Explaining and Extending the Bit-parallel Approximate String Matching Algorithm of Myers

Heikki Hyyrö

Department of Computer and Information Sciences, University of Tampere, Finland
Heikki.Hyyro@uta.fi

Abstract. The $O(\lceil mn \rceil / w)$, where m is pattern length, n is text length and w is the computer word size, bit-parallel algorithm of Myers [7] is one of the best current algorithms in the case of approximate string matching allowing insertions, deletions and substitutions. We begin this paper by deriving a practically equivalent version of the algorithm of Myers. This is done in a way, which we believe makes the logic behind the algorithm easier to understand than the original presentation. Then we show how to extend the algorithm to allow also a fourth kind of error: transposition of two adjacent characters. This is a very common type of error for example in typed text [5], but has typically been omitted from approximate string matching algorithms. Finally we present experimental results to show what kind of effect adding transposition has on the performance of the algorithm.

1. Introduction

Approximate string matching can be generally defined as searching for substrings of a text that are within a predefined edit distance threshold from a given pattern. Let *Text* = *Text*$[1\ldots n]$ denote a text of length n, *Pat* = *Pat*$[1\ldots m]$ denote a pattern of length n, $ed(\alpha,\beta)$ denote the edit distance between strings α and β, and k be the maximum allowed distance. Typically $ed(\alpha,\beta)$ is defined as the Levenshtein edit distance [6], which is the minimum number of insertions, deletions and substitutions of a single character needed in order to make α and β equal. The task of approximate string matching can be more formally defined as finding from the text all indices j for which $ed(Pat, Text[h\ldots j]) \leq k$ with some $h \leq j$.

Permitting also a fourth edit operation of transposing two adjacent characters was proposed already in a pioneering work by Damerau [2]. This operation is recognized as being important at least when searching typed text [5]. Transposition is permitted to occur between such two adjacent characters, that are permanently adjacent to each other. This means, for example, that we need 3 operations to edit "acb" into "ba", even though it would seem possible to accomplish this with only two operations by first transposing "ba" into "ab" and then adding a "c" in between them (or vice versa, deleting first the "c" and then transposing). Let $edt(\alpha,\beta)$ denote the edit distance between strings α and β when also transposition is allowed. Approximate string matching with the edit distance *edt* is defined in a similar fashion to what was done above in the case of *ed*.

A natural (and therefore also the classic) solution to approximate string matching is the use of dynamic programming, which results in an $O(mn)$ algorithm (e.g. [10]). Even though this basic approach is slow, it is still the most flexible as it is suitable not only for both *ed* and *edt*, but also for many other types of distance metrics. Ukkonen [12] has presented a so-called cut-off heuristic for the dynamic programming algorithm, which results in an $O(kn)$ expected time algorithm.

Recent literature about approximate string matching has concentrated mostly on the distance *ed*. This is perhaps because the distance *ed* is simpler to implement but almost as powerful as the distance *edt*. One exception to this rule is the nrgrep algorithm of Navarro [9], in which the $O(k\lceil m/w \rceil n)$ distance *ed* approximate string matching algorithm of Wu and Manber [13] is extended for the distance *edt*. Navarro has also made an extensive survey [8] of approximate string matching algorithms under the distance *ed*. One of the most notable recent algorithms in this area is the $O(\lceil mn \rceil / w)$ bit-vector algorithm of Myers [7]. In what follows we will first discuss the classic dynamic programming algorithm and the above mentioned algorithm of Myers, and then show how to modify the latter for the distance *edt*.

2. Dynamic programming

To make the notation simpler, let $ed_{\alpha,\beta}(i,j)$ denote $ed(\alpha[1\ldots i],\beta[1\ldots j])$ and define $\alpha[1\ldots 0] = \varepsilon$, where ε denotes the empty string. In addition, let $|\alpha|$ denote the length of the string α. The idea of the dynamic programming algorithm is to start from the trivially known values of type $ed(\alpha[1\ldots i],\varepsilon)$ and $ed(\varepsilon,\beta[1\ldots j])$, and arrive at the value $ed(\alpha,\beta) = ed_{\alpha,\beta}(|\alpha|,|\beta|)$ by recursively computing $ed_{\alpha,\beta}(i,j)$ from previously computed values $ed_{\alpha,\beta}(i\text{-}1,j\text{-}1)$, $ed_{\alpha,\beta}(i\text{-}1,j)$ and $ed_{\alpha,\beta}(i,j\text{-}1)$. This can be done by filling a dynamic programming matrix D using the following well-known recurrence.

Recurrence 1:

$$D[i,0] = i,\ D[0,j] = j.$$

$$D[i,j] = \begin{cases} D[i-1,j-1], \text{if } \alpha[i] = \beta[j]. \\ 1 + \min\{D[i-1,j-1],\ D[i-1,j],\ D[i,j-1]\}, \text{if } \alpha[i] \neq \beta[j]. \end{cases}$$

In this matrix $D[i,j] = ed_{\alpha,\beta}(i,j)$, and so $ed(\alpha,\beta) = D[|\alpha|,|\beta|]$.

Du and Chang [3] have given the following Recurrence 2 for the edit distance *edt*. In this case we denote the dynamic programming matrix by DT, and the superscript R denotes the reverse of a string (that is, if α = "abc", then α^R = "cba").

Recurrence 2:

$DT[i,-1] = DT[-1, j] = \max\{|\alpha|, |\beta|\}$.

$DT[i,0] = i,\ DT[0, j] = j$.

$$DT[i, j] = \begin{cases} DT[i-1, j-1], \text{if } \alpha[i] = \beta[j]. \\ 1 + \min\{DT[i-2, j-2], DT[i-1, j], DT[i, j-1]\}, \text{if } \alpha[i-1...i] = \beta^R[j\text{-}1...j]. \\ 1 + \min\{DT[i-1, j-1], DT[i-1, j], DT[i, j-1]\}, \text{otherwise.} \end{cases}$$

Instead of computing edit distance between strings α and β, the dynamic programming algorithm can be changed to find approximate occurrences of α from β by having an initial condition $D[0,j] = 0$ instead of $D[0,j] = j$ (or $DT[0,j] = 0$ instead of $DT[0,j] = j$). Thus, when $\alpha = Pat$ and $\beta = Text$, the situation corresponds to the earlier definition of approximate string matching. In this case $D[i,j] = \min\{ed(Pat[0\ldots i],Text[h\ldots j]), h \leq j\}$ (or $DT[i,j] = \min\{edt(Pat[0\ldots i], Text[h\ldots j]), h \leq j\}$. From now on we always refer to the versions of D and DT that have been filled in this manner.

