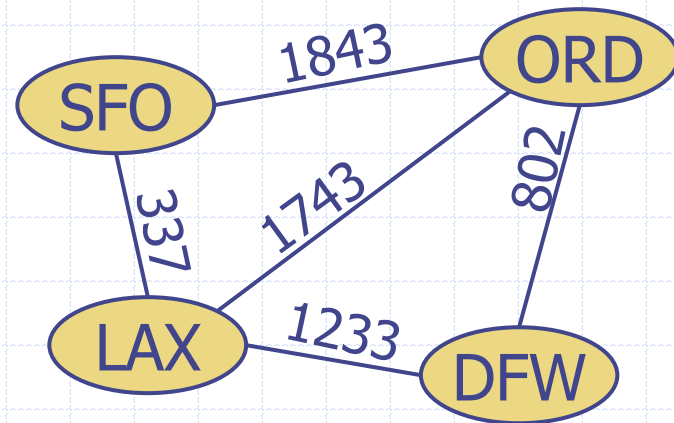# Lecture 16: BFS Graph Traversal

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. There are two primary ways that graphs are systematically explored, either using depth-first or breadth-first search. *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
  - Algorithm
  - Example
  - 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
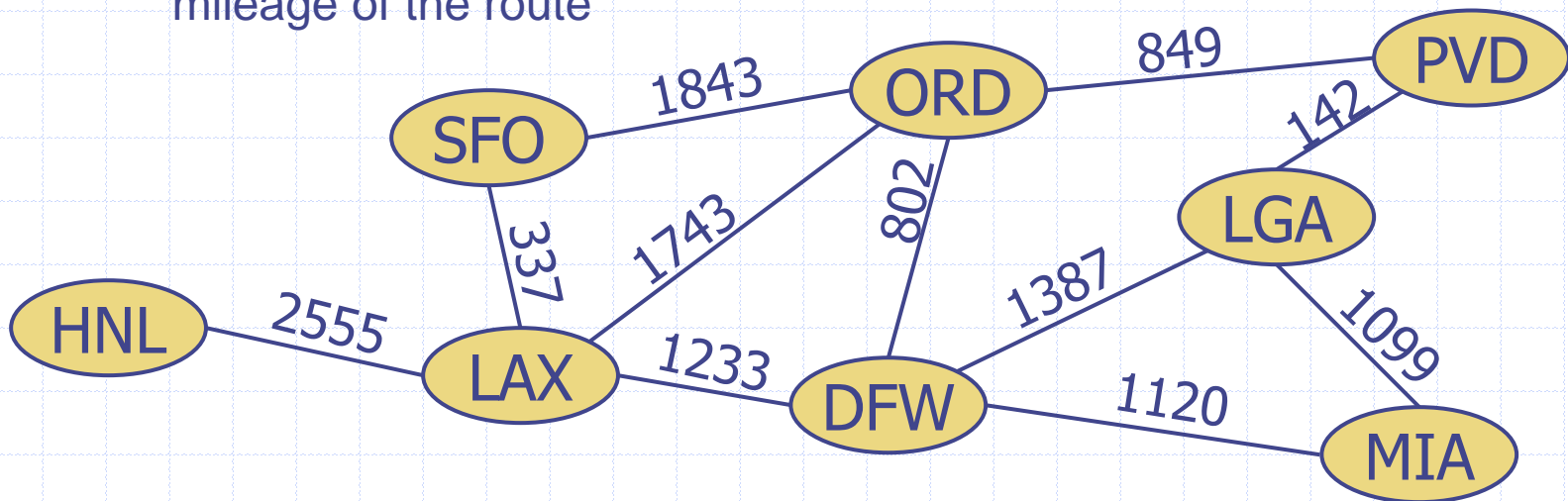
Properties

Property 1

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

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no parallel edges
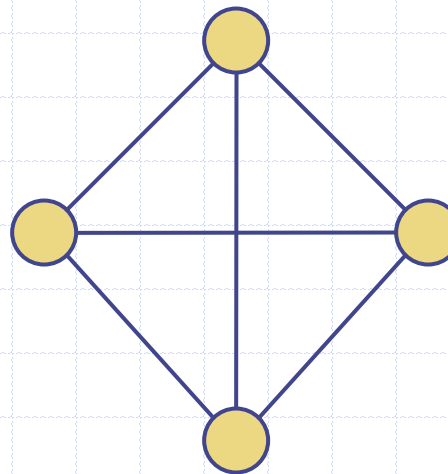
$$m \le n\,(n-1)/2$$

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

What is the bound for a directed graph?

$m \le n\ (n\text{-}1)$

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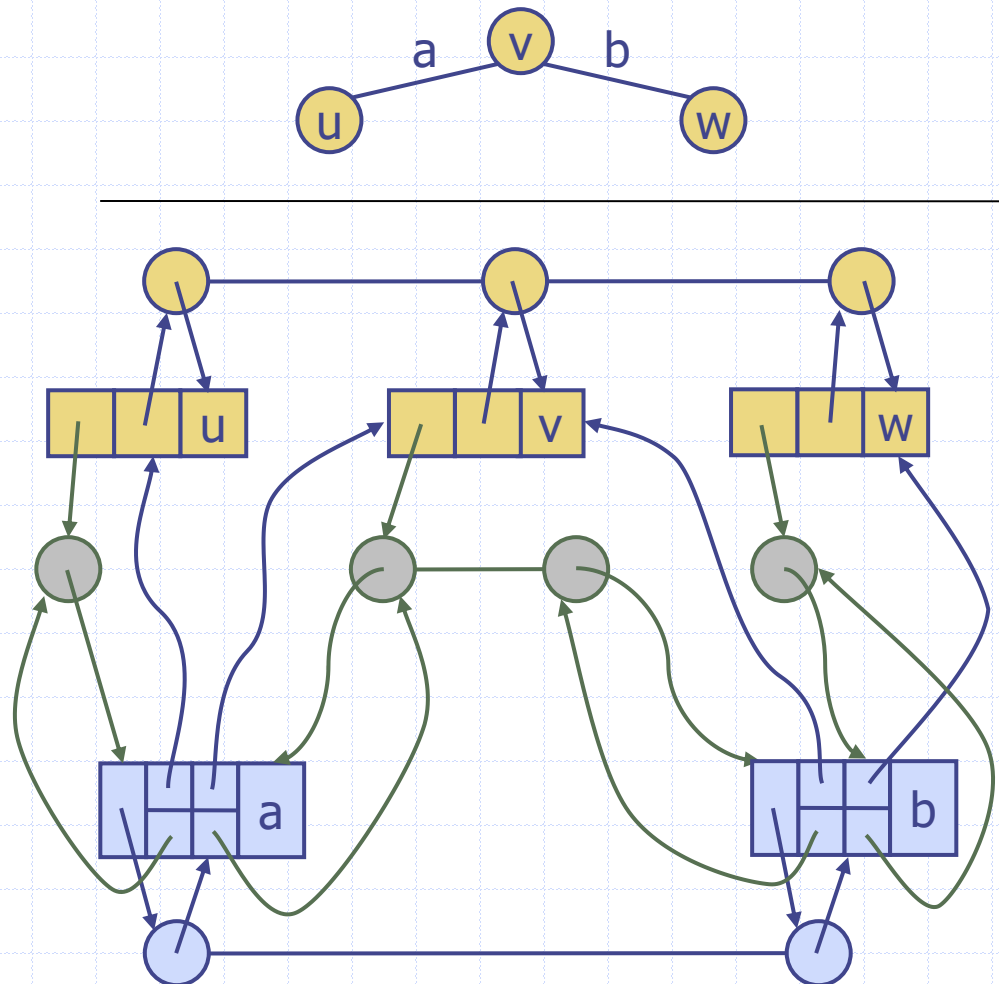 - numVertices()
  - numEdges()
  - vertices()
  - edges()
 - degree(v)

