BREADTH-FIRST AND DEPTH FIRST SEARCH

Data Structures and Algorithms

Waheed Iqbal

Department of Data Science, FCIT
University of the Punjab, Lahore, Pakistan

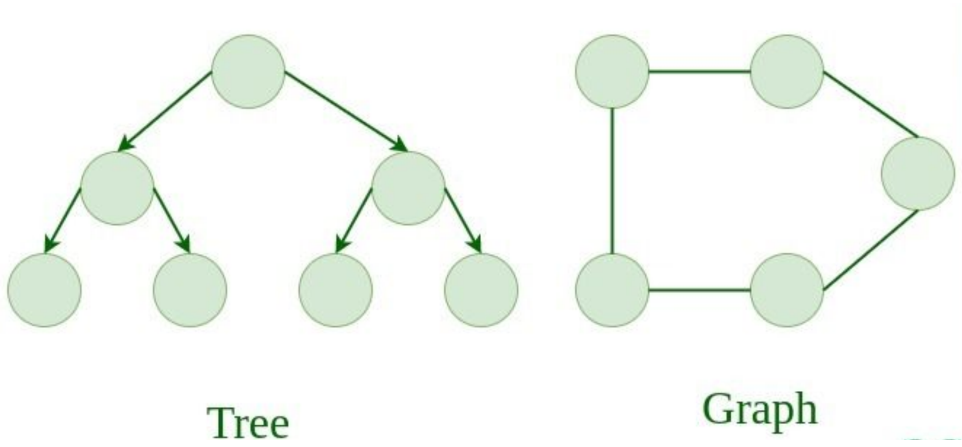
Credit

- These notes contain material from Chapter 22 of Cormen, Leiserson, Rivest, and Stein (3rd Edition).
- Lecture notes of Prof. Constantinos Daskalakis of MIT.

Tree vs Graph

- A **tree data structure** is a **hierarchical** data structure that consists of **nodes** connected by edges. Each node can have multiple child nodes, but only one parent node.

- A graph data structure is a **collection of nodes** (also called **vertices**) and **edges** that connect them. Nodes can represent entities, such as people, places, or things, while edges represent relationships between those entities.



Graph Representation

- Adjacency list and adjacency matrix may use to represent a graph **G(V, E);** where **V** and **E** represents vertices and edges respectively
- A graph could be directed or undirected
- **Sparse Graph**: number of edges (E) are minimal (|E| is much less than $|V^2|$)
- **Dense Graph**: number of edges (E) are close to maximum possible edges minimal (|E| is close to $|V^2|$)

