Bit-Parallel Approximate String Matching Algorithms with Transposition

Heikki Hyyrö [*]

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

Abstract. Using bit-parallelism has resulted in fast and practical algorithms for approximate string matching under the Levenshtein edit distance, which permits a single edit operation to insert, delete or substitute a character. Depending on the parameters of the search, currently the fastest non-filtering algorithms in practice are the $O(kn\lceil m/w\rceil)$ algorithm of Wu & Manber, the $O(\lceil km/w\rceil n)$ algorithm of Baeza-Yates & Navarro, and the $O(\lceil m/w\rceil n)$ algorithm of Myers, where m is the pattern length, n is the text length, k is the error threshold and w is the computer word size. In this paper we discuss a uniform way of modifying each of these algorithms to permit also a fourth type of edit operation: transposing two adjacent characters in the pattern. This type of edit distance is also known as the Damerau edit distance. In the end we also present an experimental comparison of the resulting algorithms.

1 Introduction

Approximate string matching is a classic problem in computer science, with applications for example in spelling correction, bioinformatics and signal processing. It has been actively studied since the sixties [8]. Approximate string matching refers in general to the task of searching for substrings of a text that are within a predefined edit distance threshold from a given pattern. Let $T_{1..n}$ be a text of length n and $P_{1..m}$ a pattern of length m. In addition let $ed(A, B)$ denote the edit distance between the strings A and B, and k be the maximum allowed distance. Using this notation, the task of approximate string matching is to find from the text all indices j for which $ed(P, T_{h..j}) \leq k$ for some $h \leq j$.

Perhaps the most common form of edit distance is the Levenshtein edit distance [6], which is defined as the minimum number of single-character insertions, deletions and substitutions (Fig. 1a) needed in order to make A and B equal. Another common form of edit distance is the Damerau edit distance [2], which is in principle an extension of the Levenshtein distance by permitting also the operation of transposing two adjacent characters (Fig. 1b). The Damerau edit

[*] Supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering.

distance is important for example in spelling error applications [5]. In this paper we use the notation $ed_L(A, B)$ to denote the Levenshtein edit distance and $ed_D(A, B)$ to denote the Damerau edit distance between A and B.

During the last decade, algorithms based on bit-parallelism have emerged as the fastest approximate string matching algorithms in practice for the Levenshtein edit distance [6]. The first of these was the $O(kn\lceil m/w \rceil)$ algorithm of Wu & Manber [15], where w is the computer word size. Later Wright [14] presented an $O(mn \log(\sigma)/w)$ algorithm, where σ is the alphabet size. Then Baeza-Yates & Navarro followed with their $O(\lceil km/w \rceil n)$ algorithm. Finally Myers [7] achieved an $O(\lceil m/w \rceil n)$ algorithm, which is an optimal speedup from the basic $O(mn)$ dynamic programming algorithm (e.g. [11]). With the exception of the algorithm of Wright, the bit-parallel algorithms dominate the other verification capable[1] algorithms with moderate pattern lengths [8].

a)

insertion:	cat → ca**s**t
deletion:	**c**at → at
substitution:	ca**t** → ca**r**

b) transposition: **ca**t → **ac**t

Fig. 1. Figure a) shows the three edit operations permitted by the Levenshtein edit distance. Figure b) shows the additional edit operation permitted by the Damerau edit distance: transposing two adjacent characters. The transposed characters are required to be/remain adjacent in the original and the modified pattern.

In this paper we show how each of the above-mentioned three best bit-parallel algorithms can be modified to use the Damerau edit distance. Navarro [9] has previously extended the algorithm of Wu & Manber [15] for the Damerau distance. But that method adds $O(k\lceil m/w \rceil)$ work to the original algorithm, whereas the additional cost of our method is only $O(\lceil m/w \rceil)$. Our method is also more general in that its principle works with also the other two algorithms [1, 7] with very little changes.

We begin by discussing the basic dynamic programming solutions for the Levenshtein and Damerau distances. In this part we also reformulate the dynamic programming solution for the Damerau edit distance into a form that is easier to handle for the bit-parallel algorithms. Then we proceed to modify the bit-parallel algorithms of Wu & Manber [15], Baeza-Yates & Navarro [1] and Myers [7] to facilitate the Damerau edit distance. Finally we present an experimental comparison of these modified algorithms.

