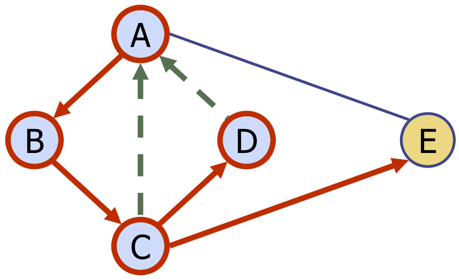
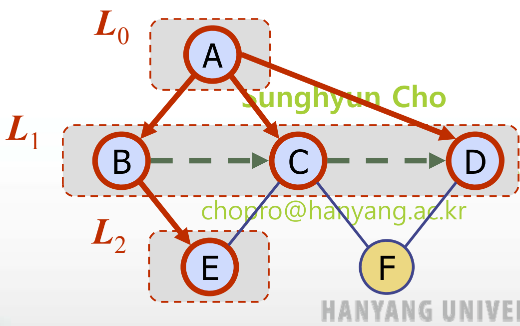
Lecture 10-2: Graph Traversals





unghyun Cho
ering
chopro@hanyang.ac.kr
HANYANG UNIVERSITY

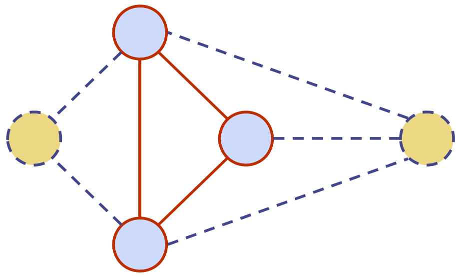
Keywords

- Depth-First Search
- Breadth-First Search

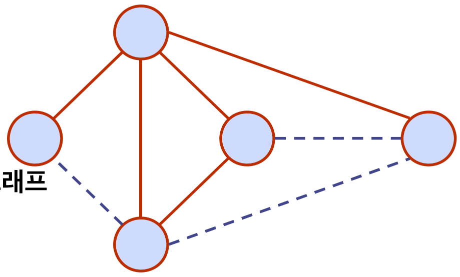
Subgraphs

- 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

그래프가 가지고 있는 모든 vertex를 가지고 있는 그래프
엣지(edge)는 빠져도됨



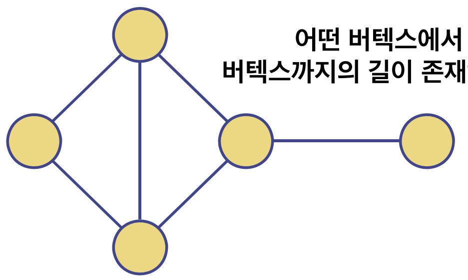
Subgraph



Spanning subgraph

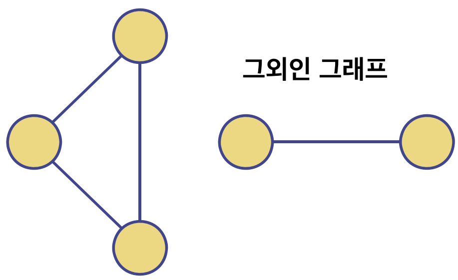
Connectivity

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



어떤 버텍스에서 다른
버텍스까지의 길이 존재하는 그래프

Connected graph



그외인 그래프

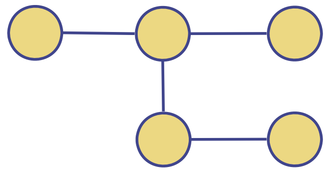
Non connected graph with two connected components

