The algebra of string comparison: Computing with sticky braids

Alexander Tiskin

Department of Mathematics and Computer Science, St Petersburg University

Introduction

Overview

Longest common subsequence under string concatenation

$lcs($"RUMPLESTILTSKIN", "STEAK"$) = 3$
$lcs($"RUMPLESTILTSKIN", "STILTON"$) = 6$

$\rightsquigarrow lcs($"RUMPLESTILTSKIN", "STEAK" + "STILTON"$) = ?$

Standard approach: dynamic programming

Divide-and-conquer as an alternative?

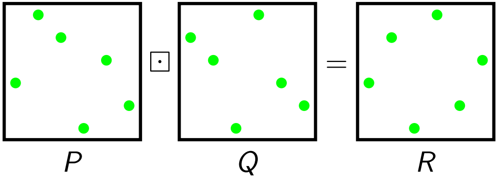
Introduction

Overview

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

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

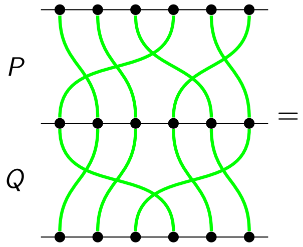
Permutation matrices under sticky multiplication



Introduction

Overview

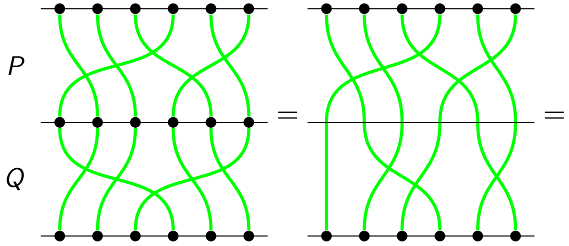
Sticky braids (a.k.a. Hecke words)



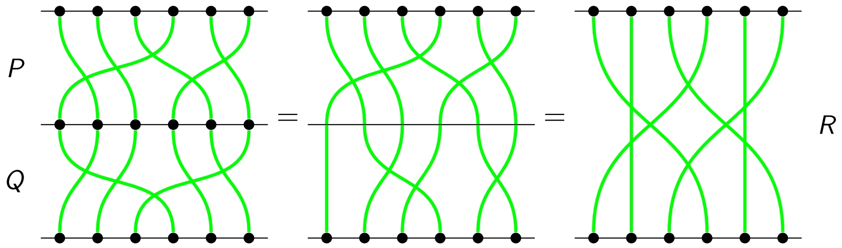
Introduction

Overview

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



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
- have connections to fields from computational geometry to combinatorics and statistical mechanics
- have practical applications in bioinformatics

Introduction
Preliminaries

"Squared paper" notation: $x^- = x - \frac{1}{2}$ $x^+ = x + \frac{1}{2}$

Integers $i, j \in \{\dots -2, -1, 0, 1, 2, \dots\} = [-\infty : +\infty] \supseteq [i, j]$

Half-integers $\hat{\imath}, \hat{\jmath} \in \{\dots -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots\} = \langle -\infty : +\infty \rangle \supseteq \langle i, j \rangle$

Planar dominance:

- $(i, j) \ll (i', j')$ iff $i < i'$ and $j < j'$ ("above-left" relation)
- $(i, j) \gtrless (i', j')$ iff $i > i'$ and $j < j'$ ("below-left" relation)

where "above/below" follow matrix convention (not the Cartesian one!)

Introduction
Preliminaries

Strings (= sequences) over an alphabet of size σ

Substrings (contiguous) vs subsequences (not necessarily contiguous)

Prefixes, suffixes (special cases of substring)

Algorithmic problems: input strings a, b of length m, n respectively

Introduction

Preliminaries

String matching: finding an exact pattern in a string

String comparison: finding similar patterns in two strings

- global: whole string a vs whole string b
- semi-local: whole string a vs substrings in b (approximate matching); prefixes in a vs suffixes in b
- local: substrings in a vs substrings in b

String matching: finding an exact pattern in a string

String comparison: finding similar patterns in two strings

- global: whole string a vs whole string b
- semi-local: whole string a vs substrings in b (approximate matching); prefixes in a vs suffixes in b
- local: substrings in a vs substrings in b

We focus on semi-local comparison:

- fundamentally important for both global and local comparison
- exciting mathematical properties

Introduction
Preliminaries

Standard approach to string comparison: dynamic programming

Our approach: the algebra of sticky braids

Can be used either iteratively, or recursively divide-and-conquer

Divide-and-conquer is more efficient for:

- comparing dynamic strings (truncation, concatenation)
- comparing compressed strings (e.g. LZ-compression)
- comparing strings in parallel

The "conquer" step involves a magic "superglue" (efficient sticky braid multiplication)

Monge and unit-Monge matrices

Monge matrices

Permutation matrix: 0/1 matrix with exactly one nonzero per row/column

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

Monge and unit-Monge matrices

Monge matrices

Permutation matrix: 0/1 matrix with exactly one nonzero per row/column

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

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

Monge and unit-Monge matrices

Monge matrices

Permutation matrix: 0/1 matrix with exactly one nonzero per row/column

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

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

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

Monge and unit-Monge matrices
Monge matrices

Permutation matrix: 0/1 matrix with exactly one nonzero per row/column

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

Cross-difference matrix (a.k.a. density matrix) of 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

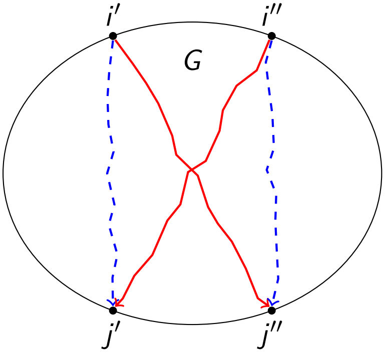
Matrix E is simple, if $(E^{\square})^{\Sigma} = E$: all zeros in left column and bottom row

Monge and unit-Monge matrices
Monge matrices

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

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

G. Monge (1746–1818)



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

Monge and unit-Monge matrices
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, an alignment graph for a pair of strings)

Monge and unit-Monge matrices
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, an alignment graph 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}

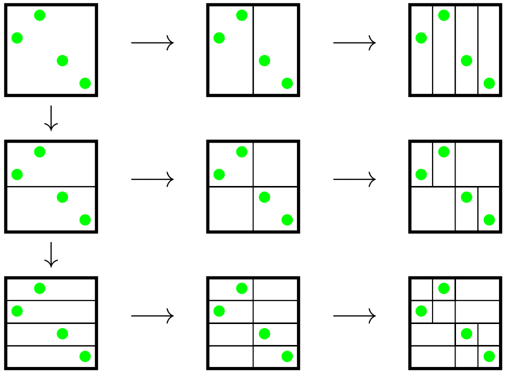
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\Sigma} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Monge and unit-Monge matrices

Implicit unit-Monge matrices

Efficient P^{Σ} queries: range tree on nonzeros of P [Bentley: 1980]

- binary search tree by i-coordinate
- under every node, binary search tree by j-coordinate



Monge and unit-Monge matrices

Implicit unit-Monge matrices

Efficient P^{Σ} queries: (contd.)

Every range tree node represents canonical rectangle (Cartesian product of canonical intervals), stores its nonzero count

Overall, $\leq n \log n$ canonical rectangles non-empty

P^{Σ} query

- $\gtrless$-dominance counting: $\sum$ nonzeros $\gtrless$-dominated by query point
- $= \sum$ nonzero counts in $\leq \log^2 n$ disjoint canonical rectangles

Total size $O(n \log n)$, query time $O(\log^2 n)$

Monge and unit-Monge matrices

Implicit unit-Monge matrices

Efficient P^Σ queries: (contd.)

Every range tree node represents canonical rectangle (Cartesian product of canonical intervals), stores its nonzero count

Overall, $\leq n \log n$ canonical rectangles non-empty

P^Σ query

- $\gtrless$-dominance counting: $\sum$ nonzeros $\gtrless$-dominated by query point
- $= \sum$ nonzero counts in $\leq \log^2 n$ disjoint canonical rectangles

Total size $O(n \log n)$, query time $O(\log^2 n)$

There are asymptotically more efficient (but less practical) data structures

Total size $O(n)$, query time $O\left(\frac{\log n}{\log \log n}\right)$ [JáJá+: 2004] [Chan, Pătraşcu: 2010]

Distance multiplication

Distance semiring (a.k.a. $(\min, +)$-semiring, tropical semiring)

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

Matrix distance multiplication

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

Intuition: shortest path distances in weighted graphs

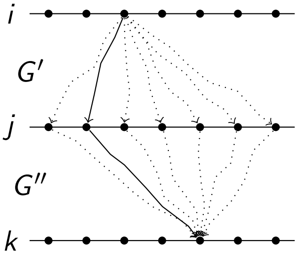
Distance multiplication

Distance 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



Distance multiplication

Recall: permutation matrices = implicit simple unit-Monge matrices

Matrix sticky multiplication (implicit distance multiplication)

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

The unit-Monge monoid $\mathcal{T}_n$:

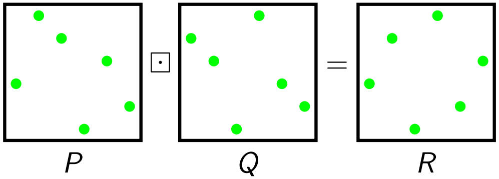
- permutation matrices under $\boxdot$
- simple unit-Monge matrices under $\odot$

Isomorphic to the Hecke monoid $H_0(\mathcal{S}_n)$

Distance multiplication

Sticky braids

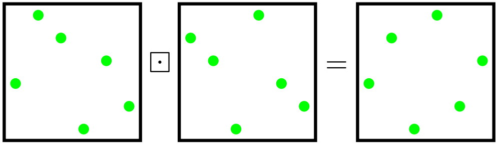
$P \boxdot Q = R$ can be seen as multiplication of sticky braids



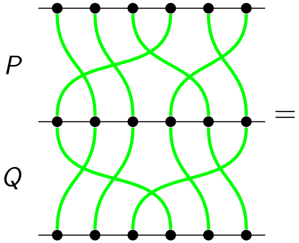
Distance multiplication

Sticky braids

$P \boxdot Q = R$ can be seen as multiplication of sticky braids



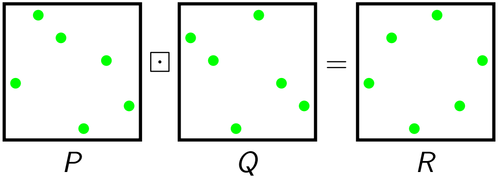
P Q R

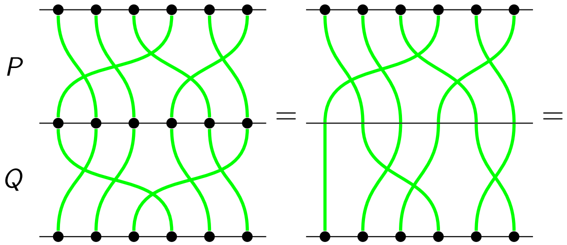


Distance multiplication

Sticky braids

$P \boxdot Q = R$ can be seen as multiplication of sticky braids

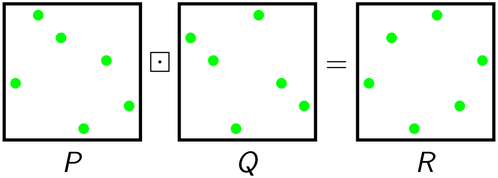


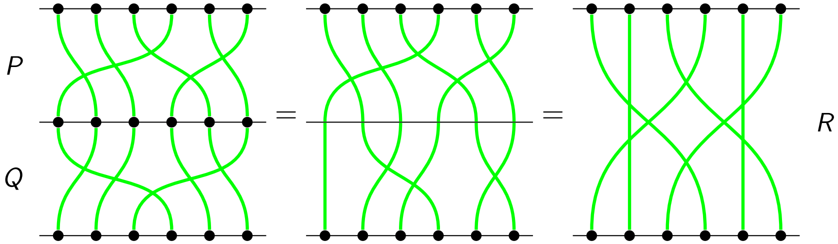


Distance multiplication

Sticky braids

$P \boxdot Q = R$ can be seen as multiplication of sticky braids



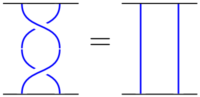


Distance multiplication
Sticky braids

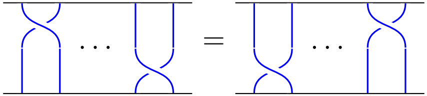
The classical braid group $\mathcal{B}_n$:

- $n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary crossings)
- ∞ elements canonical projection $\mathcal{B}_n \to \mathcal{S}_n$

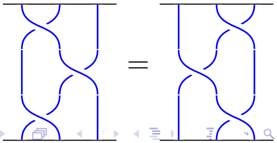
Inversion:
$g_i g_i^{-1} = 1$ for all i



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



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

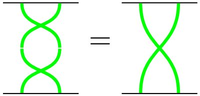


Distance multiplication
Sticky braids

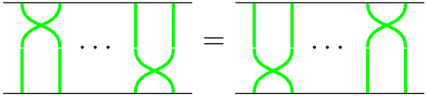
The sticky braid monoid $\mathcal{T}_n$:

- $n-1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary crossings)
- $n!$ elements canonical bijection $\mathcal{T}_n \leftrightarrow \mathcal{S}_n$

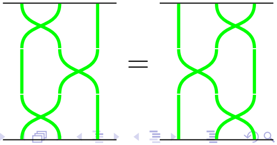
Idempotence:
$g_i^2 = g_i$ for all i



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



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



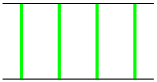
Distance multiplication

Sticky braids

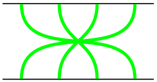
Special elements in $\mathcal{T}_n$

(Denote P^R = counterclockwise rotation of P)

Identity I: $I \boxdot x = x$

$I =$  $= \begin{bmatrix} \bullet & \cdot & \cdot & \cdot \\ \cdot & \bullet & \cdot & \cdot \\ \cdot & \cdot & \bullet & \cdot \\ \cdot & \cdot & \cdot & \bullet \end{bmatrix}$

Zero I^R: $I^R \boxdot x = I^R$

$I^R =$  $= \begin{bmatrix} \cdot & \cdot & \cdot & \bullet \\ \cdot & \cdot & \bullet & \cdot \\ \cdot & \bullet & \cdot & \cdot \\ \bullet & \cdot & \cdot & \cdot \end{bmatrix}$

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

Distance multiplication

Generalisations of sticky braids

- 0-Hecke monoid of a Coxeter group
- Hecke–Kiselman monoid
- $\mathcal{J}$-trivial monoid

Structures related to sticky braids

	idem?	far comm?	braid?
sticky braid monoid	$+$	$+$	$+$
positive braid monoid	no	$+$	$+$
locally free idemp. monoid	$+$	$+$	no
nil-Hecke monoid	$g_i^2 = 0$	$+$	$+$
symmetric group (Coxeter)	$g_i^2 = 1$	$+$	$+$
classical braid group	$g_i g_i^{-1} = 1$	$+$	$+$

Distance multiplication
Sticky braids

Computing with sticky braids

A confluent rewriting system can be obtained by software (GAP, SAGE): straightforward, but not too efficient…

Successive rewriting on words of length $O(n^2)$

Distance multiplication
Sticky braids

Computing with sticky braids

A confluent rewriting system can be obtained by software (GAP, Sage): straightforward, but not too efficient...

Successive rewriting on words of length $O(n^2)$

Denote $a, b, c, \ldots = g_1, g_2, g_3, \ldots$

$\mathcal{T}_3 = \{1, a, b, ab, ba, aba = 0\}$

$aa \to a$ $\quad bb \to b$ $\quad bab \to 0$ $\quad aba \to 0$

Distance multiplication

Sticky braids

Computing with sticky braids

A confluent rewriting system can be obtained by software (GAP, Sage): straightforward, but not too efficient...

Successive rewriting on words of length $O(n^2)$

Denote $a, b, c, \ldots = g_1, g_2, g_3, \ldots$

$\mathcal{T}_3 = \{1, a, b, ab, ba, aba = 0\}$

$aa \to a$	$bb \to b$	$bab \to 0$	$aba \to 0$

$\mathcal{T}_4 = \{1, a, b, c, ab, ac, ba, bc, cb, aba, abc, acb, bac, bcb, cba, abac, abcb, acba, bacb, bcba, abacb, abcba, bacba; abacba = 0\}$

$aa \to a$	$cc \to c$	$bab \to aba$	$cbac \to bcba$
$bb \to b$	$ca \to ac$	$cbc \to bcb$	$abacba \to 0$

Matrix sticky multiplication

Given permutation matrices P, Q, compute R, such that $P \boxdot Q = R$

Distance multiplication

Matrix sticky multiplication

Matrix sticky multiplication

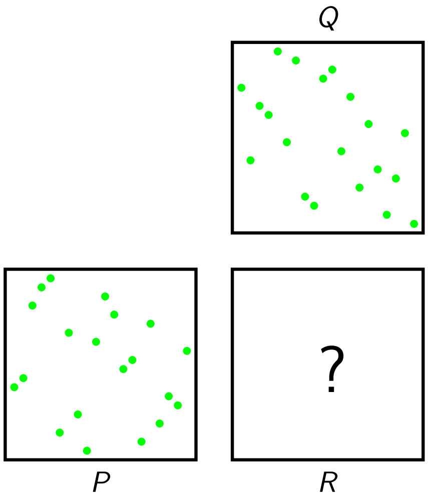
Given permutation matrices P, Q, compute R, such that $P \boxdot Q = R$

Matrix distance and sticky multiplication: 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]

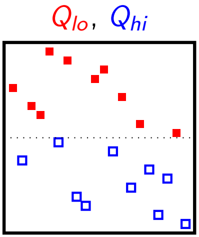
Distance multiplication

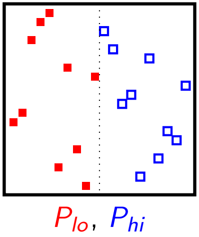
Matrix sticky multiplication



Distance multiplication

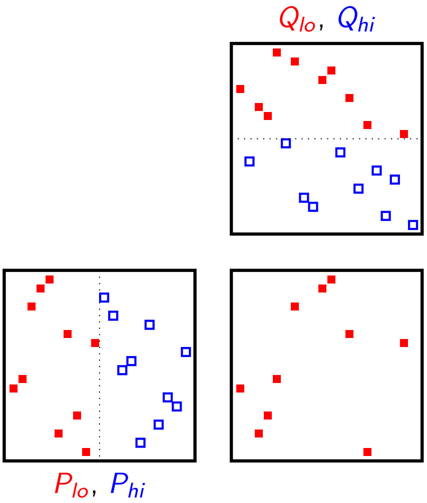
Matrix sticky multiplication





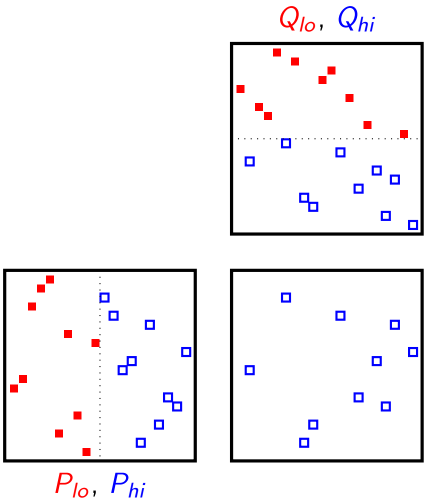
Distance multiplication

Matrix sticky multiplication



Distance multiplication

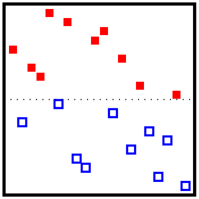
Matrix sticky multiplication

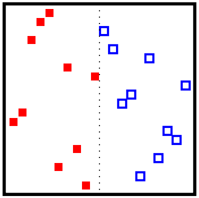


Distance multiplication

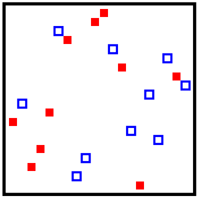
Matrix sticky multiplication

Q_{lo}, Q_{hi}





P_{lo}, P_{hi}

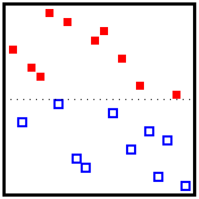


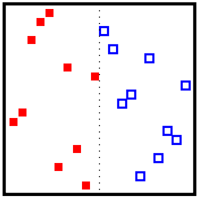
$R_{lo} + R_{hi}$

Distance multiplication

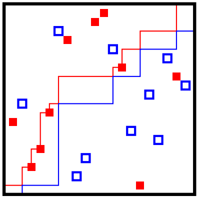
Matrix sticky multiplication

Q_{lo}, Q_{hi}





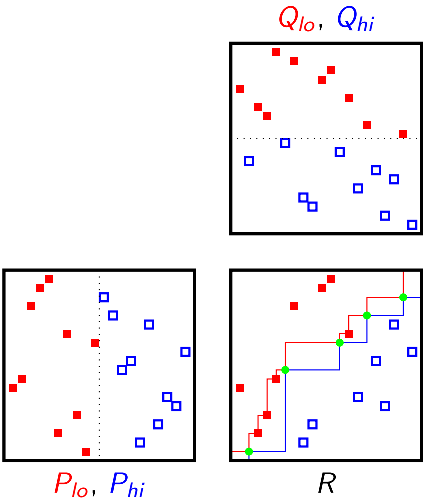
P_{lo}, P_{hi}



$R_{lo} + R_{hi}$

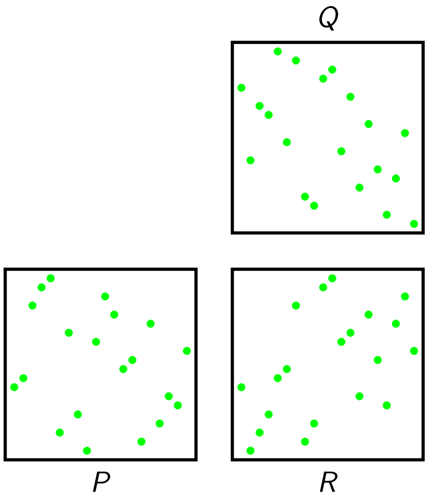
Distance multiplication

Matrix sticky multiplication



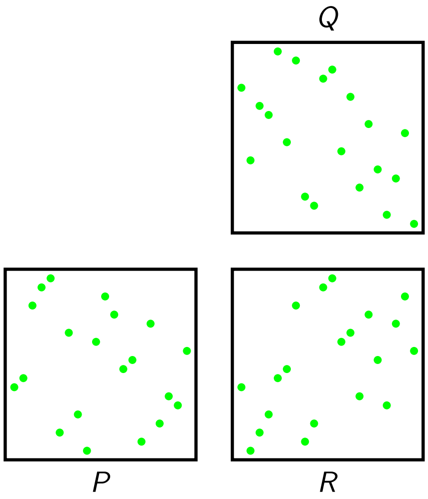
Distance multiplication

Matrix sticky multiplication



Distance multiplication

Matrix sticky multiplication



Distance multiplication

Matrix sticky multiplication

Matrix sticky multiplication: Steady Ant algorithm

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

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
$\lvert\{\text{nonzeros of } R_{hi} \text{ above-left}\}\rvert = \lvert\{\text{nonzeros of } R_{lo} \text{ below-right}\}\rvert$

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 $O(n \log n)$

Distance multiplication

Bruhat order

$P \preceq Q$ in the Bruhat order, if $Q \rightsquigarrow P$ by successive pair sorting

$$\begin{bmatrix} & \vdots & & \vdots & \\ \cdots & 0 & \cdots & 1 & \cdots \\ & \vdots & & \vdots & \\ \cdots & 1 & \cdots & 0 & \cdots \\ & \vdots & & \vdots & \end{bmatrix} \leftrightarrow \begin{bmatrix} & \vdots & & \vdots & \\ \cdots & 1 & \cdots & 0 & \cdots \\ & \vdots & & \vdots & \\ \cdots & 0 & \cdots & 1 & \cdots \\ & \vdots & & \vdots & \end{bmatrix}$$

Intuitively, $P \preceq Q$ means "P is more sorted than Q"

$I \preceq X \preceq I^R$

Distance multiplication

Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \preceq \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} \qquad \begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \not\preceq \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}$$

Bruhat order is compatible with sticky multiplication: $P \preceq Q$ implies $R \boxdot P \preceq R \boxdot Q$ and $P \boxdot R \preceq Q \boxdot R$

Bruhat order

- pivoting order in Gaussian elimination (Bruhat decomposition)
- generalises to Coxeter groups/monoids
- plays an important role in group theory and algebraic geometry (inclusion order of Schubert varieties)

Bruhat dominance: running time

$O(n^2)$	[Ehresmann: 1934; Proctor: 1982; Grigoriev: 1982]
$O(n \log n)$	[T: 2013]
$O\left(\frac{n \log n}{\log \log n}\right)$	[Gawrychowski: 2013]

Distance multiplication

Bruhat order

Ehresmann's criterion (dot criterion, related to tableau criterion)

$P \preceq Q$ iff $P^\Sigma \leq Q^\Sigma$ elementwise

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

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^\Sigma = \begin{bmatrix}0&1&2&3\\0&0&1&2\\0&0&1&1\\0&0&0&0\end{bmatrix} \not\leq \begin{bmatrix}0&1&2&3\\0&1&1&2\\0&0&0&1\\0&0&0&0\end{bmatrix} = \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}^\Sigma$$

Time $O(n^2)$

Distance multiplication

Bruhat order

Sticky product criterion

$P \preceq Q$ iff $P^R \boxdot Q = I^R$ iff $P \boxdot Q^{RRR} = I^R$

$P^R =$ counterclockwise rotation of P

Proof: $P \preceq Q$ implies $I^R = P^R \boxdot P \preceq P^R \boxdot Q$, hence $I^R = P^R \boxdot Q$

Conversely, if $P^R \boxdot Q = I^R$ and $P \neq Q$, then a pair can be sorted in Q

Time $O(n \log n)$ by matrix sticky multiplication

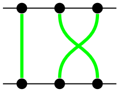
Distance multiplication

Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \preceq \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix} = I^R$$

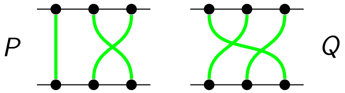
P



Distance multiplication
Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \preceq \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix} = I^R$$

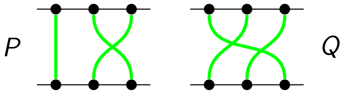


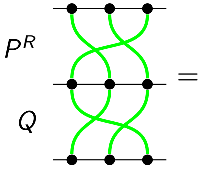
Distance multiplication

Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \preceq \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix} = I^R$$



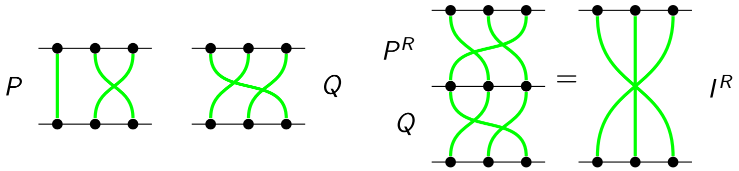


Distance multiplication

Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \preceq \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix} = \begin{bmatrix}0&0&1\\0&1&0\\1&0&0\end{bmatrix} = I^R$$

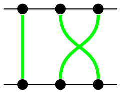


Distance multiplication
Bruhat order

$$\begin{bmatrix}1 & 0 & 0\\0 & 0 & 1\\0 & 1 & 0\end{bmatrix} \not\leq \begin{bmatrix}0 & 1 & 0\\1 & 0 & 0\\0 & 0 & 1\end{bmatrix}$$

$$\begin{bmatrix}1 & 0 & 0\\0 & 0 & 1\\0 & 1 & 0\end{bmatrix}^R \boxdot \begin{bmatrix}0 & 1 & 0\\1 & 0 & 0\\0 & 0 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\0 & 0 & 1\\1 & 0 & 0\end{bmatrix} \boxdot \begin{bmatrix}0 & 1 & 0\\1 & 0 & 0\\0 & 0 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\0 & 0 & 1\\1 & 0 & 0\end{bmatrix} \neq I^R$$

P

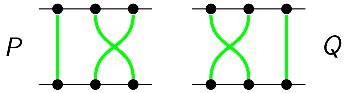


Distance multiplication

Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \not\preceq \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \neq I^R$$

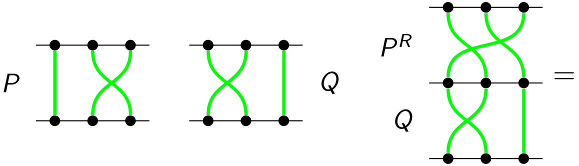


Distance multiplication

Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \not\preceq \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \neq I^R$$



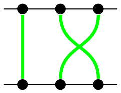
Distance multiplication

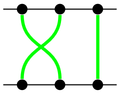
Bruhat order

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix} \not\preceq \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\0&1&0\end{bmatrix}^R \boxdot \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \boxdot \begin{bmatrix}0&1&0\\1&0&0\\0&0&1\end{bmatrix} = \begin{bmatrix}0&1&0\\0&0&1\\1&0&0\end{bmatrix} \neq I^R$$

P

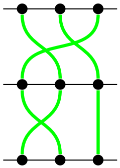




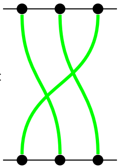
Q

P^R

Q



$=$



$\neq I^R$

Distance multiplication
Bruhat order

Alternative solution: clever implementation of Ehresmann's criterion [Gawrychowski: 2013]

The online partial sums problem: maintain array $X[1:n]$, subject to

- *update*(k, Δ): $X[k] \leftarrow X[k] + \Delta$
- *prefixsum*(k): return $\sum_{1 \le i \le k} X[i]$

Query time:

- $\Theta(\log n)$ in semigroup or group model
- $\Theta\left(\frac{\log n}{\log \log n}\right)$ in RAM model on integers [Pătrașcu, Demaine: 2004]

Gives Bruhat comparability in time $O\left(\frac{n \log n}{\log \log n}\right)$ in RAM model

Open problem: sticky multiplication in time $O\left(\frac{n \log n}{\log \log n}\right)$?

