# A Note on Bit-Parallel Alignment Computation

Heikki Hyyrö

PRESTO, Japan Science and Technology Agency

e-mail: `Heikki.Hyyro@cs.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. Typically the allowed edit operations are one or more of the following: an insertion, a deletion or a substitution of a character, or a transposition between two adjacent characters. Simple edit distance allows the first two operation types, Levenshtein edit distance the first three, and Damerau distance all four. There exist very efficient $O(\lceil m/w \rceil n)$ bit-parallel algorithms for computing each of these three distances, where $m$ is the length of $A$, $n$ is the length of $B$, and $w$ is the computed word size. In this paper we discuss augmenting the bit-parallel algorithms to recover an optimal alignment between $A$ and $B$. Such an alignment depicts how to transform $A$ into $B$ by using $ed(A, B)$ operations, where $ed(A, B)$ is the used edit distance (one of the three mentioned above). Previously Iliopoulos and Pinzon have given such an algorithm for the longest common subsequence, which in effect corresponds to the simple edit distance. We propose a simpler method, which is faster and also more general in that our method can be used with any of the above three distances.

**Keywords:** Longest common subsequence, Levenshtein edit distance, Damerau edit distance, bit-parallelism, edit script, alignment

## 1 Introduction

Edit distance is a classic measure of similarity between two strings. It is generally defined as the minimum number of edit operations that are needed in order to transform one of the strings into the other. There are different types of distances depending on what kind of operations are allowed. Two common and widely studied distances are simple edit distance and Levenshtein edit distance [Lev66]. The simple edit distance permits a single edit operation to insert or delete a character. In addition to these two, Levenshtein distance allows also the operation of substituting a character with another. Damerau distance [Dam64], which is mainly used in spelling correction, extends the Levenshtein distance by allowing a fourth operation of transposing two adjacent characters. The simple edit distance is often used indirectly in its dual form of computing the length of the longest common subsequence between the two strings. Fig. 1 shows an example of these edit distances.

Throughout this paper we will use the following notation. $A_i$ is the $i$th character of a string $A$, and $A_{i..j}$ is the substring of $A$ that begins from its $i$th character and

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **a)** | D: | go**l**d → god | **b)** | D: | go**l**d → god | **c)** | T: g**ol**d → g**lo**d |
| | I: | god → g**l**od | | I: | god → g**l**od | | S: glo**d** → glo**w** |
| | D: | glo**d** → glo | | S: | glo**d** → glo**w** | | |
| | I: | glo → glo**w** | | | | | |

Figure 1: An example of editing the string $A$ = “gold” into the string $B$ = “glow”. Figure a) uses only insertions (I) and deletions (D), as permitted by the simple edit distance. Figure b) corresponds to Levenshtein edit distance and uses also a substitution (S). Figure c) corresponds to Damerau distance that permits also the operation of transposing two adjacent characters (T).

ends at its $j$th character. If $i > j$, we define $A_{i..j}$ to denote the empty string $\epsilon$. String $C$ is a subsequence of $A$ if $A$ can be transformed into $C$ by deleting zero or more characters from $A$.

The two compared strings will be denoted by $A$ and $B$. We will denote the length of $A$ by $m$ and the length of $B$ by $n$. The edit distance between $A$ and $B$ is denoted by $ed(A, B)$. We distinguish between different types of edit distance by using a subscript: We refer to the simple edit distance, Levenshtein distance and Damerau distance between $A$ and $B$ as $ed_S(A, B)$, $ed_L(A, B)$ and $ed_D(A, B)$, respectively. The length of the longest common subsequence between $A$ and $B$ is LLCS$(A, B)$.

