Lecture 8: Dynamic Programming Preliminaries

Study Chapter 6.1-6.3

Revisit Coin-Change Problem

- So far we've tried: A greedy algorithm that does not work for all inputs (it is incorrect)
- New tricks we've learned…
  - Is there an exhaustive search algorithm?

Coin-change problem

```
def exhaustiveChange(amount, denominations):
   bestN = 100
   count = [0 for i in xrange(len(denominations))]
   while True:
      for i, coinValue in enumerate(denominations):
         count[i] += 1
         if (count[i]*coinValue < 100):
            break
         count[i] = 0
      n = sum(count)
      if n == 0:
         break
      value = sum([count[i]*denominations[i] for i in xrange(len(denominations))])
      if (value == amount):
         if (n < bestN):
            bestN = n
   return bestN
```

How many coin combinations does this routine test?

Revisit Coin-Change Problem

- Other Tricks? A branch-and-bound algorithm

```
def branchAndBoundChange(amount, denominations):
   bestN = amount
   count = [0 for i in xrange(len(denominations))]
   while True:
      for i, coinValue in enumerate(denominations):
         count[i] += 1
         if (count[i] * coinValue < amount):
            break
         count[i] = 0
      n = sum(count)
      if n == 0:
         break
      if (n > bestN):
         continue
      value = sum([count[i] * denominations[i] for i in xrange(len(denominations))])
      if (value == amount):
         if (n < bestN):
            bestN = n
   return bestN
```

Coin-change problem

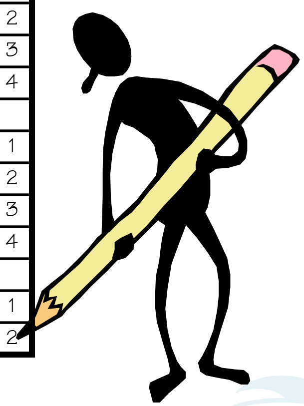
  – Correct, and works well for *most* cases, but might be as slow as an exhaustive search for some inputs.

- Is there anything else we can try?

Tabulating the Answers

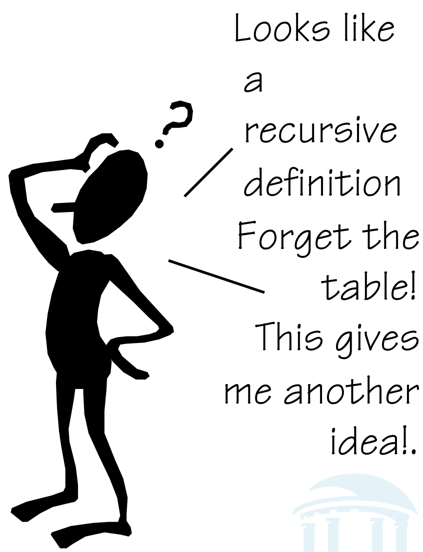
- If it is costly to compute the answer for a given input, then there may be advantages to *caching* the result of previous calculations in a table
- This trades-off time-complexity for space
- How could we fill in the table in the first place?
  - Run our best correct algorithm
  - Can the table itself be used to speed up the process?

Amt	25	20	10	5	1	Amt	25	20	10	5	1
1¢					1	42¢		2			2
2¢					2	43¢		2			3
3¢					3	44¢		2			4
4¢					4	45¢		2		1	
5¢				1		46¢		2		1	1
6¢				1	1	47¢		2		1	2
7¢				1	2	48¢		2		1	3
8¢				1	3	49¢		2		1	4
9¢				1	4	50¢	2				
10¢			1			51¢	2				1
11¢			1		1	52¢	2				2



Solutions using a Table

- Suppose you are asked to fill-in the unknown table entry for 67¢
- It must differ from previous known optimal result by at most one coin…
- So what are the possibilities?
  - BestChange(67¢) = 25¢ + BestChange(42¢), or
  - BestChange(67¢) = 20¢ + BestChange(47¢), or
  - BestChange(67¢) = 10¢ + BestChange(57¢), or
  - BestChange(67¢) = 5¢ + BestChange(62¢), or
  - BestChange(67¢) = 1¢ + BestChange(66¢)



A Recursive Coin-Change Algorithm

```
def RecursiveChange(M, c):
    if (M == 0):
        return [0 for i in xrange(len(c))]
    smallestNumberOfCoins = M+1
    for i in xrange(len(c)):
        if (M >= c[i]):
            thisChange = RecursiveChange(M - c[i], c)
            thisChange[i] += 1
            if (sum(thisChange) < smallestNumberOfCoins):
                bestChange = thisChange
                smallestNumberOfCoins = sum(thisChange)
    return bestChange
```

- The only problem is… this is still too slow
- Let's see why…

Recursion Recalculations

- We saw this before with RecursiveFibonacci( )
- Recursion often results in many redundant calls
- Even after only two levels of recursion 6 different change values are repeated multiple times
- How can we avoid this repetition?
- Cache precomputed results in a table!

```
Change(40) =   25 + Change(15)
                    25 + 10 + Change(5)
                    25 +  5 + Change(10)
               20 + Change(20)
                    20 + 20 + Change(0)
                    20 + 10 + Change(10)
                    20 +  5 + Change(15)
               10 + Change(30)
                    10 + 25 + Change(5)
                    10 + 20 + Change(10)
                    10 + 10 + Change(20)
                    10 +  5 + Change(25)
               5 + Change(35)
                    5 + 25 + Change(15)
                    5 + 20 + Change(10)
                    5 + 10 + Change(25)
                    5 +  5 + Change(30)
```

Back to Table Evaluation

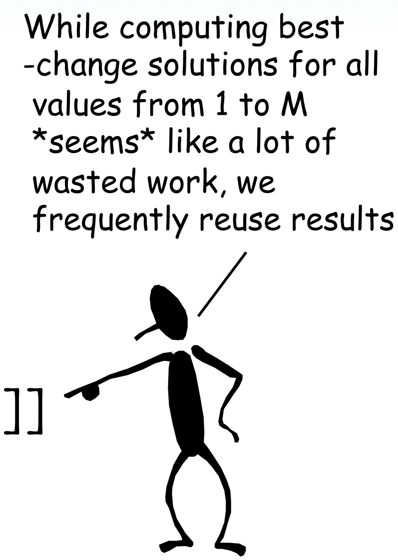
- When do we fill in the values of the table?
- We could do it lazily as needed… as each call to BestChange() progresses from M down to 1
- Or we could do it from the bottom-up – tabulating all values from 1 up to M
- Thus, instead of just trying to find the minimal number of coins to change M cents, we attempt the solve the superficially harder problem of solving for the optimal change for all values from 1 to M

1¢ = [0,0,0,0,1] | 2¢ = [0,0,0,0,2] | 3¢ = [0,0,0,0,3] | . . . | M¢ = [?,?,?,?,?]

