1 Introduction

Overview

A striking connection between three seemingly unrelated structures:

- **longest common subsequence** lengths under string concatenation
  \[ \text{lcs}(a', b), \text{lcs}(a'', b) \leadsto \text{lcs}(a = a'a'', b), \]

- **unit-Monge matrices** under distance (a.k.a. tropical) multiplication
  \[ P \circ Q = R : R[i, k] = \max_j(P[i, j] + Q[j, k]) \]

- **sticky braids** (a.k.a. Hecke words) under formal (sticky) multiplication
  \[ g_i^2 = g_i \quad g_i g_j = g_j g_i \quad \lvert j - i > 1 \rvert \quad g_i g_j g_i = g_i g_j g_i \quad \lvert j - i = 1 \rvert \]

In fact, these structures are all **isomorphic**

A surprisingly fast multiplication algorithm for these structures

Far-reaching algorithmic applications
“Squared paper” notation: $x^- = x - \frac{1}{2}$, $x^+ = x + \frac{1}{2}$

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

Half-integers $\hat{i}, \hat{j} \in \{ \ldots -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \ldots \} = ( -\infty : +\infty ) \supseteq (i, j)$

Planar dominance:

- $(i, j) \ll (i', j')$ iff $i < i'$ and $j < j'$ (“above-left” relation)
- $(i, j) \gg (i', j')$ iff $i > i'$ and $j < j'$ (“below-left” relation)

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

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

Standard approach to string comparison: **dynamic programming**

Our approach: the algebra of **sticky braids**

Can be used either for dynamic programming, or for **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 $D$:
$$D^\Sigma[i,j] = \sum_{i,j} D[i,j]$$

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

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

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

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

G. Monge (1746–1818)
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"): $P\Sigma$, where $P$ is a permutation matrix

$P$ used as implicit representation of $P\Sigma$

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

Efficient $P\Sigma$ queries: *range tree* on nonzeros of $P$ **[Bentley: 1980]**

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

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

$P\Sigma$ query

- $\succeq$-dominance counting: $\sum$ nonzeros $\succeq$-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 $O(\frac{\log n}{\log \log n})$  **[JáJá+: 2004]**

[Chan, Pătrașcu: 2010]
Distance multiplication

Distance semiring (a.k.a. (min, +)-semiring, tropical semiring)
- addition $\oplus$ given by min
- multiplication $\odot$ given by +

Matrix distance multiplication
\[ A \odot B = C \quad C[i, k] = \bigoplus_j (A[i, j] \odot B[j, k]) = \min_j (A[i, j] + B[j, k]) \]
Intuition: shortest path distances in weighted graphs

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

Recall: permutation matrices $= \text{implicit simple unit-Monge matrices}$

Matrix sticky multiplication (implicit distance multiplication)
\[ P \boxtimes Q = R \text{ iff } P^\Sigma \odot Q^\Sigma = R^\Sigma \]
The unit-Monge monoid $T_n$:
- permutation matrices under $\boxtimes$
- simple unit-Monge matrices under $\odot$
Isomorphic to the Hecke monoid $H_0(S_n)$

Sticky braids
\[ P \boxtimes Q = R \text{ can be seen as multiplication of sticky braids} \]
The classical braid group $B_n$:

- $n - 1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary crossings)
- $|B_n| = \infty$ canonical projection $B_n \to S_n$

Inversion:

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

Far commutativity:

$$g_i g_j = g_j g_i \quad j - i > 1$$

Braid relations:

$$g_i g_j g_i = g_j g_i g_j \quad j - i = 1$$

The sticky braid monoid $T_n$:

- $n - 1$ generators $g_1, g_2, \ldots, g_{n-1}$ (elementary crossings)
- $|T_n| = n!$ canonical bijection $T_n \leftrightarrow S_n$

Idempotence:

$$g_i^2 = g_i \quad \text{for all } i$$

Far commutativity:

$$g_i g_j = g_j g_i \quad j - i > 1$$

Braid relations:

$$g_i g_j g_i = g_j g_i g_j \quad j - i = 1$$

Special sticky braids in $T_n$

(Denote $P^R$ = counterclockwise rotation of $P$)

Identity $I$:

$$I \Box x = x$$

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

Zero $I^R$:

$$I^R \Box x = I^R$$

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

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

Structures related to $T_n$

<table>
<thead>
<tr>
<th></th>
<th>idem?</th>
<th>far comm?</th>
<th>braid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sticky braid monoid</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>positive braid monoid</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>locally free idemp. monid</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>nil-Hecke monoid</td>
<td>$g_i^2 = 0$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>symmetric group (Coxeter)</td>
<td>$g_i^2 = 1$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>classical braid group</td>
<td>$g_i g_i^{-1} = 1$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Generalisations of $T_n$

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

Sticky braids

Computing in $\mathcal{T}_n$: confluent rewriting system

Straightforward in computer algebra software (GAP, SAGE)

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

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

$\begin{align*}
aa &\rightarrow a \\
bb &\rightarrow b \\
ab &\rightarrow 0 \\
ba &\rightarrow 0
\end{align*}$