The classic and very flexible solution for computing various edit distances is based on dynamic programming. The three distances we discuss can be computed in $O(mn)$ time by filling an $(m+1) \times (n+1)$ dynamic programming matrix. Depending on the particular distance, several enhancements over the basic scheme have been proposed. We refer the reader to for example [Nav01, Gus97, BHR00] for an overview on the various algorithms for the different distances. For our purposes it is sufficient to mention that the $O(\lceil m/w \rceil n)$ bit-parallel algorithms [AD86, Mye99, CIPR01, Hyy03, Hyy04], where $w$ is the computer word size, are typically very practical choices at least when the alphabet size is moderate (e.g. ASCII character set). These algorithms encode the differences between adjacent cells in the dynamic programming matrix into computer words of length $w$ by using a constant number of bits per cell, and are then able to compute all values within a single word in parallel.

In this paper we consider the case of editing $A$ into $B$. There are one or more minimal *edit scripts* that correspond to the value $ed(A, B)$. A minimal edit script describes a set of $ed(A, B)$ operations which transform $A$ into $B$. There are applications, such as file comparison, where this information is essential. An edit script can be recovered from the dynamic programming matrix once it has been filled.

One common way to describe an edit script is to show the corresponding alignment for $A$ and $B$. In this paper we discuss a simple scheme to efficiently recover an optimal alignment after the difference-encoded counterpart of the dynamic programming matrix has been computed by a bit-parallel algorithm. Previously Iliopoulos and Pinzon [IP02] have proposed this type of a method for recovering a longest common subsequence for $A$ and $B$. There is a close relationship between $LLCS(A, B)$ and $ed_S(A, B)$: $ed_S(A, B) = m + n - 2 \times LLCS(A, B)$. The longest common subsequence gives effectively the same information as an optimal alignment for $ed_S(A, B)$. But the method of Iliopoulos and Pinzon is unnecessarily complicated and specifically designed for LLCS$(A, B)$ (or $ed_S(A, B)$). Our scheme is simpler, can be used with any of the three discussed edit distances, and we also verify experimentally that it is considerably faster than the method of Iliopoulos and Pinzon.

## 2 Dynamic programming

The dynamic programming methods fill an $(m+1) \times (n+1)$ dynamic programming matrix $D$, in which each cell $D[i, j]$ will eventually hold the value $ed(A_{1..i}, B_{1..j})$. As a specific example we will review the basic dynamic programming solution for Levenshtein edit distance. The other two distances are computed in a very similar manner.

The first step is to fill trivially known boundary values. Since all three distances permit insertions and deletions, they share the same boundary values $D[0, j] = ed(A_{1..0}, B_{1..j}) = ed(\epsilon, B_{1..j}) = j$ and $D[i, 0] = ed(A_{1..i}, B_{1..0}) = ed(A_{1..i}, \epsilon) = i$. The remaining cells of $D$ are then computed by using an appropriate recurrence. The complete recurrence for Levenshtein distance is as follows.

$D[i, 0] = i$, for $i \in 0 \ldots m$.
$D[0, j] = j$, for $j \in 0 \ldots n$.
When $1 \leq i \leq m$ and $1 \leq j \leq n$,

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

Here the three options in the minimum clause correspond to deleting $A_i$, inserting $B_j$ after $A_i$, or substituting $A_i$ with $B_j$, respectively.

A common way of computing the cells is to proceed in a columnwise manner. First the cells $D[1, 1], D[2, 1], \ldots D[m, 1]$, then the cells $D[1, 2], D[2, 2], \ldots D[m, 2]$, and so on until column $n$. Finally the desired edit distance is $ed(A, B) = D[m, n]$.

When matrix $D$ has been filled, a sequence of $ed(A, B)$ edit operations that transforms $A$ into $B$ can be recovered by backtracking from the cell $D[m, n]$ towards the cell $D[0, 0]$. At each step we move from $D[i, j]$ into $D[i-1, j]$, $D[i-1, j-1]$ or $D[i, j-1]$, the only restriction being that the consecutively visited cell values have to correspond to a minimal choice made in the recurrence. The corresponding edit operations can then be recorded along the way until the cell $D[0, 0]$ is reached.

