

Parameterized Computation of LCS for Two Sequences

Yuan Lin

Dept. of Computer Science,
Arkansas State University,
State University,
AR, 72467 USA
ylin@csm.astate.edu

Jeff Jenness

Dept. of Computer Science,
Arkansas State University,
State University,
AR, 72467 USA
jeffj@csm.astate.edu

Xiuzhen Huang

Dept. of Computer Science,
Arkansas State University,
State University,
AR, 72467 USA
xzhuang@csm.astate.edu

Abstract - *The problem of finding the longest common subsequence is a well-known optimization problem because of its applications, especially in bioinformatics. In this paper, by applying the techniques developed in parameterized computation, an efficient approach for the problem of finding the longest common subsequence of two sequences is presented. The parameterized approach is compared with the well-known dynamic programming approach for the problem. The parameterized approach is much more efficient, as is shown by the experimental results.*

Keywords: Parameterized computation, Longest Common Subsequence

1. Introduction

The longest common subsequence (LCS) problem is a well-known optimization problem because of its applications. The fixed alphabet version of the problem is of particular interest considering the importance of sequence comparison (e.g. multiple sequence alignment) in the fixed size alphabet world of DNA and protein sequences. (Note that in computational biology, DNA sequences are in a four-letter alphabet, and protein sequences are in a twenty-letter alphabet).

A string s is a subsequence of a string s' if s can be obtained from s' by deleting some characters in s' . For example, "ac" is a subsequence of "atcgt". Given a set of strings over an alphabet Σ , the longest common subsequence problem is to find a common subsequence that has maximum length. The alphabet Σ may be of fixed size

or of unbounded size. For example, if s_1 and s_2 are two strings and s is the longest common subsequence of s_1 and s_2 , the elements in s appears in each of s_1 and s_2 in the same order but not necessarily consecutively. By finding s , we can determine how similar of s_1 and s_2 . The longer s we can find, the more similar s_1 and s_2 are. For example, assume $s_1 = \text{"actgat"}$ and $s_2 = \text{"ctacgaga"}$. Then "cga" is a common subsequence of s_1 and s_2 , but it is not the longest. "acga" is longer than "cga" and is the longest one. So $s = \text{"acga"}$.

In this paper we focus on the problem of finding the longest common subsequence of two sequences by applying the parameterized computation techniques.

2. A brief review on parameterized computation

According to the theory of NP-completeness, many problems that have important real-world applications in life science are NP-hard. This excludes the possibility of solving them in polynomial time unless $P=NP$. For example, the problems of cleaning up data, multiple sequence alignment, closest string and maximum common substructure, are all famous NP-hard problems in computational biology [4, 8, 12, 13]. A number of approaches have been proposed in dealing with these NP-hard problems. For example, the highly acclaimed approximation approach [1] tries to come up with a *good* enough solution in polynomial time instead

of an optimal solution for an NP-hard optimization problem [6, 7, 10, 11].

The theory of parameterized computation [8] is a newly developed approach introduced to address NP-hard problems with *small* parameters. It tries to give exact algorithms for an NP-hard problem when its natural parameter is small (even if the problem size is big). A parameterized problem Q is a decision problem consisting of instances of the form (x, k) , where the integer $k \geq 0$ is called the parameter. The parameterized problem Q is fixed-parameter tractable [8] if it can be solved in time $f(k)|x|^{O(1)}$, where f is a recursive function.

For a problem in the class FPT, researchers try to come up with more efficient parameterized algorithms. There are many effective techniques for parameterized algorithm designing, such as the methods of bounded search tree and reduction to a problem kernel. For example, the Vertex Cover problem, a well-known NP-hard problem, is fixed-parameter tractable (in the class FPT).

Vertex Cover problem: given a graph G and an integer k , determine if G has a vertex cover C of k vertices, i.e., a subset C of k vertices in G such that every edge in G has at least one end in C . Here the parameter is k .

Given a graph of n vertices, there is a parameterized algorithm that can solve the Vertex Cover problem in time $O(kn + 1.286^k)$ [5].

Accompanying the work on designing efficient and practical parameterized algorithms, a theory of parameter intractability is developed. In parameterized complexity, to classify fixed-parameter intractable problems, a hierarchy, the W -hierarchy $W[t]$, $t \geq 0$, where $W[t] \subseteq W[t+1]$ for all $t \geq 0$, has been introduced, in which the 0-th level $W[0]$ is the class FPT. The hardness and completeness have been defined for each level $W[i]$ of the W -hierarchy for $i \geq 1$, and a large number of

$W[i]$ -hard parameterized problems have been identified [8]. For example, the Clique problem is $W[1]$ -hard.

