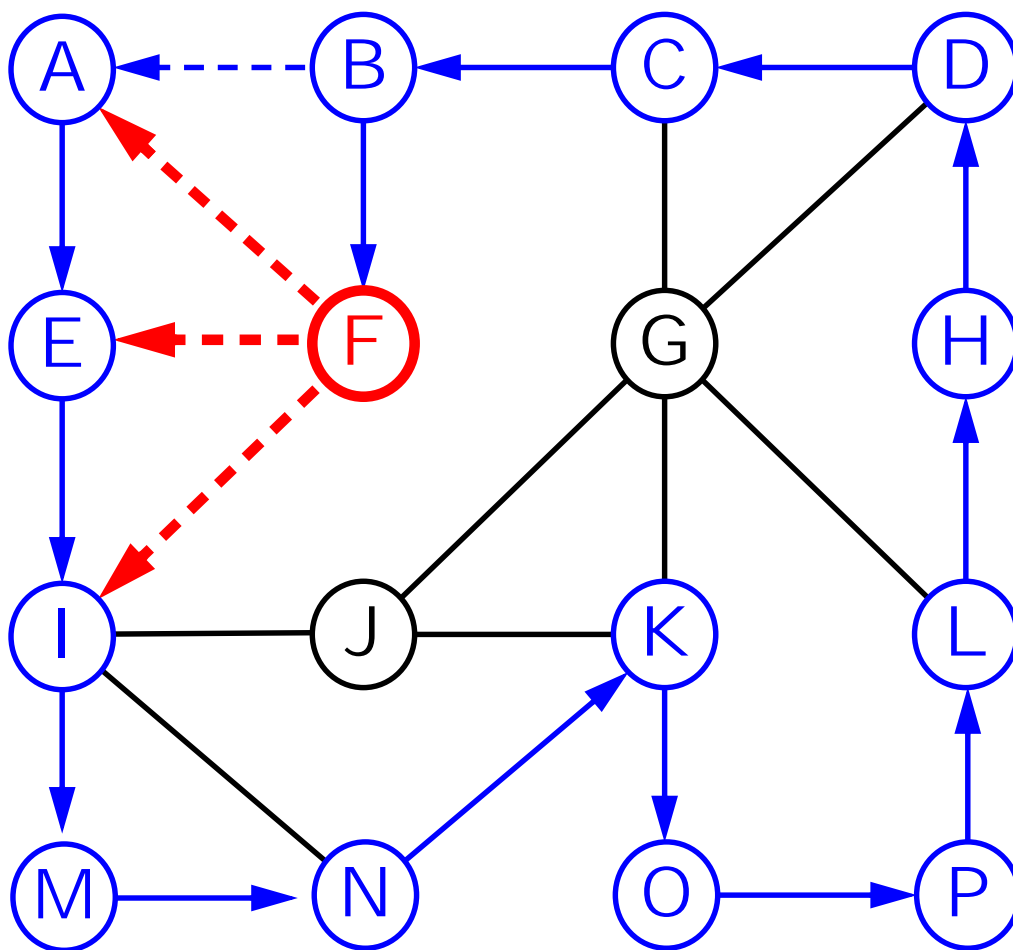


# GRAPH TRAVERSALS

- Depth-First Search
- Breadth-First Search



# Exploring a Labyrinth Without Getting Lost

- A **depth-first search (DFS)** in an undirected graph  $G$  is like wandering in a labyrinth with a string and a can of red paint without getting lost.
- We start at vertex  $s$ , tying the end of our string to the point and painting  $s$  “visited”. Next we label  $s$  as our current vertex called  $u$ .
- Now we travel along an arbitrary edge  $(u,v)$ .
- If edge  $(u,v)$  leads us to an already visited vertex  $v$  we return to  $u$ .
- If vertex  $v$  is unvisited, we unroll our string and move to  $v$ , paint  $v$  “visited”, set  $v$  as our current vertex, and repeat the previous steps.
- Eventually, we will get to a point where all incident edges on  $u$  lead to visited vertices. We then backtrack by unrolling our string to a previously visited vertex  $v$ . Then  $v$  becomes our current vertex and we repeat the previous steps.

# Exploring a Labyrinth Without Getting Lost (cont.)

- Then, if we all incident edges on  $v$  lead to visited vertices, we backtrack as we did before. We continue to backtrack along the path we have traveled, finding and exploring unexplored edges, and repeating the procedure.
- When we backtrack to vertex  $s$  and there are no more unexplored edges incident on  $s$ , we have finished our DFS search.

