International Journal of Foundations of Computer Science


# BIT-PARALLEL COMPUTATION OF LOCAL SIMILARITY SCORE MATRICES WITH UNITARY WEIGHTS

HEIKKI HYYRÖ
*Department of Computer Sciences, University of Tampere, Finland.*
heikki.hyyro@gmail.com

GONZALO NAVARRO
*Department of Computer Science, University of Chile, Chile.*
gnavarro@dcc.uchile.cl

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

Local similarity computation between two sequences permits detecting all the relevant alignments present between subsequences thereof. A well-known dynamic programming algorithm works in time $O(mn)$, $m$ and $n$ being the lengths of the subsequences. The algorithm is rather slow when applied over many sequence pairs. In this paper we present the first bit-parallel computation of the score matrix, for a simplified choice of scores. If the computer word has $w$ bits, then the resulting algorithm works in $O(mn \log \min(m, n, w)/w)$ time, achieving up to 8-fold speedups in practice. Some DNA comparison applications use precisely the simplified scores we handle, and thus our algorithm is directly applicable. In others, our method could be used as a raw filter to discard most of the strings, so the classical algorithm can be focused only on the substring pairs that can yield relevant results.

*Keywords:* Local similarity; Approximate string matching; Bit-parallelism

## 1. Introduction and Related Work

Sequence comparison is a fundamental task in Computational Biology, in order to detect relevant similarities between a pair of genetic or protein sequences [5]. Three kinds of similarities are of interest: (*i*) global similarity compares two strings as a whole, (*ii*) semiglobal (or semilocal) similarity looks for substrings of a given string that are similar to a second given string, (*iii*) local similarity looks for similar substrings of two given strings.

Similarity is usually expressed using a *score function*, which gives prizes or penalties to operations on the strings and to pairings of characters of the two strings. Usually pairing the same character in both strings involves a prize because we

have found a similarity. Pairing different characters, inserting or removing characters, involves penalties. The specific values for prizes and penalties depend on the biological model used for the similarity (for example, logarithms of mutation probabilities). The similarity is then expressed as the highest possible score of a sequence of operations that align one string to the other.

Global and semiglobal similarity find applications in other areas such as text searching. Global similarity computation is then seen as a *distance computation*. The distance is never negative, and the smaller it is, the more similar the sequences are. Semiglobal similarity can be converted into an approximate search problem, namely to find the approximate occurrences of a short pattern inside a long text. Local similarity, instead, is more specific to computational biology applications.

All these sorts of similarity computations can be easily carried out in $O(mn)$ time using dynamic programming. Given strings $A_{1\ldots m}$ and $B_{1\ldots n}$, the general method is to compute an $(m+1) \times (n+1)$ matrix $C$ whose cell $C_{i,j}$ gives the maximum score/minimum distance to align/convert $A_{\ldots i}$ to $B_{\ldots j}$. The cells of row 0 and column 0 form initially known boundary cases, and the remaining $m \times n$ cells are computed using a recurrence. For example, for global similarity score computation we may have $C_{i,0} = -i$, $C_{0,j} = -j$, and for $i, j > 0$

$$\begin{array}{lcl} C_{i,j} & = & \max(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) & = & \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{array}$$

where we have assumed that all penalties are $-1$ and prizes are $+1$. More complicated score functions can be real-valued and depend on the characters involved. The maximum score for the strings $A$ and $B$ is $C_{m,n}$.

If we are instead computing distance, we may have $C_{i,0} = i$, $C_{0,j} = j$, and

$$\begin{array}{lcl} C_{i,j} & = & \min(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} + 1,\ C_{i-1,j} + 1) \\ \text{where } \delta(A_i, B_j) & = & \text{if } A_i = B_j \text{ then } 0 \text{ else } 1 \end{array}$$

for $i, j > 0$, where we have assumed that all costs are 1. The minimum distance between $A$ and $B$ is $C_{m,n}$.

Semiglobal similarity computation is obtained by using the above formulas except that $C_{0,j} = 0$, so that an alignment of $A$ can start afresh at any position in $B$. High score/low distance at cell $C_{m,j}$ tells us that an interesting alignment ends at position $j$ in $B$.

Local similarity computation needs a somewhat different arrangement and, curiously, it seems not expressible using the distance model, but just the score model. In this case we have $C_{i,0} = C_{0,j} = 0$, and for $i, j > 0$

$$\begin{array}{lcl} C_{i,j} & = & \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) & = & \text{if } A_i = B_j \text{ then } 1 \text{ else } -1 \end{array}$$

where we remark the 0 value involved in the maximum. The objective of this zero is that if an alignment in progress has given us more penalties than prizes, then it

is better to start afresh from that position. Any cell value $C_{i,j}$ that is high enough indicates that similar substrings end at position $i$ in $A$ and $j$ in $B$.

Much effort has been done in order to efficiently compute the *distance* matrix, both for global and semiglobal alignments. In particular, *bit-parallelism* has given the best results in practice. Bit-parallelism packs several values inside a computer word and updates them all in one shot. This paradigm has been applied successfully in several other problems as well, such as exact string matching [1], longest common subsequence computation [3], and tree matching [13]. The bit-parallel algorithm that best "parallelizes" the distance matrix computation is from Myers [9], which computes semiglobal similarity and is easily adapted to compute global similarity [6, 7]. Using Myers' algorithm, both similarities can be computed in $O(mn/w)$ time using a computer word of $w$ bits, which is the optimal bit-parallel speedup. Myers' algorithm strongly relies on the fact that consecutive cells of $C_{i,j}$ differ only by $-1$, 0, or 1. Several other bit-parallel algorithms exploiting the same property exist [10].

Bit-parallel computation of *score* matrices, however, has not been attempted. Bergeron and Hamel [2] have extended Myers' scheme to handle arbitrary integer weights for substitutions, as well as a fixed weight $c$ for insertions and deletions. Their algorithm is $O(mnc \log(c)/w)$ time. This scheme does not seem to extend to compute local similarity.

Other approaches to speed up matrix computation exist. Different Four-Russians techniques [8, 14] obtain $O(mn/\log(mn))$ time. The same complexity is obtained by using a Ziv-Lempel factoring [4], which generalizes to local similarity with arbitrary weights. In practice, when applicable, bit-parallel algorithms are faster.

In this paper we present a bit-parallel local similarity algorithm inspired on Myers' scheme (and more precisely on Hyyrö's version [6]). The asymptotic running time of the algorithm is $O(mn \log \min(m, n, w)/w)$. Our algorithm assumes that aligning two characters yields a prize of $+1$ when they are equal and a penalty of $-1$ otherwise, and that inserting or deleting characters has a penalty of $-1$.

The main difficulties are (1) that the local similarity recurrence is more complicated than the one afforded by Myers (in particular, differences of $+2$ among contiguous cells are possible), and (2) that the zero in the maximization involves knowing absolute cell values, while the whole philosophy of Myers' scheme relies on storing differential values. In Sections 2 to 3.3, we solve problem (1) by a rather heavy extension of the principles behind Myers' algorithm. Problem (2) is tackled by applying bit-parallel witnesses [7] in Sections 3.4 to 4.2.

We have implemented the algorithm and compared it against plain dynamic programming, which is currently the only alternative. The experimental results are discussed in Section 5. We obtain up to 8-fold speedup over dynamic programming.

Our algorithm does not replace dynamic programming because it cannot handle other prize and penalty values. On the other hand, while score computations on protein sequences are always weighted, there are many cases of score computations on DNA sequences where our simplified model is applicable [5, Section 11.5.2 and 16.15.1]. It may also be feasible to use our method as a fast filter to discard most of the matrix and let the weighted dynamic programming algorithm concentrate only

on the matrix areas that look interesting. For example, recent research results [12] suggest that constraining substitution matrices used for database homology search to contain only values $-1$, 0, and 1, has a rather small effect on ROC measures of algorithm sensitivity and specificity.

## 2. A Bit-Parallel Design

Let us focus on the simple score function depicted in the Introduction, that is,

$$\begin{aligned} C_{i,0} &= C_{0,j} = 0 \quad \text{and, for } i,j > 0, \\ C_{i,j} &= \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \\ \text{where } \delta(A_i, B_j) &= \text{if } A_i = B_j \text{ then 1 else } -1 \end{aligned}$$

We prove now some properties of matrix $C$. Note, to start, that $C$ contains no negative values.

