CSE 332 Winter 2024
Lecture 16: Radix Sort, Graphs

Nathan Brunelle

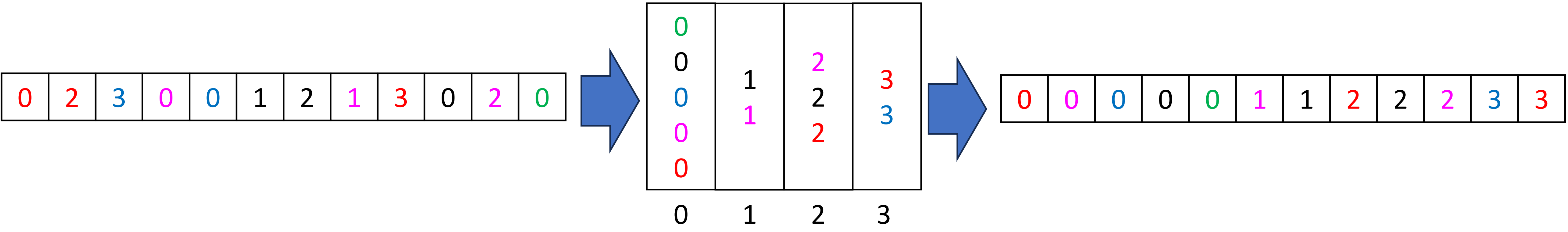
http://www.cs.uw.edu/332

“Linear Time” Sorting Algorithms

- Useable when you are able to make additional assumptions about the contents of your list (beyond the ability to compare)
  - Examples:
    - The list contains only positive integers less than k
    - The number of distinct values in the list is much smaller than the length of the list
- The running time expression will always have a term other than the list’s length to account for this assumption
  - Examples:
    - Running time might be $\Theta(k \cdot n)$ where k is the range/count of values

BucketSort

- Assumes the array contains integers between 0 and $k - 1$ (or some other small range)
- Idea:
  - Use each value as an index into an array of size k
  - Add the item into the “bucket” at that index (e.g. linked list)
  - Get sorted array by “appending” all the buckets



BucketSort Running Time

- Create array of k buckets
  - Either $\Theta(k)$ or $\Theta(1)$ depending on some things…
- Insert all n things into buckets
  - $\Theta(n)$
- Empty buckets into an array
  - $\Theta(n + k)$
- Overall:
  - $\Theta(n + k)$
- When is this better than mergesort?

Properties of BucketSort

- In-Place?
  - No
- Adaptive?
  - No
- Stable?
  - Yes!

RadixSort

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

103	801	401	323	255	823	999	101	113	901	555	512	245	800	018	121
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Place each element into a "bucket" according to its 1's place

0	1	2	3	4	5	6	7	8	9
800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999

RadixSort

- Radix: The base of a number system
  - We'll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

800	801 401 101 901 121	512	103 323 823 113		255 555 245			018	999
0	1	2	3	4	5	6	7	8	9

Place each element into a "bucket" according to its 10's place

800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999
0	1	2	3	4	5	6	7	8	9

RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

800 801 401 101 901 103	512 113 018	121 323 823		245	255 555				999
0	1	2	3	4	5	6	7	8	9

Place each element into a “bucket” according to its 100’s place

018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999
0	1	2	3	4	5	6	7	8	9

RadixSort

- Radix: The base of a number system
  - We’ll use base 10, most implementations will use larger bases
- Idea:
  - BucketSort by each digit, one at a time, from least significant to most significant

018	101 103 113 121	245 255	323	401	512 555			800 801 823	901 999
0	1	2	3	4	5	6	7	8	9

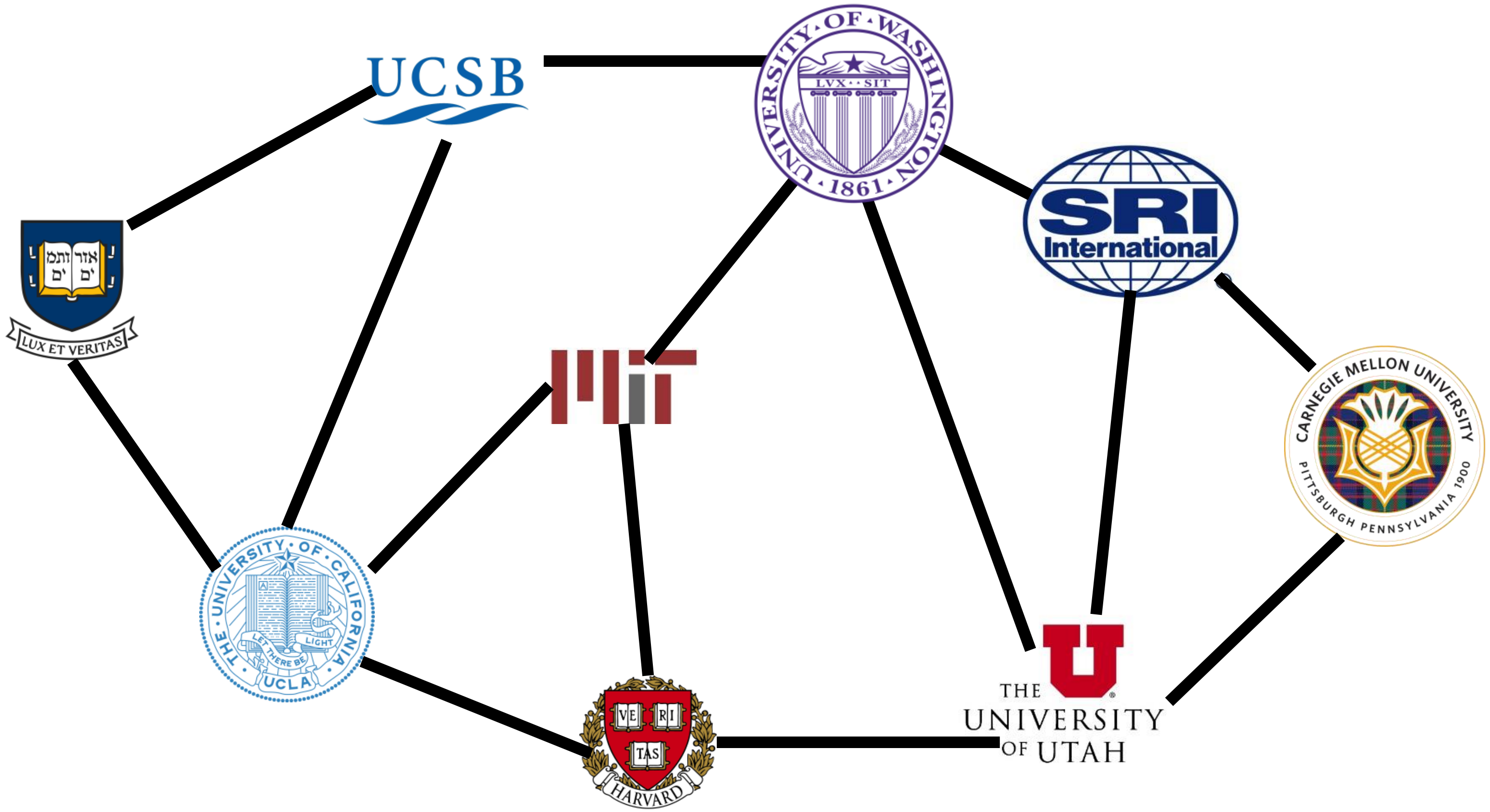
Convert back into an array

018	811	103	113	121	245	255	323	401	512	555	800	801	823	901	999
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

RadixSort Running Time

- Suppose largest value is m
- Choose a radix (base of representation) b
- BucketSort all n things using b buckets
  - $\Theta(n + k)$
- Repeat once per each digit
  - $\log_b m$ iterations
- Overall:
  - $\Theta(n \log_b m + b \log_b m)$
- In practice, you can select the value of b to optimize running time
- When is this better than mergesort?