[1] Are based on actually computing the edit distance.

2 Dynamic Programming

In the following we assume that $A_{1..0} = \epsilon$, where ϵ denotes the empty string. In addition let $|A|$ denote the length of the string A. We consider first the Levenshtein edit distance. In this case the dynamic programming algorithm fills a $(|A|+1) \times (|B|+1)$ dynamic programming table D, where in the end each cell $D[i,j]$ will hold the value $ed_L(A_{1..i}, B_{1..j})$. The algorithm begins from the trivially known values $D[i,0] = ed_L(A_{1..i}, \epsilon) = i$ and $D[0,j] = ed_L(\epsilon, B_{1..j}) = j$, and arrives at the value $D[A,B] = ed_L(A_{1..|A|}, B_{1..|B|}) = ed_L(A,B)$ by recursively computing the value $D[i,j]$ from the previously computed values $D[i-1,j-1]$, $D[i,j-1]$ and $D[i-1,j]$. This can be done using the following well-known Recurrence 1.

Recurrence 1

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

The Damerau edit distance can be computed in basically the same way, but Recurrence 1 needs a slight change. The following Recurrence 2 for the Damerau edit distance is derived from the work of Du & Chang [3]. The superscript R denotes the reverse of a string (that is, if $A =$ "abc", then $A^R =$ "cba").

Recurrence 2

$$D[i,-1] = D[-1,j] = \max(|A|,|B|).$$
$$D[i,0] = i, D[0,j] = j.$$
$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if } A_i = B_j. \\ 1 + \min(D[i-2,j-2], D[i-1,j], D[i,j-1]), \text{ if } A_{i-1..i} = \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (B_{j-1..j})^R. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ otherwise.} \end{cases}$$

Instead of computing the edit distance between strings A and B, the dynamic programming algorithm can be changed to find approximate occurrences of A somewhere inside B by changing the boundary condition $D[0,j] = j$ into $D[0,j] = 0$. In this case $D[i,j] = \min(ed_L(P_{0..i}, T_{h..j}), h \leq j)$ with the Levenshtein edit distance and $D[i,j] = \min(ed_D(P_{0..i}, T_{h..j}), h \leq j)$ with the Damerau edit distance. Thus, if we set $A = P$ and $B = T$, the situation corresponds to the earlier definition of approximate string matching. From now on we assume that the dynamic programming table D is filled in this manner.

Ukkonen ([12,13]) 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: $D[i,j] - D[i-1,j-1] = 0$ 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, it is fairly straightforward to verify that they apply also to the Damerau edit distance.

The values of the dynamic programming matrix D are usually computed by filling it in a column-wise manner for increasing j, thus effectively scanning the string B (or the text T) one character at a time from left to right. At each character the corresponding column is completely filled in the order of increasing i. This allows us to save space by storing only one or two columns at a time, since the values in column j depend only on one (Levenshtein) or two (Damerau) previous columns.

Now we reformulate Recurrence 2 into a form that is easier to use with the three bit-parallel algorithms. Our trick is to investigate how a transposition relates to a substitution. Consider comparing the strings A = "abc" and B = "acb". Then $D[2,2] = ed_D(A_{1..2}, B_{1..2}) = ed_D(\text{"ab"},\text{"ac"}) = 1$, where the one operation corresponds to substituting the first character of the transposable suffixes "bc" and"cb". When filling in the value $D[3,3] = ed_D(\text{"abc"},\text{"acb"})$, the effect of having done a single transposition can be achieved by allowing a free substitution between the latter characters of the transposable suffixes. This is the same as declaring a match between them. In this way the cost for doing the transposition has already been paid for by the substitution of the preceding step. It turns out that this idea can be developed to work correctly in all cases. We find that the following Recurrence 3 for the Damerau edit distance is in effect equivalent with Recurrence 2. It uses an auxiliary $|A| \times (|B| + 1)$ boolean table MT as it is convenient for bit-parallel algorithms. The value $MT[i,j]$ records whether there is the possibility to match or to make a free substitution when computing the value $D[i,j]$.

