Dynamic Programming: Edit Distance

Outline

1. DNA Sequence Comparison and CF
2. Change Problem
3. Manhattan Tourist Problem
4. Longest Paths in Graphs
5. Sequence Alignment
6. Edit Distance

Section 1: DNA Sequence Comparison and CF

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 a cancer-causing gene and the normal growth factor (PDGF) gene.

Russell Doolittle

http://biology.ucsd.edu/faculty/doolittle.html

Cystic Fibrosis

- **Cystic fibrosis (CF)**: A chronic and frequently fatal disease which produces an abnormally large amount of mucus.
  - Mucus is a slimy material that coats many epithelial surfaces and is secreted into fluids such as saliva.

- CF primarily affects the respiratory systems of children.

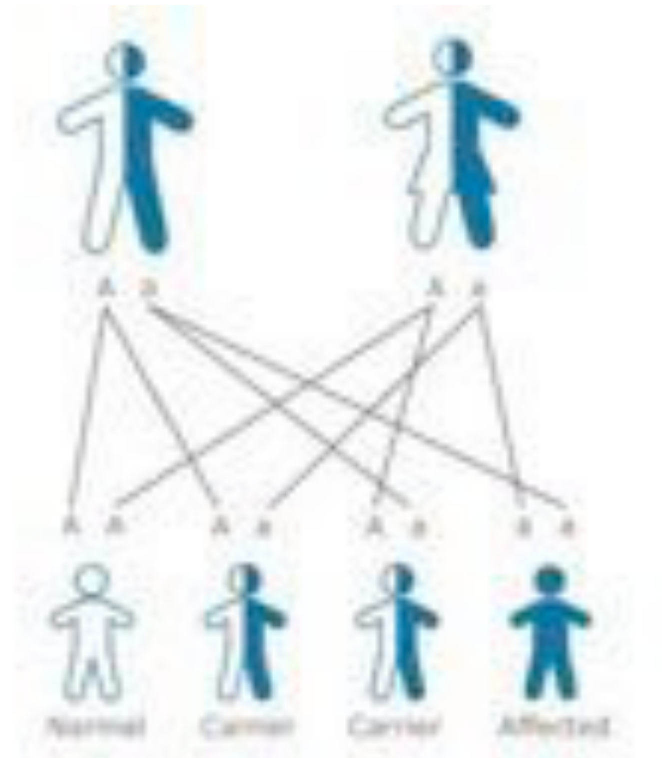
http://www.eradimaging.com/site/article.cfm?ID=327

Cystic Fibrosis: Inheritance

- In the early 1980s biologists hypothesized that CF is a genetic disorder caused by mutations in an unidentified gene.

- Heterozygous carriers are asymptomatic.

- Therefore a person must be homozygously recessive in the CF gene in order to be diagnosed with CF.



http://www.discern-genetics.org/diagram61.php

Cystic Fibrosis: Connection to Other Proteins

- **Adenosine Triphosphate (ATP)**: The energy source of all cell processes.

- ATP binding proteins are present on the cell membrane and act as transport channels.

- In 1989, biologists found a similarity between the cystic fibrosis gene and ATP binding proteins.

- This connection was plausible, given the fact that CF involves sweet secretion with abnormally high sodium levels.

Cystic Fibrosis: Mutation Analysis

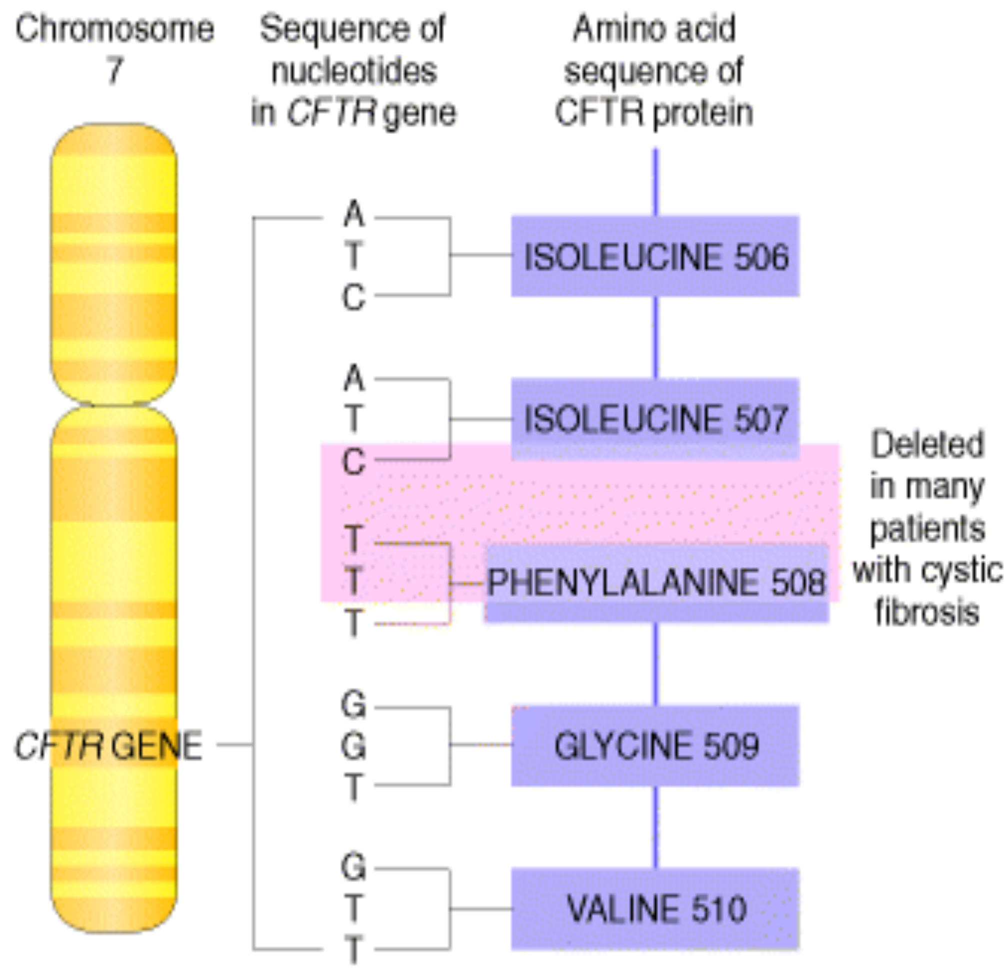
- If a high percentage of CF patients have a given mutation in the gene and the normal patients do not, then this could be an indicator of a mutation related to CF.

- 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 the CFTR Protein



Section 2: The Change Problem

Bring in the Bioinformaticians

- Similarities between a gene with known function and a gene with unknown function allow biologists to infer the function of the gene with unknown function.

- We would like to compute a similarity score between two genes to tell how likely it is that they have similar functions.

- Dynamic programming is a computing technique for revealing similarities between sequences.

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

Motivating Example: The Change Problem

- Say we want to provide change totaling 97 cents.
- We could do this in a large number of ways, but the quickest way to do it would be:
  - Three quarters = 75 cents
  - Two dimes = 20 cents
  - Two pennies = 2 cents
- Question 1: How do we know that this is quickest?
- Question 2: Can we generalize to arbitrary denominations?

The Change Problem: Formal Statement

- Goal: Convert some amount of money $\boldsymbol{M}$ into given denominations, using the fewest possible number of coins.

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

- Output: A list of d integers $i_1, i_2, \ldots, i_d$ such that

$$c_1 i_1 + c_2 i_2 + \ldots + c_d i_d = \boldsymbol{M}$$

and $i_1 + i_2 + \ldots + i_d$ is minimal.

The Change Problem: Another 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.

The Change Problem: Another 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	2	1	2	1	2		2		2

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

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

The Change Problem: Another 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	2	1	2	1	2	3	2	3	2

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

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

- Lastly, three coins are needed to make change for 7 and 9.

The Change Problem: Recurrence

- This example expresses the following recurrence relation:

$$minNumCoins(M) = \min \begin{cases} minNumCoins(M-1)+1 \\ minNumCoins(M-3)+1 \\ minNumCoins(M-5)+1 \end{cases}$$

The Change Problem: Recurrence

- In general, given the denominations ***c***: c_1, c_2, …, c_d, the recurrence relation is:

$$minNumCoins(M) = \min \begin{cases} minNumCoins(M - c_1) + 1 \\ minNumCoins(M - c_2) + 1 \\ \ldots \\ minNumCoins(M - c_d) + 1 \end{cases}$$

The Change Problem: Pseudocode

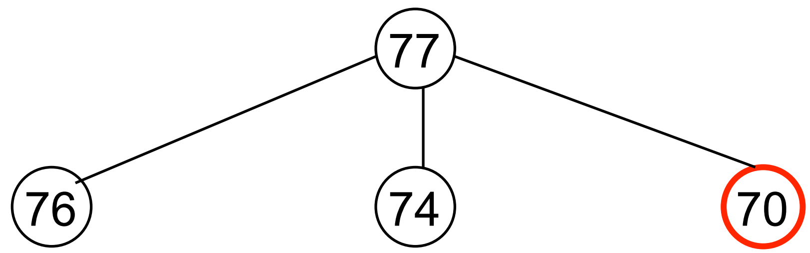
```
RecursiveChange(M,c,d)
   if M = 0
      return 0
  bestNumCoins ← infinity
  for i ← 1 to d
      if M ≥ c_i
         numCoins ← RecursiveChange(M – c_i , c, d)
         if numCoins + 1 < bestNumCoins
            bestNumCoins ← numCoins + 1
  return bestNumCoins
```

The RecursiveChange Tree: Example

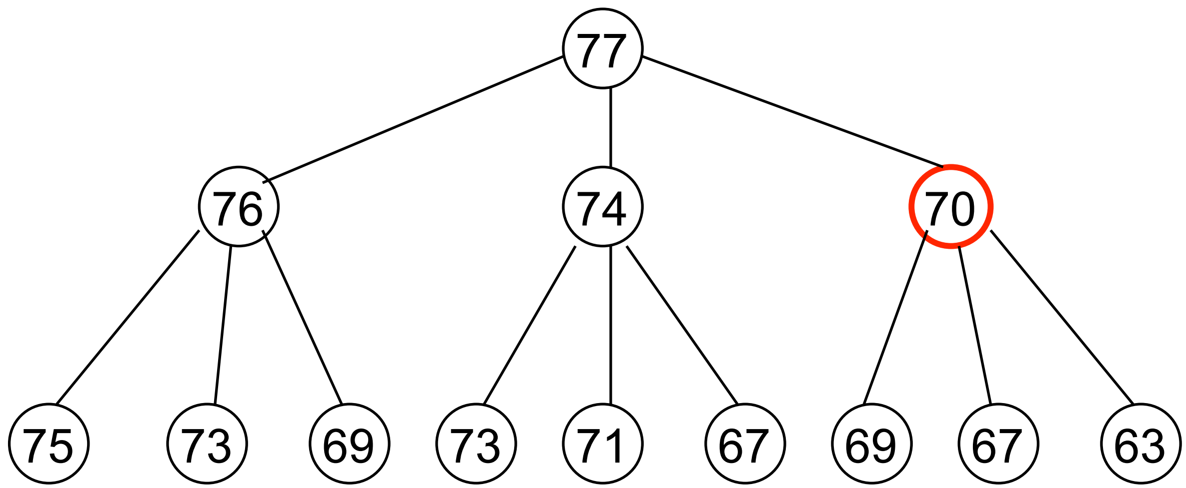
- We now will provide the tree of recursive calls if $M = 77$ and the denominations are 1, 3, and 7.
- We will outline all the occurrences of 70 cents to demonstrate how often it is called.

The RecursiveChange Tree

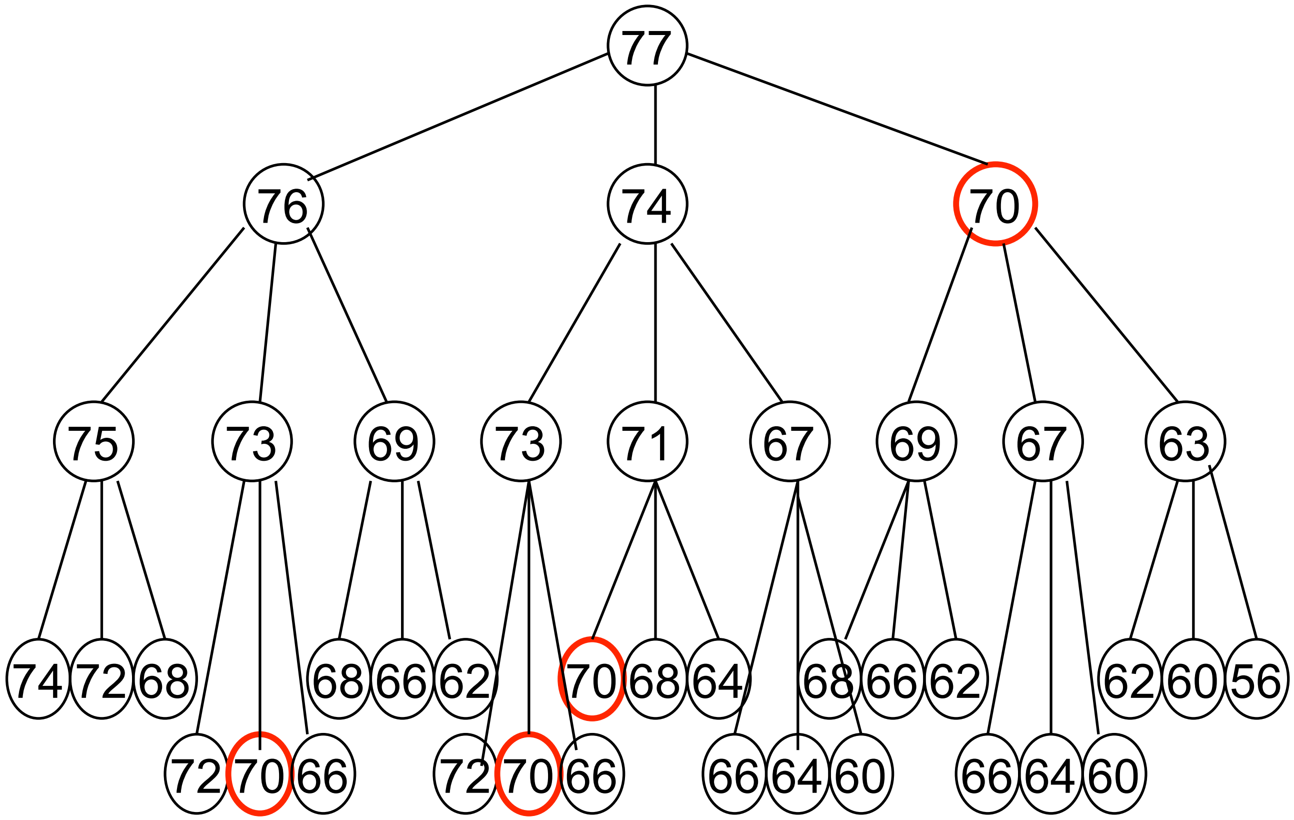
The RecursiveChange Tree



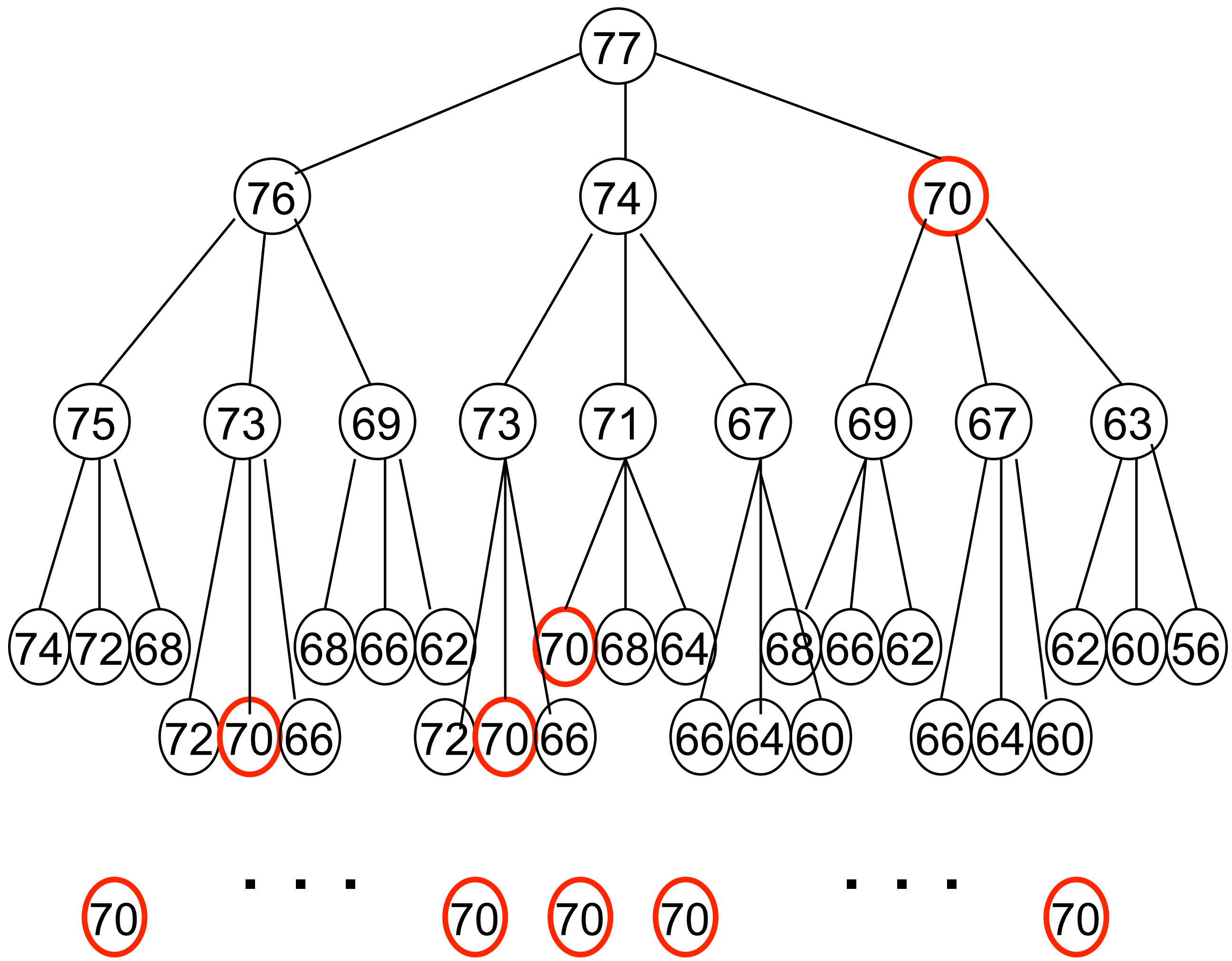
The RecursiveChange Tree



The RecursiveChange Tree



The RecursiveChange Tree



RecursiveChange: Inefficiencies

- As we can see, RecursiveChange recalculates the optimal coin combination for a given amount of money repeatedly.
- For our example of $\boldsymbol{M} = 77$, $\boldsymbol{c} = (1,3,7)$:
  - The optimal coin combination for 70 cents is computed 9 times!

RecursiveChange: Inefficiencies

- As we can see, RecursiveChange recalculates the optimal coin combination for a given amount of money repeatedly.

- For our example of $\boldsymbol{M} = 77$, $\boldsymbol{c} = (1,3,7)$:
  - The optimal coin combination for 70 cents is computed 9 times!
  - The optimal coin combination for 50 cents is computed billions of times!

