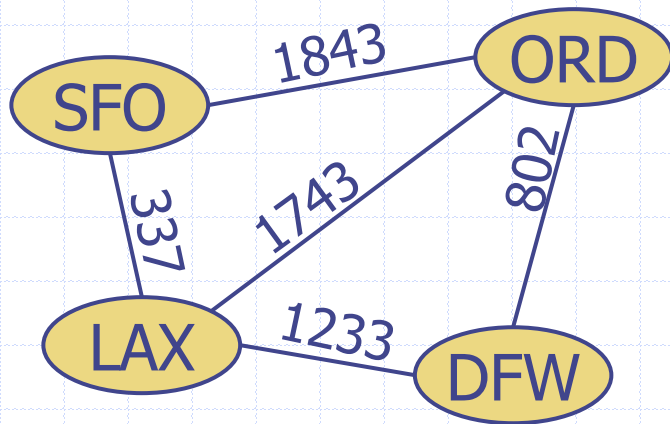


# Lecture 13:

## BFS Graph Traversal & Templates

Principle of  
Transcending



# Wholeness Statement

Graphs have many useful applications in different areas of computer science. However, to be useful we have to be able to traverse them. One of the two primary ways that graphs are systematically explored, is using the breadth-first search algorithm. *Science of Consciousness*: The TM technique provides a simple, effortless way to systematically explore the different levels of the conscious mind until the process of thinking is transcended and unbounded silence is experienced; this is the field of wholeness of individual and cosmic intelligence.

# List of Terms

- ◆ Graph
  - ◆ Vertex, vertices
  - ◆ End vertices
  - ◆ Adjacent vertices
  - ◆ Degree of a vertex
- ◆ Edges
  - ◆ Incident edges
  - ◆ Directed edge, undirected edge
  - ◆ Directed graph, undirected graph, mixed graph
- ◆ Path, simple path
- ◆ Cycle, simple cycle

# More Terms

- ◆ Subgraph

- ◆ Connectivity

- Connected Vertices (path between them)
- Connected Graph (all vertices are connected)
- Connected Component (maximal connected subgraph)

- ◆ Tree (connected, no cycles)

- ◆ Forest (one or more trees)

- ◆ Spanning Tree and Spanning Forest

# Breadth-First Search Outline and Reading

## ◆ Breadth-first search

- Example
- Algorithm
- Properties
- Analysis
- Applications

## ◆ DFS vs. BFS

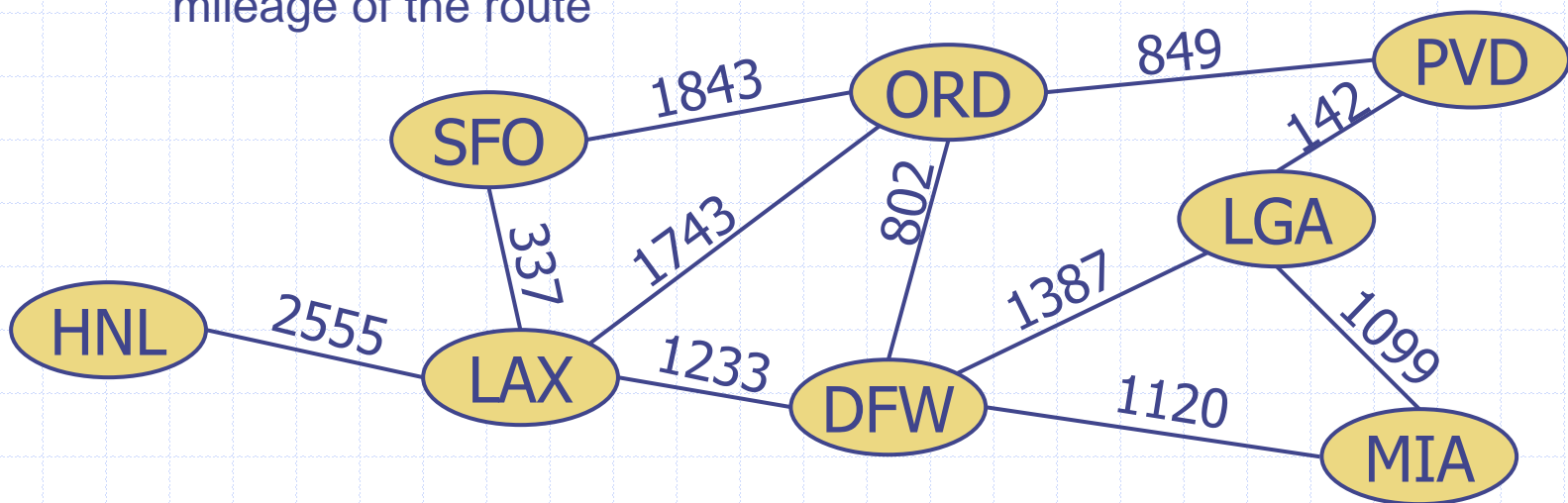
- Comparison of applications
- Comparison of edge labels

# Graph

- ◆ A graph is a pair  $(V, E)$ , where
  - $V$  is a set of nodes, called **vertices**
  - $E$  is a collection of pairs of vertices, called **edges**
  - Vertices and edges are positions and store elements

- ◆ Example:

- A vertex represents an airport and stores the three-letter airport code
- An edge represents a flight route between two airports and stores the mileage of the route



# Properties

## Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

## Property 2

In an undirected graph with no self-loops and no parallel edges

$$m \leq n(n-1)/2$$

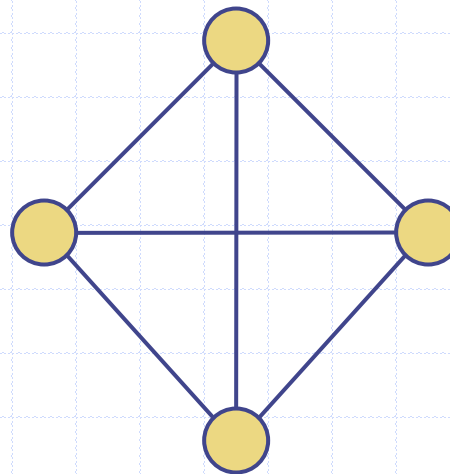
Proof: each vertex has degree at most  $(n-1)$

What is the bound for a directed graph?

$$m \leq n(n-1)$$

## Notation

$n$	number of vertices
$m$	number of edges
$\deg(v)$	degree of vertex $v$



## Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

# Main Methods of the Undirected Graph ADT

## ◆ Vertices and edges

- are Positions
- store elements

## ◆ Accessor methods

- `aVertex()`
- `incidentEdges(v)`
- `endVertices(e)`
- `opposite(v, e)`
- `areAdjacent(v, w)`

## ◆ Update methods

- `insertVertex(o)`
- `insertEdge(v, w, o)`
- `removeVertex(v)`
- `removeEdge(e)`