Recurrence 3

$$D[i,0] = i, D[0,j] = j, MT[i,0] = \textbf{false}.$$

$$MT[i,j] = \begin{cases} \textbf{true}, \text{ if } A_i = B_j \text{ or } (MT[i-1,j-1] = \textbf{false} \text{ and} \\ \qquad\qquad\qquad\qquad A_{i-1..i} = (B_{j-1..j})^R). \\ \textbf{false}, \text{ otherwise.} \end{cases}$$

$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if} MT[i,j] = \textbf{true}. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ otherwise.} \end{cases}$$

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

Clearly both formulas give the same value for $D[i,j]$ when i = 0 or 1 or j = 0 or 1. Consider now a cell $D[i,j]$ for some $j > 1$ and $i > 1$ and assume that all previous cells with nonnegative indices have been filled identically by both recurrences[2]. Let x be the value given to $D[i,j]$ by Recurrence 2 and y be the

[2] We assume that a legal filling order has been used, which means that the cells $D[i-1,j-1]$, $D[i-1,j]$ and $D[i,j-1]$ are always filled before the cell $D[i,j]$.

value given to it by Recurrence 3. The only situation in which the two formulas could possibly behave differently is when $A_i \neq B_j$ and $A_{i-1..i} = (B_{j-1..j})^R$. In the following two cases we assume that these two conditions hold.

If $D[i-1, j-1] = D[i-2, j-2] + 1$, then $MT[i-1, j-1] =$ **false** and $MT[i, j] =$ **true**, and thus $y = D[i-1, j-1]$. Since the diagonal property requires that $x \geq D[i-1, j-1]$ and now $x \leq D[i-2, j-2] + 1$, we have $x = D[i-2, j-2] + 1 = D[i-1, j-1] = y$.

Now consider the case $D[i-2, j-2] = D[i-1, j-1]$. Because $A_{i-1} = B_j \neq A_i = B_{j-1}$, this equality cannot result from a match. If it resulted from a free substitution, then $MT[i-1, j-1] =$ **true** in Recurrence 3. As $A_i \neq B_j$, the preceding means that $MT[i, j] =$ **false**. Therefore $y = 1 + \min(D[i-1, j-1], D[i-1, j], D[i, j-1])$ and $x = 1 + \min(D[i-2, j-2], D[i-1, j], D[i, j-1])$. Because $D[i-2, j-2] = D[i-1, j-1]$, the former means that $x = 1 + \min(D[i-1, j-1], D[i-1, j], D[i, j-1]) = y$. The last possibility is that the equality $D[i-2, j-2] = D[i-1, j-1]$ resulted from using the option $D[i-1, j-1] = 1 + \min(D[i-2, j-1], D[i-1, j-2])$. As $A_{i-1} = B_j$ and $A_i = B_{j-1}$, both recurrences must have set $D[i-1, j] = D[i-2, j-1]$ and $D[i, j-1] = D[i-1, j-2]$ and therefore $D[i-1, j-1] = 1 + \min(D[i-2, j-1], D[i-1, j-2]) = 1 + \min(D[i-1, j], D[i, j-1])$. Now both options in Recurrence 3 set the same value $y = D[i-1, j-1]$, and $x = 1 + \min(D[i-2, j-2], D[i-1, j], D[i, j-1]) = 1 + \min(D[i-1, j], D[i, j-1]) = D[i-1, j-1] = y$.

In each case Recurrence 2 and Recurrence 3 assigned the same value for the cell $D[i, j]$. Therefore we can state by induction that the recurrences are in effect equivalent. $\square$

The intuition behind the table MT in Recurrence 3 is that a free substitution is allowed at $D[i, j]$ if a transposition is possible at that location. But we cannot allow more than one free substitution in a row along a diagonal, as each corresponding transposition has to be paid for by a regular substitution. Therefore when a transposition has been possible at $D[i, j]$, another will not be allowed at $D[i+1, j+1]$. And as shown above, this restriction on when to permit a transposition does not affect the correctness of the scheme.

3 Modifying the Bit-parallel Algorithms

Bit-parallel algorithms are based on taking advantage of the fact that a single computer instruction manipulates bit-vectors with w bits (typically $w = 32$ or 64 in the current computers). If many data-items of an algorithm can be encoded into w bits, it may be possible to process many data-items within a single instruction (thus the name bit-parallelism) and achieve gain in time and/or space.

We use the following notation in describing bit-operations: '&' denotes bitwise "and", '|' denotes bitwise "or", '$\wedge$' denotes bitwise "xor", '$\sim$' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. The ith bit of the bit vector V is referred to as $V[i]$ and bit-positions are assumed to grow from right to left. In addition we use superscript to denote bit-repetition. As an 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 $V = 10^2 1^3 0$.

