CS 3100
Data Structures and Algorithms 2
Lecture 3: Graphs, Breadth First Search

Co-instructors: Robbie Hott and Tom Horton

Fall 2023

Readings in CLRS 4th edition:

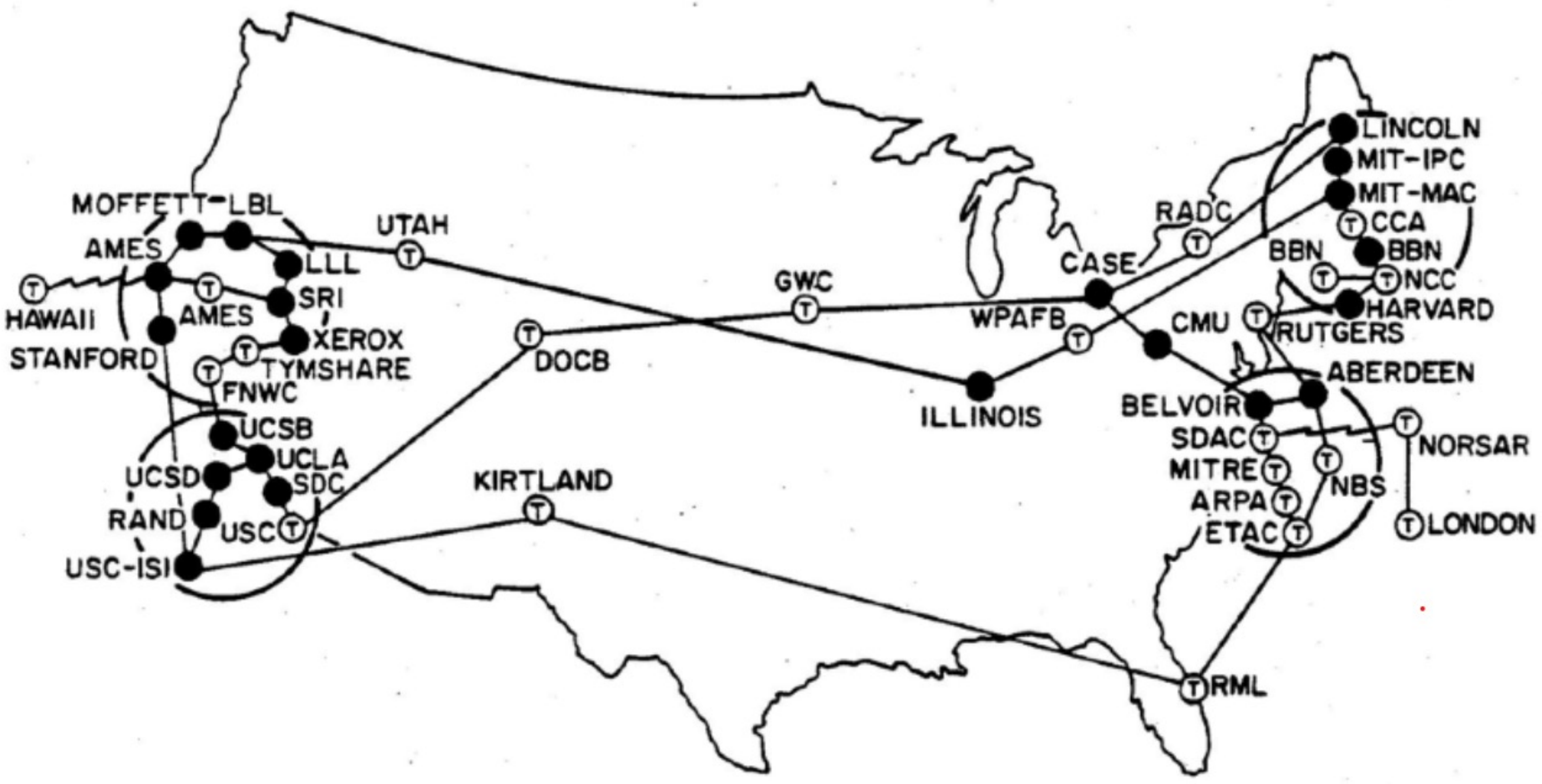
- Chapter 20, through Section 2

Announcements

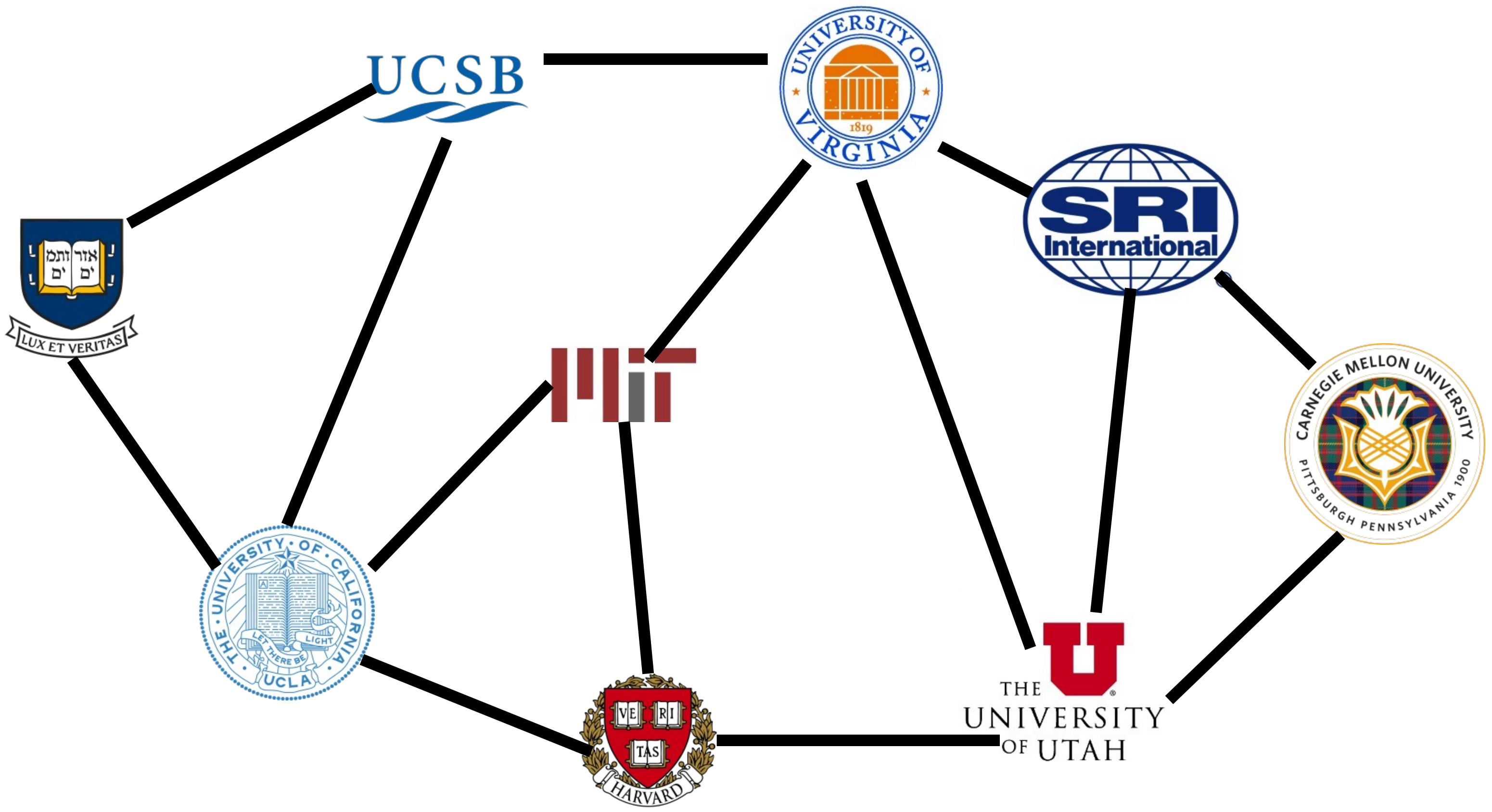
- PS1 and PA1 available this week
- In-Class Activity 8/24
  - Solution video posted this week
  - Will not count towards overall IC grade
- Discord server is available, please join!
- Office Hours
  - Prof Hott: 3-5pm Monday, 4-5pm Thursday
  - TA office hours posted soon, check our website

Computer History Trivia:
What is the ARPANET?

