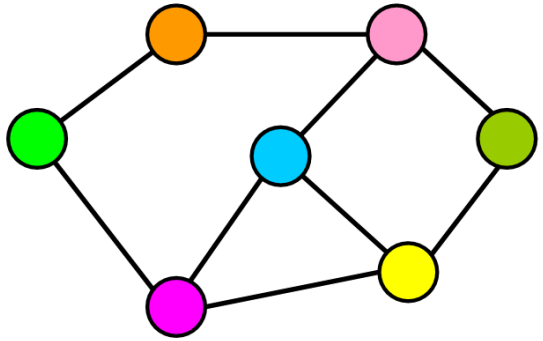
LVX
VERITAS
VIRTVS

Lesson 2:

Graphs

INFO6205: Program Structure and Algorithms



Northeastern University
College of Engineering

A graph is a formalism for representing relationships among items

Very general definition

Very general concept

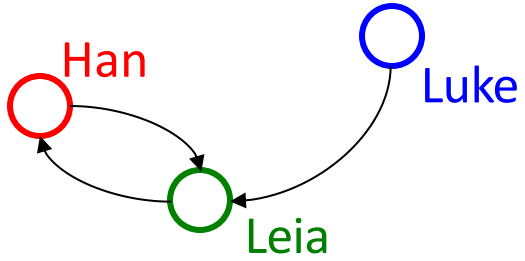
A graph is a pair: G = (V, E)

A set of vertices, also known as nodes: $V = \{v_1, v_2, \ldots, v_n\}$

A set of edges $E = \{e_1, e_2, \ldots, e_m\}$

Each edge e_i is a pair of vertices (v_j, v_k)

An edge "connects" the vertices



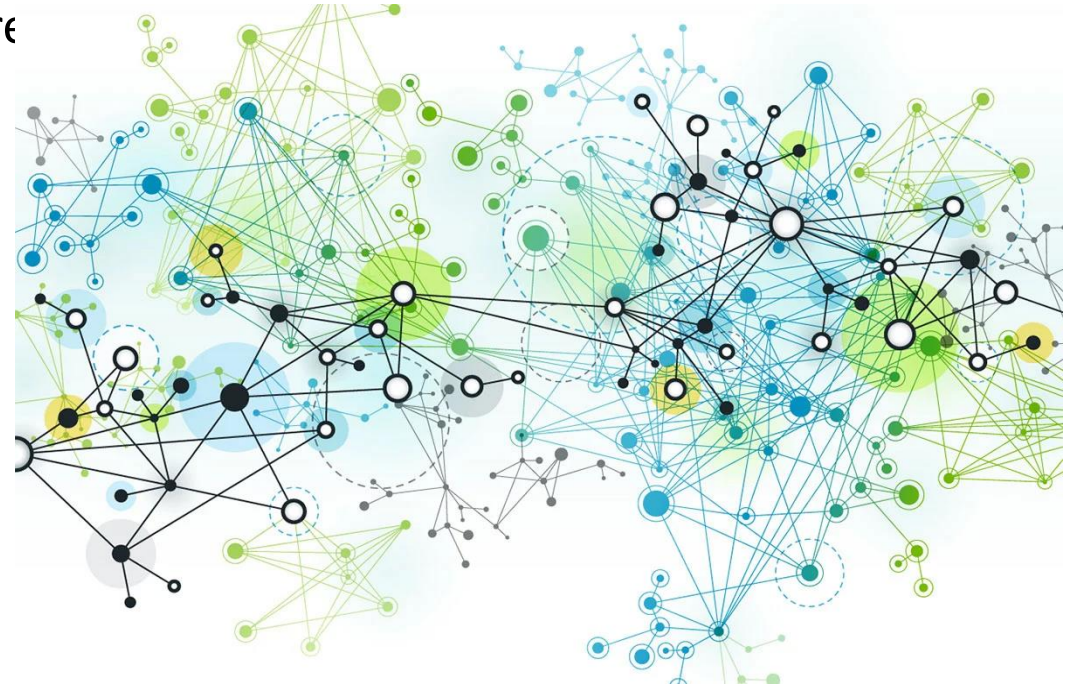
$V = \{Han, Leia, Luke\}$

$E = \{(Luke, Leia), (Han, Leia), (Leia, Han)\}$

Graphs can be *directed* or *undirected*

Some Graphs

For each example, what are the vertices and what are

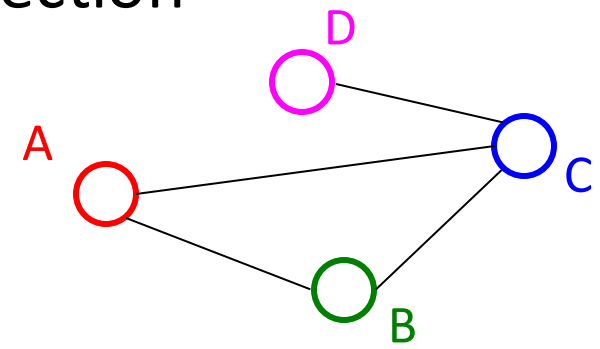


Web pages with links
Facebook friends
Methods in a program that call each other
Road maps
Airline routes
Family trees
Course pre-requisites

