



# CSE 332 Data Abstractions: Graphs and Graph Traversals

Kate Deibel  
Summer 2012

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

1

## Last Time

We introduced the idea of graphs and their associated terminology

Key terms included:

- Directed versus Undirected
- Weighted versus Unweighted
- Cyclic or Acyclic
- Connected or Disconnected
- Dense or Sparse
- Self-loops or not

These are all important concepts to consider when implementing a graph data structure

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

2

## Graph Data Structures

The two most common graph data structures

- Adjacency Matrix
- Adjacency List

Whichever is best depends on the type of graph, its properties, and what you want to do with the graph

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

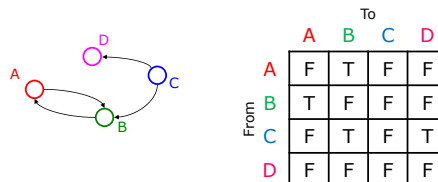
3

## Adjacency Matrix

Assign each node a number from 0 to  $|V|-1$

A  $|V| \times |V|$  matrix of Booleans (or 0 versus 1)

- Then  $M[u][v] = \text{true}$  → an edge exists from u to v
- This example is for a directed graph



July 25, 2012

CSE 332 Data Abstractions, Summer 2012

4

## Adjacency Matrix Properties

Run time to get a vertex v's out-edges?

$O(|V|)$  → iterate over v's row

Run time to get a vertex v's in-edges?

$O(|V|)$  → iterate over v's column

Run time to decide if an edge (u,v) exists?

$O(1)$  → direct lookup of  $M[u][v]$

Run time to insert an edge (u,v)?

$O(1)$  → set  $M[u][v] = \text{true}$

Run time to delete an edge (u,v)?

$O(1)$  → set  $M[u][v] = \text{false}$

Space requirements:

$O(|V|^2)$  → 2-dimensional array

Best for sparse or dense graphs?

**Dense** → We have to store every possible edge!!

		To			
		A	B	C	D
From	A	F	T	F	F
	B	T	F	F	F
	C	F	T	F	T
	D	F	F	F	F

July 25, 2012

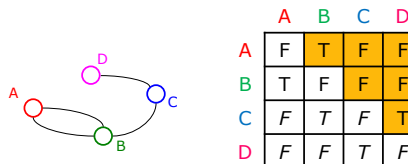
CSE 332 Data Abstractions, Summer 2012

5

## Adjacency Matrix: Undirected Graphs

How will the adjacency matrix work for an undirected graph?

- Will be symmetric about diagonal axis
- Save space by using only about half the array?



- But how would you "get all neighbors"?

July 25, 2012

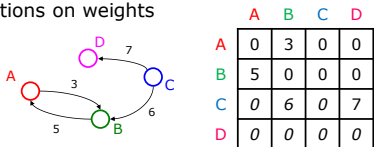
CSE 332 Data Abstractions, Summer 2012

6

### Adjacency Matrix: Weighted Graphs

How will the adjacency matrix work for a **weighted graph**?

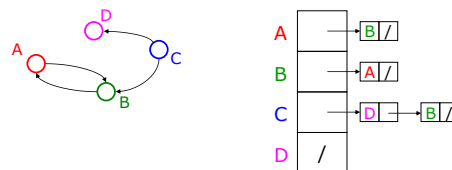
- Instead of Boolean, store a number in each cell
- Need some value to represent 'not an edge'
  - 0, -1, or some other value based on how you are using the graph
- Might need to be a separate field if no restrictions on weights



### Adjacency List

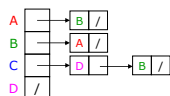
Assign each node a number from 0 to  $|V|-1$

- An array of length  $|V|$  in which each entry stores a list of all adjacent vertices (e.g., linked list)
- This example is again for a directed graph



### Adjacency List Properties

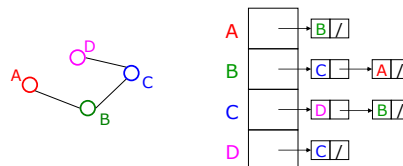
- Run time to get a vertex  $v$ 's out-edges?
  - $O(d)$  → where  $d$  is  $v$ 's out-degree
- Run time to get a vertex  $v$ 's in-edges?
  - $O(|E|)$  → check every vertex list (or keep a second list for in-edges)
- Run time to decide if an edge  $(u,v)$  exists?
  - $O(d)$  → where  $d$  is  $u$ 's out-degree
- Run time to insert an edge  $(u,v)$ ?
  - $O(1)$  → unless you need to check if it's already there
- Run time to delete an edge  $(u,v)$ ?
  - $O(d)$  → where  $d$  is  $u$ 's out-degree
- Space requirements:
  - $O(|V|+|E|)$  → vertex array plus edge nodes
- Best for sparse or dense graphs?
  - Sparse** → Only store the edges needed



### Adjacency List: Undirected Graphs

Adjacency lists also work well for **undirected graphs** with one caveat

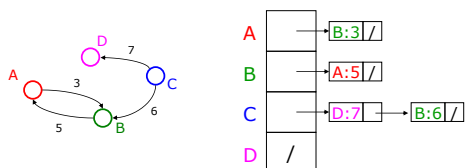
- Put each edge in two lists to support efficient "get all neighbors"
- Only an additional  $O(|E|)$  space



### Adjacency List: Weighted Graphs

Adjacency lists also work well for **weighted graphs** but where do you store the weights?

