Dynamic Programming 1: Sequence Alignment

Selected Topics in Computer Intelligence - 2015

Bioinformatics Programming

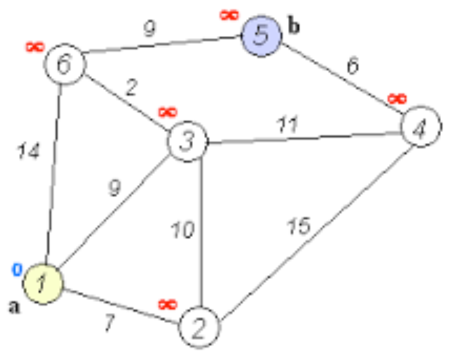
Computer Engineering, Chiang Mai University

What is Dynamic Programming?

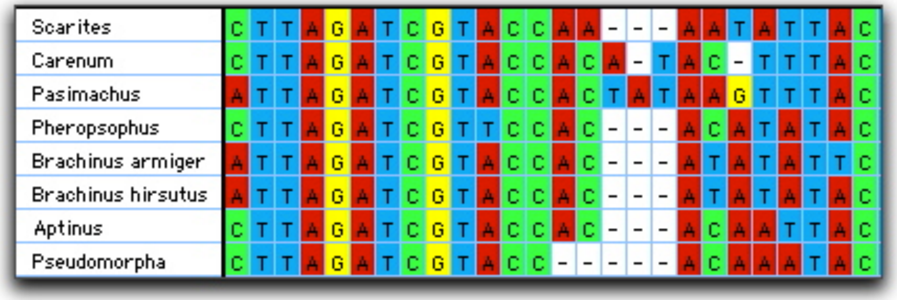
- A method for solving a complex problem by breaking it down into a collection of simpler subproblems [*wikipedia*]
  - applicable to problems with optimal substructure
  - Takes far less time than exhaustive search
- When using naive method, many subproblems are solved many times
- Dynamic programming approach seeks to a given subproblem that has been computed
  - Stored previously computed solution of subproblems
  - The next time the same solution is needed, it is simply looked up

Dynamic Programming Applications

- Applications in Mathematics, Economics
  - Optimization problems
- Computer Network – *Dijkstra's algorithm*



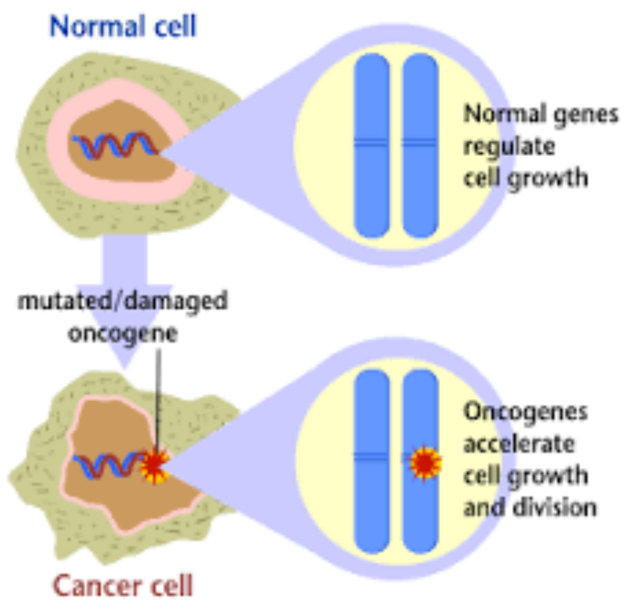
- Bioinformatics
  - Sequence alignment – DNA sequence comparison
  - Gene Prediction – make inferences about gene function
  - Protein-DNA binding
  - Transcription factor binding



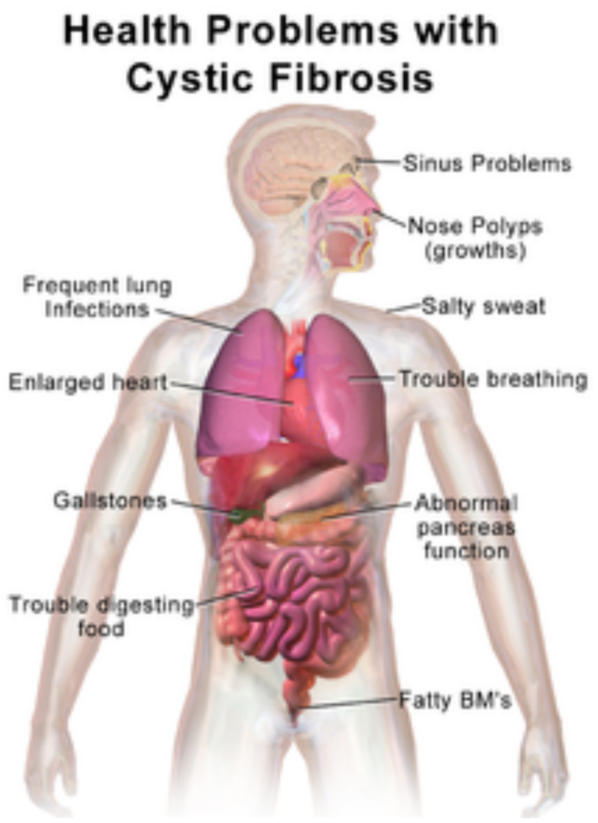
Scarites	C	T	T	A	G	A	T	C	G	T	A	C	C	A	A	-	-	-	A	A	T	A	T	T	A	C
Carenum	C	T	T	A	G	A	T	C	G	T	A	C	C	A	C	A	-	T	A	C	-	T	T	T	A	C
Pasimachus	A	T	T	A	G	A	T	C	G	T	A	C	C	A	C	T	A	T	A	A	G	T	T	T	A	C
Pheropsophus	C	T	T	A	G	A	T	C	G	T	T	C	C	A	C	-	-	-	A	C	A	T	A	T	A	C
Brachinus armiger	A	T	T	A	G	A	T	C	G	T	A	C	C	A	C	-	-	-	A	T	A	T	A	T	T	C
Brachinus hirsutus	A	T	T	A	G	A	T	C	G	T	A	C	C	A	C	-	-	-	A	T	A	T	A	T	A	C
Aptinus	C	T	T	A	G	A	T	C	G	T	A	C	C	A	C	-	-	-	A	C	A	A	T	T	A	C
Pseudomorpha	C	T	T	A	G	A	T	C	G	T	A	C	C	-	-	-	-	-	A	C	A	A	A	T	A	C

The Power of DNA Sequence Comparison

- A common approach to inferring a newly sequenced gene's function is to find similarities with genes of know function
- In 1984, a *cancer-causing v-sis oncogene* was discovered
  - The oncogene matched a normal gene with platelet-derived growth factor (PDGF)
  - Cancer might be caused by a normal growth gene being switched on at the wrong time


Normal cell
Normal genes regulate cell growth
mutated/damaged oncogene
Oncogenes accelerate cell growth and division
Cancer cell

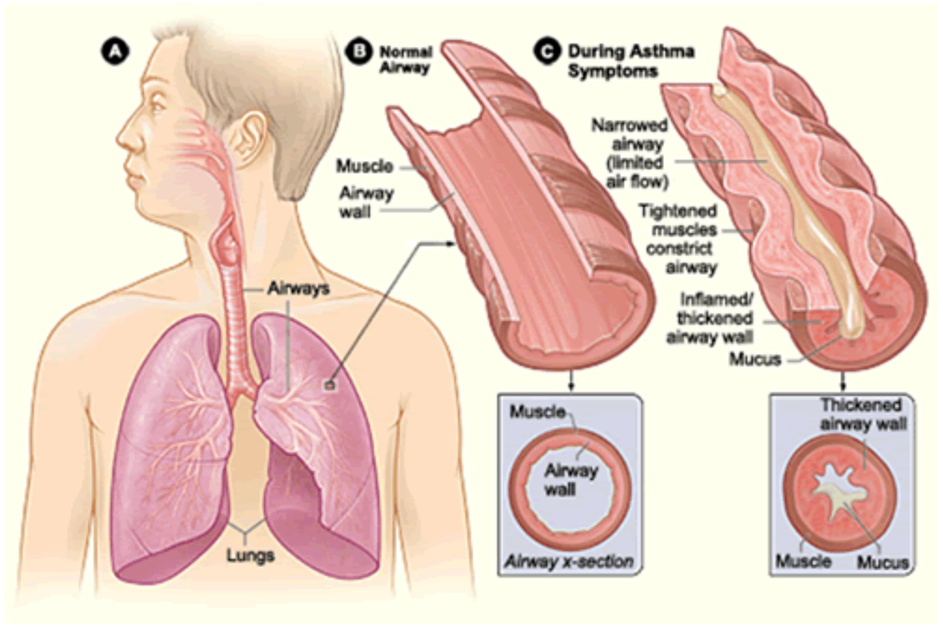
The Power of DNA Sequence Comparison

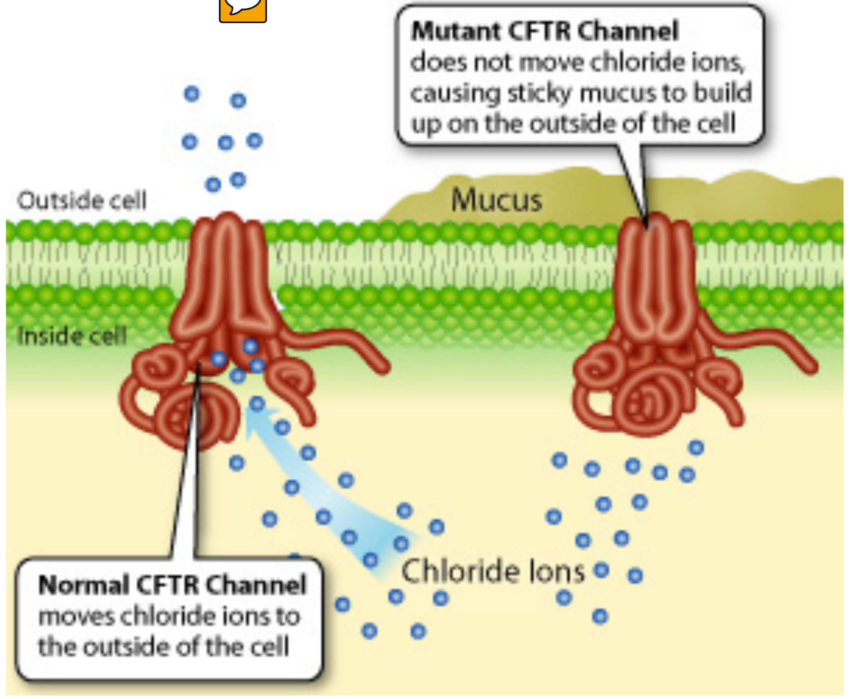


- Cystic Fibrosis (CF)
  - Defective gene causes the body to produce abnormally thick mucus that clogs the lung and leads to lung infections
  - More than 10M Americans are carriers of defective cystic fibrosis gene
- Searching for the CF gene was narrowed to a region of 1M nucleotides on the human chromosome 7

The Power of DNA Sequence Comparison

- There is similarities between some segment within the region and a gene to code for *adenosine triphosphate (ATP) binding proteins :* span the cell membrane





Mutant CFTR Channel does not move chloride ions, causing sticky mucus to build up on the outside of the cell
Outside cell
Mucus
Inside cell
Normal CFTR Channel moves chloride ions to the outside of the cell
Chloride Ions

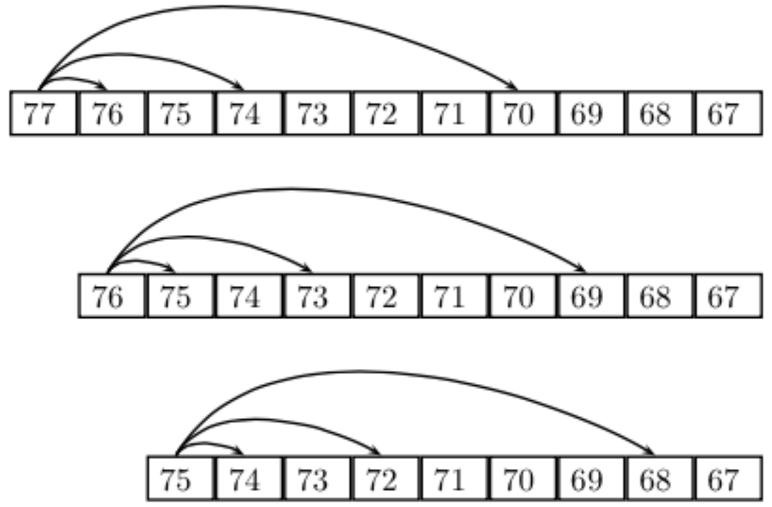
- Many applications of sequence comparison are among the key techniques for the discovery of gene function

Dynamic Programming with Change Problem

- Previously we showed that the naïve greedy solution is NOT actually a correction solution
- Recursive Approach
  - Suppose that there are 3 coin values : 1, 3, 7-cent coins

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-1} + 1 \\ bestNumCoins_{M-3} + 1 \\ bestNumCoins_{M-7} + 1 \end{cases}$$

  - Best combination for 77 cents:
    - 77-1 : 76, plus a 1-cent coin
    - 77-3 : 74, plus a 3-cent coin
    - 77-7 : 70, plus a 7-cent coin



Dynamic Programming with Change Problem

- Recursive Approach in the more general case:

$$bestNumCoins_M = \min \begin{cases} bestNumCoins_{M-c_1} + 1 \\ bestNumCoins_{M-c_2} + 1 \\ \vdots \\ bestNumCoins_{M-c_d} + 1 \end{cases} \qquad \mathbf{c} = (c_1, \dots, c_d)$$

```
RecursiveChange($M, \mathbf{c}, d$)
1  if $M = 0$
2        return 0
3  $bestNumCoins \leftarrow \infty$
4  for $i \leftarrow 1$ to $d$
5        if $M \geq c_i$
6              $numCoins \leftarrow$ RecursiveChange($M - c_i, \mathbf{c}, d$)
7              if $numCoins + 1 < bestNumCoins$
8                    $bestNumCoins \leftarrow numCoins + 1$
9  return $bestNumCoins$
```

Recalculate the optimal coin combination for a given amount repeatedly

Impractical!!!

