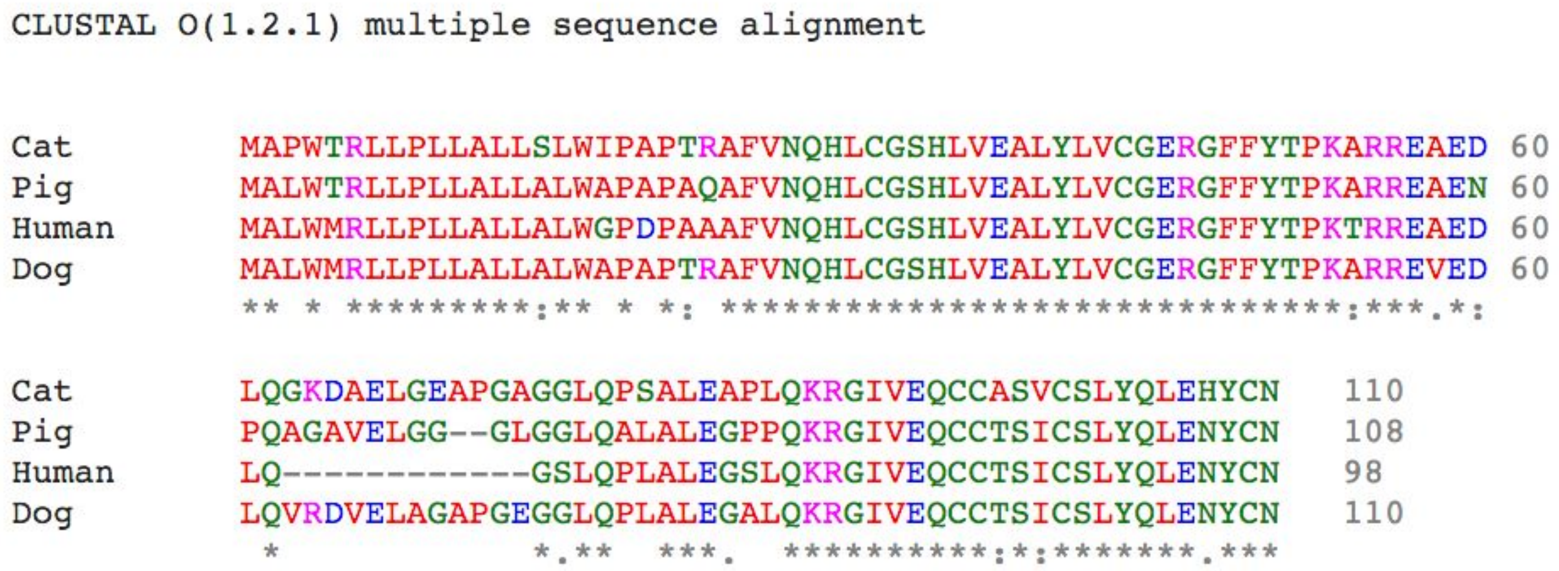
Comp 555 - BioAlgorithms - Spring 2018



```
CLUSTAL O(1.2.1) multiple sequence alignment


Cat       MAPWTRLLPLLALLSLWIPAPTRAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREAED 60
Pig       MALWTRLLPLLALLALWAPAPAQAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREAEN 60
Human     MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAED 60
Dog       MALWMRLLPLLALLALWAPAPTRAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREVED 60
          ** * *********:** * *: ******************************:***.*:

Cat       LQGKDAELGEAPGAGGLQPSALEAPLQKRGIVEQCCASVCSLYQLEHYCN   110
Pig       PQAGAVELGG--GLGGLQALALEGPPQKRGIVEQCCTSICSLYQLENYCN   108
Human     LQ------------GSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN   98
Dog       LQVRDVELAGAPGEGGLQPLALEGALQKRGIVEQCCTSICSLYQLENYCN   110
           *            *.**  ***.  **********:*:*******.***
```

Advanced Sequence Alignment

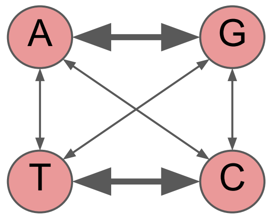
Alignment with a Scoring Matrix

- Rather *edit distance* one could use a table with costs for every symbol aligned to any other
- Scoring matrices allow alignments to consider biological constraints
- Alignments can be thought of as two sequences that differ due to mutations.
- Some types of mutations are more common, or have little or no effect on function, therefore some mismatch penalties, $\delta(v_i, w_j)$, should be less harsh than others.

Example: ***DNA transitions and transversions***

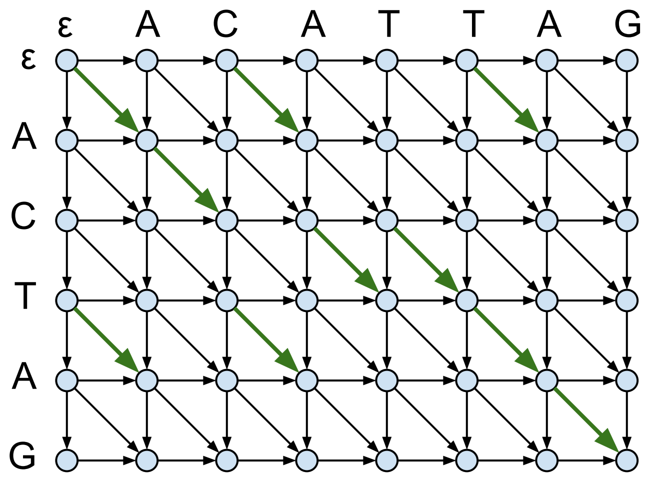
- Like LCS, we want to maximize sequence matches, so each should have a positive score (diagonal of scoring matrix)
- Unlike LCS, we need to allow for occasional mismatches, as well as INDELs.
- The 4 DNA nucleotides come in two types, *purines* (A and G), which have two-rings and *pyrimidines*, (C and T) which have only one.
- Mutations within types are far more common than mutations between types, despite there being twice as many. This higher mutation rate can be encoded as a smaller substitution penalty.
- Insertions and deletions are even less common that any substitution, thus they have even higher penalties.

