

Dynamic Programming: Sequence alignment

(based on
CS 498 SS by
Saurabh Sinha)

AN INTRODUCTION TO **BIOINFORMATICS**
ALGORITHMS, NEIL C. JONES AND PAVEL A. PEVZNER,
Chapters: 6.1, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9

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
- A normal growth gene switched on at the wrong time causes cancer !



Russell Doolittle

Cystic Fibrosis

- **Cystic fibrosis** (CF) is a chronic and frequently fatal genetic disease of the body's mucus glands. CF primarily affects the respiratory systems in children.
- If a high % of cystic fibrosis (CF) patients have a certain mutation in the gene and the normal patients don't, then that could be an indicator of a mutation that is related to CF
- A certain mutation was found in 70% of CF patients, convincing evidence that it is a predominant genetic diagnostics marker for CF



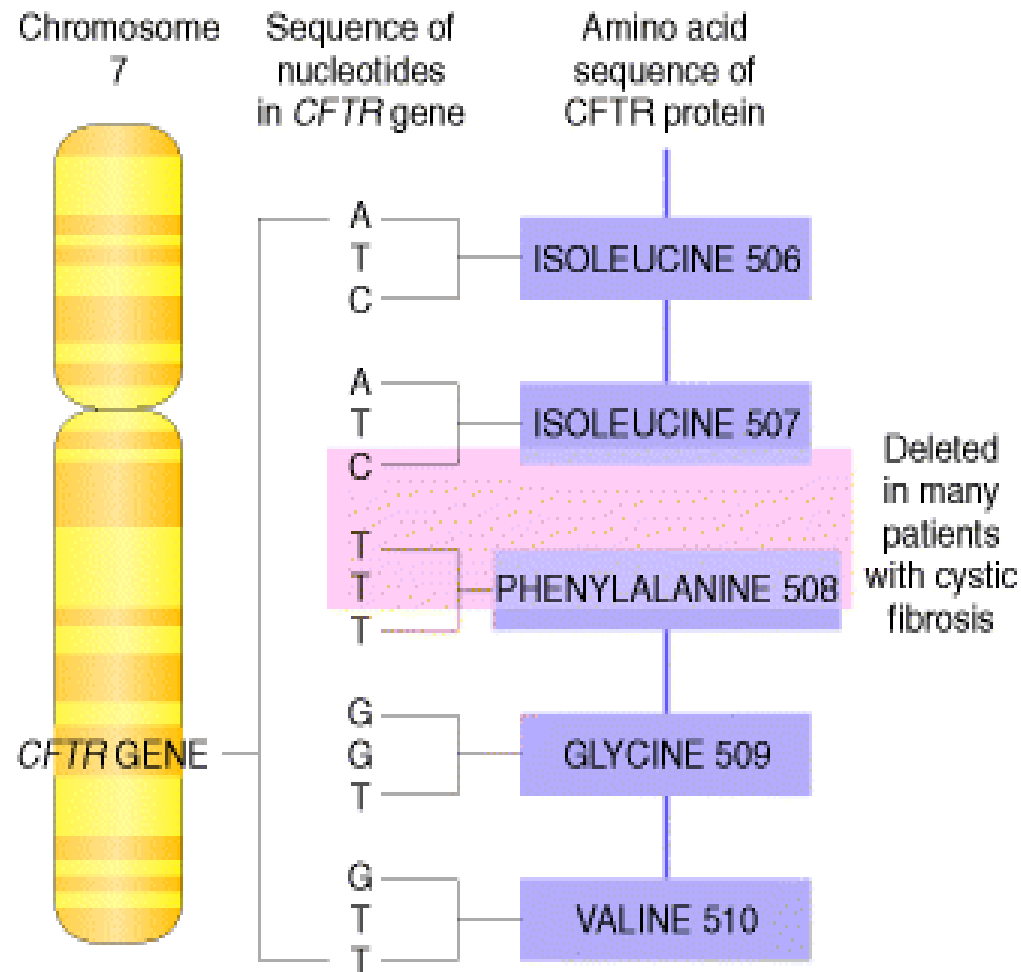
Cystic Fibrosis: Mutation Analysis

- If a high percentage of CF patients have a given mutation in the gene and the normal patients do not, then this could be an indicator of a mutation related to CF.
- A certain mutation was in fact found in 70% of CF patients, convincing evidence that it is a predominant genetic diagnostics marker for CF.

Cystic Fibrosis and the CFTR Protein

- **CFTR Protein:** A protein of 1480 amino acids that regulates a chloride ion channel.
- CFTR adjusts the “wateriness” of fluids secreted by the cell.
- Those with cystic fibrosis are missing a single amino acid in their CFTR protein (illustrated on following slide).

Cystic Fibrosis and CFTR Gene :



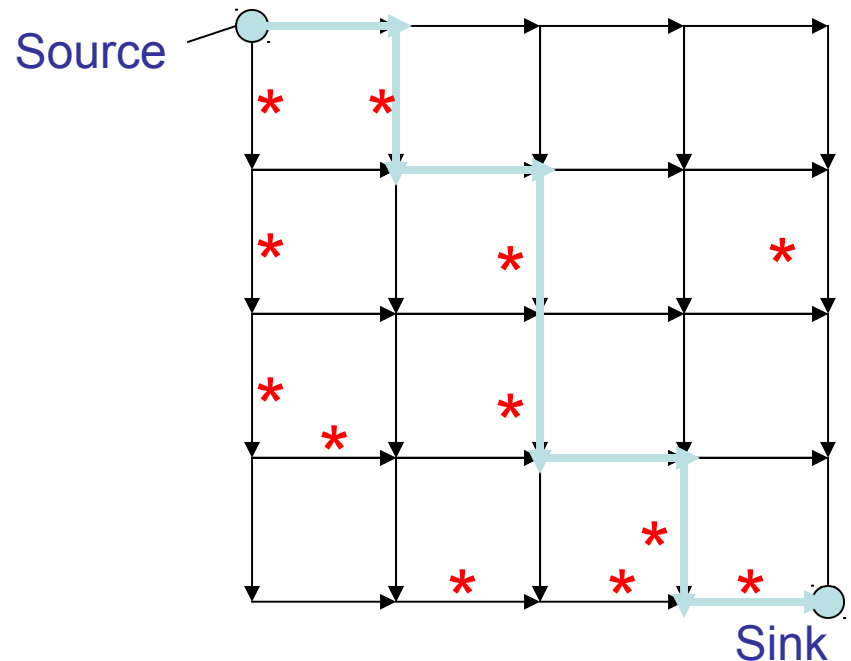
Bring in the Bioinformaticians

- 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

Motivating Dynamic Programming

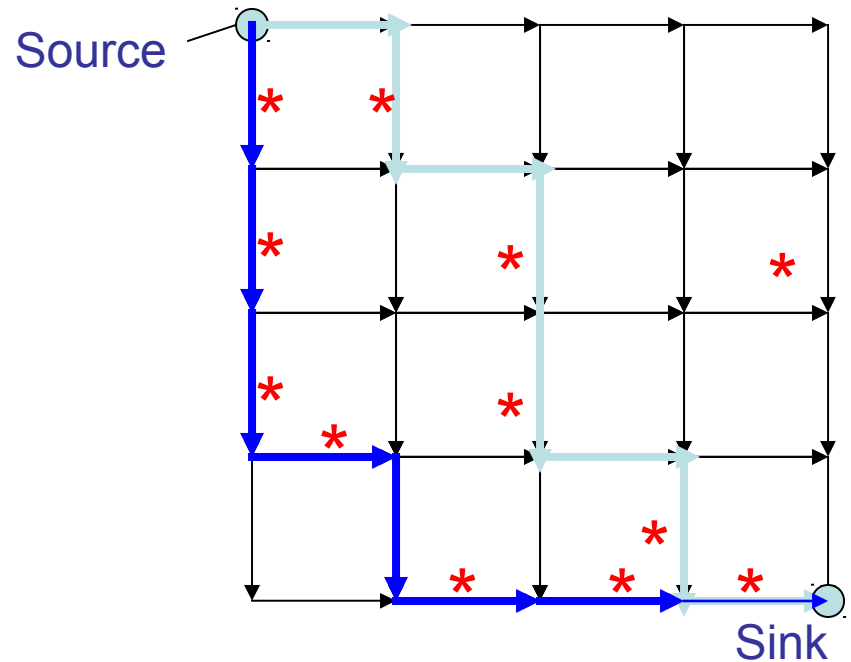
Dynamic programming example: Manhattan Tourist Problem

Imagine seeking a path (from source to sink) to travel (only eastward and southward) with the most number of attractions (*) in the Manhattan grid



Dynamic programming example: Manhattan Tourist Problem

Imagine seeking a path (from source to sink) to travel (only eastward and southward) with the most number of attractions (*) in the Manhattan grid



Manhattan Tourist Problem: Formulation

Goal: Find the longest path in a weighted grid.

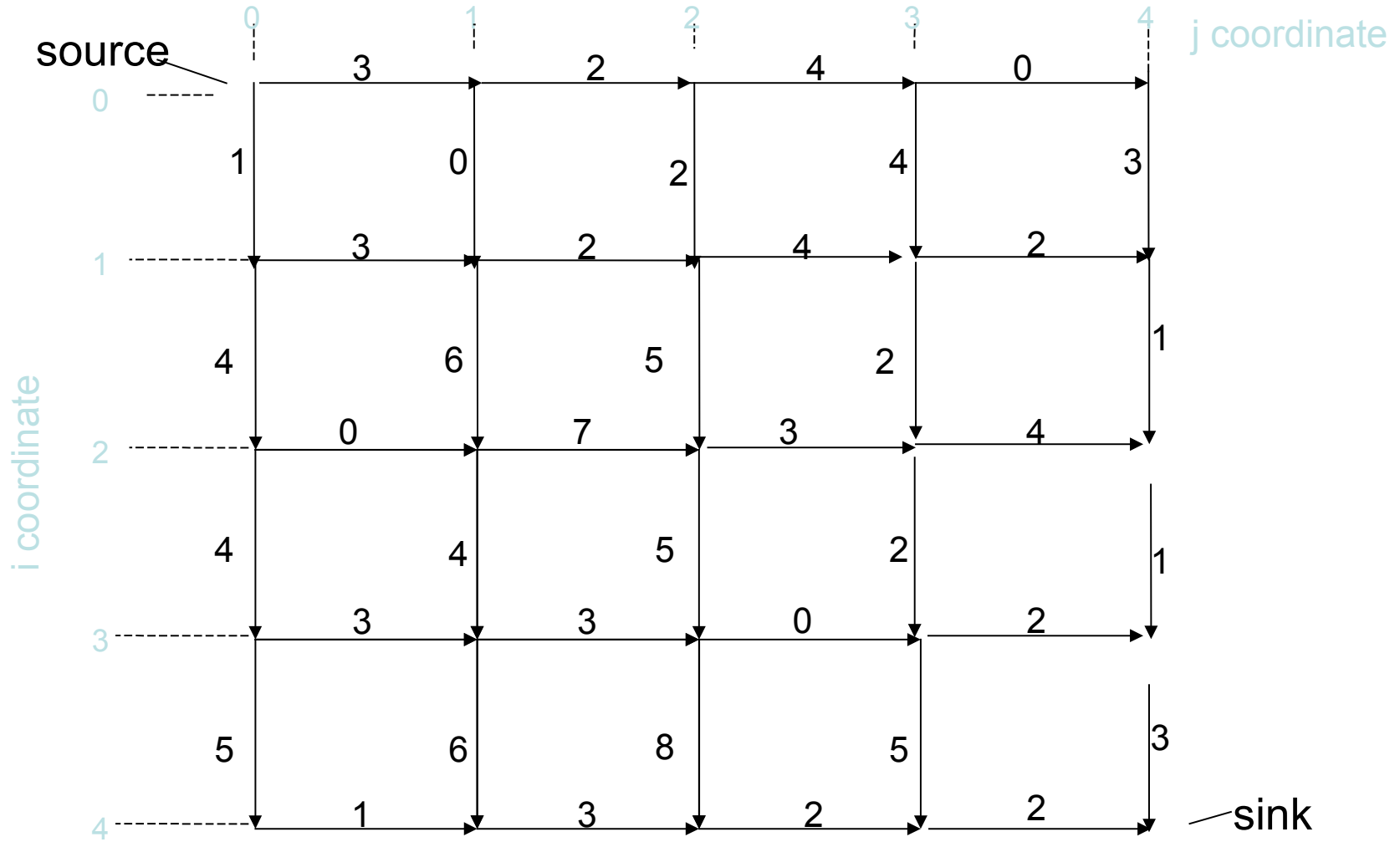
Input: A weighted grid \mathbf{G} with two distinct vertices, one labeled “*source*” and the other labeled “*sink*”

Output: A longest path in \mathbf{G} from “*source*” to “*sink*”

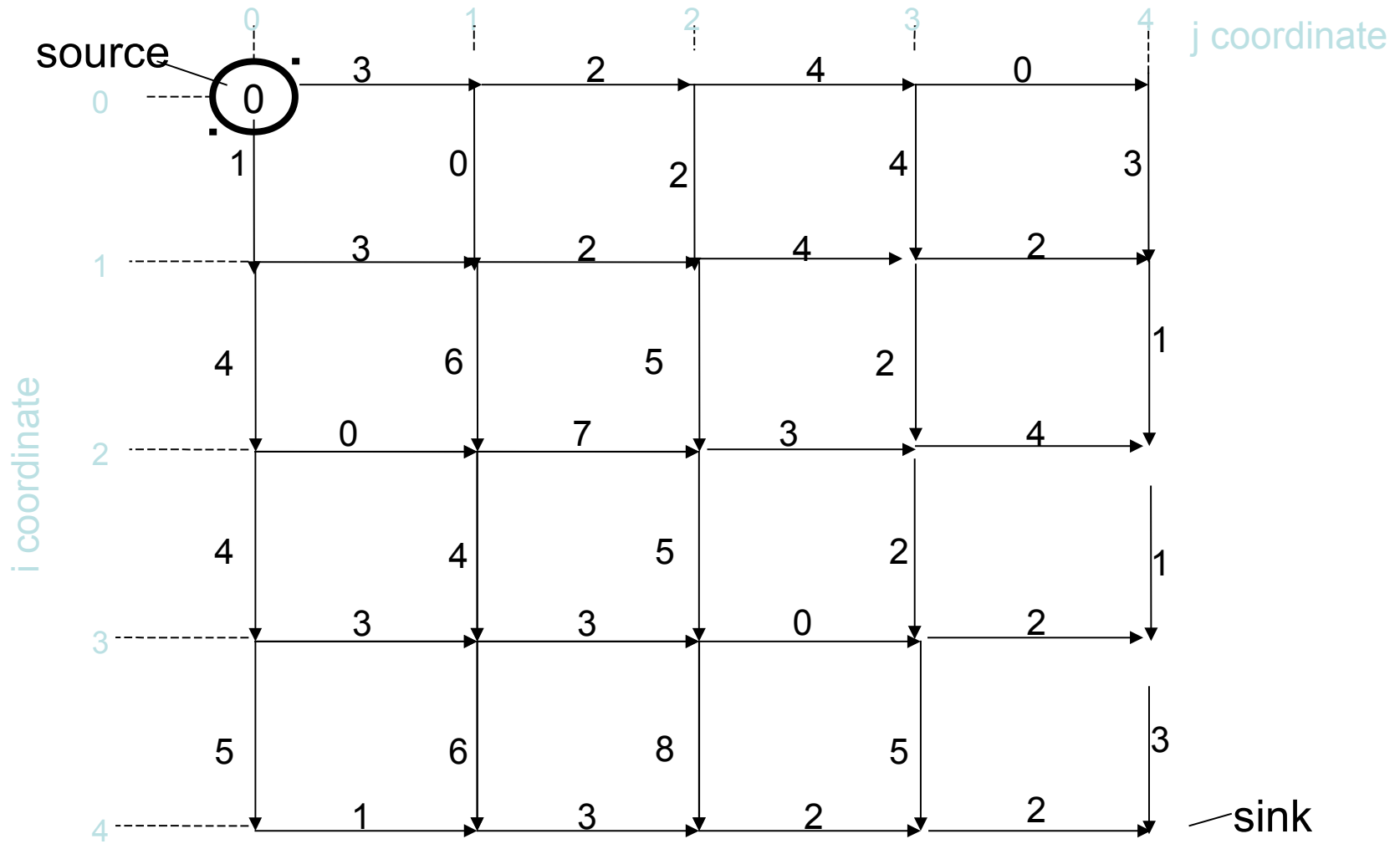
MTP Greedy Algorithm

- Our first try at solving the MTP will use a **greedy algorithm**.
- Main Idea: At each node (intersection), choose the edge (street) departing that node which has the greatest weight.

