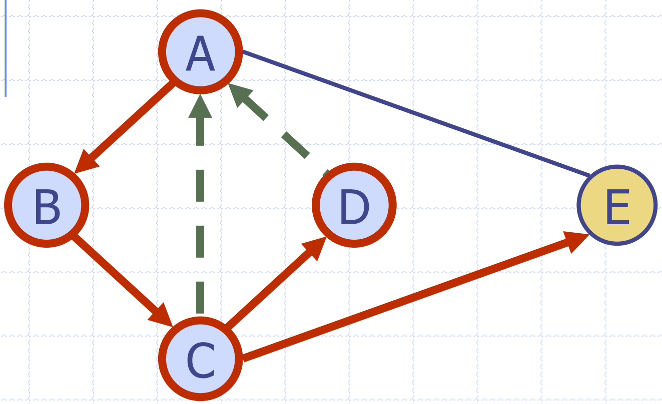
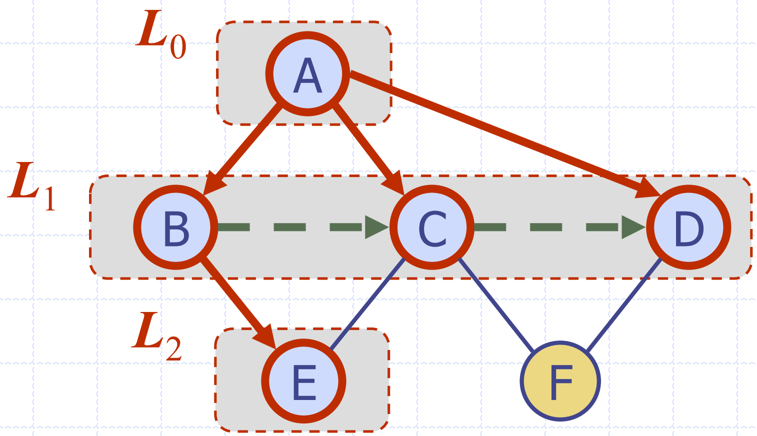
Graph Traversals





Graph Traversal

- Visiting each vertex of a graph (tree traversal is a subset)
- Alternative definition:
  - Starting from a vertex v and visiting all vertices reachable from v
- The order of visit results in different traversal
- Must make sure a vertex is visited only **once**!
  - Otherwise cycles => infinite loops!
- Why do we want to traverse graphs?
  - Finding connected components
  - Finding shortest paths, optimization
  - Network analysis
  - Graphics, programming languages, compilers
  - …

Graph Traversal – Common Applications

- Given a graph G:
  - Visiting all vertices (and also edges) of G
  - Determine if G is connected
  - Compute the connected components of G
  - Compute the spanning tree/forest of G
  - Find a path between 2 vertices in G
  - Find the "shortest path" between 2 vertices in G
  - Find a simple cycle in G (if one exists)
- 2 traversal algorithms (for now):
  - Depth-First Traversal (or Depth-First Search)
  - Breadth-First Traversal (or Breadth-First Search)

Depth-First Traversal

- Depth-first traversal is to graphs what Euler tour is to binary trees
- Given a vertex **v**, proceed along the graph as deeply as possible before backing up
- After visiting **v**, visit an unvisited adjacent vertex to **v**
- The order in which the adjacent vertices should be visited is not completely specified
- For our purposes, let's call depth first search (DFS) the depth first traversal starting from a given vertex **v**, thus DFS only visits the reachable vertices from **v**

Depth-First Search (DFS)