ARPANET

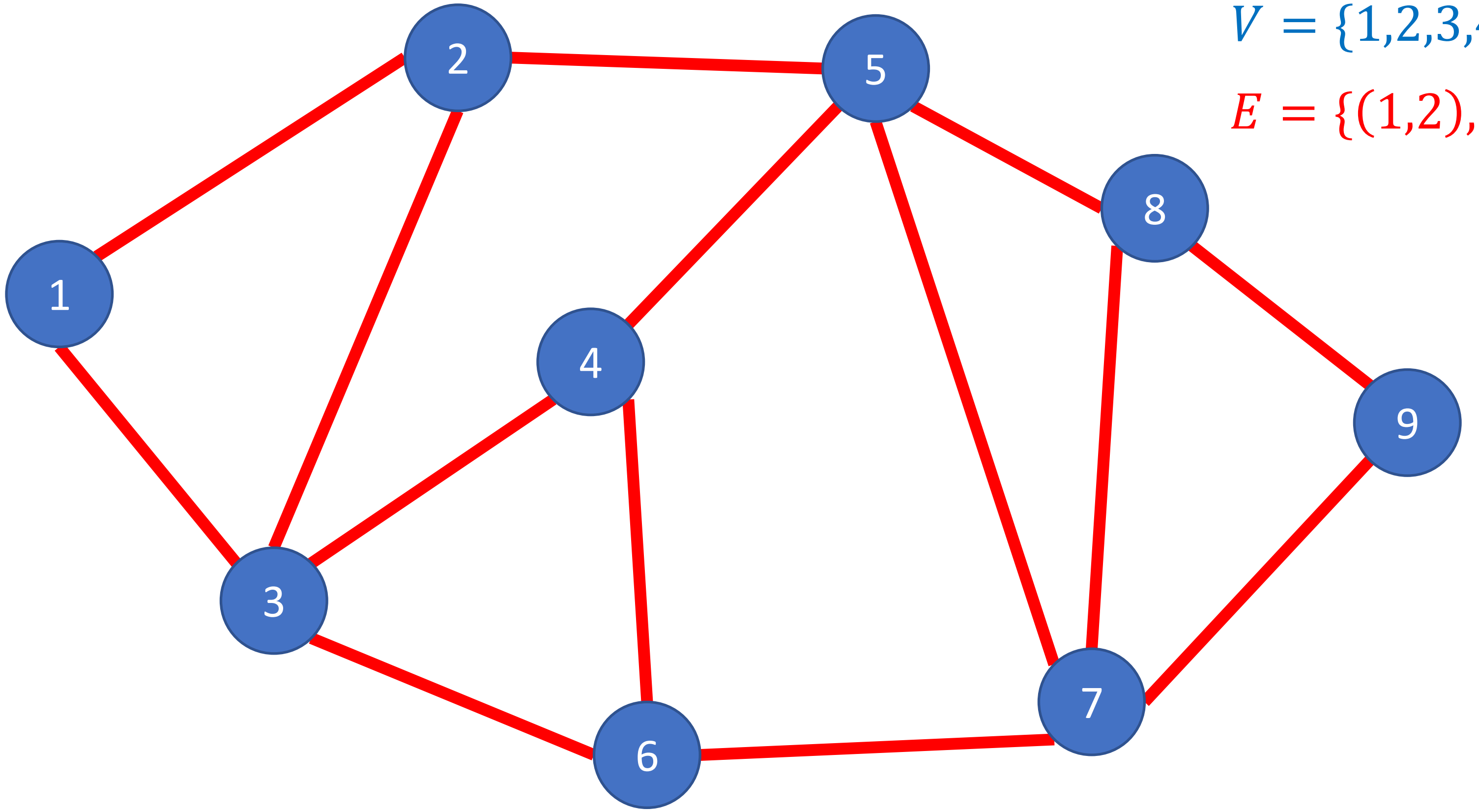

UCSB
UNIVERSITY OF WASHINGTON
LVX SIT
1861
SRI International
LUX ET VERITAS
CARNEGIE MELLON UNIVERSITY
PITTSBURGH PENNSYLVANIA 1900
THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA
VE RI TAS
HARVARD
THE UNIVERSITY OF UTAH