RecursiveChange: Inefficiencies

- As we can see, RecursiveChange recalculates the optimal coin combination for a given amount of money repeatedly.
- For our example of $\boldsymbol{M} = 77$, $\boldsymbol{c} = (1,3,7)$:
  - The optimal coin combination for 70 cents is computed 9 times!
  - The optimal coin combination for 50 cents is computed billions of times!
  - Imagine how many times the optimal coin combination for 3 cents would be calculated…

RecursiveChange: Suggested Improvements

- We're re-computing values in our algorithm more than once.
- Instead, let's save results of each computation for all amounts from 0 to M. This way, we can do a reference call to find an already computed value, instead of re-computing each time.
- The new algorithm will have running time $M*d$, where M is the amount of money and d is the number of denominations.
- This is an example of the method of **dynamic programming**.

The Change Problem: Dynamic Programming

<u>DPChange($\boldsymbol{M,c,d}$)</u>

$\quad bestNumCoins_0 \leftarrow 0$

$\quad$ **for** $\boldsymbol{m} \leftarrow 1$ to $\boldsymbol{M}$

$\qquad bestNumCoins_m \leftarrow$ infinity

$\qquad$ **for** $i \leftarrow 1$ to $\boldsymbol{d}$

$\qquad\quad$ **if** $\boldsymbol{m} \geq c_i$

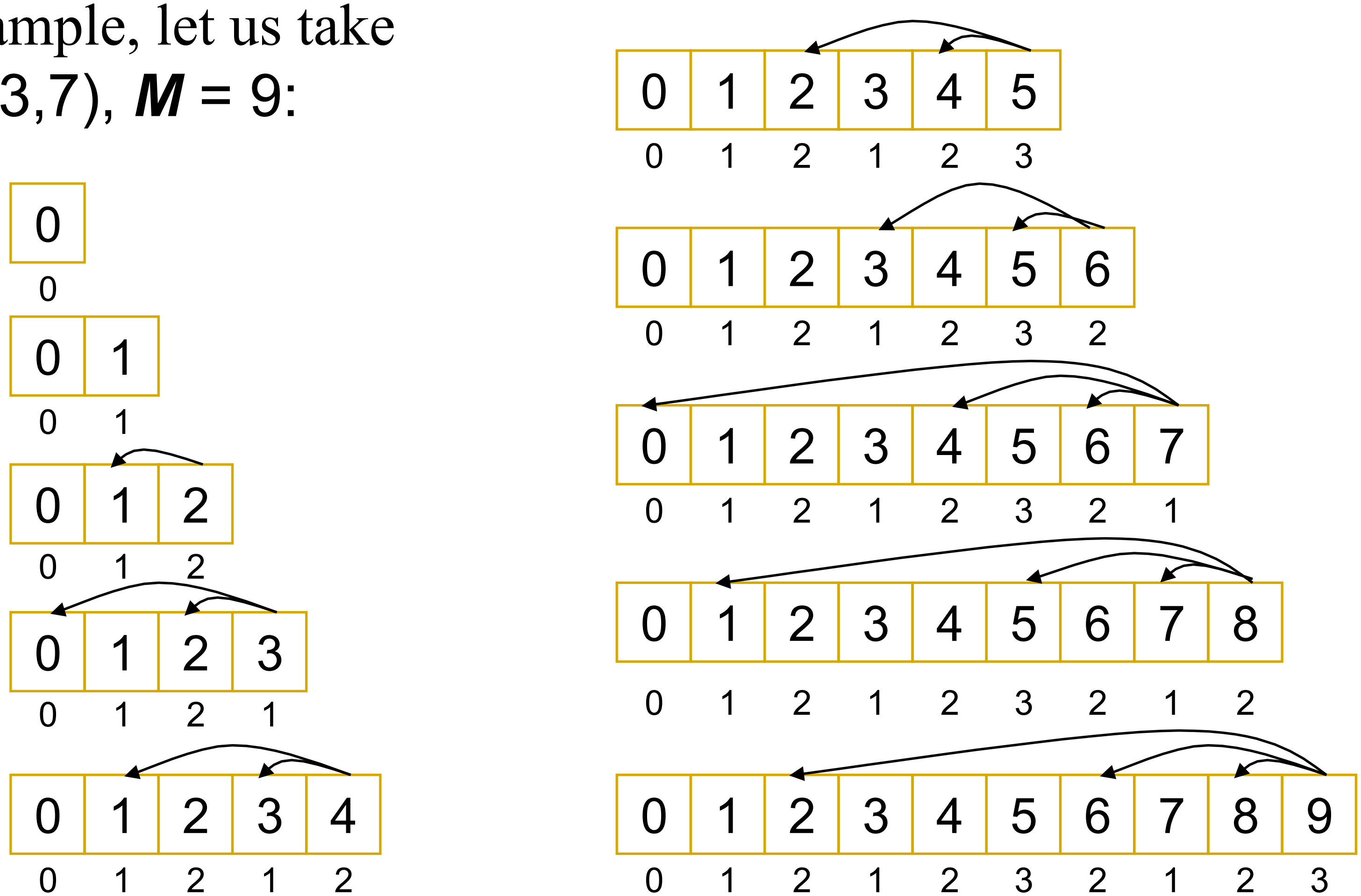
$\qquad\qquad$ **if** $bestNumCoins_{m - c_i} + 1 < bestNumCoins_m$

$\qquad\qquad\quad bestNumCoins_m \leftarrow bestNumCoins_{m - c_i} + 1$

$\quad$ **return** $bestNumCoins_M$

DPChange: Example

- For example, let us take
  $\boldsymbol{c} = (1,3,7)$, $\boldsymbol{M} = 9$:



Quick Note on Dynamic Programming

- You may have noticed that the dynamic programming algorithm provided somewhat resembles the recursive algorithm we already had.

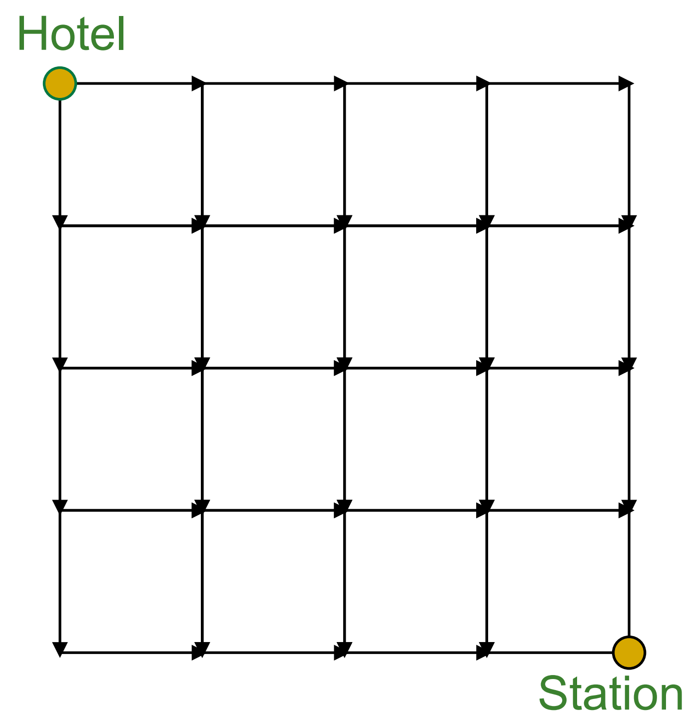
- The difference is that with recursion, we had constant repetition since we proceeded from more complicated sums "down the tree" to less complicated ones.

- With DPChange, we always "build up" from easier problem instances to the desired one, and in so doing avoid repetition.

Section 3: Manhattan Tourist Problem

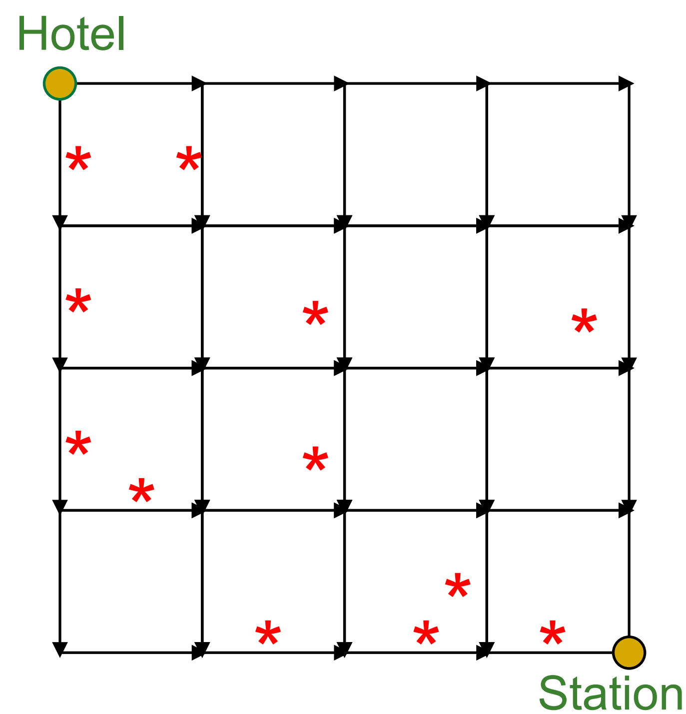
Additional Example: Manhattan Tourist Problem

- Imagine that you are a tourist in Manhattan, whose streets are represented by the grid on the right.



Additional Example: Manhattan Tourist Problem

- Imagine that you are a tourist in Manhattan, whose streets are represented by the grid on the right.
- You are leaving town, and you want to see as many attractions (represented by *) as possible.

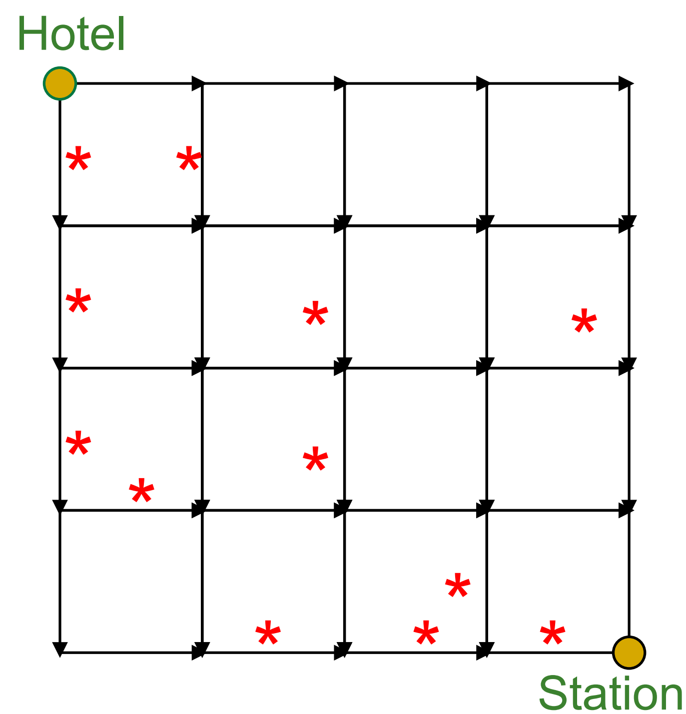


Additional Example: Manhattan Tourist Problem

- Imagine that you are a tourist in Manhattan, whose streets are represented by the grid on the right.

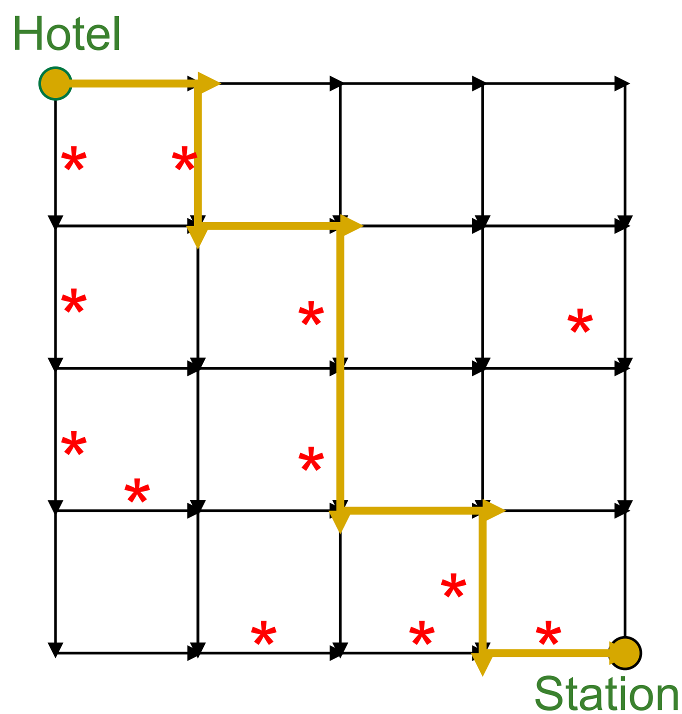
- You are leaving town, and you want to see as many attractions (represented by *) as possible.

- Your time is limited: you only have time to travel east and south.



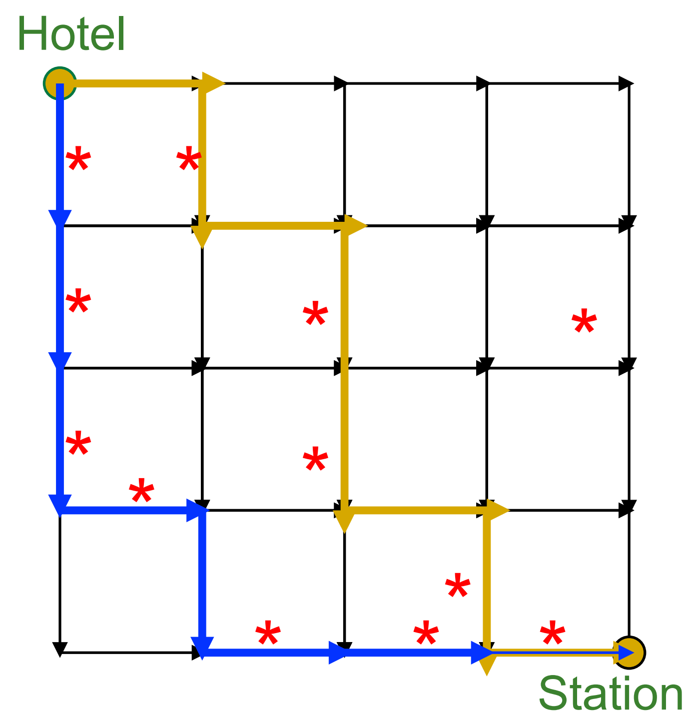
Additional Example: Manhattan Tourist Problem

- Imagine that you are a tourist in Manhattan, whose streets are represented by the grid on the right.
- You are leaving town, and you want to see as many attractions (represented by *) as possible.
- Your time is limited: you only have time to travel east and south.
- What is the best path through town?



Additional Example: Manhattan Tourist Problem

- Imagine that you are a tourist in Manhattan, whose streets are represented by the grid on the right.
- You are leaving town, and you want to see as many attractions (represented by *) as possible.
- Your time is limited: you only have time to travel east and south.
- What is the best path through town?



Manhattan Tourist Problem (MTP): Formulation

- Goal: Find the longest path in a weighted grid.

- Input: A weighted grid ***G*** with two distinct vertices, one labeled "*source*" and the other labeled "*sink.*"

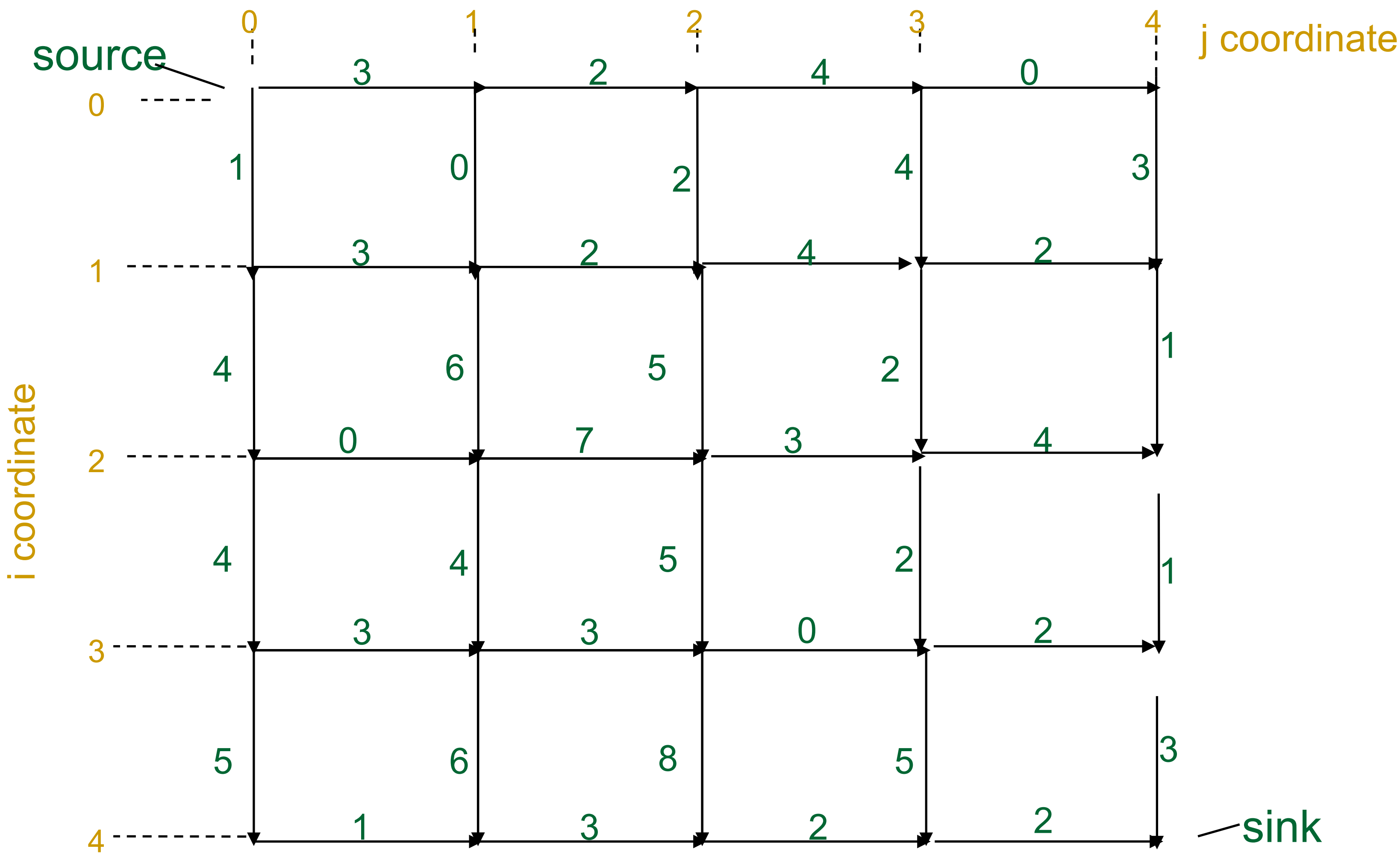
- Output: A longest path in ***G*** from "*source*" to "*sink.*"

