GRAPHS

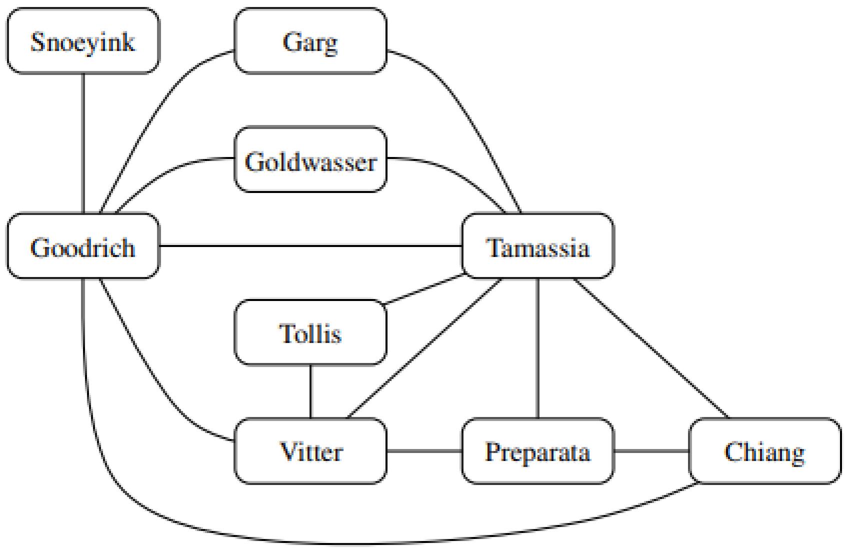
School of Artificial Intelligence

PREVIOUSLY ON GRAPH

- Graphs: relationships between pairs of objects: vertices together with a collection of connections between them (edges).
  - Mapping, transportation, computer networks electrical engineering
- Definition:
  - Set V of **vertices** and a collection E of pairs of vertices from V, called **edges**
- Edges
  - Directed: an edge (u, v) is directed from u to v if the pair (u, v) is ordered
  - Undirected: an edge (u, v) is not ordered, sometimes denoted as {u, v}
- Undirected graph: the edges are all undirected
- Directed graph (digraph): edges are all directed
- Mixed graph: directed edges + undirected edges
- Undirected graph => directed graph

GRAPHS

- Example
  - Collaboration graph (undirected)
  - Class relationships within an Object-Oriented program (directed)
  - City map (mixed graph)
  - Electric wiring/Schematics
    - Directed/undirected

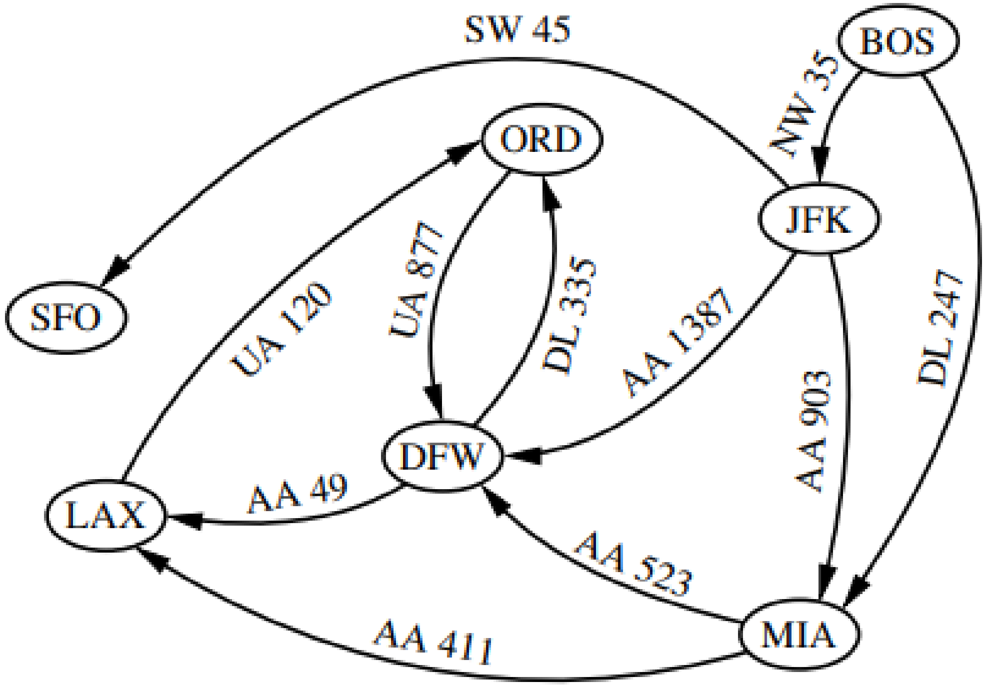


GRAPHS

- **End vertices** (endpoints)
  - Two vertices joined by an edge
  - Directed edge: origin -> destination
- **Adjacent**(相邻的) vertices: u and v are adjacent if there is an edge whose end vertices are u and v
- **Incident**(入射): an edge is incident to a vertex if the vertex is one of the edge's endpoints
- **Outgoing**(输出) edges of a vertex: directed edges whose origin is the vertex
- **Incoming** (输入) edges of a vertex: directed edges whose destination is the vertex
- **Degree** (度) of a vertex: # of incident edges of v
- **In-degree** (入度) : # of incoming edges of v
- **Out-degree** (出度) : # of outgoing edges of v

GRAPHS

- End vertices (endpoints)
  - Two vertices joined by an edge
  - Directed edge: origin -> destination
- Adjacent vertices: u and v are adjacent if there is an edge whose end vertices are u and v
- Incident: an edge is incident to a vertex if the vertex is one of the edge's endpoints
- Outgoing edges of a vertex: directed edges whose origin is the vertex
- Incoming edges of a vertex: directed edges whose destination is the vertex
- Degree of a vertex: # of incident edges of v
- In-degree: # of incoming edges of v
- Out-degree: # of outgoing edges of v

