Graphs

David Kauchak
cs302
Spring 2012

Admin

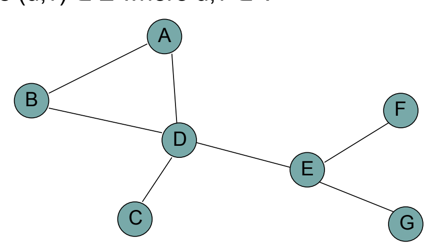
- HW 11 and 12
  - You can submit revised solutions to any problem you missed
  - Also submit your original homework
  - I'll give you up to half of the points taken off
  - Because I've given comments/feedback, make sure you explain *why* for simple questions (like run-time)
  - Also, I will expect your answers to be very clear and precise
- HW 14
  - more dynamic programming
  - will involve some programming (you may use any language installed on the lab machines)
  - may work with a partner: you and your partner must always be there when you're working on the assignment

Admin

- Registration
- Lunch today!

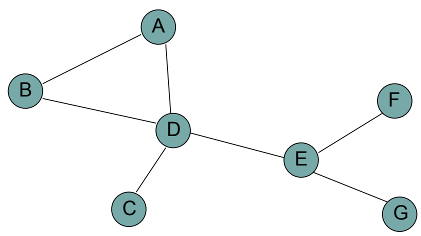
Graphs

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



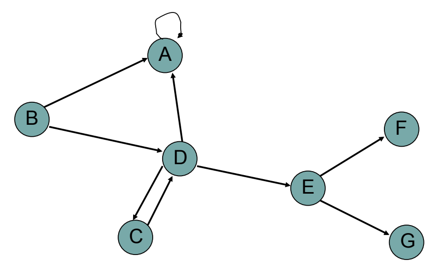
Different types of graphs

- Undirected – edges do not have a direction



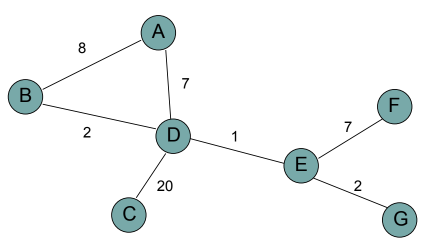
Different types of graphs

- Directed – edges **do** have a direction



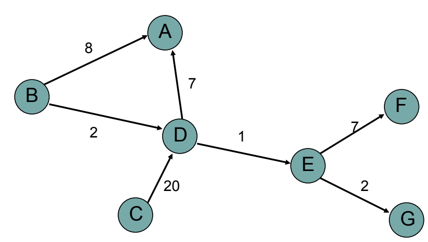
Different types of graphs

- Weighted – edges have an associated weight



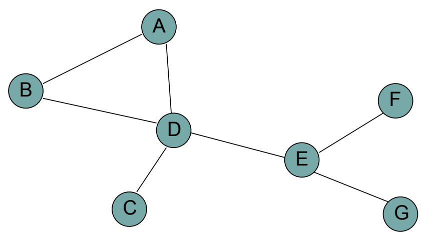
Different types of graphs

- Weighted – edges have an associated weight



Terminology

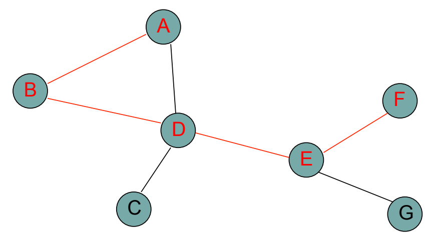
- Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$



Terminology

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

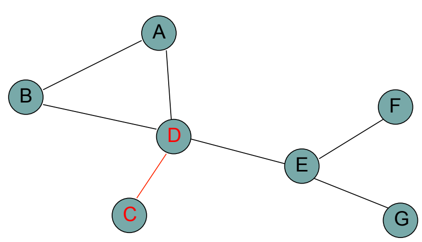
{A, B, D, E, F}



Terminology

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

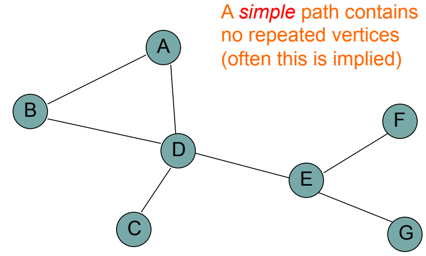
{C, D}



Terminology

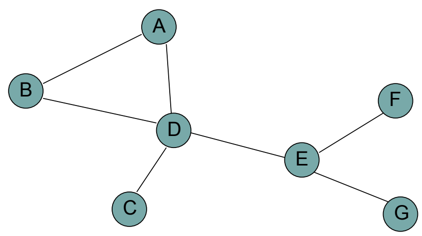
- Path – A path is a list of vertices $p_1, p_2, \ldots p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$

