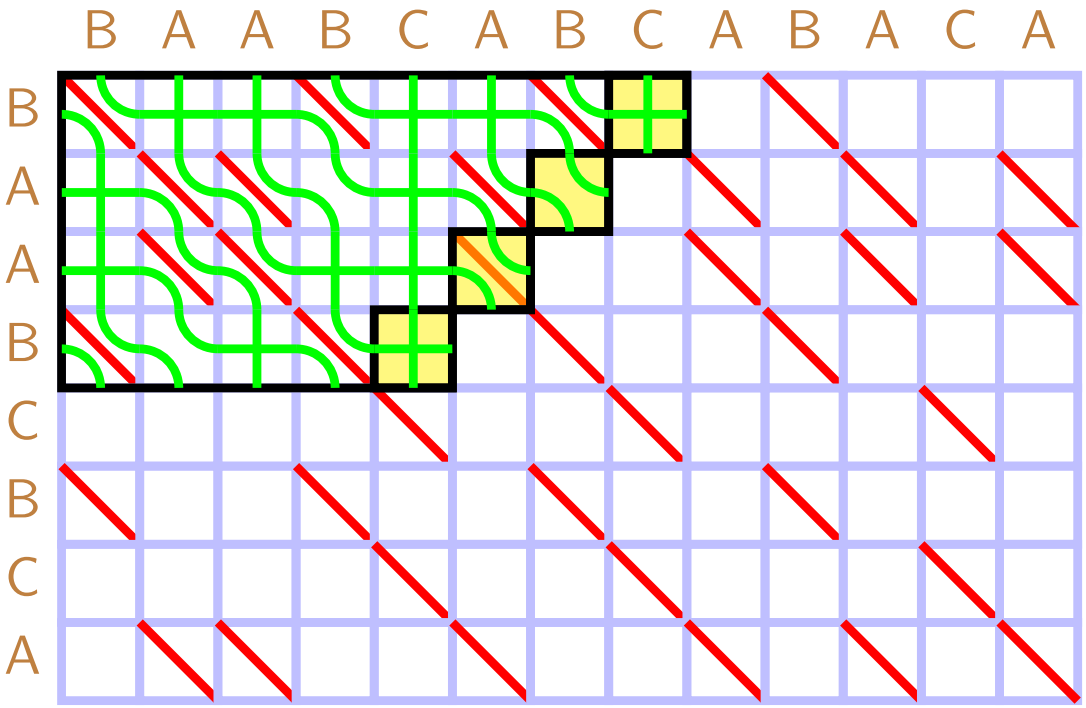
Semi-local LCS by vector-parallel iterative combing



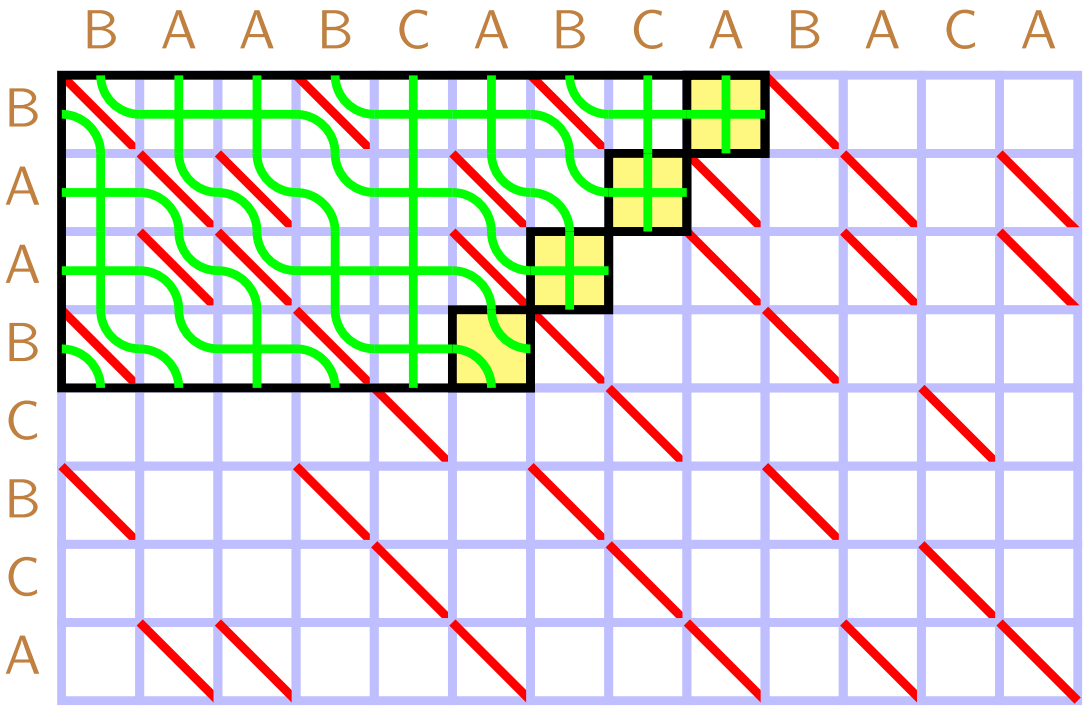
Semi-local LCS by vector-parallel iterative combing



Semi-local LCS by vector-parallel iterative combing



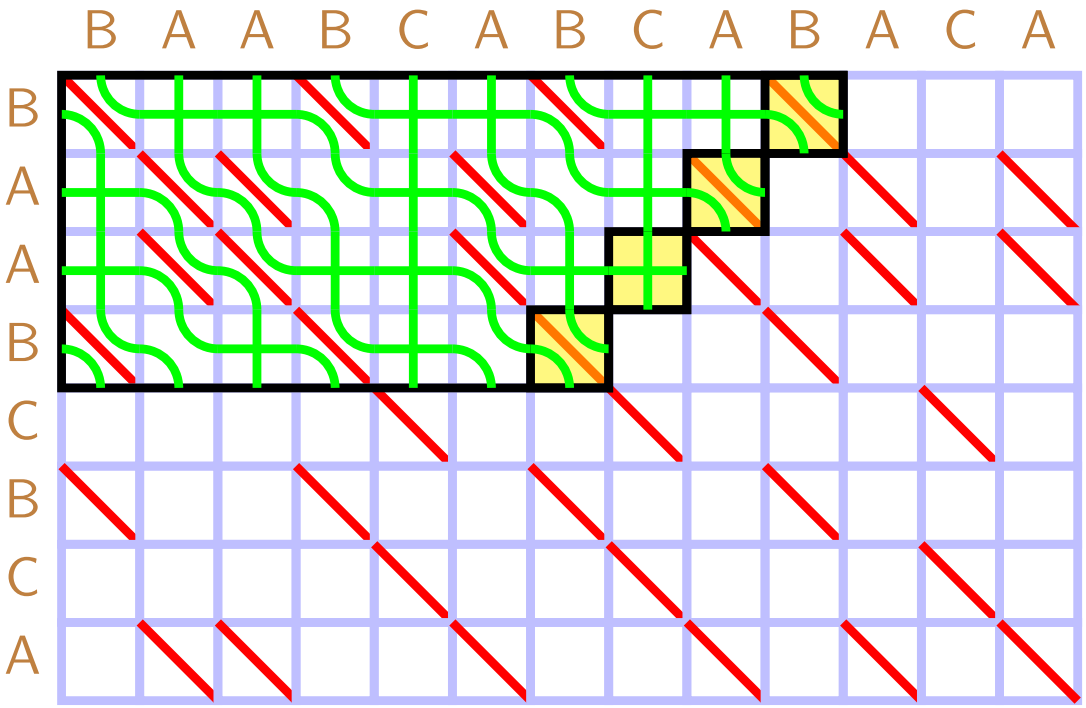
Semi-local LCS by vector-parallel iterative combing



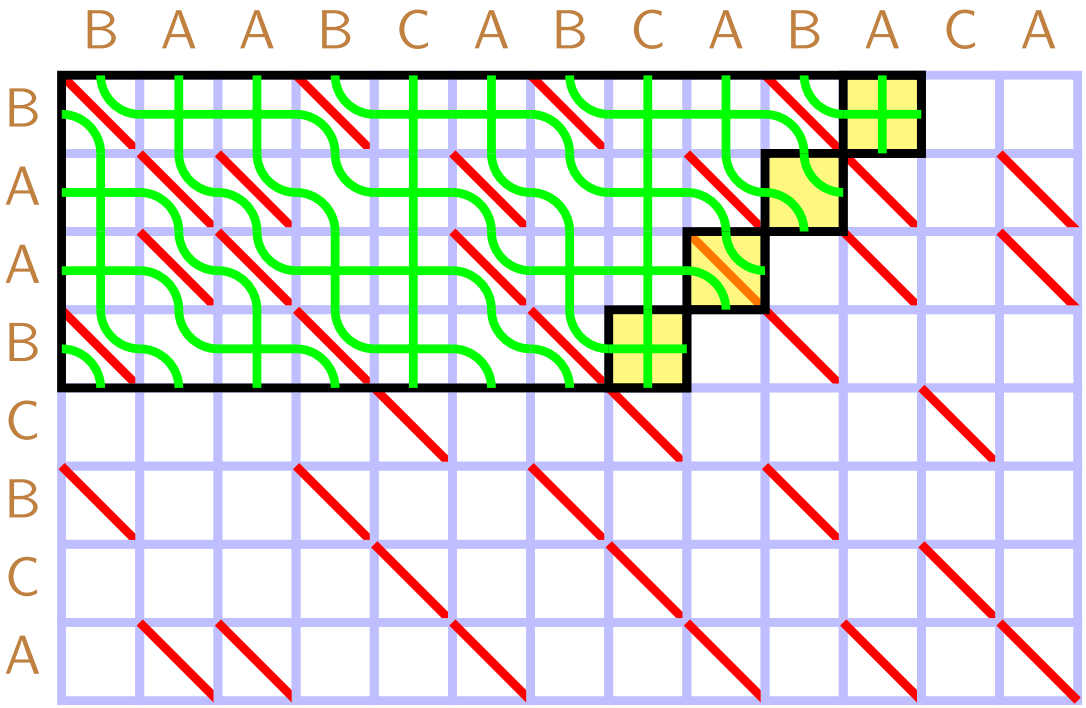
Parallel and dynamic string comparison

Vector-parallel LCS

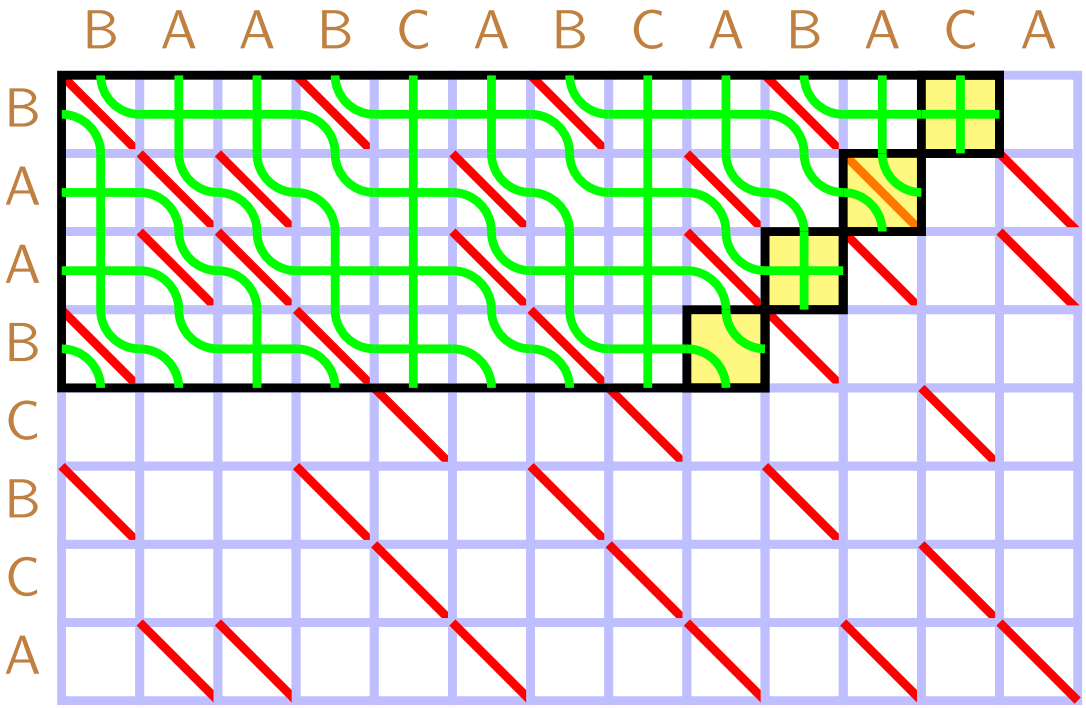
Semi-local LCS by vector-parallel iterative combing



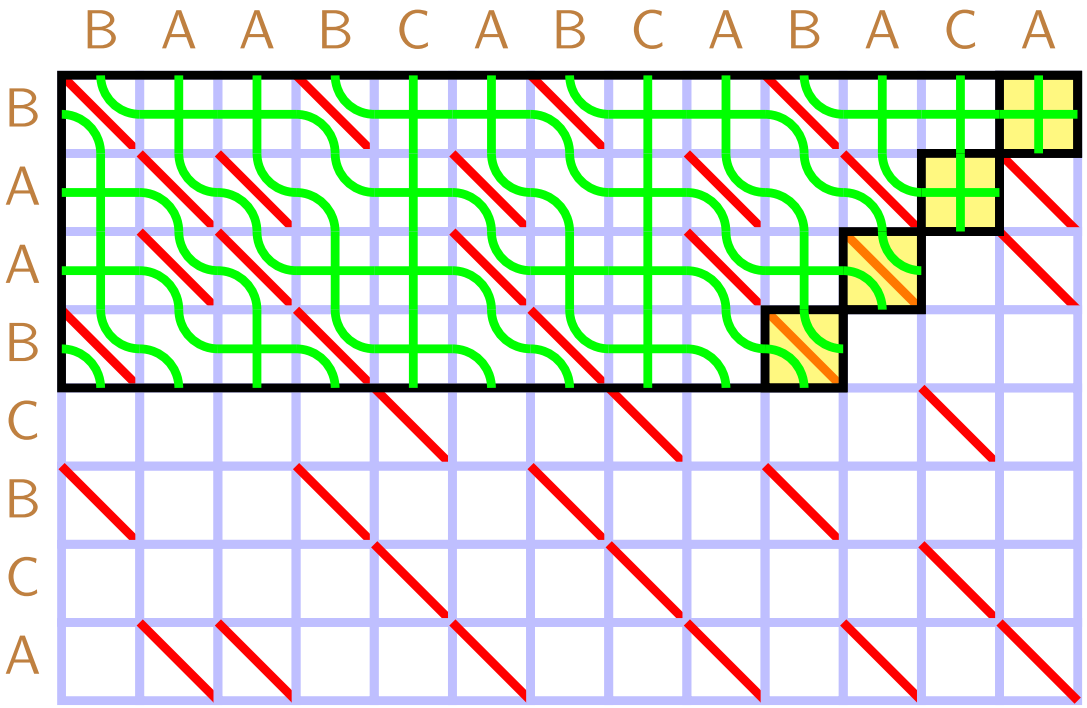
Semi-local LCS by vector-parallel iterative combing



