

Dynamic Programming: Edit Distance

Slides adapted from Jones & Pevzner 2004

<http://bix.ucsd.edu/bioalgorithms/>

DNA Sequence Comparison: Early Success Story

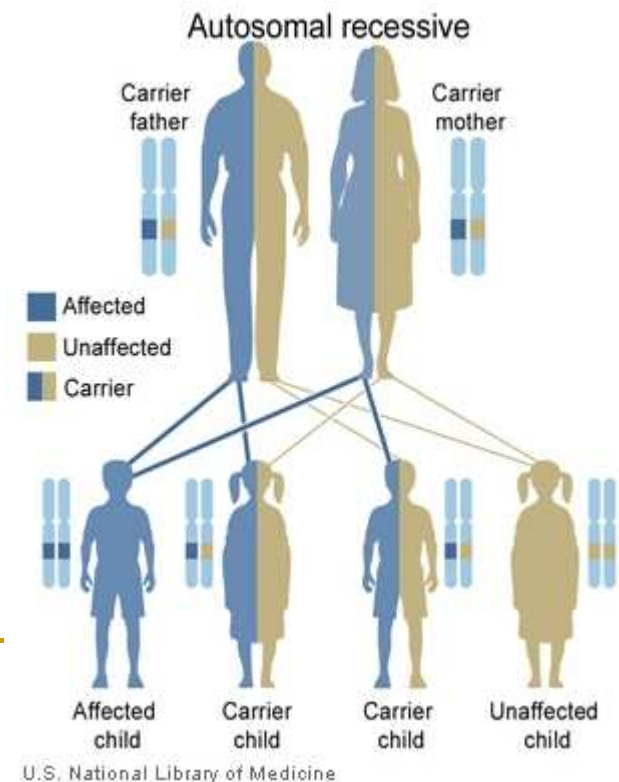
- Finding sequence similarities with genes of known function is a common approach to infer a newly sequenced gene's function
 - In 1984 Russell Doolittle and colleagues found similarities between cancer-causing gene and normal platelet-derived growth factor (PDGF) gene
-

Cystic Fibrosis

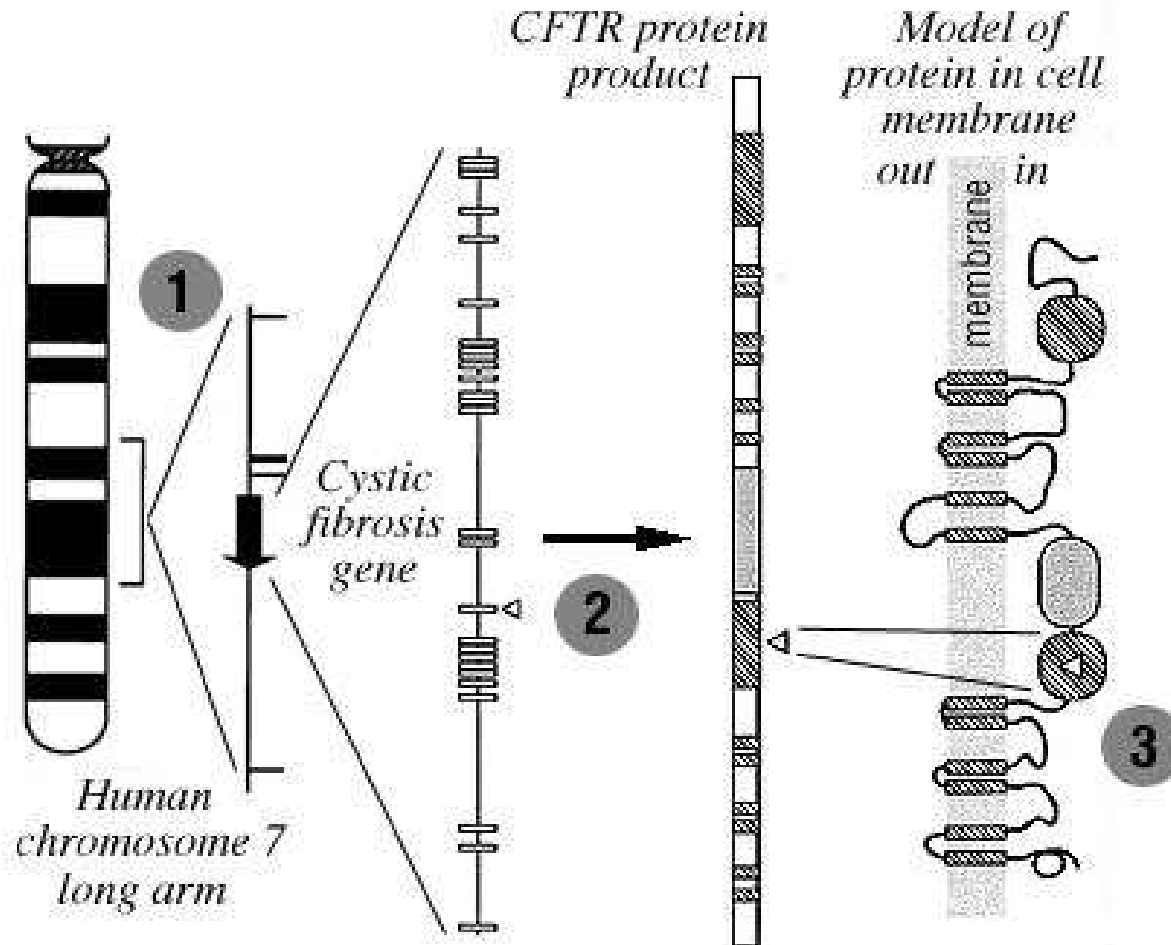
- **Cystic fibrosis** (CF) is a chronic and frequently fatal genetic disease of the body's mucus glands (abnormally high level of mucus in glands). CF primarily affects the respiratory systems in children.
 - Mucus is a slimy material that coats many epithelial surfaces and is secreted into fluids such as saliva
-

Cystic Fibrosis: Inheritance

- In early 1980s biologists hypothesized that CF is an autosomal recessive disorder caused by mutations in a gene that remained unknown until 1989
- Heterozygous carriers are asymptomatic
- Must be homozygously recessive in this gene in order to be diagnosed with CF



Cystic Fibrosis: Finding the Gene



Finding Similarities between the Cystic Fibrosis Gene and ATP binding proteins

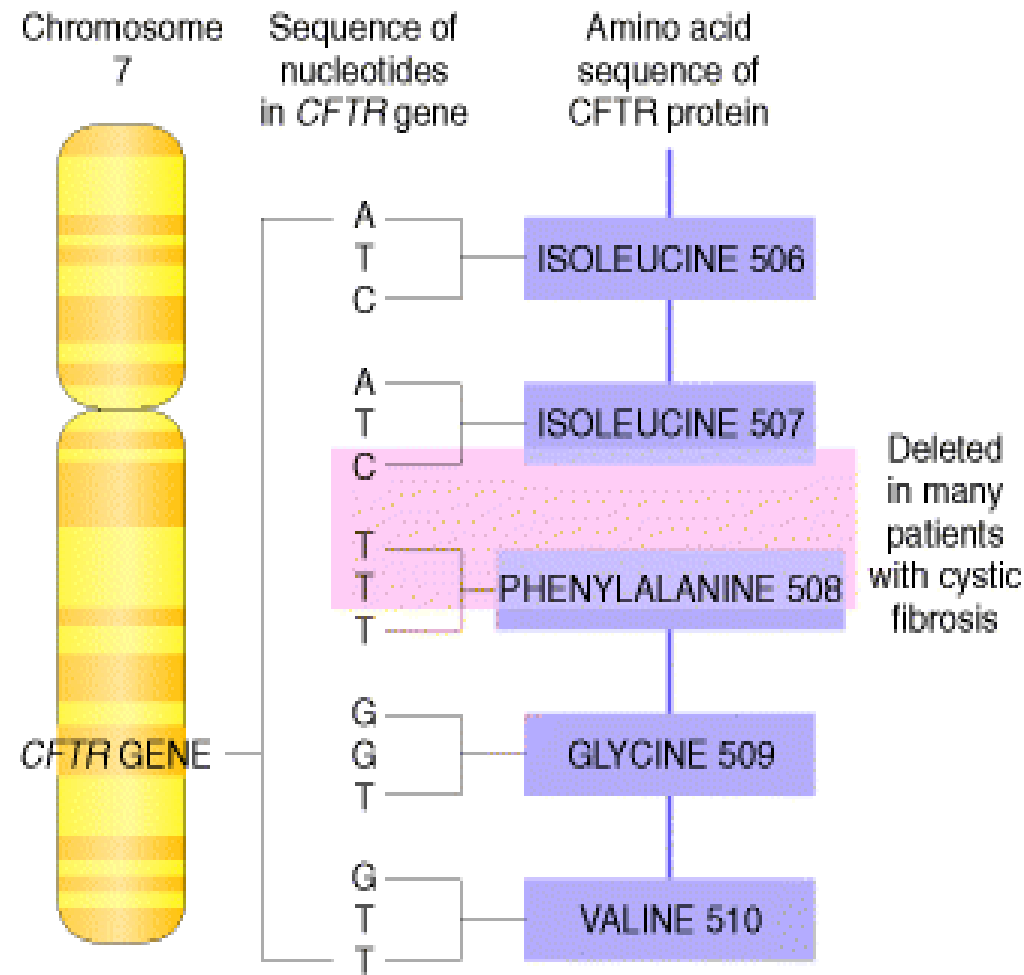
- ATP binding proteins are present on cell membrane and act as transport channel
 - In 1989 biologists found similarity between the cystic fibrosis gene and ATP binding proteins
 - A plausible function for cystic fibrosis gene, given the fact that CF involves sweat secretion with abnormally high sodium level
-

Cystic Fibrosis: Mutation Analysis

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

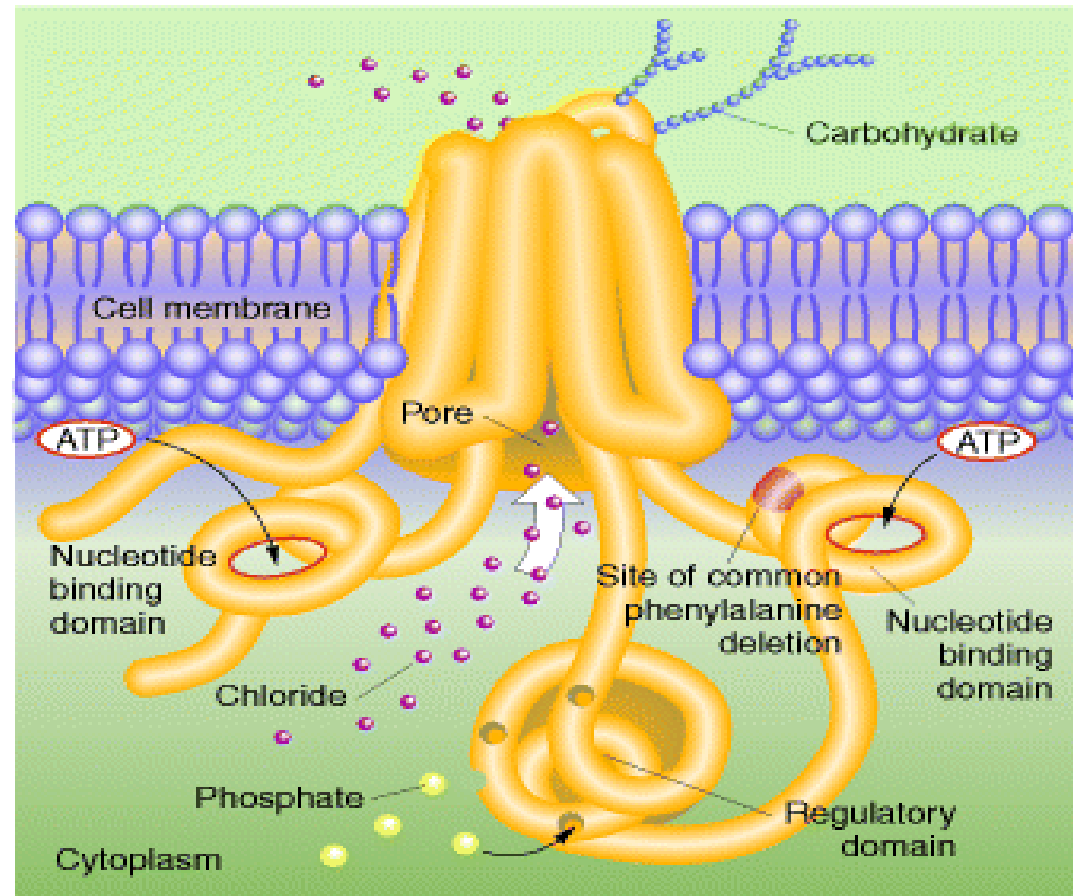
A certain mutation was found in 70% of CF patients, strongly suggesting that it is a predominant genetic diagnostics marker for CF

Cystic Fibrosis and CFTR Gene :



Cystic Fibrosis and the CFTR Protein

- **CFTR (Cystic Fibrosis Transmembrane conductance Regulator)** protein is acting in the cell membrane of epithelial cells that secrete mucus
- These cells line the airways of the nose, lungs, the stomach wall, etc.



Mechanism of Cystic Fibrosis

- The **CFTR protein** (1480 amino acids) regulates a chloride ion channel
 - Adjusts the “wateriness” of fluids secreted by the cell
 - Those with cystic fibrosis are missing one single amino acid in their CFTR
 - Mucus ends up being too thick, affecting many organs
-

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
 - The ***Change Problem*** is a good problem to introduce the idea of dynamic programming
-

The Change Problem

Goal: Convert some amount of money M into given denominations, using the fewest possible number of coins

Input: An amount of money M , and an array of d denominations $\mathbf{c} = (c_1, c_2, \dots, c_d)$, in a decreasing order of value ($c_1 > c_2 > \dots > c_d$)

Output: A list of d integers $i_1, i_2, \dots, i_d$ such that
$$c_1 i_1 + c_2 i_2 + \dots + c_d i_d = M$$
and $i_1 + i_2 + \dots + i_d$ is minimal

Change Problem: Example

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

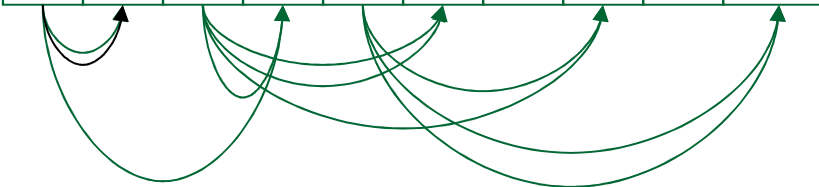
Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	1		1		1					

Only one coin is needed to make change for the values 1, 3, and 5

Change Problem: Example (cont'd)

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	1	2	1	2	1	2		2		2



However, two coins are needed to make change for the values 2, 4, 6, 8, and 10.