# Depth-First Search

**Algorithm DFS( $v$ );**

**Input:** A vertex  $v$  in a graph

**Output:** A labeling of the edges as “discovery” edges and “backedges”

**for** each edge  $e$  incident on  $v$  **do**

**if** edge  $e$  is unexplored **then**

        let  $w$  be the other endpoint of  $e$

**if** vertex  $w$  is unexplored **then**

            label  $e$  as a discovery edge

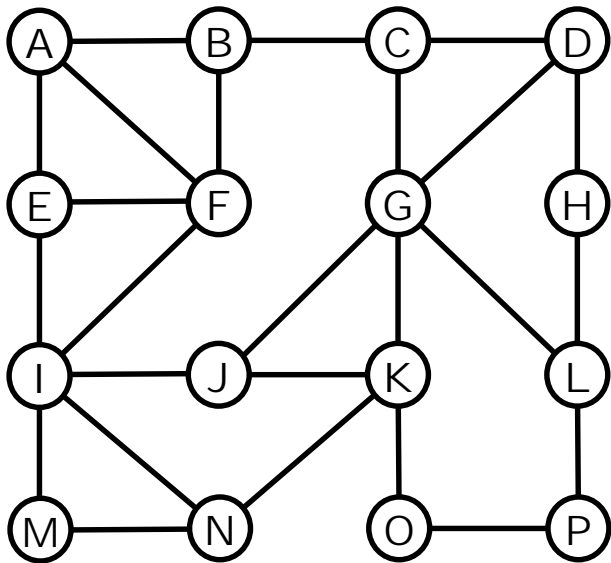
            recursively call **DFS**( $w$ )

**else**

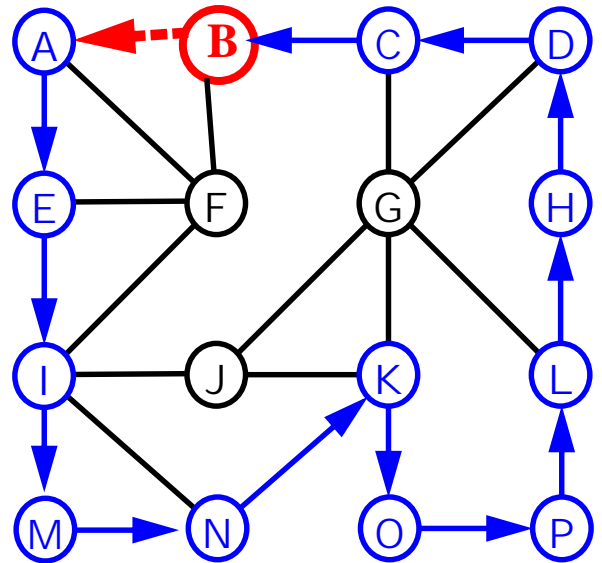
            label  $e$  as a backedge

# Depth-First Search(cont.)

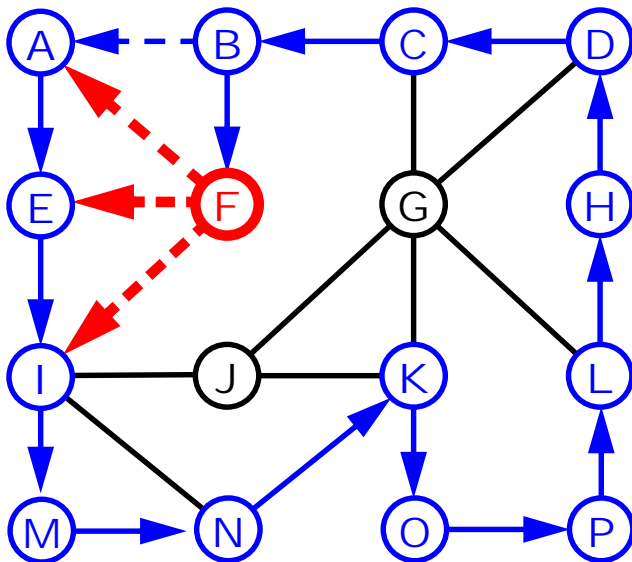
a)