Undirected Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges



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

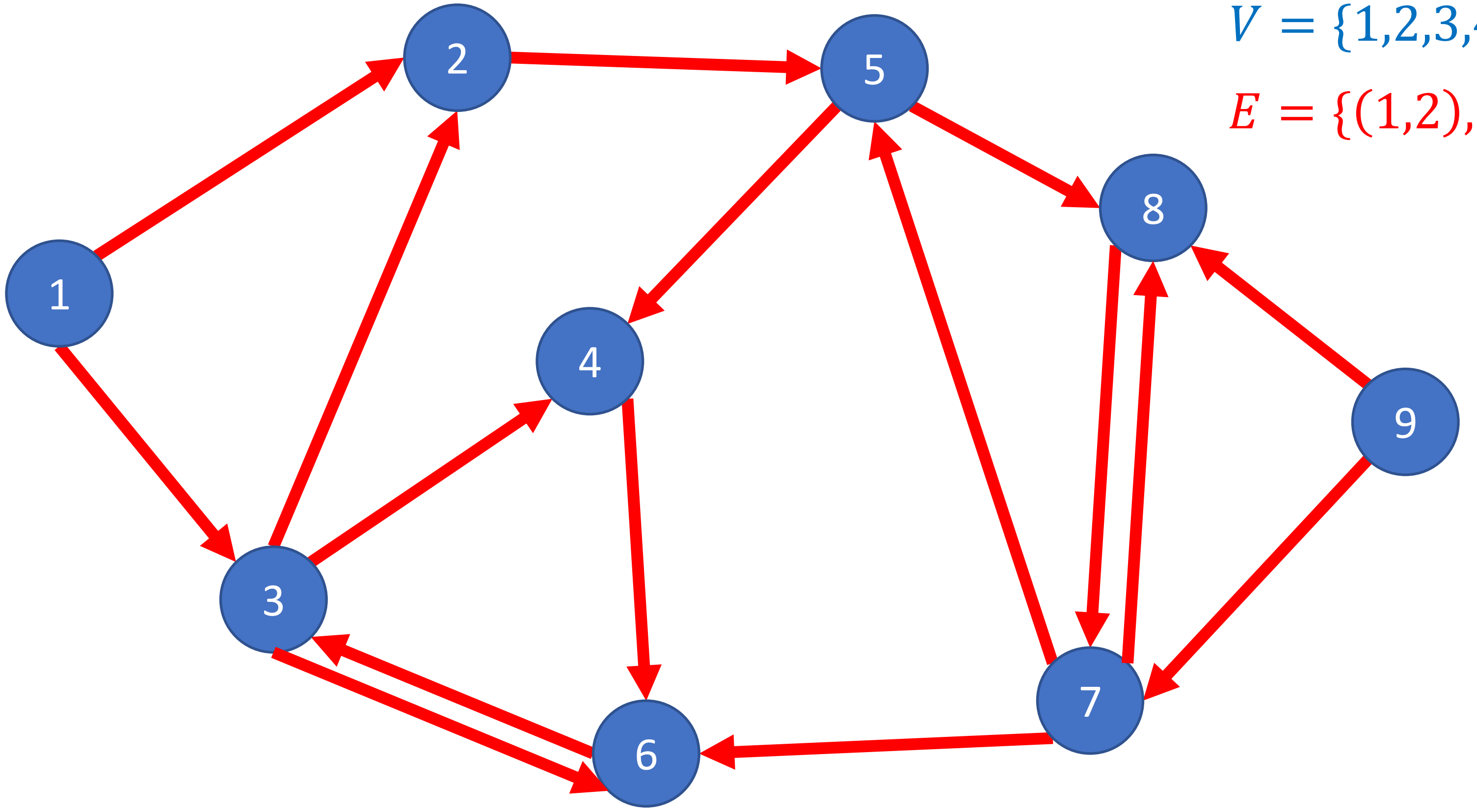
$E = \{(1,2), (2,3), (1,3), \ldots\}$

Directed Graphs

Vertices/Nodes

Definition: $G = (V, E)$

Edges

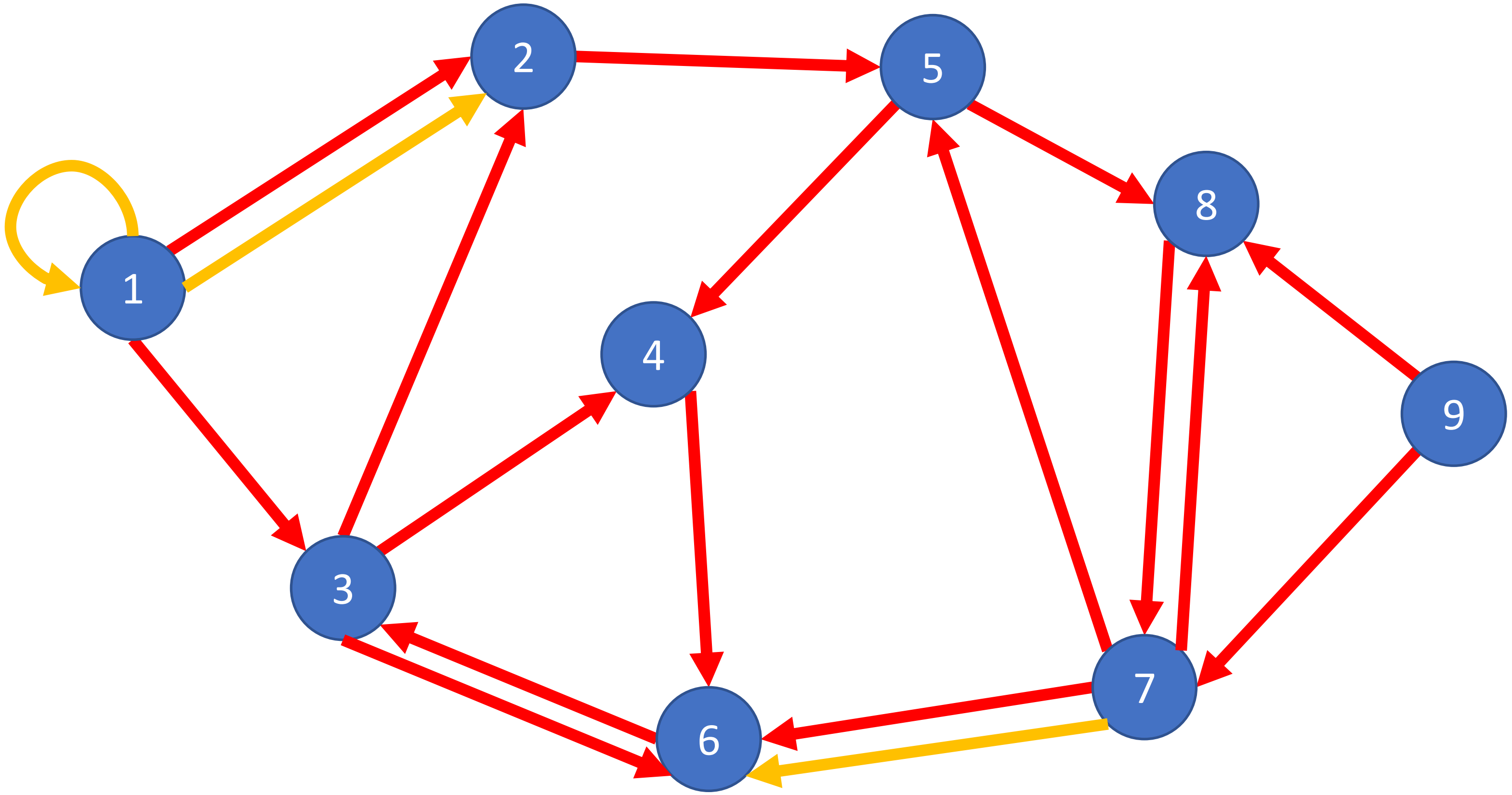


$V = \{1,2,3,4,5,6,7,8,9\}$

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

Self-Edges and Duplicate Edges

Some graphs may have duplicate edges (e.g. here we have the edge (1,2) twice).
Some may also have self-edges (e.g. here there is an edge from 1 to 1).
Graph with Neither self-edges nor duplicate edges are called **simple graphs**



