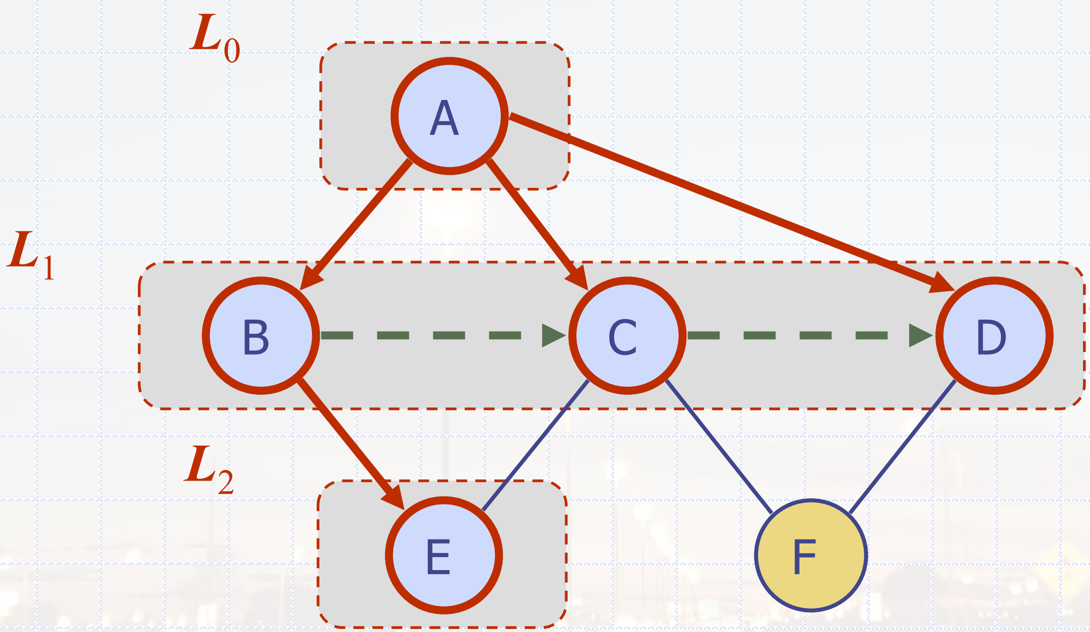
Breadth-First Search (BFS)

Dr. Aiman Hanna
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides have been extracted, modified and updated from original slides of :
Data Structures and Algorithms in Java, 5th edition. John Wiley& Sons, 2010. ISBN 978-0-470-38326-1.
Data Structures and the Java Collections Framework by William J. Collins, 3rdedition, ISBN 978-0-470-48267-4.
Both books are published by Wiley.

Coverage

- Graph Traversal
  - Breath-First Search (BFS)



Breadth-First Search (BFS)

- BFS is less adventurous than DFS, in the sense that it does not travel far, at any point of time, from where it starts

- The idea is similar to travelling between cities. First go and visit all the cities that are closest to you. If this goes fine, think about travelling to cities that are further. If this goes fine, go and travel to the ones that are even further! By the time you are adventurous, there are no more cities to visit!

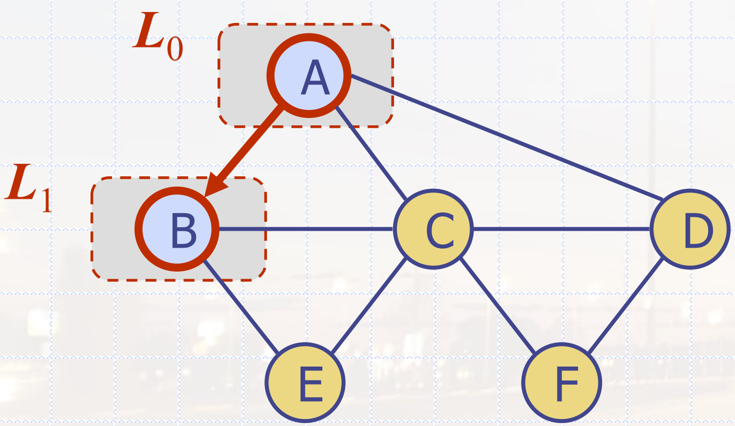
- Consequently, BFS subdivides the vertices into ***levels*** and goes "as if" in rounds to visit the vertices (*Still, equipment needed are a rolled robe, and can of paint spray*).

