Longest Common Subsequence (LCS) Retrieval in Linear Space and Single-Pass Quadratic Time for Some Special Cases of the Input

Arian J. Jacildo
Institute of Computer Science
University of the Philippines at Los Baños
4031 Laguna, Philippines
+63(049)536-2302
ajjacildo@uplb.edu.ph

Eliezer A. Albacea
Institute of Computer Science
University of the Philippines at Los Baños
4031 Laguna, Philippines
+63(049)536-2302
eaalbacea@uplb.edu.ph

ABSTRACT
Alignment of DNA sequences is one of the routines in bioinformatics. The longest common subsequence (LCS) retrieval problem can be treated as a simple version of the sequence alignment problem. A simple dynamic programming (DP) algorithm can solve the LCS retrieval of two sequences *x* and *y*, with lengths *m* and *n*, respectively, in O(*mn*) time and in O(*mn*) space. Improvements to the simple DP algorithm are either on the running time or on the memory space complexity. In 1975, Hirschberg developed an elegant divide-and-conquer (D&C) algorithm (HLCS) for LCS retrieval in O(*m*+*n*) or linear space and in O(*mn*), two-pass quadratic time. The algorithm presented in this paper, referred to as the modified Hirschberg's LCS algorithm (MHLCS), can perform LCS retrieval, still, in linear space, but in a single-pass quadratic time on some special cases of the input.

Keywords
Linear space longest common subsequence, sequence alignment.

1. INTRODUCTION
A subsequence is defined as a sequence that can be obtained by deleting zero or more symbols from a given sequence. Given input sequences *x* and *y* with lengths *m* and *n*, respectively, the pair-wise LCS retrieval problem is defined as the problem of finding a third sequence *w*, such that *w* is a subsequence of *x* and *y* and the length of *w* is maximal. The lower-bound analysis of the pair-wise version of the problem is O(*mn*) for an unbounded alphabet [1]. The multiple input sequences version of the problem is NP-hard [2]. The MHLCS algorithm we present on this paper will focus on the pair-wise version of the problem.

The LCS problem is related to the sequence alignment problem. Given input sequences *x* and *y*, the basic sequence alignment problem is characterized as the problem of inserting a minimum number of gaps in *x* and *y* so as to maximize the number of matches and minimize the number of mismatches. The total score of an alignment is based on the sum of matches, gaps and mismatches. A simple scoring using negative infinity score mismatches is essentially equivalent to LCS [3].

The LCS and sequence alignment problems are also considered as special cases of the string editing problem [4]. Given input sequences *x* and *y*, the string editing problem consists of transforming *x* into *y* by performing a series of weighted edit operations on *x* of overall minimum cost. An edit operation can be a deletion, insertion or substitution of a symbol in *x*.

A simple dynamic programming (DP) algorithm can solve the LCS retrieval problem in O(*mn*) time and O(*mn*) space [5]. In 1975, Hirschberg, incorporated a divide-and-conquer (D&C) algorithm to the DP algorithm to perform LCS retrieval in O(*mn*) or specifically in two-pass quadratic time and in O(*m*+*n*) or linear space [6]. Since then, his algorithm has been the basis for most linear space solutions to LCS, sequence alignment and other related problems.

Crochemore et. al. used bit vector representation in improving HLCS [7]. Driga et. al. developed the FastLSA[9] which improved HLCS by applying double split, horizontal and vertical split, as compared to the single horizontal split of HLCS. Guo and Hwang gave a non-DP algorithm using primal-dual algorithm for LCS, which runs in linear space for constant alphabet size or in O(*nb*) space for *b* = alphabet size [8].

The MHLCS algorithm presented in this paper follows a different approach in improving HLCS by increasing the constant factor on the space complexity to reduce the constant factor on the time complexity. It reduces the two-pass quadratic time of HLCS to a single pass quadratic time on some special cases of the input. It makes use of the additional space in the form of two additional arrays to help identify the special cases.

2. MHLCS ALGORITHM
Given input sequences $x = x[1],x[2]\ldots x[m]$ and $y = y[1],y[2]\ldots y[n]$ where m and n are the lengths of *x* and *y*, respectively and $m \leq n$, the LCS retrieval of *x* and *y* is using MHLCS is summarized in the following steps.

Step 1. Let *p* = LCS length of *x* and *y*. Compute for *p* in linear space and in a single-pass quadratic time using DP [5]. In addition, while computing for *p* via DP, we also generate array structures L and AUX.