Change via Dynamic Programming

```
def DPChange(M, c):
   change = [[0 for i in xrange(len(c))]]
   for m in xrange(1,M+1):
      bestNumCoins = m+1
      for i in xrange(len(c)):
         if (m >= c[i]):
            thisChange = [x for x in change[m - c[i]]]
            thisChange[i] += 1
            if (sum(thisChange) < bestNumCoins):
               change[m:m] = [thisChange]
               bestNumCoins = sum(thisChange)
   return change[M]
```

M d



- Recall, BruteForceChange( ) was $O(M^d)$
- DPChange( ) is O(Md)

Dynamic Programming

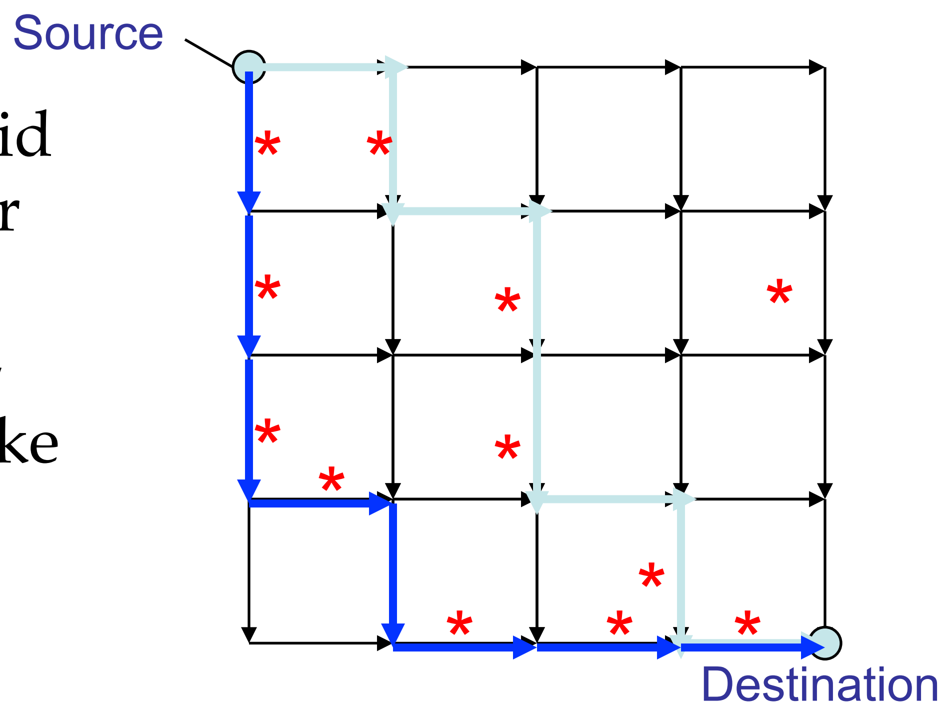
- ***Dynamic Programming*** is a technique for computing recurrence relations efficiently by storing partial or intermediate results

- Three keys to constructing a dynamic programming solution:
  1. Formulate the answer as a recurrence relation
  2. Consider all instances of the recurrence at each step
  3. Order evaluations so you will always have precomputed the needed partial results

Yet Another DP Example

Manhattan Tourist Problem (MTP)

Imagine seeking a path from source to destination in a Manhattan-like city grid that maximizes the number of attractions (*) passed. With the following caveat– at every step you must make progress towards the goal.



We treat the city map as a graph, with a "vertices" at each corner, and weighted edges along each block. The weights are the number of attractions along each block.

Manhattan Tourist Problem: Formulation

Goal: Find the maximum weighted path in a grid.

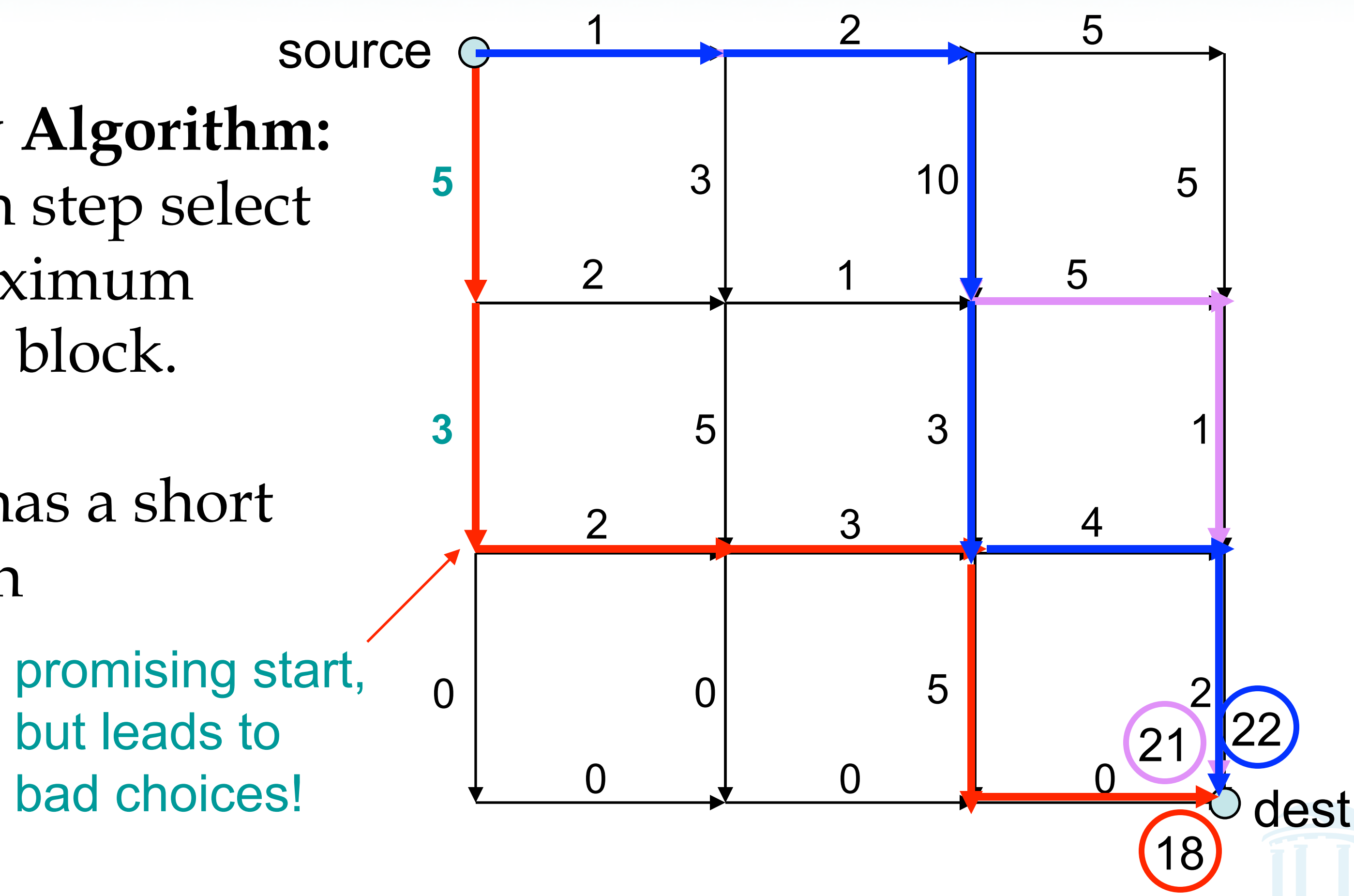
Input: A weighted grid ***G*** with two distinct vertices, one labeled "*source*" and the other labeled "*destination*"

