Elementary Graph Algorithms

ITX2010, CSX3003, IT2230

Data Structures and Algorithms, Information Structures

Learning Objectives

Students will be able to:

- Understand two most common computational representations of graphs
- Explain how to represent graph
- Describe two graph-searching algorithms and how to create them
- Present graph-search and prove result about the order in which each search visits vertices
- Application of depth-first search – Topologically sorting a directed acyclic graph

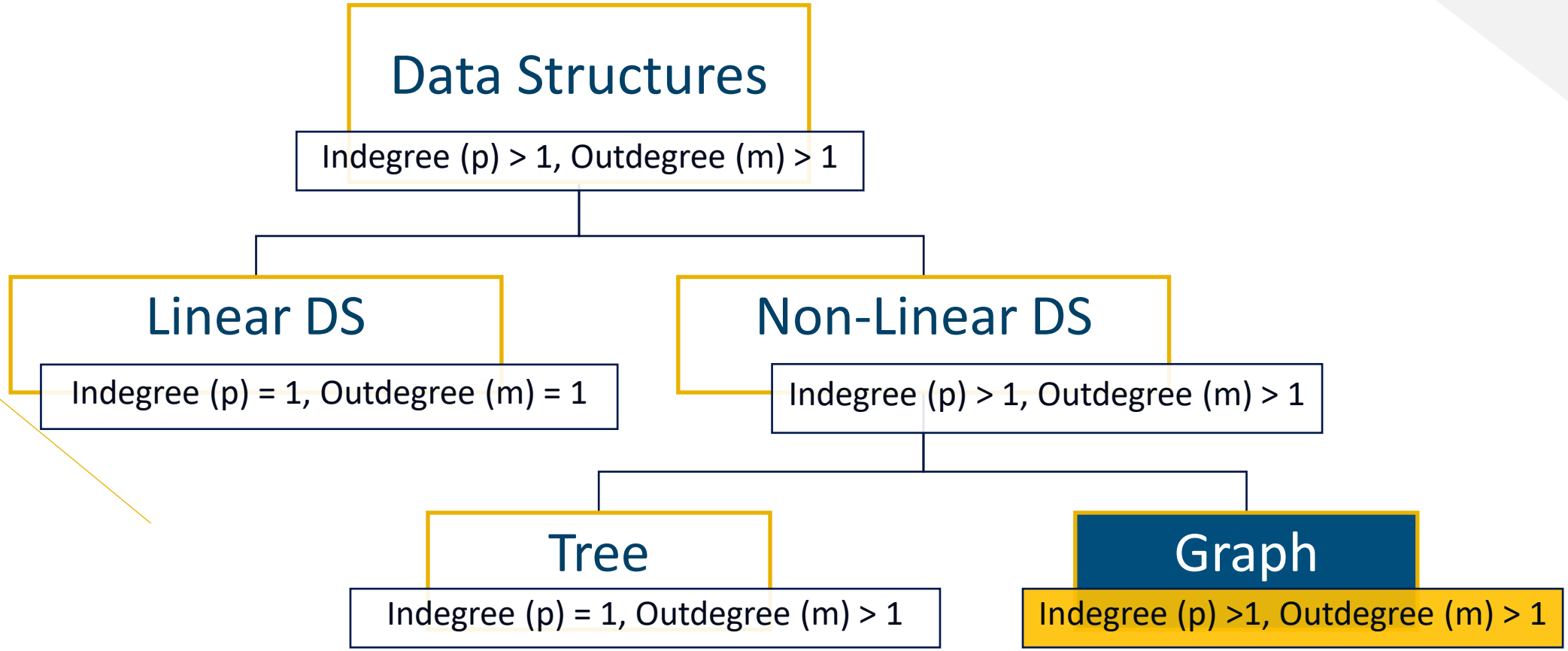
Chapter Outline

1. Representations of Graphs
   1) Adjacency-list representation
   2) Adjacency-matrix representation
2. Breadth-First Search (BFS)
3. Depth-First Search (DFS)
4. DFS Applications

22.1

Representations of Graphs

Graphs


Data Structures
Indegree (p) > 1, Outdegree (m) > 1
Linear DS
Indegree (p) = 1, Outdegree (m) = 1
Non-Linear DS
Indegree (p) > 1, Outdegree (m) > 1
Tree
Indegree (p) = 1, Outdegree (m) > 1
Graph
Indegree (p) >1, Outdegree (m) > 1

Undirected Graph

- It is a graph in which each edge has a direction to its successor.
- The flow between two vertices can go in either direction.
- An edge (u, v) – an undirected of the pair (u, v), is not ordered.

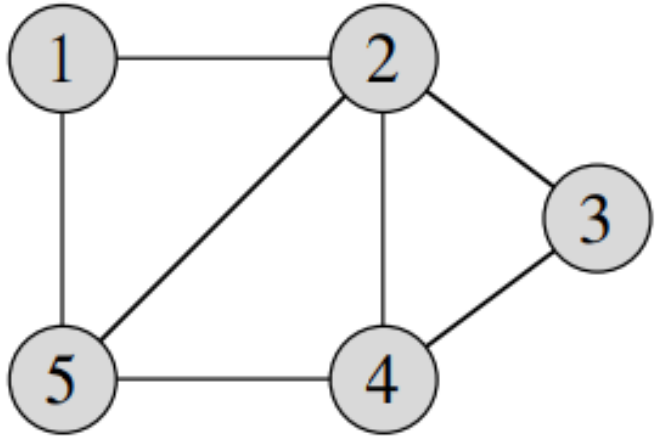


Fig 22.1 (a) An undirected graph [1]

Undirected Graph

- An undirected graph G = (V,E):
  - The vertex set V consists of its elements - called vertices.
  - The edge set E consists of unordered pairs of vertices, rather than ordered pairs – called as edges which each notated as (u, v).
  - Every edge consists of exactly two distinct vertices and self-loop is forbidden.

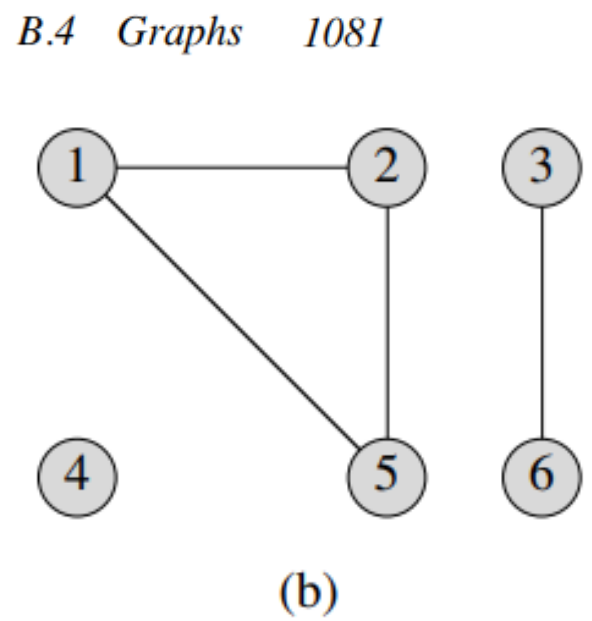


Fig B.2 (b) An undirected graph [1]

Undirected Graph

- Figure 22.1 (a) / p.528, [1]
  - An undirected graph G = (V,E) consists of:
  - 5 vertices (|V|=5) -> |V| = {1, 2, 3, 4, 5}
  - 7 edges (|E|=14 but Illustrated as 7 edges) -> |E| = {(1,2), (1,5), (2,1),(2,3), (2,4),(2,5),(3,2),(3,4),(4,2),(4,3),(4,5),(5,1),(5,2),(5,4)}

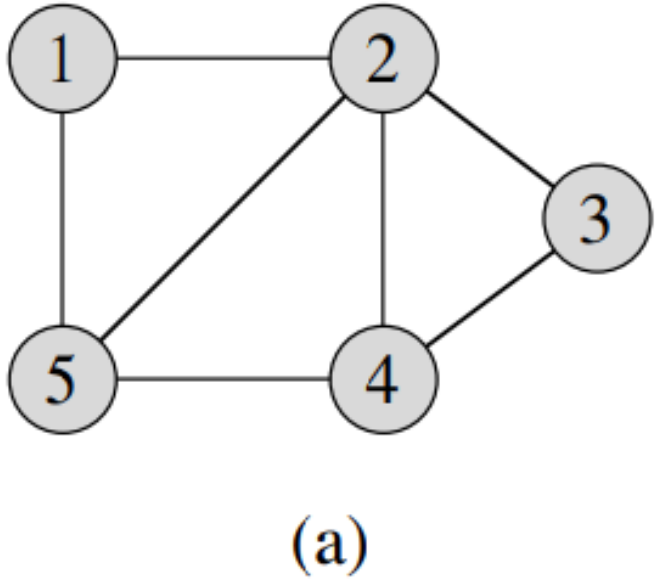


Fig 22.1 (a) An undirected graph [1]

Directed Graph

- It is sometime called “digraph”.
- It is a graph in which each edge has a direction to its successor.
- The flow along the edge between two vertices can follow only the indicate direction.
- An edge (u, v) – a directed of the pair (u, v), is ordered.

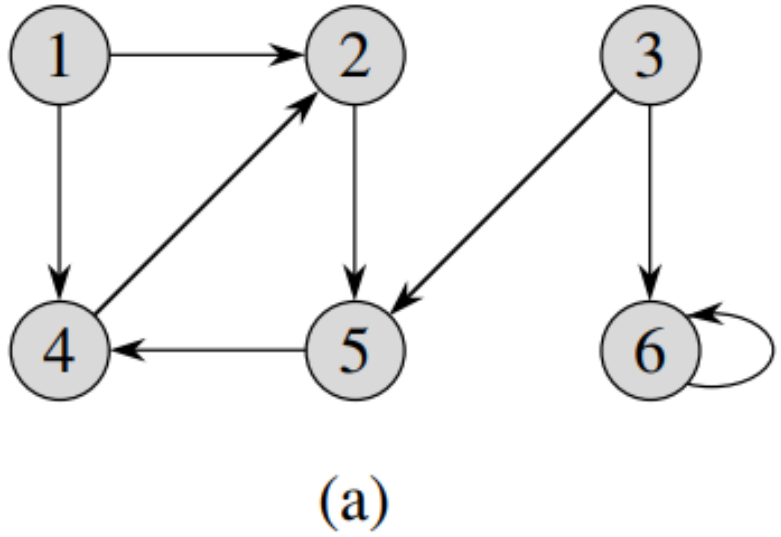


Fig 22.2 (a) A directed graph [1]

Directed Graph

- An undirected graph G = (V,E):
  - The vertex set V consists of its elements - called vertices.
  - The edge set E consists of unordered pairs of vertices, rather than ordered pairs – called as edges which each notated as (u, v).
  - Self-loop is possible.

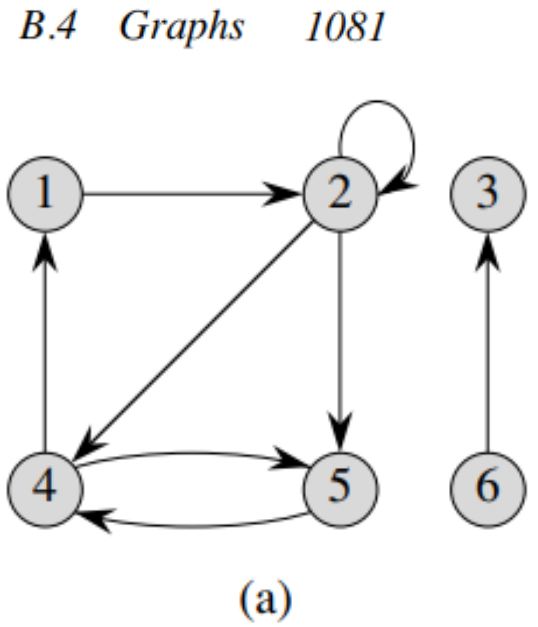


Fig B.2 (a) An undirected graph [1]

Directed Graph

- Figure 22.2 (a) / p.528, [1]
  - A directed graph G = (V,E) consists of:
  - 6 vertices (|V|=6) -> |V| = {1, 2, 3, 4, 5, 6}
  - 8 edges (|E|= 8) -> |E| = {(1,2), (1,4),(2,5),(3,5),(3,6),(4,2),(5,4),(6,6)}

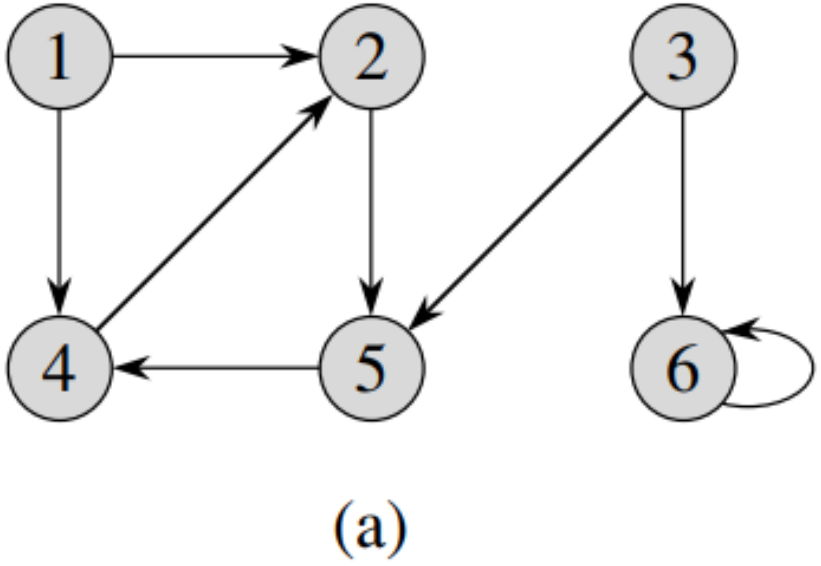


Fig 22.2 (a) A directed graph [1]

Adjacent Vertices

- If (u, v) is an edge in a graph G = (V, W), vertex b is adjacent to vertex u. [1]
- Two vertices are adjacent vertices (or neighbors) if there is an edge that directly connects pairs of those two nodes. [2]
- When the graph is directed the adjacency relation is not necessarily symmetric.
- Ex: 1 and 2 are adjacent while 2 and 3 are not.

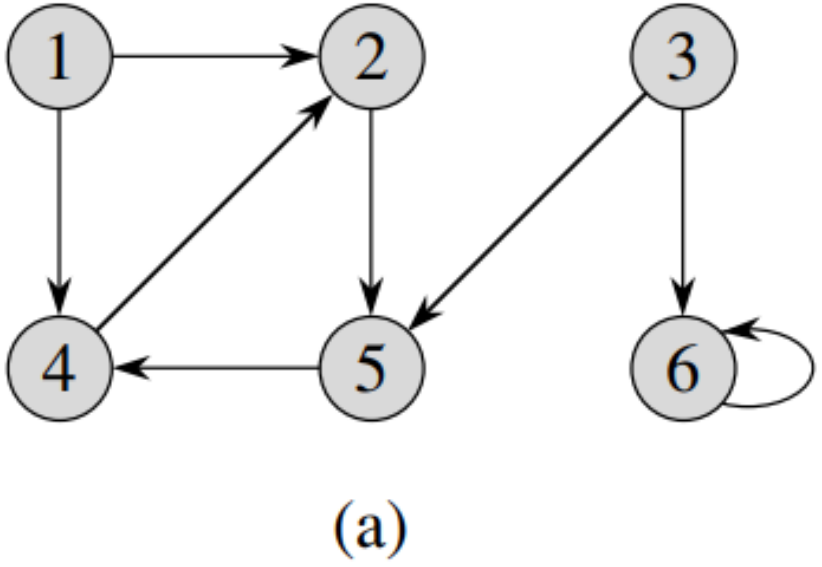


Fig 22.2 (a) A directed graph [1]

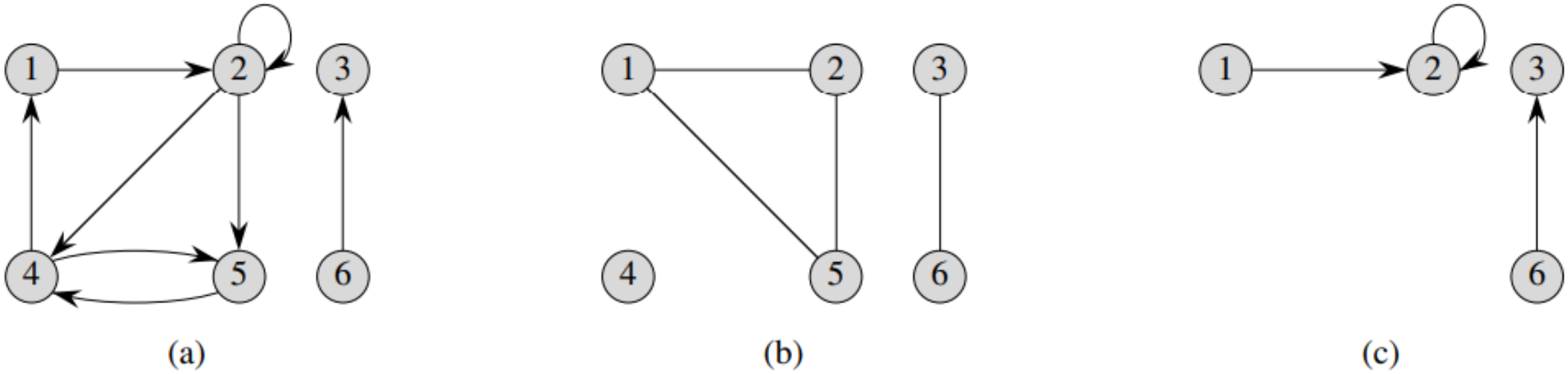


Figure B.2 Directed and undirected graphs. **(a)** A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. The edge $(2, 2)$ is a self-loop. **(b)** An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. The vertex 4 is isolated. **(c)** The subgraph of the graph in part (a) induced by the vertex set $\{1, 2, 3, 6\}$. [1]

Graph's Degree

- **The degree of a vertex** in an undirected graph is the number of edges incident on it. [1]
- **Degree** of vertex is the number of line incident to it. [2]

- Ex: Degree of vertex 1 is 2, 2 is 4.

.

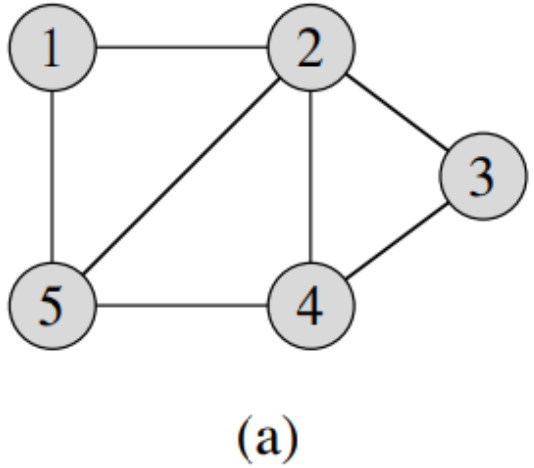
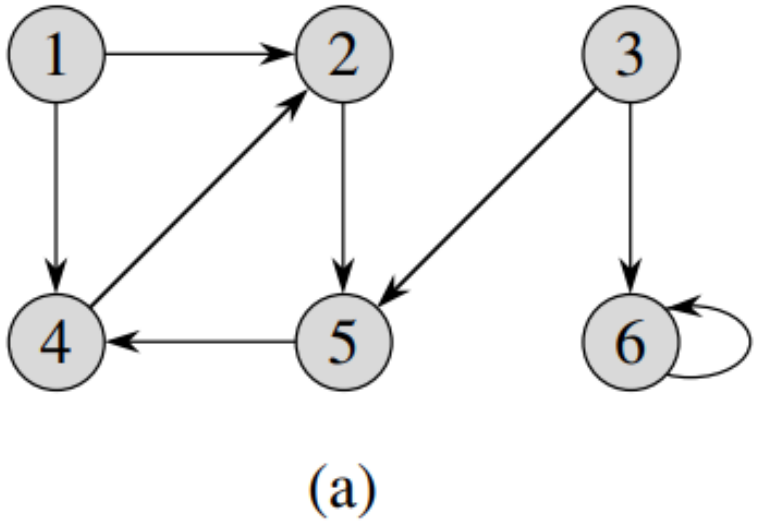


Fig 22.1 (a) An undirected graph [1]

Graph's Degree

- **Out degree** of a vertex in a graph is the number of arcs leaving the vertex.
- **In degree** of a vertex is the number of arcs entering the vertex.

- Ex: Indegree of vertex 5 is 2 and its outdegree is 1.



(a)

Fig 22.2 (b) A directed graph [1]

Graph's Degree

- A vertex whose degree 0 is isolated. [1]

- Ex: Vertex 4 in Fig B.2 (b), is isolated.

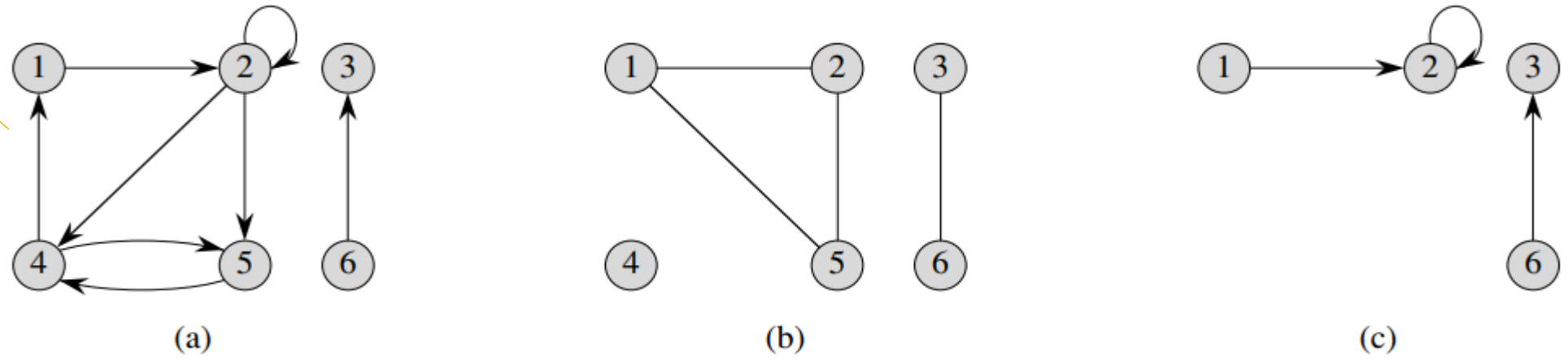


Fig B.2 Directed and Undirected graphs **[1]**

Graph's Path

- Given a path from a vertex u to a vertex u' is a sequence $<v_0, v_1, \ldots, v_k>$ of vertices ($u=v_0$, $u'=v_k$),
  - **the length of path** is the number of edges in the path.
  - u' is **reachable** from u via p if G is directed.

- The path $<v_0, v_1, \ldots, v_k>$ forms a **cycle** if $v_0 = v_k$ and the path contains at least one edge.
  - A self-loop is a cycle of length 1, like the cycle <2, 2> formed by the edge (2, 2).
  - The path <1,2,4,1> forms the same cycle as the paths <2,4,1,2> and <4,1,2,4>

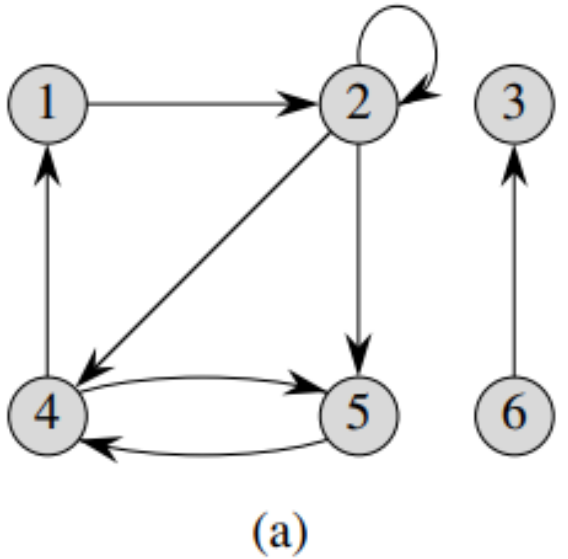
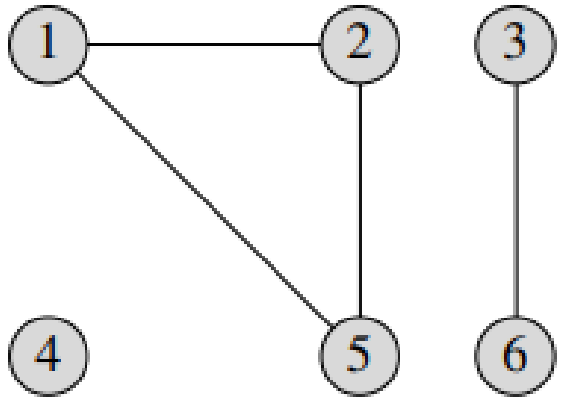


Fig B.2 Directed and Undirected graph [1]

Graph's Cycle

- A directed graph with no self-loops is **simple**.
- A graph with no cycles is **acyclic graph**.

- In an undirected graph, a path form a **(simple) cycle** if $k \geq 3$, $v_0 = v_k$, and $v_1, v_2, \ldots, v_k$ are distinct.
  - The path <1, 2, 5, 1> is a cycle.



(b)

Fig B.2 Directed and Undirected graph [1]

Connected Graph

- An undirected graph is **connected**
  - if every pair of vertices is connected by a path.
  - If it has exactly one connected component.
  - If every vertex is reachable form every other vertex.
- The **connected components** of a graph are the equivalence classes of vertices under the “is reachable from” relation.
  - Every vertex in {1,2,5} is reachable from ever other vertex in {1,2,5}.
- There are three connected components: {1, 2, 5}, {3,6} and {4}

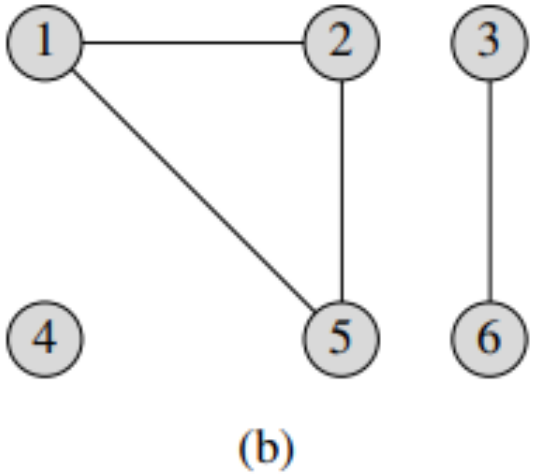


Fig B.2 Directed and Undirected graph [1]

Strongly Connected Graph

- A directed graph is **strongly connected**
  - if two vertices are reachable from each other,
  - if it has only one strongly connected component.
- The **strongly connected components** of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation.
  - All pair of vertices in {1,2, 4, 5} are mutually reachable.
  - The vertices {3, 6} do not form a strongly connected component, since 6 cannot be reached from vertex 3.
- There are three strongly connected components: {1, 2, 4, 5}, {3} and {6}.

