String comparison by transposition networks

Peter Krusche

(joint work with Alexander Tiskin)

Strings, substrings and subsequences. . .

Exact string search

Find pattern ***abab*** occurring as substring in text *bbbababababba* $\Rightarrow$

*bbb**abab**abba*
*bbbab**abab**ba*

Possible in $O(n)$ time (Automata, Boyer Moore, Knuth Morris Pratt)

Exact string search

Find pattern ***abab*** occurring as substring in text *bbbababababba* $\Rightarrow$

*bbb**abab**abba*
*bbbab**abab**ba*

Possible in $O(n)$ time (Automata, Boyer Moore, Knuth Morris Pratt)

Exact string comparison: Hamming distance

Count mismatches:

$$\text{dist}(bbbabababbba,\ \textcolor{red}{a}bb\textcolor{red}{b}baba\textcolor{red}{a}ba) = 3$$

Approximate string search

One way: subsequence matching

Find pattern ***abab*** in text *bbbababababba* as a subsequence ⇒

$$\vdots$$

*bbb****aba****ba****b****ba*

*bbb****ab****ab****ab****ba*

$$\vdots$$

Possible in $O(n)$ time by constructing automata.

Approximate string search

One way: subsequence matching

Find pattern ***abab*** in text *bbbabababba* as a subsequence $\Rightarrow$

$$\ldots$$

*bbb**aba**ba**b**ba*

*bbb**ab**ab**ab**ba*

$$\ldots$$

Possible in $O(n)$ time by constructing automata.

Approximate comparison: string alignment

Align the maximum number of letters, preserving order:

abbabbbabbaba

bbabaabbba

The aligned letters form the longest common subsequence (LCS); length of this sequence: LLCS.

$$\mathrm{dist}(x,y) = m + n - 2\,\mathrm{LLCS}(x,y)$$

Approximate comparison: string alignment

Align the maximum number of letters, preserving order:

a**bbab**bb**abb**a**ba**

bbaba**abbba**

The aligned letters form the longest common subsequence (LCS); length of this sequence: LLCS.

dist(x,y) = m + n - 2 LLCS(x,y)

Approximate comparison

LCS distance: Minimizes number of insertions/deletions to get from string x to string y.

Extensions we don't (directly) consider here:

- Edit distance: minimize the number of insertions, deletions, and exchange operations to get from one string to the other.
- Weighted case: assign weights to each operation on each pair of characters.

The LCS Problem

Complexities of classical solutions (input strings of length n, r matches, d dominant matches):

- Dynamic programming
  (Wagner & Fischer, '74) : $O(n^2)$
- ⇒ Using "Four Russians" technique
  (Masek & Paterson, '80) : $O(n^2/\log n)$
- Dominant match based
  (Hunt & Szymanski, '77) : $O((r+n)\log n)$
  (Apostolico & Guerra, '87) : $O(m \log n + d \log(mn/d))$

The LCS Problem

Complexities of classical solutions (input strings of length n, r matches, d dominant matches):

- Dynamic programming
  (Wagner & Fischer, '74) : $O(n^2)$

$\Rightarrow$ Using "Four Russians" technique
  (Masek & Paterson, '80) : $O(n^2/\log n)$

- Dominant match based
  (Hunt & Szymanski, '77) : $O((r+n)\log n)$
  (Apostolico & Guerra, '87) : $O(m\log n + d\log(mn/d))$

Seaweeds and networks. . .

What is semi-local string comparison?

Semi-local comparison: compute substring-string *highest-score matrix*

$$A(i, j) = \mathsf{LLCS}(x_i x_{i+1} \ldots x_j, y)$$

and simultaneously all string-substring, prefix-suffix and suffix-prefix LLCS.

Why compare semi-locally?

