Lecture 2: Pairwise sequence alignment*

- Biological motivation
- The notion of sequence similarity and alignment
- The algorithm that computes them: dynamic programming (DP)

For the dynamic programming technique

- Intuitive and formal definition of the recursive process
- The algorithm
- Discussion of time and space complexity

*Thanks to Prof. Ron Shamir http://www.math.tau.ac.il/~rshamir/algmb/01/algmb01.html

Biological motivations

•A large variety of biology motivated problems in computer science involve sequences (character strings). For instance:

•Reconstructing long sequences of DNA from overlapping fragments

•Comparing two or more sequences for similarities

•Searching data bases for related sequences and sub sequences

•Exploring frequently occurring patterns of nucleotides

• Finding informative elements in protein and DNA sequences

This research aims at learning about the functionality or structure of a protein without experimenting in the lab. The idea being that similar sequences produce similar proteins

Resemblance and difference of sequences

The resemblance of two DNA sequences taken from different organisms, is explained by the theory that all genetic material has a common ancestral DNA.

Differences between contemporary species are due to mutations occurring in the course of evolution.

Many of the changes are local and can be either

- Insertion – insert a letter to the sequence (character string)
- Deletion – delete a base (letter) from the DNA chain
- Substitution – replace one sequence base by another

Insertions and deletions (InDel) are the reverse of each other

Distance and Similarity

• Distance derives its definition from the concept of mutations by assigning weight to each mutation: Given two sequences the distance between them is the minimal sum of weights for the set of mutations that transform one sequence to another.

• For similarity we assign weights corresponding to base resemblance: Given two sequences the similarity between them is the maximal sum of such weights.

The Simplest Model: Edit Distance

Definition: The edit distance between two sequences is the minimal number of edit operations (indel, substitutions) needed to transform one sequence to the other.

Example: Given two sequences a c c t g a and a g c t a

The minimal number of edit operations is 2

a c c t g a	Substitute c with g	a c c t g a
a g c t g a	Delete g	a g c t g a
a g c t a		a g c t - a

Alignment

Definition: An alignment of two sequences S and T is obtained by placing the two sequences one above the other (allowing insertions of spaces), so that every character (or space) in one sequence has a unique character (or space) in the other sequence.

In the alignment model, each two char-char or char-space alignment are given a score, and the goal is to minimize the score.

Example: The aligned sequences

S1: GTAGTACAGCT-CAGTTGGGATCACAGGCTTCT

S2: GTAGAACGGCTTCAGTTG- - -TCACAGCGTTC-

Dist1: match 0, substitution 1, indel 2 => dist1 = 14.

Dist2: match 0, d(A,T)=d(C,G)=1,d(A,G)=1.5,indel=2 => dist2=14.5

Similarity: match 1, substitution 0, indel=-1.5 => similarity=16.5

1. Global alignment

In global alignment the whole sequences are aligned (allowing spaces) to maximize similarity of the sequences.

Input: Two sequences S=S[1..m], T=T[1..n], (m and n are approximately the same).

Notation: M[x,y] is the score of aligning character x with character y, for every two characters in the alphabet, including spaces.

Notation: V[j,k] denotes the score of the optimal alignment of the prefixes S[1..j] and T[1..k].

V[S,T] denotes the score of the optimal alignment of S and T.

We will give a recursive description of V[S,T], show how to compute it, and consequently get the best alignment.

Lemma: Let V[j,k] be the optimal alignment score of S[1..j] with T[1..k]. Then V[S,T]=V[m,n] has the following properties:

Base conditions: $V[j,0] = \Sigma\, M[s_r,-]$, $r=0..j$

$V[0,k] = \Sigma\, M[-,t_r]$, $r=0..k$

Recursive relation: for each j in [1..m], and k in [1..n]

$$V[j,k) = \max \begin{cases} V[j-1,k-1] + M[s_j,t_k] \\ V[j-1,k] + M[s_j,-] \\ V[j,k-1] + M[-,t_k] \end{cases}$$

This is the Dynamic Programming formulation of the best alignment problem

Tabular computation of the best score and an optimal alignment

We compute the table V[0..m,0..n] from low indices to m and n. V[m,n] will be the score of the best alignment, and the path in the table will give the best alignment

Compute V[0,k] and V[j,0] for all j, and for all k, in 1..m, 1..n, resp.

