String Matching Evaluation Methods for DNA Comparison

Izzat Alsmadi and Maryam Nuser

Computer Information Systems Dept. / Yarmouk University 21163
{ialsmadi, mnuser}@yu.edu.jo

Abstract

Studying patterns in DNA sequences has been for years the subject of many research papers in bioinformatics. This paper evaluates two algorithms used for DNA comparison. Those are: Longest Common Substring and Subsequence (LCS, LCSS). Evaluation is performed based on the different code implementations for those two algorithms. Accuracy and performance are the two major criteria that are used for the evaluation of algorithms' implementation. Results showed that while those two algorithms are popular, their implementations are not consistent through research papers or websites that use and implement those algorithms for DNA sequence comparison.

Keywords: *DNA similarity algorithms, string search, DNA sequence comparison, DNA analysis, pattern recognition, Longest Common Substring, Longest Common Subsequence*

1. Introduction

Most people share very similar gene sequences, while some regions of DNA sequences vary from one person to another with high frequency. Comparing variation in these regions allows scientists to answer the question of whether two different DNA samples come from the same person. A DNA sequence represents the genetic code contained within an organism. The genetic code is a set of sequences which define what proteins to build within the organism.

This paper will focus on the subject of documents' similarity algorithms in the scope of using those similarity algorithms for DNA comparison. Despite the fact that similarity algorithms and DNA comparison are existed and used for years through several simple and complex free and commercial tools, there is a wide spectrum of applications for the usage of DNA comparison, analysis, construction, etc.

On the theoretical side, some of the famous algorithms used in DNA comparison are: Longest Common Substring (LCS) and Longest Common Sub Sequence (LCSS). We will evaluate algorithms used to compute those algorithms in terms of performance and accuracy.

1.1 Techniques to Detect Documents Similarity

In this area, there are many methods to judge similarity between documents. A brute force approach compares the subject document with investigated documents word by word. However, in most cases, such approach is time and resources' consuming. In addition, such approach can be easily tricked through editing a small number of words in the document. A more effective approach depends or is based on metrics related to the documents such as the number of statements, paragraphs, punctuation, etc. [1, 2]. A similarity index is calculated to measure the amount of similarity between documents based on those metrics. Comparing the approach of taking the document word by word in comparison to statement or paragraph by paragraph for example can have several contradicting tradeoffs. On one side, word by word comparison can minimize the effect of changing one or a small number of words relative to the total document. However, this can be time consuming and word to word document similarity may not necessarily mean possible plagiarism especially if the algorithm did not take the position of the words into consideration. Sentence or paragraph by paragraph approach is also affected by several variances such as the difference in size between the compared documents and the amount of words edited in those statements or paragraphs.

Hashing algorithms are also used to measure documents similarity. Hashing algorithms are used originally in security to verify the integrity of an investigated disk drive and protect it from being tampered. Hashing can be calculated for a word, a paragraph, a page, or a whole document. Manber presented approximate index concept to measure similarity between strings in different documents [3]. A tool called “Sif” is developed to find similar files in a large file system. He proposed the concept of approximate index to measure the similarity of character strings between documents, which was adopted later by many similar systems. The tool we developed in this paper uses two different search algorithms. The first one searches for possible similar documents for the subject document through a directory of files. The other algorithm searches for similar documents through the Internet. Calculating similarity between documents does not require in many cases similarity in cosmetic attributes such as the file type, size, number of words, etc. The author defined a checksum algorithm called “fingerprint” that is based on defining keywords in each document and parse a certain amount of characters starting from those keywords to calculate similarity.

1.2. Why DNA Sequence Comparison?

In a DNA sequence, or a molecule of DNA, there are four nucleotide bases: Adenine, Guanine, Cytosine, and Thymine. The knowledge of a DNA sequence and gene analysis can be used in several biological, medicine and agriculture research fields such as: possible disease or abnormality diagnoses, forensics, pattern matching, biotechnology, etc. The analysis and comparison studies for DNA sequences connected information technology tools and methods to accelerate findings and knowledge in biological related sciences.

DNA sequence analysis can be used to identify possible errors or abnormality in a DNA sequence (e.g. in comparison with a normal one). It can be also used to predict the function of a particular gene and compare it with other “similar” genes from same or different organisms.

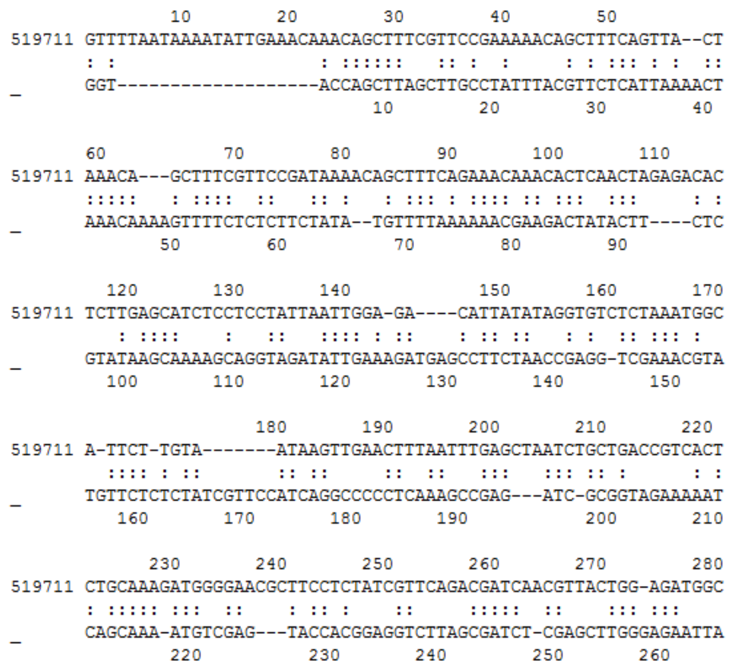
If a new DNA sequence is discovered its functionality is specified based on its similarity with known DNA sequences. Such technique is used in several medical applications and research studies.

1.3. DNA Sequence Alignment

In DNA sequence alignment process, similarity between two or more sections of genetic codes is studied in terms of quantity. Those comparisons can be used to discover information such as: evolutionary divergence, the origins of disease, and ways to apply genetic codes from one organism into another. In DNA sequence alignment, sequences are aligned and similar characters between the two sequences are tagged (Figures 1 and 2).

```
v - i n t n e r -
    |   |   | |
w r i - t - e r s
```

Figure 1. A Simple Sequence Alignment Example



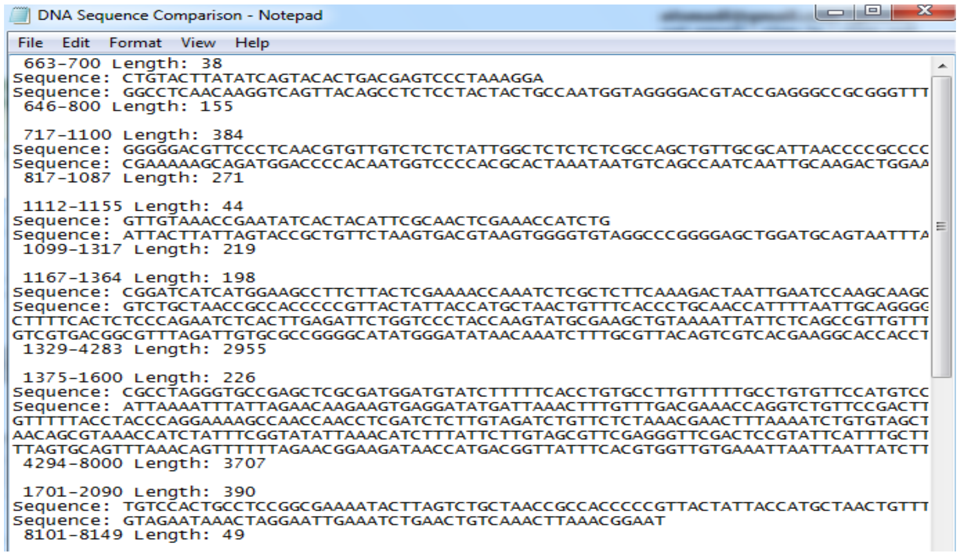
```
               10        20        30        40        50
519711 GTTTTAATAAAATATTGAAACAAACAGCTTTCGTTCCGAAAAACAGCTTTCAGTTA--CT
       : :                   : ::::::   :: :  :     : : ::: : :  ::
_      GGT-------------------ACCAGCTTAGCTTGCCTATTTACGTTCTCATTAAAACT
                                  10        20        30        40

       60           70        80        90       100       110
519711 AAACA---GCTTTCGTTCCGATAAAACAGCTTTCAGAAACAAACACTCAACTAGAGACAC
       :::::   : ::::  ::   :: :   : ::: : :::: :: :::  :::     : :
_      AAACAAAAGTTTTCTCTCTTCTATA--TGTTTTAAAAAACGAAGACTATACTT----CTC
              50        60          70        80        90

         120       130       140            150       160       170
519711 TCTTGAGCATCTCCTCCTATTAATTGGA-GA----CATTATATAGGTGTCTCTAAATGGC
          : ::::    :   ::   :::: : ::    : :: ::   : :  :: ::: :
_      GTATAAGCAAAAGCAGGTAGATATTGAAAGATGAGCCTTCTAACCGAGG-TCGAAACGTA
         100       110       120       130       140        150

                       180       190       200       210       220
519711 A-TTCT-TGTA-------ATAAGTTGAACTTTAATTTGAGCTAATCTGCTGACCGTCACT
         :::: : ::       :: ::     ::  ::   :::   ::: :: :      : :
_      TGTTCTCTCTATCGTTCCATCAGGCCCCCTCAAAGCCGAG---ATC-GCGGTAGAAAAAT
          160       170       180       190           200       210

             230       240       250       260       270        280
519711 CTGCAAAGATGGGGAACGCTTCCTCTATCGTTCAGACGATCAACGTTACTGG-AGATGGC
       : ::::: :::  ::    : :: :    ::     :::::  ::    ::: :::
_      CAGCAAA-ATGTCGAG---TACCACGGAGGTCTTAGCGATCT-CGAGCTTGGGAGAATTA
               220          230       240        250       260
```