Weighted Graphs

Vertices/Nodes

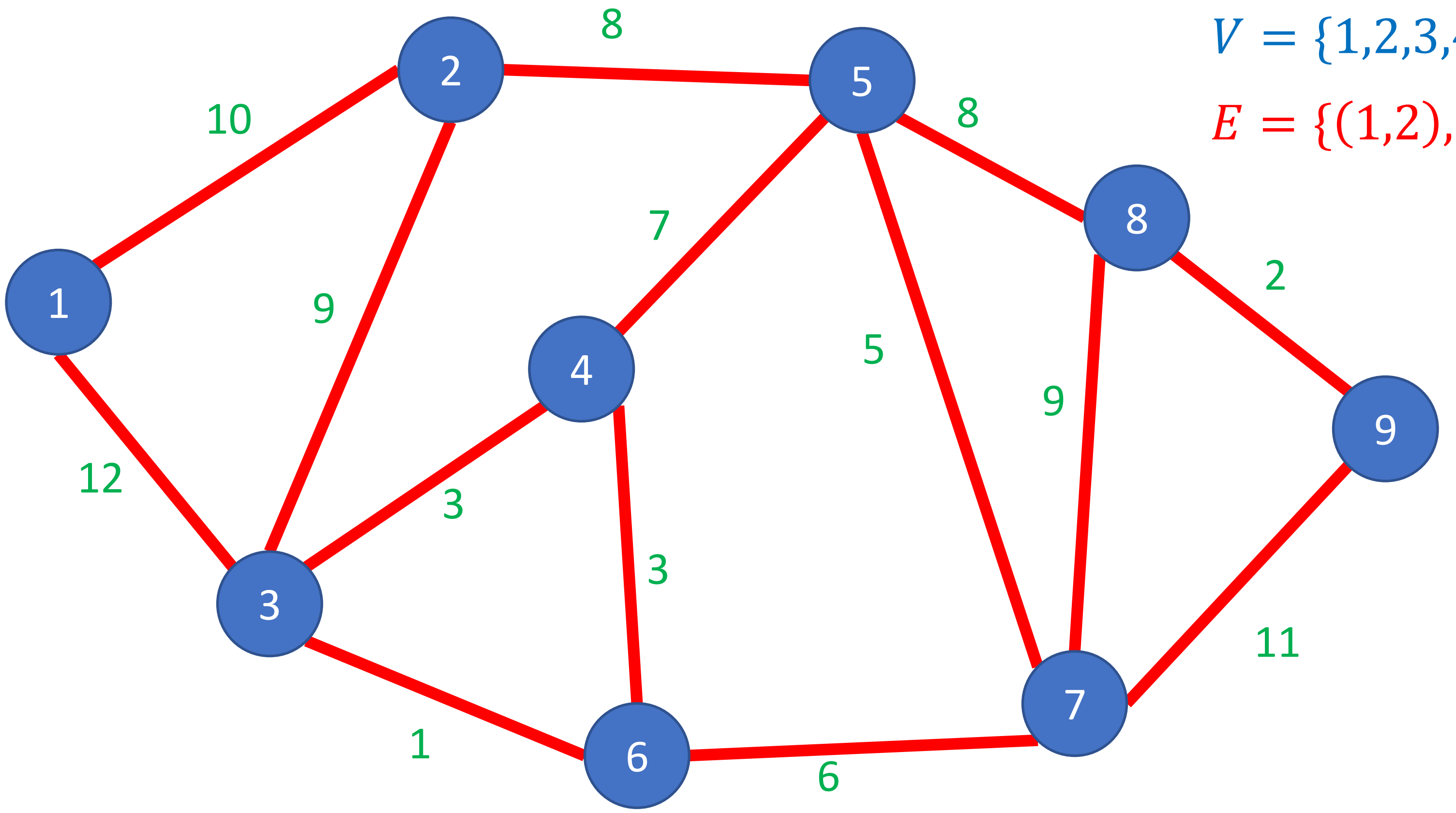
Definition: $G = (V, E)$

Edges

$w(e)$ = weight of edge e

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

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

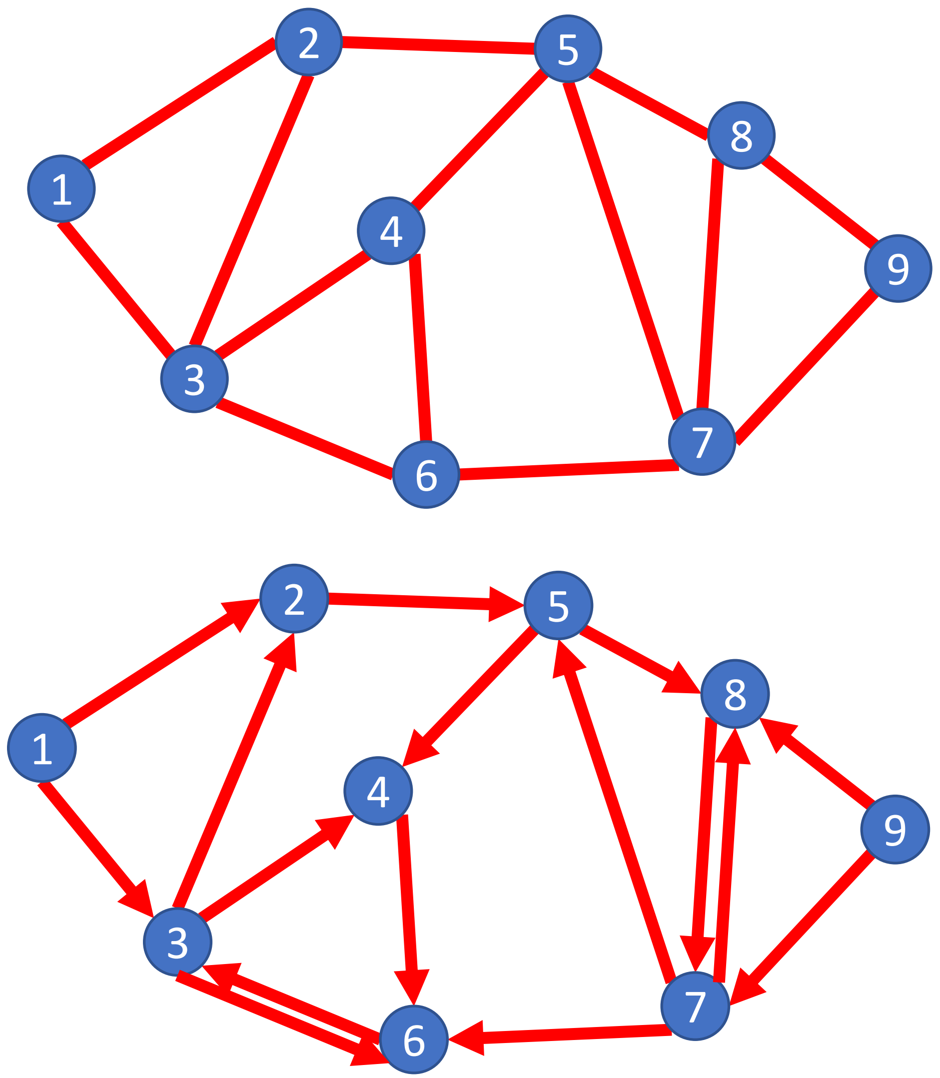


Graph Applications

- For each application below, consider:
  - What are the nodes, what are the edges?
  - Is the graph directed?
  - Is the graph simple?
  - Is the graph weighted?
- Facebook friends
- Twitter followers
- Java inheritance
- Airline Routes

Some Graph Terms

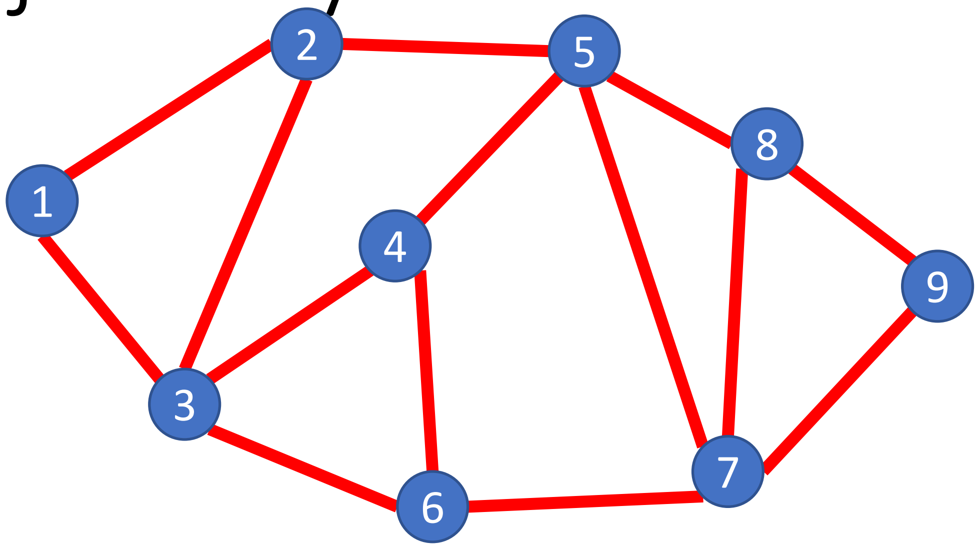
- Adjacent/Neighbors
  - Nodes are adjacent/neighbors if they share an edge
- Degree
  - Number of “neighbors” of a vertex
- Indegree
  - Number of incoming neighbors
- Outdegree
  - Number of outgoing neighbors



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)

Adjacency List



1	2	3		
2	1	3	5	
3	1	2	4	6
4	3	5	6	
5	2	4	7	8
6	3	4	7	
7	5	6	8	9
8	5	7	9	
9	7	8		

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n+m)$

Add Edge: $\Theta(1)$

Remove Edge: $\Theta(1)$

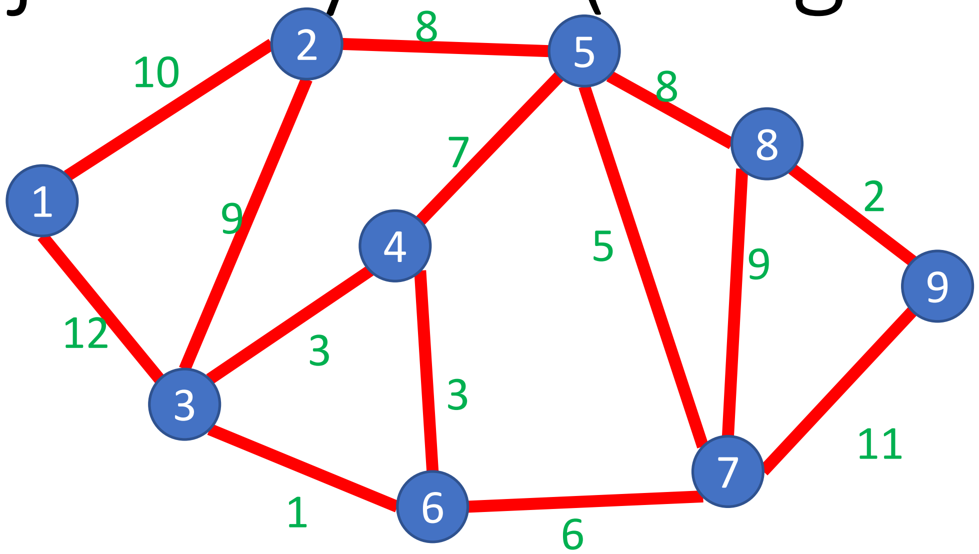
Check if Edge Exists: $\Theta(n)$

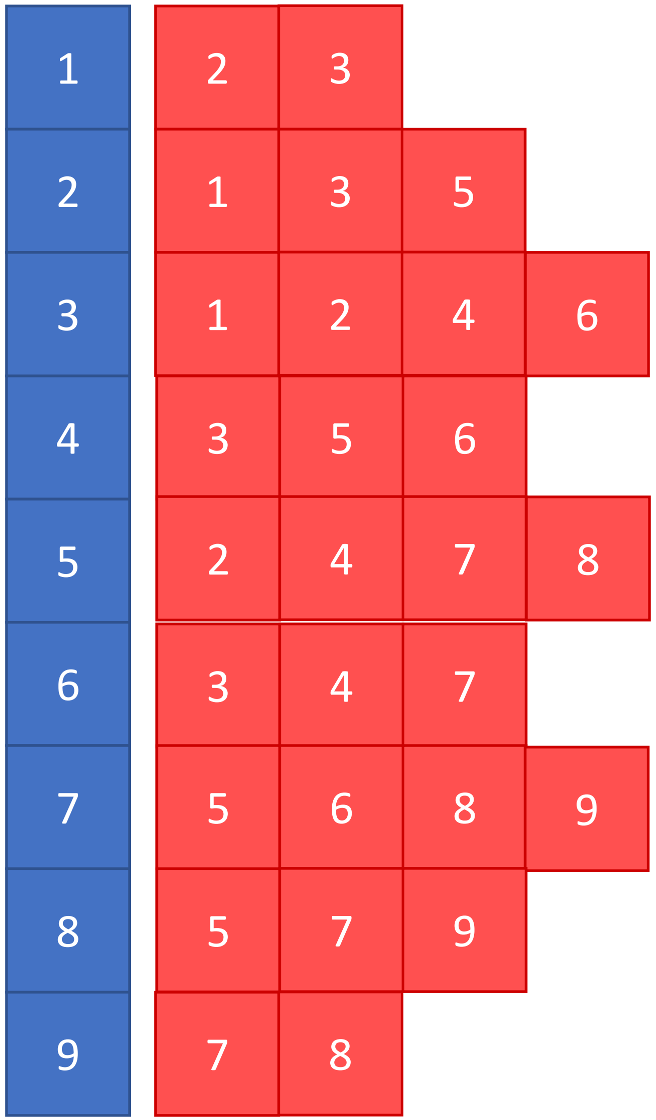
Get Neighbors (incoming): $\Theta(n+m)$

Get Neighbors (outgoing): $\Theta(\deg(v))$

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

Adjacency List (Weighted)





