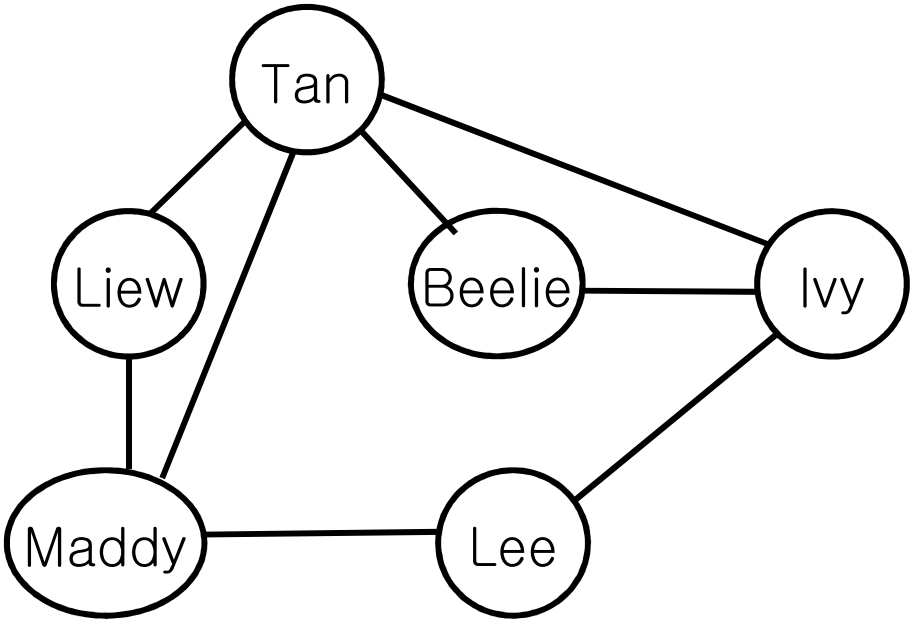
Graph Algorithms

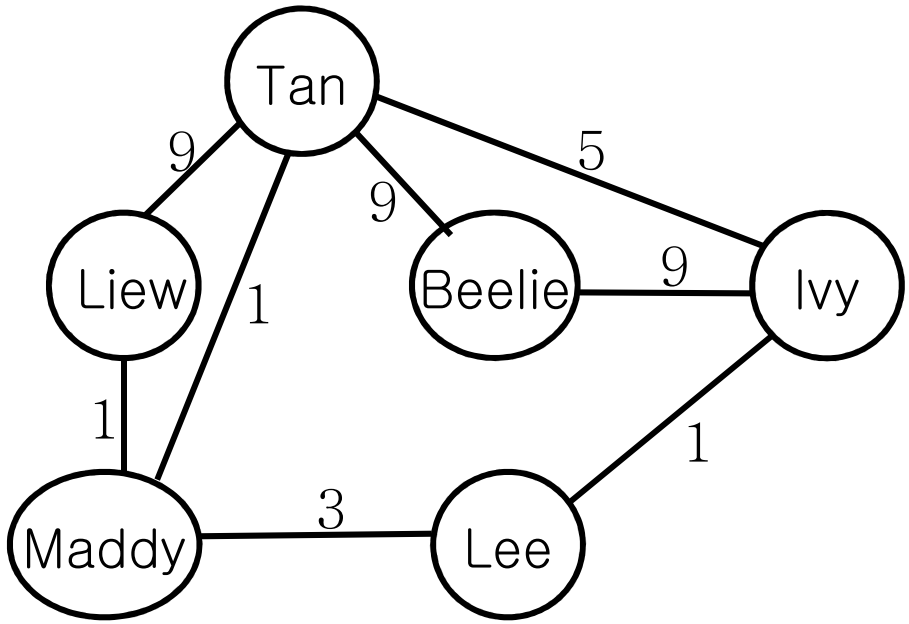
Graph

- Describes things or phenomena using vertex and edge
- Graph $G = (V, E)$
  - V: a set of vertices
  - E: a set of edges
- If two vertices are connected by an edge, they are adjacent
  - Edge indicates the relation between the two vertices

