Bit-Parallel LCS-length Computation Revisited

Heikki Hyyrö [*][†]

Abstract

The *longest common subsequence* (LCS) is a classic and well-studied measure of similarity between two strings A and B. This problem has two variants: determining the length of the LCS (LLCS), and recovering an LCS itself. In this paper we address the first of these two. Let m and n denote the lengths of the strings A and B, respectively, and w denote the computer word size. First we give a slightly improved formula for the bit-parallel $O(\lceil m/w \rceil n)$ LLCS algorithm of Crochemore et al. [4]. Then we discuss the relative performance of the bit-parallel algorithms and compare our variant against one of the best conventional LLCS algorithms. Finally we propose and evaluate an $O(\lceil d/w \rceil n)$ version of the algorithm, where d is the simple (indel) edit distance between A and B.

1 Introduction

Computing the *longest common subsequence* (LCS) between two strings is a classic problem in computer science. The LCS, or the *length* of the LCS (LLCS), between two strings gives information about their similarity, and this may be used in various applications such as text editing, file comparison or computational biology (see e.g. [16, 6]).

Let us begin by introducing some basic notation. We denote the used alphabet by Σ, and the alphabet size by σ. The length of a string A is denoted by $|A|$, A_i is the ith character of A, and $A_{i..j}$ denotes the *substring* of A that begins from its ith character and ends at the jth character. If $j < i$, we interpret $A_{1..j}$ to be the empty string ϵ. If A is nonempty, the first character of A is A_1 and $A = A_{1..|A|}$. The string C is a *subsequence* of the string A if C can be derived by deleting zero or more characters from A. Thus C is a subsequence of A if the characters $C_1 \ldots C_{|C|}$ appear in the same order, but not necessarily consecutively, in A. The string C is the *longest common subsequence* of the strings A and B, if C is a subsequence of both A and B, and no longer string with this property exists. We will use the notation LCS(A, B) to denote a longest common subsequence between the strings A and B, and LLCS(A, B) denotes the length of LCS(A, B). For simplicity, and to follow a common convention in literature, we will use the notation $m = |A|$ and $n = |B|$. Without loss of generality, we also assume that $m \leq n$.

There is a close connection between LLCS(A, B) and a *simple edit distance* between A and B. Let $ed_{id}(A, B)$ denote the edit distance between A and B that is defined as the minimum number of single-character insertions and deletions that are needed in order to transform A into B (or vice versa). Then the equality $2\times$LLCS$(A, B) = n + m - ed_{id}(A, B)$ holds (e.g. [5]).

[*]PRESTO, Japan Science and Technology Agency
[†]Department of Computer Sciences, University of Tampere, Finland; Heikki.Hyyro@cs.uta.fi

The problem of computing LCS(A,B) and/or LLCS(A,B) has received a lot of attention. Generally the algorithms that solve LCS(A,B) are built upon an algorithm for solving LLCS(A,B). In this paper we concentrate on the latter problem, that of computing LLCS(A,B). The basic $O(mn)$ solution by dynamic programming was given by Wagner and Fisher [18], and there has subsequently appeared numerous other algorithms for the problem (e.g. [8, 9, 14, 2, 13, 19, 15, 4]). We refer the reader to for example [3, 6] for a more detailed overview on these. Theoretical results on the problem include that the lower bound of the complexity is $\Omega(n \log m)$ [7], and that the quadratic worst-case complexity of the basic dynamic programming algorithm cannot be improved by any algorithm that uses 'equal/nonequal' comparisons between characters [1]. The theoretically fastest existing algorithm is the $O(n^2/\log n)$ "four Russians"-algorithm of Masek and Paterson [12]. In a rather comprehensive survey by Bergroth et al. [3], the algorithms of Kuo and Cross [11] and Rick [15] were found to be the fastest in practice. This survey, however, did not include the so-called *bit-parallel* algorithms of Allison and Dix [2] or Crochemore et al. [4], which have a run time $O(\lceil m/w \rceil n)$, where w is the computer word size, and should be very practical.

After introducing the underlying basic dynamic programming solution in Section 2, the contents of the rest of the paper are as follows.