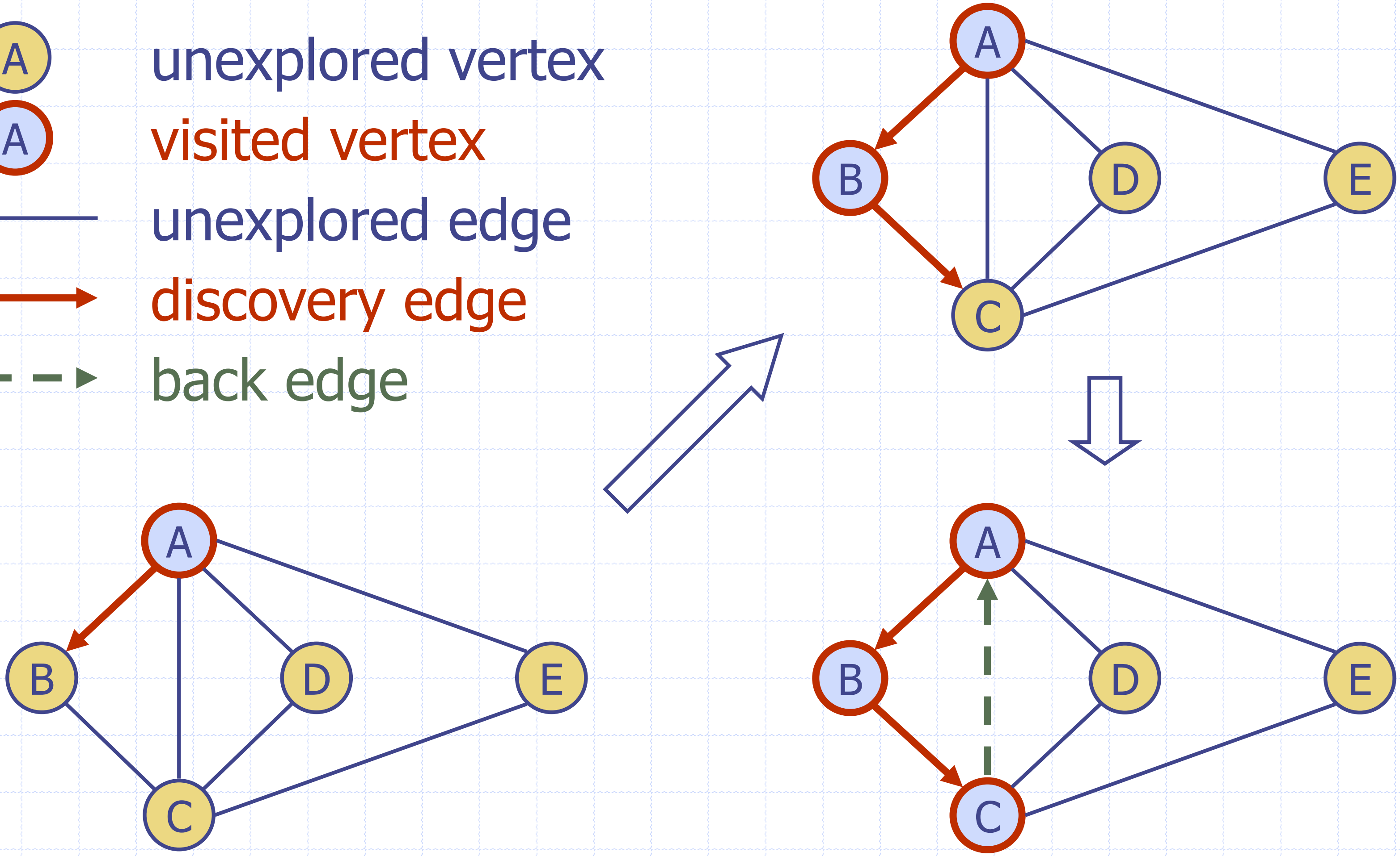
```
Algorithm DFS(G, v)
   visit(v) //visits and marks v as visited
   for u in adjacent(v)
      if not isVisited(u)
         DFS(G, u)

Algorithm DFS(G, v, tree) //tree or forest
   visit(v) //visits and marks v as visited
   for e in edges(v)
      u ← opposite(e, v)
      if not isVisited(u) //else back-edge
         addEdge(tree, v, u, e)
         DFS(G, u, tree)
```

