

# Approximate string matching in subquadratic time

Alexander Tiskin

Department of Computer Science  
University of Warwick

<http://www.dcs.warwick.ac.uk/~tiskin>

(includes an extended version of this presentation)

- 1 Semi-local string comparison
- 2 Algorithmic techniques
- 3 The seaweed algorithm
- 4 Conclusions and future work

1 Semi-local string comparison

2 Algorithmic techniques

3 The seaweed algorithm

4 Conclusions and future work

# Semi-local string comparison

*String matching*: finding an *exact* pattern in a string

*String comparison*: finding *similar* patterns in two strings

(Also known as *approximate string matching*, no relation to approximation algorithms!)

Applications: computational biology, image recognition, ...

# Semi-local string comparison

*String matching*: finding an *exact* pattern in a string

*String comparison*: finding *similar* patterns in two strings

(Also known as *approximate string matching*, no relation to approximation algorithms!)

Applications: computational biology, image recognition, ...

Standard types of string comparison:

- *global*: whole string against whole string
- *local*: substrings against substrings

Main interest of this work:

- *semi-local*: whole string against substrings; prefixes against suffixes

# Semi-local string comparison

Consider *strings* (= *sequences*) over an alphabet of size  $\sigma$

Distinguish contiguous *substrings* and not necessarily contiguous *subsequences*

Special cases of substring: *prefix*, *suffix*

Standard notation: strings  $a$ ,  $b$  of length  $m$ ,  $n$  respectively

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

# Semi-local string comparison

Consider *strings* (= *sequences*) over an alphabet of size  $\sigma$

Distinguish contiguous *substrings* and not necessarily contiguous *subsequences*

Special cases of substring: *prefix*, *suffix*

Standard notation: strings  $a$ ,  $b$  of length  $m$ ,  $n$  respectively

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

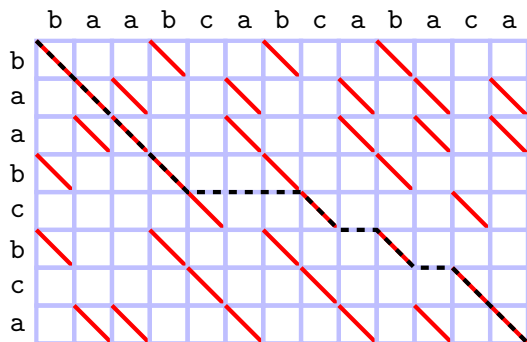
The *longest common subsequence (LCS) score*:

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

The *LCS problem*: determine the LCS score for  $a$  against  $b$

# Semi-local string comparison

The *alignment graph* (directed, acyclic)



blue = 0

red = 1

$LCS("baabcbca", "baabcabcabaca") = "baabcbca"$

LCS = highest-score corner-to-corner path

# Semi-local string comparison

LCS problem: computation time, assuming  $\sigma = O(1)$

---

$O(mn)$	[Wagner, Fischer: 1974]
$O\left(\frac{mn}{\log n}\right)$	[Masek, Paterson: 1980], [Crochemore+: 2003]

---

# Semi-local string comparison

LCS problem: computation time, assuming  $\sigma = O(1)$

---

$O(mn)$	[Wagner, Fischer: 1974]
$O\left(\frac{mn}{\log n}\right)$	[Masek, Paterson: 1980], [Crochemore+: 2003]

---

Standard approach: dynamic programming

Speedup by exhaustive precomputation of small blocks

Block size:  $t = O(\log n)$

Block interface:  $O(t)$  values, each of size  $O(1)$

Total work  $O\left(\frac{mn}{\log n}\right)$ , log-cost RAM

# Semi-local string comparison

The *semi-local LCS* problem:

- *string-substring LCS*: string  $a$  against every substring of  $b$
- *prefix-suffix LCS*: every prefix of  $a$  against every suffix of  $b$
- symmetrically, *substring-string* and *suffix-prefix LCS*

# Semi-local string comparison

The *semi-local LCS problem*:

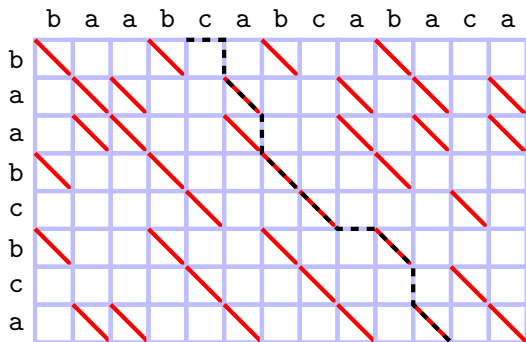
- *string-substring LCS*: string  $a$  against every substring of  $b$
- *prefix-suffix LCS*: every prefix of  $a$  against every suffix of  $b$
- symmetrically, *substring-string* and *suffix-prefix LCS*

Output: the *highest-score matrix* of  $O(n^2)$  LCS scores, allowed to be represented implicitly

Cf.: standard dynamic programming gives *prefix-prefix LCS*

# Semi-local string comparison

The alignment graph



*blue* = 0

*red* = 1

$LCS("baabcbca", "...cabca...") = "abcba"$

Semi-local LCS = all highest-score border-to-border paths  
(string-substring = top-to-bottom, etc.)

# Semi-local string comparison

A: the highest-score matrix for semi-local LCS of  $a$  and  $b$

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

$a = \text{"baabcbca"}$

$b = \text{"baabcabcbabaca"}$

$A(0, 13) = LCS(a, b) = 8$

$b' = \text{"...cabcbaba..."}$

$A(4, 11) = LCS(a, b') = 5$

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

# Semi-local string comparison

Semi-local LCS problem: output representation

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], using [Bentley: 1980]

# Semi-local string comparison

Semi-local LCS problem: output representation

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], using [Bentley: 1980]

Semi-local LCS problem: computation time, any  $\sigma$

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

Based on the *seaweed algorithm*, log-cost RAM

# Semi-local string comparison

The LCS problem is a special case of the *edit distance problem*: minimum cost to transform  $a$  into  $b$  by (weighted) character edits

Edit types: insertion, deletion, substitution

- *Levenshtein distance*:  $w_{in} = w_{del} = w_{sub} = 1$
- *LCS distance*:  $w_{in} = w_{del} = 1$ ,  $w_{sub} \geq 2$  (i.e. no substitutions)

An edit distance is *rational*, if  $\frac{w_{in} + w_{del}}{w_{sub}}$  is a rational number

# Semi-local string comparison

The LCS problem is a special case of the *edit distance problem*: minimum cost to transform  $a$  into  $b$  by (weighted) character edits

