GRAPHS

David Kauchak
CS 140 – Fall 2022

1

Admin

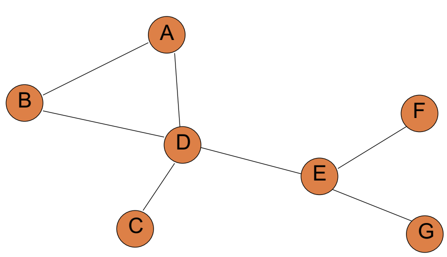
Checkpoint corrections

Assignment 3 feedback

2

Graphs

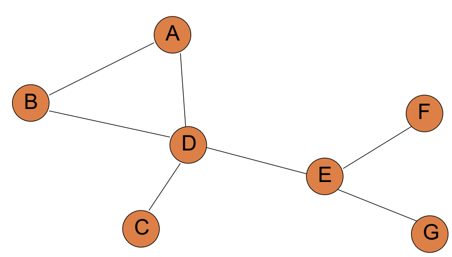
What is a graph?



3

Graphs

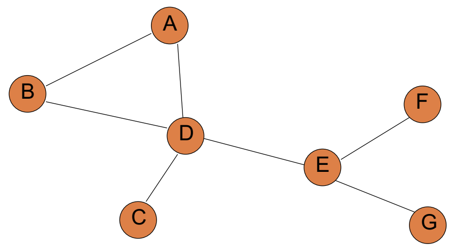
A graph is a set of vertices V and a set of edges (u,v) $\in$ E where u,v $\in$ V



4

Graphs

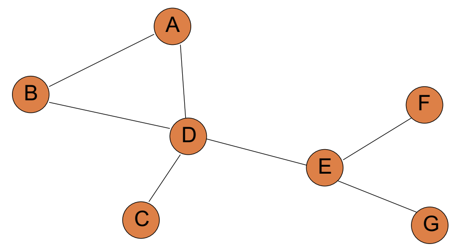
How do graphs differ? What are graph characteristics we might care about?



5

Different types of graphs

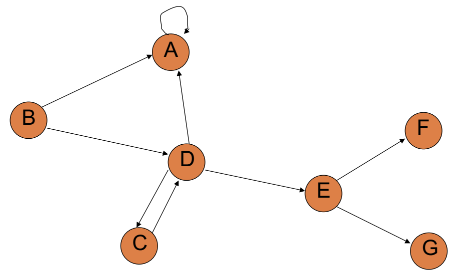
Undirected – edges do not have a direction



6

Different types of graphs

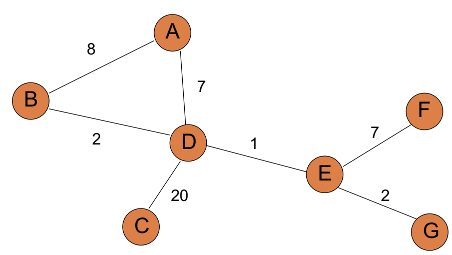
Directed – edges **do** have a direction



7

Different types of graphs

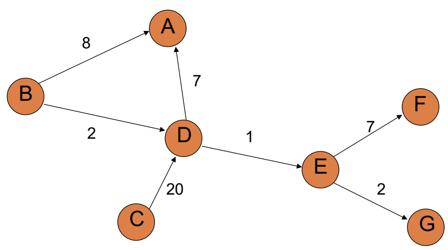
Weighted – edges have an associated weight



8

Different types of graphs

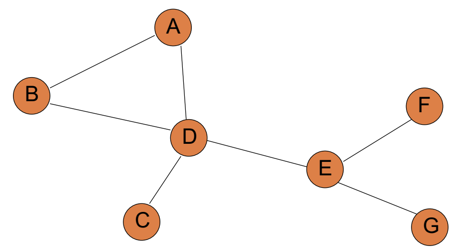
Weighted – edges have an associated weight



