Inexact Matching, Alignment

See Gusfield, Chapter 9
Dasgupta et al. Chapter 6 (Dynamic Programming)
Outline

- Yet more applications of generalized suffix trees, when combined with a least common ancestor algorithm
  - Exact matching with wildcards
  - Matching with k mismatches
- Inexact matching problems and alignment problems
Recall: A Suffix Tree for Two Strings

Example: $S_1 = xabxa$ and $S_2 = babxba$
Recall: Longest Common Substring

Linear time algorithm: Build the generalized suffix tree, then find the node with greatest string depth that is on a path to a leaf of both trees
Recall: Longest Common Substring

Linear time algorithm: Build the generalized suffix tree, then find the node with greatest string depth that is on a path to a leaf of both trees

Example: $S_1 = xabxa$ and $S_2 = babxba$
The string label of the lowest common ancestor of leaves 1,i and 2,j is the longest common prefix of the suffices $S_1[i..]$ and $S_2[j..]$
Longest Common Extension

The string label is called the *longest common extension* of \( S_1[i..] \) and \( S_2[j..] \); we’ll denote it by \( S1-S2-LCEextension(i,j) \).
Longest Common Extension

The string label is called the _longest common extension_ of $S_1[i..]$ and $S_2[j..]$; we’ll denote it by $S1-S2-LCExtension(i,j)$

$S1-S2-LCExtension(2,2)$ is $abx$
Longest Common Extension

- For $S_1 = xabxa$, $S_2 = babxba$, what is $S1-S2-LCExtension(1,4)$?
There is a linear-time algorithm for pre-processing a rooted tree, after which the lowest common ancestor of any two nodes can be found in constant time.*

[Harel and Tarjan] [Schieber and Vishkin]

* This result assumes that, given a tree with n nodes, we can write, read, indices with O(log n) bits in O(1) time.
Finding the Longest Common Ancestor/Extension

There is a linear-time algorithm for pre-processing a rooted tree, after which the lowest common ancestor of any two nodes can be found in constant time. [Harel and Tarjan] [Schieber and Vishkin]

The length of $S1-S2-LCExtension(i,j)$ can also be found in $O(1)$ time, after linear-time pre-processing, simply by finding the lowest common ancestor of leaves $1,i$ and $2,j$

Pre-processing: build the generalized suffix tree for $S1$ and $S2$, annotate each node with its string-depth, and do the pre-processing needed for lowest common ancestor
Exact Matching with Wildcards

Given pattern P[1..n] and T[1..m], where some positions include a special wildcard symbol “*”, find all occurrences of P in T in which any symbol may match “*”.

Application: identifying DNA transcription factors: proteins that bind to specific locations in DNA and regulate transcription of DNA into RNA. Transcription factors are often described using sequences of amino acids and wildcards

E.g., zing fingers:

CYS * * CYS * * * * * * * * * * * * * * * * * * * * * H I S * * H I S
Exact Matching with Wildcards

Given pattern P[1..n] and T[1..m], where some positions include a special wildcard symbol “*”, find all occurrences of P in T in which any symbol may match “*”.

Application: identifying DNA transcription factors: proteins that bind to specific locations in DNA and regulate transcription of DNA into RNA. Transcription factors are often described using sequences of amino acids and wildcards

E.g., zing fingers:

CYS * * CYS  * * * * * * * * * * * * * * * * * * * * HIS * * HIS
Exact Matching with Wildcards

Idea: Repeatedly use longest common extension queries to match pattern fragments, checking that mismatches occur only at wildcards
Exact Matching with Wildcards

stop ← false

for i from 1 to m-n+1 // check whether P occurs in T starting at position i

j ← 1, i' ← i

repeat

    e ← P-T-LCExtension(j,i')

    if j+e = n+1 then
        output that P occurs in T starting at position i
        if position j+e of P or i'+e of T is "*"
            j ← j+e+1; i' ← i'+e+1
        else stop ← true
    else stop ← false

until stop
Exact Matching with Wildcards

- If there are at most w wildcards in the pattern or text, each iteration of the for loop takes $O(w)$ time.
- Total running time is $O(m+n)$ pre-processing time (for longest common extension) plus $O(wm)$ for all iterations of the for loop.
Exact Matching with Wildcards and KMP

