Dynamic Programming: Edit Distance & Sequence Alignment

DNA Sequence Comparison: First Success Story

- Finding sequence similarities with genes of known function is a common approach to infer a newly sequenced gene’s function

- In 1984 Russell Doolittle and colleagues found similarities between cancer-causing gene and normal growth factor (PDGF) gene

Bring in the Bioinformaticians

- Gene similarities between two genes with known and unknown function alert biologists to some possibilities
- Computing a similarity score between two genes tells how likely it is that they have similar functions
- Dynamic programming is a technique for revealing similarities between genes

Alignment: 2 row representation

Given 2 DNA sequences **v** and **w**:

v	:	ATGTTAT	n = 7
w	:	ATCGTAC	m = 7

Alignment : 2 * ***k*** matrix (***k*** >= ***m***, ***n***)

letters of ***v***	A	T	--	G	T	T	A	T	--
letters of ***w***	A	T	C	G	T	--	A	--	C

Alignment: 2 row representation

Columns that contain the same letter in both rows are called matches.

Columns containing different letters are called mismatches.

The columns of the alignment containing one space are called indels.

The columns containing a space in the top row called insertions.

The columns with a space in the bottom row deletions.

letters of ***v***	**A**	**T**	**--**	**G**	**T**	**T**	**A**	**T**	**--**
letters of ***w***	**A**	**T**	**C**	**G**	**T**	**--**	**A**	**--**	**C**

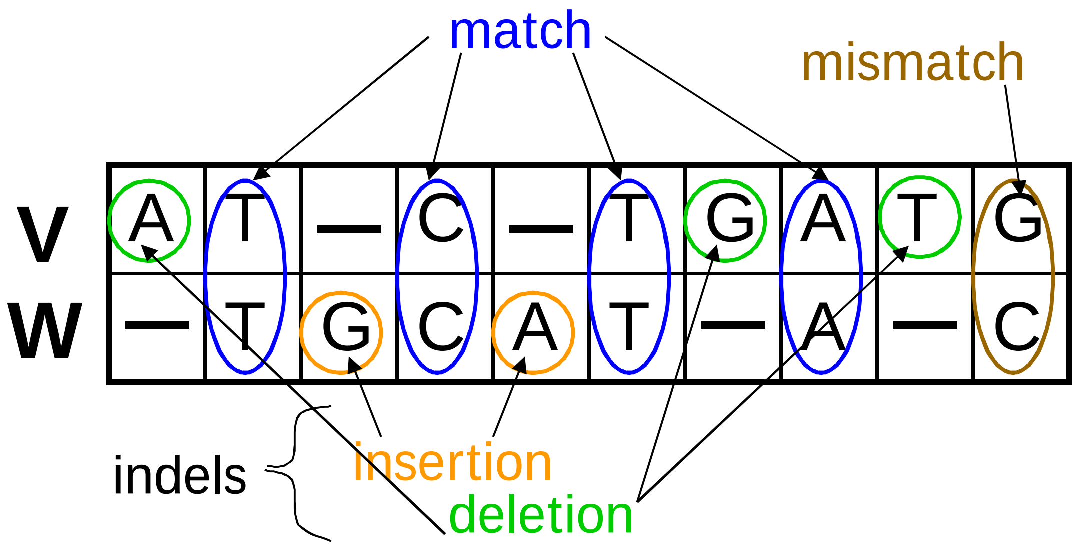
5 matches 2 insertions 2 deletions

Aligning DNA Sequences

V = ATCTGATG n = 8

W = TGCATAC m = 7

4	matches
1	mismatches
2	insertions
3	deletions



Longest Common Subsequence (LCS) – Alignment without Mismatches

- Given two sequences

  $\boldsymbol{v} = v_1\, v_2 \ldots v_m$ and $\boldsymbol{w} = w_1\, w_2 \ldots w_n$

- The LCS of ***v*** and ***w*** is a sequence of positions in

  v: $1 \leq i_1 < i_2 < \ldots < i_k \leq m$

and a sequence of positions in

  w: $1 \leq j_1 < j_2 < \ldots < j_k \leq n$

such that i_t*-th* letter *of **v** equals to* j_t*-letter of **w*** and ***t*** is maximal

LCS: Example

i coords:	0	1	2	2	3	3	4	5	6	7	8
elements of *v*		A	T	--	C	--	T	G	A	T	C
elements of *w*		--	T	G	C	A	T	--	A	--	C
j coords:	0	0	1	2	3	4	5	5	6	6	7

$(0,0)\rightarrow(1,0)\rightarrow(2,1)\rightarrow(2,2)\rightarrow(3,3)\rightarrow(3,4)\rightarrow(4,5)\rightarrow(5,5)\rightarrow(6,6)\rightarrow(7,6)\rightarrow(8,7)$

Matches shown in yellow

positions in *v*: $2 < 3 < 4 < 6 < 8$

positions in *w*: $1 < 3 < 5 < 6 < 7$

Every common subsequence is a path in 2-D grid

LCS: Dynamic Programming

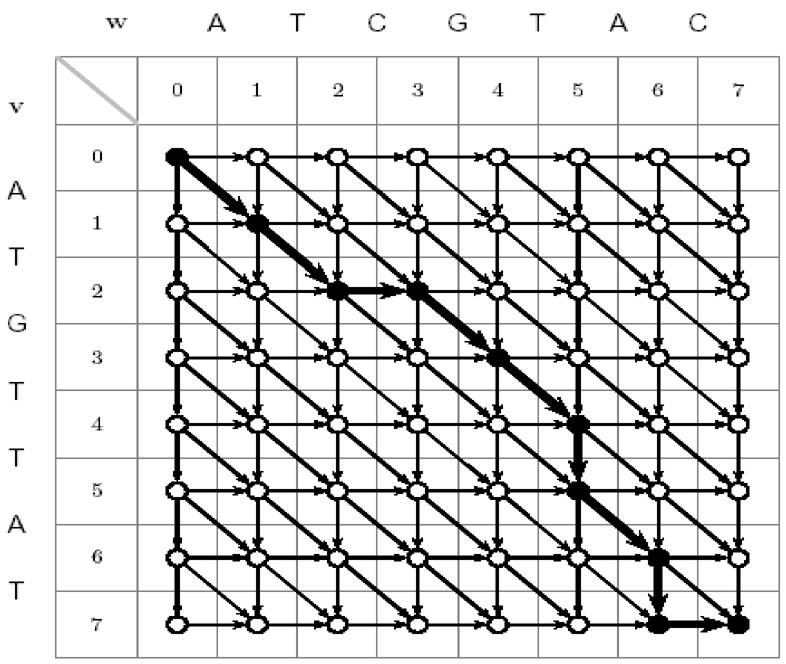
- Find the LCS of two strings

  Input: A weighted graph *G* with two distinct vertices, one labeled "*source*" one labeled "*sink*"

  Output: A longest path in *G* from "*source*" to "*sink*"

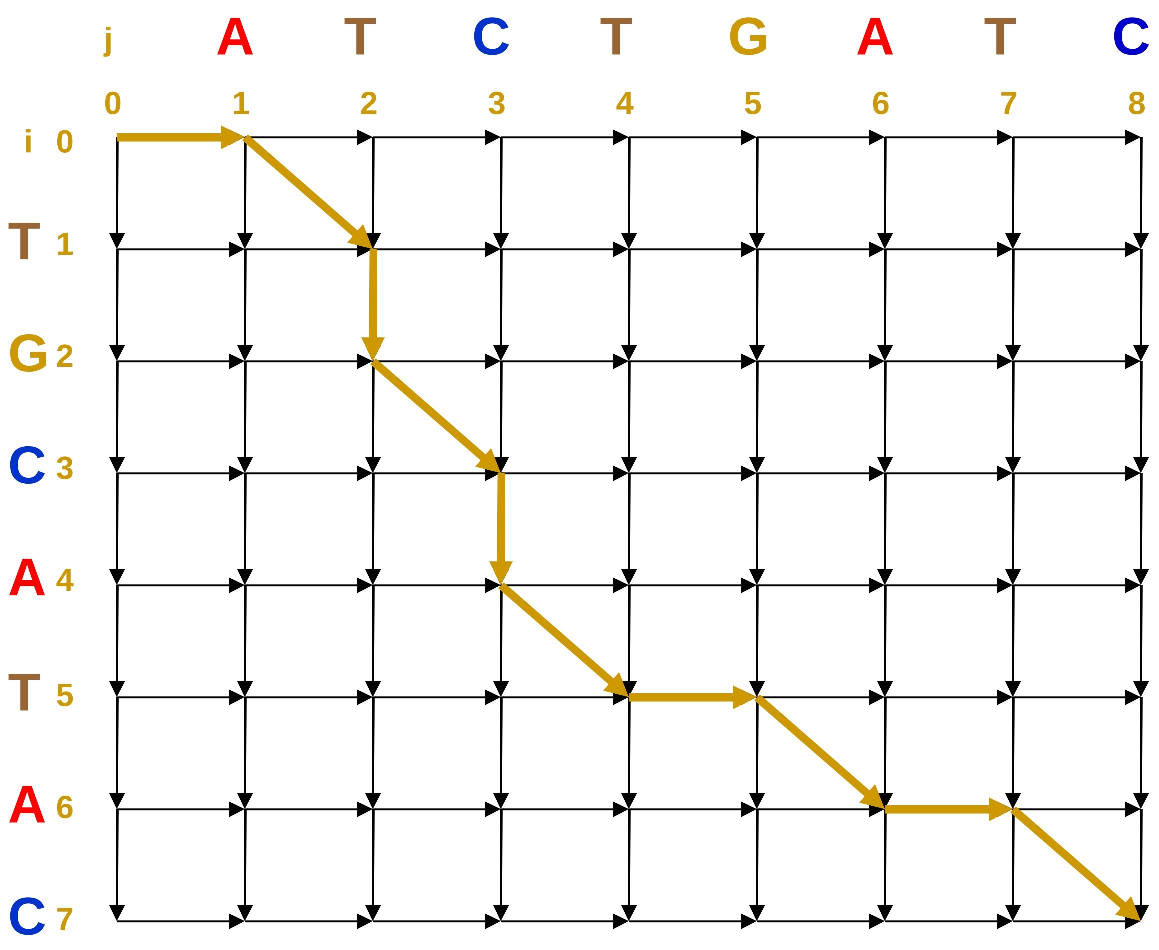
- Solve using an LCS edit graph with diagonals replaced with +1 edges

```
         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
```