cc: 가장 연결관계가 많은 것들을 모은것

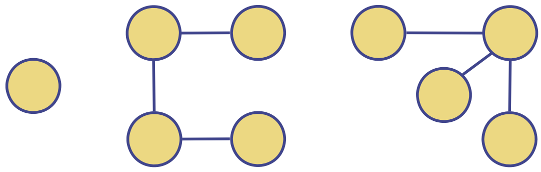
Trees and Forests

- **A (free) tree** is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



Tree



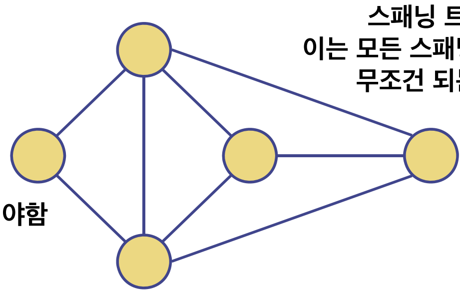
Forest

Spanning Trees and Forests

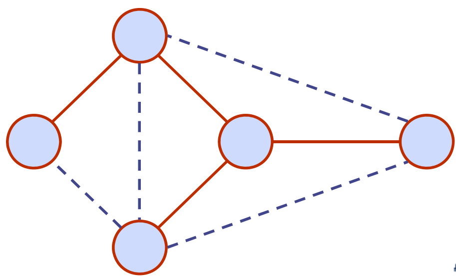
- **A spanning tree of a connected graph is a spanning subgraph that is a tree**
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest

이것은 싸이클이 존재하므로
스패닝 트리가 아니고
이는 모든 스패닝이 스패닝 트리가
무조건 되는것이 아니다

스패닝이면서
트리속성을 만족해야함



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
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - **Find and report a <u>path</u> between two given vertices**
  - **Find a <u>cycle</u> in the graph**
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm

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

```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  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
        DFS(G, v)
```