Undirected Graph

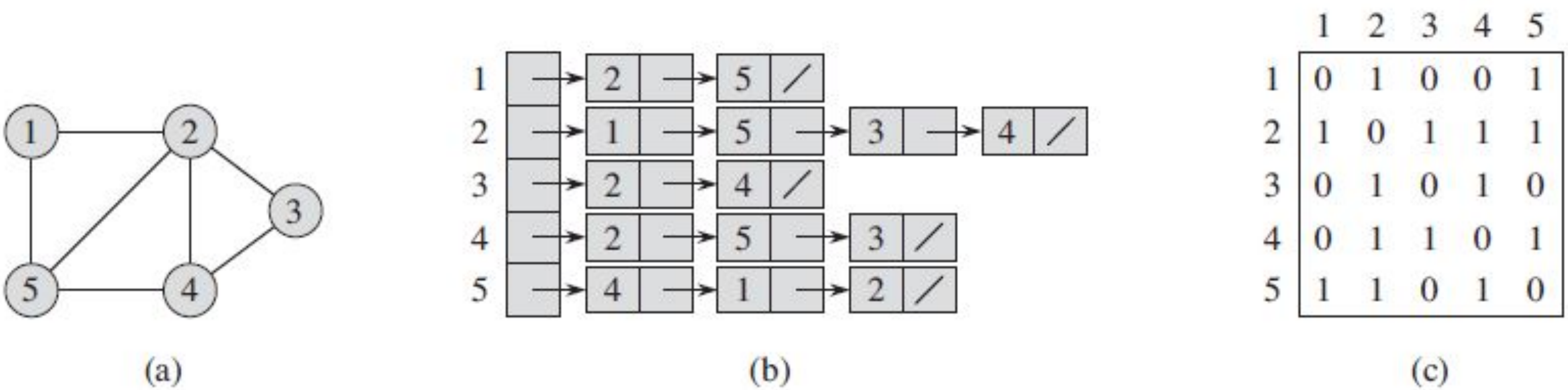
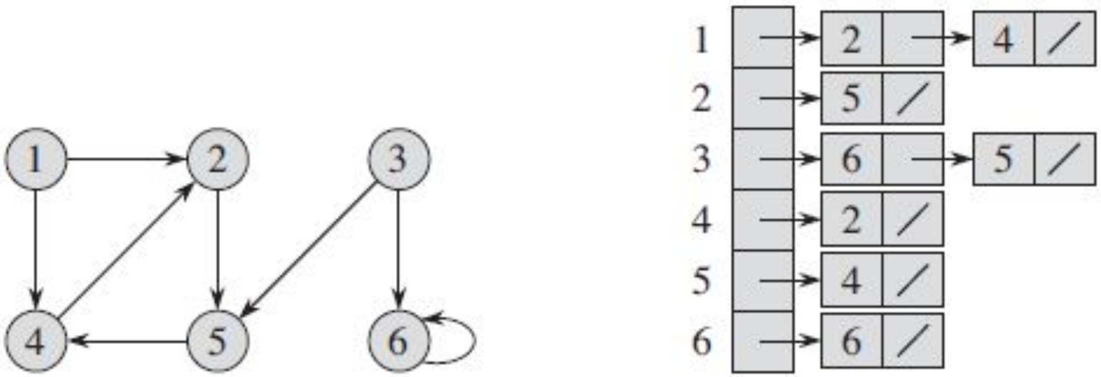


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Directed Graph



(a) (b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

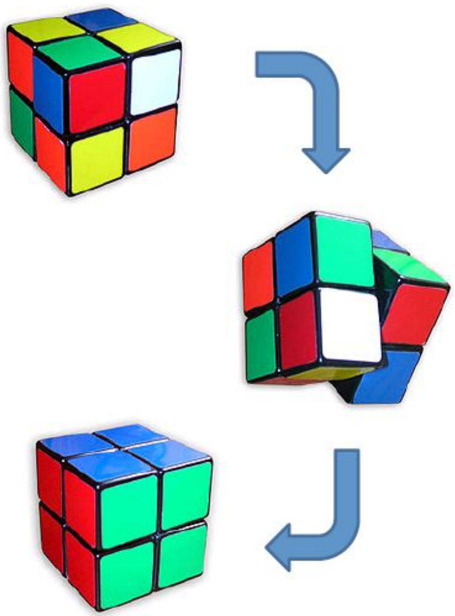
Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