$T_4 = \{1, a, b, c, ab, ac, ba, bc, cb, aba, abc, acb, bac, bcb, cba, abcba, abacba, \text{ and } abacba = 0\}$

$\begin{align*}
aa &\rightarrow a \\
cc &\rightarrow c \\
ab &\rightarrow aba \\
bc &\rightarrow cbba \\
bb &\rightarrow b \\
cb &\rightarrow ac \\
cb &\rightarrow bcb \\
abacba &\rightarrow 0
\end{align*}$

Easy to use as a “black box”, but not efficient

<table>
<thead>
<tr>
<th>Type</th>
<th>Time</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>general ⊙</td>
<td>$O(n^3)$</td>
<td>$O(n^3 (\log \log n)^3)$ [Chan: 2007]</td>
</tr>
<tr>
<td>Monge ⊙</td>
<td>$O(n^2)$</td>
<td>via [Aggarwal+: 1987]</td>
</tr>
<tr>
<td>permutation □</td>
<td>$O(n^{1.5})$</td>
<td>[T: 2006]</td>
</tr>
<tr>
<td></td>
<td>$O(n \log n)$</td>
<td>[T: 2010]</td>
</tr>
</tbody>
</table>
Distance multiplication
Matrix sticky multiplication

\[ P_B,_{lo}, P_B,_{hi} \]

\[ P_A,_{lo}, P_A,_{hi} \]
\[ P_C \]

\[ P_A \]
\[ P_C \]

Matrix sticky multiplication: Steady Ant algorithm

\[ P \boxdot Q = R \]

Divide: split range of \( j \), remove zero rows/columns from each subproblem

Two subproblems of size \( n/2 \)

\[ P_{lo} \boxdot Q_{lo} = R_{lo} \quad P_{hi} \boxdot Q_{hi} = R_{hi} \]

Conquer: tracing a balanced trail from bottom-left to top-right \( R \)

Bad nonzeros: \( \ll \)-dominated in \( R_{hi} \); \( \ll \)-dominating in \( R_{lo} \)

Trail condition: \( |\{\text{bad nonzeros in } R_{hi}\}| = |\{\text{bad nonzeros in } R_{lo}\}| \)

Bad nonzeros removed; replaced by fresh nonzeros on the trail

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

Bruhat order

\[ P \preceq Q \text{ in the Bruhat order, if } Q \leadsto P \text{ by successive pair sorting} \]

\[
\begin{pmatrix}
0 & 1 & \cdots & \cdots & \cdots \\
\vdots & \ddots & \ddots & \vdots & \vdots \\
1 & \cdots & 0 & \cdots & \cdots \\
\vdots & \vdots & \ddots & \ddots & \ddots \\
\vdots & \vdots & \cdots & \ddots & \ddots \\
\end{pmatrix} \leftrightarrow 
\begin{pmatrix}
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots \\
\end{pmatrix}
\]

Intuitively, \( P \preceq Q \) means “\( P \) is more sorted than \( Q \)”

\( I \preceq X \preceq I^R \)
Distance multiplication

Bruhat order

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{pmatrix}
\preceq
\begin{pmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{pmatrix}
\preceq
\begin{pmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{pmatrix}
\]

Bruhat order is compatible with sticky multiplication: \( P \preceq Q \) implies \( R \Box P \preceq R \Box Q \) and \( P \Box R \preceq Q \Box R \).

- 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)

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

\( P \preceq Q \) iff \( P^\Sigma \leq Q^\Sigma \) elementwise

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{pmatrix}^\Sigma =
\begin{pmatrix}
0 & 1 & 2 & 3 \\
0 & 0 & 1 & 2 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\preceq
\begin{pmatrix}
1 & 0 & 2 & 3 \\
1 & 2 & 2 & 2 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
\end{pmatrix}
\]

Time \( O(n^2) \)

Sticky product criterion

\( P \preceq Q \) iff \( P^R \Box Q = I^R \) iff \( P \Box Q^{R \Box R} = I^R \)

\( P^R = \) counterclockwise rotation of \( P \)

Proof: \( P \preceq Q \) implies \( I^R = P^R \Box P \preceq P^R \Box Q \), hence \( I^R = P^R \Box Q \)

Conversely, if \( P^R \Box Q = I^R \) 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

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix} \preceq
\begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 1 & 0 \\
\end{bmatrix} R \begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
= \begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

P

\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

Q

P^R

\[
\begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
0 & 1 & 0 \\
\end{bmatrix}
\]

Q

= I^R

Alexander Tiskin (Warwick) The algebra of string comparison 37 / 185

Introduction

1

Monge and unit-Monge matrices

2

Distance multiplication

3

Longest common subsequence

4

Alignment and edit distance

5

Sparse string comparison

7

Compressed string comparison

6

Beyond semi-locality

8

Parallel string comparison

9

Conclusions and open problems

10

Further string comparison

11

Alexander Tiskin (Warwick) The algebra of string comparison 38 / 185

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] ← X[k] + Δ
- prefixsum(k): return \(\sum_{1 \leq i \leq k} X[i]\)

Query time:

- \(\Theta(\log n)\) in semigroup or group model
- \(\Theta\left(\frac{n \log n}{\log \log n}\right)\) in RAM model on integers

[Pătraşcu, 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 \(O\left(\frac{n \log n}{\log \log n}\right)\)?