둘다 공부하기

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

Example

A unexplored vertex

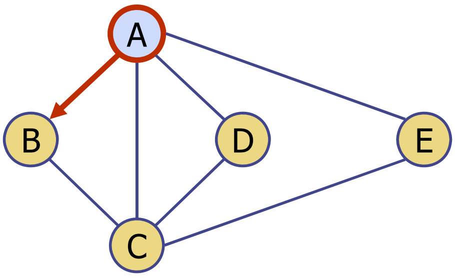
A visited vertex

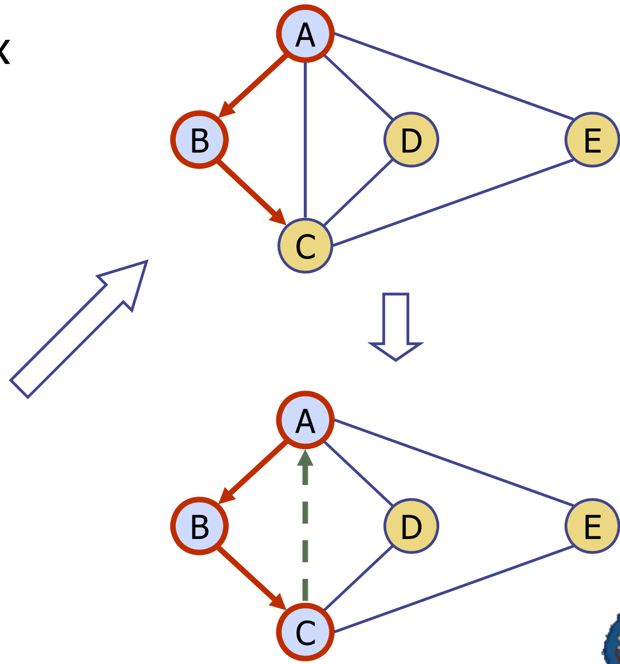
unexplored edge

discovery edge

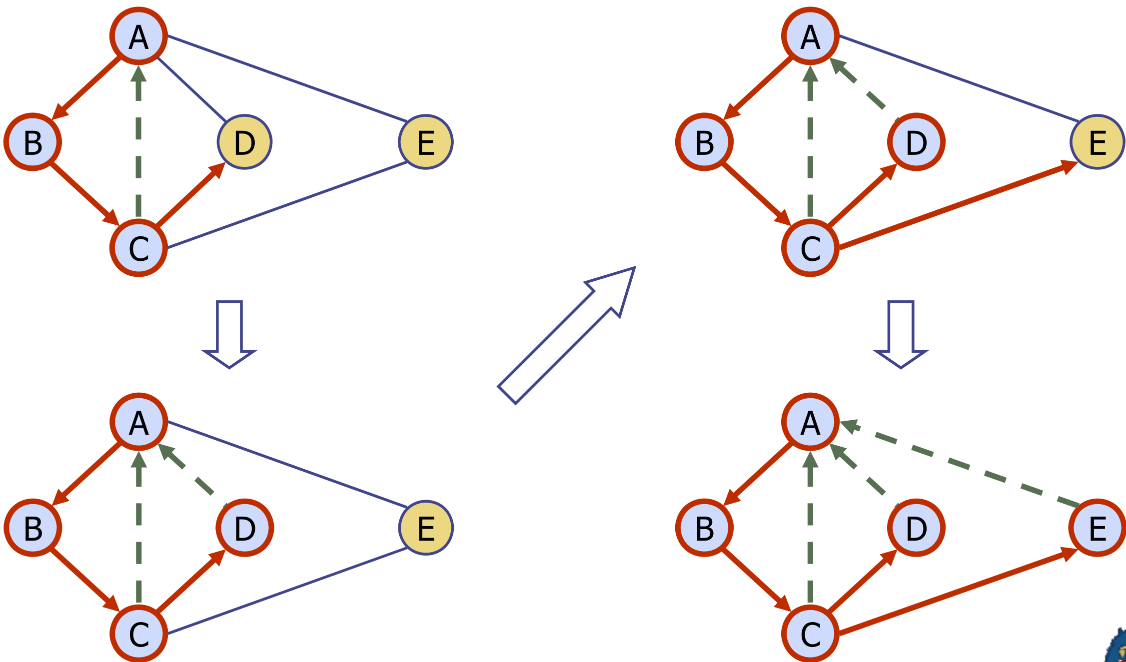
back edge

방문했는데 이미 방문한
버텍스이면 이 길을 **back edge**



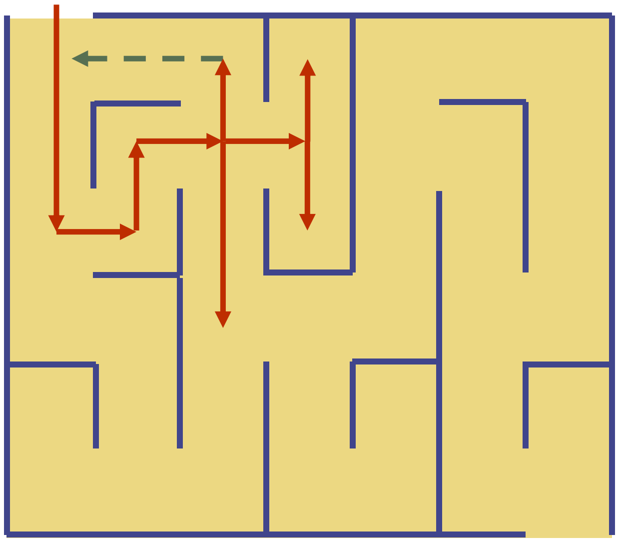


Example (cont.)



DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



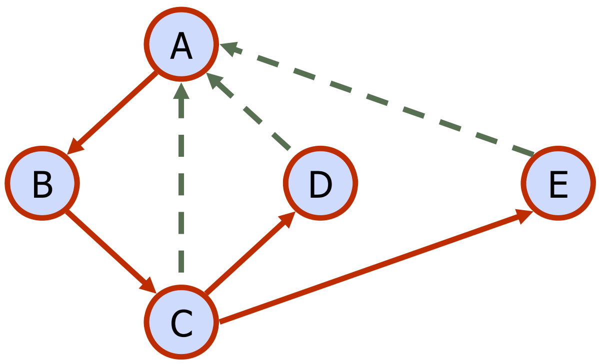
Properties of DFS

Property 1

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

Property 2

The discovery edges labeled by ***DFS*(*G*, *v*)** form a **spanning tree of the connected component of 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
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Path Finding

둘중에 하나 기말고사에 나옴

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- **As soon as destination vertex z is encountered, we return the path as the contents of the stack**

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

Cycle Finding

둘중에 하나 기말고사에 나옴

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all  e ∈ G.incidentEdges(v)
    if  getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```


Keywords

- Breadth-First Search

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
- 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

