GRAPHS

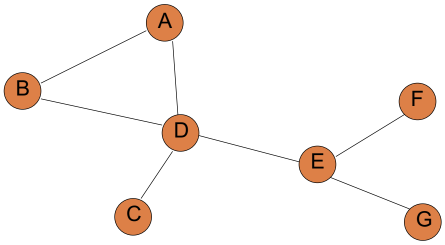
David Kauchak
CS 140 – Spring 2023

Admin

Assignment 8

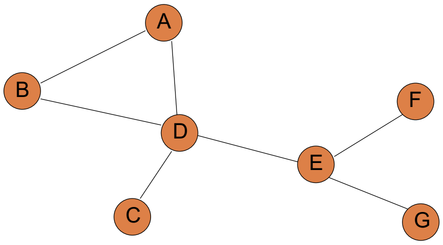
Graphs

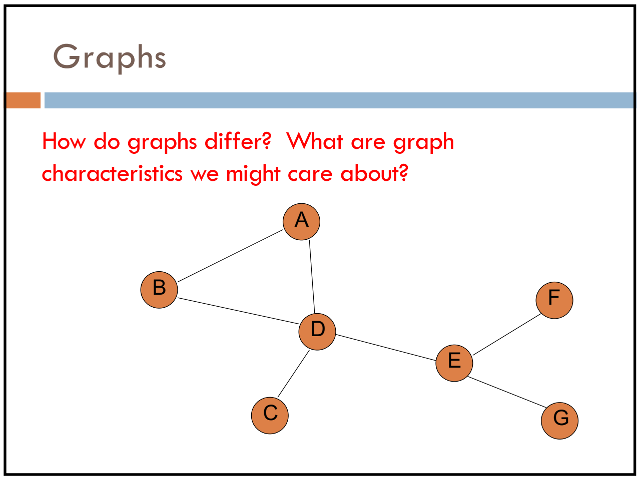
What is a graph?



Graphs

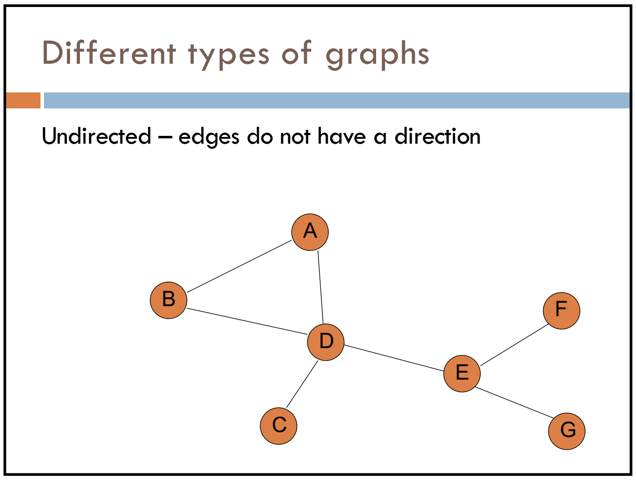
A graph is a set of vertices V and a set of edges (u,v) $\in$ E where u,v $\in$ V





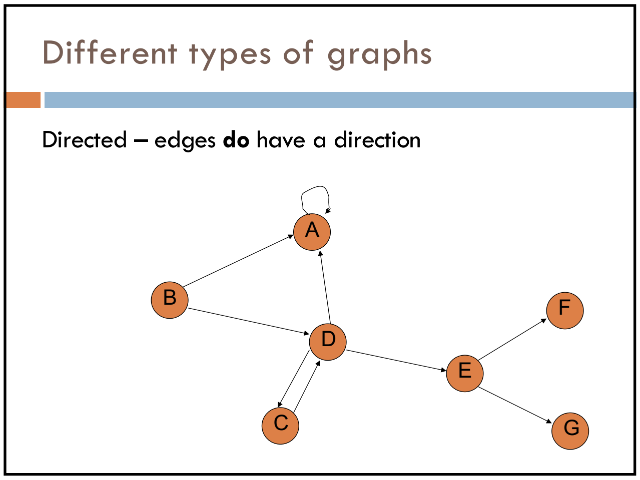
Graphs

How do graphs differ? What are graph characteristics we might care about?



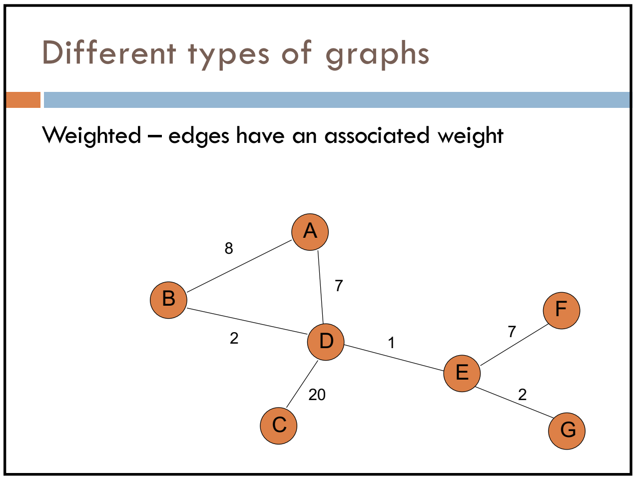
Different types of graphs

Undirected – edges do not have a direction



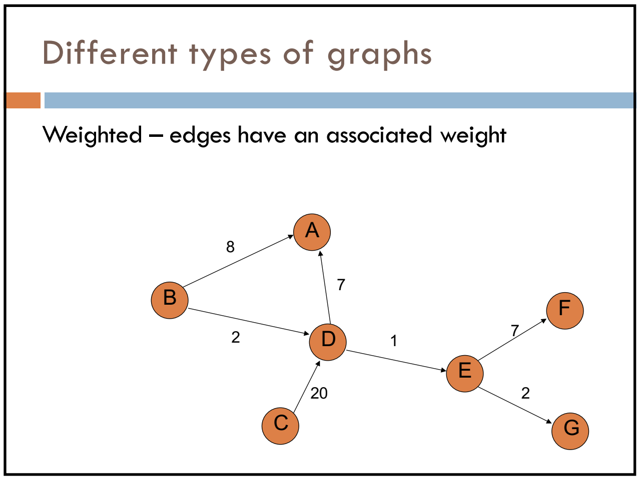
Different types of graphs

Directed – edges **do** have a direction



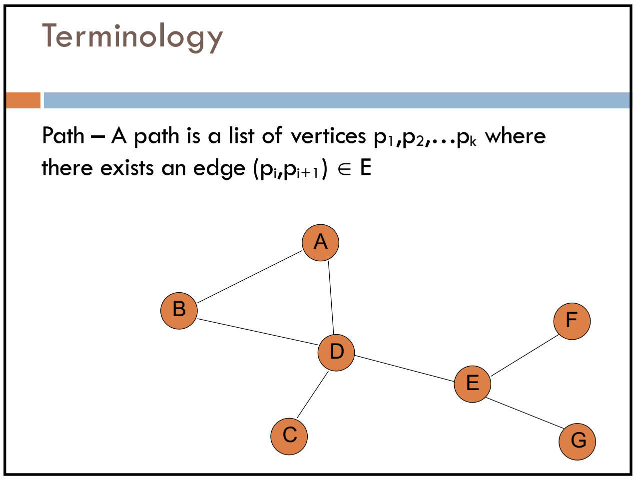
Different types of graphs

Weighted – edges have an associated weight



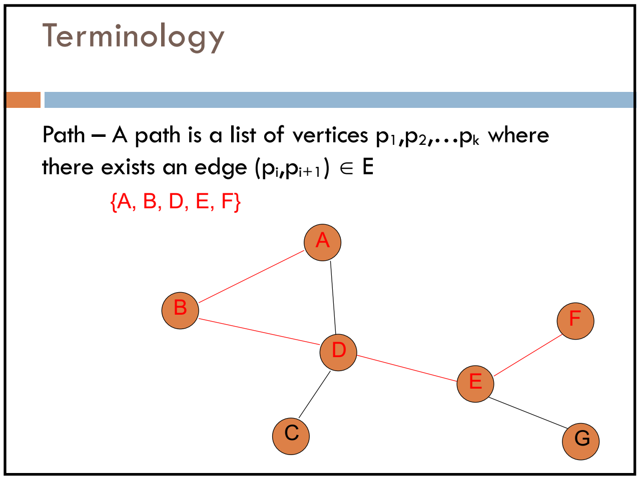
Different types of graphs

Weighted – edges have an associated weight



Terminology

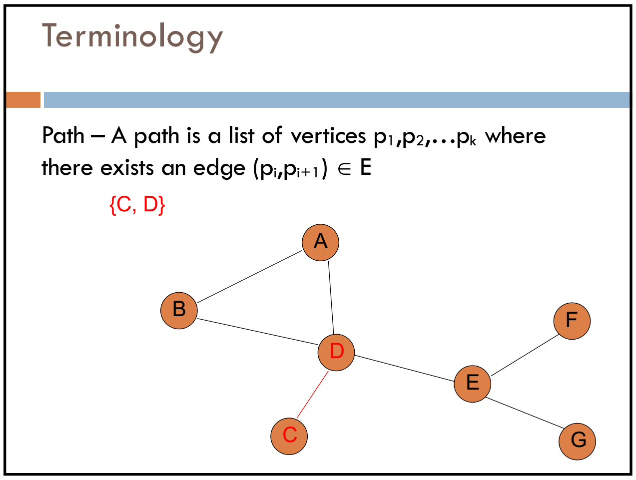
Path – A path is a list of vertices $p_1,p_2,\ldots p_k$ where there exists an edge $(p_i,p_{i+1}) \in E$



Terminology

Path – A path is a list of vertices $p_1,p_2,\ldots p_k$ where there exists an edge $(p_i,p_{i+1}) \in E$

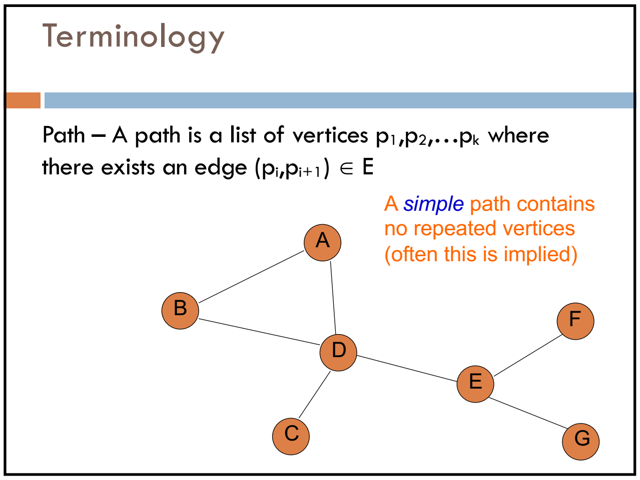
{A, B, D, E, F}



Terminology

Path – A path is a list of vertices $p_1,p_2,\ldots p_k$ where there exists an edge $(p_i,p_{i+1}) \in E$

