Lecture 12: Graph Algorithms

Tim LaRock

larock.t@northeastern.edu

bit.ly/cs3000syllabus

Business

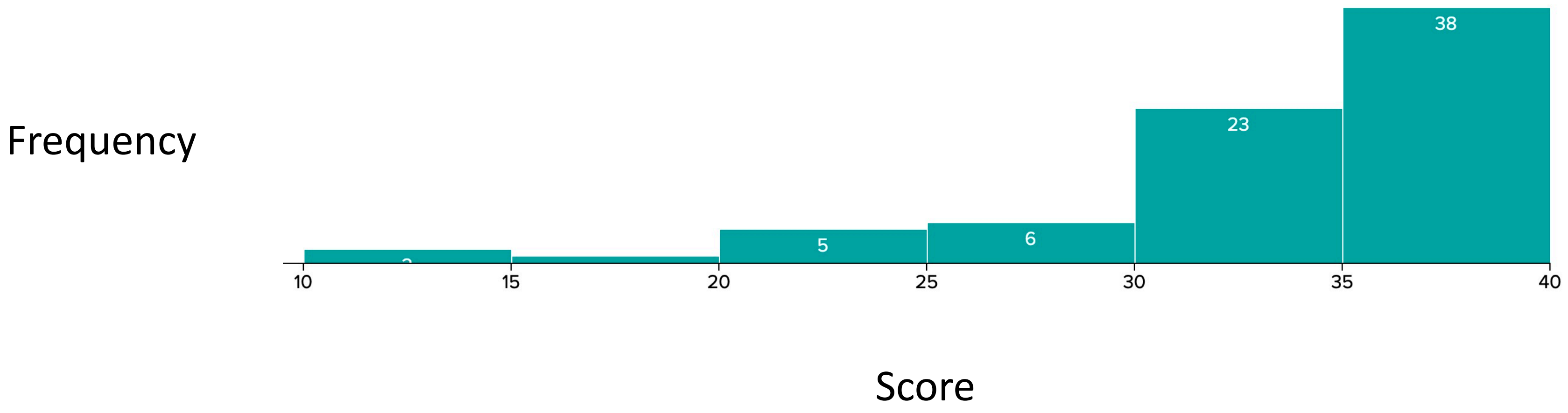
Homework 2

Midterm

Homework 3

Homework 2

- Grades released earlier – overall good work!
  - Median: 34.75
  - Mean: 32.38
  - Histogram below
- Request regrades directly on gradescope!
  - You can also email me, but gradescope is much better and likely to be faster!
- Quite a few no submissions and partial submissions – if this was a mistake or miscommunication, you need to let me know ASAP!!
  - I will try to be accommodating, but it is your responsibility to make sure you turn things in correctly!



Midterm: Some high level stuff

Overall from initial grading it seems like people did well!

- We are aiming to have your grades by the end of the week. Thank you for your patience!

Pseudocode: high level, abstract description of an algorithm

- Focus on readability and helping understand the algorithm
- Someone reading it should be able to implement it in any language without knowing any other language
  - translation should be english → implementation language, not implementation language1→implementation language2, since that requires knowing language1 which defeats the purpose!
- No strict syntax – when faced with options, choose the clearest and most concise that you an think of!
- NOT code

Recursive specification = Algorithm or recursive pseudocode

Recurrence relation = Runtime calculation like $T(n) = T(n/a) + O(f(n))$

Fibonacci Numbers: Recurrence Relation

$$f_n \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f_{n-1} + f_{n-2} & otherwise \end{cases}$$

What does the recurrence relation $T(n)$ look like?

$$T(0) = 1, T(1) = 1$$

$$T(n) = T(n-1) + T(n-2) + 1$$

```
Fib(n):
  If n = 0:
    return 0
  ElseIf n = 1:
    return 1
  Else:
    return Fib(n − 1) + Fib(n − 2)
```

Not pseudocode!

```
class fibonacci
{
  static int fib(int n)
  {
    if (n <= 1)
      return n;
    return fib(n-1) + fib(n-2);
  }
}
```

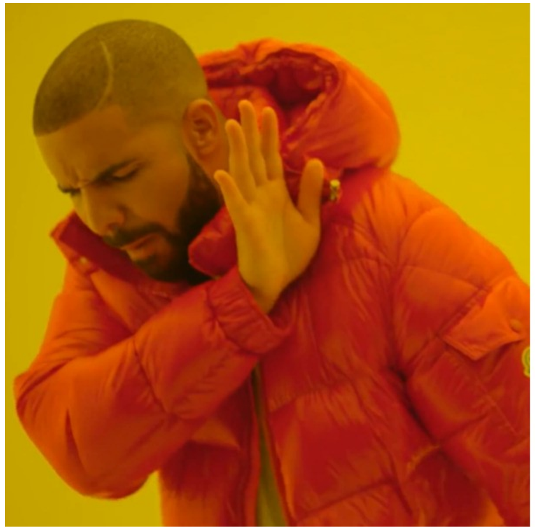
Homework 3: Graphs ’n stuff

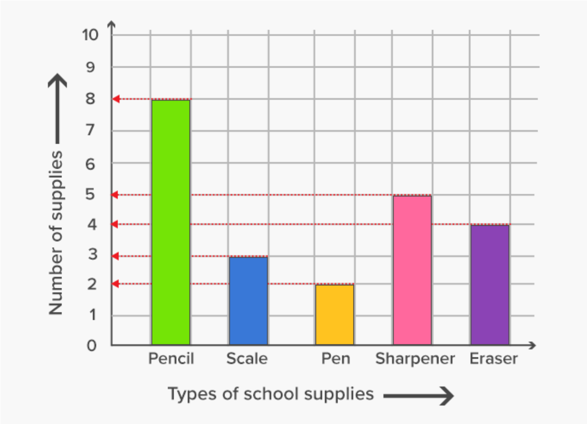
- Will be released after class
- Due next Monday June 1st at Midnight Boston time on Gradescope
  - No more canvas submissions!

- I really tried to make sure this one will be less time consuming!

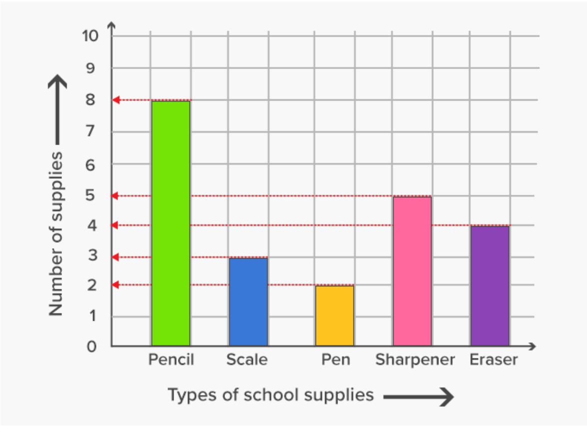
Graphs: what are they?

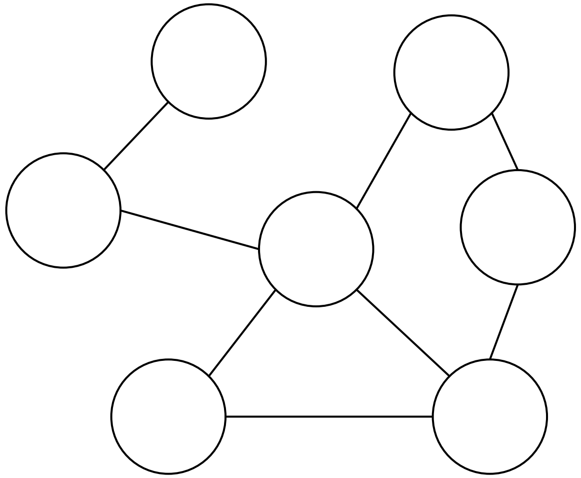
Graphs: what are they?





Graphs: what are they?





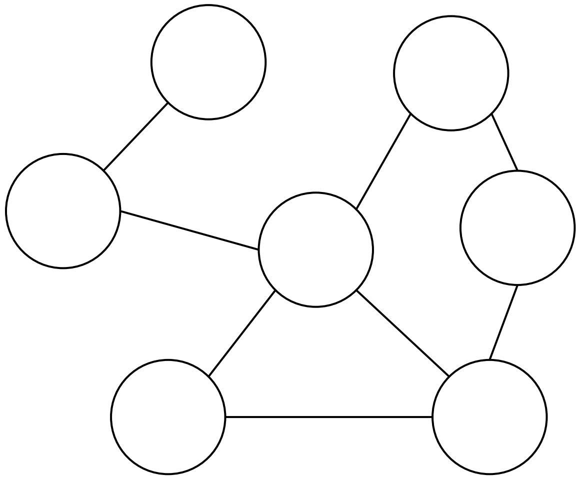
Graphs

A *graph* $G = (V, E)$ consists of
- *vertices* $v \in V$ and
- *edges* $e \in E$, indicating two vertices u, v are *connected*

We will often refer to **vertices** as ***nodes*** and to **edges** as ***links***. Consider these interchangeable!

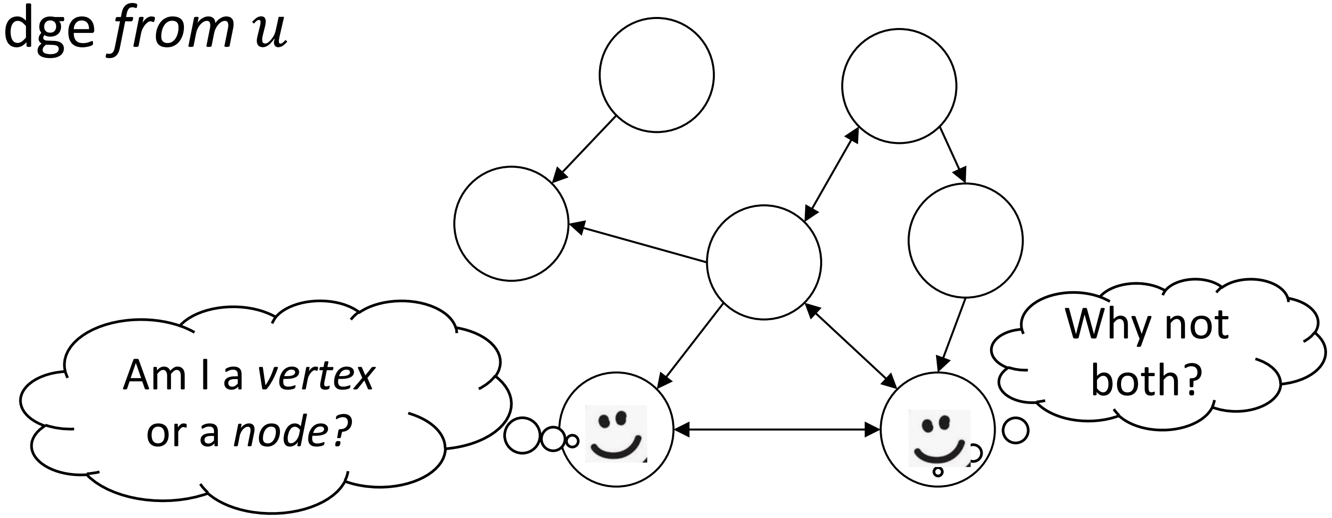
$n = |V|$, number of nodes
$m = |E|$, number of edges

Undirected



A graph is:
- *directed* if the vertices in each edge are ordered
  - If we are being precise, we say "the edge *from* u *to* v"
  - $(u, v) \in E \nRightarrow (v, u) \in E$
- *undirected* if its edges are not ordered.
  - "the edge *between* u *and* v"
  - $(u, v) \in E \Rightarrow (v, u) \in E$

Directed



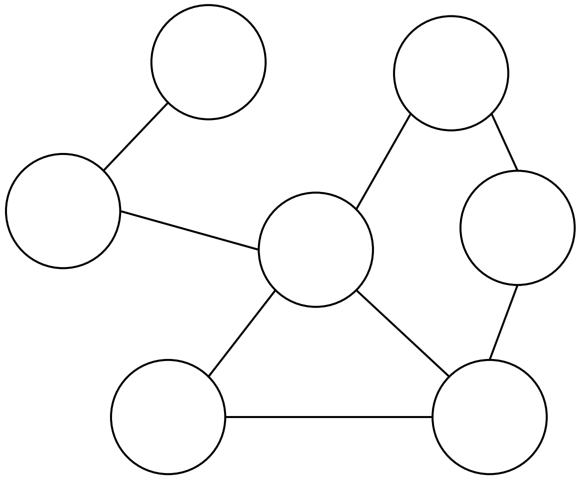
Question

Assume we have a directed graph $G = (V, E)$ that is *simple*, meaning there is at most one of each possible edge and no self-loops.