Computing LLCS$(A, B)$ can be done in similar manner. Let $L$ be the corresponding $(m+1) \times (n+1)$ dynamic programming matrix. The condition $L[i, j] = \text{LLCS}(A_{1..i}, B_{1..j})$ will hold after $L$ has been filled according to the following recurrence.

$L[i, 0] = 0$, for $i \in 0 \ldots m$.
$L[0, j] = 0$, for $j \in 0 \ldots n$.
When $1 \leq i \leq m$ and $1 \leq j \leq n$,

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

Instead of explicitly enumerating the operations of an edit script, similar information can be given in the form of an alignment between $A$ and $B$. An alignment shows the strings $A$ and $B$ on two rows in such manner, that each others counterpart characters in $A$ and $B$ are placed into the same horizontal position (the same column). In case of inserting or deleting, one of the counterparts is an empty space. In case of a substitution, the counterpart is the substituted character. In case of a transposition between two adjacent characters, the two pairs are shown above each other. And obviously, characters that are matched in the edit sequence are each others counterparts. Fig. 2 shows an example.

| | | s | u | r | g | e | r | y |
|---|---|---|---|---|---|---|---|---|
| | **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s | 1 | **0** | 1 | 2 | 3 | 4 | 5 | 6 |
| u | 2 | 1 | **0** | 1 | 2 | 3 | 4 | 5 |
| r | 3 | 2 | 1 | **0** | 1 | 2 | 3 | 4 |
| v | 4 | 3 | 2 | 1 | **1** | 2 | 3 | 4 |
| e | 5 | 4 | 3 | 2 | 2 | **1** | **2** | 3 |
| y | 6 | 5 | 4 | 3 | 3 | 2 | 2 | **2** |

```
s  u  r  v  e     y
         |     |
s  u  r  g  e  r  y
```

Figure 2: On the left: The dynamic programming matrix $D$ for computing Levenshtein edit distance between the strings $A$ = "survey" and $B$ = "surgery". The cells that are traversed during a backtrack from $D[6,7]$ into $D[0,0]$ are shown in bold. It goes as follows: $D[6,7] \rightarrow D[5,6]$: match $A_6 = B_7$. $D[5,6] \rightarrow D[5,5]$: insert $B_6$. $D[5,5] \rightarrow D[4,4]$: match $A_5 = B_5$. $D[4,4] \rightarrow D[3,3]$: substitute $A_4$ with $B_4$. $D[3,3] \rightarrow D[2,2] \rightarrow D[1,1] \rightarrow D[0,0]$: match $A_{1..3} = B_{1..3}$. On the right: an optimal alignment that corresponds to the shown edit script trace in $D$. The inserted 'r' has a space as its counterpart.

## 3 Bit-parallel algorithms

In general, bit-parallel algorithms are based on exploiting the fact that computers process information in chunks of $w$ bits, where $w$ is the computer word size. If one can encode several data items into a single length-$w$ bit-vector, then it may be possible to manipulate several items in parallel during a single computer operation. Of course the feasibility of this scheme depends highly on the type of the information and the operations one wishes to perform on them. The three types of edit distance that we discuss have turned out to be very suitable for bit-parallel computation. The bit-parallel algorithms for them reach the highest possible level of parallelism, manipulating $w$ items at once.

In this paper we use the following notation in describing bit-operations: '&' 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. The $i$th 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 a 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 $V = 10^2 1^3 0$.

The bit-parallel algorithms we build upon rely on the adjacency properties of $D$ or $L$. It is known that two adjacent cells in a column or a row differ by at most 1. That is, the conditions $D[i-1,j]-1 \leq D[i,j] \leq D[i-1,j]+1$ and $D[i,j-1]-1 \leq D[i,j] \leq D[i,j-1]+1$ hold. In $L$ the condition is stricter: the values never decrease along a column or a row, and so $L[i-1,j] \leq L[i,j] \leq L[i-1,j]+1$ and $L[i,j-1] \leq L[i,j] \leq L[i,j-1]+1$. These rules allow us to encode the values in each column of $D$ by the following length-$m$ bit-vectors:

**The vertical positive delta vector $VP_j$:**
$VP_j[i] = 1$ if and only if $D[i,j] - D[i-1,j] = 1$.

**The vertical negative delta vector $VN_j$:**
$VN_j[i] = 1$ if and only if $D[i,j] - D[i-1,j] = -1$.

Now $D[i,j] = D[0,j] + \sum_{k=1}^{i}(VP_j[k] - VN_j[k])$ if we interpret a set bit as $+1$ and an unset bit as 0.