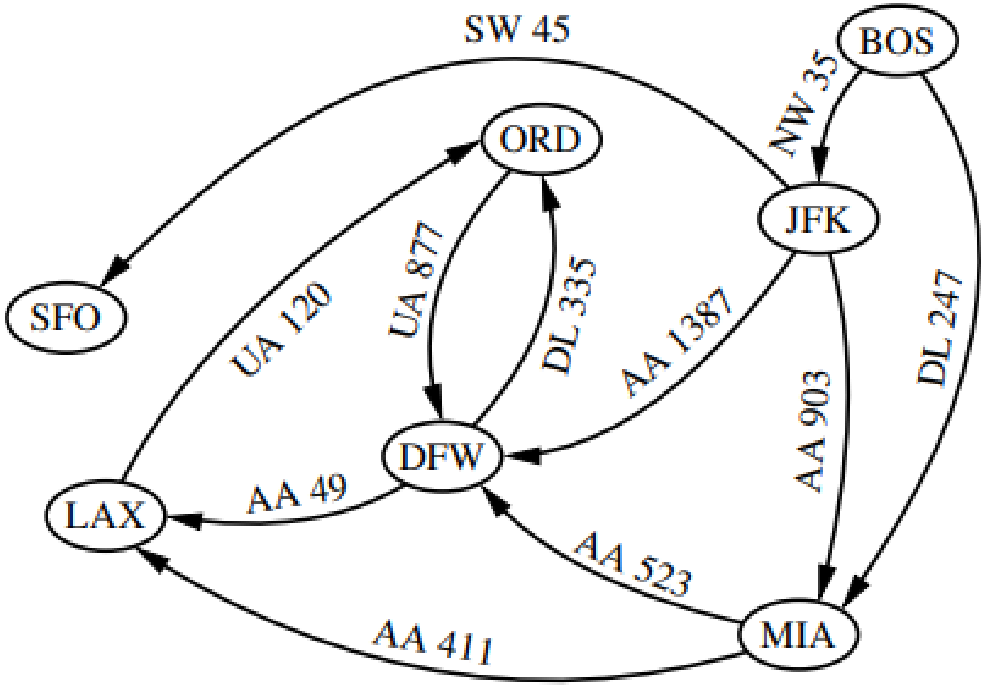


GRAPHS

- Definition:
  - Set V of **vertices** and a collection E of pairs of vertices from V, called **edges**
- E: collection, not a set
  - Two edges can have the same end vertices
  - Such edges are called **parallel** or **multiple** edges
  - **Self-loop** edges: endpoints are the same
- **Simple** graphs: graphs do not have parallel edges or self loops
- **Path**: sequence of vertices and edges, start at a vertex and ends at a vertex
  - Simple: if each vertex in the path is distinct
- **Cycle**: path that starts and ends at the same vertex, and includes at least one edge
  - Simple: if each vertex in the cycle is distinct(except for the first and last one)
- **Acyclic**: directed graph that has no directed cycles

GRAPHS

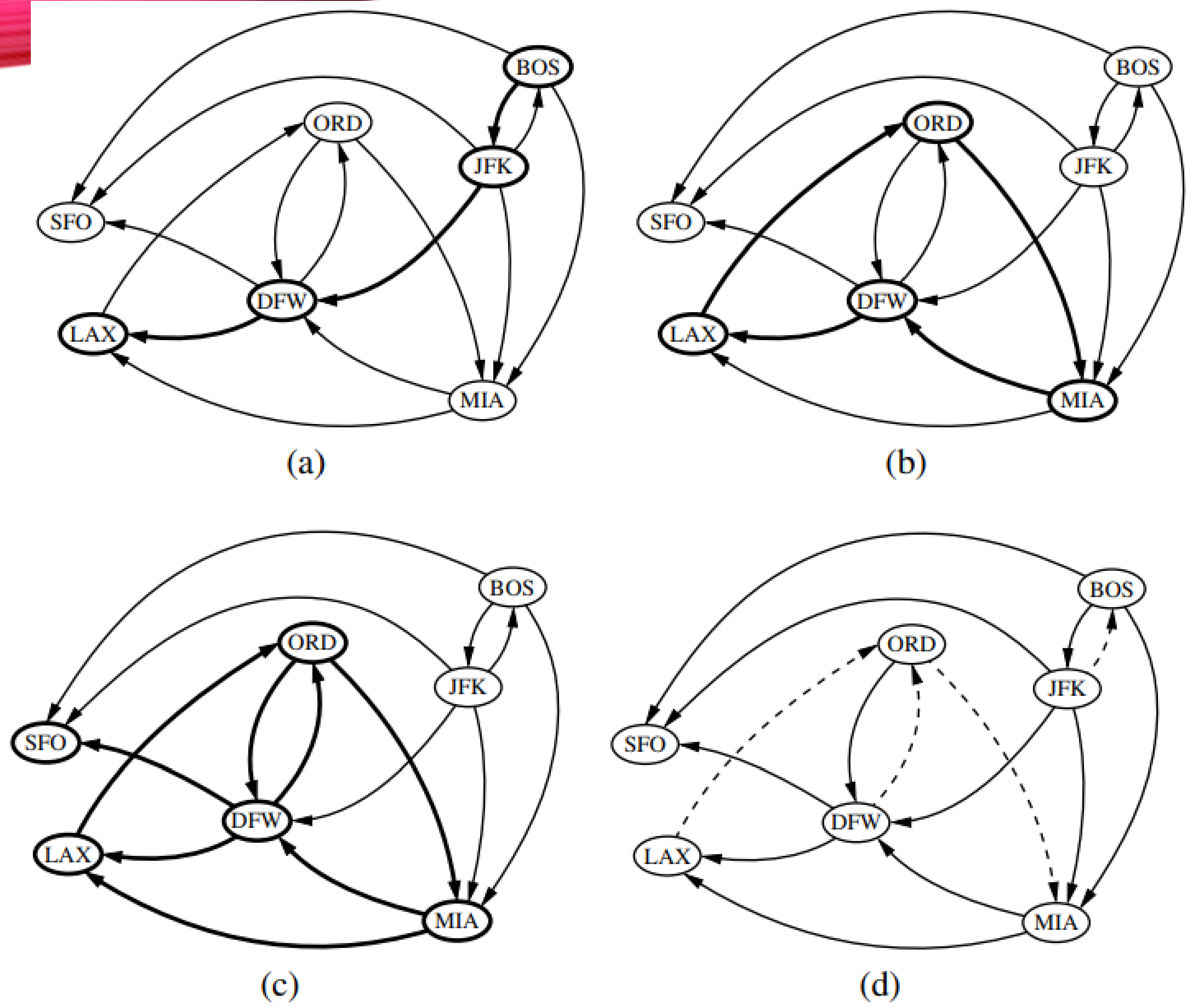
- Directed path
  - A path such that all edges are directed and are traversed along their direction
  - (BOS, NW35, JFK, AA1387, DFW)
- Directed cycle
  - A cycle such that all edges are directed and are traversed along their direction
  - (LAX, UA1200, ORD, UA877, DFW, AA49, LAX)



GRAPHS

- Reachability
  - In a graph G, u **reaches** v, and v is reachable from u, if G has a path from u to v
- Connectivity
  - G is **connected**, if for any two vertices, there is a path between them
  - Directed graph: **strongly connected**
- Subgraph
  - Graph H whose vertices and edges are subsets of the vertices and edges of G
- Spanning subgraph
  - Subgraph of G that contains all the vertices of the graph
- Forest
  - Graph without cycles
- Tree
  - Connected forest
- Spanning tree
  - Spanning subgraph that is a tree

