ADVANCED ANALYSIS OF ALGORITHMS
CPS 5440

ELEMENTARY TREE AND GRAPH ALGORITHMS REVIEW.

GRAPHS

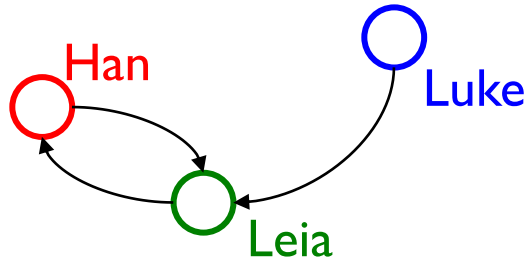
A graph is a formalism for representing relationships among items

- 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

Graphs can be *directed* or *undirected*



V = {Han,Leia,Luke}
E = {(Luke,Leia),
(Han,Leia),
(Leia,Han)}

A GRAPH ADT?

We can think of graphs as an ADT

- Operations would include $\boldsymbol{isEdge}(v_j, v_k)$
- But it is unclear what the "standard operations" would be for such an ADT

Instead, we tend to develop algorithms over graphs and then use data structures that are efficient for those algorithms

Many important problems can be solved by:

1. Formulating them in terms of graphs
2. Applying a standard graph algorithm

SOME GRAPHS

For each example, what are the vertices and what are the edges?

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

Core algorithms that work across such domains is why we are CSE

SCRATCHING THE SURFACE

- Graphs are a powerful representation and have been studied deeply
- Graph theory is a major branch of research in combinatorics and discrete mathematics
- Every branch of computer science involves graph theory to some extent

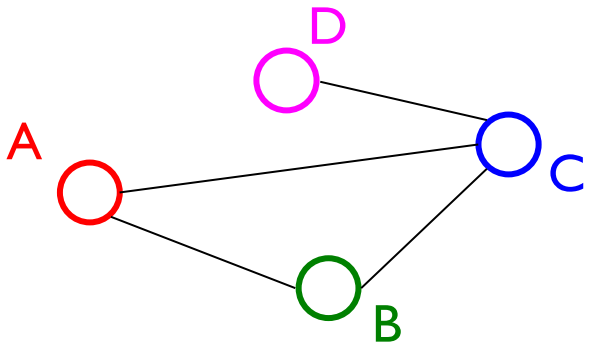
- To make formulating graphs easy and standard, we have a lot of *standard terminology* for graphs

GRAPH TERMINOLOGY

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

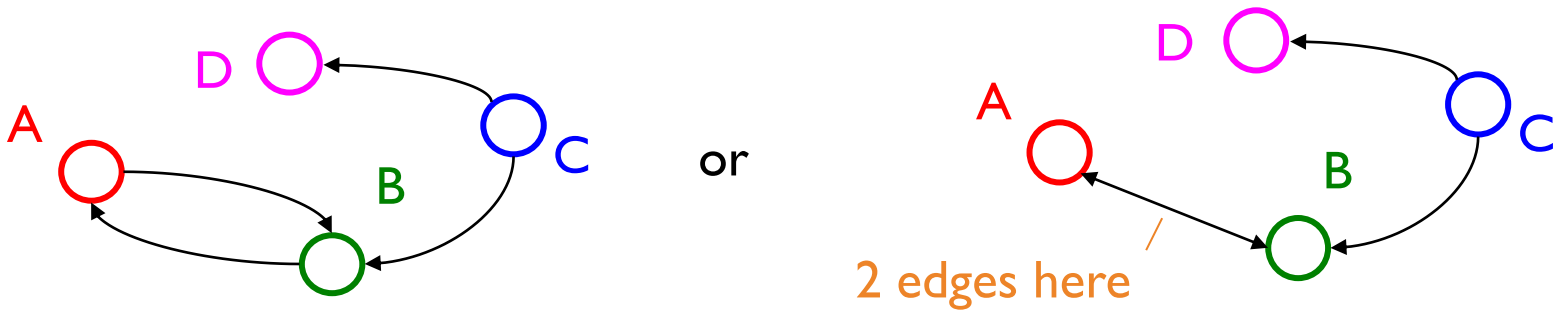
DIRECTED GRAPHS

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)