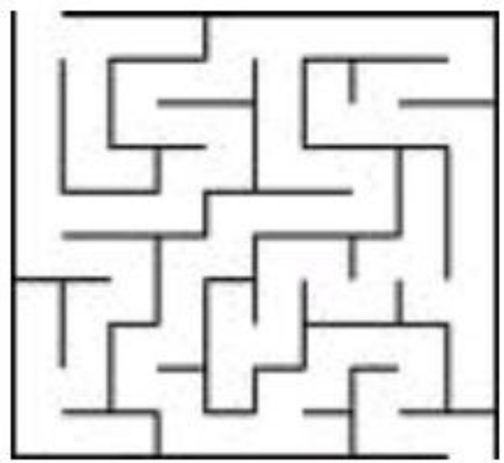
Graph Representation in Python

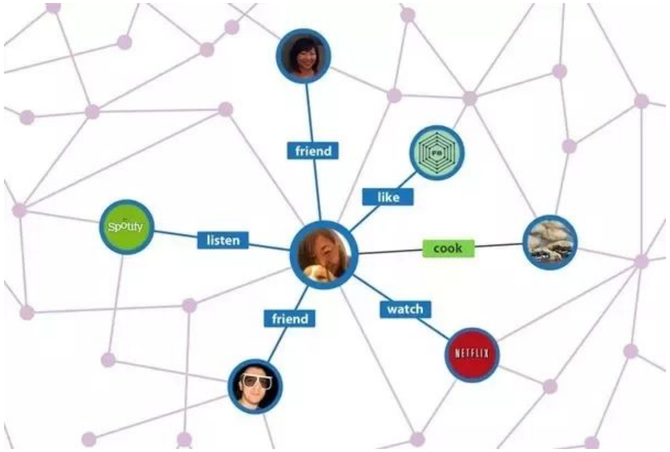
```python
# Define the graph as a dictionary
graph = {
    1: [2, 3],
    2: [1, 3, 4],
    3: [1, 2, 4],
    4: [2, 3]
}
```

```python
# Define the adjacency matrix
adj_matrix = [
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
]
```

Graphs in Action





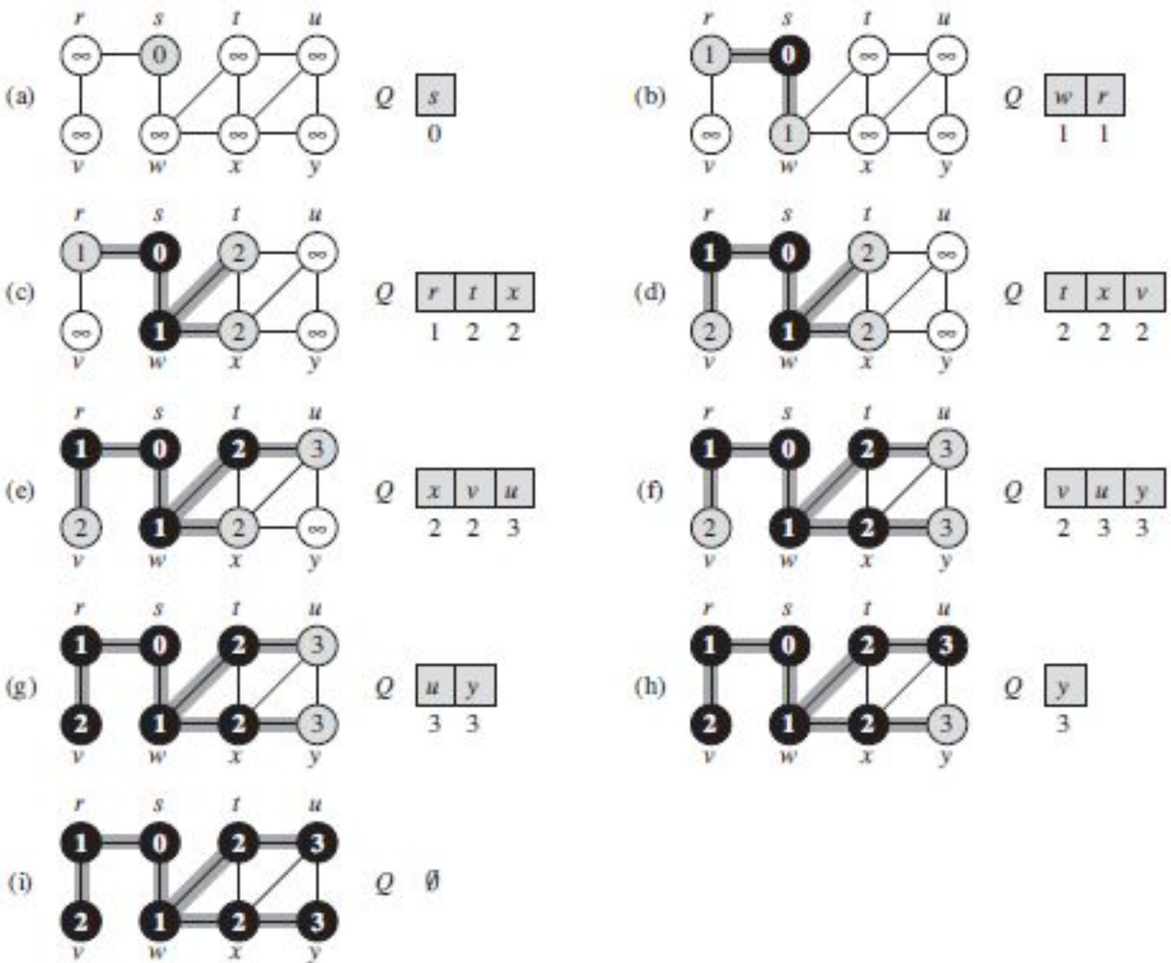


Breadth-First Search (BFS)