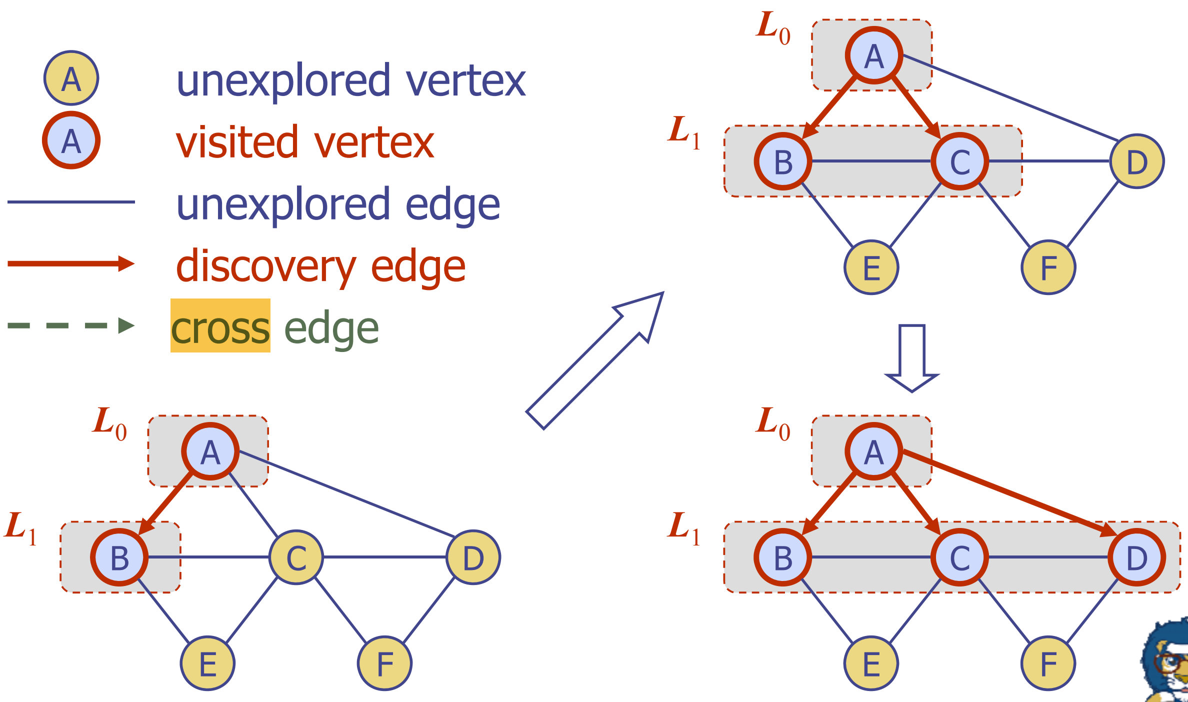
BFS Algorithm

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

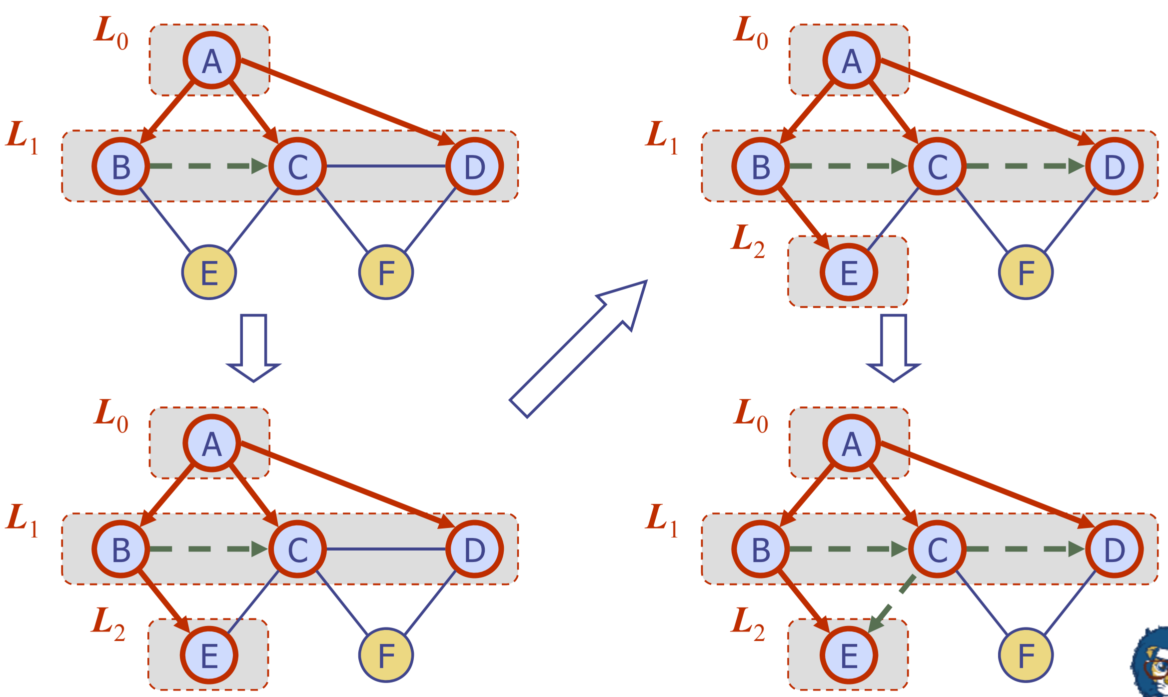
```
Algorithm BFS(G)
   Input graph G
   Output labeling of the edges
      and partition of the
      vertices of G
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

