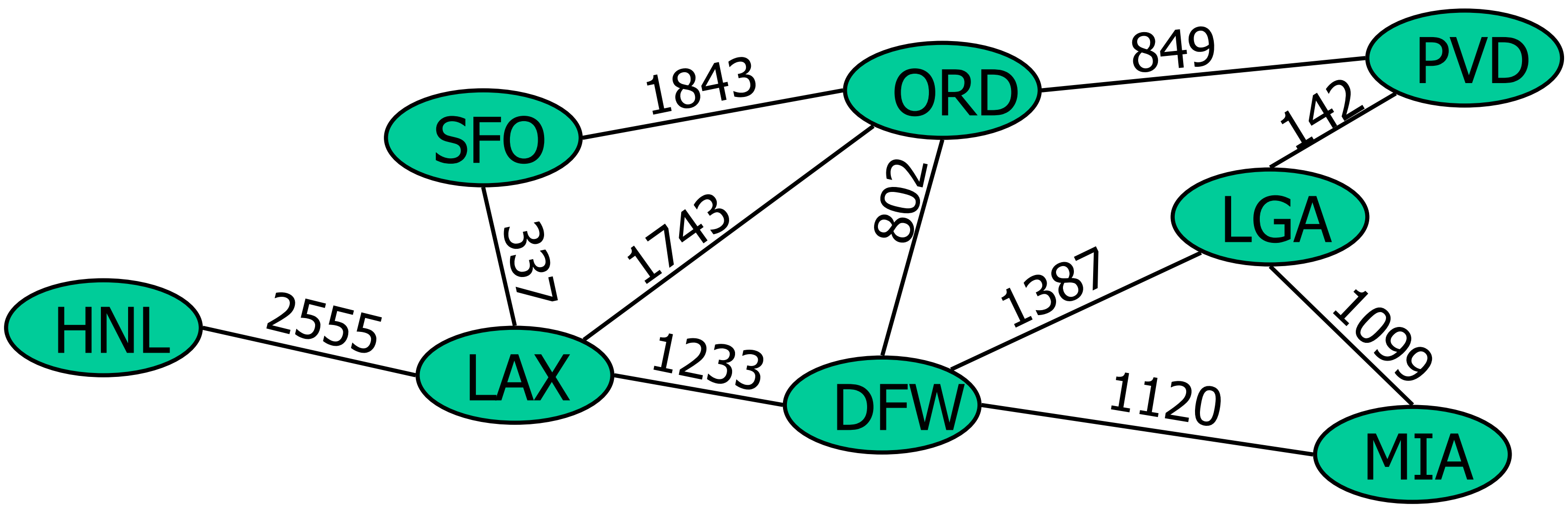
Lecture8: 그래프 (13.1절~13.3절)

김 강 희
khhkim@ssu.ac.kr

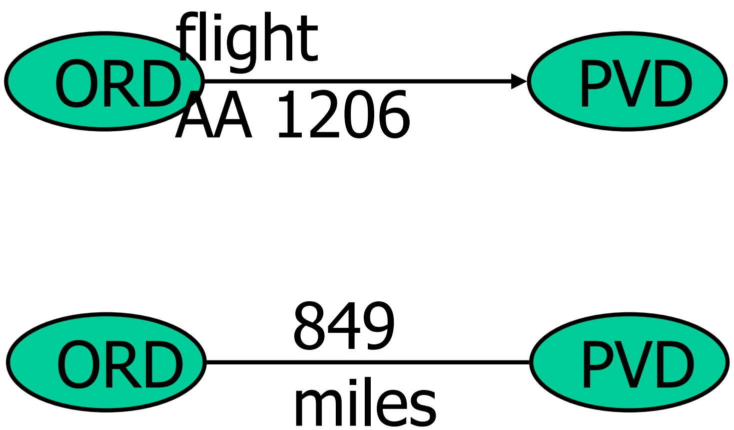
Graphs

- 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



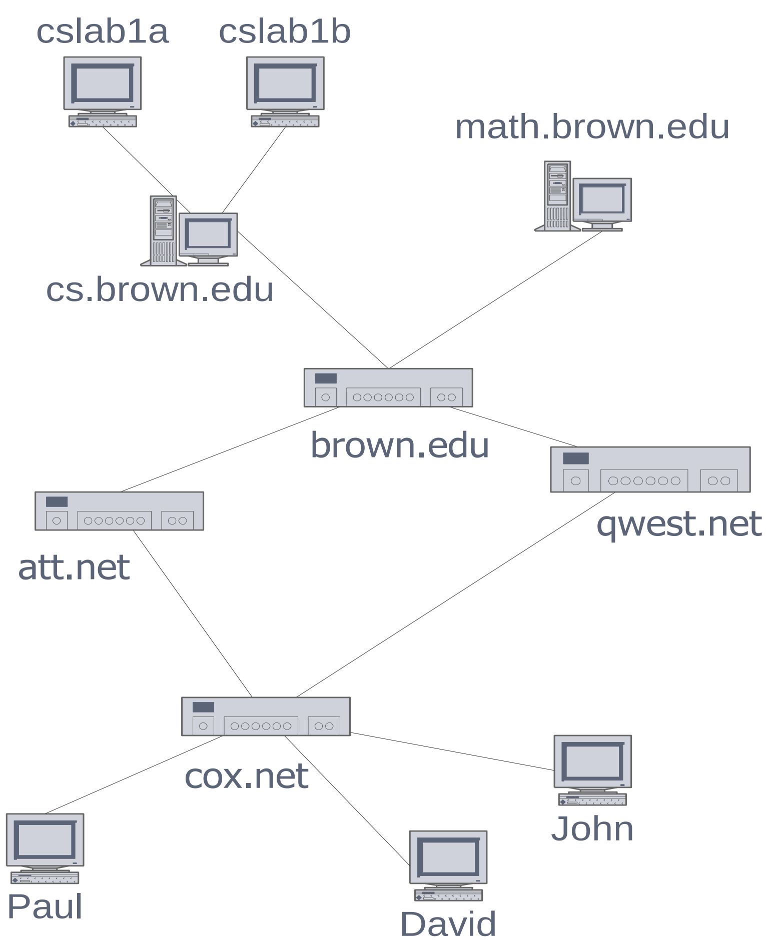
Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network



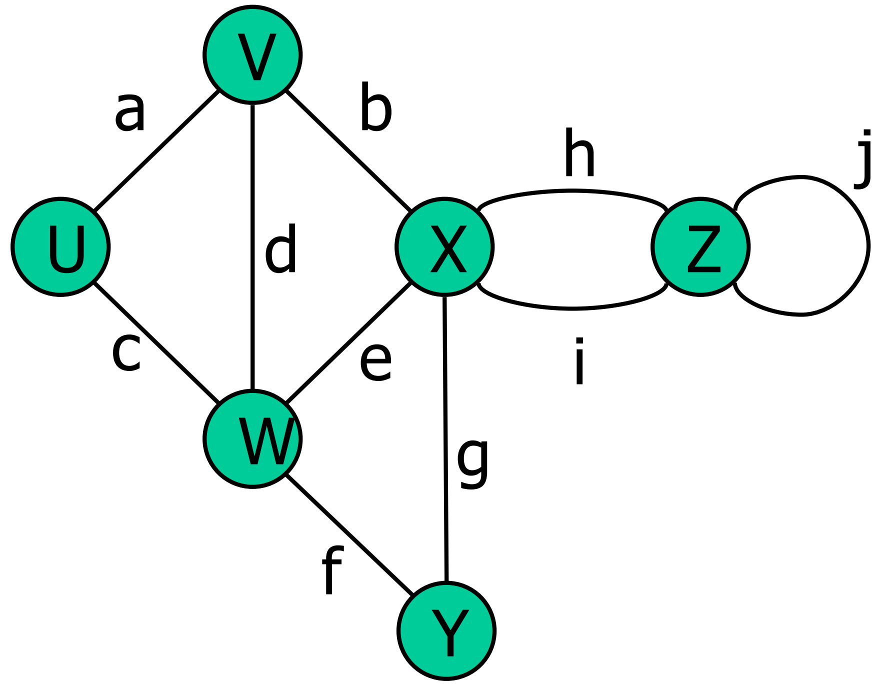
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



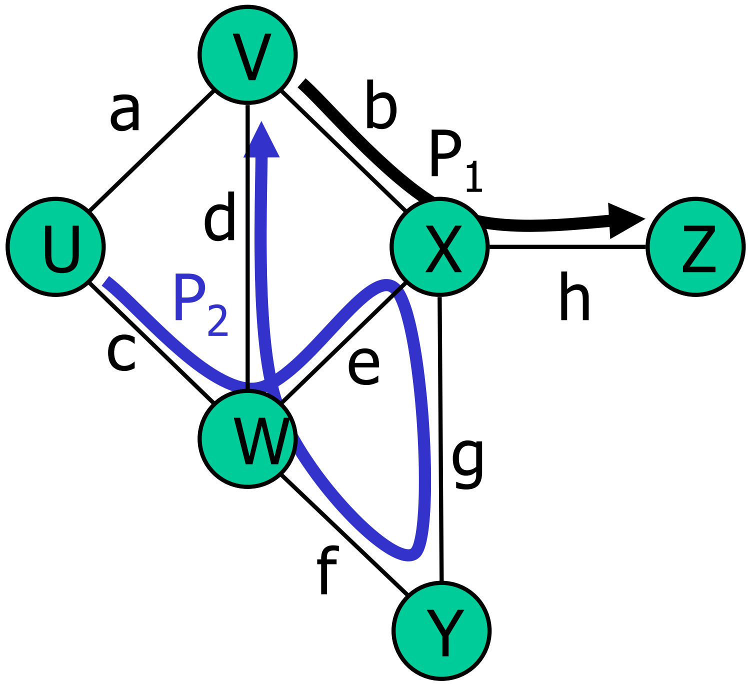
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



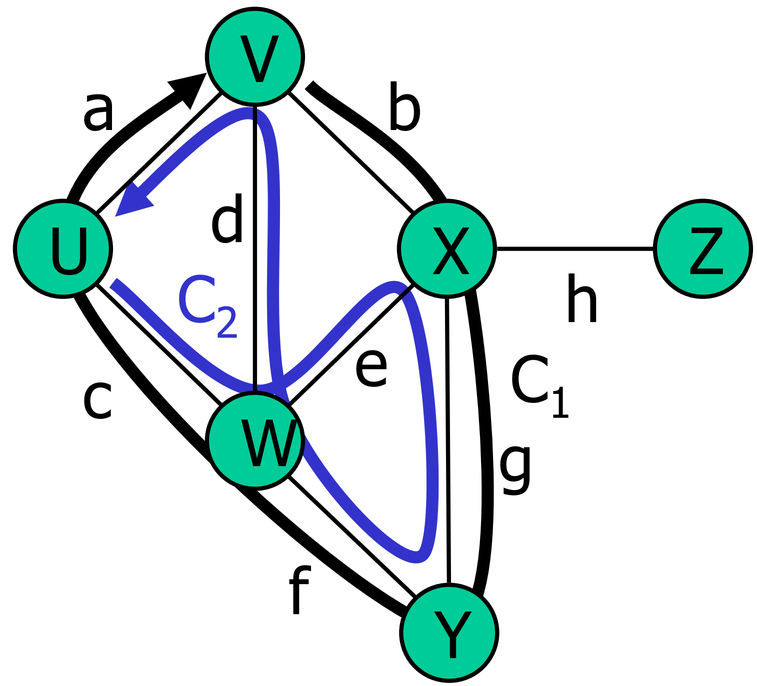
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties

Property 1

$\sum_{v} \deg(\boldsymbol{v}) = 2\boldsymbol{m}$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

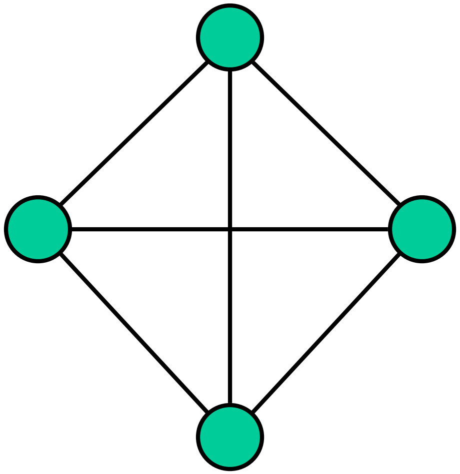
$\boldsymbol{m} \leq \boldsymbol{n}\,(\boldsymbol{n} - 1)/2$

Proof: each vertex has degree at most $(\boldsymbol{n} - 1)$

What is the bound for a directed graph?

Notation

$\boldsymbol{n}$	number of vertices
$\boldsymbol{m}$	number of edges
$\deg(\boldsymbol{v})$	degree of vertex $\boldsymbol{v}$



Example

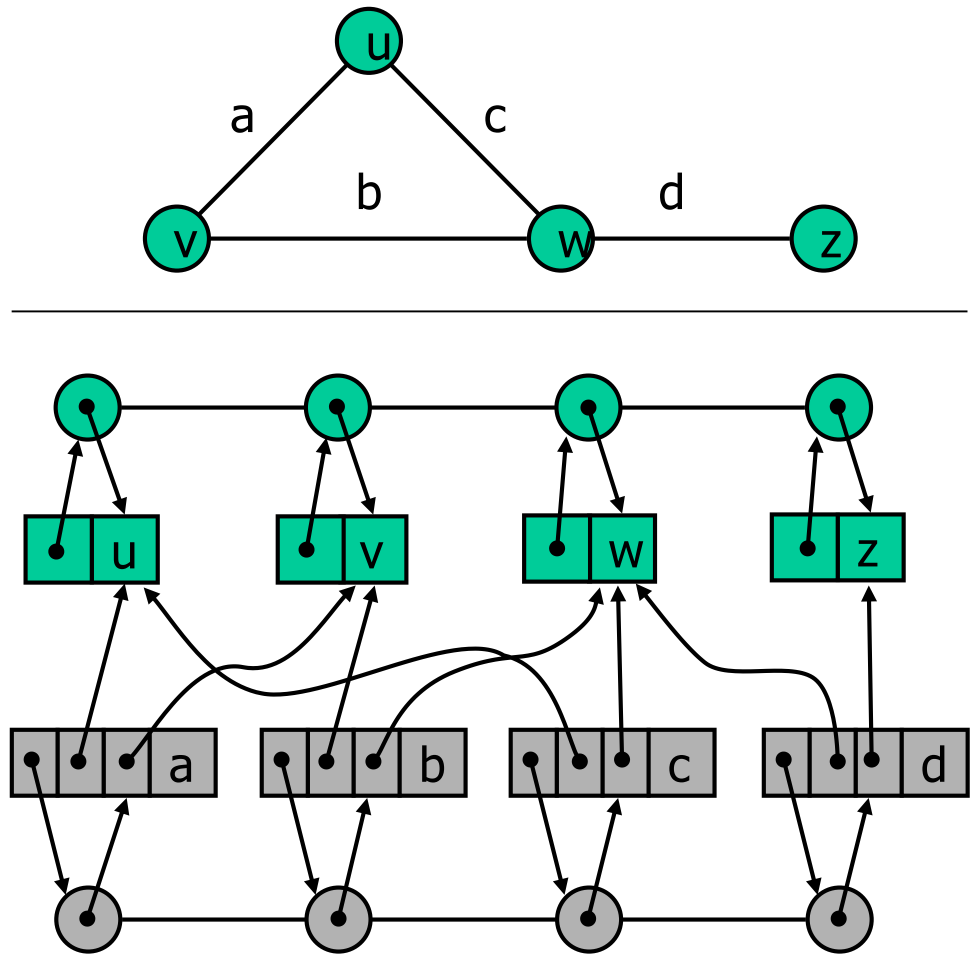
- $\boldsymbol{n} = 4$
- $\boldsymbol{m} = 6$
- $\deg(\boldsymbol{v}) = 3$

Main Methods of the Graph ADT

- Vertices and edges
  - are positions
  - store elements
- Accessor methods
  - **e.endVertices**(): a list of the two endvertices of e
  - **e.opposite**(v): the vertex opposite of v on e
  - **u.isAdjacentTo**(v): true iff u and v are adjacent
  - *v: reference to element associated with vertex v
  - *e: reference to element associated with edge e
- Update methods
  - **insertVertex**(o): insert a vertex storing element o
  - **insertEdge**(v, w, o): insert an edge (v,w) storing element o
  - **eraseVertex**(v): remove vertex v (and its incident edges)
  - **eraseEdge**(e): remove edge e
- Iterable collection methods
  - **incidentEdges**(v): list of edges incident to v
  - **vertices**(): list of all vertices in the graph
  - **edges**(): list of all edges in the graph

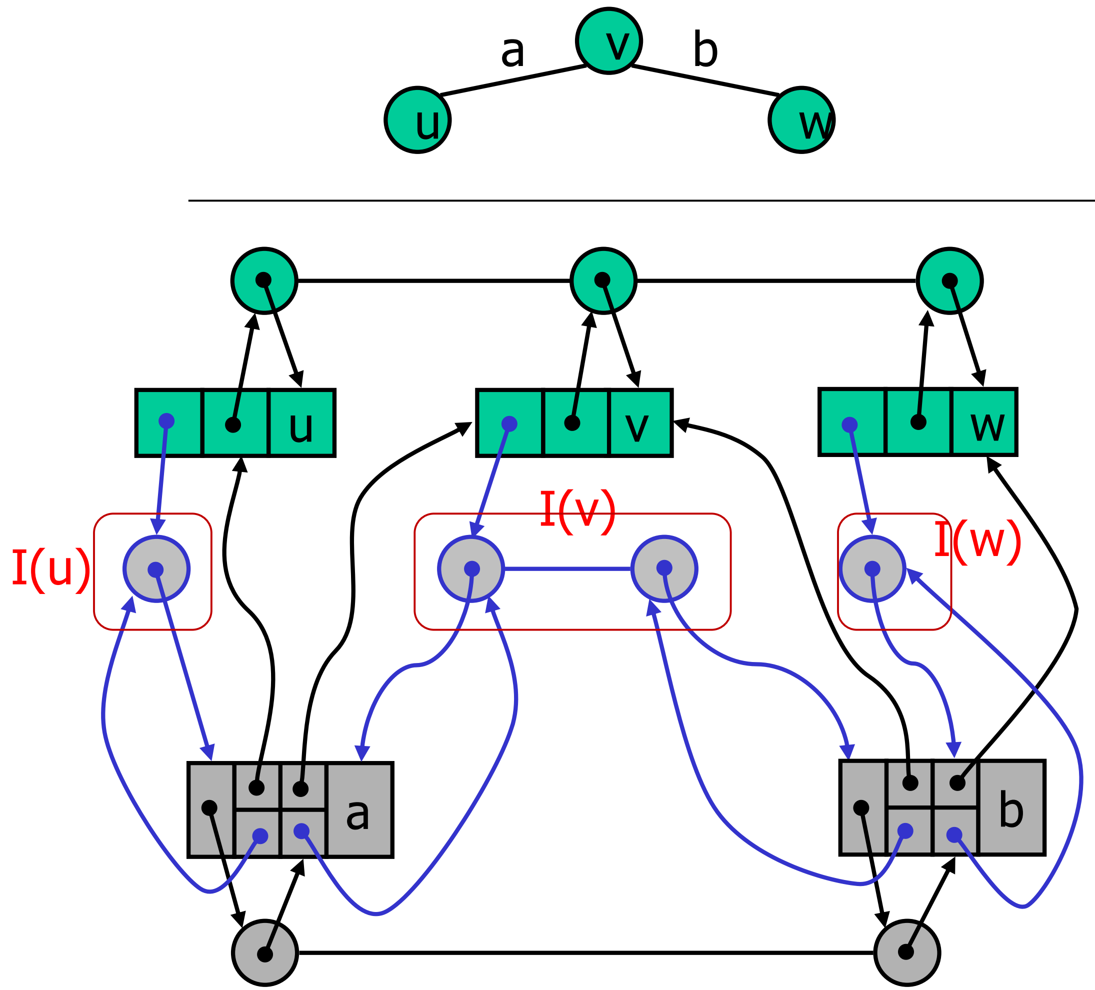
Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