Edit types: insertion, deletion, substitution

- *Levenshtein distance*:  $w_{in} = w_{del} = w_{sub} = 1$
- *LCS distance*:  $w_{in} = w_{del} = 1$ ,  $w_{sub} \geq 2$  (i.e. no substitutions)

An edit distance is *rational*, if  $\frac{w_{in} + w_{del}}{w_{sub}}$  is a rational number

The *semi-local edit distance problem*: string-substring, prefix-suffix, substring-string, suffix-prefix edit distances

The *edit distance score*: alignment score, where  $score(match) = w_{in} + w_{del}$  and  $score(mismatch) = w_{in} + w_{del} - w_{sub}$

Edit distance:  $m \cdot w_{del} + n \cdot w_{in} - score(a, b)$

Hence, rational semi-local edit distance reduces to semi-local LCS



1 Semi-local string comparison

2 **Algorithmic techniques**

3 The seaweed algorithm

4 Conclusions and future work

# Algorithmic techniques

Integers:  $\dots - 2, -1, 0, 1, 2, \dots$

Odd half-integers:  $\dots - \frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots$

All matrices are conceptually *infinite*, so we will ignore index ranges

A *permutation matrix* is a 0/1 matrix with exactly one nonzero per row and per column

# Algorithmic techniques

Integers:  $\dots - 2, -1, 0, 1, 2, \dots$

Odd half-integers:  $\dots - \frac{5}{2}, -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots$

All matrices are conceptually *infinite*, so we will ignore index ranges

A *permutation matrix* is a 0/1 matrix with exactly one nonzero per row and per column

Denote  $i^- = i - \frac{1}{2}$  and  $i^+ = i + \frac{1}{2}$

Given matrix  $D$ , its *distribution matrix* is  $D^\Sigma(i, j) = \sum_{i' > i, j' < j} D(i', j')$

We have  $D(i, j) = D^\Sigma(i^-, j^+) - D^\Sigma(i^-, j^-) - D^\Sigma(i^+, j^+) + D^\Sigma(i^+, j^-)$

$D$  is indexed by integers, and  $D^\Sigma$  by odd half-integers

# Algorithmic techniques

$A$ : the highest-score matrix for semi-local LCS of  $a$  and  $b$

$A(i, j)$ : the number of matched characters in substring of  $b$  against  $a$

$Q(i, j) = j - i - A(i, j)$ : the number of *unmatched* characters

Properties of matrix  $Q$ :

- difference between adjacent elements of  $Q$  is 0 or 1
- $Q$  is *Monge*:  $Q(i^-, j^+) + Q(i^+, j^-) \geq Q(i^-, j^-) + Q(i^+, j^+)$

These imply our key property:

- $Q = P^\Sigma$  for some permutation matrix  $P$

# Algorithmic techniques

A: the highest-score matrix for semi-local LCS of  $a$  and  $b$

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

$a = \text{"baabcbca"}$

$b = \text{"baabcabcbabaca"}$

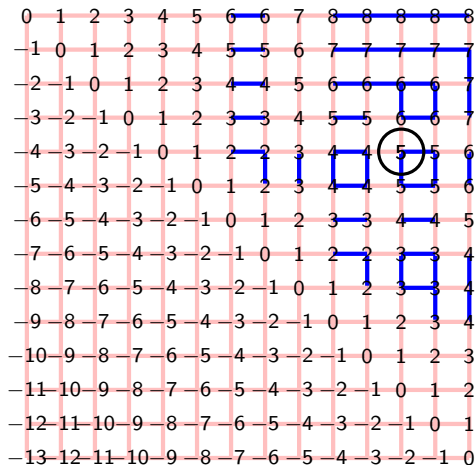
$b' = \text{"...cabcbaba..."}$

$A(4, 11) = LCS(a, b') = 5$

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

# Algorithmic techniques

A: the highest-score matrix for semi-local LCS of  $a$  and  $b$



$a = \text{"baabcbca"}$

$b = \text{"baabcabcbacaca"}$

$b' = \text{"...cabcbaba..."}$

$A(4, 11) = LCS(a, b') = 5$

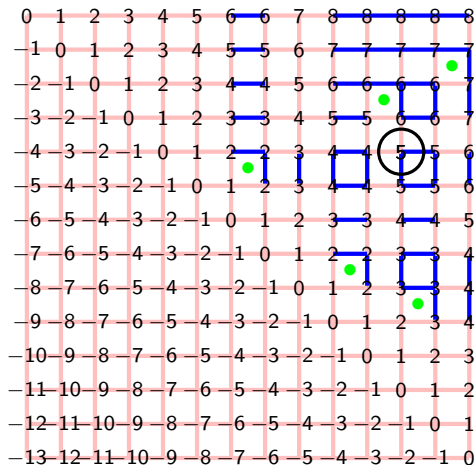
$A(i, j) = j - i$  if  $i > j$

blue: difference 0

red: difference 1

# Algorithmic techniques

A: the highest-score matrix for semi-local LCS of  $a$  and  $b$



$a = \text{"baabcbca"}$

$b = \text{"baabcabcbabaca"}$

$b' = \text{"...cabcbaba..."}$

$A(4, 11) = LCS(a, b') = 5$

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

blue: difference 0

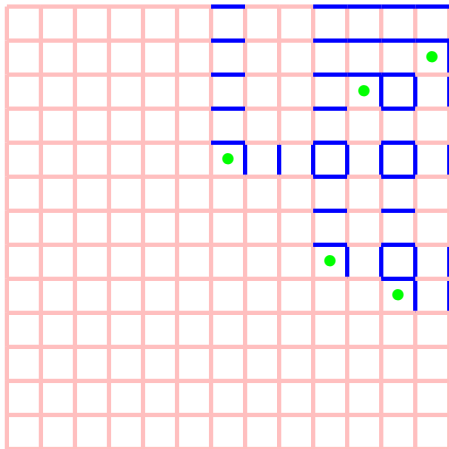
red: difference 1

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

$A(i, j) = j - i - P^\Sigma(i, j)$

# Algorithmic techniques

A: the highest-score matrix for semi-local LCS of  $a$  and  $b$



$a = \text{"baabcbca"}$

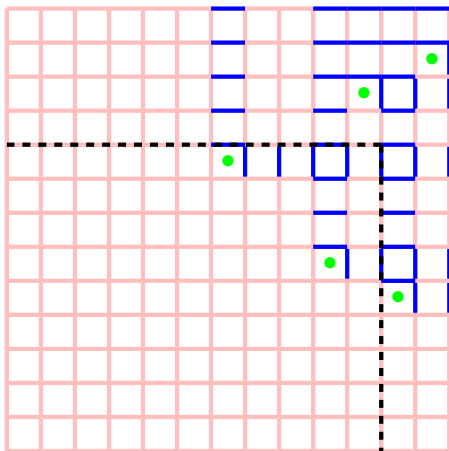
$b = \text{"baabcabcbabaca"}$

$b' = \text{"...cabcbaba..."}$

$A(4, 11) = LCS(a, b') = 5$

# Algorithmic techniques

A: the highest-score matrix for semi-local LCS of  $a$  and  $b$



$a = \text{"baabcbca"}$

$b = \text{"baabcabcbabaca"}$

$b' = \text{"...cabcbaba..."}$

$A(4, 11) = \text{LCS}(a, b') =$

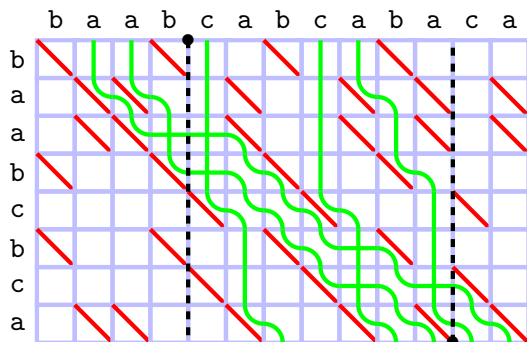
$11 - 4 - P^\Sigma(i, j) =$

$11 - 4 - 2 = 5$

$P$  gives an *implicit*  
*representation* of  $A$

# Algorithmic techniques

The alignment graph and the *seaweeds*



$a = \text{"baabcbca"}$

$b = \text{"baabcabca"}$

$b' = \text{"...cabcaba..."}$

$A(4, 11) = LCS(a, b') =$

$11 - 4 - P^\Sigma(i, j) =$

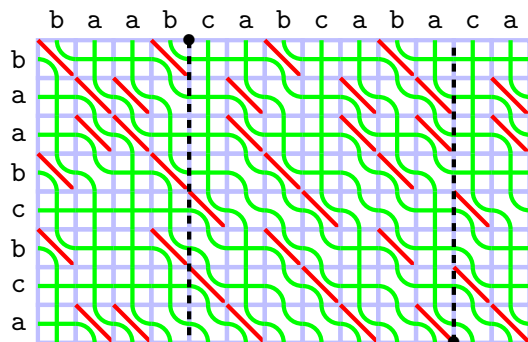
$11 - 4 - 2 = 5$

$P$  gives an *implicit representation* of  $A$

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

# Algorithmic techniques

The alignment graph and the seaweeds



$a = \text{"baabcbca"}$

$b = \text{"baabcabcbabaca"}$

$b' = \text{"...cabcbaba..."}$

$A(4, 11) = LCS(a, b') =$

$11 - 4 - P^\Sigma(i, j) =$

$11 - 4 - 2 = 5$

$P$  gives an *implicit representation* of  $A$

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

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

Gives complete border-to-border graph-theoretic matching

# Algorithmic techniques

## Gaudí's seaweeds (Casa Milà, Barcelona)



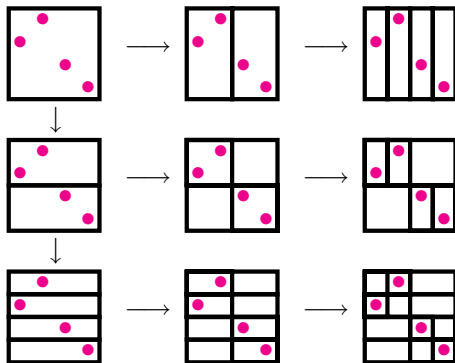
# Algorithmic techniques

Obtaining  $P^\Sigma$  (and  $A$ ) from  $P$ : *dominance counting*

The *range tree*:

[Bentley, 1980]

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



# Algorithmic techniques

Every node of the range tree represents a *canonical range* (rectangular region), and stores its point count

Overall,  $\leq n \log n$  canonical ranges are non-empty

Every range can be decomposed into  $\leq \log^2 n$  canonical ranges to answer a dominance counting query

# Algorithmic techniques

Every node of the range tree represents a *canonical range* (rectangular region), and stores its point count

Overall,  $\leq n \log n$  canonical ranges are non-empty

Every range can be decomposed into  $\leq \log^2 n$  canonical ranges to answer a dominance counting query

Dominance counting data structures

size	query time	
$O(n \log n)$	$O(\log^2 n)$	[Bentley: 1980]
$O(n)$	$O\left(\frac{\log n}{\log \log n}\right)$	[JáJá+: 2004]

# Algorithmic techniques

$A \odot B = C$ :  $(\min, +)$  matrix multiplication

$$C(i, k) = \min_j (A(i, j) + B(j, k))$$

matrix type	time	
general	$O(n^3)$	standard
Monge	$O(n^2)$	based on [Aggarwal+: 1987]
permutation <sup><math>\Sigma</math></sup>	$O(n^{1.5})$	[T: 2006]

# Algorithmic techniques

Let  $P_A^\Sigma \odot P_B^\Sigma = P_C^\Sigma$

Given  $P_A, P_B$ , need to compute  $P_C$

Divide-and-conquer on  $P_C$ . Obtain counts of nonzeros in blocks. If (and only if!) a block has at least one nonzero, recurse into half-sized subblocks.

Crucial observation: for a block of size  $r$ , only need to keep  $O(r)$  nonzeros from  $P_A, P_B$ , and to perform  $O(r)$  work

In the divide-and-conquer tree

- root: one block of size  $n$ , work  $O(n)$
- middle level: at most  $n$  blocks of size  $n^{0.5}$ , work  $O(n \cdot n^{0.5}) = O(n^{1.5})$
- leaves: at most  $n$  blocks of size 1, work  $O(n)$

Middle level dominates, total work  $O(n^{1.5})$

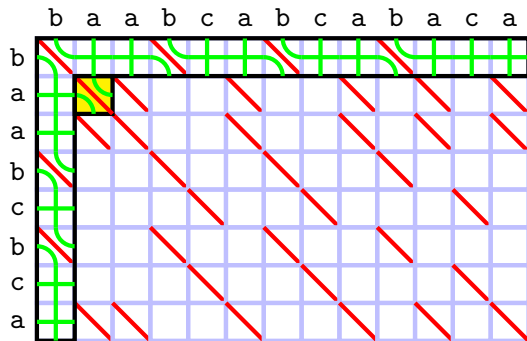


- 1 Semi-local string comparison
- 2 Algorithmic techniques
- 3 The seaweed algorithm**
- 4 Conclusions and future work