Change Problem: Example (cont'd)

Given the denominations 1, 3, and 5, what is the minimum number of coins needed to make change for a given value?

Value	1	2	3	4	5	6	7	8	9	10
Min # of coins	1	2	1	2	1	2	3	2	3	2



Lastly, three coins are needed to make change for the values 7 and 9

Change Problem: Recurrence

This example is expressed by the following recurrence relation:

$$\text{minNumCoins}(M) = \min_{\text{of}} \left\{ \begin{array}{l} \text{minNumCoins}(M-1) + 1 \\ \text{minNumCoins}(M-3) + 1 \\ \text{minNumCoins}(M-5) + 1 \end{array} \right.$$

Change Problem: Recurrence (cont'd)

Given the denominations $\mathbf{c}$: $c_1, c_2, \dots, c_d$, the recurrence relation is:

$$\text{minNumCoins}(M) = \min_{\text{of}} \left\{ \begin{array}{l} \text{minNumCoins}(M-c_1) + 1 \\ \text{minNumCoins}(M-c_2) + 1 \\ \dots \\ \text{minNumCoins}(M-c_d) + 1 \end{array} \right.$$

Change Problem: A Recursive Algorithm

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

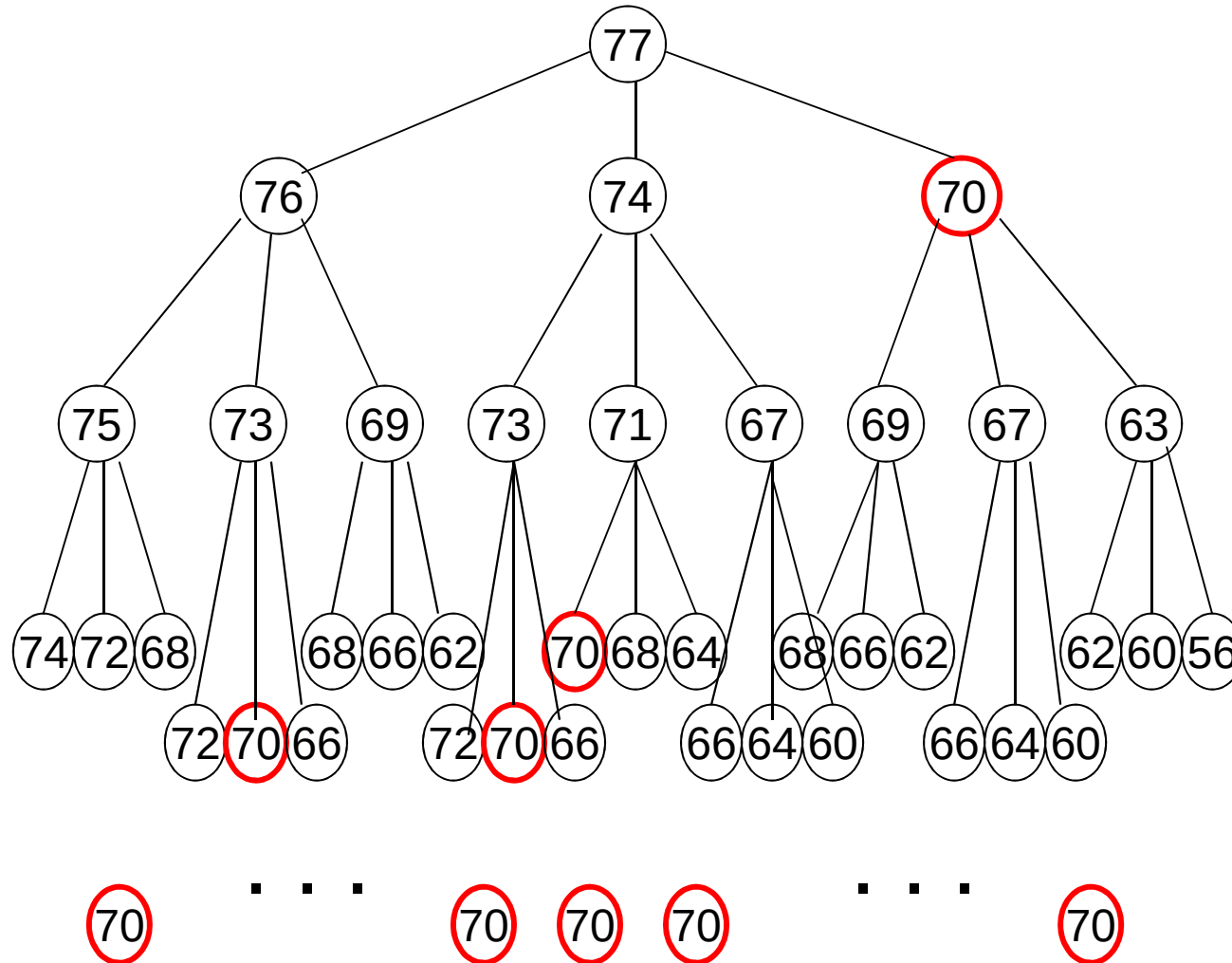
Target value: M

Coins: $c_1, \dots, c_d$

RecursiveChange Is Not Efficient

- It recalculates the optimal coin combination for a given amount of money repeatedly
- i.e., $M = 77$, $c = (1,3,7)$:
 - Optimal coin combo for 70 cents is computed **9** times!

The RecursiveChange Tree



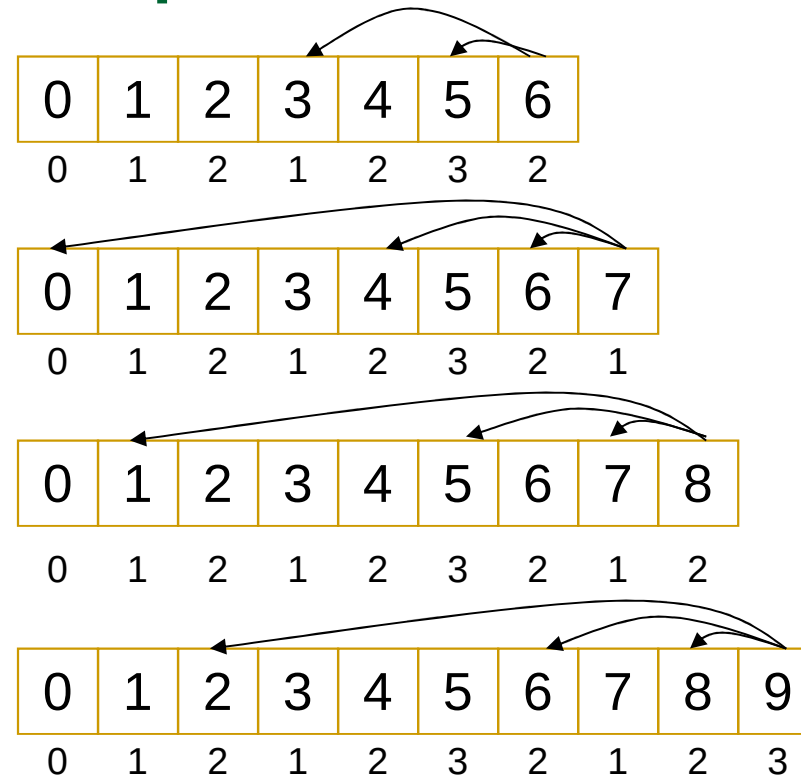
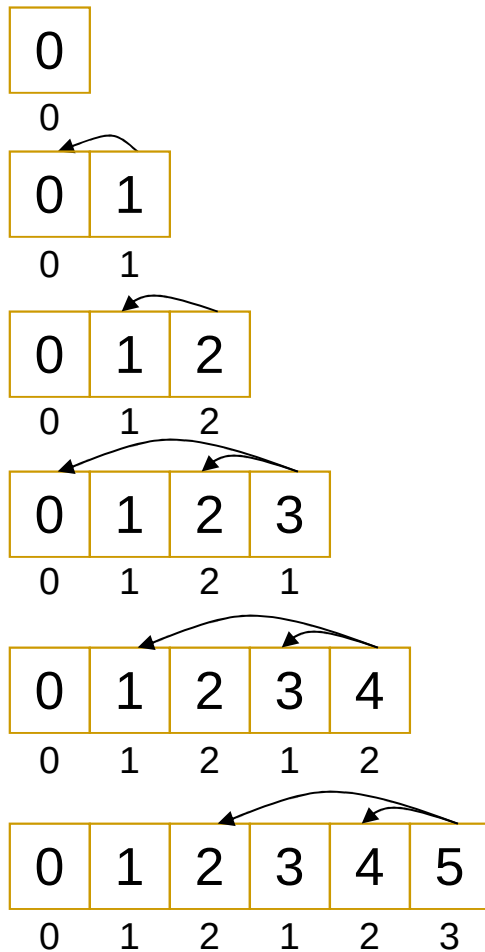
We Can Do Better

- We're re-computing values in our algorithm more than once
 - Save results of each computation for 0 to M
 - This way, we can do a reference call to find an already computed value, instead of re-computing each time
 - Running time $M*d$, where M is the value of money and d is the number of denominations
-

The Change Problem: Dynamic Programming

1. DPChange(M, c, d)
2. $bestNumCoins_0 \leftarrow 0$
3. for $m \leftarrow 1$ to M
4. $bestNumCoins_m \leftarrow \text{infinity}$
5. for $i \leftarrow 1$ to d
6. if $m \geq c_i$
7. if $bestNumCoins_{m - c_i} + 1 < bestNumCoins_m$
8. $bestNumCoins_m \leftarrow bestNumCoins_{m - c_i} + 1$
9. return $bestNumCoins_M$

DPChange: Example

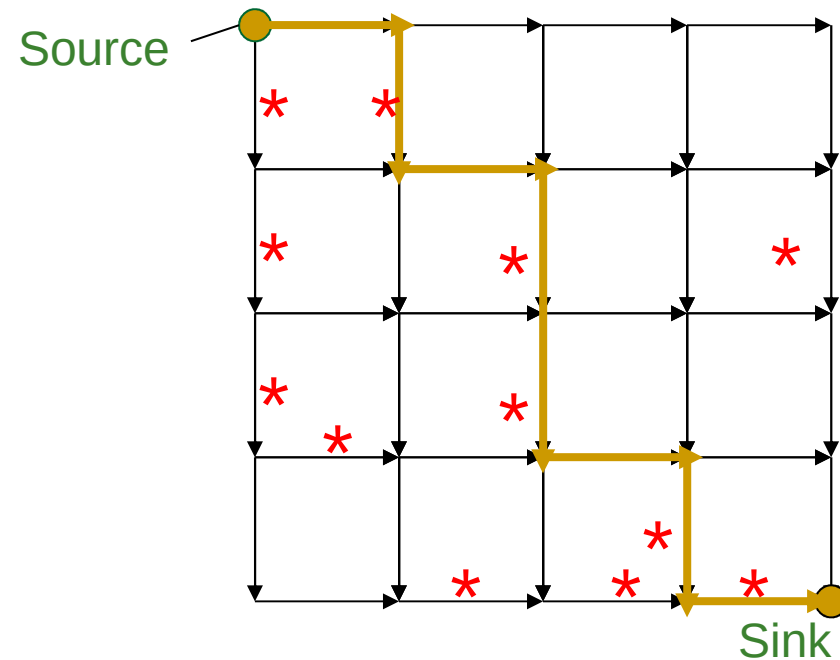


$$\mathbf{c} = (1, 3, 7)$$

$$M = 9$$

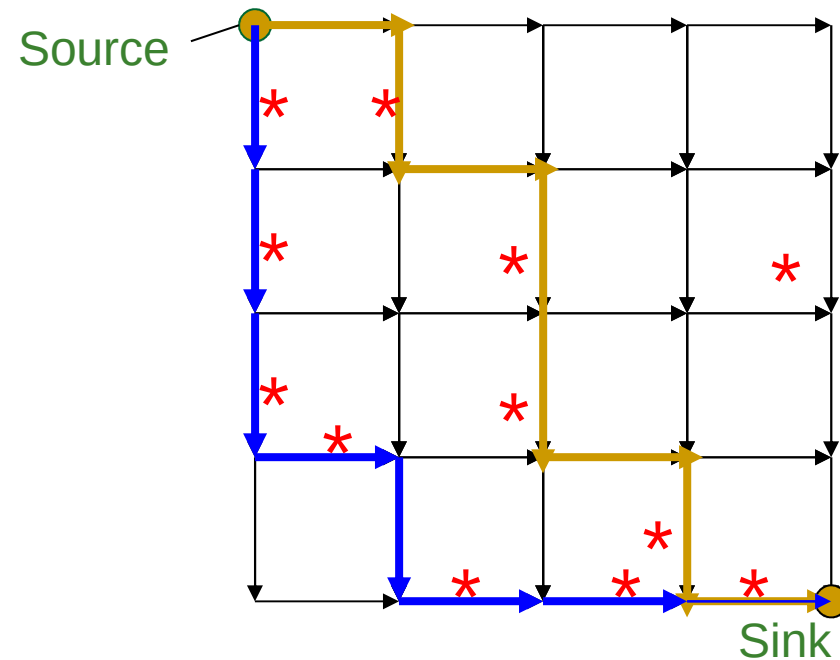
Manhattan Tourist Problem (MTP)

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 (MTP)

