Graphs

CPE111: Programming with Data Structures

Reference

- Michael T.Goodrich, Roberto Tamassia, Michael H. Goodwasser. *Data Structures and Algorithms in python*. *Chapter8, Chapter 14*. John Wiley&Sons,Inc. 2013
- http://interactivepython.org/runestone/static/pythonds/Graphs/toctree.html

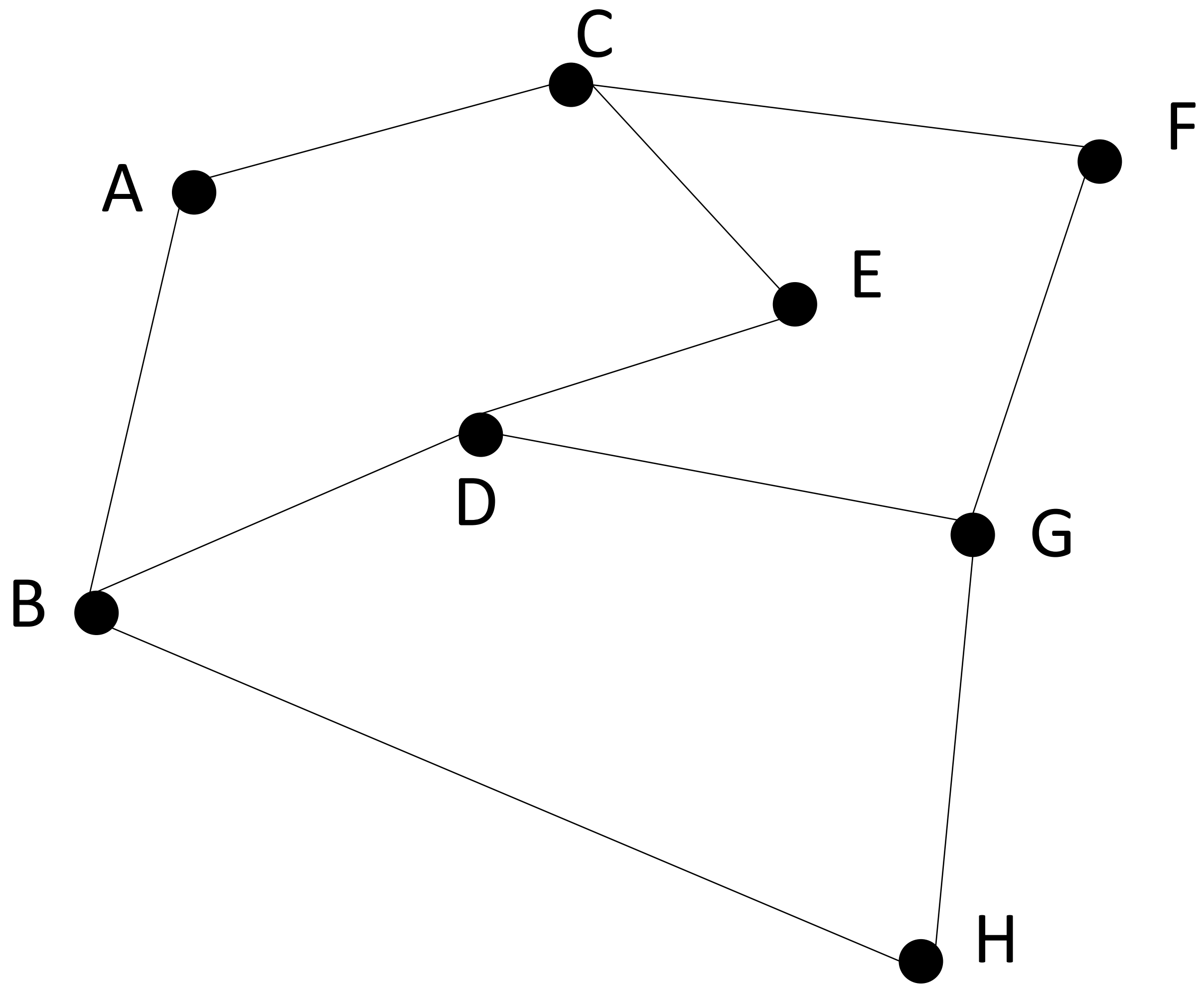
A graph represents relationships between pairs of objects. So that, a graph is composed of a set of objects, called **vertices (nodes)**, and a set of connections between them, called **edges (arcs)**.

Edges in a graph are either directed or undirected.

- An edge (u,v) is said to be directed from u to v if the pair (u,v) is ordered, with u preceding v.
- An edge (u,v) is said to be undirected if the pair (u,v) is not ordered. Undirected edges are sometimes denoted with set notation, as {u,v},

but in simple way, we use the same (u,v) in both cases

G r a p h s are visualized by drawing the vertices as ovals or rectangles and the edges as line, segments or curves connecting pairs of ovals and rectangles.



vertex

weight

edge

walk

closed walk

path

cycle

directed graph

acyclic graph

undirected graph

undirected acyclic graph

directed acyclic graph

The Graph ADT

A graph G is an ordered pair of a set V of vertices and a set E of edges →

G = (V,E)

A graph is a collection of vertices and edges by a combination of 2 data types:

- **A Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code);
- **An Edge** stores an associated object (e.g., a flight no., travel distance, cost)