Undirected Graphs

In undirected graphs, edges have no specific direction

Edges are always "two-way"



Thus, $(u, v) \in E$ implies $(v, u) \in E$.

Only one of these edges needs to be in the set

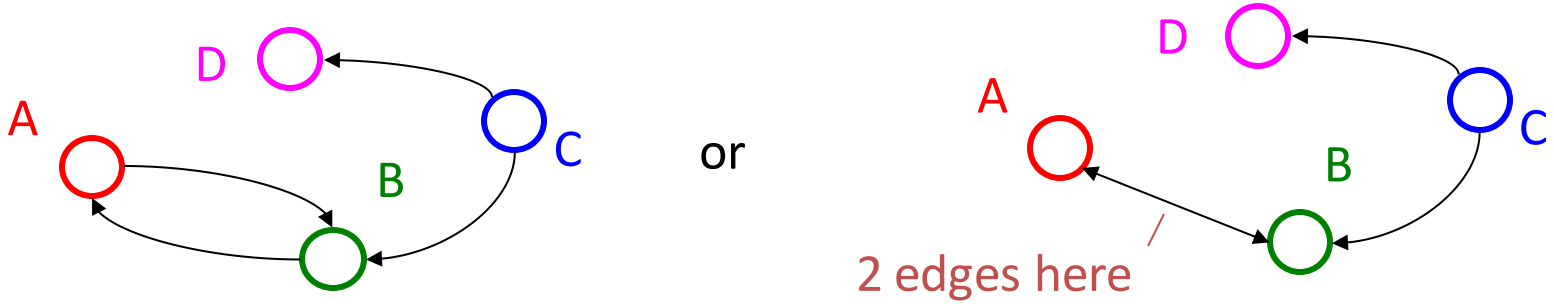
The other is implicit, so normalize how you check for it

Degree of a vertex: number of edges containing that vertex

Put another way: the number of adjacent vertices

Directed Graphs

In directed graphs (or digraphs), edges have direction



Thus, $(u, v) \in E$ does not imply $(v, u) \in E$.

Let $(u, v) \in E$ mean $u \rightarrow v$
Call u the source and v the destination
In-Degree of a vertex: number of in-bound edges (edges where the vertex is the destination)
Out-Degree of a vertex: number of out-bound edges (edges where the vertex is the source)

Self-Edges, Connectedness

A self-edge a.k.a. a loop edge is of the form (u, u)

The use/algorithm usually dictates if a graph has:

- No self edges
- Some self edges
- All self edges

A node can have a(n) degree / in-degree / out-degree of zero

A graph does not have to be connected

Even if every node has non-zero degree

More discussion of this to come

More Notation

For a graph $G = (V, E)$:

$|V|$ is the number of vertices

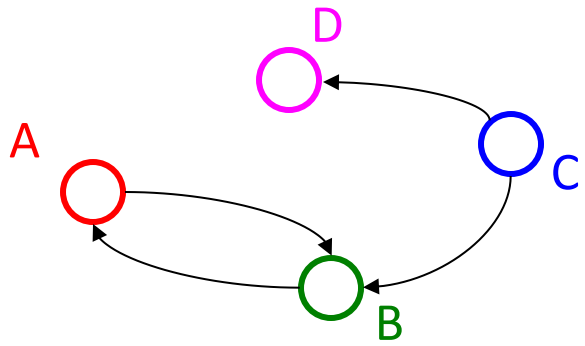
$|E|$ is the number of edges

- Minimum?
- Maximum for undirected?
- Maximum for directed?