3.1 The Bit-parallel NFA of Wu & Manber

The bit-parallel approximate string matching algorithm of Wu & Manber [15] is based on representing a non-deterministic finite automaton (NFA) by using bit-vectors. The automaton has $(k+1)$ rows, numbered from 0 to k, and each row contains m states. Let us denote the automaton as R, its row d as R_d and the state i on its row d as $R_{d,i}$. The state $R_{d,i}$ is active after reading the text up to the jth character if and only if $ed(P_{1..i}, T_{h..j}) \leq d$ for some $h \leq j$. An occurrence of the pattern with at most k errors is found when the state $R_{k,m}$ is active. Assume for now that $w \leq m$. Wu & Manber represent each row R_d as a length-m bit-vector, where the ith bit tells whether the state $R_{d,i}$ is active or not. In addition they build a length-m match vector for each character in the alphabet. We denote the match vector for the character λ as PM_λ. The ith bit of PM_λ is set if and only if $P_i = \lambda$. Initially each vector R_d has the value $0^{m-d}1^d$ (this corresponds to the boundary conditions in Recurrence 1). The formula to compute the updated values R'_d for the row-vectors R_d at text position j is the following:

```
$R'_0 \leftarrow ((R_0 << 1) \mid 0^{m-1}1)\ \&\ PM_{T_j}$
**For** $d = 1$ **to** $k$ **Do**
    $R'_d \leftarrow ((R_d << 1)\ \&\ PM_{T_j}) \mid R_{d-1} \mid (R_{d-1} << 1) \mid (R'_{d-1} << 1)$
        $\mid 0^{m-1}1$
```

The right side of the last row computes the disjunction of the different possibilities given by Recurrence 1 for a prefix of the pattern to match with d errors. The row R_0 is different as it needs to consider only matching positions between P and the character T_j, and it also has to have its first bit set after the left-shift in order to let the first character match at the current position. When $m \leq w$, the run time of this algorithm is $O(kn)$ as there are $O(k)$ operations per text character. The general run time is $O(kn\lceil m/w \rceil)$ as a vector of length m may be simulated in $O(\lceil m/w \rceil)$ time using $O(\lceil m/w \rceil)$ bit-vectors of length w. In this paper we do not discuss the details of such a multi-word implementation for any of the bit-parallel algorithms.

Navarro [9] has modified this algorithm to use the Damerau distance by essentially following Recurrence 2. He did this by appending the automaton to have a temporary state vector T_d for each R_d to keep track of the positions where transposition may occur. Initially each T_d has the value 0^m. Navarro's formula is:

```
$R'_0 \leftarrow ((R_0 << 1) \mid 0^{m-1}1)\ \&\ PM_{T_j}$
**For** $d = 1$ **to** $k$ **Do**
    $R'_d \leftarrow ((R_d << 1)\ \&\ PM_{T_j}) \mid R_{d-1} \mid (R_{d-1} << 1) \mid (R'_{d-1} << 1)$
        $\mid (T_d\ \&\ (PM_{T_j} << 1)) \mid 0^{m-1}1$
    $T'_d \leftarrow (R_{d-1} << 2)\ \&\ PM_{T_j}$
```

The formula adds $6k$ operations into the basic version for the Levenshtein edit distance.

Recurrence 3 suggests a simpler way to facilitate transposition. The only difference between it and Recurrence 1 is in the condition on when $D[i, j] = D[i-1, j-1]$: Instead of the condition $P_i = T_j$, Recurrence 3 sets the equal value if $MT[i, j] = \textbf{true}$ (here we again replaced A with P and B with T in the recurrence). We use a length-m bit-vector TC in storing the last column of the auxiliary table MT. The ith bit of TC is set iff row i of the last column of MT has the value **true**. When we arrive at text position j, TC is updated to hold the values of column j. Initially $TC = 0^m$. Based on Recurrence 3, the vector TC may be updated with the formula $TC' = PM_{T_j} \mid (((\sim TC) << 1) \,\&\, (PM_{T_j} << 1) \,\&\, PM_{T_{j-1}})$. Here the right "and" sets the bits in the pattern positions where $P_{i-1..i} = (T_{j-1..j})^R$, the left "and" sets off the ith bit if row $(i-1)$ of MT had the value **true** in the previous column, and the "or" sets the bits in the positions where $P_i = T_j$. By combining the two left-shifts we get the following complete formula for updating the R_d vectors:

```
$TC' \leftarrow PM_{T_j} \mid ((((\sim TC) \,\&\, PM_{T_j}) << 1) \,\&\, PM_{T_{j-1}})$
$R'_0 \leftarrow ((R_0 << 1) \mid 0^{m-1}1) \,\&\, PM_{T_j}$
For $d = 1$ to $k$ Do
    $R'_d \leftarrow ((R_d << 1) \,\&\, TC') \mid R_{d-1} \mid (R_{d-1} << 1) \mid (R'_{d-1} << 1)$
        $\mid 0^{m-1}1$
```