9

Terminology

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

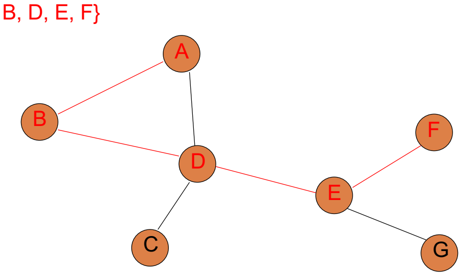


10

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}

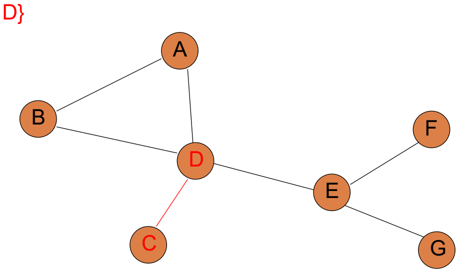


11

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}

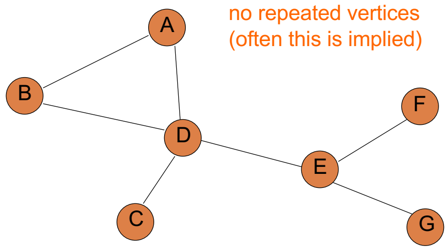


12

Terminology

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

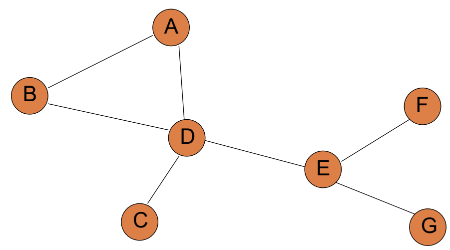
A *simple* path contains no repeated vertices (often this is implied)



13

Terminology

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



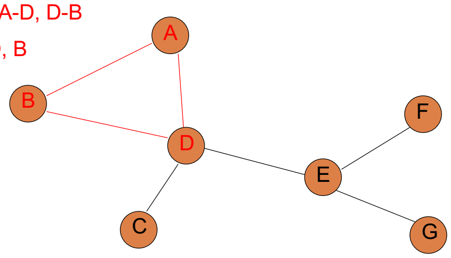
14

Terminology

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

Edges: B-A, A-D, D-B

Path: B, A, D, B

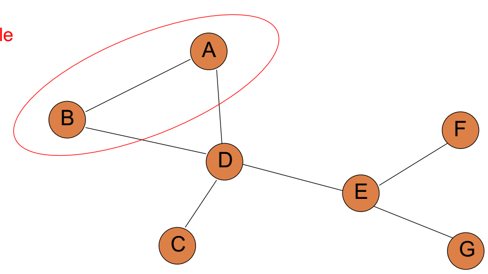


15

Terminology

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

not a cycle

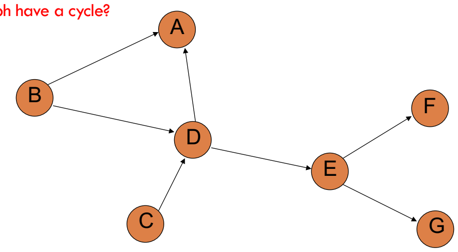


16

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?

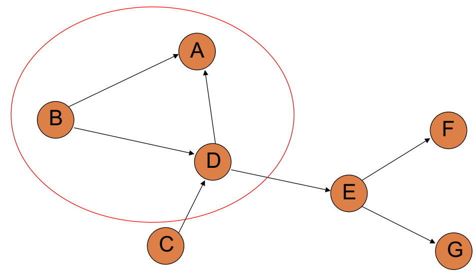


17

Terminology

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

not a cycle

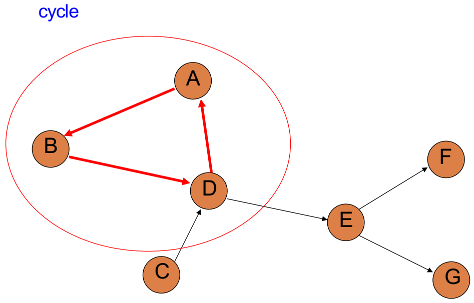


18

Terminology

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

cycle

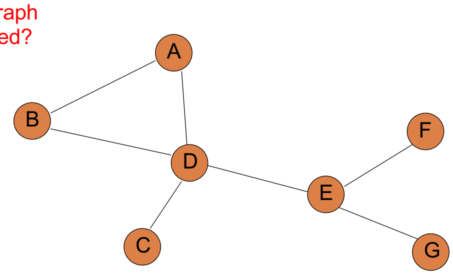


19

Terminology

Connected – every pair of vertices is connected by a path

Is this graph connected?

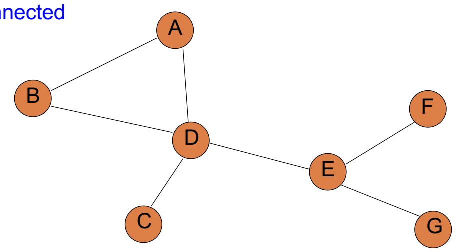


20

Terminology

Connected – every pair of vertices is connected by a path

connected

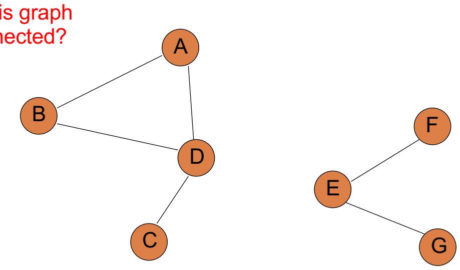


21

Terminology

Connected – every pair of vertices is connected by a path

Is this graph
connected?