δ	A	C	G	T	_
A	1	-2	-1	-2	-3
C	-2	1	-2	-1	-3
G	-1	-2	1	-2	-3
T	-1	-1	-2	1	-3
_	-3	-3	-3	-3	



Impact on Alignment

Graph includes all diagonal edges, but many with negative weights

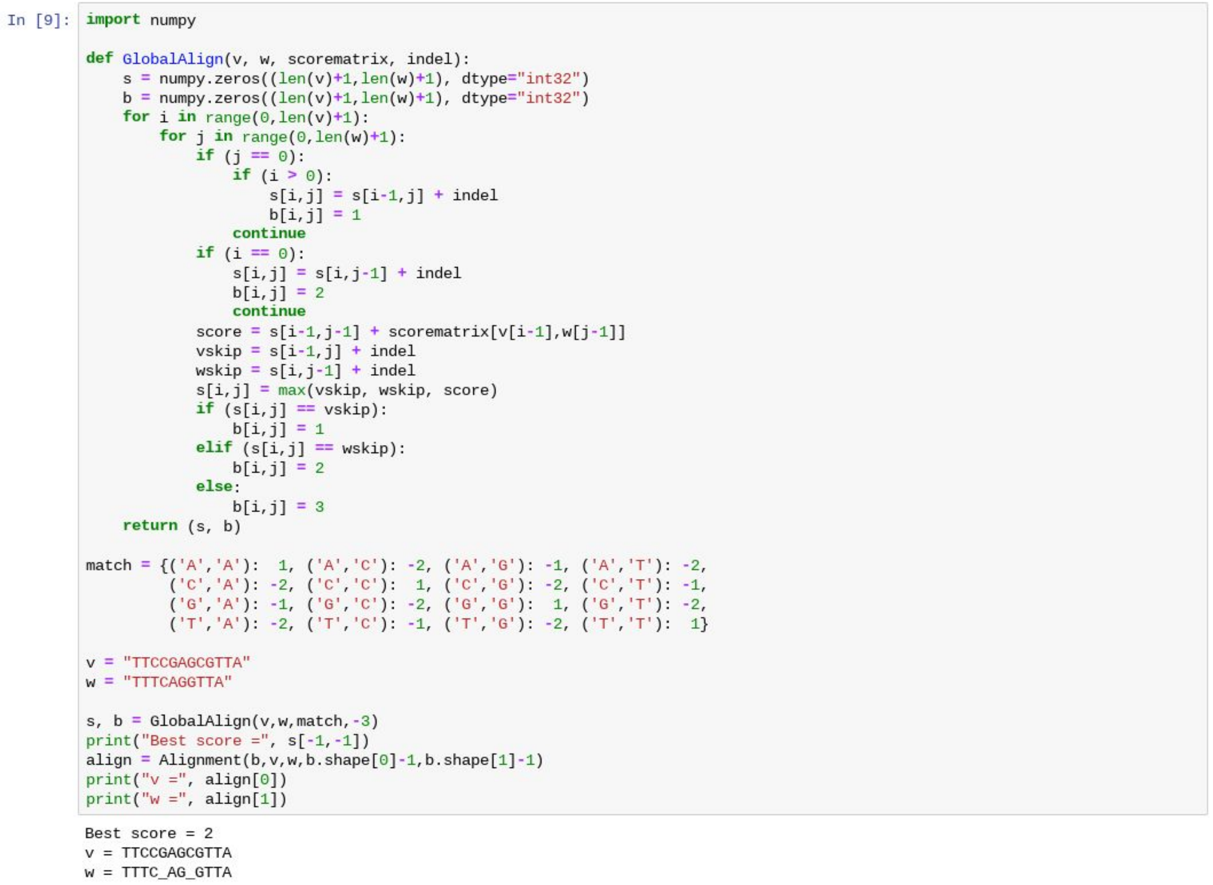


$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ s_{i-1,j} + \delta(v_i, \_) \\ s_{i-1,j} + \delta(\_, w_j) \end{cases}$$

Generalized recurrence relation

Needleman–Wunsch Alignment Algorithm

Global Alignment with a scoring matrix



```
In [9]: import numpy

        def GlobalAlign(v, w, scorematrix, indel):
            s = numpy.zeros((len(v)+1,len(w)+1), dtype="int32")
            b = numpy.zeros((len(v)+1,len(w)+1), dtype="int32")
            for i in range(0,len(v)+1):
                for j in range(0,len(w)+1):
                    if (j == 0):
                        if (i > 0):
                            s[i,j] = s[i-1,j] + indel
                            b[i,j] = 1
                        continue
                    if (i == 0):
                        s[i,j] = s[i,j-1] + indel
                        b[i,j] = 2
                        continue
                    score = s[i-1,j-1] + scorematrix[v[i-1],w[j-1]]
                    vskip = s[i-1,j] + indel
                    wskip = s[i,j-1] + indel
                    s[i,j] = max(vskip, wskip, score)
                    if (s[i,j] == vskip):
                        b[i,j] = 1
                    elif (s[i,j] == wskip):
                        b[i,j] = 2
                    else:
                        b[i,j] = 3
            return (s, b)

        match = {('A','A'):  1, ('A','C'): -2, ('A','G'): -1, ('A','T'): -2,
                 ('C','A'): -2, ('C','C'):  1, ('C','G'): -2, ('C','T'): -1,
                 ('G','A'): -1, ('G','C'): -2, ('G','G'):  1, ('G','T'): -2,
                 ('T','A'): -2, ('T','C'): -1, ('T','G'): -2, ('T','T'):  1}

        v = "TTCCGAGCGTTA"
        w = "TTTCAGGTTA"

        s, b = GlobalAlign(v,w,match,-3)
        print("Best score =", s[-1,-1])
        align = Alignment(b,v,w,b.shape[0]-1,b.shape[1]-1)
        print("v =", align[0])
        print("w =", align[1])
```

```
Best score = 2
v = TTCCGAGCGTTA
w = TTTC_AG_GTTA
```

Local vs. Global Alignment

- The *Global Alignment Problem* tries to find the highest scoring path between vertices (0,0) and (n,m) in the edit graph.
- The *Local Alignment Problem* tries to find the highest scoring subpath between *all vertex pairs* (i_1,j_1) and (i_2,j_2) in the edit graph where $i_2 > i_1$ and $j_2 > j_1$.
- In an edit graph with negatively-weighted scores, a Local Alignment may score higher than a Global Alignment

Example:

- Global Alignment finds a match for the entire sequence

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

- Local Alignment finds a long conserved subsequence

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

Local Alignments: Why?

Two genes in different species may be similar over short conserved regions and dissimilar over remaining regions.

Example:

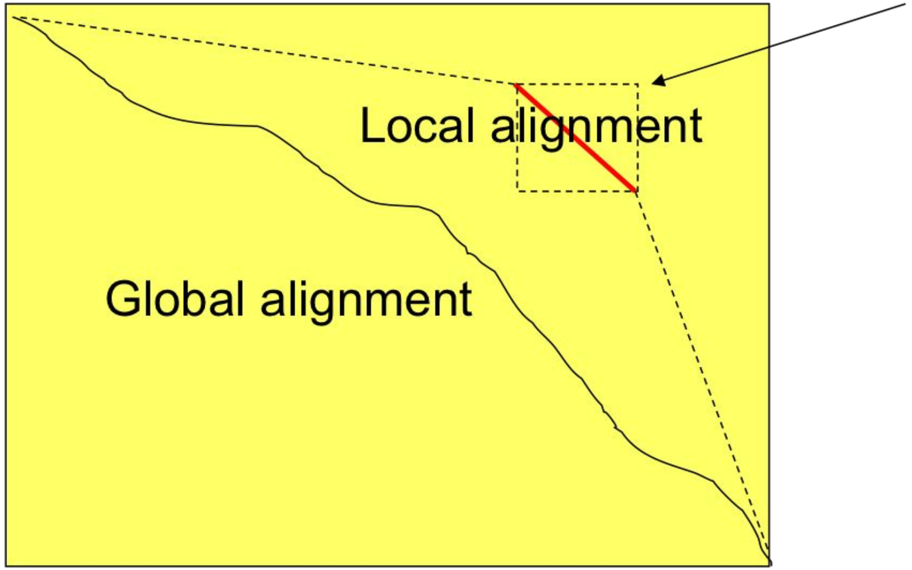
- Homeobox genes have a short region called the homeodomain that is highly conserved between species.
- A global alignment would not find the homeodomain because it would try to align the ENTIRE sequence

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 Approach

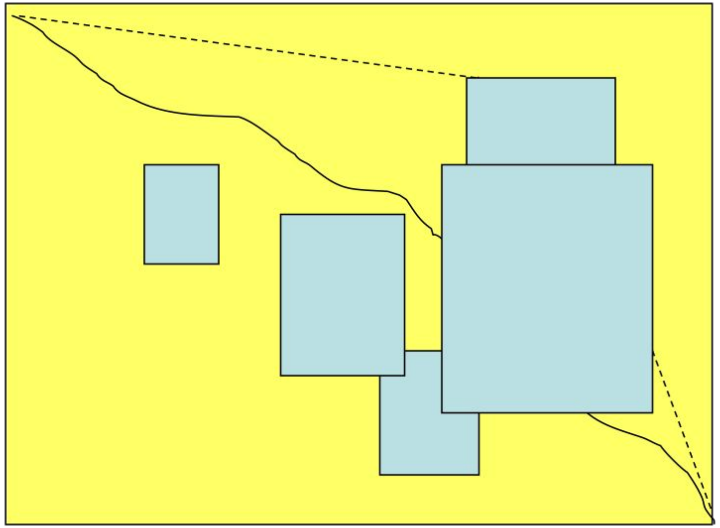
A local alignment is a subpath in a global alignment