Our formula adds a total of 6 operations into the basic version for the Levenshtein edit distance. Therefore it makes the same number of operations as Navarro's version when $k = 1$, and wins when $k > 1$.

3.2 The Bit-parallel NFA of Baeza-Yates & Navarro

Also the bit-parallel algorithm of Baeza-Yates & Navarro [1] is based on simulating the NFA R. The first d states on row R_d are trivial in that they are always active. The last $k - d$ states will be active only if the state $R_{k,m}$ is active, and as we are only interested in knowing whether there is a match with at most k errors, having the state $R_{k,m}$ is enough. These facts enable Baeza-Yates & Navarro to include only the $m - k$ states $R_{d,d+1}..R_{d,m-k+d}$ on row R_d. A further difference is in the way the states are encoded into bit-vectors. They divide R into $m-k$ diagonals $D_1, .., D_{m-k}$, where D_i is a bit-sequence that describes the states $R_{d,d+i}$ for $d = 0..k$. If a state $R_{d,i}$ is active, then all states on the same diagonal that come after $R_{d,i}$ are active, that is, the states $R_{d+h,i+h}$ for $h \geq 1$. To describe the status of the ith diagonal it suffices to record the position of the first active state in it. If the first active state on the ith diagonal is f_i, then Baeza-Yates & Navarro represent the diagonal as the bit-sequence $D_i = 0^{k+1-f_i}1^{f_i}$. The value $f_i = k+1$ means that $f_i \geq k+1$, that is, that no states on the ith diagonal of R is active. A match with at most k errors is found whenever $f_{m-k} < k+1$. The d_i bit-sequences are stored consecutively with a single separator zero-bit between two consecutive states. Let RD denote the complete diagonal representation.

Then RD is the length-$(k+2)(m-k)$ bit-sequence $0\ D_1\ 0\ D_2\ 0...0\ D_{m-k}$. We assume for now that $(k+2)(m-k) \leq w$ so that RD fits into a single bit-vector.

Baeza-Yates & Navarro encode also the pattern match vectors differently. Let PMD_λ be their pattern match vector for the character λ. The role of the bits is reversed: a 0-bit denotes a match and a 1-bit a mismatch. To align the matches with the diagonals in RD, PMD_λ has the form
$0 \quad \sim (PM_\lambda[1..k+1])\ \ 0 \quad \sim (PM_\lambda[2..k+2])\ \ 0...0 \quad \sim (PM_\lambda[m-k..m])$.

Initially no diagonal has active states and so $RD = (0\ 1^{k+1})^{m-k}$. The formula for updating RD at text position j is:

$$\begin{aligned}
x &\leftarrow (RD >> (k+2)) \mid PMD_{T_j} \\
RD' &\leftarrow ((RD << 1) \ \mid \ (0^{k+1}1)^{m-k} \\
&\quad \& \ (RD << (k+3)) \ \mid \ (0^{k+1}1)^{m-k-1}01^{k+1} \\
&\quad \& \ (((x + (0^{k+1}1)^{m-k}) \wedge x) >> 1) \\
&\quad \& \ (0\ 1^{k+1})^{m-k}
\end{aligned}$$

If $(k+2)(m-k) \leq w$, the run time of this algorithm is $O(n)$ as there is only a constant number of operations per text character. The general run time is $O(\lceil km/w \rceil n)$ as a vector of length $(k+2)(m-k)$ may be simulated in $O(\lceil km/w \rceil)$ time using $O(\lceil km/w \rceil)$ bit-vectors of length w.

