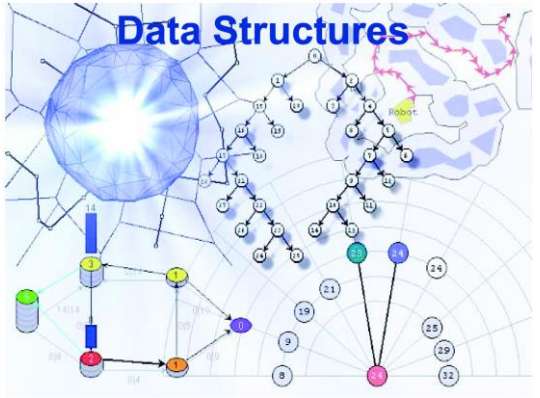
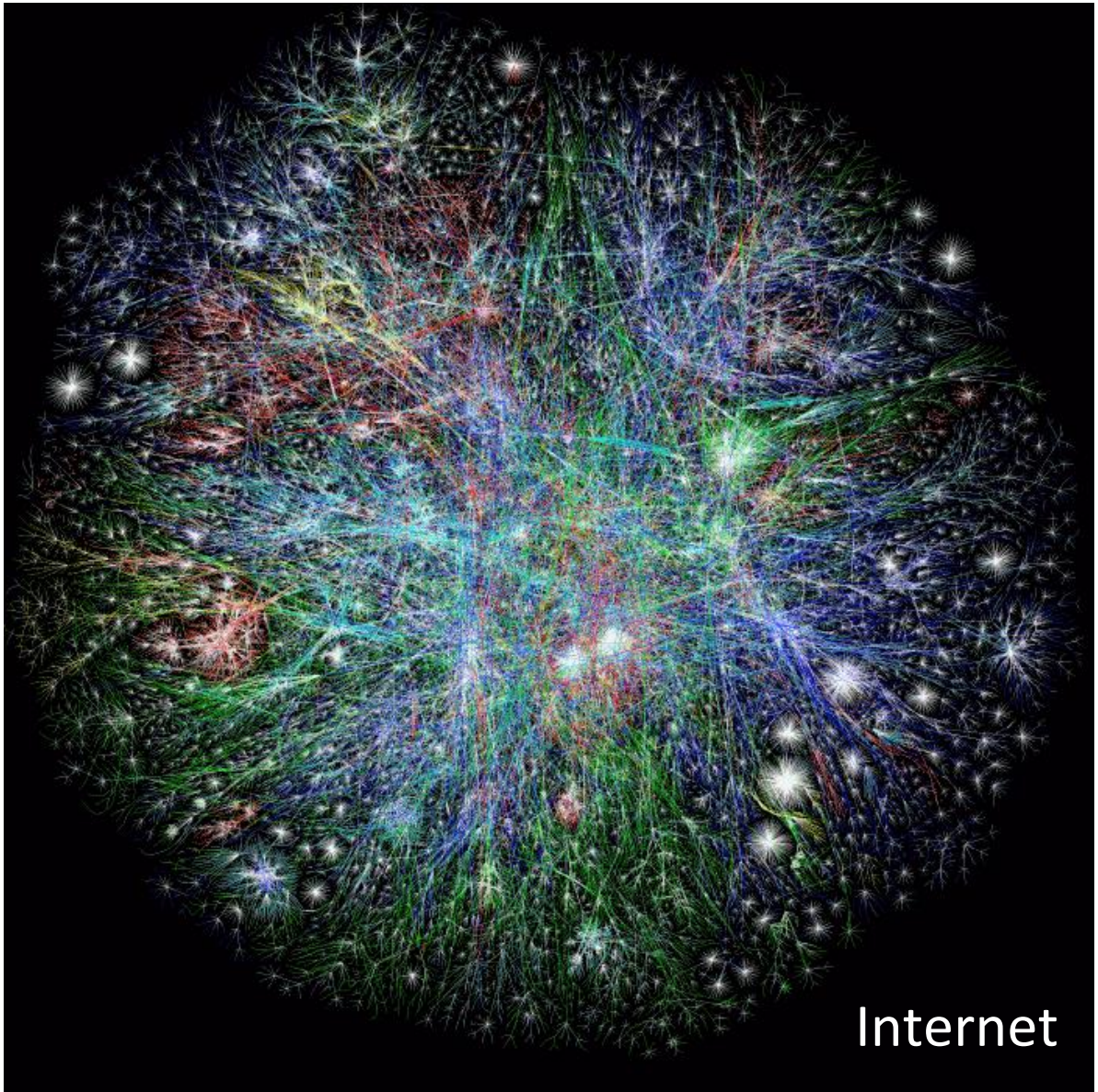
BBM 201
DATA STRUCTURES

Lecture 11:
Graphs

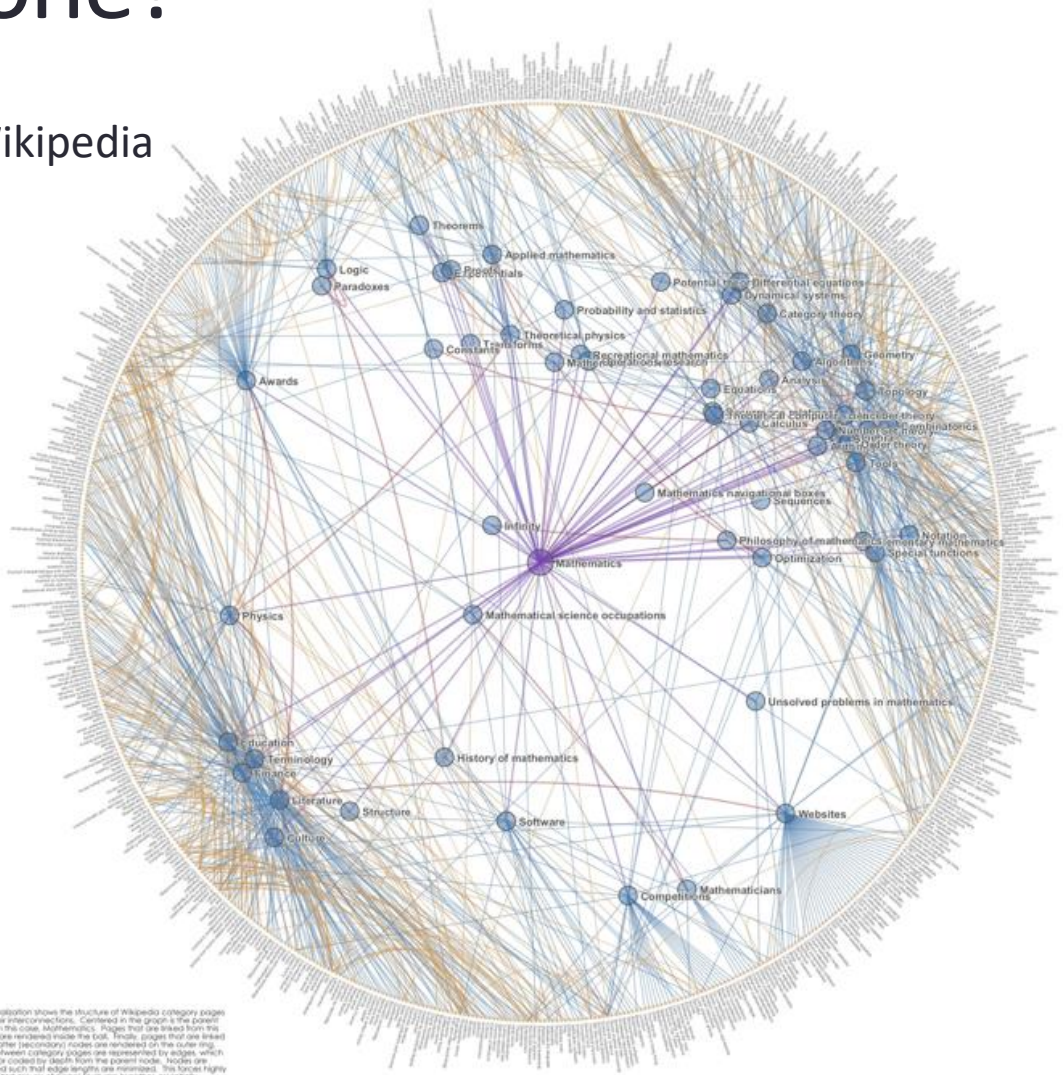


What does this graph represent?



And this one?

