A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances[1]

Heikki Hyyrö

Department of Computer and Information Sciences
33014 University of Tampere
Finland

e-mail: `Heikki.Hyyro@uta.fi`

Abstract. The edit distance between strings A and B is defined as the minimum number of edit operations needed in converting A into B or vice versa. The Levenshtein edit distance allows three types of operations: an insertion, a deletion or a substitution of a character. The Damerau edit distance allows the previous three plus in addition a transposition between two adjacent characters. To our best knowledge the best current practical algorithms for computing these edit distances run in time $O(dm)$ and $O(\sigma + \lceil m/w \rceil n)$, where d is the edit distance between the two strings, m and n are their lengths ($m \leq n$), w is the computer word size and σ is the size of the alphabet. In this paper we present an algorithm that runs in time $O(\sigma + \lceil d/w \rceil m)$. The structure of the algorithm is such, that in practice it is mostly suitable for testing whether the edit distance between two strings is within some pre-determined error threshold. We also present some initial test results with thresholded edit distance computation. In them our algorithm works faster than the original algorithm of Myers.

Key words: Levenshtein edit distance, Damerau edit distance, bit-parallelism, approximate string matching

1 Introduction

The desire to measure the similarity between two strings may arise in many applications, like for example computational biology and spelling correction. A common way to achieve this is to compute the edit distance between the strings. Throughout the paper we will assume that A is a string of length m and B is a string of length n, and that $m \leq n$. The edit distance $ed(A, B)$ between strings A and B is defined as the minimum number of edit operations needed in converting A into B or vice versa. In this paper we concentrate on two typical edit distances: the Levenshtein edit distance [Lev66] and the Damerau edit distance [Dam64]. The Levenshtein edit distance allows three edit operations, which are inserting, deleting or substituting a character (Figures 1a, 1b and 1c). In addition to these three, the Damerau edit distance allows also transposing two permanently adjacent characters (Figure 1d). When edit

[1]This work was supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering

distance is used, strings A and B are deemed similar iff their edit distance is small enough, that is iff $ed(A, B) \leq k$, where k is some pre-determined error threshold. A related problem is that of approximate string matching, which is typically defined as follows: let *pat* be a string of length m and *text* a (much longer) string of length n. The task of approximate string matching is then to find all such indices j, for which exists such $h \geq 0$ that $ed(pat, text[j - h..j]) \leq k$.

The oldest, but most flexible in terms of permitting different edit operations and/or edit operation costs, algorithms for computing edit distance (for example [WF74]) are based on dynamic programming and run in time $O(mn)$. Ukkonen [Ukk85a] has later proposed two $O(dm)$ methods, and Myers [Mye86] an $O(n + d^2)$ method. The latter is based on using a suffix tree and is not viewed as being practical (e.g. [Ste94]). With fairly little modifications these methods can also be used in computing the Damerau edit distance without affecting the asymptotic run times.

The methodology of using so-called "bit-parallelism" in developing fast and practical algorithms has recently become popular in the field of string matching. Wu and Manber [WM92] presented an $O(d\lceil m/w \rceil n)$ bit-parallel algorithm for Levenshtein edit distance -based approximate string matching, and in [Nav01] it was modified to compute both Levenshtein and Damerau edit distance. The run time remained the same. Then Baeza-Yates and Navarro presented a method, which enables an $O(\lceil dm/w \rceil n)$ algorithm for the Levenshtein edit distance. Currently this algorithm has not been extended for the Damerau edit distance. Finally Myers [Mye99] has presented an $O(\lceil m/w \rceil n)$ algorithm for approximate string matching under the Levenshtein edit distance. In [Hyy01] the algorithm was extended for computing the Damerau edit distance.

In this paper we will present an initial study on combining one of the $O(dm)$ edit distance algorithms of Ukkonen [Ukk85a] with the bit-parallel algorithm of Myers [Mye99] to obtain a faster algorithm. We begin by reviewing these underlying algorithms in the next section.