ARPANET c.1970



ARPANET

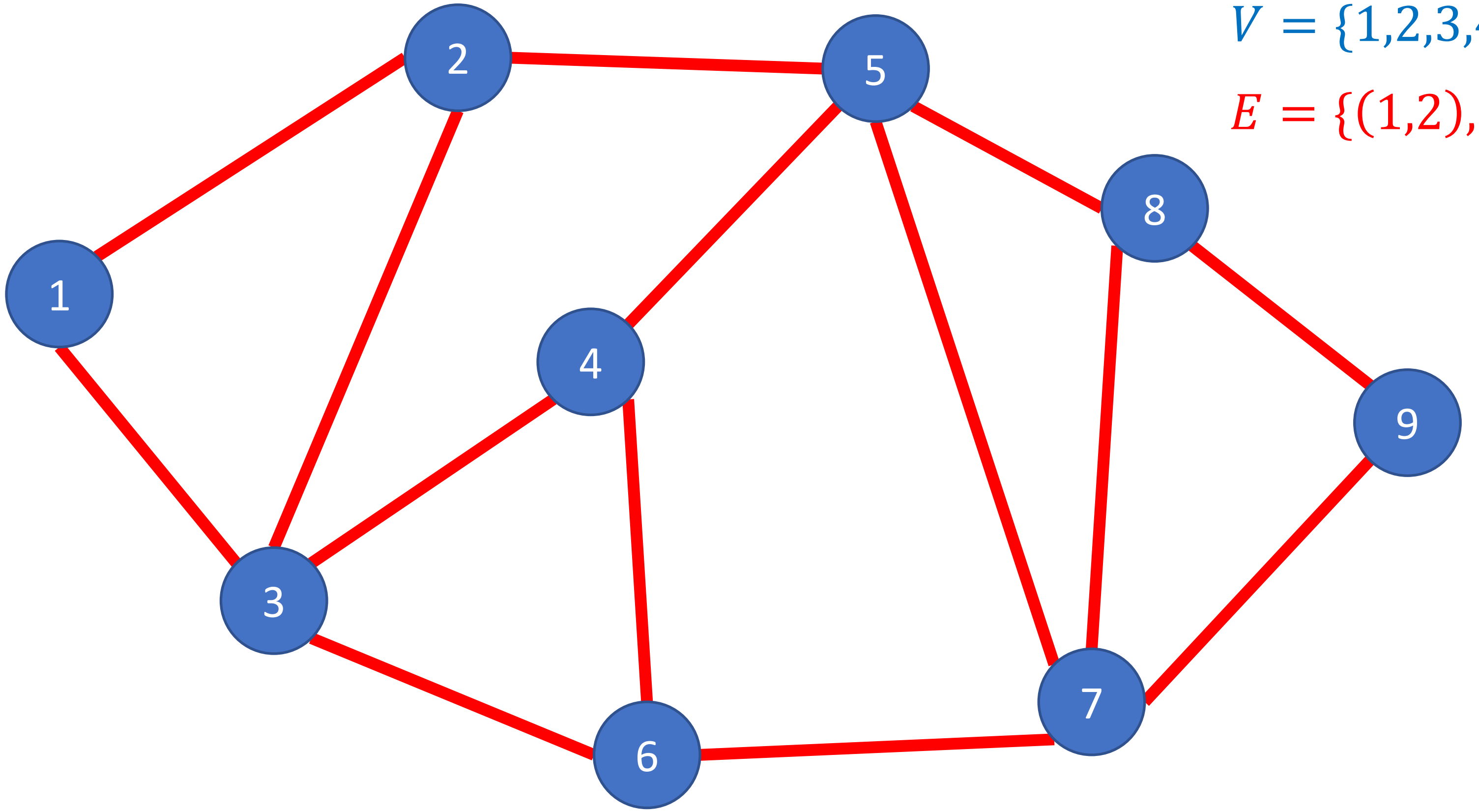

UCSB
UNIVERSITY OF VIRGINIA
1819
SRI International
LUX ET VERITAS
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PENNSYLVANIA 1900
THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA
VE RI TAS
HARVARD
THE UNIVERSITY OF UTAH

Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges



$V = \{1,2,3,4,5,6,7,8,9\}$

$E = \{(1,2), (2,3), (1,3), \dots\}$

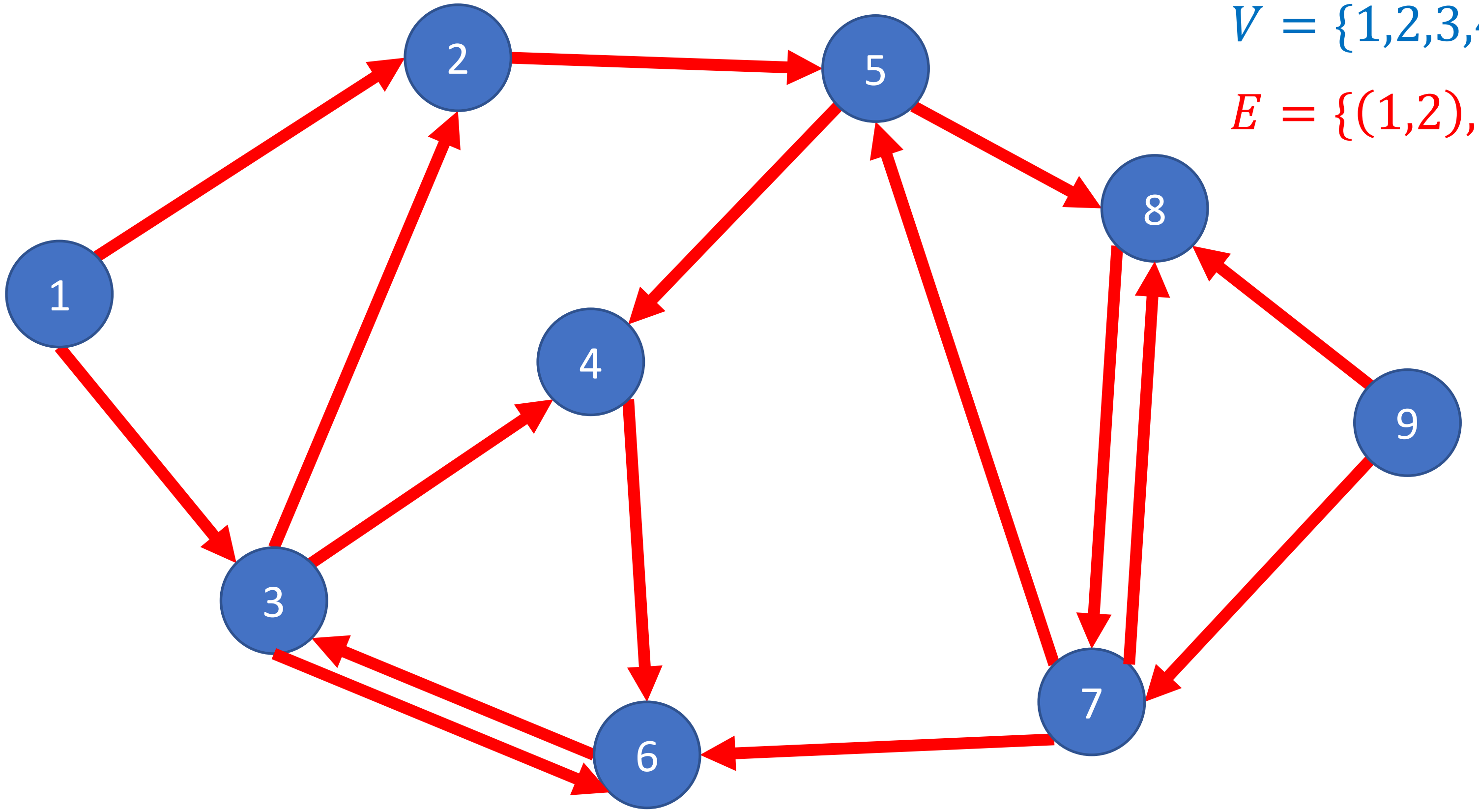
Can an edge connect a node to itself?
Not in an undirected graph, but OK in a "multigraph"

Directed Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges



$V = \{1,2,3,4,5,6,7,8,9\}$

$E = \{(1,2), (3,2), (1,3), \dots\}$

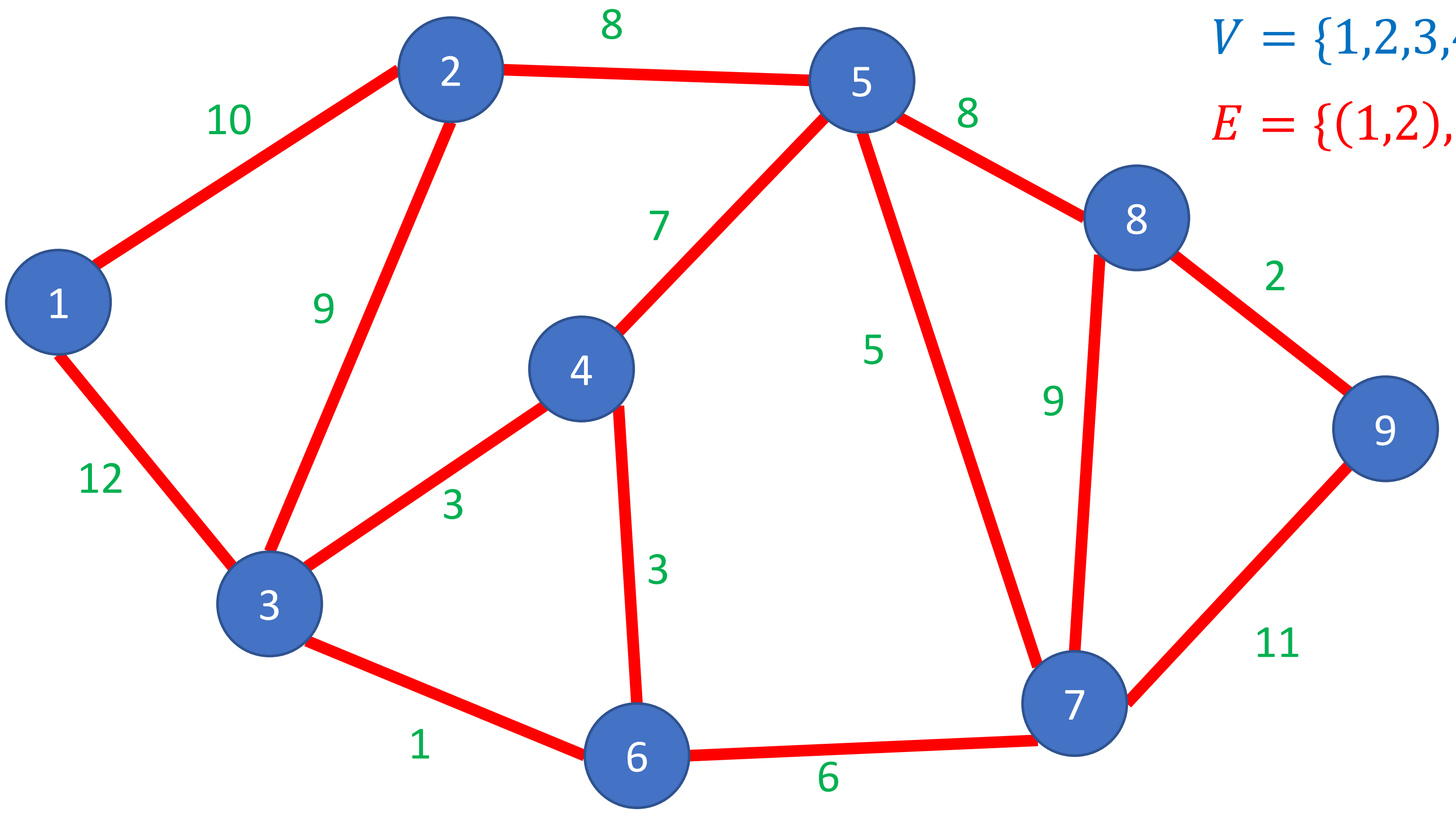
Weighted Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

