

Efficient High-Similarity String Comparison: The Waterfall Algorithm

(Extended Abstract)

Alexander Tiskin
DIMAP and Department of Computer Science
University of Warwick
Coventry CV4 7AL, United Kingdom

ABSTRACT

This paper outlines the design of a bit-parallel, multi-string algorithm for high-similarity string comparison. We present it in the framework for the longest common subsequence (LCS) problem developed by the author in [31]. The algorithm is based on a bit-parallel LCS algorithm by Crochemore et al. [14].

1. LCS AND ALIGNMENT DAG

A classical approach to string comparison is based on the following numerical measure of string similarity.

Definition 1. Given strings a, b , the longest common subsequence (LCS) problem asks for the length of the longest string that is a subsequence of both a and b . We will call this length the LCS score of strings a, b , and denote it by $lcs(a, b)$.

Example. Let $a = \text{"BAABCBCA"}$, $b = \text{"BAABCABCABACA"}$. (This example, borrowed from Alves et al. [6], will serve as a running example for this chapter.) String b of length 13 contains the whole string a of length 8 as a subsequence, therefore we have

$$lcs(a, b) = 8$$

The best known algorithms for the LCS problem run within (model-dependent) polylogarithmic factors of $O(mn)$.

A standard method for the LCS problem represents a problem instance by a *dag* (directed acyclic graph) on a rectangular grid of nodes, where every edge is assigned a score of either 0 or 1.

Example. Figure 1 shows the alignment dag for strings $a = \text{"BAABCBCA"}$, $b = \text{"BAABCABCABACA"}$. All edges are directed left-to-right and top-to-bottom. The diagonal edges of score 0 are not shown. The colour of the remaining edges indicates their scores: blue (respectively, red) corresponds to edge score 0 (respectively, 1).

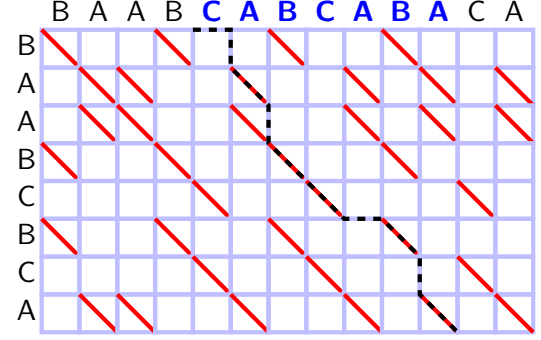


Figure 1: Alignment dag $G_{a,b}$ and a highest-scoring path

2. THE BLOW-UP TECHNIQUE

The concept of LCS score is generalised by that of (*weighted*) *alignment score* (see e.g. [20]). An *alignment* of strings a, b is obtained by putting a subsequence of a into one-to-one correspondence with a (not necessarily identical) subsequence of b , character by character and respecting the index order. The corresponding pair of characters, one from a and the other from b , are said to be *aligned*. A character that is not aligned against a character of another string is said to be aligned against a *gap* in that string. Each of the resulting character alignments is given a real *weight*:

- a pair of aligned matching characters has weight $w_M \geq 0$;
- a pair of aligned mismatching characters has weight $w_X < w_M$;
- a gap-character or character-gap pair has weight $w_G \leq \frac{1}{2}w_X$; it is normally assumed that $w_G \leq 0$ (i.e. this weight is in fact a penalty).

The intuition behind the weight inequalities is as follows: aligning a matching pair of characters is always better than aligning a mismatching pair of characters, which in its turn is never worse than leaving both characters unaligned (aligned against a gap).

Definition 2. The (*weighted*) alignment score for strings a, b is the maximum total weight across all possible alignments of a against b .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Joint EDBT/ICDT 2013 workshops, March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1599-9/13/03 ...\$15.00.

Example. The LCS alignment score is given by

$$w_M = 1 \quad w_X = w_G = 0$$

A slightly more sophisticated alignment score, intended to penalise gaps in DNA sequence alignment, is given by

$$w_M = 1 \quad w_X = 0 \quad w_G = -0.5$$

Another alignment score used for DNA sequence comparison [12, Section 1.3] is given by

$$w_M = 2 \quad w_X = -1 \quad w_G = -1.5$$

The concept of alignment dag can be naturally generalised to the weighted case. To distinguish between the weighted and unweighted cases, we will use a script font in the corresponding notation.

The weighted alignment of strings a, b corresponds to a *weighted alignment dag* $\mathcal{G}_{a,b}$, where diagonal match edges, diagonal mismatch edges, and horizontal/vertical edges have weight w_M, w_X, w_G , respectively.