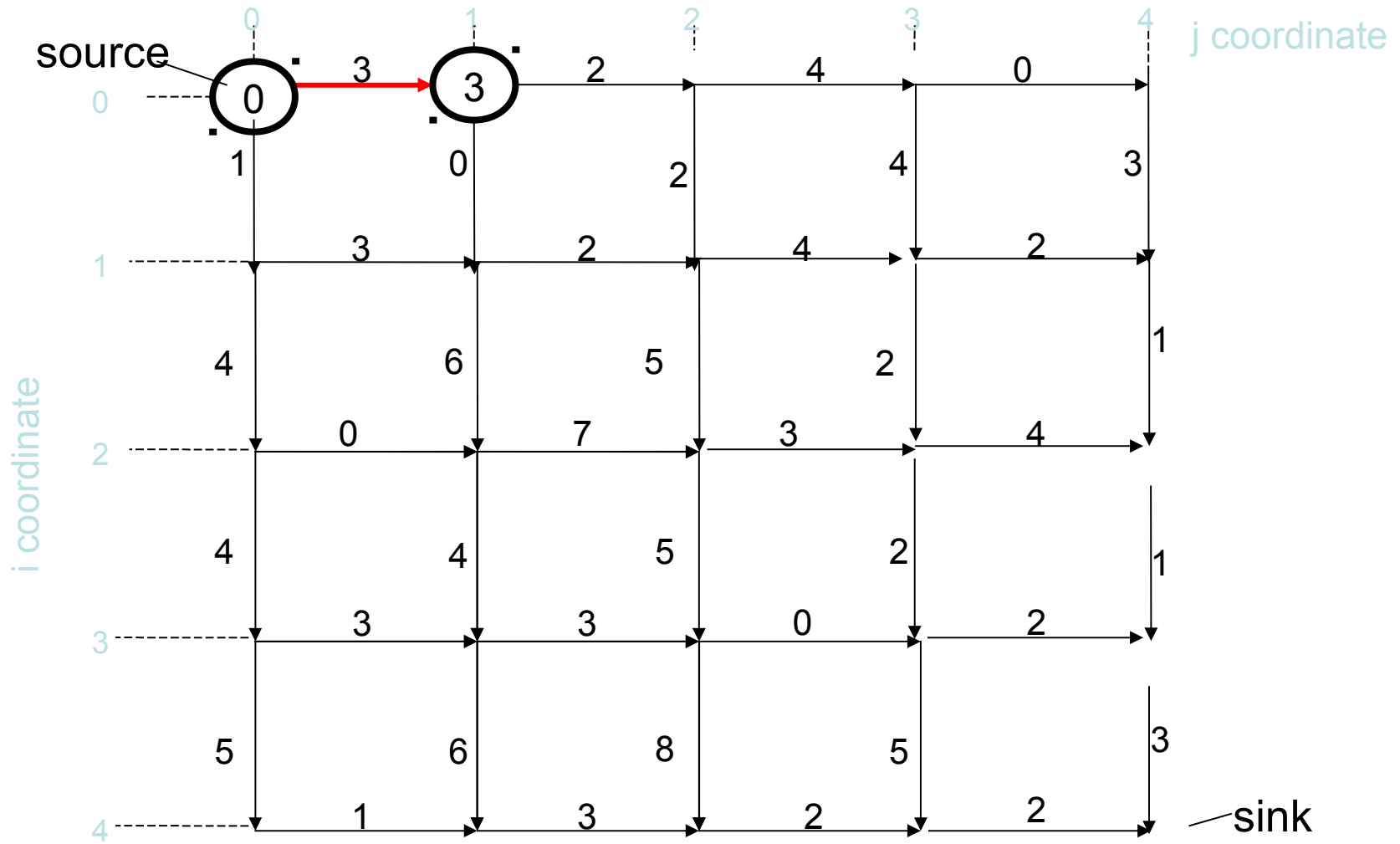
MTP Greedy Algorithm: Example



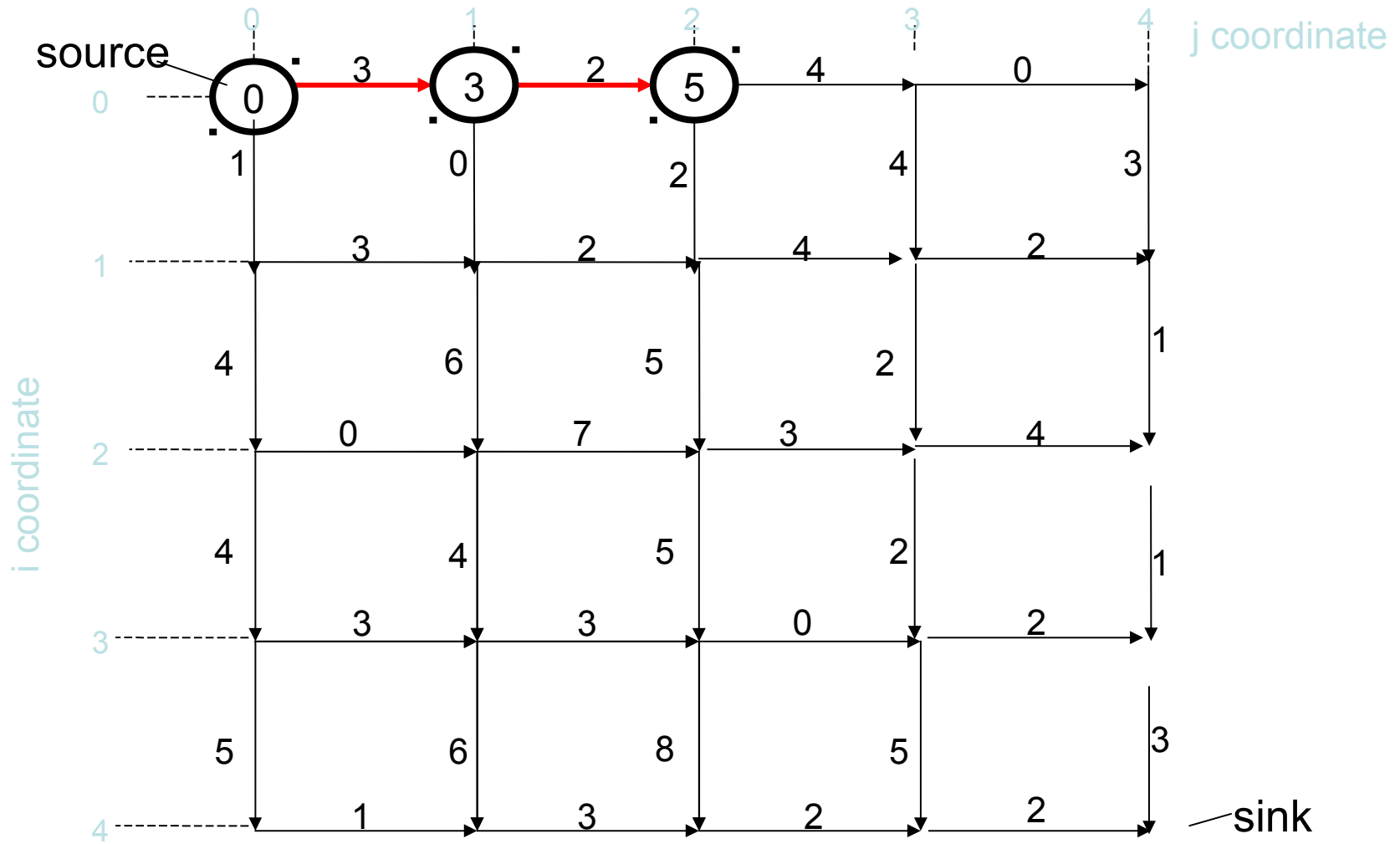
MTP Greedy Algorithm: Example



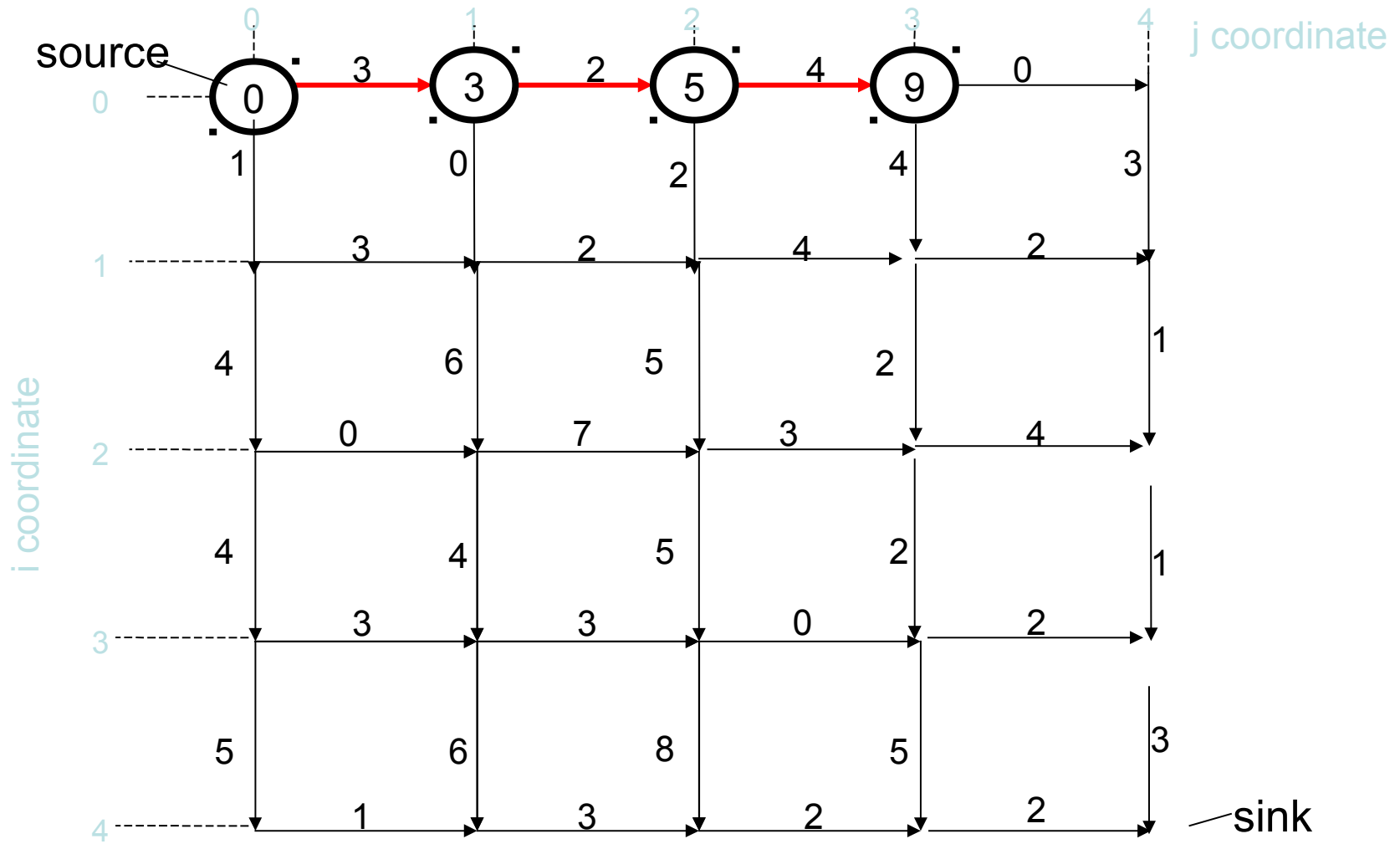
MTP Greedy Algorithm: Example



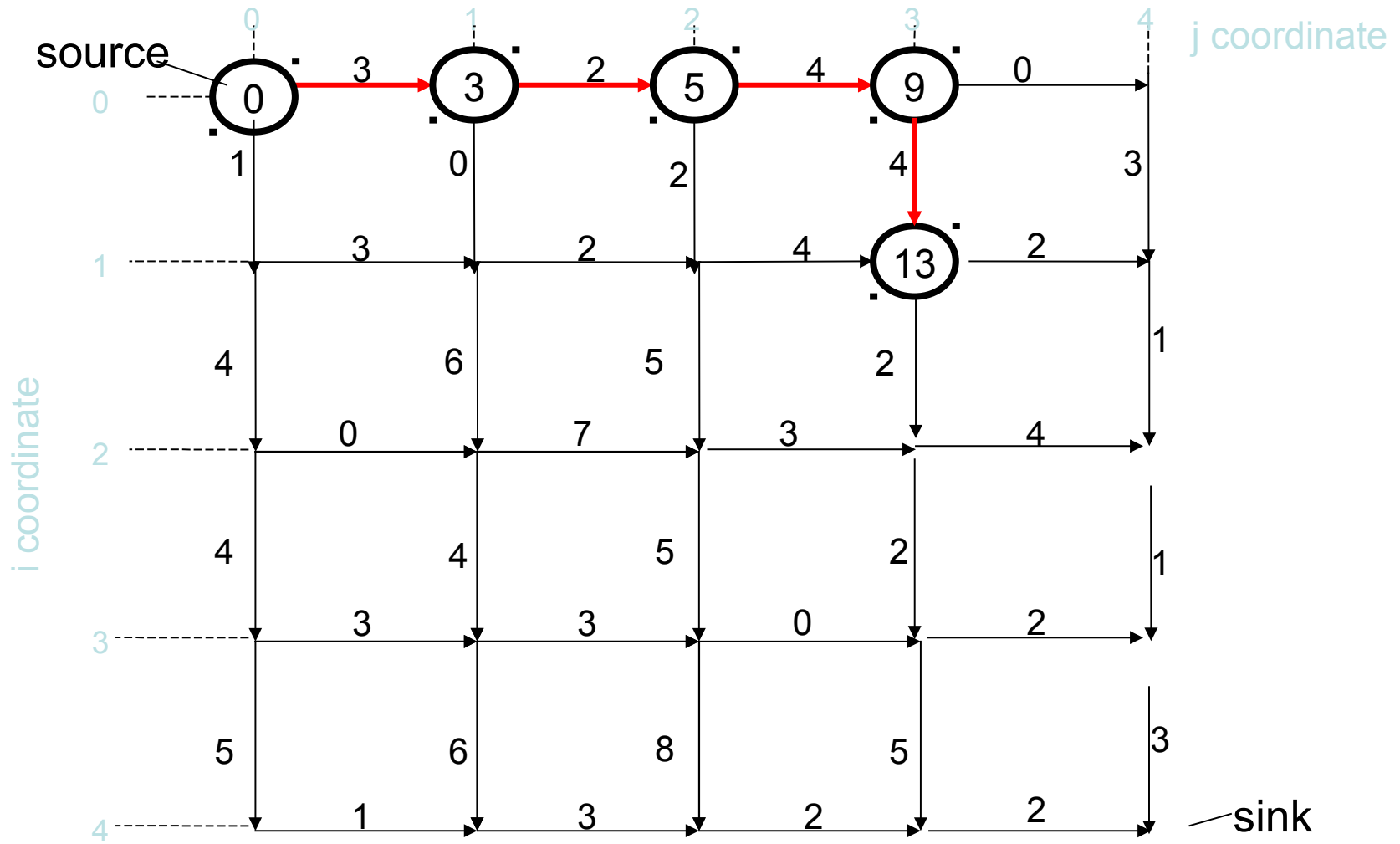
MTP Greedy Algorithm: Example



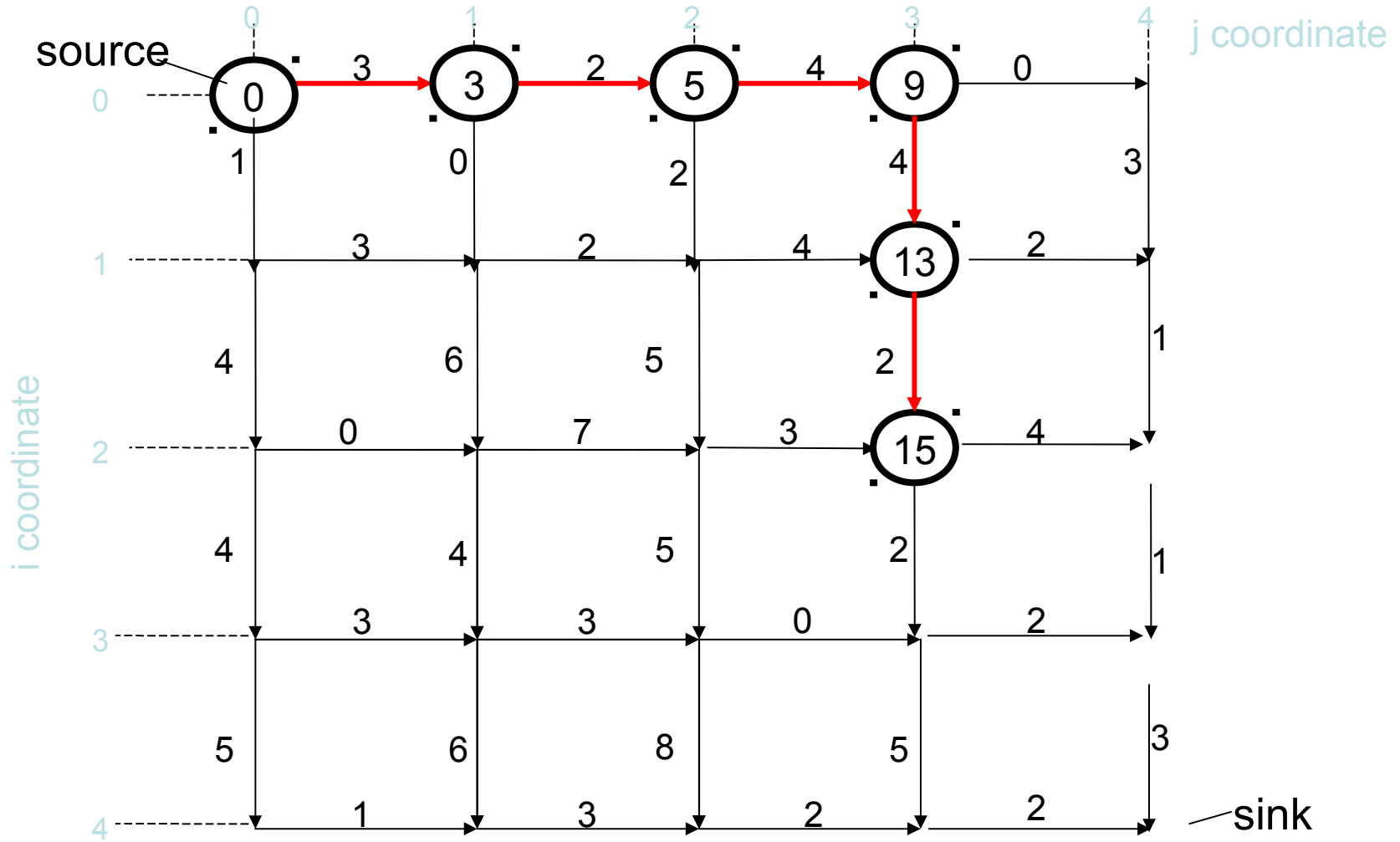
MTP Greedy Algorithm: Example



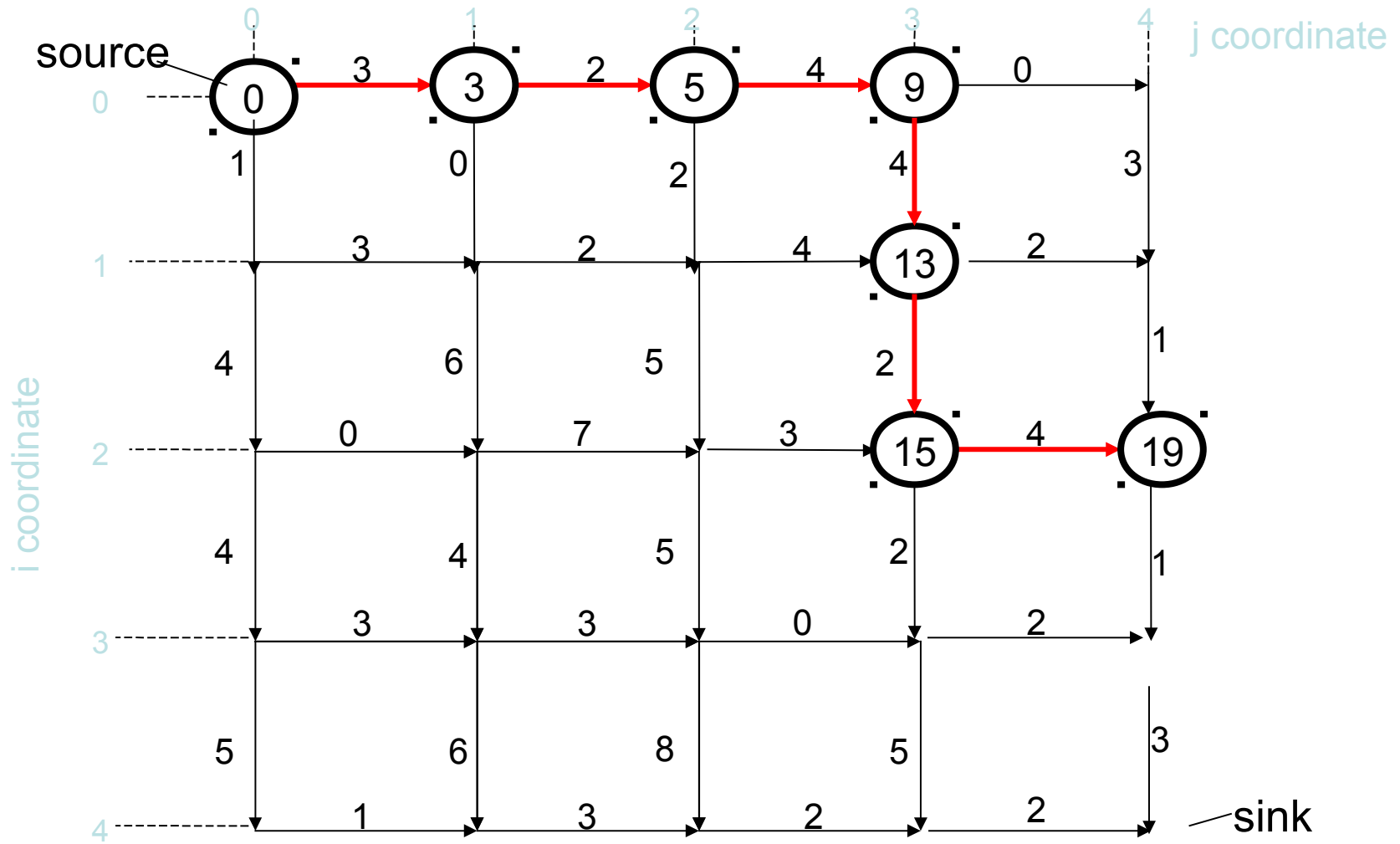
MTP Greedy Algorithm: Example



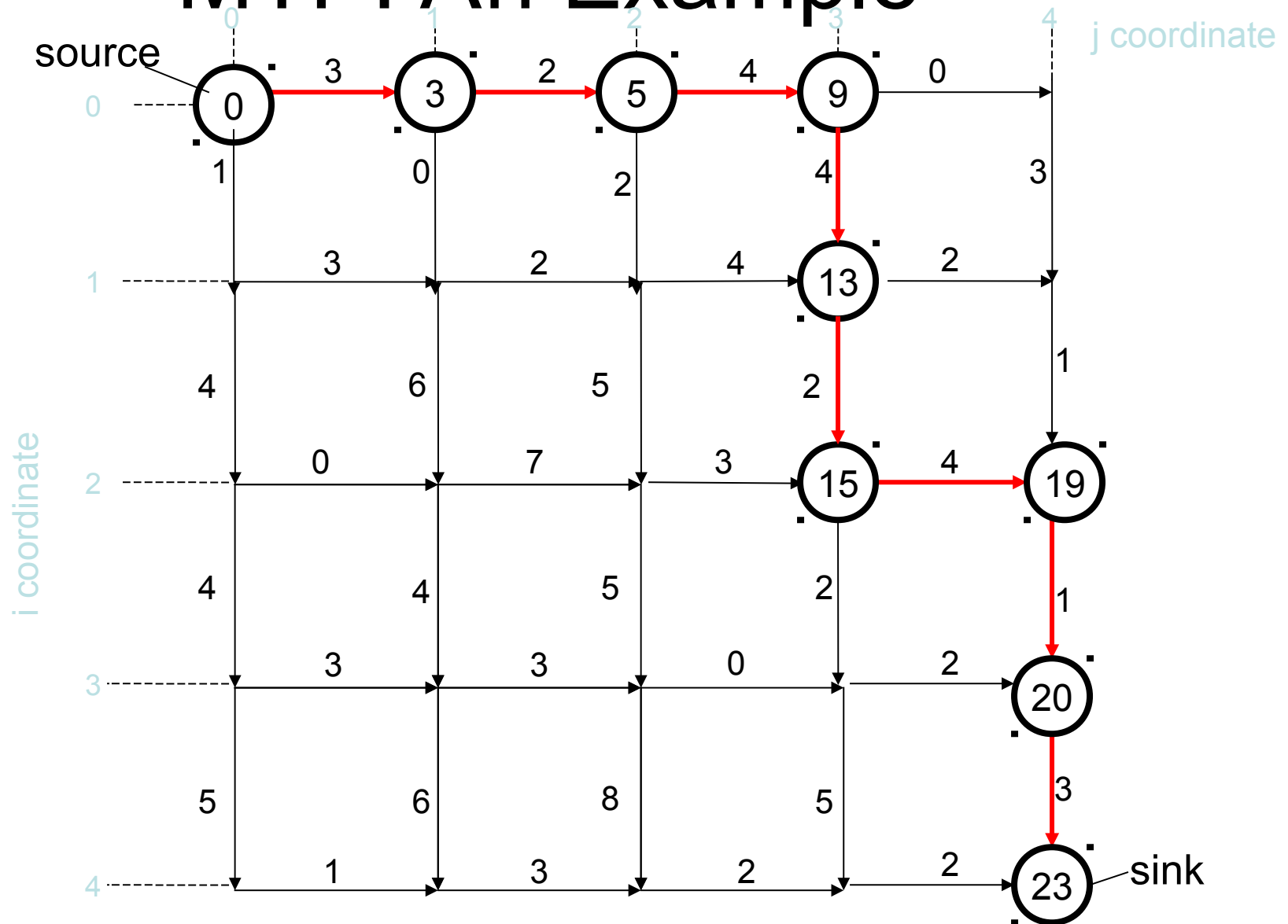