## ◆ Generic methods

- `numVertices()`
- `numEdges()`
- `vertices()`
- `edges()`
- `degree(v)`

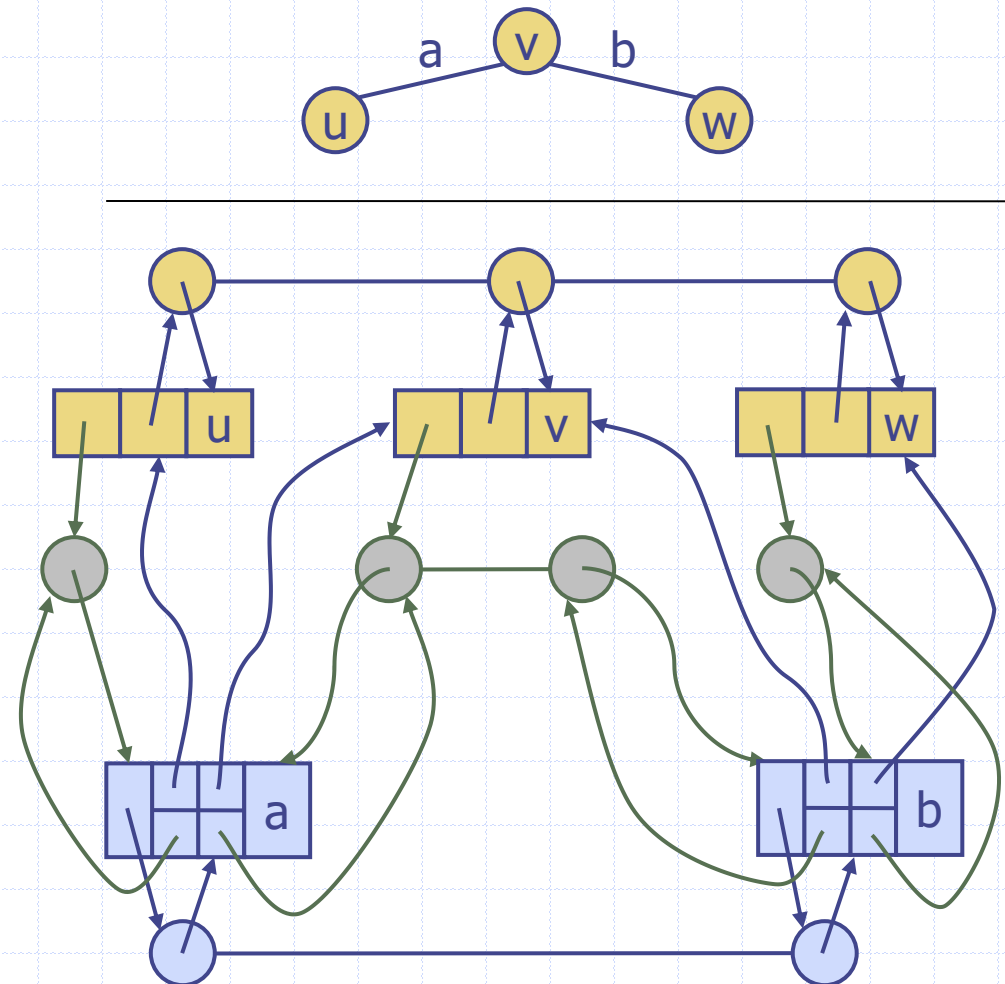


# Graph Data Structures

Adjacency list

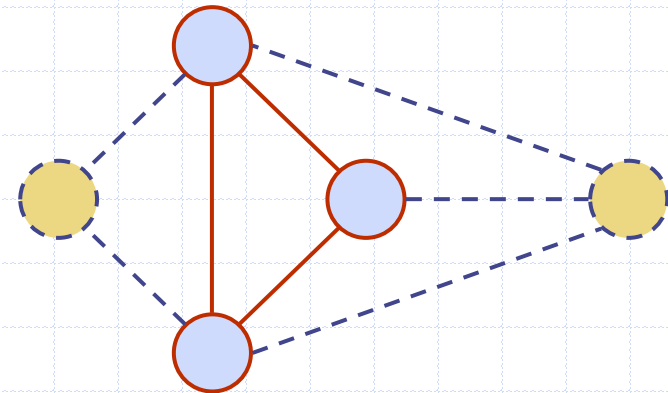
# Adjacency List Structure

- ◆ Edge list structure
- ◆ Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- ◆ Augmented edge objects
  - references to associated positions in incidence sequences of end vertices

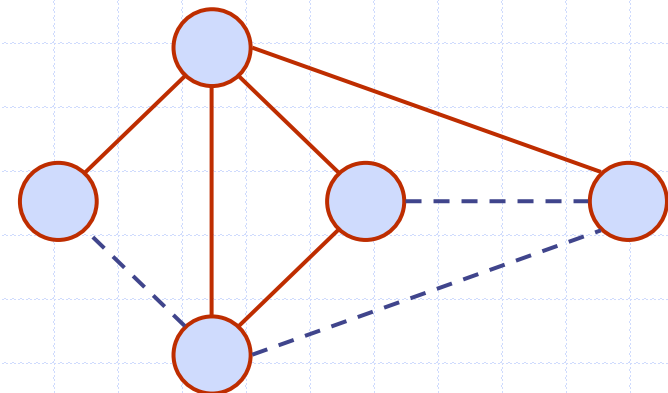


# Subgraphs

- ◆ A subgraph  $S$  of a graph  $G$  is a graph such that
  - $\text{vertices}(S) \subseteq \text{vertices}(G)$
  - $\text{edges}(S) \subseteq \text{edges}(G)$
- ◆ A spanning subgraph of  $G$  is a subgraph that contains all the vertices of  $G$ , i.e.,  $\text{vertices}(S) = \text{vertices}(G)$



Subgraph



Spanning subgraph

# Trees and Forests

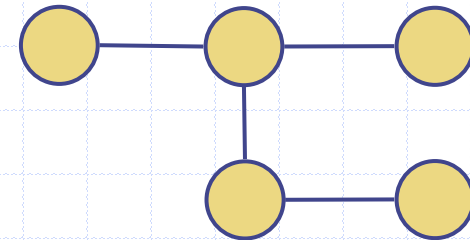
◆ A (free) *tree* is an **undirected** graph  $T$  such that

- $T$  is **connected**
- $T$  has no **cycles**

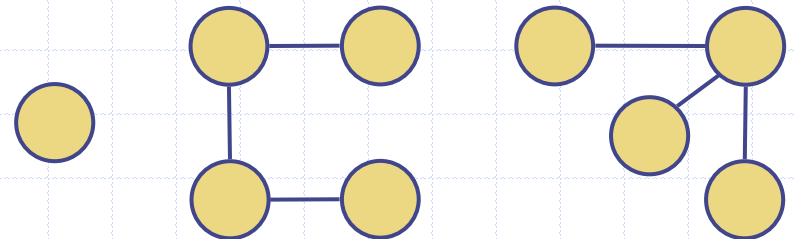
This definition is different from the definition of a rooted tree

◆ A *forest* is an undirected graph without cycles

◆ The connected components of a forest are trees



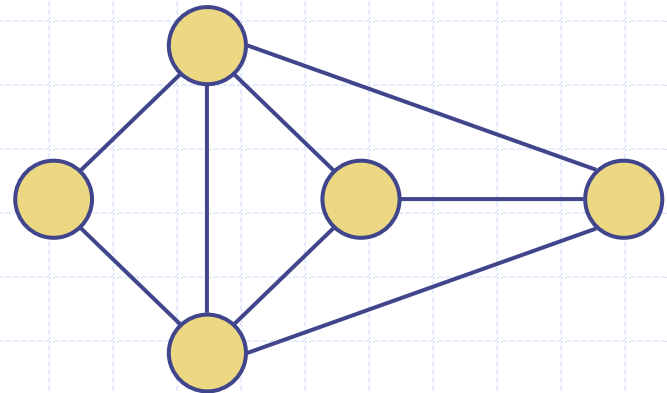
Tree



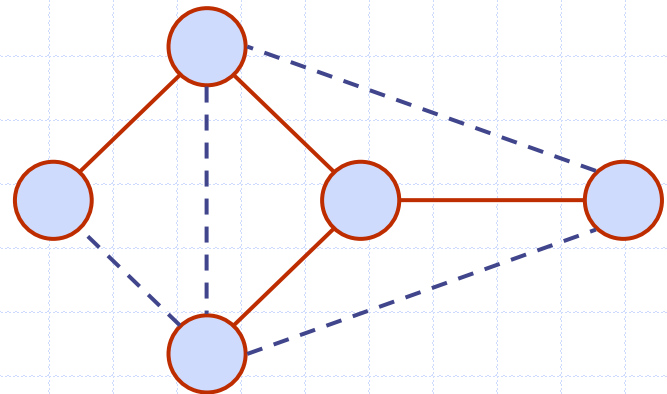
Forest

# Spanning Trees and Forests

- ◆ A spanning tree of a connected graph is a spanning subgraph that is a tree
- ◆ A spanning tree is not unique unless the graph is a tree
- ◆ Spanning trees have applications to the design of communication networks
- ◆ A spanning forest of a graph is a spanning subgraph that is a forest

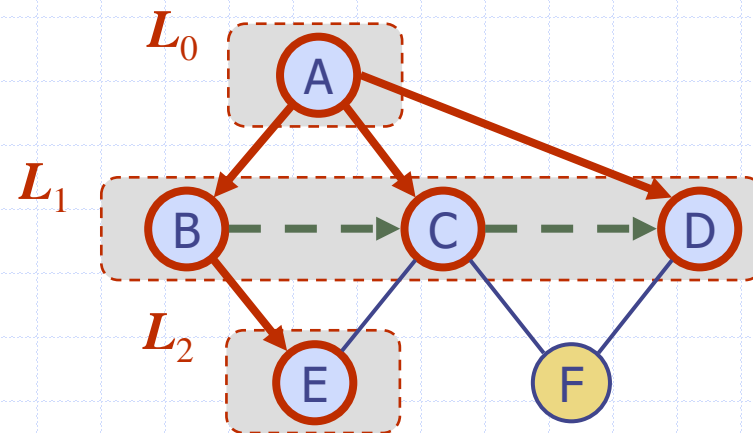


Graph



Spanning tree

# Breadth-First Search



# Example



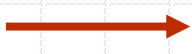
unexplored vertex



visited vertex



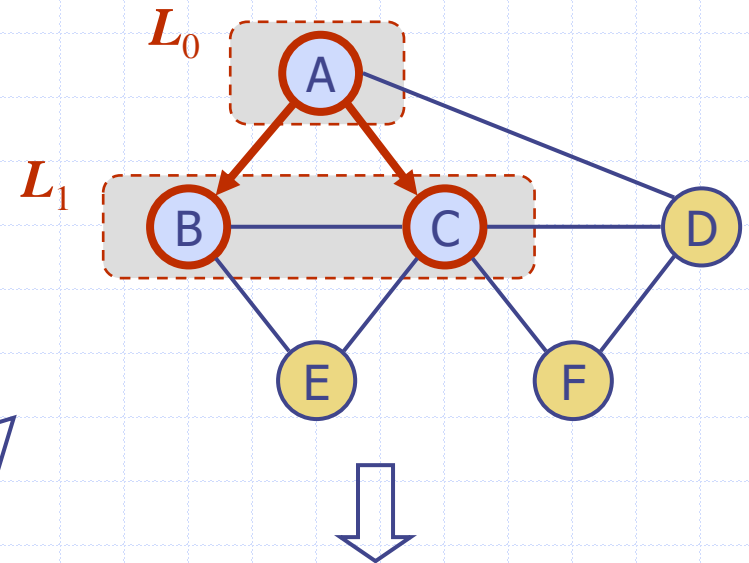
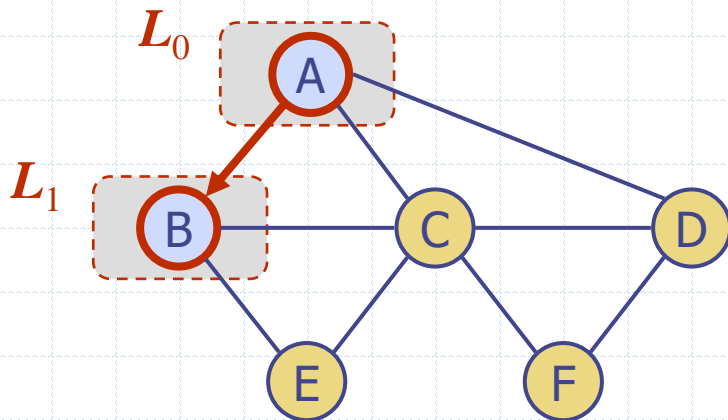
unexplored edge