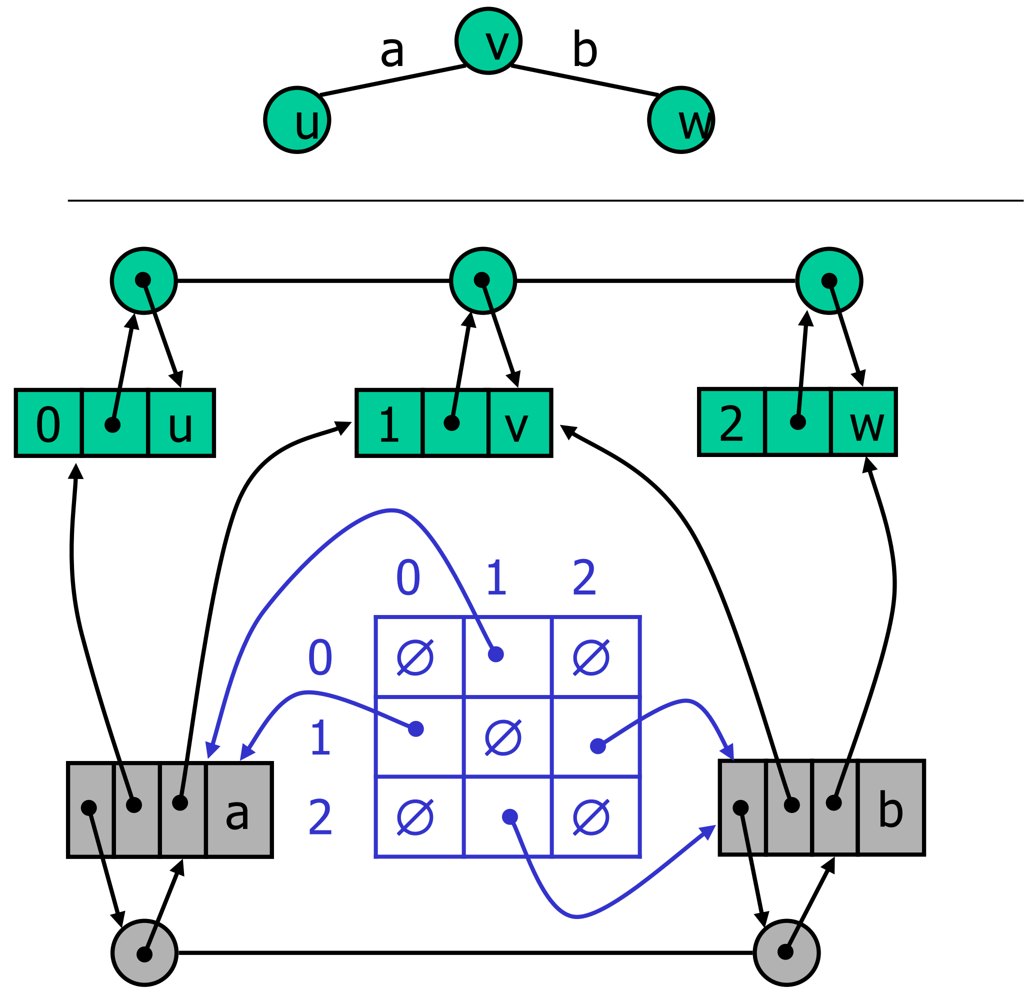
Adjacency List Structure

- Edge list structure
- Incidence sequence I(v) for each vertex v
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge

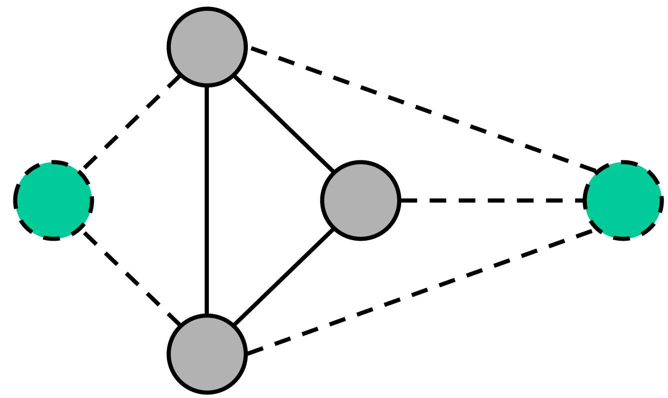


Performance

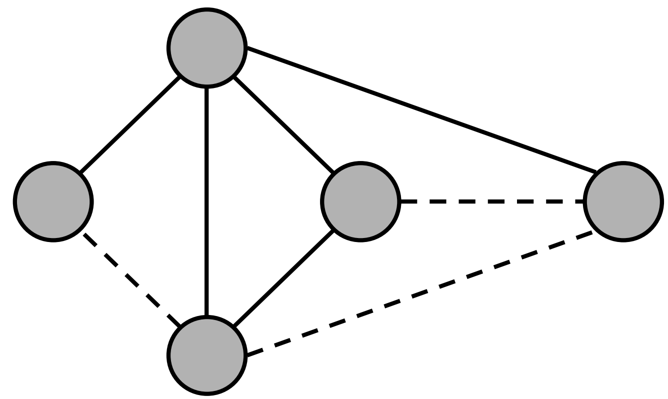
▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
v.incidentEdges()	m	$\deg(v)$	n
u.isAdjacentTo (v)	m	$\min(\deg(v), \deg(w))$	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
eraseVertex(v)	m	$\deg(v)$	n^2
eraseEdge(e)	1	1	1

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



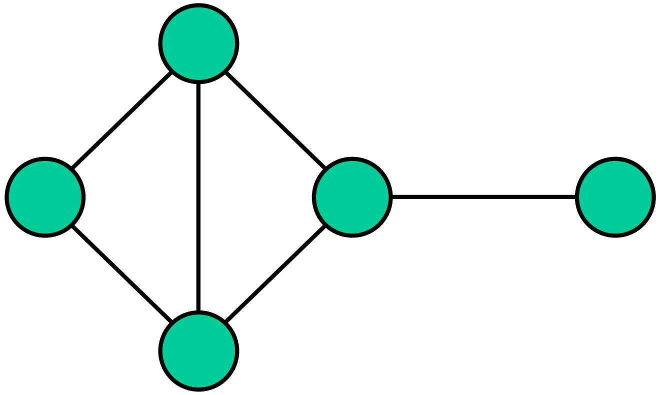
Subgraph



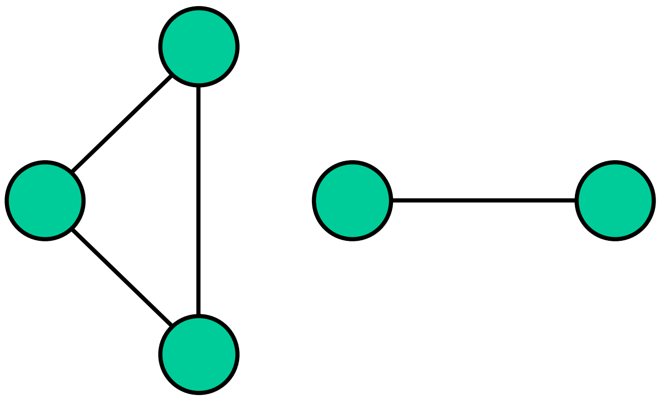
Spanning subgraph

Connectivity

- A graph is connected if 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

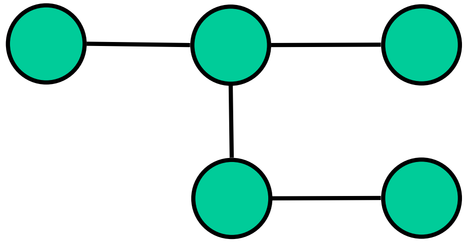


Non connected graph with two connected components

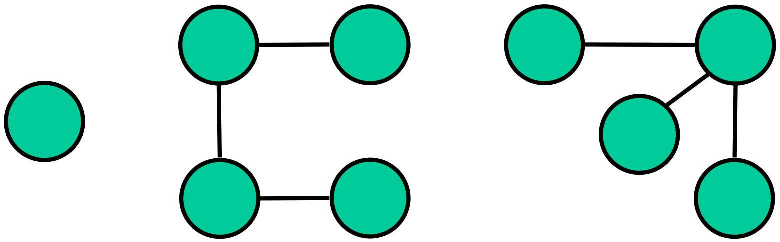
Trees and Forests

- A (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



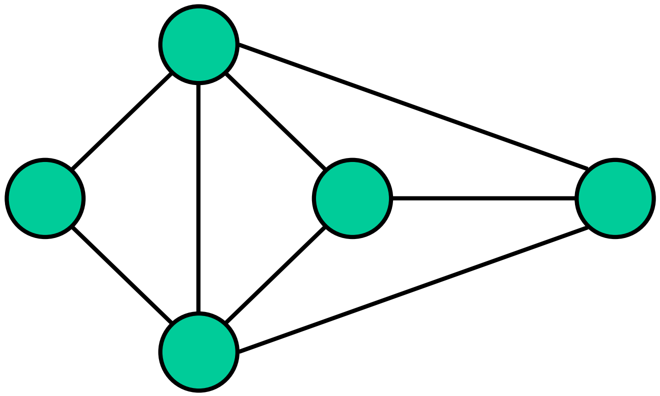
Tree



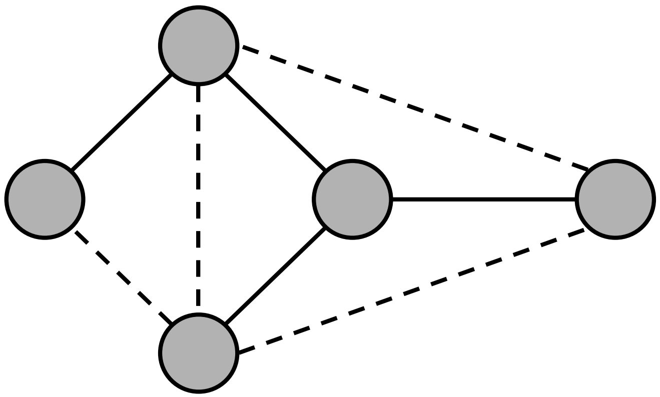
Forest

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



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 path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm

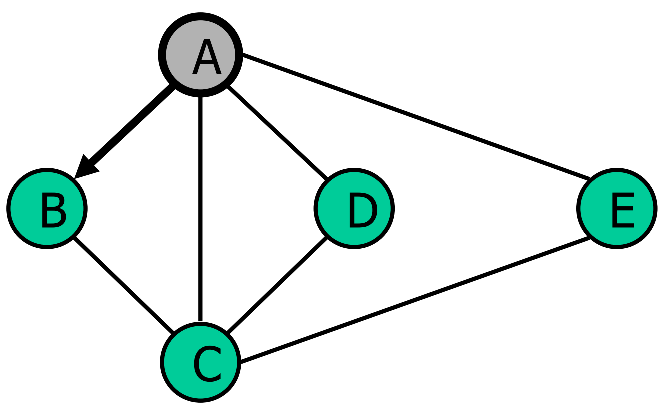
- The algorithm uses a mechanism for setting and getting “labels” of vertices and edges