Longest common subsequence

LCS

The longest common subsequence (LCS) score:

- length of longest string that is a subsequence of both a and b
- equivalently, alignment score, where $\mathit{score}(\mathit{match}) = 1$ and $\mathit{score}(\mathit{mismatch}) = 0$

In biological terms, "loss-free alignment" (vs efficient but "lossy" BLAST)

Longest common subsequence
LCS

LCS problem

LCS score for a vs b

Longest common subsequence
LCS

LCS problem

LCS score for a vs b

LCS: running time

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

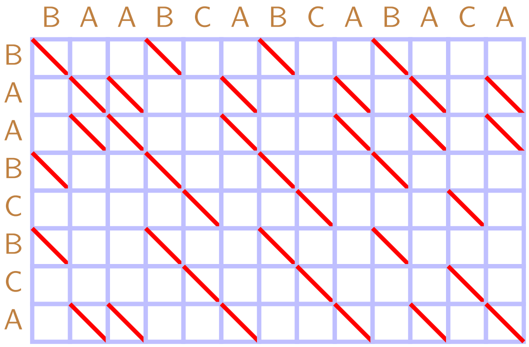
$O\left(\frac{mn}{(\log n)^c}\right)$ [Masek, Paterson: 1980] [Crochemore+: 2003]
[Paterson, Dančík: 1994] [Bille, Farach-Colton: 2008]

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

LCS in time $O(n^{2-\epsilon})$, $\epsilon > 0$, $m = n$: impossible unless SETH false
[Abboud+: 2015] [Backurs, Indyk: 2015]

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



$\textit{blue} = 0$

$\textit{red} = 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

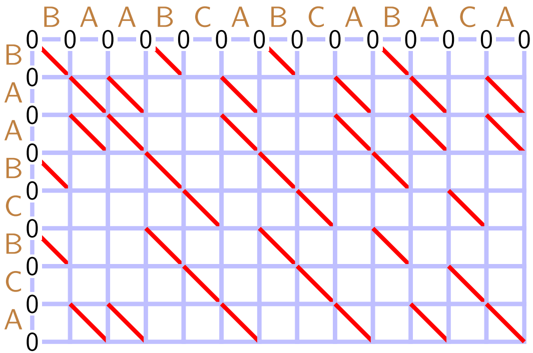
$lcs(a, \emptyset) = 0$

$lcs(\emptyset, b) = 0$

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

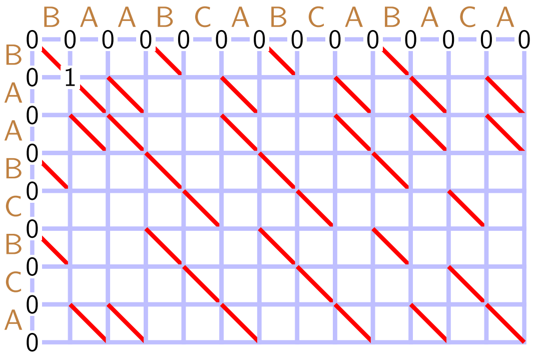
$lcs(a, \emptyset) = 0$

$lcs(\emptyset, b) = 0$

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

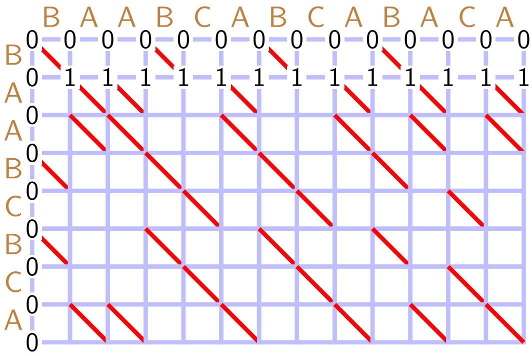
$lcs(a, \emptyset) = 0$

$lcs(\emptyset, b) = 0$

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



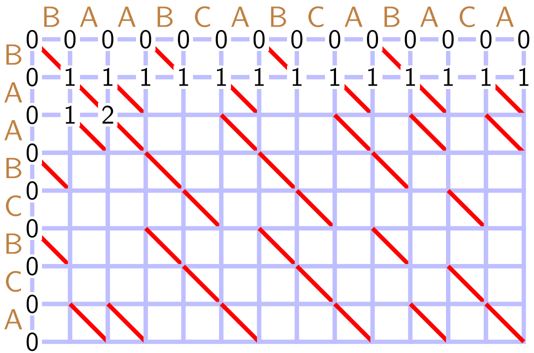
$$\textit{blue} = 0$$
$$\textit{red} = 1$$
$$a = \text{“BAABCBCA”}$$
$$b = \text{“BAABCABCABACA”}$$
$$lcs(a, b) = 8$$

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

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

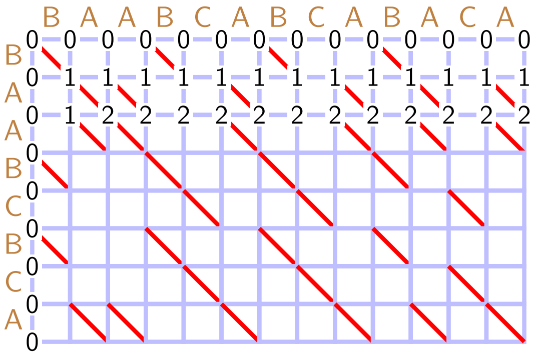
$$lcs(a, \emptyset) = 0$$

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

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

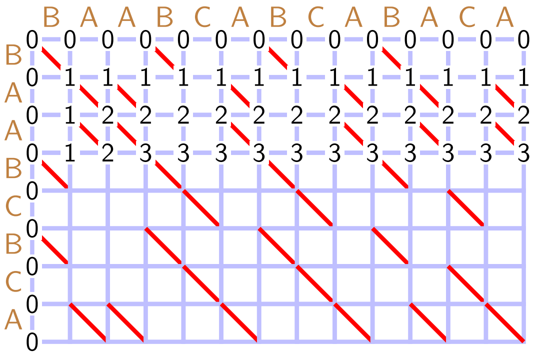
$lcs(a, \emptyset) = 0$

$lcs(\emptyset, b) = 0$

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

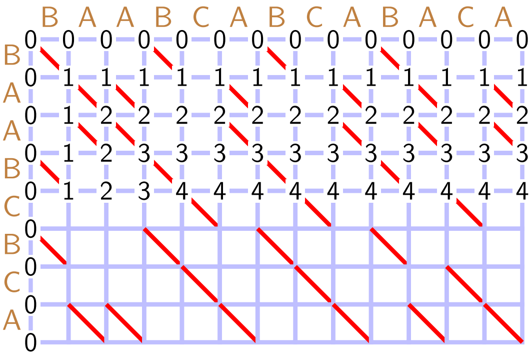
$lcs(\emptyset, b) = 0$

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

Longest common subsequence

LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

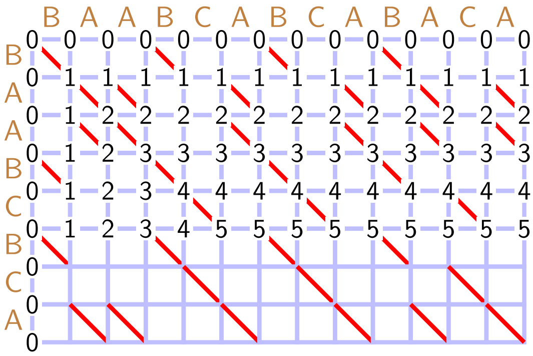
$$lcs(a, \emptyset) = 0$$

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

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



blue $= 0$

red $= 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

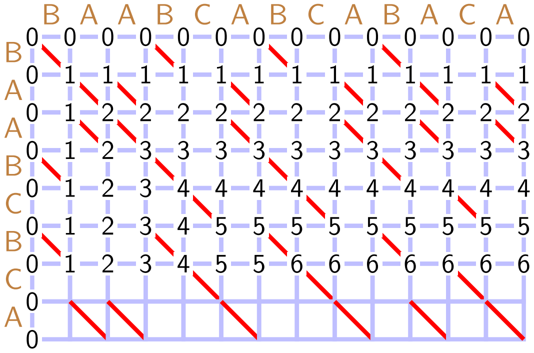
$lcs(\emptyset, b) = 0$

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

Longest common subsequence

LCS

LCS computation by classical dynamic programming (DP)



$\mathit{blue} = 0$

$\mathit{red} = 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

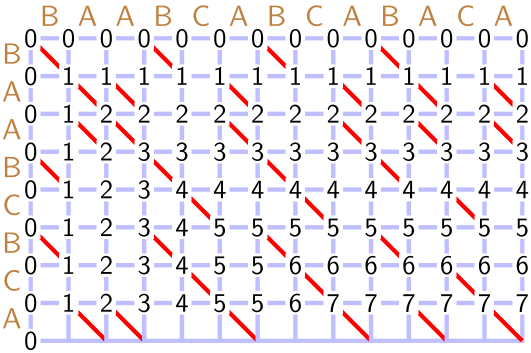
$$lcs(a, \emptyset) = 0$$

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

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



		B	A	A	B	C	A	B	C	A	B	A	C	A
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B	0	1	1	1	1	1	1	1	1	1	1	1	1	1
A	0	1	2	2	2	2	2	2	2	2	2	2	2	2
A	0	1	2	3	3	3	3	3	3	3	3	3	3	3
B	0	1	2	3	4	4	4	4	4	4	4	4	4	4
C	0	1	2	3	4	5	5	5	5	5	5	5	5	5
B	0	1	2	3	4	5	5	6	6	6	6	6	6	6
C	0	1	2	3	4	5	5	6	7	7	7	7	7	7
A	0													

blue = 0

red = 1

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

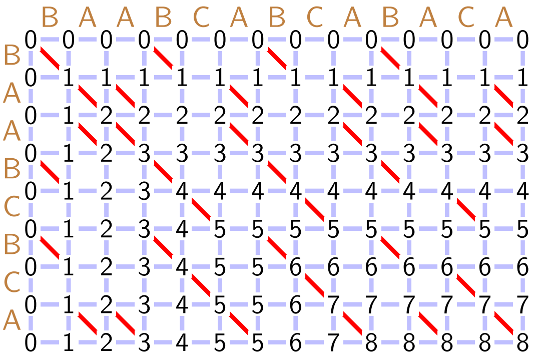
$$lcs(a, \emptyset) = 0$$

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

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

Longest common subsequence
LCS

LCS computation by classical dynamic programming (DP)



$\textit{blue} = 0$

$\textit{red} = 1$

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

$lcs(a, \emptyset) = 0$

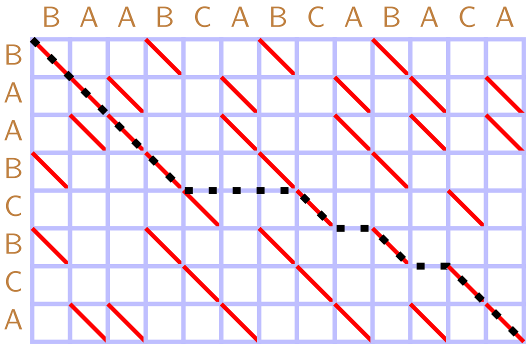
$lcs(\emptyset, b) = 0$

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

Longest common subsequence

LCS

LCS as a maximum path the LCS grid



blue = 0

red = 1

$a =$ "BAABCBCA"

$b =$ "BAABCABCABACA"

$lcs(a, b) = 8$

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

Longest common subsequence
LCS

LCS: classical dynamic programming (DP)

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

Active cell update: time $O(1)$

Overall time $O(mn)$

LCS: DP, micro-block speedup (MBS) [MP: 1980; BF: 2008]

Sweep cells in micro-blocks, in any $\ll$-compatible order

Micro-block size:

- $t = O(\log n)$ when $\sigma = O(1)$
- $t = O\left(\frac{\log n}{\log \log n}\right)$ otherwise

Micro-block interface:

- $O(t)$ characters, each $O(\log \sigma)$ bits, can be reduced to $O(\log t)$ bits
- $O(t)$ small integers, each $O(1)$ bits

Micro-block update: time $O(1)$, by precomputing all possible interfaces

Overall time $O\left(\frac{mn}{\log^2 n}\right)$ when $\sigma = O(1)$, $O\left(\frac{mn(\log \log n)^2}{\log^2 n}\right)$ otherwise

Longest common subsequence
Discussion

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

L. Carroll, *Alice in Wonderland*

Longest common subsequence
Discussion

'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:

- prepending/deleting characters (dynamic LCS)
- concatenating strings (LCS on compressed strings; parallel LCS)
- taking substrings (= local alignment)

Longest common subsequence

Discussion

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

Is dynamic programming strictly necessary to solve sequence alignment problems?

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

Longest common subsequence
Semi-local LCS

Semi-local LCS 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)

$O\big((m+n)^2\big)$ output scores can be represented implicitly

Longest common subsequence

Semi-local LCS

Semi-local LCS problem

LCS scores for a vs b:

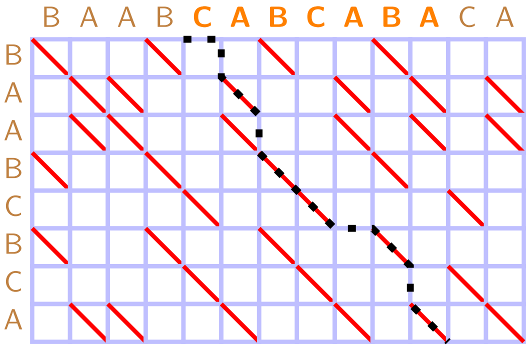
- string-substring (whole a vs every substring of b)
- prefix-suffix (every prefix of a vs every suffix of b)
- suffix-prefix (every suffix of a vs every prefix of b)
- substring-string (every substring of a vs whole b)

$O\big((m+n)^2\big)$ output scores can be represented implicitly

Longest common subsequence

Semi-local LCS

Semi-local LCS as maximum paths in the LCS grid



blue = 0

red = 1

a = "BAABCBCA"

b = "BAAB**CABCABA**CA"

$lcs(a, b[4:11]) = 5$

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

Semi-local LCS: all highest-score boundary-to-boundary paths

Longest common subsequence

Semi-local LCS

The LCS matrix H

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"

$H[i,j] = lcs(a, b\langle i : j\rangle)$

$H[4,11] = 5$

$H[i,j] = j - i$ if $i > j$

Semi-local LCS: 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]

Longest common subsequence

Semi-local LCS

The LCS matrix H and the LCS kernel P

$H(i, j)$: the number of matched characters for a vs substring $b\langle i : j\rangle$

$j - i - H(i, j)$: the number of unmatched characters

Properties of matrix $j - i - H(i, j)$:

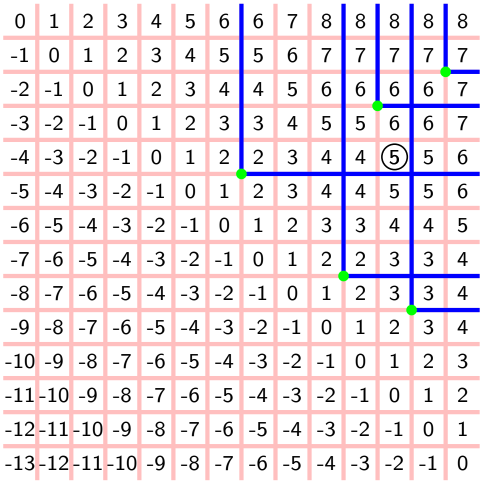
- simple unit-Monge
- therefore, $\ = P^{\Sigma}$, where $P = -H^{\square}$ is a permutation matrix

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

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

Longest common subsequence
Semi-local LCS

The LCS matrix H and the LCS kernel P



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"

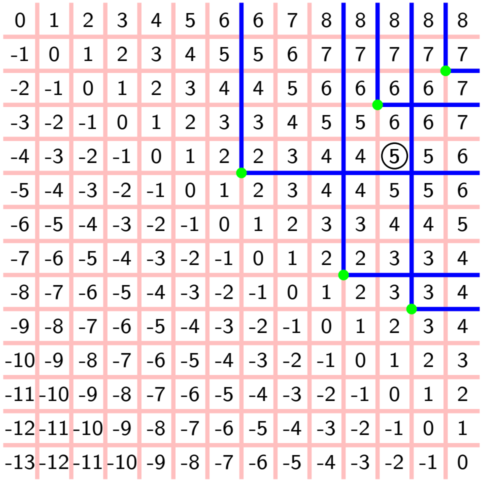
$H[i,j] = lcs(a, b\langle i : j \rangle)$

$H[4,11] = 5$

$H[i,j] = j - i$ if $i > j$

Longest common subsequence
Semi-local LCS

The LCS matrix H and the LCS kernel P



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”

$H[i,j] = lcs(a, b\langle i : j\rangle)$

$H[4,11] = 5$

$H[i,j] = j - i$ if $i > j$

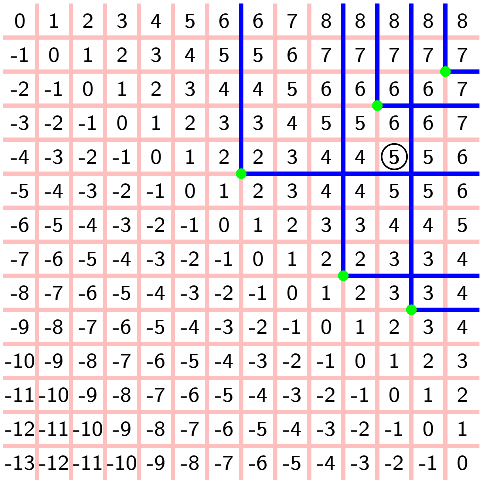
blue: difference in H is 0

red: difference in H is 1

Longest common subsequence

Semi-local LCS

The LCS matrix H and the LCS kernel P



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"

$H[i,j] = lcs(a, b\langle i : j \rangle)$

$H[4, 11] = 5$

$H[i,j] = j - i$ if $i > j$

blue: difference in H is 0

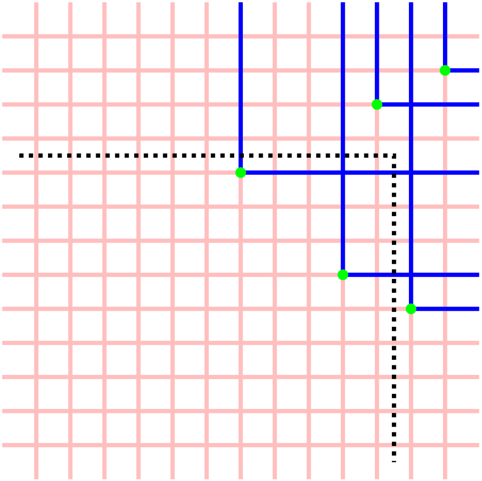
red: difference in H is 1

green: $P(i,j) = 1$

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

Longest common subsequence
Semi-local LCS

The LCS matrix H and the LCS kernel P



$a =$ "BAABCBCA"

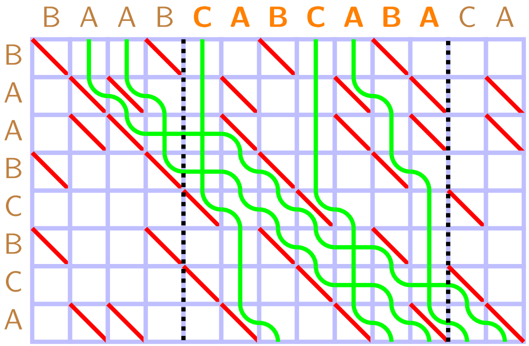
$b =$ "BAAB**CABCABA**CA"

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

Longest common subsequence

Semi-local LCS

The (reduced) sticky braid in the LCS grid



$a =$ "BAABCBCA"

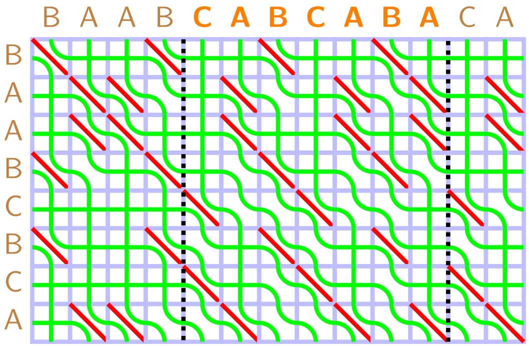
$b =$ "BAAB**CABCABA**CA"

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

$P\langle i,j\rangle = 1$ corresponds to strand *top* $i \rightsquigarrow$ *bottom* j

Longest common subsequence
Semi-local LCS

The (reduced) sticky braid in the LCS grid



$a =$ "BAABCBCA"

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

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

$P\langle i, j\rangle = 1$ corresponds to strand *top* $i \rightsquigarrow$ *bottom* j

Also define strands *top* $\rightsquigarrow$ *right*, *left* $\rightsquigarrow$ *right*, *left* $\rightsquigarrow$ *bottom*

Represent implicitly semi-local LCS for each prefix of a vs b

Longest common subsequence

Semi-local LCS

Sticky braid: a highly symmetric object (Hecke word $\in H_0(S_n)$)

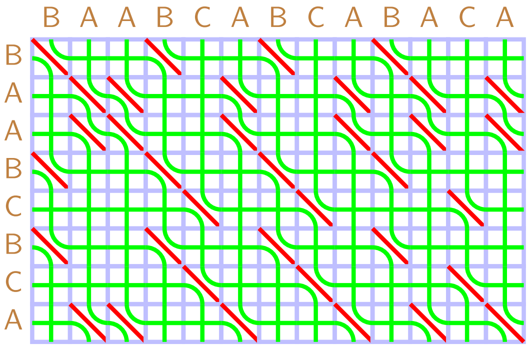
Can be built by assembling subbraids: divide-and-conquer

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

Longest common subsequence

Algorithms for semi-local LCS

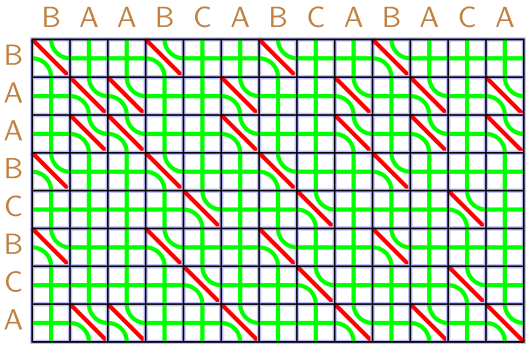
Semi-local LCS by recursive combing



Longest common subsequence

Algorithms for semi-local LCS

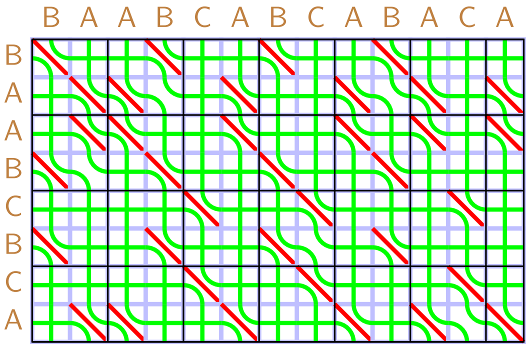
Semi-local LCS by recursive combing



Longest common subsequence

Algorithms for semi-local LCS

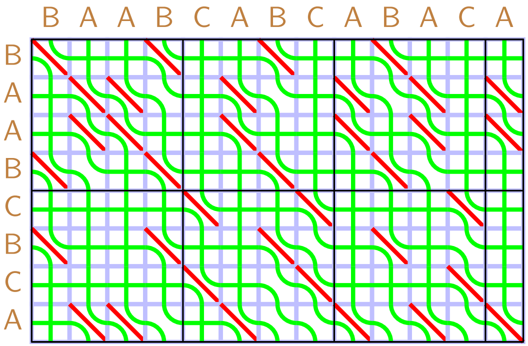
Semi-local LCS by recursive combing



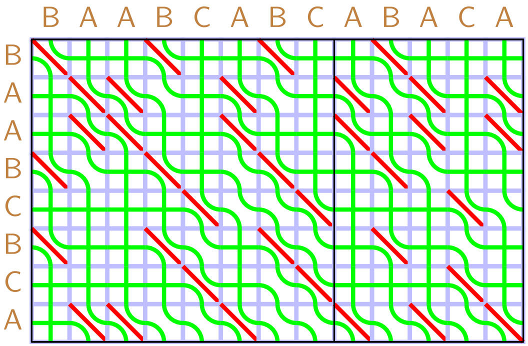
Longest common subsequence

Algorithms for semi-local LCS

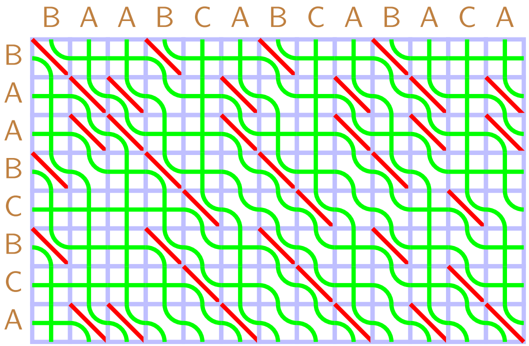
Semi-local LCS by recursive combing



Semi-local LCS by recursive combing



Semi-local LCS by recursive combing



Semi-local LCS: recursive combing

Initialise uncombed sticky braid: mismatch cell = crossing

Recursion on LCS grid

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

Recursion base: $m = n = 1$

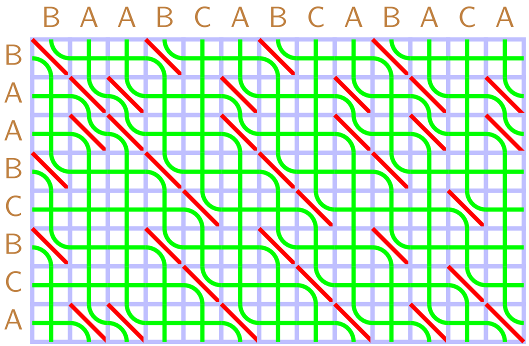
Overall time $O(mn)$

Correctness: by sticky braid relations

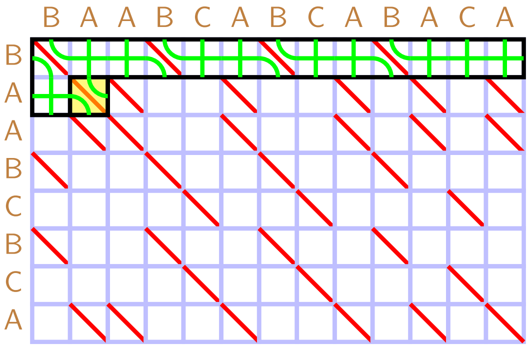
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing



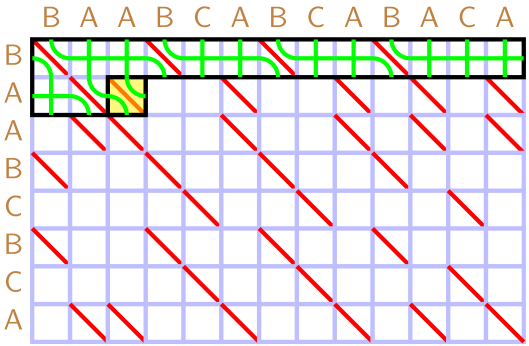
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

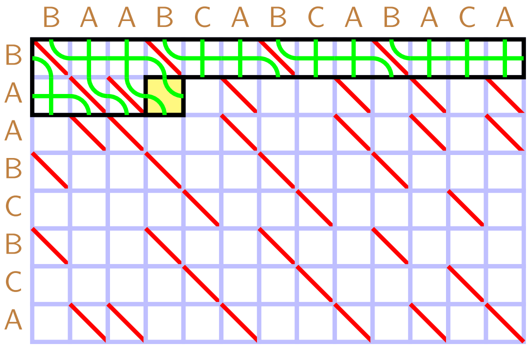
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

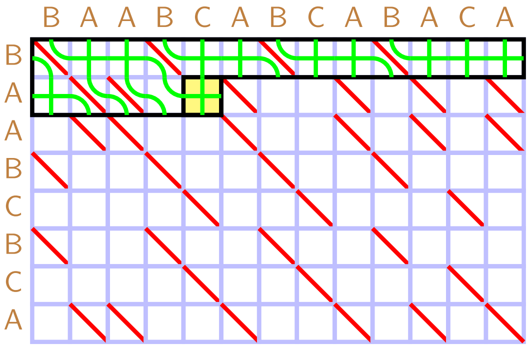
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

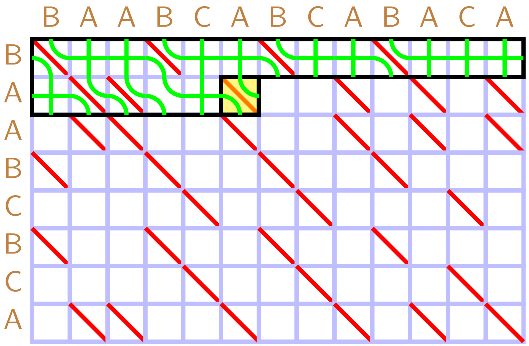
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

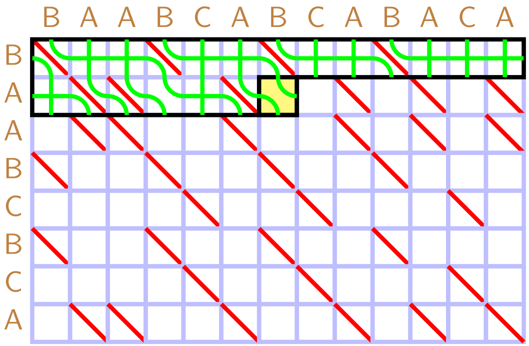
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

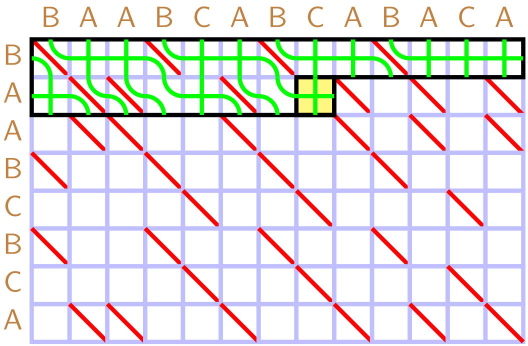
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

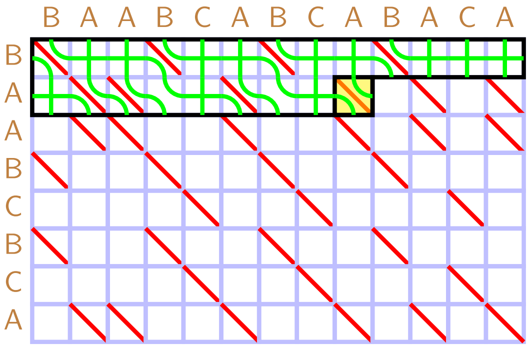
Semi-local LCS by iterative combing



Longest common subsequence

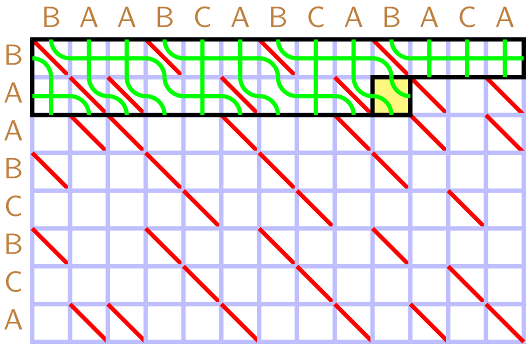
Algorithms for semi-local LCS

Semi-local LCS by iterative combing



Longest common subsequence
Algorithms for semi-local LCS

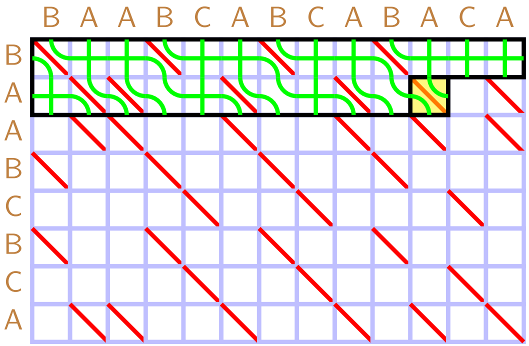
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

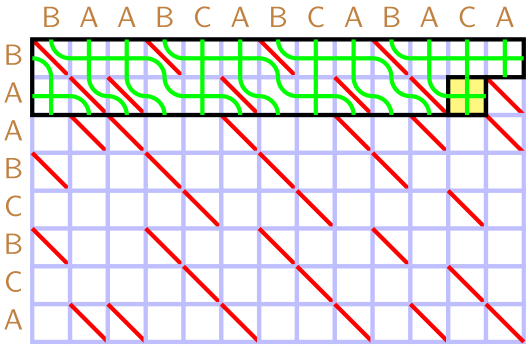
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

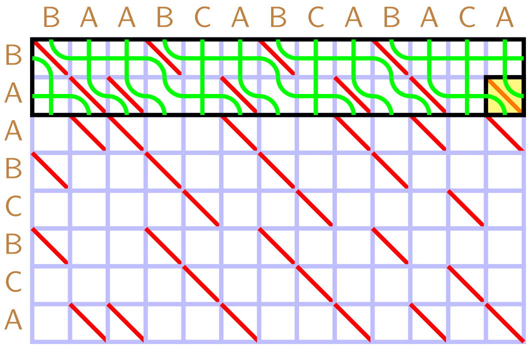
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