- In a matrix? →  $O(|V|^2)$  space
- Store a weight at each node in list →  $O(|E|)$  space



### Which is better?

Graphs are often sparse

- Streets form grids
- Airlines rarely fly to all cities

Adjacency lists generally the better choice

- Slower performance
- **HUGE** space savings

### How Huge of Space Savings?

Consider this 6x6 city street grid:

$|V| = 36$

$|E| = 6 \times 5 \times 2 + 6 \times 5 \times 2 = 120$

Adjacency Matrix:  $O(|V|^2)$

$\rightarrow 36^2 = 1296$

Adjacency List:  $O(|E| + |V|)$

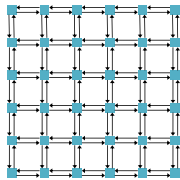
$\rightarrow 36 + 2 \times 120 = 276$  (we'll store both in and out-edges)

Savings Factor =  $276/1296 = 23/108 \approx 21\%$  of the space

In general, savings are:

$$\frac{V + E}{V^2} = \frac{1}{V} + \frac{E}{V^2}$$

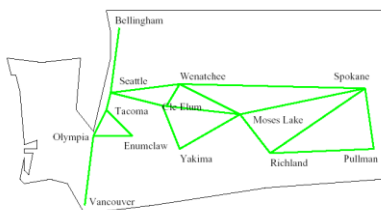
Recall that a sparse graph has  $|E| = o(|V|^2)$ , strictly less than quadratic



Might be easier to list what isn't a graph application...

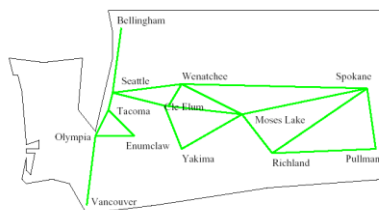
## GRAPH APPLICATIONS: TRAVERSALS

### Application: Moving Around WA State



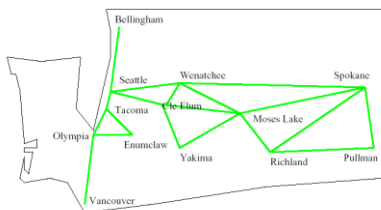
What's the *shortest* way to get from Seattle to Pullman?

### Application: Moving Around WA State



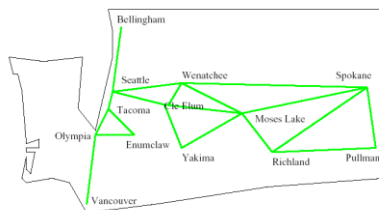
What's the *fastest* way to get from Seattle to Pullman?

### Application: Communication Reliability



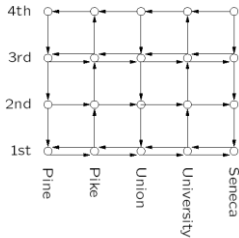
If Wenatchee's phone exchange *goes down*, can Seattle still talk to Pullman?

### Application: Communication Reliability



If Tacoma's phone exchange *goes down*, can Olympia still talk to Spokane?

## Applications: Bus Routes Downtown



If we're at 3rd and Pine, how can we get to 1st and University using Metro?  
How about 4th and Seneca?

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

19

## Graph Traversals

For an arbitrary graph and a starting node  $v$ , find all nodes reachable from  $v$  (i.e., there exists a path)

- Possibly "do something" for each node (print to output, set some field, return from iterator, etc.)

Related Problems:

- Is an undirected graph connected?
- Is a digraph weakly/strongly connected?
  - For strongly, need a cycle back to starting node

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

20

## Graph Traversals

Basic Algorithm for Traversals:

- Select a starting node
- Make a set of nodes adjacent to current node
- Visit each node in the set but "mark" each nodes after visiting them so you don't revisit them (and eventually stop)
- Repeat above but skip "marked nodes"

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

21

## In Rough Code Form

```

traverseGraph(Node start) {
  Set pending = emptySet();
  pending.add(start)
  mark start as visited
  while(pending is not empty) {
    next = pending.remove()
    for each node u adjacent to next
      if(u is not marked) {
        mark u
        pending.add(u)
      }
  }
}

```

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

22

## Running Time and Options

BFS and DFS traversal are both  $O(|V|+|E|)$  if using and adjacency list

- Queue/stack insert/removes are generally  $O(1)$
- Adjacency lists make it  $O(|V|)$  to find neighboring vertices/edges
- We will mark every node  $\rightarrow O(|V|)$
- We will touch every edge at most twice  $\rightarrow O(|E|)$

Because  $|E|$  is generally at least linear to  $|V|$ , we usually just say BFS/DFS are  $O(|E|)$

- Recall that in a connected graph  $|E| \geq |V| - 1$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

23

## The Order Matters

The order we traverse depends entirely on how add and remove work/are implemented

- DFS: a stack "depth-first graph search"
- BFS: a queue "breadth-first graph search"

DFS and BFS are "big ideas" in computer science

- Depth: recursively explore one part before going back to the other parts not yet explored
- Breadth: Explore areas closer to start node first

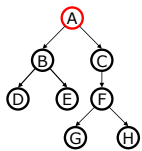
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

24

## Recursive DFS, Example with Tree

A tree is a graph and DFS and BFS are particularly easy to "see" in one



```
DFS(Node start) {
  mark and process start
  for each node u adjacent to start
  if u is not marked
    DFS(u)
}
```

Order processed: A, B, D, E, C, F, G, H

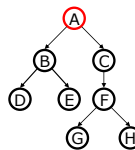
- This is a "pre-order traversal" for trees
- The marking is unneeded here but because we support arbitrary graphs, we need a means to process each node exactly once

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

25

## DFS with Stack, Example with Tree



```
DFS2(Node start) {
  initialize stack s to hold start
  mark start as visited
  while(s is not empty) {
    next = s.pop() // and "process"
    for each node u adjacent to next
      if(u is not marked)
        mark u and push onto s
  }
}
```