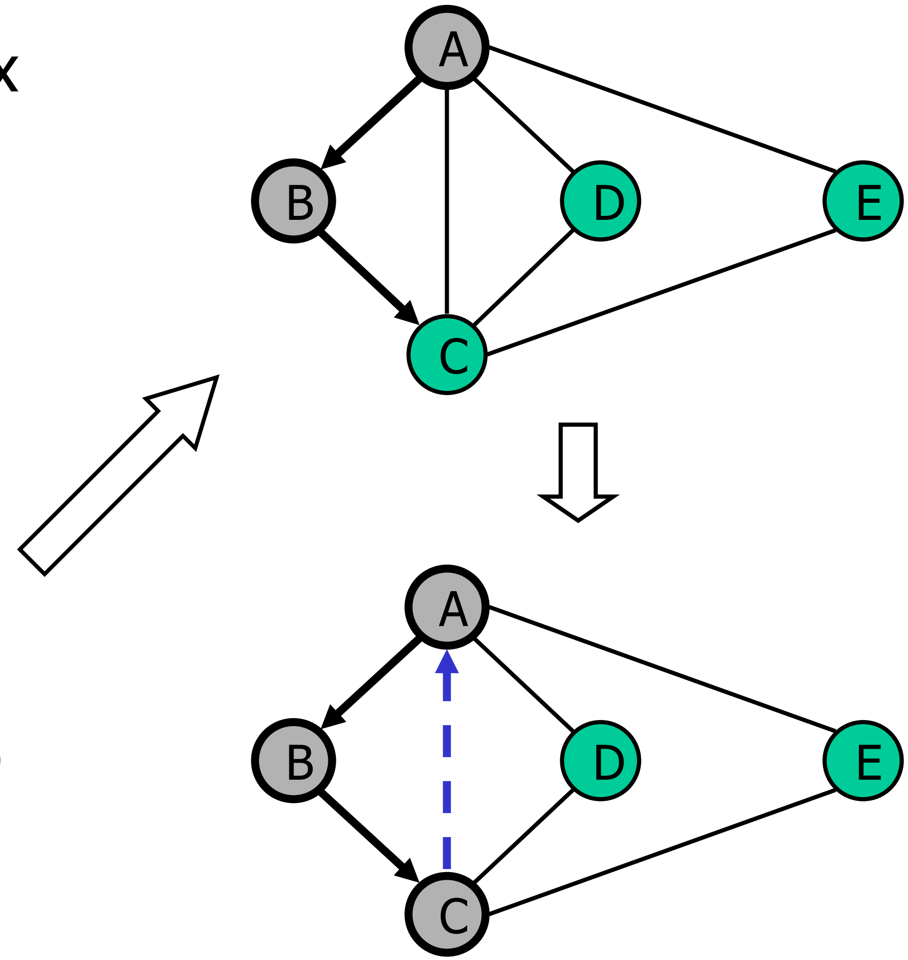
```
Algorithm DFS(G)
   Input graph G
   Output labeling of the edges of G
         as discovery edges and
         back edges
  for all u ∈ G.vertices()
   u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
   e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
   if v.getLabel() = UNEXPLORED
      DFS(G,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

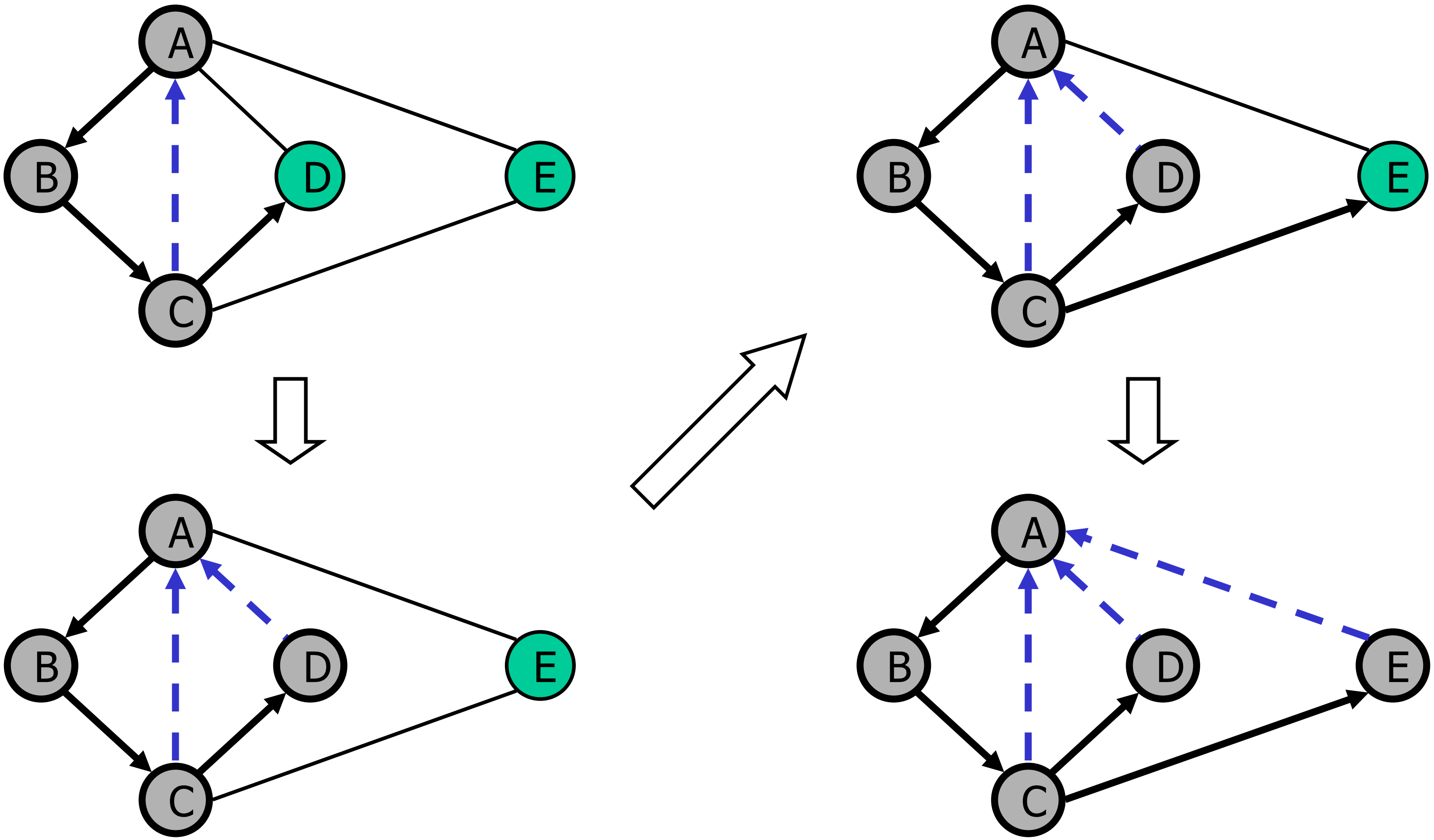
Example

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

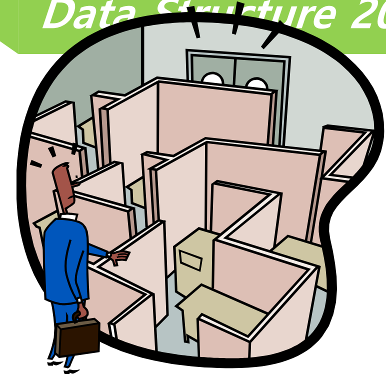




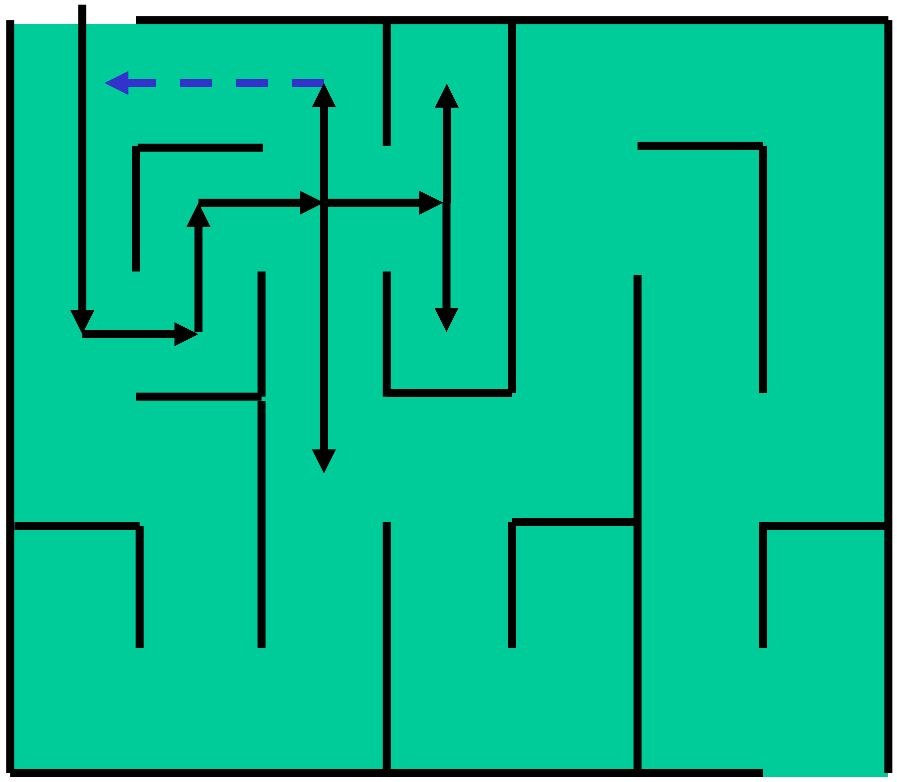
Example (cont.)



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)



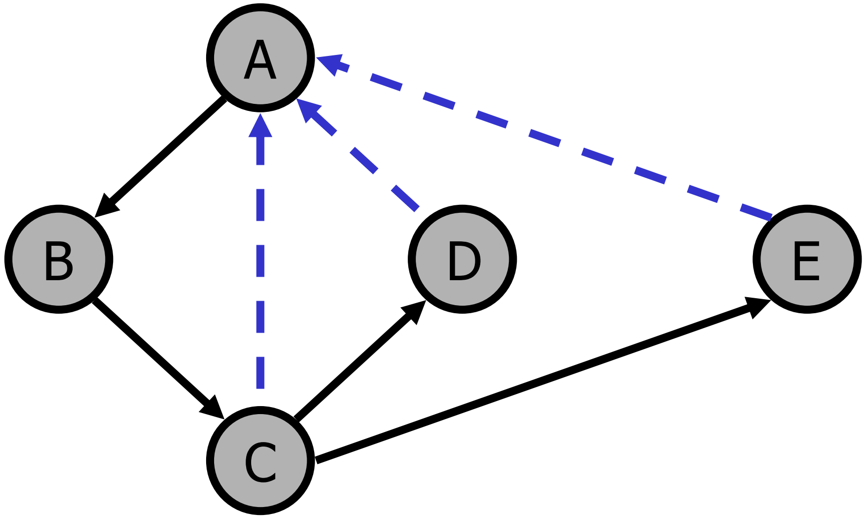
Properties of DFS

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

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

Cycle Finding

- 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)
    v.setLabel(VISITED)
    S.push(v)
    for all e ∈ v.incidentEdges()
        if e.getLabel() = UNEXPLORED
            w ← e.opposite(v)
            S.push(e)
            if w.getLabel() = UNEXPLORED
                e.setLabel(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)
```