If $(u, v) \in E$, then v is a neighbor of u (i.e., v is adjacent to u)

Order matters for directed edges:

u is not adjacent to $\mathbf{v}$ **unless** $\mathbf{(v, u) \in E}$



$V = \{A, B, C, D\}$

$E = \{(C, B), (A, B), (B, A), (C, D)\}$

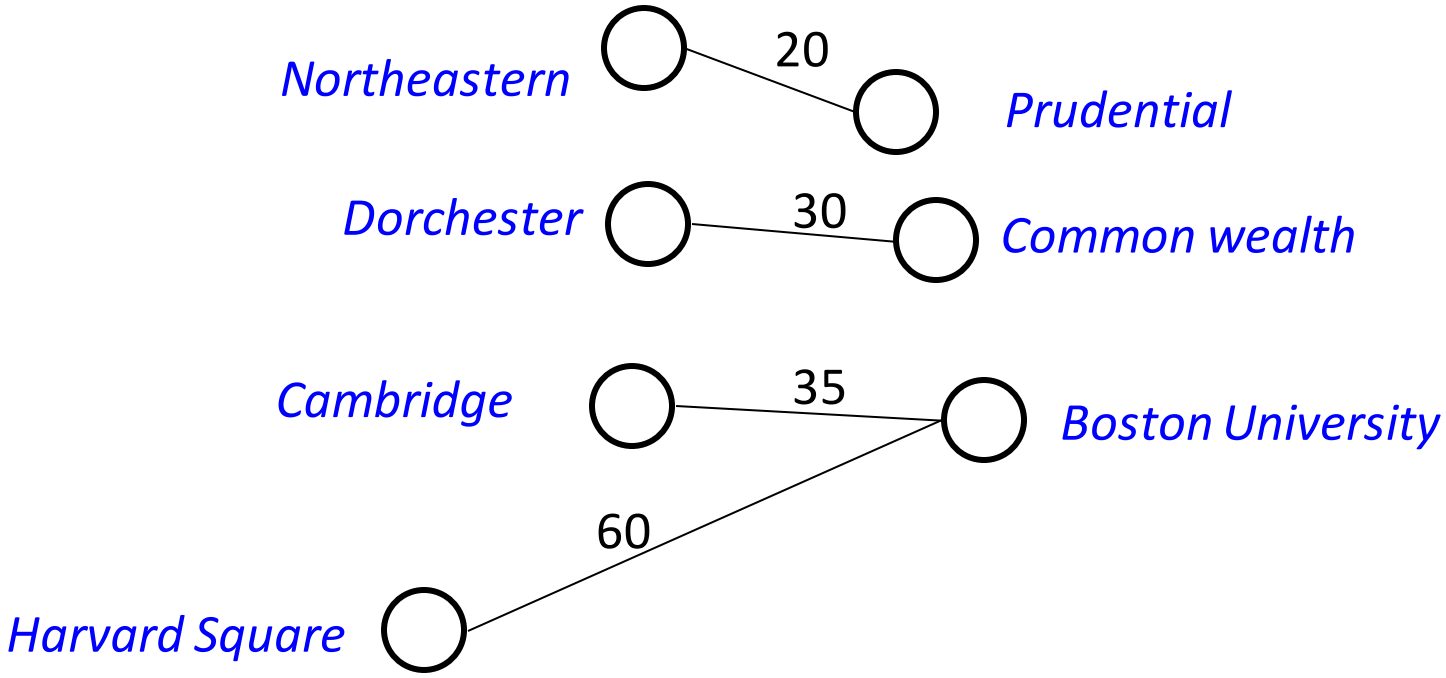
Examples Again

Which would use directed edges?
Which would have self-edges?
Which could have 0-degree nodes?

Web pages with links
Facebook friends
"Input data" for the Kevin Bacon game
Methods in a program that call each other
Road maps
Airline routes
Family trees
Course pre-requisites

Weighted Graphs

In a weighted graph, each edge has a weight or cost
Typically numeric (ints, decimals, doubles, etc.)
Orthogonal to whether graph is directed
Some graphs allow negative weights; many do not



Examples Again

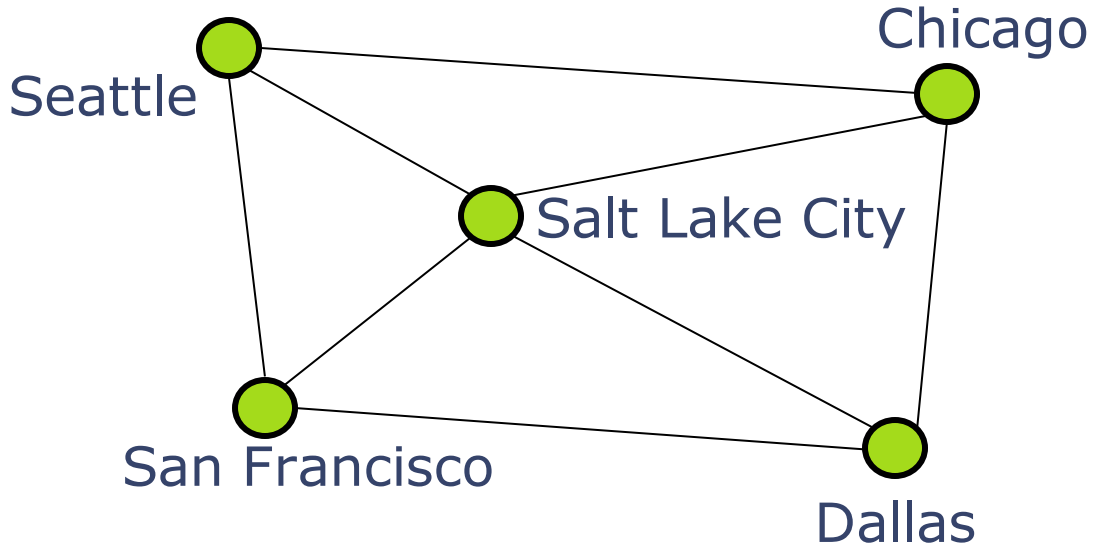
What, if anything, might weights represent for each of these?
Do negative weights make sense?

