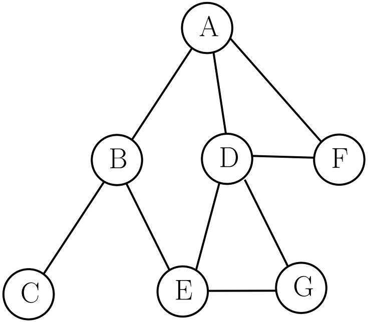
Learning Goals

- Review steps of breadth first search (BFS) and depth first search (DFS) algorithms
- Running time of BFS and DFS
- Properties of BFS and DFS trees

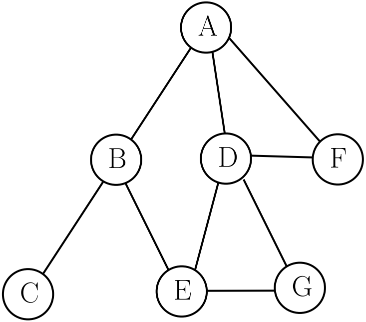
Graph Traversal

Problem: Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, decide whether there is a path connecting s and t.



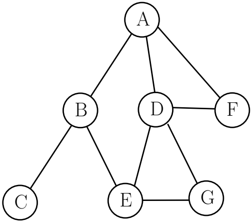
Graph Traversal

Problem: Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, decide whether there is a path connecting s and t.



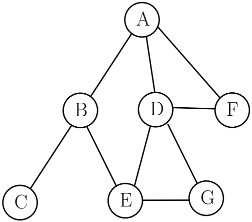
- Breadth First Search (BFS): mark s as visited, imeediately mark all neighbors of s as visited, and THEN recursively do the same for all the nodes that are newly marked as visited.

BFS example

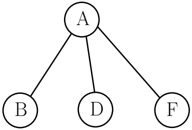


The given graph

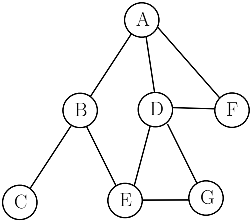
BFS example



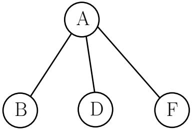
The given graph

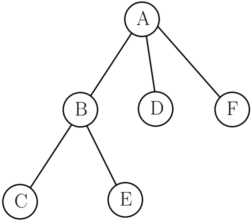


BFS example

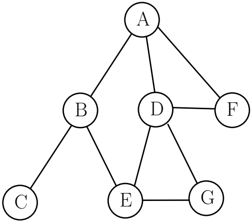


The given graph

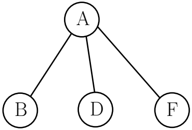


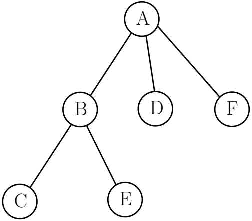


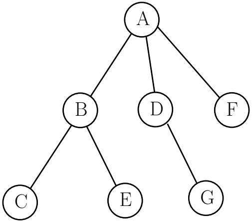
BFS example



The given graph

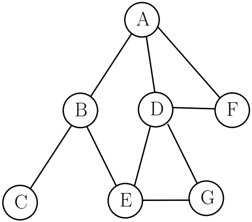




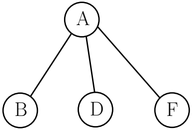


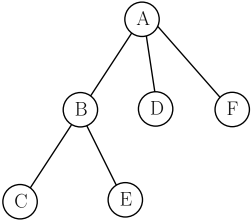
The BFS tree

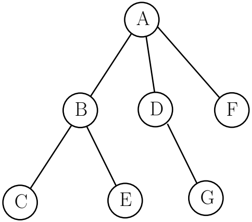
BFS example



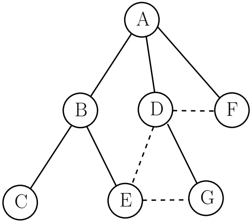
The given graph







The BFS tree



BFS tree with edges in the original graph.

BFS implementation

- Create a queue containing starting point s. Mark s as visited.

BFS implementation

- Create a queue containing starting point s. Mark s as visited.
- As long as the queue is not empty:
  - Take the node in the front of queue, mark all its neighbors as visited, and append those newly marked to the queue.