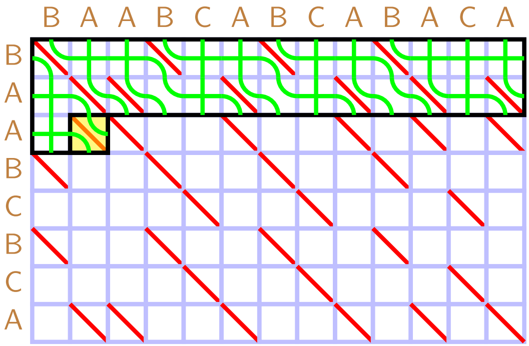
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

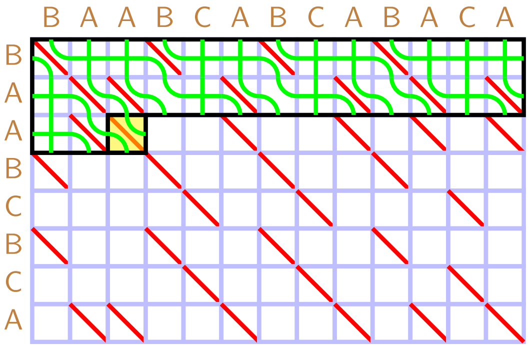
Semi-local LCS by iterative combing



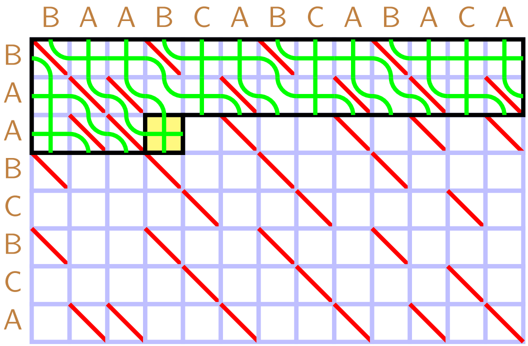
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing

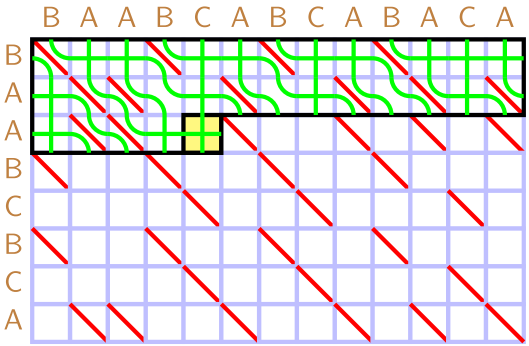


Semi-local LCS by iterative combing



Longest common subsequence
Algorithms for semi-local LCS

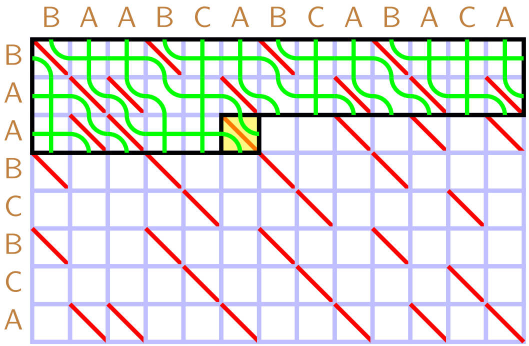
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

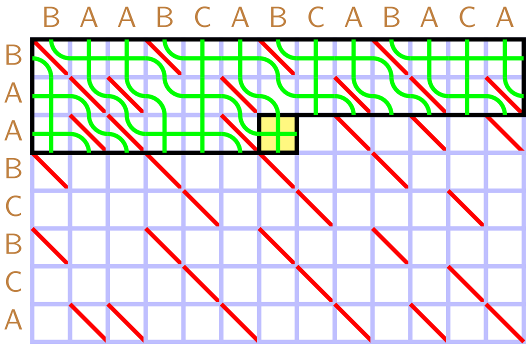
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

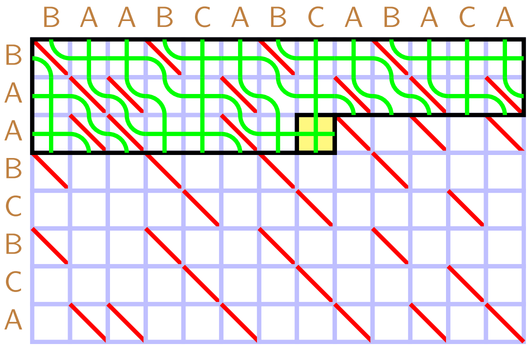
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

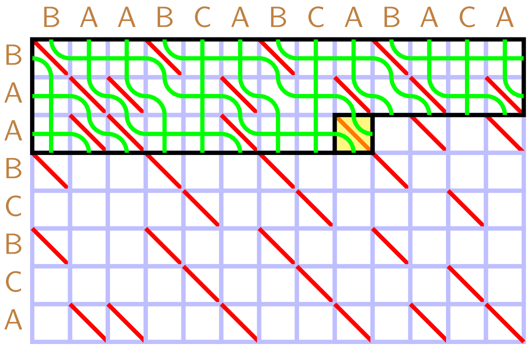
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

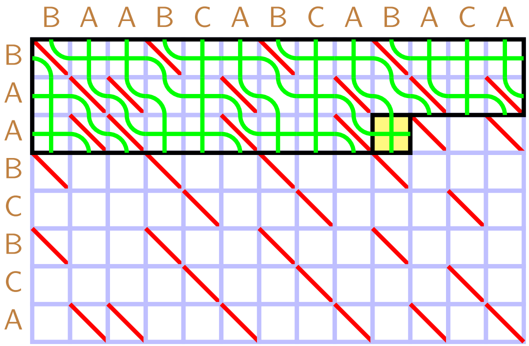
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

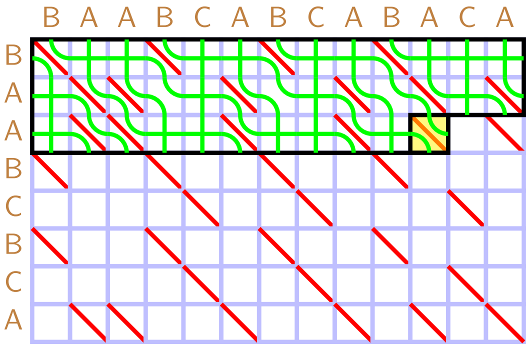
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

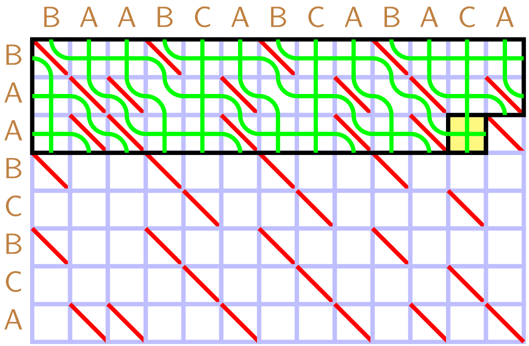
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

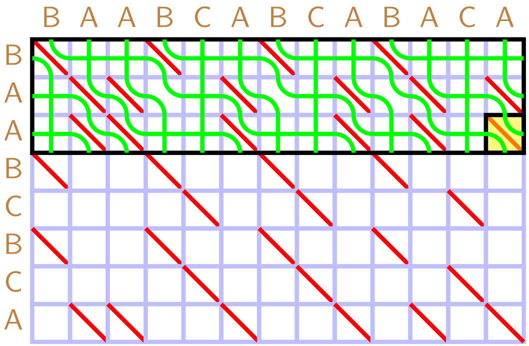
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

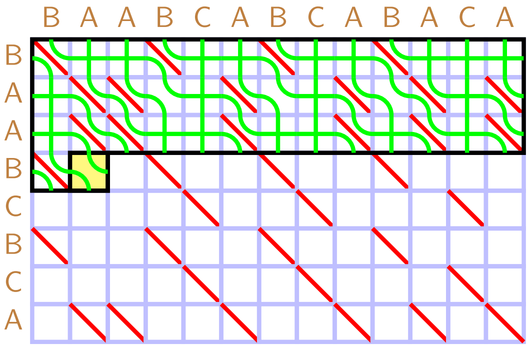
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

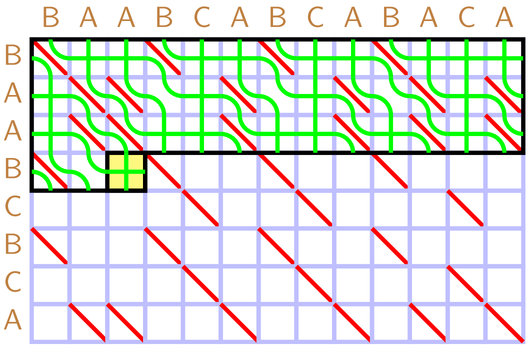
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

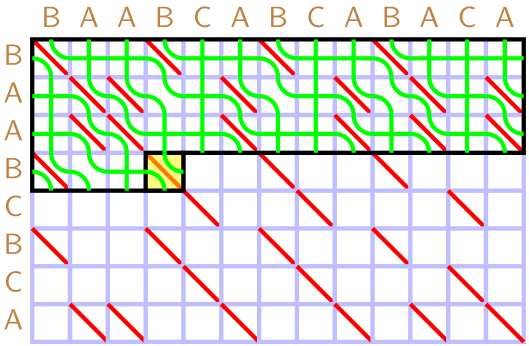
Semi-local LCS by iterative combing



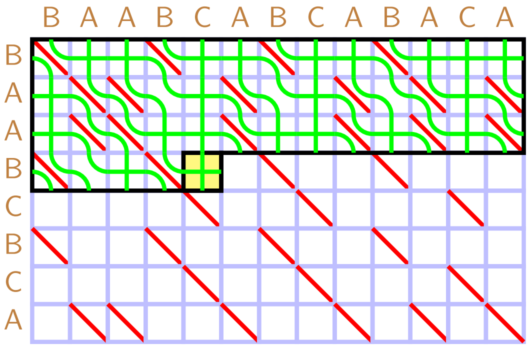
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing



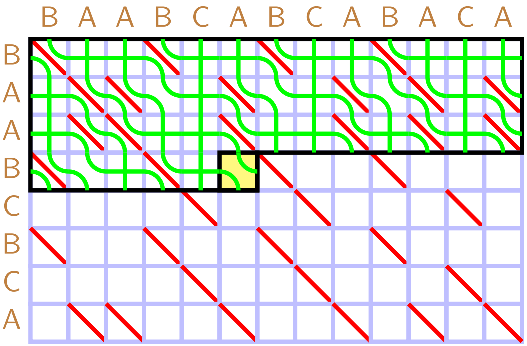
Semi-local LCS by iterative combing



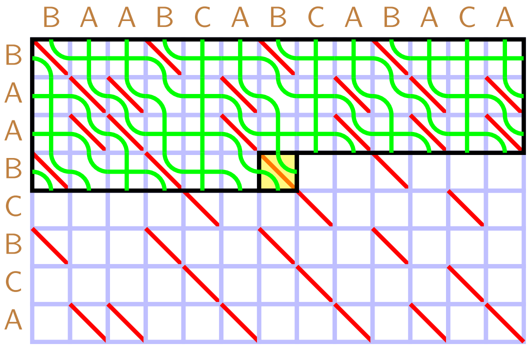
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing



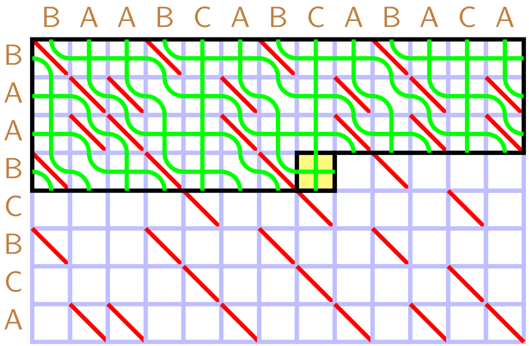
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

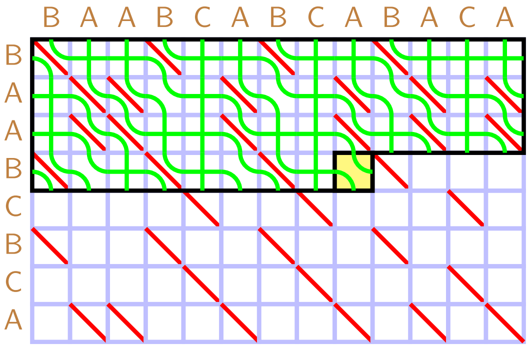
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

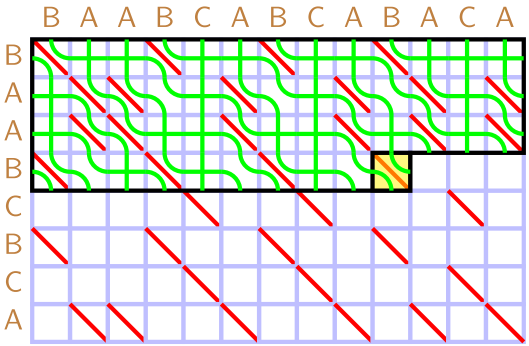
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

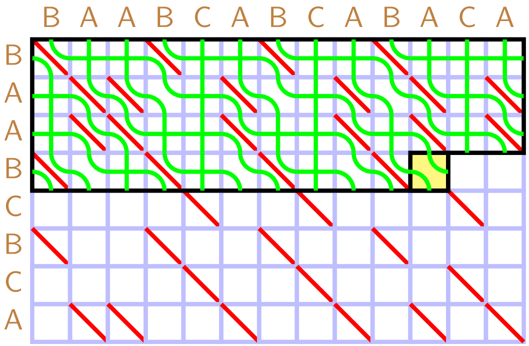
Semi-local LCS by iterative combing



Longest common subsequence

Algorithms for semi-local LCS

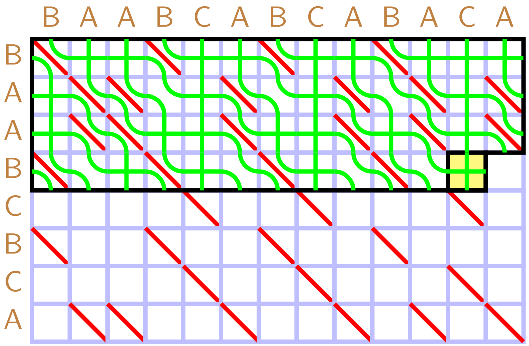
Semi-local LCS by iterative combing



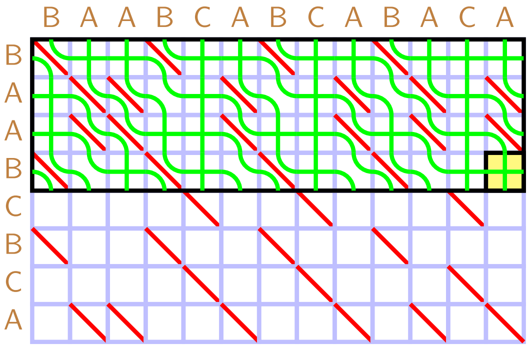
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing



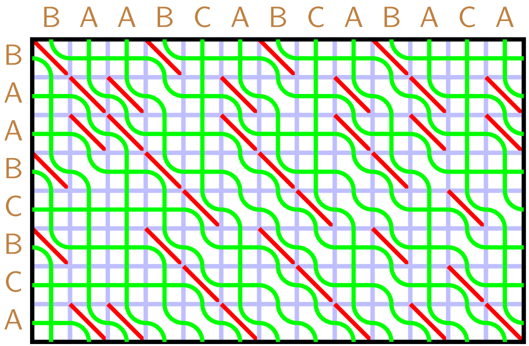
Semi-local LCS by iterative combing



Longest common subsequence

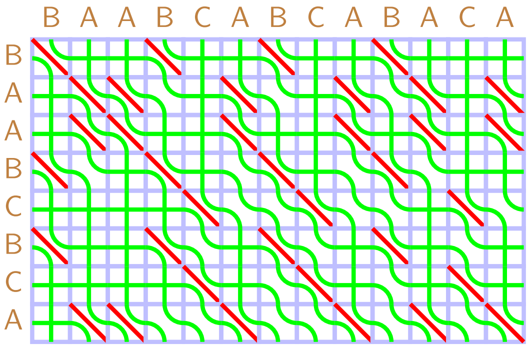
Algorithms for semi-local LCS

Semi-local LCS by iterative combing



Longest common subsequence
Algorithms for semi-local LCS

Semi-local LCS by iterative combing



Semi-local LCS: iterative combing

Initialise uncombed sticky braid: mismatch cell = crossing

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

- match cell: skip (keep strands uncrossed)
- mismatch cell: comb (uncross strands, if they crossed before)

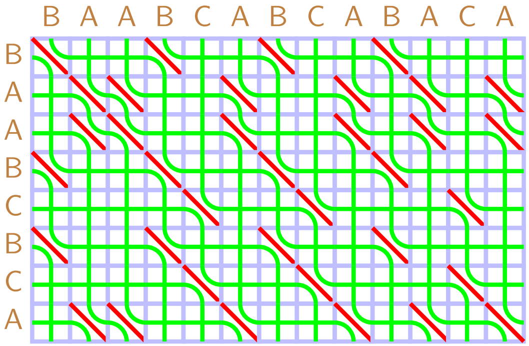
Active cell update: time $O(1)$

Overall time $O(mn)$

Correctness: by sticky braid relations

Longest common subsequence

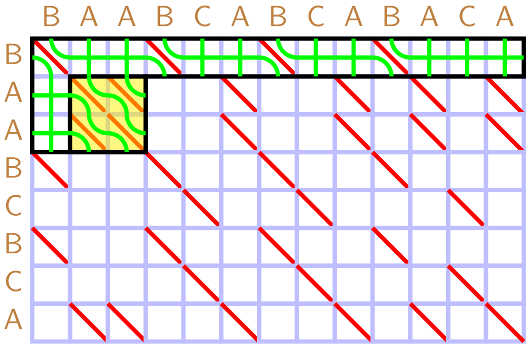
Semi-local LCS by iterative combing with micro-block speedup (MBS)



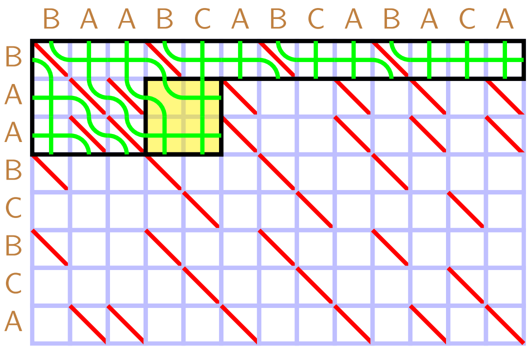
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing with micro-block speedup (MBS)



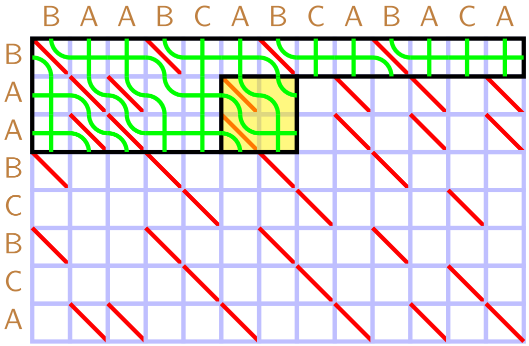
Semi-local LCS by iterative combing with micro-block speedup (MBS)



Longest common subsequence

Algorithms for semi-local LCS

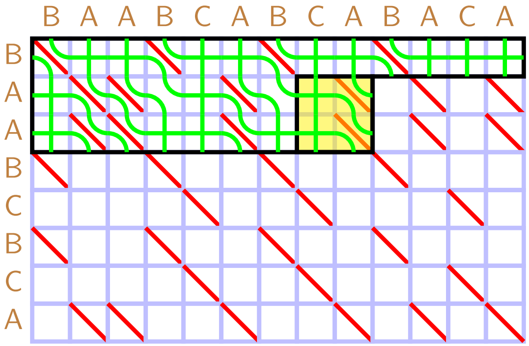
Semi-local LCS by iterative combing with micro-block speedup (MBS)



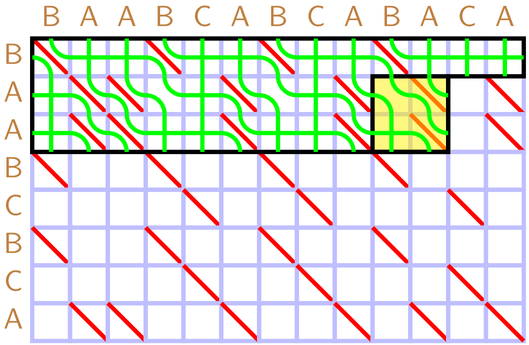
Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing with micro-block speedup (MBS)



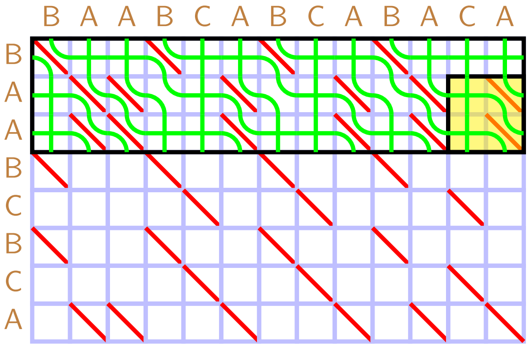
Semi-local LCS by iterative combing with micro-block speedup (MBS)



Longest common subsequence

Algorithms for semi-local LCS

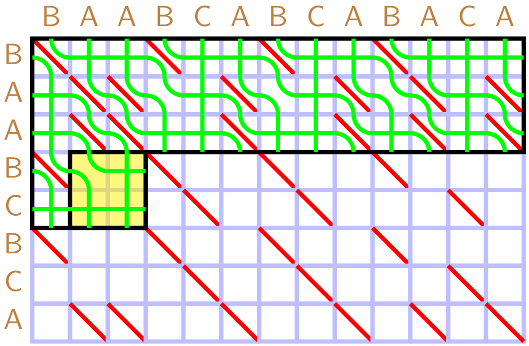
Semi-local LCS by iterative combing with micro-block speedup (MBS)



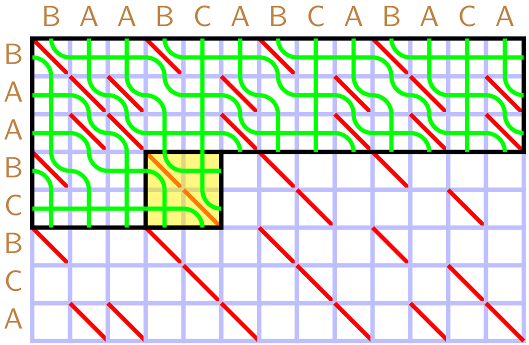
Longest common subsequence

Algorithms for semi-local LCS

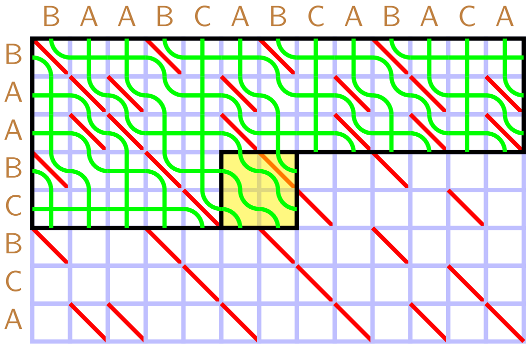
Semi-local LCS by iterative combing with micro-block speedup (MBS)



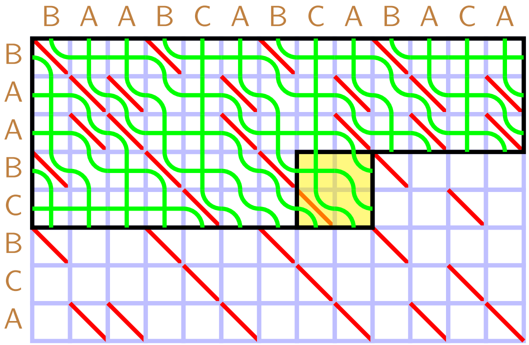
Semi-local LCS by iterative combing with micro-block speedup (MBS)



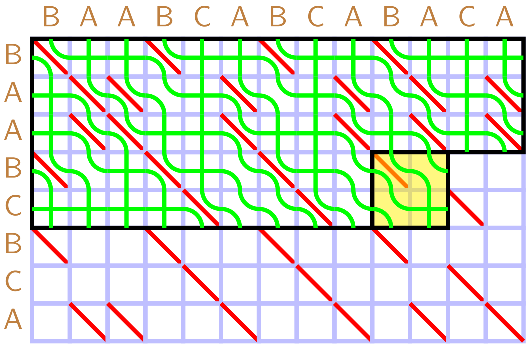
Semi-local LCS by iterative combing with micro-block speedup (MBS)



Semi-local LCS by iterative combing with micro-block speedup (MBS)

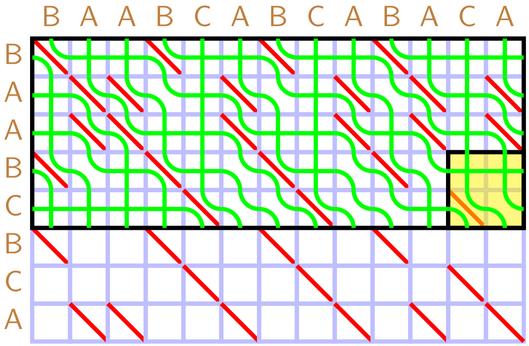


Semi-local LCS by iterative combing with micro-block speedup (MBS)

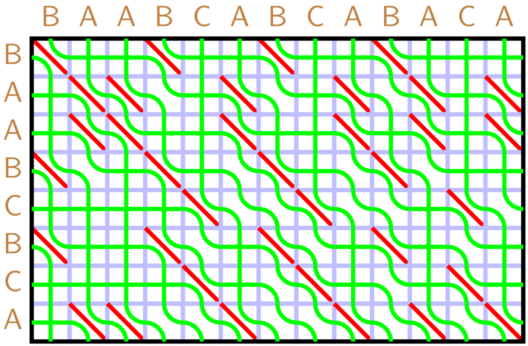


Longest common subsequence
Algorithms for semi-local LCS

Semi-local LCS by iterative combing with micro-block speedup (MBS)



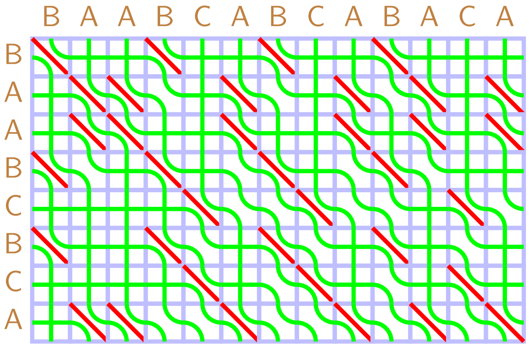
Semi-local LCS by iterative combing with micro-block speedup (MBS)



Longest common subsequence

Algorithms for semi-local LCS

Semi-local LCS by iterative combing with micro-block speedup (MBS)



Semi-local LCS: iterative combing with MBS [T: 2007]

Initialise uncombed sticky braid: mismatch cell $=$ crossing

Iterate over in micro-blocks, in any $\ll$-compatible order

Micro-block size: $t = O\left(\frac{\log n}{\log \log n}\right)$

Micro-block interface:

- $O(t)$ characters, each $O(\log \sigma)$ bits, can be reduced to $O(\log t)$ bits
- $O(t)$ integers, each $O(\log n)$ bits, can be reduced to $O(\log t)$ bits

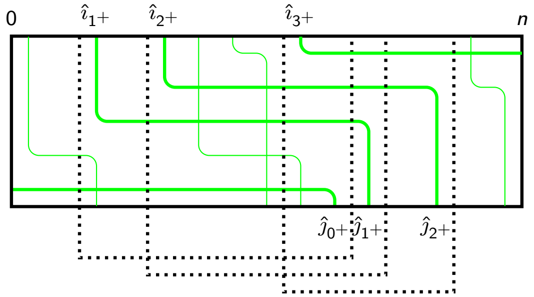
Micro-block output precomputed for every possible input

Active micro-block update: time $O(1)$ (precomputed)

Overall time $O\left(\frac{mn(\log \log n)^2}{\log^2 n}\right)$

Longest common subsequence

Subsequence matching



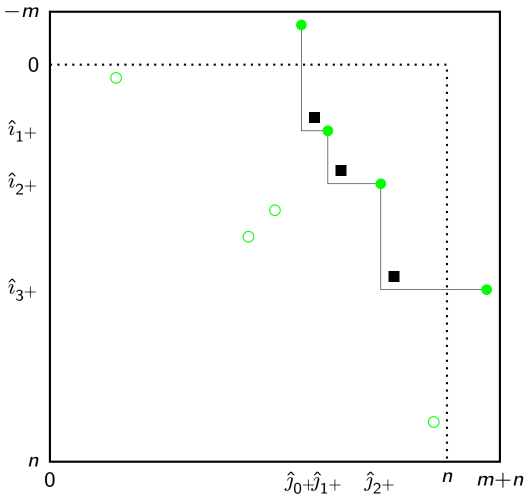
$b\langle i : j\rangle$ matching iff box $[i : j]$ not pierced left-to-right

Determined by $\ll$-chain of $\gtrless$-maximal nonzeros

$b\langle i : j\rangle$ minimally matching iff (i, j) is in the interleaved skeleton $\ll$-chain

Longest common subsequence

Subsequence matching



Alignment and edit distance

Alignment and edit distance

The LCS problem is a special case of the alignment problem

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

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

- LCS score: $(1, 0, 0)$
- Levenshtein score: $\left(1, \frac{1}{2}, 0\right)$

The LCS problem is a special case of the alignment problem

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

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

- LCS score: $(1, 0, 0)$
- Levenshtein score: $\left(1, \frac{1}{2}, 0\right)$

Scoring scheme is

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

The alignment problem

Return alignment score for *a* vs *b*

The alignment problem

Return alignment score for a vs b

Alignment: running time

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

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

The semi-local alignment problem

Analogous to the semi-local LCS problem, replacing LCS scores with alignment scores

Alignment and edit distance

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

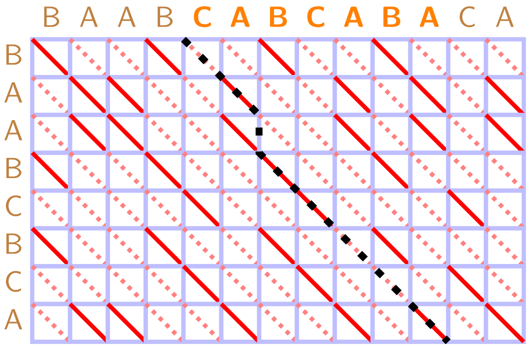
Substitution (sub) cost $-w_0 > 0$, insertion/deletion (indel) cost $-w_- > 0$

Corresponds to scoring scheme $(0, w_0, w_-)$

LCS (indel) distance: indel cost 1, sub cost 2
scoring scheme $(0, -2, -1) \rightarrow$ regular $(1, 0, 0)$

Levenshtein (indelsub) distance: indel cost 1, sub cost 1
scoring scheme $(0, -1, -1) \rightarrow$ regular $\left(1, \frac{1}{2}, 0\right)$

Semi-local alignment as maximum paths in the alignment grid



$blue = 0$

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

$red = 1$

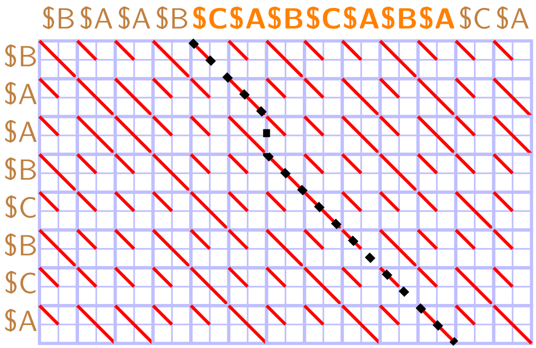
$a =$ "BAABCBCA"