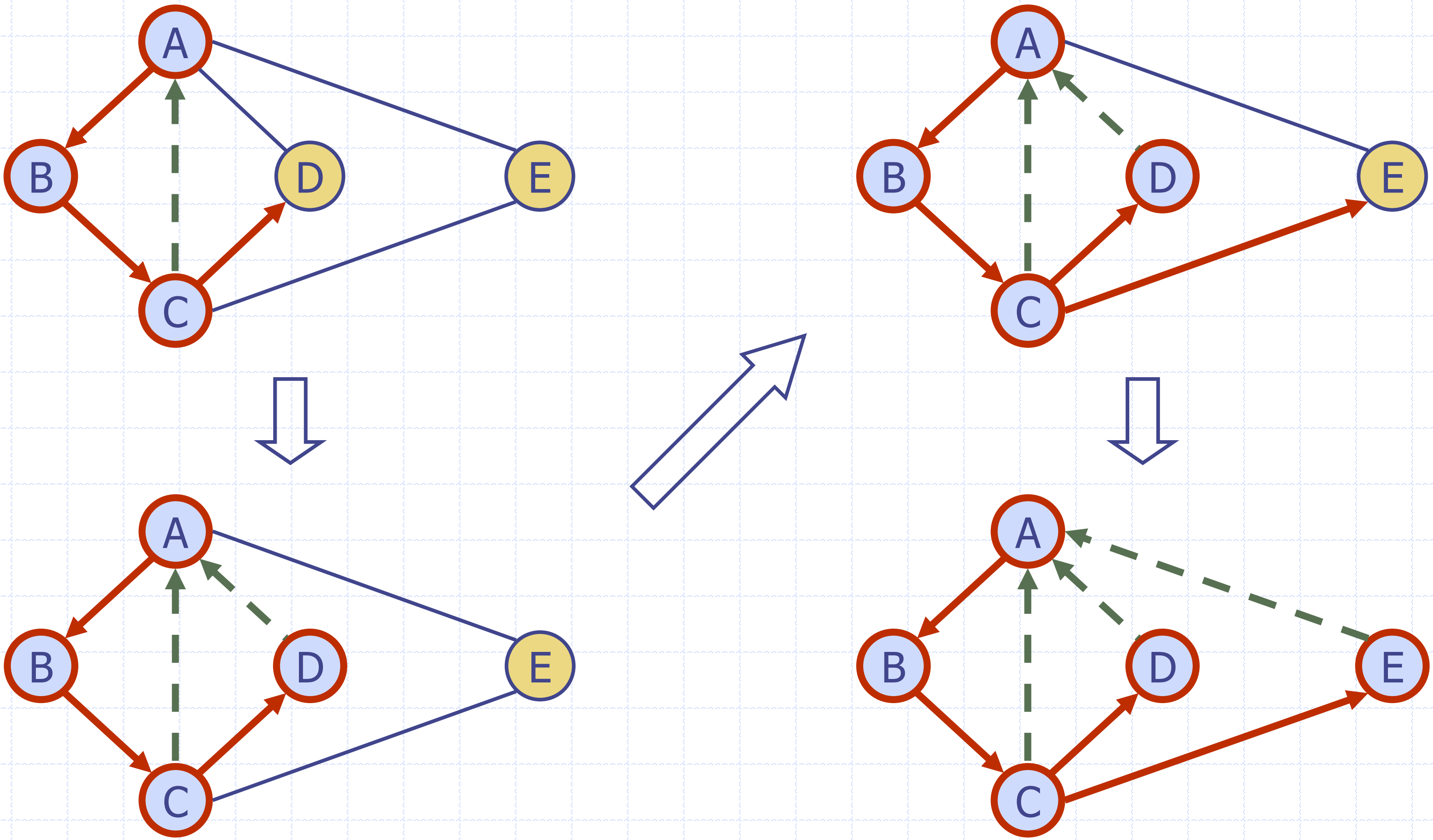
For certain algorithms, you might want to do more book-keeping

Example

- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- back edge



Example (cont.)



DFT Analysis

```
Algorithm DFS(G, v)
   visit(v) //visits and marks v as visited
   for u in adjacent(v)
      if not isVisited(u)
         DFS(G, u)
```

- Assume visit(v) is O(1)
- Each vertex is visited once, O(|V|)
- Adjacent vertices/outgoingEdges is called once per vertex, for all the edges total O(|E|)
- Thus, complexity of DFT is O(|V|+|E|)
- Note this is assuming adjacency list implementation!

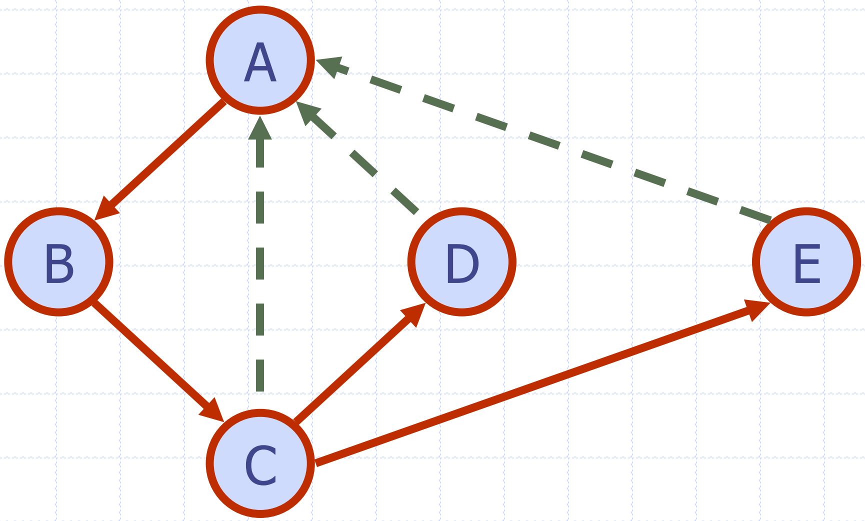
DFS Properties

Property 1

DFS(*G*, *v*) visits all the vertices and edges in the connected component of *v*

