Lecture 9:
Sequence Alignments

Study Chapter 6.4-6.8

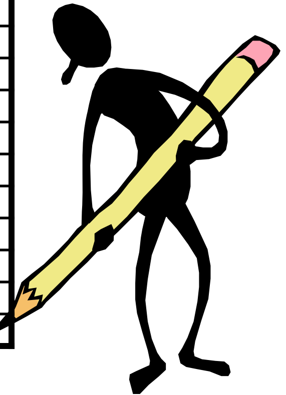
Outline

- Edit Distances
- Longest Common Subsequence
- Global Sequence Alignment
- Scoring Matrices
- Local Sequence Alignment
- Alignment with Affine Gap Penalties

Review

- ***Dynamic Programming*** is a technique for computing recurrence relations efficiently by storing partial results

Amt	25	20	10	5	1	Amt	25	20	10	5	1
1¢					1	42¢		2			2
2¢					2	43¢		2			3
3¢					3	44¢		2			4
4¢					4	45¢		2		1	
5¢				1		46¢		2		1	1
6¢				1	1	47¢		2		1	2
7¢				1	2	48¢		2		1	3
8¢				1	3	49¢		2		1	4
9¢				1	4	50¢	2				
10¢			1			51¢	2				1
11¢			1		1	52¢	2				2



- Three keys:
  1. Formulate the answer as a recurrence relation
  2. Consider all instances of the recurrence at each step
  3. Order evaluations so you will always have the needed partial results

A Biological DP Problem

- How to measure the similarity between a pair of nucleotide or amino acid sequences
- In the Motif-Searching Problem (Chapter 4) we used Hamming distance as our measure
- Is Hamming distance the best measure?
- How can we distinguish matches that occur by chance from slightly modified patterns?
- What sorts of modifications should we allow?

Best Sequence Matches

- Depends on how you define "Best"
- Consider the two DNA sequences v and w :

v : **ATATATAT**
w: **TATATATA**

- The Hamming distance: $d_H(v, w) = 8$ is large but the sequences are very similar
- What if we allowed insertions and deletions?

Allowing Insertions and Deletions

By shifting one sequence over one position:

_v_ : ATATATAT--
_w_ : --TATATATA

- The *edit* distance: $d\,(v, w) = 2$.
- Hamming distance neglects insertions and deletions in DNA

Edit Distance

Levenshtein (1966) introduced the notion of an "edit distance" between two strings as the minimum number of elementary operations (insertions, deletions, and substitutions) to transform one string into the other.

(But, he gave no solution)

$d(\mathbf{v},\mathbf{w})$ = MIN number of elementary operations to transform $\mathbf{v} \rightarrow \mathbf{w}$

Edit Distance vs Hamming Distance

Hamming distance always compares
*i*th letter of **v** with
*i*th letter of **w**

V = ATATATAT
| | | | | | | |
W = TATATATA

Just one shift
Lines them up

Edit distance may compare
*i*th letter of **v** with
*j*th letter of **w**

V = - ATATATAT
| | | | | | | |
W = TATATATA

Hamming distance:

$d_H(\mathbf{v}, \mathbf{w})=8$

Computing Hamming distance is a trivial task

Edit distance:

$d(\mathbf{v}, \mathbf{w})=2$

Computing edit distance is a non-trivial task

Edit Distance: Example

TGCATAT → ATCCGAT in 5 steps

TGCATAT	→	(DELETE last T)
TGCATA	→	(DELETE last A)
TGCAT	→	(INSERT A at front)
ATGCAT	→	(SUBSTITUTE C for 3rd G)
ATCCAT	→	(INSERT G before last A)
ATCCGAT		(Done)

What is the edit distance? 5?

Edit Distance: Example (cont'd)

TGCATAT → ATCCGAT in 4 steps

TGCATAT → (INSERT A at front)

ATGCATAT → (DELETE 6th T)

ATGCAAT → (SUBSTITUTE G for 5th A)

ATGCGAT → (SUBSTITUTE C for 3rd G)

ATCCGAT (Done)

Is 4 the minimum edit distance? 3?

A little jargon: *Since the effect of insertion in one string can be accomplished via a deletion in the other string these two operations are quite similar. Often algorithms will consider them together as a single operation called* ***INDEL***

Longest Common Subsequence

- A special case of edit distance where no substitutions are allowed
- A subsequence need not be contiguous, but order must be preserved

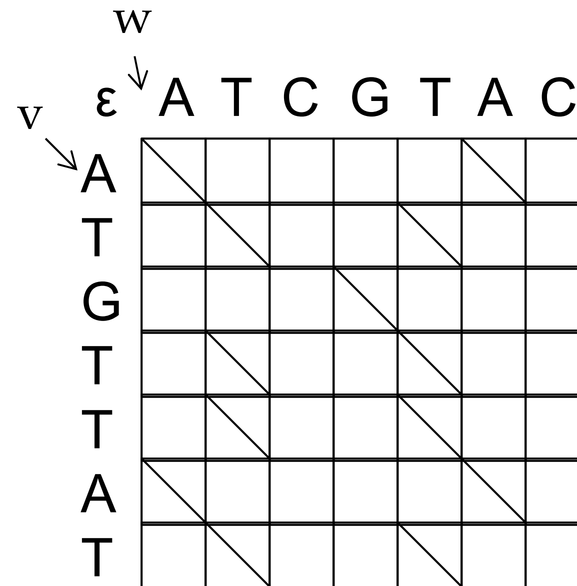
  Ex. If v = ATTGCTA then AGCA and TTTA are subsequences of v, but TGTT and ACGA are not