<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n+m)$

Add Edge: $\Theta(1)$

Remove Edge: $\Theta(1)$

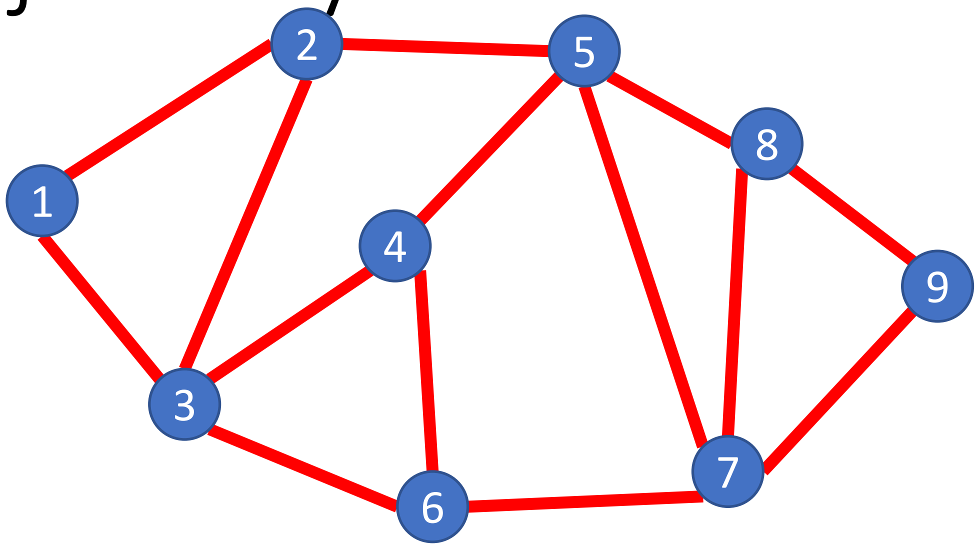
Check if Edge Exists: $\Theta(n)$

Get Neighbors (incoming): $\Theta(?)$

Get Neighbors (outgoing): $\Theta(?)$

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

Adjacency Matrix



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(?)$

Add Edge: $\Theta(?)$

Remove Edge: $\Theta(?)$

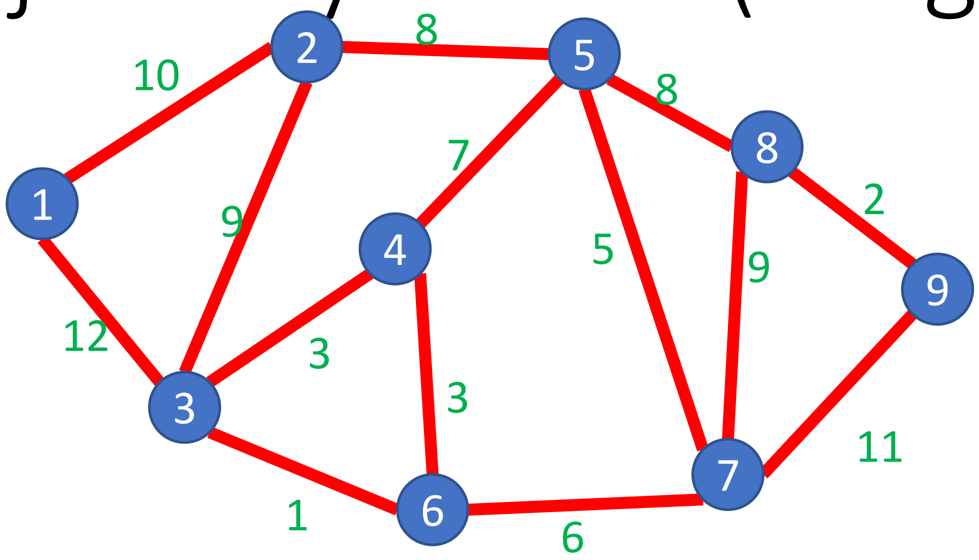
Check if Edge Exists: $\Theta(?)$

Get Neighbors (incoming): $\Theta(?)$

Get Neighbors (outgoing): $\Theta(?)$

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

Adjacency Matrix (weighted)



	A	B	C	D	E	F	G	H	I
A		1	1						
B	1		1		1				
C	1	1		1		1			
D			1		1	1			
E		1		1			1	1	
F			1	1			1		
G					1	1		1	1
H					1		1		1
I							1	1	

<u>Time/Space Tradeoffs</u>

Space to represent: $\Theta(n^2)$

Add Edge: $\Theta(1)$

Remove Edge: $\Theta(1)$

Check if Edge Exists: $\Theta(1)$

Get Neighbors (incoming): $\Theta(n)$

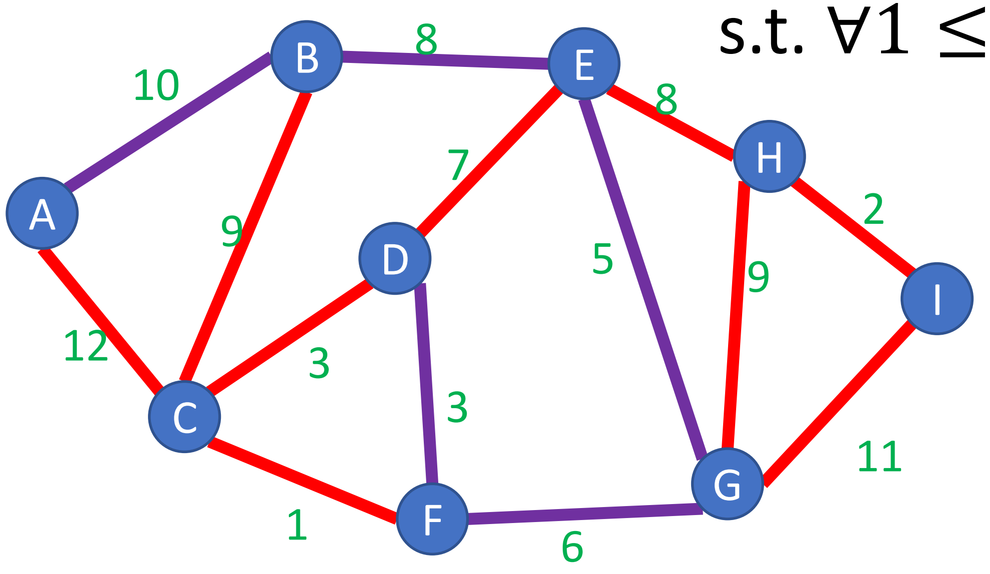
Get Neighbors (outgoing): $\Theta(n)$

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

Aside

- Almost always, adjacency lists are the better choice
- Most graphs are missing most of their edges, so the adjacency list is much more space efficient and the slower operations aren’t that bad