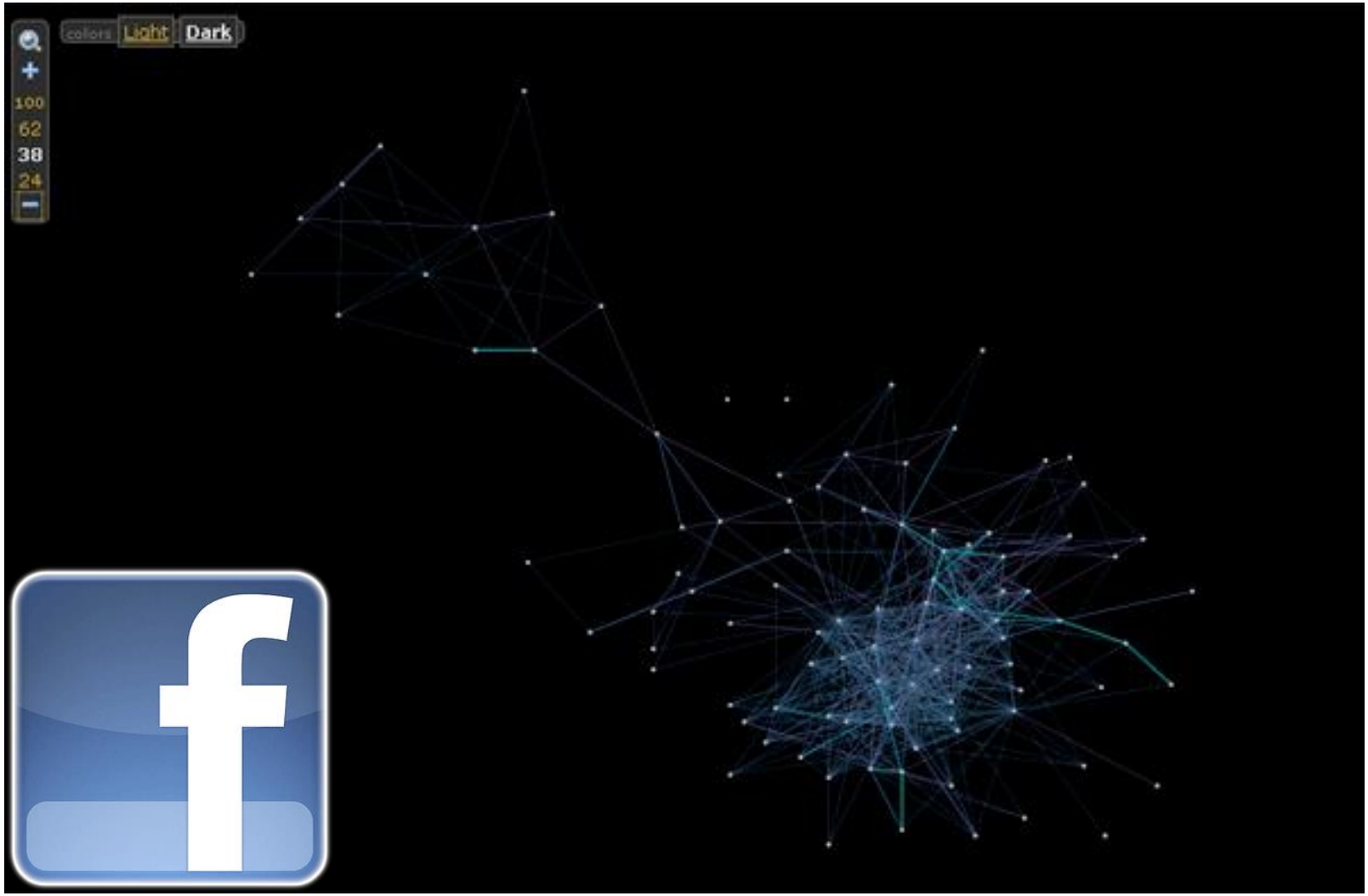
Math articles on Wikipedia



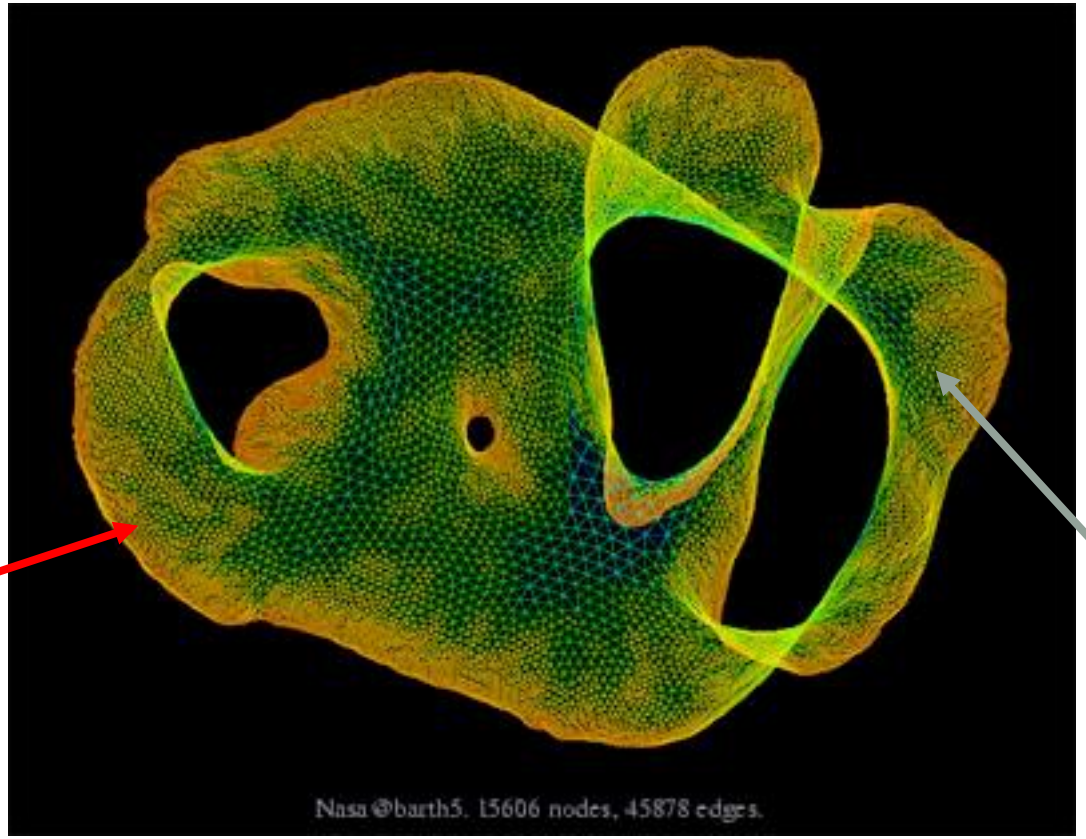
This visualization shows the structure of Wikipedia category pages and their interconnections. Centered in the graph is the parent node, in this case, Mathematics. Pages that are linked from this parent are rendered inside the disk. Finally, pages that are linked to the latter (secondary) nodes are rendered on the outer ring. Links between category pages are represented by edges, which are color coded by depth from the parent node. Nodes are clustered such that edge lengths are minimized. This forces highly connected groups of pages to clump together, essentially forming topical groups. Created by Chris Harrison, 2007.

ChrisHarrison.net

And this one?



What about large graphs?



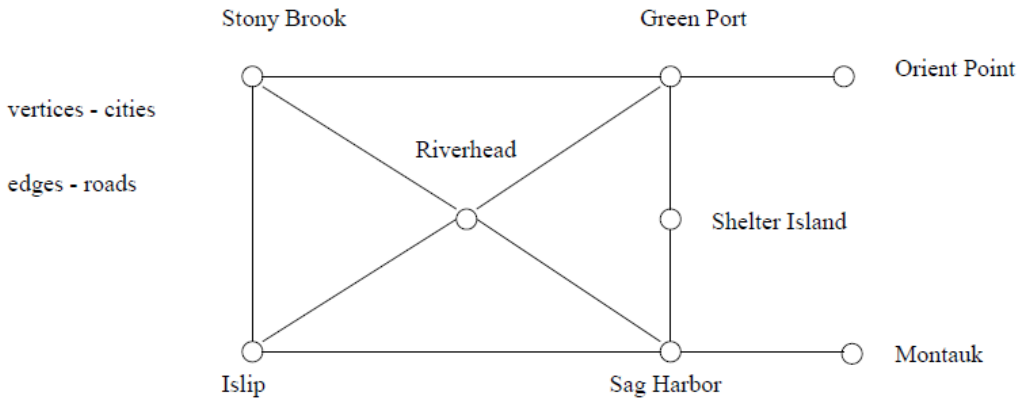
Are s and t connected?

Graphs

- Graphs are one of the unifying themes of computer science.
- A graph G = (V, E) is defined by a set of *vertices V , and* a set of *edges E consisting of ordered or unordered pairs of* vertices from V .

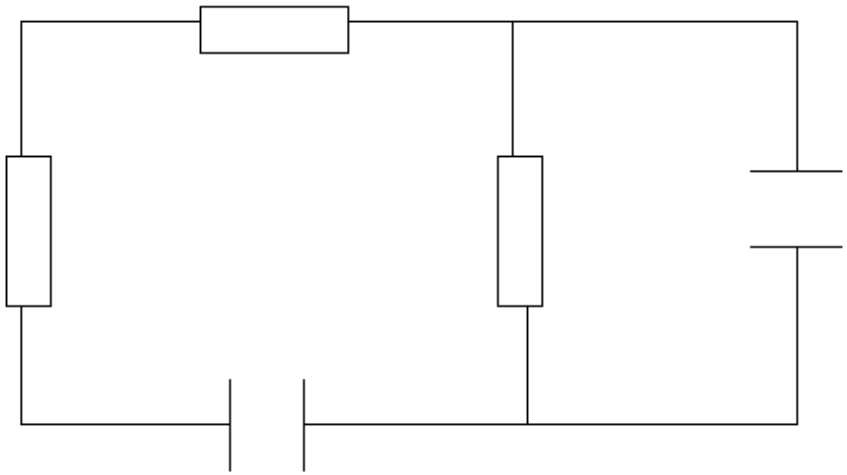
Road Networks

- In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.