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



Terminology

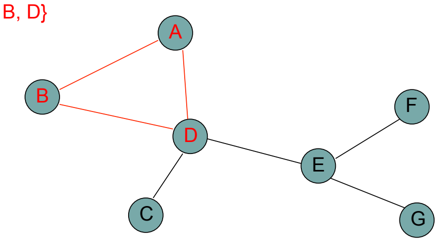
- Cycle – A subset of the edges that form a path such that the first and last node are the same



Terminology

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

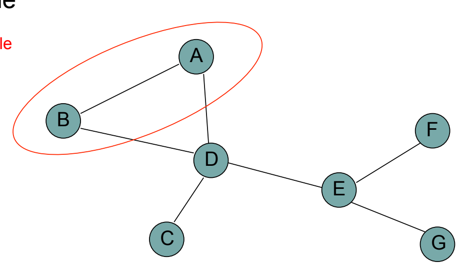
{A, B, D}



Terminology

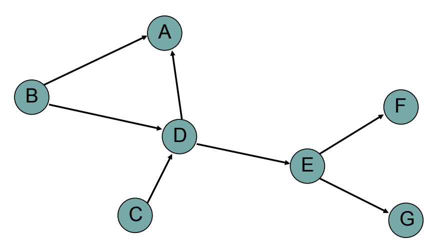
- Cycle – A subset of the edges that form a path such that the first and last node are the same

not a cycle



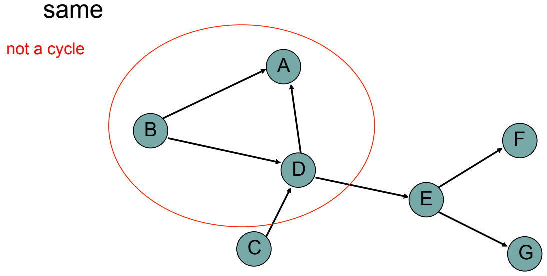
Terminology

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



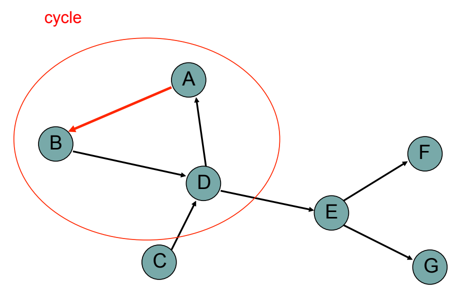
Terminology

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



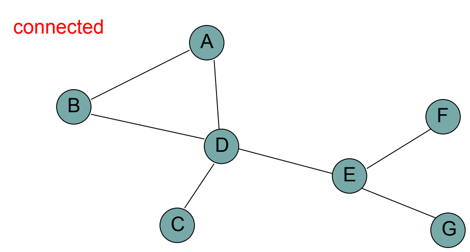
Terminology

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



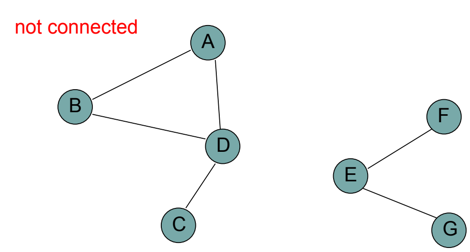
Terminology

- Connected – every pair of vertices is connected by a path



Terminology

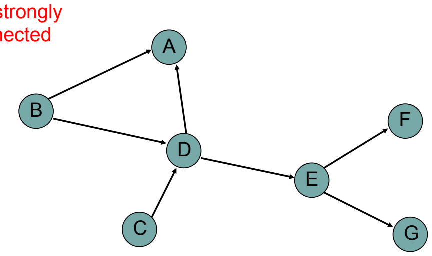
- Connected (undirected graphs) – every pair of vertices is connected by a path



