String Matching and Processing Algorithms

Dr. Natarajan Meghanathan
Associate Professor of Computer Science
Jackson State University, Jackson, MS
E-mail: natarajan.meghanathan@jsums.edu
Finding Longest Suffix Prefix Overlap
Longest Suffix Prefix Overlap

• Problem Statement: Given two strings $S$ and $P$ ($S$ – candidate suffix string and $P$ – candidate prefix string) we want to find the longest suffix of $S$ that completely overlaps with the prefix of $P$. We will call such a string the overlap string.

• Example:
  – $S = \text{“ACTAGGA”}$  \hspace{1cm} $P = \text{“GGACTAGTA”}$
  – The answer is \text{“GGA”}

• Basic Idea:
  – Compare the characters of the two strings in the inverse direction (the last character of the suffix string compared with the first character of the prefix string; the last two characters of the suffix string compared with the first two characters of the prefix string, etc) until the longest overlapping string match is found or no more comparisons are possible.
    • Keep track of the longest overlap string found as we compare the characters in the inverse direction.
    • When no more comparisons are possible, return the overlap string found so far.
Initial Processing of the Two Strings

- The length of the longest overlap cannot exceed the minimum of the length of the two strings.
- If the length of S exceeds P, then we trim the initial portion of S that would not be needed for any comparison.
- On the other hand, if the length of P exceeds S, we trim the terminal (later) portion of P that would not be needed for any comparison.
- The final result would not be impacted due to this trimming.
  - For example, if S = “ACTAGGA” of length 7 and P = “GGACT” of length 5, the longest overlap will be of length at most 5 that corresponds to the smaller string P. Hence, we trim the candidate suffix string S by removing the initial length(S) – length(P) = 7-5 = 2 characters. The trimmed S = “TAGGA”.
  - Likewise if S = “ACTAGGA” of length 7 and P = “GGACTAGTA” of length 9, the longest overlap will be of length at most 7 that corresponds to the smaller string S. Hence, we trim the candidate prefix string P by removing the terminal length(P) – length(S) = 9–7 = 2 characters. The trimmed P = “GGACTAG”
Example 1

S = ACTAGGA   P = GGACTAGTA

After trimming,

S = ACTAGGA
P = GGACTAG

Compare in the inverse order until an initial match is found or the entire length of the two strings is compared.

ACTAGGA
  GGA

ACTAGGAG
  GGA

ACTAGGAG
  GGA

ACTAGGAG
  GGA

ACTAGGAG
  GGA

ACTAGGAG
  GGA

Overlap String found so far: GGA

Longest Suffix Prefix Overlap
Example 2

$S = TACCAGCC$     $P = CCCAGCC$

After trimming,

$S = ACCAGCC$
$P = CCCAGGC$

**Comparisons**

- **ACCAGGC**  
  - Overlap String: **C**
  - After trimming: **ACCAGCC**

- **ACCAGCC**  
  - Overlap String: **CC**
  - After trimming: **ACCAGGC**

- **ACCAGGC**  
  - Overlap String: **CC**
  - After trimming: **ACCAGCC**

- **ACCAGGC**  
  - Overlap String: **CC**
  - After trimming: **ACCAGGC**

- **ACCAGGC**  
  - Overlap String: **CC**
  - After trimming: **ACCAGGC**

Return the overlap string found so far: **CC**

Longest Suffix Prefix Overlap

$TACCAGGC$  
$CCCAGGC$  
**Overlap String: CC**
Example 3

S = GAAATCAAA P = AAATCAAA

After trimming,
S = AAATCAAA P = AAATCAAA

Comparisons

<table>
<thead>
<tr>
<th>S</th>
<th>P</th>
<th>Overlap String</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAATCAAA</td>
<td>AAATCAAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAATCA</td>
<td>AAATCAAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAATCA</td>
<td>AAATCAAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAATCA</td>
<td>AAATCAAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAATCA</td>
<td>AAATCAAA</td>
<td>AAA</td>
</tr>
<tr>
<td>AAATCA</td>
<td>AAATCAAA</td>
<td>AAA</td>
</tr>
</tbody>
</table>

Return the overlap string found so far: AAATCAAA

Longest Suffix Prefix Overlap

AAATCAAA AAATCAAA
Algorithm *FindLongestOverlap*

**Inputs:** String `SuffixString`, String `PrefixString`

**Begin** *FindLongestOverlap*

// Initial processing
if length(`SuffixString`) > length(`PrefixString`) then
    `SuffixString` = `SuffixString`[length(`SuffixString`)-length(`PrefixString`), length(`SuffixString`)]
else if length(`PrefixString`) > length(`SuffixString`) then
    `PrefixString` = `PrefixString`[1, length(`SuffixString`)]
end if

*String* `overlapString` = ""
*int* `suffixIndex` = length(`SuffixString`);
*int* `prefixIndex` = 1;

while (`prefixIndex` ≤ length(`PrefixString`) AND `suffixIndex` > 0) do
    boolean `comparisonResult` = *CompareStrings*(`SuffixString`, `PrefixString`, `suffixIndex`);
    if (`comparisonResult` equals *true*) then
        `overlapString` = `SuffixString`[`suffixIndex`, length(`SuffixString`)];
    end if
    `prefixIndex` = `prefixIndex` + 1;
    `suffixIndex` = `suffixIndex` - 1;
endWhile

return `overlapString`

**End** *FindLongestOverlap*
Subroutine `CompareStrings(String S, String P, int sufIndex)`

```plaintext
int prefIndex = 1;
while (sufIndex ≤ length(S)) do
    if (S[sufIndex] != P[prefIndex]) then
        return false;
    end if
    prefIndex = prefIndex + 1
    sufIndex = sufIndex + 1
endWhile
return true;
// match observed for every character in S[sufIndex...length(S)]
// with the prefix of P, starting from index 1
```

**Time Complexity Analysis of Algorithm FindLongestOverlap**

Let $L = \text{Minimum (length of the two input strings)}$. We basically make $1, 2, 3, \ldots, L$ comparisons. The number of comparisons is thus: $1 + 2 + 3 + \ldots + L = \frac{L(L-1)}{2} = \mathcal{O}(L^2)$. This is the time-complexity of the algorithm.

**Application of Suffix Prefix Longest Overlap Algorithm:**

Useful in applications such as Genome assembly where significant suffix-prefix overlaps between pairs of DNA fragments are used to construct an optimal larger sequence (Shortest Common Superstring Problem)
Shortest Superstring Problem (SSP)

- **Problem:** Given a set of strings, find a shortest string that contains all of them.
- **Input:** Strings $s_1, s_2, \ldots, s_n$.
- **Output:** A string $s$ that contains all strings $s_1, s_2, \ldots, s_n$ as substrings such that the length of $s$ is minimized.
- **Complexity:** NP-complete. We will use a Traveling Salesman Problem (TSP-based) heuristic.
  - TSP: Given a directed complete graph (directed and weighted edges between any two pair of vertices), we will find a minimum weight path that traverses all the vertices exactly once.
- **Overlap:** Define overlap $(s_i, s_j)$ as the length of the longest prefix of $s_j$ that matches a suffix of $s_i$.

```
aaaggcatcaaatctaaaggcatcaaa
```

```
aaaggcatcaaatctaaaggcatcaat
```

Longest overlap

```
aaaggcatcaaatctaaaggcatcaat
```
SSP to TSP Problem

• Construct a complete directed graph with n vertices
  – Each vertex corresponds to one of the n strings
  – The weight of an edge between two vertices \( s_i \) and \( s_j \) is the negative of the length of the longest overlap comprising of the suffix from string \( s_i \) and the prefix from string \( s_j \):
    • For example, if the length of the longest overlap of two strings suffix of \( s_i \) and prefix of \( s_j \) is 3; then the weight of the directed edge \( s_i \rightarrow s_j \) is -3.
    • If there is no overlap of the suffix of \( s_i \) with the prefix of \( s_j \), then the weight of the directed edge \( s_i \rightarrow s_j \) is 0.

• Heuristic for the TSP problem on a directed complete graph
  – We will find a TSP path starting from each vertex. For every such starting vertex \( s_a \), we execute the following steps:
    • From \( s_a \), visit the vertex \( s_k \) whose edge weight is the minimum. From \( s_k \), visit the unvisited neighbor with whom the edge weight is the minimum. If there is a tie, choose the vertex with the smallest ID (ties are broken in numerical or lexicographic order).
    • Repeat the above process until we find a path traversing all the vertices. Note that we will stop with a TSP path.
  – Among the TSP paths found with each vertex as the starting vertex, we will choose the TSP path with the total minimum weight and extract the superstring based on the sequence of vertices (strings) visited.
Example 1: SSP to TSP Problem

f1 = ACGTA
f2 = CTTGA
f3 = ACTT
f4 = GTAAC

Path with the minimum total weight
ACGTAACCTTGA
Length of the superstring: 11

Path with the maximum total weight
GTAACGTACTTGA
Length of the superstring: 13
Example 2: SSP to TSP Problem

f1 = ACA
f2 = CTTGA
f3 = ACTT
f4 = AAC

Path with the minimum total weight

AACACTTGA

Length of the superstring: 9
Sequencing by Hybridization
Euler Path

• An Euler path is a path that visits all the edges exactly once.
• A directed graph has an Eulerian path if and only if at most one vertex has \((\text{out-degree}) - (\text{in-degree}) = 1\), at most one vertex has \((\text{in-degree}) - (\text{out-degree}) = 1\), every other vertex has equal in-degree and out-degree, and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph.

• Algorithm to find Euler path:
  – Start with the vertex whose out-degree is larger than the in-degree. If the graph has none, start with an arbitrary one.
  – If there exists a choice to take an out-going edge (that is not yet traversed and that would not lead to a path of no-return), choose the neighbor vertex with the lowest ID to visit.
  – As each vertex (other than at most 2 vertices) have equal in-degree and out-degree, every time you enter one of these vertices through an edge, you will have a corresponding edge to leave the vertex.
  – The two vertices, with a unit difference between the in-degree and out-degree, if at all exists, could be the starting and ending vertices of the Eulerian path.
  – Repeat the above process until all the edges are traversed exactly once.
Example: Euler Path (1)

The above graph has V4 with Out-degree – in-degree = 1 and V3 with In-degree – Out-degree = 1. For every other vertex, the # in-coming edges is equal to the # out-going edges. Note that 0 is also considered even.
Note that V3 is not visited now from V6 as V3 is supposed to be the ending vertex of the Eulerian path.
Sequencing by Hybridization (SBH)

- The overall goal is to do genome sequencing; i.e., to identify the sequence of nucleotides constituting the DNA strands.
- However, the strands are so long that it is not possible to elucidate them in one go.
- Hence, we break the DNA strand (of n nucleotides) into several fragments (l-length strings) of smaller size.
- We use DNA microarrays to identify each of these fragments and stitch them together.
- A DNA array is a library of all possible l-length DNA fragments (called probes or oligonucleotides) on the 4-bases. Hence, there will be 4^l probes in a DNA array.
- We will apply the solution containing the DNA fragment to the DNA array. The DNA fragment will hybridize (attach) to a complementary oligonucleotide probe sequence in the array. This will lead us to infer the nucleotide sequence of the DNA fragment.
- We will identify the l-length nucleotide sequence of several such DNA fragments and then put together the sub strings to infer the nucleotide sequence of the original DNA segment of length n.
Hybridization on DNA Array

<table>
<thead>
<tr>
<th></th>
<th>AA</th>
<th>AT</th>
<th>AG</th>
<th>AC</th>
<th>TA</th>
<th>TT</th>
<th>TG</th>
<th>TC</th>
<th>GA</th>
<th>GT</th>
<th>GG</th>
<th>GC</th>
<th>CA</th>
<th>CT</th>
<th>CG</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DNA target to be identified: **TATCCGTTT**

It binds to its complement fragments that when stitched together will constitute the sequence **ATAGGCAAAA**

Probes identified: **ATAG TAGG AGGC GGCA GCAA CAAA**
/-mer Composition

• Spectrum \( (S, l) \): unordered multiset of all possible \((n-l+1)\) \(l\)-mers in a string \(S\) of length \(n\).
  – For example, if \(n = 8\) and \(l = 3\), we have a spectrum of \(8-3+1 = 6\), 3-mers.

• The order of the individual elements in \(\text{Spectrum}(S, l)\) does not matter.

• For \(S = \text{TATGGTGC}\), all of the following are equivalent representations of \(\text{Spectrum}(S, 3)\):
  – \{TAT, ATG, TGG, GGT, GTG, TGC\}
  – \{ATG, GGT, GTG, TAT, TGC, TGG\}
  – \{TGG, TGC, TAT, GTG, GGT, ATG\}

• Different sequences may have the same spectrum:
  – \(\text{Spectrum}(\text{GTATCT}, 2)\) and \(\text{Spectrum}(\text{GTCTAT}, 2)\) could be \{AT, CT, GT, TA, TC\}
SBH Problem

• Goal: Reconstruct a string from its $l$-mer composition.
• Input: All $l$-mers of an unknown string $S$.
• Output: String $S$ such that Spectrum $(S, l) = \text{the given } l\text{-mers}.$
• We construct a directed graph as follows:
  – For each $l$-mer, construct two vertices (if one or both of them are not already present) whose labels correspond to the first $l-1$ characters and the last $l-1$ characters. The edge is labeled with the $l$-mer. The edges are directed.
• Run the algorithm to determine an Eulerian path and deduce the composite string $S$ based on the edges traversed in the Eulerian path.
Given 3-mers
ACT
CTC
CTG
TCT
TGG

The above graph is guaranteed to have an Eulerian path, because there are two vertices (AC with Outdegree – Indegree = 1 and GG with Indegree – Outdegree = 1). The rest of the vertices CT, TC and TG have the # in-coming edges equal to the # out-going edges.

AC and GG will respectively be the starting and ending vertices.

Taking the edge CT \to TG before visiting TC would lead to a path of no-return (i.e., CT \to TG \to GG from which TC and the two edges CTC and TCT can never be visited). Hence, we do not visit TG from CT first, instead visit TC from CT.

AC \to CT \to TC \to CT \to TG \to GG

ACTCTGG
Clearly satisfies the requirements for an Eulerian path. Start at AT and end at CC by traversing the chain of vertices:

AT → TG → GC → CA → AG → GG → GT → TC → CC

ATGCAGGGTCC
SBH Example 3

Possible solutions

AT → TC → CT → TG → GA → AT → TG
ATCTGATG

AT → TG → GA → AT → TC → CT → TG
ATGATCTG

GAT
TGA
ATC
TCT
ATG
CTG
SBH Example 4

ATAG   GCAA
GGCA   TAGG
AGGC   CAAA

ATA → TAG → AGG → GGC → GCA
→ CAA → AAA

ATAGGCAAAA