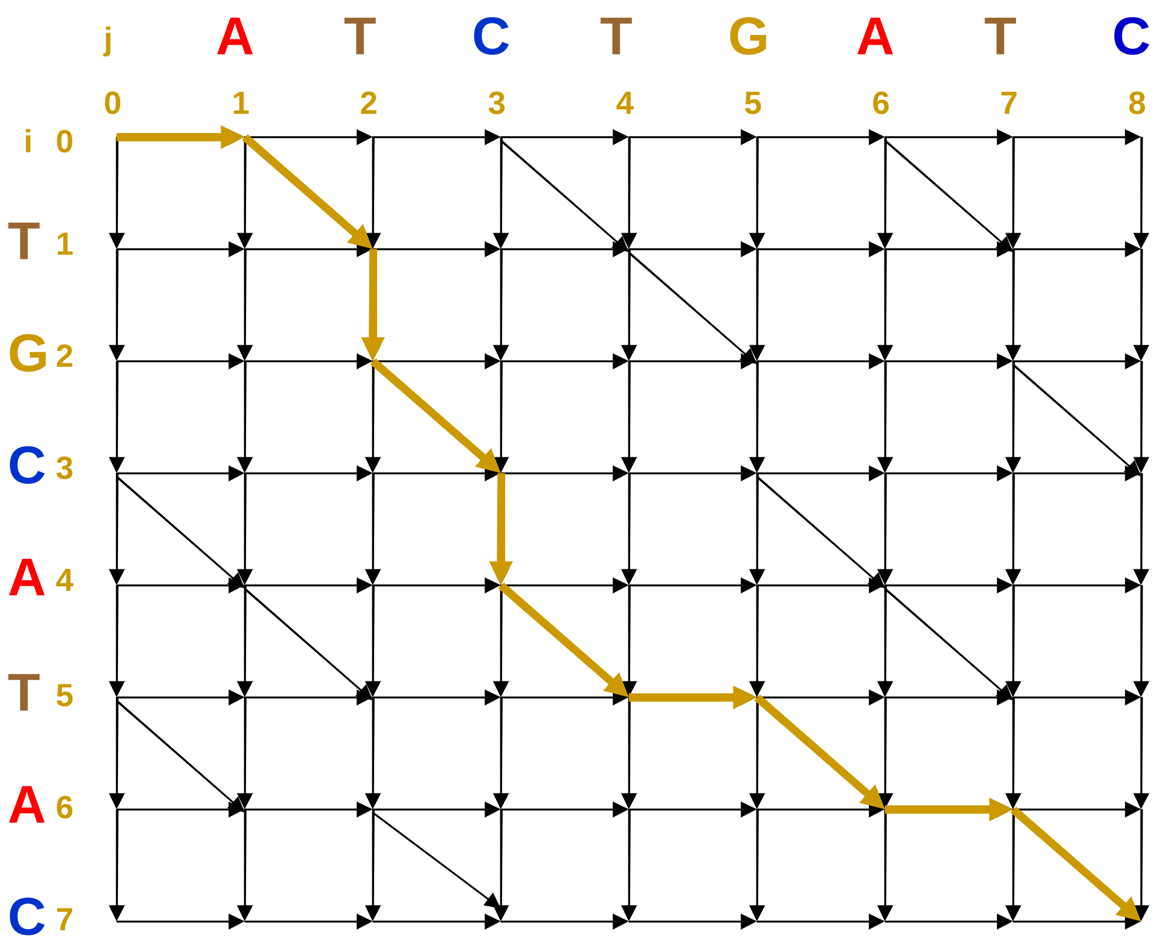


```
↘  ↘  →  ↘  ↘  ↓  ↘  ↓  →
A  T  -  G  T  T  A  T  -
A  T  C  G  T  -  A  -  C
```

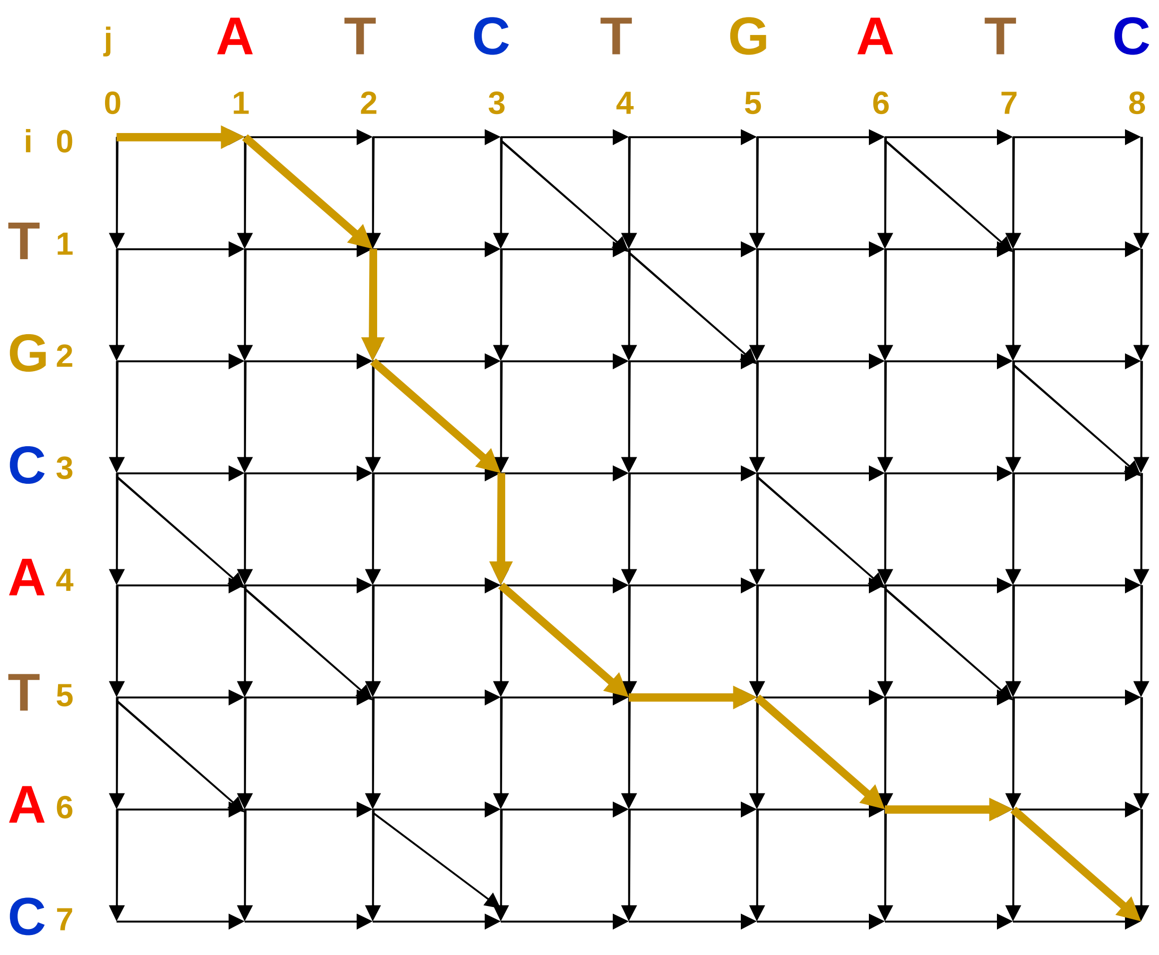
LCS Problem as Manhattan Tourist Problem



Edit Graph for LCS Problem



Edit Graph for LCS Problem



Every path is a common subsequence.

Every diagonal edge adds an extra element to common subsequence

LCS Problem: *Find a path with maximum number of diagonal edges*

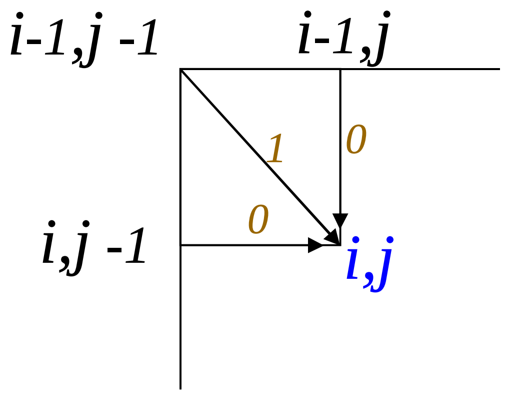
Computing LCS

Let $\boldsymbol{v}_i$ = prefix of $\boldsymbol{v}$ of length i: $v_1 \ldots v_i$