MTP Greedy Algorithm

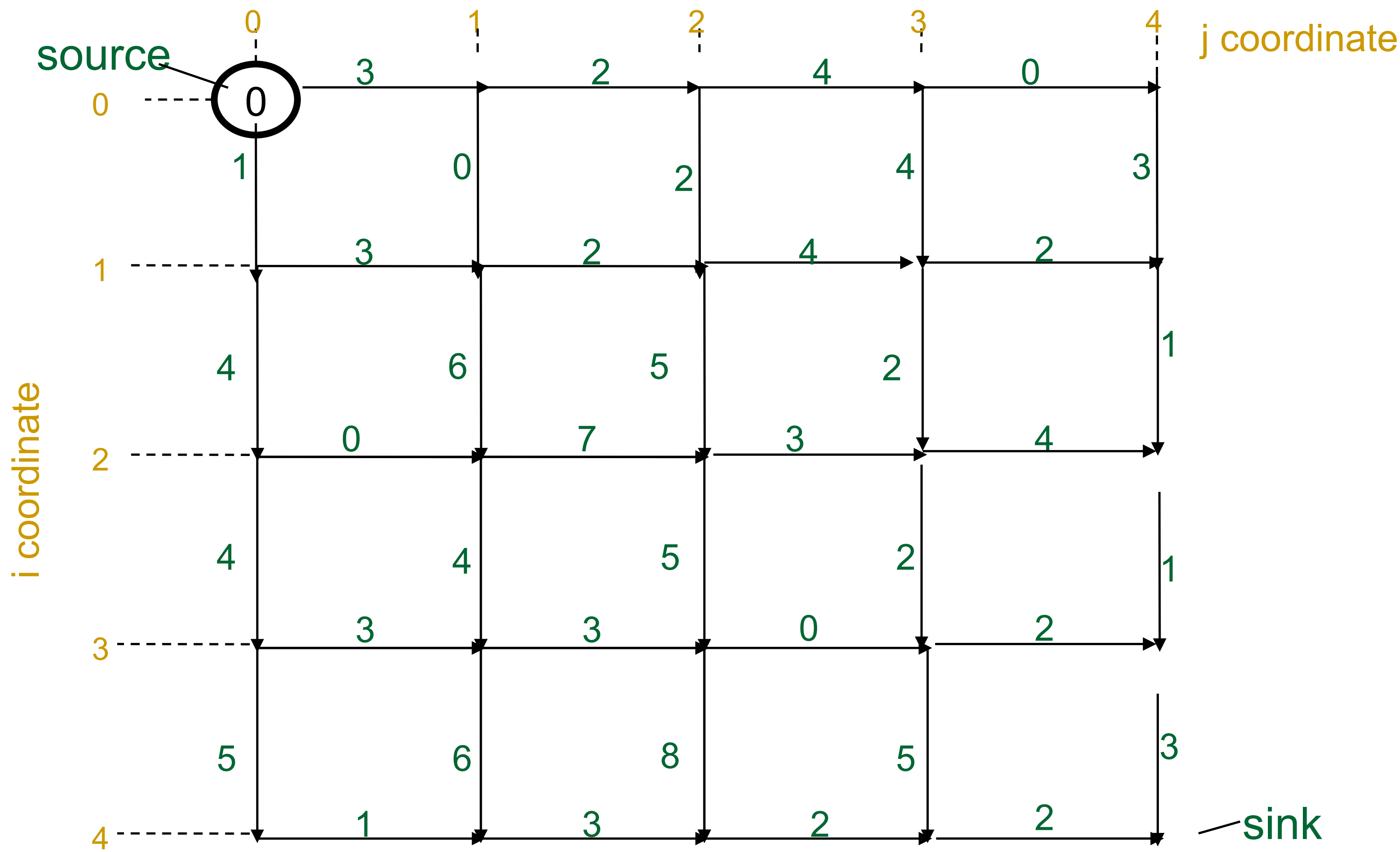
- Our first try at solving the MTP will use a **greedy algorithm**.

- Main Idea: At each node (intersection), choose the edge (street) departing that node which has the greatest weight.