Definition: Path



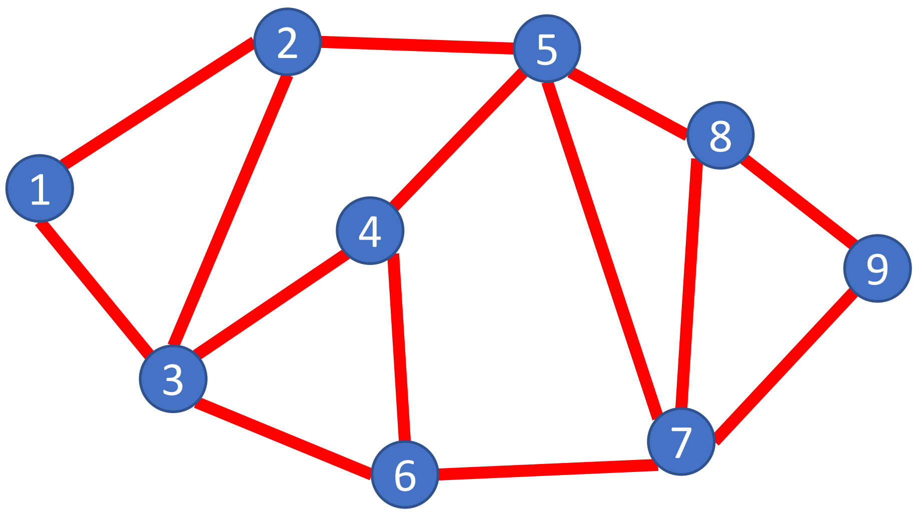
A sequence of nodes $(v_1, v_2, \dots, v_k)$
s.t. $\forall 1 \le i \le k-1, (v_i, v_{i+1}) \in E$

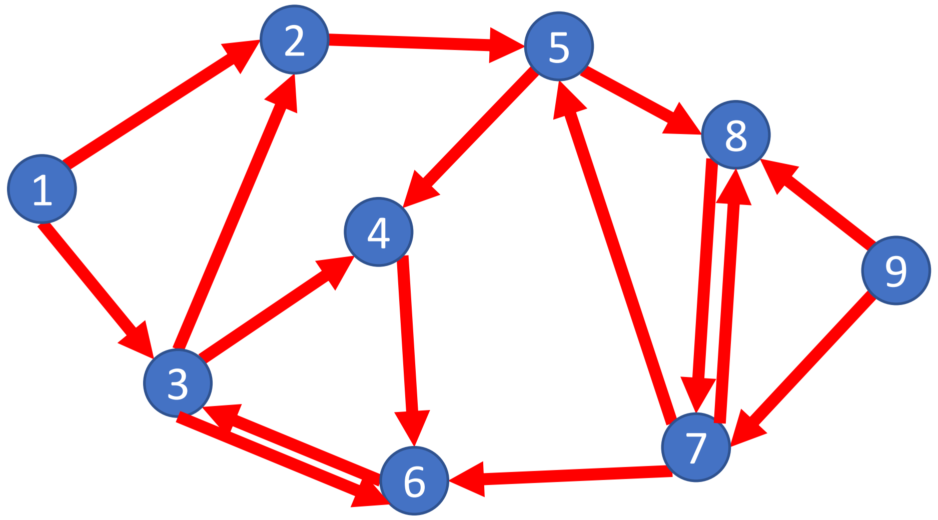
<u>Simple Path:</u>
A path in which each node appears at most once

<u>Cycle:</u>
A path which starts and ends in the same place

Definition: (Strongly) Connected Graph

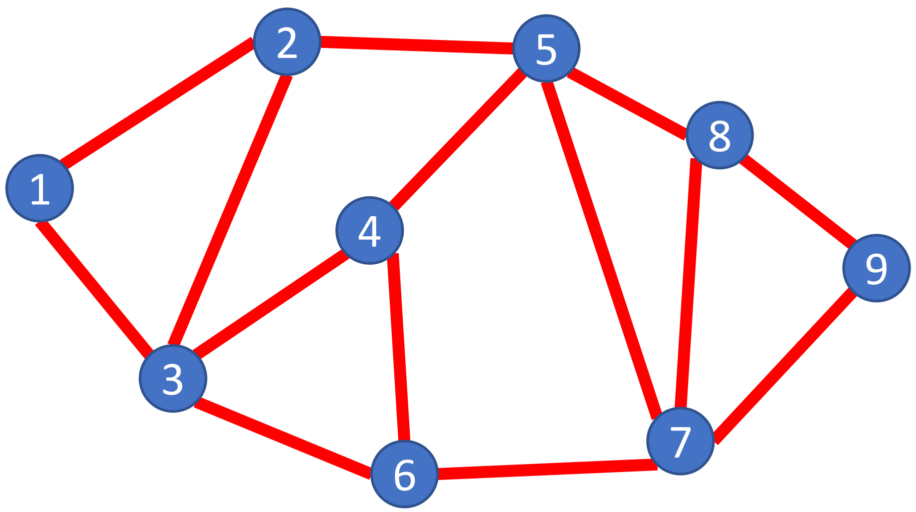
A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is a path from v_1 to v_2



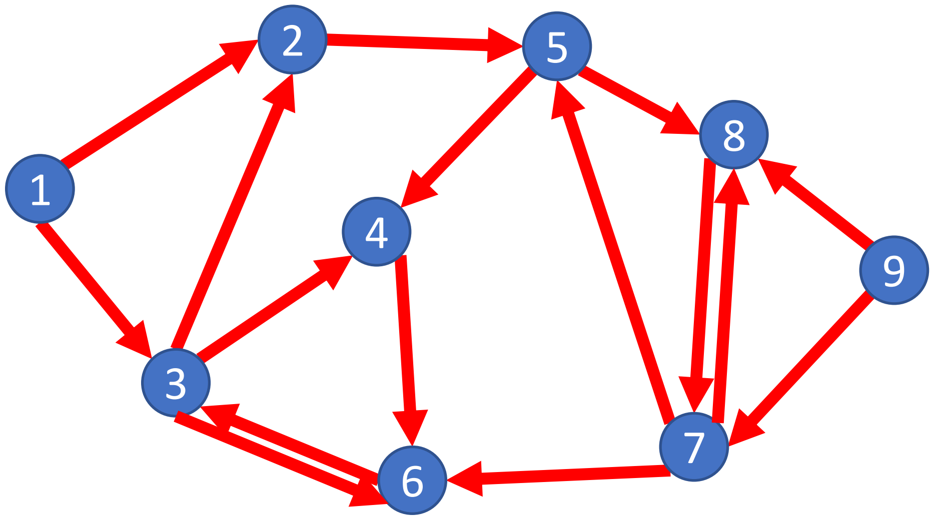


Definition: (Strongly) Connected Graph

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



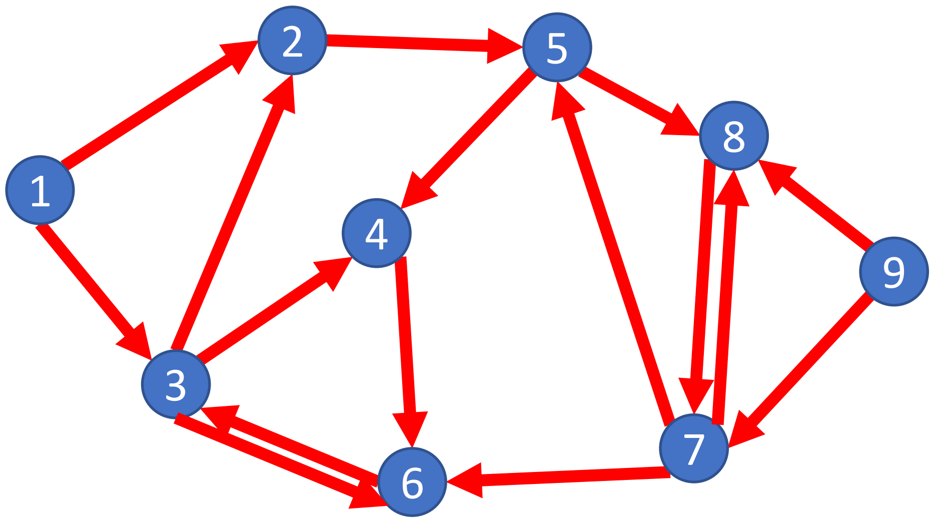
Connected



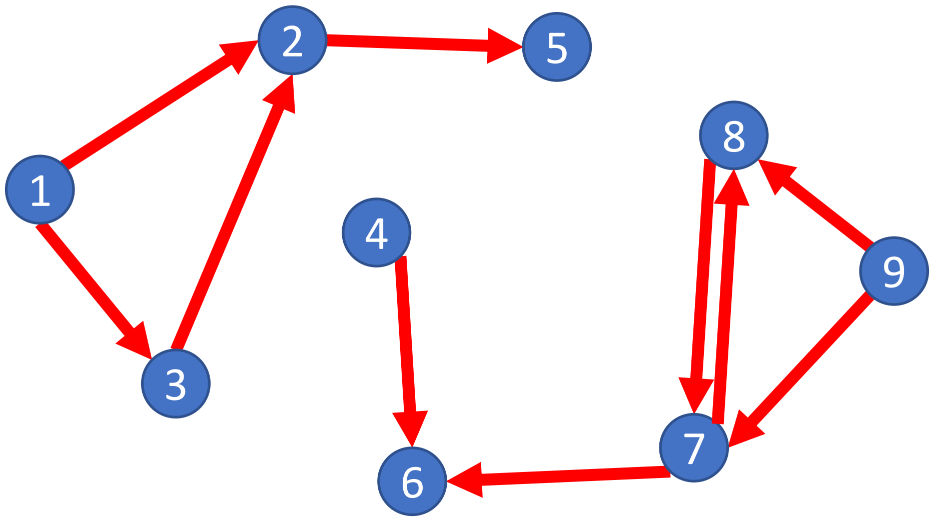
Not (strongly) Connected

Definition: Weakly Connected Graph

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 ignoring direction of edges