- Can the KMP algorithm be modified to solve exact matching with wildcards?
- Consider the case where the *’s are in the text
Knuth-Morris-Pratt (KMP) Algorithm

algorithm KMP(P[1,...,n],T[1,...,m]) // 1 ≤ n ≤ m
output: list of all numbers s such that P occurs with shift s in T
q ← 0; i ← 0;

repeat // invariant: P[1,...,q] = T[i−q+1,...,i]
  if (P[q+1] ≠ T[i+1]) // mismatch
    if (q = 0) i ← i + 1 // slide the pattern
    else q ← π(q) // slide the pattern
  else // match
    q ← q + 1; i ← i + 1;
    if q = n
      output i−n // there's an exact match of P at shift i-n
    q ← π(q); // slide the pattern to the right
  until (i = m) // end of text has been reached
Knuth-Morris-Pratt (KMP) Algorithm

algorithm KMP(P[1,...,n],T[1,...,m]) // 1 ≤ n ≤ m
output: list of all numbers s such that P occurs with shift s in T
q ← 0; i ← 0;

repeat // invariant: P[1,...,q] = T[i−q+1,...,i]
    if ((P[q+1] ≠ T[i+1]) AND (T[i+1] ≠ “*”)) //mismatch
        if (q = 0) i ← i + 1 // slide the pattern
        else q ← π(q) // slide the pattern
    else // match
        q ← q + 1; i ← i + 1;
        if q = n
            output i−n // there's an exact match of P at shift i-n
        q ← π(q); // slide the pattern to the right
until (i = m) // end of text has been reached
Knuth-Morris-Pratt (KMP) Algorithm

Consider the case where the *'s are the text. We should change the "mismatch" line of the algorithm so that there are no mismatches with wildcards.

But there is a problem: on the next mismatch, we may slide the pattern over too far, not taking advantage of the wildcard.

Try this example: $T = ab*bbba ...$, $P = abbba$

After the first mismatch at position 5, the pattern slides all the way to position 6. But there is a match at position 3.
Knuth-Morris-Pratt (KMP) Algorithm

Exercise

• Describe a modification of KMP in which the pattern can contain any number of wildcard symbols *, each of which matches an arbitrary string. For example, the pattern ABR*CAD*BRA appears in the text SCHABRAINCADBRANCH; in this case, the second * matches the empty string. The goal is to find one match.

• (This can be done in $O(m + n)$ time, where $m$ is the length of the text and $n$ is the length of the pattern)
Knuth-Morris-Pratt (KMP) Algorithm

Exercise

• Describe a modification of KMP in which the pattern can contain any number of wildcard symbols, each of which matches an arbitrary single character. For example, the pattern ABR*CAD*BRA appears in the text SCHABRUCADIBRANCH. The task is to find all matches.

• (This can be done in $O(n + wm)$ time, where $n$ is the length of the pattern, $m$ is the length of the text, and $w$ is the number of *s in the pattern.)
The k-Mismatch Problem
The k-Mismatch Problem

Given a pattern P, a text T, and a fixed number k, a k-mismatch of P is a |P|-length substring of T that matches at least |P|-k characters of P

Goal: Find all k-mismatches of P in T

For example, if P = bend, T = abentbanananaend, and k = 2, then T contains three k-matches of P: P matches substring benr with one mismatch, substring bana with two mismatches, and substring aend with one mismatch.
The k-Mismatch Problem

If k and the size of the alphabet are small, a reasonable option is to try all possibilities using standard suffix tree
The k-Mismatch Problem

for i from 1 to m-n+1
    // check whether a k-mismatch of P occurs in T at i
    j ← 1, i' ← i; count ← 0
    while count ≤ k
        e ← P-T-LCExtension(j,i')
        if j+e = n+1 then
            output that a k-mismatch of P occurs in T at i
            count ← count + 1; j ← j+e+1; i' ← i'+e+1
    endwhile
The k-Mismatch Problem

for i from 1 to m-n+1
   // check whether a k-mismatch of P occurs in T at i
   j ← 1, i' ← i; count ← 0
   while count ≤ k
      e ← P-T-LCEXTENSION(j,i')
      if j+e = n+1 then
         output that a k-mismatch of P occurs in T at i
         count ← count + 1; j ← j+e+1; i' ← i'+e+1
      endwhile

Running time: O(m+n) pre-processing time plus O(km) for all iterations of the for loop
What If the Number of Mismatches is Not Bounded?