Order processed: A, C, F, H, G, B, E, D

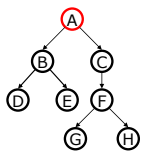
- A different order but still a perfectly fine traversal of the graph

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

26

## BFS with Queue, Example with Tree



```
BFS(Node start) {
  initialize queue q to hold start
  mark start as visited
  while(q is not empty) {
    next = q.dequeue() // and "process"
    for each node u adjacent to next
      if(u is not marked)
        mark u and enqueue onto q
  }
}
```

Order processed: A, B, C, D, E, F, G, H

- A "level-order" traversal

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

27

## DFS/BFS Comparison

BFS always finds the shortest path/optimal solution from the start vertex to the target

- Storage for BFS can be extremely large
- A  $k$ -nary tree of height  $h$  could result in a queue size of  $k^h$

DFS can use less space in finding a path

- If longest path in the graph is  $p$  and highest out-degree is  $d$  then DFS stack never has more than  $d \cdot p$  elements

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

28

## Implications

For large graphs, DFS is more memory efficient, *if we can limit the maximum path length to some fixed  $d$ .*

If we *knew* the distance from the start to the goal in advance, we could simply *not add any children to stack after level  $d$*

But what if we don't know  $d$  in advance?

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

29

## Iterative Deepening (IDFS)

Algorithms

- Try DFS up to recursion of  $K$  levels deep.
- If fail, increment  $K$  and start the entire search over

Performance:

- Like BFS, IDFS finds shortest paths
- Like DFS, IDFS uses less space
- Some work is repeated but minor compared to space savings

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

30

### Saving the Path

Our graph traversals can answer the standard *reachability* question:

"Is there a path from node x to node y?"

But what if we want to actually output the path?

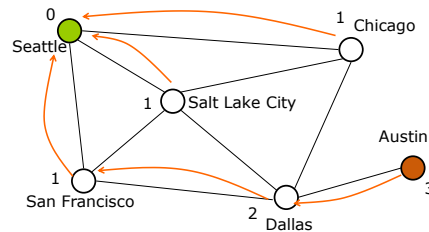
Easy:

- Store the previous node along the path: When processing *u* causes us to add *v* to the search, set *v.path* field to be *u*)
- When you reach the goal, follow path fields back to where you started (and then reverse the answer)
- What's an easy way to do the reversal? [A Stack!!](#)

### Example using BFS

What is a path from Seattle to Austin?

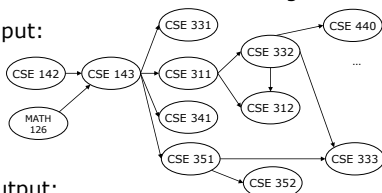
- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique



### Topological Sort

Problem: Given a DAG  $G=(V,E)$ , output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it

Example input:



Example output:

- 142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

Disclaimer: Do not use for official advising purposes! (Implies that CSE 332 is a pre-req for CSE 312 - not true!)

### Questions and Comments

Terminology:

A DAG represents a *partial order* and a topological sort produces a *total order* that is consistent with it

Why do we perform topological sorts only on DAGs?

- Because a cycle means there is no correct answer

Is there always a unique answer?

- No, there can be one or more answers depending on the provided graph

What DAGs have exactly 1 answer?

- Lists

### Uses Topological Sort

Figuring out how to finish your degree

Computing the order in which to recalculate cells in a spreadsheet

Determining the order to compile files with dependencies

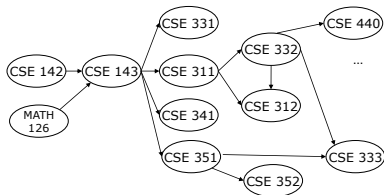
In general, use a dependency graph to find an allowed order of execution

### Topological Sort: First Approach

- Label each vertex with its in-degree
  - Think "write in a field in the vertex"
  - You could also do this with a data structure on the side
- While there are vertices not yet outputted:
  - Choose a vertex *v* labeled with in-degree of 0
  - Output *v* and "remove it" from the graph
  - For each vertex *u* adjacent to *v*, **decrement in-degree of u**
    - (i.e., *u* such that (*v*,*u*) is in *E*)

Example

Output:

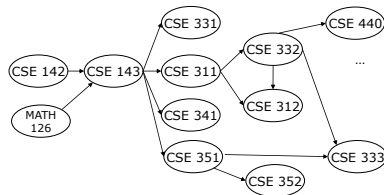


Node: 126 142 143 311 312 331 332 333 341 351 352 440  
 Removed?  
 In-deg:

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 37

Example

Output:

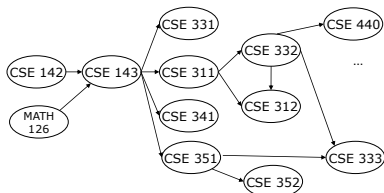


Node: 126 142 143 311 312 331 332 333 341 351 352 440  
 Removed?  
 In-deg: 0 0 2 1 2 1 1 2 1 1 1 1

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 38

Example

Output:  
126

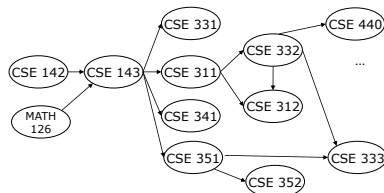


Node: 126 142 143 311 312 331 332 333 341 351 352 440  
 Removed? x  
 In-deg: 0 0 2 1 2 1 1 2 1 1 1 1  
 1

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 39

Example

Output:  
126  
142

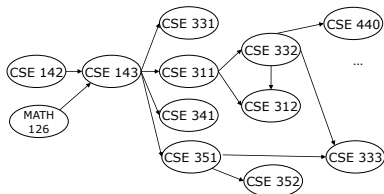


Node: 126 142 143 311 312 331 332 333 341 351 352 440  
 Removed? x x  
 In-deg: 0 0 2 1 2 1 1 2 1 1 1 1  
 1  
 0