Electronic Circuits

- In an electronic circuit, with junctions as vertices and components as edges.



vertices: junctions

edges: components

Applications

- Social networks (facebook ...)
- Courses with prerequisites
- Computer networks
- Google maps
- Airline flight schedules
- Computer games
- WWW documents
- ... (so many to list!)

Graph ADT

A graph is a formalism for representing relationships among items

- Very general definition
- Very general concept

We can think of graphs as an ADT

- Operations would include $\mathrm{isEdge}(v_j, v_k)$
- 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

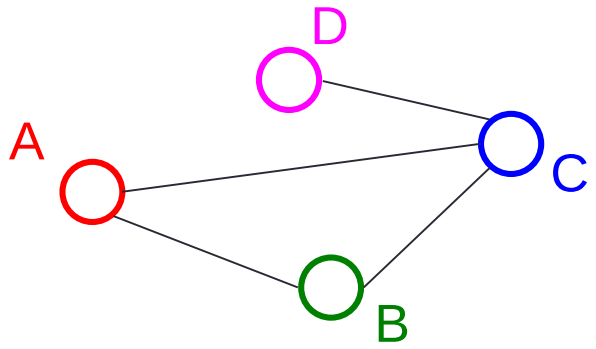
Graphs

- 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 and many problems in different disciplines use graph models to some extent
- To make formulating graphs easy and standard, we have a lot of standard *terminology*.
  - Yet, there is considerable variation in some terminology referring to similar concepts with slight differences; such as path, walk, trail, cycle, circuit, closed walk etc. Pay attention to their exact definitions in the given contexts.

Undirected Graphs

In undirected graphs, edges have no specific direction

- Edges are always "two-way"



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

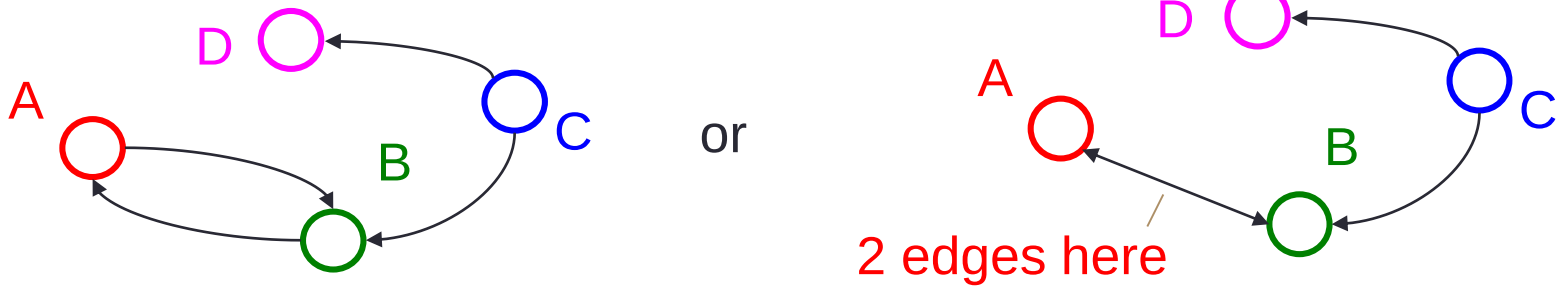
- Only one of these edges needs to be in the set
- The other is implicit, so normalize how you check for it

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)

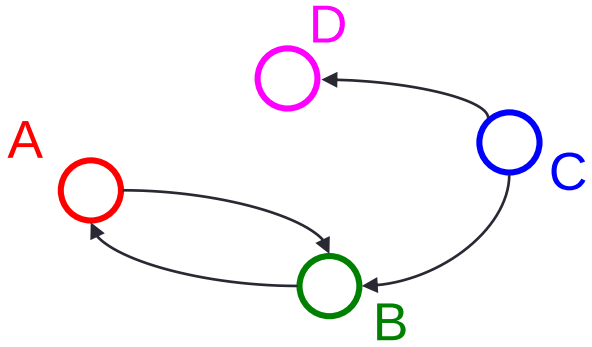
- The use/algorithm usually dictates if a graph has:
  - No self edges
  - Some self edges
  - All self edges

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 $G = (V, E)$:

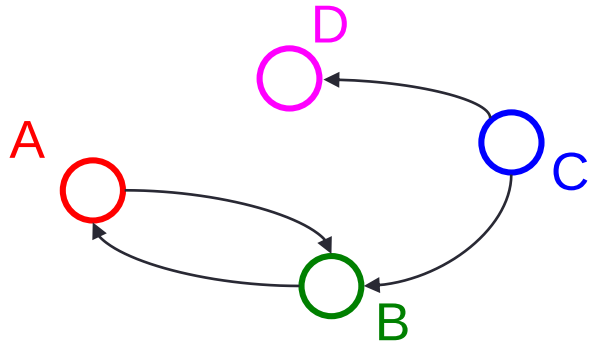
- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?

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

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 $(v, u) \in E$

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? $|V||V+1|/2 \in O(|V|^2)$ assuming self loop
  - Maximum for directed? $|V|^2 \in O(|V|^2)$

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 $(v, u) \in E$

Examples Again

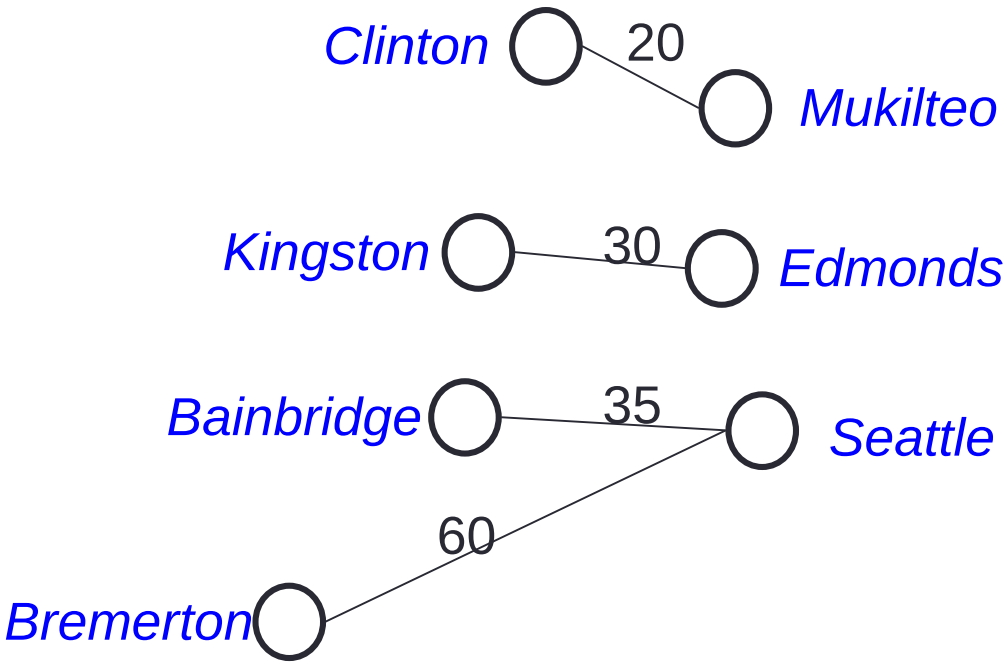
Which would use directed edges?
Which would have self-edges?
Which could have 0-degree nodes?

- Web pages with links
- Facebook friends
- 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, 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?

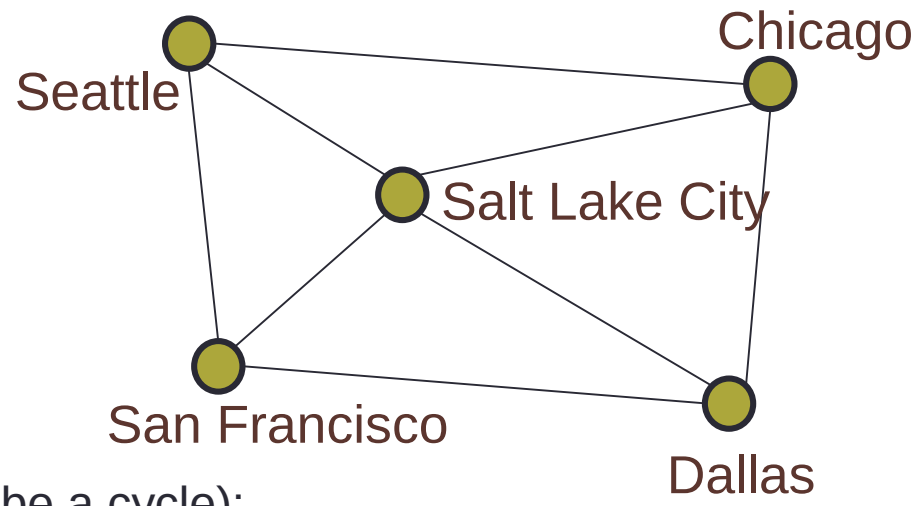
Do negative weights make sense?

