This is the accepted manuscript of the article, which has been published in String Processing and Information Retrieval: 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26–29, 2017, Proceedings: International Symposium on String Processing and Information Retrieval. Lecture Notes in Computer Science, 10508, 2017, 214-220. ISBN 978-3-319-67427-8. https://doi.org/10.1007/978-3-319-67428-5_18

Mining Bit-Parallel LCS-length Algorithms

Heikki Hyyrö

Faculty of Natural Sciences, University of Tampere, Finland
heikki.hyyro@uta.fi

Abstract. Some of the most efficient algorithms for computing the length of a longest common subsequence (LLCS) between two strings are based on so-called "bit-parallelism". They achieve $O(\lceil m/w \rceil n)$ time, where m and n are the string lengths and w is the computer word size. The first such algorithm was presented by Allison and Dix [3] and performs 6 bit-vector operations per step. The number of operations per step has later been improved to 5 by Crochemore at al. [5] and to 4 by Hyyrö [6]. In this short paper we explore whether further improvement is possible. We find that under fairly reasonable assumptions, the LLCS problem requires at least 4 bit-vector operations per step. As a byproduct we also present five new 4-operation bit-parallel LLCS algorithms.

1 Introduction

Let A and B be input strings of lengths m and n. Finding LLCS(A, B), the length of a longest common subsequence (LCS) between the strings A and B, is a classic and much studied problem in computer science. A fundamental $O(mn)$ dynamic programming solution was given by Wagner and Fischer [9], and this quadratic worst-case complexity cannot be improved by any algorithm that uses individual equal/nonequal comparisons between characters [2]. Furthermore a recent conjecture [1] claims that the LLCS problem requires at least $O(n^{2-\lambda})$ time for two strings of equal length $m = n$, for any choice of a constant $\lambda > 0$.

Numerous further LLCS algorithms have been proposed over the last few decades. Breaking the quadratic complexity bound has proven elusive, but significant practical improvements have been achieved. A comprehensive survey of LLCS algorithms by Bergroth et al. [4] found the algorithms of Kuo and Cross (`KC`) [7], Rick [8] and Wu et al. (`WMMM`) [10] to be the fastest in practice. This survey, however, did not include the already existing so-called "bit-parallel" algorithm of Allison and Dix [3]. The bit-parallel approach has been later found to be very practical. For example Hyyrö [6] reported that his improved bit-parallel algorithm (`Hyy`) dominates over `KC`. In order to explore this further, we performed a comparison between `Hyy`, `KC`, `WMMM` and basic dynamic programming (`DP`)[1]. We tested first with random strings of lengths $m = n = 50$ and then with random strings of lengths $m = n = 2000$. The alphabet size varied from 2 to

[1] The algorithm of Rick, as recommended in [4], was omitted as it was not competitive in our experiments. This was probably due to its high $O(\sigma m)$ preprocessing cost.

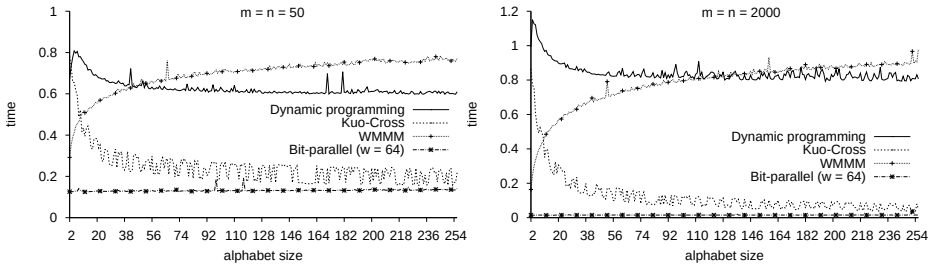

Fig. 1: LLCS algorithm tests with $m = n = 50$ (left) and $m = n = 2000$ (right).

256. The methods were implemented in C and compiled with GNU gcc using the -O3 switch. The test computer had 64-bit Ubuntu Linux 16.04, 16 GB RAM and a 2.3 GHz Intel i7-3651QM CPU. The results are shown in Fig. 1 and seem to confirm the very good performance of the bit-parallel approach.

The first bit-parallel LLCS algorithm by Allison and Dix performs 6 bit-vector operations per each character of B. Later Crochemore et al. [5] improved this to 5 and finally Hyyrö [6] to 4 operations per character, the latter being the most efficient currently known bit-parallel LLCS algorithm. In this paper we explore whether a bit-parallel algorithm that requires only 3 operations exists.