Output: A longest path in ***G*** from "*source*" to "*destination*"

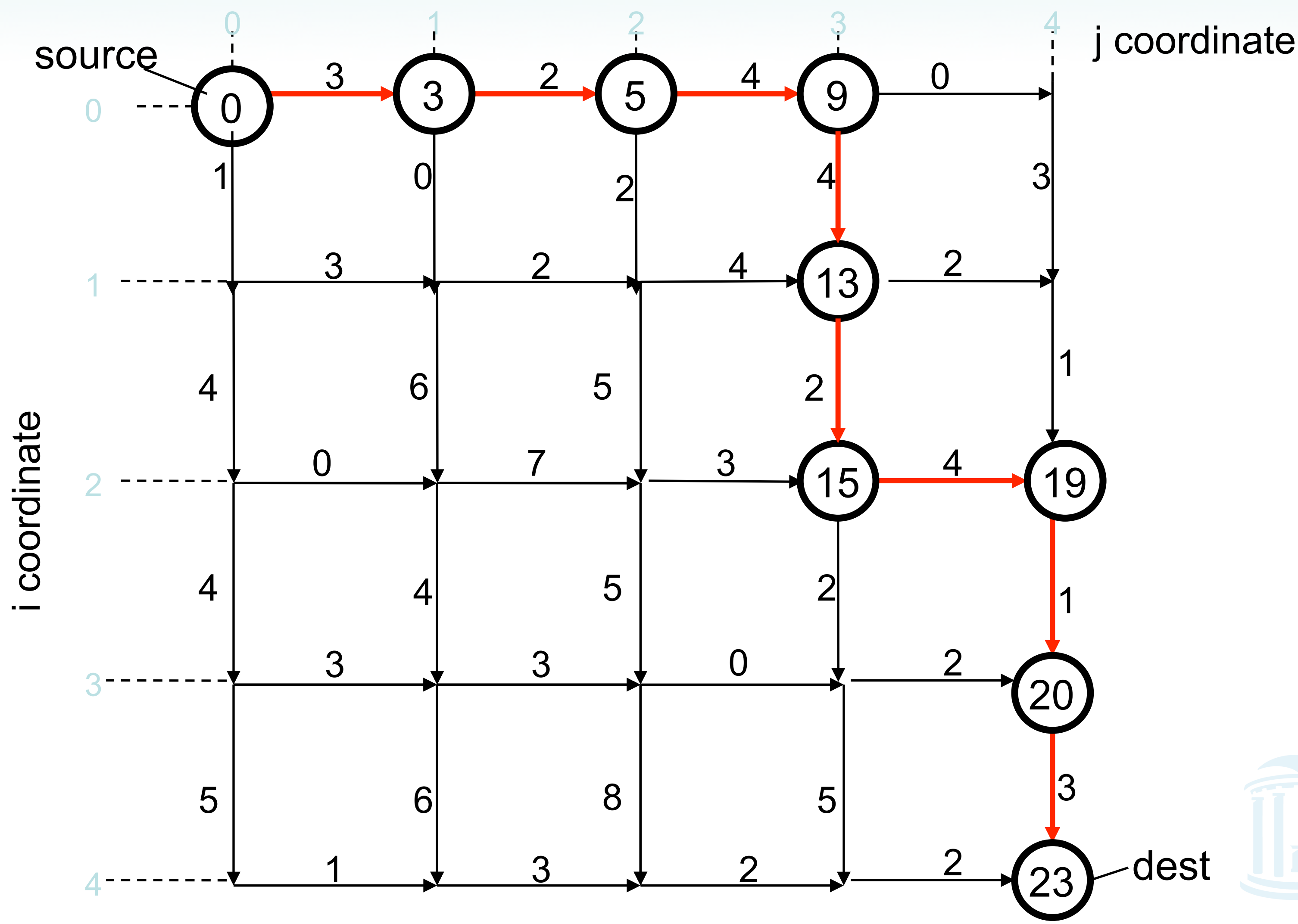
MTP: Greedy Algorithm Is Not Optimal

Greedy Algorithm:
At each step select the maximum weight block.

Greed has a short horizon



MTP as a Dynamic Program



MTP Strategy

- Instead of solving the Manhattan Tourist problem directly, (i.e. the path from (0,0) to (n,m)) we will solve a more general problem: find the longest path from (0,0) to any arbitrary vertex (i,j).

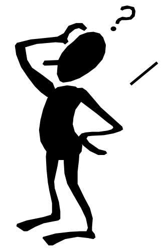
- If the longest path from (0,0) to (n,m) passes through some vertex (i,j), then the path from (0,0) to (i,j) must be the longest. Otherwise, you could increase your path by changing it.

MTP: Simple Recursive Program

What's wrong with this approach?