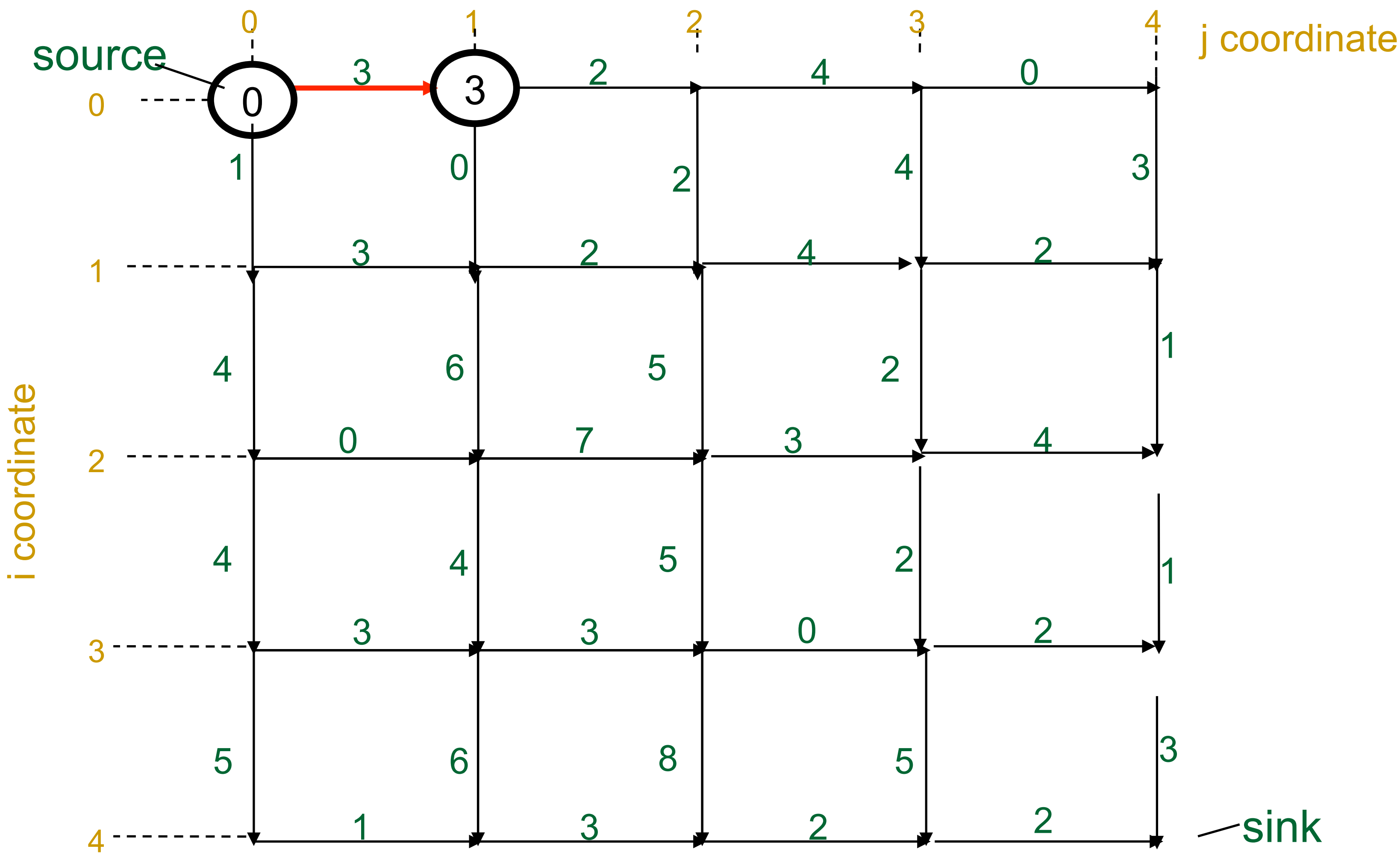
MTP Greedy Algorithm: Example



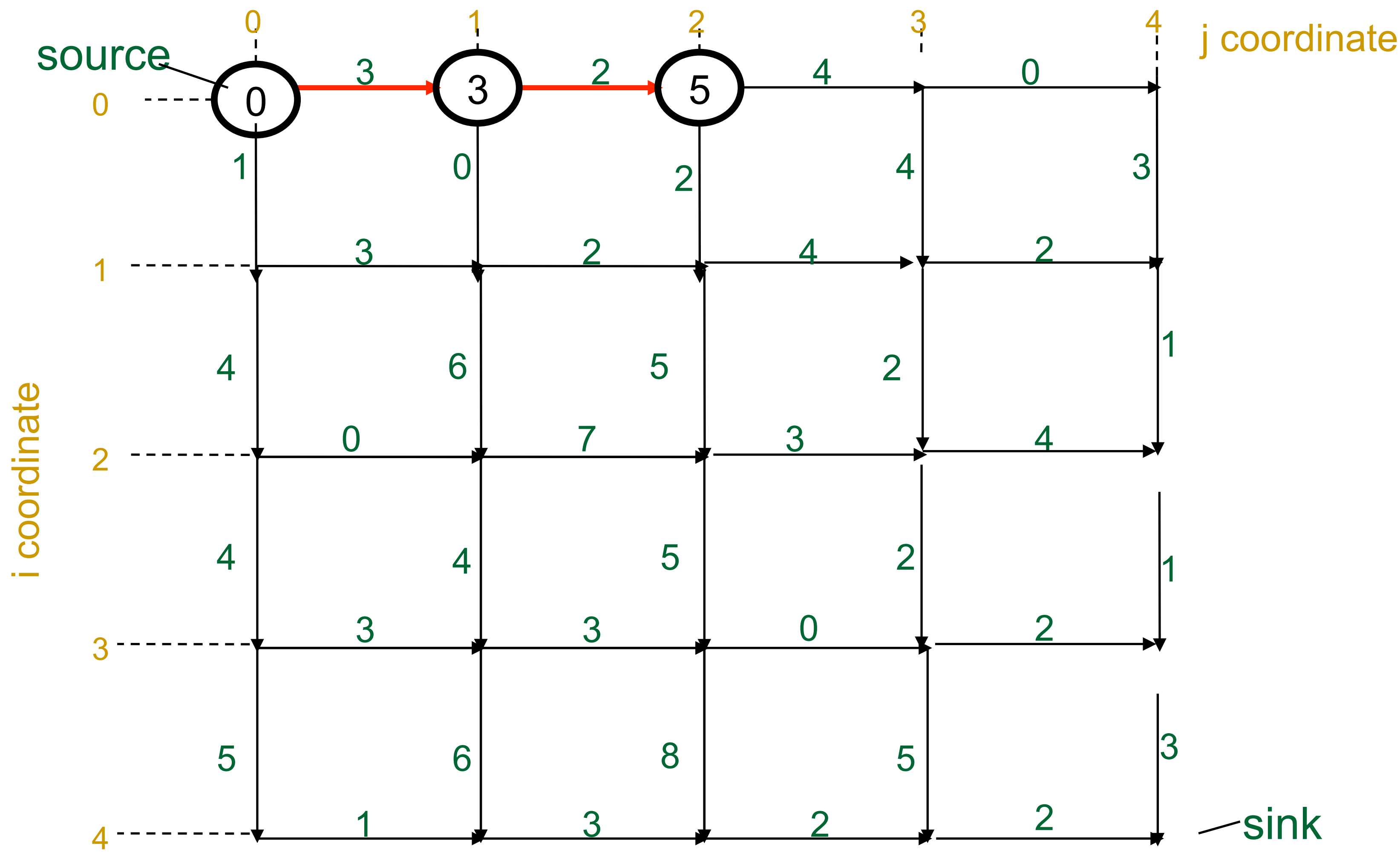
MTP Greedy Algorithm: Example



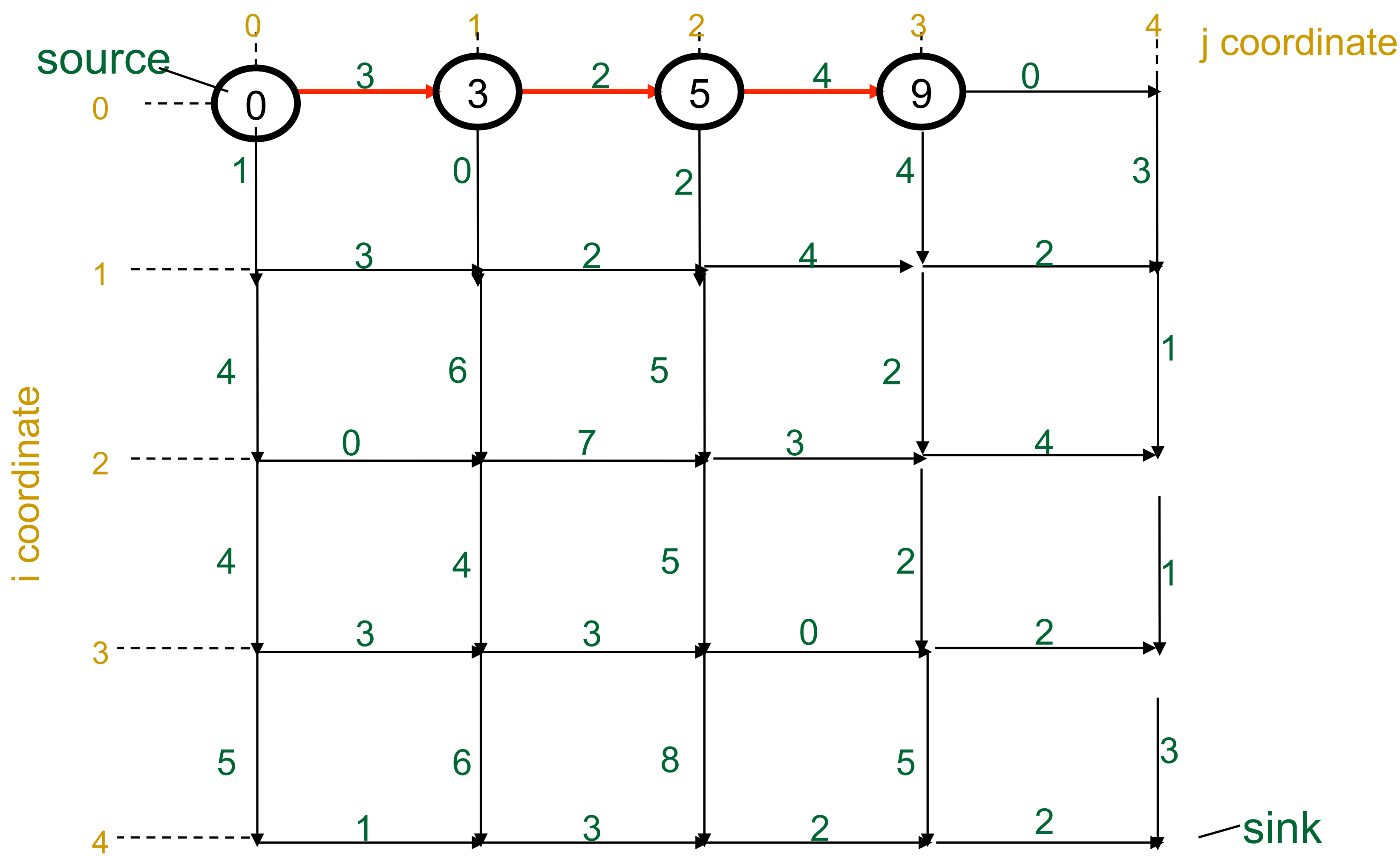
MTP Greedy Algorithm: Example



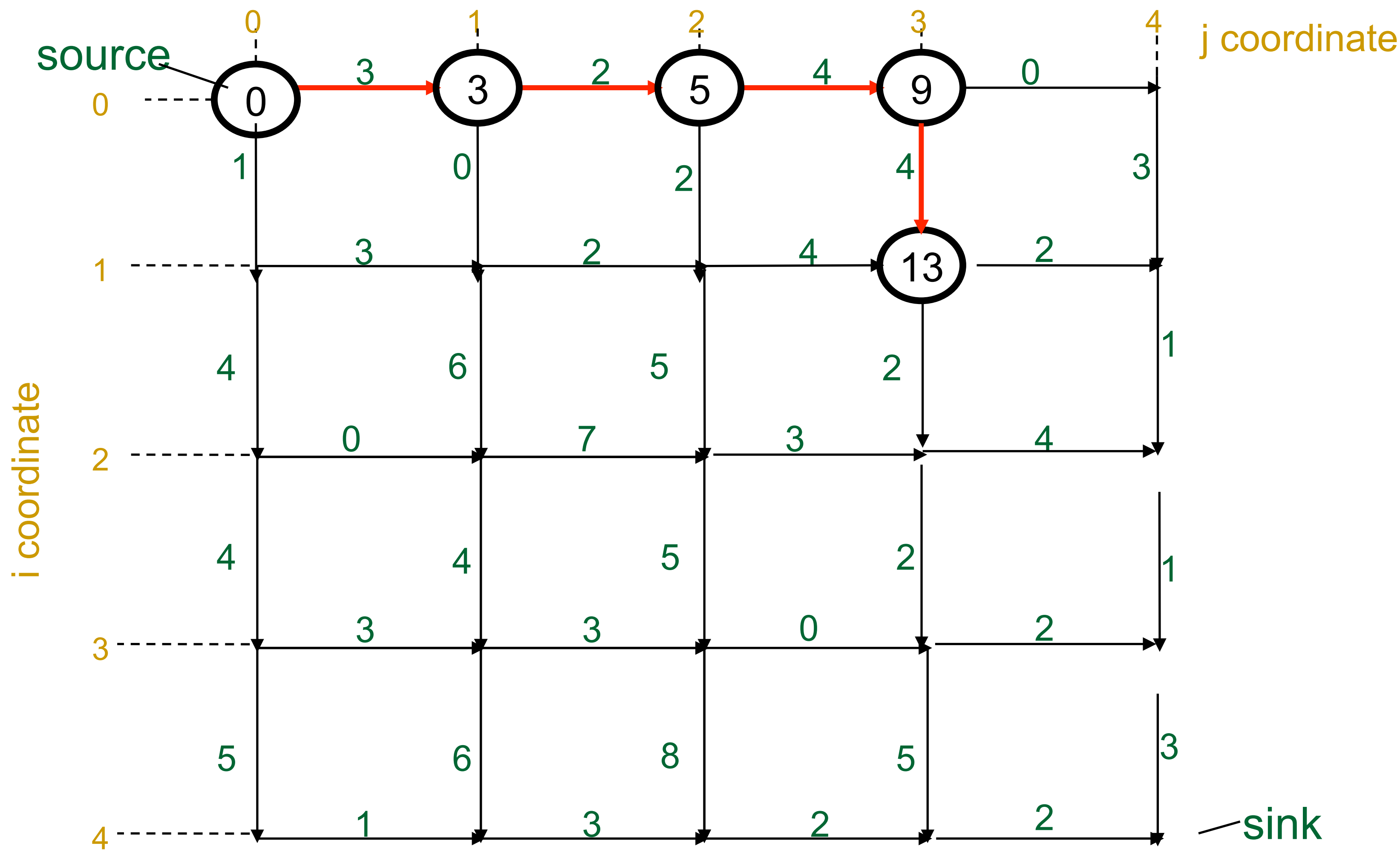
MTP Greedy Algorithm: Example



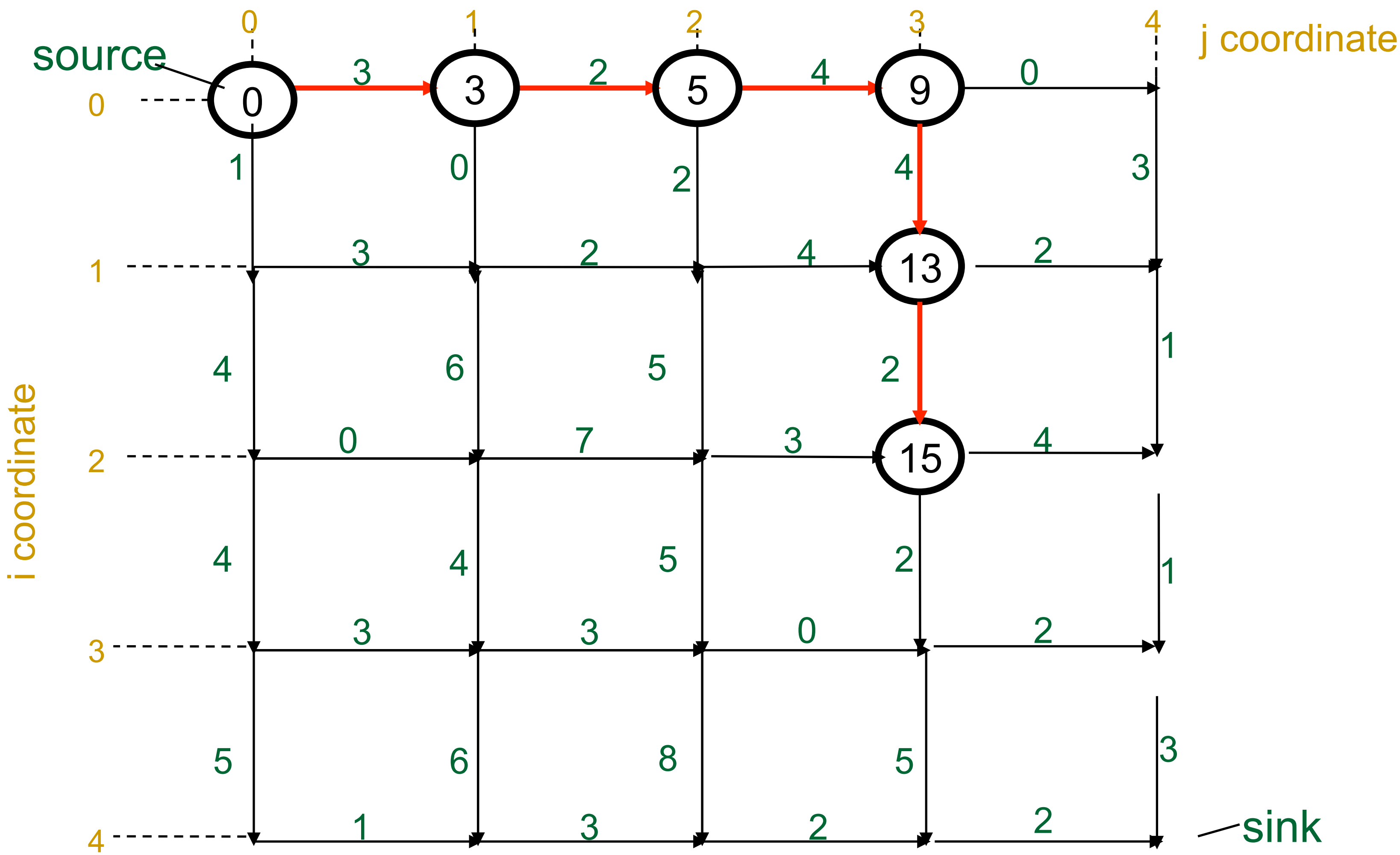
MTP Greedy Algorithm: Example



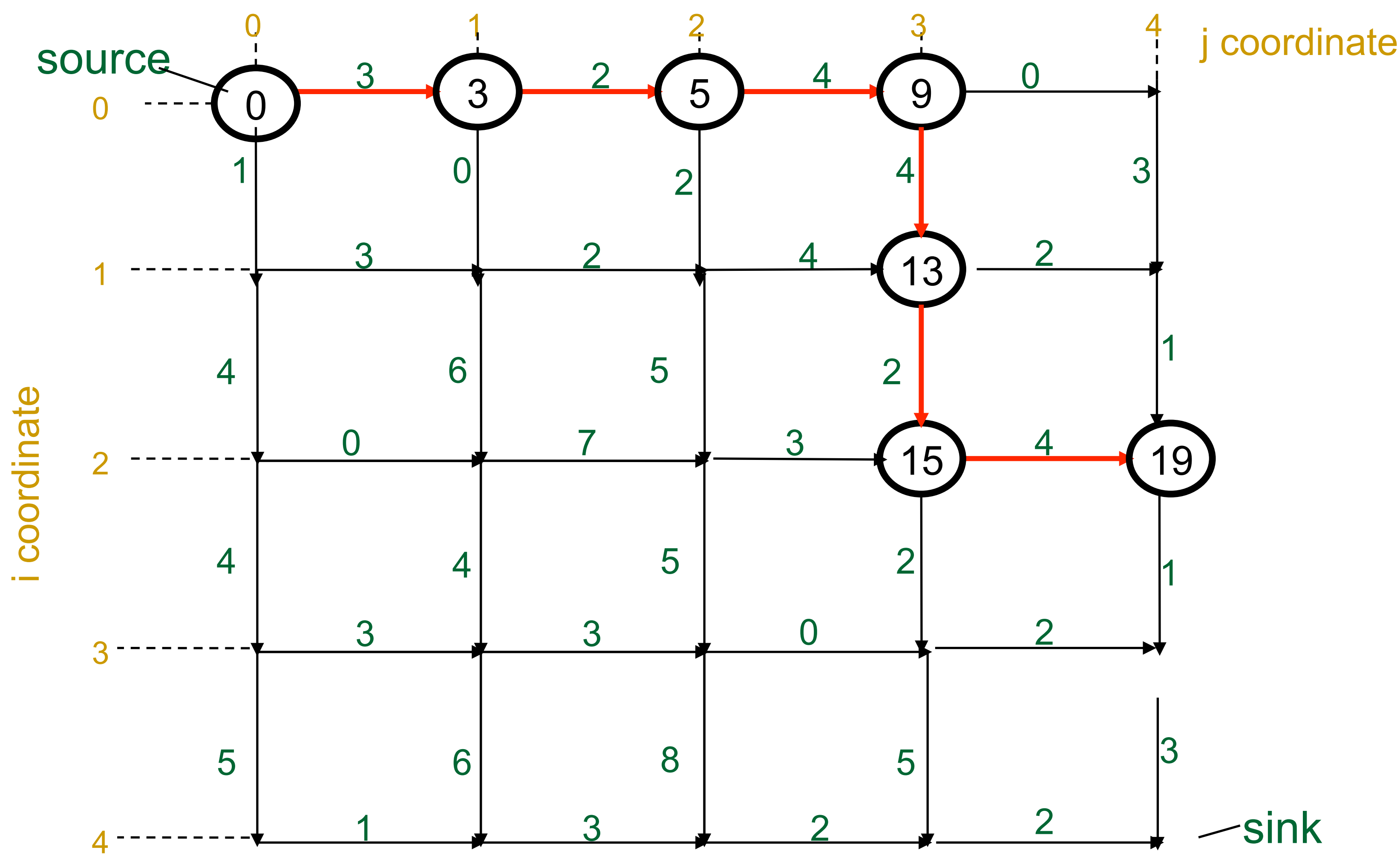
MTP Greedy Algorithm: Example



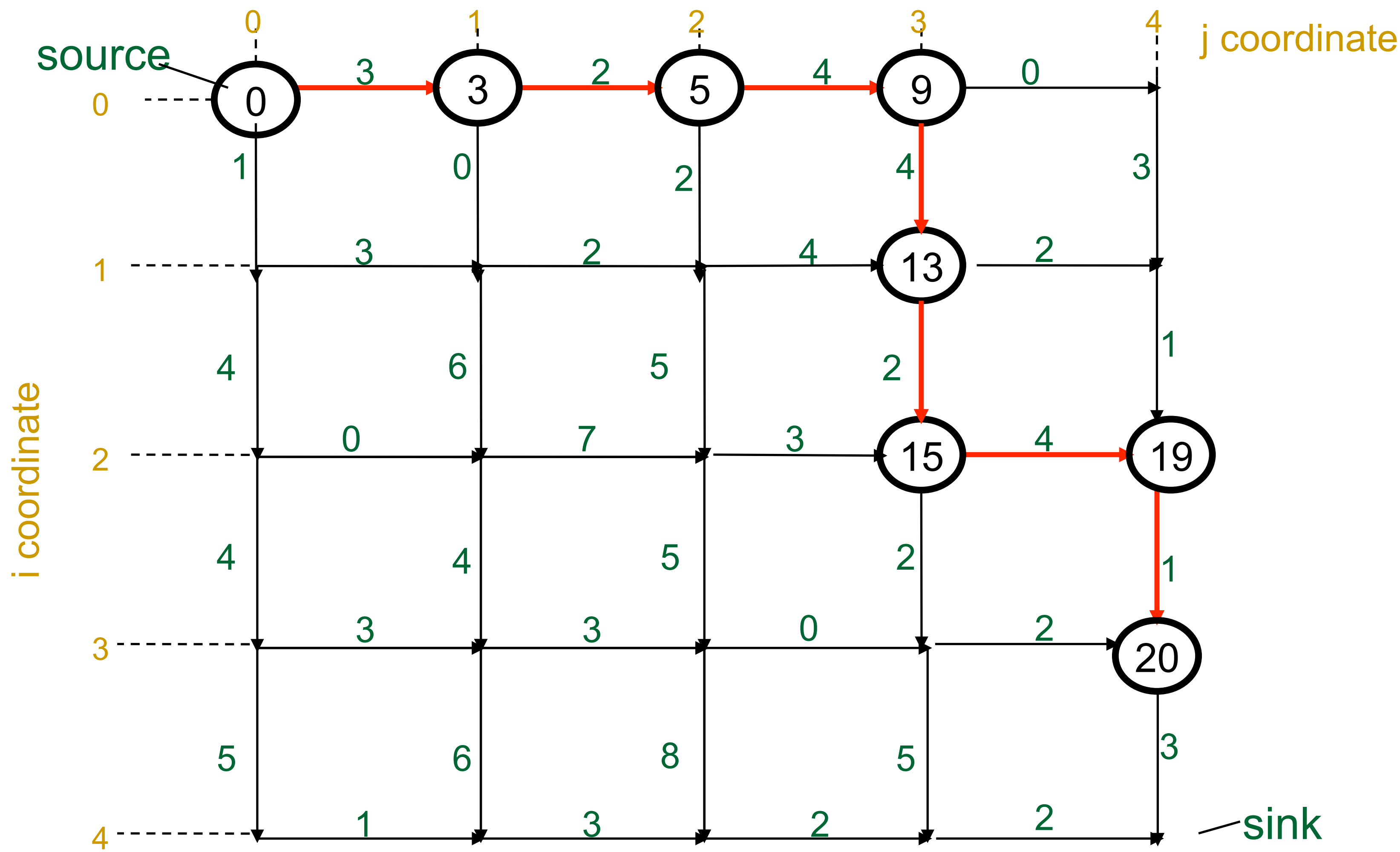
MTP Greedy Algorithm: Example



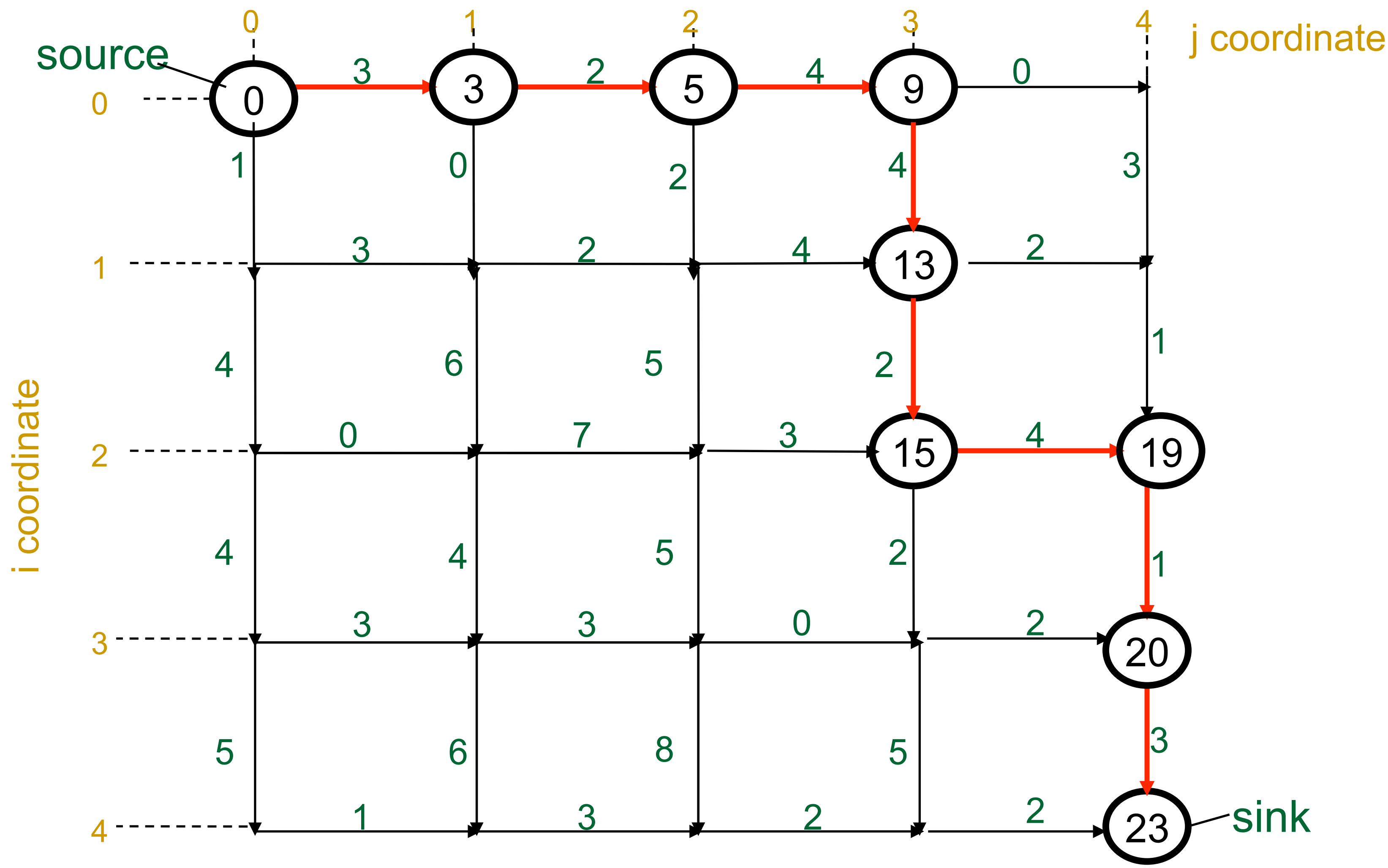
MTP Greedy Algorithm: Example