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



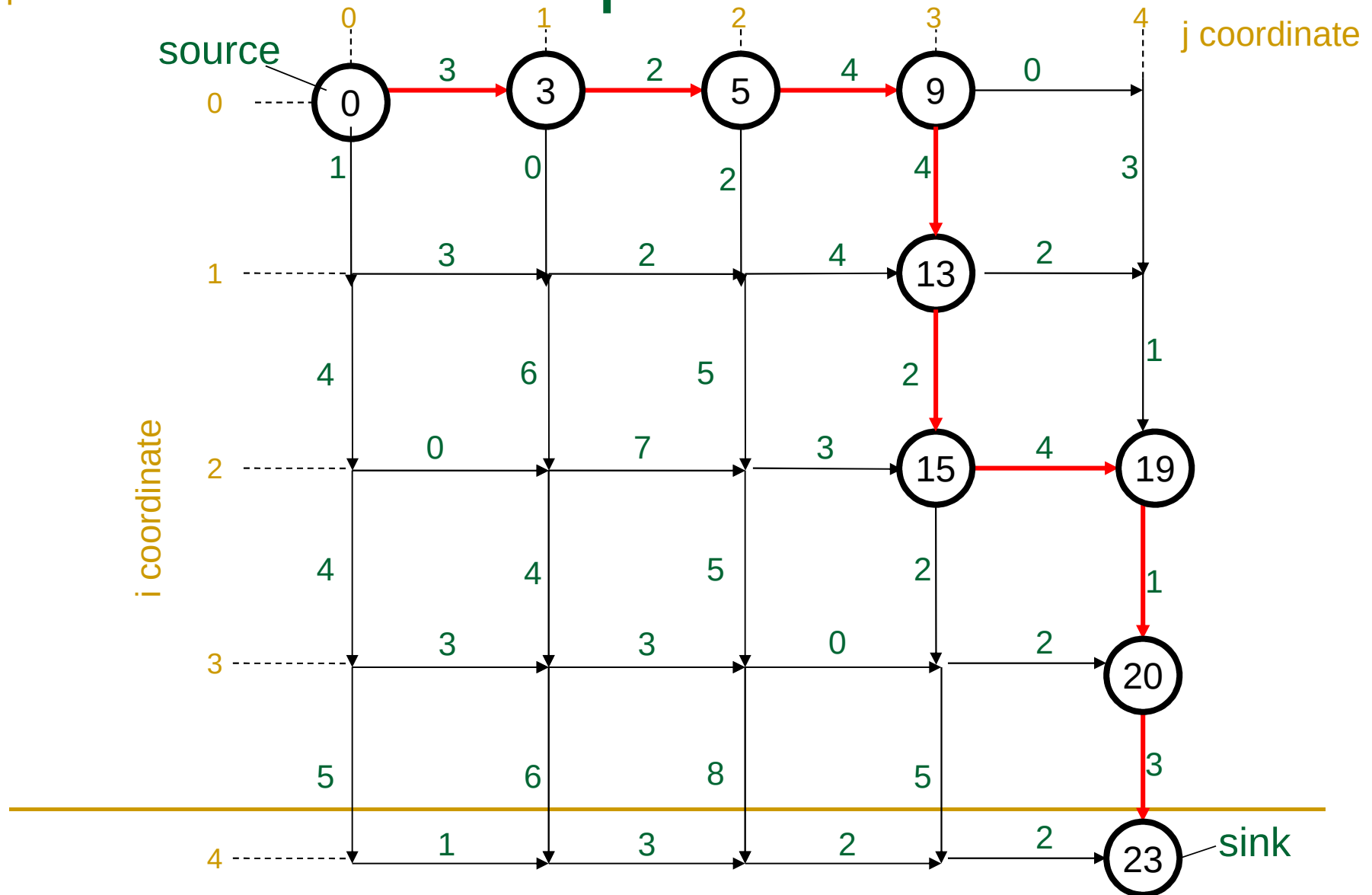
Manhattan Tourist Problem: Formulation

Goal: Find the longest path in a weighted grid.

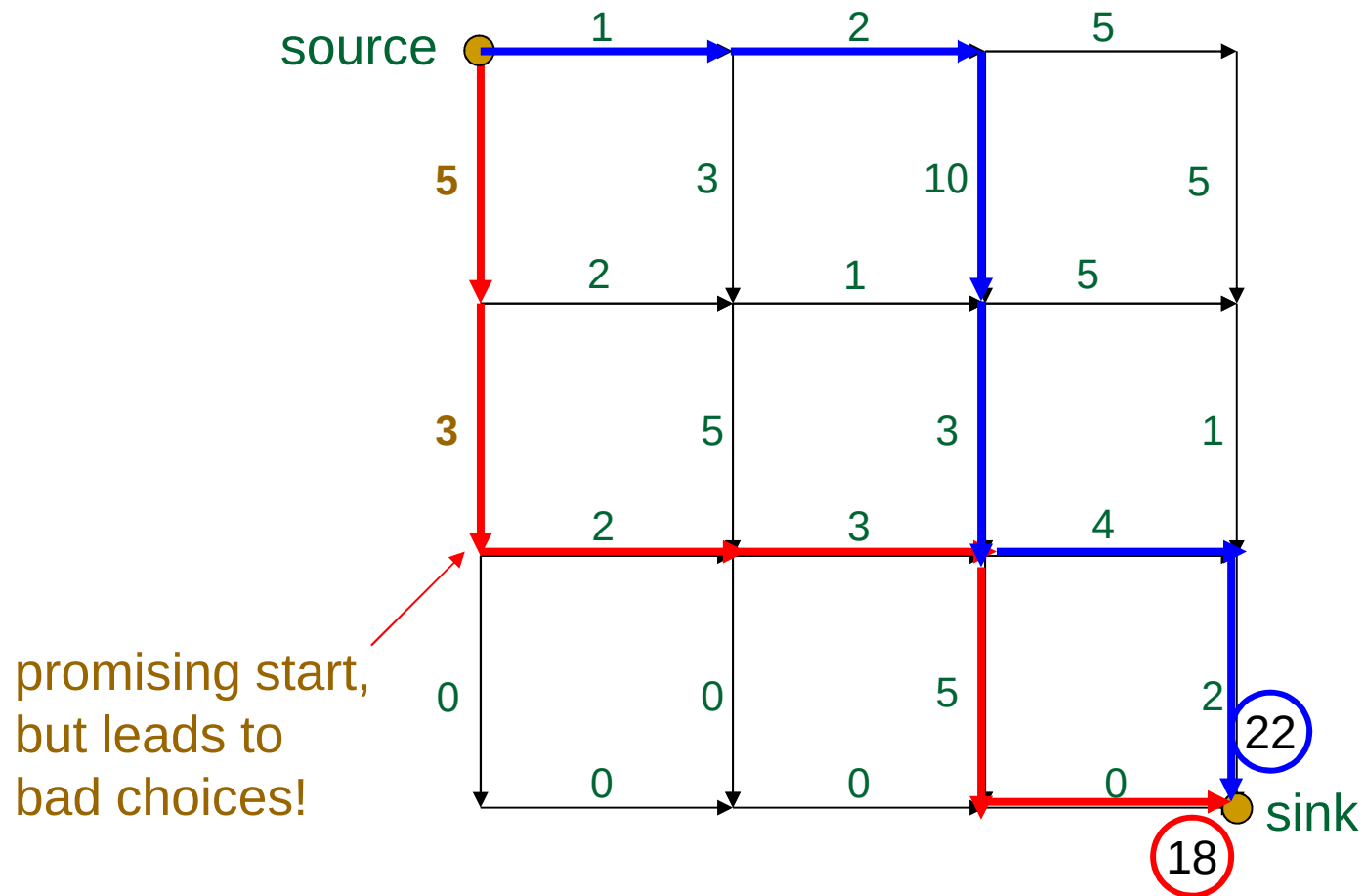
Input: A weighted grid $\mathbf{G}$ with two distinct vertices, one labeled “*source*” and the other labeled “*sink*”

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

MTP: An Example



MTP: Greedy Algorithm Is Not Optimal



MTP: Simple Recursive Program

$MT(n, m)$

if $n=0$ or $m=0$

return $MT(n, m)$

$x \leftarrow MT(n-1, m) +$

length of the edge from $(n-1, m)$ to (n, m)

$y \leftarrow MT(n, m-1) +$

length of the edge from $(n, m-1)$ to (n, m)

return $\max\{x, y\}$

MTP: Simple Recursive Program

$MT(n, m)$

$x \leftarrow MT(n-1, m) +$

length of the edge from $(n-1, m)$ to (n, m)

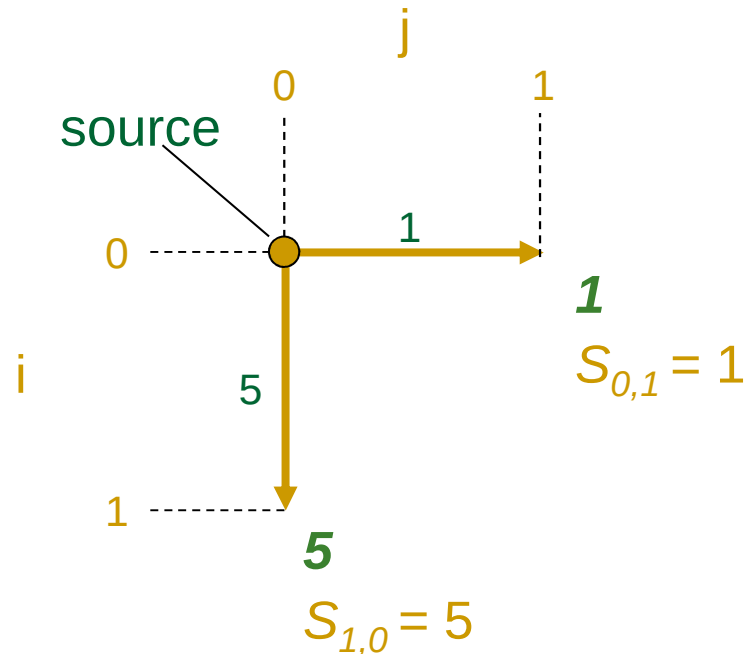
$y \leftarrow MT(n, m-1) +$

length of the edge from $(n, m-1)$ to (n, m)

return $\min\{x, y\}$

What's wrong with this approach?

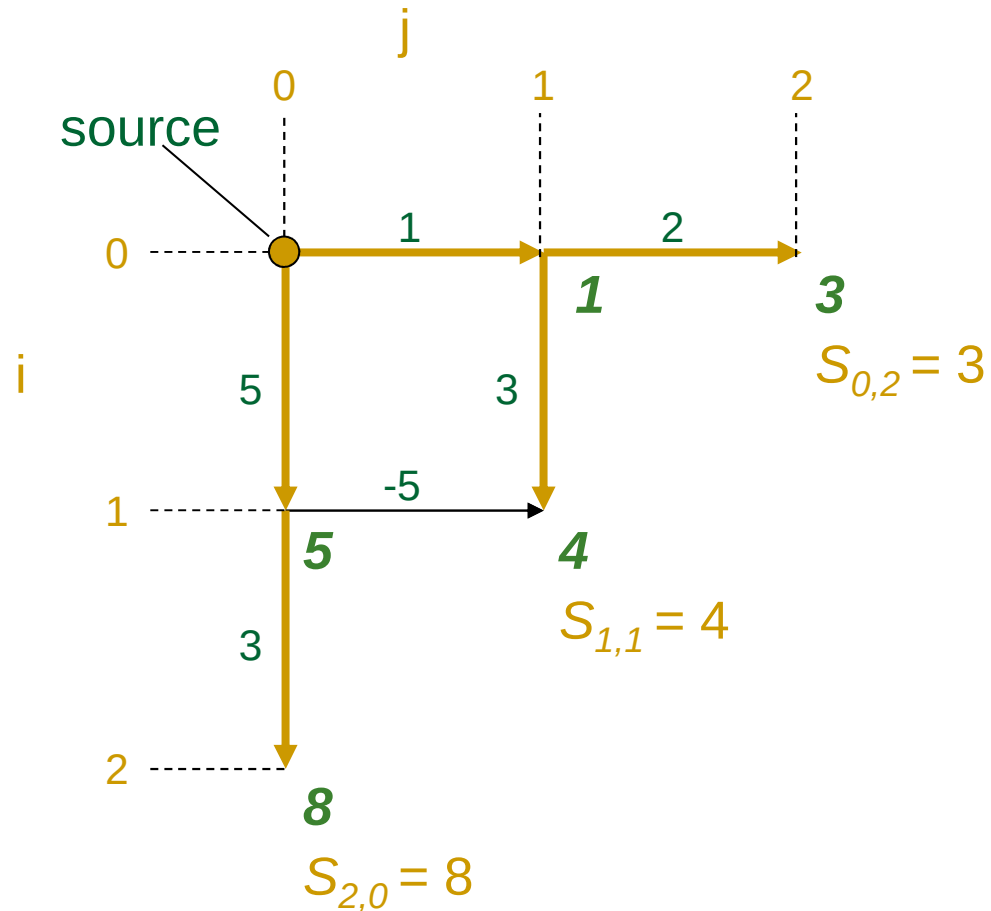
MTP: Dynamic Programming



- Calculate optimal path score for each vertex in the graph
- Each vertex's score is the maximum of the prior vertices score plus the weight of the respective edge in between

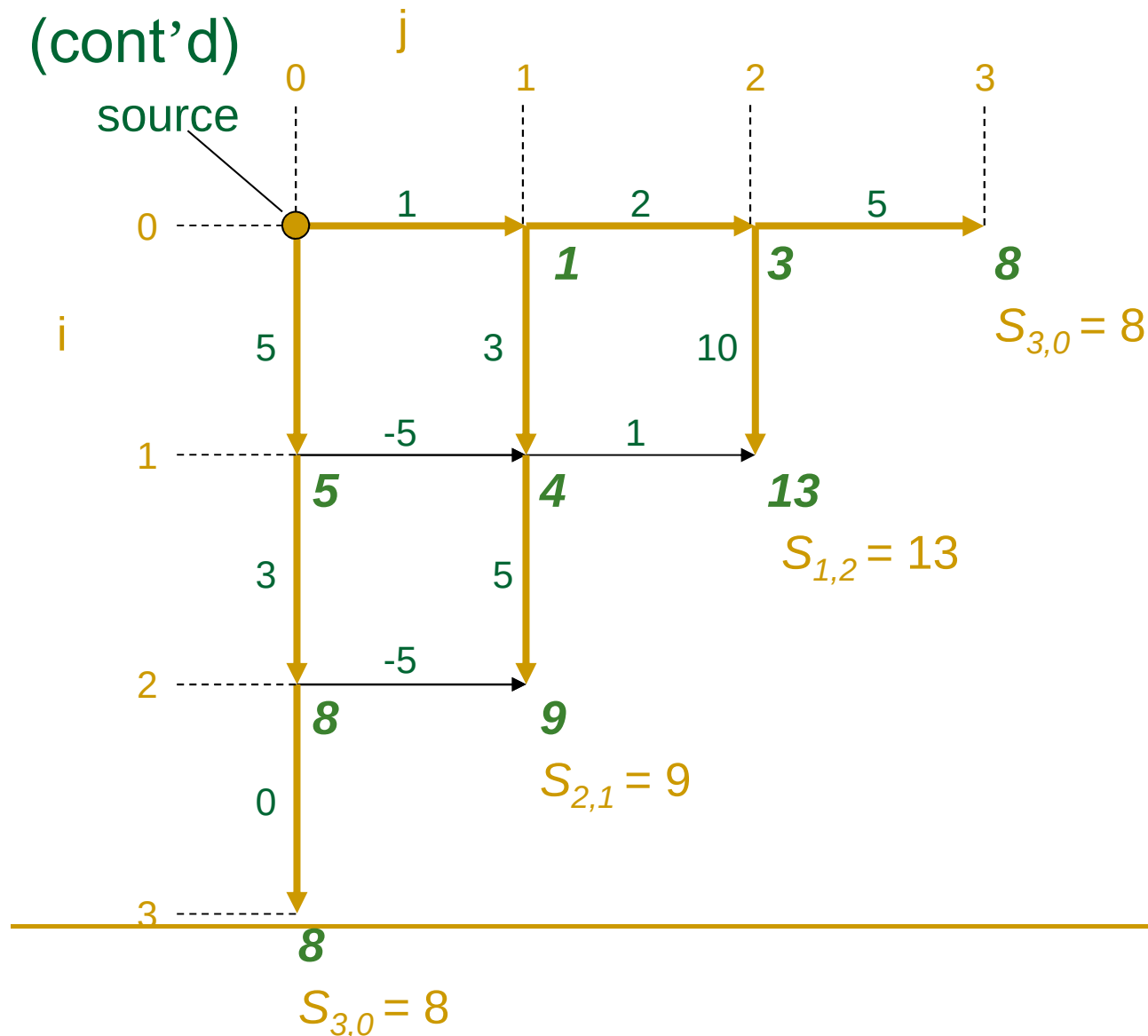
MTP: Dynamic Programming

(cont'd)