Compute a "mini" Global Alignment to get Local

Brute Force Local Alignment

Find the best global alignment among all blocks *(i_1,j_1,i_2,j_2)*

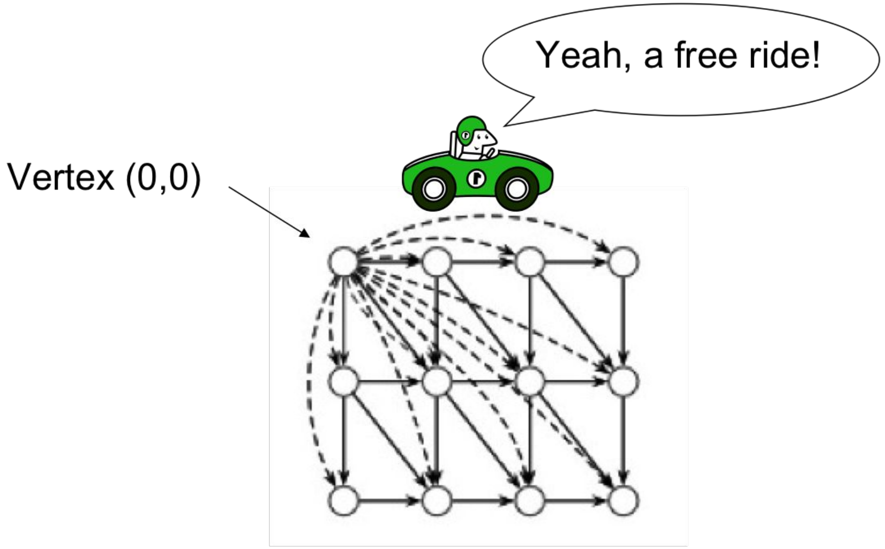


Long run time $O(n^4)$:

- In the grid of size n x n there are $O(n^2)$ vertices (i_1,j_1) that may serve as a source.
- For each such vertex computing alignments from (i_1,j_1) to (i_2,j_2) takes $O(n^2)$ time.

Local Alignment with Free Rides

- ***Key Ideas:*** Add extra edges to our graph, consider all scores in matrix



- The dashed edges represent a *free ride* from (0,0) to any other node
- The largest value of $s_{i,j}$ over the *whole score matrix* is the end point of the best local alignment (instead of $s_{n,m}$)

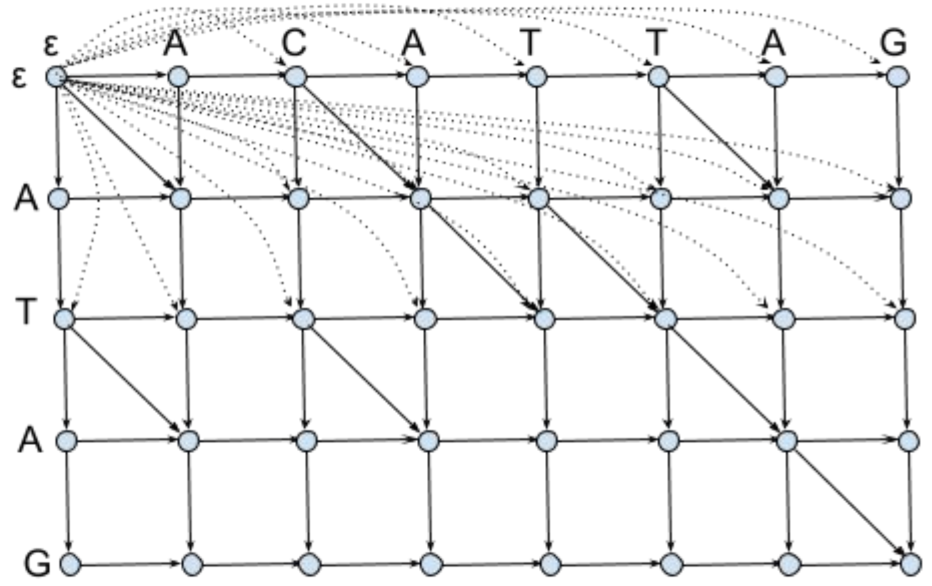
Local Alignment Recurrence

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

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

- The *zero* is our *free ride* that allows the node to restart with a score of 0 at any point
  - What does this imply?
- After solving for the entire score matrix, we then search for si,j with the highest score, this is (i_2,j_2)
- We follow our back tracking matrix until we reach a *score* of 0, whose coordinate becomes (i_1,i_1)

Smith-Waterman Local Alignment



Key Idea: Add edges from the source to any intersection. These free rides might be better than any other path reaching an intersection.

Local Alignment Example

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0												
2	C	0												
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

Match = 5, Mismatch = -4, Indel = -7

Local Alignment Example - continued

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	$S_{1,1}$											
2	C	0												
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

$$S_{1,1} = \max \begin{Bmatrix} S_{0,0} + s_{G,G} = 0 + 5 = 5 \\ S_{1,0} + w \;= 0 - 7 = -7 \\ S_{0,1} + w \;= 0 - 7 = -7 \\ 0 \end{Bmatrix} = 5$$

Match = 5, Mismatch = -4, Indel = -7

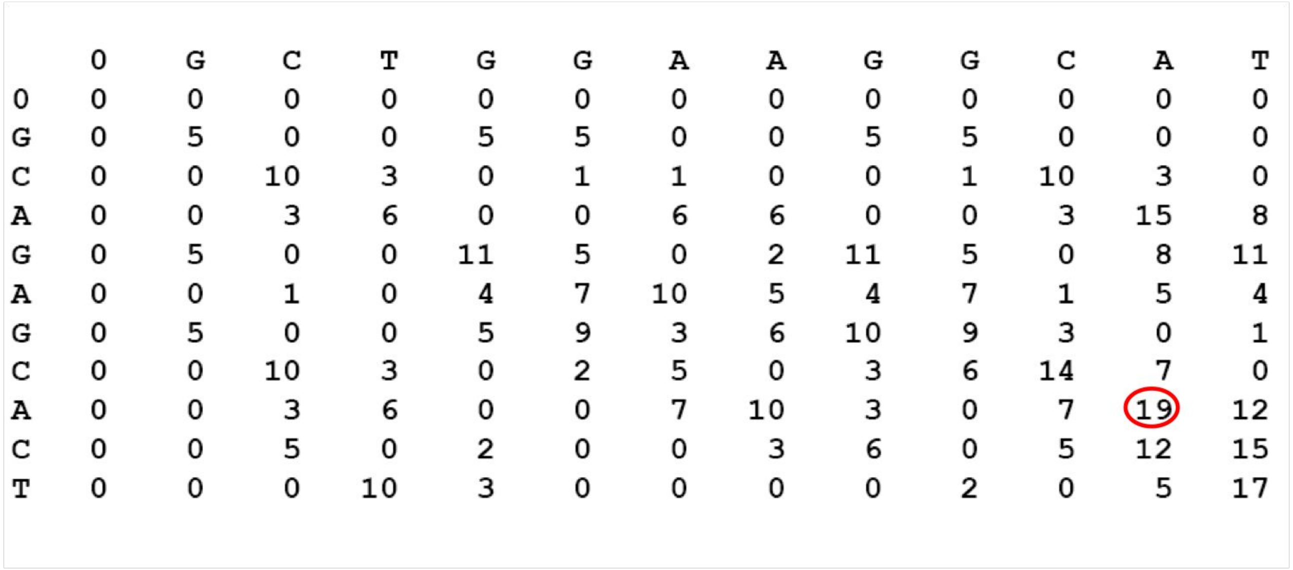
Local Alignment Example - continued

		j=0	1	2	3	4	5	6	7	8	9	10	11	12
i=		-	G	C	T	G	G	A	A	G	G	C	A	T
0	-	0	0	0	0	0	0	0	0	0	0	0	0	0
1	G	0	5	0										
2	C	0	0	$S_{2,2}$										
3	A	0												
4	G	0												
5	A	0												
6	G	0												
7	C	0												
8	A	0												
9	C	0												
10	T	0												

$$S_{2,2} = \max\begin{Bmatrix} S_{1,1} + s_{C,C} = 5 + 5 = 10 \\ S_{2,1} + w \;= 0 - 7 = -7 \\ S_{1,2} + w \;= 0 - 7 = -7 \\ 0 \end{Bmatrix} = 10$$