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
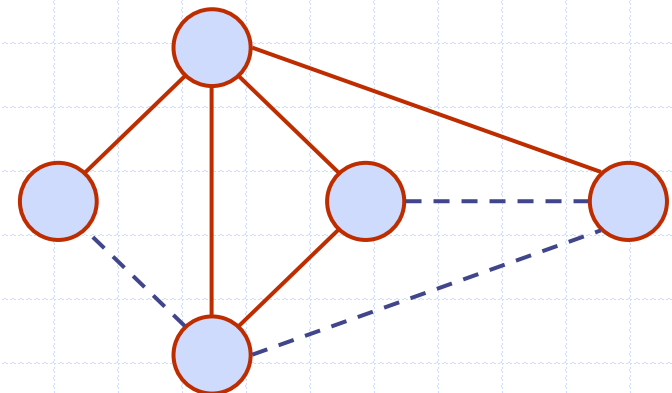 - references to associated positions in incidence sequences of end vertices

Subgraphs

◆ A subgraph S of a graph G is a graph such that
  - vertices(S) $\subseteq$ vertices(G)
  - edges(S) $\subseteq$ edges(G)

◆ A spanning subgraph of G is a subgraph that contains all the vertices of G, i.e., vertices(S) = vertices(G)
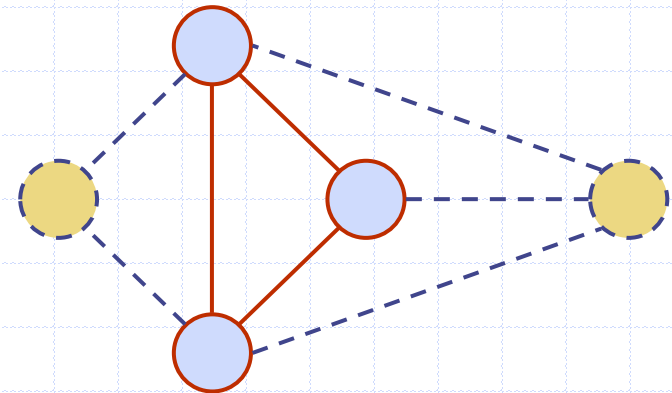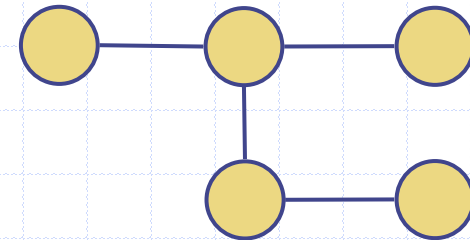
Subgraph

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
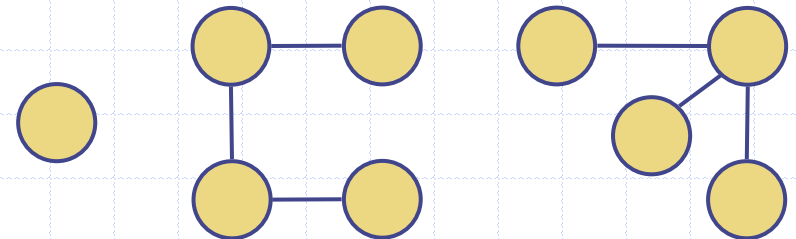
Forest

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
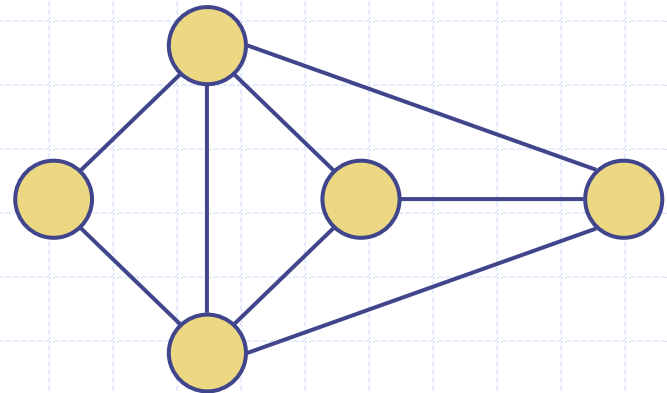
Graph
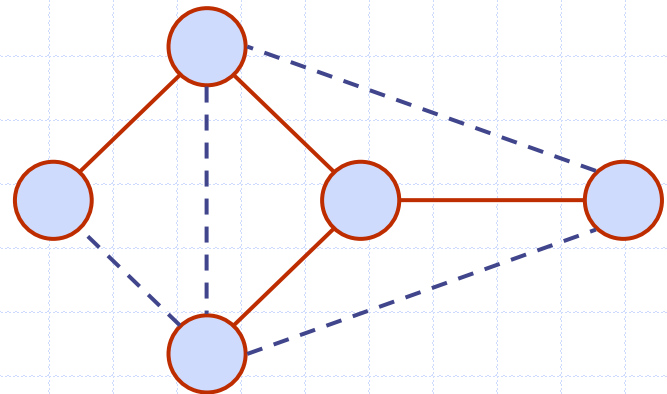
Spanning tree

