

Semi-local string comparison: Algorithmic applications

Alexander Tiskin

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

Department of Computer Science
University of Warwick

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

String comparison: find *similar* patterns in two strings

- *global*: compare whole string against whole string
- *local*: compare substrings against substrings
- *semi-local*: compare whole string against substrings (will extend this definition later)

Often called “approximate string matching” (no relation to approximation algorithms!)

Applications: computational biology, image recognition, ...

- 1 Semi-local string comparison
- 2 Efficient output representation
- 3 Fast block iteration and applications
- 4 Fast divide-and-conquer and applications
- 5 Further algorithmic ideas and applications
- 6 Conclusions and future work

- 1 Semi-local string comparison
- 2 Efficient output representation
- 3 Fast block iteration and applications
- 4 Fast divide-and-conquer and applications
- 5 Further algorithmic ideas and applications
- 6 Conclusions and future work

Semi-local string comparison

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

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

Recall: the *longest common subsequence (LCS) problem*

Determine the LCS length for string a against string b

A variant of the *edit distance problem*

$O(mn)$ [Needleman, Wunsch, 1970]

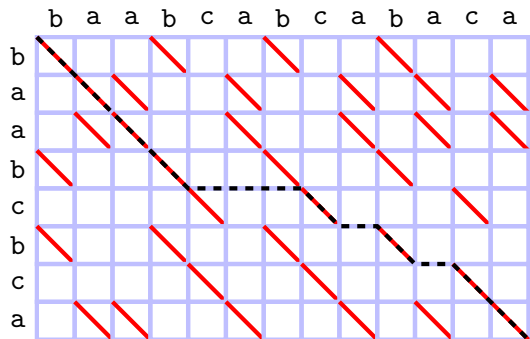
$O\left(\frac{mn \log \log n}{\log n}\right)$ implicit in [Masek, Paterson, 1980]

$O\left(\frac{mn}{\log n}\right)$ assuming $\sigma = O(1)$ [Masek, Paterson, 1980]
also [Crochemore+, 2003]

Semi-local string comparison

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

$m \leq n$



blue = 0

red = 1

Alignment graph

Longest common subsequence \sim longest source-to-sink path

Semi-local string comparison

The *semi-local longest common subsequences (LCS) problem*

Determine the LCS length for

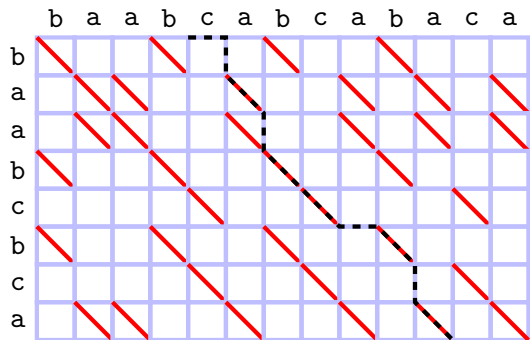
- string a against every substring of b
- every substring of a against string b
- every prefix of a against every suffix of b
- every suffix of a against every prefix of b

Total $\Theta(n^2)$ outputs, allowed to be represented implicitly

Semi-local string comparison

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

$m \leq n$



blue = 0
red = 1

Semi-local LCS \sim all longest border-to-border paths
(string-substring \sim top-to-bottom, etc.)

Semi-local string comparison

Semi-local LCS problem: the output

size $O(n^2)$

query time $O(1)$

trivial

Semi-local string comparison

Semi-local LCS problem: the output

size $O(n^2)$ query time $O(1)$ trivial

Representing the output implicitly

size $O(m^{1/2}n)$ query time $O(\log n)$ [Alves+, 2003]

size $O(n)$ query time $O(n)$ [Alves+, 2005]

size $O(n \log n)$ query time $O(\log^2 n)$ [T, 2006]

Semi-local string comparison

Semi-local LCS problem: the output

size $O(n^2)$ query time $O(1)$ trivial

Representing the output implicitly

size $O(m^{1/2}n)$ query time $O(\log n)$ [Alves+, 2003]

size $O(n)$ query time $O(n)$ [Alves+, 2005]

size $O(n \log n)$ query time $O(\log^2 n)$ [T, 2006]

In a stronger model:

size $O(n)$ query time $O\left(\frac{\log n}{\log \log n}\right)$ [T, 2006]

Semi-local string comparison

Semi-local LCS problem: computation time

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

Semi-local string comparison

Recall: the *longest increasing subsequence (LIS)* problem

Determine the LCS length for a permutation of length n against permutation $id = (1, 2, \dots, n)$

$O(n^2)$ naive

$O(n \log n)$ implicit in [Erdős, Szekeres, 1935]
also [Robinson, 1938], [Knuth, 1970], [Dijkstra, 1980]

$O(n \log \log n)$ in the RAM model [Hunt, Szymanski, 1977]
also [Chang, Wang, 1992], [Bespamyatnikh, Segal, 2000]

Semi-local string comparison

Semi-local LCS problem: running time on permutations

| | |
|--------------------------------|--|
| $O(n^2 \log n)$ | naive |
| $O(n^2)$ | restricted, [Albert+, 2003] restricted, [Chen+, 2005] |
| $O(n^{1.5} \log n)$ randomised | restricted, [Albert+, 2007] |
| $O(n^{1.5})$ | [T, 2006] |

- 1 Semi-local string comparison
- 2 Efficient output representation**
- 3 Fast block iteration and applications
- 4 Fast divide-and-conquer and applications
- 5 Further algorithmic ideas and applications
- 6 Conclusions and future work

Efficient output representation

Notation

Integers $0, 1, 2, \dots$ Odd half-integers $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \dots$

$$x \triangleleft y \Leftrightarrow y - x = 1 \quad x \triangleleft\!\! \triangleleft y \Leftrightarrow y - x = \frac{1}{2}$$

Definition

Point (i_0, j_0) *dominates* point (i, j) , if $i_0 < i$ and $j < j_0$

Efficient output representation

Definition

Point (i, j) is *A-critical*, if

$$A(i^-, j^-) \triangleleft A(i^-, j^+) = A(i^+, j^-) = A(i^+, j^+)$$

where $i^- \triangleleft i \triangleleft i^+ \quad j^- \triangleleft j \triangleleft j^+$

Notation

$d_A(i_0, j_0)$ is the number of *A-critical* points dominated by (i_0, j_0)

Efficient output representation

Lemma

$$A(i_0, j_0) = j_0 - i_0 - d_A(i_0, j_0)$$

$j_0 - i_0$: input substring length $d_A(i_0, j_0)$: unmatched characters

Proof: simple induction

More generally:

- A is a *Monge matrix*
- its *density matrix* happens to be the permutation matrix of critical points

Efficient output representation

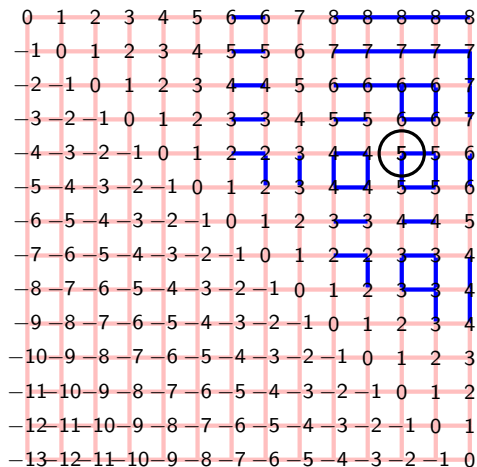
Full top-to-bottom highest-score matrix: $A(i,j)$

$0 \leq i,j \leq n$

| | | | | | | | | | | | | | |
|-----|-----|-----|-----|----|----|----|----|----|----|----|----|----|---|
| 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 |

Efficient output representation

Full top-to-bottom highest-score matrix: $A(i, j)$ $0 \leq i, j \leq n$



$$A(i^+, j) \trianglelefteq A(i^-, j)$$

$$A(i, j^-) \trianglelefteq A(i, j^+)$$

A totally monotone:

$$A(i^+, j^+) \triangleleft A(i^-, j^+) \Rightarrow A(i^+, j^-) \triangleleft A(i^-, j^-)$$

A^T totally monotone:

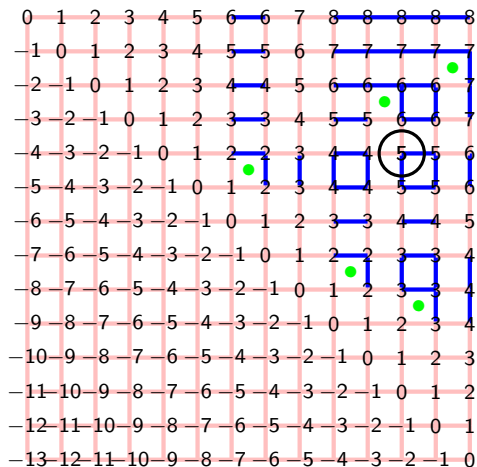
$$A(i^-, j^-) \triangleleft A(i^-, j^+) \Rightarrow A(i^+, j^-) \triangleleft A(i^+, j^+)$$

where $i^- \triangleleft i^+$, $j^- \triangleleft j^+$

blue = 0 red = 1

Efficient output representation

Full top-to-bottom highest-score matrix: $A(i, j)$ $0 \leq i, j \leq n$



$$A(i^+, j) \trianglelefteq A(i^-, j)$$

$$A(i, j^-) \trianglelefteq A(i, j^+)$$

A totally monotone:

$$A(i^+, j^+) \triangleleft A(i^-, j^+) \Rightarrow A(i^+, j^-) \triangleleft A(i^-, j^-)$$

A^T totally monotone:

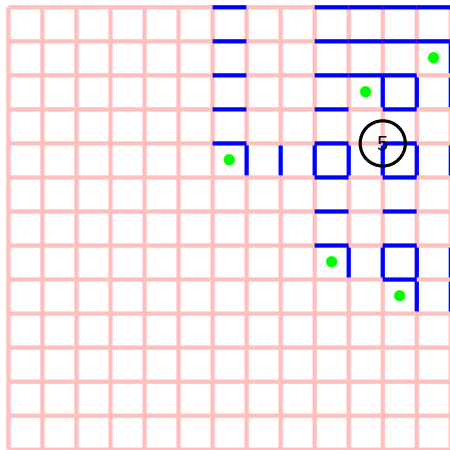
$$A(i^-, j^-) \triangleleft A(i^-, j^+) \Rightarrow A(i^+, j^-) \triangleleft A(i^+, j^+)$$

where $i^- \triangleleft i^+$, $j^- \triangleleft j^+$

blue = 0 red = 1 green = critical

Efficient output representation

Implicit top-to-bottom highest-score matrix: $A(i,j)$ $0 \leq i,j \leq n$

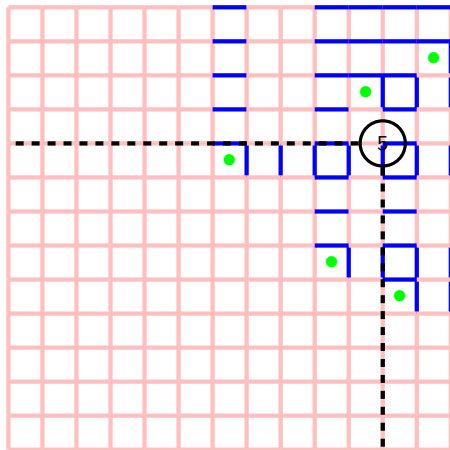


$$j_0 - i_0 - d_A(i_0, j_0) = 11 - 4 - 2 = 5$$

blue = 0 *red* = 1 *green* = critical

Efficient output representation

Implicit top-to-bottom highest-score matrix: $A(i,j)$ $0 \leq i,j \leq n$

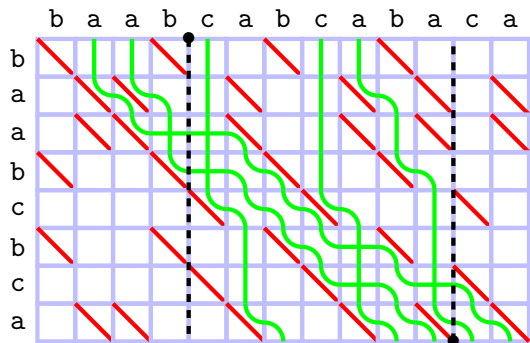


$$j_0 - i_0 - d_A(i_0, j_0) = 11 - 4 - 2 = 5$$

blue = 0 red = 1 green = critical

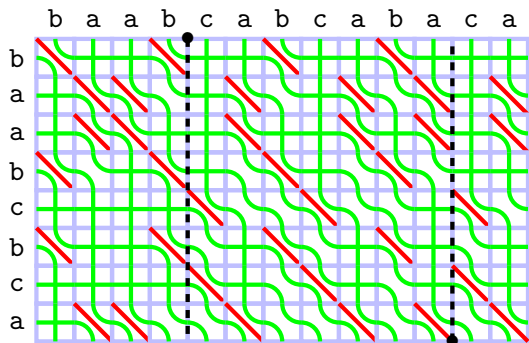
Efficient output representation

Critical point (i, j) in implicit highest-score matrix gives a *critical curve* $(top, i) \rightsquigarrow (bottom, j)$ in the alignment graph



Efficient output representation

Critical point (i, j) in implicit highest-score matrix gives a *critical curve* $(top, i) \rightsquigarrow (bottom, j)$ in the alignment graph



Also define $top \rightsquigarrow right$, $left \rightsquigarrow right$, $left \rightsquigarrow bottom$ critical curves

Gives complete border-to-border graph-theoretic matching

Efficient output representation

Gaudi's seaweeds (Casa Milà, Barcelona)



Efficient output representation

Establishing $d_A(i_0, j_0)$: *dominance counting*

Range tree:

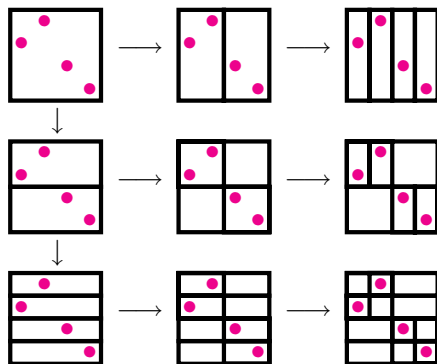
[Bentley, 1980]

- binary search tree by i -coordinate for all nodes
- rooted at its every node, binary search tree by j -coordinate for relevant nodes

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

Efficient output representation

Range tree:



Every range can be decomposed into $\leq \log^2 n$ canonical ranges

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

Efficient output representation

Theorem (Bentley, 1980)

A range tree on n points has

- *size $O(n \log n)$*
- *dominance counting query time $O(\log^2 n)$*

Theorem (JaJa+, 2004)

In the RAM model, there is a data structure on n points with

- *size $O(n)$*
- *dominance counting query time $O\left(\frac{\log n}{\log \log n}\right)$*

Efficient output representation

Corollary

Semi-local LCS lengths can be represented in

- *size $O(n \log n)$ query time $O(\log^2 n)$*
- *size $O(n)$ query time $O(\frac{\log n}{\log \log n})$ in the RAM model*

- 1 Semi-local string comparison
- 2 Efficient output representation
- 3 Fast block iteration and applications**
- 4 Fast divide-and-conquer and applications
- 5 Further algorithmic ideas and applications
- 6 Conclusions and future work

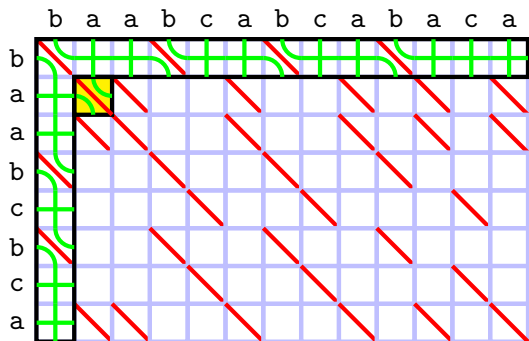
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (in new notation)

Iterate over alignment graph, tracing critical curves. Each pair of critical curves is allowed to cross at most once.