Ukkonen ([11, 12]) has studied the properties of the dynamic programming matrix. Among these there were the following two, which apply to both the edit distance and the approximate string matching versions of D:

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

It is fairly straightforward to verify that the same properties hold also for the matrix DT.

The values of the dynamic programming matrix are usually computed by filling it in a column-wise manner, thus effectively scanning the string β (or the text) one character at a time from left to right. At each character the corresponding column is completely filled. This allows us to save space by storing only one or two columns at a time, since the values in the jth column depend only on one (*ed*) or two (*edt*) previous columns.

2.1 The cut-off heuristic for improving the dynamic programming algorithm

Ukkonen proposed a so-called cut-off heuristic [12] to improve the dynamic programming algorithm. A simple consequence of the diagonal property is that if $D[i,j] > k$ (the maximum allowed error), then also $D[i+r, j+r] > k$ for $r > 0$. Suppose we are conducting approximate string matching in a column-wise manner and q is the lowest cell of the (j-1)th column of D, which has a value $\leq k$. Then only the cells $D[1, j]$, $D[2,j]$, …, $D[q+1, j]$ of the jth column need to be computed. We know that $D[i,j] > k$ for $i > q+1$, and so these cells cannot contribute into finding an occurrence. When filling the value $D[q+1,j]$, the possibly unknown value of $D[q+1,j\text{-}1]$ can be ignored. It has been proven by Chang and Lampe [1] that using this method results in an $O(kn)$ expected time algorithm.

3. A slightly modified bit-vector algorithm of Myers

Now we derive a slightly modified version of the bit-vector algorithm of Myers. In the following we assume that $|Pat| = m \leq w$ = size of the computer word. In this case the algorithm runs in time O(n). In pseudo-code we will use C-like notation for bit operations, i.e. '**&**' denotes bit-wise **AND**, '|' bit-wise **OR**, '^' bit-wise **XOR**, '<<' shifting a bit-vector to the left, and '>>' shifting a bit-vector to the right. Both types of shifts are assumed to use zero filling. The bit-positions are assumed to grow from right to left, and we denote bit-repetition with a superscript. Thus for example the bit-vector 01001 has a one in its first and fourth positions, and $1^3 0^2 1 = 111001$.

The first step is to use delta encoding in storing the dynamic programming matrix: Instead of all the actual cell values, the differences between the values of adjacent cells are recorded. Because of the diagonal and adjacency properties, the following vectors can be used in representing the dynamic programming matrix:

-The vertical positive delta vector VP_j: $VP_j[i] = 1$ iff $D[i,j] - D[i\text{-}1,j] = 1$.

-The vertical negative delta vector VN_j: $VN_j[i] = 1$ iff $D[i,j] - D[i\text{-}1,j] = \text{-}1$.

-The horizontal positive delta vector HP_j: $HP_j[i] = 1$ iff $D[i,j] - D[i,j\text{-}1] = 1$.

-The horizontal negative delta vector HN_j: $HN_j[i] = 1$ iff $D[i,j] - D[i,j\text{-}1] = \text{-}1$.

-The diagonal zero delta vector $D0_j$: $D0_j[i] = 1$ iff $D[i,j] = D[i\text{-}1,j\text{-}1]$.

Figure 1 shows an example of these vectors.

		o	n	c	e		u	p	o	n		VP_6	VN_6	HP_6	HN_6	$D0_6$
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**						
o	**1**	0	1	1	1	1	1	1	0	1		1	0	0	0	0
n	**2**	1	0	1	2	2	2	2	1	0		1	0	0	0	0
e	**3**	2	1	1	1	2	3	3	2	1		1	0	1	0	0

Fig. 1. On the left is the dynamic programming matrix for searching the pattern "one" from the text "once upon". On the right are the vectors VP_6, VN_6, HP_6, HN_6 and $D0_6$. The zero row and column are in bold, and the sixth column, which the shown vectors correspond to, is shaded.

Clearly, if we know either both VP_j and VN_j, both HP_j and HN_j, or $D0_j$, for $j = 1\ldots n$, it is possible to recover the value of any cell $D[i,j]$ by starting from a cell value known from the initial conditions of the matrix.

In addition to the above vectors, the algorithm also uses the following pattern match vector PM_λ for each character λ:

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

In the following we use the notation $PM_j = PM_{Text[j]}$. The algorithm imitates column-wise filling of the dynamic programming matrix, and calculates explicitly only the

values $D[m,j]$, for $j = 1\ldots n$. All other cell values are represented implicitly by the earlier defined delta vectors. First VP_0 and VN_0 are initialized according to the initial conditions for the cells $D[i,0]$. This means that $VP_0[i] = 1$ and $VN_0[i] = 0$, for $i = 1\ldots m$. In addition, $D[m,0]$ is initialized to the value m. Then moving from the column j-1 to the column j involves the following four steps:

1. The diagonal vector $D0_j$ is computed from PM_j, VP_{j-1} and VN_{j-1}.
2. The horizontal vectors HP_j and HN_j are computed from $D0_j$, VP_{j-1} and VN_{j-1}.
3. The value $D[m,j]$ is calculated from $D[m,j-1]$ and the horizontal delta values $HP_j[m]$ and $HN_j[m]$.
4. The vertical vectors VP_j and VN_j are computed from $D0_j$, HP_j and HN_j.

An approximate occurrence of the pattern ends at the text position j whenever $D[m,j] \leq k$ during the scan of the text.

Step 1: Computing $D0_j$. Assume that the values $VP_{j-1}[i]$, $VN_{j-1}[i]$ and $PM_j[i]$ are known. From Recurrence 1 for filling D we can see that there is the following three different ways for $D0_j[i]$ to have a value 1.

1. $D[i,j-1] = D[i-1,j-1] - 1$, i.e. $VN_{j-1}[i] = 1$. This enables the zero-difference to propagate from the left using the recurrence option $D[i,j] = D[i,j-1] + 1$.
2. $PM_j[i] = 1$. The zero-difference arises from the equality $Pat[i] = Text[j]$, which sets $D[i,j] = D[i-1,j-1]$.
3. $D[i-1,j] = D[i-1,j-1] - 1$. This enables the zero-difference to propagate from above using the recurrence option $D[i,j] = D[i-1,j] + 1$.

The first and second cases are easy to handle. All we need to do is to set $D0_j[i] = 1$ if $VN_{j-1}[i] = 1$ or/and $PM_j[i] = 1$. This means that the cases 1 and 2 can be treated for the whole vector $D0_j$ by **OR**-ing it with both VN_{j-1} and PM_j.