Terminology

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

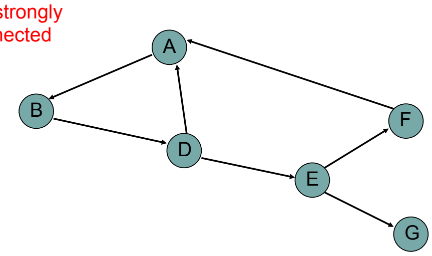
not strongly connected



Terminology

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

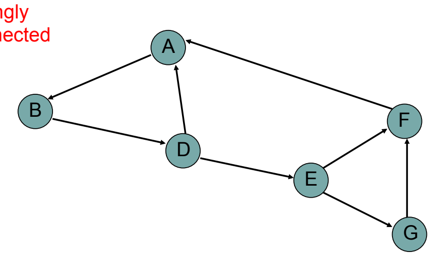
not strongly connected



Terminology

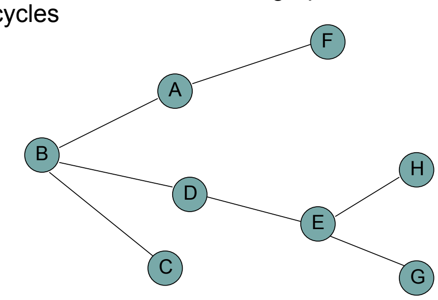
- Strongly connected (directed graphs) – Every two vertices are reachable by a path

strongly connected



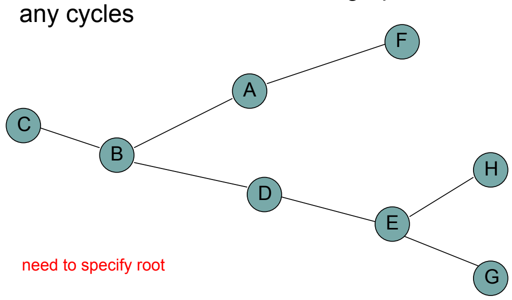
Different types of graphs

- Tree – connected, undirected graph without any cycles



Different types of graphs

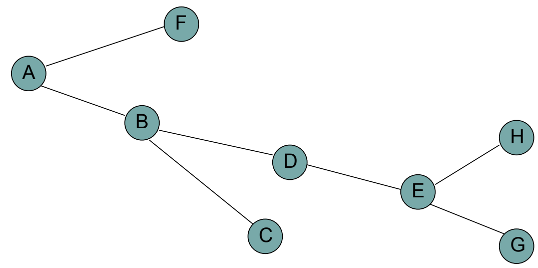
- Tree – connected, undirected graph without any cycles



need to specify root

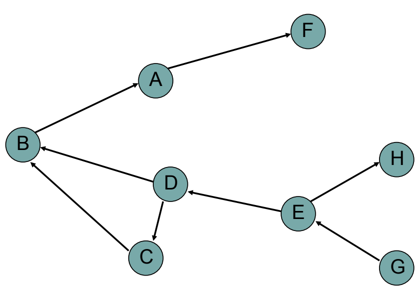
Different types of graphs

- Tree – connected, undirected graph without any cycles



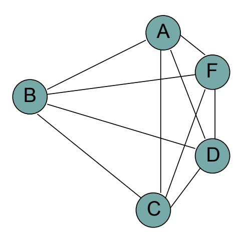
Different types of graphs

- DAG – directed, acyclic graph



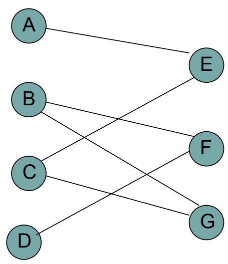
Different types of graphs

- 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 ∈ X and a vertex v ∈ Y



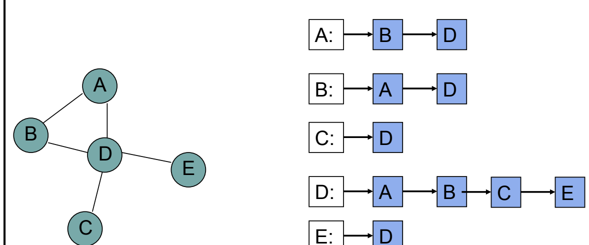
When do we see graphs in real life problems?