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

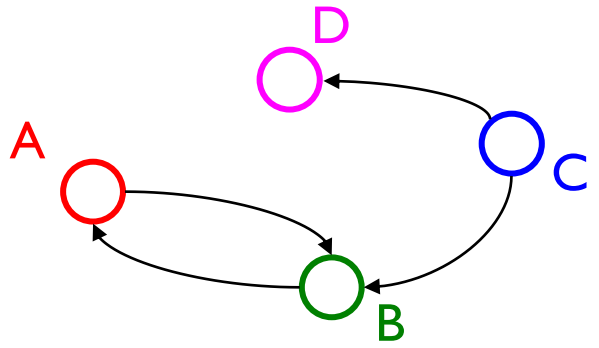
A graph does not have to be connected

- 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? 0
  - Maximum for undirected? $\frac{|V||V-1|}{2} \in O(|V|^2)$
  - Maximum for directed? $|V| * |V-1| \in O(|V|^2)$



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

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

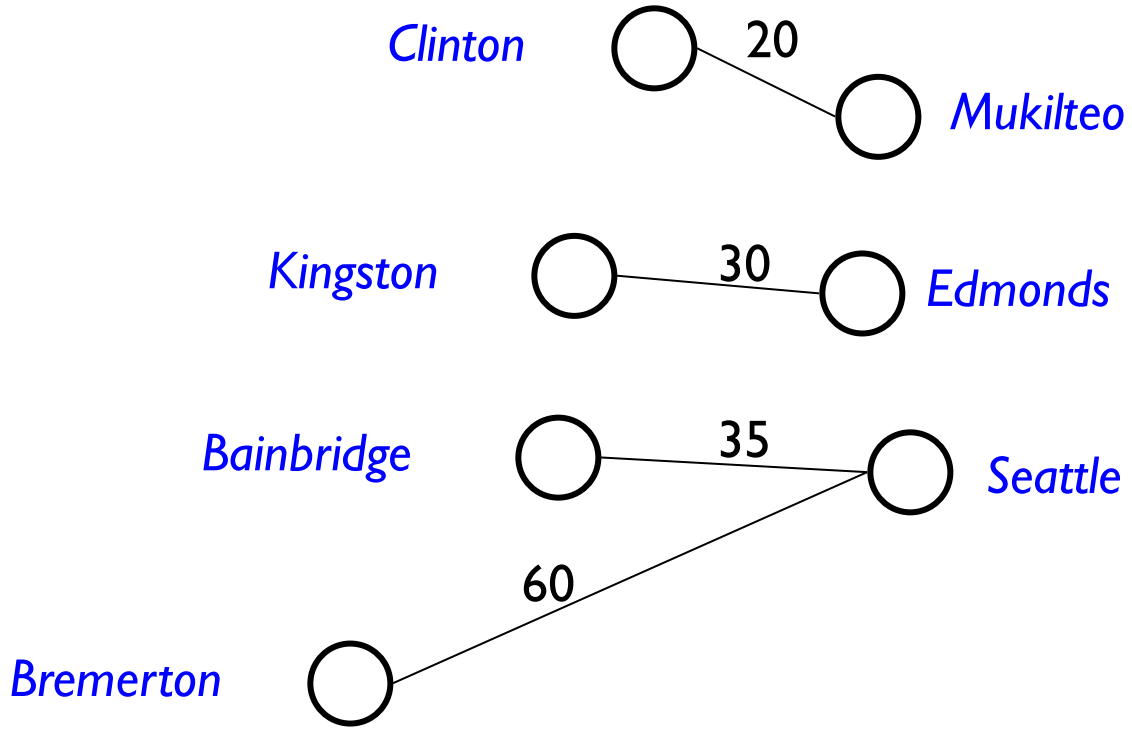
If $(u, v) \in E$, then v is a neighbor of u (i.e., v is adjacent to u)

- Order matters for directed edges: u is not adjacent to v unless $(v, u) \in E$

WEIGHTED GRAPHS

In a weighted graph, each edge has a weight or cost

- Typically, numeric (integers, decimals, doubles, etc.)
- Orthogonal to whether the graph is directed
- Some graphs allow negative weights; many do not