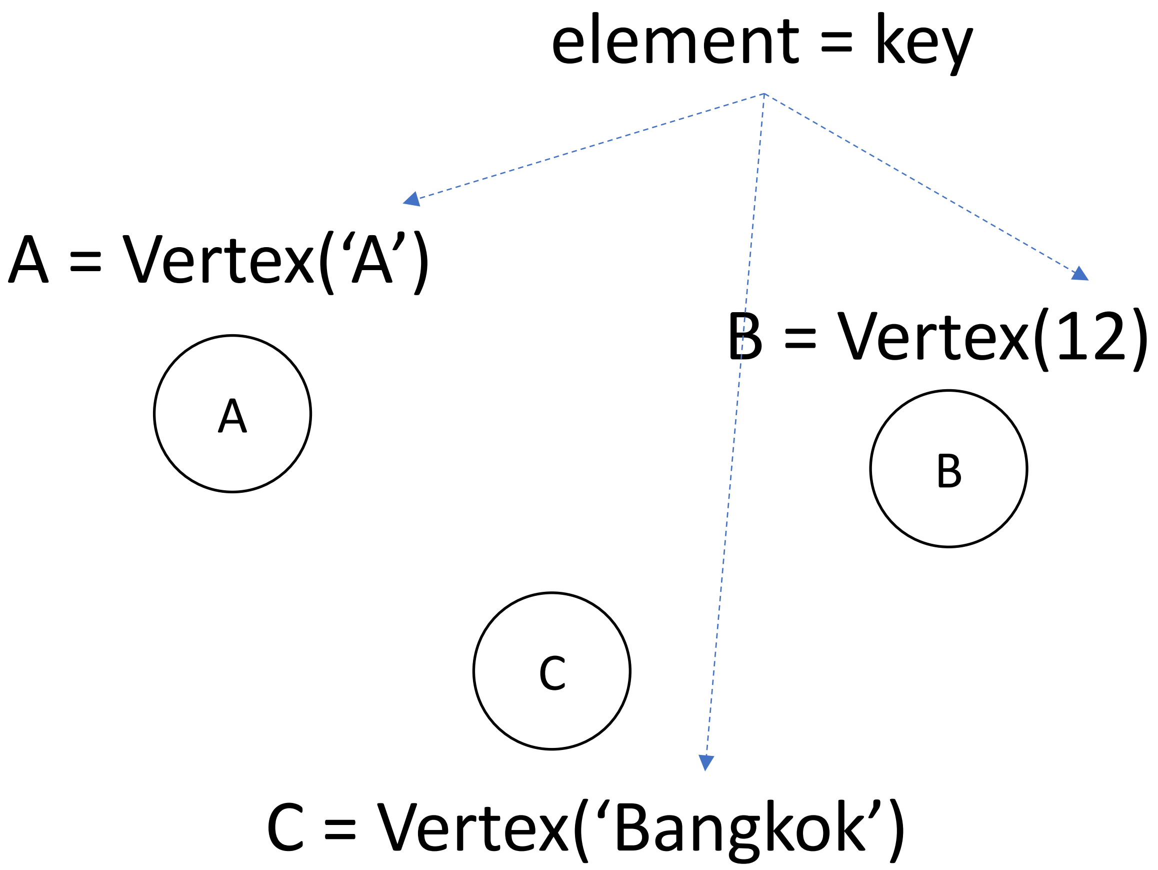
In addition, we assume that an Edge supports the following methods:

endpoints( ): Return a tuple (u,v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.

opposite(v): Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

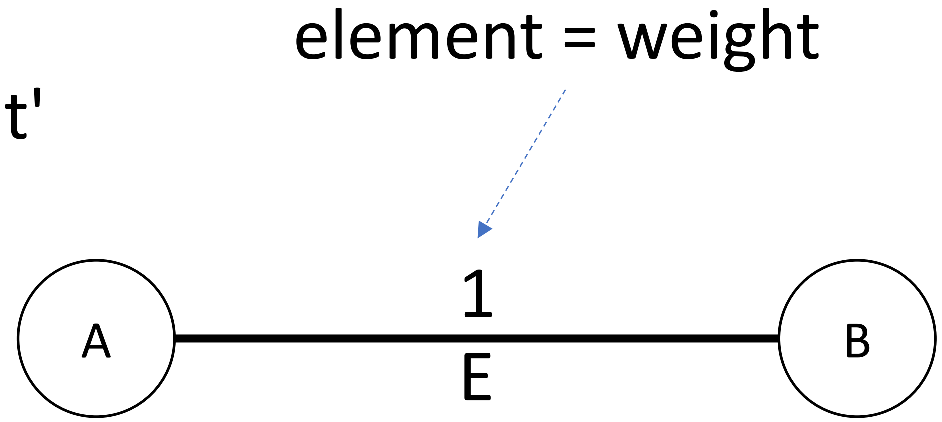
Vertex Object

```python
class Vertex:
    __slots__ = '_element'
    def __init__(self, x):
        self._element = x
    def element(self):
        return self._element
```



Edge Object

```
class Edge:
    __slots__ = '_origin' , '_destination', '_element'
    def __init__(self, u, v, x):
        self._origin = u
        self._destination = v
        self._element = x
    def endpoints(self):
        return (self._origin, self._destination)
    def opposite(self, v):
        return self._destination if v is self._origin else self._origin
    def element(self):
        return self._element
```



E = Edge(A,B,1)

E.endpoints() → (A,B)
E.opposite(A) → B
E.element() → 1

The **Graph ADT** includes the following methods:

vertex_count(): Return the number of vertices of the graph.
vertices(): Return an iteration of all the vertices of the graph.
edge_count(): Return the number of edges of the graph.
edges(): Return an iteration of all the edges of the graph.
get_edge(u,v): Return the edge from vertex *u* to vertex *v*, if one exists; otherwise return None. For an undirected graph, there is no difference between get_edge*(u,v)* and get_edge*(v,u)*.
degree(v, out=True): For an undirected graph, return the number of edges incident to vertex *v*. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex *v*, as designated by the optional parameter.

incident_edges(v, out=True): Return an iteration of all edges incident to vertex v. In the case of a <u>directed graph, report outgoing edges by default</u>; report incoming edges if the optional parameter is set to False.

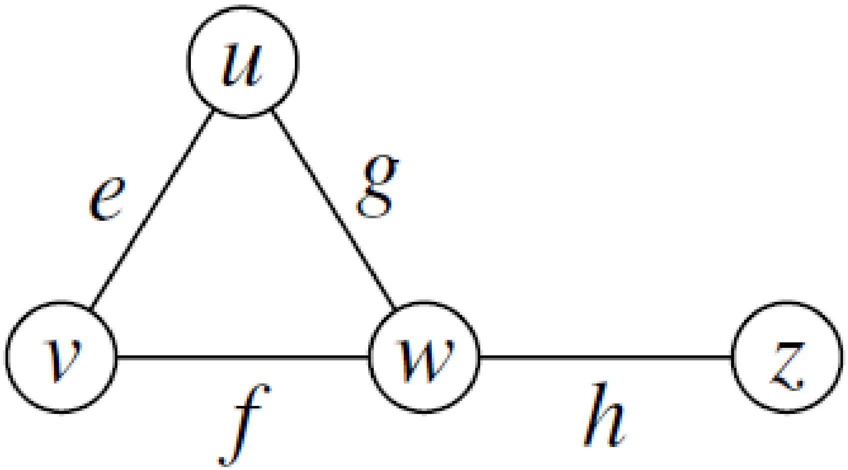
insert_vertex(x=None): Create and return a new Vertex storing element x.

insert_edge(u, v, x=None): Create and return a new Edge from vertex u to vertex v, storing element x (None by default).

remove_vertex(v): Remove vertex v and all its incident edges from the graph.

remove_edge(e): Remove edge e from the graph.