Dynamic Programming with Change Problem

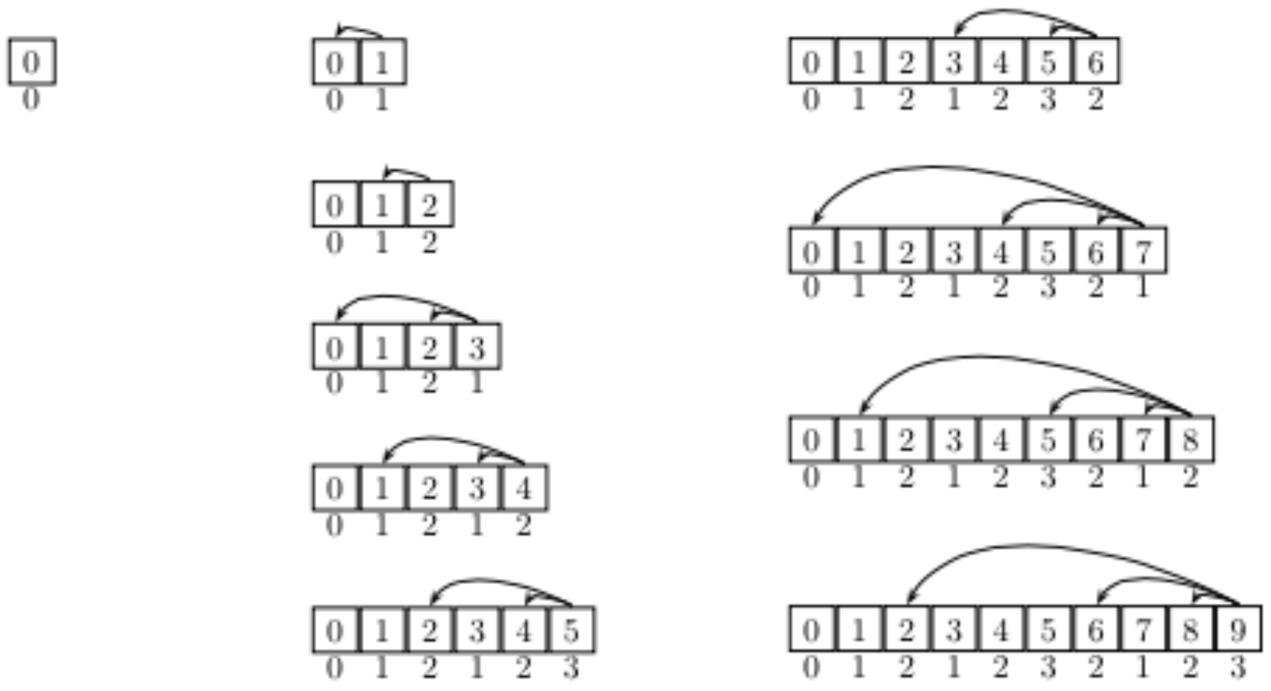
- Reversing the order of computation:
  - The solution for M relies on solutions for M - c_1, M - c_2, and so on
  - We can use previously computed solutions to form solutions to larger problems - avoiding recomputation

```
DPCHANGE($M$, $\mathbf{c}$, $d$)
1  $bestNumCoins_0 \leftarrow 0$
2  for $m \leftarrow 1$ to $M$
3      $bestNumCoins_m \leftarrow \infty$
4      for $i \leftarrow 1$ to $d$
5          if $m \geq c_i$
6              if $bestNumCoins_{m-c_i} + 1 < bestNumCoins_m$
7                  $bestNumCoins_m \leftarrow bestNumCoins_{m-c_i} + 1$
8  return $bestNumCoins_M$
```

bestNumberCoins
array storing optimal solution for a given *m*, *m=0* to *M*

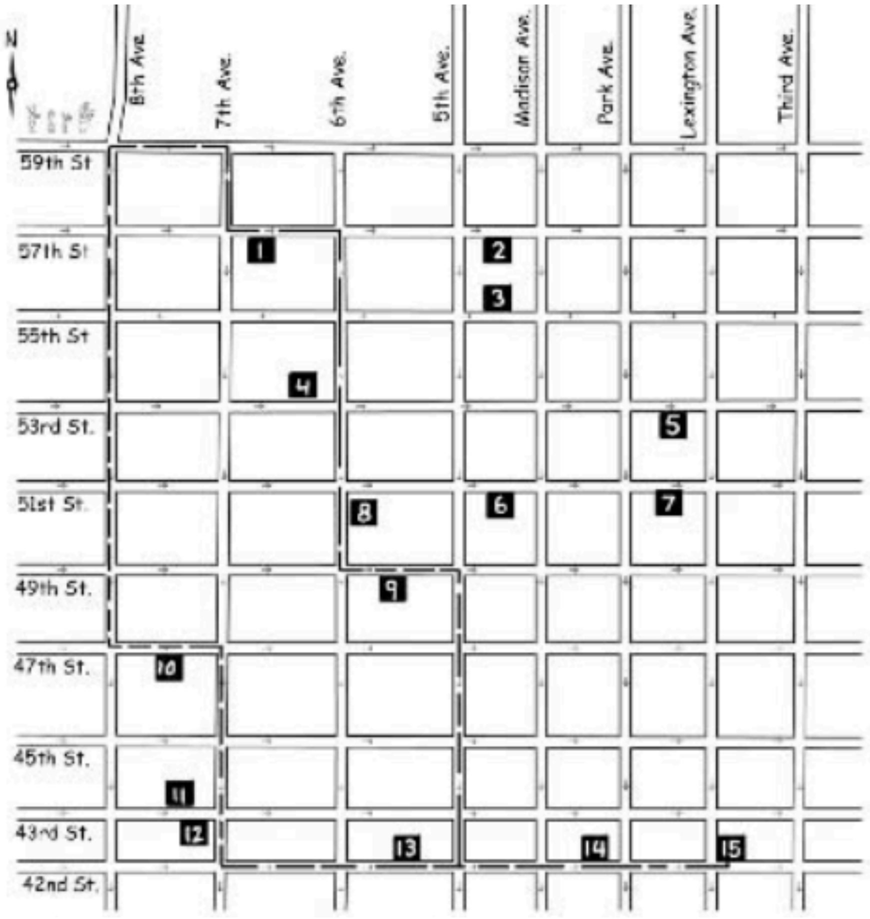
Dynamic Programming with Change Problem

- Reversing the order of computation:
  - The solution for 9 cents ($bestNumCoins_9$)
    - Depends on the solution of 8, 6, and 2 cents



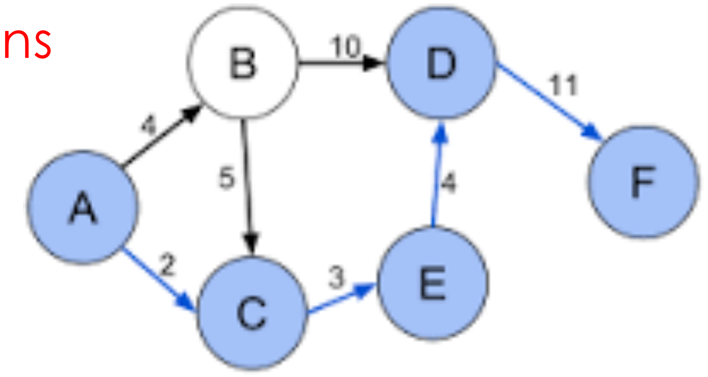
The Manhattan Tourist Problem

- Based on the Manhattan tourist problem, we can use it to describe DNA sequence alignment problem
- Manhattan Tourist Problem:
  - Tourist want to see as many as attractions as possible
  - Tourist are allowed to move along traffic direction – one way
  - Which path give the maximum number of attractions to visit?



The Manhattan Tourist Problem

- The map can be represented as a graph
  - Vertices – the intersections of streets
  - Edges – the streets that connect different intersections
    - Weight – the number of attractions on every block
- Path – a continuous sequence of edges
  - Length of a path – the sum of the weights in the path
- Goal of the Manhattan Tourist problem
  - Find the a path with the maximum # attractions
  - The longest path in the graph

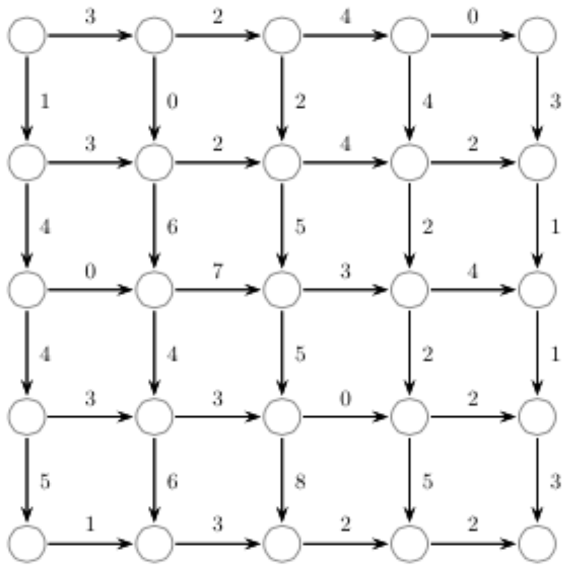


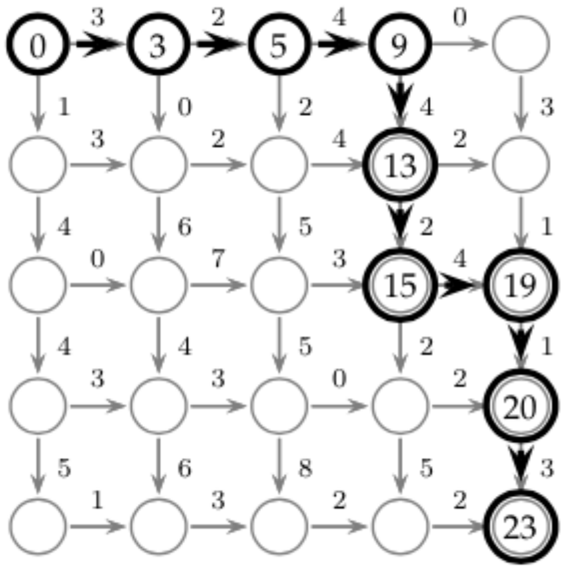
The Manhattan Tourist Problem

- Problem Formulation

Manhattan Tourist Problem:
Find a longest path in a weighted grid.

> **Input:** A weighted grid G with two distinguished vertices: a *source* and a *sink*.
>
> **Output:** A longest path in G from *source* to *sink*.





Source vertex: (0, 0)
Sink vertex: (4, 4)

The Manhattan Tourist Problem

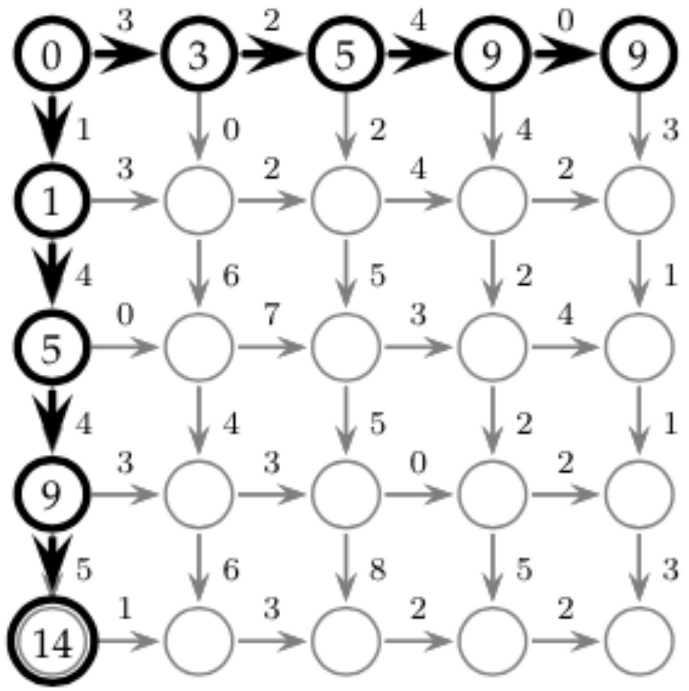
- The brute force approach is NOT an option even for a moderately large grid
- What about the greedy approach?
  - Choosing between two possible directions based on # attractions tourists would see if moved one block on each direction
  - This may provide a "good" option in the beginning

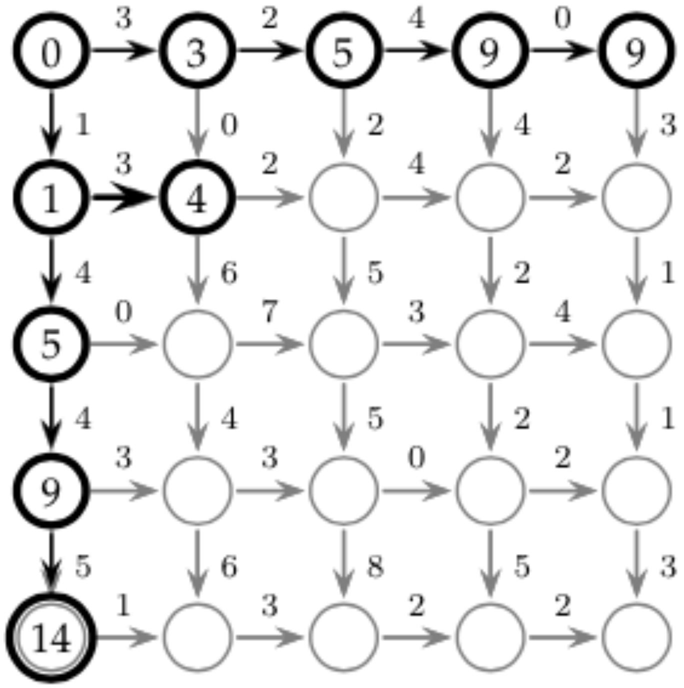
The Manhattan Tourist Problem

- The more general problem
  - Finding the longest path from source $(0, 0)$ to arbitrary vertex (i, j)
  - The length of the best path : $S_{i,j}$ $(0 \le i \le n \text{ and } 0 \le j \le m)$
  - $S_{n,m}$ – the weight of the path that is the solution to the problem
- Solving the more general problem
  - Finding the longest length to all vertex (i, j)
  - Starting from $S_{i,0}$ $(0 \le i \le n)$
  - Starting from $S_{0,j}$ $(0 \le j \le n)$

