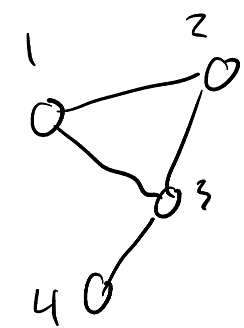
Graphs

- A graph G consists of a set of vertices V (or nodes) and a set of edges (or links, or connections) E
  - $G = (V, E)$
  - An edge (u, v) is a connection between vertices u and v
  - $|V|$ is the number of vertices
  - $|E|$ is the number of edges



- Undirected graphs
  - There is no direction to the edges

    (u, v) is the same edge as (v, u)

  - Edges are represented visually with lines

- Directed graph
  - Edges are represented by arrows
  - (u, v) is a different edge than (v, u)

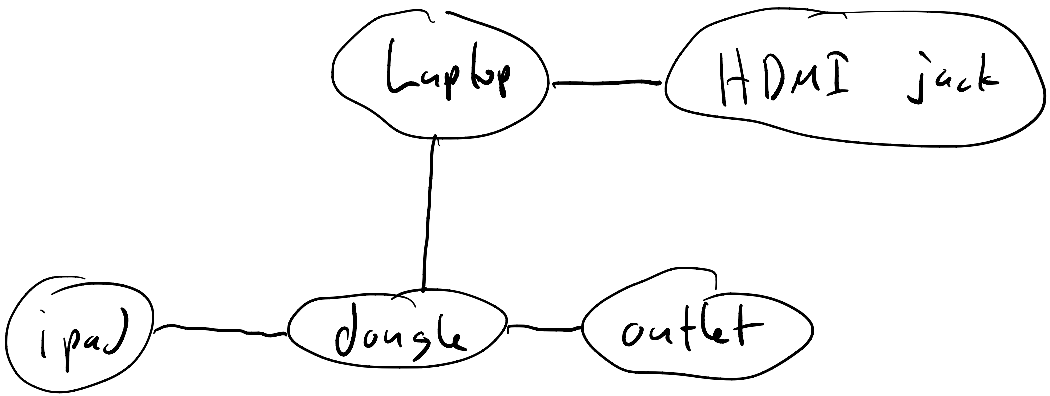
Weighted graph

- Every edge has a numerical weight

Unweighted graph

- Every edge has the same weight

My technology setup

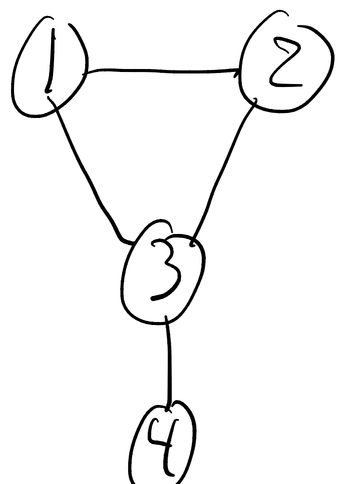


- Adjacency matrix

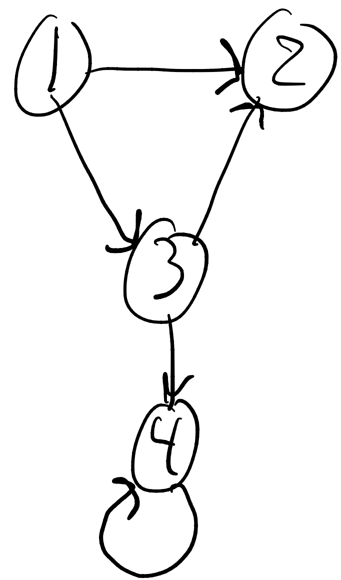
  $|V| \times |V|$ matrix

  - Each entry represents an edge (or lack of edge)

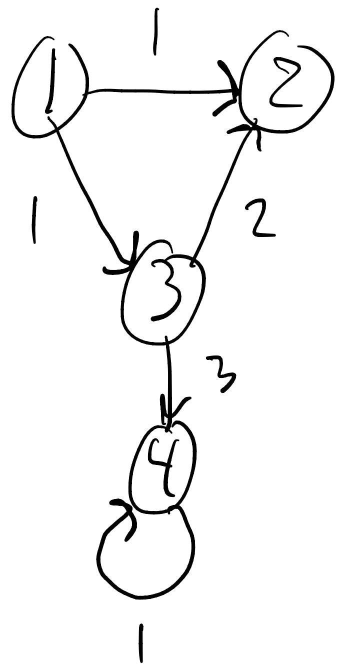
$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$