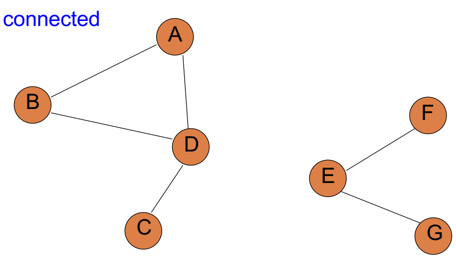


22

Terminology

Connected – every pair of vertices is connected by a path

not connected

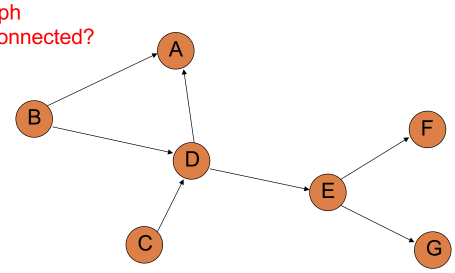


23

Terminology

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

Is this graph
strongly connected?

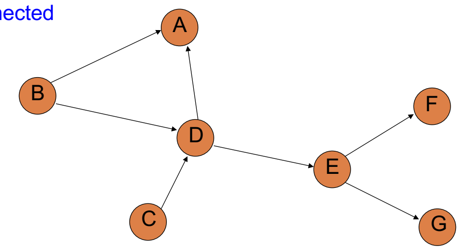


24

Terminology

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

not strongly
connected

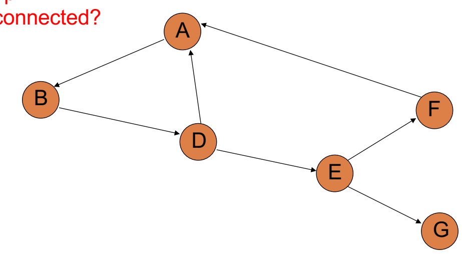


25

Terminology

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

Is this graph
strongly connected?

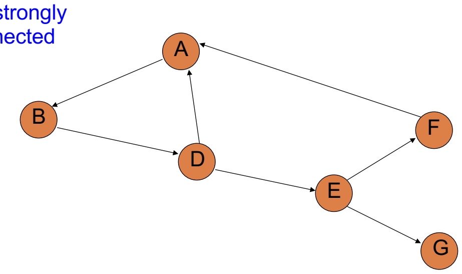


26

Terminology

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

not strongly
connected

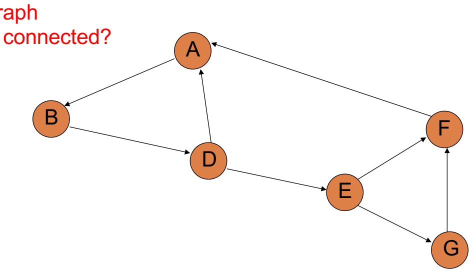


27

Terminology

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

Is this graph
strongly connected?

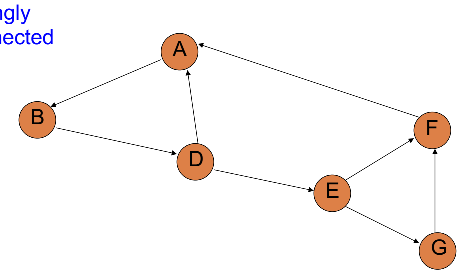


28

Terminology

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

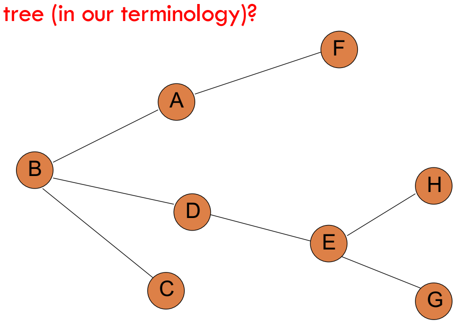
strongly
connected



29

Different types of graphs

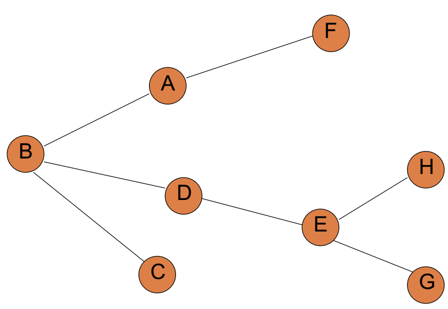
What is a tree (in our terminology)?