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 40

Example

Output:  
126  
142  
143

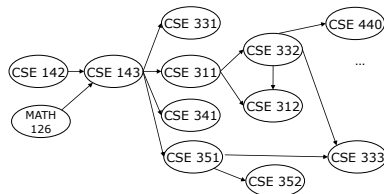


Node: 126 142 143 311 312 331 332 333 341 351 352 440  
 Removed? x x x  
 In-deg: 0 0 2 1 2 1 1 2 1 1 1 1  
 1 0 0 0 0  
 0

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 41

Example

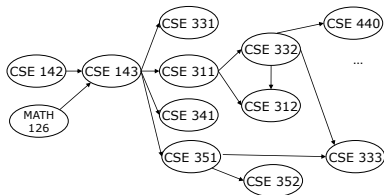
Output:  
126  
142  
143  
311



Node: 126 142 143 311 312 331 332 333 341 351 352 440  
 Removed? x x x x  
 In-deg: 0 0 2 1 2 1 1 2 1 1 1 1  
 1 0 1 0 0 0 0  
 0

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 42

Example



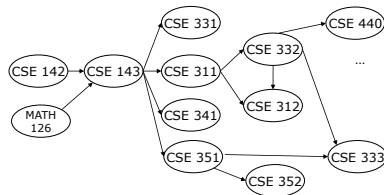
Output:

126  
142  
143  
311  
331

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x						
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0		0	0		
			0									

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 43

Example



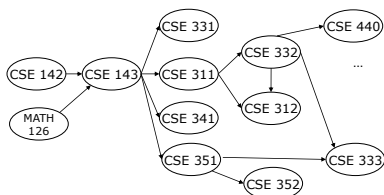
Output:

126  
142  
143  
311  
331  
332

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x	x					
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		
			0									

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 44

Example



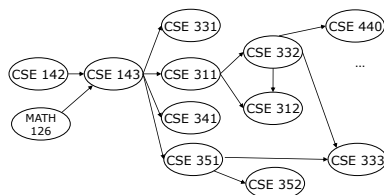
Output:

126  
142  
143  
311  
331  
332  
312

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x					
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		
			0									

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 45

Example



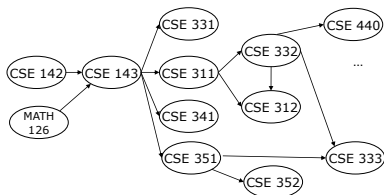
Output:

126  
142  
143  
311  
331  
332  
312  
341

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x			
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		
			0									

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 46

Example



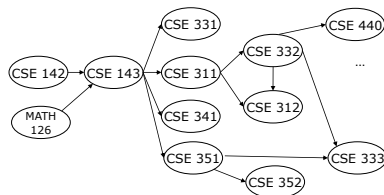
Output:

126  
142  
143  
311  
331  
332  
312  
341  
351

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x	x		
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	
			0									

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 47

Example



Output:

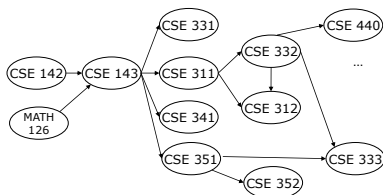
126  
142  
143  
311  
331  
332  
312  
341  
351  
333

Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x		
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	
			0									

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 48



## Example

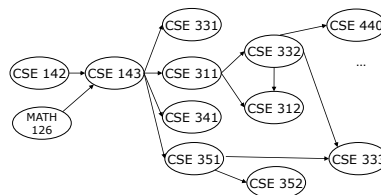


Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	x
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0	0	0	0	0	0	0	0	0	0

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 49

Output:  
 126 352  
 142  
 143  
 311  
 331  
 332  
 312  
 341  
 351  
 333

## Example



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	x
In-deg:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0	0	0	0	0	0	0	0	0	0

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 50

Output:  
 126 352  
 142 440  
 143  
 311  
 331  
 332  
 312  
 341  
 351  
 333

## Running Time?

```
labelEachVertexWithItsInDegree();
for(i=0; i < numVertices; i++) {
  v = findNewVertexOfDegreeZero();
  put v next in output
  for each w adjacent to v
    w.indegree--;
}
```

What is the worst-case running time?

- Initialization  $O(|V| + |E|)$  (assuming adjacency list)
- Sum of all find-new-vertex  $O(|V|^2)$  (because each  $O(|V|)$ )
- Sum of all decrements  $O(|E|)$  (assuming adjacency list)
- So total is  $O(|V|^2 + |E|)$  – not good for a sparse graph!

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

51

## Doing Better

Avoid searching for a zero-degree node every time!

- Keep the "pending" zero-degree nodes in a list, stack, queue, bag, or something that gives  $O(1)$  add/remove
- Order we process them affects the output but not correctness or efficiency

Using a queue:

- Label each vertex with its in-degree,
- Enqueue all 0-degree nodes
- While queue is not empty
  - $v = \text{dequeue}()$ 
    - Output  $v$  and remove it from the graph
  - For each vertex  $u$  adjacent to  $v$ , decrement the in-degree of  $u$  and if new degree is 0, enqueue it

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

52

## Running Time?

```
labelAllWithIndegreesAndEnqueueZeros();
for(i=0; i < numVertices; i++) {
  v = dequeue();
  put v next in output
  for each w adjacent to v {
    w.indegree--;
    if(w.indegree==0)
      enqueue(w);
  }
}
```

- Initialization:  $O(|V| + |E|)$  (assuming adjacency list)
- Sum of all enqueues and dequeues:  $O(|V|)$
- Sum of all decrements:  $O(|E|)$  (assuming adjacency list)
- So total is  $O(|E| + |V|)$  – much better for sparse graph!