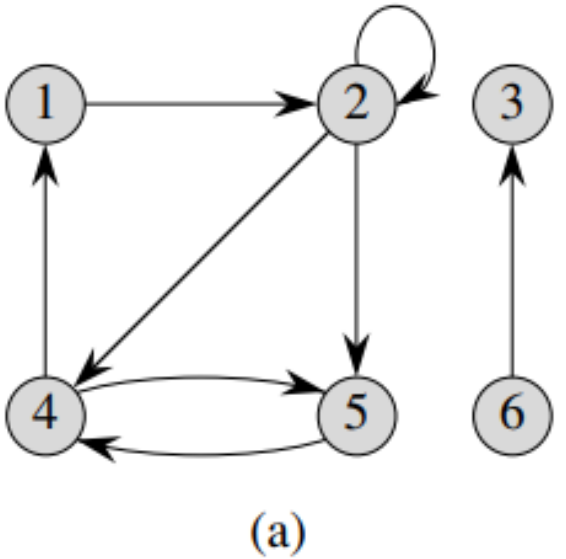
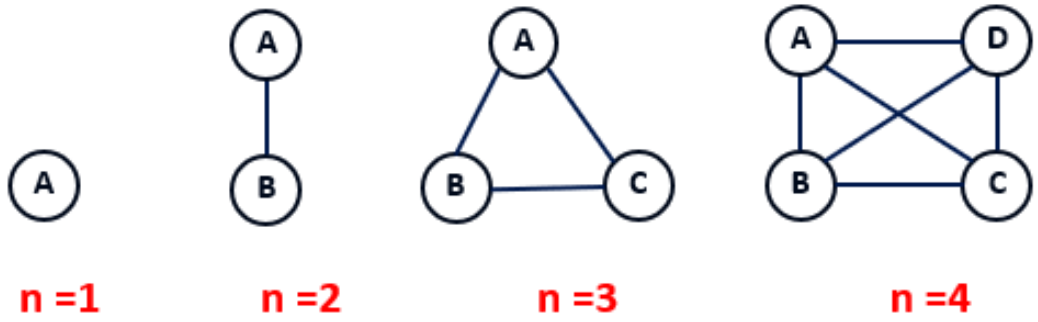


Fig B.2 Directed and Undirected graph [1]

Completed Undirected Graph

Let G=(V,E) be an undirected graph: | V | = n; | E | = e; d_i = degree of vertex i. **[2]**

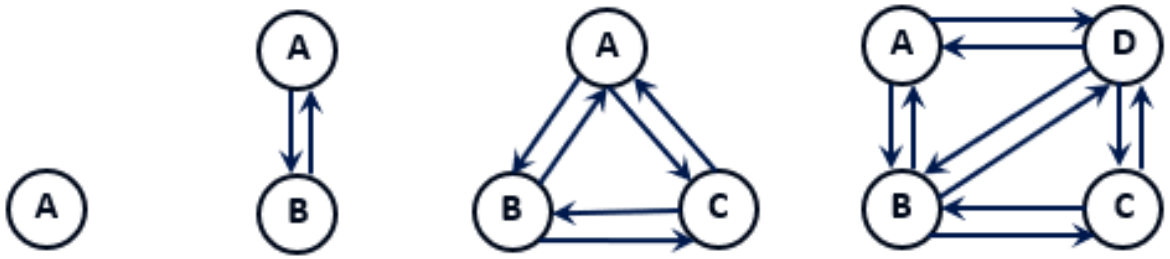
1. $\sum_{i=1}^{n} d_i$ = 2e
   - Each edge in an undirected graph is incident on exactly two vertices
   - The sum of the degrees of the vertices equals two times the number of edges.
2. $0 \leq e \leq n(n-1)/2$
   - The degree of a vertex lies between 0 and n-1.
   - The sum of the degrees lies between 0 and n(n-1)/2



Completed Directed Graph

Let G=(V,E) be a directed graph: | V | = n; | E | = e; d_i = degree of vertex i.

1. $\sum_{i=1}^{n} d_i$ (in) = $\sum_{i=1}^{n} d_i$ (out) = e
   - Each edge in a directed graph is incident on exactly two vertices
   - The sum of the degrees of the vertices equals two times the number of edges.
2. $0 \leq e \leq n(n-1)$
   - The degree of a vertex lies between 0 and n-1.
   - The sum of the degrees lies between 0 and n(n-1)



Two common Graph Representaions

Graph representations

- Either way is applicable to both directed and undirected graph. [1]

1. Adjacency-list
   - It provides a compact way to represent sparse graphs – those for which |E| is much less than $|V|^2$.

2. Adjacency-matrix
   - The graph is dense -|E| is close to $|V|^2$
   - We need to be able to tell quickly if there is an edge connecting two given vertices in shortest path algorithm.

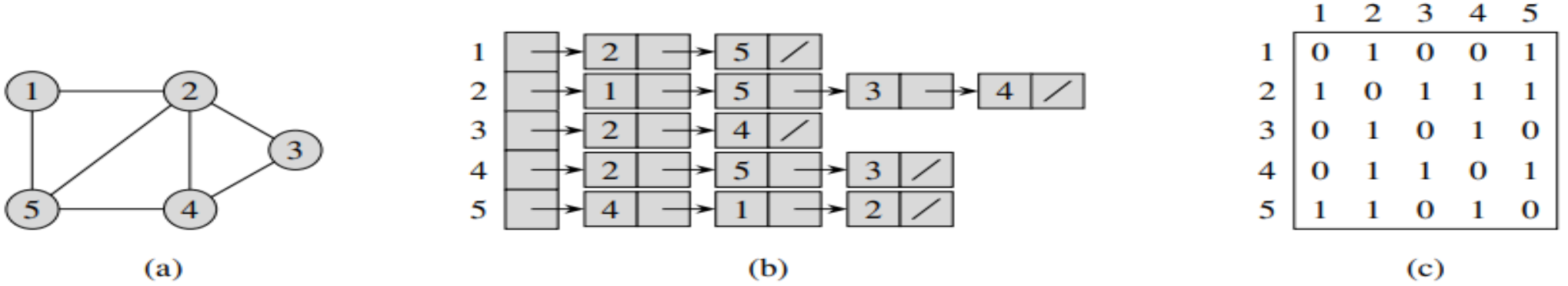


Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G. [1]

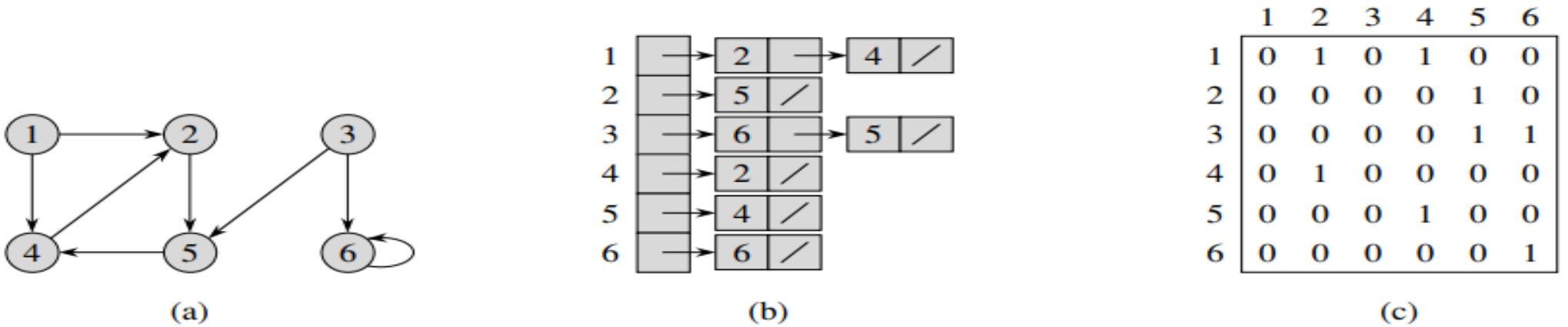


Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G. [1]

Adjacency-list: Undirected Graph

Graph representations

- It consists of an array of Adjacency of |V| list. [1]
- Given: a graph G = (V,E) and for each u ∈ V, the adjacency list – Adj[u] :
  - It contains all the vertices v such that there is an edge (u, v) ∈ E.
  - It consists of all vertices adjacent to u in G.
- The vertices in each adjacency list are typically stored in arbitrary order.

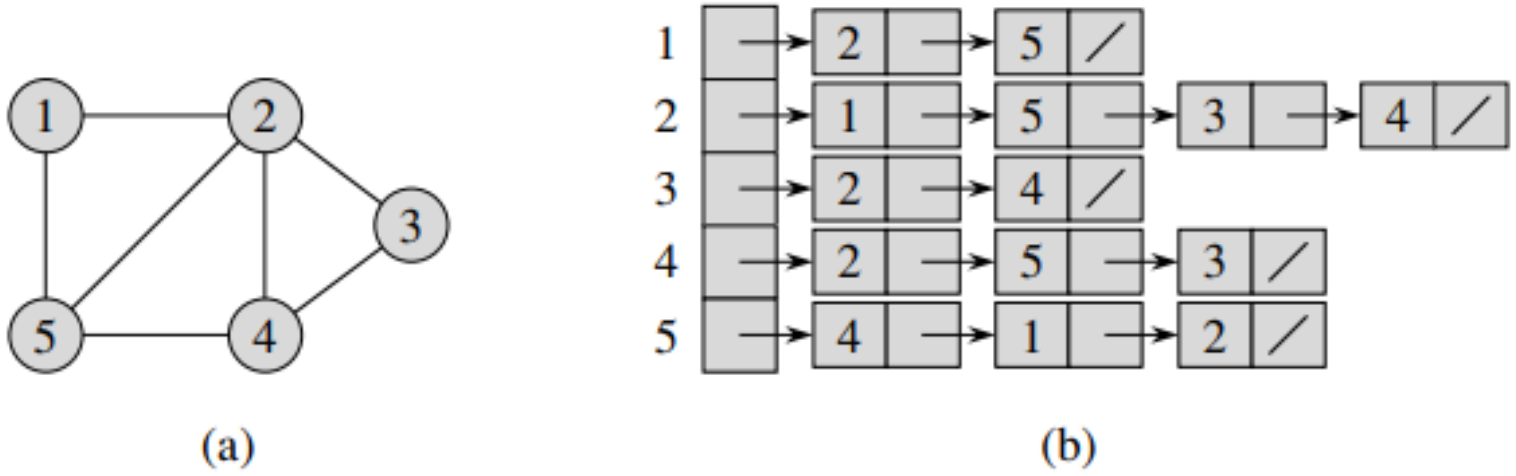


Fig 22.1 (b) An adjacency-list representation of undirected graph (a) [1]

Adjacency-list: Undirected Graph

Graph representations

- It consists of an array of Adjacency of |V| list. [1]
- Given: a graph G = (V,E) and for each u ∈ V, the adjacency list – Adj[u] :
  - It contains all the vertices v such that there is an edge (u, v) ∈ E.
  - It consists of all vertices adjacent to u in G.
- The vertices in each adjacency list are typically stored in arbitrary order.

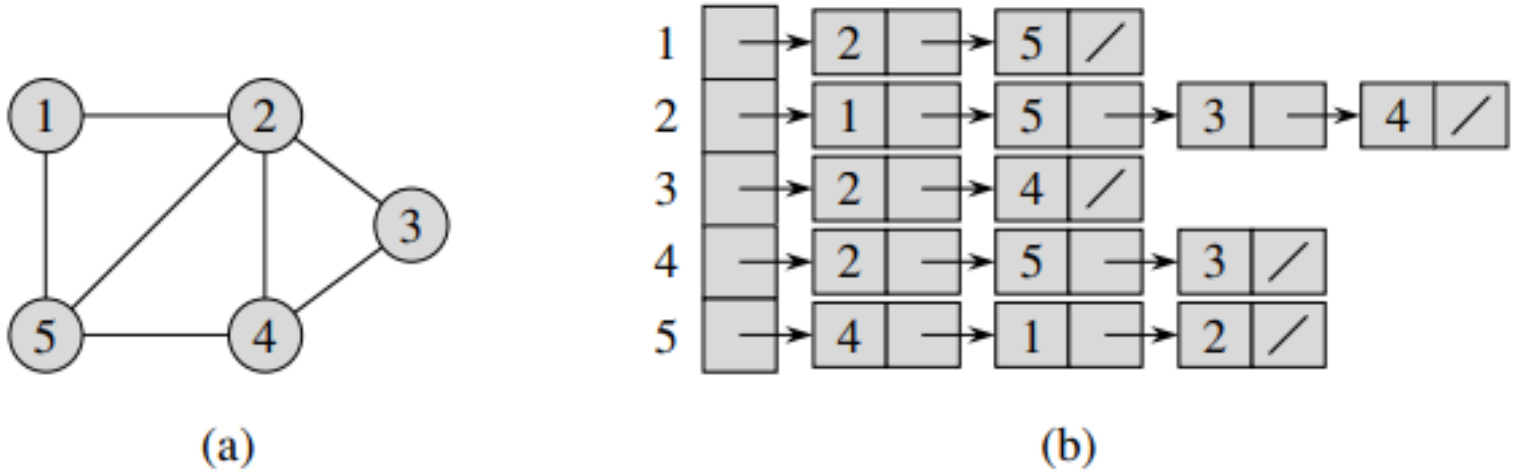


Fig 22.1 (b) An adjacency-list representation of undirected graph (a) [1]

Adjacency-list: Undirected Graph

Graph representations

- In the undirected graph, the sum of the lengths of all the adjacency list is 2|E|, since an edge of the form (u, v) is represented by having v appear in Adj[u]. [1]
  - A directed graph G = (V,E) consists of:
  - 5 vertices (|V|=5) -> |V| = {1, 2, 3, 4, 5}
  - 7 edges (|E|=14 -> 2x7 but Illustrated only 7 edges) -> |E| = {(1,2), (1,5), (2,1),(2,3), (2,4),(2,5),(3,2),(3,4),(4,2),(4,3),(4,5),(5,1),(5,2),(5,4)}

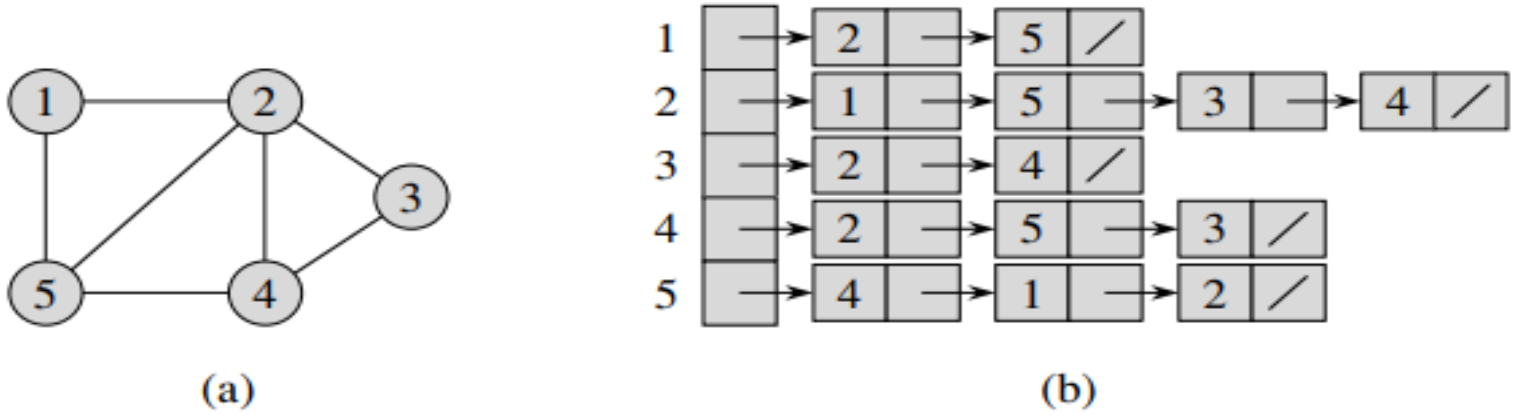
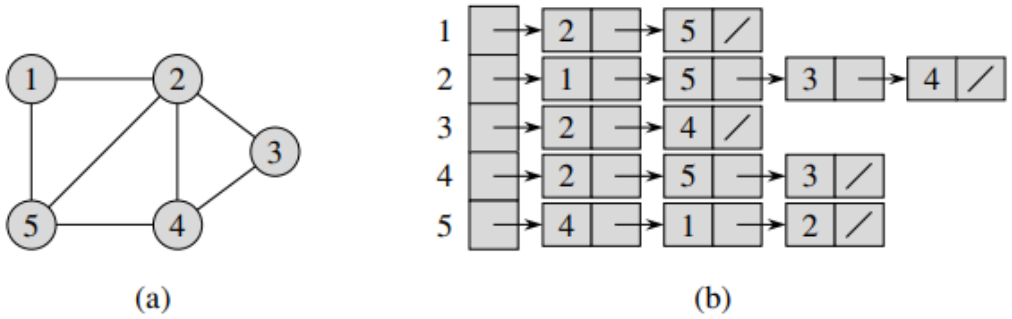


Fig 22.1 (b) An adjacency-list representation of undirected graph (a) [1]

Adjacency-list: Directed Graph

Graph representations



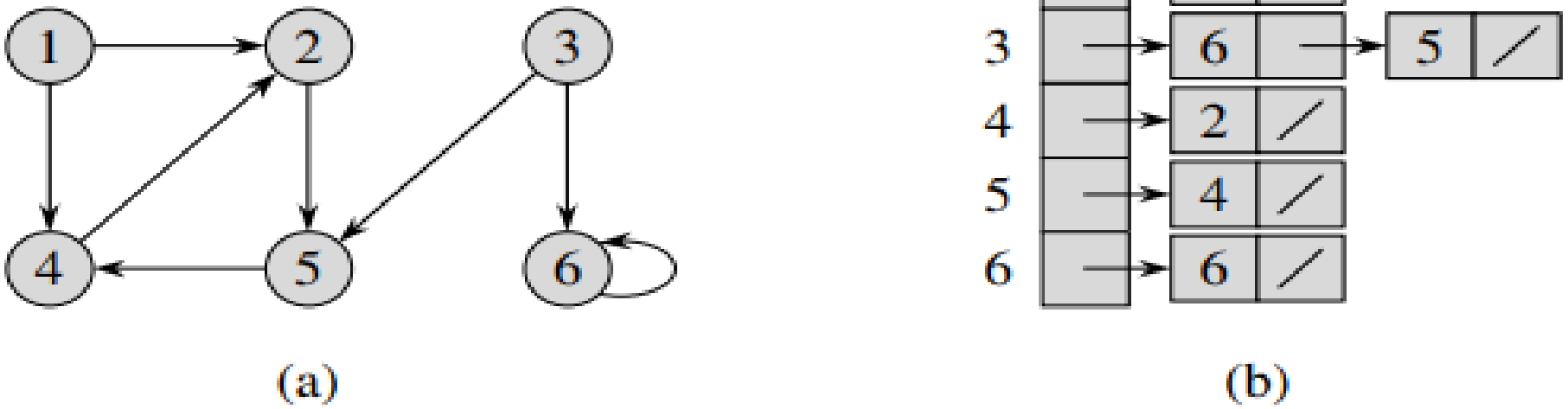


Fig 22.2 (b) An adjacency-list representation of directed graph (a) [1]

Adjacency-list: Directed Graph

Graph representations

- In the directed graph, the sum of the lengths of all the adjacency list is |E|, since an edge of the form (u, v) is represented by having v appear in Adj[u]. [1]
  - A directed graph G = (V,E) consists of:
  - 5 vertices (|V|=5) -> |V| = {1, 2, 3, 4, 5}
  - 8 edges (|E|= 8) -> |E| = {(1,2), (1,4),(2,5),(3,5),(3,6),(4,2),(5,4),(6,6)}

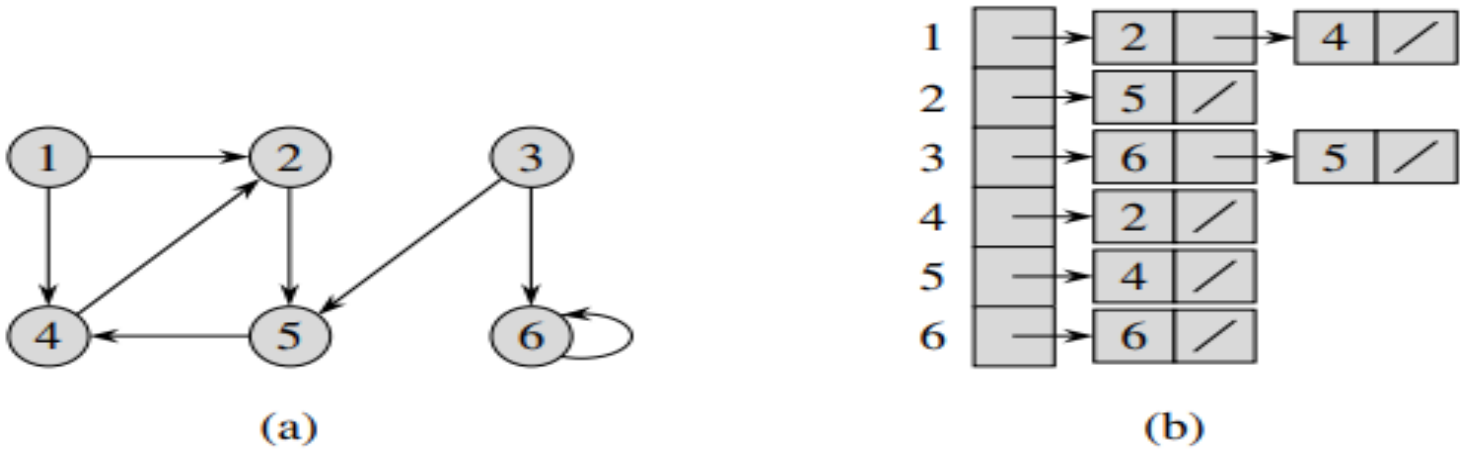
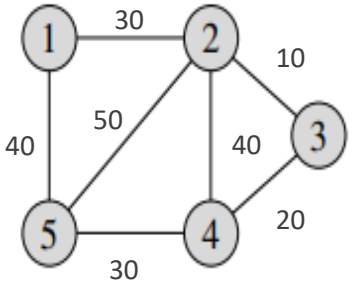


Fig 22.2 (b) An adjacency-list representation of directed graph (a) [1]

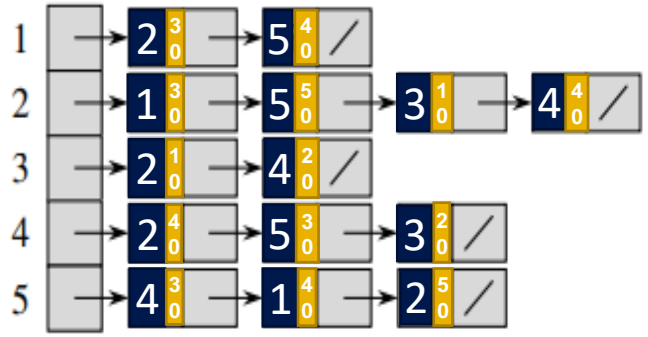
Adjacency-list: Weighted Graph

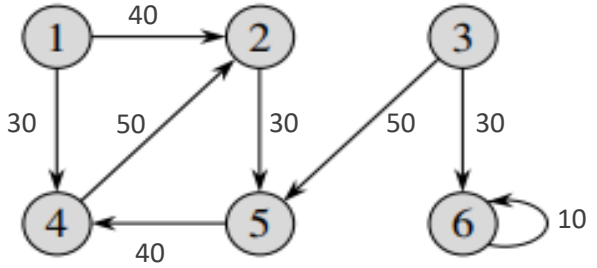
Graph representations

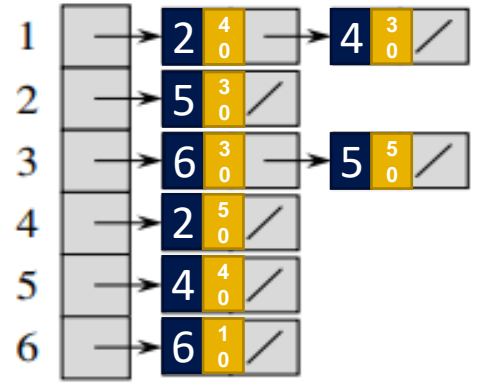
- Weighted graph is a graph which each edge has an associated weight. [1]
  - A weighted graph G = (V,E) with a weight function w : E $\rightarrow$ R
  - The weight w(u, v) of the edge (u, v) $\in$ E is stored with vertex v in u's adjacency list



(a)







(b)

Adjacency-list: Advantage

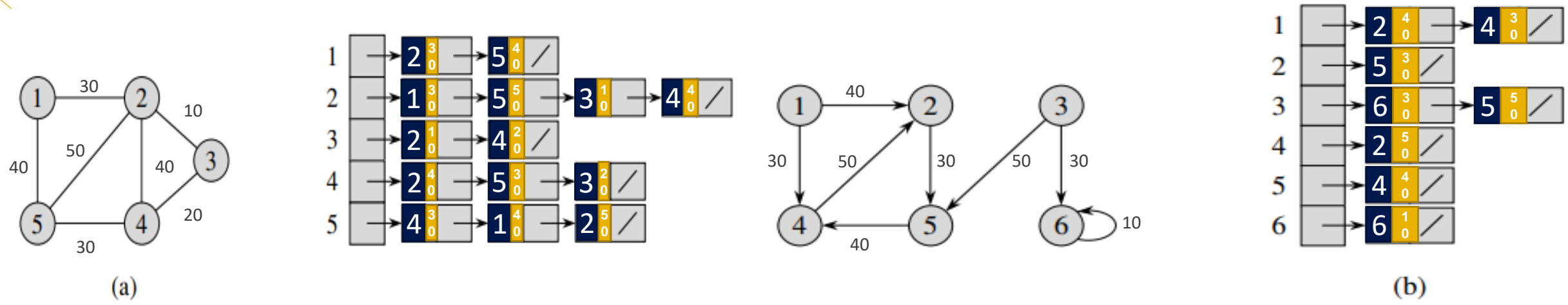
Graph representations

- Although the adjacency-list representation is asymptotically at least as efficient as the adjacency-matrix representation, [1]
  - The simplicity of an adjacency-matrix may be recommended when the graph is small. This can be remedied by an adjacency-matrix representation of graph.
  - When the graph is unweighted graph, the storage needed by the adjacency-matrix is optimized due to the 1-bit representation.

Adjacency-list: Disadvantage

Graph representations

- There is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list Adj[u]. [1]
- This can be remedied by an adjacency-matrix representation of graph.
  - At cost of using asymptotically more memory.



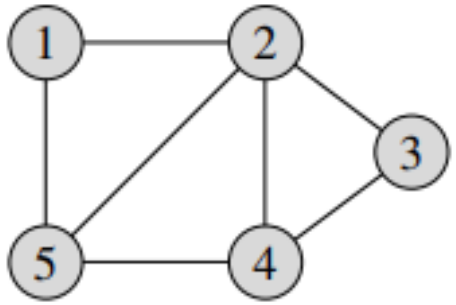
Adjacency-matrix: Undirected Graph

Graph representations

- Since in an undirected graph, (u, v) and (v, u) represents the same edge, the adjacent matrix A of an undirected graph is its own transpose: [1]

$$A = a_{i,j} = A^T = a_{j,i}$$

- In some application, it pays to store only the entries on and above the diagonal of the adjacency matrix, thereby cutting the memory needed to store the graph almost in half. [1]



(a)

	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)

Fig 22.1 (c) An adjacency-matrix representation of undirected graph (a) [1]

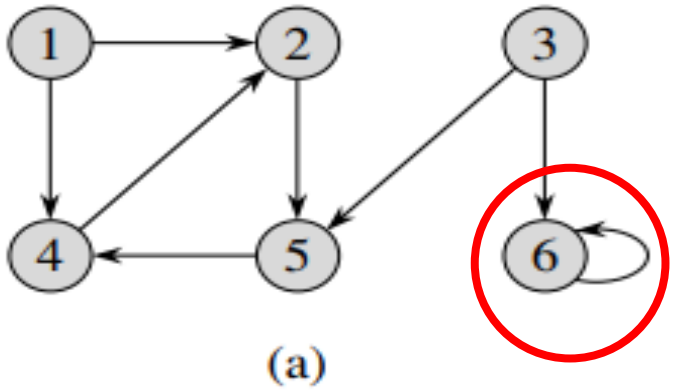
Adjacency-matrix: Directed Graph

Graph representations

- It consists of a |V| x |V| matrix A = ($a_{I,J}$) when:

$$a_{i,j} = \begin{cases} 1 \ \ if \ (i,j) \in E \ and \ vetex \ i \ is \ adjacent \ toward \ to \ j \\ 0 \ \ Otherwise \end{cases}$$

- Unlike the undirected graph, The matrix is not symmetric matrix.



