CSE221

Lecture 18:
Graph Traversals

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

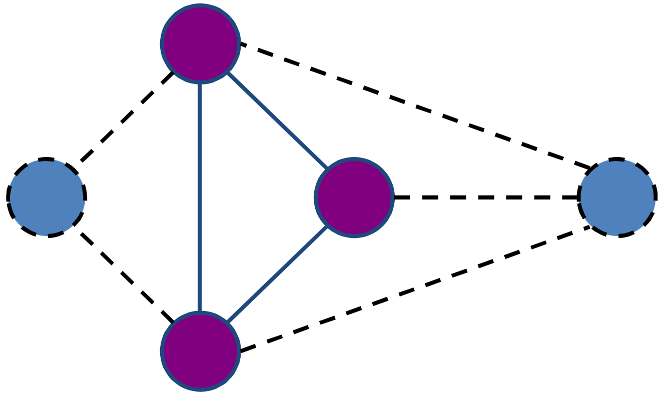
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

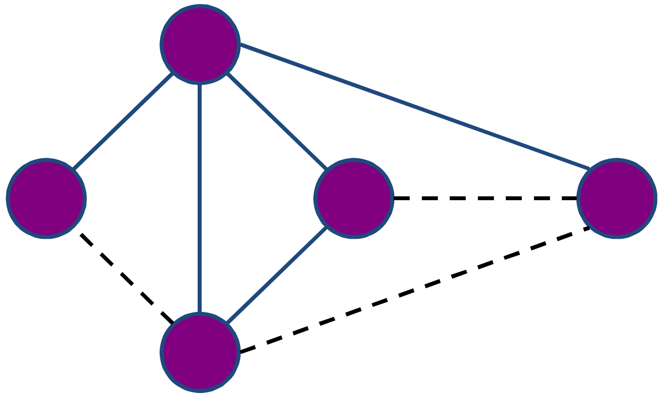
- Depth First Search (DFS)
- Breadth First Search (BFS)

Subgraphs

- A **subgraph** S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A **spanning subgraph** of G is a subgraph that contains all the vertices of G



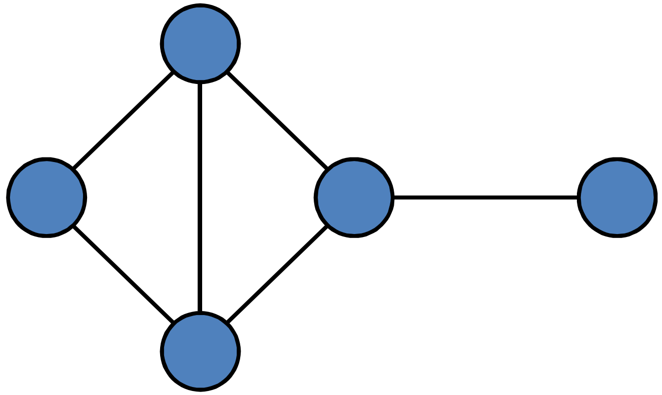
Subgraph



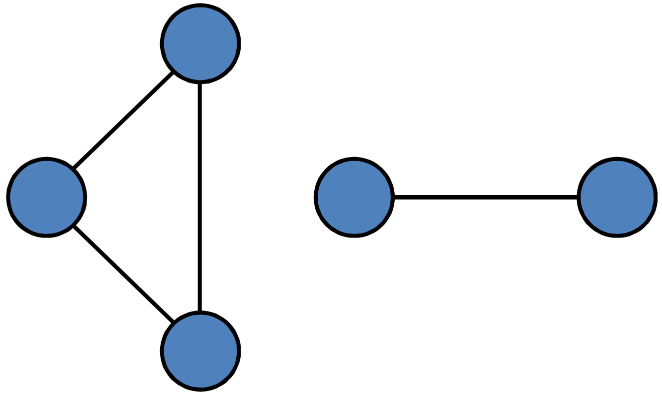
Spanning subgraph

Connectivity

- A graph is **connected** iif there is a path between every pair of vertices
- A **connected component** of a graph G is a maximal connected subgraph of G