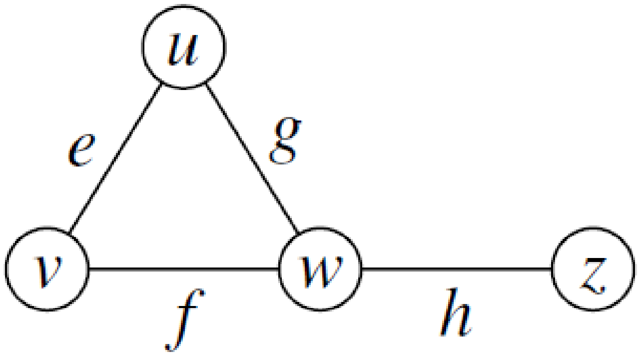
Implementation of the Graph ADT

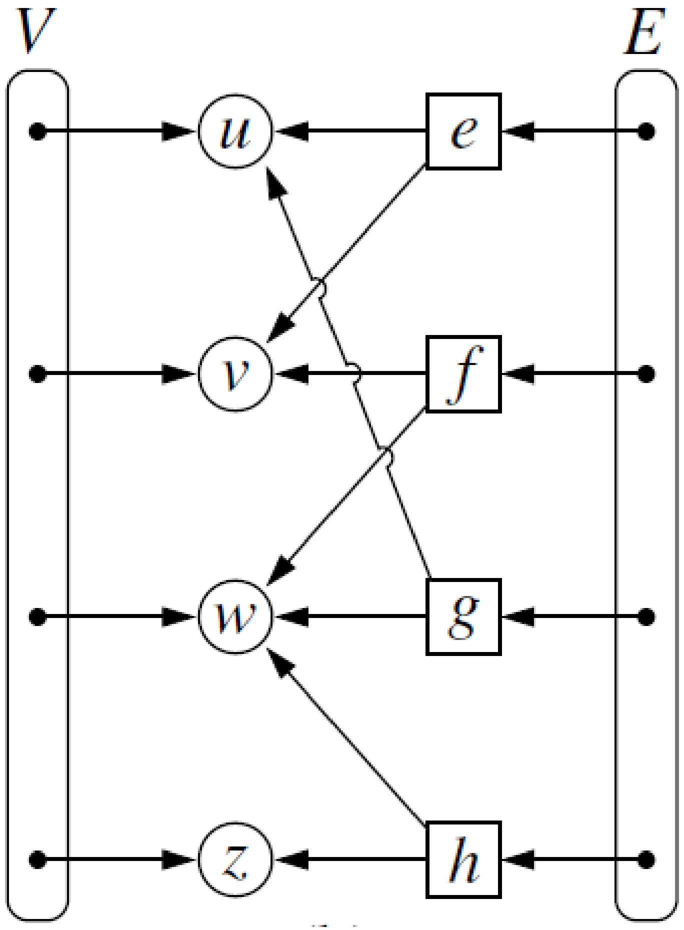


- *Edge List*
- *Adjacency List*
- *Adjacency map*
- *Adjacency matrix*

Edge List

The edge list structure is the most simple, but not the most efficient.
All vertex objects are stored in a list **V**, and all edge objects are stored in a list **E**.

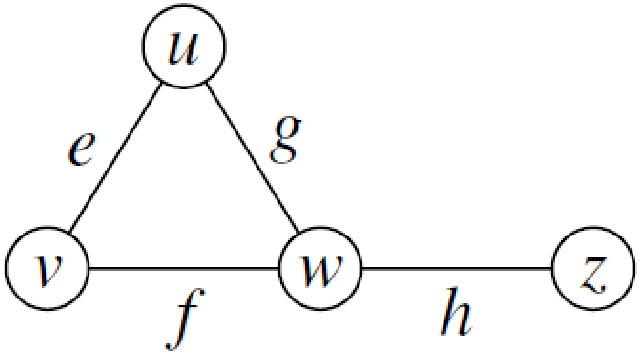


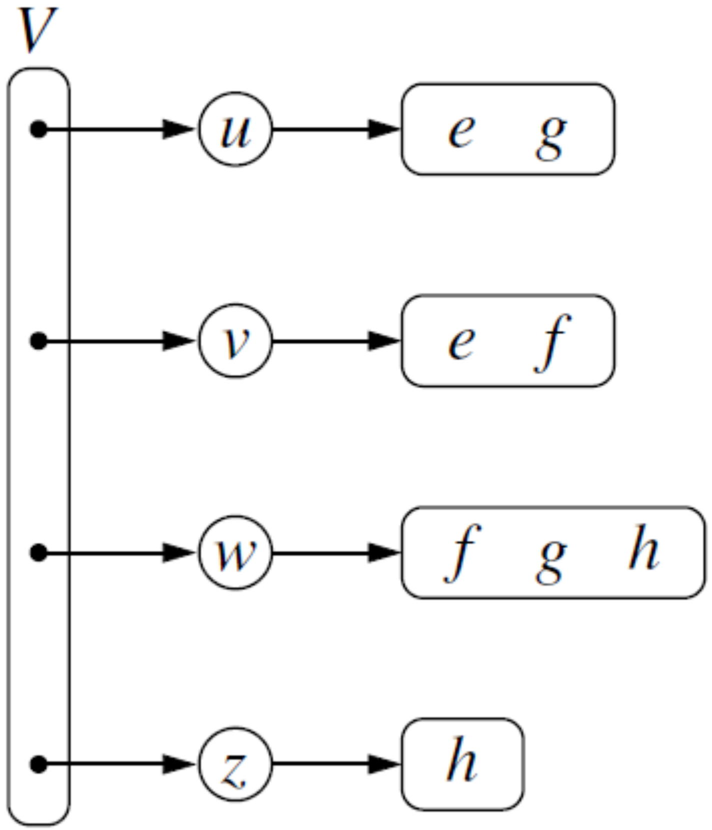


V = [u, v, w, z]
E = [
{e:[u,v]},
{f:[v,w]},
{g:[u,w]},
{h:[w,z]}
]