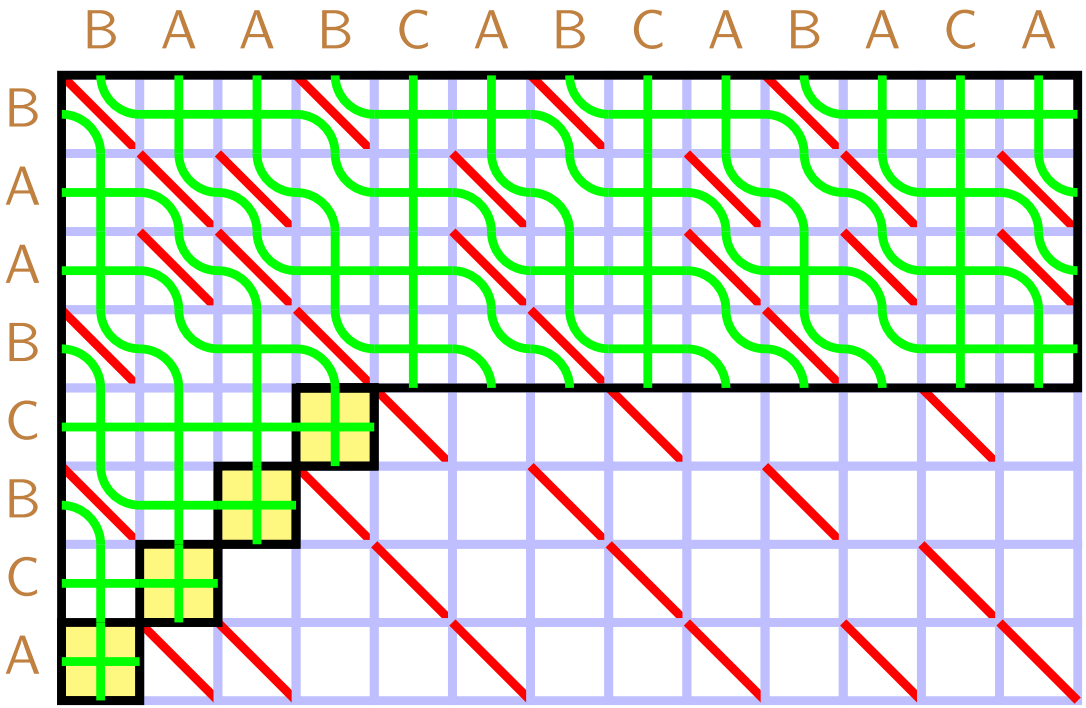
Semi-local LCS by vector-parallel iterative combing



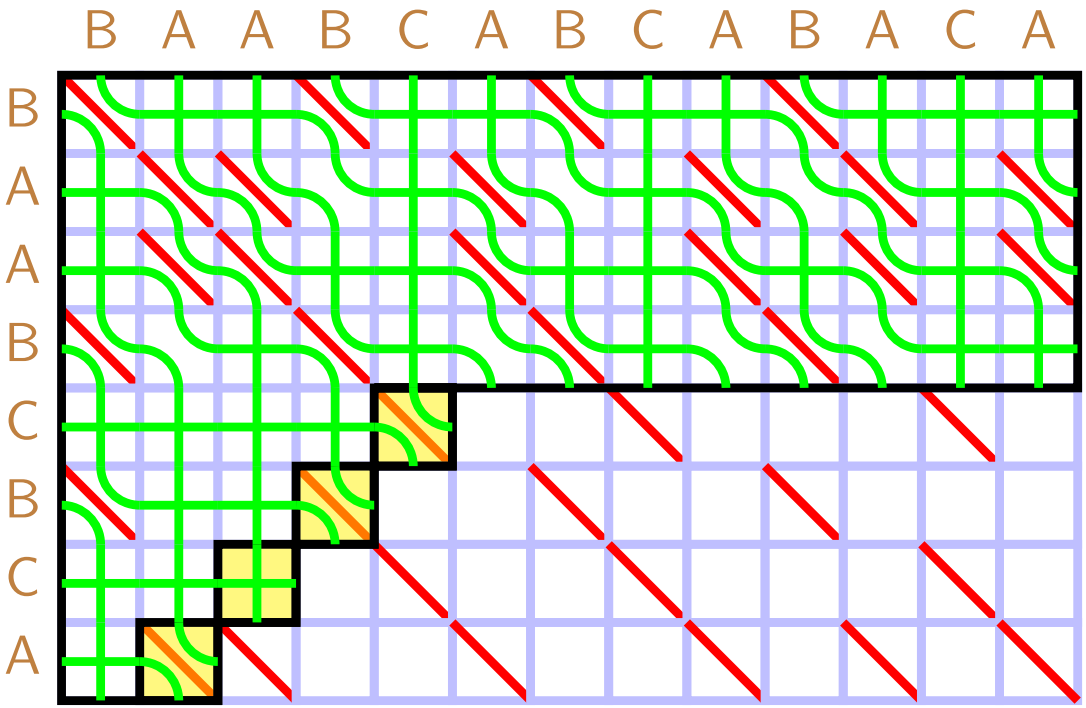
Semi-local LCS by vector-parallel iterative combing



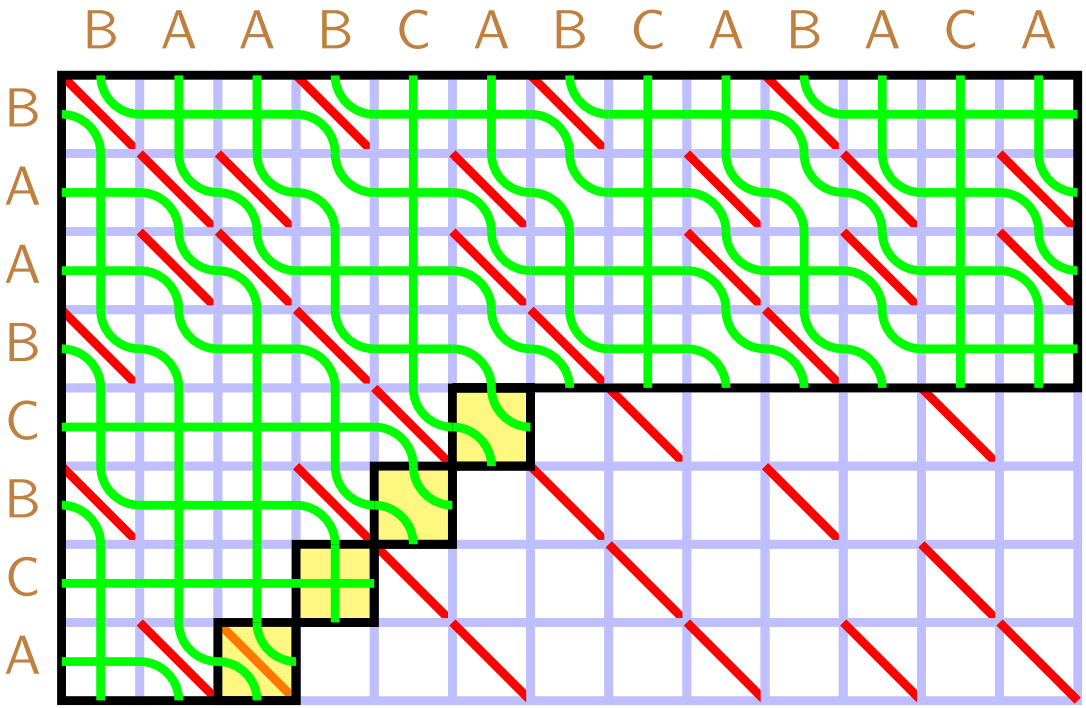
Semi-local LCS by vector-parallel iterative combing



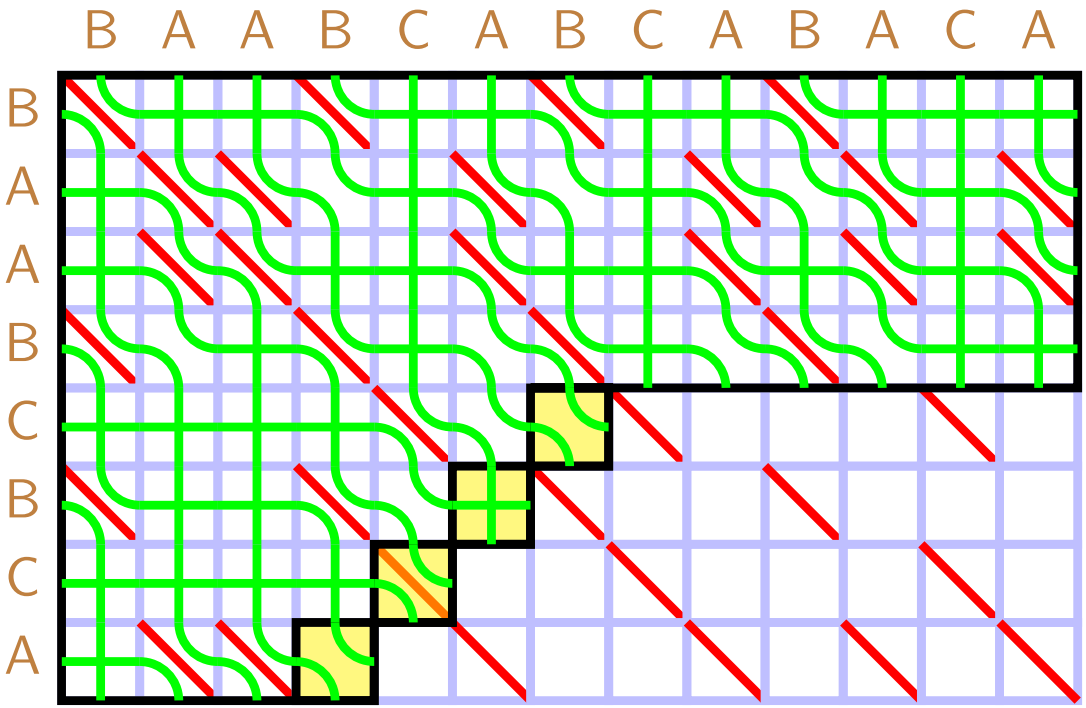
Semi-local LCS by vector-parallel iterative combing



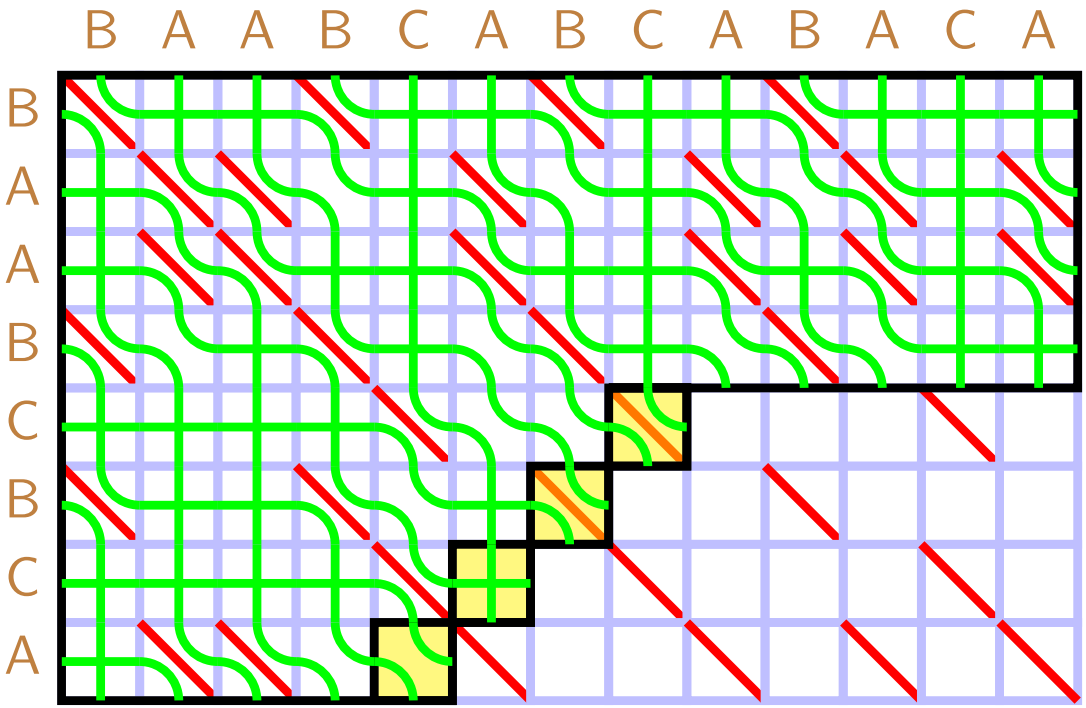
Semi-local LCS by vector-parallel iterative combing



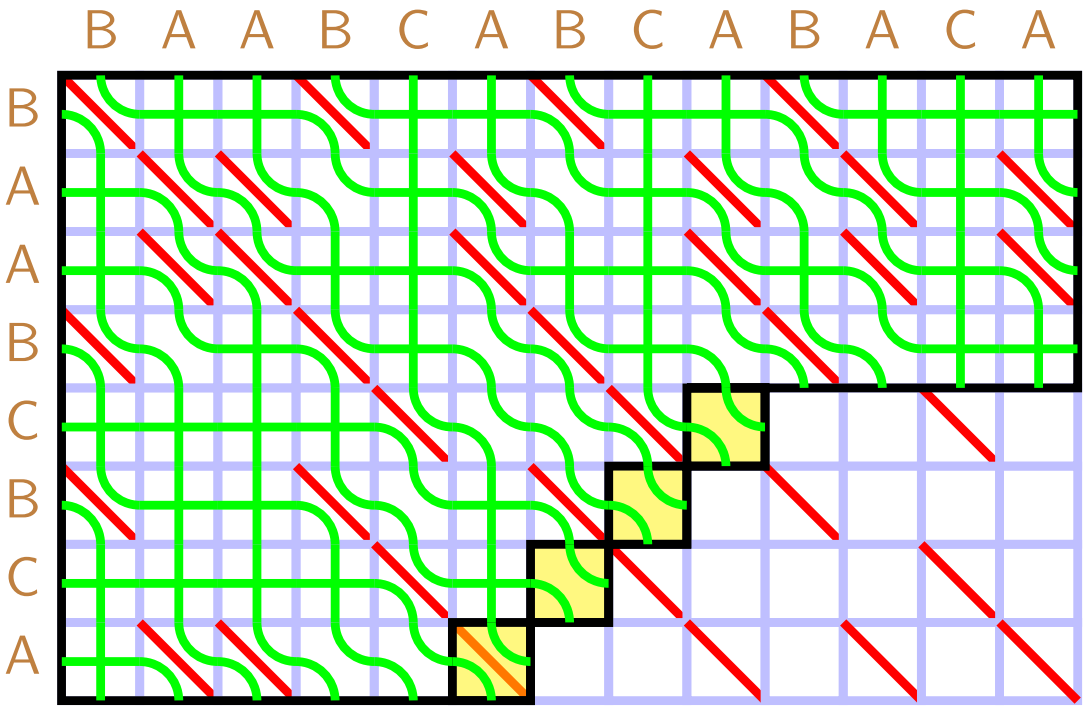
Semi-local LCS by vector-parallel iterative combing