2 Preliminaries

We assume that strings consist of characters from an alphabet Σ with alphabet size σ. S_i denotes the ith character of S and $S_{i..j}$ denotes the substring of S that starts at the ith character and ends at the jth character. A string C is a subsequence of a string A if and only if A can be transformed into C by removing zero or more characters from A. C is a longest common subsequence (LCS) of strings A and B if it is both a subsequence of A and a subsequence of B, and no longer string with this property exists. We denote the unique length of an LCS between A and B by LLCS(A,B). For example LLCS(`"chart"`, `"chatter"`) = 4, and both `"chat"` and `"char"` are corresponding LCSs of length 4.

The fundamental dynamic programming solution for LLCS computation uses Recurrence 1 to fill an $(m+1) \times (n+1)$ dynamic programming matrix L with values $L[i,j] = \text{LLCS}(A_{1..i}, B_{1..j})$. The following Observations 1 and 2 are well-known and easy to derive from Recurrence 1.

Recurrence 1. When $0 \leq i \leq m$ and $0 \leq j \leq n$:

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

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

Observation 2. *The values in a column j of L may be described by recording all rows $i \in 1,\ldots,m$ where $L[i,j] = L[i-1,j]+1$.*

L		c	h	a	t	t	e	r
	0	0	0	0	0	0	0	0
c	0	1	1	1	1	1	1	1
h	0	1	2	2	2	2	2	2
a	0	1	2	3	3	3	3	3
r	0	1	2	3	3	3	3	4
t	0	1	2	3	4	4	4	4

V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7
0	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	0	0	0	0	1
0	0	0	0	1	1	1	0

Fig. 2: The dynamic programming table L and corresponding column vectors V_j.

We use the following notation with bit-vectors: '&' 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 padding at both ends. The ith bit of the bit vector V is $V[i]$ and bit positions grow from right to left. For example the bits values of a bit-vector $V = 1011001$ are $V[1] = V[4] = V[5] = V[7] = 1$ and $V[2] = V[3] = V[6] = 0$.

Bit-parallel algorithms take advantage of the fact that digital computers perform operations on chunks of w bits, where w is the computer word size (in most recent computers $w = 64$). In case of LLCS computation, Observations 1 and 2 permit us to represent the m row values of column j of L by a length-m bit-vector V_j whose ith bit is 1 if and only if $L[i, j] = L[i - 1, j] + 1$. Now the actual value $L[m, j]$ is given by the sum $\sum_{k=1}^{m} V_j[k]$, and V_j fits into $\lceil m/w \rceil$ computer words. Fig. 2 shows an example of a dynamic programming table L and its representation by V_j vectors. Clearly also other bit encodings are possible: we may e.g. define a complemented variant $V'_j = \sim V_j$, where the ith bit of V'_j is 0 if and only if $L[i, j] = L[i - 1, j] + 1$. Note that the overall value LLCS(A, B) $= L[m, n]$ is given by the number of 1-bits in V_n (or the number of 0-bits in V'_n). The column vector V_j (or V'_j) can be computed from the previous column vector V_{j-1} (or V'_{j-1}) by a constant number of bit-vector operations. The computation requires knowledge of all rows i that have a match between the current column character $B[j]$ and the row character $A[i]$. This is facilitated by precomputing for each different character c a length-m match bit-vector $M[c]$ whose ith bit is 1 if and only if $A[i] = c$.

The 6-operation bit-parallel LLCS algorithm of Allison and Dix encodes the columns of L by V_j, and the 5-operation algorithm of Crochemore at al. and the 4-operation algorithm of Hyyrö use the complemented vectors V'_j. These three algorithms, together with the preprocessing of the match vectors $M[c]$, are described in Fig. 3. See the original articles [3, 5, 6] for details about the algorithms' logic. The algorithms run in $O(\lceil m/w \rceil n)$ time, as a bit-vector operation on length-m bit-vectors can be done in $O(\lceil m/w \rceil)$ time.

3 A Lower Bound for the Bit-Vector Operations

Given the progress from 6 to 4 bit-vector operations, a natural question is whether further improvement to 3 operations is possible. In order to make this problem approachable, we make the following four fairly reasonable assumptions:

PreprocessM$(A_{1\ldots m})$
1. **For** $c \in \Sigma$ **Do**
2. $\quad M[c] \leftarrow 0$
3. **For** $i \in 1 \ldots m$ **Do**
4. $\quad M[A_i] \leftarrow M[A_i] \mid (1 << (i-1))$

AD$(A_{1\ldots m}, B_{1\ldots n})$
1. $V_0 \leftarrow 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M[B_j] \mid V_{j-1}$
4. $\quad V_j \leftarrow P \;\&\; ((P - ((V_{j-1} << 1) \mid 1)) \;\hat{}\; P)$
5. Return **countOnebits**(V_n)