$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

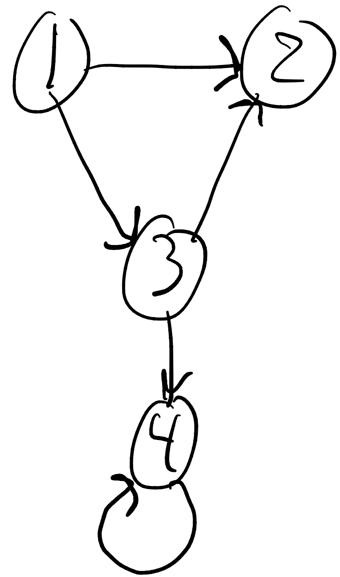


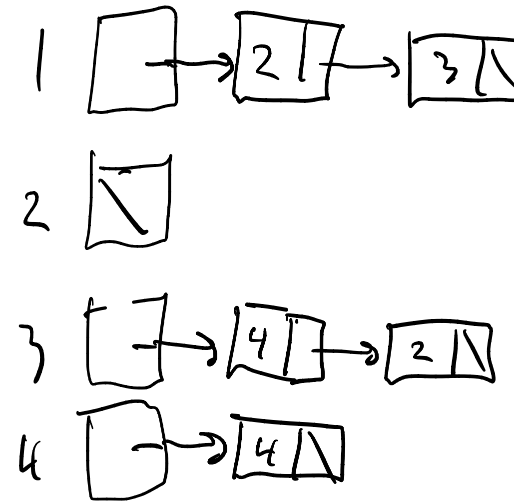
	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	1
4	0	0	0	1



	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	2	0	3
4	0	0	0	1

- Adjacency list
  - Array of $|V|$ linked lists
  - For each $u \in V$, the list $Adj[u]$ contains all vertices v such that $(u, v) \in E$
  - Directed graph
    - Sum of the lengths of the lists is $|E|$
  - Undirected graph
    - Sum of lengths is $2 \cdot |E|$


1
2
3
4


1
2
3
2
3
4
2
4
4

book uses $O(V)$ or $O(E)$
not $O(|V|)$ or $O(|E|)$

- Sparse vs dense graphs

  - Number of edges in a graph is $O(V^2)$ (upper bound)

  - If $|E|$ is close to $|V|^2$, the graph is very dense

  - A graph is sparse if $|E|$ is much less than $|V|^2$

- An adjacency matrix always requires $|V|^2$ entries
  - For sparse graphs this is not space efficient
  - Time/space tradeoff

- Lookup time

  $O(1)$ for adj. matrix

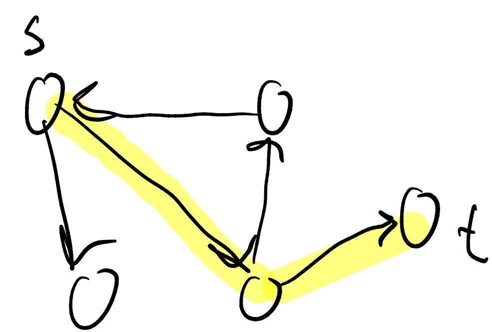
  $O(E)$ and $O(V)$ for adj. list

- Space

  $\Theta(V^2)$ for matrix

  $\Theta(E + V)$

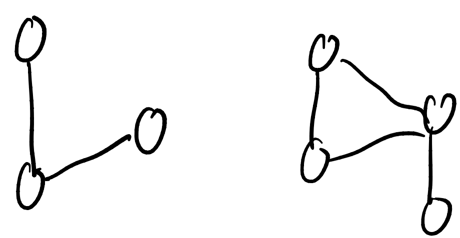
- Paths
    - A path from u to v $u \rightsquigarrow v$ is a sequence of edges connecting u and v
    - In a simple path, no vertex appears more than once (no cycles)