- Web pages with links
- Facebook friends
- 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 $[v_0, v_1, ..., v_n]$ 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]

*in some resources, circuit is the equivalent terminology referring to cycles. Regarding pat

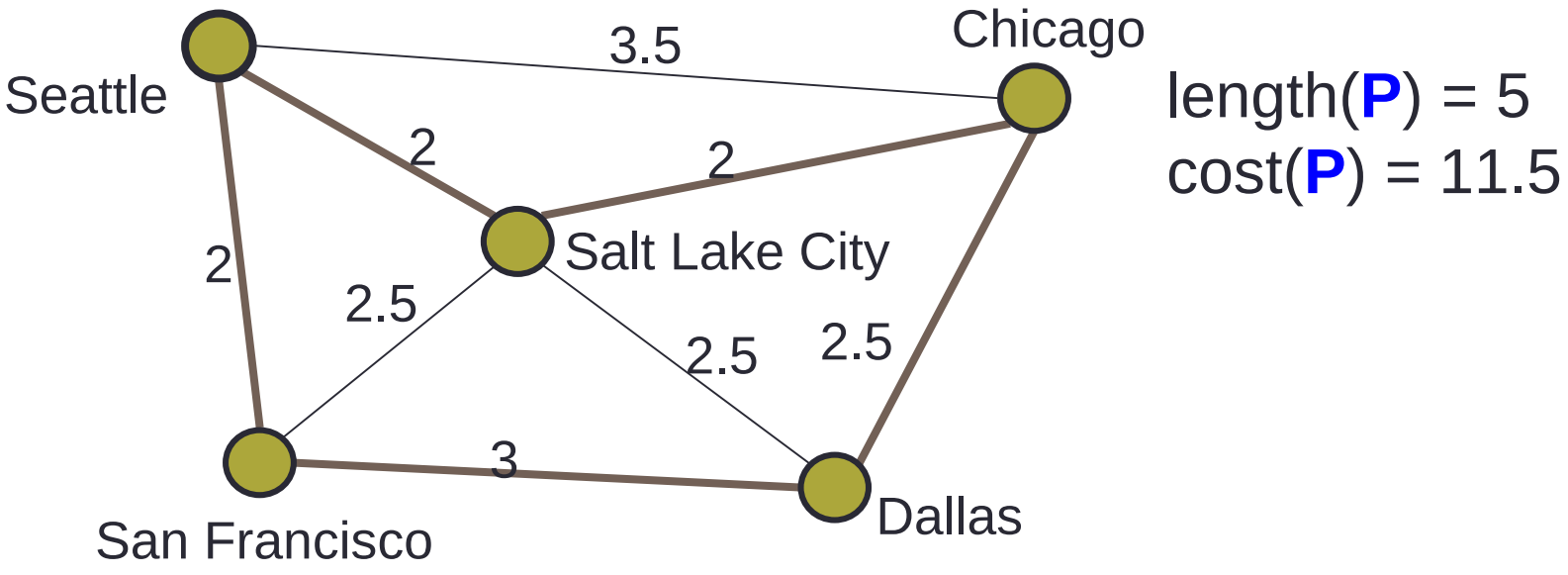
Path Length and Cost

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]



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]

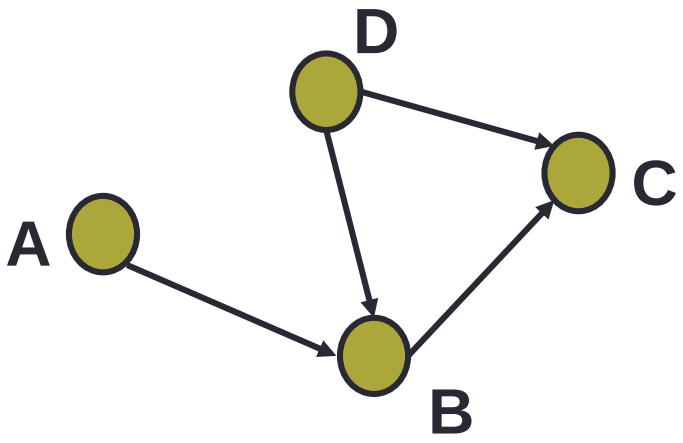
A simple cycle is a cycle and a simple path:

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

*Notice that in some resources simple path refers to the path that does not contain the sa

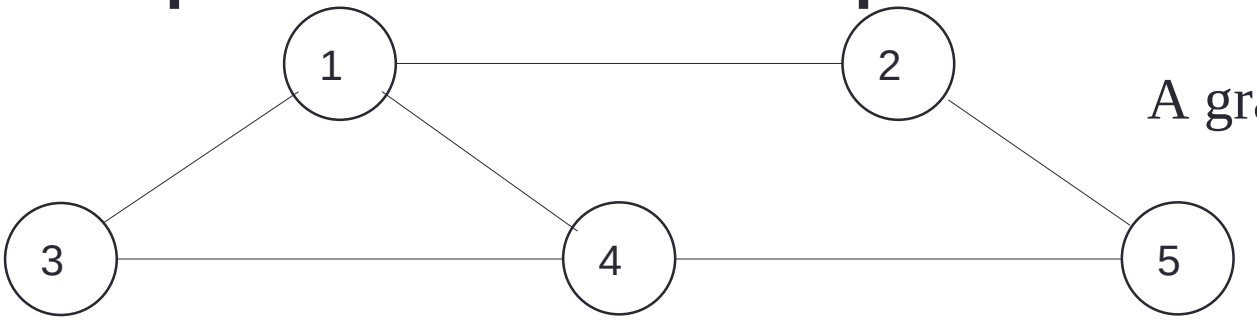
Paths and Cycles in Directed Graphs

Example:



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

Graphs – An Example



A graph G (undirected)

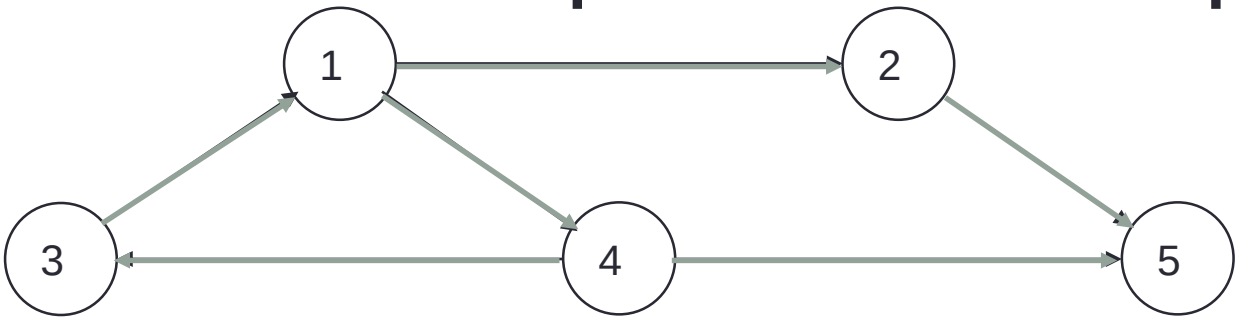
The graph G= (V,E) has 5 vertices and 6 edges:

V = {1,2,3,4,5}

E = { (1,2),(1,3),(1,4),(2,5),(3,4),(4,5), (2,1),(3,1),(4,1),(5,2),(4,3),(5,4) }

- *Adjacent:*
  1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1
- *Path:*
  1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)
- *Cycle:*
  1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)

Directed Graph – An Example



The graph G= (V,E) has 5 vertices and 6 edges:

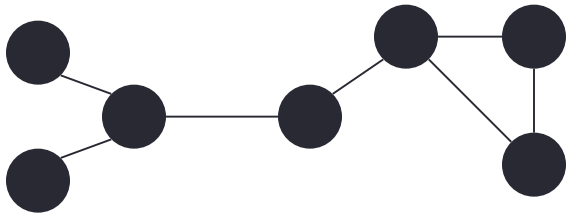
V = {1,2,3,4,5}

E = { (1,2),(1,4),(2,5),(4,5),(3,1),(4,3) }

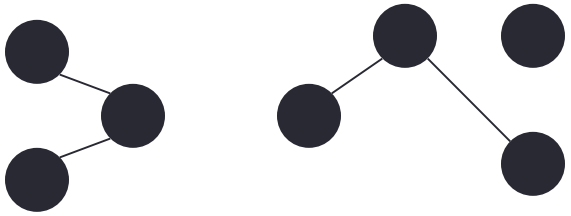
- *Adjacent:*
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- *Path:*
  1,2,5 (a directed path),
- *Cycle:*
  1,4,3,1 (a directed cycle),

Undirected Graph Connectivity

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

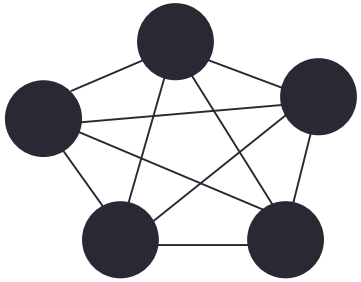


Connected graph



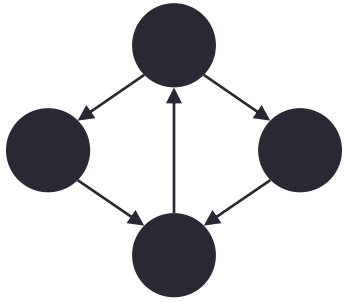
Disconnected graph

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

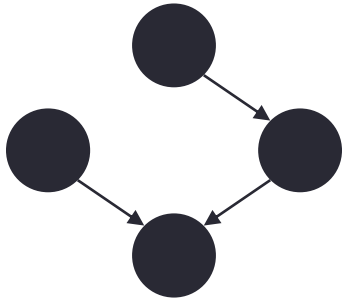


Directed Graph Connectivity

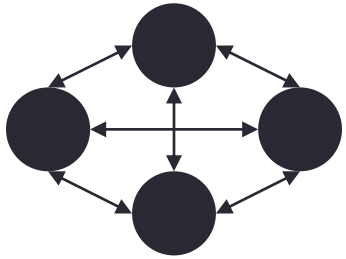
A directed graph is strongly connected if there is a path from every vertex to every other vertex



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



Examples Again

For undirected graphs: connected?
For directed graphs: strongly connected?
weakly connected?