```
Algorithm BFS(G, s)
  L_0 ← new empty sequence
  L_0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬L_i.isEmpty()
    L_{i+1} ← new empty sequence
    for all v ∈ L_i.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)
            L_{i+1}.addLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

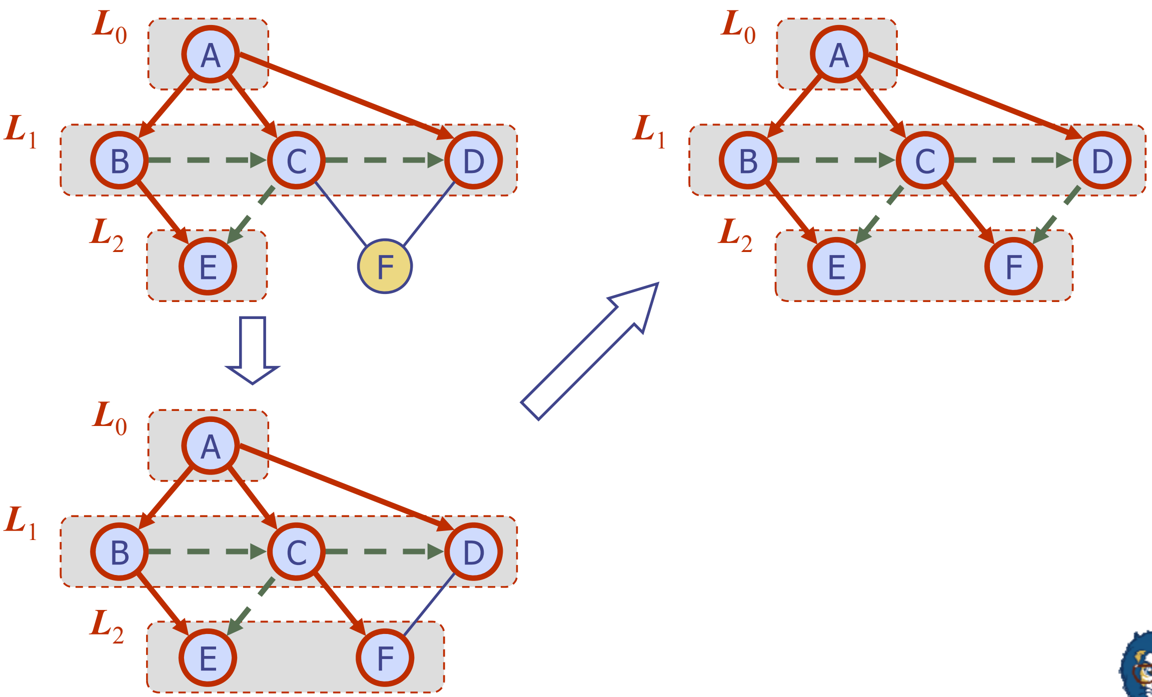
Example



Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

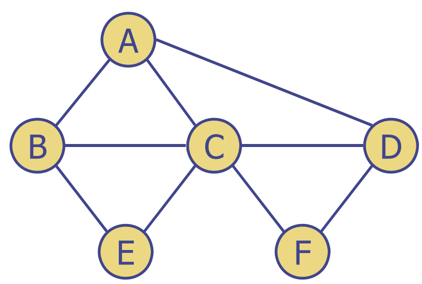
Property 2

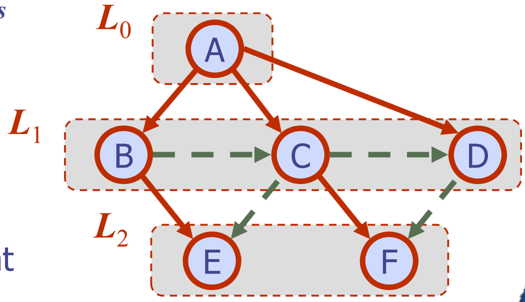
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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