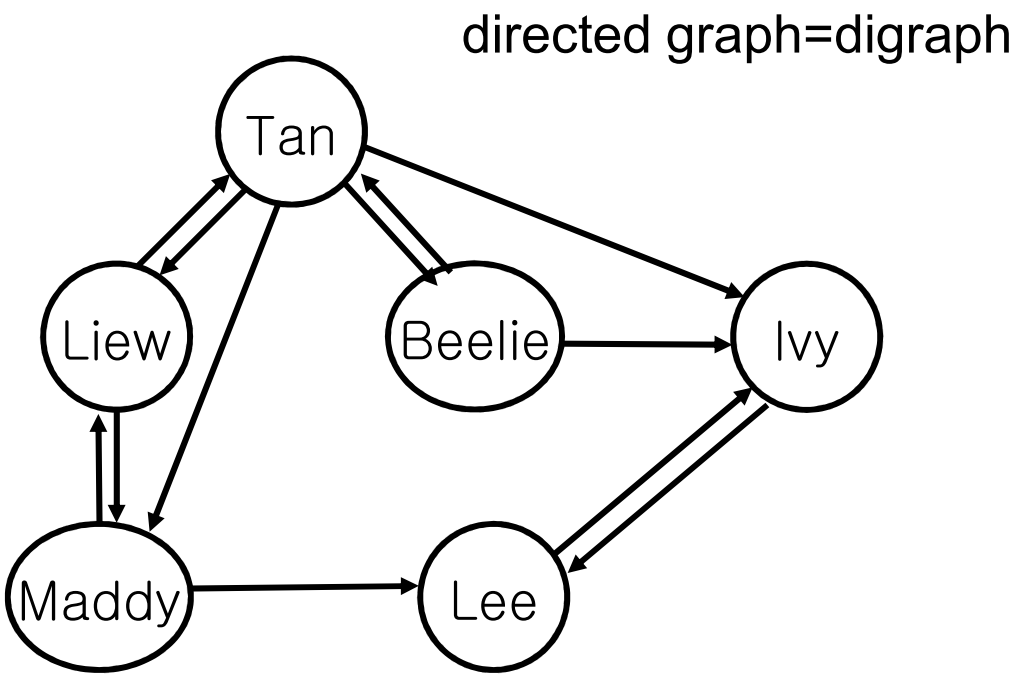
Example



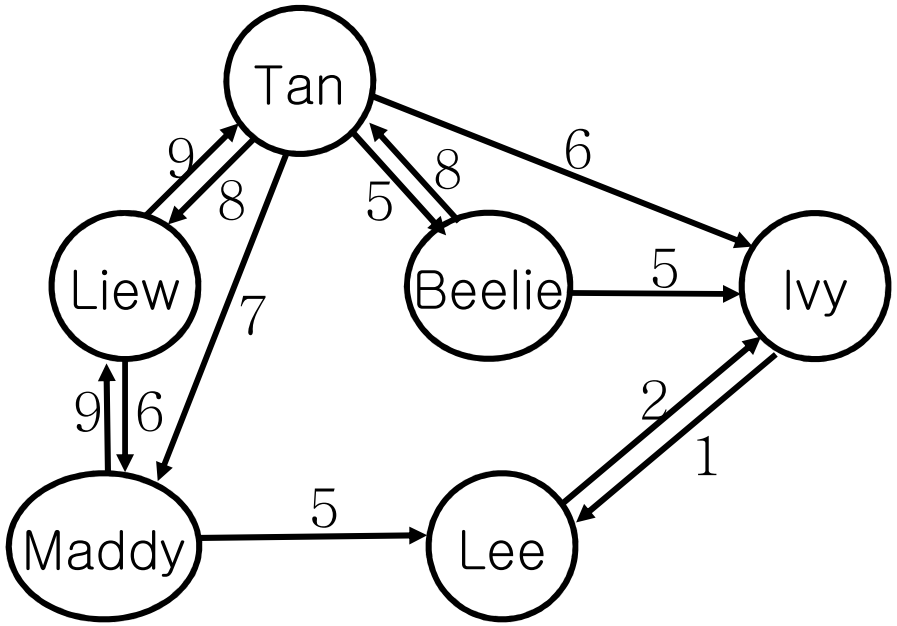
Graph for students of a certain class



Graph with certain weights



Graph with directed edges

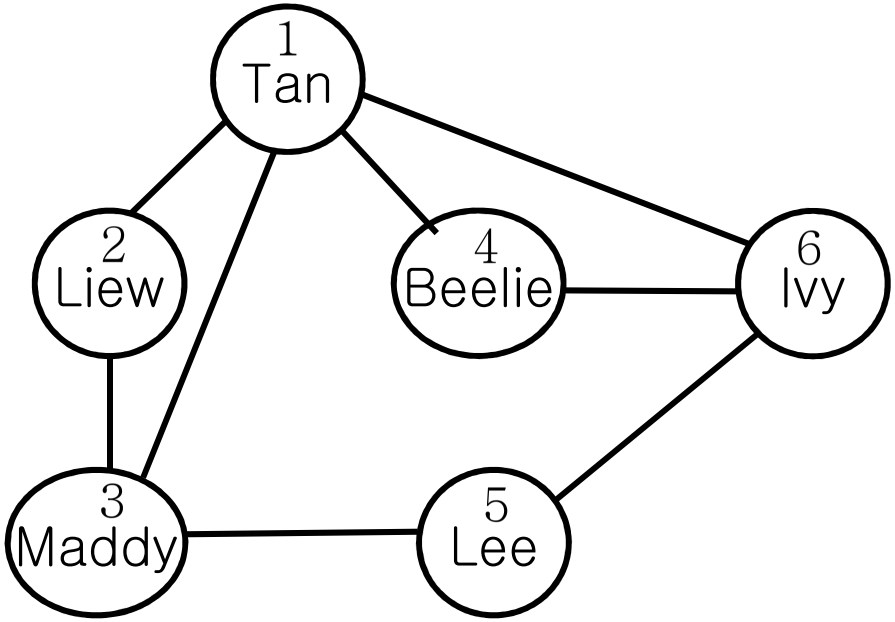


Digraph with weights

Representation of Graph 1: Adjacency Matrix

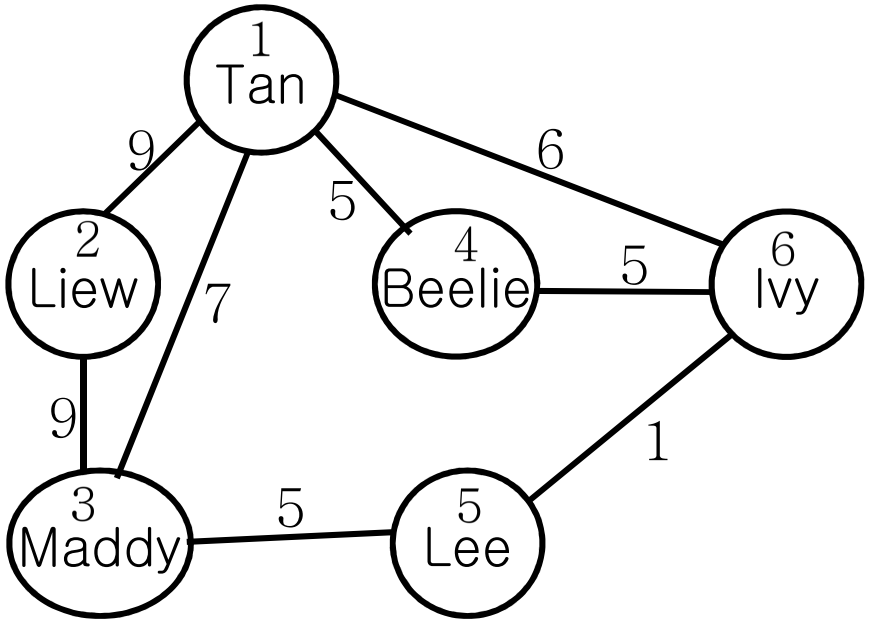
N: # of vertices

- Adjacency matrix
  - Nx N Matric
    - Element (i, j) = 1 : There is an edge between vertex i and vertex j
    - Element (i, j) = 0 : There is NO edge between vertex i and vertex j
  - If Digraph
    - Element (i, j) = 1 : There is a directed edge from vertex i to vertex j
  - If weighted graph
    - Element (i, j) = a weight for the edge



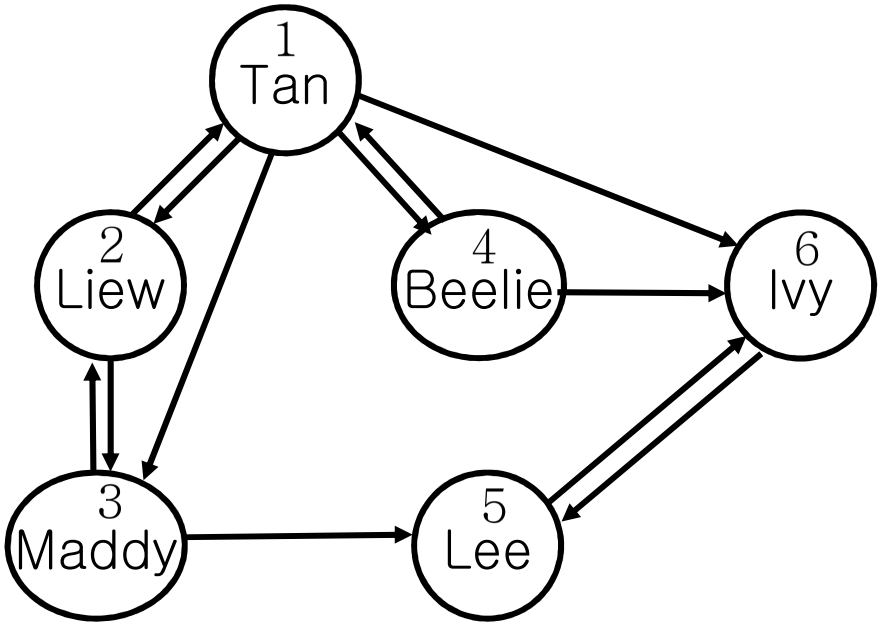
	1	2	3	4	5	6
1	0	1	1	1	0	1
2	1	0	1	0	0	0
3	1	1	0	0	1	0
4	1	0	0	0	0	1
5	0	0	1	0	0	1
6	1	0	0	1	1	0

Undirected Graph



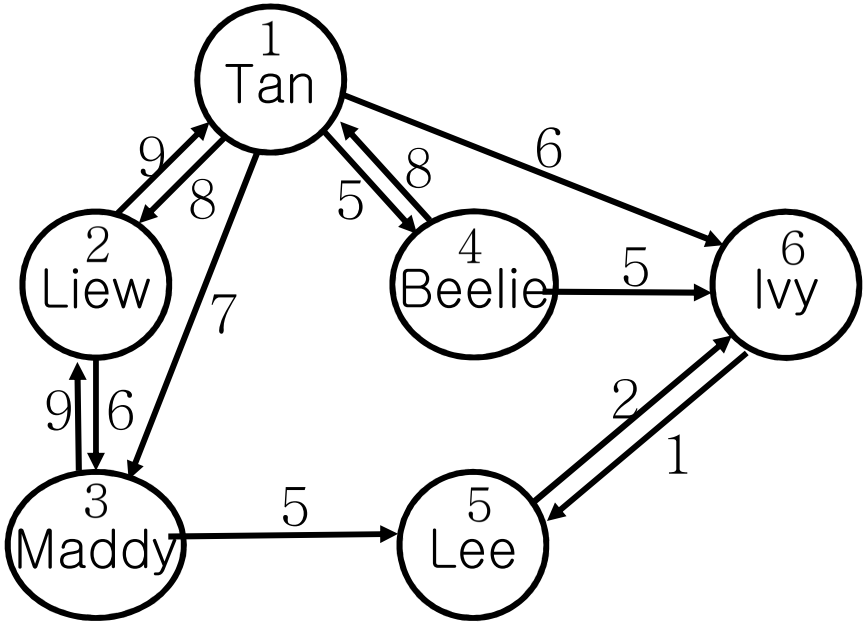
	1	2	3	4	5	6
1	0	9	7	5	0	6
2	9	0	9	0	0	0
3	7	9	0	0	5	0
4	5	0	0	0	0	5
5	0	0	5	0	0	1
6	6	0	0	5	1	0

Weighted graph



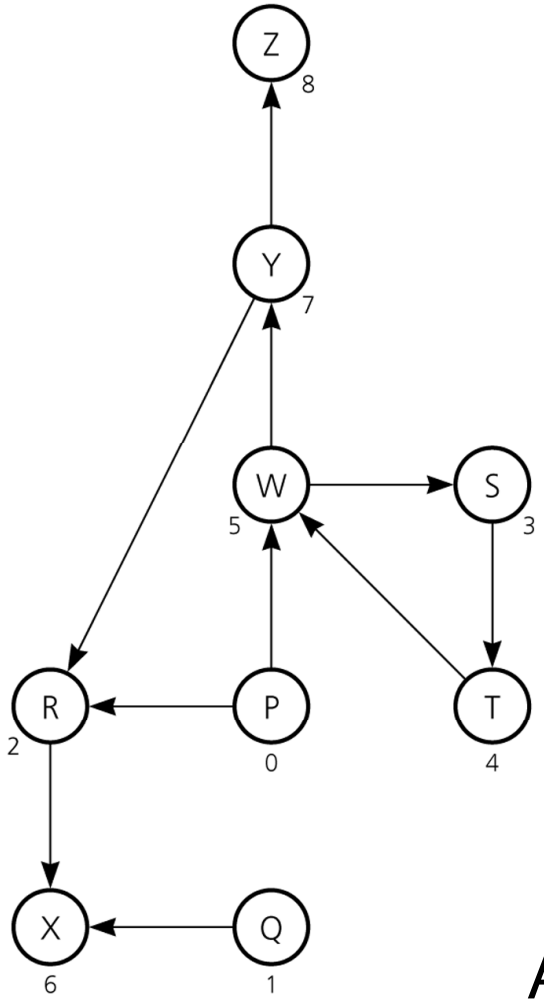
	1	2	3	4	5	6
1	0	1	1	1	0	1
2	1	0	1	0	0	0
3	0	1	0	0	1	0
4	1	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	1	0

Digraph



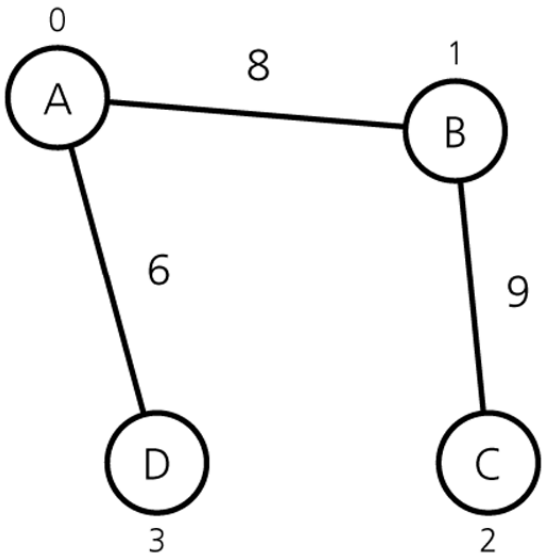
	1	2	3	4	5	6
1	0	8	7	5	0	6
2	9	0	6	0	0	0
3	0	9	0	0	5	0
4	8	0	0	0	0	5
5	0	0	0	0	0	2
6	0	0	0	0	1	0

