Aligning text to find recurring material

(1) Review of last time:
   - Introduction to MDL: a fundamentally different approach to segmenting text
     - Rather than focusing explicitly on the boundaries and how to predict them, focus instead of what comes between the boundaries
     - Material between boundaries = lexical items, stored in a lexicon
     - Text of utterances is analyzed (parse) as a sequence of lexical items
     - Goal is to find the best combination of lexicon + parse to represent the text as compactly as possible
   - Which of the following goals does an analysis based on Minimal Description Length achieve? (From last time's handout)
     - Don't leave long strings of text unanalyzed (e.g., don't play with the stove dillon)
     - Don't break text up too much (e.g., do you see the kitty)
     - Break utterances into chunks that occur elsewhere (e.g., no dillion, not nods llon)
     - Prefer segmentations that make use of more frequent words (e.g., I want to astonish him, not I want toast on ishhim)
   - A Perl script for calculating description length is the DescriptionLength.pl script, in the Assignment3 directory in the courselocker
     - Usage: ./DescriptionLength.pl lexicon-file utterances-file

(2) A note on description lengths:
   - There are many possible improvements to the way that description length is calculated, making use of observations from probability theory, information theory.
   - Understanding the purpose and function of these refinements would require a fair amount of background in these areas
   - We will not go into the details of these

(3) A major issue in finding MDL analyses:
   - How do you know what lexicon to try? How do you know how to change the lexicon in order to try to find a shorter description length?
   - In the case of the do you see the kitty text, it seems sort of obvious—and anyway, the text is short, so we can explore a large number of possibilities.
   - The problem of figuring out what to separate and what to leave as a chunk in the lexicon should become very clear to you as you attempt the homework for next week.

(4) Let's return to the simple do you see the kitty language:

   do you see the kitty
   see the kitty
   do you like the kitty

The analysis with the minimum description length for this text is:

<table>
<thead>
<tr>
<th>Lexicon</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 doyou</td>
<td>1 2 3</td>
</tr>
<tr>
<td>2 see</td>
<td>2 3</td>
</tr>
<tr>
<td>3 thekitty</td>
<td>1 4 3</td>
</tr>
<tr>
<td>4 like</td>
<td>8 characters</td>
</tr>
<tr>
<td>24 characters</td>
<td>Total description length: 32</td>
</tr>
</tbody>
</table>
The problem: how do we arrive at the lexicon in (5), given just the text in (4)?

- How about just trying all logical possibilities? (d, do, doy, doyou, doyous, etc…)
  - Would this work? What is the problem with it?
- Or how about starting with a random set of word divisions, then randomly moving them around to see if things improve?
  - Would this work? What is the problem with it?
  - ?
- Or looking for utterances that are substrings of others? (e.g., no/nodillon, seethekitty/doyouseethekitty, etc.)
  - Would this work? What is the problem with it?

Using alignment to find recurring substrings

- The kitty is shared by all three utterances
- see is shared by two utterances
- doyou is shared by two utterances
- like is left over when you factor out doyou and the kitty

➢ If only we had a way to align texts in this way, we could find these recurring substrings…

String edit distance: the number of changes needed to change one string into another

- Delete ‘doyou’
- Change ‘doy’ to ‘see’
- Delete ‘doyousee’
- Delete ‘ousee’
- Insert ‘see’

5 changes 8 changes 11 changes

- The optimal alignment of the two utterances is the one with the smallest string edit distance
- Minimal string edit distance/Levenshtein distance

Goal of this class: outline a procedure for calculating the optimal alignment/minimum string edit distance between two strings

Calculating string edit distance

Let’s compare the following two strings:

- seethekitty (String 1, n characters long, n=11)
- likethekitty (String 2, m characters long, m=12)

Goal: find the alignment in which the kitty is aligned, see and like are not.

Step 1: Construct an $m \times n$ grid
Goal: Get through the grid from top left to lower right, by moving down, right, and diagonally.

- An arbitrary example:

```
  a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   p   q   r   s   t   u   v   w   x   y
  1   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  k   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  a   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  t   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  h   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  e   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  l   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  k   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  l   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
  y   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -   -
```

A character from string 2 is consumed, but not string 1
A character from string 1 is consumed, but not string 2
A character is consumed from each string

- What alignment of the two strings is shown in this arbitrary example?

Before we continue, we need to set some priorities:

- **indel cost**: what is the penalty for inserting/deleting a segment?
- **substitution cost**: what is the penalty for substituting an a for a b?

We'll start by assuming the following default values:

- indel cost = .5
- substitution cost: \( \begin{cases} 0, & \text{if } a = b \\ 1, & \text{if } a \neq b \end{cases} \)

Step 2: Put initial values in the first row/first column

- Moving across the top row would mean deleting everything in string 1 without aligning it to anything in string 2
- Moving down the first column column would mean inserting everything in string 2 before making us of anything from string 1
- So, these cells count as insertions/deletions; get the indel cost
Step 3: Comparing characters to find the best alignment

Starting in the upper left, we’re going to calculate three things for each cell:

- The cost of substituting the current char in string 2 for the current char in string 2
- The cost of deleting the current char in string 1 without matching it up to a char in string 2
- The cost of inserting the current char in string 2 without consuming a char in string 1

For convenience, we’ll store these values in the corners of the cell:

```
<table>
<thead>
<tr>
<th>subst</th>
<th>del</th>
<th>cost</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

For the first cell in our example, the first character of string 1 is s and the first character of string 2 is tt k. They don’t match, so, using the default values in (13):

- The substitution cost = 1
- Deletion cost = .5
- Insertion cost = .5

Partway through: no characters matching yet

After some more comparisons:

And so on...
(17) With all comparisons filled in:

```
    0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  0: 0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  1: 1.5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  2: 2.5  2  2.5  3  3.5  4  4.5  5  5.5
  3: 3.5  3  3.5
  4: 4.5  4
  5: 5.5
```

When the indel cost is always .5 and the substitution cost is always 0 or 1, this step is pretty easy (even if a lot of tiny numbers start to accumulate)

(18) The next step: finding the shortest path

- For each cell, figure out which is smallest:
  - The value in the cell above it + the insertion cost
  - The value in the diagonal cell + the substitution cost
  - The value in the cell to the left + the deletion cost

(This is why we kept track of those values in the corners of the cell)

```
    0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  0: 0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  1: 1.5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  2: 2.5  2  2.5  3  3.5  4  4.5  5  5.5
  3: 3.5  3  3.5
  4: 4.5  4
  5: 5.5
```

- In this case, they are all the same
- It is helpful to remember which value(s) were the minimum, by drawing arrows back to the cells which

(19) A bit further through:

```
    0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  0: 0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  1: 1.5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5
  2: 2.5  2  2.5  3  3.5  4  4.5  5  5.5
  3: 3.5  3  3.5
  4: 4.5  4
  5: 5.5
```

(20) Try calculating the minimum values in the chart at the top of the page
(21) The final tabulation:

(22) The last step: tracing back through the chart to find the optimal alignment

- Starting at the lower right corner, follow the arrows back up and to the left
- There are sometimes multiple possible paths! There may be many possible edits that all have the same cost
- What are some conventions for following paths that would make the edits more intuitive?