2 Preliminaries

In the following discussion let A be a string of length m and B a string of length n. We also use the notation $A[u]$ to denote the uth character of A and the notation $A[u..v]$ to denote the substring of A, which begins from its uth character and ends at its vth character. The superscript R denotes the reverse string: for example if A = "ABC", then A^R = "CBA". For bit operations we use the following notation: '&' denotes bitwise"and", '|' denotes bitwise "or", '∧' denotes bitwise "xor", '∼' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. We refer to the ith bit of the bit vector V as $V[i]$. Bit-positions are assumed to grow from right to left, and we use superscript to denote bit-repetition. As en example let $V = 1001110$ be a bit vector. Then $V[1] = V[5] = V[6] = 0$, $V[2] = V[3] = V[4] = V[7] = 1$, and we could also write $V = 10^2 1^3 0$.

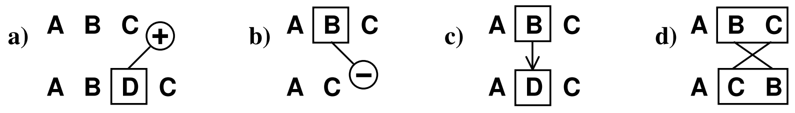

Figure 1: Four different edit operations. Figure a) shows inserting character 'D' between the last two characters of the string "ABC", which results in the string "ABCD". Figure b) shows deleting the character "B", which results in the string "AC". Figure c) shows substituting the character 'B' with the character 'D', which results in the string "ADC". Figure d) shows transposing the characters 'B' and 'C', which results in the string "ACB". Transposition is allowed only between such characters that were adjacent already in the original string.

2.1 Dynamic programming

Computing edit distance is a problem that seems to be most naturally solved with dynamic programming. The value $ed(A, B)$ can be computed by filling an $(m+1) \times (n+1)$ dynamic programming matrix D, in which the cell $D[i, j]$ contains the value $ed(A[1..i], B[1..j])$. The following well-known Recurrence 1 gives the rule for filling D when the Levenshtein edit distance is used.
Recurrence 1

$$D[i,0] = i, D[0,j] = j.$$
$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if } A[i] = B[j]. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ if } A[i] \neq B[j]. \end{cases}$$

The recurrence allows the cells with $i > 0$ and $j > 0$ to be filled in any such order, that the cell values $D[i-1,j]$, $D[i-1,j-1]$ and $D[i,j-1]$ are known at the time the cell $D[i,j]$ is filled. A common way is to use column-wise filling, where each column is filled from top to bottom (Figure 2). The Damerau edit distance can be computed otherwise identically as the Levenshtein edit distance, but using Recurrence 2 [Hyy01] instead in filling the dynamic programming matrix.
Recurrence 2

$$D[i,0] = i, D[0,j] = j.$$
$$D[i,j] = \begin{cases} D[i-1,j-1], & \text{if } A[i] = B[j]. \\ D[i-1,j-1], & \text{if } A[i-1..i] = B^R[j-1..j] \\ & \quad \text{and } D[i-1,j-1] > D[i-2,j-2]. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), & \text{otherwise.} \end{cases}$$

As the basic dynamic programming scheme fills $(m+1)(n+1)$ cells and filling each cell takes a constant number of operations, the algorithm has a run time $O(nm)$. The following two properties hold in the dynamic programming matrix [Ukk85a, Ukk85b]:

-The diagonal property:	$D[i,j] - D[i-1,j-1] = 0$ or 1.
-The adjacency property:	$D[i,j] - D[i,j-1] = -1, 0,$ or 1, and
	$D[i,j]D[i-1,j] = -1, 0,$ or 1.

Even though these rules were initially presented with the Levenshtein edit distance, they can easily be seen to apply also with the Damerau edit distance.

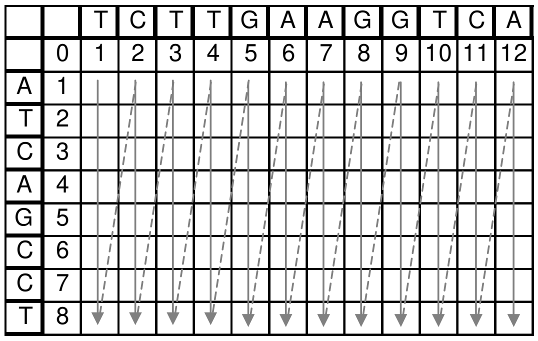

Figure 2: An example of the column-wise filling order for the dynamic programming table of strings "ATCAGCCT" and "TCTTGAAGGTCA".