- For sequences v and w, the edit distance $d(v,w)$ and the $LCS(v,w)$ are related by:

$$d(v,w) = len(v) + len(w) - 2\ len(\ LCS(v,w)\ )$$

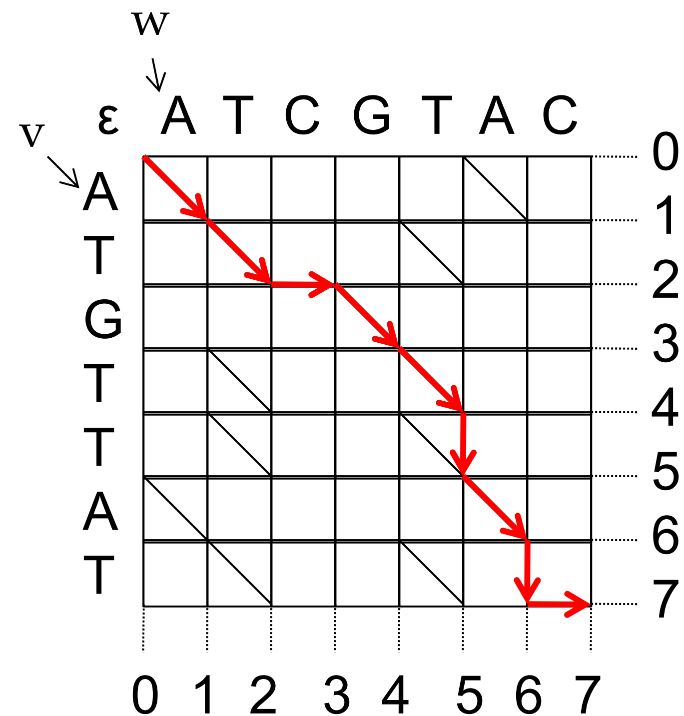
LCS as a Dynamic Program

- All possible possible alignments can be represented as a path from the string's beginning (source) to its end (destination)
- Horizontal edges add gaps in v. Vertical edges add gaps in w. Diagonal edges are a match
- Notice that we've only included valid diagonal edges in our graph



Various Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

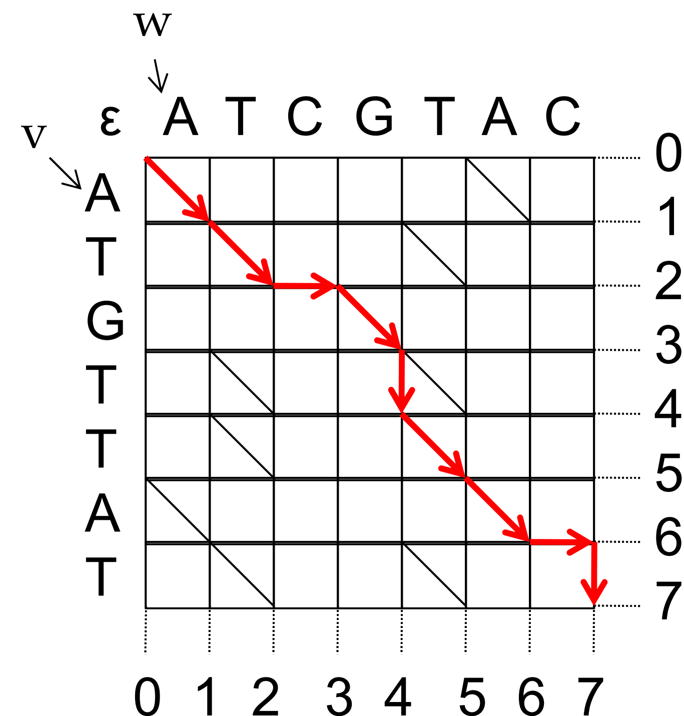



```
0 1 2 2 3 4 5 6 7 7
v A T _ G T T A T _
w A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7
```

Path:
(0,0), (1,1), (2,2), (2,3),
(3,4), (4,5), (5,5), (6,6),
(7,6), (7,7)

Various Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment

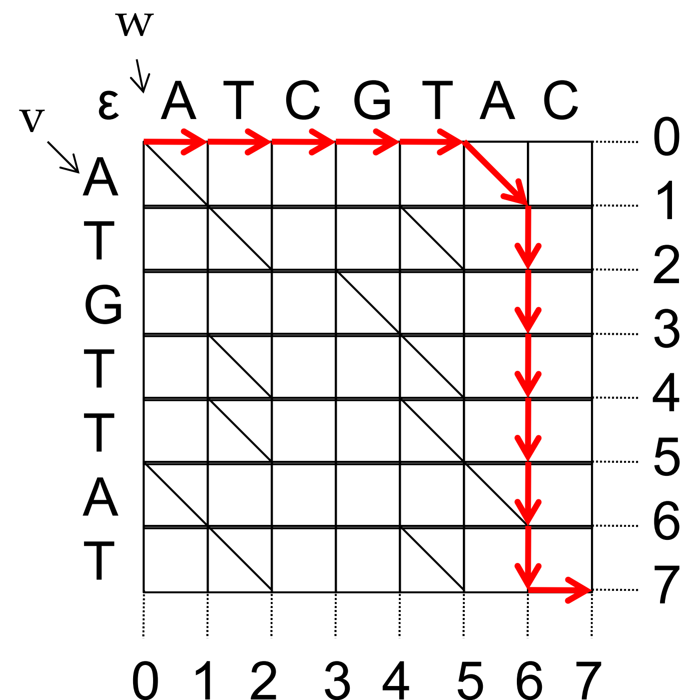



```
0 1 2 2 3 4 5 6 6 7
v A T _ G T T A _ T
w A T C G _ T A C _
0 1 2 3 4 4 5 6 7 7
```