Alexander Tiskin (Warwick) The algebra of string comparison 39 / 185

Alexander Tiskin (Warwick) The algebra of string comparison 40 / 185
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 score(match) = 1 and
  score(mismatch) = 0

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

LCS computation by dynamic programming

\[
\begin{align*}
lcs(a, \emptyset) &= 0 \\
lcs(\emptyset, b) &= 0 \\
lcs(a \alpha, b \beta) &= \begin{cases} 
\max(lcs(a \alpha, b), lcs(a, b \beta)) & \text{if } \alpha \neq \beta \\
lcs(a, b) + 1 & \text{if } \alpha = \beta 
\end{cases}
\end{align*}
\]

\[
lcs(\text{"DEFINE"}, \text{"DESIGN"}) = 4
\]

LCS as a maximum path the LCS grid

\[
\begin{array}{cccccccc}
A & B & E & F & I & S & G & N \\
D & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
E & 0 & 1 & 2 & 2 & 2 & 2 & 2 \\
S & 0 & 1 & 2 & 2 & 2 & 2 & 2 \\
I & 0 & 1 & 2 & 2 & 3 & 3 & 3 \\
G & 0 & 1 & 2 & 2 & 3 & 3 & 3 \\
N & 0 & 1 & 2 & 2 & 3 & 4 & 4 \\
\end{array}
\]

\[
score(\text{"BAABCBCA"}, \text{"BAABCABCABACA"}) = \text{len}(\text{"BAABCBCA"}) = 8
\]

LCS = highest-score path from top-left to bottom-right
Longest common subsequence

LCS

LCS: dynamic programming  [WF: 1974]
Sweep cells in any ≪-compatible order
Active cell update: time $O(1)$
Overall time $O(mn)$

Alexander Tiskin (Warwick)
The algebra of string comparison

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)

Dynamic programming 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
The semi-local LCS problem

Return 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 \(b\))

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

---

The LCS matrix \(H\)

\[
\begin{array}{ccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
2 & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
3 & 2 & 1 & 0 & 1 & 2 & 3 & 4 & 5 \\
4 & 3 & 2 & 1 & 0 & 1 & 2 & 3 & 4 \\
5 & 4 & 3 & 2 & 1 & 0 & 1 & 2 & 3 \\
6 & 5 & 4 & 3 & 2 & 1 & 0 & 1 & 2 \\
7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 1 \\
8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\end{array}
\]

\(a = \text{“BAABCBACA”}\)

\(b = \text{“BAABABBCABA”}\)

\(H(i,j) = lcs(a, b(i:j))\)

\(H(4,11) = 5\)

\(H(i,j) = j - i\) if \(i > j\)

---

Semi-local LCS: output representation and running time

<table>
<thead>
<tr>
<th>size</th>
<th>query time</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(O(n^2))</td>
<td>(O(1))</td>
<td>(O(m^2) = O(n \cdot mn))</td>
</tr>
<tr>
<td>(O(m^{1/2}n))</td>
<td>(O(\log n))</td>
<td>(O(mn))</td>
</tr>
<tr>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(mn))</td>
</tr>
<tr>
<td>(O(n \log n))</td>
<td>(O(\log^2 n))</td>
<td>(O(mn))</td>
</tr>
</tbody>
</table>

...or any 2D orthogonal range counting data structure

---

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

Semi-local LCS: all highest-score boundary-to-boundary paths
The LCS matrix \( H \) and the LCS kernel \( P \)

- \( H(i, j) \): the number of matched characters for \( a \) vs substring \( b(i : j) \)
- \( j - i - H(i, j) \): the number of unmatched characters

Properties of matrix \( j - i - H(i, j) \):
- simple unit-Monge
- therefore, \( = P^\Sigma \), where \( P = -H \oplus \) 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 \( O(\log^2 n) \)

The (reduced) sticky braid in the LCS grid

\( P(i, j) = 1 \) corresponds to strand top \( i \sim \) bottom \( j \)

Also define strands top \( \sim \) right, left \( \sim \) right, left \( \sim \) bottom

Represent implicitly semi-local LCS for each prefix of \( a \) vs \( b \)
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...
Semi-local LCS by iterative combing [T: 2006]

Initialise uncombed sticky braid: mismatch cell = crossing
Iterate over cells in any ≪-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
Micro-block speedup (MBS)

Semi-local LCS by iterative combing with MBS [T: 2007]
Initialise uncombed sticky braid: mismatch cell = crossing
Iterate over in micro-blocks, in any ≪-compatible order
Micro-block size: \( t = O(\frac{\log n}{\log \log n}) \)
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 \( O(\frac{mn(\log \log n)^2}{\log n}) \)