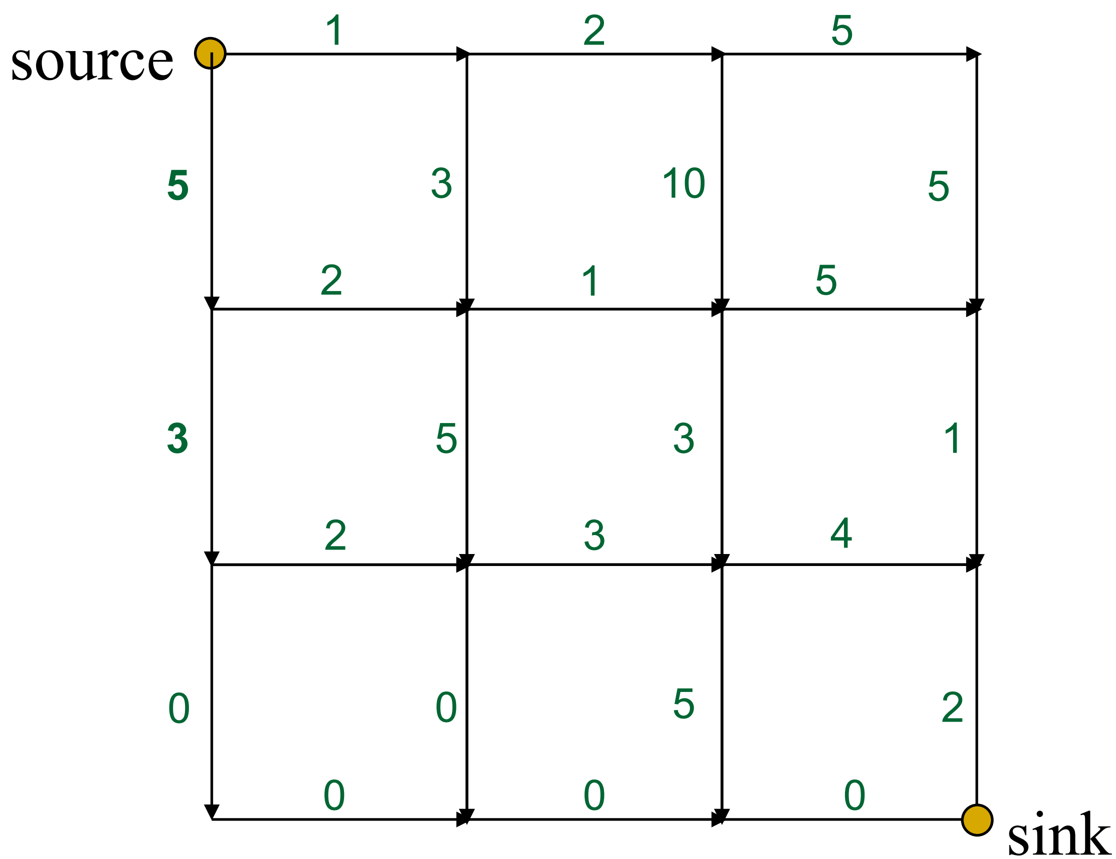
MTP Greedy Algorithm: Example



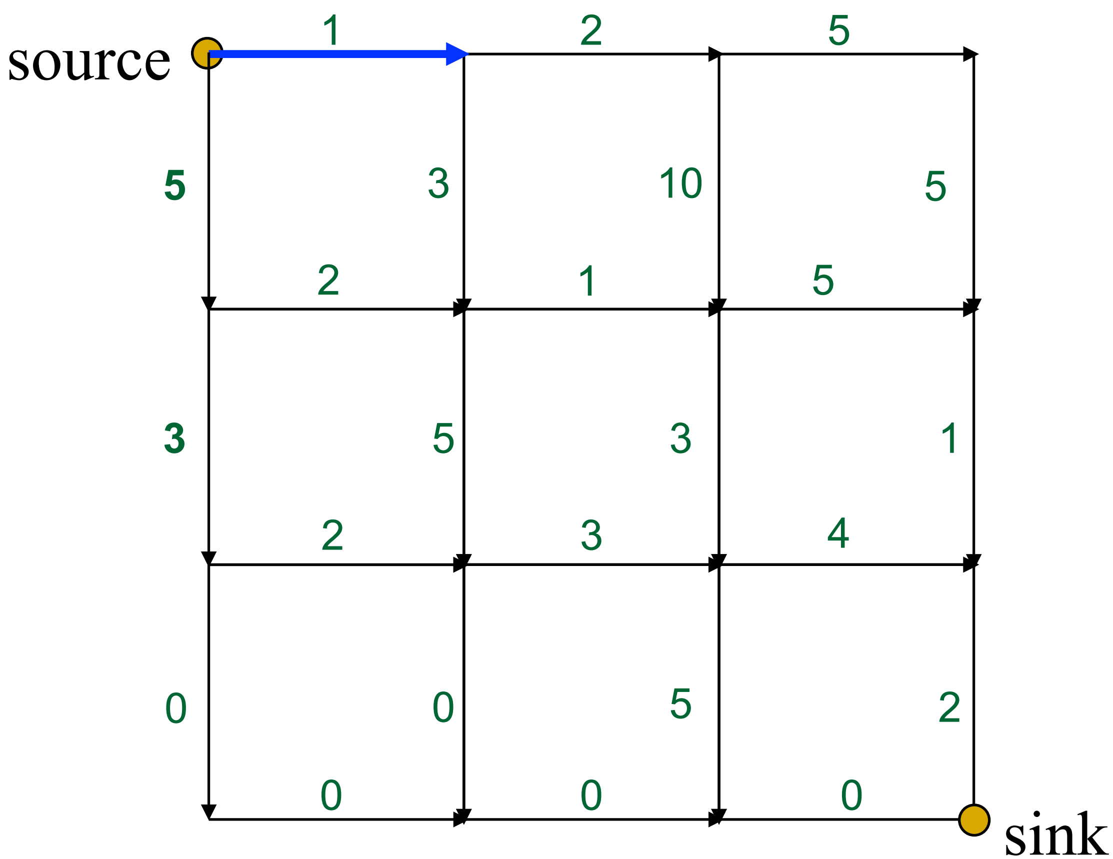
MTP Greedy Algorithm: Example



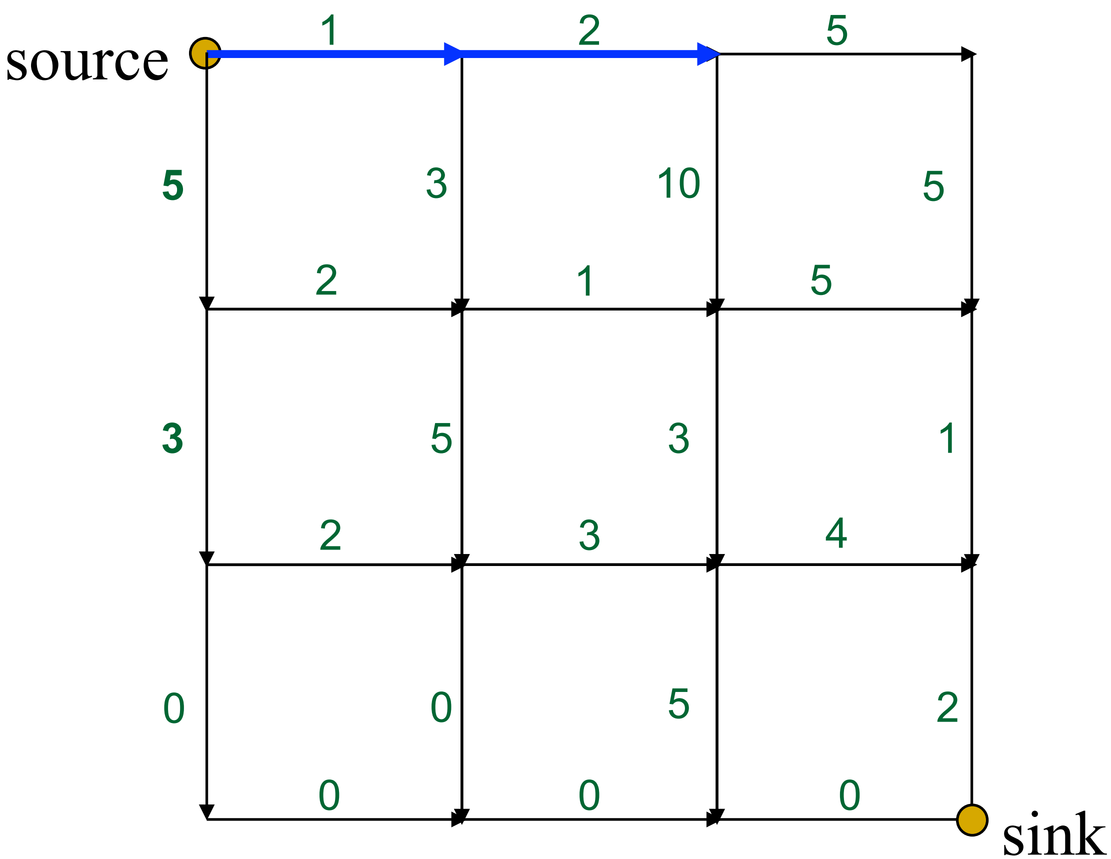
MTP Greedy Algorithm Is Not Optimal



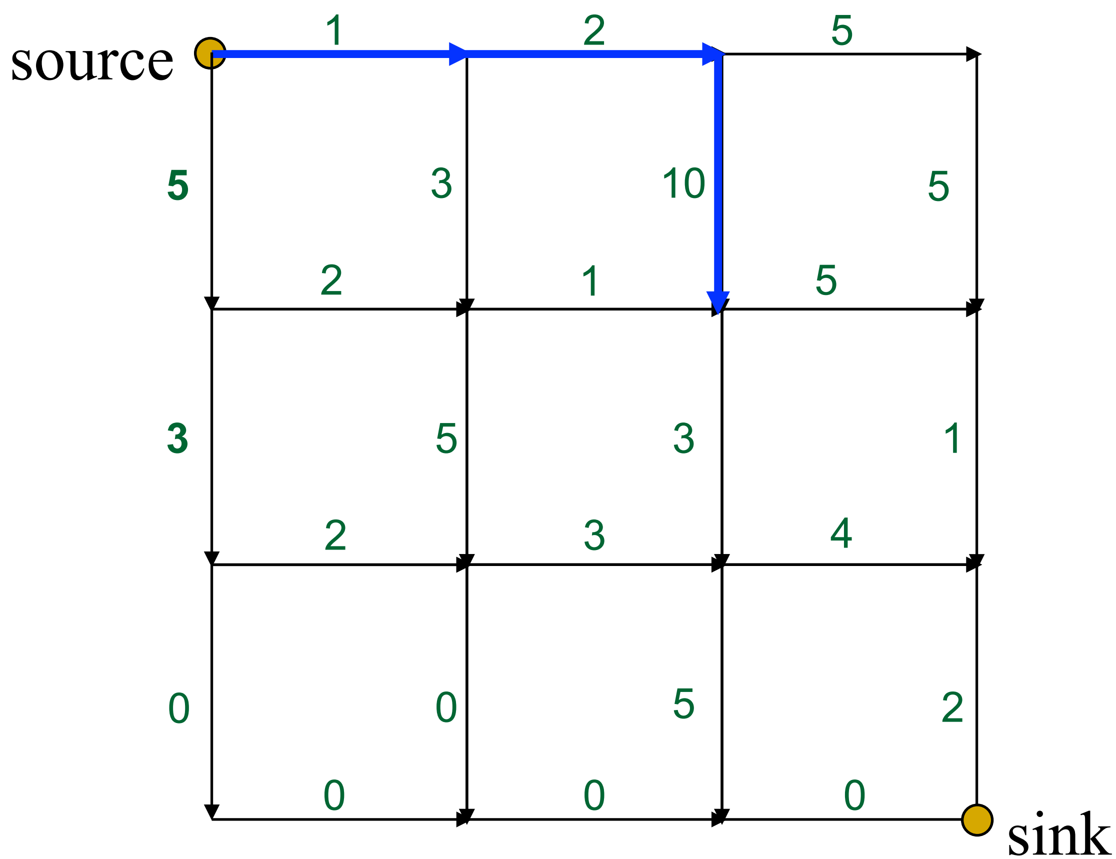
MTP Greedy Algorithm Is Not Optimal



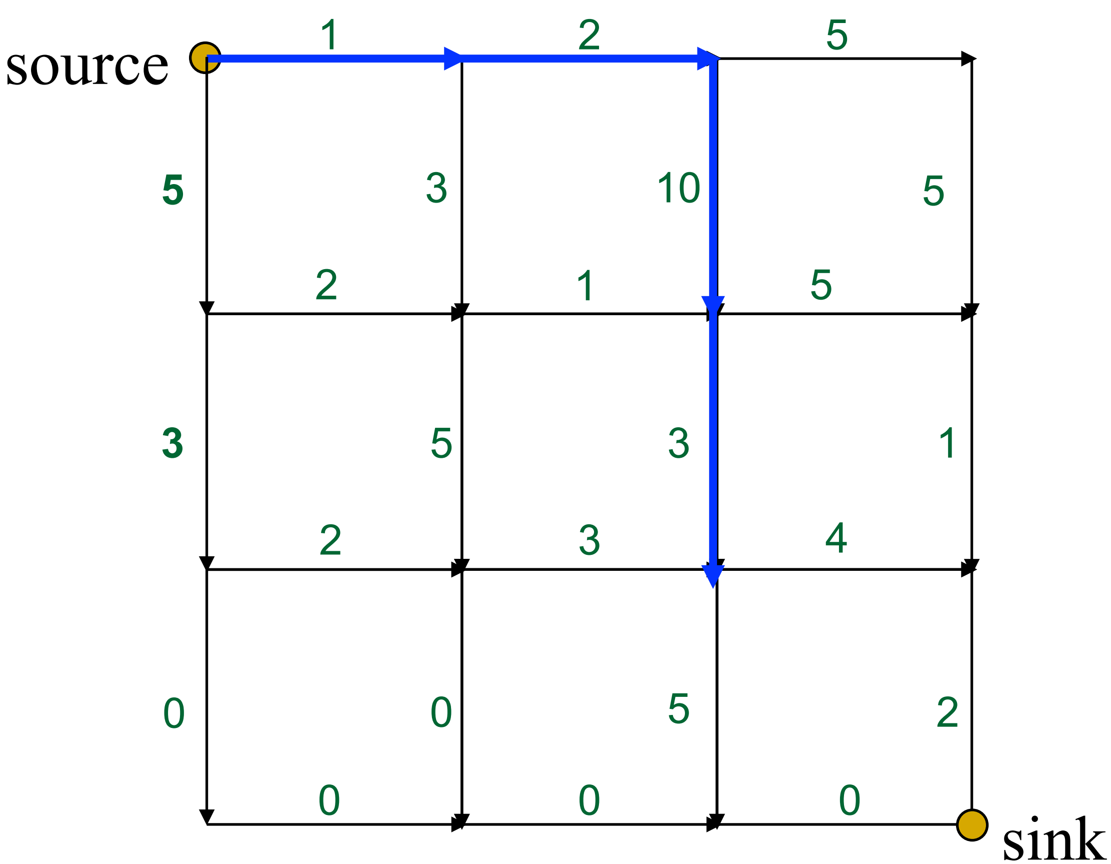
MTP Greedy Algorithm Is Not Optimal



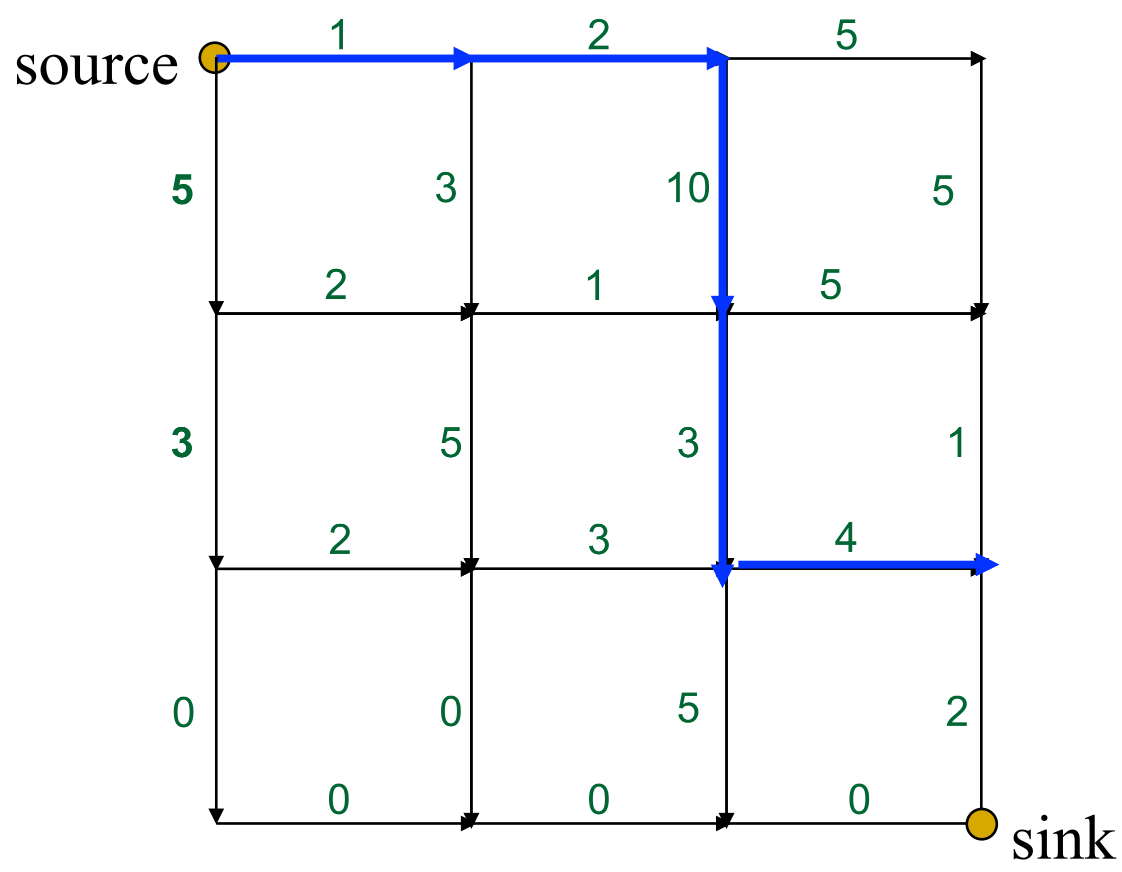
MTP Greedy Algorithm Is Not Optimal



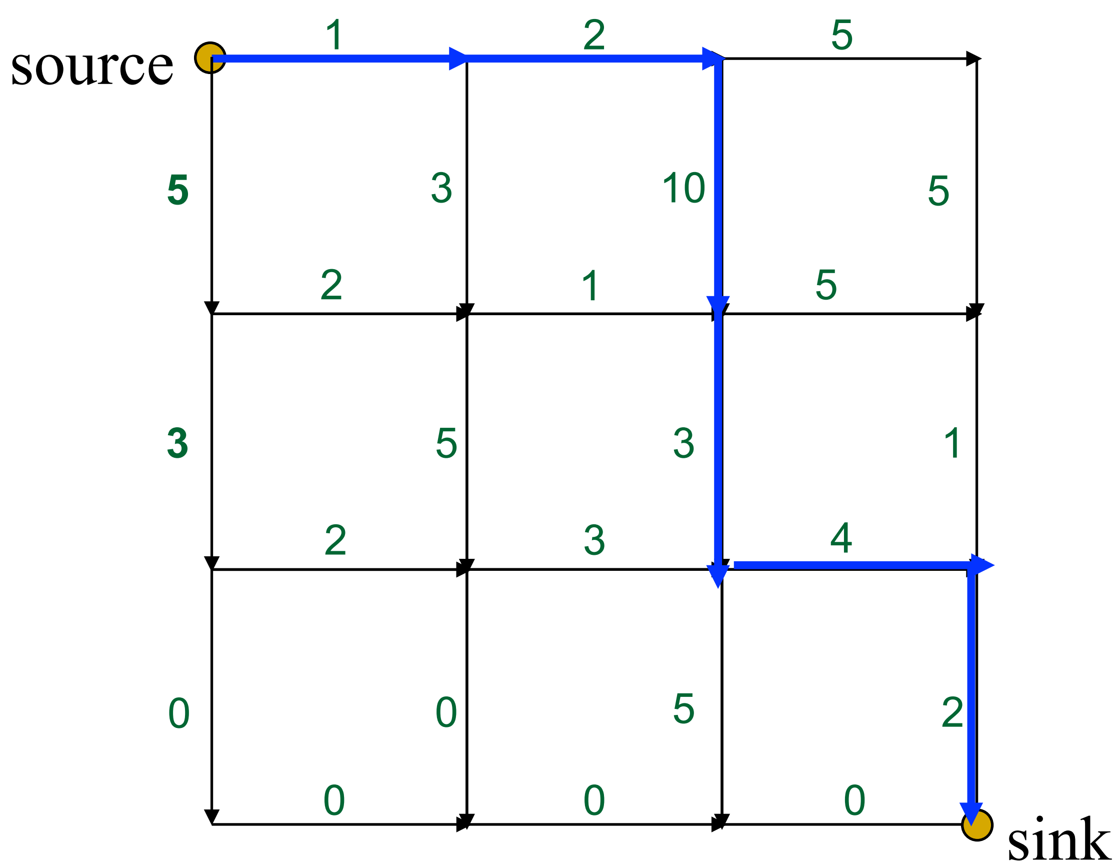
MTP Greedy Algorithm Is Not Optimal



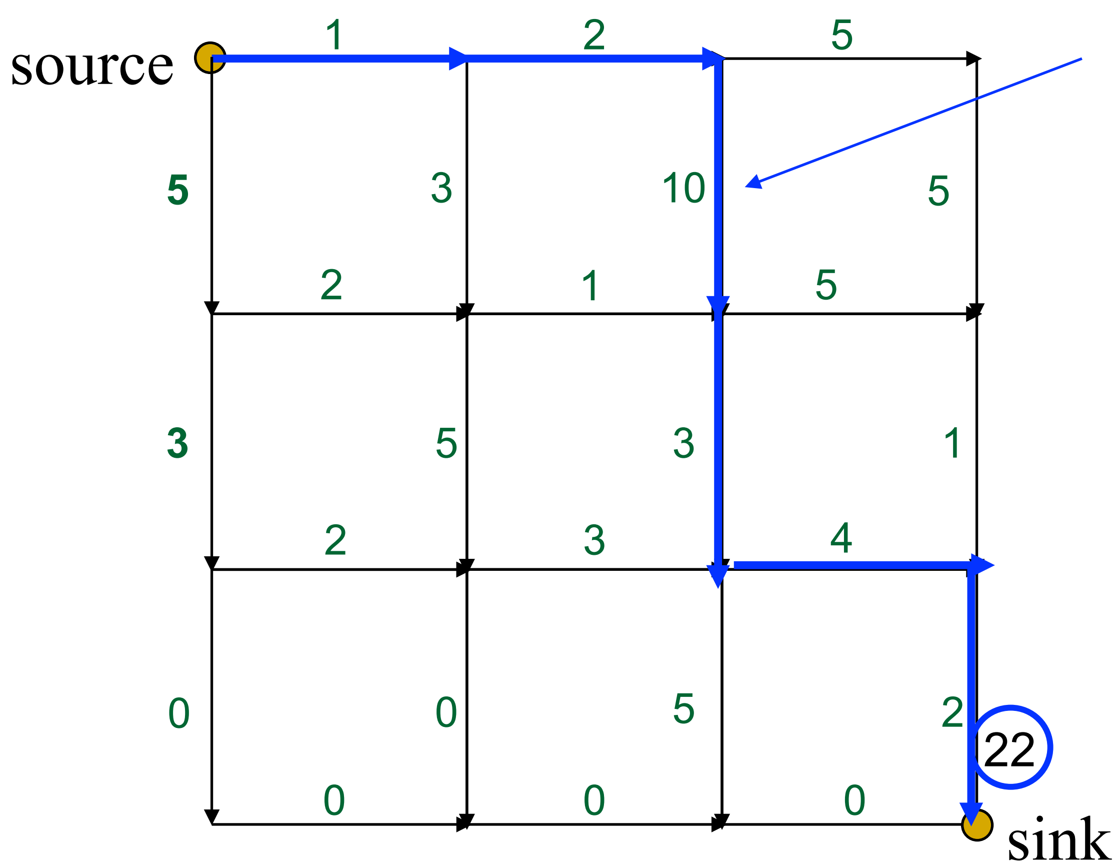