Adjacency list

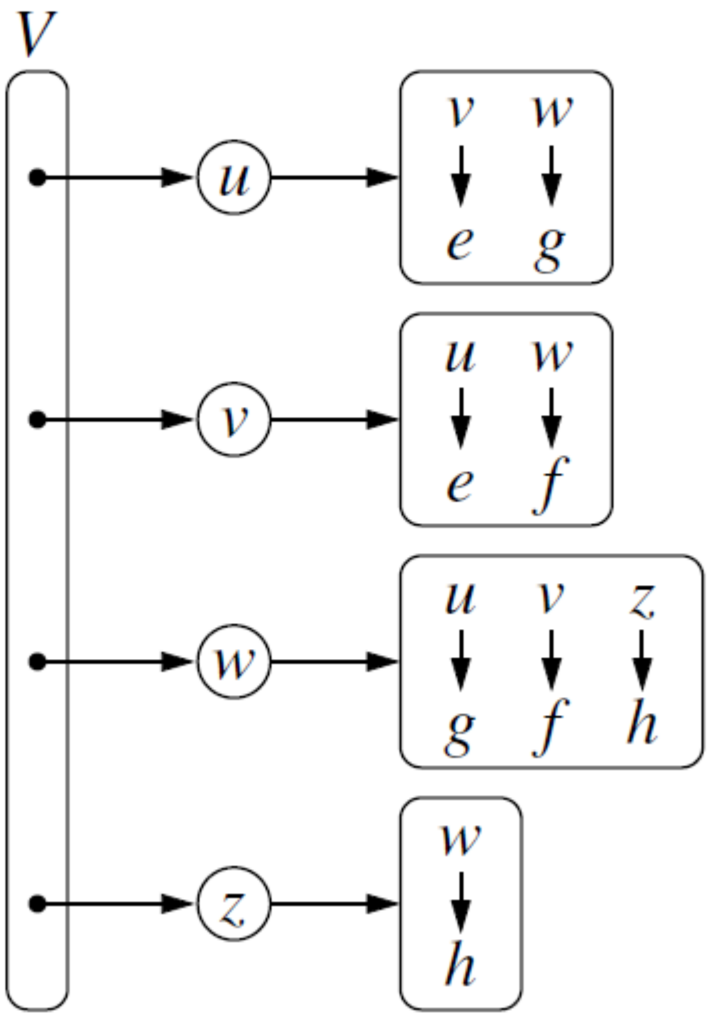
Adjacency list structure groups the edges of a graph by storing them in smaller, secondary containers that are associated with each individual vertex. Specifically, for each vertex **v**, we maintain a collection **I(v)**, called the incidence collection of **v**, whose entries are edges incident to **v**.



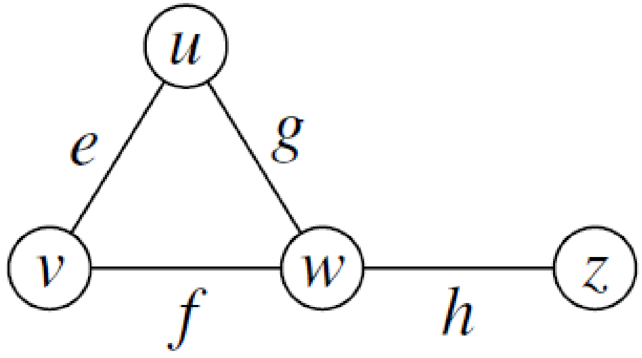



```
u, v, w, z = range(4)
V = [
        [e,g],  #u
        [e,f],  #v
        [f,g,h],#w
        [h]     #z
    ]
```

Adjacency Map



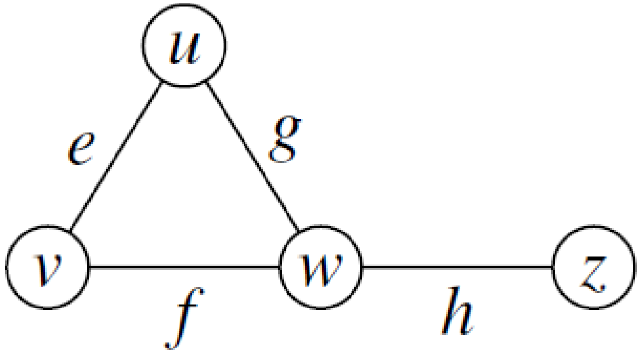
improve the performance by using a hash-based map to implement $I(v)$ for each vertex v. Specifically, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value.




```
u, v, w, z = range(4)
V = [
        {v:e, w:g},    #u
        {u:e, w:f},    #v
        {u:g, v:f, z:h},#w
        {w:h}          #z
    ]
```

Adjacency Matrix

<u>The adjacency matrix</u> structure for a graph ***G*** augments the edge list structure with a matrix ***A*** which allows us to locate an edge between a given pair of vertices Specifically, the cell ***A[i, j]*** holds a reference to the edge (u,v), if it exists.



		0	1	2	3
$u \longrightarrow$	0		e	g	
$v \longrightarrow$	1	e		f	
$w \longrightarrow$	2	g	f		h
$z \longrightarrow$	3			h	

```
u, v, w, z = range(4)
#      u  v  w  z
A = [[0, e, g, 0]   #u
      [e, 0, f, 0]   #v
      [g, f, 0, h]   #w
      [0, 0, h, 0]] #z
```

A[u][v] → e

Summary – Data Structure of Graphs

- ***Edge List***
  - an unordered list of all edges.
  - simple and minimally suffices, but there is no efficient way to locate a particular edge ***(u,v),*** or the set of all edges incident to a vertex ***v***.

- ***Adjacency List***
  - a separate list for each vertex containing those edges that are incident to the vertex.
  - The complete set of edges can be determined by taking the union of the smaller sets, while the organization allows us to more efficiently find all edges incident to a given vertex.

- ***Adjacency map***
  - similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key.
  - This allows for access to a specific edge (u,v) in $O(1)$ expected time.
- ***Adjacency matrix***
  - provides worst-case $O(1)$ access to a specific edge (u,v) by maintaining an $n \times n$ matrix, for a graph with n vertices.
  - Each entry is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the entry will be None.
  - good for a dense graph

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.	$O(1)$

Graph Traversal

- ***depth-first search (DFS)***

Depth-first search is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

- ***breadth-first search (BFS)***

Breadth-first search proceeds in rounds and subdivides the vertices into levels. BFS starts at vertex **s**, at level 0 and go to visit all vertices adjacent to the start vertex **s** and masked as level 1 and so on.

Depth-First Search Algorithm

Define:

- ***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 vertex-vertex that is neither its ancestor nor descendant.