Depth-First Search
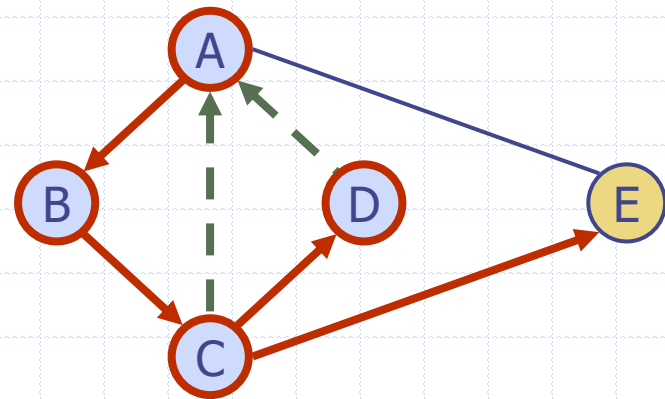
Recursive Programs

◆ The call structure can be described as a depth-first search of a rooted tree

- Each non-root vertex corresponds to a recursive call

- A tree is a logical construct, not an explicit data structure

Breadth-First Search

Example

A unexplored vertex

A visited vertex

——— unexplored edge

——▶ discovery edge

- - -▶ cross edge

L_0

L_1

Example

A — unexplored vertex

A (circled) — visited vertex
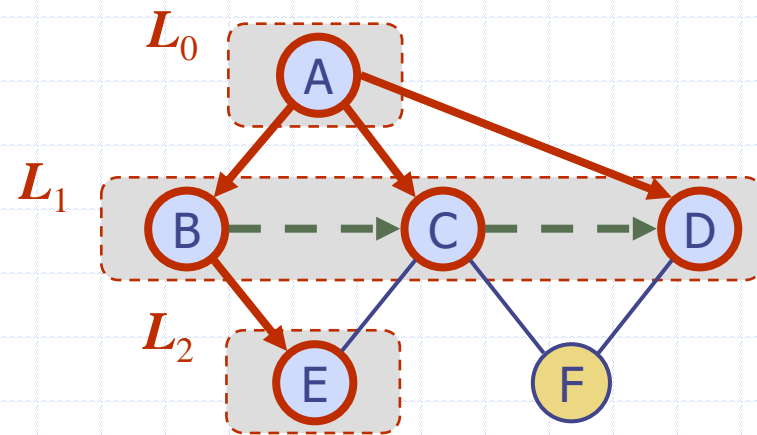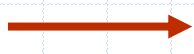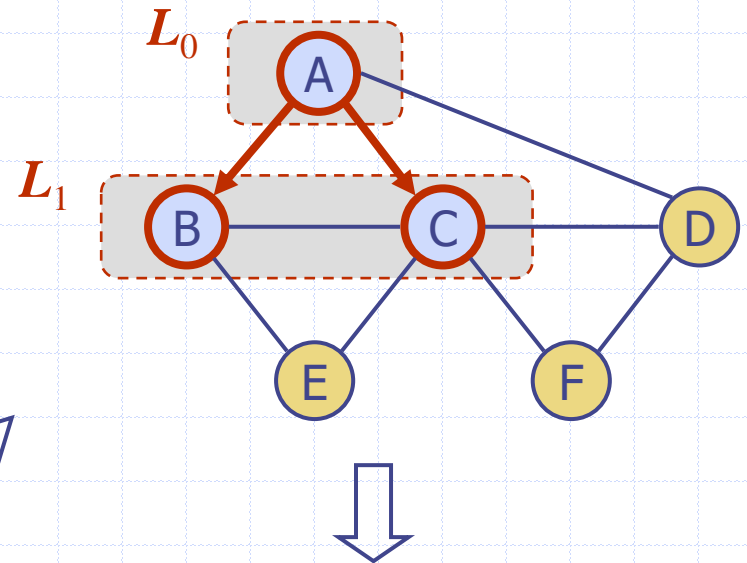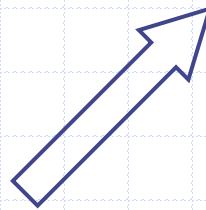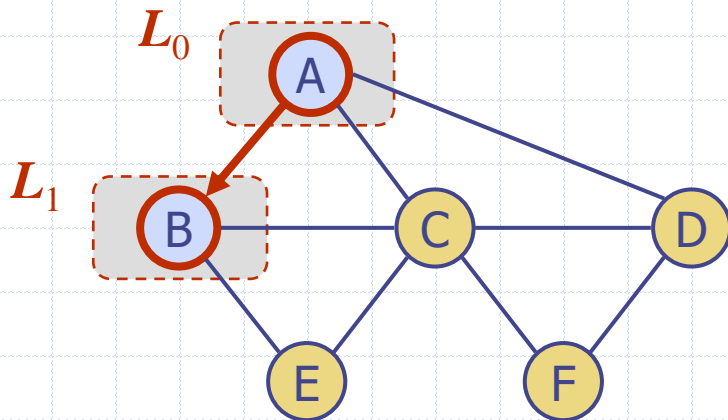
——— unexplored edge

——► discovery edge

- - -► cross edge

Example (cont.)