discovery edge



cross edge



# Example



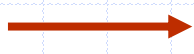
unexplored vertex



visited vertex



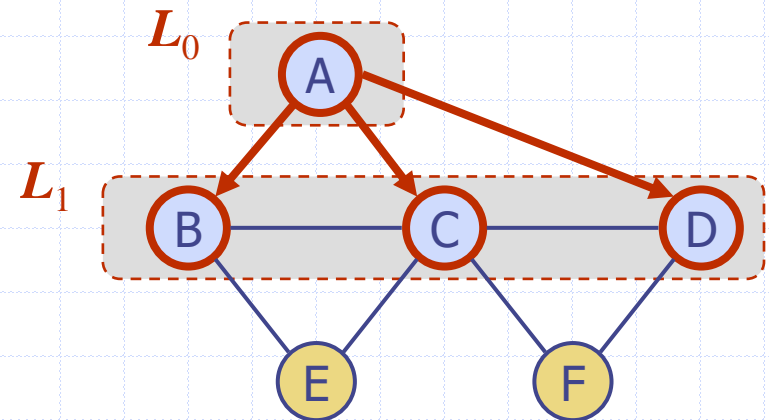
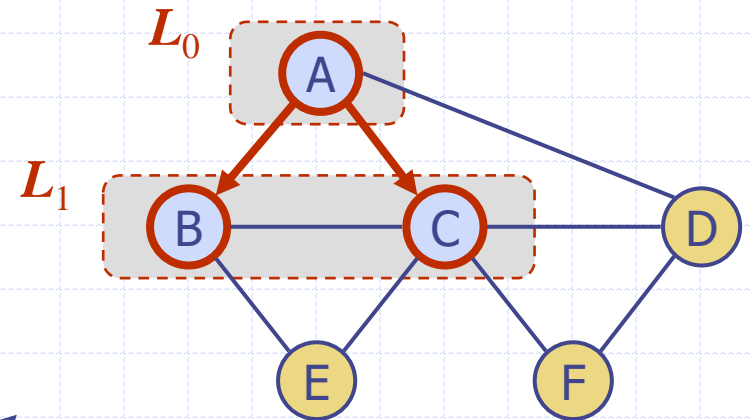
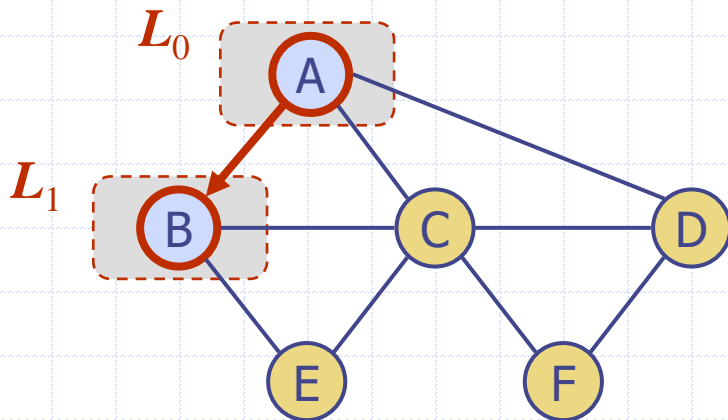
unexplored edge



discovery edge

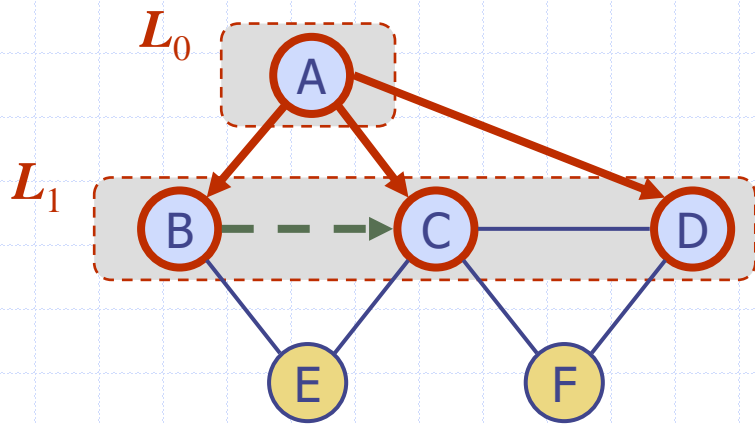


cross edge

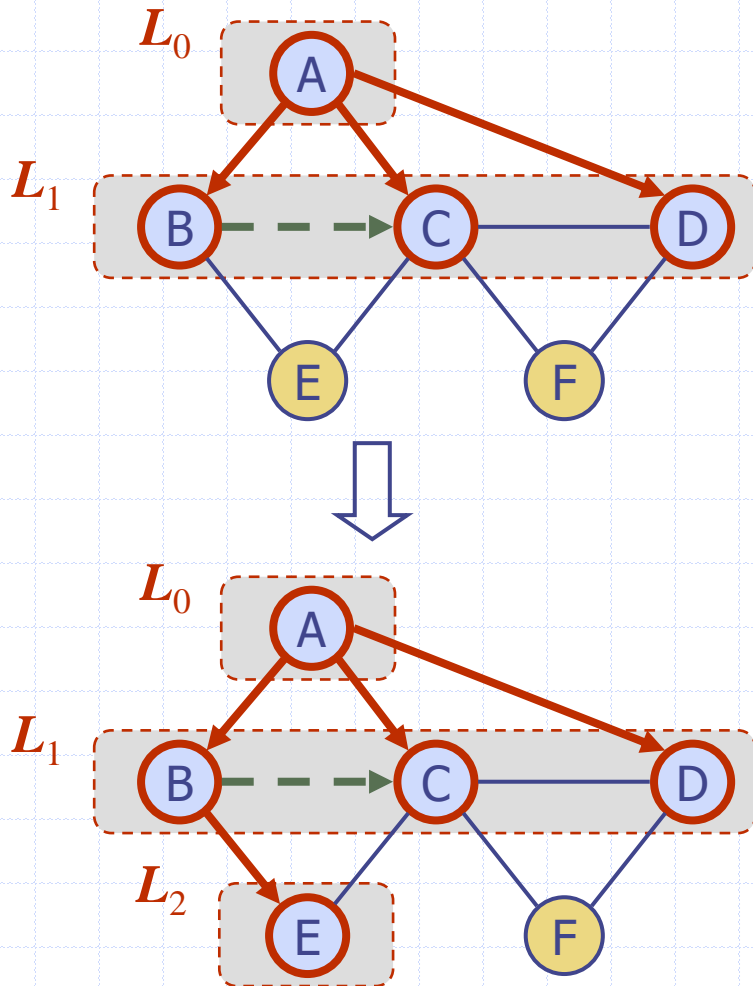




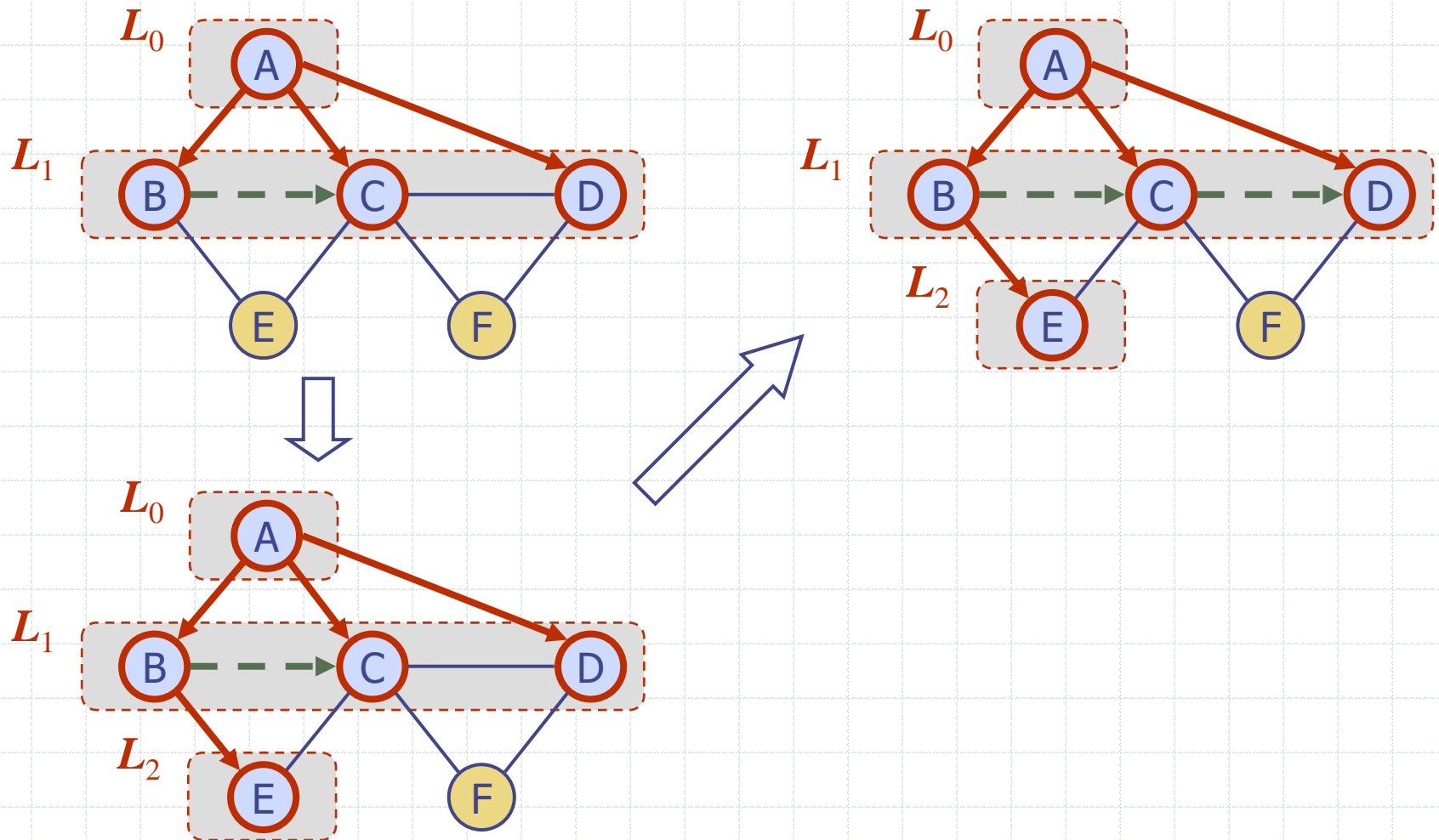
# Example (cont.)