In Section 3 we review the $O(\lceil m/w \rceil n)$ bit-parallel algorithms of Allison and Dix [2] and Crochemore et al. [4], and give a new variant of the latter. Our variant makes either one operation or one table lookup less, and becomes thus the most efficient bit-parallel LLCS algorithm in terms of the number of operations. In this part we also discuss the practical differences between the different variants, and conclude that the experimental comparison shown in [4] is extremely biased. Since we are unaware of any previous work where the bit-parallel algorithms are compared with the most practical conventional LLCS algorithms, one contribution of the current paper is a comparison between the bit-parallel approach and the algorithm of Kuo and Cross [11]. Based on the experiments reported in [3], the latter algorithm serves as a good point of comparison. We find that the bit-parallel approach indeed seems to be the most practical choice at least when the alphabet size is at most 256.

In Section 4 we propose an $O(\lceil d/w \rceil n)$ bit-parallel algorithm for computing LLCS(A,B), where $d = ed_{id}(A,B)$ is the simple edit distance between A and B. This algorithm is inherently similar to the diagonal restriction scheme that Ukkonen [17] proposed in the context of computing Levenshtein edit distance (otherwise similar to simple edit distance, but permits also substituting one character with another). A quite similar method for bit-parallel Levenshtein edit distance computation was presented in [10]. Our experiments find that combining a bit-parallel algorithm with a restriction scheme can result in a considerable speedup in a setting where m and n are similar and/or one wishes to check whether the value of LLCS(A,B) is larger than some predetermined threshold.

2 Basic Dynamic Programming

The basic $O(mn)$ solution [18] for computing LLCS(A,B) is based on filling an $(m+1) \times (n+1)$ dynamic programming matrix L, in which each cell $L[i,j]$ will eventually hold the value LLCS$(A_{1..i}, B_{1..j})$. This is achieved by using the following well-known recurrence.

Recurrence 1.

$$\begin{array}{l} \text{When } 0 \le i \le m \text{ and } 0 \le j \le n: \\ L[i,0] = 0, \quad L[0,j] = 0. \\ L[i,j] = \begin{cases} L[i-1,j-1]+1, \text{ if } A_i = B_j. \\ \max(L[i-1,j], L[i,j-1]), \text{ otherwise.} \end{cases} \end{array}$$

The first part of Recurrence 1 sets the trivially known boundary values $L[0,j] = \text{LLCS}(\epsilon, B_{1..j})$ $= 0$ and $L[i,0] = \text{LLCS}(A_{1..i}, \epsilon) = 0$ for $i = 0..m$ and $j = 0..n$. Then the other values can be filled by using the latter part of the recurrence.

The space requirement for computing LLCS(A,B) is $O(m)$, because L can be filled in columnwise manner so that only the values in the $(j-1)$th column are needed when the jth column is computed. If L is recorded completely in memory, the corresponding subsequence, LCS(A,B), can be computed by backtracking from the cell $L[m,n]$. From a cell $L[i,j]$ one can move into any such cell $L[i-1,j-1]$, $L[i-1,j]$ or $L[i,j-1]$ whose value can be the source of the value of $L[i,j]$ in terms of the dynamic programming recurrence. It is also possible to compute LCS(A,B) with only linear space by using the divide-and-conquer technique proposed by Hirschberg [8]. The scheme uses regular dynamic programming as a sub-procedure.

3 Bit-Parallel Algorithms

So-called *bit-parallel* algorithms are based on taking advantage of the fact that computers process data in chunks of w bits, where w is the computer word size. This may make it possible to encode several data items of an algorithm into a single computer word, and then operate on several data items in parallel during a single computer operation. In the basic setting, most current computers operate on 32 bits. However, most current processors also support using $w = 64$ or even $w = 128$ through specialized instruction set extensions such as MMX or SSE/SSE2. And moreover, the introduction of the AMD Athlon64 processor into the mainstream market has opened the way for 64 bits to emerge as the standard word size.