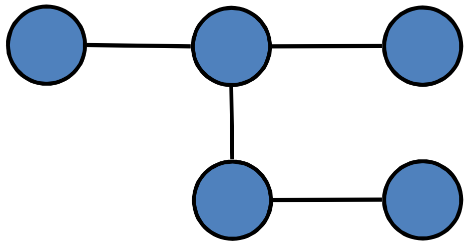
Connected graph



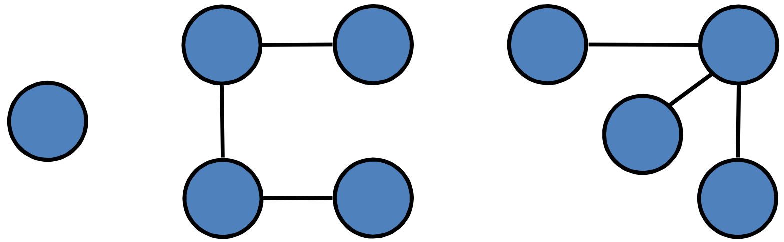
Non connected graph with two connected components

Trees and Forests

- A **tree** is an undirected graph T such that
  - T is connected
  - T has no cycles
  - Note: T is not a rooted tree
- A **forest** is an undirected graph without cycles
  - The connected components of a forest are trees



Tree

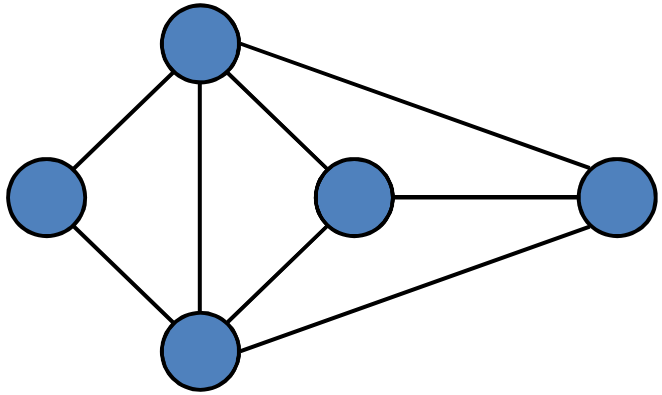


Forest

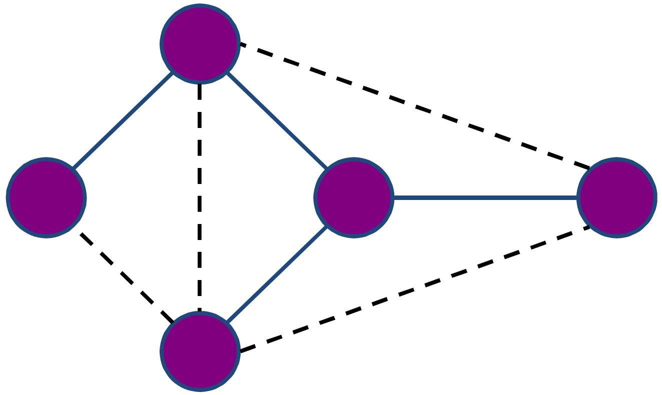
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

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
- A **spanning forest** of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G in a depth-first fashion
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time

- DFS can solve many graph problems
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
  - Find and report a path between two given vertices
  - Find a cycle in the graph

DFS Algorithm

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

Input graph ***G***

Output labeling of the edges of ***G*** as discovery edges and back edges

```
for all u ∈ G.vertices()
    u.setLabel(UNEXPLORED)
for all e ∈ G.edges()
    e.setLabel(UNEXPLORED)
for all v ∈ G.vertices()
    if v.getLabel() = UNEXPLORED
        DFS(G, v)
```