Figure 2. DNA Sequence Alignment Sample

1.4 DNA Comparison and Analysis Tools

For years, DNA comparison has been used in biology and forensics to discriminate and compares genes or genomes. Those tools vary in size, complexity and functionality based on several factors. Some small tools or websites are developed as free or open source for research or experimental purposes. Examples of such small size limited purpose tools or applications are: Double Act (http://www.hpa-bioinfotools.org.uk/pise/double_act.html), Genomatix (http://www.genomatix.de), Mobyle (http://mobyle.pasteur.fr), ALIGN, FASTA, etc. BLAST: (Basic Local Alignment Search Tool)[4] is an example of a larger scale. Most of these algorithms uses Smith–Waterman algorithm for performing sequence alignment. This algorithm which is also used in crimes' forensic investigation does not use full DNA to DNA sequence comparison. It rather selects several segments (e.g. eight segments) selected from the different locations of the DNA. BLAST uses also dynamic programming and "seeding" to find starts of possible matches. The goal is to accelerate the process of finding matches between DNA sequences as this can take a significant amount of time and resources.

Another process that can be different from one tool to another is the ranking of the different matches. This can particularly occur when more than a match is in the same size (e.g. athe, tbcd, find, all are of size 4). Through the algorithms that we will evaluate and use, we will see such difference where some algorithms use the first match; other algorithms use the last match, etc. Figure 3 shows a sample of two DNA sequence comparisons where the tool shows the areas of match between the two DNA sequences.



```
 663-700 Length: 38
Sequence: CTGTACTTATATCAGTACACTGACGAGTCCCTAAAGGA
Sequence: GGCCTCAACAAGGTCAGTTACAGCCTCTCCTACTACTGCCAATGGTAGGGGACGTACCGAGGGCCGCGGGTTT
 646-800 Length: 155

 717-1100 Length: 384
Sequence: GGGGGACGTTCCCTCAACGTGTTGTCTCTCTATTGGCTCTCTCTCGCCAGCTGTTGCGCATTAACCCCGCCCC
Sequence: CGAAAAAGCAGATGGACCCCACAATGGTCCCCACGCACTAAATAATGTCAGCCAATCAATTGCAAGACTGGAA
 817-1087 Length: 271

 1112-1155 Length: 44
Sequence: GTTGTAAACCGAATATCACTACATTCGCAACTCGAAACCATCTG
Sequence: ATTACTTATTAGTACCGCTGTTCTAAGTGACGTAAGTGGGGTGTAGGCCCGGGGAGCTGGATGCAGTAATTTA
 1099-1317 Length: 219

 1167-1364 Length: 198
Sequence: CGGATCATCATGGAAGCCTTCTTACTCGAAAACCAAATCTCGCTCTTCAAAGACTAATTGAATCCAAGCAAGC
Sequence: GTCTGCTAACCGCCACCCCCGTTACTATTACCATGCTAACTGTTTCACCCTGCAACCATTTTAATTGCAGGGG
CTTTTCACTCTCCCAGAATCTCACTTGAGATTCTGGTCCCTACCAAGTATGCGAAGCTGTAAAATTATTCTCAGCCGTTGTTT
GTCGTGACGGCGTTTAGATTGTGCGCCGGGGCATATGGGATATAACAAATCTTTGCGTTACAGTCGTCACGAAGGCACCACCT
 1329-4283 Length: 2955

 1375-1600 Length: 226
Sequence: CGCCTAGGGTGCCGAGCTCGCGATGGATGTATCTTTTTCACCTGTGCCTTGTTTTTGCCTGTGTTCCATGTCC
Sequence: ATTAAAATTTATTAGAACAAGAAGTGAGGATATGATTAAACTTTGTTTGACGAAACCAGGTCTGTTCCGACTT
GTTTTTACCTACCCAGGAAAAGCCAACCAACCTCGATCTCTTGTAGATCTGTTCTCTAAACGAACTTTAAAATCTGTGTAGCT
AACAGCGTAAACCATCTATTTCGGTATATTAAACATCTTTATTCTTGTAGCGTTCGAGGGTTCGACTCCGTATTCATTTGCTT
TTAGTGCAGTTTAAACAGTTTTTTAGAACGGAAGATAACCATGACGGTTATTTCACGTGGTTGTGAAATTAATTAATTATCTT
 4294-8000 Length: 3707

 1701-2090 Length: 390
Sequence: TGTCCACTGCCTCCGGCGAAAATACTTAGTCTGCTAACCGCCACCCCCGTTACTATTACCATGCTAACTGTTT
Sequence: GTAGAATAAACTAGGAATTGAAATCTGAACTGTCAAACTTAAACGGAAT
 8101-8149 Length: 49
```

Figure 3. A Sample Output of 2 DNA Sequences' Comparison [http://mobyle.pasteur.fr].

1.5 DNA Document Similarity Applications and Algorithms.

Longest common substring and subsequence (LCS, LCSS)

In addition to DNA matching applications, there are several types of applications that implement exact string matching algorithms. Examples of such applications include: code, document and exam plagiarism, automatic grading, file comparison, screen display, language auto correct, translate, etc.

There are several metrics and algorithms used to decide and evaluate whether and how much two DNA sequences are similar. In this paper, we will focus on evaluating two algorithms: longest common substring and longest common subsequence. Those two algorithms have been used for years in different string comparison. From now on, we will differentiate the abbreviation for Longest Common Subsequence as (LCSS) in comparison to LCS for Longest Common Substring. The main difference between LCS and LCSS is that LCS considers only consecutive characters unlike LCSS.

1.5.1 LCS: In LCS, the algorithm searches for the longest possible string between two string sequences or files. For example, LCS between the two strings:

1… This is the first string

2…. The second string

LCS is "string" and the longest string length is 6.

Some algorithms may consider it as 7 (taking the empty space before the string in consideration). Text case is usually ignored. The second longest string can be 5: strin. Table 1 shows examples of different strings and their LCS values.

Table 1. LCS for Several String Examples

String1	String2	LCS
ABAZDC	BACBAD	BA
ABCDGH	AEDFHR	A
AGGTAB	GXTXABYB	AB
GCGCAATG	GCCCTAGCG	GCG
BC BCBD C	E BCBD D BCBD A	BCBD
cs106b	Rocks	C
Abcdef	Thw	“”

A dot plot (Figure 4) is usually used to visualize showing LCS on a small scale where the longest continuous diagonal line represents LCS (after writing one of the strings vertically and the other horizontally).

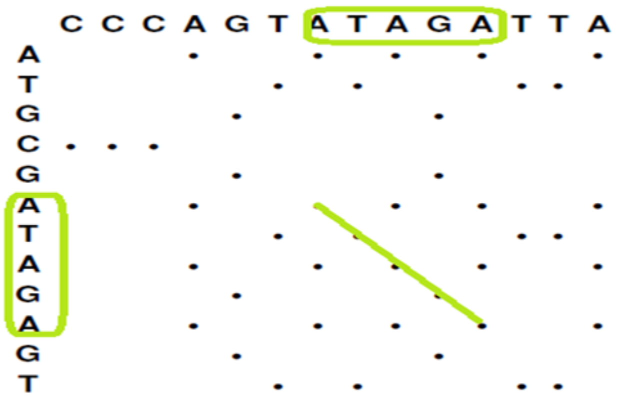


Figure 4. A Sample DNA Dot Plot

1.5.2 LCSS: A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, etc are subsequences of “abcdefg”. This means that a string of length n has 2n different possible subsequences. The LCSS problem is to find a common subsequence that is as long as possible. For example the LCSS of ggcaccacg and acggcggatacg is: ggcaacg. Table 2 shows examples of different strings and their LCSS values.