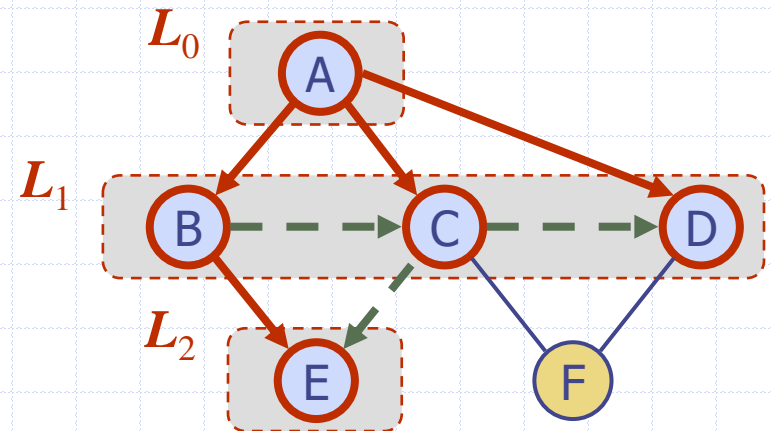
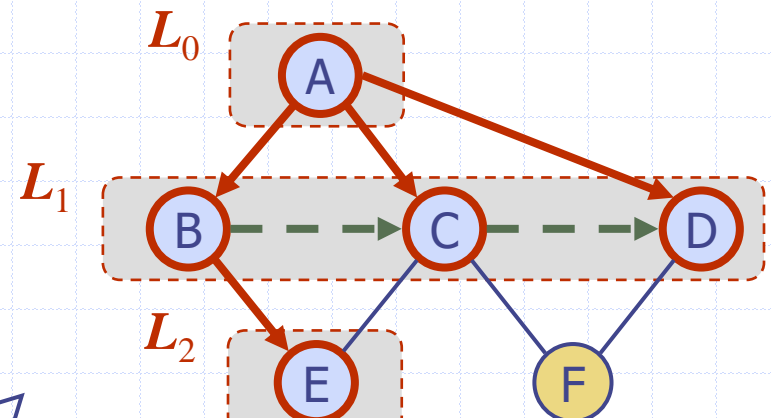
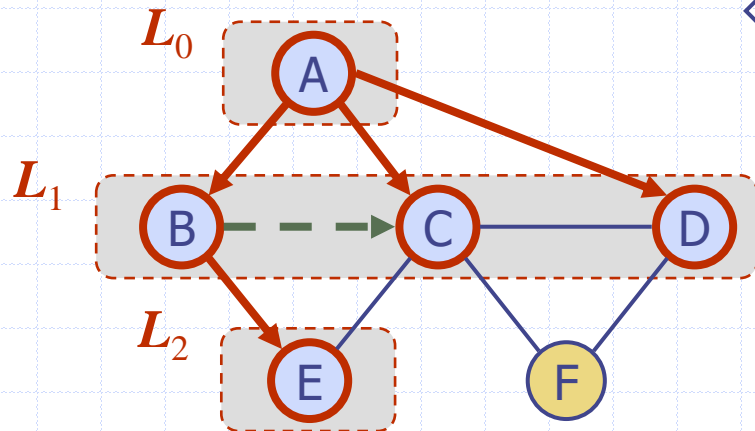
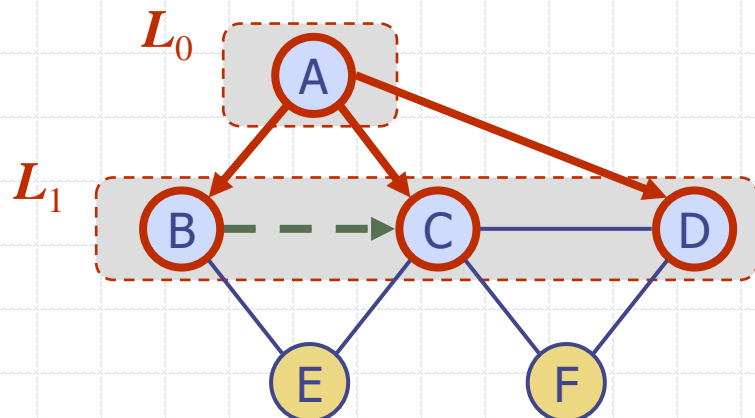
# Example (cont.)



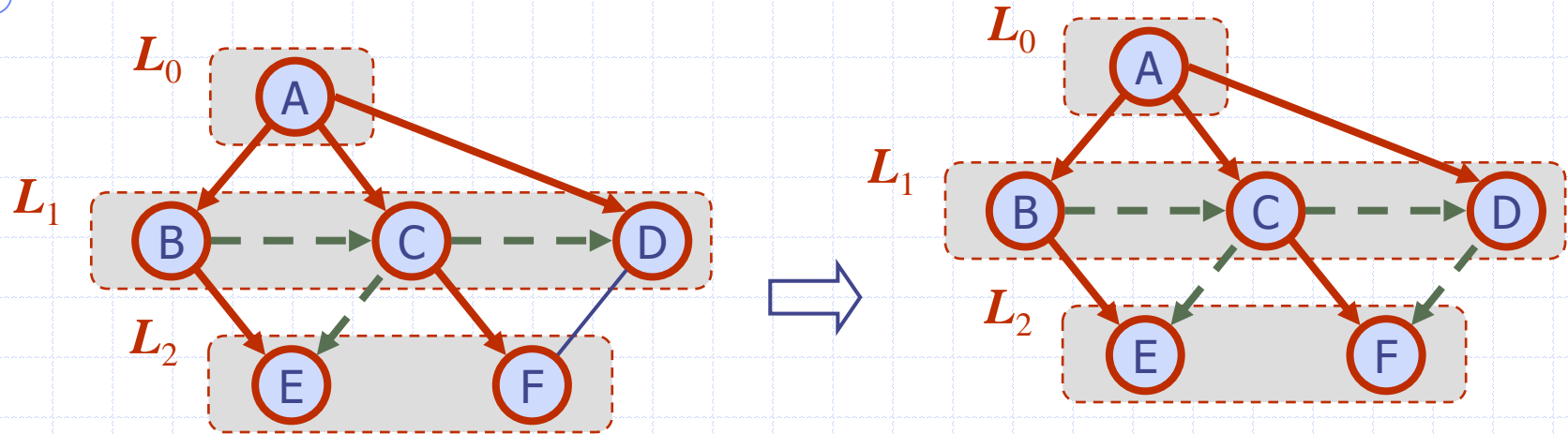
# Example (cont.)



# Example (cont.)

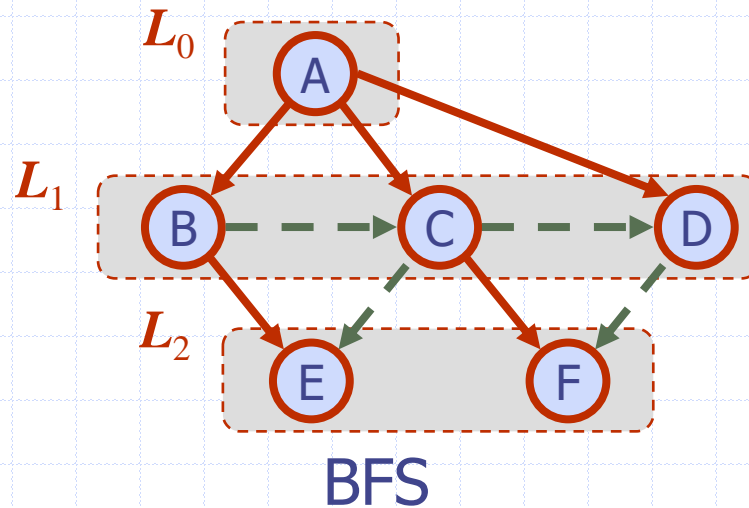


# Example (cont.)



# BFS Levels

When actually implemented, the levels are normally merged into a single list/queue



# BFS Algorithm

- ◆ The BFS algorithm using a single list/sequence/Queue

## Algorithm **BFS**( $G$ )

**Input** graph  $G$

**Output** labeling of the edges  
and partition of the  
vertices of  $G$

```
for all  $u \in G.vertices()$  do
     $setLabel(u, UNEXPLORED)$ 
for all  $e \in G.edges()$  do
     $setLabel(e, UNEXPLORED)$ 
for all  $v \in G.vertices()$  do
    if  $getLabel(v) = UNEXPLORED$ 
        BFSComponent( $G, v$ )
```