Property 2

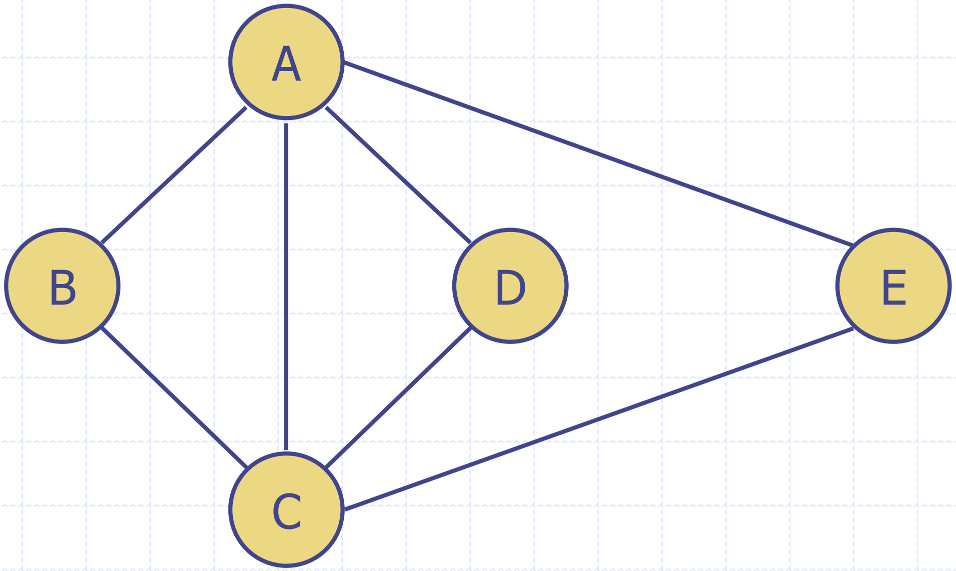
The tree built by *DFS*(*G*, *v*, *tree*) forms a spanning tree of the connected component of *v*



DFS – Iterative Version

```
Algorithm iterativeDFS(G, v)
    s ← stack()
    s.push(v)
    while not s.isEmpty()
        u ← s.pop()
        if not isVisited(u)
            visit(u) //visits & marks u as visited
        for w in adjacent(u)
            if not isVisited(w)
                s.push(w)
```

Visit Order for DFS: Recursive vs Iterative



Compare the visit order on the board.
Assuming "getNeighbors" returns the vertices in alphabetical order

DFS – Basic Applications

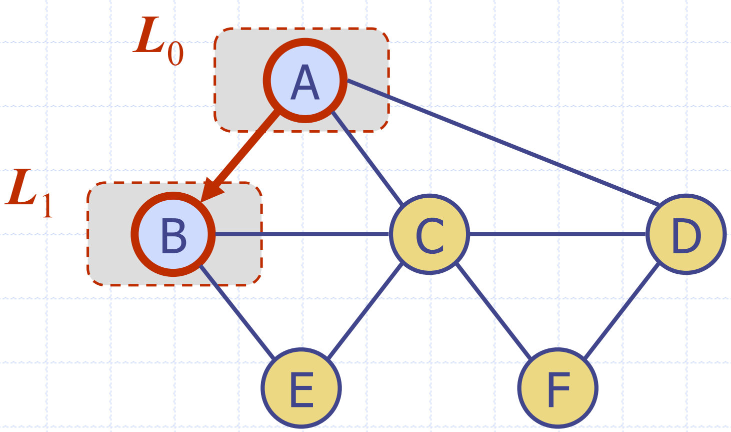
- DFS to DFT?
  - Loop over all the vertices and call DFS on unvisited ones!
- In a graph **G**, find a path between vertices **v** and **u**
  - Start from **v**, and if **u** is found, the stack represents the path
- In a graph **G**, find a simple cycle if one exists
  - Undirected graph: Visiting an already visited vertex
  - Directed graph: Visiting a vertex that is currently in the stack, you can keep a separate container for efficiency
- Find all connected components
  - Do DFT. Mark all the vertices with the same label for each DFS run if unvisited