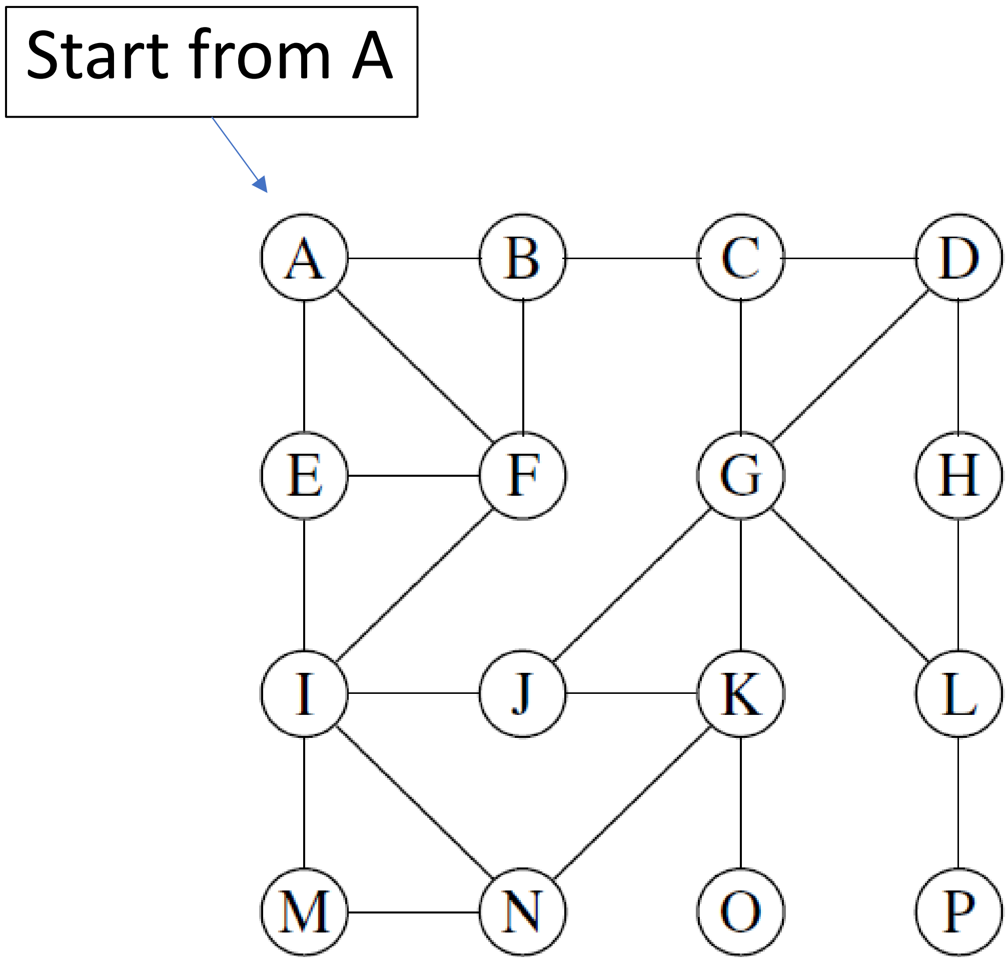
Algorithm DFS(G,u): *# **u** has already been marked as visited*

Input: A graph **G** and a vertex **u** of **G**

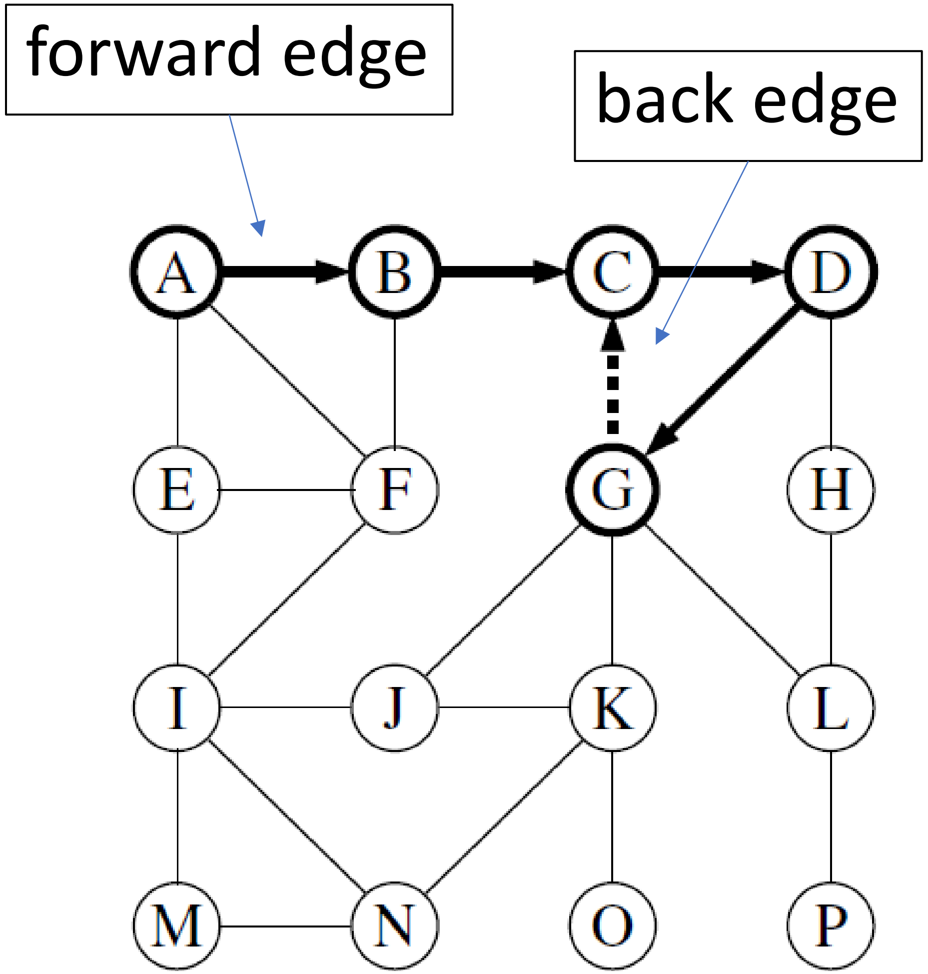
Output: A collection of vertices reachable from **u**, with their discovery edges

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)**.

```
def DFS(g,u,discovered={}):
   discovered[u] = None
   for e in g.incident_edges(u):
      v = e.opposite(u)
      if v not in discovered:
         print('add '+str(v)+' to discovered')
         discovered[v] = e
         DFS(g,v,discovered)
   return discovered.keys()
```