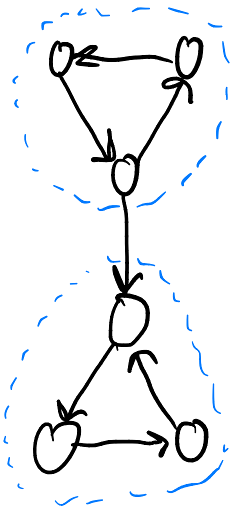
- Connected component

  - A subgraph wherein each pair of nodes is connected via a path
  - Every node is reachable from every other node
  - A graph may have multiple connected components

2 connected components

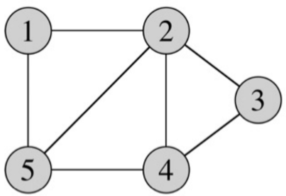


- Directed graphs
  - Strongly connected component
    - Every node is reachable from every other node
  - Weakly connected component
    - Would be strongly connected if every edge was undirected

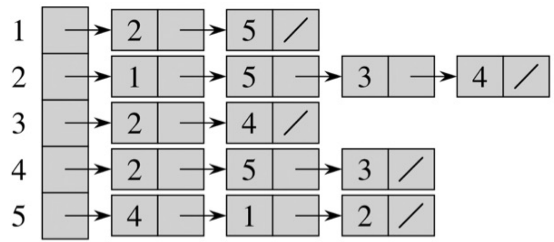


2 strongly connected components

1 weakly connected component



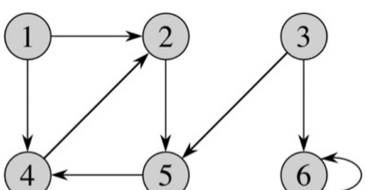
(a)



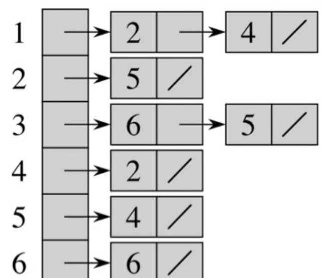
(b)

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

(c)



(a)



(b)

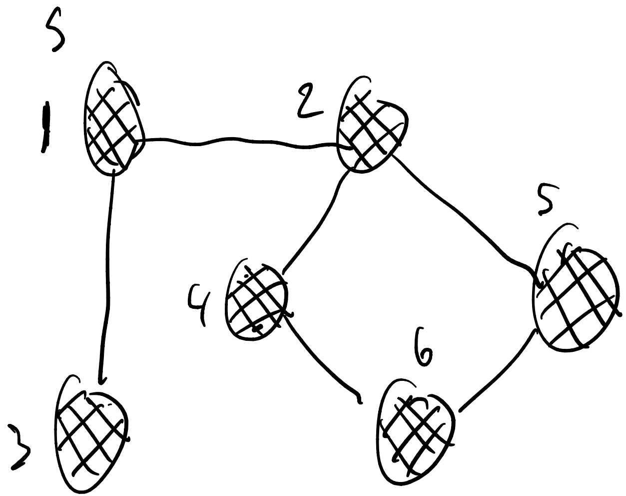
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

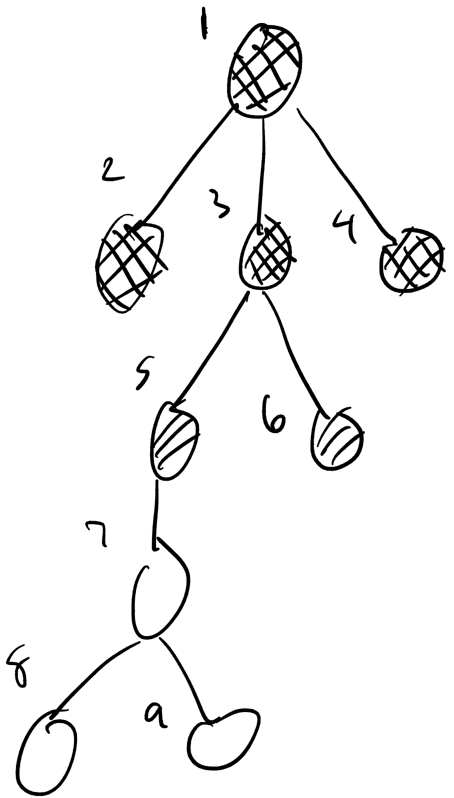
(c)