BFS implementation

- Create a queue containing starting point s. Mark s as visited.
- As long as the queue is not empty:
  - Take the node in the front of queue, mark all its neighbors as visited, and append those newly marked to the queue.
- In general this algorithm visits all the nodes in the same connected component as s.

BFS implementation

- Create a queue containing starting point s. Mark s as visited.
- As long as the queue is not empty:
  - Take the node in the front of queue, mark all its neighbors as visited, and append those newly marked to the queue.
- In general this algorithm visits all the nodes in the same connected component as s.
- s-t path: whenever t is marked visited, we find a path from s to t; if algorithm terminates without t being visited, there is no path connecting s and t.

BFS implementation

- Create a queue containing starting point s. Mark s as visited.
- As long as the queue is not empty:
  - Take the node in the front of queue, mark all its neighbors as visited, and append those newly marked to the queue.
- In general this algorithm visits all the nodes in the same connected component as s.
- s-t path: whenever t is marked visited, we find a path from s to t; if algorithm terminates without t being visited, there is no path connecting s and t.
- Running time: $O(n + m)$.
  - Throughout the course we use n to denote the number of nodes in a graph, and m the number of edges.

BFS implementation

- Create a queue containing starting point s. Mark s as visited.
- As long as the queue is not empty:
  - Take the node in the front of queue, mark all its neighbors as visited, and append those newly marked to the queue.
- In general this algorithm visits all the nodes in the same connected component as s.
- s-t path: whenever t is marked visited, we find a path from s to t; if algorithm terminates without t being visited, there is no path connecting s and t.
- Running time: $O(n+m)$.
  - Throughout the course we use n to denote the number of nodes in a graph, and m the number of edges.
  - Each node is appended to the queue at most once, and each edge is followed at most twice. (Assuming an adjancy list representation of the graph.)

BFS implementation

- Create a queue containing starting point s. Mark s as visited.
- As long as the queue is not empty:
  - Take the node in the front of queue, mark all its neighbors as visited, and append those newly marked to the queue.
- In general this algorithm visits all the nodes in the same connected component as s.
- s-t path: whenever t is marked visited, we find a path from s to t; if algorithm terminates without t being visited, there is no path connecting s and t.
- Running time: $O(n + m)$.
  - Throughout the course we use n to denote the number of nodes in a graph, and m the number of edges.
  - Each node is appended to the queue at most once, and each edge is followed at most twice. (Assuming an adjancy list representation of the graph.)
  - This is called *linear time*.

More on BFS

- It is straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors. This solves the s-t connectivity problem.

More on BFS

- It is straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors. This solves the s-t connectivity problem.

Definition (Layers of a BFS tree)

The first layer, L_1, of a BFS tree is the singleton set of the starting node s; then the $(i+1)^{\text{-st}}$ layer L_{i+1} is the set of nodes that are newly marked visited when the algorithm is processing a node in L_i.

More on BFS

- It is straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors. This solves the s-t connectivity problem.

Definition (Layers of a BFS tree)

The first layer, L_1, of a BFS tree is the singleton set of the starting node s; then the $(i+1)^{\text{-st}}$ layer L_{i+1} is the set of nodes that are newly marked visited when the algorithm is processing a node in L_i.

proposition

The shortest path from s to any node in L_i has length $i-1$ (i.e., has $i-1$ edges).

More on BFS

- It is straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors. This solves the s-t connectivity problem.

Definition (Layers of a BFS tree)

The first layer, L_1, of a BFS tree is the singleton set of the starting node s; then the $(i+1)^{\text{-st}}$ layer L_{i+1} is the set of nodes that are newly marked visited when the algorithm is processing a node in L_i.

proposition

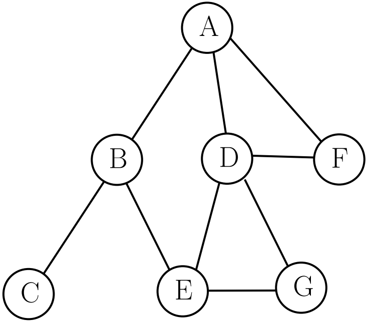
The shortest path from s to any node in L_i has length $i-1$ (i.e., has $i-1$ edges).

proposition

For any $(u,v) \in E$, in a BFS tree if u is in L_i and v in L_j, then $|i-j| \leq 1$.