- One of the simplest algorithm for searching a graph
- Given a graph $G = (V, E)$ and a distinguished ***source vertex*** ***s***, ***breadth-first*** search systematically explores the edges of G to “discover” every vertex that is reachable from s
- It computes the distance (smallest number of edges) from s to each reachable vertex

Breadth-First Search (Cont.)

Algorithm




```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Breadth-First Search (Cont.)

Analysis

- Enqueuing and dequeuing take O(1)
- Total time devoted to queue operations take O(V)
- Total time scanning adjacency lists is O(E)
- Total running time of the BFS procedure is O(V +E)

Breadth-First Search (Cont.)

Shortest Path

- The procedure BFS builds a breadth-first tree as it searches the graph

- Shortest-path from ***s*** to ***v*** as the minimum number of edges in any path from vertex ***s*** to vertex ***v***;

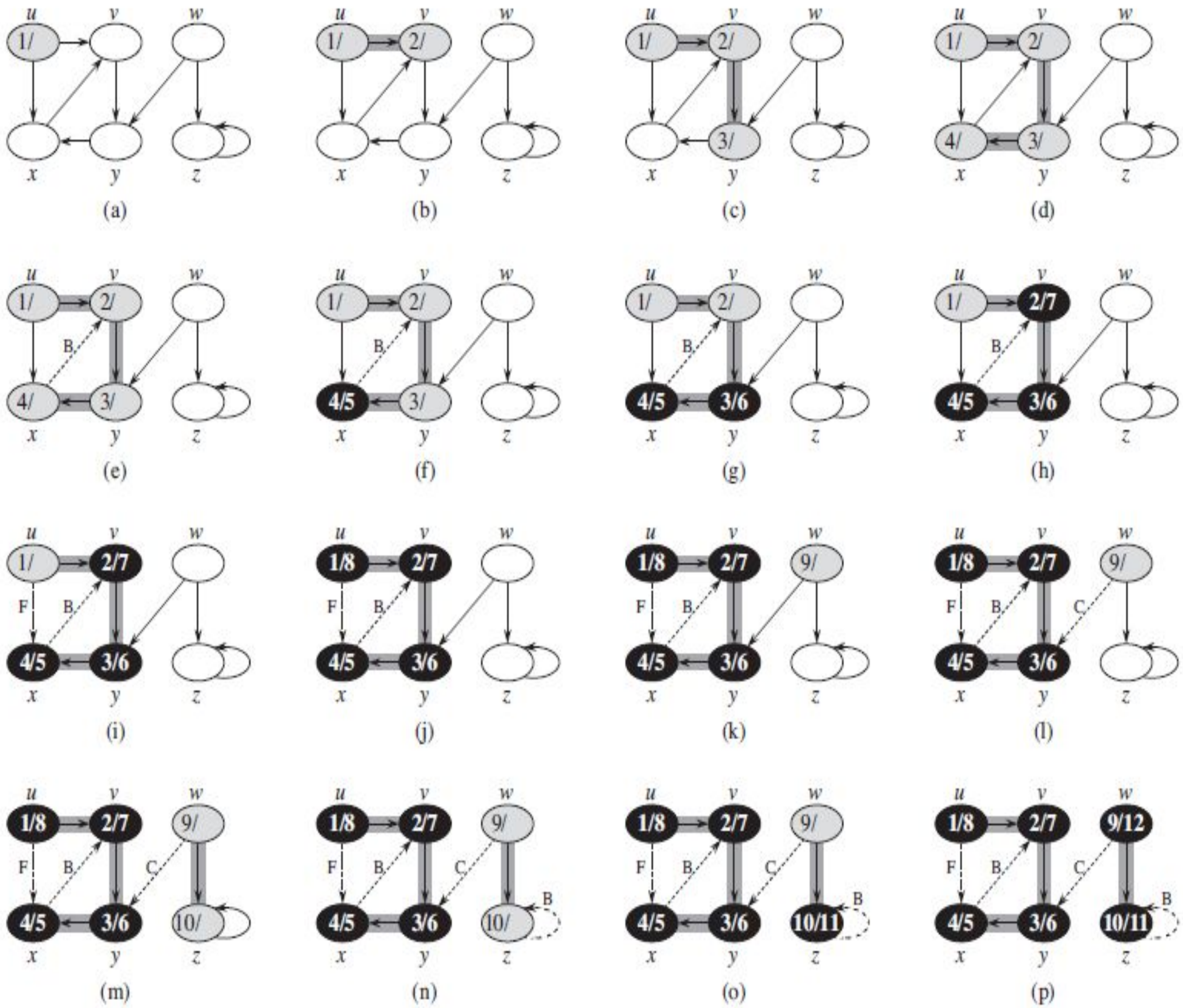
```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "no path from" s "to" v "exists"
else PRINT-PATH(G, s, v.π)
    print v
```

Depth-first Search (DFS)

- Depth-first search explores edges out of the most recently discovered vertex that still has unexplored edges leaving it.

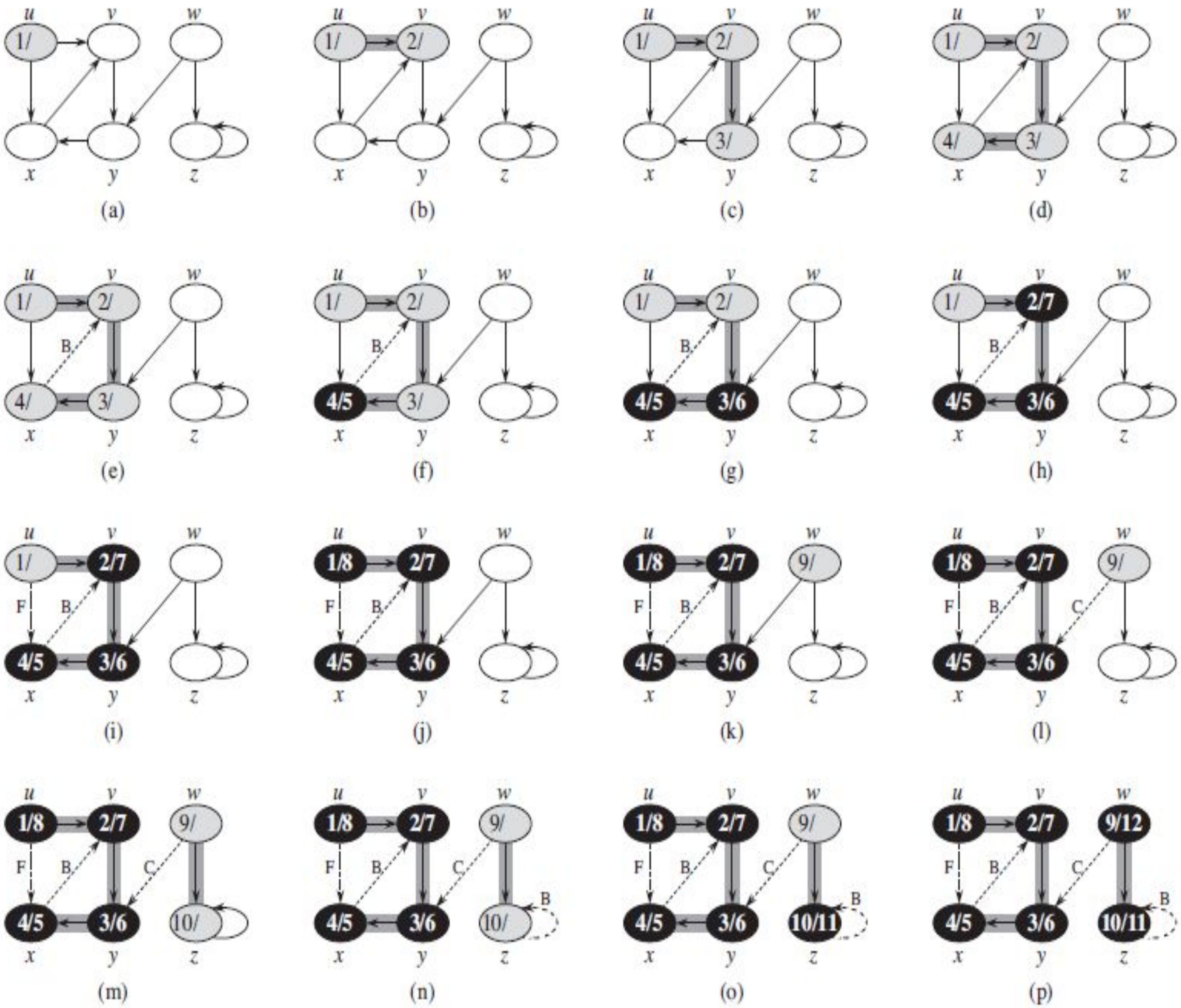
- Once all of *v*’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which was discovered.

- This process continues until we have discovered all the vertices that are reachable from the original source vertex.




