CS 584- Algorithm Design and Analysis
Sowmya Sri Indukuri

Final Project
Longest Common Subsequence Problem (LCS)

1. Abstract

The longest common subsequence problem is a classic string problem, which attempts to compute a common subsequence with the highest possible length from a given set of strings. This is one of the computational issues in bioinformatics and computational biology that has been researched most. The primary intension of the project is to perform empirical analysis of longest common subsequence algorithms such as Naïve Recursion, Dynamic Programming, Hirschberg Algorithm and solving LCS through minimum edit distance. Experimental results show the performance and efficiency of the algorithms. I have compared the efficiencies of these algorithms by their time and memory consumption. In this project we will also discuss about optimal substructure and overlapping subproblems properties of LCS problem and theoretical bounds on algorithms time complexity.

2. Introduction

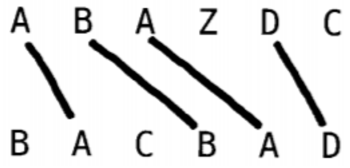
While there are many notions of similarity between strings, and many problems that we would like to optimize over strings, a natural problem (and notion of similarity) is the Longest Common Subsequence. The Longest Common Subsequence (LCS) problem is as follows. When given two strings: string S_1 of length m, and string $S_2$of length n our goal is to produce their longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings.

For example, consider:

S_1 = ABAZDC

S_2 = BACBAD

The LCS has length 4 in this case and is the string ABAD. Another way to look at it is to make a 1-1 match between some of the letters in $S_1$and some of the letters in S_2 so that none of the corresponding edges cross each other.



A classic computer science problem, the basis of data comparison programs such as the diff utility, is the longest common subsequence problem and has applications in computational linguistics and bioinformatics. Revision control systems such as Git often commonly use it to reconcile multiple changes made to a revision-controlled set of data. For genomics this type of problem happens all the time: given two DNA fragments, the LCS provides information about what they have for common and the best way to line them up. One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. Following sections address and evaluate different algorithms to solve LCS.

3. Naïve Recursion

Let $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ and wish to find length of common subsequence of X and Y. In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices $\{1,2,\ldots, m\}$ of X. Because X has 2^msubsequences, this approach requires exponential time, making it impractical for long sequences.

3.1 Pseudocode

```
m = A.length
n = B.length
lcs_length(char[] A, char []B, int m, int n)
  if (m = 0 || n = 0)
     return 0;
  else if (A[m-1] == B[n-1])
     return 1 + lcs_length(A, B, m-1, n-1);
  else
     return max(lcs_length(A, B, m, n-1), lcs_length(A, , m-1, n));
```

3.2 Complexity Analysis

- Checking = Θ(m+n) time per each subsequence.
- 2^n subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Worst-case running time = Θ ((m+n) 2^m) = exponential time.
- This algorithm is supposed to be the most unsophisticated of all and took the most time even for small string lengths. The algorithm rapidly degrades even if there is a small tilt towards the worst case.

4. Dynamic Programming

The longest common subsequence problem has two properties which can be solved using the dynamic programming approach. It has an optimal substructure: the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial. The LCS problem also has overlapping subproblems: the solution to a higher subproblem depends on the solutions to several of the lower subproblems. Problems with these two properties—optimal substructure and overlapping subproblems—can be approached by a problem-solving technique called dynamic programming, in which the solution is built up starting with the simplest subproblems. The optimal substructure of the LCS problem gives the recursive formula.

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

4.1 Memoization

The problem with the recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to lcs(), and there are exactly (m+1)(n+1) possible subproblems (a relatively small number). If there are nearly 2^n recursive calls, some of these subproblems must be being solved over and over.

The dynamic programming solution is to check whenever we want to solve a subproblem, whether we've already done it before. If so we look up the solution instead of recomputing it. Implemented in the most direct way, we just add some code to our recursive algorithm to do this look up -- this "top down", recursive version of dynamic programming is known as "memoization". After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

4.2 Pseudocode

```
m = A.length
n = B.length
  lcs(char[] A, char[] B, m, n, [][]dp)
    if(m <= 0 || n<= 0)
      return 0;
    if(dp[m-1][n-1]!= null)
      return  dp[m-1][n-1];
    if(A[m-1] == B[n-1])
      dp[m-1][n-1] = 1+lcs(A, B, m-1, n-1, dp);
      return dp[m-1][n-1];
    else
      dp[m-1][n-1] = max(lcs(A, B, m, n-1, dp), lcs(A, B, m-1, n, dp));
      return dp[m-1][n-1];
```

4.3 Complexity Analysis

- Time analysis: Each call to subproblem takes constant time. We call it once from the main routine, and at most twice every time we fill in an entry of array L. There are (m+1)(n+1) entries, so the total number of calls is at most 2(m+1)(n+1)+1 and the time is O(mn).
- As usual, this is a worst case analysis. The time might sometimes get better, if not all array entries get filled out. For instance if the two strings match exactly, we'll only fill in diagonal entries and the algorithm will be fast.
- Auxiliary Space: O(mn)

4.4 Tabulation

In the tabulation system, we just need to build a table to find the matches and in the last row and column, you get the largest number of LCS in the last cell. You then traverse the entire table that is normally a 2D array that contains the characters where the change occurs, and thus gets the LCS string. The only overhead noticeable right now is that of the integer table which is either a matrix or a 2D array. The longer the string the bigger the matrix should be.

4.5 Pseudocode

```
m = A.length
n = B.length
lcs(char[] A, char[] B, m, n)
  L[m+1][n+1]
  for i=0 to m
    for j=0 to n
      if(i==0 || j==0)
        L[i][j] = 0;
      else if (A[i-1] == B[j-1])
        L[i][j] = L[i-1][j-1] + 1;
      else
        L[i][j] = max(L[i-1][j], L[i][j-1]) //lcs length
  //print lCS
  int index = L[m][n];
  int temp = index;
  lcs[index+1]
  lcs[index] = '\0'; // Set the terminating character
  int i = m, j = n;
  while (i > 0 && j > 0)
    if (A[i-1] == B[j-1])
      // Put current character in result
      lcs[index-1] = X.charAt(i-1);
      // reduce values of i, j and index
      i--;
      j--;
      index--;
    else if (L[i-1][j] > L[i][j-1])
      i--;
    else
      j--;
  for k = 0 to  temp
    print lcs[k]
```

4.6 Complexity Analysis

- Time Complexity of the above implementation is O(mn) which is much better than the worst-case time complexity of Naive Recursive implementation.
- Base case: c[i ,j]=0 if i=0 or j=0.
- Worse case: x[i] ≠ y[j], in which case the algorithm evaluates two subproblems, each with only one parameter decremented.
- If the inputs are of size X, Y, then the grid will have X × Y entries each of which gets set in constant time. Recovering an LCS from the grid is a linear operation, so overall, this algorithm operates in O(XY) time and requires O(XY) space. If X and Y are similarly sized, we have a quadratic algorithm.

5. Extending Minimum Edit Distance

LCS problem can be shown as a special case of minimum edit distance problem. This relation is seen by noting that for an alignment with no character changes, the only possibilities for pairs in the alignment are the following:

```
...a...     ...a...     ..._...
...a...     ..._...     ...a...
```

Hence, any alignment with no character changes in minimum edit distance problem corresponds to a common subsequence, and any common subsequence corresponds to an alignment (with no character changes). Here is an illustration of such an alignment and its corresponding common subsequence of length four:

```
              _abac_us
_abac_us       |  | ||
ca__ctus      ca__ctus
```

We see that total number of characters is twice the number of pairs of characters (hence, twice the length of the common subsequence) plus the remaining number of alignment columns (each with _ as a member of the column, hence this number is exactly the cost of the alignment). On the other hand, the number of characters is of course m + n. Therefore, for any pair of corresponding alignment (with no character changes) and common subsequence, of costs a and s, respectively, we have 2s + a = m + n. Thus, the maximum possible value of s will be in a pair with the minimum possible value for a. This shows 2lcs(m, n) + edit(m, n) = m + n.

5.1 Pseudocode

```
m = A.length
n = B.length
int lcs = 0
lcs(char[] A, char[] B, m, n)
  L[m+1][n+1]
  for i=0 to m
    for j=0 to n
      if(i == 0)
        L[i][j] = j;
      else if (j == 0)
        L[m][n] = i;
      else if (A[i-1] == B[[j-1])
        L[i][j] = L[i-1][j-1];
      else
        L[i][j] = 1 + min(L[i][j-1], L[i-1][j], L[i-1][j-1]);
  lcs = (m +n - L[m][n])/2;
  return lcs;
```

5.2 Complexity Analysis

- Time Complexity: It has complexity of O(mn) since it follows the same dynamic programming approach
- Auxiliary Space **:** O(mn) space for m$\times n$ matrix.

6. Hirschberg Algorithm

In a 1975, D.S. Hirschberg describes a technique for computing an LCS using space proportional to the length of the inputs. The algorithm combines the dynamic programming technique discussed in the previous section with a classic divide and conquer approach.

The main idea behind the Hirschberg algorithm is to determine the middle point of an LCS "curve" and then, recursively, determine the quartile points.

We then calculate LCS lengths for the pair (xb, ys) and the pair (reversed(xe), reversed(ys)). If we find a y-position, k, which maximizes the sums of these forward and backwards LCS lengths, then we have a suitable position to split ys. Thus our answer reduces to finding the answer for two smaller subsequences.

Assume two strings x and y, where the length of strings are equal without loss of generality. Consider substrings $x_1 = x_{1\ldots n}$, $y_1 = y_{1\ldots m/2}$, $x_2 = x_{n\ldots 1}$, $y_2 = y_{m\ldots m/2}$where the length $|x_1|$ = $|x_2|$ and$|y_1|$ = $|y_2|$. y_1 is a prefix of y of the length m/2. y_2 is a reversed suffix of y of the length m/2 . Then calculate the longest common subsequence for x_1 and y_1, and the longest common subsequence of reversed strings x_2 andy_2.

6.1 Pseudocode

It recursively breaks into smaller sub problems that gives solution and each time it looks for the maximum LCS and not at the entire table which allows it to achieve this.

```
Hirschberg(int m, int n, string A, string B)
  String c = "";
  if (n==0)
    C = "";
  else if (m==1)
    for j = 0 to n
      if (A.charAt(0) == B.chartAt(j))
        add A.charAt(0) to c;
        break;
  else
    i = m/2;
    //solve_lcs computes lcs length as discussed in previous algorithm
    int[] l1 = solve_lcs(i, n, A.substring(0,i), B) //
    int[] l2 = solve_lcs(m-i, n, reverseString(A.substring(i)),reverseString(B));
    // findk method finds the index of the maximum sum of L1 & L2,as described
    by Hirschberg//
    int k = findk(l1, l2, n);
    String c1 = Hirschberg(i, k, A.substring(0, i), B.substring(0, k));
    String c2 = Hirschberg (m-i, n-k, A.substring(i), B.substring(k));
    c = c1 + c2;
  return c;
```

6.2 Complexity Analysis

- This algorithm of calculating LCS is quadratic in time because of the recursive calls. The divide and conquer methodology make the algorithm to have linear space.
- Searching phase in O(mn) time complexity.
- Searching phase in O(m+n) space complexity.

- In this method we can track the LCS string while keeping the space complexity at O(m+n).

7. Results and Analysis

7.1 Naïve Recursion Performance:

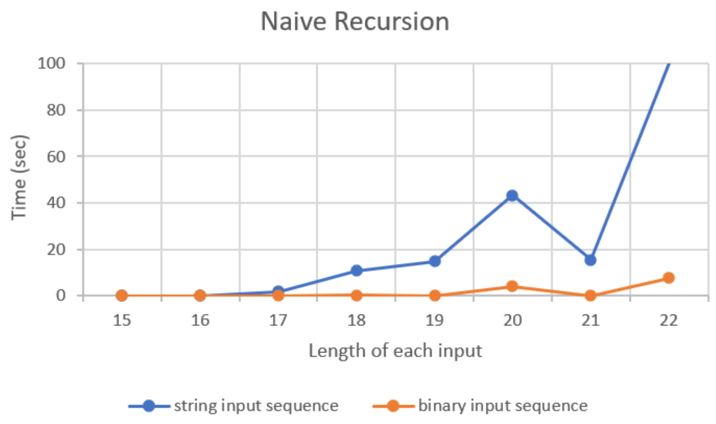
String Length	LCS length	Time Taken(s)
15	8	0.014
16	8	0.035
17	6	1.78
18	6	10.77
19	15	14.78
20	6	43.25
21	17	15.56
22	11	100.48

(Table 1)

Binary InputLength	LCS Length	Time Taken (s)
15	10	0.006
16	8	0.015
17	6	0.091
18	6	0.32
19	15	0.011
20	6	4.12
21	17	0.019
22	11	7.603

(Table 2)

It is evident from table 1 values that the naïve recursion performs worst as the string length increases and LCS length decreases. For large strings it takes exponentially very large time and it is not preferred in real time world. In table 2, I have taken another interesting set of binary sequence inputs and the results were not bad when compared to table 1. However, naïve recursive is not good for calculating LCS.



7.2 Minimum Edit Distance:

We can calculate LCS using minimum edit distance when it only deals with insertions and deletions. In table 3 shows the results when we give such an alignment with possible insertions and deletions with no substitution while Table 4 shows the result when different input alignment is given. In this case, I have taken string 1 of length 50000 and string 2 of length 1000 just to show the correctness of the algorithm

Table 3:
Alignment with no character changes

Algorithm	**LCS Length**
DP LCS (tabulation)	1000
Extended Min Edit	1000

Table 4:
Alignment with character changes

Algorithm	**LCS Length**
DP LCS (tabulation)	1000
Extended Min Edit	992

7.3 Memoized LCS Performance

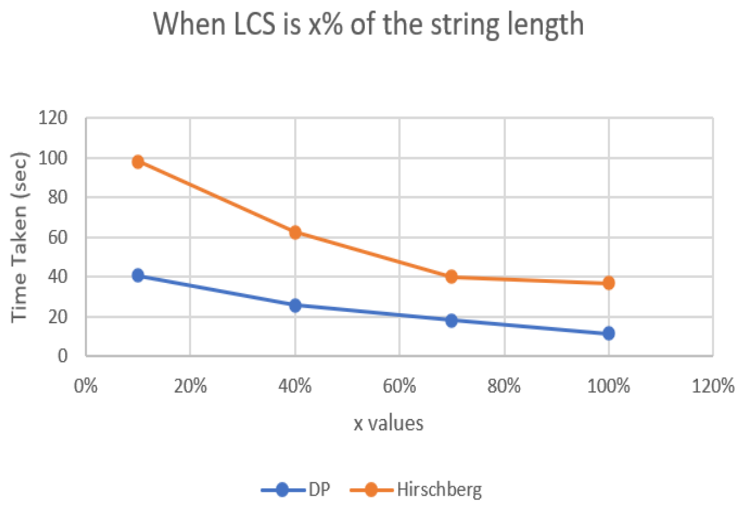
Below table shows the result of time consumption to calculate LCS using memoization, tabulation, Hirschberg with different inputs length. It is interesting that memoization is able to perform only up to string length of 3500(each string length) and later it stops performing as the input size increases, since there is so much overhead that comes with recursion—each recursive call requires that we keep the entire recursion tree in memory. However, tabulation and Hirschberg continue to perform for very large input strings (in millions).

The memoized version of LCS is thus reminiscent of earlier performance, but still strongly recursive. I've tried and couldn't measure to compare with string length greater than 3500. It's performance is better than naïve recursion for small inputs but fails to perform for large inputs. We might try to increase the recursion limit, but this approach is neither clean nor portable because setting an highest possible limit is platform-dependent.

Length of each string	**Time (sec) taken by different LCS algorithms**		
	Memoization	**DP (Tabulation)**	**Hirschberg**
2500	0.20	0.067	0.092
3500	0.56	0.089	0.27
4500	-	0.29	0.34
5500	-	0.40	0.52
6500	-	0.47	0.67

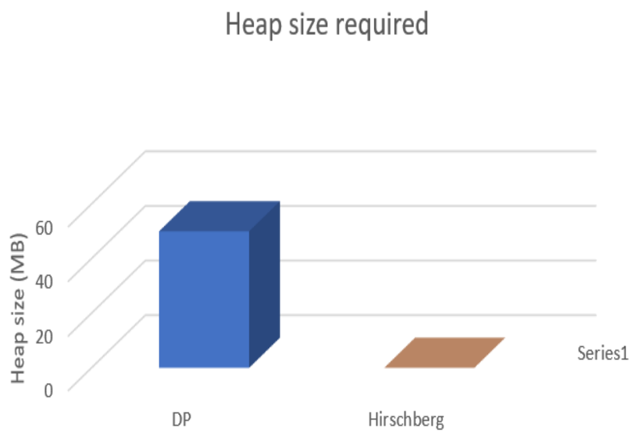
7.4 DP (tabulation)vs HIRSCHBERG with different lcs %

LCS %	DP- Time(sec)	Hirschberg-Time(sec)
10%	40.63	98.21
40%	25.87	62.75
70%	18.15	40.13
100%	11.47	36.94



In the above graph, I ran the algorithms with each string length of 60000 (took a book with many different characters) and evaluated the time consumption from worst case to best case. It is evident that dynamic programming has faster performance than Hirschberg algorithm. Coming to Hirschberg algorithm, although the space requirement is proportional to the input sizes, it continually creates new list slices in order to get the job done. It is capable of handling large inputs in using a limited amount of memory, but it takes far too long doing so.

Apart from taking large text files as my input, I have also taken another interesting DNA sequence file where each sequence comprises of only 4 alphabets (A, C, G, T) and compared to DNA sequences to calculate LCS. This kind of scenario happens in real time world to find the similarity between two DNA sequences.



To run the above algorithms with extremely large input size, I needed to increase my desktop heap size upto 4000MB. Dynamic Programming required more heap size as the size of the strings got increased to a large number. To run the algorithm with input size of 30000 and above, I needed to increase my heap size greater than the default value. Calculating m $\times n$ matrix in dynamic programming for large input strings requires relatively large amount space. We can prefer space optimized dynamic programming solution where we use an array instead of a matrix unless we don't focus on the actual LCS string. Whereas, the heap size required for Hirschberg is a way lot smaller that dynamic programming. For small inputs when I used 40MB for dynamic programming, Hirschberg required only 10kb.

8. Conclusion

As far as comparisons between the various algorithms are concerned, dynamic programming has better performance in terms of time complexity. Hirschberg proves the correctness of the algorithm and goes on to prove that it has quadratic performance but requires just a linear amount of space. From the results, DP_LCS tabulation method proves to be the most efficient algorithm which can compute LCS in a faster and more efficient way when compared to other algorithms. In addition, it is also shown that tabulation outperforms memoization to calculate LCS for large strings. But with one catch while dynamic programming needs execution heap size of more that 3000MB while the Hirschberg was able to compute with small amount of heap size. However, Hirschberg roughly required twice the time as opposed to dynamic programming. Moreover, in any scenario complexities depend on the length of the input string and the length of the LCS string. In the future, I would like to experiment with some more advanced and complex algorithms which yields linear time complexity and contrast their performance from the traditional algorithms discussed in this project.

9. References

[1] CLRS Chapter -15.4

[2] https://pdfs.semanticscholar.org/b700/48e06e1b0805838f9353e9e323fe632a0bfe.pdf

[3] Indu, Prerna: A Comparative Study of Different LongestCommon Subsequence Algorithms

[4] https://www.researchgate.net/publication/260085556_A_COMPARATIVE_STUDY_OF_VARIOUS_PARALLEL_LONGEST_COMMON_SUBSEQUENCE_LCS_ALGORITHMS

[5] http://par.cse.nsysu.edu.tw/~lcs/Introduction.php

[6] https://imada.sdu.dk/~rolf/Edu/DM823/E16/Hirschberg.pdf

[7] https://imada.sdu.dk/~rolf/Edu/DM823/E16/EditDistance.pdf