

Fast distance multiplication of unit-Monge matrices

Alexander Tiskin

Department of Computer Science
University of Warwick

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

- 1 Introduction
- 2 Semi-local string comparison
- 3 Sparse string comparison
- 4 Compressed string comparison
- 5 Conclusions and future work

- 1 Introduction
- 2 Semi-local string comparison
- 3 Sparse string comparison
- 4 Compressed string comparison
- 5 Conclusions and future work

Introduction

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, ...

Introduction

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 focus of this work:

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

Introduction

Terminology and notation

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

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

We consider finite and infinite integer matrices over integer and odd half-integer indices. For simplicity, index range will usually be ignored.

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

Introduction

Terminology and notation

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

In other words, $D^\Sigma(i, j)$ is the sum of all $D(i', j')$, where (i', j') is **dominated** by (i, j)

Introduction

Terminology and notation

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

In other words, $D^\Sigma(i, j)$ is the sum of all $D(i', j')$, where (i', j') is **dominated** by (i, j)

Given matrix E , its **density matrix** is

$$E^\square(i, j) = E(i^-, j^+) - E(i^-, j^-) - E(i^+, j^+) + E(i^+, j^-)$$

where $i^\pm = i \pm \frac{1}{2}$; D^Σ , E over integers; D , E^\square over odd half-integers

Introduction

Terminology and notation

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

In other words, $D^\Sigma(i, j)$ is the sum of all $D(i', j')$, where (i', j') is **dominated** by (i, j)

Given matrix E , its **density matrix** is

$$E^\square(i, j) = E(i^-, j^+) - E(i^-, j^-) - E(i^+, j^+) + E(i^+, j^-)$$

where $i^\pm = i \pm \frac{1}{2}$; D^Σ , E over integers; D , E^\square over odd half-integers

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

Matrix E is **simple**, if $(E^\square)^\Sigma = E$

Introduction

Terminology and notation

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

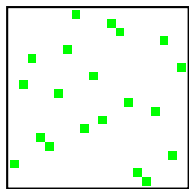
Intuition: border-to-border distances in a (weighted) planar graph

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

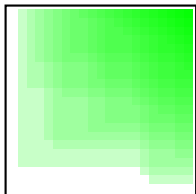
Intuition: border-to-border distances in a grid-like graph

Introduction

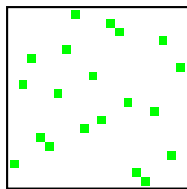
Terminology and notation



P



$P\Sigma$



$(P\Sigma)^\square = P$

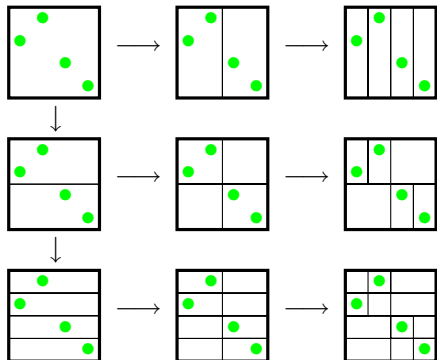
Introduction

Implicit unit-Monge matrices

Data structure for P^Σ : **range tree** on nonzeros of P

[Bentley: 1980]

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



Introduction

Implicit unit-Monge matrices

Efficient data structure for P^Σ (contd.)

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

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

Range tree supports **dominance counting** queries: how many nonzeros are dominated by a given point? Answered by decomposing query range into $\leq \log^2 n$ disjoint canonical ranges.

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

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

Introduction

Matrix distance multiplication

Matrix distance multiplication (a.k.a. **(min, +)** or **tropical** multiplication)

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

Introduction

Matrix distance multiplication

Matrix distance multiplication (a.k.a. **(min, +)** or **tropical** multiplication)

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

Matrix classes closed under distance multiplication:

- general numerical (integer, real) matrices
- Monge matrices
- simple unit-Monge matrices

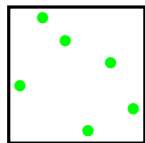
Simple unit-Monge matrices of size n form an **aperiodic monoid** (i.e. a monoid as far as possible from a group) under distance multiplication

We call it the **seaweed monoid**

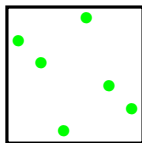
Introduction

Matrix distance multiplication

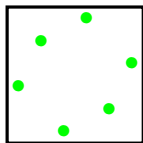
An alternative representation of the seaweed monoid: seaweeds



P_A



P_B



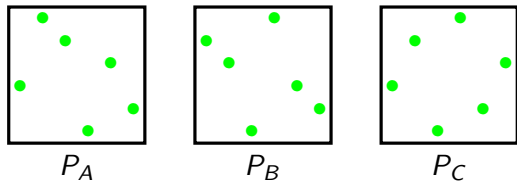
P_C

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

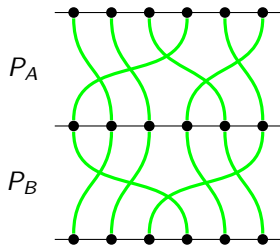
Introduction

Matrix distance multiplication

An alternative representation of the seaweed monoid: **seaweeds**



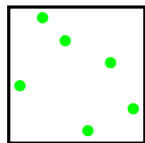
$$P_A^\Sigma \odot P_B^\Sigma = P_C^\Sigma$$