Given an arbitrary set of alignment weights, it is often convenient to normalise them so that $0 = w_G \leq w_X < w_M = 1$. To obtain such a normalisation, first observe that, given a pair of strings a, b , and arbitrary weights $w_M \geq 0, w_X < w_M, w_G \leq \frac{1}{2}w_X$, we can replace the weights respectively by $w_M + 2x, w_X + 2x, w_G + x$, for any real x . This weight transformation increases the score of every global alignment (top-left to bottom-right) path in $\mathcal{G}_{a,b}$ by $(m+n)x$. Therefore, the relative scores of different global alignment paths do not change. In particular, the maximum global alignment score is attained by the same path as before the transformation. By taking $x = -w_G$, and dividing the resulting weights by $w_M - 2w_G > 0$, we achieve the desired normalisation. (A similar method is used by Rice et al. [28]; see also [15, 21].)

Definition 3. Given original weights w_M, w_X, w_G , the corresponding normalised weights are $w_M^* = 1, w_X^* = \frac{w_X - 2w_G}{w_M - 2w_G}, w_G^* = 0$. We call the corresponding alignment score the normalised score. The original alignment score h can be restored from the normalised score h^* by reversing the normalisation: $h = h^* \cdot (w_M - 2w_G) + (m+n) \cdot w_G$.

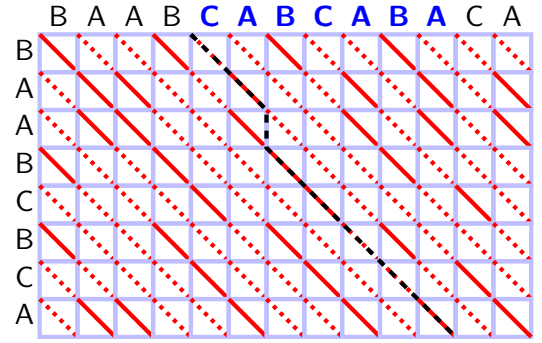
Example. In Example 2, the LCS score is already normalised. The other two scores give the normalised scores with weights $w_M^* = 1, w_X^* = \frac{0-2 \cdot (-0.5)}{1-2 \cdot (-0.5)} = 0.5$ (respectively, $w_X^* = \frac{-1-2 \cdot (-1.5)}{2-2 \cdot (-1.5)} = 0.4$), and $w_G^* = 0$.

Definition 4. A set of character alignment weights will be called rational, if all the weights are rational numbers.

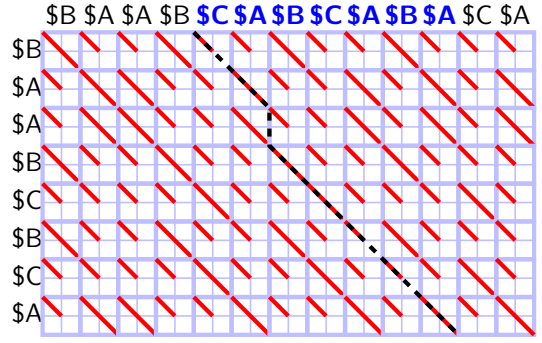
Given a rational set of normalised weights, the alignment score problem on strings a, b can be reduced to the LCS problem by the following *blow-up* procedure. Let $w_X = \frac{\mu}{\nu} < 1$, where μ, ν are positive natural numbers. We transform input strings a, b of lengths m, n into new *blown-up* strings \tilde{a}, \tilde{b} of lengths $\tilde{m} = \nu m, \tilde{n} = \nu n$. The transformation consists in replacing every character γ in each of the strings by a substring $\$^\mu \gamma^{\nu-\mu}$ of length ν (here, $\$$ is a special guard character, not present in the original strings).

Example. Figure 2 shows semi-local weighted alignment of strings a, b . We assume the normalised alignment weights $w_M = 1, w_X = 0.5, w_G = 0$.

Subfigure 2a shows the alignment dag $\mathcal{G}_{a,b}$. Match edges of weight $w_M = 1$ and mismatch edges of weight $w_M =$



(a) Weighted alignment dag $\mathcal{G}_{a,b}$



(b) Alignment dag $\mathcal{G}_{\tilde{a},\tilde{b}}$ for the blown-up strings

Figure 2: The blow-up technique

0.5 are shown respectively by solid and dotted red lines.] Subfigure 2b shows the alignment dag $\mathcal{G}_{\tilde{a},\tilde{b}}$ for the blown-up strings \tilde{a}, \tilde{b} , where $\nu = 2$.

An important special case of weighted string alignment is the *edit distance problem*. Here, the characters are assumed to match “by default”: $w_M = 0$. The mismatches and gaps are penalised: $2w_G \leq w_X < 0$. The resulting score is always nonpositive. Equivalently, we regard string a as being transformed into string b by a sequence of weighted *character edits*:

- character insertion or deletion (*indel*) has weight $-w_G > 0$;
- character *substitution* has weight $-w_X > 0$.

Definition 5. The (weighted) edit distance between strings a, b is the minimum total weight of a sequence of character edits transforming a into b . Equivalently, it is the (non-negative) absolute value of the corresponding (nonpositive) alignment score.