Alignment and edit distance
The LCS problem is a special case of the alignment problem
Scoring scheme: match \( w_+ \), mismatch \( w_0 \), gap \( w_- \)
\[ w_+ \geq 0 \quad 2w_- \leq w_0 < w_+ \quad w_- \leq 0 \]
- LCS score: \( (1, 0, 0) \)
- Levenshtein score: \( (1, \frac{1}{2}, 0) \)

Scoring scheme is
- rational: \( w_+, w_0, w_- \) are rational numbers
- regular: \( (1, w_0, 0) \), i.e. \( w_+ = 1, w_- = 0 \)

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 \( (1, \frac{1}{2}, 0) \)
Alignment and edit distance

Semi-local alignment as maximum paths in the alignment grid

Equivalent to semi-local LCS for blown-up strings

Levenshtein ("BAABCBCA", "CABCABA") = 5.5

Alignment and edit distance

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

Strings a, b blown-up; scheme \((1, \frac{\mu}{\nu}, 0)\) becomes \((\nu, \mu, 0)\)

Increase \(\times \nu^2\) in complexity (can be reduced to \(\nu\))

Alignment and edit distance

The approximate pattern matching problem

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

Assume rational scoring scheme

Approximate pattern matching: running time

\[ O(mn) \]
\[ O\left(\frac{mn}{\log n}\right) \]
\[ \sigma = O(1) \]
\[ O\left(\frac{mn(\log \log n)^2}{\log^2 n}\right) \]

[Sellers: 1980]
[Oviatt, Paterson: 1980]
[Mainke, Farach-Colton: 2008]
Alignment and edit distance
Approximate matching

Approximate pattern matching: the algorithm
Micro-block iterative combing on $a$ vs $b$ (with blow-up), time $O\left(\frac{mn(\log \log n)^2}{\log^2 n}\right)$
The implicit semi-local edit score matrix:
- an anti-Monge matrix
- approximate pattern matching $\sim$ 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]

Further string comparison
Cyclic LCS

The 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]
Further string comparison

Longest square subsequence

The longest square subsequence problem

Longest subsequence of \(a\) having form \(bb\), for some \(b\)

Longest square subsequence: running time

\[
O(m^3) \quad \text{repeated DP}
\]

\[
O(m^2) \quad \text{[Kosowski: 2004]}
\]

\[
O\left(\frac{m^2(\log \log m)^2}{\log^2 m}\right) \quad \text{[T: 2007]}
\]

Open question: generalise to longest cube subsequence, etc.

Further string comparison

Wraparound iterative combing

The periodic string-substring LCS problem

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

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 = \(m\))
Further string comparison
Wraparound iterative combing

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 seaweeds with multiplicities

The tandem LCS problem

Give LCS score for $a$ vs $b = u^k$
We have $n = kp$; may assume $k \leq m$

Tandem LCS: running time

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

Direct application of wraparound iterative combing
Further string comparison

Wraparound iterative combing

The tandem alignment problem

Give the 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

<table>
<thead>
<tr>
<th></th>
<th>global</th>
<th>cyclic</th>
<th>local</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(m^2 p) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( O(mp) )</td>
<td></td>
<td></td>
<td>naive</td>
</tr>
<tr>
<td>( O(mp \log p) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( O(mp) )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cyclic tandem alignment: the algorithm

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 \( O(mp) \)

Sparse string comparison

Semi-local LCS on permutation strings

LCS on permutation strings (LCSP)

Give 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

<table>
<thead>
<tr>
<th>LCSP: running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n \log n)$ implicit in [Erdös, Szekeres: 1935] [Robinson: 1938; Knuth: 1970; Dijkstra: 1980]</td>
</tr>
<tr>
<td>$O(n \log \log n)$ unit-RAM [Chang, Wang: 1992] [Bespamyatnikh, Segal: 2000]</td>
</tr>
</tbody>
</table>

Semi-local LCS on permutation strings

Alexander Tiskin (Warwick)

Sparse string comparison
Semi-local LCS on permutation strings

C F A E D H G B D E H C B A F G

Semi-local LCS

Give 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

Alexander Tiskin (Warwick)

Sparse string comparison
Semi-local LCS on permutation strings

D E H C B A F G

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]

Alexander Tiskin (Warwick)

Sparse string comparison
Semi-local LCS on permutation strings

C F A E D H G B D E H C B A F G
Sparse string comparison
Semi-local LCS on permutation strings

Semi-local LCSP: the algorithm

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

Longest \( k \)-increasing ((\( k-1 \))-modal) subsequence problem
Give length of longest \( k \)-increasing ((\( k-1 \))-modal) subsequence of \( b \)

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

\[
\begin{align*}
O(nk \log n) & \quad \text{(}k-1\text{-modal)} \\
O(nk \log n) & \quad \text{via [Hunt, Szymanski: 1977]} \\
O(n \log^2 n) & \quad \text{[T: 2010]}
\end{align*}
\]