$w(e)$ = weight of edge e



$V = \{1,2,3,4,5,6,7,8,9\}$

$E = \{(1,2), (2,3), (1,3), \dots\}$

Some Graph Terms

Degree

- Number of “neighbors” of a vertex

Indegree

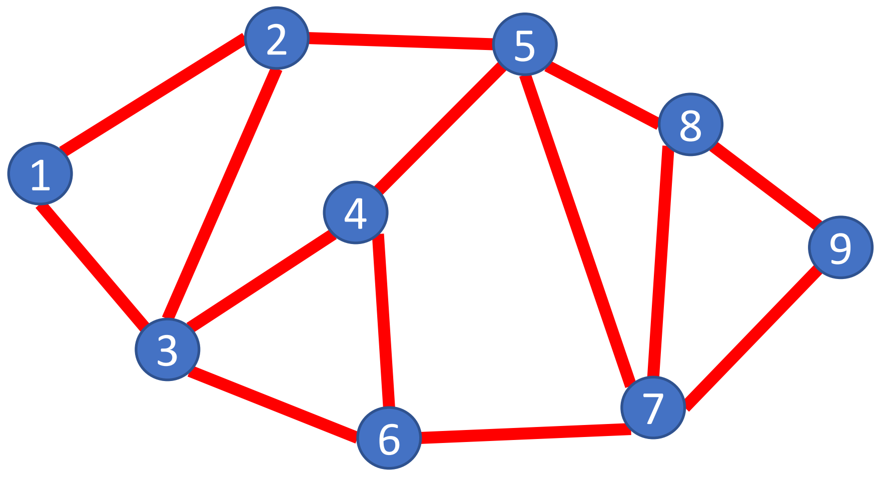
- Number of incoming edges

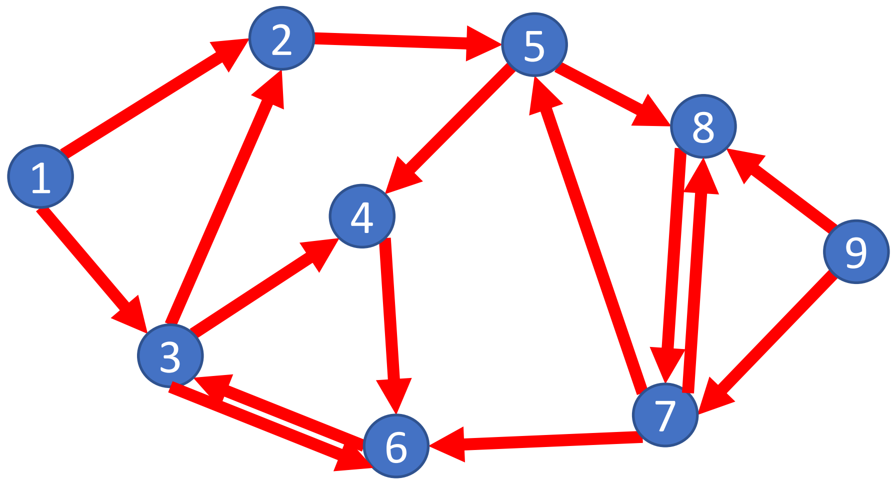
Outdegree

- Number of outgoing edges

Relative number of edges to nodes

- What’s the max number of edges for an undirected graph? Directed graph?
- Complete graph
- Sparse graph vs. dense graph



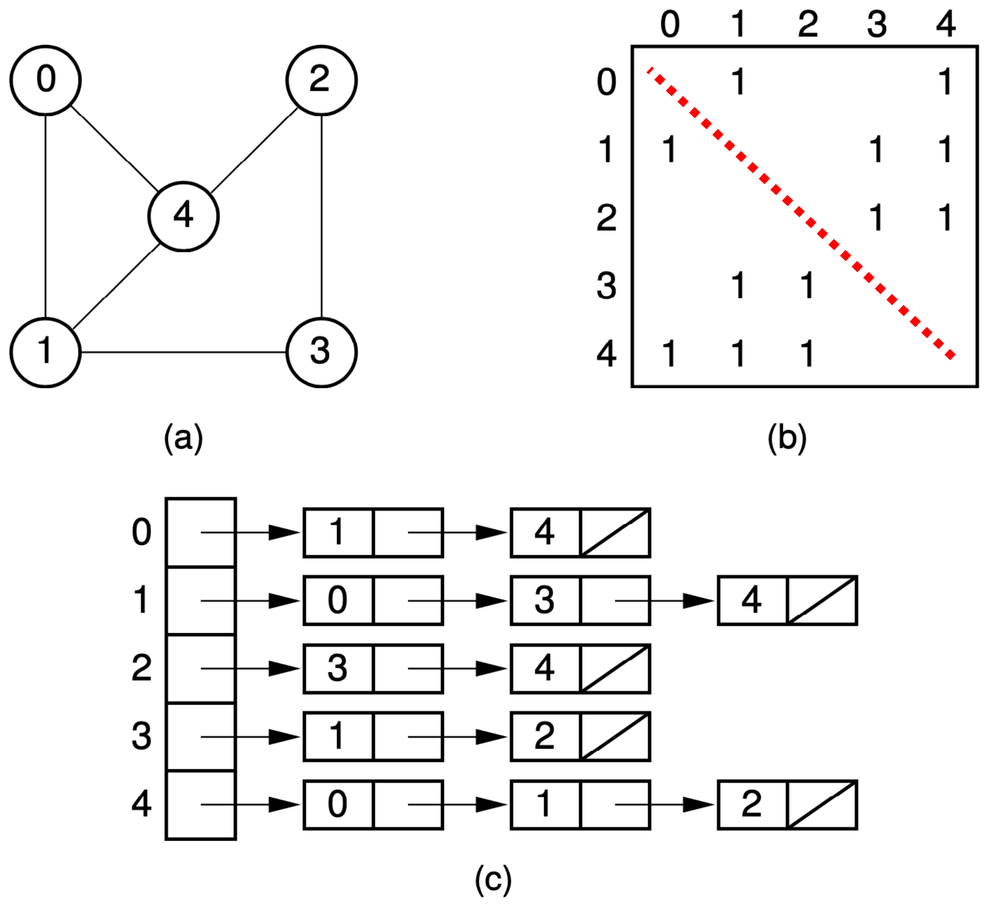


ADT Graph Operations

To represent a Graph (i.e. build a data structure) we need:

- Add Edge
- Remove Edge
- Check if Edge Exists
- Get Neighbors (incoming)
- Get Neighbors (outgoing)

Data Structures for Undirected Graphs



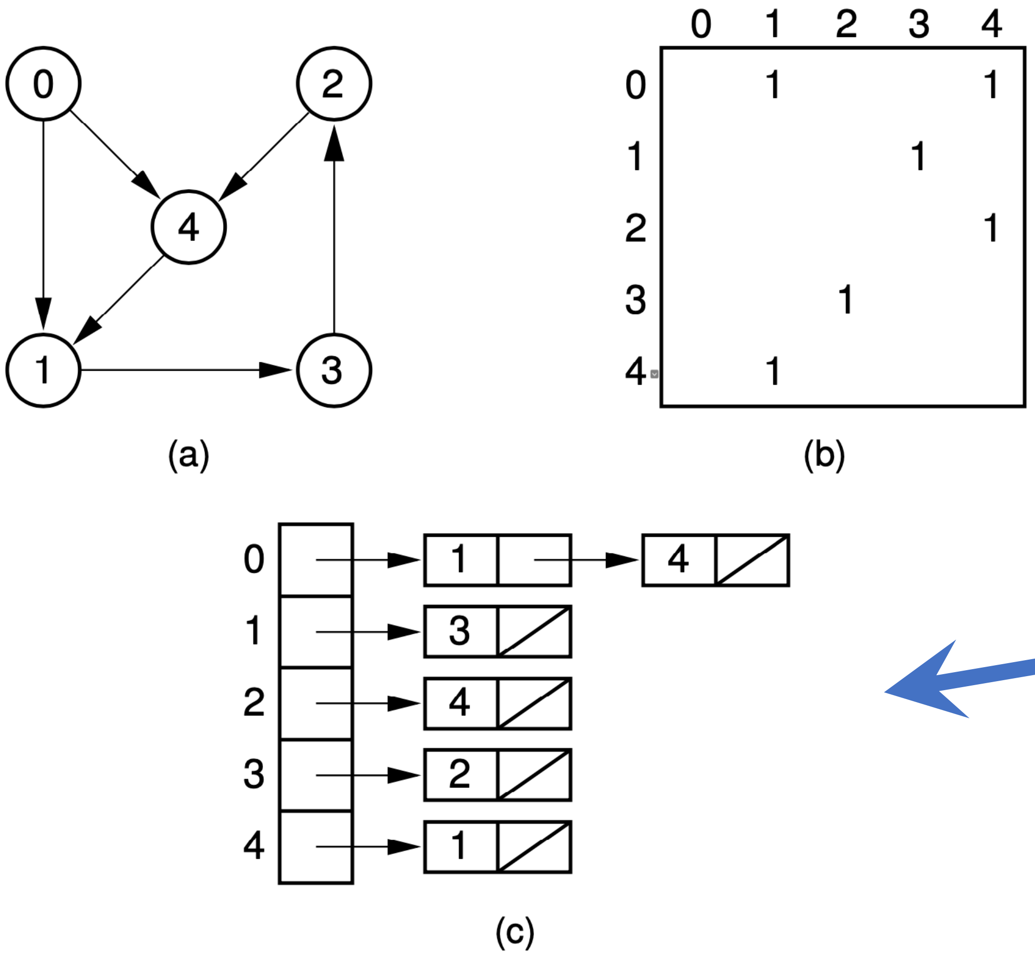
Adjacency Matrix:
A[u][v] is 1 if edge *(u,v)* exists.
Note symmetrical around diagonal. Could just store info in one half of matrix.

Adjacency List:
Note each edge *(u,v)* has an edge-node on *u*'s list and also *v*'s list.

Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Image of diagrams from
https://people.cs.vt.edu/~shaffer/Book/

Data Structures for Digraphs



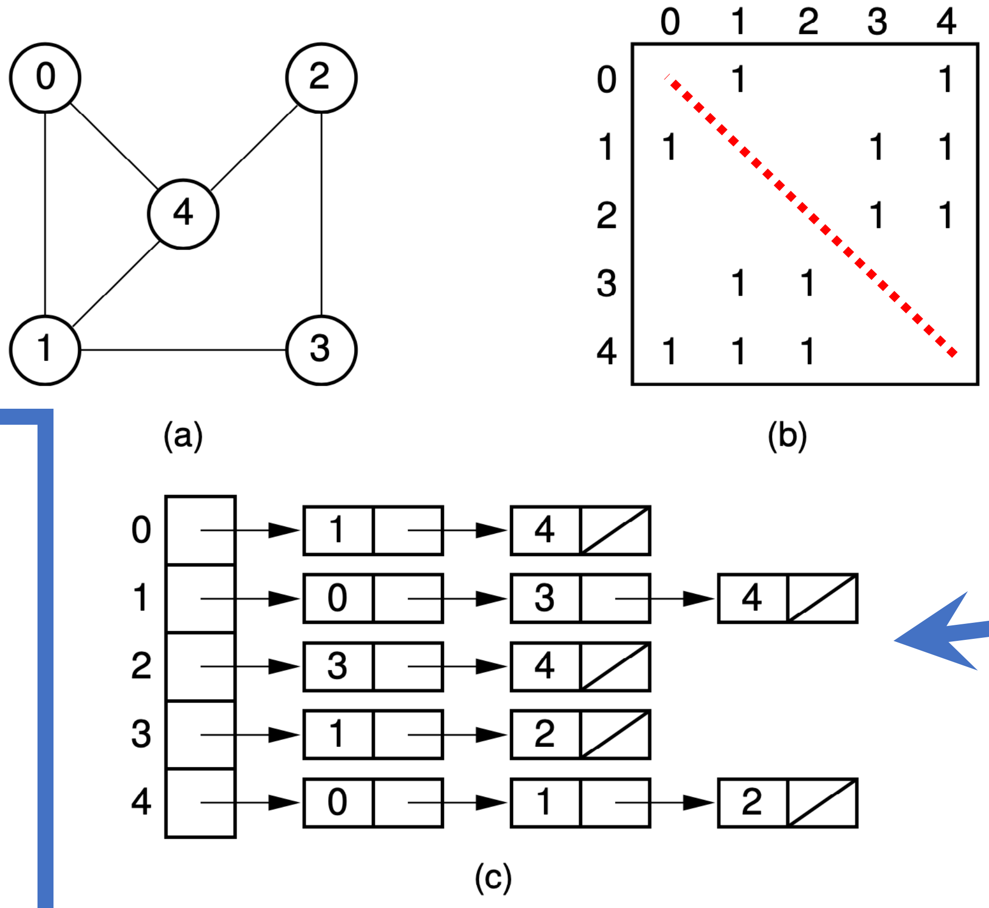
Adjacency Matrix:
Not symmetrical around diagonal for digraph.

Adjacency List:
Note each directed edge *(u,v)* has an edge-node on just one vertex's list.

Figure 11.3 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Image of diagrams from https://people.cs.vt.edu/~shaffer/Book/

Data Structures for Weighted Graphs



Adjacency Matrix:
Store weight (u,v) in matrix cell. Use 0 or negative value if edge not in graph.

Images are of **unweighted** graphs.

How would we store weights?

Adjacency List:
Add a field to the the edge node object to store the weight.

Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

Image of diagrams from https://people.cs.vt.edu/~shaffer/Book/

Operation Costs: Adjacency Matrix

Adjacency Matrix:

1. Space to represent: $\Theta(?)$
2. Add Edge: $\Theta(?)$
3. Remove Edge: $\Theta(?)$
4. Check if Edge Exists: $\Theta(?)$
5. Get Neighbors (incoming): $\Theta(?)$
6. Get Neighbors (outgoing): $\Theta(?)$

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

$$\boxed{\begin{aligned}|V| &= n\\ |E| &= m\end{aligned}}$$

Operation Costs: Adjacency Matrix

Adjacency Matrix:

1. Space to represent: $\Theta(n^2)$
2. Add Edge: $\Theta(1)$
3. Remove Edge: $\Theta(1)$
4. Check if Edge Exists: $\Theta(1)$
5. Get Neighbors (incoming): $\Theta(n)$
6. Get Neighbors (outgoing): $\Theta(n)$

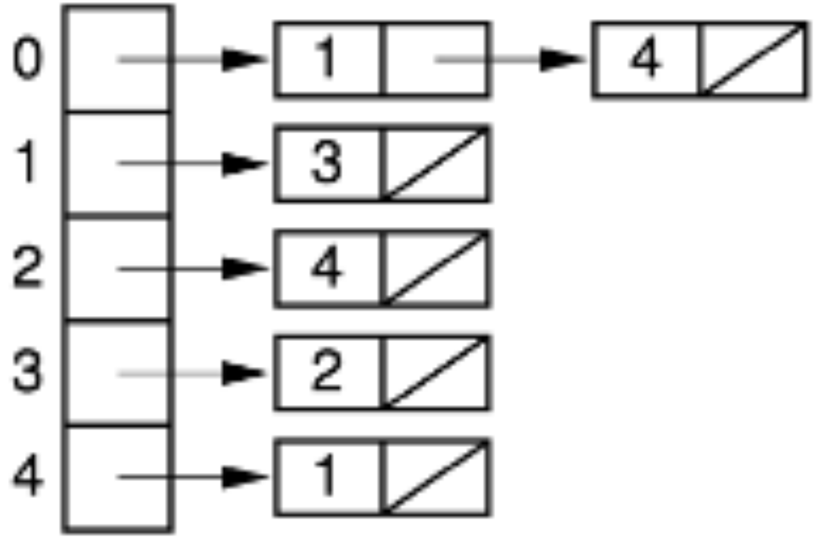
	0	1	2	3	4
0		1			1
1				1	
2					1
3			1		
4		1			

$$\boxed{\begin{array}{l}|V| = n \\ |E| = m\end{array}}$$

Operation Costs: Adjacency List

Adjacency List:

1. Space to represent: $\Theta(?)$
2. Add Edge: $\Theta(1?)$
3. Remove Edge: $\Theta(?)$
4. Check if Edge Exists: $\Theta(?)$
5. Get Neighbors (incoming): $\Theta(?)$
6. Get Neighbors (outgoing): $\Theta(?)$

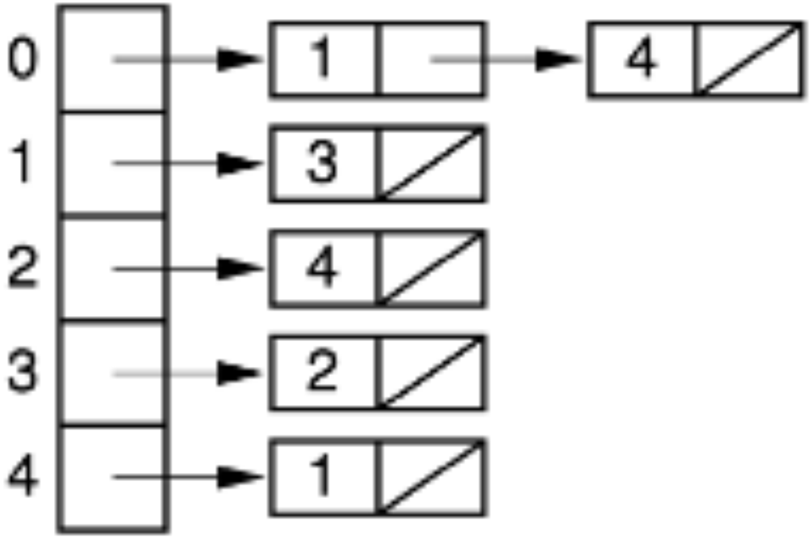


$|V| = n$
$|E| = m$

Operation Costs: Adjacency List

Adjacency List:

1. Space to represent: $\Theta(n+m)$
2. Add Edge: $\Theta(1)$
3. Remove Edge: $\Theta(n)$
4. Check if Edge Exists: $\Theta(n)$
5. Get Neighbors (incoming): $\Theta(n+m)$
6. Get Neighbors (outgoing): $\Theta(\deg(v))$



$$\boxed{\begin{aligned}|V| &= n\\ |E| &= m\end{aligned}}$$

Cost Comparison: Adjacency List vs Matrix

Adjacency List:

1. Space to represent: $\Theta(n+m)$
2. Add Edge: $\Theta(1)$
3. Remove Edge: $\Theta(n)$
4. Check if Edge Exists: $\Theta(n)$
5. Get Neighbors (incoming): $\Theta(n+m)$
6. Get Neighbors (outgoing): $\Theta(\deg(v))$

Adjacency Matrix:

1. Space to represent: $\Theta(n^2)$
2. Add Edge: $\Theta(1)$
3. Remove Edge: $\Theta(1)$
4. Check if Edge Exists: $\Theta(1)$
5. Get Neighbors (incoming): $\Theta(n)$
6. Get Neighbors (outgoing): $\Theta(n)$

$$\boxed{\begin{array}{l}|V| = n \\ |E| = m\end{array}}$$

Identifying Vertices as Strings

Vertices may be identified with strings not integers.

(1) Could use an **adjacency map** instead of an adjacency list, and also store strings in edge-nodes

(2) Programmers often have an index and/or lookup table to convert between int's and string IDs for vertices. Understand this example?

There are other ways to do this. Use your programming skills!