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

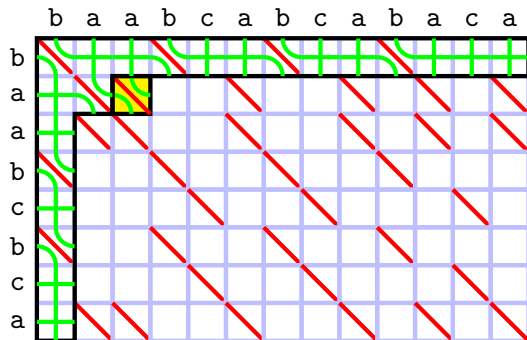
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

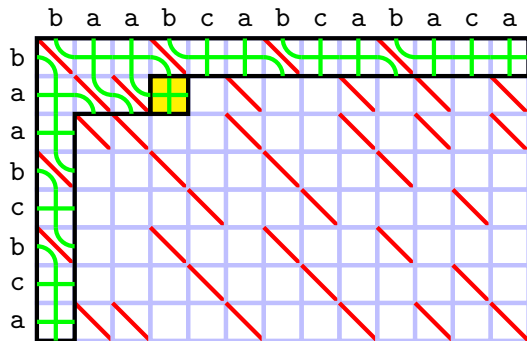
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

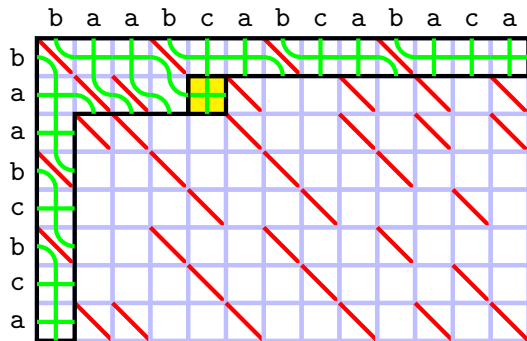
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

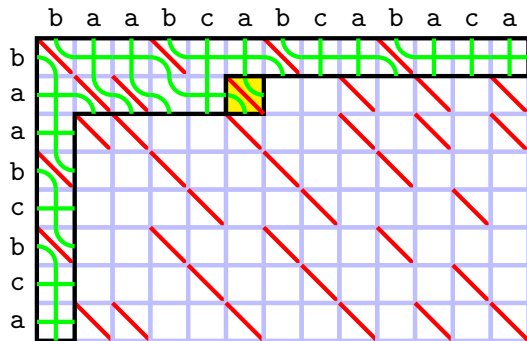
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

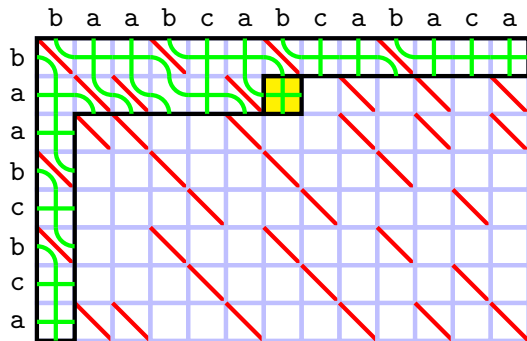
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

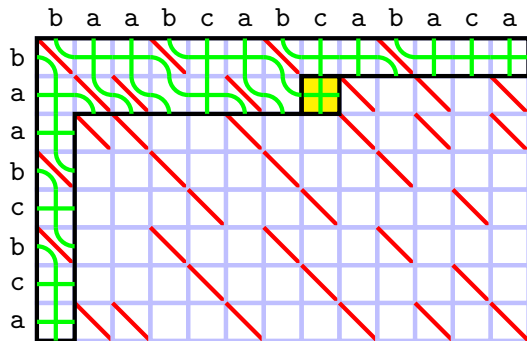
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

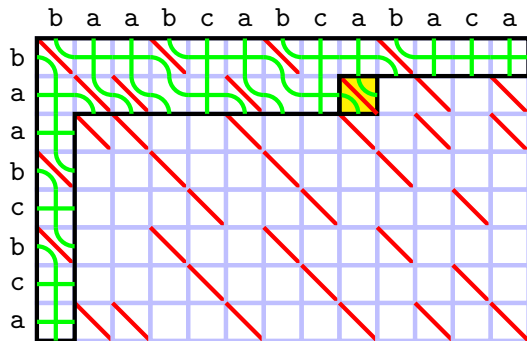
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

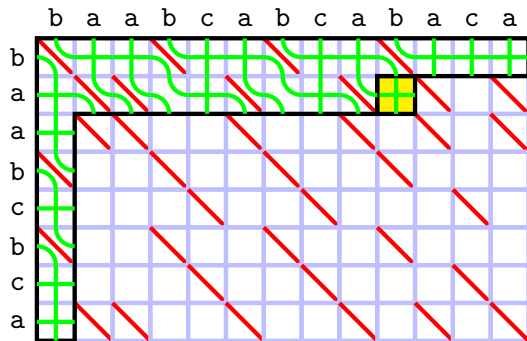
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

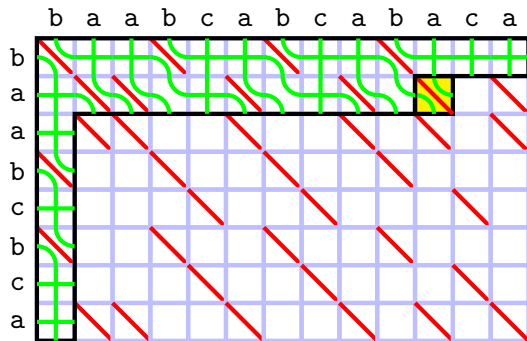
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

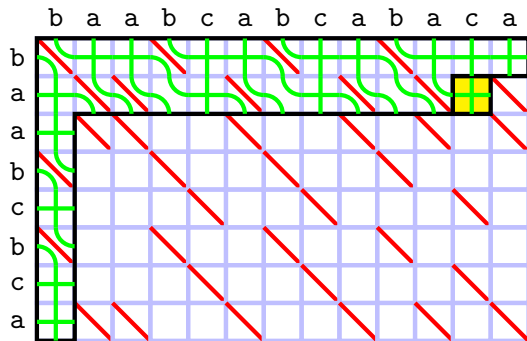
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

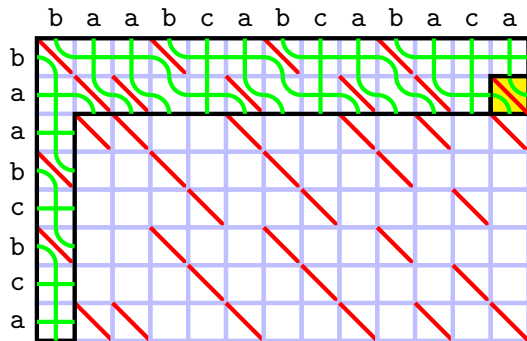
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