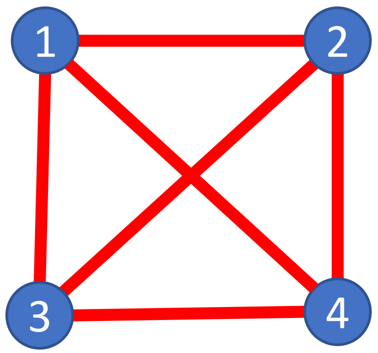
Weakly Connected



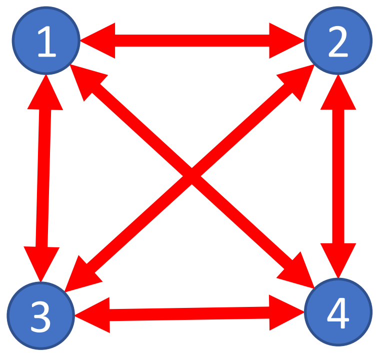
Weakly Connected

Definition: Complete Graph

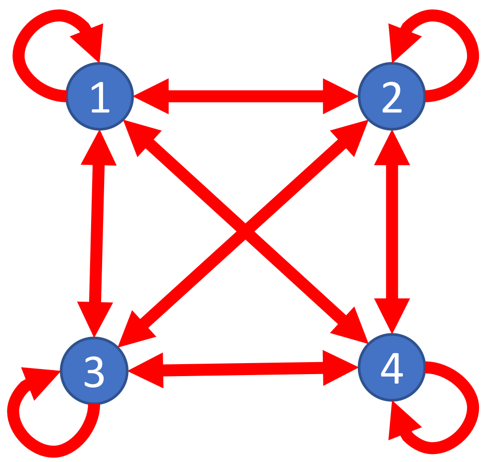
A Graph $G = (V, E)$ s.t. for any pair of nodes $v_1, v_2 \in V$ there is an edge from v_1 to v_2



Complete
Undirected Graph



Complete
Directed Graph



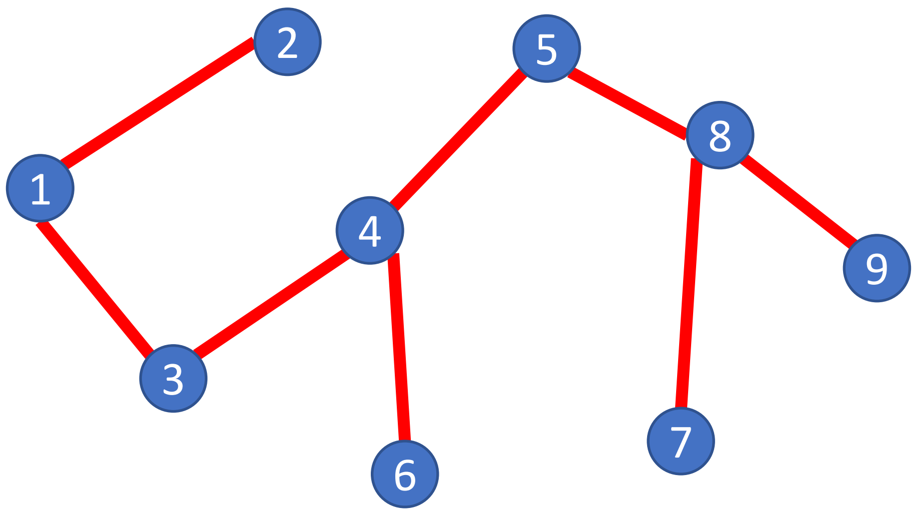
Complete Directed
Non-simple Graph

Graph Density, Data Structures, Efficiency

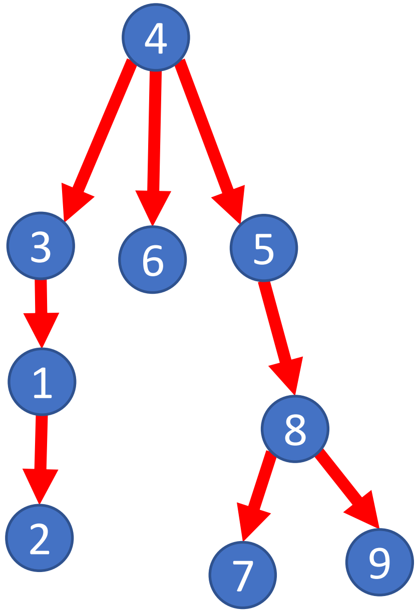
- The maximum number of edges in a graph is $\Theta(|V|^2)$:
  - Undirected and simple: $\frac{|V|(|V|-1)}{2}$
  - Directed and simple: $|V|(|V|-1)$
  - Direct and non-simple (but no duplicates): $|V|^2$
- If the graph is connected, the minimum number of edges is $|V|-1$
- If $|E| \in \Theta(|V|^2)$ we say the graph is **dense**
- If $|E| \in \Theta(|V|)$ we say the graph is **sparse**
- Because $|E|$ is not always near to $|V|^2$ we do not typically substitute $|V|^2$ for $|E|$ in running times, but leave it as a separate variable

Definition: Tree

A Graph $G = (V, E)$ is a tree if it is undirect, connected, and has no cycles (i.e. is acyclic). Often one node is identified as the "root"