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

Representing graphs

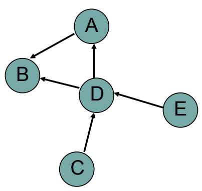
Representing graphs

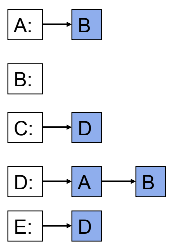
- Adjacency list – Each vertex u ∈ V contains an adjacency list of the set of vertices v such that there exists an edge (u,v) ∈ E



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

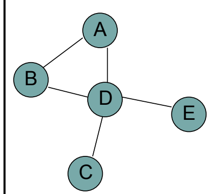




Representing graphs

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

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

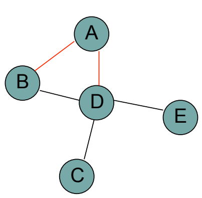


	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

Representing graphs

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

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

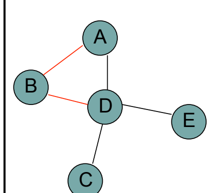


	A	B	C	D	E
A	0	**1**	0	**1**	0
B	1	0	0	1	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

Representing graphs

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

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

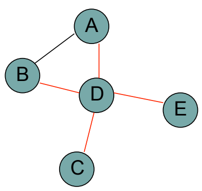


	A	B	C	D	E
A	0	1	0	1	0
B	**1**	0	0	**1**	0
C	0	0	0	1	0
D	1	1	1	0	1
E	0	0	0	1	0

Representing graphs

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

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

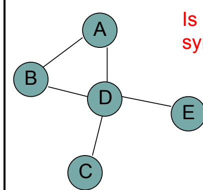


	A	B	C	D	E
A	0	1	0	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	**1**	**1**	**1**	0	**1**
E	0	0	0	1	0

Representing graphs

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

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



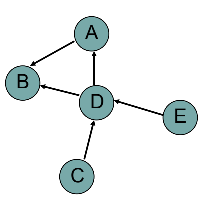
Is it always symmetric?

	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

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

Adjacency list vs. adjacency matrix

Adjacency list

- Sparse graphs (e.g. web)
- Space efficient
- Must traverse the adjacency list to discover is an edge exists

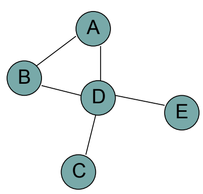
Adjacency matrix

- Dense graphs
- Constant time lookup to discover if 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

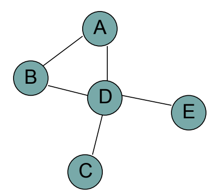
- Rather than using an adjacency list, use an adjacency hashtable



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

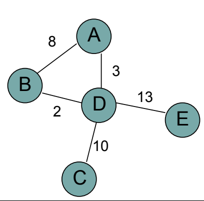


A:	hashtable [B,D]
B:	hashtable [A,D]
C:	hashtable [D]
D:	hashtable [A,B,C,E]
E:	hashtable [D]

Weighted graphs

- Adjacency list
  - store the weight as an additional field in the list

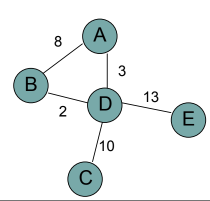
A: → B:8 → D:3



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

Graph algorithms/questions

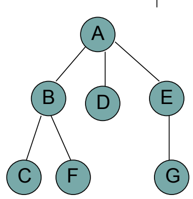
- Graph traversal (BFS, DFS)
- Shortest path from a to b
  - unweighted
  - weighted positive weights
  - negative/positive weights
- Minimum spanning trees
- Are all nodes in the graph connected?
- Is the graph bipartite?
- hw15 and hw16 ☺