- Two $N \times N$ highest-score matrices can be $(\text{min}, +)$ multiplied in $O(N^{1.5})$

$\Rightarrow$ Semi-local comparison is useful for obtaining efficient parallel LCS algorithms

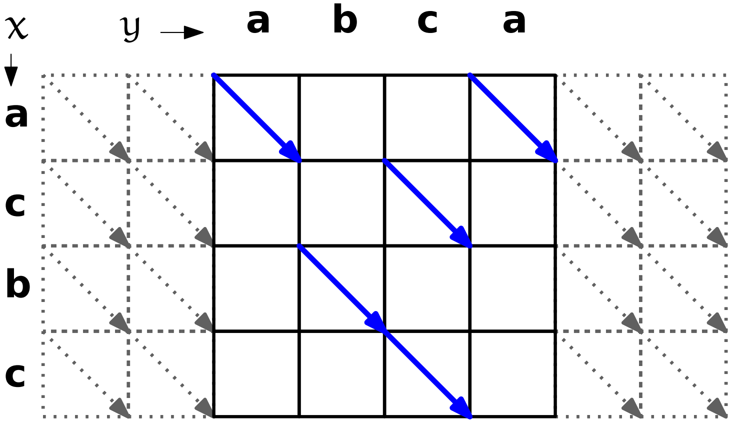
- Semi-local comparison has non-trivial algorithmic applications itself.
- One step closer to fully local comparison (substring vs. substring)

How to do it:

Use the Seaweed Algorithm, which runs in $O(n^2)$

Seaweed algorithm

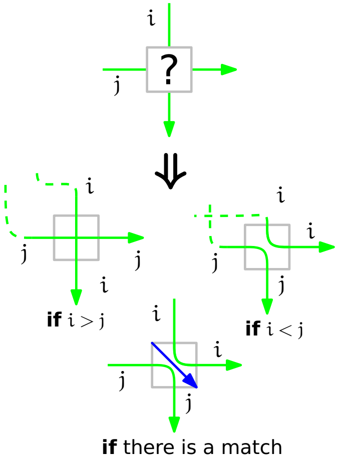
Start with *extended alignment-dag*:



Drawing this dag partitions the plane into cells.

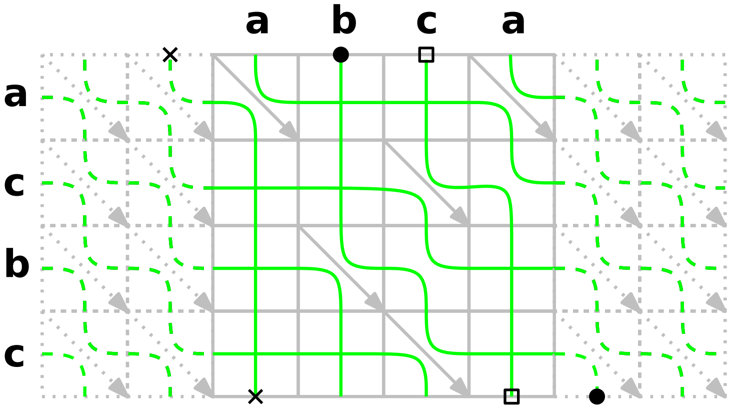
Seaweed algorithm

- We trace seaweeds through cells
- Two seaweeds cross at most once
- We are interested in start and end points of seaweeds



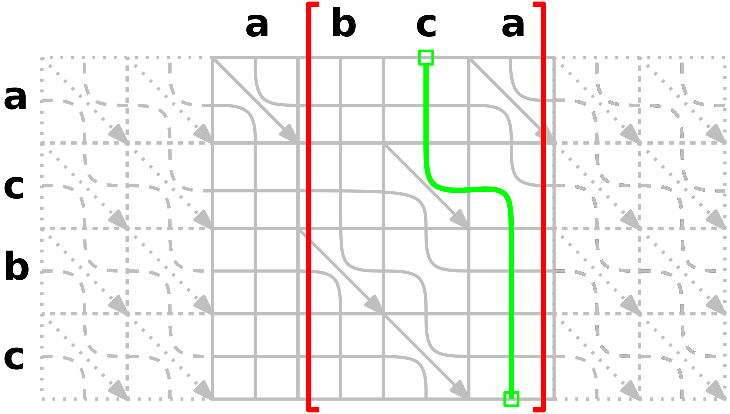
Seaweed algorithm

Querying the LCS distance by counting seaweeds:



Seaweed algorithm

Querying the LCS distance by counting seaweeds:



Comparison networks

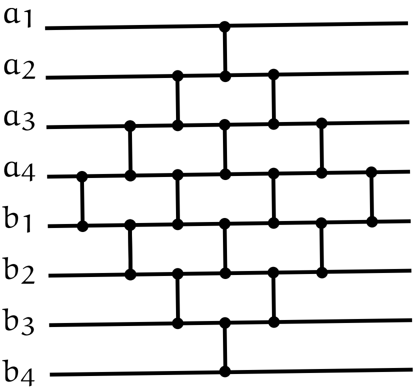
Comparison network:

- n wires connected by arbitrary number of *comparators*
- Comparators have two inputs and two outputs.
  - $\Rightarrow$ return larger value the predefined output
  - $\Rightarrow$ return smaller value at the other output.
- Traditional method for studying oblivious sorting algorithms

Transposition network: all comparators only connect adjacent wires.

Merging using a transposition network

Example (Merging two 4 element sorted sequences)

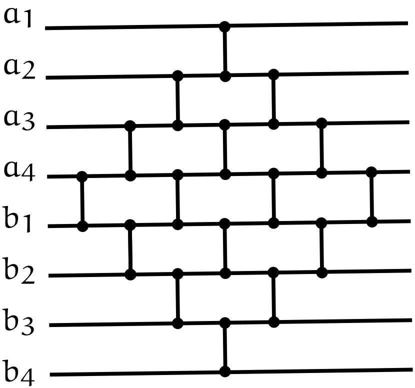


[Munter, U.S. Patent 5,216,420 ’93]

We call this a DIAMOND$(4, 4)$ **network.**

Merging using a transposition network

Example (Merging two 4 element sorted sequences)



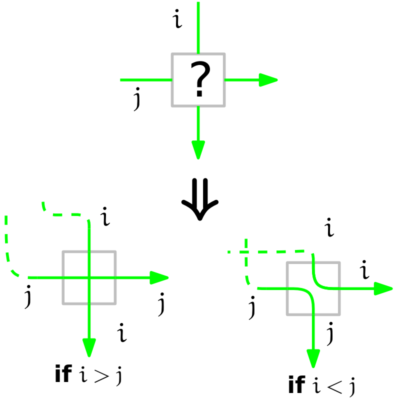
[Munter, U.S. Patent 5,216,420 ’93]

We call this a DIAMOND$(4, 4)$ network.

LCS and transposition networks

Observation

Every mismatch cell in the alignment dag behaves as a comparator.



Therefore

...we can define a transposition network to solve LCS problem.

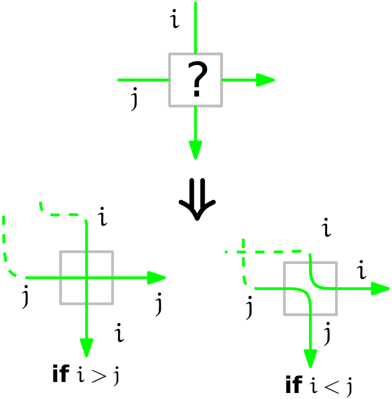
LCS and transposition networks

Observation

Every mismatch cell in the alignment dag behaves as a comparator.

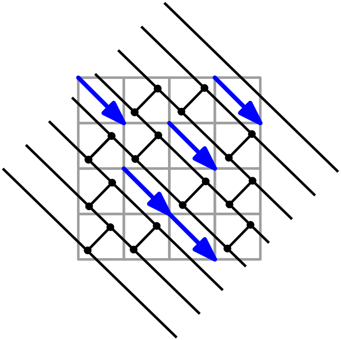
Therefore

. . . we can define a transposition network to solve LCS problem.



LCS and transposition networks

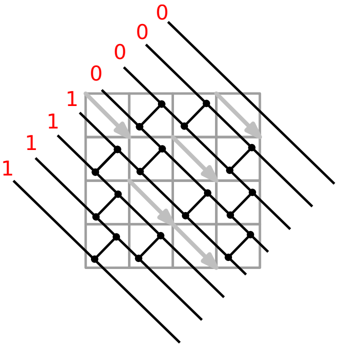
Definition (The LCSNET(x, y) network)



Remove all comparators in DIAMOND$(|x|, |y|)$ which correspond to matches.

