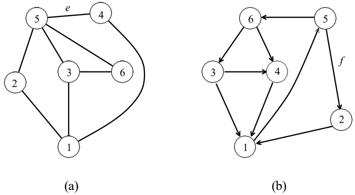
22. Graph Algorithms

Graphs

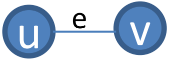


(a) An undirected graph (b) a directed graph.

- An abstract way of representing connectivity using nodes (also called vertices) and edges
- edges connect some pairs of nodes
  - Edges can be either directed or undirected
- Nodes and edges can have some auxiliary information

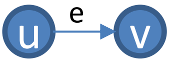
Definitions (Appendix B.4)

- An *undirected graph* G is a pair (V, E), where V is a finite set of points called *vertices* and E is a finite set of *edges*.
- An edge $e \in E$ is an unordered pair (u, v), where $u, v \in V$.



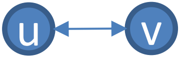
$$V = \{u, v\}, E = \{(u, v)\}$$

- In a *directed graph*, the edge e is an ordered pair (u, v). An edge (u, v) is *incident from* vertex u and is *incident to* vertex v.



$$V = \{u, v\}, E = \{(u, v)\}$$

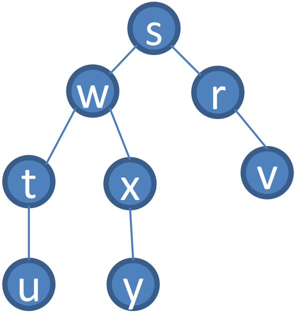
- An undirected graph can be thought of as a directed graph.



$$V = \{u, v\}, E = \{(u, v), (v, u)\}$$

- A *path* from a vertex v to a vertex u is a sequence $(v_0, v_1, v_2, \ldots, v_k)$ of vertices where $v_0 = v$, $v_k = u$, and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k - 1$.
- A vertex u' is *reachable* from a vertex u if there is a path p from u to u' in G. $u \overset{p}{\leadsto} u'$
- The *length of a path* is defined as the number of edges in the path.
- A *cycle* is a path where $v_0 = v_k$
- An undirected graph is *connected* if every pair of vertices is connected by a path.
- A *forest* is an acyclic (cycle 이 없는) graph, and a *tree* is a connected acyclic graph.
- A graph that has weights associated with each edge is called a *weighted graph*.

Tree



- A connected acyclic graph
- Most important type of special graphs
  - Many problems are easier to solve on trees
- Alternate equivalent definitions:
  - A connected graph with $n-1$ edges (where n is a number of vertices)
  - An acyclic graph with $n-1$ edges
  - There is exactly one path between every pair of nodes
  - An acyclic graph but adding any edge results in a cycle
  - A connected graph but removing any edge disconnects it

Representation of a Graph

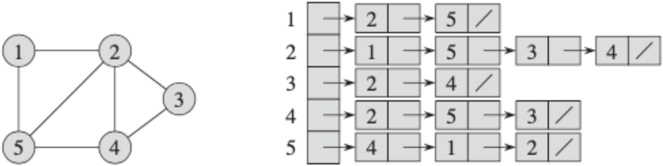
- Graphs can be represented by their adjacency matrix or adjacency list.

- Adjacency matrices have a value $a_{i,j} = 1$ if nodes i and j share an edge; 0 otherwise. In case of a weighted graph, $a_{i,j} = w_{i,j}$, the weight of the edge.

- The *adjacency list* representation of a graph $G = (V, E)$ consists of an array $Adj[1..|V|]$ of lists. Each list $Adj[v]$ is a list of all vertices adjacent to v.

- For a graph with n nodes, adjacency matrices take $\Theta(n^2)$ space and adjacency list takes $\Theta(|E|+|V|)$ space.

Undirected Graph



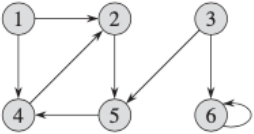
(a) (b)

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

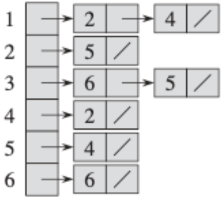
(c)

교과서에는 대부분 adjacency list 표현을 가정할 것이다.