Web pages with links
Facebook friends
"Input data" for the Kevin Bacon game
Methods in a program that call each other
Road maps
Airline routes
Family trees
Course pre-requisites

Paths and Cycles

We say "a path exists from v_0 to v_n" if there is a list of vertices $[v_0, v_1, \ldots, v_n]$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$.

A cycle is a path that begins and ends at the same node ($v_0 == v_n$)



Example path (that also happens to be a cycle):

[Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]

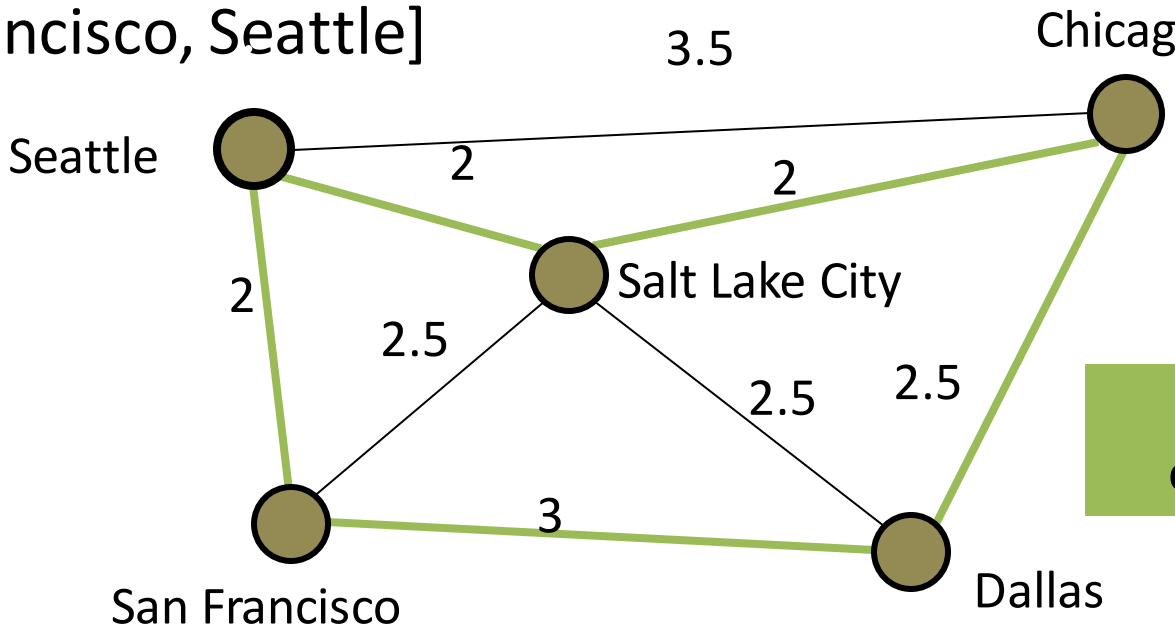
Path Length and Cost

Path length: Number of edges in a path

Path cost: Sum of the weights of each edge

Example where

P= [Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]



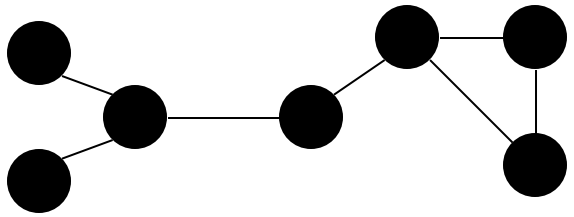
length(**P**) = 5

cost(**P**) = 11.5

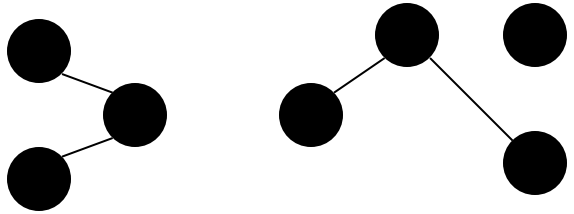
Length is sometimes called "unweighted cost"

