10. Graph Algorithms

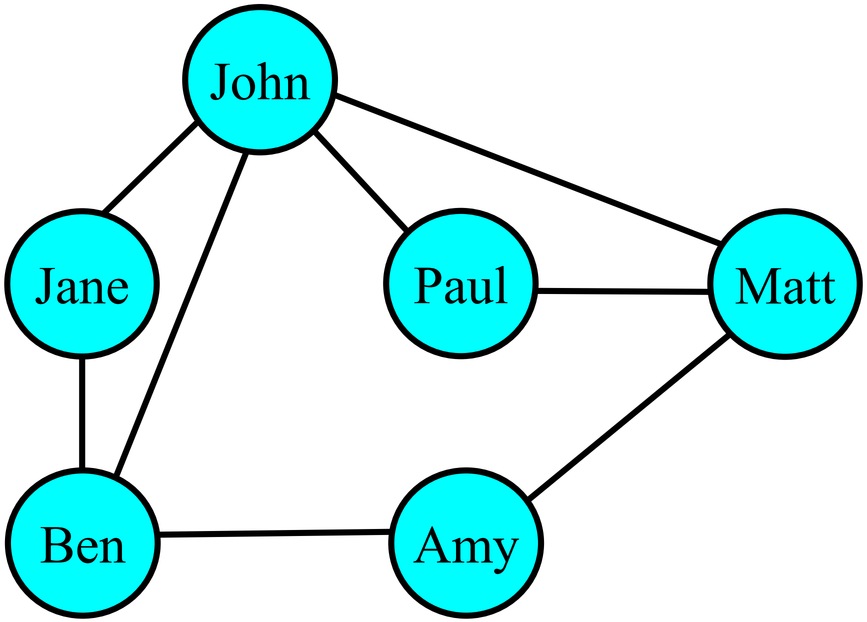
Goals

- Learn representations of graphs
- Understand depth-first search and breadth-first search
- Understand topological sort of directed acyclic graph
- Understand shortest-paths problems and algorithms corresponding to problems
- Understand strongly connected components

Graph

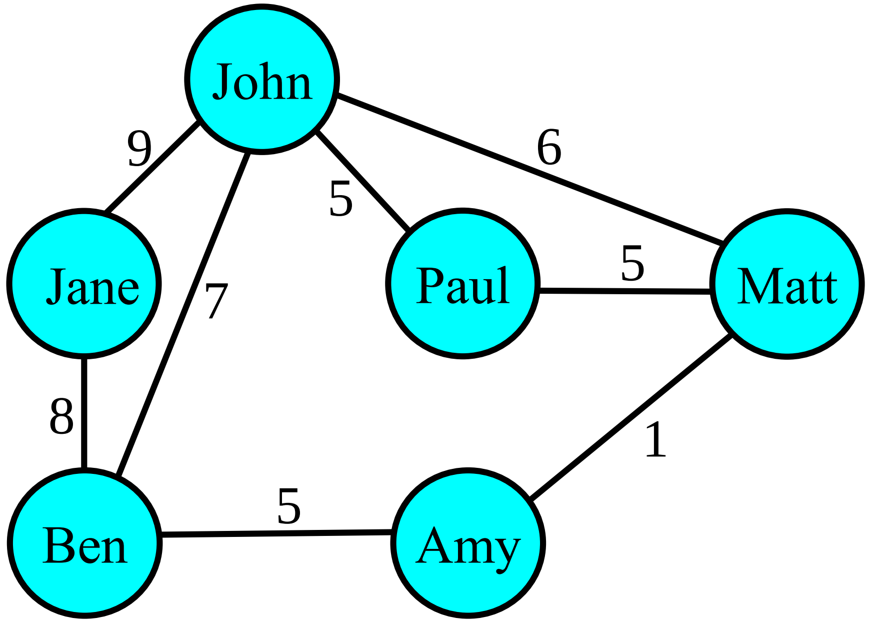
- Model objects and relationships by vertices and edges
- Graph $G = (V, E)$
  - V: set of vertices
  - E: set of edges
- Two vertices u and v are *adjacent* if there is an edge (u,v).

Graph



modeling relationships between people

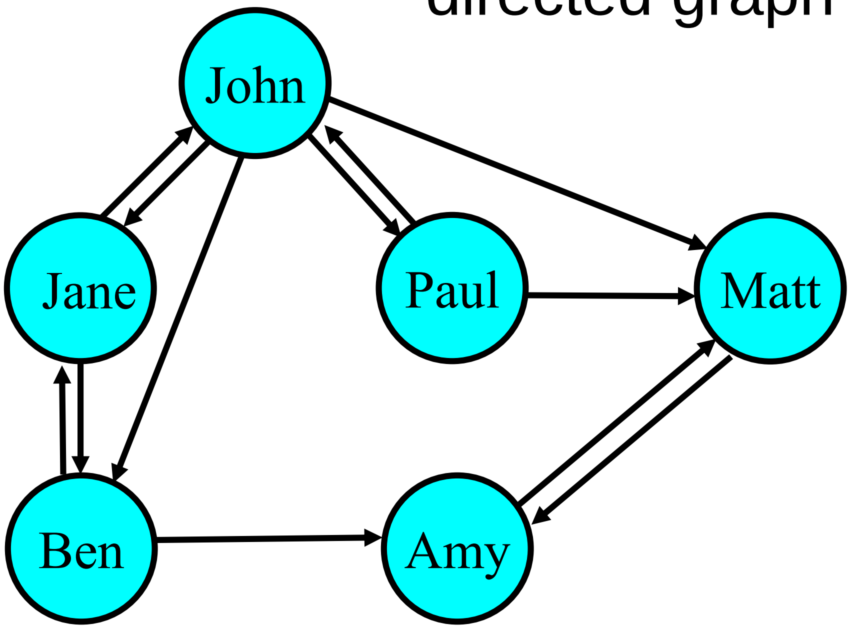
Graph



giving weights to relationships

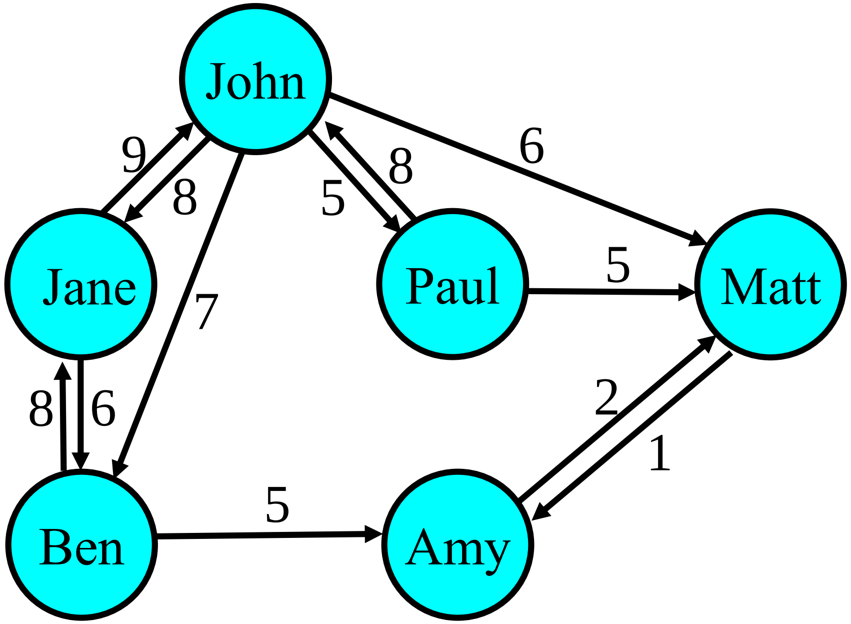
Graph

directed graph = digraph



modeling relationships with directions

Graph



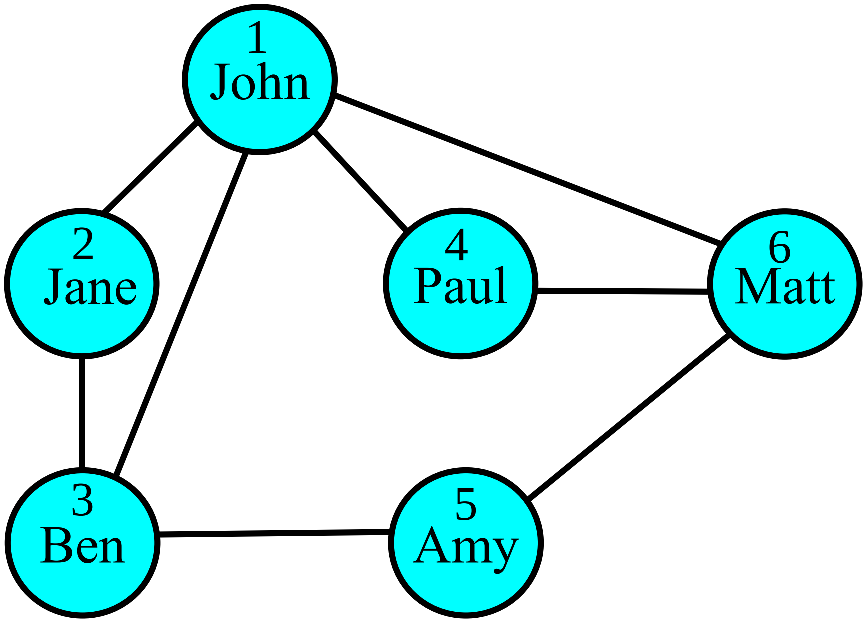
directed, weighted graph

Representation of Graph 1

N: number of vertices

- Adjacency matrix
  - $N \times N$ matrix A
    - A(i, j) = 1 : there is an edge between vertex i and vertex j
    - A(i, j) = 0 : there is no edge between vertex i and vertex j
  - Directed graph
    - A(i, j) represents an edge from vertex i to vertex j (1 or 0)
  - Graph with weights
    - A(i, j) is the weight

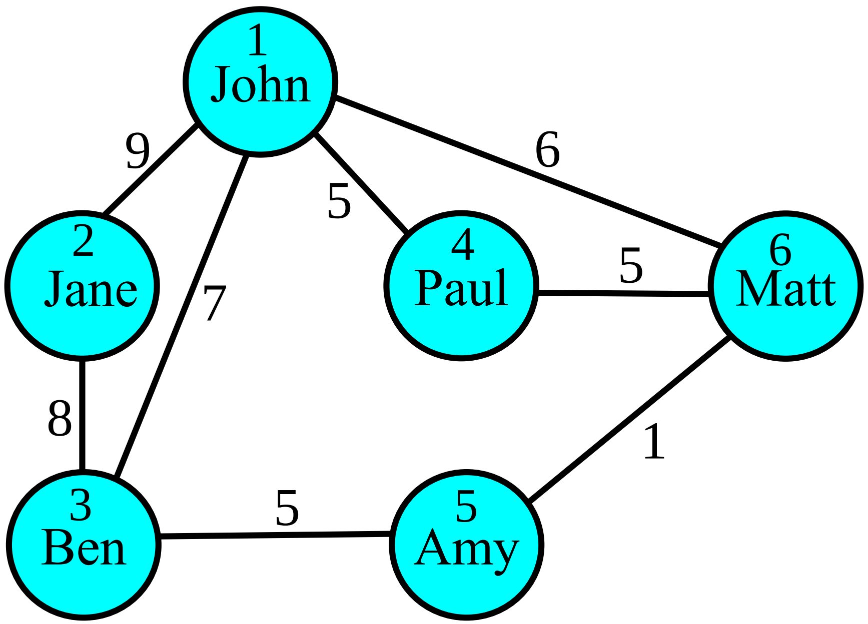
Adjacency Matrix



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

undirected graph

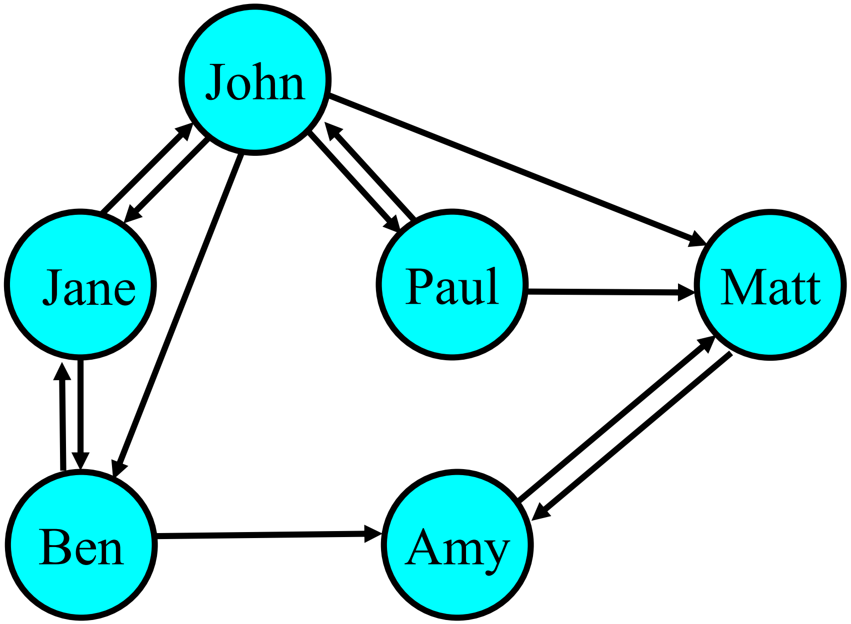
Adjacency Matrix



	1	2	3	4	5	6
1	0	9	7	5	0	6
2	9	0	8	0	0	0
3	7	8	0	0	5	0
4	5	0	0	0	0	5
5	0	0	5	0	0	1
6	6	0	0	5	1	0

undirected, weighted graph

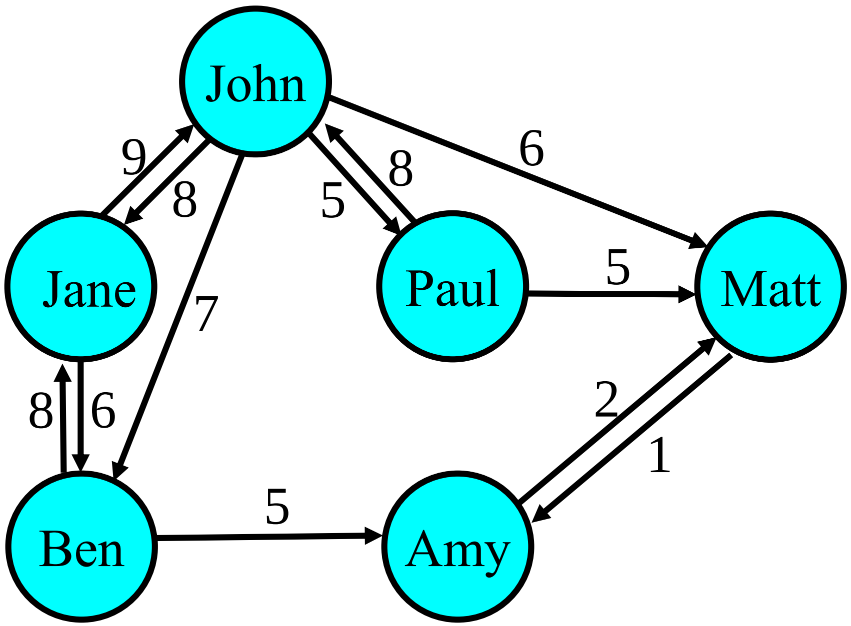
Adjacency Matrix



directed graph

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

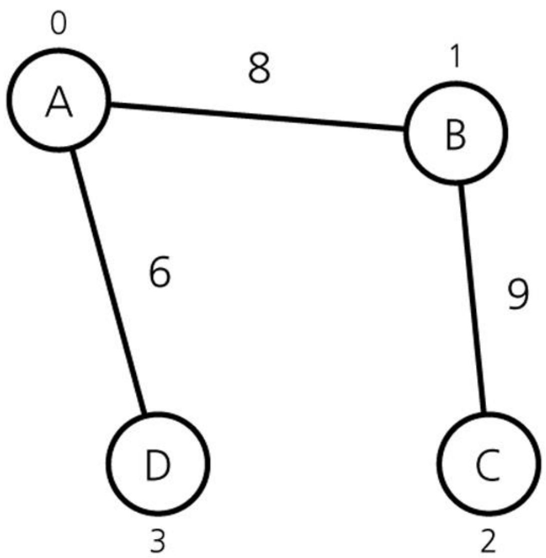
Adjacency Matrix



	1	2	3	4	5	6
1	0	8	7	5	0	6
2	9	0	6	0	0	0
3	0	8	0	0	5	0
4	8	0	0	0	0	5
5	0	0	0	0	0	2
6	0	0	0	0	1	0

directed, weighted graph

Adjacency Matrix



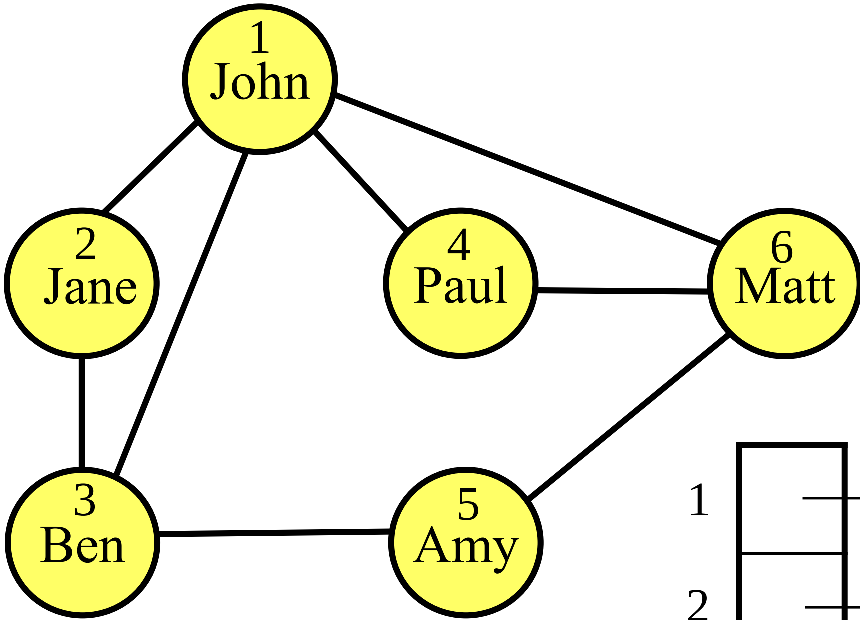
		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

undirected, weighted graph

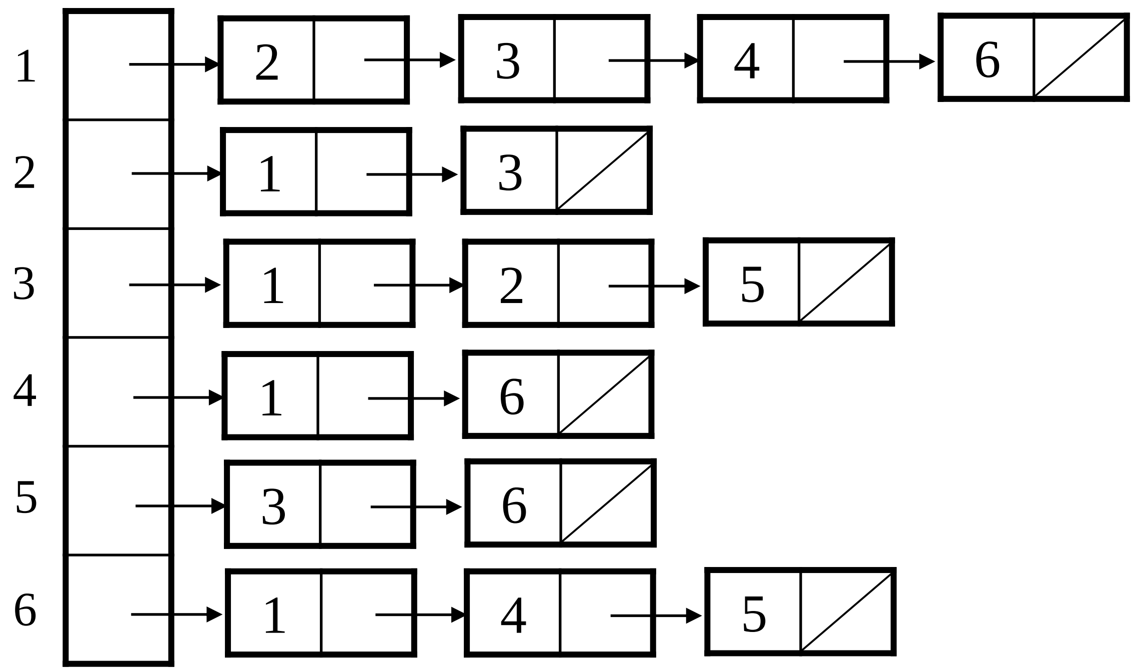
Representation of Graph 2

- Adjacency list
  - Use N adjacency lists
  - The i-th list contains the vertices adjacent to vertex i
  - Weighted graph
    - weights are stored in the lists

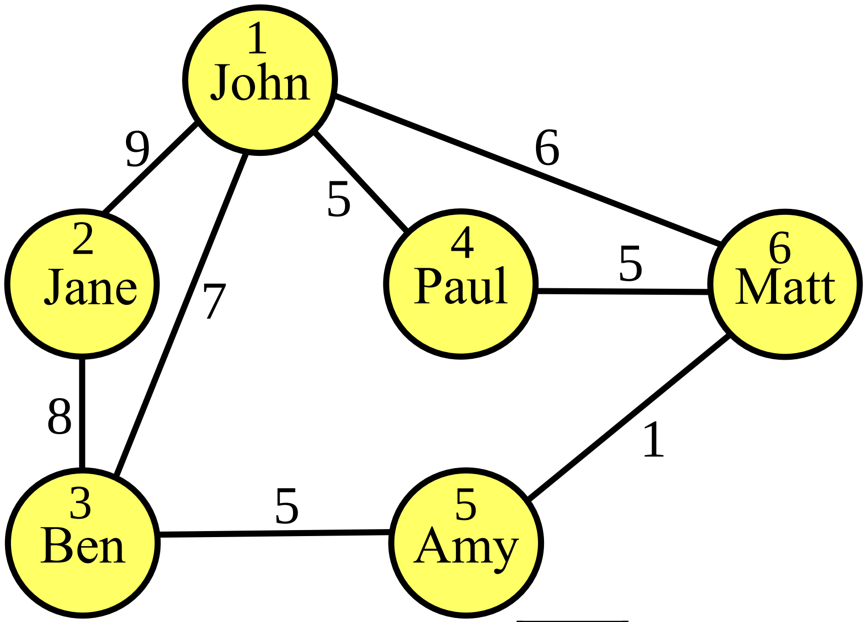
Adjacency List



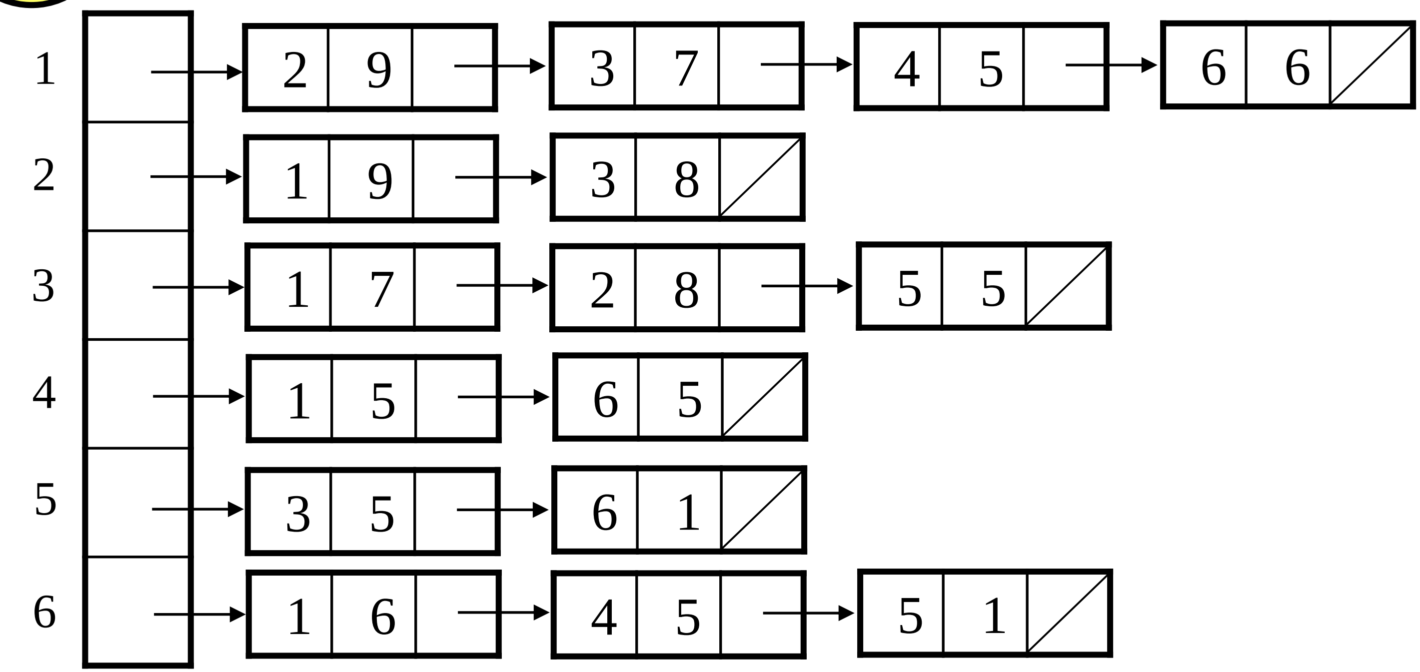
undirected graph



Adjacency List



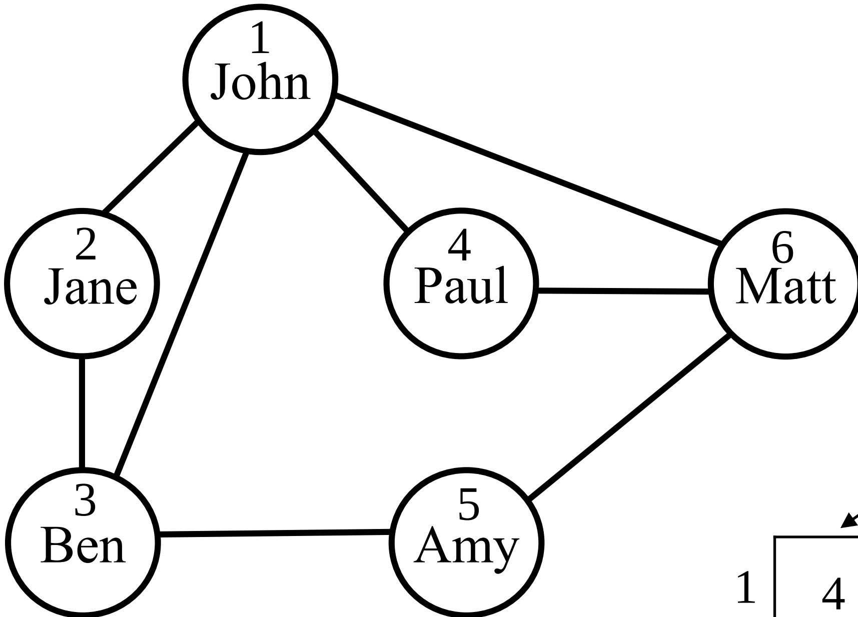
weighted graph

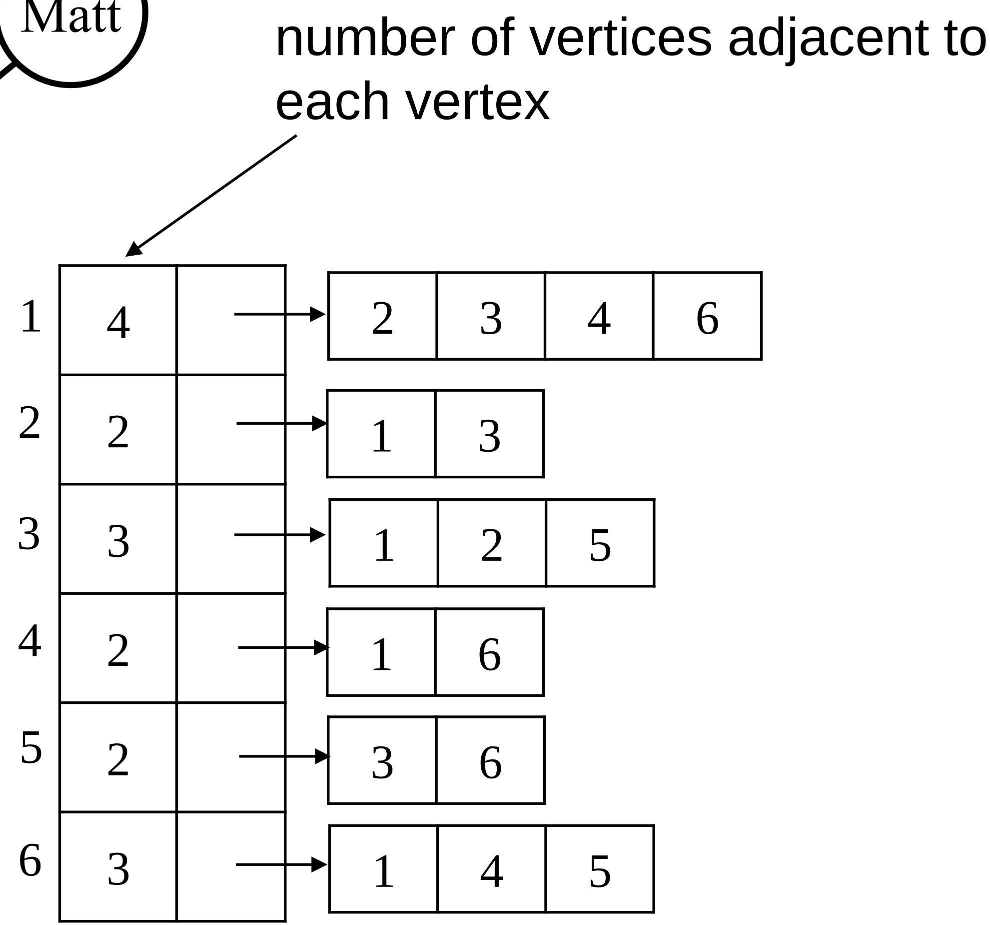


Representation of Graph 3

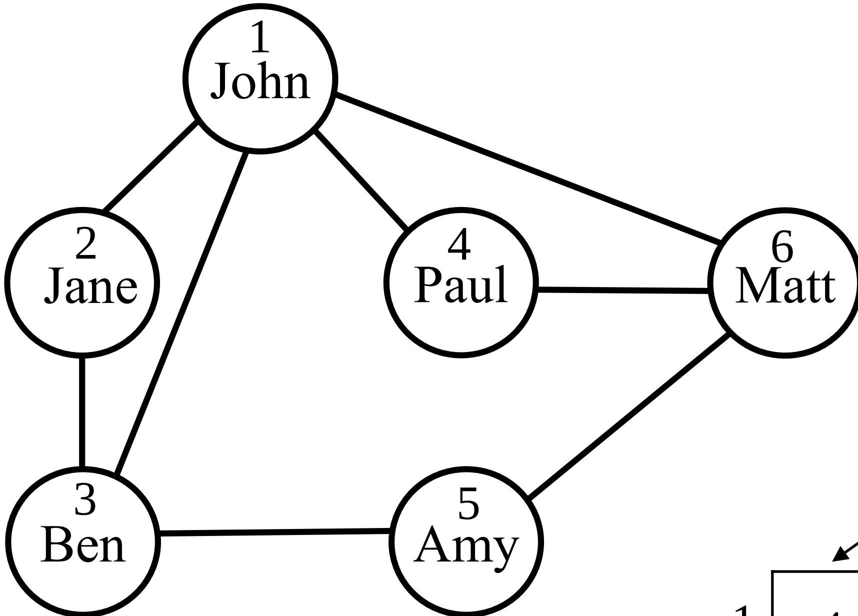
- Adjacency array
  - Use N arrays
  - The i-th array contains the vertices adjacent to vertex i
  - Weighted graph
    - weights are stored in the arrays

Adjacency Array

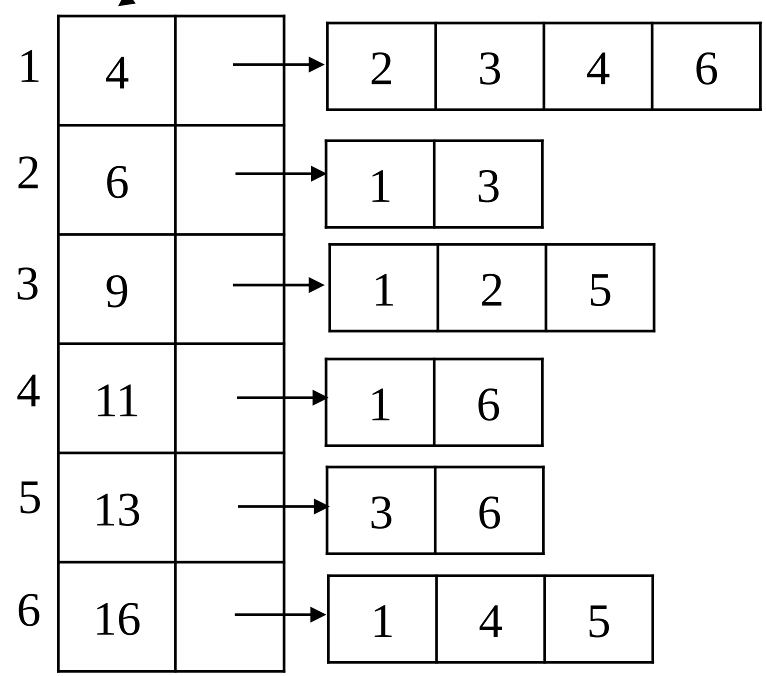




Adjacency Array



end position of vertices adjacent to each vertex in an array



Graph Search

- Two representative methods
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
- Importance of BFS and DFS
  - Basic methods for many graph algorithms
  - Deep understanding of BFS (discover time, shortest-path distance) and DFS (discover time, finish time) leads to developing good graph algorithms

BFS

```
BFS(G, v)
{
        for each v ∈ V−{s}
                visited[v] ← NO;
        visited[s] ← YES;                ▷ s: start vertex
        enqueue(Q, s);                   ▷ Q: queue
        while (Q ≠ ϕ) {
                u ← dequeue(Q);
                for each v ∈ L(u)        ▷ L(u): vertices adjacent to u
                        if (visited[v] = NO) then
                                visited[u] ← YES;
                                enqueue(Q, v);
                        }

        }
}
```

✓ Time complexity: $\Theta(|V|+|E|)$

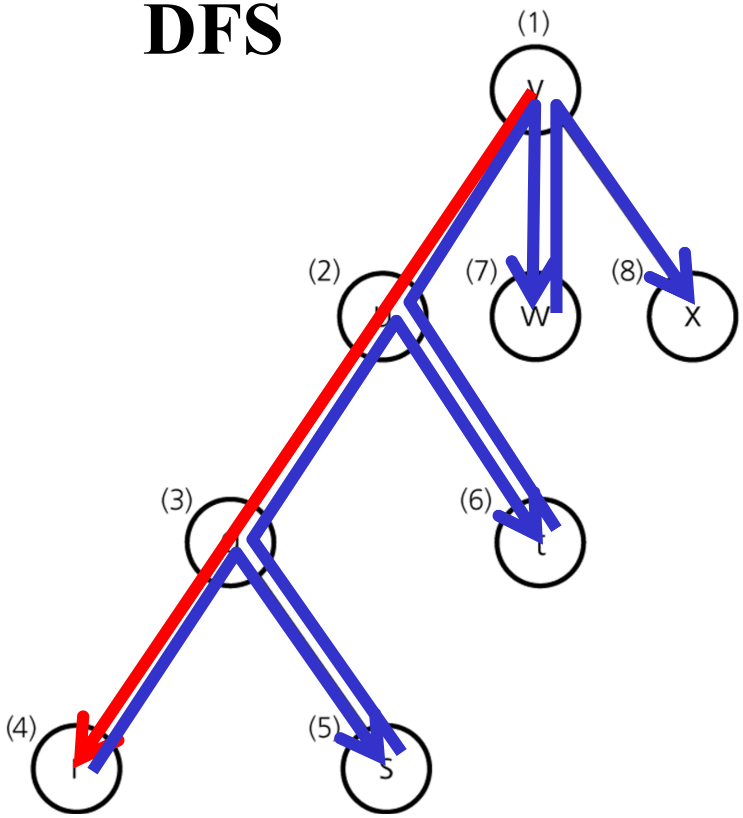
DFS

```
DFS(G)
{
        for each v ∈ V
                visited[v] ← NO;
        for each v ∈ V
                if (visited[v] = NO) then aDFS(v);
}
aDFS (v)
{
        visited[v] ← YES;
        for each x ∈ L(v)      ▷ L(v) : vertices adjacent to u
                if (visited[x] = NO) then aDFS(x);
}
```

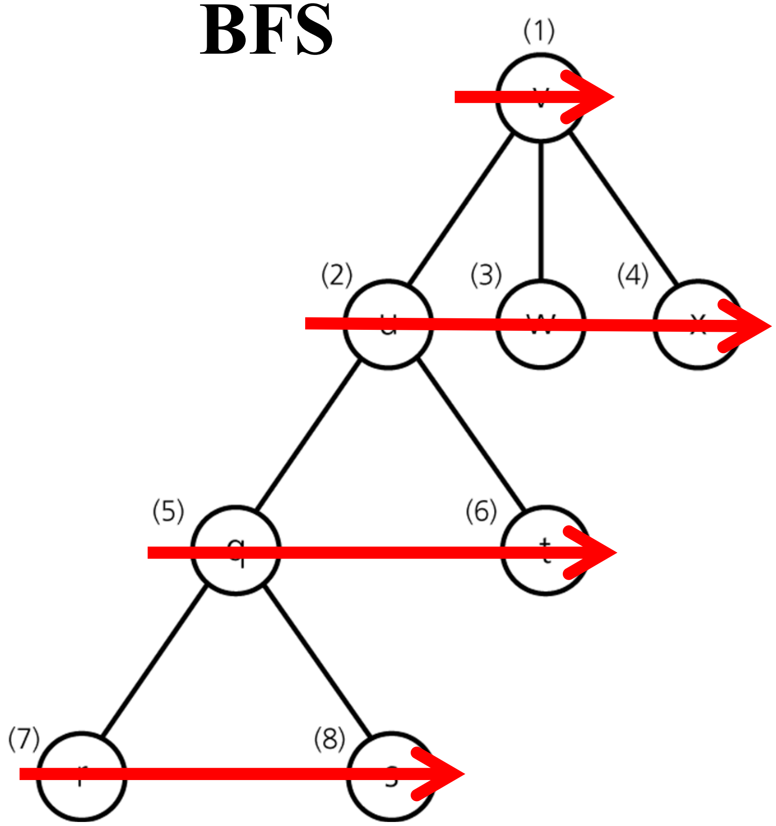
✓ Time complexity: $\Theta(|V|+|E|)$

Searching Graph with DFS/BFS

DFS



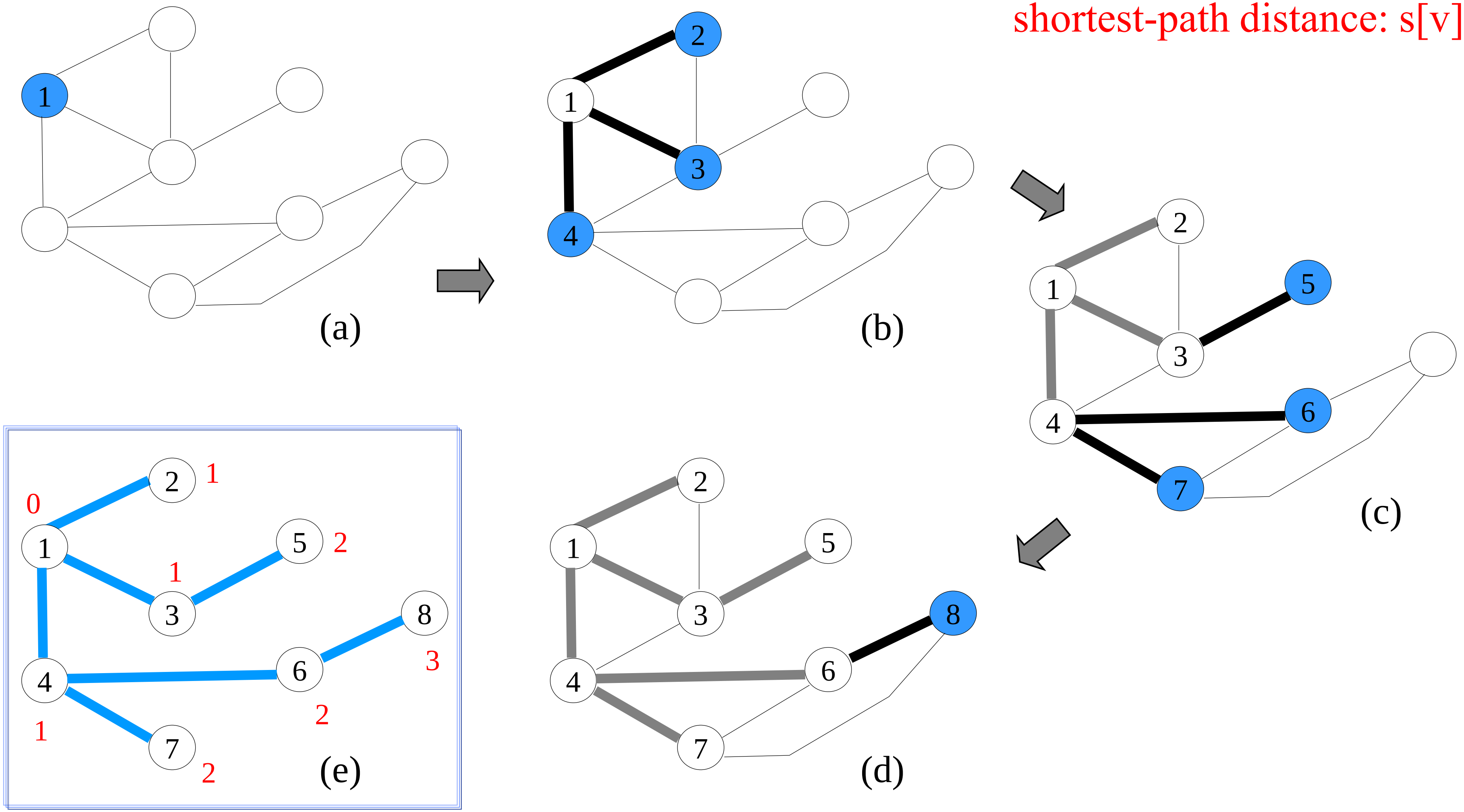
BFS



BFS

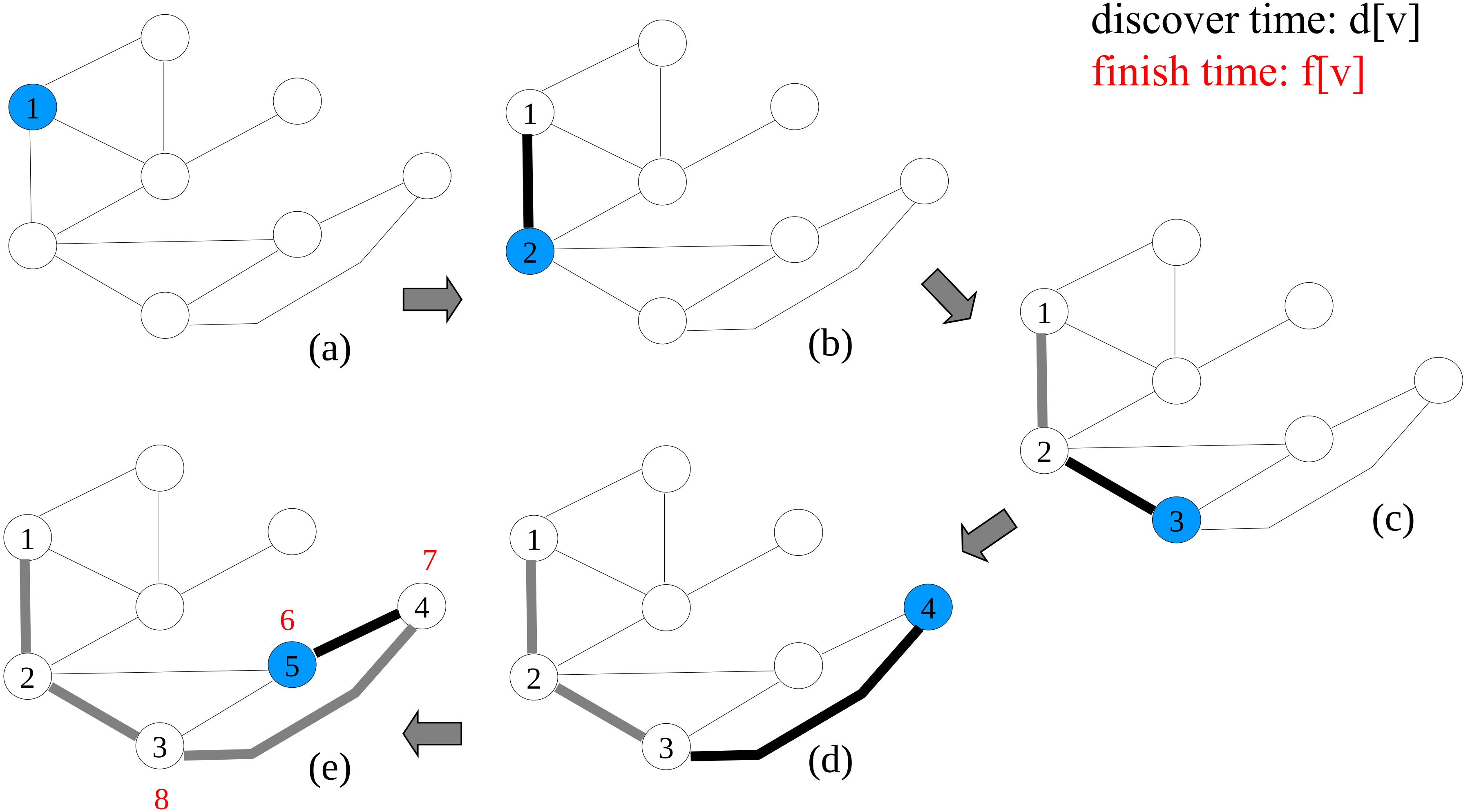
discover time: d[v]

shortest-path distance: s[v]

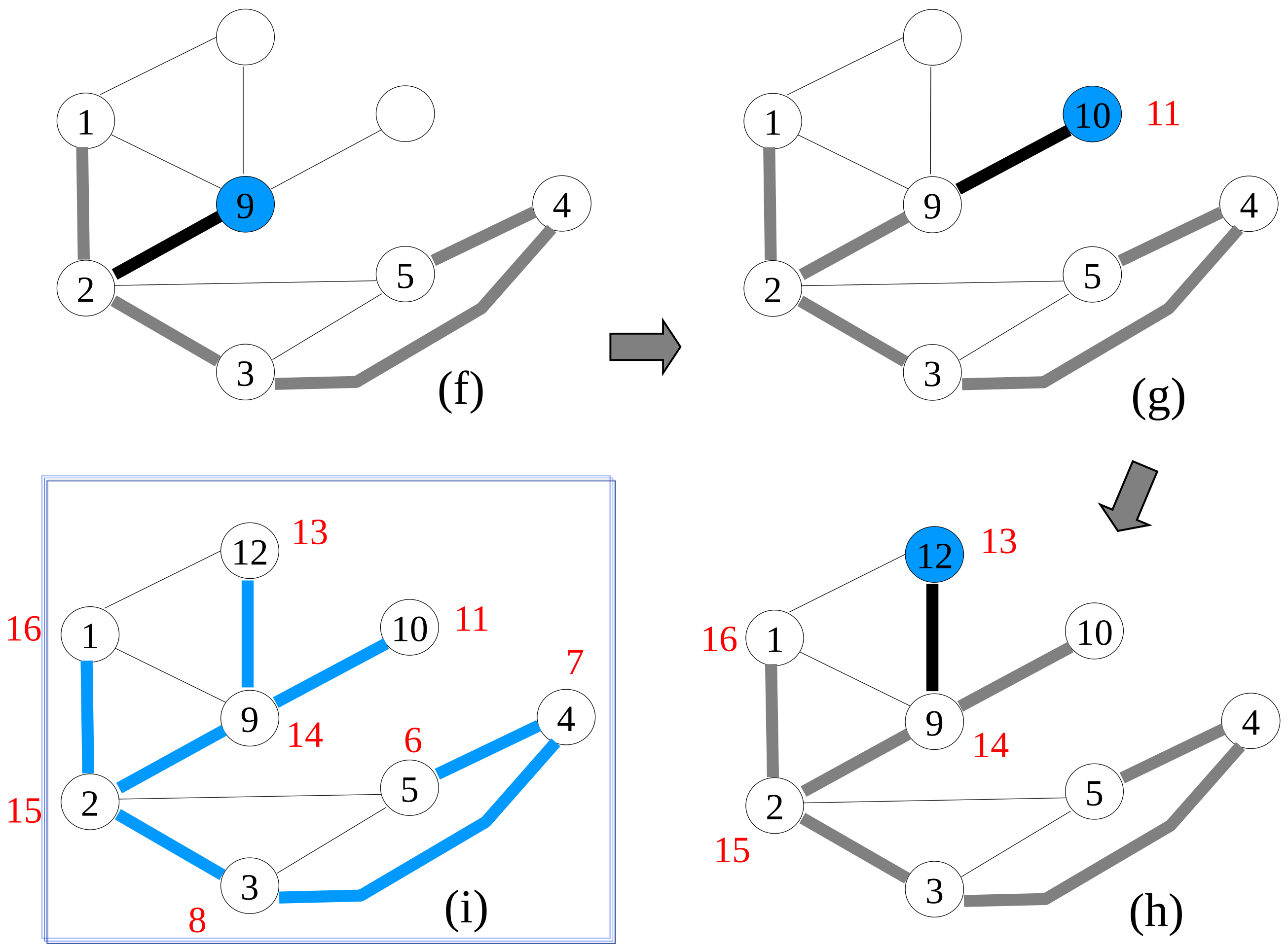


DFS

discover time: d[v]
finish time: f[v]



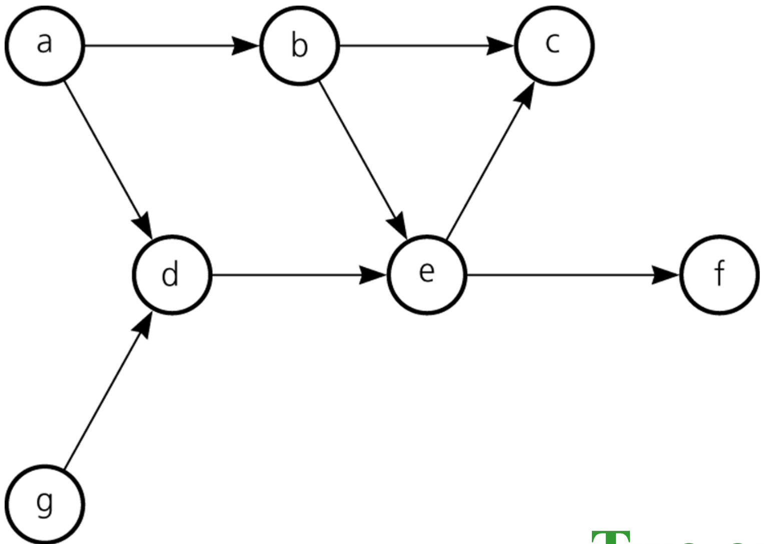
DFS



1
9
4
2
5
3
(f)
1
10
11
9
4
2
5
3
(g)
12
13
16
1
10
9
14
4
2
5
15
3
(h)
12
13
16
1
10
11
7
9
14
4
6
15
2
5
3
8
(i)

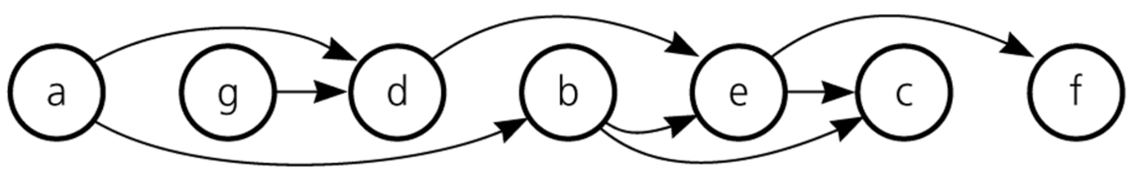
Topological Sort

- Input
  - directed acyclic graph (DAG) G
- Topological sort
  - A linear ordering of all vertices (there can be multiple orderings)
  - If G contains an edge (x, y), x appears before y in the ordering

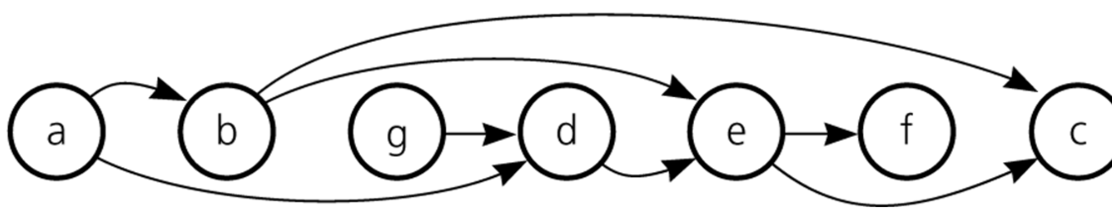


Two orderings of graph

(a)



(b)

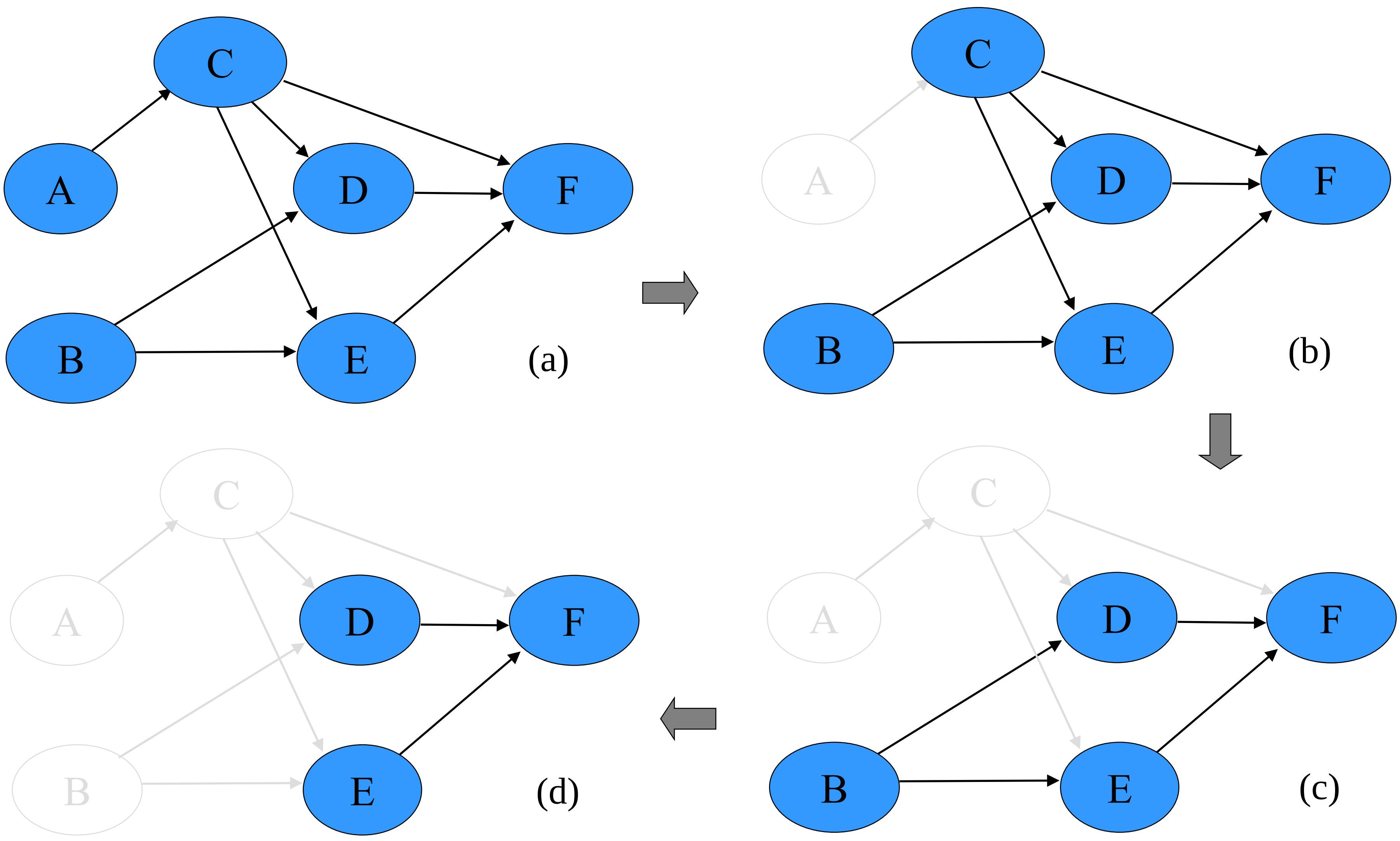


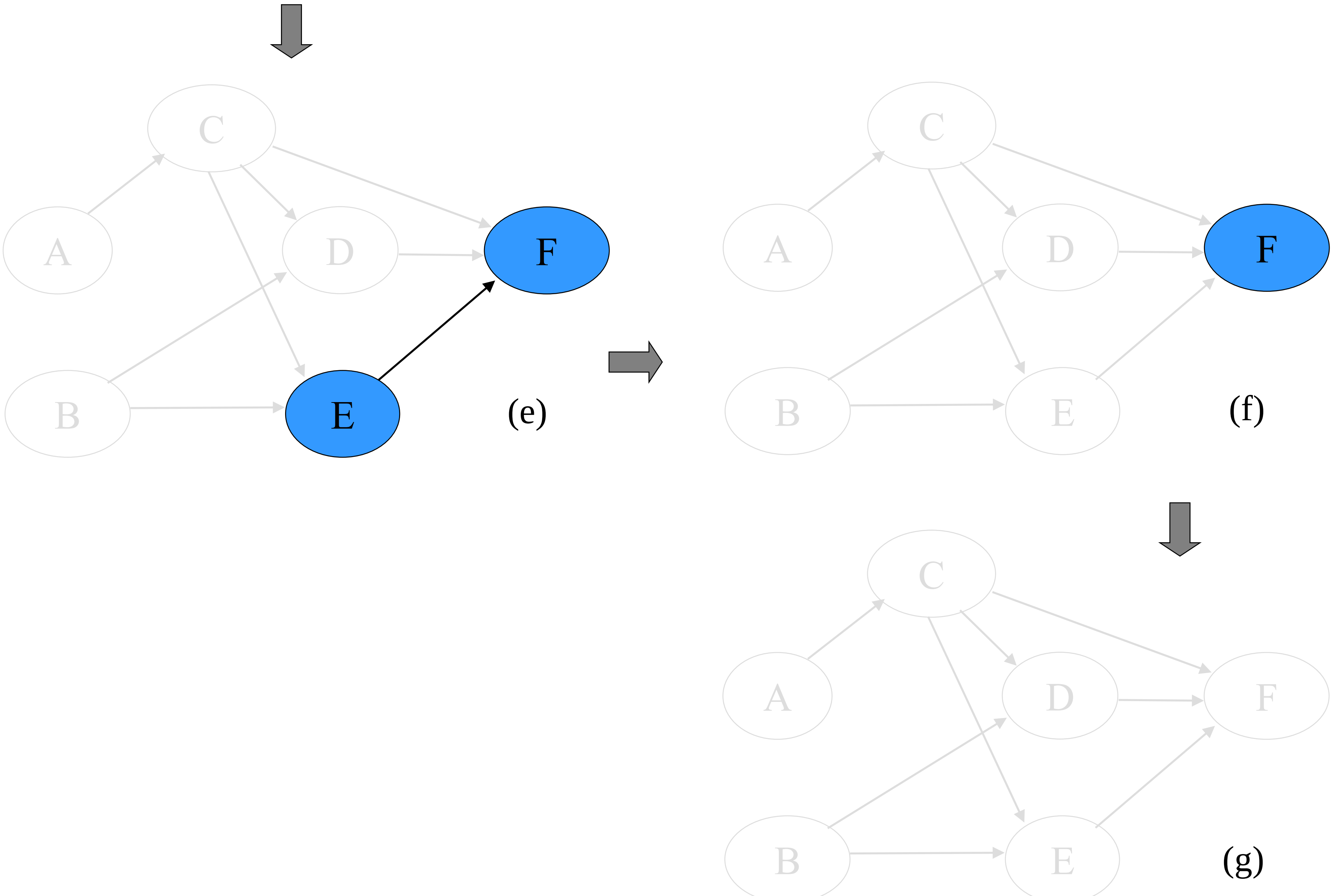
Topological Sort 1

```
topologicalSort1(G, v)
{
    for ← 1 to n {
        select a vertex u without incoming edges
        A[i] ← u;
        remove u and all outgoing edges of u
    }
    ▷ vertices in A[1…n] are topologically sorted
}
```

✓ Time complexity: $\Theta(|V|+|E|)$

Topological Sort 1




C
A
D
F
B
E
(e)
C
A
D
F
B
E
(f)
C
A
D
F
B
E
(g)

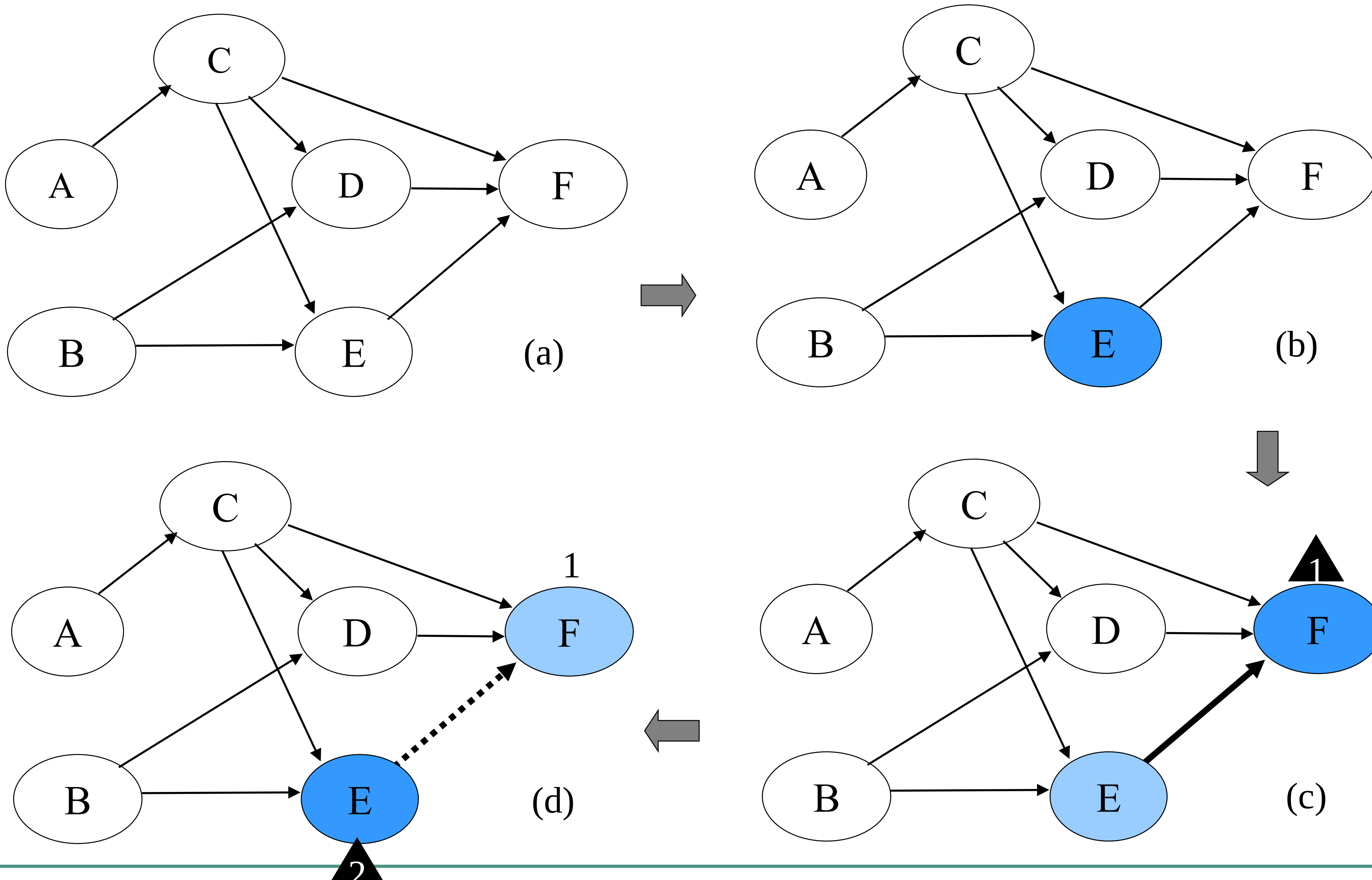
Topological Sort 2

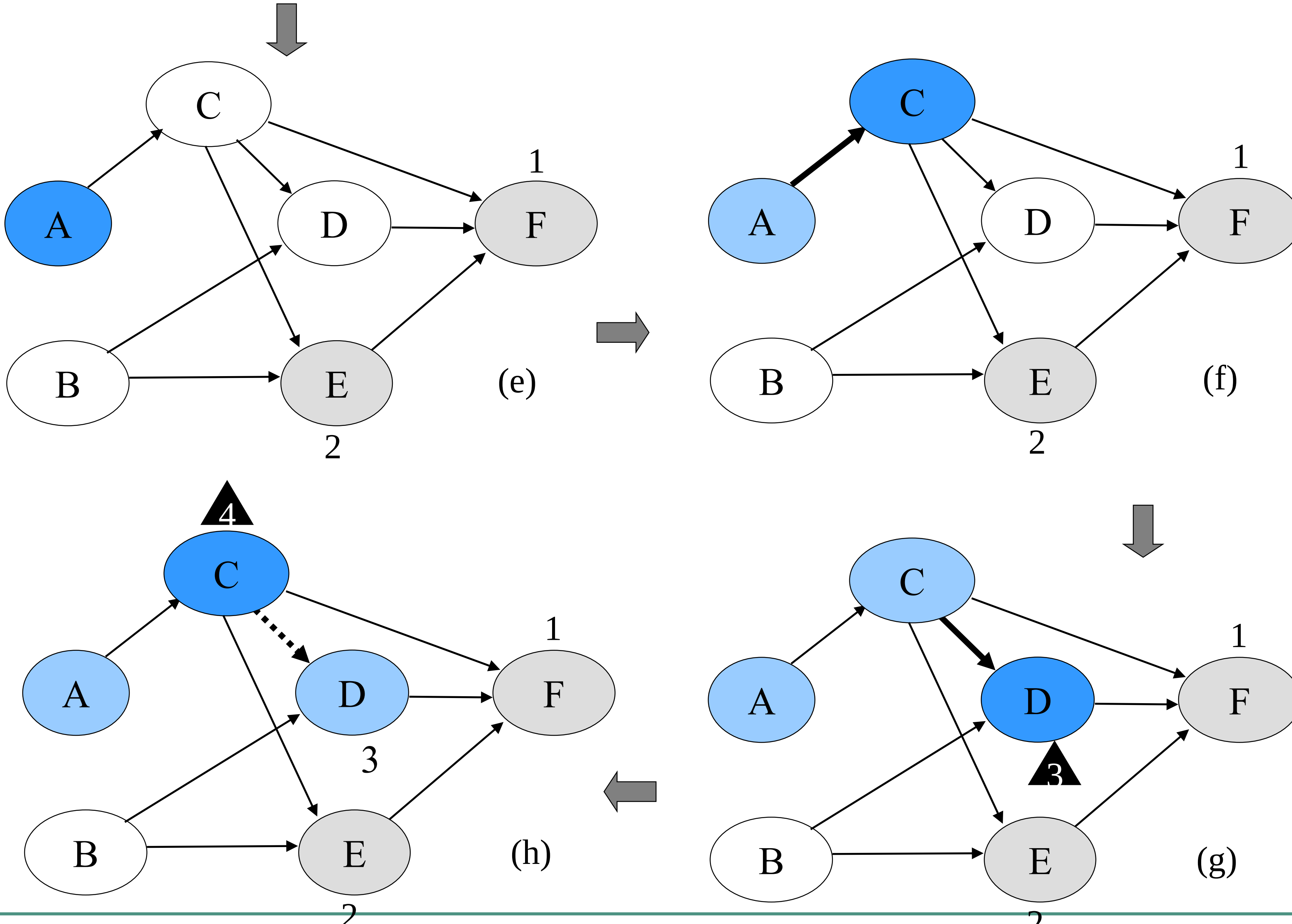
```
topologicalSort2(G)
{
      for each v∈V
            visited[v] ← NO;
      for each v∈V
            if (visited[v] = NO) then DFS-TS(v);
}
DFS-TS(v)
{
      visited[v] ← YES;
      for each x∈L(v)   ▷ L(v) : vertices adjacent to u
            if (visited[x] = NO) then DFS-TS(x);
      insert v onto the front of linked list R;
}
```

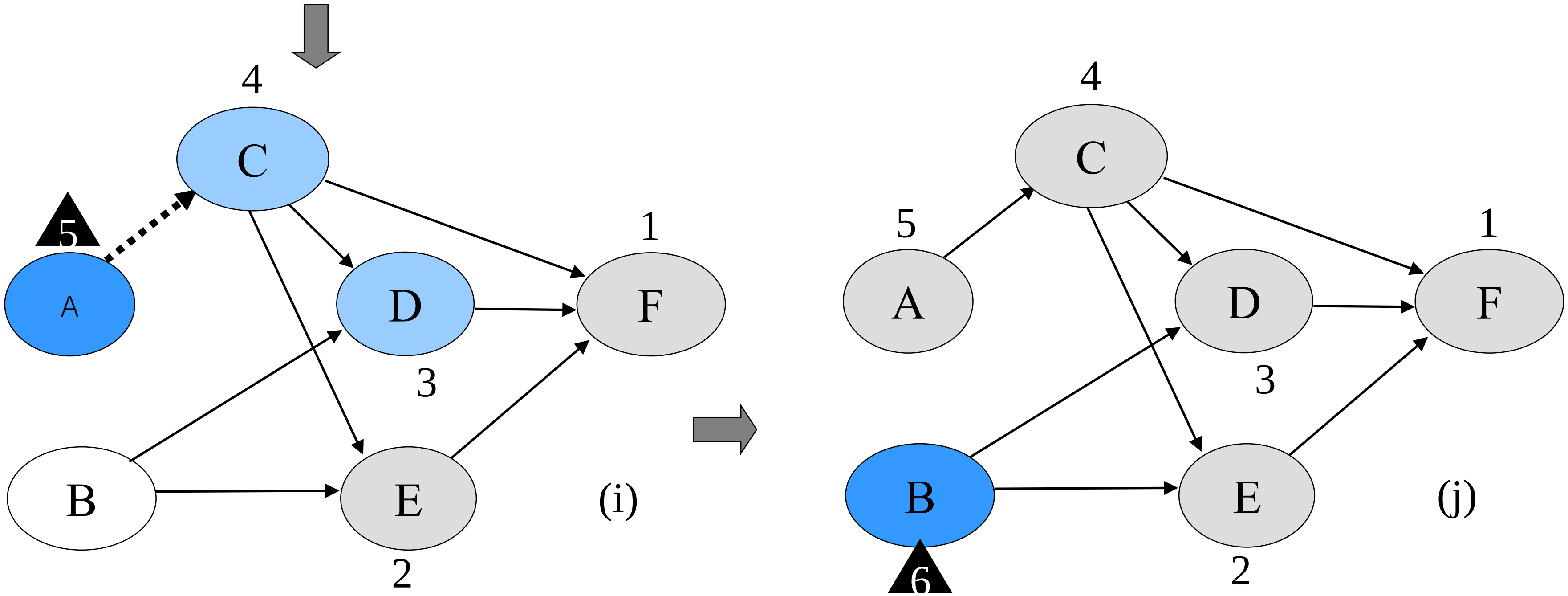
✓Time complexity: $\Theta(|V|+|E|)$

✓ R contains vertices in topologically sorted order

Topological Sort 2




C
A
D
F
1
B
E
2
(e)
C
A
D
F
1
B
E
2
(f)
4
C
A
D
F
1
3
B
E
2
(h)
C
A
D
F
1
3
B
E
2
(g)


4
C
5
A
D
F
1
3
B
E
2
(i)
4
C
5
1
A
D
F
3
B
E
(j)
6
2

Shortest Paths

- Input
  - weighted, directed graph
  - An undirected graph can be converted to a directed graph
    - Undirected edge (u, v) is converted to two directed edges (u, v) and (v, u)
- Shortest path from vertex u to vertex v
  - Path such that the sum of weights of edges on the path is minimum
  - Not defined if there is a cycle such that the sum of weights of edges on the cycle is negative

- Single-source shortest-paths problem
  - – Find a shortest path from a given source to each vertex
  - ➢ Dijkstra algorithm
    - Edge weights are non-negative (Negative edge weights not allowed)
  - ➢ Bellman-Ford algorithm
    - Negative edge weights are allowed
  - ➢ Directed acyclic graphs
- All-pairs shortest-paths problem
  - – Find a shortest path between every pair of vertices
  - ➢ Floyd-Warshall algorithm

Dijkstra Algorithm

Edge weights are non-negative

```
Dijkstra(G, r)
▷ G=(V, E): input graph
▷ r: source vertex
{
    S ← Φ ;                          ▷ S : set of vertices whose shortest-path weights are determined
    for each u∈V
        d[u] ← ∞ ;
    d[r] ← 0 ;
    while (S≠V){                     ▷ repeated n times
        u ← extractMin(V-S, d) ;
        S ← S ∪ {u};
        for each v∈L(u)              ▷ L(u) : vertices adjacent to u
            if (v∈V-S and d[u] +w[u, v] < d[v] ) then {
                    d[v] ← d[u] + w[u, v];
                    prev[v] ← u;
    }
}
```

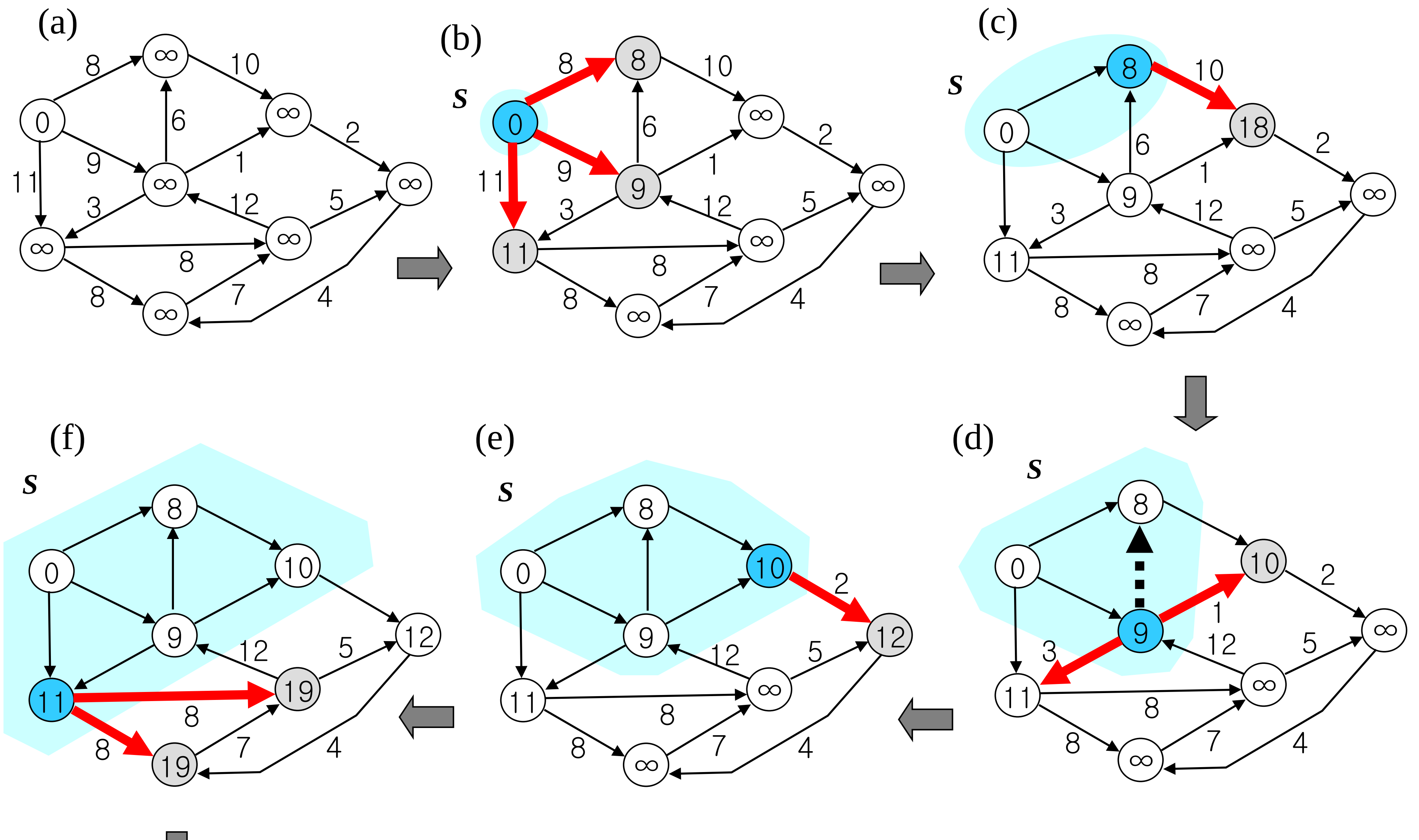
relaxation

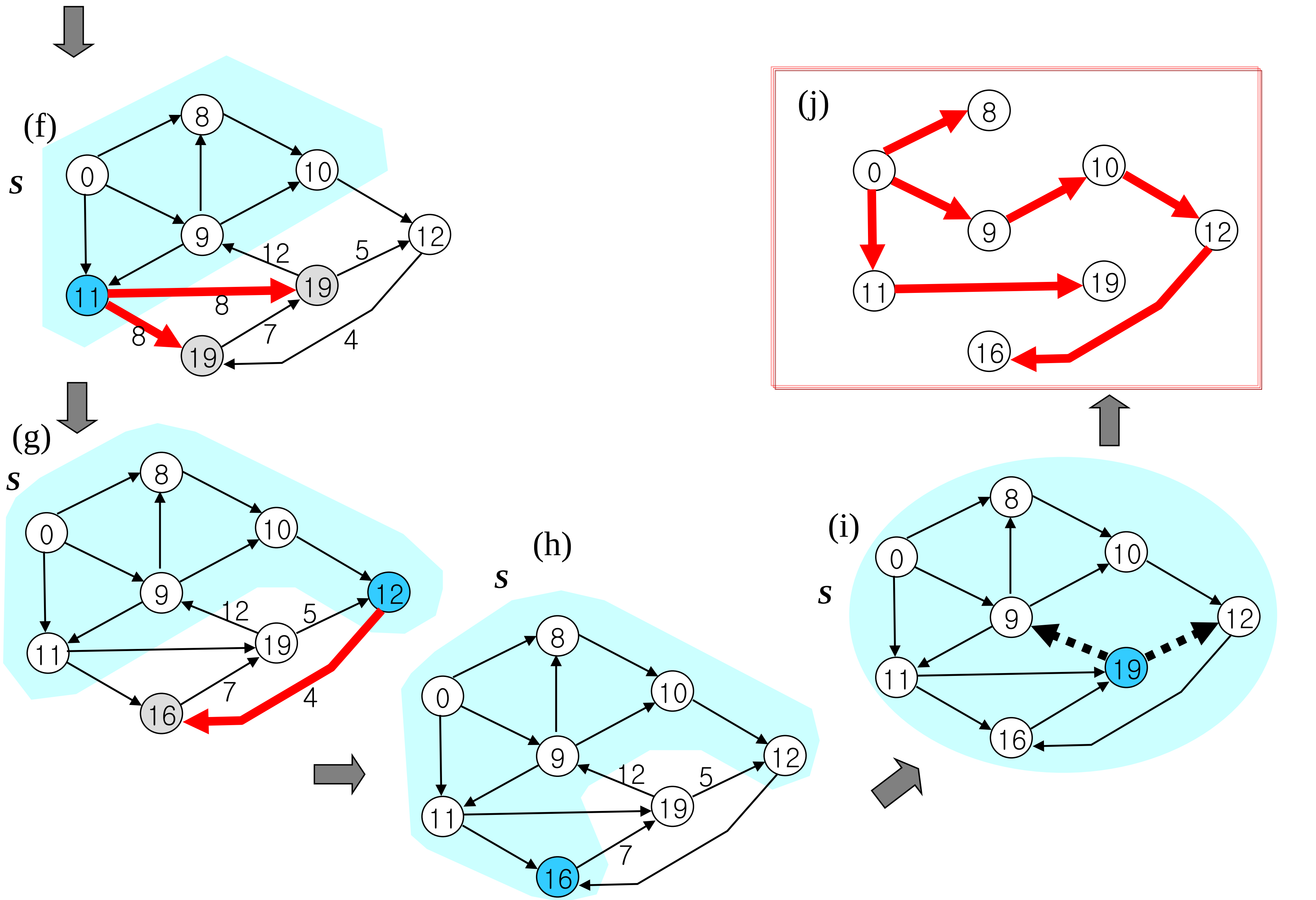
```
extractMin(Q, d[])
{
    extract vertex u in Q such that d[u] is minimum
}
```

✓Time complexity: $O(|E|\log|V|)$

using heap

Dijkstra Algorithm





(f)
s
0
8
9
10
11
12
19
19
12
5
8
8
7
4
(g)
s
16
(h)
s
(i)
s
(j)

Bellman-Ford Algorithm as Dynamic Programming

- d_t^k : shortest-path weight from source r to vertex t using at most k edges
- Goal: $d_t^{n\text{-}1}$

✓ recurrence

$$
\begin{cases}
d_r^0 = 0 \\
d_t^0 = \infty, \quad t \neq r \\
d_v^k = \min\limits_{\textit{for each edge } (u,\, v)} \{d_u^{k\text{-}1} + w_{uv}\}, \quad k > 0
\end{cases}
$$

Bellman-Ford Algorithm

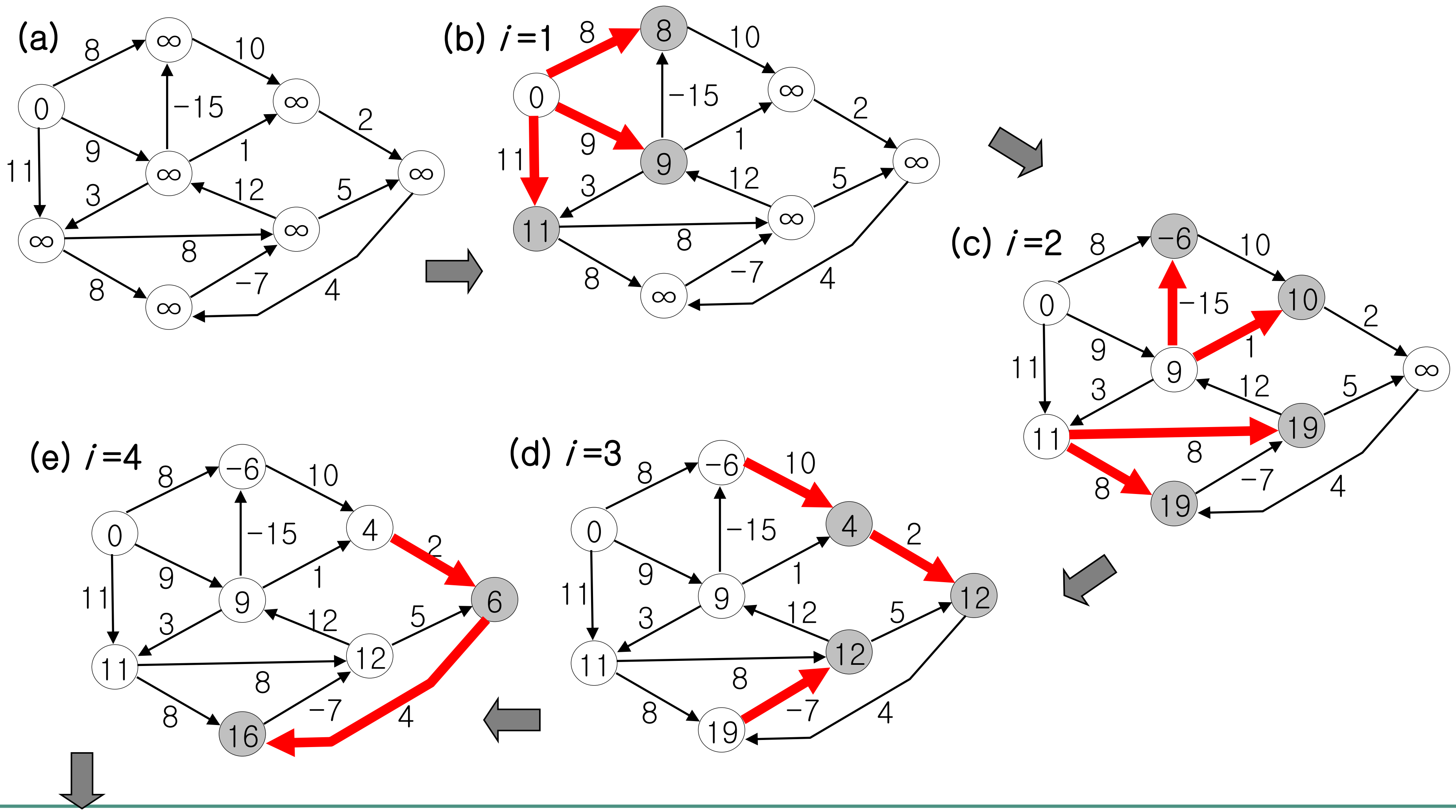
Negative edge weights allowed

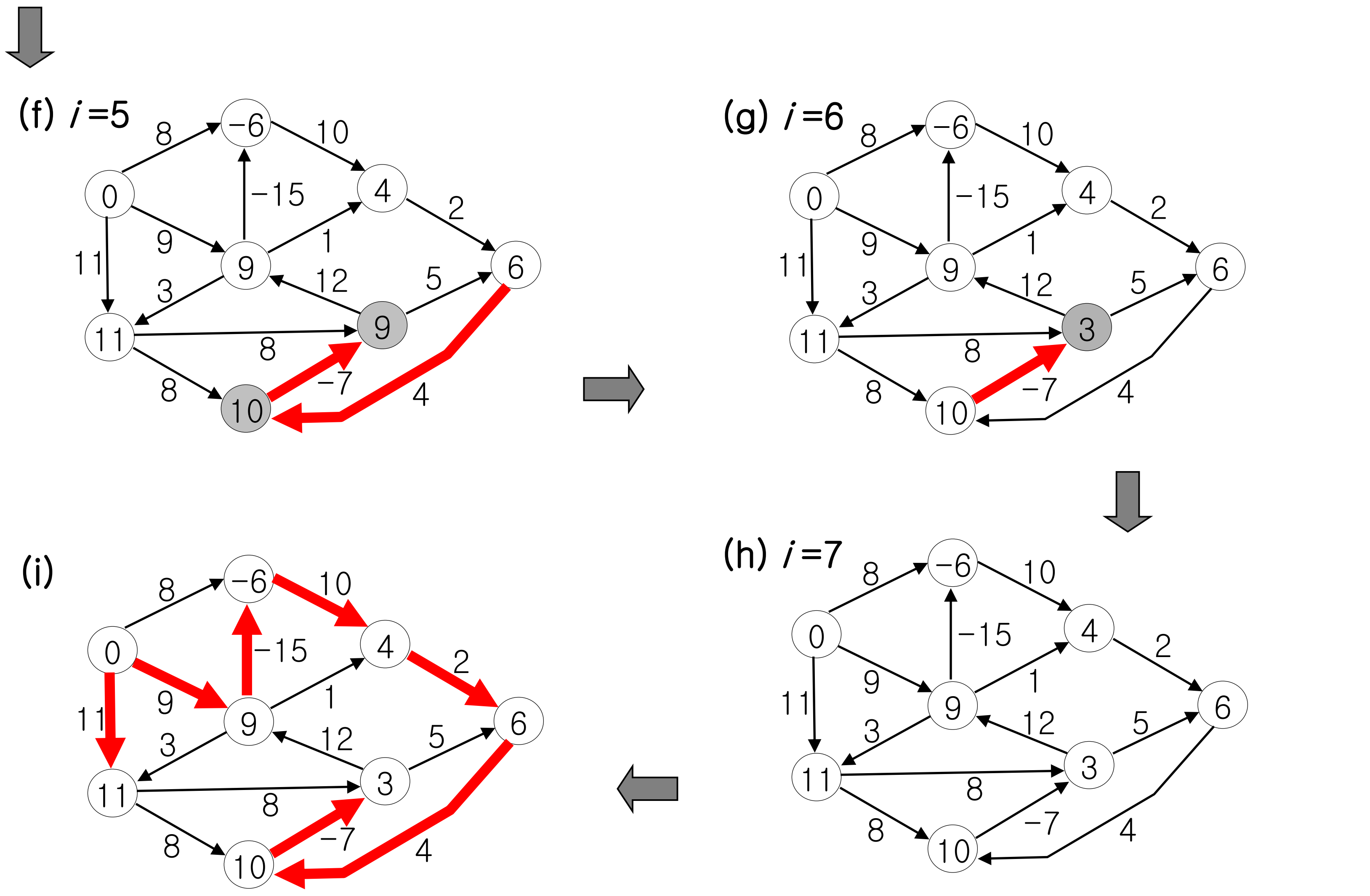
```
BellmanFord(G, r)
{
    for each u∈V
            d[u] ← ∞;
    d[r] ← 0;
    for i ← 1 to |V|-1
            for each (u, v) ∈E
                    if (d[u] + w[u, v] < d[v]) then {
                            d[v] ← d[u] + w[u, v] ;
                            prev[v] ← u;
                    }
    ▷ check for negative-weight cycle
    for each (u, v) ∈E
            if (d[u] + w[u, v] < d[v]) output "no solution";
}
```

relaxation

✓Time complexity: $\Theta(|E||V|)$

Bellman-Ford Algorithm




(f) i =5
(g) i =6
(h) i =7
(i)
0
-6
4
9
6
11
10
3
8
10
-15
2
1
12
5
-7
4

Shortest Paths in DAG

- Input: Directed Acyclic Graph (DAG)
- Single-source shortest paths in DAG can be found in linear time

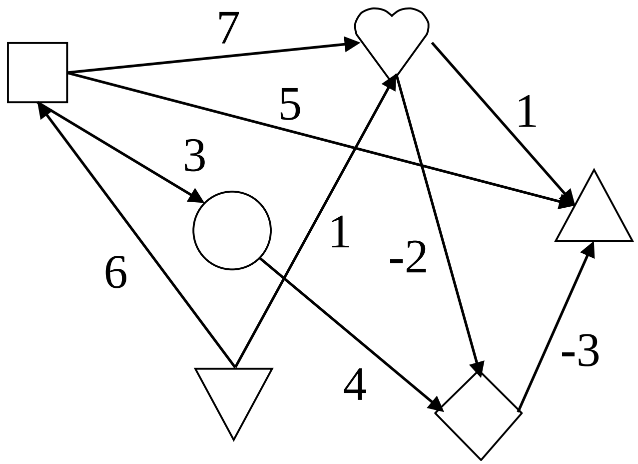
Shortest Paths in DAG

DAG-ShortestPath(G, r)
{
    for each $u \in V$
        $d_u \leftarrow \infty$;
    $d_r \leftarrow 0$;
    topologically sort the vertices of G
    for each $u \in V$ in topologically sorted order
        for each $v \in L(u)$ ▷ $L(u)$: vertices adjacent to u
            if $(d_u + w_{u,v} < d_v)$ **then** $d_v \leftarrow d_u + w_{u,v}$;
}

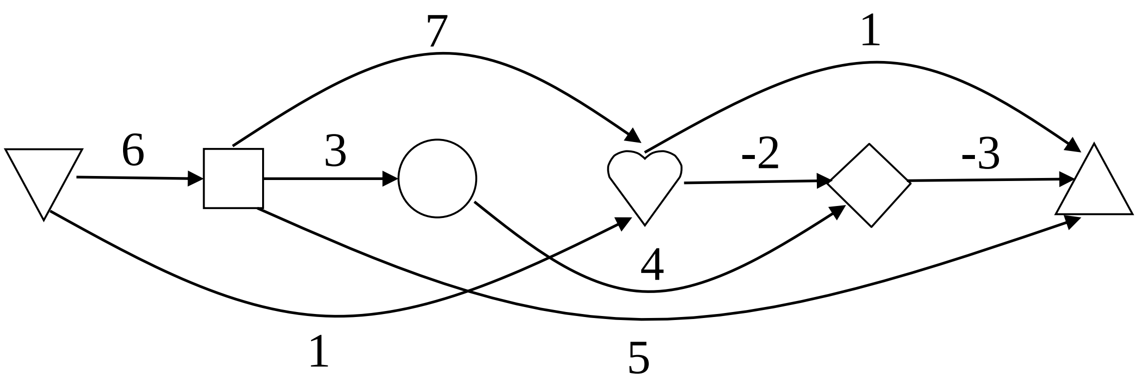
✓Time complexity: $\Theta(|V|+|E|)$

DAG-ShortestPath

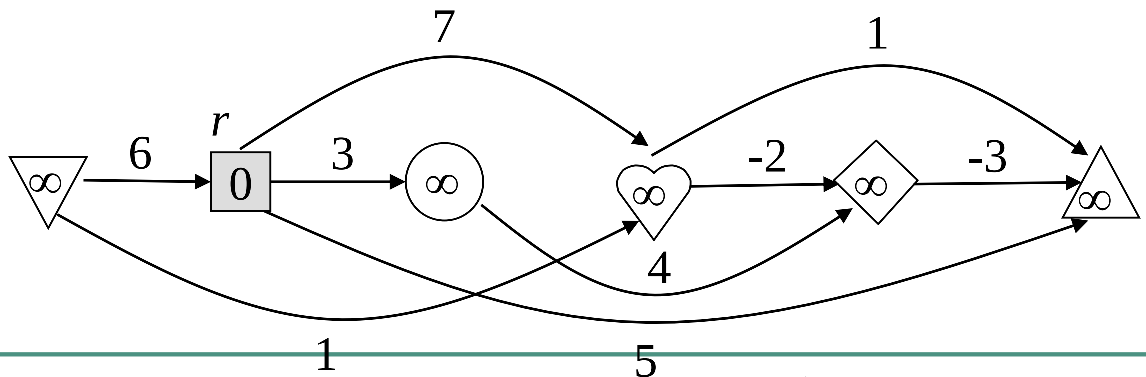
(a)



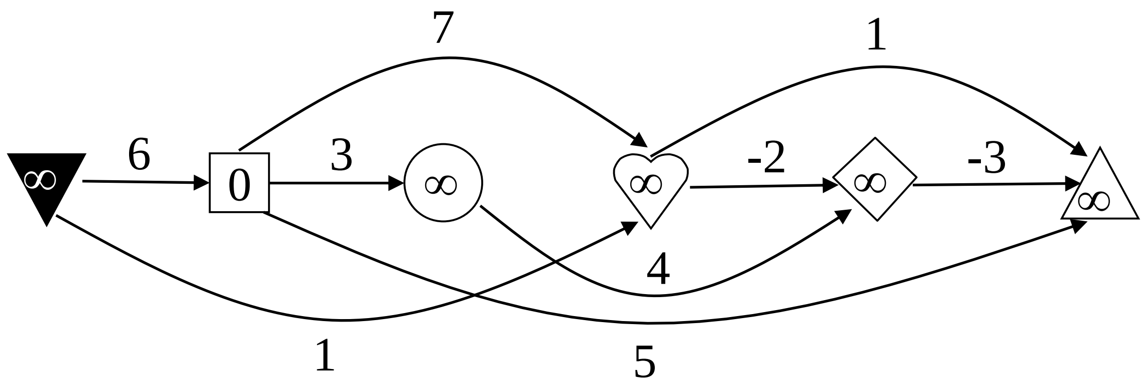
(b)



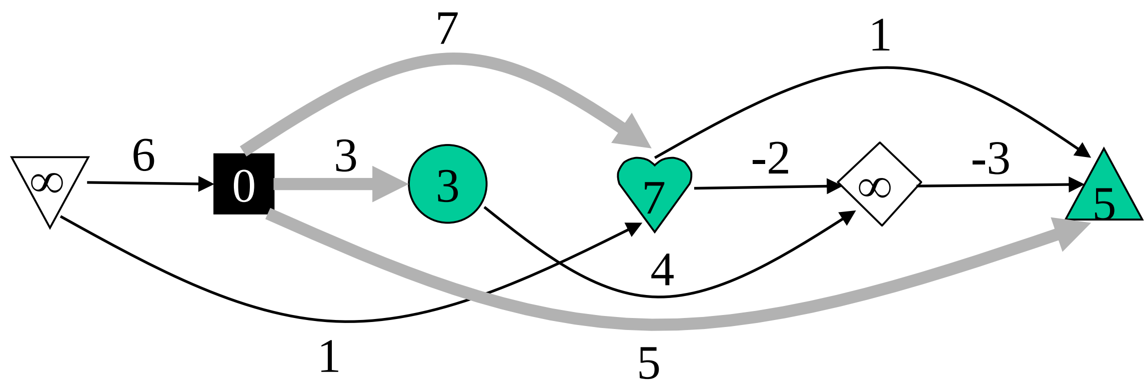
(c)



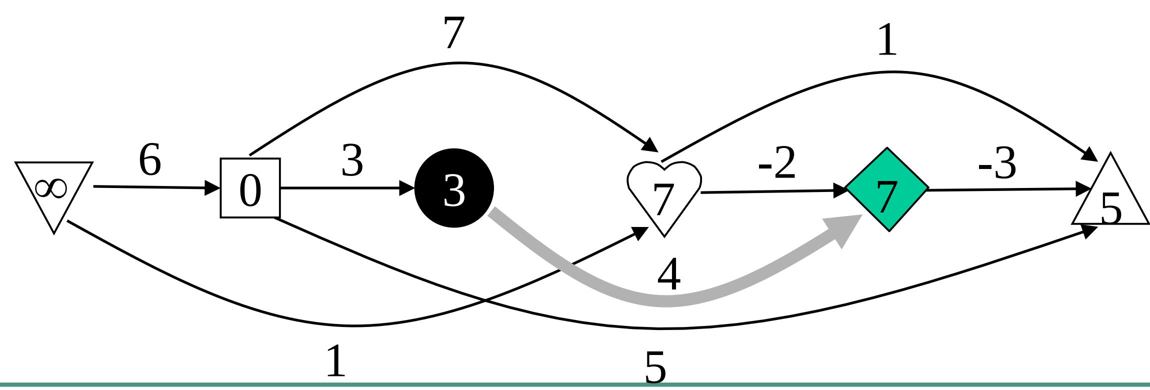
(d)



(e)



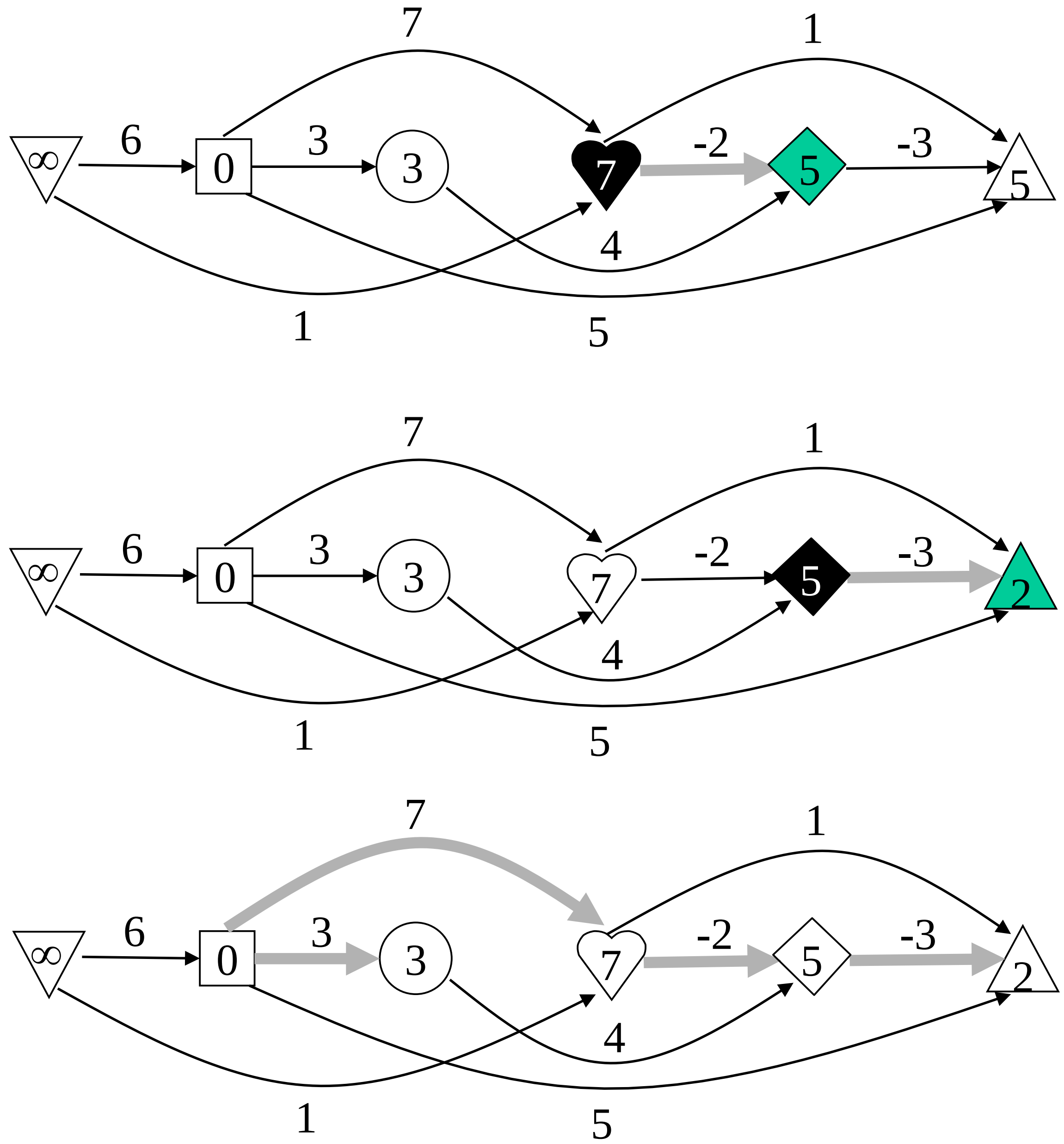
(f)



(g)

(h)

(i)

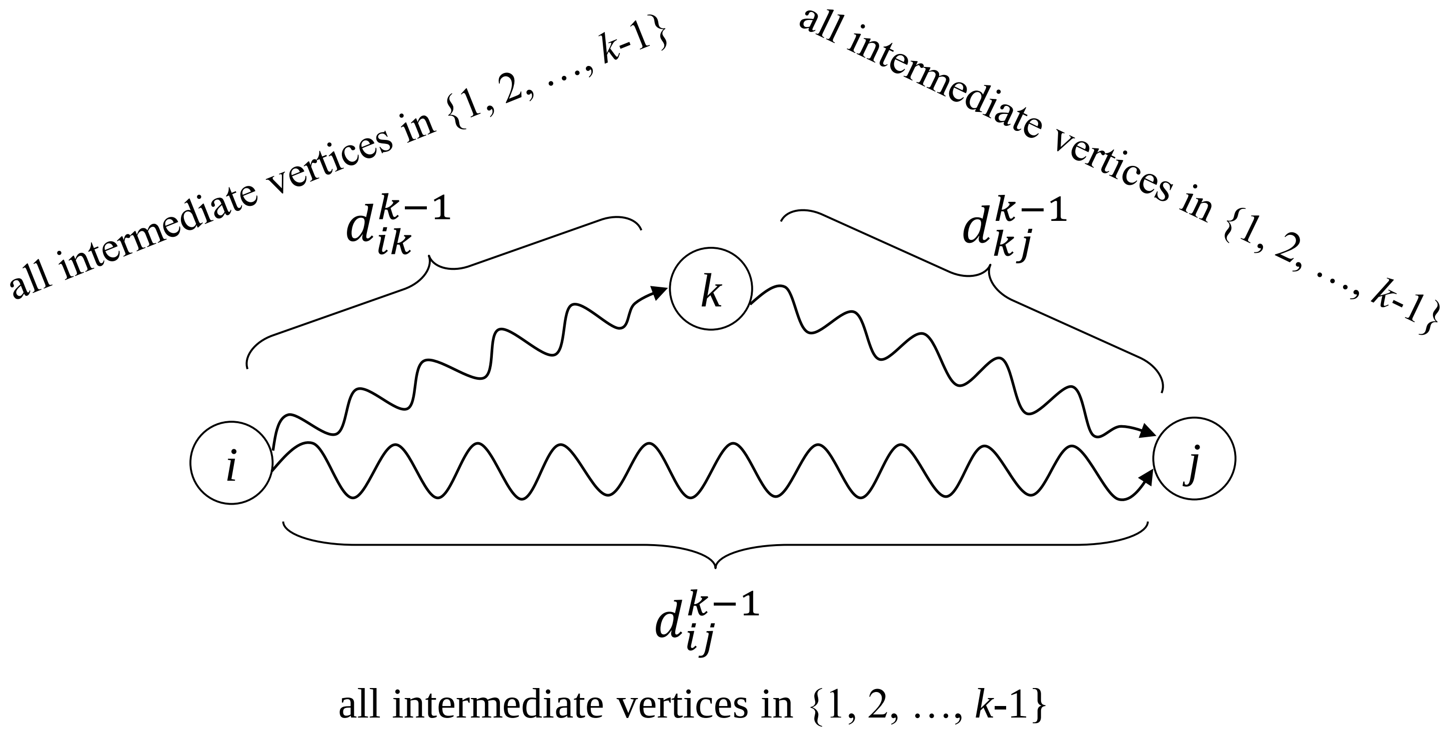


Floyd-Warshall Algorithm

- All-pairs shortest-paths problem
- Applications
  - Road Atlas
  - Navigation system
  - Network communication

d_{ij}^{k} : shortest-path weight from vertex v_i to vertex v_j using intermediate vertices in $\{v_1, v_2, \ldots, v_k\}$

$$d_{ij}^{k} = \begin{cases} w_{ij}\ , & k = 0 \\ \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}, & k \geq 1 \end{cases}$$



Floyd-Warshall Algorithm

FloydWarshall(G)
{
    for $i \leftarrow 1$ **to** n
        for $j \leftarrow 1$ **to** n
            $d^0_{ij} \leftarrow w_{ij}$;
    for $k \leftarrow 1$ **to** n $\triangleright$ intermediate vertices in $\{1, 2, \ldots, k\}$
        for $i \leftarrow 1$ **to** n $\triangleright$ i : start vertex
            for $j \leftarrow 1$ **to** n $\triangleright$ j : end vertex
                $d^k_{ij} \leftarrow \min\{d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj}\}$;
}

- ✓ Time complexity: $O(n^3)$
- ✓ It works without superscripts. Space complexity: $O(n^2)$

Strongly Connected Components

- Input: directed graph
  - A directed graph is strongly connected if for every pair of vertices u and v, u is reachable from v, and v is reachable from u.
  - A maximal subgraph which is strongly connected is called a strongly connected component.
- Find strongly connected components of a directed graph

Strongly Connected Components

stronglyConnectedComponent(G)
{

1 Run DFS on G to compute finish time $f[v]$ for each vertex v.

2 Compute G^R (transpose of G) where direction of each edge in G is reversed.

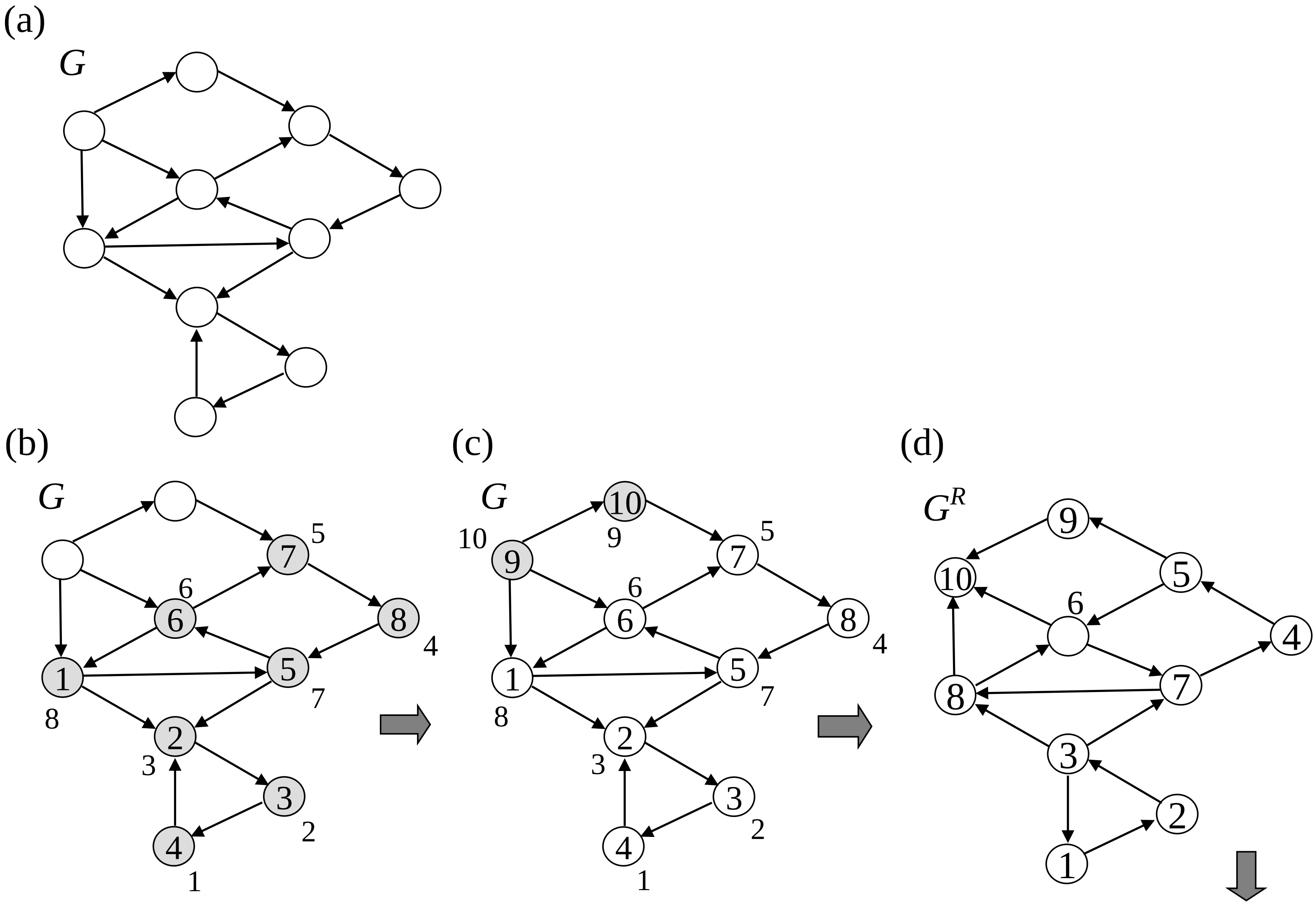
3 Run DFS on G^R (in the main loop of DFS, consider vertices in decreasing order of $f[v]$).

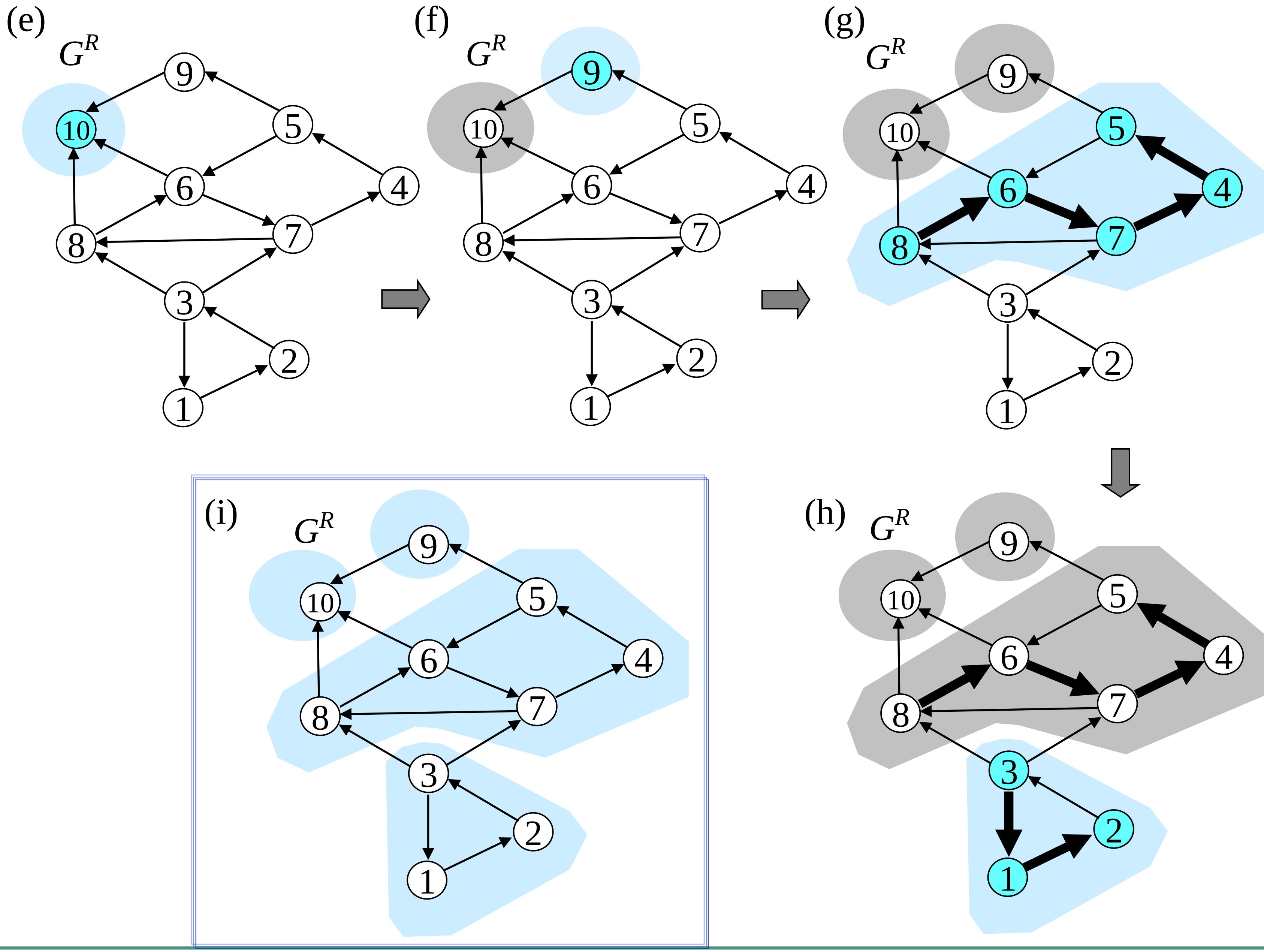
4 Output each tree made in 3 as a strongly connected component.

}

✓Time complexity: $\Theta(|V|+|E|)$

strongly Connected Component




(e)
G^R
(f)
G^R
(g)
G^R
(h)
G^R
(i)
G^R
1 2 3 4 5 6 7 8 9 10

Thank you