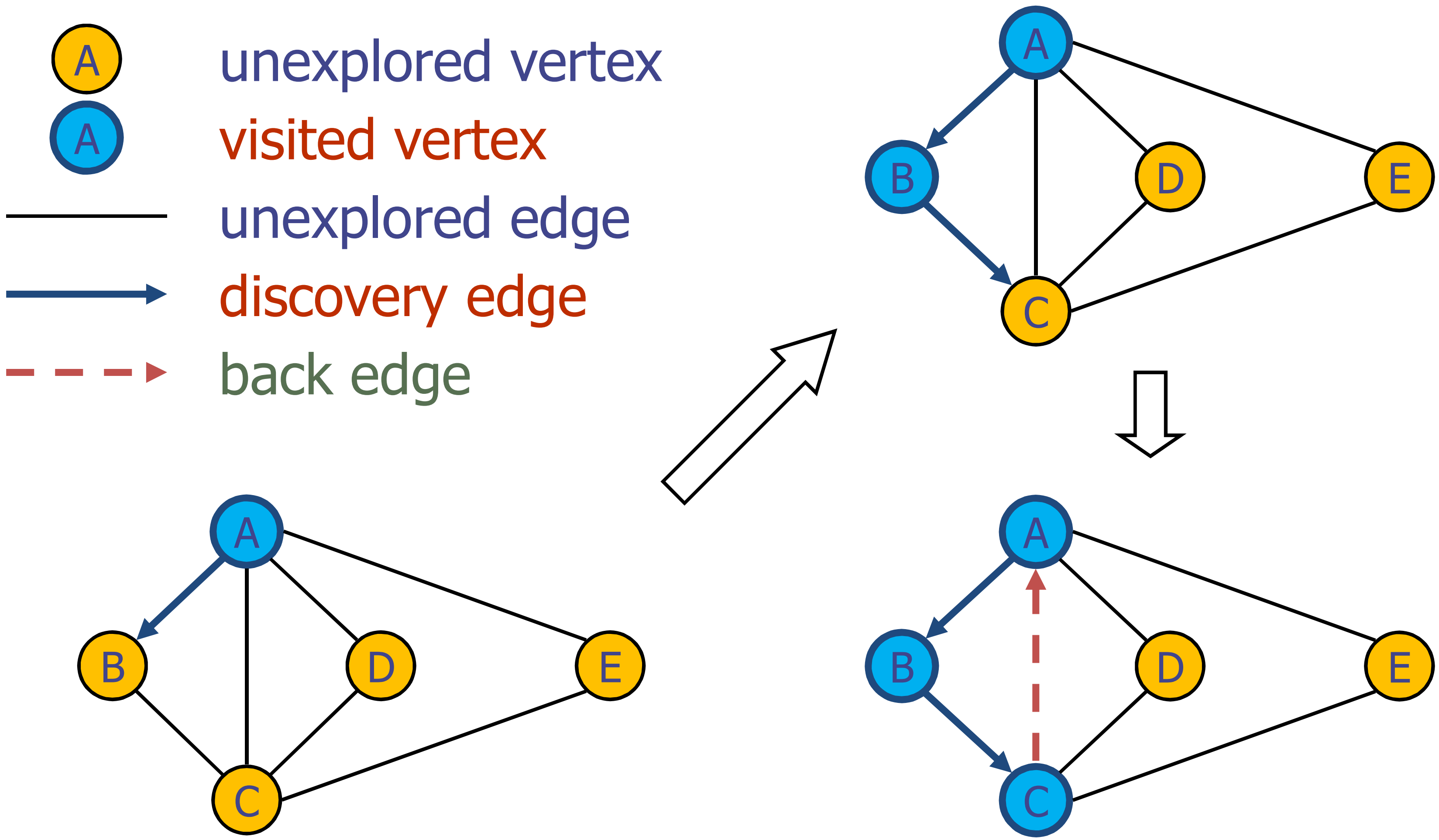
Algorithm ***DFS*(*G, v*)**

Input graph ***G*** and a start vertex ***v*** of ***G***

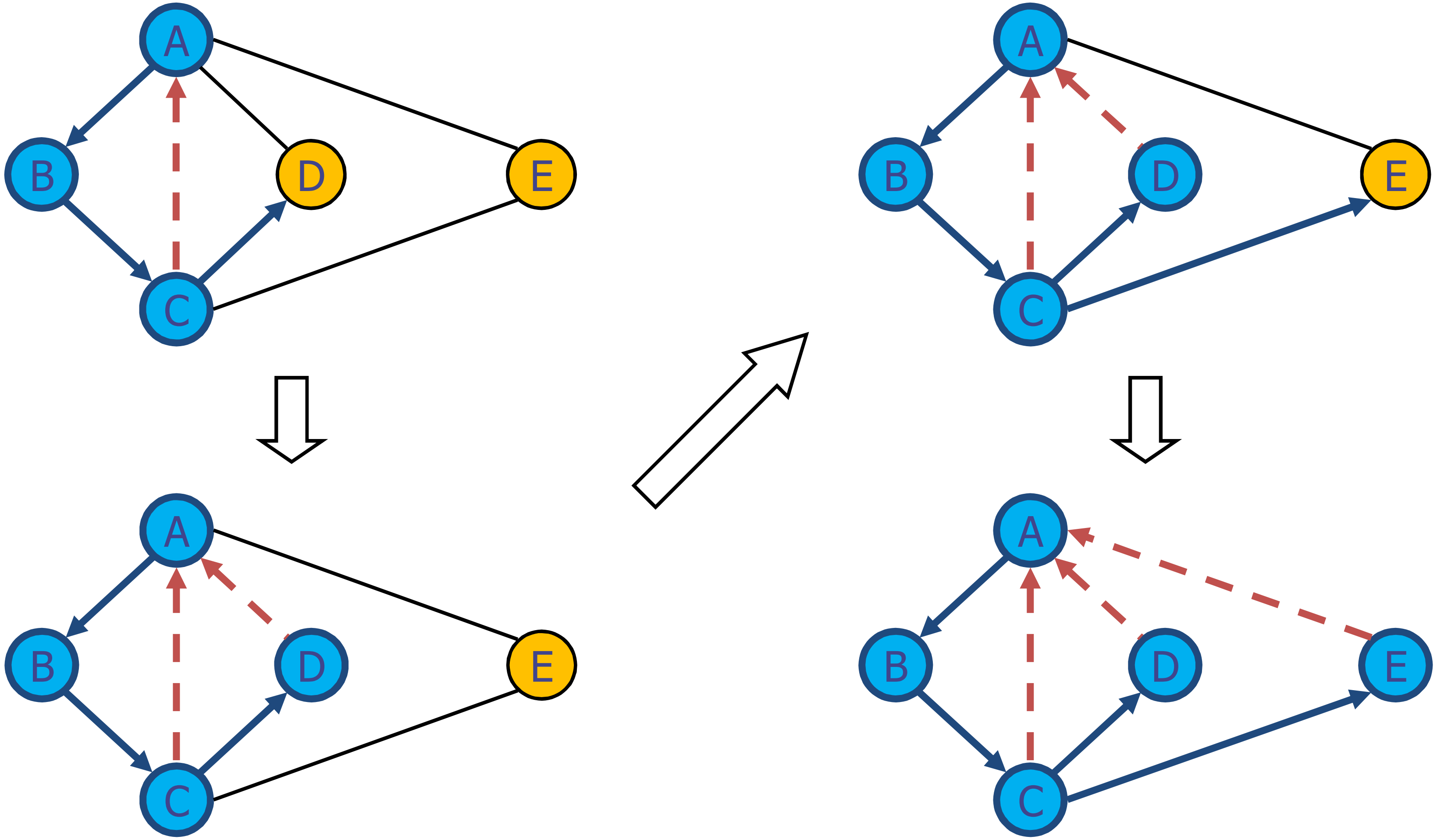
Output labeling of the edges of ***G*** in the connected component of ***v*** as discovery edges and back edges

```
v.setLabel(VISITED)
for all e ∈ G.incidentEdges(v)
    if e.getLabel() = UNEXPLORED
        w ← e.opposite(v)
        if w.getLabel() = UNEXPLORED
            e.setLabel(DISCOVERY)
            DFS(G, w)
        else
            e.setLabel(BACK)
```