2.2 Using bit-parallelism

Myers [Mye99] presented an $O(\lceil m/w \rceil n)$ algorithm for approximate string matching under the Levenshtein edit distance. Later in [Hyy01] the algorithm was slightly modified and extended for the Damerau edit distance. Originally these algorithms were designed for approximate string matching, but they can easily be modified to compute edit distance. The algorithms process the jth column of the dynamic programming matrix in $O(\lceil m/w \rceil)$ time by using bit-parallelism. This is done by using delta encoding in the matrix: instead of explicitly computing the values $D[i,j]$ for $i = 1..m$ and $j = 1..n$, the following length-m binary valued delta vectors are computed for $j = 1..n$:

-The vertical positive delta vector:	$VP_j[i] = 1$ iff $D[i,j] - D[i-1,j] = 1$.
-The vertical negative delta vector:	$VN_j[i] = 1$ iff $D[i,j] - D[i-1,j] = -1$.
-The horizontal positive delta vector:	$HP_j[i] = 1$ iff $D[i,j] - D[i,j-1] = 1$.
-The horizontal negative delta vector:	$HN_j[i] = 1$ iff $D[i,j] - D[i,j-1] = -1$.

When the values for these delta vectors are known for the $(j-1)$th column, they can be computed for the jth column in an efficient manner when the following match vector is available for each character λ.

-The match vector PM_λ : $PM_\lambda[i] = 1$ iff $A[i] = \lambda$.

For simplicity we use the notion $PM_j = PM_{B[j]}$ for the rest of the paper. It is straightforward to compute the pattern match vectors in $O(\sigma + m)$ time. In the following we assume that these vectors have already been computed and are readily available.

The delta vectors enable the value $ed(A, B[1..j])$ to be explicitly calculated for $j = 1, 2, \ldots, n$: $ed(A, B[1..j]) = ed(A, B[1..j-1]) + 1$ iff $HP_j[m] = 1$, $ed(A, B[1..j]) = ed(A, B[1..j-1]) - 1$ iff $HN_j[m] = 1$, and $ed(A, B[1..j]) = ed(A, B[1..j-1])$ otherwise. Thus after all n columns are processed, the value $ed(A, B[1..n]) = ed(A, B)$ is known. Figures 3 and 4 show the algorithms based on [Hyy01] for computing the jth column when $m \leq w$, that is, when each vector can be represented by a single bit-vector. Both algorithms are modified to compute edit distance. The algorithm in Figure 3 is for the Levenshtein edit distance, and the algorithm in Figure 4 is for the Damerau edit distance. Both algorithms involve a constant number of operations, and thus compute the delta vectors for the jth column in $O(1)$ time. In this paper we do not separately discuss the case $m > w$. As each required operation for a length-m bit vector can be simulated in $O(\lceil m/w \rceil)$ time using $\lceil m/w \rceil$ length-w bit vectors, the general runtime of the algorithms is $O(\lceil m/w \rceil)$ for each column. This results in a total time of $O(\lceil m/w \rceil n)$ over all n columns in computing $ed(A, B)$.

```
Computing the j th column (Levenshtein distance)
1.    $D0_j \leftarrow (((PM_j \mathbin{\&} VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_j \mid VN_{j-1}$
2.    $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
3.    $HN_j \leftarrow D0_j \mathbin{\&} VP_{j-1}$
4.    If $HP_j \mathbin{\&} 10^{m-1} \neq 0$ Then $D[m,j] \leftarrow D[m,j] + 1$
5.    If $HN_j \mathbin{\&} 10^{m-1} \neq 0$ Then $D[m,j] \leftarrow D[m,j] - 1$
6.    $VP_j \leftarrow (HN_j << 1) \mid \sim (D0_j \mid (HP_j << 1)) \mid 1$
7.    $VN \leftarrow D0_j \mathbin{\&} (HP_j << 1)$
```