The edit distance is a *metric*: it is nonnegative (zero on equal strings and positive otherwise), symmetric, and satisfies the triangle inequality.

Example. The *indel distance* (also called the *LCS distance* or *simple distance*) [26, 9, 11] has indel weight 1 and substitution weight 2, making a substitution equivalent to an insertion-deletion pair, and thus redundant. The corresponding *indel alignment score* is given by

$$w_M = 0 \quad w_X = -2 \quad w_G = -1$$

The *indelsub distance* (also called the *Levenshtein distance*) [23] has both indel weight and substitution weight equal to 1. The corresponding *indelsub alignment score* is given by

$$w_M = 0 \quad w_X = w_G = -1$$

3. LCS COMPUTATION AS A TRANSPOSITION NETWORK

Comparison networks were first considered as a computation model by Batchier [10] (see also [13, 4]).

Definition 6. A circuit represents a computation as a dag (directed acyclic graph). The internal nodes of a circuit are labeled by elementary operations on values, which are passed along the edges; source and sink nodes represent the inputs and outputs, respectively. A comparator node (or simply comparator) is a node of indegree and outdegree 2, which sorts its two operands in increasing order. In other words, a comparator node compares the operands on the incoming edges, and returns each of the minimum and the maximum operand on a prescribed outgoing edge. A comparison network is a circuit where all internal nodes are comparator nodes.

The most well-studied types of comparison networks are the ones that either sort their inputs, or merge two disjoint subsets of inputs. In particular, Batchier [10] gave classical merging networks with $O(n \log n)$ comparators, and sorting networks with $O(n \log^2 n)$ comparators. Ajtai et al. [2, 3] gave an asymptotically optimal sorting network with $O(n \log n)$ comparators; their construction was subsequently simplified by Paterson [27] and by Seiferas [30].

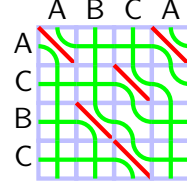
Comparison networks are usually visualised by *wire diagrams* (also known as *Knuth diagrams*), where the values propagate across the network along a set of parallel *wires*. Every comparator is represented by a directed line segment, drawn orthogonally between two (not necessarily adjacent) wires. The order in which a comparator returns the minimum and the maximum output is consistent across all the comparators in the network. The most common convention on wire diagrams (adopted e.g. by Knuth [22]) is to draw the wires horizontally, directed from left to right; sometimes (e.g. in [27]), they are drawn vertically, directed from top to bottom. In our setting, it will be convenient to draw the wires diagonally, directed from top-left to bottom-right. Comparator segments will be directed so that the minimum output is returned on the bottom-left, and the maximum on the top-right.

We will be dealing exclusively with the following restricted type of comparison network.

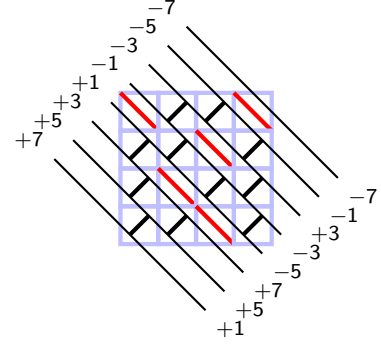
Definition 7. A comparison network is called a *transposition network*, if in its wire diagram, all the comparisons are between adjacent wires.

Every alignment dag can be associated with a transposition network.

Example. Figure 3 illustrates the transposition network for an alignment dag on strings $a = \text{“ACBC”}$, $b = \text{“ABCA”}$. Subfigure 3a shows the alignment dag $G_{a,b}$. Following our usual colour conventions, the diagonal edges of weight 1 are shown in red, and the diagonal edges of weight 0 are omitted. Subfigure 3a also shows the reduced seaweed braid laid over



(a) Alignment dag with a seaweed braid



(b) Corresponding transposition network

Figure 3: An alignment dag and its transposition network

the alignment dag $G_{a,b}$ (see [31] for an introduction to seaweed braids). Subfigure 3b shows in black the corresponding transposition network, laid diagonally over $G_{a,b}$. A cell contains a comparator, if and only if it does not contain a red diagonal edge.

Let us denote the input and output arrays of a transposition network $\mathcal{N}(G_{a,b})$ by $x\langle -m : n \rangle$ and $y\langle 0 : m+n \rangle$, respectively. Assuming all input values of the network are distinct, each value traces a well-defined path through the network. We write $\pi(\hat{i}) = \hat{j}$, if the input $x(\hat{i})$ ends up as the output $y(\hat{j})$.

Although a transposition network $\mathcal{N}(G_{a,b})$ is in general neither merging nor sorting, it is useful to consider its operation on certain kinds of input. In particular, we consider anti-sorted *binary input*: a sequence of ones, followed by a sequence of zeros. It turns out that the output of such a network can be used to obtain the LCS score of the input strings.