30

Different types of graphs

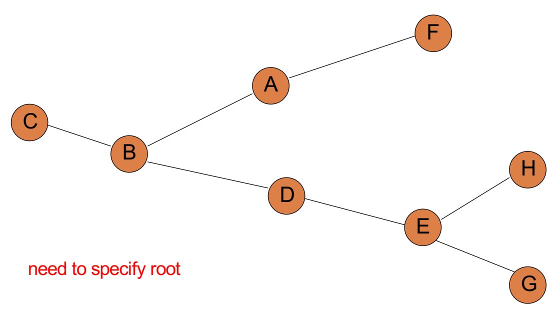
Tree – connected, undirected graph without any cycles



31

Different types of graphs

Tree – connected, undirected graph without any cycles

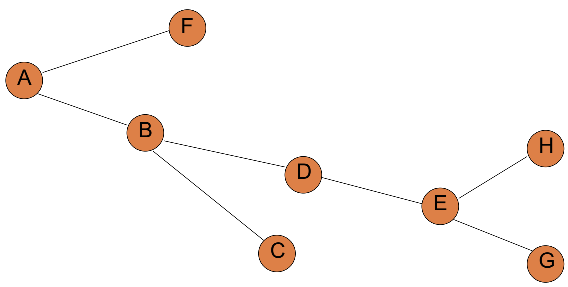


need to specify root

32

Different types of graphs

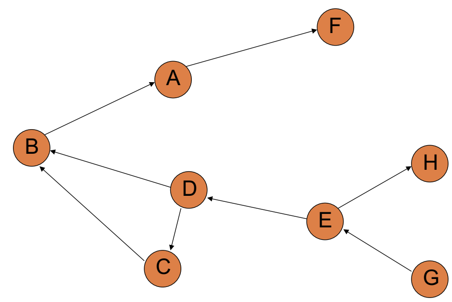
Tree – connected, undirected graph without any cycles



33

Different types of graphs

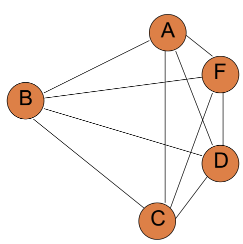
DAG – directed, acyclic graph



34

Different types of graphs

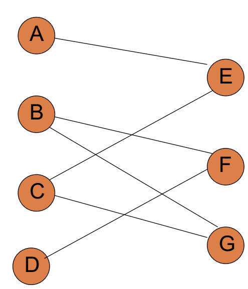
Complete graph – an edge exists between every node



35

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 ∈ X and a vertex v ∈ Y



36

When do we see graphs in real life problems?

- Transportation networks (flights, roads, etc.)
- Communication networks
- Web
- Social networks
- Circuit design
- Bayesian networks

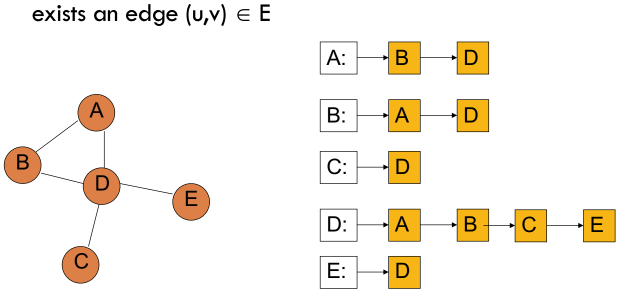
37

Representing graphs

38

Representing graphs

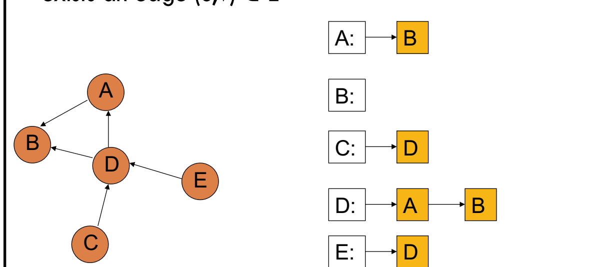
Adjacency list – Each vertex u ∈ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) ∈ E



39

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

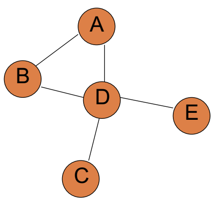


40

Representing graphs

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

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



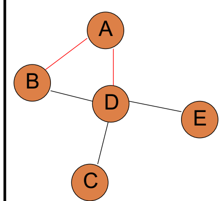
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