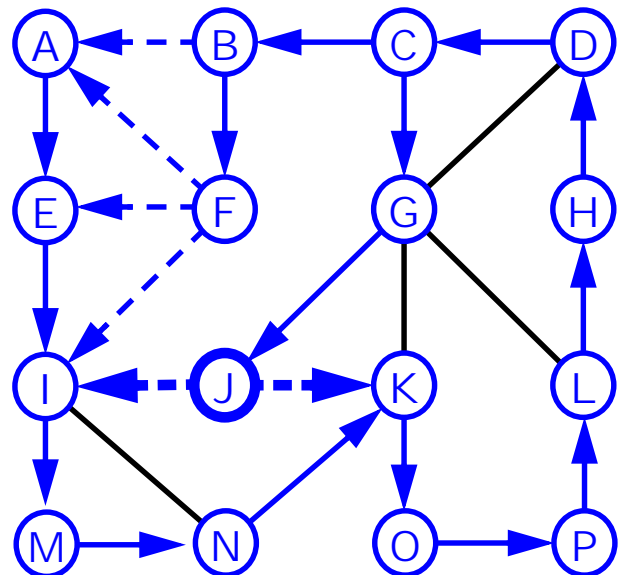
b)



c)

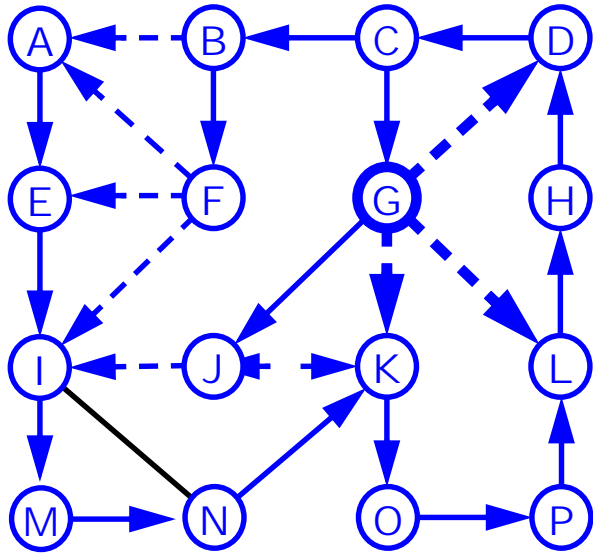


d)

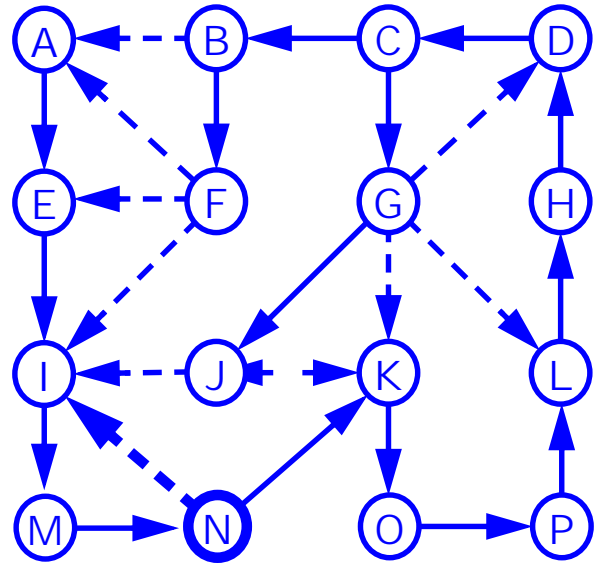


# Depth-First Search(cont.)

e)



f)



# DFS Properties

- Proposition 9.12 : Let  $G$  be an undirected graph on which a DFS traversal starting at a vertex  $s$  has been preformed. Then:
  - 1) The traversal visits all vertices in the connected component of  $s$
  - 2) The discovery edges form a spanning tree of the connected component of  $s$
- Justification of 1):
  - Let's use a contradiction argument: suppose there is at least one vertex  $v$  not visited and let  $w$  be the first unvisited vertex on some path from  $s$  to  $v$ .
  - Because  $w$  was the first unvisited vertex on the path, there is a neighbor  $u$  that has been visited.
  - But when we visited  $u$  we must have looked at  $\text{edge}(u, w)$ . Therefore  $w$  must have been visited.
  - and justification
- Justification of 2):
  - We only mark edges from when we go to unvisited vertices. So we never form a cycle of discovery edges, i.e. discovery edges form a tree.
  - This is a spanning tree because DFS visits each vertex in the connected component of  $s$

