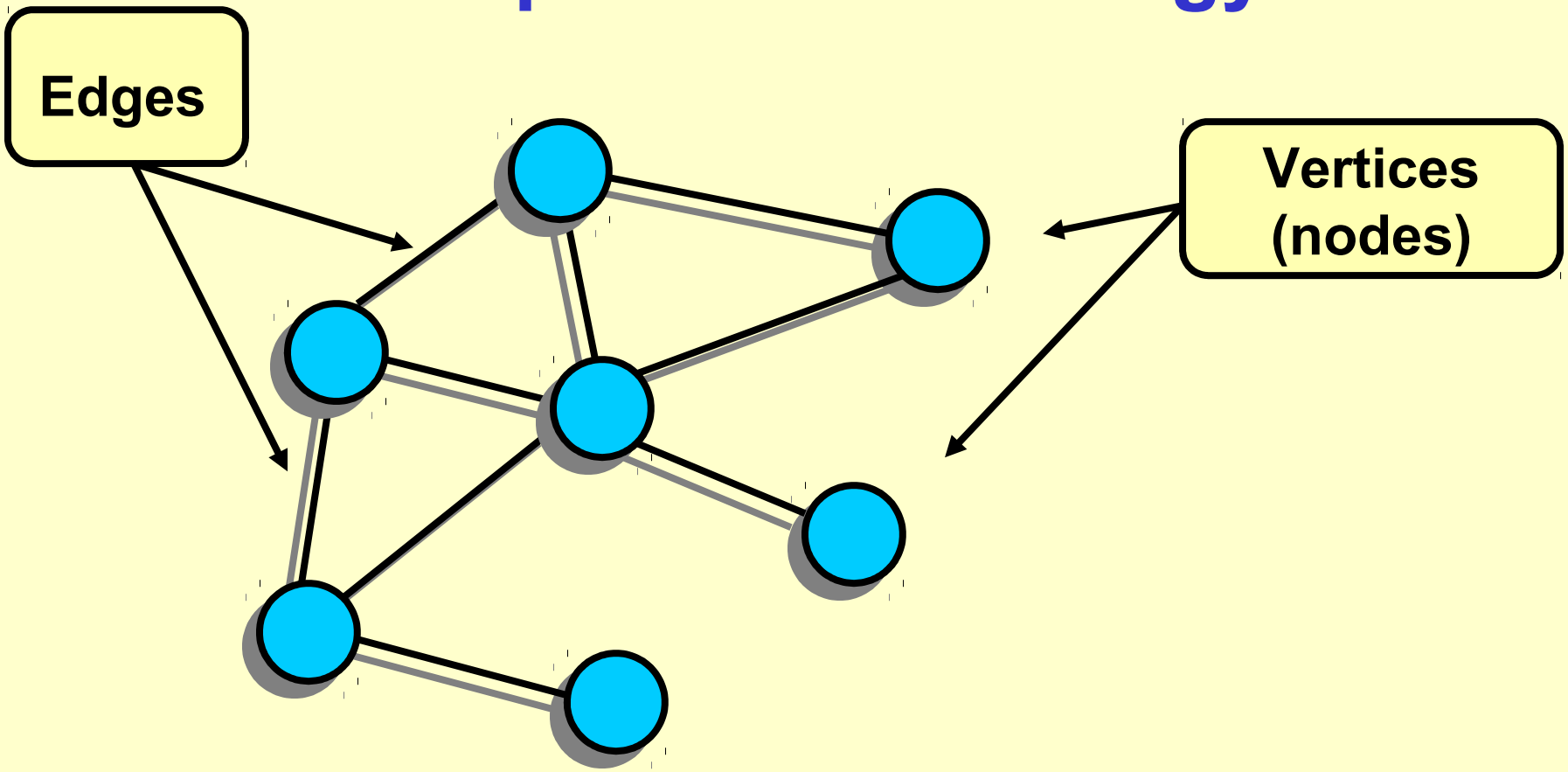
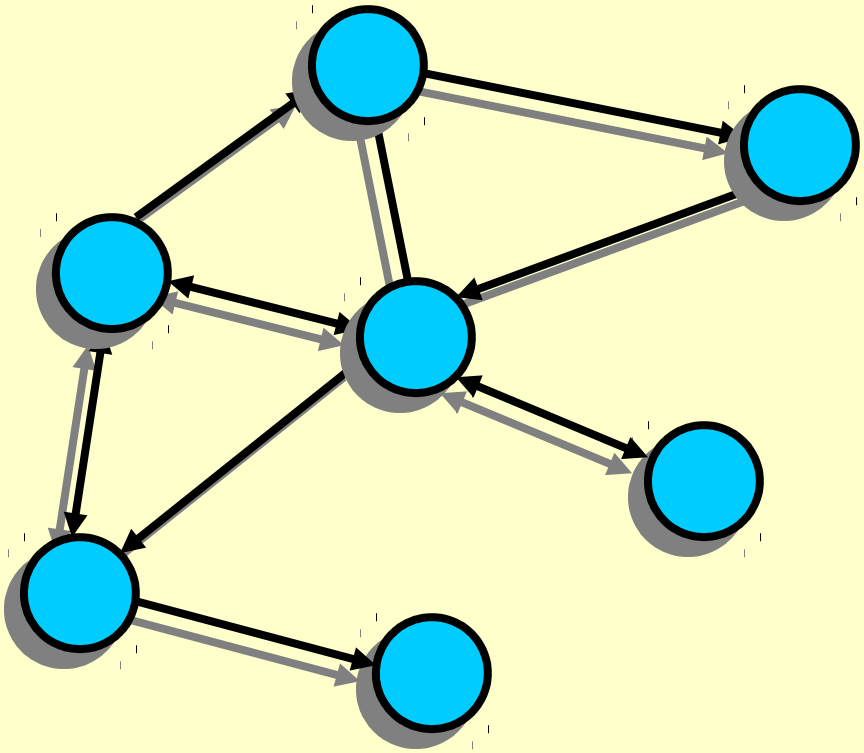
Graphs and Infinite Loops

Graphs :: Terminology



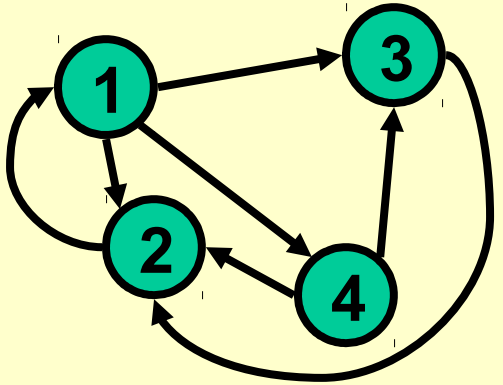
An undirected graph

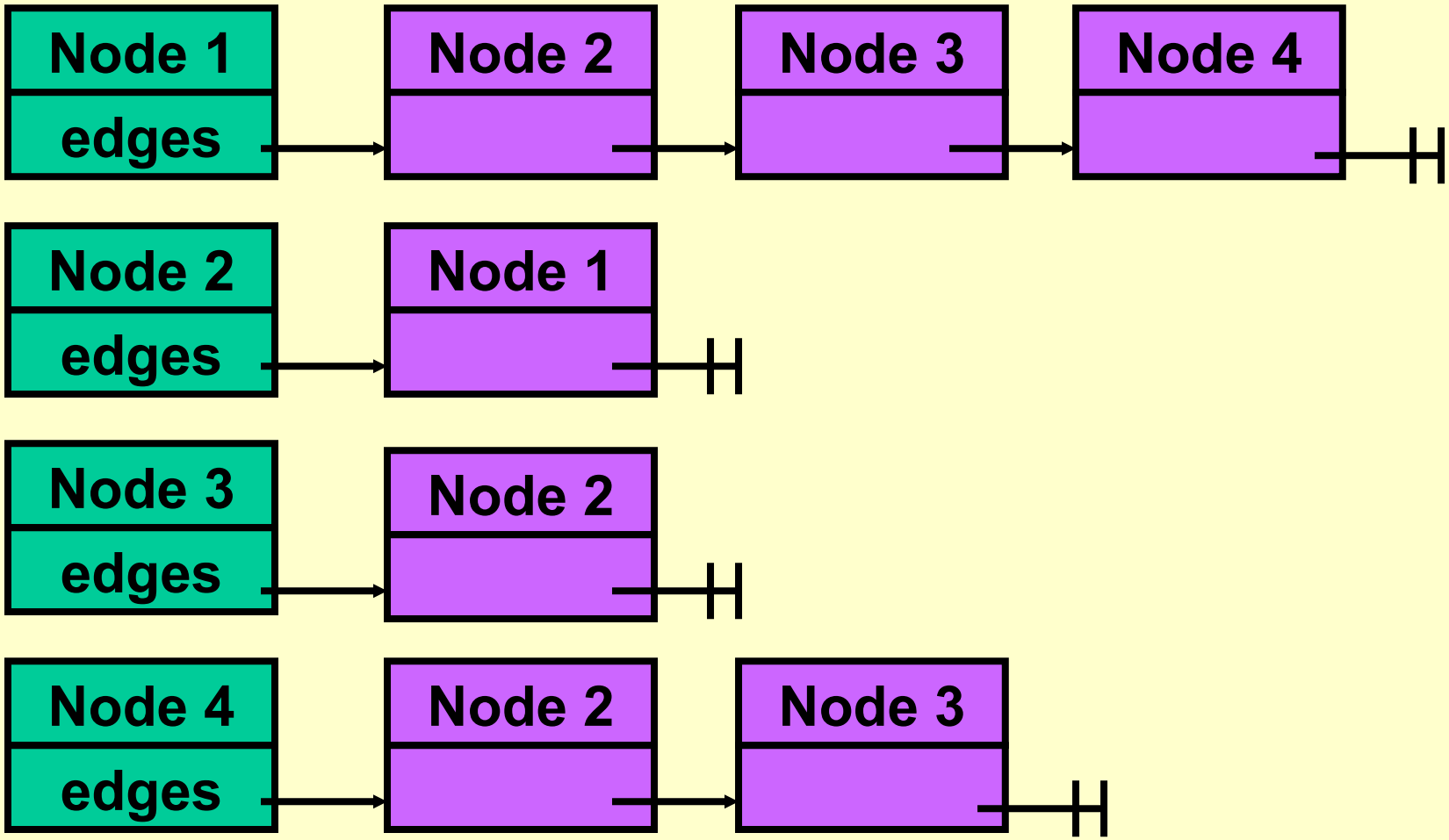
Graphs :: Terminology



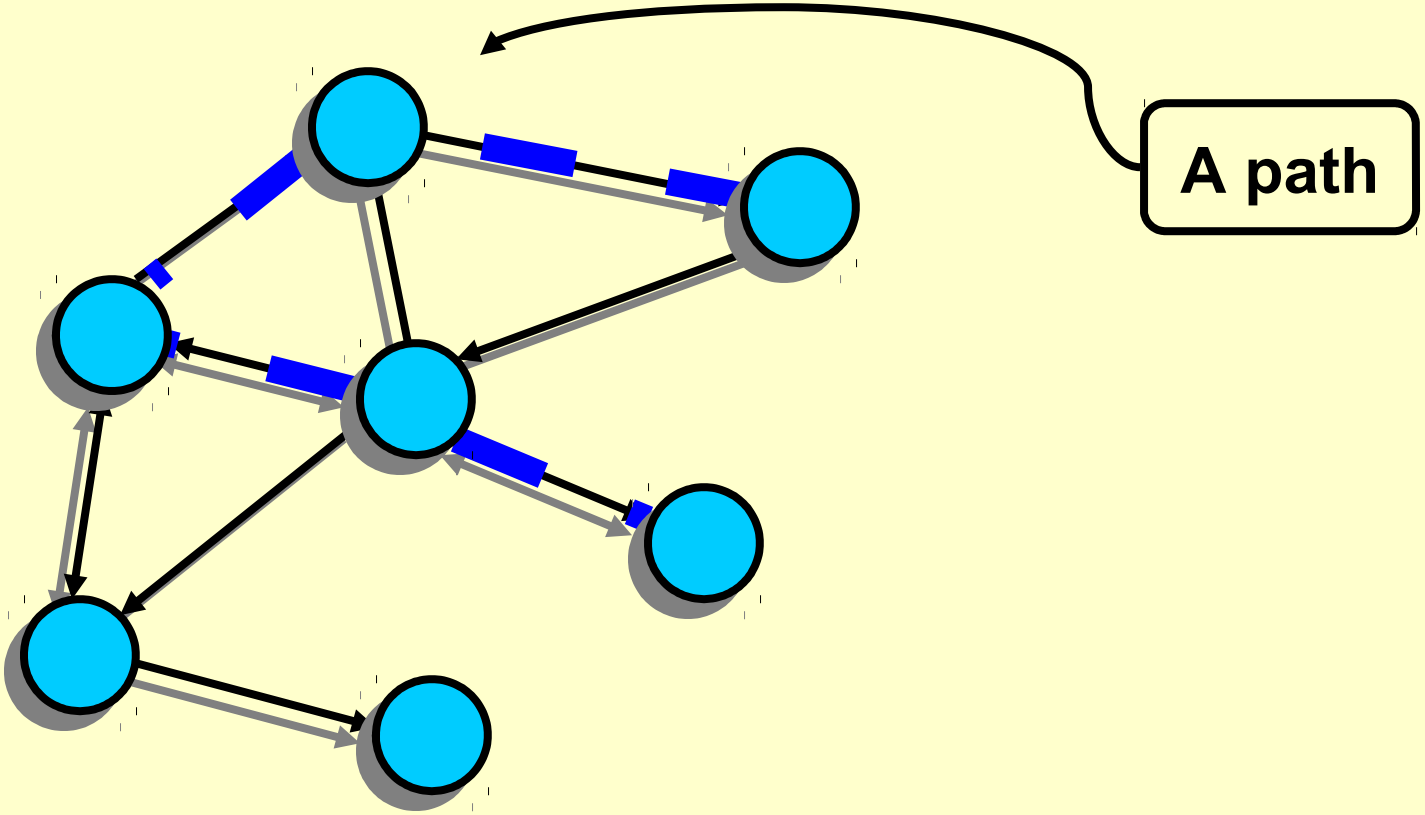
A directed graph

Internally…


1
3
2
4


Node 1
edges
Node 2
Node 3
Node 4
Node 2
edges
Node 1
Node 3
edges
Node 2
Node 4
edges
Node 2
Node 3