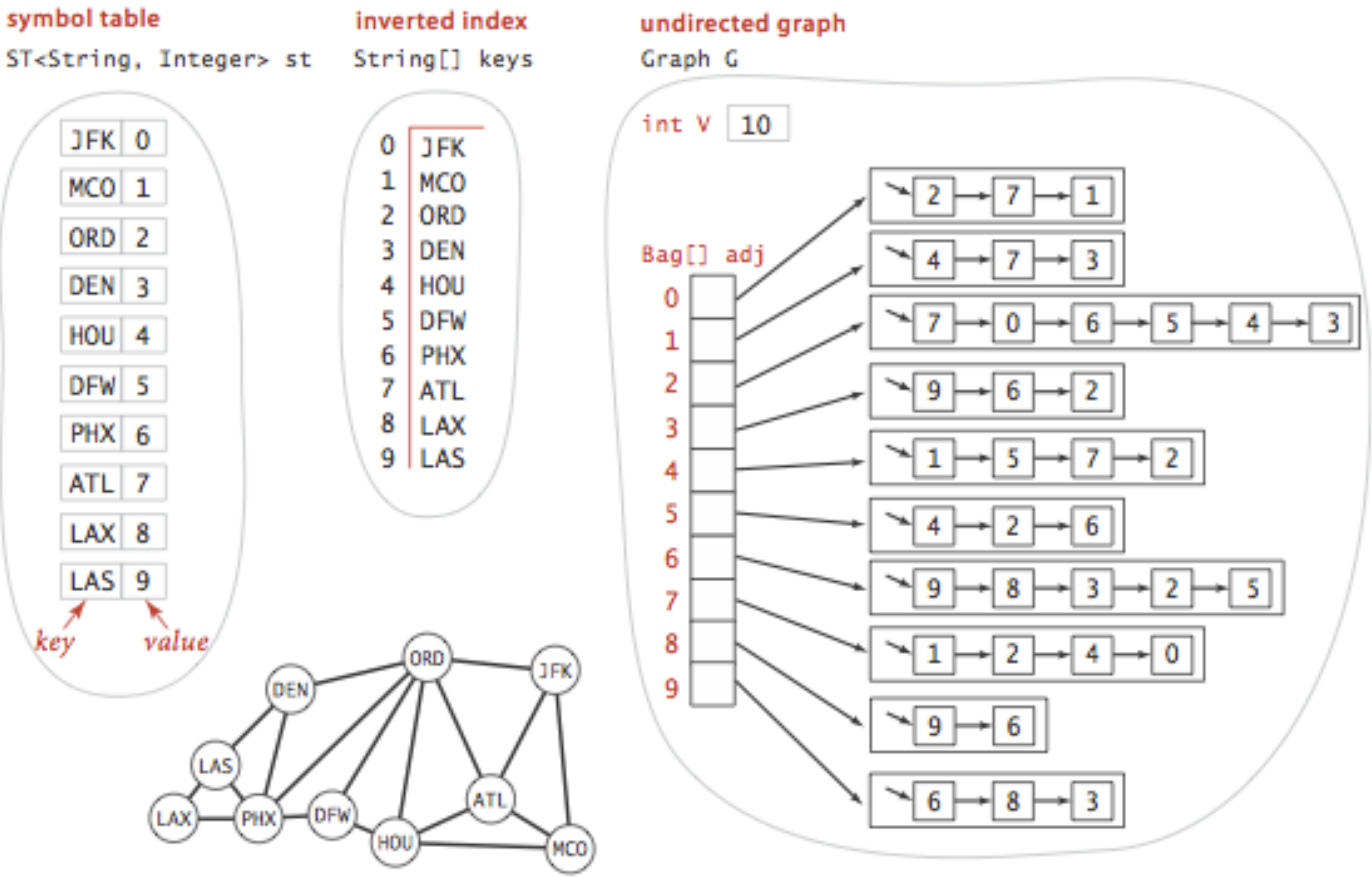
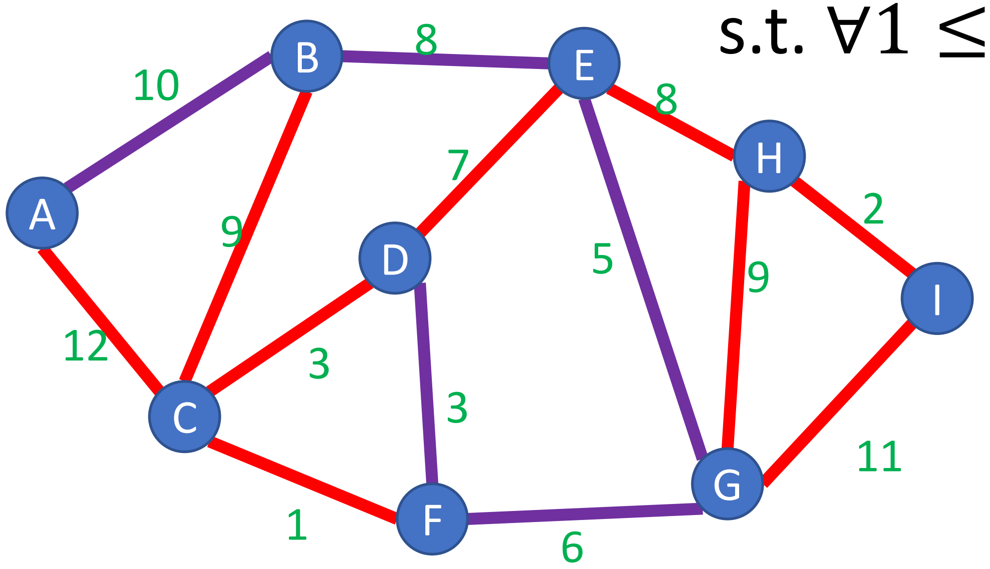


Image from https://algs4.cs.princeton.edu/home/

Definition: Path



A sequence of nodes $(v_1, v_2, \dots, v_k)$
s.t. $\forall 1 \le i \le k-1$, $(v_i, v_{i+1}) \in E$

<u>Acyclic graph:</u> has no cycles
<u>Directed Acyclic Graph (DAG):</u>
directed graph, no cycles

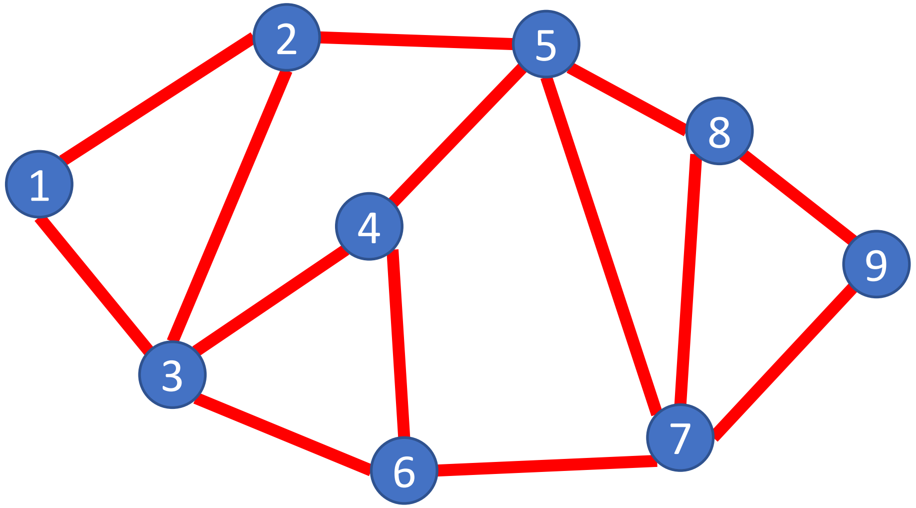
<u>Simple Path:</u>
A path in which each node appears at most once

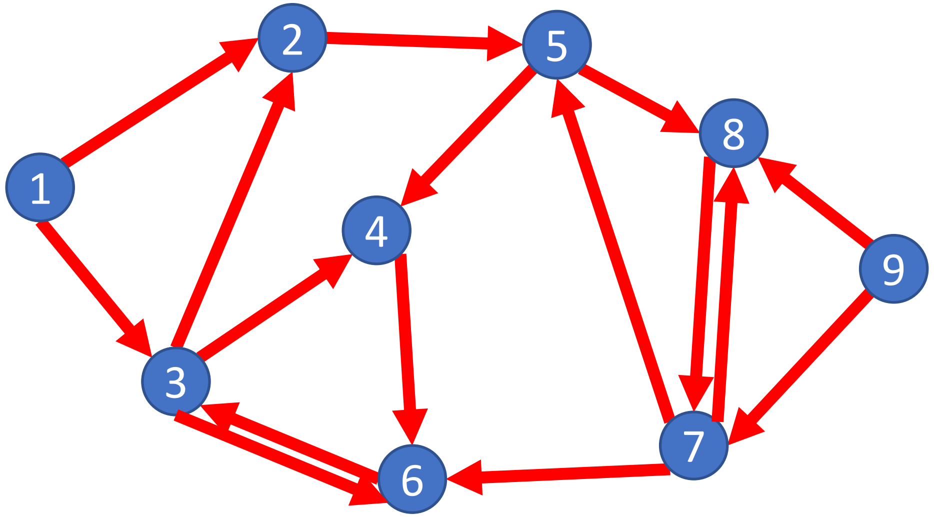
<u>Cycle:</u>
A path of > 2 nodes in which $v_1 = v_k$

Definition: Connected Graph

For a directed graph, the name for this property is **strongly connected**.

A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from v_1 to v_2





An undirected graph can have more than one connected component.

Breadth First Search

Traversing Graphs

"Traversing" means processing each vertex edge in some organized fashion by following edges between vertices

- We speak of *visiting* a vertex. Might do something while there.