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 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 superscripts to denote repetition. As an example let $V = 1011010$ be a bit vector. Then $V[1] = V[3] = V[6] = 0$, $V[2] = V[4] = V[5] = V[7] = 1$, and we could also write $V = 101^2010$ or $V = 101(10)^2$.

The first bit-parallel algorithm for computing LLCS(A,B) is due to Allison and Dix [2]. Quite recently Crochemore et al. [4] presented a second algorithm. Both of these algorithms have a run time $O(\lceil m/w \rceil n)$. Let us use the acronyms AD and CIPR, respectively, in referring to these two algorithms.

The bit-parallel algorithms for computing LLCS(A,B) are essentially based on the following observations derived from Recurrence 1. In them we consider specifically columns, but, due to symmetry, analogous observations hold also for the rows of L. Let $M(x,j)$ denote the minimum index for which $L[i,j] = x$, and define $M(x,j) = m+1$ if no such index i exists.

A well-known property of L is that the next value along a column is the same or larger by one than the previous value.

Observation 1.
$L[i, j] \in \{L[i-1, j], L[i-1, j]+1\}$.

From this it follows that if the locations of increment in column j are known, then we can calculate any value $L[i, j]$.

Observation 2.
The values in any column of L may be described by recording for it the positions where the value along the column increases by one, that is, such $i \in 1 \ldots m$ where $L[i, j] = L[i-1, j]+1$.

If $M(x, j) = i$ and $i > 0$, then $L[i-1, j] < x$, which by Observation 1 implies $L[i-1, j] = x-1$. On the other hand, since the values along a column never decrease, the condition $L[i, j] = x = L[i-1, j]+1$ implies that $M(x, j) = i$.

Observation 3.
For $i \in 1 \ldots m$, $L[i, j] = x = L[i-1, j]+1$ if and only if $M(x, j) = i$.

The value $L[i, j] = x$ that corresponds to $M(x, j)$ must propagate from column $j-1$, as otherwise we would have the contradiction $L[i-1, j] = x$. Recurrence 1 gives two options for this to happen. Either $L(i, j-1) = x$, or $L(i-1, j-1) = x-1$ and $A_i = B_j$.

Observation 4.
$M(x, j) = \min(i \mid (i = M(x, j-1)) \vee ((A_i = B_j) \wedge (M(x-1, j-1) < i))\ )$.

Following Observations 1 and 2, it is possible to encode the values within a column of L into a single length-m bit-vector. Let us define V_j as a length-m bit-vector whose ith bit is set if and only if $L[i, j] = L[i-1, j]+1$. Then clearly $L[m, j] = \sum_{k=1}^{m} V_j[k]$, which is the same as the number of set bits in V_j. Also, let V'_j denote the one's complement of V_j, that is, $V'_j = \sim V_j$. The algorithm AD encodes the columns of L by V_j, and the algorithm CIPR by V'_j. For now we discuss mainly V_j as the relationship between it and V'_j is straightforward. In addition, both algorithms use precomputed length-m pattern matching bit-vectors PM_λ, where for any character λ in the used alphabet the bit $PM_\lambda[i]$ is set if and only if $A_i = \lambda$. The crux of these bit-parallel algorithms is to compute V_j efficiently when the vectors V_{j-1} and PM_{B_j} are known. The value LLCS(A, B) can then be computed by counting the number of set bits in the vector V_n, which can be done in $O(\log_2 m)$ time (see e.g. [20]). Note that initially $V_0 = 0^m$ and $V'_0 = 1^m$. Let us make one more observation.

Observation 5.
When $i \in 1 \ldots m$, $V_j[i] = 1$ if and only if $M(x, j) = i$ for some x.

The basic principle behind AD and CIPR is to consider each column-j row-segment $u+1 \ldots v$, where $u = M(x-1, j-1)$, $v = M(x, j-1)$ and $0 < x \le L[m, j-1]+1$. In this case $u+1 \le v$. Based on Observation 5, the bit $V_{j-1}[v]$ in each such segment corresponds to the value $M(x, j-1)$. In order to cover the case $x = L[m, j-1]+1$, we can assume that there is always also the non-existent bit $V_{j-1}[m+1] = 1$. From Observation 4 we see that the bit $V_{j-1}[v]$, which formerly represented the value $M(x, j-1)$, should be moved into position k in V_j, where $k = \min(i \mid (i = M(x, j-1)) \vee ((A_i = B_j) \wedge (M(x-1, j-1) < i))\ )$. Note that the preceding includes also the possibility of keeping the bit in the same position v. Each row-segment $u+1 \ldots v$ is disjoint, and the goal is to process each segment simultaneously by using bit-parallelism.