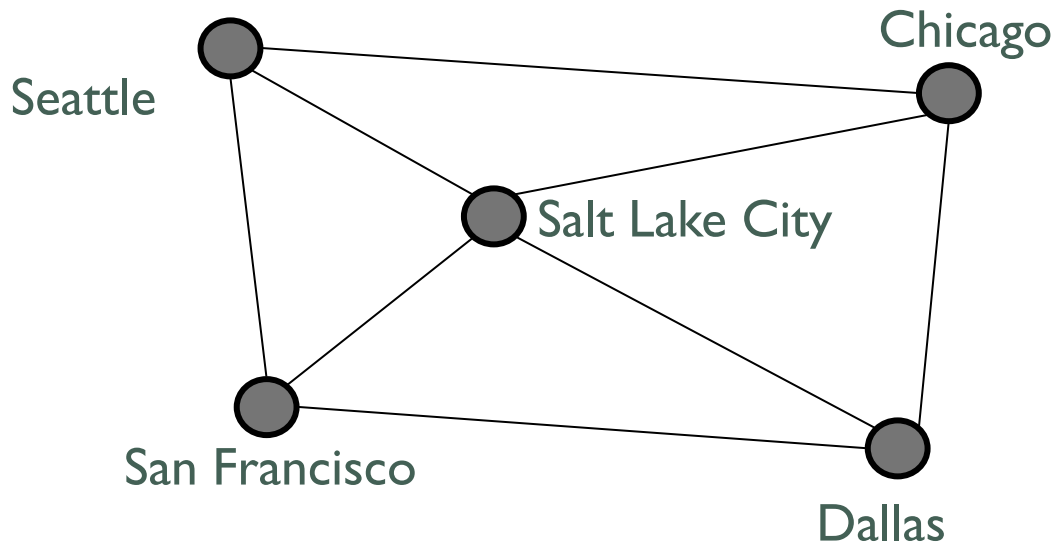


PATHS AND CYCLES

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]

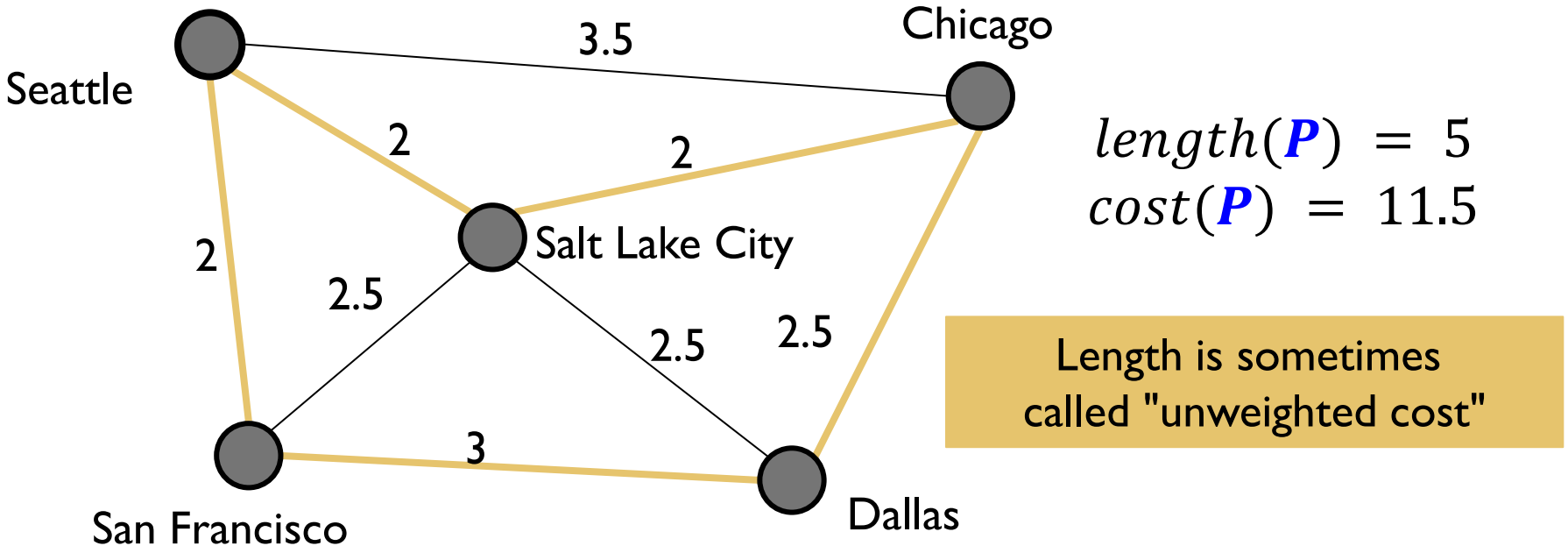


PATH LENGTH AND COST

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]



$length(\boldsymbol{P}) = 5$

$cost(\boldsymbol{P}) = 11.5$

Length is sometimes called "unweighted cost"

SIMPLE PATHS AND CYCLES

A simple path repeats no vertices (except the first might be the last):

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

A cycle is a path that ends where it begins:

- [Seattle, Salt Lake City, Seattle, Dallas, Seattle]