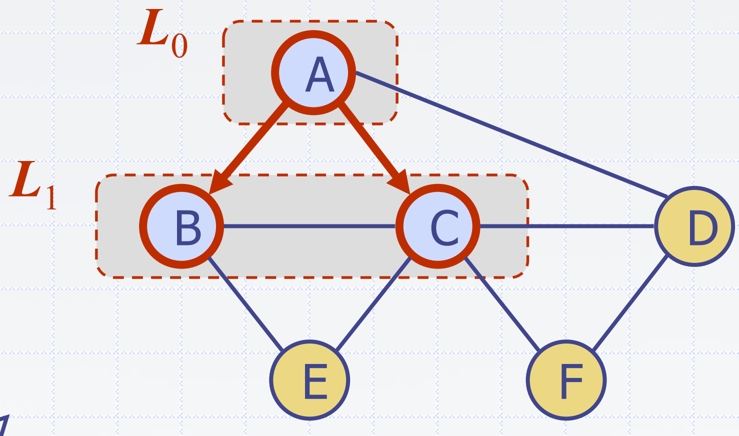
BFS

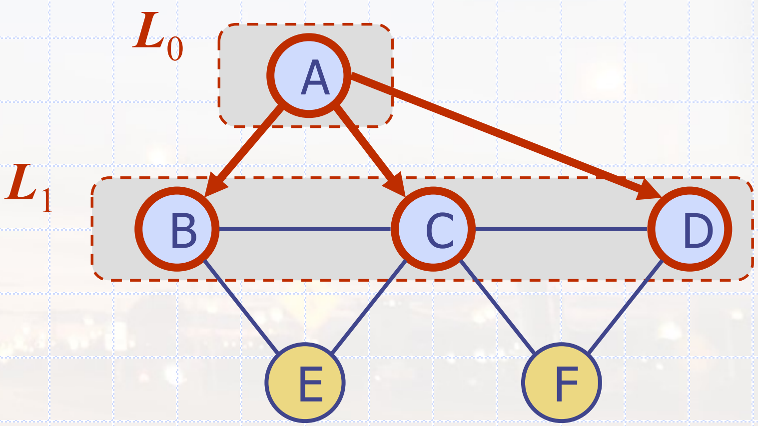
- BFS performs as follows:
  - Start at some vertex s, which is considered as ***level 0***. This vertex is also referred to as the "**anchor**" (All following explorations will be based on a distance from that anchor)
  - Unroll the robe with length equivalent to the length of one edge. Go to all the vertices that you can reach with that length. These vertices belong to ***level 1***. Paint the vertices that you visit (to mark them as *VISTED*)
  - Once all the vertices in level 1 are visited, unroll the robe further to twice the length of an edge. Similarly go and visit all the vertices that you can reach with that length. These vertices form ***level 2***. If the exploration through an edge leads to an already visited vertex, mark that edge as "**cross**"; otherwise mark it as "**discovery**"
  - Repeat the above operation, visiting further levels, until no more vertices are there to visit

BFS Example1

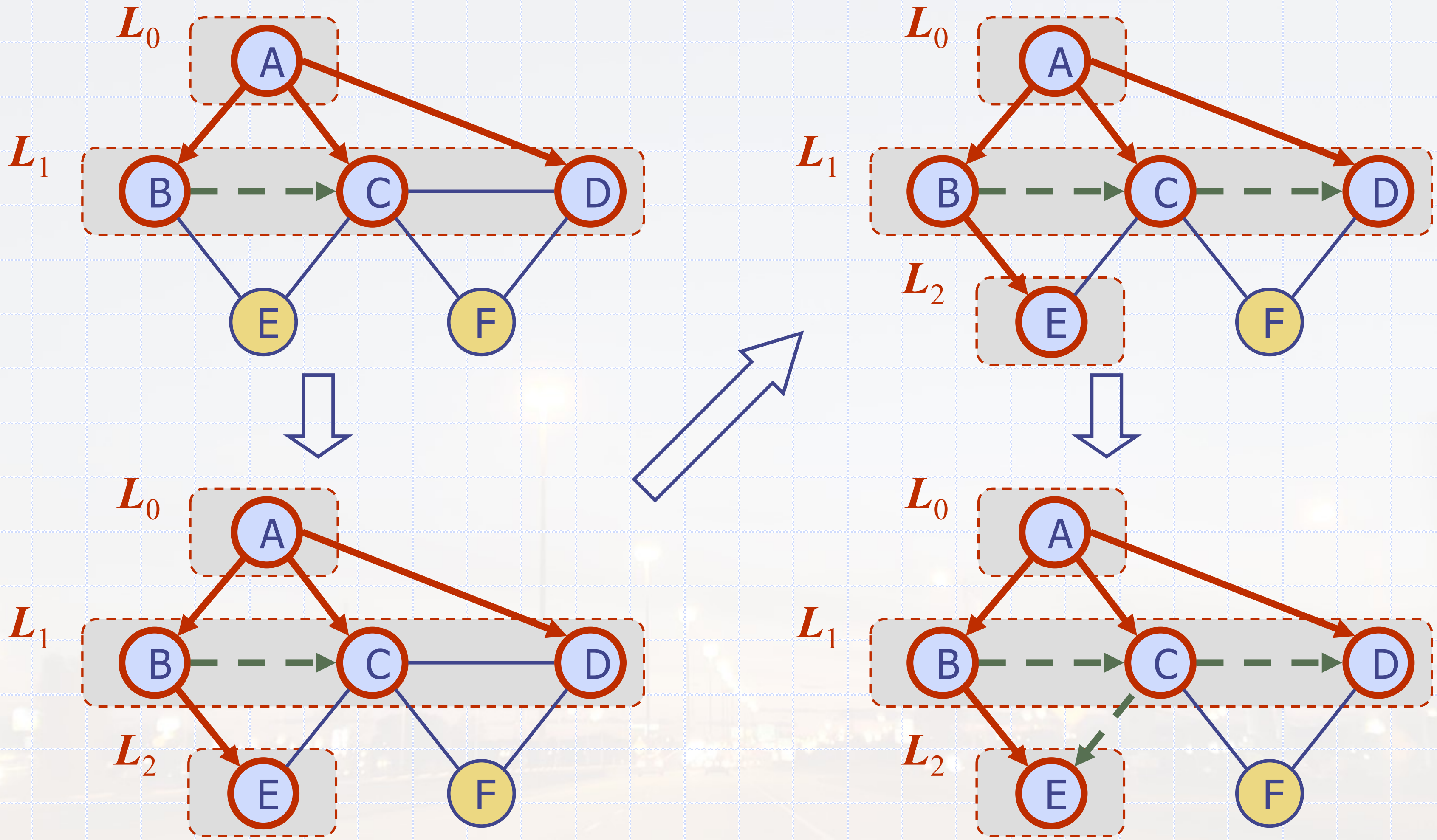
- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- cross edge