Example

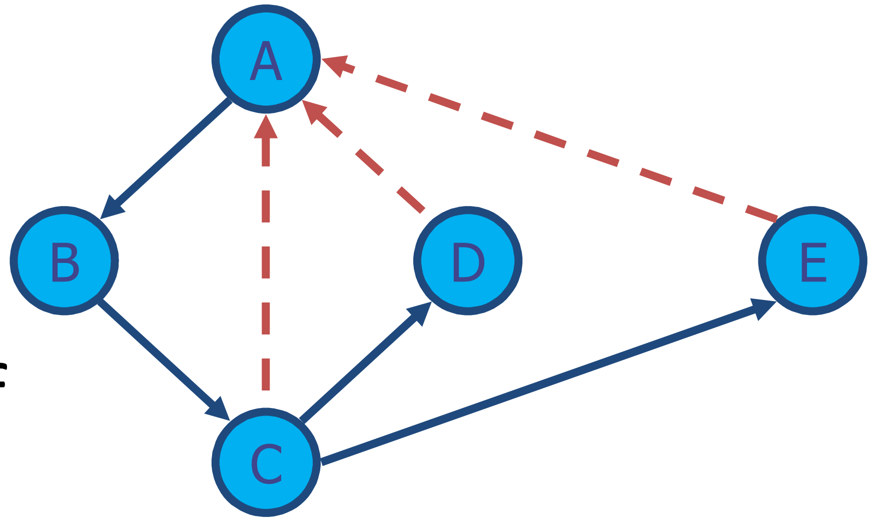

A
unexplored vertex
A
visited vertex
unexplored edge
discovery edge
back edge
A
B
D
E
C
A
B
D
E
C
A
B
D
E
C

Example



Properties of DFS

1) $\boldsymbol{DFS(G, v)}$ visits all the vertices and edges in the connected component of $\boldsymbol{v}$
2) The discovery edges labeled by $\boldsymbol{DFS(G, v)}$ form a spanning tree of the connected component of $\boldsymbol{v}$



Analysis of DFS

- 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 BACK
- DFS is called exactly once on each vertex, and that every edge is examined exactly twice, once from each of its end vertices.
- Method incidentEdges which is called once for each vertex v method takes O(degree(v)) for v provided that the graph is represented by the adjacency list structure
    - Recall that $\Sigma_v \deg(v) = 2m$
- DFS runs in $O(n + m)$ time

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Path Finding

- Find a path between two given vertices ***v*** and ***z***
- Use a stack ***S***
  - Keep track of the path between vertex ***v*** and the current vertex
  - As soon as vertex ***z*** is encountered, return the contents of the stack as the path from ***v*** to ***z***

```
Algorithm pathDFS(G, v, z)
  v.setLabel(VISITED)
  S.push(v)
  if v = z
    return S
  for all e ∈ v.incidentEdges()
    if e.getLabel() = UNEXPLORED
      w ← e.opposite(v)
      if w.getLabel() = UNEXPLORED
        e.setLabel(DISCOVERY)
        pathDFS(G, w, z)
      else
        e.setLabel(BACK)
  S.pop()
```

Cycle Finding

- Find a cycle
- Use a stack S
  - Keep track of the path between vertex v and the current vertex
  - As soon as a back edge (v, w) is encountered, return the portion of the stack from the top to vertex w as a cycle

Algorithm *cycleDFS*(G, v)
    v.setLabel(*VISITED*)
    S.push(v)
    for all $e \in$ *v.incidentEdges*()
        if *e.getLabel*() = *UNEXPLORED*
            $w \leftarrow$ *e.opposite*(v)
            if *w.getLabel*() = *UNEXPLORED*
                e.setLabel(*DISCOVERY*)
                cycleDFS(G, w)
            else
                S.push(w)
                return S
    S.pop()