In case of the simple edit distance $ed_S(A,B)$, only the vector $VP_j$ is needed if one computes LLCS$(A,B)$ instead and defines $VP_j$ to encode differences in $L$ instead of $D$.

In general we need $O(\lceil m/w \rceil)$ bit-vectors of length $w$ in order to represent a length-$m$ bit-vector. When each length-$w$ segment of the bit-vectors can be computed in constant time, the overall running time for computing the vertical delta vectors for $j = 1 \ldots n$ is $O(\lceil m/w \rceil n)$. Among the discussed edit distances, the first $O(\lceil m/w \rceil n)$ bit-parallel algorithm was given by Allison and Dix [AD86] for computing LLCS$(A,B)$. Later Myers [Mye99] presented an $O(\lceil m/w \rceil n)$ algorithm for approximate string matching under Levenshtein edit distance. That algorithm can be easily modified for computing edit distance [HN02], and a way to modify it for Damerau distance was presented in [Hyy03]. Crochemore et al. [CIPR01] and Hyyrö [Hyy04] have given alternative $O(\lceil m/w \rceil n)$ algorithms for computing LLCS$(A,B)$.

We will not go into details of the bit-parallel algorithms themselves. For this paper the relevant thing is that we may assume that all vectors $VP_j$ and $VN_j$ for $D$ (or $VP_j$ for $L$) may be computed in $O(\lceil m/w \rceil n)$ time. We concentrate on the post-processing step of recovering an alignment once these vectors are known for $j = 1 \ldots n$.

## 4 Tracing a script

In case of the whole matrix $D$, recovering an alignment is simple since the backtracking procedure can directly check the values in the neighboring cells. But this is slightly more complicated if we assume only the existence of the vertical delta vectors $VP_j$ and $VN_j$. The method of Iliopoulos and Pinzon [IP02] resorted to computing also horizontal differences to overcome this difficulty, although the algorithm in itself did not directly correspond to a backtracking procedure. We now note some rules that enable backtracking in $D$ when only the vertical deltas are known.

Let us begin by considering the longest common subsequence computation. Assume that the backtracking procedure in matrix $L$ is in the cell $L[i,j]$. From the recurrence of $L$ we know that we can move vertically to the cell $L[i-1,j]$ if and only if $L[i,j] = L[i-1,j]$ (or $VP_j[i] = 0$). This poses no problems. Therefore let the first phase of the backtracking involve going vertically towards row 0 as long as possible, that is, as long as the corresponding bits in the vector $VP_j$ are not set. Once we cannot move vertically, we are either at row 0 or the condition $L[i-1,j] = L[i,j]-1$ holds. In the first case we are done, as the remaining steps must go directly along row 0 to $L[0,0]$. In the latter case we know that $L[i-1,j-1] \leq L[i-1,j] = L[i,j]-1$. Consider now having the equality $L[i-1,j-1] = L[i,j-1]$ at row $i$ in column $j-1$. Since the adjacency property states that $L[i,j-1] \geq L[i,j]-1$, we then have $L[i-1,j-1] = L[i,j-1] = L[i,j]-1$, and the only possible source for the value $L[i,j]$ is a match $A_i = B_j$. On the other hand, if $L[i-1,j-1] = L[i,j-1]-1$, then either $L[i,j-1] = L[i-1,j]+1 = L[i,j]$ or $L[i,j-1] = L[i-1,j] = L[i,j]-1$. The latter case would also have the value $L[i-1,j-1] = L[i,j]-2$, which contradicts with the recurrence for $L$ as there is no possible source for the value $L[i,j]$. Thus the former case $L[i,j-1] = L[i-1,j]+1 = L[i,j]$ holds and we can move horizontally to the cell $L[i,j-1]$.