{C, D}

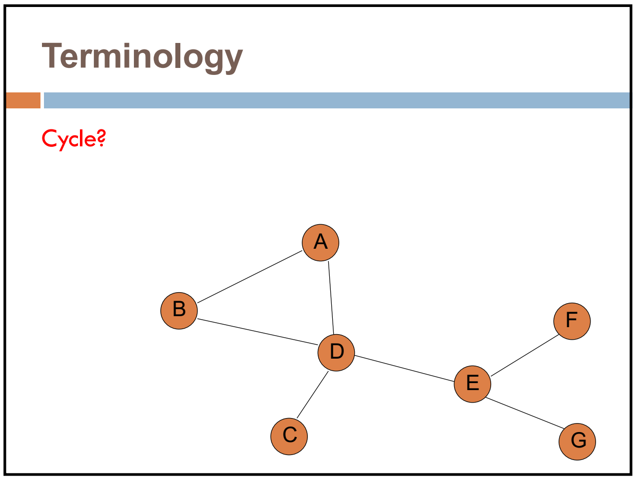


Terminology

Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$

A *simple* path contains no repeated vertices (often this is implied)

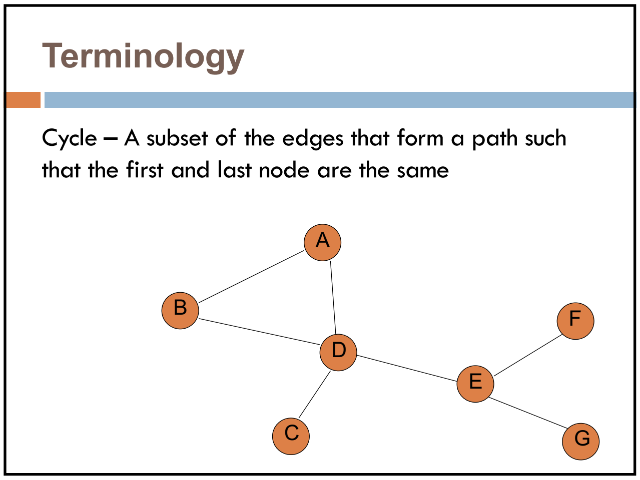
14



Terminology

Cycle?

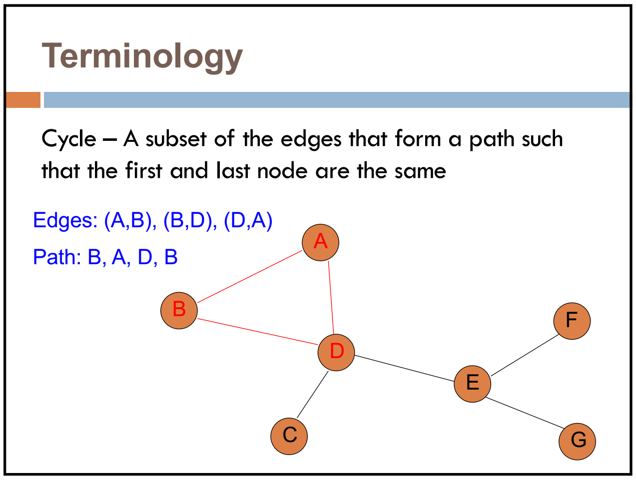
15



Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

16



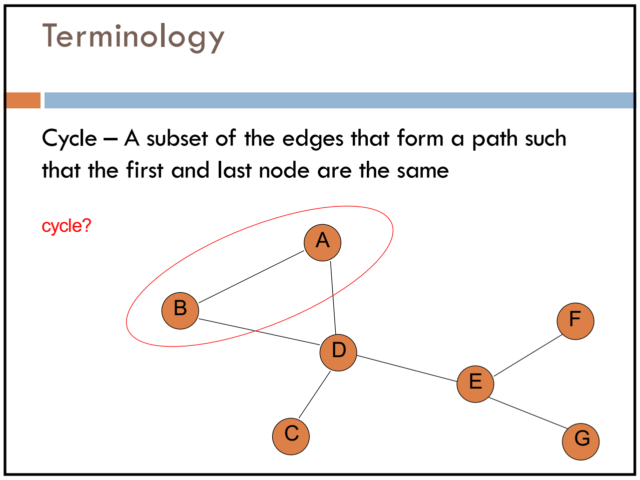
Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

Edges: (A,B), (B,D), (D,A)

Path: B, A, D, B

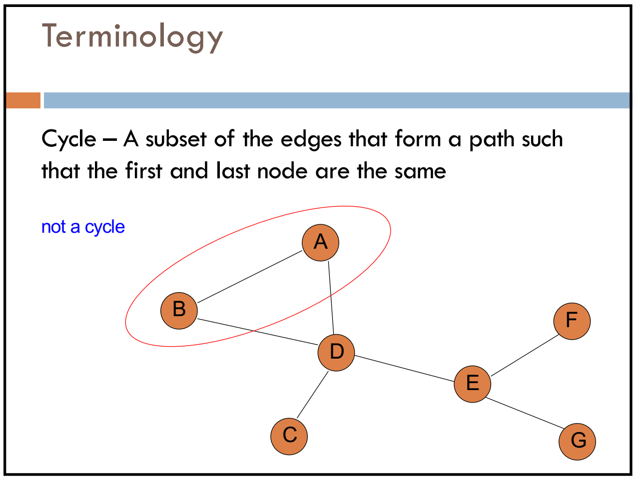
17



Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

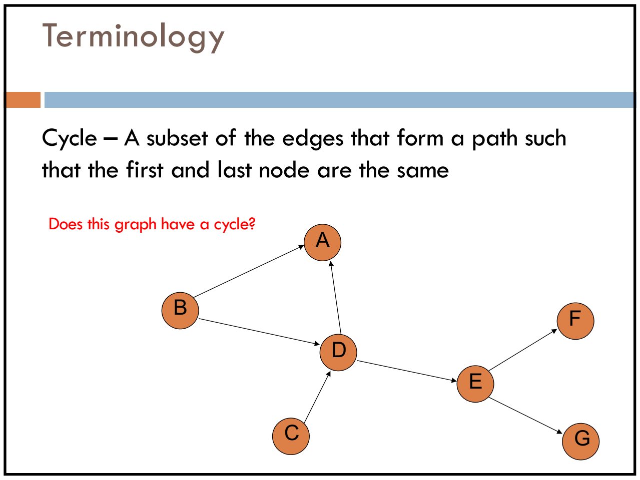
cycle?



Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

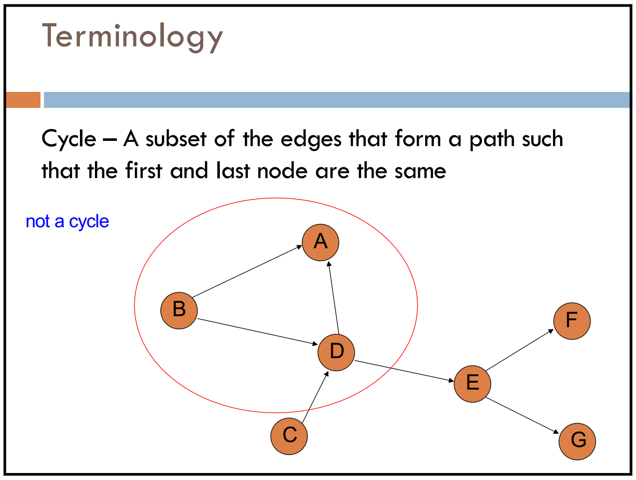
not a cycle



Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

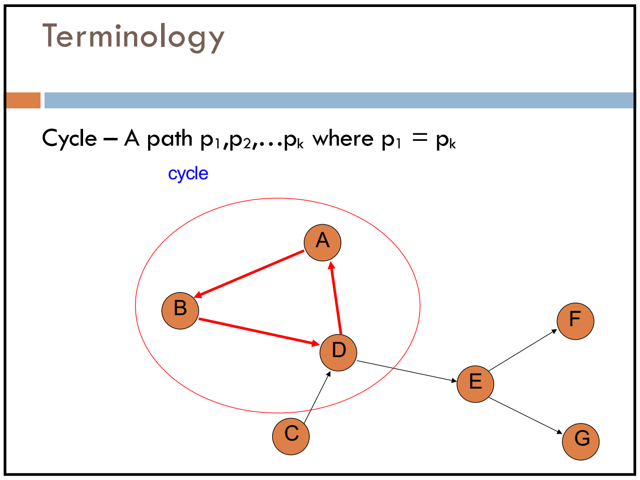
Does this graph have a cycle?



Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

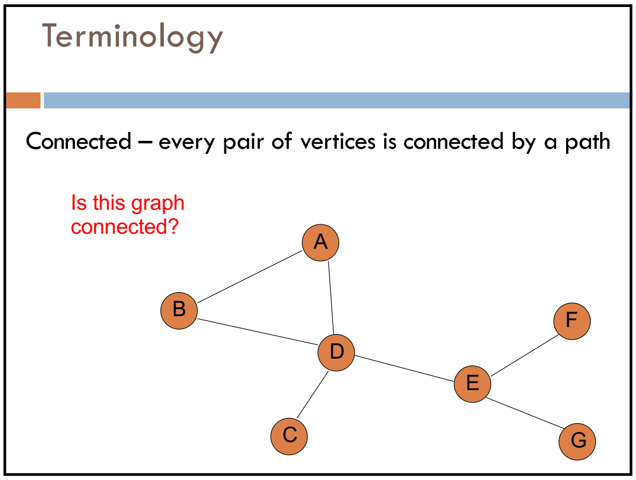
not a cycle



Terminology

Cycle – A path $p_1,p_2,\ldots p_k$ where $p_1 = p_k$

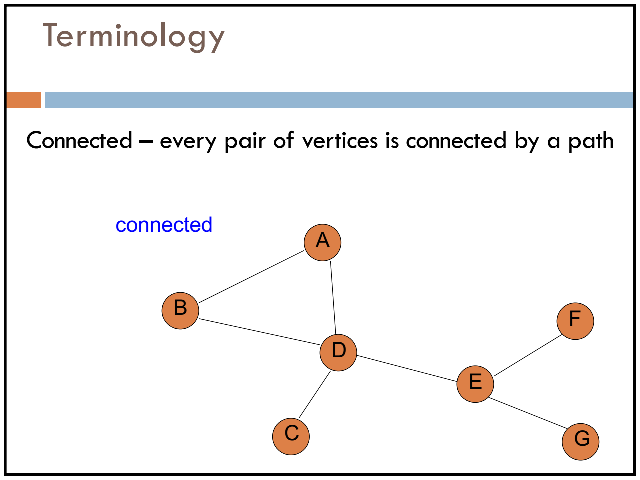
cycle



Terminology

Connected – every pair of vertices is connected by a path

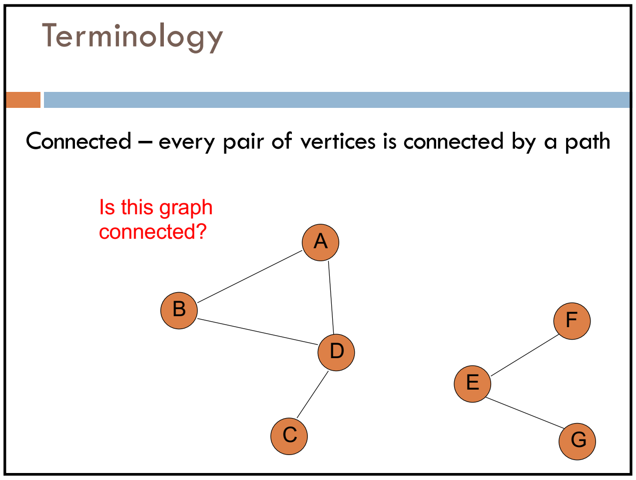
Is this graph connected?