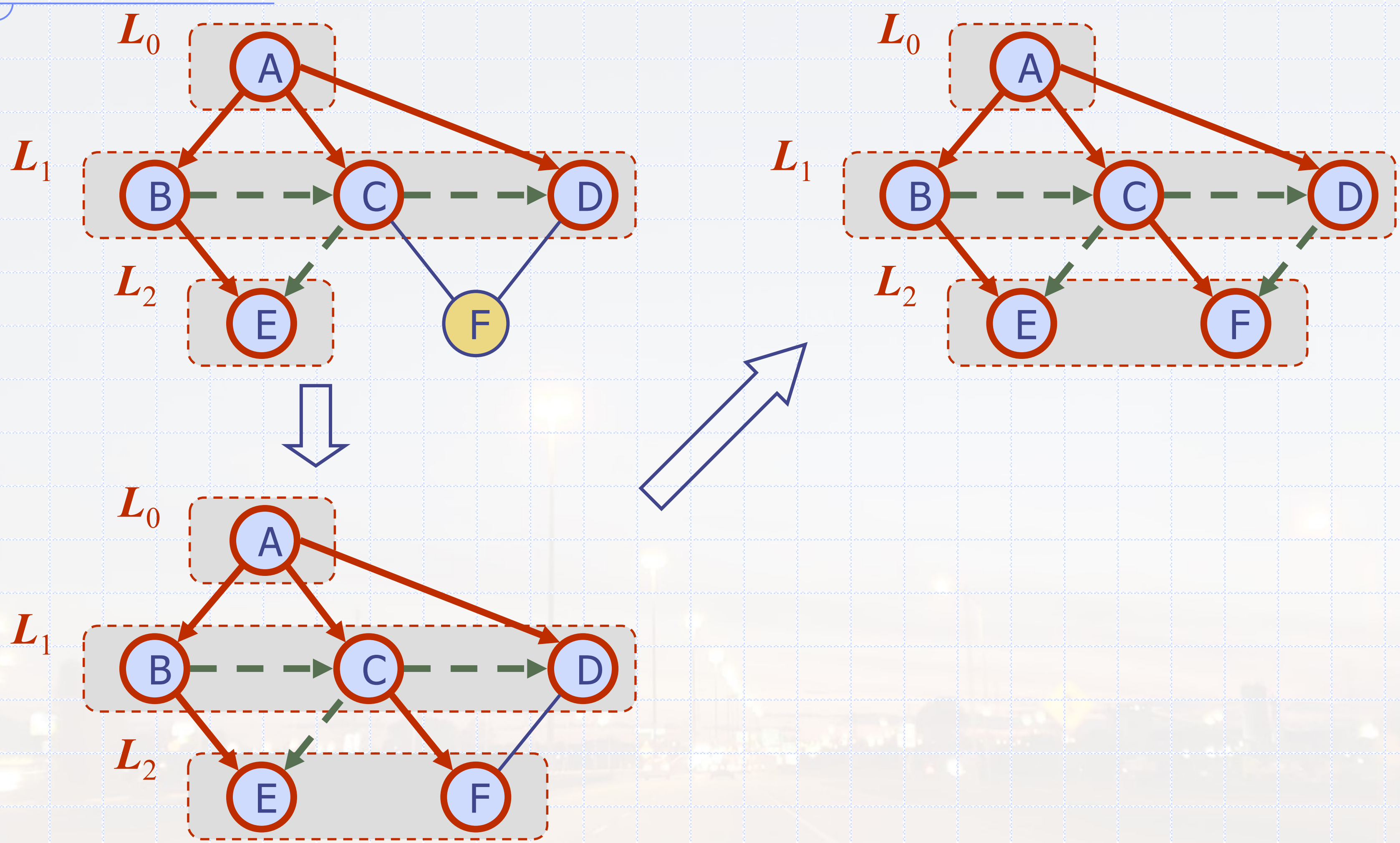




BFS Example1 (cont.)



BFS Example1 (cont.)



BFS Algorithm

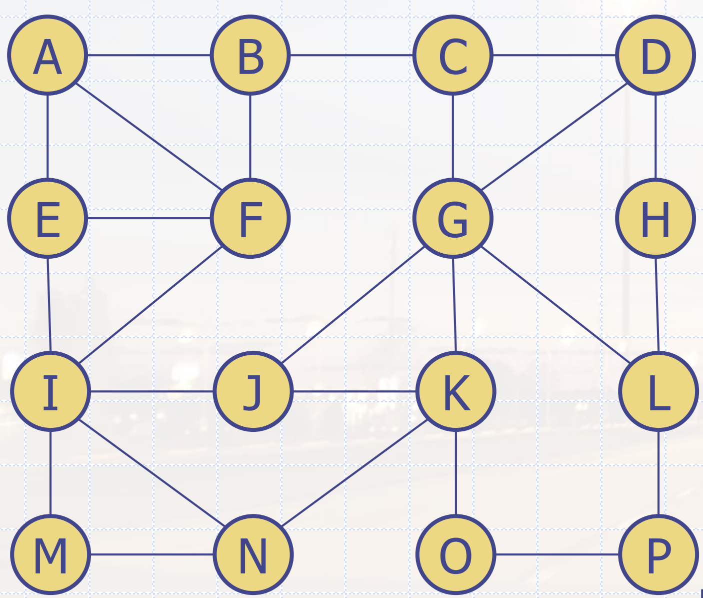
- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

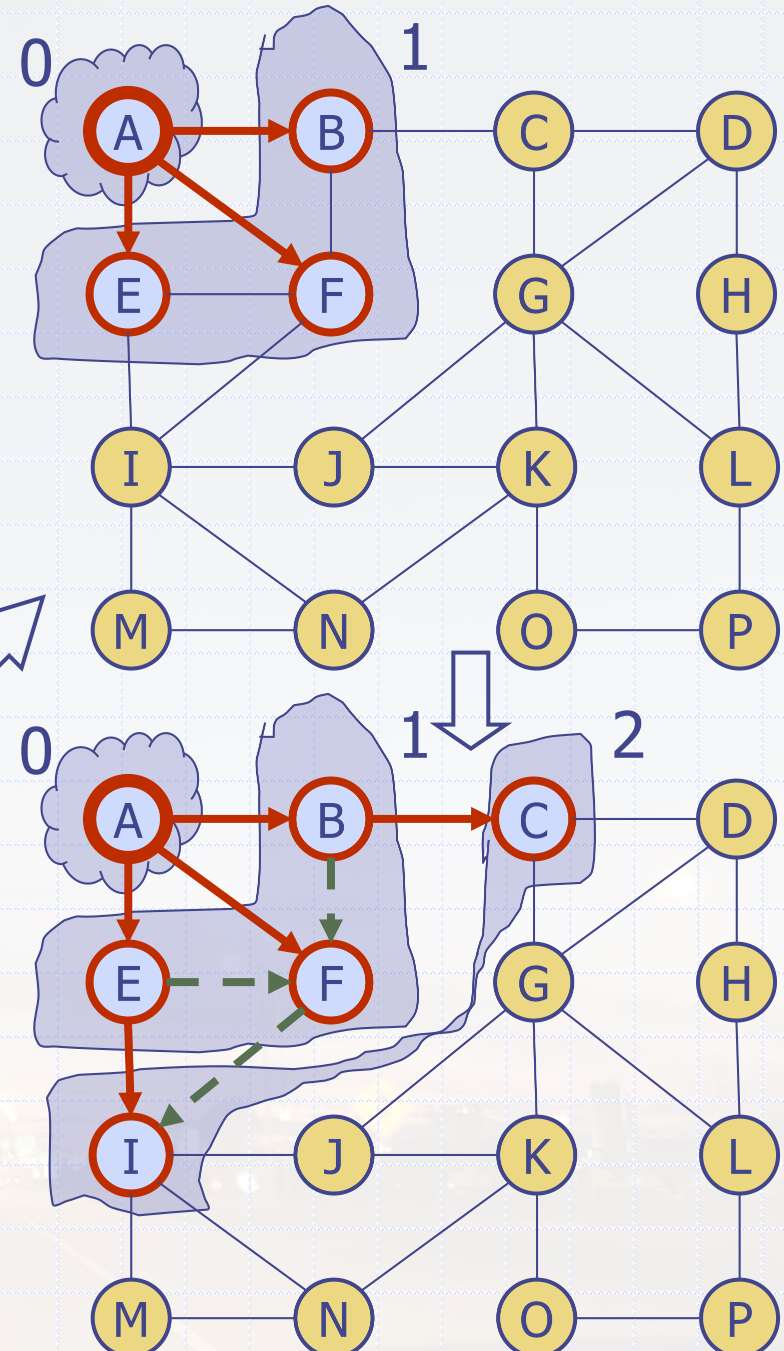
```
Algorithm BFS(G)
   Input graph G
   Output labeling of the edges
      and partition of the
      vertices of G
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
   L0 ← new empty sequence
   L0.addLast(s)
   setLabel(s, VISITED)
   i ← 0
   while ¬Li.isEmpty()
      Li+1 ← new empty sequence
      for all v ∈ Li.elements()
         for all e ∈ G.incidentEdges(v)
            if getLabel(e) = UNEXPLORED
               w ← opposite(v,e)
               if getLabel(w) = UNEXPLORED
                  setLabel(e, DISCOVERY)
                  setLabel(w, VISITED)
                  Li+1.addLast(w)
               else
                  setLabel(e, CROSS)
      i ← i +1
```

BFS Example2

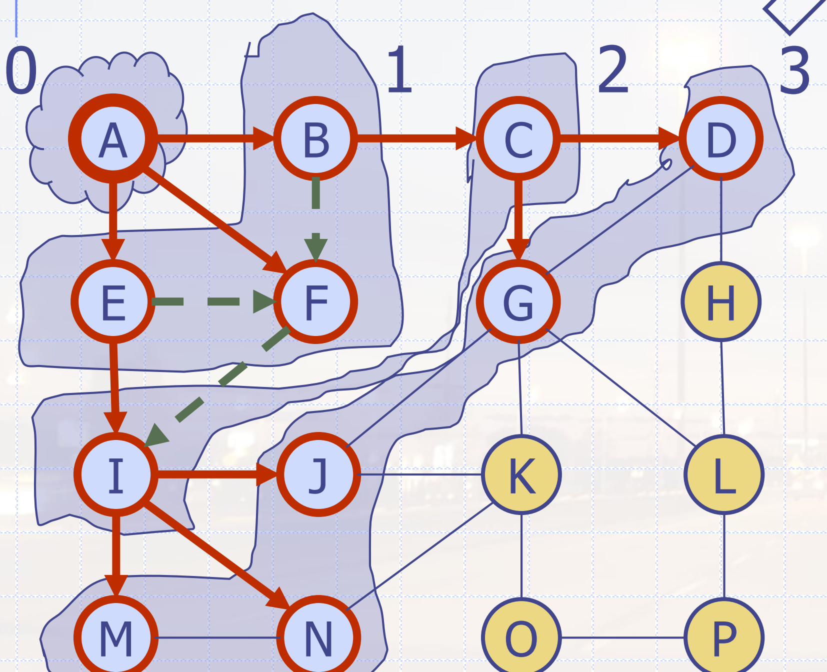
- A — unexplored vertex
- A — visited vertex (Thicker one indicates the anchor)
- unexplored edge
- discovery edge
- cross edge

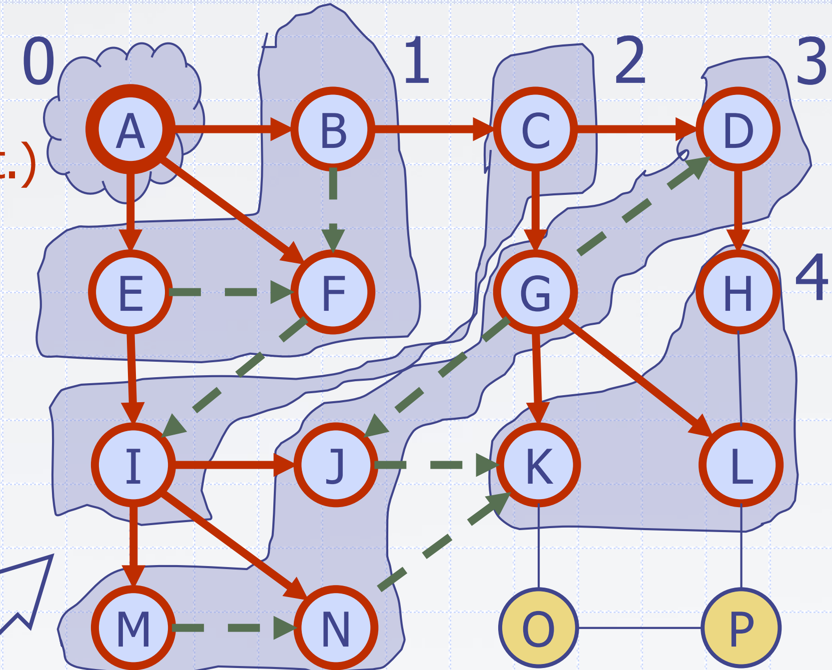


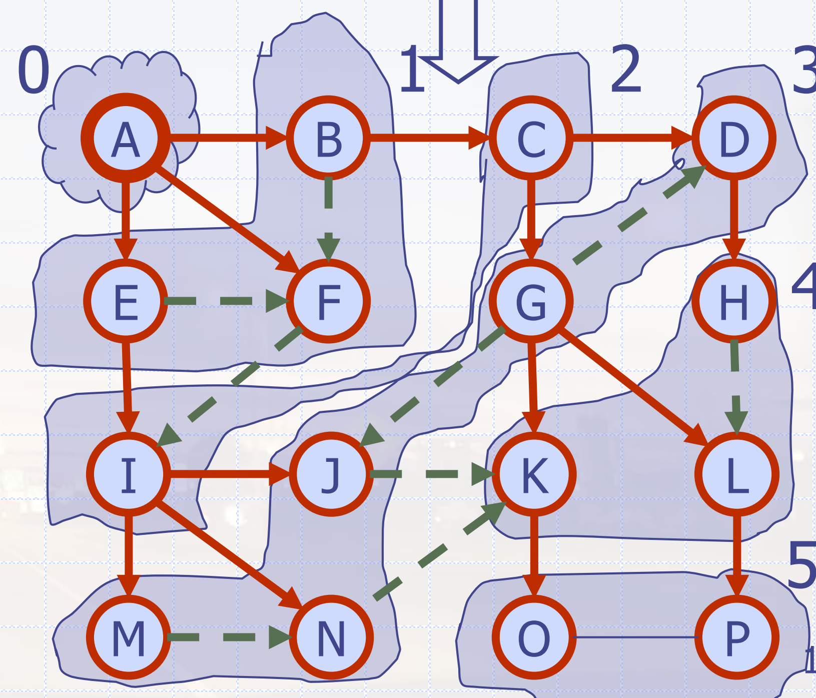


BFS Example2 (cont.)

- A unexplored vertex
- A visited vertex (Thicker one indicates the anchor)
- unexplored edge
- discovery edge
- cross edge







BFS Example2 (cont.)

A unexplored vertex

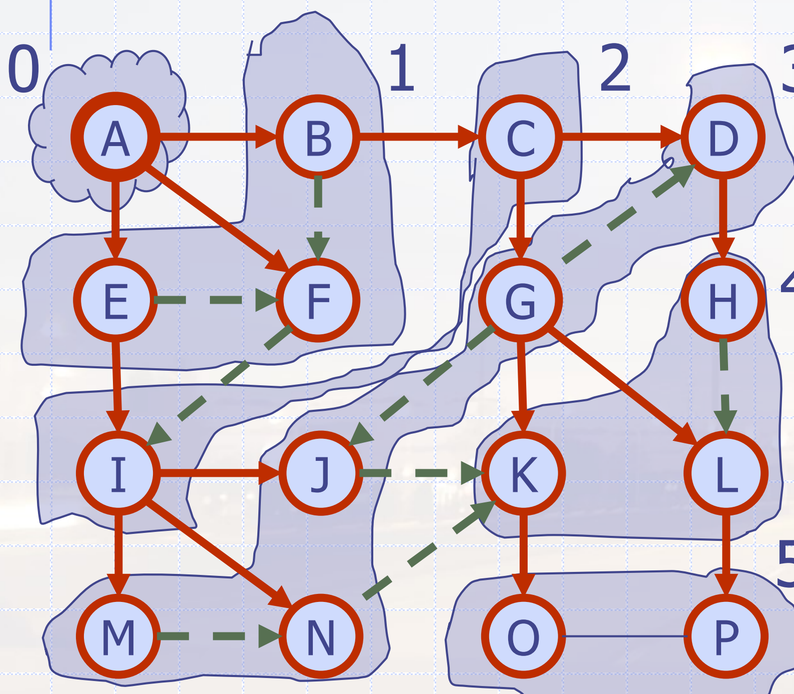
A visited vertex

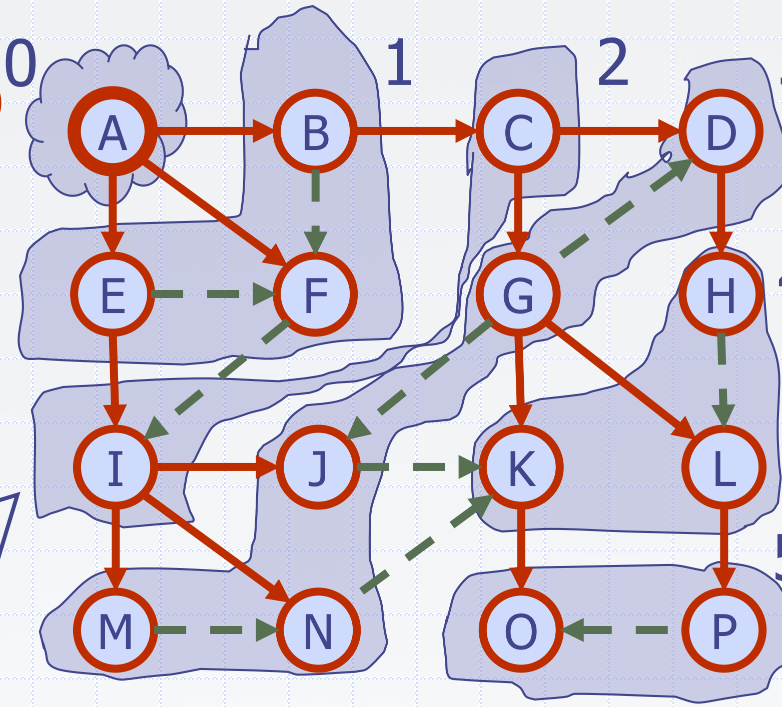
(Thicker one indicates the anchor)

unexplored edge

discovery edge

cross edge





Nothing is added to Level 6. So, terminate as Level 6 is empty

BFS Analysis

- Notice that:
  - Discovery edges form a spanning tree, which is referred to as ***BFS tree***
  - Edges to already visited vertices are called "cross" edges and not "back" edges since these edges do not connect vertices to their ancestors. In fact all of these (non-tree) edges neither connect a vertex to its ancestor, nor to its descendant on the BFS tree
  - We never go back to the anchor; instead we always start from the nodes of $level_i$ to go to $level_{i+1}$

BFS Analysis

- Using a BFS traversal over a graph G, it is possible to solve the following problems:
  (note: that is actually done using algorithms and subroutines that are slightly different than the above simplified shown algorithm)
  - Visit all the vertices and edges of G
  - Determine whether G is connected
  - Compute spanning tree of G is G is connected
  - Compute the spanning forest of G (all spanning trees) if G is a non-connected graph
  - Compute the connected components of G
  - Find a cycle of G, or report that G has no cycles
  - <u>Given a start vertex s of G, compute for every vertex v of G, a path with the **minimum number of edges** between s and v, or report that no such path exists</u>
- However, what is the complexity of BFS?

BFS Analysis

- BFS on a graph with n vertices and m edges takes $O(n + m)$ time since:
  - 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 a sequence L_i
  - 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

- The other problems that BFS can solve can also be achieved in $O(n + m)$

BFS Properties

Notation

G_s: connected component of s

Property 1

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

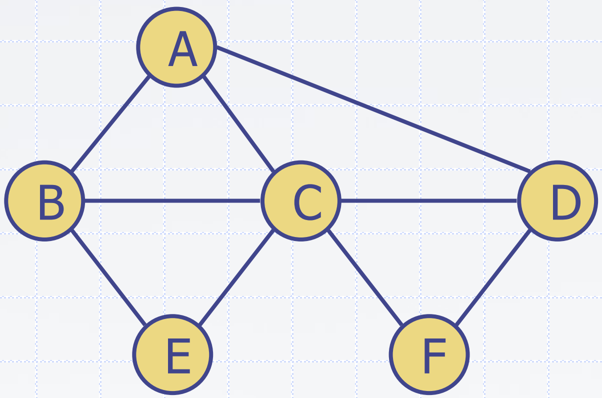
Property 2

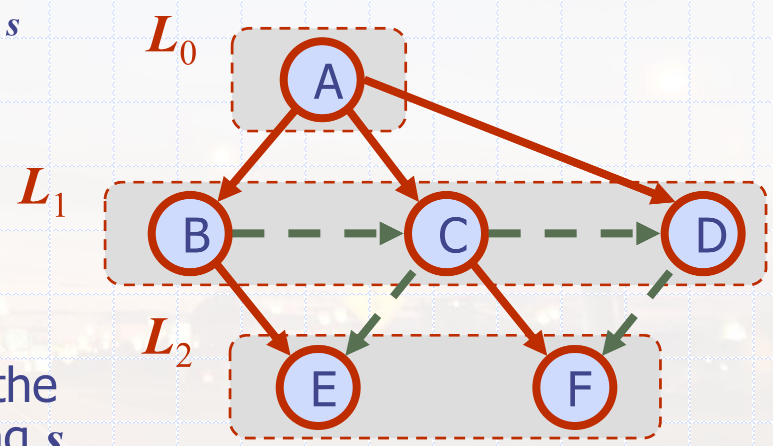
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

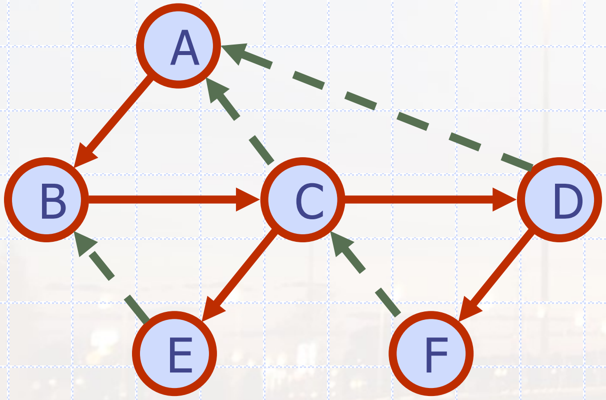
- The path of T_s from s to v has i edges
- Every path from s to v in G_s has the least number of edges connecting s to v (i.e., **shortest path** from s to v)



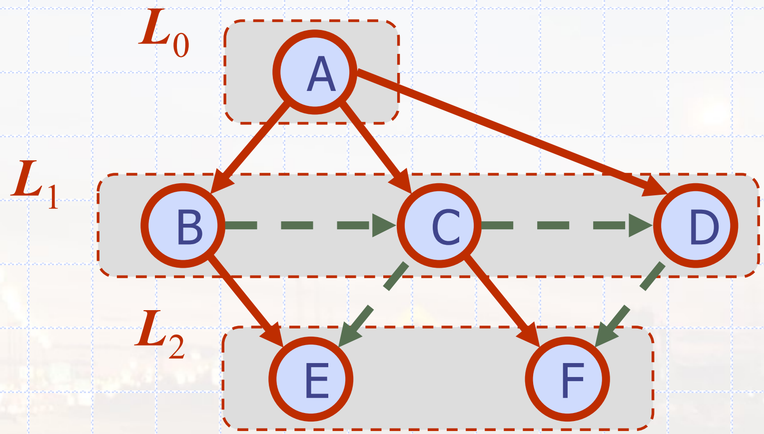


DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components (See next slide for more details)	√	



DFS

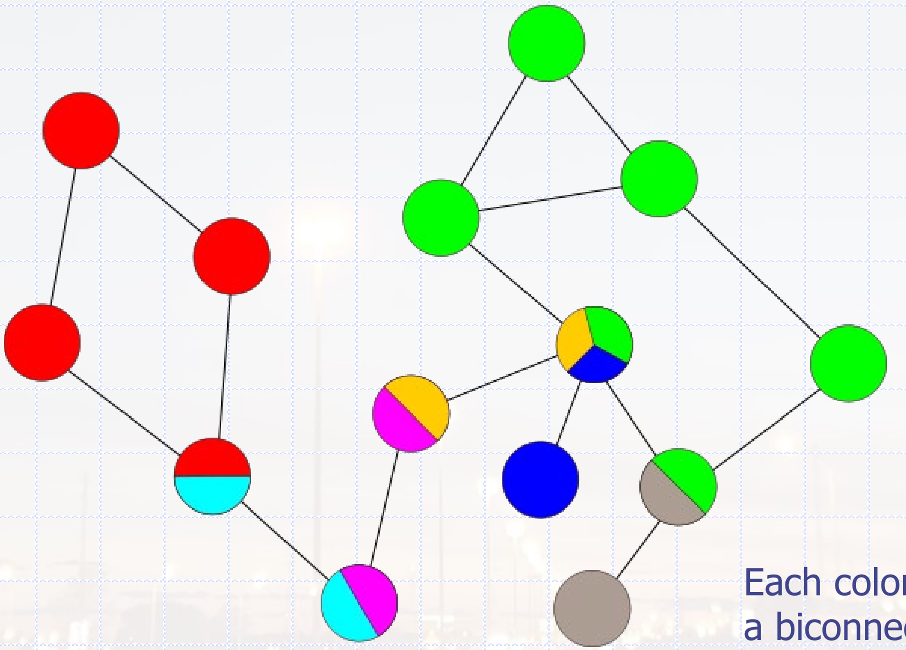


BFS

DFS vs. BFS (cont.)

Biconnected components: Two biconnected components of a graph share at most one vertex in common. A vertex is an articulation point if and only if it is common to more than one biconnected component.

Example*:



Each color corresponds to a biconnected component. Multi-colored vertices are cut vertices, and thus belong to multiple biconnected components*

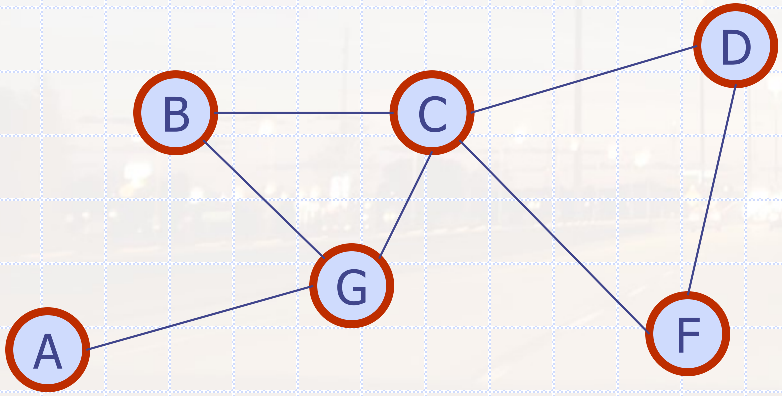
*Source: wikipedia.org

DFS vs. BFS (cont.)

Biconnected components

An *articulation vertex* (or *cut vertex*) is a vertex whose removal increases the number of connected components. A graph is *biconnected* if it has no articulation vertices. Articulation points are important for networks since they represent single point of failure.

DFS can be used to find the articulation points in a graph and compute its disconnected components in $O(n + m)$ time. Click here for more details (see also the notes section of this slide). Many other good sources are also available both online and offline.



The graph has three biconnected components:
{B, C, G},
{C, F, D}, and
{A, G}

C & G are articulation points

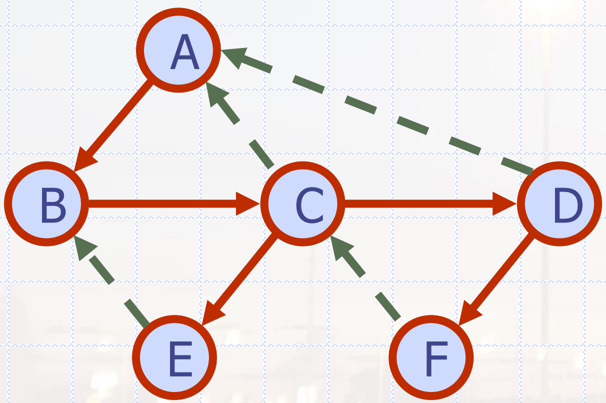
DFS vs. BFS (cont.)

Back edge (v,w)

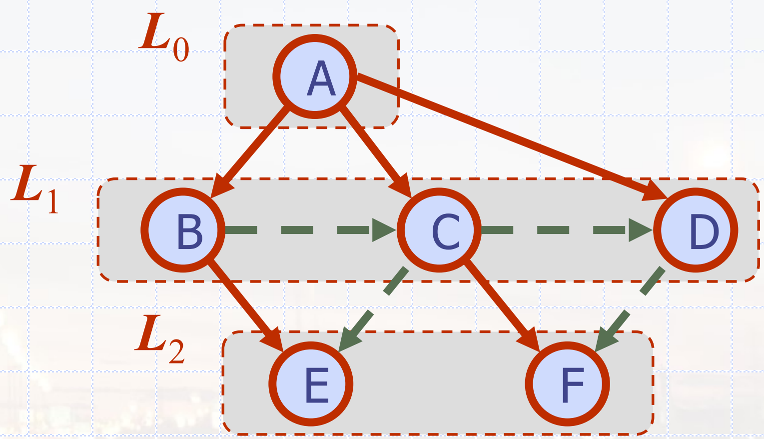
- w is an ancestor of v in the tree of discovery edges. For example, A is an ancestor of C (there is a back edge form C to A)

Cross edge (v,w)

- w is in the same level as v or in the next level. w and w are neither ancestors nor descendent in the BFS tree



DFS



BFS