The third case, however, is the trickiest part of the algorithm. But Myers has presented a nice solution for it. It can be seen that $D[i-1,j] = D[i-1,j-1]$ iff $D[i,j] = D[i-1,j-1]$ and $D[i-1,j] = D[i-2,j-1] = D[i,j] - 1$. This translates into saying that $D[i-1,j] = D[i-1,j-1]$ iff $D0_j[i] = 1$, $D0_j[i-1] = 1$ and $VP_j[i-1] = 1$. On the other hand when $VN_j[i-1] = 0$, $D0_j[i-1] = 1$ iff either case 1 or case 2 applies for the row i-1. This means that $D[i-1,j] = D[i-1,j-1] - 1$ iff $VP_j[i-1] = 1$ and also $PM_j[i-1] = 1$ or $D[i-2,j] = D[i-2,j-1] - 1$. By recursively applying the preceding reasoning for the second term, $D[i-2,j] = D[i-2,j-1] - 1$, of the or, we get that $D[i-1,j] = D[i-1,j-1] - 1$ iff $VP_j[i-1] = 1$ and also $PM_j[i-1] = 1$ or $VP_j[i-2] = 1$ and also $PM_j[i-2] = 1$ or $D[i-3,j] = D[i-3,j-1] - 1$. When we continue in this manner, always expanding the last term of form $D[i-q,j] = D[i-q,j-1] - 1$, we arrive at some $h < i-1$, for which $PM_j[h] = 1$ and the recursion can stop. This is because the initial conditions on the dynamic programming matrix guarantee that $D[0,j] \neq D[0,j-1] - 1$. Thus we have the following rule for the case 3:

$$D[i-1,j] = D[i-1,j-1] - 1 \text{ iff } \exists h\text{: } PM_j[h] = 1 \text{ and } VP_{j-1}[q] = 1, \text{ for } q = h\ldots i-1.$$

The above rule states that $D[i\text{-}1,j] = D[i\text{-}1,j\text{-}1] - 1$ if and only if the (i-1)th bit of the vector $VP_{j\text{-}1}$ belongs to a such run of consecutive one bits, that there is also a match between the character *Text*[j] and some character *Pat*[h] of the pattern, which overlaps the run of consecutive bits above the (i-1)th bit (Figure 2).

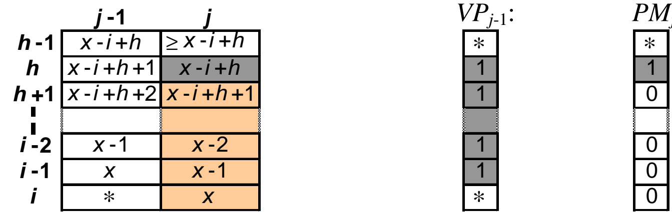

Fig. 2. On the left are rows h-1, $h,\ldots,i$ of the (j-1)th and jth columns of the matrix D. If $D[i\text{-}1,j] = x - 1 = D[i\text{-}1,j\text{-}1] - 1$, then there must be a match between *Text*[j] and some character *Pat*[h] somewhere above the ith row. The vertical delta is positive at least from the (i-1)th row up to the hth row. The corresponding segments of the vectors $VP_{j\text{-}1}$ and PM_j are shown on the right. An asterisk indicates that the corresponding cell/bit may have different values depending on the situation.

Myers noted that the way, in which the run of consecutive bits propagates down a diagonal zero-difference, resembles the carry-effect of integer addition. When $PM_j[h] = 1$ and $VP_{j\text{-}1}[q] = 1$ for $q = h\ldots i\text{-}1$, we know from the previous discussion that $D0_j[q] = 1$ for $q = h\ldots i$. Now if we add $PM_j[h]$ and $VP_{j\text{-}1}[h\ldots i\text{-}1]$ together, the carry effect causes the bits $h\ldots i$-1 of $VP_{j\text{-}1}$ to change from 1 to 0, and the bit i to change either from 1 to 0 or from 0 to 1 depending on its original value. Suppose we **XOR** the bits $h\ldots i$ of the result of the addition $PM_j[h] + VP_{j\text{-}1}[h\ldots i\text{-}1]$ with the original bits $h\ldots i$ of $VP_{j\text{-}1}$. Then the bits $h\ldots i$ will all have the value 1, which is exactly the desired result. From PM_j we can extract only those bits i, for which also $VP_{j\text{-}1}[i] = 1$, by **AND**-ing PM_j with $VP_{j\text{-}1}$. When we then add this vector PM_j **&** $VP_{j\text{-}1}$ together with $VP_{j\text{-}1}$ and **XOR** the result (PM_j **&** $VP_{j\text{-}1}$) + $VP_{j\text{-}1}$ with $VP_{j\text{-}1}$, we get almost the desired result for the whole vector. There are only two differences. One is the situation, in which there are several bits of PM_j that have value 1 inside the same continuous run of ones in $VP_{j\text{-}1}$. This causes the **XOR**-operation to turn off some of these bits, because they will have a value 1 before and after the addition. The second is that the bit, which corresponds to the first match along a consecutive run of ones in $VP_{j\text{-}1}$, will also be set even though the horizontal delta above it is not –1. But neither of these two is a problem in terms of the correctness of the vector $D0_j$, because the corresponding bits will be set anyway when handling the case 2. Figure 3 shows an example.

Putting together all the pieces for the cases 1, 2 and 3, we arrive at the following formula for computing $D0_j$:

$$D0_j = (((PM_j \mathbin{\&} VP_{j\text{-}1}) + VP_{j\text{-}1}) \wedge VP_{j\text{-}1}) \mid PM_j \mid VN_{j\text{-}1}.$$

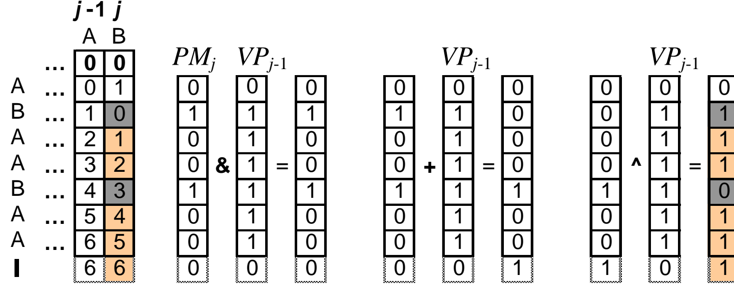

Fig. 3. An example of handling the third case in computing $D0_j$, when $Text[j\text{-}1\ldots j]$ = "AB" and $Pat[1\ldots7]$ = "ABAABAA". As can be seen from the filled column j, a match propagates diagonal zero deltas downwards as long as the vertical delta in the preceding column j-1 has a value +1. First the matching bits in PM_j, that overlap a segment of ones in $VP_{j\text{-}1}$, are extracted by **AND**-ing PM_j and $VP_{j\text{-}1}$. Then the resulting vector is added together with $VP_{j\text{-}1}$ to change the bit value in those positions, which get a diagonal zero delta from above. Finally these changed bits are set to 1 by **XOR**-ing the result of the addition with the original $VP_{j\text{-}1}$. The darker shading marks the locations, where a match causes a diagonal zero delta, and the lighter shading the positions, where a diagonal zero delta propagates from above.