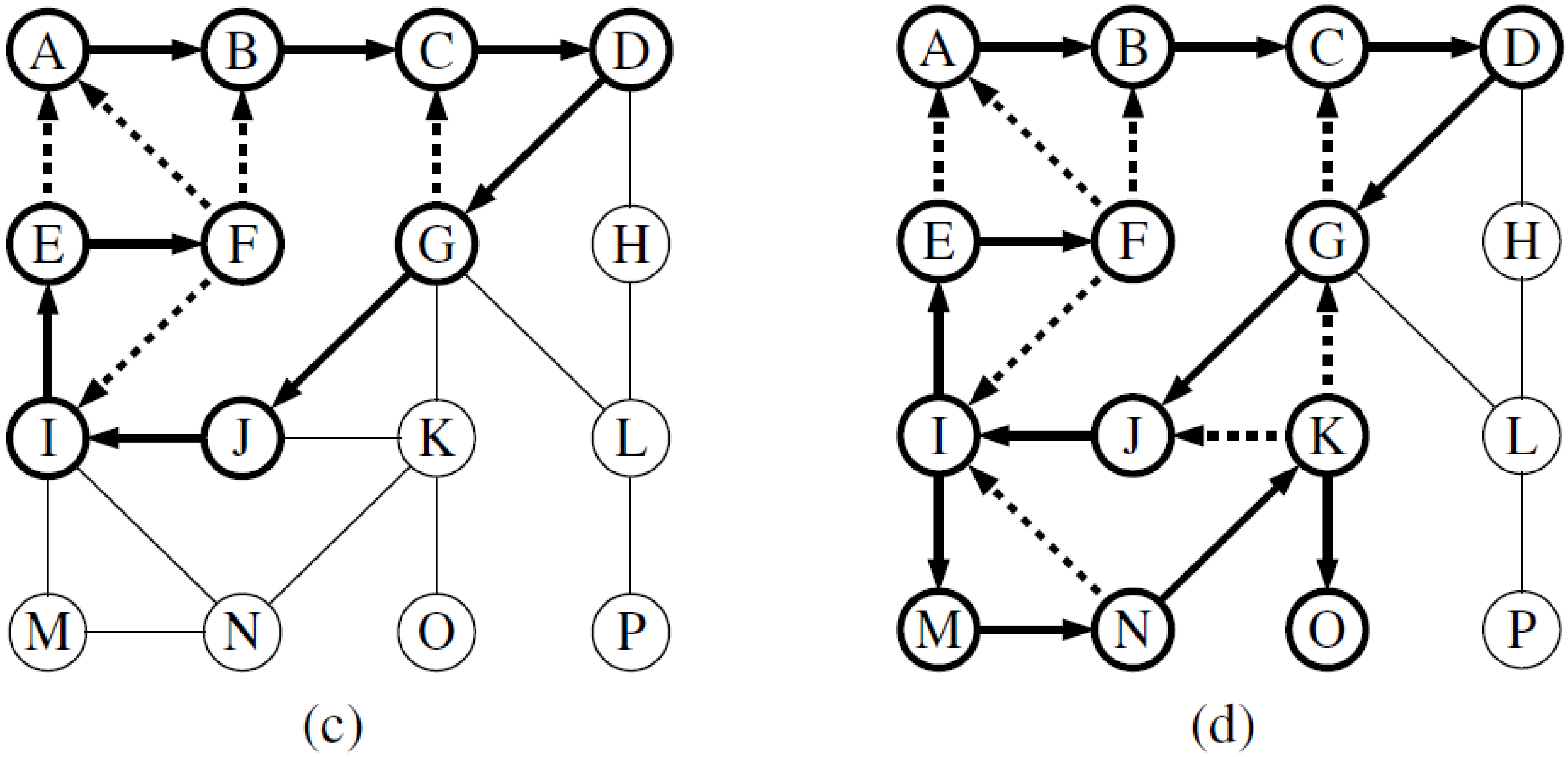

Start from A
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

(a)


forward edge
back edge
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

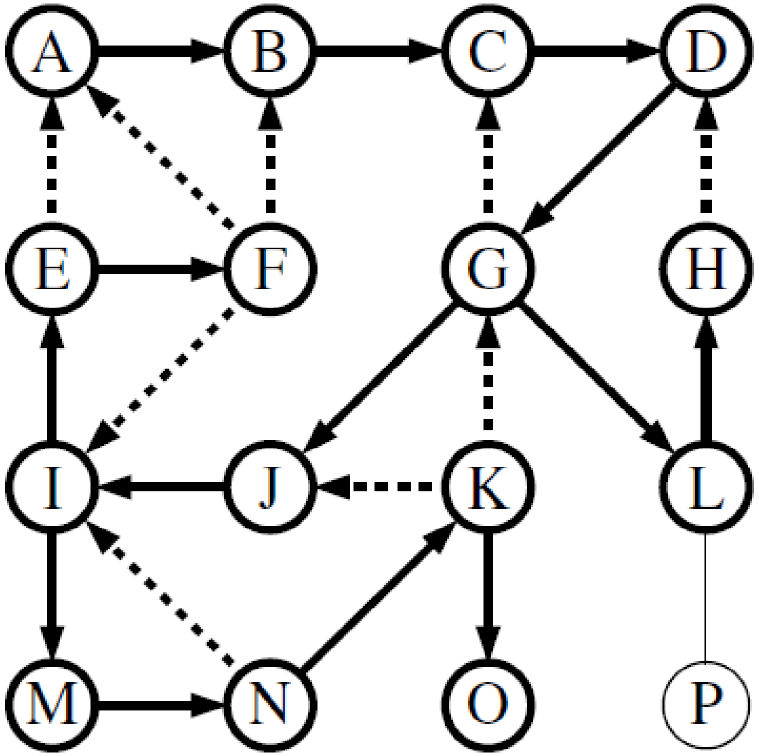
(b)

Stop at O , track back to G and resume dfs

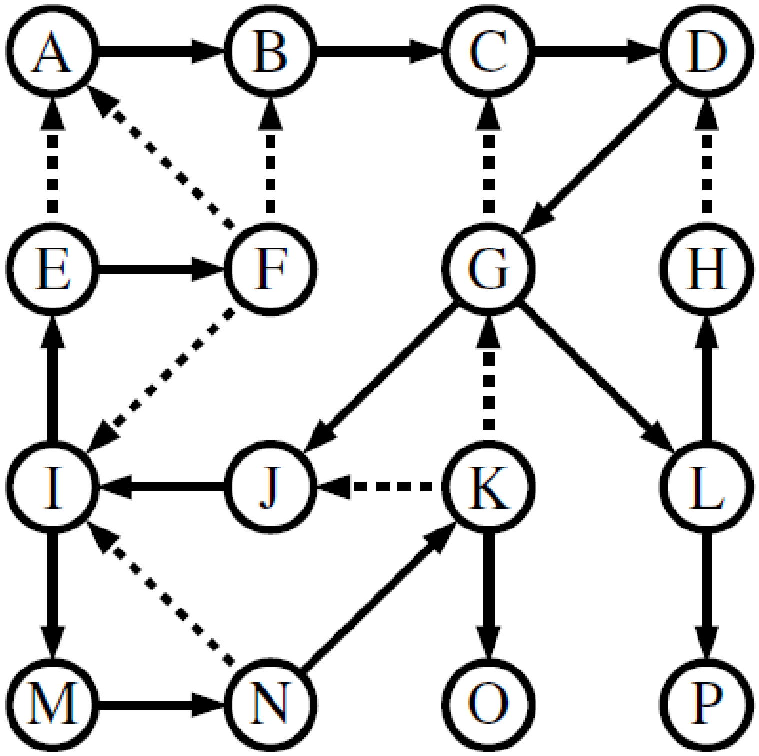


Stop at F , track back to I and resume dfs

Stop at H , track back to L and resume dfs



(e)