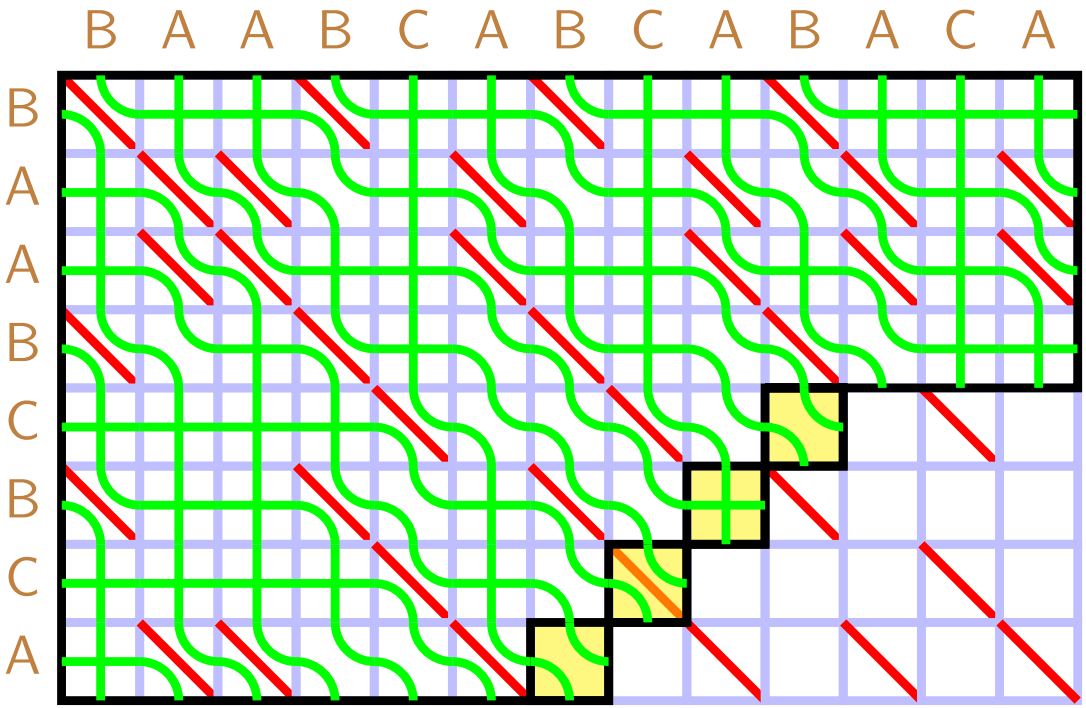
Semi-local LCS by vector-parallel iterative combing



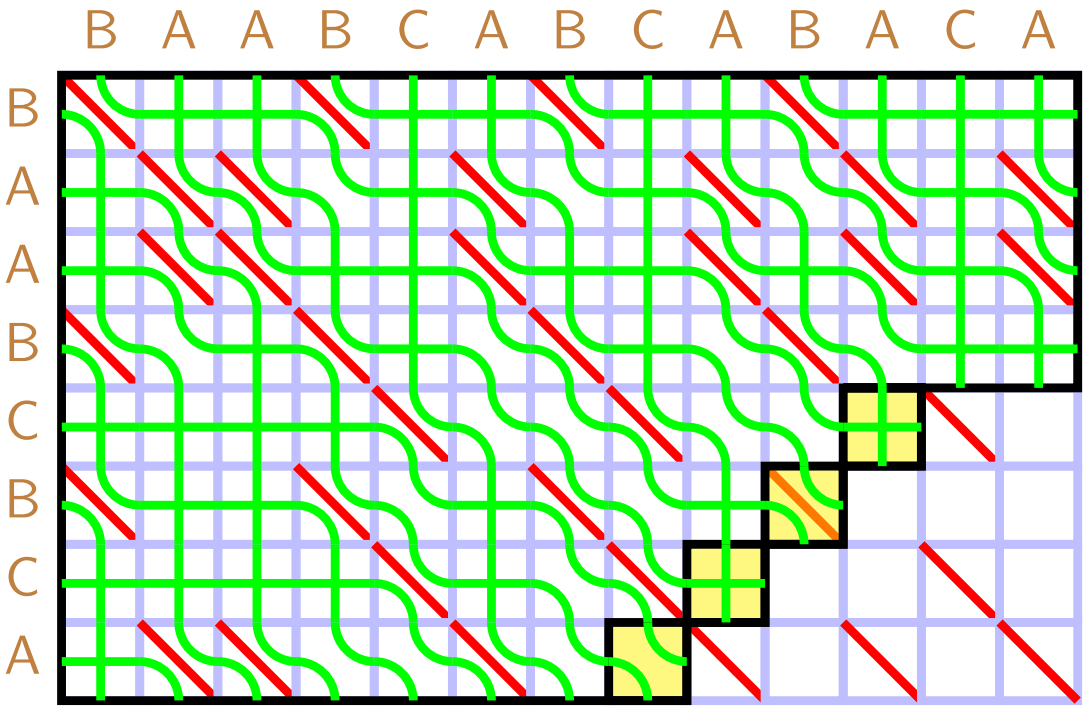
Semi-local LCS by vector-parallel iterative combing



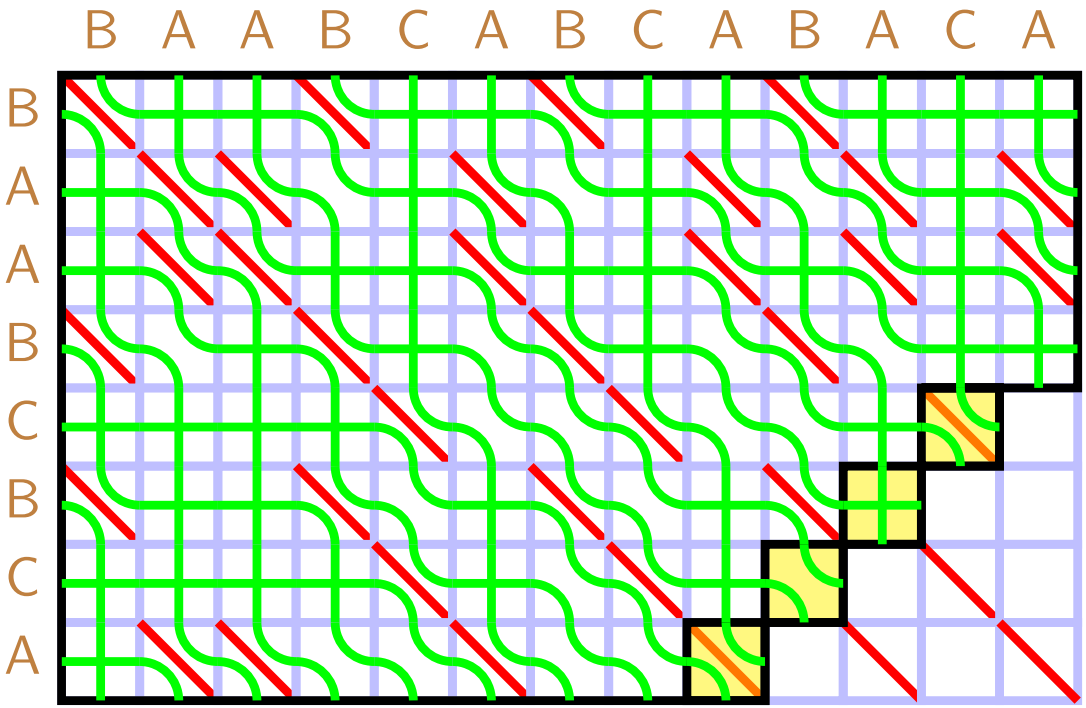
Semi-local LCS by vector-parallel iterative combing



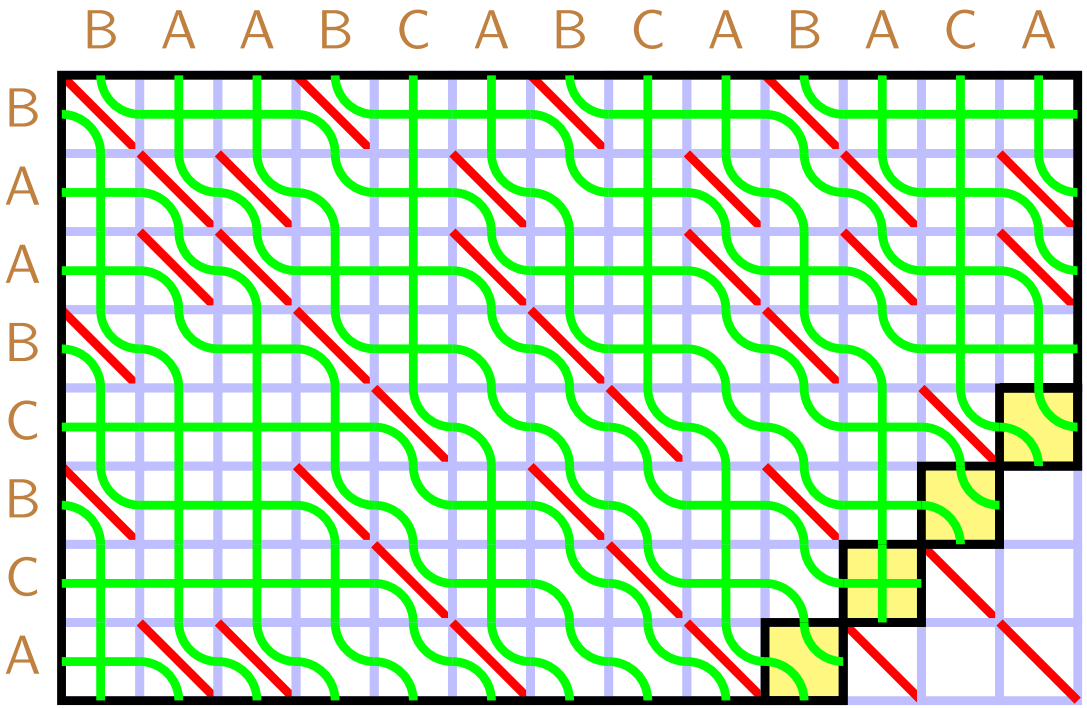
Semi-local LCS by vector-parallel iterative combing



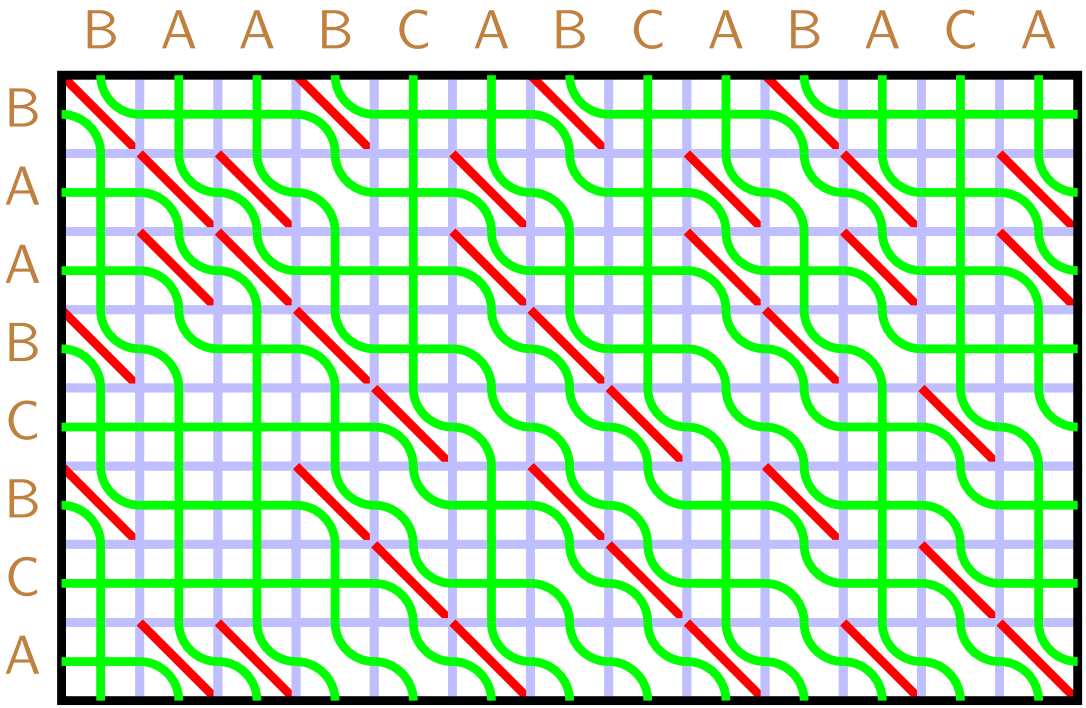
Semi-local LCS by vector-parallel iterative combing



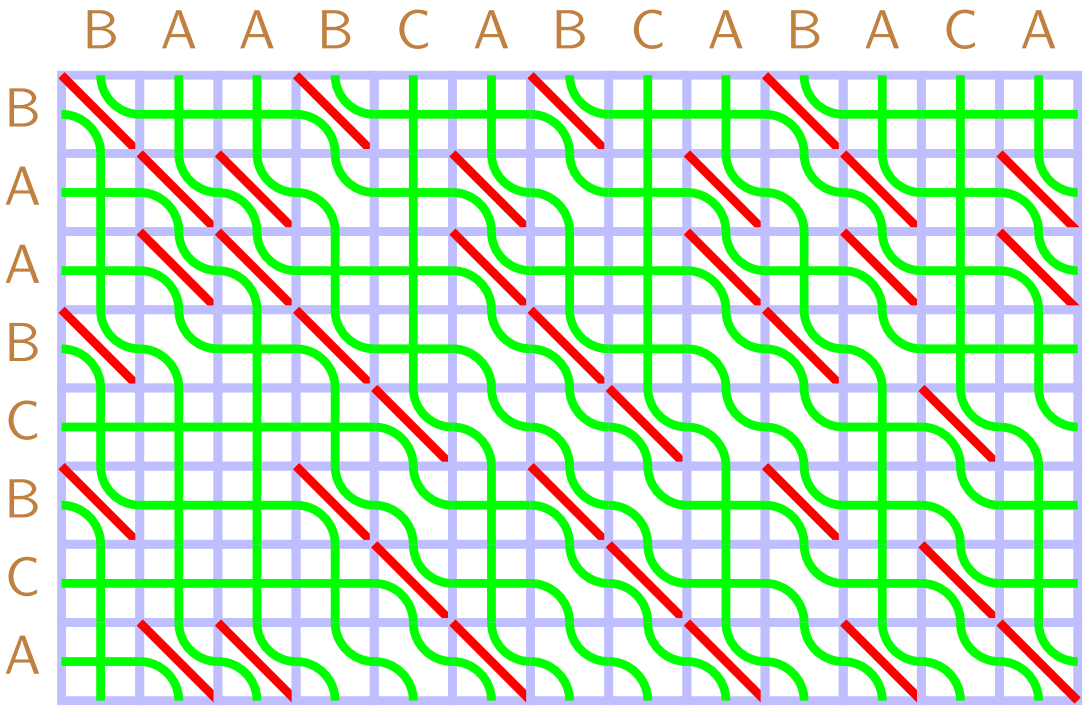
Semi-local LCS by vector-parallel iterative combing



Semi-local LCS by vector-parallel iterative combing



Semi-local LCS by vector-parallel iterative combing



Semi-local LCS by vector-parallel iterative combing [Krusche, T: 2009]

Initialise as saturated braid: mismatch cell = crossing

Iterate over cells in antidiagonal vectors

- match cell: skip, keep strands untangled
- mismatch cell: untangle strands if crossed before

Active vector update: independent cells, time $O(1)$

Overall time $O\left(\frac{mn}{v}\right)$

Parallel and dynamic string comparison

Vector-parallel LCS

Semi-local LCS by vector-parallel iterative combing (contd.)