MTP Greedy Algorithm: Example



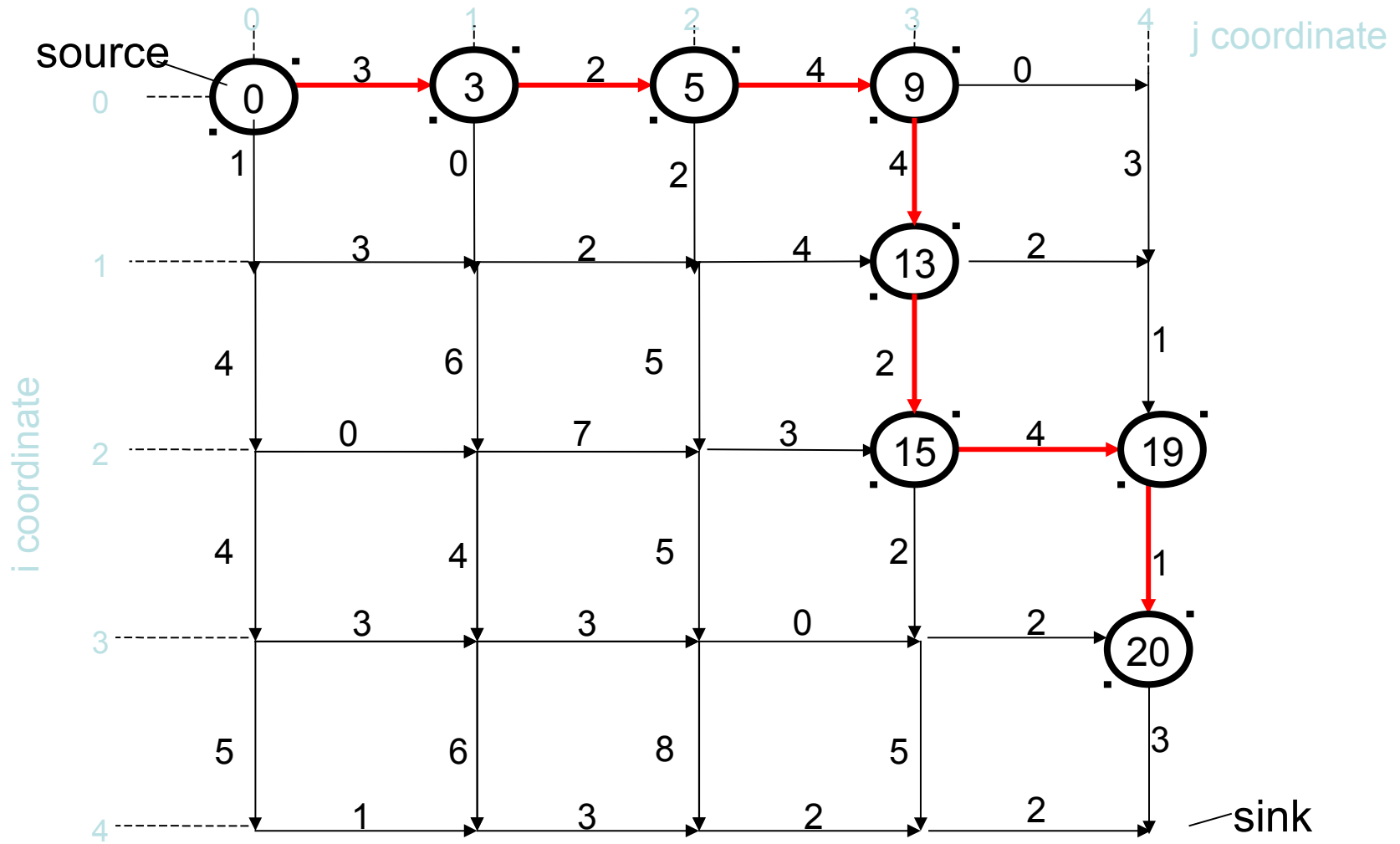
MTP Greedy Algorithm: Example



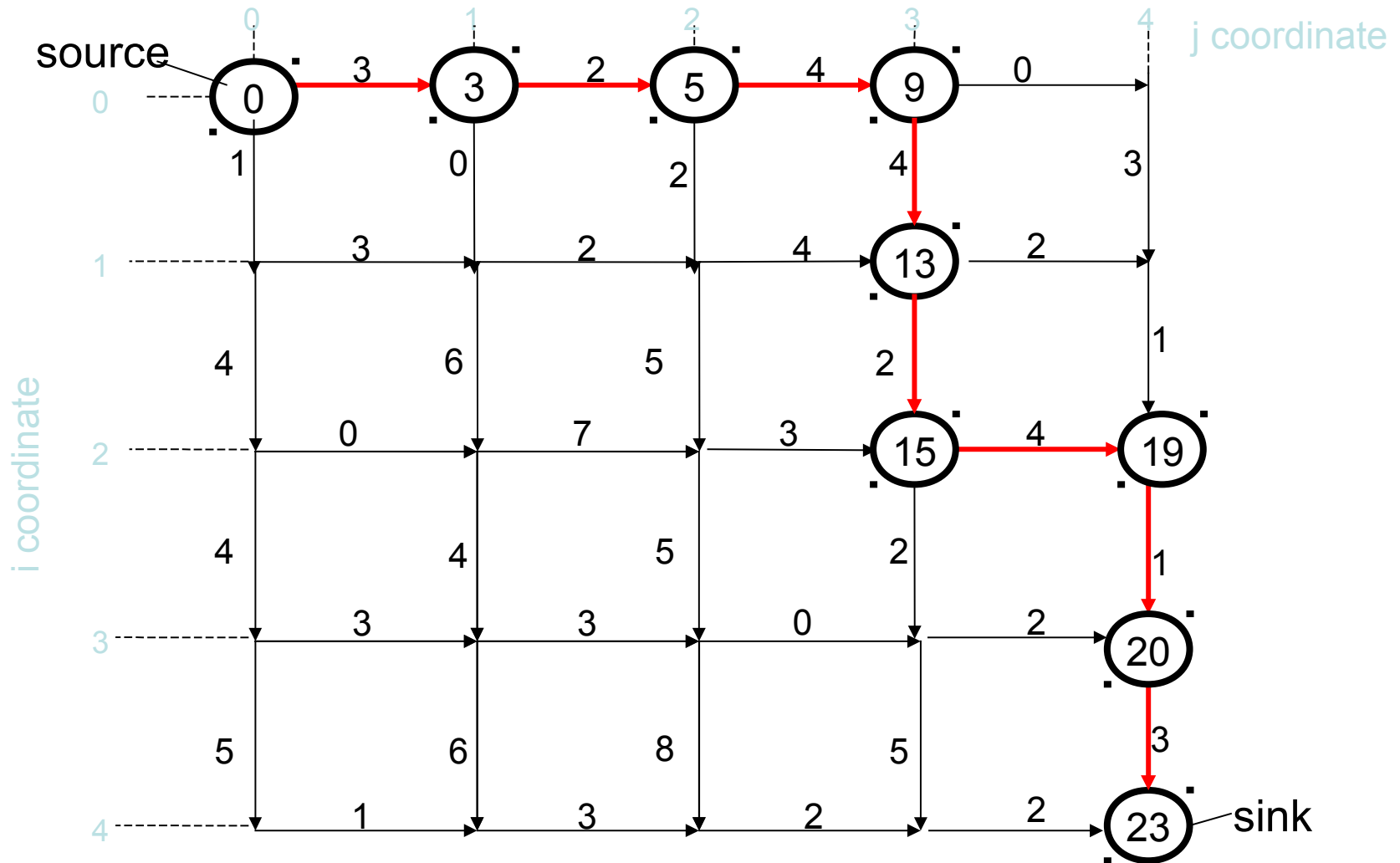
MTP: An Example



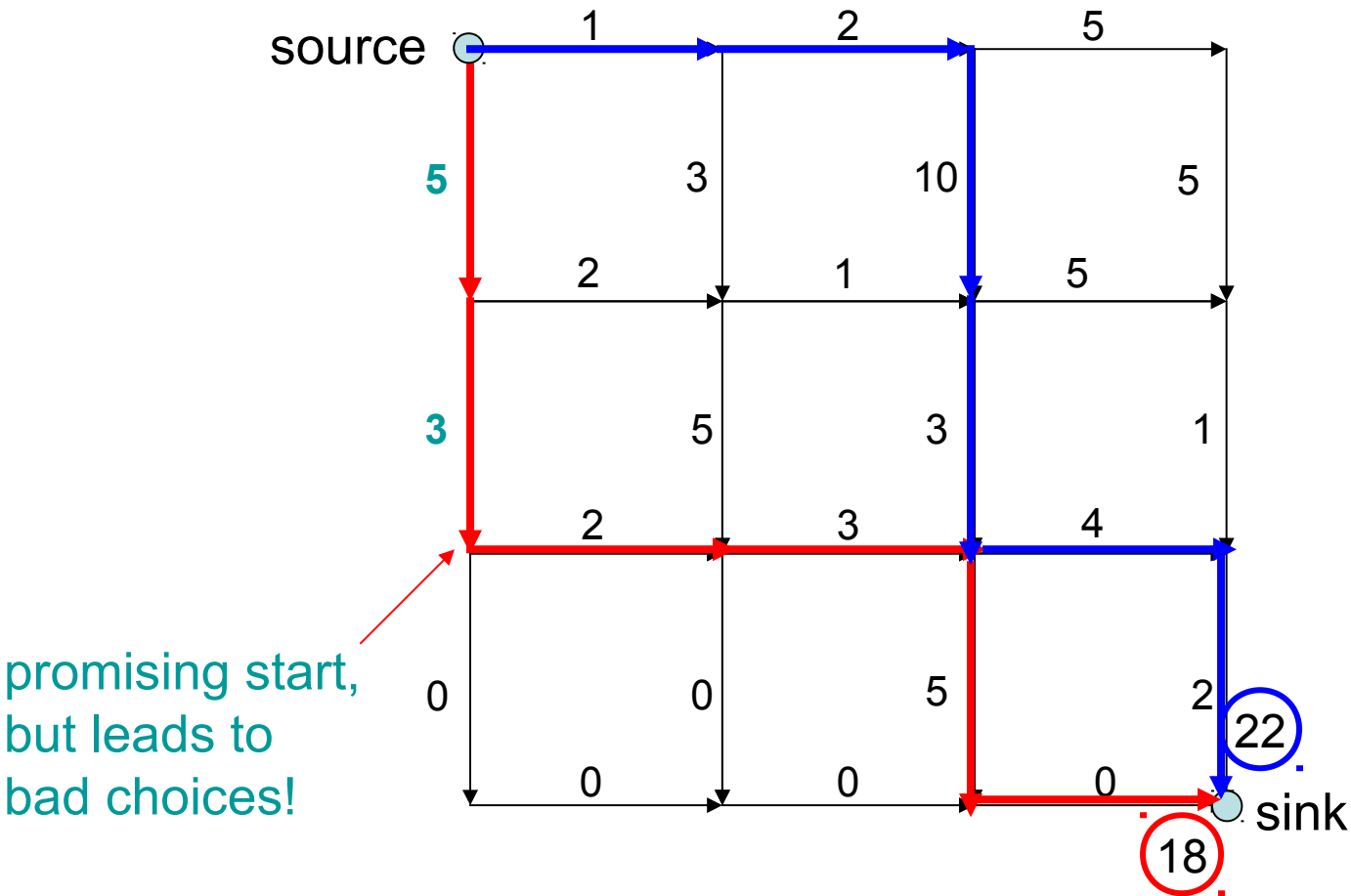
MTP Greedy Algorithm: Example



MTP Greedy Algorithm: Example



MTP: Greedy Algorithm Is Not Optimal



MTP: Simple Recursive Program

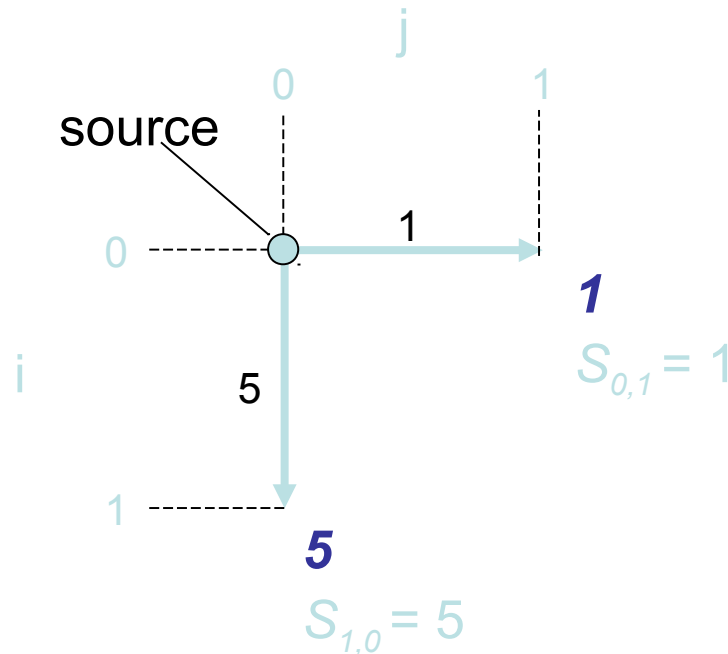
1. **MT**(n, m)
2. **if** $n=0$ or $m=0$
3. **return** $MT(n, m)$
4. $x \leftarrow MT(n-1, m) + \text{length of the edge from } (n-1, m) \text{ to } (n, m)$
