AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

22BIO211: Intelligence of Biological Systems - 2

# SEQUENCE ALIGNMENT USING MANHATTAN LIKE GRAPHS

Dr. Manjusha Nair M
Amrita School of Computing, Amritapuri

Email : manjushanair@am.amrita.edu
Contact No: 9447745519

# Manhattan to DAG

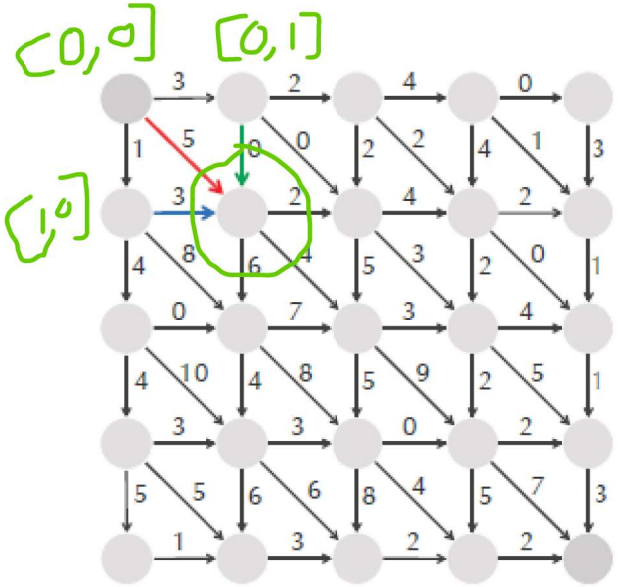
- To adapt MANHATTAN TOURIST problem for alignment graphs with diagonal edges,
  - *An LCS can be represented by a longest path in the alignment graph.*
  - *Various alignment applications are much more complex than the Longest Common Subsequence Problem*
    - and require building a DAG with appropriately chosen edge weights in order to model the specifics of a biological problem.

# Dynamic Programming in DAG

- Given a node ***b*** in a DAG, let $\mathbf{s_b}$ denote the length of a longest path from the source to ***b***.
- We call node ***a*** a predecessor of node ***b*** if there is an edge connecting ***a*** to ***b*** in the DAG;
- The **indegree** of a node is equal to the number of its predecessors.
- The score $\mathbf{s_b}$ of node ***b*** with indegree ***k*** is computed as a maximum of k terms:

$$s_b = \max_{\text{all predecessors } a \text{ of node } b} \{s_a + \text{weight of edge from } a \text{ to } b\}.$$

# Dynamic Programming in DAG




- Node (1, 1) has three predecessors
  - *Can arrive at (1, 1) by traveling right from (1, 0), down from (0, 1), or diagonally from (0, 0)*

# Dynamic Programming in DAG

- Assuming that we have already computed $s_{0,0}$, $s_{0,1}$, and $s_{1,0}$, we can therefore compute $s_{1,1}$ as the maximum of three values:

$$s_{1,1} = \max \begin{cases} s_{0,1} + \text{weight of edge } \downarrow \text{ connecting } (0,1) \text{ to } (1,1) = 3+0=3 \\ s_{1,0} + \text{ weight of edge } \rightarrow \text{ connecting } (1,0) \text{ to } (1,1) = 1+3=4 \\ s_{0,0} + \text{ weight of edge } \searrow \text{ connecting } (0,0) \text{ to } (1,1) = 0+5=5 \end{cases}$$

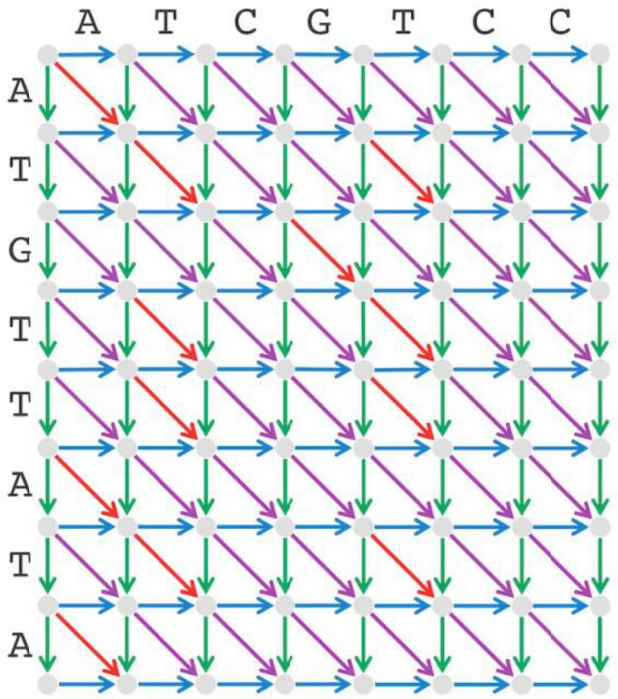
To compute scores for any node $(i,j)$ of this graph, we use the following recurrence:

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

# Dynamic Programming in DAG

- An analogous argument can be applied to the alignment graph to compute the length of an LCS between sequences *v* and *w*.
- Since all edges have weight *0* except for diagonal edges of weight *1* that represent matches (vi = wj), we obtain the following recurrence for computing the length of an LCS.

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



# Sequence alignment Example -scoring

- Find the alignment of two sequences

  GAATTCAGTTA

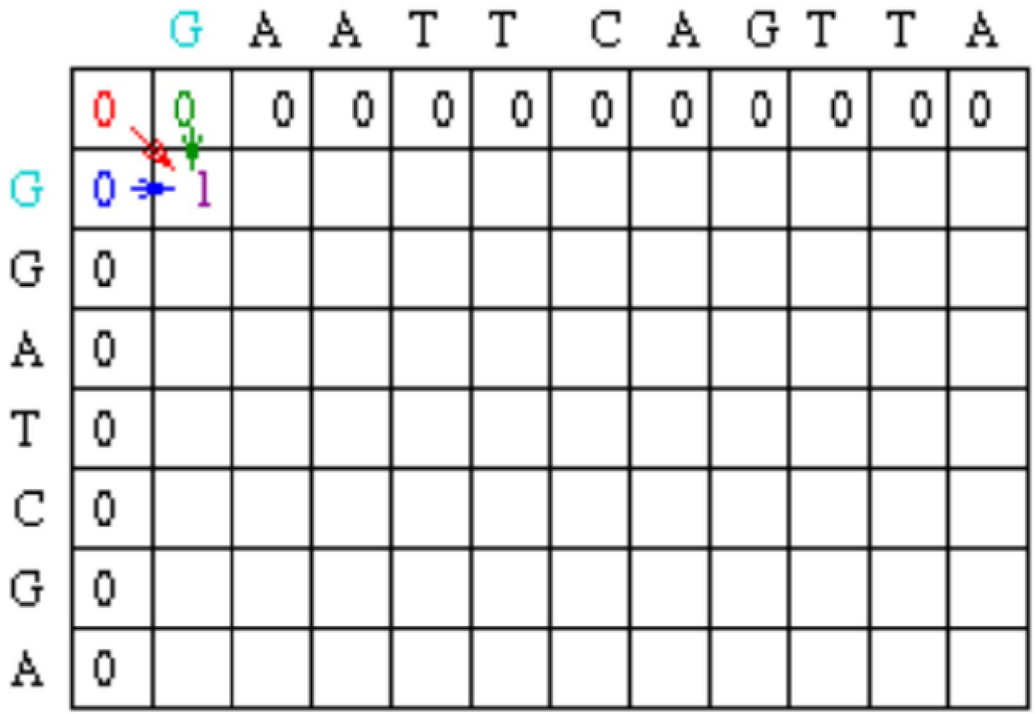
  GGATCGA

- Given
  - *Score(match) = 1*
  - *Score(mismatch) = 0*
  - *Score(gap) = 0*
  - *Score(0,0) = 0*

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

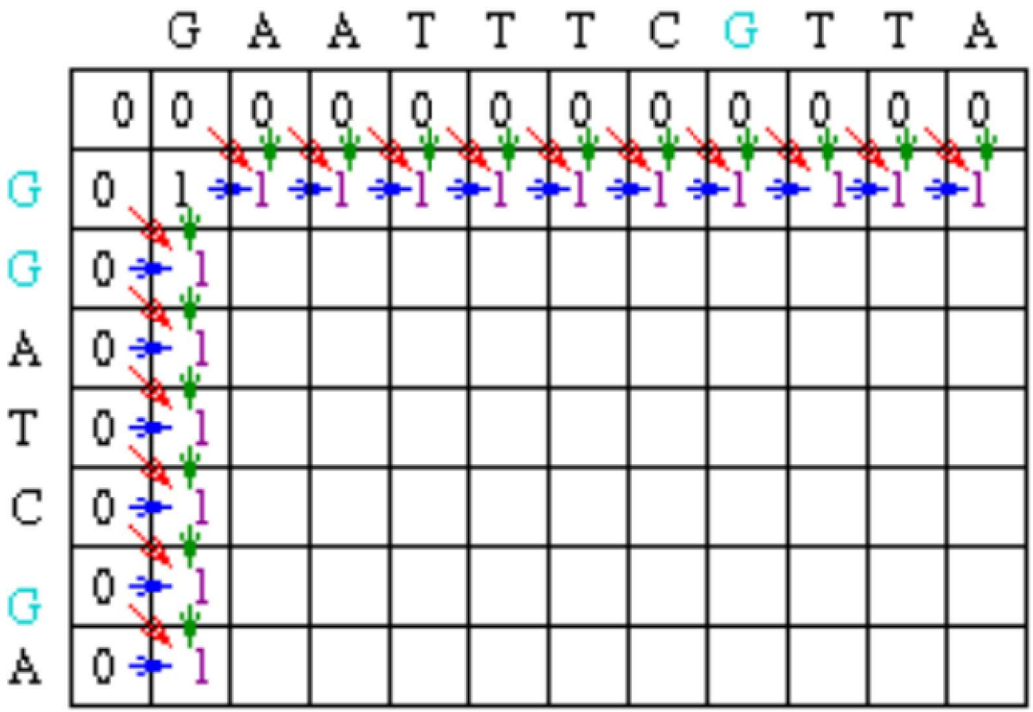
# Sequence alignment Example -scoring

- Added match score of 1 between G and G

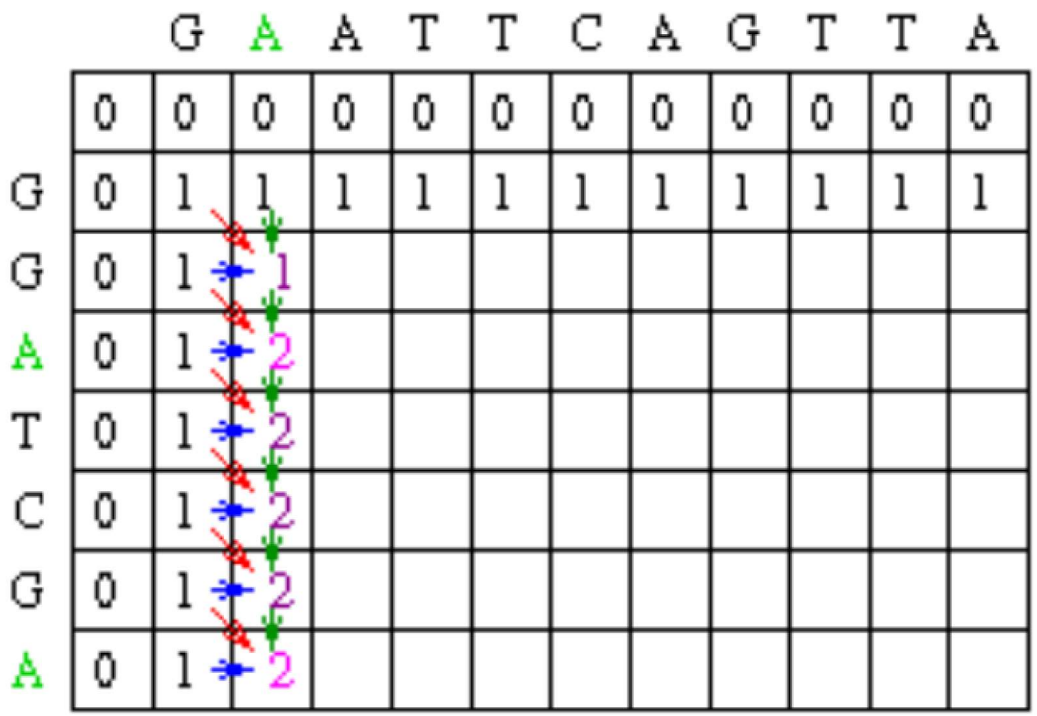


| | | G | A | A | T | T | C | A | G | T | T | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 1 | | | | | | | | | | |
| G | 0 | | | | | | | | | | | |
| A | 0 | | | | | | | | | | | |
| T | 0 | | | | | | | | | | | |
| C | 0 | | | | | | | | | | | |
| G | 0 | | | | | | | | | | | |
| A | 0 | | | | | | | | | | | |

# Sequence alignment Example -scoring



# Sequence alignment Example -scoring



# Sequence alignment Example - scoring

Final Result

| | | G | A | A | T | T | C | A | G | T | T | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| G | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| A | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 3 |
| T | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| C | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |
| G | 0 | 1 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 5 | 5 |
| A | 0 | 1 | 2 | 3 | 3 | 3 | 3 | 4 | 5 | 5 | 5 | **6** |

# Topological orderings

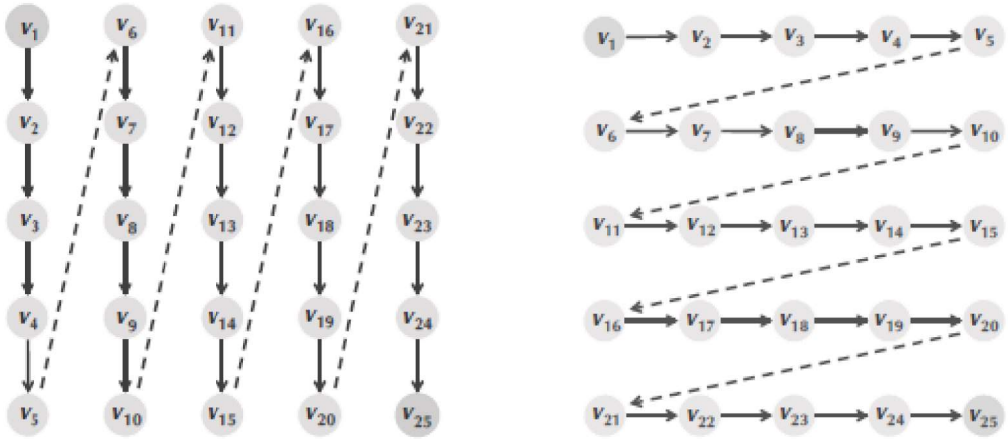
- To use dynamic programming to find the length of a longest path in a DAG, we must decide on the order in which to visit nodes when computing the values $s_b$ according to the recurrence.

$$s_b = \max_{\text{all predecessors } a \text{ of node } b} \{s_a + \text{weight of edge from } a \text{ to } b\}.$$

- This ordering of nodes is important, since by the time we reach node $b$, the values $s_a$ for all its predecessors must have already been computed.
- We need to ensure that we would never consider a node before visiting all of its predecessors.
- Formally, an ordering of nodes (a1, . . . , ak) in a DAG is called a **topological ordering** if every edge (ai, aj) of the DAG connects a node with a smaller index to a node with a larger index, i.e., i < j.

# Topological orderings

- MANHATTANTOURIST is able to find a longest path in a rectangular grid because
  - *pseudocode implicitly orders nodes according to the "column-by-column" topological ordering .*
- The "row-by-row" ordering gives another topological ordering of a rectangular grid.
- It can be proven that any DAG has a topological ordering, and that this topological ordering can be constructed in time proportional to the number of edges in the graph



# Topological orderings

- Once we have a topological ordering, we can compute the length of the longest path from source to sink
  - *by visiting the nodes of the DAG in the order dictated by the topological ordering*
- For simplicity, we assume that the source node is the only node with indegree 0 in Graph.

# Longest Path in the DAG - Scoring

- Algorithm

```
LongestPath(Graph, source, sink)
  for each node b in Graph
    s_b ← −∞
  s_source ← 0
  topologically order Graph
  for each node b in Graph (following the topological order)
    s_b ← max_{all predecessors a of node b} {s_a + weight of edge from a to b}
  return s_sink
```

# Longest Path in the DAG - Scoring

- Since every edge participates in only a single recurrence, the runtime of LONGESTPATH is proportional to the number of edges in the DAG.
- We can now efficiently compute the length of a longest path in an arbitrary DAG

# Summary

- Manhattan to DAG
- Dynamic Programming in DAG
- Topological Ordering
- Longest Path in the DAG