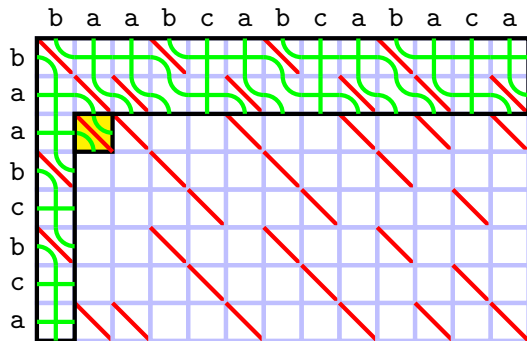
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

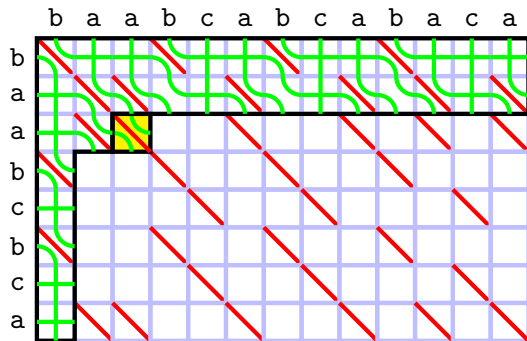
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

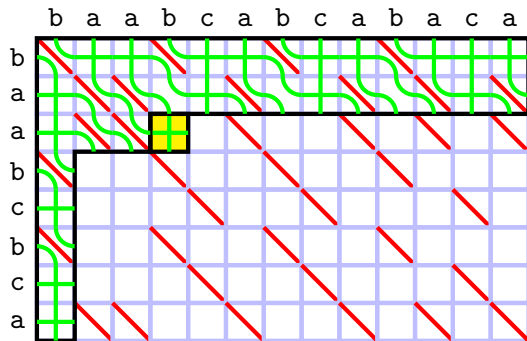
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

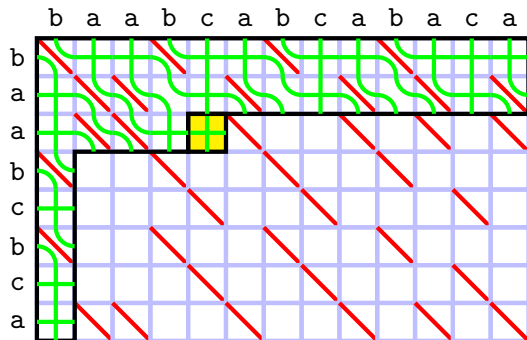
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

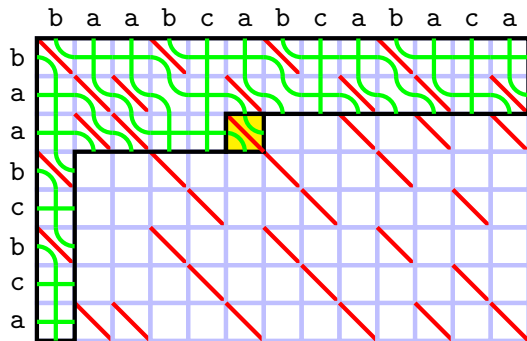
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

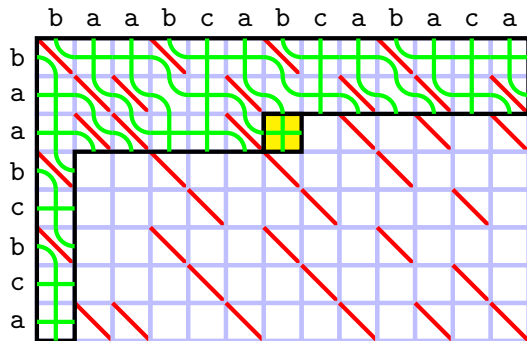
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

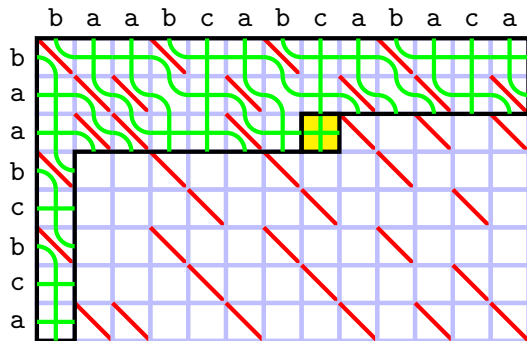
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

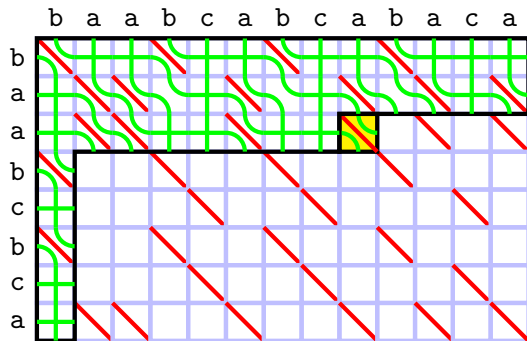
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

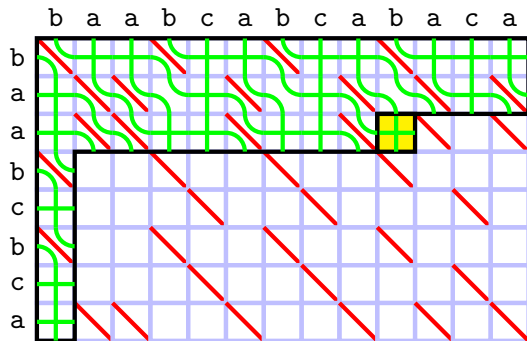
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

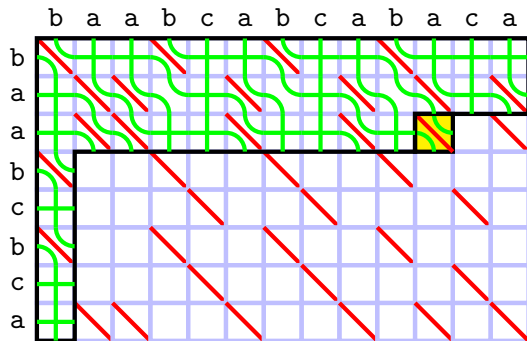
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