Weighted Digraph



		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Another example of digraph

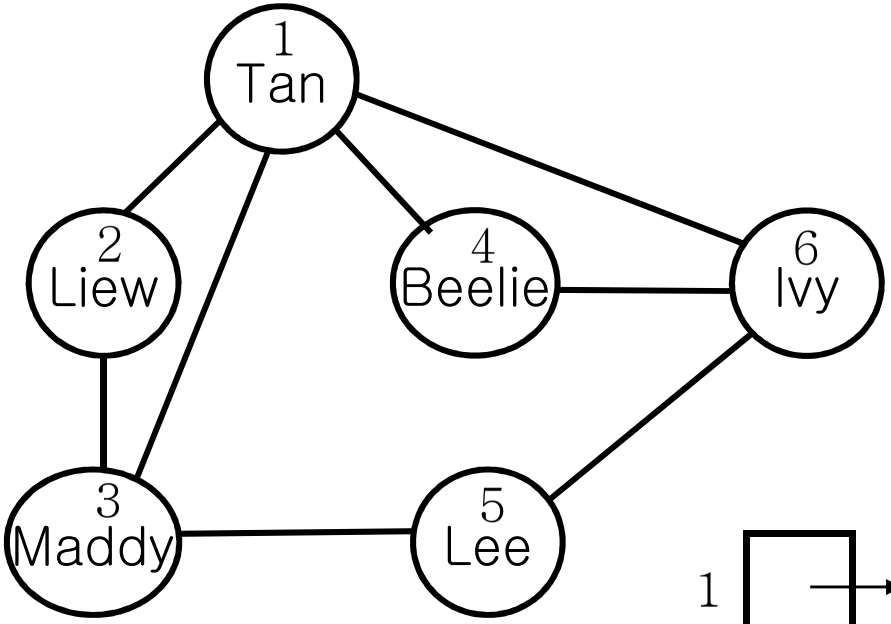


		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

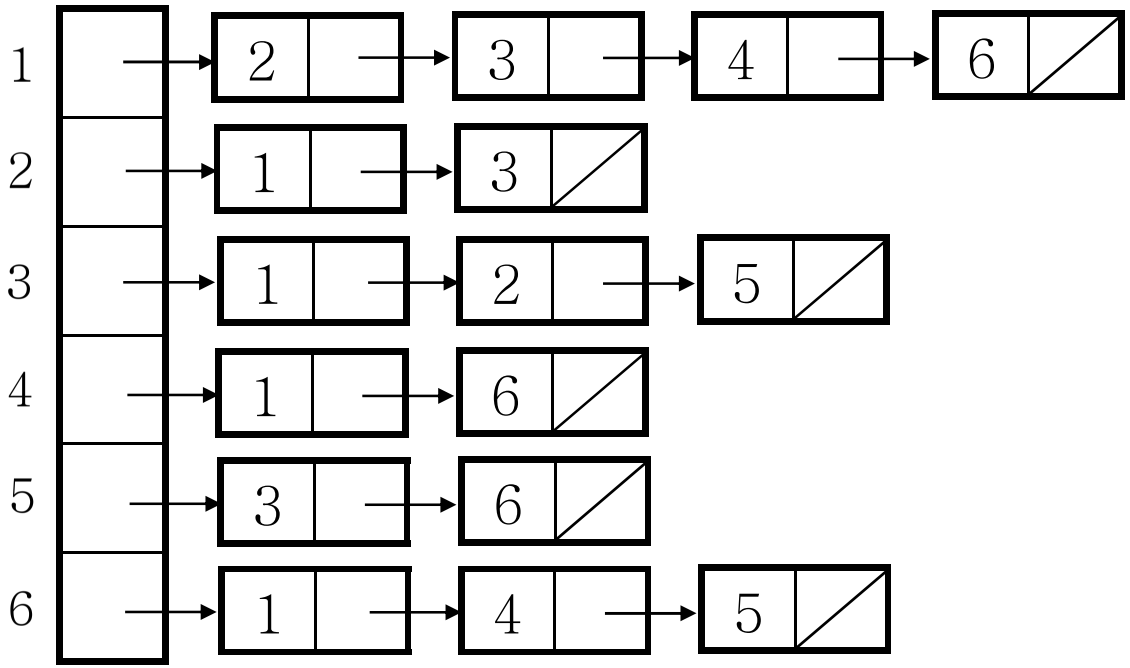
Another example of weighted graph

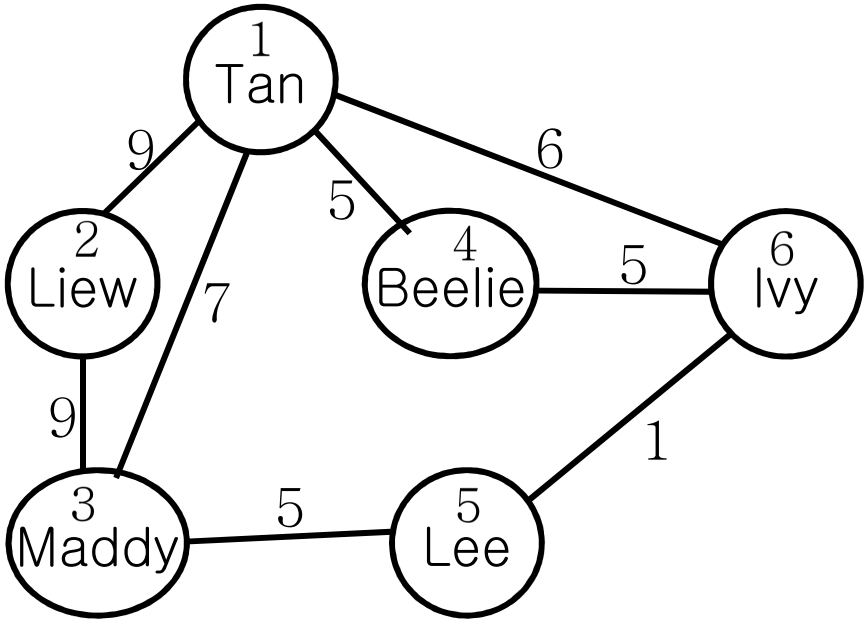
Representation of Graph 2: Adjacency List

- Adjacency list
  - N Linked Lists
  - i^{th} list has vertices adjacent to vertex i
  - In case of weighted graph
    - The list contains weights too

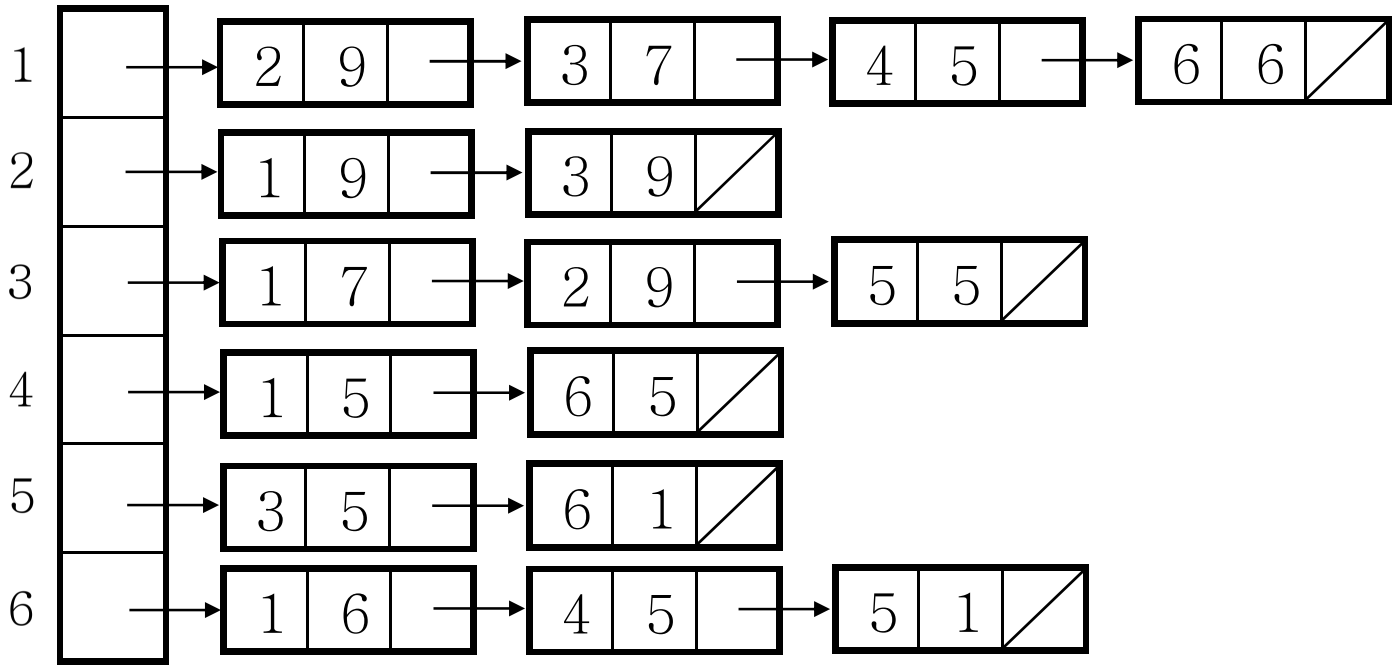

1
Tan
2
Liew
4
Beelie
6
Ivy
3
Maddy
5
Lee

Graph


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


1
Tan
2
Liew
3
Maddy
4
Beelie
5
Lee
6
Ivy
9
6
5
9
7
5
1
5

Digraph


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

Graph Traversal

- Two main methods
  - BFS (Breadth-First Search)
  - DFS (Depth-First Search)