Graphs :: Terminology

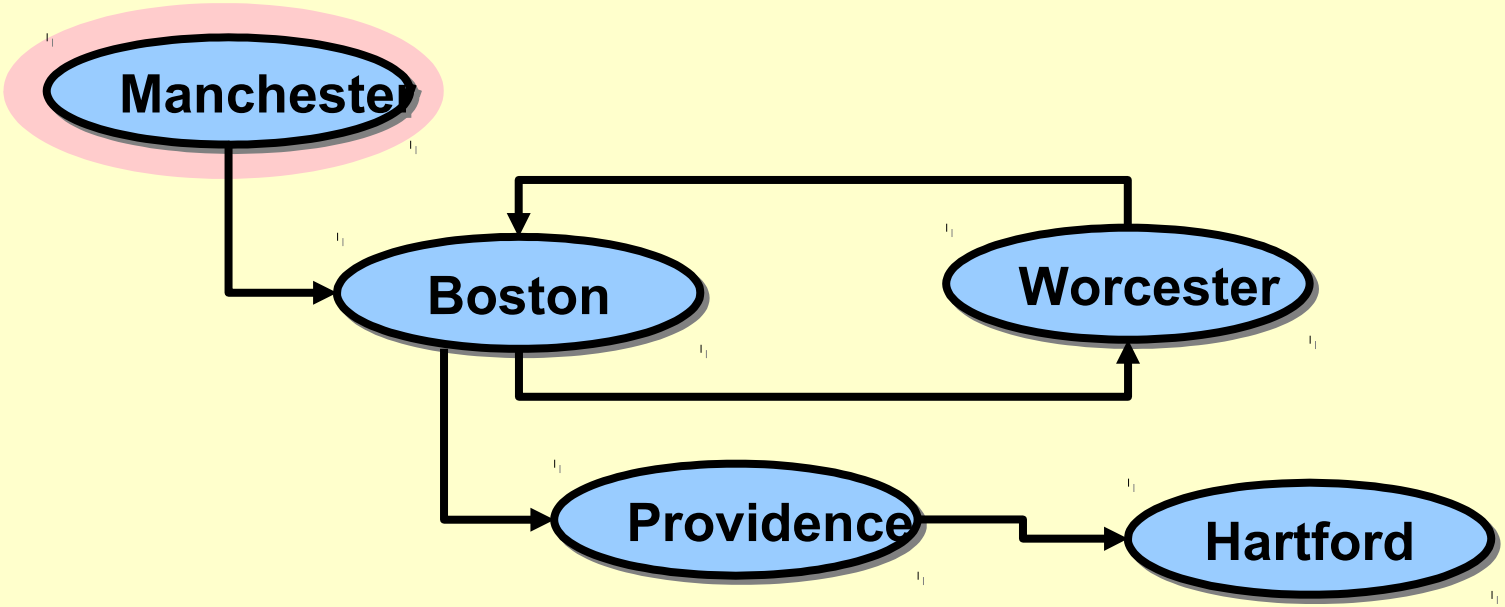


A directed graph

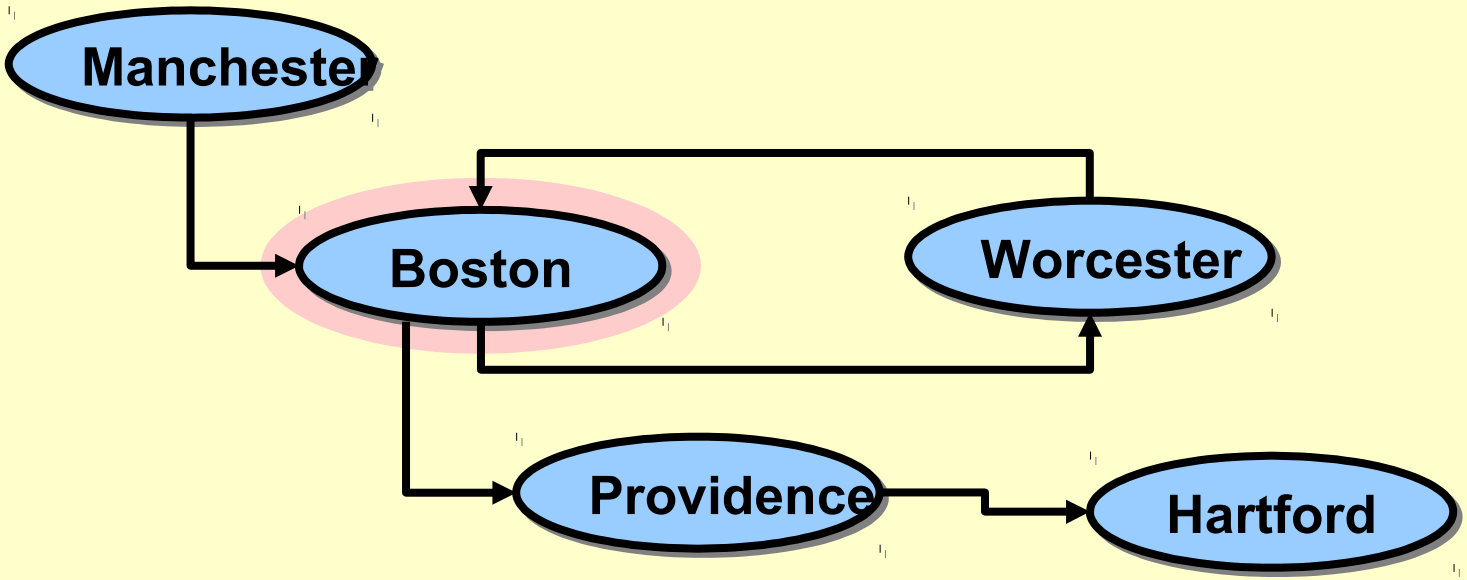
```
public class Node {
	private String cityname;
	private LinkedList<Node> getsTo;

	boolean hasRoute(Node to) {
		if (this.equals(to))
			return true;
		else {
			for (Node c : this.getsTo) {
				if (c.hasRoute(to)) {
					return true;
				}
			}
			return false;
		}
	}
}
```

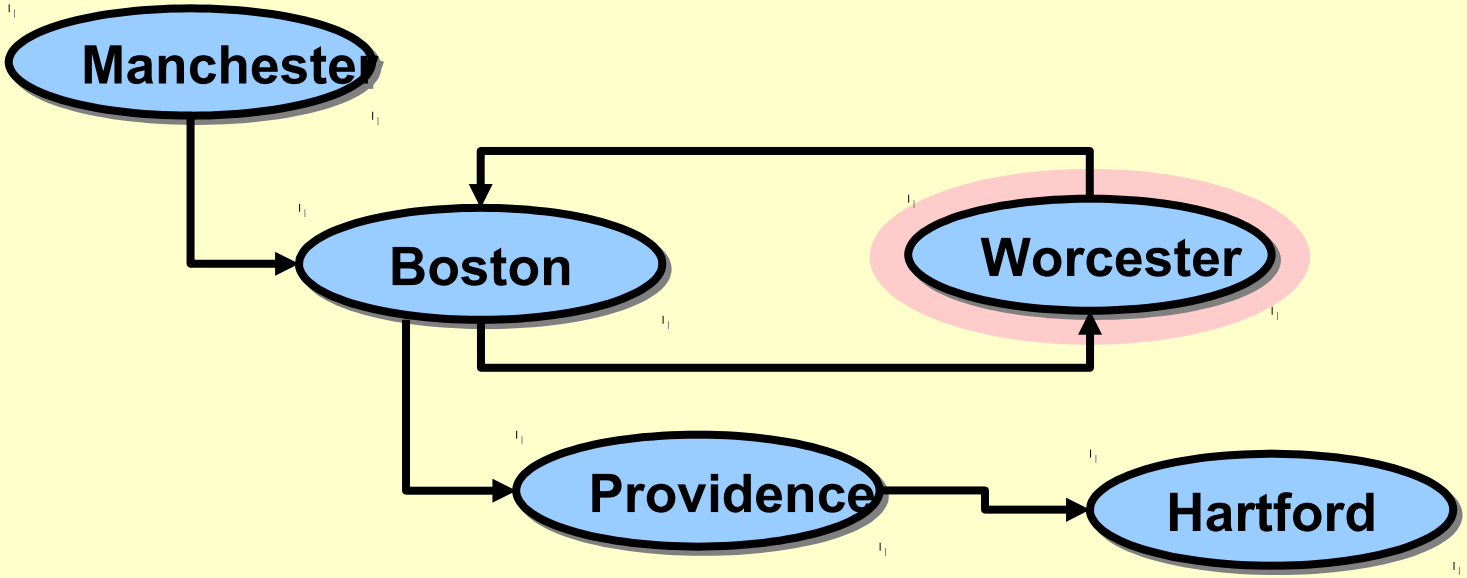
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov)
```



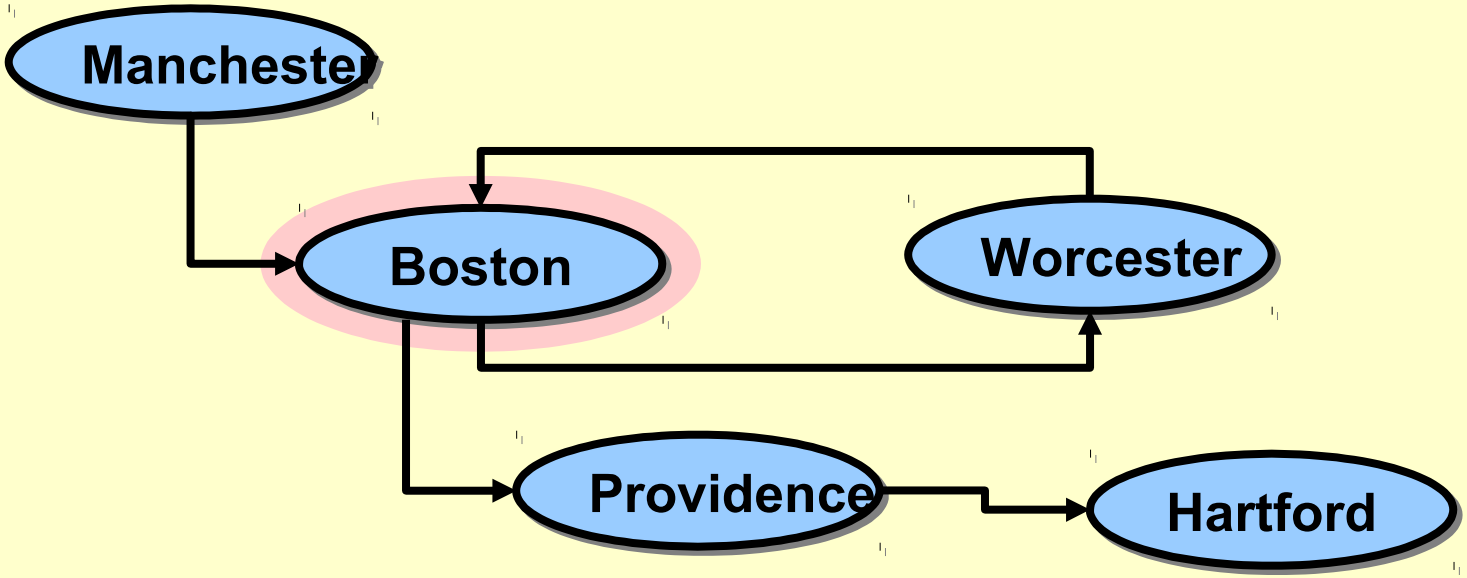
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov)
 => bost.hasRoute(prov)
```



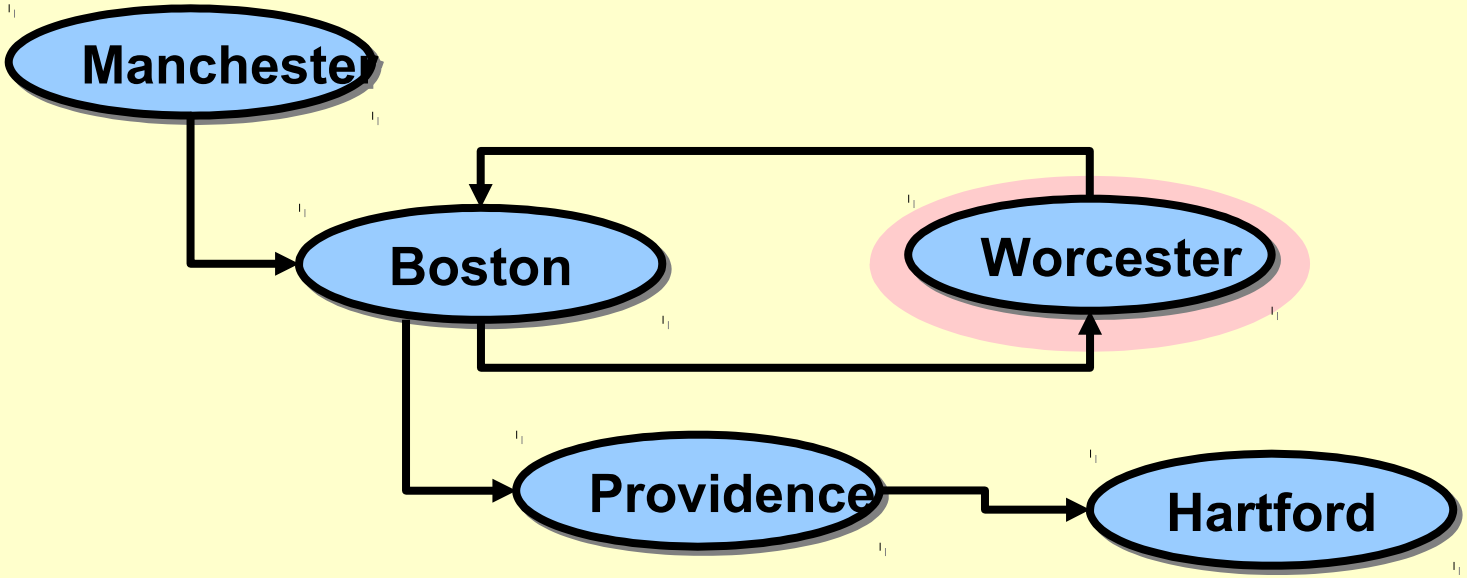
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov)
 => bost.hasRoute(prov)
 => worc.hasRoute(prov)
```