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

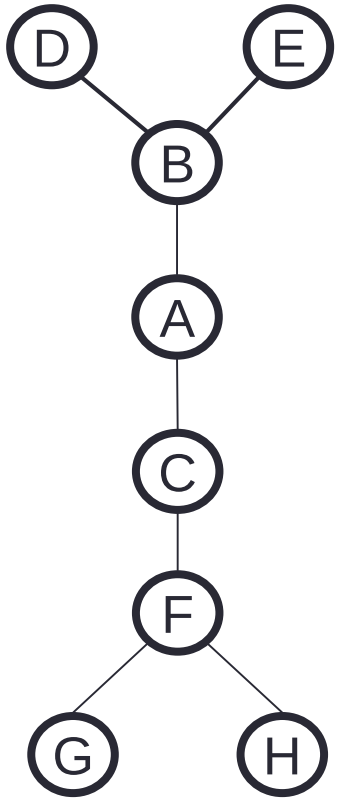
Trees as Graphs

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 and "love"?



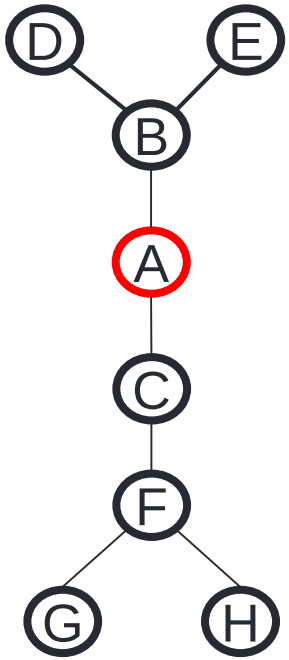
Rooted Trees

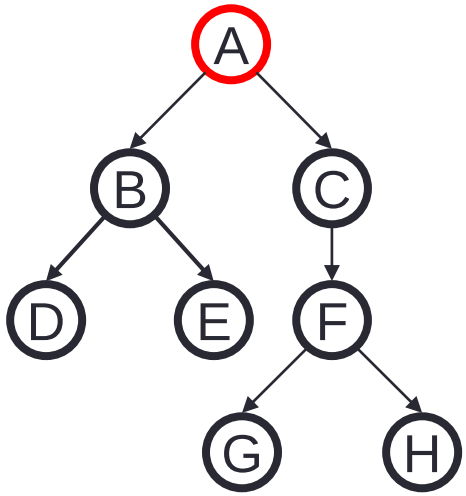
We are more accustomed to rooted trees where:

- 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





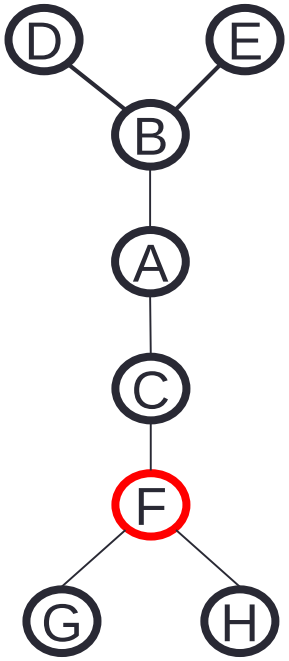
Rooted Trees

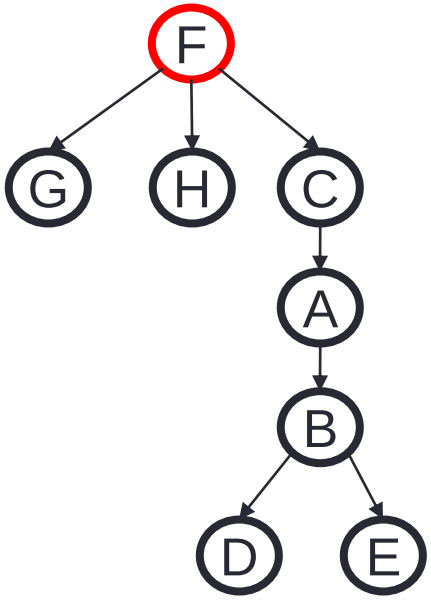
We are more accustomed to rooted trees where:

- 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

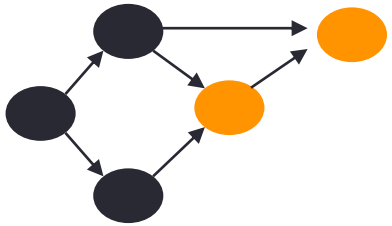




Directed Acyclic Graphs (DAGs)

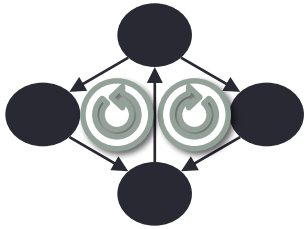
A DAG is a directed graph with no cycles

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



A DAG that is not a rooted directed tree

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



A directed graph with cycles

Examples Again

Which of our directed-graph examples do you expect to be a DAG?

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

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 simple graph is *connected*, then $|E| \geq |V|\text{-}1$

(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"

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

Graph Implementations

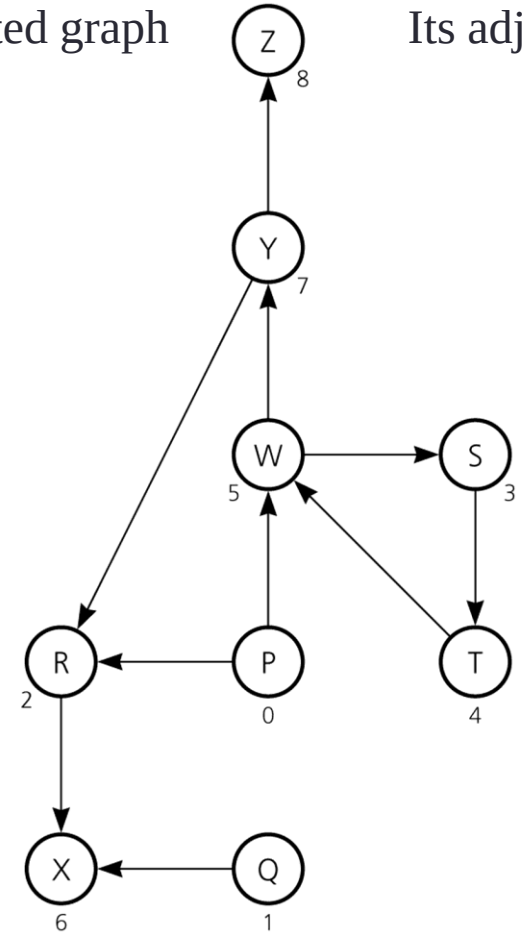
- The two most common implementations of a graph are:
  - ***Adjacency Matrix***
    - A two dimensional array
  - ***Adjacency List***
    - For each vertex we keep a list of adjacent vertices

Adjacency Matrix

- An **adjacency matrix** for a graph with *n* vertices is an ***n* by *n* array *matrix*** such that
  - *matrix[i][j]* is 1 (true) if there is an edge from vertex *i* to vertex *j*
  - 0 (false) otherwise.
- When the graph is **weighted:**
  - *matrix[i][j]* is weight that labels edge from vertex *i* to vertex *j* instead of simply 1,
  - *matrix[i][j]* equals to ∞ instead of 0 when there is no edge from vertex *i* to *j*
- Adjacency matrix for an undirected graph is symmetrical.
  - i.e. *matrix[i][j]* is equal to *matrix[j][i]*
- Space requirement **$O(|V|^2)$**
  - Acceptable if the graph is dense.