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

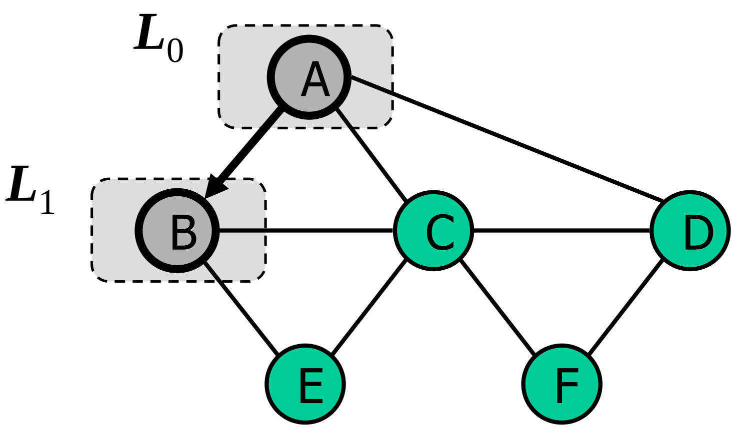
- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

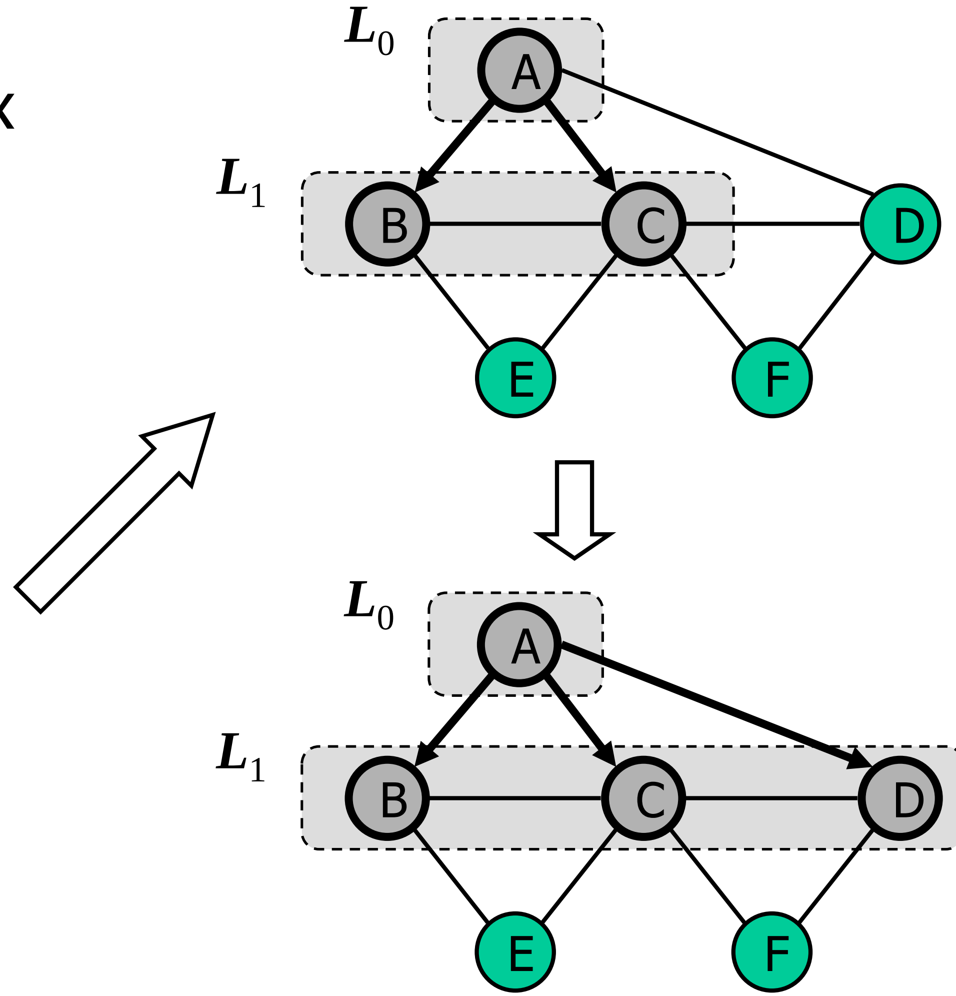
```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
    u.setLabel(UNEXPLORED)
  for all e ∈ G.edges()
    e.setLabel(UNEXPLORED)
  for all v ∈ G.vertices()
    if v.getLabel() = UNEXPLORED
        BFS(G,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

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

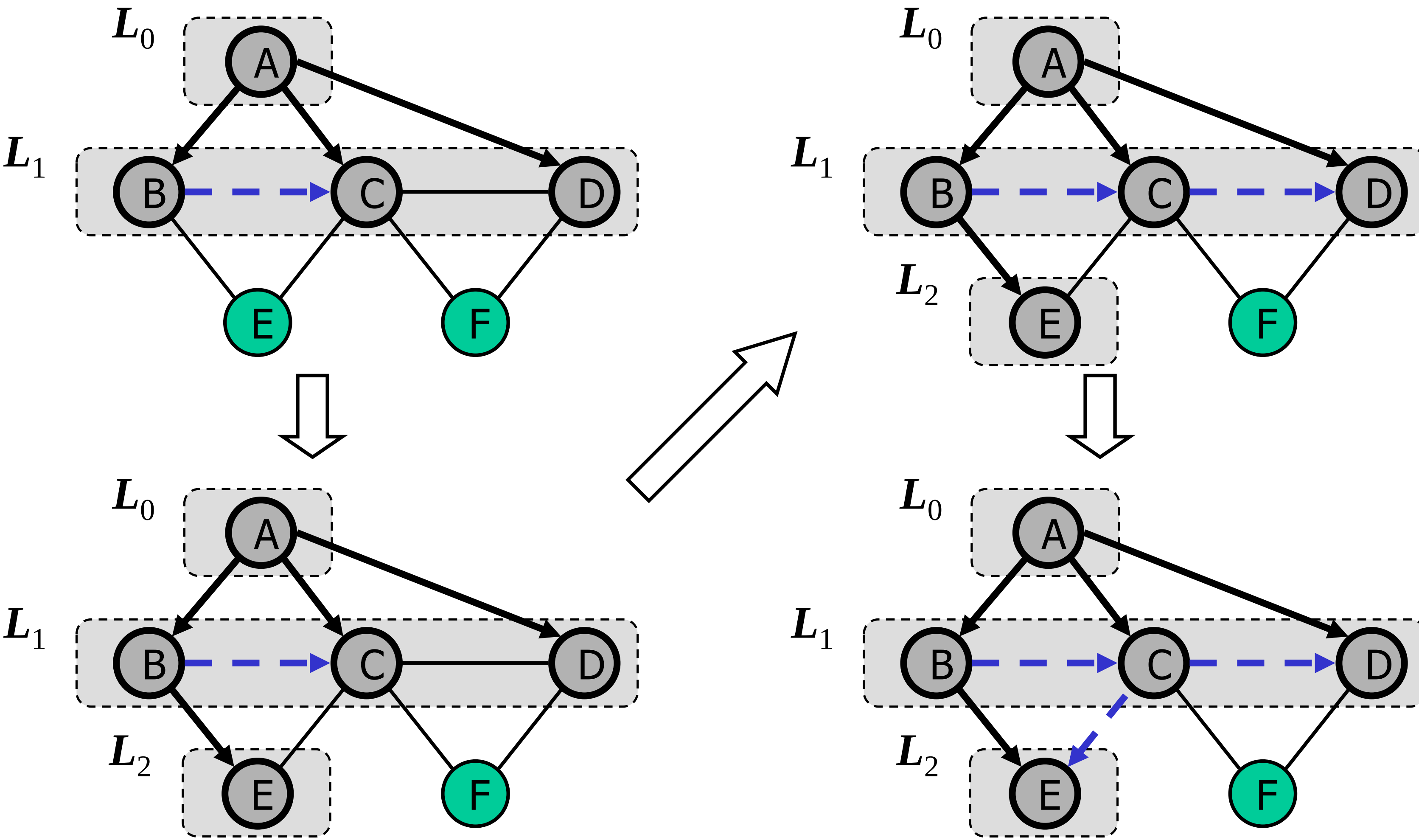
Example

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

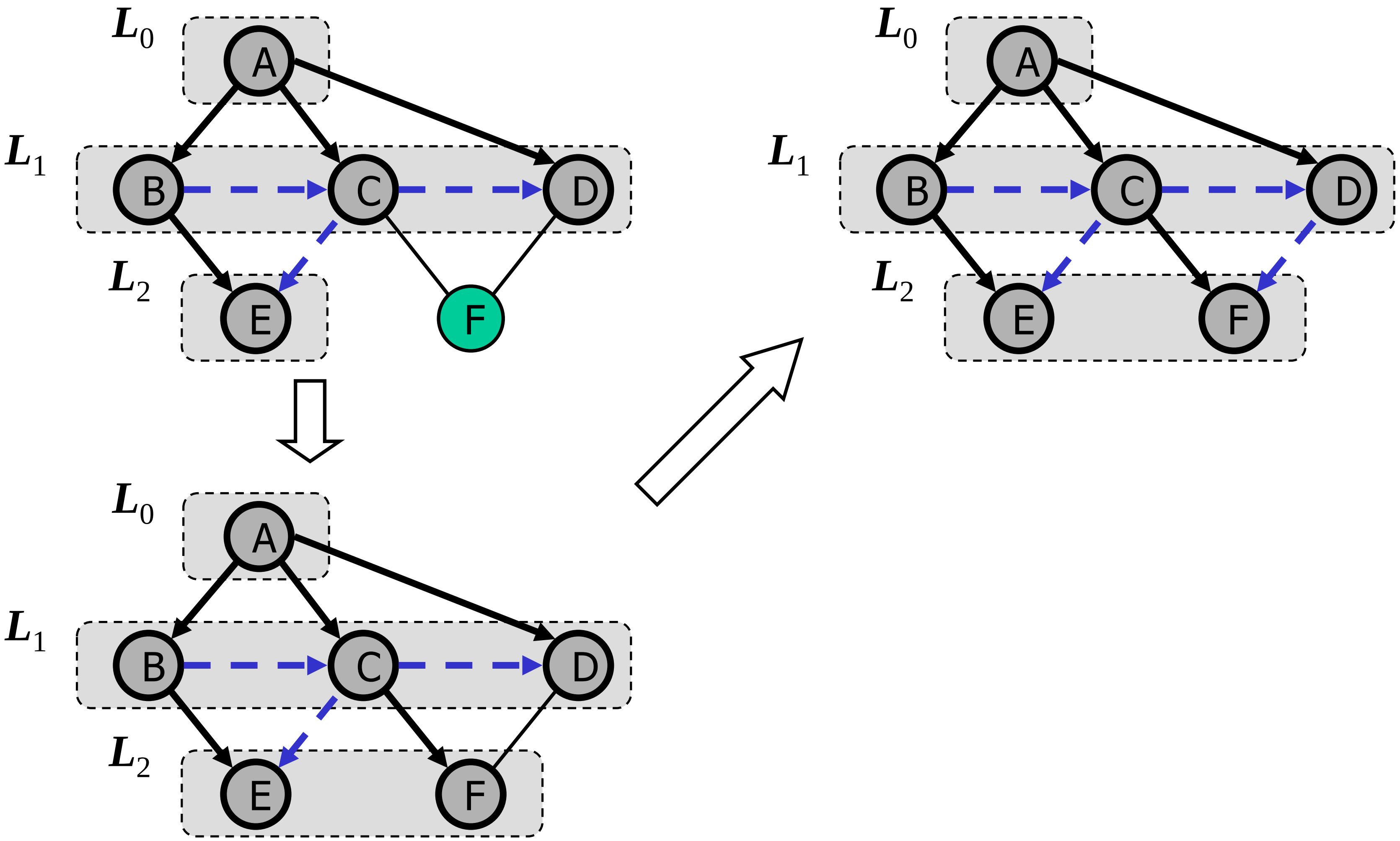




Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

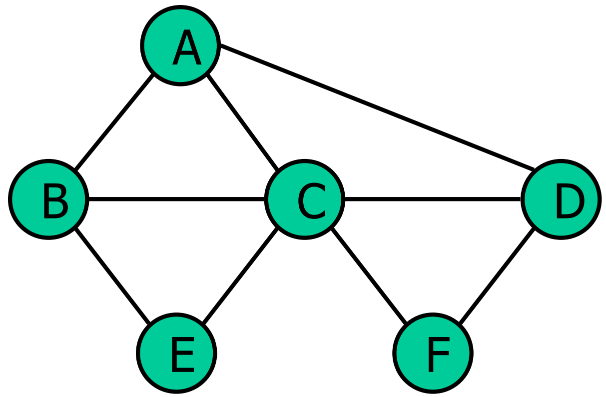
Property 2

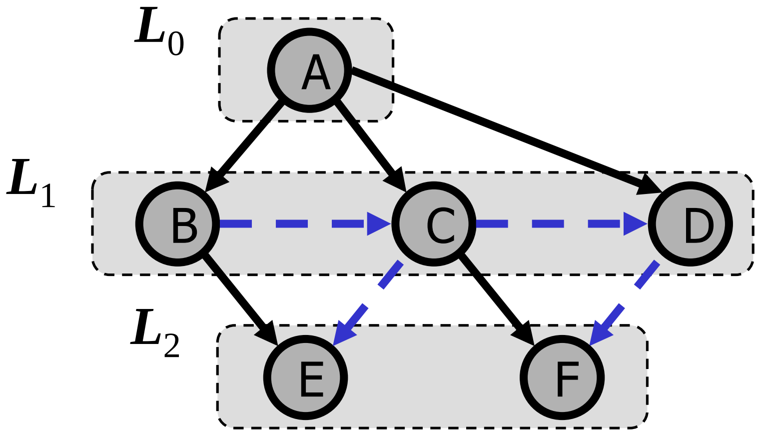
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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