## Algorithm **BFSComponent**( $G, s$ )

```
 $Q \leftarrow$  new empty Queue
 $Q.enqueue(s)$ 
 $setLabel(s, VISITED)$ 
while  $Q.size() > 0$  do
     $v \leftarrow Q.dequeue()$ 
    for all  $e \in G.incidentEdges(v)$  do
        if  $getLabel(e) = UNEXPLORED$  then
             $w \leftarrow G.opposite(v, e)$ 
            if  $getLabel(w) = UNEXPLORED$ 
                then
                     $setLabel(e, DISCOVERY)$ 
                     $setLabel(w, VISITED)$ 
                     $Q.enqueue(w)$ 
            else
                 $setLabel(e, CROSS)$ 
```

# Properties

## Notation

$G_s$ : connected component of  $s$

## Property 1

**BFScomponent**( $G, s$ ) visits all the vertices and edges of  $G_s$

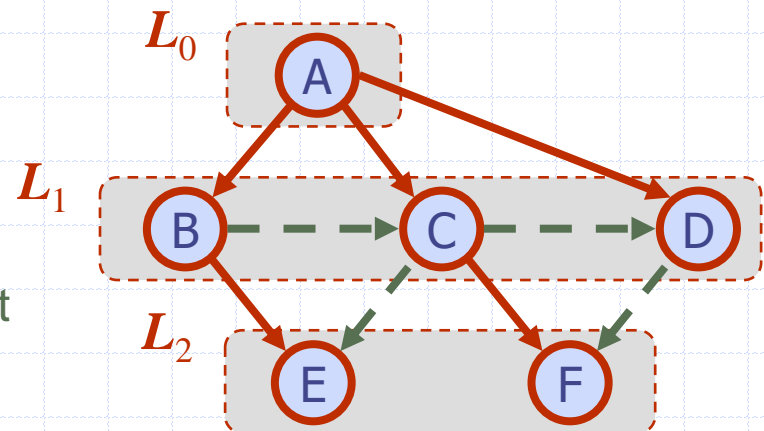
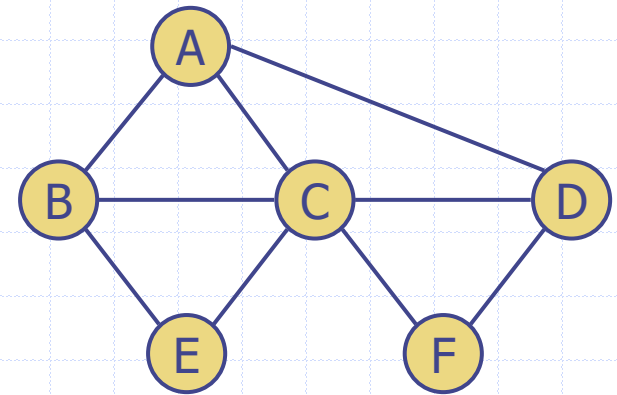
## Property 2

The discovery edges labeled by **BFScomponent**( $G, s$ ) form a spanning tree  $T_s$  of  $G_s$

## Property 3

For each vertex  $v$  in  $L_i$

- The path of  $T_s$  from  $s$  to  $v$  has  $i$  edges
- Every path from  $s$  to  $v$  in  $G_s$  has at least  $i$  edges





# Analysis

- ◆ Setting/getting a vertex/edge label takes  $O(1)$  time
- ◆ Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- ◆ Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- ◆ Each vertex is inserted once into the queue  $Q$
- ◆ Method `incidentEdges` is called once for each vertex
- ◆ BFS runs in  $O(n + m)$  time provided the graph is represented by the adjacency list structure
  - Recall that  $\sum_v \deg(v) = 2m$

# Breadth-First Search

- ◆ Breadth-first search (BFS) is a general technique for traversing a graph
- ◆ A BFS traversal of a graph  $G$ 
  - Visits all the vertices and edges of  $G$
  - Determines whether  $G$  is connected
  - Computes the connected components of  $G$
  - Computes a spanning forest of  $G$

# Breadth-First Search

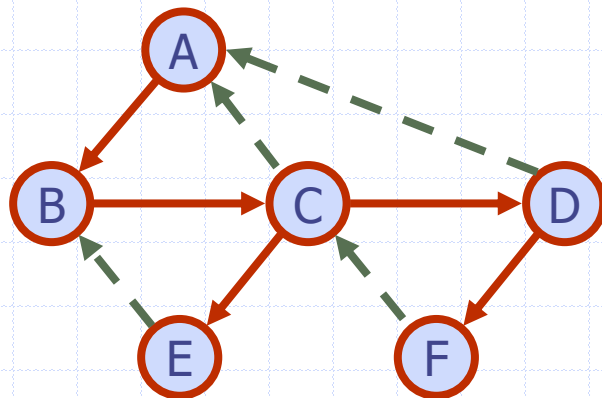
- ◆ BFS on a graph with  $n$  vertices and  $m$  edges takes  $O(n + m)$  time
- ◆ BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

# Applications

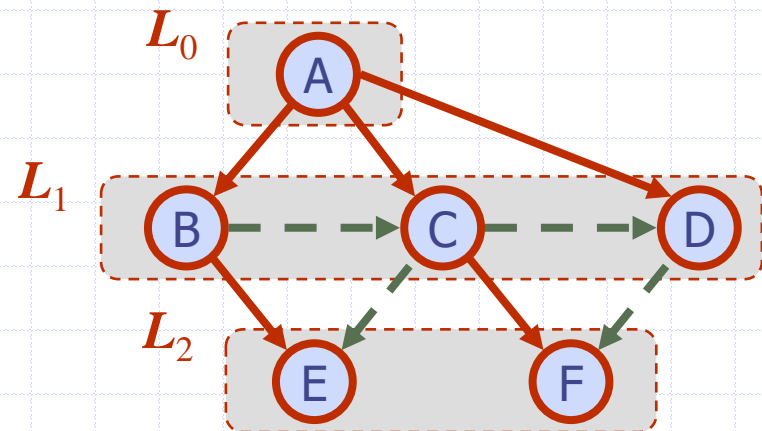
- ◆ Using the template method pattern, we can specialize the BFS traversal of a graph  $G$  to solve the following problems in  $O(n + m)$  time
  - Compute the connected components of  $G$
  - Compute a spanning forest of  $G$
  - Find a simple cycle in  $G$ , or report that  $G$  is a forest
  - Given two vertices of  $G$ , find a path in  $G$  between them with the minimum number of edges, or report that no such path exists

# DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	



DFS

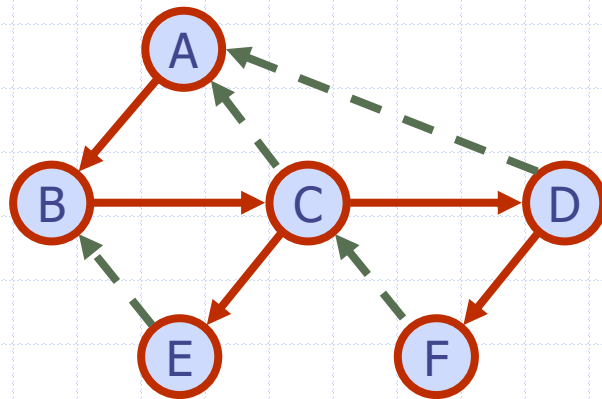


BFS

# DFS vs. BFS (cont.)

## Back edge ( $v, w$ )

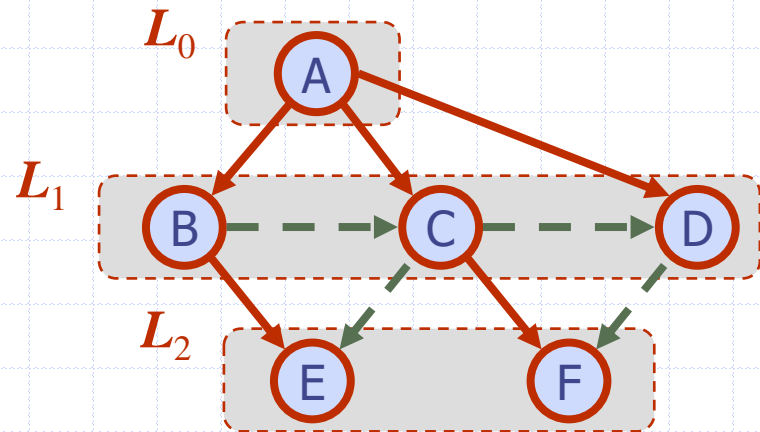
- $w$  is an ancestor of  $v$  in the tree of discovery edges



DFS

## Cross edge ( $v, w$ )

- $w$  is in the same level as  $v$  or in the next level in the tree of discovery edges



BFS

# Main Point

1. During breadth-first search of a graph, the search repeatedly takes one step in all directions until all vertices and edges are visited. This is a bit like searching for fulfillment in waking state, i.e., floating on the surface of the mind or through one's daily activity.

*Science of Consciousness:* In contrast, Transcendental Meditation takes the mind immediately and effortlessly to the deepest levels where true fulfillment can be gained.