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

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

Alignment and edit distance

Equivalent to semi-local LCS for blown-up strings



blue $= 0$

red $= 1$

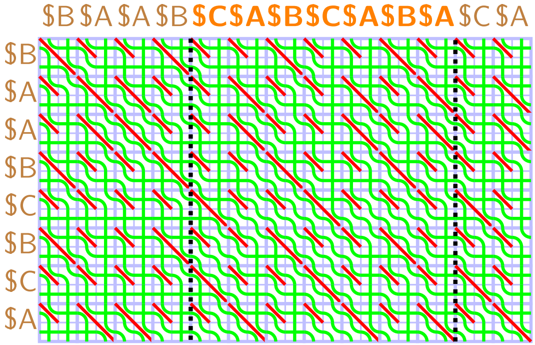
$a =$ "BAABCBCA"

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

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

Alignment and edit distance

Rational-weighted semi-local alignment via semi-local LCS



a, *b*, scheme $\left(1, \frac{\mu}{\nu}, 0\right) \to \mathfrak{a}, \mathfrak{b}$, scheme $(\nu, \mu, 0)$

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

Approximate matching problem

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

Assume rational scoring scheme

Approximate matching: running time

$O(mn)$		[Sellers: 1980]
$O\left(\frac{mn}{\log n}\right)$	$\sigma = O(1)$	via [Masek, Paterson: 1980]
$O\left(\frac{mn(\log\log n)^2}{\log^2 n}\right)$		via [Bille, Farach-Colton: 2008]

Approximate matching

Semi-local alignment by iterative combing on *a* vs *b* (with blow-up and MBS), time $O\left(\frac{mn(\log\log n)^2}{\log^2 n}\right)$

Implicit semi-local alignment matrix:

- anti-Monge
- approximate matching scores $\sim$ column minima

Column minima in $O(n)$ element queries [Aggarwal+: 1987]

Each query in time $O(\log^2 n)$ using the range tree representation, combined query time negligible

Overall running time $O\left(\frac{mn(\log\log n)^2}{\log^2 n}\right)$, same as [Bille, Farach-Colton: 2008]

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

indel R	$O(n)$		classical DP
insert LR	$O(n)$	a fixed	[Landau+: 1998]
			[Kim, Park: 2004]
insert LR	$O(n)$		[Ishida+: 2005]
indel LR	$O(n)$	also semi-local	[T: 2008]
indel any	$O(n(\log n)^2)$	also semi-local	[Charalampopoulos+: 2020]

Dynamic LCS

Maintain a hierarchy of substring LCS kernels

Further string comparison
Cyclic LCS

Cyclic LCS problem

Maximum LCS score for *a* vs all cyclic rotations of *b*

Cyclic LCS: running time

$O(mn^2)$	repeated DP
$O(mn \log m)$	[Maes: 1990]
$O(mn)$	[Bunke, Bühler: 1993; Landau+: 1998; Schmidt: 1998]
$O\left(\frac{mn(\log\log n)^2}{\log^2 n}\right)$	[T: 2007]

Cyclic LCS

Micro-block iterative combing on a vs bb, time $O\left(\frac{mn(\log\log n)^2}{\log^2 n}\right)$

Make n string-substring LCS queries, time negligible

Further string comparison
Periodic LCS

Periodic string-substring LCS problem

(Implicit) LCS scores for a vs every substring of $b = \dots uuu \dots = u^{\pm\infty}$

Let u be of length p

May assume that every character of a occurs in u (otherwise delete it)

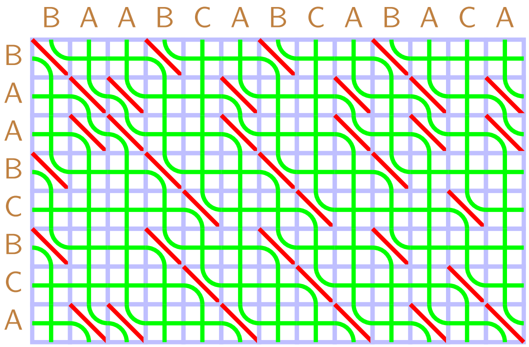
Only substrings of b of length $\leq mp$ (otherwise LCS score $= m$)

Periodic string-substring LCS: running time

$O(mnp)$	naive
$O(mp)$	[T: 2009]

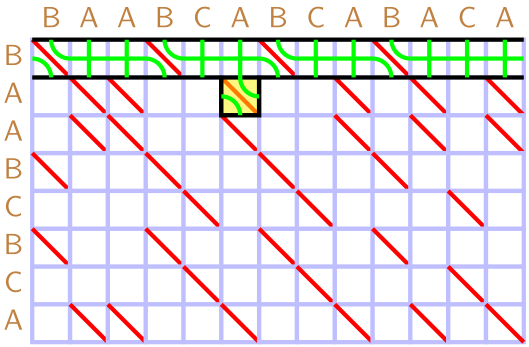
Further string comparison

Periodic LCS



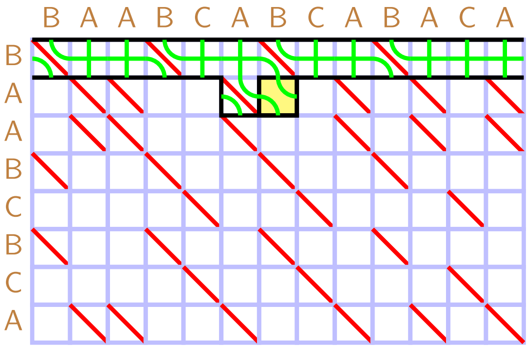
Further string comparison

Periodic LCS



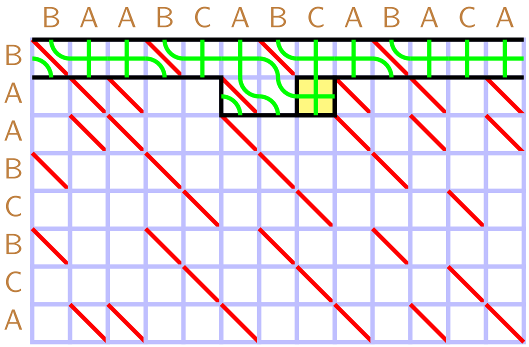
Further string comparison

Periodic LCS



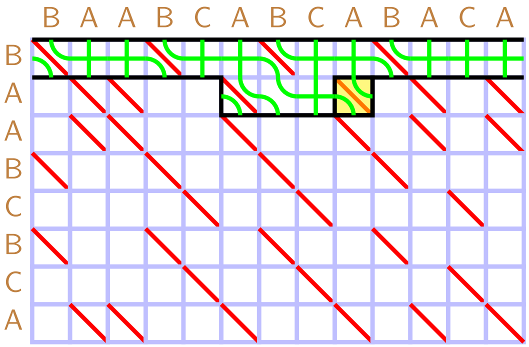
Further string comparison

Periodic LCS

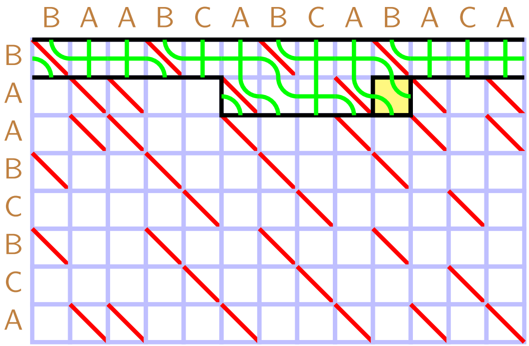


Further string comparison

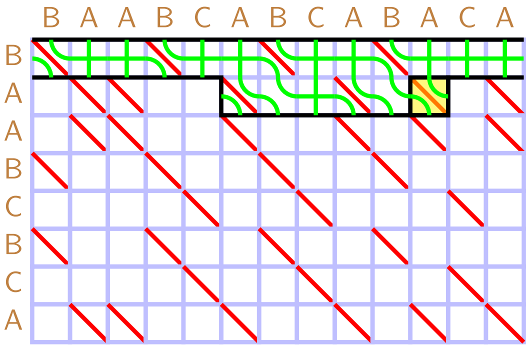
Periodic LCS



Further string comparison
Periodic LCS

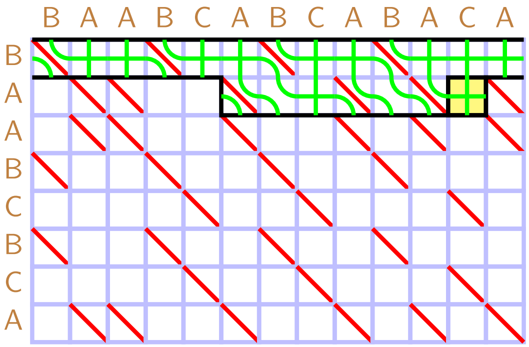


Further string comparison
Periodic LCS



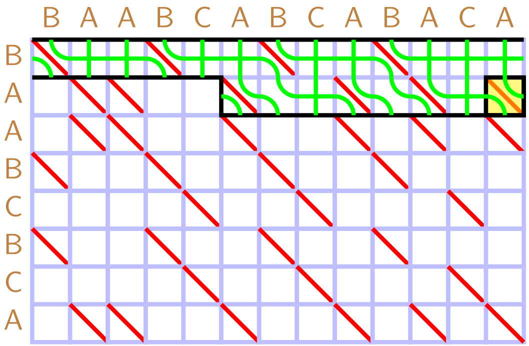
Further string comparison

Periodic LCS



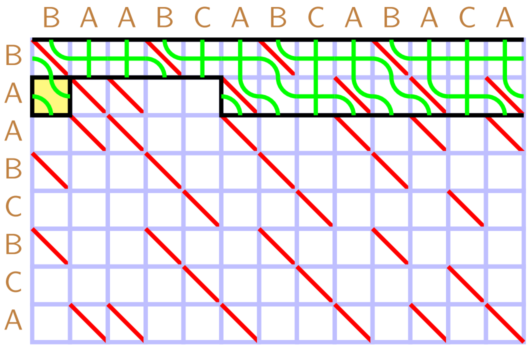
Further string comparison

Periodic LCS



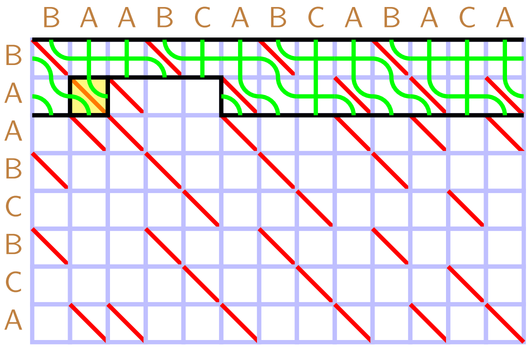
Further string comparison

Periodic LCS



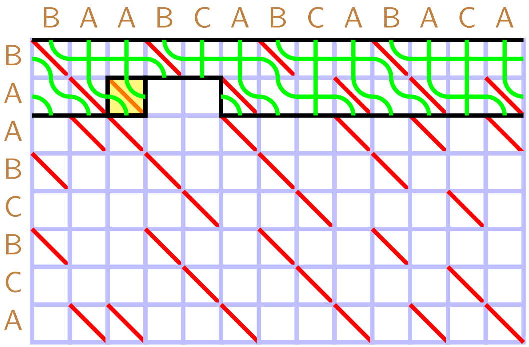
Further string comparison

Periodic LCS



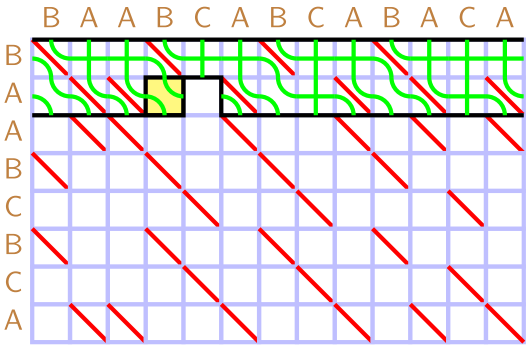
Further string comparison

Periodic LCS



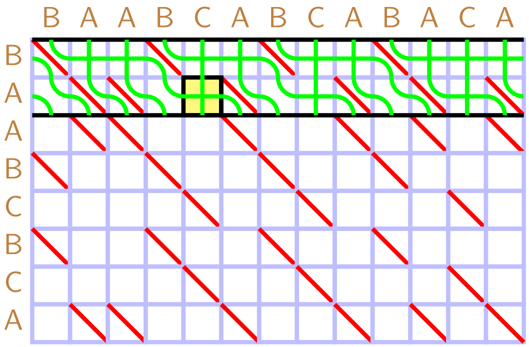
Further string comparison

Periodic LCS



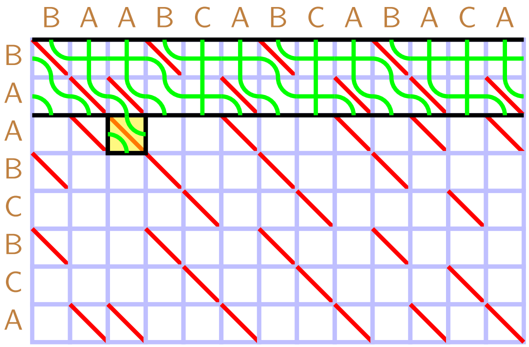
Further string comparison

Periodic LCS



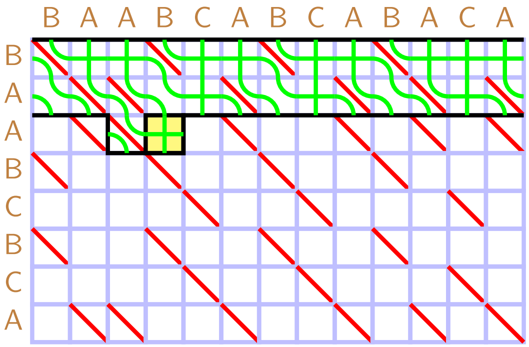
Further string comparison

Periodic LCS



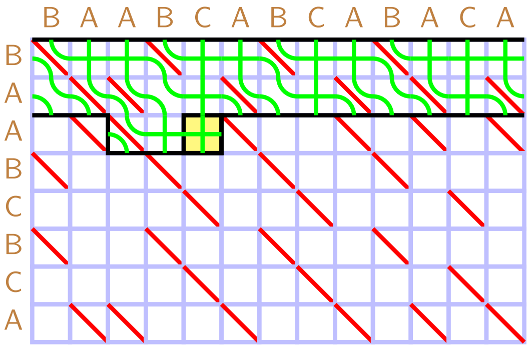
Further string comparison

Periodic LCS



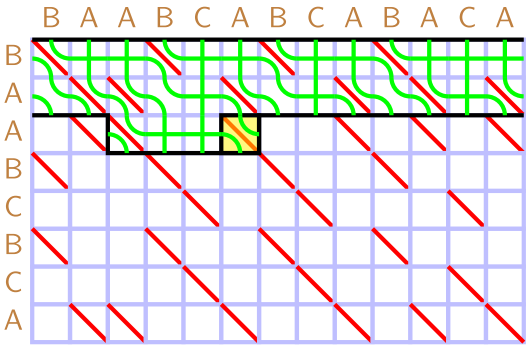
Further string comparison

Periodic LCS



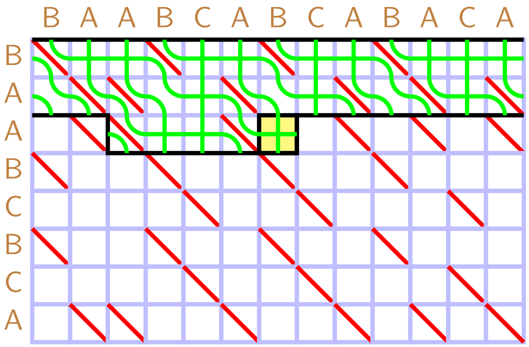
Further string comparison

Periodic LCS



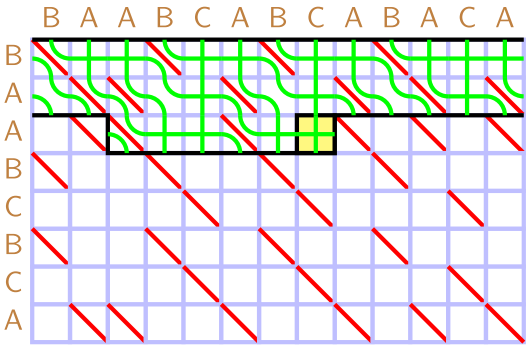
Further string comparison

Periodic LCS



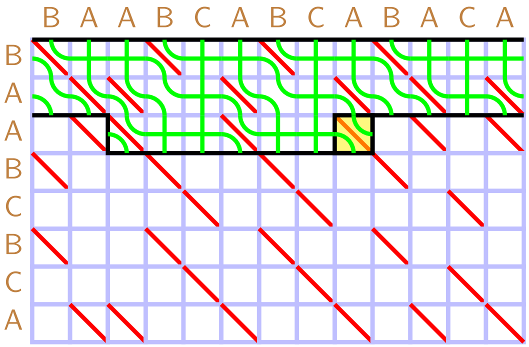
Further string comparison

Periodic LCS



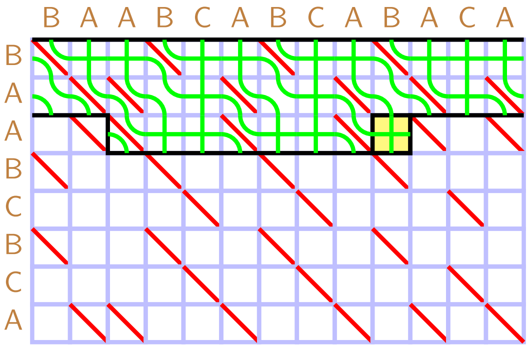
Further string comparison

Periodic LCS



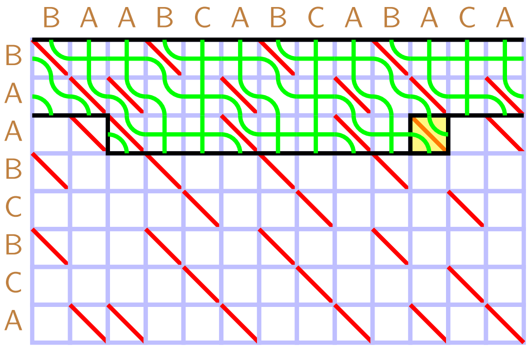
Further string comparison

Periodic LCS



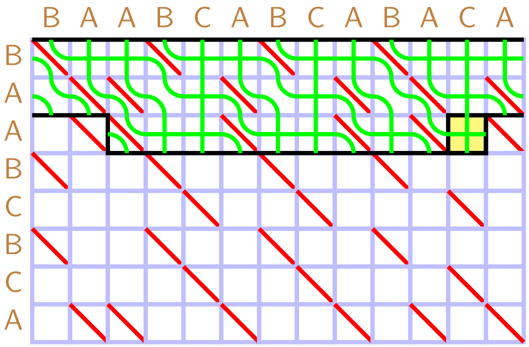
Further string comparison

Periodic LCS



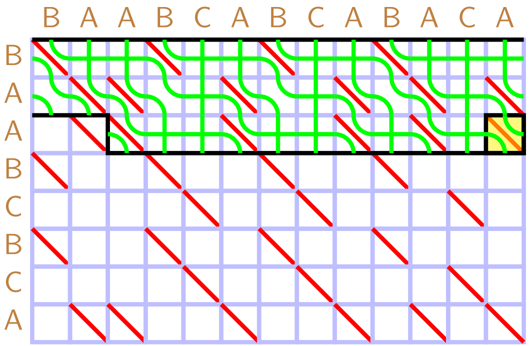
Further string comparison

Periodic LCS



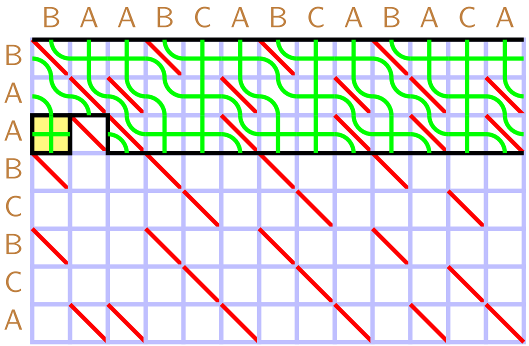
Further string comparison

Periodic LCS



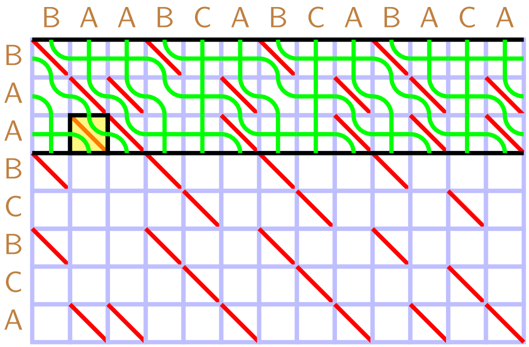
Further string comparison

Periodic LCS



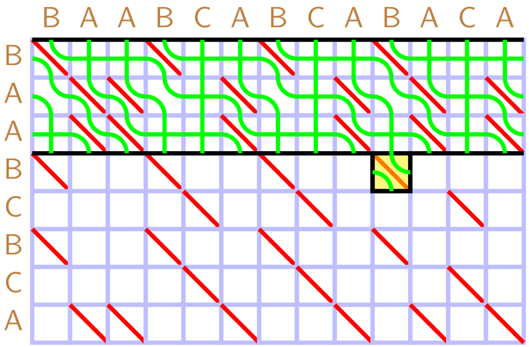
Further string comparison

Periodic LCS



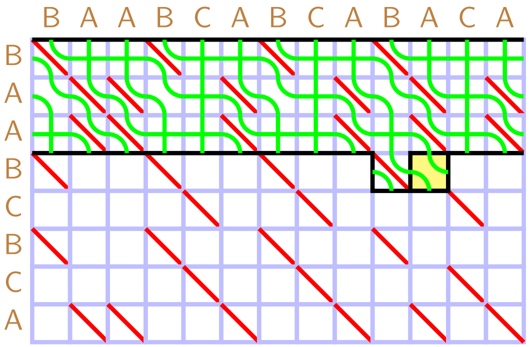
Further string comparison

Periodic LCS



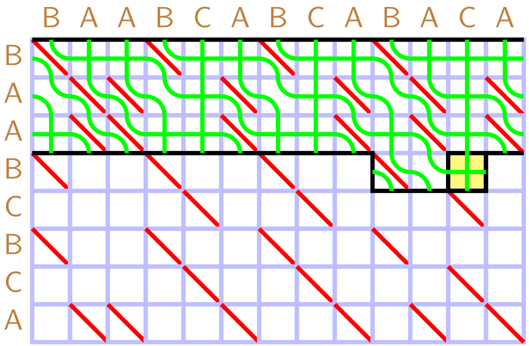
Further string comparison

Periodic LCS



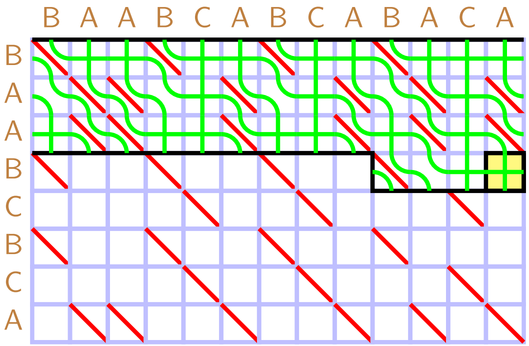
Further string comparison

Periodic LCS



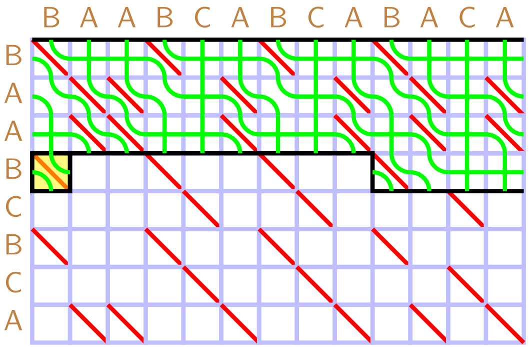
Further string comparison

Periodic LCS



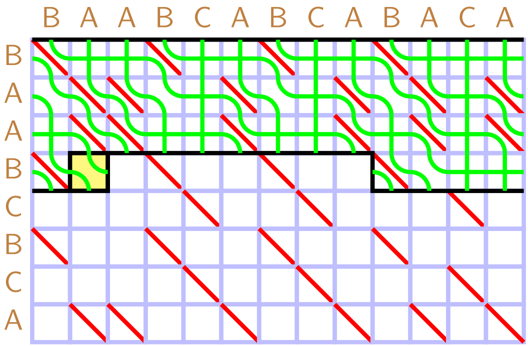
Further string comparison

Periodic LCS



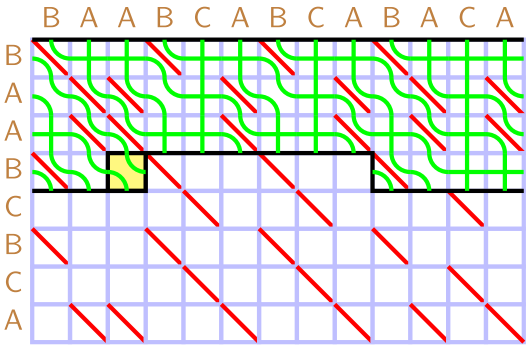
Further string comparison

Periodic LCS



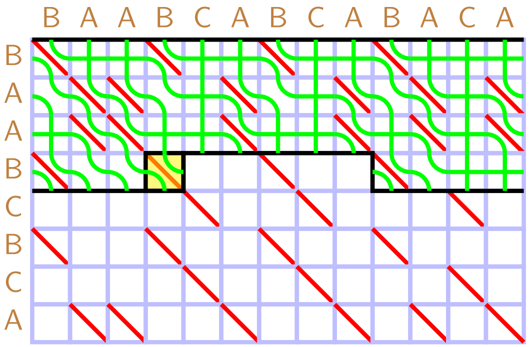
Further string comparison

Periodic LCS



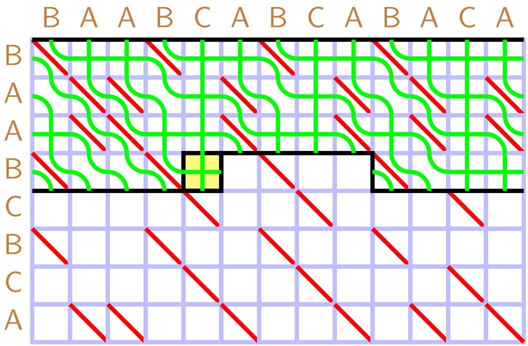
Further string comparison

Periodic LCS



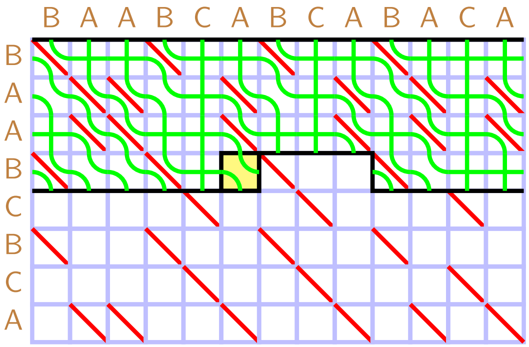
Further string comparison

Periodic LCS



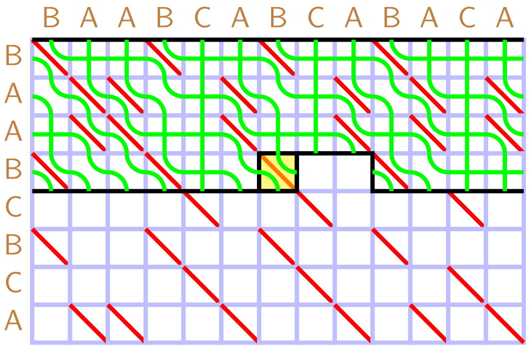
Further string comparison

Periodic LCS



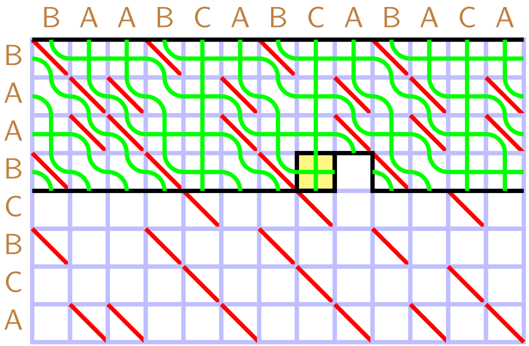
Further string comparison

Periodic LCS



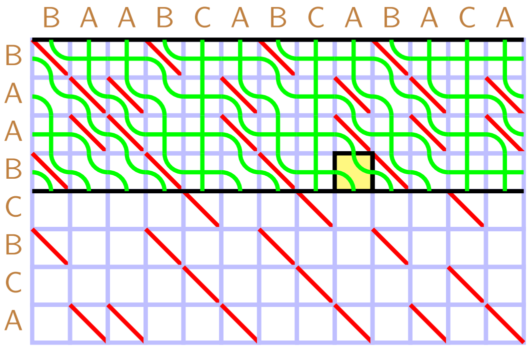
Further string comparison

Periodic LCS



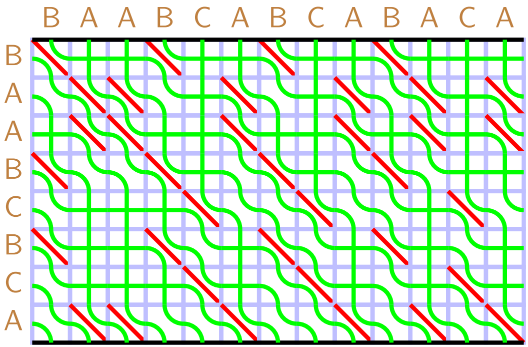
Further string comparison

Periodic LCS



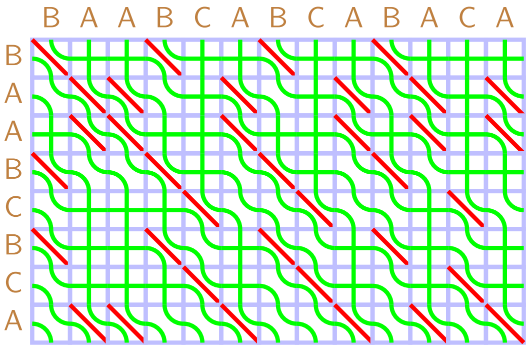
Further string comparison

Periodic LCS



Further string comparison

Periodic LCS



Further string comparison
Periodic LCS

Periodic string-substring LCS: Wraparound iterative combing

Initialise uncombed sticky braid: mismatch cell = crossing

Sweep cells row-by-row: each row starts at match cell, wraps at boundary

Sweep cells in any $\ll$-compatible order

- match cell: skip (keep uncrossed)
- mismatch cell: comb (uncross) iff the same strand pair already crossed before, possibly in a different period

Cell update: time $O(1)$

Overall time $O(mn)$

String-substring LCS score: count strands with multiplicities

Tandem LCS problem

LCS score for a vs $b = u^k$

We have $n = kp$; may assume $k \leq m$

Tandem LCS: running time

$O(mkp)$	naive
$O\big(m(k+p)\big)$	[Landau, Ziv-Ukelson: 2001]
$O(mp)$	[T: 2009]

Direct application of wraparound iterative combing

Further string comparison

Periodic LCS

Tandem alignment problem

Substring closest to a in $b = u^{\pm\infty}$ by alignment score among

- global: substrings u^k of length kp across all k
- cyclic: substrings of length kp across all k
- local: substrings of any length

Tandem alignment: running time

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

Further string comparison

Periodic LCS

Cyclic tandem alignment

Periodic iterative combing for a vs b (with blow-up), time $O(mp)$

For each $k \in [1 : m]$:

- solve tandem LCS (under given alignment score) for a vs u^k
- obtain scores for a vs p successive substrings of b of length kp by LCS batch query: time $O(1)$ per substring

Running time $O(mp)$

Further string comparison

Longest square subsequence

Longest square subsequence problem

Longest subsequence of a having form bb, for some b

Further string comparison

Longest square subsequence

Longest square subsequence problem

Longest subsequence of a having form bb, for some b

Longest square subsequence: running time

$O(m^3)$	repeated DP
$O(m^2)$	[Kosowski: 2004]
$O\left(\frac{m^2(\log\log m)^2}{\log^2 m}\right)$	[T: 2007]

Longest square subsequence

Micro-block iterative combing on a vs a, time $O\left(\frac{m^2(\log\log m)^2}{\log^2 m}\right)$

Make $m-1$ prefix-suffix LCS queries, time negligible

Open question: generalise to longest cube subsequence, etc.