Recall traversal of binary trees:

- Several strategies: In-order, pre-order, post-order
- Traversal strategy implies an order of visits
- We used recursion to describe and implement these

Graphs can be used to model interesting, complex relationships

- Often traversal used just to process the set of vertices or edges
- Sometimes traversal can identify interesting properties of the graph
- Sometimes traversal (perhaps modified, enhanced) can answer interesting questions about the problem-instance that the graph models

BFS: Specific Input/Output

Input:

- A graph ***G***
- single start vertex ***s***

Output:

- Distance from ***s*** to each node in ***G*** (distance = number of edges)
- Breadth-First Tree of ***G*** with root ***s***

Strategy:

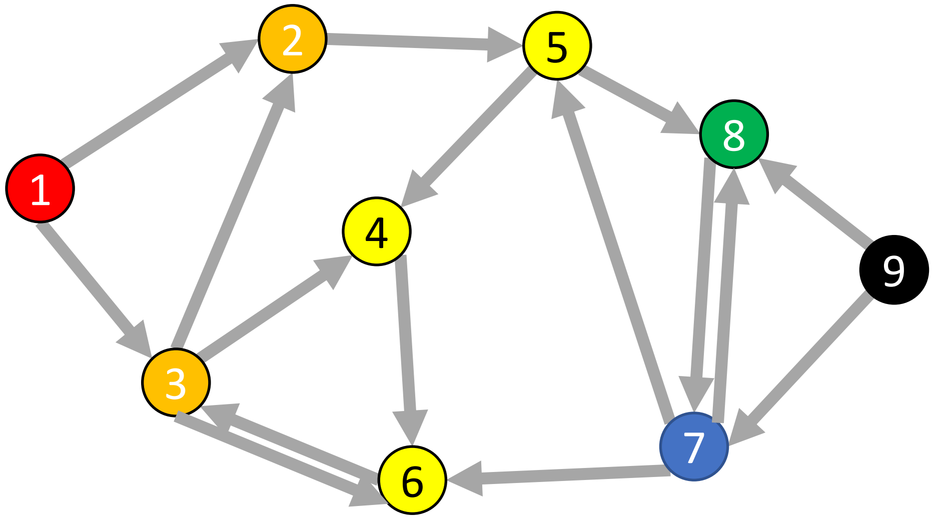
Start with node ***s***, visit all neighbors of ***s***, then all neighbors of neighbors of ***s***, …

Important: The paths in this BFS tree represent the **shortest paths** from s to each node in G

- But edge weight's (if any) not used, so "short" is in terms of number of edges in path

BFS

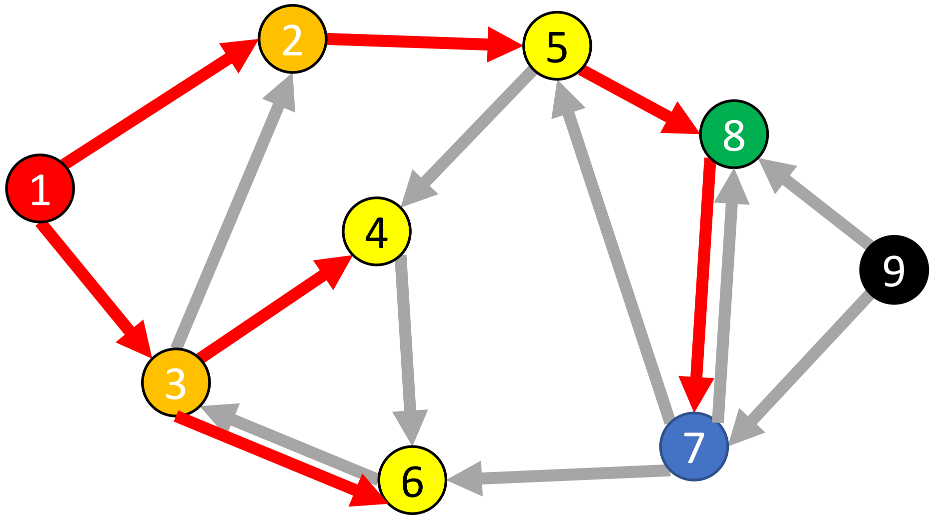
```
def bfs(graph, s):
        toVisit.enqueue(s)
        mark s as "seen"
        While toVisit is not empty:
                current = toVisit.dequeue()
                for v in neighbors(current):
                        if v not seen:
                                mark v as seen
                                toVisit.enqueue(v)
```



BFS: Shortest Path

```
def bfs(graph, s, t):
        layer = 0
        toVisit.enqueue(s)
        depth[s] = layer
        While toVisit is not empty:
                current = toVisit.dequeue()
                layer = depth [current]
                for v in neighbors(current):
                        if v does not have a depth:
                                depth[v]=layer+1
                                toVisit.enqueue(v)
        return depth[t]
```

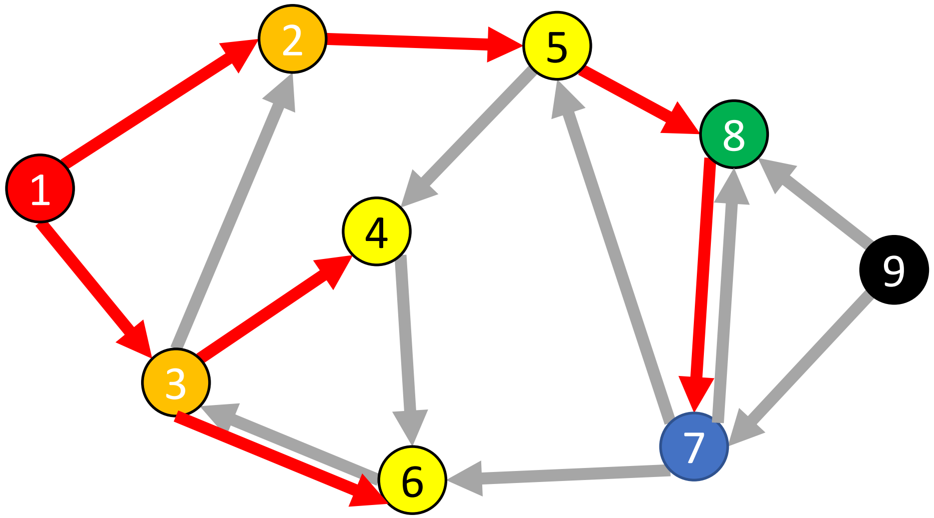
Idea: when it's seen, remember its "layer" depth!



BFS: Shortest Path

```
def shortest_path(graph, s, t):
    layer = 0
    depth = [-1,-1,-1,…]  # Length matches |V|
    toVisit.enqueue(s)
    mark a as “seen”
    depth[s] = 0
    While toVisit is not empty:
        current = toVisit.dequeue()
        layer = depth[current]
        if current == t:
            return layer
        for v in neighbors(current):
            if v not seen:
                mark v as seen
                toVisit.enqueue(v)
                depth[v] = layer + 1
```

Idea: when it’s seen, remember its “layer” depth!



Breadth-first search from CLRS 20.2

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

From CLRS

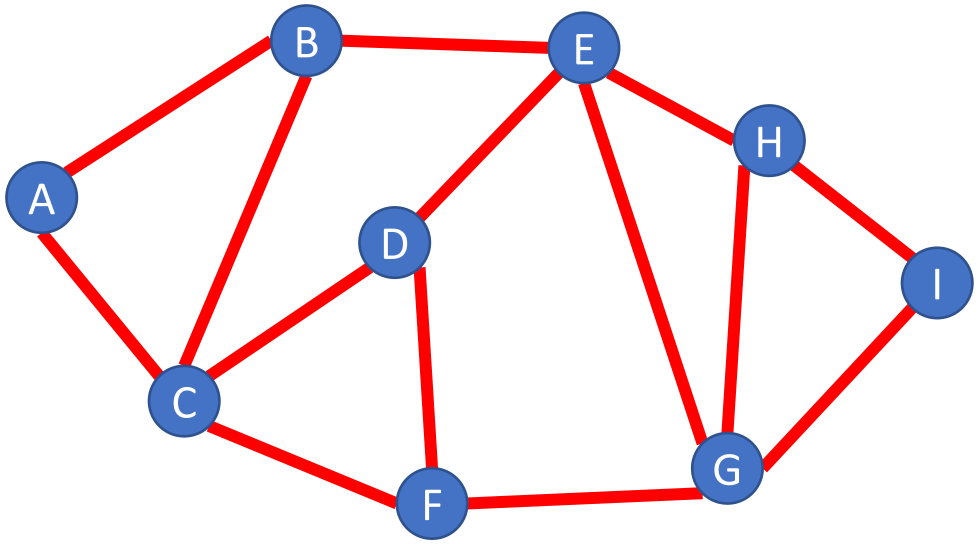
Vertices here have some properties:

- *color = white/gray/black*
- *d = distance from start node*
- *pi = parent in tree, i.e. v.pi is vertex by which v was* connected to BFS tree

Color meanings here:

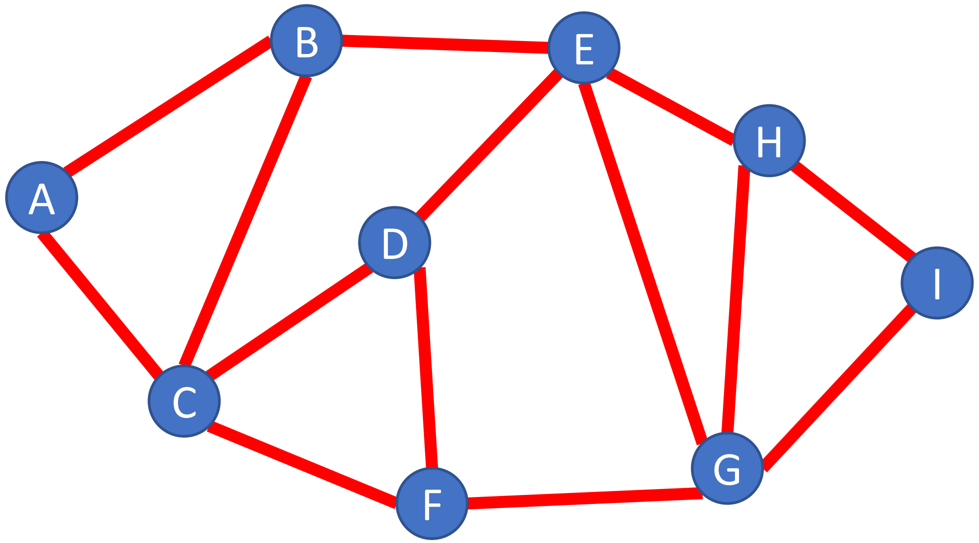
- White: haven’t seen this vertex yet
- Gray: vertex has been seen and added to the queue for processing later
- Black: vertex has been removed from queue and its neighbors seen and added to the queue