Match = 5, Mismatch = -4, Indel = -7

Local Alignment Example - continued

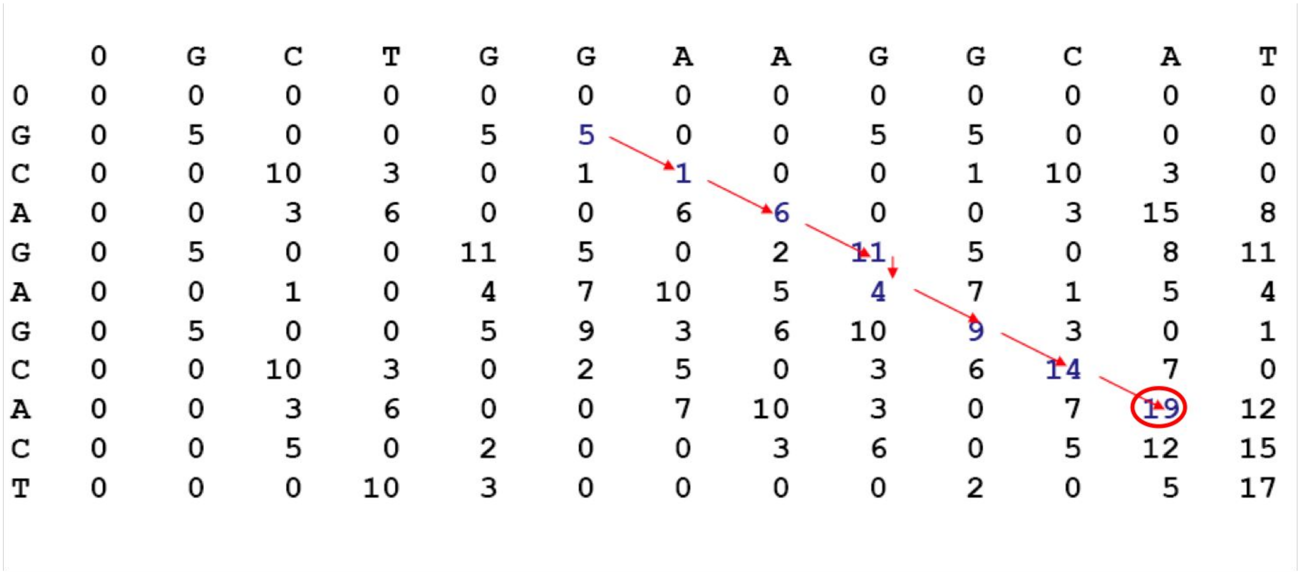


	0	G	C	T	G	G	A	A	G	G	C	A	T
0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	0	0	5	5	0	0	5	5	0	0	0
C	0	0	10	3	0	1	1	0	0	1	10	3	0
A	0	0	3	6	0	0	6	6	0	0	3	15	8
G	0	5	0	0	11	5	0	2	11	5	0	8	11
A	0	0	1	0	4	7	10	5	4	7	1	5	4
G	0	5	0	0	5	9	3	6	10	9	3	0	1
C	0	0	10	3	0	2	5	0	3	6	14	7	0
A	0	0	3	6	0	0	7	10	3	0	7	19	12
C	0	0	5	0	2	0	0	3	6	0	5	12	15
T	0	0	0	10	3	0	0	0	0	2	0	5	17

Match = 5, Mismatch = -4, Indel = -7

- Once the matrix is filled in we find the best alignment
- Rather than using the score of the last entry as we did for a global alignment, we search for the entire matrix for the maximum entry (*O(m n)* steps)

Local Alignment Example - continued



	0	G	C	T	G	G	A	A	G	G	C	A	T
0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	5	0	0	5	5	0	0	5	5	0	0	0
C	0	0	10	3	0	1	1	0	0	1	10	3	0
A	0	0	3	6	0	0	6	6	0	0	3	15	8
G	0	5	0	0	11	5	0	2	11	5	0	8	11
A	0	0	1	0	4	7	10	5	4	7	1	5	4
G	0	5	0	0	5	9	3	6	10	9	3	0	1
C	0	0	10	3	0	2	5	0	3	6	14	7	0
A	0	0	3	6	0	0	7	10	3	0	7	19	12
C	0	0	5	0	2	0	0	3	6	0	5	12	15
T	0	0	0	10	3	0	0	0	0	2	0	5	17

Match = 5, Mismatch = -4, Indel = -7

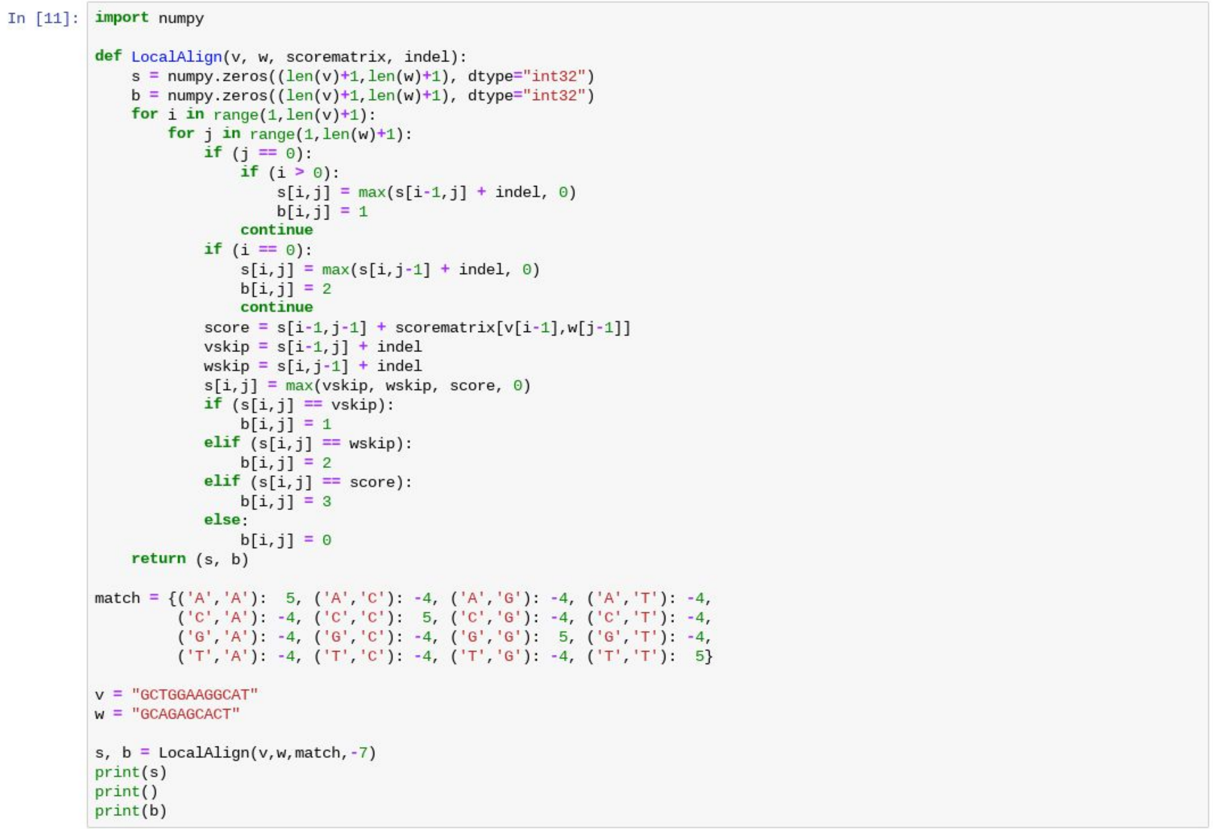
- From the largest score attained, then backtrack from there until a beginning "0" is reached to find the alignment.