# Running Time Analysis

- Remember:
  - DFS is called on each vertex exactly once.
  - For every edge is examined exactly twice, once from each of its vertices
- For  $n_s$  vertices and  $m_s$  edges in the connected component of the vertex  $s$ , a DFS starting at  $s$  runs in  $O(n_s + m_s)$  time if:
  - The graph is represented in a data structure, like the adjacency list, where vertex and edge methods take constant time
  - Marking the vertex as explored and testing to see if a vertex has been explored takes  $O(1)$
  - We have a way of systematically considering the edges incident on the current vertex so we do not examine the same edge twice.



# Marking Vertices

- Let's look at ways to mark vertices in a way that satisfies the above condition.
- Extend vertex positions to store a variable for marking
- Use a hash table mechanism which satisfies the above condition in the probabilistic sense, because it supports the mark and test operations in  $O(1)$  expected time

# The Template Method Pattern

- the **template method** pattern provides a *generic computation mechanism* that can be specialized by redefining certain steps
- to apply this pattern, we design a class that
  - implements the *skeleton* of an algorithm
  - invokes auxiliary methods that can be redefined by its subclasses to perform useful computations
- **Benefits**
  - makes the correctness of the specialized computations rely on that of the skeleton algorithm
  - demonstrates the power of class inheritance
  - provides code reuse
  - encourages the development of generic code
- **Examples**
  - *generic traversal of a binary tree* (which includes preorder, inorder, and postorder) and its applications
  - *generic depth-first search of an undirected graph* and its applications

# Generic Depth First Search

```
public abstract class DFS {
    protected Object dfsVisit(Vertex v) {
        protected InspectableGraph graph;
        protected Object visitResult;
        initResult();
        startVisit(v);
        mark(v);
        for (Enumeration inEdges = graph.incidentEdges(v);
             inEdges.hasMoreElements();) {
            Edge nextEdge = (Edge) inEdges.nextElement();
            if (!isMarked(nextEdge)) { // found an unexplored edge
                mark(nextEdge);
                Vertex w = graph.opposite(v, nextEdge);
                if (!isMarked(w)) { // discovery edge
                    mark(nextEdge);
                    traverseDiscovery(nextEdge, v);
                    if (!isDone())
                        visitResult = dfsVisit(w); }
                else // back edge
                    traverseBack(nextEdge, v);
            }
        }
        finishVisit(v);
        return result();
    }
}
```

# Auxiliary Methods of the Generic DFS

```
public Object execute(InspectableGraph g, Vertex start,
                     Object info) {
    graph = g;
    return null;
}

protected void setResult() {}

protected void startVisit(Vertex v) {}

protected void traverseDiscovery(Edge e, Vertex from) {}

protected void traverseBack(Edge e, Vertex from) {}

protected boolean isDone() { return false; }

protected void finishVisit(Vertex v) {}

protected Object result() { return new Object(); }
```

# Specializing the Generic DFS

- class **FindPath** specializes **DFS** to return a path from vertex **start** to vertex **target**.

```
public class FindPathDFS extends DFS {
    protected Sequence path;
    protected boolean done;
    protected Vertex target;
    public Object execute(InspectableGraph g, Vertex start,
                          Object info) {
        super.execute(g, start, info);
        path = new NodeSequence();
        done = false;
        target = (Vertex) info;
        dfsVisit(start);
        return path.elements();
    }
    protected void startVisit(Vertex v) {
        path.insertFirst(v);
        if (v == target) { done = true; }
    }
    protected void finishVisit(Vertex v) {
        if (!done) path.remove(path.first());
    }
    protected boolean isDone() { return done; }
}
```

# Other Specializations of the Generic DFS

- **FindAllVertices** specializes **DFS** to return an enumeration of the vertices in the connected component containing the **start** vertex.

```
public class FindAllVerticesDFS extends DFS {  
    protected Sequence vertices;  
    public Object execute(InspectableGraph g, Vertex start,  
                          Object info) {  
        super.execute(g, start, info);  
        vertices = new NodeSequence();  
        dfsVisit(start);  
        return vertices.elements();  
    }  
  
    public void startVisit(Vertex v) {  
        vertices.insertLast(v);  
    }  
}
```