LCS problem on permutation strings (LCSP)

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

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

LCSP generalisations

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

Sparse string comparison
Semi-local LCS 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]

Semi-local LCSP problem

Semi-local LCS scores for a vs b; in each of a, b all characters distinct

Equivalent/similar to

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

Sparse string comparison

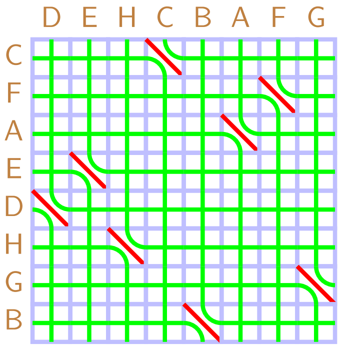
Semi-local LCS on permutation strings

Semi-local LCSP: 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
Semi-local LCS on permutation strings

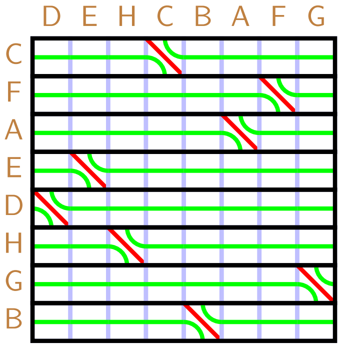
Semi-local LCSP by sparse recursive combing



Sparse string comparison

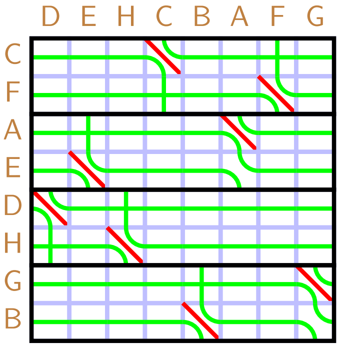
Semi-local LCS on permutation strings

Semi-local LCSP by sparse recursive combing



Sparse string comparison
Semi-local LCS on permutation strings

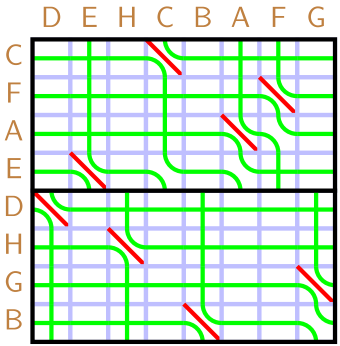
Semi-local LCSP by sparse recursive combing



Sparse string comparison

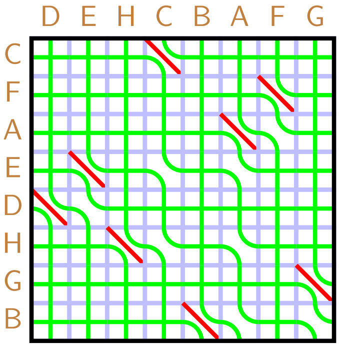
Semi-local LCS on permutation strings

Semi-local LCSP by sparse recursive combing



Sparse string comparison
Semi-local LCS on permutation strings

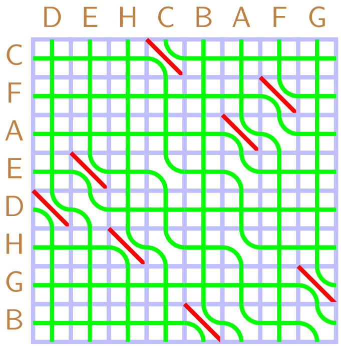
Semi-local LCSP by sparse recursive combing



Sparse string comparison

Semi-local LCS on permutation strings

Semi-local LCSP by sparse recursive combing



Semi-local LCSP: 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 $O(n \log^2 n)$

Sparse string comparison

Longest piecewise monotone subsequences

A k-increasing sequence: concatenation of k increasing sequences

A $(k-1)$-modal sequence: concatenation of k alternating increasing and decreasing sequences

Longest k-increasing ($(k-1)$-modal) subsequence problem

Length of longest k-increasing ($(k-1)$-modal) subsequence of b

Longest k-increasing ($(k-1)$-modal) subsequence: running time

$O(nk \log n)$	$(k-1)$-modal	[Demange+: 2007]
$O(nk \log n)$		via [Hunt, Szymanski: 1977]
$O(n \log^2 n)$		[T: 2010]

Sparse string comparison

Longest piecewise monotone subsequences

Longest k-increasing subsequence

Case $k \leq \log n$: sparse LCS for id^k vs b, time $O(nk \log n)$

Case $k \geq \log n$:

Semi-local LCSP for id vs b: time $O(n \log^2 n)$

Extract string-substring LCSP for id vs b

String-substring LCS for id^k vs b by at most $2 \log k$ instances of $\boxdot$-square/multiply: time $\log k \cdot O(n \log n) = O(n \log^2 n)$

Query the LCS score for id^k vs b: time negligible

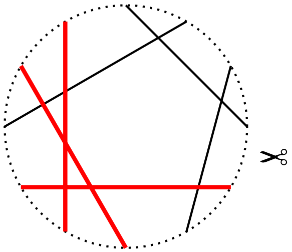
Overall time $O(n \log^2 n)$

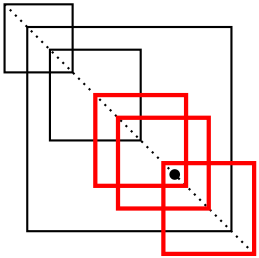
Longest $(k-1)$-modal subsequence: analogous

Max-clique in a circle graph

Given a circle with n chords, find max-size subset of intersecting chords

Applications: chemistry, biology (most nested pseudoknot in RNA)

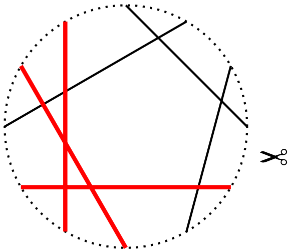


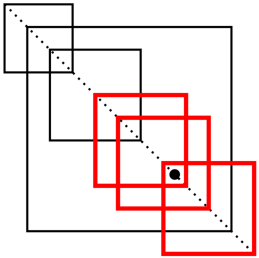


Max-clique in a circle graph

Given a circle with n chords, find max-size subset of intersecting chords

Applications: chemistry, biology (most nested pseudoknot in RNA)





Interval model: cut and straighten the circle; chords become intervals

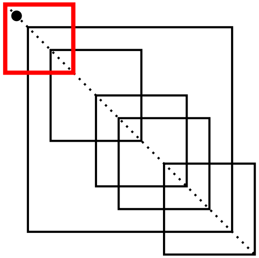
Chords intersect iff intervals overlap, i.e. intersect without containment

Max-clique in a circle graph: running time

$exp(n)$	naive
$O(n^3)$	[Gavril: 1973]
$O(n^2)$	[Rotem, Urrutia: 1981; Hsu: 1985]
	[Masuda+: 1990; Apostolico+: 1992]
$O(n^{1.5})$	[T: 2006]
$O(n \log^2 n)$	[T: 2010]

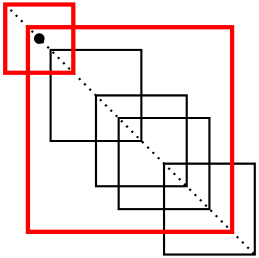
Sparse string comparison

Maximum clique in a circle graph



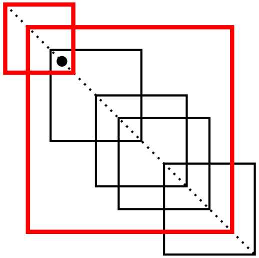
Sparse string comparison

Maximum clique in a circle graph



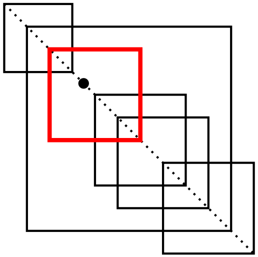
Sparse string comparison

Maximum clique in a circle graph



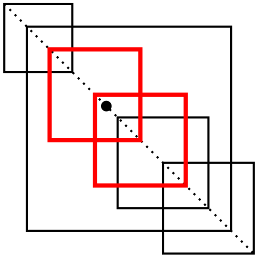
Sparse string comparison

Maximum clique in a circle graph



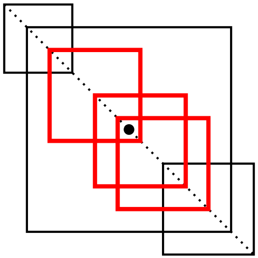
Sparse string comparison

Maximum clique in a circle graph



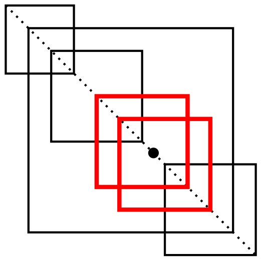
Sparse string comparison

Maximum clique in a circle graph



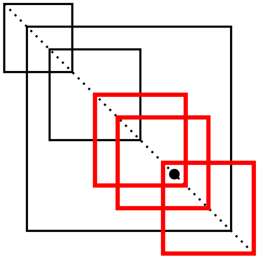
Sparse string comparison

Maximum clique in a circle graph

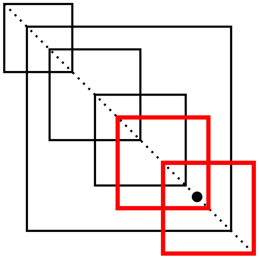


Sparse string comparison

Maximum clique in a circle graph

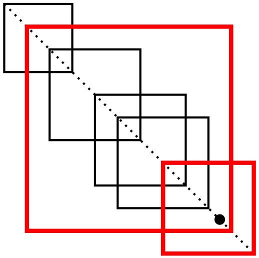


Sparse string comparison
Maximum clique in a circle graph



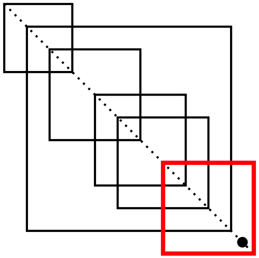
Sparse string comparison

Maximum clique in a circle graph



Sparse string comparison

Maximum clique in a circle graph



Sparse string comparison

Maximum clique in a circle graph

Max-clique in a circle graph

Helly property: if any set of intervals intersect pairwise, then they all intersect at a common point

Compute LCS kernel, build range tree: time $O(n \log^2 n)$

Run through all $2n + 1$ possible common intersection points

For each point, find a maximum subset of covering overlapping segments by a prefix-suffix LCS query: time $(2n + 1) \cdot O(\log^2 n) = O(n \log^2 n)$

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

Sparse string comparison
Maximum clique in a circle graph

Max-clique in a circle graph

The maximum clique problem in a circle graph, sensitive e.g. to

- number e of edges; $e \leq n^2$
- size l of maximum clique; $l \leq n$
- cutwidth d of interval model (max number of intervals covering a point); $l \leq d \leq n$

Parameterised maximum clique in a circle graph: running time

$O(n \log n + e)$	[Apostolico+: 1992]
$O(n \log n + nl \log(n/l))$	[Apostolico+: 1992]
$O(n \log n + n \log^2 d)$	NEW

Sparse string comparison
Maximum clique in a circle graph

Max-clique in a circle graph, parameterised by cutwidth

For each diagonal block of size d, compute LCS kernel, build LCS oracle: time $n/d \cdot O(d \log^2 d) = O(n \log^2 d)$

Extend each diagonal block to a quadrant: time $O(n \log^2 d)$

Run through all $2n + 1$ possible common intersection points

For each point, find a maximum subset of covering overlapping segments by a prefix-suffix LCS query: time $O(n \log^2 d)$

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

Compressed string comparison

Grammar compression

Notation: pattern p of length m; text t of length n

A GC-string (grammar-compressed string) t is 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, $n = O(2^{\bar{n}})$

Example: Fibonacci string "ABAABABAABAAB"

$t_1 = \mathrm{A} \quad t_2 = t_1\mathrm{B} \quad t_3 = t_2t_1 \quad t_4 = t_3t_2 \quad t_5 = t_4t_3 \quad t_6 = t_5t_4$

Compressed string comparison

Grammar compression

Grammar-compression covers various compression types, e.g. LZ78, LZW (not LZ77 directly)

Simplifying assumption: arithmetic up to n runs in $O(1)$

This assumption can be removed by careful index remapping

Compressed string comparison

Extended substring-string LCS on GC-strings

LCS: running time ($r = m + n$, $\bar{r} = \bar{m} + \bar{n}$)

p	t			
plain	plain	$O(mn)$		[Wagner, Fischer: 1974]
		$O\left(\frac{mn}{\log^2 m}\right)$		[Masek, Paterson: 1980]
				[Crochemore+: 2003]
plain	GC	$O(m^3\bar{n} + \dots)$	gen. CFG	[Myers: 1995]
		$O(m^{1.5}\bar{n})$	ext subs-s	[T: 2008]
		$O(m \log m \cdot \bar{n})$	ext subs-s	[T: 2010]
GC	GC	NP-hard		[Lifshits: 2005]
		$O(r^{1.2}\bar{r}^{1.4})$	R weights	[Hermelin+: 2009]
		$O(r \log r \cdot \bar{r})$		[T: 2010]
		$O\big(r \log(r/\bar{r}) \cdot \bar{r}\big)$		[Hermelin+: 2010]
		$O\big(r \log^{1/2}(r/\bar{r}) \cdot \bar{r}\big)$		[Gawrychowski: 2012]

Compressed string comparison

Extended substring-string LCS on GC-strings

Substring-string LCS (plain pattern, GC text): the algorithm

For every k, compute by recursion the appropriate part of semi-local LCS for p vs t_k, using matrix sticky multiplication: time $O(m \log m \cdot \bar{n})$

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

Compressed string comparison

Subsequence matching in GC-strings

The subsequence recognition problem

Does pattern p appear in text t as a subsequence?

Subsequence recognition: running time

p	t		
plain	plain	$O(n)$	greedy
plain	GC	$O(m\bar{n})$	greedy
GC	GC	NP-hard	[Lifshits: 2005]

The subsequence matching problem

Find all minimally matching substrings of t with respect to p

Substring of t is matching, if p is a subsequence of t

Matching substring of t is minimally matching, if none of its proper substrings are matching

Compressed string comparison

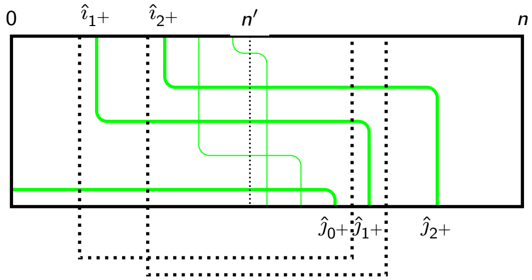
Subsequence matching in GC-strings

Subsequence matching: running time (+ *output*)

p	t		
plain	plain	$O(mn)$	[Mannila+: 1995]
		$O\left(\frac{mn}{\log m}\right)$	[Das+: 1997]
		$O(c^m + n)$	[Boasson+: 2001]
		$O(m + n\sigma)$	[Troníček: 2001]
plain	GC	$O(m^2 \log m\bar{n})$	[Cégielski+: 2006]
		$O(m^{1.5}\bar{n})$	[T: 2008]
		$O(m \log m \cdot \bar{n})$	[T: 2010]
		$O(m \cdot \bar{n})$	[Yamamoto+: 2011]
GC	GC	NP-hard	[Lifshits: 2005]

Compressed string comparison

Subsequence matching in GC-strings



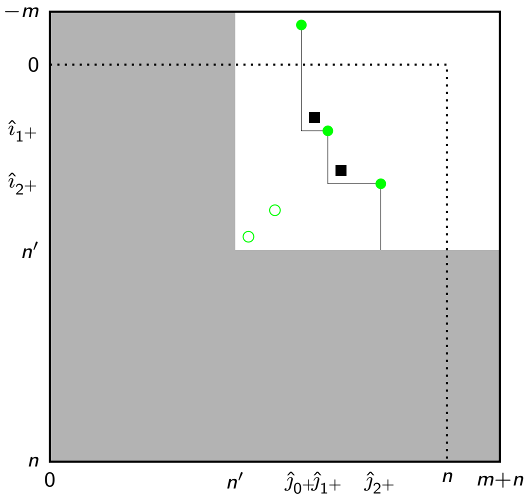
$b\langle i : j\rangle$ matching iff box $[i : j]$ not pierced left-to-right

Determined by $\ll$-chain of $\gtrless$-maximal nonzeros

$b\langle i : j\rangle$ minimally matching iff (i, j) is in the interleaved skeleton $\ll$-chain

Compressed string comparison

Subsequence matching in GC-strings



Local SM (plain pattern, GC text): the algorithm

For every k, compute by recursion the appropriate part of semi-local LCS for p vs t_k, using matrix sticky multiplication: time $O(m \log m \cdot \bar{n})$

Local SM (plain pattern, GC text): the algorithm

For every k, compute by recursion the appropriate part of semi-local LCS for p vs t_k, using matrix sticky multiplication: time $O(m \log m \cdot \bar{n})$

Given an assignment $t = t' t''$, find by recursion

- minimally matching substrings in t'
- minimally matching substrings in t''

Compressed string comparison

Local SM (plain pattern, GC text): the algorithm

For every k, compute by recursion the appropriate part of semi-local LCS for p vs t_k, using matrix sticky multiplication: time $O(m \log m \cdot \bar{n})$

Given an assignment $t = t't''$, find by recursion

- minimally matching substrings in t'
- minimally matching substrings in t''

Then, find $\ll$-chain of $\gtrless$-maximal nonzeros in time $\bar{n} \cdot O(m) = O(m\bar{n})$

Its skeleton $\ll$-chain: minimally matching substrings in t overlapping t', t''

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

Compressed string comparison

Approximate matching in GC-strings

The edit distance matching problem

Find all substrings of t, for which edit distance to p is at most k

Compressed string comparison

Approximate matching in GC-strings

Edit distance matching: running time (+ *output*)

p	t		
plain	plain	$O(mn)$	[Sellers: 1980]
		$O(mk)$	[Landau, Vishkin: 1989]
		$O(m + n + \frac{nk^4}{m})$	[Cole, Hariharan: 2002]
plain	GC	$O(m\bar{n}k^2)$	[Kärkkäinen+: 2003]
		$O(m\bar{n}k + \bar{n}\log n)$	[LV: 1989] via [Bille+: 2010]
		$O(m\bar{n} + \bar{n}k^4 + \bar{n}\log n)$	[CH: 2002] via [Bille+: 2010]
		$O(m \log m \cdot \bar{n})$	[T: 2014]
GC	GC	NP-hard	[Lifshits: 2005]

(Also many specialised variants for LZ compression)

Compressed string comparison

Approximate matching in GC-strings

Edit distance matching (plain pattern, GC text): the algorithm

Algorithm structure similar to local subsequence recognition

$\ll$-chains replaced by $m \times m$ submatrices

Extra ingredients:

- the blow-up technique: reduction of edit distances to LCS scores
- implicit matrix searching, replaces $\ll$-chain interleaving

Monge row minima: "SMAWK" $O(m)$ [Aggarwal+: 1987]

Implicit unit-Monge row minima:

$O(m \log \log m)$ [T: 2012]
$O(m)$ [Gawrychowski: 2012]

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

Beyond semi-locality
Window-substring LCS

A *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

Beyond semi-locality
Window-substring LCS

A *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]

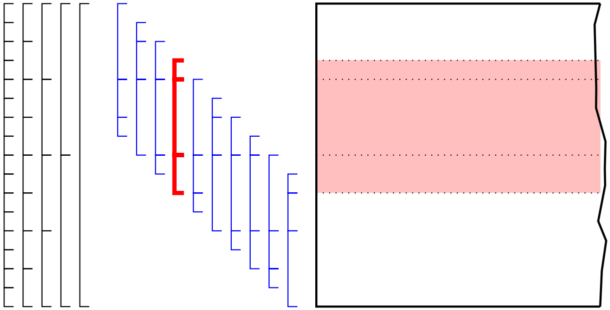
Beyond semi-locality

Window-substring LCS

Window-substring LCS

Compute LCS kernels for canonical substrings of a vs b

Compute LCS kernels for every w-window of a vs b as sticky product of canonical substring kernels: overall time $O(mn)$



Beyond semi-locality
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

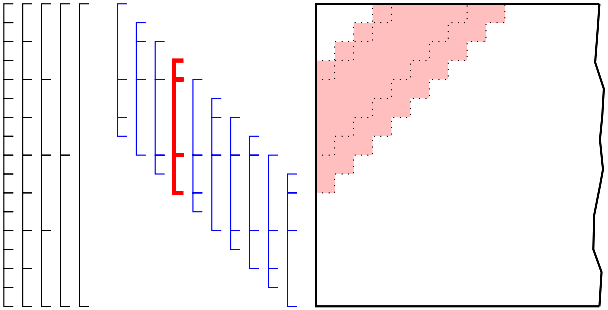
Beyond semi-locality

Window-substring LCS

Fixed total length LCS

Compute antidiagonal analogue of LCS kernels for canonical intervals

Compute antidiagonal analogue of LCS kernels for 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 weak genome conservation

Running time dominated by window-substring LCS

Beyond semi-locality

Fragment-substring LCS

Given a set of m (possibly overlapping) fragment substrings in a

Fragment-substring LCS problem

LCS score for every fragment of a vs every substring of b

Beyond semi-locality
Fragment-substring LCS

Given a set of m (possibly overlapping) fragment substrings in a

Fragment-substring LCS problem

LCS score for every fragment of a vs every substring of b

Fragment-substring LCS: running time

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

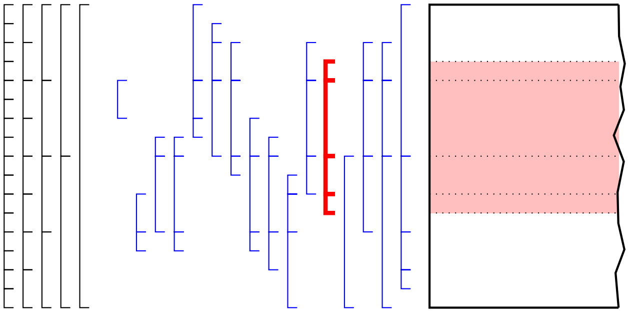
$O(mn \log^2 m)$ [T: 2008]

Beyond semi-locality
Fragment-substring LCS

Fragment-substring LCS

Compute LCS kernels for canonical substrings of a vs b

Compute LCS kernels for every fragment of a vs b as sticky product of canonical substring kernels: overall time $O(mn)$



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]

Beyond semi-locality
Bounded-length Smith–Waterman alignment

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

Smith–Waterman (SW) alignment

For every prefix of a and every prefix of b, return maximum alignment score between their suffixes

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

SW-alignment: running time

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

Beyond semi-locality

Bounded-length Smith–Waterman alignment

SW-alignment

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

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

Beyond semi-locality

Bounded-length Smith–Waterman alignment

SW-alignment 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

Beyond semi-locality

Bounded-length Smith–Waterman alignment

Bounded-length one-sided SW (B1LSW) alignment

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

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

$$h_{1len \geq w}[l, j] = \max_{k,i;l-k \geq w} align\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$$

Bounded total length SW (BTLSW) alignment

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

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

$$h_{tlen \geq w}[l,j] = \max_{k,i; l-k+j-i \geq w} align\bigl(a\langle k : l\rangle, b\langle i : j\rangle\bigr)$$

B(1,T)LSW-alignment: 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 BTLSW-alignment: as BTLSW-alignment, except

- $|a'| + |b'| \geq (1 - \epsilon)w$
- $align(a', b') \geq h_{tlen \geq w}[l, j]$

Beyond semi-locality

Bounded-length Smith–Waterman alignment

B1LSW-alignment

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

Implicit unit-Monge matrix searching: $h_{1len=w}[l,j] = \max_{i \in [0:j]} H_{a\langle k:l \rangle,b}[i,j]$

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

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

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

Beyond semi-locality

Bounded-length Smith–Waterman alignment

Normalised SW-alignment [Arslan, Eğecioğlu, Pevzner: 2001]

$nalign(a, b) = \frac{align(a,b)}{|a|+|b|}$

Normalised bounded total length SW (BTLSW) alignment

Given $w \geq 0$, for every prefix of a and every prefix of b, return max normalised alignment score between their suffixes a', b', $|a'| + |b'| \geq w$

$$\hbar_{tlen \geq w}[l, j] = \max_{k,i; l-k+j-i \geq w} nalign\big(a\langle k : l\rangle, b\langle i : j\rangle\big)$$

Absolute normalised BTLSW-alignment

Given $w \geq 0$, return max normalised BTLSW-alignment score across all prefixes of a, b

$\max_{l,j} \hbar_{tlen \geq w}[l, j]$

Beyond semi-locality
Bounded-length Smith–Waterman alignment

Absolute normalised BTLSW-alignment: 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 BTLSW-alignment: as normalised BTLSW-alignment, except

- $|a'| + |b'| \geq (1-\epsilon)w$
- $nalign(a', b') \geq \hbar_{tlen \geq w}[l, j]$

Beyond semi-locality

Bounded-length Smith–Waterman alignment

Absolute normalised BTLSW-alignment

Fractional programming: binary search for optimal alignment score

$\leq \log n$ iterations

Each iteration: BTLSW-alignment with adjusted scoring scheme

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

String comparison by transposition networks
Transposition networks

Comparison network: a circuit of comparators, each sorting a pair of values

Classical model for non-branching merging, sorting, selection. . .

Comparison networks are visualised by wire diagrams

Transposition network: all comparisons are between adjacent wires

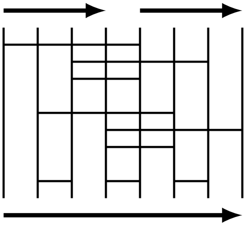
Sorting network: comparison/transposition network that sorts the input

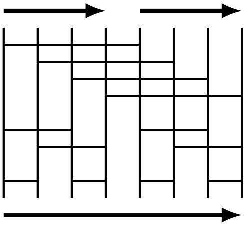
String comparison by transposition networks

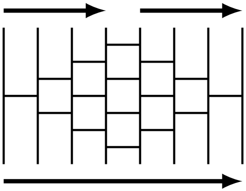
Transposition networks

Merging $n' + n'' = n$ values

	size	depth		
comp net	$O(n \log n)$	$O(\log n)$	optimal	[Batcher: 1968]
trans net	$n'n''$	$n' + n'' - 1$	optimal	bubblemerge

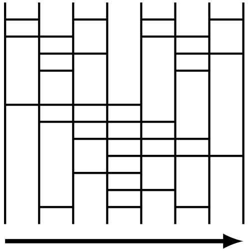
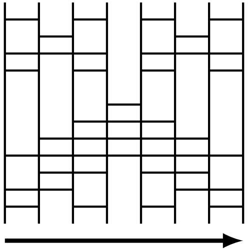
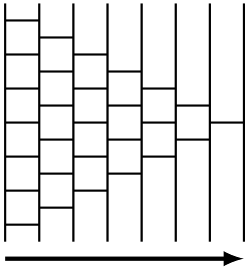






Sorting n values

	size	depth		
comp net	$O(n(\log n)^2)$	$O((\log n)^2)$		[Batcher: 1968]
	$O(n \log n)$	$O(\log n)$	optimal	[Ajtai+: 1983]
trans net	$\binom{n}{2} = \frac{n(n-1)}{2}$	$2n-3$	optimal	bubblesort or similar

String comparison by transposition networks

Transposition networks

Connections to

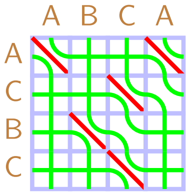
- graph theory (expanders)
- probability (rich theory of random transposition sorting networks)
- statistical mechanics (stochastic particle interaction processes)

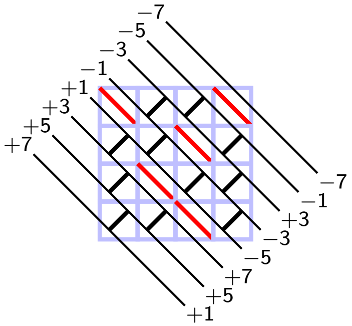
Applications: parallel algorithms, network design

String comparison by transposition networks

Transposition networks

Semi-local LCS: iterative combing as a transposition network

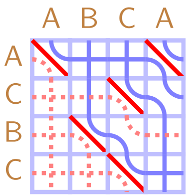


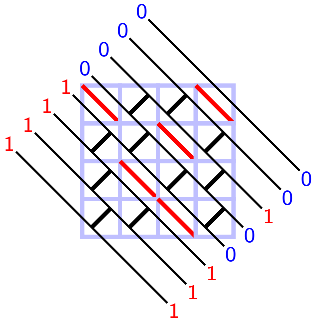


Inputs distinct, anti-sorted; each value traces a strand in sticky braid

Character mismatches correspond to comparators

Global LCS: iterative combing with binary input





Inputs 0/1, anti-sorted; sticky braid no longer defined uniquely (depends on whether equal values get swapped)

Parameterised string comparison

String comparison sensitive e.g. to

- low similarity: small $\lambda = lcs(a, b)$
- high similarity: small $\kappa = dist_{LCS}(a, b) = m + n - 2\lambda$

Can also use alignment score or edit distance

Assume $m = n$, therefore $\kappa = 2(n - \lambda)$

String comparison by transposition networks

Parameterised string comparison

Low-similarity comparison: small λ

- sparse set of matches, may need to look at them all
- preprocess matches for fast searching, time $O(n \log \sigma)$

High-similarity comparison: small κ

- set of matches may be dense, but only need to look at small subset
- no need to preprocess, linear search is OK

Flexible comparison: sensitive to both high and low similarity, e.g. by both comparison types running alongside each other

Parameterised string comparison: running time

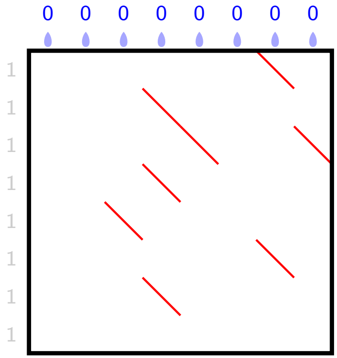
Low-similarity, after preprocessing in $O(n \log \sigma)$		
$O(n\lambda)$		[Hirschberg: 1977]
		[Apostolico, Guerra: 1985]
		[Apostolico+: 1992]

High-similarity, no preprocessing		
$O(n \cdot \kappa)$		[Ukkonen: 1985]
		[Myers: 1986]

Flexible		
$O(\lambda \cdot \kappa \cdot \log n)$	no preproc	[Myers: 1986; Wu+: 1990]
$O(\lambda \cdot \kappa)$	after preproc	[Rick: 1995]