Table 2. LCSS for Several String Examples

String1	String2	LCSS
ABAZDC	BACBAD	ABAD
ABCDGH	AEDFHR	ADH
AGGTAB	GXTXABYB	GTAB
GCGCAATG	GCCCTAGCG	GCCCT
BC BCBD C	E BCBD D BCBD A	BCBD
cs106b	Rocks	Cs
Abcdef	Thw	“”

Other examples of DNA sequence similarity metrics

1. Percent of similarity. In some cases, maybe we are looking for a minimum cutoff match in percentage between two DNA sequences. For example, we are looking for at least 70 % match between two DNAs. How we define that those two DNAs at least 70 % match or common?

2. Longest Repeated Substring Problem (LRSP) or Exact String Matching (ESM). These methods compare two DNA sequences for a possible identical match (e.g. forensic investigation). Given a DNA, look through a DNA database for similarities. This is usually used in computer crimes' forensics where investigators are interested in finding the most appropriate match for a particular DNA sequence. As a DNA complete sequence is very large, several sections are taken from several locations. The matched DNA is considered as the one which has all sections match that of the subject DNA sequence.

3. Finding Palindrome: In DNA a complemented palindrome is a sequence of base pairs that reads the same backwards and forward across the double strand. The enzymes that cut these specific sites are called restriction enzymes. Therefore by looking for complemented palindromes we can identify the binding sites for restriction enzymes.

4. Minimal edit distance (aka Sequence alignment): Edit distance can be thought of as the "difference" between two strings. The difference between two strings is measured by counting the number of edit operations which must be performed, character by character, to transform one string into another. These edit operations are: R = replace, I = insert, D = delete, and M = match. For example, to transform the string "cat" to the string "chat" we can insert (I) the character 'h' between the 'c' and 'a' of "cat", yielding the string "chat".

2. Related Work

Several methods were suggested to find sequence similarity. Some of these search for exact matches between sequences with no alignment [5, 6, 7] while others allow for insertions or deletions trying to find the best possible alignment [4].

The first description of a sequence similarity search method that allows insertions and deletions was published in [8] where a computer program for finding similarities in the amino acid sequences of two proteins was developed. .

Some similarity algorithms depend on the longest common subsequence (LCS) idea that is commonly used in computer science to find the similarity between different sequences. In [9], the authors introduced new variants of LCS problem and presented efficient algorithms to solve them. They showed the ability of their algorithms to solve several molecular biology problems.

Furthermore, a parallel version of the LCS algorithm that finds the alignment between DNA and protein sequences was built in BLAST. The algorithm was tested and showed an increase in the performance of about 24-30% than the serial LCS.

In [10], the authors used the LCS as a building block for their proposed algorithm that searches for specific motifs in a DNA database. Then the algorithm was generalized to solve the common sub-sequence problem from the computational aspect. Although the complexity of the algorithm is exponential in general but it is polynomial when the threshold value (t) and the length of the largest common subsequence (c) are sufficiently close.

Another efficient algorithm to solve the LCS problem was presented in [11]. A solution of a variant of the algorithm namely constrained LCS that gets its motivation from computational bio-informatics was also suggested.

A search tool was developed in [12] that works on molecules with the SMILES format and searches the database for the specified user's query.

One area where DNA similarity algorithms are used is in compression techniques. The idea is to find regions of repeated subsequences and write them once saving a space of repeating these subsequences. This was applied in [10] where a ratio of 4.2% similarity was found within the same sequence and reached 18% when comparing 16 chromosomal sequences.

In [13], two algorithms for similarity measure between sequential data were proposed. Each algorithm uses a different data structure. The algorithms were tested on network data intrusion detection and showed a linear running time in the sequence length. The algorithm can be applied also in security and bioinformatics.

Biological sequences are very large in size and require algorithms that work with large scale data. Therefore, the new technology should be utilized to speed up procedures. Challenges still face researchers in integrating data exploration tools with a variety of different architectural requirements and natural programming models. A case study that was applied on DNA sequence analysis presents these challenges [14].

Several software tools were built to find the similarity between biological sequences. Basic Local Alignment Search Tool (BLAST) [4] is one of the most commonly used web tools for comparing primary biological sequence information whether proteins or DNA sequences. One problem that may occur with web tools is the semantic type mismatch in scientific workflows. This problem was tackled in [15] and a similarity search on DNA sequences was applied that guarantee semantic type correctness in scientific workflows.

Another tool that is used for multiple sequence alignment is DIALIGN which combines both local and global alignment features and uses dynamic programming in its algorithm [16]. An accelerator was built in hardware to improve the performance of this tool and experiments show a clear progress in the retrieval of alignments for large biological sequences.

Several other tools were built whether for DNA sequences or proteins [17, 18, 19]. A study that compares several tools was done experimentally in [20] which shows that new variations of old algorithms were efficient in practice. In addition, the authors mentioned that the algorithms efficiency depends on the processor and compiler.

3- Goals and Approaches

We can calculate DNA sequences similarity based on:

1. Number of string matches (strings of length above 2) to the total size of the DNA sequences.

2. The number of characters in the maximum string match between the two DNA sequences

LCS and LCSS are two popular metrics to measure the level of similarity between two DNA sequences. We will evaluate different implementations for the algorithms LCS and LCSS based on performance and accuracy.

It is noticed while surveying related research papers and articles that there are some conflicting results in calculating LCS and LCSS. In this experiment, we tried to define the different approaches used to develop those algorithms in order to compare their results in terms of accuracy and performance.

All those algorithms are implemented in C# and Java programming languages. Some of those codes are taken from research resources, while we developed other algorithms based on either algorithmic description or pseudo codes described in the literature.

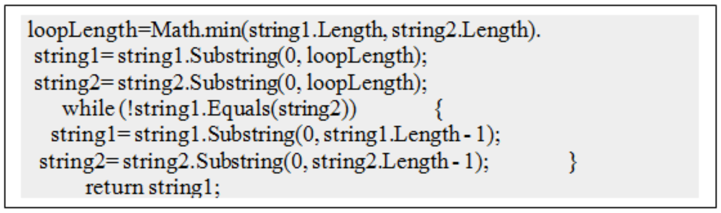
In the following sections, we will describe in generic pseudo code the algorithms used to evaluate both LCS and LCSS.

3.1 LCS Pseudo Codes

Research papers, websites and literature, have several methods to code the Longest Common Substring (LCS) algorithm. Some of those methods use generic code of methods and variables. Others utilize new types of data structure for fast access and information retrieval. Dynamic programming is one of the techniques used to find the best solution in the shortest time. Following is a description of the different algorithms used in the experiments as a pseudo code. All algorithms use the same generic code shield structure for calling the two DNA sequences to evaluate and then for saving the results and calculating the overall required time.

- **LCS 1**

This first simple algorithm loops through all possible string combinations from the two strings in comparison, tests and compares them to find the longest possible match. Such algorithm assumes no previous knowledge of where the longest path can be and hence loops thoroughly through the two strings to return the longest common string. In this algorithm one string is set to be the reference and the other string to loop through. If there are several longest common strings with the same length, a first or default can be defined. Other approaches or versions of this code take the loop count as the length of the smaller string (e.g. n = minimum (string1.Length, string2.Length). The algorithm may get seriously slow if the length of both strings is large. Later on, LCS is calculated as:

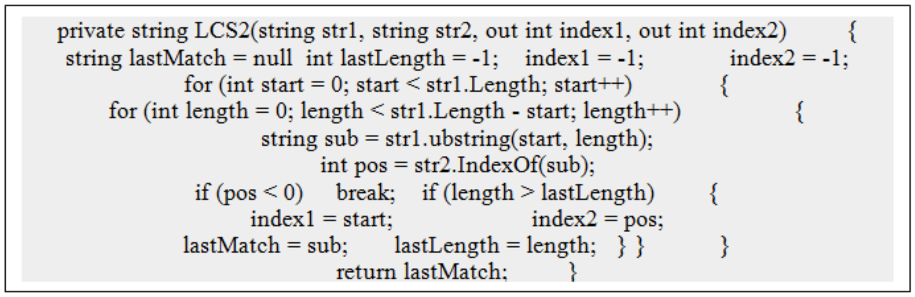


```
loopLength=Math.min(string1.Length, string2.Length).
 string1= string1.Substring(0, loopLength);
 string2= string2.Substring(0, loopLength);
     while (!string1.Equals(string2))          {
   string1= string1.Substring(0, string1.Length - 1);
  string2= string2.Substring(0, string2.Length - 1);          }
         return string1;
```

Figure 5. LCS------Algorithm1

- **LCS 2**

This algorithm also uses a generic code similar to LCS1 with two loops. Figure 6 shows the pseudo code for the algorithm.

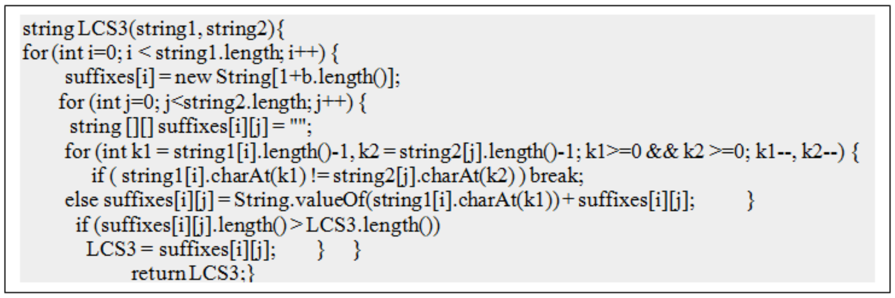


```
private string LCS2(string str1, string str2, out int index1, out int index2)      {
 string lastMatch = null  int lastLength = -1;    index1 = -1;          index2 = -1;
                 for (int start = 0; start < str1.Length; start++)          {
        for (int length = 0; length < str1.Length - start; length++)             {
                         string sub = str1.ubstring(start, length);
                                int pos = str2.IndexOf(sub);
                  if (pos < 0)     break;   if (length > lastLength)       {
                         index1 = start;                  index2 = pos;
                  lastMatch = sub;       lastLength = length;   } }          }
                              return lastMatch;         }
```

Figure 6. LCS------Algorithm 2

- **LCS 3**

Similar to most traditional implementations of LCS, LCS 3 uses a two dimensional array structure and two nested loops. In this specific implementation, three loops are used. The goal of the third inner loop is to improve performance and reduce the number of cycles.

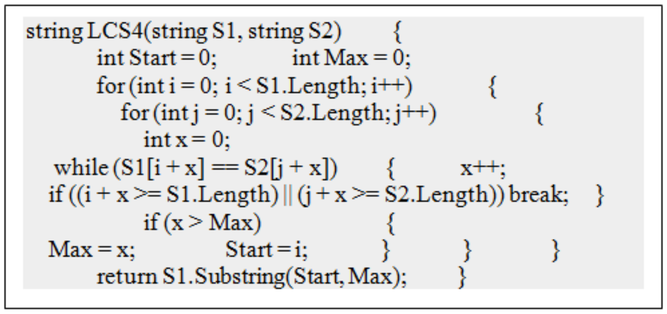


```
string LCS3(string1, string2){
for (int i=0; i < string1.length; i++) {
      suffixes[i] = new String[1+b.length()];
     for (int j=0; j<string2.length; j++) {
       string [][] suffixes[i][j] = "";
      for (int k1 = string1[i].length()-1, k2 = string2[j].length()-1; k1>=0 && k2 >=0; k1--, k2--) {
          if ( string1[i].charAt(k1) != string2[j].charAt(k2) ) break;
      else suffixes[i][j] = String.valueOf(string1[i].charAt(k1)) + suffixes[i][j];        }
        if (suffixes[i][j].length() > LCS3.length())
          LCS3 = suffixes[i][j];      }    }
                 return LCS3;}
```

Figure 7. LCS------Algorithm 3

- **LCS 4**

This algorithm is also similar to the previous algorithm with minor changes. If there are several possible matches (with the same length), the output can be different. This may explain why popular tools that can measure LCS such as: Blast (www.ncbi.nlm.nih.gov/BLAST), MB (http://www.molbiosoft.de/), Double Act (http://www.hpa-bioinfotools.org.uk/pise/double_act.html), etc may show different results. The algorithm shows that the last match is selected if there is more than one match.

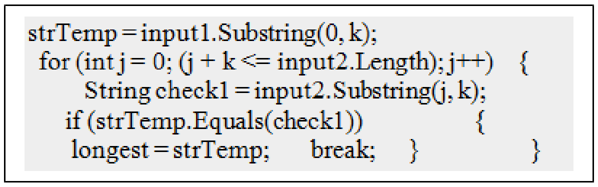


```
string LCS4(string S1, string S2)       {
          int Start = 0;            int Max = 0;
          for (int i = 0; i < S1.Length; i++)          {
             for (int j = 0; j < S2.Length; j++)             {
                int x = 0;
    while (S1[i + x] == S2[j + x])       {          x++;
   if ((i + x >= S1.Length) || (j + x >= S2.Length)) break;    }
                if (x > Max)                {
   Max = x;              Start = i;          }          }          }
          return S1.Substring(Start, Max);       }
```

Figure 8. LCS------Algorithm 4

- **LCS 5**

This variation aims to improve the performance of LCS4. Because the two nested loops in LCS4 can be time consuming especially when the strings are long, the inner loop may loop in a shorter cycle (j + k <= input2.Length) where k is an integer variable that counts the length of the maximum LCS already found. This means that if we find a LCS of length 5, there is no need to find or search for LCS that is less than 5.

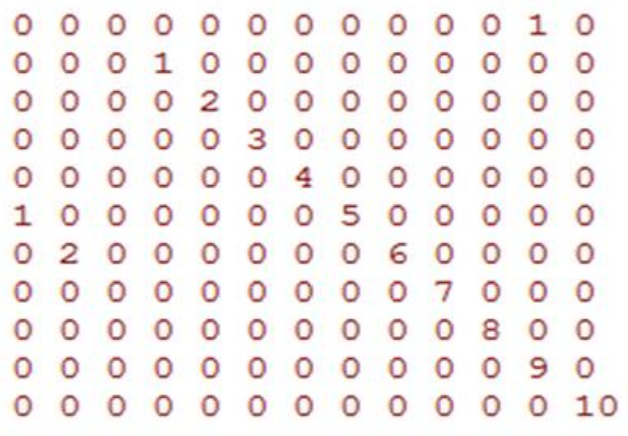


```
strTemp = input1.Substring(0, k);
 for (int j = 0; (j + k <= input2.Length); j++)   {
     String check1 = input2.Substring(j, k);
    if (strTemp.Equals(check1))              {
     longest = strTemp;      break;    }           }
```

Figure 9. LCS------Algorithm 5

- **LCS 6**

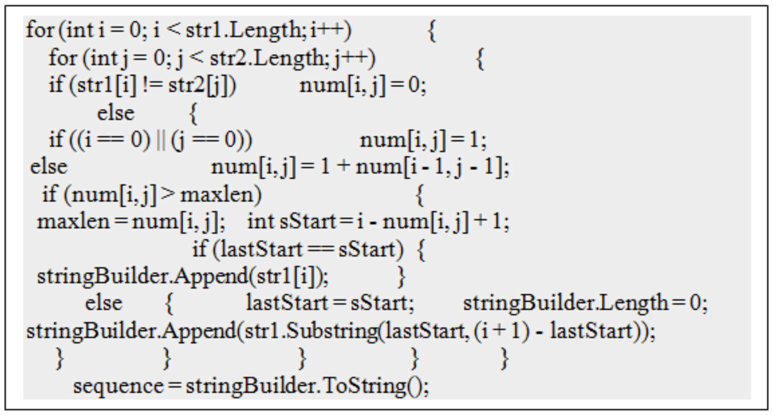
In this algorithm, a two dimensional array that has the size of the two strings in comparison is created. Each pair of parallel characters in the strings is compared. If the characters are not equal, the array value in that location is set to zero, and if they are equal, the value is set to zero in the first location which is incremented for each consecutive match. To retrieve LCS, a variable is set to a default value (e.g. one) and compared with the values in the array to find the largest value. This algorithm utilizes dynamic program method to look for the best feasible solution. Figure 10 shows a sample output from LCS 6.



```
0  0  0  0  0  0  0  0  0  0  0  1  0
0  0  0  1  0  0  0  0  0  0  0  0  0
0  0  0  0  2  0  0  0  0  0  0  0  0
0  0  0  0  0  3  0  0  0  0  0  0  0
0  0  0  0  0  0  4  0  0  0  0  0  0
1  0  0  0  0  0  0  5  0  0  0  0  0
0  2  0  0  0  0  0  0  6  0  0  0  0
0  0  0  0  0  0  0  0  0  7  0  0  0
0  0  0  0  0  0  0  0  0  0  8  0  0
0  0  0  0  0  0  0  0  0  0  0  9  0
0  0  0  0  0  0  0  0  0  0  0  0  10
```

Figure 10. A Sample Output as a Result from LCS 6

Linear, integer, dynamic programming, etc are different levels of an operational research or Artificial Intelligent (AI) field that set a matrix of input requirements and constraints for a solution, and run a solution engine to find the best possible feasible solution that can achieve all those requirements. The constraints in this case are that the LCS or the solution string should be part of both input strings (i.e. LCS(string1, string2) is part of string1 and string2), and its length should be more than 1 and less than the length of either string. The last constraint is that this LCS should be the longest. This means that there can be several string matches between those two strings and we are looking for the longest.

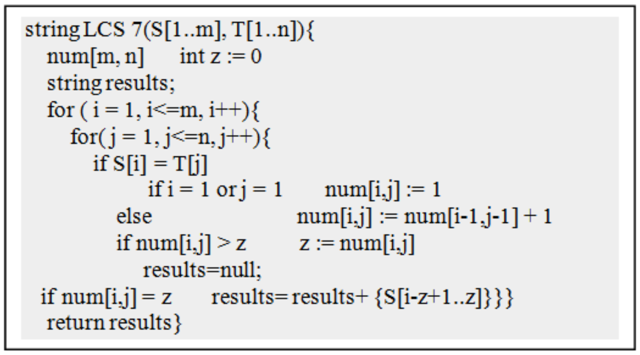


```
for (int i = 0; i < str1.Length; i++)          {
   for (int j = 0; j < str2.Length; j++)          {
   if (str1[i] != str2[j])        num[i, j] = 0;
         else       {
   if ((i == 0) || (j == 0))                num[i, j] = 1;
 else                     num[i, j] = 1 + num[i - 1, j - 1];
  if (num[i, j] > maxlen)                 {
 maxlen = num[i, j];   int sStart = i - num[i, j] + 1;
                     if (lastStart == sStart)  {
 stringBuilder.Append(str1[i]);          }
       else      {         lastStart = sStart;      stringBuilder.Length = 0;
stringBuilder.Append(str1.Substring(lastStart, (i + 1) - lastStart));
   }             }               }              }           }
      sequence = stringBuilder.ToString();
```

Figure 11. LCS------Algorithm 6

- **LCS 7**

This algorithm utilizes also dynamic programming to find the best feasible solution. The code is shown in figure 12.



```
stringLCS 7(S[1..m], T[1..n]){
   num[m, n]     int z := 0
   string results;
   for ( i = 1, i<=m, i++){
      for( j = 1, j<=n, j++){
         if S[i] = T[j]
                if i = 1 or j = 1      num[i,j] := 1
            else                   num[i,j] := num[i-1,j-1] + 1
            if num[i,j] > z        z := num[i,j]
               results=null;
  if num[i,j] = z     results= results+ {S[i-z+1..z]}}}
   return results}
```

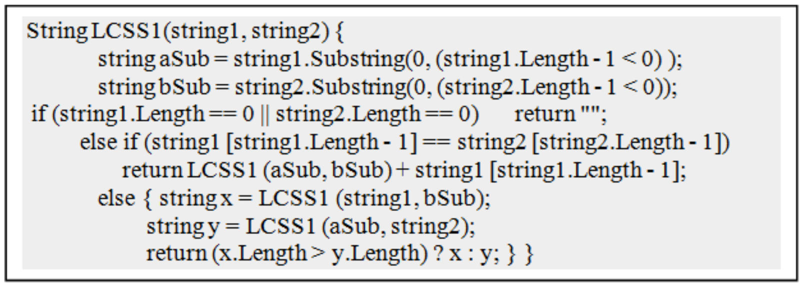
Figure 12. LCS------Algorithm 7

3.2 LCSS Algorithms

Longest Common Sub Sequence (LCSS) algorithm is expected to take longer time to solve and in some cases such method may consume all memory resources for long strings. LCSS does not require matching strings to be in the exact same order and location in the two strings. For example, between the two strings: "123456" and "1224533324", LCS is 2 (12) while LCSS is 4 (1234).

- **LCSS1**

The first algorithm uses recursion to keep checking if there is a further match. As mentioned earlier, solving LCSS requires longer time and resources in comparison to LCS. Therefore, the algorithm was very slow especially for large strings.

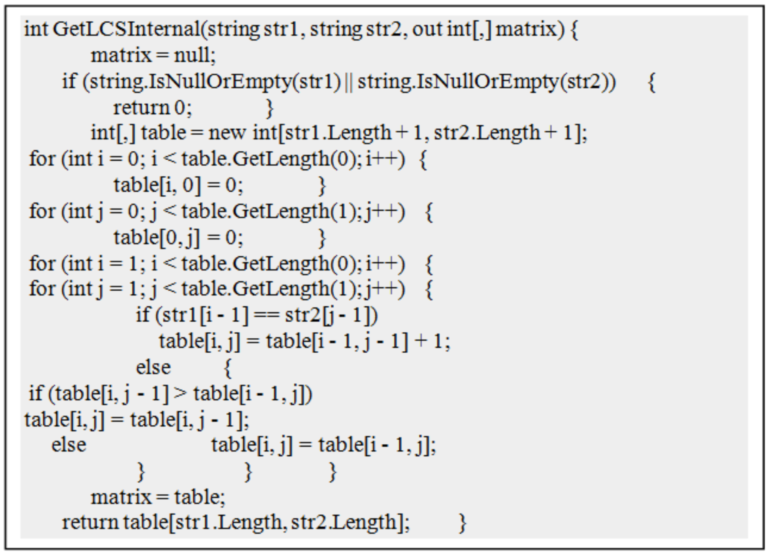


```
String LCSS1(string1, string2) {
        string aSub = string1.Substring(0, (string1.Length - 1 < 0) );
        string bSub = string2.Substring(0, (string2.Length - 1 < 0));
 if (string1.Length == 0 || string2.Length == 0)      return "";
      else if (string1 [string1.Length - 1] == string2 [string2.Length - 1])
            return LCSS1 (aSub, bSub) + string1 [string1.Length - 1];
        else { string x = LCSS1 (string1, bSub);
              string y = LCSS1 (aSub, string2);
              return (x.Length > y.Length) ? x : y; } }
```

Figure 13. LCSS------Algorithm 1

- **LCSS2**

The idea of this algorithm was inspired from Wiki books (www.thefullwiki.org). The algorithm first draws a table with the two strings in rows and columns (as characters). In each character match between the two strings a number (e.g. number 1) is added. For each consecutive match, the number is calculated. However, at this time, next (i.e. non consecutive) matches are also counted. Figure 14 shows the algorithm.



```
int GetLCSInternal(string str1, string str2, out int[,] matrix) {
         matrix = null;
     if (string.IsNullOrEmpty(str1) || string.IsNullOrEmpty(str2))     {
            return 0;            }
         int[,] table = new int[str1.Length + 1, str2.Length + 1];
 for (int i = 0; i < table.GetLength(0); i++)  {
            table[i, 0] = 0;            }
 for (int j = 0; j < table.GetLength(1); j++)   {
            table[0, j] = 0;            }
 for (int i = 1; i < table.GetLength(0); i++)   {
 for (int j = 1; j < table.GetLength(1); j++)   {
               if (str1[i - 1] == str2[j - 1])
                  table[i, j] = table[i - 1, j - 1] + 1;
               else        {
 if (table[i, j - 1] > table[i - 1, j])
table[i, j] = table[i, j - 1];
    else                  table[i, j] = table[i - 1, j];
               }            }         }
         matrix = table;
     return table[str1.Length, str2.Length];       }
```

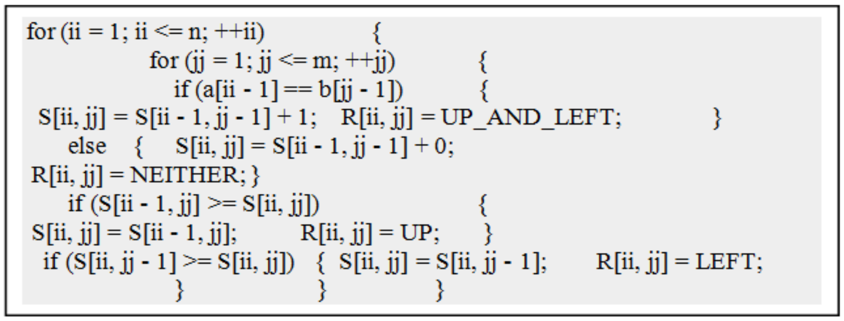
Figure 14. LCSS------Algorithm 2

- **LCSS3**

LCSS 3 utilizes dynamic programming and uses partially similar approach to LCSS 2. Figure 15 presents the algorithm where the strings in capital are "constants" which indicate a direction in the backtracking array.

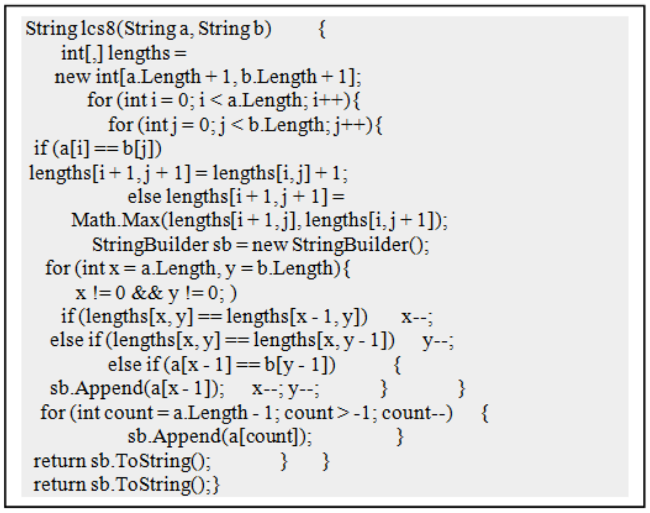
- **LCSS4**

LCSS4 algorithm represents another data structure approach for building a two dimensional array. Through the two nested loops, a backtrack process is applied whenever there is a two-character match between the two strings. Figure 16 explains the steps.



```
for (ii = 1; ii <= n; ++ii)                {
                    for (jj = 1; jj <= m; ++jj)           {
                        if (a[ii - 1] == b[jj - 1])           {
 S[ii, jj] = S[ii - 1, jj - 1] + 1;    R[ii, jj] = UP_AND_LEFT;              }
     else    {     S[ii, jj] = S[ii - 1, jj - 1] + 0;
R[ii, jj] = NEITHER; }
     if (S[ii - 1, jj] >= S[ii, jj])                       {
S[ii, jj] = S[ii - 1, jj];         R[ii, jj] = UP;       }
  if (S[ii, jj - 1] >= S[ii, jj])   {  S[ii, jj] = S[ii, jj - 1];       R[ii, jj] = LEFT;
                         }                  }               }
```

Figure 15. LCSS------Algorithm 3



```
String lcs8(String a, String b)        {
     int[,] lengths =
    new int[a.Length + 1, b.Length + 1];
         for (int i = 0; i < a.Length; i++){
            for (int j = 0; j < b.Length; j++){
 if (a[i] == b[j])
lengths[i + 1, j + 1] = lengths[i, j] + 1;
               else lengths[i + 1, j + 1] =
      Math.Max(lengths[i + 1, j], lengths[i, j + 1]);
          StringBuilder sb = new StringBuilder();
  for (int x = a.Length, y = b.Length){
       x != 0 && y != 0; )
     if (lengths[x, y] == lengths[x - 1, y])       x--;
   else if (lengths[x, y] == lengths[x, y - 1])      y--;
            else if (a[x - 1] == b[y - 1])         {
   sb.Append(a[x - 1]);     x--; y--;           }           }
 for (int count = a.Length - 1; count > -1; count--)     {
               sb.Append(a[count]);              }
 return sb.ToString();            }     }
 return sb.ToString();}
```

Figure 16. LCSS------Algorithm 4

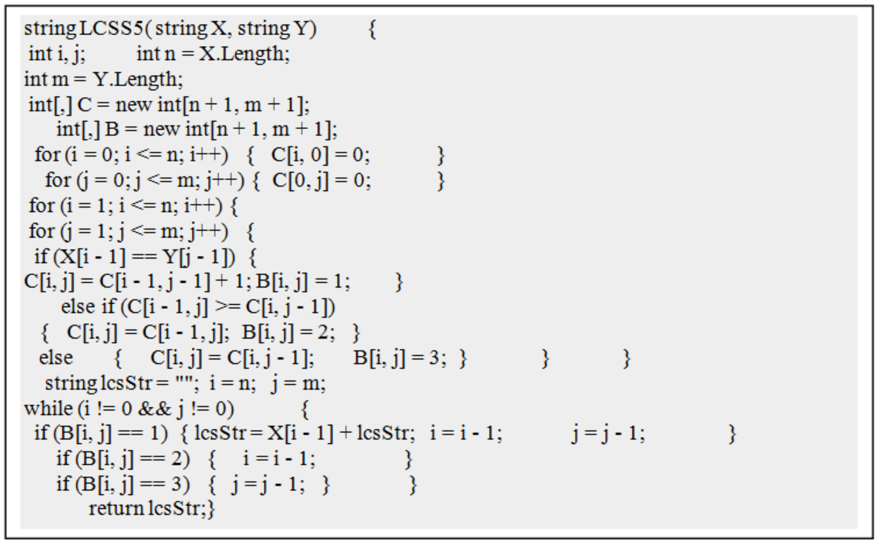
- **LCSS5**

LCSS5 is another example or version of the dynamic programming approach that uses that back track process whenever a match is found between two characters. The pseudo code of the algorithm is shown in figure17.

- **LCSS6**

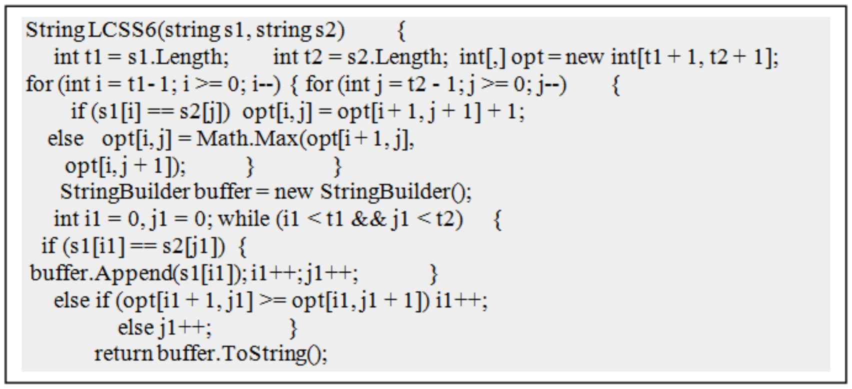
This algorithm is also somewhat similar to algorithm 4. The code is shown in Figure 18.

In order to evaluate LCS and LCSS algorithms in terms of accuracy and performance, a dataset of relatively large DNA sequences is selected. Following is a brief description of the selected dataset.



```
stringLCSS5( string X, string Y)        {
 int i, j;        int n = X.Length;
int m = Y.Length;
 int[,] C = new int[n + 1, m + 1];
     int[,] B = new int[n + 1, m + 1];
  for (i = 0; i <= n; i++)   {   C[i, 0] = 0;          }
    for (j = 0; j <= m; j++) {  C[0, j] = 0;          }
 for (i = 1; i <= n; i++) {
 for (j = 1; j <= m; j++)   {
  if (X[i - 1] == Y[j - 1])  {
C[i, j] = C[i - 1, j - 1] + 1; B[i, j] = 1;       }
      else if (C[i - 1, j] >= C[i, j - 1])
   {   C[i, j] = C[i - 1, j];  B[i, j] = 2;  }
   else      {     C[i, j] = C[i, j - 1];      B[i, j] = 3;  }          }          }
    string lcsStr = "";  i = n;   j = m;
while (i != 0 && j != 0)          {
  if (B[i, j] == 1)  { lcsStr = X[i - 1] + lcsStr;  i = i - 1;          j = j - 1;            }
     if (B[i, j] == 2)   {    i = i - 1;             }
     if (B[i, j] == 3)   {  j = j - 1;  }            }
          return lcsStr;}
```

Figure 17. LCSS------Algorithm 5

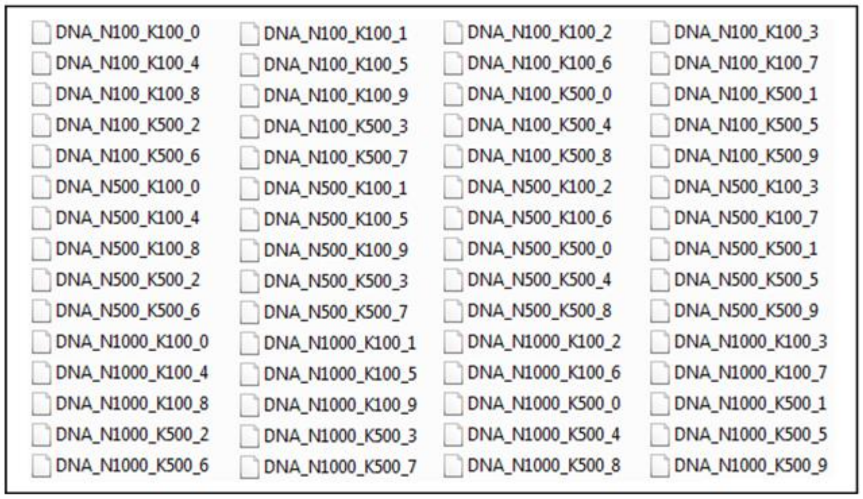


```
String LCSS6(string s1, string s2)       {
    int t1 = s1.Length;       int t2 = s2.Length;  int[,] opt = new int[t1 + 1, t2 + 1];
for (int i = t1 - 1; i >= 0; i--) { for (int j = t2 - 1; j >= 0; j--)      {
       if (s1[i] == s2[j])  opt[i, j] = opt[i + 1, j + 1] + 1;
   else   opt[i, j] = Math.Max(opt[i + 1, j],
     opt[i, j + 1]);        }           }
     StringBuilder buffer = new StringBuilder();
    int i1 = 0, j1 = 0; while (i1 < t1 && j1 < t2)     {
  if (s1[i1] == s2[j1])  {
 buffer.Append(s1[i1]); i1++; j1++;           }
    else if (opt[i1 + 1, j1] >= opt[i1, j1 + 1]) i1++;
             else j1++;            }
         return buffer.ToString();
```

Figure 18. LCSS------Algorithm 6

4. Case Study

To evaluate LCS and LCSS algorithms a dataset of DNA sequences is selected. The DNA sequences' dataset is taken from NCBI Viral Genomes (http://www.ncbi.nlm.nih.gov/genomes). Sequences are randomly selected from different genome sequences. Sequence datasets are truncated to a specific length (K) and a number of sequences (N). The dataset include sequences of lengths 100, 500, and 1000. Figure 19 below shows the names of those DNA sequences where the name reflects the number of sequences (N) and the length (K).



DNA_N100_K100_0	DNA_N100_K100_1	DNA_N100_K100_2	DNA_N100_K100_3
DNA_N100_K100_4	DNA_N100_K100_5	DNA_N100_K100_6	DNA_N100_K100_7
DNA_N100_K100_8	DNA_N100_K100_9	DNA_N100_K500_0	DNA_N100_K500_1
DNA_N100_K500_2	DNA_N100_K500_3	DNA_N100_K500_4	DNA_N100_K500_5
DNA_N100_K500_6	DNA_N100_K500_7	DNA_N100_K500_8	DNA_N100_K500_9
DNA_N500_K100_0	DNA_N500_K100_1	DNA_N500_K100_2	DNA_N500_K100_3
DNA_N500_K100_4	DNA_N500_K100_5	DNA_N500_K100_6	DNA_N500_K100_7
DNA_N500_K100_8	DNA_N500_K100_9	DNA_N500_K500_0	DNA_N500_K500_1
DNA_N500_K500_2	DNA_N500_K500_3	DNA_N500_K500_4	DNA_N500_K500_5
DNA_N500_K500_6	DNA_N500_K500_7	DNA_N500_K500_8	DNA_N500_K500_9
DNA_N1000_K100_0	DNA_N1000_K100_1	DNA_N1000_K100_2	DNA_N1000_K100_3
DNA_N1000_K100_4	DNA_N1000_K100_5	DNA_N1000_K100_6	DNA_N1000_K100_7
DNA_N1000_K100_8	DNA_N1000_K100_9	DNA_N1000_K500_0	DNA_N1000_K500_1
DNA_N1000_K500_2	DNA_N1000_K500_3	DNA_N1000_K500_4	DNA_N1000_K500_5
DNA_N1000_K500_6	DNA_N1000_K500_7	DNA_N1000_K500_8	DNA_N1000_K500_9

Figure 19. The DNA Sequences used in the Experiment

5. Analysis and Comparison

In this part, LCS and LCSS algorithms are compared each in a separate section. Algorithms are going to be compared for accuracy and performance.

1. LCS Accuracy Comparison

Before using the experimental datasets mentioned earlier and in order to visually verify reliability, there is a need to check results manually. We will first select small size strings and compare results from all algorithms with expected results. We will use the same examples described earlier. Table 3 shows a sample result from LCS accuracy comparison on simple examples.

Table 3. LCS Accuracy Comparison

Strings		Exp.	Algorithms		
1	2		1	2	3
ABAZDC	BACBAD	BA	BA	BA	BA
ABCDGH	AEDFHR	A	D	A	A
AGGTAB	GXTXABYB	AB	AB	A	AB
GCGCAATG	GCCCTAGCG	GCG	GCG	GCG	GC
BCBCBDC	EBCBDDBCBA	BCBD	BCBD	BCBD	BCBD
cs106bbbc	rocksbbc	bbc	bbc	bb	Bb
Abcdef	Thw	“”	“”	“”	“”
ABAZDC	BACBAD	Alg 4	Alg 5	Alg 6	Alg 7
ABCDGH	AEDFHR	BA	BA	BA	BA

AGGTAB	GXTXABYB	A	A	A	A
GCGCAATG	GCCCTAGCG	AB	AB	AB	AB
BCBCBDC	EBCBDDBCBA	GCG	GCG	GCG	GCG
cs106bbbc	rocksbbc	BCBD	BCBD	BCBD	BCBD
Abcdef	Thw	bbc	Bbc	bbc	bbc
ABAZDC	BACBAD	whT	“”	“”	“”

Results show that the majority of the algorithms have consistent accurate results in comparison with the manual verification of the results. For the second example (table 3, 2cnd row), Algorithm 1 is the only algorithm that shows “D” as the match instead of “A”. This is not an error and it depends on the default selection once more than one match is found. Note that all algorithms select the first match while algorithm one selects the last match. Algorithms two, three and four have errors in rows: 4, 6 and 7. For this small testing dataset, we can say that in terms of reliability or accuracy of results, we can trust algorithms: 1, 5, 6 and 7 as they showed consistent expected results in all tested rows or examples.

Accuracy testing is also applied to the experimental dataset. Due to size and visualization limitations, we will show here only a small portion of the dataset. Table 4 shows accuracy test results on a sample of the experimental dataset.

Table 4. Accuracy Test for a Sample of the Experimental Dataset

Algorithms						
1	**2**	**3**	**4**	**5**	**6**	**7**
TCGTTCCGA	TCGTTCCGA	TCGTTCCGA	TCGTTCCGA	TCGTTCCGA	TCGTTCCGA	TCGTTCCGA
GCTTTCG	GCTTTCG	TCCGATA	TCCGATA	TCCGATA	TCCGATA	TCCGATA
AATAAAAT	AATAAAAT	CCGAAAAA	CCGAAAAA	CCGAAAAA	CCGAAAAA	CCGAAAAA
AAAATATT	AAAATATT	GTTACTAA	GTTACTAA	GTTACTAA	GTTACTAA	GTTACTAA

The small sample in the previous Table shows that different LCS algorithms may show different accurate results when the size of the compared strings are the same. As explained earlier, this depends on what is the default string to display as an output if there is more than a match with the same size. Some algorithms concentrate in finding the size of the LCS only; in that case, table4 will have the same value for all its entries.

2. LCS Performance Comparison

Table 5 below shows a summary of selected results from applying all LCS algorithms on the experimental dataset. Results showed that specific algorithms such as: Algorithms 1 and 5, are significantly slow relative to the other algorithms that are somewhat similar and fast.

Table 5. Performance Test for a Sample of the Experimental Dataset

Strings				Algorithms			
1		2		1	2-3	4-5	6-7
N	K	N	K	Time in seconds-average for several selections			
100	100	100	100	12	6	6	6
100	100	100	500	17	6	5	6
100	100	500	100	23	6	7	6
100	100	500	500	20	6	6	6
100	100	1000	100	20	9	6	6
100	100	1000	500	19	7	6	5
100	500	100	500	27	5	6	5
500	100	500	100	24	5	12	6
500	500	500	500	25	7	11	6
1000	100	1000	100	27	5	81	6
1000	500	1000	500	25	6	70	6

3. LCSS Accuracy Comparison

Similar to LCS, to evaluate accuracy on LCSS, we first used simple examples that can be evaluated visually. Table 6 shows the results of this initial accuracy test on LCSS algorithms.

Table 6. LCSS Accuracy Comparison on Initial Examples

Strings		Algorithms						
1	2	Expected	1	2	3	4	5	6
ABA ZDC	BACB AD	ABAD	ABA D	AB AD	ABA D	DABC AB	ABA D	ABA D
ABC DGH	AEDF HR	ADH	ADH	A	ADH	RHFD EA	ADH	ADH
AGG TAB	GXTX ABYB	GTAB	GTA B	AB	GTA B	BYBA XTXG	GTA B	GTA B
GCGC AA TG	GCCC TAGC G	GCCTG, GCGC, GCCAG	GCC TG	GG CG	GCC AG	GCGA TCCC G	GCC TG	GCG CG
BCBC BDC	EBCB DDBC BA	BCBC B	BCB DC	B	BCB CB	ABCB DDBC BE	BCB DC	BCB CB
cs106 bbbc	Rocks bbc	Csbbc	csbbc	csb bc	Csbb c	ccbbsk cor	csbbc	csbb c
Abcde f	Thw	“”	“”	“”	“”	whT	“”	“”

LCSS accuracy evaluation in the previous table shows that algorithm four has a serious accuracy problem in almost all examples, and algorithm two has also some accuracy problems in some examples. It should be mentioned that there are no relations between LCS and LCSS algorithms developed in this paper although algorithm four in both cases is shown to be inaccurate. For LCSS accuracy purposes, we will discard algorithms two and four and consider only the rest. It is possible that the implementation of those algorithms (2 and 4) needs tuning. However, since this is not consistent in most cases, debugging such algorithms can be time consuming.

4. LCSS Performance Comparison

Table 7 shows the results of evaluating the performance of LCSS algorithms.

Table 7. Performance Test for a Sample of the Experimental Dataset

Strings				**Algorithms**				
1		**2**		**2**	**3**	**4**	**5**	**6**
N	K	N	K	Time in seconds-average for several selections				
100	100	100	100	7	7	7	7	7
100	100	100	500	6	6	5	5	6
100	100	500	100	18	7	6	9	6
100	100	500	500	8	6	6	6	5
100	100	1000	100	7	7	6	6	6
100	100	1000	500	8	8	5	6	6
100	500	100	500	11	5	5	4	5
500	100	500	100	22	5	6	5	7
500	500	500	500	21	8	5	5	4
1000	100	1000	100	65	8	6	5	5
1000	500	1000	500	68	9	6	32	9

Results from Table 7 above show that as expected LCSS calculation takes longer time in comparison to LCS. Algorithm one was very slow and in some cases time either expands to hours or exhausts system memory and causes a crash and hence their values were excluded. Algorithm two is then relatively slower than the other algorithms. Another finding is that time is not perfectly increasing with the increase in the size of the sequence.

6. Conclusion

In this paper, we evaluated the code implementation of two widely popular DNA sequence comparison algorithms: Longest common substring and longest common subsequence. A survey of those widely used algorithms in bioinformatics and DNA sequence comparison showed that they have different implementations. In addition, evaluating the same DNA sequences on different tools may show different results. While some of the differences are shown to be expected and are part of the different default considerations or interpretations of those algorithms, other results showed that implementations for the same algorithm are somewhat different and inconsistent. Using new programming data structures and algorithms showed significant improvement in terms of the efficiency in finding the solution. Further, reduction algorithms and techniques should be used to reduce the calculation speed.

References

[1] S. Grier, "A tool that detects plagiarism in Pascal programs", ACM SIGCSE Bulletin, vol. 13, no. 1, (**1981**), pp. 15-20.
[2] J. A. W. Faidhi and S. K. Robinson, "An empirical approach for detecting program similarity within a university programming environment", Computers & Education, vol. 11, no. 1, (**1987**), pp. 11-19.
[3] U. Manber, "Finding similar files in a large file system[C/OL]", In: Proceedings of the Winter USENIX Conference, (**1994**), pp. 1-10.
[4] BLAST, http://blast.ncbi.nlm.nih.gov/Blast.cgi, (**2011)** September.
[5] G. Huang, H. Zhou, Y. Li and L. Xu, "Alignment-free comparison of genome sequences by a new numerical characterization", Journal of Theoretical Biology, vol. 281, no. 1, (**2011**), pp. 107-112.

[6] C. Yu, S.-Y. Cheng, R. L. He and S. S. -T. Yau, "Protein map: An alignment-free sequence comparison method based on various properties of amino acids", Gene, vol. 486, (**2011**), pp. 110-118.
[7] Y. Guo and T. -m. Wang, "A new method to analyze the similarity of the DNA sequences", Journal of Molecular Structure: THEOCHEM, vol. 853, (**2008**), pp. 62–67.
[8] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", Journal of Molecular Biology, vol. 48, no. 3, (**1970**), pp. 443–53. doi:10.1016/0022-2836(70)90057-4. PMID 5420325, http://linkinghub.elsevier.com/retrieve/pii/0022-2836(70)90057-4.
[9] C. S. Iliopoulos and M. S. Rahman, "Algorithms for Computing Variants of the Longest Common Subsequence Problem", Theoretical Computer Science archive Journal, vol. 395, no. 2-3, (**2008**), pp. 255-267.
[10] C. -P. P. Wu, N. -F. Law and W. -C. Siu, "Cross chromosomal similarity for DNA sequence compression", Bioinformation, vol. 2, no. 9, (**2008**), pp. 412-416.
[11] C. S. Iliopoulos and M. S. Rahman, "New Efficient Algorithms for LCS and Constrained LCS Problem", In Proceedings of the Third ACiD Workshop Durham, UK, vol. 9 of Texts in Algorithmics. King's College London, (**2007**), pp. 83-94.
[12] A. F. Klaib, Z. Zainol, N. H. Ahamed, R. Ahmad and W. Hussin, "Application of Exact String Matching Algorithms towards SMILES Representation of Chemical Structure", International journal of computer and information science and engineering, (**2007**), pp. 497-501.
[13] K. Rieck, P. Laskov and K. -R. M¨uller, "Efficient Algorithms for Similarity Measures over Sequential Data: A Look Beyond Kernels", DAGM 2006, LNCS 4174, (**2006**), pp. 374–383.
[14] G. Fox, X. Qiu, S. Beason, J. Y. Choi, M. Rho, H. Tang, N. Devadasan and G. Liu, "Case Studies in Data Intensive Computing", Large Scale DNA Sequence Analysis as the Million Sequence Challenge and Biomedical Computing, (**2009**).
[15] K. Derouiche and D. A. Nicole, "Semantically Resolving Type Mismatches in Scientific Workflows", OTM 2007 Workshops, Part I, LNCS 4805, (**2007**), pp. 125–135, Springer-Verlag Berlin Heidelberg 2007.
[16] DIALIGN, http://dialign.gobics.de/, (**2011**) September.
[17] D. Rose, J. Hertel, K. Reiche, P. F. Stadler and J. Hackermüller, "NcDNAlign: Plausible multiple alignments of non-protein-coding genomic Sequences", Genomics, vol. 92, no. 1, **(2008)**, pp. 65-74.
[18] E. Dong, J. Smith, S. Heinze, N. Alexander and J. Meiler, "BCL::Align—Sequence alignment and fold recognition with a custom scoring function online", Gene, vol. 422, no. 1-2, **(2008)**, pp. 41-46.
[19] B. Vishnepolsky and M. Pirtskhalava, "ALIGN MTX—An optimal pairwise textual sequence alignment program, adapted for using in sequence-structure alignment", Computational Biology and Chemistry, vol. 33, no. 3, **(2009)**, pp. 235-238.
[20] P. Kalsi, H. Peltola and J. Tarhio, "Comparison of Exact String Matching Algorithms for Biological Sequences", In: Proc. BIRD '08, 2nd International Conference on Bioinformatics Research and Development (ed. M. Elloumi et al.). Communications in Computer and Information Science 13, Springer (**2008)**, pp. 417-426.

Authors

Izzat M. Alsmadi

Izzat Alsmadi is an associate professor in the department of computer information systems at Yarmouk University in Jordan. He obtained his Ph.D degree in software engineering from NDSU (USA), his second master in software engineering from NDSU (USA) and his first master in CIS from University of Phoenix (USA). He had a B.sc degree in telecommunication engineering from Mutah university in Jordan. He has several published books, journals and conference articles largely in software engineering and information retrieval fields.

Maryam S. Muser

Maryam Nuser is an assistant professor in the department of computer Information Systems at Yarmouk University Jordan. She received a BSc degree in Computer Science from Yarmouk University in 1995, Msc degree from the University of Arkansas, USA in 2002, and a PhD degree from the University of Arkansas in 2004 with the same major.

She worked as a dept. Head for the CIS dept. at Yarmouk University during the period 2006-2008. She has several publications in international and local journals, conferences, and books.

.