41

Representing graphs

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

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



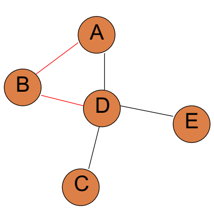
	A	B	C	D	E
A	0	**1**	0	**1**	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

42

Representing graphs

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

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



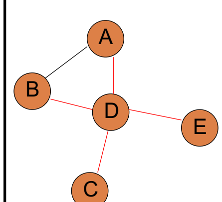
	A	B	C	D	E
A	0	1	0	1	0
B	**1**	0	0	**1**	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

43

Representing graphs

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

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



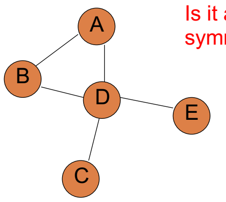
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	**1**	**1**	**1**	0	**1**
E	0	0	0	1	0

44

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?

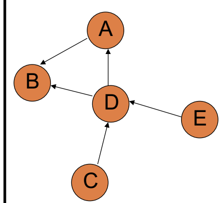
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

45

Representing graphs

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

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



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	0	0
C	0	0	0	1	0
D	1	1	0	0	0
E	0	0	0	1	0

46

Adjacency list vs. adjacency matrix

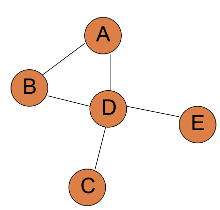
Adjacency list	Adjacency matrix
Sparse graphs (e.g. web)	Dense graphs
Space efficient	Constant time lookup to discover if an edge exists
Must traverse the adjacency list to discover is an edge exists	Simple to implement
	For non-weighted graphs, only requires boolean matrix

Can we get the best of both worlds?

47

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]

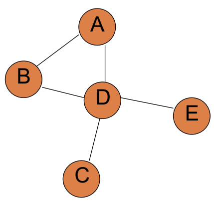
48

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]

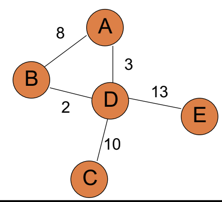
49

Weighted graphs

Adjacency list

- store the weight as an additional field in the list

A: → B:8 → D:3



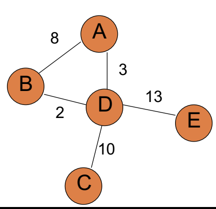
50

Weighted graphs

Adjacency matrix

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

	A	**B**	**C**	**D**	**E**
A	0	8	0	3	0
B	8	0	0	2	0
C	0	0	0	10	0
D	3	2	10	0	13
E	0	0	0	13	0



51

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?

52

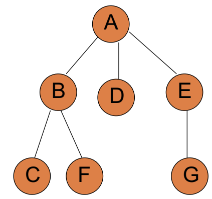
Breadth First Search (BFS) on Trees

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

53

Tree BFS

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



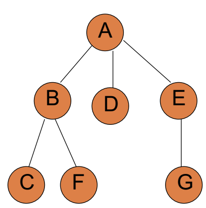
Q:

Visited:

54

Tree BFS

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



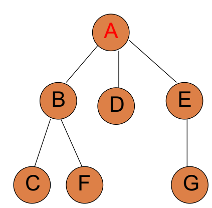
Q: A

Visited:

55

Tree BFS

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



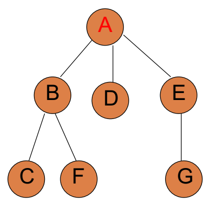
Q:

Visited: A

56

Tree BFS

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



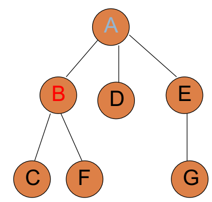
Q: B, D, E

Visited: A

57

Tree BFS

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



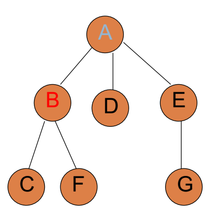
Q: D, E

Visited: A B

58

Tree BFS

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



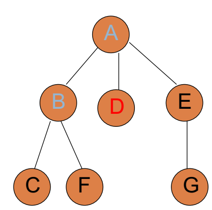
Q: D, E, C, F

Visited: A B

59

Tree BFS

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



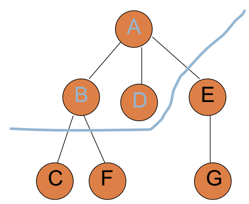
Q: E, C, F

Visited: A B D

60

Tree BFS

```
TreeBFS(T)
ENQUEUE(Q, ROOT(T))
while !EMPTY(Q)
        v ← DEQUEUE(Q)
        VISIT(v)
        for all c ∈ CHILDREN(v)
                ENQUEUE(Q, c)
```