Implementation: based on $O(mnw)$ repeated iterative combing, ideas from optimal $O(mn)$ algorithm; effective time $O(mnw^{0.5})$ [Krusche: 2010]

Alignment scores (match 1, mismatch 0, gap -0.5) via grid blow-up: slowdown $\times 4$ over LCS scores

C++, Intel assembly (x86, x86_64, MMX/SSE2 data parallelism)

SMP parallelism (two processors)

Single processor:

- speedup $\times 10$ over heavily optimised, bit-parallel naive algorithm
- speedup $\times 7$ over ad-hoc heuristics

Two processors: extra speedup $\times 2$, near-perfect parallelism

Parallel and dynamic string comparison

Vector-parallel LCS

Semi-local LCS by vector-parallel iterative combing (contd.)

Used by biologists to study evolutionary conservation in non-coding DNA between distant relative species

- helps to understand transcriptional regulation
- conservation for 100 mlns of years implies function

Plants: Arabidopsis thaliana (thale cress), Carica papaya (papaya), Populus trichocarpa (poplar), Vitis vinifera (grape)

[Picot+: 2010] [Baxter+: 2012]

Insects: Nasonia vitripennis (parasitic wasp) vs each of Apis mellifera, Atta cephalotes, Solenopsis invicta, Drosophila melanogaster, Megaselia scalaris, Aedes aegypti, Bombyx mori, Danaus plexippus, Heliconius melpomene, Dendroctonus ponderosae, Tribolium castaneum, Acyrthosiphon pisum

[Davies+: 2015]

the plant journal

SEB SOCIETY FOR EXPERIMENTAL BIOLOGY

The Plant Journal (2010) **64**, 165–176 doi: 10.1111/j.1365-313X.2010.04314.x

TECHNICAL ADVANCE

Evolutionary analysis of regulatory sequences (EARS) in plants

Emma Picot[1,2], Peter Krusche[3], Alexander Tiskin[3], Isabelle Carré[2] and Sascha Ott[4,*]
[1]*Systems Biology Doctoral Training Centre, University of Warwick, Coventry CV4 7AL, UK,*
[2]*Department of Biological Sciences, University of Warwick, Coventry CV4 7AL, UK,*
[3]*Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK, and*
[4]*Warwick Systems Biology Centre, University of Warwick, Coventry CV4 7AL, UK*

Received 17 May 2010; revised 6 July 2010; accepted 12 July 2010.
*For correspondence (fax +44 2476 575 795; e-mail s.ott@warwick.ac.uk).

SUMMARY

Identification of regulatory sequences within non-coding regions of DNA is an essential step towards elucidation of gene networks. This approach constitutes a major challenge, however, as only a very small fraction of non-coding DNA is thought to contribute to gene regulation. The mapping of regulatory regions traditionally involves the laborious construction of promoter deletion series which are then fused to reporter genes and assayed in transgenic organisms. Bioinformatic methods can be used to scan sequences for

LARGE-SCALE BIOLOGY ARTICLE

Conserved Noncoding Sequences Highlight Shared Components of Regulatory Networks in Dicotyledonous Plants[W]

Laura Baxter,[a,1] Aleksey Jironkin,[a,1] Richard Hickman,[a,1] Jay Moore,[a] Christopher Barrington,[a] Peter Krusche,[a] Nigel P. Dyer,[b] Vicky Buchanan-Wollaston,[a,c] Alexander Tiskin,[d] Jim Beynon,[a,c] Katherine Denby,[a,c] and Sascha Ott[a,2]

[a] Warwick Systems Biology Centre, University of Warwick, Coventry CV4 7AL, United Kingdom
[b] Molecular Organisation and Assembly in Cells Doctoral Training Centre, University of Warwick, Coventry CV4 7AL, United Kingdom
[c] School of Life Sciences, University of Warwick, Coventry CV4 7AL, United Kingdom
[d] Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom

Conserved noncoding sequences (CNSs) in DNA are reliable pointers to regulatory elements controlling gene expression. Using a comparative genomics approach with four dicotyledonous plant species (*Arabidopsis thaliana*, papaya [*Carica papaya*], poplar [*Populus trichocarpa*], and grape [*Vitis vinifera*]), we detected hundreds of CNSs upstream of *Arabidopsis* genes. Distinct positioning, length, and enrichment for transcription factor binding sites suggest these CNSs play a functional role in transcriptional regulation. The enrichment of transcription factors within the set of genes associated with CNS is consistent with the hypothesis that together they form part of a conserved transcriptional network whose function is to regulate other transcription factors and control development. We identified a set of promoters where regulatory mechanisms are likely to be shared between the model organism *Arabidopsis* and other dicots, providing areas of focus for further research.

Davies *et al. BMC Evolutionary Biology* (2015) 15:227
DOI 10.1186/s12862-015-0499-6

BMC Evolutionary Biology

RESEARCH ARTICLE Open Access

Analysis of 5′ gene regions reveals extraordinary conservation of novel non-coding sequences in a wide range of animals

Nathaniel J. Davies[1], Peter Krusche[2], Eran Tauber[1*] and Sascha Ott[2]

Abstract

Background: Phylogenetic footprinting is a comparative method based on the principle that functional sequence elements will acquire fewer mutations over time than non-functional sequences. Successful comparisons of distantly related species will thus yield highly important sequence elements likely to serve fundamental biological roles. RNA regulatory elements are less well understood than those in DNA. In this study we use the emerging model organism *Nasonia vitripennis*, a parasitic wasp, in a comparative analysis against 12 insect genomes to identify deeply conserved non-coding elements (CNEs) conserved in large groups of insects, with a focus on 5′ UTRs and promoter sequences.

Results: We report the identification of 322 CNEs conserved across a broad range of insect orders. The identified regions are associated with regulatory and developmental genes, and contain short footprints revealing aspects of their likely function in translational regulation. The most ancient regions identified in our analysis were all found to overlap transcribed regions of genes, reflecting stronger conservation of translational regulatory elements than transcriptional

Parallel and dynamic string comparison

Vector-parallel LCS

Semi-local LCS by vector-parallel iterative combing (contd.)

Combing logic: $(c', s') \leftarrow comb(s, c, \mu) = \text{if } \mu \text{ then } (s, c) \text{ else } sort(s, c)$

AVX-512: `VPMINUW/VPMAXUW`

> *Performs a SIMD compare of the packed unsigned [16-bit] integers in the second source operand and the first source operand and returns the minimum/maximum value for each pair of integers to the destination operand... The destination operand is conditionally updated based on writemask. (Intel® 64 and IA-32 Architectures Software Developer's Manual, May 2019)*

Perfect match with combing logic: source s, c; destination c', s'; mask μ

Parallel and dynamic string comparison

Vector-parallel LCS

Semi-local LCS by vector-parallel iterative combing (contd.)

String a in (non-overlapping) blocks of length $v = 512/16 = 32$

String b in (slightly overlapping) blocks of length $2^{16} = 65536$

Two antidiagonal frontier vectors: horizontal strands, vertical strands

Each strand assigned unique 16-bit value

Block sizes chosen to prevent strand value overflow

Parallel and dynamic string comparison

Vector-parallel LCS

Semi-local LCS by vector-parallel iterative combing (contd.)

Inner loop (instruction counts):

- compare block in a vs v-window in b; form match mask (1 instr)
- sort strand pairs between frontier vectors, conditional on match mask (`VPMINUW`/`VPMAXUW`, 2 instr)
- advance v-window in b: pull in new char; discard old char (1 instr)
- advance frontier vectors: pull in new vertical strand; push out old vertical strand (2 instr)

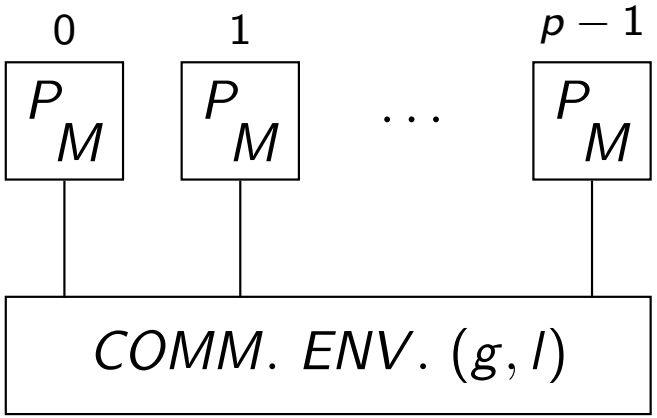
Loop unrolling:

- $\times 2$ to prevent vector swapping between even/odd iterations
- $\times 16$ to keep all data in registers

Bulk-Synchronous Parallel (BSP) computer

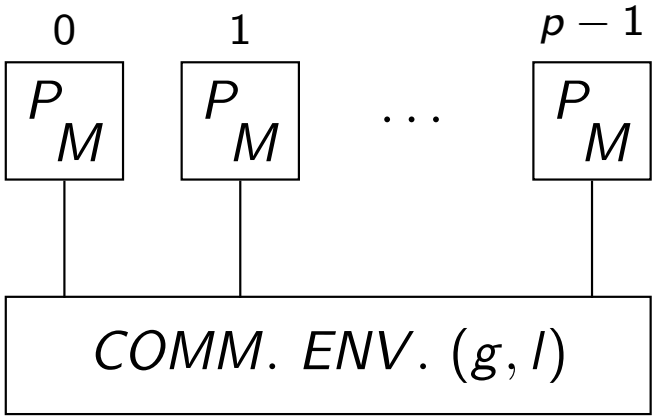
[Valiant: 1990]

Simple, realistic general-purpose parallel model



Bulk-Synchronous Parallel (BSP) computer [Valiant: 1990]

Simple, realistic general-purpose parallel model



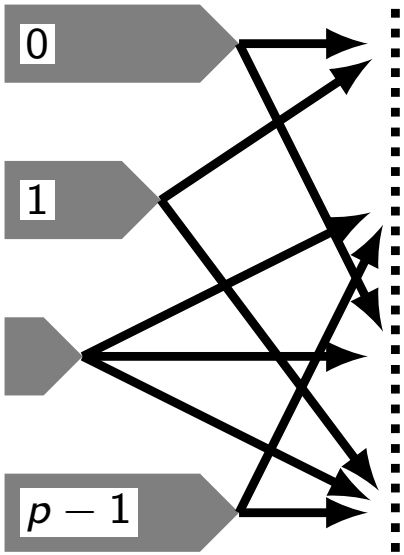
Contains

- p processors, each with local memory (1 time unit/operation)
- communication environment, including a network and an external memory (g time units/data unit communicated)
- barrier synchronisation mechanism (l time units/synchronisation)

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

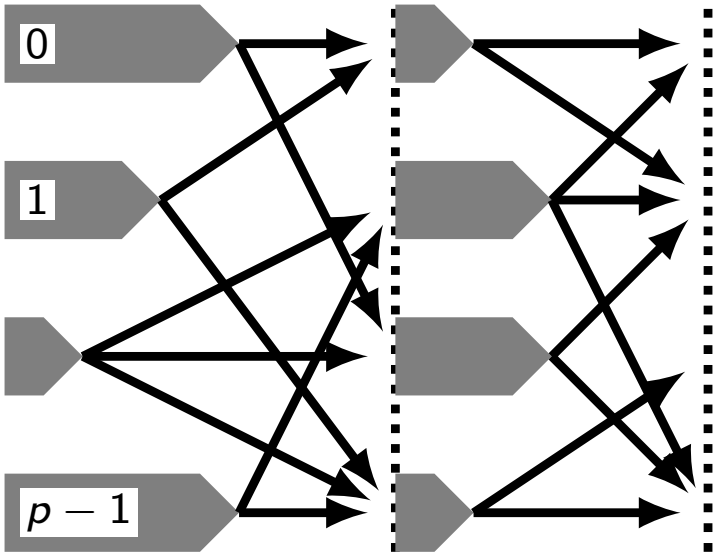
BSP computation: sequence of parallel supersteps



Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

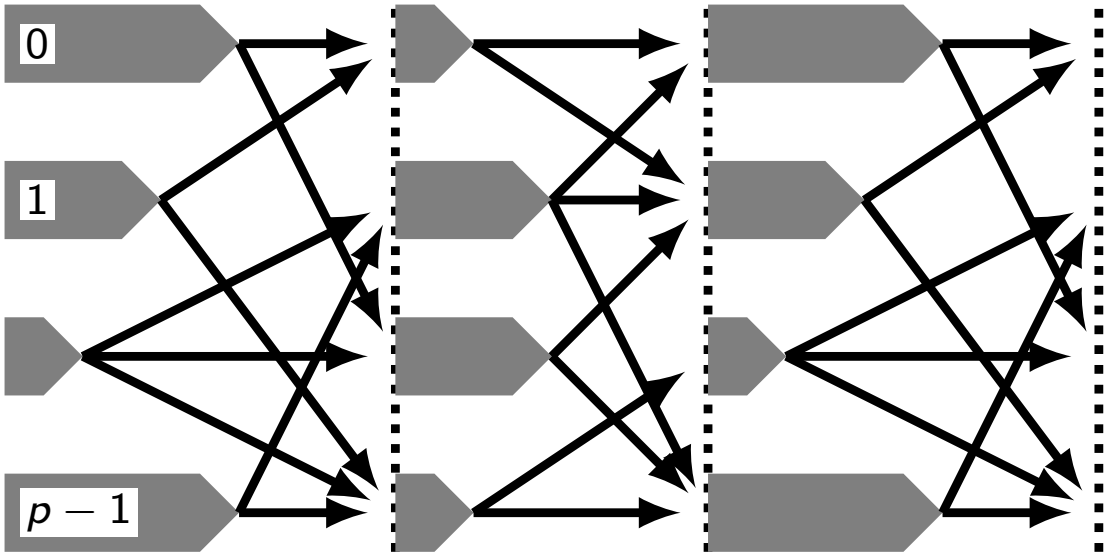
BSP computation: sequence of parallel supersteps