```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov)
 => bost.hasRoute(prov)
 => worc.hasRoute(prov)
 => bost.hasRoute(prov)
```

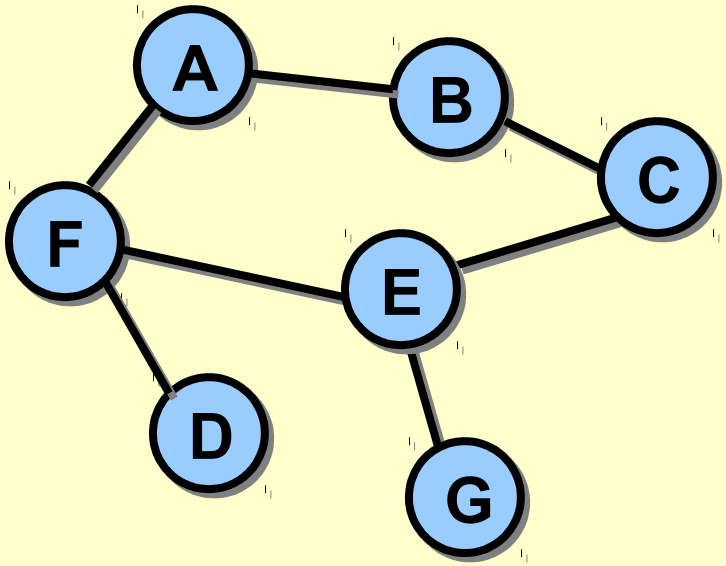