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)
```

Tree BFS

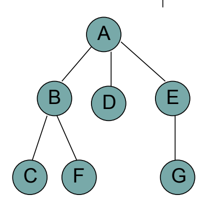
```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
      v ← Dequeue(Q)
      Visit(v)
      for all c ∈ Children(v)
          Enqueue(Q, c)
```



Q:

Tree BFS

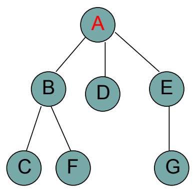
```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
      v ← Dequeue(Q)
      Visit(v)
      for all c ∈ Children(v)
          Enqueue(Q, c)
```



Q: A

Tree BFS

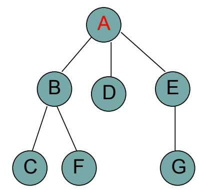
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q:

Tree BFS

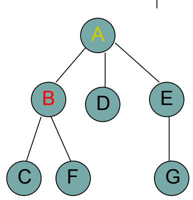
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: B, D, E

Tree BFS

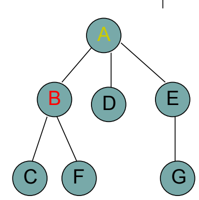
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: D, E

Tree BFS

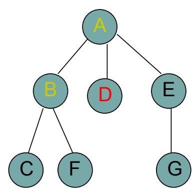
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```



Q: D, E, C, F

Tree BFS

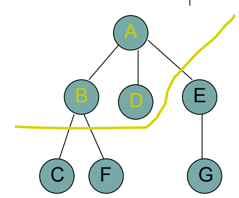
```
TreeBFS(T)
  Enqueue(Q, Root(T))
  while !Empty(Q)
      v ← Dequeue(Q)
      Visit(v)
      for all c ∈ Children(v)
          Enqueue(Q, c)
```



Q: E, C, F

Tree BFS

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

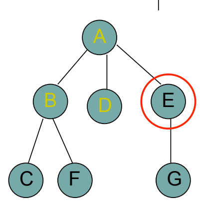


Q: E, C, F

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

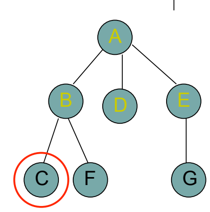
Tree BFS

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



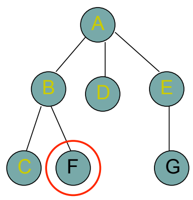
Tree BFS

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



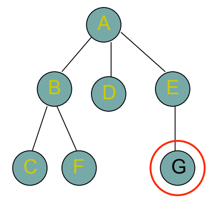
Tree BFS

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



Tree BFS

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



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 ← Dequeue(Q)
    Visit(v)
    for all c ∈ Children(v)
        Enqueue(Q, c)
```

Tree BFS

- Does it visit all of the nodes?

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

Running time of Tree BFS

- Adjacency list
  - How many times does it visit each vertex?
  - How many times is each edge traversed?
  - O(|V|+|E|)
- Adjacency matrix
  - For each vertex visited, how much work is done?
  - O(|V|^2)

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

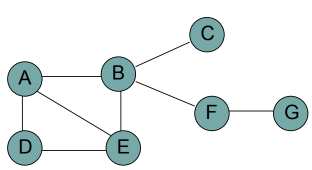
BFS Recursively

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)
```

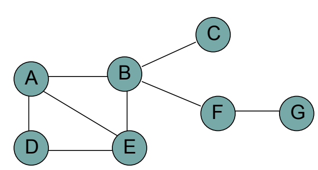
BFS for graphs

- What needs to change for graphs?
- Need to make sure we don't visit a node multiple times



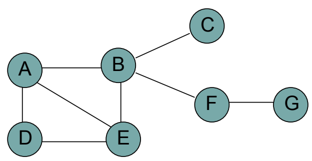

```
BFS(G, s)
  for each v ∈ V
        dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
              if dist[v] = ∞
                    Enqueue(Q, v)
                    dist[v] ← dist[u] + 1
```

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



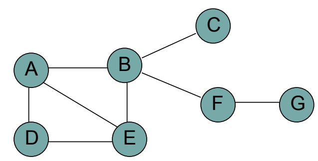
```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

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



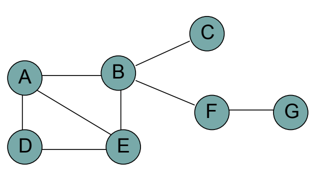

```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

set all nodes as unseen



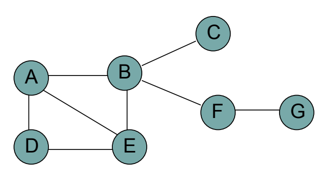