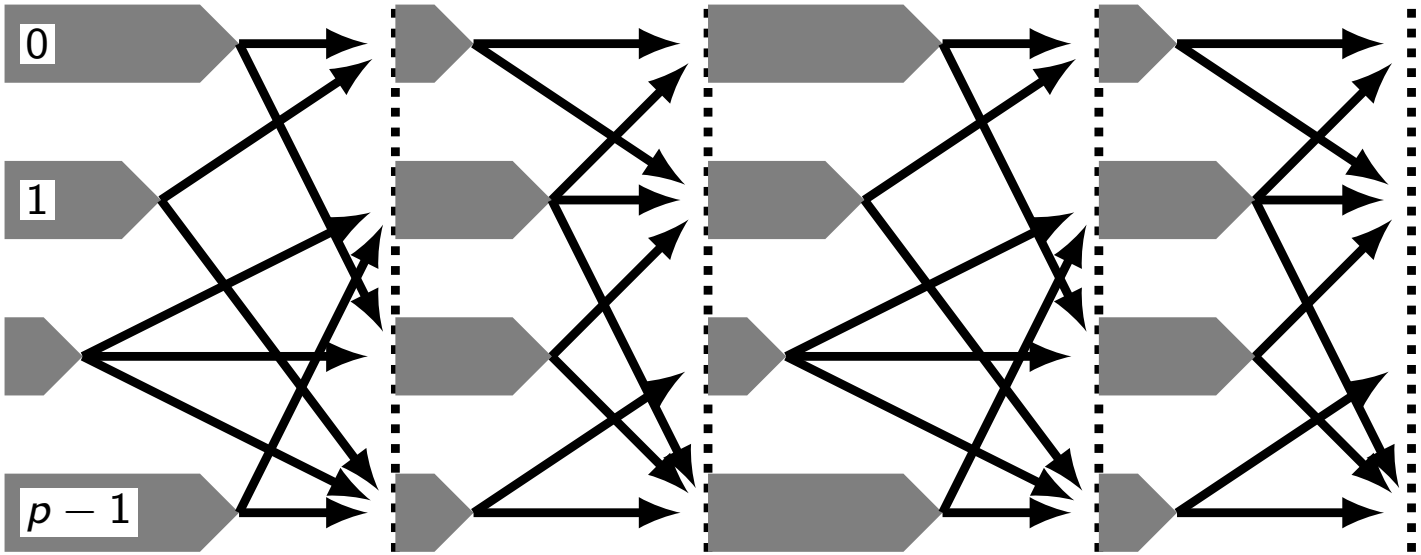
Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

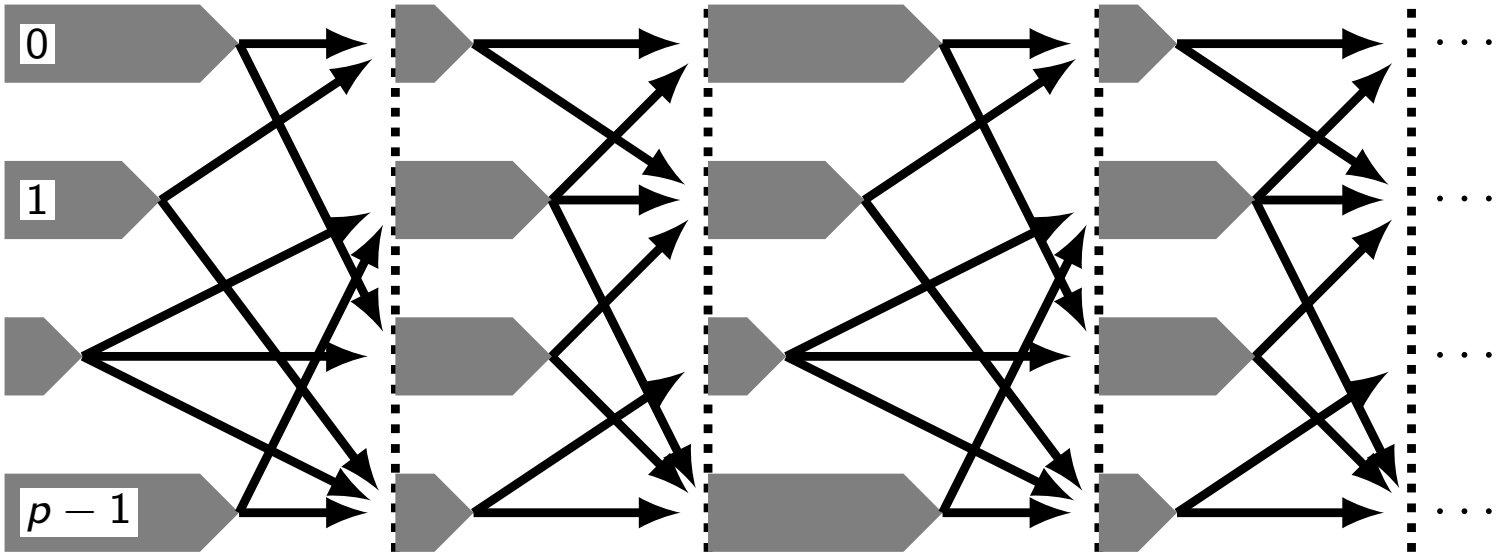
BSP computation: sequence of parallel supersteps



BSP computation: sequence of parallel supersteps



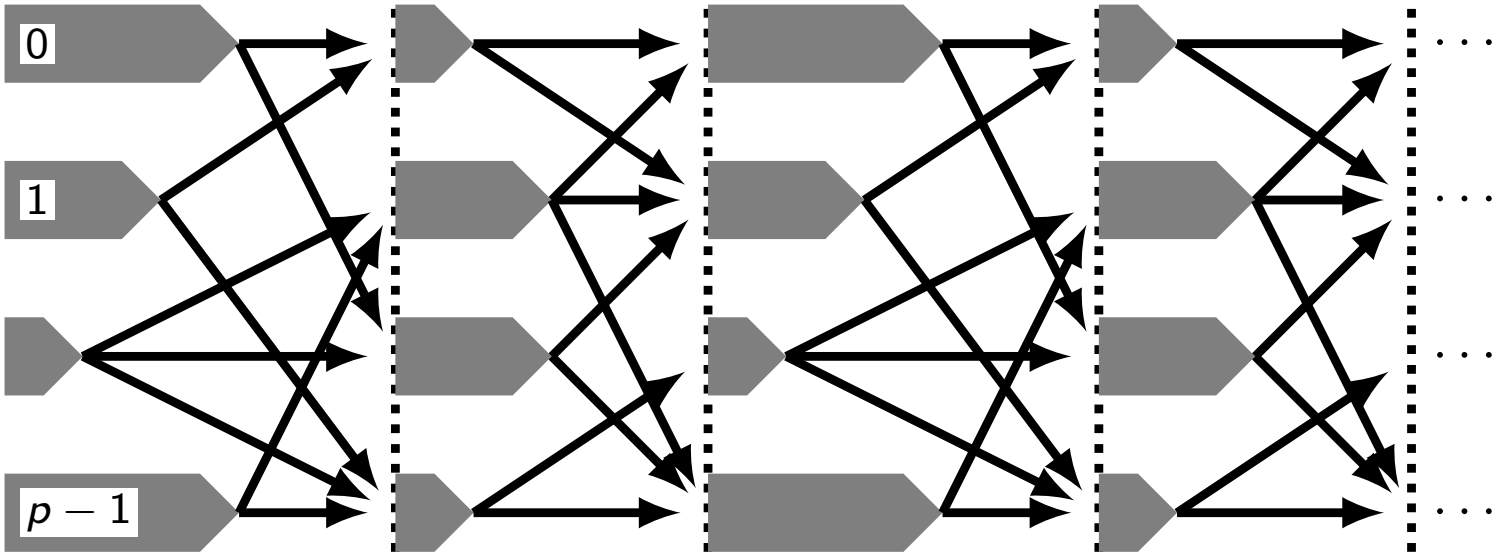
BSP computation: sequence of parallel supersteps



Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

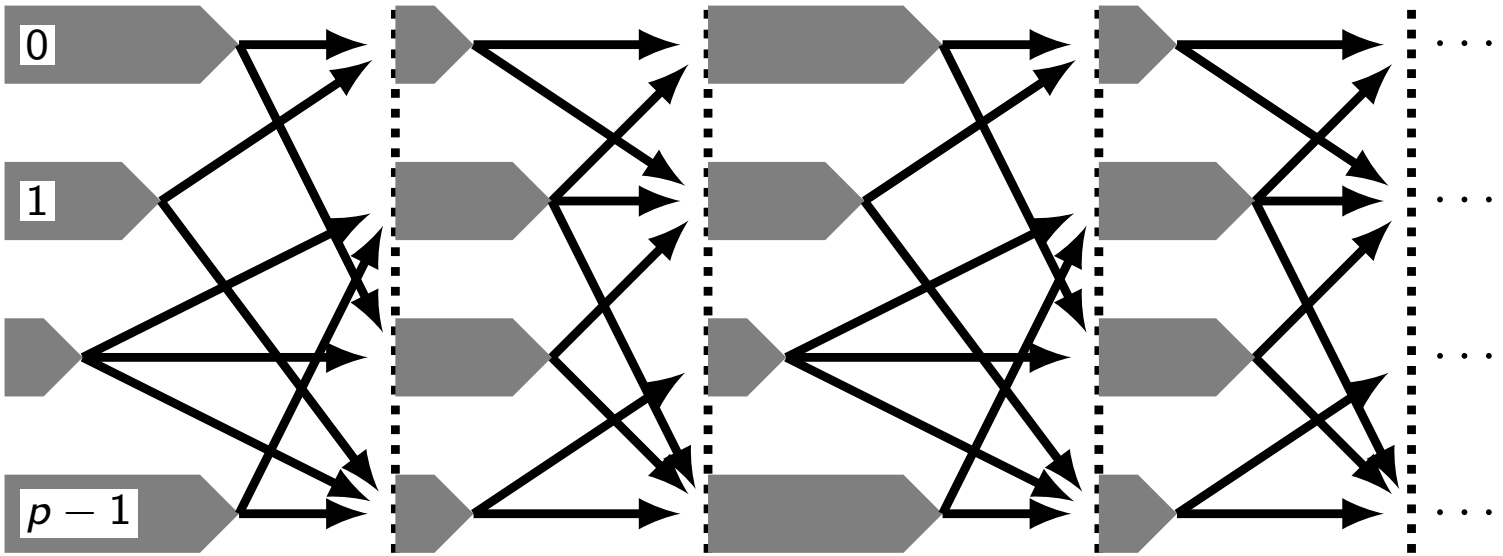
BSP computation: sequence of parallel supersteps



Asynchronous computation/communication within supersteps (includes data exchange with external memory)

Synchronisation before/after each superstep

BSP computation: sequence of parallel supersteps



Asynchronous computation/communication within supersteps (includes data exchange with external memory)

Synchronisation before/after each superstep

Cf. CSP: parallel collection of sequential processes

Compositional cost model

For individual processor *proc* in superstep *sstep*:

- $comp(sstep, proc)$: the amount of local computation and local memory operations by processor *proc* in superstep *sstep*
- $comm(sstep, proc)$: the amount of data sent and received by processor *proc* in superstep *sstep*

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

Compositional cost model

For individual processor *proc* in superstep *sstep*:

- $comp(sstep, proc)$: the amount of local computation and local memory operations by processor *proc* in superstep *sstep*
- $comm(sstep, proc)$: the amount of data sent and received by processor *proc* in superstep *sstep*

For the whole BSP computer in one superstep *sstep*:

- $comp(sstep) = \max_{0 \leq proc < p} comp(sstep, proc)$
- $comm(sstep) = \max_{0 \leq proc < p} comm(sstep, proc)$
- $cost(sstep) = comp(sstep) + comm(sstep) \cdot g + l$

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

For the whole BSP computation with *sync* supersteps:

- $comp = \sum_{0 \le sstep < sync} comp(sstep)$
- $comm = \sum_{0 \le sstep < sync} comm(sstep)$
- $cost = \sum_{0 \le sstep < sync} cost(sstep) = comp + comm \cdot g + sync \cdot l$

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

For the whole BSP computation with *sync* supersteps:

- $comp = \sum_{0 \le sstep < sync} comp(sstep)$
- $comm = \sum_{0 \le sstep < sync} comm(sstep)$
- $cost = \sum_{0 \le sstep < sync} cost(sstep) = comp + comm \cdot g + sync \cdot l$

The input/output data are stored in the external memory; the cost of input/output is included in *comm*

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

For the whole BSP computation with *sync* supersteps:

- $comp = \sum_{0 \le sstep < sync} comp(sstep)$
- $comm = \sum_{0 \le sstep < sync} comm(sstep)$
- $cost = \sum_{0 \le sstep < sync} cost(sstep) = comp + comm \cdot g + sync \cdot l$

The input/output data are stored in the external memory; the cost of input/output is included in *comm*

E.g. for a particular linear system solver with an $n \times n$ matrix:

$comp = O(n^3/p)$ $\quad$ $comm = O(n^2/p^{1/2})$ $\quad$ $sync = O(p^{1/2})$

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

BSP software: industrial projects

- Google's Pregel [2010]
- Apache Spark (`hama.apache.org`) [2010]
- Apache Giraph (`giraph.apache.org`) [2011]

BSP software: research projects

- Oxford BSP (`www.bsp-worldwide.org/implmnts/oxtool`) [1998]
- Paderborn PUB (`www2.cs.uni-paderborn.de/~pub`) [1998]
- BSML (`traclifo.univ-orleans.fr/BSML`) [1998]
- BSPonMPI (`bsponmpi.sourceforge.net`) [2006]
- Multicore BSP (`www.multicorebsp.com`) [2011]
- Epiphany BSP (`www.codu.in/ebsp`) [2015]
- Petuum (`petuum.org`) [2015]

The ordered 2D grid

$grid_2(n)$

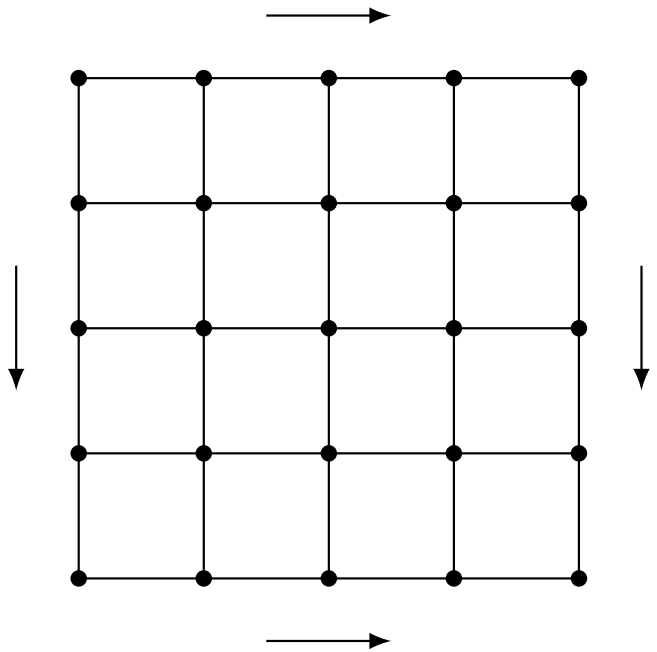
nodes arranged in an $n \times n$ grid

edges directed top-to-bottom, left-to-right

$\leq 2n$ inputs (to left/top borders)