For j=1 to m

For k=1 to n

Calculate V[j,k] using V[j-1,k-1],V[j-1,k],V[j,k-1]

using the MAX equation seen before.

Example: Align "CATGT" and "ACGCTG".

Values: match 2, mismatch and space –1

	k	0	1	2	3	4	5
j			C	A	T	G	T
0		0	-1	-2	-3	-4	-5
1	A	-1	-1	1			
2	C	-2					
3	G	-3					
4	C	-4					
5	T	-5					
6	G	-6					

Filling the matrix and finding the best score V[6,5]=2

	k	0	1	2	3	4	5
j			C	A	T	G	T
0		0	-1	-2	-3	-4	-5
1	A	-1	-1	1	0	-1	-2
2	C	-2	1	0	0	-1	-2
3	G	-3	0	0	-1	2	1
4	C	-4	-1	-1	-1	1	1
5	T	-5	-2	-2	1	0	3
6	G	-6	-3	-3	0	3	2

The best score for aligning S and T

Backtracking the paths from V[6,5] to V[0,0] to find the good alignments

	k	0	1	2	3	4	5
j			C	A	T	G	T
0		0	-1	-2	-3	-4	-5
1	A	-1	-1	1	0	-1	-2
2	C	-2	1	0	0	-1	-2
3	G	-3	0	0	-1	2	1
4	C	-4	-1	-1	-1	1	1
5	T	-5	-2	-2	1	0	3
6	G	-6	-3	-3	0	3	2

The high path gives the alignment

CATG - T -

- ACGCTG

Time and space complexity

•Clearly the computation of V take O(mn) time.

•And the space needed is O(mn) if backtrack is required.

•Space is easily reduced to O(m+n) if just the score V[m,n] is required

For huge length DNA the space is a bottleneck. Hirschberg [1] developed a space reduction algorithm for the dynamic programming reducing the O(mn) space to O(min(m,n)).

[1] D.S. Hirschberg. Algorithms for the longest common subsequence problem. J. ACM. 24:664-675, 1977

Models of alignment

We will talk on four alignment problems and their solutions:

1. Global alignment – Sequences S and T are roughly the same length. What is their similarity? Find the best alignment.
2. Local alignment. For any S and T, what is the maximum similarity between subsequences of S and T. Find most similar subsequences.
3. Ends free alignment. Find the best alignment between subsequences of S and T, when at least one subsequence is a prefix of the original sequence, and the other is a suffix. (Will be taught in section.)
4. Gap penalty. Find a best alignment between S and T using the gap penalty function

Definition. A gap is a maximal contiguous run of spaces in a single sequence within a given alignment. A gap penalty function is a function that measures the cost of a gap as a function of its length.

2. Local alignment

In many applications sequences are not highly similar as a whole but they may contain subsequences with high resemblance.

Problem definition: Given two sequences S=S[1..m], T=T[1..n], find the subsequences α of S and β of T, whose (global) similarity is maximal over all subsequences.

Motivation:

• In DNA non-coding regions (introns) are subject to more mutations than coding regions (exons). Search for local alignment of two DNA stretches is more likely to find the best match between two exons.

• Proteins of different kinds and of different species often exhibit local similarities which might be "functional subunits" of the protein. Finding these *domains* is done by local alignment.

Computing local alignment

Definition. Given two sequences, S and T, and two indices, j and k, the *local suffix alignment* problem is finding a (possibly empty) suffix α of S[1..j] and β of T[1..k] such that the value of their alignment is maximal over alignments of all suffixes of S[1..j] and T[1..k] .

• The solution of the local alignment is the maximal solution for the local suffix alignment over all indices j and k.