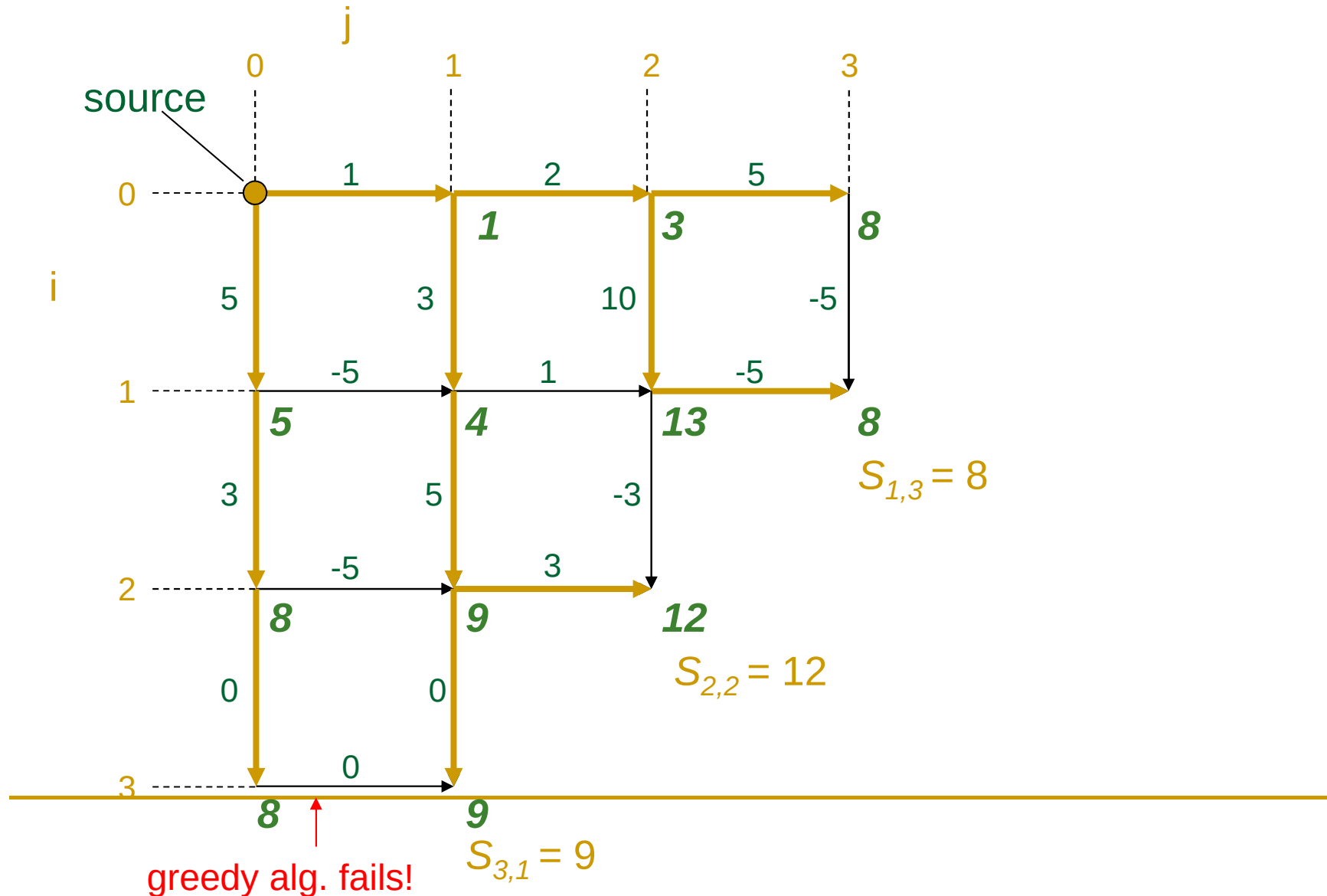


MTP: Dynamic Programming

(cont'd)

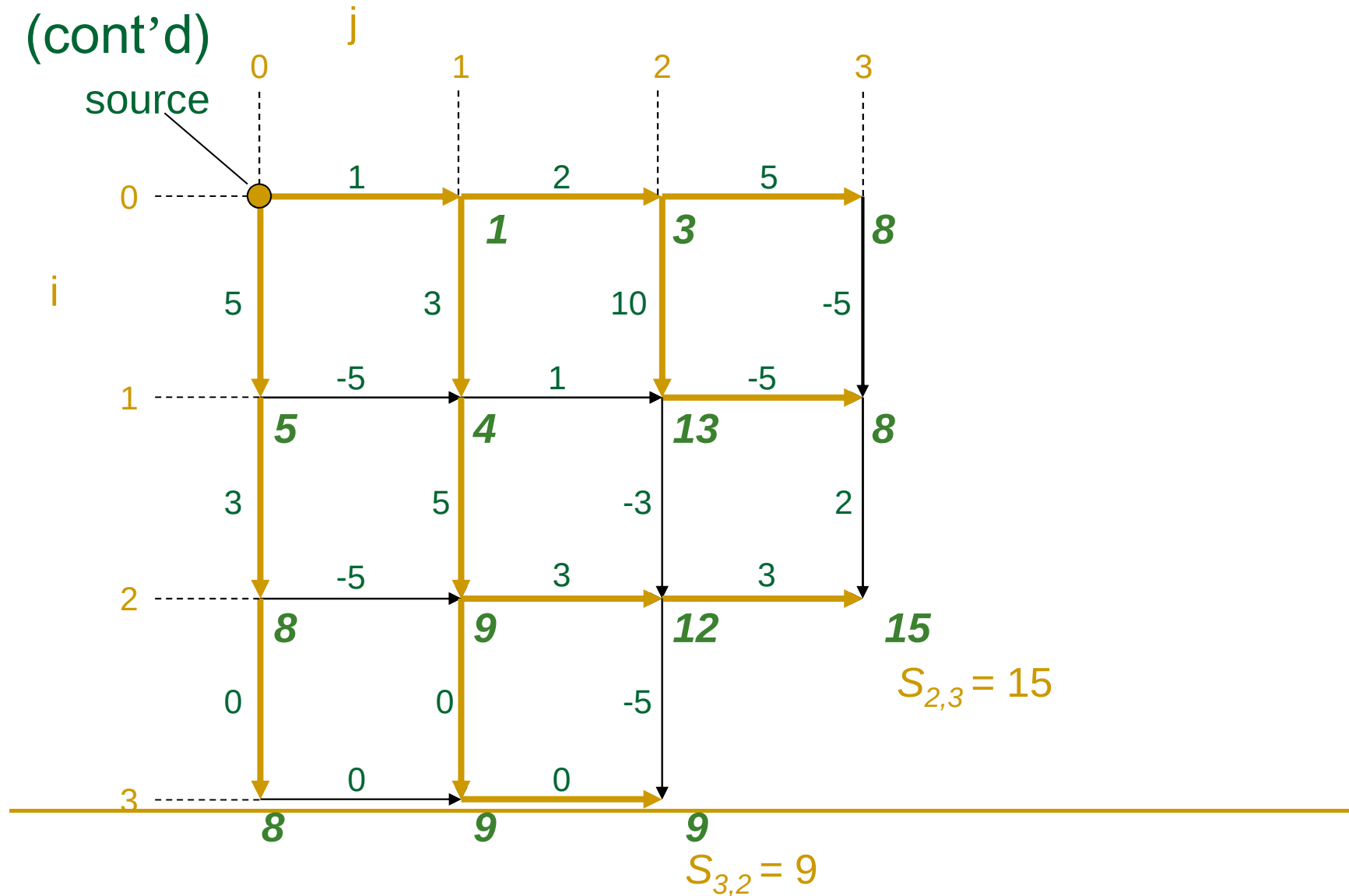


MTP: Dynamic Programming (cont'd)



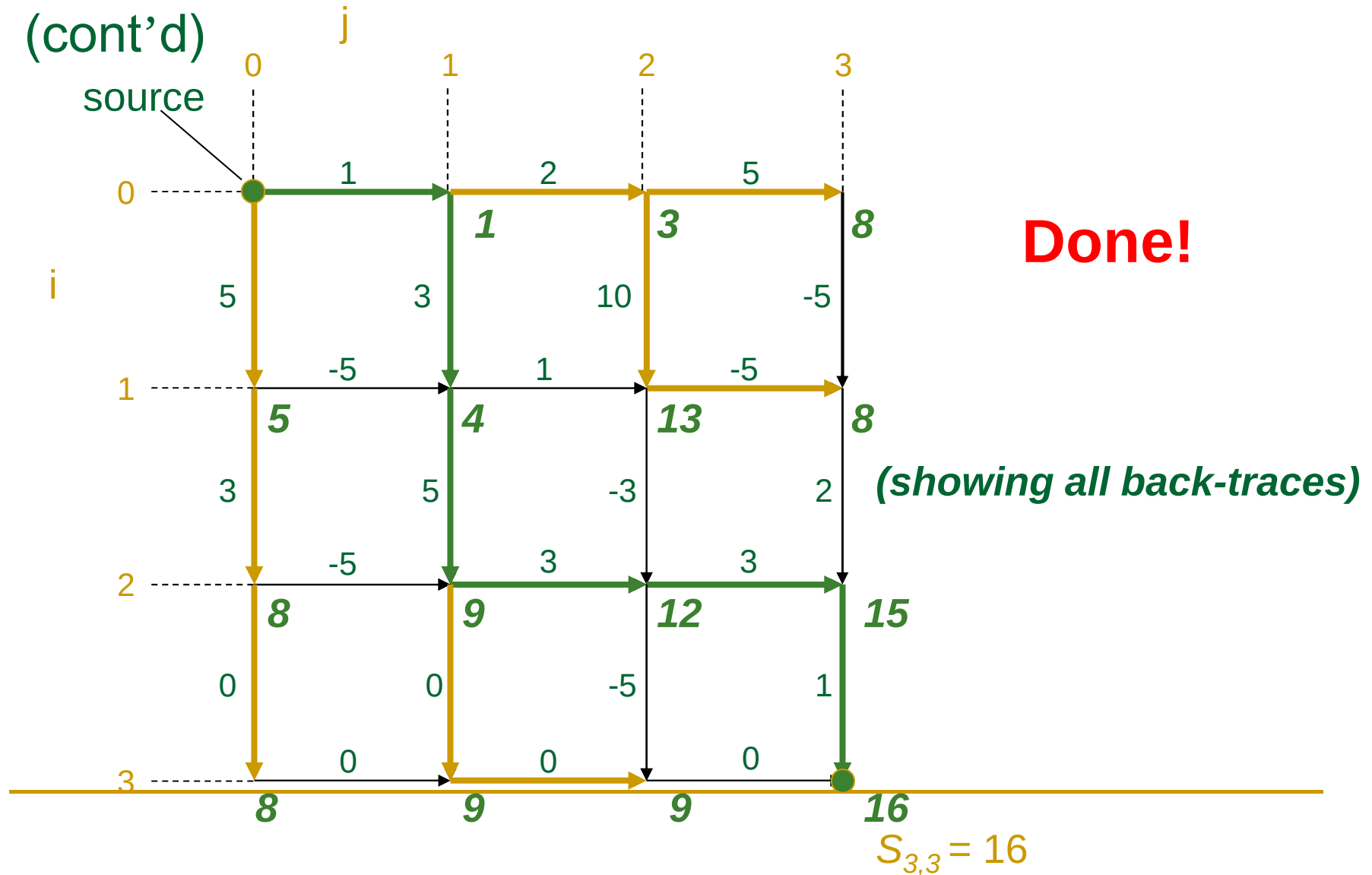
MTP: Dynamic Programming

(cont'd)



MTP: Dynamic Programming

(cont'd)



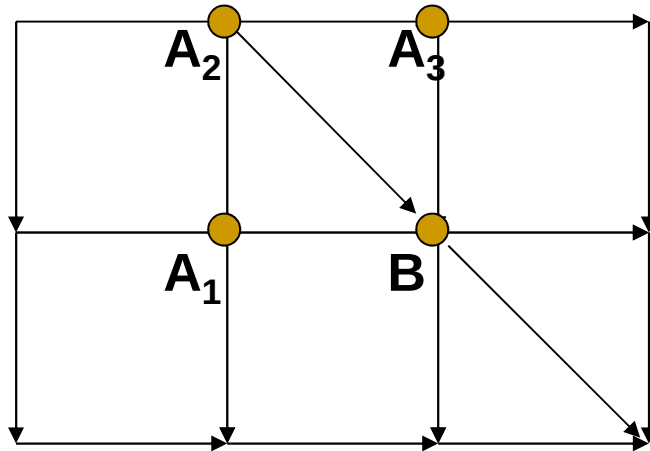
MTP: Recurrence

Computing the score 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:

$$s_B = \max \text{ of } \begin{cases} s_{A_1} + \text{weight of the edge } (A_1, B) \\ s_{A_2} + \text{weight of the edge } (A_2, B) \\ s_{A_3} + \text{weight of the edge } (A_3, B) \end{cases}$$