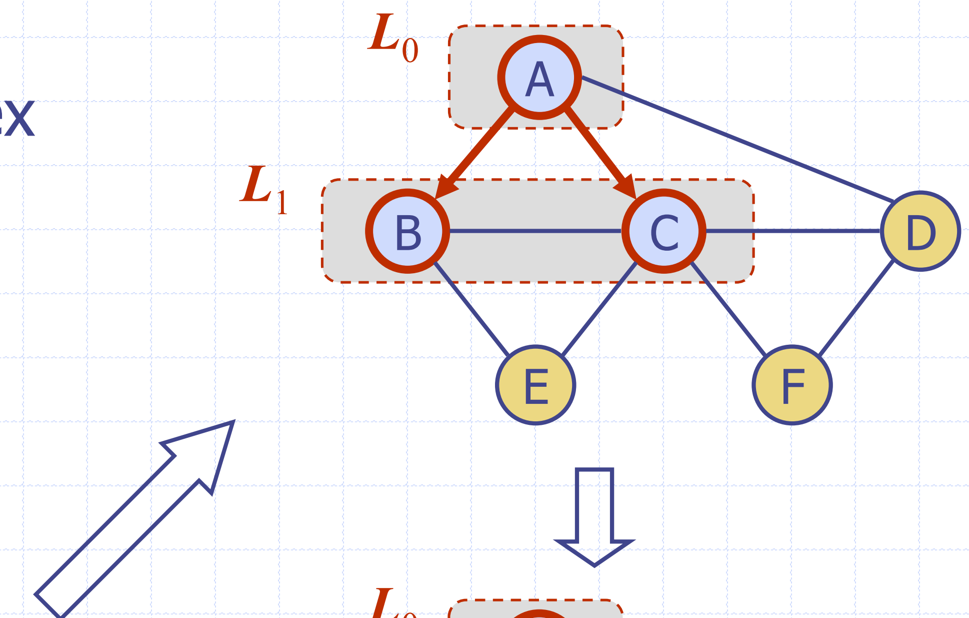
Breadth-First Traversal

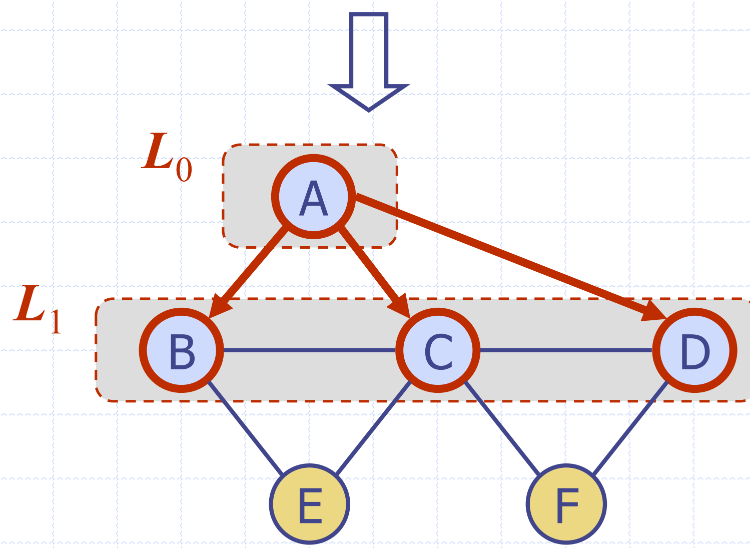
- Given a vertex **v**, proceed as "horizontally" as possible before going deep
- After visiting **v**, visit all the unvisited adjacent vertices of **v**, before visiting their adjacent vertices
- The order in which the adjacent vertices should be visited is not completely specified

Example

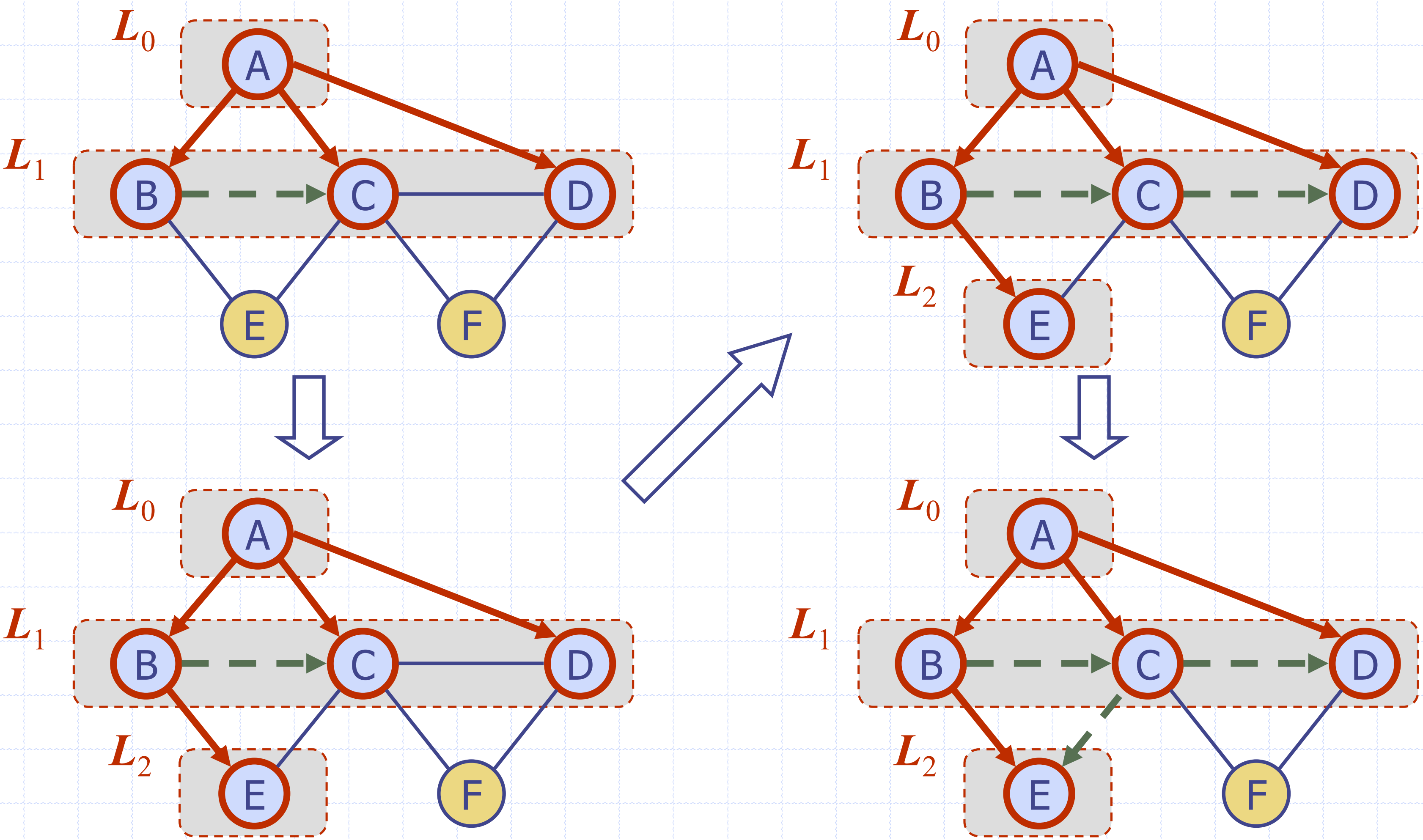
- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- cross edge