The Manhattan Tourist Problem

- Example

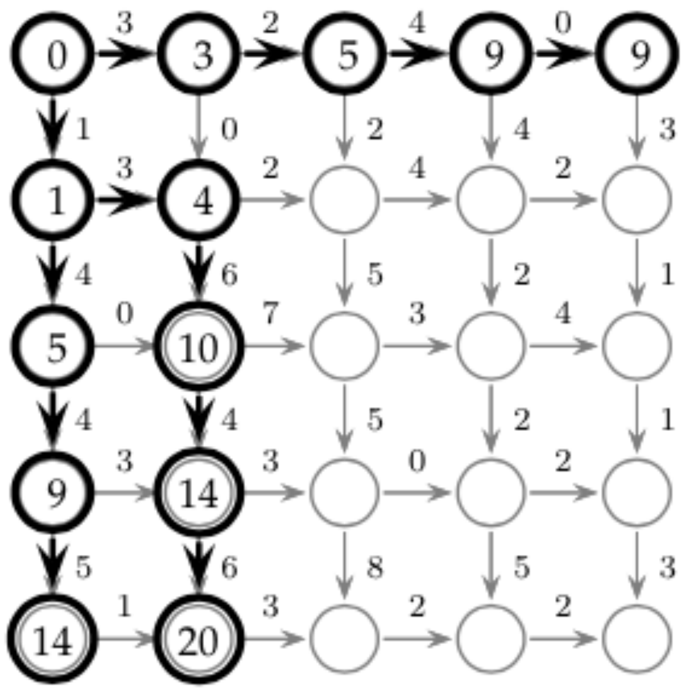


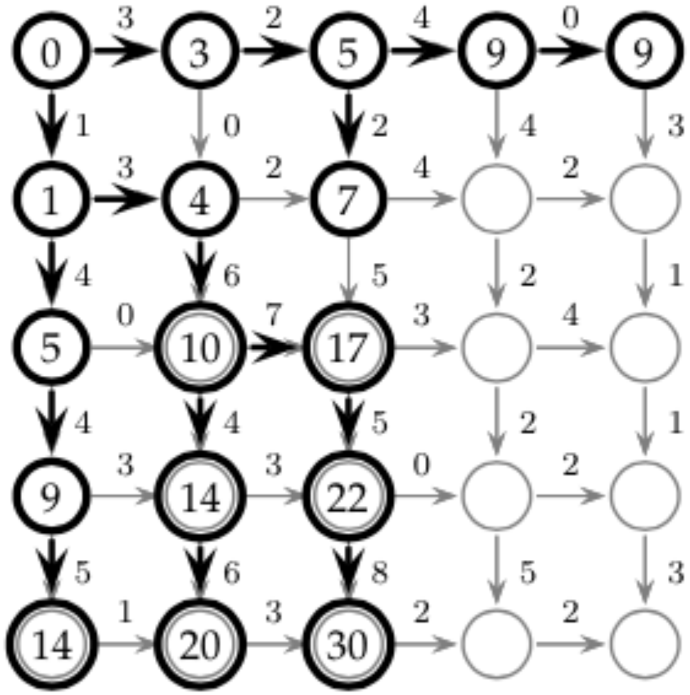


$$s_{1,1} = \max \begin{cases} s_{0,1} + \text{ weight of the edge (block) between (0,1) and (1,1)} \\ s_{1,0} + \text{ weight of the edge (block) between (1,0) and (1,1)} \end{cases}$$

The Manhattan Tourist Problem

- Example

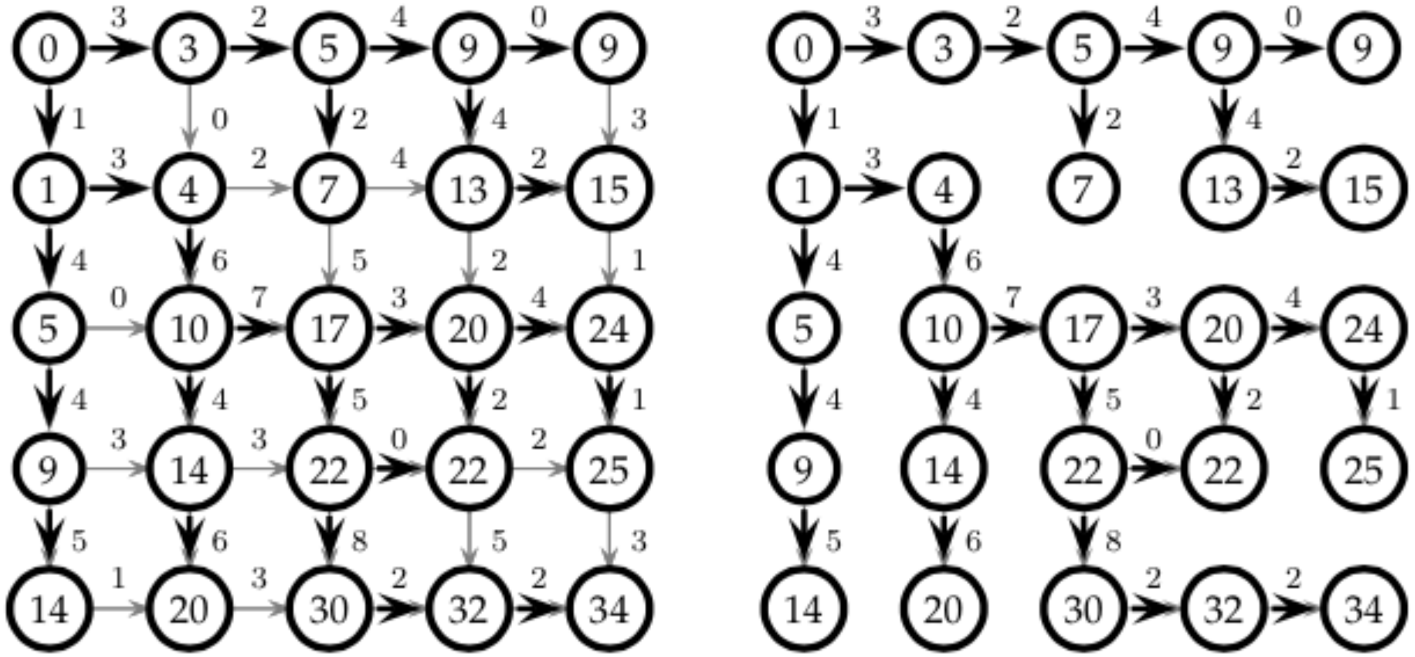




$$s_{1,2} = \max \begin{cases} s_{1,1} + \text{ weight of the edge between (1,1) and (1,2)} \\ s_{0,2} + \text{ weight of the edge between (0,2) and (1,2)} \end{cases}$$

The Manhattan Tourist Problem

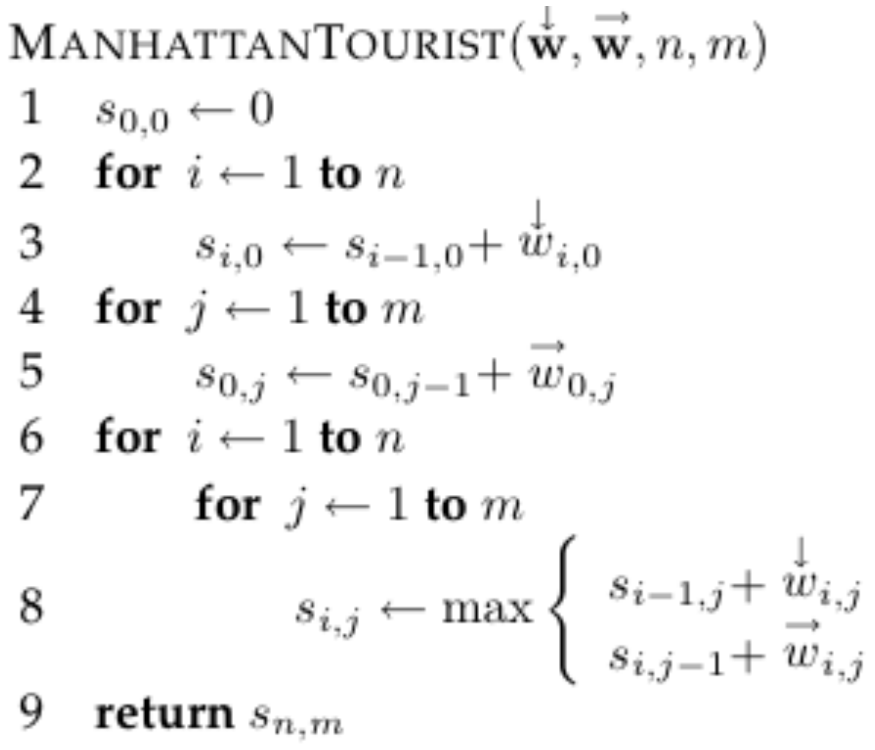
- Example



$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{ weight of the edge between } (i-1,j) \text{ and } (i,j) \\ s_{i,j-1} + \text{ weight of the edge between } (i,j-1) \text{ and } (i,j) \end{cases}$$

The Manhattan Tourist Problem

- The algorithm MANHATTANTOURIST



```
MANHATTANTOURIST($\overset{\downarrow}{\mathbf{w}}, \overset{\rightarrow}{\mathbf{w}}, n, m$)
1  $s_{0,0} \leftarrow 0$
2  for $i \leftarrow 1$ to $n$
3      $s_{i,0} \leftarrow s_{i-1,0} + \overset{\downarrow}{w}_{i,0}$
4  for $j \leftarrow 1$ to $m$
5      $s_{0,j} \leftarrow s_{0,j-1} + \overset{\rightarrow}{w}_{0,j}$
6  for $i \leftarrow 1$ to $n$
7      for $j \leftarrow 1$ to $m$
8          $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} + \overset{\downarrow}{w}_{i,j} \\ s_{i,j-1} + \overset{\rightarrow}{w}_{i,j} \end{cases}$
9  return $s_{n,m}$
```

$\overset{\downarrow}{w}$: 2D array of vertical weights

$\vec{w}$: 2D array of horizontal weights

$\overset{\downarrow}{w}_{i,j}$: weight of the edge from $(i\text{-}1, j)$ to (i, j)

$\vec{w}_{i,j}$: weight of the edge from $(i, j\text{-}1)$ to (i, j)

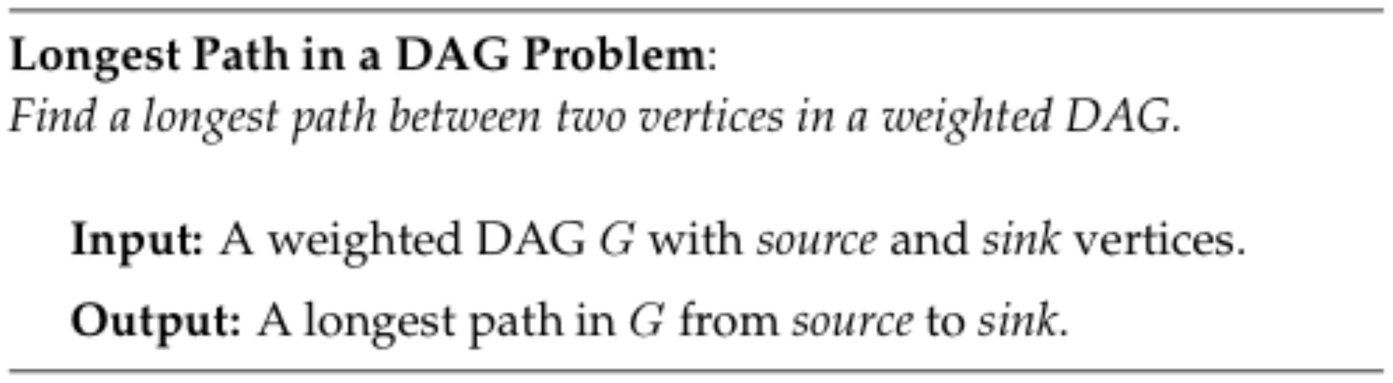
The Manhattan Tourist Problem

- Most of the dynamic programming algorithms in the context of DNA sequence comparison will be similar to MANHATTANTOURIST – e.g., sequence comparison algorithm
  - Using appropriate model to the specific biological problem
  - Defining the weight that reflect the cost of mutation
- However, the real Manhattan is NOT a perfect grid
  - We can use *Directed Acyclic Graph* (DAGs) to represent the imperfect grid

Longest Path in DAGs

- Directed Acyclic Graphs
  - Directed edges – one directional edge
  - No directed cycles – no loops
  - Graph representation:

$G = (V, E)$, V is the set of vertices, E is the set of edges



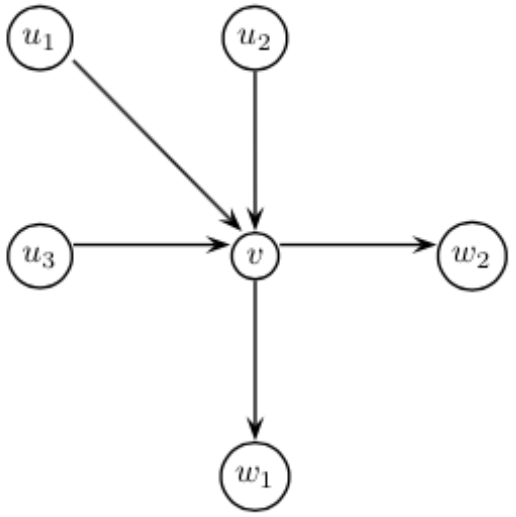
Longest Path in a DAG Problem:
Find a longest path between two vertices in a weighted DAG.

Input: A weighted DAG G with *source* and *sink* vertices.