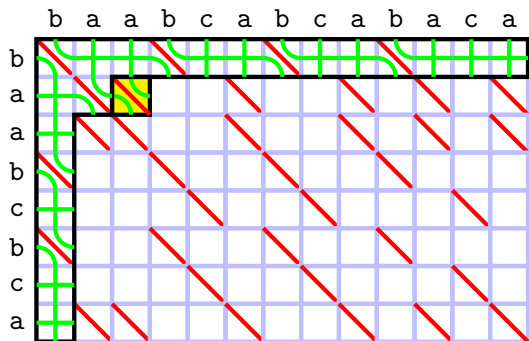
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



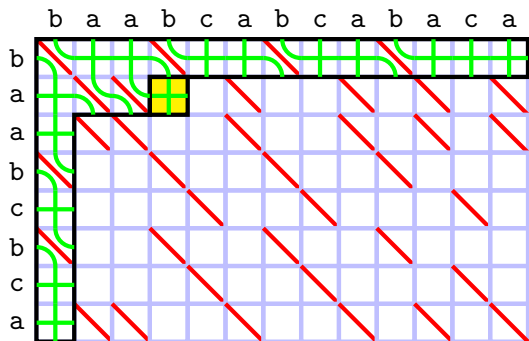
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



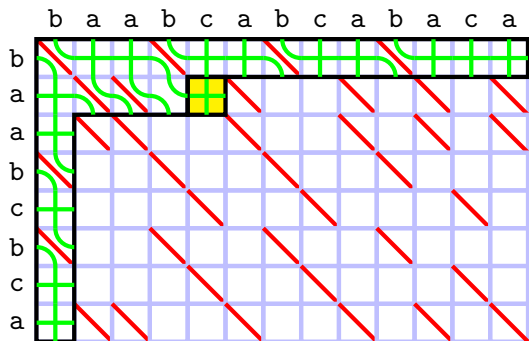
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



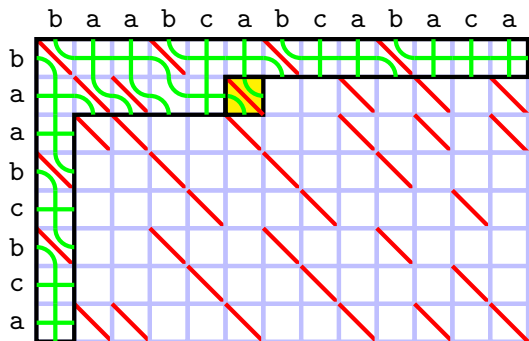
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



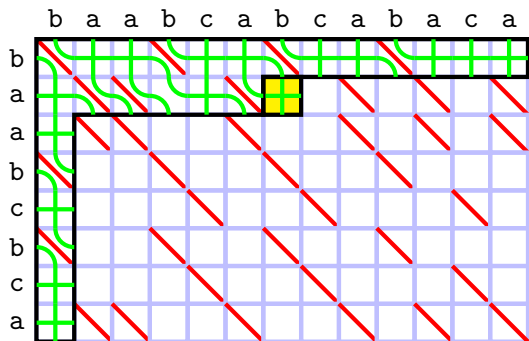
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



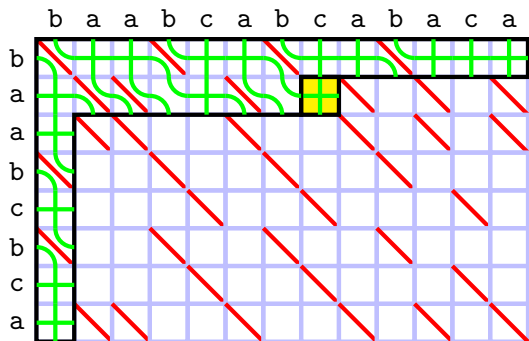
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



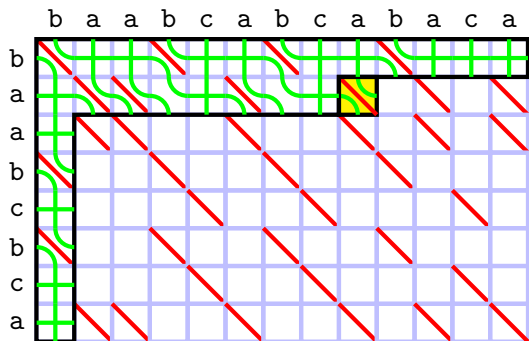
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



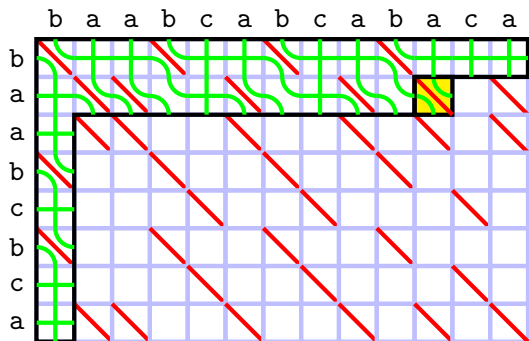
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



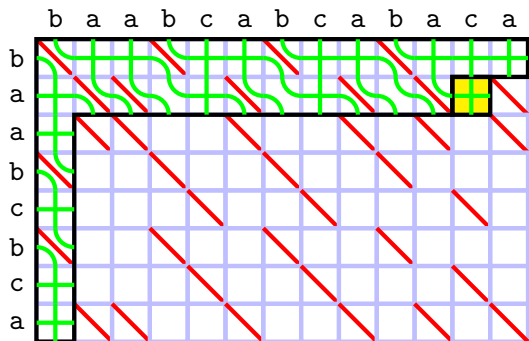
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



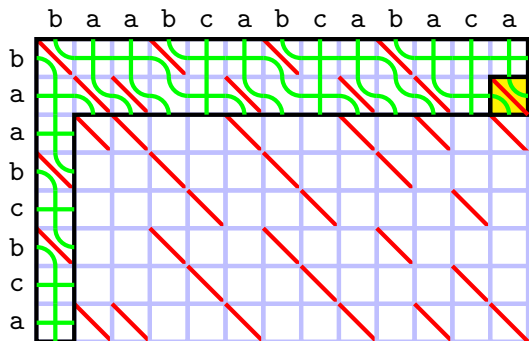
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



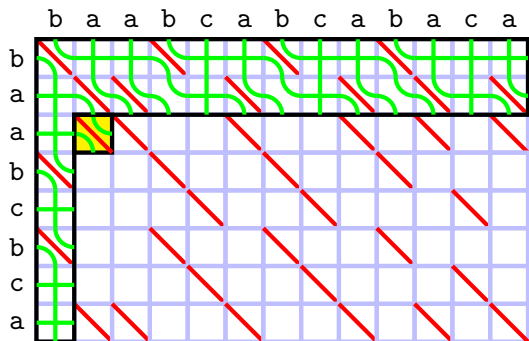
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



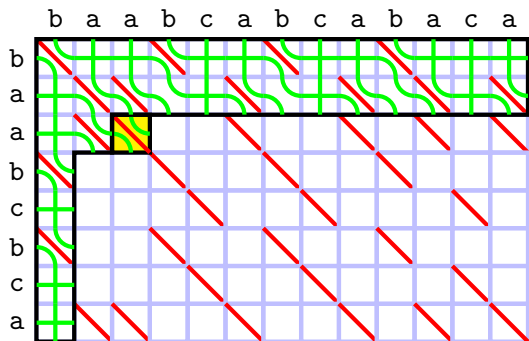
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



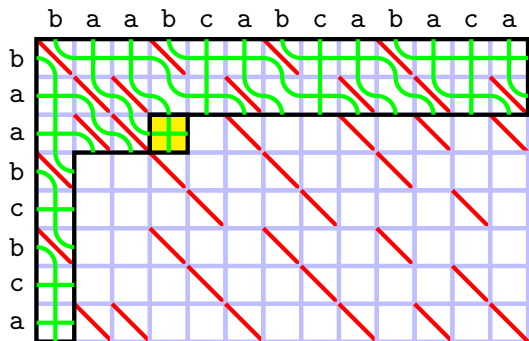
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



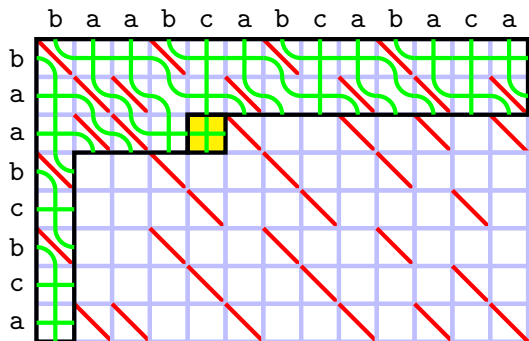
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



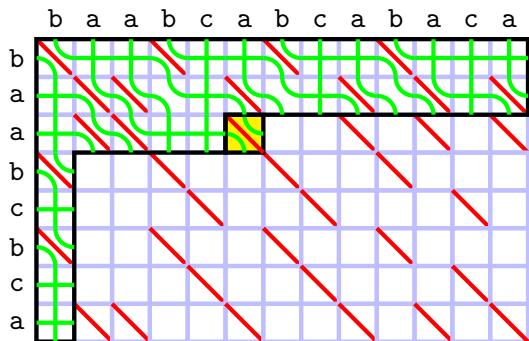
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