```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov)
 => bost.hasRoute(prov)
 => worc.hasRoute(prov)
 => bost.hasRoute(prov)
 => worc.hasRoute(prov)
 … etc.
```

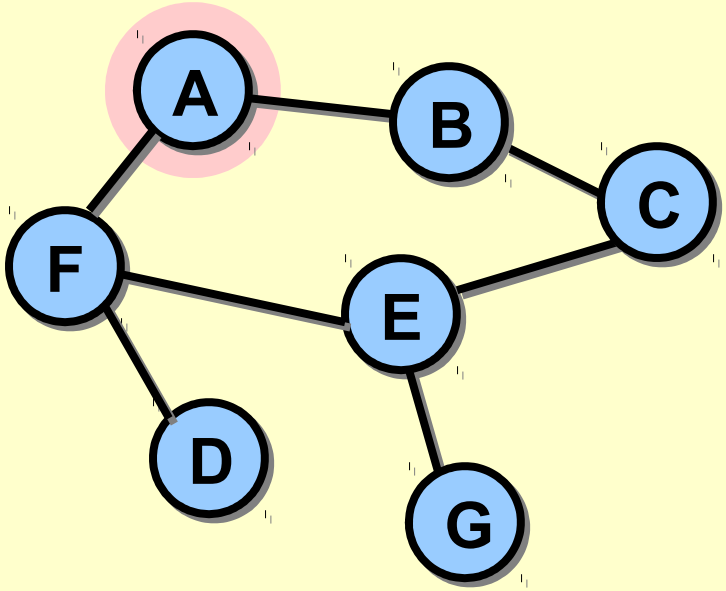


Graphs :: Searching

Let's see if there exists a path from A to G.



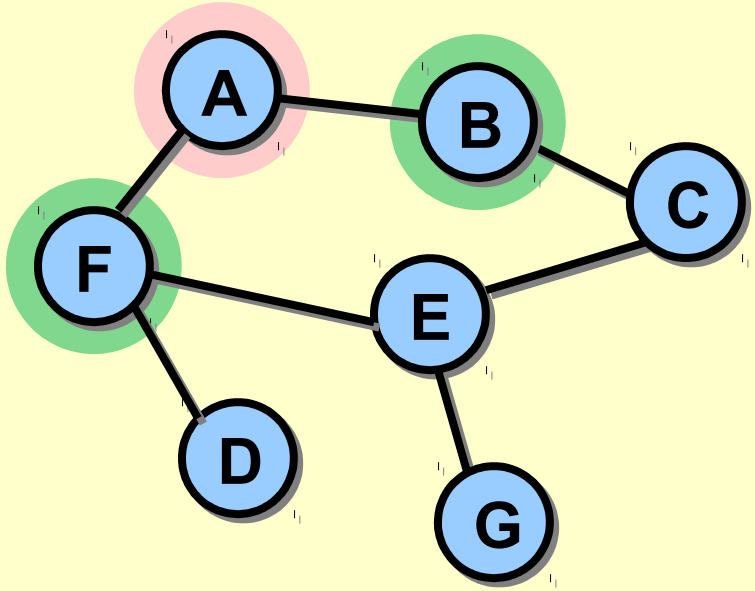
Graphs :: Searching

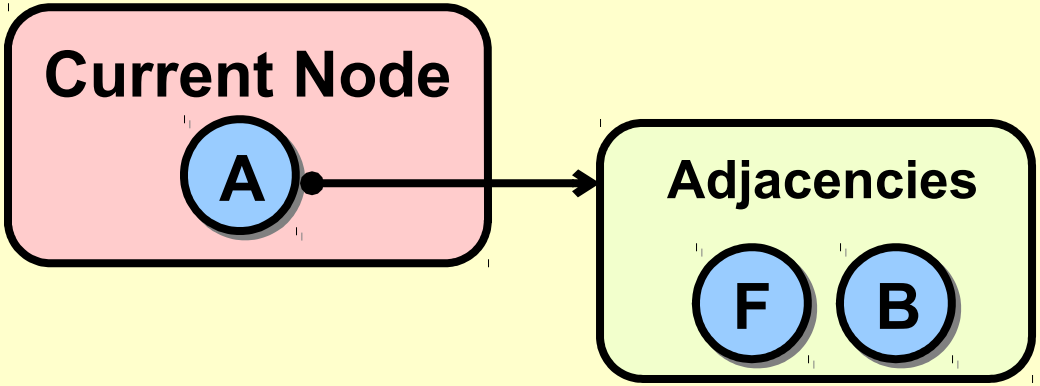


Current Node

A

Graphs :: Searching

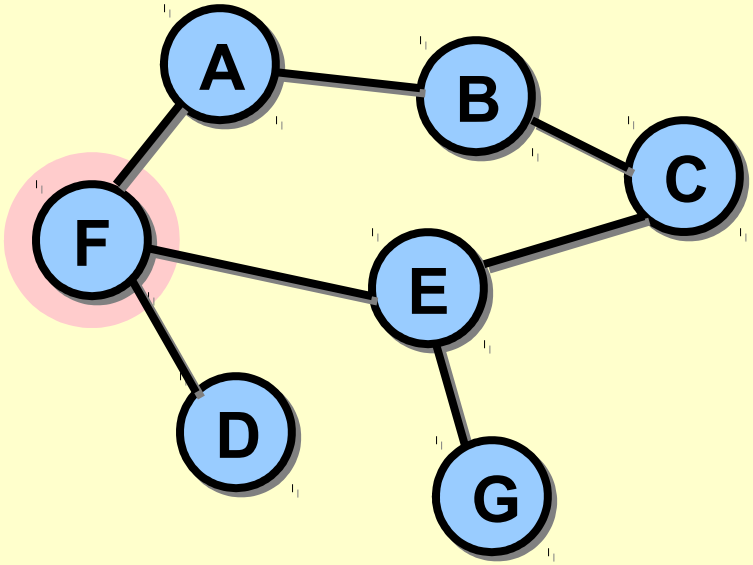




Graphs :: Searching

Visited

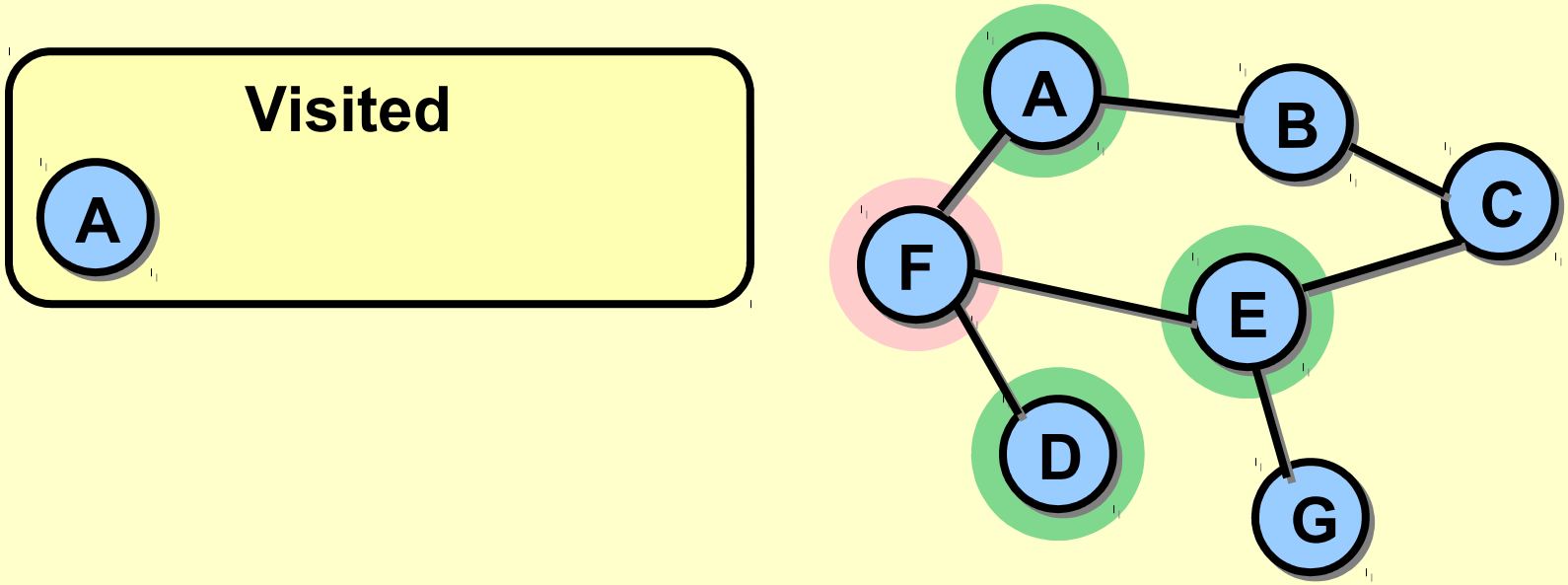
A



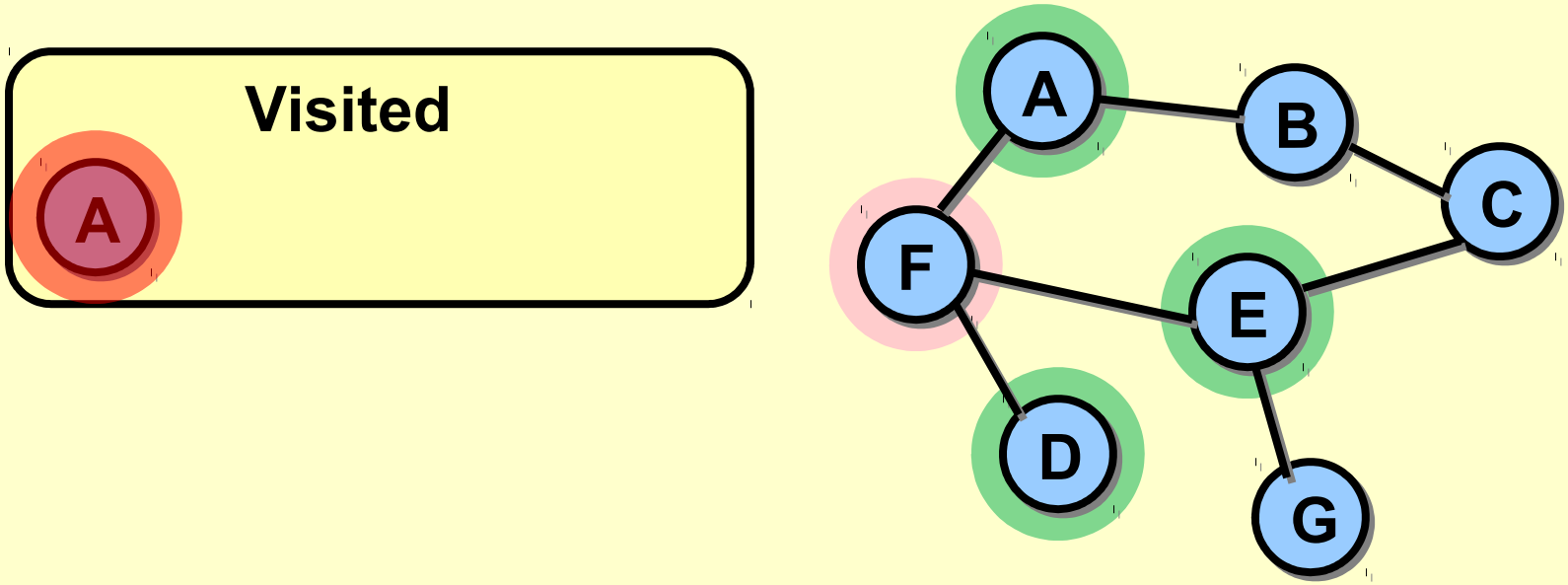
Current Node

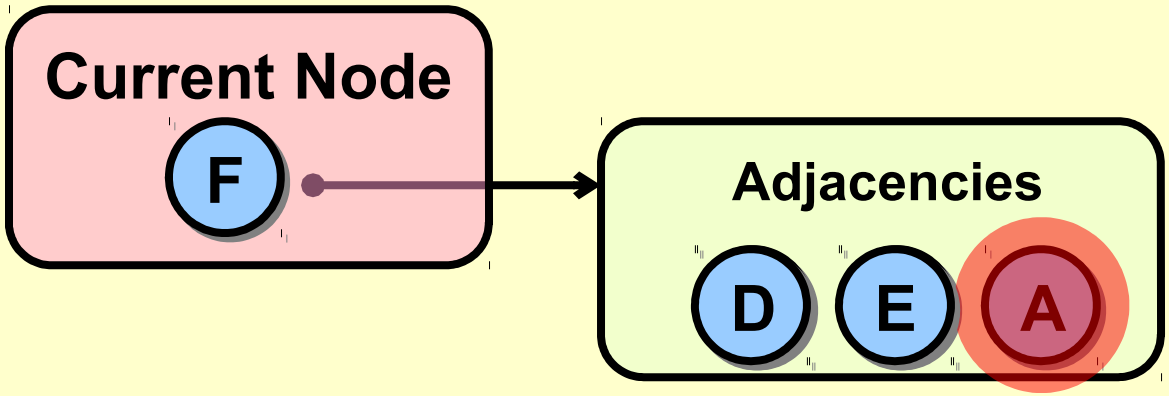
F

Graphs :: Searching



Graphs :: Searching



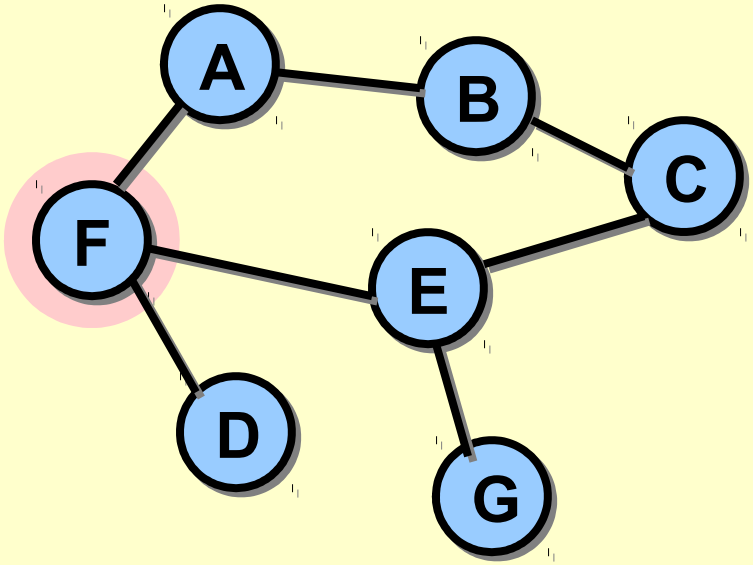


BUT WAIT! **We already visited node A.**

Graphs :: Searching

Visited

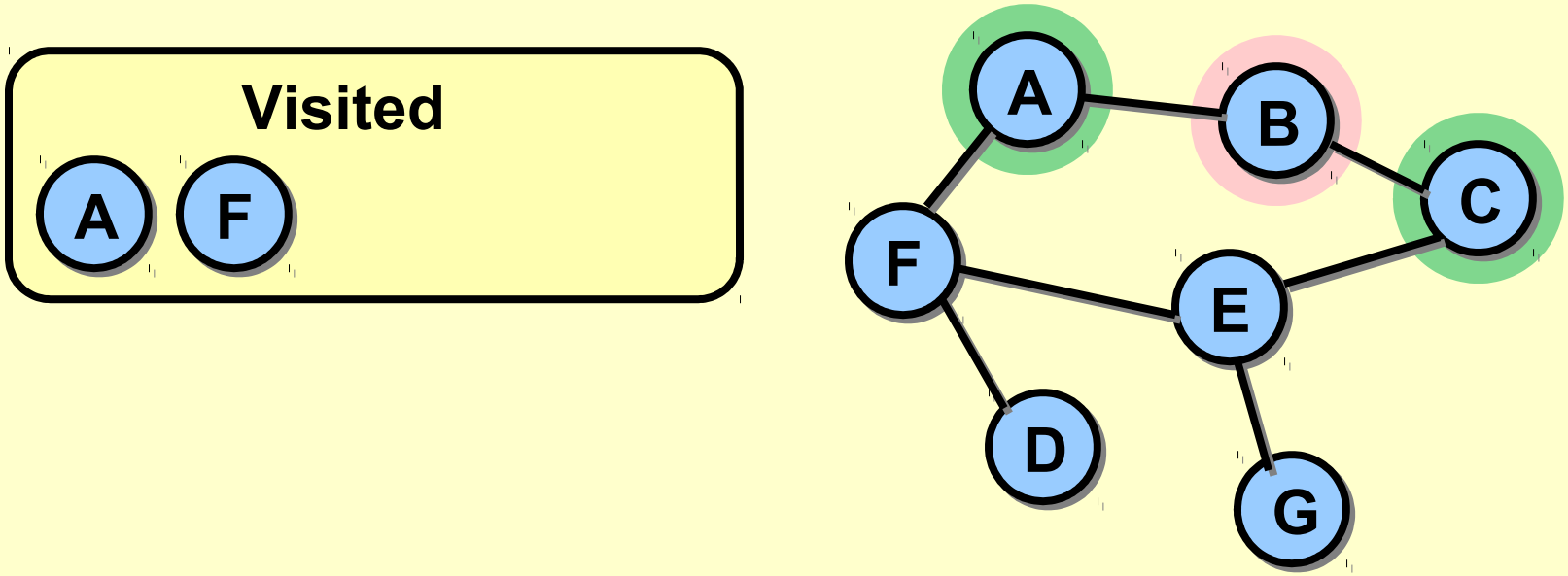
A F



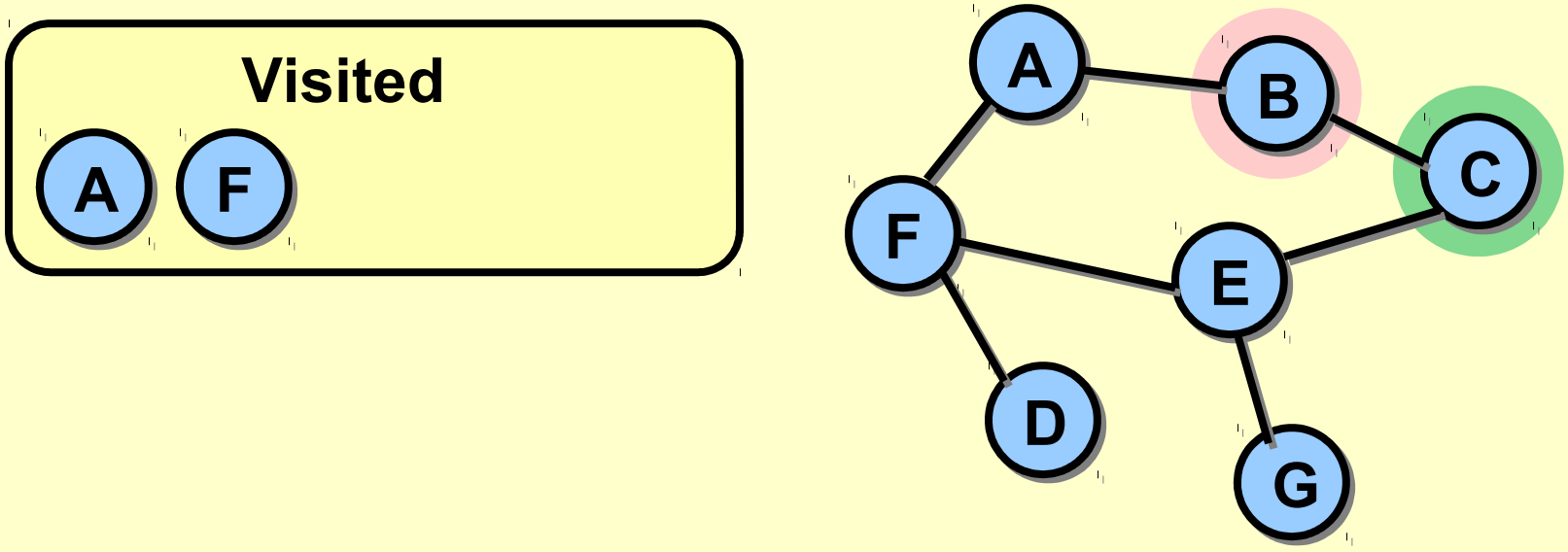
Current Node

F

Graphs :: Searching



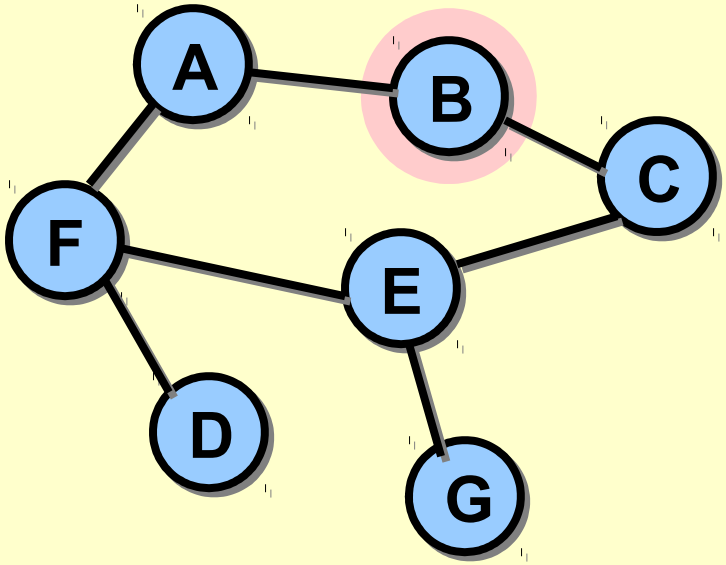
Graphs :: Searching



Graphs :: Searching

Visited

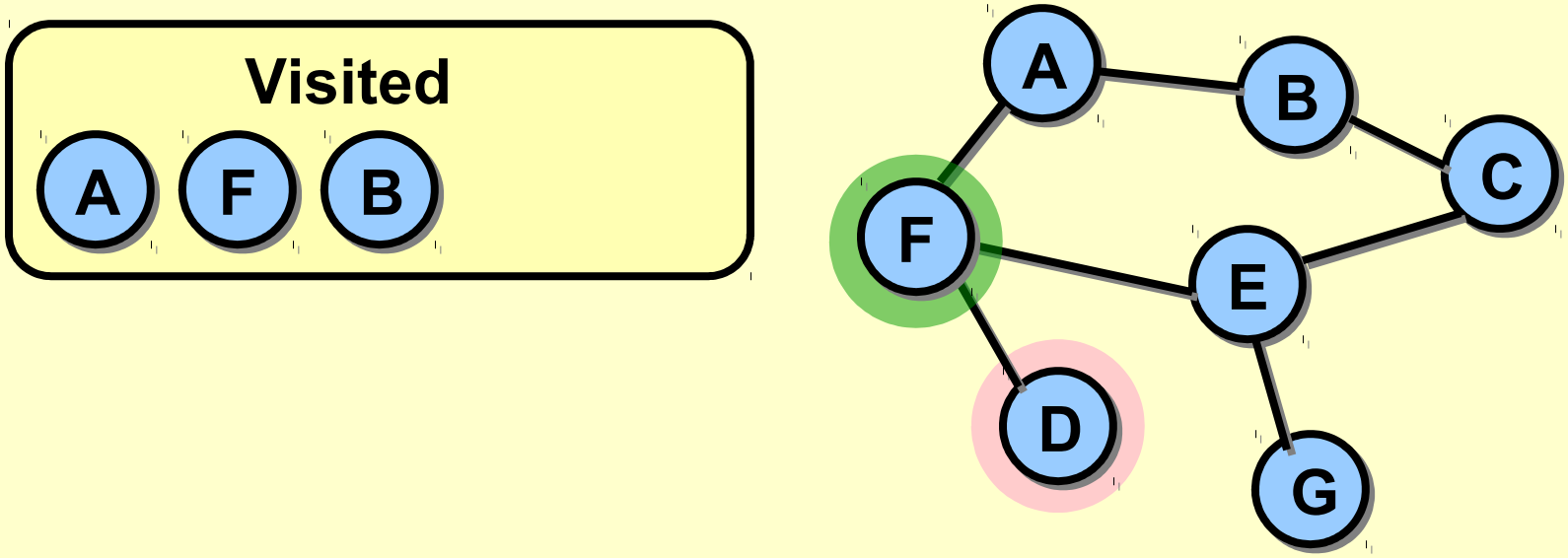
A F B



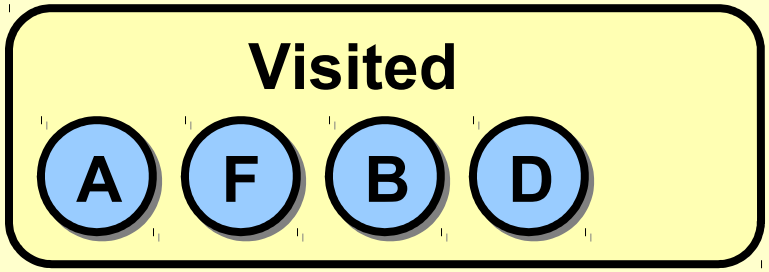
Current Node

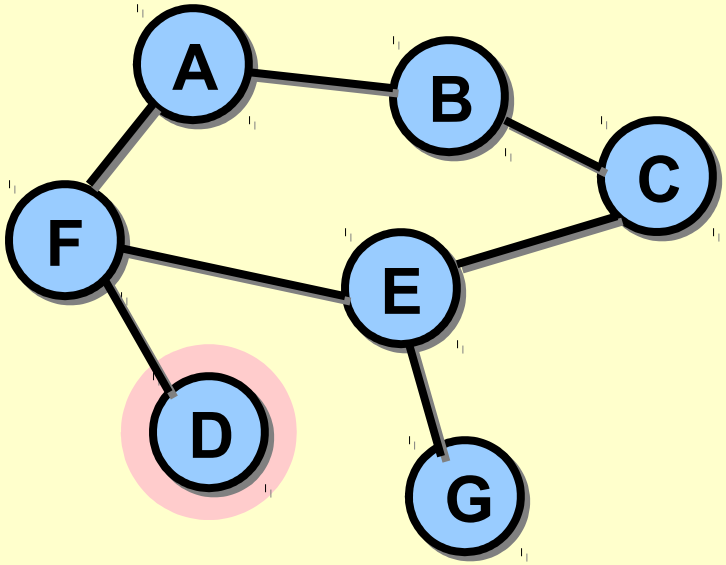
B

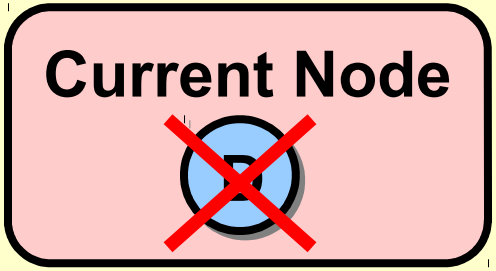
Graphs :: Searching