Step 2: Computing HP_j and HN_j. At this point we can assume that, in addition to the vectors $VP_{j\text{-}1}$, $VN_{j\text{-}1}$ and PM_j, also the vector $D0_j$ is known.

It can be seen from the adjacency and diagonal properties that $HP_j[i] = 1$ iff $D[i,j\text{-}1] = D[i\text{-}1,j\text{-}1] - 1$, or $D[i,j] = D[i\text{-}1,i\text{-}1] + 1$ and $D[i,j\text{-}1] = D[i\text{-}1,j\text{-}1]$ (Figures 4a and 4b). In terms of the delta vectors this means that $HP_j[i] = 1$ iff $VN_{j\text{-}1}[i] = 1$ or $D0_j[i] = 0$ and $VP_{j\text{-}1}[i] = 0$ and $VN_{j\text{-}1} = 0$. Because the left side of the preceding or has only the condition $VN_{j\text{-}1}[i] = 1$, the requirement $VN_{j\text{-}1}[i] = 0$ on the right side can be removed as it is implicitly expressed by the former. This results in the following formula for computing the vector $HP_j[i]$:

$$HP_j = VN_{j\text{-}1} \mid \sim(D0_j \mid VP_{j\text{-}1}).$$

In similar fashion as for HP_j, we can see that $HN_j[i] = 1$ iff $D[i,j] = D[i\text{-}1,j\text{-}1]$ and $D[i,j\text{-}1] = D[i\text{-}1,j\text{-}1] + 1$ (Figure 4c). This results in the rule $VN_j[i] = 1$ iff $D0_j[i] = 1$ and $VP_{j\text{-}1}[i] = 1$, and so we have the following formula for computing the vector HN_j:

$$HN_j = D0_j \mathbin{\&} VP_{j\text{-}1}.$$

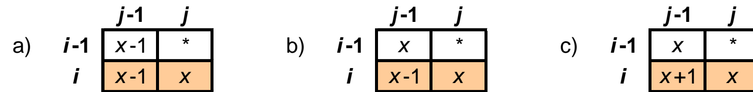

Fig. 4. The figures a) and b) show the only possible combinations for the cells $D[i\text{-}1,j\text{-}1]$, $D[i,j\text{-}1]$ and $D[i,j]$, in which $D[i,j] = x = D[i,j\text{-}1] + 1$. Similarly, figure c) shows the only case, in which $D[i,j] = x = D[i,j\text{-}1] - 1$.

Step 3: Computing the value $D[m,j]$. After computing the vectors HP_j and HN_j, the value $D[m,j]$ is easy to calculate from $D[m,j\text{-}1]$. If $HP_j[m] = 1$, then $D[m,j] = D[m,j\text{-}1] + 1$, and if $HN_j[m] = 1$, then $D[m,j] = D[m,j\text{-}1] - 1$. Otherwise $D[m,j] = D[m,j\text{-}1]$.

Step 4: Computing VP_j and VN_j. This step is diagonally symmetric with step 2 of computing HP_j and HN_j.

By imitating the case of HP_j we have that $VP_j[i] = 1$ iff $HN_j[i\text{-}1] = 1$ or $D0_j[i] = 1$ and $HP_j[i\text{-}1] = 0$. Now we need to align the (i-1)th row bits $HN_j[i\text{-}1]$ and $HP_j[i\text{-}1]$ with the ith row bit $VP_j[i]$. This means shifting the former two one step down (that is, to the left). After shifting these two vectors left, their first bits represent the values $VP_j[0]$ and $VN_j[0]$, which are not explicitly represented in the algorithm. These two values correspond to the difference $D[0,j] - D[0,j\text{-}1]$. Since we assume zero filling, shifting HN_j and HP_j one step to the left introduces a zero in their first positions. This is the same as using the values $HP_j[0] = 0$ and $HN_j[0] = 0$, which correctly corresponds to the initial condition $D[0,j] = 0$ (i.e. $D[0,j] - D[0,j\text{-}1] = 0$) of approximate string matching. If we were to use the algorithm of Myers for computing edit distance, the newly introduced zero bit of the vector HP_j would have to be changed into a one to use the value $HP_j[0] = 1$, which corresponds to the initial condition $D[0,j] = j$ (i.e. $D[0,j] - D[0,j\text{-}1] = 1$) of computing edit distance. The resulting formula for computing the vector VP_j is then:

$$VP_j = (HN_j << 1) \mid \sim(D0_j \mid (HP_j << 1)).$$

By imitating this time the case of HN_j, we have that $VN_j[i] = 1$ iff $D0_j[i] = 1$ and $HP_j[i\text{-}1] = 1$. Again the (i-1)th row bit $HP_j[i\text{-}1]$ has to be shifted one step down to align it with the ith row bit VP_j. The same comment as above, about setting the newly introduced bit of VP_j into a one in the case of computing edit distance, applies also here. We get the following formula for computing the vector VN_j:

$$VN_j = D0_j \;\&\; (HP_j << 1).$$

The algorithms corresponding to steps 1 - 4 is given in Figure 5. In an actual implementation HP_j should be shifted and stored between lines 14 and 15 so that the result can be used both in lines 15 and 16 and one shift is saved. In addition some of the vectors can share the same variable, and only the currently needed values of the difference vectors are kept in memory in a similar fashion to what was discussed about saving space in the case of the dynamic programming matrix.