Now we have the following rule for cell $L[i,j]$:

**$VP_j[i] = 0$:** Move to the cell $L[i-1,j]$, and record that the counterpart of $A_i$ is a space.

**$VP_j[i] = 1$:** Move to column $j-1$ and check the value $VP_{j-1}[i]$. If it is 1, then go to the cell $L[i,j-1]$ and record that the counterpart of $B_j$ is a space. Otherwise go to the cell $L[i-1,j-1]$ and record that the counterpart of $A_i$ is $B_j$ (and they match).

Let us now consider Levenshtein or Damerau distances in similar manner. If the backtracking is in cell $D[i,j]$, we can go to the cell $D[i-1,j]$ if and only if $VP_j[i] = 1$. If $VP_j[i] = 0$, let us consider when the only choice for the backtracking is to move into $D[i,j-1]$. That happens only if we have $D[i,j-1] = D[i,j]-1$ and $D[i,j] = D[i-1,j-1]$. But in this case we must have $D[i,j-1] = D[i-1,j-1]-1$, a condition we can check from $VN_{j-1}[i]$. This gives the following backtracking rule for cell $D[i,j]$ for Levenshtein and Damerau distances.

**$VP_j[i] = 1$:** Move to the cell $D[i-1,j]$, and record that the counterpart of $A_i$ is a space.

**$VP_j[i] = 0$:** Move to column $j-1$ and check the value $VN_{j-1}[i]$. If it is 1, then go to the cell $D[i,j-1]$ and record that the counterpart of $B_j$ is a space. Otherwise go to the cell $D[i-1,j-1]$ and record that the counterpart of $A_i$ is $B_j$ (it may be a match, a substitution, or a part of a transposed character-pair).

These rules for the two distances are inherently similar and enable composing a single procedure for backtracking that works with all three distances. One just needs to feed the checked vectors as parameters, possibly in negated form. Fig. 3 shows the pseudocode for this kind of a general scheme. The shown pseudocode operates on bit-vectors of length $m$.

Our basic backtracking procedure takes $O(m+n)$ time. If implemented exactly as originally described in [IP02], the method of Iliopoulos and Pinzon takes $O(\lceil m/w \rceil n)$ time in the post-processing stage. But a simple modification of concentrating only on the currently processed length-$w$ part of the matrix column enables us to implement it in $O(\lceil m/w \rceil + n)$ time. Also our backtracking method can be modified in a corresponding way to have the running time $O(\lceil m/w \rceil + n)$.

```
RecoverAlignment($delta1$, $delta2$)
1.  $i \leftarrow m$, $j \leftarrow n$
2.  While $i > 0$ and $j > 0$ Do
3.      If the bit $delta1_j[i]$ is set Then
4.          Output the pair ($A_i$, ‘ ’)
5.          $i \leftarrow i - 1$
6.      Else
7.          If the bit $delta2_{j-1}[i]$ is set Then
8.              Output the pair (‘ ’, $B_j$)
9.          Else
10.             Output the pair ($A_i$, $B_j$)
11.             $i \leftarrow i - 1$
12.         $j \leftarrow j - 1$
13. While $i > 0$ Do
14.     Output the pair ($A_i$, ‘ ’)
15.     $i \leftarrow i - 1$
16. While $j > 0$ Do
17.     Output the pair (‘ ’, $B_j$)
18.     $j \leftarrow j - 1$
```

Figure 3: The general scheme for recovering an alignment from the vertical delta vectors. In the case of matrix $L$ for LLCS$(A, B)$, the corresponding alignment is recovered by executing **RecoverAlignment(**$\sim VP$**,** $VP$**)**. In the case of matrix $D$, one should execute **RecoverAlignment(**$VP$**,** $VN$**)**.