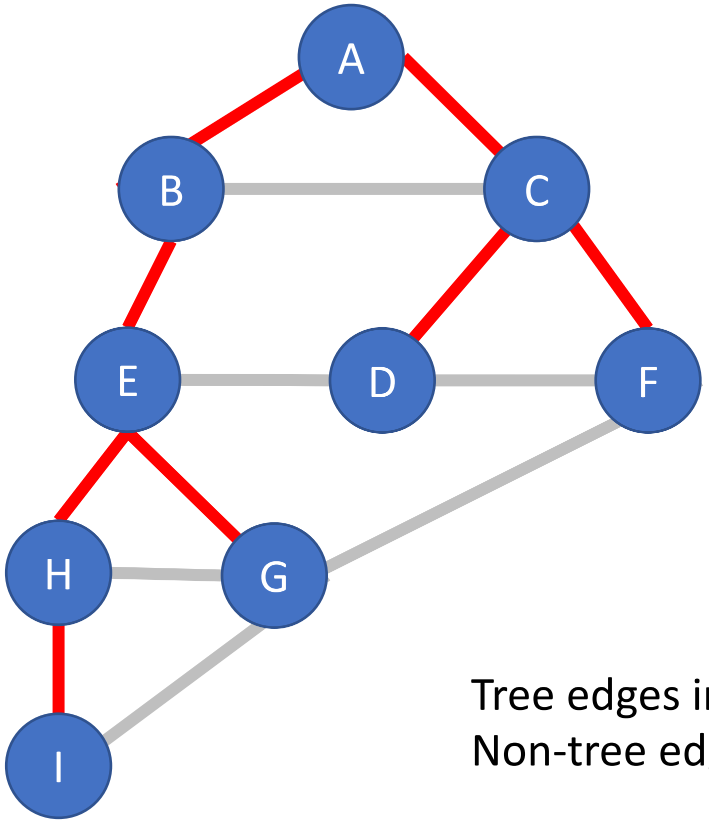
Tree View of BFS Search Results



Draw BFS tree starting at A

Tree View of BFS Search Results





Tree edges in red
Non-tree edges in gray

Analysis for Breadth-first search

For a graph having V vertices and E edges

- Each edge is processed once in the while loop for a cost of $\theta(E)$
- Each vertex is put into the queue once and removed from the queue and processed once, for a cost $\theta(V)$
  - Also, cost of initializing colors or depth arrays is $\theta(V)$

Total **time-complexity**: $\theta(V+E)$

- For graph algorithms this is called “linear”

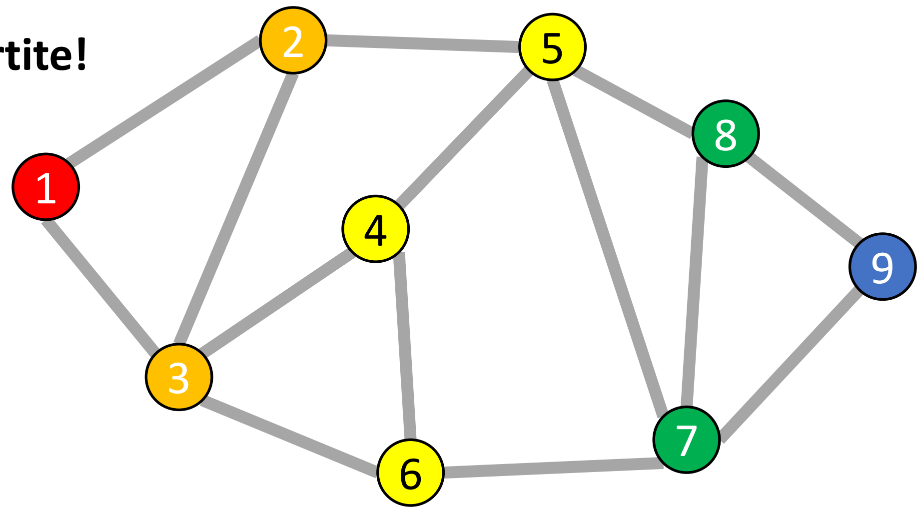
Space complexity: extra space is used for queue and also depth/color arrays, so $\theta(V)$

Definition: Bipartite

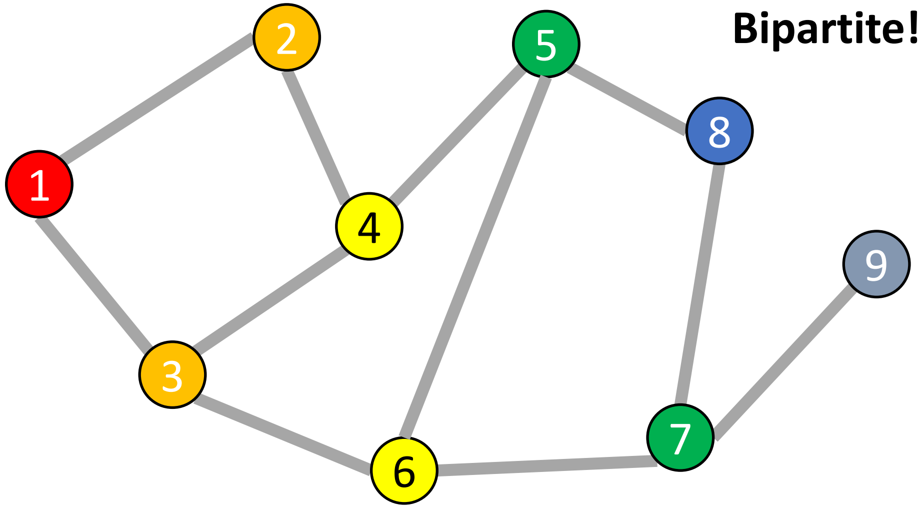
A (undirected) graph is Bipartite provided every vertex can be assigned to one of two teams such that every edge “crosses” teams

- Alternative: Every vertex can be given one of two colors such that no edges connect same-color nodes

Not Bipartite!



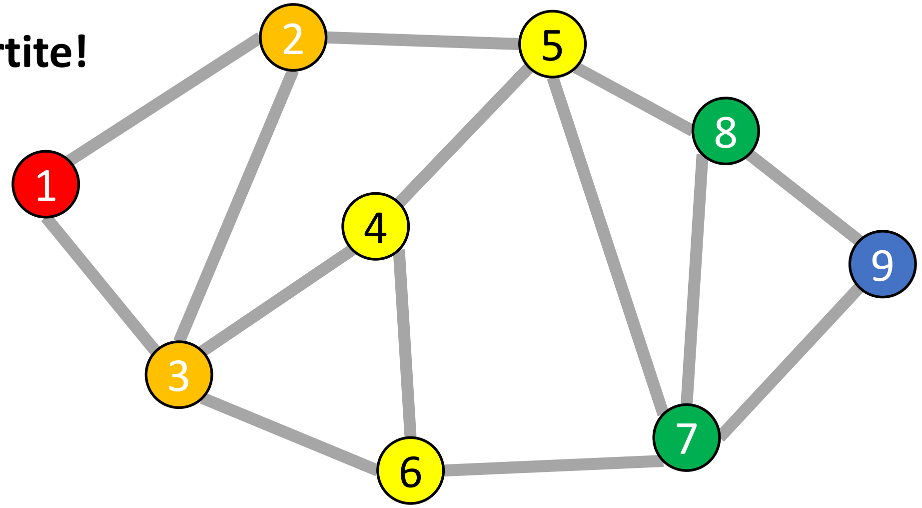
Bipartite!



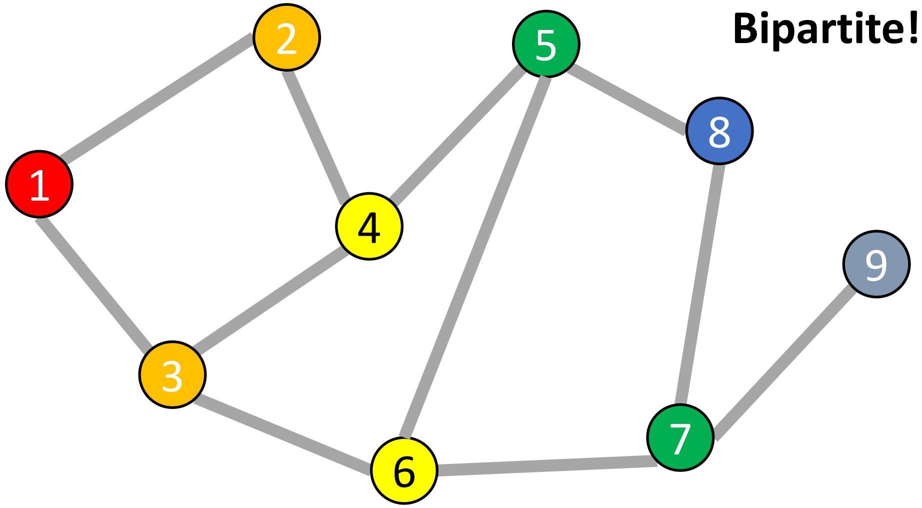
Odd Length Cycles

A graph is bipartite if and only if it has no odd length cycles

Not Bipartite!



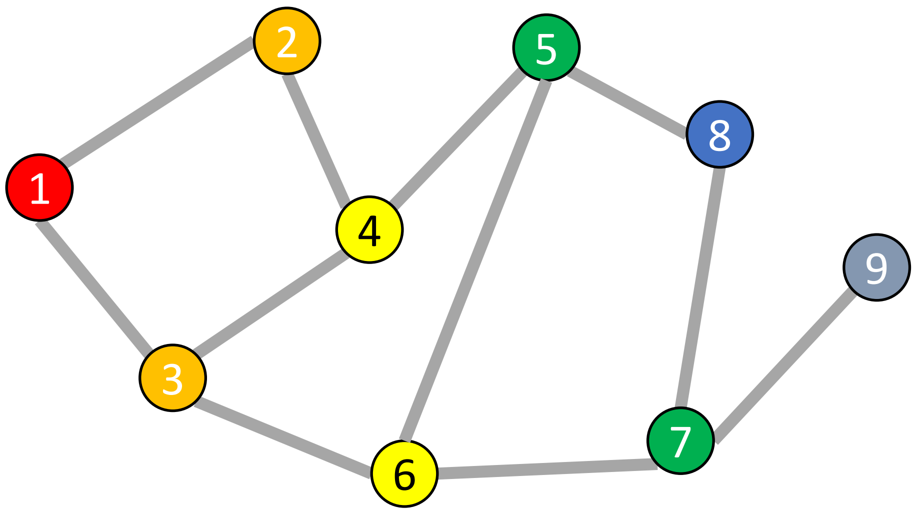
Bipartite!