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

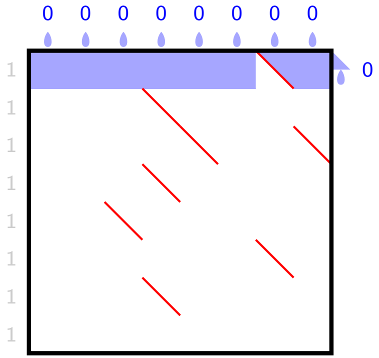


String comparison by transposition networks

Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

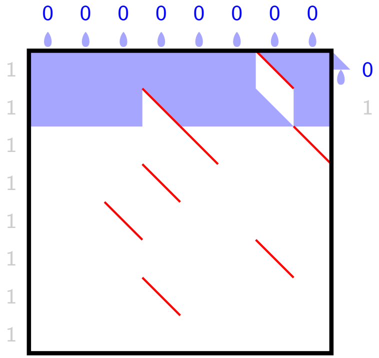


String comparison by transposition networks

Parameterised string comparison

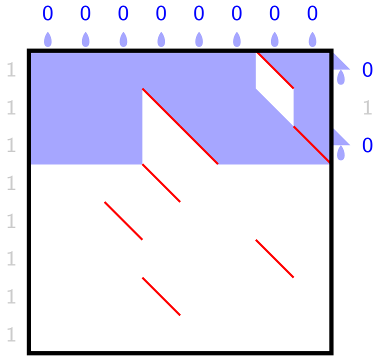
Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

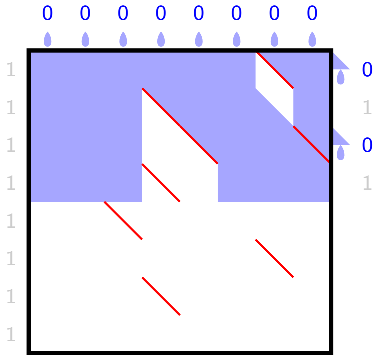


String comparison by transposition networks

Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

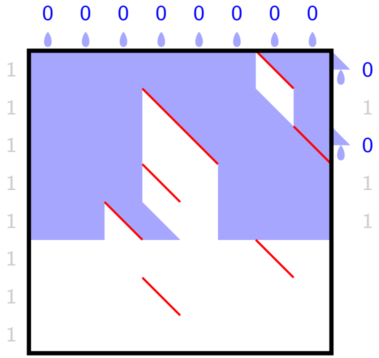
Low-similarity: $O(n \cdot \lambda)$



String comparison by transposition networks
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

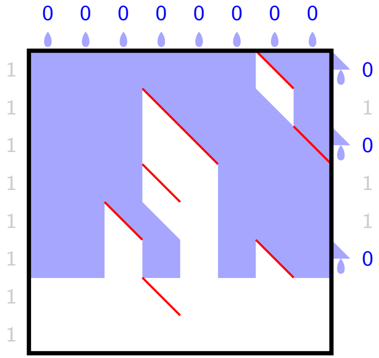


String comparison by transposition networks

Parameterised string comparison

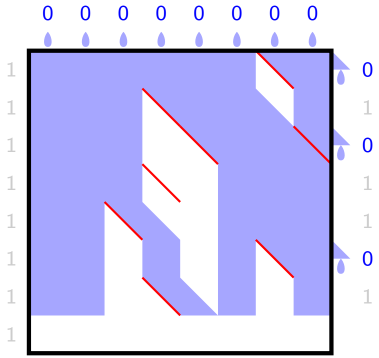
Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



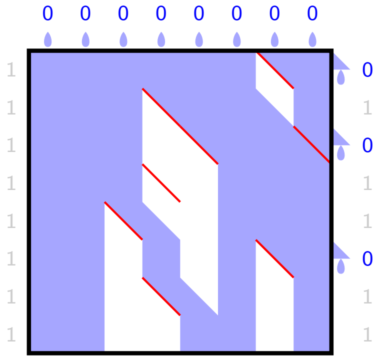
Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

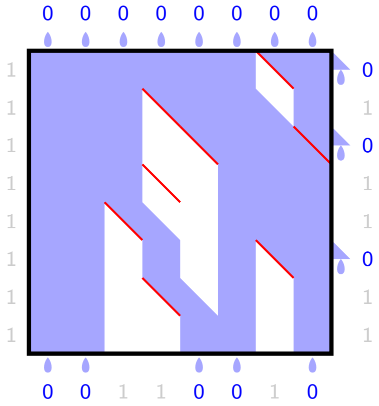


String comparison by transposition networks

Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

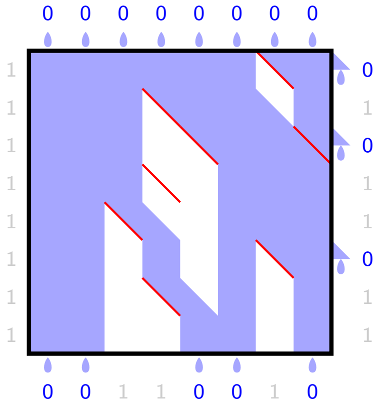


String comparison by transposition networks

Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

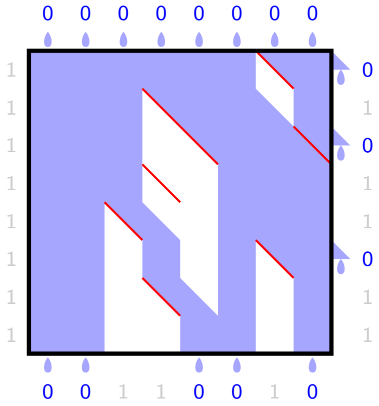


String comparison by transposition networks

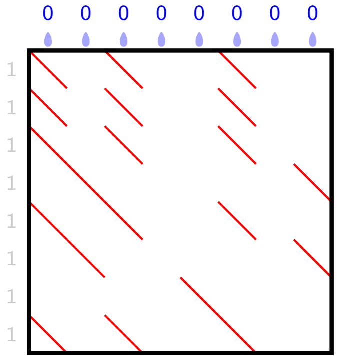
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



High-similarity: $O(n \cdot \kappa)$

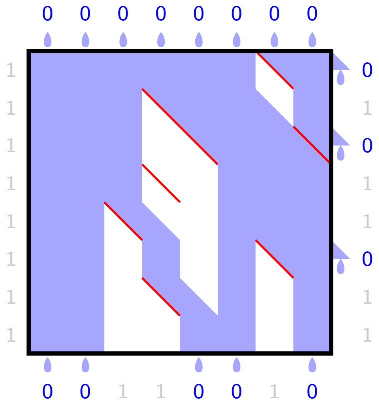


String comparison by transposition networks

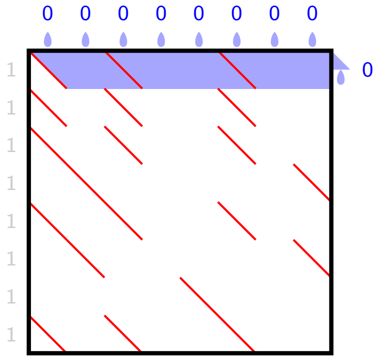
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

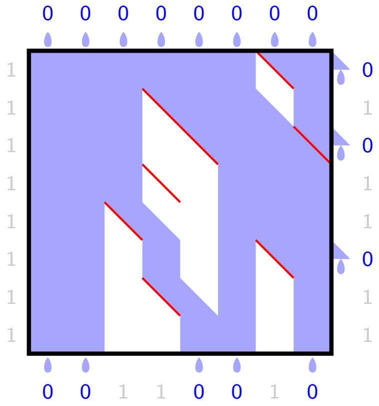


High-similarity: $O(n \cdot \kappa)$

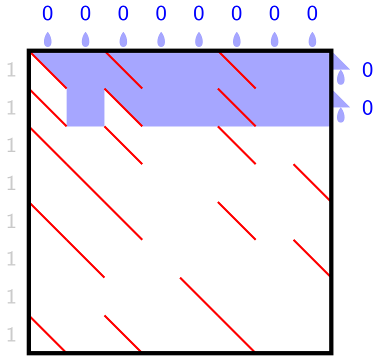


Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



High-similarity: $O(n \cdot \kappa)$



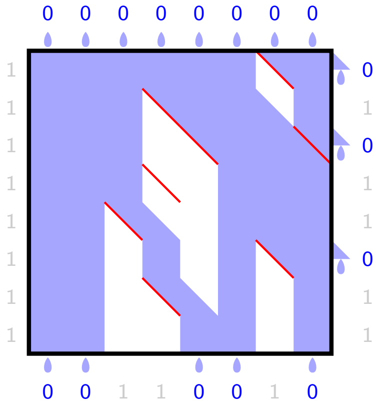
String comparison by transposition networks

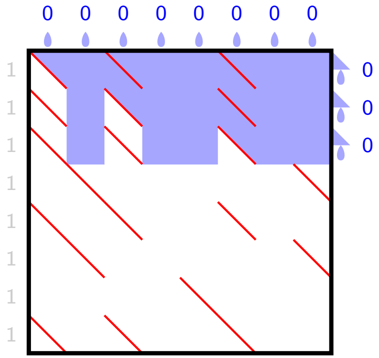
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

High-similarity: $O(n \cdot \kappa)$





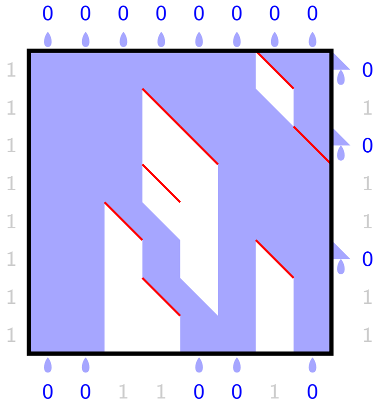
String comparison by transposition networks

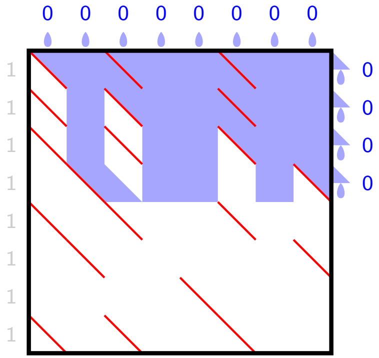
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

High-similarity: $O(n \cdot \kappa)$



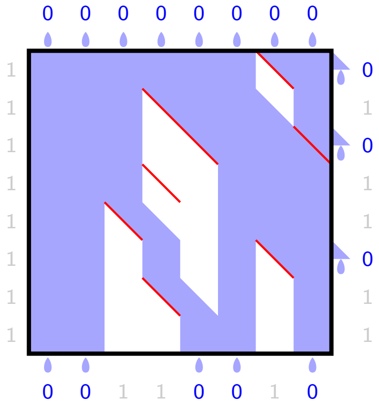


String comparison by transposition networks

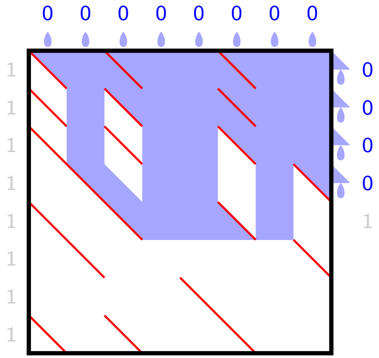
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



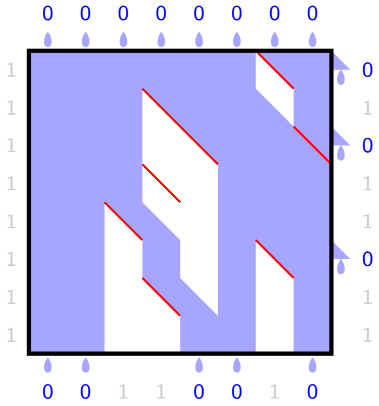
High-similarity: $O(n \cdot \kappa)$



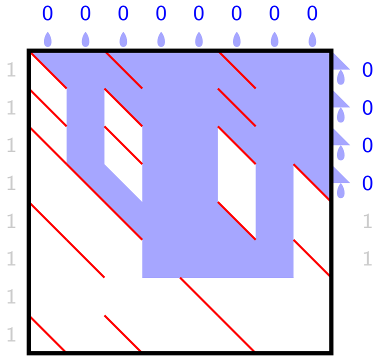
String comparison by transposition networks
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



High-similarity: $O(n \cdot \kappa)$

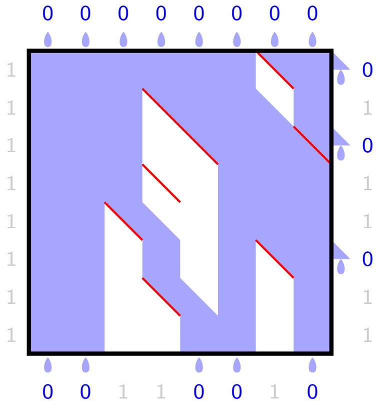


String comparison by transposition networks

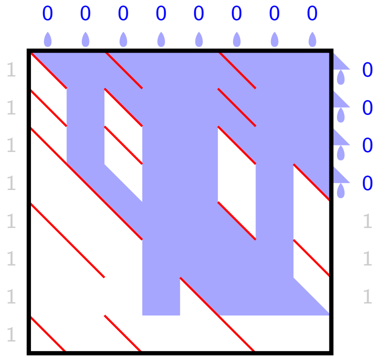
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



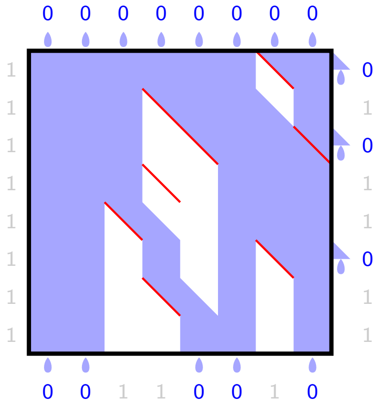
High-similarity: $O(n \cdot \kappa)$

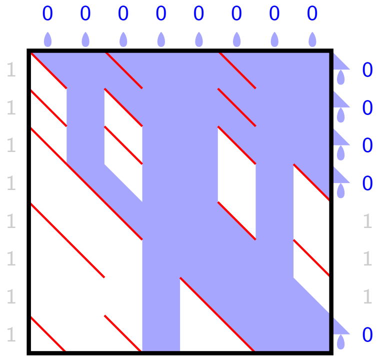


Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$

High-similarity: $O(n \cdot \kappa)$



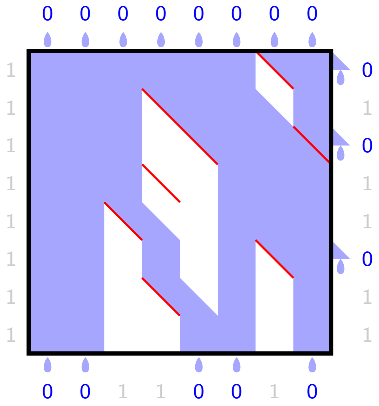


String comparison by transposition networks

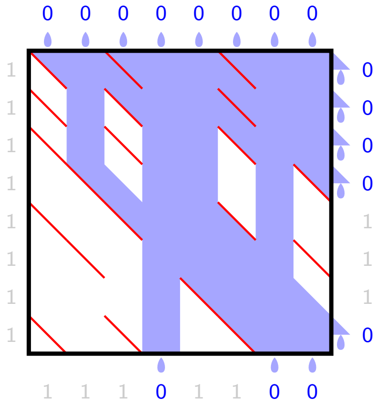
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



High-similarity: $O(n \cdot \kappa)$

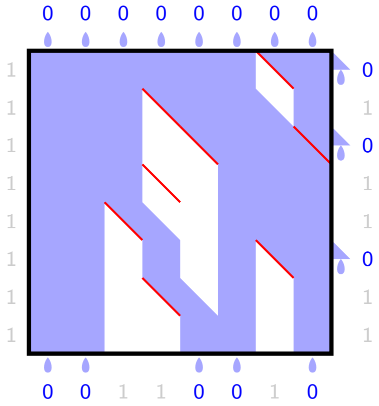


String comparison by transposition networks

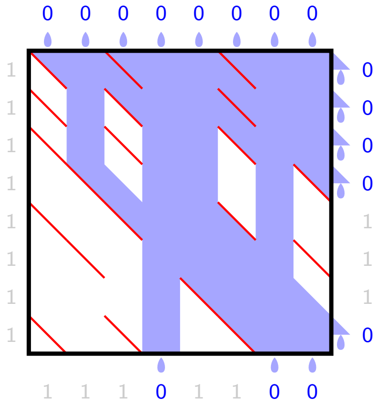
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



High-similarity: $O(n \cdot \kappa)$

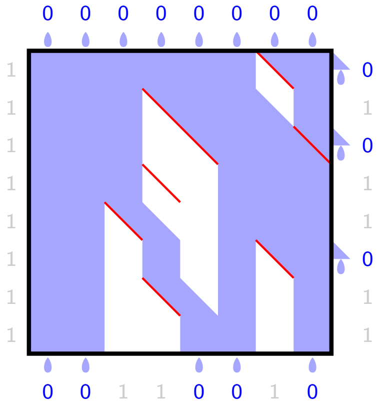


String comparison by transposition networks

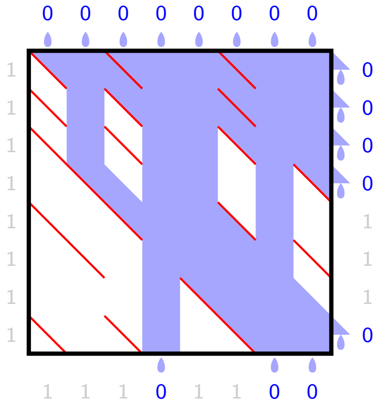
Parameterised string comparison

Parameterised string comparison: the waterfall algorithm

Low-similarity: $O(n \cdot \lambda)$



High-similarity: $O(n \cdot \kappa)$



Trace 0s through network in contiguous jets

String comparison by transposition networks
Comparing random strings

a, b: uniformly random permutation strings of length n, alphabet size n

Equivalent to LIS of a uniformly random permutation

$lcs(a, b) \sim 2n^{1/2} \qquad n \to \infty$ [Vershik, Kerov: 1977]

Transposition network: $n^2 - n$ random comparisons determined by a, b

String comparison by transposition networks
Comparing random strings

a, b: uniform Bernoulli sequences of length n, alphabet size $\sigma = O(1)$

$lcs(a, b) \sim \gamma_\sigma n \qquad n \to \infty$ [Chvátal, Sankoff: 1975]

γ_σ: Chvátal–Sankoff constants

Precise values of γ_σ unknown even for $\sigma = 2$

$0.7881 \leq \gamma_2 \leq 0.8263$ [Lueker: 2009; ...]

$\gamma_2 \approx 0.812$ by experiment [Cox, Bukh: 2020; ...]

Transposition network: $\sim n^2/\sigma$ random comparisons determined by a, b

Bernoulli model: pretend all n^2 comparisons are mutually independent (instead of just $2n$ mutually independent characters)

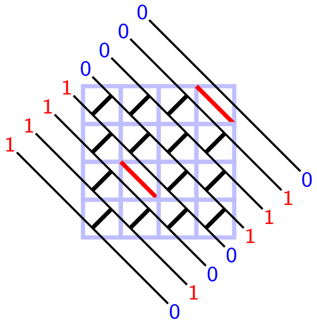
String comparison by transposition networks
Comparing random strings

Transposition network interpreted as a stochastic process:

- discrete-time TASEP particle-hole process (two linked processes, vertical and horizontal)
- Young diagram corner growth
- six-vertex model of statistical mechanics

Global scale: step initial condition (head of a traffic jam dissipated from initial shock in a rarefaction wave)

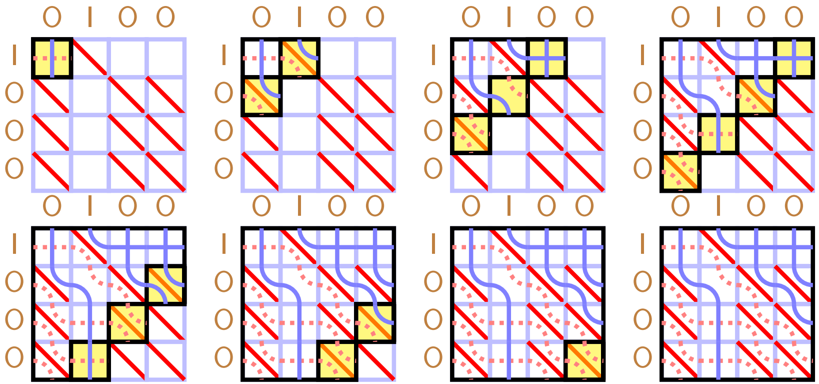
Local scale: space and time invariance



String comparison by transposition networks

Comparing random strings

Binary LCS: transposition network as a particle-hole process



Step initial condition: inputs 0/1, anti-sorted

String comparison by transposition networks
Comparing random strings

Hydrodynamic limit asymptotics provided by scalar conservation law PDE

$\frac{\partial}{\partial t}u + \frac{\partial}{\partial x}f(u) = 0$

$u(x,t)$: particle density

$f(u)$: flux determined by particle-hole swap rate

Bernoulli model: discrete-time TASEP with swap rate $1/2$

Exact analytic solution for step initial condition

$\gamma_2^B = 2(\sqrt{2} - 1) = 0.8284...$

[Seppäläinen: 1997; Majumdar, Nechaev: 2005; ...]

Refining the Bernoulli model [T: NEW]

Instead of uniform comparison probability $1/2$, set separate conditional probabilities for particle-hole, hole-particle, particle-particle, hole-hole comparison

Discrete-time TASEP swap rate now determined only by particle-hole comparison probability, which is free to be different from $1/2$ (if balanced out by the other three)

String comparison by transposition networks

Comparing random strings

Bernoulli approximation for random LCS

Free parameters of refined Bernoulli model fitted to constraints of the LCS model:

- unconditional comparison probability still $1/2$
- space and time invariance
- particle-hole symmetry (for strings of equal lengths)
- dependence of comparisons within LCS grid

System of algebraic equations of degree 7 in 5 variables

$\gamma_2^R = 0.81405\ldots$: just 0.002 away from experimental $\gamma_2 \approx 0.812$

Exact asymptotic for LCS model: still open problem (but there is hope)

Parallel string comparison
Bit parallelism

Bit-parallel computation: in bit vectors of size v with Boolean logic and integer arithmetic

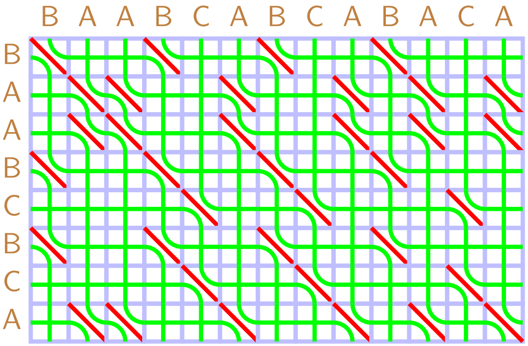
Bit-parallel LCS: running time

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

Parallel string comparison

Bit parallelism

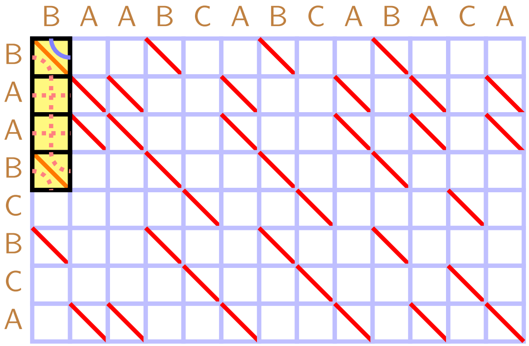
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

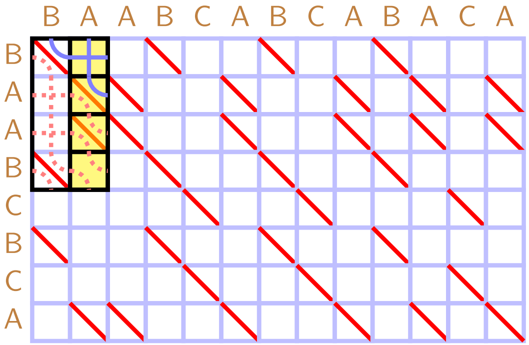
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

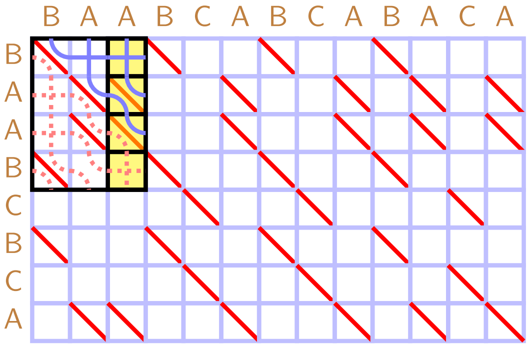
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

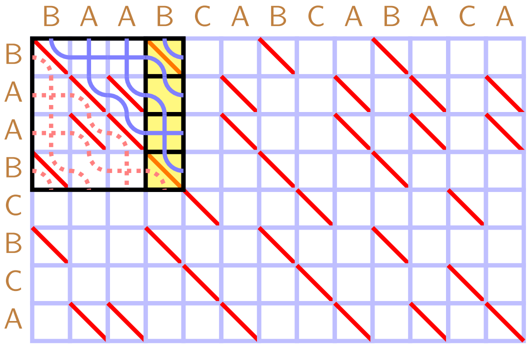
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

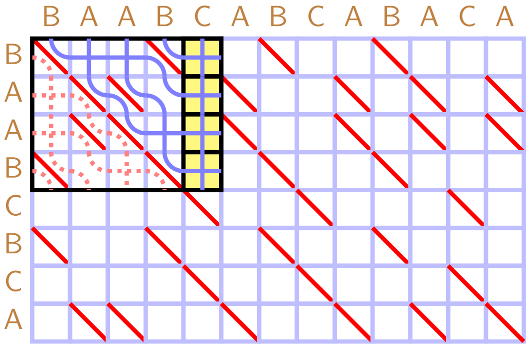
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

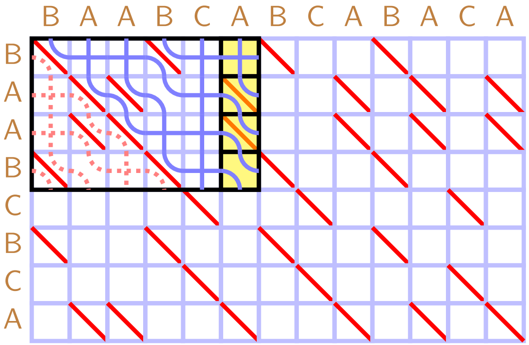
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

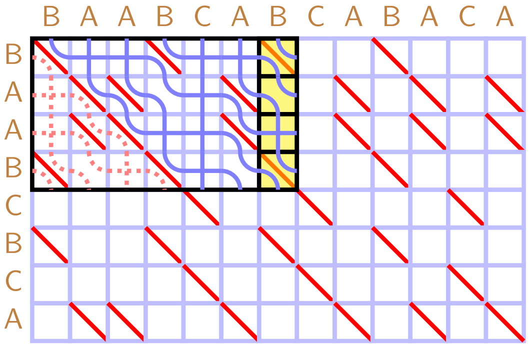
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

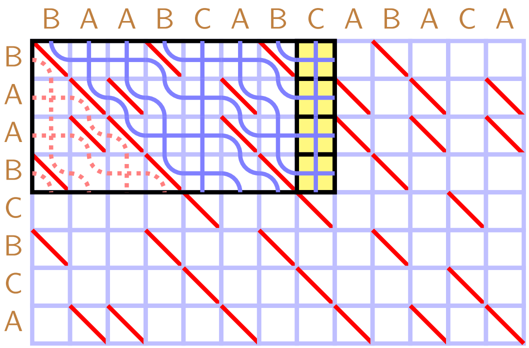
LCS by bit-parallel iterative combing (tiles)



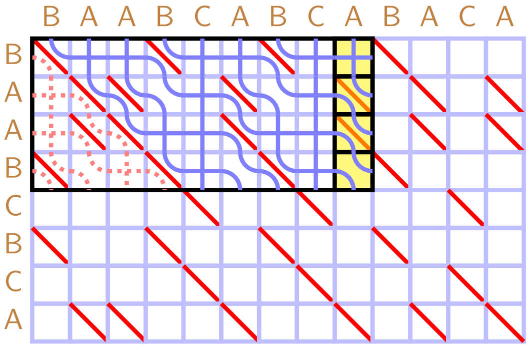
Parallel string comparison

Bit parallelism

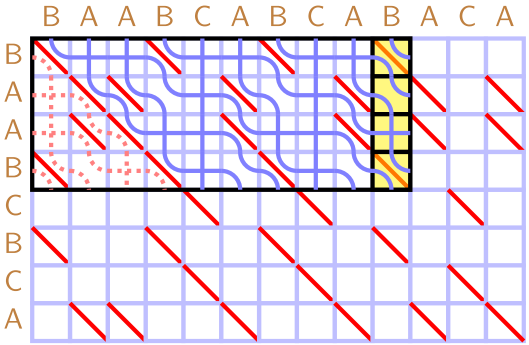
LCS by bit-parallel iterative combing (tiles)



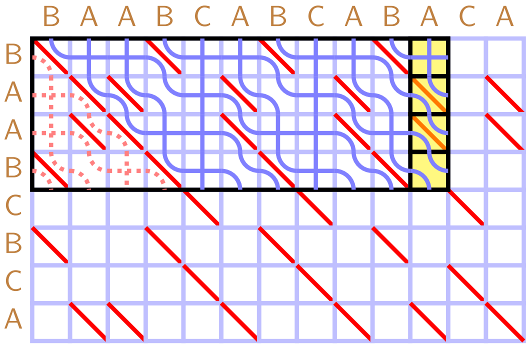
LCS by bit-parallel iterative combing (tiles)



LCS by bit-parallel iterative combing (tiles)

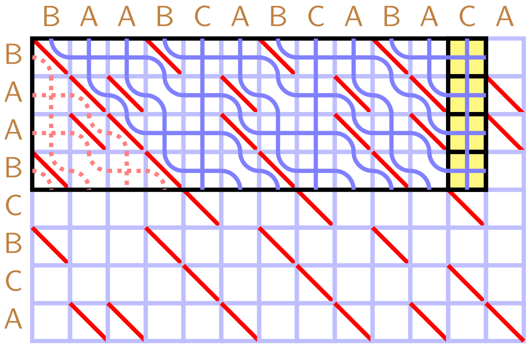


LCS by bit-parallel iterative combing (tiles)



Parallel string comparison
Bit parallelism

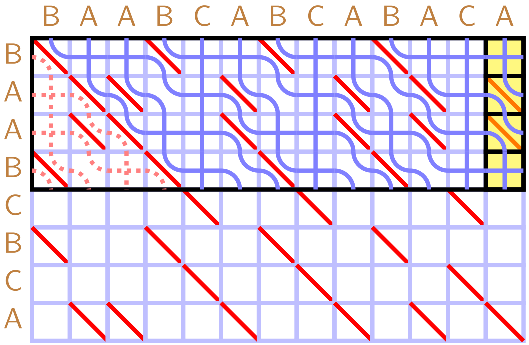
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

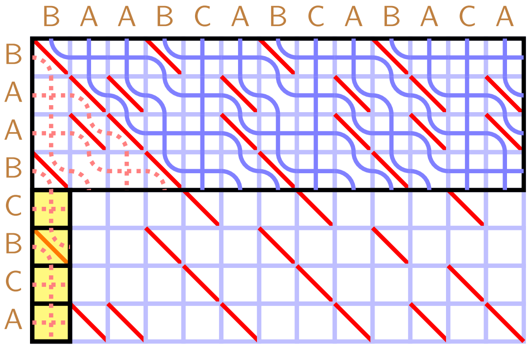
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

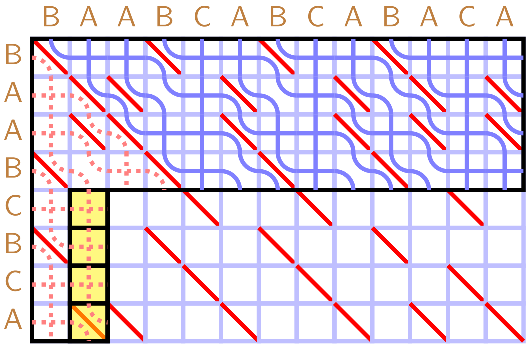
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