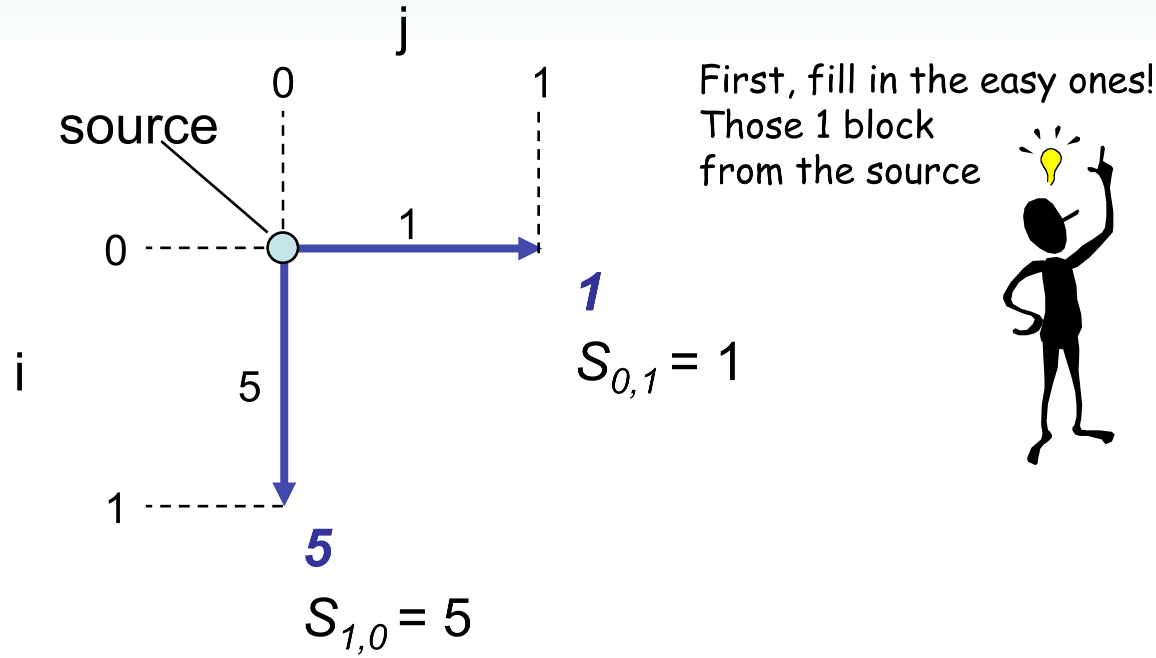
<ins>MT(*n,m*)</ins>

if n = 0 and m = 0
  return 0
if n = 0
  return *MT(0,m-1)* + len(edge) from *(0,m-1) to (0,m)*
if m = 0
  return *MT(n-1, 0)* + len(edge) from *(n-1,0) to (n,0)*
x ← *MT(n-1,m)* + len(edge) from *(n- 1,m) to (n,m)*
y ← *MT(n,m-1)* + len(edge) from *(n,m-1) to (n,m)*
return *max{x,y}*



We saw this in our recursive change algorithm. It computes the same paths multiple times

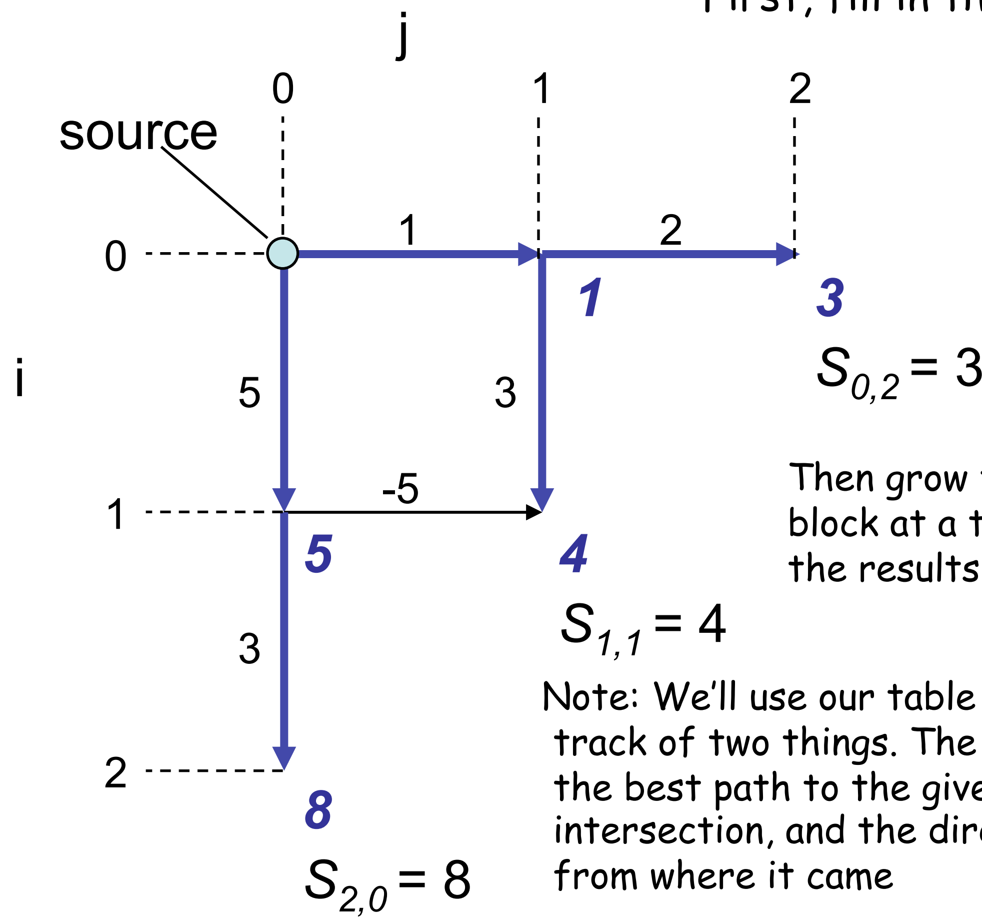
MTP: Ordering Evaluations



- 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 connecting edge in between

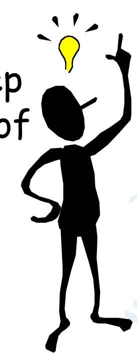
MTP: Dynamic Programming (cont'd)

First, fill in the easy ones!