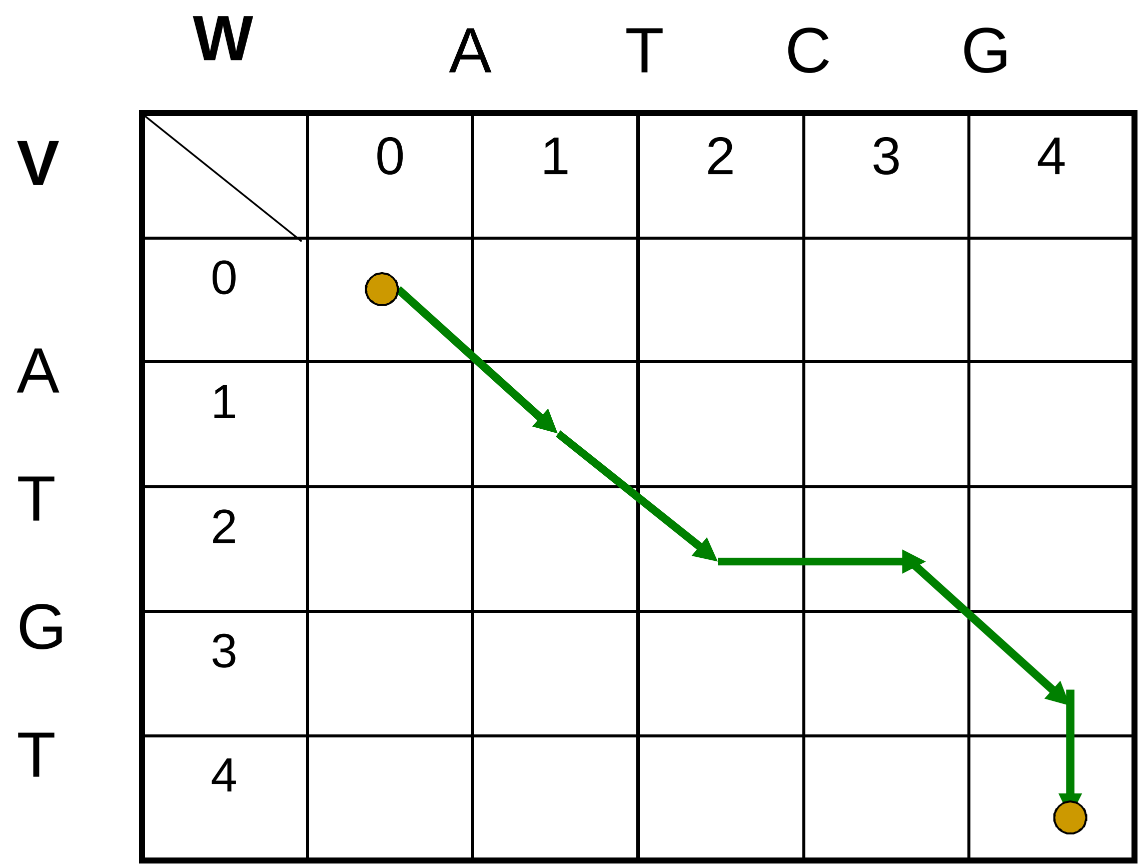
and $\boldsymbol{w}_j$ = prefix of $\boldsymbol{w}$ of length j: $w_1 \ldots w_j$

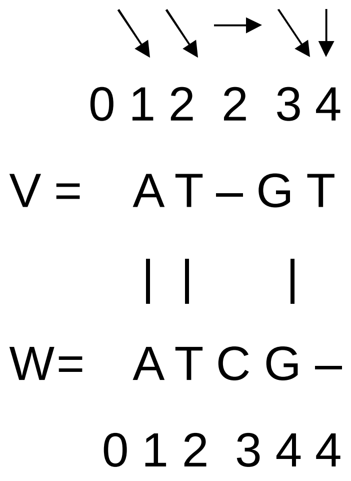
The length of LCS($\boldsymbol{v}_i$,$\boldsymbol{w}_j$) is computed by:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1 \text{ if } v_i = w_j \end{cases}$$



Every Path in the Grid Corresponds to an Alignment





Aligning Sequences without Insertions and Deletions: Hamming Distance

Given two DNA sequences ***v*** and ***w*** :

```
v :  ATATATAT
w :  TATATATA
```

- The Hamming distance: $d_H(\mathbf{v}, \mathbf{w}) = 8$ is large but the sequences are very similar

Aligning Sequences with Insertions and Deletions

By shifting one sequence over one position:

```
v :  A T A T A T A T - -
w :  - - T A T A T A T A
```

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

Edit Distance

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

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

Hamming distance:

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

Computing Hamming distance
is a trivial task

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

Make it all line up

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

```
v  = - ATATATAT
       | | | | | | |
w  = TATATATA
```

Hamming distance: $d(\mathbf{v}, \mathbf{w})=8$

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 vs Hamming Distance

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

```
v  = ATATATAT
      | | |  | |  | | |
w = TATATATA
```

Hamming distance:

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

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

```
v  = - ATATATAT
       | | |  | |  | |
w = TATATATA
```

Edit distance:

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

(one insertion and one deletion)

How to find what ***j*** goes with what ***i*** ???

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)

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)

ATGCATA → (substitute G for 5th A)

ATGCGTA → (substitute C for 3rd G)

ATCCGAT (Done)

Edit Distance: Example (cont'd)

TGCATAT → ATCCGAT in 4 steps

TGCATAT → (insert A at front)

ATGCATAT → (delete 6th T)

ATGCATA → (substitute G for 5th A)

ATGCGTA → (substitute C for 3rd G)

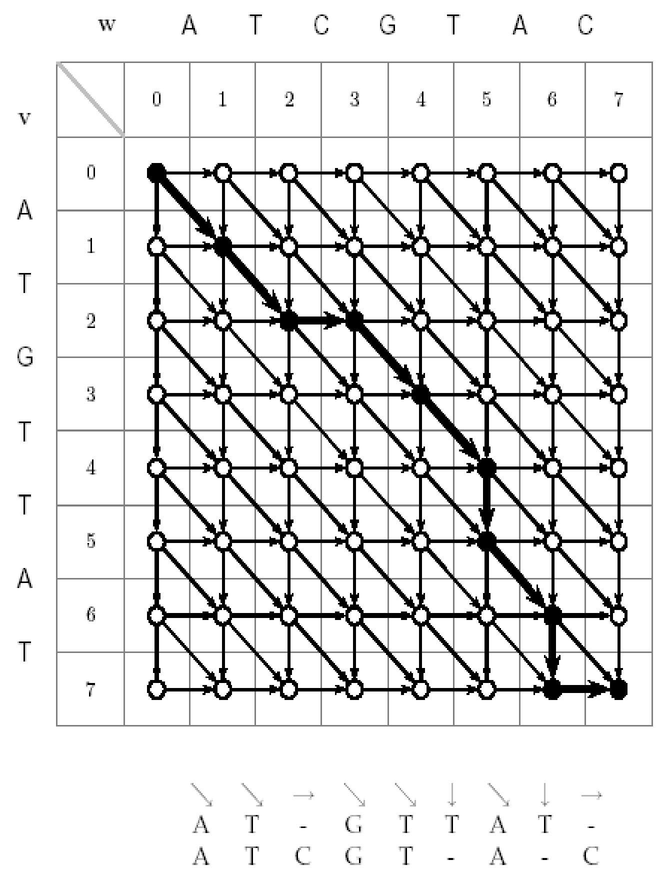
ATCCGAT (Done)

Can it be done in 3 steps???

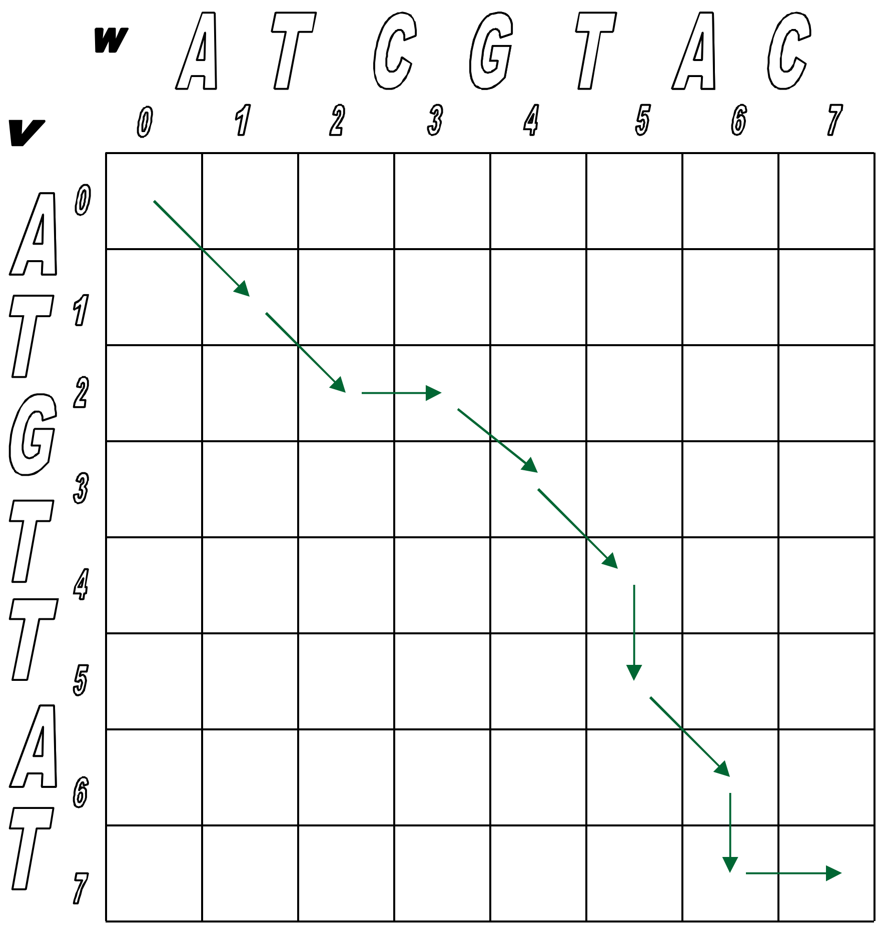
The Alignment Grid

```
       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
```