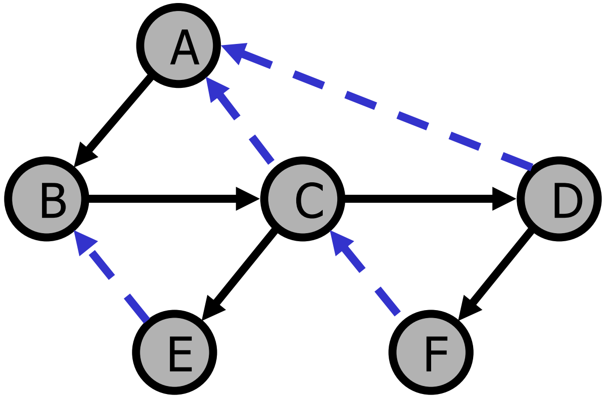
- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once 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

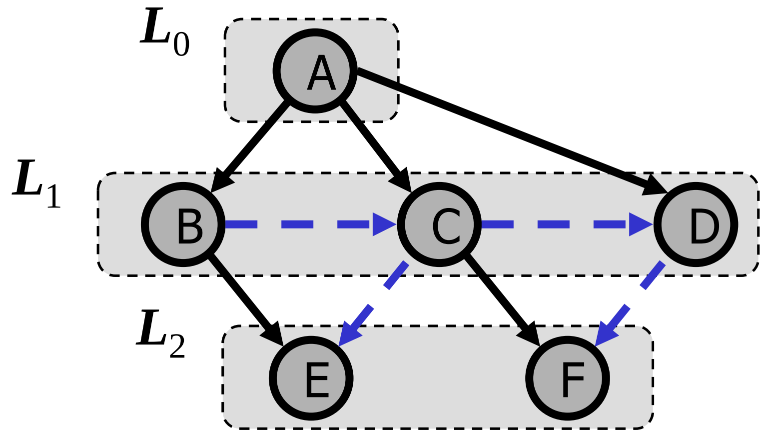
- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n+m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find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

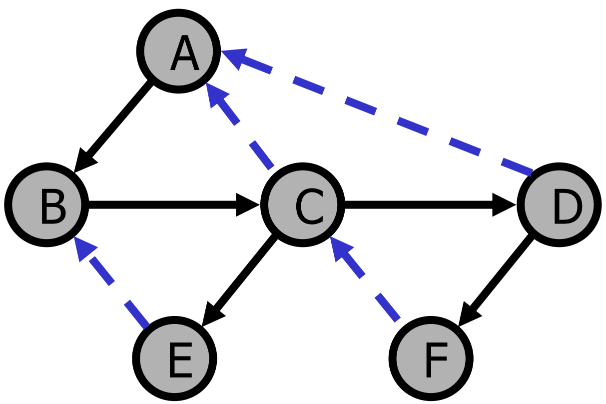


BFS

DFS vs. BFS (cont.)

Back edge (v,w)

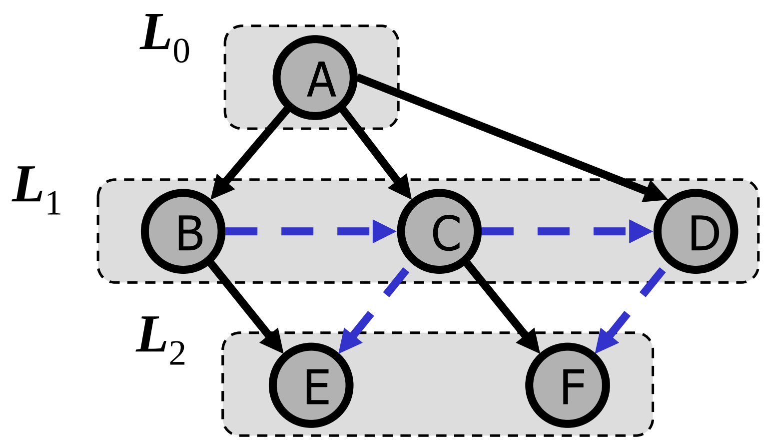
- w is an ancestor of v in the tree of discovery edges



DFS

Cross edge (v,w)

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



BFS

DFS 예제

(a) 입력그래프

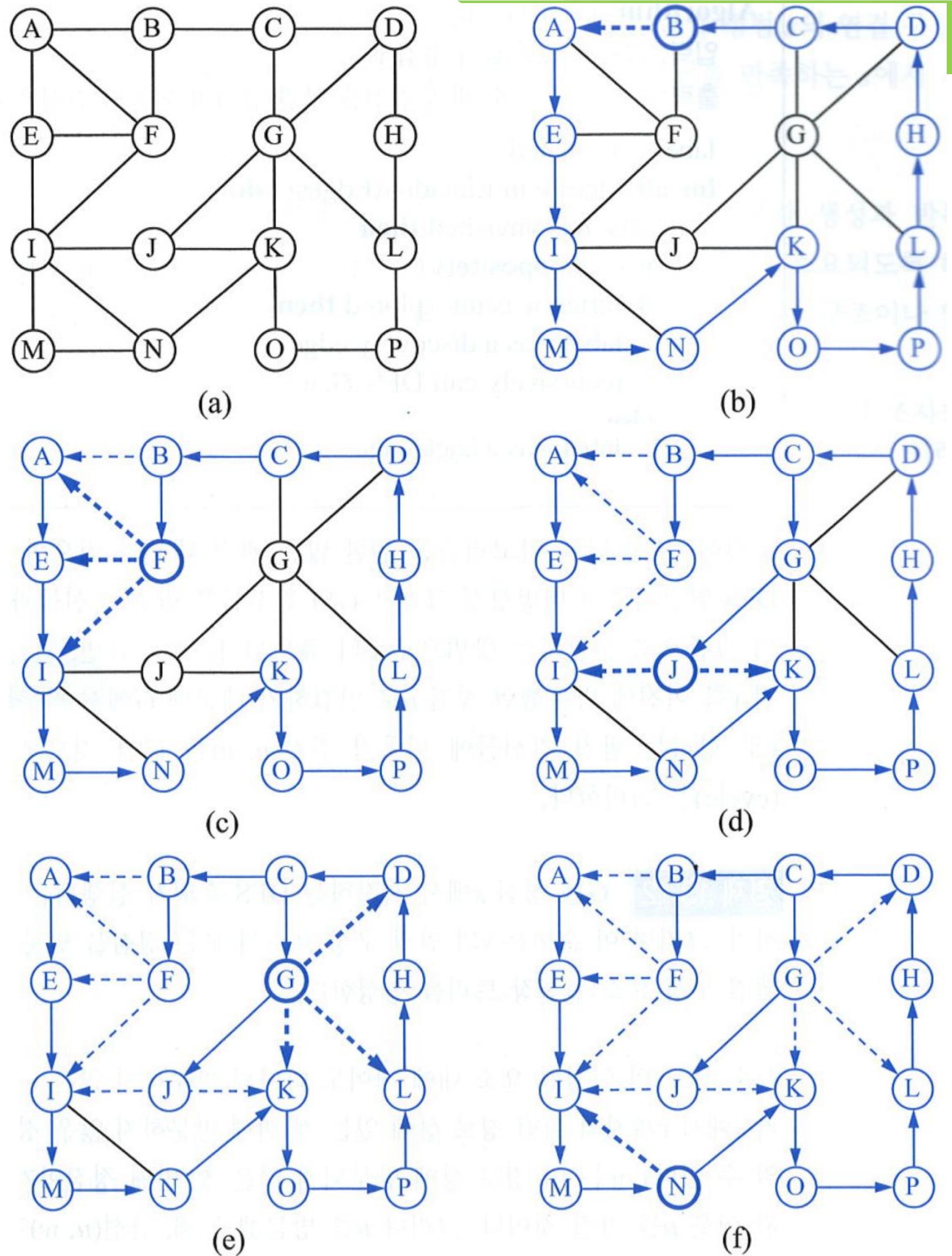
(b) A로부터 back edge (B,A)를 만날 때까지 찾아간 discovery edge의 경로

(c) 막다른 곳인 F에 도달

(d) c로 되돌아간 후 edge(C,G)로 다시 시작, 또 다른 막다른 곳인 J에 도달

(e) G로 되돌아간 후

(f) N으로 되돌아간 후



DFS 예제: DFSMain.exe