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

Longest Path in DAGs

- Directed Acyclic Graphs
  - *indegree* - # edges entering a vertex
  - *outdegree* - # edges leaving a vertex
  - *Predecessor* – any vertex that can be reached by traveling backwards along inbound edge



- Suppose a vertex v has *indegree* of 3
  - Set of predecessor $\{ u_1, u_2, u_3 \}$
  - The longest path to v:

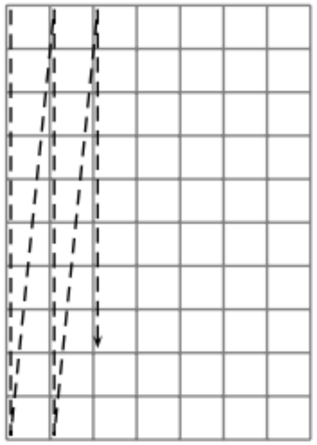
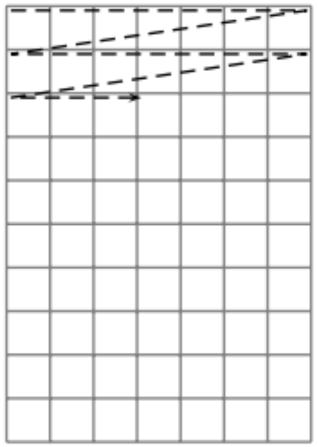
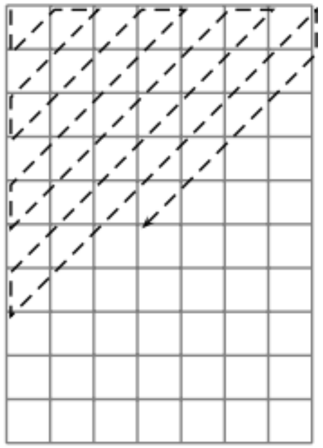
$$s_v = \max \begin{cases} s_{u_1} + \text{weight of edge from } u_1 \text{ to } v \\ s_{u_2} + \text{weight of edge from } u_2 \text{ to } v \\ s_{u_3} + \text{weight of edge from } u_3 \text{ to } v \end{cases}$$

Longest Path in DAGs

- In general,

$$s_v = \max_{u \in Predecessors(v)} (s_u + \text{ weight of edge from } u \text{ to } v)$$

- The order to visit the vertices – *topological ordering*
  - S_u of all predecessors of v must have been computed before visiting the vertex v
- Three popular strategies
  - Column by column
  - Row by row
  - Diagonal by diagonal

Edit Distance and Alignment

- What is "sequence similarity"?
- Hamming distance is NOT typically used to compare DNA sequence or protein sequences
  - Assumes that the i^{th} symbol of one sequence is aligned against the i^{th} symbol of the other
- i^{th} symbol in one sequence is often corresponds to a symbol at a different (and unknown) position in the other
  - DNA replication errors cause substitutions, insertions, and deletions leading to modified DNA text

Edit Distance and Alignment

- Example of sequence similarity

```
A T A T A T A T -
  : : : : : : :
- T A T A T A T A
```

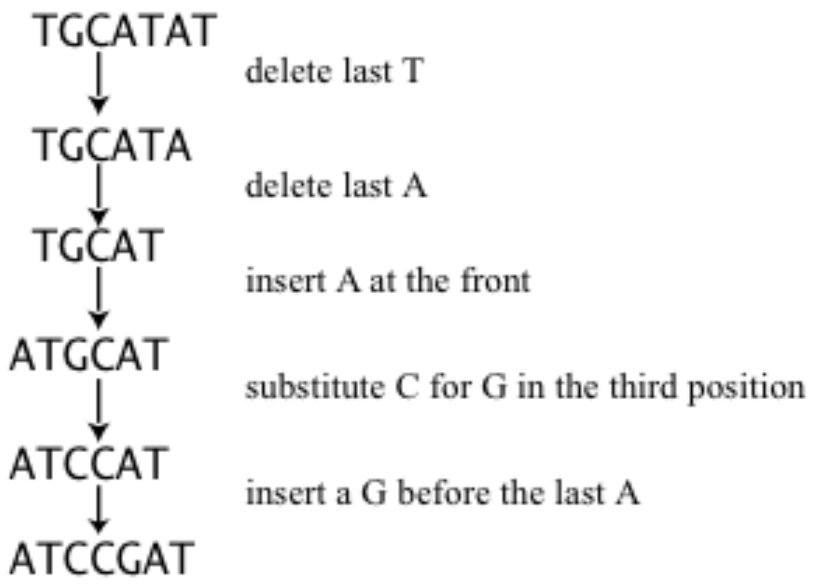
(a) Alignment of ATATATAT against TATATATA.

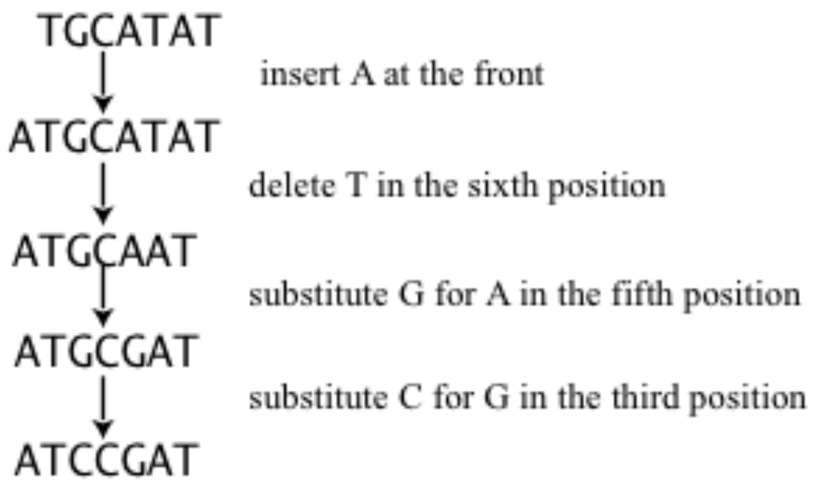
```
A T A T A T A T
  : : : :   : :
- T A T A - A T
```

(b) Alignment of ATATATAT against TATAAT.

Edit Distance and Alignment

- Edit Distance
  - Introduced by *Valadimir Levenshtein* in 1966
  - The minimum # editing operations needed to transform one string into another
    - Insertion, deletion, and substitution




TGCATAT
insert A at the front
ATGCATAT
delete T in the sixth position
ATGCAAT
substitute G for A in the fifth position
ATGCGAT
substitute C for G in the third position
ATCCGAT

Edit distance between the strings is 4

Edit Distance and Alignment

- Edit Distance allows to compare strings of different length
- The alignment of string v (n-character) and w (m-character)
  - Two-row matrix with v in the 1st row and w in the 2nd row
  - Characters in each string appear in order, NOT necessarily adjacently
  - ***Matches*** – columns that contain the same letter in both row
  - ***Mismatches*** – columns that contain different letter
  - ***indels*** – columns that contain one space
    - ***insertions*** – contains space in the top row
    - ***deletions*** – contains space in the bottom row

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

- Resulting matrix and a path:

$$\binom{0}{0}\binom{1}{1}\binom{2}{2}\binom{2}{3}\binom{3}{4}\binom{4}{5}\binom{5}{5}\binom{6}{6}\binom{7}{6}\binom{7}{7}$$

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

Edit Distance and Alignment

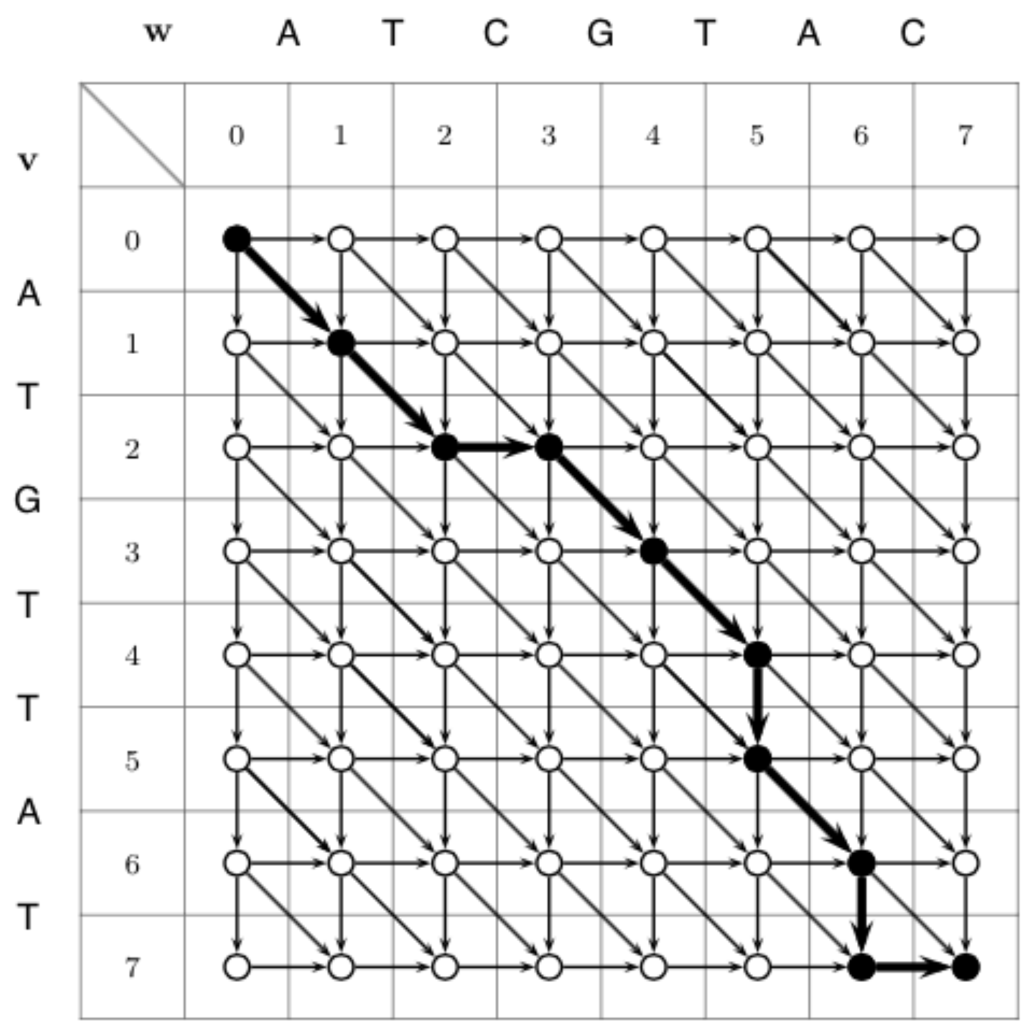
Alignment Grid

row	→	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
col	→	0	1	2	3	4	5	5	6	6	7

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

Every alignment corresponds to a path in the alignment grid from *(0, 0)* to *(n, m)*

By choosing different scoring function, we can solve different string comparison prob.



Longest Common Subsequences

- The simplest form of a sequence similarity analysis
- LCS problem allows only insertion and deletion
  - Eliminates substitute operation
- Subsequence (of a string)
  - An ordered sequence of characters from a string
  - Not necessarily consecutive

If v = ATTGCTA then

AGCA, ATTA are subsequences of v

TGTT, TCG are ***NOT*** subsequences

Longest Common Subsequences

- Common subsequence of two string: v and w

$$v_{i_t} = w_{j_t} \text{ for } 1 \le t \le k.$$

$$1 \le i_1 < i_2 < \cdots < i_k \le n$$

$$1 \le j_1 < j_2 < \cdots < j_k \le m$$

- e.g., TCTA is common subsequence to …

  ATCTGAT and TGCATA

- We are looking for the longest CS

  $s(v, w)$ – length of LCS of v and w

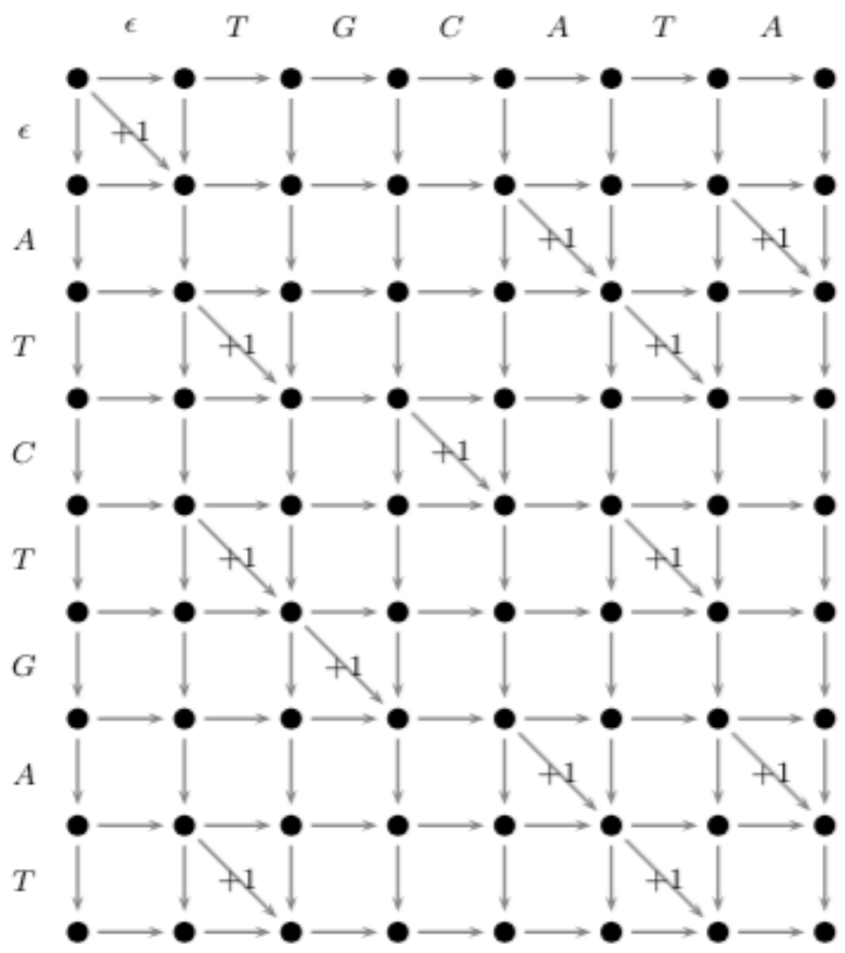
  $d(v, w)$ – edit distance between v and w