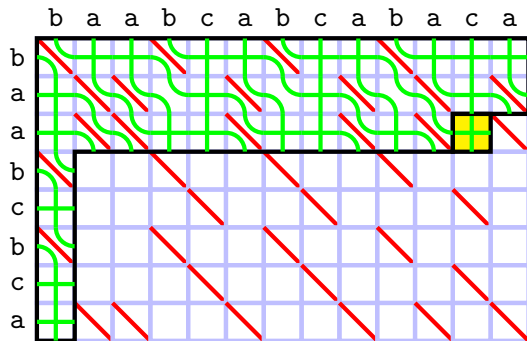
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

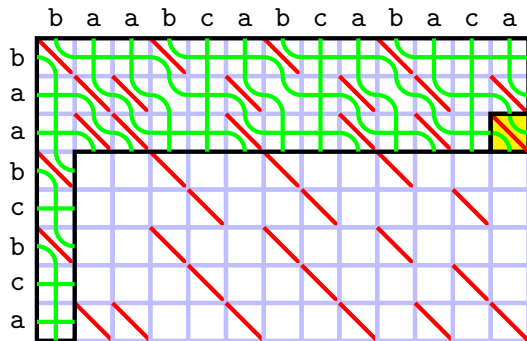
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

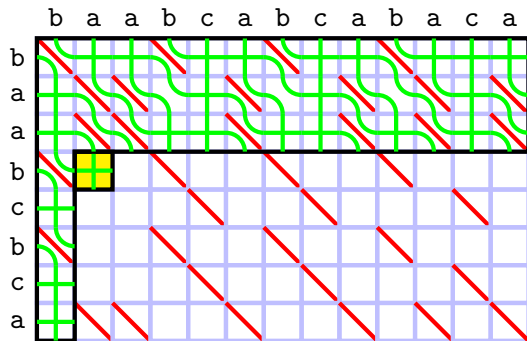
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

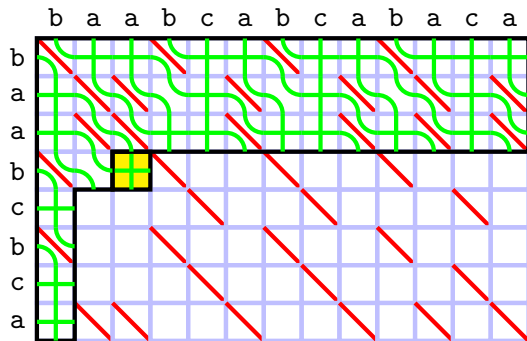
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

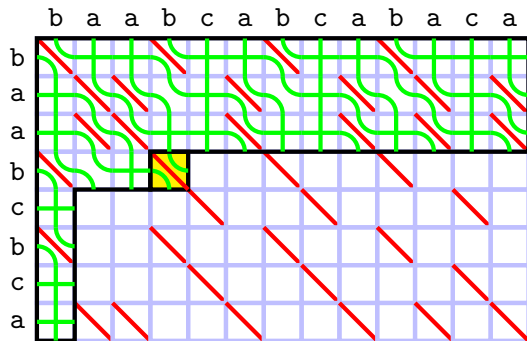
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

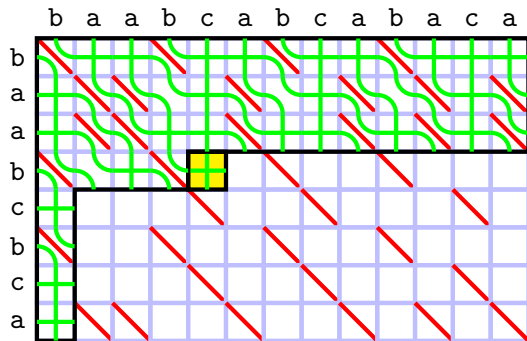
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

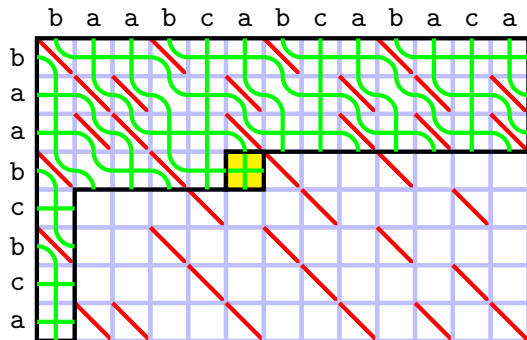
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

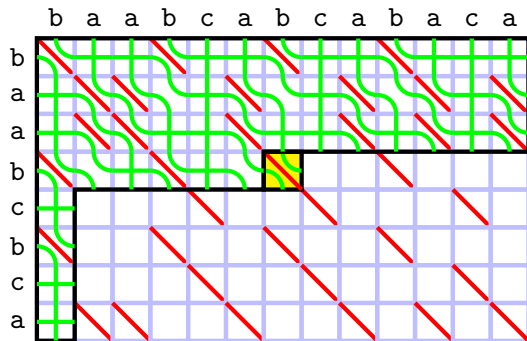
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

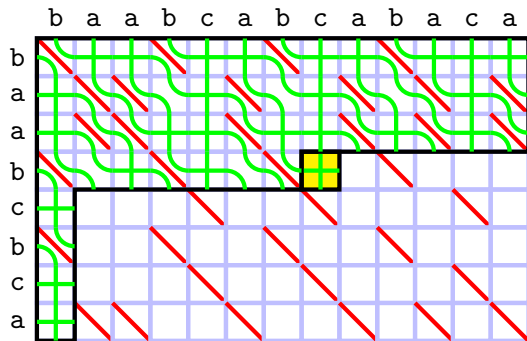
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

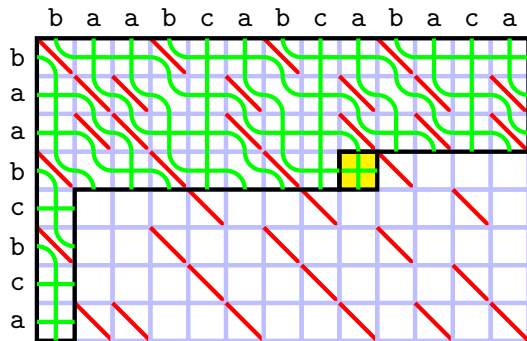
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

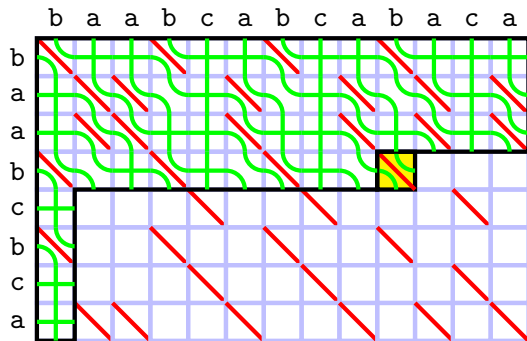
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