5. $y \leftarrow MT(n, m-1) + \text{length of the edge from } (n, m-1) \text{ to } (n, m)$
6. **return** $\max\{x, y\}$

What's wrong with this approach?

Here's what's wrong

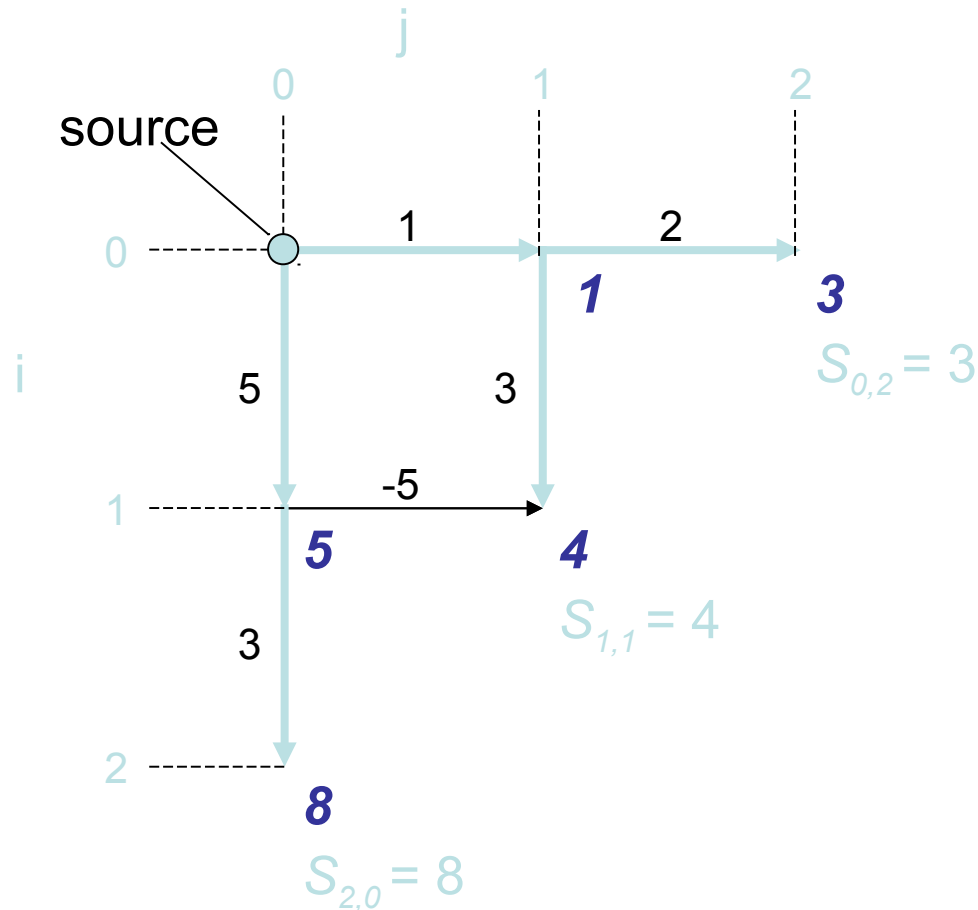
- $M(n, m)$ needs $M(n, m-1)$ and $M(n-1, m)$
- Both of these need $M(n-1, m-1)$
- So $M(n-1, m-1)$ will be computed at least twice
- Dynamic programming: the same idea as this recursive algorithm, but keep all intermediate results in a table and reuse