CIPR$(A_{1\ldots m}, B_{1\ldots n})$
1. $V'_0 \leftarrow \sim 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad V'_j \leftarrow (V'_{j-1} + (V'_{j-1} \;\&\; M[B_j])) \mid (V'_{j-1} \;\&\; (\sim M[B_j]))$
4. Return **countZerobits**(V'_n)

Hyy$(A_{1\ldots m}, B_{1\ldots n})$
1. $V'_0 \leftarrow \sim 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M[B_j] \;\&\; V'_{j-1}$
4. $\quad V'_j \leftarrow (V'_{j-1} - P) \mid (V'_{j-1} + P)$
5. Return **countZerobits**(V'_n)

Fig. 3: The bit-parallel LLCS algorithms of Allison and Dix (`AD`), Crochemore et al. (`CIPR`) and Hyyrö (`Hyy`). Also preprocessing of the M-vectors is shown.

1. Generality: the algorithm must work with all bit-vector lengths w.
2. Input: a length-m bit-vector representing the increment positions in column $j-1$ and a length-m bit vector representing the matching rows for B_j.
3. One-to-one correspondence between rows and bit-positions: each bit position in the bit-vectors corresponds to a certain row i.
4. Universality: the algorithm uses only commonly available operations.

These assumptions seem fairly reasonable as we in practice can afford to use only 1 bit per row: a scheme that uses k bits per row needs km/w computer words per column, and hence the number of operations should be less than $4/k$ in order to improve on the best current bit-parallel algorithm.

We tackled the problem by enumerating and testing all possible 3-operation bit-parallel algorithms that fulfill the preceding assumptions. This was feasible as the number of operations is so small. The enumeration proceeded roughly as follows and more or less corresponds to a 10-level deep nested for/while-loop:

1. Enumerate over all ways to select operands for 3 operations. Below the values C_1 and C_2 are constants (more than two would be redundant) and R_i refers to the result of the ith operation. Each operation selects operands as follows:
   - The 1st operation selects 2 operands from $\{V_{j-1}, C_1, C_2, M[c]\}$.
   - The 2nd operation selects 2 operands from $\{V_{j-1}, C_1, C_2, M[c], R_1\}$.
   - The 3rd operation selects 2 operands from $\{V_{j-1}, C_1, C_2, M[c], R_1, R_2\}$.
2. Under step 1, iterate over all permutations of $1, \ldots, w$, where each defines one possible mapping from bits to rows in the column and match vectors: which bit position corresponds to which row.
3. Under step 2, iterate over all 2^w length-w bit vectors, where each defines one possible set of bit roles for the column vectors: the ith bit defines whether an increment $L[i,j] = L[i-1,j] + 1$ is recorded as a 1 or 0 bit.
4. Under step 3, iterate over all 2^w length-w bit vectors, where each defines one possible set of bit roles for the match vectors: the ith bit defines whether a match $A[i] = c$ is recorded into $M[c]$ as a 1 or 0 bit.

5. Under step 4, iterate over all possible constant values (ie. all 2^w possible values for C_1 and C_2, independent of each other).
6. Under step 5, iterate over all possible ways to select 3 operations out of the set of permitted operations.
   (a) For each selection of 3 operations, check whether the formula is correct by checking if the formula produces correct result (into R_3) with all inputs (combinations of previous column vector and current match vector).
      i. The result is verified by comparing R_3 with the result of basic dynamic programming (permuting the row positions and inverting bit values where necessary).

Although this exceeds the typical assumptions of bit-parallel algorithms, the search allowed each operation to be (1) any of the 16 possible binary logical operations, (2) an arithmetic operator +, -, * or /, or (3) a left or right shift that uses either 1-bits or the left/rightmost bit for padding (allowed shift lengths are $1, \ldots, w-1$). Note that 0-padding shifts are expressed by multiplication and division (with one of the constants, such as C_1, specifying the multiplier).

We ran the exhaustive search using a fixed small length $w = 4$. The computation took roughly 50 minutes and was unable to find a working 3-operation formula. This provides support for a claim that the existing 4-operation bit-parallel LLCS algorithm is optimal within the constraints described before. In order to gain confidence in the correctness of the procedure, we modified the implementation to do an exhaustive search over all 4-operation combinations. As this would have otherwise taken too much time, the 4-operation test was restricted to consider only two linear mappings from bits to rows: the basic order, where the ith bit corresponds to row i, and a reverse order, where the ith bit corresponds to the row $m - i + 1$. The other parts of the implementations were left intact. The 4-operation run took roughly 1 hour and found dozens of correct 4-operation algorithms. Many of these used non-standard logical bit-vector operations, but a total of 6 essentially different algorithms used only the universally supported C-style arithmetic and logical operations. One of these was the algorithm `Hyy` of Hyyrö [6] and the rest were new. We note that the search also found the `CIPR` algorithm of Crochemore et al. [5], as that algorithm makes only 4 operations when the non-standard logical "X and not Y"-operation counts as one operation. Finding these formulas provides some further confidence that the search procedure works correctly.