MTP Greedy Algorithm Is Not Optimal



MTP Greedy Algorithm Is Not Optimal

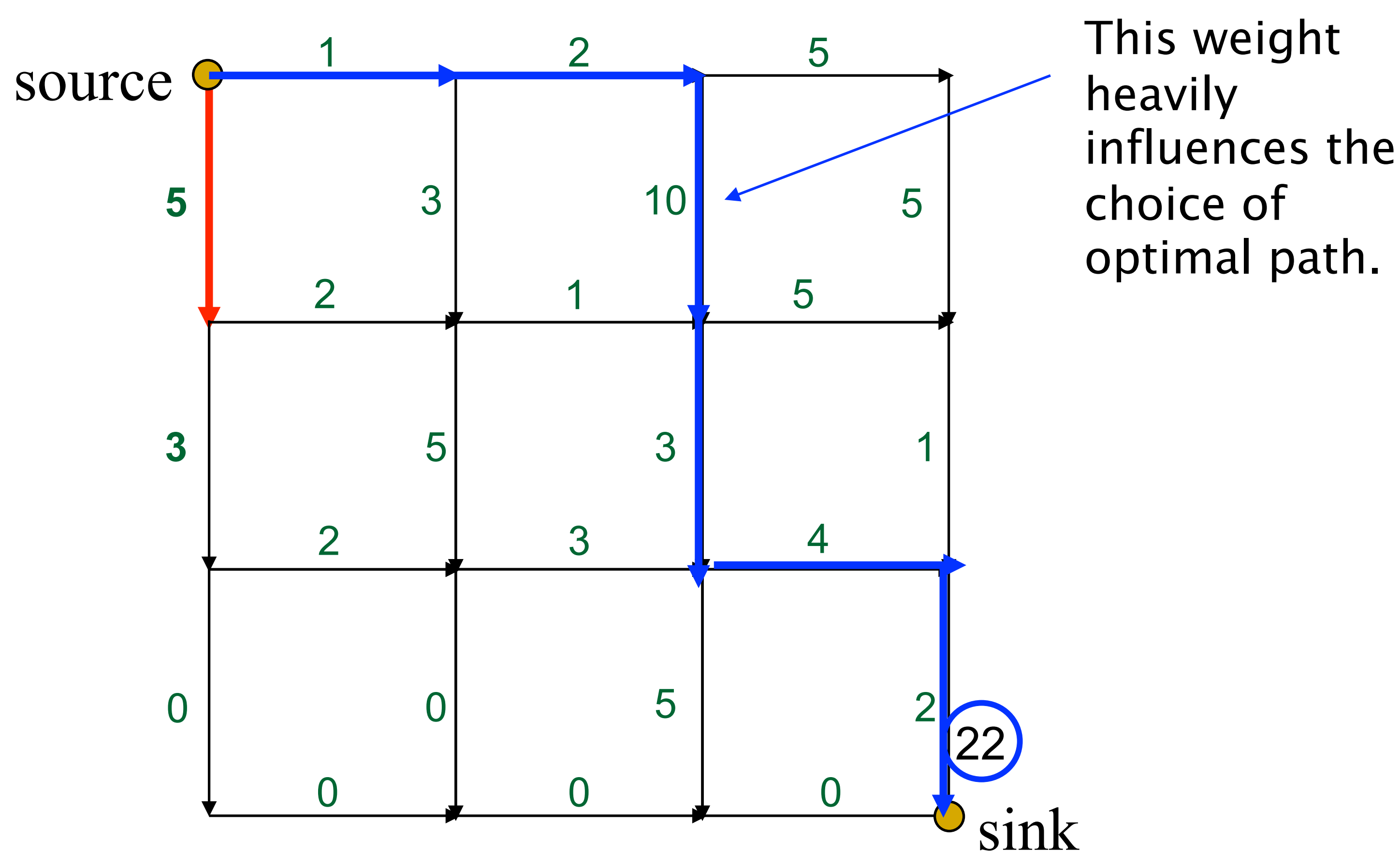


MTP Greedy Algorithm Is Not Optimal

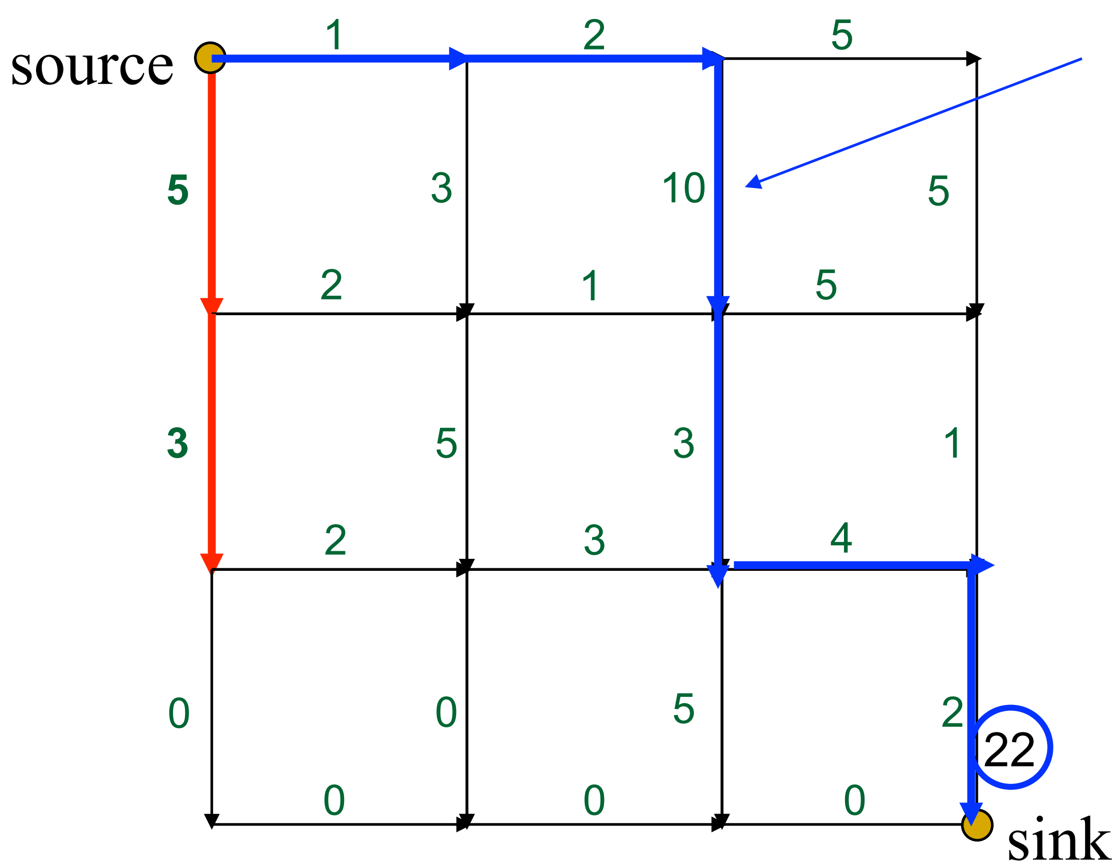


This weight heavily influences the choice of optimal path.

MTP Greedy Algorithm Is Not Optimal

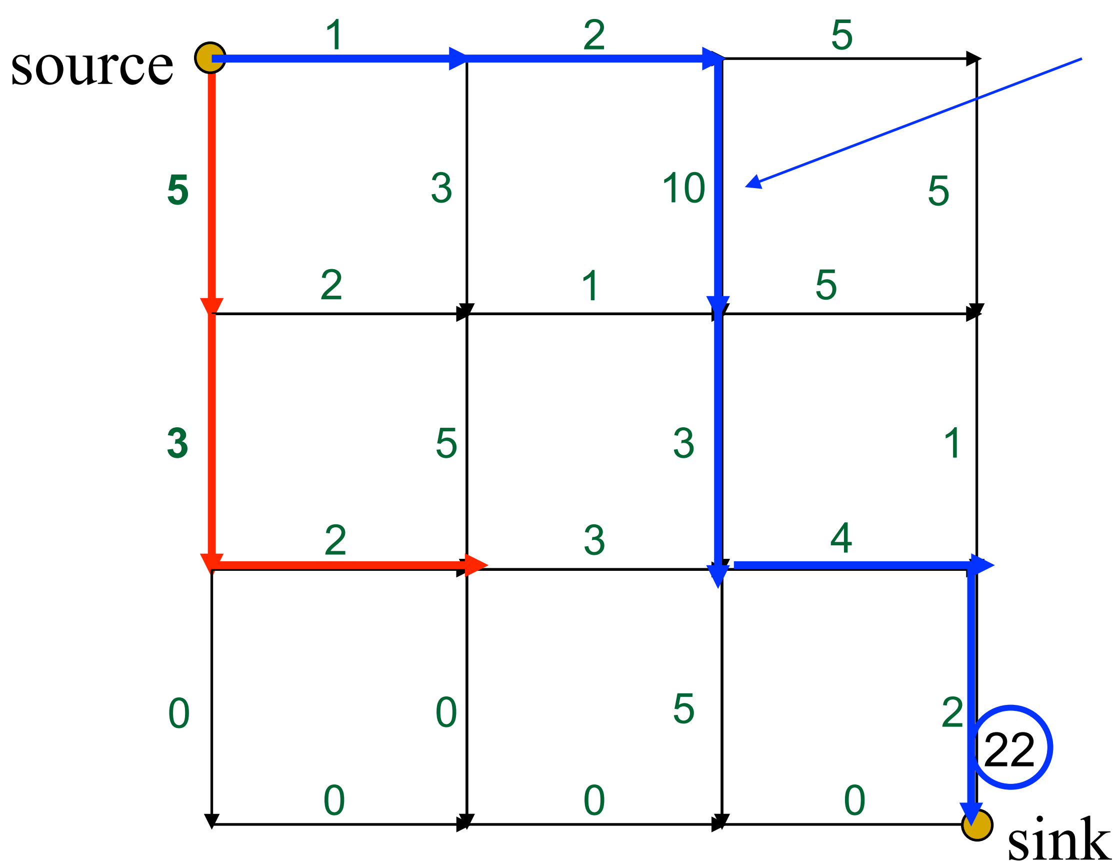


MTP Greedy Algorithm Is Not Optimal



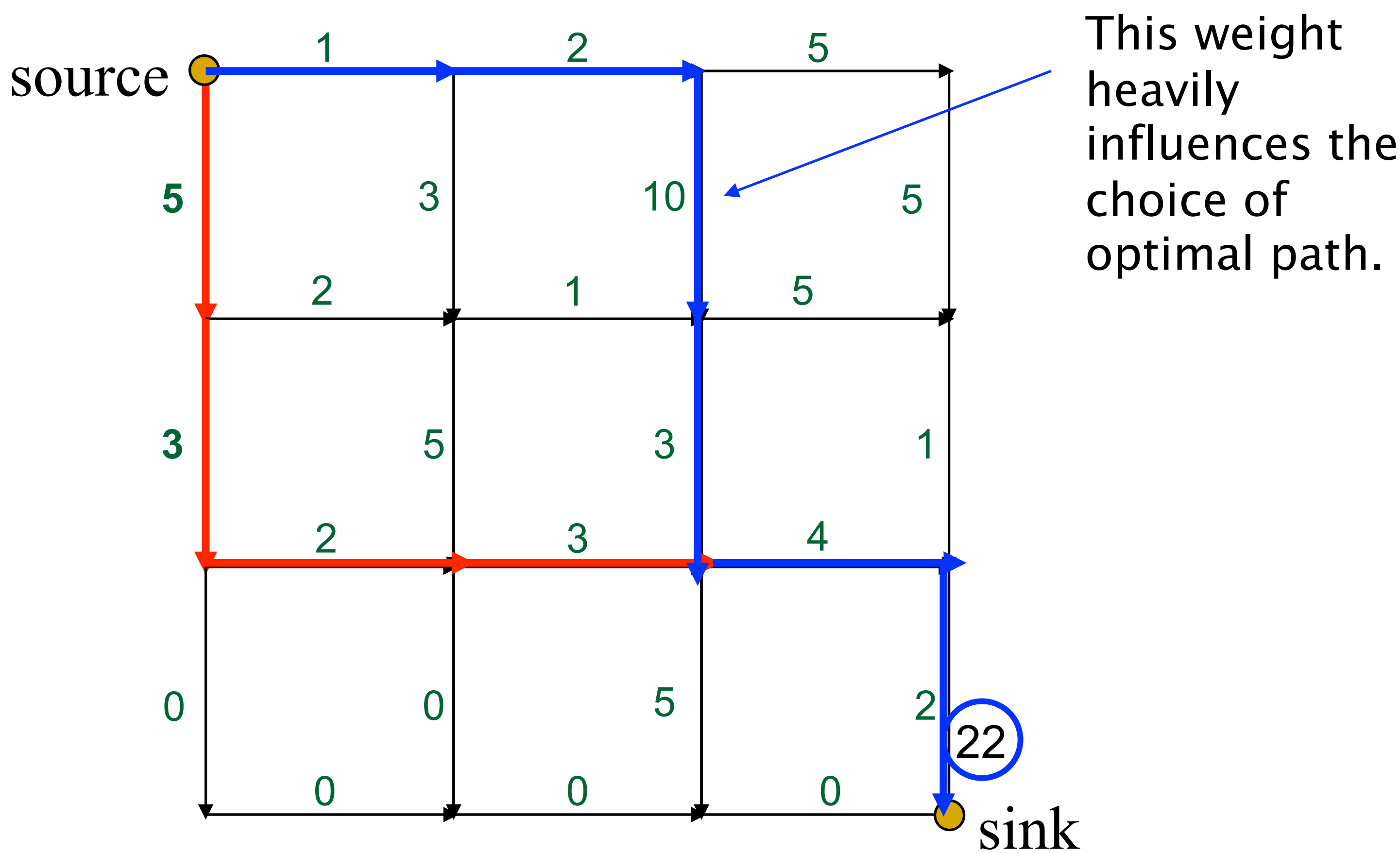
This weight heavily influences the choice of optimal path.

MTP Greedy Algorithm Is Not Optimal

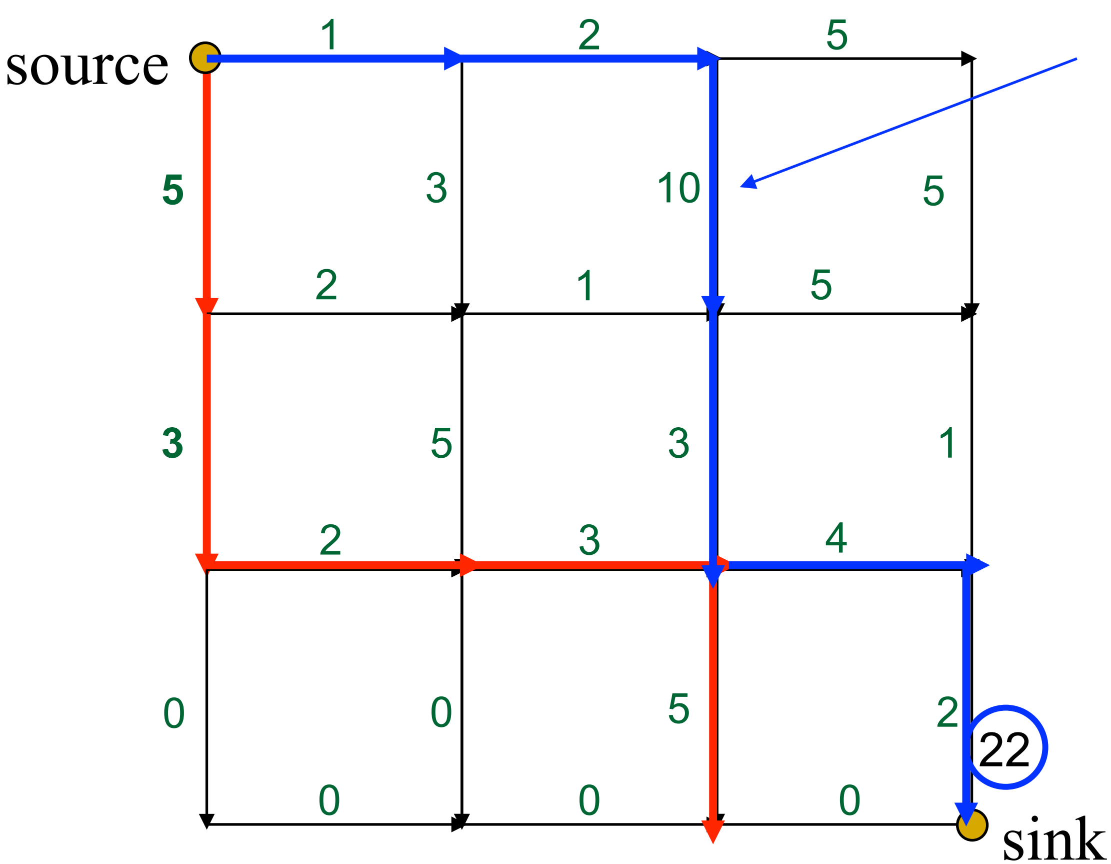


This weight heavily influences the choice of optimal path.