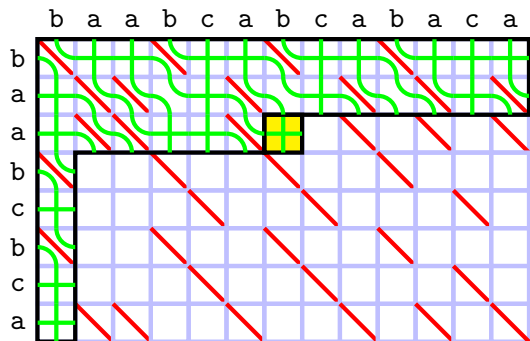
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



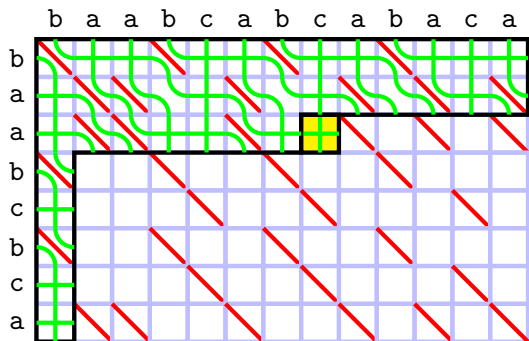
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



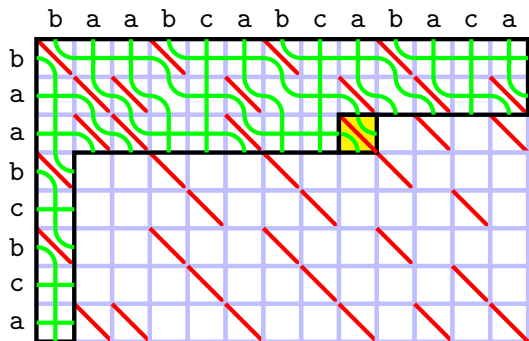
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



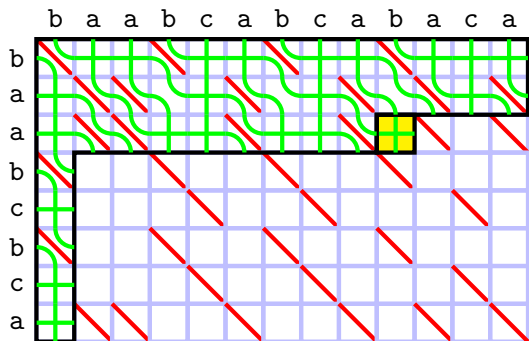
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



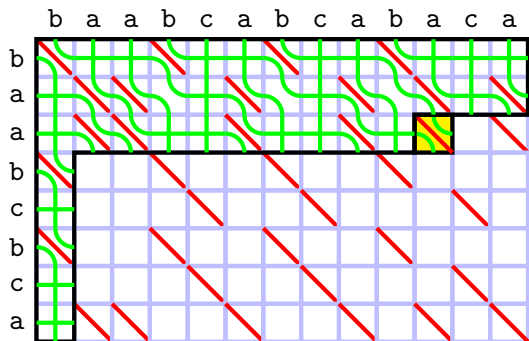
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



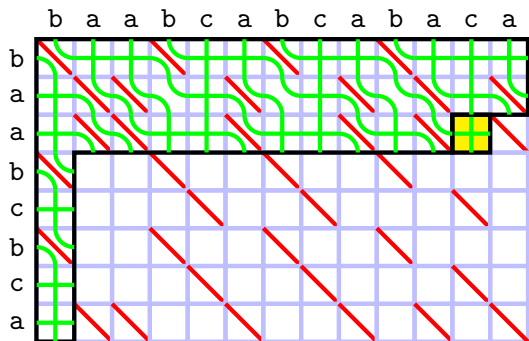
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



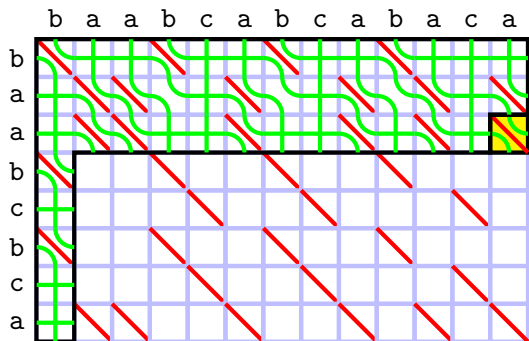
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



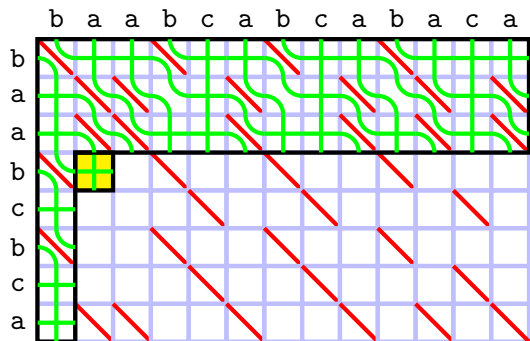
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



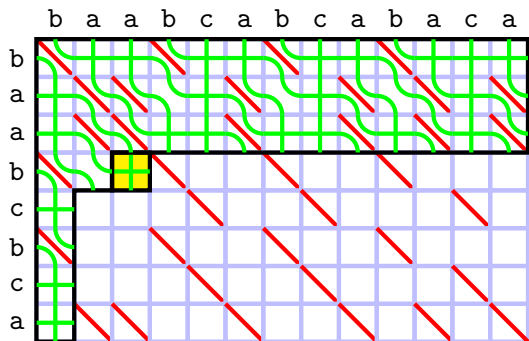
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



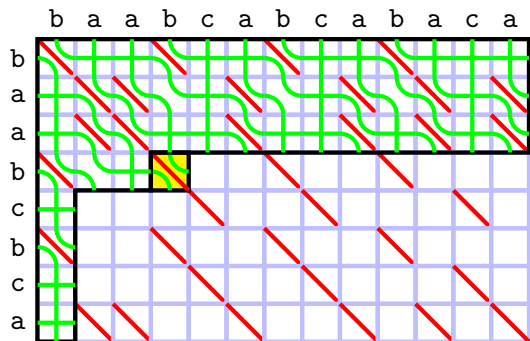
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



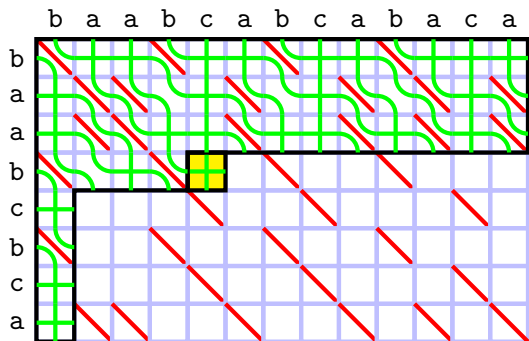
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



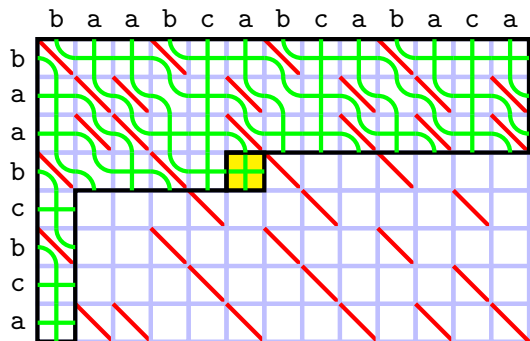
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



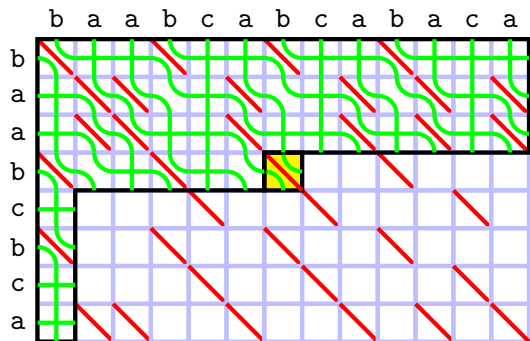
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