L_0

A

L_1

B → C → D

E   F
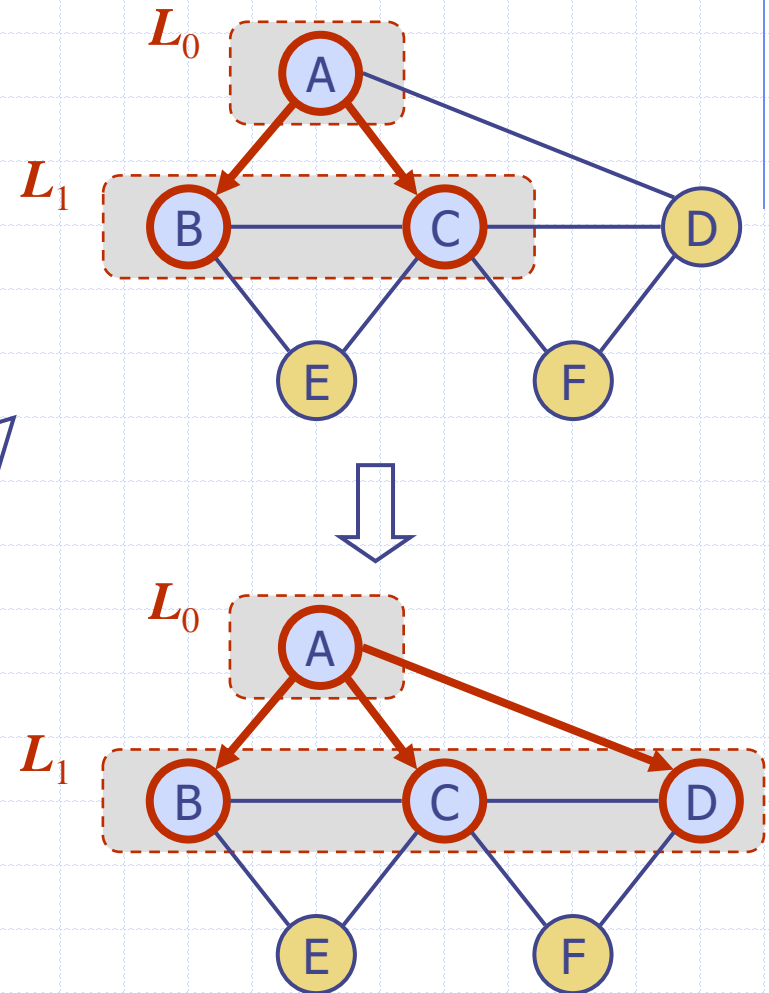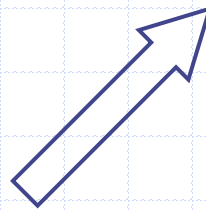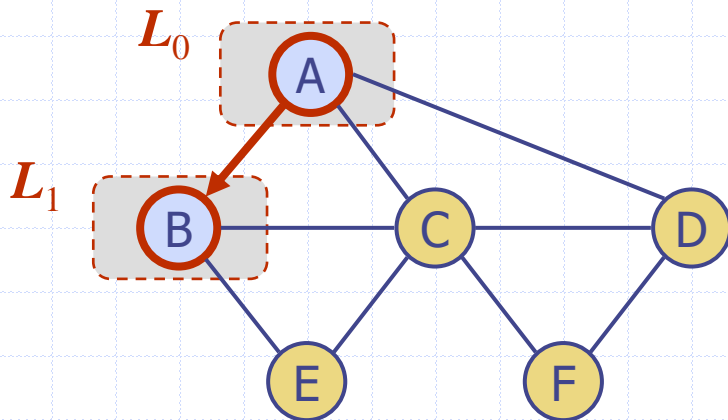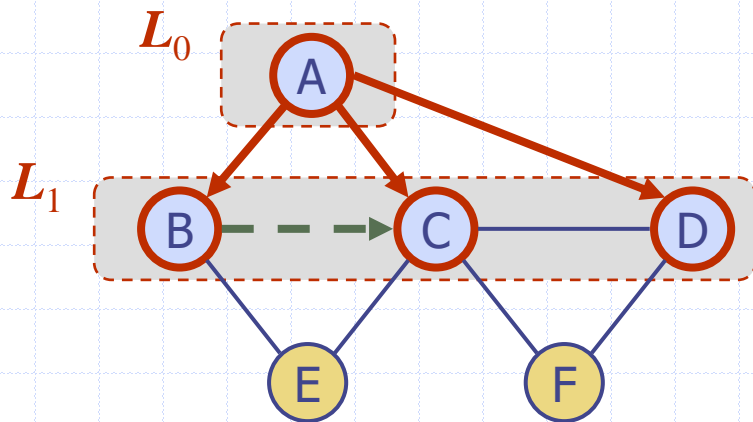