Terminology

Connected – every pair of vertices is connected by a path

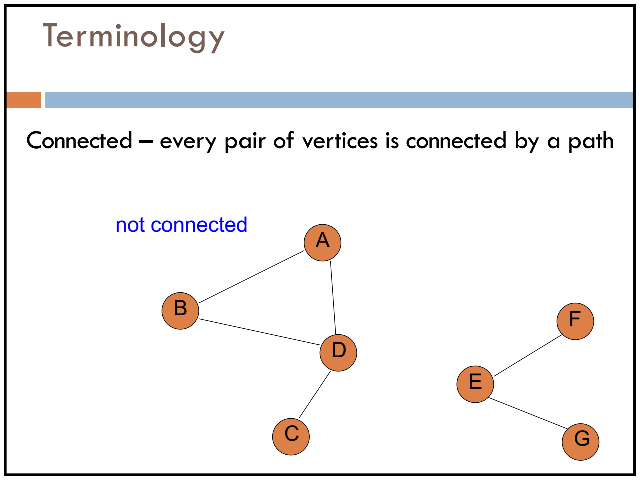
connected



Terminology

Connected – every pair of vertices is connected by a path

Is this graph connected?

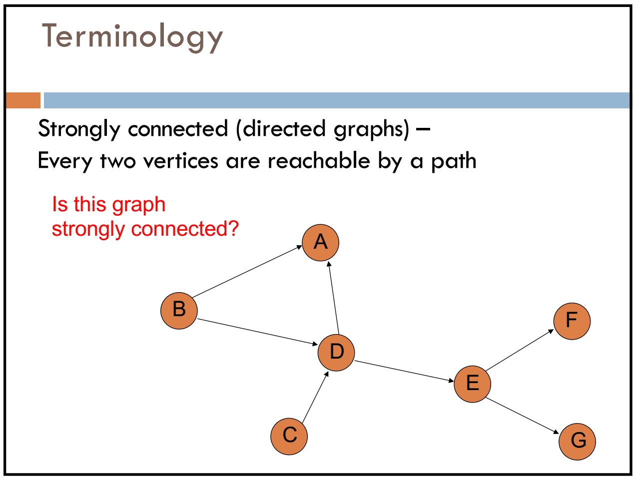


Terminology

Connected – every pair of vertices is connected by a path

not connected

26

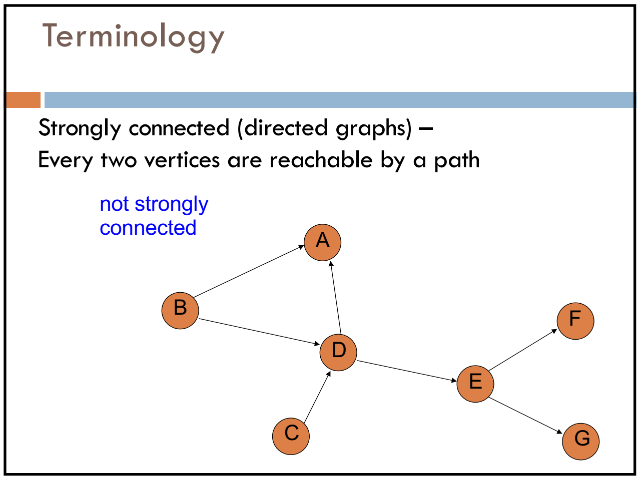


Terminology

Strongly connected (directed graphs) –
Every two vertices are reachable by a path

Is this graph
strongly connected?

27

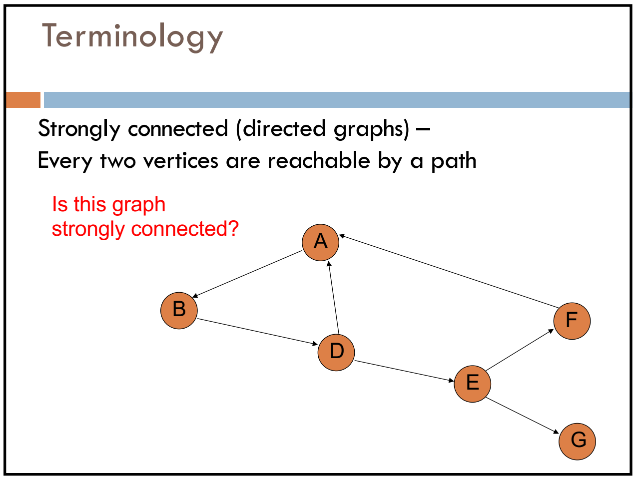


Terminology

Strongly connected (directed graphs) –
Every two vertices are reachable by a path

not strongly
connected

28

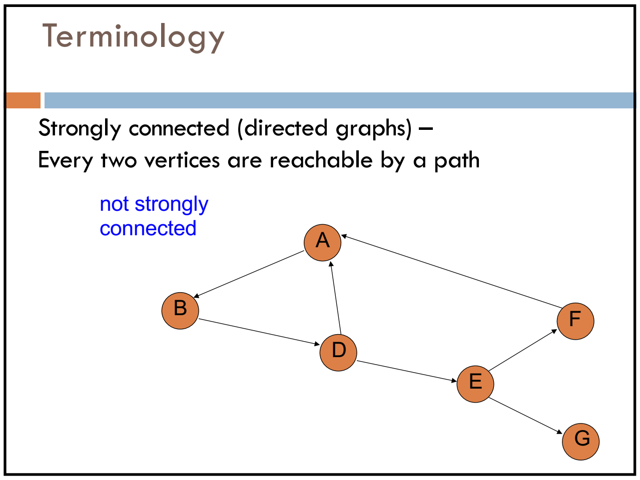


Terminology

Strongly connected (directed graphs) –
Every two vertices are reachable by a path

Is this graph
strongly connected?

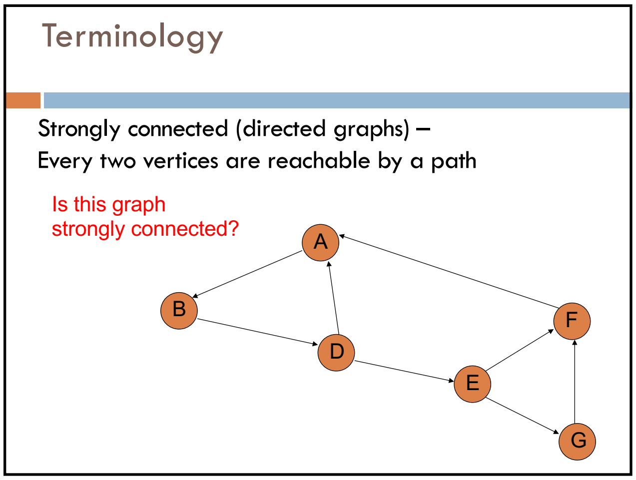
29



Terminology

Strongly connected (directed graphs) –
Every two vertices are reachable by a path

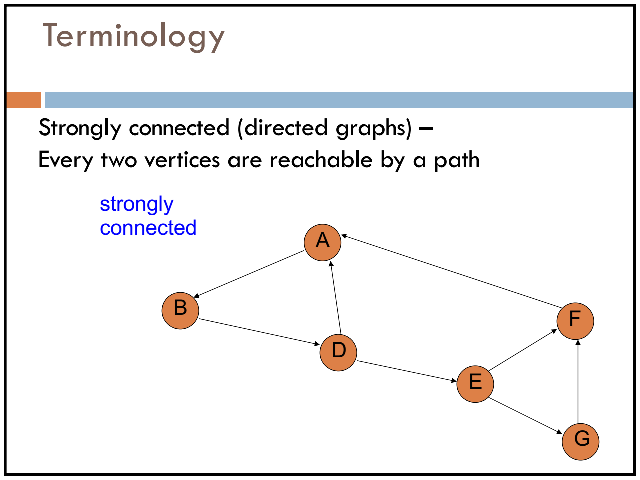
not strongly
connected



Terminology

Strongly connected (directed graphs) –
Every two vertices are reachable by a path

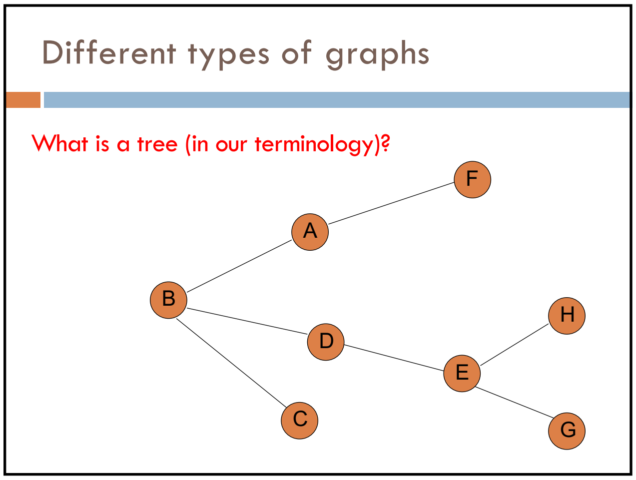
Is this graph
strongly connected?



Terminology

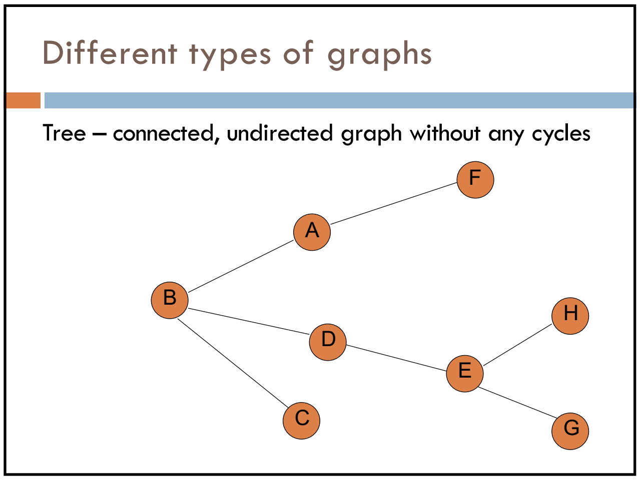
Strongly connected (directed graphs) –
Every two vertices are reachable by a path

strongly
connected



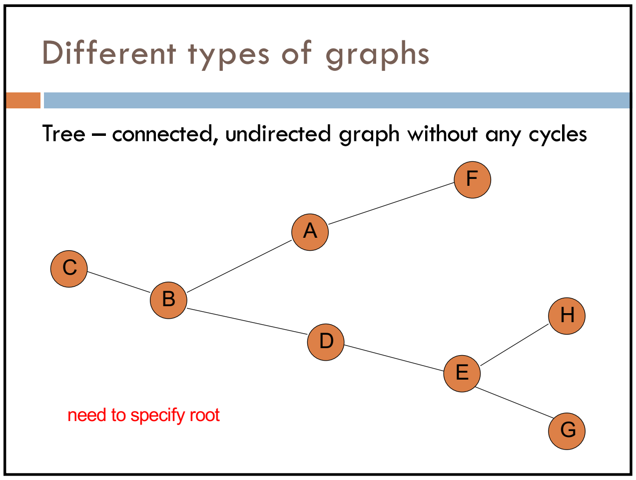
Different types of graphs

What is a tree (in our terminology)?