Local Alignment Example - continued

```
G C T G G A A G - G C A T
        |   | |   | | |
        G C A G A G C A C T
```

6 matches: 6 × 5 = 30
1 mismatch: -4
1 indel: -7
Total: 19

Local Alignment Code



```
In [11]: import numpy

         def LocalAlign(v, w, scorematrix, indel):
             s = numpy.zeros((len(v)+1,len(w)+1), dtype="int32")
             b = numpy.zeros((len(v)+1,len(w)+1), dtype="int32")
             for i in range(1,len(v)+1):
                 for j in range(1,len(w)+1):
                     if (j == 0):
                         if (i > 0):
                             s[i,j] = max(s[i-1,j] + indel, 0)
                             b[i,j] = 1
                         continue
                     if (i == 0):
                         s[i,j] = max(s[i,j-1] + indel, 0)
                         b[i,j] = 2
                         continue
                     score = s[i-1,j-1] + scorematrix[v[i-1],w[j-1]]
                     vskip = s[i-1,j] + indel
                     wskip = s[i,j-1] + indel
                     s[i,j] = max(vskip, wskip, score, 0)
                     if (s[i,j] == vskip):
                         b[i,j] = 1
                     elif (s[i,j] == wskip):
                         b[i,j] = 2
                     elif (s[i,j] == score):
                         b[i,j] = 3
                     else:
                         b[i,j] = 0
             return (s, b)

         match = {('A','A'):  5, ('A','C'): -4, ('A','G'): -4, ('A','T'): -4,
                  ('C','A'): -4, ('C','C'):  5, ('C','G'): -4, ('C','T'): -4,
                  ('G','A'): -4, ('G','C'): -4, ('G','G'):  5, ('G','T'): -4,
                  ('T','A'): -4, ('T','C'): -4, ('T','G'): -4, ('T','T'):  5}

         v = "GCTGGAAGGCAT"
         w = "GCAGAGCACT"

         s, b = LocalAlign(v,w,match,-7)
         print(s)
         print()
         print(b)
```

```
[[ 0  0  0  0  0  0  0  0  0  0  0]
 [ 0  5  0  0  5  0  5  0  0  0  0]
 [ 0  0 10  3  0  1  0 10  3  5  0]
 [ 0  0  3  6  0  0  0  3  6  0 10]
 [ 0  5  0  0 11  4  5  0  0  2  3]
 [ 0  5  1  0  5  7  9  2  0  0  0]
 [ 0  0  1  6  0 10  3  5  7  0  0]
 [ 0  0  0  6  2  5  6  0 10  3  0]
 [ 0  5  0  0 11  4 10  3  3  6  0]
 [ 0  5  1  0  5  7  9  6  0  0  2]
 [ 0  0 10  3  0  1  3 14  7  5  0]
 [ 0  0  3 15  8  5  0  7 19 12  5]
 [ 0  0  0  8 11  4  1  0 12 15 17]]

[[0 0 0 0 0 0 0 0 0 0 0]
 [0 3 0 0 3 0 3 0 0 0 0]
 [0 0 3 2 0 3 0 3 2 3 0]
 [0 0 1 3 0 0 0 1 3 0 3]
 [0 3 0 0 3 2 3 0 0 3 1]
 [0 3 3 0 3 3 3 2 0 0 0]
 [0 0 3 3 0 3 2 3 3 2 0]
 [0 0 0 3 3 3 3 0 3 2 0]
 [0 3 0 0 3 2 3 2 1 3 0]
 [0 3 3 0 3 3 3 3 0 0 3]
 [0 0 3 2 0 3 3 3 2 3 0]
 [0 0 1 3 2 3 0 1 3 2 2]
 [0 0 0 1 3 2 3 1 1 3 3]]
```

Scoring Indels: Naive Approach

```
ATCTTCAGCCATAAAAGATGAAGTT        Reference
ATCTTCAGCCAAAGATGAAGTT           3 base deletion relative to the reference

ATCTTCAGCC---AAAGATGAAGTT        version 1
ATCTTCAGCCA---AAGATGAAGTT        version 2
ATCTTCAGCCA--A-AGATGAAGTT        version 3
ATCTTCAGCCA-AA--GATGAAGTT        version 4
ATCTTCAGCCA-A-A-GATGAAGTT        version 5

ATCTTCAGCCATATGTGAAAGATGAAGTT    4 base insertion
```

- A fixed penalty σ is given to every indel:
  - -σ for 1 indel,
  - -2σ for 2 consecutive indels
  - -3σ for 3 consecutive indels, etc.
- Can be too severe penalty for a series of 100 consecutive indels
  - large insertions or deletions might result from a single event

Affine Gap Penalties

In nature, a series of k indels often come as a single, albeit rare, event rather than as a series of muliple events

```
AT___GC          A_TG__C
ATTGAGC          ATTGAGC
```

This is more likely. Explained by one event

Normal scoring would give the same score for both alignments

This is less likely. Requires 2 events.

Accounting for Gaps

- Gaps- contiguous sequence of indels in a row
- Modify the scoring for a gap of length x to be:

$$-(\rho + \sigma x)$$

- where $\rho+\sigma > 0$ is the penalty for introducing a gap:

ρ = gap opening penalty

- and σ is the cost of extending it further ($\rho+\sigma \gg \sigma$):

σ = gap extension penalty

- because you do not want to add too much of a penalty for further extending the gap, once it is opened.

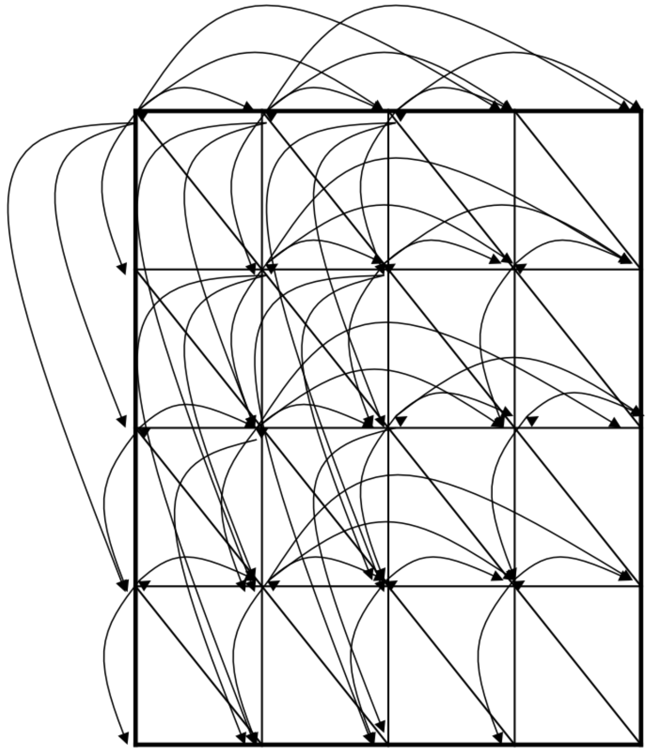
Adding Affine Gap Penalties to our Graph

- To reflect affine gap penalties we have to add "long" horizontal and vertical edges to the edit graph.
- Each such edge of length x should have weight

$$-\rho - x \cdot \sigma$$

- There are many such edges!
- Adding them to the graph increases the running time of the alignment algorithm by a factor of n (where n is the number of vertices)
- So the complexity increases from $O(n^2)$ to $O(n^3)$

Can we do it some other way?