Q: E, C, F

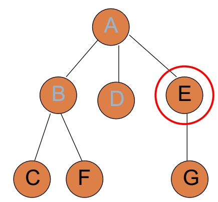
Visited: A B D

Frontier: the set of vertices that have been visited so far

61

Tree BFS

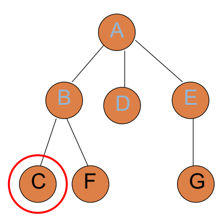
```
TreeBFS(T)
ENQUEUE(Q, ROOT(T))
while !EMPTY(Q)
        v ← DEQUEUE(Q)
        VISIT(v)
        for all c ∈ CHILDREN(v)
                ENQUEUE(Q, c)
```



62

Tree BFS

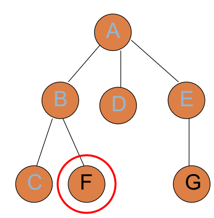
```
TreeBFS(T)
ENQUEUE(Q, ROOT(T))
while !EMPTY(Q)
        v ← DEQUEUE(Q)
        VISIT(v)
        for all c ∈ CHILDREN(v)
                ENQUEUE(Q, c)
```



63

Tree BFS

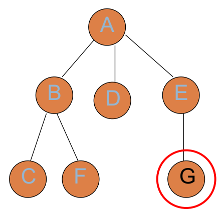
```
TreeBFS(T)
ENQUEUE(Q, ROOT(T))
while !EMPTY(Q)
        v ← DEQUEUE(Q)
        VISIT(v)
        for all c ∈ CHILDREN(v)
                ENQUEUE(Q, c)
```



64

Tree BFS

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



65

Tree BFS

What order does the algorithm traverse the nodes?

BFS traversal visits the nodes in increasing distance from the root

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

66

Tree BFS

Does it 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)

67

Running time of Tree BFS

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

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

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

68

BFS Recursively

Hard to do!

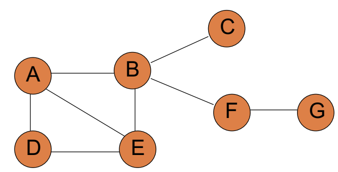
```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
      v ← Dequeue(Q)
      Visit(v)
      for all c ∈ Children(v)
          Enqueue(Q, c)
```

69

BFS for graphs

What needs to change for graphs?

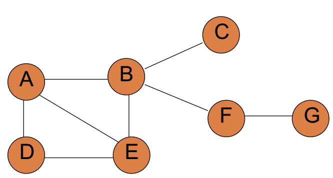
Need to make sure we don't visit a node multiple times



70

BFS for graphs

What order will BFS visit starting at A (break ties to visit based alphabetically, with earlier first)?

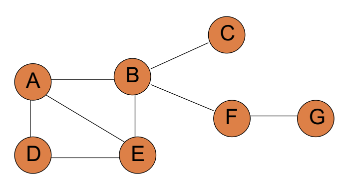


71

BFS for graphs

What order will BFS visit starting at A (break ties to visit based alphabetically, with earlier first)?

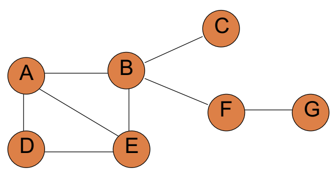
A B D E C F G



72

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

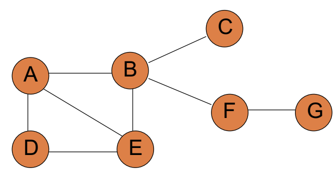
distance variable keeps track of how far from the starting node and whether we've seen the node yet



73

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

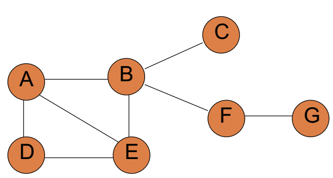
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



74

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

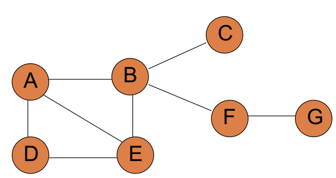
set all nodes as unseen



75

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```

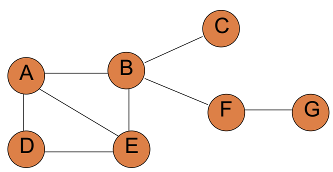
check if the node has been seen



76

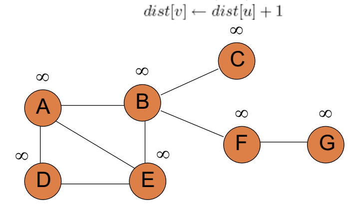
```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(U)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```

set the node as seen and record distance



77

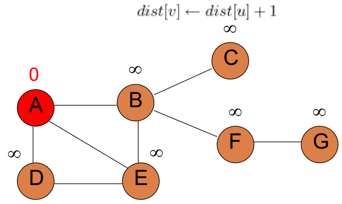
```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(U)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```