Theorem 1. Consider an alignment dag $G_{a,b}$ and the corresponding semi-local seaweed matrix $P_{a,b}$. Let the transposition network $\mathcal{N}(G_{a,b})$ operate on an anti-sorted input array $x\langle -m : n \rangle$, consisting of m 1-values and n 0-values:

$$x(\hat{i}) = \begin{cases} 1 & \text{if } \hat{i} \in \langle -m : 0 \rangle \\ 0 & \text{if } \hat{i} \in \langle 0 : n \rangle \end{cases}$$

Let $y\langle 0 : m+n \rangle$ be the output array of $\mathcal{N}(G_{a,b})$. Then we

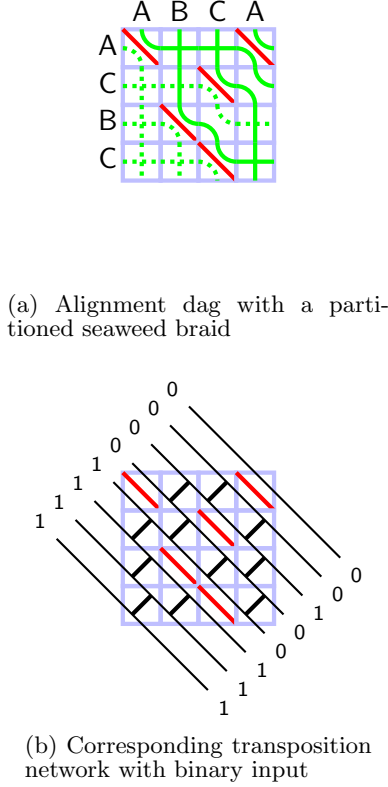


Figure 4: LCS computation by a transposition network

have

$$lcs(a, b) = \sum_{j \in \langle 0:n \rangle} y(j) = m - \sum_{j \in \langle n:m+n \rangle} y(j)$$

Example. Figure 4 illustrates Theorem 1 on the same pair of strings as Figure 3. The seaweeds originating at the top (respectively, the left-hand side) of the dag are shown by solid (respectively, dotted) lines. Subfigure 4b shows the transposition network of Subfigure 3b, laid over the alignment dag $G_{a,b}$. The network is given an anti-sorted binary input. The path of each 0-value (respectively, 1-value) corresponds to a solid (respectively, dotted) seaweed. We have

$$lcs(a, b) = \sum_{j \in \langle 0:4 \rangle} y(j) = 3$$

$$lcs(a, b) = 4 - \sum_{j \in \langle 4:8 \rangle} y(j) = 4 - 1 = 3$$

as claimed by Theorem 1.

4. PARAMETERISED LCS

An algorithm’s complexity is most commonly defined to be a function of a single argument: the input size. However, in the pursuit of efficiency, algorithms may also be designed to be sensitive to various other parameters of the input. In the context of string comparison, the two most relevant parameters are:

- the input strings’ alignment score; we consider primarily the LCS score $\lambda = lcs(a, b)$;

- the input strings’ edit distance; we consider primarily the LCS distance $\kappa = dist_{LCS}(a, b) = m + n - 2\lambda$.

In this section, we study algorithms for the LCS problem that are sensitive to these parameters. We consider *low-similarity* and *high-similarity* LCS algorithms. The running times of such algorithms are parameterised respectively by λ and κ , so that advantage can be taken of the low value of a parameter. More generally, one can also use weighted alignment scores or weighted edit distances (e.g. the Levenshtein distance) as parameters.

As we aim for algorithms that, for a low value of the parameter, run substantially faster than $\Theta(mn)$, we cannot afford to perform all the mn pairwise comparisons of characters from each string. We assume a character comparison model that allows comparison outcomes “equal”, “less than” and “greater than”, so that the “missing” comparisons can be obtained by transitivity, and algorithms with running time $o(mn)$ become possible.

For simplicity, we ignore the trivial cases $\lambda = 0$ (the two input strings have no characters in common) and $\kappa = 0$ (the two input strings are identical). As usual, we denote the alphabet size by σ . Without loss of generality, we assume $m \leq n$.

Low-similarity comparison (sensitive to low λ).

In this type of comparison, the overall number of matching character pairs will be low. To locate these matching pairs effectively, the input strings a and b are preprocessed into a data structure that allows efficient queries defined by the *string identification problem* [1, 17, 29]. The preprocessing builds a binary search tree on each input string, and returns a partitioning of both a and b into character equality classes. This preprocessing procedure runs in time $m \log \sigma$ (respectively, $n \log \sigma$).

After the preprocessing, the low-similarity LCS problem can be solved by one of the algorithms due to Hirschberg [16], Hsu and Du [17] (see also Apostolico [7]), Apostolico and Guerra [9]. All these algorithms run in time $O(n\lambda)$. Apostolico, Browne and Guerra [8] proposed another algorithm that requires no preprocessing, and runs in time $O(n\lambda \log \sigma)$ and linear space.