Step 2. Generate a candidate LCS *w* with length equal to *p*, such that *w* is a subsequence of *y* and that each symbol in *w* starting from *w*[1] to *w*[*p*] is matched with at least one *x*[*i*]. We will show that this can be done in linear space and linear time using the information stored in L.

Step 3. Check if *w* is also a sub-sequence of *x*. This process will also be done in linear space and linear time, using the additional information stored in AUX.

If *w* is also a sub-sequence of *x* then, we already have solved the LCS retrieval in linear space and single-pass quadratic time. If *w* is not a sub-sequence of *x*, then we extract a partial LCS from *w*.

Step 4. Incorporate Steps 1 to 3 to HLCS and use *w* to reduce the number of recursion in HLCS.

The following sub-sections will discuss the details of each step.

2.1 Step 1 – solve for *p*, L and AUX via DP

Solving for p = LCS length of *x* and y with lengths *m* and *n*, respectively, can be characterized by the recurrence in Figure 1, which means that $p = c(m,n)$.

$$c(m,n) = \begin{cases} 0 & m = 0 \text{ or } n = 0 \\ c(m-1,n-1)+1 & x[i] = y[j] \\ \max(c(m-1,n),c(m,n-1)) & otherwise \end{cases}$$

Figure 1. LCS length recurrence relation.

Solving the recurrence can be done via DP using a matrix of size *m* by *n*. The computation via DP will update the contents of a matrix one row at a time. Hence, it is possible to do it using only two rows instead of a full matrix. These two rows can be reused alternately to store the previous and new row values.

```
01  for i=0 to m
02  for j=0 to n
03    if i==0 or j==0
04      c[i mod 2][j]=0
05    else if x[i]==y[j]
06      if c[(i-1) mod 2][j-1]+1 > c[i mod 2][j]
07        AUX[j]=i
08      c[i mod 2][j]=c[(i-1) mod 2][j-1]+1;
09    else c[i mod 2][j]=
10      max(c[(i-1) mod 2][j],c[i mod 2][j-1])
11    Let L be a copy of the last row c[m mod 2]
```

Figure 2. Pseudocode of a linear space DP algorithm for LCS length computation.

Figure 2 shows a pseudocode of a linear space version of the LCS length computation, which also includes the computation of the array structures L and AUX (lines 06, 07, 11). All row access on matrix `c` are applied with a `mod 2` operation keeping row access limited only to the two rows `c[0]` and `c[1]`.

Figure 3 illustrates the results of computing for *p* using the pseudocode in Figure 2 when *x*=ADCB and *y*=ADABCA. The boxed values above array L are the final values of the two rows `c[0]` and `c[1]`, the unboxed values represent the values of earlier computations. The value of *p* is stored in the last column of the last row which is encircled value: `c[m mod 2][n] = c[4 mod 2][6]=c[0][6]=3`.

In line 11 of Figure 2 and as shown in Figure 3, L is simply a copy of the last row generated in the LCS length computation.

In lines 06 and 09 of Figure 2, AUX[*j*] is updated with the value of a row index *i* only when a match found between x[*i*] and y[*j*], and the value that will be assigned to c[i *mod* 2][*j*] is the highest value so far in column *j*. In Figure 3, the shaded values represent the cases when AUX[*j*] is updated with row index *i*.

		j	0	1	2	3	4	5	6
row	i		y[j]	A	D	A	B	C	A
c[0]	0	x[i]	0	0	0	0	0	0	0
c[1]	1	A	0	1	1	1	1	1	1
c[0]	2	D	0	1	2	2	2	2	2
c[1]	3	C	0	1	2	2	2	3	3
c[0]	4	B	0	1	2	2	3	3	(3)
L			0	1	2	2	3	3	3
AUX				1	2	1	4	3	1

Figure 3. Results of the linear space LCS length computation.

2.2 Step 2 – solve for candidate LCS *w*

We use L, which is defined in Section 2.1 as the last row generated in computing for *p*, to generate candidate LCS *w,* where *w* is a subsequence of *y*, the length of *w* = *p*, and each symbol in *w* starting from *w*[1] to *w*[*p*] is matched to at least one *x*[*i*].

Theorem 1. Candidate LCS *w* can be generated as a subsequence of *y*, where *w*[*k*] = y[*f*(*k*)] where *k* = 1 to *p* and each symbol *w*[k] is matched to at least one *x*[i], in linear time.

Proof: We can prove Theorem 1, using the succeeding lemmas. The first part of the theorem can be proven by Lemma 1 and the last part by Lemma 2.□

Lemma 1. L is sorted in ascending order from left to right containing all values 0 to *p*.

Proof: The assignment statements in lines 08 and 09 in Figure 2, ensures that at the end of the computation of *p*, L is sorted in ascending order from left to right containing all values 0 to *p*. □

With Lemma 1, we can get the index of the leftmost occurrence of each number *k*, as a function of *k,* for each *k* = 1 to *p*, *f*(*k*) = *j* such that L[*j*]=*k* and *j* is minimal.

Lemma 2. Each y[*f*(*k*)] matches at least one *x*[*i*].

Proof: If we assume that Lemma 2 is false then it means that y[*f(k)*] did not match with any *x*[*i*].

Case 1: L[*f(k)*] may have taken its value from the cell to its left, this contradicts the fact that L[*f(k)*] is a leftmost occurrence in L.

Case 2: L[*f(k)*] may have taken its value from a match above it, this contradicts the assumption that y[*f(k)*] did not match any *x*[*i*].

There are only two possible cases, both of which result in a contradiction, hence the claim. □

2.3 Step 3 – generate partial LCS from *w*

In the computation of p in Section 2.1 AUX[j] is defined to keep the row index i of a match encountered when updating v = c[i *mod* 2][j] and v is greater than or equal to any value from c[i mod 2][j] up to c[1 *mod* 2][j]. Each value L[j] is a maximum value in column j, hence, x[AUX[$f(k)$]] matches w[k] for k = 1 to p. So, in order to check if w is also a subsequence of x, we only need to check if the values AUX[(f(k)], for k = 1 to p, is monotonically increasing. This is how Hirschberg defined a subsequence in his 1975 paper [6]. If w is a subsequence of x then w is a valid LCS of x and y, else we generate a partial LCS based on w[k] where k is part of the monotonically decreasing values of AUX[(f(k)] for k = p down to 1.

2.4 Step 4 – use *w* to reduce HLCS recursions

To discuss how we incorporate Steps 1 to 3 of MHLCS in solving the sub-problems of HLCS, we discuss first the concepts of HLCS problem partitioning in section 2.4.1, and the modified partitioning in 2.4.2.

2.4.1 *HLCS Partitioning*

Figure 4 shows the conceptual diagram of HLCS partitioning. The process of HLCS partitioning starts by bisecting x equally to $x1$ and $x2$. Then, L1 is generated as the last row in computing the LCS length of x1 and y, and similarly L2 is generated as the last row in computing the LCS length of the reversed x2 and reversed y. Pivot column j is then computed as the maximum value of (L1[j] + L2[n–j]) where j goes from 0 to n. Pivot column j is then used to divide the problem into two smaller sub-problems H1 and H2. The sum of the areas representing the two sub-problems H1 and H2 is half the area of the original problem regardless of the position of pivot column j. Hence, the total running time of HLCS is given by T(a)=T(a/2) + a where a=mn. Following the recurrence T(a) = a + a/2 + a/4 + a/8 +… = 2a = 2mn = O(mn). The space complexity of HLCS is dominated by the size of arrays L1 and L2 used in searching for the pivot column j in the first partitioning of the original problem. Smaller portions of these arrays are reused in solving the sub-problems. Hence, the total space needed by HLCS is only linear.

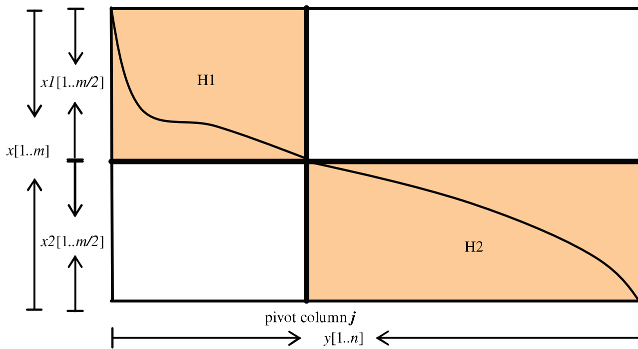


Figure 4. Conceptual view of HLCS problem partitioning.

2.4.2 *MHLCS Partitioning*

We can generate the auxiliary space needed by MHLCS at the same time that HLCS generates the last rows generated in computing pivot column j. We can reduce the sizes of the sub-problems based on the partial LCS that can be solved in each sub-problem.

Figure 5 illustrates the partitioning of MHLCS as compared to the partitioning of HLCS shown in Figure 4. We define a function $mhlcs(x,m,y,n)$ as a function that retrieves the LCS of two input sequences x and y with lengths m and n respectively, $m \leq n$. $mhlcs(x,m,y,n)$ = MH1 + $temp1$+ $temp2$ + MH2 is the concatenation of the parts of the LCS where $temp1$ and $temp2$ are the partial LCS generated while MH1 and MH2 are the reduced versions of the sub-problems. The dimension parameters: $rowpivot1$, $colpivot1$, $colpivot2$ and $rowpivot2$ are determined based on the length of the partial LCS $temp1$ and $temp2$. The area of MH1 and MH2 are determined by $colpivot1$*$rowpivot1$ and (n-$colpivot2$)*(m/2-rowpivot2), respectively.

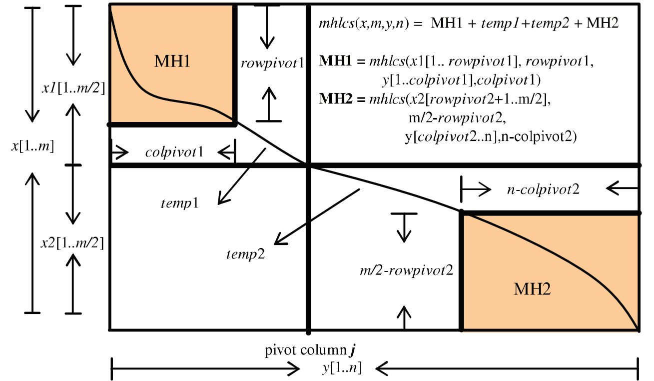


Figure 5. Conceptual view of MHLCS problem partitioning.

3. SPECIAL CASES OF MHLCS

A special case is defined as the part of a sub-problem that MHLCS can solve using the candidate LCS generated for the sub-problem.

Figure 6 shows an example when an entire sub-problem falls as a special case for MHLCS. It illustrates the computations made after a sub-problem is identified and shows the computations made for L, AUX and the partial LCS. The shaded cells in L, are the leftmost occurrences of numbers 1, 2 and 3. The indices of these cells 1, 2, and 4 are used to initialize the candidate LCS w=y[1],y[2],y[4]=ADB. Since the corresponding row indices stored in AUX are monotonically increasing: AUX[1]=1, AUX[2]=2, AUX[4]=3, then w=ADB is a valid LCS for this sub-problem.

		j	0	1	2	3	4	5	6
row	i		y[j]	A	D	A	B	C	A
c[0]	0	x[i]	0	0	0	0	0	0	0
C[1]	1	A	0	1	1	1	1	1	1
C[0]	2	D	0	1	2	2	2	2	2
C[1]	3	C	0	1	2	2	2	3	3
C[0]	4	B	0	1	2	2	3	3	3
L			0	1	2	2	3	3	3
AUX				1	2	1	4	3	1

Figure 6. Example of a full special case.

On the other hand, Figure 7, shows an example when only a part of a sub-problem falls as a special case. Figure 7 uses a similar setup as in Figure 6 but uses a different input example. The shaded cells in L, are the leftmost occurrences of numbers 1, 2 and 3. The indices of these cells 1, 4 and 5 are used to initialize the candidate LCS *w*=*y*[1],*y*[4],*y*[5]=BDC. And since the values in the corresponding auxiliary array AUX[1]=4, AUX[4]=2, AUX[5]=3 are not monotonically increasing, then the partial LCS will be extracted based on the monotonically decreasing value in the auxiliary space starting from the right given by AUX[5]=3 and AUX[4]=2. The corresponding partial LCS that can be extracted is given by *y*[4],*y*[5]=DC. The remaining part will serve as the reduced sub-problem part which will be solved recursively.

		j	0	1	2	3	4	5	6
row	i		y[j]	B	A	A	D	C	A
c[0]	0	x[i]	0	0	0	0	0	0	0
c[1]	1	A	0	0	1	1	1	1	1
c[0]	2	D	0	0	1	1	2	2	2
c[1]	3	C	0	0	1	1	2	3	3
c[0]	4	B	0	1	1	1	2	3	3
L			0	1	1	1	2	3	3
AUX				4	1	1	2	3	1

Figure 7. Example of a partial special case.