Undirected Graph Connectivity

An undirected graph is connected if for all pairs of vertices u≠v, there exists a *path* from u to v

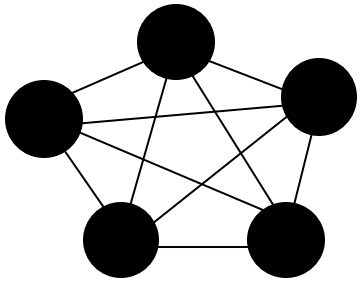


Connected graph



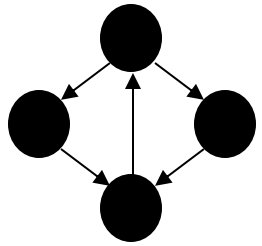
Disconnected graph

An undirected graph is complete, or fully connected, if for all pairs of vertices **u≠v there exists an *edge* from u to v**

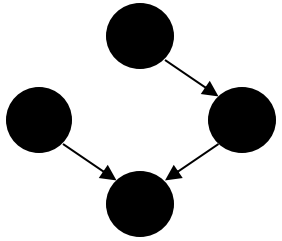


Directed Graph Connectivity

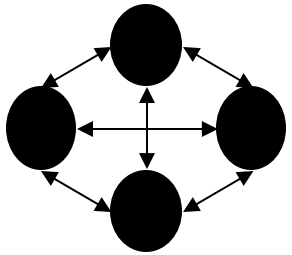
A directed graph is strongly connected if there is a path from every vertex to every other vertex



A directed graph is weakly connected if there is a path from every vertex to every other vertex *ignoring direction of edges*



A direct graph is complete or fully connected, if for all pairs of vertices **$u \neq v$, there exists an *edge* from u to v**



Examples Again

For undirected graphs: connected?

For directed graphs: strongly connected?

weakly connected?

Web pages with links

Facebook friends

"Input data" for the Kevin Bacon game

Methods in a program that call each other

Road maps

Airline routes

Family trees

Course pre-requisites

Graph Data Structures

Graphs are like good friends - they always know how to connect!

What's the Data Structure?

Graphs are often useful for lots of data and questions
Example: "What's the lowest-cost path from x to y"

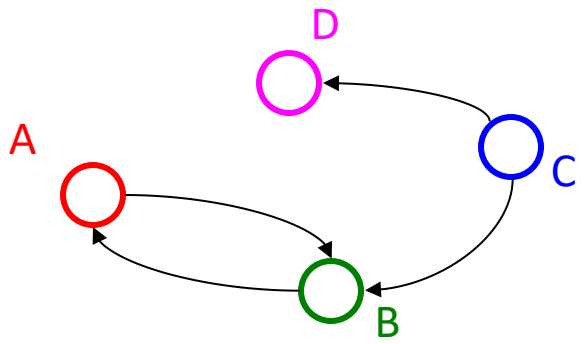
But we need a data structure that represents graphs

Which data structure is "best" can depend on:
properties of the graph (e.g., dense versus sparse)
the common queries about the graph ("is (u ,v) an edge?" vs "what are the neighbors of node u?")

We will discuss two standard graph representations
Adjacency Matrix and Adjacency List
Different trade-offs, particularly time versus space

Adjacency Matrix

Assign each node a number from 0 to |V|-1
A |V| x |V| matrix of Booleans (or 0 vs. 1)
Then M[u][v] == true means there is an edge from u to v



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

Running time to:

Get a vertex's out-edges:

Get a vertex's in-edges:

Decide if some edge exists:

Insert an edge:

Delete an edge:

Space requirements:

Best for sparse or dense graphs?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Adjacency Matrix Properties

Running time to:

- Get a vertex's out-edges: $O(|V|)$
- Get a vertex's in-edges: $O(|V|)$
- Decide if some edge exists: $O(1)$
- Insert an edge: $O(1)$
- Delete an edge: $O(1)$

Space requirements:

$O(|V|^2)$

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

Best for sparse or dense graphs? *dense*

Adjacency Matrix Properties

How will the adjacency matrix vary for an undirected graph?

Will be symmetric about diagonal axis

Matrix: Could we save space by using only about half the array?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	T	F

But how would you "get all neighbors"?