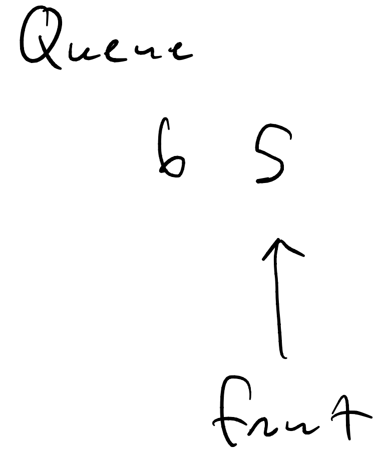
Breadth-first Search

- Visits a vertices that are reachable from a source vertex s
- Finds vertices that are 1 edge away from s, then those that are 2 edges away, etc.
- As the algorithm runs, each vertex is in one of 3 states
  - White - undescovered
  - Grey - discovered, but has undescovered neighbors
  - Black - discovered, and all neighbors discovered

Queue

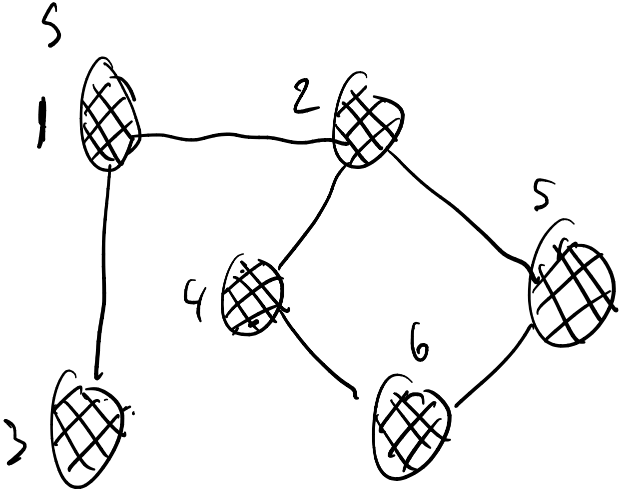




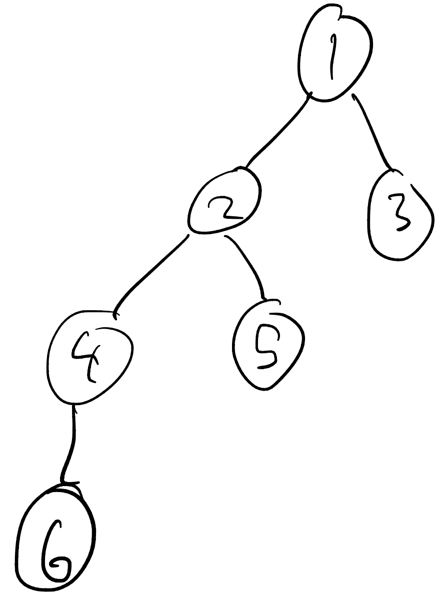


BFS on a tree visits nodes layer by layer

- BFS predecessors
    - All nodes except s are found via predecessors
    - The graph that shows predecessor relationships is a tree rooted at s
        - The BFS tree
    - The shortest path from s to any vertex v is the path from s to v in the BFS tree