$\leq 2n$ outputs (from right/bottom borders)

size n^2 depth $2n - 1$



The ordered 2D grid

$grid_2(n)$

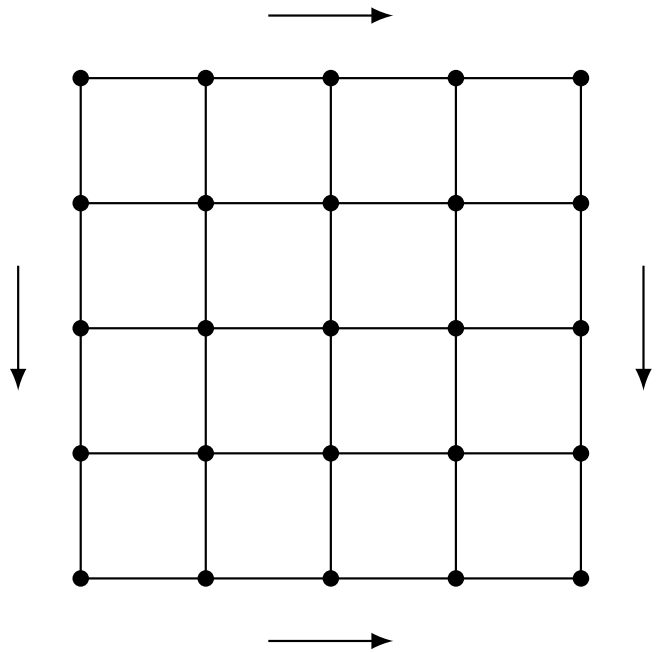
nodes arranged in an $n \times n$ grid

edges directed top-to-bottom, left-to-right

$\leq 2n$ inputs (to left/top borders)

$\leq 2n$ outputs (from right/bottom borders)

size n^2 depth $2n - 1$



Applications: triangular linear system; discretised PDE via Gauss–Seidel iteration (single step); 1D cellular automata; dynamic programming

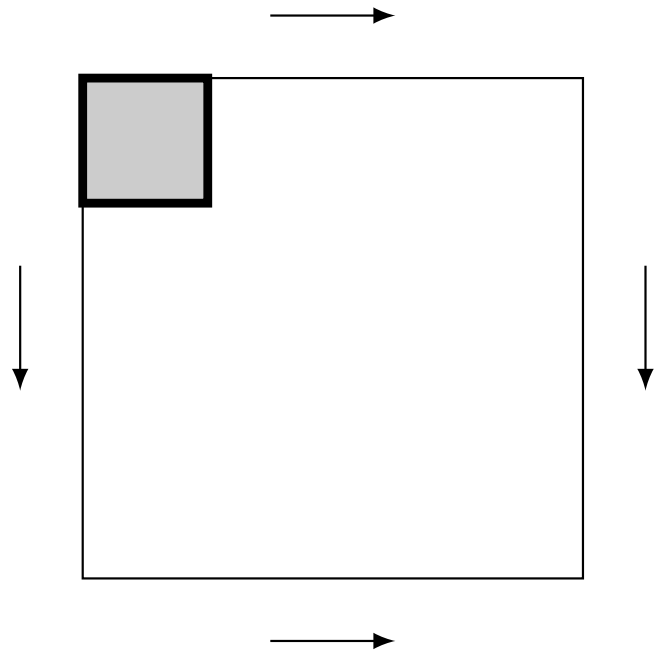
Sequential work $O(n^2)$

BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer

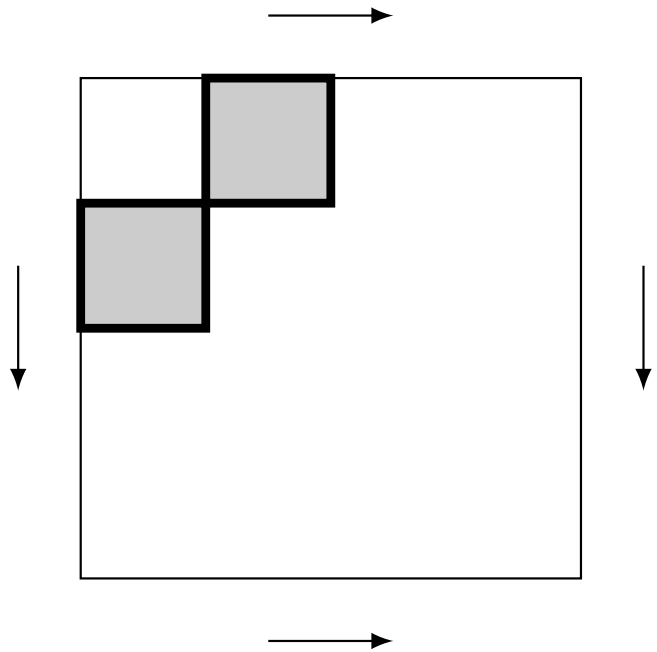


BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer

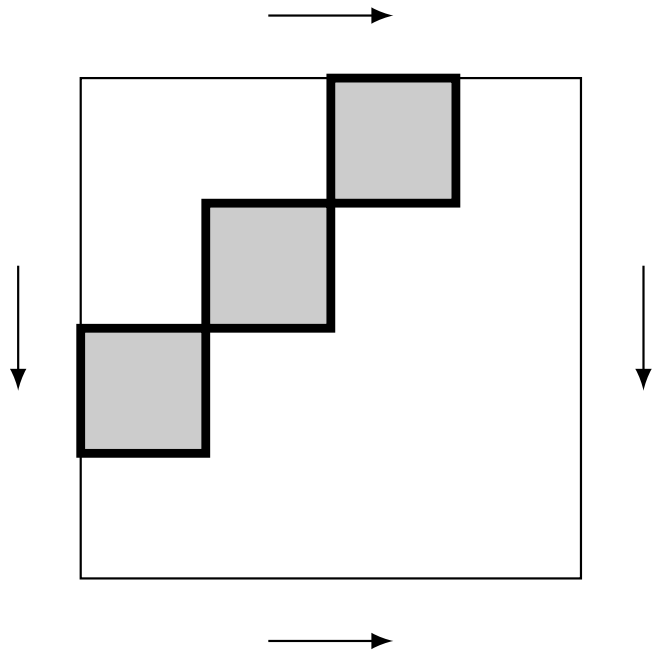


BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer

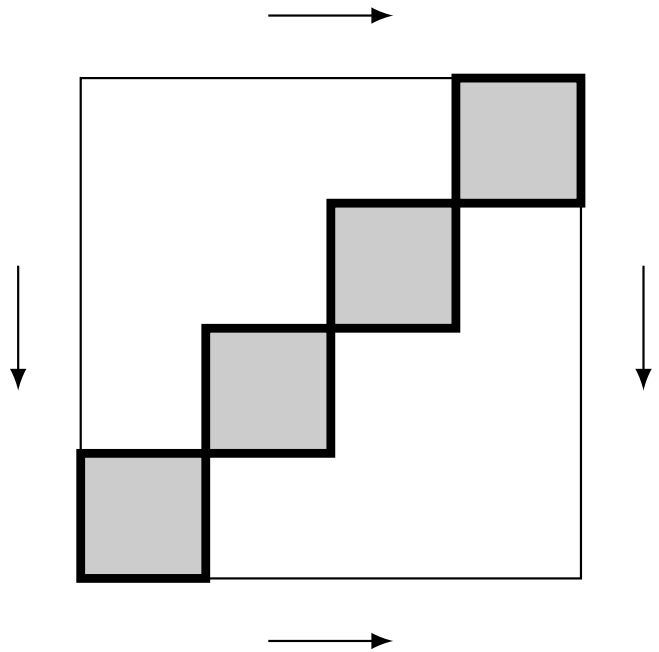


BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer



Parallel and dynamic string comparison

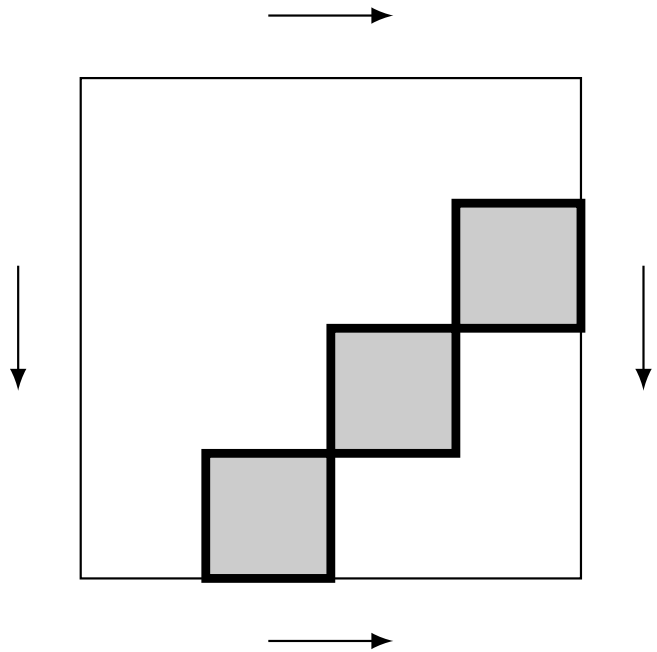
Bulk-synchronous parallel LCS

BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer



Parallel and dynamic string comparison

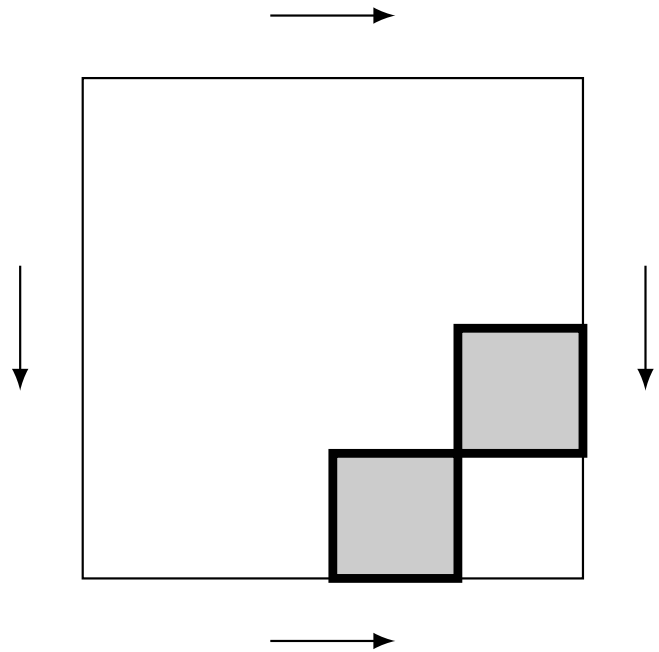
Bulk-synchronous parallel LCS

BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer



Parallel and dynamic string comparison

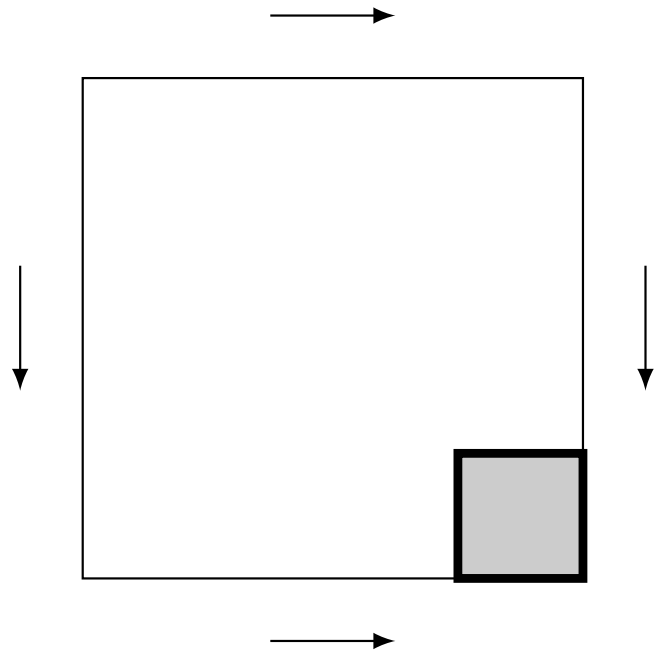
Bulk-synchronous parallel LCS

BSP ordered 2D grid computation

$grid_2(n)$

Consists of a $p \times p$ grid of blocks, each isomorphic to $grid_2(n/p)$

The blocks can be arranged into $2p - 1$ anti-diagonal layers, with $\leq p$ independent blocks in each layer



BSP ordered 2D grid computation (contd.)

The computation proceeds in $2p - 1$ stages, each computing a layer of blocks. In a stage:

- every block assigned to a different processor (some processors idle)
- the processor reads the $2n/p$ block inputs, computes the block, and writes back the $2n/p$ block outputs

BSP ordered 2D grid computation (contd.)

The computation proceeds in $2p - 1$ stages, each computing a layer of blocks. In a stage:

- every block assigned to a different processor (some processors idle)
- the processor reads the $2n/p$ block inputs, computes the block, and writes back the $2n/p$ block outputs

comp: $(2p - 1) \cdot O\big((n/p)^2\big) = O(p \cdot n^2/p^2) = O(n^2/p)$

comm: $(2p - 1) \cdot O(n/p) = O(n)$

BSP ordered 2D grid computation (contd.)

The computation proceeds in $2p-1$ stages, each computing a layer of blocks. In a stage:

- every block assigned to a different processor (some processors idle)
- the processor reads the $2n/p$ block inputs, computes the block, and writes back the $2n/p$ block outputs

comp: $(2p-1)\cdot O\big((n/p)^2\big) = O(p\cdot n^2/p^2) = O(n^2/p)$

comm: $(2p-1)\cdot O(n/p) = O(n)$

$n \geq p$

comp $O(n^2/p)$	*comm* $O(n)$	*sync* $O(p)$

BSP LCS

Classical dynamic programming mapped directly onto ordered 2D grid

$comp = O(n^2/p)$ $comm = O(n)$ $sync = O(p)$

Parallel and dynamic string comparison

Bulk-synchronous parallel LCS

BSP LCS

Classical dynamic programming mapped directly onto ordered 2D grid

$\boxed{comp = O(n^2/p)}$ $\boxed{comm = O(n)}$ $\boxed{sync = O(p)}$

comm is not scalable (i.e. does not decrease with increasing p) :-(

LCS with scalable comm, e.g. $comm = O(n/p^{1/2})$?

LCS with $sync = O(\log p)$? $sync = O(1)$?

BSP LCS (cont.)