(a)

	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

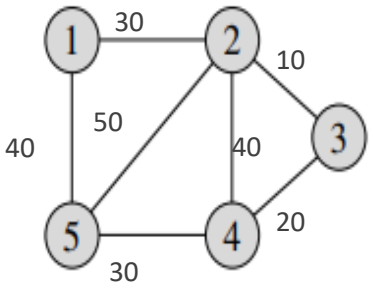
(c)

Fig 22.2 (c) An adjacency-list representation of directed graph (a) [1]

Adjacency-matrix: Weighted Graph

Graph representations

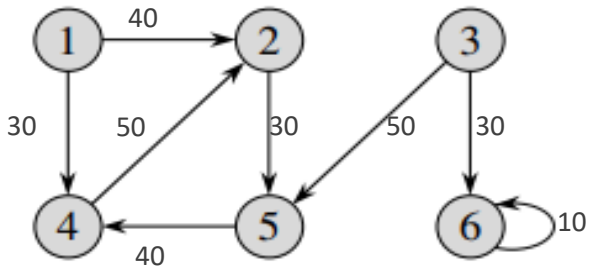
- Like the adjacency-list, the adjacency-matrix can be used for representing the weighted graph. [1]
- If an edge does not exist, a NIL value – 0 or α, can be represented.



(a)

	1	2	3	4	5
1	0	30	0	0	40
2	30	0	10	40	50
3	0	10	0	20	0
4	0	40	20	0	30
5	40	50	0	30	0

(c)



	1	2	3	4	5	6
1	0	40	0	30	0	0
2	0	0	0	0	30	0
3	0	0	0	0	50	30
4	0	50	0	0	0	0
5	0	0	0	40	0	0
6	0	0	0	0	0	10

(c)

22.2

Breadth-first Search

Breadth-first Search

- This algorithm works in both directed and undirected graph.
- It is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms – such as Prim's minimum-spanning-tree algorithm, Dijkstra's single-source shortest-paths algorithm. [1]
- This search aims systematically explores the edges of a graph to every vertex that is reachable from a distinguished source vertex. [1]
  - It also produces a "breadth-first tree" with root that contains all reachable vertices.

Breadth-first Search

Concept of work

- It colors each vertex white, grey or black.
  - All vertices start out white.
  - All vertices adjacent to black vertices have been discovered.
  - Gray vertices may have some adjacent white discovered vertices (undiscovered) or black discovered vertices.

- It start at the source vertex and viewed as its root. [1]
  - Whenever a white vertex u – as the ancestor or parent, is discovered, the vertex v – as its descendant or child, and edges (u,v) will be added to the tree.

Breadth-first Search

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

BFS(G is represented as adjacency list, **s** is a source vertex**)**

r		s	v	
s		w	r	
t		w	x	u
u		t	x	y
v		r		
w		s	t	x
x		w	t	u	y
y		x	u	

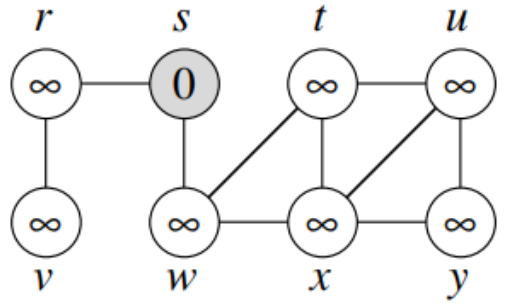


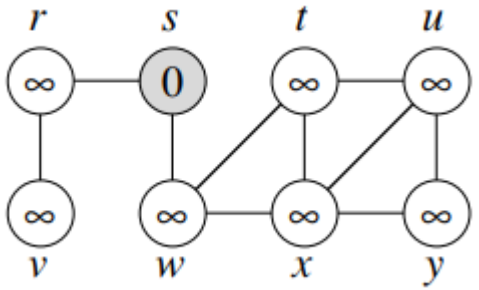
Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

Breadth-first Search

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

(a)



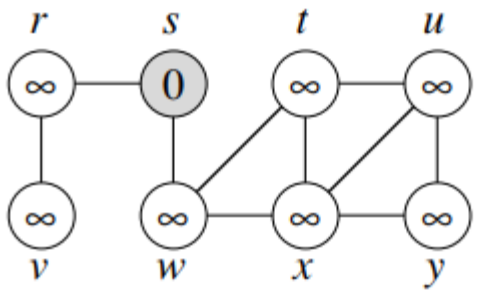
V[G]	d[u]	π[u]
r	α	
s	0	NIL
t	α	
u	α	
v	α	
w	α	
x	α	
y	α	
Enqueue		
Q	ϕ	
Dequeue		

Breadth-first Search


```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

(a)



Q | s | 0

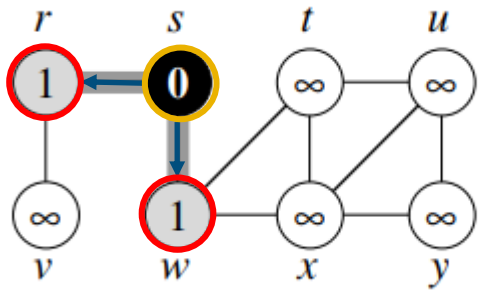
V[G]	d[u]	π[u]
r	α	
s	0	NIL
t	α	
u	α	
v	α	
w	α	
x	α	
y	α	
Enqueue	s	
Q	s	
Dequeue		

Breadth-first Search

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

(b)



Q | w | r |
1 | 1

V[G]	d[u]	π[u]
r	α	
s	0	NIL
t	α	
u	α	
v	α	
w	α	
x	α	
y	α	
Enqueue	s	
Q	s	
Dequeue	s	
Color [s] = BLACK		

r		s	v	
s		w	r	
t		w	x	u
u		t	x	y
v		r		
w		s	t	x
x		w	t	u	y
y		x	u	

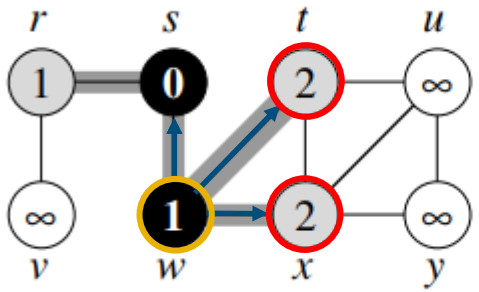
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	α	
u	α	
v	α	
w	1	s
x	α	
y	α	
Enqueue	w, r	
Q	w, r	
Dequeue	w	
Color[w] = BLACK		

Breadth-first Search

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

(c)



Q

r	t	x
1	2	2

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	α	
u	α	
v	α	
w	1	s
x	α	
y	α	
Enqueue	w, r	
Q	w, r	
Dequeue	w	
Color[w] = BLACK		

r		s	v	
s		r	w	
t		w	x	u
u		t	x	y
v		r		
w		s	t	x
x		w	t	u
y		x	u	

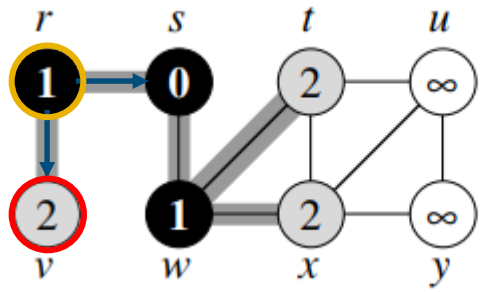
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	*2*	*w*
u	α	
v	α	
w	1	s
x	*2*	*w*
y	α	
Enqueue	t, x	
Q	r, t, x	
Dequeue	r	
Color[r] = BLACK		

Breadth-first Search

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

(d)



Q | t | x | v
2 2 2

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	*2*	*w*
u	α	
v	α	
w	1	s
x	*2*	*w*
y	α	
Enqueue	t, x	
Q	r, t, x	
Dequeue	r	
Color[r] = BLACK		

r		*s*	*v*
s		*r*	*w*
t		*w*	*x*	*u*
u		*t*	*x*	*y*
v		*r*
w		*s*	*t*	*x*
x		*w*	*t*	*u*	*y*
y		*x*	*u*

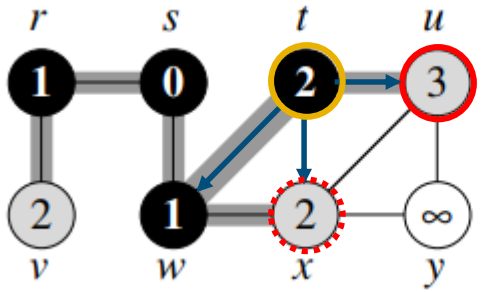
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	α	
v	*2*	*r*
w	1	s
x	2	w
y	α	
Enqueue	v	
Q	t, x, v	
Dequeue	t	
Color[t] = BLACK		

Breadth-first Search


```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

(e)



Q | x | v | u
2 2 3

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	α	
v	***2***	***r***
w	1	s
x	2	w
y	α	
Enqueue	v	
Q	t, x, v	
Dequeue	t	
Color[t] = BLACK		

r		s	v	
s		r	w	
t		w	x	u
u		t	x	y
v		r		
w		s	t	x
x		w	t	u
y		x	u	

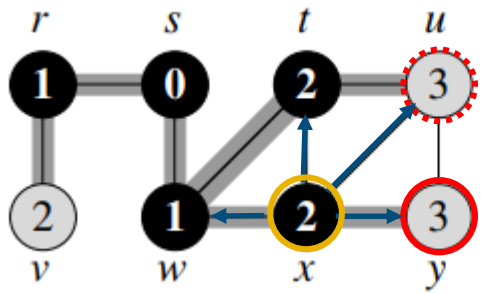
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	***3***	***t***
v	2	r
w	1	s
x	2	w
y	α	
Enqueue	u	
Q	x, v. u	
Dequeue	x	
Color[x] = BLACK		

Breadth-first Search


```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

(f)



Q | v | u | y |
2 3 3

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	*3*	*t*
v	2	r
w	1	s
x	2	w
y	α	
Enqueue	u	
Q	x, v. u	
Dequeue	x	
Color[x] = BLACK		

r		*s*	*v*	
s		*r*	*w*	
t		*w*	*x*	*u*
u		*t*	*x*	*y*
v		*r*		
w		*s*	*t*	*x*
x		*w*	*t*	*u*	*y*
y		*x*	*u*	

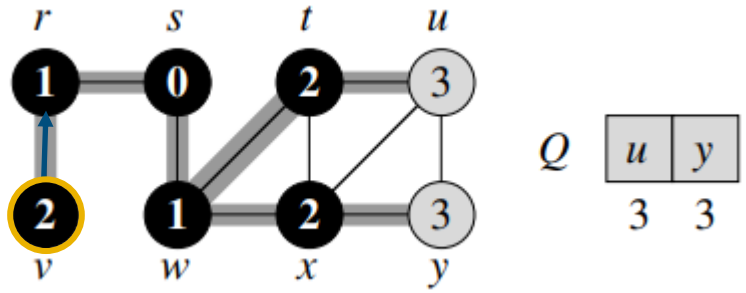
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	*3*	*x*
Enqueue	y	
Q	v. u, y	
Dequeue	v	
Color[v] = BLACK		

Breadth-first Search

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

(g)



V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	***3***	*x*
Enqueue	y	
Q	v. u, y	
Dequeue	v	
Color[v] = BLACK		

r		*s*	*v*	
s		*r*	*w*	
t		*w*	*x*	*u*
u		*t*	*x*	*y*
v		*r*		
w		*s*	*t*	*x*
x		*w*	*t*	*u*	*y*
y		*x*	*u*	

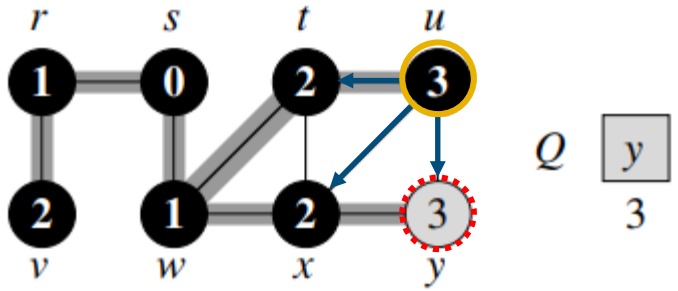
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	u, y	
Dequeue	u	
Color[u] = BLACK		

Breadth-first Search

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

(h)



Q | y

3

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	u, y	
Dequeue	u	
Color[u] = BLACK		

r		s	v
s		r	w
t		w	x	u
u		t	x	y
v		r	
w		s	t	x
x		w	t	u	y
y		x	u	

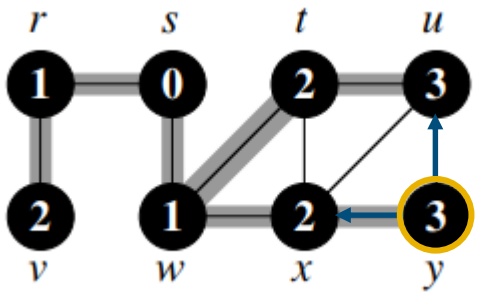
V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	y	
Dequeue	y	
Color[y] = BLACK		

Breadth-first Search


```
BFS(G, s)
 1  for each vertex u ∈ V[G] − {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       do u ← DEQUEUE(Q)
12          for each v ∈ Adj[u]
13               do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

(i)



$Q \quad \emptyset$

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	y	
Dequeue	y	
Color[y] = BLACK		

V[G]	d[u]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	ϕ	
Dequeue	None	
-		

Breadth-first Search

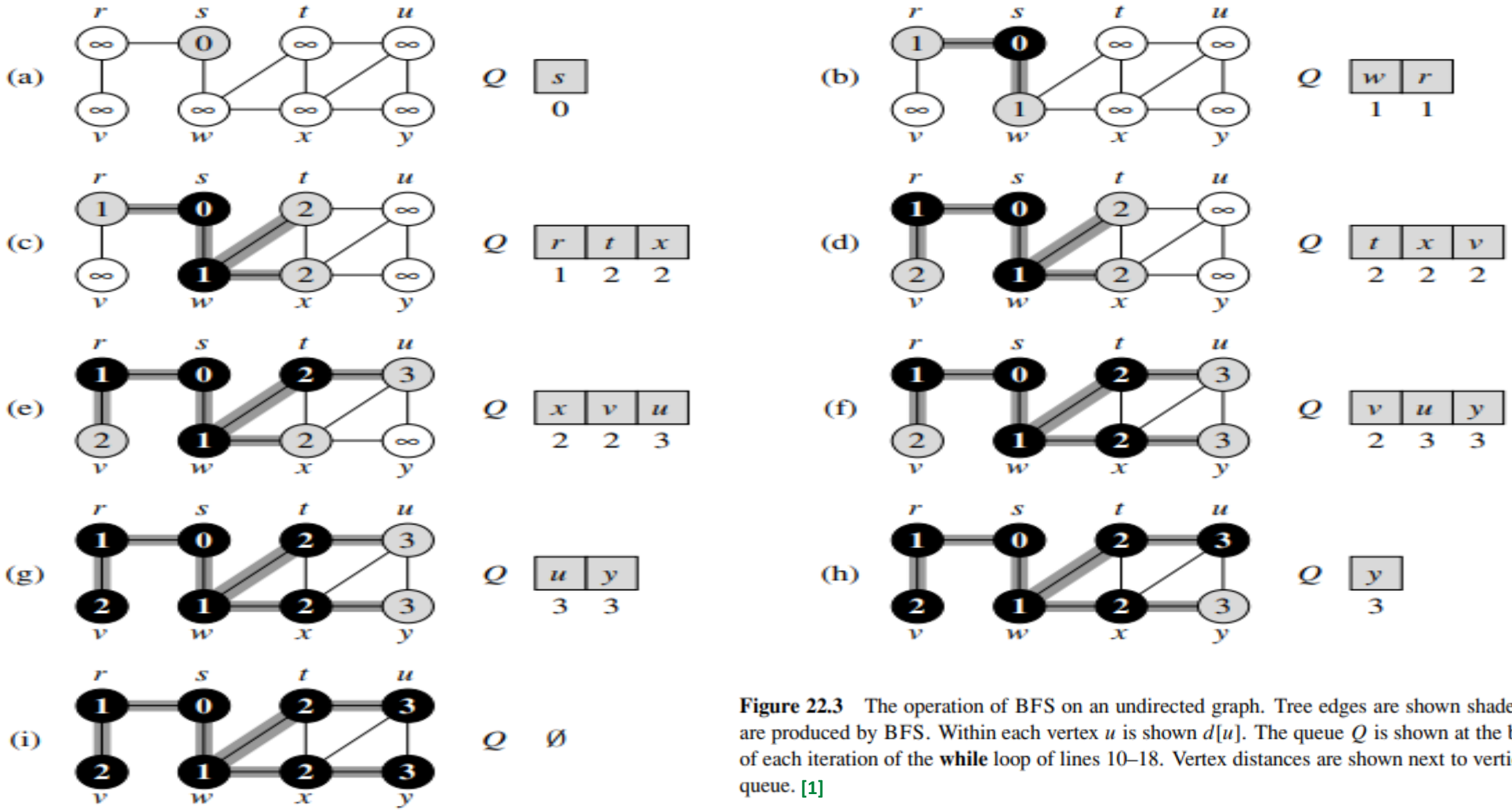
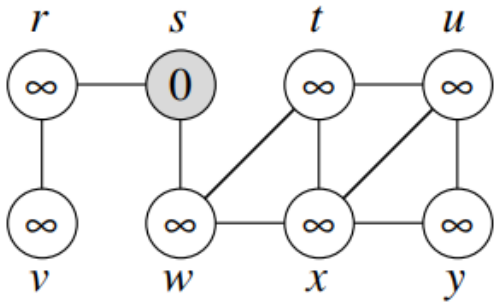


Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]

Breadth-first Search

Analysis

- The total running time of BFS is O(V+E)
  - The adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once.
  - The sum of lengths of all adjacency list is $\Theta(E)$, the total time spent in scanning adjacency lists is O(E).
  - BFS runs in linear in the size of the adjacency list representation of G.



r		s	v	
s		w	r	
t		w	x	u
u		t	x	y
v		r		
w		s	t	x
x		w	t	u	y
y		x	u	

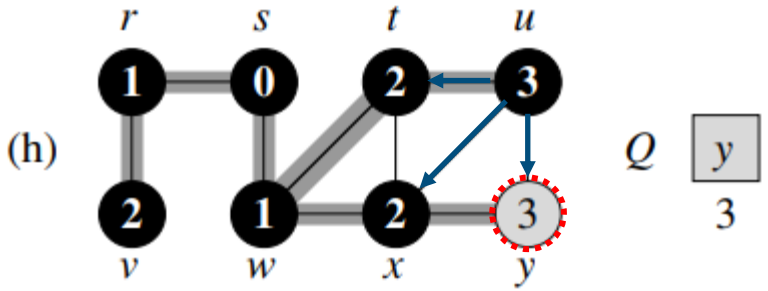
Breadth-first Search

22.2

Shortest paths

- BFS finds the distance to each reachable vertex in a graph - d[v], as the shortest-path distance - δ (s, v).

Figure 22.3 The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. Within each vertex u is shown $d[u]$. The queue Q is shown at the beginning of each iteration of the **while** loop of lines 10–18. Vertex distances are shown next to vertices in the queue. [1]



V[G]	d[v]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	u, y	
Dequeue	u	
Color[u] = BLACK		

r		s	v	
s		r	w	
t		w	x	u
u		t	x	y
v		r		
w		s	t	x
x		w	t	u	y
y		x	u	

V[G]	d[v]	π[u]
r	1	s
s	0	NIL
t	2	w
u	3	t
v	2	r
w	1	s
x	2	w
y	3	x
Enqueue	None	
Q	y	
Dequeue	y	
Color[y] = BLACK		

22.3

Depth-first Search

Depth-first Search

- To search “deeper” in the graph, entire edges along its path are explored in sequence.

- Like BFS, it records entire edges in a path by setting a destination vertex (or v’s predecessor field, π[v]) to u.

- Unlike BFS, whose predecessor subgraph forms a tree, the predecessor subgraph may be composed of several trees, because the search may be repeated form multiple sources. **[1]**

Depth-first Search

Predecessor subgraph

- Unlike BFS, the predecessor subgraph forms a dept-first forest composed of several depth-first trees. **[1]**

  G = (V, E_π) contains a **tree edge set** -> $\boldsymbol{E_\pi} = \{(\pi[v], v) : v \epsilon V \ and \ \pi[v] \neq NIL\}$

- DFS timestamps each vertex as follows:

1. d[v] records when v is first discovered (v is grayed)
2. f[v] records when the search finishes examining v's adjacency list (v is blacken).

Depth-first Search

Classification of edges

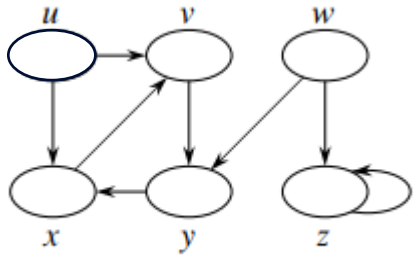
- The search classifies the edges of graph as follows. [1]
  1. **Tree edge**, if v - **WHITE** vertex, was the first discovered.
  2. **Back edge** is those edges (u, v) connecting a vertex u to ancestor v – **GRAY** vertex, in a depth-first tree and a self-loop or edges (u, u) in a directed graphs.
  3. **Forward edge** is nontree edge (u,v) connecting a vertex u to a descendant v in a depth-fist tree.
  4. **Cross edges** are all other edges which they can go between vertices – **BLACK** vertices, in the same or different depth-first trees.

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



V[G]	d[u] / f[u]	π[v]
u		NIL
v		NIL
w		NIL
x		NIL
y		NIL
z		NIL

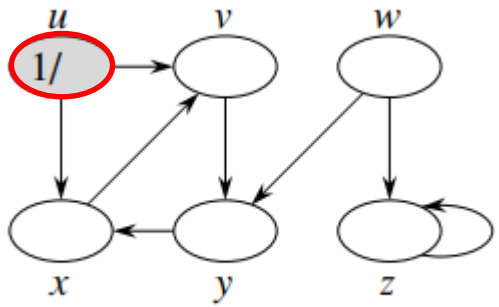
u		*v*	*x*
v		*y*	
w		*y*	*z*
x		*v*	
y		*x*	
z		*z*	

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(a)

V[G]	d[u] / f[u]	π[v]
u		NIL
v		NIL
w		NIL
x		NIL
y		NIL
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

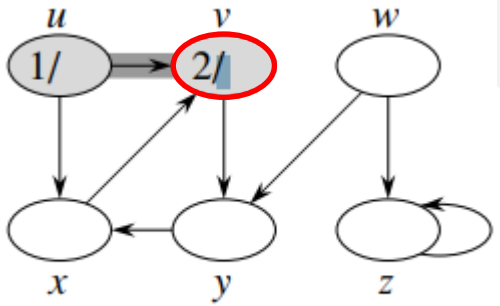
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v		NIL
w		NIL
x		NIL
y		NIL
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(b)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v		NIL
w		NIL
x		NIL
y		NIL
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

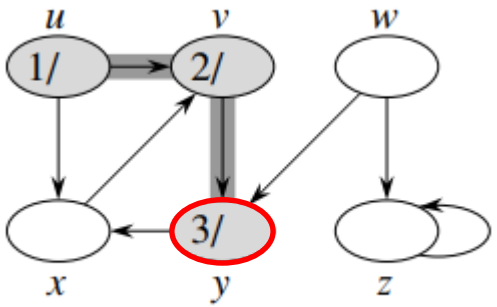
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u
w		NIL
x		NIL
y		NIL
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(c)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u
w		NIL
x		NIL
y		NIL
z		NIL

u	v	x
v	y	
w	y	z
x	v	
y	x	
z	z	

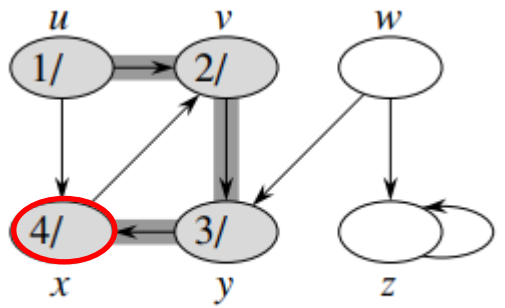
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u
w		NIL
x		NIL
y	3/	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(d)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u
w		NIL
x		NIL
y	3/	v
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

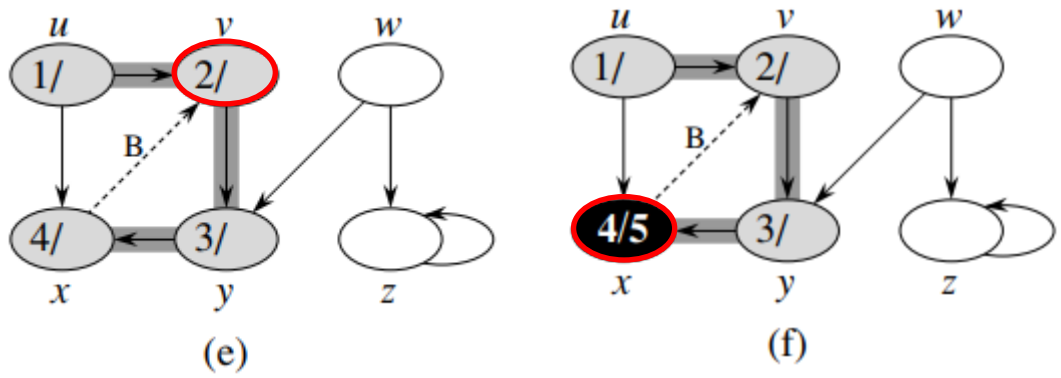
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u
w		NIL
x	4/	y
y	3/	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u
w		NIL
x	4/	y
y	3/	v
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

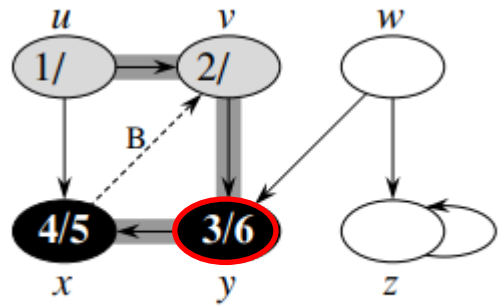
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u, B-x
w		NIL
x	4/5	y
y	3/	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(g)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u, B-x
w		NIL
x	4/5	y
y	3/	v
z		NIL

u	v	x
v	y	
w	y	z
x	v	
y	x	
z	z	

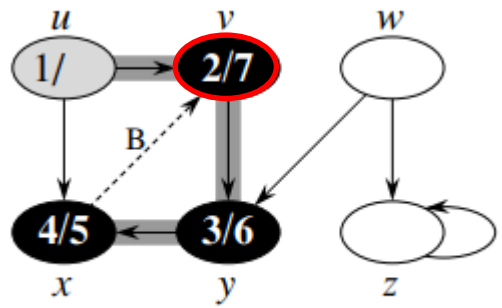
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u, B-x
w		NIL
x	4/5	y
y	3/6	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(h)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/	u, B-x
w		NIL
x	4/5	y
y	3/6	v
z		NIL

u	v	x
v	y	
w	y	z
x	v	
y	x	
z	z	

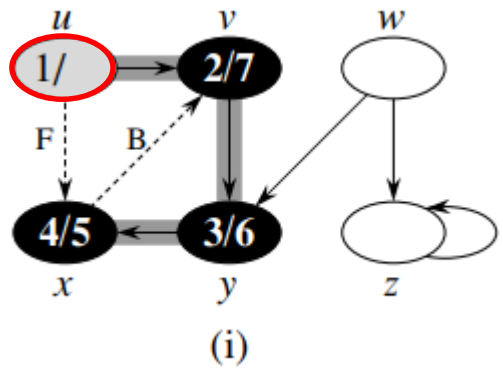
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/7	u, B-x
w		NIL
x	4/5	y
y	3/6	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(i)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/7	u, B-x
w		NIL
x	4/5	y
y	3/6	v
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