Adjacency Matrix – Example1

A directed graph

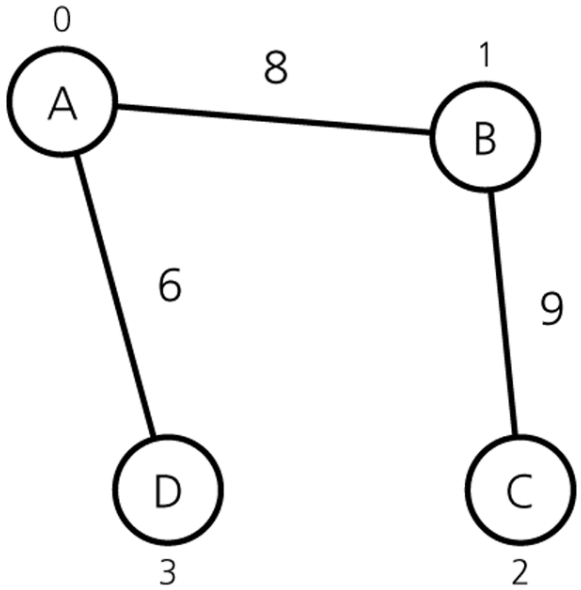


Its adjacency matrix

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Adjacency Matrix – Example2

An Undirected Weighted Graph



Its Adjacency Matrix

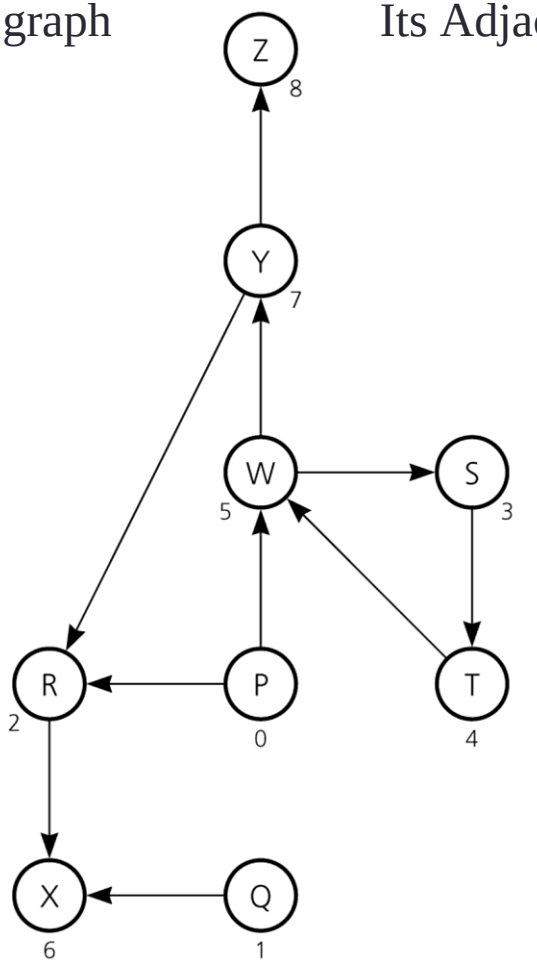
		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency List

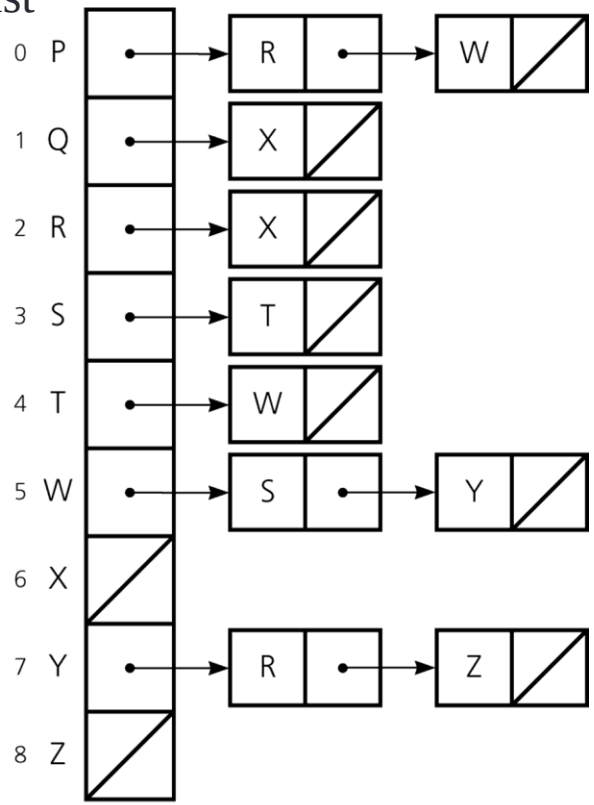
- An **adjacency list** for a graph with *n* vertices numbered 0,1,...,n-1 consists of *n* linked lists. The i^{th} linked list has a node for vertex *j* if and only if the graph contains an edge from vertex *i* to vertex *j*.
- Adjacency list is a better solution if the graph is sparse.
- Space requirement is **O(|E| + |V|),** which is linear in the size of the graph.
- In an undirected graph each edge (v, w) appears in two lists.
  - Space requirement is doubled.

Adjacency List – Example1

A directed graph

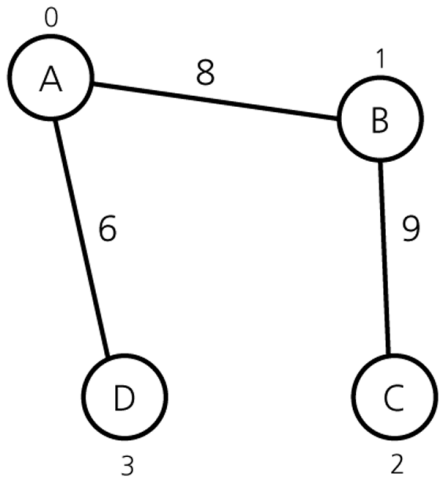


Its Adjacency List

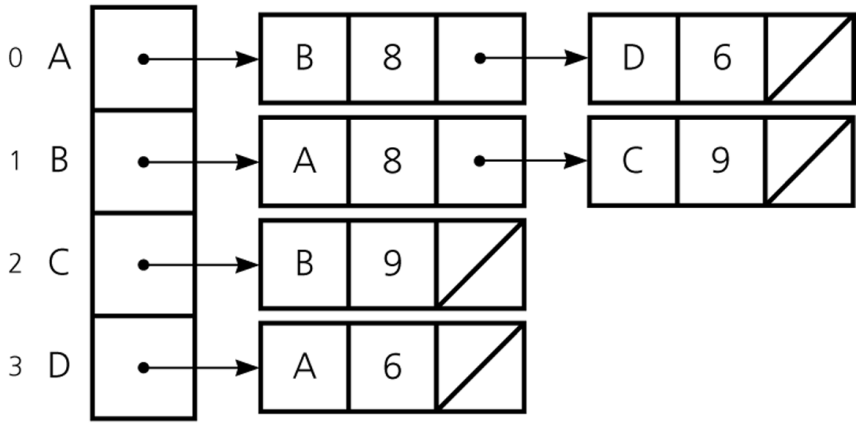


Adjacency List – Example2

An Undirected Weighted Graph



Its Adjacency List



Adjacency Matrix vs Adjacency List

- Two common graph operations:
  1. **Determine whether there is an edge** from vertex i to vertex j.
  2. **Find all vertices adjacent** to a given vertex i.

- An adjacency matrix supports operation 1 more efficiently.
- An adjacency list supports operation 2 more efficiently.

- An adjacency list often requires less space than an adjacency matrix.
  - Adjacency Matrix: Space requirement is $O(|V|^2)$
  - Adjacency List : Space requirement is $O(|E| + |V|)$, which is linear in the size of the graph.
  - Adjacency matrix is better if the graph is dense (too many edges)
  - Adjacency list is better if the graph is sparse (few edges)

Tradeoffs Between Adjacency Lists and Adjacency Matrices

- Faster to test if (x; y) exists?
- Faster to find vertex degree?
- Less memory on sparse graphs?
- Less memory on dense graphs?
- Edge insertion or deletion?
- Faster to traverse the graph?
- Better for most problems?

Tradeoffs Between Adjacency Lists and Adjacency Matrices

- Faster to test if (x; y) exists? **matrices**
- Faster to find vertex degree? **lists**
- Less memory on sparse graphs? **lists – (m+ n) vs. (n^2)**
- Less memory on dense graphs? **matrices (small win)**
- Edge insertion or deletion? **matrices O(1)**
- Faster to traverse the graph? **lists – (m + n) vs. (n^2)**
- Better for most problems? **lists**

Graph Representations in C++

- No graph in STL but alternatives available
  - Boost
  - LEMON
  - LEDA

- Combine STL data structures to represent a graph
  - Best representation depends on the application

How do we represent a graph in C++?