Different types of graphs

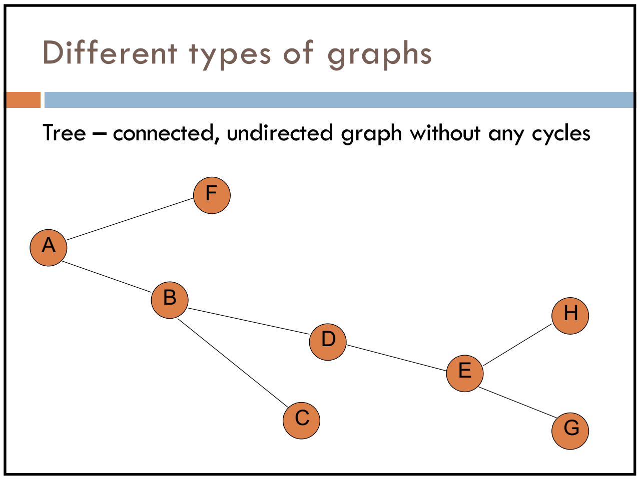
Tree – connected, undirected graph without any cycles



Different types of graphs

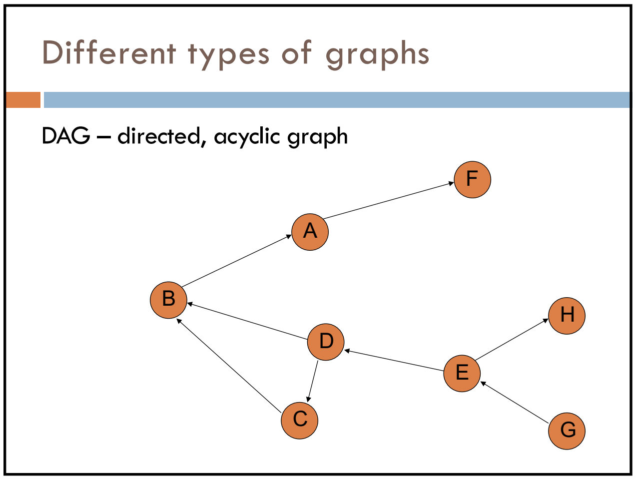
Tree – connected, undirected graph without any cycles

need to specify root



Different types of graphs

Tree – connected, undirected graph without any cycles

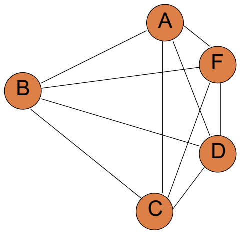


Different types of graphs

DAG – directed, acyclic graph

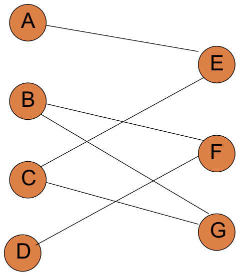
Different types of graphs

Complete graph – an edge exists between every node



Different types of graphs

Bipartite graph – a graph where every vertex can be partitioned into two sets X and Y such that all edges connect a vertex $u \in X$ and a vertex $v \in Y$



When do we see graphs in real life problems?

Transportation networks (flights, roads, etc.)

Communication networks

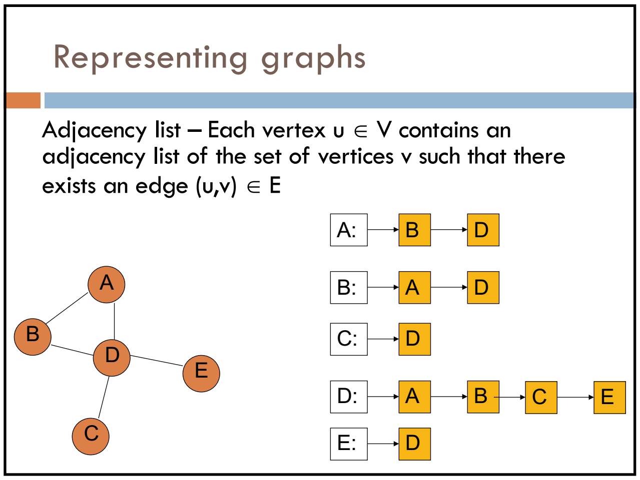
Web

Social networks

Circuit design

Bayesian networks

Representing graphs



Representing graphs

Adjacency list – Each vertex u ∈ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) ∈ E

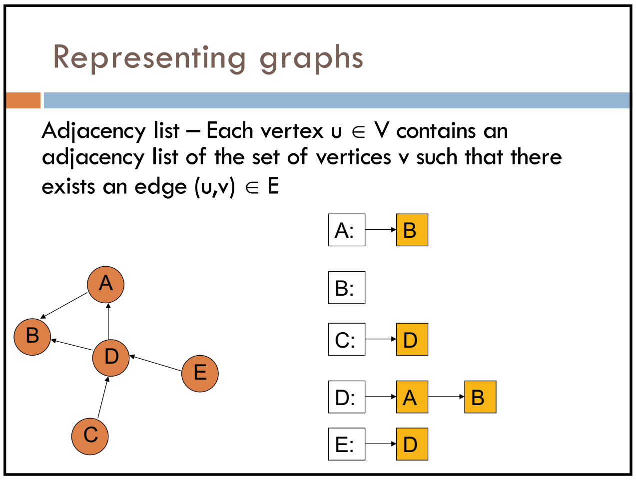
A: → B → D

B: → A → D

C: → D

D: → A → B → C → E

E: → D



Representing graphs

Adjacency list – Each vertex u ∈ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) ∈ E

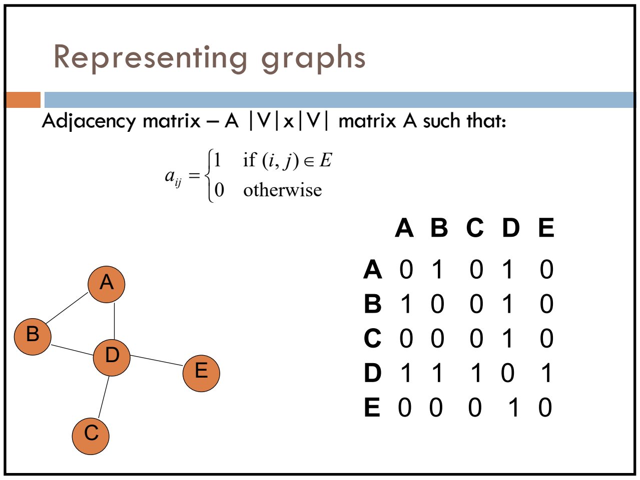
A: → B

B:

C: → D

D: → A → B

E: → D

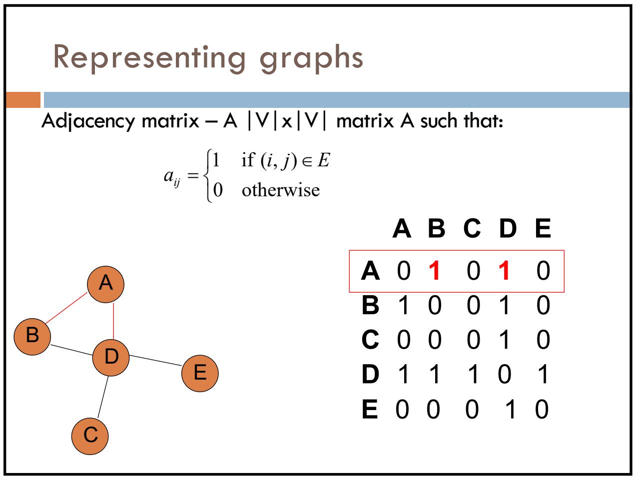


Representing graphs

Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \mathbf{A} & 0 & 1 & 0 & 1 & 0 \\ \mathbf{B} & 1 & 0 & 0 & 1 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 1 & 0 & 1 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$

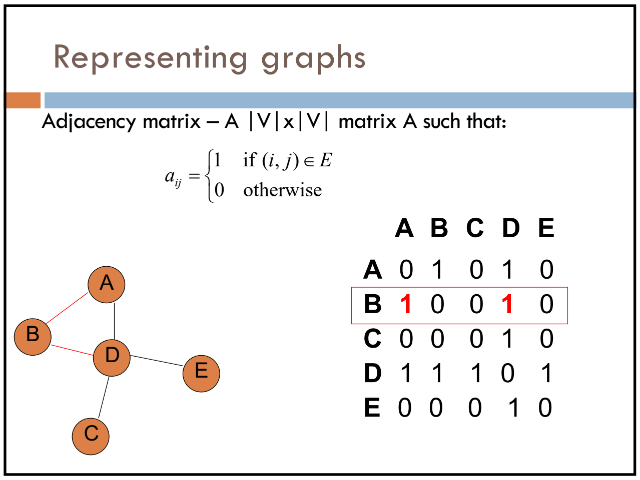


Representing graphs

Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \mathbf{A} & 0 & 1 & 0 & 1 & 0 \\ \mathbf{B} & 1 & 0 & 0 & 1 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 1 & 0 & 1 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$



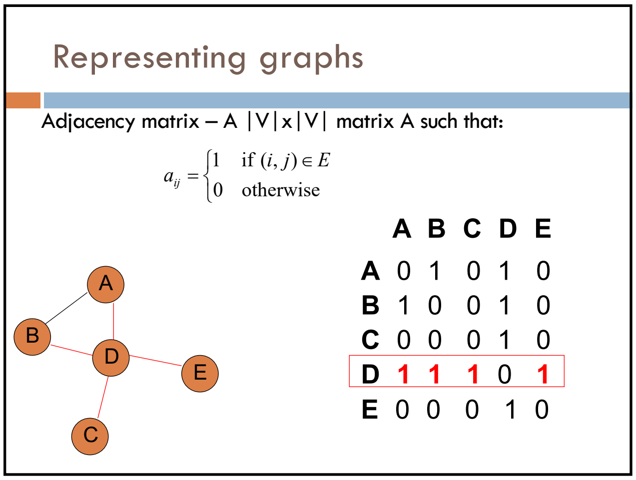
Representing graphs

Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \hline \mathbf{A} & 0 & 1 & 0 & 1 & 0 \\ \mathbf{B} & 1 & 0 & 0 & 1 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 1 & 0 & 1 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$

46



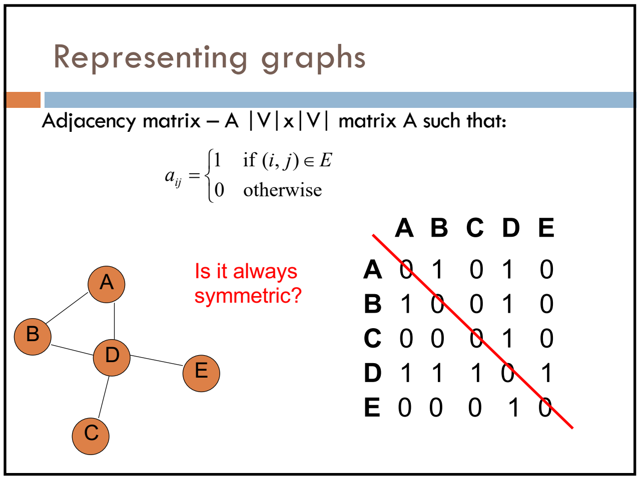
Representing graphs

Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \hline \mathbf{A} & 0 & 1 & 0 & 1 & 0 \\ \mathbf{B} & 1 & 0 & 0 & 1 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 1 & 0 & 1 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$

47



Representing graphs

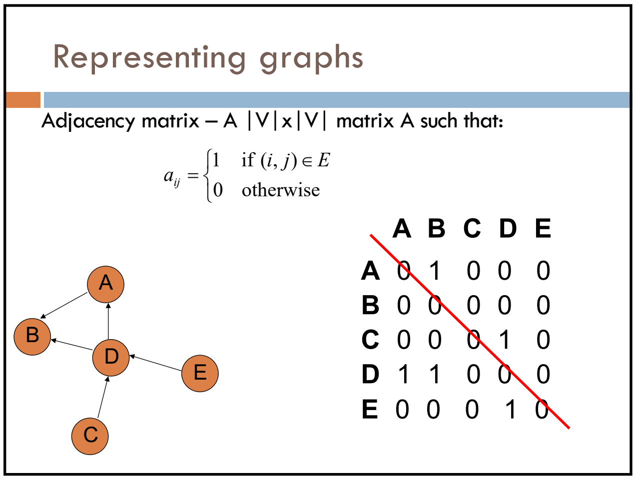
Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Is it always symmetric?