# More Specializations of the Generic DFS

- **ConnectivityTest** uses a specialized **DFS** to test if a graph is connected.

```
public class ConnectivityTest {  
    protected static DFS tester = new FindAllVerticesDFS();  
    public static boolean isConnected(InspectableGraph g)  
    {  
        if (g.numVertices() == 0) return true; //empty is  
                                                //connected  
  
        Vertex start = (Vertex)g.vertices().nextElement();  
        Enumeration compVerts =  
            (Enumeration)tester.execute(g, start, null);  
        // count how many elements are in the enumeration  
        int count = 0;  
        while (compVerts.hasMoreElements()) {  
            compVerts.nextElement();  
            count++;  
        }  
        if (count == g.numVertices()) return true;  
        return false;  
    }  
}
```

# Another Specialization of the Generic DFS

- **FindCycle** specializes **DFS** to determine if the connected component of the **start** vertex contains a **cycle**, and if so return it.

```
public class FindCycleDFS extends DFS {  
    protected Sequence path;  
    protected boolean done;  
    protected Vertex cycleStart;  
    public Object execute(InspectableGraph g, Vertex start,  
                          Object info) {  
        super.execute(g, start, info);  
        path = new NodeSequence();  
        done = false;  
        dfsVisit(start);  
        //copy the vertices up to cycleStart from the path to  
        //the cycle sequence.  
        Sequence theCycle = new NodeSequence();  
        Enumeration pathVerts = path.elements();
```



```

while (pathVerts.hasMoreElements()) {
    Vertex v = (Vertex)pathVerts.nextElement();
    theCycle.insertFirst(v);
    if ( v == cycleStart) {
        break;
    }
}
return theCycle.elements();
}

protected void startVisit(Vertex v) {path.insertFirst(v);}
protected void finishVisit(Vertex v) {
    if (done) {path.remove(path.first());}
}

//When a back edge is found, the graph has a cycle
protected void traverseBack(Edge e, Vertex from) {
    Enumeration pathVerts = path.elements();
    cycleStart = graph.opposite(from, e);
    done = true;
}

protected boolean isDone() {return done;}
}

```

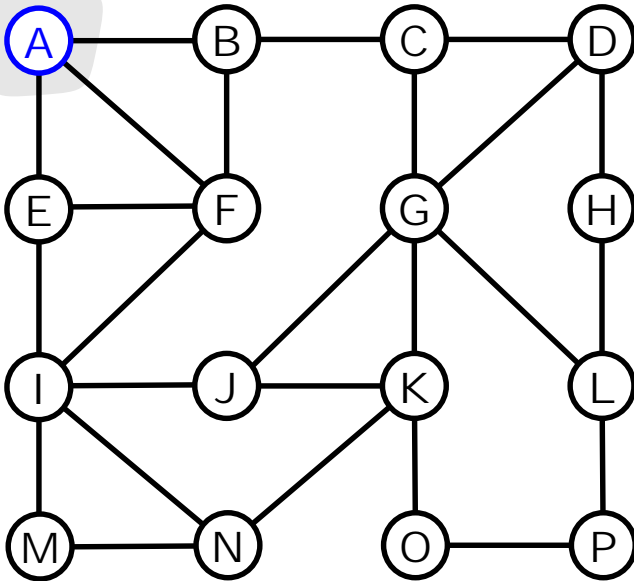
# Breadth-First Search

- Like DFS, a **Breadth-First Search** (**BFS**) traverses a connected component of a graph, and in doing so defines a spanning tree with several useful properties
  - The starting vertex  $s$  has level 0, and, as in DFS, defines that point as an “anchor.”
  - In the first round, the string is unrolled the length of one edge, and all of the edges that are only one edge away from the anchor are visited.
  - These edges are placed into level 1
  - In the second round, all the new edges that can be reached by unrolling the string 2 edges are visited and placed in level 2.
  - This continues until every vertex has been assigned a level.
  - The label of any vertex  $v$  corresponds to the length of the shortest path from  $s$  to  $v$ .