Fig. 4 shows the five new 4-operation bit-parallel LLCS algorithms. All use the natural top-down mapping between bits and rows, where the ith bit corresponds to row i, and uniform bit roles (the meaning of 0 and 1 bits is the same in all rows of the same vector). Some use complemented match vectors: we define $M'[c]$ as a match vector whose ith bit is 0 if and only if $A[i] = c$. The names `11a`, `11b`, `10`, `00a` and `00b` reflect the bit roles. The first letter is 1 if the algorithm uses M and 0 if it uses the complemented M' match vectors, respectively. The second letter is 1 if the algorithm uses the V_j and 0 if it uses the complemented V'_j column vectors. According to our preliminary experiments, all these 4-operation variants have virtually identical practical performance.

11a$(A_{1\ldots m}, B_{1\ldots n})$
1. $V_0 \leftarrow 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M[B_j] \mid V_{j-1}$
4. $\quad V_j \leftarrow P \;\&\; (V_{j-1} + V_{j-1} - P)$
5. Return **countOnebits**(V_n)

11b$(A_{1\ldots m}, B_{1\ldots n})$
1. $V_0 \leftarrow 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M[B_j] \mid V_{j-1}$
4. $\quad V_j \leftarrow P \;\&\; (V_{j-1} - (V_{j-1} \;\hat{}\; P))$
5. Return **countOnebits**(V_n)

10$(A_{1\ldots m}, B_{1\ldots n})$
1. $V'_0 \leftarrow \sim 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M[B_j] \;\&\; V'_{j-1}$
4. $\quad V'_j \leftarrow (V'_{j-1} \;\hat{}\; P) \mid (V'_{j-1} + P)$
5. Return **countZerobits**(V'_n)

PreprocessM'$(A_{1\ldots m})$
1. **For** $c \in \Sigma$ **Do**
2. $\quad M'[c] \leftarrow \sim 0$
3. **For** $i \in 1 \ldots m$ **Do**
4. $\quad M'[A_i] \leftarrow M'[A_i] \;\&\; \sim(1 << (i-1))$

00a$(A_{1\ldots m}, B_{1\ldots n})$
1. $V'_0 \leftarrow \sim 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M'[B_j] \;\&\; V'_{j-1}$
4. $\quad V'_j \leftarrow P \mid (V'_{j-1} + V'_{j-1} - P)$
5. Return **countZerobits**(V'_n)

00b$(A_{1\ldots m}, B_{1\ldots n})$
1. $V'_0 \leftarrow \sim 0$
2. **For** $j \in 1 \ldots n$ **Do**
3. $\quad P \leftarrow M'[B_j] \;\&\; V'_{j-1}$
4. $\quad V'_j \leftarrow P \mid (V'_{j-1} + (V'_{j-1} \;\hat{}\; P))$
5. Return **countZerobits**(V'_n)

Fig. 4: The five new 4-operation bit-parallel LLCS algorithms found by our exhaustive search. Also preprocessing the complemented M'-vectors is shown.

References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. In: Proc. 56th Annual IEEE Symposium on Foundations of Computer Science. pp. 59–78 (2015)
2. Aho, A.V., Hirschberg, D.S., Ullman, J.D.: Bounds on the complexity of the longest common subsequence problem. Journal of the ACM 23(1), 1–12 (1976)
3. Allison, L., Dix, T.L.: A bit-string longest common subsequence algorithm. Information Processing Letters 23, 305–310 (1986)
4. Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proc. 7th International Symposium on String Processing and Information Retrieval. pp. 39–48 (2000)
5. Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J., Reid, J.F.: A fast and practical bit-vector algorithm for the longest common subsequence problem. Information Processing Letters 80, 279–285 (2001)
6. Hyyrö, H.: Bit-parallel LCS-length computation revisited. In: Proc. 15th Australasian Workshop on Combinatorial Algorithms. pp. 16–27 (2004)
7. Kuo, S., Cross, G.R.: An improved algorithm to find the length of the longest common subsequence of two strings. ACM SIGIR Forum 23(3-4), 89–99 (1989)
8. Rick, C.: A new flexible algorithm for the longest common subsequence problem. In: Proc. 6th Annual Symposium on Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 937, pp. 340–351 (1995)
9. Wagner, R.A., Fischer, M.J.: The string-to-string correction problem. Journal of the ACM 21(1), 168–173 (1974)
10. Wu, S., Manber, U., Myers, G., Miller, W.: An $O(NP)$ sequence comparison algorithm. Information Processing Letters 35, 317–323 (1990)