Figure 3: Computation of the jth column using a modification of the $D0_j$-based version of the algorithm of Myers (for the case $m \leq w$).

```
Computing the j th column (Damerau distance)
1.    $D0_j \leftarrow (((\sim D0_{j-1}) \mathbin{\&} PM_j) << 1) \mathbin{\&} PM_{j-1}$
2.    $D0_j \leftarrow D0_j \mid (((PM_j \mathbin{\&} VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_j \mid VN_{j-1}$
3.    $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
4.    $HN_j \leftarrow D0_j \mathbin{\&} VP_{j-1}$
5.    If $HP_j \mathbin{\&} 10^{m-1} \neq 0$ Then $D[m,j] \leftarrow D[m,j] + 1$
6.    If $HN_j \mathbin{\&} 10^{m-1} \neq 0$ Then $D[m,j] \leftarrow D[m,j] - 1$
7.    $VP_j \leftarrow (HN_j << 1) \mid \sim (D0_j \mid (HP_j << 1)) \mid 1$
8.    $VN \leftarrow D0_j \mathbin{\&} (HP_j << 1)$
```

Figure 4: Computation of the jth column using a modification of the $D0_j$-based version of the algorithm of Myers with transposition (for the case $m \leq w$).

2.3 Filling only a necessary portion of the matrix

Ukkonen [Ukk85a] presented a method to try to cut down the area of the dynamic programming matrix that is filled. By a q-diagonal we refer to the diagonal, which consists of the cells $D[i, j]$ for which $j - i = q$. From the diagonal and adjacency properties Ukkonen concluded that if $ed(A, B) \leq t$ and $m \leq n$, then it is sufficient to fill only the cells in the diagonals $-\lfloor (t - n + m)/2 \rfloor, -\lfloor (t - n + m)/2 \rfloor + 1, \ldots, \lfloor (t + n - m)/2 \rfloor$ of the dynamic programming matrix. All the other cell values can be assumed to have an infinite value without affecting correct computation of the value $D[m, n] = ed(A, B)$. He used this rule by beginning with $t = (n - m) + 1$ and filling the above-mentioned diagonal interval of the dynamic programming matrix. If the result is $D[m,n] > t$, t is doubled. Eventually $D[m,n] \leq t$, and in this case it is known that $D[m,n] = ed(A,B)$. The run time of this procedure is dominated by the computation involving the last value of t. As this value is $< 2 \times ed(A,B)$ and with each value of t the computation takes $O(t \times \min(m,n))$ time, the overall run time is $O(ed(A,B) \times \min(m,n))$.

In addition Ukkonen proposed a dynamic "cutoff" method to improve the practical performance of the diagonal restriction method. Assume that column-wise order is used in filling the cells $D[i,j]$ inside the required diagonals $-\lfloor(t-n+m)/2\rfloor, -\lfloor(t-n+m)/2\rfloor+1, \ldots, \lfloor(t+n-m)/2\rfloor$. Let r_u hold the diagonal number of the upmost and r_l the diagonal number of the lowest cell that was deemed to have to be filled in the jth column. Then due to the diagonal property we can try to shrink the diagonal region by decrementing r_u as long as $D[r_u, j] > t$ and incrementing r_l as long as $D[r_l, j] > t$. Then at the $(j+1)$th column it is enough to fill the cells in the diagonals $r_l \ldots r_u$. If $r_l > r_u$ the diagonal region vanishes and it is known that $ed(A,B) > t$.

This method of "guessing" a starting limit t for the edit distance and then doubling it if necessary is not really practical for actual edit distance computation. Even though the asymptotic run time is good, it involves large constant factor whenever $ed(A,B)$ is large. But the method works well in practice in thresholded edit distance computation, as then one can immediately set $t = k$ and only a single pass is needed.

3 Our Method

In this section we present a bit-parallel version of the diagonal restriction scheme of Ukkonen, which was briefly discussed in Section 2. In the following we concentrate on the case where the computer word size w is large enough to cover the required diagonal region. Let l_v denote the length of the delta vectors. Then our assumption means that $w \geq l_v = \min(m, \lfloor(t-n+m)/2\rfloor + \lfloor(t+n-m)/2\rfloor + 1)$. Note that in this case each of the pattern match vectors PM_λ may have to be encoded with more than one bit vector: If $m > w$, then PM_λ consists of $\lceil m/w \rceil$ bit vectors.