Analysis

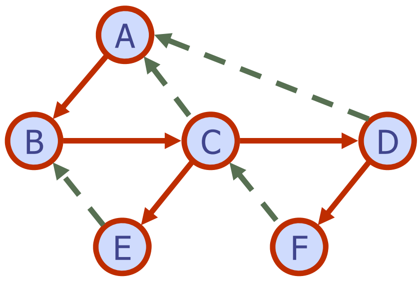
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once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

Applications

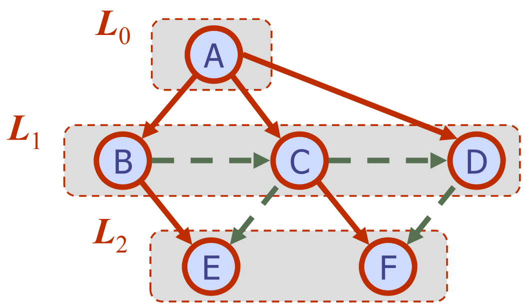
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, 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

Biconnected Graph

- ***Articulation point:*** **An Articulation point in a connected graph is a vertex that, if delete, would break the graph into two or more pieces (connected component).**
- ***Biconnected graph:*** **A graph with no articulation point called biconnected. In other words, a graph is biconnected if and only if any vertex is deleted, the graph remains connected.**
- ***Biconnected component:*** **A biconnected component of a graph is a maximal biconnected subgraph- a biconnected subgraph that is not properly contained in a larger biconnected subgraph.**
- **A graph that is not biconnected can divide into biconnected components, sets of nodes mutually accessible via two distinct paths.**

The graphs we discuss below are all about loop-free undirected ones.

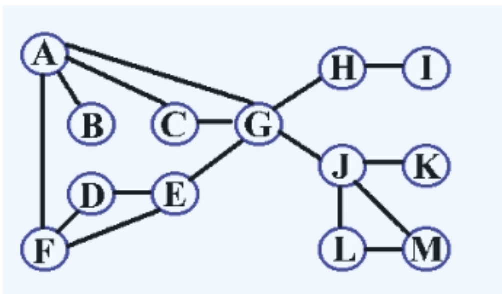


Figure 1. The graph G that is not biconnected

[Example] **Graph G in Figure 1:**

- Articulation points: A, H, G, J
- Biconnected components: {A, C, G, D, E, F} 、{G, J, L, B} 、B 、H 、I 、K

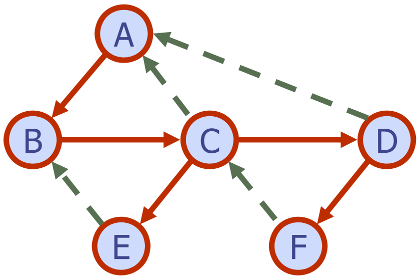
DFS vs. BFS (cont.)

Back edge (v,w)

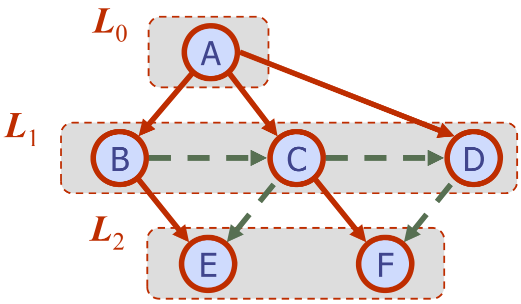
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

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



DFS



BFS

Biconnected Graph

How to find all articulation points in a given graph?

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

1) For every vertex v, do following

…..a) Remove v from graph

…..b) See if the graph remains connected (We can either use BFS or DFS)

…..c) Add v back to the graph

Time complexity of above method is O(V*(V+E)) for a graph represented using adjacency list. Can we do better?

connectivity깨지는 지확인은 dfs

A O(V+E) algorithm to find all Articulation Points (APs)

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1) u is root of DFS tree and it has at least two children.

이 두 조건을 만족하면 articulation point(절단 점)이다

2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

Q & A

한양대학교 ERICA 캠퍼스