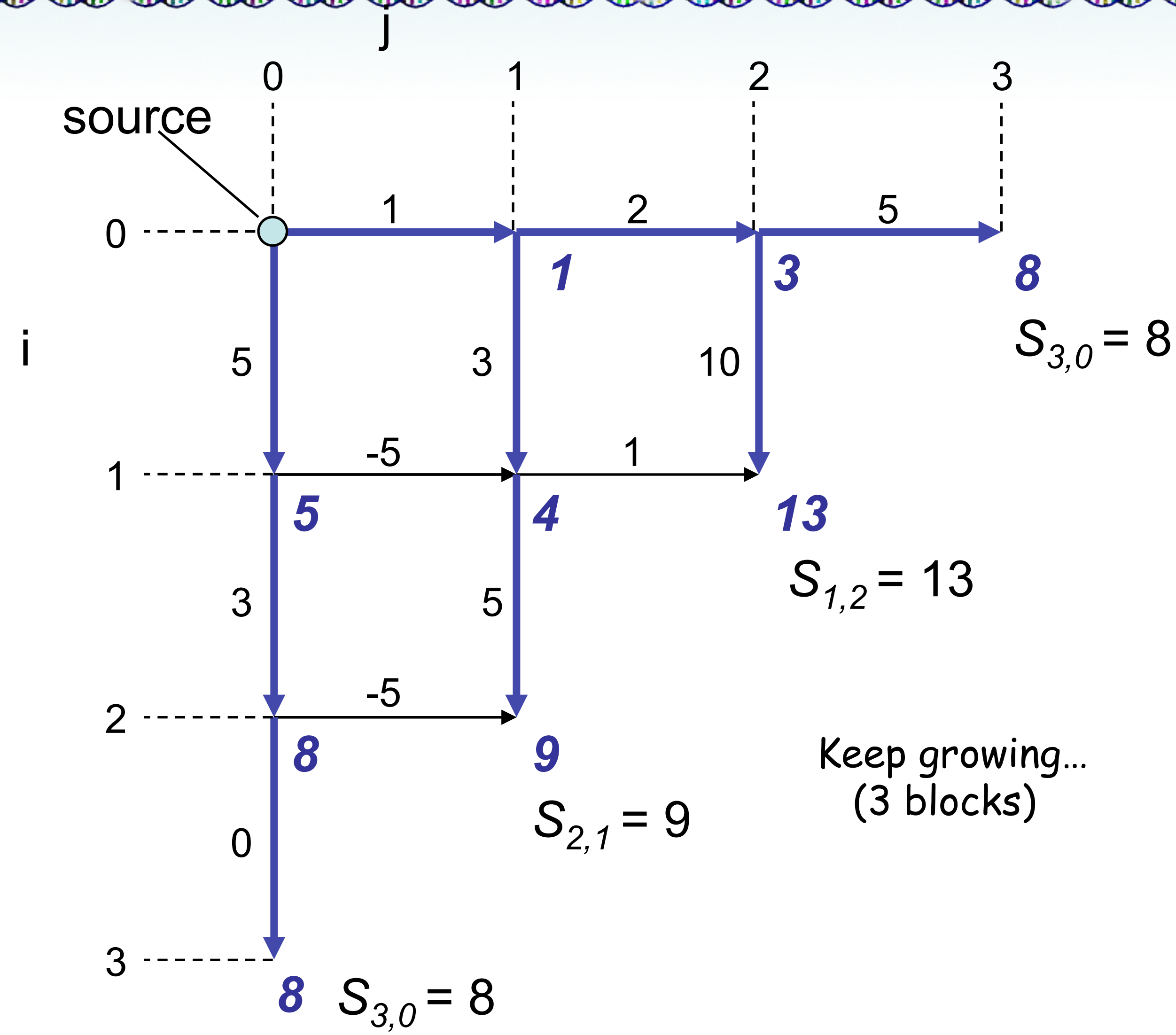


Then grow the solution a block at a time while tabulating the results for each intersection

Note: We'll use our table to keep track of two things. The value of the best path to the given intersection, and the direction from where it came

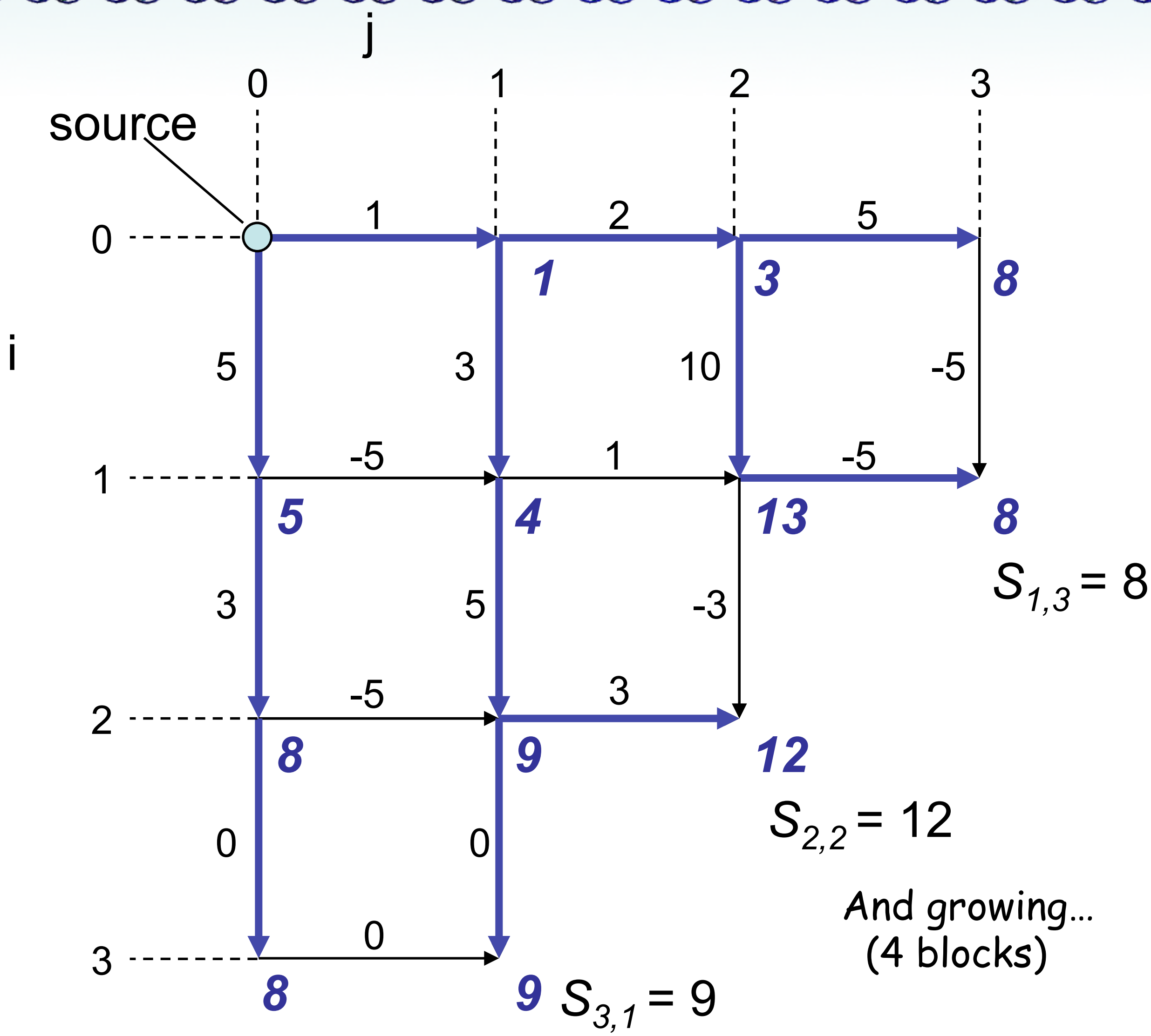


MTP: Dynamic Programming (cont'd)