```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```




```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

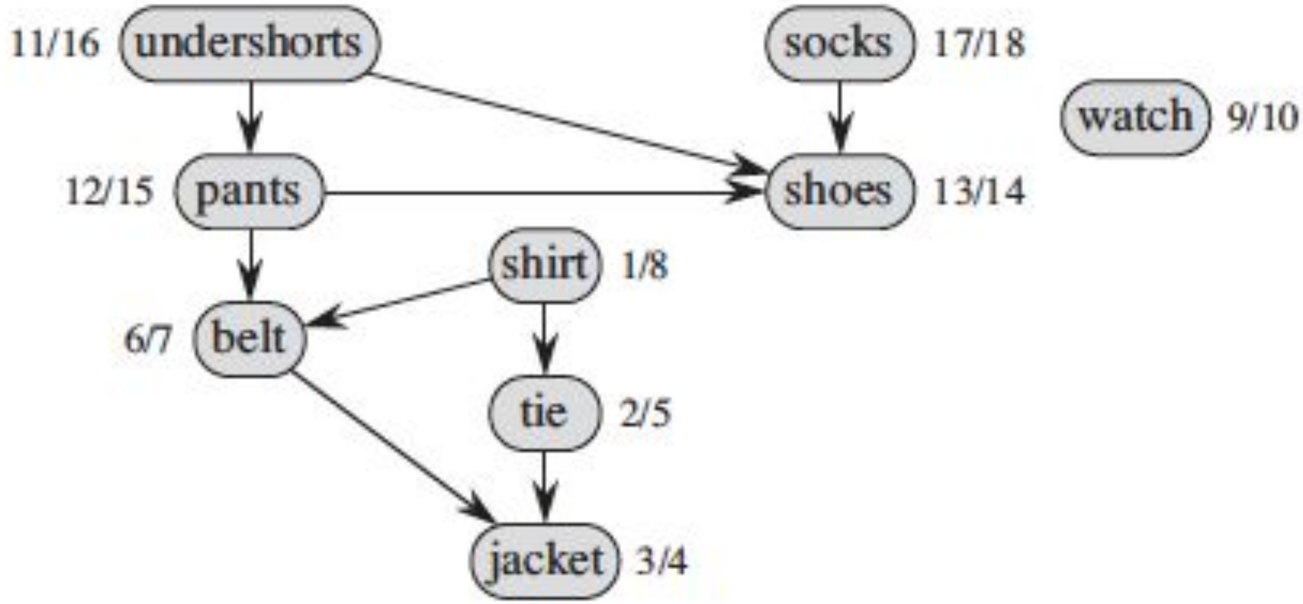
- DFS is O(V) exclusive of DFS-VIST
- DFS-VISIT is O(E)
- The running time of DFS is therefore O(V+E)

Tradeoffs

- Solving Rubik's cube?
  - BFS gives shortest solution
- Robot exploring a building?
  - Robot can trace out the exploration path
  - Just drops markers behind
- Only difference is "next vertex" choice
  - BFS uses a queue
  - DFS uses a stack (recursion)

Topological Sort

(a)




```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

(b)