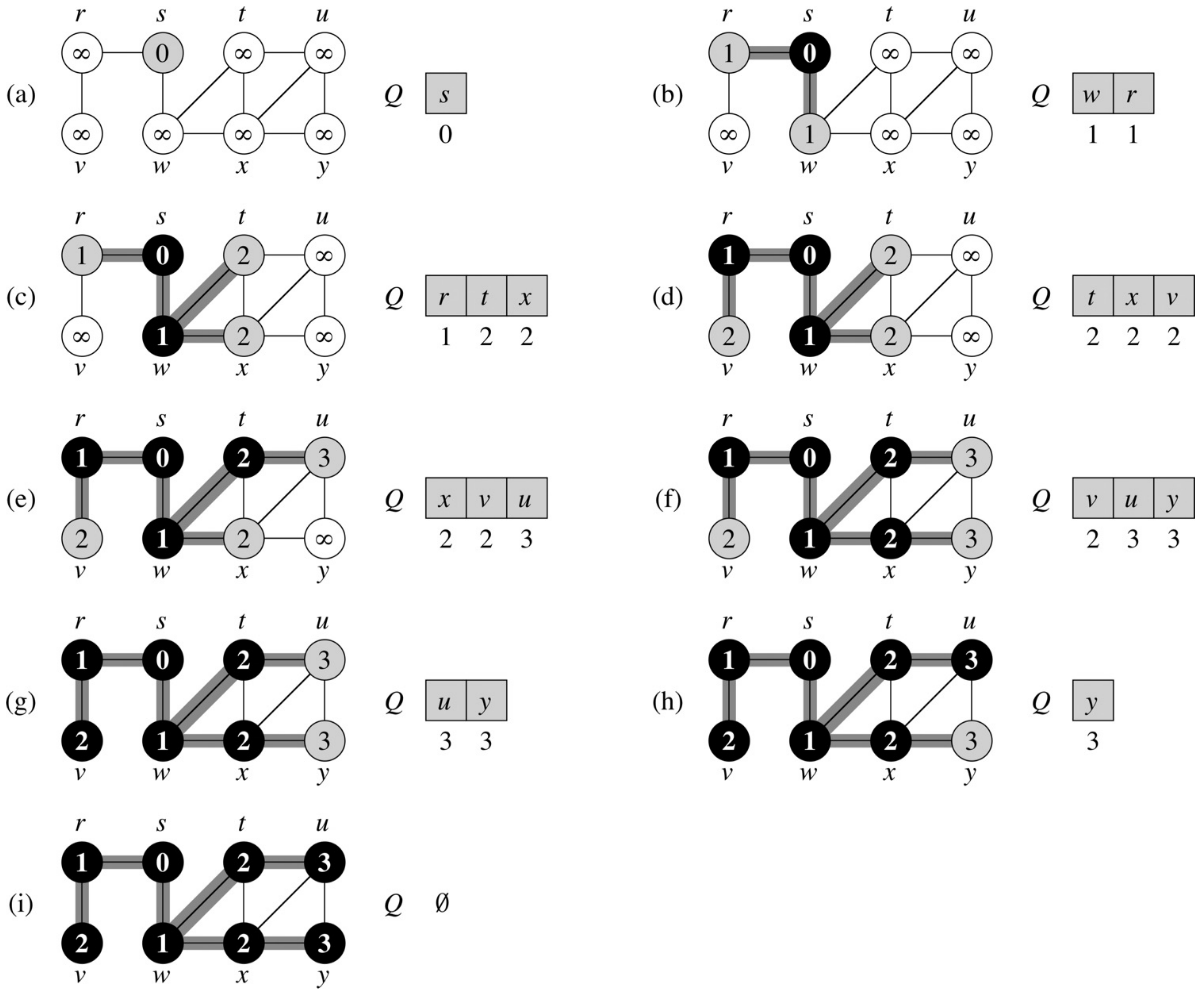


BFS tree



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

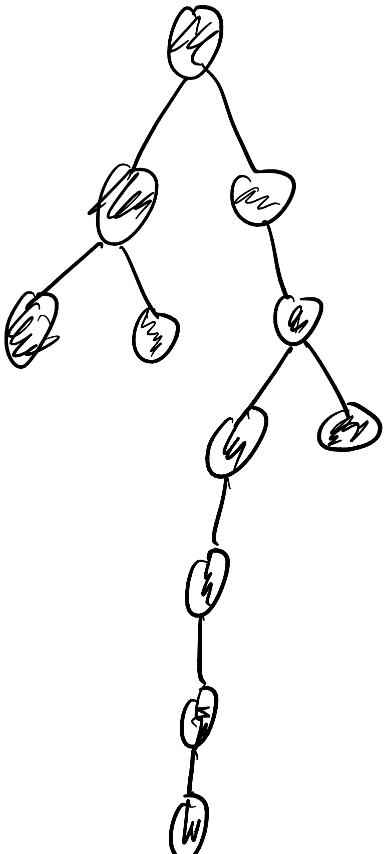
- BFS time complexity
    - Every node is enqueued at most once (when it is white)
    - A node turns grey when it is enqueued
    - All edges are checked at most twice (for undirected graphs)
    - $O(V + E)$
        - Not Θ because the graph might not be connected