BFS: Bipartite Graph?

Idea: Check for edges in the same layer!

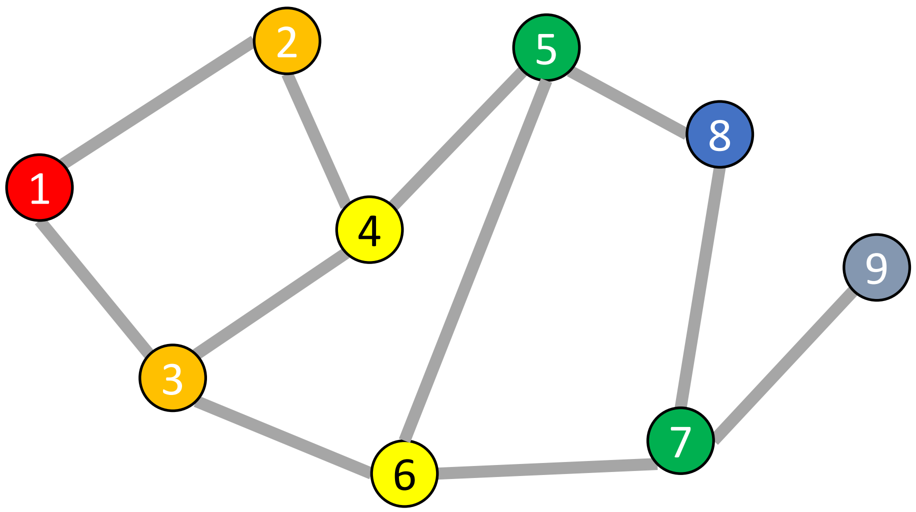
```
def isBipartite(graph, s):
        toVisit.enqueue(s)
        mark s as "seen"
        While toVisit is not empty:
                current = toVisit.dequeue()
                for v in neighbors(current):
                        if v not seen:
                                mark v as seen
                                toVisit.enqueue(v)
```



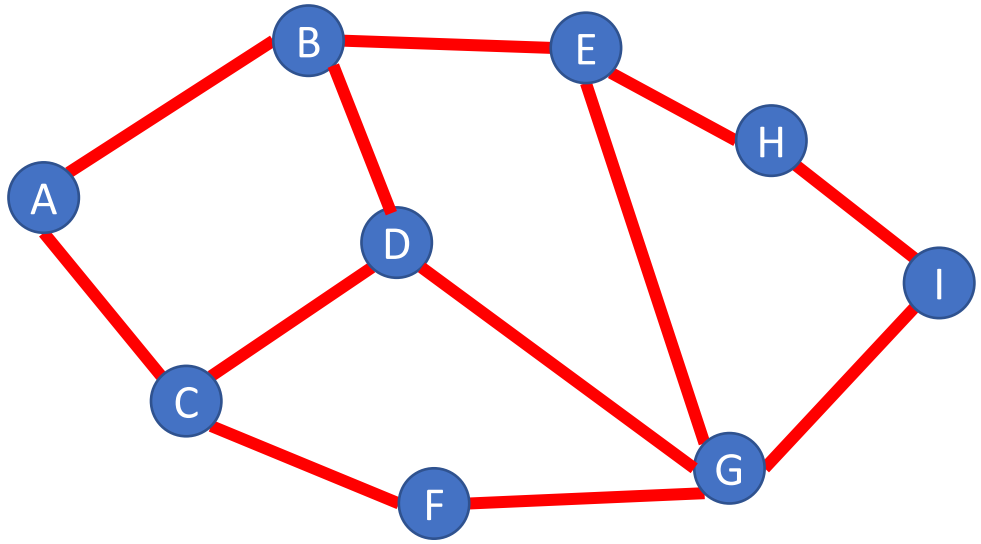
BFS: Bipartite Graph?

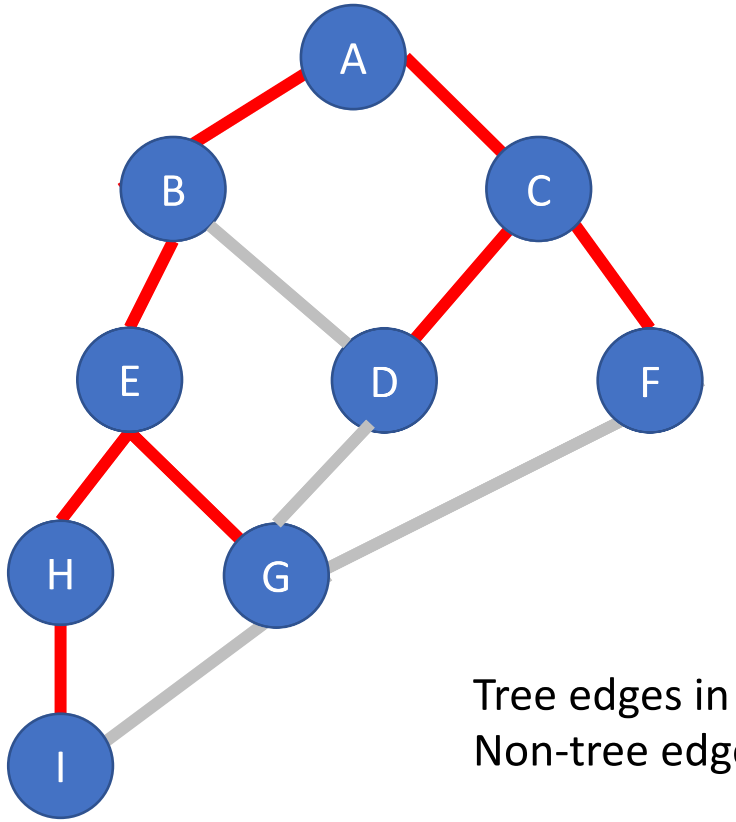
```
def isBipartite(graph, s):
    layer = 0
    depth = [-1,-1,-1,…]  # Length matches |V|
    toVisit.enqueue(s)
    depth[s] = 0
    While toVisit is not empty:
        current = toVisit.dequeue()
        layer = depth[current]
        for v in neighbors(current):
            if v not seen:
                depth[v] = layer+1
                toVisit.enqueue(v)
            elif depth[v] == depth[current]:
                return False
    return True
```

Idea: Check for edges in the same layer!



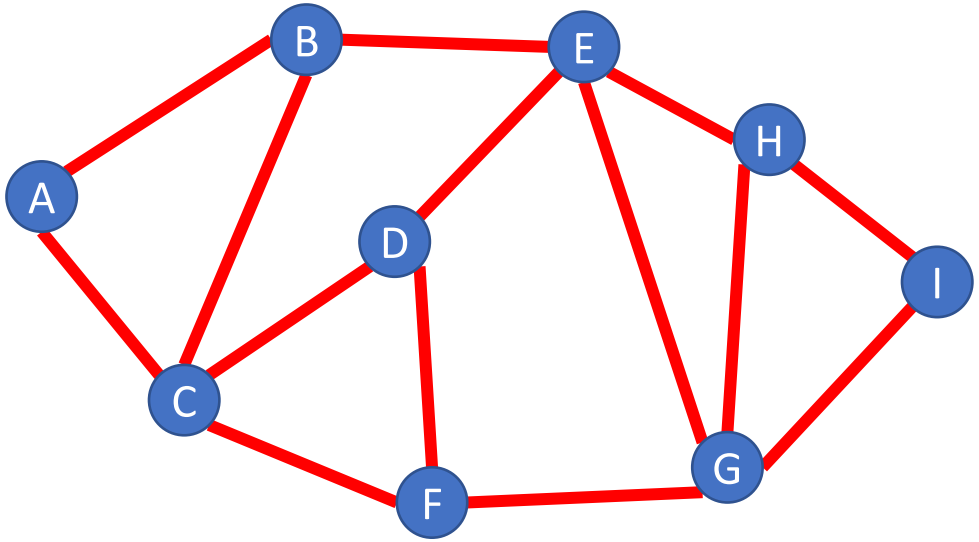
BFS Tree for a Bipartite Graph

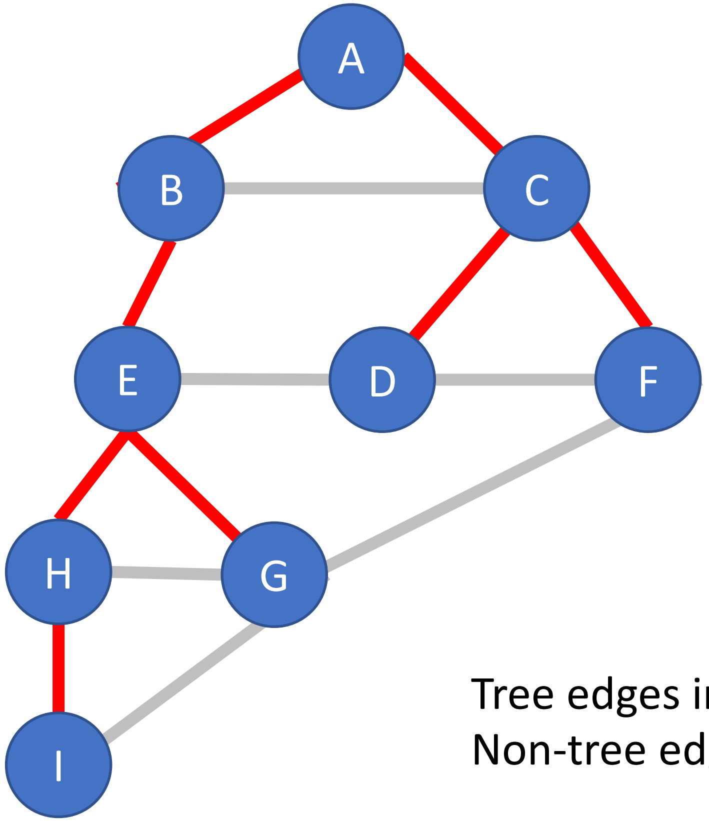




Tree edges in red
Non-tree edges in gray

BFS Tree for a Non-Bipartite Graph





Tree edges in red
Non-tree edges in gray

What’s Next?

Depth-first Search, another traversal strategy

And problems DFS can solve for us