Because of the different way of representing R, our way of modifying the algorithm of Wu & Manber to use the Damerau edit distance does not work here without some changes. Now we use a bit-vector TCD instead of the vector TC of the previous section. TCD has the same function as TC, but its form corresponds to the algorithm of Baeza-Yates & Navarro. First of all the meaning of the bit-values is reversed: now a 0-bit corresponds to the value **true** and a 1-bit to the value **false** in the table TR of Recurrence 3. The second change is in the way we compute the positions where $P_{i-1..i} = (T_{j-1..j})^R$. Because of the interleaving 0-bits in the pattern match vector PMD_λ, the formula $(PMD_{T_j} << 1) \ \mid \ PMD_{T_{j-1}}$ does not correctly set only those bits to zero that correspond to a transposable position (note that also the roles of '&' and '|' are reversed). But by inspecting the form of BPD_λ we notice that the desired effect is achieved by using the formula $(PMD_{T_j} >> (k+2)) \ \mid \ PMD_{T_{j-1}}$. Shifting $(k+2)$ bits to the right causes the $(i-1)$th diagonal to align with the ith diagonal, and this previous diagonal handles the matches one step to the left in the pattern. The only delicacy in doing this is the fact that now the first diagonal will have no match-data. Because we need to have made a substitution before making a free substitution that corresponds to a transposition, a transposition will be possible only in diagonals $2..m-k$. Thus the missing data can be replaced with mismatches. Note that we do not need to consider the states not present in the reduced automaton of Baeza-Yates & Navarro. By similar reasoning also the previous values of TCD will be shifted $(k+2)$ bits to the right instead of 1 bit to the left, and its missing data can be replaced by '**false**' values. Initially TCD has only '**false**' values and so $TCD = (0\ 1^{k+1})^{m-k}$. The modified formula for updating RD at text position j is:

$$\begin{aligned}
&TCD' \leftarrow PMD_{T_j} \;\&\; (((((\sim TCD) \;|\; PMD_{T_j}) >> (k+2)) \;|\; PMD_{T_{j-1}}) \\
&\qquad |\; 01^{k+1}0^{(m-k-1)(k+2)}) \\
&x \leftarrow (RD >> (k+2)) \;|\; TCD' \\
&RD' \leftarrow ((RD << 1) \;|\; (0^{k+1}1)^{m-k} \\
&\qquad \& \; (RD << (k+3)) \;|\; (0^{k+1}1)^{m-k-1}01^{k+1} \\
&\qquad \& \; (((x + (0^{k+1}1)^{m-k}) \wedge x) >> 1) \\
&\qquad \& \; (0\;1^{k+1})^{m-k}
\end{aligned}$$

Now the number of added operations is 7, as one "extra" operation arises from having to set the missing values (second row).

3.3 Myers' Bit-parallel Computation of D

The bit-parallel algorithm of Myers [7] is quite different from the previous two algorithms. We describe it here in a slightly simpler way than the original, even though the logic is in principle the same. The algorithm is based on representing the dynamic programming table D with vertical, horizontal and diagonal differences (see the adjacency and diagonal properties in Section 2). This is done by using the following length-m bit-vectors:

-The vertical positive delta vector VP:
  $VP[i] = 1$ at text position j iff $D[i,j] - D[i-1,j] = 1$.
-The vertical negative delta vector VN:
  $VN[i] = 1$ at text position j iff $D[i,j] - D[i-1,j] = -1$.
-The horizontal positive delta vector HP:
  $HP[i] = 1$ at text position j iff $D[i,j] - D[i,j-1] = 1$.
-The horizontal negative delta vector HN:
  $HN[i] = 1$ at text position j iff $D[i,j] - D[i,j-1] = -1$.
-The diagonal zero delta vector $D0$:
  $D0[i] = 1$ at text position j iff $D[i,j] = D[i-1,j-1]$.

In the original work of Myers the information of the vector $D0$ was represented by two separate vectors x_v and x_h.