The only difference between this version and the original algorithm of Myers is that he uses two vectors XV_j and XH_j instead of a single diagonal vector $D0_j$. In fact $D0_j = XV_j$ **OR** XH_j, and XV_j corresponds to the cases 2 and 3 and XH_j to the cases 1 and 2 of the computation of $D0_j$. This difference has no significance as long as regular sequential approximate string matching with the distance *ed* is concerned. But having a single diagonal vector has proved useful for example when the algorithm of Myers is modified for use in the ABNDM algorithm [4].

```
D-Myers(Pat[1…m], Text[1…n], k)
1.      Preprocessing
2.         For λ ∈ all characters Do
3.            PM_λ ← 0
4.         For i ∈ 1…m Do
5.            PM_Pat[i] ← PM_Pat[i] | 0^(m-i)10^(i-1)
6.         VP_0 ← 1^m, VN_0 ← 0, D[m,j] ← m
7.      Searching
8.         For j ∈ 1…n Do
9.            D0_j ← (((PM_Text[j] & VP_j-1) + VP_j-1) ^ VP_j-1) | PM_Text[j] | VN_j-1
10.           HP_j ← VN_j-1 | ~(D0_j | VP_j-1)
11.           HN_j ← D0_j & VP_j-1
12.           If HP_j & 10^(m-1) ≠ 0 Then D[m,j] ← D[m,j] + 1
13.           If HN_j & 10^(m-1) ≠ 0 Then D[m,j] ← D[m,j] - 1
14.           If D[m,j] ≤ k Then report a match ending at Text[j]
15.           VP_j ← (HN_j << 1) | ~(D0_j | (HP_j << 1))
16.           VN_j ← D0_j & (HP_j << 1)
```

Fig. 5. Our $D0_j$-based version of the algorithm of Myers (for the case $m \leq w$).

4. Adding transposition into the algorithm

Now we consider modifying the bit-vector algorithm of Myers to use the edit distance *edt*. The trick we use is to consider how a transposition relates to a zero-difference along the diagonal. Consider the strings α = "abc" and β = "acb". Without allowing transposition we would have D["ab","ac"] = 1, where this one operation corresponds to substituting the first character of the transposable pair. When filling in the value D["abc","acb"], the effect of having done a single transposition can be achieved by allowing a free substitution between the latter character-pair of the transposable pair. This is the same as declaring a match between them. In this way the cost for doing the transposition has already been paid by the substitution of the preceding step. It turns out that this kind of method works correctly in all cases. In general we claim that the following Recurrence 3 for *edt* is in effect equivalent with Recurrence 2 in Section 2.

Recurrence 3:

$$DT[i,0] = i,\ DT[0,j] = j.$$

$$DT[i,j] = \begin{cases} DT[i-1,j-1], \text{if } \alpha[i] = \beta[j]. \\ DT[i-1,j-1], \text{if } \alpha[i-1...i] = \beta^R[j\text{-}1...j] \text{ and } DT[i-1,j-1] = DT[i-2,j-2]+1. \\ 1 + \min\{DT[i-1,j-1],\ DT[i-1,j],\ DT[i,j-1]\}, \text{otherwise.} \end{cases}$$

Now we prove by induction that Recurrence 2 and Recurrence 3 give the same values for DT[i,j] when $i \geq 0$ and $j \geq 0$.

Clearly both formulas give the same value for $DT[i,j]$ when $i = 0$ or 1 or $j = 0$ or 1. Consider now a cell $DT[i,j]$ for some $j > 1$ and $i > 1$, and assume that all previous cells with nonnegative indices have been filled identically by both recursive formulas. Let

x be the value given to $DT[i,j]$ by Recurrence 2 and y be the value given to it by Recurrence 3. The only situation in which the two formulas could possibly behave differently is when $\alpha[i] \neq \beta[j]$ and $\alpha[i\text{-}1\ldots i] = \beta^R[j\text{-}1\ldots j]$. In the followint cases we assume these two conditions to be true.

If $DT[i\text{-}2,j\text{-}2] + 1 = DT[i\text{-}1,j\text{-}1]$, then $y = DT[i,j] = DT[i\text{-}1,j\text{-}1]$, and since the diagonal property requires that $x \geq DT[i\text{-}1,j\text{-}1]$, we have $x = DT[i,j] = DT[i\text{-}2,j\text{-}2] + 1 = DT[i\text{-}1,j\text{-}1] = y$.

Now consider the case $DT[i\text{-}2,j\text{-}2] = DT[i\text{-}1,j\text{-}1]$. Then $y = DT[i,j] = 1 + \min\{DT[i\text{-}1,j\text{-}1], DT[i\text{-}1,j], DT[i,j\text{-}1]\}$ and $x = DT[i,j] = 1 + \min\{DT[i\text{-}2,j\text{-}2], DT[i\text{-}1,j], DT[i,j\text{-}1]\}$, and since $DT[i\text{-}2,j\text{-}2] = DT[i\text{-}1,j\text{-}1]$, we have $x = 1 + \min\{DT[i\text{-}1,j\text{-}1], DT[i\text{-}1,j], DT[i,j\text{-}1]\} = y$.

In both cases, Recurrence 2 and Recurrence 3 assigned the same value for the cell $DT[i,j]$. Therefore we can state by induction that the recurrences are in effect equivalent.

We use here the same notation for the delta vectors regardless of whether the underlying matrix is DT or D. So for example when dealing with transposition, it is assumed that $D0_j[i] = 1$ iff $DT[i,j] = DT[i\text{-}1,j\text{-}1]$.

Following Recurrence 3, it is fairly simple to add transposition into the algorithm of Myers. In addition to the cases when the edit distance ed is used, we also have $D0_j[i] = 1$ if $Pat[i\text{-}1\ldots i] = Text^R[j\text{-}1\ldots j]$ and $DT[i\text{-}1,j\text{-}1] = DT[i\text{-}2,j\text{-}2] + 1$. The condition $Pat[i\text{-}1\ldots i] = Text^R[j\text{-}1\ldots j]$ is true iff $PM_{j\text{-}1}[i] = 1$ and $PM_j[i\text{-}1] = 1$, which can happen only if $j > 1$ and $i > 1$. In this case the second condition $DT[i\text{-}1,j\text{-}1] = DT[i\text{-}2,j\text{-}2] + 1$ is true iff $D0_{j\text{-}1}[i\text{-}1] = 0$. Let TR_j be a transposition vector that corresponds to these conditions. That is, $TR_j[i] = 1$ iff $PM_{j\text{-}1}[i] = 1$, $PM_j[i\text{-}1] = 1$ and $D0_{j\text{-}1}[i\text{-}1] = 0$. The following formula computes the vector TR_j correctly:

$$TR_j = ((\sim D0_{j\text{-}1}) \mathbin{\&} PM_j) << 1) \mathbin{\&} PM_{j\text{-}1}.$$

It follows, that the algorithm shown in Figure 5 can be modified to handle transposition by adding the above formula between the lines 8 and 9, and **OR**-ing $D0_j$ with TR_j before the line 10. This version of the algorithm is shown in Figure 6.

```
TR-D0-Myers(Pat[1…m], Text[1…n], k)
1.      Preprocessing as in Figure 5 except: PM_Text[0] ← 0, DT[m,j] ← m
2.      Searching
3.         For j ∈ 1…n Do
4.            TR_j = ((~D0_j-1) & PM_Text[j]) << 1) & PM_Text[j-1]
5.            D0_j ← (((PM_Text[j] & VP_j-1) + VP_j-1) ^ VP_j-1) | PM_Text[j] | VN_j-1
6.            D0_j ← D0_j | TR_j
7.            HP_j ← VN_j-1 | ~(D0_j | VP_j-1)
8.            HN_j ← D0_j & VP_j-1
9.            If HP_j & 10^(m-1) ≠ 0 Then DT[m,j] ← DT[m,j] + 1
10.           If HN_j & 10^(m-1) ≠ 0 Then DT[m,j] ← DT[m,j] − 1
11.           If DT[m,j] ≤ k Then report a match ending at Text[j]
12.           VP_j ← (HN_j << 1) | ~(D0_j | (HP_j << 1))
13.           VN_j ← D0_j & (HP_j << 1)
```

Fig. 6. $D0_j$-based algorithm of Myers with transposition (for the case $m \leq w$).