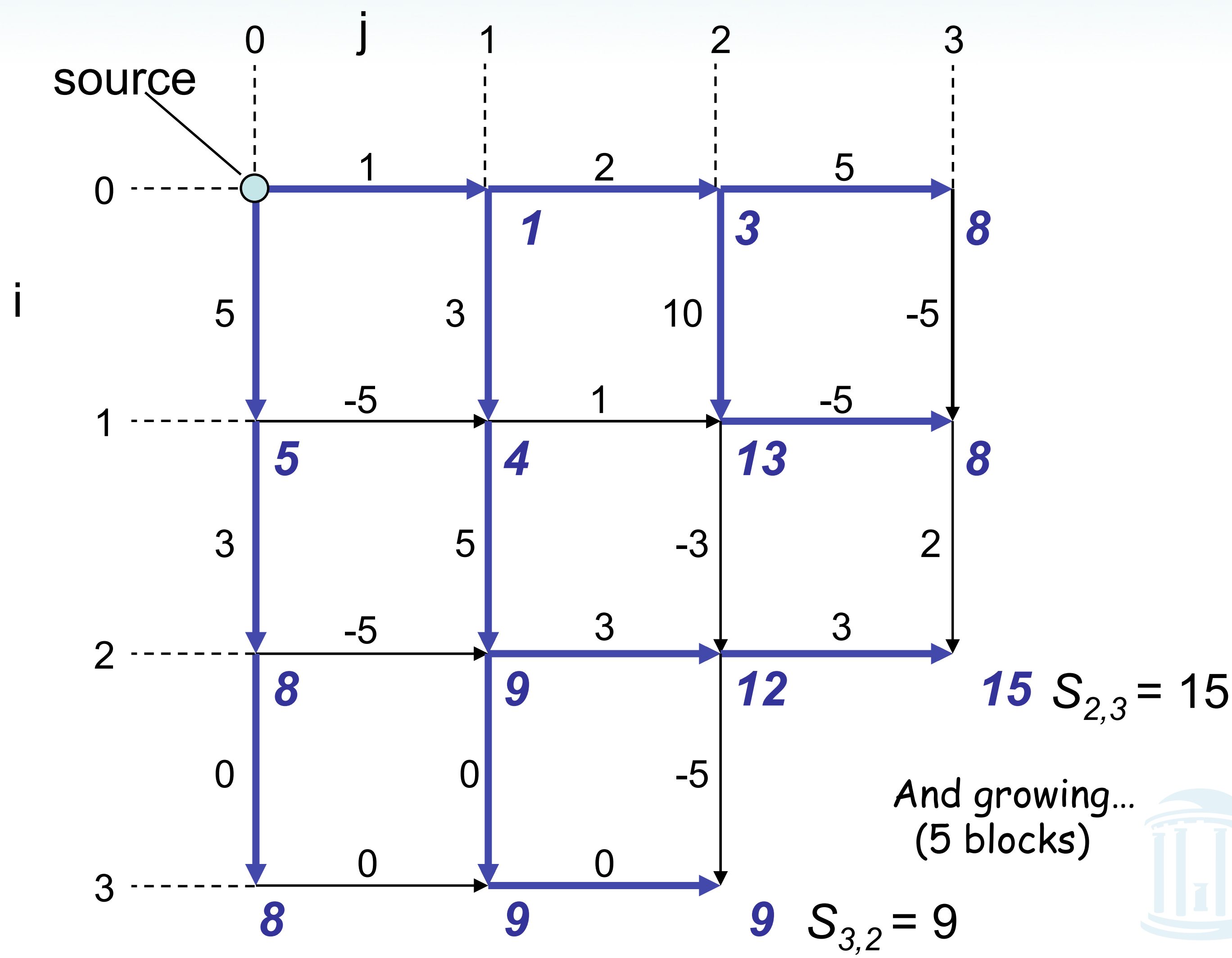
Keep growing…
(3 blocks)

MTP: Dynamic Programming (cont'd)