```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

check if the node has been seen

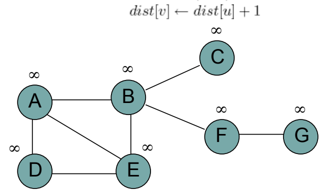



```
BFS(G, s)
  for each v ∈ V
      dist[v] = ∞
  dist[s] = 0
  Enqueue(Q, s)
  while !Empty(Q)
      u ← Dequeue(Q)
      Visit(u)
      for each edge (u, v) ∈ E
          if dist[v] = ∞
              Enqueue(Q, v)
              dist[v] ← dist[u] + 1
```

set the node as seen and record distance

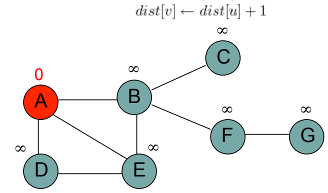


```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```



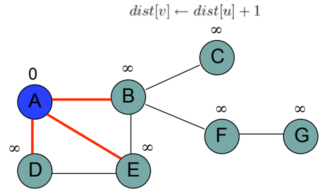

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: A



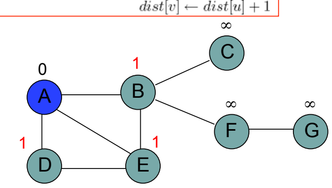

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q:



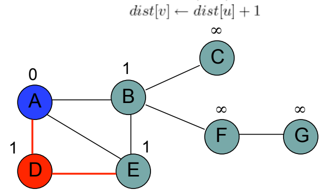

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Q: D, E, B



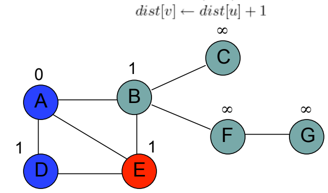
```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
    u ← Dequeue(Q)
    Visit(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            Enqueue(Q, v)
            dist[v] ← dist[u] + 1
```

Q: E, B



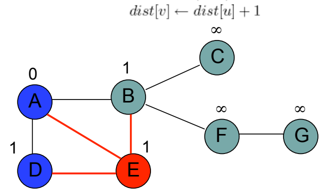

```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
    u ← Dequeue(Q)
    Visit(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            Enqueue(Q, v)
            dist[v] ← dist[u] + 1
```

Q: B



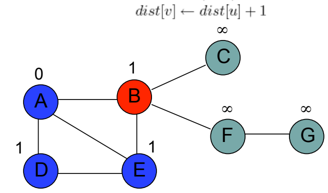

```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
    u ← Dequeue(Q)
    Visit(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            Enqueue(Q, v)
            dist[v] ← dist[u] + 1
```

Q: B



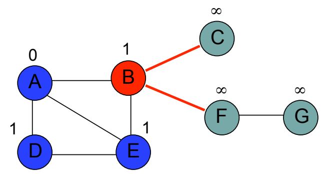

```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
    u ← Dequeue(Q)
    Visit(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            Enqueue(Q, v)
            dist[v] ← dist[u] + 1
```

Q:



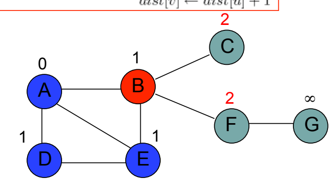
```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```

Q:

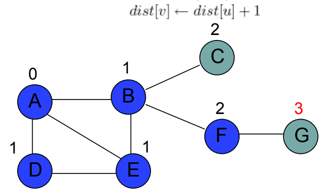



```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```

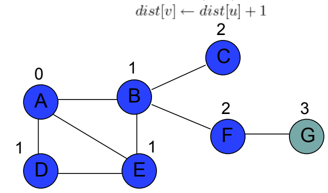
Q: F, C



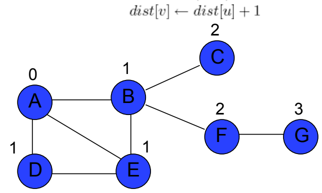

```
BFS(G, s)
for each v ∈ V
    dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
    u ← DEQUEUE(Q)
    VISIT(u)
    for each edge (u, v) ∈ E
        if dist[v] = ∞
            ENQUEUE(Q, v)
            dist[v] ← dist[u] + 1
```




```
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
```

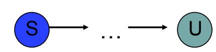