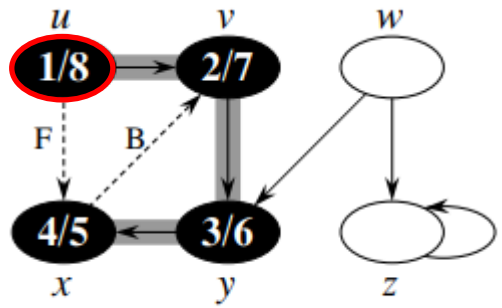
V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/7	u, B-x
w		NIL
x	4/5	y
y	3/6	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(j)

V[G]	d[u] / f[u]	π[v]
u	1/	NIL
v	2/7	u, B-x
w		NIL
x	4/5	y
y	3/6	v
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

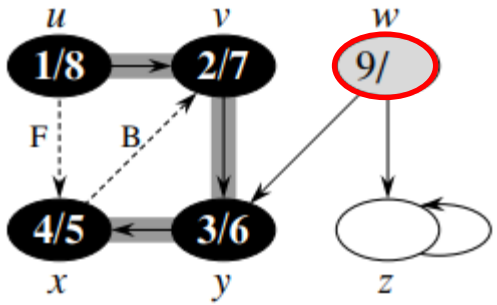
V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w		NIL
x	4/5	y, F-u
y	3/6	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(k)

V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w		NIL
x	4/5	y, F-u
y	3/6	v
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

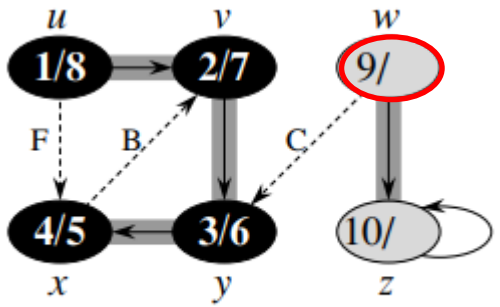
V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(m)

V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

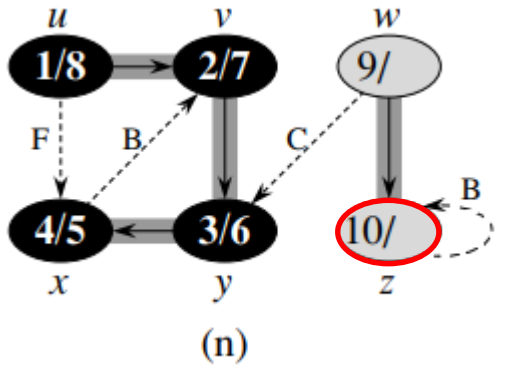
V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z		NIL

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2      do color[u] ← WHITE
3         π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6      do if color[u] = WHITE
7         then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5      do if color[v] = WHITE
6         then π[v] ← u
7              DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(n)

V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z		NIL

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

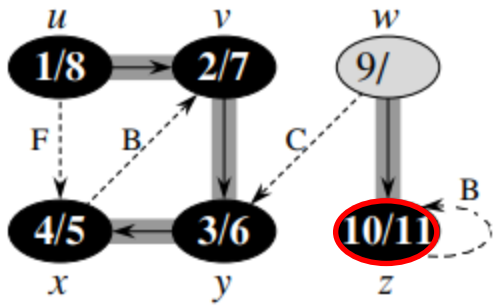
V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z	10/	w

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2      do color[u] ← WHITE
3         π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6      do if color[u] = WHITE
7         then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5      do if color[v] = WHITE
6         then π[v] ← u
7              DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



(o)

V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z	10/	w

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

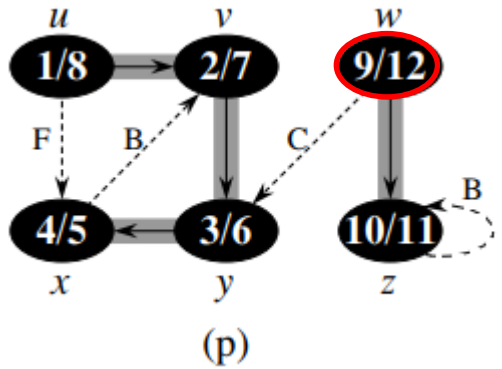
V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z	10/11	w, B-x

Depth-first Search

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
```

```
DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]

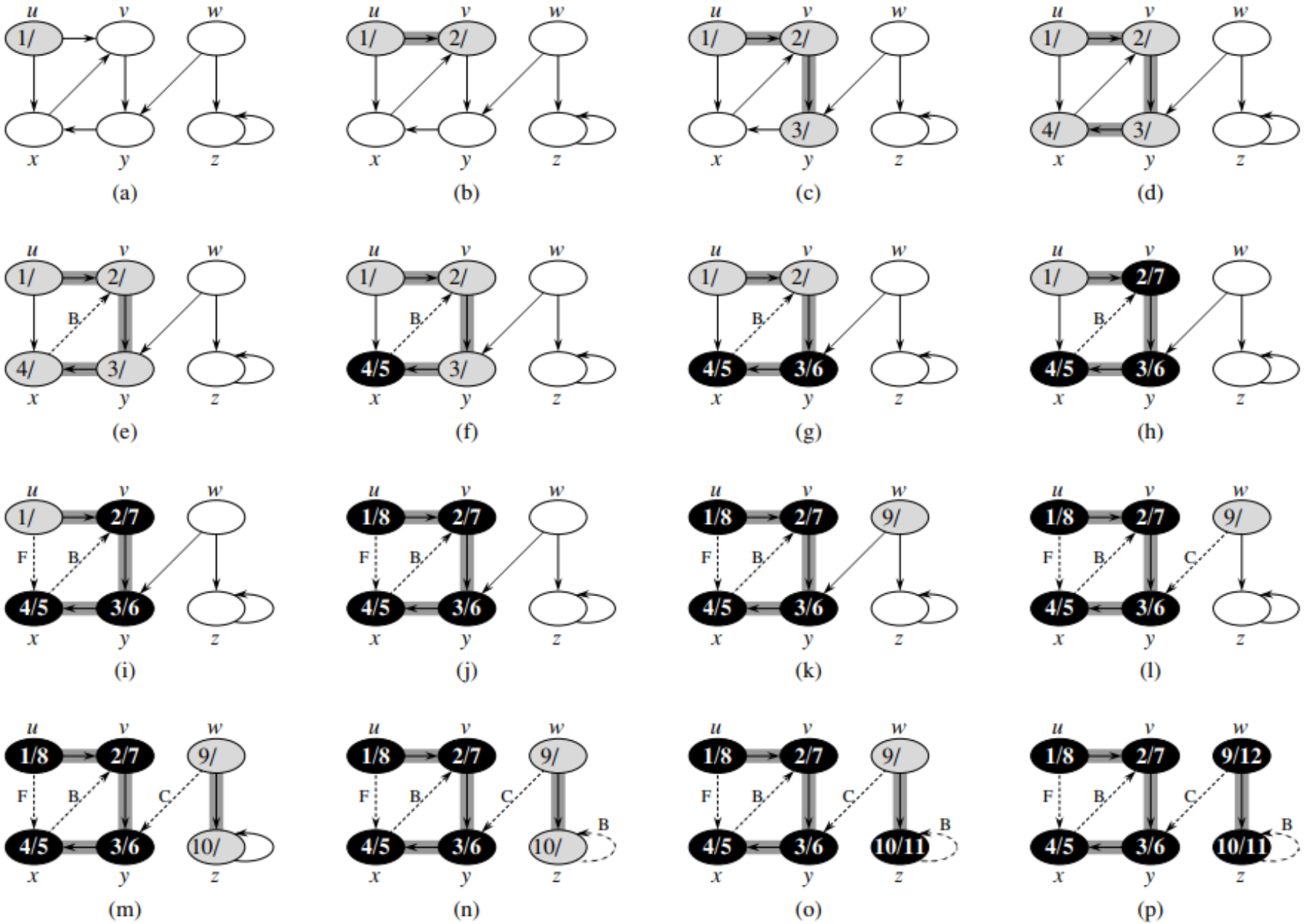


V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z	10/11	w, B-x

u		v	x
v		y	
w		y	z
x		v	
y		x	
z		z	

V[G]	d[u] / f[u]	π[v]
u	1/8	NIL
v	2/7	u, B-x
w	9/12	NIL
x	4/5	y, F-u
y	3/6	v, C-w
z	10/11	w, B-x

Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]



Depth-first Search

Running Time

```
DFS(G)
1  for each vertex u ∈ V[G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)

DFS-VISIT(u)
1  color[u] ← GRAY        ▷ White vertex u has just been discovered.
2  time ← time + 1
3  d[u] ← time
4  for each v ∈ Adj[u]    ▷ Explore edge (u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] ← BLACK       ▷ Blacken u; it is finished.
9  f[u] ← time ← time + 1
```

- The loop on line 1-3 takes time $\Theta(V)$.
- The total cost of executing line 4-7 of DFS-VISIT is $\Theta(E)$.
- The running time of DFS is $\Theta(V+E)$. [1]

Depth-first Search

Properties

- The most basic property of DFS is that the predecessor subgraph which does indeed from a forest of tees. [1]
  - The structure of depth-first trees exactly mirrors the structure of recursive calls of DFS-VISIT.

- Another important property of DFS is that discovery and finishing times have parenthesis structure.

Depth-first Search

Properties

Theorem 22.7 (Parenthesis theorem)

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:

- the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in a depth-first tree, or
- the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in a depth-first tree.

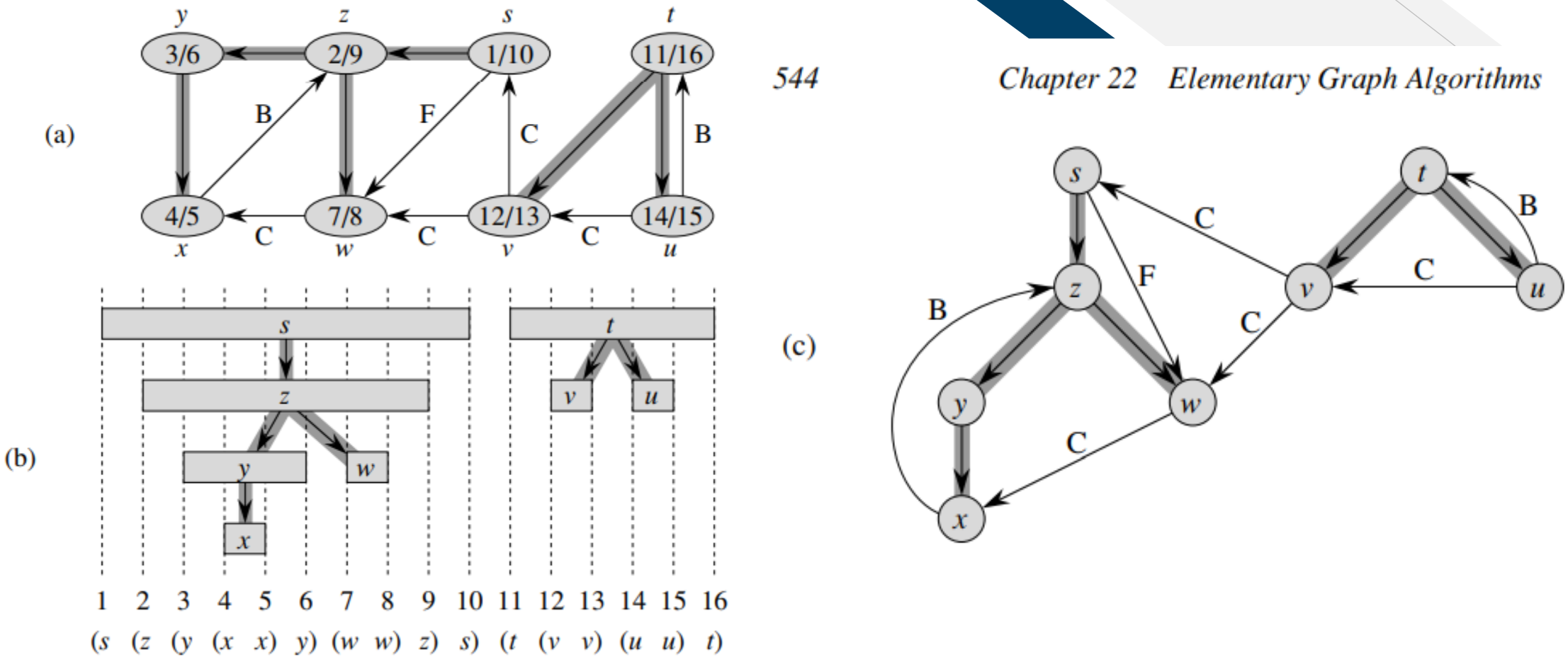


Figure 22.5 Properties of depth-first search. **(a)** The result of a depth-first search of a directed graph. Vertices are timestamped and edge types are indicated as in Figure 22.4. **(b)** Intervals for the discovery time and finishing time of each vertex correspond to the parenthesization shown. Each rectangle spans the interval given by the discovery and finishing times of the corresponding vertex. Tree edges are shown. If two intervals overlap, then one is nested within the other, and the vertex corresponding to the smaller interval is a descendant of the vertex corresponding to the larger. **(c)** The graph of part (a) redrawn with all tree and forward edges going down within a depth-first tree and all back edges going up from a descendant to an ancestor.

Depth-first Search

Classification of edge

We can define four edge types in terms of the depth-first forest G_π produced by a depth-first search on G. [1]

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

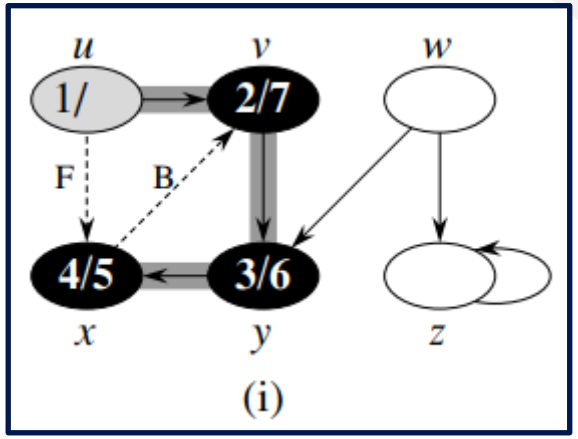
Depth-first Search

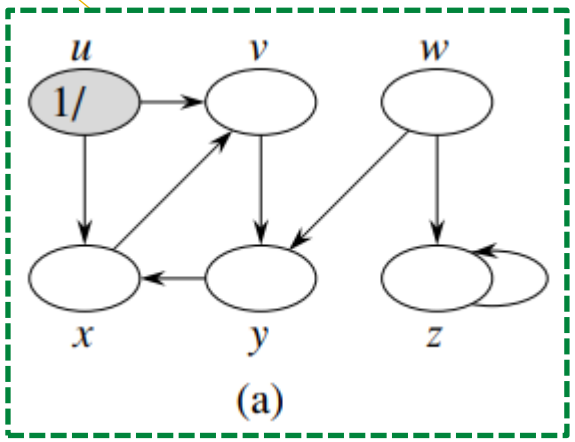
Classification of edge

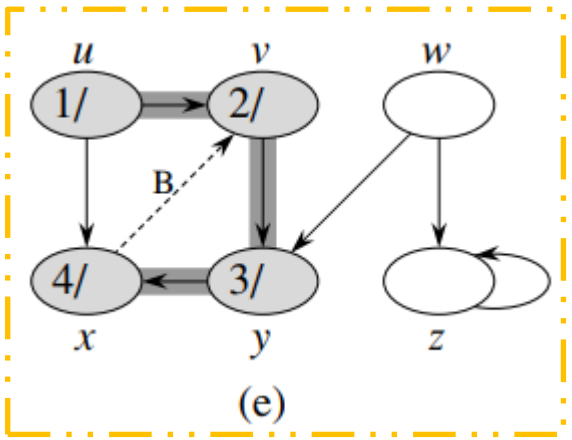
Figure 22.4 The progress of the depth-first-search algorithm DFS on a directed graph. As edges are explored by the algorithm, they are shown as either shaded (if they are tree edges) or dashed (otherwise). Nontree edges are labeled B, C, or F according to whether they are back, cross, or forward edges. Vertices are timestamped by discovery time/finishing time. [1]

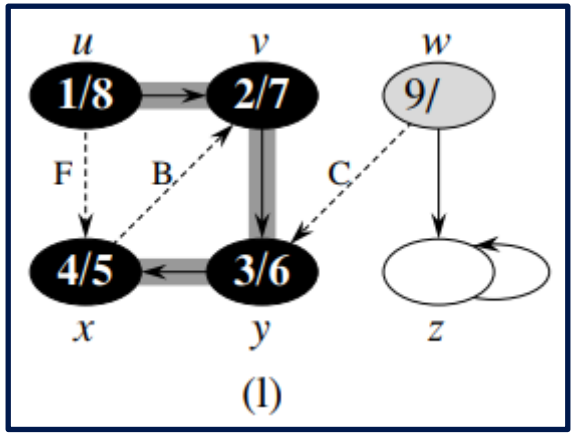
The DFS algorithm can be modified to classify edges as it encounters them. The key idea is that each edge (u, v) can be classified by the color of the vertex v that is reached when the edge is first explored (except that forward and cross edges are not distinguished): [1]

1. WHITE indicates a tree edge,
2. GRAY indicates a back edge, and
3. BLACK indicates a forward or cross edge.









22.4

DFS Applications

Topological Sort

- DFS can be used to perform a topological sort of a directed acyclic graph. [1]
- It is a linear ordering of all vertices.
- It is different from the usual kind of sorting.

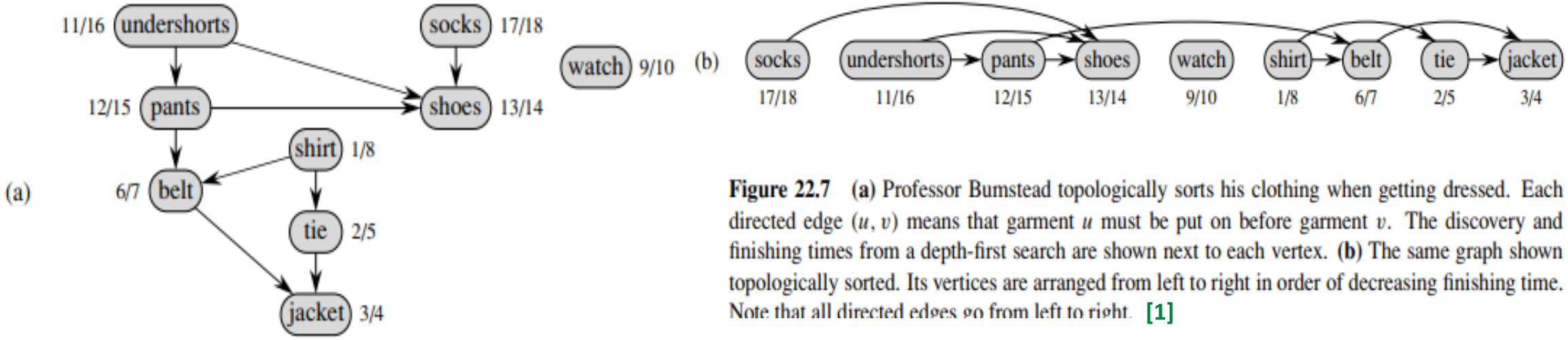
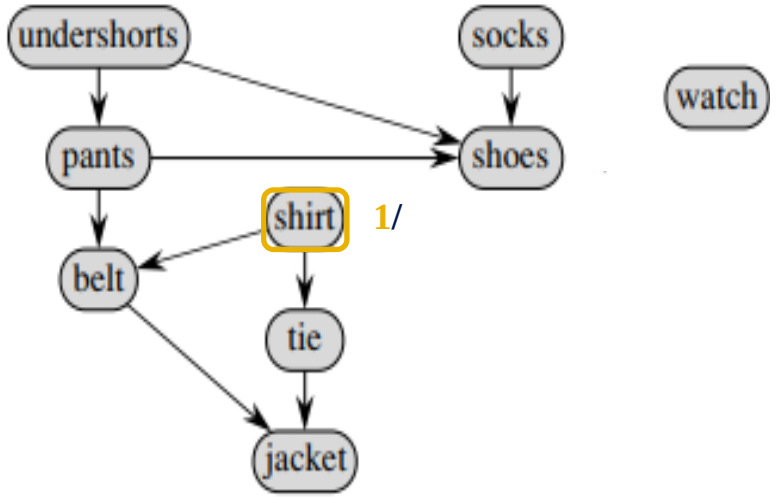


Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right. [1]

Topological Sort

TOPOLOGICAL-SORT(G) [1]

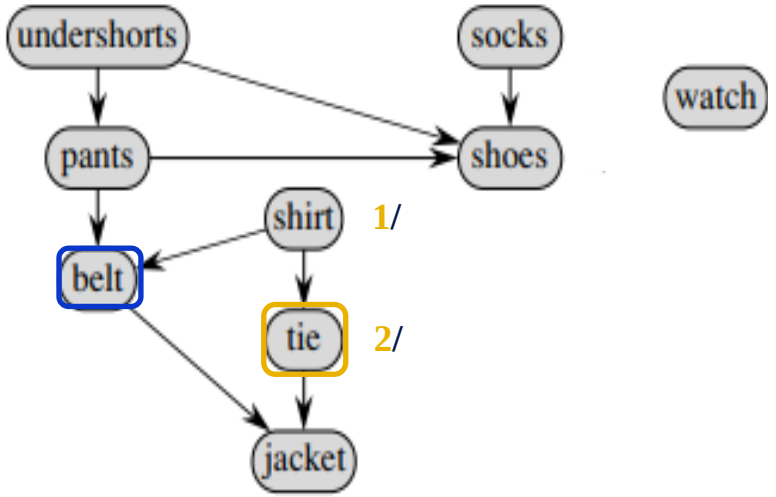
1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices



(a.1)

List

550 Chapter 22 Elementary Graph Algorithms



(a.2)

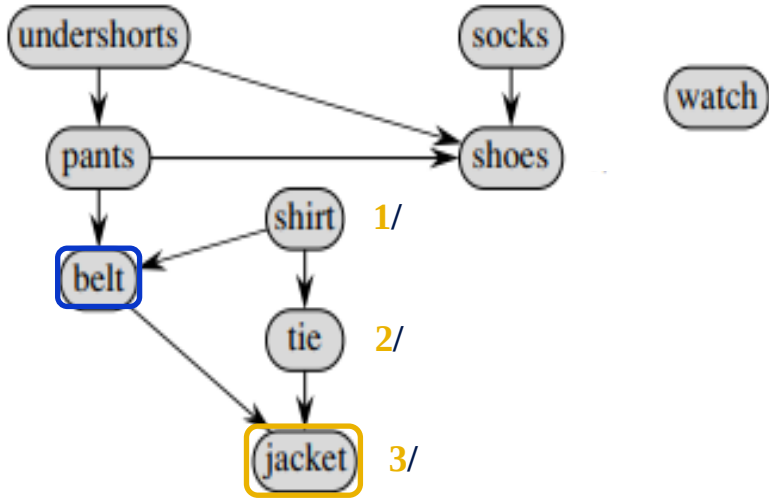
List

Topological Sort

TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

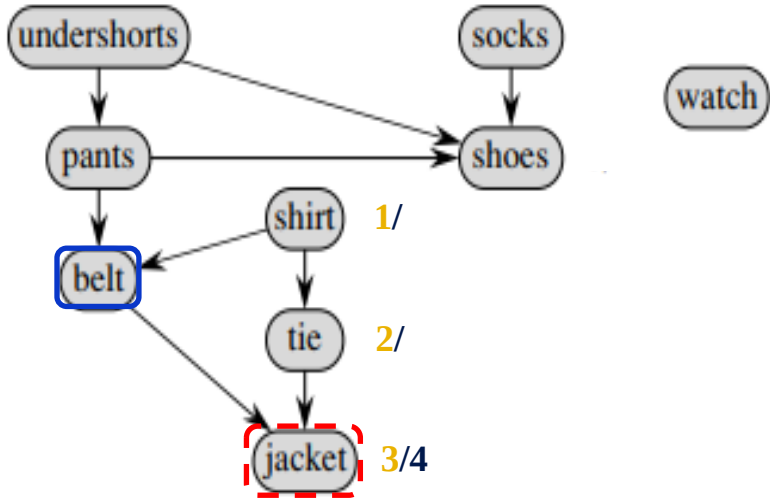
(a.3)



List

550 Chapter 22 Elementary Graph Algorithms

(a.4)



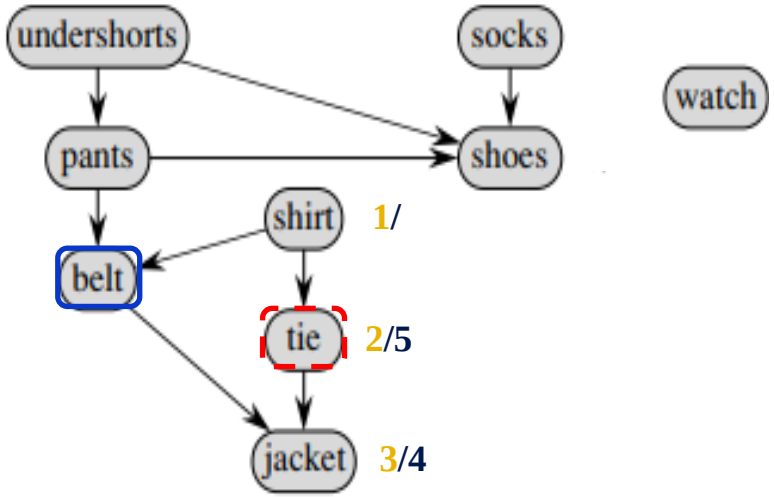
List → jacket -

Topological Sort

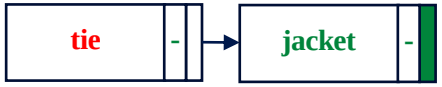
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.5)

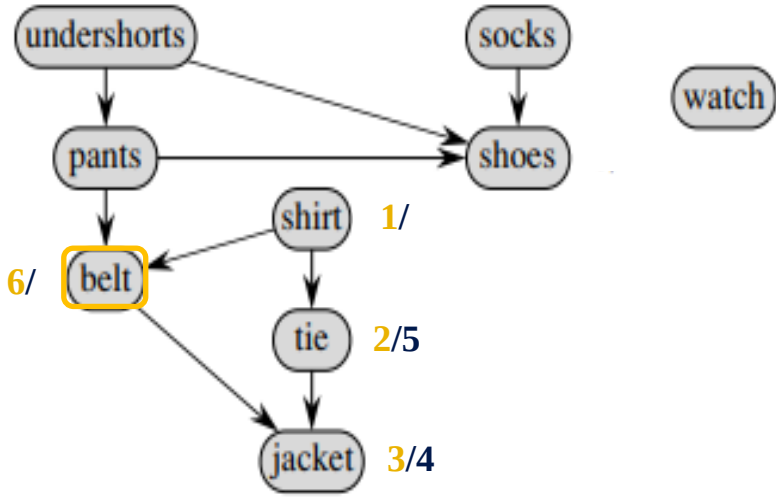


(b)