Initially $VP = 1^m$ and $VN = 0^m$. At text position j the algorithm first computes the vector $D0$ by using the old values VP and VN and the pattern match vector PM_{T_j} (Section 3.1). Then the new HP and HN are computed by using $D0$ and the old VP and VN. Then finally the vectors VP and VN are updated by using the new $D0$, HN and HP. The complete formula for computing the updated vectors $D0', HP', HN', VN'$ and VP' at text position j is:

$$\begin{aligned}
&D0' \leftarrow (((PM_{T_j} \;\&\; VP) + VP) \wedge VP) \;|\; PM_{T_j} \;|\; VN \\
&HP' \leftarrow VN \;|\; \sim (D0' \;|\; VP) \\
&HN' \leftarrow VP \;\&\; D0' \\
&VP' \leftarrow (HN' << 1) \;|\; \sim (D0' \;|\; (HP' << 1)) \\
&VN' \leftarrow (HP' << 1) \;\&\; D0'
\end{aligned}$$

The current value of the dynamic programming cell $D[m, j]$ can be updated at each text position j by using the horizontal delta vectors (the initial value is $D[m, 0] = m$). A match of the pattern with at most k errors is found whenever $D[m, j] \leq k$.

If $m \leq w$, the run time of this algorithm is $O(n)$ as there is again only a constant number of operations per text character. The general run time is $O(\lceil m/w \rceil n)$ as a vector of length m may be simulated in $O(\lceil m/w \rceil)$ time using $O(\lceil m/w \rceil)$ bit-vectors of length w.

Because the algorithm of Myers uses the same pattern match vectors as the algorithm of Wu & Manber, it can be modified to use the Damerau distance by using exactly the same method as we used in Section 3.1. Thus the formula to update the vectors at text position j is simply:

$$\begin{aligned}
&TC' \leftarrow PM_{T_j} \ \mid \ ((((\sim TC) \ \& \ PM_{T_j}) << 1) \ \& \ PM_{T_{j-1}}) \\
&D0' \leftarrow (((TC' \ \& \ VP) + VP) \wedge VP) \ \mid \ TC' \ \mid \ VN \\
&HP' \leftarrow VN \ \mid \ \sim (D0' \ \mid \ VP) \\
&HN' \leftarrow VP \ \& \ D0' \\
&VP' \leftarrow (HN' << 1) \ \mid \ \sim (D0' \ \mid \ (HP' << 1)) \\
&VN' \leftarrow (HP' << 1) \ \& \ D0'
\end{aligned}$$

There is again 6 added operations.

4 Test Results

We implemented and tested a Damerau edit distance -version of each of the three discussed bit-parallel algorithms. The version of the algorithm of Wu & Manber was implemented from scratch by us, and the other two were modified using the original implementations from those authors. We compared also the versions for the Levenshtein edit distance to see how our modification affects the respective performance of the algorithms. The computer used in the tests was a 600 Mhz Pentium 3 with 256 MB RAM and Linux OS. All code was compiled with GCC 3.2.1 and full optimization switched on.

The tests involved patterns of lengths 10, 20, and 30, and with each pattern length m the tested k values were $1..\lfloor m/2 \rfloor$. There were 50 randomly picked patterns for each (m, k)-combination. The searched text was a 10 MB sample from Wall Street Journal articles taken from the TREC-collection [4].

The version of the algorithm of Baeza-Yates & Navarro was the one from [10], which includes a smart mechanism to keep only a required part of the automaton active when it needs several bit-vectors. As the patterns lengths were $\leq w = 32$, the other two algorithms did not need such a mechanism.

Fig. 2 shows the results. In general the algorithms compare quite similarly to each other with and without our modification to use the Damerau edit distance. It is seen that with the Levenshtein edit distance the algorithm of Wu & Manber becomes slowest when $k \geq 4$, whereas with the Damerau edit distance it becomes slowest already at $k = 3$. The algorithm of Baeza-Yates & Navarro is typically the fastest for low error levels irrespective of which of the two distances we use. But