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) : Adjacency List of v
                if (visited[x] = NO) then aDFS(u);
}
```

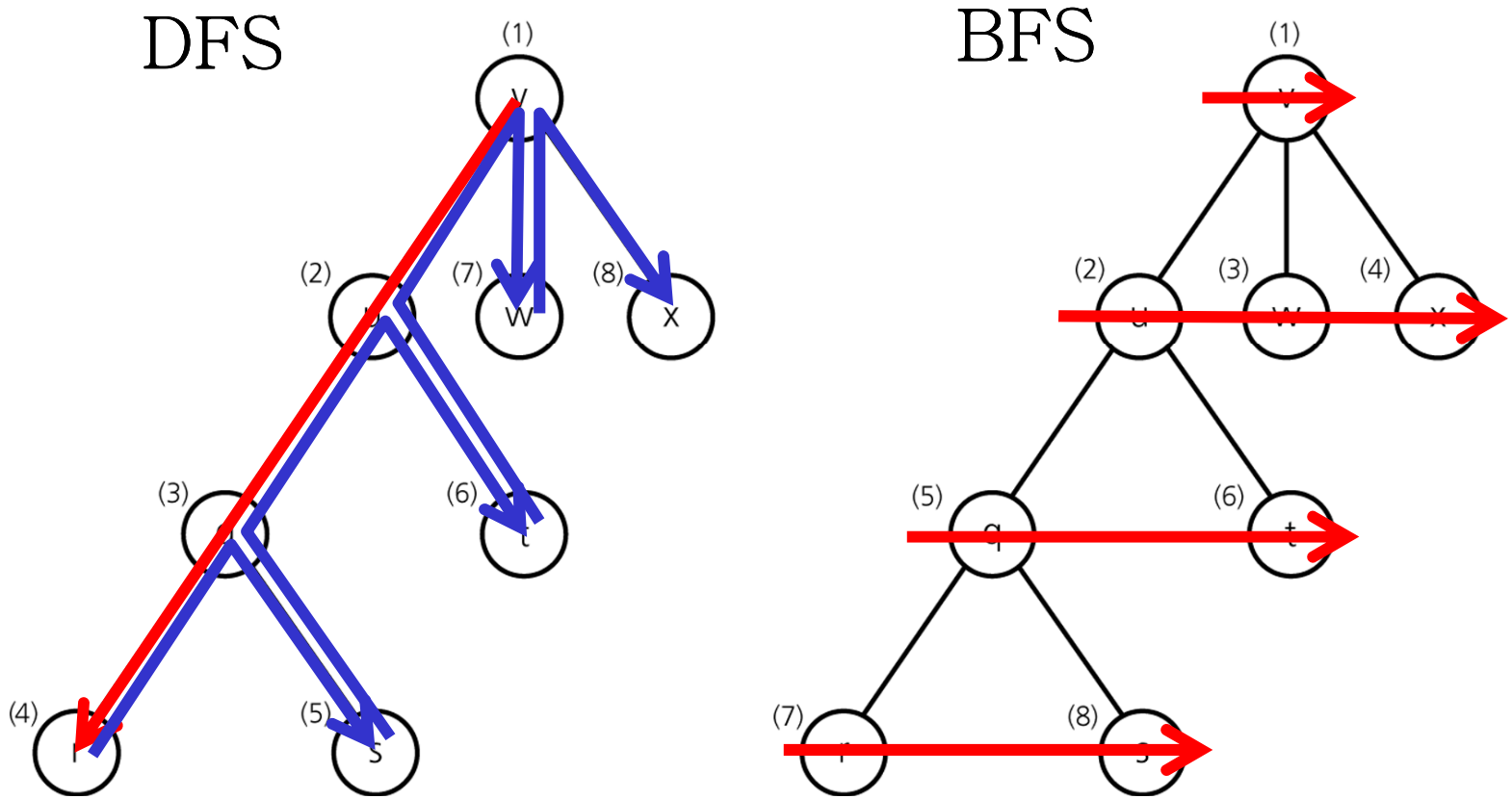
✓Running Time: $\Theta(|V|+|E|)$

BFS

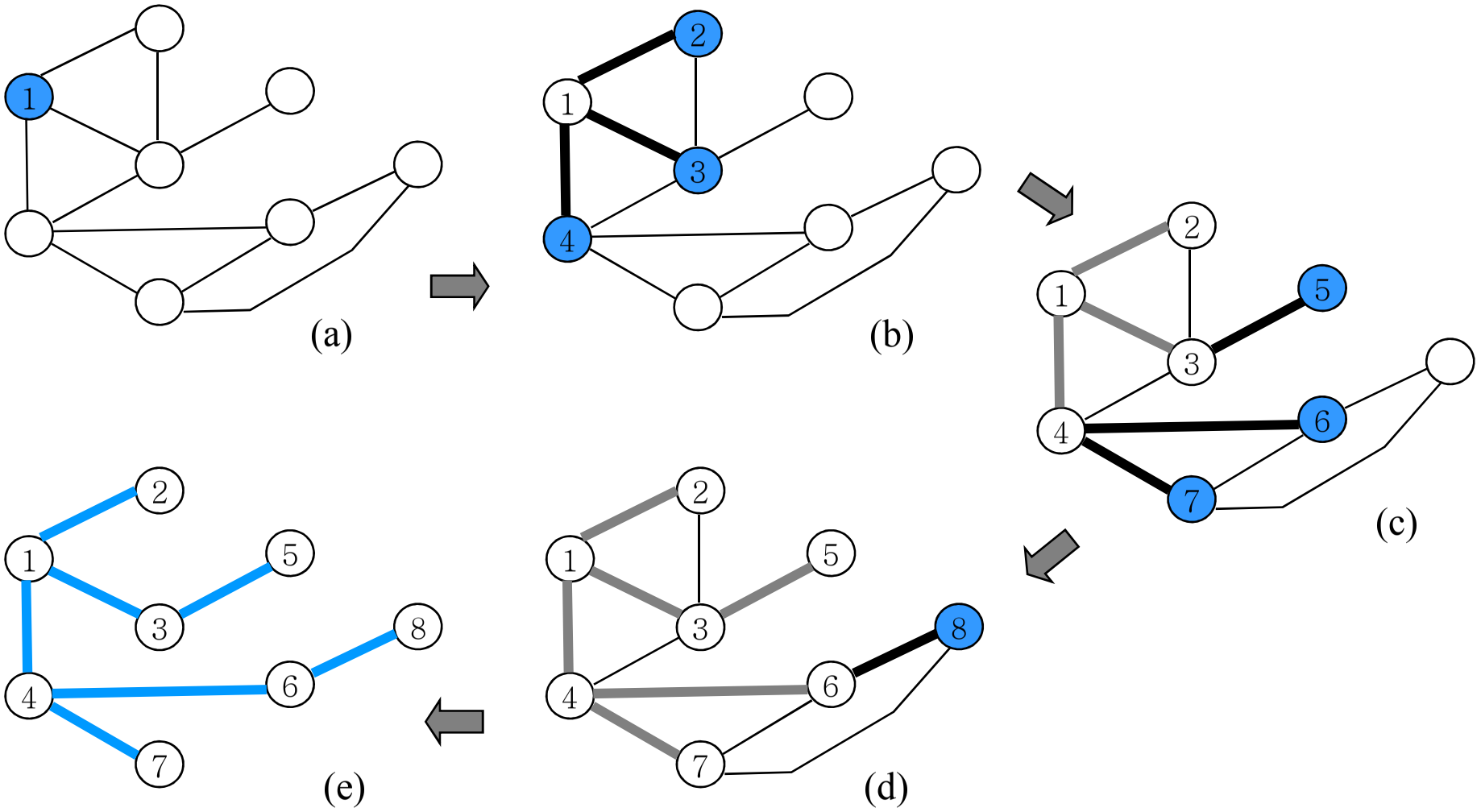
```
BFS(G, v)
{
        for each v ∈ V
                visited[v] ← NO;
        visited[s] ← YES;
        enqueue(Q, s);
        while (Q ≠ Φ) {
                u ← dequeue(Q);
                for each v ∈ L(u)
                        if (visited[v] = NO) then
                                visited[u] ← YES;
                                enqueue(Q, v);
                        }