GRAPHS



GRAPHS

- If G is a graph with m edges and vertex set V, then

$$\sum_{v \text{ in } V} deg(v) = 2m.$$

- If G is a directed graph with m edges and vertex set V, then

$$\sum_{v \text{ in } V} indeg(v) = \sum_{v \text{ in } V} outdeg(v) = m.$$

- Let G be a simple graph with n vertices and m edges.
  - If G is undirected, then m <= n(n-1)/2
  - If G is directed, then m <= n(n-1)
- Let G be a undirected graph with n vertices and m edges
  - If G is connected, then m >= n-1
  - If G is a tree, then m = n-1
  - If G is a forest then m <= n-1

GRAPHS

- Graph ADT
- Endpoints: return a tuple (u, v) such that vertex u is the origin and v the destination
- Opposite(v): if v is one endpoint of the edge, return the other endpoint
- Vertex_count(): returns # of vertices
- Vertices(): returns an iteration of all vertices of the graph
- Edge_count(): returns # of edges
- Edges(): returns an iteration of all edges
- Get_edge(u, v): returns the edge from u to v, if one exists, otherwise return None

GRAPHS

- Graph ADT
- degree(v, out= True): returns the # of edges incident to v
- incident_edges(v, out=True): returns an iteration of edges incident to v
- insert_vertex(x = None): create and return a new Vertex storing element x
- insert_edge(u, v, x = None): create and return a new Vertex from u to v
- remove_vertex(v): remove v and all its incident edges from the graph
- remove_edge(e): remove e from the graph

DATA STRUCTURES FOR GRAPHS

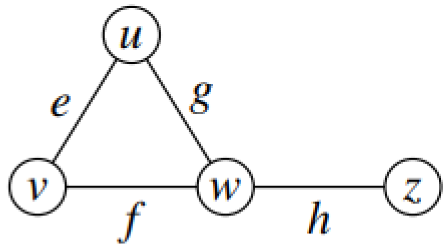
- Edge list
  - An unordered list of all edges, no efficient way to locate a particular edge (u, v), or the set of all edges incident to a vertex v
- Adjacency list
  - For each vertex, a list containing edges that are incident to the vertex
  - Complete set of edges can be determined by taking the union of the smaller sets
- Adjacency map
  - Similar to adjacency list, but secondary container for edges are maps
  - O(1) expected time to access a specific edge(u,v)
- Adjacency matrix
  - Worst case O(1) access to a specific edge (u, v) by maintaining a n*n matrix for each graph with n vertices

DATA STRUCTURES FOR GRAPHS

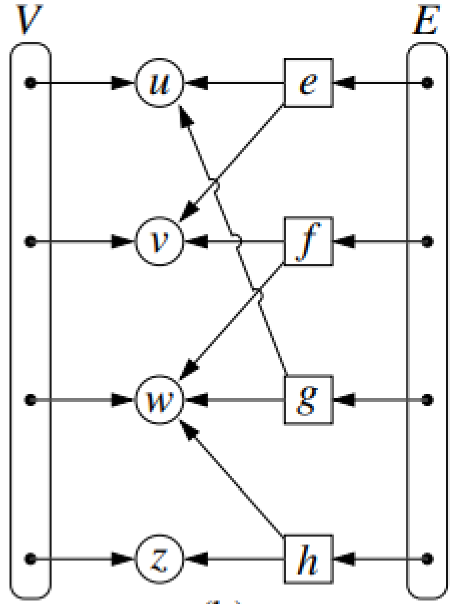
Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

EDGE LIST

- Collections V and E are represented with doubly linked lists (PositionalList)
- Vertex objects
  - Reference to element x
  - Reference to position of the vertex instance in the list V
- Edge objects
  - Reference to element x
  - Reference to the vertex objects associated with the endpoint vertices of e
  - Reference to the position of the edge instance in list E



(a)



(b)

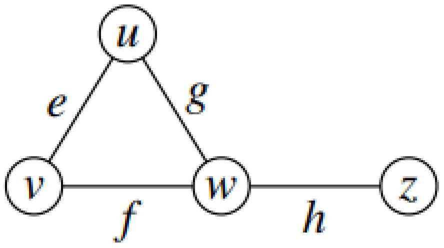
EDGE LIST

- Performance
- Space: O(n + m)
  - n vertices and m edges
- Running time
  - vertices(): O(n)
  - edges(): O(m)
  - get_edge(): O(m)
    - Most significant limitation
  - remove_vertex(v): O(m)
    - Why?

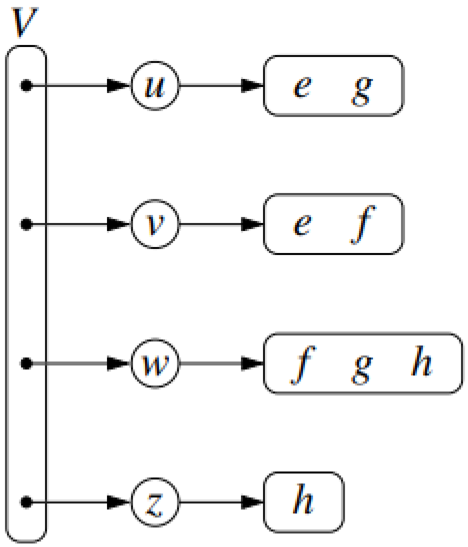
Operation	Running Time
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

ADJACENCY LIST

- Secondary containers for edges that are associated with each individual vertex
- For each v, maintain a collection I(v) called **incidence collection** of v
- Primary structure: collection V of vertices
  - Positional list
- Each vertex instance
  - Direct reference to its I(v) incidence collection



(a)



(b)

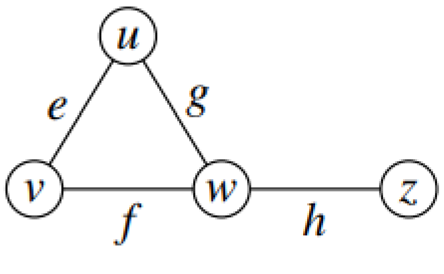
ADJACENCY LIST

- Performance
- Space: O(n + m)
  - n vertices and m edges
- Running time
  - vertices(): O(n)
  - edges(): O(m)
  - get_edge(): O(min(deg(u), deg(v)))
    - Search through either I(u) or I(v)
  - remove_vertex(v): O(dev(v))