MTP Greedy Algorithm Is Not Optimal

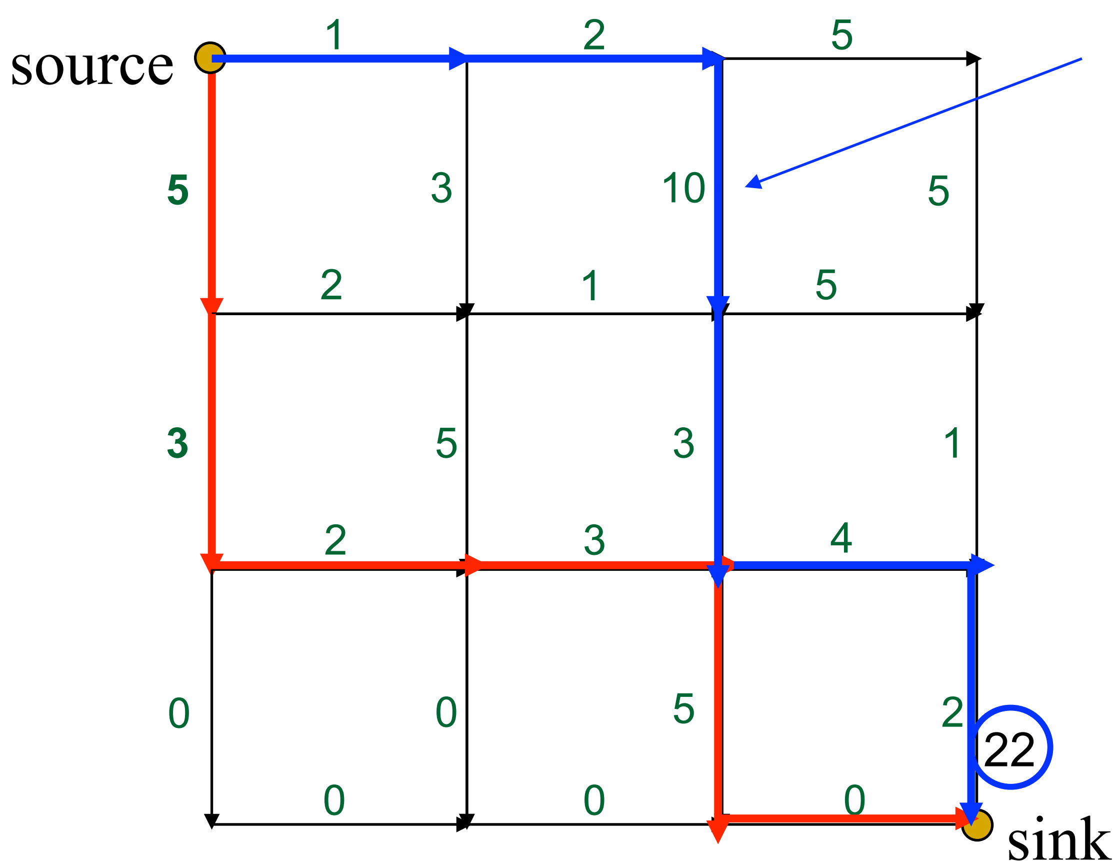


MTP Greedy Algorithm Is Not Optimal



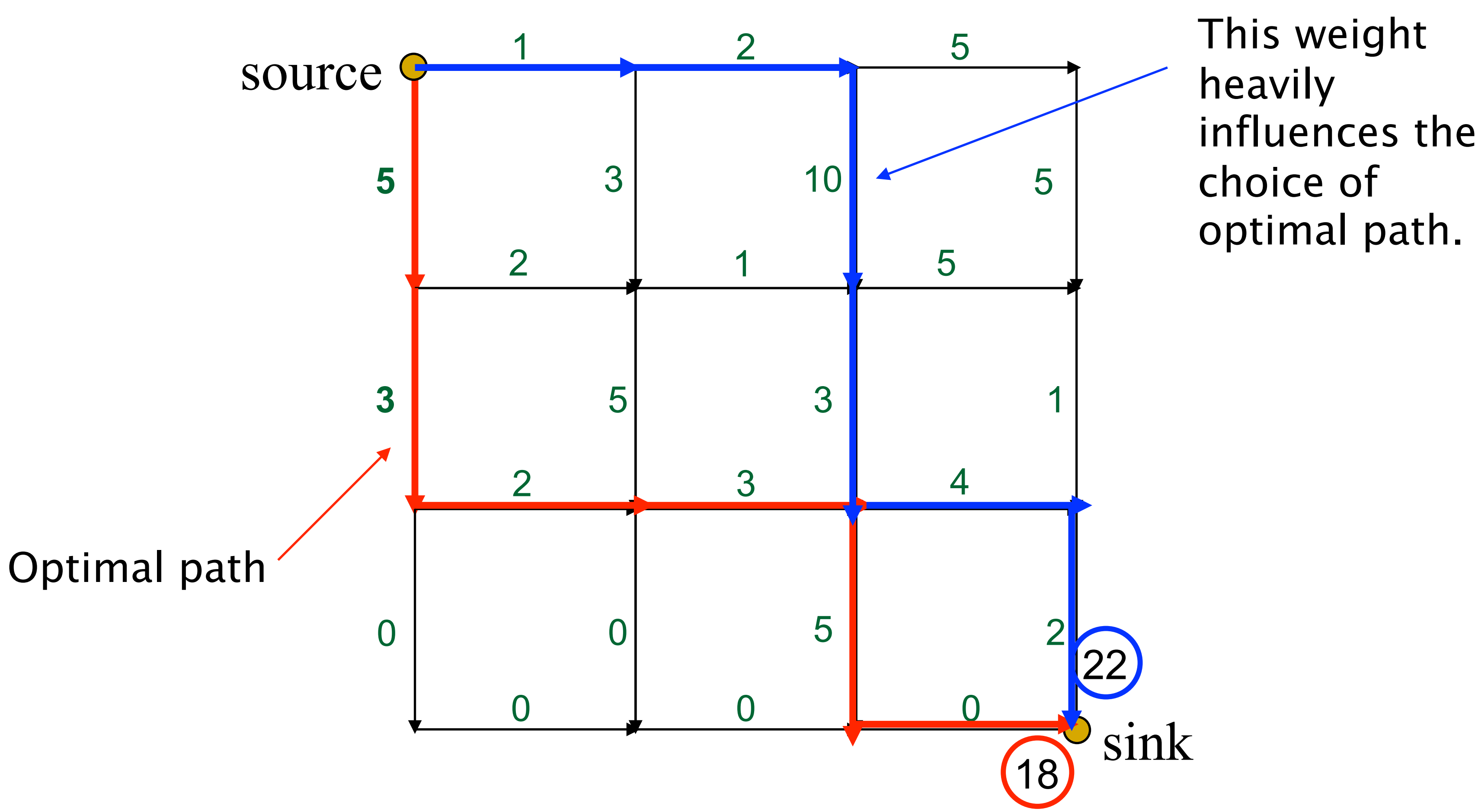
This weight heavily influences the choice of optimal path.

MTP Greedy Algorithm Is Not Optimal



This weight heavily influences the choice of optimal path.

MTP Greedy Algorithm Is Not Optimal



MTP: Simple Recursive Program

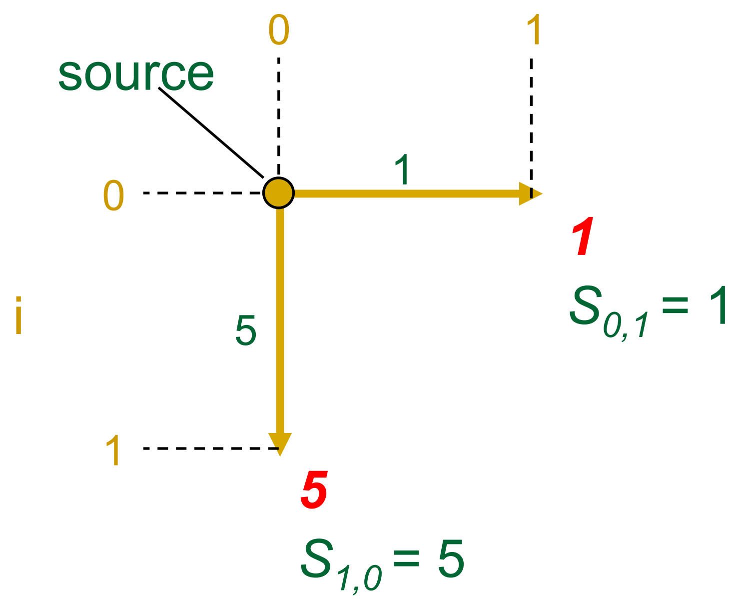
```
MT(n,m)
  if n=0 or m=0
    return MT(n,m)
  x ← MT(n-1,m)+
            length of the edge from (n- 1,m) to (n,m)
  y ← MT(n,m-1)+
            length of the edge from (n,m-1) to (n,m)
  return max{x,y}
```

MTP: Simple Recursive Program

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

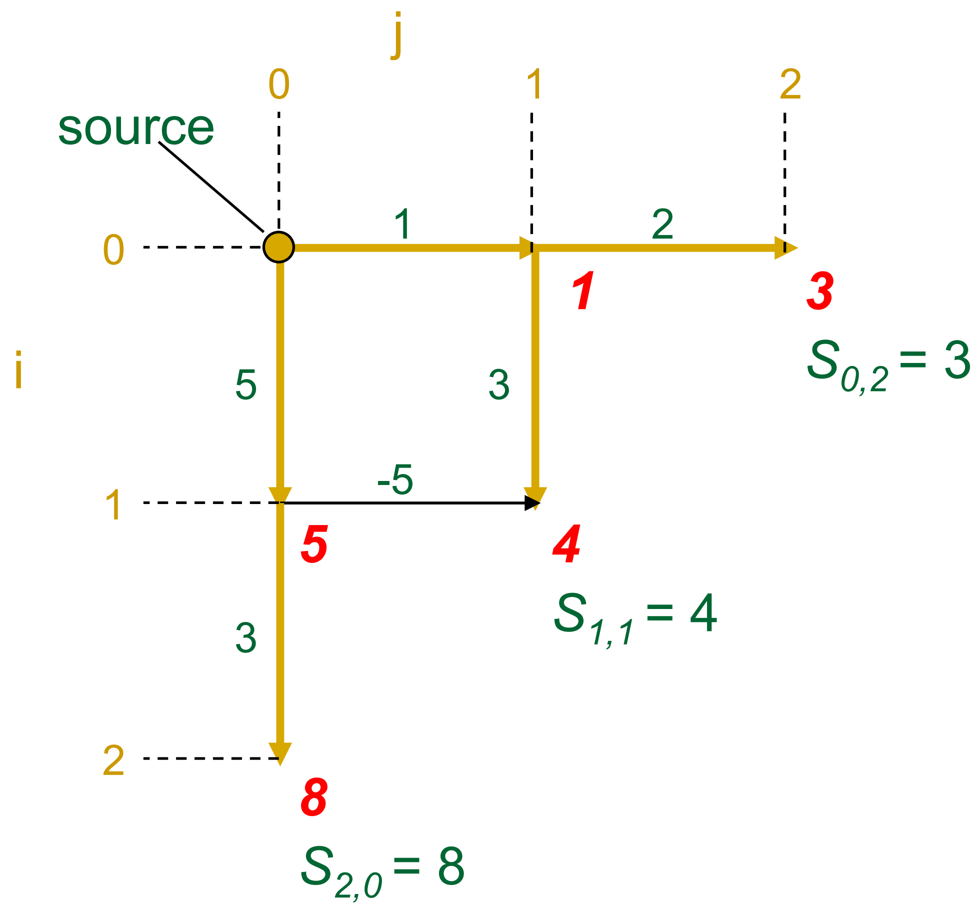
- **What's wrong with this approach?**

MTP: Dynamic Programming



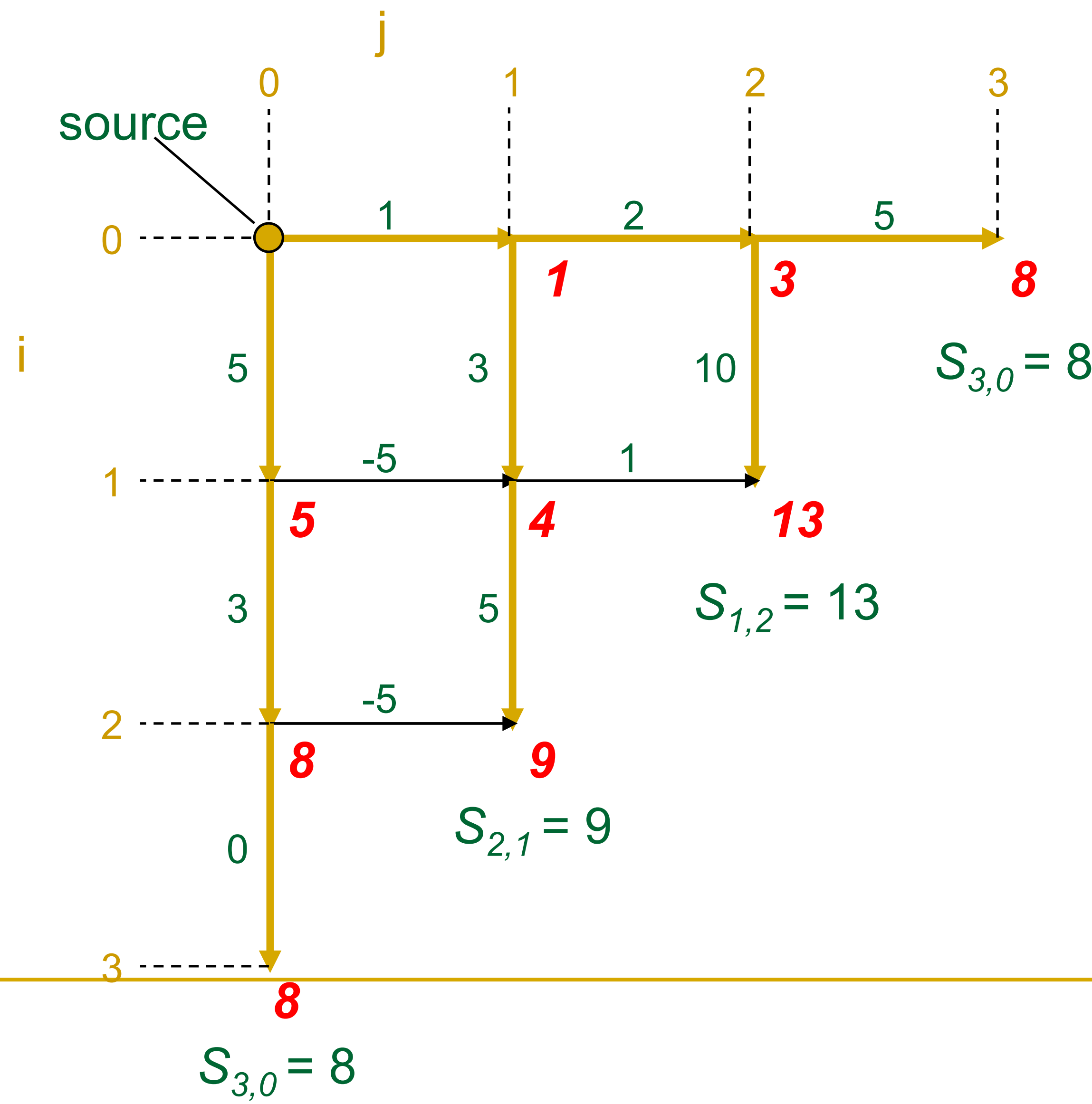
- We calculate the optimal path score for each vertex in the graph.
- A given vertex's score is the maximum sum of incoming edge weight and prior vertex's score (along that incoming edge).

MTP: Dynamic Programming



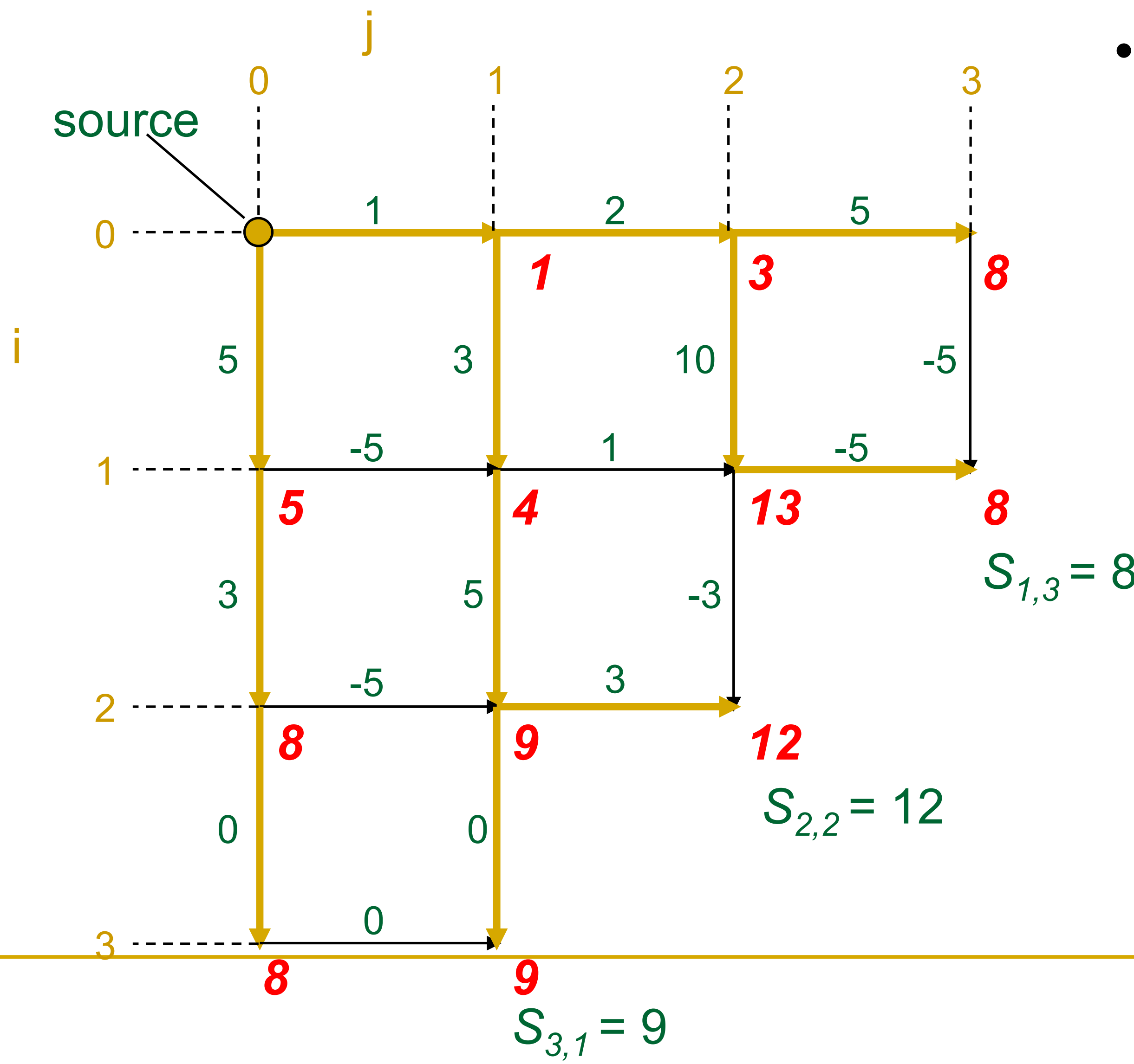
- Gold edges represent edges selected by the algorithm as maximal.

MTP: Dynamic Programming



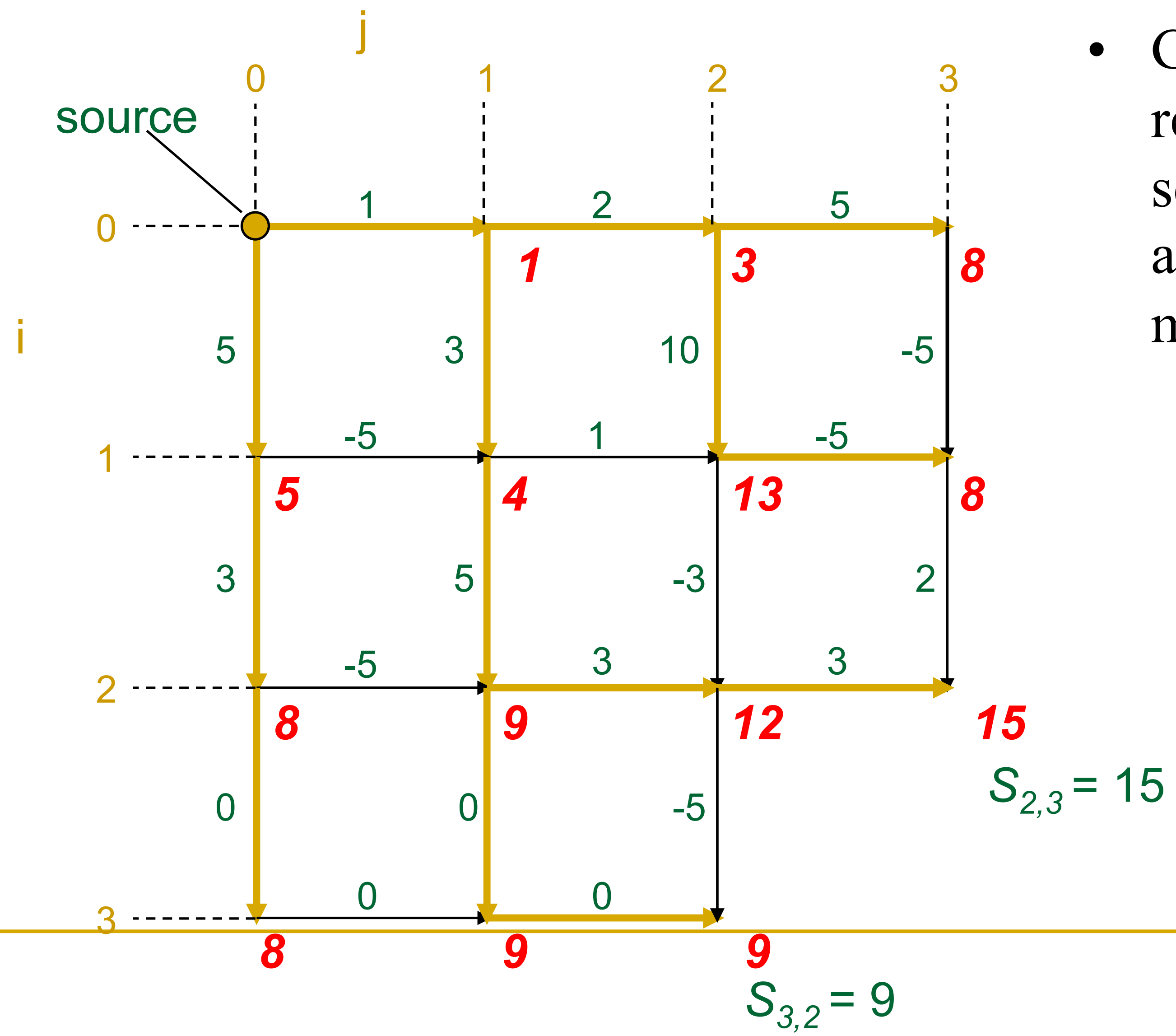
- Gold edges represent edges selected by the algorithm as maximal.