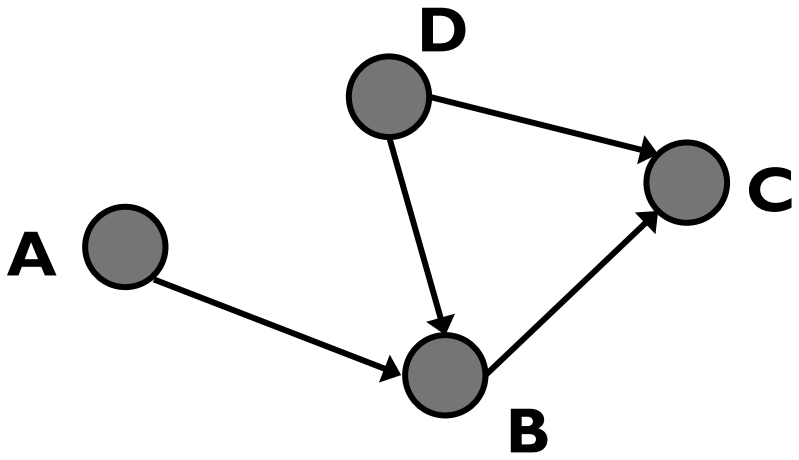
A simple cycle is a cycle and a simple path:

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

PATHS AND CYCLES IN DIRECTED GRAPHS

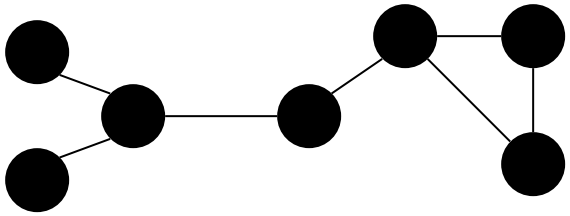
Example:

- Is there a path from A to D? NO
- Does the graph contain any cycles? NO

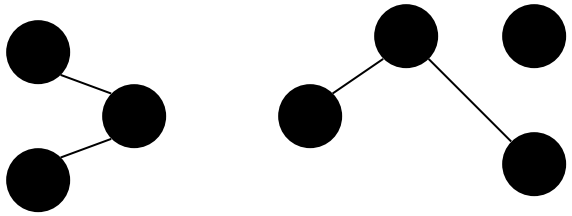


UNDIRECTED GRAPH CONNECTIVITY

An UG is connected if for all pairs of vertices $u \neq v$, there exists a *path* from u to v

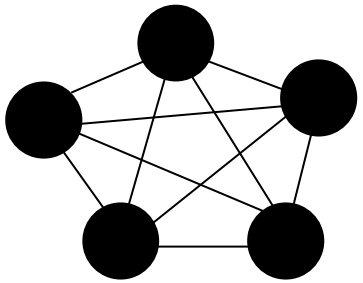


Connected graph



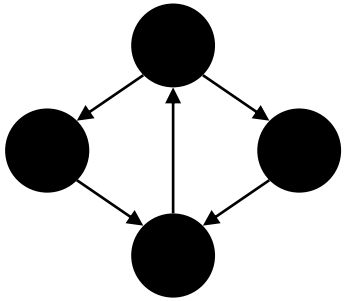
Disconnected graph

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

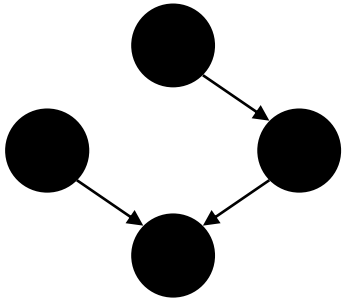


DIRECTED GRAPH CONNECTIVITY

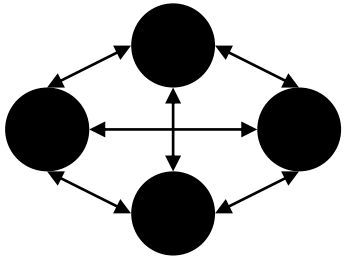
A directed graph is strongly connected if there is a path from every vertex to every other vertex