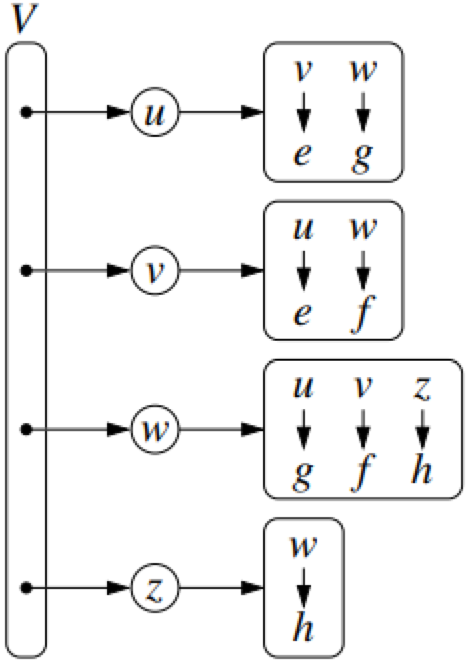
Operation	**Running Time**
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v)	$O(\min(\deg(u),\deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

ADJACENCY MAP

- Adjacency list
  - I(v) uses O(deg(v)) space
  - O(dev(v)) time
- Performance improvement
  - Hash-based map for I(v)
  - get_edge(u,v) can run in expected O(1) time, worst case O(min(deg(u), deg(v))



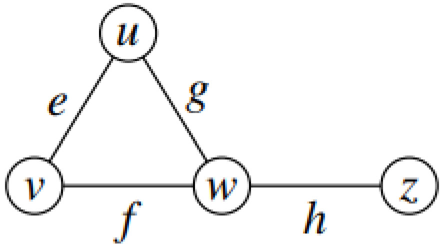
(a)



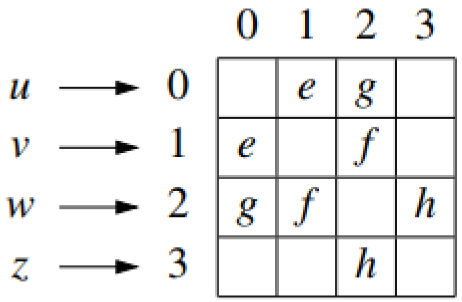
(b)

ADJACENCY MATRIX

- Matrix A (n by n) to locate an edge between a given pair of vertices in worst-case O(1) time
- Verticeis as integers in {0, 1, …, n-1}
- A[I, j] holds a reference to the edge (u, v) if one exists
- Edge (u, v) can be accessed in worst-case O(1) time
- $O(n^2)$ space usage
- Matrix can be used to store only Boolean values, if edges do not store any additional data



(a)



(b)

THIS LECTURE: GRAPH TRAVERSALS

- Graph Traversals: a systematic procedure for exploring a graph by examining all of its vertices and edges
- Why traversal
  - Compute a path from u to v
  - Given a start vertex s of G, for every vertex v of G, compute the shortest path
  - Test whether G is connected
  - Compute a spanning tree of G if G is connected
  - Compute the connected components of G
  - Compute a cycle in G
  - Compute a directed path from u to v (for directed graphs)
  - Determine if G is asyclic (for directed graphs)
  - Determine if G is strongly connected (for directed graphs)

THIS LECTURE: GRAPH TRAVERSALS

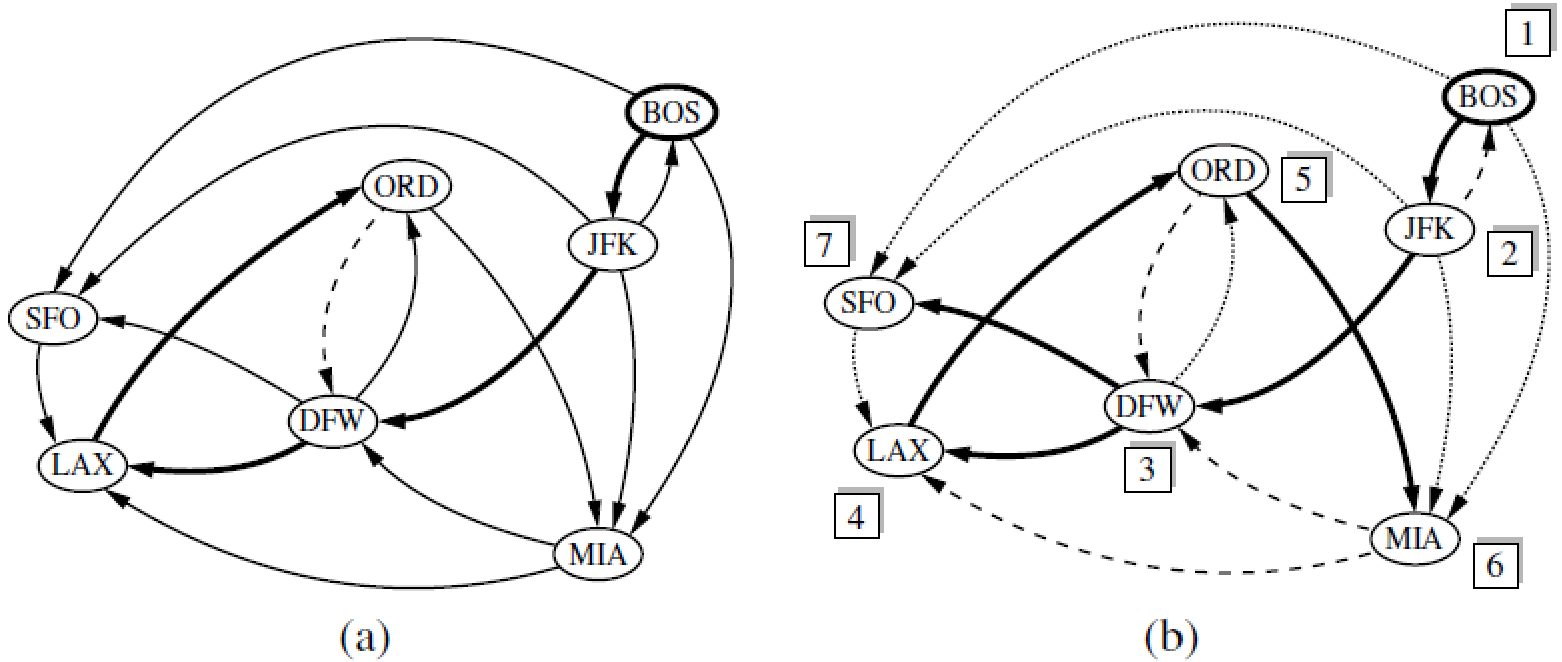
- **Depth First Search**
  - Maze solving
- Begin at a starting vertex s
  - S is the current vertex u
- For each edge of s
  - If it leads to a visited vertex, ignore
  - If it leads to an unvisited vertex v, make v the current vertex u, make u "visited"
  - repeat

Algorithm DFS(G,u): {We assume u has already been marked as visited}
  Input: A graph G and a vertex u of G
  Output: A collection of vertices reachable from u, with their discovery edges
  for each outgoing edge e = (u,v) of u **do**
    if vertex v has not been visited **then**
      Mark vertex v as visited (via edge e).
      Recursively call DFS(G,v).