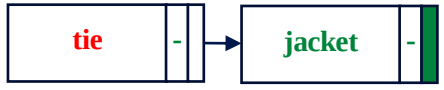


550 Chapter 22 Elementary Graph Algorithms

(a.6)



(b)

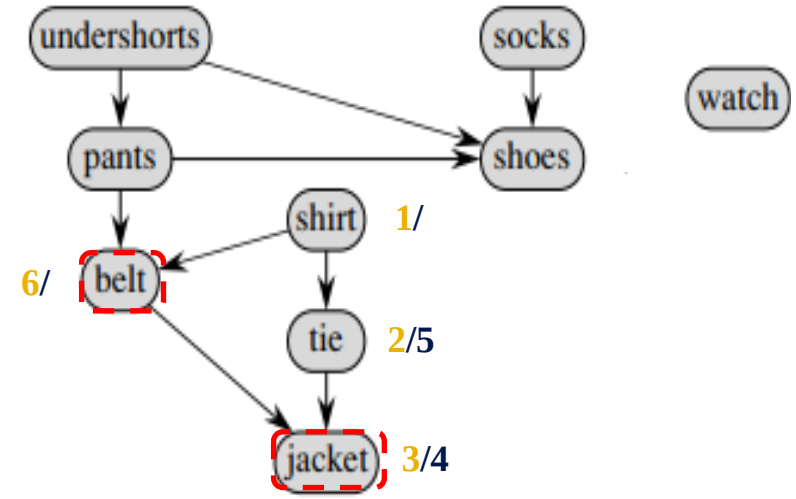


Topological Sort

TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.7)



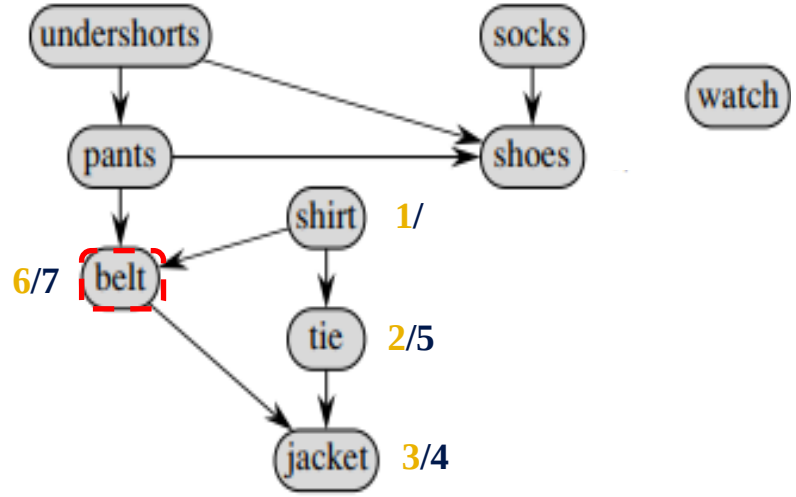
(b)

Topological Sort

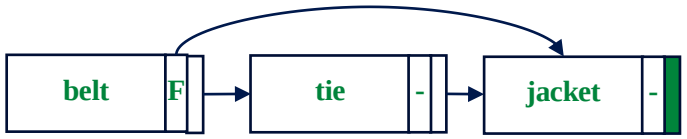
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.8)



(b)

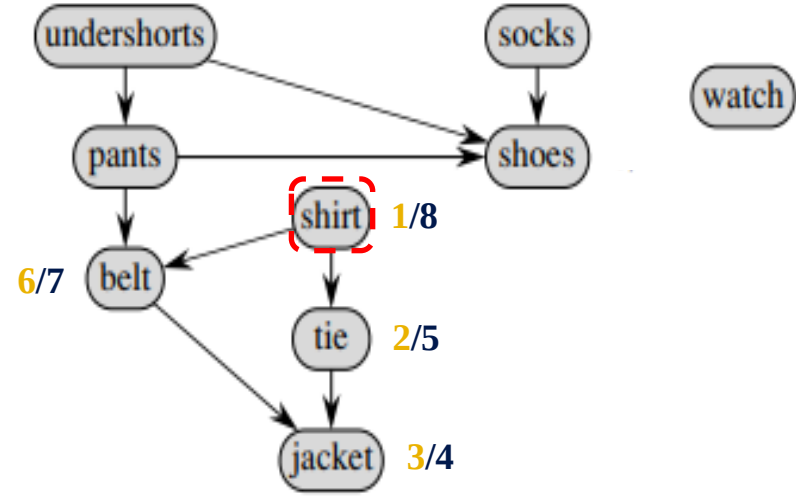


Topological Sort

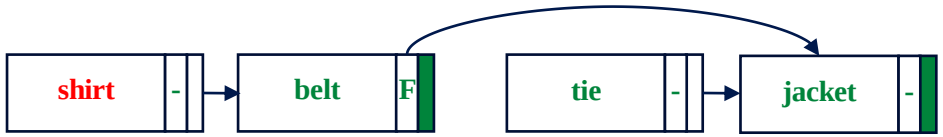
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.9)



(b)

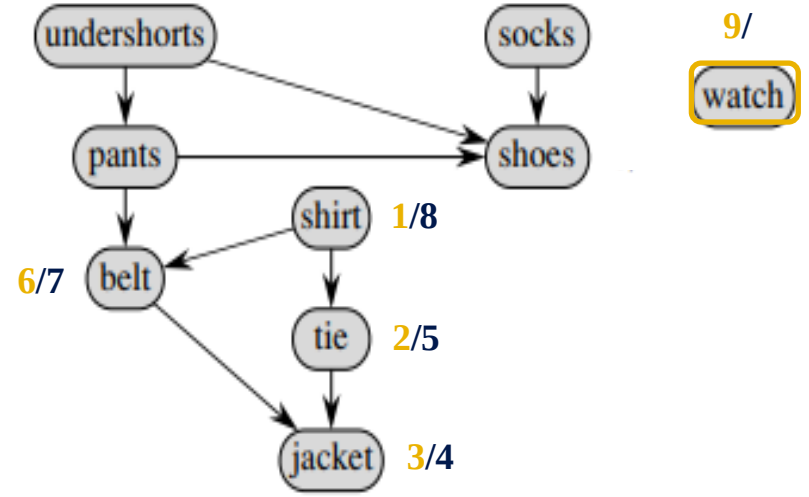


Topological Sort

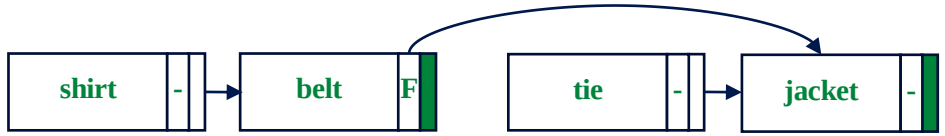
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.10)



(b)

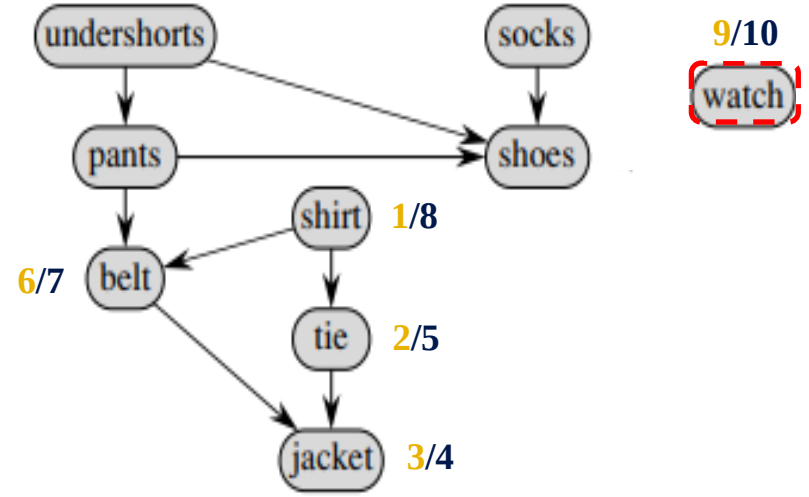


Topological Sort

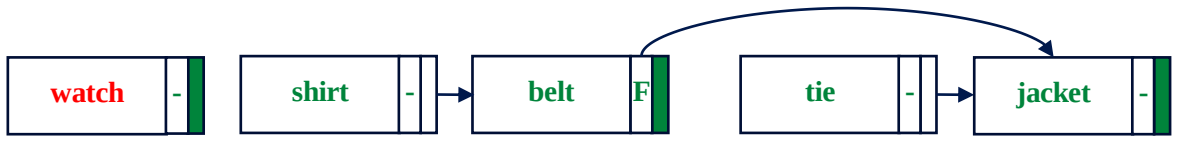
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.11)



(b)

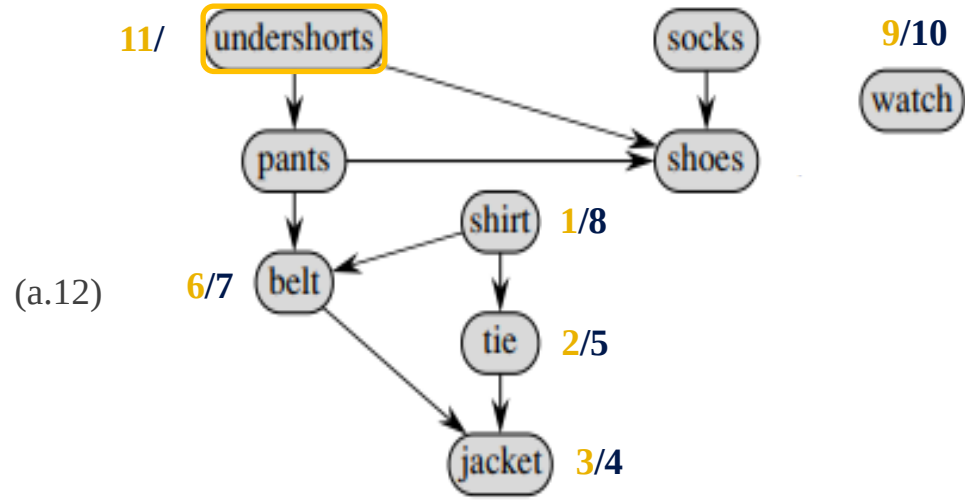


Topological Sort

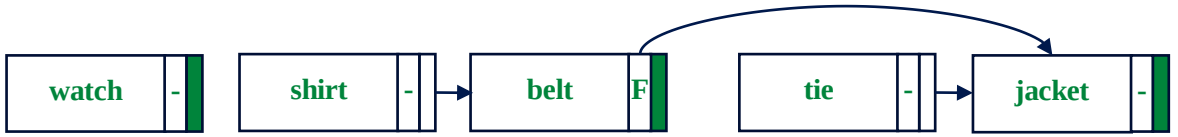
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.12)



(b)

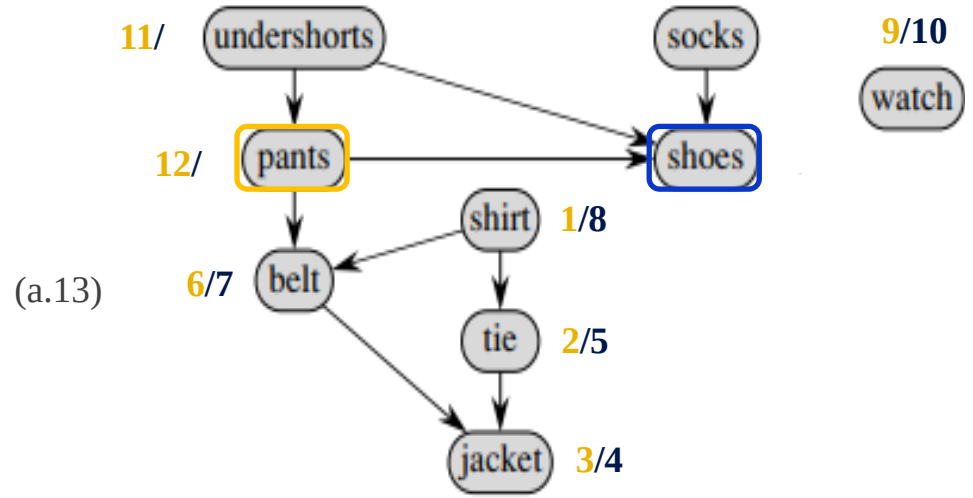


Topological Sort

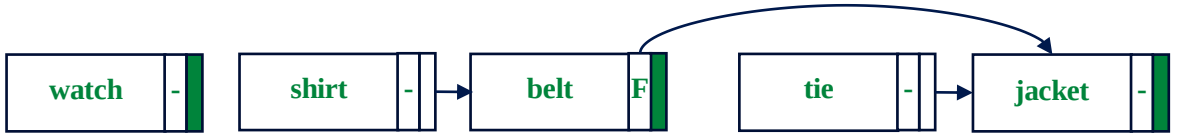
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.13)



(b)

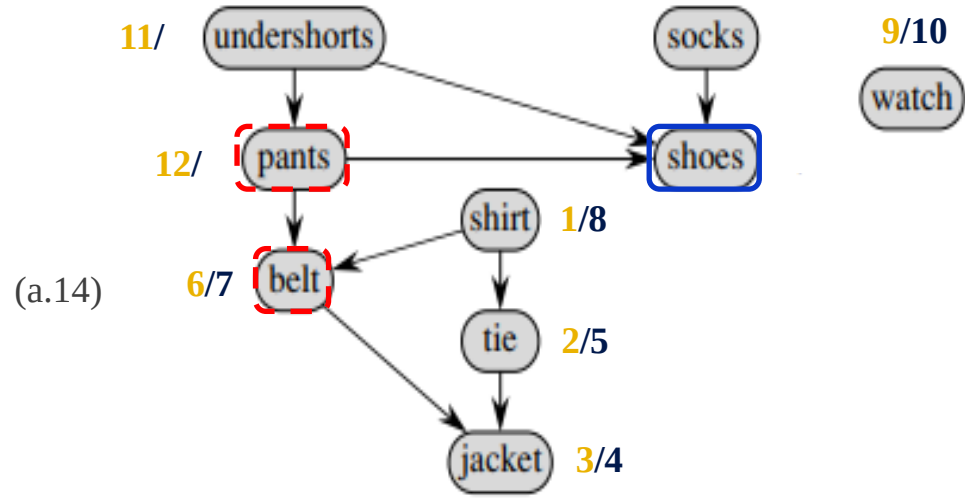


Topological Sort

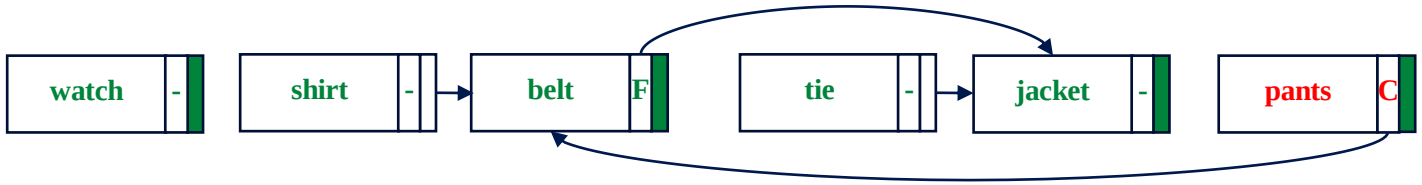
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.14)



(b)

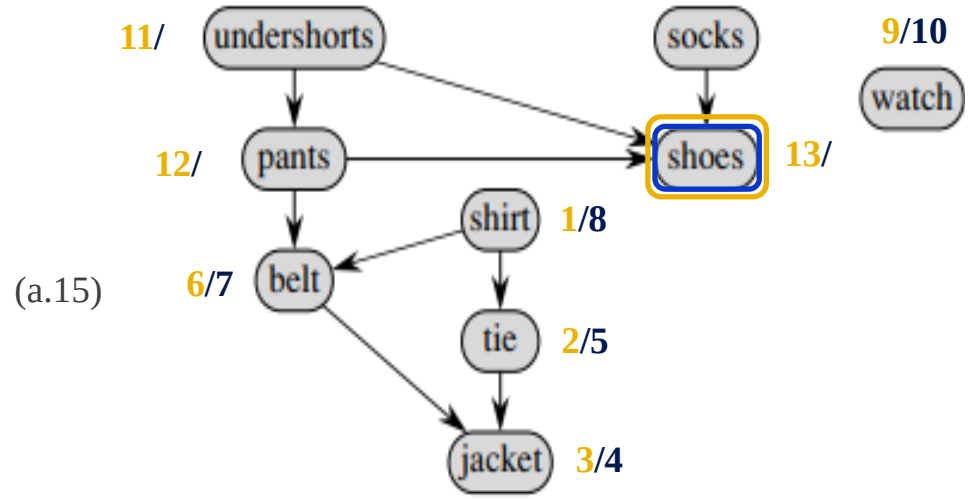


Topological Sort

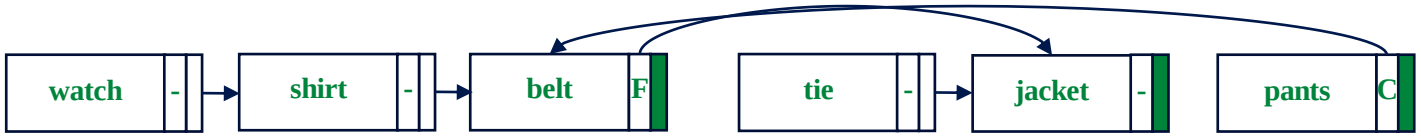
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.15)



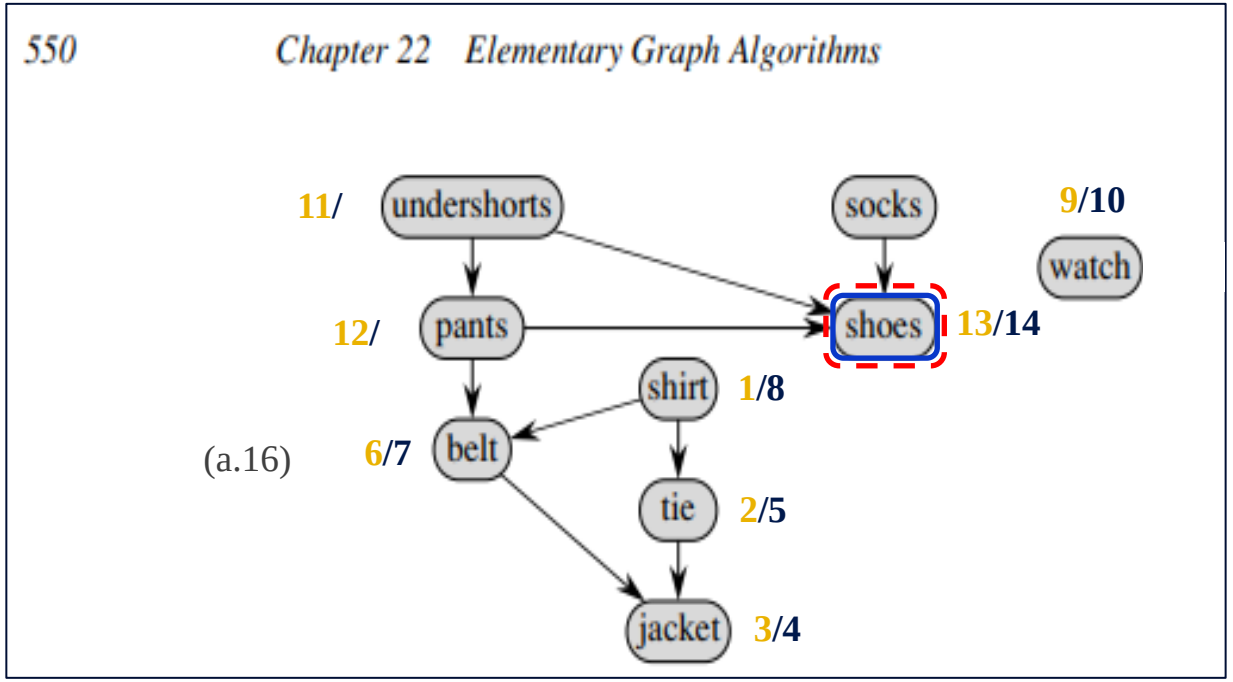
(b)



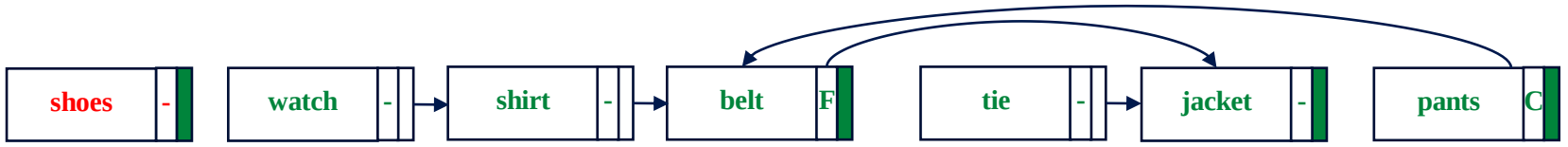
Topological Sort

TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices



(a.16)

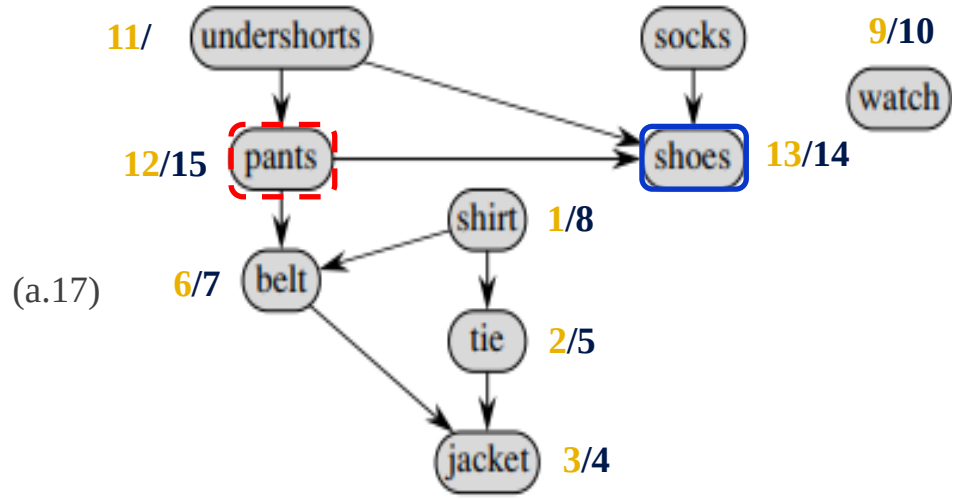


Topological Sort

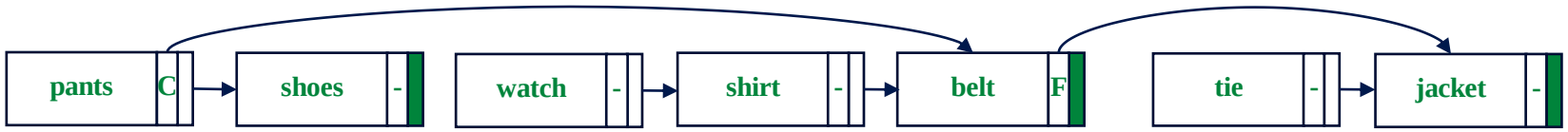
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.17)



(b)

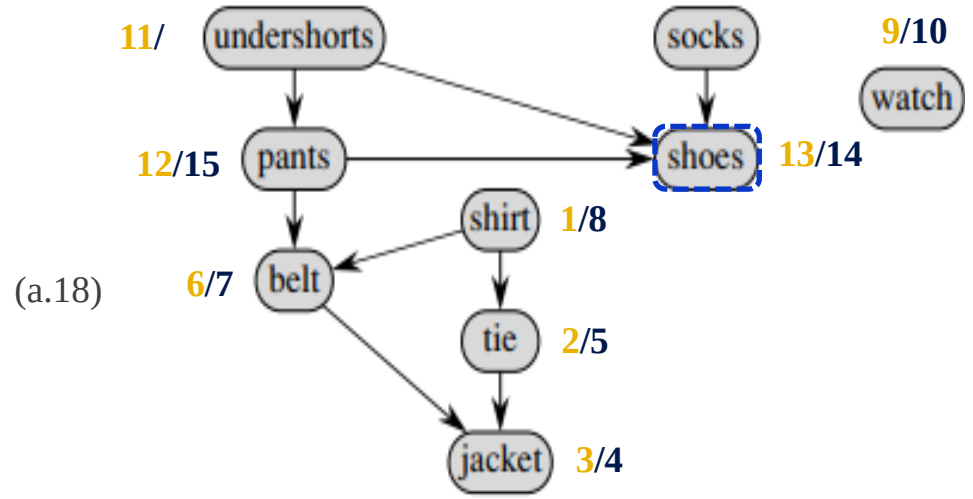


Topological Sort

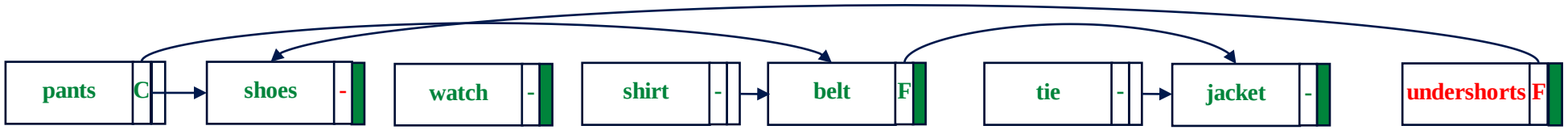
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.18)



(b)

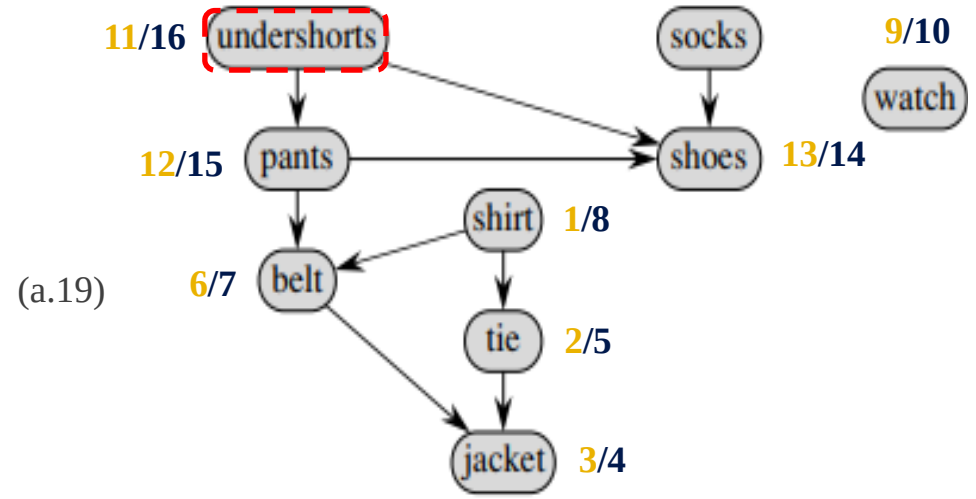


Topological Sort

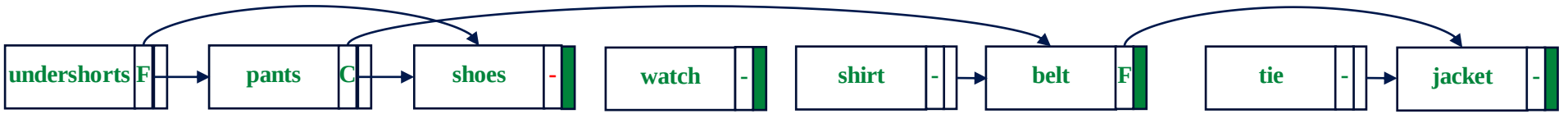
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.19)