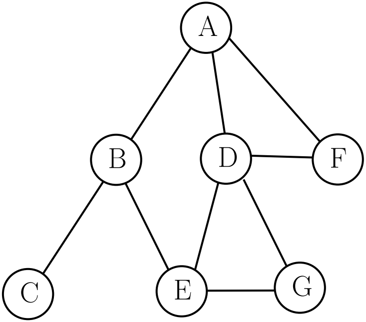
Depth First Search

Problem: Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, decide whether there is a path connecting s and t.



Depth First Search

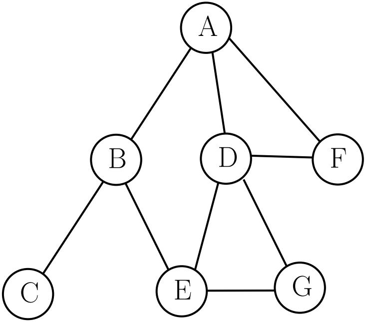
Problem: Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, decide whether there is a path connecting s and t.



- Depth First Search (DFS) idea: go headlong till a dead end, then backtrack.

Depth First Search

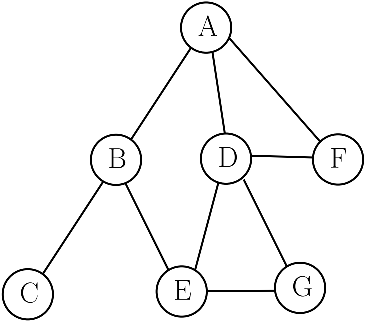
Problem: Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, decide whether there is a path connecting s and t.



- Depth First Search (DFS) idea: go headlong till a dead end, then backtrack.
- Mark s as visited, then for each neighbor of s, if it is unmarked, recursively do the same.

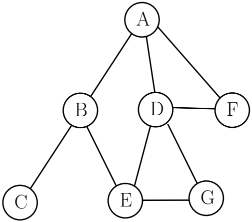
Depth First Search

Problem: Given an undirected graph $G = (V, E)$ and two nodes $s, t \in V$, decide whether there is a path connecting s and t.



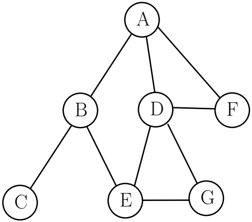
- Depth First Search (DFS) idea: go headlong till a dead end, then backtrack.
- Mark s as visited, then for each neighbor of s, if it is unmarked, recursively do the same.
- In other words, process the first neighbor of s completely before going on to the next unvisited neighbor.