78

```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(U)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```

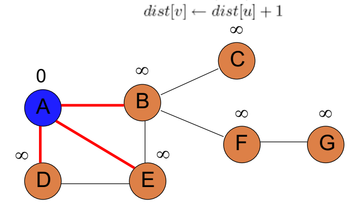
Q: A



79

```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(U)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```

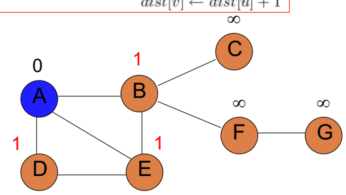
Q:



80

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

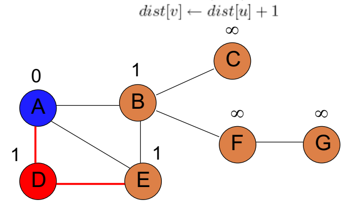
Q: D, E, B



81

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

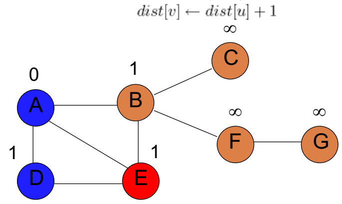
Q: E, B



82

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

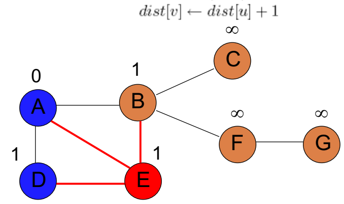
Q: B



83

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

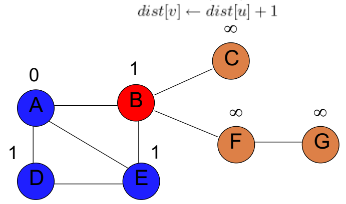
Q: B



84

```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

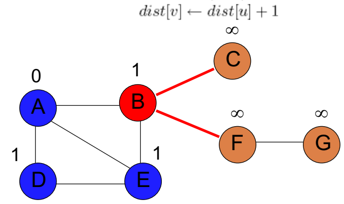
Q:



85

```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

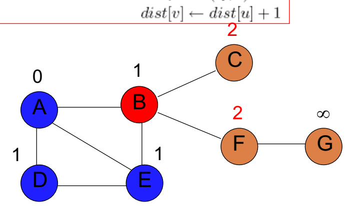
Q:



86

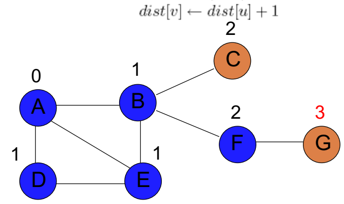
```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

Q: F, C



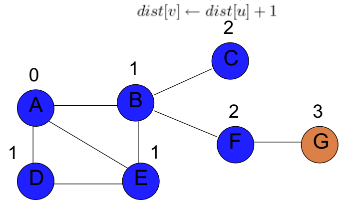
87

```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```



88

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```



89

Runtime of BFS

Nothing changed over our analysis of TreeBFS

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

```
TREEBFS(T)
ENQUEUE(Q, ROOT(T))
while !EMPTY(Q)
        v ← DEQUEUE(Q)
        VISIT(v)
        for all c ∈ CHILDREN(v)
                ENQUEUE(Q, c)
```

91

Runtime of BFS

Adjacency list: O(|V| + |E|)

Adjacency matrix: O(|V|^2)

(we won't assumed connectedness)

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

92

Depth First Search (DFS)

```
TREEDFS(T)
PUSH(S, ROOT(T))
while !EMPTY(S)
        v ← POP(S)
        VISIT(v)
        for all c ∈ CHILDREN(v)
                PUSH(S, c)
```

93

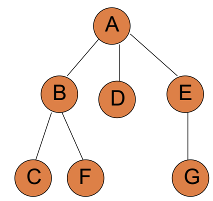
Depth First Search (DFS)

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

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

94

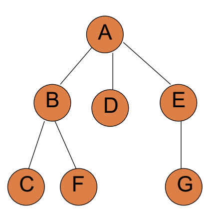
Tree DFS



Stack

95

Tree DFS

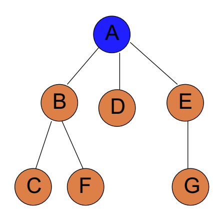


A

Stack

96

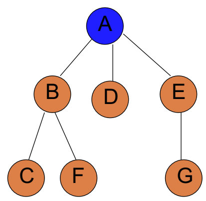
Tree DFS



Stack

97

Tree DFS

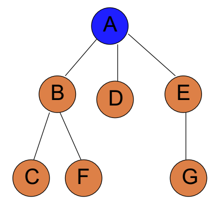


B
D
E

Stack

98

Tree DFS



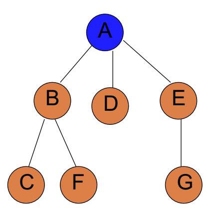
B
D
E

Stack

What does this assume about how we add them to the stack?