3.1 Allison-Dix

Assume that the bit-vectors V_{j-1} and PM_{B_j} are known and set initially $V_j = V_{j-1}$. From Observation 4 we see, that now the set bit $V_j[v]$, which previously represented the value $M(x, j-1)$, should be moved into the first position $k \in u+1 \ldots v-1$ for which $PM_{B_j}[k] = 1$ in order to represent the value $M(x, j)$. If no such k exists, the bit should remain at its current position. If we set $U = V_j \mid PM_{B_j}$, the correct new position for the vth bit is the first (rightmost) set bit in the segment $u+1 \ldots v$ of U: When k is the index of such a first set bit, then $k = \min(i \mid (i = M(x, j-1)) \vee ((A_i = B_j) \wedge (M(x-1, j-1) < i))\ )$. Note that the preceding includes the possibility of replacing the vth bit by itself. By adapting the original work of Allison and Dix, we have the following steps to achieve this change.

1 Set $V_j = V_{j-1}$.

2 Set $U = V_j \mid PM_{B_j}$.

3 Subtract $0^{m-1-u}10^u$ from U in order to set off the first set bit in the segment $U[u+1..v]$.

4 Record all bits that changed in Step 3 by performing an exclusive or (^) between U before and after the subtraction.

5 Perform a bitwise and (&) between the result of Step 4 and the original U from Step 2 in order to discard those changed bits, if any, that the subtraction in Step 3 changed from 0 to 1 in the beginning of $U[u+1..v]$.

Performing the preceding procedure for one segment does not interfere with any other segment. Thus the procedure can be done for all $x = 1 \ldots L[m, j-1]+1$ in parallel. The only change is that instead of subtracting $0^{m-1-u}10^u$, we subtract $(V_{j-1} << 1) \mid 0^{m-1}1$ from U. This will subtract one from the first position of each segment simultaneously. Note that it does not matter that the bit $V_j[m+1] = 1$ does not actually exist. Fig. 1 shows a pseudocode for the whole algorithm as well as computing the PM_λ vectors. The algorithm writes the value V_j over the value V_{j-1} in a single variable V. There are 6 operations per a character of B. We remark that an alternative form for line 5 of the algorithm would be $V \leftarrow U \ \& \ (\sim (U - ((V << 1) \mid 0^{m-1}1)))$.

```
ComputePM(A)
1.  For $\lambda \in \Sigma$ Do $PM_\lambda \leftarrow 0^m$
2.  For $i \in 1 \ldots m$ Do $PM_{A_i} \leftarrow PM_{A_i} \mid 0^{m-i}10^{i-1}$

AD-LLCS(A, B)
1.  ComputePM(A)
2.  $V \leftarrow 0^m$
3.  For $j \in 1 \ldots n$ Do
4.      $U \leftarrow V \mid PM_{B_j}$
5.      $V \leftarrow U \ \& \ ((U - ((V << 1) \mid 0^{m-1}1)) \wedge U)$
6.  Return the number of set bits in $V$
```

Figure 1: Preprocessing the PM-table, and the AD algorithm for computing LLCS(A, B).

3.2 Crochemore et al.

Assume that the bit-vectors $V'_{j-1} = \sim V_{j-1}$ and PM_{B_j} are known, and set initially $V'_j = V'_{j-1}$. Now the unset bit V'_j, which previously represented the value $M(x, j-1)$, should be moved into the first position $k \in u+1 \ldots v-1$ for which $PM_{B_j}[k] = 1$. If no such k exists, the unset bit should not be moved. By imitating from the original work of Crochemore et al., we have the following steps to achieve this move.