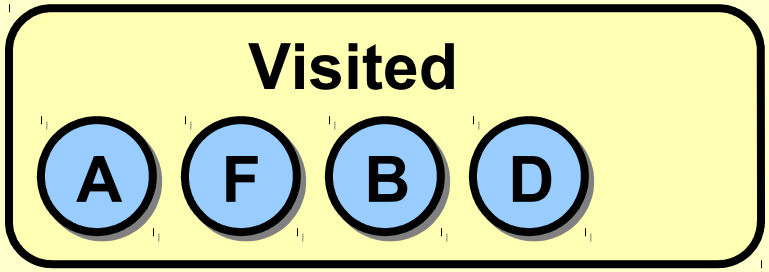
Graphs :: Searching

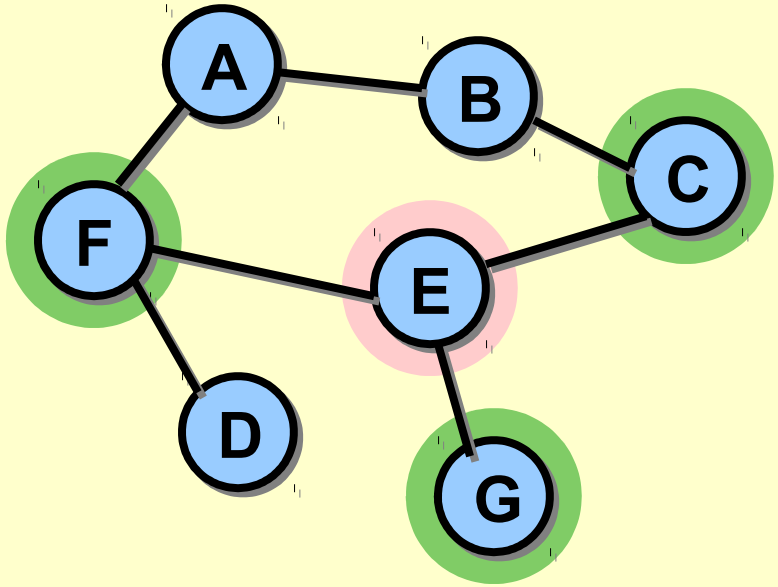


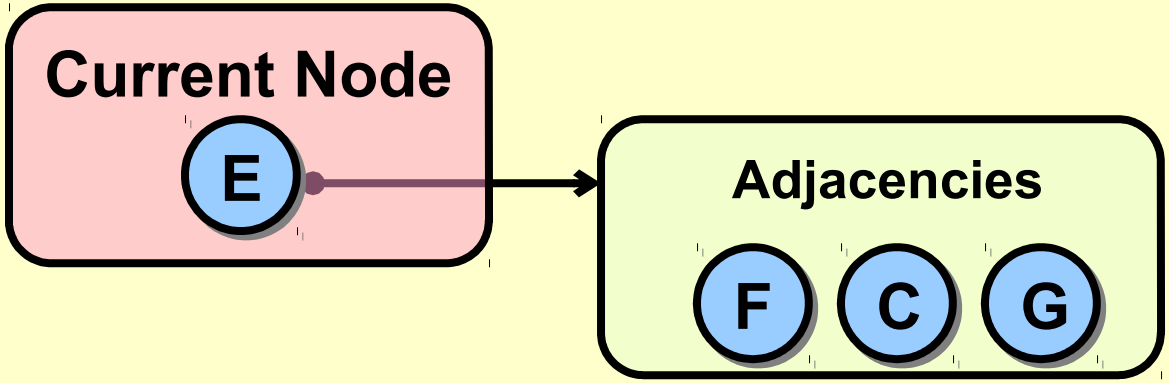




Graphs :: Searching

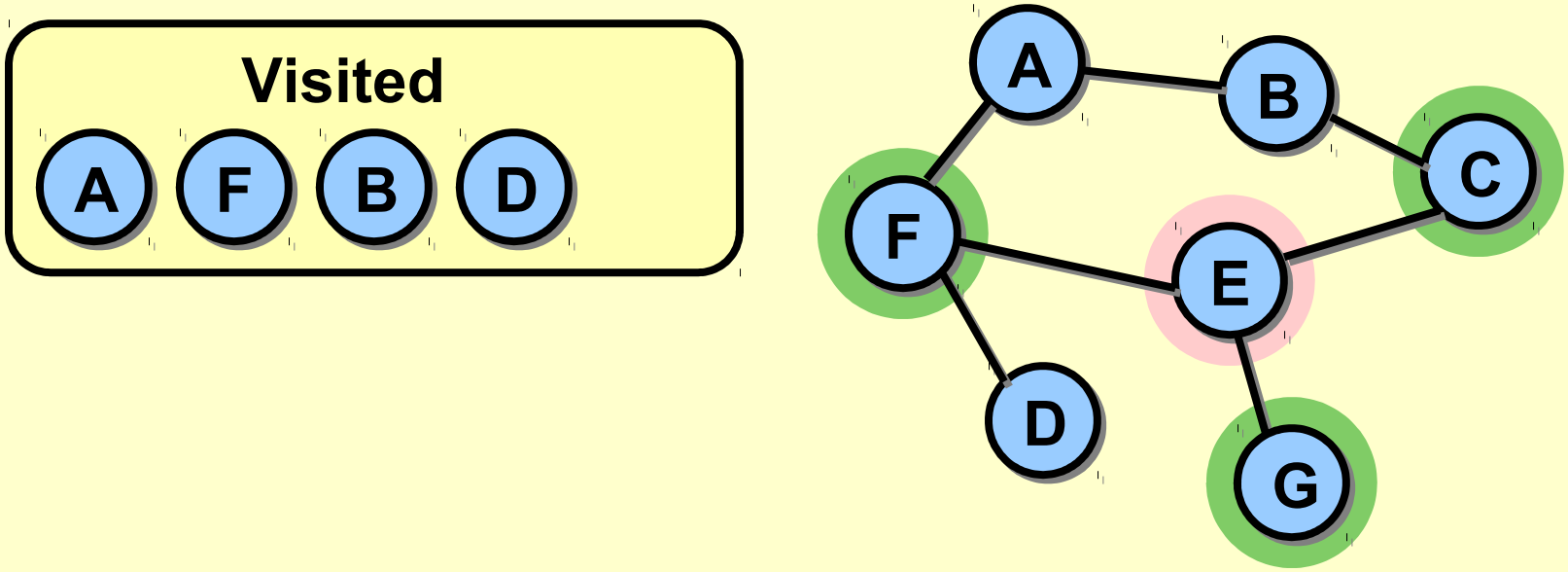




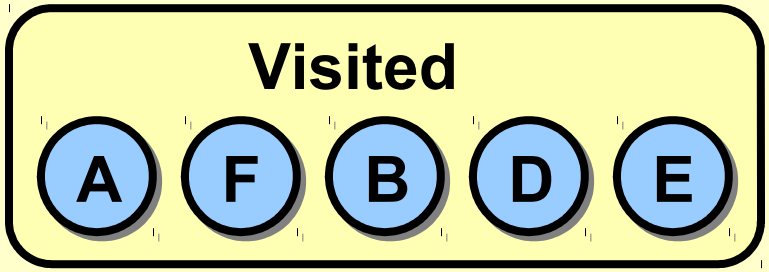


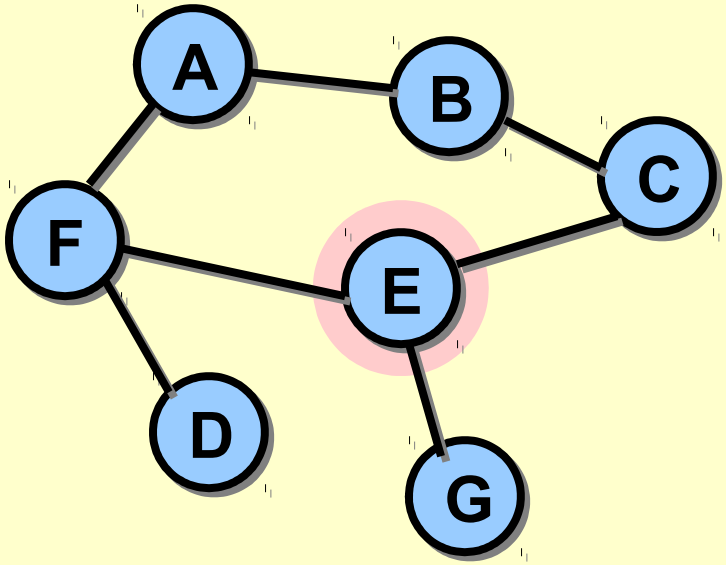
We find that E is adjacent to F, C, G.

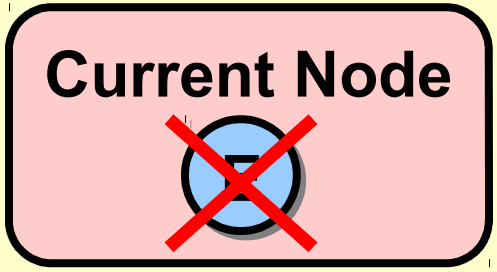
Graphs :: Searching



Graphs :: Searching



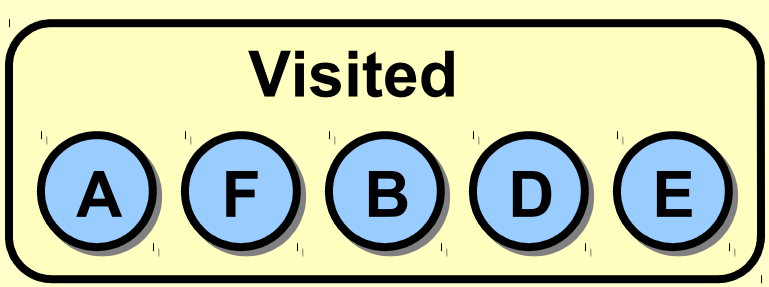


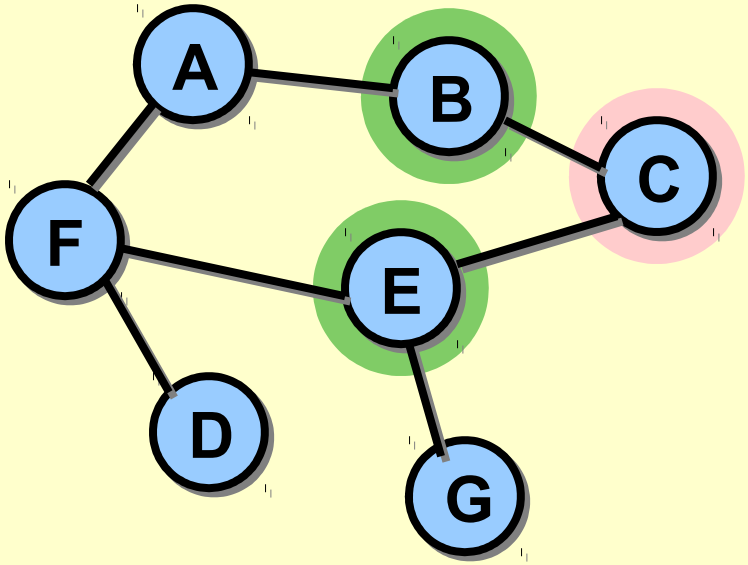


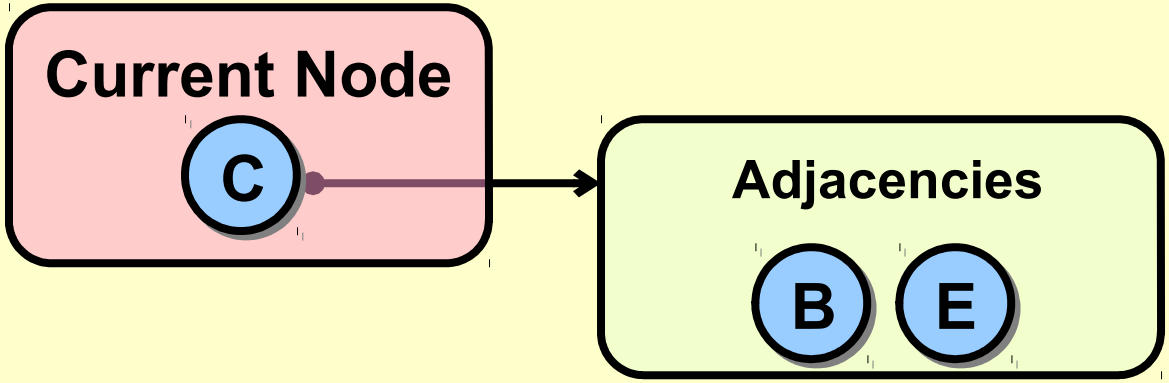
We're done with E.

Promote it to the visited list.

Graphs :: Searching

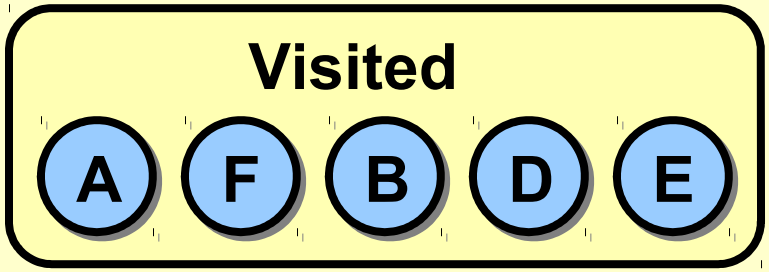


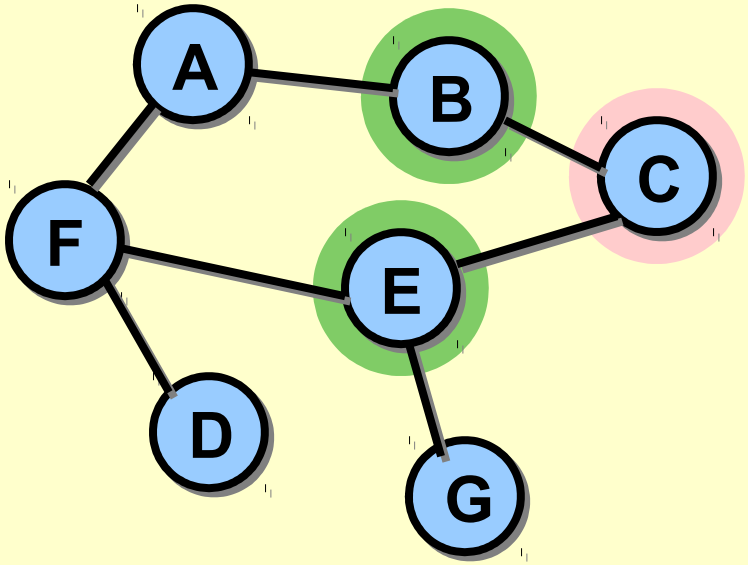


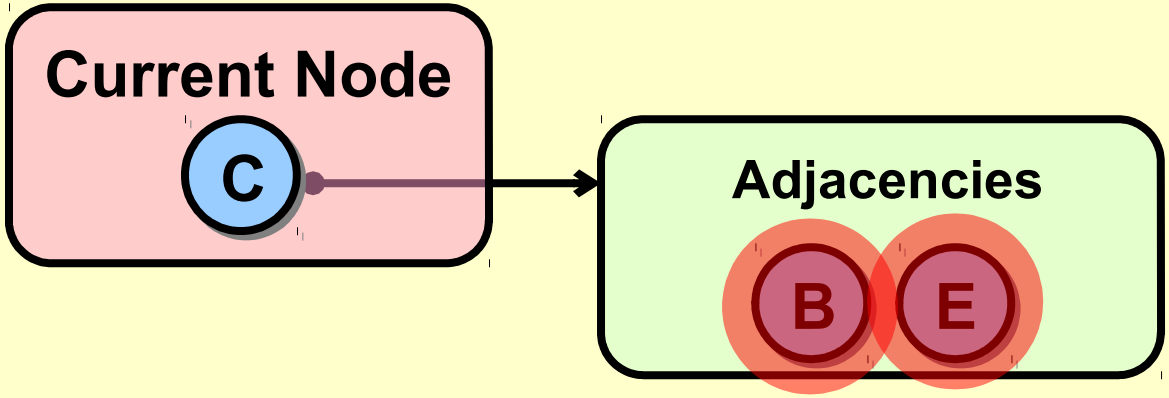


What does C contribute?

Graphs :: Searching

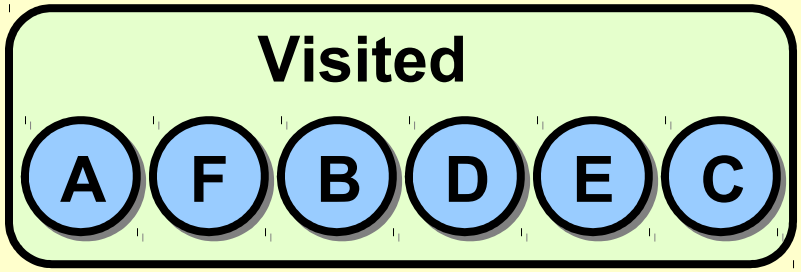


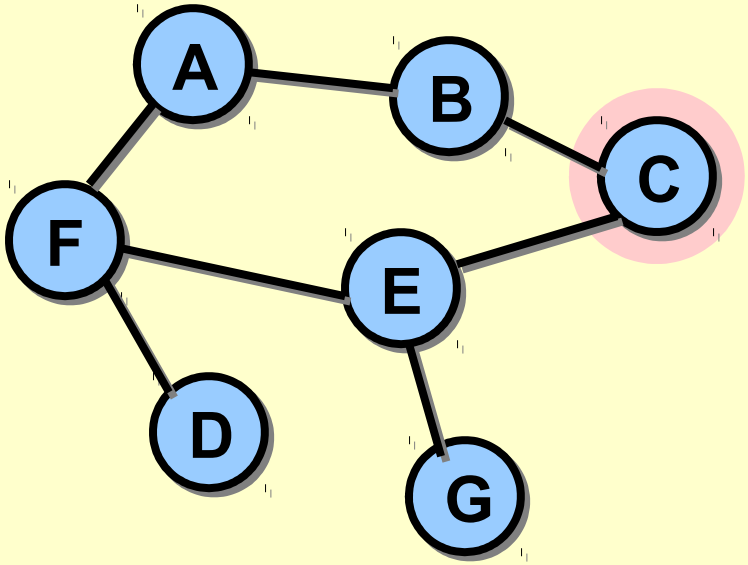


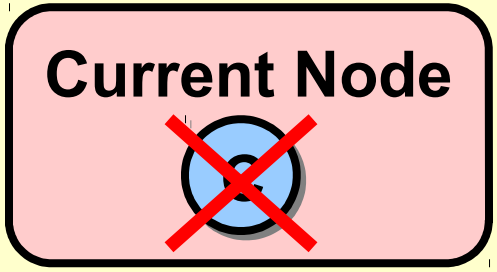


What does C contribute?

Graphs :: Searching



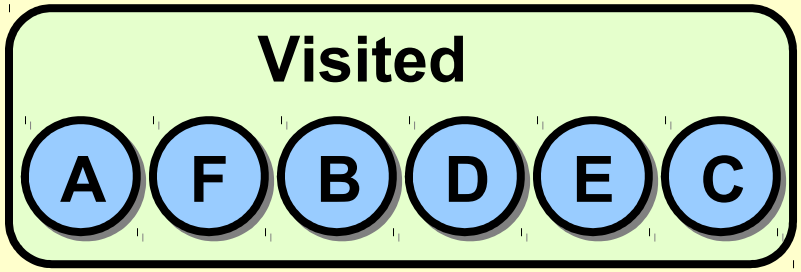


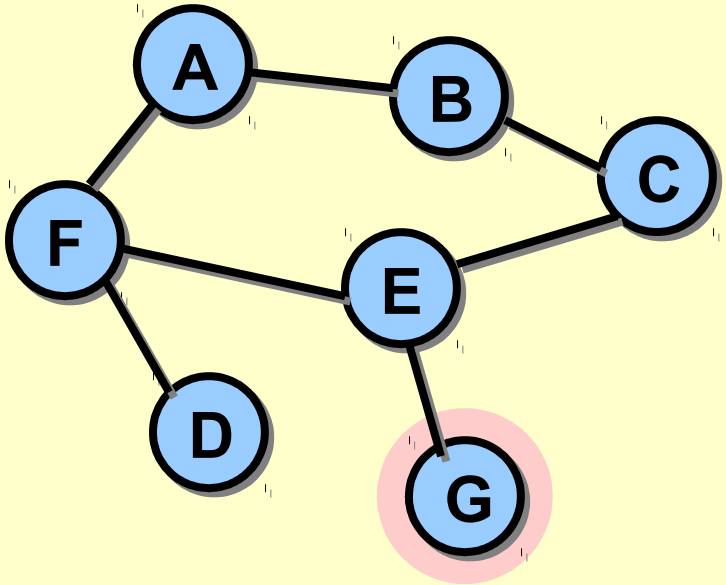


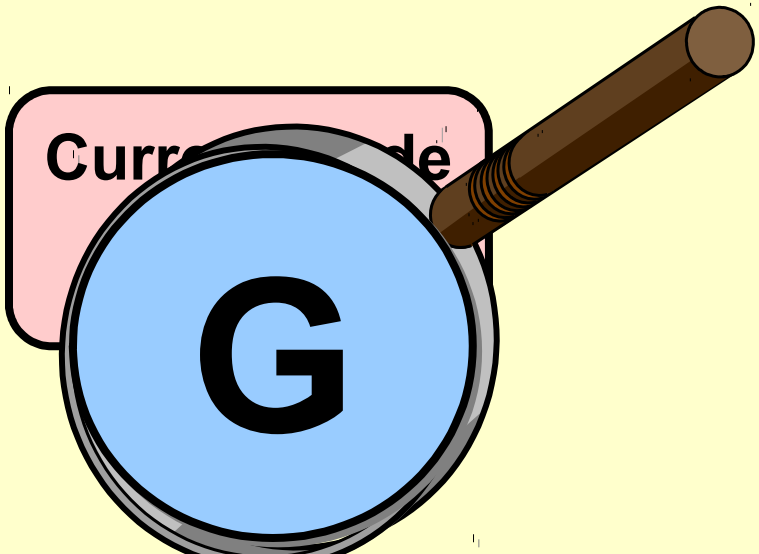
There's nothing new C can add . . .

The next node up is ...

Graphs :: Searching







G, the goal node.

We're done.

Whew.