99

Tree DFS



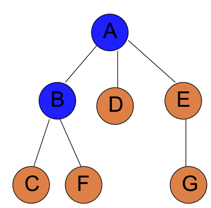
B
D
E

Stack

Added from right to left: E, then D, then B

100

Tree DFS

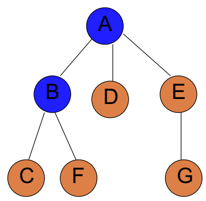


D
E

Stack

101

Tree DFS

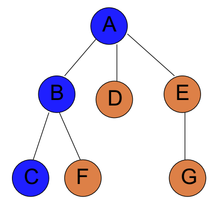


C
F
D
E

Stack

102

Tree DFS

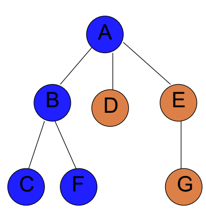


F
D
E

Stack

103

Tree DFS



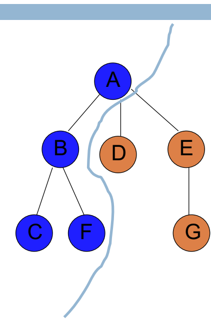
D
E

Frontier?

Stack

104

Tree DFS

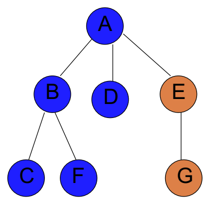


D
E

Stack

105

Tree DFS

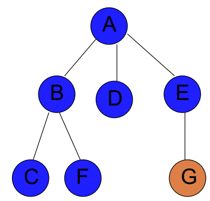


E

Stack

106

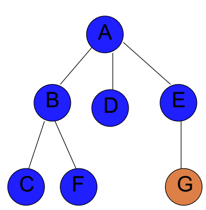
Tree DFS



Stack

107

Tree DFS

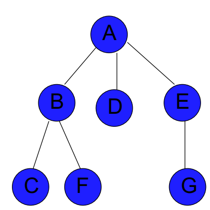


G

Stack

108

Tree DFS



Stack

109

DFS on graphs

```
DFS(G)
  for all v ∈ V
      visited[u] ← false
  for all v ∈ V
      if !visited[v]
          DFS-Visit(v)

DFS-Visit(u)
  visited[u] ← true
  PreVisit(u)
  for all edges (u, v) ∈ E
      if !visited[v]
          DFS-Visit(v)
  PostVisit(u)
```

110

DFS on graphs

```
DFS(G)
  for all v ∈ V
      visited[u] ← false
  for all v ∈ V
      if !visited[v]
          DFS-Visit(v)

DFS-Visit(u)
  visited[u] ← true
  PreVisit(u)
  for all edges (u, v) ∈ E
      if !visited[v]
          DFS-Visit(v)
  PostVisit(u)
```

mark all nodes as not visited

111

DFS on graphs

```
DFS(G)
  for all v ∈ V
      visited[u] ← false
  for all v ∈ V
      if !visited[v]
          DFS-Visit(v)

DFS-Visit(u)
  visited[u] ← true
  PreVisit(u)
  for all edges (u, v) ∈ E
      if !visited[v]
          DFS-Visit(v)
  PostVisit(u)
```

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

112

DFS on graphs

```
DFS(G)
  for all v ∈ V
      visited[u] ← false
  for all v ∈ V
      if !visited[v]
          DFS-Visit(v)
```

What happened to the stack?

```
DFS-Visit(u)
  visited[u] ← true
  PreVisit(u)
  for all edges (u, v) ∈ E
      if !visited[v]
          DFS-Visit(v)
  PostVisit(u)
```

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

113

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

114

Is DFS correct?

Does DFS visit all of the nodes in a graph?

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

115

Running time?

Like BFS

- Visits each node exactly once
- Processes each edge exactly twice (for an undirected graph)
- $\theta(|V|+|E|)$

116

Connectedness

Given an undirected graph, for every node $u \in V$, 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: $O(|V| + |E|)$

117