1 Set $V'_j = V'_{j-1}$.

2 Set $U = V'_j \ \& \ PM_{B_j}$ in order to obtain only those matching bits set that belong to a segment $k \in u+1 \ldots v-1$.

3 Perform the addition $W = U + V'_j$ in order to unset the bit in the first matching position in the segment $V'_j[u+1..v]$. If the segment contains a match, adding the rightmost match bit will create a carry-effect that sets the bit $V'[v]$ and unsets all bits in-between. If there is more than one match, the effect of adding their bits is that non-rightmost matching positions will again get a set bit.

4 Set $V'_j = W \mid (V'_j \ \& \ (\sim PM_{B_j}))$ in order to set those bits back to 1 that were possibly set to 0 by the carry effect in Step 3. Note that the equality $V'_j = V'_{j-1}$ was still true before this step.

It is again the case that performing the preceding procedure for one segment does not interfere with any other segment. Thus the procedure handles all $M(x, j)$, $x = 1 \ldots L[m, j-1] + 1$, in parallel. Fig. 2 shows a pseudocode for the algorithm. Here we omit using an auxiliary variable U. The basic version of the algorithm makes 5 operations, but as noted by Crochemore et al., each value $\sim PM_{B_j}$ can be precomputed into a table. This reduces the number of operations into 4, but adds a table access. Thus the version with 5 operations is better in practice.

```
CIPR-LLCS($A, B$)
1.  ComputePM($A$)
2.  $V' \leftarrow 1^m$
3.  For $j \in 1 \ldots n$ Do
4.      $V' \leftarrow (V' + (V' \ \& \ PM_{B_j})) \mid (V' \ \& \ (\sim PM_{B_j}))$
5.  Return the number of unset bits in $V'$
```

Figure 2: The CIPR algorithm for computing LLCS(A, B).

3.3 Our Improved Formula

Now we propose an improvement on the CIPR algorithm by changing the way the value $V'_j \ \& \ (\sim PM_{B_j})$ is computed. We note that $V'_j = (V'_j \ \& \ PM_{B_j}) \mid (V'_j \ \& \ (\sim PM_{B_j})) = (V'_j \ \& \ PM_{B_j}) + (V'_j \ \& \ (\sim PM_{B_j}))$, since the vectors $V'_j \ \& \ PM_{B_j}$ and $V'_j \ \& \ (\sim PM_{B_j})$ have no set bits in the same position. This enables us to use the right-hand side of the equation $V'_j \ \& \ (\sim PM_{B_j}) = V'_j - (V'_j \ \& \ PM_{B_j})$. Although the number of operations is the same on both sides, we can take advantage of the fact that the value $V'_j \ \& \ PM_{B_j}$ is used also elsewhere in the original form of the algorithm. Fig. 3 shows a pseudocode for our variant of the algorithm. The number of operations is now 4 without adding any extra precomputation or table lookup costs. Thus, in terms of the number of operations, this becomes the most efficient bit-parallel LLCS algorithm.

```
**OUR-LLCS**$(A, B)$
1.  **ComputePM**$(A)$
2.  $V' \leftarrow 1^m$
3.  **For** $j \in 1 \ldots n$ **Do**
4.      $U \leftarrow V'$ & $PM_{B_j}$
5.      $V' \leftarrow (V' + U) \mid (V' - U)$
6.  **Return the number of unset bits in** $V'$
```

Figure 3: Our algorithm for computing LLCS(A, B).

3.4 Performance notes

The bit-parallel algorithms that we have discussed make a constant number of bit-operations per a character of B, but over bit-vectors of length m. Thus the run time is $O(n)$ if $m \leq w$. As it is possible to simulate a length-m bit-vector in $O(\lceil m/w \rceil)$ time by using $O(\lceil m/w \rceil)$ vectors of length w, the general run time is $O(\lceil m/w \rceil n)$. In practice the matrix L is covered with $\lceil m/w \rceil n$ length-w vectors, and each of these is computed in almost the same manner as in the case where $m \leq w$. The only difference is that now the possible carry bits in the addition/subtraction operations, as well as the wth bit in the case of shifting left, must be recorded for use in processing the vector below the current one. Fig. 4 shows two possible tiling orders in which to compute the vectors that cover L.