(a)
r s t u
∞ 0 ∞ ∞
∞ ∞ ∞ ∞
v w x y
Q s
0
(b)
r s t u
1 0 ∞ ∞
∞ 1 ∞ ∞
v w x y
Q w r
1 1
(c)
r s t u
1 0 2 ∞
∞ 1 2 ∞
v w x y
Q r t x
1 2 2
(d)
r s t u
1 0 2 ∞
2 1 2 ∞
v w x y
Q t x v
2 2 2
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q ∅

Depth First Search (DFS)

- Works its way deeper into a graph until it reaches a node with no undiscovered neighbors, then backtracks
    - Use recursion or a stack to keep track of where to backtrack



DFS(G)

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

u.d is the discovery time of u

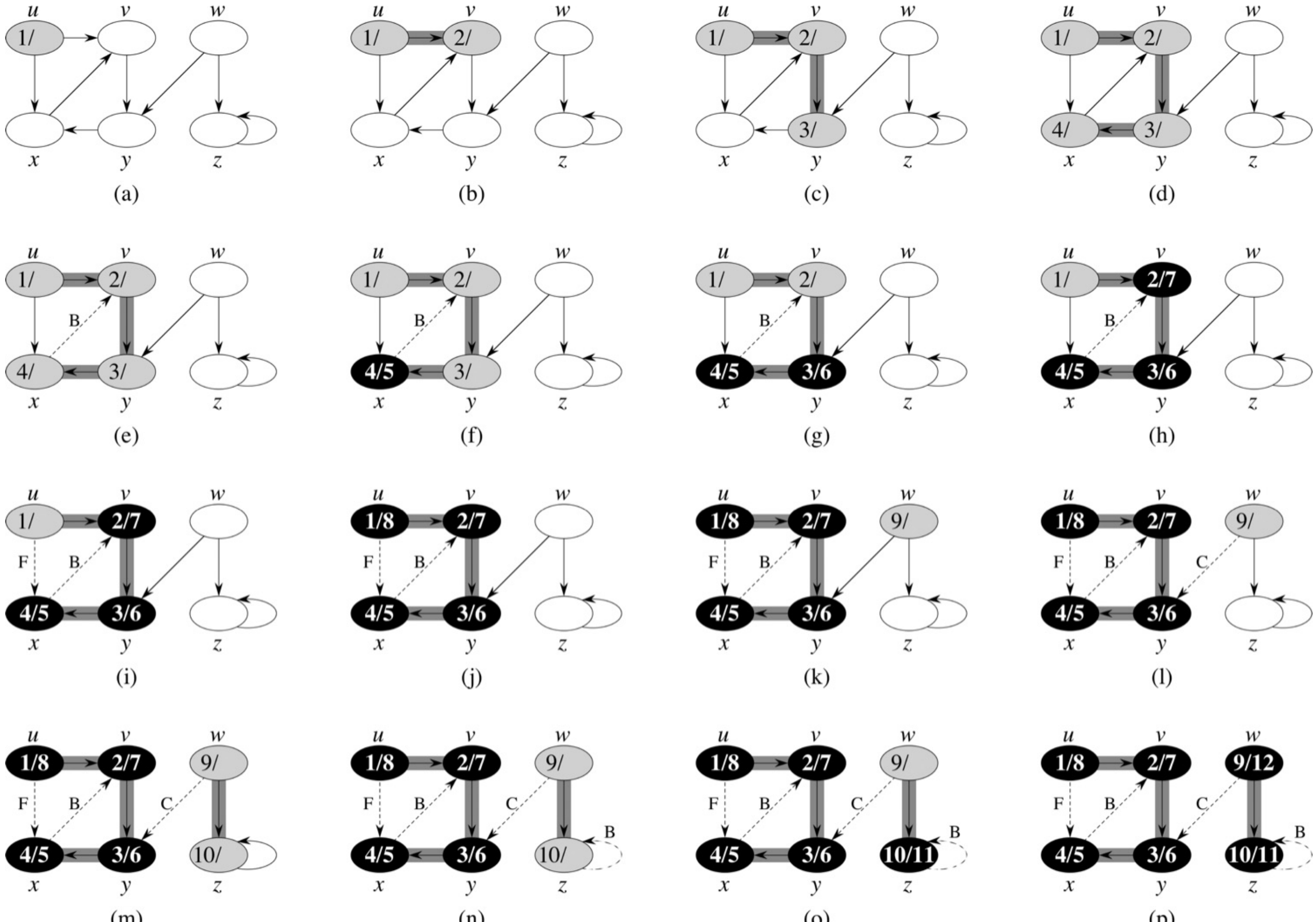
u.f is the finishing time

$\Theta(V + E)$ time

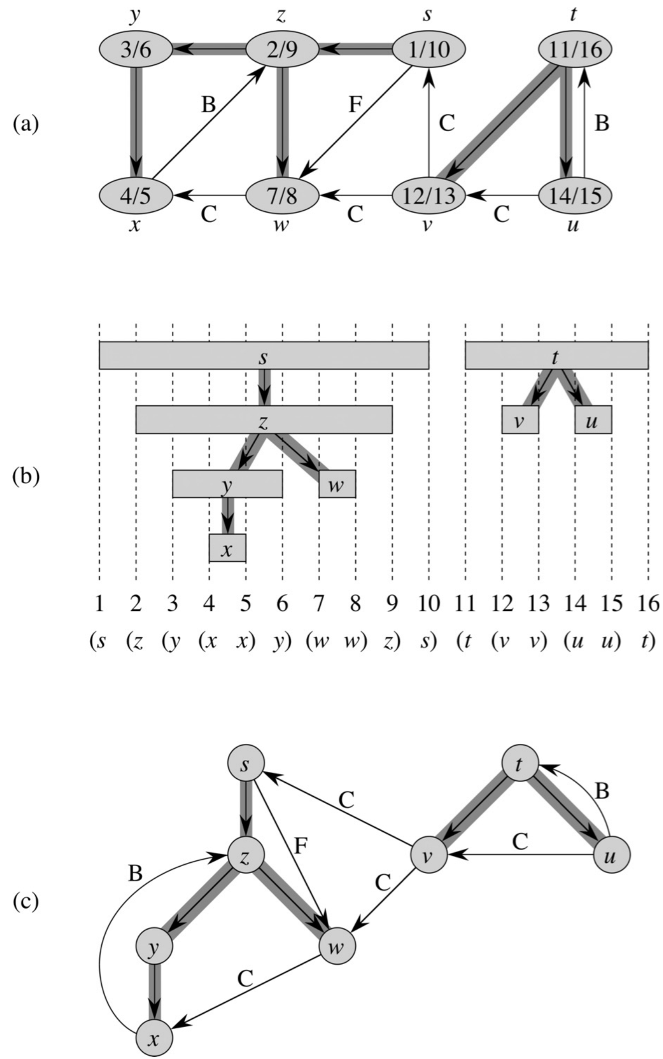
- DFS-Visit is called once for every vertex
- Every edge is checked once for directed graphs, twice for undirected

DFS-VISIT(G, u)