LCS and transposition networks

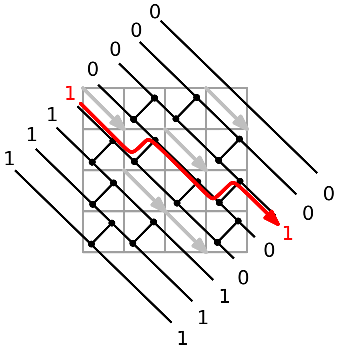
Definition (The LCSNET(x, y) network)



Initialize network with $|x|$ ones and $|y|$ zeros.

LCS and transposition networks

Definition (The LCSNET(x, y) network)



Count the number of ones which reach the right hand side $\Rightarrow$ LCS distance.

An application: Sparse string comparison...

Kinds of sparseness

1. Few match cells in the alignment dag

   Extreme case: comparison of permutation strings

2. Strings very similar (LCS is long / edit distance is short)

   Not necessarily correlated with number of matches

3. Strings very dissimilar (LCS is short)

   Correlated with number of matches, but still different measure

Obtaining all matches

- We want to work match by match...

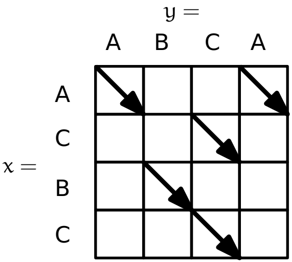
$\Rightarrow$ How to obtain list of matches in less than $O(n^2)$ time?

1. For small alphabets ($|\Sigma| < n$): in $O(n \log |\Sigma|)$ by counting character frequencies
2. For large alphabets: in $O(n \log n)$ by sorting one of the input strings and binary search

(if characters can only be tested for equality, $\Omega(n^2)$ is a lower bound)

Classical approach for sparse comparison

Trace antichains and their contours:

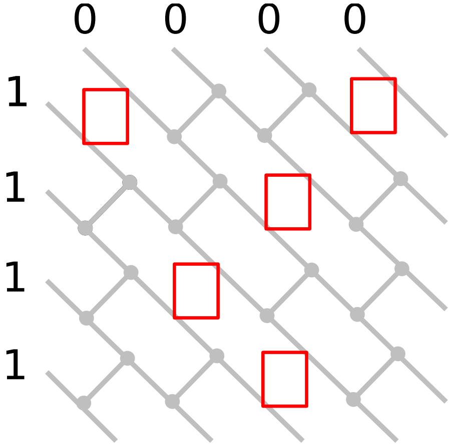


$$L = \begin{pmatrix} \boxed{1} & 1 & 1 & 1 \\ 1 & 1 & \boxed{2} & 2 \\ 1 & \boxed{2} & 2 & 2 \\ 1 & 2 & \boxed{3} & 3 \end{pmatrix}$$

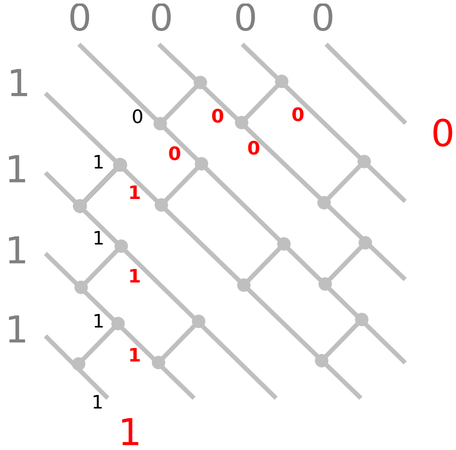
(a) input strings and alignment dag

(b) prefix-prefix LCS lengths and contours

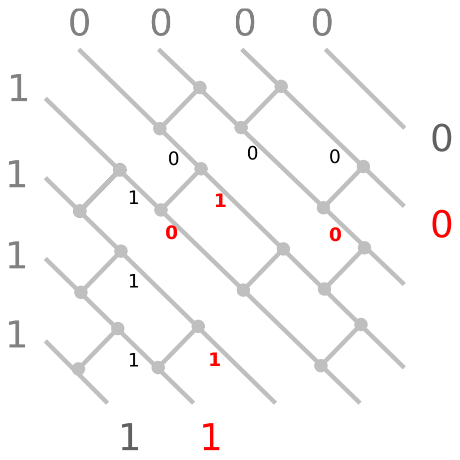
The same with 0-1 transposition networks