High-similarity comparison (sensitive to low κ).

In this type of comparison, the overall number of matching character pairs may be as high as mn . However, an efficient high-similarity LCS algorithm may not need to look at all these matches; speaking informally, a good algorithm “will know where to look for relevant matches”. In fact, it is sufficient to consider $O(n \cdot \kappa)$ character matches overall. In contrast to low-similarity comparison, there is no need for preprocessing the input strings.

Efficient high-similarity LCS algorithms have been given by Ukkonen [32], Myers [24], Wu et al. [33]. All these algorithms run in time $O(n \cdot \kappa)$. Apostolico, Browne and Guerra [8] proposed another algorithm that runs in time $O(n \cdot \kappa)$ and linear space.

Flexible comparison (sensitive to both low λ and low κ).

This type of comparison can be achieved by preprocessing the input string as for low-similarity comparison, and then running both comparison types alongside each other.

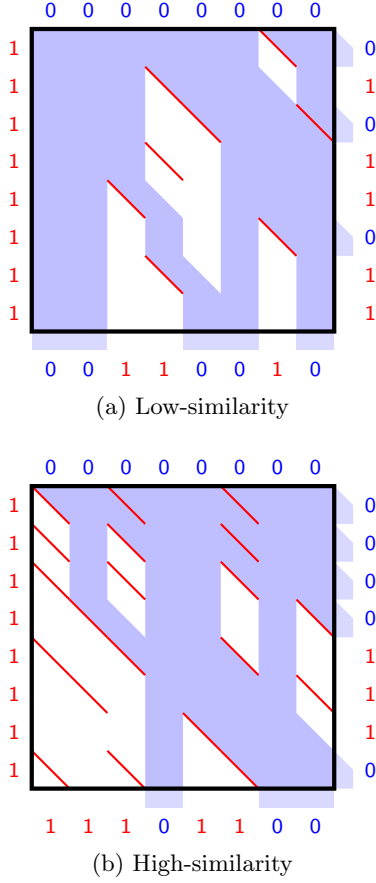


Figure 5: The waterfall algorithm

However, dedicated flexible comparison algorithms have also been proposed. In particular, the flexible LCS algorithm by Hirschberg [16] runs in time $O(\lambda\kappa \log n)$, and one by Rick [29] in time $O(\lambda\kappa)$ and linear space.

We now describe an algorithm based on the comparison network method. The algorithm is sensitive to both parameters λ and κ , providing flexible LCS computation efficient in both the low- and the high-similarity case. Our algorithm matches existing flexible-LCS algorithms in running time. We call it the *waterfall algorithm*. For a formal description of the algorithm, see [31].

The name “waterfall algorithm” is justified by the following interpretation. Let us think of the 0-values as a viscous, non-compressible “discrete liquid” that flows through the alignment dag under gravity. The blocks of adjacent 0-values correspond to “jets” in this “liquid”; these “jets” may split or merge while flowing through the dag. Initially, there is just a single “jet” flowing vertically down through the top boundary of the dag. The diagonal match edges form barriers for the “liquid”: whenever such a barrier is hit from above by a “jet”, the “jet” splits into two, and the right “subjet” gets displaced by one unit, following the inclination of the barrier. After the displacement, the right “subjet” may touch another “jet” to its right, in which case the two touching “jets” merge into one. The amount of “liquid” that flows out of the dag at its right-hand side (respectively, its bottom) corresponds to the LCS score λ (respectively, the LCS distance μ) of the input strings.

Example. Figure 5 illustrates two separate runs of the waterfall algorithm:

- Subfigure 5a shows the low-similarity case, with input strings $a = \text{“ABCBDABE”}$, $b = \text{“FFDBCAC”}$;
- Subfigure 5b shows the high-similarity case, with input strings $a = \text{“AAAABABCA”}$, $b = \text{“ABADCADB”}$.

Following our usual convention, the diagonal edges in match cells are shown in red. The rest of the alignment dag, as well as the input strings, are kept implicit. The anti-sorted input to the transposition network is represented by the sequences of red 1-values and blue 0-values along the left (respectively, the top) boundary of the alignment dag.

The iterative procedure of block splitting and merging is shown by filling in the interior of the alignment dag with blue and white pattern as follows. Every horizontal edge in the alignment dag corresponds to a blue (respectively, white) streak in the pattern, whenever that edge is crossed by an (implicit) wire carrying a 0- (respectively, 1-) value. Therefore, for each row index l in the dag, the corresponding sequence of blocks and gaps is represented by a sequence of continuous blue and white streaks in a horizontal line at level l . The transition of block sequences between every pair of successive rows is shown by connecting the corresponding pairs of blue streaks with a blue strip. For the vertical (respectively, diagonal) transition of a single 0-value, the connecting strip has the shape of a unit square (respectively, unit-width parallelogram).