# BFS - A Graphical Representation

a)

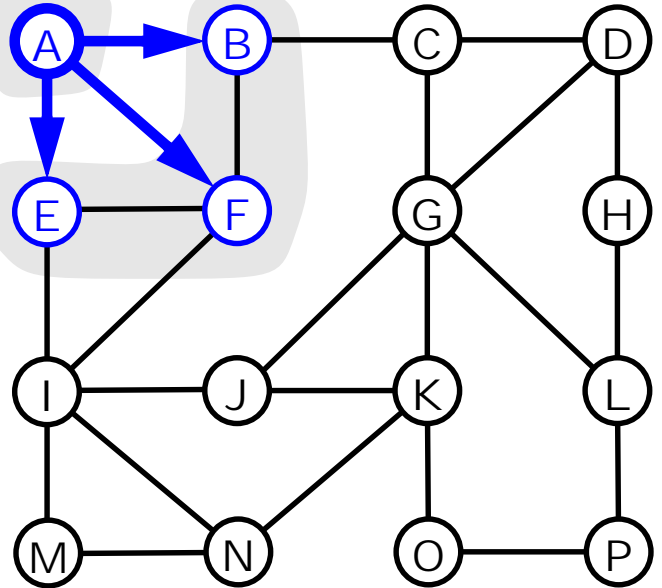
0



b)

0

1

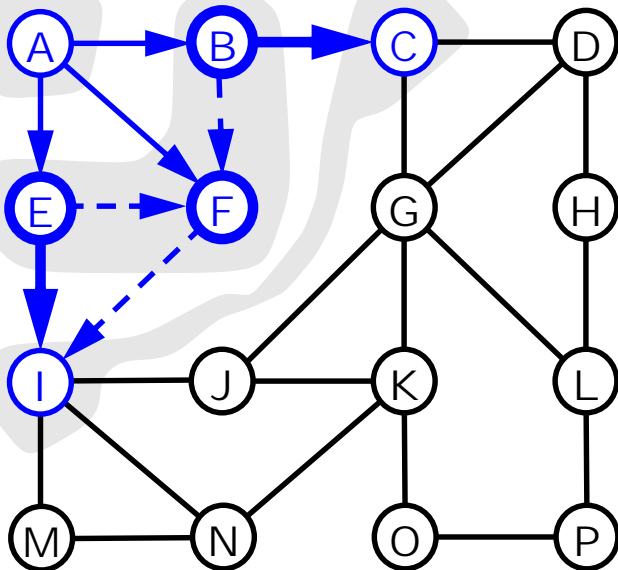


c)

0

1

2



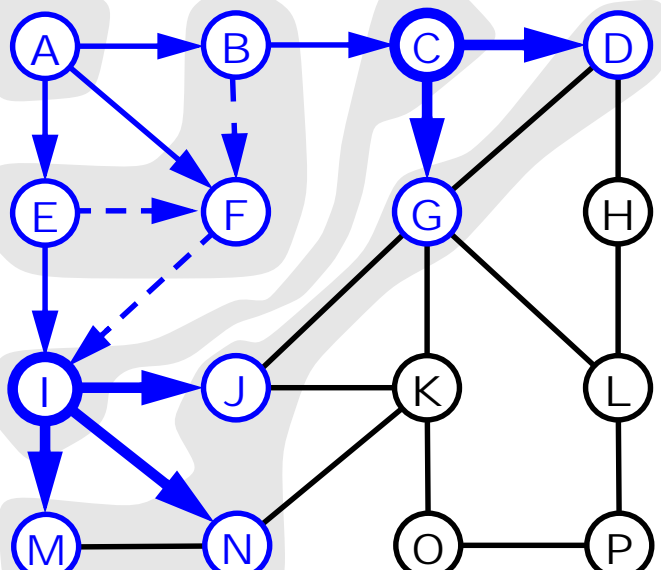
d)