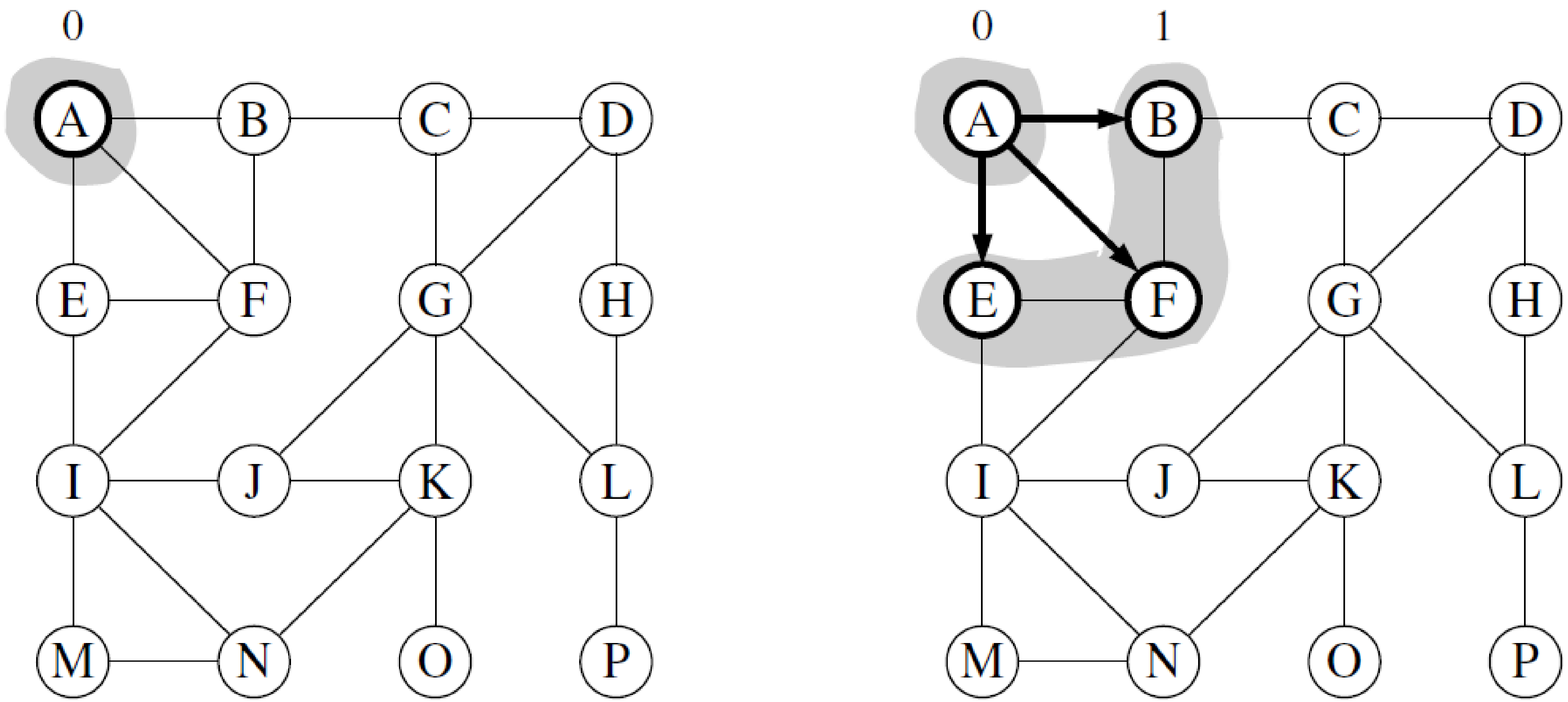


(f)

Output: [A B C D G J I E F M N K O L H P]

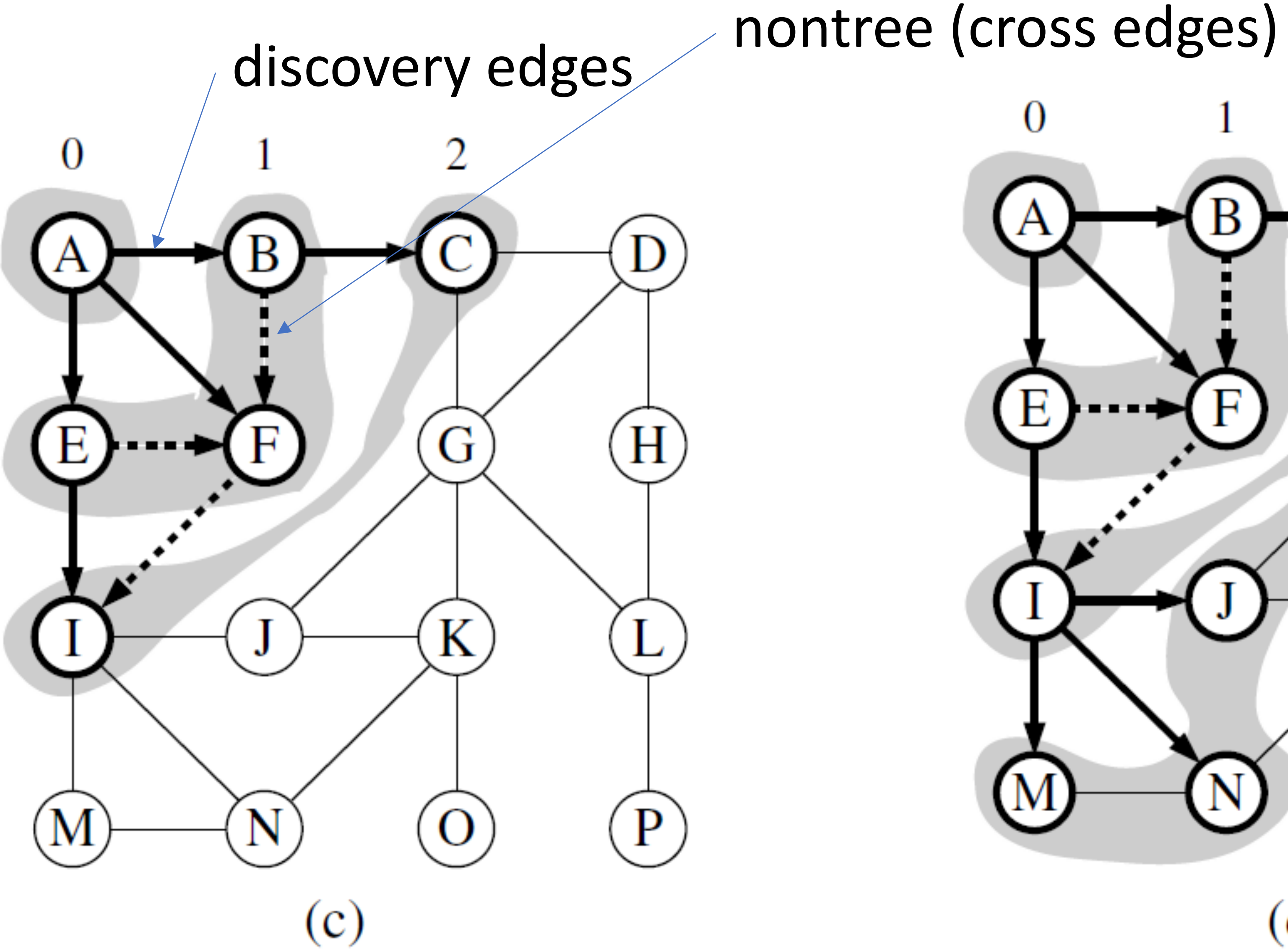
Breadth-First Search Algorithm

```
def BFS(g, s, discovered = {}):
  level = [s] # first level includes only s
  discovered[s] = None
  while len(level) > 0:
    next_level = [ ] # prepare to gather newly found vertices
    for u in level:
      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
          next_level.append(v) # v will be further considered in next pass
    level = next_level # relabel 'next' level to become current
```