The output of the transposition network is represented by the mixed sequence of red 1-values and blue 0-values along the bottom and right boundaries of the alignment dag. Each output 0-value is also shown by a blue tab. By Theorem 1, we have

$$lcs(a, b) = \sum_{j \in \langle 0:8 \rangle} y(j) = 3 = 8 - \sum_{j \in \langle 4:8 \rangle} y(j) = 8 - 5 = 3$$

$$lcs(a, b) = \sum_{j \in \langle 0:8 \rangle} y(j) = 5 = 8 - \sum_{j \in \langle 4:8 \rangle} y(j) = 8 - 3 = 5$$

in Subfigures 5a and 5b, respectively.

Just as the splitting/merging procedure in the waterfall algorithm is not symmetric with respect to blocks and gaps, so the pattern in Figure 5 is not symmetric either with respect to horizontal and vertical directions, or with respect to 0- and 1-values. For this reason, while the blocks are represented by the usual blue colour, the gaps are left uncoloured (i.e. are represented by the “background” white). The red colour, which we would normally use to represent 1-values, is not present in the pattern.

Parameterised LCS algorithms are closely related to *threshold LCS* algorithms. Here, a threshold value for the parameter λ or κ is given, and the algorithm is required to output the LCS score of the input strings, as long as this value is below the threshold, or to report “excess”, if the LCS score (respectively, LCS distance) exceeds the threshold; in the latter case, there is no requirement for the score to be output explicitly. In general, a threshold algorithm can be obtained from a parameterised algorithm by setting an appropriate time-out threshold as a function of the parameter threshold, and reporting “excess”, if the algorithm’s running time exceeds the time-out threshold. Alternatively, a threshold LCS algorithm can be obtained from the waterfall algorithm by reporting “excess”, if the current number of blocks exceeds the threshold value for λ (respectively, κ).

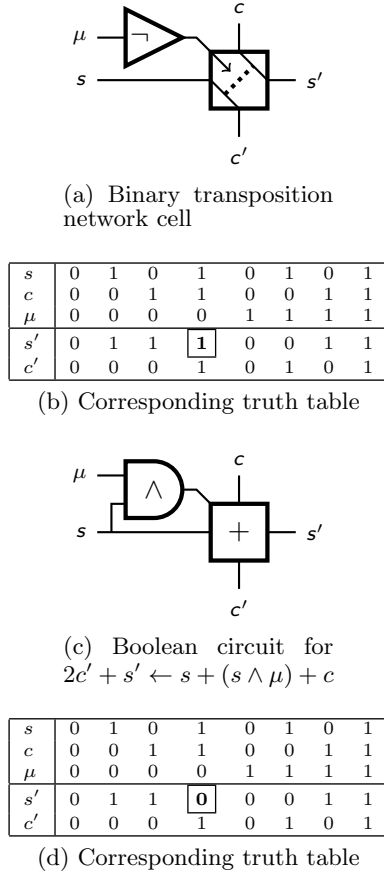


Figure 6: Binary transposition network via a Boolean circuit

5. BIT-PARALLEL LCS

The most efficient practical method for computing the (global) LCS score for a pair of strings is by *bit-parallel* algorithms. These algorithms take advantage of bitwise Boolean operations on bit vectors available in modern processors, often in combination with arithmetic operations on the same vectors as integers. We denote by w the *word* (standard bit vector) length; in modern general-purpose processors, word length is often $w = 64$.

Early bit-parallel string comparison algorithms were given by Allison and Dix [5] and by Myers [25]. Crochemore et al. [14] proposed an efficient bit-parallel LCS algorithm, running in time $O(mn/w)$. For every w cells of the alignment dag $G_{a,b}$, the algorithm only performs five elementary operations (one arithmetic and four Boolean). Hyvrö [18] improved this to four operations (two arithmetic and two Boolean).

Both algorithms [14, 18] can be viewed as an implementation of a binary transposition network, described in Section 3, by standard bit-parallel processor instructions. Figure 6 shows the main idea of such an implementation for the algorithm of [14]. Consider a cell in a binary transposition network, and let us denote its input bits by s , c , and its output bits by s' , c' , as shown in Subfigure 6a. Let us denote by μ an extra input bit, which takes value 1 if and only if the current dag cell is a match cell. The operation of a network cell is fully described by the truth table in Subfigure 6b.

Now consider a Boolean circuit shown in Subfigure 6c. The circuit consists of an \wedge -gate and a *full adder* element, which adds its three input bits arithmetically, and returns the sum as two separate output bits. Let us again denote the input bits by s , c , μ , and the output bits by s' , c' . Then, the circuit computes a Boolean-arithmetic expression

$$2c' + s' \leftarrow s + (s \wedge \mu) + c$$

The operation of such a circuit is fully described by the truth table in Subfigure 6d.