```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
Enqueue(Q, s)
while !Empty(Q)
        u ← Dequeue(Q)
        Visit(u)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        Enqueue(Q, v)
                        dist[v] ← dist[u] + 1
```



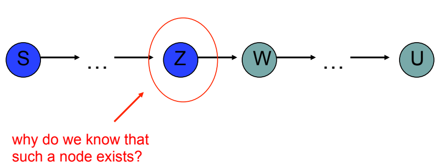
Is BFS correct?

- Does it visit all nodes reachable from the starting node?
- Can you prove it?
- Assume we "miss" some node 'u', i.e. a path exists, but we don't visit 'u'



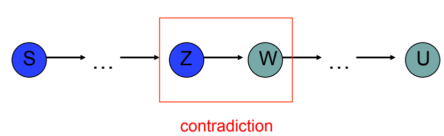
Is BFS correct?

- Does it visit all nodes reachable from the starting node?
- Can you prove it?
- Find the last node along the path to 'u' that was visited



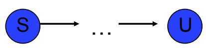
Is BFS correct?

- Does it visit all nodes reachable from the starting node?
- Can you prove it?
- We visited 'z' but not 'w', which is a contradiction, given the pseudocode



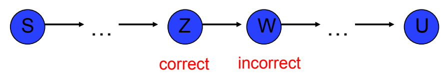
Is BFS correct?

- Does it correctly label each node with the shortest distance from the starting node?
- Assume the algorithm labels a node with a longer distance. Call that node 'u'



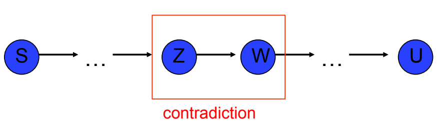
Is BFS correct?

- Does it correctly label each node with the shortest distance from the starting node?
- Find the last node in the path with the correct distance



Is BFS correct?

- Does it correctly label each node with the shortest distance from the starting node?
- Find the last node in the path with the correct distance



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)
```

Runtime of BFS

- Adjacency list: O(|V| + |E|)
- Adjacency matrix: O(|V|²)

```
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
```

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)
```

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)
```

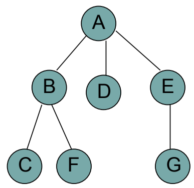
```
TreeBFS(T)
Enqueue(Q, Root(T))
while !Empty(Q)
        v ← Dequeue(Q)
        Visit(v)
        for all c ∈ Children(v)
                Enqueue(Q, c)
```

Depth First Search (DFS)

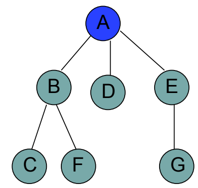
```
TreeDFS(T)
Push(S, Root(T))
while !Empty(S)
        v ← Pop(S)
        Visit(v)
        for all c ∈ Children(v)
                Push(S, c)
```

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

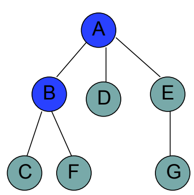
Tree DFS



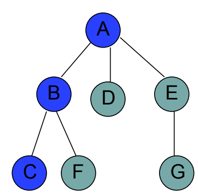
Tree DFS



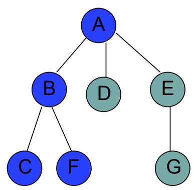
Tree DFS



Tree DFS

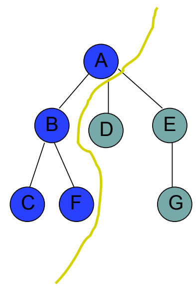


Tree DFS

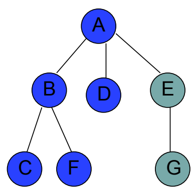


Frontier?

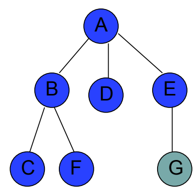
Tree DFS



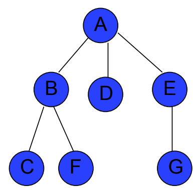
Tree DFS



Tree DFS



Tree DFS



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)
```

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

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

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)
```

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

Is DFS correct?

- Does DFS visit all of the nodes in a graph?

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

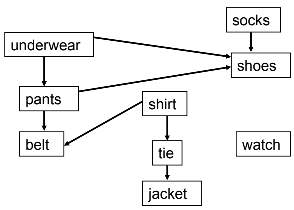
Running time?

Like BFS

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

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