**Lemma 1** *Given the above definition of matrix $C$, it holds*

$$\begin{aligned} C_{i,j} - C_{i-1,j-1} &\in -1, 0, +1 && \textit{for any } i,j > 0 \\ C_{i,j} - C_{i,j-1} &\in -1, 0, +1, +2 && \textit{for any } i \geq 0, j > 0 \\ C_{i,j} - C_{i-1,j} &\in -1, 0, +1, +2 && \textit{for any } i > 0, j \geq 0 \end{aligned}$$

**Proof.** We proceed inductively, so we assume it proved for any $(i', j')$ such that $j' < j$, or $j' = j$ and $i' < i$. The base cases are immediate. Now, for the inductive case, let us start with the first proposition. The option $C_{i-1,j-1} + \delta(A_i, B_j)$ in the "max" clause of the formula for $C_{i,j}$ guarantees that $C_{i,j} - C_{i-1,j-1} \geq -1$. Inductive Hypothesis tells us that $C_{i-1,j} \leq C_{i-1,j-1} + 2$ and $C_{i,j-1} \leq C_{i-1,j-1} + 2$, and thus $C_{i,j} = \max(0,\ C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i,j-1} - 1,\ C_{i-1,j} - 1) \leq \max(C_{i-1,j-1} + \delta(A_i, B_j),\ C_{i-1,j-1} + 1,\ C_{i-1,j-1} + 1) = C_{i-1,j-1} + 1$. Here we removed the zero from the "max" clause as it is known that $C_{i-1,j-1} + 1 \geq 1 > 0$. By combining the two previous observations, we have that $-1 \leq C_{i,j} - C_{i-1,j-1} \leq 1$.

Let us now consider the second proposition. First we note that $C_{i,j} - C_{i,j-1} \geq -1$ because of the option $C_{i,j-1} - 1$ inside the "max" clause. From our Inductive Hypothesis and the above-proved first proposition we have that $C_{i,j-1} \geq C_{i-1,j-1} - 1 \geq C_{i,j} - 1 - 1 = C_{i,j} - 2$. Thus $-1 \leq C_{i,j} - C_{i,j-1} \leq 2$. The third proposition is symmetric with the second and comes out similarly. □

Given the ranges of values proved for consecutive differences, we will represent matrix $C$ incrementally using the following *bit matrices*:

$$\begin{aligned} M_{i,j} &\equiv A_i = B_j & \qquad DP_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = +1 \\ Z_{i,j} &\equiv C_{i,j} = 0 & \qquad DZ_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = 0 \\ & & \qquad DM_{i,j} &\equiv C_{i,j} - C_{i-1,j-1} = -1 \\ HT_{i,j} &\equiv C_{i,j} - C_{i,j-1} = +2 & \qquad VT_{i,j} &\equiv C_{i,j} - C_{i-1,j} = +2 \\ HP_{i,j} &\equiv C_{i,j} - C_{i,j-1} = +1 & \qquad VP_{i,j} &\equiv C_{i,j} - C_{i-1,j} = +1 \\ HZ_{i,j} &\equiv C_{i,j} - C_{i,j-1} = 0 & \qquad VZ_{i,j} &\equiv C_{i,j} - C_{i-1,j} = 0 \\ HM_{i,j} &\equiv C_{i,j} - C_{i,j-1} = -1 & \qquad VM_{i,j} &\equiv C_{i,j} - C_{i-1,j} = -1 \end{aligned}$$

Here $M$ and $Z$ stand for "match" and "zero", respectively. $D$, $H$, and $V$ stand for "diagonal", "horizontal", and "vertical", respectively. $T$, $P$, $Z$, and $M$ stand for "plus two", "plus one", "zero", and "minus one", respectively. When a cell refers to a value out of bounds, such as $HP_{i,0}$, its value is not really important.

The above information clearly represents the cells of matrix $C$. For example,

$$C_{i,j} = \sum_{r=1}^{i}(2 \times VT_{r,j} + 1 \times VP_{r,j} - 1 \times VM_{r,j})$$

The next step is to derive logical properties that relate those bit matrices, so as to permit an efficient bit-parallel implementation.

$DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee HT_{i-1,j}$ :

It is clear that if either $A_i = B_j$, $C_{i,j-1} = C_{i-1,j-1}+2$, or $C_{i-1,j} = C_{i-1,j-1}+2$, then $C_{i,j} = C_{i-1,j-1} + 1$. Moreover, if none of them hold, there is no way for $C_{i,j}$ to get the value $C_{i-1,j-1} + 1$.

$DZ_{i,j} \equiv \sim DP_{i,j} \wedge (Z_{i-1,j-1} \vee VP_{i,j-1} \vee HP_{i-1,j})$ :

From the score recurrence we can easily derive the rule that $C_{i,j} = C_{i-1,j-1}$ if and only if $C_{i,j} \neq C_{i-1,j-1} + 1$ and $\max(0, C_{i,j-1} - 1, C_{i-1,j} - 1) = C_{i-1,j-1}$. Moreover, since $0 \leq C_{i-1,j-1}$ and the condition $C_{i,j} \neq C_{i-1,j-1} + 1$ implies that $C_{i,j-1} < C_{i-1,j-1}+2$ and $C_{i-1,j} < C_{i-1,j-1}+2$, it turns out that already $C_{i-1,j-1} \geq \max(0, C_{i,j-1} - 1, C_{i-1,j} - 1)$, so the condition $\max(0, C_{i,j-1} - 1, C_{i-1,j}-1) = C_{i-1,j-1}$ can be changed into the form $C_{i-1,j-1} \in \{0,\ C_{i,j-1}-1,\ C_{i-1,j} - 1\}$. This results in the above formula for $DZ_{i,j}$.

$DM_{i,j} \equiv \sim (DP_{i,j} \vee DZ_{i,j})$ : As it is the only remaining choice.

$HT_{i,j} \equiv DP_{i,j} \wedge VM_{i,j-1}$ :

From now on we build on $D*$ and the other bit matrices, by exhaustively examining all the choices for $C_{i,j} - C_{i-1,j-1}$ using submatrices where the lower right cell is $C_{i,j} = x$ and the upper left can thus have a value $x - 1$, $x$ or $x + 1$. The lower left cell is $C_{i,j-1}$, which in this particular item must have the value $x - 2$. We discard cases that are not possible according to Lemma 1 and express the remaining cases as logical conditions. We put "$\times$" in the remaining corner to signal impossible cases.

| | |
|---|---|
| $x-1$ | |
| $x-2$ | $x$ |

| | |
|---|---|
| $x$ | $\times$ |
| $x-2$ | $x$ |

| | |
|---|---|
| $x+1$ | $\times$ |
| $x-2$ | $x$ |

$HP_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (DZ_{i,j} \wedge VM_{i,j-1})$ :

| | |
|---|---|
| $x-1$ | |
| $x-1$ | $x$ |

| | |
|---|---|
| $x$ | |
| $x-1$ | $x$ |

| | |
|---|---|
| $x+1$ | $\times$ |
| $x-1$ | $x$ |

$HM_{i,j} \equiv VT_{i,j-1} \vee (DZ_{i,j} \wedge VP_{i,j-1}) \vee (DM_{i,j} \wedge VZ_{i,j-1})$ :

| $x-1$ | |
|---|---|
| $x+1$ | $x$ |

| $x$ | |
|---|---|
| $x+1$ | $x$ |

| $x+1$ | |
|---|---|
| $x+1$ | $x$ |

Note the simplification in the first condition since $VT_{i,j-1} \Rightarrow DP_{i,j}$.

$HZ_{i,j} \equiv \sim (HT_{i,j} \vee HP_{i,j} \vee HM_{i,j})$ : As it is the only remaining choice.

$VT_{i,j} \equiv DP_{i,j} \wedge HM_{i-1,j}$**:**

Now we focus on the upper right corner.

| $x-1$ | $x-2$ |
|---|---|
| | $x$ |

| $x$ | $x-2$ |
|---|---|
| $\times$ | $x$ |

| $x+1$ | $x-2$ |
|---|---|
| $\times$ | $x$ |

$VP_{i,j} \equiv (DP_{i,j} \wedge HZ_{i-1,j}) \vee (DZ_{i,j} \wedge HM_{i-1,j})$ :

| $x-1$ | $x-1$ |
|---|---|
| | $x$ |

| $x$ | $x-1$ |
|---|---|
| | $x$ |

| $x+1$ | $x-1$ |
|---|---|
| $\times$ | $x$ |

$VM_{i,j} \equiv HT_{i-1,j} \vee (DZ_{i,j} \wedge HP_{i-1,j}) \vee (DM_{i,j} \wedge HZ_{i-1,j})$ :

| $x-1$ | $x+1$ |
|---|---|
| | $x$ |

| $x$ | $x+1$ |
|---|---|
| | $x$ |

| $x+1$ | $x+1$ |
|---|---|
| | $x$ |

$VZ_{i,j} \equiv \sim (VT_{i,j} \vee VP_{i,j} \vee VM_{i,j})$ : As it is the only remaining choice.

## 3. A Bit-Parallel Algorithm

Up to now we have focused on how to compute the $C$ matrix without regard for which should be the output of the algorithm. As explained, computational biologists are interested in matrix positions where the local score exceeds some threshold $k$. Those positions are then subject of further analysis.