MTP: Dynamic Programming



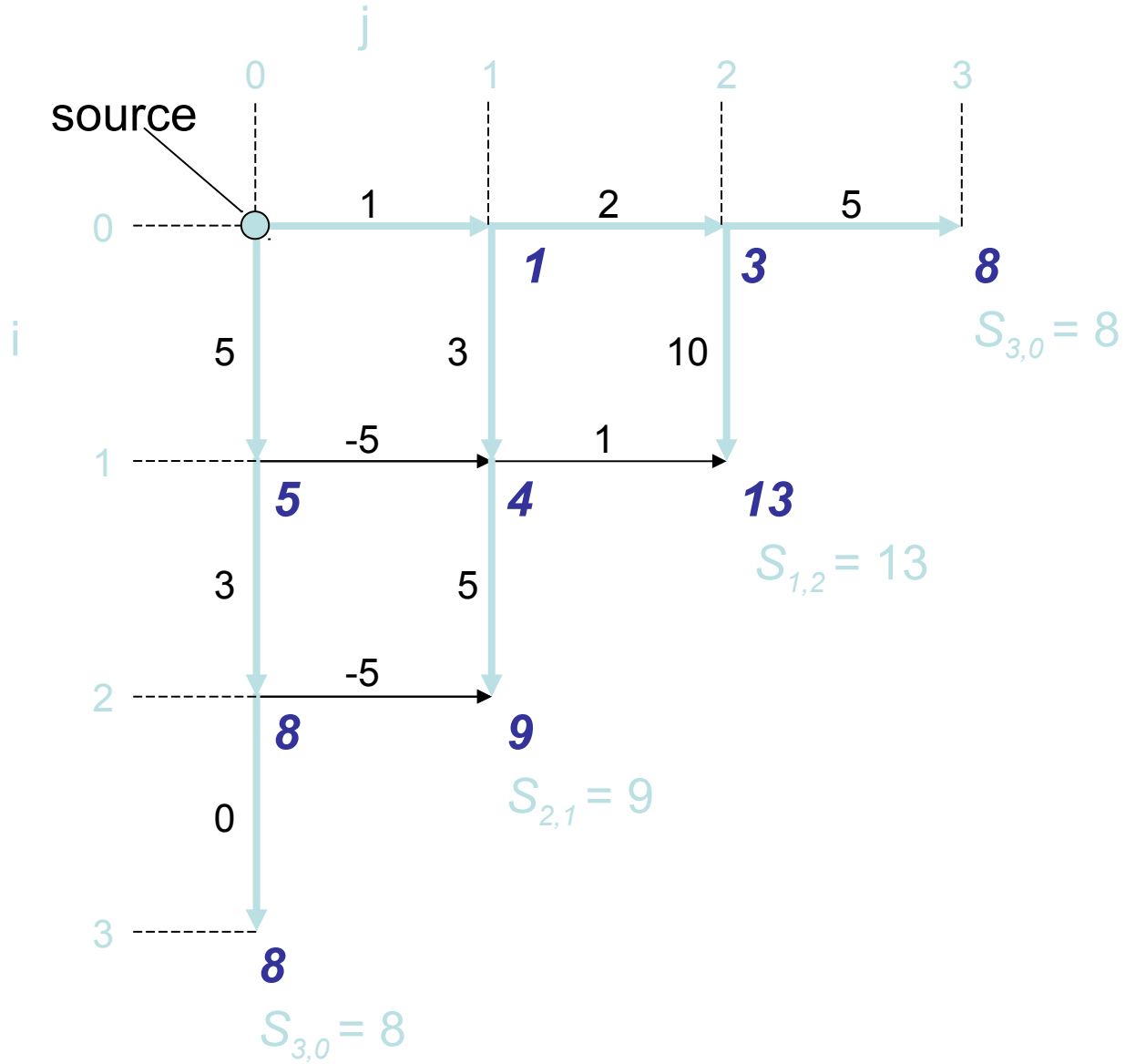
- Calculate optimal path score for each vertex in the graph
- A given vertex's score is the maximum sum of incoming edge weight and prior vertex's score (along that incoming edge).

MTP: Dynamic Programming (cont'd)

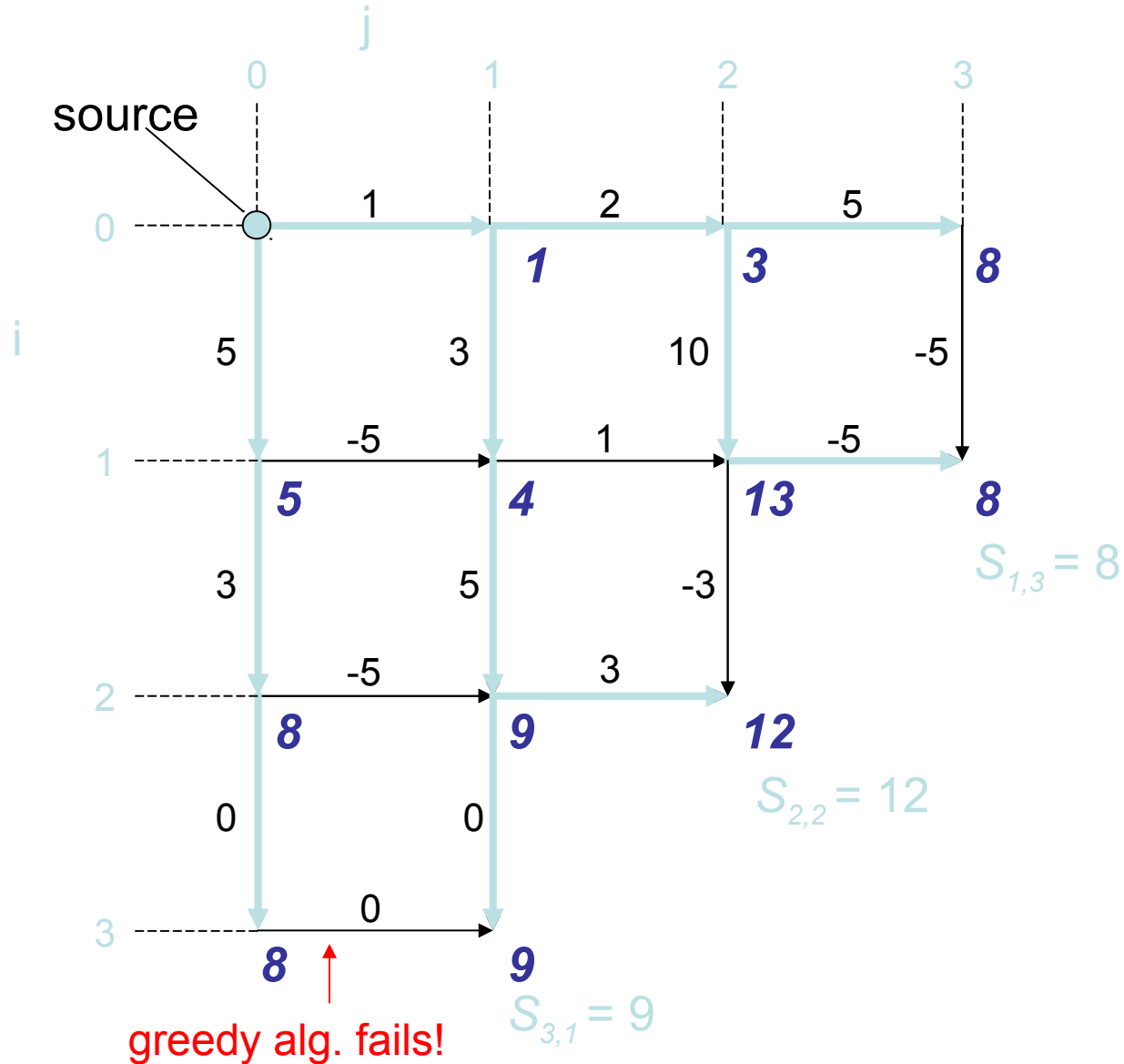


- Light blue edges represent edges selected by the algorithm as maximal.

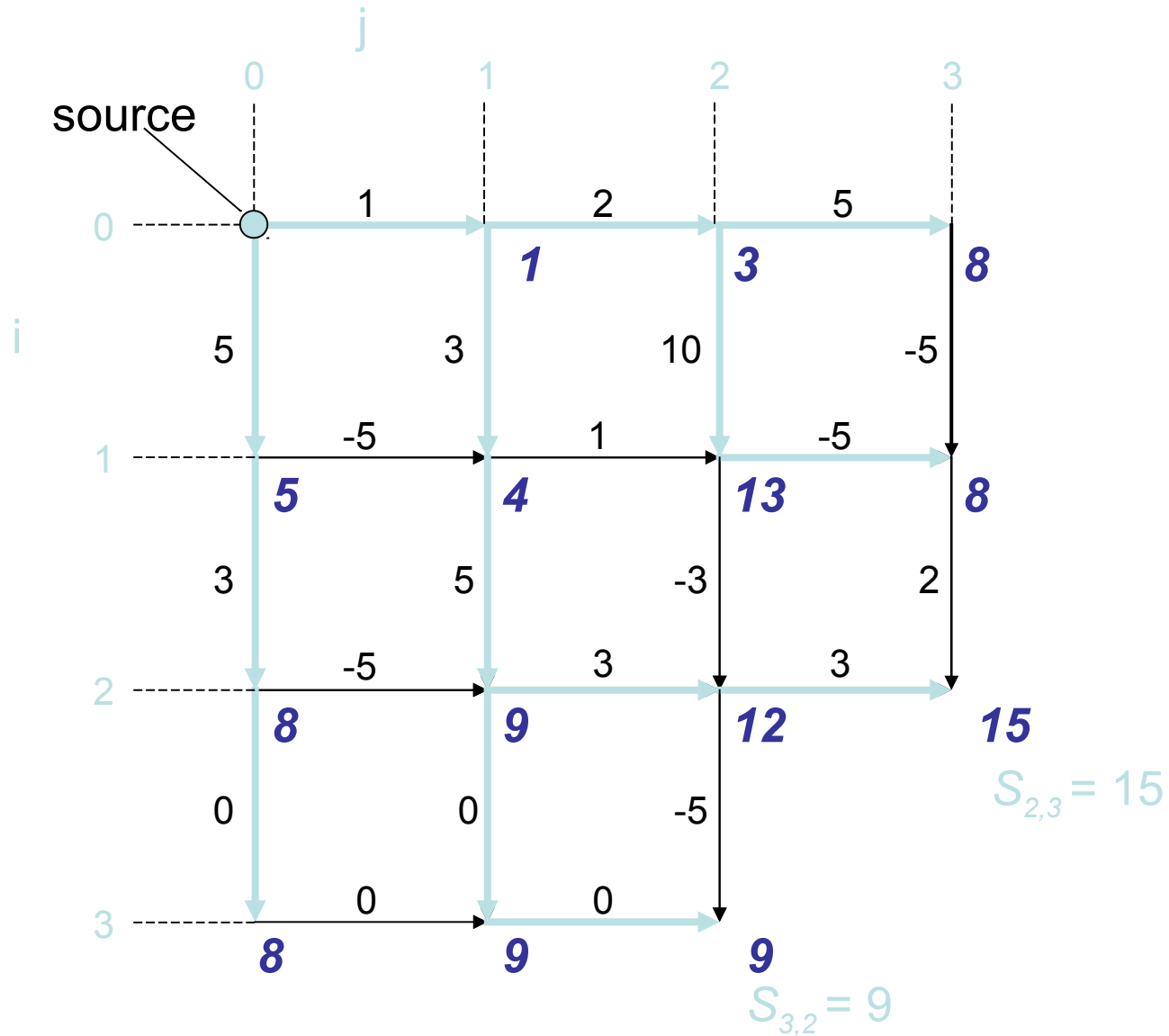
MTP: Dynamic Programming (cont'd)



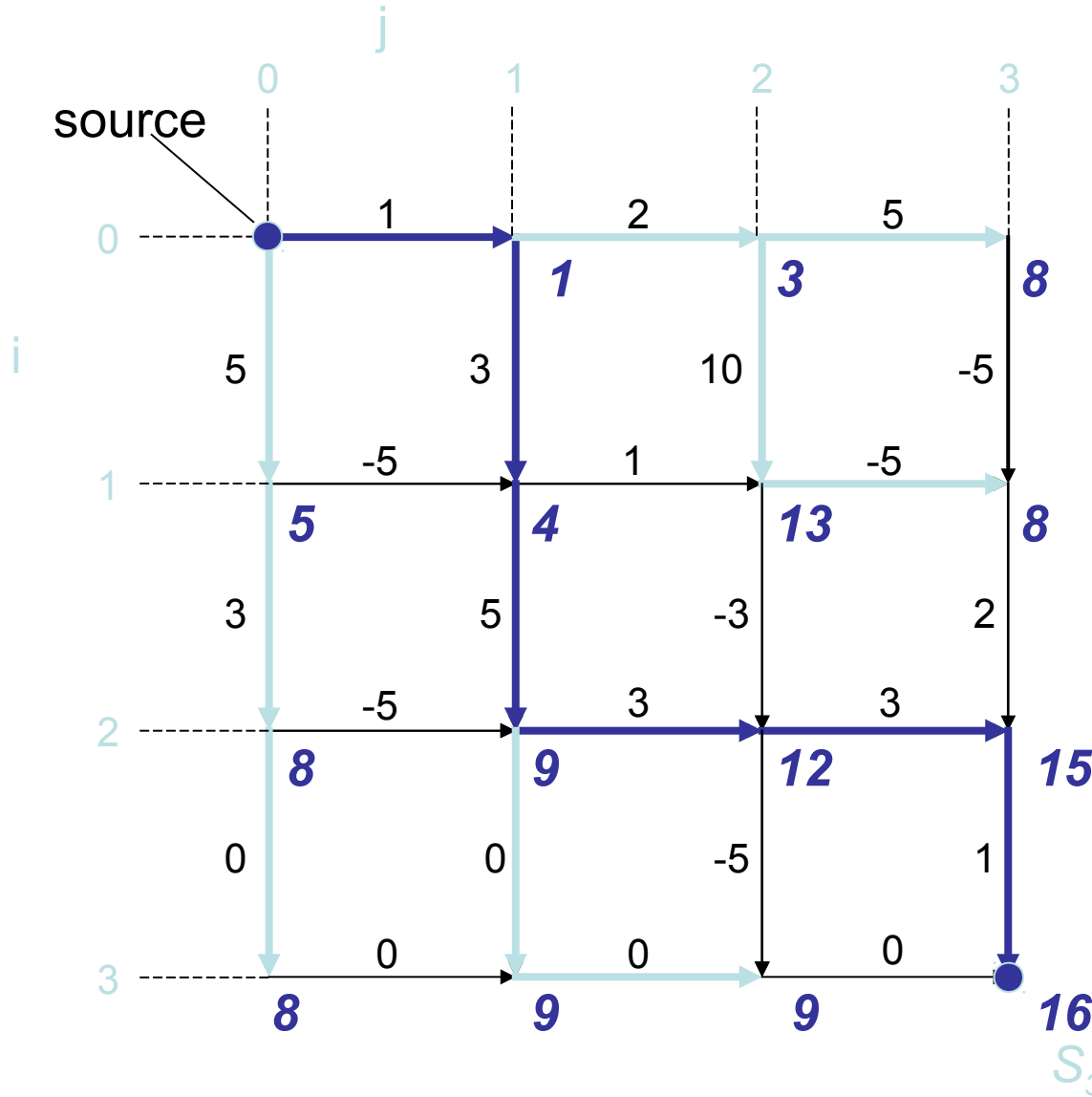
MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



Done!

