Efficient High-Similarity String Comparison: The Waterfall Algorithm

Alexander Tiskin

Department of Computer Science
University of Warwick
http://go.warwick.ac.uk/alextiskin

1. Semi-local string comparison

2. The transposition network method

Consider strings (= sequences) over an alphabet of size σ

Distinguish contiguous substrings and not necessarily contiguous subsequences

Special cases of substring: prefix, suffix

Notation: strings a, b of length m, n respectively

Assume where necessary: $m \leq n$; m, n reasonably close

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" (unlike "lossy" BLAST)

The LCS problem

Give the LCS score for a vs b

LCS: running time

$O(mn)$		[Wagner, Fischer: 1974]
$O\left(\frac{mn}{\log^2 n}\right)$	$\sigma = O(1)$	[Masek, Paterson: 1980]
		[Crochemore+: 2003]
$O\left(\frac{mn(\log\log n)^2}{\log^2 n}\right)$		[Paterson, Dančík: 1994]
		[Bille, Farach-Colton: 2008]

Running time varies depending on the RAM model version

We assume word-RAM with word size $\log n$ (where it matters)

LCS computation by dynamic programming

$lcs(a, \text{""}) = 0$
$lcs(\text{""}, b) = 0$

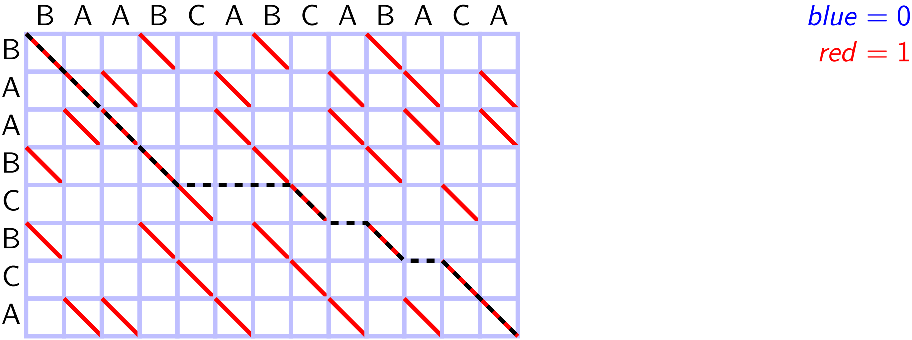
$$lcs(a\alpha, b\beta) = \begin{cases} \max(lcs(a\alpha, b), lcs(a, b\beta)) & \text{if } \alpha \neq \beta \\ lcs(a, b) + 1 & \text{if } \alpha = \beta \end{cases}$$

	*	d	e	f	i	n	e
*	0	0	0	0	0	0	0
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

$lcs(\text{"define"}, \text{"design"}) = 4$

$LCS(a, b)$ can be "traced back" through the table at no extra asymptotic cost

LCS on the alignment graph (directed, acyclic)



score("BAABCBCA", "BAABCABCABACA") = *len*("BAABCBCA") = 8

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

LCS: dynamic programming [WF: 1974]

Sweep cells in any $\ll$-compatible order

Cell update: time $O(1)$

Overall time $O(mn)$

LCS: micro-block dynamic programming [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

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

L. Carroll, *Alice in Wonderland*

Standard approach by dynamic programming

Sometimes dynamic programming can be run from both ends for extra flexibility

Is there a better, fully flexible alternative (e.g. for comparing compressed strings, comparing strings dynamically or in parallel, etc.)?

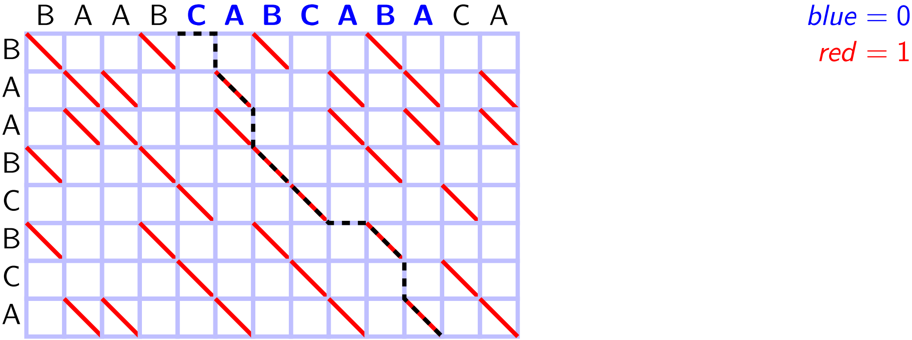
The semi-local LCS problem

Give the (implicit) matrix of $O\big((m+n)^2\big)$ LCS scores:

- string-substring LCS: string a vs every substring of b
- prefix-suffix LCS: every prefix of a vs every suffix of b
- suffix-prefix LCS: every suffix of a vs every prefix of b
- substring-string LCS: every substring of a vs string b

Cf.: dynamic programming gives prefix-prefix LCS

Semi-local LCS on the alignment graph



score("BAABCBCA", "**CABCABA**") = *len*("ABCBA") = 5

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

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

The score 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"

$H(i, j) = score(a, b\langle i : j \rangle)$

$H(4, 11) = 5$

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

Semi-local LCS: output representation and running time

size	query time		
$O(n^2)$	$O(1)$		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		
$O(mn^2)$		naive
$O(mn)$	string-substring	[Schmidt: 1998; Alves+: 2005]
$O(mn)$		[T: 2006]
$O\left(\frac{mn}{\log^{0.5} n}\right)$		[T: 2006]
$O\left(\frac{mn(\log \log n)^2}{\log^2 n}\right)$		[T: 2007]

Semi-local string comparison

Score matrices and seaweed matrices

The score matrix H and the seaweed matrix 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 seaweed matrix, giving an implicit representation of H

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

The score matrix H and the seaweed matrix 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"

$H(i,j) = score(a, b\langle i : j\rangle)$

$H(4,11) = 5$

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

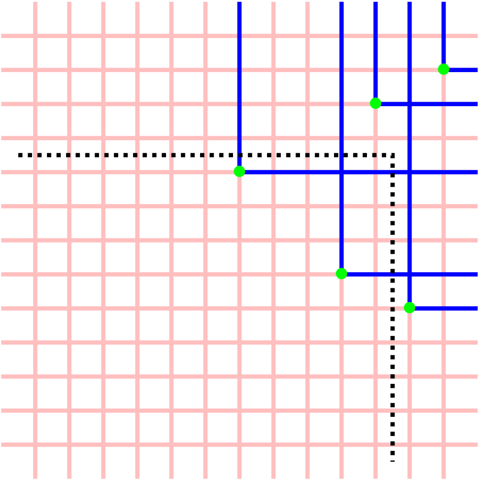
blue: difference in H is 0

red: difference in H is 1

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

$H(i,j) = j - i - P^{\Sigma}(i,j)$

The score matrix H and the seaweed matrix P

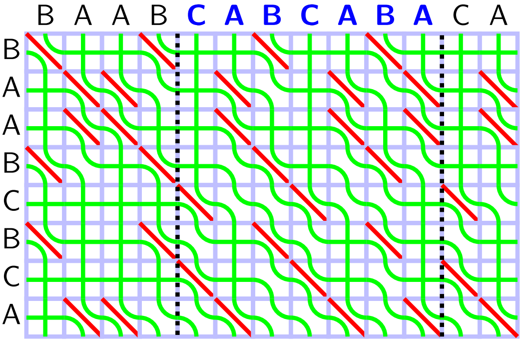


$a =$ "BAABCBCA"

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

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

The seaweed braid in the alignment graph



$a =$ "BAABCBCA"

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

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

$P(i, j) = 1$ corresponds to seaweed *top* $i \rightsquigarrow$ *bottom* j

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

Gives bijection between top-left and bottom-right graph boundaries

Seaweed braid: a highly symmetric object (element of the 0-Hecke monoid of the symmetric group)

Can be built recursively by assembling subbraids from separate parts

Highly flexible: local alignment, compression, parallel computation...

The LCS problem is a special case of the weighted alignment score problem with weighted matches (w_M), mismatches (w_X) and gaps (w_G)

- LCS score: $w_M = 1$, $w_X = w_G = 0$
- Levenshtein score: $w_M = 2$, $w_X = 1$, $w_G = 0$

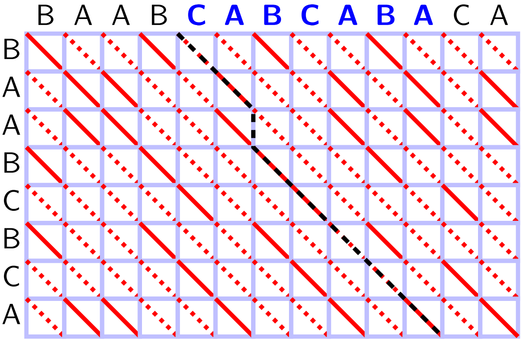
Alignment score is rational, if w_M, w_X, w_G are rational numbers