        }
}
```

✓Running Time: $\Theta(|V|+|E|)$

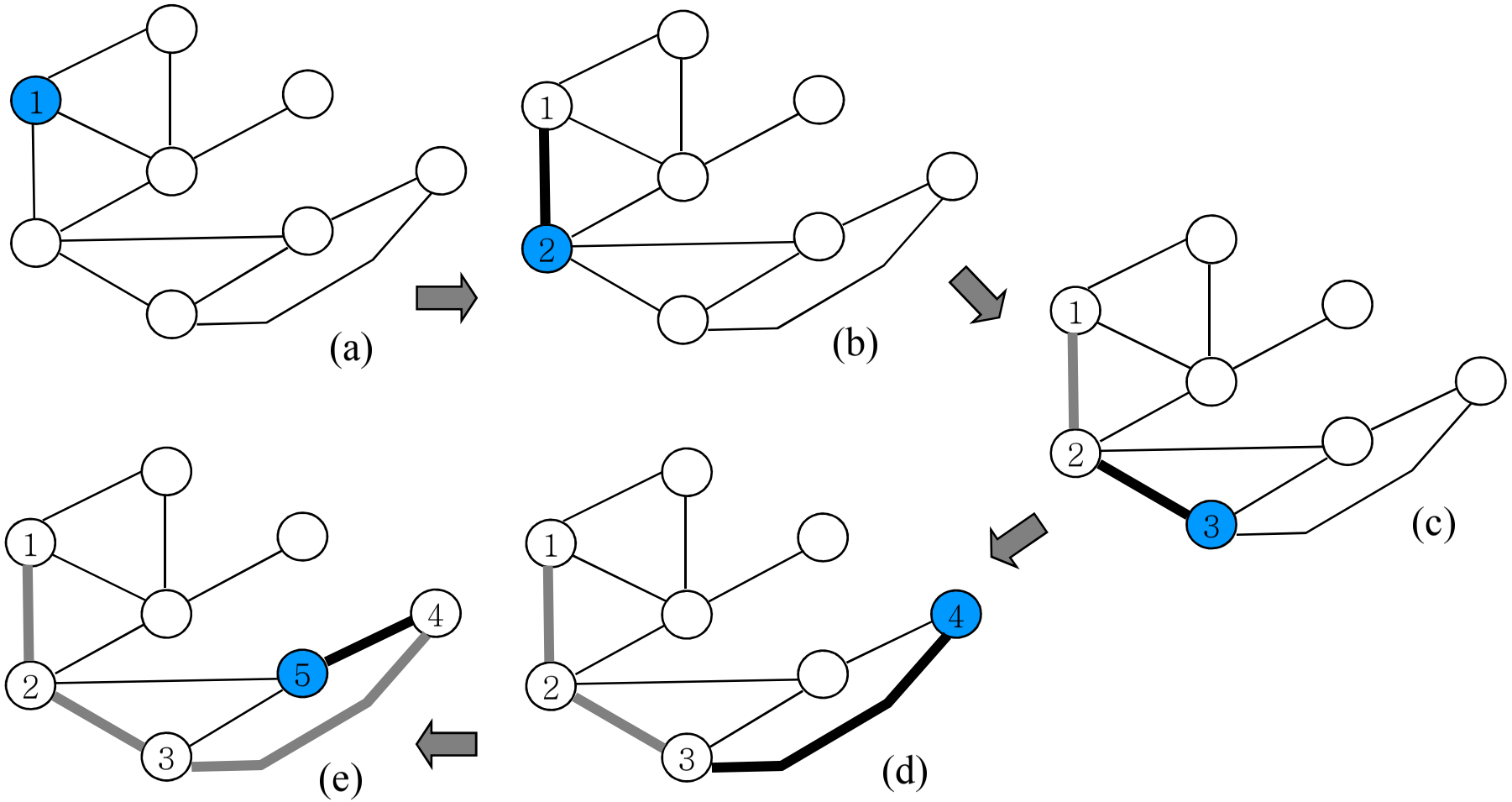
Traverse a tree with DFS/BFS



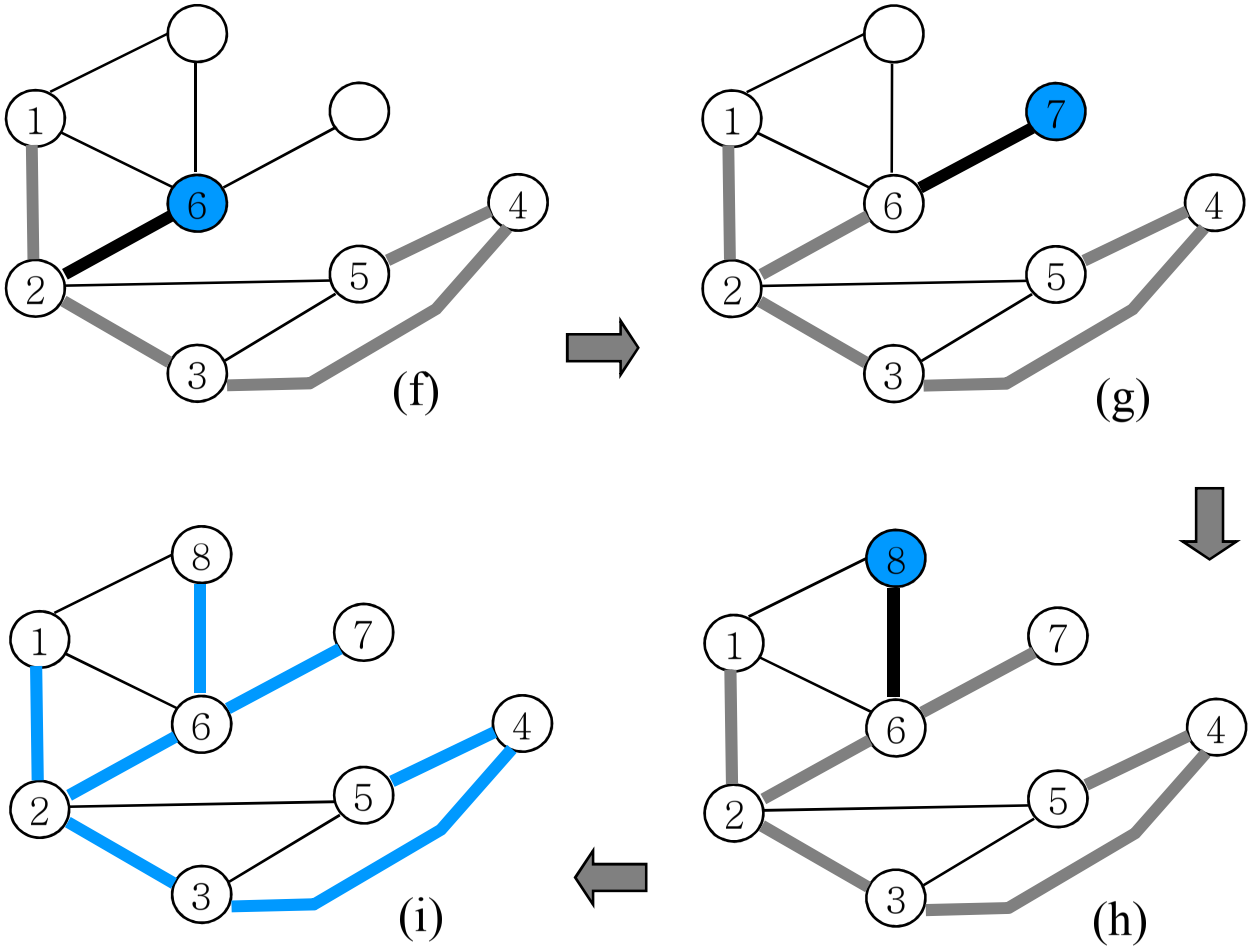
BFS case



DFS case



DFS case (continued)



Minimum Spanning Trees (MST)

- The original graph is undirected connected graph
  - connected graph has edges for all pairs of vertices
- Tree
  - Connected graph with no cycle
  - And a tree with n vertices always has n-1 edges
- A spanning tree of graph *G*
  - A tree with G's vertices and a subset of G's edges
- A MST of *G*
  - A spanning tree of which the sum is minimum

Prim Algorithm

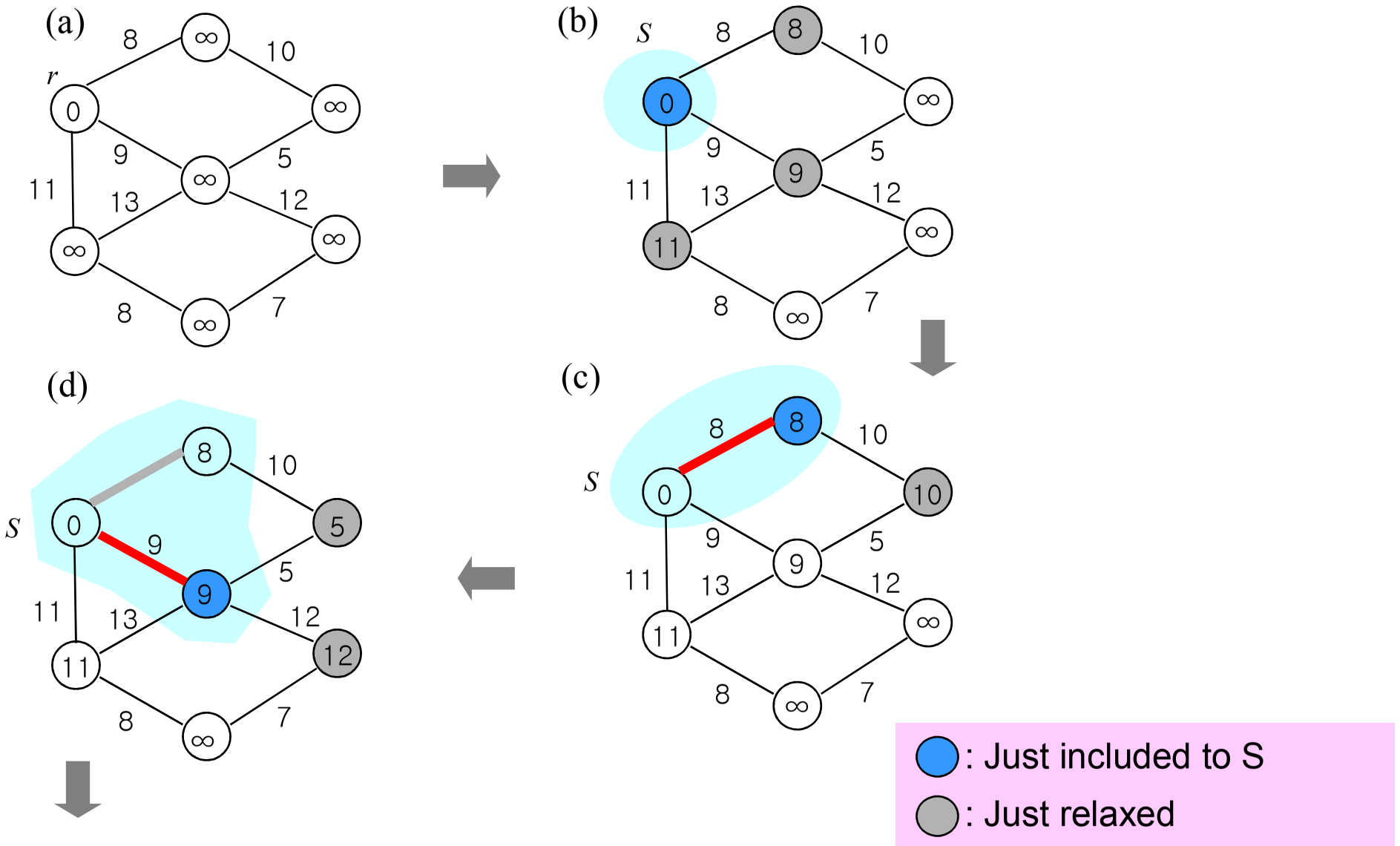
```
Prim (G, r)
{
    S ← Φ ;