Precomputing the PM_λ vectors takes $O(\sigma \lceil m/w \rceil + m)$ time in the general case.

The space requirement of the algorithms is $O(\sigma \lceil m/w \rceil)$, which includes the PM_λ vectors plus the work space for processing L one column at a time, always keeping only the currently needed column in memory.

In [4] Crochemore et al. reported that CIPR is up to 8 times faster than AD in practice. Since the difference in the number of operations between the two algorithms is very tiny, this result is suspicious. A major factor here is how the bit-parallel algorithms are implemented in the general case where $m > w$: it seems that Crochemore et al. used a horribly slow implementation of AD. We have implemented a general version of each of the three discussed bit-parallel algorithms. For each algorithm we used the same framework of covering the

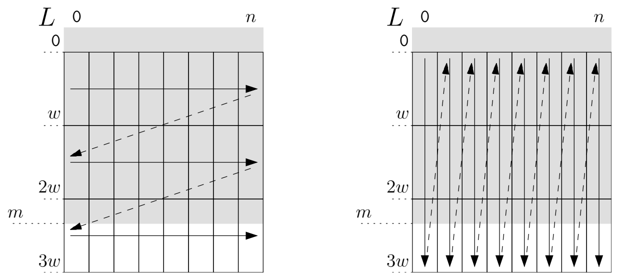

Figure 4: Row- and columnwise tiling orders, respectively, for covering L with length-w bit-vectors.

matrix L by vectors of length w in columnwise manner (see the right side of Fig. 4), so that the only differences were in the details of how each vector is computed.

As we wanted to verify that our test is not biased, we inspected the assembly code generated by the compiler. From this we found that in the most efficient case the overhead of the framework was 11 instructions in the loop that processes the tiles for a single column. The breakdown of this was:

3: Reading the left neighbor vector, recording the currently computed vector, and then advancing in the array of column vectors.

2: Reading the current length-w piece from PM_{B_j} and advancing to the next piece.

2: Enforcing the loop condition.

4: Recording/using the information that has to be propagated from a vector to its neighboring vector below.

In addition each method used 4 or 5 machine instructions, as the instruction "$\mid 0^{m-1}1$" of AD on line 5 of Fig. 1 was absorbed into the information propagation mechanism. Based on this we may claim that our implementations are just about as efficient as possible, the total being 15 machine instructions for the most efficiently compiled version of our variant. When we tested the code on a Sparc Ultra 2 with GCC 3.3.1 compiler and the "-O9" optimization switch, there was absolutely no difference between any of the three variants. On an Athlon64 we found that AD was the slowest, CIPR was roughly 5% faster, and our variant was 15% faster than CIPR. We, however, attribute some of this difference to pure chance about how well the compiler manages to optimize each algorithm. We tried values of $m = n$ in the range $30 \ldots 3000$ using randomly generated strings over various alphabet sizes (although the effect of the alphabet size is negligible in comparing these algorithms with each other). We thus conclude that the experiment shown in [4] was strongly misleading, as the practical difference between any two of the variants is nowhere near 800%.

In [2] Allison and Dix compared AD with the basic dynamic programming algorithm, and in [4] Crochemore et al. compared CIPR only against AD. In order to characterize the relative performance of the bit-parallel approach in comparison to the conventional algorithms, we compared our variant against the $O(M + n(\text{LLCS}(A, B) + \log n))$ algorithm of Kuo and Cross [11] (KC). Here M is the number of matches $A_i = B_j$ within the array L. A rather comprehensive survey by Bergoth et al. [3] found KC to be the fastest or the second fastest, depending on the alphabet size, conventional algorithm for computing $\text{LLCS}(A, B)$ (for example several times faster than dynamic programming). The implementation of KC was from the authors of the above mentioned survey [3].