Equivalent to LCS score on blown-up strings

Edit distance: minimum cost to transform a into b by weighted character edits (insertion, deletion, substitution)

Corresponds to weighted alignment score with $w_M = 0$, insertion/deletion weight $-w_G$, substitution weight $-w_X$

Weighted alignment graph



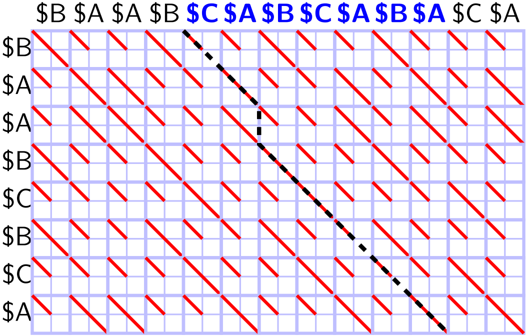
blue $= 0$
red (solid) $= 2$
red (dotted) $= 1$

Levenshtein *score*("BAABCBCA", "**CABCABA**") $= 11$

Semi-local string comparison

Weighted alignment

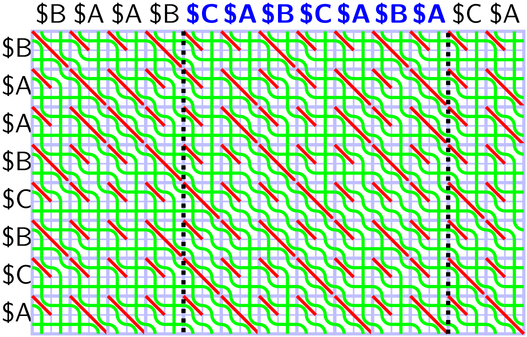
Alignment graph for blown-up strings



blue = 0
red = 0.5 or 1

Levenshtein *score*("BAABCBCA", "**CABCABA**") $= 2 \cdot 5.5$

Rational-weighted semi-local alignment reduced to semi-local LCS



Let $w_M = 1$, $w_X = \frac{\mu}{\nu}$, $w_G = 0$

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

Comparison network: a circuit of comparators

A comparator sorts two inputs and outputs them in prescribed order

Comparison networks traditionally used for non-branching merging/sorting

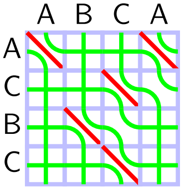
Classical comparison networks

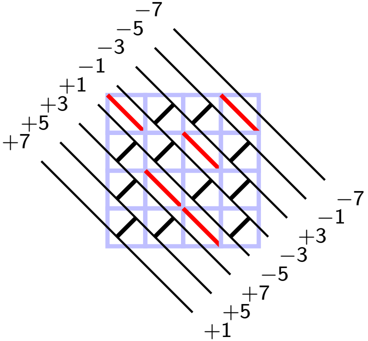
	# comparators	
merging	$O(n \log n)$	[Batcher: 1968]
sorting	$O(n \log^2 n)$	[Batcher: 1968]
	$O(n \log n)$	[Ajtai+: 1983]

Comparison networks are visualised by wire diagrams

Transposition network: all comparisons are between adjacent wires

Seaweed combing as a transposition network





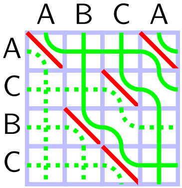
Character mismatches correspond to comparators

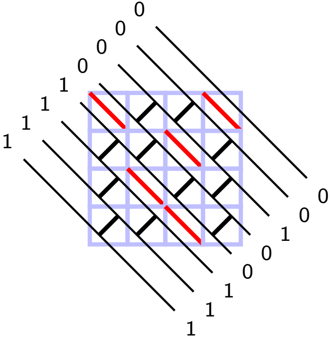
Inputs anti-sorted (sorted in reverse); each value traces a seaweed

The transposition network method

Transposition networks

Global LCS: transposition network with binary input





Inputs still anti-sorted, but may not be distinct

Comparison between equal values is indeterminate

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 weighted alignment score or edit distance