Alignment:

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

$$d(v, w) = n + m - 2s(v, w)$$

Longest Common Subsequences



- Every common subsequence corresponds to an alignment with no mismatches
  - By removing all diagonal edges whose characters do NOT match, we get an LCS edit graph
- Looks like Manhattan Tourist Problem?

$$s_{i,0} = s_{0,j} = 0 \qquad \begin{array}{l} 1 \le i \le n \\ 1 \le j \le m \end{array}$$

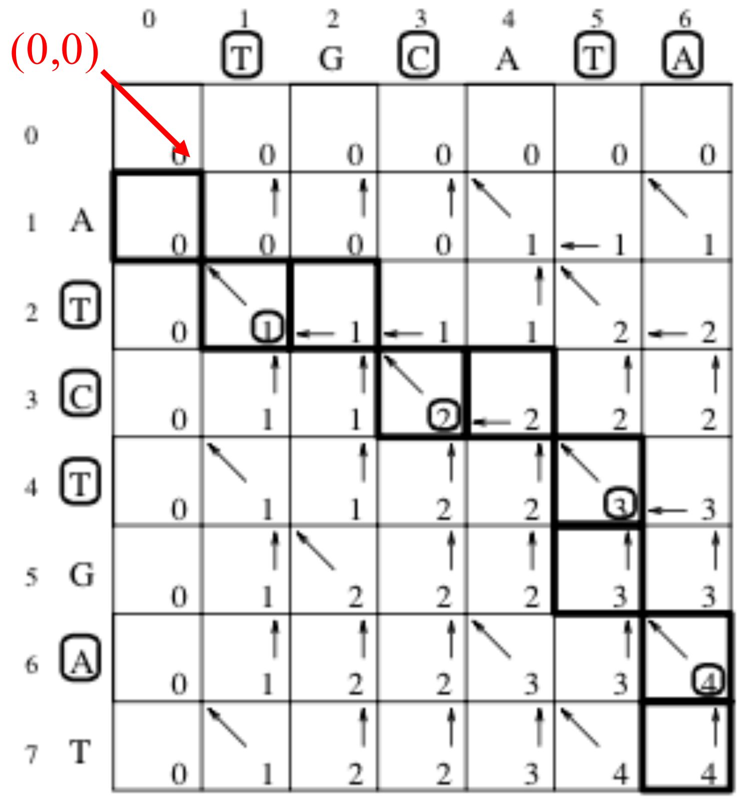
$$s_{i,j} = \max \begin{cases} s_{i-1,j} + 0 \\ s_{i,j-1} + 0 \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_j \end{cases}$$

Longest Common Subsequences

LCS algorithm – similarity score

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

$n = 7$
$m = 6$



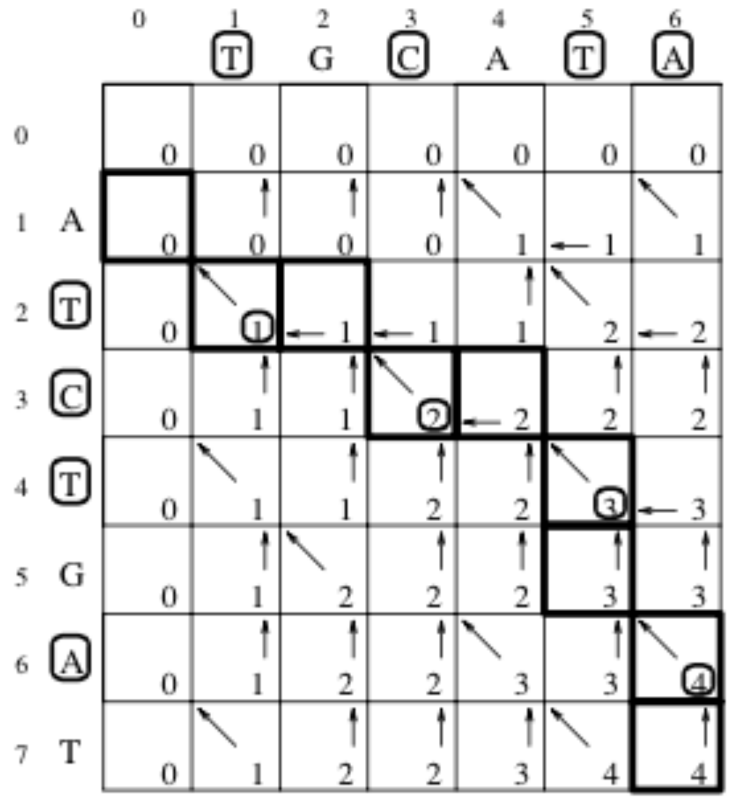
b - backtracking pointers:

- Takes $\leftarrow$, $\uparrow$, or $\nwarrow$ (deletion, insertion, match)
- specify which of the cases holds

Longest Common Subsequences

- PRINTLCS algorithm
  - Recursively print out

```
PRINTLCS($\mathbf{b}, \mathbf{v}, i, j$)
  if $i = 0$ or $j = 0$
        return
  if $b_{i,j} =$ "$\nwarrow$"
        PRINTLCS($\mathbf{b}, \mathbf{v}, i-1, j-1$)
        print $v_i$
  else
        if $b_{i,j} =$ "$\uparrow$"
              PRINTLCS($\mathbf{b}, \mathbf{v}, i-1, j$)
        else
              PRINTLCS($\mathbf{b}, \mathbf{v}, i, j-1$)
```

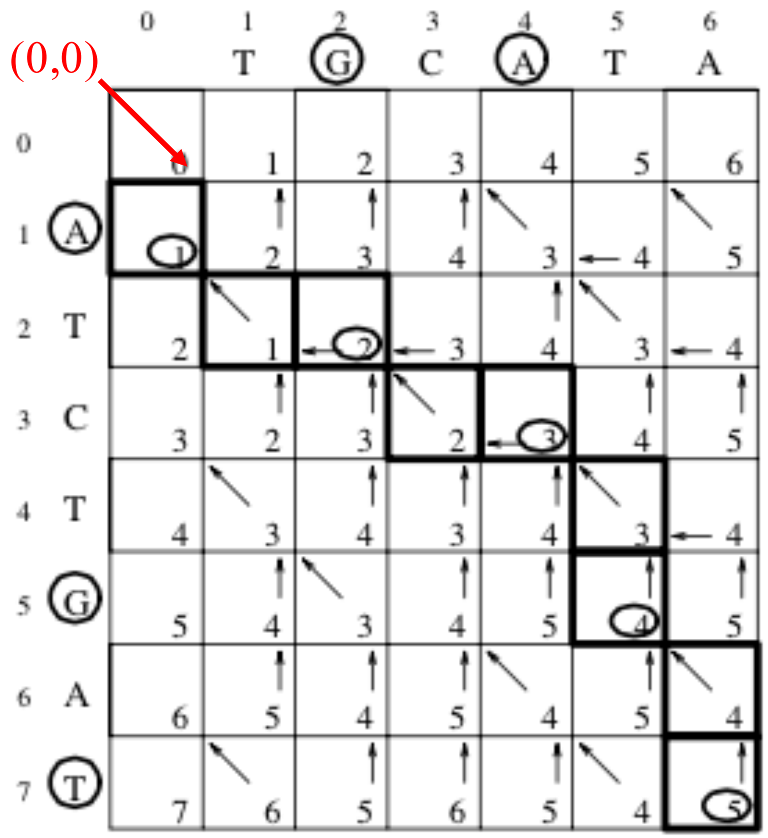


b - backtracking pointers:

- Takes $\leftarrow$, $\uparrow$, or $\nwarrow$ (deletion, insertion, match)
- specify which of the cases holds

Longest Common Subsequences

- LCS algorithm – edit distance
  - Initial conditions: $d_{i,0} = i,\ d_{0,j} = j$ $\quad 1 \le i \le n \quad 1 \le j \le m$

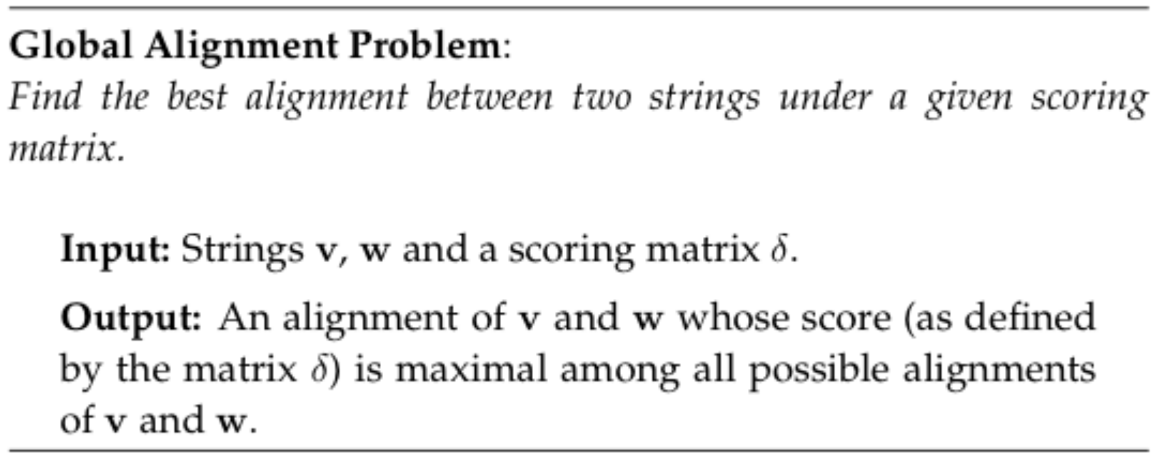


$n = 7$
$m = 6$

$$d_{i,j} = \min \begin{cases} d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \\ d_{i-1,j-1}, & \text{if } v_i = w_j \end{cases}$$

Global Sequence Alignment

- Generalizing scoring
  - Extend the k-letter alphabet to include gap, '–'
    - k is typically 4 (for DNA) and 20 (for protein)
  - δ - scoring matrix of size $(k+1) \times (k+1)$
    - $\delta(x,y)$ – the score of column (x, y)
    - Alignment score – sum of the scores of the columns



Global Alignment Problem:
Find the best alignment between two strings under a given scoring matrix.

Input: Strings **v**, **w** and a scoring matrix δ.

Output: An alignment of **v** and **w** whose score (as defined by the matrix δ) is maximal among all possible alignments of **v** and **w**.

Global Sequence Alignment

- *Needleman-Wunsch Algorithm*
  - Score $s_{i,j}$ of an optimal alignment:

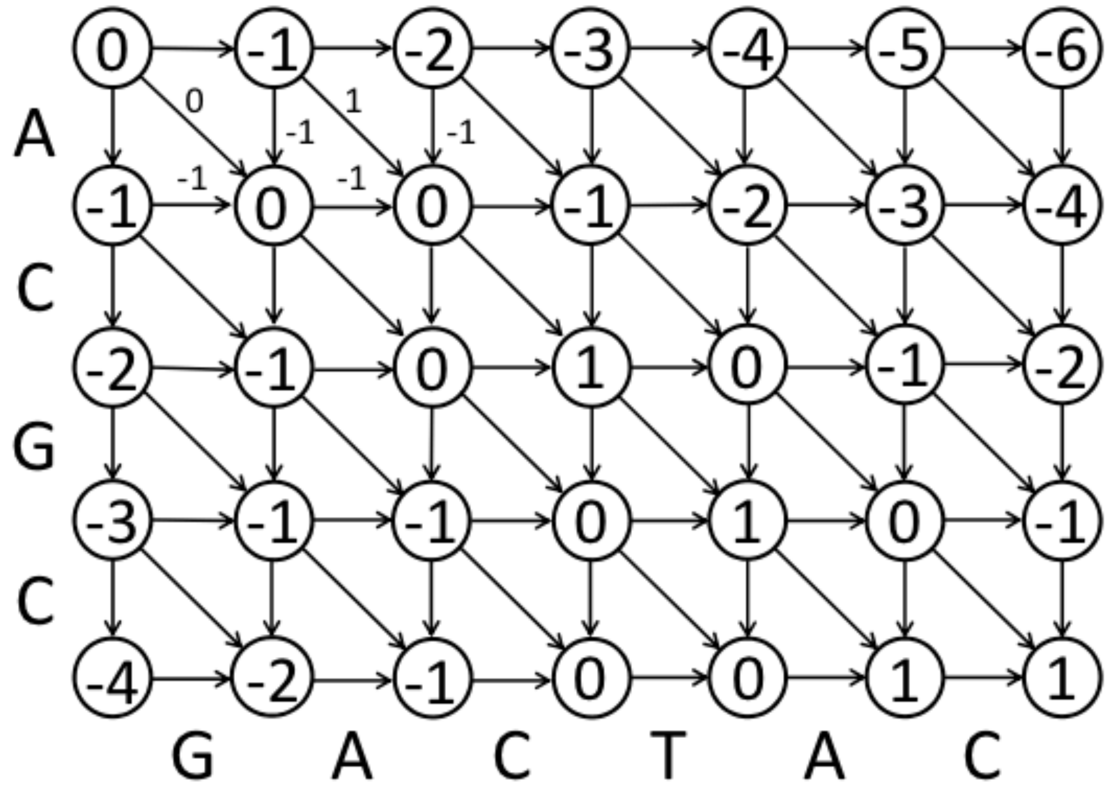
$$s_{i,j} = \max \begin{cases} s_{i-1,j} - \sigma \\ s_{i,j-1} - \sigma \\ s_{i-1,j-1} - \mu, \text{ if } v_i \neq w_j \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$$