Hence our algorithm will receive two strings $A$ and $B$, as well as a threshold value $k$, and will point out all the positions $(i, j)$ of matrix $C$ where the score of the local alignment between $A_{\ldots i}$ and $B_{\ldots j}$ is at least $k$, that is, where $C_{i,j} \geq k$.

The idea of the bit-parallel algorithm is to process $C$ column by column (just like the standard dynamic programming algorithm). However, the bit-parallel algorithm will process all the column in one shot, not row by row. In this section we assume $m \leq w$, that is, we can pack all bits of a column $G_j = G_{1\ldots m,j}$ in a single computer word, for any matrix $G$. Note that row zero is not represented. When needed, the $i$th bit of vector $G_j$ will be written as $G_j(i) = G_{i,j}$.

Therefore, our computation will proceed with column bit vectors $DP_j$, $DM_j$, $DZ_j$, and so on, for $j = 0 \ldots n$, each packed in a computer word. After step $j$ of the algorithm, the vectors will hold the bits corresponding to column $j$ of the matrix.

We will use common $C$ instructions to handle bits: "&" as the bitwise-and, "|" as the bitwise-or, "^" as the bitwise-xor, "~" as the bitwise-not, and "<<" to shift all the bits one position to the left and enter a zero at the rightmost position. We will also treat bit vectors as integers and perform arithmetic operations on them.

In a precomputation step, explained in Section 3.1, the "match" matrix $M$ is built in a suitable way for bit-parallel processing. The boundary conditions of matrix $C$ are handled by giving the proper values to $Z_0$ and $V*_0$ vectors, namely $VP_0 = VM_0 = VT_0 = 0$ and $Z_0 = VZ_0 = 2^m - 1$. Then we process the characters of $B$ (matrix columns) one by one. Each step $j$ computes the bit vectors for column $j$ from the vectors of column $j-1$. First, the diagonal vectors $D*_j$ as well as the horizontal vector $HP_j$ are computed. Vector $HP_j$ is computed already at this stage as we use it in computing $DZ_j$. This part is complex and is explained in Section 3.2. Then the rest of the horizontal and vertical vectors $H*_j$ and $V*_j$ are easy to compute, as explained in Section 3.3. Finally, in Section 3.4, we show how to find and report high enough scores in column $j$, and how the same mechanism handles also computing vector $Z_j$. The way this last part is done is again slightly complicated and uses a technique that is rather different from all the rest.

### *3.1. Computing Matrix M*

Matrix $M$ is represented as a table indexed by alphabet characters. $M[c]$ is a bit vector such that $M[c](i) = 1$ iff $A_i = c$. This table is precomputed before filling matrix $C$. This way the cell value $M_{i,j}$ is actually represented by $M[B_j](i)$.

Matrix $M$ is precomputed in $O(m+|\Sigma|)$ time, where $\Sigma$ is the alphabet of $A$ and $B$, as follows. First initialize $M[c] \leftarrow 0$ for every $c \in \Sigma$ and then traverse string $A$ character-wise, setting bit $M[A_i](i) \leftarrow 1$.

### *3.2. Computing Vectors D*j and HPj*

Let us start with $DP_j$. As seen in Section 2, $DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee HT_{i-1,j}$. Since we are computing all the values at column $j$ in one shot, component $HT_{i-1,j}$ is troublesome because it is not yet computed ($M_{i,j} = M[B_j](i)$ is known so it is not problematic). Let us expand $HT_{i-1,j}$ using its definition:

$$DP_{i,j} \equiv M_{i,j} \vee VT_{i,j-1} \vee (DP_{i-1,j} \wedge VM_{i-1,j-1})$$

where now the problematic value belongs to the same $DP$ column. Let us express this recurrence in vector form. We define temporary vectors $X(i) \equiv M[B_j](i) \vee VT_{j-1}(i)$ and $Y(i) \equiv VM_{j-1}(i)$. Then the recurrence for vector $DP_j$ is

$$DP_j(i) \equiv X(i) \vee (DP_j(i-1) \wedge Y(i-1))$$

This particular kind of circular dependency has already been solved by Myers [9] in his simpler formulation for edit distance computation. Following Hyyrö's explanation [6, 11], we unroll $DP_j(i-1)$ to obtain

$$DP_j(i) \equiv X(i) \vee (X(i-1) \wedge Y(i-1)) \vee (DP_j(i-2) \wedge Y(i-1) \wedge Y(i-2))$$

and unrolling repeatedly we obtain

$$DP_j(i) \equiv \vee_{r=0}^{i} \left(X(i-r) \wedge \left(\wedge_{s=i-r}^{i-1} Y(s)\right)\right)$$

that is, any bit set in $X$ before position $i$ can propagate through a sequence of bits set in $Y$ that reach position $i-1$, so as to set position $i$ in $DP_j$. Myers [9] has shown that the above formula can be computed using bit-parallelism as follows:

$X \leftarrow M[B_j] \mid VT_{j-1}$
$Y \leftarrow VM_{j-1}$
$DP_j \leftarrow ((Y + (X \,\&\, Y)) \,\hat{}\, Y) \mid X$

Let us now consider $DZ$. From Section 2 we have

$$DZ_{i,j} \;\equiv\; \sim DP_{i,j} \;\wedge\; (Z_{i-1,j-1} \;\vee\; VP_{i,j-1} \;\vee\; HP_{i-1,j})$$

where the value $HP_{i-1,j}$ is currently unknown. But it turns out that vector $HP_j$ can be computed once the vector $DP_j$ is known. In Section 2 we gave the formula

$$HP_{i,j} \;\equiv\; (DP_{i,j} \;\wedge\; VZ_{i,j-1}) \;\vee\; (DZ_{i,j} \;\wedge\; VM_{i,j-1})$$

for it. If we look at the situation where the condition $DZ_{i,j} \;\wedge\; VM_{i,j-1}$ is true, we can have $C_{i,j} = x$ only if $C_{i-1,j} = x+1$, that is, only if $HP_{i-1,j}$ is true. Also, $DP_{i,j}$ must obviously be false. Hence, $DZ_{i,j} \wedge VM_{i,j-1} \Rightarrow HP_{i-1,j} \wedge VM_{i,j-1} \wedge \sim DP_{i,j}$. Moreover, it is straightforward to see that the condition $DZ_{i,j} \;\wedge\; VM_{i,j-1}$ is true whenever $HP_{i-1,j} \;\wedge\; VM_{i,j-1} \wedge \sim DP_{i,j}$ is true, and thus we have the following alternative formula for $HP_{i,j}$:

$$HP_{i,j} \;\equiv\; (DP_{i,j} \;\wedge\; VZ_{i,j-1}) \;\vee\; (HP_{i-1,j} \;\wedge\; VM_{i,j-1} \wedge \sim DP_{i,j})$$

The circular dependency on $HP_j$ can be solved in similar fashion as in the case of computing vector $DP_j$. In this case, defining temporary vectors $X$ and $Y$ such that $X(i) \equiv DP_j(i) \wedge VZ_{j-1}(i)$ and $Y(i) \equiv VM_{j-1}(i+1) \wedge \sim DP_j(i+1)$, the preceding formula for $HP_{i,j}$ gets the vector form

$$HP_j(i) \;\equiv\; X(i) \;\vee\; (HP_j(i-1) \;\wedge\; Y(i-1))$$

which is identical to the previous circular dependency for computing $DP_j$. We get immediately the following bit-parallel formula for computing $HP_j$:

$X \leftarrow DP_j \,\&\, VZ_{j-1}$
$Y \leftarrow (VM_{j-1} \,\&\, \sim DP_j) >> 1$
$HP_j \leftarrow ((Y + (X \,\&\, Y)) \,\hat{}\, Y) \mid X$

Once vector $HP_j$ is available, computing the vector $DZ_j$ becomes easy: a straightforward conversion of its formula leads into the following bit-parallel code.

$DZ_j \leftarrow \sim DP_j \,\&\, (((Z_{j-1} << 1) \mid 1) \mid VP_{j-1} \mid (HP_j << 1))$

where, after the shift of $Z_{j-1}$ we have introduced a "1" at its lowest bit to reflect the fact that $C_{0,j-1} = 0$ (that is, $Z_{0,j-1} = 1$) for any $j$ (recall that row zero of $Z$ is not represented). Similarly, $HP_{0,j} = 0$ because $C_{0,j} - C_{0,j-1} = 0 \neq 1$, so we leave the new rightmost bit in zero after shifting $HP_j$.