```
public class Node {
	private String cityname;
	private LinkedList<Node> getsTo;

	boolean hasRoute(Node to) {
		if (this.equals(to))
			return true;
		else {
			for (Node c : this.getsTo) {
				if (c.hasRoute(to)) {
					return true;
				}
			}
			return false;
		}
	}
}
```

```
public class Node {
	private String cityname;
	private LinkedList<Node> getsTo;

	boolean hasRoute(Node to, LinkedList<Node> visited) {
		if (this.equals(to))
			return true;
		else {
			for (Node c : this.getsTo) {
				if (c.hasRoute(to)) {
					return true;
				}
			}
			return false;
		}
	}
}
```

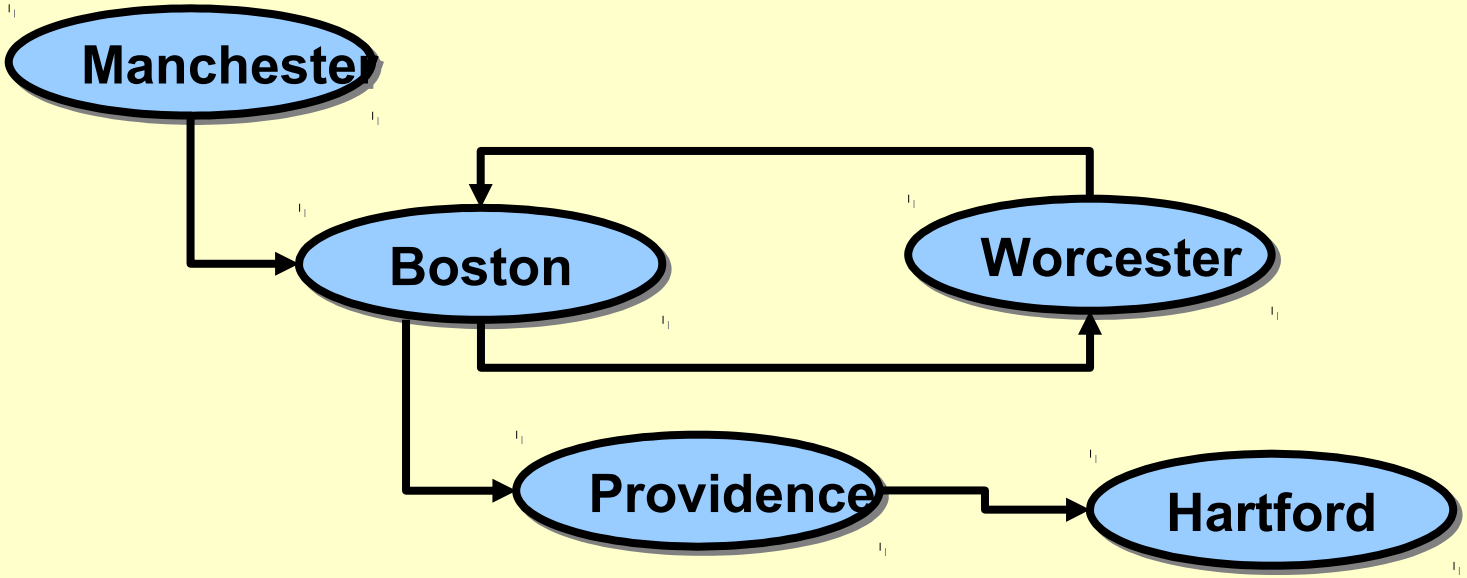
```
public class Node {
	private String cityname;
	private LinkedList<Node> getsTo;