MTP: Dynamic Programming



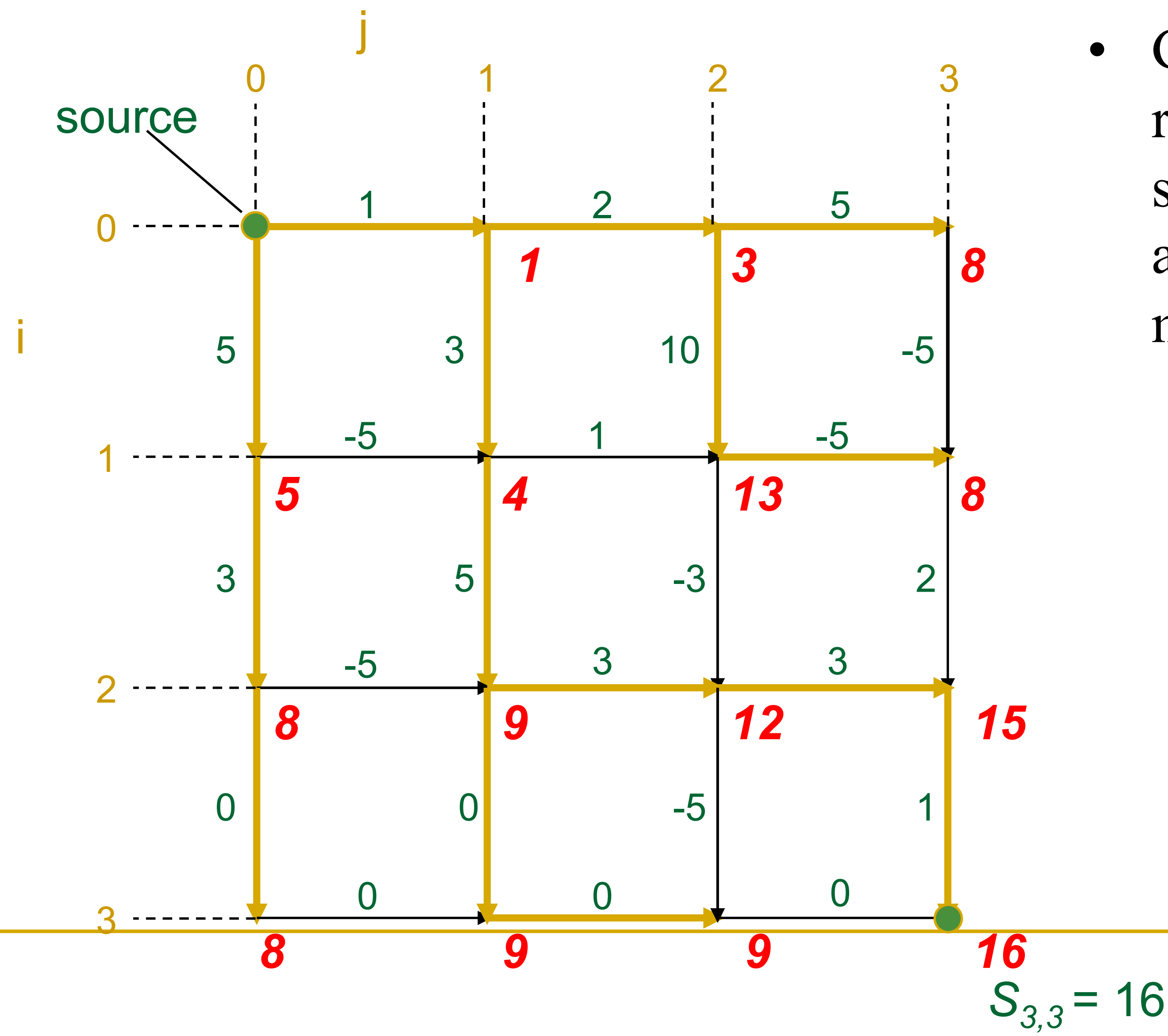
- Gold edges represent edges selected by the algorithm as maximal.

MTP: Dynamic Programming



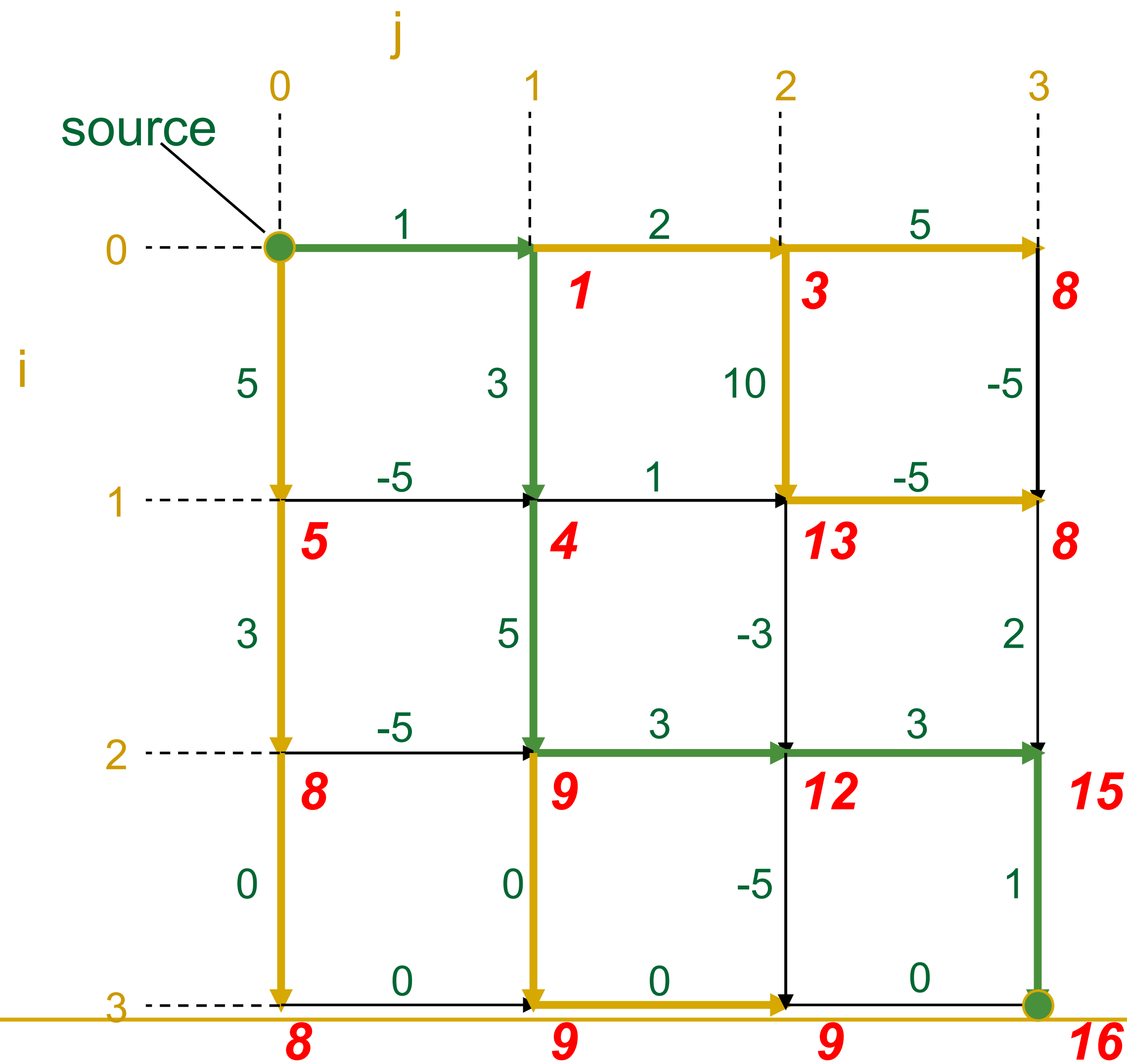
- Gold edges represent edges selected by the algorithm as maximal.

MTP: Dynamic Programming



- Gold edges represent edges selected by the algorithm as maximal.

MTP: Dynamic Programming



- Gold edges represent edges selected by the algorithm as maximal.
- Once we reach the sink, we backtrack along gold edges to the source to find the optimal (green) path.

MTP: Running Time with Dynamic Programming

- The score $s_{i,\,j}$ for a point *(i,j)* is given by the recurrence:

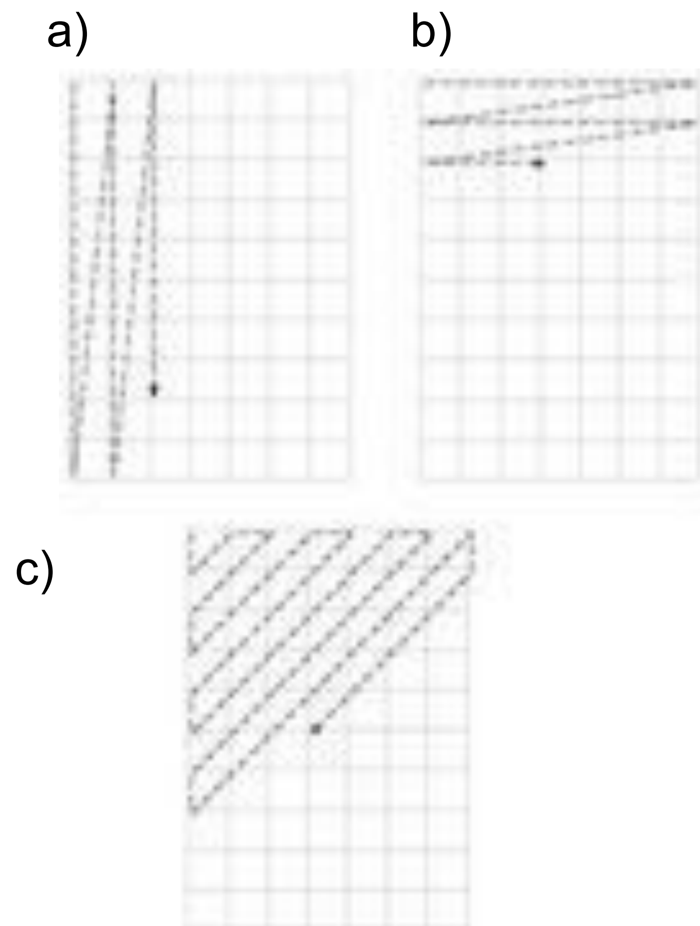
$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of edge between } (i-1, j) \text{ and } (i, j) \\ s_{i,j-1} + \text{weight of edge between } (i, j-1) \text{ and } (i, j) \end{cases}$$

- The running time is ***n x m*** for an ***n*** by ***m*** grid.
    - (***n*** = # of rows, ***m*** = # of columns)

Traversing the Manhattan Grid

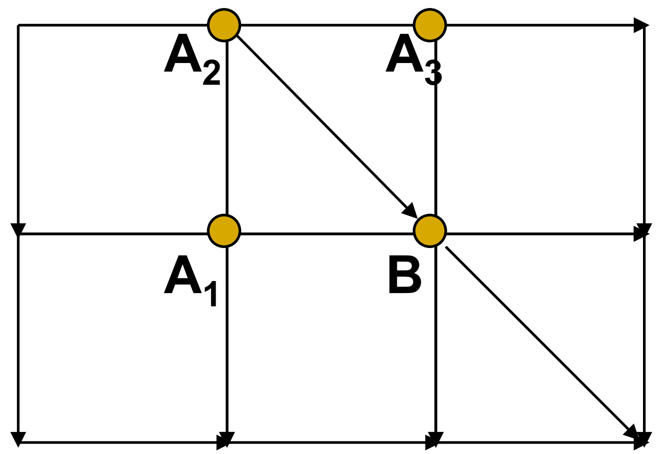
- So that we don't repeat ourselves, we need a strategy for actually traversing through the Manhattan grid to calculate the distances.

- Three common strategies:
  a) Column by column
  b) Row by row
  c) Along diagonals



Manhattan Is Not a Rectangular Grid

- What about diagonals?



- The score at point **B** is given by the recurrence:

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

Section 4:
Longest Path in a Graph

Recursion for an Arbitrary Graph

- We would like to compute the score for point ***x*** in an arbitrary graph.

- Let *Predecessors*(***x***) be the set of vertices with edges leading into ***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 ***E*** edges is O(***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.

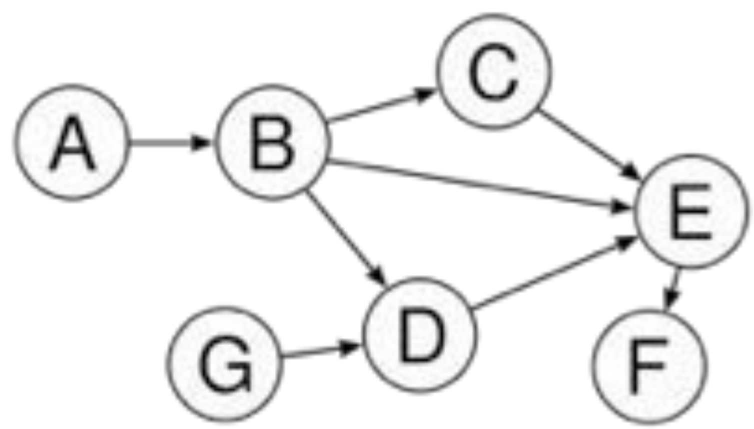
- In the Manhattan graph, we escaped this problem by requiring that we could only move east or south. This is what we would like to generalize…

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.



http://commons.wikimedia.org/wiki/File:Directed_acyclic_graph.svg

Some Graph Theory Terminology

- **Topological Ordering**: A labeling of the vertices of a DAG (from 1 to n, say) such that 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, then all edges go from left to right.

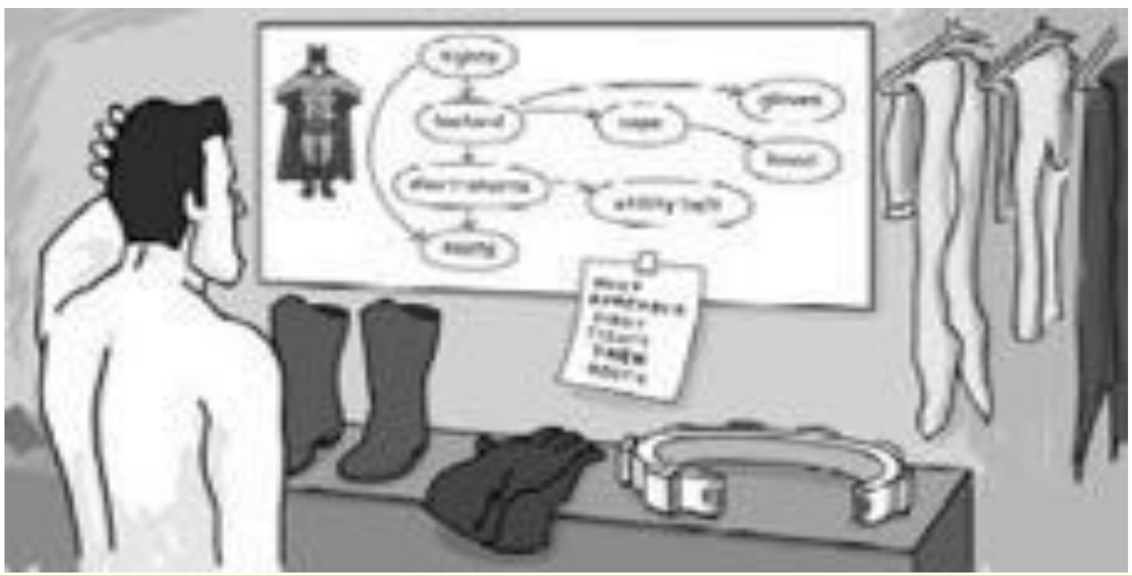
- **Theorem**: Every DAG has a topological ordering.

- What this means: Every DAG has a source node (1) and a sink node (n).

Topological Ordering: Example

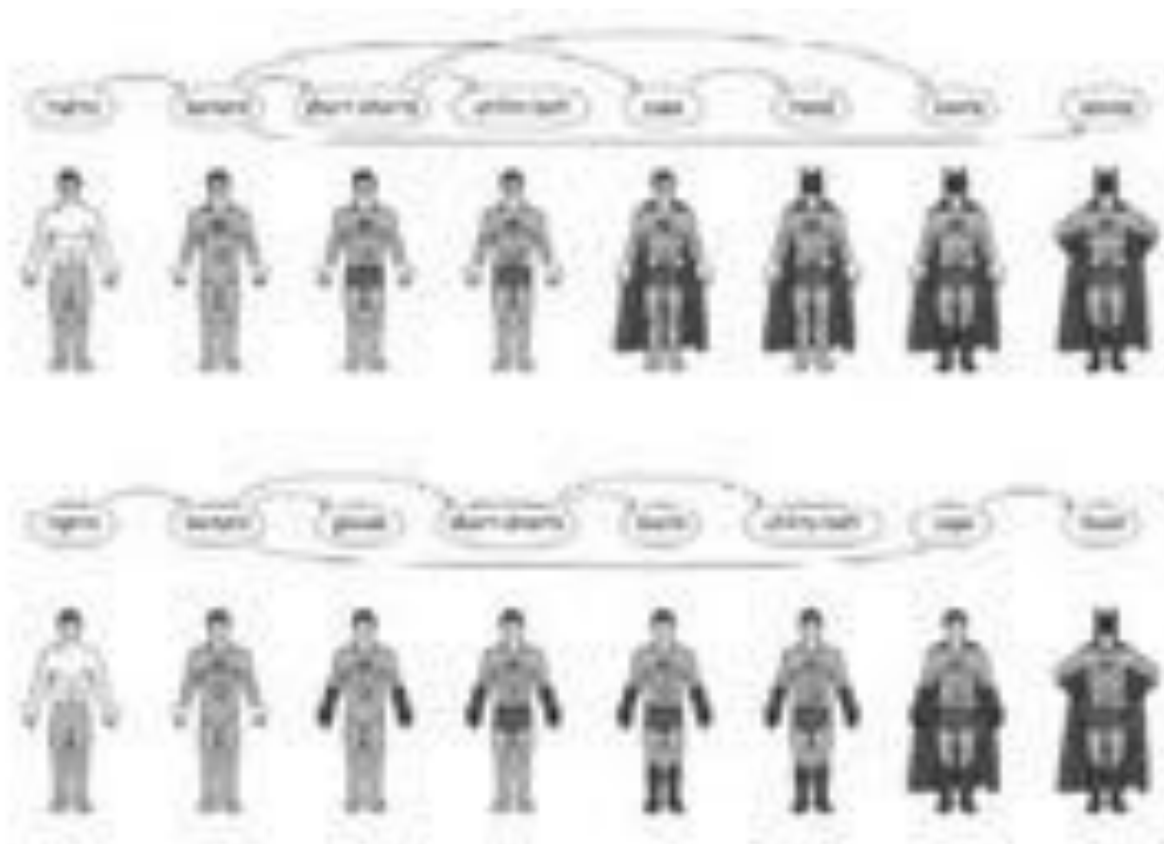
- A superhero’s costume can be represented by a DAG: he can’t put his boots on before his tights!

- 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 ***G*** with source and sink vertices.
- Output: A longest path in ***G*** from source to sink.
- **Note**: Now we know that we can apply a topological ordering to ***G***, and then use dynamic programming to find the longest path in ***G.***