The computer used in the test was an AMD Athlon64 with 1.5 GB memory and running Linux. The code was compiled with GCC 3.3.2 and the "-O9" optimization switch. We compiled both a 32-bit and a 64-bit version of our bit-parallel algorithm. Fig. 5 shows the results with randomly generated strings of lengths $m = n = 2000$ for alphabet sizes $4, 8, 12, \ldots, 256$. It is evident how KC improves as the average values of $\text{LLCS}(A, B)$ and M become smaller with increasing alphabet size. But both the 32-bit and the 64-bit versions of the bit-parallel algorithm were still faster even at the highest tested alphabet size $\sigma = 256$. The bit-parallel approach truly is practical, especially as its performance is highly independent of the properties of the input strings.

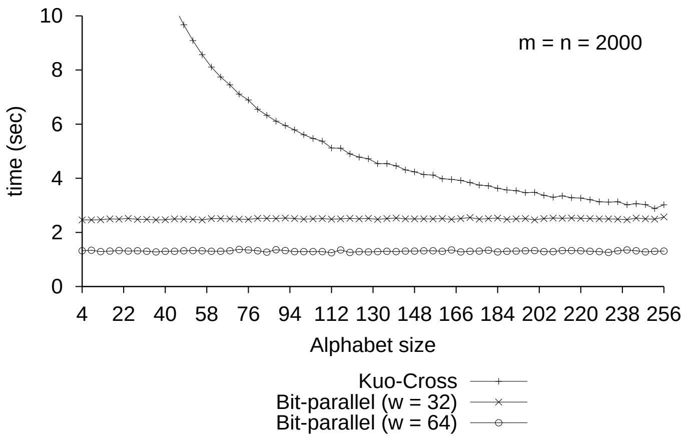

Figure 5: The plot shows the total time for comparing randomly generated pattern pairs 6000 times.

4 Restricting the Computation

There may be situations where one can either make an educated guess about the value of LLCS(A, B), or one is only interested in checking whether LLCS(A, B) is larger than some predetermined threshold. For example, in the case of a 4-letter alphabet such as DNA, the expected value of LLCS(A, B) may be roughly $0.65m$ when $m = n$. Given a threshold t for the value LLCS(A, B), we now propose a method of restricting the bit-parallel computation so that its run time for checking whether LLCS$(A, B) \geq t$ is $O(n\lceil d'/w\rceil)$, where $d' = n + m - 2t$. This leads also to an algorithm with run time $O(n\lceil d/w\rceil)$ for computing LLCS(A, B), where $d = ed_{id}(A, B) = n + m - 2\times$ LLCS(A, B).

Let diagonal x refer to the cells $L[i, j]$ for which $j - i = x$. The method is based on the following simple observation.

Observation 6.
If LLCS$(A, B) \geq t$, then all matches $A_i = B_j$ in L that correspond to matches in LCS(A, B) occur within the diagonals $-(m - t) \ldots n - t$.

If LLCS$(A, B) \geq t$, computing the value LLCS(A, B) does not require filling any cell $L[i, j]$ that is outside the diagonal region described in Observation 6. Thus when we cover L with the length-w bit-vectors, we need to compute only such tiles that overlap with some of the diagonals $-(m - t) \ldots n - t$. Fig. 6 depicts the situation. The diagonal region contains $(n - t) + 1 + (m - t) = m + n + 1 - 2t \leq ed_{id}(A, B) + 1$ diagonals. Therefore we need to tile $O(\lceil ed_{id}(A, B)/w\rceil)$ vectors in each column, and the total run time is $O(\lceil ed_{id}(A, B)/w\rceil n)$.

If the threshold value t is not known beforehand, one can begin by guessing that $t = m$ and checking if the restricted algorithm comes up with a value LLCS$(A, B) \geq t$. If this is not the case, one can keep doubling the number of diagonals (effectively the current guess for $ed_{id}(A, B)$) until the corresponding check determines that LLCS$(A, B) \geq t$. When this process stops after h iterations, the current number of diagonals must be less than $2 \times ed_{id}(A, B)$, since otherwise already a previous check would have ended the process. The last check costs $O(\lceil ed_{id}(A, B)/w\rceil n)$, and since $\sum_{k=0}^{h} 1/2^k < 2$, also the total work is