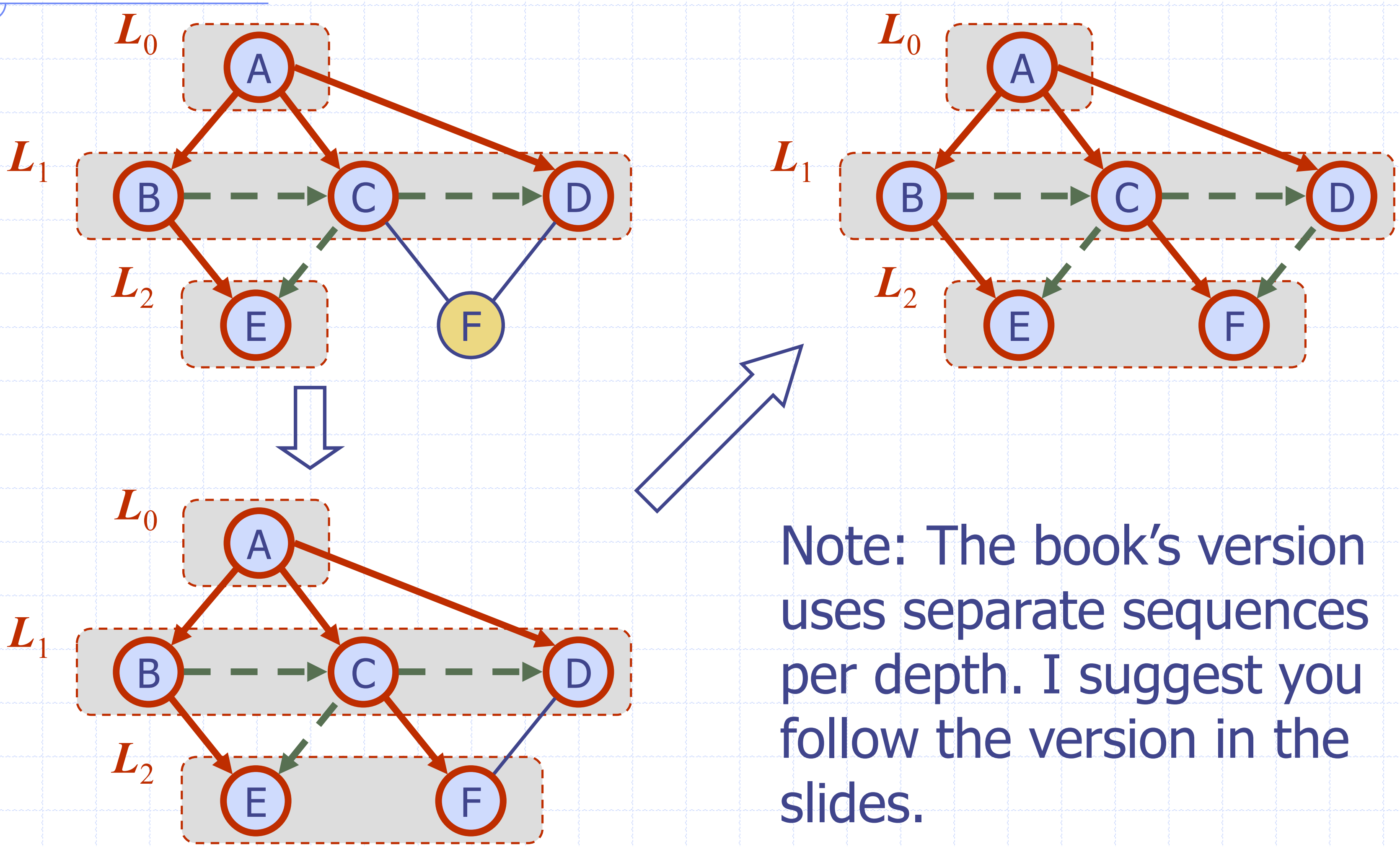




Example (cont.)