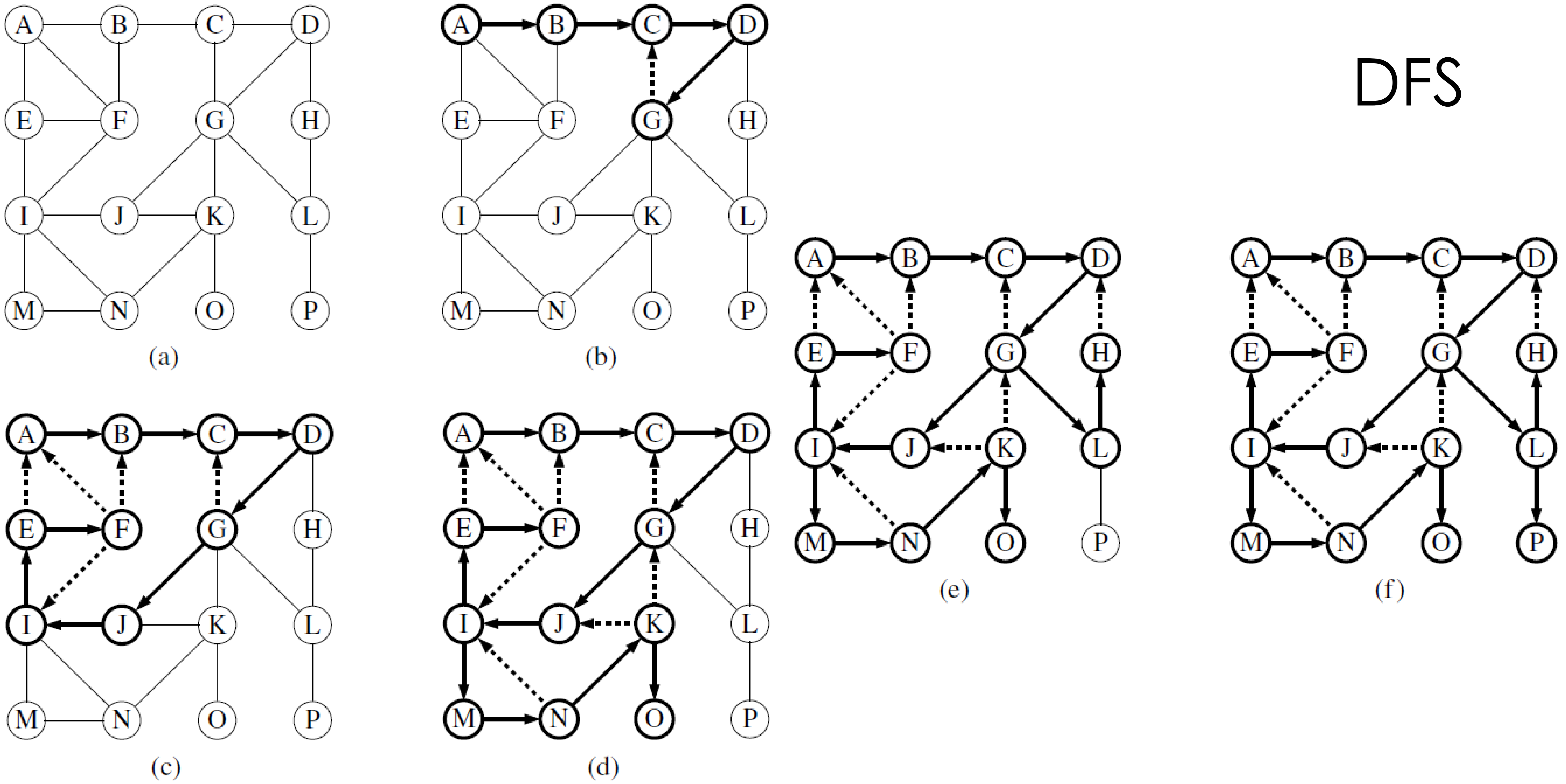
DFS

- **Depth First Search**
- Depth-first search tree: rooted at s
- When an edge d(u, v) is used to discover a new vertex v, the edge is known as a discovery edge (**tree edge**)
- **Nontree edges**
  - **Back edges**: connect a vertex to an ancestor in the DFS tree
  - **Forward edges**: connect a vertex to a descendant in the DFS tree
  - **Cross edges**: connect a vertex to a vertex that is neither its ancestor nor its descendant

DFS


BOS
ORD
JFK
SFO
DFW
LAX
MIA
(a)
1
BOS
ORD
5
7
JFK
2
SFO
DFW
LAX
3
4
MIA
6
(b)

DFS



DFS

- **Depth First Search**
  - G = undirected graph, when a DFS is performed at a starting vertex s, the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s
  - G = directed graph, a DFS on G starting at vertex s visits all the vertices of G that are reachable from s. Also the DFS tree contains directed paths from s to very vertex reachable from s
- Running time
  - $O(n_s + m_s)$
    - n_s : # of vertices reachable from a vertex s
    - m_s : # of incident edges to vertices reachable from s

DFS

- Implementation

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):          # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:              # v is an unvisited vertex
      discovered[v] = e                  # e is the tree edge that discovered v
      DFS(g, v, discovered)              # recursively explore from v
```

```
def construct_path(u, v, discovered):
  path = [ ]
  if v in discovered:
    # we build list from v to u and th
    path.append(v)
    walk = v
    while walk is not u:
      e = discovered[walk]
      parent = e.opposite(walk)
      path.append(parent)
      walk = parent
    path.reverse( )
  return path
```

DFS

- Applications of DFS
  - Testing for connectivity
    - Undirected graph, start from an arbitrary vertex and test if len(discovered) == n
    - Directed graph, test for strong connection O(n(n+m)) time, test for each (u, v) pairs
      - O(n+m) is achieved by 2 DFSs, how?
  - Computing all connected components
  - Detecting cycles
    - Determine if there are **back edges**

```
def DFS_complete(g):
  """Perform DFS for entire

  Result maps each vertex v
  (Vertices that are roots of
  """
  forest = { }
  for u in g.vertices( ):
    if u not in forest:
      forest[u] = None
      DFS(g, u, forest)
  return forest
```

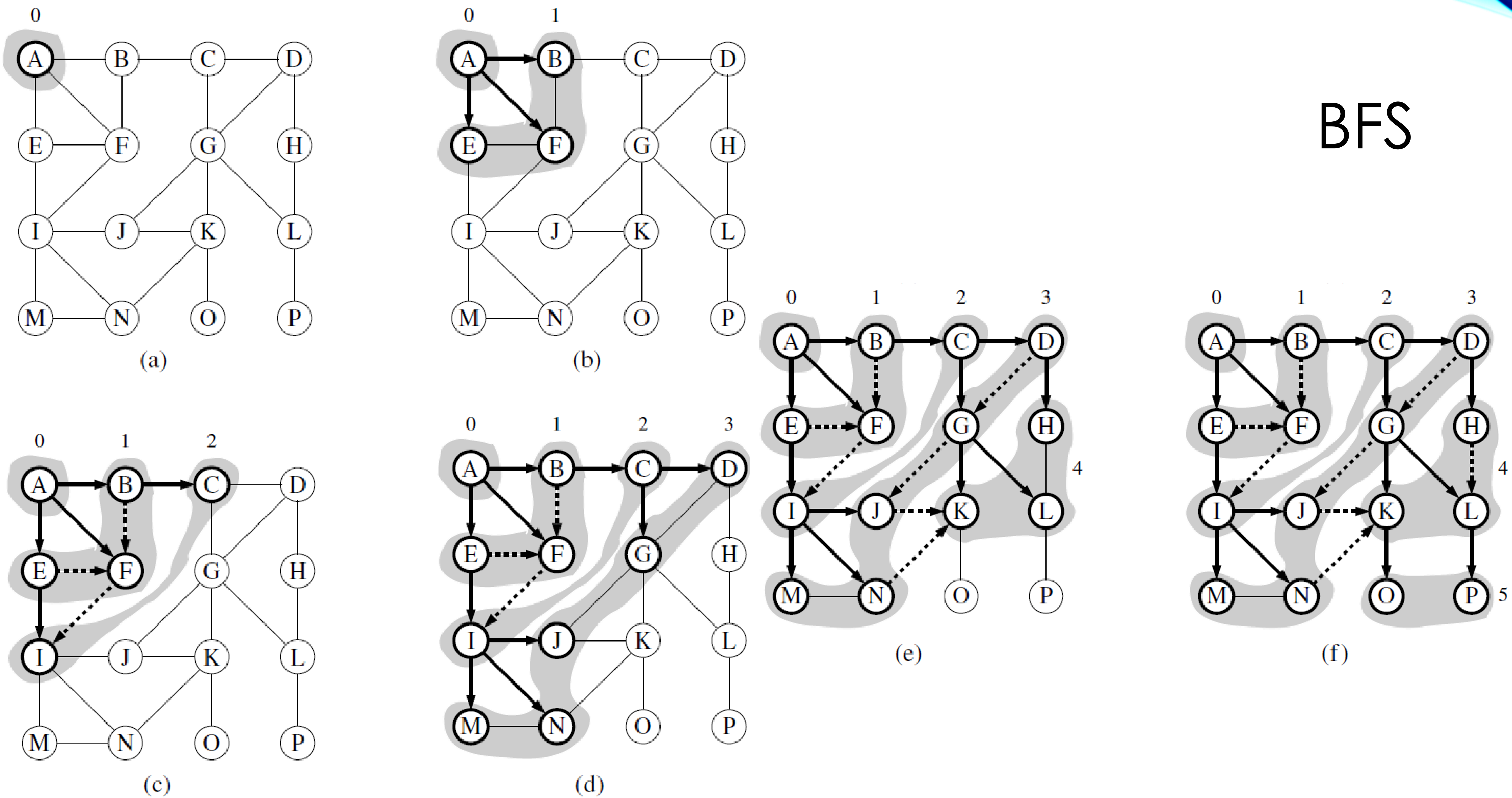
BFS

- **Breadth First Search**
  - Web crawling
  - Social network
  - Garbage collection
  - Rubic's cube
- Start from vertex s
- One 'level' at a time
- Level = all adjacent vertex from s,
  - If vertex u is unvisited, mark u as 'visited'
  - If vertex u is visited, discard
- Proceed to the next 'level'

```
def BFS(g, s, discovered):
  """Perform BFS of the undiscove

  discovered is a dictionary mappin
  discover it during the BFS (s sho
  Newly discovered vertices will be
  """
  level = [s]
  while len(level) > 0:
    next_level = [ ]
    for u in level:
      for e in g.incident_edges(u):
        v = e.opposite(u)
        if v not in discovered:
          discovered[v] = e
          next_level.append(v)
    level = next_level
```

BFS



0
A B C D
E F G H
I J K L
M N O P
(a)
0 1
A B C D
E F G H
I J K L
M N O P
(b)
0 1 2
A B C D
E F G H
I J K L
M N O P
(c)
0 1 2 3
A B C D
E F G H
I J K L
M N O P
(d)
0 1 2 3
A B C D
E F G H
4
I J K L
M N O P
(e)
0 1 2 3
A B C D
E F G H
4
I J K L
M N O P
5
(f)

BFS

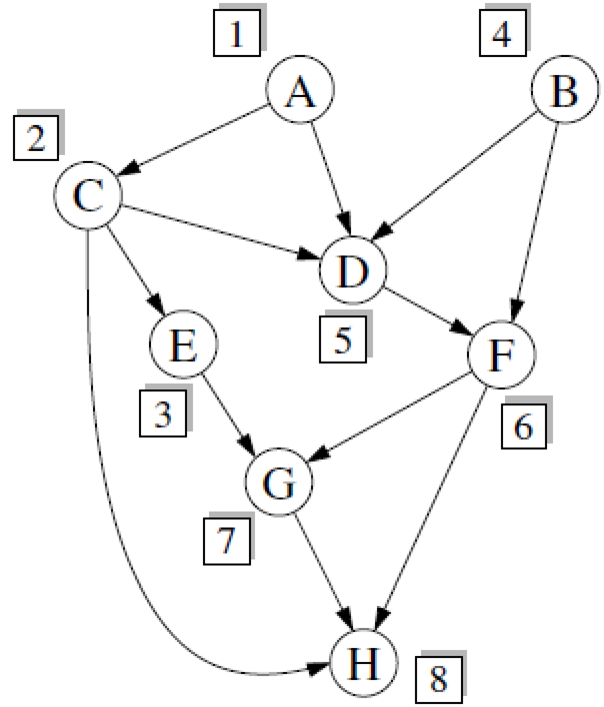
- **Nontree edges**
  - Undirected graph: all non tree edges are cross edges
  - Directed graph: back edges or cross edges
- Properties:
  - If G is a graph on which a BFS traversal starting at vertex s has been performed
    - The traversal visits all vertices of G that are reachable from s
    - For each vertex v at level i, the path of the BFS tree T between s and v has i edges and any other path of G from s to v has at least i edges
    - If (u, v) is an edge that is not in the BFS tree, then the level number of v can be at most 1 greater than the leve number of u
  - If G is a graph with n vertices and m edges, BFS traversal of G takes O(n+m) time

TRANSITIVE CLOSURE (传递闭包)

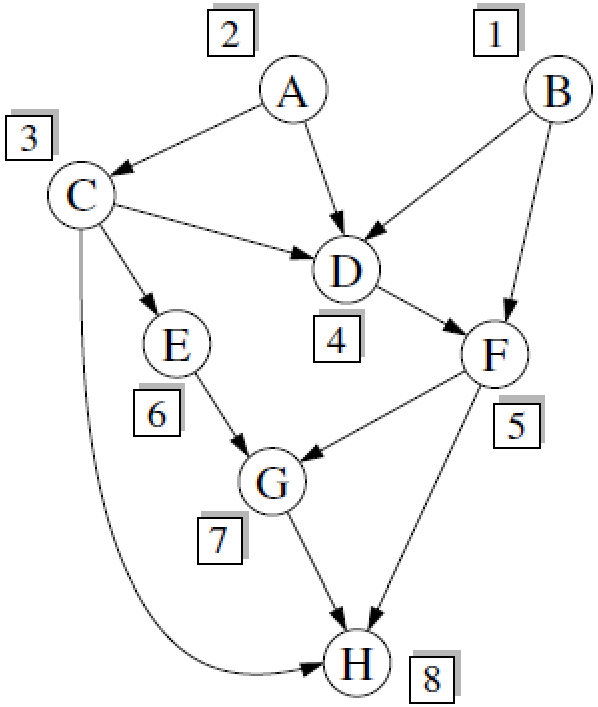
- Reachability problem in a directed graph
  - DFS or BFS: O(n + m) time
- In certain applications, we want to answer many reachability queries more efficiently
- Transitive closure of a directed graph G
  - Is itself a directed graph G', such that
  - The vertices of G' are the same as the vertices of G
  - G' has an edge (u, v), when ever G has a directed path from u to v
- Computation of transitive closure O(n(n+m))
  - N DFSs
- Floyd-Warshall: $O(n^3)$ – not covered in this course

DIRECTED ACYCLIC GRAPHS

- Directed Acyclic Graph (DAG)
  - Directed graphs without directed cycles
- Applications
  - Prerequisites between courses of a degree program
  - Inheritance between classes of an object-oriented program
  - Scheduling constraints between the tasks of a project
- Topological Ordering
  - If G is a directed graph with n vertices
  - Topological ordering of G: an ordering $v1, v2, \ldots, vn$ of vertices of G such that for every edge (vi, vj) of G, it is the case that $i < j$



(a)

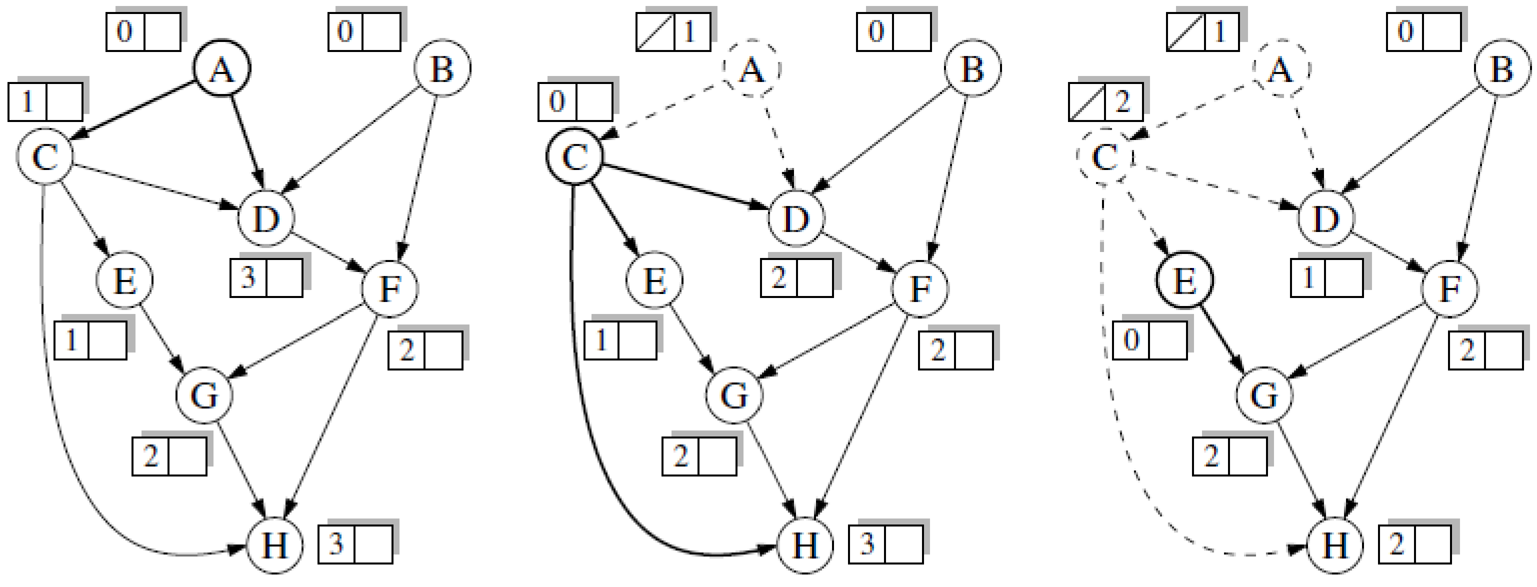


(b)