The above way for computing $HP_j$ and $DZ_j$ involves a right-shift, which may be inconvenient in practice (see Section 4.1). We have found a slightly indirect, but

in fact more efficient, alternative that avoids using right-shift. Let us consider $HP'_j$, a slightly modified version of $HP_j$, that is defined by the following formula.

$$HP'_{i,j} \equiv (DP_{i,j} \wedge VZ_{i,j-1}) \vee (HP'_{i-1,j} \wedge VM_{i,j-1})$$

The only difference to the previously shown alternative formula for $HP_{i,j}$ is that "$\wedge \sim DP_{i,j}$" has been omitted from the right side. We claim that we may use $HP'_{i,j}$ instead of $HP_{i,j}$ in the formula for $DZ_{i,j}$.

**Lemma 2**

$$DZ_{i,j} \equiv \sim DP_{i,j} \wedge (Z_{i-1,j-1} \vee VP_{i,j-1} \vee HP'_{i-1,j})$$

**Proof.** Let us consider the value $HP_{i-1,j}$ versus $HP'_{i-1,j}$, which makes the above formula different from the original. It is clear from the definitions of $HP_{i,j}$ and $HP'i,j$ that $HP_{i-1,j} \equiv HP'_{i-1,j}$ in all other cases except the single case where $HP'_{i-2,j} \wedge VM_{i-1,j-1} \wedge DP_{i-1,j} \equiv 1$. We may assume that $C_{i-1,j-1} = x$. When $VM_{i-1,j-1} \wedge DP_{i,j} \equiv 1$, we know that $C_{i-2,j-1} = x+1$ and $C_{i-1,j} = x+2$. Now Lemma 1 ensures that $C_{i,j} \geq C_{i-1,j} - 1 = x+2-1 = x+1$, and further that $C_{i,j} \leq C_{i-1,j-1} + 1 = x+1$. Hence $C_{i,j} = x+1 = C_{i-1,j-1}+1$. This means that $DP_{i,j} \equiv 1$, $\sim DP_{i,j} \equiv 0$, and $DZ_{i,j} \equiv 0$. We see that the claimed formula is correct also in this case as its value is 0 when $\sim DP_{i,j} \equiv 0$. □

We find it efficient to compute the values $HP'_{i,j}$ in a "preshifted" form, as the formula for $DZ_{i,j}$ uses the value $HP'_{i-1,j}$. Let $U$ be such an auxiliary matrix that $U_{i,j} = HP'_{i-1,j}$. Its formula is as follows.

$$U_{i,j} \equiv HP'_{i-1,j} \equiv (DP_{i-1,j} \wedge VZ_{i-1,j-1}) \vee (U_{i-1,j} \wedge VM_{i-1,j-1})$$

The circular dependency on this preshifted $HP'_j$ can again be solved in the same way as in the case of computing vector $DP_j$. Now we define the temporary vectors $X(i) \equiv DP_j(i-1) \wedge VZ_{j-1}(i-1)$ and $Y(i) \equiv VM_{j-1}(i)$. This gives the following vector form for the above formula of preshifted $HP'_{i,j}$.

$$U_j(i) \equiv X(i) \vee (U_j(i-1) \wedge Y(i-1))$$

Now $U_j$ has the following bit-parallel formula.

$X \leftarrow (DP_j \;\&\; VZ_{j-1}) << 1$
$U_j \leftarrow ((VM_j + (X \;\&\; VM_j)) \wedge VM_j) \mid X$

Then we modify the previous bit-parallel formula for $DZ_j$ to use $U_j$.

$DZ_j \leftarrow \sim DP_j \;\&\; (((Z_{j-1} << 1) \mid 1) \mid VP_{j-1} \mid U_j)$

Once $DZ_j$ is available, $HP_j$ may be computed using its original formula. The translation to bit-parallel code is straightforward.

$HP_j \leftarrow (DP_j \;\&\; VZ_{j-1}) \mid (DZ_j \;\&\; VM_{j-1})$

We can record the value $(DP_j \;\&\; VZ_{j-1})$ when computing $U_j$, so that we do not need to compute it again when computing $HP_j$. This way the alternative solution makes one less operation than the first method for computing $HP_j$ and $DZ_j$.

Finally, we have the following simple bit-parallel formula for $DM_j$.

$$DM_j \leftarrow \sim(DP_j \mid DZ_j)$$

### *3.3. Computing Other Vectors $H*_j$ and $V*_j$*

Once $DP_j$, $HP_j$, $DM_j$, and $DZ_j$ have been computed for the current column $j$, the rest flows easily by following the formulas used in Section 2. Again, when we shift a bit vector to the left, we add or not a "1" bit at the rightmost position depending on which is the value of that vector at the unrepresented row zero.

$$\begin{aligned}
HT_j &\leftarrow DP_j \mathbin{\&} VM_{j-1} \\
HM_j &\leftarrow VT_{j-1} \mid (DZ_j \mathbin{\&} VP_{j-1}) \mid (DM_j \mathbin{\&} VZ_{j-1}) \\
HZ_j &\leftarrow \sim(HT_j \mid HP_j \mid HM_j) \\
VT_j &\leftarrow DP_j \mathbin{\&} (HM_j << 1) \\
VP_j &\leftarrow (DP_j \mathbin{\&} ((HZ_j << 1) \mid 1)) \mid (DZ_j \mathbin{\&} (HM_j << 1)) \\
VM_j &\leftarrow (HT_j << 1) \mid (DZ_j \mathbin{\&} (HP_j << 1)) \mid (DM_j \mathbin{\&} ((HZ_j << 1) \mid 1)) \\
VZ_j &\leftarrow \sim(VT_j \mid VP_j \mid VM_j)
\end{aligned}$$

### *3.4. Keeping Scores and Computing Vector $Z_j$*

Once the bit vectors for column $j$ have been computed, we check whether some cell values in column $j$ of matrix $C$ exceed the matching threshold $k$. At the same time it is also convenient to check which cells have the value zero and record those positions into vector $Z_j$. Unfortunately the differential information of the bit vectors does not allow us to make this in any simple and fast way. The naive approach would be to use the difference information between adjacent cell values to compute and check the cell values $C_{1\ldots m,j}$. This would take $O(m)$ time per column, making the overall running time $O(mn)$, the same as with classical dynamic programming.

On the other hand, as shown by Myers [9], a single value $C_{i,1\ldots n}$ can be tracked in constant time per column by using the horizontal vectors $H*_j$. The problem is that we need to track all the rows $i$, falling again to $O(m)$ time per column.

Our approach is to set up multiple *witnesses* into a single bit vector, and then scan the column in parallel with the witnesses. Each witness will be associated with some $i$ and keep track of the cell values $C_{i,1\ldots n}$, that is, the cell values on row $i$ of $C$. A somewhat similar method was used in [7] as part of an approximate string matching algorithm.

Let $MW_j$ be a length-$m$ bit vector that holds the multiple witnesses at column $j$ and let $Q$ denote the number of bits taken by each witness. Then $MW_j$ can hold $r = \lfloor m/Q \rfloor$ witnesses. Let $MW_j\{i\}$ denote a witness that has its first bit in position $i$ of $MW_j$. $MW_j\{i\}$ occupies the bits $MW_j(i \ldots i+Q-1)$ and keeps track of the cell values on row $i$ of $C$. The first witness is always $MW_j\{1\}$, and the rest are spread evenly into $MW_j$. This can be done in such manner that the largest empty gap after the region of any witness is $\lceil (m-rQ)/r \rceil$. Let us define $Q' = Q + \lceil (m-rQ)/r \rceil$, that is, $Q'$ gives the maximum distance between the first bit of a witness and the first bit of the next witness or, for the last witness, the position after the last bit of the whole vector.

Assume that $C_{i,j} = x$ and the witness $MW_j\{i\}$ exists. For reasons that become clear below, we record the value $x$ into $MW_j\{i\}$ in the form $2^{Q-1} - x$. To guarantee that the witnesses can represent all possible score values from zero to $\min(m, n)$, the parameter $Q$ is determined as the minimum number for which $2^{Q-1} \geq \min(m, n)$, that is, $Q = \lceil \log_2 \min(m, n) \rceil + 1$. Figure 1 exemplifies (vectors $S$, $E$, $K$ will be introduced soon).

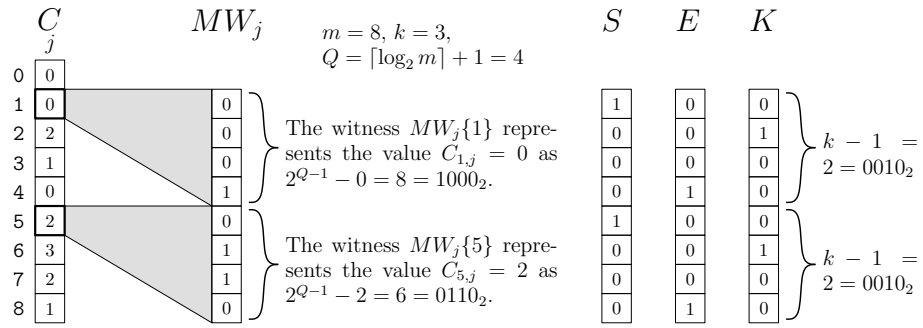


Fig. 1. Example of usage of $MW$, $S$, $E$, and $K$ vectors.