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \hline \mathbf{A} & 0 & 1 & 0 & 1 & 0 \\ \mathbf{B} & 1 & 0 & 0 & 1 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 1 & 0 & 1 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$

48



Representing graphs

Adjacency matrix – A |V|x|V| matrix A such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \hline \mathbf{A} & 0 & 1 & 0 & 0 & 0 \\ \mathbf{B} & 0 & 0 & 0 & 0 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 1 & 0 \\ \mathbf{D} & 1 & 1 & 0 & 0 & 0 \\ \mathbf{E} & 0 & 0 & 0 & 1 & 0 \end{array}$$

49

Adjacency list vs. adjacency matrix

Adjacency list	Adjacency matrix

Pros and cons?

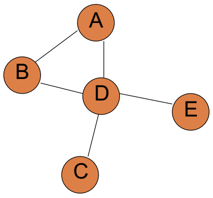
Adjacency list vs. adjacency matrix

Adjacency list	Adjacency matrix
Sparse graphs (e.g. web)	Dense graphs
Space efficient	Constant time lookup to discover if an edge exists
Must traverse the adjacency list to discover is an edge exists	Simple to implement
	For non-weighted graphs, only requires boolean matrix

Can we get the best of both worlds?

Sparse adjacency matrix

Rather than using an adjacency list, use an adjacency hashtable



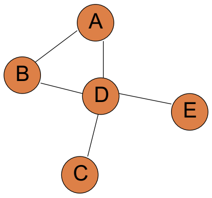
A:	hashtable [B,D]
B:	hashtable [A,D]
C:	hashtable [D]
D:	hashtable [A,B,C,E]
E:	hashtable [D]

Sparse adjacency matrix

Constant time lookup

Space efficient

Not good for dense graphs, why?

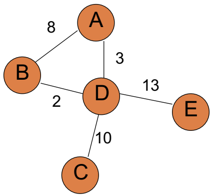


A:	hashtable [B,D]
B:	hashtable [A,D]
C:	hashtable [D]
D:	hashtable [A,B,C,E]
E:	hashtable [D]

Weighted graphs

Adjacency list

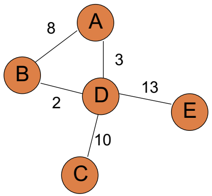
- store the weight as an additional field in the list



Weighted graphs

Adjacency matrix

$$a_{ij} = \begin{cases} weight & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$



$$\begin{array}{c|ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} & \mathbf{E} \\ \mathbf{A} & 0 & 8 & 0 & 3 & 0 \\ \mathbf{B} & 8 & 0 & 0 & 2 & 0 \\ \mathbf{C} & 0 & 0 & 0 & 10 & 0 \\ \mathbf{D} & 3 & 2 & 10 & 0 & 13 \\ \mathbf{E} & 0 & 0 & 0 & 13 & 0 \end{array}$$

Graph algorithms/questions

Graph traversal (BFS, DFS)

Shortest path from a to b

- unweighted
- weighted positive weights
- negative/positive weights

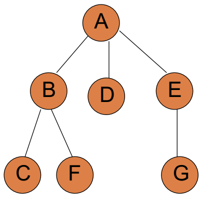
Minimum spanning trees

Are all nodes in the graph connected?

Is the graph bipartite?

DFS and BFS

How are they implemented?



What would be the result starting at A?
If you ask for the children of a node, they're given in alphabetical order.

Run-time (in terms of V and E):

- adjacency list
- adjacency matrix

Search implemented

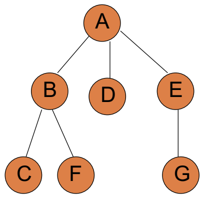
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
    v ← Dequeue(Q)
    Visit(v)
    for all c ∈ Children(v)
        Enqueue(Q, c)
```

```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
    v ← Pop(S)
    Visit(v)
    for all c ∈ Children(v)
        Push(S, c)
```

```
TreeDFS(v)
    visit(v)
    if not leaf(v)
        for all c in children(x)
            TreeDFS(v)
```

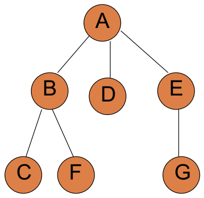
BFS

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
    v ← Dequeue(Q)
    Visit(v)
    for all c ∈ Children(v)
        Enqueue(Q, c)
```



BFS

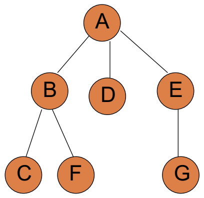
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
    v ← Dequeue(Q)
    Visit(v)
    for all c ∈ Children(v)
        Enqueue(Q, c)
```



A B D E C F G

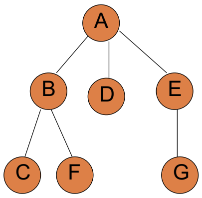
DFS

```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
    v ← Pop(S)
    Visit(v)
    for all c ∈ Children(v)
        Push(S, c)
```



DFS

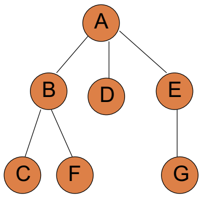
```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```



A E G D B F C

DFS

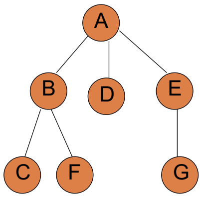
```
TreeDFS(v)
    visit(v)
    if not leaf(v)
        for all c in children(x)
            TreeDFS(v)
```



What changes?

DFS

```
TreeDFS(v)
    visit(v)
    if not leaf(v)
        for all c in children(x)
            TreeDFS(v)
```



A B C F D E G

Running time of BFS/DFS

Adjacency list
- How many times does it visit each vertex?
- How many times is each edge traversed?
- θ(|V|+|E|) – for trees, i.e., assuming a connected graph

Adjacency matrix
- For each vertex visited, how much work is done?
- θ(|V|^2) – for trees, i.e., assuming a connected graph

```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

DFS/BFS

Do they visit all of the nodes?

If the graph is connected or strongly connected

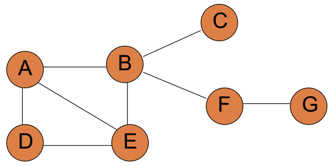
```
TreeBFS($T$)
Enqueue($Q$, Root($T$))
while !Empty($Q$)
        $v \leftarrow$ Dequeue($Q$)
        Visit($v$)
        for all $c \in$ Children($v$)
                Enqueue($Q, c$)
```

```
TreeDFS($T$)
Push($S$, Root($T$))
while !Empty($S$)
        $v \leftarrow$ Pop($S$)
        Visit($v$)
        for all $c \in$ Children($v$)
                Push($S, c$)
```

DFS/BFS for graphs

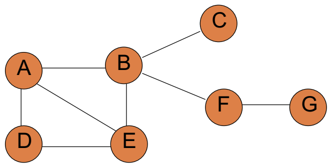
What needs to change for graphs?

Need to make sure we don't visit a node multiple times



BFS for graphs

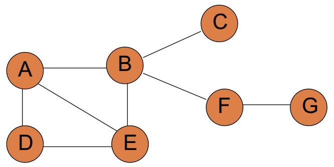
What order will BFS visit starting at A (again, assume children are enumerated alphabetically)?

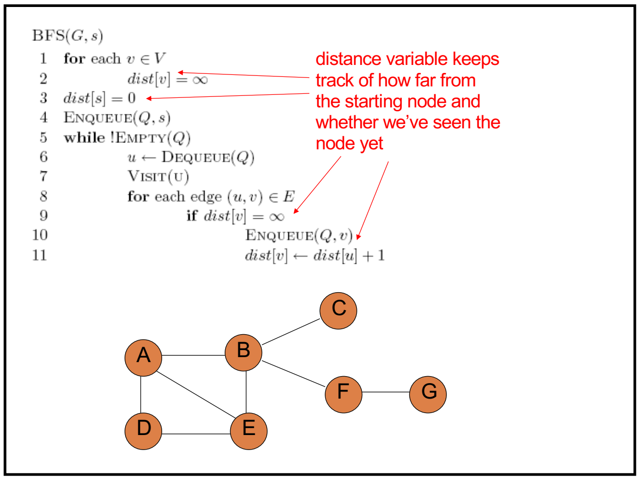


BFS for graphs

What order will BFS visit starting at A (again, assume children are enumerated alphabetically)?

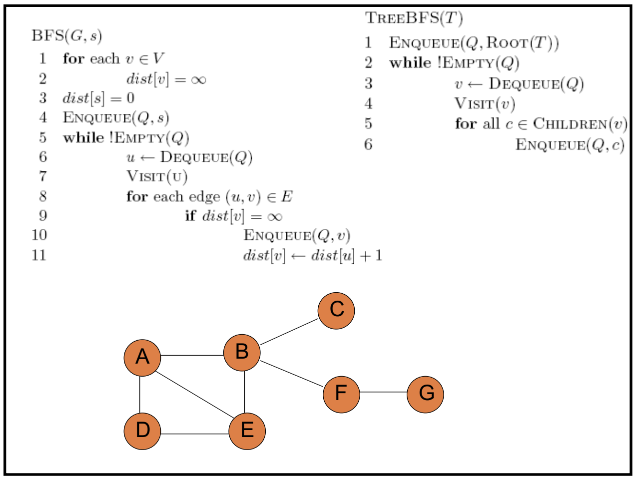
A B D E C F G





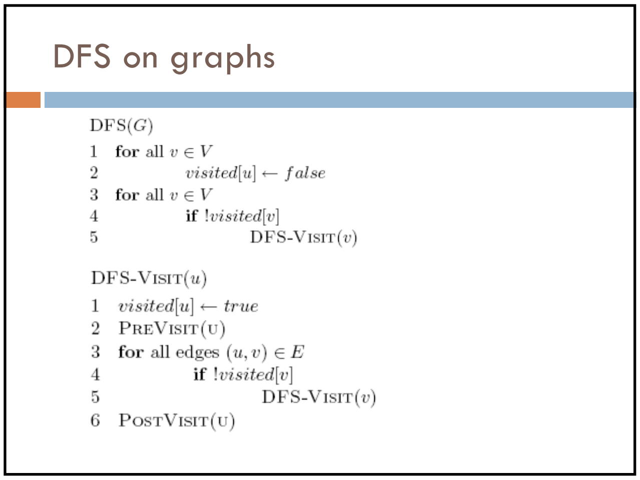
```
BFS($G, s$)
for each $v \in V$
    $dist[v] = \infty$
$dist[s] = 0$
ENQUEUE($Q, s$)
while !EMPTY($Q$)
    $u \leftarrow$ DEQUEUE($Q$)
    VISIT(U)
    for each edge $(u, v) \in E$
        if $dist[v] = \infty$
            ENQUEUE($Q, v$)
            $dist[v] \leftarrow dist[u] + 1$
```

distance variable keeps track of how far from the starting node and whether we've seen the node yet