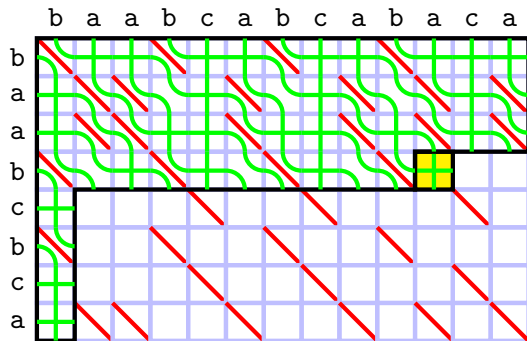
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

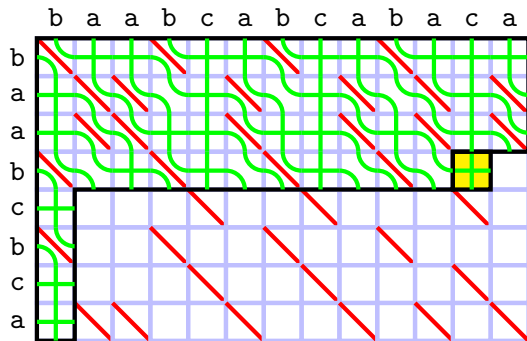
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

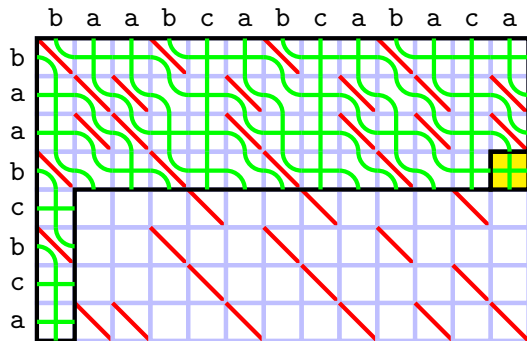
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

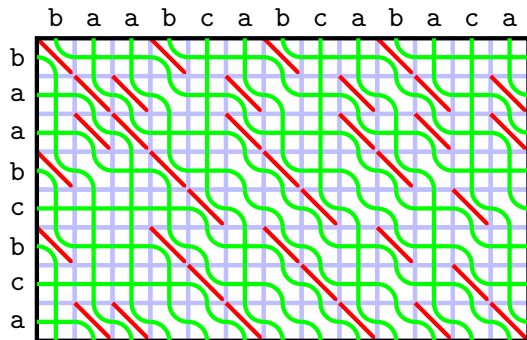
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *seaweed algorithm*

[T: 2006]



Iterate over alignment graph, tracing seaweeds

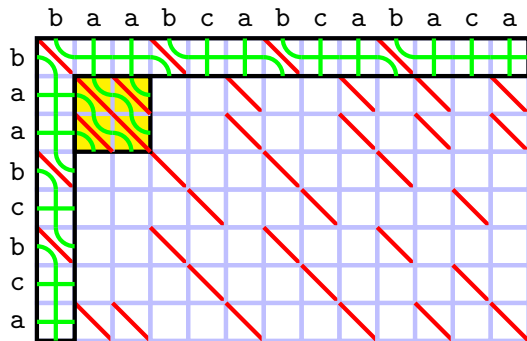
Each pair of seaweeds is allowed to cross at most once

Time  $O(mn)$

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

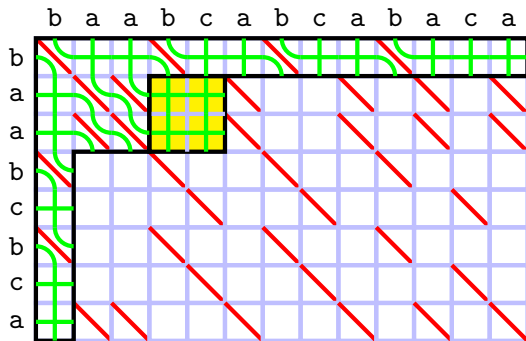
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

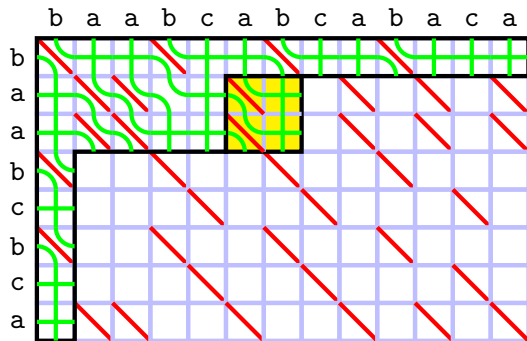
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

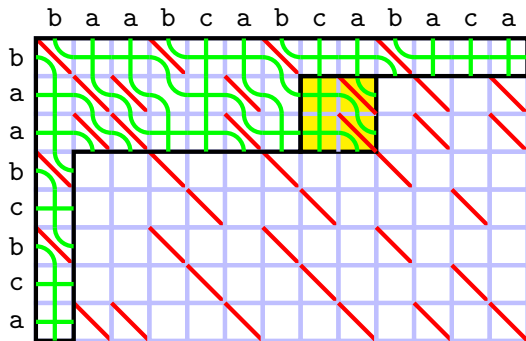
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

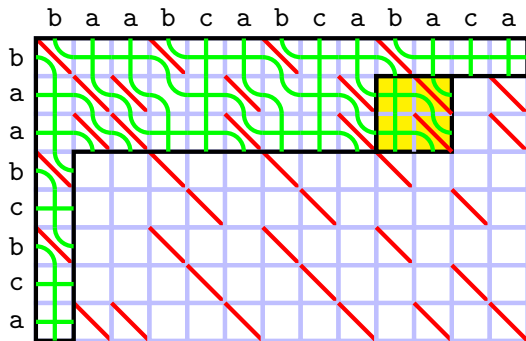
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

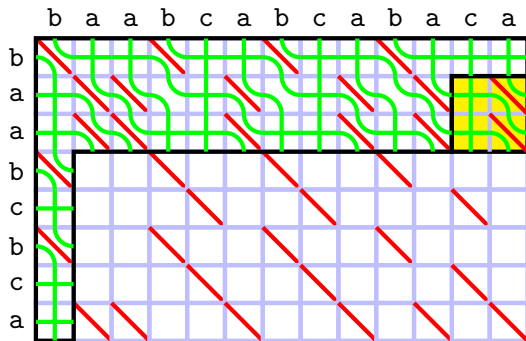
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