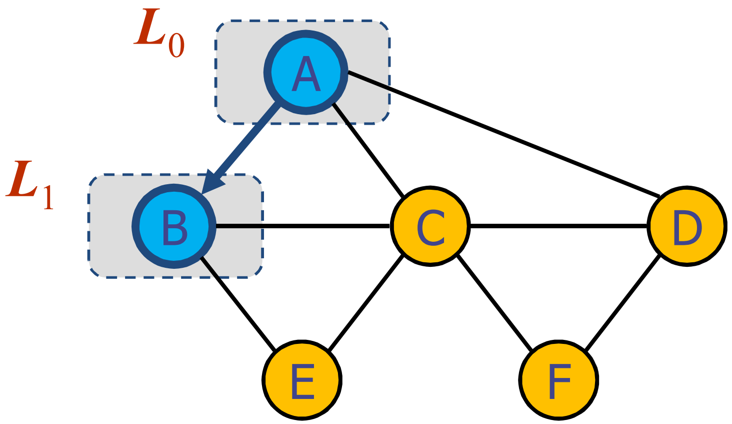
Breadth-First Search

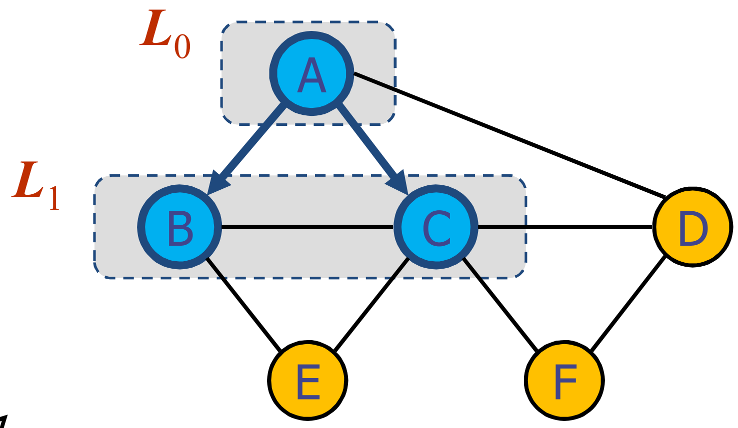
- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G in a breadth-first fashion
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time

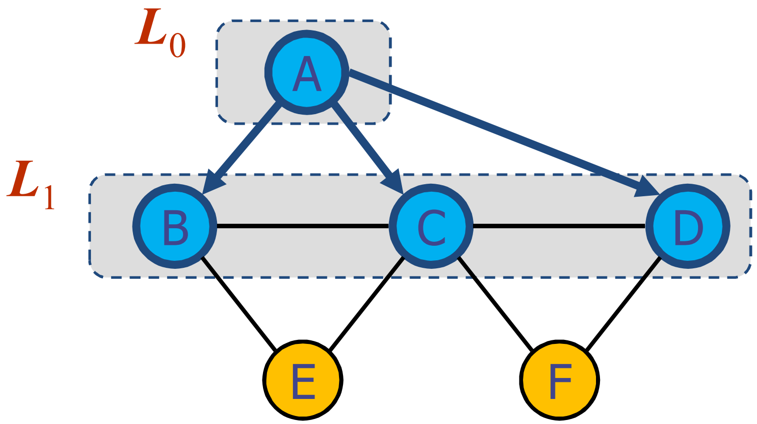
- BFS can solve many graph problems
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
  - Find and report a path with the **minimum number of edges** between two given vertices
  - Find a cycle in the graph