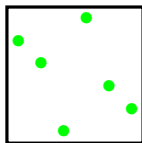
Introduction

Matrix distance multiplication

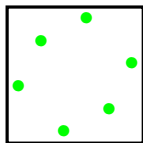
An alternative representation of the seaweed monoid: **seaweeds**



P_A

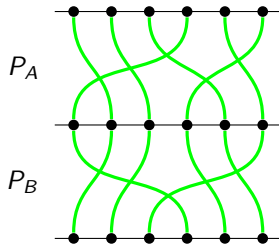


P_B



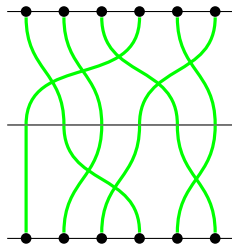
P_C

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



P_A

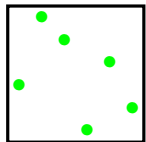
P_B



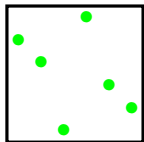
Introduction

Matrix distance multiplication

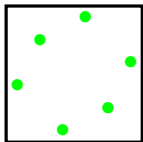
An alternative representation of the seaweed monoid: **seaweeds**



P_A

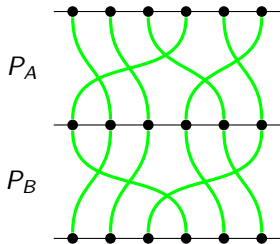


P_B



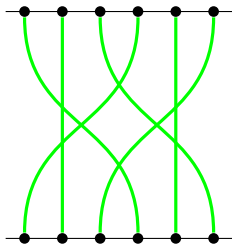
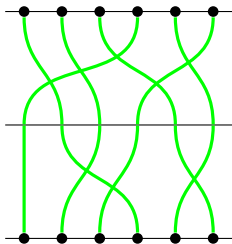
P_C

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



P_A

P_B



P_C

Introduction

Matrix distance multiplication

Seaweeds: an algebraic structure similar to a braid, except

- seaweed crossings are always level (not underpass/overpass)
- seaweed crossings are idempotent, i.e. two seaweeds can cross at most once

Seaweed composition is associative but no inverse (since an existing crossing cannot be cancelled)

Identity element ($1 \odot x = x$): identity permutation, all seaweeds uncrossed

Zero ($0 \odot x = 0$): reverse identity permutation, all seaweeds crossing

Introduction

Matrix distance multiplication

The seaweed monoid:

- $n!$ elements (permutations of size n)
- $n - 1$ generators g_1, g_2, \dots, g_{n-1} (neighbour crossings)

$$g_i^2 = g_i \quad 1 \leq i \leq n - 1 \quad (\text{idempotence})$$

$$g_i g_j = g_j g_i \quad |j - i| > 1 \quad (\text{far commutativity})$$

$$g_i g_{i+1} g_i = g_{i+1} g_i g_{i+1} \quad 1 \leq i < n - 1 \quad (\text{braid relations})$$

Computation: by a confluent rewriting system, can be obtained by

SEMIGROUPE software

[Pin: 97]

Generalisation: monoid of Bruhat cell closures in a Lie group

[Timashev: 94]

Introduction

Matrix distance multiplication

The seaweed monoid, $n = 3$

Generators: $1, a = g_1, b = g_2$

Non-generator elements: $ab, ba, aba = 0$

Rewriting system:

$$aa \rightarrow a$$

$$bb \rightarrow b$$

$$bab \rightarrow 0$$

$$aba \rightarrow 0$$

Introduction

Matrix distance multiplication

The seaweed monoid, $n = 4$

Generators: $1, a = g_1, b = g_2, c = g_3$

Non-generator elements: $ab, ac, ba, bc, cb, aba, abc, acb, bac, bcb, cba, abac, abcb, acba, bacb, bcba, abacb, abcba, bacba, abacba = 0$

Rewriting system:

$$aa \rightarrow a$$

$$bb \rightarrow b$$

$$ca \rightarrow ac$$

$$cc \rightarrow c$$

$$bab \rightarrow aba$$

$$cbc \rightarrow bcb$$

$$cbac \rightarrow bcba$$

$$abacba \rightarrow 0$$

Introduction

Matrix distance multiplication

The **implicit matrix distance multiplication** problem on simple unit-Monge matrices: given P_A , P_B , compute P_C , such that $P_A^\Sigma \odot P_B^\Sigma = P_C^\Sigma$

Introduction

Matrix distance multiplication

Matrix distance multiplication: running time

matrix type	time	
general	$O(n^3)$	standard
Monge	$O(n^2)$	by [Aggarwal+: 1987]
simple unit-Monge	$O(n^{1.5})$	[T: 2006]
(implicit)	$O(n \log n)$	[T: NEW]

Introduction

Matrix distance multiplication

Matrix distance multiplication (implicit): the algorithm

$$P_C^\Sigma(i, k) = \min_j (P_A^\Sigma(i, j) + P_B^\Sigma(j, k))$$

Divide-and-conquer on the range of j

Divide P_A horizontally, P_B vertically; two subproblems of effective size $n/2$:

$$P_{A,lo}^\Sigma \odot P_{B,lo}^\Sigma = P_{C,lo}^\Sigma \quad P_{A,hi}^\Sigma \odot P_{B,hi}^\Sigma = P_{C,hi}^\Sigma$$