Manhattan Is Not A Perfect Grid (cont'd)

Computing the score for point x is given by the recurrence relation:

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

- Predecessors (x) – set of vertices that have edges leading to x
- The running time for a graph $G(\mathbf{V}, \mathbf{E})$ ($\mathbf{V}$ is the set of all vertices and $\mathbf{E}$ is the set of all edges) is $O(\mathbf{E})$ since each edge is evaluated once

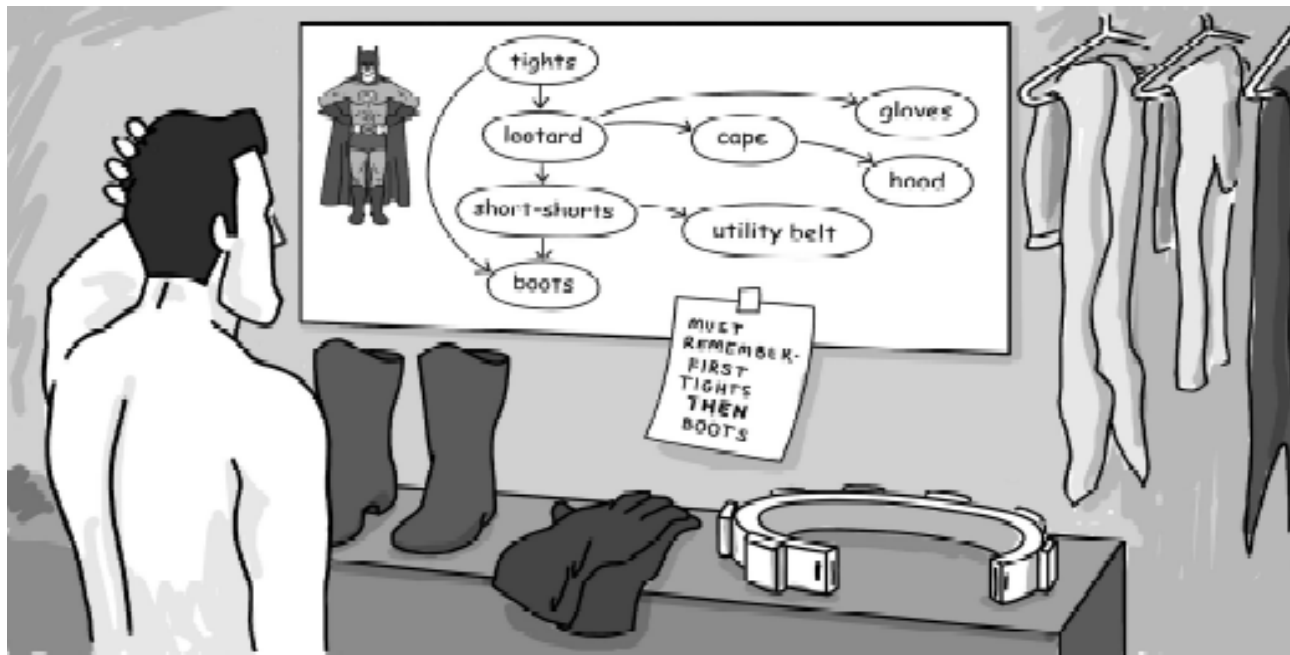
Traveling in the Grid

- The only hitch is that one must decide on the order in which to visit the vertices
- 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
- Try to find such order for a directed cycle

???

DAG: Directed Acyclic Graph

- Since Manhattan is not a perfect regular grid, we represent it as a DAG
- DAG for *Dressing in the morning* problem

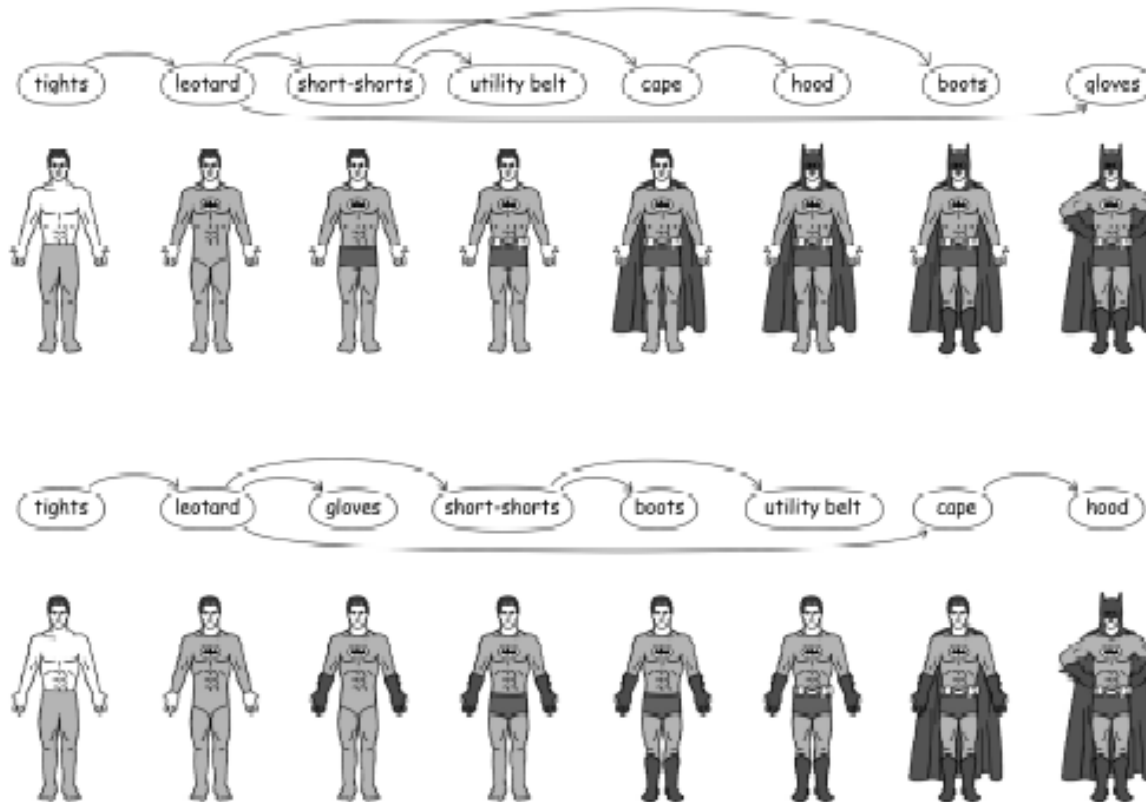


Topological Ordering

- 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.
-

Topological ordering

- 2 different topological orderings of the DAG



Longest Path in DAG Problem

- Goal: 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 DAG: Dynamic Programming

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

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

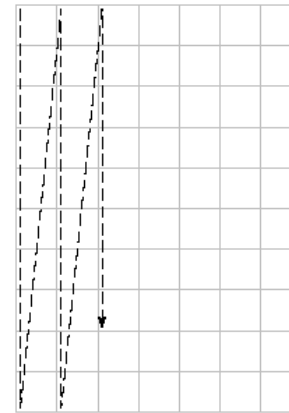
In General:

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

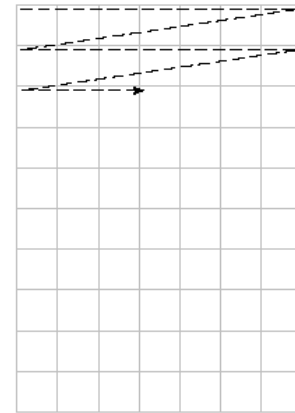
Traversing the Manhattan Grid

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

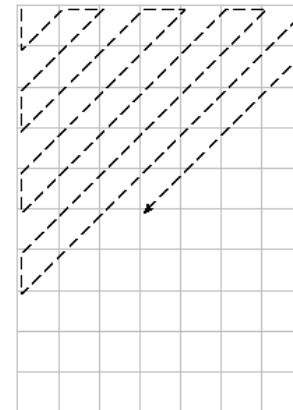
a)



b)



c)



Alignment: 2 row representation

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

v : A T C T G A T $m = 7$
w : T G C A T A $n = 6$

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

letters of **v**

letters of **w**

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

4 matches

2 insertions

2 deletions

Aligning DNA Sequences

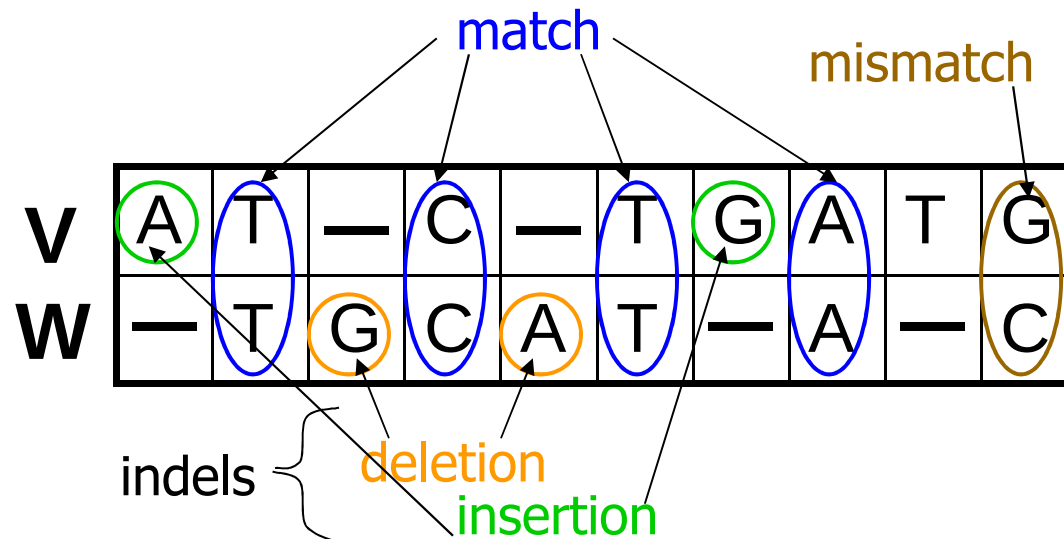
V = ATCTGATG

n = 8

W = TGCATAC

m = 7

4 matches
1 mismatches
2 insertions
2 deletions



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

LCS: Dynamic Programming

- Find the LCS of two strings

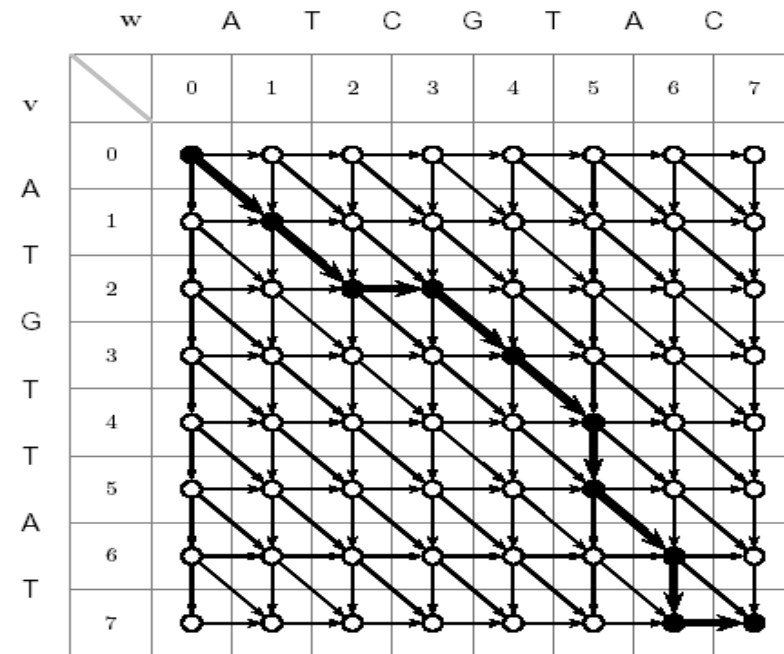
Input: A weighted graph G with two distinct vertices, one labeled “*source*” one labeled “*sink*”

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

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

```

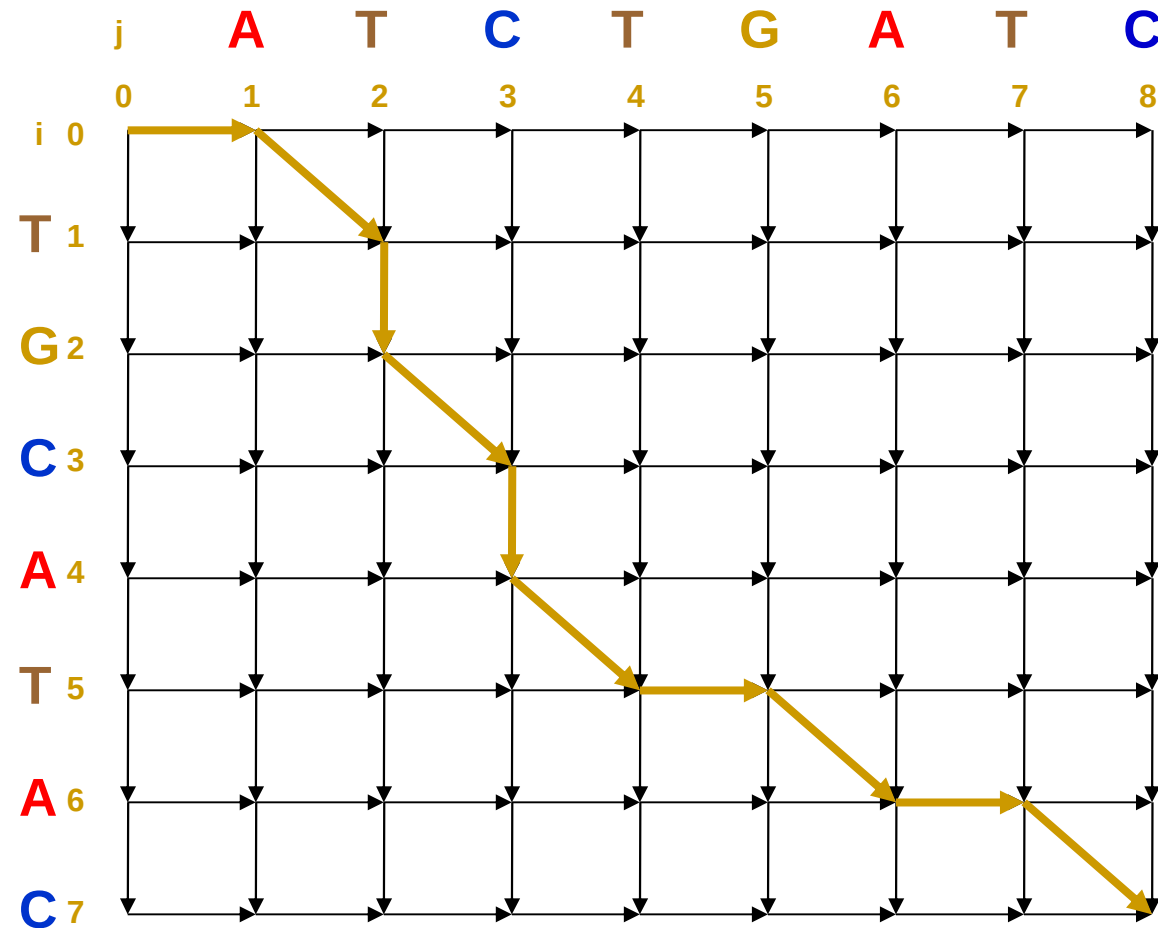
      0  1  2  2  3  4  5  6  7  7
v  =  A  T  -  G  T  T  A  T  -
      |  |  |  |  |  |  |  |
w  =  A  T  C  G  T  -  A  -  C
      0  1  2  3  4  5  5  6  6  7
  