Path:
(0,0), (1,1), (2,2), (2,3), (3,4), (4,4), (5,5), (6,6), (6,7), (7,7)

Even Bad Alignments

- Introduce coordinates for the grid
- All valid paths from the source to the destination represent some alignment




```
0 0 0 0 0 0 1 2 3 4 5 6 7 7
v _ _ _ _ _ A T G T T A T _
w A T C G T A _ _ _ _ _ _ T
0 1 2 3 4 5 6 6 6 6 6 6 6 7
```

```
Path:
  (0,0),  (0,1),  (0,2),  (0,3),
  (0,4),  (0,5),  (1,6),  (2,6),
  (3,6),  (4,6),  (5,6),  (6,6),
  (7,6),  (7,7)
```

What makes a Good Alignment?

- Using as many diagonal segments (matches) as possible
- The end of a good alignment from (j…k) begins with a good alignment from (i..j)
- Same as Manhattan Tourist problem, where the sites are the diagonal streets!
- Set diagonal street weights = 1, and horizontal and vertical weights = 0

Alignment: Dynamic Program

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_i \quad \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

	w	A	T	C	G	T	A	C
v								
A								
T								
G								
T								
T								
A								
T								

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0							
T	0							
G	0							
T	0							
T	0							
A	0							
T	0							

Initialize *1st* row and *1st* column to be all zeroes.

Or, to be more precise, initialize *0th* row and *0th column to be all zeroes.*

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1						
G	0	1						
T	0	1						
T	0	1						
A	0	1						
T	0	1						

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_i \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2					
T	0	1	2					
T	0	1	2					
A	0	1	2					
T	0	1	2					

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & if\ v_i = w_i \ \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2				
T	0	1	2	2				
A	0	1	2	2				
T	0	1	2	2				

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_i \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3			
A	0	1	2	2	3			
T	0	1	2	2	3			

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & if\ v_i = w_i \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4		
T	0	1	2	2	3	4		

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & if\ v_i = w_i \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_i \ \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	5

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & if\ v_i = w_i \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

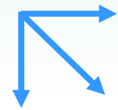
Dynamic Programming Example

	w	A	T	C	G	T	A	C
v	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
T	0	1	2	2	2	2	2	2
G	0	1	2	2	3	3	3	3
T	0	1	2	2	3	4	4	4
T	0	1	2	2	3	4	4	4
A	0	1	2	2	3	4	5	5
T	0	1	2	2	3	4	5	5

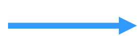
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \textit{if } v_i = w_i \searrow \\ s_{i-1,j} & \downarrow \\ s_{i,j-1} & \rightarrow \end{cases}$$

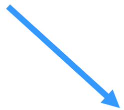
```
v = AT-GTTA-T
w = ATCG-TAC-
```

Alignment: Backtracking

 Arrows show where the score came from.

if from the top

 if from the left

 if $v_i = w_j$

Our table only keeps track of the longest common subsequence so far. How do we figure out what the subsequence is?

We'll need a second table to keep track of the decisions we made… and we'll use it to backtrack to our answer.

LCS Code

```
def LCS(v, w):
    s = zeros((len(v)+1,len(w)+1), Int)
    b = zeros((len(v)+1,len(w)+1), Int)
    for i in xrange(1,len(v)+1):
        for j in xrange(1,len(w)+1):
            if (v[i-1] == w[j-1]):
                s[i,j] = max(s[i-1,j], s[i,j-1], s[i-1,j-1] + 1)
            else:
                s[i,j] = max(s[i-1,j], s[i,j-1])
            if (s[i,j] == s[i,j-1]):
               b[i,j] = 1  →
            elif (s[i,j] == s[i-1,j]):
               b[i,j] = 2  ↓
            else:
               b[i,j] = 3  ↘
    return (s[len(v),len(w)], b)
```

Backtracking Code

```
def PrintLCS(b,v,i,j):
    if ((i == 0) or (j == 0)):
        return
    if (b[i,j] == 3):
        PrintLCS(b,v,i-1,j-1)
        print v[i-1],
    else:
        if (b[i,j] == 2):
            PrintLCS(b,v,i-1,j)
        else:
            PrintLCS(b,v,i,j-1)
```

Changing the Scoring

- Longest Common Subsequence (LCS) problem
  - the simplest form of sequence alignment
  - allows only insertions and deletions (no mismatches).
- In the LCS Problem, we scored 1 for matches and 0 for indels
- Consider penalizing indels and mismatches with negative scores
- Simplest *scoring schema*:

  $+1$: **match premium**

  $-\mu$: **mismatch penalty**

  $-\sigma$: **indel penalty**

Simple Scoring

- When mismatches are penalized by $-\mu$
- indels are penalized by $-\sigma$
- matches are rewarded with $+1$

the resulting score is:

$$\text{score} = \#\textit{matches} - \mu(\#\textit{mismatches}) - \sigma\,(\#\textit{indels})$$

The Global Alignment Problem

Find the best alignment between two strings under a given scoring schema

Input : Strings **v** and **w** and a scoring schema
Output : Alignment of maximum score

↑ or → = -б

↘ { = *1* if match, = -μ if mismatch

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + 1 & \text{if } v_i = w_j \\ s_{i-1,j-1} - \mu & \text{if } v_i \neq w_j \\ s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \end{cases}$$

μ : mismatch penalty
σ : indel penalty

Scoring Matrices

To generalize scoring, consider a (4+1) x(4+1) **scoring matrix** δ for nucleotides {A,C,T,G}.