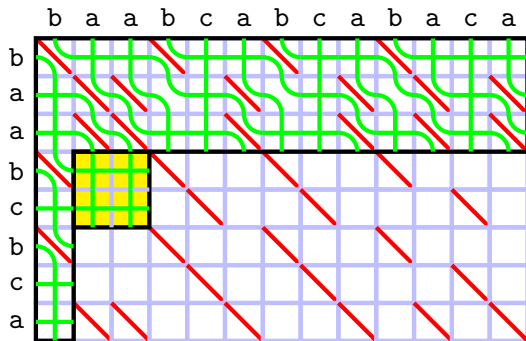
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

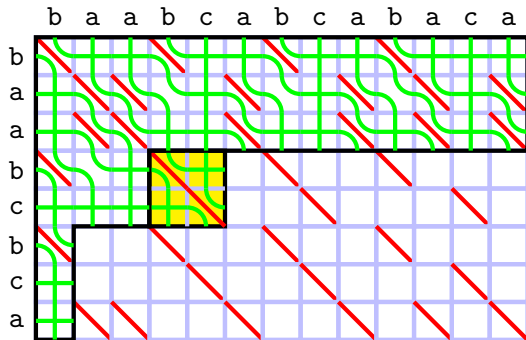
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

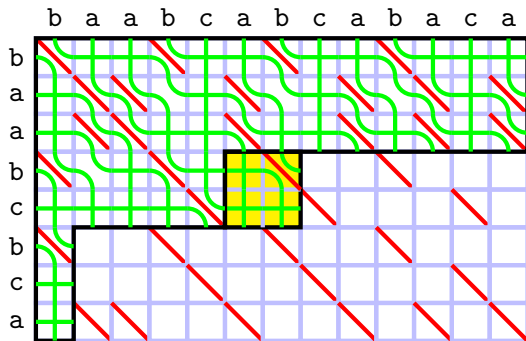
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

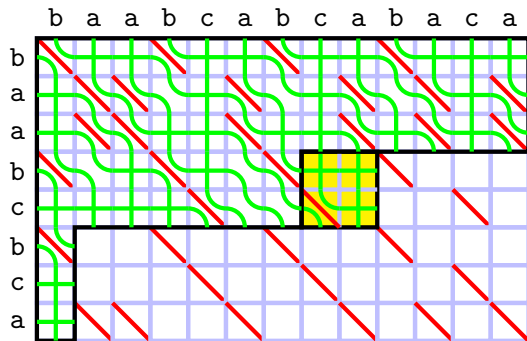
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

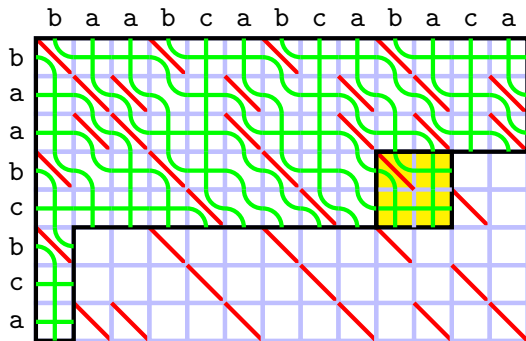
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

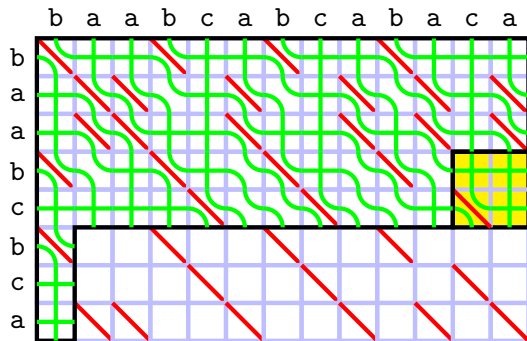
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

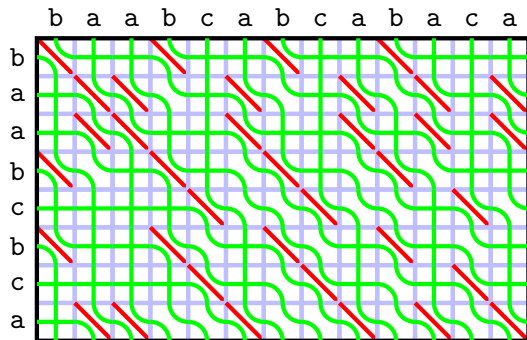
Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The *block seaweed algorithm*

[T: 2007]



Iterate over alignment graph in small blocks

Use precomputed mapping of all possible block inputs to outputs

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

# The seaweed algorithm

The block seaweed algorithm

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

Block interface:  $O(t)$  values (input chars and seaweeds), each of size  $O(\log n)$  but can be compressed to  $O(\log \log n)$  by a recursive scheme

Total work  $O\left(\frac{m}{t} \cdot \frac{n}{t} \cdot t \log \log n\right) = O\left(\frac{mn(\log \log n)^2}{\log n}\right)$ , log-cost RAM

# The seaweed algorithm

The *approximate matching* problem: determine all substrings of  $b$  that are within edit distance  $k$  of  $a$

Special case of the *edit distance matching* problem: for each position in  $b$ , determine the substring closest to  $a$

---

$O(mn)$	[Sellers: 1980], [Myers: 1986], [Landau, Vishkin: 1988]
	[Galil, Park: 1990], [Ukkonen, Wood: 1993]
$O\left(\frac{mn}{\log n}\right)$	unit-cost RAM [Wu+:1996]
$O\left(\frac{mn(\log \log n)^2}{\log n}\right)$	log-cost RAM, NEW

---

The first subquadratic algorithm for approximate matching/edit distance matching without unit-cost RAM assumption

Answers an open problem by [Cormode, Muthukrishnan: 2007]

# The seaweed algorithm

The approximate matching and edit distance matching problems

Algorithm: Run the block seaweed algorithm on  $a$  against  $b$  under the given edit score

Outputs the implicit semi-local edit distance matrix:

- an anti-Monge matrix
- the edit distance matching problem solved by its row minima

Row minima in an (anti-)Monge matrix can be found in  $O(n)$  element queries [Aggarwal+: 1987]

Each element query runs in time  $O(\log^2 n)$  using the range tree representation

Total work is dominated by the block seaweed algorithm:  $O\left(\frac{mn(\log \log n)^2}{\log n}\right)$



- 1 Semi-local string comparison
- 2 Algorithmic techniques
- 3 The seaweed algorithm
- 4 Conclusions and future work

# Conclusions and future work

Semi-local LCS problem:

- implicit representation by permutation matrices
- permutation <sup>$\Sigma$</sup>  matrix multiplication in time  $O(n^{1.5})$
- generalisation to rational edit scores

The seaweed algorithm:

- quadratic semi-local LCS
- subquadratic semi-local LCS
- improvements on related problems
- subquadratic approximate matching

Further work:

- new techniques: real-weighted edit scores?
- new applications: ???