# Template Method Pattern

Depth-first search is to graphs  
what the Euler tour is to binary  
trees



# Recall Our Earlier Example of the Template Method Pattern in Java

```
public abstract class EulerTour {  
    protected void visitExternal(BinaryTree tree, Position p, Object[ ] r) {}  
    protected void visitPreOrder(BinaryTree tree, Position p, Object[ ] r) {}  
    protected void visitInOrder(BinaryTree tree, Position p, Object[ ] r) {}  
    protected void visitPostOrder(BinaryTree tree, Position p, Object[ ] r) {}  
    protected Object eulerTour(BinaryTree tree, Position p) {  
        Object[ ] result = new Object[3];  
        if tree.isExternal(p) { visitExternal(tree, p, result); }  
        else {  
            visitPreOrder(tree, p, result);  
            result[1] = eulerTour(tree, tree.leftChild(p));  
            visitInOrder(tree, p, result);  
            result[2] = eulerTour(tree, tree.rightChild(p));  
            visitPostOrder(tree, p, result);  
        }  
        return result[0];  
    }  
}
```

# Template Method Pattern

- ◆ Generic algorithm that can be specialized by redefining certain steps
- ◆ Implemented by means of an abstract Java class
- ◆ Visit methods that can be redefined/overridden by subclasses
- ◆ Template method `eulerTour`
  - Recursively called on the left and right children
  - A `result` array that keeps track of the output of the recursive calls to `eulerTour`
  - `result[0]` keeps track of the final output of the `eulerTour` method
  - `result[1]` keeps track of the output of the recursive call of `eulerTour` on the left child
  - `result[2]` keeps track of the output of the recursive call of `eulerTour` on the right child

# Specializations of EulerTour

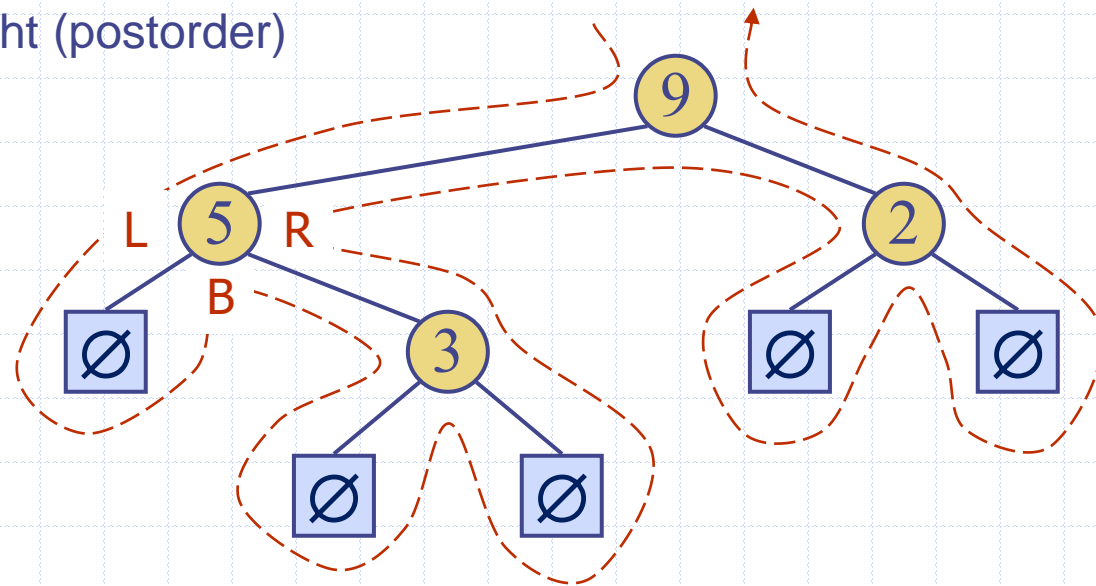
```
public class Sum extends EulerTour {  
    // Sums the integers in a Binary Tree of Integers  
  
    public Integer sum(BinaryTree tree) {  
        return eulerTour(tree, tree.root());  
    }  
  
    protected void visitExternal(BinaryTree t, Position p, Object[ ] res) {  
        result[0] = new Integer(0);  
    }  
  
    protected void visitPostOrder(BinaryTree t, Position p, Object[ ] result) {  
        result[0] = (Integer) result[1] + (Integer) result[2] + p.element()  
    }  
    ...  
}
```

# Specializations of EulerTour

```
public class Sum extends EulerTour {  
    // Sums the integers in a Binary Tree of Integers (another way)  
    public Integer sum(BinaryTree tree) {  
        return eulerTour(tree, tree.root());  
    }  
    protected void visitExternal(BinaryTree t, Position p, Object[ ] result) {  
        result[0] = new Integer(0);  
    }  
    protected void visitPreOrder(BinaryTree t, Position p, Object[ ] result) {  
        result[0] = p.element()  
    }  
    protected void visitInOrder(BinaryTree t, Position p, Object[ ] result) {  
        result[0] = (Integer) result[1] + (Integer) result[0]  
    }  
    protected void visitPostOrder(BinaryTree t, Position p, Object[ ] result) {  
        result[0] = (Integer) result[2] + (Integer) result[0]  
    }  
}
```

# Euler Tour Traversal

- ◆ Generic traversal of a binary tree
- ◆ Includes as special cases the preorder, postorder, and inorder traversals
- ◆ Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



# Exercise on Binary Trees

- ◆ Generic methods:
  - integer `size()`
  - boolean `isEmpty()`
  - objectIterator `elements()`
  - positionIterator `positions()`
- ◆ Accessor methods:
  - position `root()`
  - position `parent(p)`
  - positionIterator `children(p)`
- ◆ Query methods:
  - boolean `isInternal(p)`
  - boolean `isExternal(p)`
  - boolean `isRoot(p)`
- ◆ Update methods:
  - `swapElements(p, q)`
  - object `replaceElement(p, o)`
- ◆ Additional BinaryTree methods:
  - position `leftChild(p)`
  - position `rightChild(p)`
  - position `sibling(p)`

## Exercise:

- ◆ Write a method to calculate the height of a binary tree

Algorithm `height(T)`

Hint: you also need a helper function with argument Position `p`

Algorithm `heightHelper(T, p)`

# Example

- ◆ Using the template, design a Java method **height**(T) to calculate the height of a given binary tree T.

# Example

```
class Height extends EulerTour { // too much Java
```

```
    Object height(T) {  
        return eulerTour(T, T.root());  
    }  
}
```

- We want to abstract away as many details as we can when designing without omitting too many details;
- This is why we use pseudo code



# Euler Tour Template (pseudo-code)

**Algorithm** *eulerTour*(T, v)

result ← new Array(3)    // 3 element array

**if** T.isExternal(v) **then**

*visitExternal*(T, v, result)

**else**

*visitPreOrder*(T, v, result)

    result[1] ← *eulerTour*(T, T.leftChild(v))

*visitInOrder*(T, v, result)

    result[2] ← *eulerTour*(T, T.rightChild(v))

*visitPostOrder*(T, v, result)

**return** result[0]

# Exercise

- ◆ Using the template, design a pseudo code algorithm **height**(T) to calculate the height of a given tree T.

# Specialization (Subclass) of EulerTour

- ◆ We show how to specialize class EulerTour to calculate the height of a binary tree
- ◆ Create a subclass Height of EulerTour

```
// class Height extends EulerTour
```

```
Algorithm height(T) // always need top level method
```

```
    return eulerTour(T, T.root()) // need to call template method
```

```
Algorithm visitExternal(T, p, result) // override hook method to insert actions
```

```
    result[0] = 0
```

```
Algorithm visitPostOrder(T, p, result) // override hook method to insert actions
```

```
    result[0] = 1 + MAX(result[1], result[2])
```

# Template Version of DFS

## Exercise: write `isConnected(G)`

Algorithm *DFS*(*G*)

**Input** graph *G*

**Output** the edges of *G* are  
labeled as discovery edges  
and back edges

*initResult*( *G* )

```
for all u ∈ G.vertices()  
    setLabel(u, UNEXPLORED)  
    preInitVertex(u)  
for all e ∈ G.edges()  
    setLabel(e, UNEXPLORED)  
    preInitEdge(e)  
for all v ∈ G.vertices()  
    if getLabel(v) = UNEXPLORED  
        preComponentVisit(G, v)  
        DFScomponent(G, v)  
        postComponentVisit(G, v)
```

**return** *result*( *G* )

Algorithm *DFScomponent*(*G*, *v*)

*setLabel*(*v*, VISITED)

*startVertexVisit*(*G*, *v*)

for all *e* ∈ *G.incidentEdges*(*v*)

*preEdgeVisit*(*G*, *v*, *e*)

if *getLabel*(*e*) = UNEXPLORED

*w* ← *opposite*(*v*, *e*)

*edgeVisit*(*G*, *v*, *e*, *w*)

if *getLabel*(*w*) = UNEXPLORED

*setLabel*(*e*, DISCOVERY)

*preDiscoveryVisit*(*G*, *v*, *e*, *w*)

*DFScomponent*(*G*, *w*)

*postDiscoveryVisit*(*G*, *v*, *e*, *w*)

else

*setLabel*(*e*, BACK)

*backEdgeVisit*(*G*, *v*, *e*, *w*)

*finishVertexVisit*(*G*, *v*)

# Path Finding

## Override hook operations

```
Algorithm DFSComponent( $G, v$ )
  setLabel( $v, VISITED$ )
  startVertexVisit( $G, v$ )
  for all  $e \in G.incidentEdges(v)$ 
    preEdgeVisit( $G, v, e$ )
    if getLabel( $e$ ) = UNEXPLORED
       $w \leftarrow opposite(v, e)$ 
      edgeVisit( $G, v, e, w$ )
      if getLabel( $w$ ) = UNEXPLORED
        setLabel( $e, DISCOVERY$ )
        preDiscoveryVisit( $G, v, e, w$ )
        DFSComponent( $G, w$ )
        postDiscoveryVisit( $G, v, e, w$ )
      else
        setLabel( $e, BACK$ )
        backEdgeVisit( $G, v, e, w$ )
  finishVertexVisit( $G, v$ )
```

```
Algorithm pathDFS( $G, v, z, S$ )
  setLabel( $v, VISITED$ )
   $S.push(v)$ 
  if  $v = z$  then
     $path \leftarrow S.elements()$ 
  for all  $e \in G.incidentEdges(v)$  do
    if getLabel( $e$ ) = UNEXPLORED then
       $w \leftarrow opposite(v, e)$ 
      if getLabel( $w$ ) = UNEXPLORED then
        setLabel( $e, DISCOVERY$ )
         $S.push(e)$ 
        pathDFS( $G, w, z, S$ )
         $S.pop()$  {  $e$  must be popped }
      else
        setLabel( $e, BACK$ )
   $S.pop()$  {  $v$  must be popped }
```