In the case of an amino acid sequence alignment, the scoring matrix would be a (20+1)x(20+1) size.

The addition of 1 is to include the score for comparison of a gap character "-".

This will simplify the algorithm as follows:

$$s_{i,j} = max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Making a Scoring Matrix

- Scoring matrices are created based on biological evidence.
- Alignments can be thought of as two sequences that differ due to mutations.
- Some of these mutations have little effect on the protein's function, therefore some penalties, $\delta(v_i , w_j)$, are less harsh than others.

Scoring Matrix: Example

	A	R	N	K
A	5	-2	-1	-1
R	-	7	-1	3
N	-	-	7	0
K	-	-	-	6

AKRANR

KAAANK

-1 + (-1) + (-2) + 5 + 7 + 3 = 11

- Notice that although R (arginine) and K (lysine) are different amino acids, they have a positive score.
- Why? They are both positively charged amino acids and hydrophillic → may not greatly change function of protein.

Conservation

- Amino acid changes that tend to preserve the electro-chemical properties of the original residue
  - Polar to polar
    - aspartate → glutamate
  - Nonpolar to nonpolar
    - alanine → valine
  - Similarly behaving residues
    - leucine to isoleucine

Scoring matrices

- Amino acid substitution matrices
  - PAM
  - BLOSUM

- DNA substitution matrices
  - DNA is less conserved than protein sequences
  - Less effective to compare coding regions at nucleotide level

PAM

- **P**oint **A**ccepted **M**utation (Dayhoff et al.)
  - based on observed differences in closely related proteins
- 1 PAM = PAM_1 = 1% average change of all amino acid positions
  - Unit of time
  - After 100 PAMs of evolution, not every residue will have changed
    - some residues may have mutated several times
    - some residues may have returned to their original state
    - some residues may not changed at all

PAM_X

- $PAM_x = PAM_1^x$
  - $PAM_{250} = PAM_1^{250}$
- PAM_{250} is a widely used scoring matrix:

	Ala A	Arg R	Asn N	Asp D	Cys C	Gln Q	Glu E	Gly G	His H	Ile I	Leu L	Lys K	...
Ala A	13	6	9	9	5	8	9	12	6	8	6	7	...
Arg R	3	17	4	3	2	5	3	2	6	3	2	9	
Asn N	4	4	6	7	2	5	6	4	6	3	2	5	
Asp D	5	4	8	11	1	7	10	5	6	3	2	5	
Cys C	2	1	1	1	52	1	1	2	2	2	1	1	
Gln Q	3	5	5	6	1	10	7	3	7	2	3	5	
...													
Trp W	0	2	0	0	0	0	0	0	1	0	1	0	
Tyr Y	1	1	2	1	3	1	1	1	3	2	2	1	
Val V	7	4	4	4	4	4	4	4	5	4	15	10	

BLOSUM

- **Bl**ocks **Su**bstitution **M**atrix
- Scores derived from *observations* of the frequencies of substitutions in blocks of local alignments in related proteins
- Matrix name indicates evolutionary distance
  - BLOSUM62 was created using sequences sharing no more than 62% identity

The Blosum50 Scoring Matrix

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	5	-2	-1	-2	-1	-1	-1	0	-2	-1	-2	-1	-1	-3	-1	1	0	-3	-2	0	-2	-1	-1	-5
R	-2	7	-1	-2	-4	1	0	-3	0	-4	-3	3	-2	-3	-3	-1	-1	-3	-1	-3	-1	0	-1	-5
N	-1	-1	7	2	-2	0	0	0	1	-3	-4	0	-2	-4	-2	1	0	-4	-2	-3	4	0	-1	-5
D	-2	-2	2	8	-4	0	2	-1	-1	-4	-4	-1	-4	-5	-1	0	-1	-5	-3	-4	5	1	-1	-5
C	-1	-4	-2	-4	13	-3	-3	-3	-3	-2	-2	-3	-2	-2	-4	-1	-1	-5	-3	-1	-3	-3	-2	-5
Q	-1	1	0	0	-3	7	2	-2	1	-3	-2	2	0	-4	-1	0	-1	-1	-1	-3	0	4	-1	-5
E	-1	0	0	2	-3	2	6	-3	0	-4	-3	1	-2	-3	-1	-1	-1	-3	-2	-3	1	5	-1	-5
G	0	-3	0	-1	-3	-2	-3	8	-2	-4	-4	-2	-3	-4	-2	0	-2	-3	-3	-4	-1	-2	-2	-5
H	-2	0	1	-1	-3	1	0	-2	10	-4	-3	0	-1	-1	-2	-1	-2	-3	2	-4	0	0	-1	-5
I	-1	-4	-3	-4	-2	-3	-4	-4	-4	5	2	-3	2	0	-3	-3	-1	-3	-1	4	-4	-3	-1	-5
L	-2	-3	-4	-4	-2	-2	-3	-4	-3	2	5	-3	3	1	-4	-3	-1	-2	-1	1	-4	-3	-1	-5
K	-1	3	0	-1	-3	2	1	-2	0	-3	-3	6	-2	-4	-1	0	-1	-3	-2	-3	0	1	-1	-5
M	-1	-2	-2	-4	-2	0	-2	-3	-1	2	3	-2	7	0	-3	-2	-1	-1	0	1	-3	-1	-1	-5
F	-3	-3	-4	-5	-2	-4	-3	-4	-1	0	1	-4	0	8	-4	-3	-2	1	4	-1	-4	-4	-2	-5
P	-1	-3	-2	-1	-4	-1	-1	-2	-2	-3	-4	-1	-3	-4	10	-1	-1	-4	-3	-3	-2	-1	-2	-5
S	1	-1	1	0	-1	0	-1	0	-1	-3	-3	0	-2	-3	-1	5	2	-4	-2	-2	0	0	-1	-5
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	2	5	-3	-2	0	0	-1	0	-5
W	-3	-3	-4	-5	-5	-1	-3	-3	-3	-3	-2	-3	-1	1	-4	-4	-3	15	2	-3	-5	-2	-3	-5
Y	-2	-1	-2	-3	-3	-1	-2	-3	2	-1	-1	-2	0	4	-3	-2	-2	2	8	-1	-3	-2	-1	-5
V	0	-3	-3	-4	-1	-3	-3	-4	-4	4	1	-3	1	-1	-3	-2	0	-3	-1	5	-4	-3	-1	-5
B	-2	-1	4	5	-3	0	1	-1	0	-4	-4	0	-3	-4	-2	0	0	-5	-3	-4	5	2	-1	-5
Z	-1	0	0	1	-3	4	5	-2	0	-3	-3	1	-1	-4	-1	0	-1	-2	-2	-3	2	5	-1	-5
X	-1	-1	-1	-1	-2	-1	-1	-2	-1	-1	-1	-1	-1	-2	-2	-1	0	-3	-1	-1	-1	-1	-1	-5
*****	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	-5	1

Local vs. Global Alignment

- The <u>Global Alignment Problem</u> tries to find the longest path between vertices *(0,0)* and (*n,m*) in the edit graph.

- The <u>Local Alignment Problem</u> tries to find the longest path among paths between **arbitrary vertices** (*i,j*) and (*i′, j′*) in the edit graph.

Local vs. Global Alignment

- The Global Alignment Problem tries to find the longest path between vertices *(0,0)* and (*n*,*m*) in the edit graph.

- The Local Alignment Problem tries to find the longest path among paths between **arbitrary vertices** (*i*,*j*) and (*i′*, *j′*) in the edit graph.
- In the edit graph with negatively-scored edges, Local Alignment may score higher than Global Alignment

Local vs. Global Alignment (cont'd)

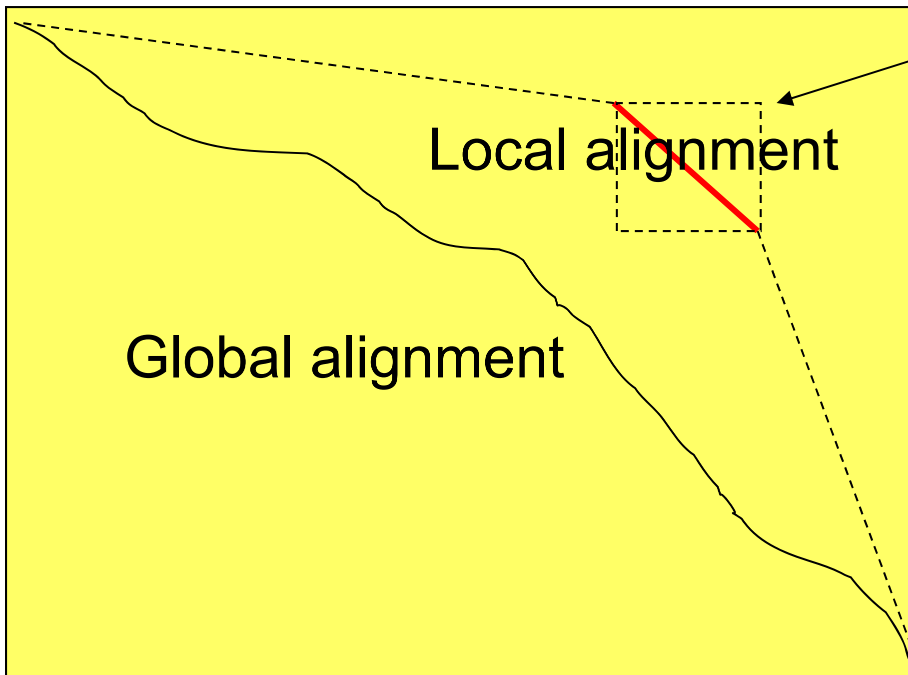
- Global Alignment

```
--T—-CC-C-AGT—-TATGT-CAGGGACACG—A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || | | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG—T-CAGAT--C
```

- Local Alignment—better alignment to find conserved segment

```
                         tccCAGTTATGTCAGgggacacgagcatgcagagac
                            ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

Local Alignment: Example



Compute a "mini" Global Alignment to get Local

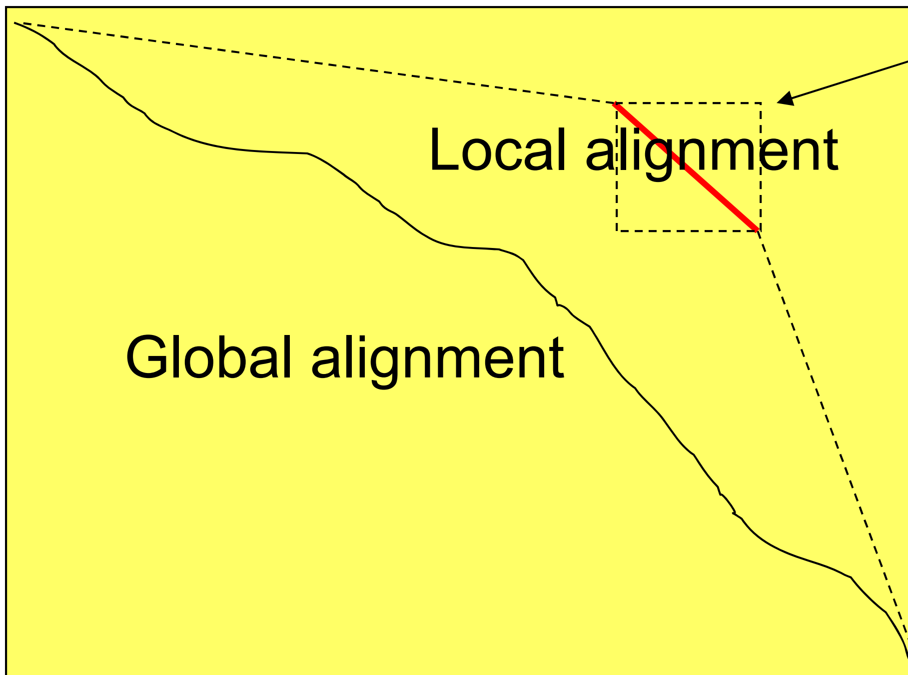
Local Alignments: Why?

- Two genes in different species may be similar over short conserved regions and dissimilar over remaining regions.
- Example:
  - Homeobox genes have a short region called the *homeodomain* that is highly conserved between species.
  - A global alignment would not find the homeodomain because it would try to align the ENTIRE sequence

The Local Alignment Problem

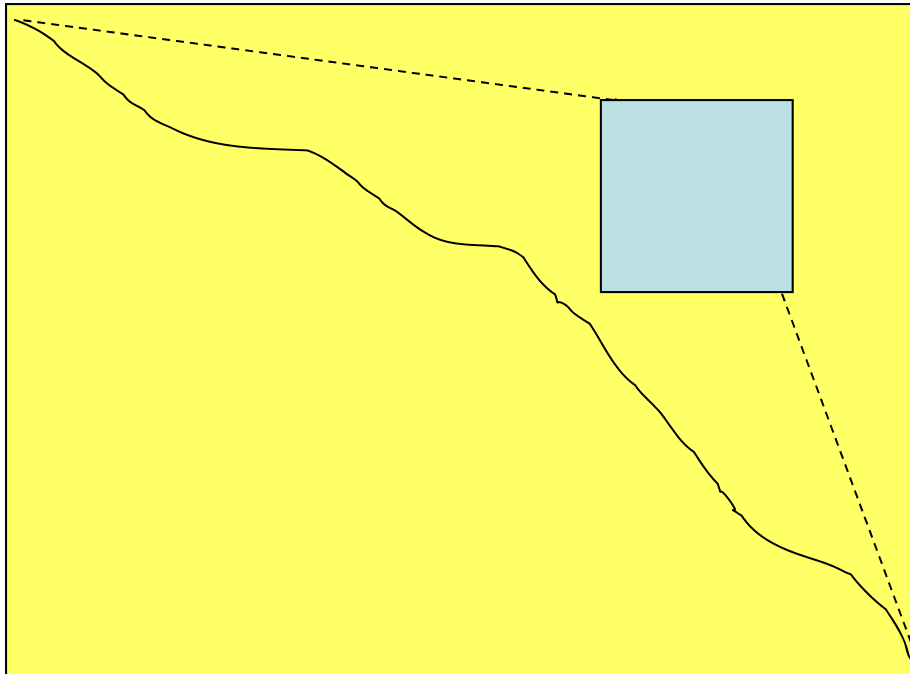
- Goal: Find the best local alignment between two strings
- Input : Strings **v, w** and scoring matrix δ
- Output : Alignment of *substrings* of **v** and **w** whose alignment score is maximum among all possible alignment of all possible substrings

Local Alignment: Example

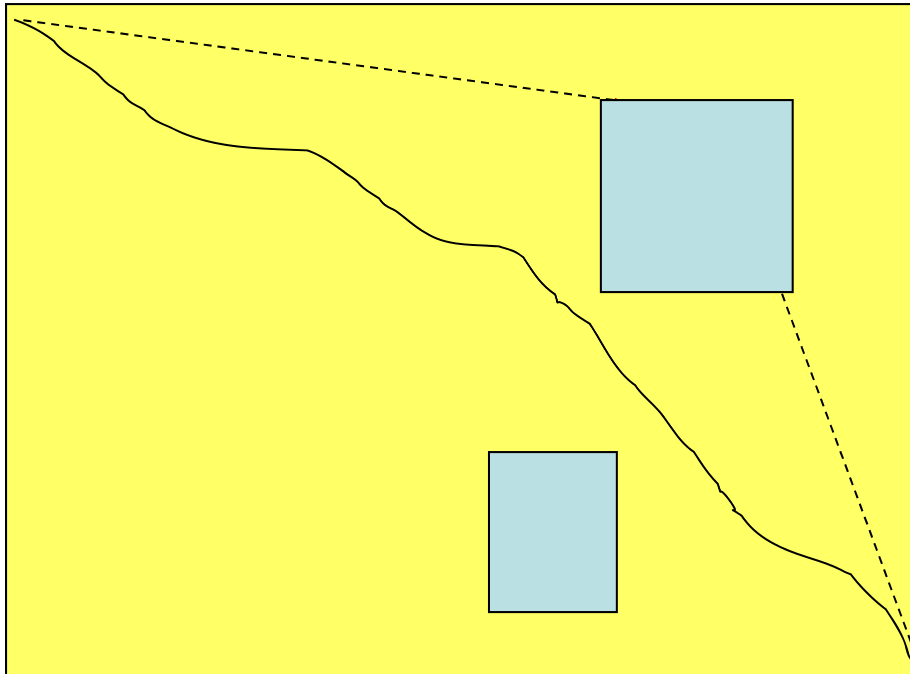


Compute a “mini” Global Alignment to get Local

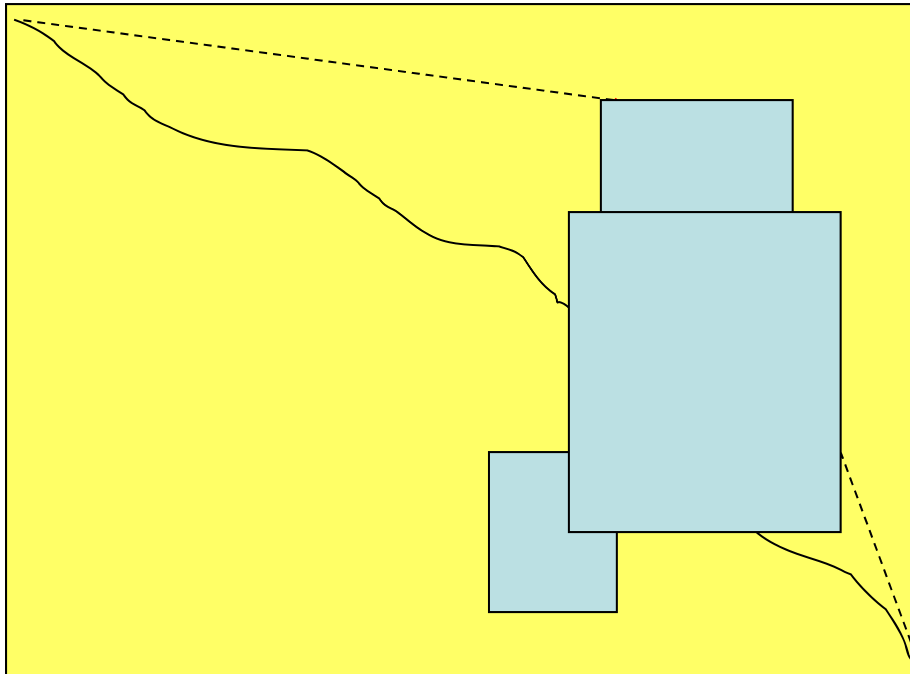
Local Alignment: Example



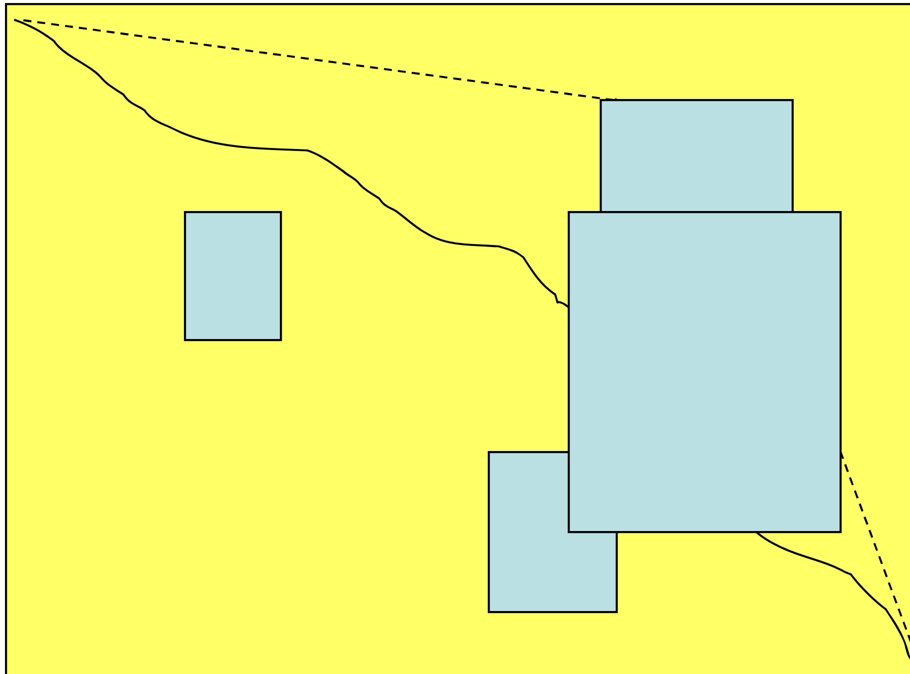
Local Alignment: Example



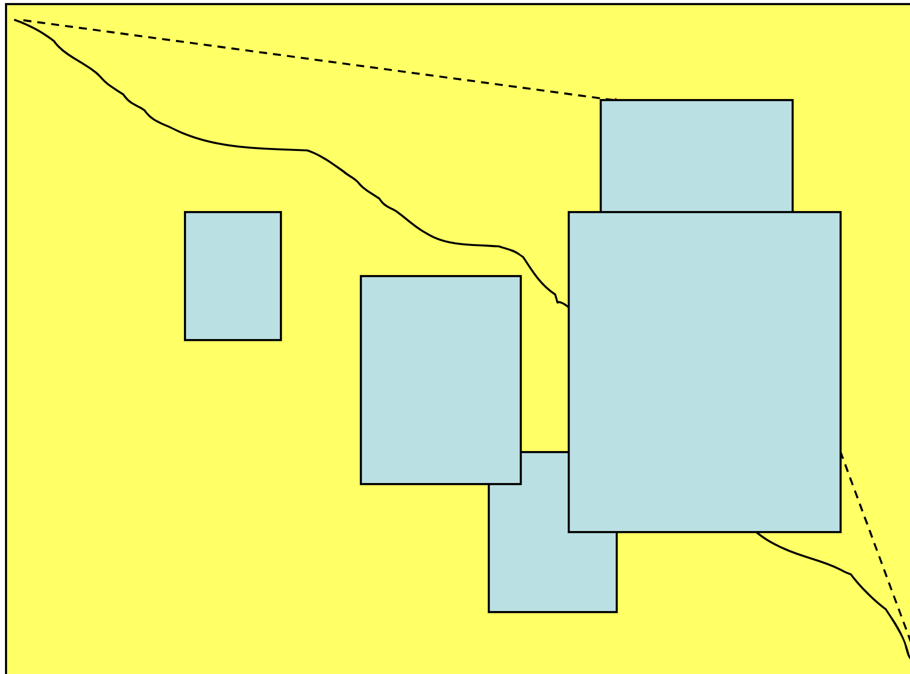
Local Alignment: Example



Local Alignment: Example



Local Alignment: Example

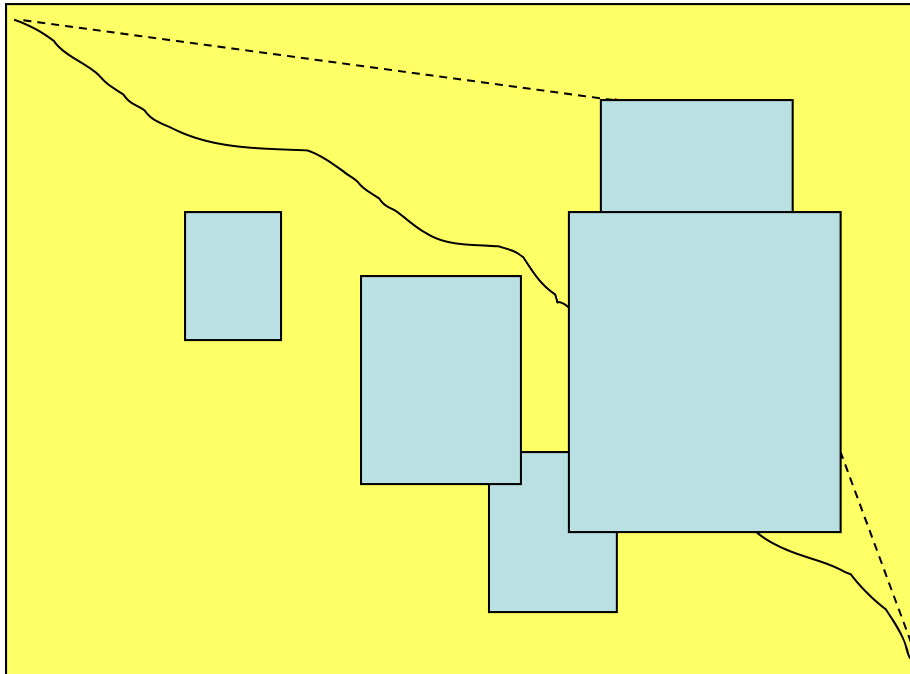


Local Alignment: Running Time

- Long run time $O(n^4)$:

  - In the grid of size n x n there are $\sim n^2$ vertices (i,j) that may serve as a source.
  - For each such vertex computing alignments from (i,j) to (i',j') takes $O(n^2)$ time

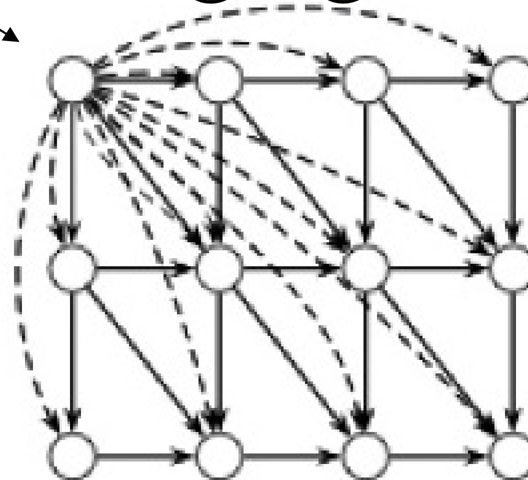
- We can do better by building "free rides" into the score



Local Alignment: Free Rides

Yeah, a free ride!

Vertex (0,0)



The dashed edges represent the free rides from (0,0) to every other node.

The Local Alignment Recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.

- The recurrence:

$$s_{i,j} = max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Notice there is only this change from the original recurrence of a Global Alignment

The Local Alignment Recurrence

- The largest value of $s_{i,j}$ over the whole edit graph is the score of the best local alignment.

- The recurrence:

$$s_{i,j} = max \begin{cases} 0 \\ s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, -) \\ s_{i,j-1} + \delta(-, w_j) \end{cases}$$

Power of ZERO: there is only this change from the original recurrence of a Global Alignment - since there is only one “free ride” edge entering into every vertex

Next Time

- We finish Dynamic programming
- Alignment with Gap Penalities
- Multiple Alignment problem
- Gene Prediction
  - Statistical Approaches
  - Similarity Approaches
- Splice Alignments