3.1 Diagonal tiling

The basic idea is to mimic the diagonal restriction method of Ukkonen by tiling the vertical delta vectors diagonally instead of horizontally (Figure 5a). We achieve this by modifying slightly the way the vertical delta vectors VP_j and VN_j are used: Before processing the jth column the vertical vectors VP_{j-1} and VN_{j-1} are shifted one step up (to the right in terms of the bit vector) (Figure 5b). When the vertical vectors are shifted up, their new lowest bit-values $VP_j[l_v]$ and $VN_j[l_v]$ are not explicitly known. This turns out not to be a problem. From the diagonal and adjacency properties we can see that the only situation which could be troublesome is if we would incorrectly have a value $VN_j[l_v] = 1$. This is impossible, because it can happen only if $D0_j$ has an "extra" set bit at position $l_v + 1$ and $HP_j[l_v] = 1$, and these two conditions cannot simultaneously be true.

In addition to the obvious way of first computing VP_j and VN_j in normal fashion and then shifting them up (to the right) when processing the $(j+1)$th column, we propose also a second option. It can be seen that essentially the same shifting effect

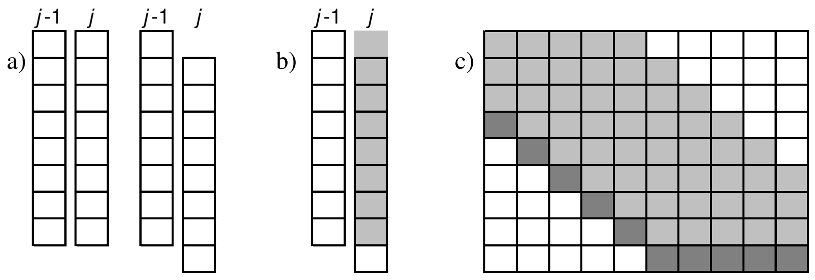

Figure 5: a) Horizontal tiling (left) and diagonal tiling (right). b) The figure shows how the diagonal step aligns the $(j-1)$th column vector one step above the jth column vector. c) The digure depicts in gray the region of diagonals, which are filled according to Ukkonen's rule. The cells on the lower boundary are in darker tone.

can be achieved already when the vectors VP_j and VN_j are computed by making the following changes on the last two lines of the algorithms in Figures 3 and 4:

-The diagonal zero delta vector $D0_j$ is shifted one step to the right on the second last line.

-The left shifts of the horizontal delta vectors are removed.

-The OR-operation of VP_j with 1 is removed.

This second alternative uses less bit operations, but the choice between the two may depend on other practical issues. For example if several bit vectors have to be used in encoding $D0_j$, the column-wise top-to-bottom order may make it more difficult to shift $D0_j$ up than shifting both VP_j and VN_j down.

We also modify the way some cell values are explicitly maintained. We choose to calculate the values along the lower boundary of the filled area of the dynamic programming matrix (Figure 5c). For two diagonally consecutive cells $D[i-1, j-1]$ and $D[i, j]$ along the diagonal part of the boundary this means setting $D[i, j] = D[i-1, j-1]$ if $D0_j[l_v] = 1$, and $D[i, j] = D[i-1, j-1] + 1$ otherwise. The horizontal part of the boundary is handled in similar fashion as in the original algorithm of Myers: For horizontally consecutive cells $D[i, j-1]$ and $D[i, j]$ along the horizontal part of the boundary we set $D[i, j] = D[i, j-1] + 1$ if $HP_j[l_v] = 1, D[i, j] = D[i, j-1] - 1$ if $HN_j[l_v] = 1$, and $D[i, j] = D[i, j-1]$ otherwise. Here we assume that the vector length l_v is appropriately decremented as the diagonally shifted vectors would start to protrude below the lower boundary.

Another necessary modification is in the way the pattern match vector PM_j is used. Since we are gradually moving the delta vectors down, the match vector has to be aligned correctly. This is easily achieved in $O(1)$ time by shifting and OR-ing the corresponding at most two match vectors.

The last necessary modifications concern the first line of the algorithm for the Damerau edit distance in Figure 4. First of all the diagonal delta vector $D0_j$ is shifted down (left), which is not necessary when the vectors are tiled diagonally. Because of similar reason the vector PM_{j-1} has to be shifted one step up (to the right). This means that also the value $PM_{j-1}[l_v + 1]$ will have to be present in the match vector PM_{j-1}. We do not deal with this separately, but assume for now on that $l_v + 1 \leq w$ when dealing with the Damerau edit distance. Another option would be to set the last bit separately, which can be done in $O(1)$ time.