- Every alignment path is from source to sink



Alignment as a Path in the Edit Graph

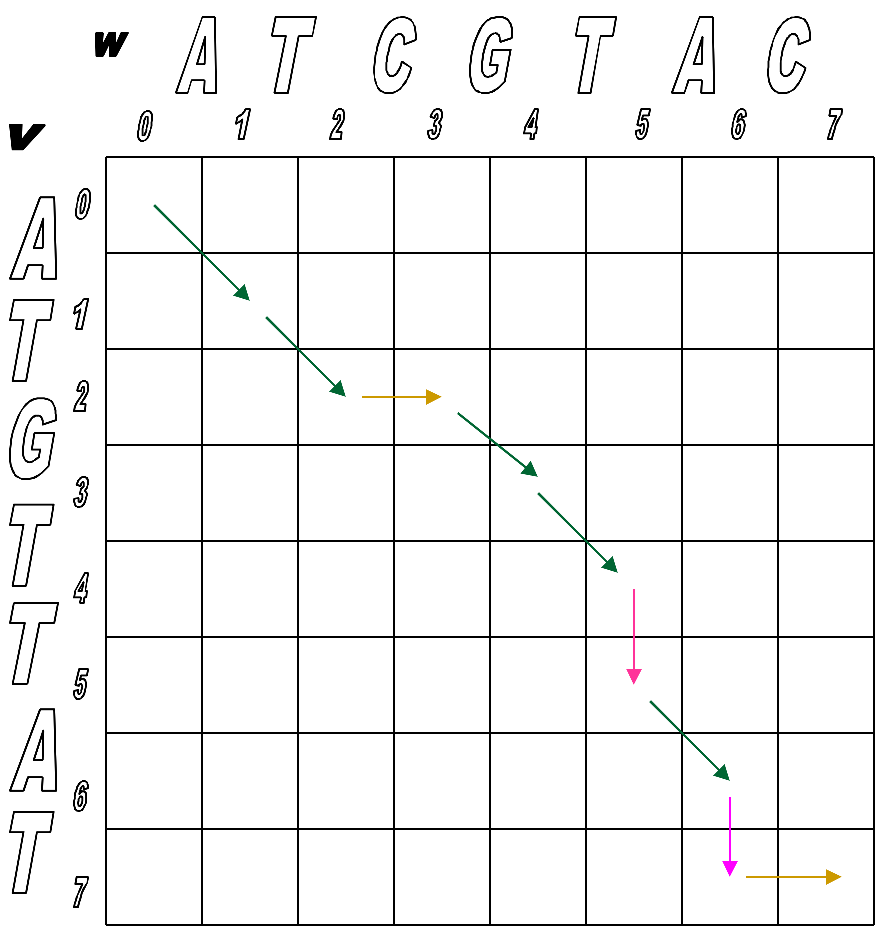


```
0 1 2 2 3 4 5 6 7 7
  A T _ G T T A T _
  A T C G T _ A _ C
0 1 2 3 4 5 5 6 6 7
```

- Corresponding path -

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

Alignments in Edit Graph (cont'd)

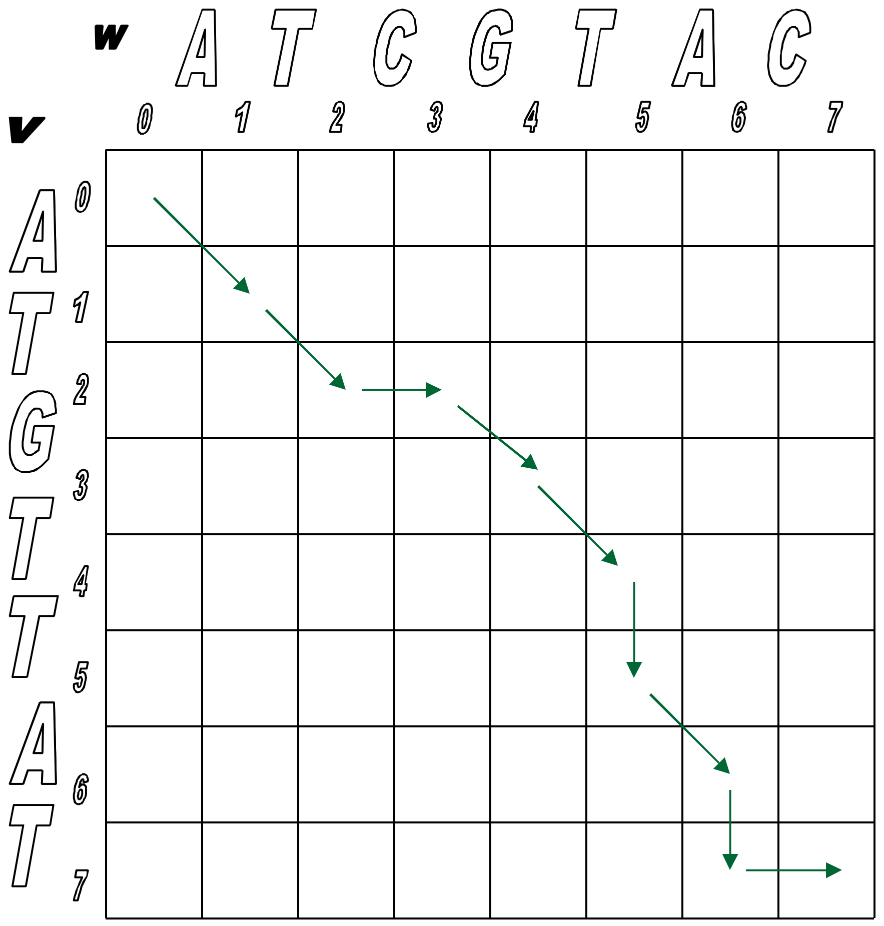


$\downarrow$ and $\rightarrow$ represent indels in **v** and **w** with score 0.

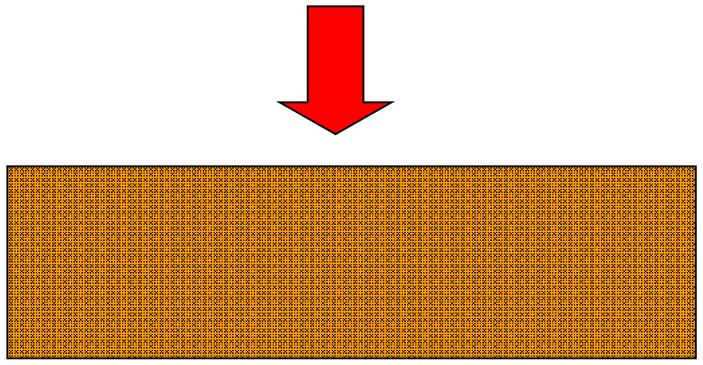
$\searrow$ represent matches with score 1.

- The score of the alignment path is 5.

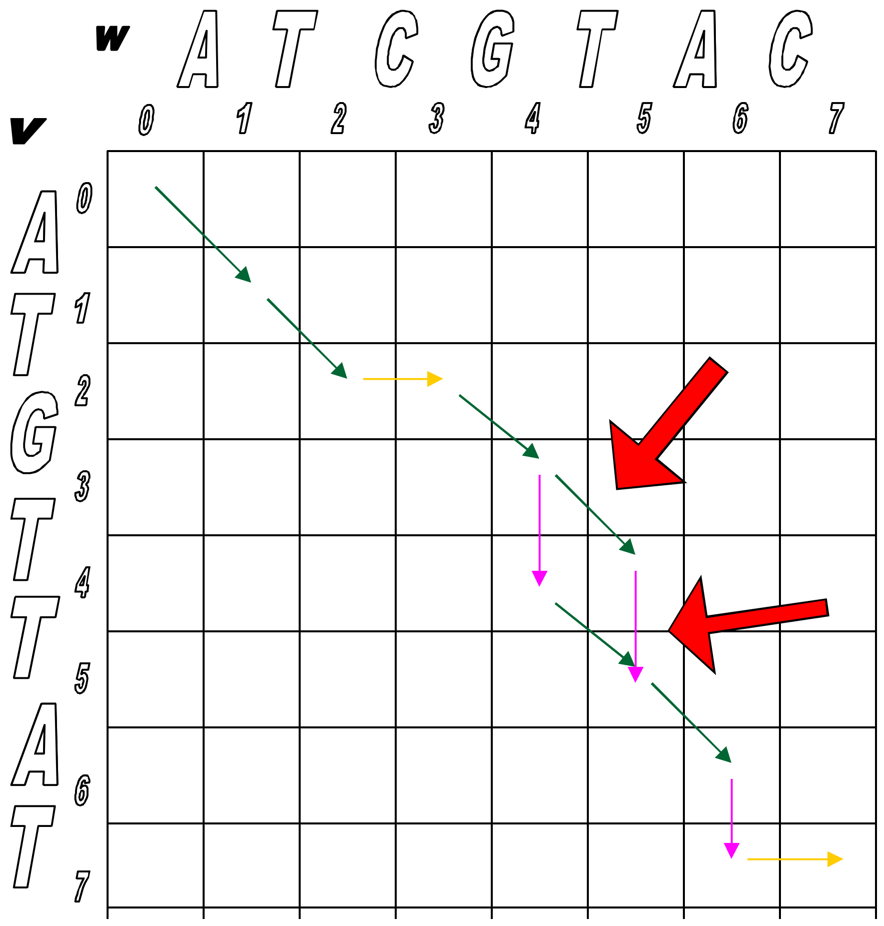
Alignment as a Path in the Edit Graph



Every path in the edit graph corresponds to an alignment:



Alignment as a Path in the Edit Graph



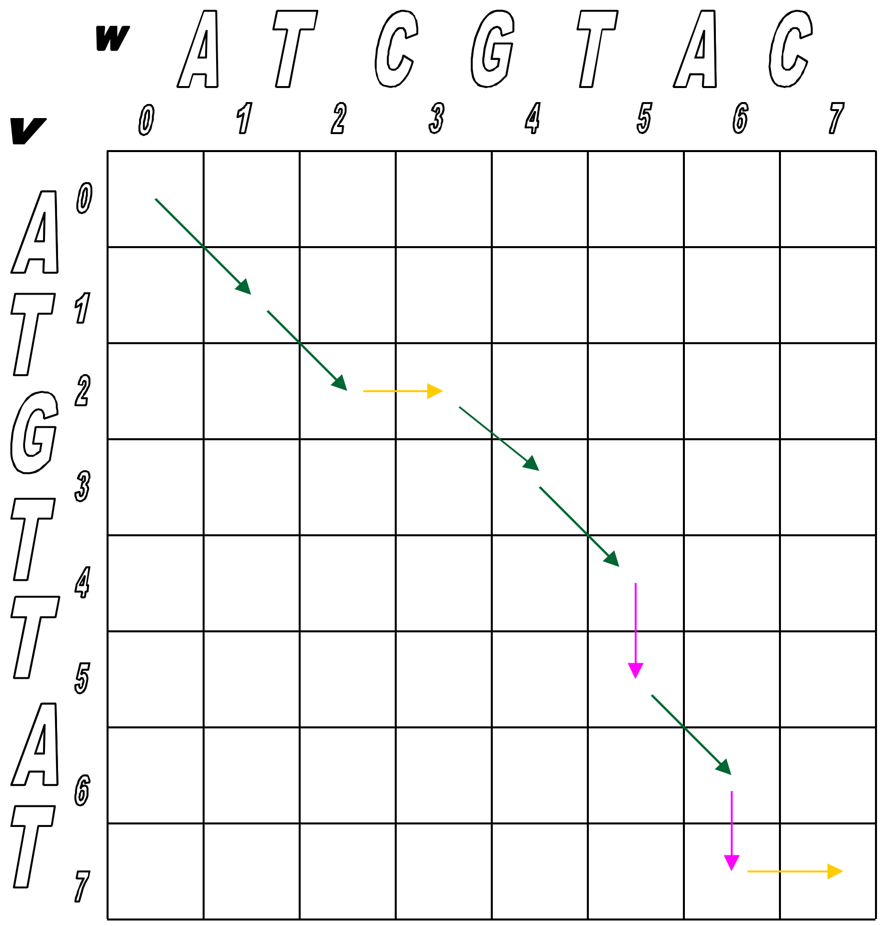
<u>Old Alignment</u>

```
     0122345677
v=   AT_GTTAT_
w=   ATCGT_A_C
     0123455667
```

<u>New Alignment</u>

```
     0122345677
v=   AT_GTTAT_
w=   ATCG_TA_C
     0123445667
```

Alignment as a Path in the Edit Graph



```
     012234567 7
v=    AT_GTTAT_
w=    ATCGT_A_C
     0123455667
```

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

Alignment: Dynamic Programming

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

Dynamic Programming Example

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

v	w		A	T	C	G	T	A	C
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	↖ 1	← 1	← 1	← 1	← 1	↖ 1	← 1
T	2	0	↑ 1						
G	3	0	↑ 1						
T	4	0	↑ 1						
T	5	0	↑ 1						
A	6	0	↖ 1						
T	7	0	↑ 1						

$$S_{i,j} = \max \begin{cases} S_{i-1,\,j-1} & \leftarrow \text{value from NW +1, if } v_i = w_j \quad \nwarrow \\ S_{i-1,\,j} & \leftarrow \text{value from North (top)} \quad \uparrow \\ S_{i,\,j-1} & \leftarrow \text{value from West (left)} \quad \leftarrow \end{cases}$$

Alignment: Backtracking

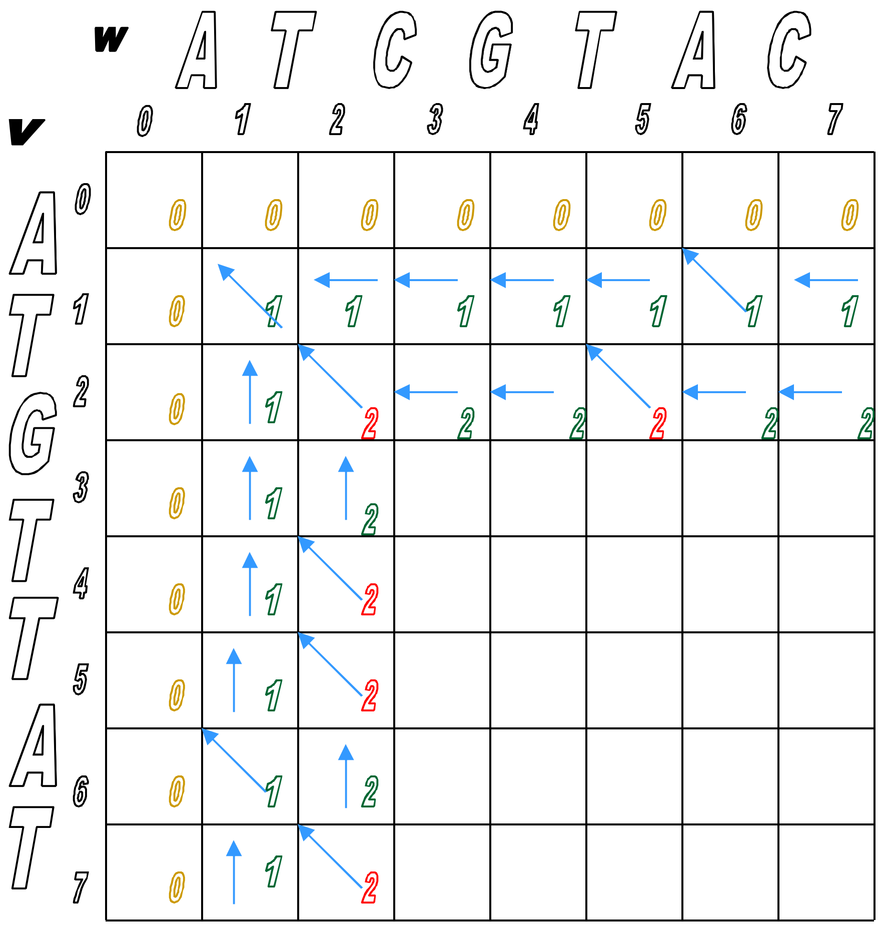
Arrows ↖ ↑ ← show where the score originated from.

↑ if from the top

← if from the left

↖ if $v_i = w_j$

Backtracking Example



Find a match in row and column 2.

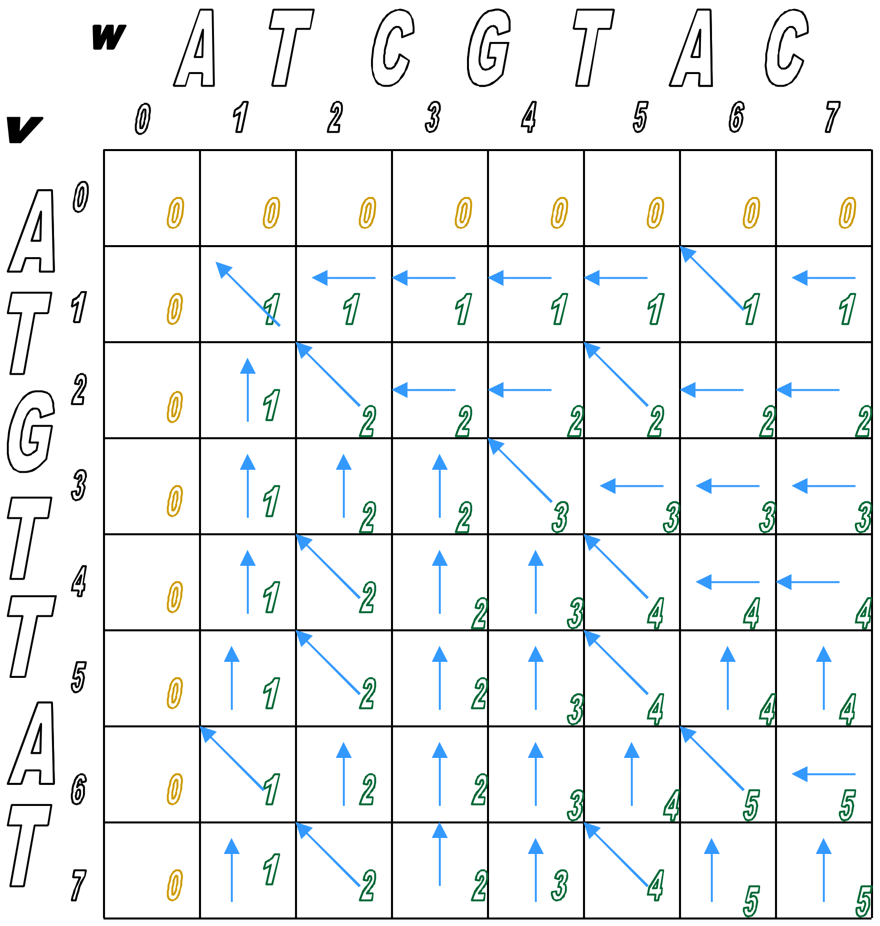
i=2, *j*=2,5 is a match (T).

j=2, *i*=4,5,7 is a match (T).

Since $v_i = w_j,\ s_{i,j} = s_{i-1,j-1} + 1$

$$s_{2,2} = [s_{1,1} = 1] + 1$$
$$s_{2,5} = [s_{1,4} = 1] + 1$$
$$s_{4,2} = [s_{3,1} = 1] + 1$$
$$s_{5,2} = [s_{4,1} = 1] + 1$$
$$s_{7,2} = [s_{6,1} = 1] + 1$$

Backtracking Example



v \ w			A	T	C	G	T	A	C
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	↖ 1	← 1	← 1	← 1	← 1	↖ 1	← 1
T	2	0	↑ 1	↖ 2	← 2	← 2	↖ 2	← 2	← 2
G	3	0	↑ 1	↑ 2	↑ 2	↖ 3	← 3	← 3	← 3
T	4	0	↑ 1	↖ 2	↑ 2	↑ 3	↖ 4	← 4	← 4
T	5	0	↑ 1	↖ 2	↑ 2	↑ 3	↖ 4	↑ 4	↑ 4
A	6	0	↖ 1	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5
T	7	0	↑ 1	↖ 2	↑ 2	↑ 3	↖ 4	↑ 5	↑ 5

Continuing with the dynamic programming algorithm gives this result.