Robert Sedgewick, Algorithms in C++, Part 5 (Graph Algorithms) (code)

Adjacency Matrix

```
set<Node> nodes;
map<Node, map<Node, bool> > adjacencyMatrix;
```

```
void addNode(){
    nodes.insert(node);
    map<Node,bool> matrixRow;
    for(set<Node>::iterator it = nodes.begin();
            it != nodes.end();
            it++){
        matrixRow.insert(make_pair(*it,false));
        adjacencyMatrix.at(*it).insert(make_pair(node, false));
    }
    adjacencyMatrix.insert(make_pair(node,matrixRow));
}
```

Adjacency Matrix

```
set<Node> nodes;
map<Node, map<Node, bool> > adjacencyMatrix;
```

```
void addEdge(Node node1, Node node2){
    adjacencyMatrix.at(node1).at(node2) = true;
    adjacencyMatrix.at(node2).at(node1) = true;
}
```

```
bool connected(Node node1, Node node2){
    return adjacencyMatrix.at(node1).at(node2);
}
```

Adjacency Matrix

```
set<Node> nodes;
map<Node, map<Node, bool> > adjacencyMatrix;
```

- Space
  - nodes.size() = number of nodes, n
  - adjacency matrix has 1 entry for every pair of nodes, $O(n^2)$
  - Total space is $O(n^2)$

Adjacency Matrix

```
set<Node> nodes;
map<Node, map<Node, bool> > adjacencyMatrix;
```

```
bool connected(Node node1, Node node2){
    return adjacencyMatrix.at(node1).at(node2);
}
```

- Runtime of connected
  - O(log(n))
  - Can get O(1) with other data structures
    - unordered_map
    - vector

map: Balanced Binary Tree implementation

Adjacency Matrix - Alternate

```
vector<Node> nodes;
vector<vector<bool> > adjacencyMatrix;
```

- If nodes have consecutive ID’s starting at 0
- Use ID’s to access vector in O(1) time
- Not robust
  - Must maintain ID’s
  - Deleting a node means a missing row and column

Nodes

- Graph stores
  - A collection of nodes
  - Tracks all edges
- Query if nodes are connected

```
class Node {
    int id;
    string payload;
};

bool operator<(const Node& node1, const Node& node2){
    return node1.id < node2.id;
}
```

Adjacency List in C++

```
set<Node> nodes;
map<Node, list<Node> > adjacencyList;
```

```
void addNode(Node node){
    nodes.insert(node);
    list<Node> nodeList;
    adjacencyList.insert(make_pair(node,nodeList));
}
```

list<T>: doubly linked list

Adjacency List in C++

```
set<Node> nodes;
map<Node, list<Node> > adjacencyList;
```

```
void addEdge(Node node1, Node node2){
    adjacencyList.at(node1).push_back(node2);
    adjacencyList.at(node2).push_back(node1);
}
```

```
list<Node> getAllNeighbors(Node node){
    return adjacencyList.at(node);
}
```

Playing it Safe

- Protection from misuse of the function

```
list<Node> getAllNeighborsSafe(Node node){
    try{
        return adjacencyList.at(node);
    }catch(out_of_range const& exception){
        cout << "Node not found. Looking for node " << node.id;
        cout << " with payload " << node.payload << endl;
        list<Node> emptyList;
        return emptyList;
    }
}
```

Adjacency List in C++

```
set<Node> nodes;
map<Node, list<Node> > adjacencyList;
```

- Space
  - nodes.size() = number of nodes, n
  - adjacency list has 1 entry for every edge, m
  - Total space is n+m
  - Great for sparse graphs
    - Small m (<< n^2)

Adjacency List in C++

```
set<Node> nodes;
map<Node, list<Node> > adjacencyList;
```

```
list<Node> getAllNeighbors(Node node){
    return adjacencyList.at(node);
}
```

- Runtime of getAllNeighbors
  - O(log(n))
  - O(m_u) to iterate list (number of neighbors of node)
- Could we return the list in O(1) time?
  - unordered_map
  - hash table

GRAPH ALGORITHMS

Graph Traversals

- A **graph-traversal** algorithm starts from a vertex v, **visits all of the vertices** that can be reachable from the vertex v.
  - A graph-traversal algorithm visits all vertices if and only if the graph is connected.
- A **connected component** is the subset of vertices visited during a traversal algorithm that begins at a given vertex.
- A graph-traversal algorithm must **mark each vertex** during a visit and must never visit a vertex more than once.
  - Thus, if a graph contains a cycle, the graph-traversal algorithm can avoid infinite loop.
- We look at two graph-traversal algorithms:
  - **Depth-First Traversal**
  - **Breadth-First Traversal**

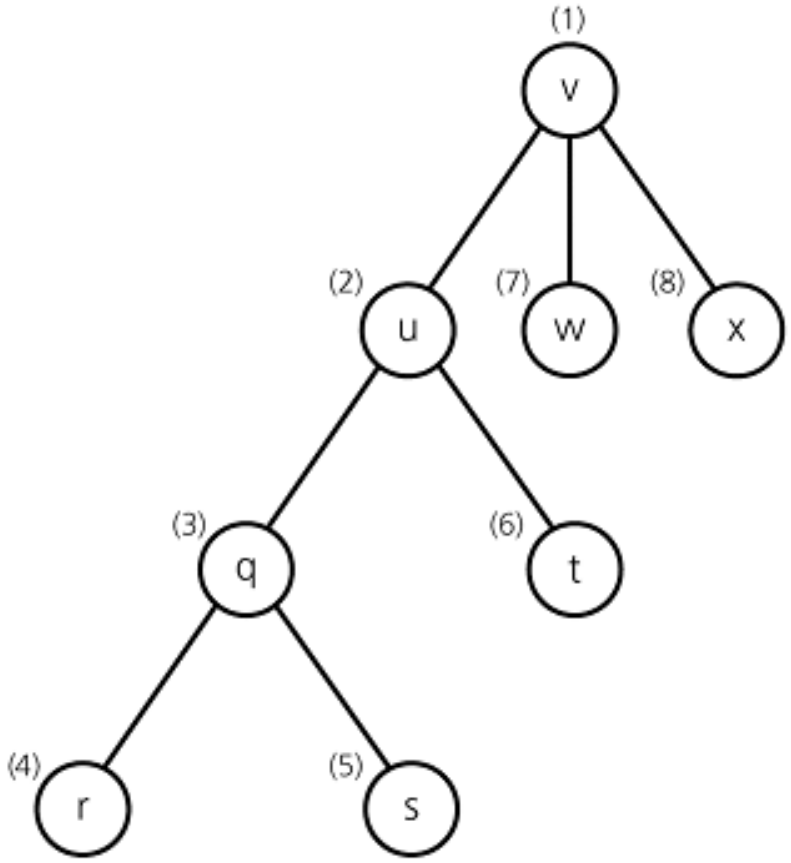
Depth-First Traversal

- For a given vertex v, **depth-first traversal** algorithm proceeds along a path from v as deeply into the graph as possible before backing up.

- That is, after visiting a vertex v, the algorithm visits (if possible) an unvisited adjacent vertex to vertex v.

- The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - We may visit the vertices adjacent to v in sorted order.

Depth-First Traversal – Example



- A depth-first traversal of the graph starting from vertex v.

- Visit a vertex, then visit a vertex adjacent to that vertex.

- If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.

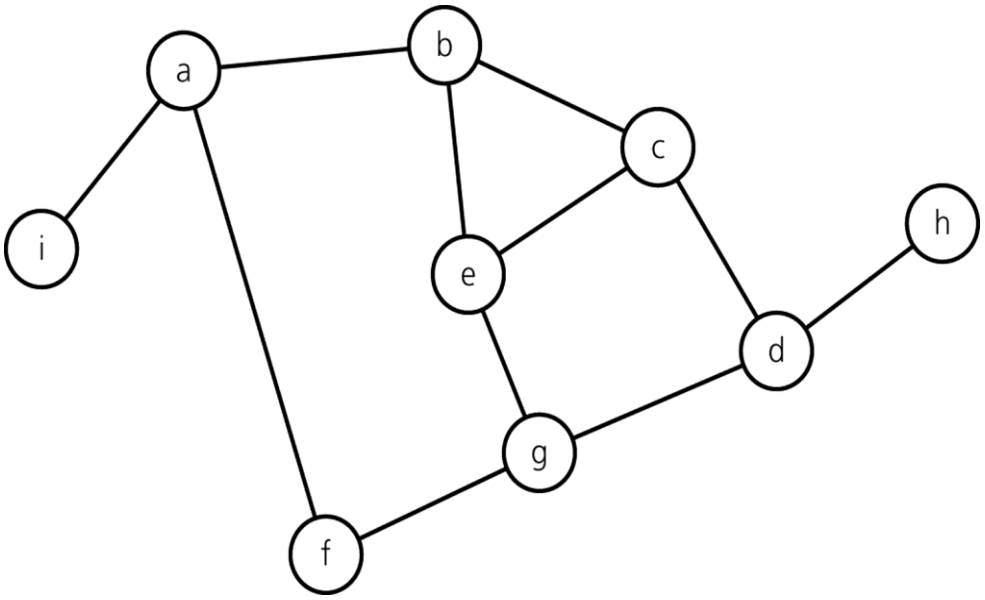
Recursive Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy
// Recursive Version
  Mark v as visited;
  for (each unvisited vertex u adjacent to v)
        dft(u)
}
```

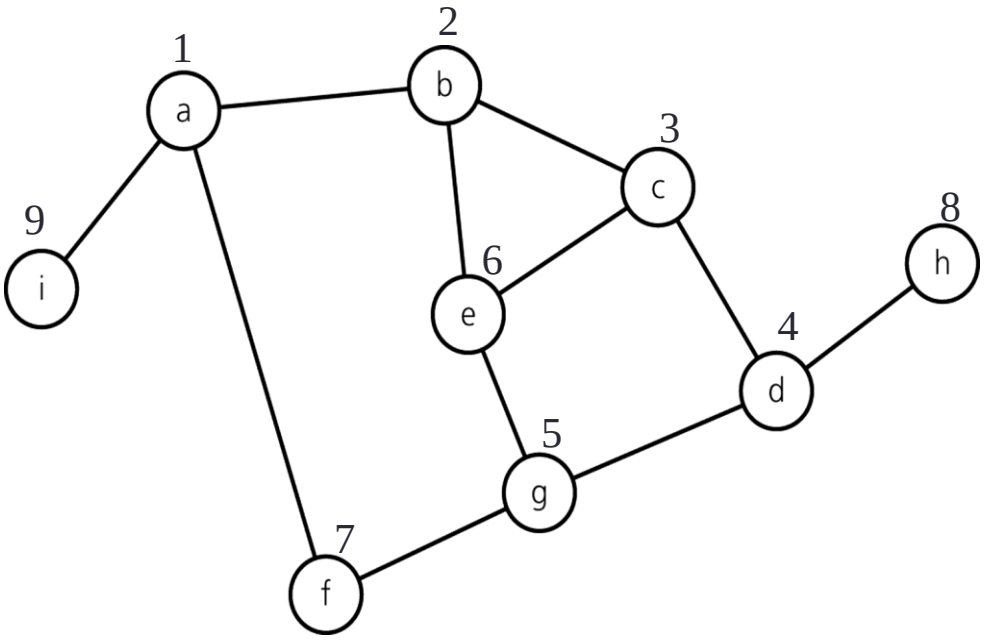
Iterative Depth-First Traversal Algorithm