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

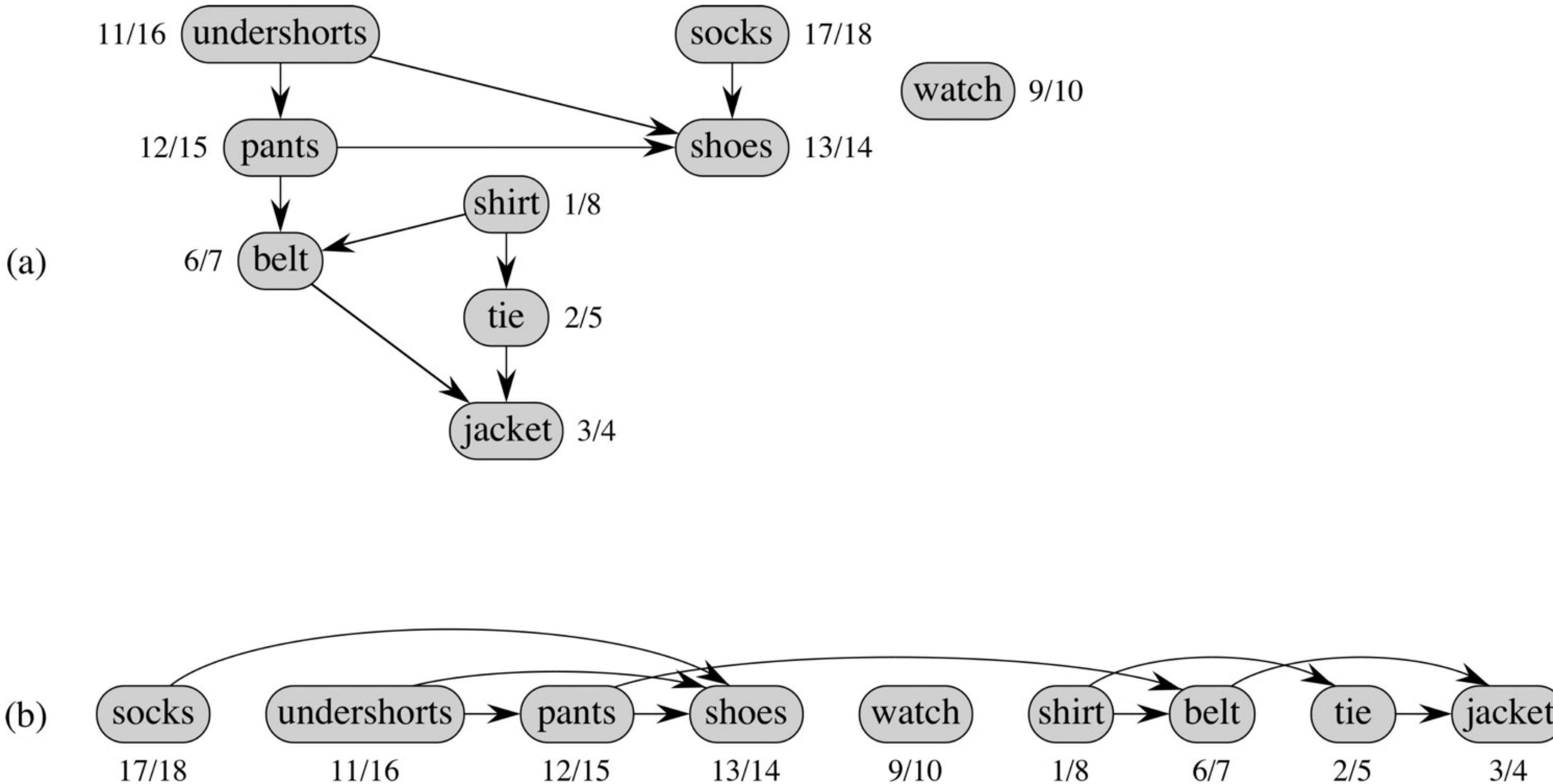


u v w
1/
x y z
(a)
u v w
1/ 2/
x y z
(b)
u v w
1/ 2/
3/
x y z
(c)
u v w
1/ 2/
4/ 3/
x y z
(d)
u v w
1/ 2/
B
4/ 3/
x y z
(e)
u v w
1/ 2/
B
4/5 3/
x y z
(f)
u v w
1/ 2/
B
4/5 3/6
x y z
(g)
u v w
1/ 2/7
B
4/5 3/6
x y z
(h)
u v w
1/ 2/7
F B
4/5 3/6
x y z
(i)
u v w
1/8 2/7
F B
4/5 3/6
x y z
(j)
u v w
1/8 2/7 9/
F B
4/5 3/6
x y z
(k)
u v w
1/8 2/7 9/
F B C
4/5 3/6
x y z
(l)
u v w
1/8 2/7 9/
F B C
4/5 3/6 10/
x y z
(m)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/
x y z
(n)
u v w
1/8 2/7 9/
F B C B
4/5 3/6 10/11
x y z
(o)
u v w
1/8 2/7 9/12
F B C B
4/5 3/6 10/11
x y z
(p)


(a)
y
z
s
t
3/6
2/9
1/10
11/16
B
F
C
B
4/5
7/8
12/13
14/15
C
C
C
x
w
v
u
(b)
s
t
z
v
u
y
w
x
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)
(c)
s
t
C
B
F
C
z
v
u
B
C
y
w
C
x

- Topological sort
  - Uses DFS to form a linear ordering of vertices
  - Most useful for directed acyclic graphs (DAGs)
    - If edge (u, v) is in the graph, u comes before v in the ordering

- In a graph representing dependencies of events, a topological sort gives an order in which to carry out events
- Perform a DFS
  - When finished with a vertex, put it at the front of a list


(a)
11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4
(b)
socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4