With these conventions the witnesses have the following properties:

**(1)** The $Q$th bit of $MW_j\{i\}$ is set if and only if $C_{i,j} = 0$.

**(2)** Adding some value $x$ to $C_{i,j}$ corresponds to subtracting $x$ from $MW_j\{i\}$, and vice versa.

**(3)** If we add $k - 1$ to $MW_j\{i\}$, then the $Q$th bit of $MW_j\{i\}$ is set if and only if $C_{i,j} < k$.

The witnesses are initialized to $MW_0\{i\} = 2^{Q-1}$ since all values in column 0 of $C$ are zero. After that the witness values are computed by using the horizontal vectors. For example, if $MW_{j-1}\{i\} = x$ and the $i$th bit of $HT_j$ is set, then $MW_j\{i\} = MW_{j-1}\{i\} - 2 = x - 2$ (note that we subtracted the $+2$ due to property (2)). When $MW_{j-1}$ and the horizontal vectors $H*_j$ are available, all witnesses $MW_j\{1\} \ldots MW_j\{r\}$ may be computed in bit-parallel fashion. To achieve this, we use a "start" bit mask $S$ with bits set in those locations that correspond to the first bits of witnesses. Then, the whole witness vector $MW_j$ may be computed as:

$$MW_j \leftarrow MW_{j-1} - 2(HT_j \;\&\; S) - (HP_j \;\&\; S) + (HM_j \;\&\; S)$$

Once $MW_j$ and the vertical vectors $V*_j$ are available, all cell values in column $j$ of $C$ can be scanned in bit-parallel manner. First we copy $MW_j$ into an auxiliary vector $X$. At this stage each witness $MW_j\{i\}$ copied into $X$ represents the value $C_{i,j}$. Then each witness $MW_j$ is updated $Q' - 1$ times. First to represent the value $C_{i+1,j}$, then the value $C_{i+2,j}$, and so on until the value $C_{i+Q'-1,j}$. After $Q' - 1$ iterations, all cells of column $j$ have been scanned (some possibly twice if $Q' \neq Q$). At each stage of the scan we check the current witness values for matches or zeros.

For this we use an "end" bit mask $E \leftarrow S << (Q-1)$ that has a bit set in those positions that correspond to the last bits of the witnesses. In addition we use a bit mask $K$ that holds the value $k-1$ at each witness location.

When the witnesses $MW_j\{i\}$ in $X$ represent the cells $C_{i+h,j}$, the vector $((X + K)\ \&\ E) >> (Q-1-h)$ has bits set in those positions $u$ where $C_{u,j} < k$, and the vector $(X\ \&\ E) >> (Q-1-h)$ has bits set in those positions $u$ where $C_{u,j} = 0$.

Our strategy for checking matches is to record during the scan whether column $j$ contains any matches or not. These may then be checked more carefully, if desired, but if all matching locations are recorded exactly, the running time becomes again $O(mn)$ in the worst case.

The match checking is done by using an auxiliary vector $Y$ that is initialized by setting $Y \leftarrow E$. When $MW_j\{i\}$ represents $C_{i+h,j}$, we set $Y \leftarrow Y\ \&\ (X+K)$. There is at least one match in column $j$ if and only if $Y \neq E$ after the $Q'$ iterations (consisting of the initial stage and $Q'-1$ update stages). The zero vector $Z_j$ is computed by initializing it to zero and setting $Z_j \leftarrow Z_j \mid ((X\ \&\ E) >> (Q-1-h))$ when $MW_j\{i\}$ represents $C_{i+h,j}$.

Figure 2 gives the complete algorithm. We use the alternative solution for computing $HP_j$ and $DZ_j$. Note that, by carefully choosing the update order, we manage to keep only one copy of each vector.

### *3.5. Analysis*

Under the assumption that $m \leq w$, each bit-wise/arithmetic operation in our algorithm takes constant time. In this case computing the $M$ table takes $O(m+|\Sigma|)$ time, and the rest of the algorithm in Figure 2 clearly runs in time $O(nQ')$. Since $Q' < 2Q$ and $Q = \lceil \log_2 \min(m,n) \rceil + 1$, we have that $nQ' = O(n \log \min(m,n))$ and the total running time is $O(|\Sigma| + m + n \log \min(m,n))$.

If the pattern length $m$ is not bounded by $w$, it is straightforward to implement length-$m$ bit vectors by using $\lceil m/w \rceil$ vectors of length $w$. In addition, each bit-wise/arithmetic operation on such bit-vectors can be performed in $O(\lceil m/w \rceil)$ time. This results in the time $O(m + \lceil m/w \rceil |\Sigma|)$ for computing the $M$ table, and the run time of the rest of the algorithm is multiplied by a factor of $O(\lceil m/w \rceil)$, which yields $O(mn \log \min(m,n)/w)$, taking the alphabet size as a constant for simplicity.

## 4. Improving the Algorithm

In this section we discuss how to improve our algorithm both in theory and practice. A practical consideration is to operate on size-$w$ tiles instead of complete length-$m$ bit vectors. We manage to achieve this by bounding the size of the witnesses. This also improves the asymptotic complexity of the original algorithm.

### *4.1. A Practical Tiling Mechanism*

When $m > w$, it is not efficient in practice to implement the algorithm by simply simulating complete length-$m$ bit-vectors: This would result in reading/writing each length-$w$ vector-segment of a bit-vector from/to memory many more times than

**LocalScores** $(A_{1\ldots m},\ B_{1\ldots n},\ k)$

1. $Q \leftarrow \lceil \log_2 \min(m,n) \rceil + 1$
2. $r \leftarrow \lfloor m/Q \rfloor$
3. $S \leftarrow$ distribute evenly $r$ witnesses and mark their first bit
4. $E \leftarrow S << (Q-1)$
5. $K \leftarrow S \times (k-1)$
6. $Q' \leftarrow Q + \lceil (m - rQ)/r \rceil$
7. $MW \leftarrow E$
8. **For** $c \in \Sigma$ **Do** $M[c] \leftarrow 0$
9. **For** $i \in 1 \ldots m$ **Do** $M[A_i] \leftarrow M[A_i] \mid 2^{i-1}$
10. $VP, VM, VT \leftarrow 0, VZ, Z \leftarrow 2^m - 1$
11. **For** $j \in 1 \ldots n$ **Do**
12. $\quad X \leftarrow M[B_j] \mid VT$
13. $\quad DP \leftarrow ((VM + (X \ \& \ VM)) \ \hat{} \ VM) \mid X$
14. $\quad X \leftarrow (DP \ \& \ VZ) << 1$
15. $\quad U \leftarrow ((VM + (X \ \& \ VM)) \ \hat{} \ VM) \mid X$
16. $\quad DZ \leftarrow \sim DP \ \& \ (((Z << 1) \mid 1) \mid VP \mid U)$
17. $\quad HP \leftarrow (DP \ \& \ VZ) \mid (DZ \ \& \ VM)$
18. $\quad DM \leftarrow \sim (DP \mid DZ)$
19. $\quad HT \leftarrow DP \ \& \ VM$
20. $\quad HM \leftarrow VT \mid (DZ \ \& \ VP) \mid (DM \ \& \ VZ)$
21. $\quad HZ \leftarrow \sim (HT \mid HP \mid HM)$
22. $\quad VT \leftarrow DP \ \& \ (HM << 1)$
23. $\quad VP \leftarrow (DP \ \& \ ((HZ << 1) \mid 1)) \mid (DZ \ \& \ (HM << 1))$
24. $\quad VM \leftarrow (HT << 1) \mid (DZ \ \& \ (HP << 1)) \mid (DM \ \& \ ((HZ << 1) \mid 1))$
25. $\quad VZ \leftarrow \sim (VT \mid VP \mid VM)$
26. $\quad MW \leftarrow MW - 2(HT \ \& \ S) - (HP \ \& \ S) + (HM \ \& \ S)$
27. $\quad X \leftarrow MW$
28. $\quad Y \leftarrow E$
29. $\quad Z \leftarrow 0$
30. $\quad$ **For** $h \in 0 \ldots Q' - 1$ **Do**
31. $\quad\quad Z \leftarrow Z \mid ((X \ \& \ E) >> (Q - 1 - h))$
32. $\quad\quad Y \leftarrow Y \ \& \ (X + K)$
33. $\quad\quad X \leftarrow X - 2((VT_j >> h) \ \& \ S) - ((VP_j >> h) \ \& \ S)$
$\quad\quad\quad\quad + ((VM_j >> h) \ \& \ S)$
34. $\quad$ **If** $Y \neq E$ **Then** `Record a match at column` $j$

Fig. 2. Complete bit-parallel algorithm to compute local similarity. Some optimizations have been discarded for clarity.

necessary, which is a considerable slowdown. A better approach is to consider that the matrix $C$ will be covered by a *tiling* of length-$w$ computer words, and each tile will be processed individually. This is similar to the suggestion of Myers [9]. Processing a tile involves the same steps as the basic single-word algorithm from the previous section, but instead of operating on length-$m$ vectors, the vectors involved in the computation will be length-$w$ segments that correspond to the cells covered by the current tile-level. Fig. 3a depicts two typical ways of tiling matrix $C$.