Note that the truth tables in Subfigures 6b and 6d differ in just the one highlighted bit. This difference can be corrected by two extra Boolean operations, resulting in a Boolean-arithmetic expression that fully implements the operation of a single transposition network cell:

$$2c' + s' \leftarrow (s + (s \wedge \mu) + c) \vee (s \wedge \neg \mu) \quad (1)$$

Now consider a row of n cells in the alignment dag $G_{a,b}$, assuming for the moment $n \leq w$. Let S denote a word variable that will hold the input s , and then the output s' , for each cell, least significant bit first. Likewise, let M denote a word constant that holds the match parameter μ for each cell, least significant bit first. (The input c and output c' will not be represented explicitly, but will instead correspond to a propagating carry bit in word integer addition, from the least significant bit all the way to the most significant bit.) The operation of the transposition network $\mathcal{N}(G_{a,b})$ in the given row of cells corresponds to evaluating an expression

$$S \leftarrow (S + (S \wedge M)) \vee (S \wedge \neg M) \quad (2)$$

which is obtained from (1) by identifying the output c' of each cell with the input c of the next cell in the row. Here, the Boolean operations are bitwise, and the addition is on integers represented by the words. Note that for an exact correspondence with (2), the roles of 0-values and 1-values in the waterfall algorithm must be exchanged (or, alternatively, the cells must be composed into words by columns, rather than by rows).

If $n > w$, then each row of the alignment dag is partitioned into $\lceil n/w \rceil$ words of length w . In this case, expression (2) needs to be modified to allow carry propagation from each word to the next word in its row.

The described bit-parallel approach can be extended to provide even more efficient string comparison in the case of highly similar strings. We consider the threshold version of the high-similarity LCS problem, as described in Section 4. Let κ denote a threshold on $\text{dist}_{LCS}(a, b)$, i.e. the LCS distance between the input strings a, b . We assume for simplicity that the value κ is odd, and that $m = n$. Under these assumptions, a highest-scoring path in the alignment dag $G_{a,b}$ must lie strictly within a symmetric diagonal band of width $\kappa + 1$, unless $\text{dist}_{LCS}(a, b)$ exceeds κ . Hence, the waterfall algorithm can be modified as follows. First, we ignore all character matches outside the band, as well as on the band's lower-left and upper-right boundaries. We also ignore all the (explicit) input 0-values and all the (implicit) input 1-values outside the band; therefore, we only have $\frac{\kappa+1}{2}$ (explicit) input 0-values at the top of the band, and $\frac{\kappa+1}{2}$ (implicit) input 1-values at the left-hand side of the band. Finally, we create artificial *separator matches* along the bottom-left boundary of the band. The LCS score $\text{lcs}(a, b)$ can be obtained by counting the output 0-values at the bottom of the band (or, symmetrically, the output 1-values at the right-hand side of

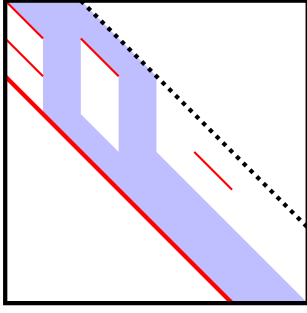


Figure 7: High-similarity bit-parallel waterfall algorithm

the band). The algorithm reports “exceed”, if all the input 0-values are output at the bottom of the band. The algorithm runs in time $O(m\kappa/w)$.

Figure 7 shows a run of the resulting high-similarity bit-parallel waterfall algorithm, for $\kappa = 3$. The visual conventions are similar to those in Figures 5a, 5b. Note that the band is of width $\kappa + 1 = 4$, and that there are $\frac{\kappa+1}{2} = 2$ input 0-values at the band’s top. Both these values end up as output 0-values at the band’s bottom, hence the algorithm returns “exceed” in the given run.

The described algorithm is particularly easy to implement when the bandwidth $m + 1 \leq w - 1$. In such a case, every row of the band fits into a single word. The bit-parallel five- (respectively, four-) instruction sequence of either of [14, 18] can be used; the only modification required is an extra shift instruction in each row, to account for the band right-shifting by 1 when moving to the next row.

Still further optimisation is possible in the case of *multi-string* comparison. This type of comparison has been considered e.g. by Hyrö et al. [19]. Here, we asked to compute the LCS score for string a against each of the r strings b_0, \dots, b_{r-1} , all of length n . We assume that we are given a single threshold κ on all $\text{dist}_{\text{LCS}}(a, b_s)$, $0 \leq s < r$. As before, we assume for simplicity that the value κ is odd, and that $m = n$. The problem can be solved by r independent runs of the high-similarity bit-parallel waterfall algorithm. However, it is possible to combine these runs into a single bit-parallel computation, where in each step, we evaluate a single row from every one of the r bands. The bands are packed together in a single super-band of width $r(\kappa + 1)$; individual bands within the super-band are separated by diagonals of separator matches.