# Overriding hook methods in a subclass FindSimplePath

Algorithm **findSimplePath**(*G*, *u*, *v*) // always need top level method that calls DFS  
    *S* ← new empty stack {*S* is a subclass field}  
    *z* ← *v* {*z* is a subclass field & is the target vertex}  
    *path* ← ∅ {*path* is a subclass field & is the path from *u* to *v*}  
    for all *u* ∈ *G.vertices*()  
        setLabel(*u*, UNEXPLORED)  
    for all *e* ∈ *G.edges*()  
        setLabel(*e*, UNEXPLORED)  
    DFScomponent(*G*, *u*)  
    return(*path*)

Algorithm **startVertexVisit**(*G*, *v*)  
    *S*.push(*v*)  
    if *v*=*z* then {*z* is a subclass field & is the target}  
        *path* ← *S*.elements() {*path* is a subclass field & is the result}

Algorithm **preDiscoveryVisit**(*G*, *v*, *e*, *w*)  
    *S*.push(*e*)

Algorithm **postDiscoveryVisit**(*G*, *v*, *e*, *w*)  
    *S*.pop() {pop *e* off the stack}

Algorithm **finishVertexVisit**(*G*, *v*)  
    *S*.pop() {pop *v* off the stack}

# Template Version of DFS (v2)

Algorithm *DFS*(*G*)

**Input** graph *G*

**Output** the edges of *G* are  
labeled as discovery edges  
and back edges

*initResult*( *G* )

```
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
    preInitVertex(u)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
    preInitEdge(e)
for all v ∈ G.vertices()
    if isNextComponent(G, v)
        preComponentVisit(G, v)
        DFScomponent(G, v)
        postComponentVisit(G, v)
```

**return** *result*( *G* )

Algorithm *isNextComponent*(*G*, *v*)

**return** *getLabel*(*v*) = UNEXPLORED

Algorithm *DFScomponent*(*G*, *v*)

*setLabel*(*v*, VISITED)

*beginVertexVisit*(*G*, *v*)

**for all** *e* ∈ *G.incidentEdges*(*v*)

*preEdgeVisit*(*G*, *v*, *e*, *w*)

**if** *getLabel*(*e*) = UNEXPLORED

*w* ← *opposite*(*v*, *e*)

*edgeVisit*(*G*, *v*, *e*, *w*)

**if** *getLabel*(*w*) = UNEXPLORED

*setLabel*(*e*, DISCOVERY)

*preDiscoveryVisit*(*G*, *v*, *e*, *w*)

*DFScomponent*(*G*, *w*)

*postDiscoveryVisit*(*G*, *v*, *e*, *w*)

**else**

*setLabel*(*e*, BACK)

*backEdgeVisit*(*G*, *v*, *e*, *w*)

*finishVertexVisit*(*G*, *v*)

# Overriding hook methods in a subclass FindSimplePath (v2)

Algorithm **findSimplePath**(*G*, *u*, *v*) // always need top level method that calls DFS

start $\leftarrow$ <i>u</i>	{start is a subclass field & is the starting vertex}
dest $\leftarrow$ <i>v</i>	{dest is a subclass field & is the destination vertex}
<i>S</i> $\leftarrow$ new empty stack	{ <i>S</i> is a subclass field}
path $\leftarrow$ $\emptyset$	{path is a subclass field & is the path from <i>u</i> to <i>v</i> }
return DFS( <i>G</i> )	

Algorithm **result**(*G*)

return(path)

Algorithm **isNextComponent**(*G*, *v*)

return *v*=start {start the component traversal at vertex start}

Algorithm **beginVertexVisit**(*G*, *v*)

*S*.push(*v*)  
if *v*=dest then {dest is a subclass field & is the destination vertex}  
    path  $\leftarrow$  *S*.elements() {path is a subclass field & is the result}

Algorithm **preDiscoveryVisit**(*G*, *v*, *e*, *w*)

*S*.push(*e*)

Algorithm **postDiscoveryVisit**(*G*, *v*, *e*, *w*)

*S*.pop() {pop *e* off the stack}

Algorithm **finishVertexVisit**(*G*, *v*)

*S*.pop() {pop *v* off the stack}



# Overriding hook methods in a subclass FindSimplePath (v3)

Algorithm **findSimplePath**(*G*, *u*, *v*) // always need top level method that calls DFS  
    *start*  $\leftarrow$  *u*                                    {start is a subclass field & is the starting vertex}  
    *dest*  $\leftarrow$  *v*                                    {dest is a subclass field & is the destination vertex}  
    **return** *DFS*(*G*)

Algorithm **initResult**(*G*) // simplify top level by moving as much as possible to template methods  
    *S*  $\leftarrow$  new empty stack                        {S is a subclass field}  
    *path*  $\leftarrow$   $\emptyset$                                 {path is a subclass field & is the path from u to v}

Algorithm **result**(*G*)  
    **return**(*path*)

Algorithm **isNextComponent**(*G*, *v*)  
    **return** *v*=*start*                                {start the component traversal at vertex start}

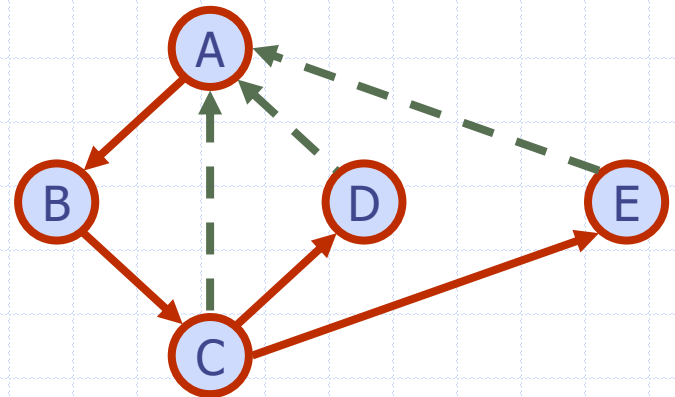
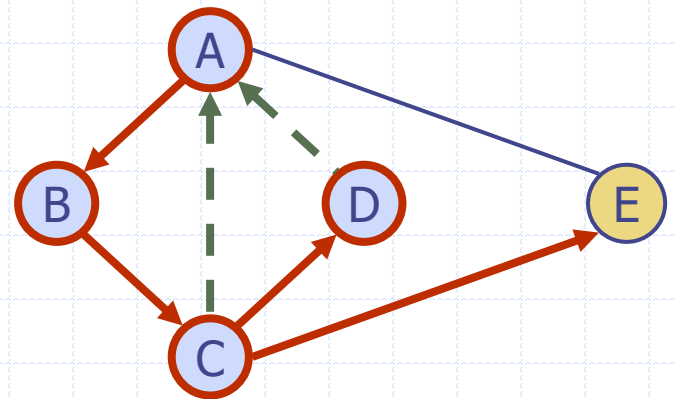
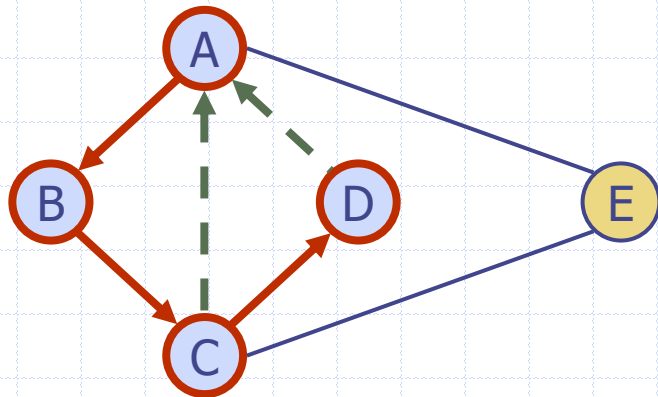
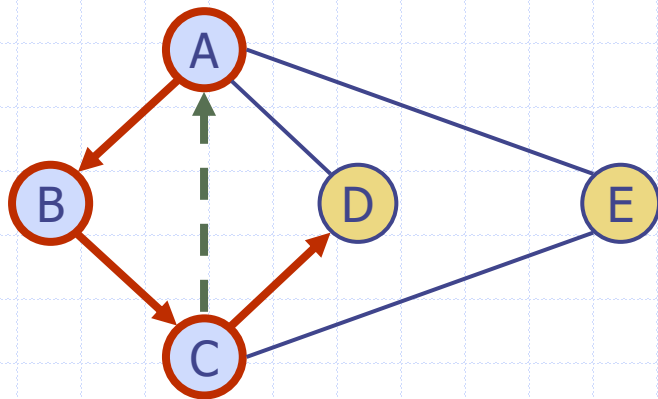
Algorithm **beginVertexVisit**(*G*, *v*)  
    *S*.push(*v*)  
    if *v*=*dest* then                                {dest is a subclass field & is the destination vertex}  
        *path*  $\leftarrow$  *S*.elements() {path is a subclass field & is the result}

Algorithm **preDiscoveryVisit**(*G*, *v*, *e*, *w*)  
    *S*.push(*e*)

Algorithm **postDiscoveryVisit**(*G*, *v*, *e*, *w*)  
    *S*.pop()                                        {pop *e* off the stack}

Algorithm **finishVertexVisit**(*G*, *v*)  
    *S*.pop()                                        {pop *v* off the stack}

# DFS Example (cont.)



# Template Version of DFS (v2)

Algorithm *DFS*(*G*)