5. Handling long patterns

When the pattern does not fit into a single computer word, the algorithm of Myers must use multiple words in representing the deltas between the (j-1)th and jth columns of the dynamic programming matrix. This can be done by simulating a longer vector. We discuss here one such method, which uses $\lceil m/w \rceil$ vector blocks so that the rth block of vectors represents the rows $(r\text{-}1)w+1\ldots rw$. In the basic case this results in an $O(\lceil mn \rceil / w)$ algorithm. In the following discussion we denote by $VP_{r,j}$ the rth vertical positive delta vector of the jth column. Then $VP_{r,j}[i] = 1$ iff $D[(r\text{-}1)w+i,\ j] - D[(r\text{-}1)w+i\text{-}1,\ j] = 1$. We use similar notation also with all the other vectors in the algorithm, so that for example $PM_{r,j}[i] = 1$ iff $Pat[(r\text{-}1)w+i] = Text[j]$.

5.1 Basic use of the vector blocks

The only real differences between using multiple blocks of vectors instead of a single one are ensuring that the blocks interact correctly with each other, and naturally that now r blocks of vectors need to be computed for each column. The vector blocks are computed in order of growing r (i.e. first the vector block consisting of $D0_{1,j}$, $HP_{1,j}$, $HN_{1,j}$, $VP_{1,j}$ and $HP_{1,j}$, then the second vector block consisting of $D0_{2,j}$, $HP_{2,j}$, $HN_{2,j}$, $VP_{2,j}$ and $HP_{2,j}$, and so on, see Figure 7a). Looking at the four steps listed in the beginning of Section 4 we can identify two locations in the algorithm, where some information from the computation of the (r-1)th vector block is needed when computing the rth vector block and $r > 1$.

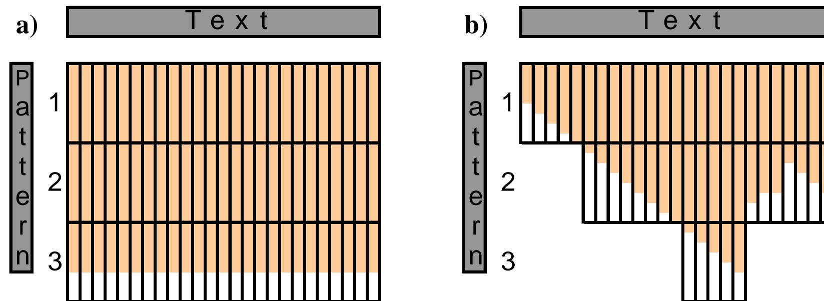

Fig. 7. A situation in which $\lceil m/w \rceil = 3$, and therefore three vector blocks are needed to represent a column of the dynamic programming matrix. The figure a) on the left depicts filling the matrix completely, and the shaded area corresponds to the cells filled by the regular dynamic programming algorithm. The figure b) on the right depicts using the cut-off heuristic. The shaded area corresponds to the cells filled by the dynamic programming algorithm with the cut-off heuristic, and only the minimum set of vector blocks needed to cover these is shown.

The first place is the computation of the diagonal zero delta vector. If a continuous run of ones ends in the lowest bit of the vector $VP_{r\text{-}1,j}$, the addition, which propagates diagonal zero-deltas down along these one bits, would propagate diagonal zero-deltas also to the q+1 first (uppermost) rows in the vector $D0_{r,j}$, where q is the number of consecutive one bits in the beginning of the vector $VP_{r,j}$. This happens exactly when

$HN_{r\text{-}1,j}[w]$ = 1. The correct propagation of the zero-deltas can be ensured by setting, before the value $PM_{r,j}$ is used, $PM_{r,j}[1] = 1$ if $HN_{r\text{-}1,j}[w] = 1$. This procedure leads both to setting the first bit of $D0_{r+1,j}$, and continuing the zero-delta propagation further down along the possible continuous one bits in the beginning of the vector $VP_{r+1,j}$.

The second place is the computation of the vertical delta vectors, which involves shifting the horizontal vectors $HP_{r,\mathrm{j}}$ and $HN_{r,j}$ to the left. For this to work correctly, the horizontal differences $D[(r\text{-}1)w,j] - D[(r\text{-}1)w,j]$, represented by $HP_{r\text{-}1,j}[w]$ and $HN_{r\text{-}1,j}[w]$, and $D[(r\text{-}1)w+1,j] - D[(r\text{-}1)w+1,j\text{-}1]$, represented by $HP_{r,j}[1]$ and $HN_{r,j}[1]$, should remain next to each other in the column. Thus the values $HP_{r,j}[1] = HP_{r\text{-}1,j}[w]$ and $HN_{r,j}[1] = HN_{r\text{-}1,j}[w]$ should be set after the shifts.