Fig. 3b shows the $h$th length-$w$ tile in column $j$. It covers the cells $C_{(h-1)w+1,j}$, ..., $C_{hw,j}$. Let us use the superscript $(h)$ to denote the $h$th tile so that for example $DP_j^{(h)}$ is a length-$w$ vector that holds the values of the $h$th length-$w$ segment of $DP_j$, i.e. $DP_j^{(h)} = DP_j((h-1)w+1 \ldots hw)$. Here we may assume that the vectors have zero bits in the possible areas that correspond to rows $i > m$. Fig. 3b also shows the left and/or upper neighboring cells/tiles that are involved in processing the tile. The tile to the left, i.e. $h$th tile in column $j-1$, will provide the necessary vector segments $VM_{j-1}^{(h)}$, $VZ_{j-1}^{(h)}$, $VP_{j-1}^{(h)}$ and $VT_{j-1}^{(h)}$. The cell values $C_{(h-1)w,j-1}$ and $C_{(h-1)w,j}$ provide information that enables us to perform the necessary one-step left-shifts of the vectors. The left-shifts bring information from the $w$th row of the $(h-1)$th tile to the first row of the $h$th tile. For example, after left shifting the vector $Z_j^{(h)}$ by one position, we must set its first bit to 1 if $C_{(h-1)w,j} = 0$. As a second example, after left shifting $HT_j^{(h)}$ by one position, we must set its first bit to 1 if $C_{(h-1)w,j} = C_{(h-1)w,j-1} + 2$.

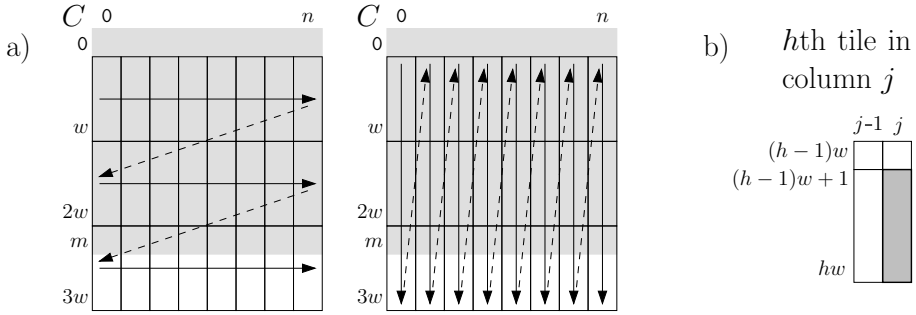


Fig. 3. a) Two tiling orders for processing matrix $C$. b) The $h$th tile in column $j$ (shaded) and the neighboring cells that are involved in processing it.

The tiling order is limited only by the requirement that the above described neighboring information has to be available when processing a given tile. We find it practical to use the leftmost order shown in Fig. 3a. In this case processing the tiles can be done according to the following sketch. Here we will use the variables $ZB$, $HMB$, $HZB$, $HPB$, and $HTB$ to set the first bit of $Z_j^{(h)}$, $HM_j^{(h)}$, $HZ_j^{(h)}$, $HP_j^{(h)}$, and $HT_j^{(h)}$, respectively, after the vector has been shifted left one position. In Section 4.2, we will fill in some missing details such as how to manage the witnesses when operating on individual length-$w$ tiles.

1. Initialize the boundary values of $C$, as well as the vectors $Q$, $E$, and $S$.
2. Repeat steps 3–4 for $h = 1, \ldots, \lceil m/w \rceil$, and then stop.

3. Initialize the relevant values for $h$th tile row. This includes computing array $M$ to contain the $h$th tiles of the length-$m$ match vectors, setting $C^{(0)}_{w,0} = 0$, and initializing $MW^{(h)}$.

4. Repeat steps 4.1–4.4 for $j = 1, \ldots, n$.

   4.1. Read the value $C^{(h-1)}_{w,j}$ and compare it to the value $C^{(h-1)}_{w,j-1}$ that is known from the previous tile or initialization.

   4.2. If $C^{(h-1)}_{w,j} = 0$, set $ZB$ to 1, and otherwise to 0. In similar way, set $HMB$ to 1 if $C^{(h-1)}_{w,j} = C^{(h-1)}_{w,j-1} - 1$, and so on also for $HZB$, $HPB$, and $HTB$.

   4.3. Perform the computations of the basic bit-parallel algorithm in Fig. 2, where each vector contains the $h$th tile of the corresponding length-$m$ vector. In doing this, perform the left shifts in the form $((Z << 1) \mid ZB)$, $((HM << 1) \mid HMB)$, $((HZ << 1) \mid HZB)$, $((HP << 1) \mid HPB)$, and $((HT << 1) \mid HTB)$ in order to set the first bits correctly.

   4.4. After the computation, record the value $C^{(h)}_{w,j}$ for later use by the possible $(h+1)$th tile in column $j$.

### *4.2. Bounding the Witnesses*

An immediate question about using the tiling scheme is how to fit one or more witnesses into a length-$w$ tile in the general case where $\min(m, n)$ is independent of $w$. We achieve this by using delta encoding in storing the values in different length-$w$ tiles of the bit vectors. For the $h$th tile in column $j$, we will use its middle value $C^{(h)}_{\lfloor w/2 \rfloor, j} = C_{(h-1)w+\lfloor w/2 \rfloor, j}$ as the point of comparison. Let $\mu^{(h)}_j$ denote the comparison point $C^{(h)}_{\lfloor w/2 \rfloor, j}$. From Lemma 1 we have that the values $C^{(h)}_{1 \ldots w, j}$, must be in the range $\mu^{(h)}_j - 2\lfloor (w-1)/2 \rfloor \ldots \mu^{(h)}_j + 2\lfloor w/2 \rfloor$.

Consider a witness $MW^{(h)}_j(i) = MW_j((h-1)w + i)$. We will record the value $C^{(h)}_{i,j} = y$ to $MW^{(h)}_j(i)$ in the form $2^{Q-1} - (y - \mu^{(h)}_j)$, and keep track of the values $\mu^{(h)}_j$ separately. Now the values in the witnesses lie in the range $2^{Q-1} - 2\lfloor w/2 \rfloor \ldots 2^{Q-1} + 2\lfloor (w-1)/2 \rfloor$. For reasons that are given in Section 4.2.3, we will choose the value $Q$ to be the minimal choice that can represent the range $-2w + 3 \ldots 2w - 2$ (i.e. witness values $2^{Q-1} - 2w + 2 \ldots 2^{Q-1} + 2w - 3$). This is achieved by setting $Q = \lceil \log_2 \min(m, n, 2w-2) \rceil + 1$.

#### 4.2.1. Setting Up the Witnesses

In the tiling-based scheme, the witnesses will have similar structure at each tiling level $h$, and their regions do not cross the boundaries between different tiling levels. Once $Q$ has been chosen, $MW^{(h)}_j$ will be set to hold $r = \lfloor w/Q \rfloor$ witnesses. There will be room for at least one witness as long as $w \geq 4$. The witnesses are again spread evenly. Now $Q' = Q + \max(0, \lceil (\min(m, w) - rQ)/r \rceil)$ gives the maximum distance between the first bit of a witness and the first bit of the next witness or, for the last witness, the position after the last bit of the whole vector. A difference

to the setup before is that now the witnesses have more than $m$ bits available for them if $m < w$. To ease matters in Section 4.2.3, we position the last witness to be $MW_j^{(h)}(w - Q' + 1)$ if $m > w$: in this case the region of the last witness consists of the last $Q'$ bits of $MW_j^{(h)}$.

Then the vectors $S^{(h)}$, $E^{(h)}$, and $K^{(h)}$ will be initialized in same way as before.