```
Inserting vertices ..
Inserted vertices: A   B   C   D   E   F   G   H   I   J   K   L   M   N   O   P
Inserting edges ..
Inserted edges:
Edge(A, B, d(10)), Edge(A, E, d(10)), Edge(A, F, d(15)), Edge(B, A, d(10)), Edge(B, C, d(10)),
Edge(B, F, d(10)), Edge(C, B, d(10)), Edge(C, D, d(10)), Edge(C, G, d(10)), Edge(D, C, d(10)),
Edge(D, G, d(15)), Edge(D, H, d(10)), Edge(E, A, d(10)), Edge(E, F, d(10)), Edge(E, I, d(10)),
Edge(F, A, d(15)), Edge(F, B, d(10)), Edge(F, E, d(10)), Edge(F, I, d(15)), Edge(G, C, d(10)),
Edge(G, D, d(15)), Edge(G, J, d(15)), Edge(G, K, d(10)), Edge(G, L, d(15)), Edge(H, D, d(10)),
Edge(H, L, d(10)), Edge(I, E, d(10)), Edge(I, F, d(15)), Edge(I, J, d(10)), Edge(I, M, d(10)),
Edge(I, N, d(15)), Edge(J, G, d(15)), Edge(J, I, d(10)), Edge(J, K, d(10)), Edge(K, G, d(10)),
Edge(K, J, d(10)), Edge(K, N, d(15)), Edge(K, O, d(10)), Edge(L, G, d(15)), Edge(L, H, d(10)),
Edge(L, P, d(10)), Edge(M, I, d(10)), Edge(M, N, d(10)), Edge(N, I, d(15)), Edge(N, K, d(15)),
Edge(N, M, d(10)), Edge(O, K, d(10)), Edge(O, P, d(10)), Edge(P, L, d(10)), Edge(P, O, d(10)),

Print out Graph based on Adjacency List ..
 A  Edge(A, B, d(10))  Edge(A, E, d(10))  Edge(A, F, d(15))
 B  Edge(B, A, d(10))  Edge(B, C, d(10))  Edge(B, F, d(10))
 C  Edge(C, B, d(10))  Edge(C, D, d(10))  Edge(C, G, d(10))
 D  Edge(D, C, d(10))  Edge(D, G, d(15))  Edge(D, H, d(10))
 E  Edge(E, A, d(10))  Edge(E, F, d(10))  Edge(E, I, d(10))
 F  Edge(F, A, d(15))  Edge(F, B, d(10))  Edge(F, E, d(10))  Edge(F, I, d(15))
 G  Edge(G, C, d(10))  Edge(G, D, d(15))  Edge(G, J, d(15))  Edge(G, K, d(10))  Edge(G, L, d(15))
 H  Edge(H, D, d(10))  Edge(H, L, d(10))
 I  Edge(I, E, d(10))  Edge(I, F, d(15))  Edge(I, J, d(10))  Edge(I, M, d(10))  Edge(I, N, d(15))
 J  Edge(J, G, d(15))  Edge(J, I, d(10))  Edge(J, K, d(10))
 K  Edge(K, G, d(10))  Edge(K, J, d(10))  Edge(K, N, d(15))  Edge(K, O, d(10))
 L  Edge(L, G, d(15))  Edge(L, H, d(10))  Edge(L, P, d(10))
 M  Edge(M, I, d(10))  Edge(M, N, d(10))
 N  Edge(N, I, d(15))  Edge(N, K, d(15))  Edge(N, M, d(10))
 O  Edge(O, K, d(10))  Edge(O, P, d(10))
 P  Edge(P, L, d(10))  Edge(P, O, d(10))
```

DFS 예제: DFSMain.exe

```
Testing dfsGraph...
Connectivity of graph:
     |   A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P
-----+-------------------------------------------------------------------------------
   A |  +00   10  +00  +00   10   15  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00
   B |   10  +00   10  +00  +00   10  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00
   C |  +00   10  +00   10  +00  +00   10  +00  +00  +00  +00  +00  +00  +00  +00  +00
   D |  +00  +00   10  +00  +00  +00   15   10  +00  +00  +00  +00  +00  +00  +00  +00
   E |   10  +00  +00  +00  +00   10  +00  +00   10  +00  +00  +00  +00  +00  +00  +00
   F |   15   10  +00  +00   10  +00  +00  +00   15  +00  +00  +00  +00  +00  +00  +00
   G |  +00  +00   10   15  +00  +00  +00  +00  +00   15   10   15  +00  +00  +00  +00
   H |  +00  +00  +00   10  +00  +00  +00  +00  +00  +00  +00   10  +00  +00  +00  +00
   I |  +00  +00  +00  +00   10   15  +00  +00  +00   10  +00  +00   10   15  +00  +00
   J |  +00  +00  +00  +00  +00  +00   15  +00   10  +00   10  +00  +00  +00  +00  +00
   K |  +00  +00  +00  +00  +00  +00   10  +00  +00   10  +00  +00  +00   15   10  +00
   L |  +00  +00  +00  +00  +00  +00   15   10  +00  +00  +00  +00  +00  +00  +00   10
   M |  +00  +00  +00  +00  +00  +00  +00  +00   10  +00  +00  +00  +00   10  +00  +00
   N |  +00  +00  +00  +00  +00  +00  +00  +00   15  +00   15  +00   10  +00  +00  +00
   O |  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00   10  +00  +00  +00  +00   10
   P |  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00  +00   10  +00  +00   10  +00
Path (A => P) : A    B    C    D    G    J    I    M    N    K    O    P
Path (P => A) : P    L    G    C    B    A
```

Graph.h

```
class Graph { // Graph based on Adjacency Matrix
 public:
  class Vertex {
  private:
    string name;
    int ID;
    VertexStatus vtxStatus;
  }; // end class Vertex
  typedef std::list<Vertex> VtxList;
  class Edge {
  private:
    Vertex vrtx_1;
    Vertex vrtx_2;
    Vertex* pVrtx_1;
    Vertex* pVrtx_2;
    int distance;
    EdgeStatus edgeStatus;
  }; // end class Edge
 public:
  typedef std::list<Edge> EdgeList;
  typedef std::list<Vertex>::iterator VtxItor;
  typedef std::list<Edge>::iterator EdgeItor;
 private:
  Vertex* pVrtxArray;
  EdgeList* pAdjLstArray;
  int num_vertices;
};
```

DFSMain.cpp & DFS.cpp

```
int main() {   // Topology of Figure 13.6 (p. 608)
  Vertex v[NUM_NODES] = {    Vertex("A",  0, UN_VISITED), … };
  Graph::Edge edges[NUM_EDGES] = { Edge(v[0], v[1], 10),    Edge(v[1], v[0], 10), … };
  Graph simpleGraph(NUM_NODES);
  for (int i=0; i<NUM_NODES; i++) simpleGraph.insertVertex(v[i]);
  for (int i=0; i<NUM_EDGES; i++) simpleGraph.insertEdge(edges[i]);
  DepthFirstSearch dfsGraph(simpleGraph);
  dfsGraph.showConnectivity();

  VtxList path;
  dfsGraph.findPath(v[0], v[15], path);
  cout << "Path (" << v[0] << " => " << v[15] << ") : ";
  for (VtxItor vItor = path.begin(); vItor != path.end(); ++vItor)
    cout << *vItor << "    ";
  cout << endl;

  dfsGraph.findPath(v[15], v[0], path);
  cout << "Path (" << v[15] << " => " << v[0] << ") : ";
  for (VtxItor vItor = path.begin(); vItor != path.end(); ++vItor)
    cout << *vItor << "    ";
  cout << endl;
  return 0;
}
```

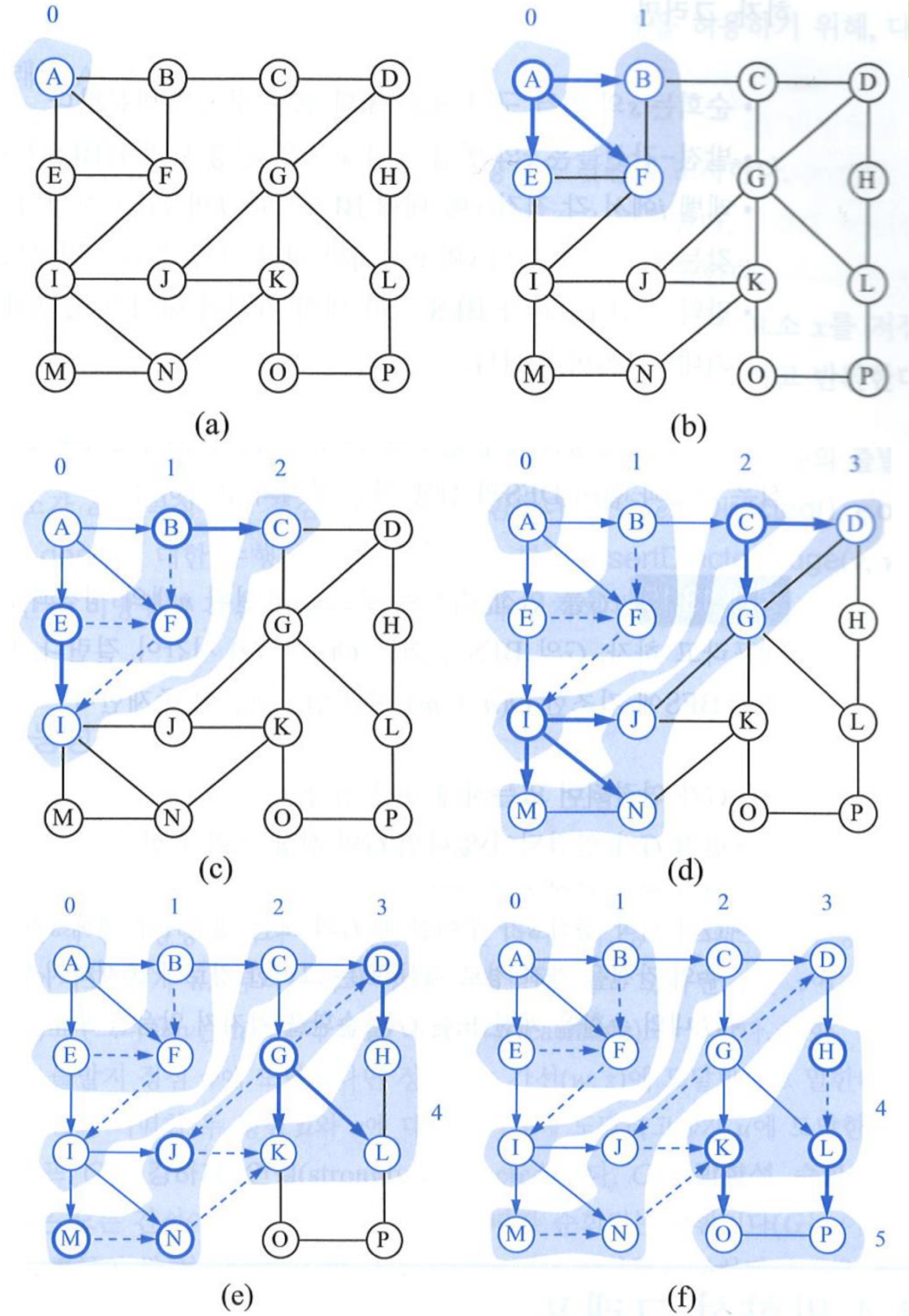
```
void DepthFirstSearch::findPath(Vertex &start, Vertex &target, VertexList& path) {
  initialize();   path.clear();      path.push_back(start);
  dfsTraversal(start, target, path);
}
```

DFS.cpp