0

1

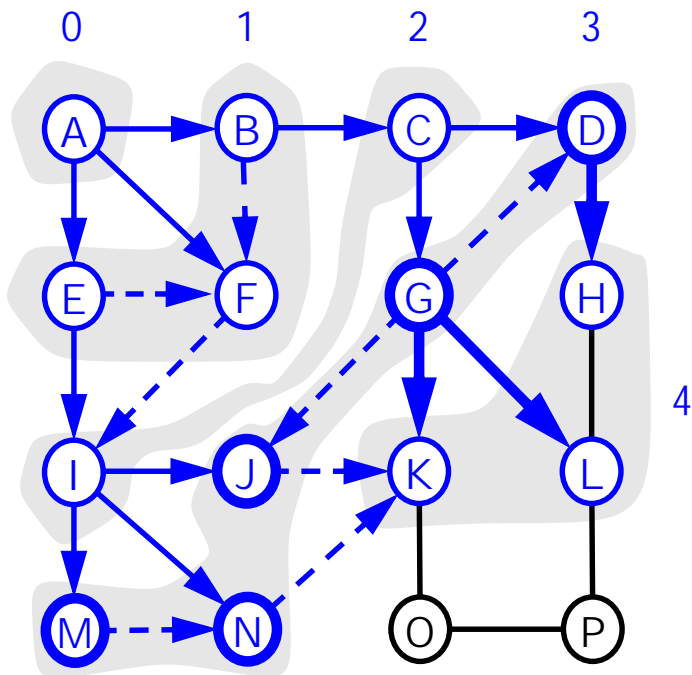
2

3

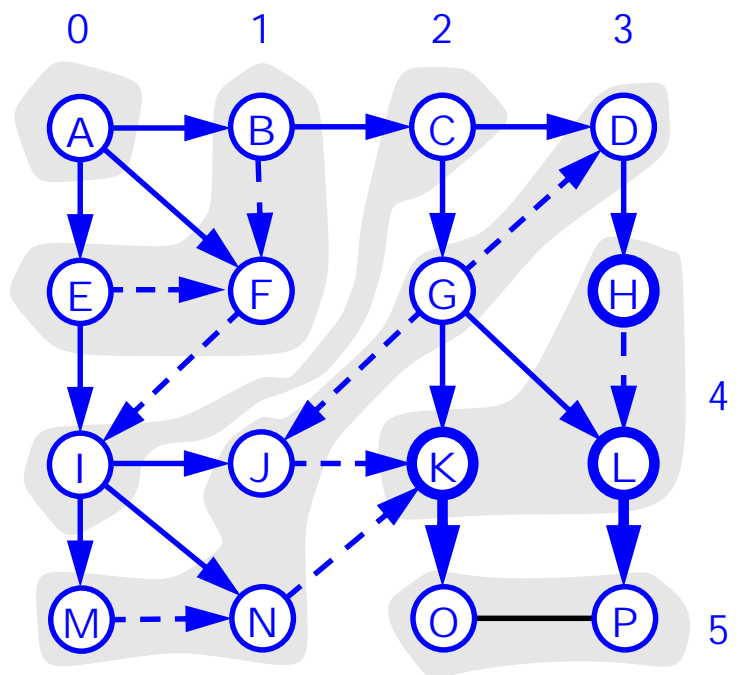


# More BFS

e)



f)



# BFS Pseudo-Code

Algorithm **BFS**( $s$ ):

**Input:** A vertex  $s$  in a graph

**Output:** A labeling of the edges as “discovery” edges  
and “cross edges”

initialize container  $L_0$  to contain vertex  $s$

$i \leftarrow 0$

while  $L_i$  is not empty do

    create container  $L_{i+1}$  to initially be empty

    for each vertex  $v$  in  $L_i$  do

        if edge  $e$  incident on  $v$  do

            let  $w$  be the other endpoint of  $e$

            if vertex  $w$  is unexplored then

                label  $e$  as a discovery edge

                insert  $w$  into  $L_{i+1}$

            else

                label  $e$  as a cross edge

$i \leftarrow i + 1$

# Properties of BFS

- **Proposition:** Let  $G$  be an undirected graph on which a BFS traversal starting at vertex  $s$  has been performed. Then
  - The traversal visits all vertices in the connected component of  $s$ .
  - The discovery-edges form a spanning tree  $T$ , which we call the BFS tree, of the connected component of  $s$
  - For each vertex  $v$  at level  $i$ , the path of the BFS tree  $T$  between  $s$  and  $v$  has  $i$  edges, and any other path of  $G$  between  $s$  and  $v$  has at least  $i$  edges.
  - If  $(u, v)$  is an edge that is not in the BFS tree, then the level numbers of  $u$  and  $v$  differ by at most one.
- **Proposition:** Let  $G$  be a graph with  $n$  vertices and  $m$  edges. A BFS traversal of  $G$  takes time  $O(n + m)$ . Also, there exist  $O(n + m)$  time algorithms based on BFS for the following problems:
  - Testing whether  $G$  is connected.
  - Computing a spanning tree of  $G$
  - Computing the connected components of  $G$
  - Computing, for every vertex  $v$  of  $G$ , the minimum number of edges of any path between  $s$  and  $v$ .