Example (cont.)



Note: The book's version uses separate sequences per depth. I suggest you follow the version in the slides.

BFS

```
Algorithm BFS(G, v)
   q ← queue()
   q.push(v)
   while not q.isEmpty()
      u ← q.pop()
      if not isVisited(u)
         visit(u) //visits & marks u as visited
      for w in adjacent(u)
         if not isVisited(w)
            q.push(w)
```

BFS – Edge Version

```
Algorithm BFS(G, v, tree)
  q ← queue()
  q.push(v)
  while not q.isEmpty()
    u ← q.pop()
    if not isVisited(u)
      visit(u) //visits & marks u as visited
    for e in edges(u)
      w ← opposite(e, u)
      addEdge(tree, w, u, e)
      if not isVisited(w) //else cross-edge
        q.push(w)
```

With DFS, finding the back-edges or with BFS finding the cross-edges can be useful for certain applications

BFT Analysis

- Assume visit(v) is O(1)
- Each vertex is visited once, O(|V|)
- Each vertex is inserted once into the queue
- Adjacent vertices/outgoingEdges is called once per vertex, for all the edges total O(|E|)
- Thus, complexity of DFT is O(|V|+|E|)
- Note this is assuming adjacency list implementation!

Properties

Notation

G_s: connected component of s

Property 1

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

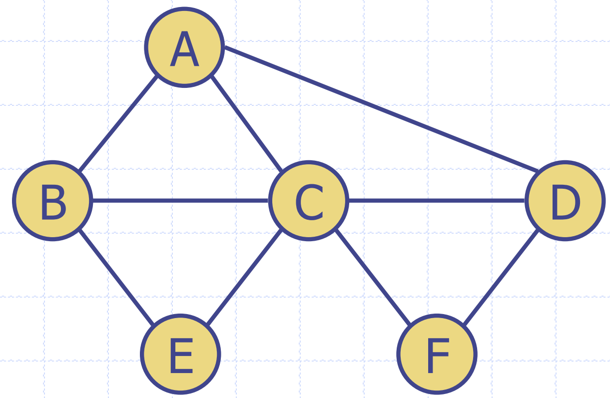
Property 2

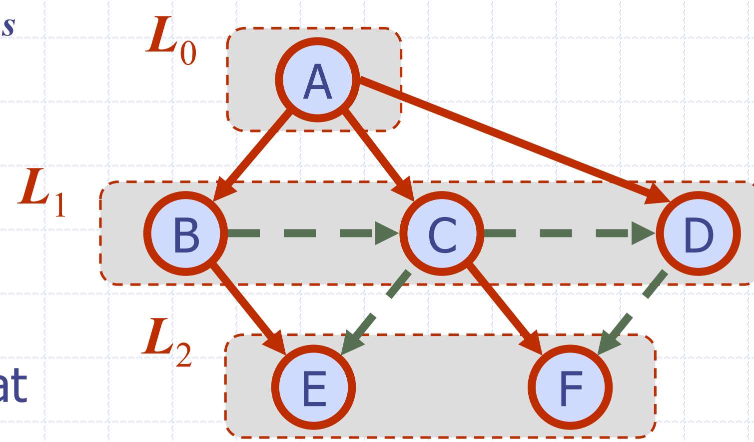
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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
- **Shortest Path!**





BFS – Basic Applications

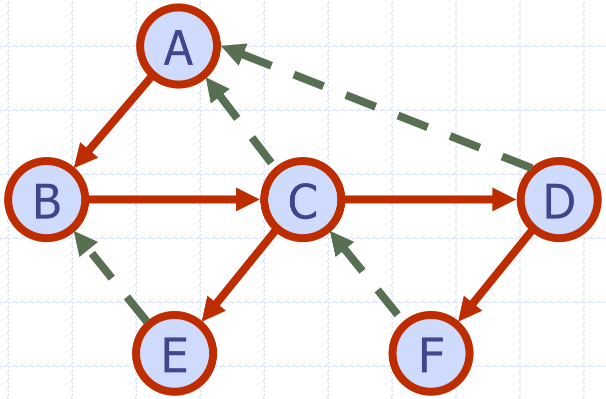
- BFS to BFT?
  - Loop over all the vertices and call DFS on unvisited ones!
- In a graph **G**, find a path between vertices **v** and **u**. This path is the shortest path if graph is unweighted!
  - Start from **v** and keep a list of "parents" as you visit the nodes. If you reach **u**, then use the list of parents from this vertex to get the path (which will be in reverse order)
- Find all connected components
  - Do BFT. Mark all the vertices with the same label for each BFS run if unvisited
- You can use BFS to find cycles but DFS is preferred.