	boolean hasRoute(Node to, LinkedList<Node> visited) {
		if (this.equals(to))
			return true;
		else if (visited.contains(this))
			return false;
		else {
			for (Node c : this.getsTo) {
				if (c.hasRoute(to)) {
					return true;
				}
			}
			return false;
		}
	}
}
```

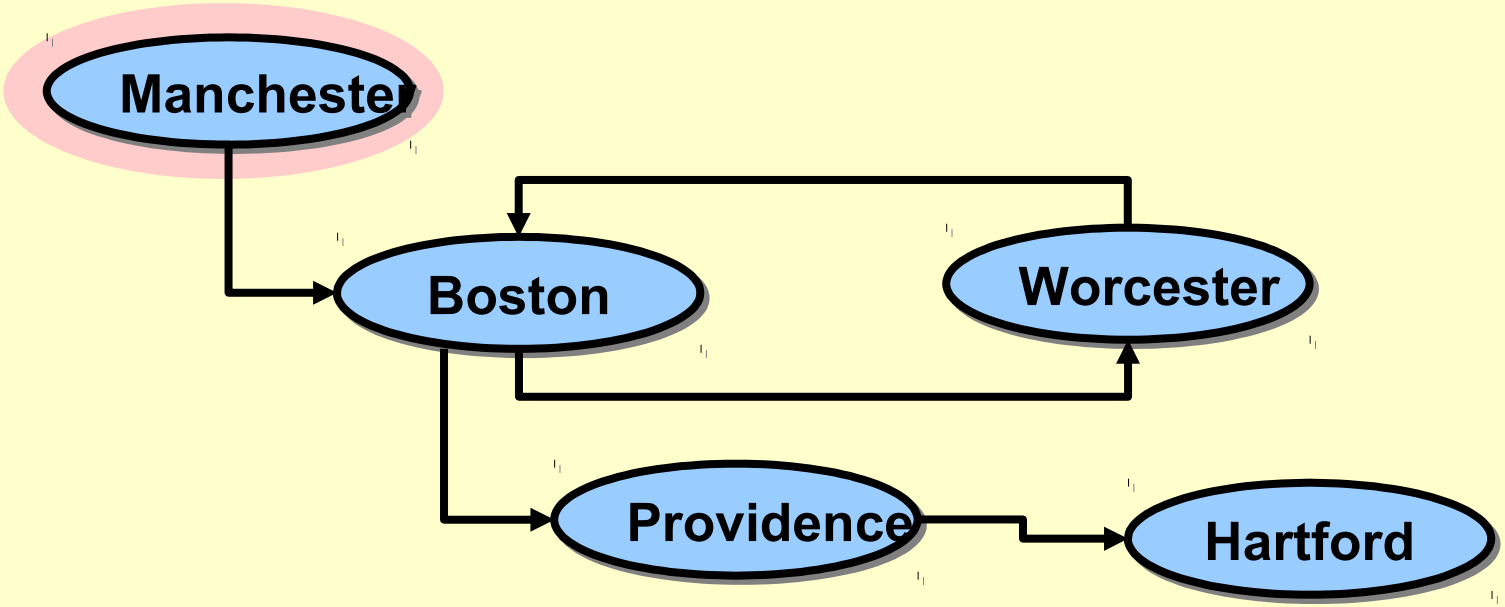
```
public class Node {
	private String cityname;
	private LinkedList<Node> getsTo;

	boolean hasRoute(Node to, LinkedList<Node> visited) {
		if (this.equals(to))
			return true;
		else if (visited.contains(this))
			return false;
		else {
			visited.add(this);
			for (Node c : this.getsTo) {
				if (c.hasRoute(to)) {
					return true;
				}
			}
			return false;
		}
	}
}
```

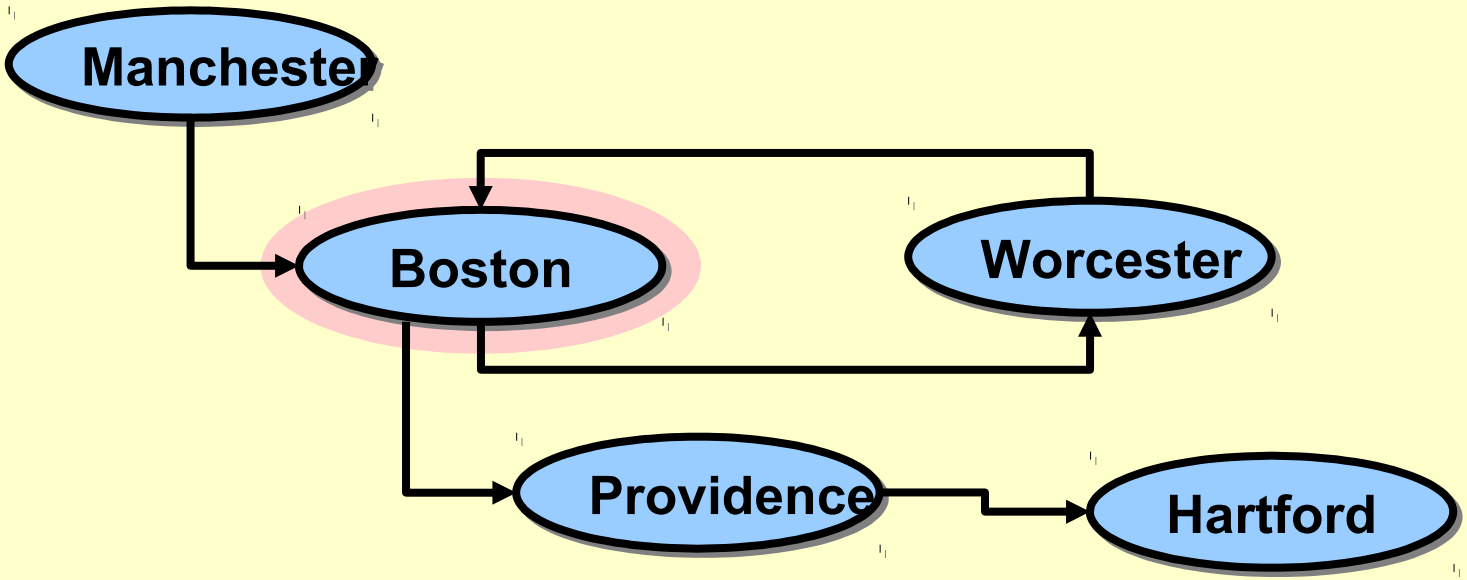
```
G.hasRoute(manc,prov)
```



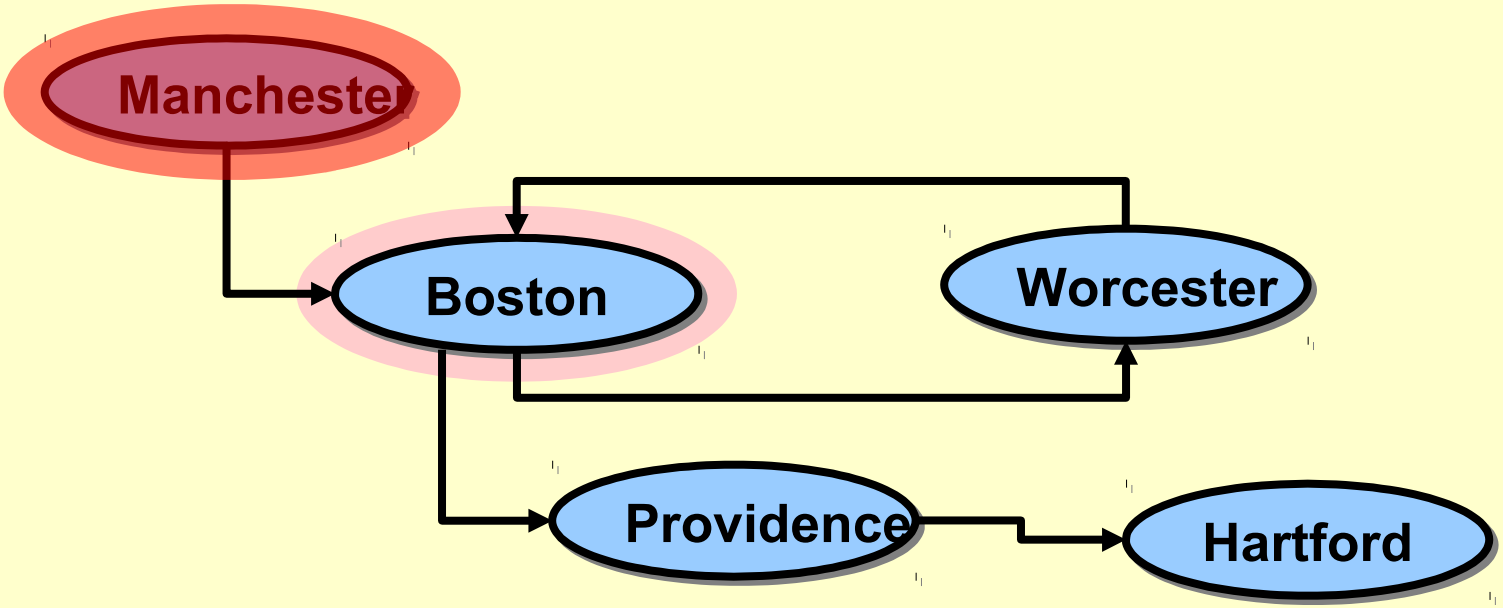
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
```



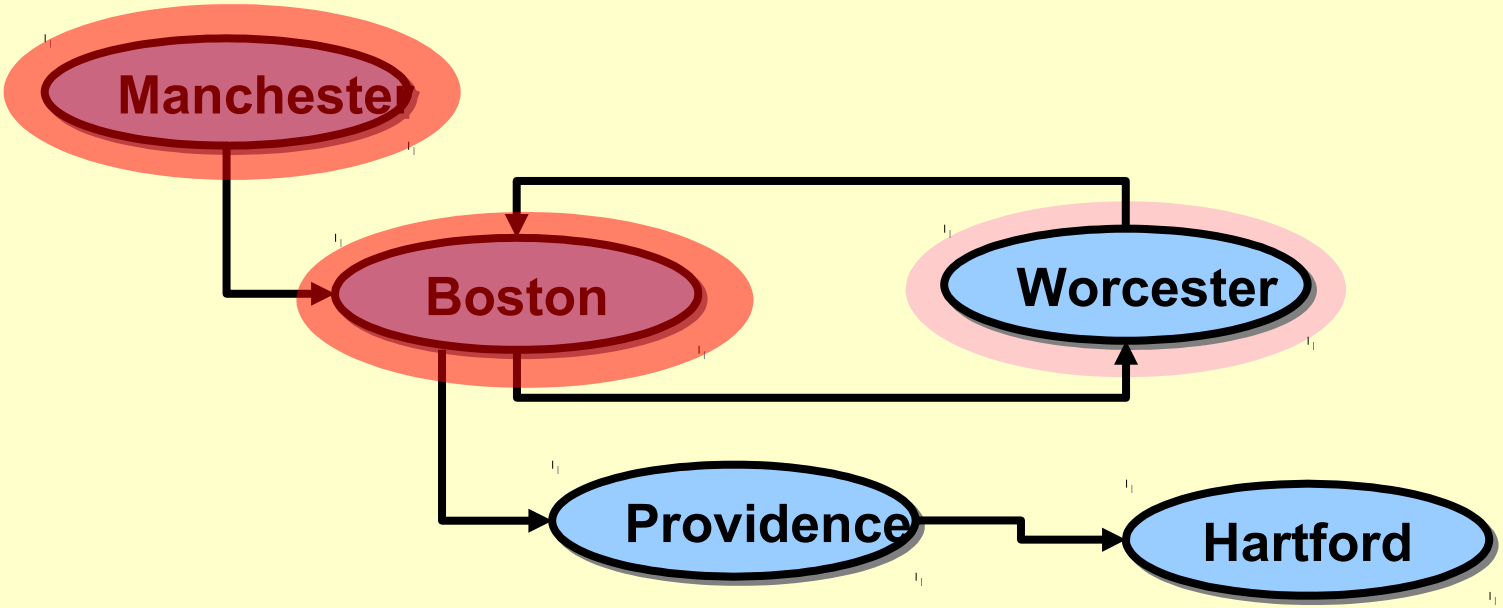
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
 => bost.hasRoute(prov, List(manc))
```



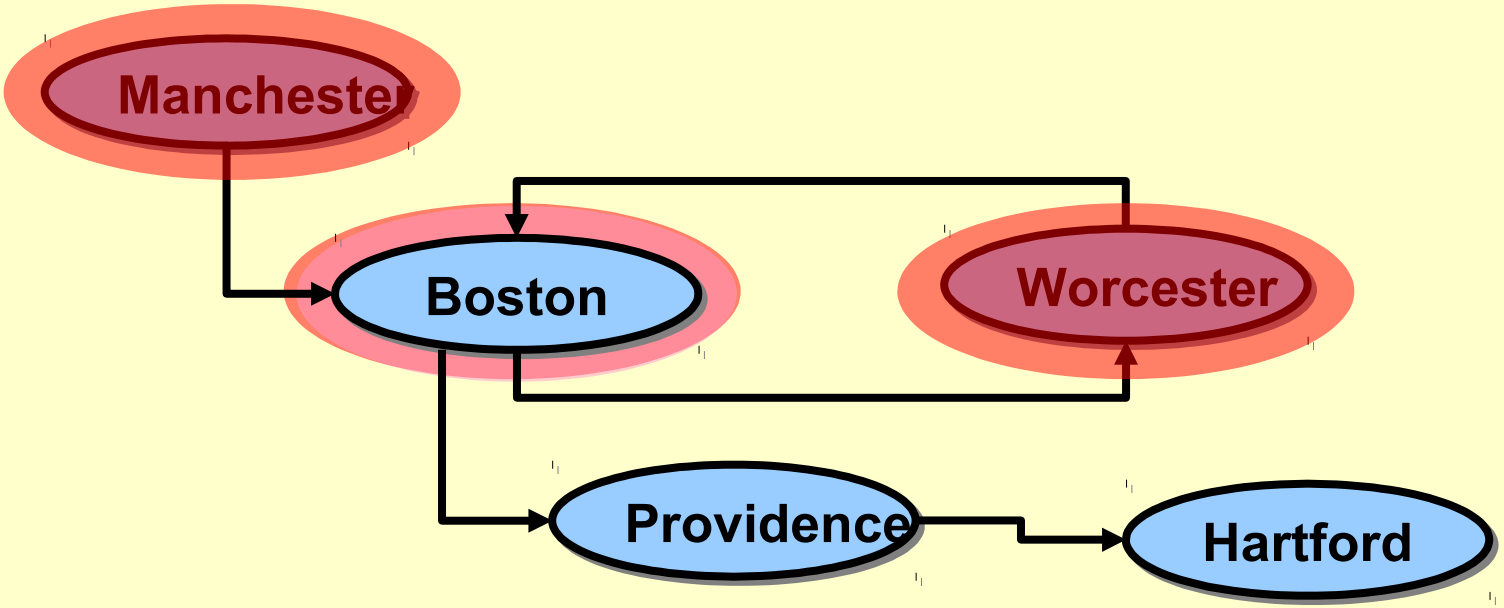
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
 => bost.hasRoute(prov, List(manc))
```