Main idea: \( \text{lcs}(id^k, b) \) (respectively, \( \text{lcs}(id^{id^{k/2}}, b) \))

Sparse string comparison
Maximum clique in a circle graph

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

Longest \( k \)-increasing subsequence: algorithm A
Sparse LCS for \( id^k \) vs \( b \): time \( O(nk \log n) \)

Longest \( k \)-increasing subsequence: algorithm B
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 \( \square \)-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 \( O(n \log^2 n) \)

Algorithm B faster than Algorithm A for \( k \geq \log n \)
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]
$O(n \log n + n \log (n/l))$ [Apostolico+: 1992]
$O(n \log n + n \log^2 d)$ NEW

Max-clique in a circle graph: the algorithm

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

\[ t_1 = A \quad t_2 = t_1B \quad t_3 = t_2t_1 \quad t_4 = t_3t_2 \quad t_5 = t_4t_3 \quad t_6 = t_5t_4 \]

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

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

---

The global subsequence recognition problem

Does pattern $p$ appear in text $t$ as a subsequence?

Global subsequence recognition: running time

<table>
<thead>
<tr>
<th>$p$</th>
<th>$t$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain</td>
<td>plain</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>plain</td>
<td>GC</td>
<td>$O(m\bar{n})$</td>
</tr>
<tr>
<td>GC</td>
<td>GC</td>
<td>NP-hard</td>
</tr>
</tbody>
</table>

[Lifshits: 2005]

---

The local subsequence recognition 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

Local subsequence recognition: running time (+ output)

<table>
<thead>
<tr>
<th>$p$</th>
<th>$t$</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain</td>
<td>plain</td>
<td>$O(mn)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O\left(\frac{mn}{\log m}\right)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(c^m + n)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(m + n\sigma)$</td>
</tr>
<tr>
<td>plain</td>
<td>GC</td>
<td>$O(m^{1.5} \bar{n})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(m^{1.5} \bar{n})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(m \log m \cdot \bar{n})$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(m \cdot \bar{n})$</td>
</tr>
<tr>
<td>GC</td>
<td>GC</td>
<td>NP-hard</td>
</tr>
</tbody>
</table>
Compressed string comparison
Subsequence recognition on GC-strings

Local subsequence recognition (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 $\gg$-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
Subsequence recognition on GC-strings

$\hat{i}_0^+$ $\hat{i}_1^+$ $\hat{i}_2^+$

$b(i : j)$ matching iff box $[i : j]$ not pierced left-to-right

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

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

The edit distance matching problem

Find all substrings of $t$, for which edit distance to $p$ is at most $k$
Compressed string comparison
Edit distance matching

Edit distance matching: running time (+ output)

<table>
<thead>
<tr>
<th>Pattern Type</th>
<th>Algorithm Time Bound</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain (p,t)</td>
<td>$O(mn)$</td>
<td>Sellers: 1980</td>
</tr>
<tr>
<td>GC (p,t)</td>
<td>$O(mk)$</td>
<td>[Landau, Vishkin: 1989]</td>
</tr>
<tr>
<td></td>
<td>$O(m + n + nk^4/m)$</td>
<td>Cole, Hariharan: 2002</td>
</tr>
<tr>
<td>Plain GC</td>
<td>$O(mnk^2)$</td>
<td>Kärkkäinen+: 2003</td>
</tr>
<tr>
<td></td>
<td>$O(mnk + n\log n)$</td>
<td>[LV: 1989] via [Bille+: 2010]</td>
</tr>
<tr>
<td></td>
<td>$O(mn + nk^4 + n\log n)$</td>
<td>[CH: 2002] via [Bille+: 2010]</td>
</tr>
<tr>
<td></td>
<td>$O(m\log m \cdot n)$</td>
<td>[T: 2014]</td>
</tr>
<tr>
<td>GC GC</td>
<td>NP-hard</td>
<td>Lifshits: 2005</td>
</tr>
</tbody>
</table>

(Also many specialised variants for LZ compression)

Compressed string comparison
Edit distance matching

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

Algorithm structure similar to local subsequence recognition

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

Alignment on strings $p$, $t$ of size $m$, $n$ equivalent to LCS on blown-up strings $p$, $t$ of size $m = \nu m$, $n = \nu n$
Beyond semi-locality
Window-substring LCS

A \textit{w-window}: any substring of fixed length \( w \)

Window-substring LCS
Give LCS score for every \( w \)-window of \( a \) vs every substring of \( b \)

Window-substring LCS: running time

\[ O(mn^2w) = mn \text{ runs} \cdot O(nw) \]
\[ O(mnw) = m \text{ runs} \cdot O(nw) \]
\[ O(mn) \quad \text{[Krusche, T: 2010]} \]

Window-substring LCS: the algorithm

Compute LCS kernels for \textit{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 \( O(mn) \)

Fixed total length LCS

Give LCS score for every substring \( a' \) of \( a \) vs every substring \( b' \) of \( b \), where \( \text{length}(a) + \text{length}(b) = w \).