Figures 6 and 7 show the algorithms for computing the vectors at the jth column when diagonal tiling is used. We do not show separate versions for the different cases of updating the cell value at the lower boundary. It is done using one of the previously mentioned ways of using $D0_j$ (diagonal stage) or HP_j and HN_j (horizontal stage).

When $l_v \leq w$, each column of the dynamic programming matrix is computed in $O(1)$ time, which results in the total time being $O(\sigma + n)$ including also time for preprocessing the pattern match vectors. In the general case, in which $l_v > w$, each length-l_v vector can be simulated by using $\lceil l_v/w \rceil$ length-w vectors. This can be done in $O(\lceil l_v/w \rceil)$ time per operation, and therefore the algorithm has in general a run time $O(\sigma + \lceil l_v/w \rceil n)$, which is $O(\sigma + ed(A,B) \times n)$ as $l_v = O(ed(A,B))$. The slightly more favourable time complexity of $O(\sigma + ed(A,B) \times m)$ in the general case can be achieved by simply reversing the roles of the strings A and B: We still have that $l_v = O(ed(A,B))$, but now there is m columns instead of n. In this case the cost of preprocessing the match vectors is $O(\sigma + n)$, but the above complexities hold since $n = O(ed(A,B) \times m)$ when $n > m$.

```
Computing the jth column in diagonal tiling (Levenshtein distance)
1.    Build the correct match vector into $PM_j$
2.    $D0_j \leftarrow (((PM_j \,\&\, VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_j \mid VN_{j-1}$
3.    $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
4.    $HN_j \leftarrow D0_j \,\&\, VP_{j-1}$
5.    Update the appropriate cell value at the lower boundary.
6.    $VP_j \leftarrow HN_j \mid \sim ((D0_j >> 1) \mid HP_j)$
7.    $VN \leftarrow (D0_j >> 1) \,\&\, HP_j$
```

Figure 6: Computation of the jth column with the Levenshtein edit distance and diagonal tiling (for the case $l_v \leq w$).

```
Computing the jth column in diagonal tiling (Damerau distance)
1.    Build the correct match vector into $PM_j$
2.    $D0_j \leftarrow (\sim D0_{j-1}) \,\&\, (PM_j << 1) \,\&\, (PM_{j-1} >> 1)$
3.    $D0_j \leftarrow D0_j \mid (((PM_j \,\&\, VP_{j-1}) + VP_{j-1}) \wedge VP_{j-1}) \mid PM_j \mid VN_{j-1}$
4.    $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
5.    $HN_j \leftarrow D0_j \,\&\, VP_{j-1}$
6.    Update the appropriate cell value at the lower boundary.
7.    $VP_j \leftarrow HN_j \mid \sim ((D0_j >> 1) \mid HP_j)$
8.    $VN \leftarrow (D0_j >> 1) \,\&\, HP_j$
```

Figure 7: Computation of the jth column with the Damerau edit distance and diagonal tiling (for the case $l_v \leq w$).

4 Test Results

In this section we present initial test results for our algorithm in the case of computing the Levenshtein edit distance. We concentrate only on the case where one wants to check whether the edit distance between two strings A and B is below some predetermined error-threshold k. This is because the principle of the algorithm makes it in practice most suitable for this type of use. Therefore all tested algorithms used a scheme similar to the cutoff method briefly discussed in the end of Section 2.3. As a baseline we also show the runtime of using the $O(\lceil m/w \rceil n)$ bit-parallel algorithm of Myers.

The test consisted of repeatedly selecting two substrings in pseudo-random fashion from the DNA-sequence of the baker's yeast, and then testing whether their Levenshtein edit distance is at most k. The computer used in the tests was a 600Mhz Pentium 3 with 256MB RAM and running Microsoft Windows 2000. The code was programmed in C and compiled with Microsoft Visual C++ 6.0 with full optimization. The tested algorithms were:

MYERS: The algorithm of Myers [Mye99] (Section 2.2) modified to compute edit distance. The run time of the algorithm does not depend on the number of errors allowed. The underlying implementation is from the original author.