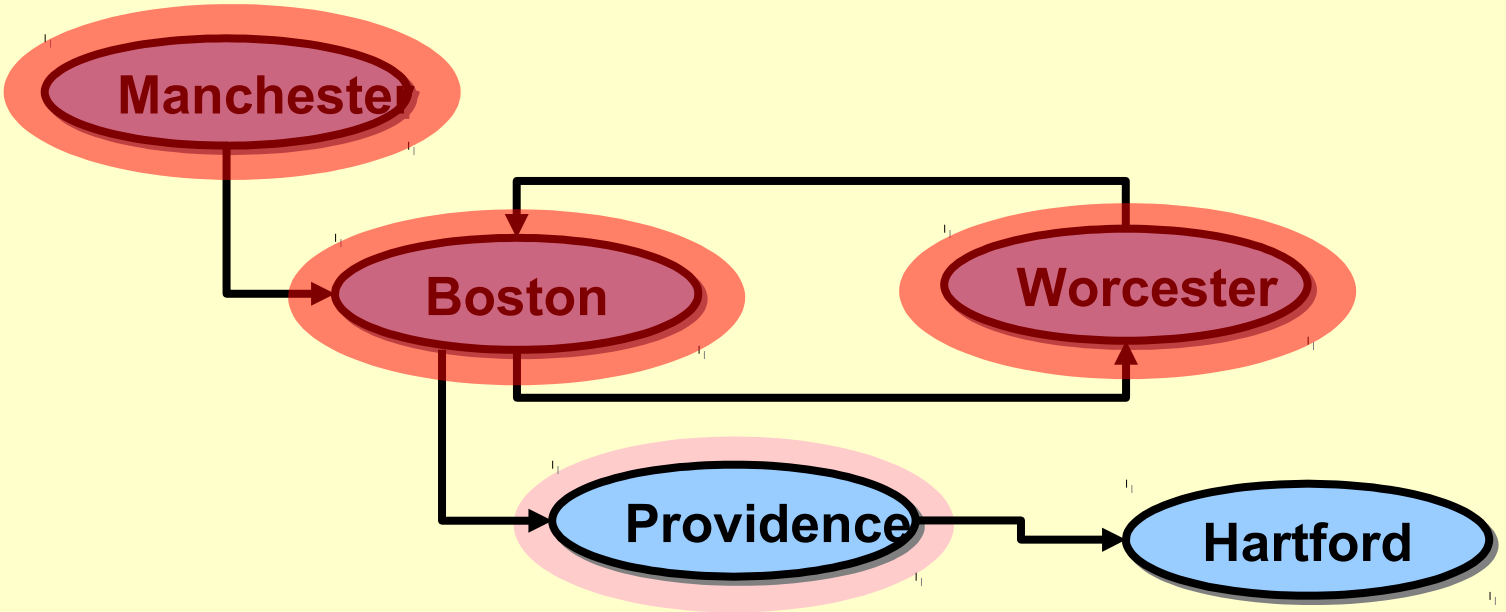
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
 => bost.hasRoute(prov, List(manc))
 => worc.hasRoute(prov, List(manc,bost))
```



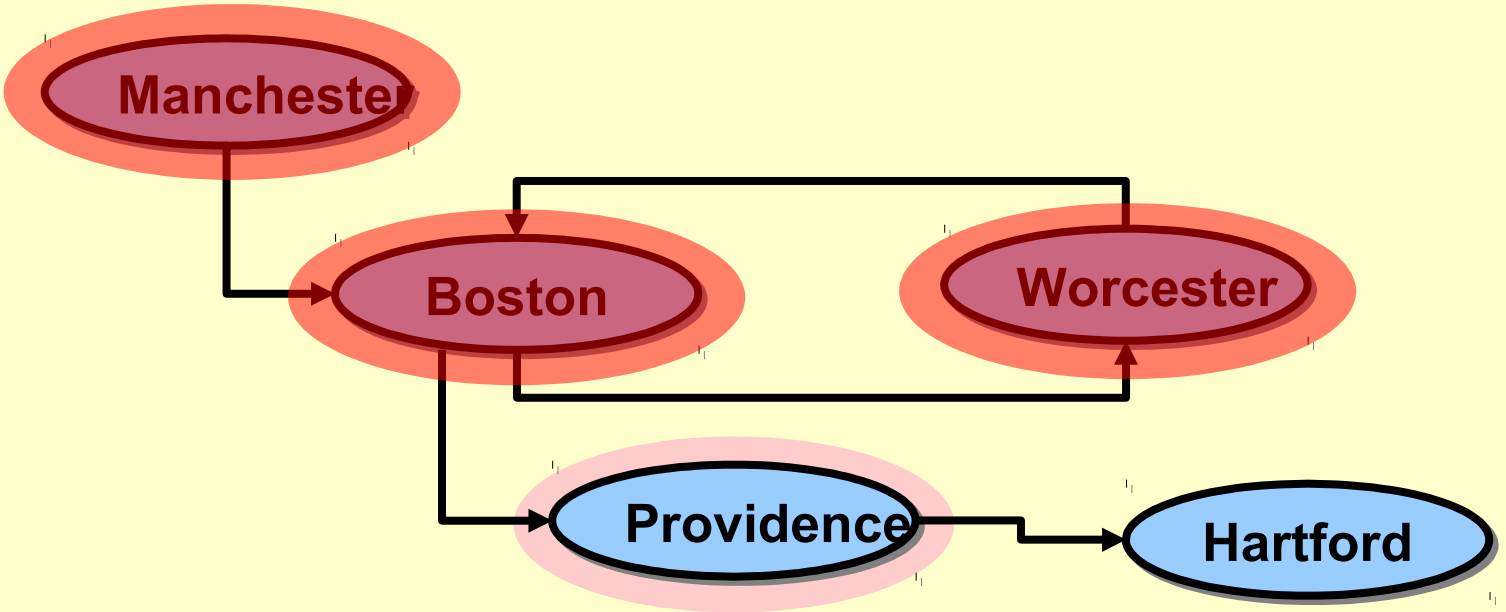
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
 => bost.hasRoute(prov, List(manc))
 => worc.hasRoute(prov, List(manc,bost))
 => bost.hasRoute(prov, List(manc,bost,worc))
```



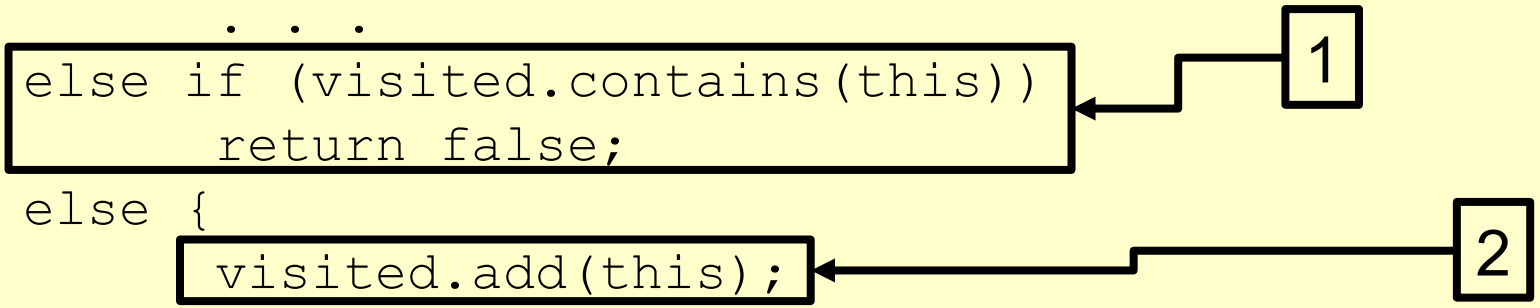
```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
 => bost.hasRoute(prov, List(manc))
 => worc.hasRoute(prov, List(manc,bost))
 => bost.hasRoute(prov, List(manc,bost,worc))
 => prov.hasRoute(prov, List(manc,bost,worc))
```



```
G.hasRoute(manc,prov)
 => manc.hasRoute(prov, List())
 => bost.hasRoute(prov, List(manc))
 => worc.hasRoute(prov, List(manc,bost))
 => bost.hasRoute(prov, List(manc,bost,worc))
 => prov.hasRoute(prov, List(manc,bost,worc))
 => returns true
```

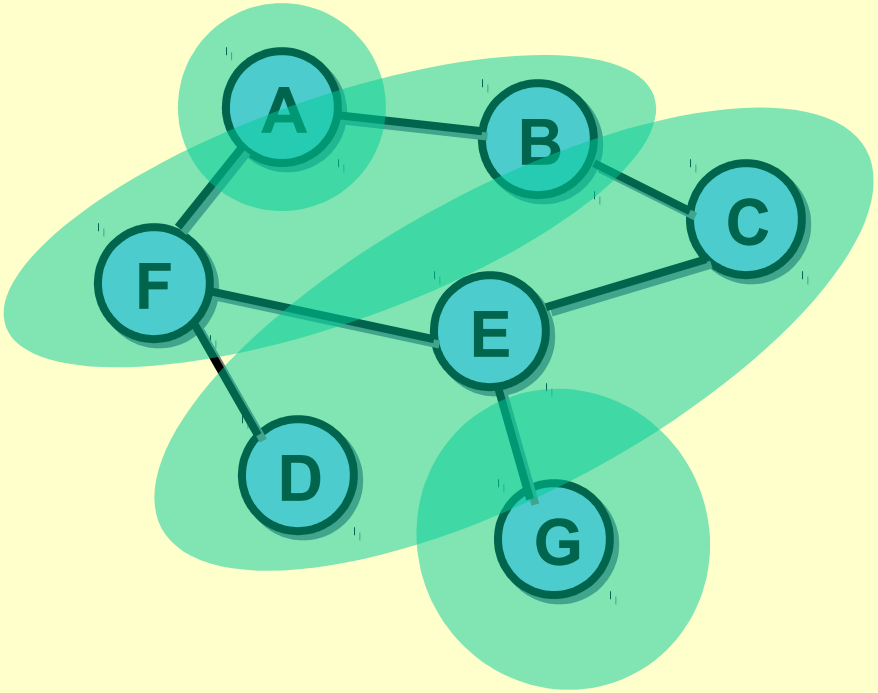


```
/**
   * Determine whether there exists a route from this Node to
   * the given node
   *
   * INVARIANT: Node n is in visited iff n.hasRoute has been called
   * since the most recent call to hasRoute on the overall graph.
   *
   * TERMINATES because
   *    1. code checks visited list before recurring,
   *    2. visited list grows every time hasRoute is called from
   *       a new node,
   *    3. the invariant guarantees that nodes added to
   *       visited remain there until computation completes, and
   *    4. there are a finite number of possible nodes on which
   *       to call hasRoute.
   */
boolean hasRoute(Node to, LinkedList<Node> visited) {
        . . .
    else if (visited.contains(this))
          return false;
    else {
         visited.add(this);
        . . .
    }
}
```



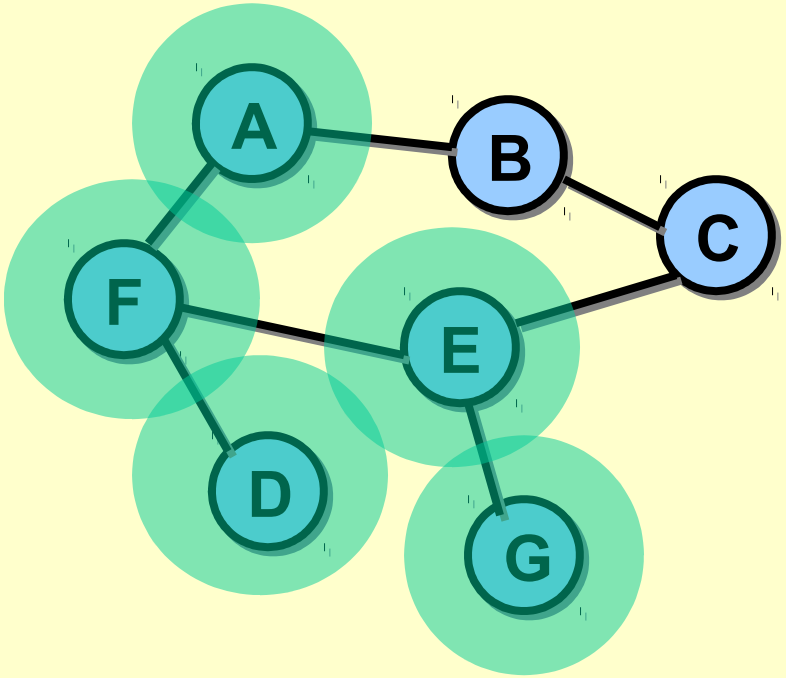
BFS: Step-by-Step

BFS will examine all nodes one step away, then two steps away, then three, etc.



DFS: Leap then Look

When the DFS discovers a new node, it races down that branch . . .



Only if it hits a dead end will it back up and examine other adjacent nodes.

Questions?