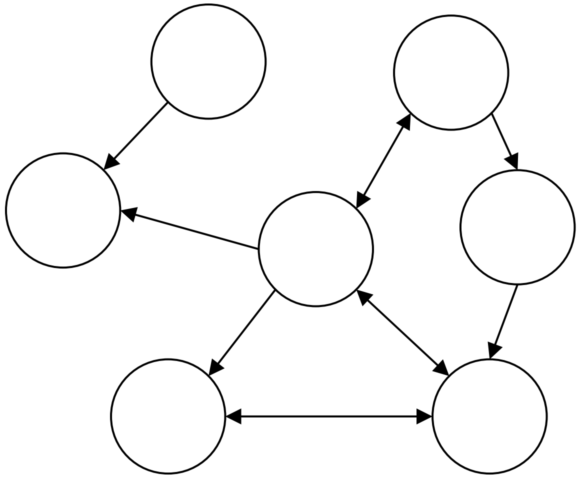
What is the maximum size of the set of edges E?

What about in an undirected graph?

Undirected

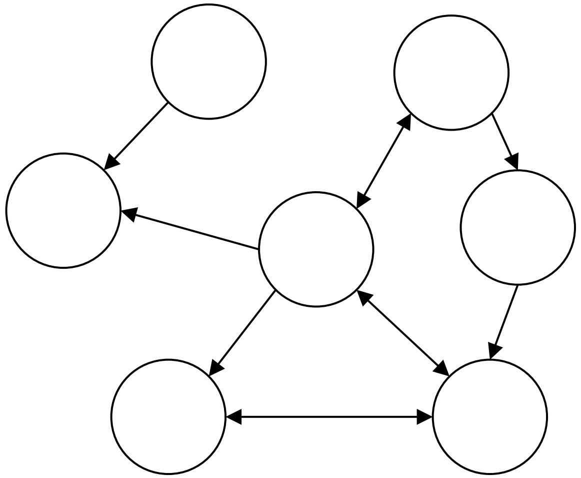


Directed



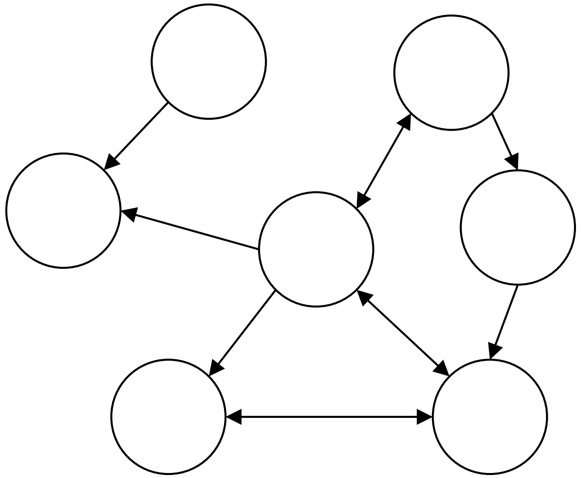
Proof by contradiction

Given a simple directed graph $G = (V, E)$ with $n = |V|$ nodes, we want to prove that the maximum size of the edge set E is $|\mathrm{E}| = n \cdot (n-1)$.



Proof by contradiction

Given a simple directed graph $G = (V, E)$ with $n = |V|$ nodes, we want to prove that the maximum size of the edge set E is $|\mathrm{E}| = n \cdot (n-1)$.

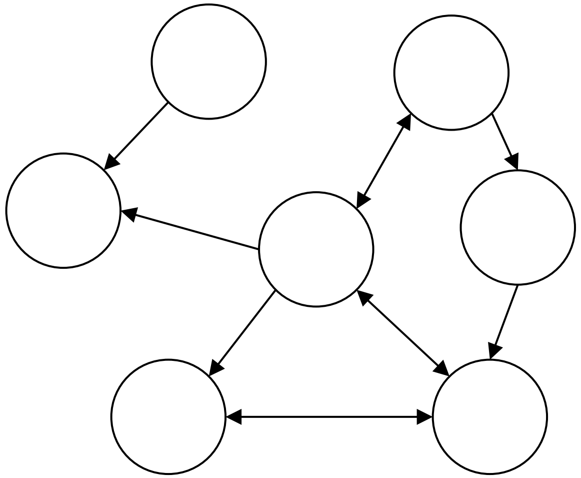


Assume for contradiction that we have a graph with $|E| > n \cdot (n-1)$.

Proof by contradiction

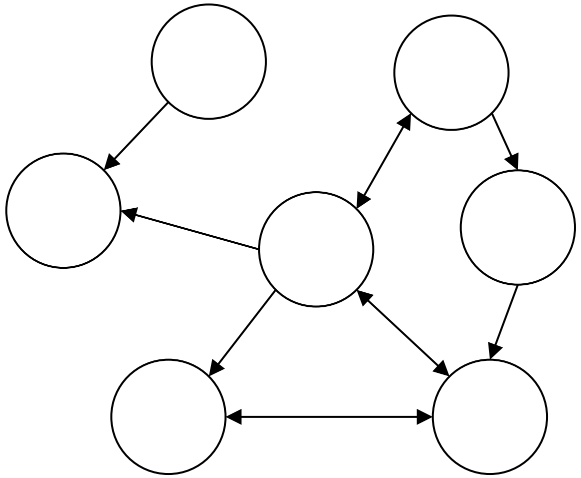
Given a simple directed graph $G = (V, E)$ with $n = |V|$ nodes, we want to prove that the maximum size of the edge set E is $|\mathrm{E}| = n \cdot (n-1)$.

Assume for contradiction that we have a graph with $|E| > n \cdot (n-1)$. This implies that there is some node u with more than $(n-1)$ neighbors.



Proof by contradiction

Given a simple directed graph $G = (V, E)$ with $n = |V|$ nodes, we want to prove that the maximum size of the edge set E is $|\mathrm{E}| = n \cdot (n-1)$.

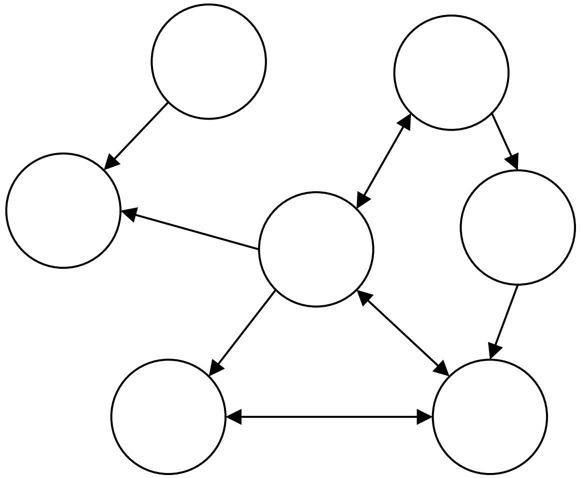


Assume for contradiction that we have a graph with $|E| > n \cdot (n-1)$. This implies that there is some node u with more than $(n-1)$ neighbors. Since there are only n nodes in G, this implies that either u connects to some node twice, or it connects to itself.

Proof by contradiction

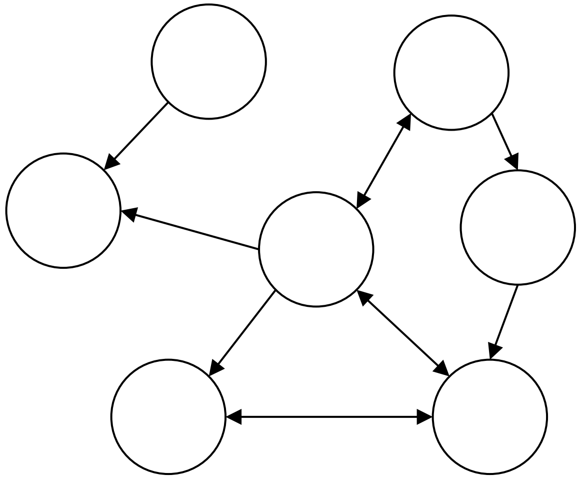
Given a simple directed graph $G = (V, E)$ with $n = |V|$ nodes, we want to prove that the maximum size of the edge set E is $|\mathrm{E}| = n \cdot (n - 1)$.

Assume for contradiction that we have a graph with $|E| > n \cdot (n - 1)$. This implies that there is some node u with more than $(n - 1)$ neighbors. Since there are only n nodes in G, this implies that either u connects to some node twice, or it connects to itself. But we have assumed that G is simple. Contradiction!



Proof by contradiction steps

1. State the claim and all assumptions

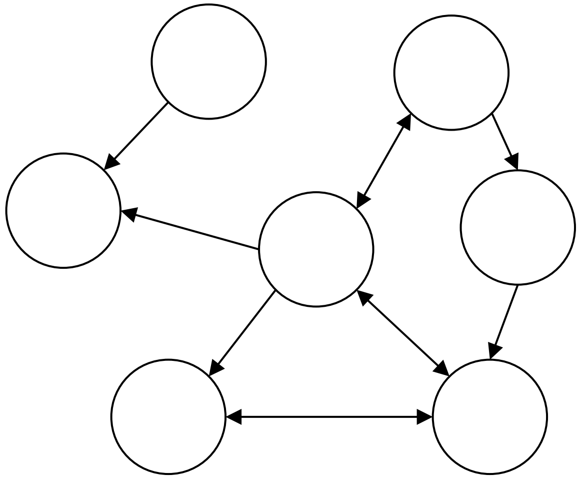


2. Assume we have an example where it is not true

3. Show that this cannot be the case given the assumptions we made

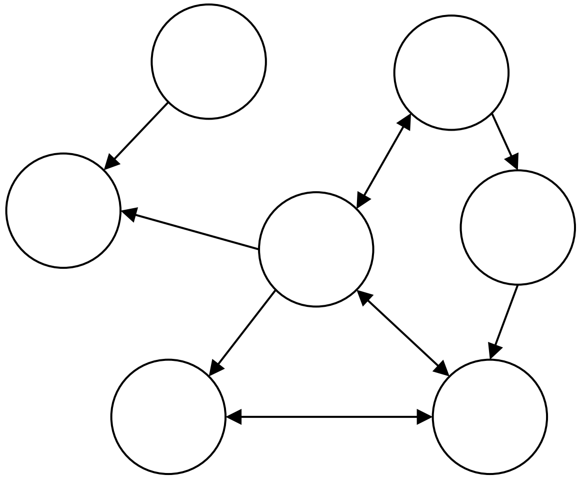
Proof by contradiction steps

1. State the claim and all assumptions
   - Given a **simple directed graph** $G = (V, E)$ with $\boldsymbol{n = |V|}$ **nodes**, we want to prove that **the maximum size of the edge set $\boldsymbol{E}$ is $|\mathbf{E}| = \boldsymbol{n \cdot (n-1)}$**.
2. Assume we have an example where it is not true
3. Show that this cannot be the case given the assumptions we made



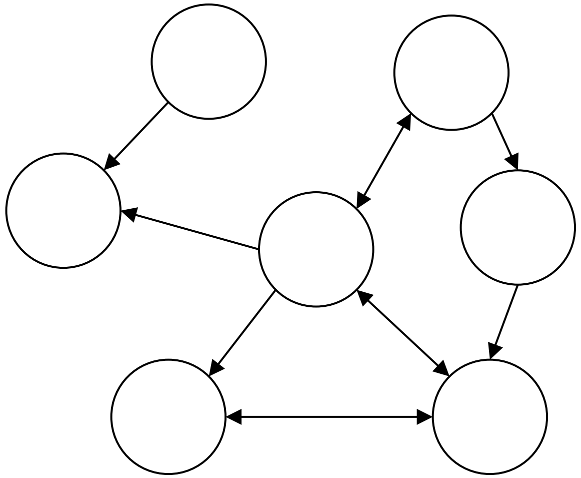
Proof by contradiction steps

1. State the claim and all assumptions
   - Given a **simple directed graph** $G = (V, E)$ with $\boldsymbol{n = |V|}$ **nodes**, we want to prove that **the maximum size of the edge set $\boldsymbol{E}$ is $\boldsymbol{|\mathrm{E}| = n \cdot (n-1)}$**.
2. Assume we have an example where it is not true
   - Assume for contradiction that we have a graph with $\boldsymbol{|E| > n \cdot (n-1)}$.
3. Show that this cannot be the case given the assumptions we made



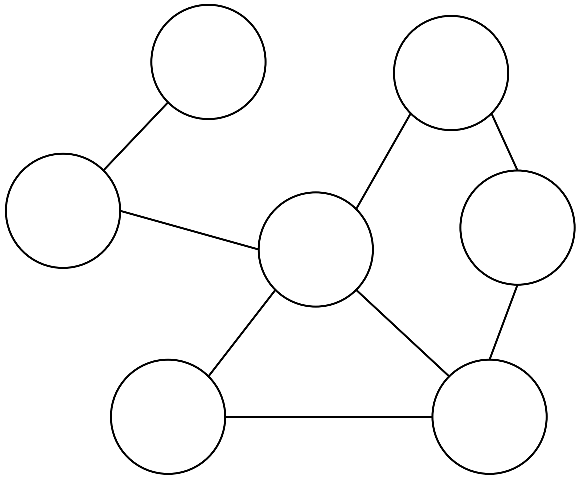
Proof by contradiction steps

1. State the claim and all assumptions
   - Given a **simple directed graph** $G = (V, E)$ with $\boldsymbol{n = |V|}$ **nodes**, we want to prove that **the maximum size of the edge set $\boldsymbol{E}$ is $|\mathbf{E}| = \boldsymbol{n \cdot (n-1)}$**.
2. Assume we have an example where it is not true
   - Assume for contradiction that we have a graph with $|\boldsymbol{E}| > \boldsymbol{n} \cdot (\boldsymbol{n} - \boldsymbol{1})$.
3. Show that this cannot be the case given the assumptions we made
   - This implies that there is some node u with more than $(n-1)$ neighbors…But we have assumed that G is simple. Contradiction!



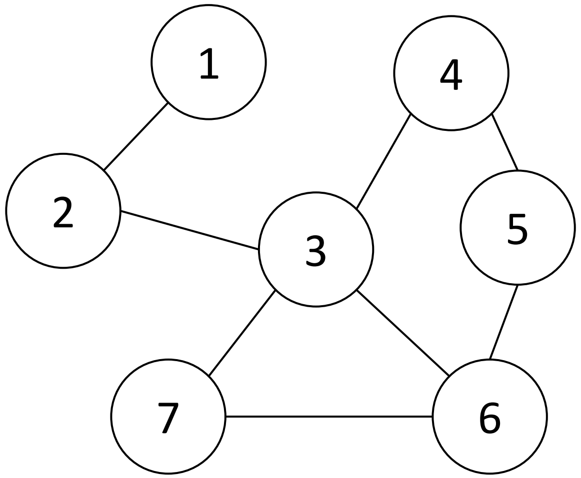
Data Structures for Graphs

Undirected



Data Structures for Graphs

Undirected



Data Structures for Graphs

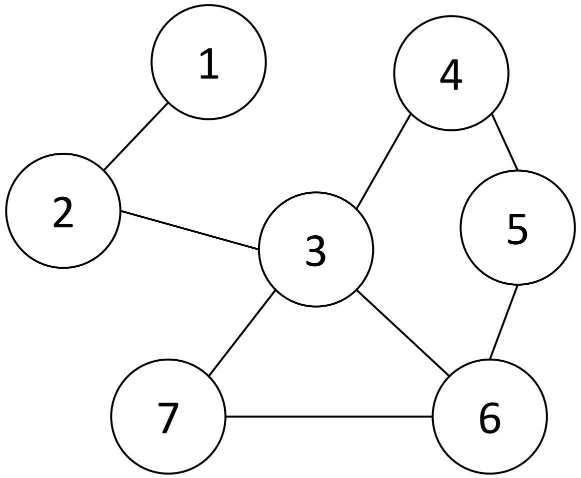
Edgelist: A list of tuples (u, v) representing the edges in a graph G

- Advantage: Very simple to interpret
- Disadvantages:
    - Edge lookup/insertion/deletion is $O(m)$

<u>Edgelist</u>

(1,2)
(2,3)
(3,4)
(4,5)
(5,6)
(6,3)
(6,7)
(7,3)

Undirected



Data Structures for Graphs

Edgelist: A list of tuples (u, v) representing the edges in a graph G

- Advantage: Very simple to interpret
- Disadvantages:
  - Edge lookup/insertion/deletion is $O(m)$

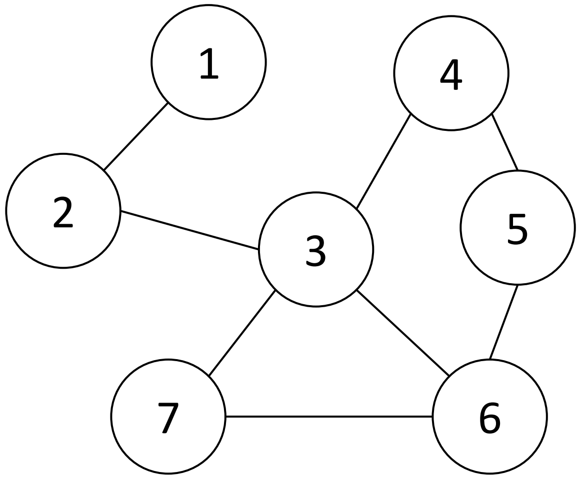
Adjacency List: A list of lists where the first item is a node u and all items in the list are connected to u

- Advantages:
  - Stores same information as edgelist
  - Edge lookup/insertion/deletion can be as fast as $O(n)$
- Disadvantage: stores redundant info for undirected graphs

Edgelist
(1,2)
(2,3)
(3,4)
(4,5)
(5,6)
(6,3)
(6,7)
(7,3)

Adjacency List
[1, 2]
[2, 1, 3]
[3, 4, 6, 7]
[4, 3, 5]
[5, 4, 6]
[6, 3, 5, 7]
[7, 3, 6]

Undirected



Data Structures for Graphs

Edgelist: A list of tuples (u, v) representing the edges in a graph G

- Advantage: Very simple to interpret
- Disadvantages:
  - Edge lookup/insertion/deletion is $O(m)$

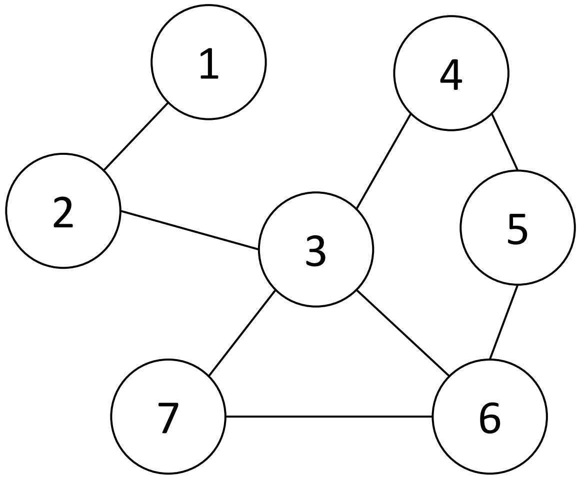
Adjacency List: A list of lists where the first item is a node u and all items in the list are connected to u

- Advantages:
  - Stores same information as edgelist
  - Edge lookup/insertion/deletion can be as fast as $O(n)$
- Disadvantage: stores redundant info for undirected graphs

Adjacency Matrix: A matrix $A[1..n, 1..n]$ where each entry $A[i, j]$ is 1 if an edge exists between nodes i and j and 0 otherwise

- Advantages:
  - Simple way to represent dense graphs (many entries 1)
  - Edge lookup/insertion/deletion is $O(1)$
  - Spectral graph analysis/linear algebraic operations
- Disadvantages:
  - Wastes space when many entries are 0
  - Stores redundant info for undirected graphs

Undirected



<u>Edgelist</u>

(1,2)
(2,3)
(3,4)
(4,5)
(5,6)
(6,3)
(6,7)
(7,3)

<u>Adjacency List</u>

[1, 2]
[2, 1, 3]
[3, 4, 6, 7]
[4, 3, 5]
[5, 4, 6]
[6, 3, 5, 7]
[7, 3, 6]

<u>Adjacency Matrix</u>

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	0	1	0	1	0	1	1
4	0	0	1	0	1	0	0
5	0	0	0	1	0	1	0
6	0	0	1	0	1	0	1
7	0	0	1	0	0	6	0

Data Structures for Graphs

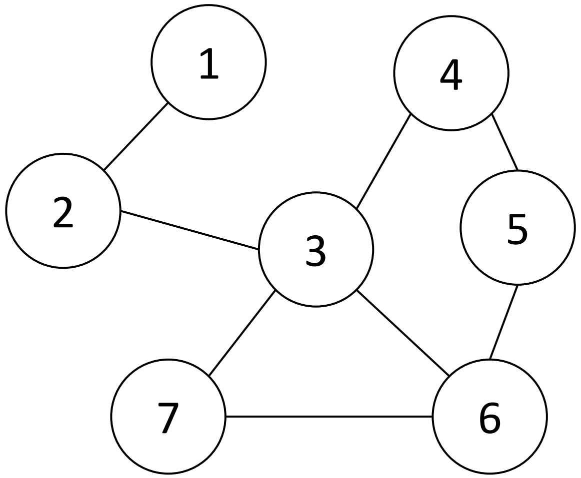
All of these data structures can be modified to make computations faster or more space efficient

- Example: Using a lookup table/dictionary to store an adjacency list would let us return the list of neighbors for a node in $O(1)$ time!

How we store a graph is a choice we make for every algorithm we design

- There is no one size fits all! Different problems will call for different data structures.
- Per Erickson: Usually we don’t need arbitrary edge lookup, so it doesn’t make sense to optimize for that all the time!

Undirected



Edgelist

(1,2)
(2,3)
(3,4)
(4,5)
(5,6)
(6,3)
(6,7)
(7,3)

Adjacency List

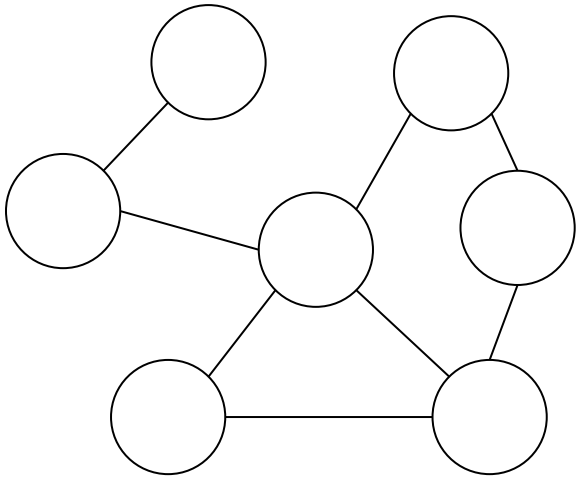
[1, 2]
[2, 1, 3]
[3, 4, 6, 7]
[4, 3, 5]
[5, 4, 6]
[6, 3, 5, 7]
[7, 3, 6]

Adjacency Matrix

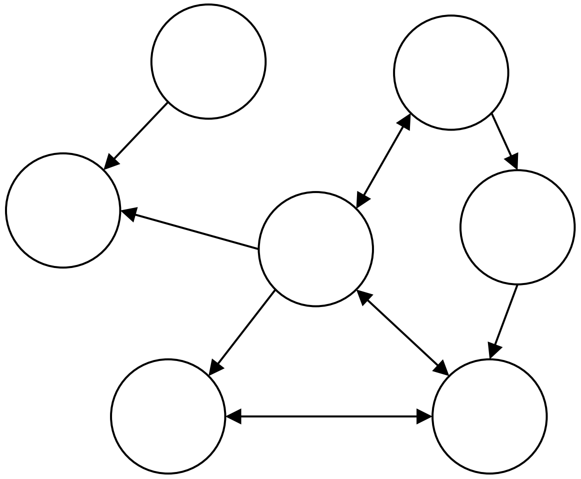
	1	2	3	4	5	6	7
1	0	1	0	0	0	0	0
2	1	0	1	0	0	0	0
3	0	1	0	1	0	1	1
4	0	0	1	0	1	0	0
5	0	0	0	1	0	1	0
6	0	0	1	0	1	0	1
7	0	0	1	0	0	6	0

Paths through graphs

Undirected

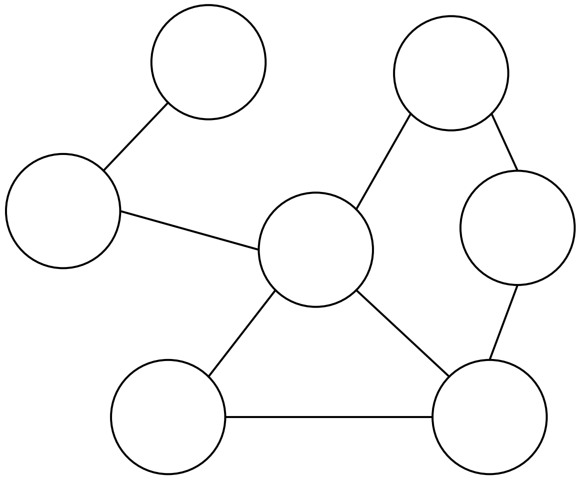


Directed



Paths through graphs

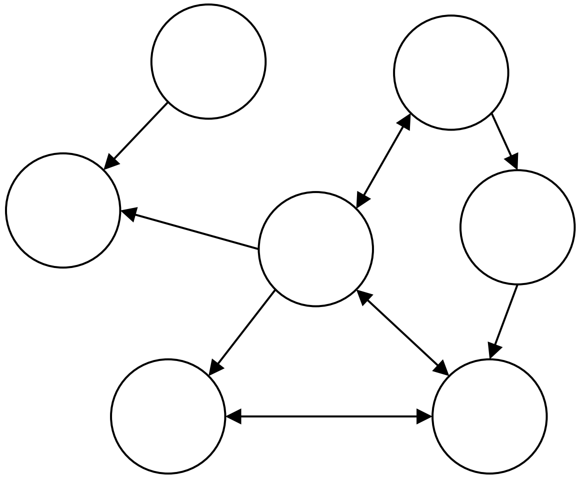
Undirected



A *path* P from vertex v_1 to vertex v_k is an ordered sequence of consecutive edges from E where each node is visited at most once.

$$P = \{(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)\}$$

Directed



A path visiting k nodes has length $k - 1$, since the length is the number of edges traversed.

Paths through graphs

A *path* P from vertex v_1 to vertex v_k is an ordered sequence of consecutive edges from E where each node is visited at most once.

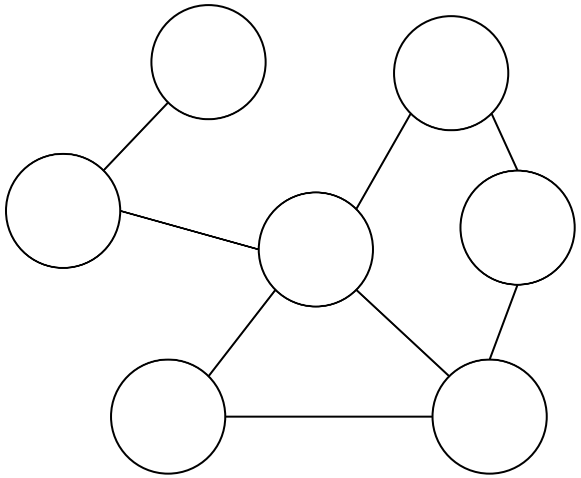
$$P = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$$

A path visiting k nodes has length $k - 1$, since the length is the number of edges traversed.

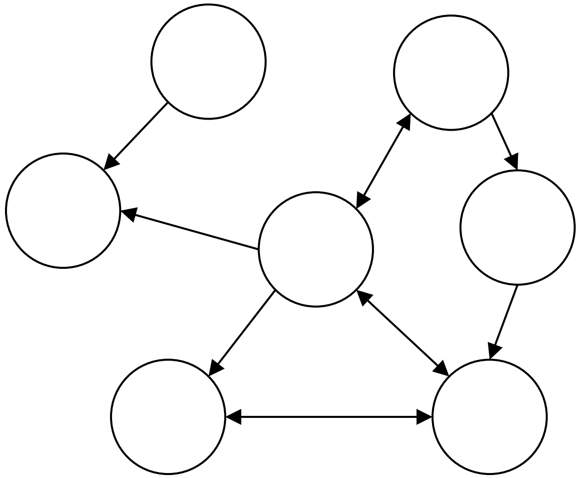
A *walk* through a graph is similar to a path, but nodes can be visited more than once. A walk is *closed* if it starts and ends with the same node; otherwise it is called *open*.

A *cycle* is a closed walk that visits any node except the first at most once.

Undirected



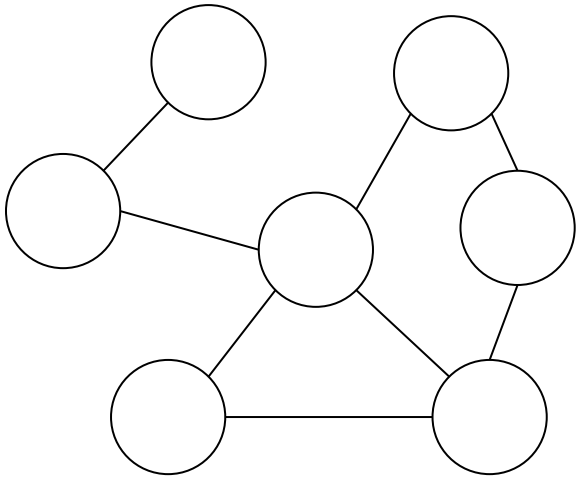
Directed



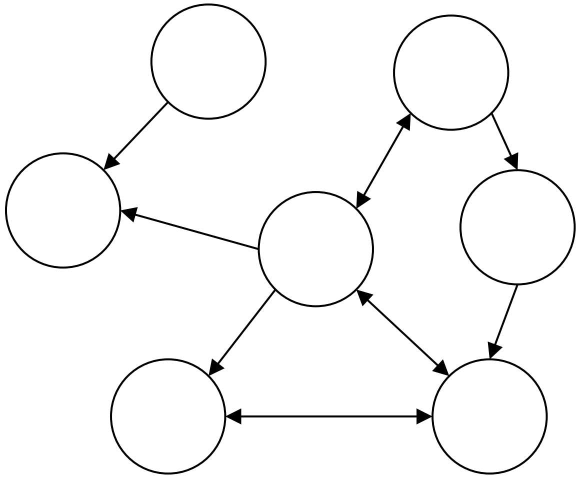
Reachability & Connectivity

A node v is *reachable* from a node u if there is a path from u to v.

Undirected



Directed

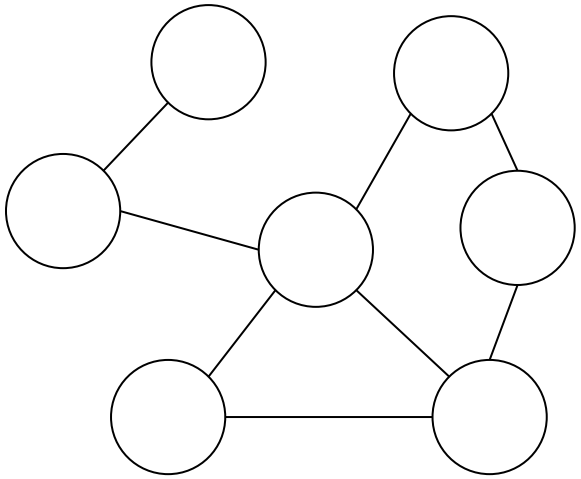


Reachability & Connectivity

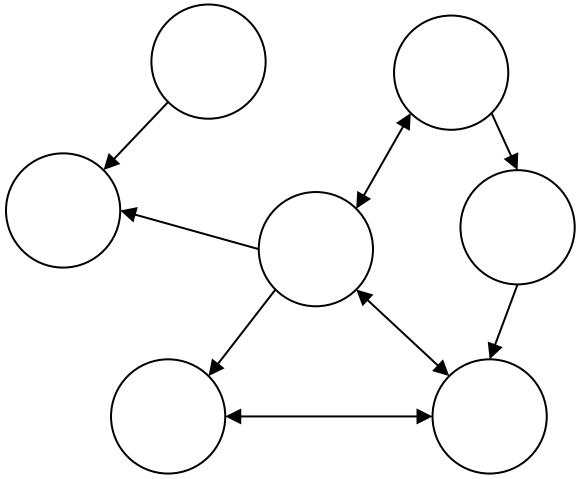
A node v is *reachable* from a node u if there is a path from u to v.

A graph $G = (V, E)$ is *connected* if for every pair of nodes u, v, the node v is reachable from u.

Undirected



Directed



Reachability & Connectivity

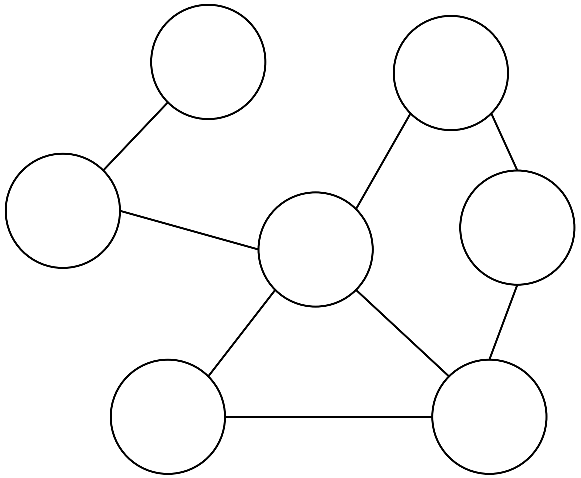
A node v is *reachable* from a node u if there is a path from u to v.

A graph $G = (V, E)$ is *connected* if for every pair of nodes u, v, the node v is reachable from u.

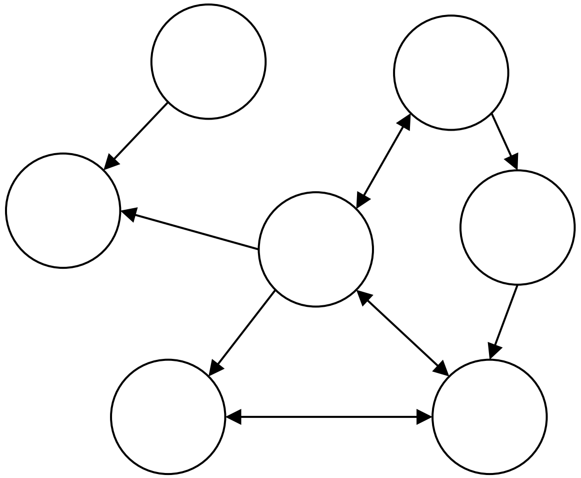
In a directed graph, we have two types of connectivity:

- Strongly Connected: there is a path both from u to v and from v to u.
- Weakly Connected: there is a path either from u to v or from v to u

Undirected



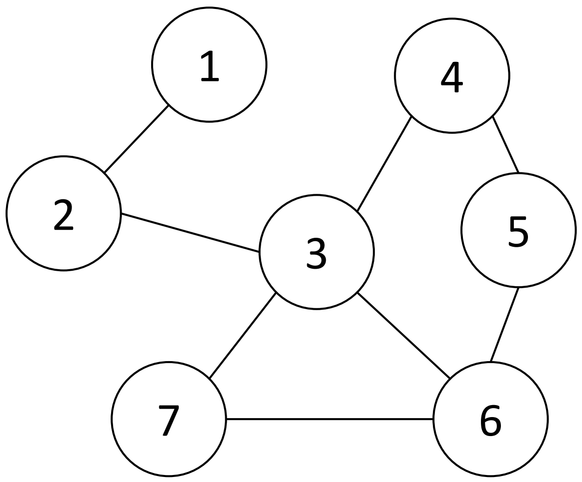
Directed



Exploring a graph: Reachability

Assume we have an undirected graph $G = (V, E)$ and we want to determine whether the graph is connected.

We need an algorithm that will tell us whether every node is reachable from every other node.



Idea: traverse the graph edge by edge.

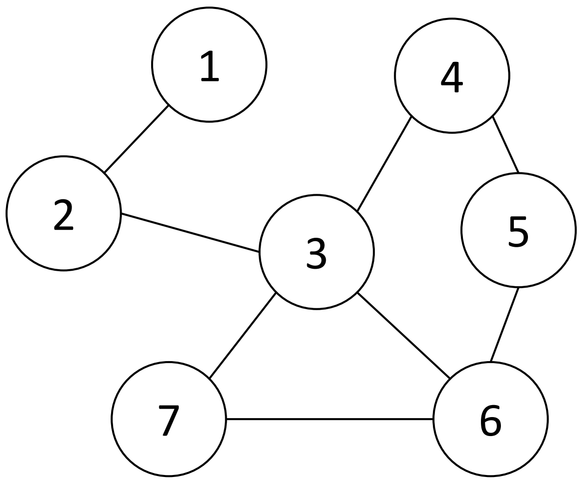
If we can reach every node without restarting, we know the graph is connected!

Exploring a graph: Breadth First Search

```
BFS(G = (V,E)):
  Q ← empty queue
  visited ← ∅

  Append node 1 to Q
  While Q is not empty:
    u ← next node in Q
    For v ∈ Neighbors(u):
      if v ∉ visited
        Append v to Q
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```



Idea: traverse the graph edge by edge.

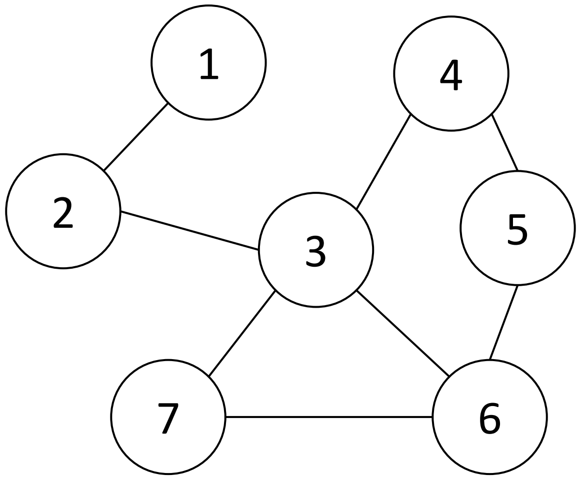
If we can reach every node without restarting, we know the graph is connected!

Breadth First Search Running time

```
BFS(G = (V,E)):
  Q ← empty queue
  visited ← ∅

  Append node 1 to Q
  While Q is not empty:
    u ← next node in Q
    For v ∈ Neighbors(u):
      if v ∉ visited
        Append v to Q
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```



By definition, we visit every node once, so we immediately have $O(n)$ to start.

At each node, we check if each of its neighbors has been visited already.

Observation: this is the same as visiting every *edge*! Thus we also have $O(m)$.

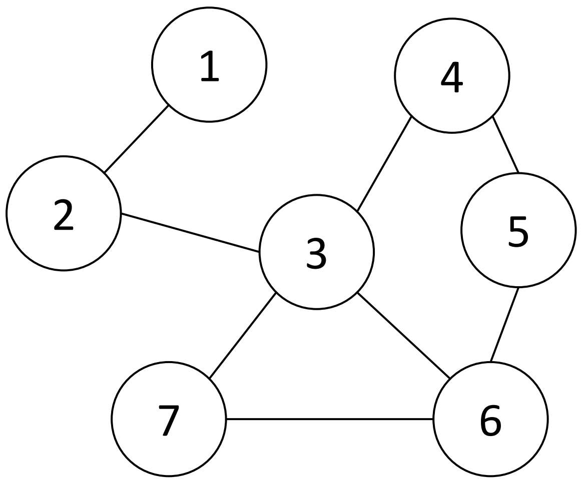
Therefore, the running time of BFS is $O(n + m)$.

Exploring a graph: Depth First Search

```
DFS(G = (V,E)):
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```



Idea: traverse the graph edge by edge.

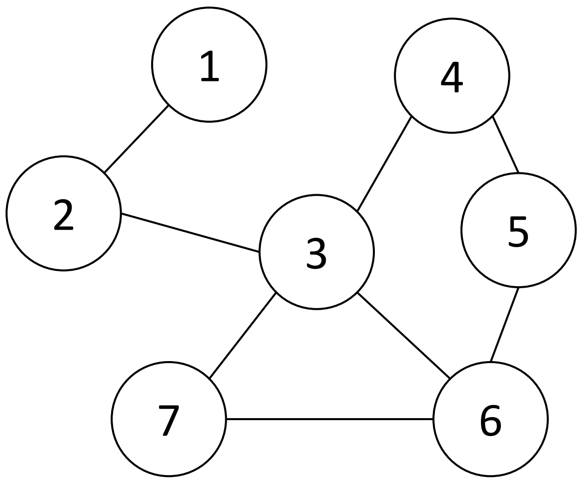
If we can reach every node without restarting, we know the graph is connected!

Depth First Search Running time

```
DFS(G = (V,E)):
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```

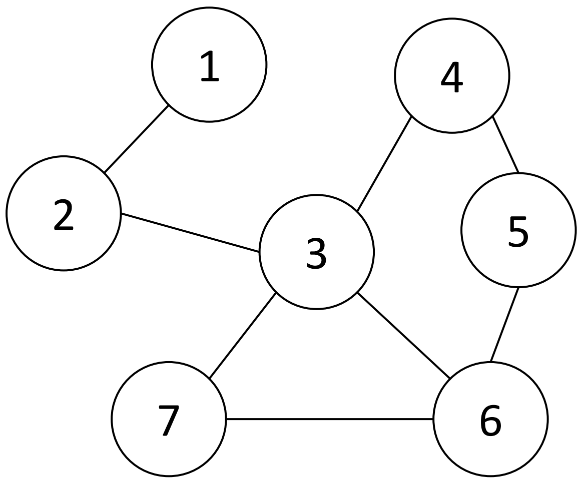


Depth First Search Running time

```
DFS(G = (V,E)):
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```



Same argument as BFS!

$O(n+m)$

Note: These algorithms have recursive equivalents!

```
DFS(G = (V, E)):
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```

Our iterative DFS really just makes the recursive stack explicit!

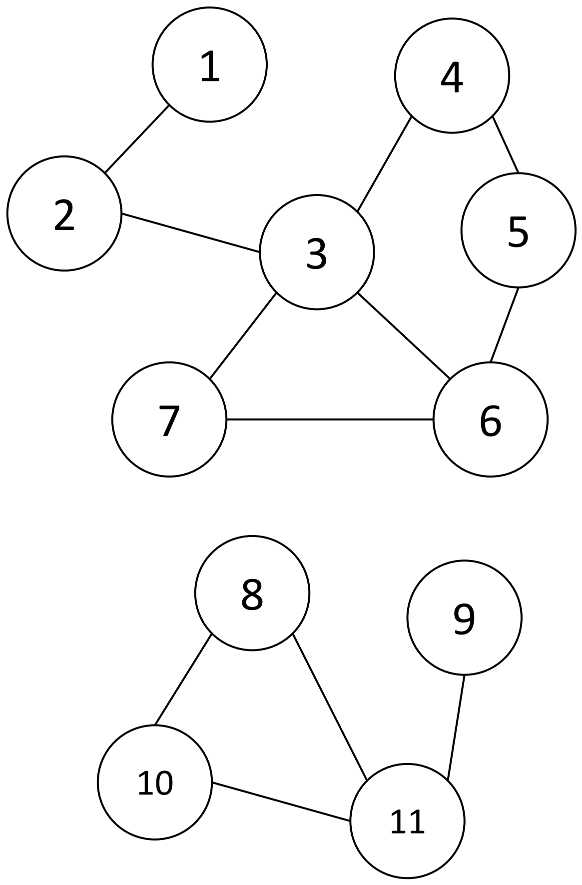
```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

Exploring Connected Components

Subgraphs & Components

A graph $G' = (V', E')$ is a *subgraph* of another graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A graph is trivially a subgraph of itself. We usually exclude this case and unless otherwise specified we mean *proper* subgraphs.



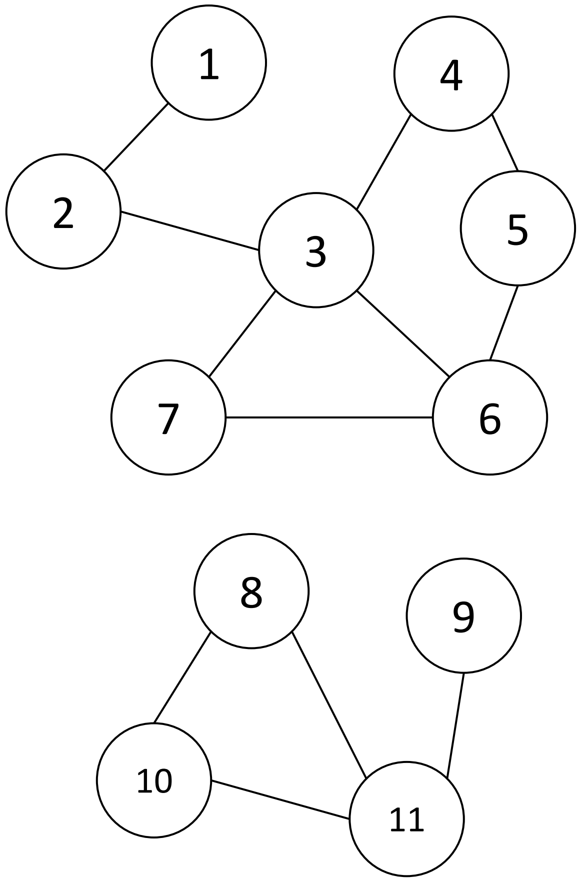
Subgraphs & Components

Every graph is made up of 1 or more *components*, which are maximal connected subgraphs.

Two nodes are in the same component if they are mutually reachable.

Nodes are in different components if they cannot be reached from one another.

We can use our exploration algorithms to find connected components!

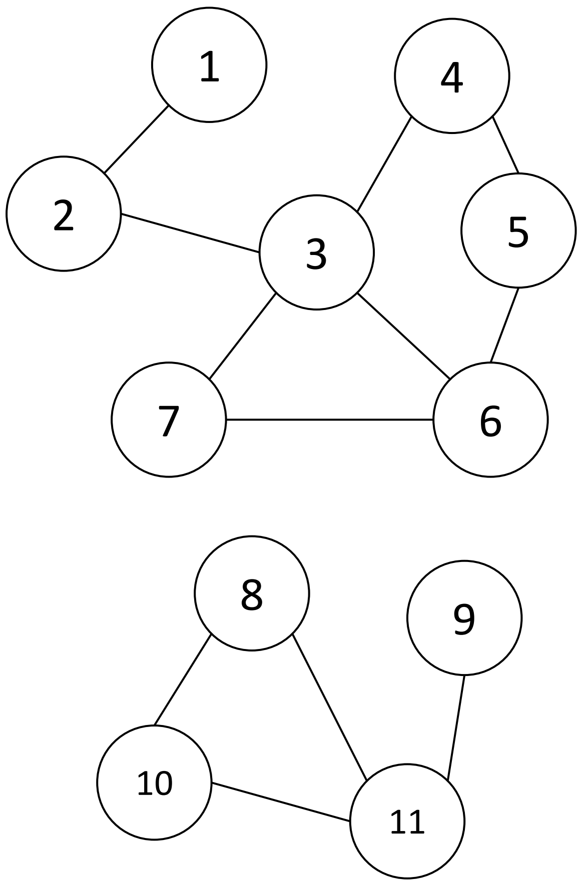


Finding Undirected Components with DFS

```
ComponentsDFS(G = (V, E)):
  component[v] = -1 For all v ∈ V
  comp = 1
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  component[1] = comp
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
        component[v] = comp
    Add u to visited

    If S is empty AND |visited| < |V|:
      Choose a node v ∈ V −visited
      Push v onto S
      comp = comp+1
      component[v] = comp
```

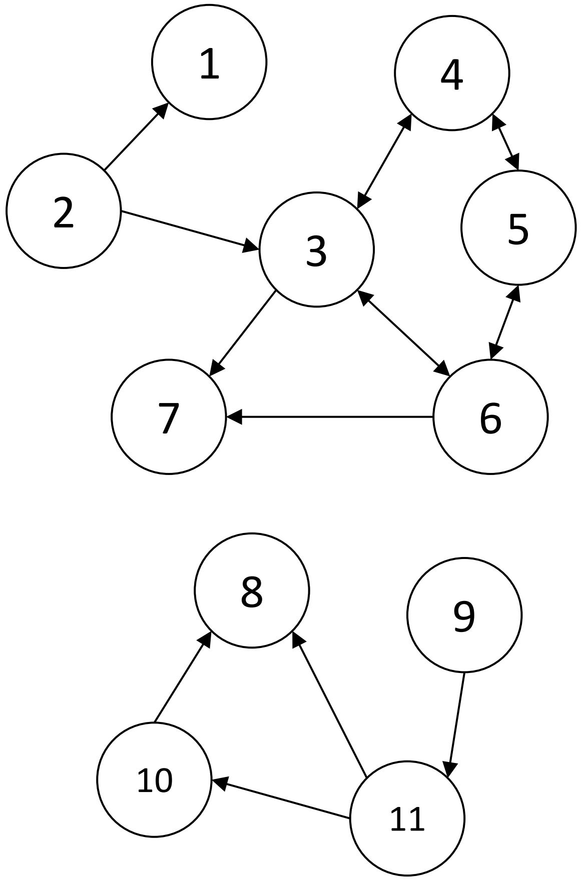


What about directed graphs?

Recall: Two types of connected components in directed graphs

1. Weakly connected: for every pair (u, v), at least one node is reachable from the other.

2. Strongly connected: for every pair (u, v), both nodes are reachable from the other



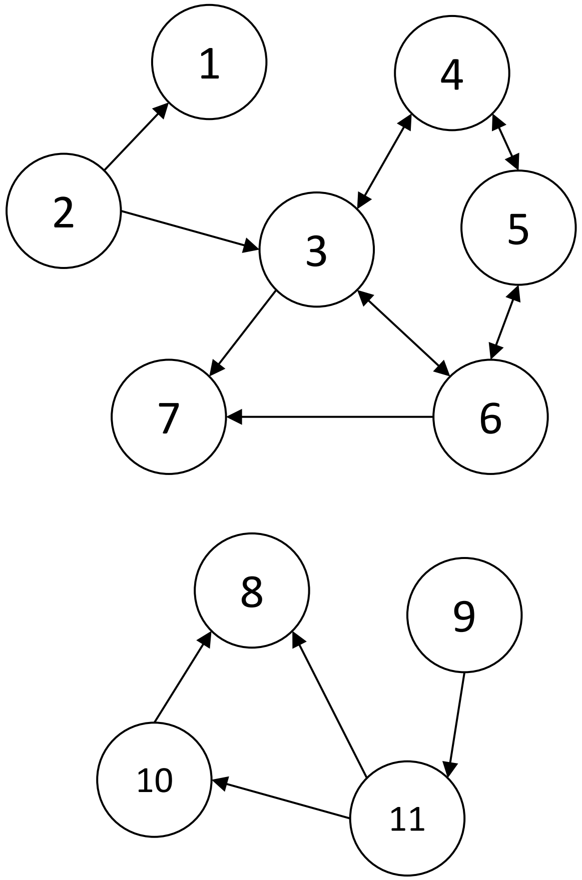
Finding Strongly Connected Components

```
SCC(G = (V,E)):
  Let G^R be G with all edges "reversed"

  Let comp[u] ← −1 for all u
  Let c ← 0

  For u from 1..n:
    if comp[u] = -1:
      Let S be the nodes found by DFS(G,u)
      Let T be the nodes found by DFS(G^R,u)
      // intersection of S and T is a SCC!
      label comp[v]= c for all v ∈ S ∩ T
      let c ← c + 1

  Return comp
```



Pause: What have we done so far?

We defined two graph traversal algorithms that can help us determine *reachability* between nodes and overall *connectivity* of a graph

- DFS: Stack based algorithm
- BFS: Queue based algorithm
- Both can be written either recursively or iteratively

We showed how we can use these algorithms to discover the *components* of a graph

- In undirected graphs, it is enough to just run our traversal algorithm until every node is visited once, assigning to a new component every time we "run out" of nodes
- In directed graphs, we need to check both directions to get *strongly connected* components.
    - We achieve this by cleverly running DFS from the same node twice, first on the input graph as usual, then on the graph with reversed edges. The intersection of the reachable sets for these two DFS calls is a strongly connected component!

Typology of Edges in DFS

For every node discovered during a DFS execution, we can keep track of its parent.

Typology of Edges in DFS

For every node discovered during a DFS execution, we can keep track of its parent.

```
DFS(G = (V,E)):
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
        parent[v] ← u
    Add u to visited

  If |visited| = |V|:
    return True
  Else:
    return False
```

Typology of Edges in DFS

For every node discovered during a DFS execution, we can keep track of its parent.

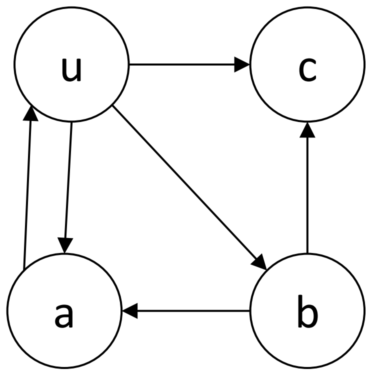
The graph of the parent-child relationships is a tree where each edge can be assigned to one of four types:

Tree edge:
- Explore new nodes

Forward edge:
- Ancestor to descendant

Backward edge:
- Descendant to ancestor

Cross edges:
- No ancestral relationship



Typology of Edges in DFS

For every node discovered during a DFS execution, we can keep track of its parent.

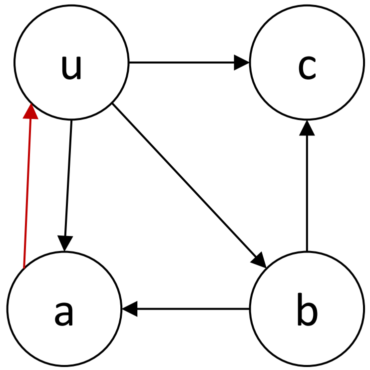
The graph of the parent-child relationships is a tree where each edge can be assigned to one of four types:

Tree edge: $(u, a), (u, b), (b, c)$
- Explore new nodes

Forward edge: (u, c)
- Ancestor to descendant

Backward edge: (a, u)
- Descendant to ancestor

Cross edges: (b, a)
- No ancestral relationship



Backwards edges identify *cycles* in the graph!

A cycle is a closed walk (starts and ends at the same vertex) that visits each vertex in the walk at most once.

Post-Order

A *post-ordering* of a graph $G = (V, E)$ is an ordering of the nodes based on when they were marked visited by DFS.

To get a post-order, we maintain a global clock variable that is initialized to 1.

Every time we add a node to the visited set, we set its post-order value to the current value of clock, then increment clock.

```
DFS(G = (V,E)):
  S ← empty stack
  visited ← ∅

  Push node 1 onto S
  While S is not empty:
    u ← pop from S
    For v ∈ Neighbors(u):
      if v ∉ visited:
        Push v onto S
    Add u to visited
    post-visit(u)
```

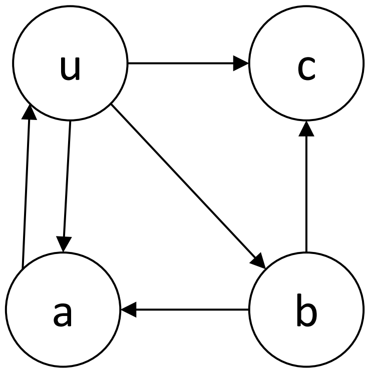
```
post-visit(u):
  set postorder[u] = clock
  clock ← clock + 1
```

Post-Order

A *post-ordering* of a graph $G = (V, E)$ is an ordering of the nodes based on when they were marked visited by DFS.

To get a post-order, we maintain a global clock variable that is initialized to 1.

Every time we add a node to the visited set, we set its post-order value to the current value of clock, then increment clock.



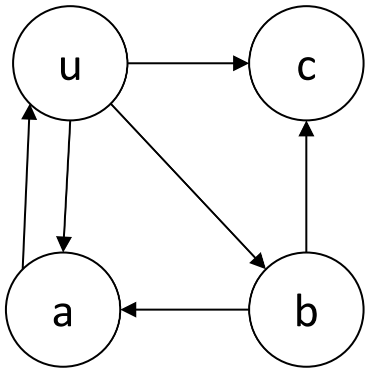
Vertex	u	a	b	c
Postorder				

Post-Order

A *post-ordering* of a graph $G = (V, E)$ is an ordering of the nodes based on when they were marked visited by DFS.

To get a post-order, we maintain a global clock variable that is initialized to 1.

Every time we add a node to the visited set, we set its post-order value to the current value of clock, then increment clock.

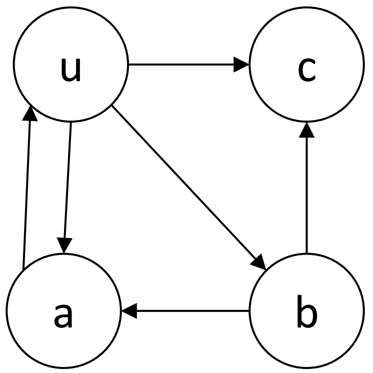


Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why?

- DFS(u) can’t finish until its children are finished
- If postorder[u] < postorder[v], then DFS(u) finishes before DFS(v), meaning DFS(v) was not called by DFS(u)
- For this situation to arise, when we ran DFS(u), we must have had $v \in$ visited, implying DFS(v) ran first
- Which means DFS(v) started first but finished after DFS(u), which can only happen for a backwards edge!

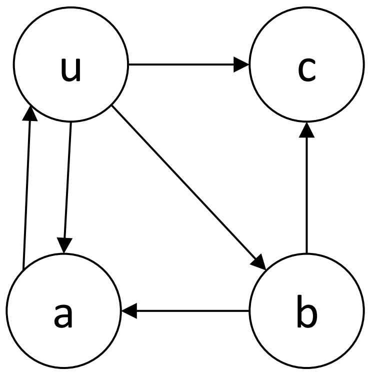


Vertex	u	a	b	c
Postorder	4	1	3	2

Post-Order

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why?

- DFS(u) can't finish until its children are finished
- If postorder[u] < postorder[v], then DFS(u) finishes before DFS(v), meaning DFS(v) was not called by DFS(u)
- For this situation to arise, when we ran DFS(u), we must have had $v \in$ visited, implying DFS(v) ran first
- Which means DFS(v) started first but finished after DFS(u), which can only happen for a backwards edge!



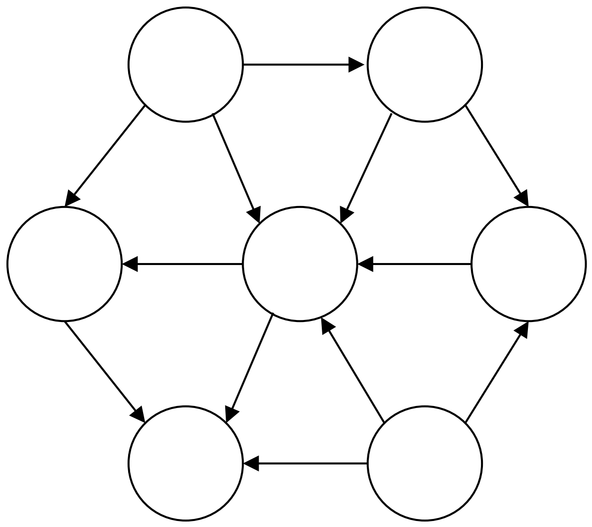
In our example, (u,v) is (a,u) and postorder[a] < postorder[u]!

Vertex	u	a	b	c
Postorder	4	1	3	2

Directed Acyclic Graphs and Topological Ordering

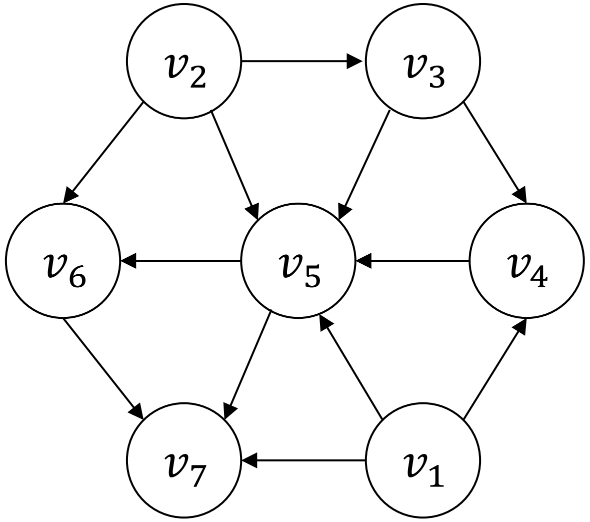
Directed Acyclic Graph (DAG)

- A directed graph with no cycles
- Represent precedence relationships
  - “this” comes before “that”
  - “this” is prior to “that”

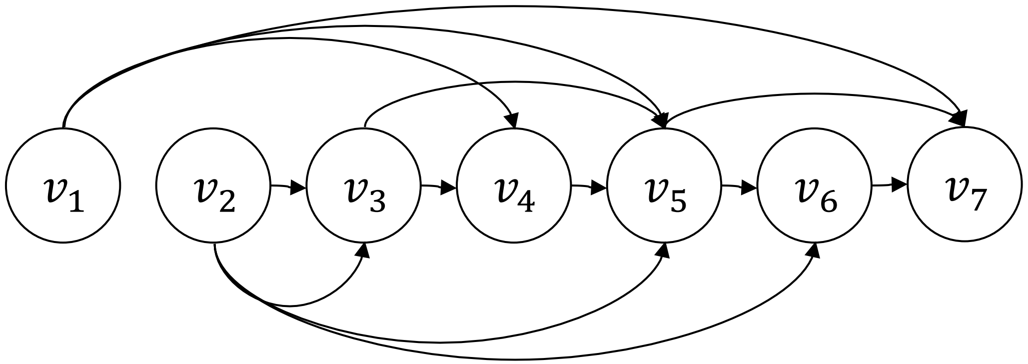


Directed Acyclic Graph (DAG)

- A directed graph with no cycles
- Represent precedence relationships
  - “this” comes before “that”
  - “this” is prior to “that”



A *topological ordering* of a directed graph is a labeling of the nodes so that all edges point “forward”, meaning for all directed edges $(v_i, v_j), j > i$



Claim: If G has a topological ordering it is a DAG

Two problems in one

Problem 1: Is G a DAG?

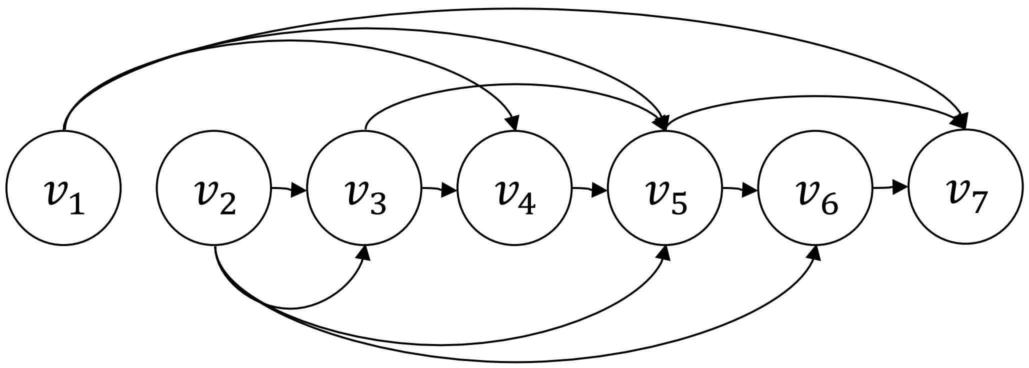
Problem 2: Given a directed graph, can it be topologically ordered?

Claim: G has a topological ordering if and only if G is a DAG

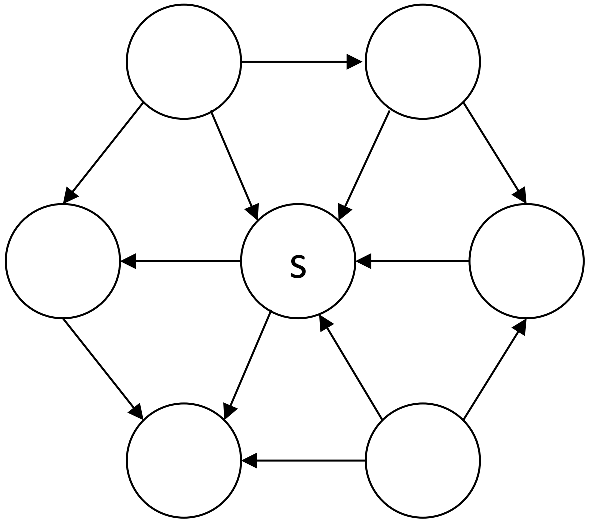
- We will design an algorithm that either outputs a topological ordering or that the graph is not a DAG

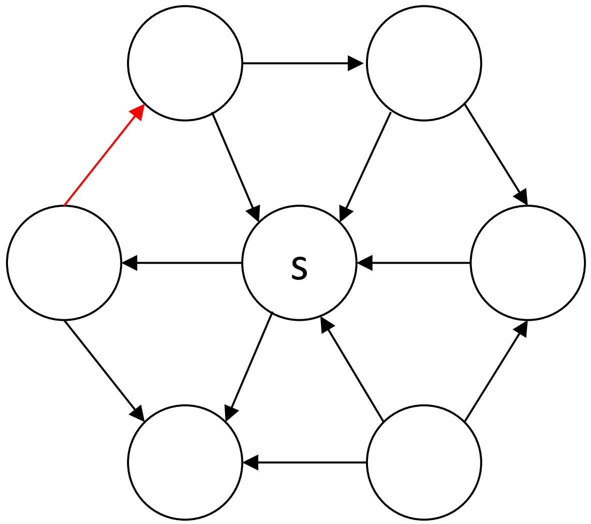
Two problems in one

Observation:
In a topological ordering, there is a node with no incoming edges!



Observation:
In a DAG, there is a node with no incoming edges!





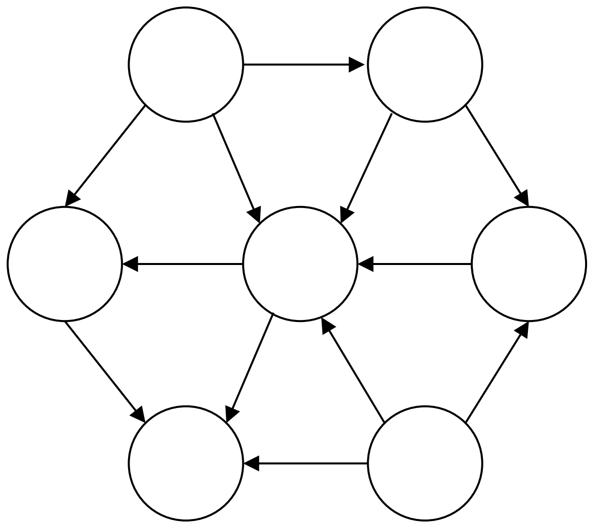
Check by following incoming links backwards until you find a node that has none, or you find a cycle.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.

We can prove this by induction on n.



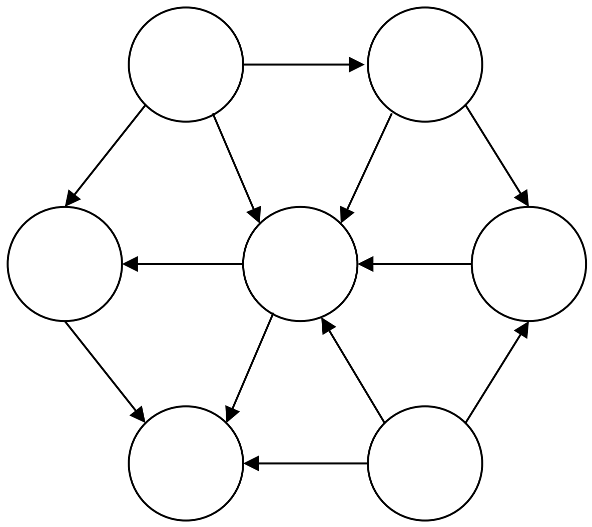
Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.

We can prove this by induction on n.

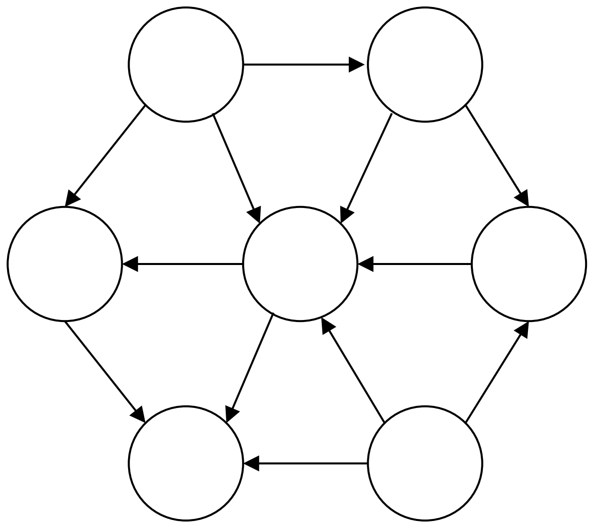
Base case: $n = 1$; trivially true



Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

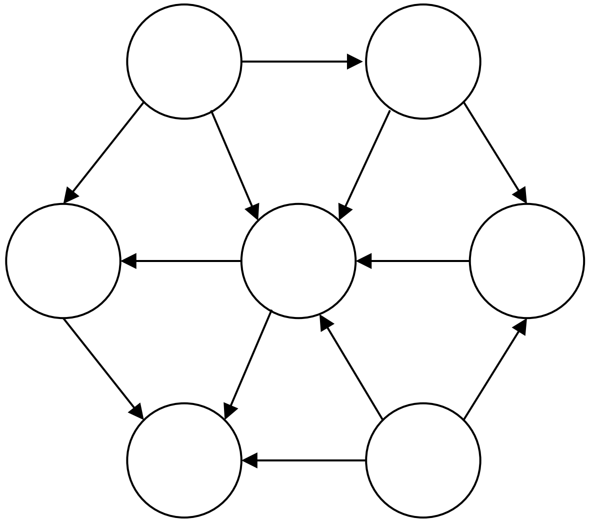
Base case: $n = 1$; trivially true

Inductive step:
- Assume topological ordering exists for DAGs up to n nodes.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

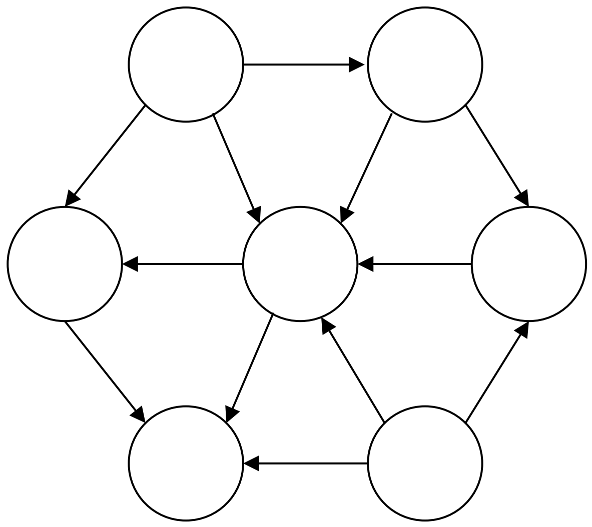
Base case: $n = 1$; trivially true

Inductive step:
- Assume topological ordering exists for DAGS up to n nodes.
- Given a dag on $n + 1$ nodes, identify a node with no incoming edges. We know at least one exists.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

Base case: $n = 1$; trivially true

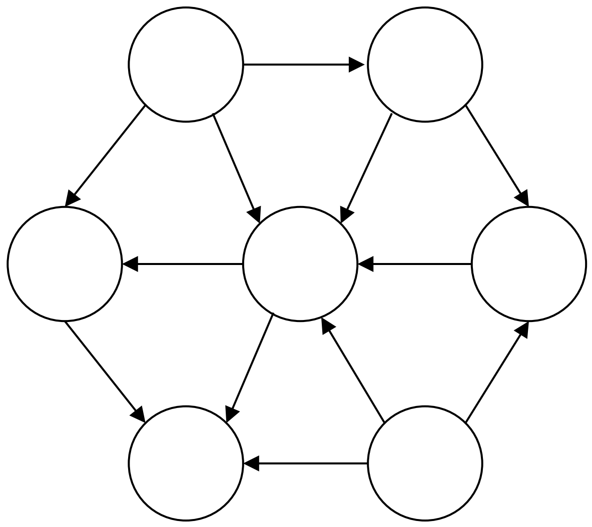
Inductive step:

- Assume topological ordering exists for DAGS up to n nodes.
- Given a dag on $n + 1$ nodes, identify a node with no incoming edges. We know at least one exists.
- Remove this node, and the remaining DAG on n nodes has a topological ordering by the inductive hypothesis.

Does every DAG have a node with no incoming edges?

Claim: For every DAG on $n \in \mathbb{N}$ nodes, there is a topological ordering.

Lemma (from previous slide): Every DAG has a node with no incoming edges.



We can prove this by induction on n.

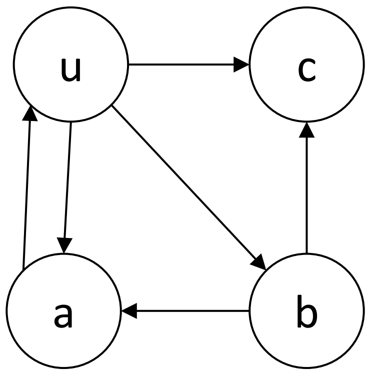
Base case: $n = 1$; trivially true

Inductive step:
- Assume topological ordering exists for DAGS up to n nodes.
- Given a dag on $n + 1$ nodes, identify a node with no incoming edges. We know at least one exists.
- Remove this node, and the remaining DAG on n nodes has a topological ordering by the inductive hypothesis.
- Since the node we removed has no incoming edges, it can be trivially added to the beginning of the ordering. Hence the claim.

Reminder: Post-ordering identifies backwards edges!

Observation: If postorder[u] < postorder[v], then (u, v) is a backwards edge! Why?

- DFS(u) can't finish until its children are finished
- If postorder[u] < postorder[v], then DFS(u) finishes before DFS(v), meaning DFS(v) was not called by DFS(u)
- For this situation to arise, when we ran DFS(u), we must have had $v \in$ visited, implying DFS(v) ran first
- Which means DFS(v) started first but finished after DFS(u), which can only happen for a backwards edge!



In our example, (u,v) is (a,u) and postorder[a] < postorder[u]!

Vertex	u	a	b	c
Postorder	4	1	3	2

Topological Orderings

Claim: Ordering nodes by decreasing post-order gives a topological ordering.

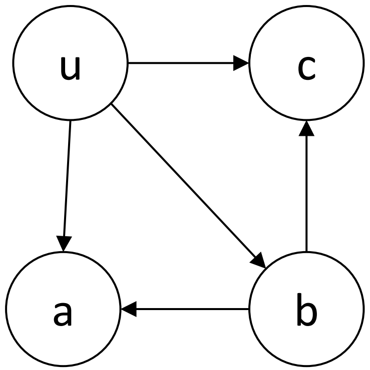
Proof:

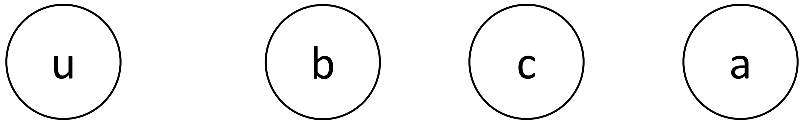
- We know that a DAG has no backward edges, since backward edges imply the presence of cycles.
- Suppose the decreasing post-ordering is not a topological ordering
  - There must be an edge (u,v) such that postorder[u] < postorder[v]
  - But such an edge would be a backward edge, implying a cycle
  - We showed such an edge can’t exist in a DAG. Contradiction!

Topological Orderings

Claim: Ordering nodes by decreasing post-order gives a topological ordering.

Example:





Vertex	u	a	b	c
Postorder	4	1	3	2

Topological Ordering Wrap

A DAG is a directed graph with no cycles.

For any DAG, we can find a topological ordering in $O(n+m)$ time using DFS, since a reverse post-ordering is a topological ordering.

If we are not sure our input graph is a DAG, we can still use DFS to identify backwards edges in the DFS tree, which imply cycles.

```
TopologicalOrdering(G = (V,E)):
  Run DFS(G) with post-order
  If ∃_{u,v} s.t. postorder[u] < postorder[v]:
    Return False
  Else:
    Return reverse(postorder)
```

Much more to come on graphs!

Tomorrow: Shortest paths and betweenness centrality

Suggested Reading assignment: Erickson up through Chapter 8.5

Homework 3 will be out shortly after class. Get started early!

Midterm grades coming later this week, please be patient!