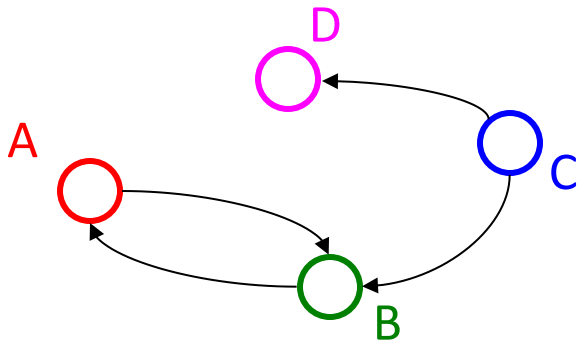
Adjacency Matrix Properties

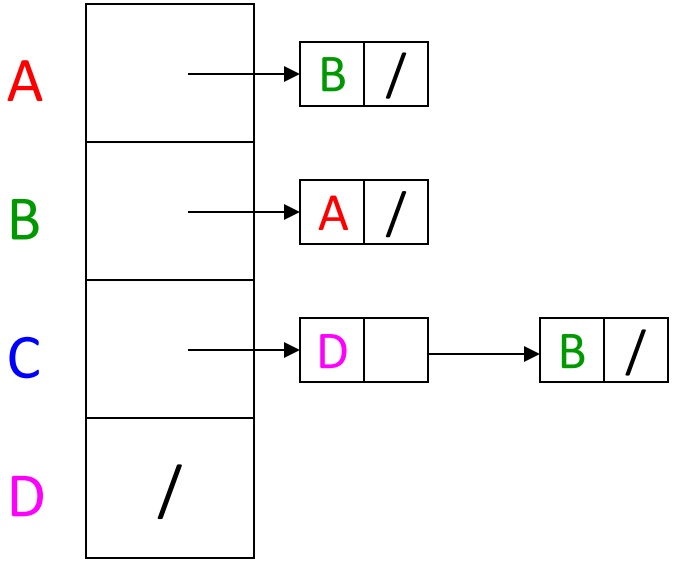
How can we adapt the representation for weighted graphs?
Instead of Boolean, store a number in each cell
Need some value to represent ‘not an edge’

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)





Adjacency List Properties

Running time to:

Get a vertex's out-edges:

Get a vertex's in-edges:

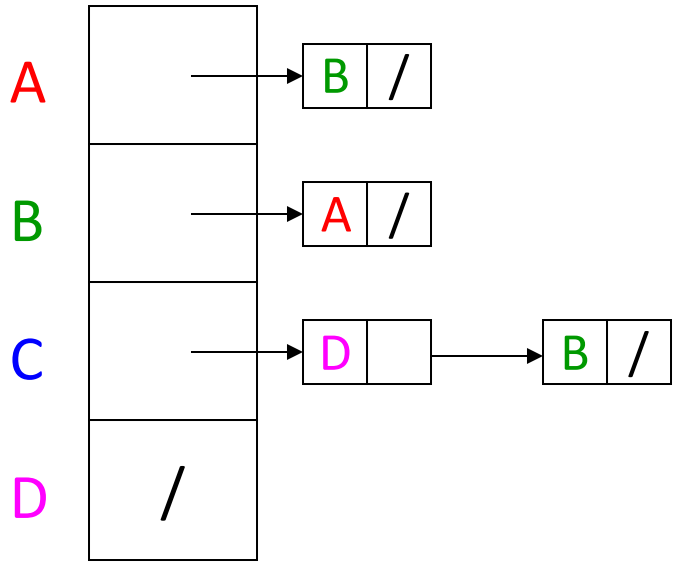
Decide if some edge exists:

Insert an edge:

Delete an edge:

Space requirements:

Best for sparse or dense graphs?



Adjacency List Properties

Running time to:

Get a vertex's out-edges:

$O(d)$ where d is out-degree of vertex

Get a vertex's in-edges:

$O(|E|)$ (could keep a second adjacency list for this!)

Decide if some edge exists:

$O(d)$ where d is out-degree of source

Insert an edge:

$O(1)$ (unless you need to check if it's already there)

Delete an edge:

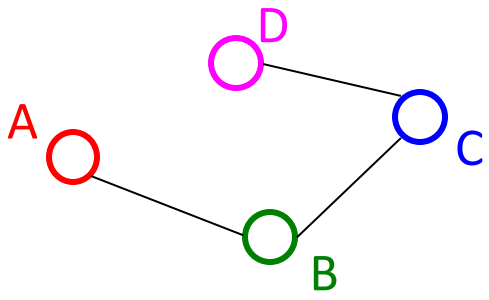
$O(d)$ where d is out-degree of source

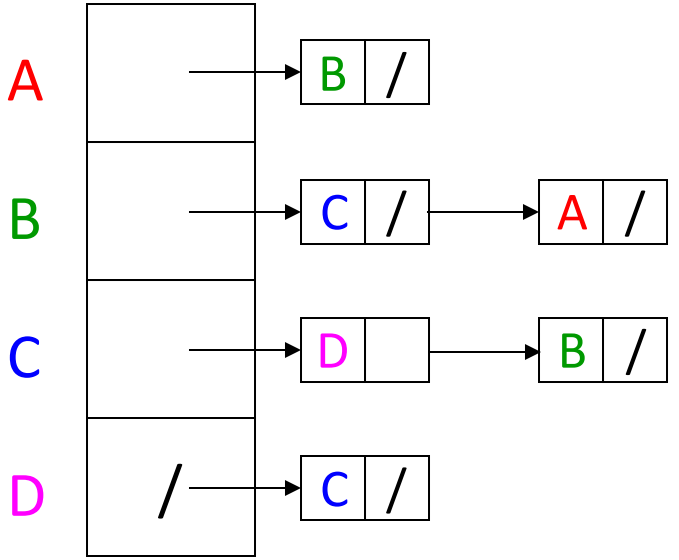
Space requirements: $O(|V|+|E|)$

Best for sparse or dense graphs? *sparse*

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"





Which is better?

Graphs are often sparse
Streets form grids
Airlines rarely fly to all cities

Adjacency lists should generally be your default choice
Slower performance compensated by greater space savings

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

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"

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)
            }
        }
    }
}
```

Running Time and Options

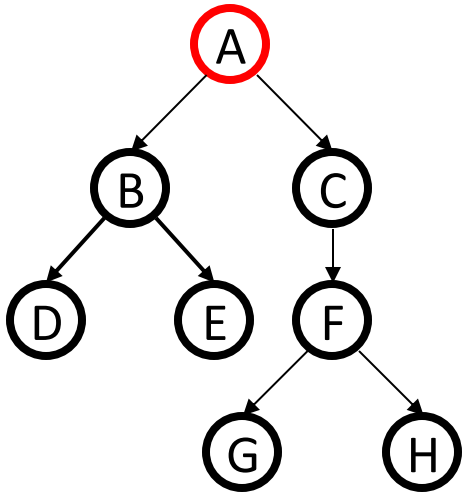
Assuming add and remove are $O(1)$, entire traversal is $O(|E|)$ if using an adjacency list

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

Recursive DFS, Example with Tree

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



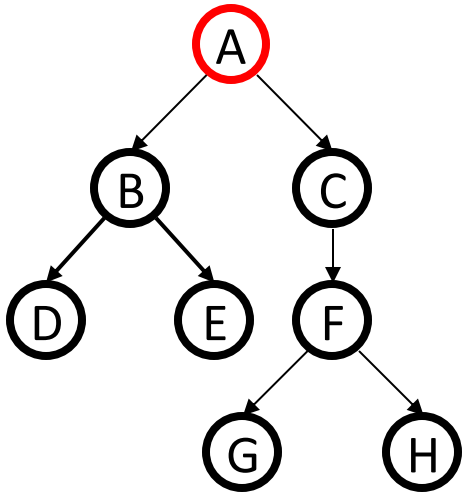

```
DFS(Node start) {
    mark and process start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```

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

DFS with Stack, Example with Tree

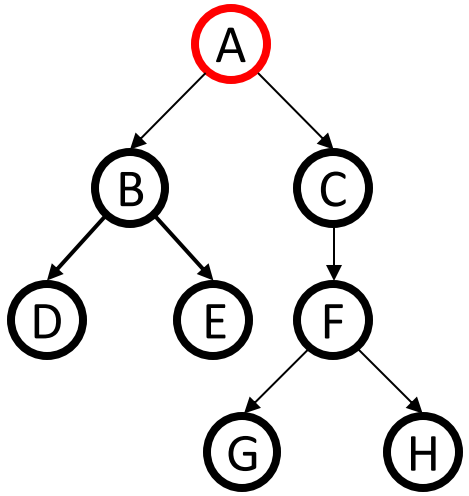



```
DFS2(Node start) {
  initialize stack s to hold start
  mark start as visited
  while(s is not empty) {
     next = s.pop() // and "process"
     for each node u adjacent to next
        if(u is not marked)
           mark u and push onto s
  }
}
```

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

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

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

DFS/BFS Comparison

BFS always finds the shortest path (or "optimal solution") from the starting node to a target node

