

# **Data Structures and Algorithms**

**Session 18. April 1, 2009**

**Instructor: Bert Huang**

**<http://www.cs.columbia.edu/~bert/courses/3137>**

# Announcements

- \* Homework 4 due next class
- \* Huffman compression must handle any characters in dictionary.txt
- \* Spell checker can ignore case

# Review

- \* Rehashing
- \* String hash function example
- \* Graphs
  - \* Terminology and properties
  - \* Implementation

# Today's Plan

- \* Topological Sort
- \* Shortest Path
  - \* Unweighted version
  - \* Weighted version

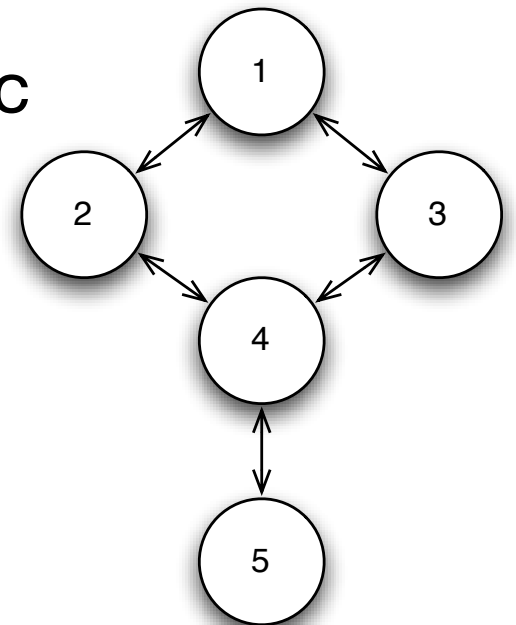
# Implementation

- \* Option 1:
  - \* Store all nodes in an indexed list
  - \* Represent edges with **adjacency matrix**
- \* Option 2:
  - \* Explicitly store **adjacency lists**

# Adjacency Matrices

- \* 2d-array **A** of boolean variables
- \*  $A[i][j]$  is true when node **i** is adjacent to node **j**
- \* If graph is undirected, **A** is symmetric

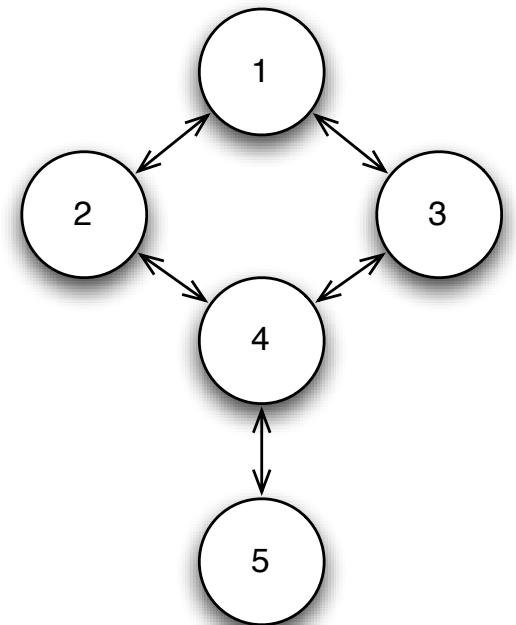
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	0
4	0	1	1	0	1
5	0	0	0	1	0



# Adjacency Lists

- ✱ Each node stores references to its neighbors

1	2	3		
2	1	4		
3	1	4		
4	2	3	5	
5	4			



# Math Notation for Graphs

- \* Set Notation:

- \*  $v \in V$  (v is in V)

- \*  $U \cup V$  (union)

- \*  $U \cap V$  (intersection)

- \*  $U \subset V$   
(U is a subset of V)

- \*  $G = \{V, E\}$

- \* G is the graph

- \* V is set of vertices

- \* E is set of edges

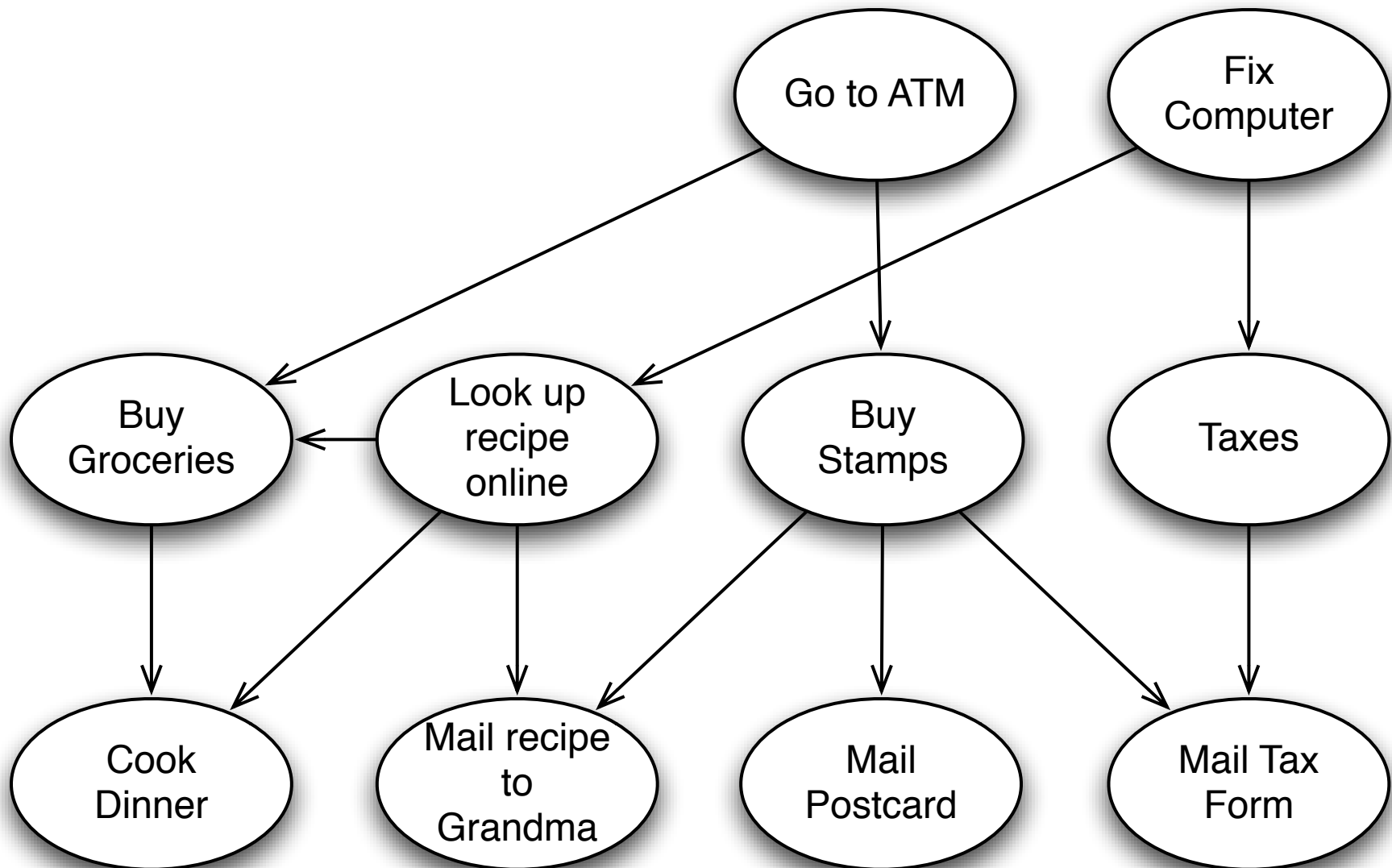
- \*  $(v_i, v_j) \in E$

- \*  $|V| = N = \text{size of } V$

# Topological Sort

- \* Problem definition:
  - \* Given a directed acyclic graph  $G$ , order the nodes such that for each edge  $(v_i, v_j) \in E$ ,  $v_i$  is before  $v_j$  in the ordering.
  - \* e.g., scheduling errands when some tasks depend on other tasks being completed.

# Topological Sort Ex.

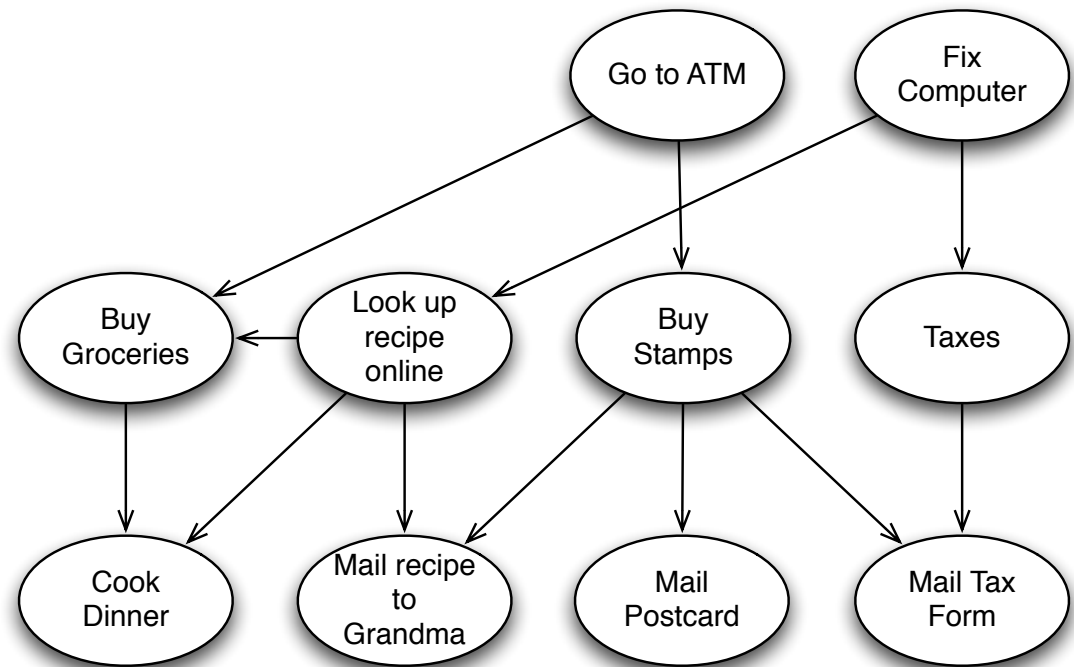


# Topological Sort Naïve Algorithm

- \* **Degree** means # of edges,  
**indegree** means # of incoming edges
- \* 1. Compute the **indegree** of all nodes
- \* 2. Print any node with indegree 0
- \* 3. Remove the node we just printed. Go to 1.
- \* Which nodes' indegrees change?

# Topological Sort Better Algorithm

- \* 1. Compute all indegrees
- \* 2. Put all indegree 0 nodes into a Collection
- \* 3. Print and remove a node from Collection
- \* 4. Decrement indegrees of the node's neighbors.
- \* 5. If any neighbor has indegree 0, place in Collection. Go to 3.



ATM	comp	groceries	recipe	stamps	taxes	cook	grandma	postcard	mail taxes
0	0	2	1	1	1	2	2	1	2

# Topological Sort

## Running time

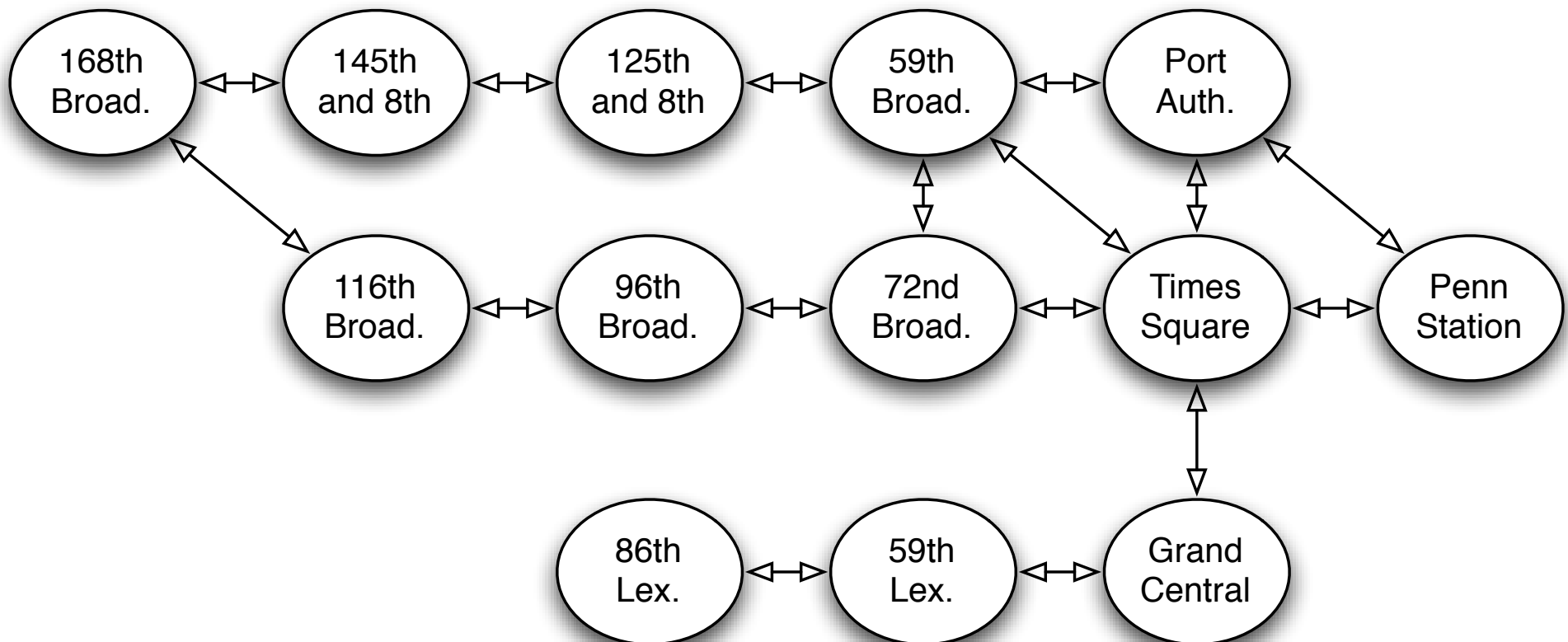
- \* Initial indegree computation:  $O(|E|)$ 
  - \* Unless we update indegree as we build graph
- \*  $|V|$  nodes must be enqueued/dequeued
- \* Dequeue requires operation for outgoing edges
- \* Each edge is used, but never repeated
- \* Total running time  $O(|V| + |E|)$

# Shortest Path

- \* Given  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ , and a node  $\mathbf{s} \in \mathbf{V}$ , find the shortest (weighted) path from  $\mathbf{s}$  to every other vertex in  $\mathbf{G}$ .
- \* Motivating example: subway travel
  - \* Nodes are junctions, transfer locations
  - \* Edge weights are estimated time of travel

# Approximate MTA Express Stop Subgraph

\* A few inaccuracies (don't use this to plan any trips)

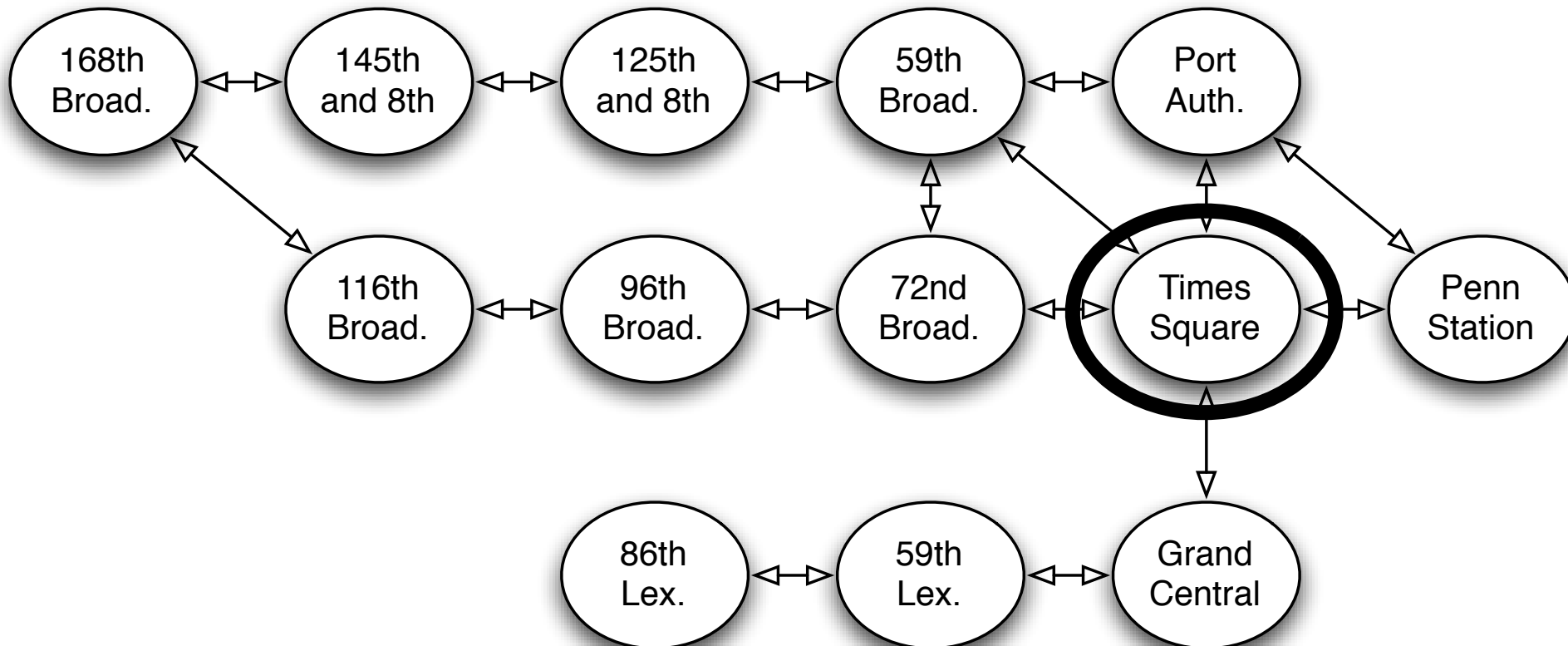


# Breadth First Search

- \* Like a level-order traversal
- \* Find all adjacent nodes (level 1)
- \* Find *new* nodes adjacent to level 1 nodes (level 2)
- \* ... and so on
- \* We can implement this with a queue

# Unweighted Shortest Path Algorithm

- \* Set node  $s$ ' distance to 0 and enqueue  $s$ .
- \* Then repeat the following:
  - \* Dequeue node  $v$ . For unset neighbor  $u$ :
    - \* set neighbor  $u$ 's distance to  $v$ 's distance +1
    - \* mark that we reached  $v$  from  $u$
    - \* enqueue  $u$



	168 <sup>th</sup> Broad.	145 <sup>th</sup> Broad.	125 <sup>th</sup> 8th	59 <sup>th</sup> Broad.	Port Auth.	116 <sup>th</sup> Broad.	96 <sup>th</sup> Broad.	72 <sup>nd</sup> Broad.	Times Sq.	Penn St.	86 <sup>th</sup> Lex.	59 <sup>th</sup> Lex.	Grand Centr.
dist									0				
prev									source				

# Weighted Shortest Path

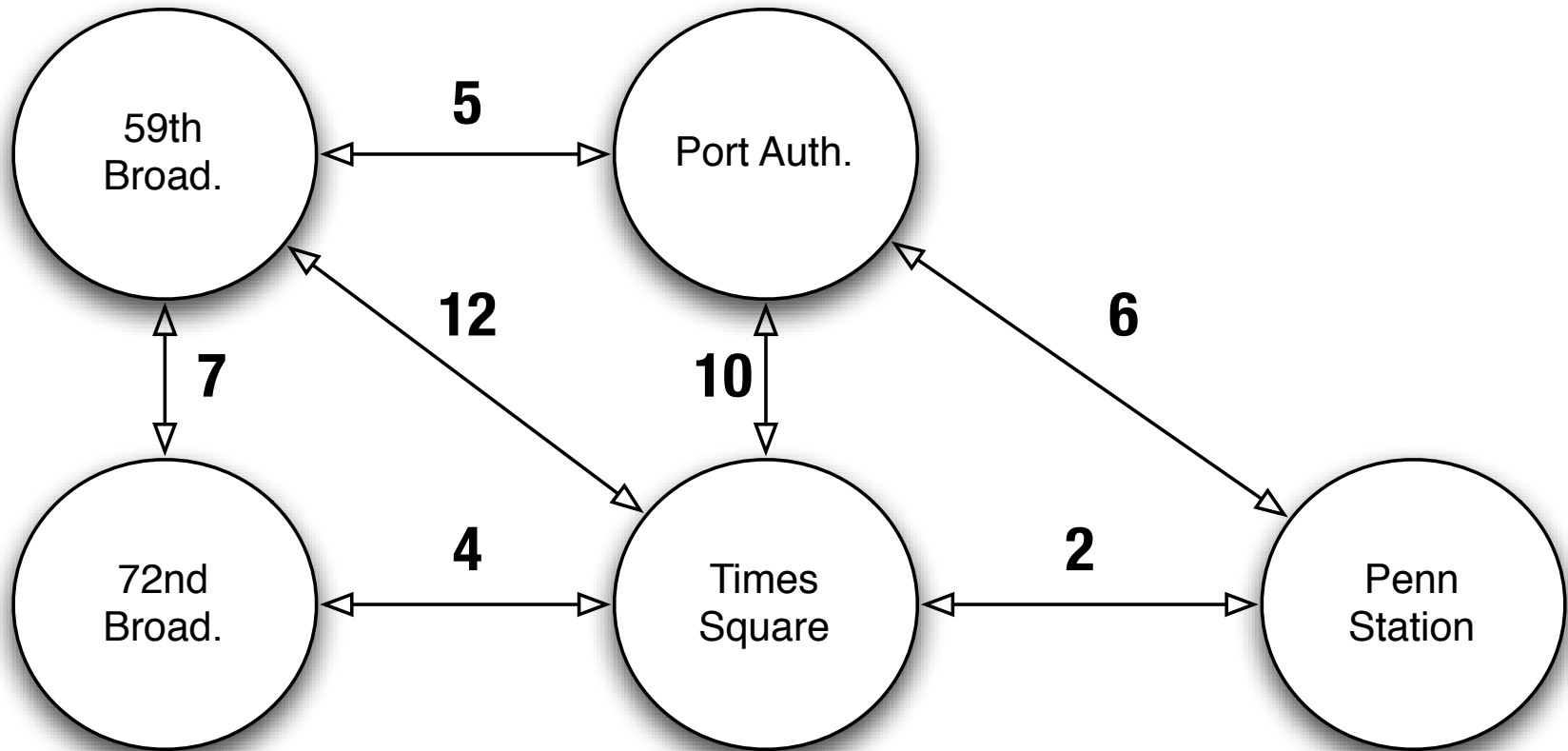
- \* The problem becomes more difficult when edges have different weights
- \* Weights represent different costs on using that edge
- \* Standard algorithm is **Dijkstra's Algorithm**

# Dijkstra's Algorithm

- \* Keep distance overestimates  $D(v)$  for each node  $v$  (all non-source nodes are initially infinite)
- \* 1. Choose node  $v$  with smallest *unknown* distance
- \* 2. Declare that  $v$ 's shortest distance is *known*
- \* 3. Update distance estimates for neighbors

# Updating Distances

- \* For each of  $\mathbf{v}$ 's neighbors,  $\mathbf{w}$ ,
- \* if  $\min(\mathbf{D}(\mathbf{v}) + \text{weight}(\mathbf{v}, \mathbf{w}), \mathbf{D}(\mathbf{w}))$
- \* i.e., update  $\mathbf{D}(\mathbf{w})$  if the path going through  $\mathbf{v}$  is cheaper than the best path so far to  $\mathbf{w}$



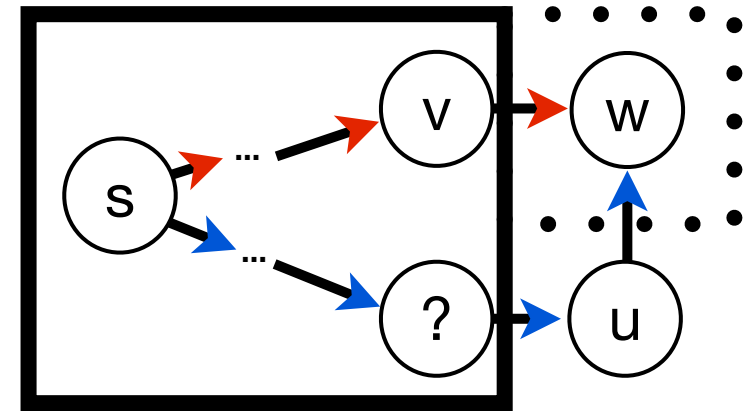
59 <sup>th</sup> Broad.	Port Auth.	72 <sup>nd</sup> Broad	Times Sq.	Penn St.
inf	inf	inf	inf	0
?	?	?	?	home

# Dijkstra's Algorithm Analysis

- \* First, convince ourselves that the algorithm works.
- \* At each stage, we have a set of nodes whose shortest paths we know
- \* In the base case, the set is the source node.
- \* Inductive step: if we have a correct set, is greedily adding the shortest neighbor correct?

# Proof by Contradiction (Sketch)

- \* Contradiction: Dijkstra's finds a shortest path to node **w** through **v**, but there exists an even shorter path
- \* This shorter path must pass from inside our known set to outside.
- \* Call the 1<sup>st</sup> node in cheaper path outside our set **u**
- \* The path to **u** must be shorter than the path to **w**
  - \* But then we would have chosen **u** instead



# Computational Cost

- \* Keep a priority queue of all unknown nodes
- \* Each stage requires a **deleteMin**, and then some **decreaseKeys** (the # of neighbors of node)
- \* We call **decreaseKey** once per edge, we call **deleteMin** once per vertex
- \* Both operations are  $O(\log |V|)$
- \* Total cost:  $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$

# Reading

- \* Weiss Section 9.3 (today's material)
- \* Weiss Section 9.4 (Monday's material)