    Mark a vertex r as visited and include it to S
    while (S ≠ V) {
        Find a minimum edge (x,y) among the edges connecting vertices of S to
    vertices of V-S ▷ (x∈S, y∈V-S)
        Mark y as visited and include it to S
    }
}
```

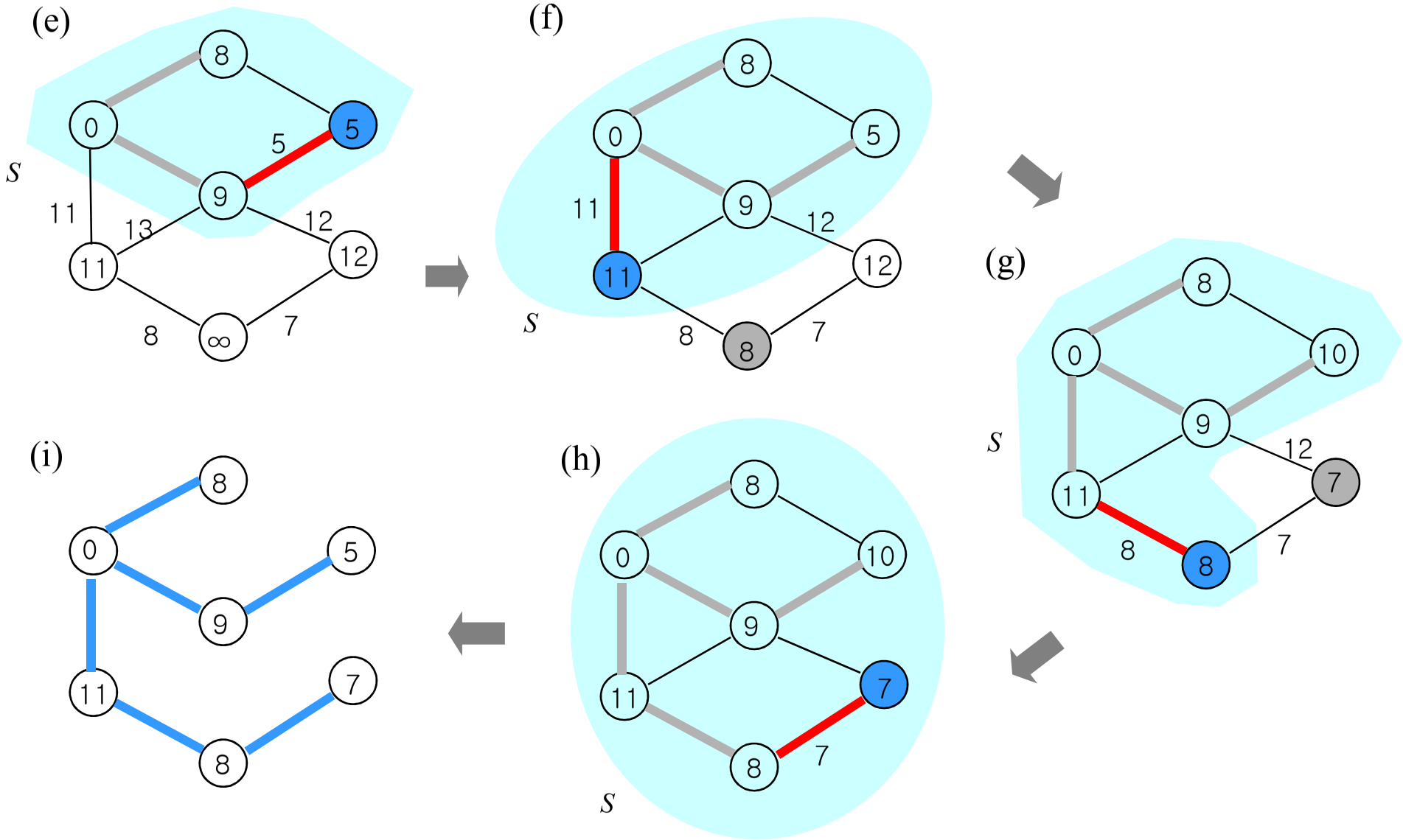
- ✓ Prim Algorithm is a greedy algorithm
- ✓ One rare example of greedy algorithms that guarantees an optimal solution

✓Running Time: $O(|E|\log|V|)$

Using Heap

Prim Algorithm's Example



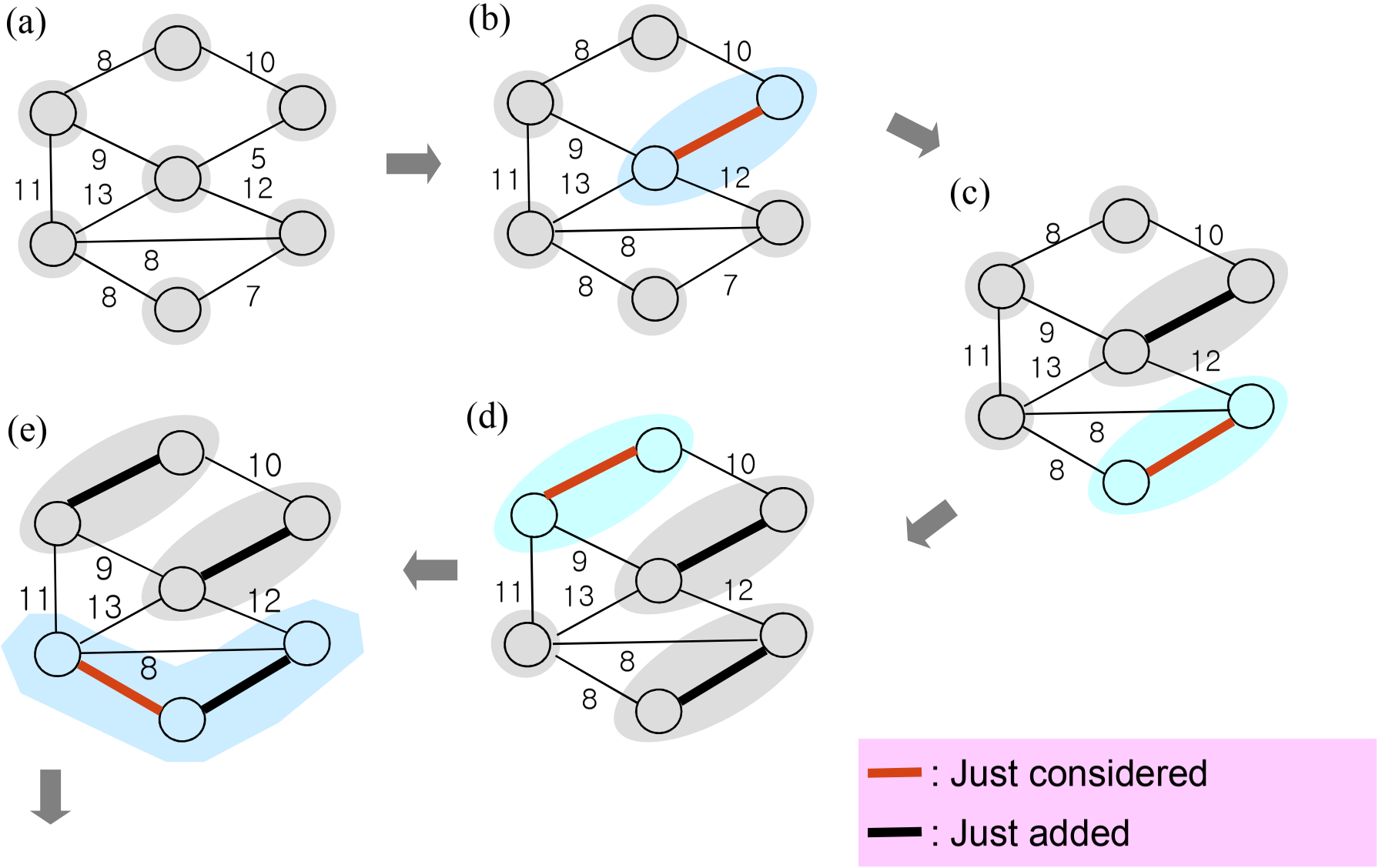

(e)
(f)
(g)
(h)
(i)
S
S
S
S
8
0
5
9
11
12
13
11
12
∞
8
7
5
11
8
8
7
8
0
10
9
12
11
7
8
7
8
0
10
9
11
7
8
7
8
0
5
9
11
7
8

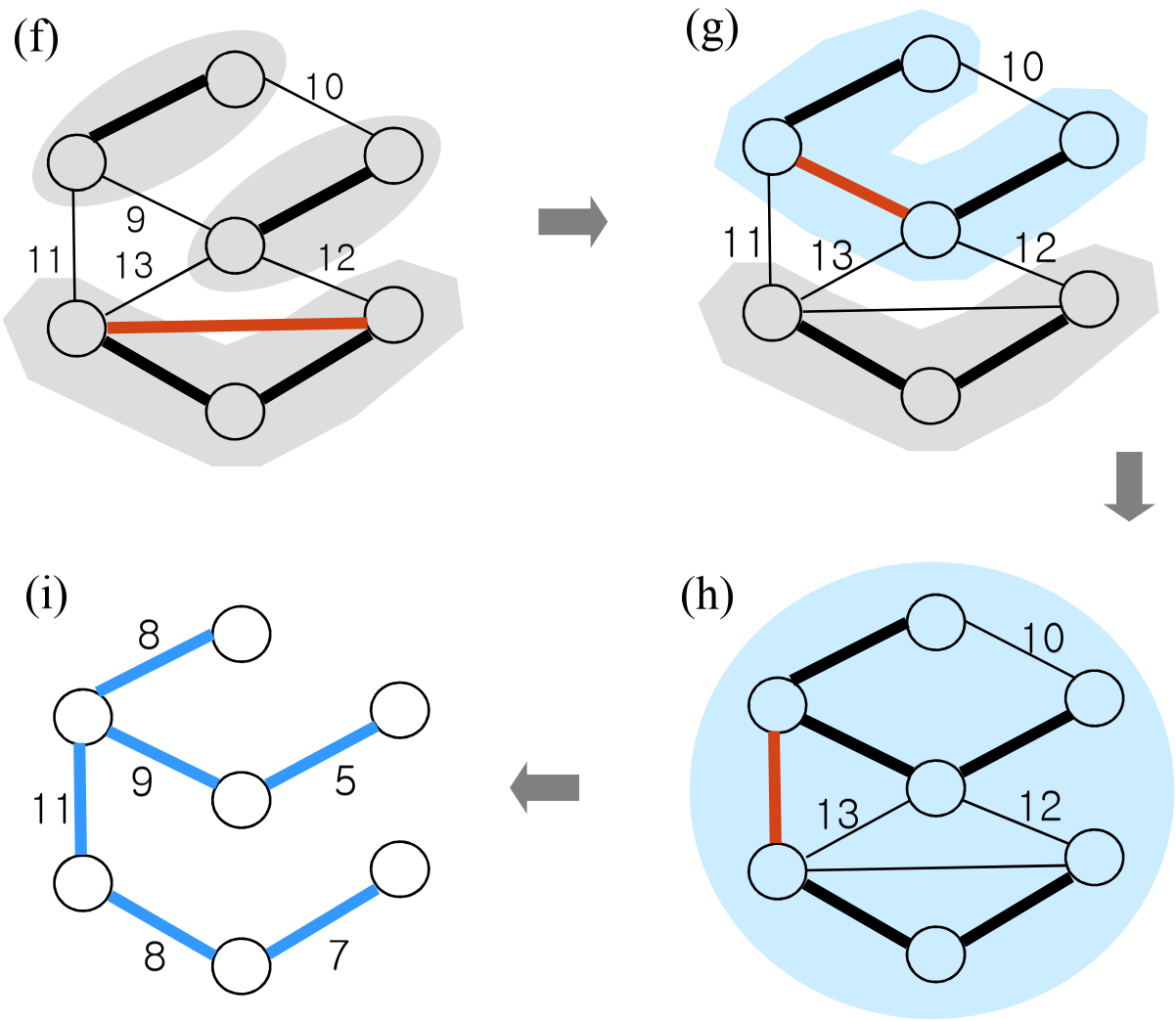
Kruskal Algorithm

```
Kruskal (G, r)
{
    T ← Φ ; ▷ T : Spanning tree
    Initialize n sets with one vertex
    Sort all edges in ascending order of weights
    while (# of edges in T < n-1) {
        Remove an edge (u, v) with minumum weight
        If u and v are from different sets {
            Merge the two sets
            T ← T∪{u, v)};
        }
    }
}
```

✓Running Time: $O(|E|\log|V|)$

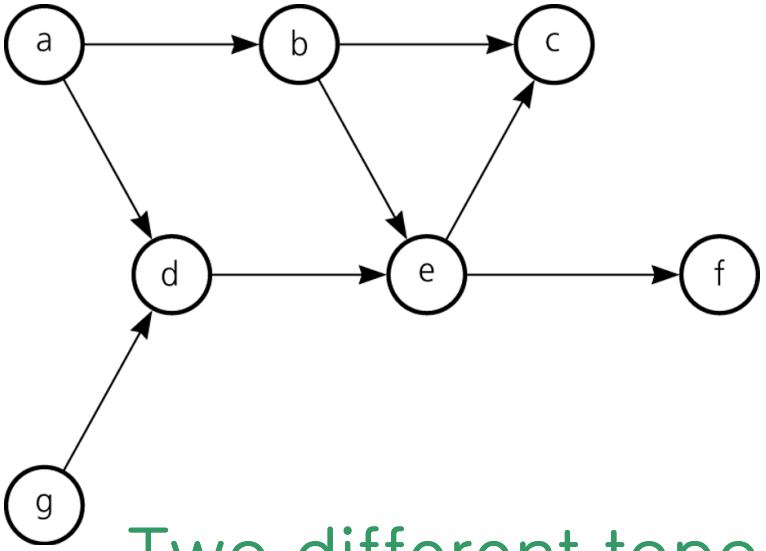
Kruskal Algorithm's Example




(f)
10
9
11
13
12
(g)
10
11
13
12
(h)
10
13
12
(i)
8
9
5
11
8
7

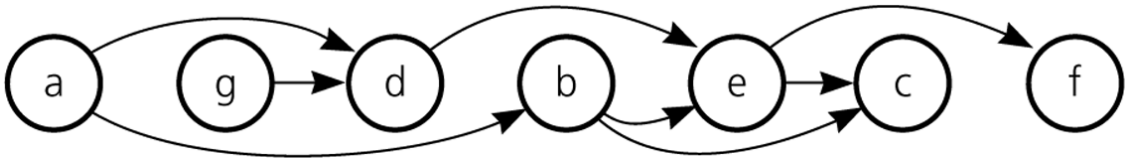
Topological Sort

- For digraphs
- Topological Sort
  - Enumerate all vertices
  - If there is an edge from x to y, enumerate x before y
  - There can be multiple topological orders for one digraph

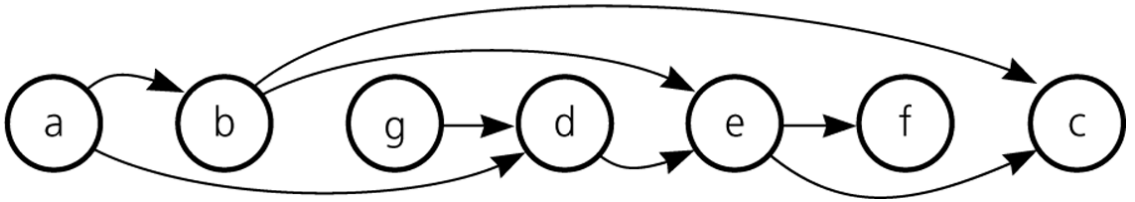


Two different topological orders for the graph

(a)



(b)

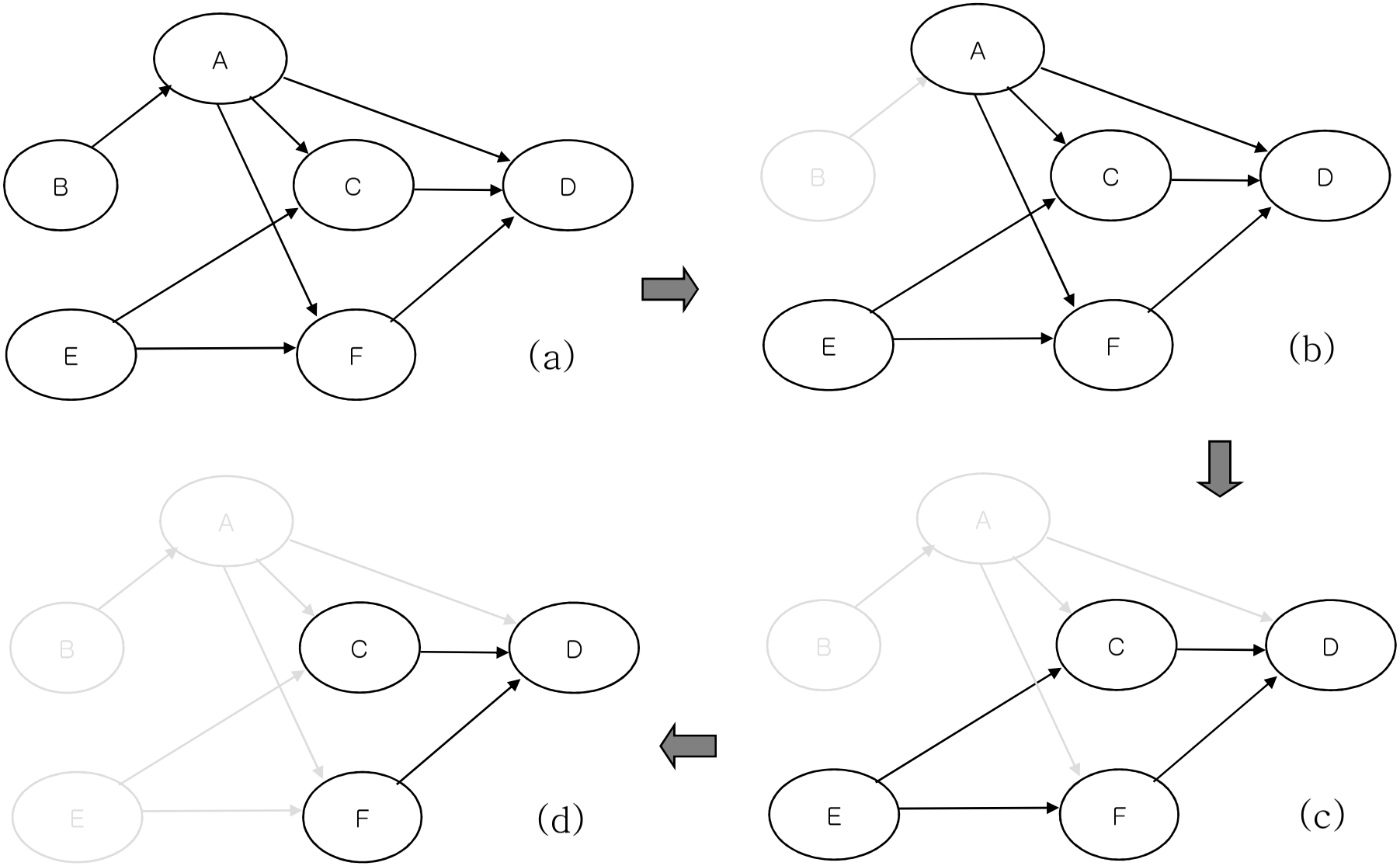


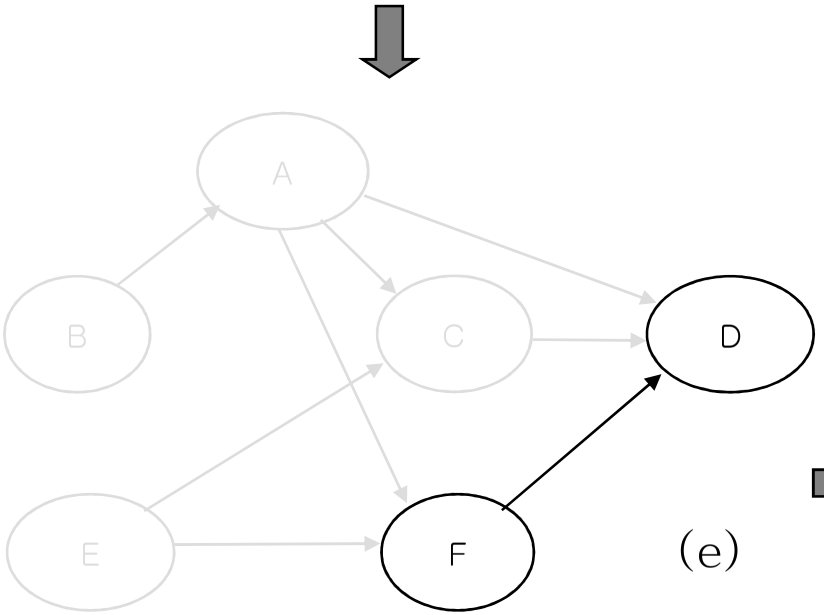
Topological Sort 1