(b)

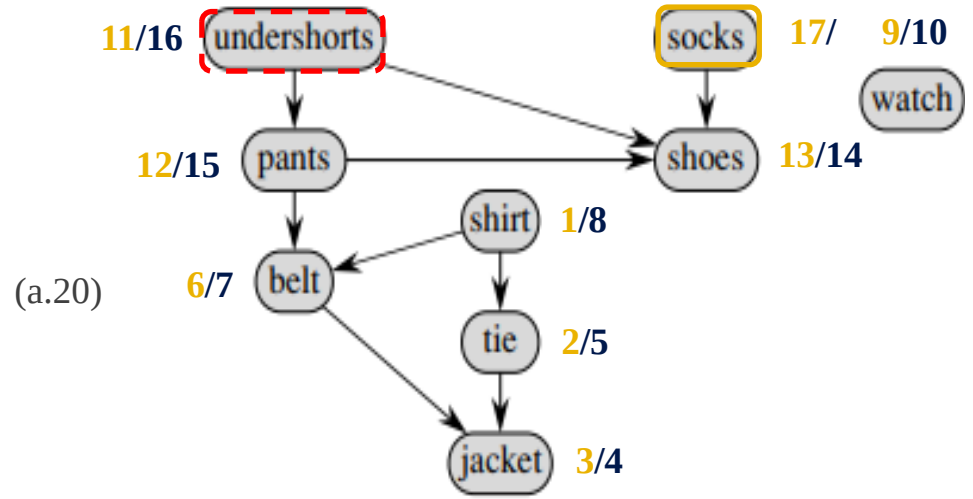


Topological Sort

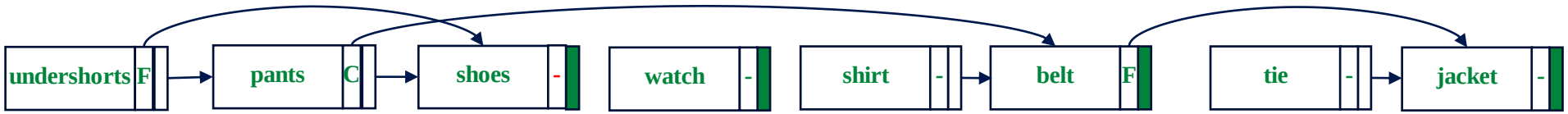
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.20)



(b)

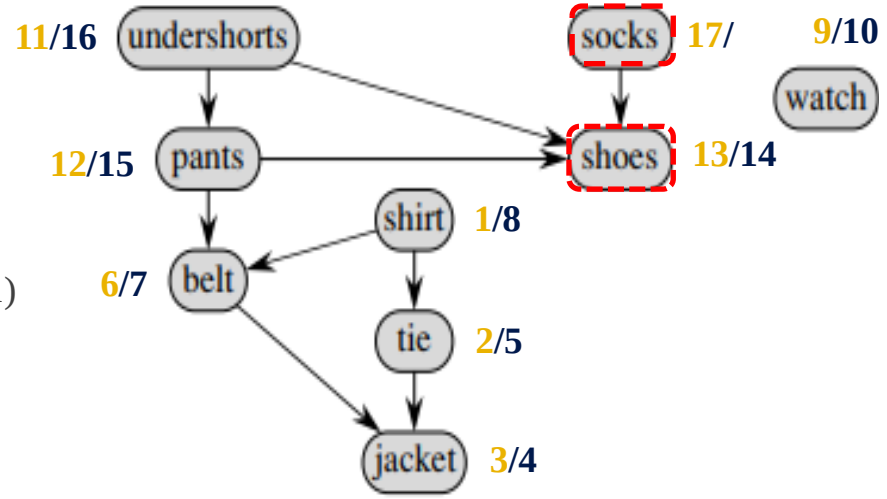


Topological Sort

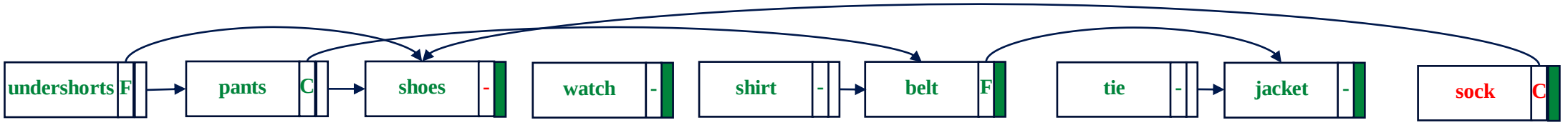
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.21)



(b)

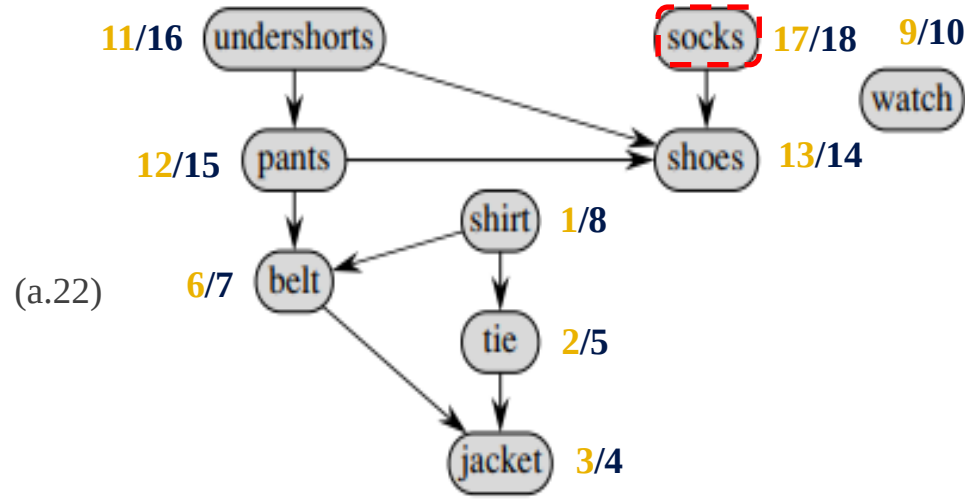


Topological Sort

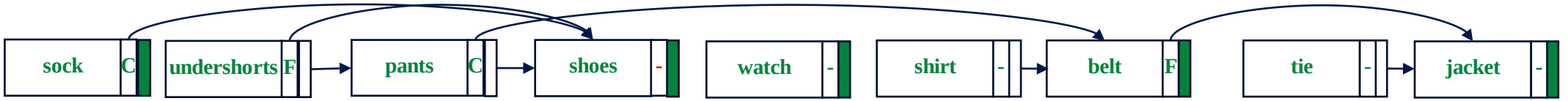
TOPOLOGICAL-SORT(G) [1]

1 call DFS(G) to compute finishing times $f[v]$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

(a.22)



(b)



Topological Sort

Running Time

We can perform a topological sort in time (V + E), since depth-first search takes (V + E) time and it takes O(1) time to insert each of the |V| vertices onto the front of the linked list.

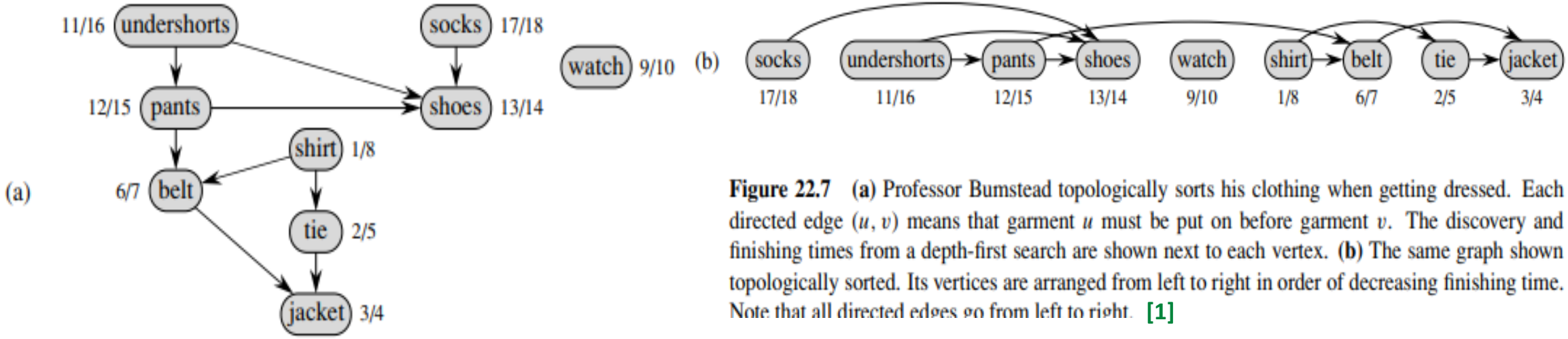


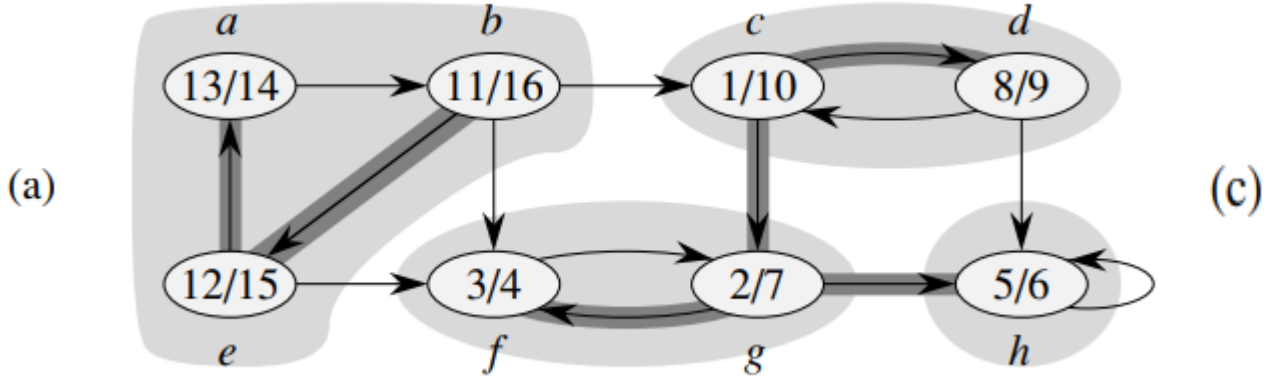
Figure 22.7 **(a)** Professor Bumstead topologically sorts his clothing when getting dressed. Each directed edge (u, v) means that garment u must be put on before garment v. The discovery and finishing times from a depth-first search are shown next to each vertex. **(b)** The same graph shown topologically sorted. Its vertices are arranged from left to right in order of decreasing finishing time. Note that all directed edges go from left to right. [1]

Strongly Connected Component

DFS Application

- It can help find cluster of highly interconnected vertices in a graph and simplify a graph (or reduced graph).
- DFS and Topological sort can be used to find strongly connected components (SCC) of a directed graph.

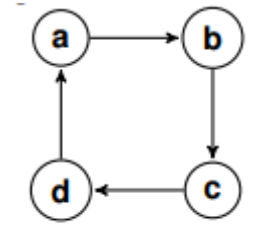
Figure 22.9 **(a)** A directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph G^T, the transpose of G. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component. [1]



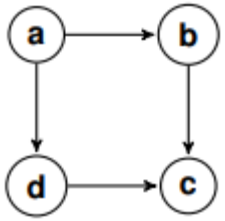
Strongly Connected Component

Connectness in digraph

- **Strongly connected**
  - If there is a path from u to v and from v to u whenever u and v are vertices in the graph.
- **Weakly connected**
  - If there is a path between every two vertices in the underlying directed graph.
- **Connected components**
  - The subgraph of a directed graph that are strongly connected but not contained in lager strongly connected subgraphs.
  - It is the maximal strongly connected subgraphs.
  - It is called the strongly connected component.

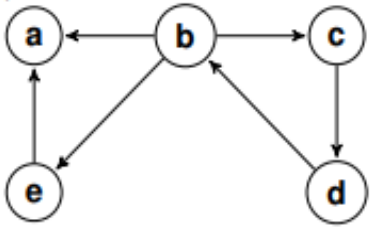


Strongly connected



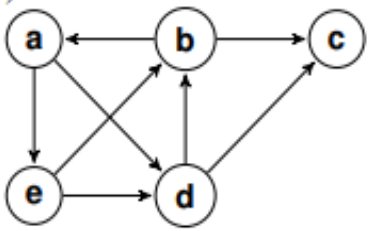
Weakly connected

a)



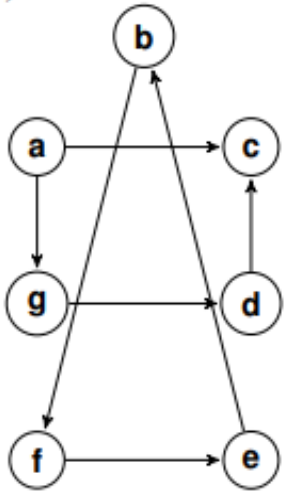
This graph is not strongly connected because there are no paths that start with a. This graph is weakly connected because the underlying undirected graph is connected.

b)

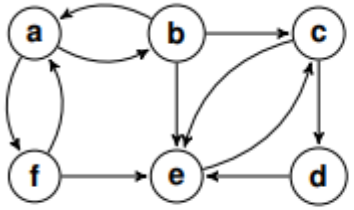


This graph is not strongly connected because there are no paths that start with c. This graph is weakly connected because the underlying undirected graph is connected.

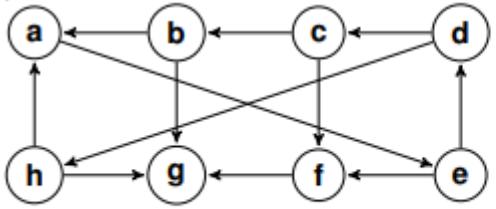
c)



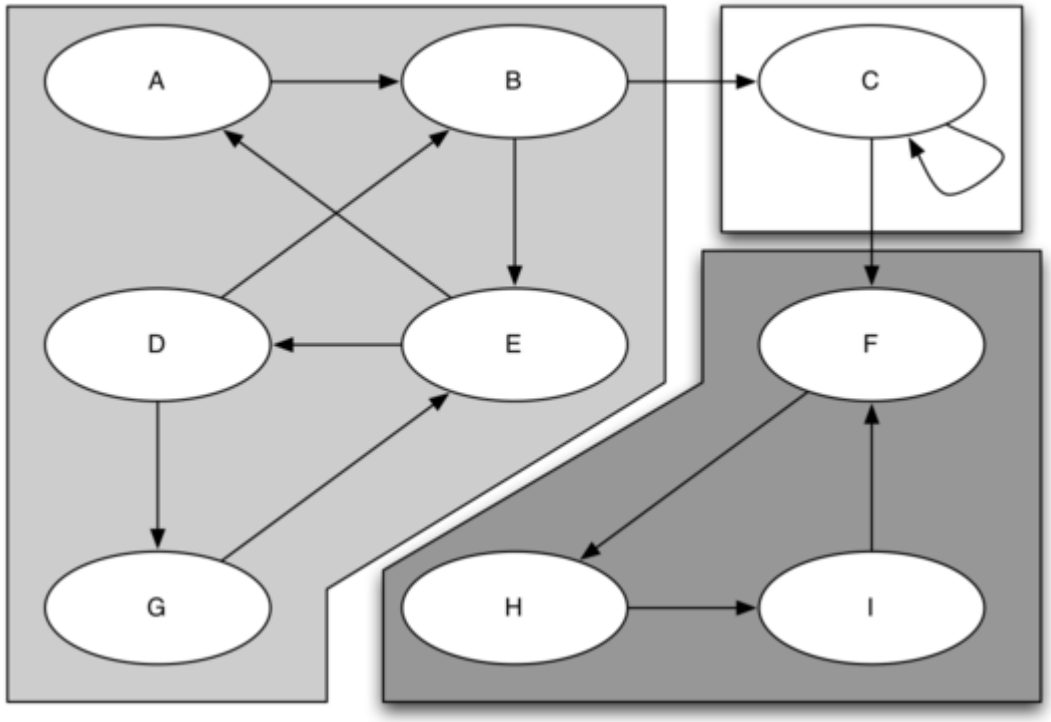
This graph is not strongly connected nor weakly connected because the underlying undirected graph is not connected.



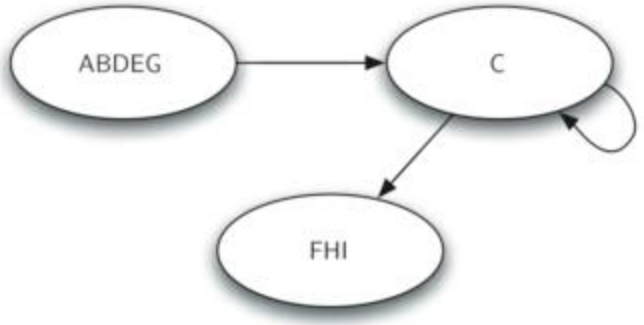
$\{a, b, f\}$ and $\{c, d, e\}$ are the strongly connected components of the graph.



$\{a, b, c, d, e, h\}$, $\{f\}$, and $\{g\}$ are the strongly connected components of the graph.



A Directed Graph with Three Strongly Connected Components [3]

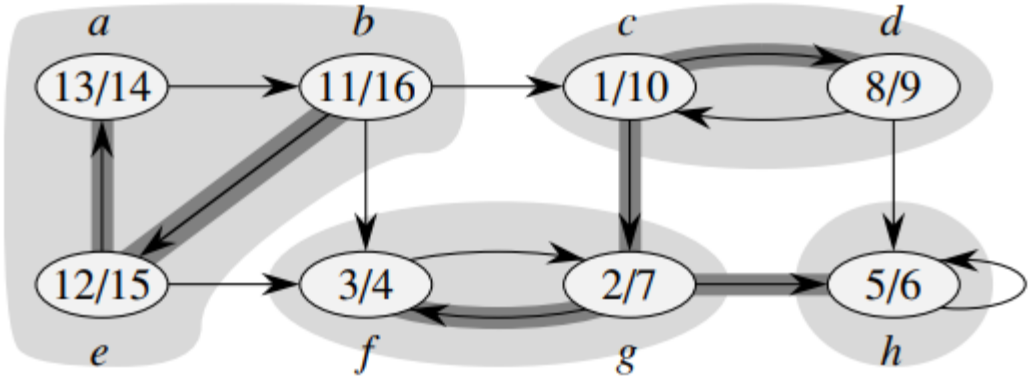


The Reduced Graph [3]

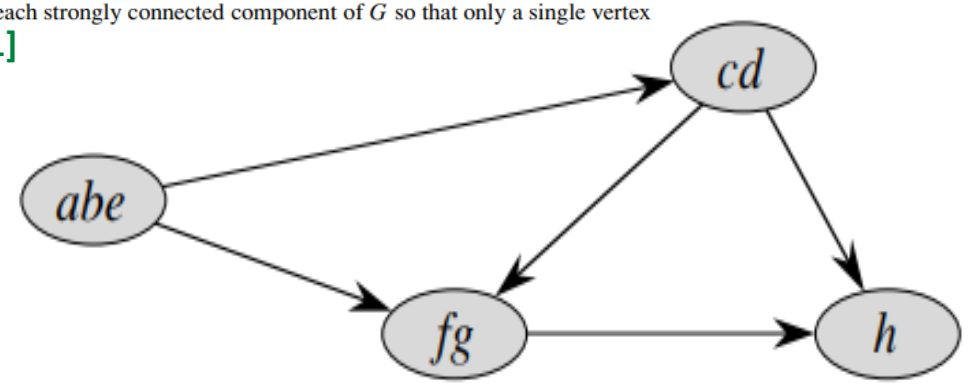
553

Figure 22.9 **(a)** A directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph G^T, the transpose of G. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component. [1]

(a)



(c)



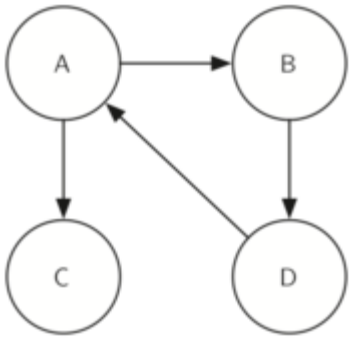
Strongly Connected Component

DFS Application

Chapter 22 Elementary Graph Algorithms [1]

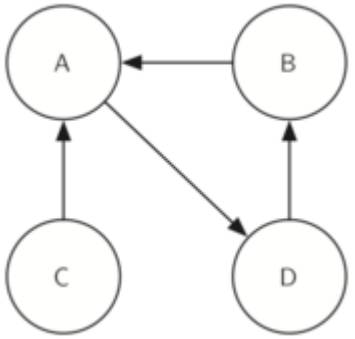
STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $f[u]$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component



A Graph G

	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	1	0	0	0

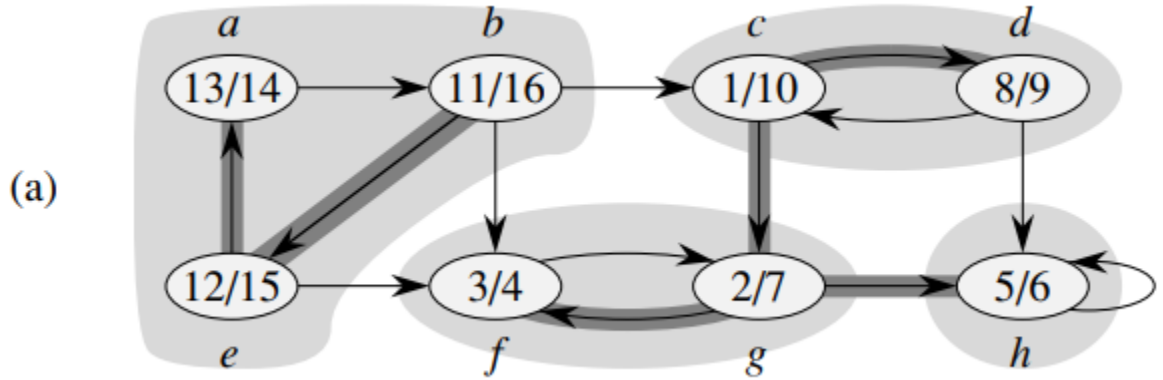


Its Transpose G^T

	A	B	C	D
A	0	0	0	1
B	1	0	0	0
C	1	0	0	0
D	0	1	0	0

Strongly Connected Component

DFS Application



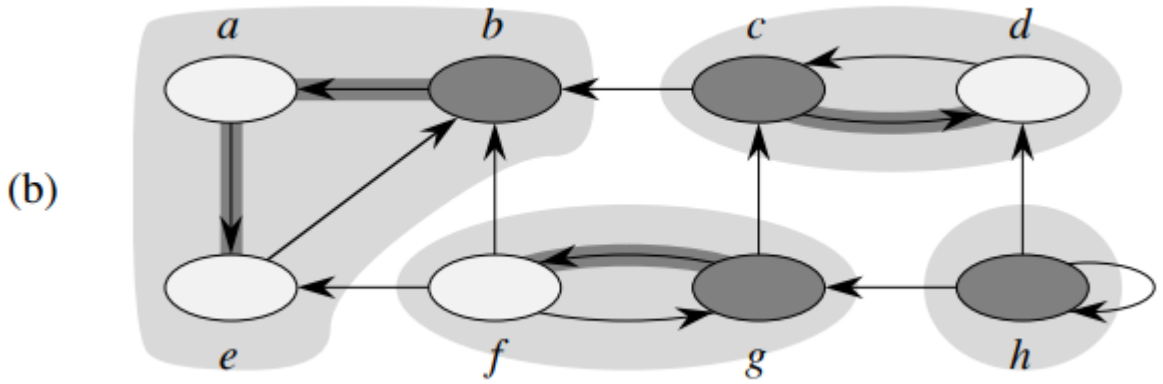
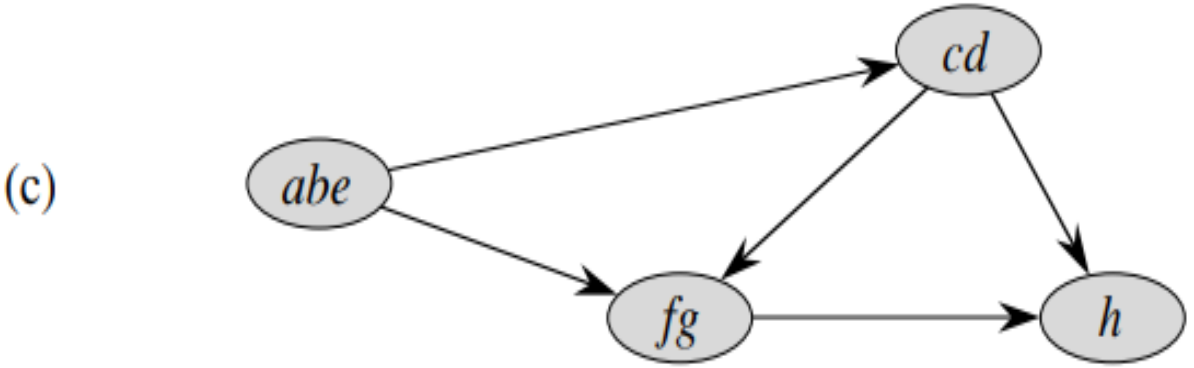


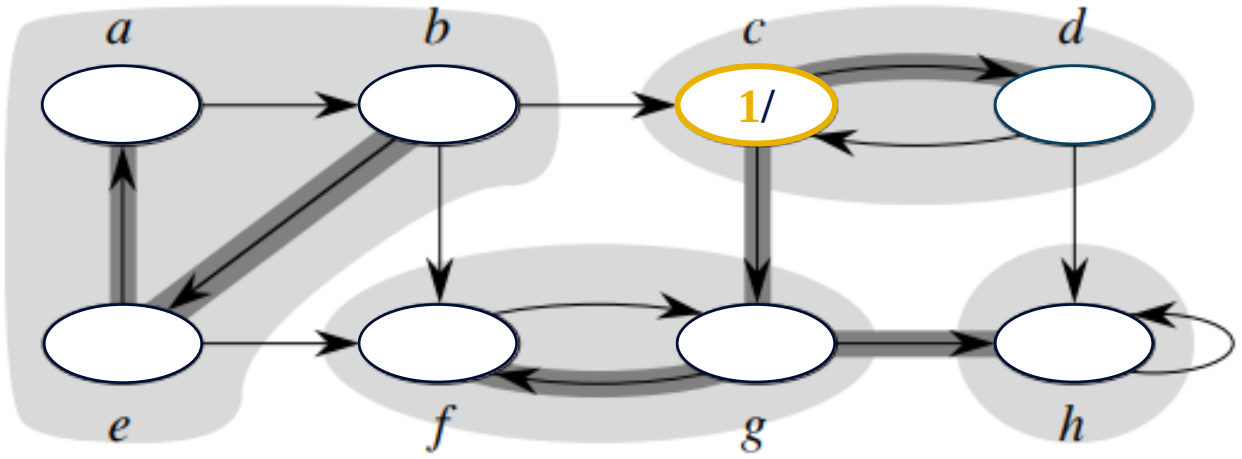
Figure 22.9 **(a)** A directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph G^T, the transpose of G. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component. [1]



Strongly Connected Component

DFS Application

(a.1)