Example

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

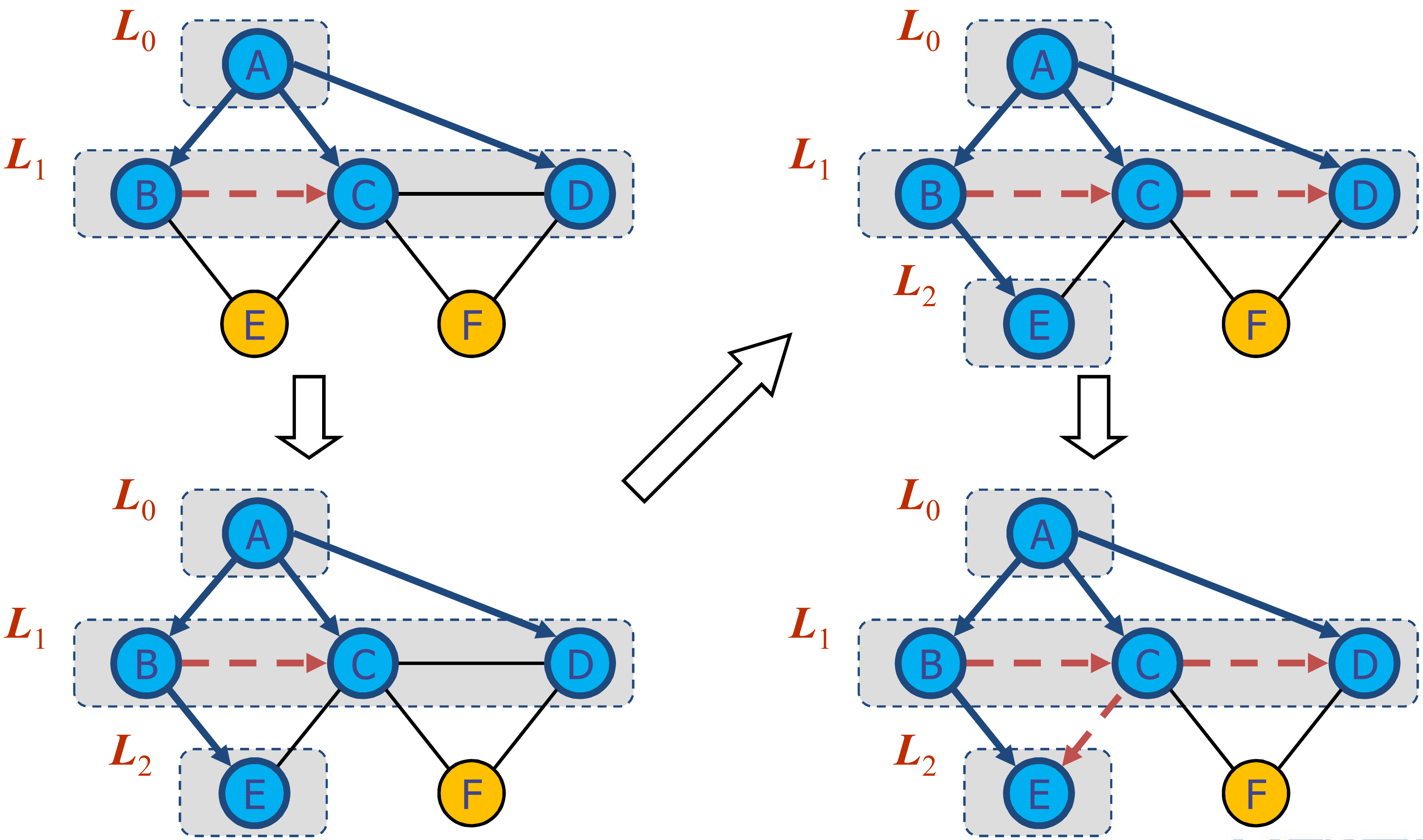




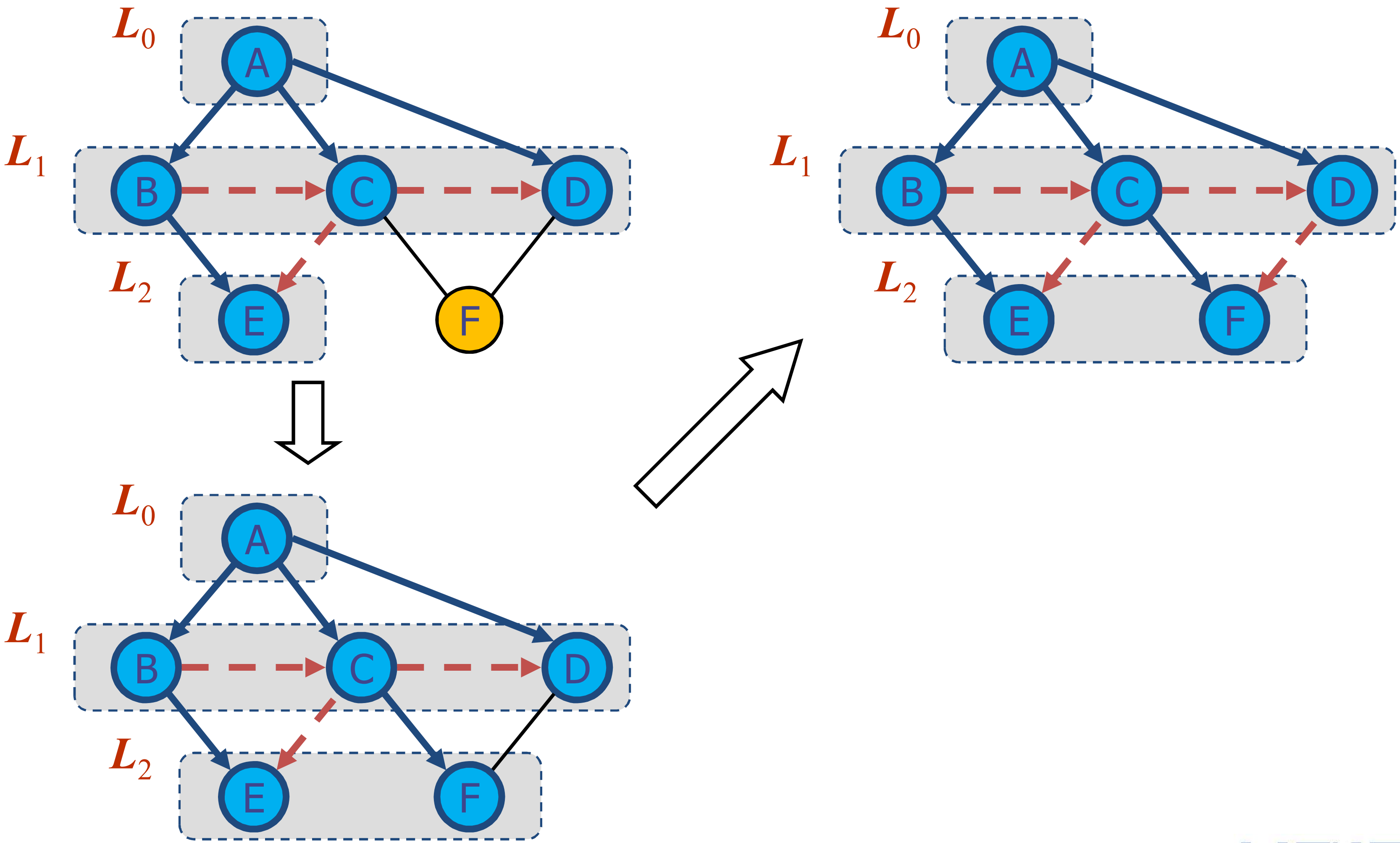


ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Example



Example



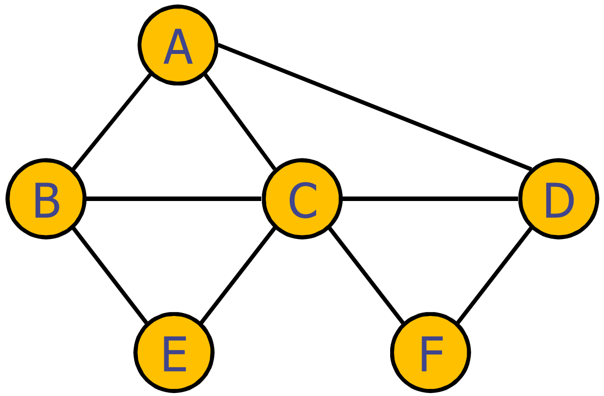
BFS Algorithm

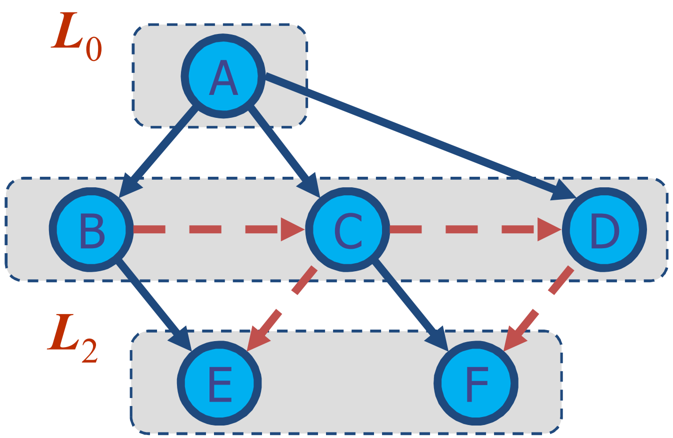
```
Algorithm BFS(G)
  Input graph G
  Output labeling of the edges
      and partition of the
      vertices of G

  for all u ∈ G.vertices()
    u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
    e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
    if v.getLabel() = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L_0 ← new empty sequence
  L_0.insertBack(s)
  s.setLabel(VISITED)
  i ← 0
  while ¬L_i.empty()
    L_{i+1} ← new empty sequence
    for all v ∈ L_i.elements()
      for all e ∈ v.incidentEdges()
        if e.getLabel() = UNEXPLORED
          w ← e.opposite(v)
          if w.getLabel() = UNEXPLORED
            e.setLabel(DISCOVERY)
            w.setLabel(VISITED)
            L_{i+1}.insertBack(w)
          else
            e.setLabel(CROSS)
    i ← i + 1
```

Properties of BFS

1) $BFS(G, s)$ visits all the vertices and edges in the connected components of s
2) The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of the connected component of s
3) For each vertex v in L_i
   - The path of T_s from s to v has i edges (i.e., shortest path)
   - Every path from s to v in G_s has at least i edges





Questions?

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY