Edit Distance – Levenshtein

Sequence Alignment – Needleman & Wunsch

Not in Book

http://en.wikipedia.org/wiki/Levenshtein_distance

EDIT DISTANCE

Edit Distance

- Mutation in DNA is evolutionary.
- DNA replication errors cause
  - Substitutions
  - Insertions
  - Deletions
- of nucleotides, leading to “edited” DNA texts

Edit Distance: Definition

- Introduced by Vladimir Levenshtein in 1966
- The Edit Distance between two strings is the minimum number of editing operations needed to transform one string into another
- Operations are:
  - Insertion of a symbol
  - Deletion of a symbol
  - Substitution of one symbol for another

Example

- How would you transform:
  - X: TGCATAT
- To the string:
  - Y: ATCCGAT

Edit Distance

- How many insertions, deletions, substitutions will transform one string into another?

- Backtracking will give us the steps used to convert one string to another

Recursive Solution

- Let d_{ij} = the minimum edit distance of $x_1x_2x_3..x_i$ and $y_1y_2y_3..y_i$

$$d[i,j] = \begin{cases} d[i-1,j-1] \quad if\ x_i = y_i \\ min \begin{cases} d[i-1,j]+1 \quad if\ x_i \neq y_i \\ d[i,j-1]+1 \quad if\ x_i \neq y_i \\ d[i-1,j-1]+1 \quad if\ x_i \neq y_i \end{cases} \end{cases}$$

Match

Deletion from X

Insertion to X

Substitution

Backtracking

- No need to keep track of the arrows
- Just know that:
  - Match/Substitution: Diagonal
  - Insertion: Horizontal (Left)
  - Deletion: Vertical (Up)

Example

- X = ATCGTT
- Y = AGTTAC

		A	G	T	T	A	C
	0	1	2	3	4	5	6
A	1						
T	2						
C	3						
G	4						
T	5						
T	6						

Kleinberg, Tardos, Algorithm Design, Pearson Addison Wesley, 2006, p 278

http://www.aw-bc.com/info/kleinberg/

SEQUENCE ALIGNMENT

Sequence Alignment

- Edit Distance:
  - Gave the minimum number of changes to convert one string into another
- Sequence Alignment
  - Maximizes the similarity by giving weights to types of differences

Sequence Alignment

- Needleman-Wunsch
- Similarity based on gaps and mismatches
- Generalized form of Levenshtein
  - additional parameters:
    - gap penalty, δ
    - mismatch cost ($\alpha_{x,y}$; $\alpha_{x,x} = 0$)

Recurrence

- Two strings $x_1 \ldots x_m$ and $y_1 \ldots y_n$
- In an optimal alignment, *M*, at least one of the following is true:
  - (x_m, y_n) is in M
  - x_m is not matched
  - y_n is not matched

Recurrence

- So, for i and j > 0

$$opt(i,j) = \min[\alpha_{xi,yj} + opt(i-1,j-1),$$
$$\delta + opt(i-1,j), \quad // \ x_i \text{ is not matched}$$
$$\delta + opt(i,j-1)\ ] \quad // \ y_j \text{ is not matched}$$

Example

- Assume that:
  - δ = 2
  - α (v, v) = 1
  - α (c, c) = 1
  - α (v, c) = 3

- What is the cost of aligning the strings:
  - mean
  - name

SPACE-EFFICIENT SEQUENCE ALIGNMENT

Sequence Alignment Space Usage

- $O(n^2)$ is pretty low space usage
- However, for a 10GB genome, you'd need a huge amount of memory
- Can we use less?
  - Hirschberg's algorithm
  - 1975

Linear Space for Alignment Scores

- If you are only interested in the cost of the alignment, you need to only use O(n) space
- How?
  - When filling the entries, we only ever look at the current and previous cols
  - Only keep those two in memory

Space-Efficient-Alignment (X, Y)

```
Space-Efficient-Alignment(X, Y)
  Array B[0 ... m, 0 ... 1]
  Initialize B[i, 0] = iδ for each i (just as in column 0 of A)
  For j = 1, ..., n
     B[0, 1] = jδ (since this corresponds to entry A[0, j])
     For i = 1, ..., m
          B[i, 1] = min[α_{x_i y_j} + B[i − 1, 0],
                        δ + B[i − 1, 1],  δ + B[i, 0]]
     Endfor
     Move column 1 of B to column 0 to make room for next iteration:
          Update B[i, 0] = B[i, 1] for each i
  Endfor
```

Actual Alignment

- How do we recover the actual alignment?

- Do we need the entire matrix?

Divide-and-Conquer-Alignment (X,Y)

```
Divide-and-Conquer-Alignment(X, Y)
  Let m be the number of symbols in X
  Let n be the number of symbols in Y
  If m ≤ 2 or n ≤ 2 then
     Compute optimal alignment using Alignment(X, Y)
  Call Space-Efficient-Alignment(X, Y[1 : n/2])
  Call Backward-Space-Efficient-Alignment(X, Y[n/2 + 1 : n])
  Let q be the index minimizing f(q, n/2) + g(q, n/2)
  Add (q, n/2) to global list P
  Divide-and-Conquer-Alignment(X[1 : q], Y[1 : n/2])
  Divide-and-Conquer-Alignment(X[q + 1 : n], Y[n/2 + 1 : n])
  Return P
```