Longest Common Substring

CS:255 Design and Analysis of Algorithms
Computer Science Department
San Jose State University
San Jose, CA-95192

Project Guide:
Dr Sami Khuri

Team Members:
Avinash Anantharamu SJSU #:008629907
Akash Patel SJSU #:008638799
# Table of Contents

1. Background.......................................................................................................................... 02  
2. Problem Definition ........................................................................................................... 03  
3. Approaches to Solve LCS................................................................................................. 04  
4. Directions for Future Research ....................................................................................... 11  
5. Conclusion ........................................................................................................................ 13  
6. Performance Analysis ...................................................................................................... 14  
7. User Guide ........................................................................................................................ 17  
8. Description of Data Sources ........................................................................................... 21  
9. References ....................................................................................................................... 22
1. Background

The longest common substring problem is a classic problem in string analysis. In 1970 the famous computer scientist Donald E. Knuth conjectured that a linear time algorithm for this problem would be impossible. To the retrieval of his honor, it should be stressed that the first linear time construction algorithm of suffix trees became known years after he made the conjecture. In fact, given that knowledge of suffix trees, a solution to the longest common substring problem is almost trivial.

Given m strings \( T^{(1)} , T^{(2)} , \ldots , T^{(m)} \) of total length n, a simple \( O(m \cdot n) \) time solution to the k-common substring problem can be obtained by a bottom-up traversal of the generalized suffix tree \( T \) of \( T^{(1)} , T^{(2)} , \ldots , T^{(m)} \), k-common repeated substring problem. This problem can also be solved in \( O(m \cdot n) \) time by a bottom-up traversal of the generalized suffix tree of \( T^{(1)} , T^{(2)} , \ldots , T^{(m)} \), but our goal is to solve it in optimal \( O(n) \) time.

In its simplest form, the longest common substring problem is to find a longest substring common to two or multiple strings. Using Ukkonen suffix trees, this problem can be solved in linear time and space.

The aim of our project is to find an optimal solution, for a large input dataset containing Alphabets and compare its performance with other approaches to search the longest common sub-string which occurs in all the input sequence.
2. Problem Definition

Given two or more strings, $S$ of length $m$ and $T$ of length $n$, find the longest strings which are substrings of both $S$ and $T$.

A generalization is the $k$-common substring problem. Given the set of strings $S = \{S_1, S_2, \ldots, S_k\}$. Here $|S_i| = n_i$ and $\Sigma n_i = N$. Find for each $2 \leq k \leq K$, the longest strings which occur as substrings of all $k$ strings.

### Input

<table>
<thead>
<tr>
<th>AGCTGCTAAGCTGCTAAGCTGCTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGCTGCTATGAGCTGCTATGACAT</td>
</tr>
<tr>
<td>AGCTGCTAAGCTGCTAAGCTGCTA</td>
</tr>
<tr>
<td>AGCTGCTAAGCTGCTAAGCTGCTA</td>
</tr>
</tbody>
</table>

### Longest Common Substring

<table>
<thead>
<tr>
<th>AGCTGCTAAGCTGCTAAGCTGCTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGCTGCTAAGCTGCTAAGCTGCTA</td>
</tr>
<tr>
<td>AGCTGCTATGAGCTGCTATGACAT</td>
</tr>
<tr>
<td>AGCTGCTAAGCTGCTAAGCTGCTA</td>
</tr>
<tr>
<td>AGCTGCTAAGCTGCTAAGCTGCTA</td>
</tr>
</tbody>
</table>
3. Approaches to Solve LCS

1 Brute Force

In Computer Science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem’s statement.

Working

```
ABABCA
BABCA
ABCBA
LCS := ABC
```

Pseudo Code

Function bruteforce_lcs(String input1, String input2)
begin
    for y:0 to input1.length()
        do
            for z = 0 to input2.length()
                do
                    counter = 0;
                    while input1.charAt(y + counter) = input2.charAt(z + counter)
                        do
                            counter++;
                        end while
                    if (((y + counter) >= input1.length()) || ((z + counter) >= input2.length()))
                        do
                            break;
                        end if
                    else if counter > longest_common_substring
                        do
                            longest_common_substring = counter;
                            start = y;
                        end if
                end if
        end for
end function
Approaches to Solve LCS

Time Complexity

Brute Force Technique to compute Longest common substrings of any length is trail and error method as we try all possible combinations before concluding as which substring among both the strings is the largest common substring.

To Compute Basic Operations

```
for y:0 to input1.length()
    for z = 0 to input2.length()
        while input1.charAt(y + counter) = input2.charAt(z + counter)
```

Assuming length of both the strings to be \( n \)  
Number of Basic Operations

\[
\begin{align*}
\sum_{y=0}^{n-1} \sum_{z=0}^{n-1} \sum_{k=0}^{n-1} 1 \\
\sum_{y=0}^{n-1} \sum_{z=0}^{n-1} n \\
\sum_{y=0}^{n-1} n \times n
\end{align*}
\]

Total Number of Basic Operations = \( n^*n^*n = n^3 \)

Time Complexity = \( O(n^3) \)

2 Dynamic Programming

Dynamic Programming is a generalization of Divide and Conquer. It is based on principle of optimality. Dynamic Programming is a bottom up method that checks all possible sub-problems once and finds optimal solution.

Recurrence Relation

\[
\text{LCS}(S[1...m], T[1...n]) =
\begin{cases}
\text{LCS}(S[1...m-1], T[1...n-1]) + 1 & \text{if } S[m] = T[n] \\
0 & \text{Otherwise}
\end{cases}
\]
Approaches to Solve LCS

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Longest Common Substring obtained from backtracking

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

Pseudo Code

function LCS(S[1...m],T[1...n])
begin
L:=array(1..m,1..n)
z:=0
ret:={ }
for i=1 to m
do
  for j=1 to n
do
    if S[i]==T[j]
do
      if i==1 or j==1
do
        L[i,j]=1;
      end if
else
  do
    L[i,j]=L[i-1,j-1]+1;;
  if L[i,j]>z
do
    z:=L[i,j]
Approaches to Solve LCS

\[ \text{ret} := \{S[i-z+1..i]\} \]

\[ \text{end if} \]

\[ \text{else if } L[i,j] = z \]

\[ \text{ret} := \text{ret UNION } \{S[i-z+1..i]\} \]

\[ \text{end if} \]

\[ \text{end if} \]

\[ \text{else} \]

\[ \text{do} \]

\[ L[i,j] = 0 \]

\[ \text{end if} \]

\[ \text{end for} \]

\[ \text{return ret} \]

\[ \text{end} \]

**Time Complexity**

Dynamic Programming Approach to compute Longest common substrings of any length requires table to be constructed which of the the size \((n+1) * (m+1)\) before we backtrack to arrive at possible solutions.

**To Compute Basic Operations**

\[ \text{for } x = 1 \text{ to } n \]

\[ \text{for } y = 1 \text{ to } m \]

\[ \sum_{x=0}^{x-n} \sum_{y=0}^{y-m} 1 \]

\[ \sum_{x=0}^{x-n} m \]

**Total Number of Basic Operations** = \(n*m = n*m\)

**Time Complexity =** \(O(n*m)\)

**3 Suffix Tree**

- A suffix tree \(ST\) for an \(m\)-character string \(S\) is a rooted directed tree with exactly \(m\) Leaves numbered 1 to \(m\)
Approaches to Solve LCS

- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of $S$.
- No two edges out of a node can have edge-labels beginning with the same character.
- The key feature of the suffix tree is that for any leaf $i$, the concatenation of the edge labels on the path from the root to the leaf $i$ exactly spells out the suffix of $S$ that starts at position $i$.
- A tree like data structure for solving problems involving strings.
- Allow the storage of all substrings of a given string in linear space.
- Simple algorithm to solve string pattern matching problem in linear time.
Approaches to Solve LCS

Longest Common Sub-String

<table>
<thead>
<tr>
<th>'ab'</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bc'</td>
</tr>
</tbody>
</table>

Pseudo Code

Ukkonen Algorithim

Pseudo code for Ukk:
Construct tree \( T_1 \).
for \( i = 1 \) to \( m-1 \) do
begin {phase \( i+1 \)}
    for \( j = 1 \) to \( i+1 \) do
        begin {extension \( j \)}
            In the current tree find the end of the path from the root labeled \( t[...j] \). If necessary, extend that path by adding character \( t[i+1] \), thus ensuring that string \( t[i...i+1] \) is in the tree.
        end;
    end;
end;

SuffixTree::update(char* s, int len)
begin
...
    int i;
    int j;
...
    for \( j = 0 \), \( i = 0 \) to len
do
        while \( j <= i \)
Approaches to Solve LCS


do

... all the work ...

end while

end for

end

Time Complexity

By combining all of the speed-ups, we can now construct a suffix tree representing the string \( S[1..m] \) in \( O(m) \) time and in \( O(m) \) space. i.e linear time within linear space.

Time Complexity Comparison

<table>
<thead>
<tr>
<th>Approach</th>
<th>Worst Case Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>( O(n^3) )</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>( O(mn) )</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>( O(n) )</td>
</tr>
</tbody>
</table>
4. Directions for future research

1. The experimental results described in the previous chapters have shown the time required to compute largest common substring has been improving with time from $O(n^3)$ using traditional Brute force approach to $O(mn)$ using Dynamic Programming to $O(n \log n)$ using Suffix arrays which uses indexing to $O(n)$ using Ukkonen Suffix Tree construction technique.

2. Analyze the input characteristic and determine which approaches needs to use.

3. In Dynamic programming following tricks can be used to reduce the memory usage of an implementation:-

   - Keep only the last and current row of the Dynamic Programming table to save memory $O(\min(m, n))$ instead of $O(nm)$.

   - Store only non-zero values in the rows. This can be done using hash tables instead of arrays. This is useful for large alphabets.

4. Implement longest common substring problem using McCreight and Weiner, different Hashing techniques such as roller hash in conjunction with above techniques to aim to see if there could be any improvement in time complexity and reduce basic operations from current levels.

5. Look at problems that can be solved using Fast Exact Algorithms (Heuristic) for the Closest String and Substring Problems. The objective is to compute a string $s^1$ of length $L$ such that the Hamming distance $d(s^1, s)$ between $s^1$ and each string in $S$ is at most $d$. Such a string $s^1$ is called a center string of the given strings. Of course, center strings may not exist. In that case, an algorithm for the problem should output a special symbol to indicate this fact. Consensus of a protein family All these applications share a task that requires the design of a
Directions for future research

new DNA or protein string that is very similar to (a substring of) each of the given strings.

6. The closest string and substring problems have wide applications in computational biology. This has motivated researchers to come up with heuristics without performance guarantee.
5. Conclusion

We have implemented Brute force, Dynamic Programming and suffix tree approaches to find the longest common substring. Most of the results were in line with the standard running time of the algorithms as indicated in table below.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Basic Operations</th>
<th>Execution Time (mille seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(ms)</td>
<td>Brute Force</td>
<td>( n^3 )</td>
</tr>
<tr>
<td></td>
<td>Dynamic Programming</td>
<td>( m \times n )</td>
</tr>
<tr>
<td></td>
<td>Suffix Tree</td>
<td>( n )</td>
</tr>
</tbody>
</table>

Results were measured on Intel Core i-7 2.00 GHZ processor 4GB Ram System

<table>
<thead>
<tr>
<th>Approach</th>
<th>Basic Operations</th>
<th>Execution Time (mille seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(ms)</td>
<td>Brute Force</td>
<td>( n^3 )</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>( m \times n )</td>
<td>120</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>( n )</td>
<td>60</td>
</tr>
</tbody>
</table>

1000 Inputs

10000 Inputs
6. Performance Analysis

The tests were performed on 1000 strings and 10000 strings using brute force, dynamic programming and suffix tree and it was seen suffix tree had best performance as it achieved a time complexity $O(n)$ and space complexity $O(n)$. This was followed by Dynamic Programming as it achieved a time complexity $O(m*n)$ and space complexity $O(n*n)$ followed by Brute force as it achieved a time complexity $O(n*n*n)$ and space complexity $O(n*n*n)$.

**Results were Measured on Intel Core i-7 2.00 GHZ processor 4GB Ram System**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Basic Operations</th>
<th>Time Complexity</th>
<th>Execution Time (mille seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>$n^3$</td>
<td></td>
<td>129</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>$m*n$</td>
<td></td>
<td>68</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>$n$</td>
<td></td>
<td>29</td>
</tr>
</tbody>
</table>

**1000 Inputs**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Basic Operations</th>
<th>Time Complexity</th>
<th>Execution Time (mille seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute Force</td>
<td>$n^3$</td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>$m*n$</td>
<td></td>
<td>120</td>
</tr>
<tr>
<td>Suffix Tree</td>
<td>$n$</td>
<td></td>
<td>60</td>
</tr>
</tbody>
</table>

**10000 Inputs**

The experimental results are in line with theoretical estimated values as shown in graph below.
7. User Guide

Please follow the below steps to run the program

1) Copy the package “AlgoProject” and paste it in the c:/d: drive

2) Add all the Jar files of the package to the class path

3) Refer: [http://docs.oracle.com/javase/1.3/docs/tooldocs/win32/classpath.html](http://docs.oracle.com/javase/1.3/docs/tooldocs/win32/classpath.html) to add the jars to the class path

4) Open the command prompt

5) Navigate to the package “AlgoProject”

6) To compile all files: javac *.java

7) Type Java MainApplet

1. Main Screen Snapshot:
2. Brute Force Screen Snapshot:

![Brute Force Screen Snapshot]

3. Dynamic Programming Screen Snapshot:

![Dynamic Programming Screen Snapshot]
4. Suffix Tree
5. Comparison Screenshot

![Comparison Screenshot 1](image1)

![Comparison Screenshot 2](image2)
Description of Data Sources

To perform the time and performance analyses presented in above chapters we used data from a range of sources, including:

- Pre Initialized random data for large data set to measure time complexity of 1000 and 10000 input using three techniques discussed above.
- Wikipedia where recurrence relation used for computing table in dynamic programming and backtracking was used.
- Esko Ukkonen’s suffix tree algorithm in linear time and linear space.
- Brute force was designed by trial and error method manually so we got desired result and all possible combinations were exhausted before displaying result.
- We have also made use of Mark Nelson’s algorithm for fast suffix tree operations in C++.
- Java Complete Reference 7th edition was used to implement applets.
References

- Ukkonen’s Algorithm: http://en.wikipedia.org/wiki/Ukkonen's_algorithm
- Suffix tree and suffix array techniques for pattern analysis in strings http://www.cs.helsinki.fi/u/ukkonen/
- McCreight's Algorithm www.cs.helsinki.fi/u/tpkarkka/opetus/11s/spa/lecture09-4up.ps
- Data Structures And Algorithms In Java By Adam Drozdek
- Introduction to Algorithms 3rd Edition by Cormen, Leiserson, Rivest, and Stein
- Keep Learning http://illya-keeplearning.blogspot.com