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



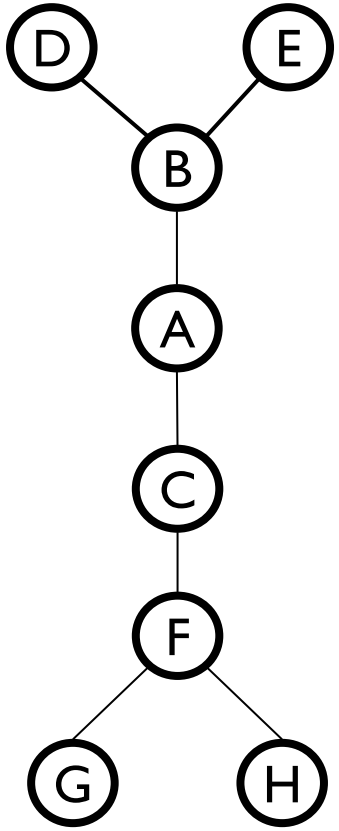
TREES AS GRAPHS

When talking about graphs, we say a tree is a graph that is:

- undirected
- acyclic
- connected

All trees are graphs, but NOT all graphs are trees

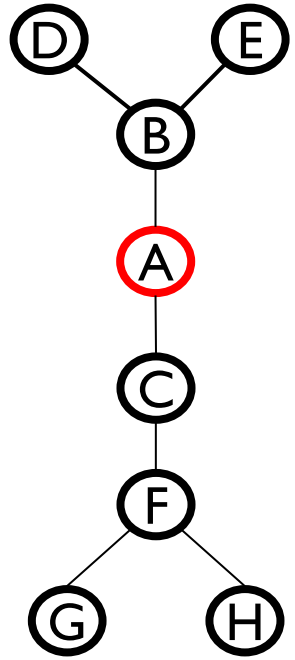
How does this relate to the trees we know?



ROOTED TREES

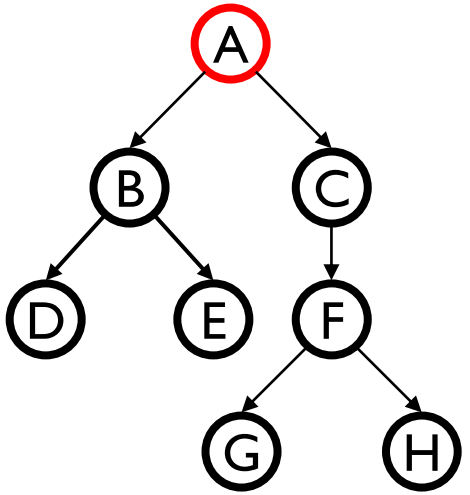
We are more accustomed to rooted trees where:

- We identify a unique root
- We think of edges as directed: parent to children



Picking a root gives a unique rooted tree

- The tree is simply drawn differently and with undirected edges



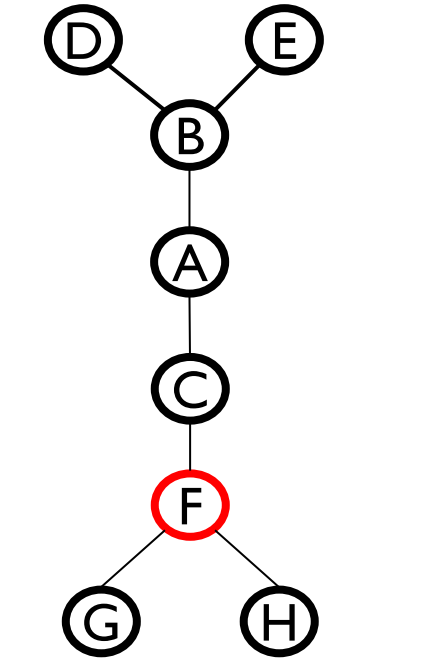
ROOTED TREES

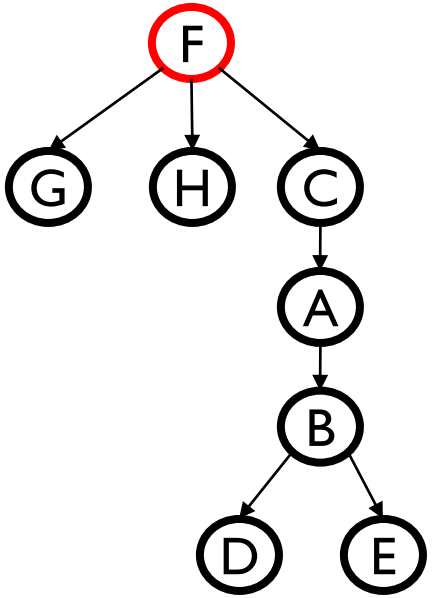
We are more accustomed to rooted trees where:

- We identify a unique root
- We think of edges as directed: parent to children

Picking a root gives a unique rooted tree

- The tree is simply drawn differently and with undirected edges

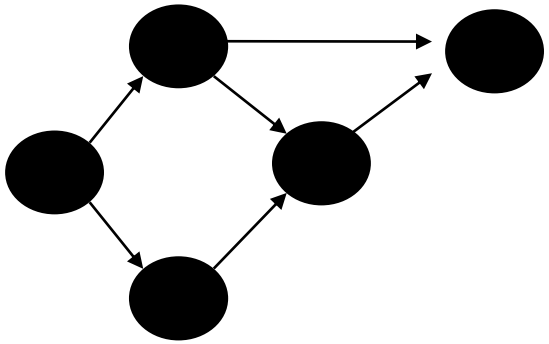




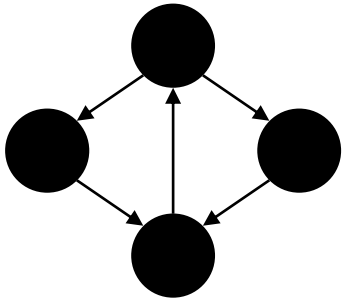
DIRECTED ACYCLIC GRAPHS (DAGS)

A DAG is a directed graph with no directed cycles

- Every rooted directed tree is a DAG
- But not every DAG is a rooted directed tree



- Every DAG is a directed graph
- But not every directed graph is a DAG



DENSITY / SPARSITY

Recall:

In an undirected graph, $0 \leq |E| < |V|^2$

Recall:

In a directed graph, $0 \leq |E| \leq |V|^2$

So, for any graph, $|E|$ is $O|V|^2)$

Another fact:

If an undirected graph is *connected*, then $|E| \geq |V| - 1$ (pigeonhole principle)

DENSITY / SPARSITY

$|E|$ is often much smaller than its maximum size

We do not always approximate as $|E|$ as $O(|V|^2)$

- This is a correct bound, but often not tight

If $|E|$ is $\Theta(|V|^2)$ (the bound is tight), we say the graph is dense

- More sloppily, dense means "lots of edges"

If $|E|$ is $O(|V|)$ we say the graph is sparse

- More sloppily, sparse means "most possible edges missing"

GRAPH DATA STRUCTURES

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

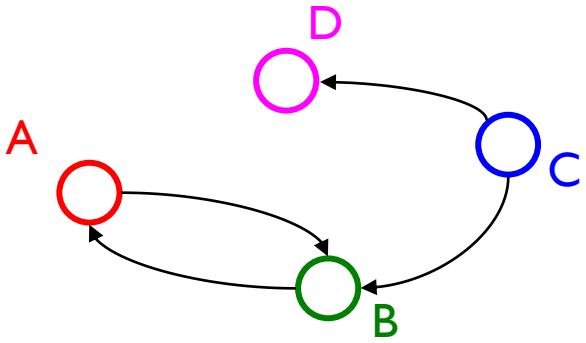
- Adjacency Matrix and Adjacency List
- Different trade-offs, particularly time versus space

ADJACENCY MATRIX

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

A $|V| * |V|$ 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:

- Storage space: $O(|V|^2)$
- Adding a vertex: $O(|V|^2)$
- Removing a vertex: $O(|V|^2)$
- Adding an edge: $O(1)$
- Removing an edge: $O(1)$
- Decide if some edge exists: $O(1)$
- Get a vertex's out-edges: $O(|V|)$
- Get a vertex's in-edges: $O(|V|)$

Best for sparse or dense graphs? Dense

	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

How will the adjacency matrix vary for an undirected graph?

- Will be symmetric about the diagonal axis
- Matrix: could we save space by using only about half the array?
- But how would you "get all neighbors"?

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

ADJACENCY MATRIX PROPERTIES

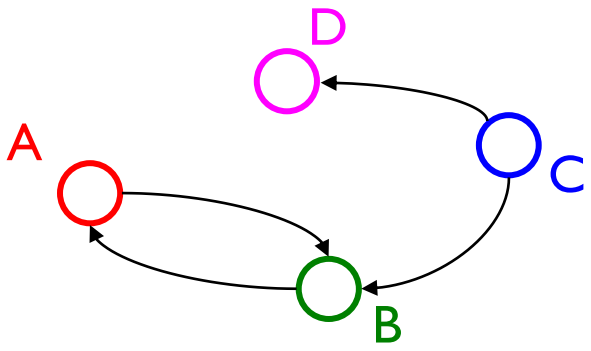
How can we adapt the representation for weighted graphs?

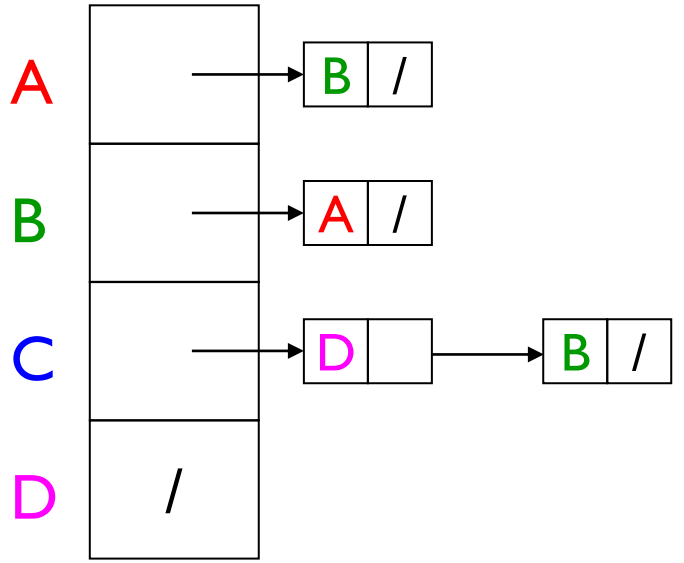
- Instead of Boolean, store a number in each cell
- Need some value to represent 'not an edge'
  - $0, -1,$ or some other value based on how you are using the graph
  - Might need to be a separate field if no restrictions on weights

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)



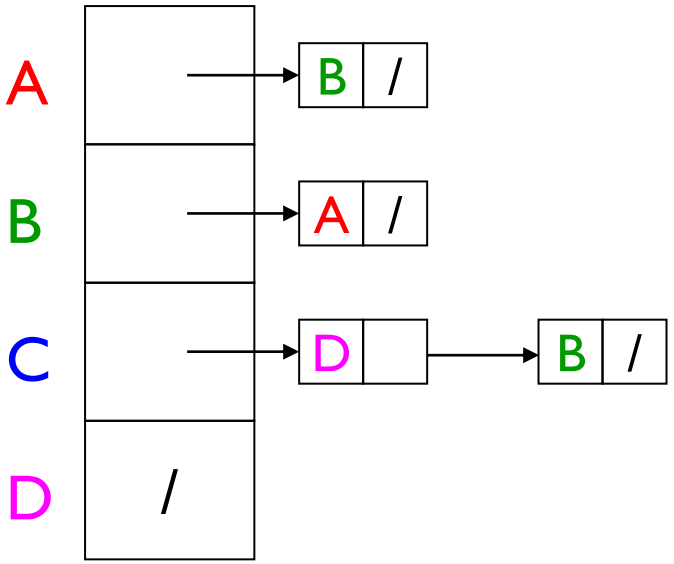


ADJACENCY LIST PROPERTIES

Running time to:

- Storage space: $O(|V| + |E|)$
- Adding a vertex: $O(|V|)$
- Removing a vertex: $O(|V| + |E|)$
- Adding an edge: $O(|V|)$
- Removing an edge: $O(|V|)$
- Decide if some edge exists: $O(|V|)$
- Get a vertex's out-edges: $O(|V|)$
- Get a vertex's in-edges: $O(|V| + |E|)$

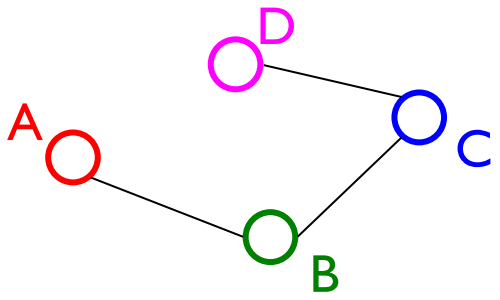
Best for sparse or dense graphs? *Sparse*