```
void DepthFirstSearch::dfsTraversal(Vertex& v, Vertex& target, VertexList& path) {
  visit(v);
  if (v == target) {     done = true;     return;    }
  EdgeList incidentEdges;
  incidentEdges.clear();
  graph.incidentEdges(v, incidentEdges);
  EdgeItor pe = incidentEdges.begin();
  while (!isDone() && pe != incidentEdges.end()) {
    Edge e = *pe++;
    EdgeStatus eStat = getEdgeStatus(e);
    if (eStat == EDGE_UN_VISITED) {
      visit(e);
      Vertex w = e.opposite(v);
      if (!isVisited(w)) {
          path.push_back(w);
          setEdgeStatus(e, DISCOVERY);
          if (!isDone()) {
            dfsTraversal(w, target, path);
            if (!isDone()) {
              Vertex last_pushed = path.back(); // for debugging
              path.pop_back();
            }
          }
      } else {
          setEdgeStatus(e, BACK);
      }
    }
  } // end of while()
}
```

BFS 예제

(a) 입력그래프
(b) 레벨 1의 발견
(c) 레벨 2의 발견
(d) 레벨 3의 발견
(e) 레벨 4의 발견
(f) 레벨 5의 발견



BFS 예제: BFSMain.exe

```
Testing Breadth First Search
    |   A    B    C    D    E    F    G    H    I    J    K    L    M    N    O    P
----+-------------------------------------------------------------------------------
 A  |    0   10  +oo  +oo   10   15  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
 B  |   10    0   10  +oo  +oo   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
 C  |  +oo   10    0   10  +oo  +oo   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
 D  |  +oo  +oo   10    0  +oo  +oo   15   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo
 E  |   10  +oo  +oo  +oo    0   10  +oo  +oo   10  +oo  +oo  +oo  +oo  +oo  +oo  +oo
 F  |   15   10  +oo  +oo   10    0  +oo  +oo   15  +oo  +oo  +oo  +oo  +oo  +oo  +oo
 G  |  +oo  +oo   10   15  +oo  +oo    0  +oo  +oo   15   10   15  +oo  +oo  +oo  +oo
 H  |  +oo  +oo  +oo   10  +oo  +oo  +oo    0  +oo  +oo  +oo   10  +oo  +oo  +oo  +oo
 I  |  +oo  +oo  +oo  +oo   10   15  +oo  +oo    0   10  +oo  +oo   10   15  +oo  +oo
 J  |  +oo  +oo  +oo  +oo  +oo  +oo   15  +oo   10    0   10  +oo  +oo  +oo  +oo  +oo
 K  |  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo   10    0  +oo  +oo   15   10  +oo
 L  |  +oo  +oo  +oo  +oo  +oo  +oo   15   10  +oo  +oo  +oo    0  +oo  +oo  +oo   10
 M  |  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo  +oo    0   10  +oo  +oo
 N  |  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   15  +oo   15  +oo   10    0  +oo  +oo
 O  |  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo  +oo    0   10
 P  |  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo  +oo   10  +oo  +oo   10    0

round [ 1]  A    0, B   10, C   20, D  +oo, E   10, F   15, G  +oo, H  +oo, I  +oo, J  +oo, K  +oo, L  +oo, M  +oo, N  +oo, O  +oo, P  +oo,
round [ 2]  A    0, B   10, C   20, D  +oo, E   10, F   15, G  +oo, H  +oo, I   20, J  +oo, K  +oo, L  +oo, M  +oo, N  +oo, O  +oo, P  +oo,
round [ 3]  A    0, B   10, C   20, D  +oo, E   10, F   15, G  +oo, H  +oo, I   20, J  +oo, K  +oo, L  +oo, M  +oo, N  +oo, O  +oo, P  +oo,
round [ 4]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H  +oo, I   20, J  +oo, K  +oo, L  +oo, M  +oo, N  +oo, O  +oo, P  +oo,
round [ 5]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H  +oo, I   20, J   30, K  +oo, L  +oo, M   30, N   35, O  +oo, P  +oo,
round [ 6]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K  +oo, L  +oo, M   30, N   35, O  +oo, P  +oo,
round [ 7]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O  +oo, P  +oo,
round [ 8]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O  +oo, P  +oo,
round [ 9]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O  +oo, P  +oo,
round [10]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O  +oo, P  +oo,
round [11]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O  +oo, P  +oo,
round [12]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O   50, P  +oo,
round [13]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O   50, P   55,
round [14]  A    0, B   10, C   20, D   30, E   10, F   15, G   30, H   40, I   20, J   30, K   40, L   45, M   30, N   35, O   50, P   55,
round [15]  reached to the target node !!
Least Cost = 55
Path found by BFS (shortest) from A to P : A    B    C    G    L    P
```

BFSMain.cpp & BFS.cpp

```
int main() {   // Topology of Figure 13.6 (p. 608)
  Vertex v[NUM_NODES] = {    Vertex("A",  0, UN_VISITED), … };
  Graph::Edge edges[NUM_EDGES] = { Edge(v[0], v[1], 10),    Edge(v[1], v[0], 10), … };
  Graph simpleGraph(NUM_NODES);
  for (int i=0; i<NUM_NODES; i++) simpleGraph.insertVertex(v[i]);
  for (int i=0; i<NUM_EDGES; i++) simpleGraph.insertEdge(edges[i]);

  BreadthFirstSearch bfsGraph(simpleGraph);
  VtxList path;
  bfsGraph.findShortestPath(v[0], v[15], path);
  cout << "Path found by BFS (shortest) from " << v[0] << " to " << v[15] << " : ";
  for (VtxItor vItor = path.begin(); vItor != path.end(); ++vItor)
    cout << *vItor << "    ";
  cout << endl;
  return 0;
}
```

```
void BreadthFirstSearch::findShortestPath(Vertex &s, Vertex &target, VertexList& path) {
  initialize();
  path.clear();

  start = s;
  initDistMtrx();
  printDistMtrx();
  bfsTraversal(start, target, path);
}
```

BFS.cpp

```
void BreadthFirstSearch::bfsTraversal(Vertex& s, Vertex& target, VertexList& path) {
  int** ppDistMtrx;  int* pLeastCost; int* pPrev;
  int num_nodes, num_selected, minID, minCost;
  BFS_PROCESS_STATUS* pBFS_Process_Stat;

  Vertex* pVrtxArray;
  Vertex vrtx, *pPrevVrtx, v;
  Edge e;
  int start_vrtxid, target_vrtxid, curVrtx_ID, vrtxID;
  EdgeList* pAdjLstArray;

  pVrtxArray = graph.getpVrtxArray();
  pAdjLstArray = graph.getpAdjLstArray();
  start_vrtxid = start.getID();
  target_vrtxid = target.getID();

  num_nodes = graph.getNumVertices();
  ppDistMtrx = getppDistMtrx();

  pLeastCost = new int[num_nodes];
  pPrev = new int[num_nodes];
  pBFS_Process_Stat = new BFS_PROCESS_STATUS[num_nodes];
  // initialize L(n) = w(start, n);
  for (int i=0; i< num_nodes; i++) {
    pLeastCost[i] = ppDistMtrx[start_vrtxid][i];
    pPrev[i] = start_vrtxid;
    pBFS_Process_Stat[i] = NOT_SELECTED;
  }
  pBFS_Process_Stat[start_vrtxid] = SELECTED;
  num_selected = 1;
  int round = 0;
```

BFS.cpp

```
while (num_selected < num_nodes) {
  round++;
  cout << "=== round " << round << " ==== " << endl;
  minID = -1;
  minCost = PLUS_INF;
  for (int i=0; i<num_nodes; i++)  {  // find current node with LeastCost
    if ((pLeastCost[i] < minCost) && (pBFS_Process_Stat[i] != SELECTED)) {
        minID = i; minCost = pLeastCost[i];
    }
  }
  if (minID == -1) {
    cout <<  "No Path!!"  << endl; break;
  } else {
    pBFS_Process_Stat[minID] = SELECTED;
    num_selected++;
    if (minID == target_vrtxid) {
        cout << "reached to the target node !!" << endl;
        cout << "Least Cost = " << minCost << endl;
        vrtxID = minID;
        do {
          vrtx = pVrtxArray[vrtxID];
          path.push_front(vrtx);
          vrtxID = pPrev[vrtxID];
        } while (vrtxID != start_vrtxid);
        vrtx = pVrtxArray[vrtxID];
        path.push_front(vrtx); // start node
        break;
    }
  }
```

BFS.cpp

```
    int pLS, ppDistMtrx_i;
    for (int i=0; i<num_nodes; i++) {
      pLS = pLeastCost[i];
      ppDistMtrx_i = ppDistMtrx[minID][i];

      if ( (pBFS_Process_Stat[i] != SELECTED) &&
              (pLeastCost[i] > (pLeastCost[minID] + ppDistMtrx[minID][i]))) {
           pPrev[i] = minID;
           pLeastCost[i] = pLeastCost[minID] + ppDistMtrx[minID][i];

      }
    } // end of for()
  }  // end of while()
} // end of bfsTraversal
```

감사합니다!