- Once we reach the sink, we backtrack along gold edges to the source to find the optimal (blue) path.

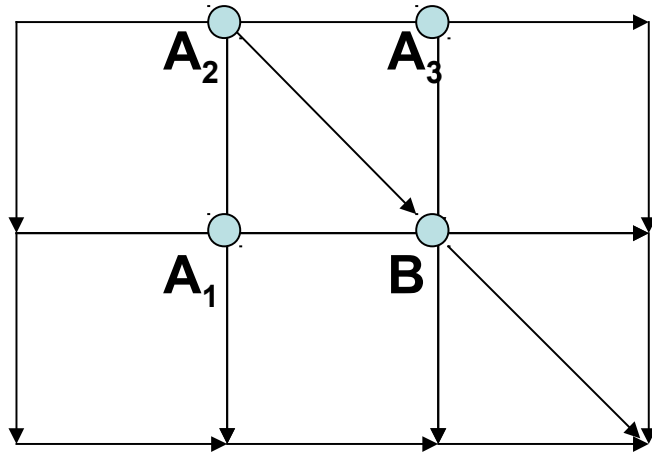
MTP: Recurrence

Computing the score $s_{i,j}$ for a point (i,j) by the recurrence relation:

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

The running time is **$n \times m$** for a **n** by **m** grid
(**n** = # of rows, **m** = # of columns)

Manhattan Is Not A Perfect Grid



What about diagonals?

- The score at point B is given by:

$$s_B = \max \left\{ \begin{array}{l} s_{A_1} + \text{weight of the edge } (A_1, B) \\ s_{A_2} + \text{weight of the edge } (A_2, B) \\ s_{A_3} + \text{weight of the edge } (A_3, B) \end{array} \right.$$

Recursion for an Arbitrary Graph

- We would like to compute the score for point \mathbf{x} in an arbitrary graph.
- Let $Predecessors(\mathbf{x})$ be the set of vertices with edges leading into \mathbf{x} . Then the recurrence is given by:

$$s_x = \max_{y \text{ in } Predecessors(x)} \left\{ s_y + \text{weight of vertex } (y, x) \right\}$$

- The running time for a graph with \mathbf{E} edges is $O(\mathbf{E})$, since each edge is evaluated once.

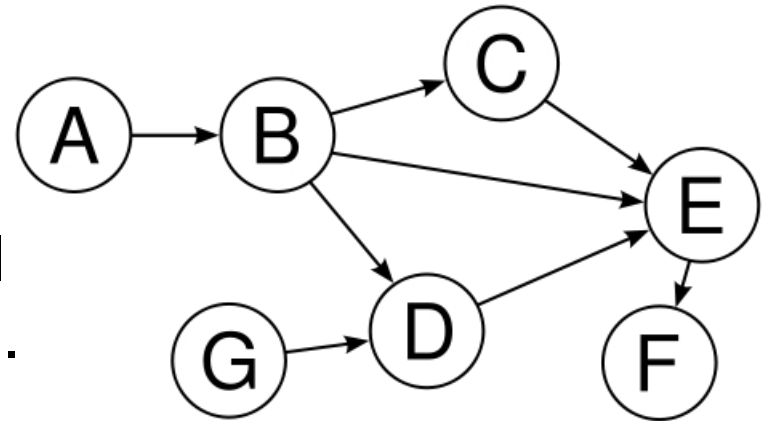
Recursion for an Arbitrary Graph: Problem

- The only hitch is that we must decide on the order in which we visit the vertices.
- By the time the vertex x is analyzed, the values s_y for all its predecessors y should already be computed.
- If the graph has a cycle, we will get stuck in the pattern of going over and over the same cycle.
- In the Manhattan graph, we escaped this problem by requiring that we could only move east or south. This is what we would like to generalize...

Some Graph Theory

Terminology

- **Directed Acyclic Graph (DAG):** A graph in which each edge is provided an orientation, and which has no cycles.
 - We represent the edges of a DAG with directed arrows.
- In the following example, we can move along the edge from B to C, but not from C to B.
- Note that BCE does not form a cycle, since we cannot travel from B to C to E and back to B.

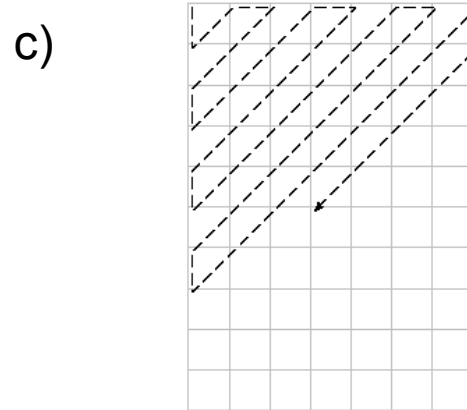
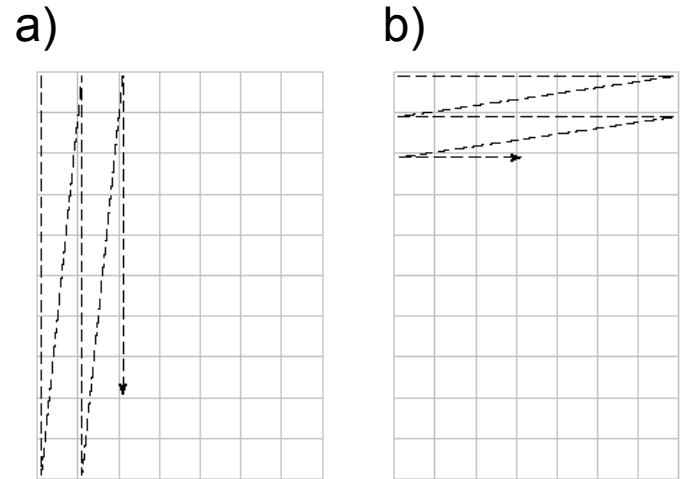


Traveling in the Grid

- By the time the vertex x is analyzed, the values s_y for all its predecessors y should be computed – otherwise we are in trouble.
- We need to traverse the vertices in some order
- For a grid, can traverse vertices row by row, column by column, or diagonal by diagonal
- If we had, instead of a (directed) grid, an arbitrary DAG (directed acyclic graph) ?

Traversing the Manhattan Grid

- 3 different strategies:
 - a) Column by column
 - b) Row by row
 - c) Along diagonals

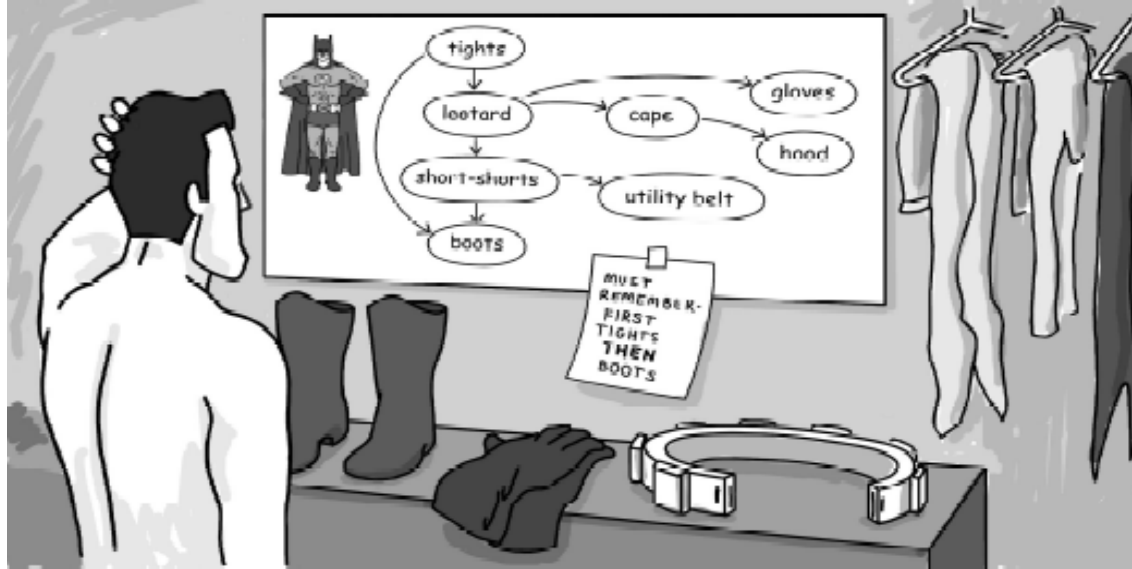


Topological Ordering

- A numbering of vertices of the graph is called ***topological ordering*** of the DAG if every edge of the DAG connects a vertex with a smaller label to a vertex with a larger label
- In other words, if vertices are positioned on a line in an increasing order of labels then all edges go from left to right.
- How to obtain a topological sort (???)

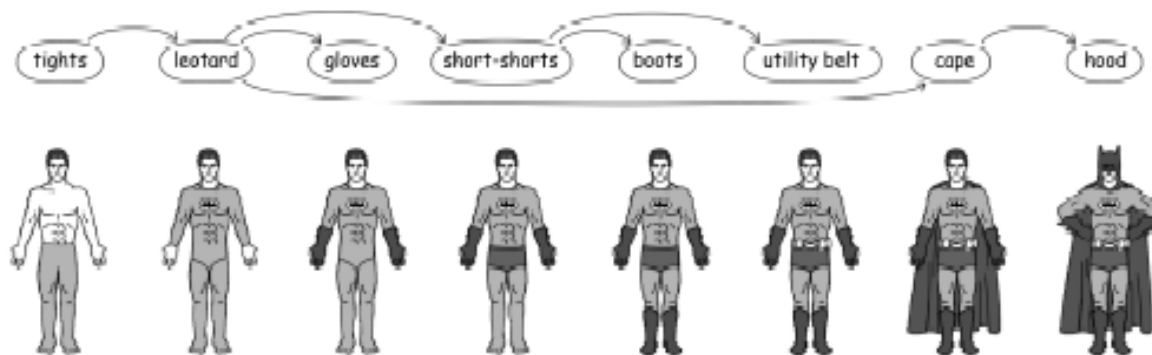
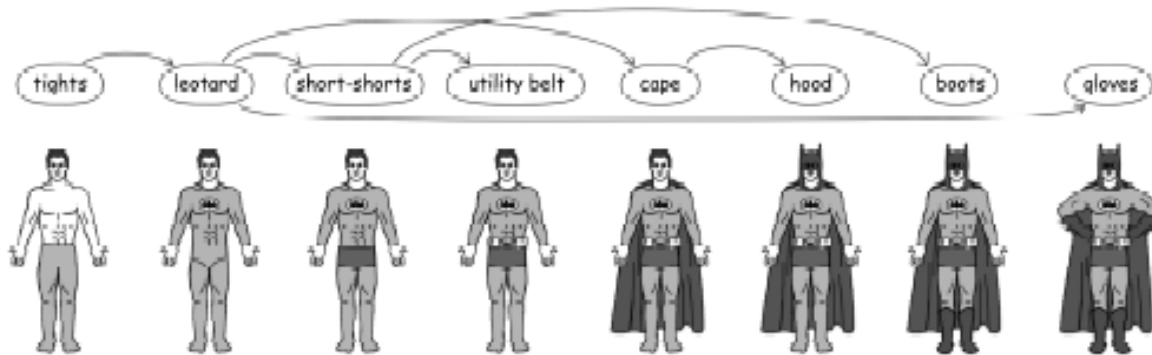
Topological Ordering: Example

- A superhero's costume can be represented by a DAG: he can't put his boots on before his tights!
- He also would like an understandable representation of the graph, so that he can dress quickly.



Topological Ordering: Example

- Here are two different topological orderings of his DAG:



Longest Path in a DAG: Formulation

- Goal: Find a longest path between two vertices in a weighted DAG.
- Input: A weighted DAG \mathbf{G} with source and sink vertices.
- Output: A longest path in \mathbf{G} from source to sink.
- **Note**: Now we know that we can apply a topological ordering to \mathbf{G} , and then use dynamic programming to find the longest path in \mathbf{G} .

Longest Path in DAG: Dynamic Programming

- Suppose vertex v has indegree 3 and predecessors $\{u_1, u_2, u_3\}$
- Longest path to v from source is:

$$s_v = \max_{\text{of}} \left\{ \begin{array}{l} s_{u_1} + \text{weight of edge from } u_1 \text{ to } v \\ s_{u_2} + \text{weight of edge from } u_2 \text{ to } v \\ s_{u_3} + \text{weight of edge from } u_3 \text{ to } v \end{array} \right.$$

In General:

$$s_v = \max_u (s_u + \text{weight of edge from } \mathbf{u} \text{ to } \mathbf{v})$$

Alignment

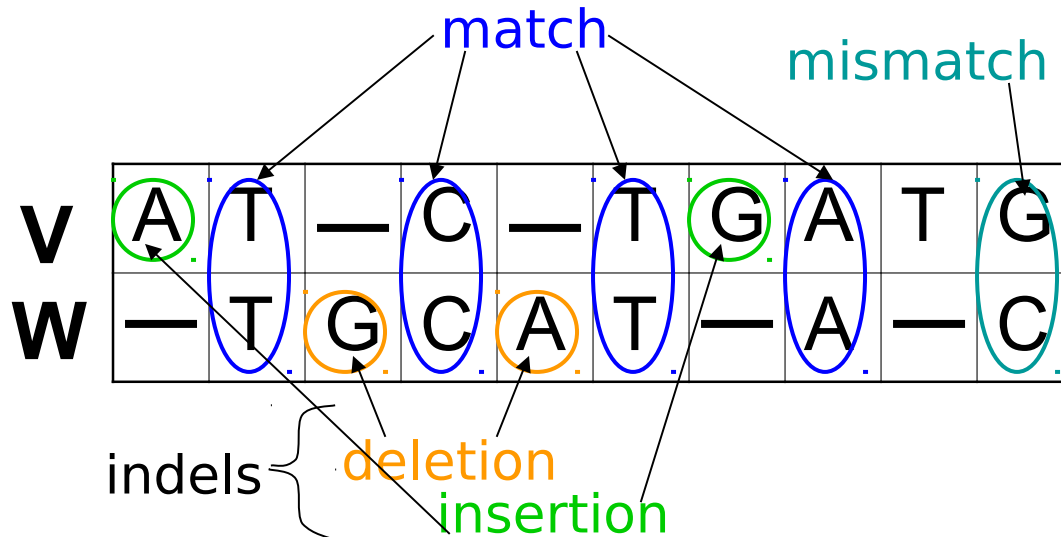
Aligning DNA Sequences

Alignment : $2 \times k$ matrix ($k \geq m, n$)

V = ATCTGATG $n = 8$

W = TGCATAC $m = 7$

4 matches
1 mismatches
2 insertions
2 deletions



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 $\mathbf{v} \rightarrow \mathbf{w}$