## 5 Test results

We implemented the backtracking procedure and tested it in the case of $L$ matrix of longest common subsequence computation. This choice was made so that we could compare its performance against the method of Iliopoulos and Pinzon. Both tested methods were implemented by us. Instead of the original $O(\lceil m/w \rceil n)$ post-processing phase, we used a more efficient $O(\lceil m/w \rceil + n)$ scheme in the method of Iliopoulos and Pinzon. Our method used the basic $O(m+n)$ backtracking scheme. Despite the fact that backtracking is a rather low-cost procedure in comparison to the cost of first computing the vertical delta vectors, we chose to measure overall execution time that includes both computing the vectors and backtracking in them. The tested strings were randomly generated, and we used alphabet sizes 4 and 25. The computer was a 1.3 Ghz Intel Pentium M with 256 MB RAM and Windows XP operating system, and the code was compiled with MS Visual C++ 6.0 with full optimization options. The number of repetitions varied depending on the case in order to get feasible timings. The results are shown in Fig. 4. The numbers show the percentage of the run time of the method of Iliopoulos and Pinzon (IP) when compared to our scheme. Even though the backtracking should have a very low cost in comparison to the computation of the vectors, using our method instead has a noticeable impact even for a relatively high $m, n$.

| $n = m$ | 30 | 50 | 100 | 300 | 500 | 1000 | 3000 | 5000 |
|---|---|---|---|---|---|---|---|---|
| IP($\sigma = 4$) | 195 | 155 | 145 | 114 | 111 | 106 | 103 | 103 |
| IP($\sigma = 25$) | 211 | 160 | 156 | 117 | 114 | 106 | 102 | 101 |

Figure 4: The results for the method of Iliopoulos and Pinzon as a percentage of the run time of our method. We tested with alphabet sizes $\sigma = 4$ and $\sigma = 25$.

## 6 Conclusion

Bit-parallel algorithms are in many cases the most efficient choice in practice for computing the simple, Levenshtein or Damerau distance, or for computing the length of the longest common subsequence. In this paper we proposed and evaluated a simple and uniform way to recover an optimal alignment for the compared strings after a bit-parallel algorithm has computed all vertical delta vectors of the corresponding dynamic programming matrix. We found that our method is more efficient than the previous method proposed by Iliopoulos and Pinzon [IP02]. Our method has also the benefit that the same scheme works with all three distances we discussed. The discussed methods for retrieving an alignment need $O(\lceil m/w \rceil n)$ space for storing the vertical delta vectors for $j = 1 \ldots n$. Thus if $A$ and/or $B$ are long, the space requirements may become too large. In such cases one should use for example the divide-and-conquer scheme proposed by Hirschberg [Hir78] that requires only linear space. In [CIP01] Crochemore, Iliopoulos and Pinzon discussed combining that scheme with bit-parallel LLCS($A, B$) computation.

## References

[AD86] L. Allison and T. L. Dix. A bit-string longest common subsequence algorithm. *Information Processing Letters*, 23:305–310, 1986.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE'00)*, pages 39–48, 2000.

[CIP01] M. Crochemore, C. S. Iliopoulos, and Y. J. Pinzon. Speeding-up Hirschberg and Hunt-Szymanski LCS algorithms. In *Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 59–67. IEEE CS Press, 2001.

[CIPR01] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80:279–285, 2001.

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

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[Hir78] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Information Processing Letters*, 7(1):40–41, 1978.

[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.

[Hyy03] H. Hyyrö. Bit-parallel approximate string matching algorithms with transposition. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, LNCS 2857, pages 66–79, 2003.

[Hyy04] H. Hyyrö. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, 2004.

[IP02] C. S. Iliopoulos and Y. J. Pinzon. Recovering an lcs in $O(n^2/w)$ time and space. *Columbian Journal of Computation*, 3(1):41–51, 2002.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

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

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.