Clique problem: given a graph G and an integer k , determine if G has a clique C of k vertices, i.e., a subset C of k vertices in G such that there is an edge in G between any two of these k vertices, i.e., the k vertices induce a complete subgraph of G . Here the parameter is k .

The Clique problem is also a well-known NP-hard problem [9]. The Clique problem can be solved in time $O(n^k)$, based on the enumeration of all the vertex subsets of size k for a given graph with n vertices. Now it has become commonly accepted that no $W[1]$ -hard (and $W[i]$ -hard, $i > 1$) problem, (such as the Clique problem), can be solved in time $f(k)n^{O(1)}$ for any function f . $W[1]$ -hardness has served as the hypothesis for fixed-parameter intractability. Examples include a recent result by Papadimitriou and Yannakakis [14], showing that the Database Query Evaluation problem is $W[1]$ -hard. This provides strong evidence that the problem cannot be solved by an algorithm whose running time is of the form $f(k)n^{O(1)}$, thus excluding the possibility of a practical algorithm for the problem even if the parameter k (the size of the query) is small as in most practical cases.

3. The approach based on dynamic programming

A brute-force algorithm for LCS is to find out all the subsequences of X then check all the subsequences if they were the subsequences of Y and find the longest one. If the length of X was m , there will be 2^m subsequences of X . Thus the problem will be solved in exponential time $O(2^m)$. The most common way to solve the problem of finding the longest common subsequence of two sequences is to use the approach based on dynamic programming. We give a discussion on this approach.

3.1 Characterizing the longest common sequence and a recursive solution

The following theorem is proved in [3]:

Theorem Let $X = \langle x_1 x_2 \dots x_m \rangle$ and $Y = \langle y_1 y_2 \dots y_n \rangle$ be sequences, and $Z = \langle z_1 z_2 \dots z_k \rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

The x_m here means the m th element of the sequence X , and the X_{m-1} means the first to the $(m-1)$ th elements of the sequence X . For example, for the sequence $X = \langle a b c d e \rangle$, $x_5 = e$ and $X_4 = \langle a b c d \rangle$.

There is a recursive solution for the LCS problem. Base on the Theorem, let us define $c[i, j]$ to be the length of a longest common subsequence of sequences X_i and Y_j . When $x_i = y_j$, $c[i, j] = c[i-1, j-1] + 1$; $x_i \neq y_j$, $c[i, j] = \max(c[i-1, j], c[i, j-1])$. And we set the stop rule for the recursive solution that when $i = 0$ or $j = 0$, $c[i, j] = 0$.

The recursive formula can be written as follows:

$$c[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1]+1, & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]), & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

That we can have a recursive algorithm as follow

```

LCS1-LENGTH (i, j)
1 if i = 0 or j = 0
2 then return 0
3 else if  $x_i = y_j$ 
4 then do LCS1-LENGTH (i-1, j-1) + 1
5 else if LCS-LENGTH (i-1, j) >=
LCS-LENGTH (i, j-1)

```

```

6 then do LCS-LENGTH (i-1, j)
7 else do LCS-LENGTH (i, j-1)

```

The algorithm will be done in an exponential time. This is because the recursive calls will re-compute $c(i, j)$ that might have been computed. For example, if we want to have the value of $c(3, 4)$, we need to call the function to have the value of $c(2, 3)$, $c(2, 4)$, $c(3, 3)$, and for every value, we need to call the function again and again. It will duplicate the computation and increase the complexity.

3.2 Dynamic programming

Base on the above formula, there is a dynamic programming to solve LCS problem. We use a $((n+1) \times (m+1))$ table to implement the function $c(i, j)$ (n is the length of X and m is the length of Y). The basic idea of the table building is like this:

- (1) $c[i, j] = 0$ when $i = 0$ or $j = 0$
- (2) for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, m$, compute $c[i, j]$ follow the aforementioned formula.
- (3) At the same time, we also get the $b[i, j]$ base on the way of $c[i, j]$ computing.
 - (i) if $c[i, j] = c[i-1, j-1]+1$ then $b[i, j] = \backslash$
 - (ii) if $c[i, j] = c[i-1, j]$ then $b[i, j] = \uparrow$
 - (iii) if $c[i, j] = c[i, j-1]$ then $b[i, j] = \leftarrow$

Now we have the dynamic programming as follow [3].

```

LCS2-LENGTH (X,Y)
m = length[X]
n = length[Y]
for i = 0 to m
do c[i, 0] = 0
for j = 0 to n
do c[0, j] = 0
for i = 1 to m
do for j = 1 to n
do if  $x_i = y_j$ 

```

```

    then c[i, j] = c[i-1, j-1]+1
    b[i, j] = "\"
else if c[i-1, j] >= c[i, j-1]
    then c[i, j] = c[i-1, j]
    b[i, j] = "|"
else c[i, j] = c[i, j-1]
    b[i, j] = "-"
return c and b

```

3.3 Constructing an LCS

When $x_i = y_j$, $b(i, j) = "\"$ so we can get the elements of the LCS follow the arrows of $b(i, j)$ shows. For example, we can know that the longest common subsequence of sequences $X = "ACDBAD"$ and $Y = "CDACBABA">$ is $Z = "ACBA"$. The following recursive solution prints out a LCS of two sequences X and Y [3].

```

PRINT-LCS (b, X, i, j)
1 if i = 0 or j = 0
2 then return
3 if b[i, j] = "\"
4 then PRINT - LCS (b, X, i-1, j-1)
5 print xi
6 else if b[i, j] = "|"
7 then PRINT - LCS (b, X, i-1, j)
8 else PRINT - LCS (b, X, i, j-1)

```

Follow the arrows in $b[i, j]$, if it is $"\"$, go to the upper- left of it, if it is $"-"$, go to the left of it, and if it is $"|"$, go to the top of it. The worse case is that there is no common subsequence for the two sequences, the $b[i, j]$ must be $"|"$ or $"-"$. In this case, the running time of the program is the longest and is $O(m + n)$.

4. The parameterized approach

To compute the longest common subsequence of two given sequences s_1 and s_2 of the same length n , the above approach based on dynamic programming for solving

the problem is to fill a two dimensional dynamic programming table where each entry represents the length of the longest common subsequence between the corresponding prefix of s_1 and the corresponding prefix of s_2 [3]. There are n^2 entries to be filled in the two dimensional table. Consider a diagonal band of entries starting from the middle diagonal. The basic idea of the parameterized approach is to ignore entries outside the chosen band. This approach needs to fill up a band c , thus it takes linear time $O(cn)$, with c as the parameter. In fact, the banded alignment idea has been investigated to some extent in the alignment literature [2].

The following is the pseudo-code of the parameterized approach for the LCS of two sequences. Please note the difference from the dynamic programming approach we discussed in the previous section.

PARAM-LCS2(X, Y)

```

m = length[X]
n = length[Y]
for i = 0 to m
    do c[i, 0] = 0
for j = 0 to n
    do c[0, j] = 0
for i = 1 to m
    do for j = 1 to n
        if ((i, j) is within BAND) {
            do if xi = yj
                then
                    c[i, j] = c[i-1, j-1]+1
                    b[i, j] = "\"
            UP_AND_LEFT
            else if c[i-1, j] >= c[i, j-1]
                then
                    c[i, j] = c[i-1, j]
                    b[i, j] = UP
                else
                    c[i, j] = c[i, j-1]
                    b[i, j] = LEFT
        }
}

```

// Trace the backtracking matrix.

```

ii = n;
jj = m;
pos = S[ii][jj];
lcs = (char *) malloc( (pos+1) *
sizeof(char) );
while ( i > 0 || j > 0 ) {
    if ( b[i][j] == UP_AND_LEFT ) {
        i--;
        j--;
        lcs[pos--] = a[i];
    }
    else if ( b[i][j] == UP ) {
        i--;
    }
    else if ( b[i][j] == LEFT ) {
        j--;
    }
    else {
        printf("band is not big
enough! Optimal not found!\n");
    }
}

```

Experiment results show the efficiency of the parameterized approach. Especially when the two given sequences are very similar, we could pick a relatively small value for the band c in order to achieve the optimal solution, i.e., the longest common subsequences of the given two sequences. We provide the following two sets of experiment results:

(1) Given string1 of length = 1020, strings2 of length = 1106, the longest common subsequence of the two strings:

LCS=
“cgtgctgtttcatatgaccgggatactgatggcgtacaca
ttagataaacgtatgaagatttgagtacaaaactttcgaataC
TCGTCA Taaacaaaaatagcttgaatgagatccctgg
gatgactttgctctcccgattttgaatatgtgctgagaattgat
gtcacaatcggaacaacgcggcccacggctgtttcatatac
cgggacCCTGCCGTTGGGCATctaacattagat
aaaaaatttgagtacaaaactttccgaataaactgcttga
gatccctgggatgactttgctctcccgattttgaatatgtgct
gagaattgatgtcacaatcggaacaacgcggcccacggctg
tttcatataaccgggacCCTGCCGTTGGGCATcta
acattagataaaaaaatttgagtacaaaactttccgaataaac
tagtcttgagtatccctgggatgactttgctctcccgattttgaa
tatgtgctgagaattgatgtcacaatcggaacaacgcggccc

acggctgtttcatataaccgggacCCTGCCGTTGGG
CATctaacattagataaaaaaatttgagtacaaaactttccg
aataaactagcttggagtatccctgggatgactttgctctcccg
attttgaatatgaatat”

Table 1. Evaluation of the parameterized approach based on different band values (with two given sequences of length more than 1000).

Value of the parameter	Time needed	Find the optimal solution (yes/no)
band = 100	0.140000 seconds	no
band = 200	0.190000 seconds	yes
band = 400	0.220000 seconds	yes
band = 500	0.230000 seconds	yes
band = 600	0.250000 seconds	yes
band = 700	0.270000 seconds	yes
band = 1000	0.280000 seconds	yes
No band	0.280000 seconds	yes

(2) Given string1 of length = 2775, strings2 of length = 2775, the longest common subsequence of the two strings is omitted here.

Table 2. Evaluation of the parameterized approach based on different band values (with two given sequences of length more than 2500).

Value of the parameter	Time needed	Find the optimal solution (yes/no)
band = 100	0.870000 seconds	yes
band = 300	1.010000 seconds	yes

band = 500	1.150000 seconds	yes
band = 700	1.270000 seconds	yes
band = 900	1.370000 seconds	yes
band = 1100	1.480000 seconds	yes
band = 1300	1.690000 seconds	yes
No band	1.860000 seconds	yes

5. Summary

In this paper the problem for finding the longest common subsequence of two sequences is investigated. First the paper gives a discussion on the well-known approach based on dynamic programming, which has time complexity of $O(n^2)$. Then it presents a parameterized approach for the problem. By choosing a proper parameter, the band c , the approach achieves a time complexity of $O(cn)$. The approach is compared with the well-known dynamic programming approach. The parameterized approach is much more efficient, especially when it is applied to find the longest common subsequence of two large scale sequences.

6. References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi, *complexity and Approximation, Combinatorial Optimization Problems and Their Approximability Properties*, Springer-Verlag, 1999.

[2] K.M. Chao, W.R. Pearson, and W. Miller, "Aligning two sequences within a specific diagonal band," *Computer Applications in the Biosciences* 8, pp. 481-487, 1992.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Second Edition, the MIT press 2001.

[4] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon, "Solving large FPT problems on coarse-grained parallel machines," *Journal of Computer and System Sciences*, vol. 67, pp. 691-701, 2003.

[5] J. Chen, I. Kanj, and W. Jia, "Vertex cover: further observations and further improvements," *Journal of Algorithms*, vol. 41, pp. 280-301, 2001.

[6] X. Deng, G. Li, Z. Li, B. Ma, and L. Wang, "A PTAS for distinguishing (sub)string selection," *Lecture Notes in Computer Science*, vol. 2380, pp. 740-751, 2002.

[7] X. Deng, G. Li, Z. Li, B. Ma, and L. Wang, "Genetic design of drugs without side-effects," *SIAM Journal on Computing*, vol. 32, pp. 1073-1090, 2003.

[8] R. Downey and M. Fellows, *Parameterized Complexity*, Springer, New York, 1999.

[9] M. R. Gary and D. S. Johnson, *Computers and Intractability: a Guide to the Theory of NP-Completeness*, W. H. Freeman and Co, 1979.

[10] T. Jiang and M. Li, "On the Approximation of Shortest Common Supersequences and Longest Common Subsequences," *SIAM Journal on Computing*, vol. 24, pp. 1112-1139, 1995.

[11] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang, "Distinguishing string selection problems," *Information and Computation*, vol. 185, No. 1, pp. 41-55, 2003.

[12] M. Li, B. Ma, and L. Wang, "On the closest string and substring problems," *Journal of the ACM*, vol. 49, pp. 157-171, 2002.

[13] J. W. Raymond and P. Willett, "Maximum common subgraph isomorphism

algorithms for the matching of chemical structures,” *Journal of Computer-aided Molecular Design*, vol. 16, pp. 521-533, 2002.

[14] C. Papadimitriou and M. Yannakakis, “On the complexity of database queries,” *Journal of Computer and System Sciences*, vol. 58, pp. 407-427, 1999.