```
BFS($G, s$)
for each $v \in V$
    $dist[v] = \infty$
$dist[s] = 0$
ENQUEUE($Q, s$)
while !EMPTY($Q$)
    $u \leftarrow$ DEQUEUE($Q$)
    VISIT(U)
    for each edge $(u, v) \in E$
        if $dist[v] = \infty$
            ENQUEUE($Q, v$)
            $dist[v] \leftarrow dist[u] + 1$
```

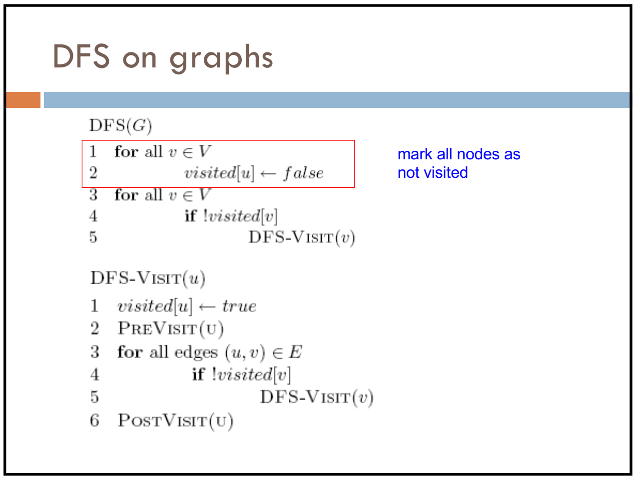
```
TREEBFS($T$)
ENQUEUE($Q$, ROOT($T$))
while !EMPTY($Q$)
    $v \leftarrow$ DEQUEUE($Q$)
    VISIT($v$)
    for all $c \in$ CHILDREN($v$)
        ENQUEUE($Q, c$)
```



DFS on graphs

```
DFS($G$)
for all $v \in V$
    $visited[u] \leftarrow false$
for all $v \in V$
    if $!visited[v]$
        DFS-VISIT($v$)
```

```
DFS-VISIT($u$)
$visited[u] \leftarrow true$
PREVISIT(U)
for all edges $(u, v) \in E$
    if $!visited[v]$
        DFS-VISIT($v$)
POSTVISIT(U)
```



DFS on graphs

```
DFS($G$)
for all $v \in V$
    $visited[u] \leftarrow false$
for all $v \in V$
    if $!visited[v]$
        DFS-VISIT($v$)
```

mark all nodes as not visited

```
DFS-VISIT($u$)
$visited[u] \leftarrow true$
PREVISIT(U)
for all edges $(u, v) \in E$
    if $!visited[v]$
        DFS-VISIT($v$)
POSTVISIT(U)
```

DFS on graphs

```
DFS(G)
for all $v \in V$
    $visited[u] \leftarrow false$
for all $v \in V$
    if $!visited[v]$
        DFS-Visit($v$)

DFS-Visit($u$)
$visited[u] \leftarrow true$
PreVisit(u)
for all edges $(u, v) \in E$
    if $!visited[v]$
        DFS-Visit($v$)
PostVisit(u)
```

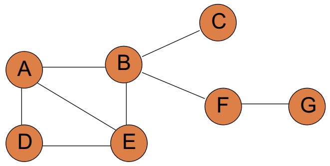
until **all** nodes have been visited repeatedly call DFS-Visit

DFS for graphs

What order will DFS visit starting at A (again, assume children are enumerated alphabetically)?

```
DFS(G)
for all $v \in V$
    $visited[u] \leftarrow false$
for all $v \in V$
    if $!visited[v]$
        DFS-Visit($v$)

DFS-Visit($u$)
$visited[u] \leftarrow true$
PreVisit(u)
for all edges $(u, v) \in E$
    if $!visited[v]$
        DFS-Visit($v$)
PostVisit(u)
```



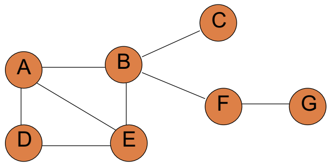
DFS for graphs

What order will DFS visit starting at A (again, assume children are enumerated alphabetically)?

```
DFS(G)
for all $v \in V$
    $visited[u] \leftarrow false$
for all $v \in V$
    if $!visited[v]$
        DFS-Visit($v$)

DFS-Visit($u$)
$visited[u] \leftarrow true$
PreVisit(u)
for all edges $(u, v) \in E$
    if $!visited[v]$
        DFS-Visit($v$)
PostVisit(u)
```

A B C E D F G



What does DFS do?

Finds connected components

Each call to DFS-Visit from DFS starts exploring a new set of connected components

Helps us understand the structure/connectedness of a graph

Running time of graph BFS/DFS

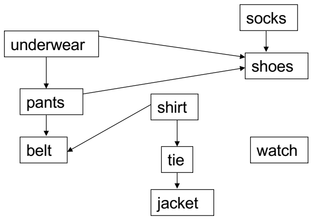
Nothing changes!

Adjacency list
- $O(|V|+|E|)$

Adjacency matrix
- $O(|V|^2)$

DAGs

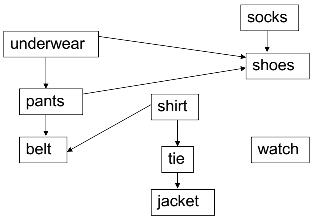
Can represent dependency graphs



Topological sort

A linear ordering of all the vertices such that for all edges (u,v) ∈ E, u appears before v in the ordering

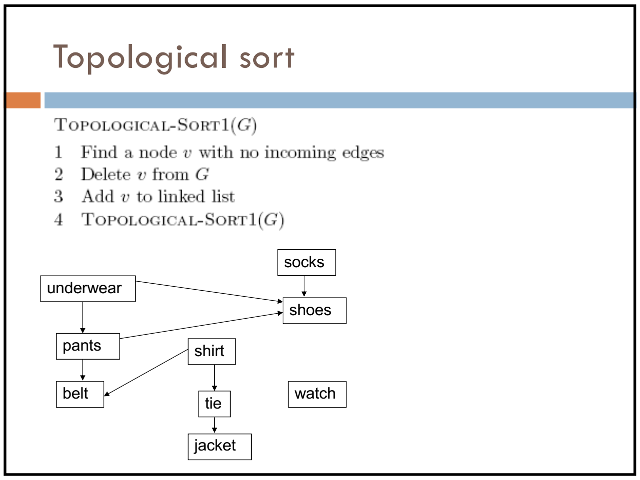
An ordering of the nodes that "obeys" the dependencies, i.e. an activity can't happen until it's dependent activities have happened



watch
underwear
pants
shirt
belt
tie
socks
shoes
jacket

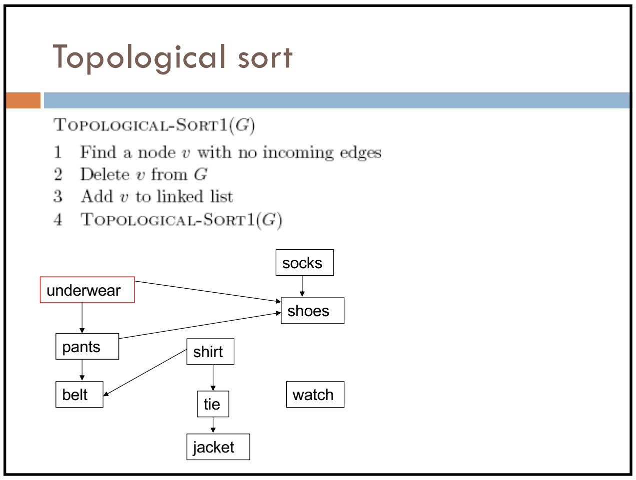
Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node $v$ with no incoming edges
    Delete $v$ from $G$
    Add $v$ to linked list
    TOPOLOGICAL-SORT1($G$)
```



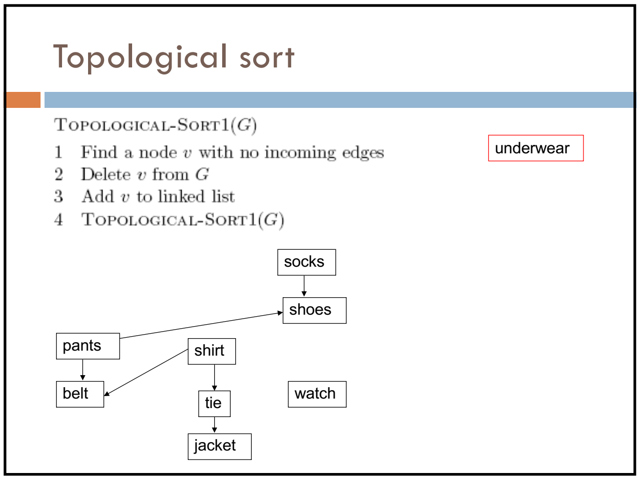
Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```



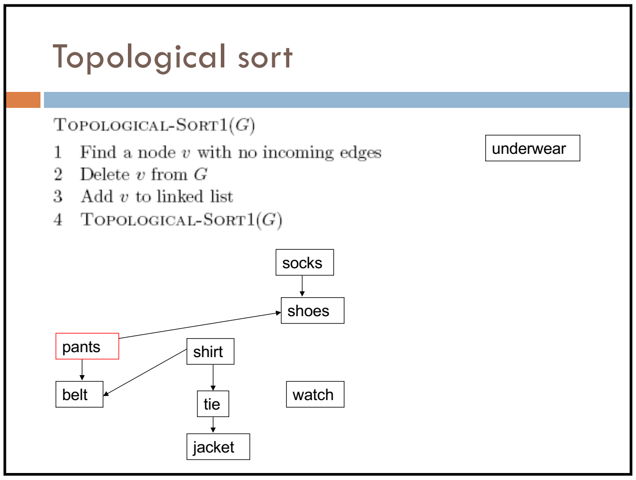
Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```



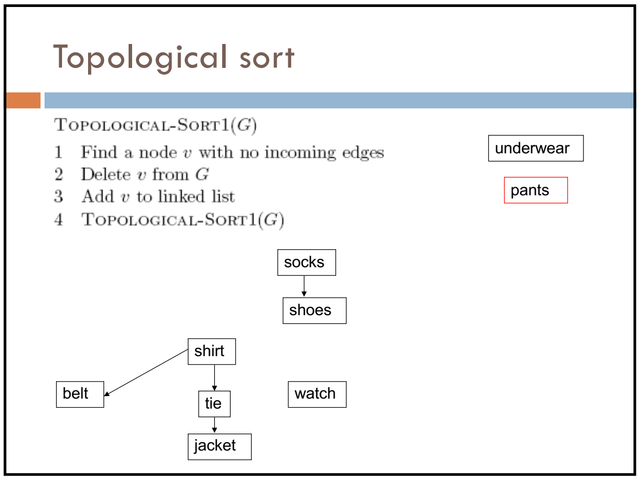
Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```