Figure 8 shows a run of the resulting high-similarity bit-parallel multi-string waterfall algorithm, for $\kappa = 3$ and $r = 4$. The leftmost band (band 0) is identical to the band in Figure 7. The algorithm returns “exceed” for bands 0, 1, 3. For band 2, we have a single 0-value output at the bottom of the band, hence $\text{lcs}(a, b_2) = n - 1$ by Theorem 1.

Finally, note that all the described techniques for bit-parallel string comparison are compatible with integer-weighted alignment, by application of the blow-up technique described in Section 2. For example, we can obtain bit-parallel algorithms for Levenshtein alignment score (or, symmetrically, Levenshtein edit distance) by blowing up the alignment dag by a factor of 2 in each dimension.

6. REFERENCES

- [1] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23:1–12, 1976.
- [2] M. Ajtai, J. Komlós, and E. Szemerédi. An $o(n \log n)$ sorting network. In *Proceedings of the 15th ACM STOC*, pages 1–9, 1983.
- [3] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.
- [4] S. W. Al-Haj Baddar and K. E. Batchier. *Designing Sorting Networks: A New Paradigm*. Springer, 2011.
- [5] L. Allison and T. I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23(5):305–310, 1986.
- [6] C. E. R. Alves, E. N. Cáceres, and S. W. Song. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics*, 156(7):1025–1035, 2008.
- [7] A. Apostolico. Remark on the Hsu–Du new algorithm for the longest common subsequence problem. *Information Processing Letters*, 25(4):235–236, 1987.
- [8] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92(1):3–17, 1992.
- [9] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987.
- [10] K. E. Batchier. Sorting networks and their applications. In *AFIPS Conference Proceedings*, volume 32, pages 307–314. Thompson Book Company, 1968.
- [11] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of the 7th SPIRE*, pages 39–48, 2000.
- [12] Kun-Mao Chao and Louxin Zhang. *Sequence Comparison: Theory and Methods*, volume 7 of *Computational Biology Series*. Springer, 2009.
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw–Hill, second edition, 2001.
- [14] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the Longest Common Subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001.
- [15] D. Gusfield, K. Balasubramanian, and D. Naor. Parametric optimization of sequence alignment. *Algorithmica*, 12:312–326, 1994.
- [16] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977.
- [17] W. J. Hsu and M. W. Du. New algorithms for the lcs problem. *Journal of Computer and System Sciences*, 29:133–152, 1984.
- [18] H. Hyrö. Bit-parallel lcs-length computation revisited. In *Proceedings of AWOCA*, 2004.
- [19] H. Hyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate string matching. In

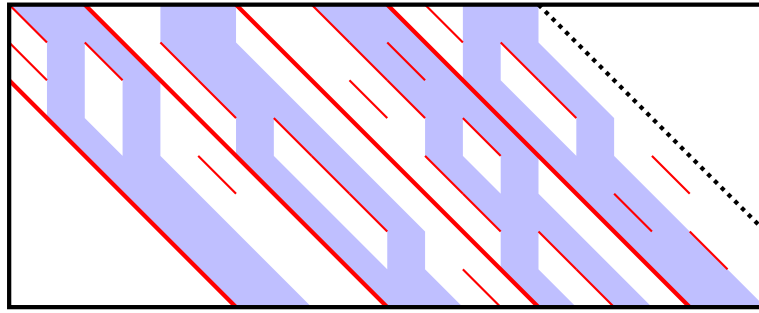


Figure 8: High-similarity bit-parallel multi-string waterfall algorithm

Proceedings of WEA, volume 3059 of *Lecture Notes in Computer Science*, pages 285–298, 2004.

- [20] B. N. Jackson and S. Aluru. Pairwise sequence alignment. In *Handbook of Computational Molecular Biology*, Chapman and Hall/CRC Computer and Information Science Series, chapter 1, pages 1–1 – 1–31. Chapman and Hall/CRC, 2006.
- [21] N. C. Jones and P. A. Pevzner. *An Introduction to Bioinformatics Algorithms*. Computational Molecular Biology. The MIT Press, 2004.
- [22] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1998.
- [23] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.
- [24] E. W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.
- [25] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.
- [26] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.
- [27] M. S. Paterson. Improved sorting networks with $o(\log n)$ depth. *Algorithmica*, 5(1):75–92, 1990.
- [28] S. V. Rice, H. Bunke, and T. A. Nartker. Classes of cost functions for string edit distance. *Algorithmica*, 18:271–280, 1997.
- [29] C. Rick. Simple and fast linear space computation of longest common subsequences. *Information Processing Letters*, 75(6):275–281, 2000.
- [30] J. Seiferas. Sorting networks of logarithmic depth, further simplified. *Algorithmica*, 53(3):374–384, 2009.
- [31] A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. Technical Report 0707.3619, arXiv.
- [32] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.
- [33] S. Wu, U. Manber, G. Myers, and W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990.