Other Basic Applications

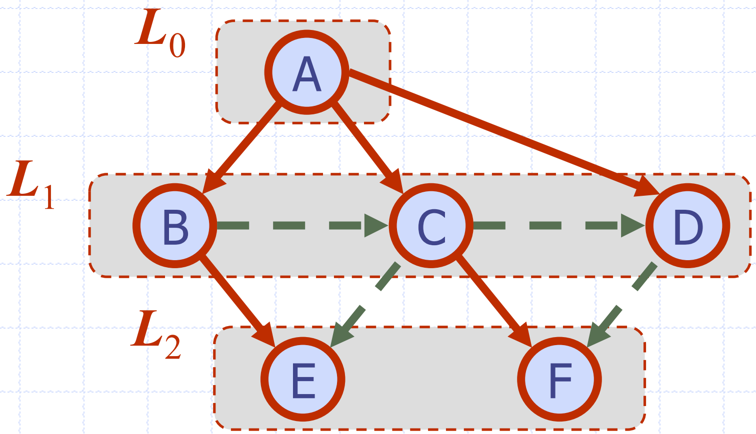
- Both DFS a can be used to find connected components, paths and cycles
- They can be used to find *spanning trees* and *forests*
  - Just use the edge version of the algorithms and build the tree
  - To get a spanning forest, use the traversals
- They can be used to find the *reachable set* of a vertex (all the reachable vertices)
  - Start either BFS or DFS, label or keep a list of all the vertices that you visit
  - For undirected graphs, if v is in the reachable set of u then u is in the reachable set of v

DFS vs. BFS

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



DFS



BFS

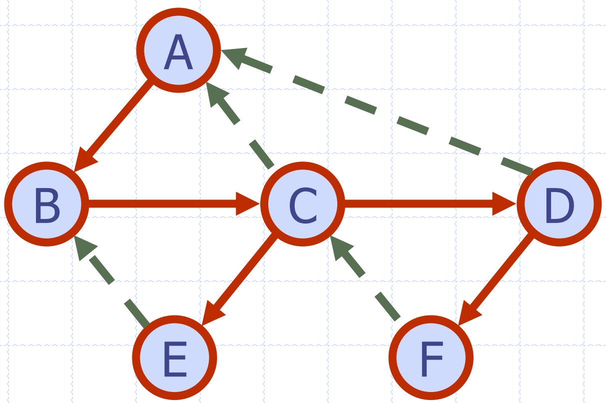
DFS vs. BFS (cont.)

Back edge (v,w)

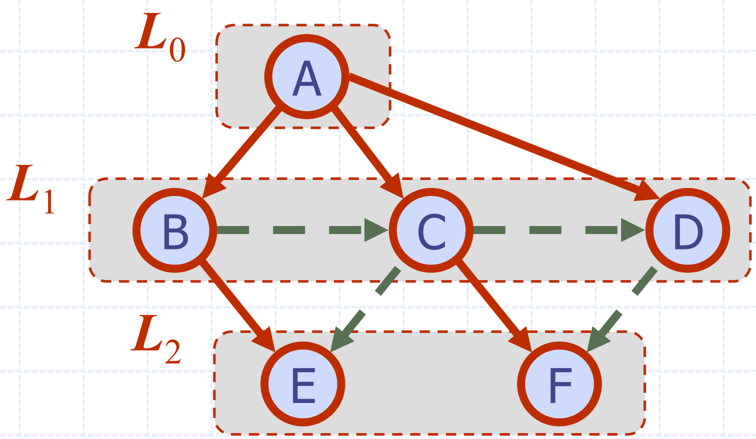
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

- w is in the same level as v or in the next level



DFS



BFS

General Graph Search

- Notice any similarities between iterative DFS and BFS?

1. Initialize: Add the initial vertex in a container
2. Chose the next vertex to be visited
3. Visit it if it is not already visited
4. Add its neighbors to the container if not visited
5. Go to step 2 until the container is empty

- The way that the next vertex to be visited is chosen results in different algorithms
  - Container = stack then DFS
  - Container = queue then BFS
  - Container = priority queue then ???

General Graph Search

```
Algorithm generalGraphSearch(G, v)
    frontier ← container()
    frontier.push(v)
    while not q.isEmpty()
        u ← frontier.pop()
        if not isVisited(u)
            visit(u)
        for w in adjacent(u)
            if not isVisited(w)
                frontier.push(w)
```