```
dft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using depth-first strategy: Iterative Version
  s.createStack();
  // push v into the stack and mark it
  s.push(v);
  Mark v as visited;
  while (!s.isEmpty()) {
     if (no unvisited vertices are adjacent to the vertex on
          the top of stack)
        s.pop();  // backtrack
     else {
        Select an unvisited vertex u adjacent to the vertex
           on the top of the stack;
        s.push(u);
        Mark u as visited;
     }
  }
}
```

Trace of Iterative DFT – starting from vertex a



Trace of Iterative DFT – starting from vertex a



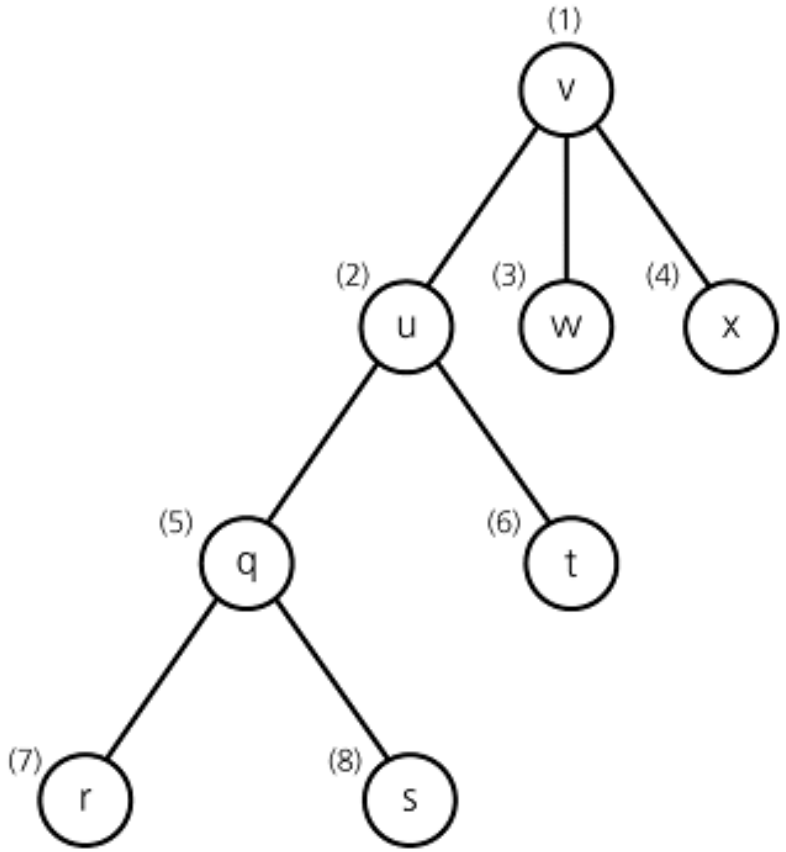
Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

Breadth-First Traversal

- After visiting a given vertex v, the **breadth-first traversal** algorithm visits every vertex adjacent to v that it can before visiting any other vertex.

- The breadth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  - We may visit the vertices adjacent to v in sorted order.

Breadth-First Traversal – Example



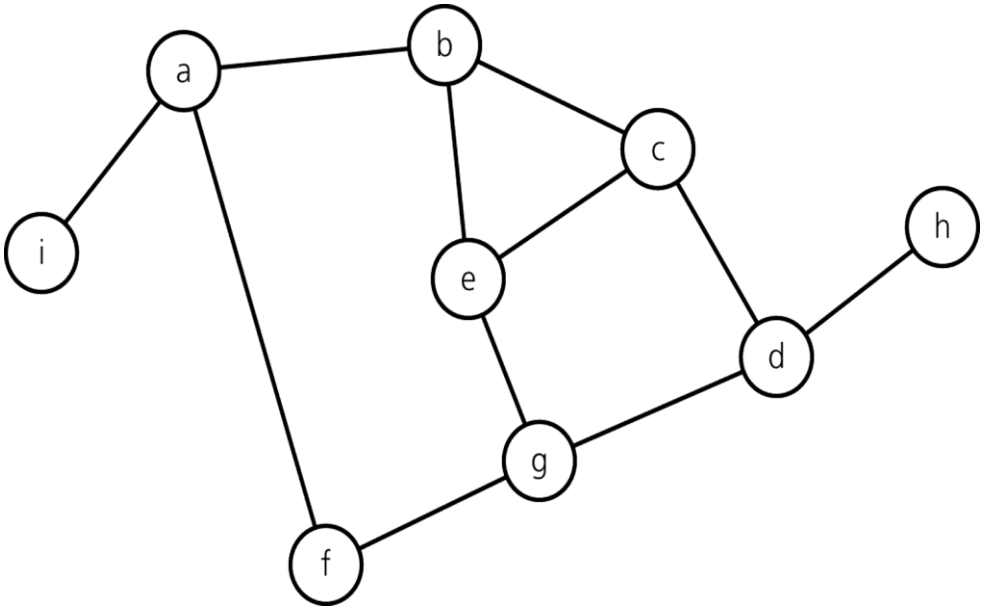
- A breadth-first traversal of the graph starting from vertex v.

- Visit a vertex, then visit all vertices adjacent to that vertex.

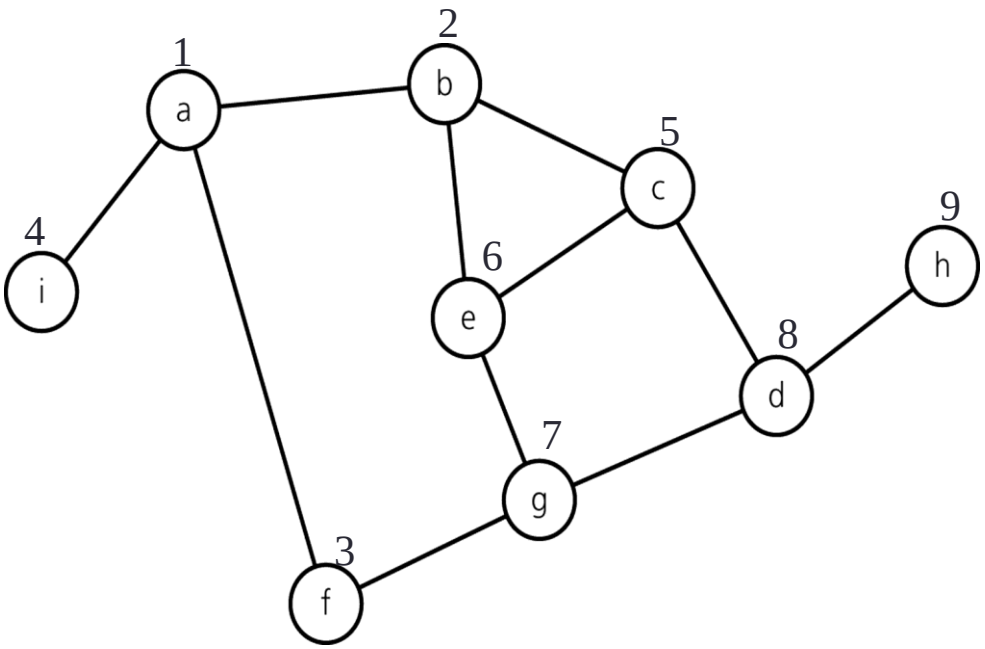
Iterative Breadth-First Traversal Algorithm

```
bft(in v:Vertex) {
// Traverses a graph beginning at vertex v
// by using breadth-first strategy: Iterative Version
  q.createQueue();
  // add v to the queue and mark it
  Mark v as visited;
   q.enqueue(v);
  while (!q.isEmpty()) {
     q.dequeue(w);
     for (each unvisited vertex u adjacent to w) {
        Mark u as visited;
        q.enqueue(u);
     }
  }
}
```

Trace of Iterative BFT – starting from vertex a



Trace of Iterative BFT – starting from vertex a



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)