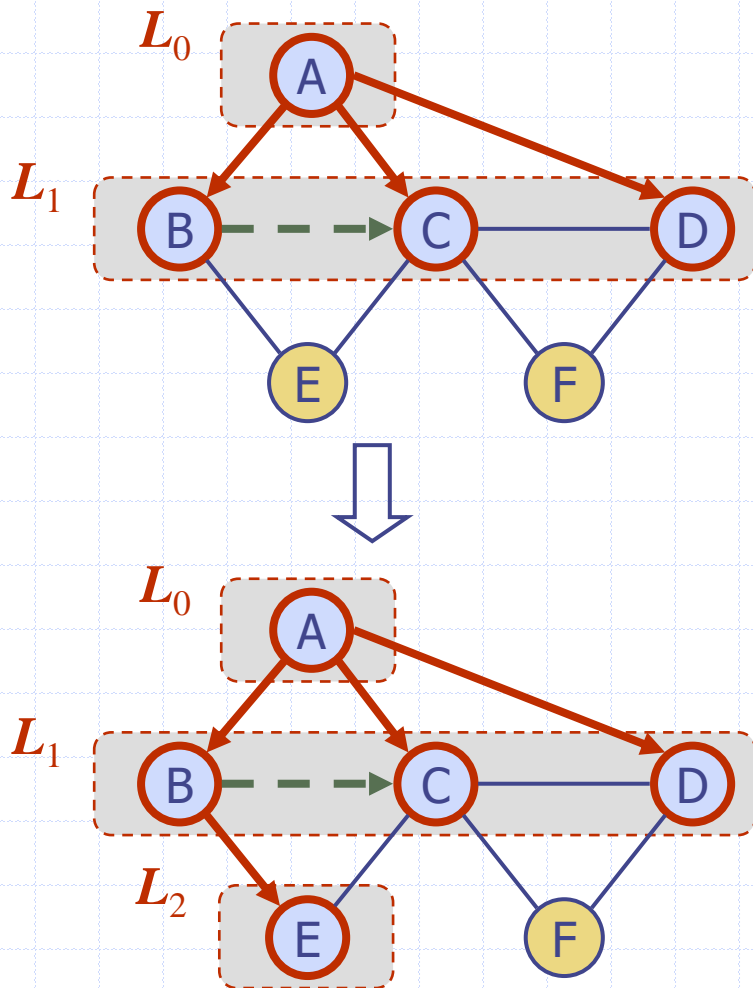
Example (cont.)
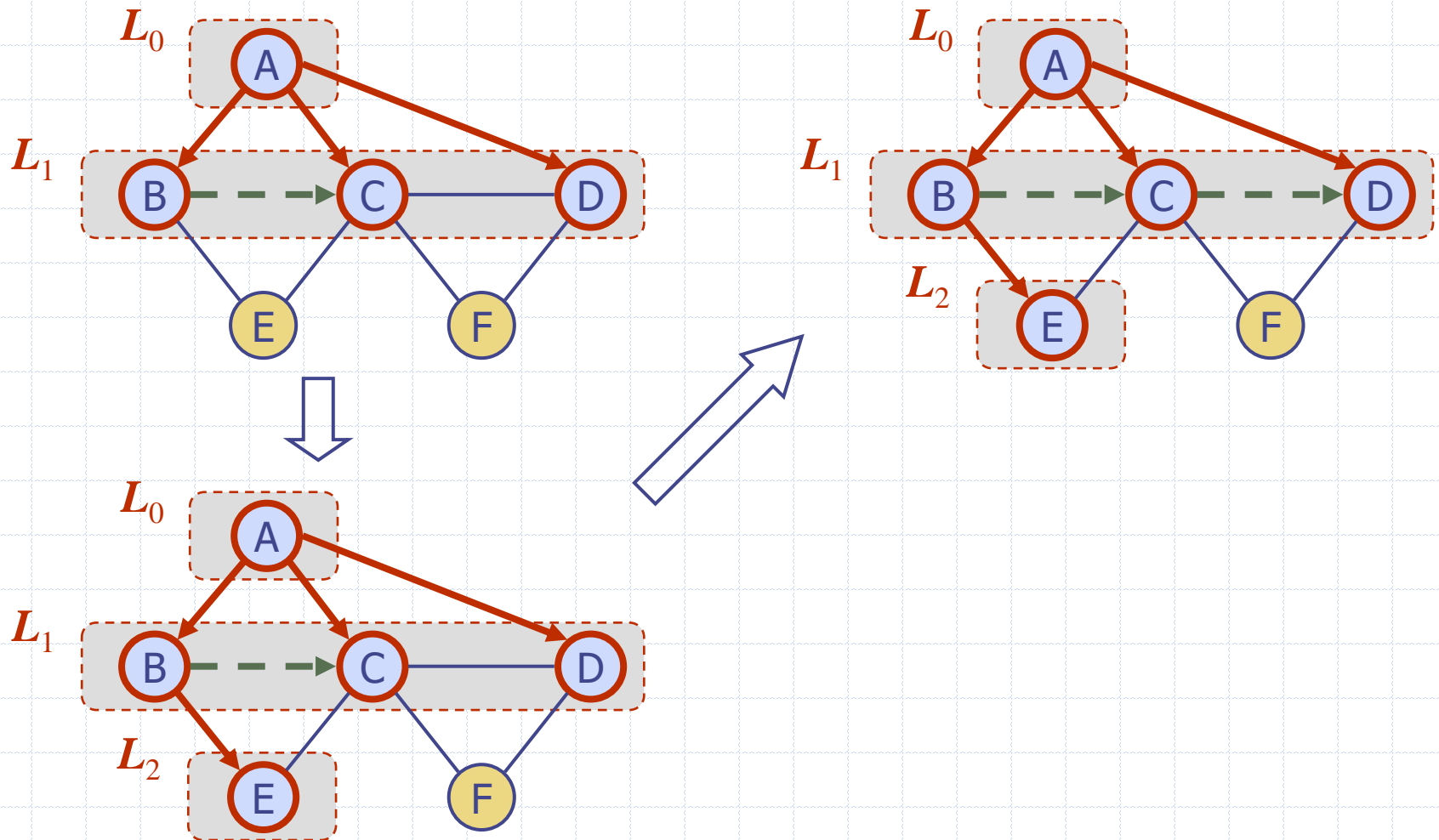
Example (cont.)
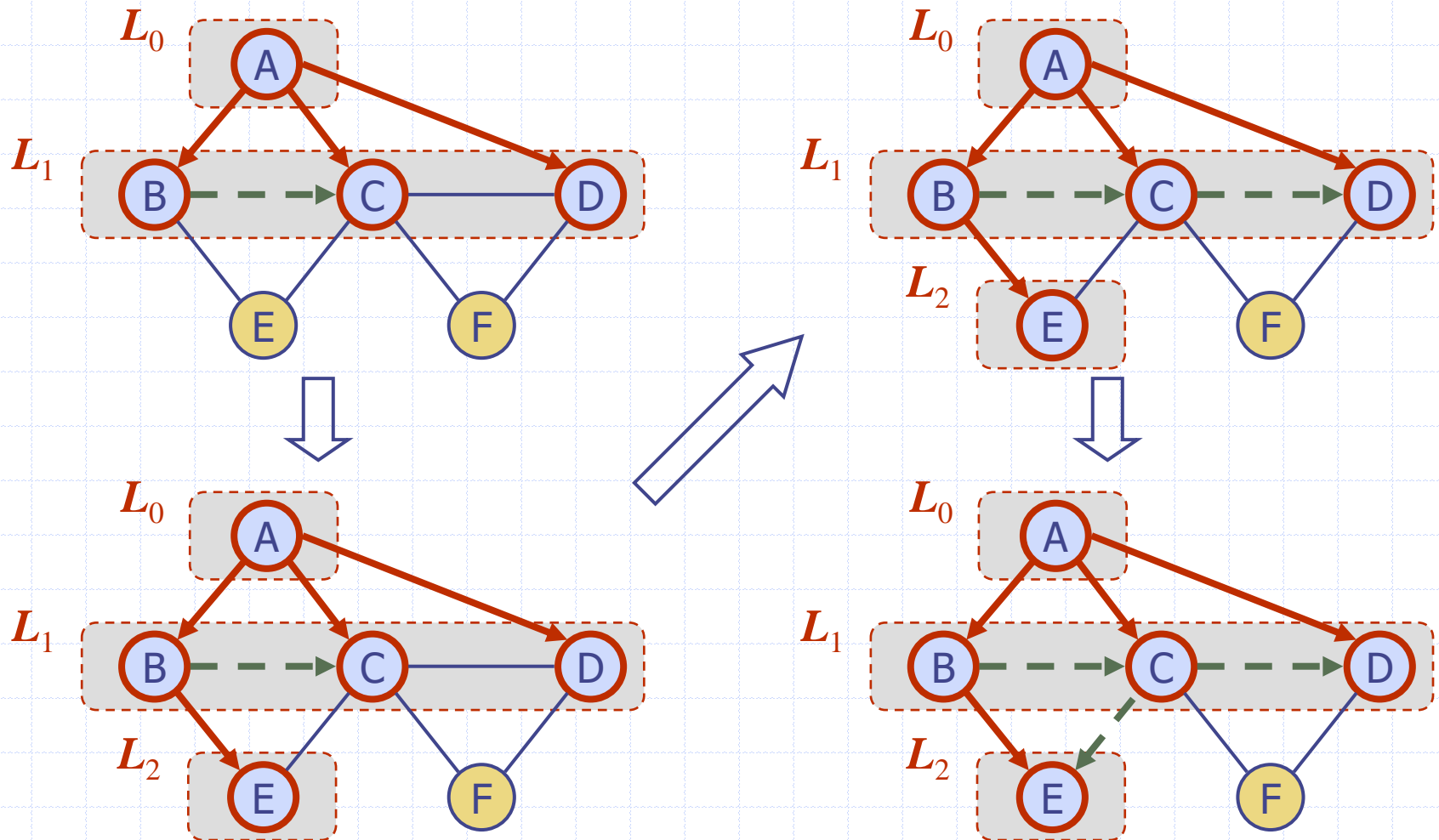
Example (cont.)