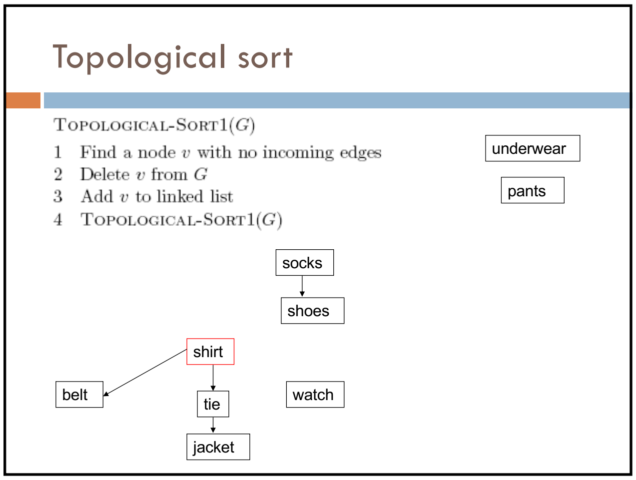
Topological sort

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```



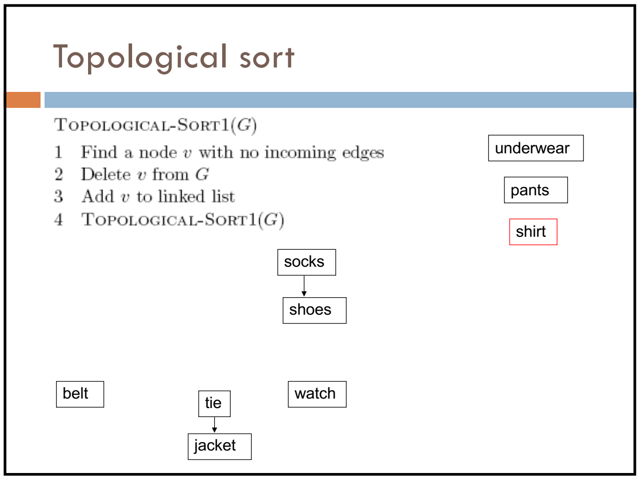
Topological sort

```
TOPOLOGICAL-SORT1($G$)
Find a node $v$ with no incoming edges
Delete $v$ from $G$
Add $v$ to linked list
TOPOLOGICAL-SORT1($G$)
```



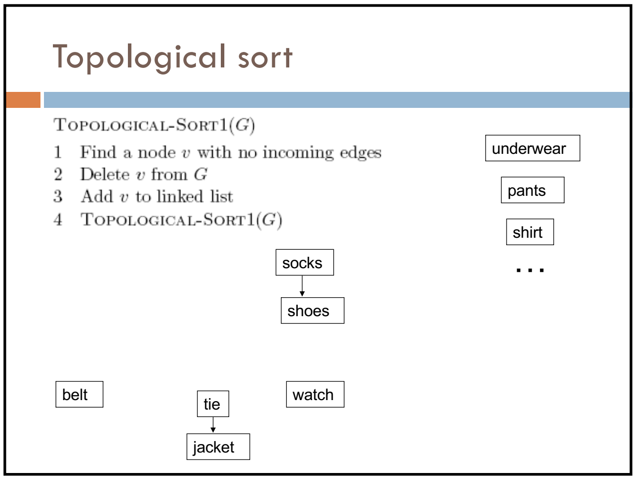
Topological sort

```
TOPOLOGICAL-SORT1($G$)
Find a node $v$ with no incoming edges
Delete $v$ from $G$
Add $v$ to linked list
TOPOLOGICAL-SORT1($G$)
```



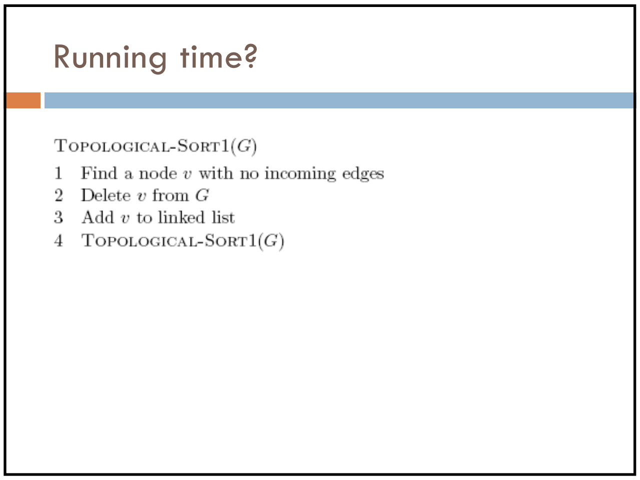
Topological sort

```
TOPOLOGICAL-SORT1($G$)
Find a node $v$ with no incoming edges
Delete $v$ from $G$
Add $v$ to linked list
TOPOLOGICAL-SORT1($G$)
```



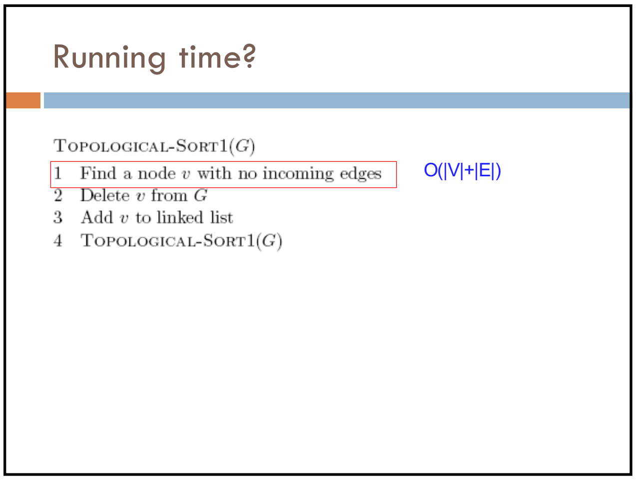
Topological sort

```
TOPOLOGICAL-SORT1($G$)
Find a node $v$ with no incoming edges
Delete $v$ from $G$
Add $v$ to linked list
TOPOLOGICAL-SORT1($G$)
```



Running time?

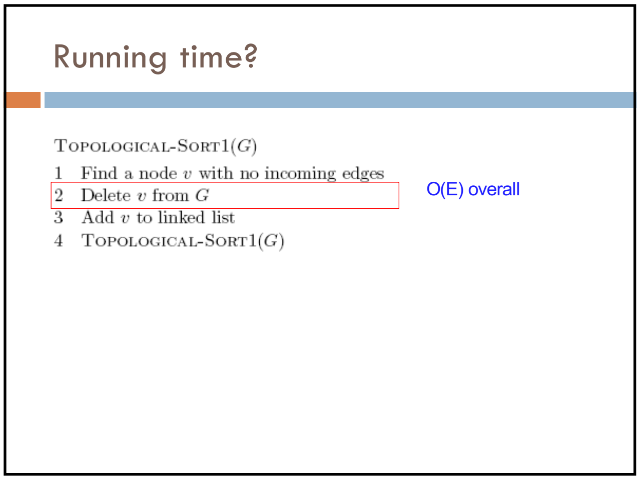
```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```



Running time?

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```

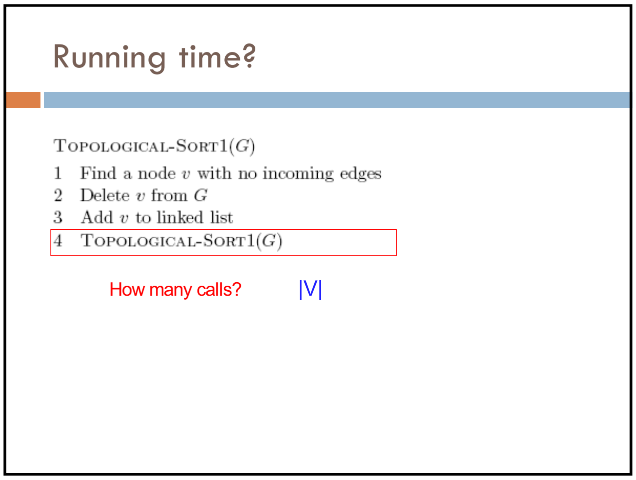
$O(|V|+|E|)$



Running time?

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```

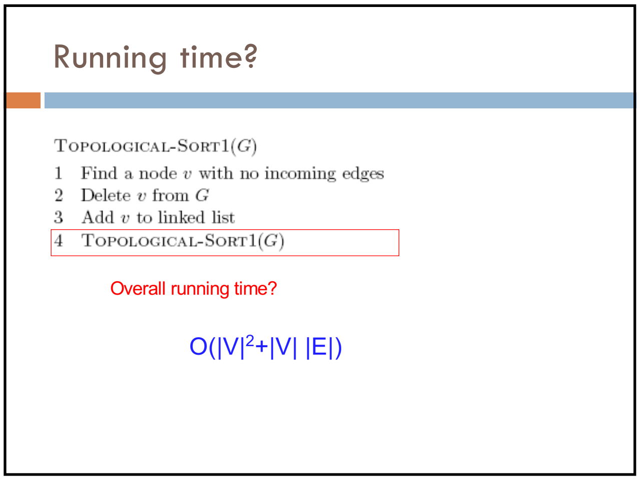
$O(E)$ overall



Running time?

```
TOPOLOGICAL-SORT1(G)
    Find a node v with no incoming edges
    Delete v from G
    Add v to linked list
    TOPOLOGICAL-SORT1(G)
```

How many calls? $|V|$

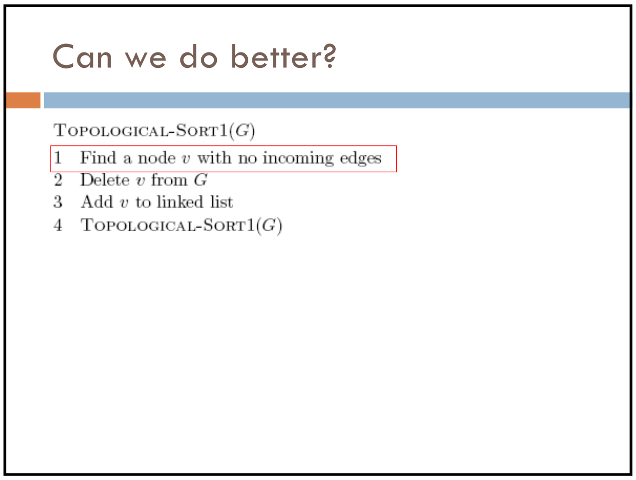


Running time?