Longest Common Subsequence (LCS) – Alignment without Mismatches

- Given two sequences

$$\mathbf{v} = v_1 v_2 \dots v_m \text{ and } \mathbf{w} = w_1 w_2 \dots w_n$$

- The LCS of \mathbf{v} and \mathbf{w} is a sequence of positions in

$$\mathbf{v}: 1 \leq i_1 < i_2 < \dots < i_t \leq m$$

and a sequence of positions in

$$\mathbf{w}: 1 \leq j_1 < j_2 < \dots < j_t \leq n$$

such that i_t -th letter of \mathbf{v} equals to j_t -letter of \mathbf{w} and t is maximal

LCS: Example

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

Computing LCS

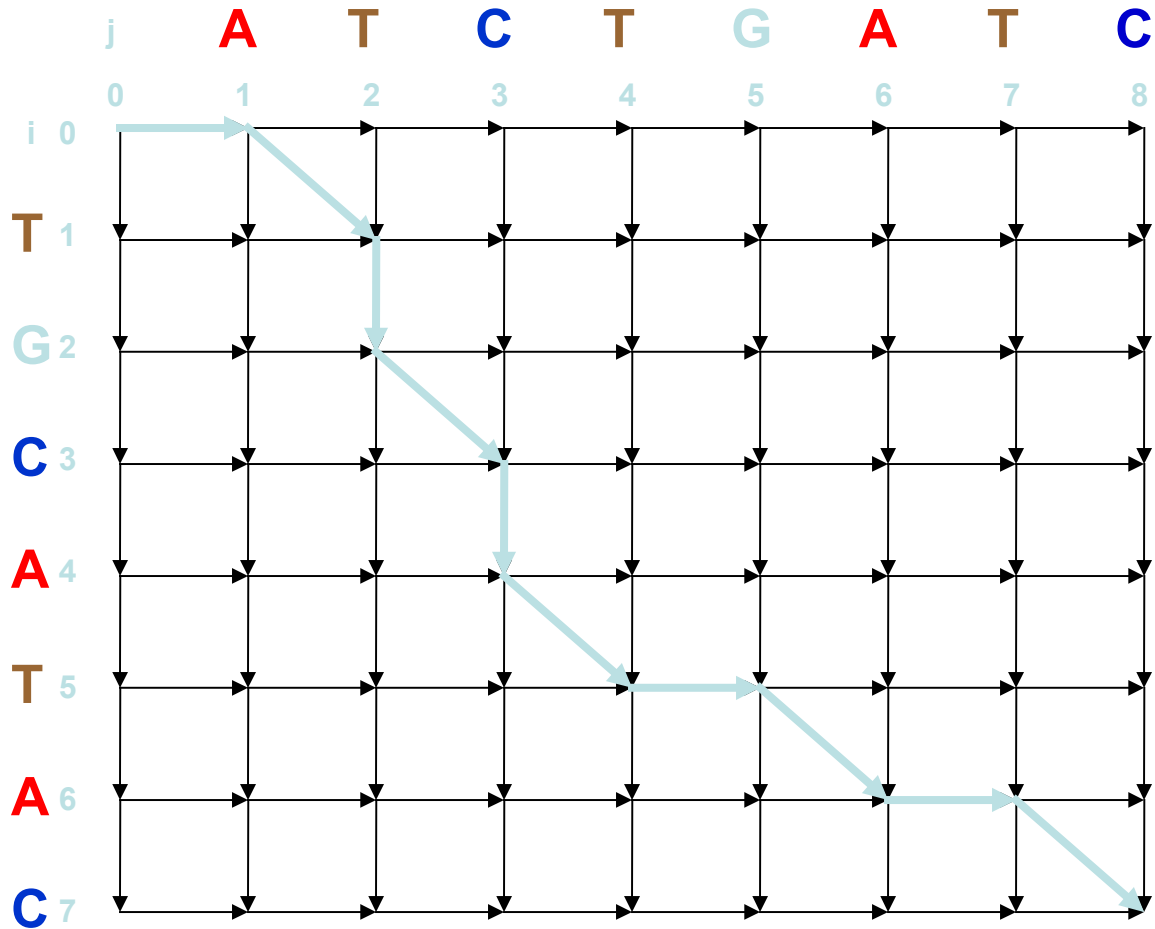
Let \mathbf{v}_i = prefix of \mathbf{v} of length i : $v_1 \dots v_i$

and \mathbf{w}_j = prefix of \mathbf{w} of length j : $w_1 \dots w_j$

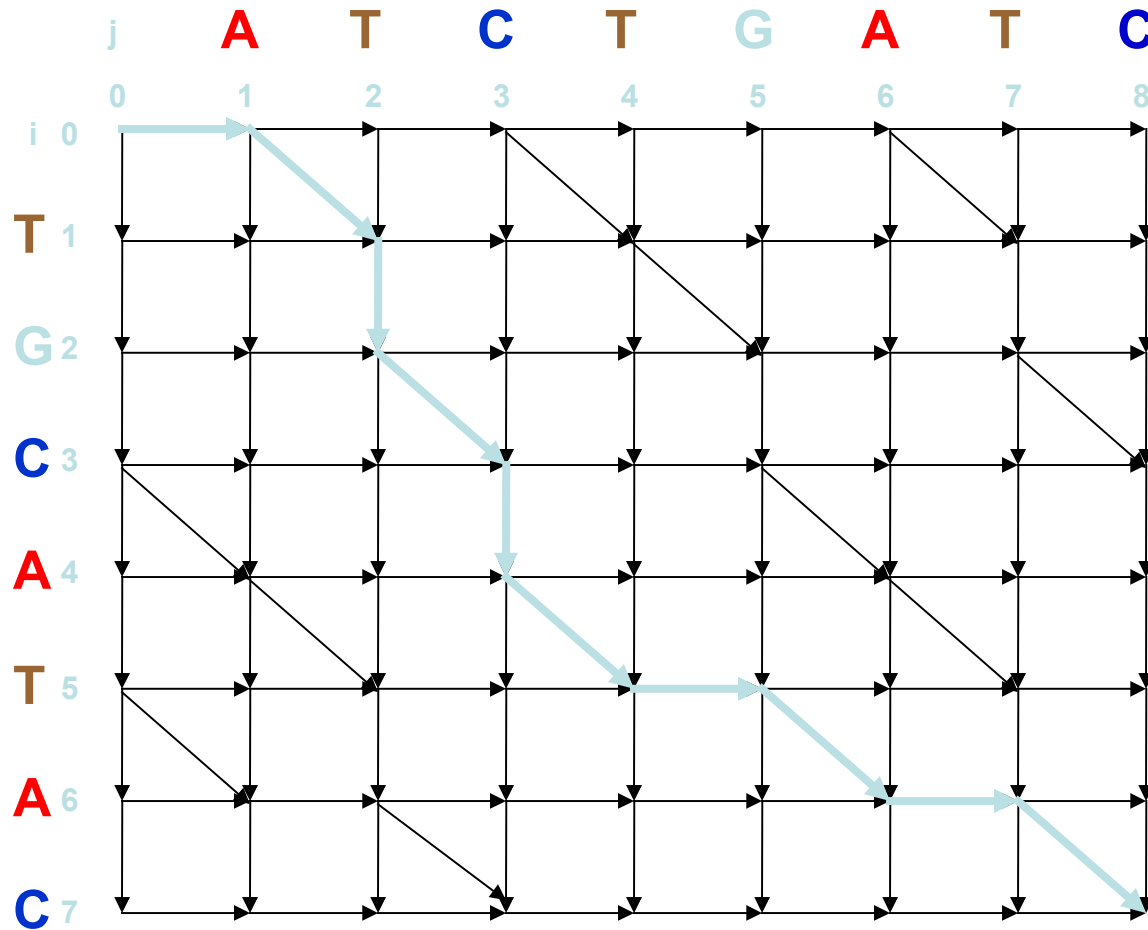
The length of $\text{LCS}(\mathbf{v}_i, \mathbf{w}_j)$ is computed by:

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

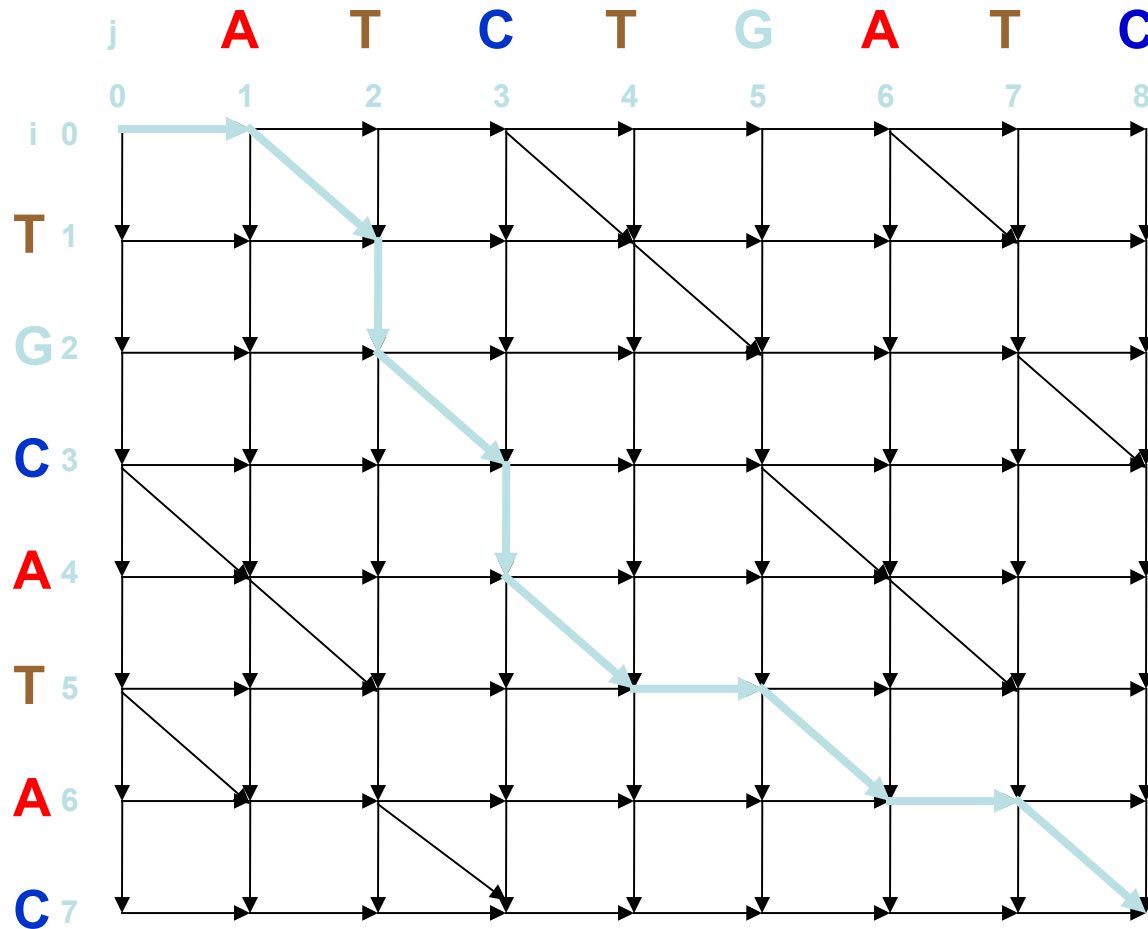
LCS Problem as Manhattan Tourist Problem



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

Backtracking


- $s_{i,j}$ allows us to compute the length of LCS for v_i and w_j
- $s_{m,n}$ gives us the length of the LCS for v and w
- How do we print the actual LCS ?
- At each step, we chose an optimal decision $s_{i,j} = \max (\dots)$
- Record which of the choices was made in order to obtain this max

Computing LCS

Let \mathbf{v}_i = prefix of \mathbf{v} of length i : $v_1 \dots v_i$

and \mathbf{w}_j = prefix of \mathbf{w} of length j : $w_1 \dots w_j$

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

$$s_{i,j} = \max \begin{cases} s_{i-1,j} & \text{ (indicated by a pink arrow pointing down)} \\ s_{i,j-1} & \text{ (indicated by a yellow arrow pointing right)} \\ s_{i-1,j-1} + 1 & \text{ if } v_i = w_j \end{cases}$$


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.        $s_{i,j} \leftarrow \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1, \text{ if } v_i = w_j \end{cases}$ 
9.       if  $s_{i,j} = s_{i-1,j}$ 
10.        if  $s_{i,j} = s_{i,j-1}$ 
11.        if  $s_{i,j} = s_{i-1,j-1} + 1$ 
12.         $b_{i,j} \leftarrow$ 
13.   return  $(s_{n,m}, b)$ 
```


Printing LCS: Backtracking

```
1. PrintLCS(b, v, i, j)
2.     if i = 0 or j = 0
3.         return
4.     if bi,j = ↖
5.         PrintLCS(b, v, i-1, j-1)
6.         print vi
7.     else
8.         if bi,j = ↑
9.             PrintLCS(b, v, i-1, j)
10.        else
11.            PrintLCS(b, v, i, j-1)
```

From LCS to Alignment

- The **Longest Common Subsequence** 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

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} - \sigma \\ s_{i-1,j-1} + 1, & \text{if } v_i = w_i \text{ (match)} \\ s_{i-1,j-1} - \mu, & \text{if } v_i \neq w_i \text{ (mismatch)} \end{cases}$$

μ =mismatch penalty,

σ : indel penalty

Scoring Matrices

To generalize scoring, consider a $(4+1) \times (4+1)$ **scoring matrix** δ .

In the case of an amino acid sequence alignment, the scoring matrix would be a $(20+1) \times (20+1)$ 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: Positive Mismatches

- Notice that although R and K are different amino acids, they have a positive mismatch score.
- Why? They are both positively charged amino acids → this mismatch will not greatly change the function of the protein.

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

AKRANR

KAAANK

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

Scoring Matrix: Positive Mismatches

- Notice that although R and K are different amino acids, they have a positive mismatch score.
- Why? They are both positively charged amino acids → this mismatch will not greatly change the function of the protein.

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

AKRANR

KAAANK

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

Scoring Matrix: Positive Mismatches

- Notice that although R and K are different amino acids, they have a positive mismatch score.
- Why? They are both positively charged amino acids → this mismatch will not greatly change the function of the protein.

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

AKRANR

KAAANK

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

Scoring matrices

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

PAM

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

PAM_x

- $\text{PAM}_x = \text{PAM}_1^x$
 - $\text{PAM}_{250} = \text{PAM}_1^{250}$
- PAM_{250} is a widely used scoring matrix:

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

BLOSUM

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

The Blossum50 Scoring Matrix

[illegible]

Conservation

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

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