Alignment: Dynamic Programming

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

Alignment: Dynamic Programming

$$s_{i,j} = \max \begin{cases} s_{i-1,\, j-1} + 1 \text{ if } v_i = w_j & \searrow \\ s_{i-1,\, j} + 0 & \downarrow \\ s_{i,\, j-1} + 0 & \rightarrow \end{cases}$$

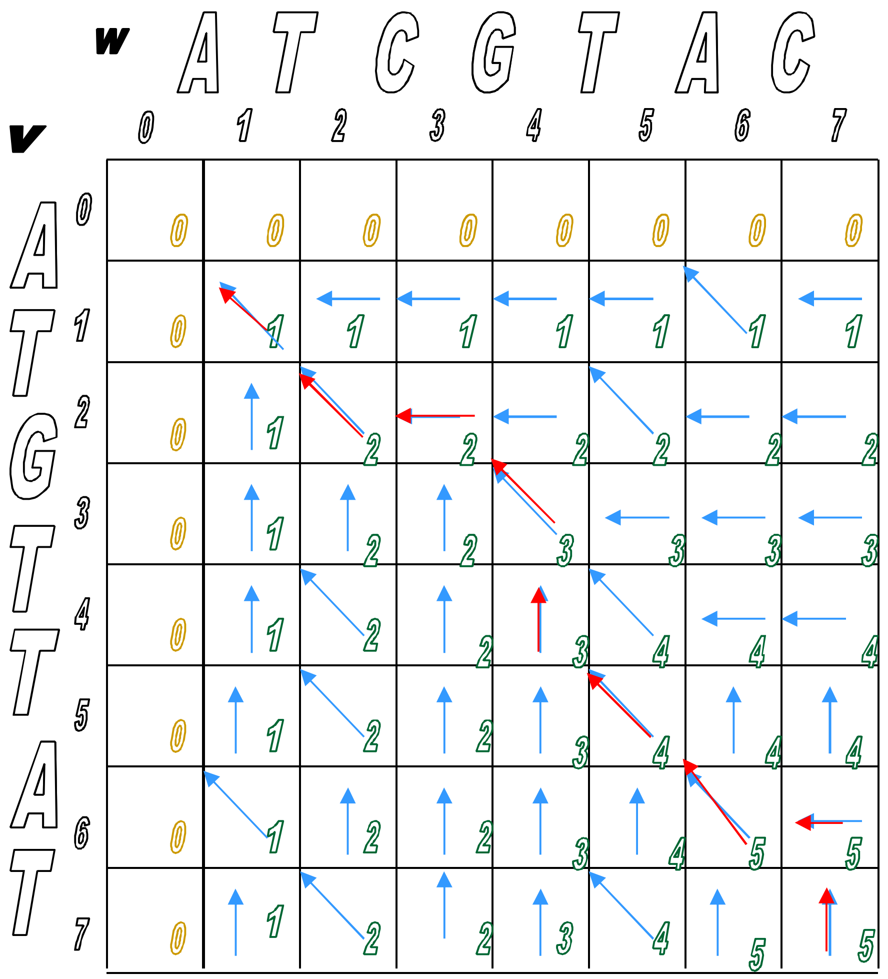
This recurrence corresponds to the Manhattan Tourist problem (three incoming edges into a vertex) with all horizontal and vertical edges weighted by zero.

LCS Algorithm

```
1.  LCS(v,w)
2.     for $i \leftarrow 1$ to $n$
3.        $s_{i,0} \leftarrow 0$
4.     for $j \leftarrow 1$ to $m$
5.        $s_{0,j} \leftarrow 0$
6.     for $i \leftarrow 1$ to $n$
7.        for $j \leftarrow 1$ to $m$
8.
9.           $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$
10.
11.
•            $b_{i,j} \leftarrow \begin{cases} \text{“} \uparrow \text{”} & \textbf{if } s_{i,j} = s_{i-1,j} \\ \text{“} \leftarrow \text{”} & \textbf{if } s_{i,j} = s_{i,j-1} \\ \text{“} \nwarrow \text{”} & \textbf{if } s_{i,j} = s_{i-1,j-1} + 1 \end{cases}$
•
•      return ($s_{n,m}$, b)
```

Now What?

- LCS(v,w) created the alignment grid
- Now we need a way to read the best alignment of v and w
- Follow the arrows backwards from sink



Printing LCS: Backtracking

```
1.  PrintLCS(b,v,$i$,$j$)
2.      if $i = 0$ or $j = 0$
3.          return
4.      if $b_{i,j}$ = “ $\nwarrow$ “
5.          PrintLCS(b,v,$i-1$,$j-1$)
6.          print $v_i$
7.      else
8.          if $b_{i,j}$ = “ $\uparrow$ “
9.              PrintLCS(b,v,$i-1$,$j$)
10.         else
11.             PrintLCS(b,v,$i$,$j-1$)
```

LCS Runtime

- It takes O(*nm*) time to fill in the *nxm* dynamic programming matrix.
- Why O(*nm*)? The pseudocode consists of a nested “for” loop inside of another “for” loop to set up a *nxm* matrix.

Sequence Alignment

From LCS to Alignment: Change up the Scoring

- The 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**

  -*μ* : **mismatch penalty**

  -*σ* : **indel penalty**

Simple Scoring

- When mismatches are penalized by $-\mu$, indels are penalized by $-\sigma$,

  and matches are rewarded with $+1$,

  the resulting score is:

$$\#matches - \mu(\#mismatches) - \sigma\,(\#indels)$$

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

$\uparrow\rightarrow = -\sigma$

$\searrow \begin{cases} = 1 \text{ if match} \\ = -\mu \text{ if mismatch} \end{cases}$

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

μ : mismatch penalty

σ : indel penalty

Scoring Matrices

To generalize scoring, consider a (4+1) x(4+1) **scoring matrix** δ.

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, $\delta(v_i , w_j)$, will be 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 and K are different amino acids, they have a positive score.*
- *Why? They are both positively charged amino acids → will not greatly change function of protein.*

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.

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)

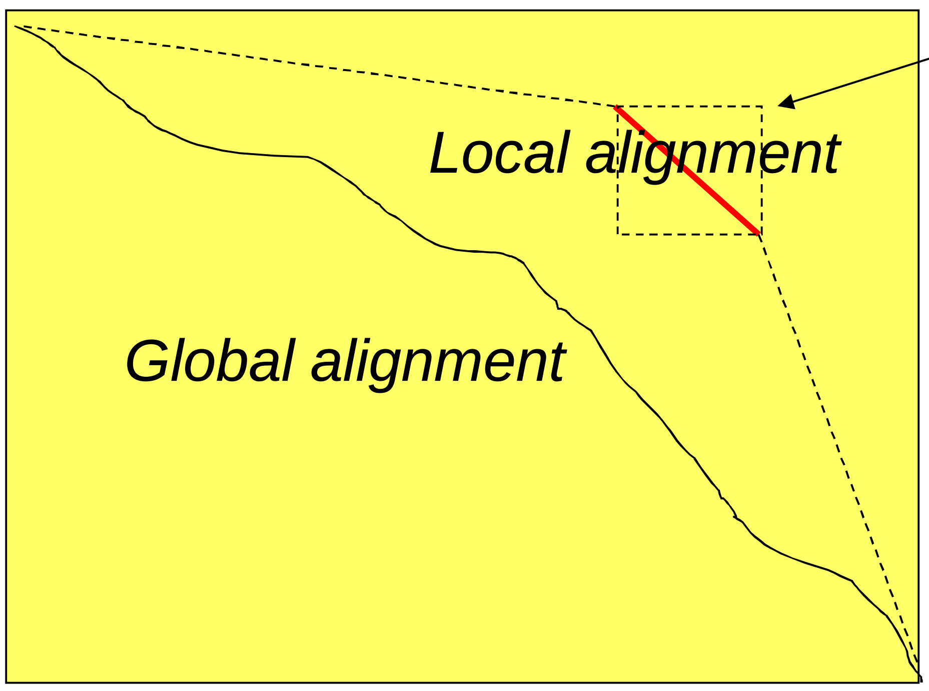
- *Global Alignment*

```
--T—-CC-C-AGT—-TATGT-CAGGGGACACG—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 alignments of all possible substrings

The Problem with this Problem

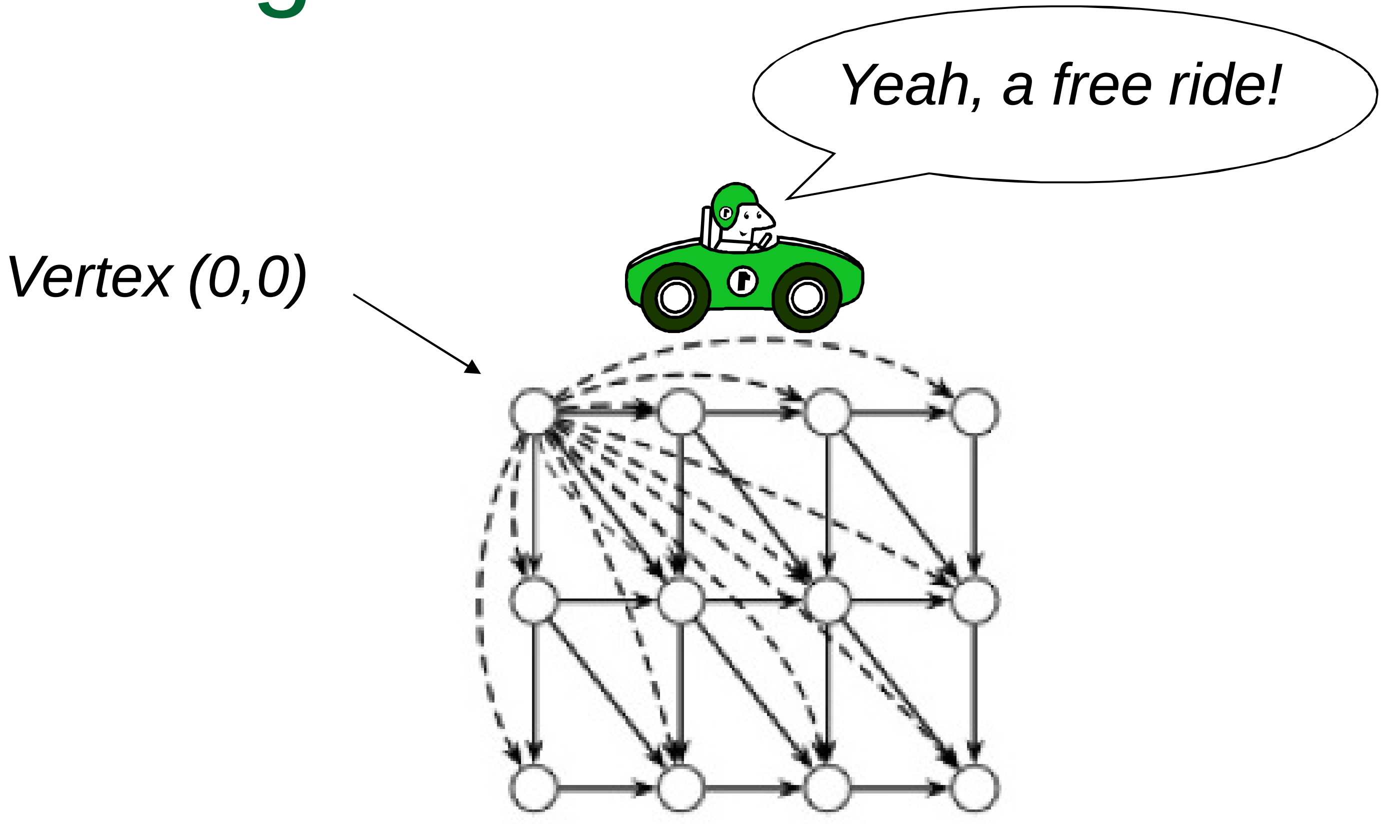
- High running time $O(n^4)$:

  - In the grid of size *n x n* there are ~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.

- This can be remedied by giving free rides

Local Alignment: Free Rides



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:*

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

***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*

Scoring Indels: Naive Approach

- A fixed penalty σ is given to every indel:
  - $-\sigma$ 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

Affine Gap Penalties

- In nature, a series of *k* indels often come as a single event rather than a series of *k* single nucleotide events:

```
ATA__GC        ATAG_GC
ATATTGC        AT_GTGC
```

This is more likely.

Normal scoring would give the same score for both alignments

This is less likely.

Accounting for Gaps

- *Gaps*- contiguous sequence of spaces in one of the rows

- Score for a gap of length *x* is:

  $-(\rho + \sigma x)$

  where ρ >0 is the penalty for introducing a gap:

  gap opening penalty

  ρ will be large relative to σ*:*

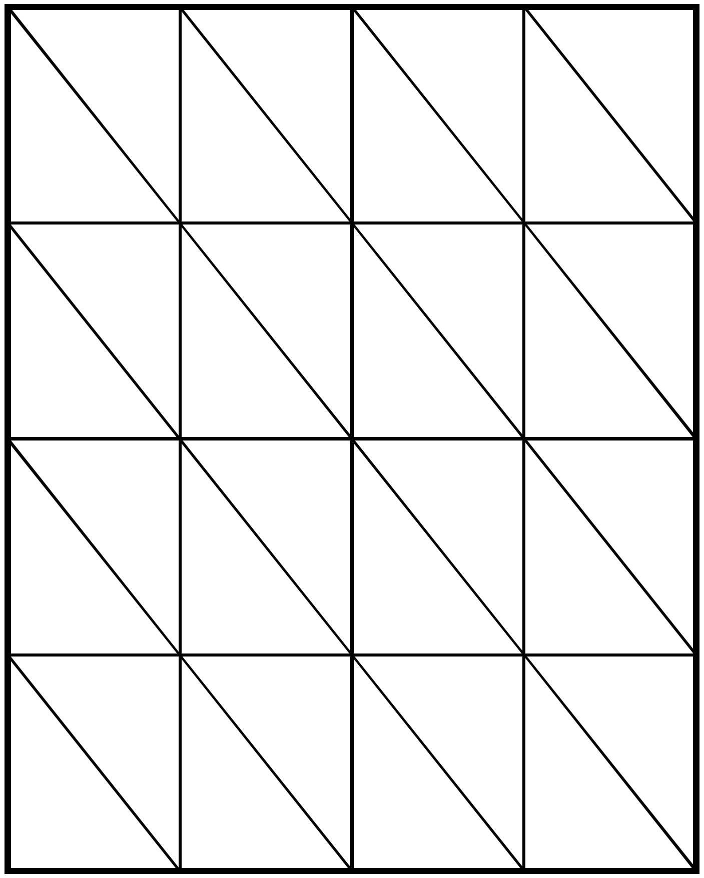
  gap extension penalty

  because you do not want to add too much of a penalty for extending the gap.

Affine Gap Penalties

- Gap penalties:
  - $-\rho-\sigma$ when there is 1 indel
  - $-\rho-2\sigma$ when there are 2 indels
  - $-\rho-3\sigma$ when there are 3 indels, etc.
  - $-\rho- x\cdot\sigma$ (-gap opening - *x* gap extensions)
- Somehow reduced penalties (as compared to naïve scoring) are given to runs of horizontal and vertical edges

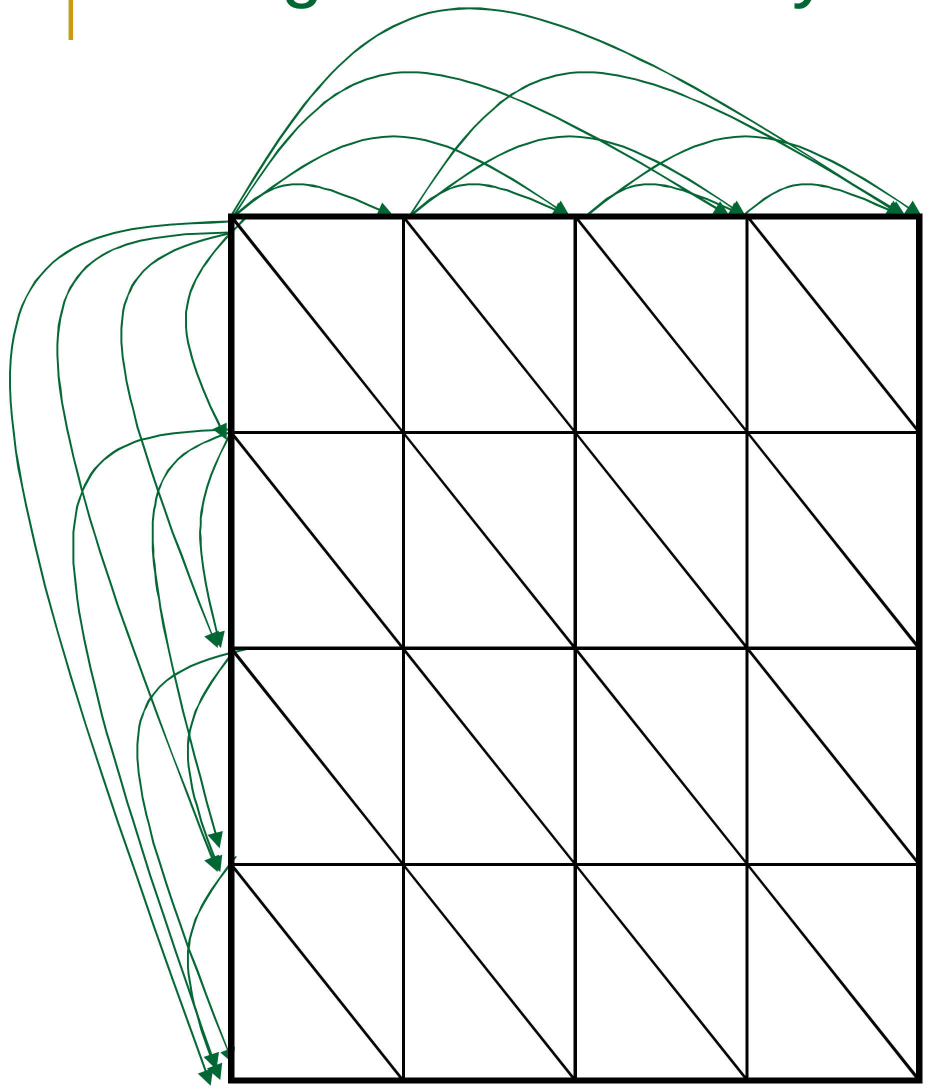
Affine Gap Penalties and Edit Graph



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

$$-\rho - x * \sigma$$

Adding "Affine Penalty" Edges to the Edit Graph

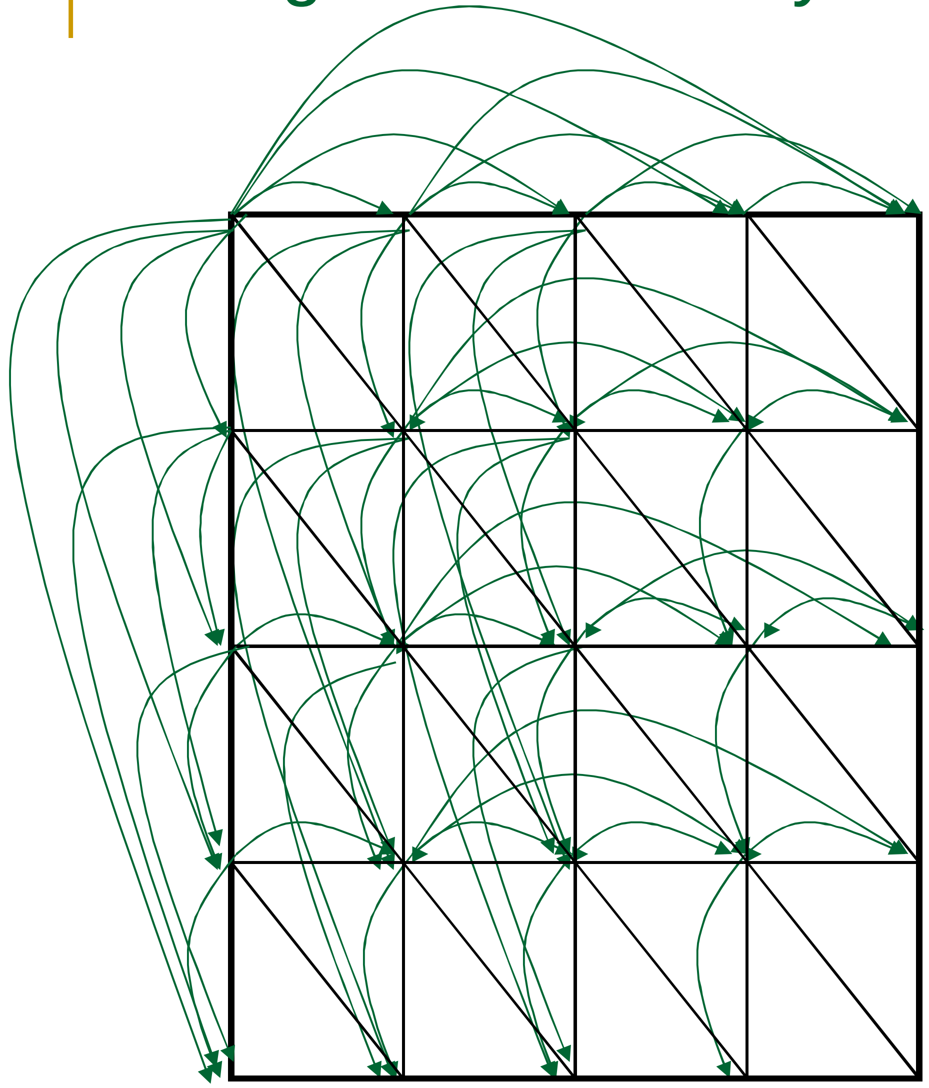


There are many such edges!