No algorithm is known that is linear in the lengths of the pattern and the text

The unbounded-mismatch problem is a special case of the edit distance problem: given two strings $x$ and $y$, find the min-cost alignment between $x$ and $y$, or equivalently the min-cost way to edit $x$ by substitutions, insertions or deletions in order to obtain $y$
String Alignment

Two possible alignments of SNOWY and SUNNY:

```
S  N  O  W  Y
S U N N Y
```

```
-  S  N  O  W  Y
-  S U N - - N Y
```

A cost function $c(a,b)$ assigns a cost to each pair of symbols $a$ and $b$, including the gap symbol.

Goal: Given strings $x[1..m]$ and $y[1..n]$, and a cost function $c(\ , \ )$, find an alignment of $x$ and $y$ with minimum cost.
String Alignment

Let $E(i,j)$ be the cost of the min-cost alignment of prefixes $x[1..i]$ and $y[1..j]$

The cases where $i$ or $j$ are 0 are easy

Otherwise, three possibilities:

• $x_i$, $y_j$ are in the rightmost positions of the alignment
  
  In this case, the cost must be: $E(i-1,j-1) + c(x_i, y_j)$

• $x_i$, gap are in the rightmost positions
  
  In this case, the cost must be: $E(i-1,j-1) + c(x_i, \text{gap})$

• gap, $y_j$ are in the rightmost positions
  
  In this case, the cost must be: $E(i-1,j-1) + c(\text{gap}, y_j)$
String Alignment

Recurrence: When \( i, j \geq 1 \)

\[
E(i,j) = \min \left\{ E(i-1,j-1) + c(x_i, y_j), \right. \\
E(i-1,j-1) + c(x_i, \text{gap}), \right. \\
E(i-1,j-1) + c(\text{gap}, y_j) \left. \right\} 
\]

Also,

\[
E(i,0) = \sum_{k \text{ from 1 to } i} c(x_k, \text{gap}) \\
E(0,j) = \sum_{k \text{ from 1 to } j} c(\text{gap}, y_k) 
\]

We can calculate the \( E(i,j) \) values in increasing order of \( i \) and \( j \)
String Alignment

Example: \( x = \text{EXPONENTIAL}, \ y = \text{POLYNOMIAL} \)

The cost of a pair of matching symbols is 0 and of non-matching symbols is 1

For this example, the minimum alignment cost is 6

\[
\begin{align*}
\text{EXPO} & \quad \text{POLY} \\
\text{NEN} & \quad \text{NOMI} \\
\text{NTIAL} & \quad \text{NAL} \\
\text{—} & \quad \text{—} \\
\end{align*}
\]
String Alignment

Example: $x = \text{EXPONENTIAL}, \ y = \text{POLYNOMIAL}$

The cost of a pair of matching symbols is 0 and of non-matching symbols is 1

For this example, the minimum alignment cost is 6

```
  E X P O N E N T I A L
  --POLYNOMIAL
```
<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>O</th>
<th>L</th>
<th>Y</th>
<th>N</th>
<th>O</th>
<th>M</th>
<th>I</th>
<th>A</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td><strong>X</strong></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td><strong>P</strong></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td><strong>O</strong></td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>N</strong></td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>I</strong></td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td><strong>L</strong></td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
## String Alignment

<table>
<thead>
<tr>
<th></th>
<th>POLY</th>
<th>NOMIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>-</td>
</tr>
</tbody>
</table>

The red line indicates the alignment of the two strings.
• Suffix trees and their variants provide elegant and efficient solutions to many problems on strings; Gusfield describes many more such problems in his textbook

• More general alignment problems seem to require dynamic programming methods, which are less efficient in time and space

• Tardos and Kleinberg do describe a divide-and-conquer approach that reduces the space for alignment to linear space