5.2 Long patterns and transposition

Computing the vector blocks involves two additional difficulties when also transpositions are allowed. They both arise from the left shift that is done in computing the rth block transposition vector $TR_{r,j}$ = $((\sim D0_{r,j\text{-}1})$ **&** $PM_{r,j}) << 1)$ **&** $PM_{r,j\text{-}1}$. As with $HP_{r,j}$ and $HN_{r,j}$ in Section 5.1, we must ensure that the shift sets the bit $D0_{r,j\text{-}1}[1]$ to have the value $D0_{r\text{-}1,j\text{-}1}[w]$, and the bit $PM_{r,j}[1]$ to have the value $PM_{r\text{-}1,j}[w]$. Figure 8 shows the algorithm for computing the rth block in column j when transposition is allowed.

```
ComputeBlock(r,j)
1.    X ← PM_{r,Text[j]}
2.    TR_{r,j} ← ((~D0_{r,j-1}) & X) << 1) & PM_{r,Text[j-1]}
3.    TR_{r,j} ← TR_{r,j} | ((~D0_{r-1,j-1}) & PM_{r-1,Text[j]}) >> (w-1)) & PM_{r,Text[j-1]}
4.    If HN_{r-1,j} & 10^{w-1} ≠ 0 Then X ← X | 1
5.    D0_{r,j} ← (((X & VP_{r,j-1}) + VP_{r,j-1}) ^ VP_{r,j-1}) | X | VN_{r,j-1}
6.    D0_{r,j} ← D0_{r,j} | TR_{r,j}
7.    HP_{r,j} ← VN_{r,j-1} | ~(D0_{r,j} | VP_{r,j-1})
8.    HN_{r,j} ← D0_{r,j} & VP_{r,j-1}
9.    If HP_{r,j} & 10^{m-1} ≠ 0 Then DT[rw,j] ← DT[rw,j] + 1
10.   If HN_{r,j} & 10^{m-1} ≠ 0 Then DT[rw,j] ← DT[rw,j] - 1
11.   X  ← HP_{r,j} << 1
12.   If HP_{r-1,j} & 10^{w-1} ≠ 0 Then X ← X | 1
13.   VP_{r,j} ← (HN_{r,j} << 1) | ~(D0_{r,j} | X))
14.   If HN_{r-1,j} & 10^{w-1} ≠ 0 Then VP_{r,j} ← VP_{r,j} | 1
15.   VN_{r,j} ← D0_{r,j} & X
```

Fig. 8. The algorithm for computing the rth vector block in column j when transposition is allowed. This version is shown as it is for clarity, and there is room for optimization. For example we could merge lines 2 and 3.

5.3 Using the cut-off heuristic in computing the blocks

In this discussion we concentrate on the case of using the edit distance *ed*, but the method can be used in exactly the same way also when the distance function *edt* is used.

In what follows we propose a method for using cut-off heuristic in computing the vector blocks. This differs only slightly from the one proposed by Myers and results in a $O(kn/w)$ expected time algorithm. The goal is to compute the vector blocks only as far down in the column that is needed in order to cover the area of the dynamic programming matrix that the Ukkonen cut-off heuristic would fill (Figure 7b). In the jth column the rth block is called active if $D[i,j\text{-}1] \leq k$ for some $i \geq (r\text{-}1)w + 1$. According to the cut-off heuristic, only the active blocks need to be computed. To help in doing this, in a way explained later, the value $D[(r\text{-}1)w,j]$ is explicitly maintained for all blocks.

To make the algorithm uniform (and possibly more efficient), Myers did not separately calculate the value of the cell $D[m,j]$ even if the pattern is not equally divided by w. He rather added $w\lceil m/w\rceil$ - m so-called wild card characters, which match with every character, to the end of the pattern. This makes the value of the cell $D[m,j]$ propagate diagonally with the matches into the cell $D[w\lceil m/w\rceil,j+w\lceil m/w\rceil\text{-}m]$. Thus an approximate occurrence of the pattern ends at the text character $Text[j\text{-}w\lceil m/w\rceil+m]$ whenever $D[w\lceil m/w\rceil,j] \leq k$. If all $\lceil m/w\rceil$ blocks are active after the last text character of the text has been processed, the possible endpoints of occurrences in the area $Text[n\text{-}w\lceil m/w\rceil\text{-}m+1\ldots n]$ can be checked by starting from the cell $D[w\lceil m/w\rceil,n]$ and then computing the values $D[w\lceil m/w\rceil\text{-}1,n]$, $D[w\lceil m/w\rceil\text{-}2,n],\ldots,$ $D[m,n]$ using the vectors $VN_{\lceil m/w\rceil,n}$ and $VN_{\lceil m/w\rceil,n}$. An occurrence ends at the character $Text[n\text{-}i+m]$ whenever $D[i,n] \leq k$ and $i \geq m$[1]. Also we use this scheme.

From the initial conditions of the dynamic programming matrix it is known that $D[i,0] > k$ for $i > k$. Thus in the first column the first $\lceil k/w\rceil$ vector blocks are active and need to be computed. Let b_j denote the number of active the vector blocks in the jth column. Note that then the b_jth block is the lowest active block and $b_1 = \lceil k/w\rceil$. The value for b_j is determined as follows.

In the jth column the $(b_{j\text{-}1}+1)$th block becomes active only iff $D[b_{j\text{-}1}w,j\text{-}1] = k = D[b_{j\text{-}1}w+1,j]$. This follows from the adjacency and diagonal properties and the fact that, by the definition of $b_{j\text{-}1}$, the cells $D[b_{j\text{-}1}w+1,j\text{-}1]$, $D[b_{j\text{-}1}w+2,j\text{-}1],\ldots,$ $D[b_{j\text{-}1}w,j\text{-}1]$ have a value $> k$. But instead of checking for the whole condition, as described by Myers, we choose to only test whether $D[b_{j\text{-}1}w,j\text{-}1] = k$. This is because this simpler rule works with both distances *ed* and *edt*, and in a brief test we found it to perform virtually as well as the full check. Also the asymptotic runtime is still the same $O(kn/w)$ expected time.

In processing the jth column, initially the first $b_{j\text{-}1}$ blocks are computed. Then if $D[b_{j\text{-}1}w,j] = k$, we set $b_j = b_{j\text{-}1}+ 1$. After this, the just-activated b_jth vector block is initialized by setting $VP_{b,j} = 1^w$ and $VN_{b,j} = 0$ with $b = b_j$. This corresponds to having $D[(b_{j-1})w+1,j] = k$, $D[(b_{j-1})w+2,j] = k+1,\ldots,$ $D[b_jw,j] = k+w\text{-}1$ and does not affect the correct behavior of the algorithm. These cells are known to have a value $> k$ and their accurate values are therefore obsolete in terms of finding approximate matches.

It is difficult to determine in an exact and efficient manner whether the b_jth block becomes inactive when moving into the $(j+1)$th column. This lead Myers to use the seemingly crude rule, based on the adjacency property, that the blocks $b_j\text{-}q$, $b_j\text{-}q+1,\ldots,$ b_j become inactive, and thus $b_{j+1} = b_j\text{-}q\text{-}1$, if $D[(b_j\text{-}p)w,j] > k + w$ for $p = 0\ldots q$. But the method works well in practice. We for example tested a stricter rule, that the blocks

---

[1] This is based on an actual implementation by Myers, and is not mentioned in his article [7].

b_j-q, b_j-q+1,…, b_j become inactive if $D[(b_j\text{-}p)w,j] + D[(b_j\text{-}p\text{-}1)w,j] > 2k + w$ for $p = 0…q$, but the overhead of having to add the two cell values overwhelmed the benefit of filling slightly less vector blocks.