Example (cont.)
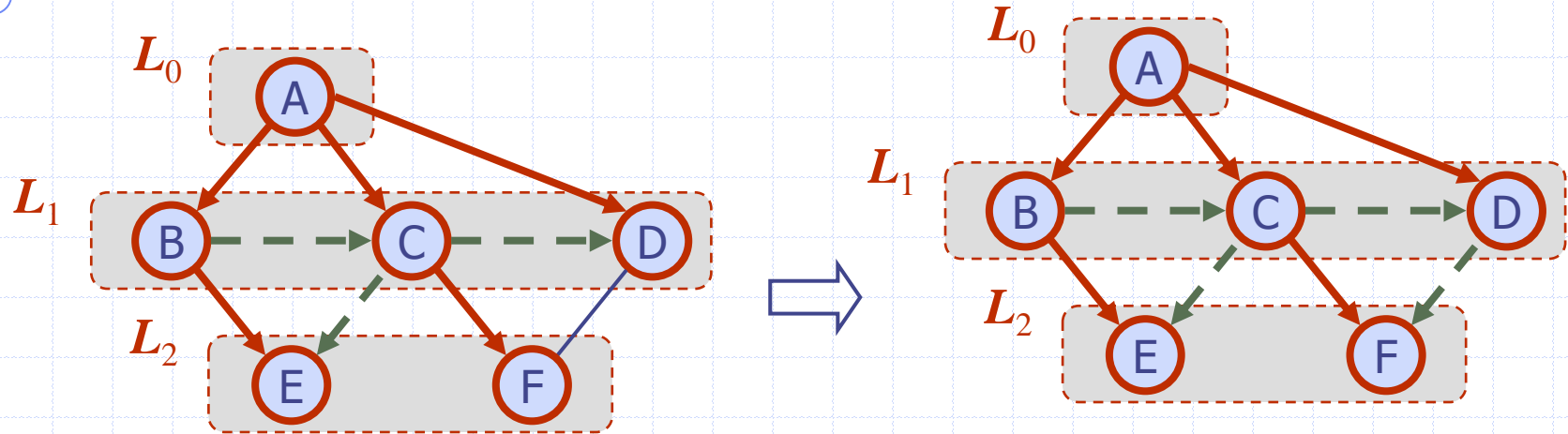
BFS Levels

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

BFS
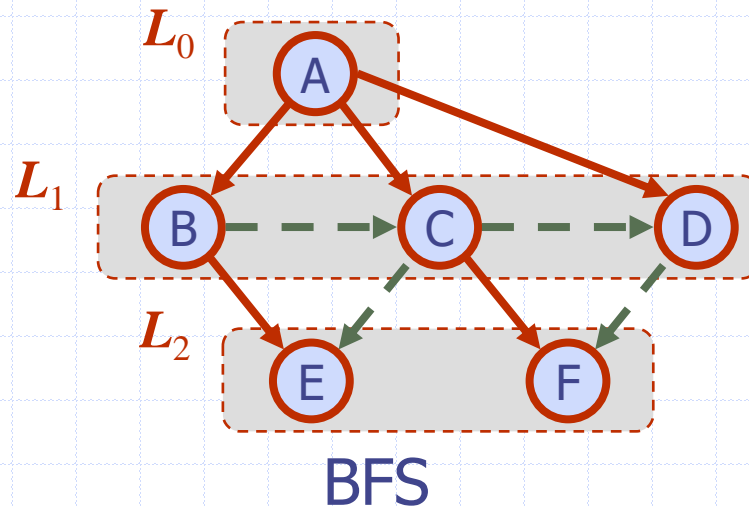
BFS Algorithm

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

Algorithm *BFS*(*G*)
   Input graph *G*
   Output labeling of the edges
      and partition of the
      vertices of *G*
  for all *u* ∈ *G.vertices*() **do**
   setLabel(*u, UNEXPLORED*)
  for all *e* ∈ *G.edges*() **do**
   setLabel(*e, UNEXPLORED*)
  for all *v* ∈ *G.vertices*() **do**
   if *getLabel*(*v*) = *UNEXPLORED*
    BFScomponent(*G, v*)\

Algorithm *BFScomponent*(*G, s*)
  Q ← new empty Queue
  Q.enqueue(*s*)
  setLabel(*s, VISITED*)
  while !*Q.isEmpty*() **do**
    v ← *Q.dequeue* ()
    for all *e* ∈ *G.incidentEdges*(*v*) **do**
     if *getLabel*(*e*) = *UNEXPLORED* **then**
      w ← *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