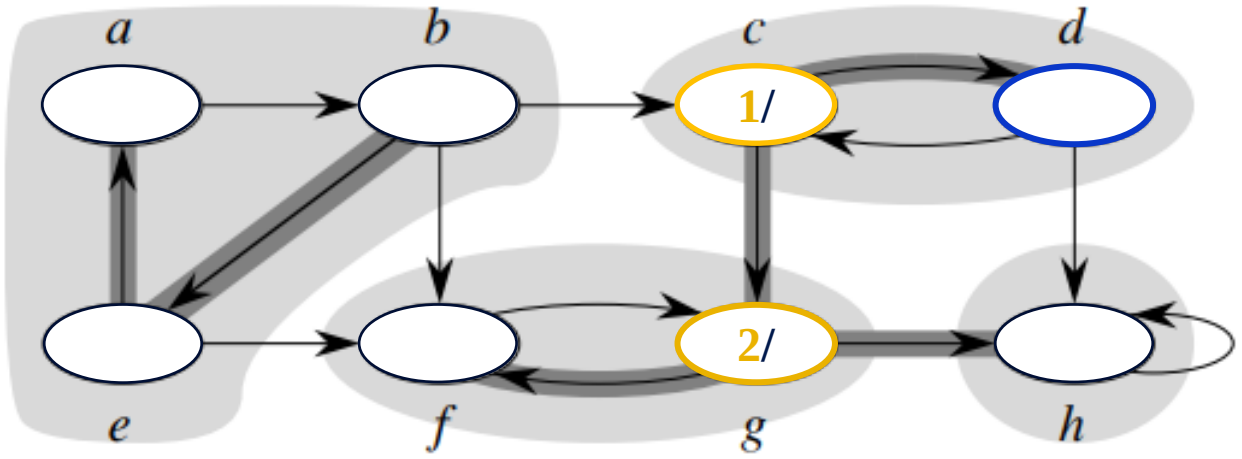


Stack | - |

Strongly Connected Component

DFS Application

(a.2)

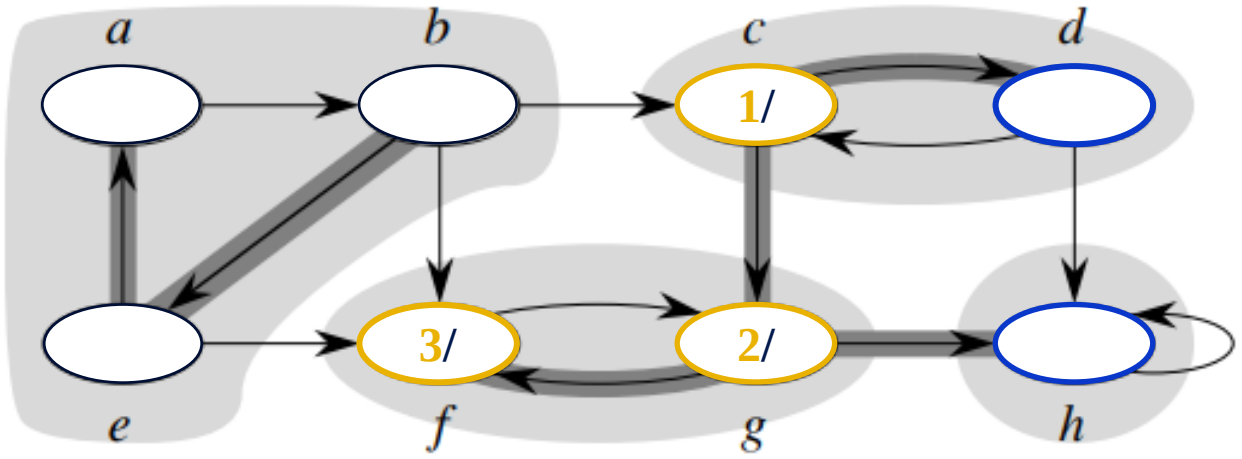


Stack | - |

Strongly Connected Component

DFS Application

(a.3)

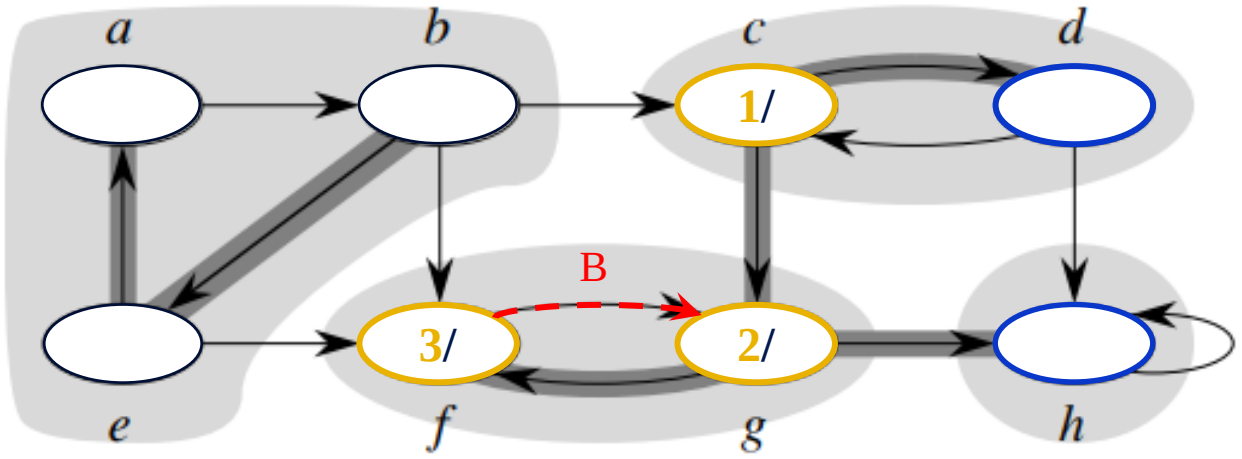


Stack | - |

Strongly Connected Component

DFS Application

(a.4)

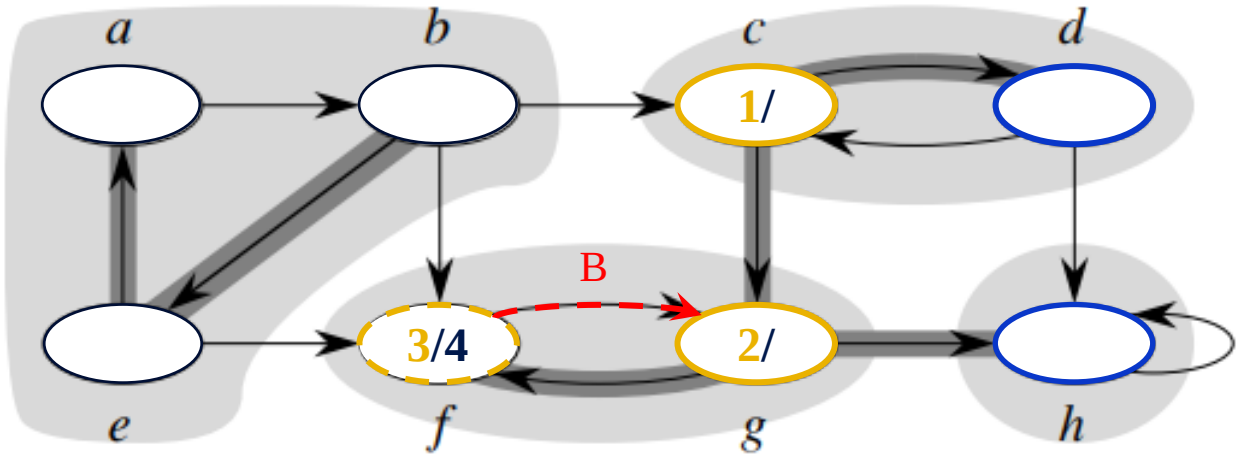


Stack -

Strongly Connected Component

DFS Application

(a.5)

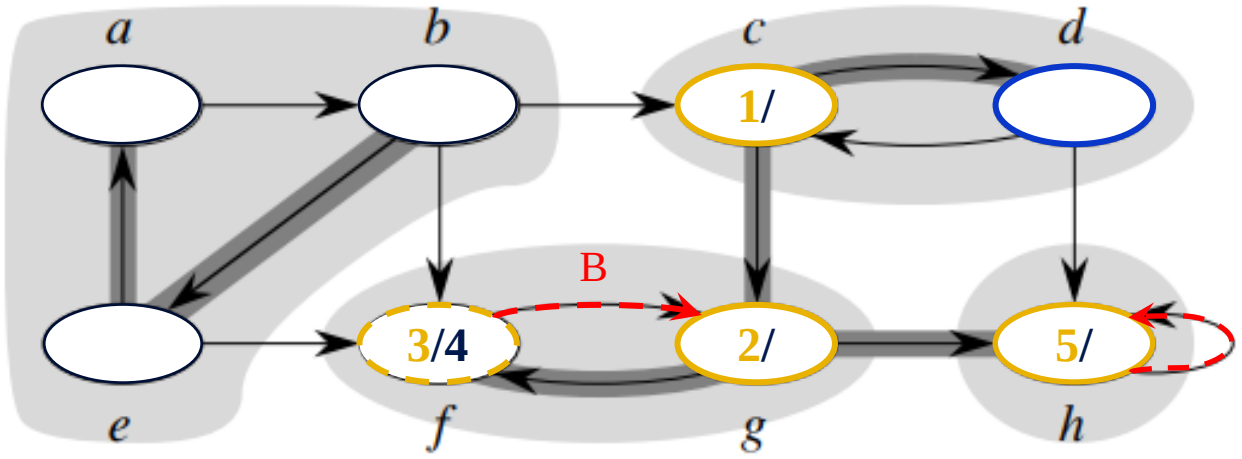


Stack | f |

Strongly Connected Component

DFS Application

(a.6)

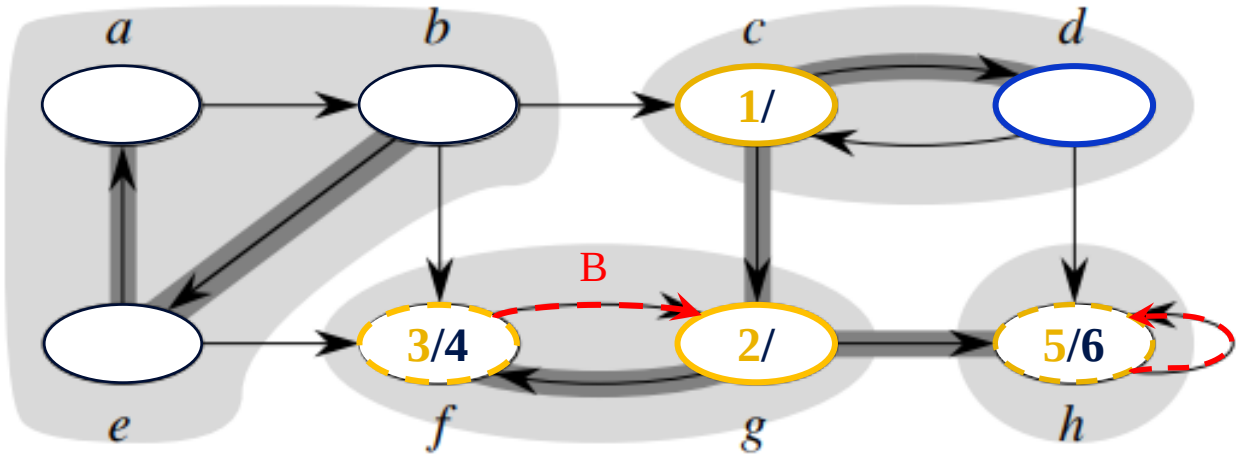


Stack | f |

Strongly Connected Component

DFS Application

(a.7)

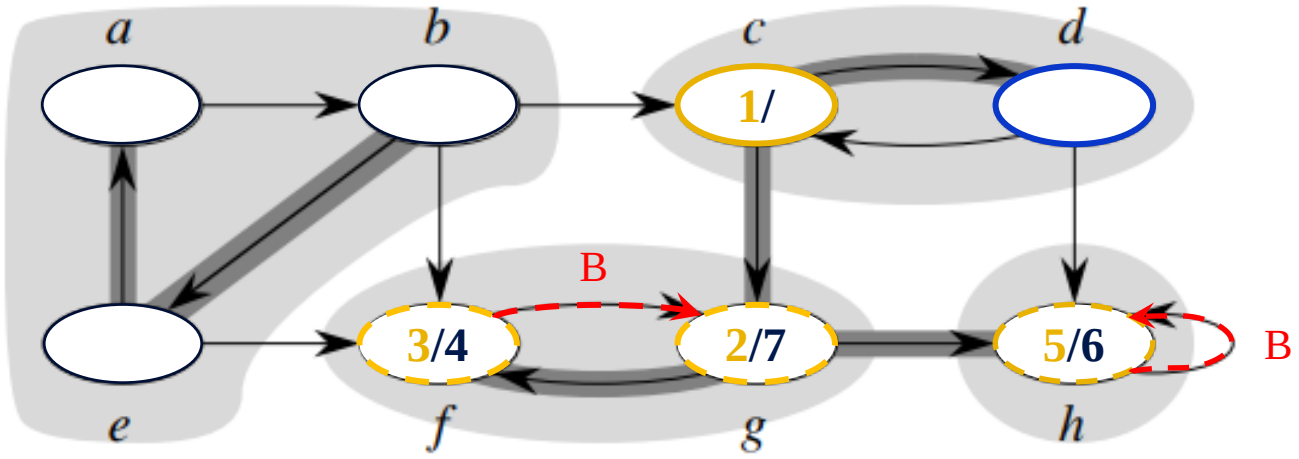


Stack | f | h |

Strongly Connected Component

DFS Application

(a.8)

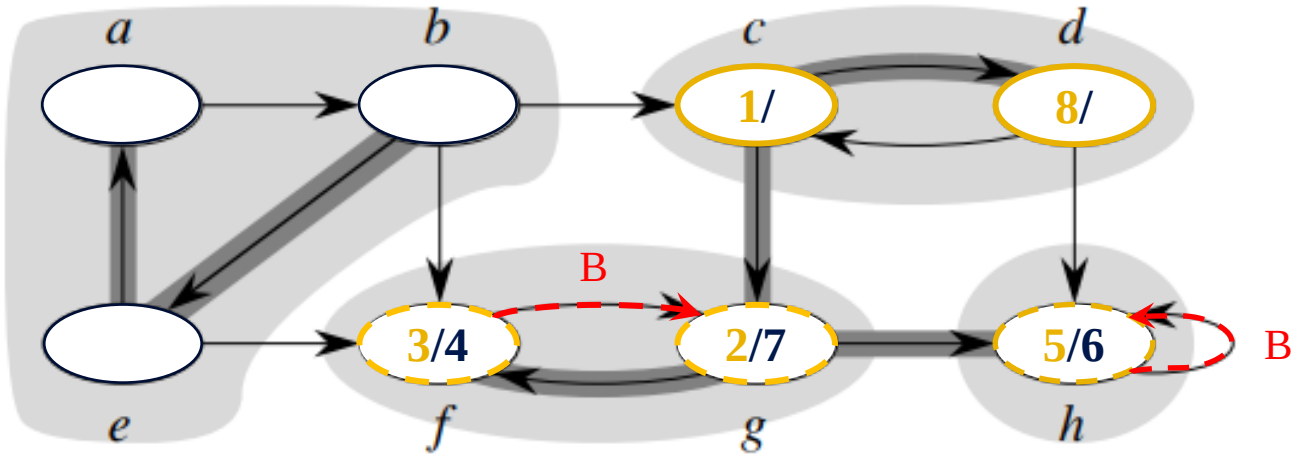


Stack | f | h |

Strongly Connected Component

DFS Application

(a.9)

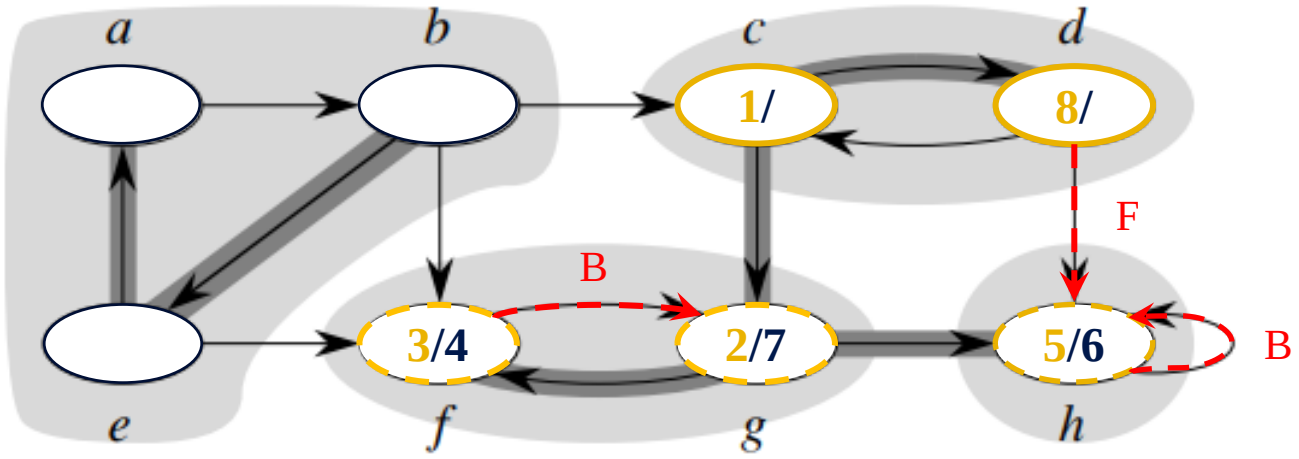


Stack | f | h |

Strongly Connected Component

DFS Application

(a.10)

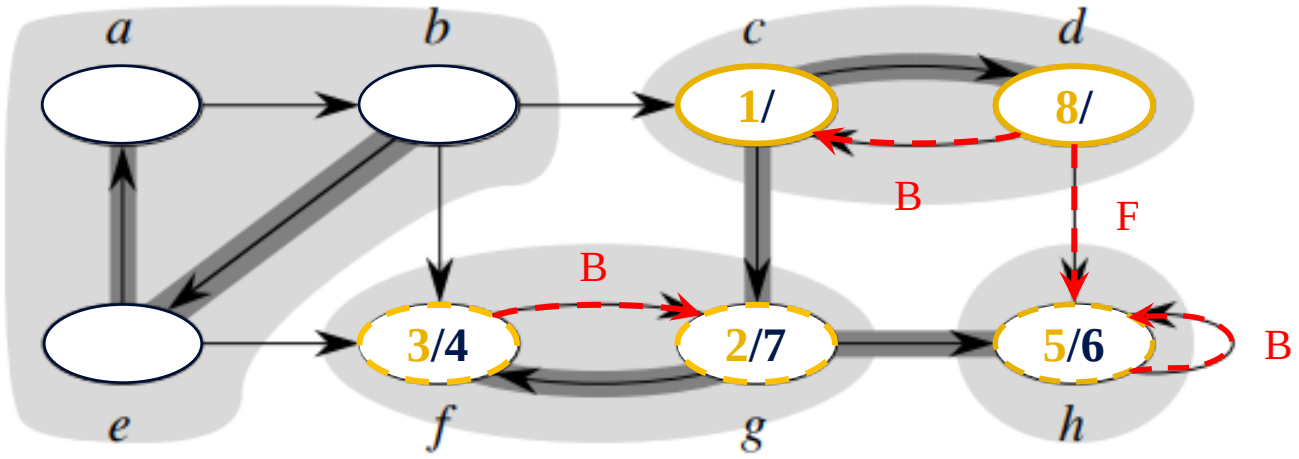


Stack | f | h |

Strongly Connected Component

DFS Application

(a.11)

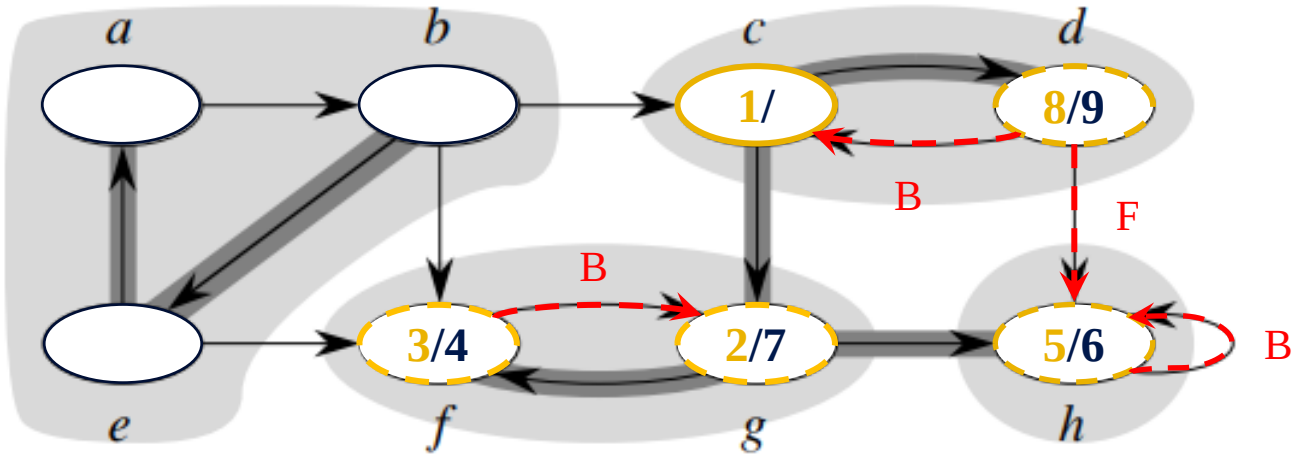


Stack | f | h |

Strongly Connected Component

DFS Application

(a.12)



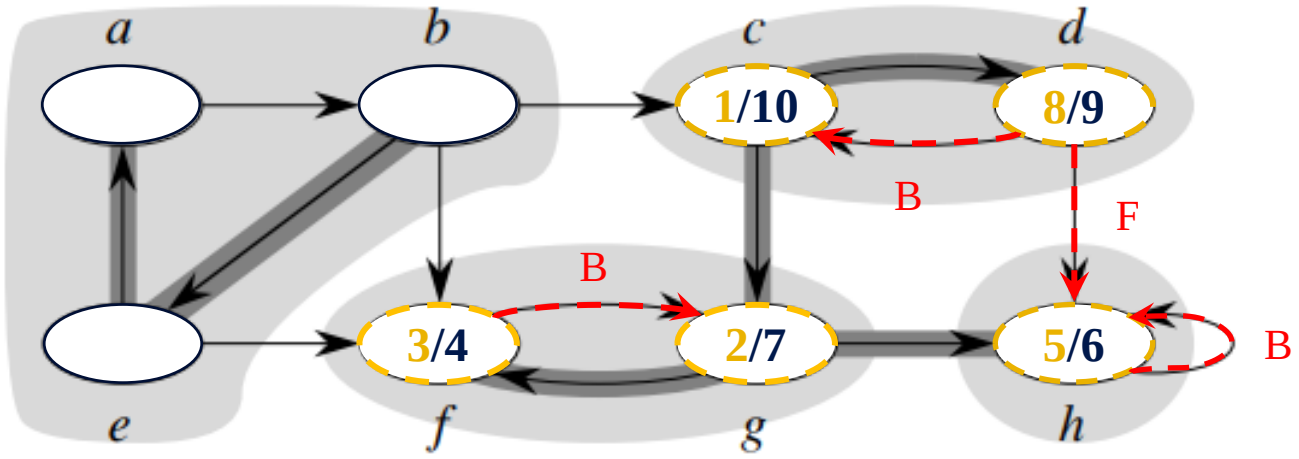
Stack

f	h	d

Strongly Connected Component

DFS Application

(a.13)



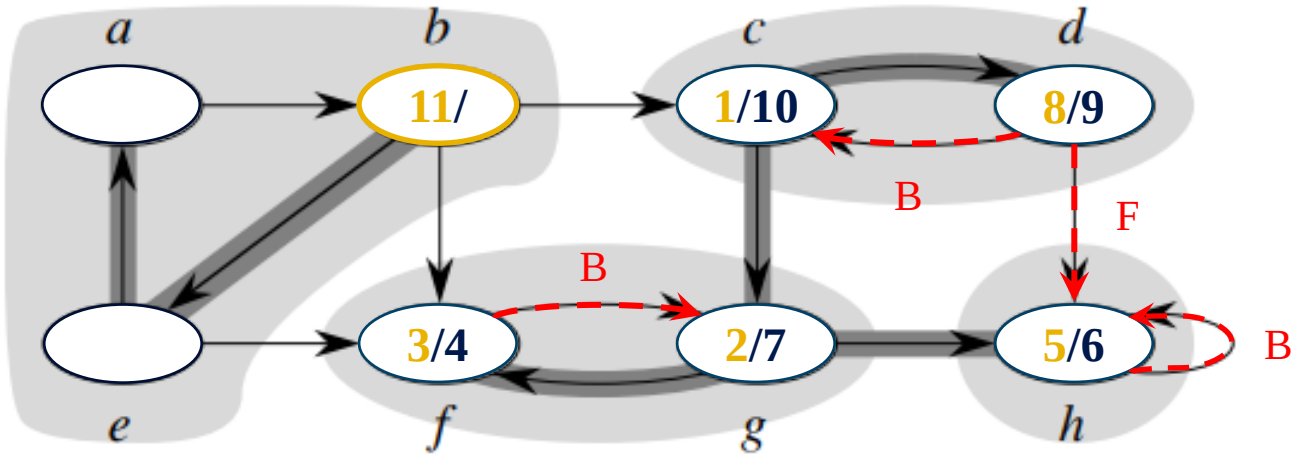
Stack

f	h	d

Strongly Connected Component

DFS Application

(a.14)

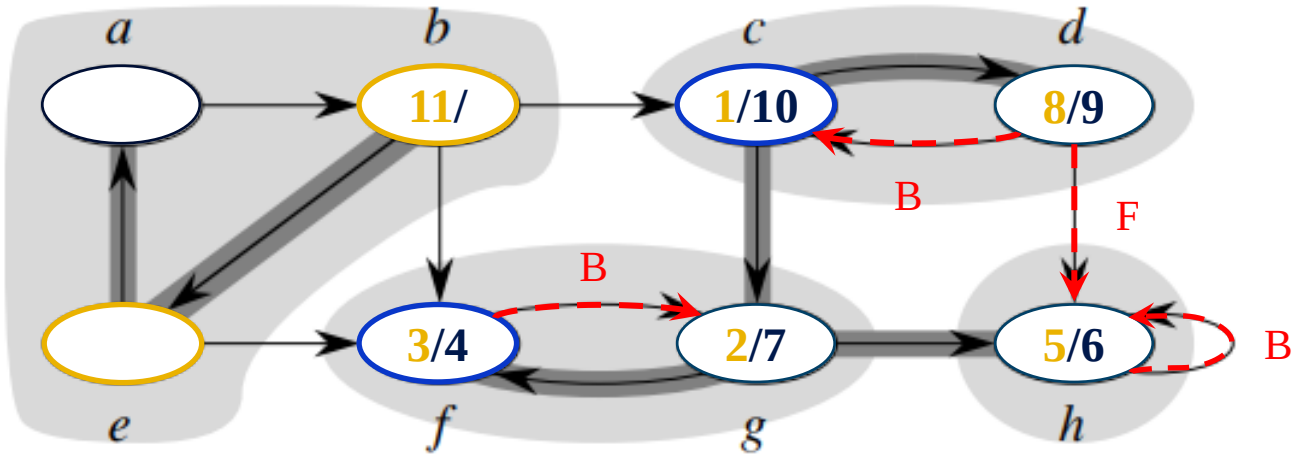


Stack | f | h | d |

Strongly Connected Component

DFS Application

(a.15)