4. ANALYSIS OF MHLCS

In this section, we analytically prove the correctness of MHLCS and its improvements.

Theorem 2. MHLCS is correct.

Proof: The recursive part of MHLCS rides on the correctness of the HLCS. The partial LCS extracted as *w*[*k*] where *k* is part of the monotonically decreasing values of AUX[(*f*(*k*)] for $k = p$ down to 1, ensures that the partial LCS extracted is valid.

Theorem 3. MHLCS will maintain the quadratic time and linear-space complexity of HLCS on the worst-case.

Proof: When no special cases are found, MHLCS will be reduced to HLCS. MHLCS will incur an overhead cost of O(*t*) where *t*=number of matches and *t* is at most *mn*, hence, MHLCS will maintain a quadratic time complexity. MHLCS only added linear space auxiliary structures to HLCS, hence, also maintaining the linear space complexity. □

In practice, the case when there are *t*=*mn* happens only when the two input sequences are taken from an alphabet with size *b*=1. This, however, is not a worst-case scenario at all for MHLCS since it is in fact, one of the full special cases for MHLCS. When the alphabet size $b > 1$ and assuming the symbols are equally likely to occur in the input sequences, then the total matches is reduced from *mn* to (*mn*)/*b*. So, when the alphabet size $b > 1$, the overhead for MHLCS will be further reduced. For practical applications, it is unlikely that MHLCS will incur an overhead of *mn* operations.

Theorem 4. MHLCS will speed up HLCS from two-pass quadratic time (2*O(*mn*)) to one-pass quadratic time ((1*O(*mn*)) for some special cases of the input.

Proof: When the candidate LCS *w* generated solves entirely the first two sub-problems and there is no significant overhead due to the number of matches, then, MHLCS will run in a single-pass quadratic time ((1*O(*mn*)) whereas HLCS will still have to solve sub-problems recursively and eventually run in two-pass quadratic time (2*O(*mn*)).□

5. EMPIRICAL RESULTS

5.1 Input Generation

There are two parameters used in generating the inputs: first is the length of the input sequences, N = 100, 000 and 50, 000 and second is the perturbation rate PR = 0%, 1%, 5%, 10%, 25%, 50%, 75% and 100%. The first input sequence is always generated by randomly selecting one character at a time from an alphabet with size twenty. The second input sequence is generated as a perturbed version of the first input sequence. The perturbation of the second input sequence is performed using the following steps:

Step 1. Make a copy of the first input sequence and store it as the second input sequence.

Step 2. Randomly select (PR*N)/50 perturbation points out of a total of N indices of the second input sequence e.g. at PR=5% and N=100, 000 so that there will be 5000/50 = 100 perturbation points.

Step 3. In each perturbation point, randomly select from the two possible actions: either insert 50 characters randomly selected from the same alphabet or delete 50 characters from the second input sequence.

Step 4. To keep similar input sequences lengths, either append new characters, randomly selected from the same alphabet, at the end of the second input sequence or truncate it.

The idea of the perturbation rate is to generate a second input sequence that will maintain (1-PR)*N characters from the first

sequence. For example at 0% perturbation rate the second input sequence generated is an exact copy of the first input sequence and at 100% perturbation the second input sequence is generated randomly just like the first input. Hence, the smaller the perturbation rate the higher the similarity of the two input sequences.

There are also five random instances generated for each setup, hence, there is a total of 80 pairs of input sequences (2 input lengths * 5 instances * 8 perturbation rates).

5.2 Implementation Setup

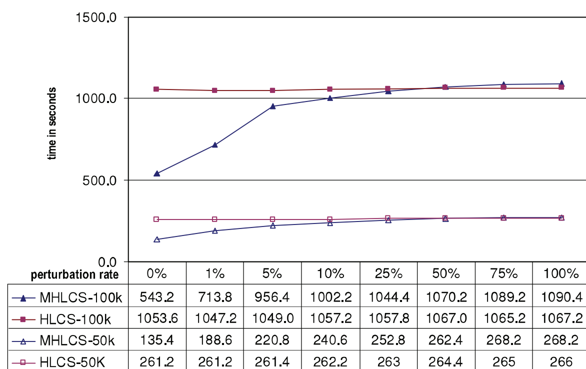
The implementations of MHLCS and HLCS were done using C language and were run on an Athlon 900MHz PC with 256MB DDR3 memory. The two implementations were run sequentially on a dedicated PC to process the 80 different pairs of input sequences.