  - Mismatches are penalized by: $-\mu$
  - Indels (or gaps) are penalized by: $-\sigma$
  - Matches are rewarded with: $+1$
  - Resulting score:

$$\#matches - \mu \cdot \#mismatches - \sigma \cdot \#indels$$

Global Sequence Alignment

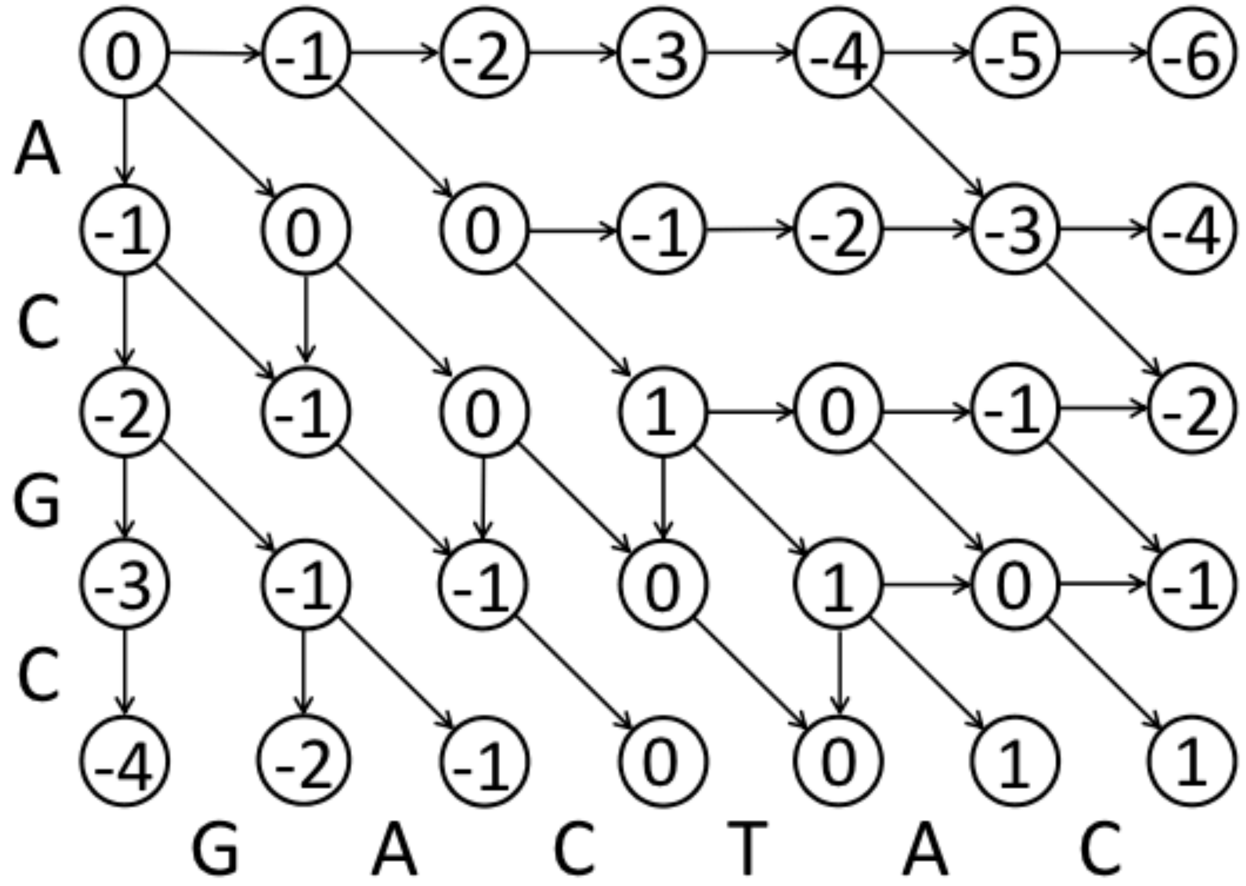
- Example



Scores: Match +1 Mismatch 0 Gap -1

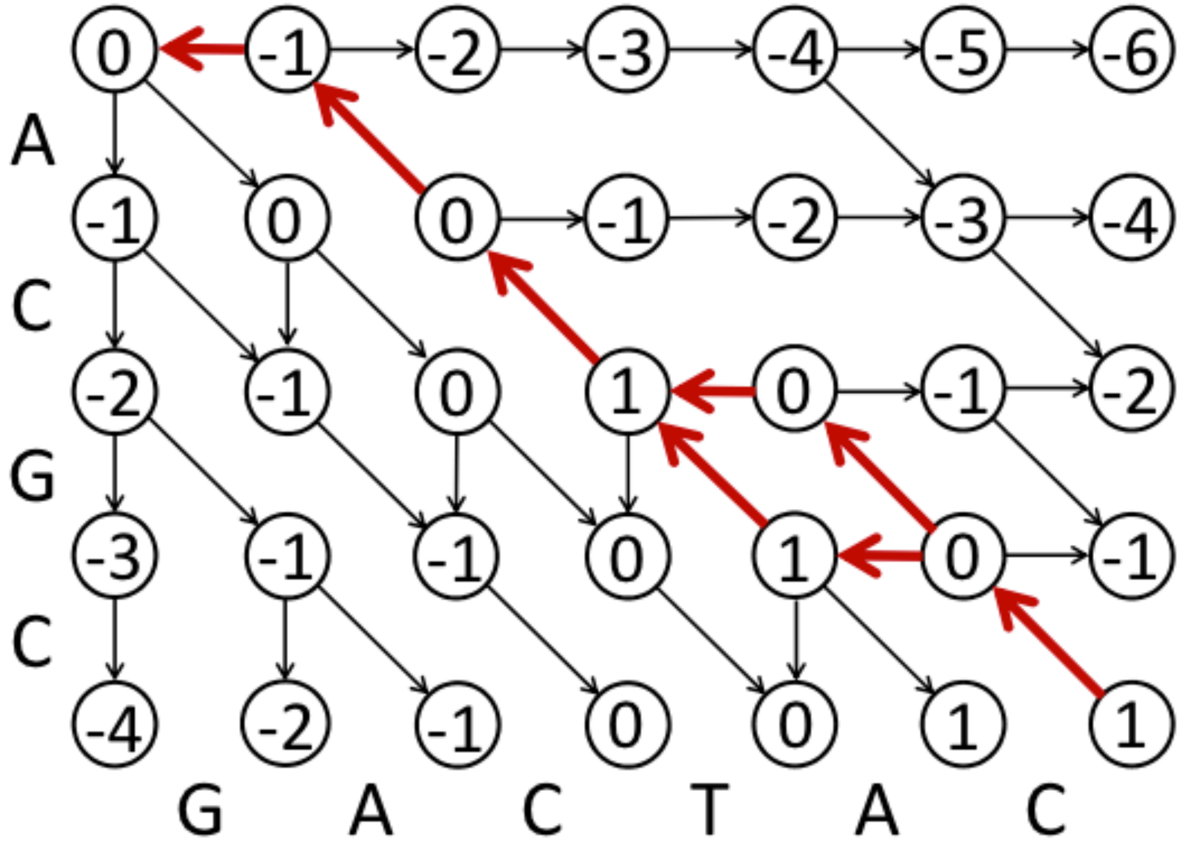
Global Sequence Alignment

- Example (cont.)



Global Sequence Alignment

- Example (cont.)



Optimal alignments:

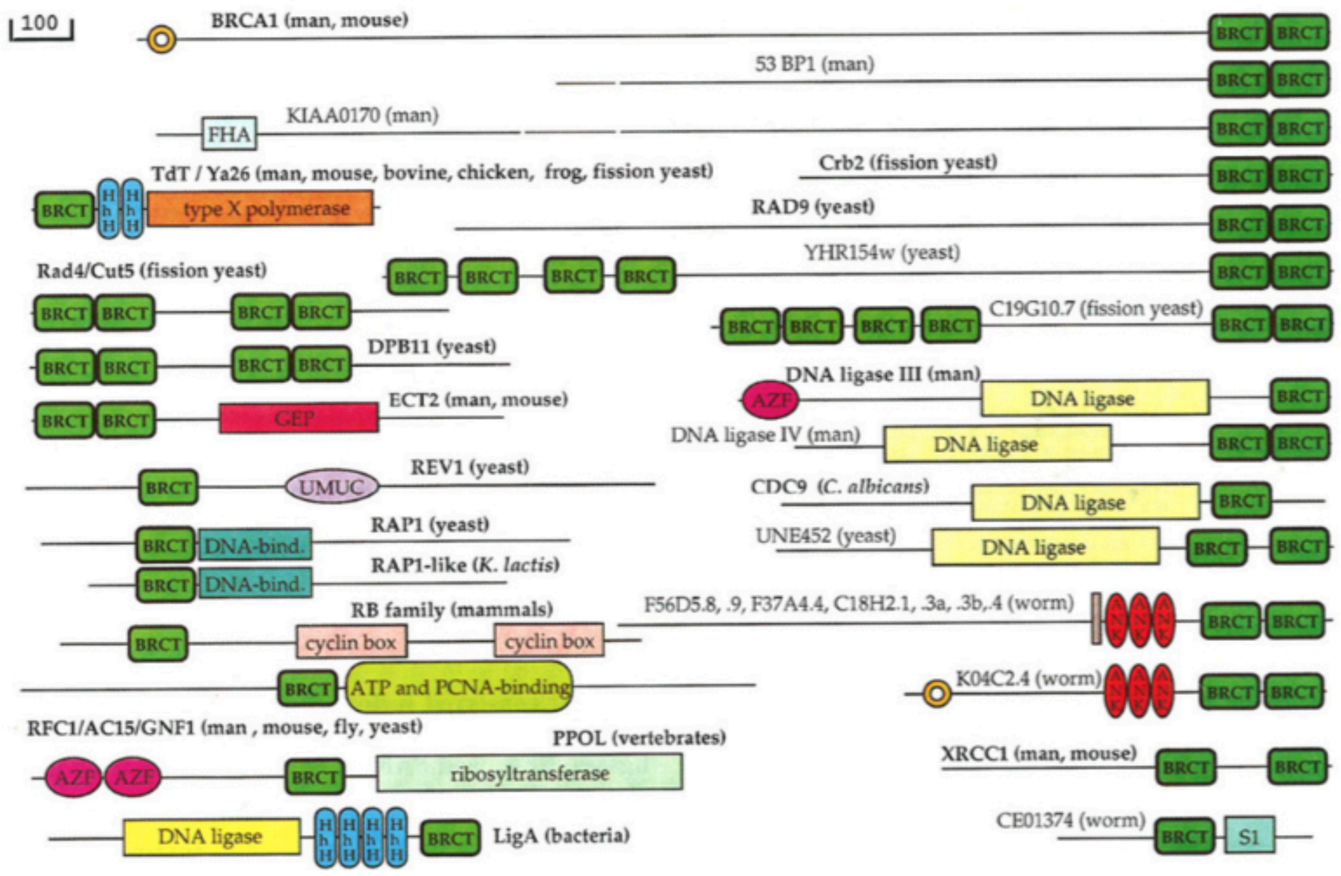
```
-ACG-C        -AC-GC
GACTAC  and   GACTAC
```

Local Sequence Alignment

- GSA problem seek similarities between 2 entire strings
  - Protein sequences from the same family
  - Often very conserved
  - Almost have the same length in organisms
- In many biological applications
  - Score of an alignment between substrings may be larger than that of the entire strings
- *Homeobox genes* – regulate embryonic development
  - Present in variety of species (very different)
  - One region in each gene is highly conserved

Local Sequence Alignment

- Family of proteins shared only isolated regions of similarity were found - superfamily



Local Sequence Alignment

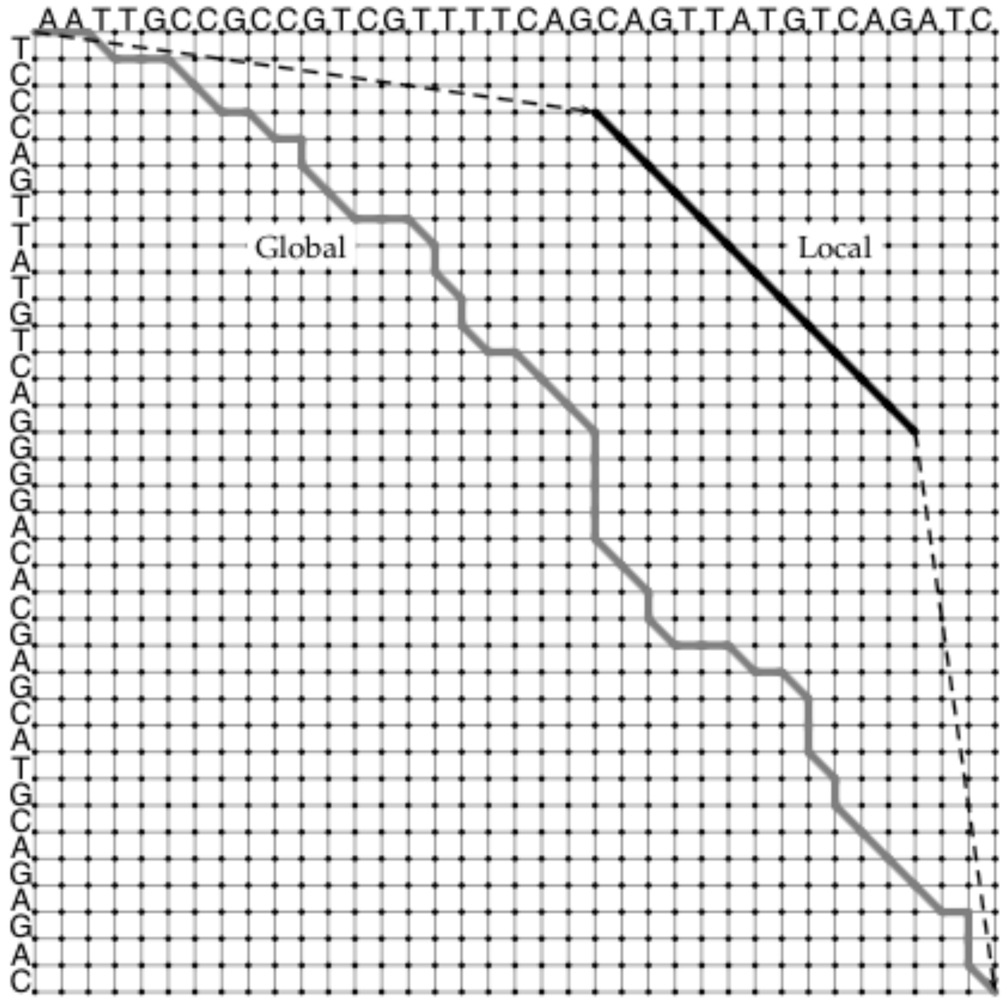
- Global vs. Local Alignment

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

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

Local Sequence Alignment

- Global vs. Local Alignment



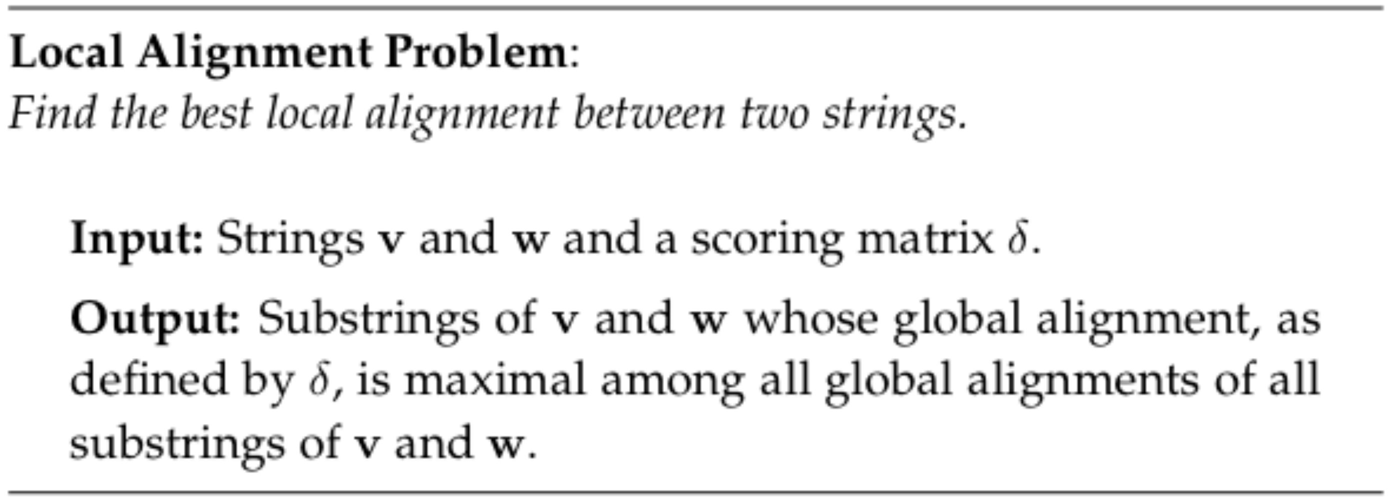
AATTGCCGCCGTCGTTTTCAGCAGTTATGTCAGATC
TCCCAGTTATGTCAGGGGACACGAGCATGCAGAGAC
Global
Local

Local alignment has much worse score according to the global scheme

Local alignment correctly locates the conserved domain

Local Sequence Alignment

- In 1981, Temple Smith & Michael Waterman modified the global algorithm that solves the local sequence alignment
- Biologist attempts to maximize the alignment score over substring $v_i \ldots v_{i'}$ of v and $w_i \ldots w_{i'}$ of w
  - This is Local Alignment Problem



Local Alignment Problem:
Find the best local alignment between two strings.

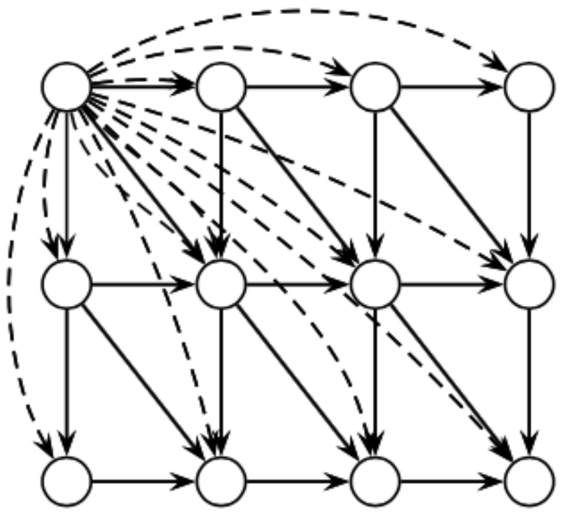
Input: Strings **v** and **w** and a scoring matrix δ.

Output: Substrings of **v** and **w** whose global alignment, as defined by δ, is maximal among all global alignments of all substrings of **v** and **w**.

Local Sequence Alignment

- Finding the longest path among paths between arbitrary vertices *(i, j)* and *(i', j')* in the edit graph
- Making vertex *(0, 0)* a predecessor of every vertex *(i, j)*
  - Adding edges of weight *0* from *(0, 0)* to every vertices
  - Provide a "free ride" from source to any other vertices
- Finding the longest path to every other vertex

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



Local Sequence Alignment

- Backtracking starts at the highest scoring matrix cell and proceeds until a cell with score=0
  - Giving the highest scoring local alignment
- Scoring matrix (δor *H*):

Note: this algorithm puts
v in row direction
w in column direction

$$H(i,0) = 0,\ 0 \le i \le m$$
$$H(0,j) = 0,\ 0 \le j \le n$$
$$H(i,j) = \max \begin{Bmatrix} 0 & \\ H(i-1,j-1) + \ s(a_i,b_j) & \text{Match/Mismatch} \\ \max_{k\ge1}\{H(i-k,j) + \ W_k\} & \text{Deletion} \\ \max_{l\ge1}\{H(i,j-l) + \ W_l\} & \text{Insertion} \end{Bmatrix},\ 1 \le i \le m, 1 \le j \le n$$

$s(a,b)$ is a similarity function on the alphabet
$H(i,j)$ – is the maximum Similarity-Score between a suffix of a[1...i] and a suffix of b[1...j]
W_i is the gap-scoring scheme

Local Sequence Alignment

- ACACACTA vs. AGCACACA

$$H(i,j) = \max \begin{cases} 0 \\ H(i-1,j-1) + s(a_i,b_j) \\ \max_{k\geq 1}\{H(i-k,j) + W_k\} \\ \max_{l\geq 1}\{H(i,j-l) + W_l\} \end{cases}$$

$s(a,b) = +2$ if $a = b$ (match), -1 if $a \neq b$ (mismatch)
$W_i = -i$

$$H = \left(\begin{array}{cccccccccc} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 3 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 2 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 4 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{array}\right)$$

$$T = \left(\begin{array}{cccccccccc} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\ C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \uparrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \uparrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow \end{array}\right)$$

Local Sequence Alignment

- ACACACTA vs. AGCACACA

$$H = \left(\begin{array}{cccccccccc} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 2 \\ G & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ C & 0 & 0 & 3 & 2 & 3 & 2 & 3 & 2 & 1 \\ A & 0 & 2 & 2 & 5 & 4 & 5 & 4 & 3 & 4 \\ C & 0 & 1 & 4 & 4 & 7 & 6 & 7 & 6 & 5 \\ A & 0 & 3 & 3 & 6 & 6 & 9 & 8 & 7 & 8 \\ C & 0 & 2 & 4 & 5 & 8 & 8 & 11 & 10 & 9 \\ A & 0 & 4 & 3 & 6 & 7 & 10 & 10 & 10 & 12 \end{array}\right) \qquad T = \left(\begin{array}{cccccccccc} & - & A & C & A & C & A & C & T & A \\ - & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A & 0 & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ G & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow & \uparrow \\ C & 0 & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \uparrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow & \nwarrow \\ C & 0 & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ A & 0 & \uparrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \uparrow & \nwarrow & \nwarrow \end{array}\right)$$

Backtracking: (8,8), (7,7), (7,6), (6,5), (5,4), (4,3), (3,2), (2,1), (1,1), and (0,0),

```
A-CACACTA
AGCACAC-A
```

Local Sequence Alignment

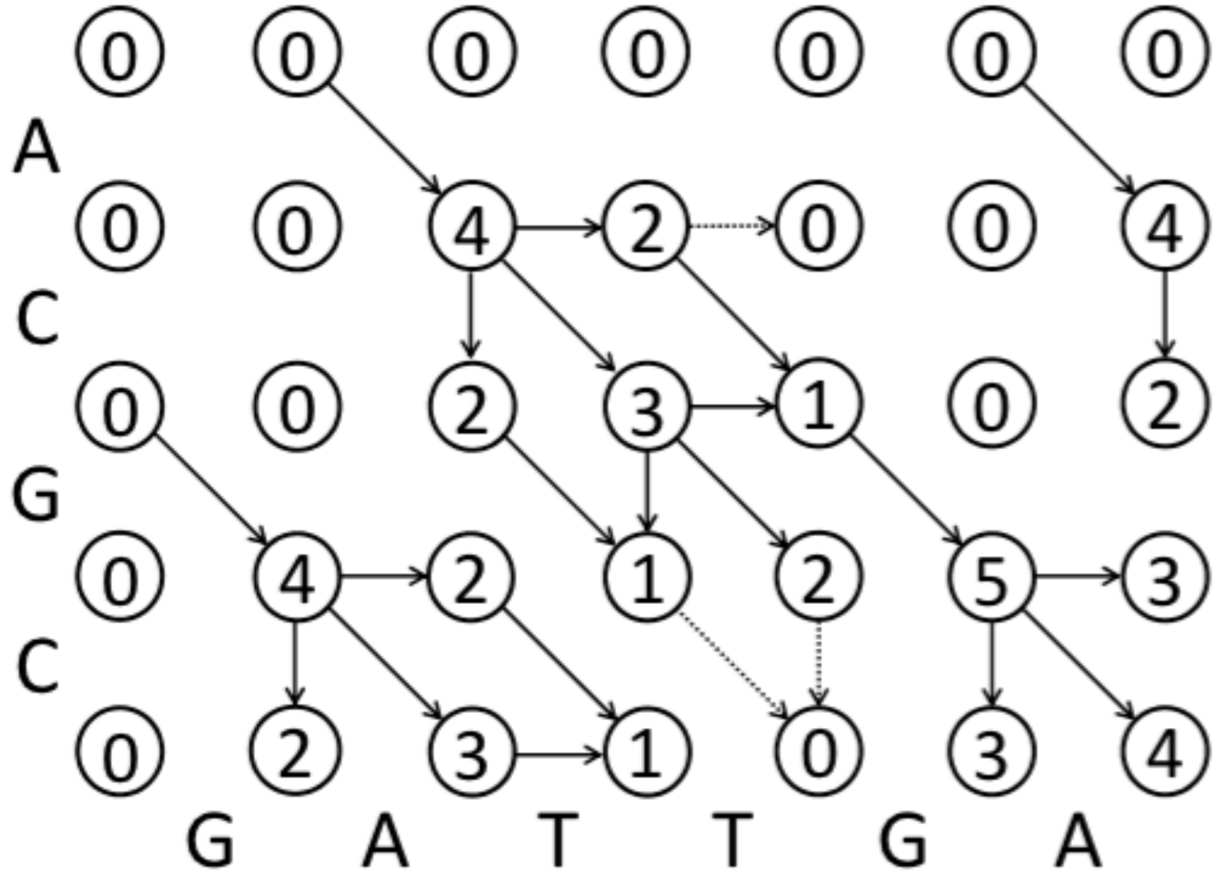
- Example

Two modifications to Needleman-Wunsch:

1) Allow a node to start at 0.

2) Record the highest-scoring node, and trace back from there.

Why does this algorithm yield an optimal local alignment?



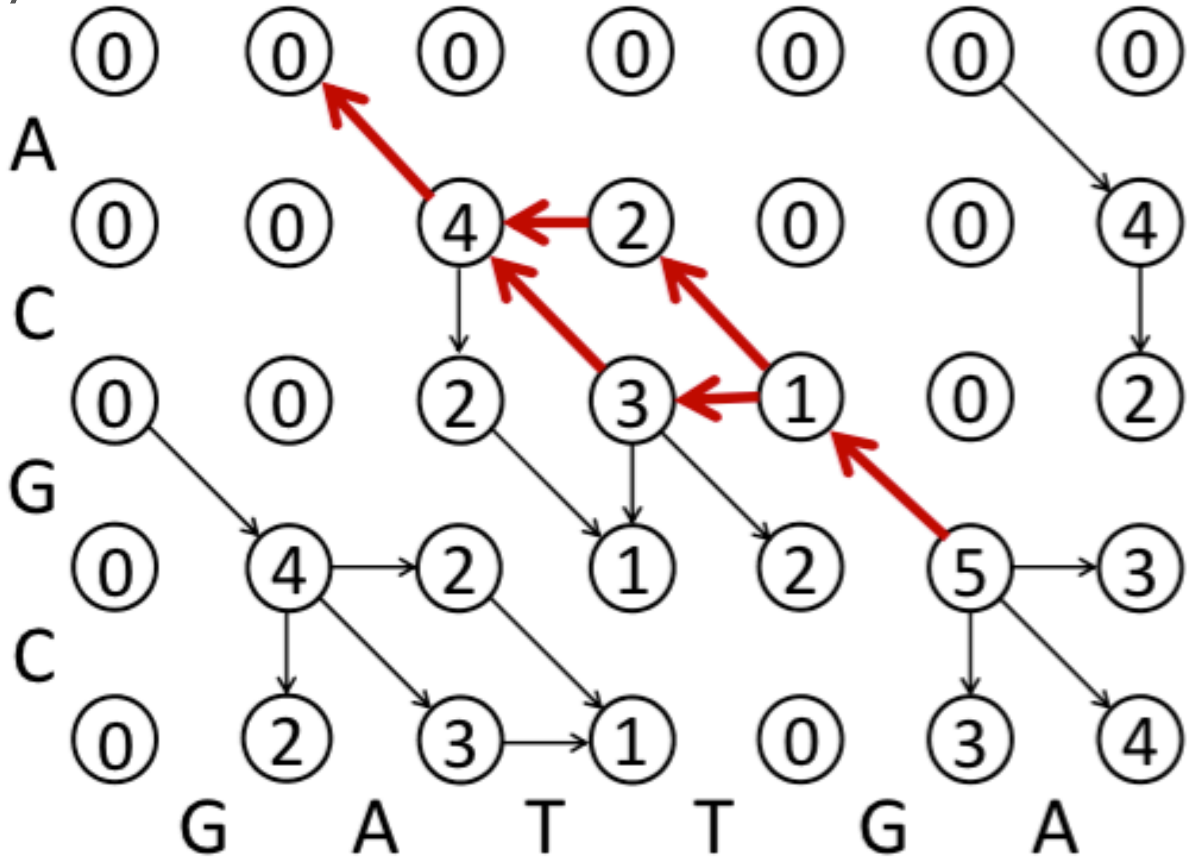
Scores: Match +4 Mismatch -1 Gap -2

Local Sequence Alignment

- Example (cont.)

Optimal local alignments, or *subalignments*:

```
AC−G       A−CG
      and
ATTG       ATTG
```



Questions:

Can one find other *locally optimal* subalignments?

How can they be defined?

Scores: Match +4 Mismatch -1 Gap -2

Local Sequence Alignment

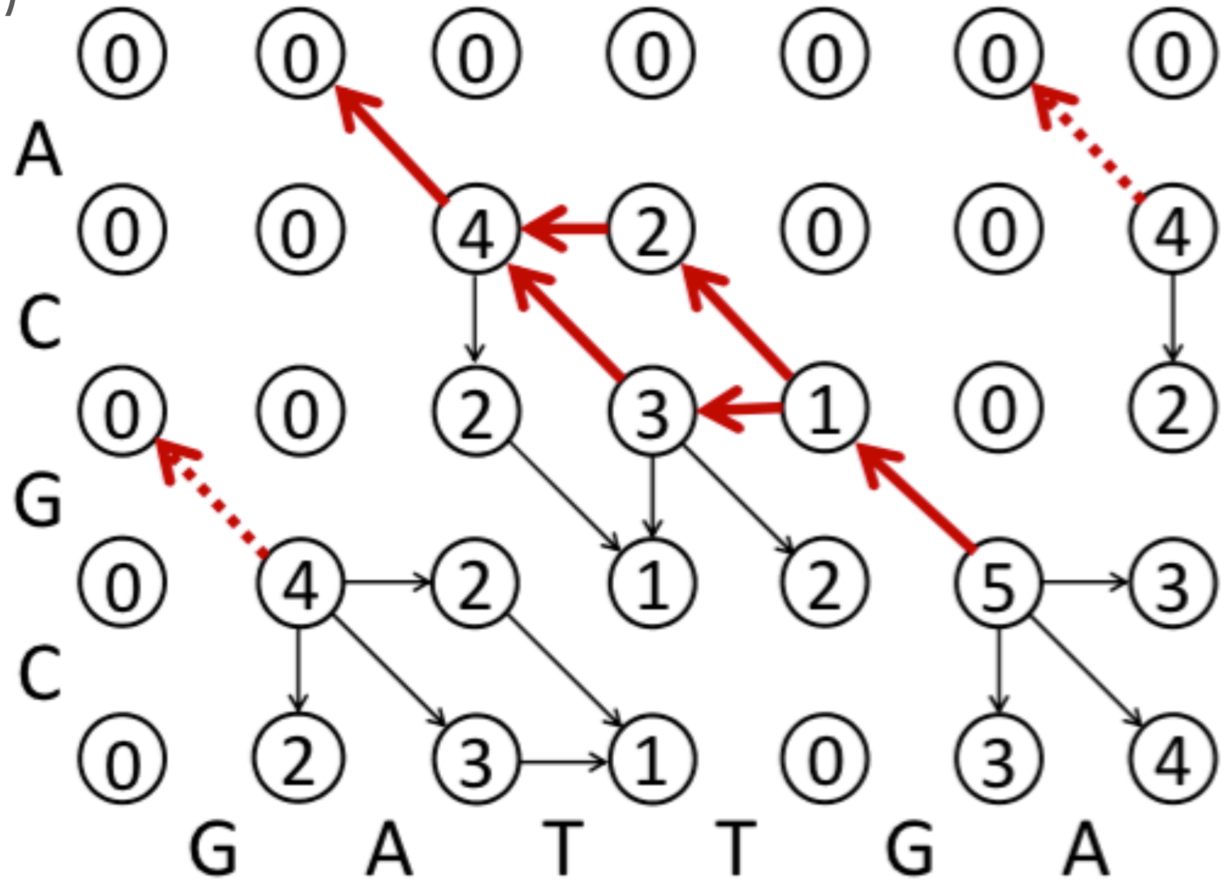
- Example (cont.)

Optimal subalignments:

AC–G and A–CG
ATTG   ATTG

Additional, locally optimal subalignments:

G and A
G   A



Scores: Match +4 Mismatch -1 Gap -2

Multiple Sequences Alignment

- Biologically similar proteins may NOT exhibit a strong sequence similarity
  - Pairwise alignment can fail to identify biological related sequences
- Simultaneous comparison of many sequences often allows to find similarities that are invisible in pairwise comparison

Multiple Sequences Alignment

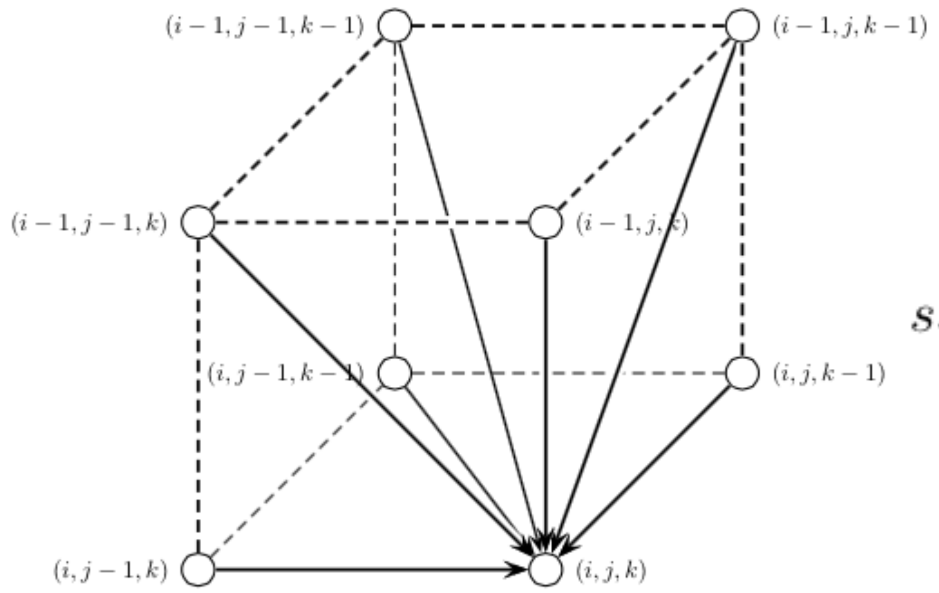
- Multiple alignment of strings $v_1, \ldots, v_k$

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-GAC
  |  || |  ||  | | | |||    || |  | |  | ||||   |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--T-CAGAT--C
 ||||| |   X|||| |          || XXX|||  | ||| |  |
-ATTGC-G--ATTCGTAT------GGGACA-TGGATGCATGCAG-TGAC
```

  - No column in a multiple alignment contains only spaces
  - This a generalization of the pairwise alignment, $k > 2$ sequences
  - Multiple alignment score:
    - Sum of the columns, with optimal alignment (max the score)
  - Consensus of an alignment:
    - A string of the most common characters in each column

Multiple Sequences Alignment

- Suppose that we have 3 sequences: u, v, and w
  - We want to find the "best" alignment of all three
  - Every multiple alignment corresponds to a path in 3D Manhattan like edit graph !!!
  - To get to vertex (i, j, k) in a 3D edit graph:



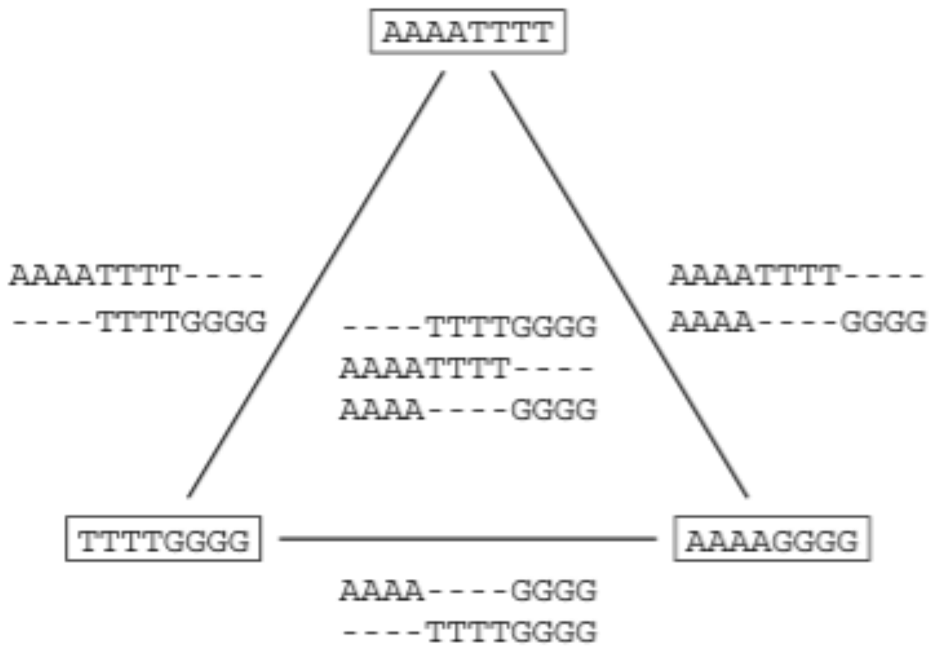
$$s_{i,j,k} = \max \begin{cases} s_{i-1,j,k} & +\delta(v_i, -, -) \\ s_{i,j-1,k} & +\delta(-, w_j, -) \\ s_{i,j,k-1} & +\delta(-, -, u_k) \\ s_{i-1,j-1,k} & +\delta(v_i, w_j, -) \\ s_{i-1,j,k-1} & +\delta(v_i, -, u_k) \\ s_{i,j-1,k-1} & +\delta(-, w_j, u_k) \\ s_{i-1,j-1,k-1} & +\delta(v_i, w_j, u_k)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

Multiple Sequences Alignment

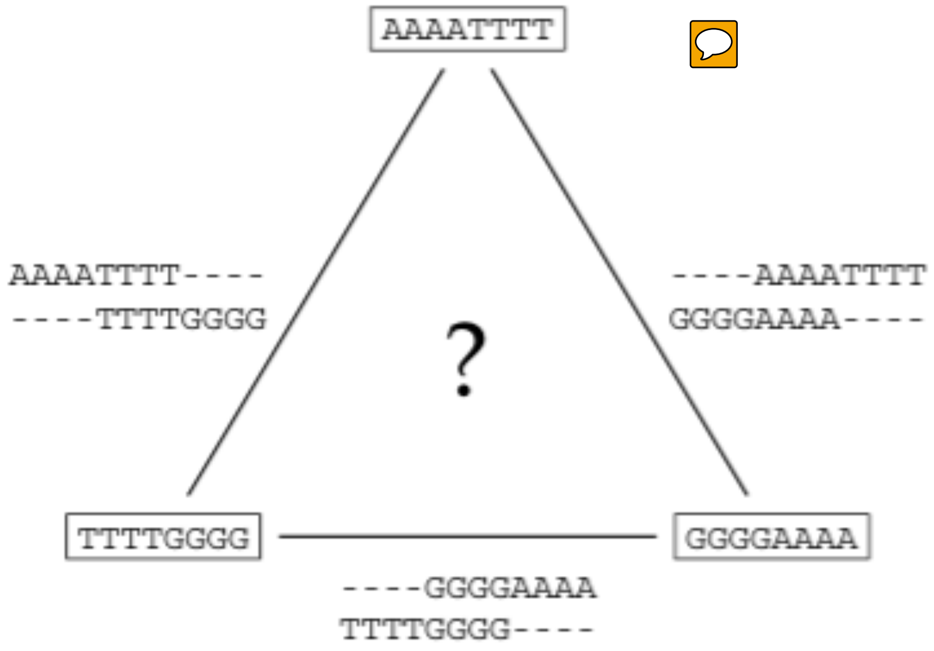
- Some improvements of the algorithm, and many heuristics have been proposed
  - Compute all optimal pairwise alignment between very pair of strings
  - Combine them together in such a way that pairwise alignments induced by multiple alignment are close to the optimal ones
- Not always possible to combine…

Multiple Sequences Alignment

- Compatible vs. Incompatible pairwise alignments



(a) Compatible pairwise alignments



(b) Incompatible pairwise alignments

Multiple Sequences Alignment

- Greedy progressive multiple alignment
  - Iteratively adds one string to the growing multiple alignment
  - Select a pair of strings with greatest similarity and merge them into a new string
    - "*once a gap, always a gap*" principle
  - The choice of the closest strings at the beginning provide the most reliable information about a real alignment – seed
  - If we start with bad seed, the error will propagate all the way to the whole multiple alignment

Multiple Sequences Alignment

- k-dimensional scoring matrices are NOT very practical
- The choice of scoring function can affect the quality of the resulting alignment
  - No single scoring approach is perfect in all circumstances
- We want to assignment higher scores to the columns with a low variation in letters
  - High scores correspond to highly conserved sequences

Multiple Sequences Alignment

- Entropy approach:
  - The score is the sum of the entropies of the columns, defined as

$$\sum_{x \in \mathcal{A}'} p_x \log p_x \qquad \text{where } p_x \text{ is the frequency of letter } x \in \mathcal{A}'$$

  - The more conserved the column, the larger the entropy score
    - A column that has each nucleotides present k/4 times will have a *score = 4 x (1/4 log (1/4)) = -2*
    - A completely conserved column has a *score = 0*
  - It can be difficult to design efficient algorithms that optimize this scoring function