Adding Two More Tables

Affine Gap penalties can be more easily expressed in terms of 3 recurrences

Keep track of these intermediate values in two new tables

$$t_{i,j} = \max \begin{cases} t_{i-1,j} - \sigma \\ s_{i-1,j} - (\rho + \sigma) \end{cases}$$

Continue Gap in *w* (deletion)

Start Gap in *w* (deletion): from middle

$$u_{i,j} = \max \begin{cases} u_{i,j-1} - \sigma \\ s_{i,j-1} - (\rho + \sigma) \end{cases}$$

Continue Gap in *v* (insertion)

Start Gap in *v* (insertion):from middle

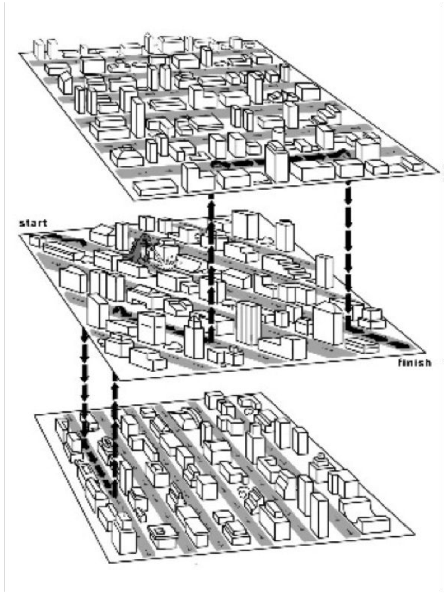
$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) \\ t_{i,j} \\ u_{i,j} \end{cases}$$

Match or Mismatch

End deletion: from top

End insertion: from left

A 3-level Manhattan Grid



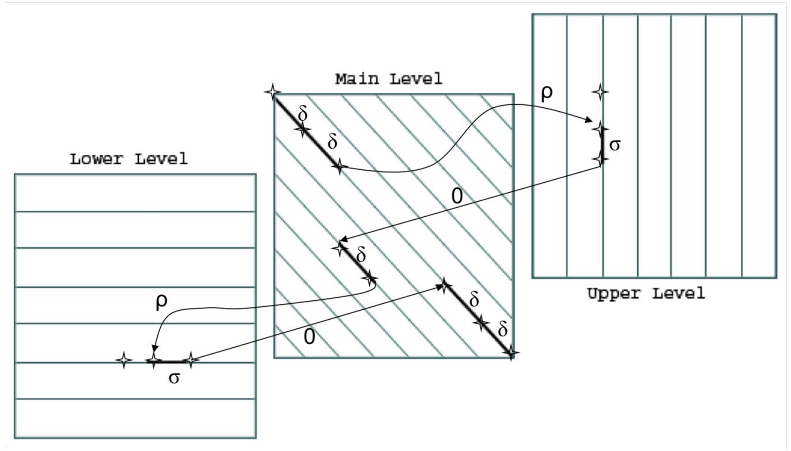
Gaps in w (t-table)

Matches/Mismatches (s-table)

Gaps in v (u-table)

- The three recurrences for the scoring algorithm creates a 3-layered graph.
- The top level creates/extends gaps in the sequence *w*.
- The bottom level creates/extends gaps in sequence *v*.
- The middle level extends matches and mismatches.

Switching between Layers



- Levels:
  - The main level is for diagonal edges
  - The lower level is for horizontal edges
  - The upper level is for vertical edges
- A jumping penalty is assigned to moving from the main level to either the upper level or the lower level (-ρ - σ)
- There is a gap extension penalty for each continuation on a level other than the main level (-σ)

Multiple versus Pairwise Alignment

- Up until now we have only tried to align two sequences.
- What about more than two? And why?
- A faint similarity between two sequences becomes significant if present in many
- Multiple alignments can reveal subtle similarities that pairwise alignments do not reveal

Generalizing Pairwise Alignment

- Alignment of 2 sequences is represented as a 2-row matrix
- In a similar way, we represent alignment of 3 sequences as a 3-row matrix

```
A T _ G C G _
A _ C G T _ A
A T C A C _ A
```

- Score: more conserved columns, better alignment

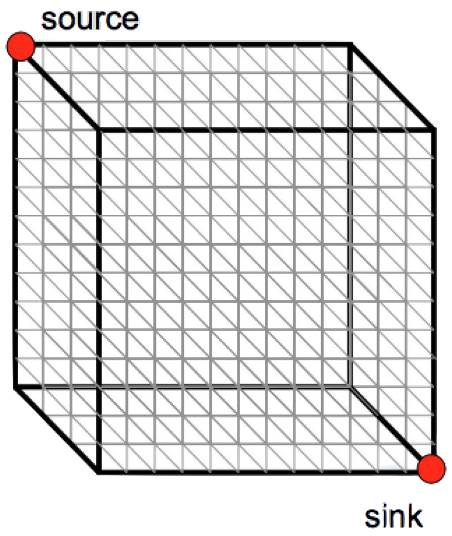
Three-D Alignment Paths

- An alignment of 3 sequences: ATGC, AATC, ATGC

0	1	1	2	3	4	*x* coordinate
	A	--	T	G	C	
0	1	2	3	3	4	*y* coordinate
	A	A	T	--	C	
0	0	1	2	3	4	*z* coordinate
	--	A	T	G	C	

- Resulting path in (x,y,z) space:
  $(0,0,0) \rightarrow (1,1,0) \rightarrow (1,2,1) \rightarrow (2,3,2) \rightarrow (3,3,3) \rightarrow (4,4,4)$
- Is there a better one?

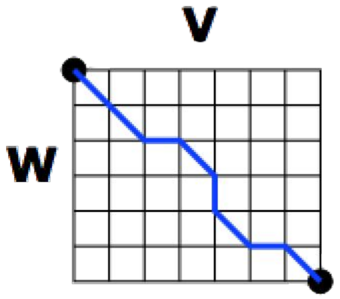
Aligning Three Sequences



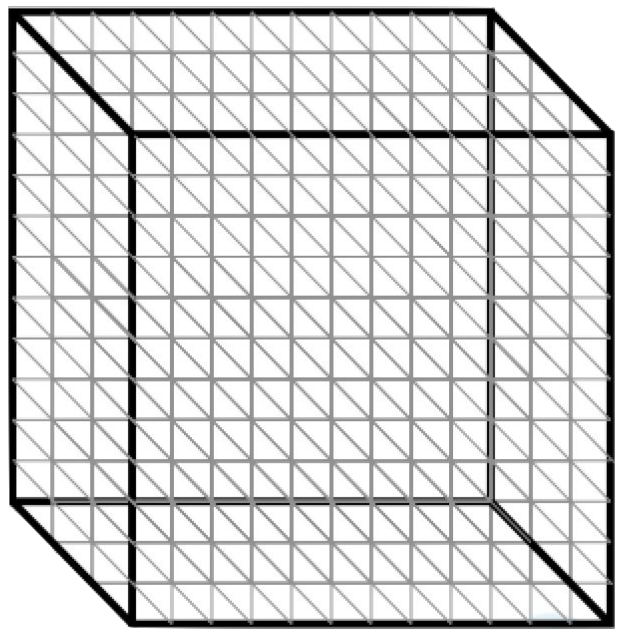
- Same strategy as aligning two sequences
- Use a 3-D “Manhattan Cube”, with each axis representing a sequence to align
- For global alignments, go from source to sink

2-sequence vs 3-sequence Alignment

- In a 2-D grid there are 3 approaches to each intersection
- I'm now ignoring
  - Free-passes
  - Affine jumps
- How about 3-D?
- How does this impact our recurrence relations?