Parallel semi-local LCS: local comp, comm, sync

comp	*comm*	*sync*		
$O(n^2/p)$	$O(n)$	$O(p)$	global	2D grid + classical DP
↑	$O(n \log p)$	$O(\log p)$	str-substr	[Alves+: 2006]
↑	$O(n)$	$O(p)$	semi-local	2D grid + iter combing
↑	$O\left(\frac{n}{p^{1/2}}\right)$	$O(\log^2 p)$	↑	[Krusche, T: 2007]
↑	$O\left(\frac{n \log p}{p^{1/2}}\right)$	$O(\log p)$	↑	[Krusche, T: 2010]
↑	$O(n)$	$O(\log \log p)$	↑	[T: 2020]
↑	$O(np^{\epsilon})$	$O(\log(1/\epsilon))$	↑	[T: 2020]
↑	$O\left(\frac{n}{p^{1/2}}\right)$	$O(\log p)$	↑	[T: 2020]

BSP semi-local LCS, sequential kernel composition

Divide-and-conquer: partition $G_{a,b}$ into regular $p^{1/2} \times p^{1/2}$ grid of blocks

Ground phase: each processor assigned a subproblem of size $\frac{n}{p^{1/2}}$

- receives subproblem's input substrings a' of a, a'' of b
- solves subproblem: LCS kernel $P_{a',b'}$ of size $O\left(\frac{n}{p^{1/2}}\right)$

Conquer phase: a designated processor

- collects the p LCS kernels, total size $p \cdot O\left(\frac{n}{p^{1/2}}\right) = O(np^{1/2})$
- performs recursive combing ascent: LCS kernel $P_{a,b}$ of size $O(n)$

$$comp = O\left(\frac{n^2}{p}\right) + O(np^{1/2} \log n) = O\left(\frac{n^2}{p}\right)$$

$$comm = O\left(\frac{n}{p^{1/2}}\right) + O(np^{1/2}) = O(np^{1/2})$$

$$sync = O(1) + O(1) = O(1)$$

BSP semi-local LCS, sequential kernel composition (contd.)

Improving *comm* (with only slight deterioration in *sync*)

Conquer phase: the processors

- perform recursive combing ascent: LCS kernel $P_{a,b}$ of size $O(n)$
- $\log p$ ascent levels bundled into $\log\log p$ supersteps

$$comp = O\left(\frac{n^2}{p}\right) + O(\ldots + n\log n) = O\left(\frac{n^2}{p}\right)$$

$$comm = O\left(\frac{n}{p^{1/2}}\right) + O(\ldots + n) = O(n)$$

$$sync = O(1) + O(\log\log p) = O(\log\log p)$$

BSP semi-local LCS, sequential kernel composition (contd.)

Improving *comm* (with only slight or no deterioration in *sync*), alternative version

Let $\epsilon > 0$

Conquer phase: the processors

- perform recursive combing ascent: LCS kernel $P_{a,b}$ of size $O(n)$
- $\log p$ ascent levels bundled into $\log(1/\epsilon)$ supersteps, the final one dominates

$$comp = O\left(\frac{n^2}{p}\right) + O(\ldots + n \log n) = O\left(\frac{n^2}{p}\right)$$

$$comm = O\left(\frac{n}{p^{1/2}}\right) + O(np^{\epsilon}) = O(np^{\epsilon})$$

$$sync = O(1) + O(\log(1/\epsilon)) = O(\log(1/\epsilon))$$

Matrix sticky multiplication: parallel Steady Ant

$P \boxdot Q = R \qquad R^{\Sigma}(i,k) = \min_j\big(P^{\Sigma}(i,j) + Q^{\Sigma}(j,k)\big)$

Divide-and-conquer on the range of j

Divide phase: partition range of j into p subranges; induces partitioning of P, Q

Ground phase: each processor assigned a subproblem of size $\frac{n}{p}$

- receives subproblem's input permutation matrices
- solves subproblem: permutation matrix of size $\frac{n}{p}$

Conquer phase: $\log p$ levels; in each level

- partition subproblem's solution matrix R into a regular $p \times p$ grid of blocks
- sample R in block corners: total $(p+1)^2$ samples
- Steady Ant's trail passes through only $\leq 2p-1$ essential blocks

Matrix sticky multiplication: parallel Steady Ant (contd.)

Each processor

- assigned an essential block
- finds the trail's entry point into block; traces the trail

$$comp = O\left(\frac{n}{p}\right) + O\left(\frac{n \log n}{p}\right) + O\left(\frac{n \log p}{p}\right) = O\left(\frac{n \log n}{p}\right)$$

$$comm = O\left(\frac{n}{p}\right) + O\left(\frac{n}{p}\right) + O\left(\frac{n \log p}{p}\right) = O\left(\frac{n \log p}{p}\right)$$

$$sync = O(1) + O(1) + O(\log p) = O(\log p)$$

Matrix sticky multiplication: parallel Steady Ant (contd.)

Improving *sync* (at the expense of work-optimality)

- in conquer phase, total of $\leq 2p \log p$ essential blocks across all levels
- a processor recomputes the whole recursive ascent for its essential block

$$comp = O\left(\frac{n}{p}\right) + O\left(\frac{n \log n}{p}\right) + O\left(\frac{n \log^2 p}{p}\right) = O\left(\frac{n(\log n + \log^2 p)}{p}\right)$$

$$comm = O\left(\frac{n}{p}\right) + O\left(\frac{n}{p}\right) + O\left(\frac{n \log p}{p}\right) = O\left(\frac{n \log p}{p}\right)$$

$$sync = O(1) + O(1) + O(1) = O(1)$$

Work-optimality

- weakened to quasi-work-optimality
- preserved assuming superpolynomial slackness $n \geq 2^{(\log p)^2}$

BSP semi-local LCS, parallel kernel composition

Divide-and-conquer: partition $G_{a,b}$ into regular $p^{1/2} \times p^{1/2}$ grid of blocks

Ground phase: each processor assigned a subproblem of size $\frac{n}{p^{1/2}}$

- receives subproblem's input substrings a' of a, a'' of b
- solves subproblem: LCS kernel $P_{a',b'}$ of size $O\left(\frac{n}{p^{1/2}}\right)$

Conquer phase: $\log p$ levels of parallel matrix sticky multiplication

$$comp = O\left(\frac{n}{p^{1/2}}\right) + O\left(\frac{n^2}{p}\right) + O\left(\frac{n(\log n + \log^2 p)}{p^{1/2}}\right) = O\left(\frac{n^2}{p}\right)$$

$$comm = O\left(\frac{n}{p^{1/2}}\right) + O\left(\frac{n}{p^{1/2}}\right) + O\left(\frac{n}{p^{1/2}}\right) = O\left(\frac{n}{p^{1/2}}\right)$$

$$sync = O(1) + O(1) + O(\log p) = O(\log p)$$

Dynamic LCS problem

Maintain current LCS score under updates to one or both input strings

Both input strings are streams, updated on-line:

- inserting/deleting characters at left or right
- inserting/deleting characters at arbitrary position
- inserting/deleting substrings

Assume for simplicity $m \approx n$, i.e. $m = \Theta(n)$

Dynamic LCS: update time

in+del	right	$O(n)$		classical DP
in	left+right	$O(n)$	a fixed	[Landau+: 1998]
				[Kim, Park: 2004]
in	left+right	$O(n)$		[Ishida+: 2005]
in+del	left+right	$O(n)$	also semi-local	[T: 2008]
in+del	anywhere	$O(n(\log n)^2)$	also semi-local	
				[Charalampopoulos+: 2020]

Dynamic LCS ([Charalampopoulos+: 2020])

Maintain a hierarchy of LCS kernels for canonical substrings

Processing an update: $O(\log n)$ runs of kernel $\boxdot$-multiplication

Dynamic String Alignment

Panagiotis Charalampopoulos
Department of Informatics, King's College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Tomasz Kociumaka
Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Shay Mozes
Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Israel
smozes@idc.ac.il

Abstract

We consider the problem of dynamically maintaining an optimal alignment of two strings, each of length at most n, as they undergo insertions, deletions, and substitutions of letters. The string alignment problem generalizes the longest common subsequence (LCS) problem and the edit distance problem (also with non-unit costs, as long as insertions and deletions cost the same). The conditional lower bound of Backurs and Indyk [J. Comput. 2018] for computing the LCS in the static case implies that strongly sublinear update time for the dynamic string alignment problem would refute the Strong Exponential Time Hypothesis. We essentially match this lower bound when the alignment weights are constants, by showing how to process each update in $\tilde{\mathcal{O}}(n)$ time.[1] When the weights are integers bounded in absolute value by some $w = n^{\mathcal{O}(1)}$, we can maintain the alignment in $\tilde{\mathcal{O}}(n \cdot \min\{\sqrt{n}, w\})$ time per update. For the $\tilde{\mathcal{O}}(nw)$-time algorithm, we heavily rely on Tiskin's

1 Introduction

2 Unit-Monge matrices and sticky braids

3 Fundamentals of string comparison

4 Rational-weighted string comparison

5 Cyclic and periodic string comparison

6 Sparse string comparison

7 Compressed string comparison

8 Local string comparison

Conclusions

Surprising algebra of string comparison: LCS core; sticky braids; $\boxdot$-multiplication in time $O(n \log n)$

Semi-local LCS: simple efficient algorithms (recursive, iterative combing)

Alignment and edit distance: approximate matching; parameterised comparison

Further string comparison: dynamic LCS; LCS on periodic strings

Sparse string comparison: permutation strings; max-clique in a circle graph

Compressed string comparison: LCS and approximate matching on GC-strings (e.g. LZ78-compressed)

Local string comparison: window, fragment, spliced alignment; local LCS oracle; sweeping alignment

Parallel string comparison: bit-, vector-parallel; comm- and sync- efficient

Open problems

Open problems (theory):

- real-weighted alignment
- compressed LCS: more compression models
- parallel LCS: further improvement in comm, sync
- lower bounds, e.g. $\Omega(n \log n)$ for $\boxdot$-mult, $\Omega(n(\log n)^2)$ for local LIS
- LCS on random strings: Chvatal–Sankoff problem
- further connections, e.g. fast approximation for LCS or edit distance

Open problems (applications):

- more implementation effort on modern architectures
- good pre-filtering (e.g. q-grams)?
- further applications in biology, software engineering

Thank you

CS Space

Удивительная алгебра сравнения строк (часть 2)

А. В. Тискин

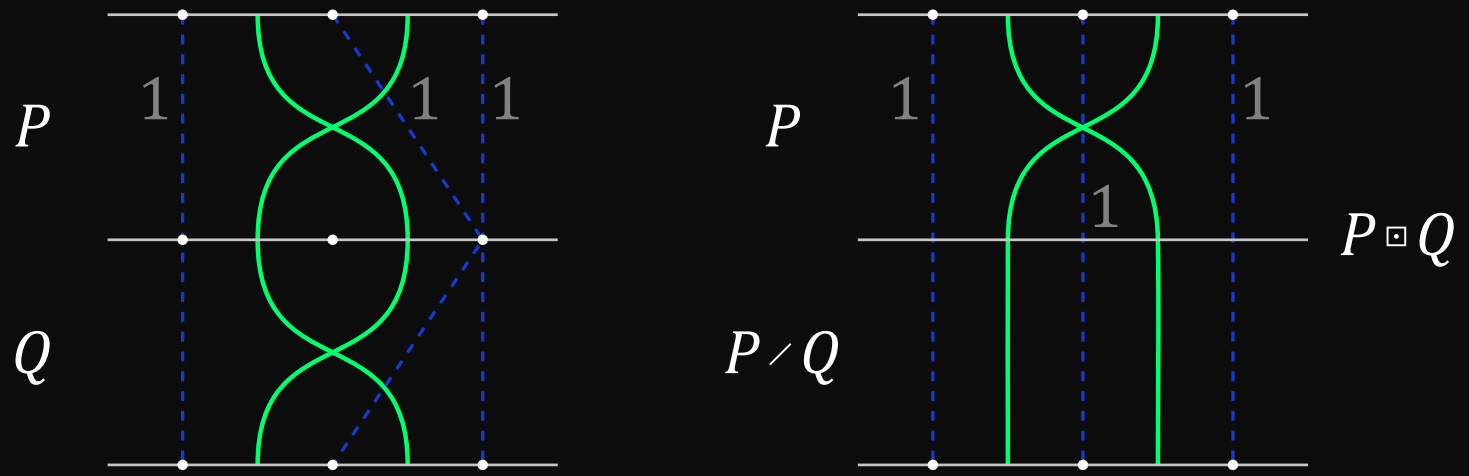
DPhil (Oxford), доцент МКН СПбГУ

Б. Золотов

аспирант МКН СПбГУ

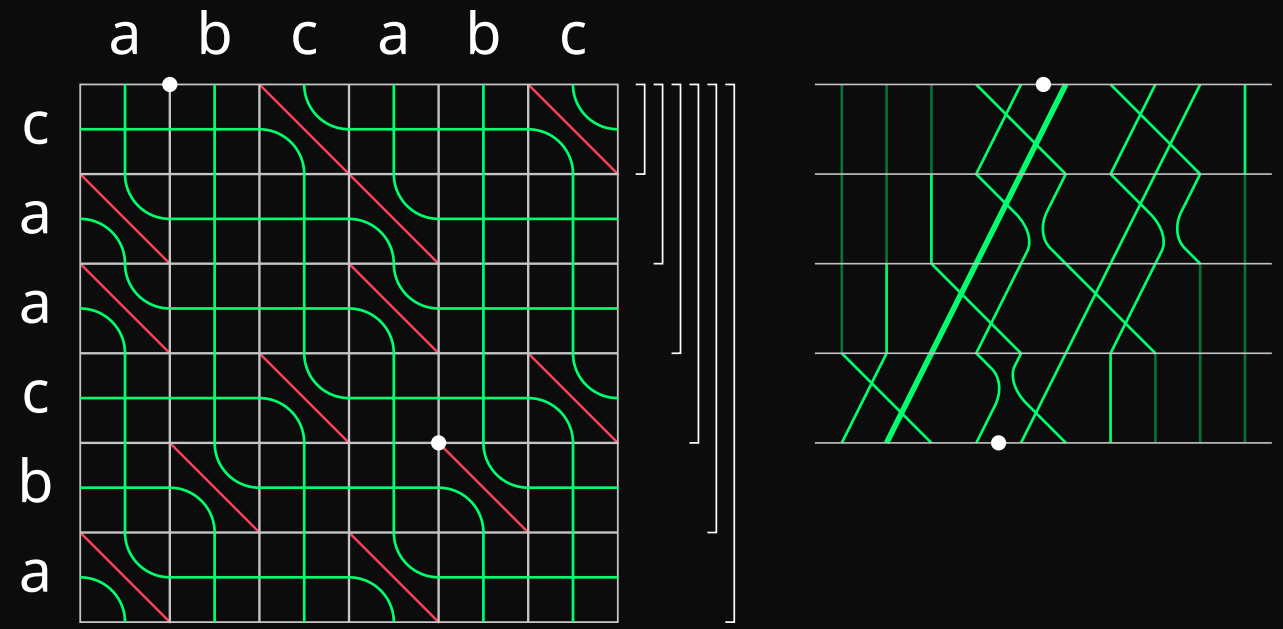
Соответствие между $\odot$ и $\boxdot$

Минимальный пример: $\boxdot$-квадрат транспозиции — это она же.



Dodrans-local LCS

Задача: для пары строк a, b предподсчитать оракул, который сможет быстро отвечать на запросы вида $\mathrm{LCS}(a[\mathbf{0} : i_1], b[j_0 : j_1])$.



Посчитаем все префиксные $\boxdot$-произведения перестановок, будем хранить в структуре данных, отвечающей на *range queries*.