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)

Adding "Affine Penalty" Edges to the Edit Graph

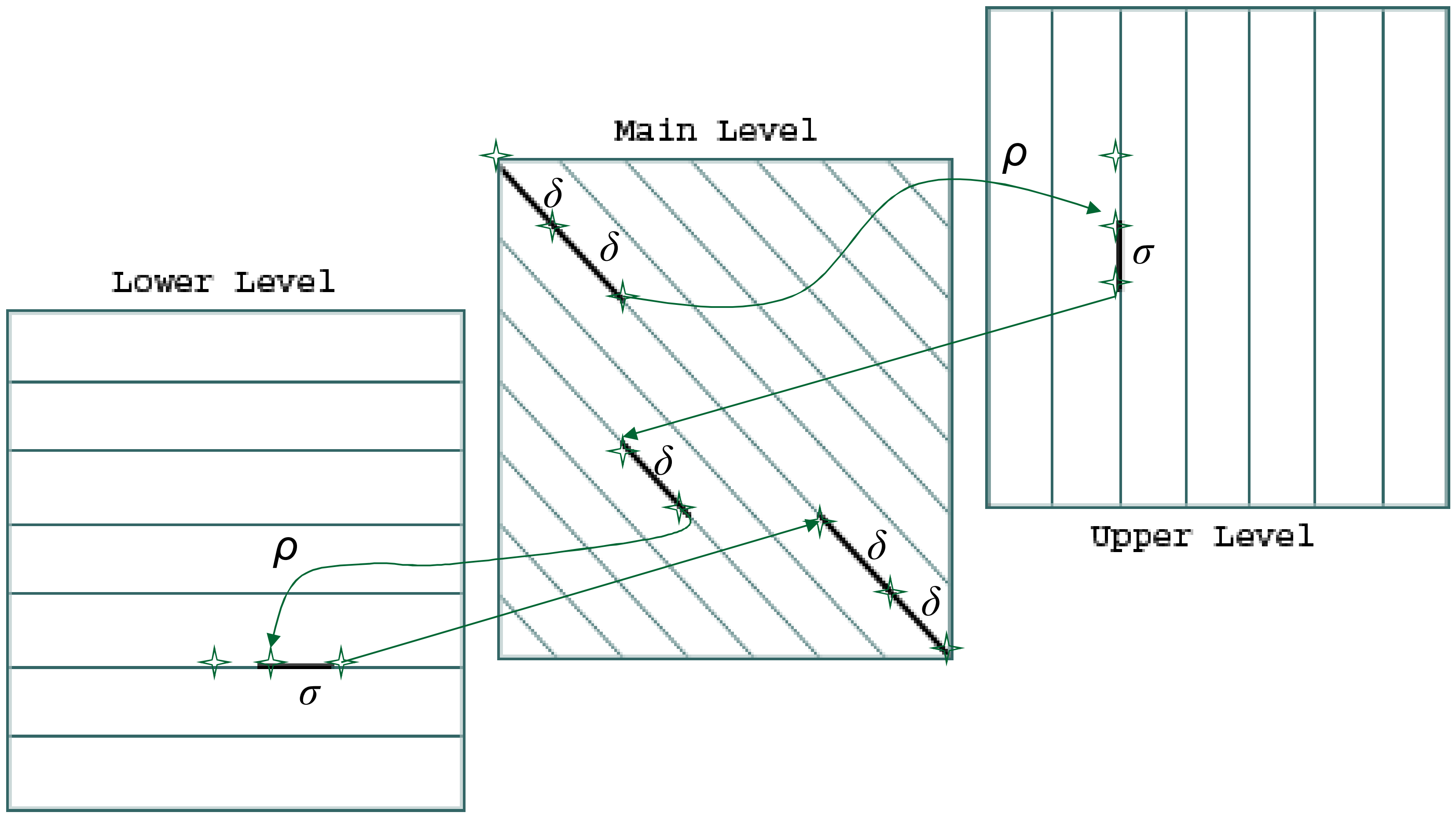


There are many such edges!

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)

Manhattan in 3 Layers



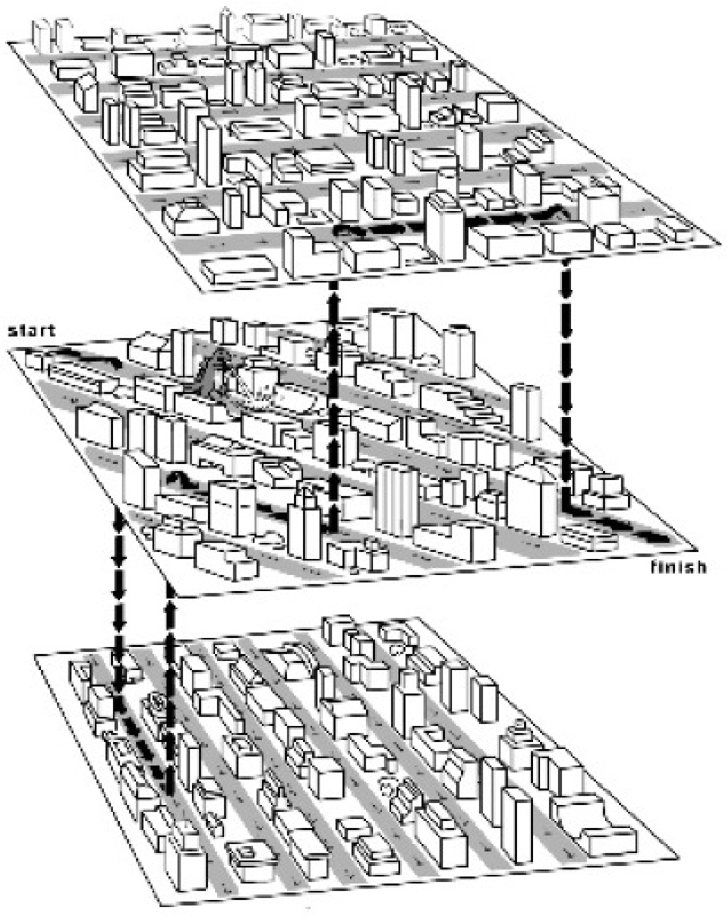
Affine Gap Penalties and 3 Layer Manhattan Grid

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

Switching between 3 Layers

- Levels:
  - The **main level** is for diagonal edges
  - The **upper level** is for horizontal edges
  - The **lower 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 ($-\rho - \sigma$)
- There is a gap extension penalty for each continuation on a level other than the main level ($-\sigma$)

The 3-leveled Manhattan Grid



Gaps in v

Matches/Mismatches

Gaps in w

Affine Gap Penalty Recurrences

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

Continue Gap in w (deletion)

Start Gap in w (deletion): from middle

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

Continue Gap in v (insertion)

Start Gap in v (insertion):from middle

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

Match or Mismatch

End deletion: from top

End insertion: from bottom