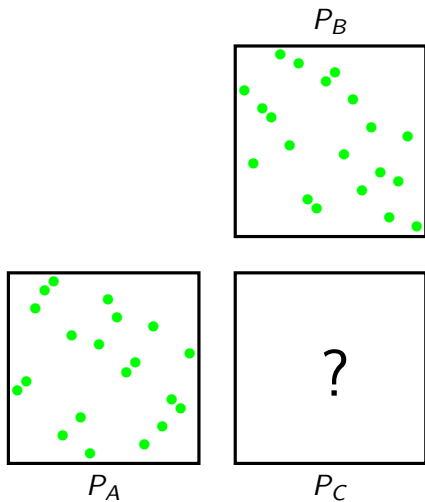
Conquer: most (but not all!) nonzeros of $P_{C,lo}$, $P_{C,hi}$ appear in P_C

Missing nonzeros can be obtained in time $O(n)$ using the Monge property

Overall time $O(n \log n)$

Introduction

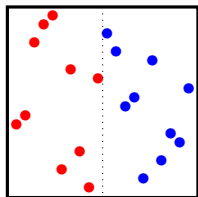
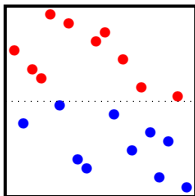
Matrix distance multiplication



Introduction

Matrix distance multiplication

$P_{B,lo}, P_{B,hi}$

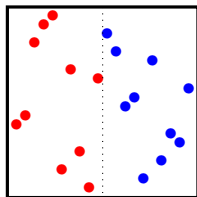
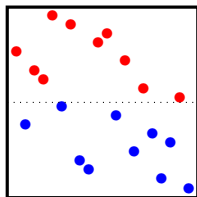


$P_{A,lo}, P_{A,hi}$

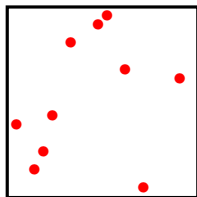
Introduction

Matrix distance multiplication

$P_{B,lo}, P_{B,hi}$



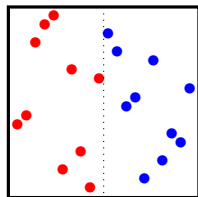
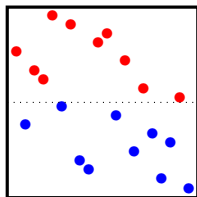
$P_{A,lo}, P_{A,hi}$



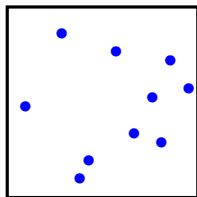
Introduction

Matrix distance multiplication

$P_{B,lo}, P_{B,hi}$



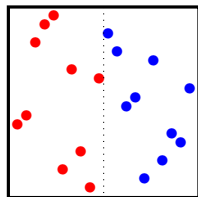
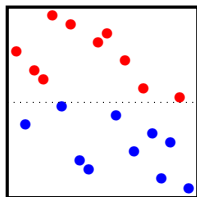
$P_{A,lo}, P_{A,hi}$



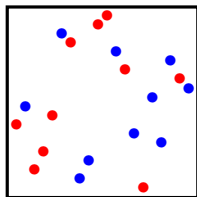
Introduction

Matrix distance multiplication

$P_{B,lo}, P_{B,hi}$



$P_{A,lo}, P_{A,hi}$



$P_{C,lo} + P_{C,hi}$

Introduction

Matrix distance multiplication



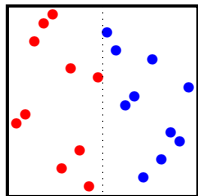
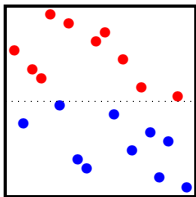
VS



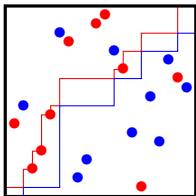
Introduction

Matrix distance multiplication

$P_{B,lo}, P_{B,hi}$



$P_{A,lo}, P_{A,hi}$

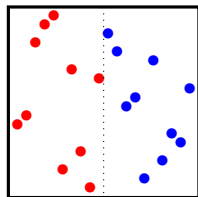
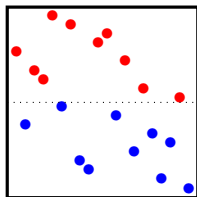


$P_{C,lo} + P_{C,hi}$

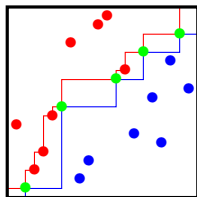
Introduction

Matrix distance multiplication

$P_{B,lo}, P_{B,hi}$



$P_{A,lo}, P_{A,hi}$



P_C

- 1 Introduction
- 2 Semi-local string comparison**
- 3 Sparse string comparison
- 4 Compressed string comparison
- 5 Conclusions and future work

Semi-local string comparison

Semi-local LCS and edit distance

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

Semi-local string comparison

Semi-local LCS and edit distance

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**:

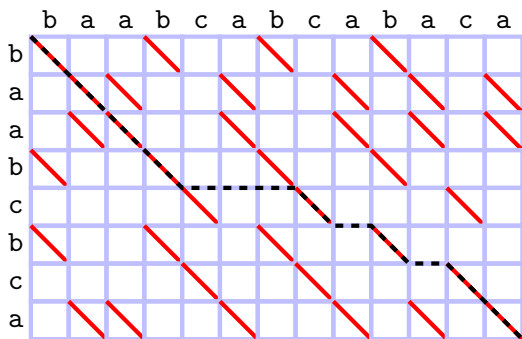
- length of longest string that is a subsequence of both a and b
- equivalently, **alignment score**, where $score(match) = 1$ and $score(mismatch) = 0$

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

Semi-local string comparison

Semi-local LCS and edit distance

LCS on the **alignment graph** (directed, acyclic)



blue = 0

red = 1

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

LCS = highest-score corner-to-corner path

Semi-local string comparison

Semi-local LCS and edit distance

LCS: running time

$$O(mn)$$

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

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

$$\sigma = O(1)$$

[Wagner, Fischer: 1974]

[Masek, Paterson: 1980]

[Crochemore+: 2003]

[Paterson, Dancik: 1994]

Semi-local string comparison

Semi-local LCS and edit distance

LCS: the dynamic programming algorithm [Wagner, Fischer: 1974]

Sweep alignment graph, respecting node dependencies

Running time $O(mn)$

LCS: the block dynamic programming algorithm

[Masek, Paterson: 1980]

Sweep alignment graph in square blocks, respecting block dependencies

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

Block interface: $O(t)$ inputs/outputs, each of size $O(\log \sigma)$

Use precomputed mapping of all possible input/output combinations

Running time $O\left(\frac{mn}{\log n}\right)$ when $\sigma = O(1)$, even on log-cost RAM

Semi-local string comparison

Semi-local LCS and edit distance

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

Semi-local LCS and edit distance

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 $O(m^2 + n^2)$ LCS scores, allowed to be represented implicitly

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

Semi-local string comparison

Semi-local LCS and edit distance

The **three-way semi-local LCS** problem:

- **string-substring, prefix-suffix, suffix-prefix LCS**
- no substring-string LCS

Semi-local string comparison

Semi-local LCS and edit distance

The **three-way semi-local LCS** problem:

- **string-substring, prefix-suffix, suffix-prefix LCS**
- no substring-string LCS

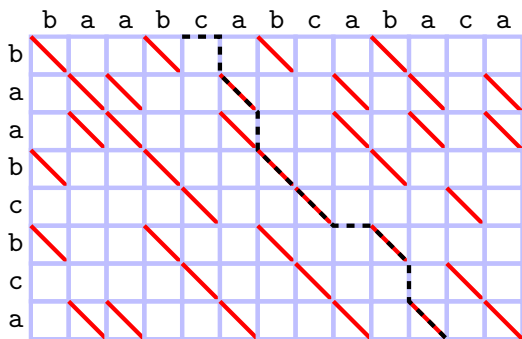
Output: the (partially filled) **three-way highest-score matrix** of $O(n^2)$ LCS scores, allowed to be represented implicitly

Suitable for $m \gg n$

Semi-local string comparison

Semi-local LCS and edit distance

Semi-local LCS on 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

Semi-local LCS and edit distance

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

Semi-local LCS and edit distance

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

Edit distance: $m \cdot w_{del} + n \cdot w_{in} - score(a, b)$, where $score(match) = w_{in} + w_{del}$ and $score(mismatch) = w_{in} + w_{del} - w_{sub}$

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

Semi-local string comparison

Highest-score matrices

Semi-local LCS: the highest-score matrix

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	6	7
-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

Highest-score matrices

Semi-local LCS: highest-score matrix 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]

Semi-local string comparison

Highest-score matrices

Semi-local LCS: running time

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

Semi-local string comparison

Highest-score matrices

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 **un**matched characters

Properties of matrix Q :

- Q is simple unit-Monge
- therefore, $Q = P^\Sigma$ for some permutation matrix P

$P = Q^\square = -A^\square$ is an **implicit representation** of A

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

Semi-local string comparison

Highest-score matrices

Semi-local LCS: the highest-score matrix

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	7	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 $i > j$

Semi-local string comparison

Highest-score matrices

Semi-local LCS: the highest-score matrix

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	7	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

$a = \text{"baabcbca"}$

$b = \text{"baabcabcbacaca"}$

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

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

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

blue: difference 0

red: difference 1

Semi-local string comparison

Highest-score matrices

Semi-local LCS: the highest-score matrix

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	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	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 $i > j$

blue: difference 0

red: difference 1

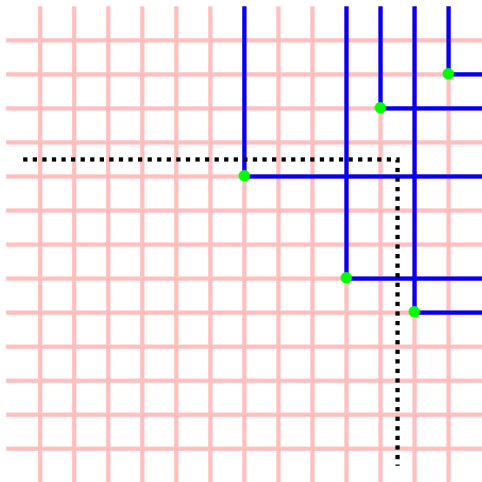
green: $P(i, j) = 1$

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

Semi-local string comparison

Highest-score matrices

Semi-local LCS: the highest-score matrix



$a = \text{"baabcbca"}$

$b = \text{"baabcabcbabaca"}$

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

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

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

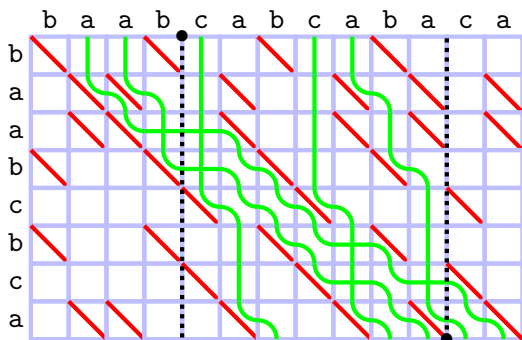
$11 - 4 - 2 = 5$

P gives an **implicit representation** of A

Semi-local string comparison

Highest-score matrices

The **seaweeds** in the alignment graph



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

$a = \text{"baabcbca"}$

$b = \text{"baabcabcbaca"}$

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

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

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

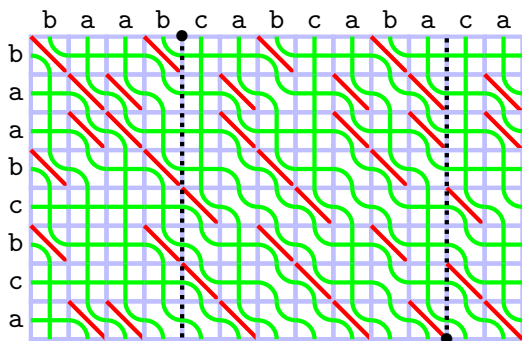
$11 - 4 - 2 = 5$

P gives an **implicit representation** of A

Semi-local string comparison

Highest-score matrices

The **seaweeds** in the alignment graph



$a = \text{"baabcbca"}$

$b = \text{"baabcabcbaca"}$

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

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

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

$11 - 4 - 2 = 5$

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

- 1 Introduction
- 2 Semi-local string comparison
- 3 Sparse string comparison**
- 4 Compressed string comparison
- 5 Conclusions and future work

Sparse string comparison

Sparse semi-local LCS

A **permutation string** (**permutation**): all characters distinct

The **LCS** problem on **permutations**: $m = n$ matches

Equivalent to

- **longest increasing subsequence (LIS)** in a permutation
- **maximum clique** in a **permutation graph**
- **maximum planar matching** in an embedded bipartite graph

The **semi-local LCS** problem on **permutations**

Equivalent to

- **longest increasing subsequence (LIS)** in every substring of a permutation

Sparse string comparison

Sparse semi-local LCS

LCS on permutations: running time

$O(n \log n)$

implicit in [Erdős, Szekeres: 1935]
[Robinson: 1938]
[Knuth: 1970]
[Dijkstra: 1980]

$O(n \log \log n)$ unit-cost RAM

[Chang, Wang: 1992]
[Bespamyatnikh, Segal: 2000]

Sparse string comparison

Sparse semi-local LCS

Semi-local LCS on permutations: 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: NEW]

Sparse string comparison

Sparse semi-local LCS

Semi-local LCS on permutations: the algorithm

Divide-and-conquer on the alignment graph

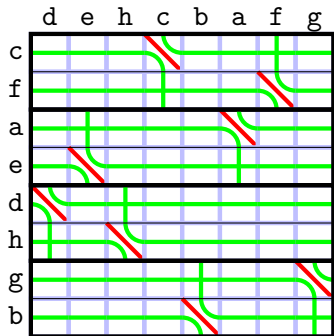
Divide graph (say) horizontally; two subproblems of effective size $n/2$

Conquer: implicit distance multiplication of unit-Monge matrices, time $O(n \log n)$

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

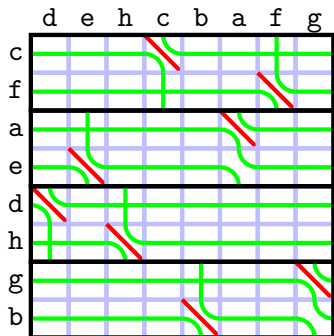
Sparse string comparison

Sparse semi-local LCS



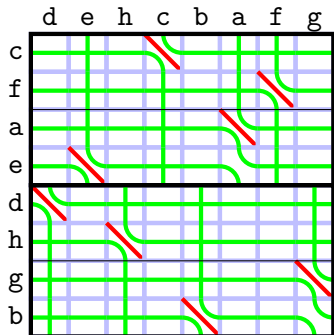
Sparse string comparison

Sparse semi-local LCS



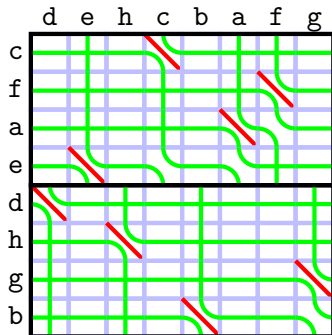
Sparse string comparison

Sparse semi-local LCS



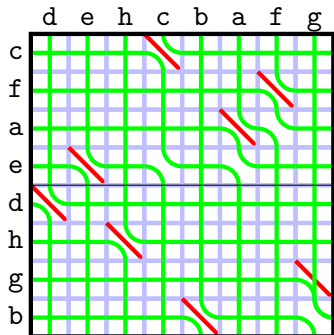
Sparse string comparison

Sparse semi-local LCS



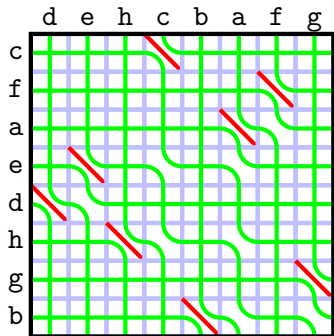
Sparse string comparison

Sparse semi-local LCS



Sparse string comparison

Sparse semi-local LCS



Sparse string comparison

Longest piecewise monotone subsequences

A **k -piece increasing sequence**: a concatenation of k increasing sequences

A **k -modal sequence**: a concatenation of k alternating increasing and decreasing sequences

The **longest k -piece increasing (k -modal) subsequence** problem: given permutation a , find its longest subsequence of the specified type

Main idea:

- k -piece increasing: align a^k against permutation id
- k -modal: similar, but with alternating reversals in a^k

Sparse string comparison

Longest piecewise monotone subsequences

Longest k -piece increasing (k -modal) subsequence: running time

$O(nk \log n)$ k -modal

[Demange+: 2007]

$O(nk \log n)$

via [Hunt, Szymanski: 1977]

$O(n \log^2 n)$

[T: NEW]

Sparse string comparison

Longest piecewise monotone subsequences

Longest k -piece increasing subsequence: algorithm A

Direct alignment of a^k against id as sparse LCS

Overall time $O(nk \log n)$

Sparse string comparison

Longest piecewise monotone subsequences

Longest k -piece increasing subsequence: algorithm A

Direct alignment of a^k against id as sparse LCS

Overall time $O(nk \log n)$

Longest k -piece increasing subsequence: algorithm B

Three-way semi-local LCS on a against id , highest-score matrix A

Compute $A^{\odot k}$ (A to distance power k) by square/multiply

Query the LCS of a^k against id

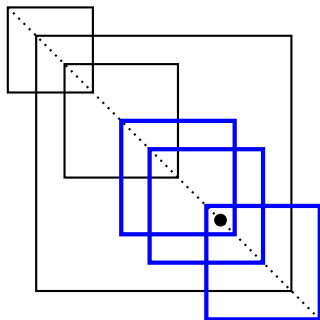
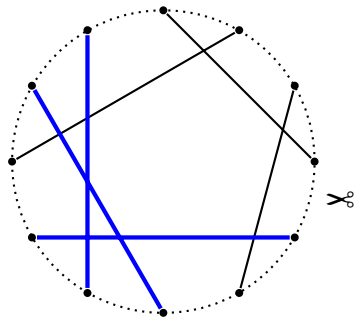
Overall time $O(n \log^2 n) + O(n \log^2 n) + O(n) = O(n \log^2 n)$

Algorithm B faster than Algorithm A unless $k \leq \log n$

Sparse string comparison

Maximum clique in a circle graph

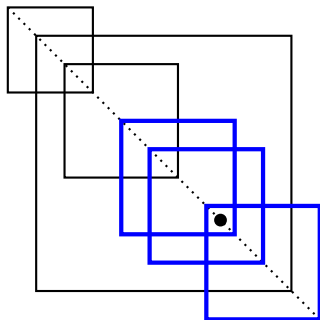
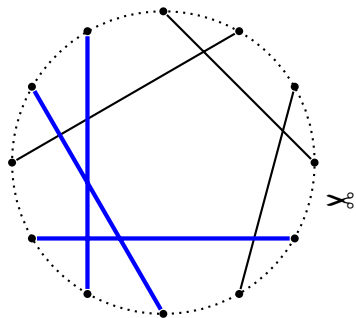
The **maximum clique** problem in a **circle graph**: given a circle with n chords, determine the maximum-size subset of pairwise intersecting chords



Sparse string comparison

Maximum clique in a circle graph

The **maximum clique** problem in a **circle graph**: given a circle with n chords, determine the maximum-size subset of pairwise intersecting chords



Standard reduction to an **interval model**: cut the circle and lay it out on the line; chords become intervals (here drawn as square diagonals)

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

Sparse string comparison

Maximum clique in a circle graph

Maximum clique in a circle graph: running time

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

Sparse string comparison

Maximum clique in a circle graph

Maximum clique in a circle graph: the algorithm

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

Run semi-local LCS on permutations, build range tree: time $O(n \log^2 n)$

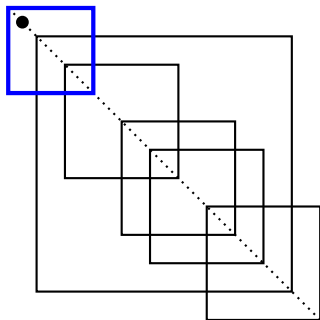
Check all $2n + 1$ possible common intersection points

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

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

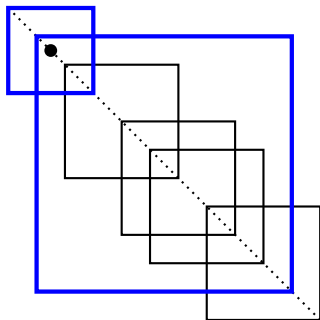
Sparse string comparison

Maximum clique in a circle graph



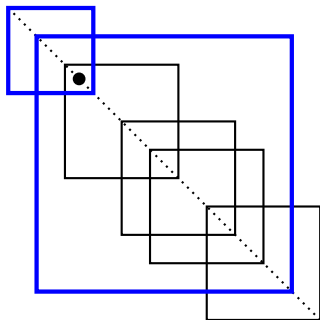
Sparse string comparison

Maximum clique in a circle graph



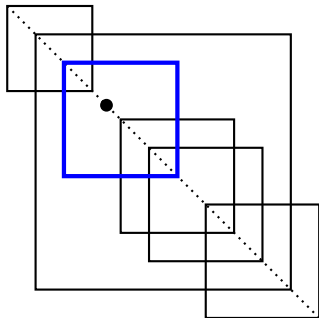
Sparse string comparison

Maximum clique in a circle graph



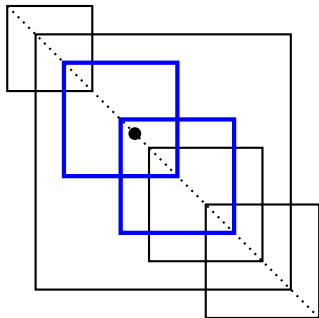
Sparse string comparison

Maximum clique in a circle graph



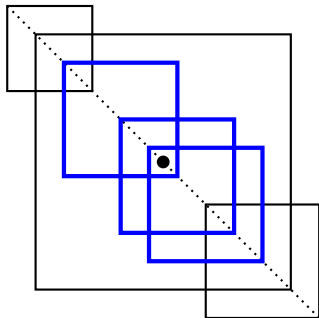
Sparse string comparison

Maximum clique in a circle graph



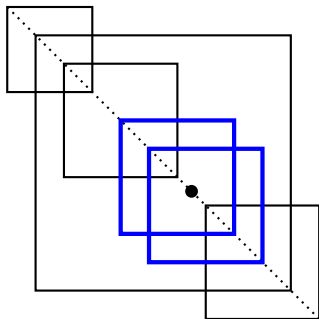
Sparse string comparison

Maximum clique in a circle graph



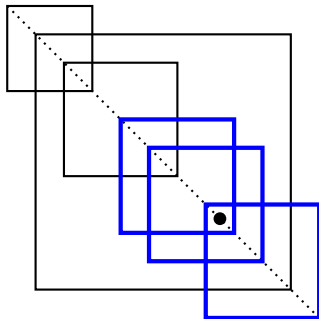
Sparse string comparison

Maximum clique in a circle graph



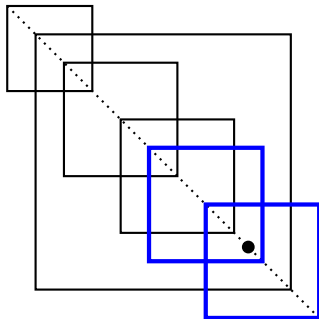
Sparse string comparison

Maximum clique in a circle graph



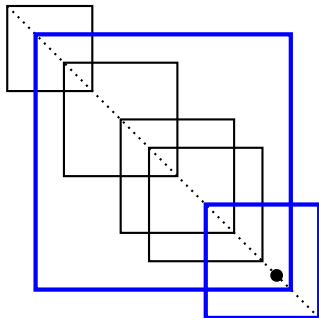
Sparse string comparison

Maximum clique in a circle graph



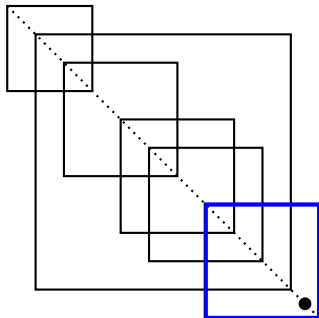
Sparse string comparison

Maximum clique in a circle graph



Sparse string comparison

Maximum clique in a circle graph



Sparse string comparison

Maximum clique in a circle graph

Parameterised maximum clique: running time sensitive e.g. to

- the number of edges e
- the size l of maximum clique
- the maximum number d of intervals covering a point

We have $l \leq d \leq n$, $e \leq n^2$

Sparse string comparison

Maximum clique in a circle graph

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

Sparse string comparison

Maximum clique in a circle graph

Parameterised maximum clique in a circle graph: the algorithm

Run semi-local LCS on diagonal blocks of size d , build range trees: time $O(n/d \cdot d \log^2 d) = O(n \log^2 d)$

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

Check all $2n + 1$ possible common intersection points as before: time $O(n \log^2 d)$

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

- 1 Introduction
- 2 Semi-local string comparison
- 3 Sparse string comparison
- 4 Compressed string comparison**
- 5 Conclusions and future work

Compressed string comparison

Grammar compression

Notation: text T of length m ; pattern P of length n

A **GC-string (grammar-compressed string)** T is a straight-line program (context-free grammar) generating $T = T_{\bar{m}}$ by \bar{m} assignments of the form

- $T_r = \alpha$, where α is an alphabet character
- $T_r = T_s T_t$, where $s, t < r$

Covers various compression types, e.g. Lempel–Ziv

Can have $m = O(c^{\bar{m}})$. Assume index arithmetic on T still $O(1)$.

Compressed string comparison

Three-way semi-local LCS on GC-strings

LCS: running time

T	P			
plain	plain	$O(mn)$		[Wagner, Fischer: 1974]
		$O(\frac{mn}{\log n})$		[Masek, Paterson: 1980]
				[Crochemore+: 2003]
GC	plain	$O(\bar{m}n^3 + \dots)$	general CFG	[Myers: 1995]
		$O(\bar{m}n^{1.5})$	3-way semi	[T: 2008]
		$O(\bar{m}n \log n)$	3-way semi	[T: NEW]
GC	GC	NP-hard		[Lifshits: 2005]

Compressed string comparison

Three-way semi-local LCS on GC-strings

Three-way semi-local LCS (GC text, plain pattern): the algorithm

Algorithm: for every T_r , three-way semi-local LCS against P by distance matrix multiplication

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

Compressed string comparison

Subsequence recognition on GC-strings

The **global subsequence recognition** problem: does T contain P as a subsequence?

The **local subsequence recognition** problem: determine the number of minimal substrings of T containing P as a subsequence

Compressed string comparison

Subsequence recognition on GC-strings

Global subsequence recognition: running time

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

Compressed string comparison

Subsequence recognition on GC-strings

Local subsequence recognition: running time

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

Compressed string comparison

Subsequence recognition on GC-strings

Local subsequence recognition (GC text, plain pattern): the algorithm

Algorithm: first, three-way semi-local LCS for every T_r against P in time $O(\bar{m}n \log n)$

Compressed string comparison

Subsequence recognition on GC-strings

Local subsequence recognition (GC text, plain pattern): the algorithm

Algorithm: first, three-way semi-local LCS for every T_r against P in time $O(\bar{m}n \log n)$

Given an assignment $T = T' T''$, count by recursion

- minimal substrings in T' containing P as subsequence
- minimal substrings in T'' containing P as subsequence

Compressed string comparison

Subsequence recognition on GC-strings

Local subsequence recognition (GC text, plain pattern): the algorithm

Algorithm: first, three-way semi-local LCS for every T_r against P in time $O(\bar{m}n \log n)$

Given an assignment $T = T' T''$, count by recursion

- minimal substrings in T' containing P as subsequence
- minimal substrings in T'' containing P as subsequence

Then for each prefix-suffix split $P = P' P''$, count by prefix-suffix and suffix-prefix LCS queries

- minimal suffixes of T' containing P' as subsequence
- minimal prefixes of T'' containing P'' as subsequence

Compressed string comparison

Subsequence recognition on GC-strings

Local subsequence recognition (GC text, plain pattern): the algorithm

Algorithm: first, three-way semi-local LCS for every T_r against P in time $O(\bar{m}n \log n)$

Given an assignment $T = T' T''$, count by recursion

- minimal substrings in T' containing P as subsequence
- minimal substrings in T'' containing P as subsequence

Then for each prefix-suffix split $P = P' P''$, count by prefix-suffix and suffix-prefix LCS queries

- minimal suffixes of T' containing P' as subsequence
- minimal prefixes of T'' containing P'' as subsequence

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

- 1 Introduction
- 2 Semi-local string comparison
- 3 Sparse string comparison
- 4 Compressed string comparison
- 5 Conclusions and future work**

Conclusions and future work

Implicit unit-Monge matrices:

- the seaweed monoid
- distance multiplication in time $O(n \log n)$
- next: lower bound

Semi-local LCS problem:

- representation by implicit unit-Monge matrices
- implicit unit-Monge matrix distance multiplication in time $O(n \log n)$
- generalisation to rational alignment scores
- next: real alignment scores

The seaweed and block seaweed algorithms:

- a simple algorithm for semi-local LCS
- semi-local LCS in time $o(mn)$
- improvements on related problems

Conclusions and future work

Sparse string comparison:

- semi-local LCS on permutations in time $O(n \log^2 n)$
- maximum clique in a circle graph in time $O(n \log^2 n)$
- improvements on related problems

Compressed string comparison:

- three-way semi-local LCS on GC text against plain pattern
- subsequence recognition in GC-strings
- next: approximate matching in GC-strings

Beyond semi-locality:

- quasi-local string comparison
- sparse spliced alignment
- next: full locality

Centre for Discrete Mathematics and Its Applications (DIMAP), University of Warwick (Coventry, United Kingdom)

Joint research centre between:

- Mathematics Institute
- Department of Computer Science
- Warwick Business School

Vacancies at all levels (including PhD), for details: google “dimap”



References I