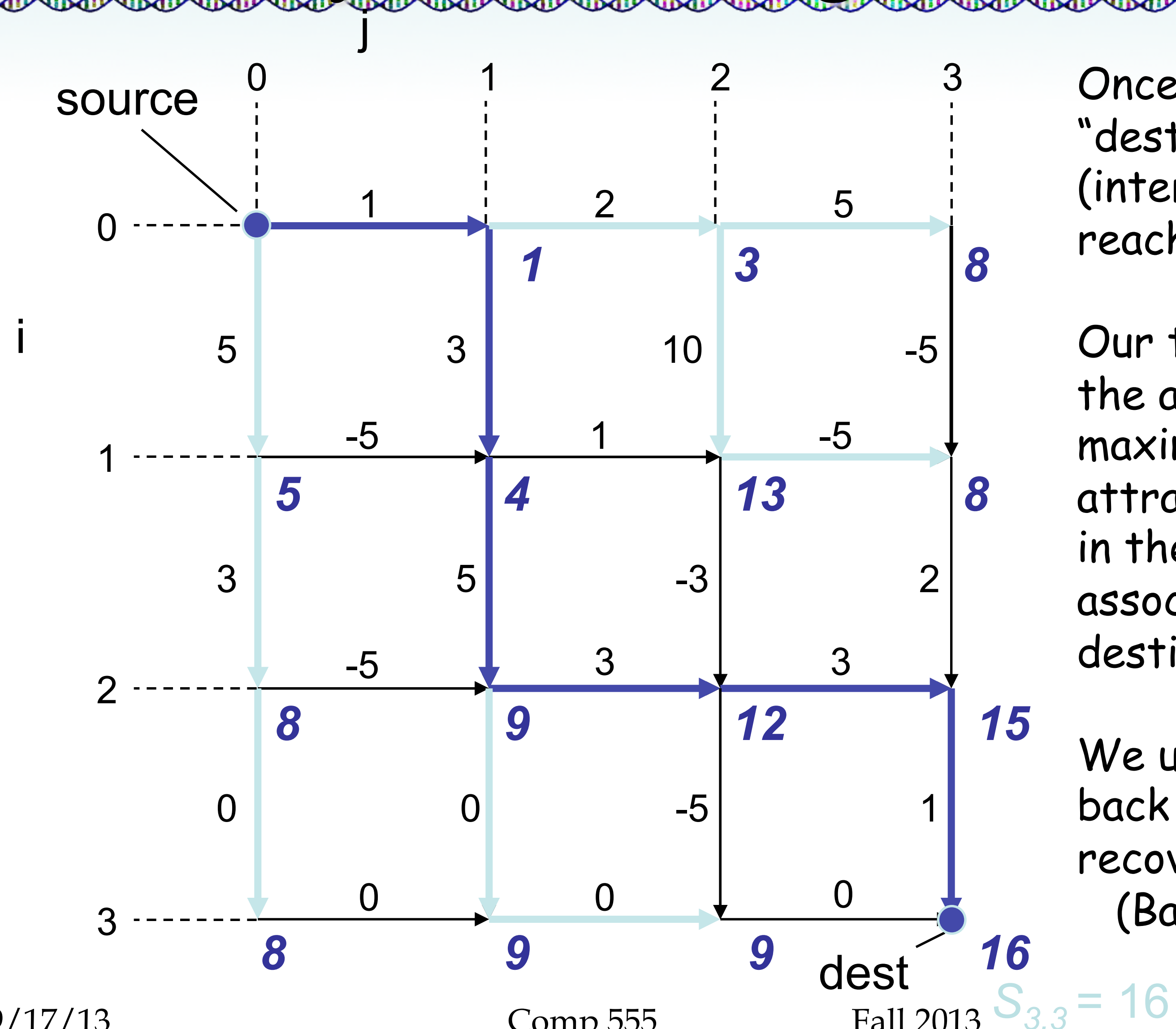


And growing…
(4 blocks)

MTP: Dynamic Programming (cont'd)



MTP: Dynamic Programming (cont'd)



Once the "destination" node (intersection) is reached, we're done.

Our table will have the answer of the maximum number of attractions stored in the entry associated with the destination.

We use the "links" back in the table to recover the path.
(Backtracking)

MTP: Recurrence

Computing the score for a point *(i,j)* by the recurrence relation:

Path to the intersection from the left

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

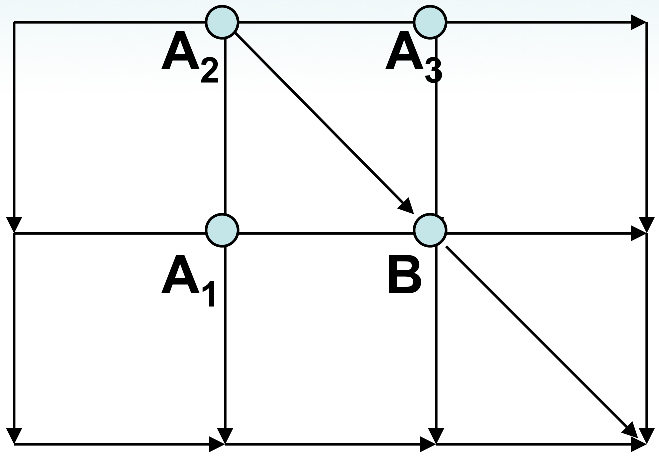
Path to the intersection from above

The running time is ***n x m*** for a ***n*** by ***m*** grid

(You visit all intersections once, and performed 2 tests)

(***n*** = # of rows, ***m*** = # of columns)

Manhattan Is Not A Perfect Grid



What about diagonals?

Broadway, Greenwich, etc.

- Easy to fix. Just adds more recursion cases.
- The score at point B is given by:

$$s_B = \max \begin{cases} s_{A1} + \text{weight of the edge } (A_1, B) \\ s_{A2} + \text{weight of the edge } (A_2, B) \\ s_{A3} + \text{weight of the edge } (A_3, B) \end{cases}$$

Generalizing Manhattan to a Directed Graph

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

$$s_x = \max_{\text{of}} \begin{cases} s_y + \text{weight of vertex } (y, x) \text{ where} \\ \qquad\qquad y \text{ in Predecessors}(x) \end{cases}$$

- Predecessors (x) – set of vertices having edges leading to x
- The running time for a graph G(***V***, ***E***) (***V*** is the set of all vertices and ***E*** is the set of all edges) is O(***E***) since each edge is considered once

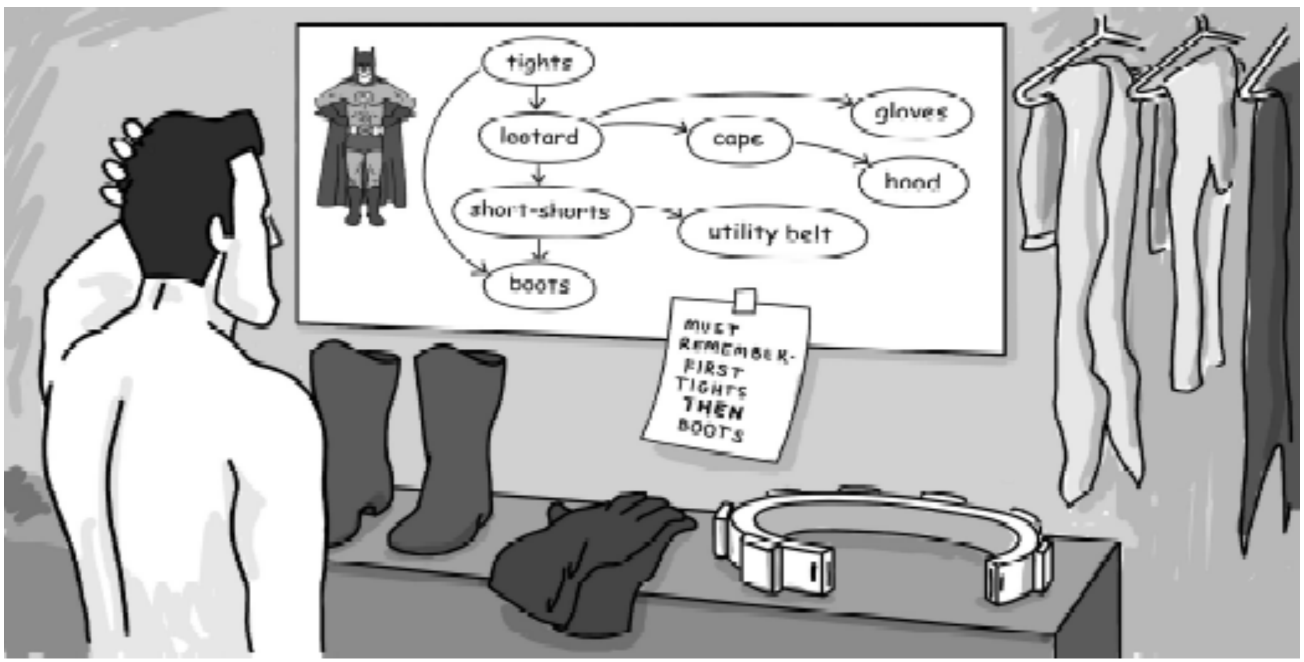
Traveling in the Grid

- The only hitch is that one must decide on an *order* to visit the vertices
- We must assure that 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
- How to find such order for any directed graph?