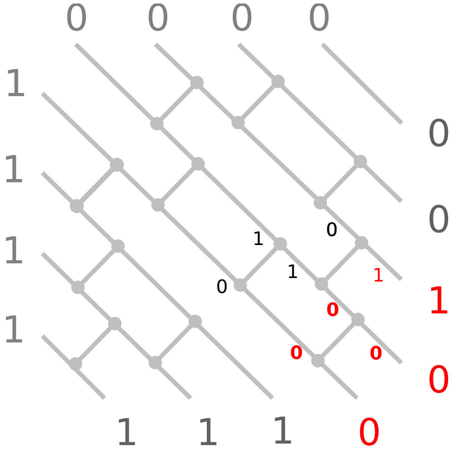
The same with 0-1 transposition networks



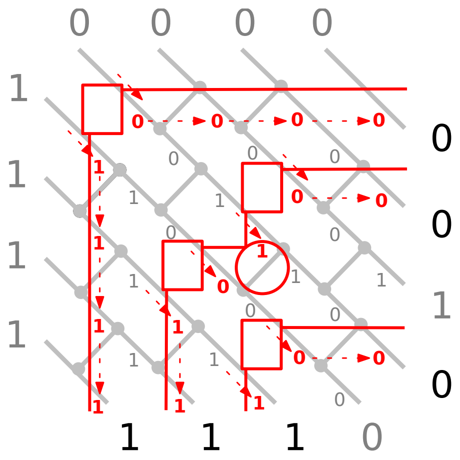
The same with 0-1 transposition networks



The same with 0-1 transposition networks

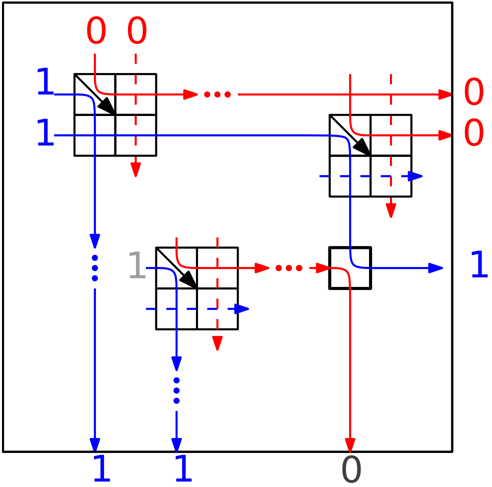


The same with 0-1 transposition networks



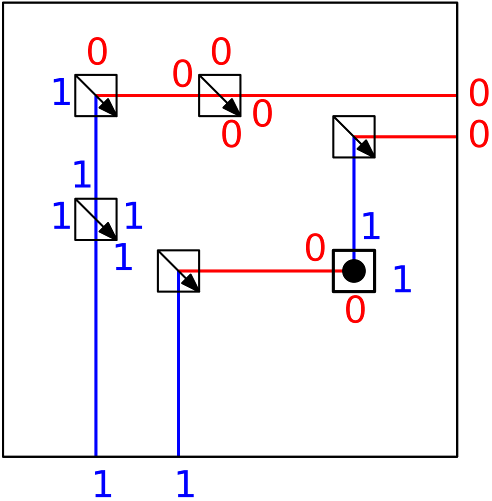
Classifying cells

Cells can be classified according to their 0-1 transposition network inputs:



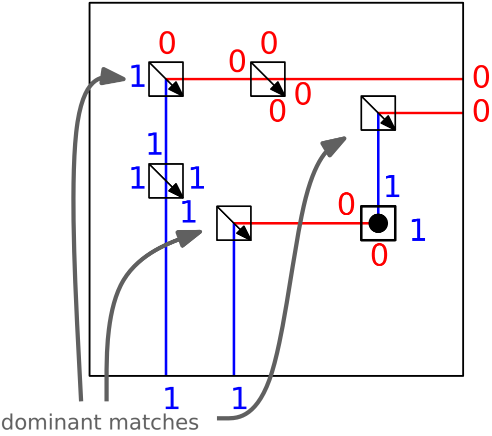
Classifying cells

Cells can be classified according to their 0-1 transposition network inputs:



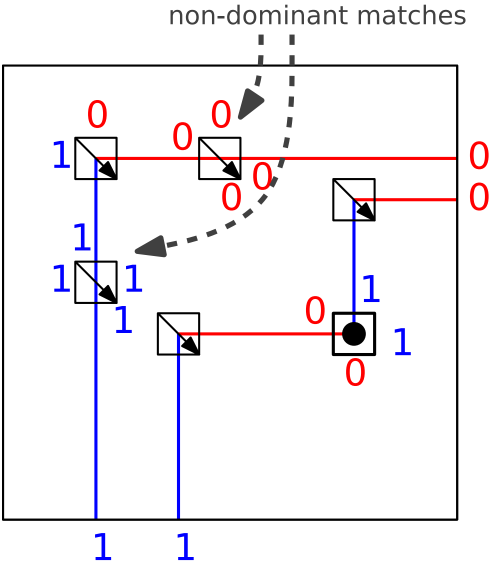
Classifying cells

Cells can be classified according to their 0-1 transposition network inputs:



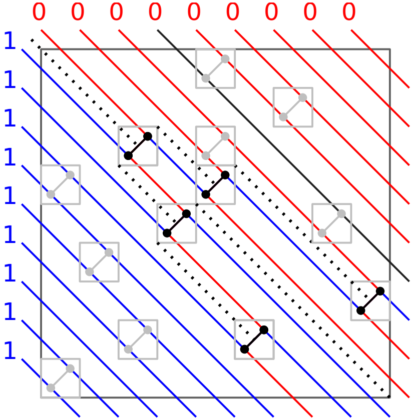
Classifying cells

Cells can be classified according to their 0-1 transposition network inputs:



More about cells

Not all cells in the 0-1 transposition network need to be evaluated:

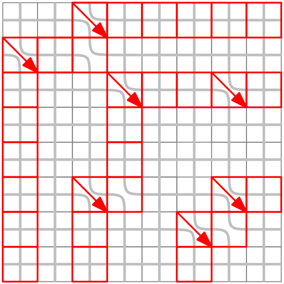


Parameterized LCS computation

We obtain a simple algorithm for computing the LCS in $O(p(n-p))$ by tracing zeros and ones (which is equivalent to the best known result).

Parameterized LCS computation

We can also obtain an algorithm for semi-local comparison which runs in $O(np)$:



Summary

- Transposition networks provide unified view on different LCS/semi-local comparison algorithms
- Transposition networks allow to derive parameterized algorithms for global LCS more easily than existing approaches.
- We have new parameterized algorithms for sparse semi-local string comparison running in $O(np + \textit{preproc}(n))$ and $O(n\sqrt{r} + \textit{preproc}(n))$

Outlook

Some further work:

- Generalize to other types of distances (edit distance).
- Apply techniques to LCS with non-overlapping inversions
- Another interesting problem: string comparison in streaming models (i.e. we can only read strings once/predefined number of times)

Thanks!

. . . any questions?

Bibliography

- Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals,.
  Sov. Phys. Dokl. **6** (1966) 707–710
- G. Navarro: A guided tour to approximate string matching.
  ACM Computing Surveys **33**(1) (2001) 31–88
- C. Alves, E. Caceres, S. Song: An all-substrings common subsequence algorithm.
  Discrete Applied Mathematics **156**(7) (April 2008) 1025–1035
- A. Tiskin: Semi-local string comparison: Algorithmic techniques and applications.
  Mathematics in Computer Science, to appear. arXiv: 0707.3619