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

The transposition network method

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

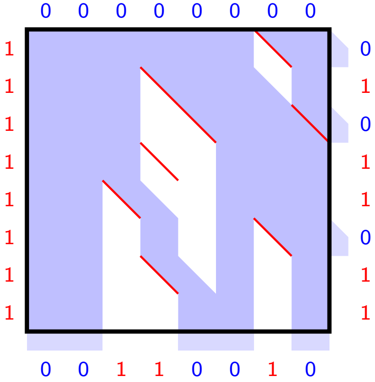
Low-similarity, after preprocessing in $O(n \log \sigma)$		
$O(n\lambda)$		[Hirschberg: 1977]
		[Apostolico, Guerra: 1985]
		[Apostolico+: 1992]

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

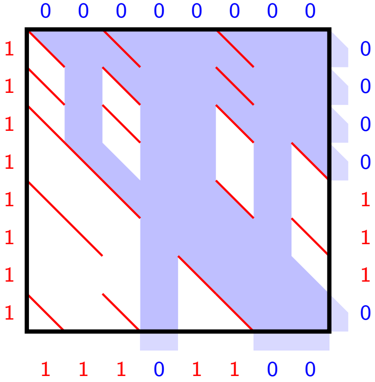
Flexible		
$O(\lambda \cdot \kappa \cdot \log n)$	no preproc	[Myers: 1986; Wu+: 1990]
$O(\lambda \cdot \kappa)$	after preproc	[Rick: 1995]

Parameterised string comparison: the waterfall algorithm

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



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



Trace 0s through network in contiguous blocks and gaps

The dynamic LCS problem

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

Both input strings are streams, updated on-line:

- appending 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

left	right		
–	app+del		standard DP [Wagner, Fischer: 1974]
app	app	a fixed	[Landau+: 1998], [Kim, Park: 2004]
app	app		[Ishida+: 2005]
app+del	app+del		[T: NEW]

Main idea:

- for append only, maintain seaweed matrix $P_{a,b}$
- for append+delete, maintain partial seaweed layout by tracing a transposition network

Bit-parallel string comparison

String comparison using standard instructions on words of size w

Bit-parallel string comparison: running time

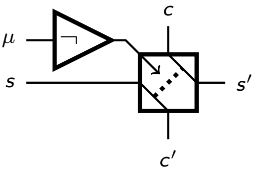
$O(mn/w)$ [Allison, Dix: 1986; Myers: 1999; Crochemore+: 2001]

The transposition network method

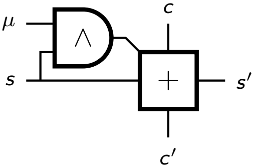
Bit-parallel string comparison

Bit-parallel string comparison: binary transposition network

In every cell: input bits s, c; output bits s', c'; match/mismatch flag μ



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



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

$2c + s \leftarrow (s + (s \wedge \mu) + c) \vee (s \wedge \neg\mu)$

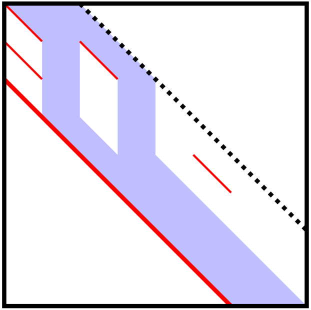
$S \leftarrow (S + (S \wedge M)) \vee (S \wedge \neg M)$, where S, M are words of bits s, μ

The transposition network method

Bit-parallel string comparison

High-similarity bit-parallel string comparison

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



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

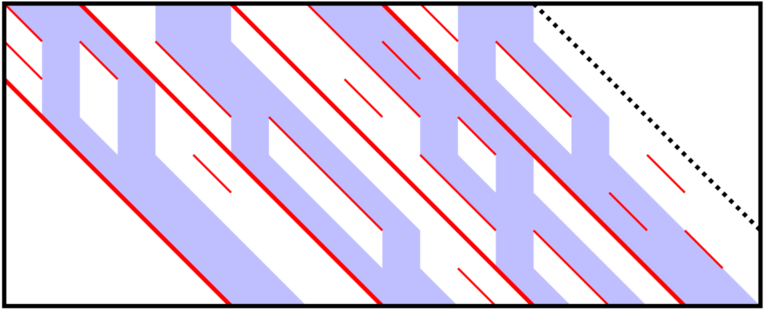
Band waterfall supported from below by separator matches

The transposition network method

Bit-parallel string comparison

High-similarity bit-parallel multi-string comparison: 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 $O(nr\kappa/w)$

Each band's waterfall supported from below by separator matches