**Input** graph *G*

**Output** the edges of *G* are  
labeled as discovery edges  
and back edges

*initResult*( *G* )

```
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
    preInitVertex(u)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
    preInitEdge(e)
for all v ∈ G.vertices()
    if isNextComponent(G, v)
        preComponentVisit(G, v)
        DFScomponent(G, v)
        postComponentVisit(G, v)
```

**return** *result*( *G* )

Algorithm *isNextComponent*(*G*, *v*)

**return** *getLabel*(*v*) = UNEXPLORED

Algorithm *DFScomponent*(*G*, *v*)

*setLabel*(*v*, VISITED)

*beginVertexVisit*(*G*, *v*)

**for all** *e* ∈ *G.incidentEdges*(*v*)

*preEdgeVisit*(*G*, *v*, *e*, *w*)

**if** *getLabel*(*e*) = UNEXPLORED

*w* ← *opposite*(*v*, *e*)

*edgeVisit*(*G*, *v*, *e*, *w*)

**if** *getLabel*(*w*) = UNEXPLORED

*setLabel*(*e*, DISCOVERY)

*preDiscoveryVisit*(*G*, *v*, *e*, *w*)

*DFScomponent*(*G*, *w*)

*postDiscoveryVisit*(*G*, *v*, *e*, *w*)

**else**

*setLabel*(*e*, BACK)

*backEdgeVisit*(*G*, *v*, *e*, *w*)

*finishVertexVisit*(*G*, *v*)



# Exercise: Cycle Finding

## Override hook operations

```
Algorithm DFSComponent(G, v)
  setLabel(v, VISITED)
  startVertexVisit(v)
  for all e ∈ G.incidentEdges(v)
    preEdgeVisit(G, v, e, w)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v, e)
      edgeVisit(G, v, e, w)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        preDiscoveryVisit(G, v, e, w)
        DFSComponent(G, w)
        postDiscoveryVisit(G, v, e, w)
      else
        setLabel(e, BACK)
        backEdgeVisit(G, v, e, w)
  finishVertexVisit(G, v)
```

```
Algorithm cycleDFS(G, v)
  setLabel(v, VISITED)
  if cycle ≠ null then return
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v, e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        cycleDFS(G, w)
        S.pop()
      else
        setLabel(e, BACK)
        S.push(w)
        S.push(e)
        cycle ← new empty sequence
        o ← w
        do
          cycle.insertLast(o)
          o ← S.pop()
        while o ≠ w
      return
  S.pop()
```

# Overriding template methods in subclass FindCycles Version 1

Algorithm **startVertexVisit**( $G, v$ )

if !cycleFound then S.push(v)

Algorithm **finishVertexVisit**( $G, v$ )

if !cycleFound then S.pop()

Algorithm **preDiscoveryVisit**( $G, v, e, w$ )

if !cycleFound then S.push(e)

Algorithm **postDiscoveryVisit**( $G, v, e, w$ )

if !cycleFound then S.pop()

Algorithm **backEdgeVisit**( $G, v, e, w$ )

if !cycleFound then

    S.push(e)

**cycle** ← new empty sequence

**o** ←  $w$

    do

**cycle.insertLast**(**o**)

**o** ← S.pop()

    while **o** ≠  $w$

    cycleFound ← true {cycleFound is a subclass field, initially false}

# Exercise: Cycle Finding

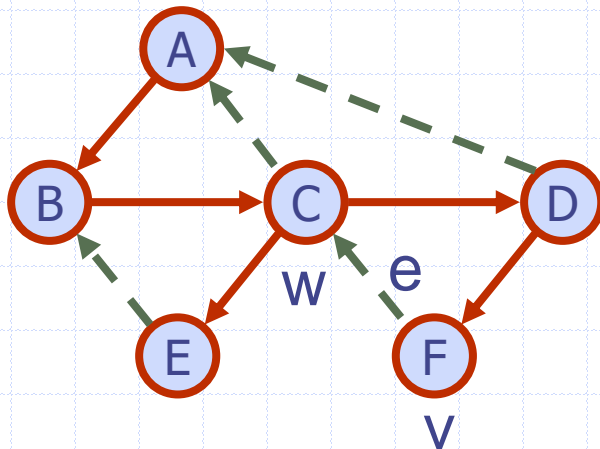
## Override hook operations

```
Algorithm DFSComponent(G, v)
  setLabel(v, VISITED)
  startVertexVisit(v)
  for all e ∈ G.incidentEdges(v)
    preEdgeVisit(G, v, e, w)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v, e)
      edgeVisit(G, v, e, w)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        preDiscoveryVisit(G, v, e, w)
        DFSComponent(G, w)
        postDiscoveryVisit(G, v, e, w)
      else
        setLabel(e, BACK)
        backEdgeVisit(G, v, e, w)
  finishVertexVisit(G, v)
```

```
Algorithm cycleDFS(G, v)
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v, e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        setParent(w, e)
        cycleDFS(G, w)
      else
        setLabel(e, BACK)
        cycle ← buildCycle(G, v, e, w)
        cycles.insertLast(cycle)
```

// *buildCycle* is defined on the next slide

# buildCycle Helper Method



DFS

```
Algorithm buildCycle( $G, v, e, w$ )  
   $cycle \leftarrow$  new empty Sequence  
   $cycle.insertLast(w)$   
   $cycle.insertLast(e)$   
   $x \leftarrow v$   
  while  $x \neq w$  do  
     $cycle.insertLast(x)$   
     $e2 \leftarrow getParent(x)$   
     $cycle.insertLast(e2)$   
     $x \leftarrow G.opposite(e2, x)$   
  
   $cycle.insertLast(w)$ 
```



- ◆ What additional method(s) do we need to create or need to override?
- ◆ We need the `findCycle(G)` method that calls `DFS(G)`
  - // always need top level method that calls DFS or BFS
- ◆ Initialize `cycleFound` boolean variable in method `initResult`
- ◆ Otherwise nothing can be executed, e.g., the hook methods are not executed

# Template Version of DFS

**Algorithm** *DFS*(*G*)

**Input** graph *G*

**Output** the edges of *G* are  
labeled as discovery edges  
and back edges

*initResult*( *G* )

```
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
    postInitVertex(u)
for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
    postInitEdge(e)
for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        preComponentVisit(G, v)
        DFSComponent(G, v)
        postComponentVisit(G, v)
```

**return** *result*( *G* )

**Algorithm** *DFSComponent*(*G*, *v*)

*setLabel*(*v*, VISITED)

*startVertexVisit*(*G*, *v*)

**for all** *e* ∈ *G.incidentEdges*(*v*)

*preEdgeVisit*(*G*, *v*, *e*, *w*)

**if** *getLabel*(*e*) = UNEXPLORED

*w* ← *opposite*(*v*, *e*)

*edgeVisit*(*G*, *v*, *e*)

**if** *getLabel*(*w*) = UNEXPLORED

*setLabel*(*e*, DISCOVERY)

*preDiscoveryVisit*(*G*, *v*, *e*, *w*)

*DFSComponent*(*G*, *w*)

*postDiscoveryVisit*(*G*, *v*, *e*, *w*)

**else**

*setLabel*(*e*, BACK)

*backEdgeVisit*(*G*, *v*, *e*, *w*)

*finishVertexVisit*(*G*, *v*)

# Overriding template methods in subclass FindCycles Version 2

**Algorithm** *findCycle*(*G*) // here is the top-level method that calls DFS  
    **return** DFS(*G*)

**Algorithm** *initResult*(*G*)  
    *cycle* ← null  
    *cycleFound* ← false

**Algorithm** *result*(*G*)  
    **return** *cycle*

**Algorithm** *preDiscoveryVisit*(*G*, *v*, *e*, *w*)  
    setParent(*w*, *e*)

**Algorithm** *backEdgeVisit* (*G*, *v*, *e*, *w*)  
    **if** ! *cycleFound* **then**  
        *cycle* ← buildCycle(*G*, *v*, *e*, *w*)  
        *cycleFound* ← true // *cycleFound* is a subclass field, initially false

# FindCycles Version 3

## return as many cycles as we can

**Algorithm** *findCycle*(*G*) // here is the top-level method that calls DFS  
    **return** DFS(*G*)

**Algorithm** *initResult*(*G*)  
    *cycles*  $\leftarrow$  new empty Sequence // collect all cycles in this Sequence

**Algorithm** *result*(*G*)  
    **return** *cycles*

**Algorithm** *preDiscoveryVisit*(*G*, *v*, *e*, *w*)  
    setParent(*w*, *e*)

**Algorithm** *backEdgeVisit* (*G*, *v*, *e*, *w*)  
    *cycle*  $\leftarrow$  *buildCycle*(*G*, *v*, *e*, *w*)  
    *cycles*.insertLast(*cycle*) // collect all cycles, initially empty

# Main Point

2. The Template Method Pattern implements the changing and non-changing parts of an algorithm in the superclass; it then allows subclasses to override certain (changeable) steps of an algorithm without modifying the basic structure of the original algorithm.

*Science of Consciousness:* The changing and non-changing aspects of creation are unified in the field pure intelligence that we experience every day during our TM program.

# Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Almost any algorithm for solving a problem on a graph or digraph requires examining or processing each vertex or edge.
2. Depth-first and breadth-first search are two particularly useful and efficient search strategies requiring linear time if implemented using adjacency lists.

3. **Transcendental Consciousness** is the goal of all searches, the field of complete fulfillment.
4. **Impulses within Transcendental Consciousness**: The dynamic natural laws within this unbounded field govern all activities and evolution of the universe.
5. **Wholeness moving within itself**: In Unity Consciousness, one experiences that the self-referral activity of the unified field gives rise to the whole breadth and depth of the universe.