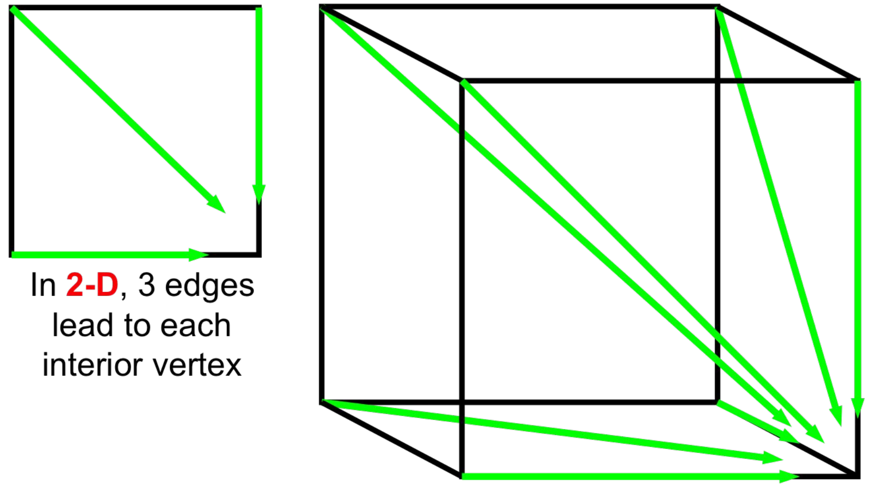


2-D edit graph



3-D edit graph

A 2-D versus a 3-D neighborhood


In **2-D**, 3 edges lead to each interior vertex

In **3-D**, 7 edges lead to each interior vertex

- 2-D *[(i-1,j-1), (i-1,j), (i,j-1)]* → *(i,j)* (3 directions)
- 3-D *[(i-1,j-1,k-1), (i-1,j,k), (i,j-1,k), (i,j,k-1), (i,j-1,k-1), (i-1,j,k-1), (i-1,j-1,k),]* → *(i,j,k)* (7 directions)
- N-D (2^N -1 directions)

Structure of a 3-D Alignment Cell

There are three path types

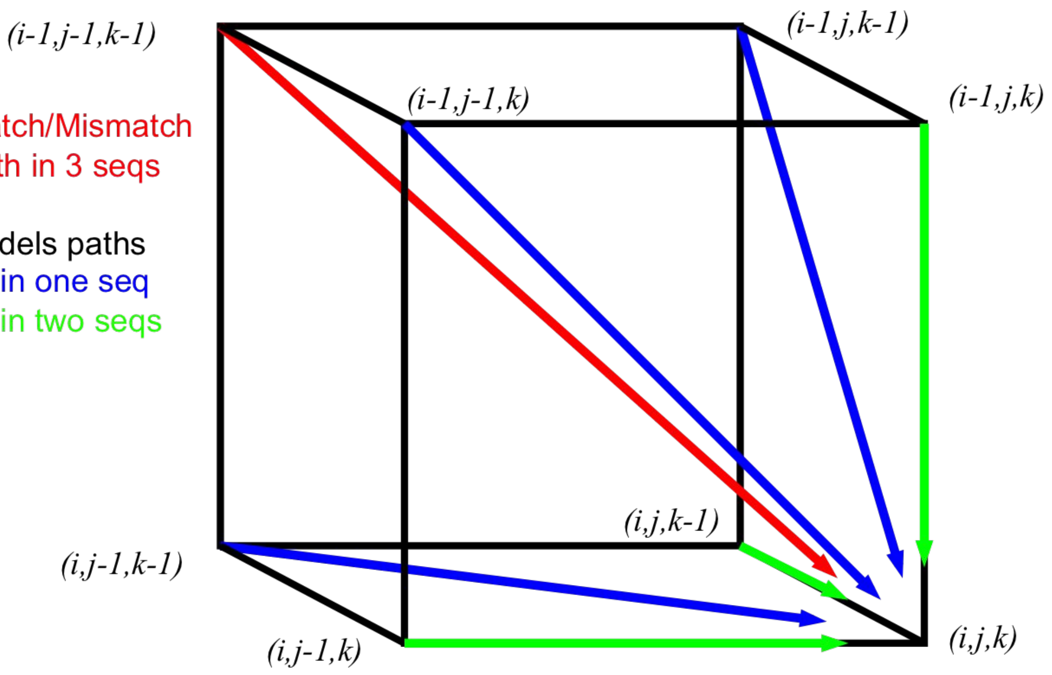
1. Consume a character in all 3 sequences (diagonal "red" edge)

2. Consumes characters in 2 of 3 sequences ("blue" diagonals on faces)

3. Consuumes a character from only one sequence ("green" edges"

1- Match/Mismatch Path in 3 seqs

6 – indels paths
3 in one seq
3 in two seqs



Multiple Alignment: Recursion Relation

- $$s_{i,j,k} = \max \begin{cases} s_{i-1,j-1,k-1} + \delta(v_i, w_j, u_k) & \text{cube diagonal: no indels} \\ s_{i-1,j-1,k} + \delta(v_i, w_j, \_) & \\ s_{i-1,j,k-1} + \delta(v_i, \_, u_k) & \text{face diagonal: one indel} \\ s_{i,j-1,k-1} + \delta(\_, w_j, u_k) & \\ s_{i-1,j,k} + \delta(v_i, \_, \_) & \\ s_{i,j-1,k} + \delta(\_, w_j, \_) & \text{Lattice edge: two indels} \\ s_{i,j,k-1} + \delta(\_, \_, u_k) & \end{cases}$$

- $\delta(x, y, z)$ is an entry in the 3-D scoring matrix

Scoring matrix has 5^3 entries

Multiple Alignment: Running Time

- For 3 sequences of length n, the run time is $7n^3$; $O(n^3)$
- For k sequences, build a k-dimensional Manhattan, with run time $(2^k-1)(n^k)$; $O(2^k n^k)$
- Conclusion: dynamic programming approach for alignment between two sequences is easily extended to k sequences but it is impractical due to exponential running time

Example:

To align 6, 100-base sequences, there are 63 directions to consider and 10^{12} cells to compute

Compare to aligning all 6(5-1)/2 = 15 pairs, each with 3 directions and 10,000 cells

Multiple Alignment Induces Pairwise Alignments

Every multiple alignment induces pairwise alignments

```
x:    AC-GCGG-C
y:    AC-GC-GAG
z:    GCCGC-GAG
```

Induces:

```
x: ACGCGG-C;    x: AC-GCGG-C;    y: AC-GCGAG
y: ACGC-GAC;    z: GCCGC-GAG;    z: GCCGCGAG
```

Inverse Problem

Do Pairwise Alignments imply a Multiple Alignment?

- Given 3 arbitrary pairwise alignments:

```
x: ACGCTGG-C;     x: AC-GCTGG-C;     y: AC-GC-GAG
y: ACGC--GAC;     z: GCCGCA-GAG;     z: GCCGCAGAG
```

- Can we construct a multiple alignment that induces them?

NOT ALWAYS

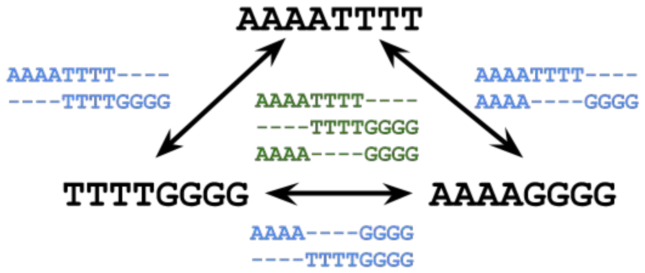
- Why? Because pairwise alignments can be arbitrarily inconsistent

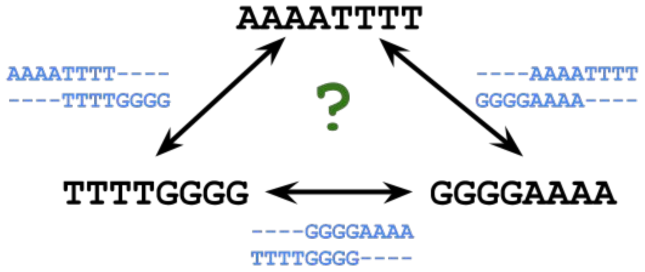
Combining Optimal Pairwise Alignments

- In some cases we can combine pairwise alignments into a single multiple alignment
- But, in others we cannot because one alignment makes a choice that is inconsistent with the overall best choice

```
AAAATTTT--------              ----AAAATTTT----
----TTTTGGGG----     -OR-     --------TTTTGGGG
--------GGGGAAAA              GGGGAAAA--------
```

- Is there another way?