Running time: \( O(mn) \), same as window-substring LCS
Beyond semi-locality
Window-substring LCS

**Fixed total length LCS: the algorithm**
- 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**
- Give 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

**Fragment-substring LCS**
- Given a set of $m$ (possibly overlapping) **fragment** substrings in $a$

**Fragment-substring LCS**
- Give LCS score for every fragment of $a$ vs every substring of $b$

**Fragment-substring LCS: running time**
- $O(m^2n) = m$ runs · $O(mn)$ repeated iterative combing
- $O(mn\log^2 m)$ [T: NEW]
Beyond semi-locality

Spliced alignment

Find 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

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

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

\[
h[l, j] = \max_{k,i} \text{align}(a(k : l), b(i : j))
\]

SW-alignment: running time

\[
O(mn) \quad \text{[Smith, Waterman: 1981]}
\]

SW-alignment maximises alignment score irrespective of length

May not always be the most relevant 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

Threshold \( w \geq 0 \)

Bounded-length SW (BLSW) alignment

For every prefix of \( a \) and every prefix of \( b \), return maximum alignment score between their suffixes \( a', b' \), where \( \text{length}(a') \geq w \)

\[
\begin{align*}
    h[l, j] &= \max_{k, i} \text{align}(a(l:k), b(i:j)) \\
    h_{\text{len} \geq w}[l, j] &= \max_{k, i; l-k \geq w} \text{align}(a(l:k), b(i:j))
\end{align*}
\]

**B(T)LSW-alignment: running time**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running time</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact</td>
<td>( O(mn^2) )</td>
<td>[Arslan, Eğcioğlu: 2004]</td>
</tr>
<tr>
<td>Approx ( \epsilon )-relaxed</td>
<td>( O(mn/\epsilon) )</td>
<td>[Arslan, Eğcioğlu: 2004]</td>
</tr>
<tr>
<td>Exact, rational</td>
<td>( O(mn) )</td>
<td>[T: NEW]</td>
</tr>
</tbody>
</table>

Approximate \( \epsilon \)-relaxed BTLSW-alignment:

- \( \text{length}(a') + \text{length}(b') \geq (1 - \epsilon)w \)
- \( \text{align}(a', b') \) same or higher as for \( \text{length}(a') + \text{length}(b') \geq w \)

BTLSW: similar; first stage replaced by fixed total length alignment
Beyond semi-locality
Bounded-length Smith–Waterman alignment

Normalised SW (NSW)-alignment \[\text{nalign}(a, b) = \frac{\text{align}(a, b)}{\text{length}(a) + \text{length}(b)}\] [Arslan, Eğecioğlu, Pevzner: 2001]

Threshold \(w \geq 0\)

Bounded total length NSW (BTLNSW) alignment

For every prefix of \(a\) and every prefix of \(b\), return max normalised alignment score between their suffixes \(a', b'\), where \(\text{length}(a') + \text{length}(b') \geq w\)

\[h[l, j] = \max_{k, i} \text{align}(a[k:l], b[i:j])\]

\[h_{\text{tlen} \geq w}[l, j] = \max_{k, i, l - k + j - i \geq w} \text{nalign}(a[k:l], b[i:j])\]

---

Parallel string comparison
Transposition networks

Comparison network: a circuit of comparators

A comparator sorts two inputs and outputs them in prescribed order

Classical model for non-branching merging/sorting

Classical comparison networks

<table>
<thead>
<tr>
<th>Type</th>
<th># comparators</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>merging</td>
<td>(O(n \log n))</td>
<td>Batcher: 1968</td>
</tr>
<tr>
<td>sorting</td>
<td>(O(n \log^2 n))</td>
<td>Batcher: 1968</td>
</tr>
<tr>
<td></td>
<td>(O(n \log n))</td>
<td>Ajtai+: 1983</td>
</tr>
</tbody>
</table>

Comparison networks are visualised by wire diagrams

Transposition network: all comparisons are between adjacent wires
Parallel string comparison
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

Parallel string comparison
Parameterised string comparison

Parameterised string comparison

String comparison sensitive e.g. to
- low similarity: small $\lambda = \text{LCS}(a, b)$
- high similarity: small $\kappa = \text{dist}_{\text{LCS}}(a, b) = m + n - 2\lambda$

Can also use alignment score or edit distance

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

Low-similarity comparison: small $\lambda$
- 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 $\kappa$
- 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

<table>
<thead>
<tr>
<th>Low-similarity, after preprocessing in $O(n \log \sigma)$</th>
<th>Low-similarity: $O(n \cdot \lambda)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n \lambda)$</td>
<td>[Hirschberg: 1977]</td>
</tr>
<tr>
<td></td>
<td>[Apostolico, Guerra: 1985]</td>
</tr>
<tr>
<td></td>
<td>[Apostolico+: 1992]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High-similarity, no preprocessing</th>
<th>High-similarity: $O(n \cdot \kappa)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n \cdot \kappa)$</td>
<td>[Ukkonen: 1985]</td>
</tr>
<tr>
<td></td>
<td>[Myers: 1986]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flexible</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(\lambda \cdot \kappa \cdot \log n)$ no preproc</td>
<td>$O(\lambda \cdot \kappa)$ after preproc</td>
</tr>
<tr>
<td></td>
<td>[Myers: 1986; Wu+: 1990]</td>
</tr>
<tr>
<td></td>
<td>[Rick: 1995]</td>
</tr>
</tbody>
</table>

### Trace 0s through network in contiguous blocks and gaps

### Dynamic string comparison

#### The dynamic LCS problem

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

Both input strings are **streams**, updated on-line:

- prepending/appendign characters at left or right
- deleting characters at left or right

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

Goal: **linear time** per update

- $O(n)$ per update of $a$ ($n = |b|$)
- $O(m)$ per update of $b$ ($m = |a|$)

#### Dynamic LCS in linear time: update models

- prep+
- prep
- prep+del
- app+del
- app
- app+del

#### Dynamic semi-local LCS in linear time: update models

- prep+del
- app+del
- [T: NEW]
Parallel string comparison

Bit-parallel LCS

Bit-parallel computation

Computation with Boolean logic and arithmetic on bit vectors of size \( v \)

Bit-parallel LCS: running time

\[ O\left( \frac{mn}{v} \right) \] [Allison, Dix: 1986; Myers: 1999; Crochemore+: 2001]

Iterating through LCS grid in vertical/horizontal bit vectors

Combing two sticky braid strands in every cell

Vector cells not independent: dependencies within iteration resolved by integer addition (carry propagation)

Bit-parallel LCS

Binary transposition network; cell inputs \( s, c \); outputs \( s', c' \)

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

\[
\begin{array}{c|cccccccc}
\mu & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
s & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\mu & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
s' & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\
c' & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Compare to \( 2c + s \leftarrow s + (s \land \mu) + c \)

\[
\begin{array}{c|cccccccc}
\mu & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
s & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\mu & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
s' & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
c' & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

High-similarity bit-parallel LCS

\[ \kappa = \text{dist}_{LCS}(a, b) \] Assume \( \kappa \) odd, \( m = n \)

Waterfall algorithm within diagonal band of width \( \kappa + 1 \): time \( O(n\kappa/w) \)

Band waterfall supported from below by separator matches
High-similarity bit-parallel multi-string LCS: $a$ vs $b_0, \ldots, b_{r-1}$

$$\kappa_i = \text{dist}_{\text{LCS}}(a, b_i) \leq \kappa \quad 0 \leq i < r$$

Waterfalls within $r$ diagonal bands of width $\kappa + 1$: time $O(nr\kappa/w)$

Each band’s waterfall supported from below by separator matches

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-parallel string comparison

Vector-parallel semi-local LCS by 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-parallel string comparison

Vector-parallel implementation of alignment plots [Krusche: 2010]
Based on $O(mnw)$ repeated iterative combing approach, using some ideas from the optimal $O(mn)$ algorithm: resulting time $O(mnw^{0.5})$
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-parallel string comparison

Vector-parallel implementation of alignment plots (contd.)
Used by Warwick biologists to study conservation in plant non-coding DNA
More sensitive than BLAST and other heuristic (seed-and-extend) methods
Found hundreds of conserved non-coding regions between
- papaya (Carica papaya)
- poplar (Populus trichocarpa)
- thale cress (Arabidopsis thaliana)
- grape (Vitis vinifera)
Evolutionary analysis of regulatory sequences (EARS) in plants

Emma Picot¹,², Peter Krusche³, Alexander Tiskin⁴, Isabelle Carré² and Sascha Ott⁴,⁵

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 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 −0.3 for a gap. Thus, for a 60-bp window, the highest score possible is 60.

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-


Web service: http://wsbc.warwick.ac.uk/ears
Warwick press release (with video): http://warwick.ac.uk/press/discovery_of_100

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

Nathaniel J. Davies¹, Peter Krusche², Eran Tauber³ and Sascha Ott²

A 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 seaweed 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.


Web service: http://wsbc.warwick.ac.uk/ears
Warwick press release (with video): http://warwick.ac.uk/press/discovery_of_100

Parallel string comparison
Vector-parallel string comparison

Intel’s AVX-512 instruction set: 32× 512-bit registers
Each register accessible as vector of 64× 8-bit ints; 32× 16-bit ints; . . .
Supported by recent CPU microarchitectures
- Skylake-X (2017): expensive high-end desktops
- Cannon Lake (2018): mainstream PCs, not yet volume-produced
- Cascade Lake (2019): servers
  - . . .
None widely available yet
Software can be developed using Intel Software Development Emulator

Alexander Tiskin (Warwick)  The algebra of string comparison  165 / 185

Experimental AVX-512 implementation of semi-local LCS [Orekhov, T: 2019]

Transposition network; cell inputs s, c; outputs s’, c’
Combing logic: (c’, s’) ← if ¬µ then sort(s, c) else (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

Experimental AVX-512 implementation of semi-local LCS (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

Experimental AVX-512 implementation of semi-local LCS (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:
- ×2 to prevent vector swapping between even/odd iterations
- ×16 to keep all data in registers
Parallel string comparison
Bulk-synchronous parallelism

Bulk-Synchronous Parallel (BSP) computer

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)

For the whole BSP computer in one superstep \( sstep \):

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

For the whole BSP computation with \( sync \) supersteps:

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

The input/output data are stored in the external memory; the cost of input/output is included in \( \text{comm} \)

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

- \( \text{comp} = O(n^3/p) \)
- \( \text{comm} = O(n^2/p^{1/2}) \)
- \( \text{sync} = O(p^{1/2}) \)
Parallel string comparison

 Bulk-synchronous parallelism

BSP software: industrial projects

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

BSP software: research projects

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

The ordered 2D grid

grid_2(n)

nodes arranged in an n \times n grid
edges directed top-to-bottom, left-to-right
\leq 2n inputs (to left/top borders)
\leq 2n outputs (from right/bottom borders)
size n^2 \text{ 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 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 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

\[\text{comp: } (2p - 1) \cdot O((n/p)^2) = O(p \cdot n^2/p^2) = O(n^2/p)\]
\[\text{comm: } (2p - 1) \cdot O(n/p) = O(n)\]
\[n \geq p\]

\[\text{comp } O(n^2/p) \]
\[\text{comm } O(n) \]
\[\text{sync } O(p)\]
Parallel LCS

The 2D grid algorithm solves the LCS problem (and many others) by dynamic programming

\[ \text{comp} = O(n^2/p) \quad \text{comm} = O(n) \quad \text{sync} = O(p) \]

\text{comm} \text{ is not scalable (i.e. does not decrease with increasing } p) \quad :-(\)

Can scalable \text{comm} be achieved for the LCS problem? Yes

---

Matrix sticky multiplication: parallel Steady Ant

\[ R^\Sigma(i, k) = \min_j (P^\Sigma(i, j) + Q^\Sigma(j, k)) \]

Divide-and-conquer on the range of \( j \): two subproblems on \( n/2 \)-matrices

Conquer: trace Steady Ant’s trail from bottom-left to top-right of \( R \)
- partition \( n \times n \) index space into a regular \( p \times p \) grid of blocks
- sample matrix \( R \) in block corners: total \( (p + 1)^2 \) samples
- Steady Ant’s trail passes through \( \leq 2p - 1 \) blocks
- trace it within each block sequentially: \( \text{comp}, \text{comm} = O(n/p) \)

\( \log p \) recursion levels, “flattened” to \( O(1) \) supersteps

\[ \text{comp} = O\left(\frac{n \log n}{p}\right) \quad \text{comm} = O\left(\frac{n \log p}{p}\right) \quad \text{sync} = O(1) \]
Parallel LCSP (cont.)

Parallel semi-local LCSP: running time, comm, sync

<table>
<thead>
<tr>
<th></th>
<th>comp</th>
<th>comm</th>
<th>sync</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O\left(\frac{n \log^2 n}{p}\right)$</td>
<td>$O\left(\frac{n \log^2 p}{p}\right)$</td>
<td>$O\left(\log^2 p\right)$</td>
</tr>
<tr>
<td></td>
<td>$O\left(\log^2 p\right)$</td>
<td>$O\left(\log p\right)$</td>
<td>$O\left(1\right)$</td>
</tr>
</tbody>
</table>

Open problems
- $\text{comp } O\left(\frac{n \log n}{p}\right)$ for LCSP?
- $\text{comp } O\left(\frac{n \log^2 n}{p}\right)$, $\text{comm } O\left(\frac{n \log p}{p}\right)$ or $\text{sync } O\left(1\right)$ for semi-local LCSP?

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

Iterative combing and micro-block speedup:
- a simple algorithm for semi-local LCS
- semi-local LCS in time $O(mn)$, and even slightly $o(mn)$

Sparse string comparison:
- fast LCS on permutation strings
- fast max-clique in a circle graph

Conclusions and open problems (2)

Compressed string comparison:
- LCS on GC-strings
- approximate matching on plain pattern against GC-text

Parallel string comparison:
- comm- and sync- efficient parallel LCS

Transposition networks:
- simple interpretation of parameterised and bit-parallel LCS
- dynamic LCS

Beyond semi-locality:
- window alignment and sparse spliced alignment
Conclusions and open problems (3)

Open problems (theory):
- real-weighted alignment?
- max-clique in circle graph: $O(n \log^2 n) \rightsquigarrow O(n \log n)$?
- compressed LCS: other compression models?
- parallel LCS: comm per processor $O\left(\frac{n \log p}{p^{1/2}}\right) \rightsquigarrow O\left(\frac{n}{p^{1/2}}\right)$?
- lower bounds?

Open problems (biological applications):
- real-weighted alignment?
- good pre-filtering (e.g. $q$-grams)?
- application for multiple alignment? (NB: NP-hard)

Future work:
- real-weighted alignment?
- good pre-filtering (e.g. $q$-grams)?
- application for multiple alignment? (NB: NP-hard)