By following the method used by Hyyrö and Navarro in the forward verification phase of their Myers-based version of the ABNDM heuristic [4], it is in principle possible to keep track of the exact location of the lowest cell with a value $\leq k$ in the current column without worsening the asymptotical complexity of the algorithm. But the approach seemed to be quite slow in practice when we briefly tested this.

Figure 9 shows using our slightly modified version of the cut-off heuristic with the vector blocks and the matrix DT. As the original version of Myers, the algorithm runs in $O(kn/w)$ expected time.

```
CutOffVectorBlocks(Pat[1…m], Text[1…n], k)
1.    Preprocessing
2.       For r ∈ 1…⌈m/w⌉ Do
3.          For λ ∈ all characters Do
4.             PM_{r,λ} ← 0
5.          For i ∈ 1…min{w,m-(r-1)w} Do
6.             PM_{r,Pat[(r-1)w+i]} ← PM_{r,Pat[(r-1)w+i]} | 0^{w-i}10^{i-1}
7.       For λ ∈ all characters Do
8.          PM_{⌈m/w⌉,λ} ← PM_{⌈m/w⌉,λ} | 1^{w⌈m/w⌉-m}0^{m-w(⌈m/w⌉-1)}
9.       b ← ⌈k/w⌉
10.      For r ∈ 1…b Do
11.         VP_{r,0} ← 1^m, VN_{r,0} ← 0, DT[rw,0] ← rw
12.   Searching
13.      For j ∈ 1…n do
14.         For r ∈ 1…b Do
15.            ComputeBlock(r,j)
16.         If DT[bw,j-1] = k Then
17.            VP_{b+1,j} ← 1^w, VN_{b+1,j} ← 0, DT[(b+1)w,j] ← k+w, b ← b+1
18.         Else
19.            While DT[bw,j] > k+w Do
20.               b ← b-1
21.         If b = ⌈m/w⌉ and DT[bw,j] ≤ k Then
22.            report a match ending at Text[j-w⌈m/w⌉+m]
23.      If b = ⌈m/w⌉ Then
24.         For i ∈ 1…bw-m Do
25.            If VP_{b,n} & 10^{w-i} ≠ 0 Then DT[bw-i,n] ← DT[bw-i-1,n]-1
26.            If VN_{b,n} & 10^{w-i} ≠ 0 Then DT[bw-i,n] ← DT[bw-i-1,n]+1
27.            If DT[bw-i,n] ≤ k Then a match ends at Text[n-bw+i+m]
```

Fig. 9. An algorithm for using the cut-off heuristic with vector blocks.

6. Test results

In this section we present some test results concerning the effect that adding transposition has on the algorithm of Myers. The computer used in the tests was a 32-bit dual-processor Pentium3 550 Mhz with 256 MB RAM and Linux OS, and the code was compiled with GCC and full optimization. Each test consisted of searching 100 random patterns from a 10 MB random text. We used the original code from

Myers as the basis for all implementations so that they would be of comparable quality. When the pattern did not fit into the computer word, the cut-off heuristic was used. Figure 10 shows the results for m = 10, 20, … 150 and $k = m/5$. As can be seen, adding transposition brings very little extra cost as long as the pattern fits into a single computer word. With longer patterns the difference is roughly 20%. This bigger gap with the longer patterns arises from the extra work needed with *edt* in handling the block boundaries.

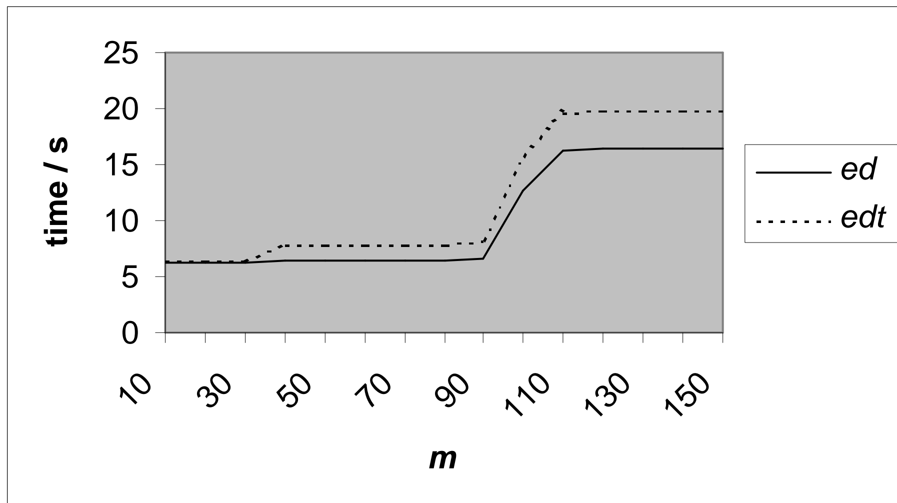

Fig. 10. The test results for searching 100 patterns in 10 MB of text with $k = m/5$.

7. Conclusions

The $O(\lceil mn \rceil / w)$ bit-parallel algorithm of Myers [7] is one of the leading current approximate string matching algorithm when the underlying edit distance permits insertions, deletions and substitutions. In this paper we presented, in what we think is a more intuitive way than the original, a slightly different but practically equivalent version of the algorithm of Myers. Then we extended the algorithm to also permit transposition of two adjacent characters. The resulting algorithm was found to be almost as fast as the original when the pattern fits into a single computer word, and roughly 20% slower with longer patterns.

Acknowledgments

We would like to thank Gonzalo Navarro for some useful comments on a very early draft of the ideas in the paper.

References

1. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of CPM'92*, LNCS 644: 172-181, 1992.
2. F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3): 171-176, 1964.
3. M .W. Du and S. C. Chang. A model and a fast algorithm for multiple errors spelling correction. *Acta Informatica*, 29: 281-302, 1992.
4. H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. To apper in *Proceedings of CPM'2002*.
5. K. Kukich. Automatically correcting words in text. *ACM Computing Surveys*, 24(4): 377-439, 1992.
6. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Sov. Phys. Dokl.* 10: 707-710, 1966.
7. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3): 395-415, 1999.
8. G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys* 33(1): 31-88, 2001.
9. G. Navarro. NR-grep: a Fast and Flexible Pattern Matching Tool. *Software Practice and Experience* 31: 1265-1312, 2001.
10. P. H. Sellers. On the theory of computation of evolutionary distances. *SIAM Journal of Applied Mathematics*, 26: 787-793, 1974.
11. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64: 100-118, 1985.
12. E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6: 132-137, 1985.
13. S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10): 83-91, 1992.