???

DAG: Directed Acyclic Graph

- Since most cities are not perfect regular grids, we represent paths in them as a DAGs
- DAG for *Dressing in the morning* problem



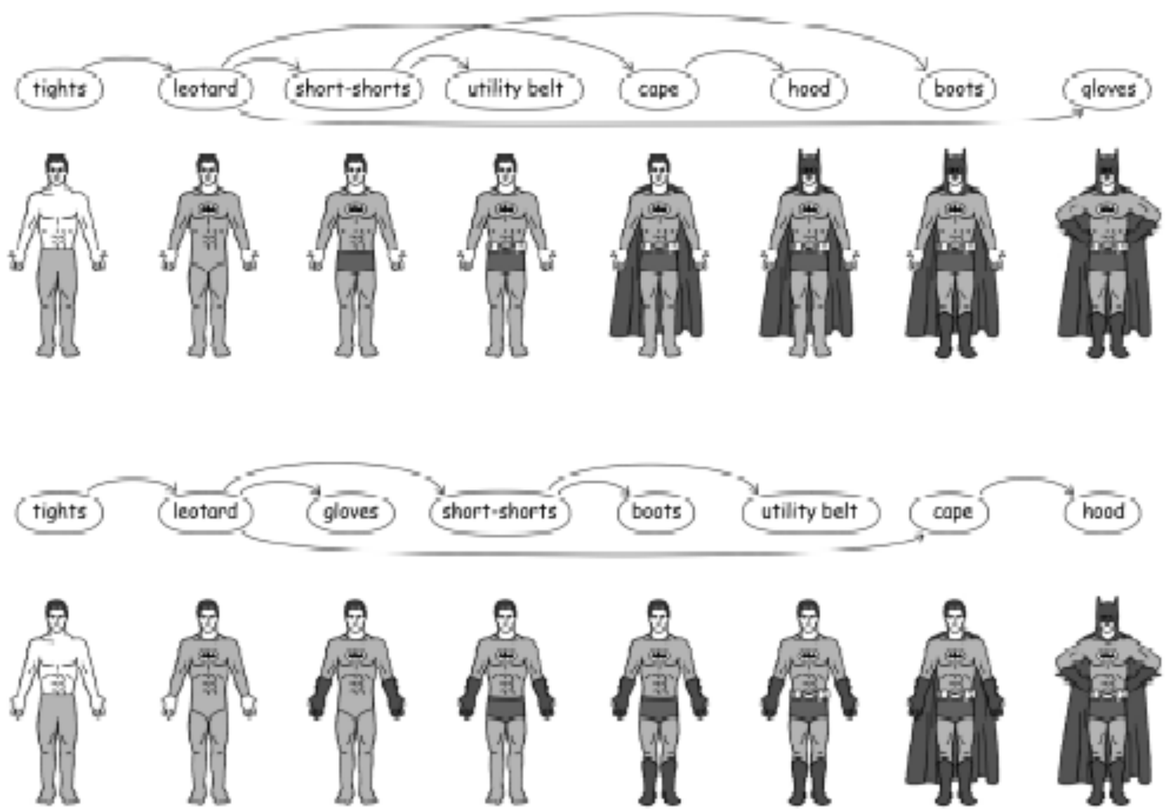
Topological Ordering

- 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

- <u>Goal</u>: Find a longest path between two vertices in a weighted DAG
- <u>Input</u>: A weighted DAG ***G*** with source and destination vertices
- <u>Output</u>: A longest path in ***G*** from source to destination

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:

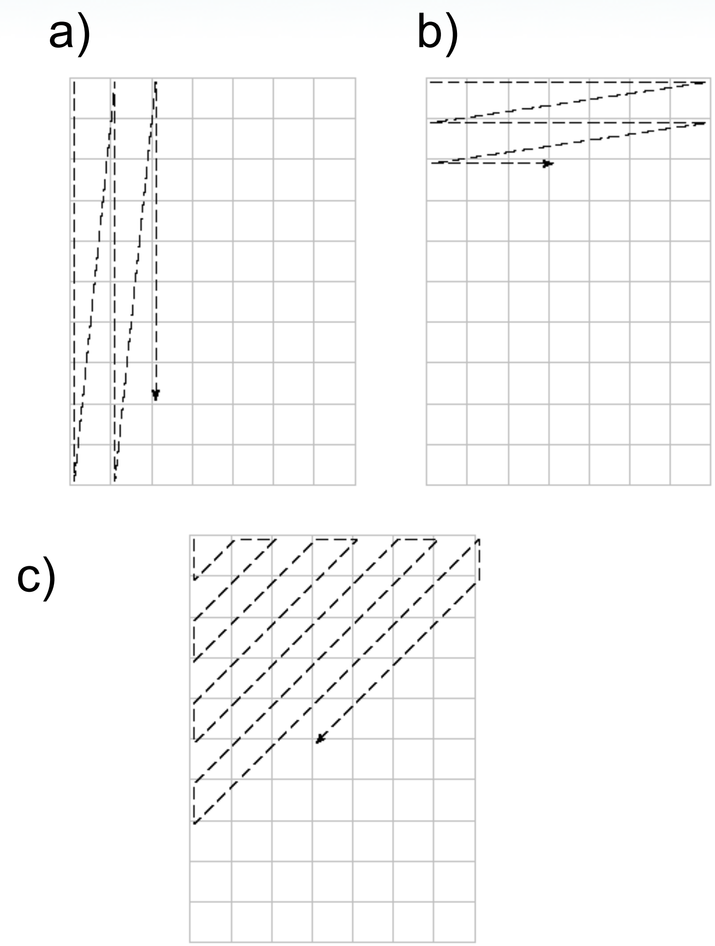
$$s_v = \textbf{max of} \begin{cases} s_{u_1} + \text{weight of edge from } u_1 \text{ to } v \\ s_{u_2} + \text{weight of edge from } u_2 \text{ to } v \\ s_{u_3} + \text{weight of edge from } u_3 \text{ to } v \end{cases}$$

<u>In General</u>:

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

Traversing the Manhattan Grid

- We chose to evaluate our table in a particular order. Uniform distances from the source (all points one block away, then 2 blocks, etc.)
- Other strategies:
  - a) Column by column
  - b) Row by row
  - c) Along diagonals
- This choice can have performance implications



Next Time

- Return to biology
- Solving sequence alignments using Dynamic Programming