```

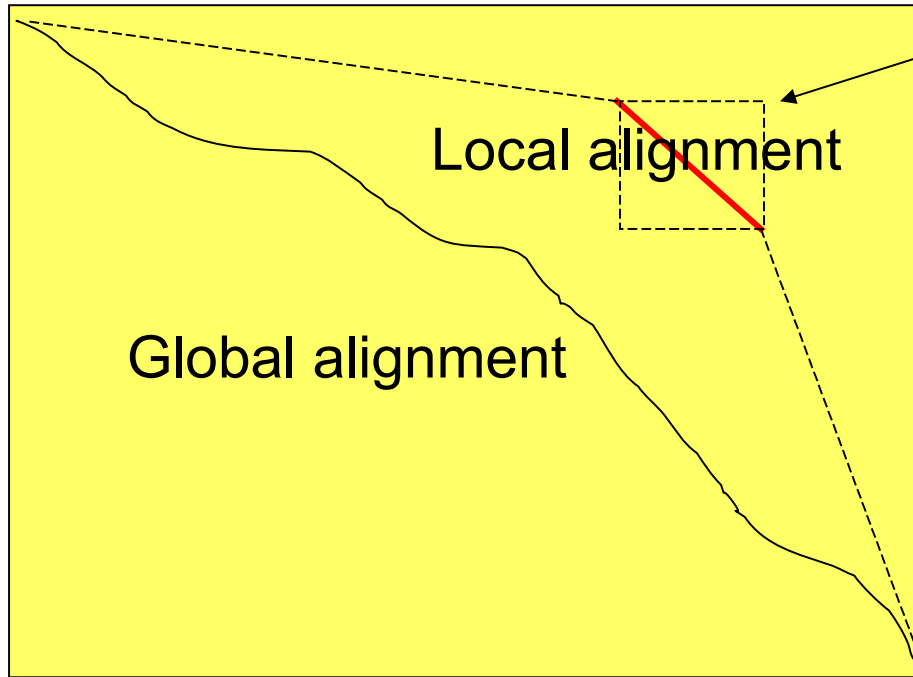
--T--CC-C-AGT--TATGT-CAGGGGACACG-A-GCATGCAGA-GAC
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
AATTGCCGCC-GTCGT-T-TTCAG-----CA-GTTATG-T-CAGAT--C
  
```

- Local Alignment—better alignment to find conserved segment

```

                tccCAGTTATGTCAGgggacacgagcatgcagagac
                  |||||
aattgccgccgtcggttttcagCAGTTATGTCAGatc
  
```

Local Alignment: Example



Compute a "mini"
Global Alignment to
get Local Alignment

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

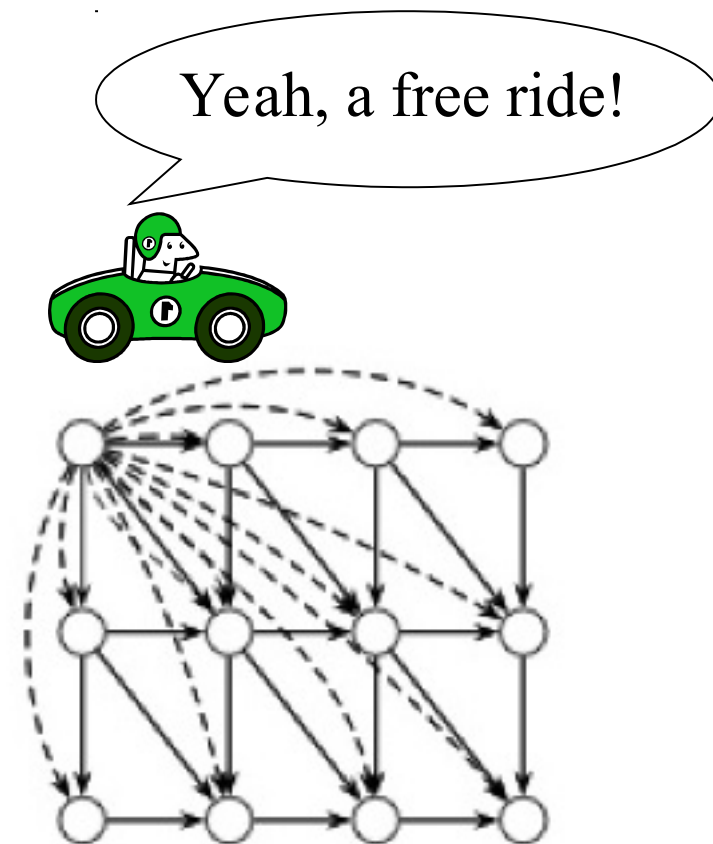
- Goal: Find the best local alignment between two strings
- Input : Strings \mathbf{v} , \mathbf{w} and scoring matrix δ
- Output : Alignment of substrings of \mathbf{v} and \mathbf{w} whose alignment score is maximum among all possible alignment of all possible substrings

The Problem Is ...

- Long run time $O(n^4)$:
 - In the grid of size $n \times n$ there are $\sim n^2$ vertices (i,j) that may serve as a source.
 - For each such vertex computing alignments from (i,j) to (i',j') takes $O(n^2)$ time.
- This can be remedied by allowing every point to be the starting point

Local Alignment Solution: Free Rides

- The solution actually comes from *adding* vertices to the edit graph.
- The dashed edges represent the “free rides” from $(0, 0)$ to every other node.
 - Each “free ride” is assigned an edge weight of 0.
 - If we start at $(0, 0)$ instead of (i, j) and maximize the longest path to (i', j') , we will obtain the local alignment.



The Local Alignment Recurrence

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

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

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

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
ATATTGC

↑
This is more
likely.

↖ ↗
Normal scoring would
give the same score
for both alignments

ATAG GC
AT_GTGC

↑
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 $\rho > 0$ is the penalty for introducing a gap:

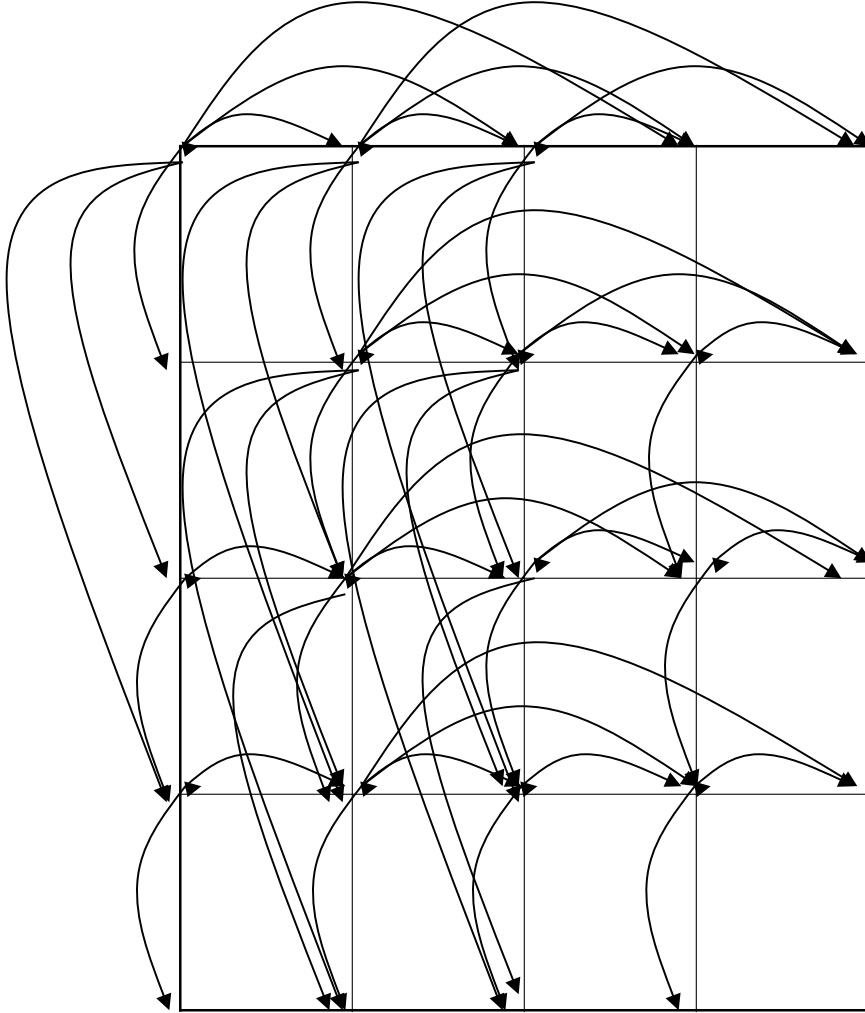
gap opening penalty

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.

Adding “Affine Penalty” Edges to the Edit Graph

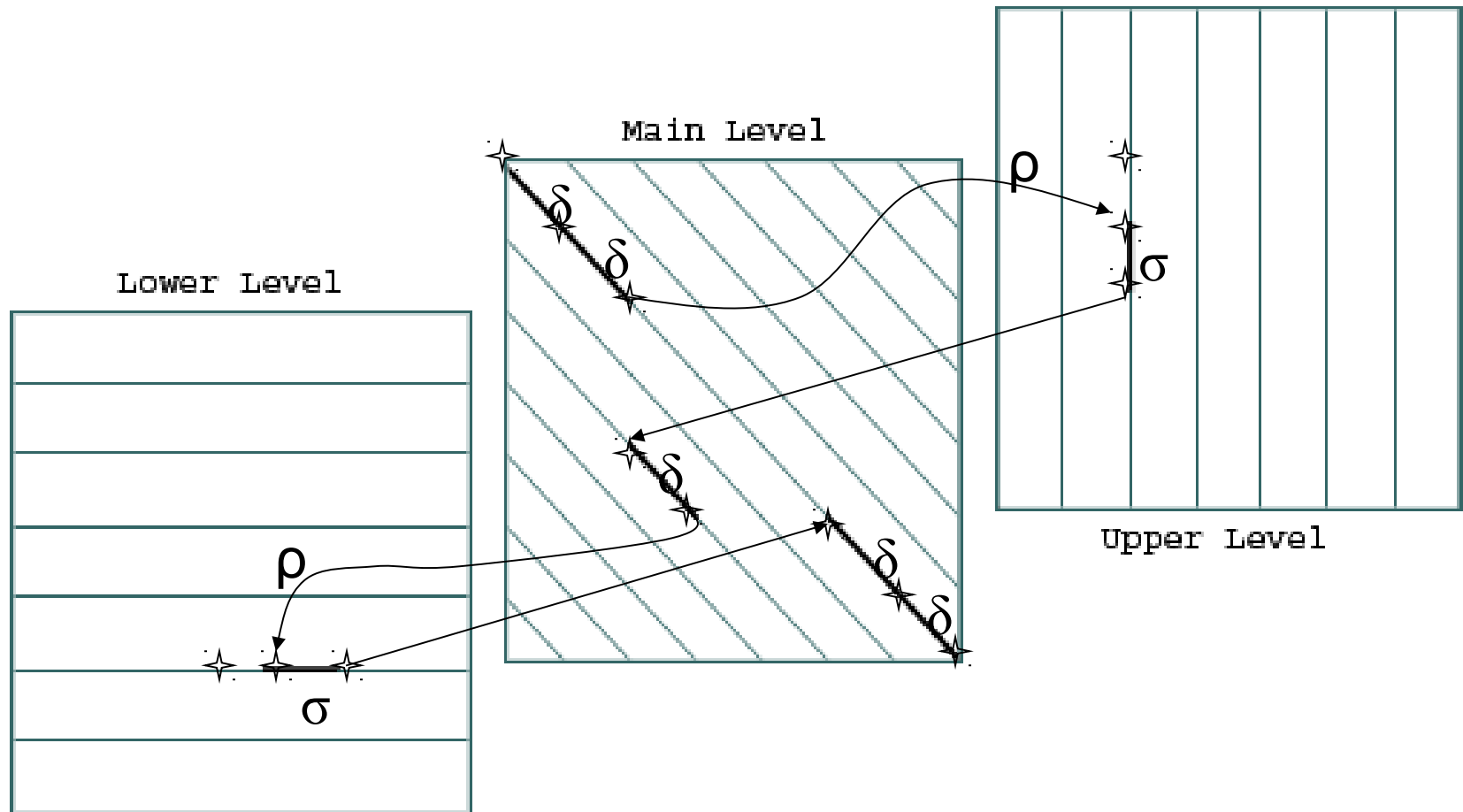


There are many such edges!

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



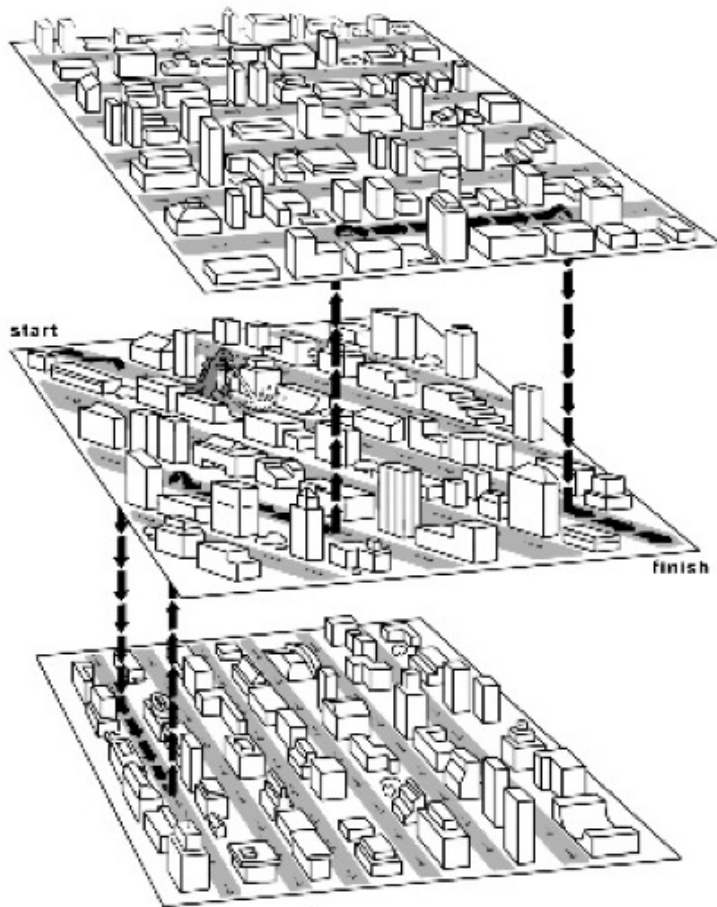
Affine Gap Penalties and 3 Layer Manhattan Grid

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

Switching between 3 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 ($-\rho - \sigma$)
- 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 w

Matches/Mismatches

Gaps in v

Affine Gap Penalty Recurrences

$$s_{i,j}^{\downarrow} = \max \begin{cases} s_{i-1,j}^{\downarrow} - \sigma & \text{Continue gap in } w \text{ (deletion)} \\ s_{i-1,j} - (\rho + \sigma) & \text{Start gap in } w \text{ (deletion): from middle} \end{cases}$$

$$s_{i,j}^{\rightarrow} = \max \begin{cases} s_{i,j-1}^{\rightarrow} - \sigma & \text{Continue gap in } v \text{ (insertion)} \\ s_{i,j-1} - (\rho + \sigma) & \text{Start gap in } v \text{ (insertion): from middle} \end{cases}$$

$$s_{i,j} = \max \begin{cases} s_{i-1,j-1} + \delta(v_i, w_j) & \text{Match or mismatch} \\ s_{i,j}^{\downarrow} & \text{End deletion: from top} \\ s_{i,j}^{\rightarrow} & \text{End insertion: from bottom} \end{cases}$$