```



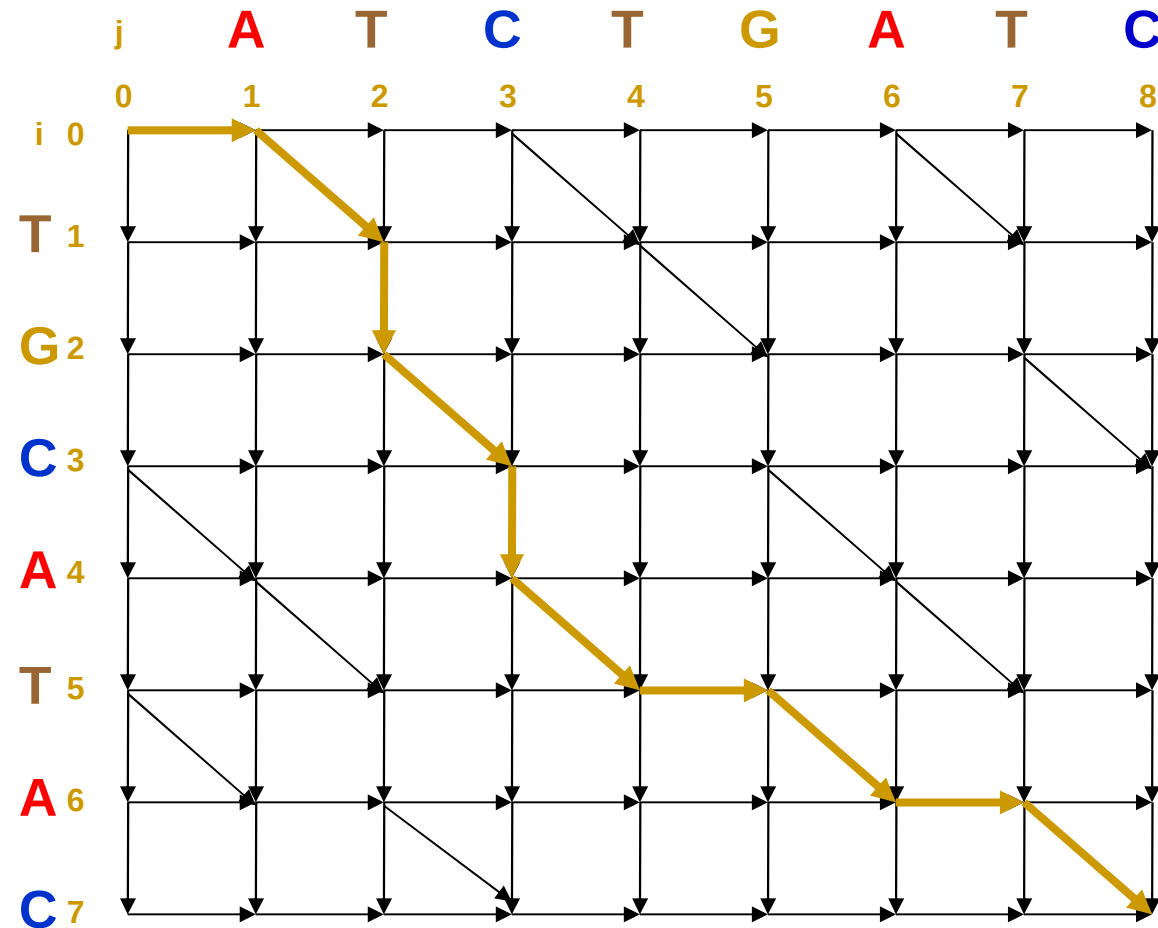
```

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

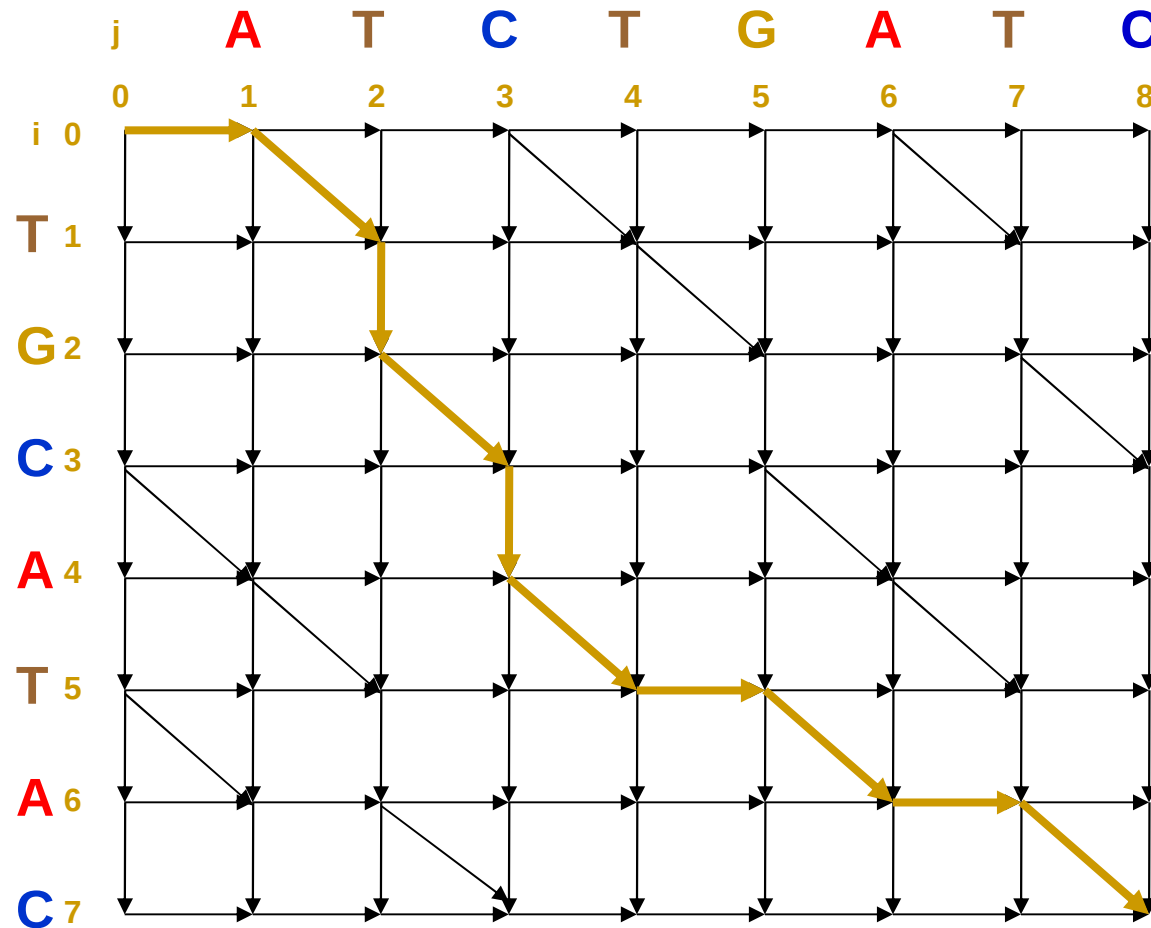
LCS Problem as Manhattan Tourist Problem



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

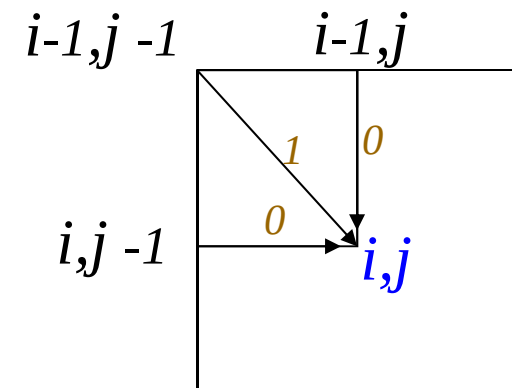
and $\mathbf{w}_j$ = prefix of $\mathbf{w}$ of length j : $w_1 \dots w_j$

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

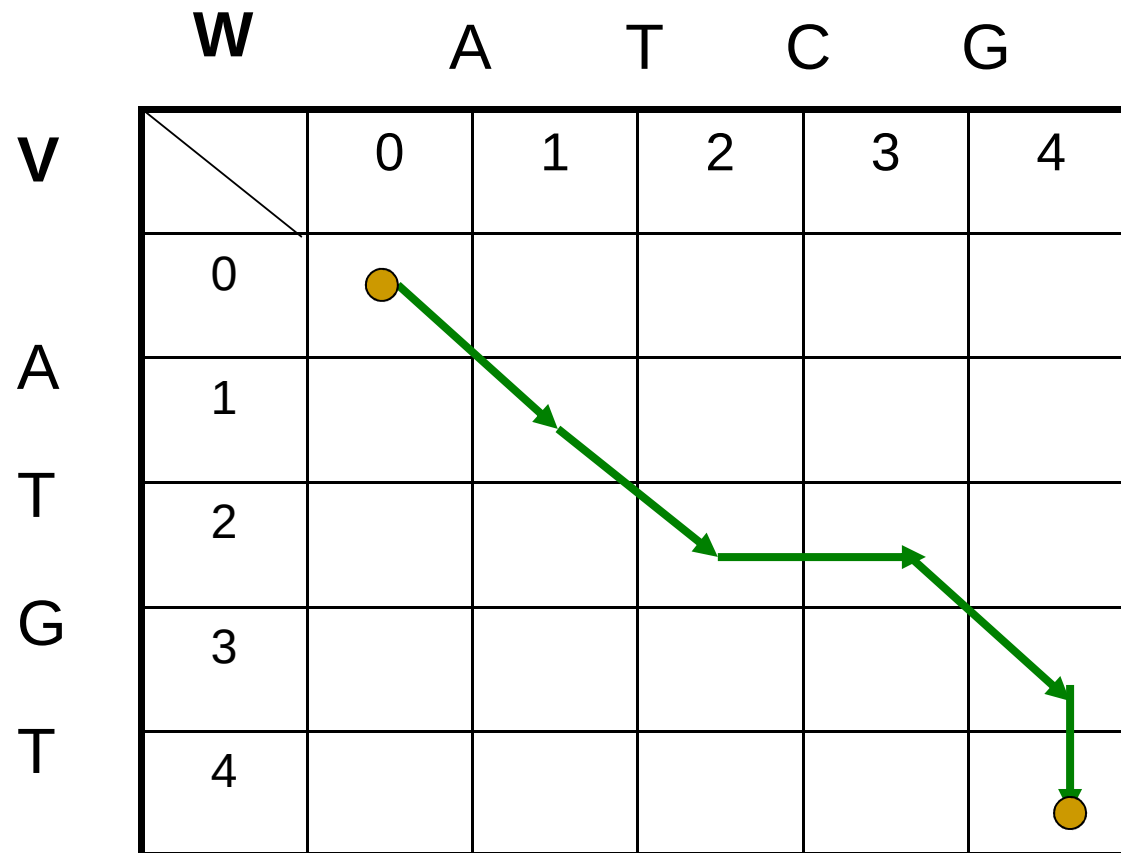
$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \text{ if } v_i = w_j \end{cases}$$

Computing LCS (cont'd)

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



Every Path in the Grid Corresponds to an Alignment



↘ ↘ → ↘ ↘
 0 1 2 2 3 4
 V = A T - G T
 | | |
 W = A T C G -
 0 1 2 3 4 4

Aligning Sequences without Insertions and Deletions: Hamming Distance

Given two DNA sequences $\mathbf{v}$ and $\mathbf{w}$:

$\mathbf{v}$: A T A T A T A T

$\mathbf{w}$: T A T A T A T A

- 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 $\mathbf{v}$ with

i -th letter of $\mathbf{w}$

$\mathbf{v}$	=	A	T	A	T	A	T
$\mathbf{w}$	=	T	A	T	A	T	A

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 $\mathbf{v}$ with
 i -th letter of $\mathbf{w}$

$\mathbf{v} = \text{ATATATAT}$
| | | | | | | |
 $\mathbf{w} = \text{TATATATA}$

Just one shift
— — — — — — — —
Make it all line up →

Hamming distance:

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

Computing Hamming distance
is a **trivial** task

Edit distance
may compare

i -th letter of $\mathbf{v}$ with
 j -th letter of $\mathbf{w}$

$\mathbf{v} = -\text{ATATATAT}$
| | | | | | | |
 $\mathbf{w} = \text{TATATATA}$

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 $\mathbf{v}$ with
 i -th letter of $\mathbf{w}$

$\mathbf{v} = \text{ATATATAT}$
 | | | | |
 $\mathbf{w} = \text{TATATATA}$

Hamming distance:

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

Edit distance
may compare

i -th letter of $\mathbf{v}$ with
 j -th letter of $\mathbf{w}$

$\mathbf{v} = - \text{ATATATAT}$
 | | | | |
 $\mathbf{w} = \text{TATATATA}$

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^T → (delete last ^T)

TGCAT^A → (delete last ^A)

TGCAT → (insert ^A at front)

^AT^GCAT → (substitute ^C for 3rd ^G)

AT^CCAT → (insert ^G before last A)

ATCC^GAT (Done)

Edit Distance: Example

TGCATAT → ATCCGAT in 5 steps

TGCATAT^T → (delete last ^T)

TGCAT^A → (delete last ^A)

TGCAT → (insert ^A at front)

^AT^GCAT → (substitute ^C for 3rd ^G)

AT^CCAT → (insert ^G before last A)

ATCC^GAT (Done)

What is the edit distance? 5?

Edit Distance: Example (cont'd)

TGCATAT → ATCCGAT in 4 steps

TGCATAT → (insert **A** at front)

ATGCATAT**T** → (delete 6th **T**)

ATGC**A**TA → (substitute **G** for 5th **A**)

AT**G**CGTA → (substitute **C** for 3rd **G**)

AT**C**CGAT (Done)

Edit Distance: Example (cont'd)

TGCATAT → ATCCGAT in 4 steps

TGCATAT → (insert **A** at front)

ATGCATAT**T** → (delete 6th **T**)

ATGC**A**TA → (substitute **G** for 5th **A**)

AT**G**CGTA → (substitute **C** for 3rd **G**)

AT**C**CGAT (Done)

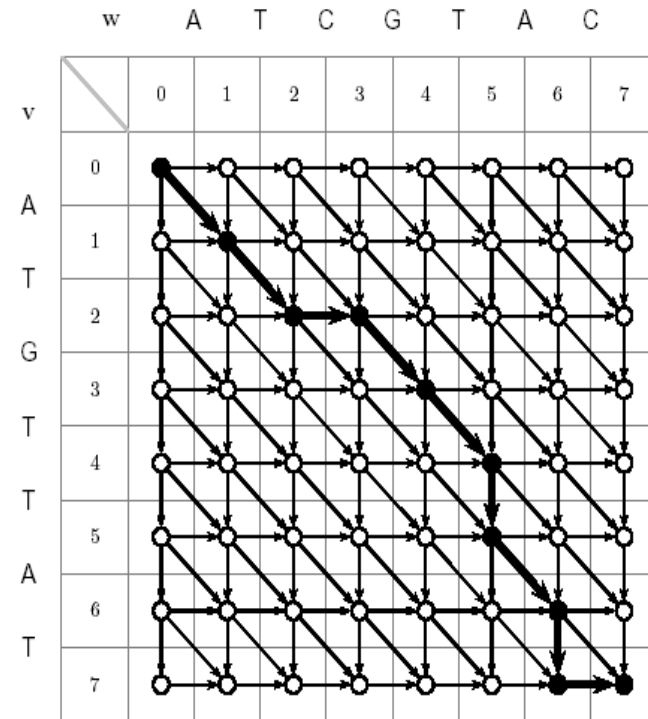
Can it be done in 3 steps???

The Alignment Grid

- Every alignment path is from source to sink