DFS example

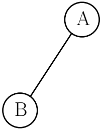


The given graph

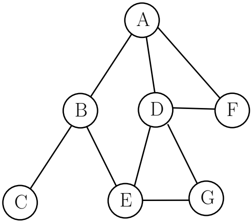
DFS example



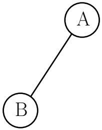
The given graph

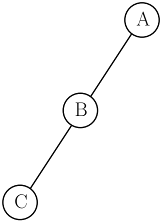


DFS example

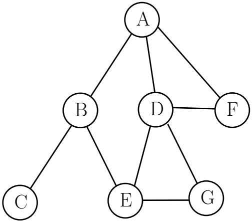


The given graph

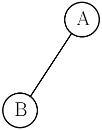


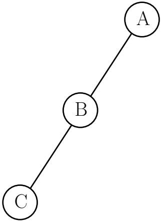


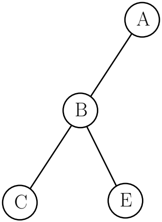
DFS example



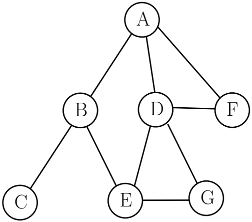
The given graph



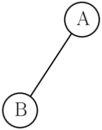


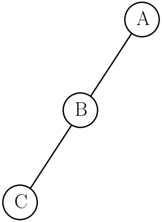


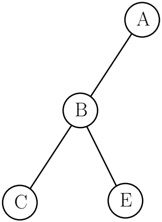
DFS example

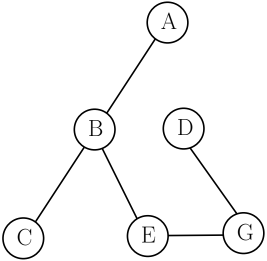


The given graph

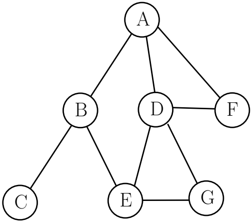




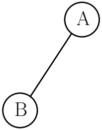


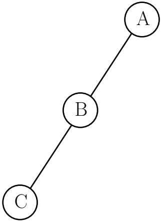


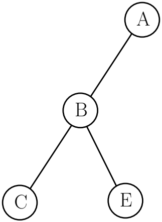
DFS example

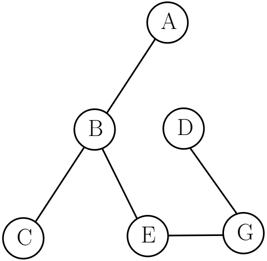


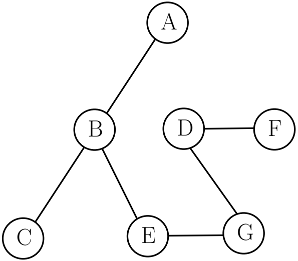
The given graph





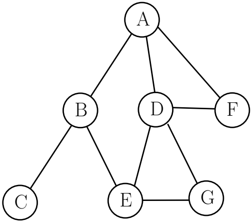




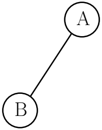


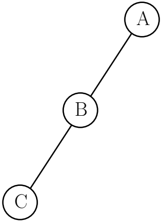
A DFS tree

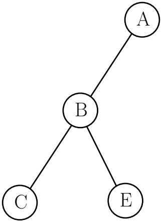
DFS example

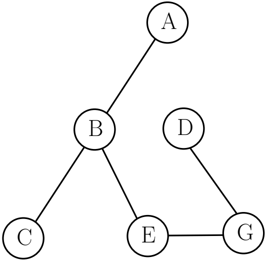


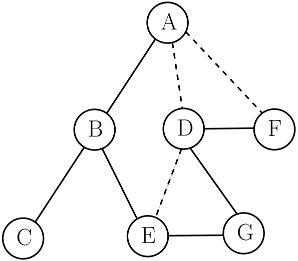
The given graph











A DFS tree

DFS implementation

- Create a stack containing starting point s.

DFS implementation

- Create a stack containing starting point s.
- As long as the stack is not empty:
  - Pop the node on top of the stack. If it is unmarked, mark it as visited, then put all its neighbors onto the stack.

DFS implementation

- Create a stack containing starting point s.
- As long as the stack is not empty:
  - Pop the node on top of the stack. If it is unmarked, mark it as visited, then put all its neighbors onto the stack.
- DFS also visits all the nodes in the same connected component as s.

DFS implementation

- Create a stack containing starting point s.
- As long as the stack is not empty:
  - Pop the node on top of the stack. If it is unmarked, mark it as visited, then put all its neighbors onto the stack.
- DFS also visits all the nodes in the same connected component as s.
- Running time: $O(n+m)$.
  - Each edge is followed at most twice; each "stacking" follows from an edge visit.

More on DFS

- It is also straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors.

More on DFS

- It is also straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors.

proposition

If $(u, v) \in E$ in an undirected graph, let T be a DFS tree with $u, v \in T$. Then either u is an ancestor of v or v is an ancestor of u.

More on DFS

- It is also straightforward to generalize the algorithm to directed graphs — just follow outgoing edges when visiting neighbors.

proposition

If $(u, v) \in E$ in an undirected graph, let T be a DFS tree with $u, v \in T$. Then either u is an ancestor of v or v is an ancestor of u.

Proof.

Either u is added to T before v or v before u. If u is added first, when processing u, the algorithm checks the edge (u, v) before backtracking. If v is unmarked at the time, (u, v) is added to T, and v is a child of u; if v is already marked visited at the time, it is marked between the start and end of the processing of u, and hence is a descendant of u.
The other case (when v is added first) follows the same argument. □