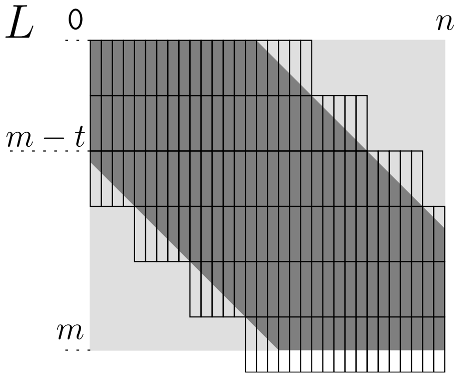

Figure 6: The portion of L that needs to be filled under the threshold t is shaded with dark grey. The figure depicts how only that portion is covered with bit-vectors.

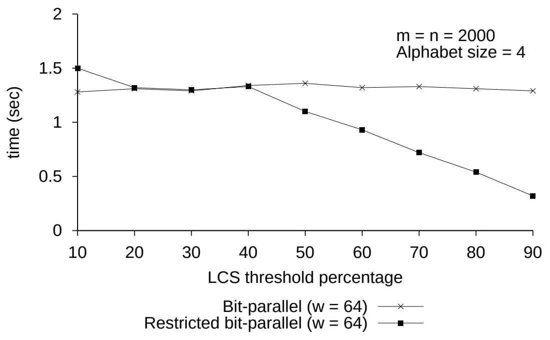

Figure 7: The plot shows the total time for comparing randomly generated pattern pairs 6000 times.

$O(\lceil ed_{id}(A,B)/w\rceil n)$.

We implemented and tested the restriction method that checks whether LLCS(A,B) $\geq t$ when a predetermined t is given. The test included randomly generated pattern pairs with lengths $m = n = 2000$, and the alphabet size was 4. We tested with estimates of t from 10% of m up to 90% of m, in 10% increments. Fig. 7 shows the results. It can be seen that the restriction method is effective for moderately long patterns of (nearly) equal length when the estimate of t is at least 50% of m. Such is the case for example with DNA sequences.

5 Conclusions

Computing the length of the longest common subsequence between two strings is a classic and much studied problem. There has previously appeared two bit-parallel algorithms for solving the problem: AD by Allison and Dix [2], and recently CIPR by Crochemore et al. [4]. In this paper we presented a new more efficient variant of CIPR, and noticed how their original comparison between CIPR and AD was extremely misleading. Then we showed a comparison between our new variant and the algorithm of Kuo and Cross [11] in order to characterize the relative performance of the bit-parallel methods and the best conventional methods. Finally we proposed and evaluated a restriction method for checking whether LLCS$(A, B) \geq t$. This leads also into an $O(\lceil ed_{id}(A, B)/w \rceil n)$ scheme for computing LLCS(A, B), where $ed_{id}(A, B)$ is simple edit distance.

Acknowledgements

We would like to thank Harri Hakonen for generously providing us with an efficient implementation of the algorithm of Kuo and Cross that was used in the survey [3].

References

[1] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, 1976.

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

[3] 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.

[4] 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. Appeared also in proc. AWOCA 2000.

[5] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, UK, 1994.

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

[7] D. S. Hirschberg. An information theoretic lower bound for the longest common subsequence problem. *Communications of the ACM*, 18(6):341–343, 1975.

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

[9] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20:350–353, 1977.

[10] H. Hyyrö. A bit-vector algorithm for computing Levenshtein and Damerau edit distances. *Nordic Journal of Computing*, 10:1–11, 2003.

[11] S. Kuo and G. R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. *ACM SIGIR Forum*, 23(3-4):89–99, 1989.

[12] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. of Computer and System Sciences*, 20:18–31, 1980.

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

[14] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.

[15] C. Rick. A new flexible algorithm for the longest common subsequence problem. In *Proc. 6th Combinatorial Pattern Matching (CPM'95)*, LNCS 937, pages 340–351, 1995.

[16] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

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

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

[19] S. Wu, U. Manber, G. Myers, and W. Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35:317–323, 1990.

[20] E. M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms - Theory and Practice.* Prentice-Hall, 1977.