```

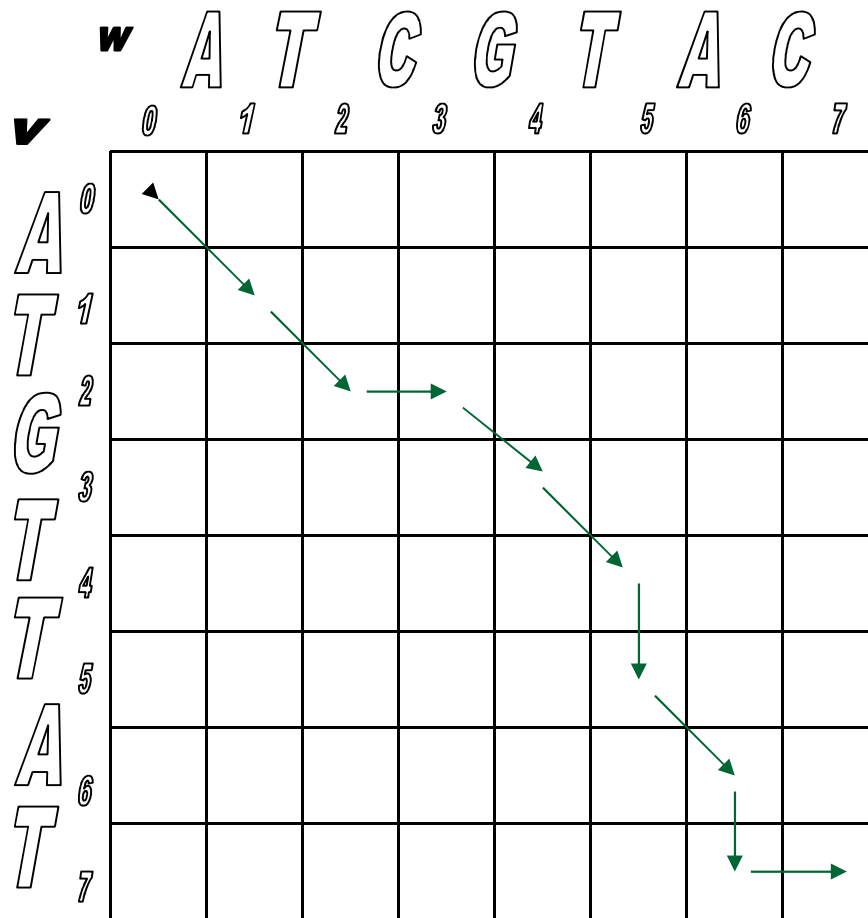
      0  1  2  2  3  4  5  6  7  7
v  =   A  T  -  G  T  T  A  T  -
      |  |  |  |  |  |  |  |
w  =   A  T  C  G  T  -  A  -  C
      0  1  2  3  4  5  5  6  6  7
  
```



```

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

Alignment as a Path in the Edit Graph

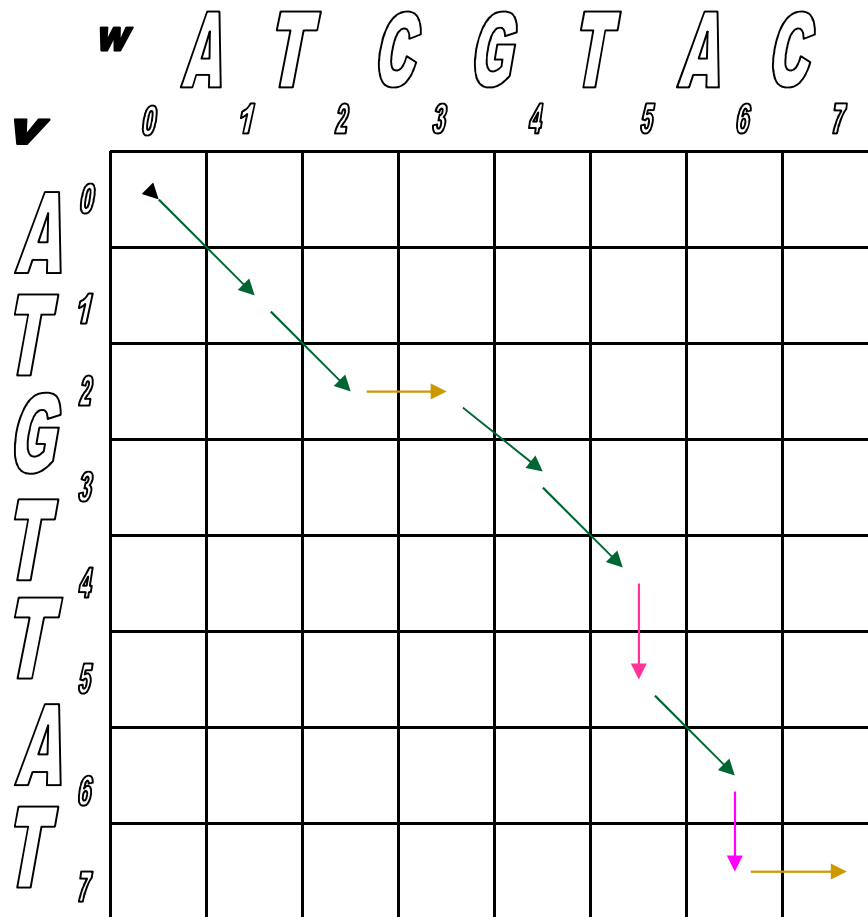


0	1	2	2	3	4	5	6	7	7
	A	T	_	G	T	T	A	T	_
	A	T	C	G	T	_	A	_	C
0	1	2	3	4	5	5	6	6	7

- 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)

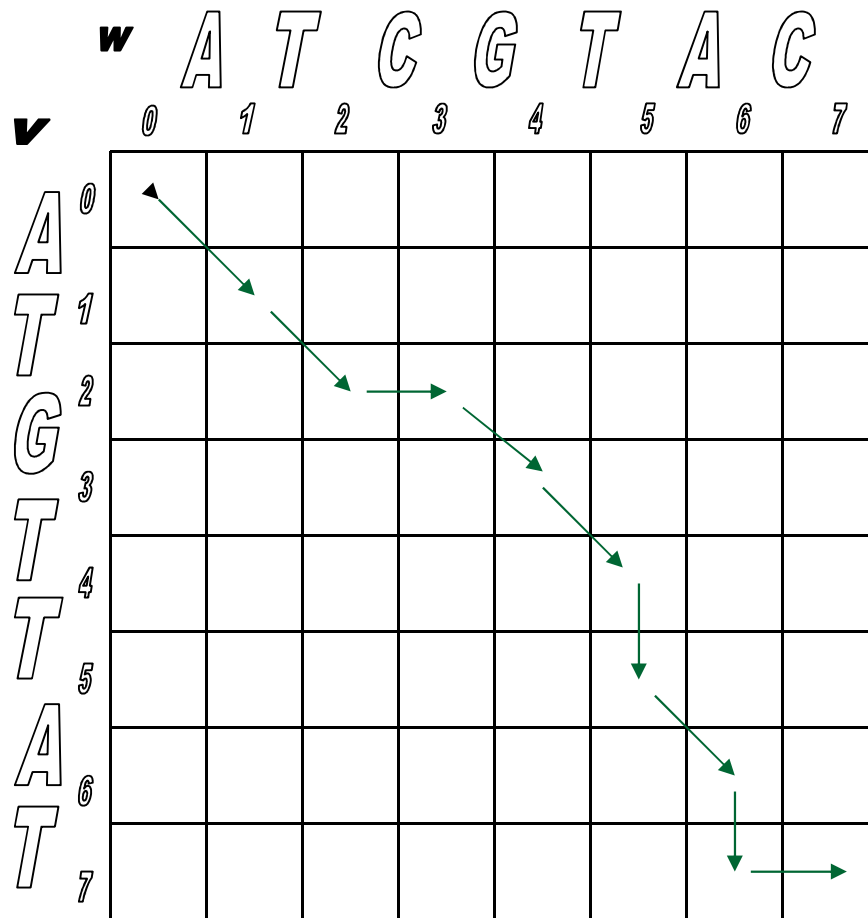


↓ and → represent indels in v and w with score 0.

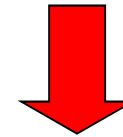
↘ represent matches with score 1.

- The score of the alignment path is 5.

Alignment as a Path in the Edit Graph

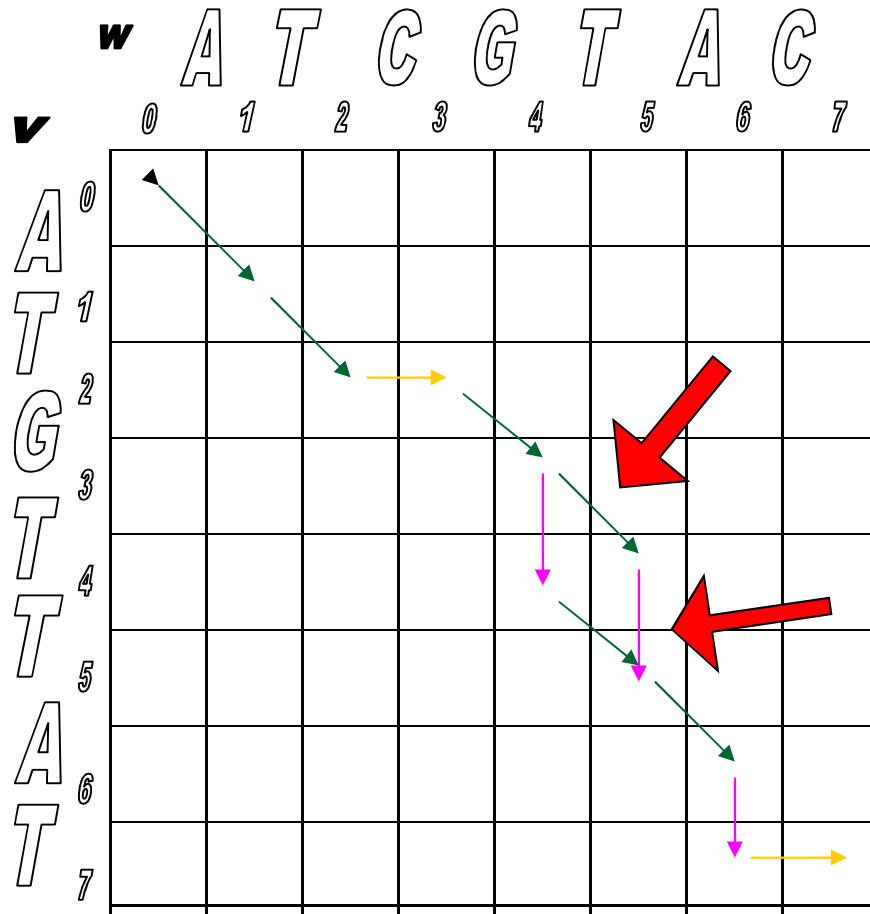


Every path in the edit graph corresponds to an alignment:



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

Alignment as a Path in the Edit Graph



Old Alignment

0122345677

v= AT_GTTAT_

w= ATCGT_A_C

0123455667

New Alignment

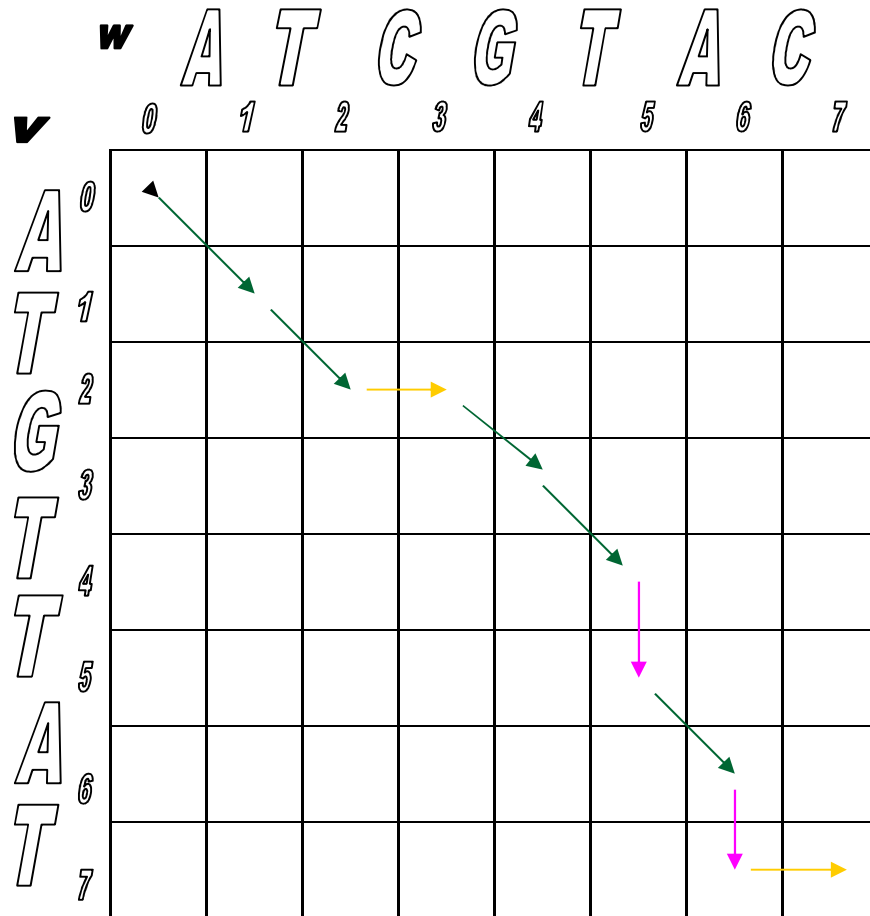
0122345677

v= AT_GTTAT_

w= ATCG_TA_C

0123445667

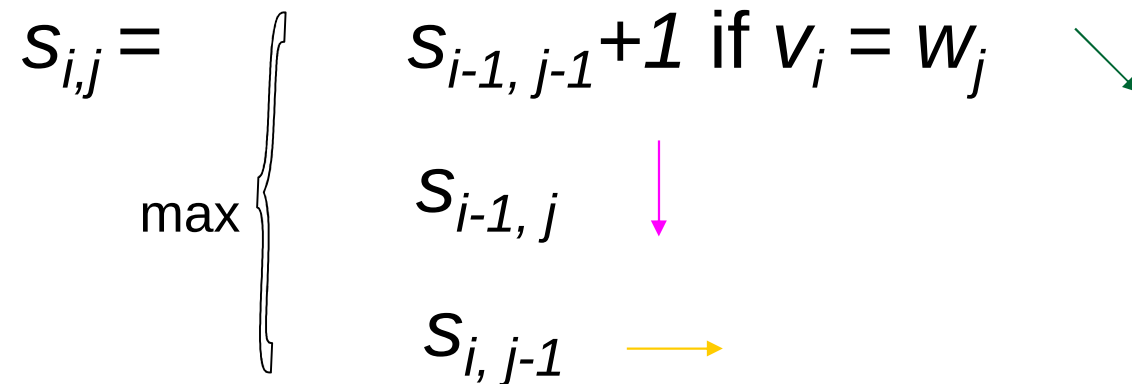
Alignment as a Path in the Edit Graph



01234567
 v= AT_GTTAT_
 w= ATCGT_A_C
 01234567

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

Alignment: Dynamic Programming

$$s_{i,j} = \max \left\{ \begin{array}{l} s_{i-1,j-1} + 1 \text{ if } v_i = w_j \\ s_{i-1,j} \\ s_{i,j-1} \end{array} \right.$$


Dynamic Programming Example

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

Initialize 1^{st} row and 1^{st} column to be all zeroes.

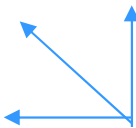
Or, to be more precise, initialize 0^{th} row and 0^{th} column to be all zeroes.

Dynamic Programming Example

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

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

Alignment: Backtracking

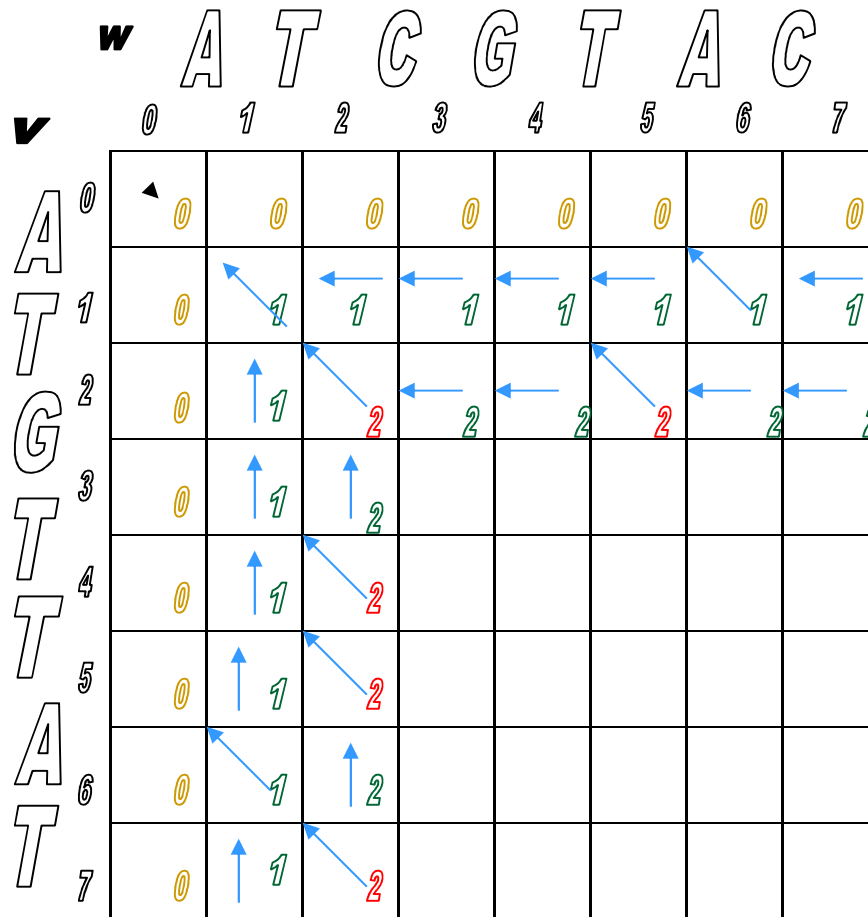
Arrows  show where the score originated from.

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

$$s_{2,2} = [s_{1,1} = 1] + 1$$

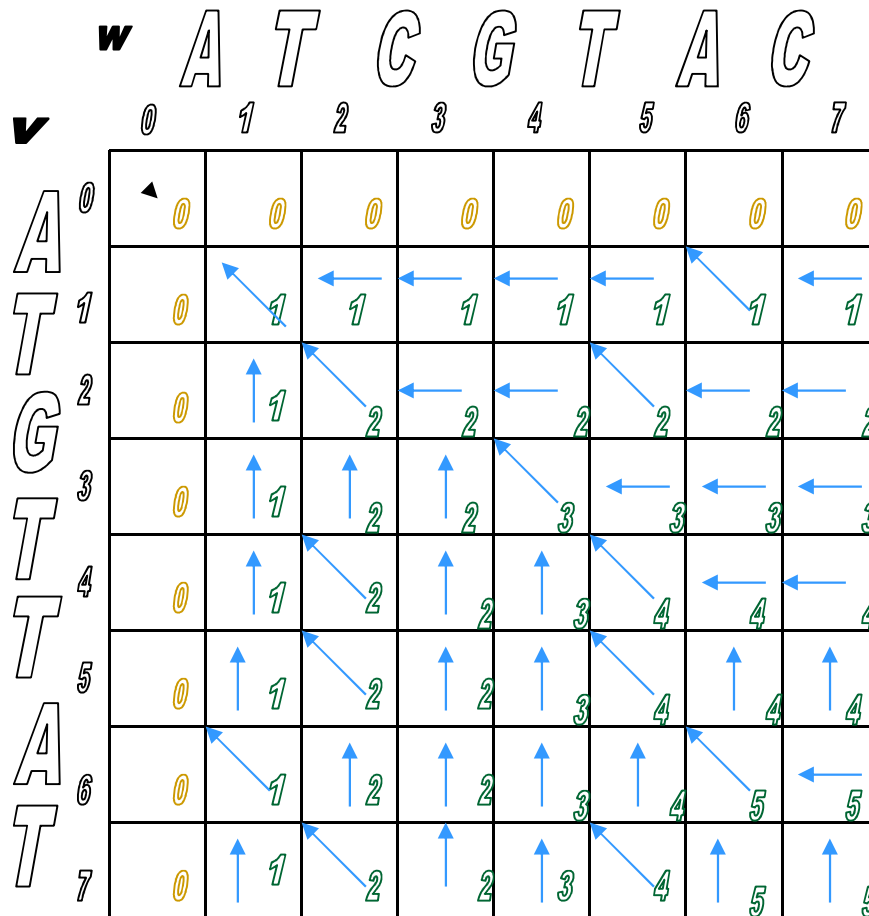
$$s_{2,5} = [s_{1,4} = 1] + 1$$

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

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

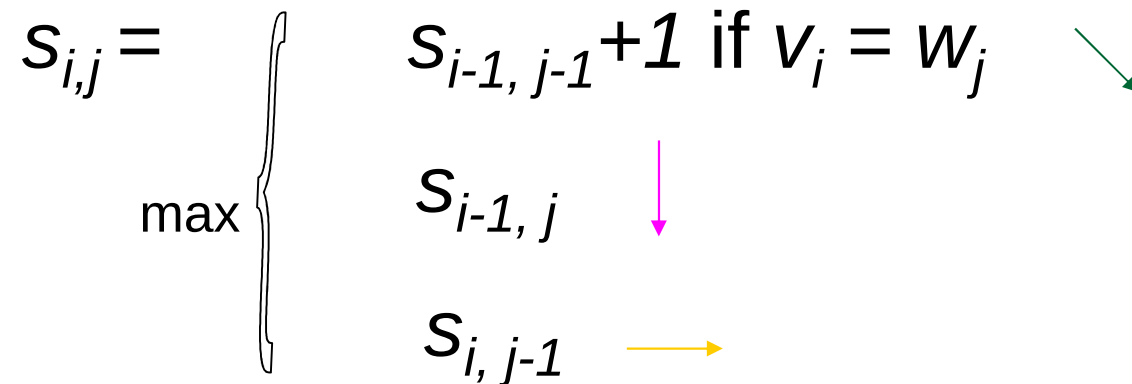
$$s_{7,2} = [s_{6,1} = 1] + 1$$

Backtracking Example



Continuing with the dynamic programming algorithm gives this result.

Alignment: Dynamic Programming

$$s_{i,j} = \max \left\{ \begin{array}{l} s_{i-1,j-1} + 1 \text{ if } v_i = w_j \\ s_{i-1,j} \\ s_{i,j-1} \end{array} \right.$$


Alignment: Dynamic Programming

$$s_{i,j} = \max \left\{ \begin{array}{l} s_{i-1,j-1} + 1 \text{ if } v_i = w_j \quad \searrow \\ s_{i-1,j} + 0 \quad \downarrow \\ s_{i,j-1} + 0 \quad \rightarrow \end{array} \right.$$

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

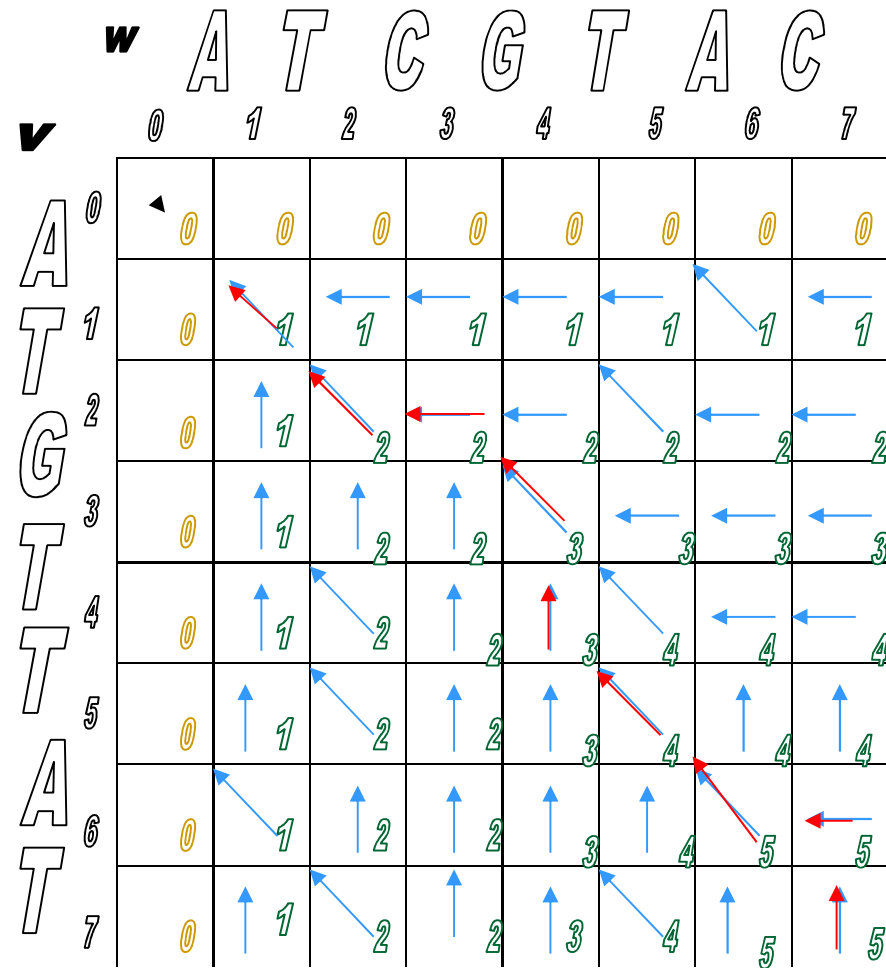
```

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

```

Now What?

- $\text{LCS}(v, w)$ 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 $n \times m$ dynamic programming matrix.
 - Why $O(nm)$? The pseudocode consists of a nested “for” loop inside of another “for” loop to set up a $n \times m$ matrix.
-