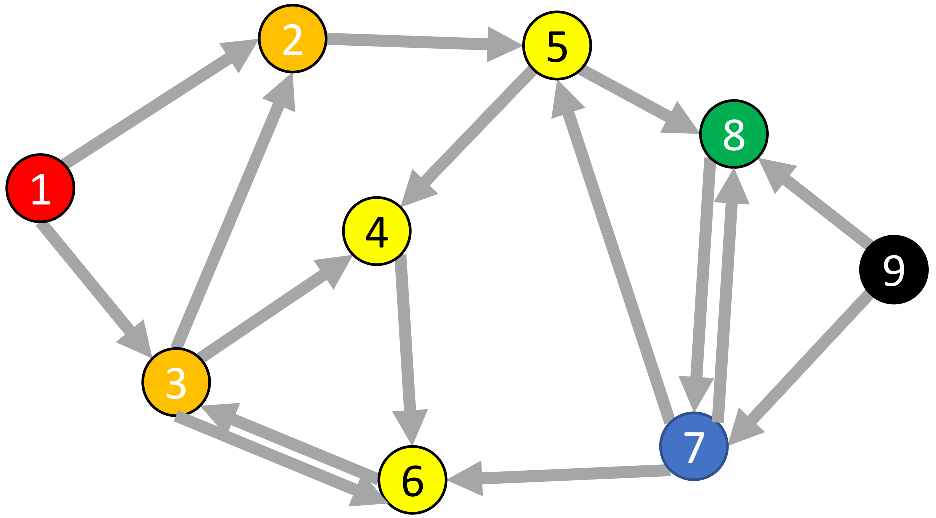
A Tree



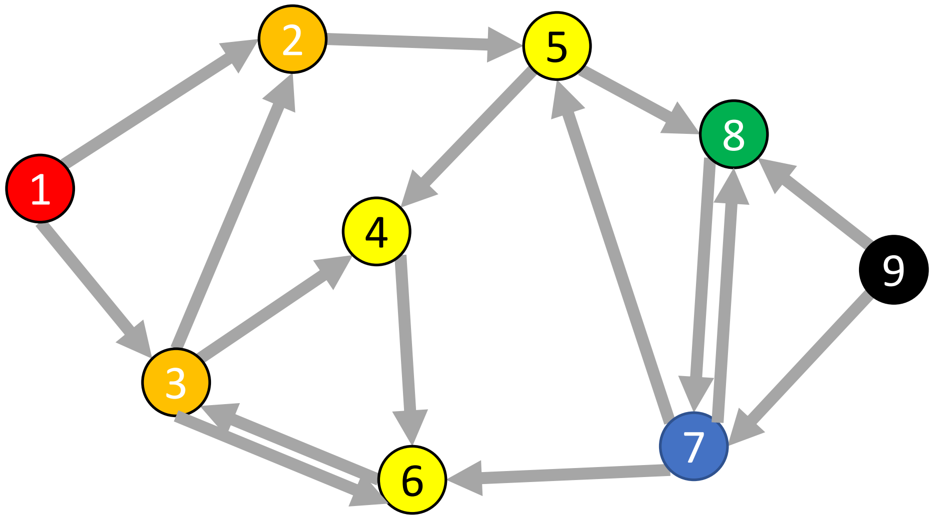
A Rooted Tree

Breadth-First Search

- Input: a node s
- Behavior: Start with node s, visit all neighbors of s, then all neighbors of neighbors of s, …
- Output:
  - How long is the shortest path?
  - Is the graph connected?



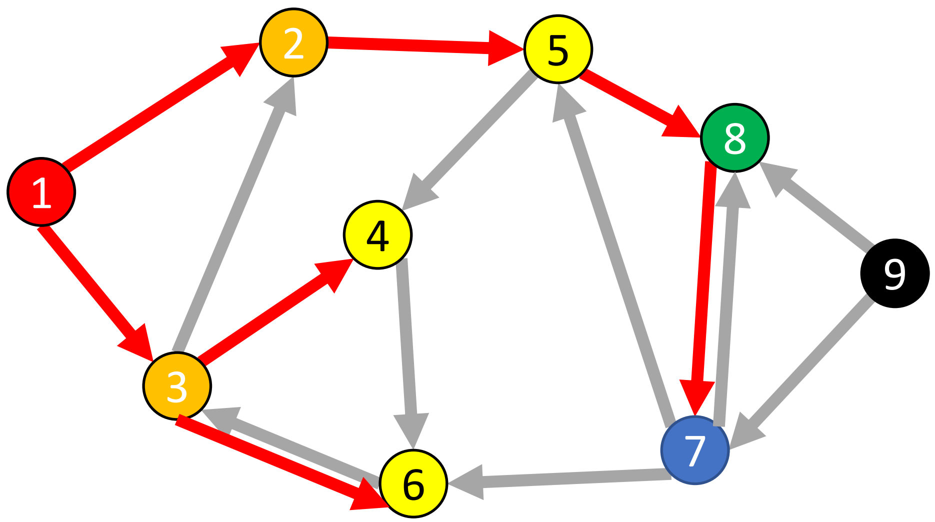
BFS



Running time: $\Theta(|V| + |E|)$

```
void bfs(graph, s){
        found = new Queue();
        found.enqueue(s);
        mark s as "visited";
        While (!found.isEmpty()){
                current = found.dequeue();
                for (v : neighbors(current)){
                        if (! v marked "visited"){
                                mark v as "visited";
                                found.enqueue(v);
                        }
                }
        }
}
```

Shortest Path (unweighted)



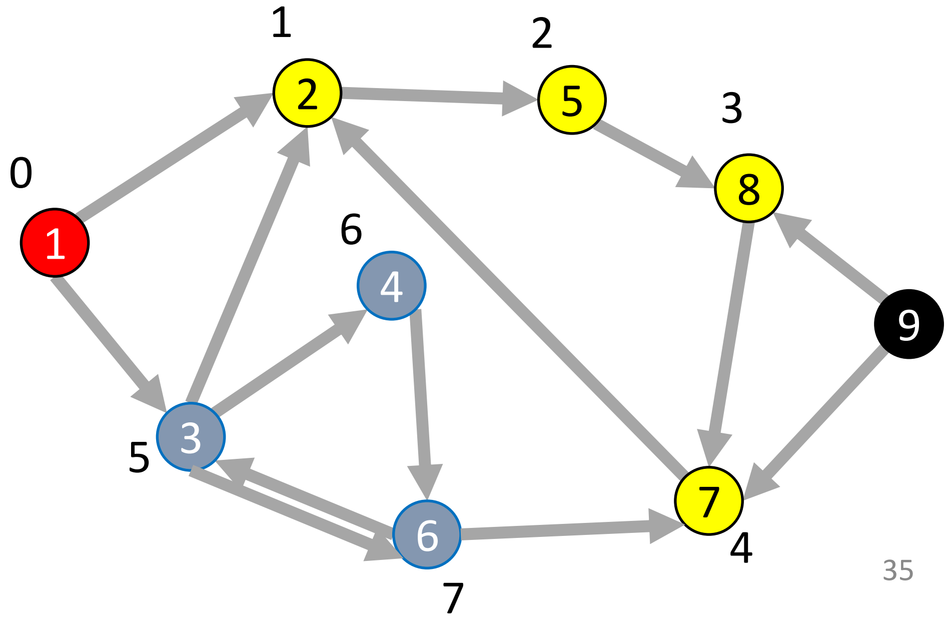
Idea: when it's seen, remember its "layer" depth!

```
int shortestPath(graph, s, t){
        found = new Queue();
        layer = 0;
        found.enqueue(s);
        mark s as "visited";
        While (!found.isEmpty()){
                current = found.dequeue();
                layer = depth of current;
                for (v : neighbors(current)){
                        if (! v marked "visited"){
                                mark v as "visited";
                                depth of v = layer + 1;
                                found.enqueue(v);
                        }
                }
        }
        return depth of t;
}
```

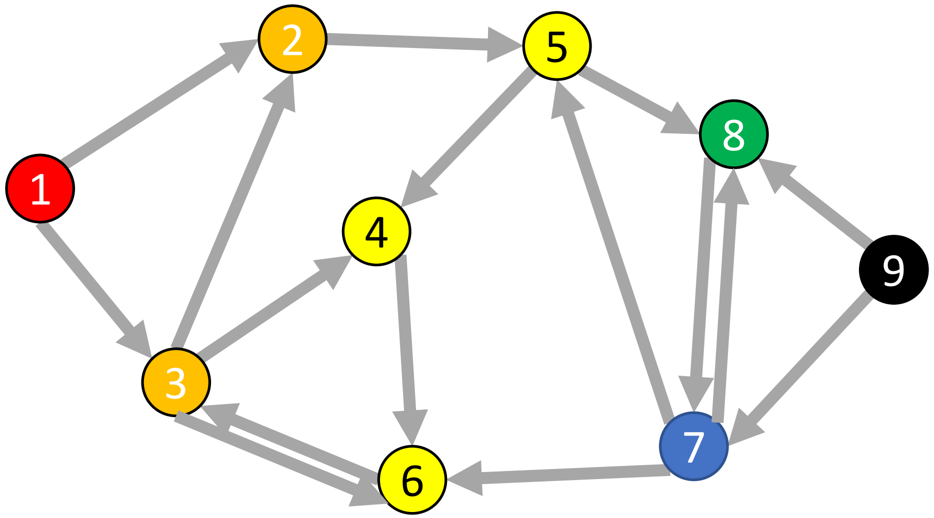
Depth-First Search

Depth-First Search

- Input: a node s
- Behavior: Start with node s, visit one neighbor of s, then all nodes reachable from that neighbor of s, then another neighbor of s,…
- Output:
  - Does the graph have a cycle?
  - A **topological sort** of the graph.



DFS (non-recursive)



Running time: $\Theta(|V| + |E|)$

```
void dfs(graph, s){
        found = new Stack();
        found.pop(s);
        mark s as "visited";
        While (!found.isEmpty()){
                current = found.pop();
                for (v : neighbors(current)){
                        if (! v marked "visited"){
                                mark v as "visited";
                                found.push(v);
                        }
                }
        }
}
```

DFS Recursively (more common)

```
void dfs(graph, curr){
        mark curr as "visited";
        for (v : neighbors(current)){
                if (! v marked "visited"){
                        dfs(graph, v);
                }
        }
        mark curr as "done";
}
```