MYERS(cutoff): The algorithm of Myers using cutoff modified to compute edit distance. The underlying implementation (including the cutoff-mechanism) is from the original author.

UKKA(cutoff): The method of Ukkonen based on filling only a restricted region of diagonals in the dynamic programming matrix and using the cutoff method (Section 2.3).

UKKB(cutoff): . The method of Ukkonen [Ukk85a] based on computing the values in the dynamic programming matrix in increasing order. That is, the method first fills in the cells that get a value 0, then the cells that get a value 1, and so on until the cell $D[m,n]$ gets a value.

OURS(cutoff): Our method of combining the diagonal restriction scheme of Ukkonen with the bit-parallel algorithm of Myers (Section 3). We implemented a similar cutoff method as was used by Hyyrö and Navarro with edit distance computation in their version of the ABNDM algorithm [HN02].

The results ase shown in Figure 8. We tested sequence pairs with lengths 100, 1000 and 10000, and error thresholds of 10%, 20% and 50% of the sequence length (for example $k = 100$, 200 and 500 for the sequence length $m = n = 1000$). It can be seen that in the case of $k = 10$ and $m = 100$ UKKB(cutoff) is the fastest, but in all other tested cases our method becomes the fastest, being 8%-38% faster than the original cutoff method of Myers that is modified to compute edit distance. The good performance of UKKB(cutoff) with a low value of k is not surprising as its expected run time has been shown to be $O(m + k^2)$. [Mye86].

	$m = n = 100$			$m = n = 1000$			$m = n = 10000$		
error limit (%)	10	20	50	10	20	50	10	20	50
UKKA(cutoff)	1,92	5,93	32,6	13,5	52,7	322	13,1	54,9	351
UKKB(cutoff)	1,23	3,02	14,9	6,17	22,9	139	5,57	22,4	146
MYERS(cutoff)	2,46	3,23	4,07	2,47	4,48	15,9	0,71	2,35	13,4
OURS(cutoff)	2,27	2,47	3,32	1,96	3,08	10,5	0,48	1,47	9,03
MYERS	4,24			17,0			14,5		

Figure 8: The results (in seconds) for thresholded edit distance computation between pairs of randomly chosen DNA-sequences from the genome of the baker's yeast. The error threshold is shown as the percentage of the pattern length (tested pattern pairs had equal length). The number of processed sequence pairs was 100000 for $m = n = 100$, 10000 for $m = n = 1000$, and 100 for $m = n = 10000$.

Conclusions and further considerations

In this paper we discussed how bit-parallelism and a diagonal restriction scheme can be combined to achieve an algorithm for computing edit distance, which has an asymtotic run time of $O(\sigma + \lceil d/w \rceil m)$. In practice the algorithm is mostly suitable for checking whether $ed(A, B) \leq k$, where k is a pre-determined error threshold. In this task the initial tests showed our algorithm to be competitive against other tested algorithms [Ukk85a, Mye99], which have run times $O(dm)$ and $O(\sigma + mn/w)$, respectively.

During the preparation of this article we noticed that there seems to be a lack of comprehensive experimental comparison of the relative performance between different algorithms for computing edit distance. Thus we are planning to fill this gap in the near future by composing a fairly comprehensive survey on algorithms for computing edit distance. The survey will also include a more comprehensive test with our algorithm.

We would also like to point out that the algorithm pseudocodes we have shown have not been optimized to remain more clear. Practical implementations could for example avoid shifting the same variable twice and maintain only the needed delta vector values in the memory (the delta vectors in the jth column are only needed when processing the $(j + 1)th$ column).

References

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[HN02] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'2002)*, LNCS 2373, pages 203–224, 2002.

[Hyy01] H. Hyyrö. Explaining and extending the bit-parallel algorithm of Myers. Technical Report A-2001-10, University of Tampere, Finland, 2001.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*

[Mye86] G. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav01] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[Ste94] G. A. Stephen. *String Searching Algorithms.* World Scientific, 1994.

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[WF74] R. Wagner and M. Fisher. The string to string correction problem. *Journal of the ACM*, 21:168–178, 1974.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.