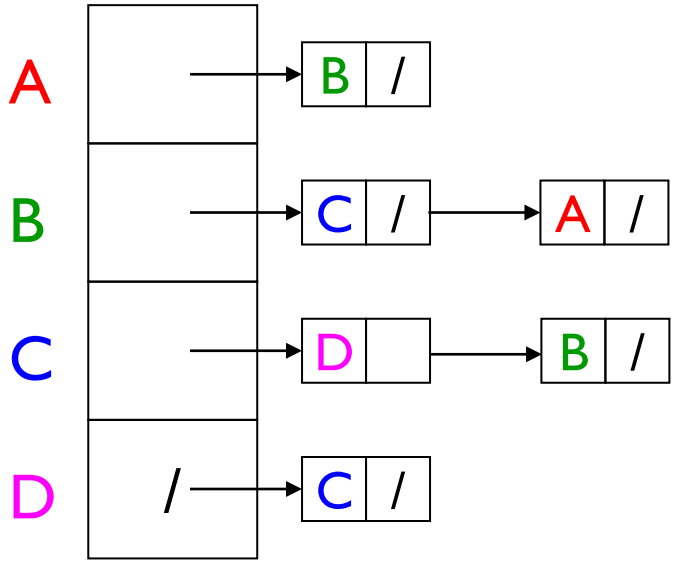


UNDIRECTED GRAPHS

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

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

APPLICATIONS OF GRAPHS: TRAVERSALS

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?

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 on how add/remove work are implemented

- DFS: a stack "depth-first graph search“ / Last In First Out
- BFS: a queue "breadth-first graph search“ / First In First Out

DFS and BFS are "big ideas" in computer science

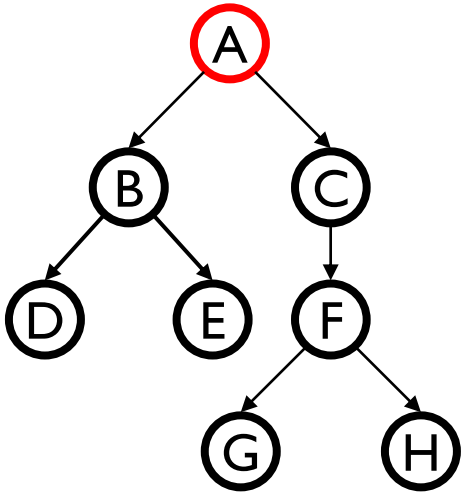
- Depth: recursively explore one part before going back to the other parts not yet explored
- Breadth: Explore areas closer to start node first

RECURSIVE DFS, EXAMPLE WITH TREE

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

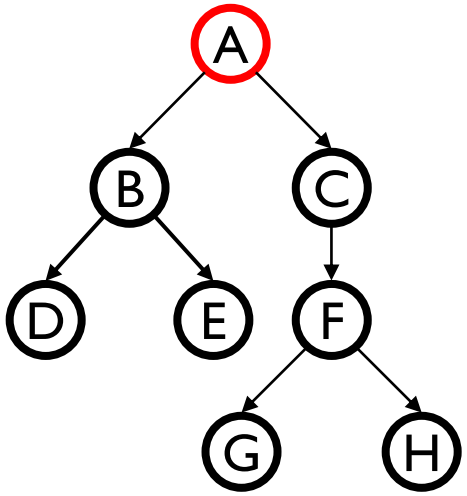
Order processed: A, B, D, E, C, F, G, H

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

DFS WITH STACK, EXAMPLE WITH TREE

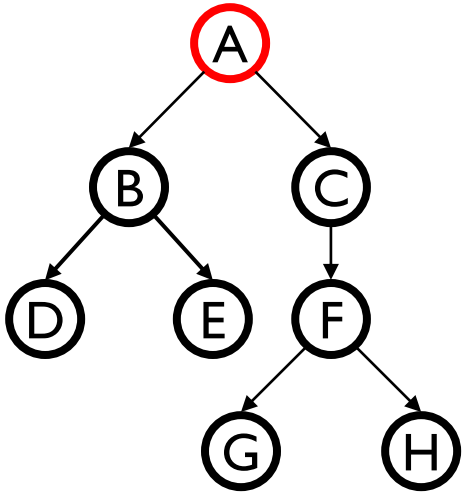



```
DFS2(Node start) {
   initialize stack s to hold start
   mark start as visited
   while(s is not empty) {
        next = s.pop() // and "process"
        for each node u adjacent to next
            if(u is not marked)
                mark u and push onto s
   }
}
```

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 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 the longest path in the graph is p and the highest out-degree is d, then DFS stack never has more than $d \times p$ 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?

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 output the path?

Easy:

- Store the previous node along the path:
  When processing u causes us to add v to the search, set the $v.path$ field to be u)
- When you reach the goal, follow path fields back to where you started (and then reverse the answer)

MORE GRAPH ALGORITHMS

Strongly connected components
Spanning trees
Finding a shortest path is one thing
What happens when we consider weighted edges (as in distances)?

QUESTIONS/ANSWERS

REFERENCES

CSE 332 Data Abstractions: Graphs and Graph Traversals. Kate Deibel, Summer 2012