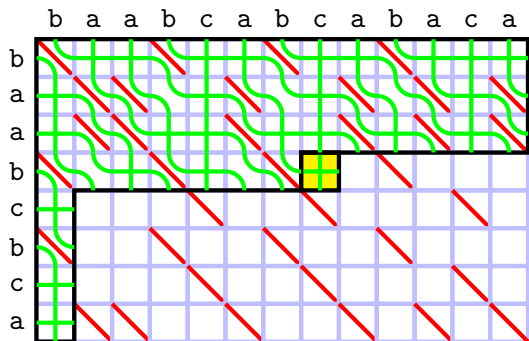
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



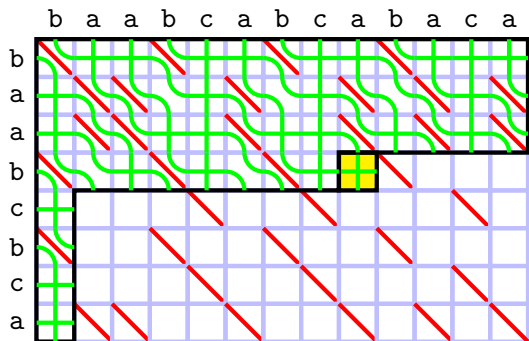
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



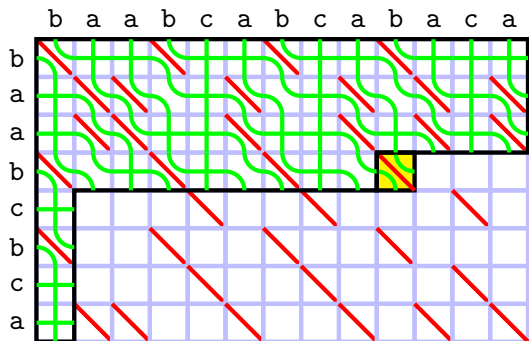
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



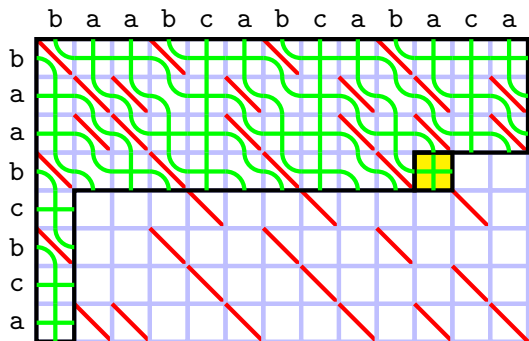
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



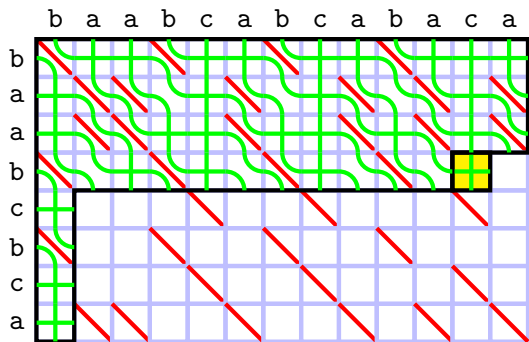
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



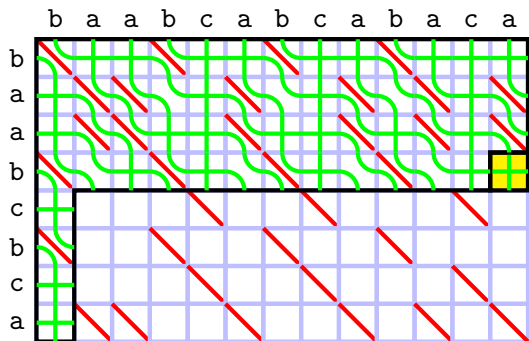
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



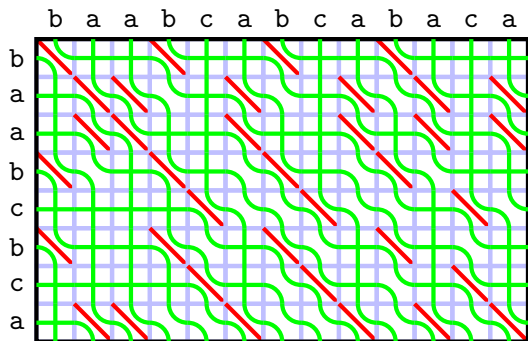
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



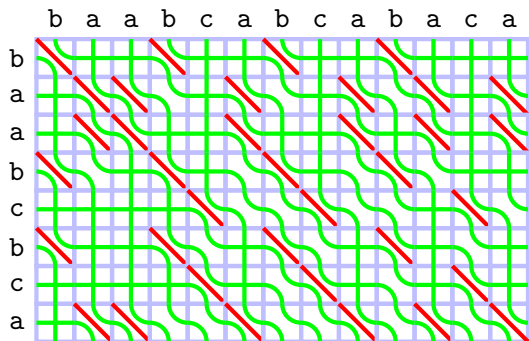
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



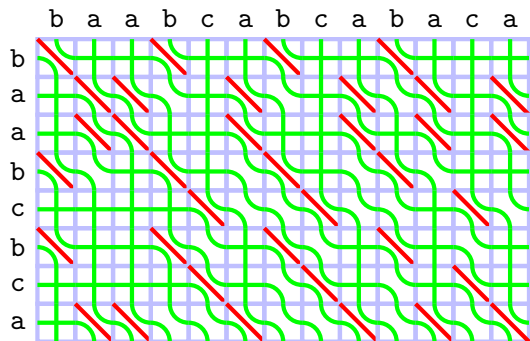
Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



Fast block iteration and applications

Semi-local LCS by Schmidt/Alves+ (contd.)



Time $O(mn)$

Fast block iteration and applications

Theorem (new)

Semi-local LCS can be computed in time $O\left(\frac{mn \log \log n}{\log n}\right)$ and memory $O(n)$

Must assume m, n are reasonably close: $\frac{\log n}{\log \log n} \leq m \leq n$

Fast block iteration and applications

Theorem (new)

Semi-local LCS can be computed in time $O\left(\frac{mn \log \log n}{\log n}\right)$ and memory $O(n)$

Must assume m, n are reasonably close: $\frac{\log n}{\log \log n} \leq m \leq n$

Proof: Iterate over alignment graph in small blocks, tracing critical curves

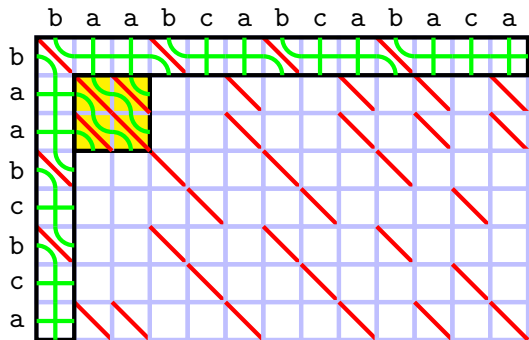
Classical technique by Arlazarov+: when blocks sufficiently small, better to precompute *all possible* block transitions in advance

Threshold block size $t = \frac{\log n}{4 \log \log n}$

For each of at most $(t!)^2$ possible block types, precompute the mapping of $(t!)^2$ possible inputs to $(t!)^2$ possible outputs □