5.3 Comparison of Actual Results

Figure 8, shows the graph of actual running time in seconds of MHLCS and HLCS based on varying perturbation rates and different lengths of inputs. There are four graphs shown namely: MHLCS-100k, HLCS-100k, MHLCS-50k and HLCS-50k. The names correspond to the algorithm and the input lengths used.

The results show that at lower perturbation rates, MHLCS performs faster than HLCS with maximum improvement of almost 50% at exactly similar input sequences. At lower perturbation rates or highly similar input sequences MHLCS can exploit the occurrence of special cases and can reduce larger number of recursions. Hence, confirming the claim that MHLCS can perform a single-pass quadratic time algorithm on some special cases.

The results show that at higher perturbation rates, up to 100%, MHLCS performs similarly as fast as HLCS with minimal overhead. It means that MHLCS was able to compensate for its overhead in checking for special cases with the reduction in the recursions.



perturbation rate	0%	1%	5%	10%	25%	50%	75%	100%
MHLCS-100k	543.2	713.8	956.4	1002.2	1044.4	1070.2	1089.2	1090.4
HLCS-100k	1053.6	1047.2	1049.0	1057.2	1057.8	1067.0	1065.2	1067.2
MHLCS-50k	135.4	188.6	220.8	240.6	252.8	262.4	268.2	268.2
HLCS-50K	261.2	261.2	261.4	262.2	263	264.4	265	266

Figure 8. Graphs of actual running times in seconds

It must also be noted in the same implementation setup, allocating a quadratic space matrix based on either of the two input lengths will result to an insufficient memory error.

6. CONCLUSION

In conclusion, MHLCS - the algorithm presented in this study, has improved the constant factor in the running time performance of HLCS - Hirschberg's algorithm, from two-pass quadratic time to single pass-quadratic time on some special cases of input. Empirical results verified that MHLCS can indeed reduce the actual running time of HLCS by 50% in highly similar input sequences and maintain similar actual running times with minimal overhead in highly random input sequences.

We consider applications such as comparison of different versions of similar documents and also bioinformatics applications where the inputs are from a given sequence and its slightly mutated version, as possible real-world applications for MHLCS.

7. ACKNOWLEDGEMENTS

The authors thank the Institute of Computer Science and the College of Arts and Sciences of the University of the Philippines at Los Baños for their financial support through ICS-GF #2326103 and CAS-TF #8217300, respectively.

8. REFERENCES

[1] J. D. Ullman, A. V. Aho, and D. S. Hirschberg, "Bounds on the complexity of the longest common subsequence problem," J. ACM, vol. 23, no. 1, pp 1-12, 1976.

[2] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. New York, NY, USA: W. H. Freeman & Co., 1990.

[3] J. M. Samaniego, "Alignment problems in bioinformatics: A guided tour," in SMACS 2004: 2nd Symposium on Mathematical Aspects of Computer Science Preproceedings. Baguio City, Philippines: Computing Science Society of the Philippines, 2004, pp. 8-15.

[4] A. Apostolico, "String editing and longest common subsequences," pp. 361-398, 1997.

[5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, Introduction to Algorithms 2nd ed. McGraw-Hill Higher Education, 2001.

[6] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," Commun. ACM, vol. 18, no. 6, pp 341-343, 1975.

[7] M. Crochemore, C. S. Ilipoulos, and Y. J. Pinzon, "Speeding-up hirschberg and hunt-szymanski lcs algorithms," Fundam. Inf., vol. 56, no. 1, pp. 89-103, 2003.

[8] J. Y. Guo and F. K. Hwang, "An almost-linear time and linear space algorithm for the longest common subsequence problem," Inf. Process. Lett., vol. 94, no. 3 pp. 131-135, 2005.

[9] A. Driga, P. Lu, J. Schaeffer, D. Szafron, K. Charter, and I. Parsons, "Fastlsa: A fast, linear-space, parallel and sequential algorithm for sequence alignment," Algorithmica, vol. 45, no. 3, pp 337-375, 2006.