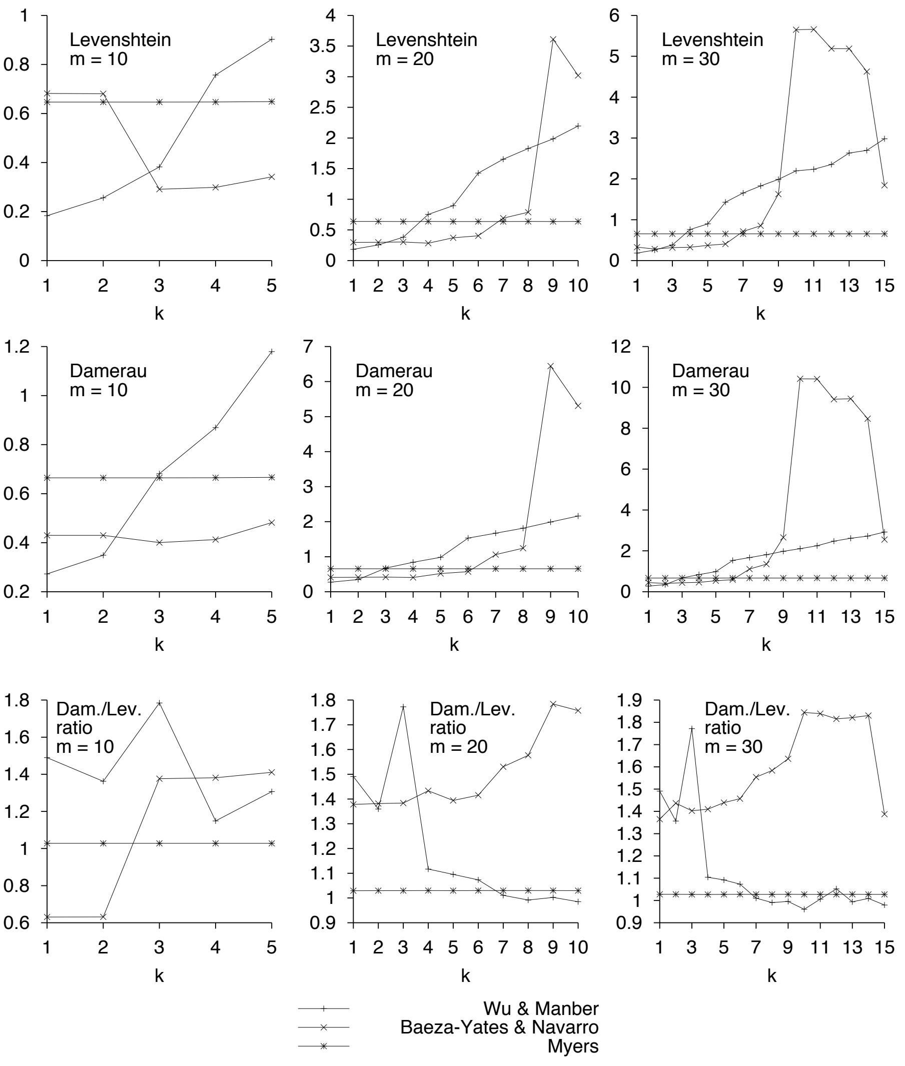

Fig. 2. The two first rows show the average time for searching a pattern from a 10 MB sample of Wall Street Journal articles taken from TREC-collection. The first row shows the results for the Levenshtein edit distance and the second row for the Damerau edit distance. The third row shows the ratio of the run times with and without the modification.

its advantage over the algorithm of Myers becomes smaller under the Damerau edit distance. The algorithm of Myers is affected very little by the modification, and it is the fastest algorithm when the error level k/m is large and the algorithm of Baeza-Yates & Navarro needs more bit-vectors in representing the automaton.

The algorithm of Baeza-Yates & Navarro behaved oddly with the Levenshtein edit distance in the case $m = 10$ and $k < 3$. We found no other reason than some intrinsic property of the compiler optimizer or the processor pipeline for the bad performance with these two values (even worse than the version modified to use the Damerau edit distance).

5 Conclusions

Bit-parallel algorithms are currently the fastest approximate string matching algorithms when Levenshtein edit distance is used. In particular the algorithms of Wu & Manber [15], Baeza-Yates & Navarro [1] and Myers [7] dominate the field when the pattern length and the error level are moderate [8]. In this paper we showed how these algorithms can be modified to use the Damerau edit distance, which is an important distance especially in natural language [5]. Our modification adds only a constant amount of work per bit-vector the algorithm needs in encoding the pattern, and it is general in that essentially the same modification works with all the above-mentioned three bit-parallel algorithms. It also improves upon Navarro's [9] previous modification of the algorithm of Wu & Manber to use the Damerau edit distance. In the experiments we found that the respective performance of the algorithms is not changed much by the modification.

References

1. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
2. F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. 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. D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.
5. K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.
6. 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.*
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 (SPE)*, 31:1265–1312, 2001.
10. G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. *Algorithmica*, 30(4):473–502, 2001.
11. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
12. Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
13. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
14. A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.
15. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.