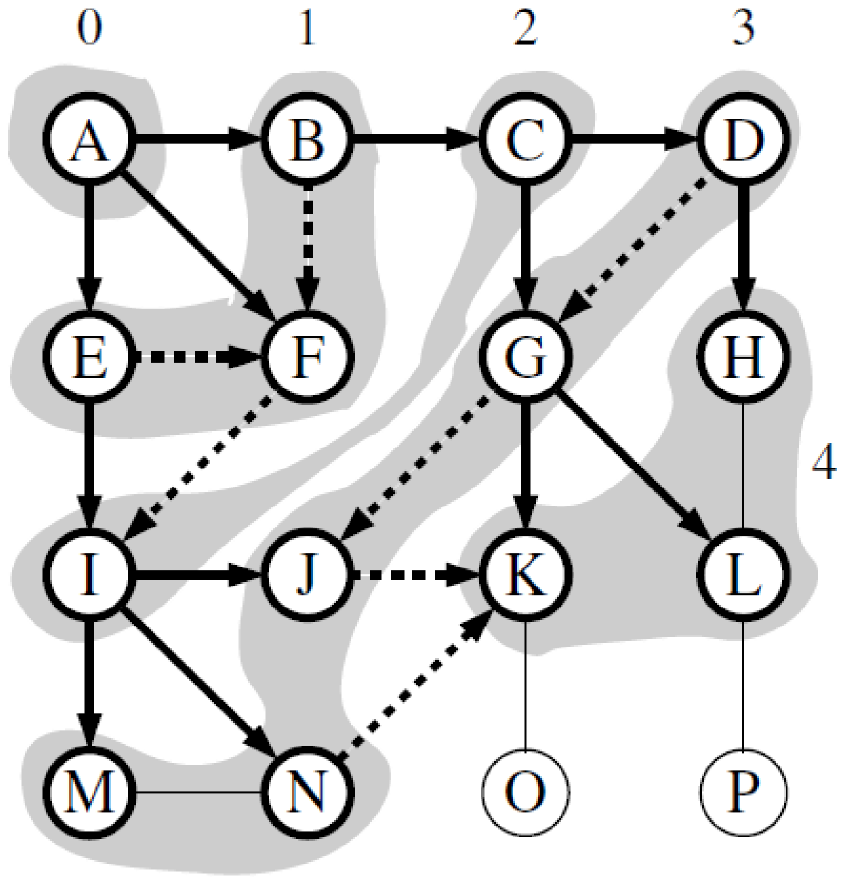


(a)

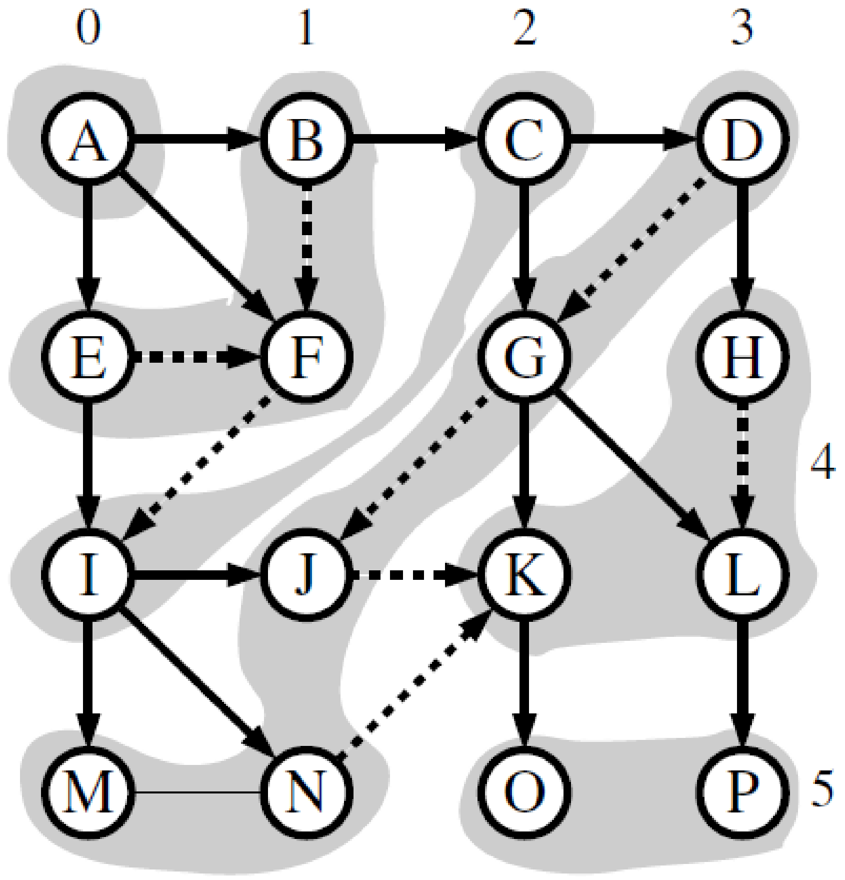
(b)



nontree (cross edges)
discovery edges
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)



(e)



(f)

From the BFS traversal algorithm,

a path in a **breadth-first search tree** rooted at **s** to **v**

=

the shortest path from **s** to **v**

***for non-weighted graph**