Terminology and restrictions

• V[j,k] denotes the value of the optimal local suffix alignment for j,k

• The weights of the editing operations are limited by

M[x,y] >= 0 if x and y match

M[x,y] <= 0 if x,y do not match or one of them is a space

The recursive formulation is very similar to that of global alignment

•Base conditions: $V[j,0] = 0$, $V[0,k] = 0$ for each j,k

•Recursive relation: for each j in [1..m], and k in [1..n]

$$V[j,k) = \max \begin{cases} 0 \\ V[j-1,k-1] + M[s_j,t_k] \\ V[j-1,k] + M[s_j,-] \\ V[j,k-1] + M[-,t_k] \end{cases}$$

•Compute j' and k' such that $V[j',k'] = \max \{V[j,k] \mid \text{for all } j,k\}$

Example: Locally align "CATGT" and "ACGCTG".

Values: match 2, mismatch and space –1

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

Result: **CATG**
C- TG

V[6,4] is maximal

Gap Penalty - definitions

A gap is a maximal consecutive run of spaces in an alignment.

Gaps help create alignments that conform better to biological models, and fit patterns that one expects to find in meaningful alignments.

We denote the *number of gaps* in an alignment by #gaps.
The *length of a gap* is the number of indel operaions in it.

Example:
S = ATTC - - GA–TGGACC
T = A - -CGTGATT - - - CC

This alignment can be described as having 7 matches, no mismatch, 4 gaps and 8 spaces (indel).

Various gap penalty models

1. The gap penalty we used so far was a linear function of gap length:

*#indel*indel_cost*

2. We might want to asign the whole gap a constant cost Wg.

Here the cost of a space is zero, M[j,-], M[-,k] = 0.

Now we get S' and T' after the spaces have been inserted to S and T respectively, and have to find an alignment that maximizes

$\Sigma_k \{M(S'[k],T'[k])\} + Wg*(\#gaps)$

3. In affine gap penalty we assign a cost W_o for "openning" a gap and a cost W_e for "extending" a gap. The penalty of one gap of length q is

$W_{total} = W_o + qW_e$

Affine gap penalty algorithm

Similar to constant gap penalty the goal now is to find an alignment that maximizes

$\Sigma_k \{M(S'[k],T'[k])\} + W_o*(\#gaps) + W_e*(\#spaces)$

To align sequences S and T, consider the prefixes S[1..j], T[1..k]

They can be aligned in 3 different ways

1. S _____j
   T_____k
2. S_____j--------
   T_____k
3. S_____j
   T_____k----

Let us denote by

- G[j,k] the maximum value of any alignment of type 1
- E[j,k] the maximum value of any alignment of type 2
- F[j,k] the maximum value of any alignment of type 3
- V[j,k] the maximum value of an alignment

We will define 3 recurrence relations and proceed to the alignment table as before

Take E[j,k] for example

2. S_____j--------
   T_____k

There are two possible cases for the previous alignment

(i) It looked the same (was type 2). In this case we need only to add extension value W_e forming $E[j,k-1] + W_e$

(ii) It looked like type 1, in this case we need to add gap opening cost and gap extension, so the new cost is $V[j,k-1] + W_o + W_e$

$E[j,k] = max \{E[j,k-1] + W_e, V[j,k-1] + W_o + W_e\}$

G and F are calculated similarly, and V just takes the maximum over all three. Hence,

The recursive definition:

Base conditions: $V[0,0] = 0$

$V[j,0] = F[j,0] = Wo+j*We$

$V[0,k] = E[0,k] = Wo+k*We$

$F[0,i]=E[i,0] = -\infty$

Recurrence relations:

$V[j,k] = \max \{E[j,k], F[j,k], G[j,k]\}$

where $G[j,k] = V[j-1,k-1] + M[S[j], T[k]]$

$E[j,k] = \max \{E[j,k-1] + W_e, V[j,k-1] + W_o + W_e\}$

$F[j,k] = \max \{F[j-1,k] + W_e, V[j-1,k] + W_o + W_e\}$

Time and space complexity O(mn) as before.