```
def topological_sort(g):
  """Return a list of verticies of directed acy

  If graph g has a cycle, the result will be inc
  """
  topo = [ ]                # a list of vertices
  ready = [ ]               # list of vertices th
  incount = { }             # keep track of in-
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)      # pa
    if incount[u] == 0:                  # if
      ready.append(u)                    # it
  while len(ready) > 0:
    u = ready.pop( )                     # u
    topo.append(u)                       # ad
    for e in g.incident_edges(u):        # co
      v = e.opposite(u)
      incount[v] −= 1                    # v
      if incount[v] == 0:
        ready.append(v)
  return topo
```

DIRECTED AC

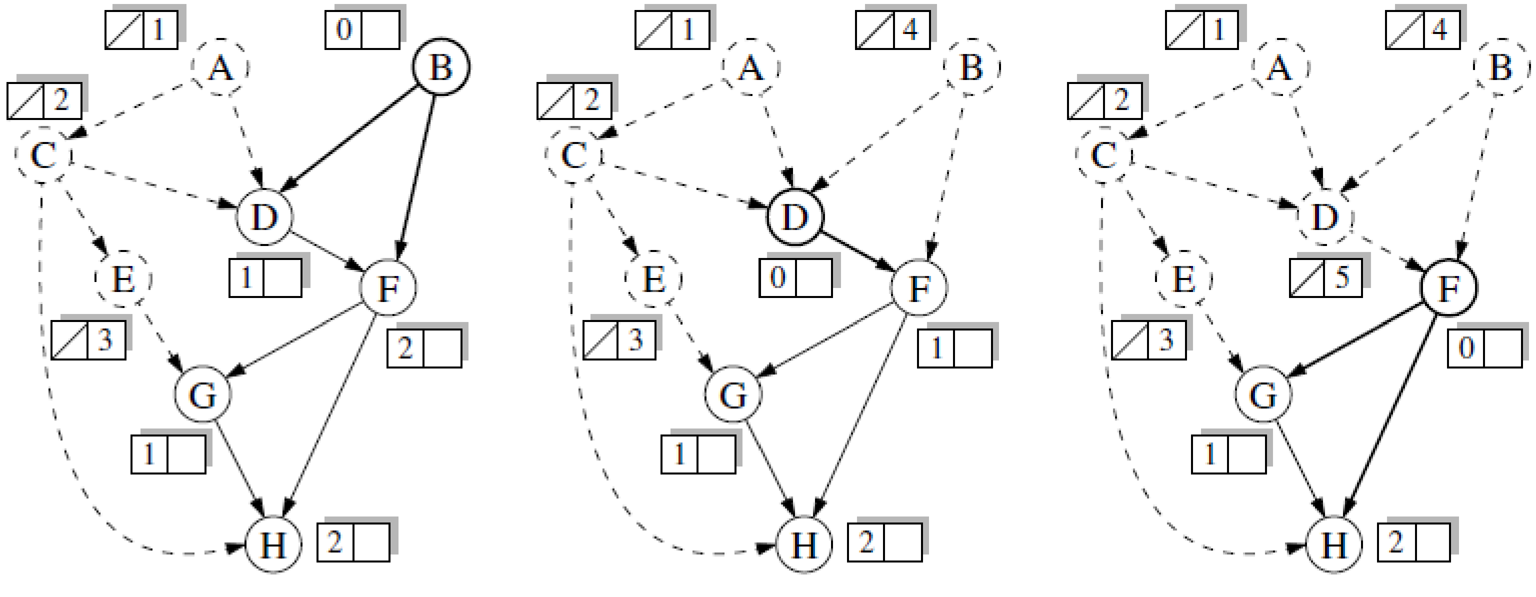


(a) (b) (c)

```
def topological_sort(g):
  """Return a list of verticies of directed a

  If graph g has a cycle, the result will be
  """
  topo = [ ]                    # a list of vertice
  ready = [ ]                   # list of vertices
  incount = { }                 # keep track of i
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)        #
    if incount[u] == 0:                    #
      ready.append(u)                      #
  while len(ready) > 0:
    u = ready.pop( )                       #
    topo.append(u)                         #
    for e in g.incident_edges(u):          #
      v = e.opposite(u)
      incount[v] −= 1                      #
      if incount[v] == 0:
        ready.append(v)
  return topo
```

DIRECTED AC

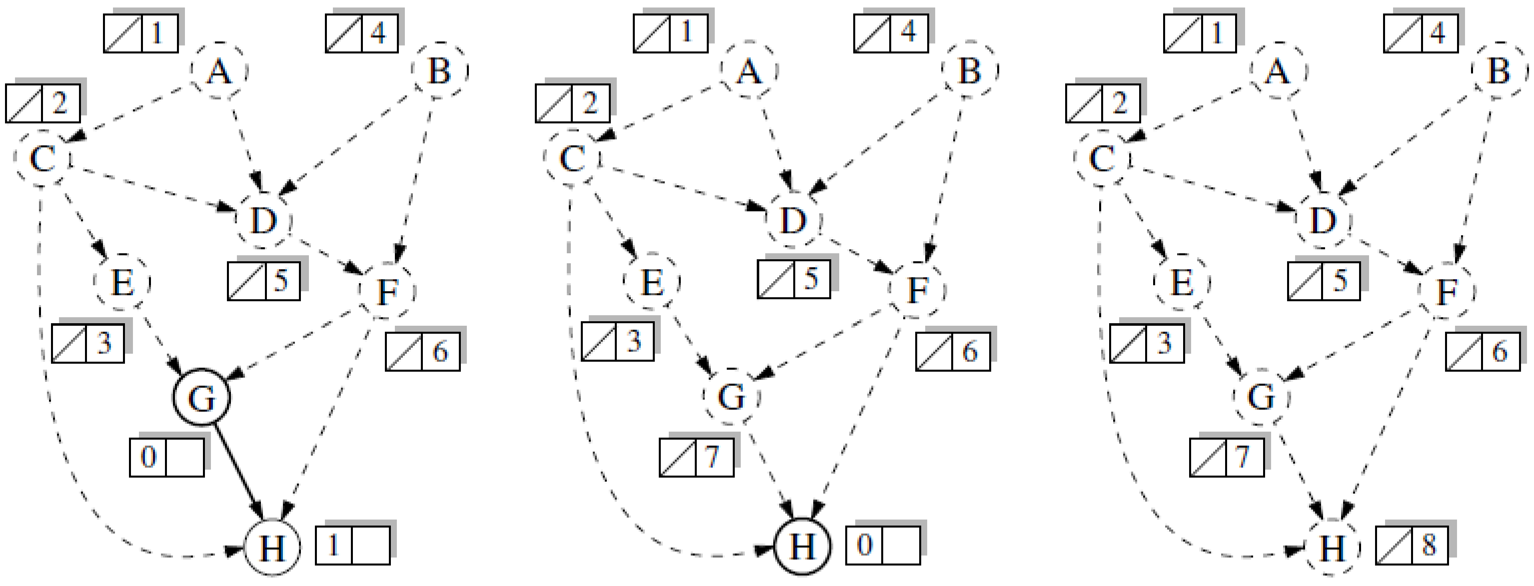


(d) (e) (f)

```
def topological_sort(g):
  """Return a list of verticies of directed

  If graph g has a cycle, the result will be
  """
  topo = [ ]                  # a list of verti
  ready = [ ]                 # list of vertice
  incount = { }               # keep track of
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)      #
    if incount[u] == 0:                  #
      ready.append(u)                    #
  while len(ready) > 0:
    u = ready.pop( )                     #
    topo.append(u)                       #
    for e in g.incident_edges(u):        #
      v = e.opposite(u)
      incount[v] −= 1                    #
      if incount[v] == 0:
        ready.append(v)
  return topo
```

DIRECTED AC



(g) (h) (i)

```
def topological_sort(g):
  """Return a list of verticies of directe

  If graph g has a cycle, the result will b
  """
  topo = [ ]                # a list of vert
  ready = [ ]               # list of vertic
  incount = { }             # keep track c
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)
    if incount[u] == 0:
      ready.append(u)
  while len(ready) > 0:
    u = ready.pop( )
    topo.append(u)
    for e in g.incident_edges(u):
      v = e.opposite(u)
      incount[v] −= 1
      if incount[v] == 0:
        ready.append(v)
  return topo
```

THANKS

See you in the next session!