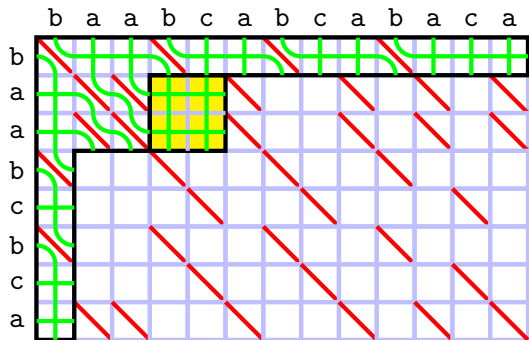
Fast block iteration and applications

Semi-local LCS: the new algorithm



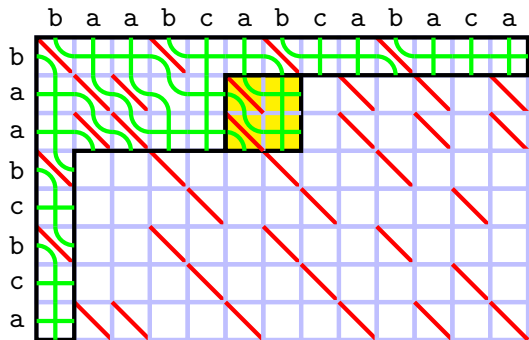
Fast block iteration and applications

Semi-local LCS: the new algorithm



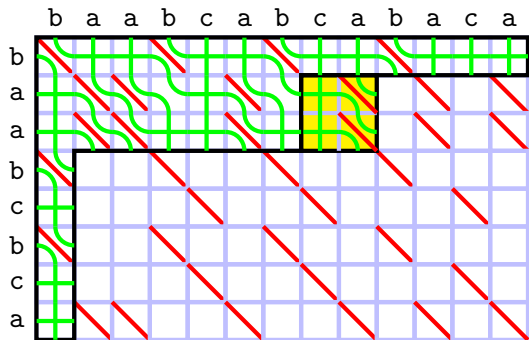
Fast block iteration and applications

Semi-local LCS: the new algorithm



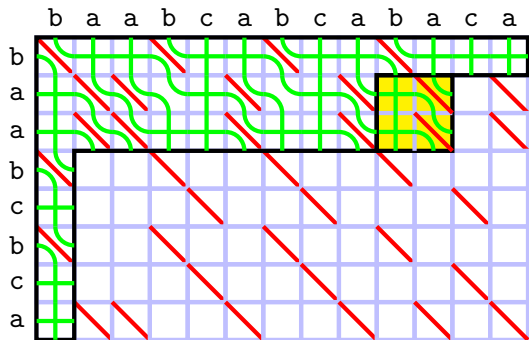
Fast block iteration and applications

Semi-local LCS: the new algorithm



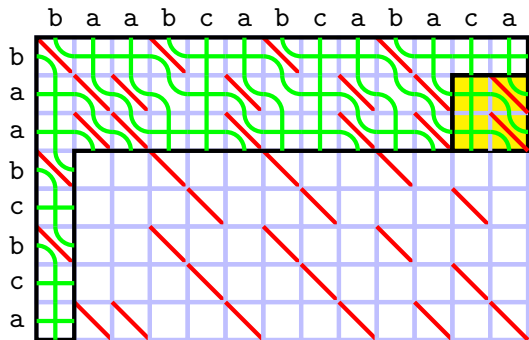
Fast block iteration and applications

Semi-local LCS: the new algorithm



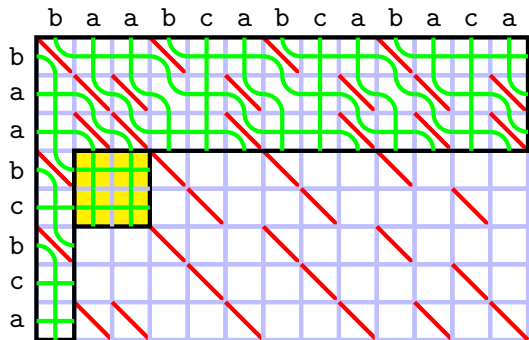
Fast block iteration and applications

Semi-local LCS: the new algorithm



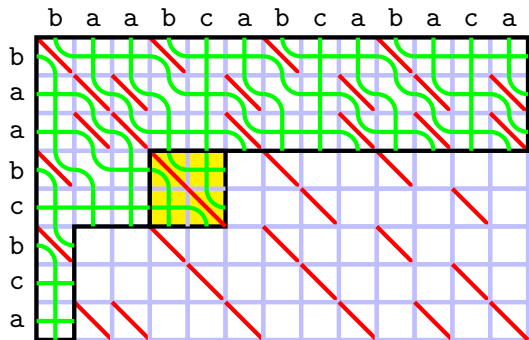
Fast block iteration and applications

Semi-local LCS: the new algorithm



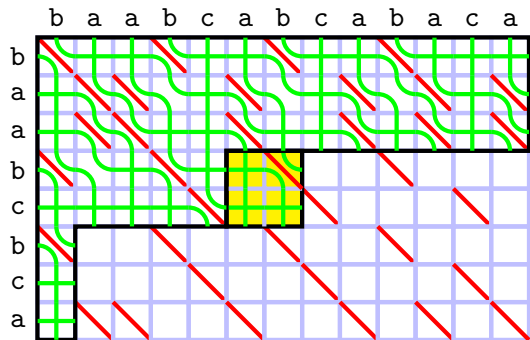
Fast block iteration and applications

Semi-local LCS: the new algorithm



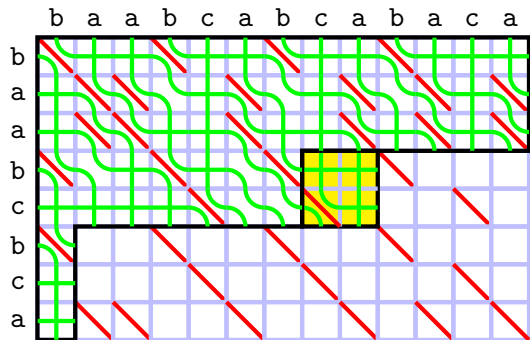
Fast block iteration and applications

Semi-local LCS: the new algorithm



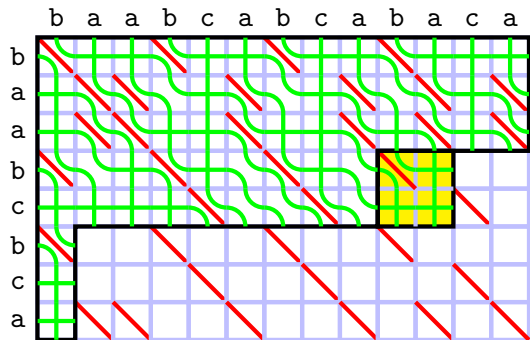
Fast block iteration and applications

Semi-local LCS: the new algorithm



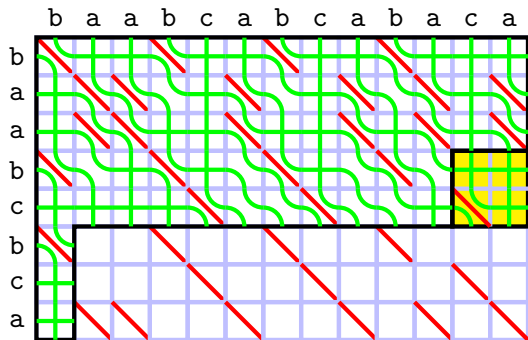
Fast block iteration and applications

Semi-local LCS: the new algorithm



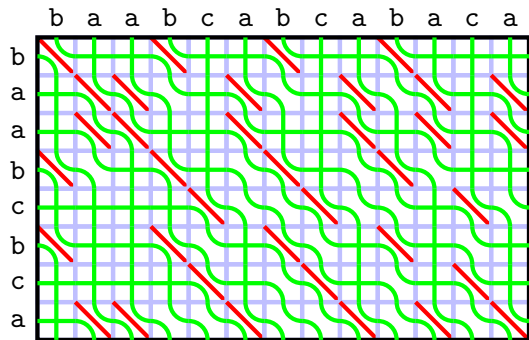
Fast block iteration and applications

Semi-local LCS: the new algorithm



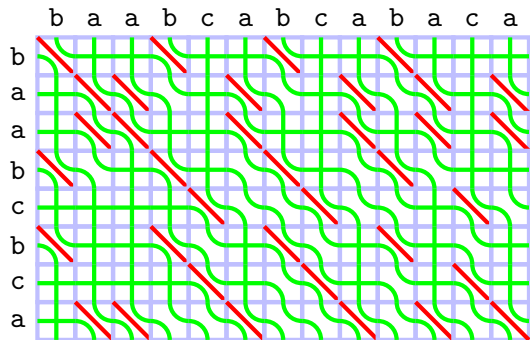
Fast block iteration and applications

Semi-local LCS: the new algorithm



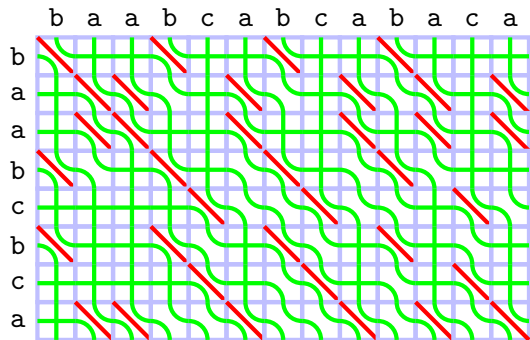
Fast block iteration and applications

Semi-local LCS: the new algorithm



Fast block iteration and applications

Semi-local LCS: the new algorithm



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

Fast block iteration and applications

Application: *LCS problem on cyclic strings*

Determine the maximum LCS length for string a against all cyclic rotations of b

Used in handwriting recognition

Fast block iteration and applications

Application: *LCS problem on cyclic strings*

Determine the maximum LCS length for string a against all cyclic rotations of b

Used in handwriting recognition

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

$O(mn \log m)$ [Maes, 1990]

$O(mn)$ [Bunke, Bühler, 1993]

also [Landau+, 1998], [Schmidt, 1998]

Fast block iteration and applications

Application: *LCS problem on cyclic strings*

Determine the maximum LCS length for string a against all cyclic rotations of b

Used in handwriting recognition

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

$O(mn \log m)$ [Maes, 1990]

$O(mn)$ [Bunke, Bühler, 1993]

also [Landau+, 1998], [Schmidt, 1998]

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

Run semi-local LCS on a against bb , then perform n string-substring LCS queries

Fast block iteration and applications

Application: the *longest repeated subsequence* problem

Given string a of length n , determine its longest subsequence that is a square (i.e. a concatenation of two identical strings)

Related to “tandem repeats” in genome

Fast block iteration and applications

Application: the *longest repeated subsequence* problem

Given string a of length n , determine its longest subsequence that is a square (i.e. a concatenation of two identical strings)

Related to “tandem repeats” in genome

$O(n^3)$ (somewhat) naive

$O(n^2)$ [Kosowski, 2004]

Fast block iteration and applications

Application: the *longest repeated subsequence* problem

Given string a of length n , determine its longest subsequence that is a square (i.e. a concatenation of two identical strings)

Related to “tandem repeats” in genome

$O(n^3)$ (somewhat) naive

$O(n^2)$ [Kosowski, 2004]

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

Run semi-local LCS on a against itself, then perform n suffix-prefix LCS queries

- 1 Semi-local string comparison
- 2 Efficient output representation
- 3 Fast block iteration and applications
- 4 Fast divide-and-conquer and applications**
- 5 Further algorithmic ideas and applications
- 6 Conclusions and future work

Fast divide-and-conquer and applications

Computing highest-score matrices by divide-and-conquer

“Divide”: partitioning the alignment dag recursively into strips or square blocks

“Conquer”: ($max, +$) highest-score matrix multiplication

Fast divide-and-conquer and applications

Computing highest-score matrices by divide-and-conquer

“Divide”: partitioning the alignment dag recursively into strips or square blocks

“Conquer”: $(max, +)$ highest-score matrix multiplication

General matrices: time $O(n^3)$

Monge (= planar distance) matrices: time $O(n^2)$

Implicit highest-score matrices: time $O(n^{1.5})$ [T, 2006]

Fast divide-and-conquer and applications

Theorem (T, 2006)

Let A, B be implicit highest-score matrices of size n . The $(\max, +)$ product $AB = C$ can be computed in time $O(n^{1.5})$.

Fast divide-and-conquer and applications

Theorem (T, 2006)

Let A, B be implicit highest-score matrices of size n . The $(\max, +)$ product $AB = C$ can be computed in time $O(n^{1.5})$.

Proof: divide-and-conquer on the product C

Obtain counts of C -critical points in square blocks. If (and only if!) the block has at least one C -critical point, recurse into half-sized subblocks.

Crucial observation: for a block of size r , only need to keep $O(r)$ data from A, B , and to perform $O(r)$ work

Fast divide-and-conquer and applications

Proof (contd.)

In the divide-and-conquer tree

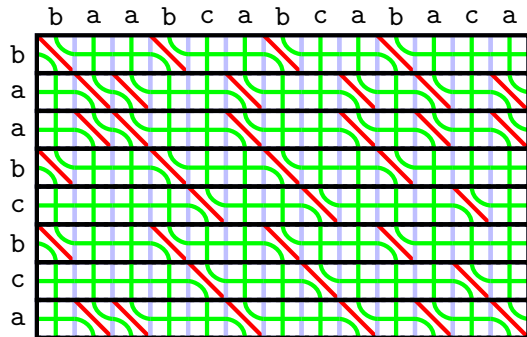
- root: one block of size n , work $O(n)$
- middle level: at most n blocks of size $n^{0.5}$, work $O(n \cdot n^{0.5}) = O(n^{1.5})$
- leaves: at most n blocks of size 1, work $O(n)$

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



Fast divide-and-conquer and applications

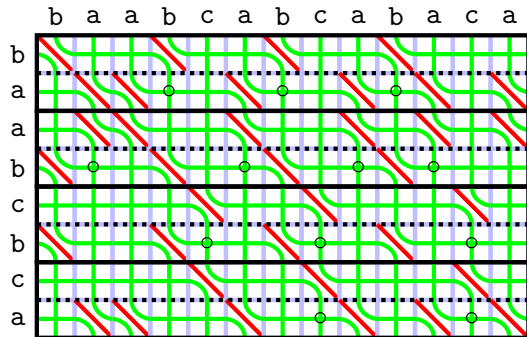
Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = *critical*

Fast divide-and-conquer and applications

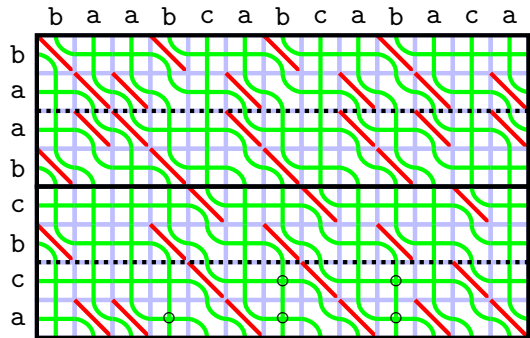
Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = critical

Fast divide-and-conquer and applications

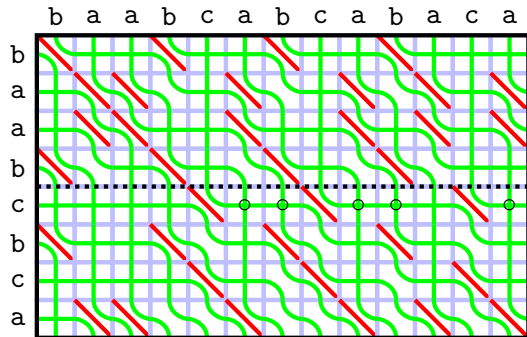
Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = *critical*

Fast divide-and-conquer and applications

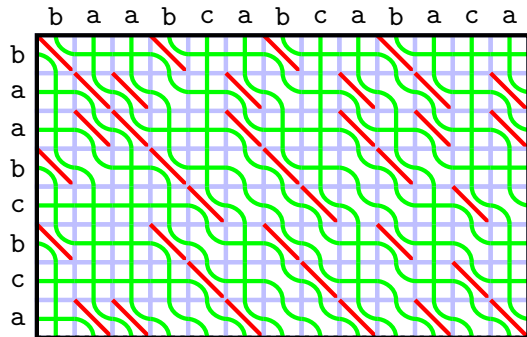
Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = critical

Fast divide-and-conquer and applications

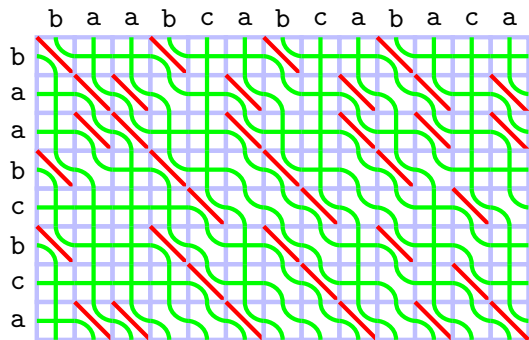
Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = *critical*

Fast divide-and-conquer and applications

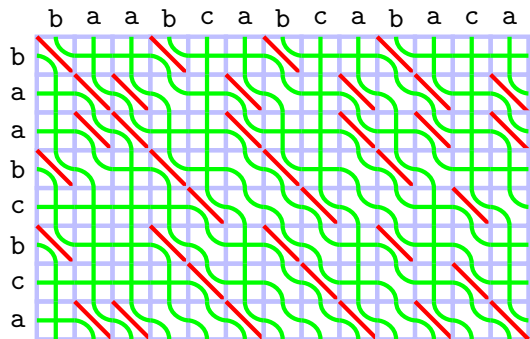
Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = *critical*

Fast divide-and-conquer and applications

Alternative method for semi-local LCS: divide-and-conquer on strips, each “conquer” runs in time $O(n^{1.5})$



blue = 0
red = 1
green = critical

Time $O(mn)$, but can be efficiently adapted to special cases

Fast divide-and-conquer and applications

Theorem (T, 2006)

Semi-local LCS on permutations of length n can be computed in time $O(n^{1.5})$ and memory $O(n)$

Fast divide-and-conquer and applications

Theorem (T, 2006)

Semi-local LCS on permutations of length n can be computed in time $O(n^{1.5})$ and memory $O(n)$

Proof: in a strip of height k , at most k critical curves non-trivial

Two such strips can be “conquered” in time $O(k^{1.5})$

In every recursion level:

- number of subproblems goes up by a factor of 2
- time per subproblem goes down by a factor of $2^{1.5}$

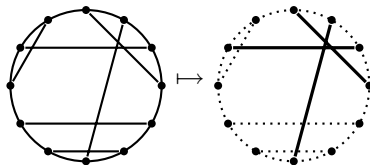
Hence, the top level dominates with time $O(n^{1.5})$



Fast divide-and-conquer and applications

Application: the *maximum clique problem in a circle graph*

Given a circle with n chords, determine the maximum-size subset of pairwise intersecting chords



Fast divide-and-conquer and applications

Application: the *maximum clique problem in a circle graph* (contd.)

$\exp(n)$ naive

$O(n^3)$ [Gavril, 1973]

$O(n^2)$ [Rotem, Urrutia, 1981]

also [Hsu, 1985], [Masuda+, 1990], [Apostolico+, 1992]

$O(n^{1.5})$ NEW

Finds (implicitly) all *maximal* cliques

Fast divide-and-conquer and applications

Application: the *maximum clique problem in a circle graph* (contd.)

Standard reduction to an *interval model*: cut the circle and lay it out on the line; chords become intervals

The interval model can be represented by permutation a of size $2n$

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

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

Fast divide-and-conquer and applications

Application: the *maximum clique problem in a circle graph* (contd.)

Algorithm idea: check all $2n + 1$ possible common intersection points

For each candidate intersection point, need to find the maximum subset of covering segments without pairwise containment

Equivalent to prefix-suffix LCS on permutations a, id

Run semi-local LCS on a, id and build range tree: time $O(n^{1.5})$

Query prefix-suffix LCS for each candidate intersection point: time $(2n + 1) \cdot O(\log^2 n) = O(n \log^2 n)$

Overall time $O(n^{1.5}) + O(n \log^2 n) = O(n^{1.5})$

- 1 Semi-local string comparison
- 2 Efficient output representation
- 3 Fast block iteration and applications
- 4 Fast divide-and-conquer and applications
- 5 Further algorithmic ideas and applications**
- 6 Conclusions and future work

Further algorithmic ideas and applications

Suppose $m \gg n$, so the alignment graph is very “tall and thin”

The *partial implicit highest-score matrix*: top-to-bottom, top-to-right, left-to-bottom critical curves

Overall $O(n)$ data, ignores $O(m)$ left-to-right critical curves

Allows string-substring, prefix-suffix and suffix-prefix (but not substring-string) LCS queries

Still sufficient for divide-and-conquer on strips, “conquer” time $O(n^{1.5})$

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text*

Text compression model: a straight-line program (SLP, context-free grammar) generating the text by m assignments of the form

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

Let $T = T_m$. Denote the uncompressed size of T by M .

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

Note M can be $O(c^m)$. Assume address arithmetic on T still $O(1)$.

Pattern string P is given explicitly

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Global subsequence recognition: does a contain b as a subsequence?

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Global subsequence recognition: does a contain b as a subsequence?

On an uncompressed text:

$O(M)$

greedy

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Global subsequence recognition: does a contain b as a subsequence?

On an uncompressed text:

$O(M)$ greedy

On an SLP-compressed text:

$O(mn)$ greedy

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Local subsequence recognition: determine the number of minimal substrings of a containing b as a subsequence

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Local subsequence recognition: determine the number of minimal substrings of a containing b as a subsequence

On an uncompressed text:

$O(Mn)$ [Mannila+, 1995]

$O\left(\frac{Mn}{\log n}\right)$ [Das+, 1997]

$O(M + c^n)$ [Boasson+, 2001]

$O(M\sigma + n)$ [Troniček, 2001]

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Local subsequence recognition: determine the number of minimal substrings of a containing b as a subsequence

On an uncompressed text:

$O(Mn)$ [Mannila+, 1995]

$O\left(\frac{Mn}{\log n}\right)$ [Das+, 1997]

$O(M + c^n)$ [Boasson+, 2001]

$O(M\sigma + n)$ [Troniček, 2001]

On an SLP-compressed text:

$O(mn^2 \log n)$ [Cégielski+, 2006]

$O(mn^{1.5})$ NEW

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Algorithm idea: build a partial implicit highest-score matrix for every T_r by divide-and-conquer in time $O(mn^{1.5})$

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Algorithm idea: build a partial implicit highest-score matrix for every T_r by divide-and-conquer in time $O(mn^{1.5})$

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

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

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Algorithm idea: build a partial implicit highest-score matrix for every T_r by divide-and-conquer in time $O(mn^{1.5})$

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

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

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

Further algorithmic ideas and applications

Application: *subsequence recognition in compressed text* (contd.)

Algorithm idea: build a partial implicit highest-score matrix for every T_r by divide-and-conquer in time $O(mn^{1.5})$

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

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

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

Overall time $O(mn^{1.5}) + O(mn \log^2 n) = O(mn^{1.5})$

Further algorithmic ideas and applications

More efficient divide-and-conquer on strips

Suppose a strip is “short and fat”: for such a strip, $m \ll n$

The “conquer” time can be improved from $n^{1.5}$ to $mn^{0.5}$, even if the counterpart strip being “conquered” is tall!

Further algorithmic ideas and applications

More efficient divide-and-conquer on strips

Suppose a strip is “short and fat”: for such a strip, $m \ll n$

The “conquer” time can be improved from $n^{1.5}$ to $mn^{0.5}$, even if the counterpart strip being “conquered” is tall!

The *quasi-local LCS problem*

Given m overlapping *prescribed substrings* in string a , solve the semi-local LCS problem for every prescribed substring against b

Further algorithmic ideas and applications

More efficient divide-and-conquer on strips

Suppose a strip is “short and fat”: for such a strip, $m \ll n$

The “conquer” time can be improved from $n^{1.5}$ to $mn^{0.5}$, even if the counterpart strip being “conquered” is tall!

The *quasi-local LCS problem*

Given m overlapping *prescribed substrings* in string a , solve the semi-local LCS problem for every prescribed substring against b

$O(m^2 n)$ naive

$O(m^{1.25} n)$ NEW

Further algorithmic ideas and applications

The *sparse spliced alignment* problem

Given m overlapping *prescribed substrings* in string a , determine the chain of non-overlapping prescribed substrings giving the highest LCS score against string b

Describes assembly of an unknown genome from candidate exons, given a known similar genome

Further algorithmic ideas and applications

The *sparse spliced alignment* problem

Given m overlapping *prescribed substrings* in string a , determine the chain of non-overlapping prescribed substrings giving the highest LCS score against string b

Describes assembly of an unknown genome from candidate exons, given a known similar genome

Assume $m = n$

$O(n^3)$ [Gelfand+, 1996]

$O(n^{2.5})$ [Kent+, 2006]

$O(n^{2.25})$ NEW

- 1 Semi-local string comparison
- 2 Efficient output representation
- 3 Fast block iteration and applications
- 4 Fast divide-and-conquer and applications
- 5 Further algorithmic ideas and applications
- 6 Conclusions and future work**

Conclusions and future work

Have given efficient algorithms for

- semi-local LCS (output represented implicitly)
- semi-local LCS on permutations
- related techniques

Conclusions and future work

Have given efficient algorithms for

- semi-local LCS (output represented implicitly)
- semi-local LCS on permutations
- related techniques

Used as an “algorithmic plug-in”, leads to improvements in

- cyclic LCS
- longest repeated subsequence
- maximum clique in a circle graph
- local subsequence recognition in compressed text
- sparse spliced alignment

Conclusions and future work

Have given efficient algorithms for

- semi-local LCS (output represented implicitly)
- semi-local LCS on permutations
- related techniques

Used as an “algorithmic plug-in”, leads to improvements in

- cyclic LCS
- longest repeated subsequence
- maximum clique in a circle graph
- local subsequence recognition in compressed text
- sparse spliced alignment

Potential further improvements:

- extension to real-weighted edit scores
- new interesting applications

References I



A. Tiskin.

All semi-local longest common subsequences in subquadratic time.

In *Proceedings of CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 352–363, 2006.



A. Tiskin.

Longest common subsequences in permutations and maximum cliques in circle graphs.

In *Proceedings of CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 271–282, 2006.