### 4.2.2. Keeping Track of $\mu_j^{(h)}$

It is straightforward to keep track of the value $\mu_j^{(h)}$ by using $HM_j$, $HZ_j$, $HP_j$, and $HT_j$. Initially $\mu_j^{(h)} = C_{\lfloor w/2 \rfloor,0}^{(h)} = 0$. Let $\Delta_j^{(h)}$ denote the change of the comparison point when we move from column $j-1$ to column $j$ on the $h$th tile row. The value $\Delta_j^{(h)} = \mu_j^{(h)} - \mu_{j-1}^{(h)}$ may be computed by setting $\Delta_j^{(h)} = (2(HT_j \ \& \ 2^{\lfloor (w-1)/2 \rfloor}) + (HP_j \ \& \ 2^{\lfloor (w-1)/2 \rfloor}) - (HM_j \ \& \ 2^{\lfloor (w-1)/2 \rfloor})) >> \lfloor (w-1)/2 \rfloor$. After this we set $\mu_j^{(h)} = \mu_{j-1}^{(h)} + \Delta_j^{(h)}$, and also adjust accordingly the witnesses in $MW_j^{(h)}$ by setting $MW_j^{(h)} = MW_j^{(h)} + (\Delta_j^{(h)} \times SM^{(h)})$.

### 4.2.3. Computing $Z_j^{(h)}$, Checking for Matches, and Recording $C_{w,j}^{(h)}$

We follow the same principles as in Section 3.4. But since $MW_j^{(h)}$ now contains values relative to $\mu_j^{(h)}$, we first re-adjust the witnesses to represent their actual values $C_{i,j}^{(h)}$: Instead of $MW_j^{(h)}$, we set the value $MW_j^{(h)} - (\mu_j^{(h)} \times SM^{(h)})$ into the auxiliary vector $X^{(h)}$. This may cause one or more witnesses to overflow, but we simply ignore this for now. Then we perform $Q'$ iterations as depicted in Section 3.4: First $Z_j^{(h)} \leftarrow 0$ and an auxiliary vector $Y^{(h)} \leftarrow E^{(h)}$. During each iteration $Y^{(h)} \leftarrow Y^{(h)} \ \& \ (X^{(h)} + K^{(h)})$ and $Z_j^{(h)} \leftarrow Z_j^{(h)} \mid ((X^{(h)} \ \& \ E^{(h)}) >> (Q - 1 - h))$. After the $Q'$ iterations, we check the validity of $Z_j^{(h)}$ and the auxiliary match checking vector $Y^{(h)}$.

If $\mu_j^{(h)} > 2\lfloor (w-1)/2 \rfloor$, then $C_{i,j}^{(h)} \geq \mu_j^{(h)} - 2\lfloor (w-1)/2 \rfloor > 0$ for $i = 1 \ldots w$ and it is correct to reset $Z_j^{(h)} \leftarrow 0$. On the other hand, if $\mu_j^{(h)} \leq 2\lfloor (w-1)/2 \rfloor$, then $0 \leq C_{i,j}^{(h)} \leq \mu_j^{(h)} + 2\lfloor w/2 \rfloor \leq 2\lfloor (w-1)/2 \rfloor + 2\lfloor w/2 \rfloor = 2w - 2$. As the witnesses can represent the range $-2w+3 \ldots 2w-2$, none of the witnesses have overflown in this case and $Z_j^{(h)}$ has already been computed correctly during the $Q'$ iterations.

In similar way, if $\mu_j^{(h)} \geq k + 2\lfloor (w-1)/2 \rfloor$, then $C_{i,j}^{(h)} \geq k$ and we can declare a match regardless of the value of $Y^{(h)}$. If $k - 2\lfloor w/2 \rfloor \leq \mu_j^{(h)} < k + 2\lfloor (w-1)/2 \rfloor$, then $k - 2w + 2 = k - 2\lfloor w/2 - 2\lfloor (w-1)/2 \rfloor \leq \mu_j^{(h)} - 2\lfloor (w-1)/2 \rfloor \leq C_{i,j}^{(h)} \leq \mu_j^{(h)} + 2\lfloor w/2 \rfloor < k + 2\lfloor (w-1)/2 \rfloor + 2\lfloor w/2 \rfloor = k + 2w - 2$. Match checking uses the vector $X^{(h)} + K^{(h)}$, which has the effect of decrementing the values represented in the witnesses by $k - 1$. Hence if $k - 2\lfloor w/2 \rfloor \leq \mu_j^{(h)} < k + 2\lfloor (w-1)/2 \rfloor$, then $-2w + 3 \leq C_{i,j}^{(h)} - (k-1) < 2w - 1$. Like above, in this case none of the witnesses in $X^{(h)} + K^{(h)}$ are in an overflown state and $Y^{(h)}$ can be used in normal fashion for match checking. Finally, if $\mu_j^{(h)} < k - 2\lfloor w/2 \rfloor$, then $C_{i,j}^{(h)} \leq \mu_j^{(h)} + 2\lfloor w/2 \rfloor < k$

and there is no match.

As the witnesses are set up so that the last witness has a region of $Q'$ bits[a], the last witness in the auxiliary vector $X^{(h)}$ will represent $C^{(h)}_{w,j} = y$ in the form $2^{Q-1} - (y - \mu^{(h)}_j)$ after the $Q'$ iterations. We decode the value by setting $C^{(h)}_{w,j} = \mu^{(h)}_j + 2^{Q-1} - (X^{(h)} >> (w - Q'))$.

Now we have the complete improved algorithm. Figure 4 shows the code for it.

### *4.3. Analysis of the Improved Algorithm*

The manipulated bit vectors have length $w$, so each operation on them is done in constant time. For each tiling level $h$, we compute the $M$ table in $O(\min(m, w)+|\Sigma|)$ time, and then process $n$ tiles. Each tile takes $O(Q') = O(Q) = O(\log \min(m, n, w))$ time. There is a total of $O(m/w)$ tiling levels. By combining these, we have the total running time $O((m/w)(\min(m, w) + |\Sigma| + n \log \min(m, n, w))) = O(m + m|\Sigma|/w + mn \log \min(m, n, w)/w)$. This is $O(mn \log \min(m, n, w)/w)$ for $|\Sigma| = O(n)$

Compared to the best bit-parallel complexity for global and semiglobal similarity (actually, for distance computation), $O(mn/w)$, we have a logarithmic penalty factor because of the use of local similarity. At this point it should be clear that we can compute global and semiglobal scores (rather than distances) within the same $O(mn/w)$ complexity, just by removing the use of vector $Z_j$ and checking the score only at a single cell or a single row. This removes the need for the witnesses and their logarithmic penalty.

## 5. Experimental Results

We implemented a general $O(mn \log \min(m, n, w)/w)$ version of our algorithm and compared it to the plain dynamic programming algorithm. Both algorithms were programmed in C, and we tried to make both implementations as efficient as possible. The test computer was a 64-bit Sparc Ultra 2 with 128 MB RAM, and the codes were compiled with GCC 3.3.1 with optimization switched on. The test strings were randomly selected DNA sequences from the genome of S.cerevisiae (baker's yeast). The test contained two different types of scenarios. In the first we tested with short patterns and a long text. This test involved the matching thresholds $k = 1$ and $k = m - 1$ to see what kind of effect the value of $k$ has. In the second we tested aligning patterns and texts that have the same length, and this time we show only the case $k = m - 1$ (in the first test we found the results to be highly independent on $k$). The results are shown in Fig. 5. They are well in accordance with the asymptotic running time $O(mn \log \min(m, n, w)/w)$.

When $m \leq n$ and $m < w = 64$, we use only $m$ bits in each bit vector and the running time becomes $O(mn \log m/m)$. The experimental results in Fig.5 (left) exhibit how the speedup factor is proportional to $\log m/m$. In this case our algorithm is from roughly 1.2 ($m = 4$) up to 6.2 ($m = 32$) times faster than the basic dynamic programming algorithm.

[a]This is for $m > w$. If $m \leq w$, we may compute $C^{(h)}_{w,j}$ wrong, but the value will never be used.