L_0
L_1
L_2
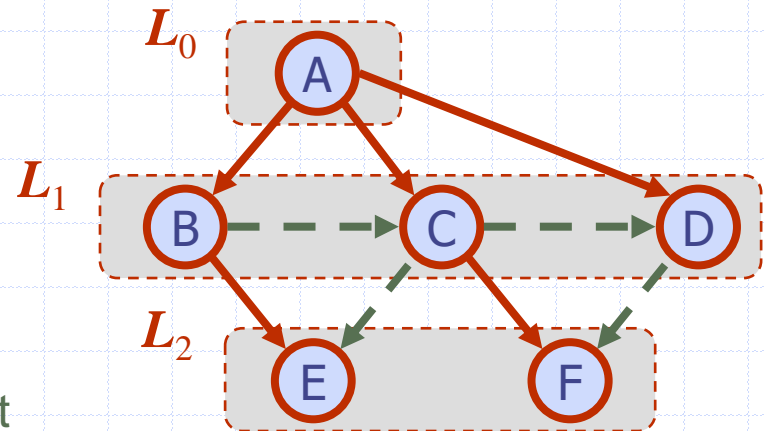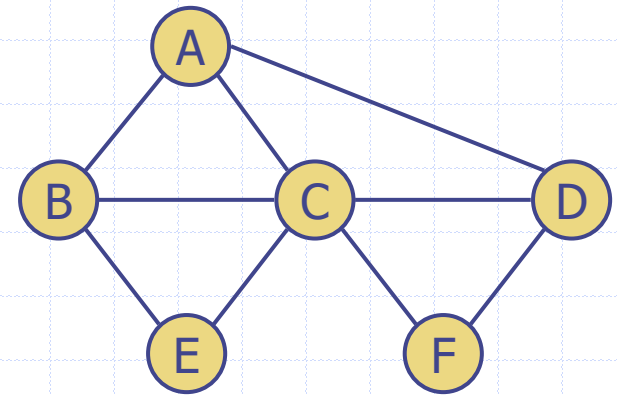
Analysis

- 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

L_0 L_1 L_2
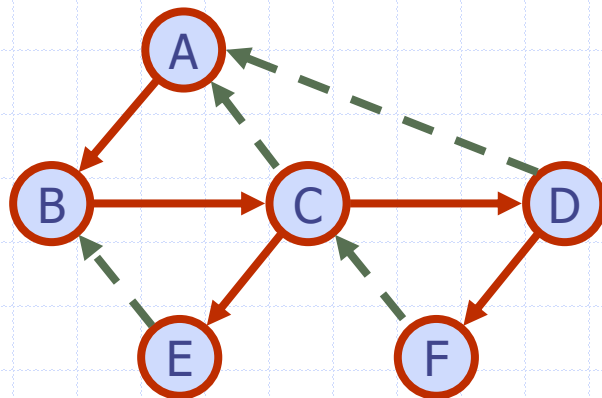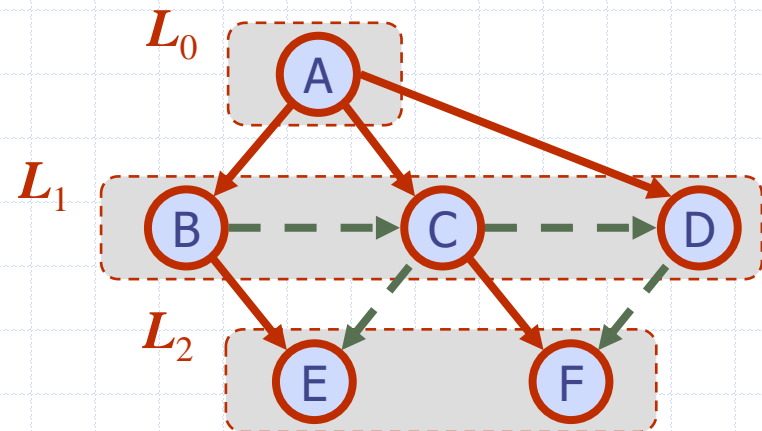
BFS

DFS vs. BFS (cont.)

Back edge (*v,w*)

- *w* is an ancestor of *v* in the tree of discovery edges
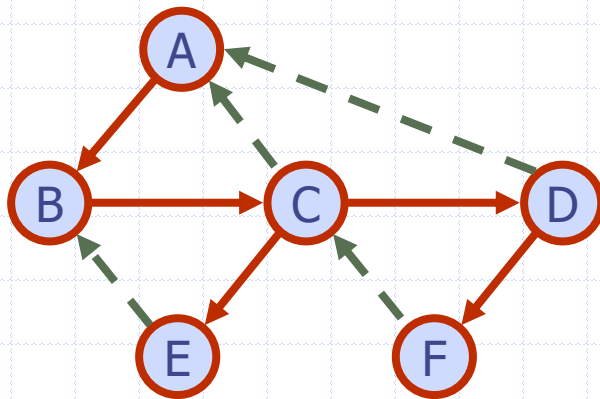
Cross edge (*v,w*)

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

DFS
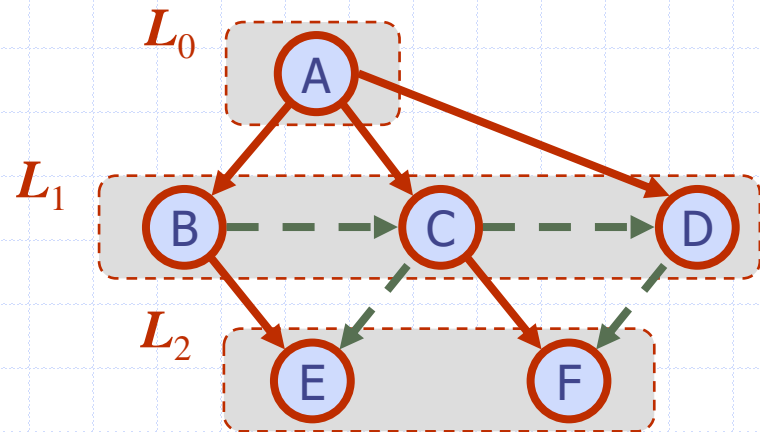
BFS

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

Example of the Template Method Pattern in JavaScript

- Generic algorithm that can be specialized by redefining certain steps
- Implemented by means of an abstract JavaScript class
- Visit methods that can be redefined by subclasses
- Template method eulerTour
  - Recursively called on the left and right children
  - A result array result with elements result[0], result[1], and result[2] keeps track of the output of the recursive calls to eulerTour

```javascript
class EulerTour {
    visitExternal(T, p, result) { }
    visitPreOrder(T, p, result) { }
    visitInOrder(T, p, result) { }
    visitPostOrder(T, p, result) { }
    eulerTour(T, p) {
        let result = new Array(3);
        if (T.isExternal(p)) {
                this.visitExternal(T, p, result);
        } else {
                this.visitPreOrder(T, p, result);
                result[0] = this.eulerTour(this._tree.leftChild(p));
                this.visitInOrder(T, p, result);
                result[2] = eulerTour(this._tree.rightChild(p));
                this.visitPostOrder(T, p, result);
        }
        return result[1];
    }
}
```