```
TOPOLOGICAL-SORT1($G$)
Find a node $v$ with no incoming edges
Delete $v$ from $G$
Add $v$ to linked list
TOPOLOGICAL-SORT1($G$)
```

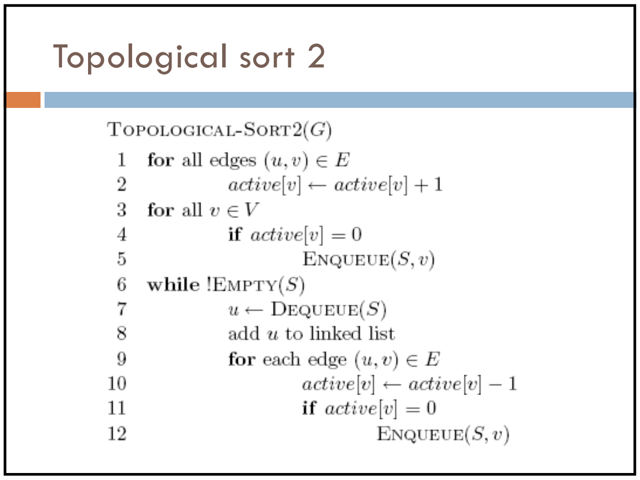
Overall running time?

$O(|V|^2+|V|\ |E|)$



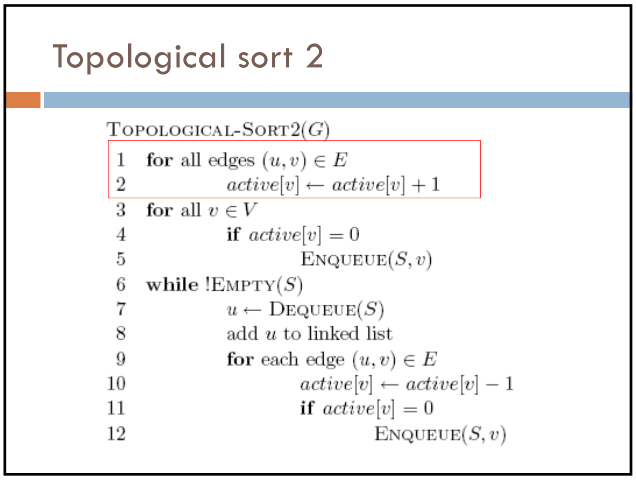
Can we do better?

```
TOPOLOGICAL-SORT1($G$)
Find a node $v$ with no incoming edges
Delete $v$ from $G$
Add $v$ to linked list
TOPOLOGICAL-SORT1($G$)
```



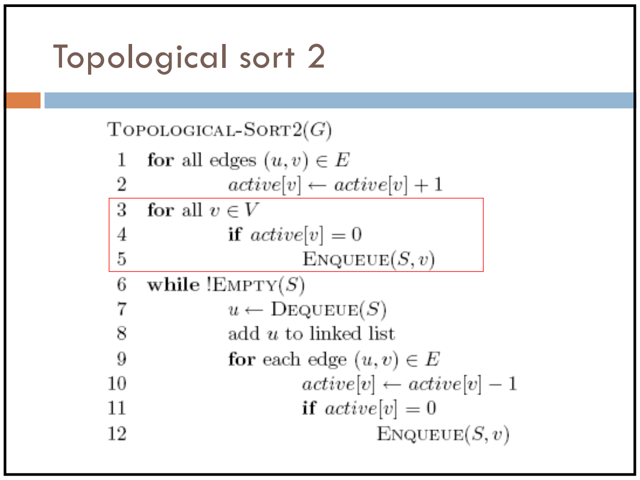
Topological sort 2

```
TOPOLOGICAL-SORT2($G$)
for all edges $(u, v) \in E$
    $active[v] \leftarrow active[v] + 1$
for all $v \in V$
    if $active[v] = 0$
        ENQUEUE($S, v$)
while !EMPTY($S$)
    $u \leftarrow$ DEQUEUE($S$)
    add $u$ to linked list
    for each edge $(u, v) \in E$
        $active[v] \leftarrow active[v] - 1$
        if $active[v] = 0$
            ENQUEUE($S, v$)
```



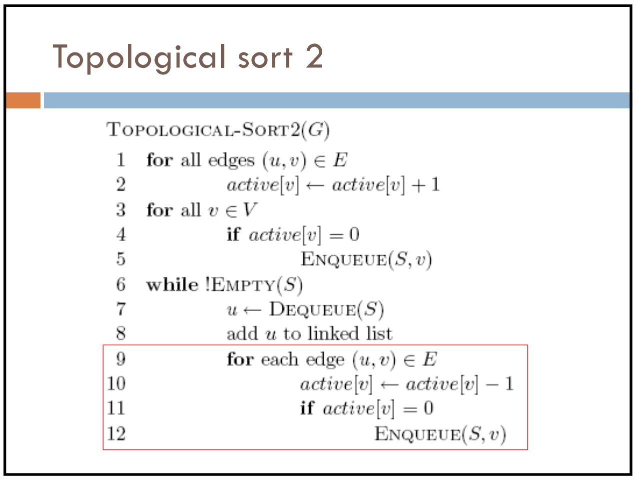
Topological sort 2

```
TOPOLOGICAL-SORT2($G$)
for all edges $(u, v) \in E$
    $active[v] \leftarrow active[v] + 1$
for all $v \in V$
    if $active[v] = 0$
        ENQUEUE($S, v$)
while !EMPTY($S$)
    $u \leftarrow$ DEQUEUE($S$)
    add $u$ to linked list
    for each edge $(u, v) \in E$
        $active[v] \leftarrow active[v] - 1$
        if $active[v] = 0$
            ENQUEUE($S, v$)
```



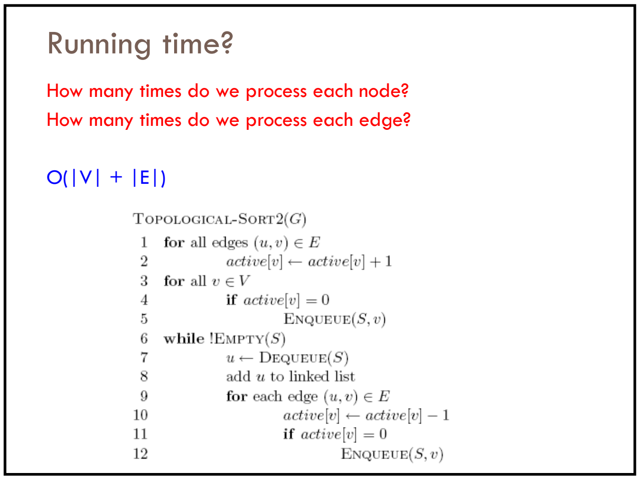
Topological sort 2

```
TOPOLOGICAL-SORT2($G$)
for all edges $(u, v) \in E$
        $active[v] \leftarrow active[v] + 1$
for all $v \in V$
        if $active[v] = 0$
                ENQUEUE($S, v$)
while !EMPTY($S$)
        $u \leftarrow$ DEQUEUE($S$)
        add $u$ to linked list
        for each edge $(u, v) \in E$
                $active[v] \leftarrow active[v] - 1$
                if $active[v] = 0$
                        ENQUEUE($S, v$)
```



Topological sort 2

```
TOPOLOGICAL-SORT2($G$)
for all edges $(u, v) \in E$
        $active[v] \leftarrow active[v] + 1$
for all $v \in V$
        if $active[v] = 0$
                ENQUEUE($S, v$)
while !EMPTY($S$)
        $u \leftarrow$ DEQUEUE($S$)
        add $u$ to linked list
        for each edge $(u, v) \in E$
                $active[v] \leftarrow active[v] - 1$
                if $active[v] = 0$
                        ENQUEUE($S, v$)
```



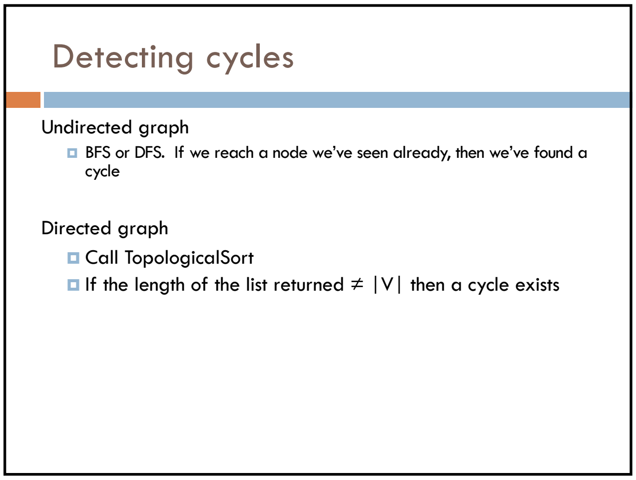
Running time?

How many times do we process each node?

How many times do we process each edge?

O(|V| + |E|)

```
TOPOLOGICAL-SORT2($G$)
for all edges $(u, v) \in E$
        $active[v] \leftarrow active[v] + 1$
for all $v \in V$
        if $active[v] = 0$
                ENQUEUE($S, v$)
while !EMPTY($S$)
        $u \leftarrow$ DEQUEUE($S$)
        add $u$ to linked list
        for each edge $(u, v) \in E$
                $active[v] \leftarrow active[v] - 1$
                if $active[v] = 0$
                        ENQUEUE($S, v$)
```



Detecting cycles

Undirected graph

- BFS or DFS. If we reach a node we've seen already, then we've found a cycle

Directed graph

- Call TopologicalSort
- If the length of the list returned ≠ |V| then a cycle exists

Connectedness

Given an undirected graph, for every node u ∈ V, can we reach all other nodes in the graph?
Algorithm + running time

Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

Running time: $O(|V| + |E|)$

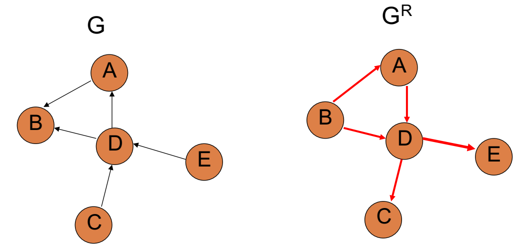
Strongly connected

Given a directed graph, can we reach any node v from any other node u?

Can we do the same thing?

Transpose of a graph

Given a graph G, we can calculate the transpose of a graph G^R by reversing the direction of all the edges



Running time to calculate G^R? $\theta(|V| + |E|)$

Strongly connected

Strongly-Connected(G)
- Run DFS-Visit or BFS from some node u
- If not all nodes are visited: return false
- Create graph G^R
- Run DFS-Visit or BFS on G^R from node u
- If not all nodes are visited: return false
- return true

Is it correct?

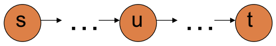
What do we know after the first pass?

- Starting at u, we can reach every node

What do we know after the second pass?

- All nodes can reach u. Why?
- We can get from u to every node in G^R, therefore, if we reverse the edges (i.e. G), then we have a path from every node to u

Which means that any node can reach any other node. Given any two nodes s and t we can create a path through u



Runtime?

```
Strongly-Connected(G)
-  Run DFS-Visit or BFS from some node u       O(|V| + |E|)
-  If not all nodes are visited: return false  O(|V|)
-  Create graph G^R                            θ(|V| + |E|)
-  Run DFS-Visit or BFS on G^R from node u     O(|V| + |E|)
-  If not all nodes are visited: return false  O(|V|)
-  return true
```

O(|V| + |E|)

Shortest path algorithms

Dijkstra's

Bellman-Ford

Floyd-Warshall

Johnson's

Shortest path algorithms