**ImpLocalScores** $(A_{1\ldots m},\ B_{1\ldots n},\ k)$

1. $Q \leftarrow \lceil \log_2 \min(m, n, 2w-2) \rceil + 1, \quad m' \leftarrow \min(m, w)$
2. $r \leftarrow \lfloor w/Q \rfloor, \quad BB \leftarrow 2^{\lfloor (w-1)/2 \rfloor}$
3. $S \leftarrow$ distribute evenly $r$ witnesses and mark their first bit
4. $E \leftarrow S << (Q-1), \quad K \leftarrow S \times (k-1)$
5. $Q' \leftarrow \max(Q + \lceil (m' - rQ)/r \rceil, Q)$
6. **For** $i \in 1 \ldots n$ **Do** $C[i] \leftarrow 0$
7. **For** $h \in 1 \ldots \lceil m/w \rceil$ **Do**
8. **For** $c \in \Sigma$ **Do** $M[c] \leftarrow 0$
9. **For** $i \in (h-1)w + 1 \ldots \min(m, hw)$ **Do** $M[A_i] \leftarrow M[A_i] \mid 2^{(i-1) \bmod w}$
10. $VP, VM, VT \leftarrow 0, \ VZ, Z \leftarrow 2^w - 1, \ MW \leftarrow E, \ prevC \leftarrow 0, \ \mu \leftarrow 0$
11. **For** $j \in 1 \ldots n$ **Do**
12. **If** $C[j] = 0$ **Then** $ZB \leftarrow 1$ **Else** $ZB \leftarrow 0$
13. **If** $C[j] = prevC - 1$ **Then** $HMB \leftarrow 1$ **Else** $HMB \leftarrow 0$
14. **If** $C[j] = prevC$ **Then** $HZB \leftarrow 1$ **Else** $HZB \leftarrow 0$
15. **If** $C[j] = prevC + 1$ **Then** $HPB \leftarrow 1$ **Else** $HPB \leftarrow 0$
16. **If** $C[j] = prevC + 2$ **Then** $HTB \leftarrow 1$ **Else** $HTB \leftarrow 0$
17. $X \leftarrow M[B_j] \mid VT \mid HTB$
18. $DP \leftarrow ((VM + (X \ \& \ VM)) \ \hat{} \ VM) \mid X$
19. $X \leftarrow ((DP \ \& \ VZ) << 1) \mid HPB$
20. $U \leftarrow ((VM + (X \ \& \ VM)) \ \hat{} \ VM) \mid X$
21. $DZ \leftarrow \sim DP \ \& \ (((Z << 1) \mid ZB) \mid VP \mid U)$
22. $HP \leftarrow (DP \ \& \ VZ) \mid (DZ \ \& \ VM)$
23. $DM \leftarrow \sim (DP \mid DZ), \ HT \leftarrow DP \ \& \ VM$
24. $HM \leftarrow VT \mid (DZ \ \& \ VP) \mid (DM \ \& \ VZ)$
25. $HZ \leftarrow \sim (HT \mid HP \mid HM)$
26. $VT \leftarrow DP \ \& \ ((HM << 1) \mid HMB)$
27. $VP \leftarrow (DP \ \& \ ((HZ << 1) \mid HZB)) \mid (DZ \ \& \ ((HM << 1) \mid HMB))$
28. $VM \leftarrow ((HT << 1) \mid HTB) \mid (DZ \ \& \ ((HP << 1) \mid HPB) \mid (DM \ \& \ ((HZ << 1) \mid HZB))$
29. $VZ \leftarrow \sim (VT \mid VP \mid VM)$
30. $\Delta \leftarrow (2(HT \ \& \ BB) + (HP \ \& \ BB) - (HM \ \& \ BB)) >> \lfloor (w-1)/2 \rfloor$
31. $\mu \leftarrow \mu + \Delta$
32. $MW \leftarrow MW - 2(HT \ \& \ S) - (HP \ \& \ S) + (HM \ \& \ S) + (\Delta \times SM)$
33. $X \leftarrow MW - (\mu \times SM), \ Y \leftarrow E, \ Z \leftarrow 0$
34. **For** $h \in 0 \ldots Q' - 1$ **Do**
35. $Z \leftarrow Z \mid ((X \ \& \ E) >> (Q - 1 - h))$
36. $Y \leftarrow Y \ \& \ (X + K)$
37. $X \leftarrow X - 2((VT_j >> h) \ \& \ S) - ((VP_j >> h) \ \& \ S) + ((VM_j >> h) \ \& \ S)$
38. $prevC \leftarrow C[j]$
39. $C[j] \leftarrow \mu + 2^{Q-1} - (X >> (w - Q'))$
40. **If** $\mu > 2 \times \lfloor (w-1)/2 \rfloor$ **Then** $Z \leftarrow 0$
41. **If** $(Y \neq E$ **and** $\mu \geq k - 2 \times \lfloor w/2 \rfloor)$ **or** $\mu > k + 2 \times \lfloor (w-1)/2 \rfloor$ **Then**
42. `Record a match in the` $h$`th tile of column` $j$

Fig. 4. An improved algorithm to compute local similarity. Some optimizations have been discarded for clarity.

When $m \leq n$ and $m \geq w = 64$, the asymptotic running time is $O(mn \log w/w)$, i.e. the speedup factor becomes fixed to $\log w/w$. The experimental results in Fig.5 (right) exhibit this, as our algorithm is roughly 7.2 times faster when $m \geq 256$. In the case $m = 128$ our algorithm is roughly 8.5 times faster. This deviation is explained by the fact that our algorithm implementation uses the delta encoding described in Section 4.2 only once $\lceil \log_2(2w-2) \rceil < \lceil \log_2 \min(m,n) \rceil$.

Overall the results indicate that our algorithm works well in practice as well as in theory.

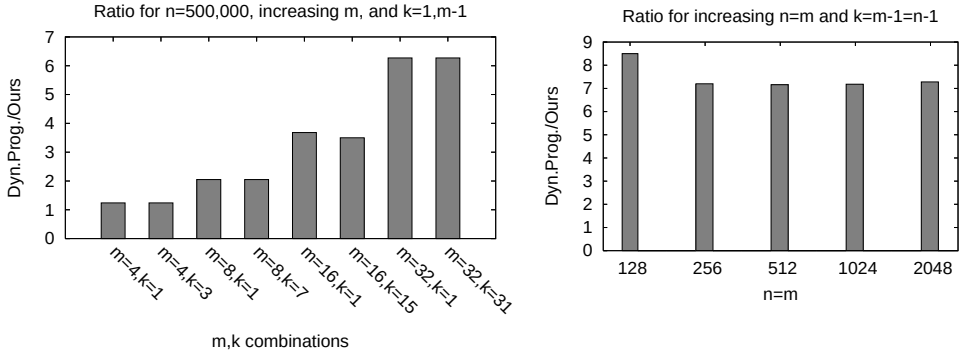


Fig. 5. Speedup factor of our bit-parallel algorithm over the basic dynamic programming algorithm. On the left, aligning long against short strings. On the right, aligning strings of the same length.

## 6. Conclusions

We have presented the first bit-parallel algorithm to compute local similarity score between two strings, which is a common task in computational biology. While dynamic programming, the only existing algorithm, takes time $O(mn)$ ($m$ and $n$ being the lengths of the strings), our algorithm needs time $O(mn \log \min(m,n,w)/w)$ using a computer word of $w$ bits. Our experiments show up to 8-fold speedups.

Our algorithm cannot replace dynamic programming because it cannot handle prize and penalty values other than $\pm 1$. However, there are some DNA-related applications where they use precisely those $\pm 1$ penalties [5]. It is also feasible to use such simplified weights as a fast preliminary filter to discard clearly uninteresting areas of the matrix. Recent research [12] suggests that this is promising.

Several issues are left for future research. An interesting one from the bit-parallel perspective is to investigate whether it is possible to "pack" the logical conditions describing the differences between the matrix cells in a way that makes the overall formula faster to compute. E.g. we use four bit vectors $VT$, $VP$, $VZ$, $VM$, to describe four possible values $+2$, $+1$, $0$, $-1$, whereas two bit vectors could suffice.

Another way to speed up the computation is an adaptive configuration of witnesses. If most values in matrix $C$ are low, one would not really need $\log \min(m,n,w)$ bits to represent them, but rather could process most of the matrix with a denser witness configuration. Say that the values to represent do not (usually) exceed $q$,

then one could use $\log q$ bits per witness, so as to have $m/\log q$ witnesses, obtaining $O(mn \log(q)/w)$ average time. This requires that the algorithm adapts the witness spacing according to the matrix values as the computation progresses.

More ambitious and longer-term goals are accommodating other cost functions apart from the unitary-cost one; and trying to obtain optimal speedup, removing the term $O(\log \min(m, n, w))$ from the cost formula.


### Acknowledgments

The second author was partially funded by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.


### References


1. R. Baeza-Yates and G. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, 35(10):74–82, 1992.
2. A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–65, 2002.
3. M. Crochemore, C. S. Iliopoulos, and Y. J. Pinzon. Speeding-up Hirschberg and Hunt-Szymanski LCS Algorithms. *Fundamenta Informaticae*, 56(1-2):89–103, 2003.
4. M. Crochemore, G. Landau, and M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted scoring matrices. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 679–688, 2002.
5. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
6. H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Dept. of Computer and Information Sciences, University of Tampere, Tampere, Finland, 2001.
7. H. Hyyrö and G. Navarro. Bit-parallel witnesses and their applications to approximate string matching. *Algorithmica*, 41(3):203–231, 2005.
8. W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. of Computer and System Sciences*, 20:18–31, 1980.
9. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.
10. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
11. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.
12. S. Smith. Homology Search with Binary and Trinary Scoring Matrices. *International Journal of Bioinformatics Research and Applications* 2(2):119–131, 2006.
13. H. Tsuji, A. Ishino, and M. Takeda. A Bit-Parallel Tree Matching Algorithm for Patterns with Horizontal VLDC's. In *Proc. 12th String Processing and Information Retrieval (SPIRE'05)*, LNCS 3772, pages 388–398, 2005.
14. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.