Directed Graph



(a)



(b)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)

Graph Traversal

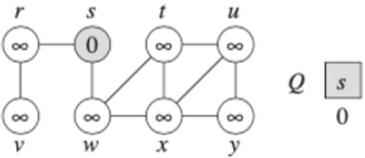
- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- 교과서 예에서 특별한 언급이 없으면 vertex 는 알파벳 순으로 처리한다.
- We will cover two algorithms
  - Breadth-First Search (BFS): uses queue
  - Depth-First Search (DFS): uses recursion (stack)

22.2 Breadth-First Search (0)

BFS(G, s) initialization

```
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞          not discovered
    u.π = NIL
s.color = GRAY       discovered
s.d = 0
s.π = NIL            starting point
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

(a)



depth in the BFS tree

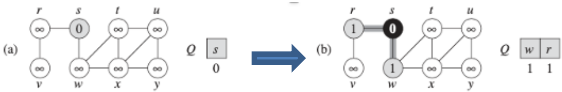
22.2 Breadth-First Search (1)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞                 not discovered
    u.π = NIL
s.color = GRAY              discovered
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Finished

predecessor

depth in the BFS tree



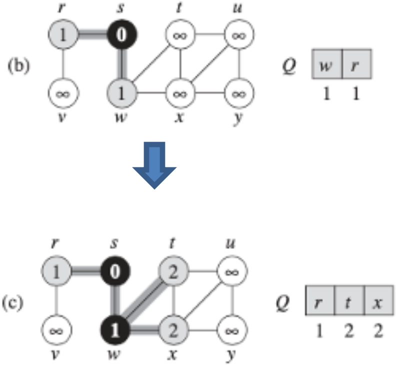
22.2 Breadth-First Search (2)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞          not discovered
    u.π = NIL
s.color = GRAY       discovered
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Finished

predecessor

depth in the BFS tree



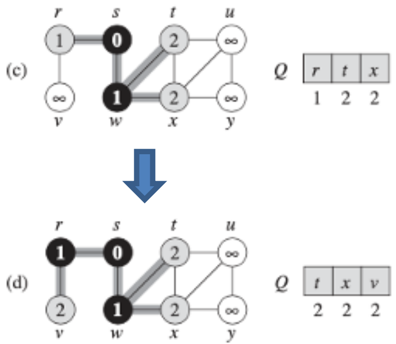
22.2 Breadth-First Search (3)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞            not discovered
    u.π = NIL
s.color = GRAY         discovered
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Finished

predecessor

depth in the BFS tree



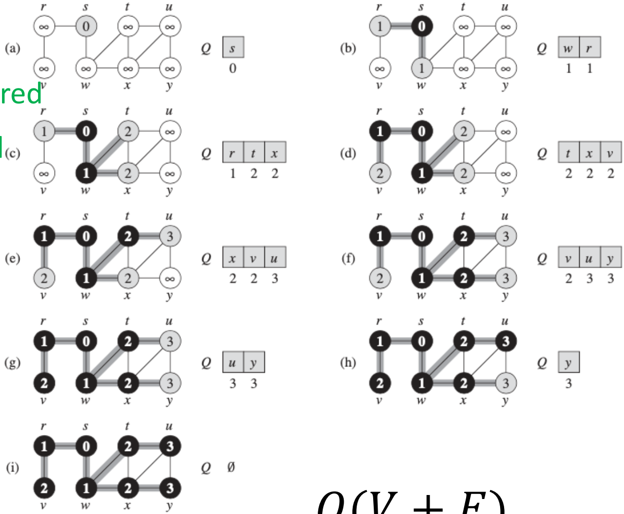
22.2 Breadth-First Search (8)

```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞          not discovered
    u.π = NIL
s.color = GRAY       discovered
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Finished

predecessor

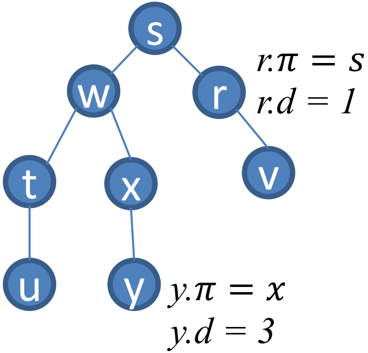
depth in the BFS tree



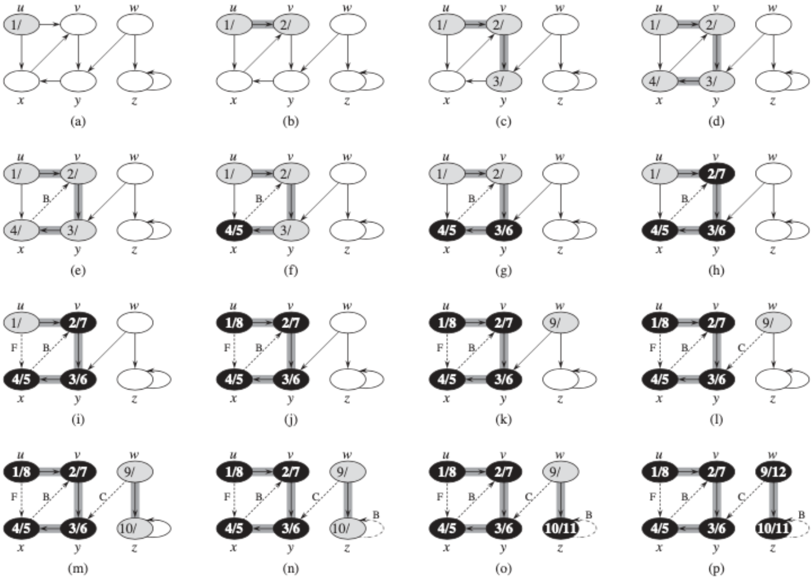
$$O(V + E)$$

Breadth-first tree (BFS tree)

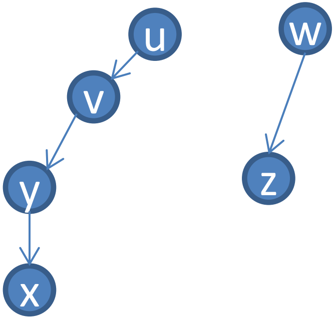
BFS 는 vertex s 로부터 reachable vertex v 에 대한 shortest path distance $\delta(s, v)$ 를 모두 계산한다. 그 값은 $v.d$ 이다.



22.3 Depth-First Search for a directed graph



Depth-first forest (DFS forest)



DFS algorithm

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

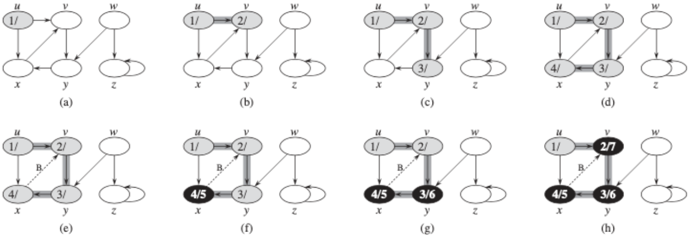
not discovered

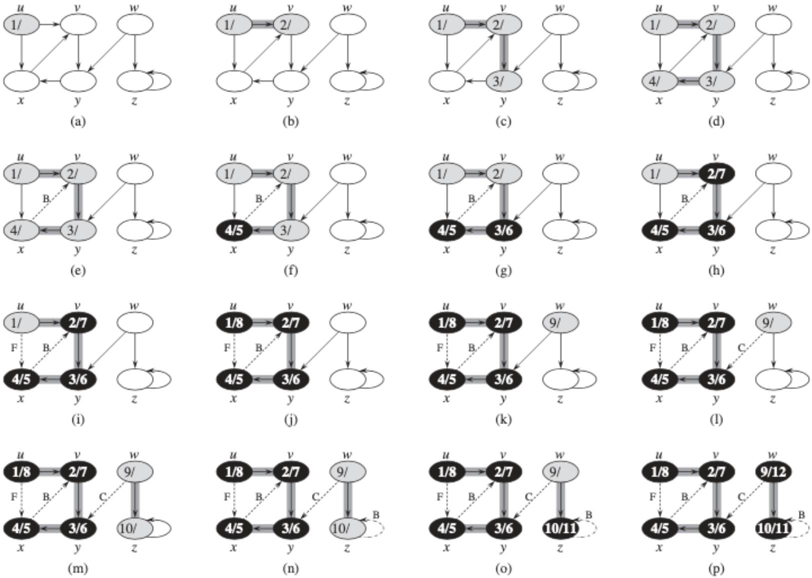
$$O(V+E)$$

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

discovered

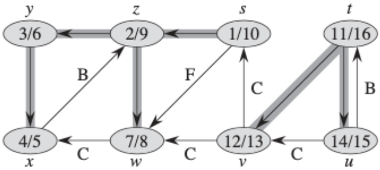
finished(=visited)

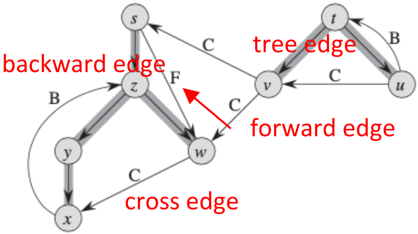


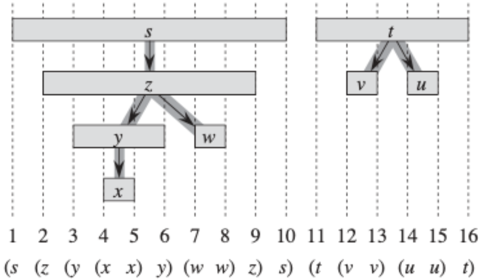

(a)
(b)
(c)
(d)
(e)
(f)
(g)
(h)
(i)
(j)
(k)
(l)
(m)
(n)
(o)
(p)

Properties of DFS

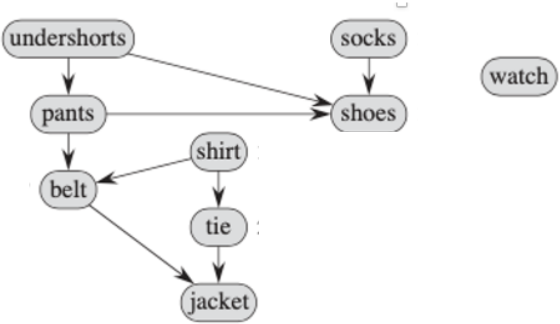
다음과 같은 timestamped directed graph 가 만들어진다.







22.4 Topological Sort

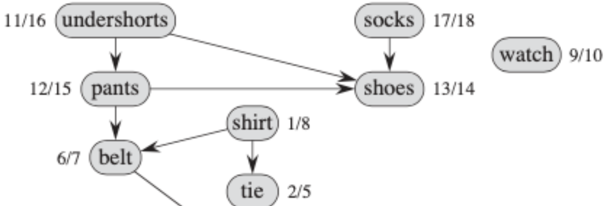


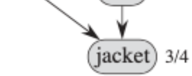
A ***topological sort*** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v) then u appears before v in the ordering.

A ***topological sort*** of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v) then u appears before v in the ordering.

dag : directed acyclic graph

(a)

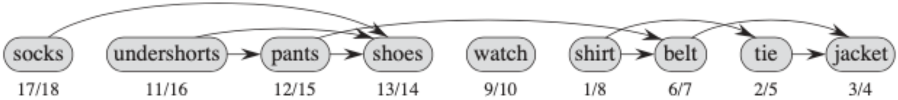




TOPOLOGICAL-SORT(G)

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

(b)

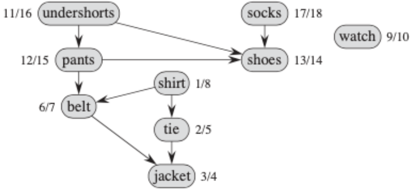


Theorem 22.12

TOPOLOGICAL-SORT algorithm produces a topological sort of the directed acyclic graph provided as its input.

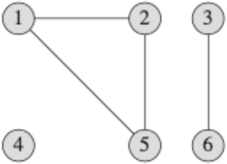
```
TOPOLOGICAL-SORT(G)
call DFS(G) to compute finishing times v.f for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

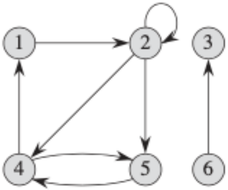
Proof Suppose that DFS is run on a given dag G = {V,E} to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices u,v in V , if G contains an edge from u to v, then v.*f* < u.*f*.



22.5 Strongly connected components

- An undirected graph is ***connected*** if every vertex is reachable from all other vertices. The ***connected components*** of a graph are the equivalence classes of vertices under the “is reachable from” relation.
{1,2,5},{3,6},{4}





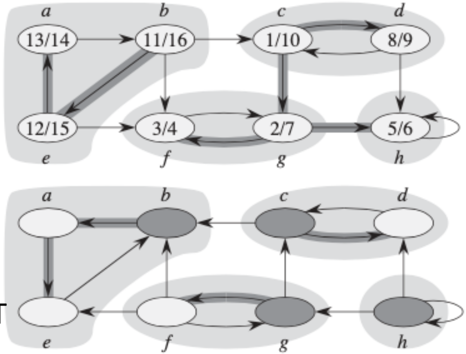
- A directed graph is ***strongly connected*** if every two vertices are reachable from each other. The ***strongly connected components*** of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation.
{1,2,4,5}, {3}, {6}

STRONGLY-CONNECTED-COMPONENTS(G)

```
1  call DFS(G) to compute finishing times u.f 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 u.f (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
```

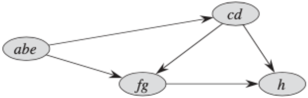
G

(a)



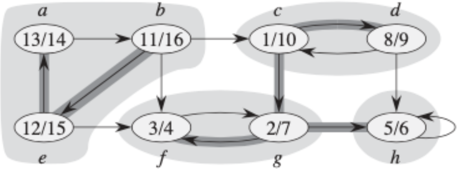
(b)

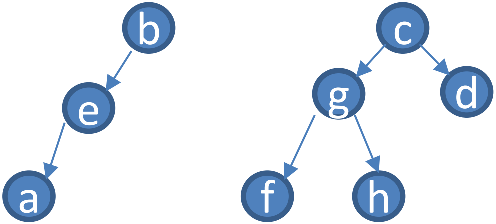
G^T



1. DFS(G)

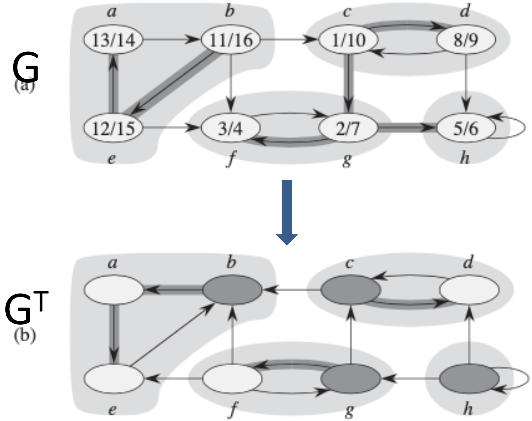
(a)





2. compute G^T

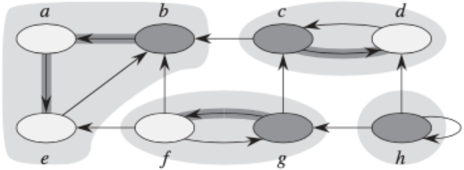
All edge directions are reversed.



3. DFS(G^T)

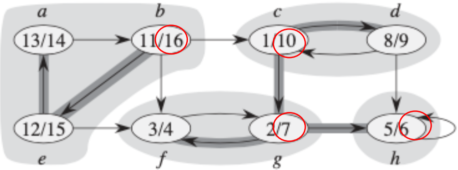
Vertices are selected in order of decreasing u.f

G^T



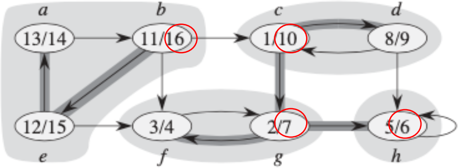
Practice) compute u.s/u.f in G^T

(a)



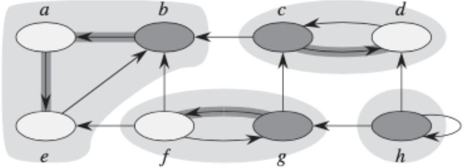
4. SCC

G

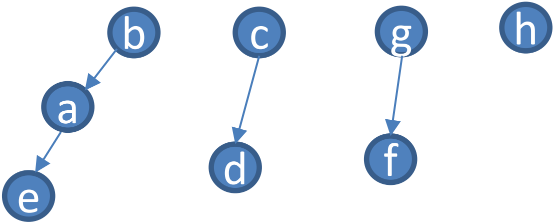


a
b
c
d
13/14
11/16
1/10
8/9
12/15
3/4
2/7
5/6
e
f
g
h

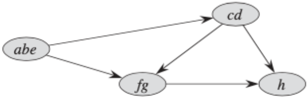
(b)



a
b
c
d
e
f
g
h



b
c
g
h
a
d
f
e



cd
abe
fg
h

Euler tour

An ***Euler tour*** of a strongly connected, directed graph $G = (V,E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

- G has an Euler tour iff in-degree(v) = out-degree(v) for all $v \in V$
- The Euler tour algorithm runs in $O(E)$.