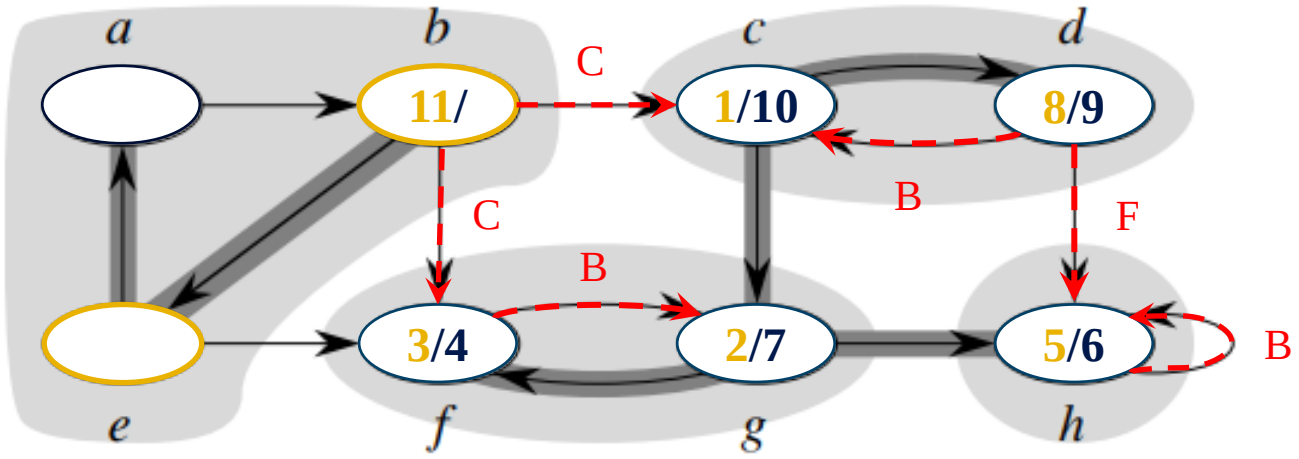


Stack | f | h | d |

Strongly Connected Component

DFS Application

(a.16)

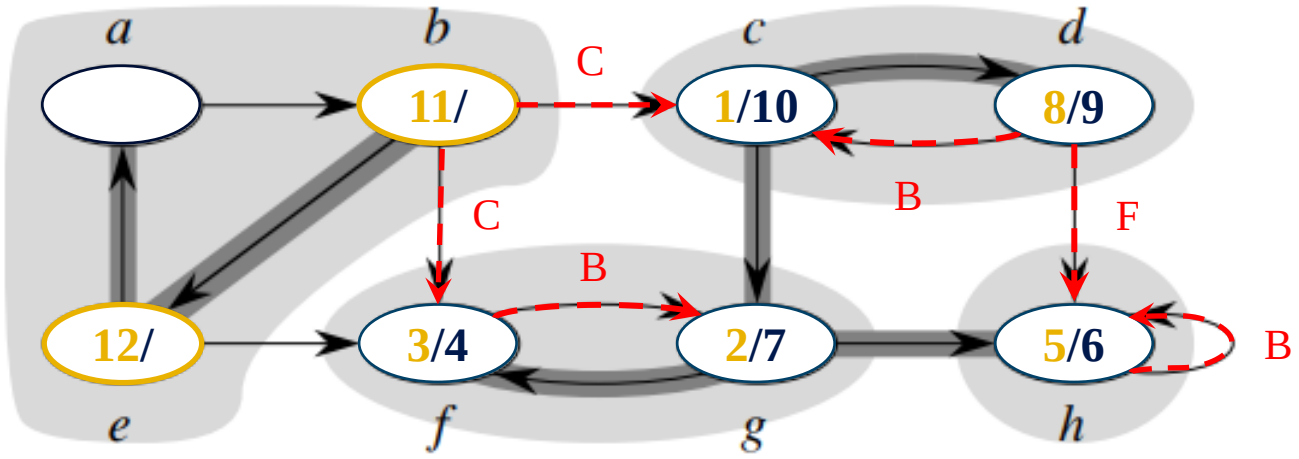


Stack | f | h | d |

Strongly Connected Component

DFS Application

(a.17)



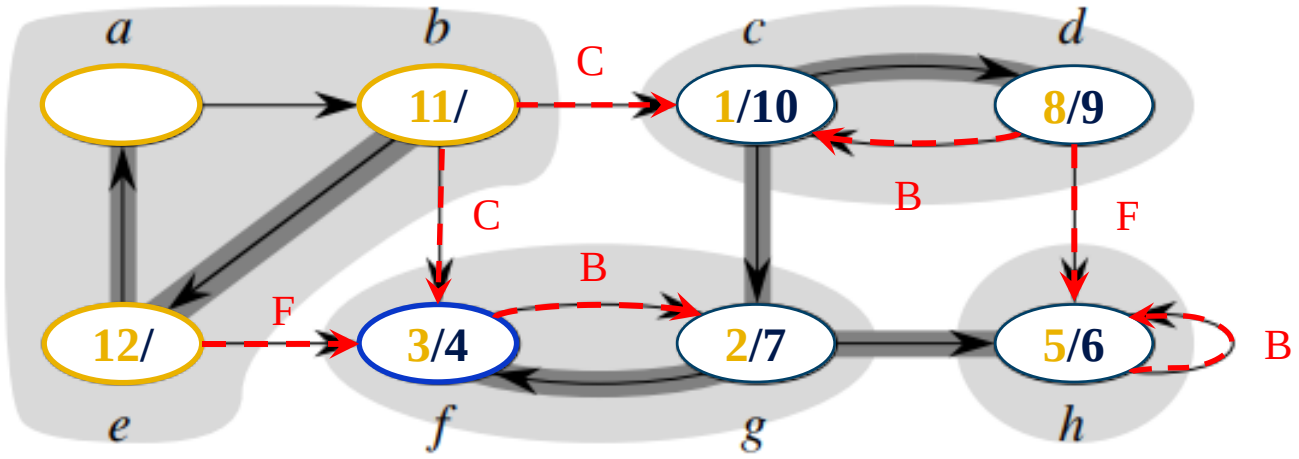
Stack

f	h	d

Strongly Connected Component

DFS Application

(a.18)



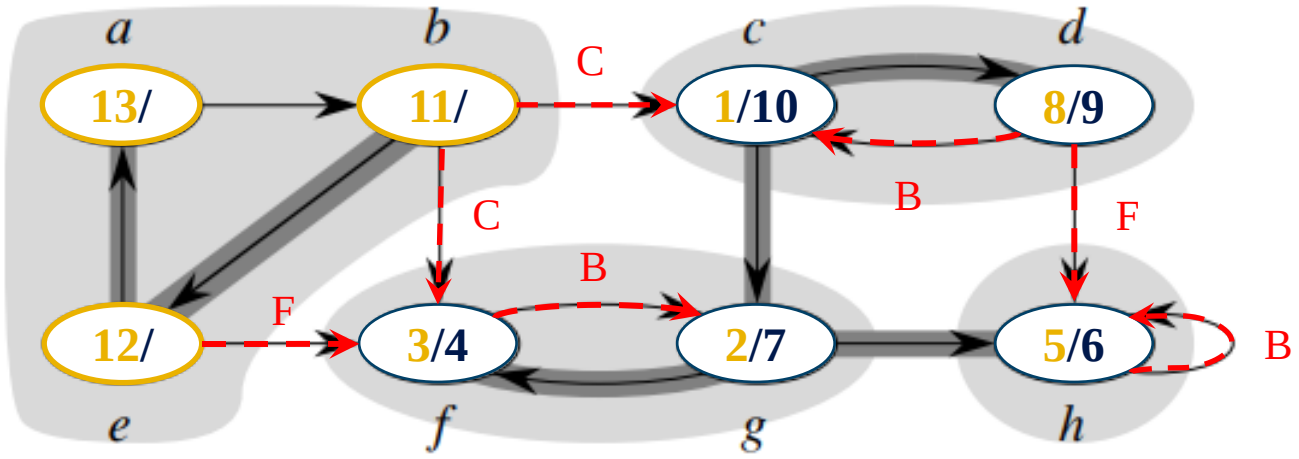
Stack

f	h	d

Strongly Connected Component

DFS Application

(a.19)



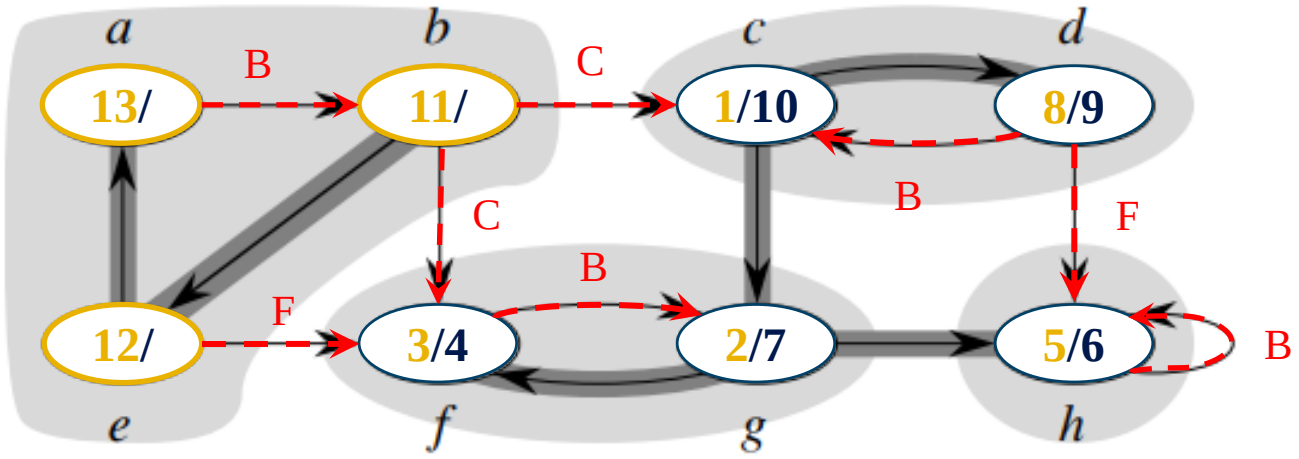
Stack

f	h	d

Strongly Connected Component

DFS Application

(a.20)



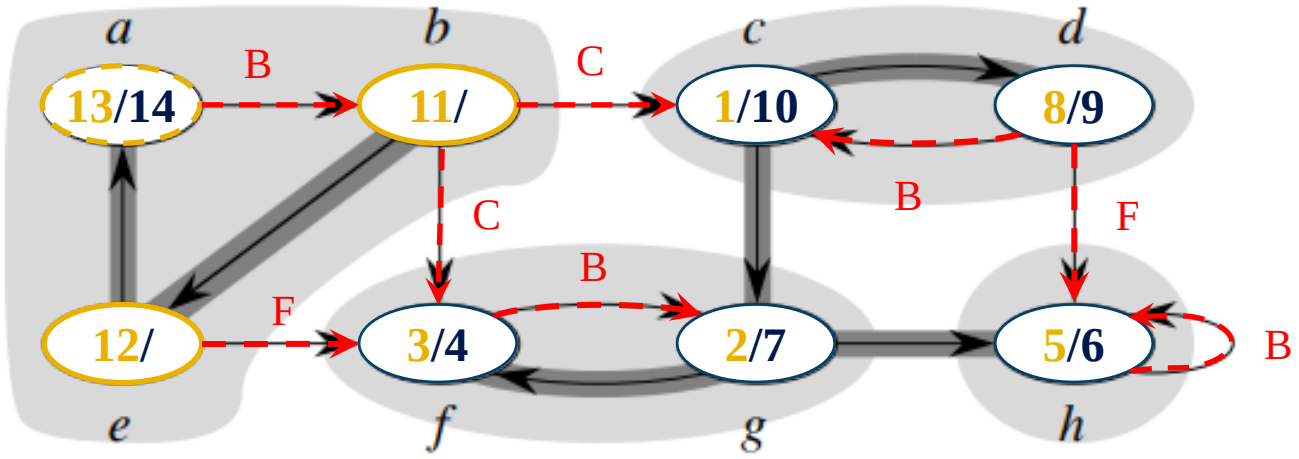
Stack

f	h	d

Strongly Connected Component

DFS Application

(a.21)



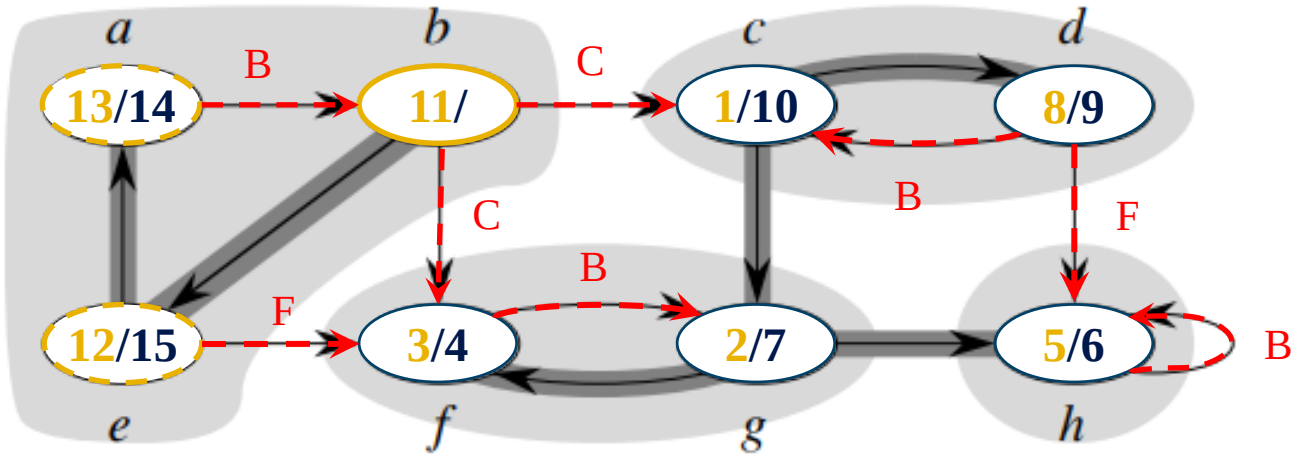
Stack

f	h	d	a

Strongly Connected Component

DFS Application

(a.22)



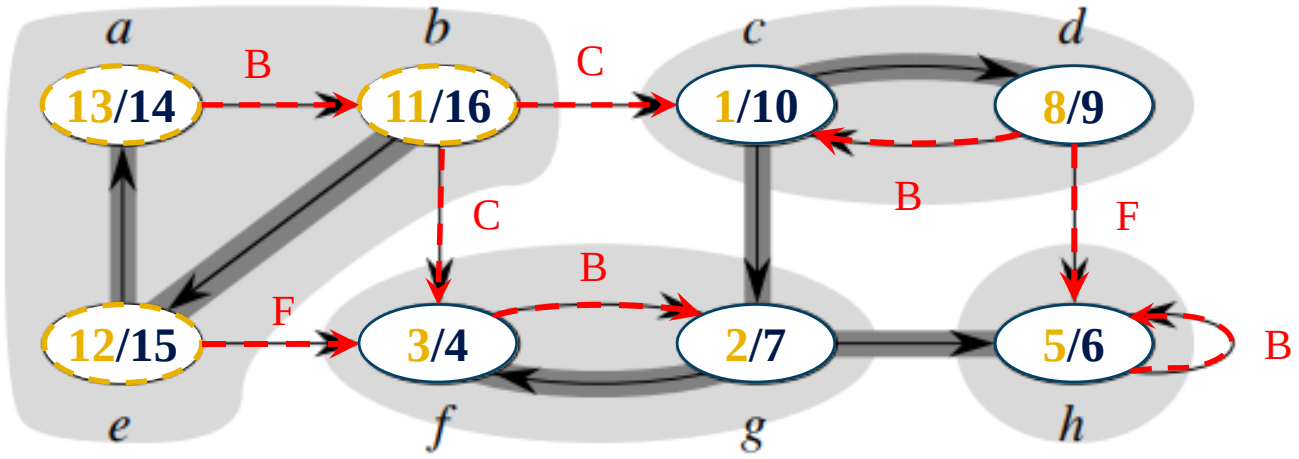
Stack

f	h	d	a

Strongly Connected Component

DFS Application

(a.23)



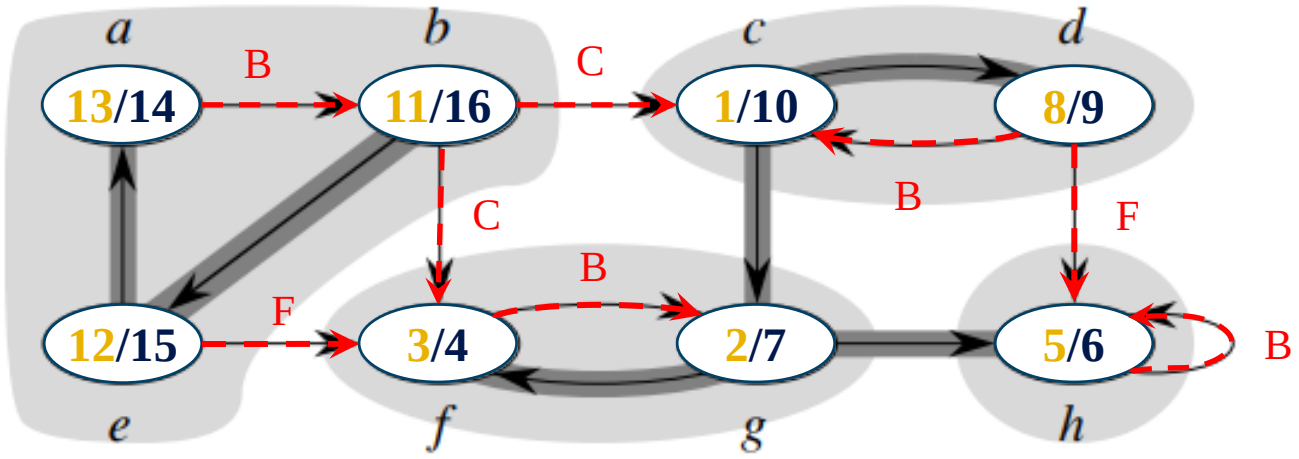
Stack

f	h	d	a

Strongly Connected Component

DFS Application

(a.24)



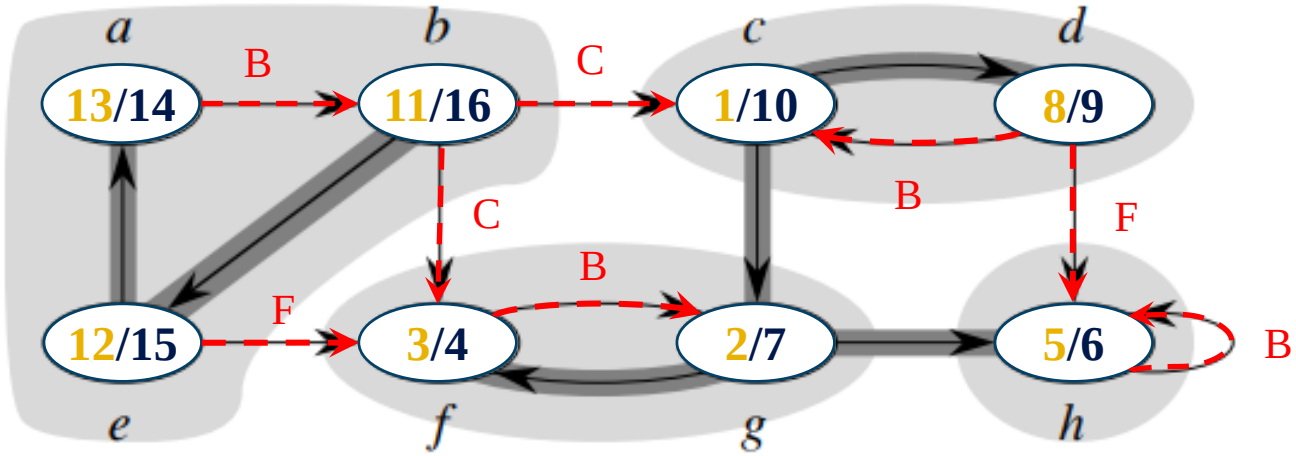
Stack

f	h	d	a

Strongly Connected Component

DFS Application

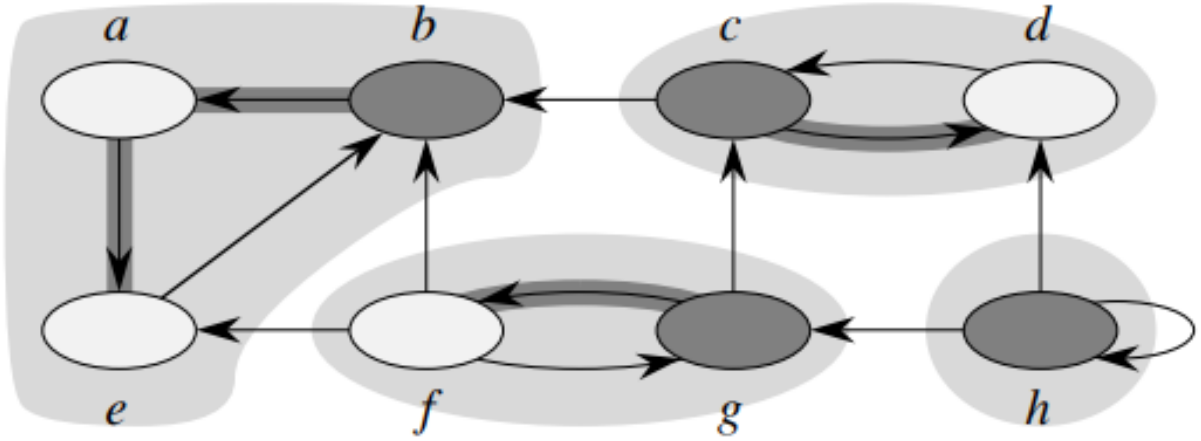
(a)



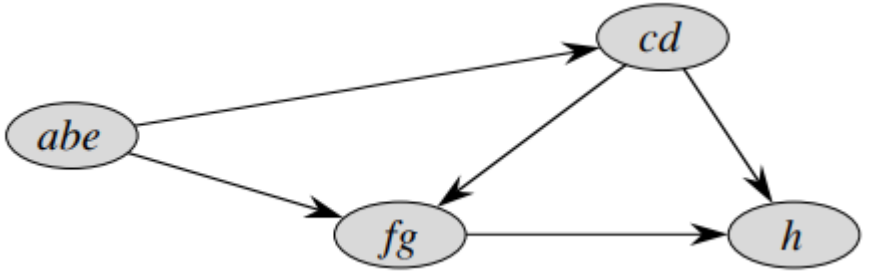
Stack

f	h	d	a

(b)

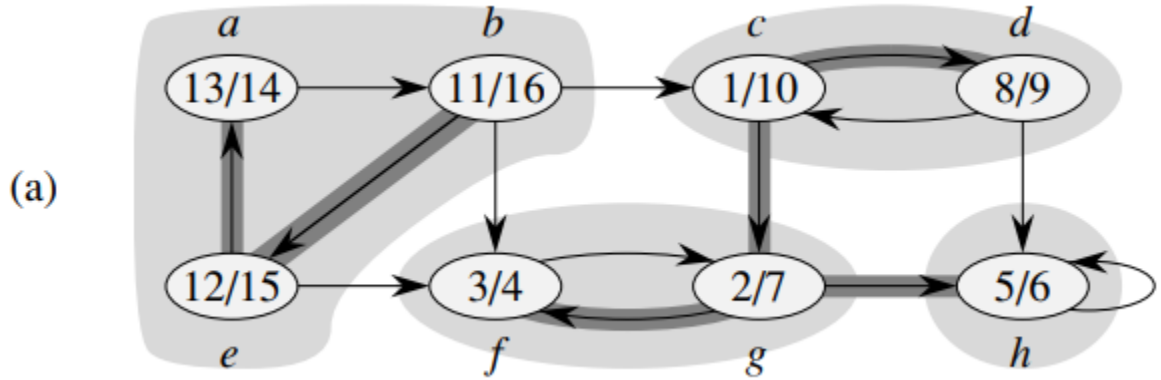


(c)



Strongly Connected Component

DFS Application



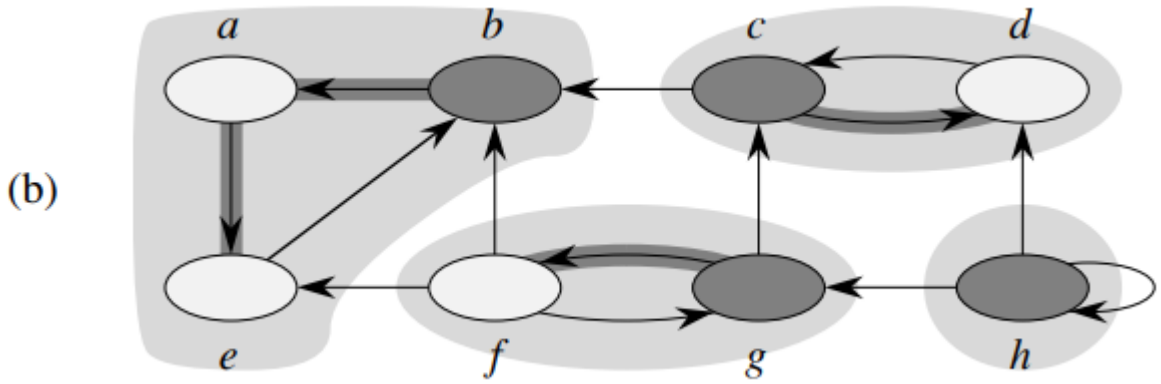
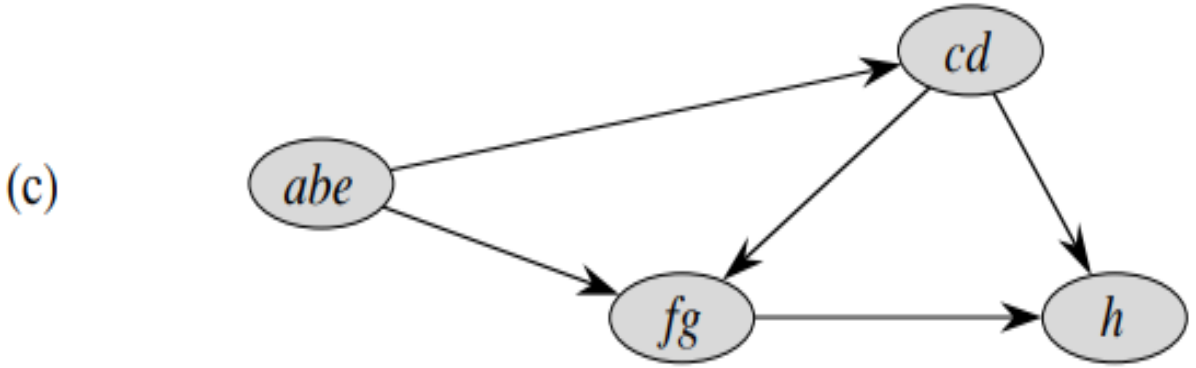


Figure 22.9 **(a)** A directed graph G. The strongly connected components of G are shown as shaded regions. Each vertex is labeled with its discovery and finishing times. Tree edges are shaded. **(b)** The graph G^T, the transpose of G. The depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS is shown, with tree edges shaded. Each strongly connected component corresponds to one depth-first tree. Vertices b, c, g, and h, which are heavily shaded, are the roots of the depth-first trees produced by the depth-first search of G^T. **(c)** The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component. [1]



Strongly Connected Component

DFS Application

Chapter 22 Elementary Graph Algorithms [1]

STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $f[u]$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

- DFS can decompose a directed graph into its strongly connected component with the linear time Θ (V+E). [1]

References

Texts | Integrated Development Environment (IDE)

[1] Introduction to Algorithms, Second Edition, Thomas H. C., Charles E. L., Ronald L. R., Clifford S., The MIT Press, McGraw-Hill Book Company, Second Edition 2001.

[2] Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Willy & Sons Inc., 2013.

[3]https://runestone.academy/runestone/books/published/pythonds/Graphs/StronglyConnectedComponents.html

[4] https://www.cs.usfca.edu/~galles/visualization/