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

53

## What about connectedness?

What happens if a graph is disconnected?

- With DFS?
- With BFS?
- With Topological Sorting?

All of these can be used to find connected components of the graph

- One just needs to start a new search at an unmarked node

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

54

Discovered by a most curmudgeonly man....

**MOST COMMON TRAVERSAL:  
FINDING SHORTEST PATHS**

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

55

*Finding the Shortest Path*

Question:

Given a graph G and two vertices v and u, what is the shortest path (shortest length) from v to u?

Solution:

Breadth-First Search starting at u will find minimum path length from v to u in time  $O(|E|+(|V|))$

Actually, the search can be easily extended to find minimum path length from v to every node

- Still  $O(|E|+(|V|))$
- No faster solution (in the worst-case) exists even if just focusing on one destination node

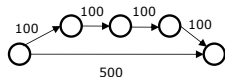
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

57

*Why BFS Will Not Work*

The shortest cost path may not have the fewest edges (shortest length)



This happens frequently with airline tickets

- Which is why I travel through Atlanta all too often to get to Kentucky from Seattle

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

59

*Finding the Shortest Path*

The graph traversals discussed so far work with path length (number of edges)but not path cost

Breadth-First Search found minimum path length from v to u in time  $O(|E|+(|V|))$

- Actually, can find the minimum path length from v to every node
  - Still  $O(|E|+(|V|))$
  - No faster way for a "distinguished" destination in the worst-case

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

56

*But That Was Path Length*

Path length is the number of edges in a path

Path cost is sum of the weight of edges in a path

New Question:

Given a weighted graph and node v, what is the minimum-cost path from v to every node?

We could phrase this as from a node v to u, but it is asymptotically no harder than for one destination

Solution:

Let's try BFS... it worked before, right?

July 25, 2012

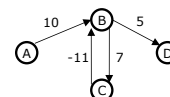
CSE 332 Data Abstractions, Summer 2012

58

*Regarding Negative Weights*

Negative edge weights are a can of worms

- If a cycle is negative, then the shortest path is  $-\infty$  (just repeat the cycle)



We will assume that there are no negative edge weights

- Today's algorithm gives erroneous results if edges can be negative

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

60

### Dijkstra's Algorithm—The Man



Named after its inventor Edsger Dijkstra (1930-2002)

Truly one of the "founders" of computer science

This is just one of his many contributions

*"Computer science is no more about computers than astronomy is about telescopes"*

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

61

### Dijkstra's Algorithm—The Idea

His algorithm is similar to BFS, but adapted to handle weights

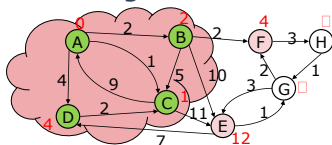
- A priority queue will prove useful for efficiency
- Grow set of nodes whose shortest distance has been computed
- Nodes not in the set will have a "best distance so far"

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

62

### Dijkstra's Algorithm—The Cloud



Initial State:

- Start node has cost 0
- All other nodes have cost  $\infty$

At each step:

- Pick closest unknown vertex  $v$
- Add it to the "cloud" of known vertices
- Update distances for nodes with edges from  $v$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

63

### The Algorithm

1. For each node  $v \neq \text{source}$ ,  
Set  $v.\text{cost} = \infty$  and  $v.\text{known} = \text{false}$
2. Set  $\text{source}.\text{cost} = 0$  and  $\text{source}.\text{known} = \text{true}$
3. While there are unknown nodes in the graph
  - a) Select the unknown node  $v$  with lowest cost
  - b) Mark  $v$  as known
  - c) For each edge  $(v, u)$  with weight  $w$ ,  

$$c_1 = v.\text{cost} + w \quad // \text{cost of best path through } v \text{ to } u$$

$$c_2 = u.\text{cost} \quad // \text{cost of best path to } u \text{ previously known}$$
 if  $(c_1 < c_2)$  // if the path through  $v$  is better  

$$u.\text{cost} = c_1$$

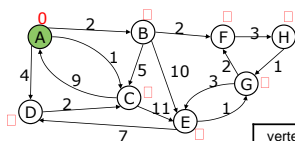
$$u.\text{path} = v \quad // \text{for computing actual paths}$$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

64

### Example #1



vertex	known?	cost	path
A			
B			
C			
D			
E			
F			
G			
H			

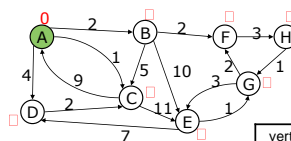
Order Added to Known Set:

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

65

### Example #1



vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	
H		??	

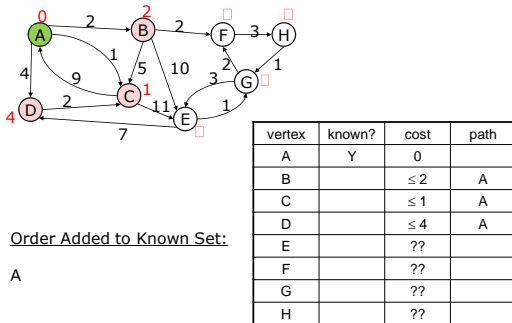
Order Added to Known Set:

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

66

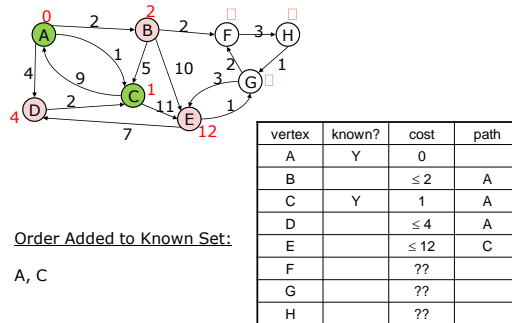
Example #1



Order Added to Known Set:

A

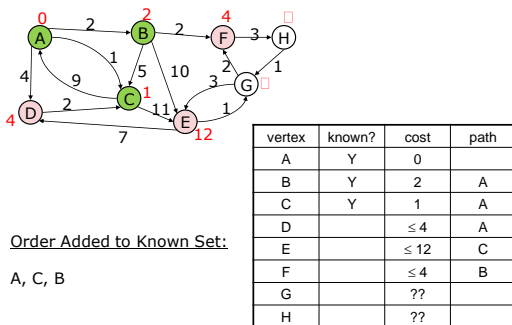
Example #1



Order Added to Known Set:

A, C

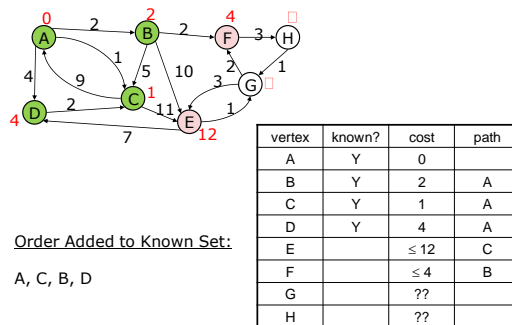
Example #1



Order Added to Known Set:

A, C, B

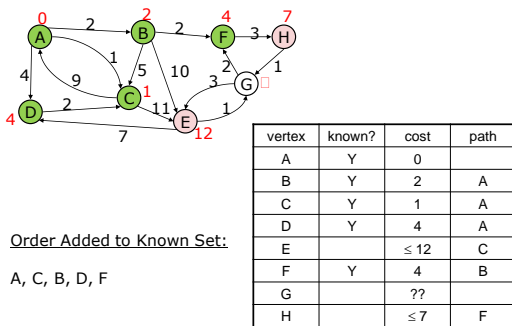
Example #1



Order Added to Known Set:

A, C, B, D

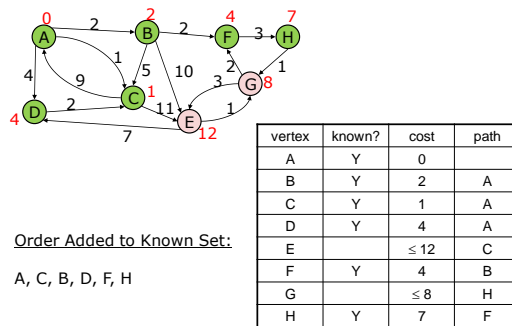
Example #1



Order Added to Known Set:

A, C, B, D, F

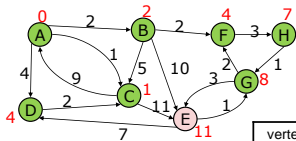
Example #1



Order Added to Known Set:

A, C, B, D, F, H

### Example #1



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

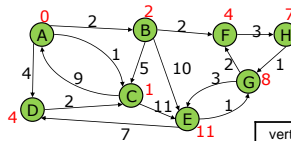
A, C, B, D, F, H, G

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

73

### Example #1



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G, E

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

74

### Important Features

When a vertex is marked known, the cost of the shortest path to that node is known

- The path is also known by following back-pointers

While a vertex is still not known, another shorter path to it **might** still be found

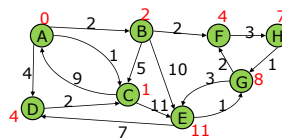
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

75

### Interpreting the Results

Now that we're done, how do we get the path from, say, A to E?



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G, E

July 25, 2012

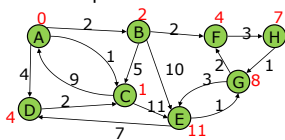
CSE 332 Data Abstractions, Summer 2012

76

### Stopping Short

How would this have worked differently if we were only interested in:

- the path from A to G?
- the path from A to E?



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

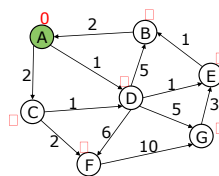
A, C, B, D, F, H, G, E

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

77

### Example #2



vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

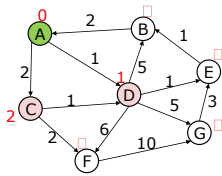
Order Added to Known Set:

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

78

Example #2



Order Added to Known Set:

A

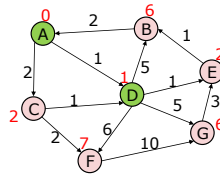
vertex	known?	cost	path
A	Y	0	
B		??	
C		≤ 2	A
D		≤ 1	A
E		??	
F		??	
G		??	

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

79

Example #2



Order Added to Known Set:

A, D

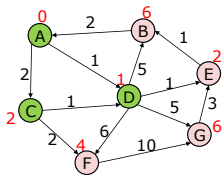
vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C		≤ 2	A
D	Y	1	A
E		≤ 2	D
F		≤ 7	D
G		≤ 6	D

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

80

Example #2



Order Added to Known Set:

A, D, C

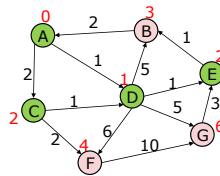
vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C	Y	2	A
D	Y	1	A
E		≤ 2	D
F		≤ 4	C
G		≤ 6	D

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

81

Example #2



Order Added to Known Set:

A, D, C, E

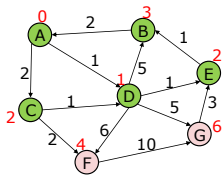
vertex	known?	cost	path
A	Y	0	
B		≤ 3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

82

Example #2



Order Added to Known Set:

A, D, C, E, B

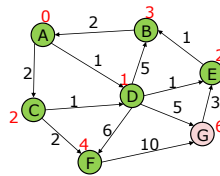
vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		≤ 4	C
G		≤ 6	D

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

83

Example #2



Order Added to Known Set:

A, D, C, E, B, F

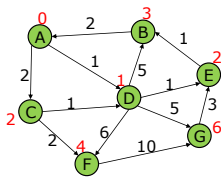
vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G		≤ 6	D

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

84

### Example #2



vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

Order Added to Known Set:

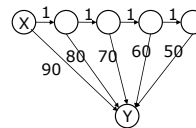
A, D, C, E, B, F, G

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

85

### Example #3



How will the best-cost-so-far for Y proceed?

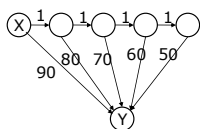
Is this expensive?

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

86

### Example #3



How will the best-cost-so-far for Y proceed?

90, 81, 72, 63, 54

Is this expensive?

No, each edge is processed only once

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

87

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

88

### Where are We?

Have described Dijkstra's algorithm

- For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges

What should we do next?

- Prove the algorithm is correct
- Analyze its efficiency

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

89

### Correctness: Rough Intuition

All "known" vertices have the correct shortest path

- True initially: shortest path to start node has cost 0
- If it stays true every time we mark a node as "known", then by induction this holds and eventually every vertex will be "known"

What we need to prove:

- When we mark a vertex as "known", we cannot ever discover a shorter path later in the algorithm
- If we could, then the algorithm fails

How we prove it:

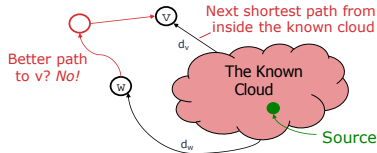
- This holds only because Dijkstra's algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction...

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

90

### Proof of Correctness (Rough Sketch)



- Suppose  $v$  is the next node to be marked known ("added to the cloud")  
 The best-known path to  $v$  must have only nodes "in the cloud"
- We have selected it, and we only know about paths through the cloud to a node at the edge of the cloud
- Assume the actual shortest path to  $v$  is different
- It is not entirely within the cloud, or else we would know about it
  - So it must use non-cloud nodes. Let  $w$  be the first non-cloud node on this path
  - The part of the path up to  $w$  is already known and must be shorter than the best-known path to  $v$ :  $d_w + \dots < d_v \rightarrow d_w < d_v$
  - Ergo,  $w$  should have been picked before  $v$ . Contradiction.

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 91

### Efficiency, First Approach

- Use pseudocode to determine asymptotic run-time
- Important: note that each edge is processed only once

```

dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost) {
          a.cost = b.cost + weight((b,a))
          a.path = b
        }
  }
}
    
```

Complexity annotations on the right:

- $O(|V|)$  for the initialization loop
- $O(|V|^2)$  for the while loop (finding the next node)
- $O(|E|)$  for the inner edge loop
- $O(|V|^2)$  for the total complexity

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 92

### Improving Asymptotic Running Time

So far we have an abysmal  $O(|V|^2)$

We had a similar "problem" with topological sort being  $O(|V|^2)$  due to each iteration looking for the node to process next

- We solved it with a queue of zero-degree nodes
- But here we need the lowest-cost node and costs can change as we process edges

Solution?

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 93

### Improving Asymptotic Running Time

We will use a priority queue

- Hold all unknown nodes
- Priority will be their current cost

But we need to update costs

- Priority queue must have a decreaseKey operation
- For efficiency, each node should maintain a reference from its position in the queue
  - Eliminates need for  $O(\log n)$  lookup
  - Conceptually simple, but can be a pain to code up

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 94

### Efficiency, Second Approach

Use pseudocode to determine asymptotic run-time

- Note that deleteMin() and decreaseKey() operations are independent of each other

```

dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost) {
          decreaseKey(a, "new cost - old cost")
          a.path = b
        }
  }
}
    
```

Complexity annotations on the right:

- $O(|V|)$  for the initialization loop
- $O(|V|)$  for building the heap
- $O(|V| \log |V|)$  for the while loop (deleteMin)
- $O(|E| \log |V|)$  for the inner edge loop (decreaseKey)
- $O(|V| \log |V| + |E| \log |V|)$  for the total complexity

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 95

### Dense versus Sparse Again

First approach:  $O(|V|^2)$

Second approach:  $O(|V| \log |V| + |E| \log |V|)$

So which is better?

- Sparse:  $O(|V| \log |V| + |E| \log |V|)$ 
  - If  $|E| = \Theta(|V|)$ , then  $O(|E| \log |V|)$
- Dense:  $O(|V|^2)$ 
  - If  $|E| = \Theta(|V|^2)$ , then  $|E| \log |V| > |V|^2$
- Neither sparse or dense?
  - Second approach still likely to be better

July 25, 2012 CSE 332 Data Abstractions, Summer 2012 96



*But...*

Remember these are worst-case and asymptotic

Priority queue might have worse constant factors

On the other hand, for "normal graphs"

- We might rarely call decreaseKey
- We might not percolate far
- This would make  $|E|\log|V|$  more like  $|E|$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

97

*What about connectedness?*

What happens if a graph is disconnected?

Unmarked/unvisited nodes will continue to have a cost of infinity

- Must be careful to do addition correctly:  
 $\infty + (\text{finite value}) = \infty$
- One speed-up would be to stop once a deleteMin() returns  $\infty$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

98



July 25, 2012

CSE 332 Data Abstractions, Summer 2012

99

*All-Pairs Shortest Path*

Dijkstra's algorithm requires a starting vertex

What if you want to find the shortest path between all pairs of vertices in the graph?

- Run Dijkstra's for each vertex  $v$ ?
- Can we do better? Yep

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

100

*Dynamic Programming*

An algorithmic technique that systematically records the answers to sub-problems in a table and re-uses those recorded results.

Simple Example:

Calculating the Nth Fibonacci number:

$$\text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2)$$

Recursion would be insanely expensive,

But it is cheap if you already know the results of prior computations

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

101

*Floyd-Warshall All-Pairs Shortest Path*

Dynamic programming algorithm for finding shortest paths between all vertices

Even works for negative edge weights

- Only meaningful in no negative cycles
- Can be used to detect such negative cycles
- Idea: Check to see if there is a path from  $v$  to  $v$  that has a negative cost

Overall performance:

- Time:  $O(|V|^3)$
- Space:  $O(|V|^2)$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

102

### The Algorithm

$M[u][v]$  stores the cost of the best path from  $u$  to  $v$   
 Initialized to cost of edge between  $M[u][v]$

The algorithm:

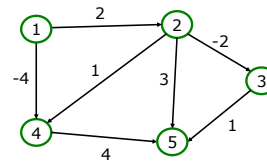
```
for (int k = 1; k <= V; k++)
  for (int i = 1; i <= V; i++)
    for (int j = 1; j <= V; j++)
      if ( M[i][k] + M[k][j] < M[i][j] )
        M[i][j] = M[i][k] + M[k][j]
```

**Invariant:**

After the  $k^{\text{th}}$  iteration, the matrix  $M$  includes the shortest path between all pairs that use on only vertices  $1..k$  as intermediate vertices in the paths

### Floyd-Warshall

#### All-Pairs Shortest Path



Initial state of the matrix:

	1	2	3	4	5
1	0	2	$\infty$	-4	$\infty$
2	$\infty$	0	-2	1	3
3	$\infty$	$\infty$	0	$\infty$	1
4	$\infty$	$\infty$	$\infty$	0	4
5	$\infty$	$\infty$	$\infty$	$\infty$	0

Note that non $\infty$ edges are indicated in some manner, such as infinity

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

103

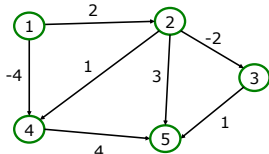
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

104

### Floyd-Warshall

#### All-Pairs Shortest Path



$k = 1$

	1	2	3	4	5
1	0	2	$\infty$	-4	$\infty$
2	$\infty$	0	-2	1	3
3	$\infty$	$\infty$	0	$\infty$	1
4	$\infty$	$\infty$	$\infty$	0	4
5	$\infty$	$\infty$	$\infty$	$\infty$	0

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

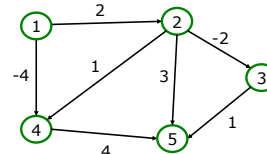
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

105

### Floyd-Warshall

#### All-Pairs Shortest Path



$k = 2$

	1	2	3	4	5
1	0	2	0	-4	5
2	$\infty$	0	-2	1	3
3	$\infty$	$\infty$	0	$\infty$	1
4	$\infty$	$\infty$	$\infty$	0	4
5	$\infty$	$\infty$	$\infty$	$\infty$	0

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

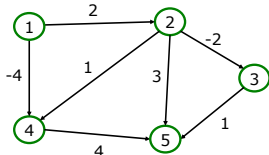
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

106

### Floyd-Warshall

#### All-Pairs Shortest Path



$k = 3$

	1	2	3	4	5
1	0	2	0	-4	1
2	$\infty$	0	-2	1	-1
3	$\infty$	$\infty$	0	$\infty$	1
4	$\infty$	$\infty$	$\infty$	0	4
5	$\infty$	$\infty$	$\infty$	$\infty$	0

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

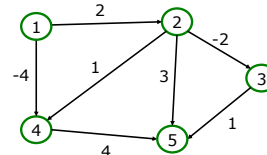
July 25, 2012

CSE 332 Data Abstractions, Summer 2012

107

### Floyd-Warshall

#### All-Pairs Shortest Path



$k = 4$

	1	2	3	4	5
1	0	2	0	-4	0
2	$\infty$	0	-2	1	-1
3	$\infty$	$\infty$	0	$\infty$	1
4	$\infty$	$\infty$	$\infty$	0	4
5	$\infty$	$\infty$	$\infty$	$\infty$	0

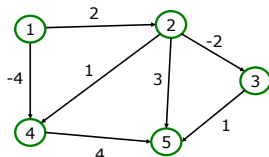
$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

108

Floyd-Warshall  
All-Pairs Shortest Path



k = 5

	1	2	3	4	5
1	0	2	0	-4	0
2	$\infty$	0	-2	1	-1
3	$\infty$	$\infty$	0	$\infty$	1
4	$\infty$	$\infty$	$\infty$	0	4
5	$\infty$	$\infty$	$\infty$	$\infty$	0

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

*What about connectedness?*

What happens if a graph is disconnected?

Floyd-Warshall will still calculate all-pair shortest paths.

Some will remain  $\infty$  to indicate that no path exists between those vertices

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

109

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

110

*What Comes Next?*

In the logical course progression, we would study the next graph topic:

### **Minimum Spanning Trees**

They are trees... that span... minimally!! Woo!!

But alas, we need to align lectures with projects and homework, so we will instead

- Start parallelism and concurrency
- Come back to graphs at the end of the course

July 25, 2012

CSE 332 Data Abstractions, Summer 2012

111