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**

Implications

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

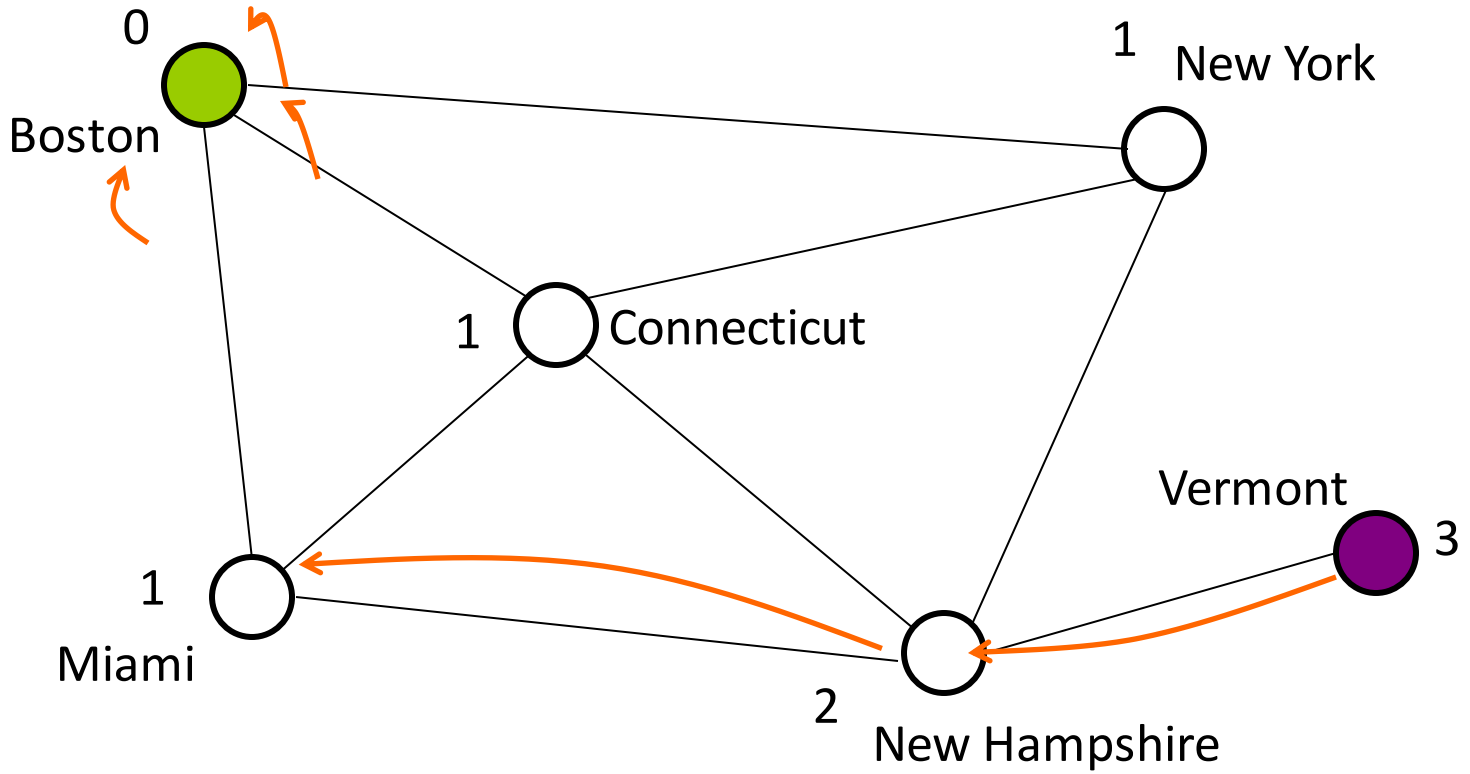
If we *knew* the distance from the start to the goal in advance, we could simply *not add any children to stack after level d*

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

Example using BFS

What is a path from Boston to Vermont?

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



Time For a Quiz!

Question 1: Which of the following is an application of graphs in real life?

a) Writing a text document
b) Storing integers in an array
c) Modeling social networks
d) Sorting a list of elements

Question 2: Which of the following data structures is commonly used to implement a graph?

a) Array
b) Linked list
c) Stack
d) Queue

Question 3: Which traversal algorithm is typically used to traverse a graph in a systematic way?

a) Breadth-first traversal
b) Depth-first traversal
c) Preorder traversal
d) Inorder traversal

Question 4: In a weighted graph, what do edge weights represent?

a) The distance between two nodes
b) The color of the edge
c) The direction of the edge
d) The name of the edge

Question 5: What is a graph in the context of data structures?

a) A visual representation of data
b) A linear data structure
c) A non-linear data structure
d) A mathematical equation

Question 6: If a graph is connected and has 'n' nodes, what is the minimum number of edges it must have to be a tree?

a) n
b) n - 1
c) n + 1
d) 2n