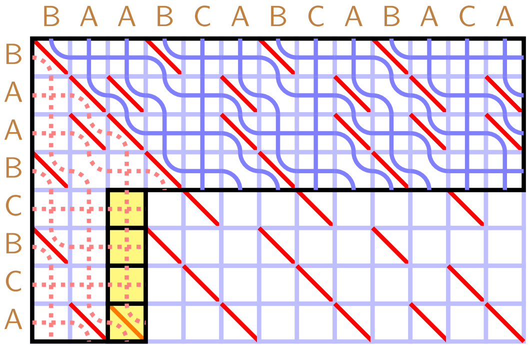
Bit parallelism

LCS by bit-parallel iterative combing (tiles)



Parallel string comparison
Bit parallelism

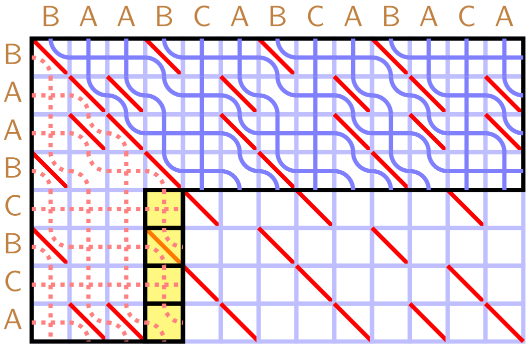
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

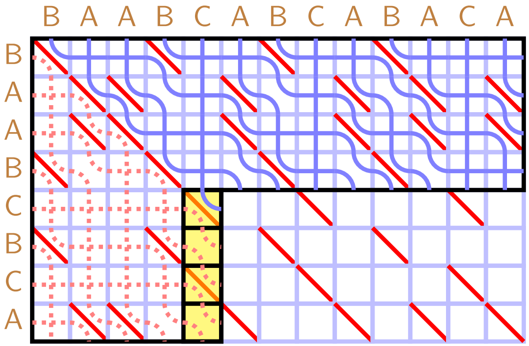
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

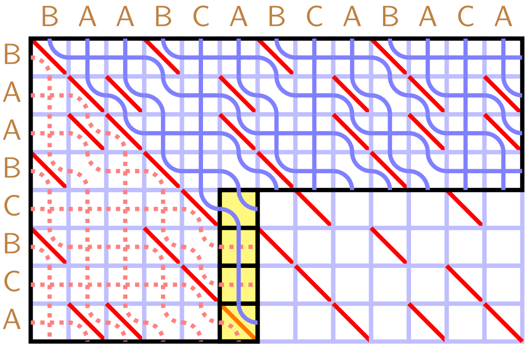
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

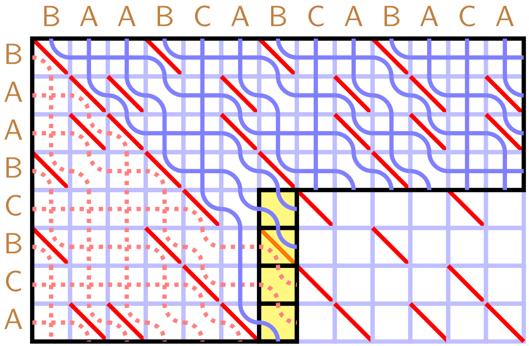
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

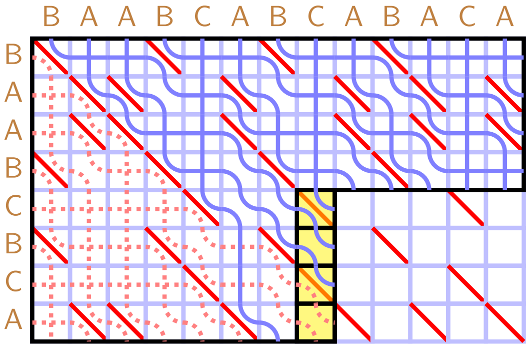
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

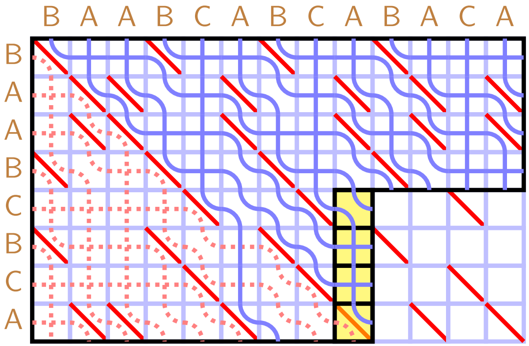
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

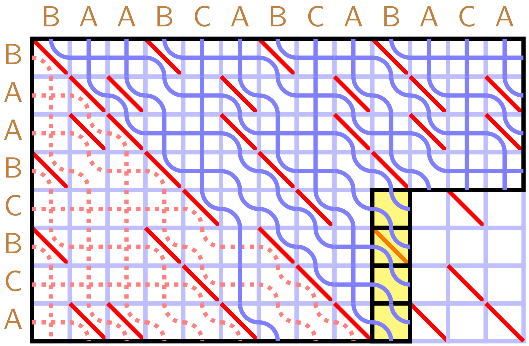
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

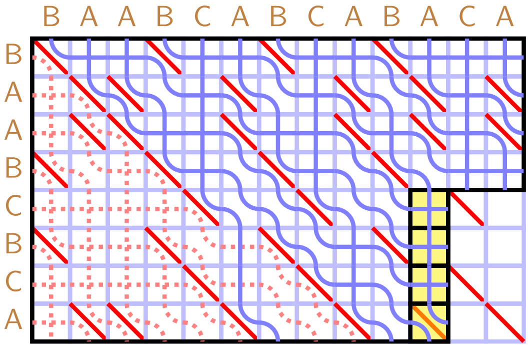
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

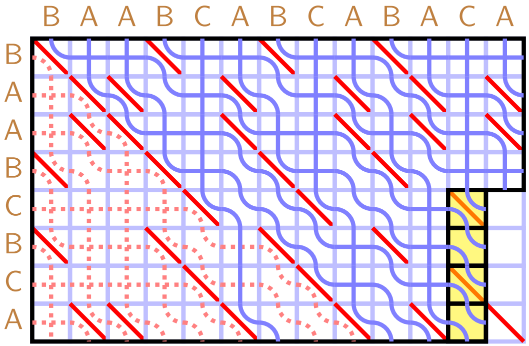
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

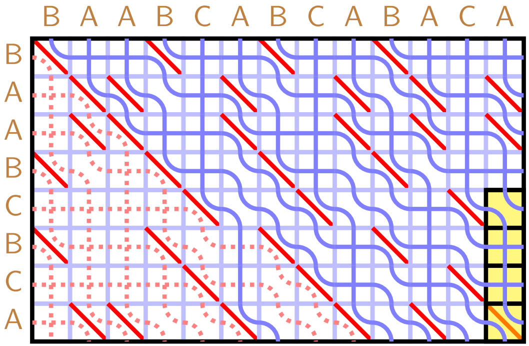
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

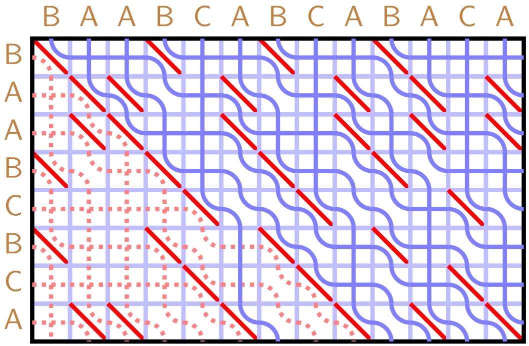
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

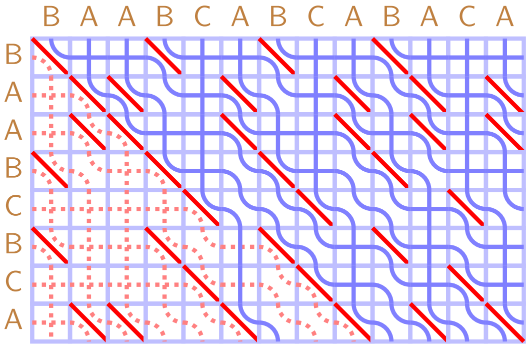
LCS by bit-parallel iterative combing (tiles)



Parallel string comparison

Bit parallelism

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

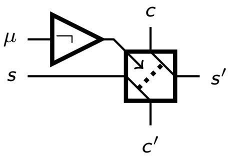
Overall time $O\left(\frac{mn}{v}\right)$

Parallel string comparison
Bit parallelism

LCS by bit-parallel iterative combing (tiles)

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

Combing bit logic: $(c', s') \leftarrow \mu\,?\,(s, c) : (s \wedge c, s \vee c)$



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

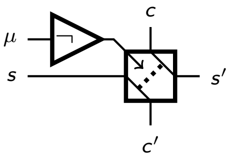
Parallel string comparison

Bit parallelism

LCS by bit-parallel iterative combing (tiles)

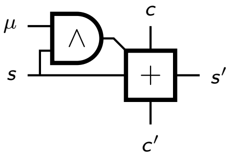
μ: match flag $a[i] = b[j]$ s, c: input bits

Combing bit logic: $(c', s') \leftarrow \mu\,?\,(s, c) : (s \wedge c, s \vee c)$



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

Implementation: $(c', s') \leftarrow s + (s \wedge \mu) + c$, 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 string comparison

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

Combing vector logic (carry bit passing left implicit)

- vertical tiles; single vector of size v
- arbitrary binary Boolean functions; shifts
- standard arithmetic, including $A + B + carry$
- precomputed match flags M, $\neg M$

...in 4 bit-vector operations

$(c', S') \leftarrow (S + (S \wedge M) + c) \vee (S \wedge \neg M)$ [Crochemore+: 2001]

Parallel string comparison

Bit parallelism

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

Combing vector logic (carry bit passing left implicit)

- vertical tiles; single vector of size v
- arbitrary binary Boolean functions; shifts
- standard arithmetic, including $A + B + carry$
- precomputed match flags M, $\neg M$

...in 4 bit-vector operations

$(c', S') \leftarrow (S + (S \wedge M) + c) \vee (S \wedge \neg M)$ [Crochemore+: 2001]

$(c', S') \leftarrow (S + (S \wedge M) + c) \vee (S - (S \wedge M))$ [Hyyrö: 2004]

Parallel string comparison
Bit parallelism

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

Combing vector logic (carry bit passing left implicit)

- vertical tiles; single vector of size v
- arbitrary binary Boolean functions; shifts
- standard arithmetic, including $A + B + \mathit{carry}$
- precomputed match flags M, $\neg M$

...in 4 bit-vector operations

$(c', S') \leftarrow (S + (S \wedge M) + c) \vee (S \wedge \neg M)$ [Crochemore+: 2001]

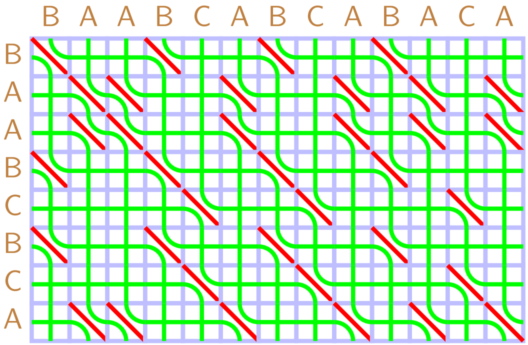
$(c', S') \leftarrow (S + (S \wedge M) + c) \vee (S - (S \wedge M))$ [Hyyrö: 2004]

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

Parallel string comparison

Bit parallelism

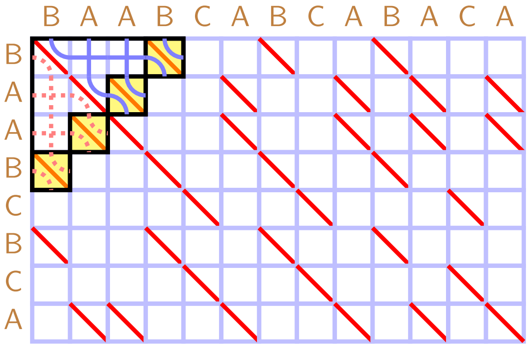
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

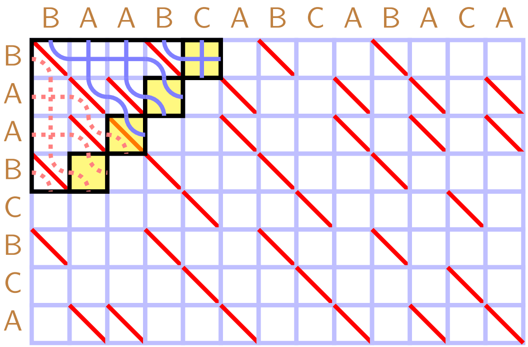
Bit parallelism

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



Parallel string comparison
Bit parallelism

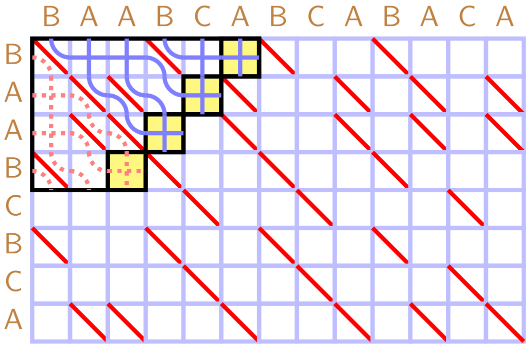
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

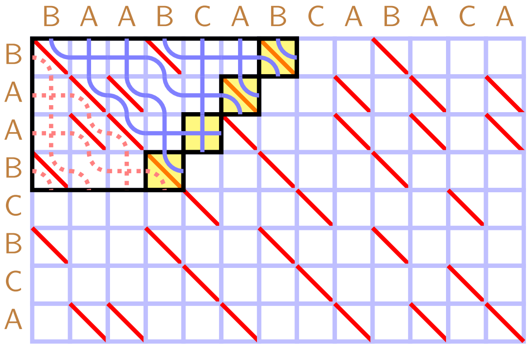
LCS by bit-parallel iterative combing (anti-diagonals)



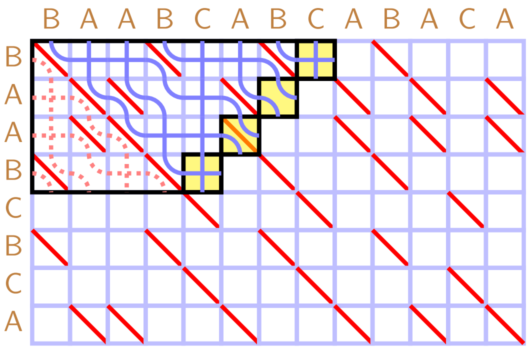
Parallel string comparison

Bit parallelism

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



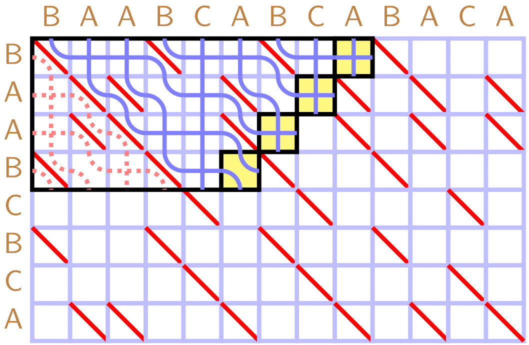
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

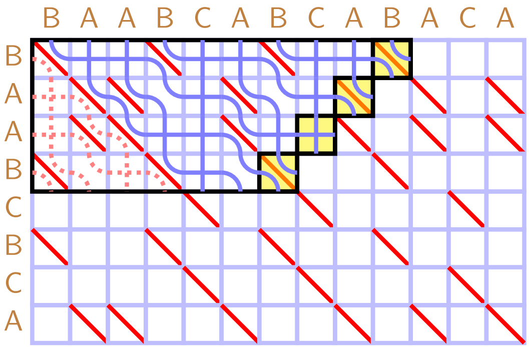
Bit parallelism

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



Parallel string comparison
Bit parallelism

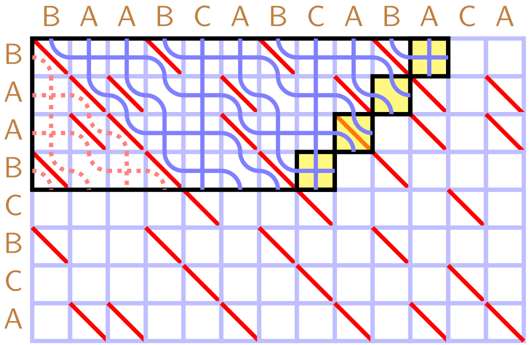
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

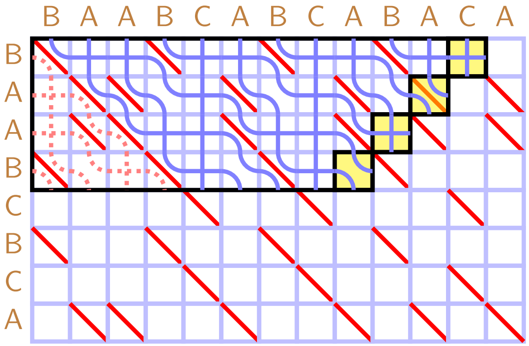
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

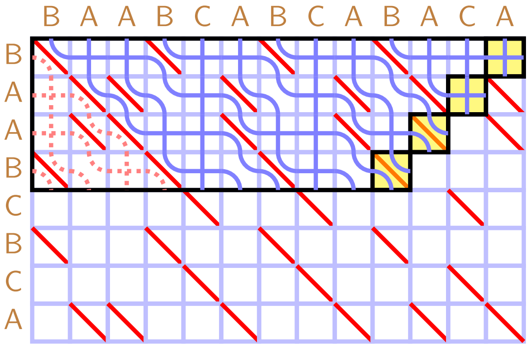
Bit parallelism

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



Parallel string comparison

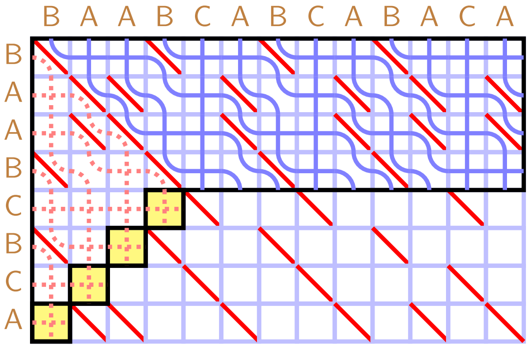
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

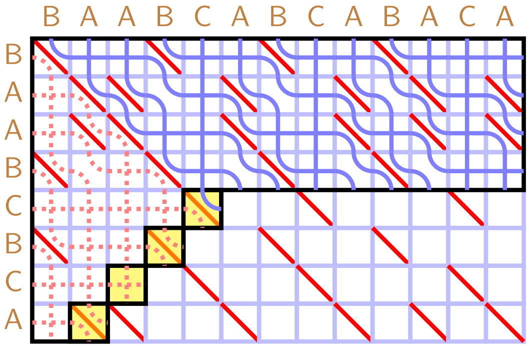
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

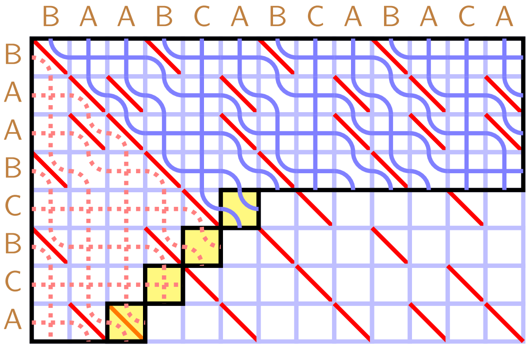
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

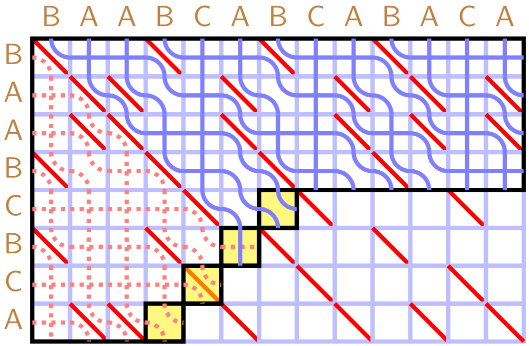
Bit parallelism

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



Parallel string comparison
Bit parallelism

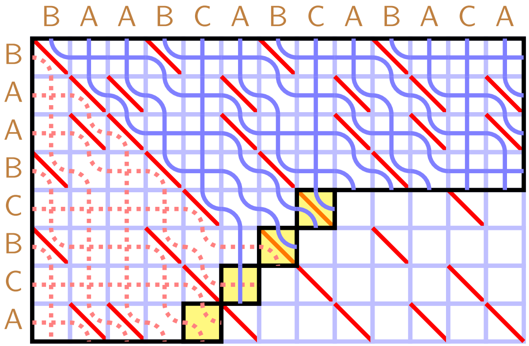
LCS by bit-parallel iterative combing (anti-diagonals)



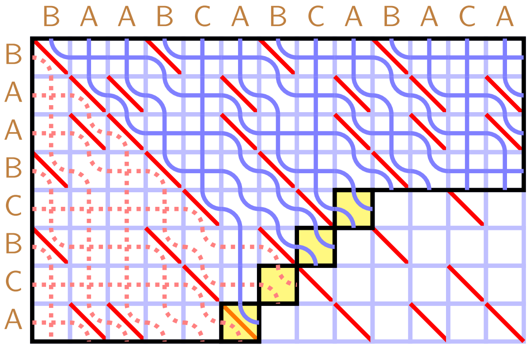
Parallel string comparison

Bit parallelism

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



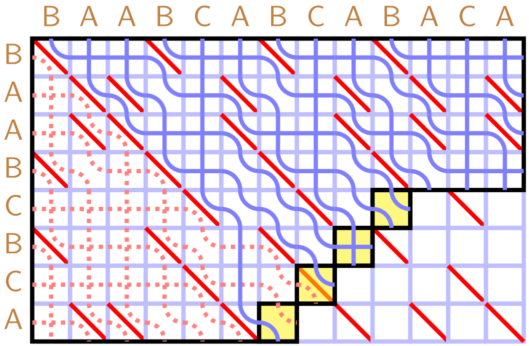
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

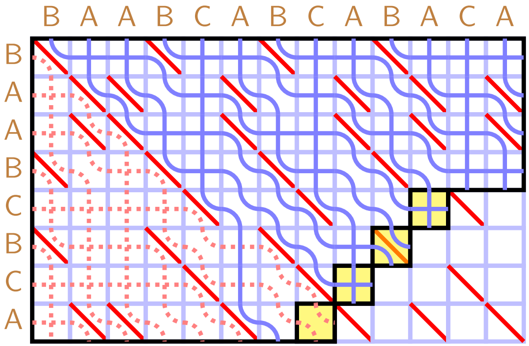
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

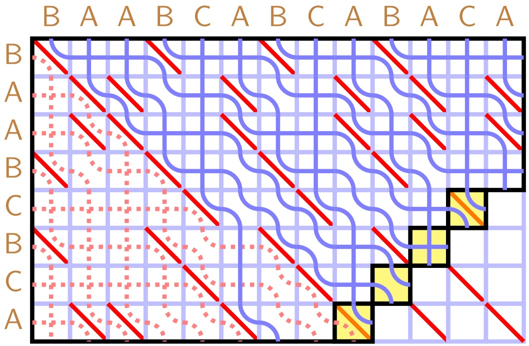
Bit parallelism

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

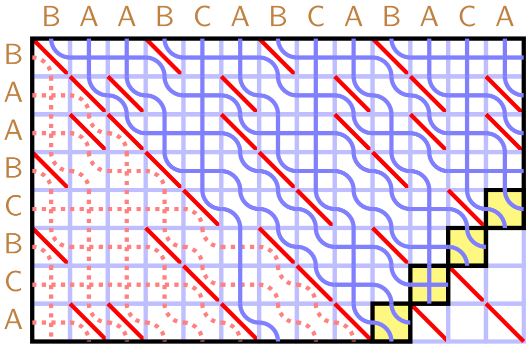


Parallel string comparison
Bit parallelism

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



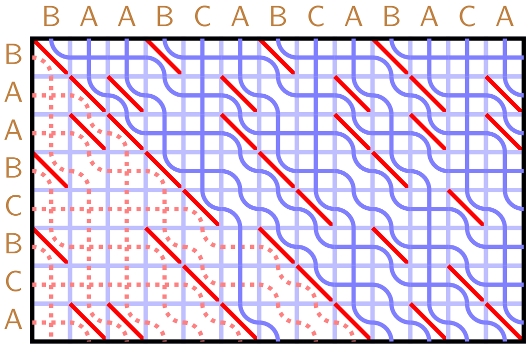
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

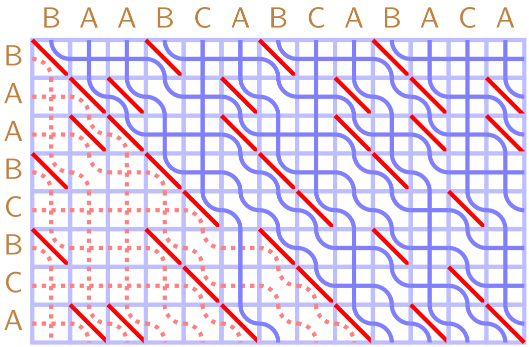
LCS by bit-parallel iterative combing (anti-diagonals)



Parallel string comparison

Bit parallelism

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



Parallel string comparison
Bit parallelism

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

Intel's AVX–512 instruction set: $32\times$ 512-bit registers

Each register accessible as vector of 512 bits

Supported by recent CPU microarchitectures

- Skylake (2017)
- Cannon Lake (2018)
- Cascade Lake (2019)
- ...

Also via Intel Software Development Emulator

Parallel string comparison
Bit parallelism

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

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

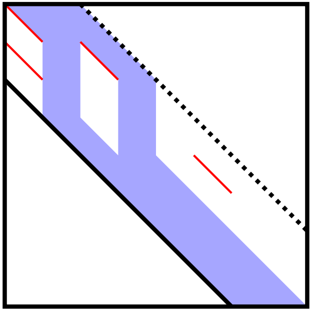
- arbitrary bitwise Boolean function of three bit-vector arguments
- in-place, replacing first argument

  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

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



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

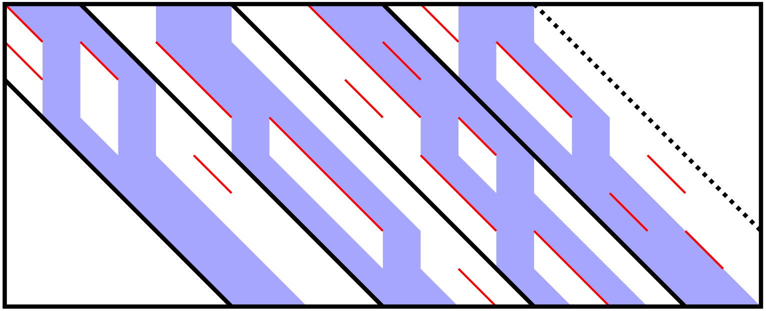
Band waterfall supported from below by separator matches

Parallel string comparison

Bit parallelism

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

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



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

Parallel string comparison

Vector parallelism

Vector-parallel computation

Computation with standard instructions on integer vectors of size v

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

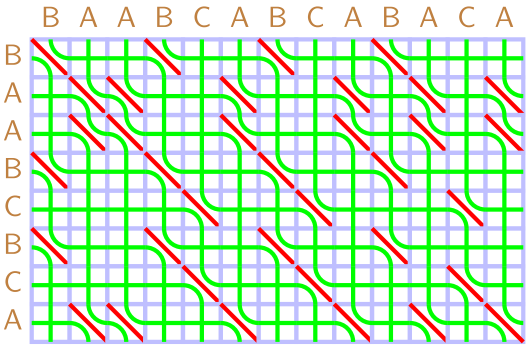
Combing two sticky braid strands in every cell

Vector cells independent

Parallel string comparison

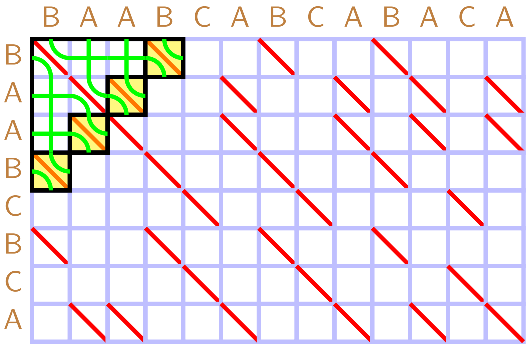
Vector parallelism

Semi-local LCS by vector-parallel iterative combing



Parallel string comparison
Vector parallelism

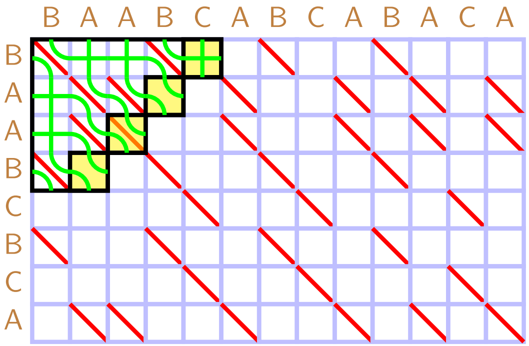
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

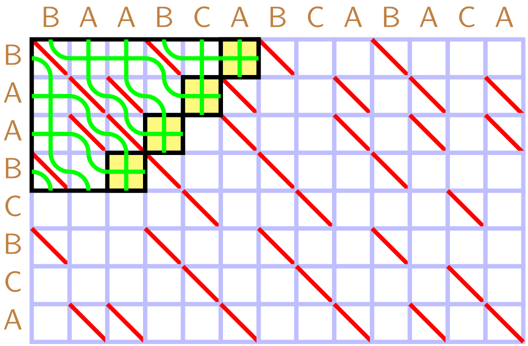
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

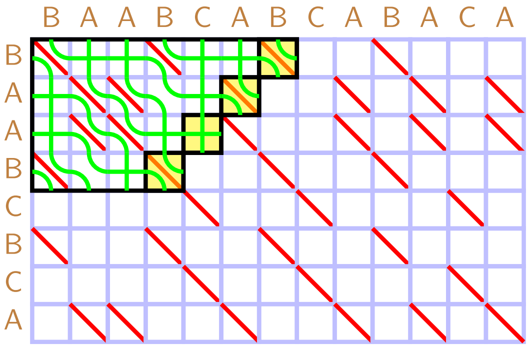
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

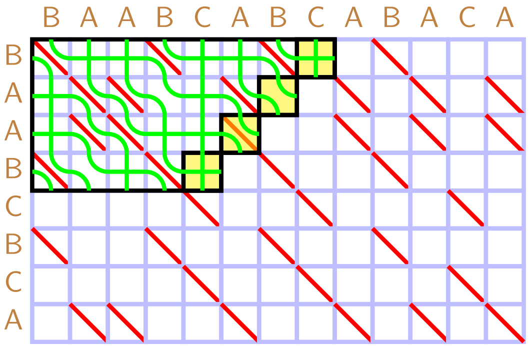
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

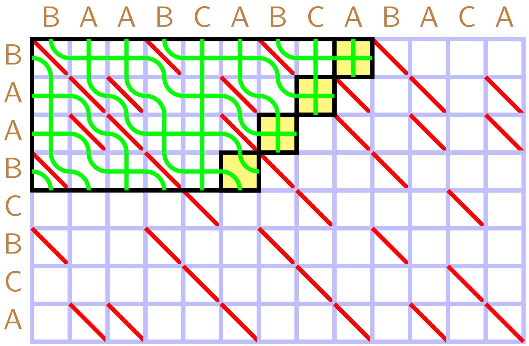
Semi-local LCS by vector-parallel iterative combing



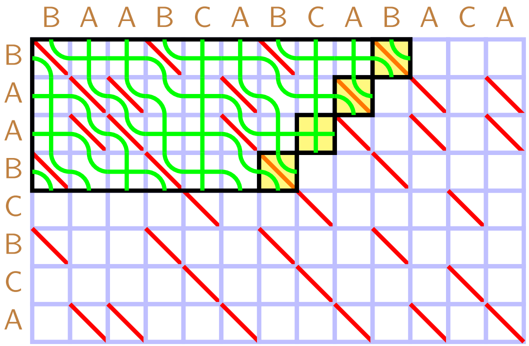
Parallel string comparison

Vector parallelism

Semi-local LCS by vector-parallel iterative combing



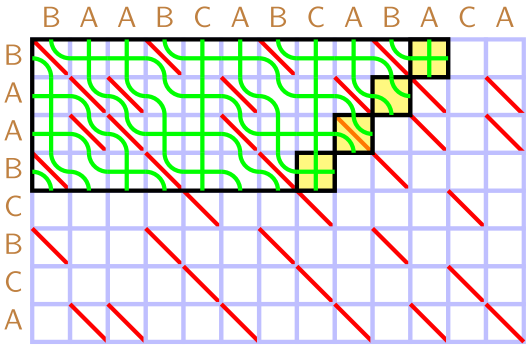
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

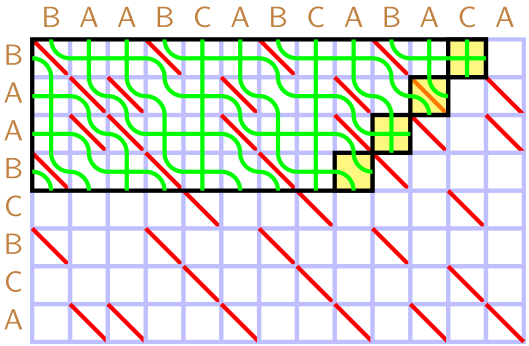
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

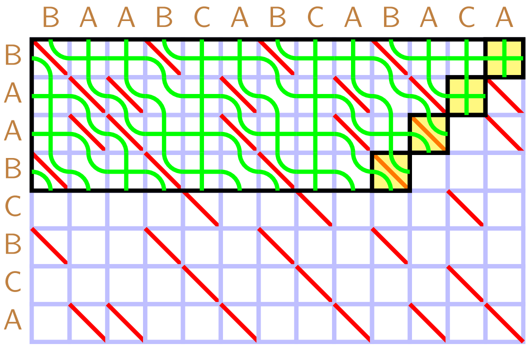
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

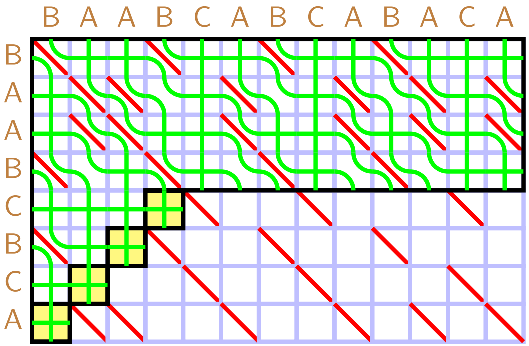
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

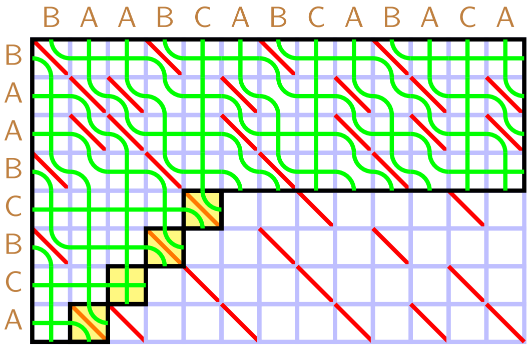
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

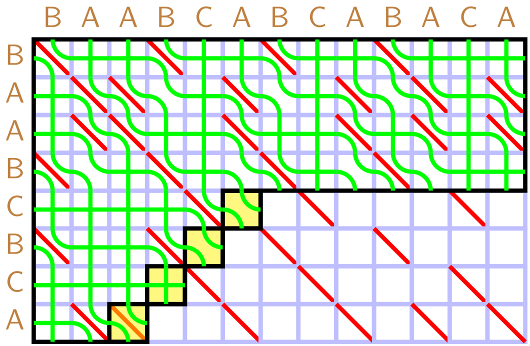
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

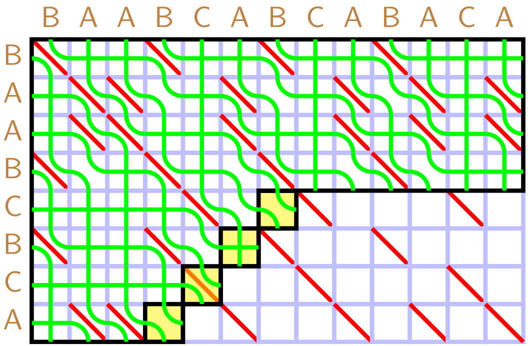
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

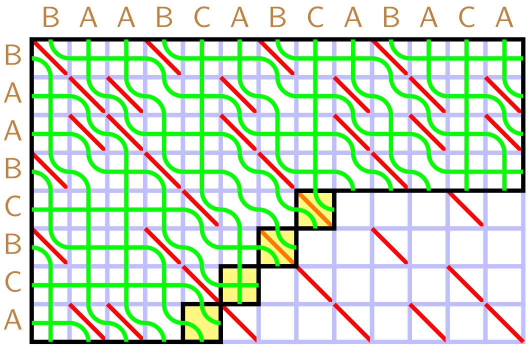
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

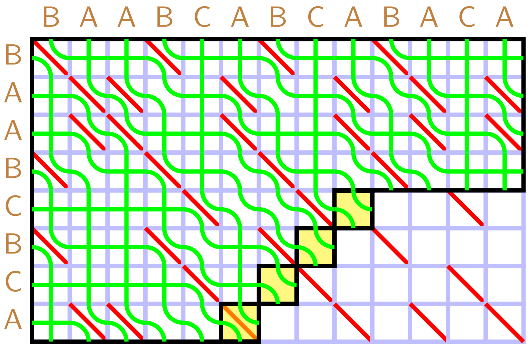
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

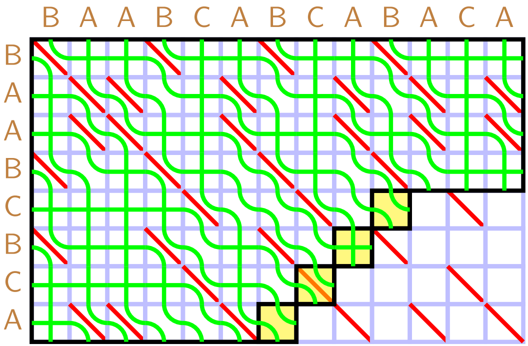
Vector parallelism

Semi-local LCS by vector-parallel iterative combing

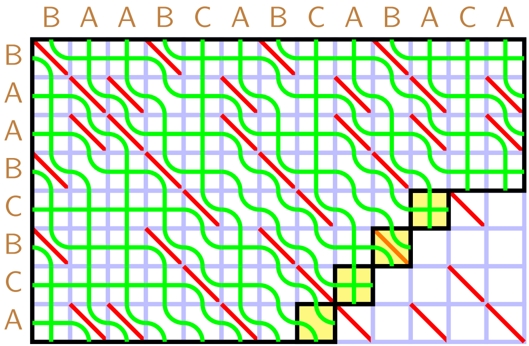


Parallel string comparison
Vector parallelism

Semi-local LCS by vector-parallel iterative combing



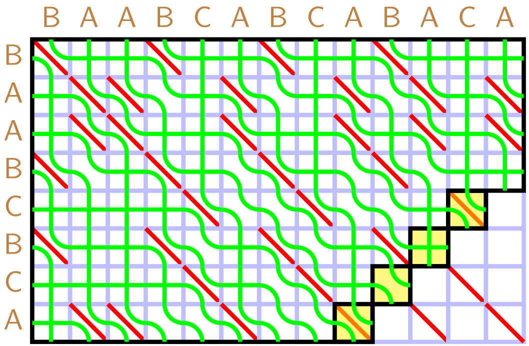
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

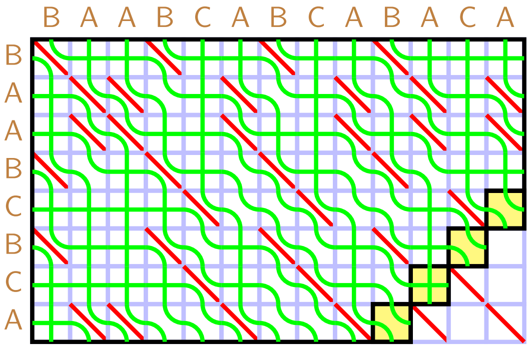
Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

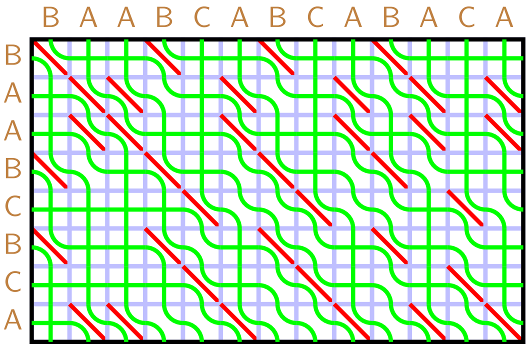
Semi-local LCS by vector-parallel iterative combing



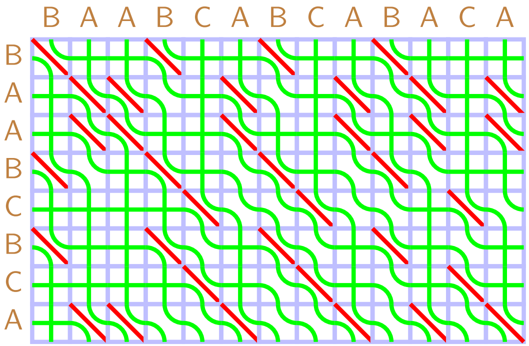
Parallel string comparison

Vector parallelism

Semi-local LCS by vector-parallel iterative combing



Semi-local LCS by vector-parallel iterative combing



Parallel string comparison

Vector parallelism

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

Initialise uncombed sticky braid: mismatch cell = crossing

Iterate over cells in antidiagonal vectors

- match cell: skip (keep strands uncrossed)
- mismatch cell: comb (uncross strands, if they crossed before)

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

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

Parallel string comparison
Vector parallelism

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

Implementation based on $O(mnw)$ repeated iterative combing approach, using some ideas from the optimal $O(mn)$ algorithm: resulting 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 string comparison

Vector parallelism

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

Used by biologists to study evolutionary conservation in non-coding DNA; helps to understand transcriptional regulation

More sensitive than BLAST and other heuristic (seed-and-extend) methods

Parallel string comparison

Vector parallelism

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

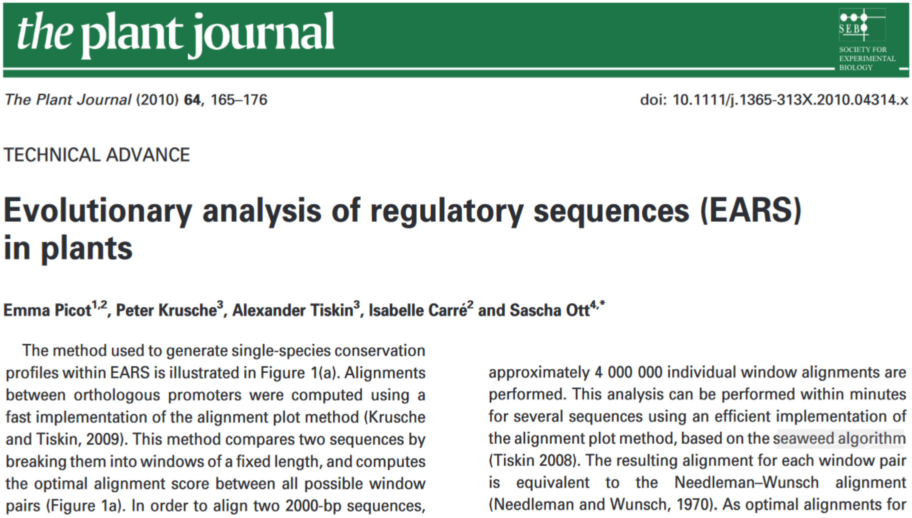
Conserved non-coding regions between distant relative species

Conservation for 100 mlns of years implies function

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

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

Parallel string comparison
Vector parallelism



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

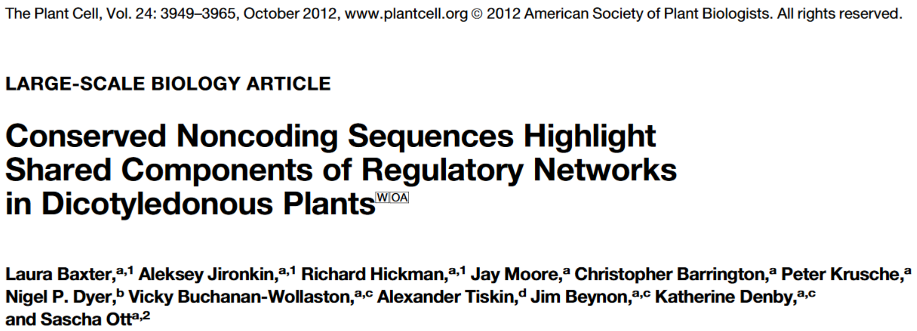
Evolutionary analysis of regulatory sequences (EARS) in plants

Emma Picot[1,2], Peter Krusche[3], Alexander Tiskin[3], Isabelle Carré[2] and Sascha Ott[4,*]

The method used to generate single-species conservation profiles within EARS is illustrated in Figure 1(a). Alignments between orthologous promoters were computed using a fast implementation of the alignment plot method (Krusche and Tiskin, 2009). This method compares two sequences by breaking them into windows of a fixed length, and computes the optimal alignment score between all possible window pairs (Figure 1a). In order to align two 2000-bp sequences, approximately 4 000 000 individual window alignments are performed. This analysis can be performed within minutes for several sequences using an efficient implementation of the alignment plot method, based on the seaweed algorithm (Tiskin 2008). The resulting alignment for each window pair is equivalent to the Needleman–Wunsch alignment (Needleman and Wunsch, 1970). As optimal alignments for

Parallel string comparison

Vector parallelism

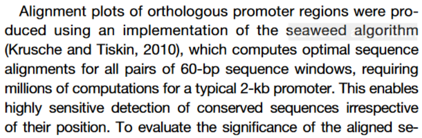


The Plant Cell, Vol. 24: 3949–3965, October 2012, www.plantcell.org © 2012 American Society of Plant Biologists. All rights reserved.

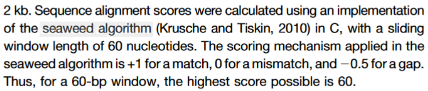
LARGE-SCALE BIOLOGY ARTICLE

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

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]



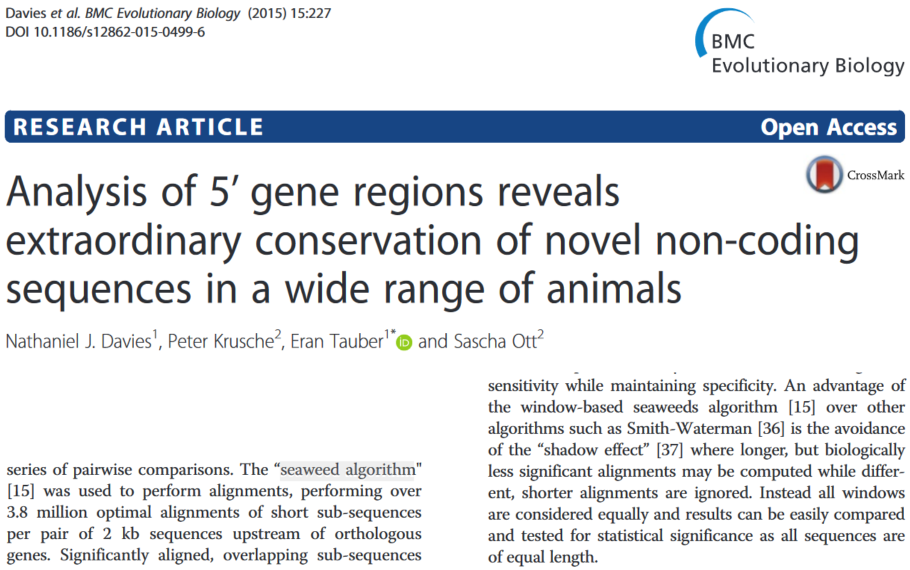
Alignment plots of orthologous promoter regions were produced using an implementation of the seaweed algorithm (Krusche and Tiskin, 2010), which computes optimal sequence alignments for all pairs of 60-bp sequence windows, requiring millions of computations for a typical 2-kb promoter. This enables highly sensitive detection of conserved sequences irrespective of their position. To evaluate the significance of the aligned se-



2 kb. Sequence alignment scores were calculated using an implementation of the seaweed algorithm (Krusche and Tiskin, 2010) in C, with a sliding window length of 60 nucleotides. The scoring mechanism applied in the seaweed algorithm is +1 for a match, 0 for a mismatch, and −0.5 for a gap. Thus, for a 60-bp window, the highest score possible is 60.

Parallel string comparison

Vector parallelism



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

BMC Evolutionary Biology

RESEARCH ARTICLE **Open Access**

CrossMark

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]

series of pairwise comparisons. The "seaweed algorithm" [15] was used to perform alignments, performing over 3.8 million optimal alignments of short sub-sequences per pair of 2 kb sequences upstream of orthologous genes. Significantly aligned, overlapping sub-sequences

sensitivity while maintaining specificity. An advantage of the window-based seaweeds algorithm [15] over other algorithms such as Smith-Waterman [36] is the avoidance of the "shadow effect" [37] where longer, but biologically less significant alignments may be computed while different, shorter alignments are ignored. Instead all windows are considered equally and results can be easily compared and tested for statistical significance as all sequences are of equal length.

Parallel string comparison
Vector parallelism

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

[Picot+: 2010] Evolutionary Analysis of Regulatory Sequences (EARS) in Plants. *Plant Journal.*

[Baxter+: 2012] Conserved Noncoding Sequences Highlight Shared Components of Regulatory Networks in Dicotyledonous Plants. *Plant Cell.*

Web service: `http://wsbc.warwick.ac.uk/ears`

Warwick press release (with video):
`http://warwick.ac.uk/press/discovery_of_100`

[Davies+: 2015] Analysis of $5'$ gene regions reveals extraordinary conservation of novel non-coding sequences in a wide range of animals. *BMC Evolutionary Biology.*

Parallel string comparison

Vector parallelism

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

Intel's AVX–512 instruction set: $32\times$ 512-bit registers

Each register accessible as vector of $64\times$ 8-bit ints; $32\times$ 16-bit ints; . . .

Supported by recent CPU microarchitectures

- Skylake (2017)
- Cannon Lake (2018)
- Cascade Lake (2019)
- . . .

Also via Intel Software Development Emulator

Parallel string comparison
Vector parallelism

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

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

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 string comparison

Vector parallelism

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 string comparison

Vector parallelism

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:

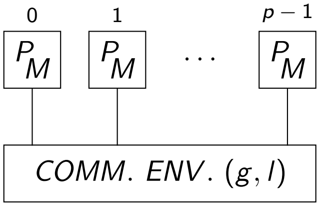
- $\times 2$ to prevent vector swapping between even/odd iterations
- $\times 16$ to keep all data in registers

Parallel string comparison

Bulk-synchronous parallelism

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

Simple, realistic general-purpose parallel model

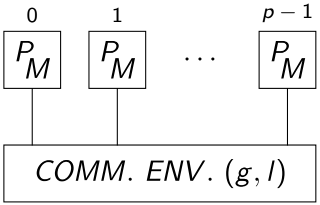


Parallel string comparison

Bulk-synchronous parallelism

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

Simple, realistic general-purpose parallel model



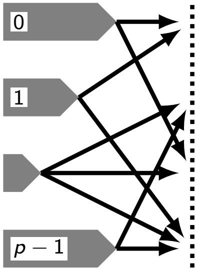
Contains

- 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 string comparison

Bulk-synchronous parallelism

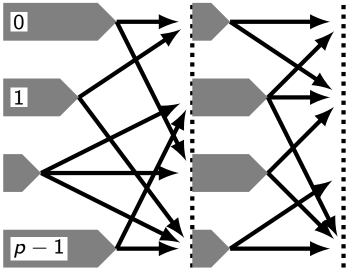
BSP computation: sequence of parallel supersteps



Parallel string comparison

Bulk-synchronous parallelism

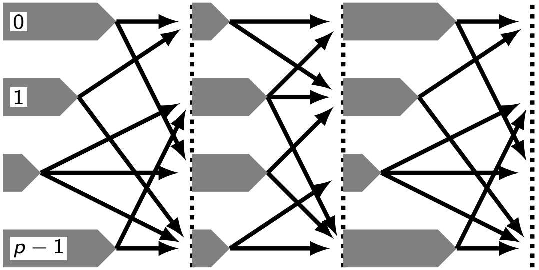
BSP computation: sequence of parallel supersteps



Parallel string comparison

Bulk-synchronous parallelism

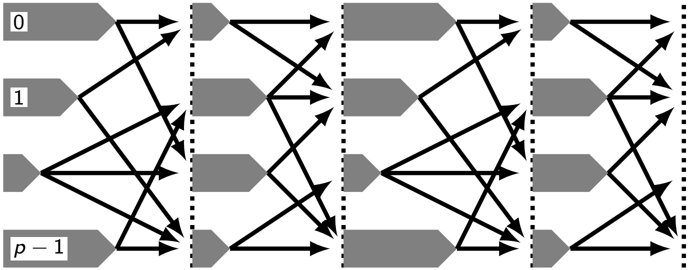
BSP computation: sequence of parallel supersteps



Parallel string comparison

Bulk-synchronous parallelism

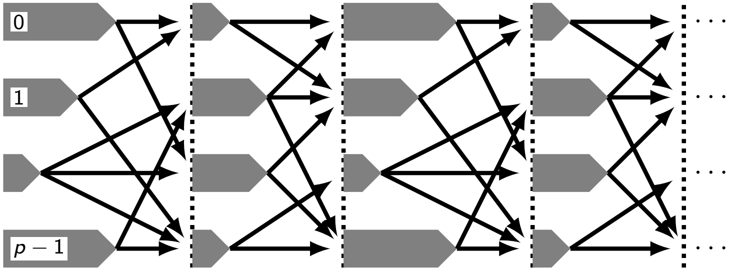
BSP computation: sequence of parallel supersteps



Parallel string comparison

Bulk-synchronous parallelism

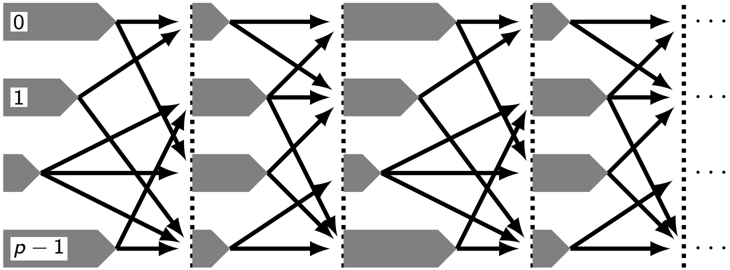
BSP computation: sequence of parallel supersteps



Parallel string comparison

Bulk-synchronous parallelism

BSP computation: sequence of parallel supersteps



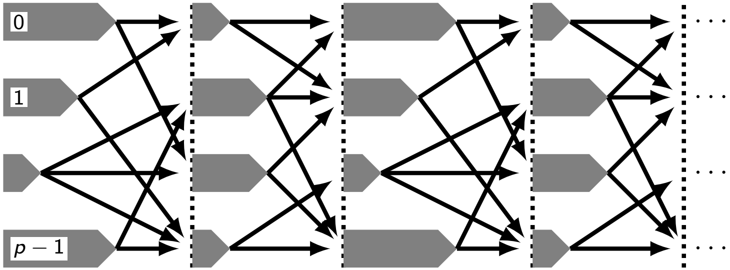
Asynchronous computation/communication within supersteps (includes data exchange with external memory)

Synchronisation before/after each superstep

Parallel string comparison

Bulk-synchronous parallelism

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

Parallel string comparison

Bulk-synchronous parallelism

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 string comparison

Bulk-synchronous parallelism

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*:

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

For the whole BSP computation with *sync* supersteps:

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

Parallel string comparison
Bulk-synchronous parallelism

For the whole BSP computation with *sync* supersteps:

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

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

Parallel string comparison

Bulk-synchronous parallelism

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:

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

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]

Parallel string comparison

Bulk-synchronous parallelism

The ordered 2D grid

$grid_2(n)$

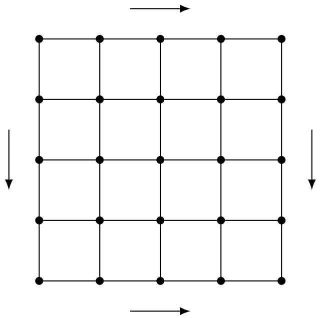
nodes arranged in an $n \times n$ grid

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 $2n - 1$



Parallel string comparison

The ordered 2D grid

$grid_2(n)$

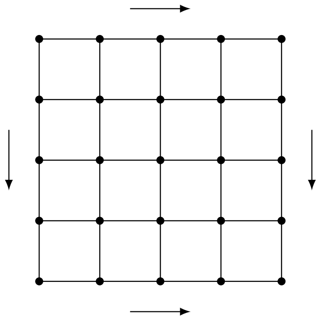
nodes arranged in an $n \times n$ grid

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 $O(n^2)$

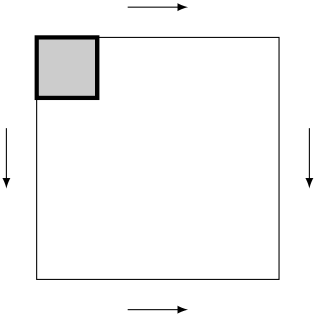
Parallel string comparison

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 string comparison

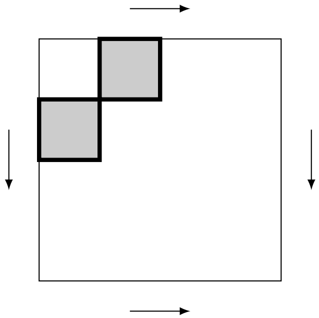
Bulk-synchronous parallelism

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



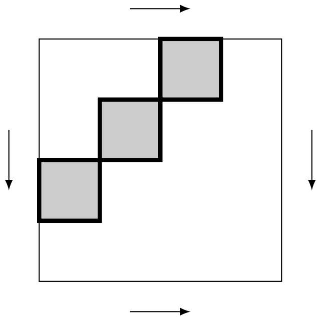
Parallel string comparison
Bulk-synchronous parallelism

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



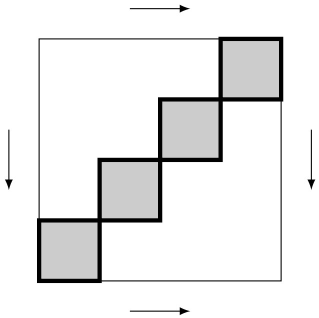
Parallel string comparison
Bulk-synchronous parallelism

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 string comparison

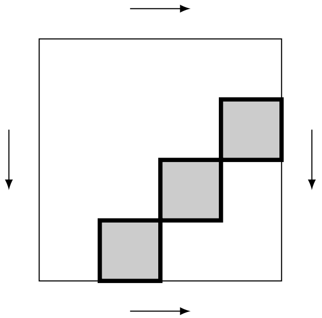
Bulk-synchronous parallelism

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 string comparison

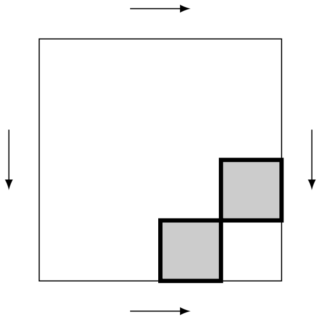
Bulk-synchronous parallelism

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 string comparison

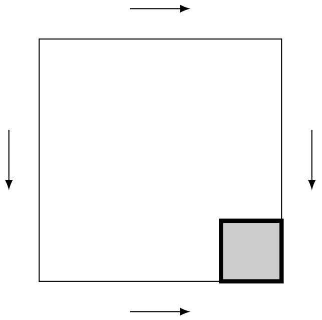
Bulk-synchronous parallelism

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 string comparison

Bulk-synchronous parallelism

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

Parallel string comparison

Bulk-synchronous parallelism

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

Parallel string comparison

Bulk-synchronous parallelism

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

$n \geq p$

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

BSP LCS

Classical dynamic programming mapped directly onto ordered 2D grid

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

Parallel string comparison

Bulk-synchronous parallelism

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

Parallel string comparison

Bulk-synchronous parallelism

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

Parallel string comparison

Bulk-synchronous parallelism

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 $O(n)$

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

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

$$\boxed{sync = O(1) + O(1) = O(1)}$$

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

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

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

$$\boxed{sync = O(1) + O(\log\log p) = O(\log\log p)}$$

Parallel string comparison

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

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

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

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

Parallel string comparison

Bulk-synchronous parallelism

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

Parallel string comparison

Bulk-synchronous parallelism

Matrix sticky multiplication: parallel Steady Ant (contd.)

Each processor

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

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

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

$$\boxed{sync = O(1) + O(1) + O(\log p) = O(\log p)}$$

Parallel string comparison

Bulk-synchronous parallelism

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

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

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

$$\boxed{sync = O(1) + O(1) + O(1) = O(1)}$$

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

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

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

$$\boxed{sync = O(1) + O(1) + O(\log p) = O(\log p)}$$

Parallel string comparison

Bulk-synchronous parallelism

Parallel string comparison

Bulk-synchronous parallelism

Parallel string comparison

Bulk-synchronous parallelism

Conclusions and open problems (1)

Implicit unit-Monge matrices:

- distance multiplication in time $O(n \log n)$
- isomorphic to sticky braids

Semi-local LCS:

- represented by implicit unit-Monge matrices
- generalises to rational alignment

Recursive and iterative combing:

- simple algorithms for semi-local LCS
- semi-local LCS in time $O(mn)$, and even slightly $o(mn)$

Conclusions and open problems (2)

Further string comparison:

- dynamic LCS with arbitrary indels
- efficient LCS on periodic strings

Sparse string comparison:

- fast LCS on permutation strings
- fast max-clique in a circle graph

Compressed string comparison:

- LCS on GC-strings
- approximate matching on plain pattern against GC-text

Conclusions and open problems (3)

Parallel string comparison:

- interpretation of sticky braids as transposition networks
- parameterised LCS
- bit-parallel and vector-parallel LCS
- random LCS: refined Bernoulli approximation
- comm- and sync- efficient parallel LCS

Beyond semi-locality:

- window alignment and sparse spliced alignment

Conclusions and open problems (4)

Open problems (theory):

- real-weighted LCS (sticky braids can be endowed with weights; resulting monoid is much larger than the unweighted one; fast multiplication seems difficult)
- compressed LCS: other compression models?
- parallel LCS: comm per processor $O\left(\frac{n}{p^{1/2}}\right)$, sync $O(1)$?
- lower bounds?
- random LCS: Chvatal–Sankoff constants. . .

Conclusions and open problems (5)

Open problems (applications):

- implementation (two teams in St Petersburg)
- good pre-filtering (e.g. q-grams)?
- further applications in biology
- applications in software engineering (code maintenance)
- application for multiple alignment? (NB: NP-hard)

Thank you