Локальная задача LCS

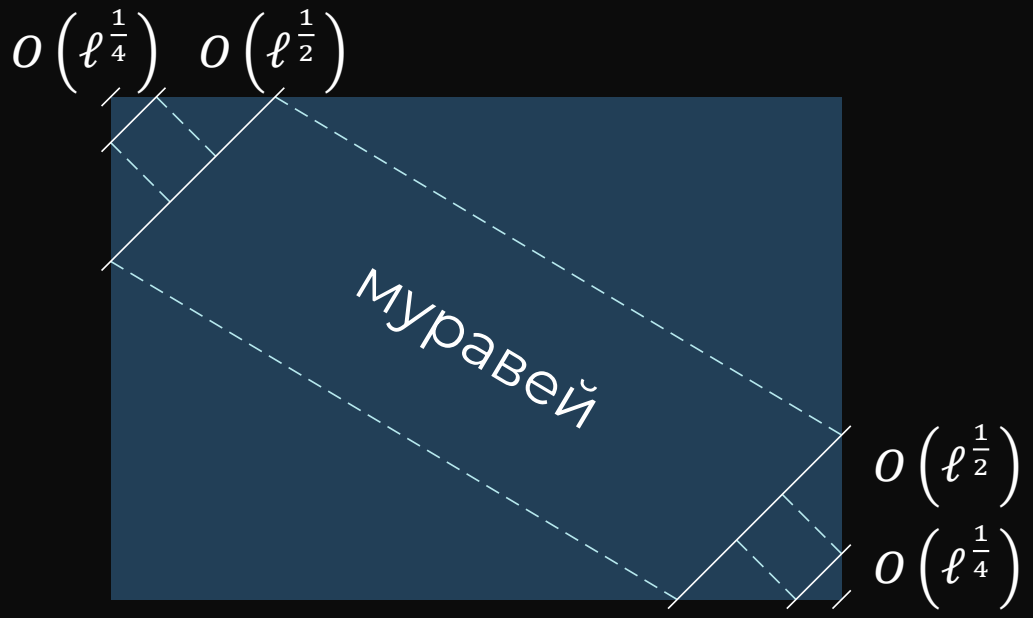
Задача: для пары строк a, b предподсчитать оракул, который сможет быстро отвечать на запросы вида $\mathrm{LCS}(a[i_0 : i_1], b[j_0 : j_1])$.

Запрос соответствует прямоугольнику на решётке, координаты вершин которого — $i_0, i_1; j_0, j_1$.

Local LCS: оракул Sakai

Предподсчёт за $\tilde{O}\left(n^2\right)$, запрос за $\tilde{O}\left(\sqrt{\ell}\right)$. *Sakai, 2022*
Здесь ℓ — размер прямоугольника, соотв. запросу.

Для уровней, делящихся на $\frac{n}{2^r}$, посчитаем перестановки между теми, разность которых не превосходит $\left(\frac{n}{2^r}\right)^2$.



Запрос — $\boxdot$-перемножение подперестановок и *range queries*.

Local LCS: оракул Ch+

Charalampopoulos, Gawrychowski, Mozes, Weimann, 2021.

Если хранить перестановки в *дереве отрезков*, то между противоположными вершинами прямоугольника будет $\log n$ шагов по перестановкам.

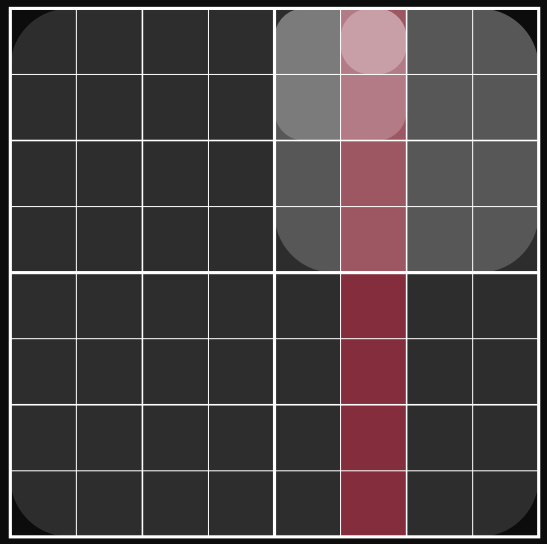
Трудность — выбор индекса, в который приходит очередной шаг.

Ch+ используют *чёрный ящик MSSP*, а *мы знаем*, как улучшить время работы, применив муравья.

Динамическое выравнивание

Задача: обновлять LCS при вставке/удалении символа произвольной из двух строк

Иерархия, в которой у каждого прямоугольника четыре потомка. Внутри каждого — посчитана перестановка.

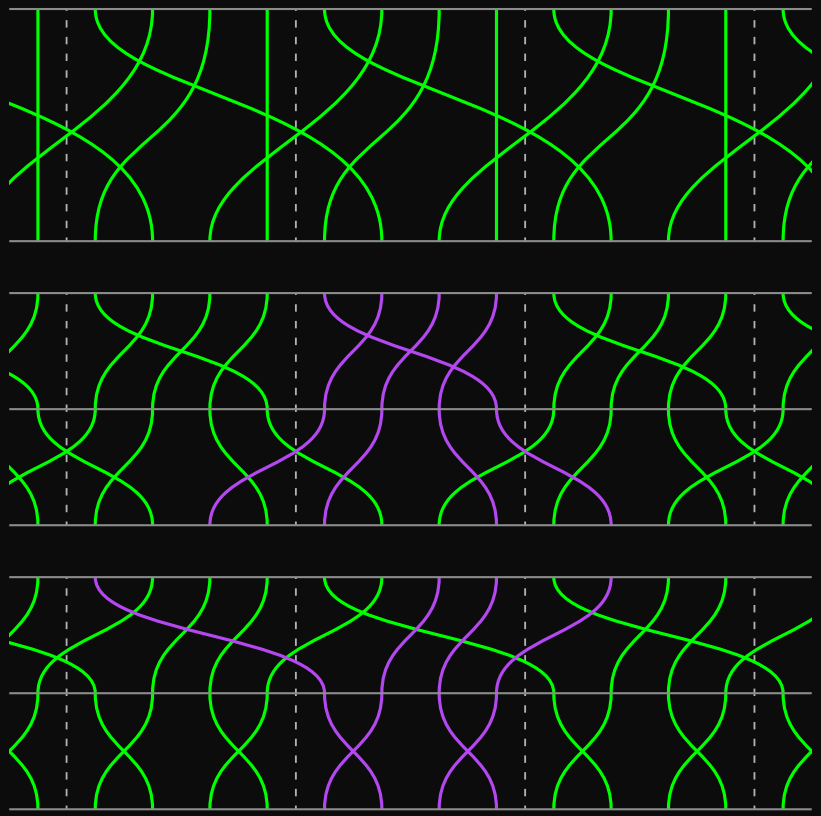


Суммарный размер перестановок, изменённых при вставке/удалении символа, на каждом уровне иерархии, — $O(n)$. *Charalampopoulos, Kociumaka, Mozes, 2020*

Аффинные перестановки

Научимся работать с перестановками в *аффинном моноиде Гекке*.

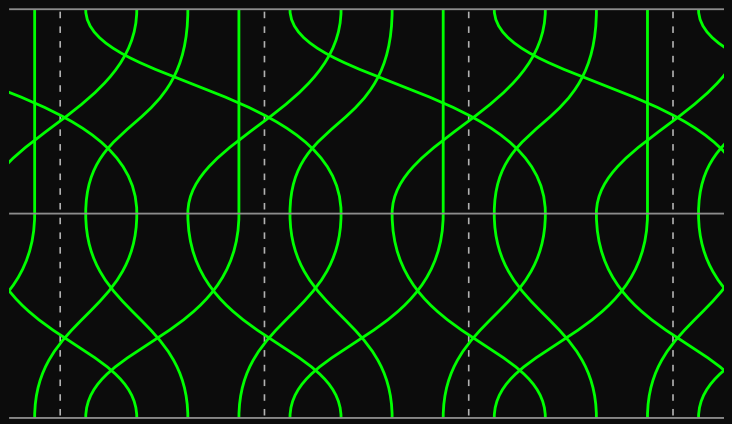
Gaevoy, Tiskin, Zolotov, 2025

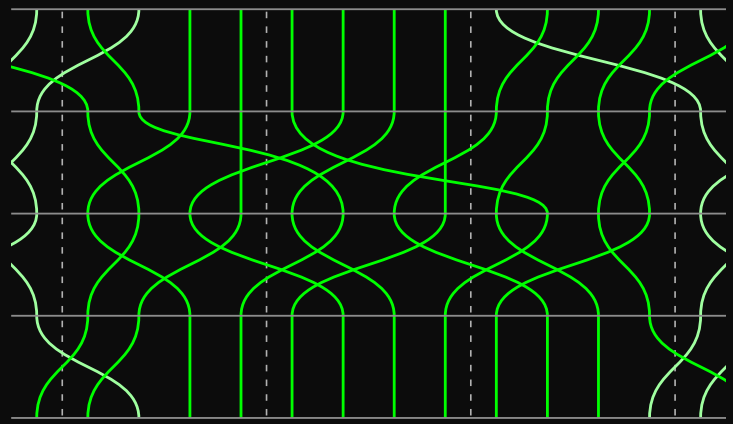


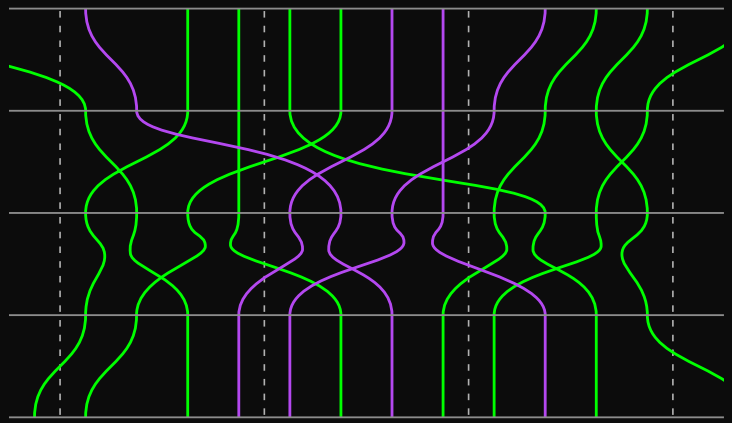
Аффинная перестановка раскладывается на произведение *перестановки конечного типа* и *грассмановой перестановки*.

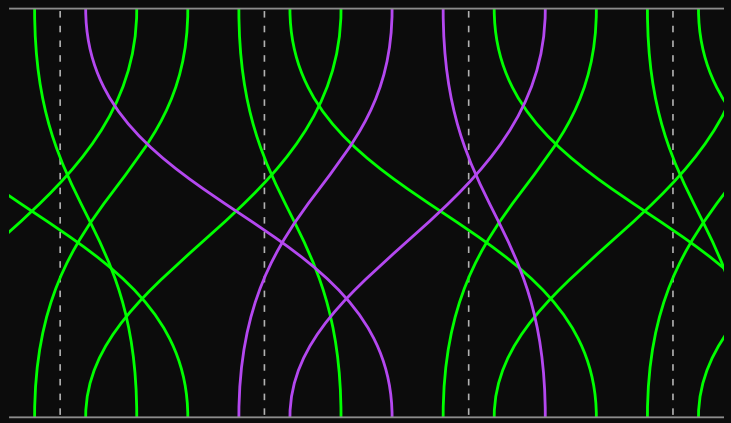
Аффинное ⊡-умножение

Можно свести ⊡-умножение аффинных перестановок к обычному ⊡ их *трёх соседних периодов*.









Периодическая задача LCS

Задача: найти LCS(a^k, b^m).

- Возвести перестановку в $\boxdot$-степень k;
- посчитать, сколько копий каждой нити пересекает m периодов.

Время — $O(|a| \cdot |b| \ + \ |b| \cdot \log |b| \cdot \log k)$.

Приближенный поиск подстрок

Charalampopoulos, Kociumaka, Wellnitz 2022

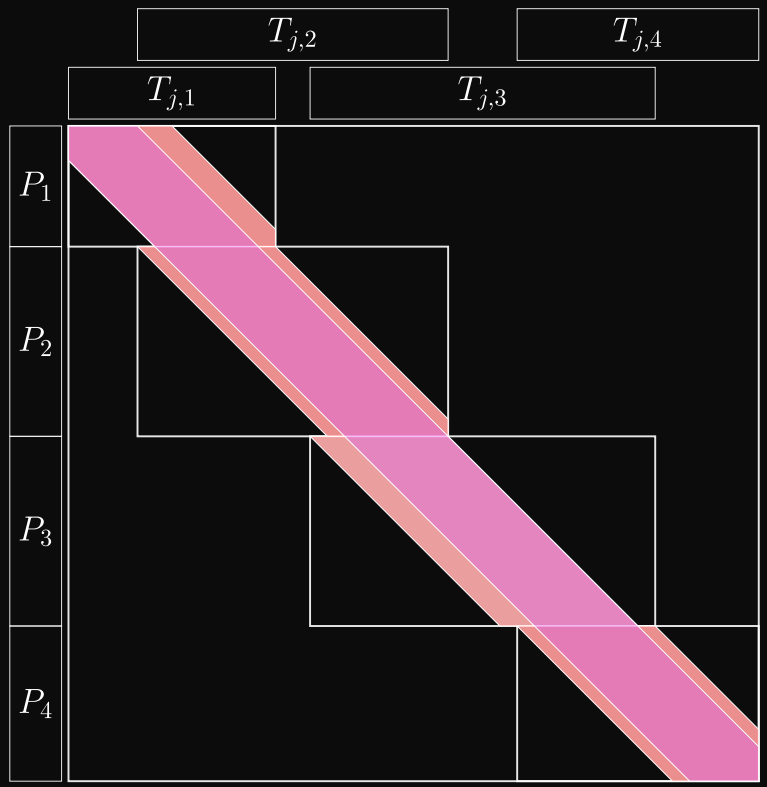
Задача: найти в тексте T подстроки, отличающиеся от шаблона P не более чем на редакционное расстояние k.

Ключевой шаг решения этой задачи — *dynamic puzzle matching* строк с малым редакционным расстоянием:

Дана эталонная строка U и семейство $\mathcal{F}$. Известно, что $\sum_{u \in \mathcal{F}} \delta(u, U) = O(k)$. Последовательность пар $(P_1, T_1) \dots (P_Z, T_Z)$; $P_i, T_i \in \mathcal{F}$; пары могут в неё динамически вставляться и удаляться. Поддерживать вхождения $P_1 \dots P_Z$ в $T_1 \dots T_Z$ с не более чем k редакциями, быстро обновлять при вставке/удалении пары.

Dynamic puzzle maching

Не отходим более чем на k от главной диагонали — монжевы матрицы расстояний размером $O(k)$.



Построение: малое суммарное δ — применим динамическое выравнивание, чтобы построить матрицы расстояний для всех возможных пар.

Спасибо за внимание!

- ***Dodrans-local LCS:*** храним много перестановок, полученных муравьём
- ***Local LCS:*** дерево отрезков из перестановок с операцией $\boxdot$
- ***Dynamic LCS:*** дерево отрезков из перестановок переменного размера
- ***Periodic LCS:*** аффинное $\boxdot$-умножение и его непосредственное применение
- ***Approximate pattern matching:*** применение динамического выравнивания при ограниченном редакционном расстоянии

https://t.me/boris_a_z