Specialization (Subclass) of EulerTour

◆ We show how to specialize class EulerTour to evaluate an arithmetic expression

◆ Assumptions

- External nodes store Integer objects
- Internal nodes store Operator objects supporting method
  operation (Integer, Integer)

```
class CalculateHeight extends EulerTour {

    visitExternal(T, p, result) {
        result[1] = 0;
    }

    visitPostOrder(T, p, result) {
        result[1] = 1 + Math.max(result[0], result[2]);
    }

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

}
```

Euler Tour Template (pseudo-code)

Algorithm *EulerTour*(T, *v*)

    result ← **new Array(3)**

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

        visitExternal(T, v, result)

    else

        visitPreOrder(T, v, result)

        result[0] ← *EulerTour*(T, T.*leftChild*(*v*))

        visitInOrder(v, result)

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

        visitPostOrder(T, v, result)

    return result[1]

Height using the Template (pseudo-code)

Algorithm *visitExternal*(T, *v, result*)

    result[1] ← **0**

Algorithm *visitPostOrder*(T, *v, result*)

    result[1] ← **1 + MAX**(result[0], result[2])

Algorithm *height*(T)

    return *EulerTour*(T, T.root())

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*() **do**
    setLabel(*u, UNEXPLORED*)
    postInitVertex(*u*)
  for all *e* ∈ *G.edges*() **do**
    setLabel(*e, UNEXPLORED*)
    postInitEdge(*e*)
  for all *v* ∈ *G.vertices*() **do**
    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*) **do**
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        preDiscoveryVisit(*G, v, e, w*)
        DFS(*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 *DFS*(*G, v*)
   setLabel(*v, VISITED*)
   startVertexVisit(*G, v*)
   for all *e* ∈ *G.incidentEdges*(*v*) **do**
      if *getLabel*(*e*) = *UNEXPLORED* **then**
        w ← *opposite*(*v,e*)
        if *getLabel*(*w*) = *UNEXPLORED*
        then
           setLabel(*e, DISCOVERY*)
          preDiscoveryVisit(*G, v, e, w*)
          DFS(*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 ← *S.elements*()
   for all *e* ∈ *G.incidentEdges*(*v*) **do**
      if *getLabel*(*e*) = *UNEXPLORED* **then**
        w ← *opposite*(*v, e*)
        if *getLabel*(*w*) = *UNEXPLORED* **then**
          setLabel(*e, DISCOVERY*)
          S.push(*e*)
          pathDFS(*G, w, z*)
          S.pop()   { *e* gets popped }
        else
          setLabel(*e, BACK*)

   S.pop()   { *v* gets popped }

Overriding hook methods in a subclass FindSimplePath

Algorithm *findPath*(*G, u, v*)
    S ← new empty stack   {S is a subclass field}
    z ← v   {z is a subclass field & is the target vertex}
    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*() **do**
    setLabel(*u, UNEXPLORED*)
    postInitVertex(*u*)
  for all *e* ∈ *G.edges*() **do**
    setLabel(*e, UNEXPLORED*)
    postInitEdge(*e*)
  for all *v* ∈ *G.vertices*() **do**
    if *isNextComponent*(*G, v*)
      preComponentVisit(*G, v*)
      DFS(*G, v*)
      postComponentVisit(*G, v*)

  return *result*(**G**)

Algorithm *isNextComponent*(*G, v*)
  return *getLabel*(*v*) = *UNEXPLORED*

Algorithm *DFS*(*G, v*)
  setLabel(*v, VISITED*)
  startVertexVisit(G, *v*)
  for all *e* ∈ *G.incidentEdges*(*v*) **do**
    if *getLabel*(*e*) = *UNEXPLORED*
      w ← *opposite*(*v,e*)
      if *getLabel*(*w*) = *UNEXPLORED*
        setLabel(*e, DISCOVERY*)
        preDiscoveryVisit(*G, v, e, w*)
        DFS(*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 *findPath*(*G, u, v*)
    S ← new empty stack   {S is a subclass field}
    start ← u   {start is a subclass field & is the starting vertex}
    dest ← v   {dest is a subclass field & is the destination vertex}
    path ← ∅   {path is a subclass field & is the path from u to v}
    return DFS(*G*)

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

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

Algorithm *startVertexVisit*(*G, v*)
    S.push(v)
    if v=dest then   {dest is a subclass field & is the destination vertex}
        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}

Exercise: Cycle Finding
Override hook operations

Algorithm *DFScomponent*(*G, v*)

    setLabel(*v, VISITED*)

    startVertexVisit(G, *v*)

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

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

          w ← *opposite*(*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*)

Algorithm *cycleDFS*(*G, v*)

    setLabel(*v, VISITED*)

    if *cycle* != ∅ **then return**

    S.push(*v*)

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

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

          w ← *opposite*(*v,e*)

          S.push(*e*)

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

            setLabel(*e, DISCOVERY*)

            cycleDFS(*G, w*)

            S.pop()

        else

          setLabel(*e, BACK*)

          cycle ← new empty List

          o ← *w*

          do

            cycle.insertLast(*o*)

            o ← *S.pop*()

          while *o* != *w*

    S.pop()

Exercise: Cycle Finding Override hook operations (v2)

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

Overriding template methods in subclass FindCycles

Algorithm *startVertexVisit*(*G, v*)
   if ! cycleFound   then   S.push(v)

Algorithm *finishVertexVisit*(*G, v*)
   if ! cycleFound   then   S.pop()

Algorithm *preEdgeVisit*(*G, v, e*)
   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
       cycle ← new empty sequence
       o ← *w*
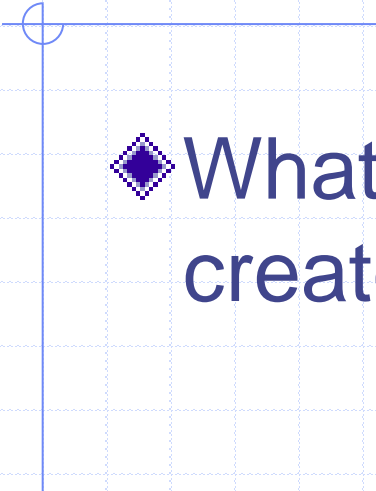   **do**
         cycle.insertLast(*o*)
         o ← *S.pop*()
       while *o* ≠ *w*
       cycleFound ← true   {cycleFound is a subclass field, initially false}

◆ What additional method(s) do we need create or need to override?

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

  return *result*(*G*)

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

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.