Multiple Alignment from Pairwise Alignments

- From an optimal multiple alignment, we can infer pairwise alignments between all pairs of sequences, but they are not necessarily optimal
- It is difficult to infer a “good” multiple alignment from optimal pairwise alignments between all sequences
- Are we stuck, or is there some other trick?

Multiple Alignment using a Profile Scores

- We used profile scores earlier when we discussed Motif finding

```
          -  A  G  G  C  T  A  T  C  A  C  C  T  G
          T  A  G  -  C  T  A  C  C  A  -  -  -  G
          C  A  G  -  C  T  A  C  C  A  -  -  -  G
          C  A  G  -  C  T  A  T  C  A  C  -  G  G
          C  A  G  -  C  T  A  T  C  G  C  -  G  G

    A     0  5  0  0  0  0  5  0  0  4  0  0  0  0
    C     3  0  0  0  5  0  0  2  5  0  3  1  0  0
    G     0  0  5  1  0  0  0  0  0  1  0  0  2  5
    T     1  0  0  0  0  5  0  3  0  0  0  0  1  0
    -     1  0  0  4  0  0  0  0  0  0  2  4  2  0
```

- Thus far we have aligned sequences against other sequences
- Can we align a sequence against a profile?
- Can we align a profile against a profile?

Aligning Alignments

A more general version of the multi-alignment problem:

- Given two alignments, can we align them?

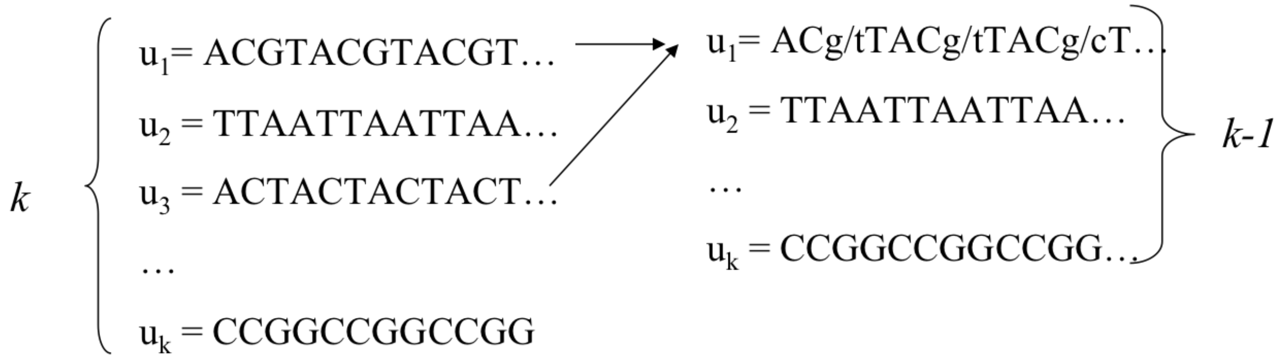
```
x: GGGCACTGCAT
y: GGTTACGTC--    Alignment 1
z: GGGAACTGCAG

w: GGACGTACC--    Alignment 2
v: GGACCT-----
```

- Idea: don’t use the sequences, but align their profiles

```
x: GGGCAC=TGCAT
y: GGTTAC=GTC--
z: GGGAAC=TGCAG     Combined Alignment
   ||  || | |
w: GG==ACGTACC--
v: GG==ACCT-----
```

Profile-Based Multiple Alignment: A Greedy Approach

- Choose the ***most similar pair*** of strings and combine them into a profile, thereby reducing alignment of *k* sequences to an alignment of of *k-1* sequences/profiles.
- **Repeat**
- This is a heuristic *greedy* method



Example

- Consider these 4 sequences

```
s1:    GATTCA
s2:    GTCTGA
s3:    GATATT
s4:    GTCAGC
```

- with the scoring matrix: {Match = 1, Mismatch = -1, Indel = -1}

Example (continued)

- There are 4 choose 2 = 6 possible pairwise alignments

s_2: `GTCTGA`
s_4: `GTCAGC (score = 2)`

s_1: `GAT-TCA`
s_2: `G-TCTGA (score = 1)`

s_1: `GAT-TCA`
s_3: `GATAT-T (score  = 1)`

s_1: `GATTCA--`
s_4: `G-T-CAGC (score = 0)`

s_2: `G-TCTGA`
s_3: `GATAT-T (score  = -1)`

s_3: `GAT-ATT`
s_4: `G-TCAGC (score = -1)`

- The best pairwise score, 2, is between s_2 and s_4

Example (continued)

- Combine s2 and s4:

```
 s2:  G T C T G A
      | | |   |           →        s2,4:  G T C t/a G a/c
s4:  G T C A G C
```

- Giving a set of three sequences:

```
s1  :    G  A  T  T  C  A
s3  :    G  A  T  A  T  T
s2,4:    G  T  C t/a G a/c
```

- **Repeat** for 3 choose 2 = 3 possible pairwise alignments

```
s1  :  GAT-TCA
s3  :  GATAT-T (score  = 1 + 1 + 1 - 1 + 1 - 1 - 1 = 1)

s1  :  GAT-TCA
s2,4:  G-TCtGa (score  = 2 - 2 + 2 - 2 + ½ - 1 + ½ = 0)

s3  :  GATAT-T
s2,4:  G-TCtGa (score  = 2 - 2 + 2 - 2 + ½ - 1 - 1 = -1½)
```

Next Time

- Other Dynamic Programming problems
- Divide-and-Conquer
- Other approaches to sequence alignment

Prologue to Experiment 3
OH, WHAT TO WEAR? WHAT TO WEAR?
SUCH A VAST WARDROBE! HOW WILL I EVER FIND THE RIGHT COMBINATION?
PERHAPS I CAN HELP!
WHO ARE YOU?
MACE
I'M HERE TO TELL YOU ABOUT "DynaMix"– THE NEW "SELECTION OPTIMIZATION SOFTWARE FROM ALGORITHMIC ANOMALY!
HOW DOES IT WORK?
WE USE DYNAMIC PROGRAMMING TO RECURSIVELY CREATE AN OUTFIT THAT YOU'LL LOOK JUST STUNNING IN!
NOW, IF WE WANT TO DO THIS IN O(n²) TIME, WE'LL HAVE TO HURRY!
LET'S BEGIN! THAT SHIRT AND THOSE PANTS YOU'RE HOLDING – HOW GOOD DO THEY LOOK TOGETHER, ON A SCALE FROM 1 TO 10?
UM... SEVEN?
EXCELLENT! I'LL ENTER THAT INTO OUR ARRAY!
THOSE SHOES WITH THAT NECKLACE?
THREE.
THOSE OTHER SHOES WITH THAT SHIRT?
SIX.
THIS NECKLACE WITH THESE PANTS?
EIGHT.
THAT SHIRT WITH THOSE PANTS?
TWO.
THREE HOURS LATER...
THERE! I'VE TOLD YOU EVERYTHING!
AND NOW, IN A FEW SECONDS, THE PARADIGM WILL HAVE WORKED ITS MAGIC!
VOILA! ALL DONE! SOOO...YOU'LL WANT TO WEAR THESE PANTS, THAT SHIRT, THOSE SHOES, AND THIS NECKLACE!
LATER THAT NIGHT...
DARLING, YOU LOOK ABSOLUTELY DYNAMIC!
SOMEHOW, I KNEW HE WAS GOING TO SAY THAT!
BUT HOW DOES IT WORK? TURN THE PAGE TO FIND OUT!