```
topologicalSort1(G)
{
    for ← 1 to n {
        Choose a vertex u with no incoming edge
        A[i] ← u;
        Remove the vertex u and all outgoing edges of u
    }
    ▷ Array A[1…n] has vertices topologically sorted
}
```

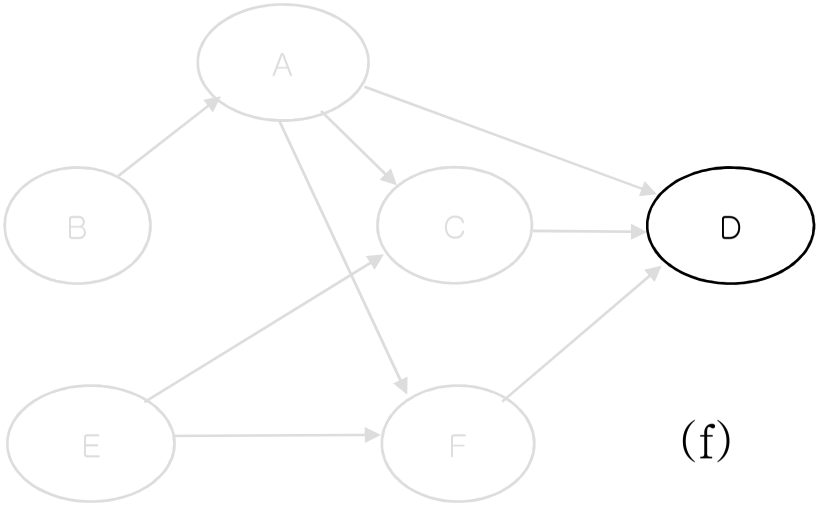
✓Running Time: $\Theta(|V|+|E|)$

Topological Sort 1's Example

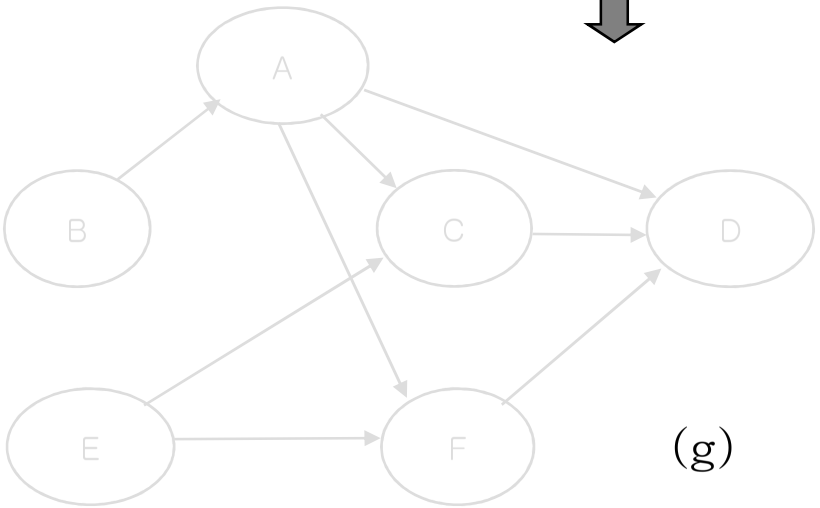




(e)



(f)



(g)

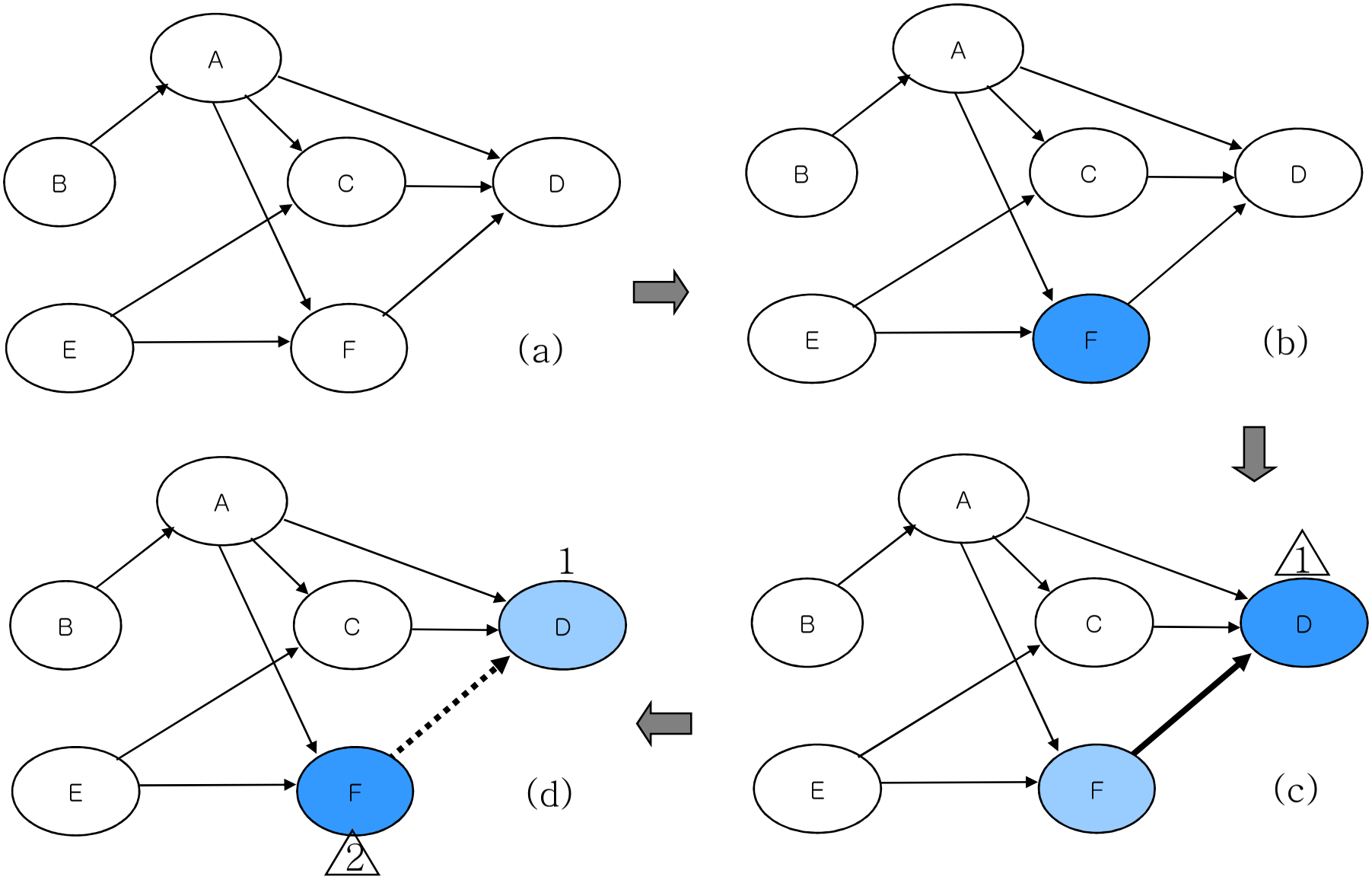
Topological Sort 2

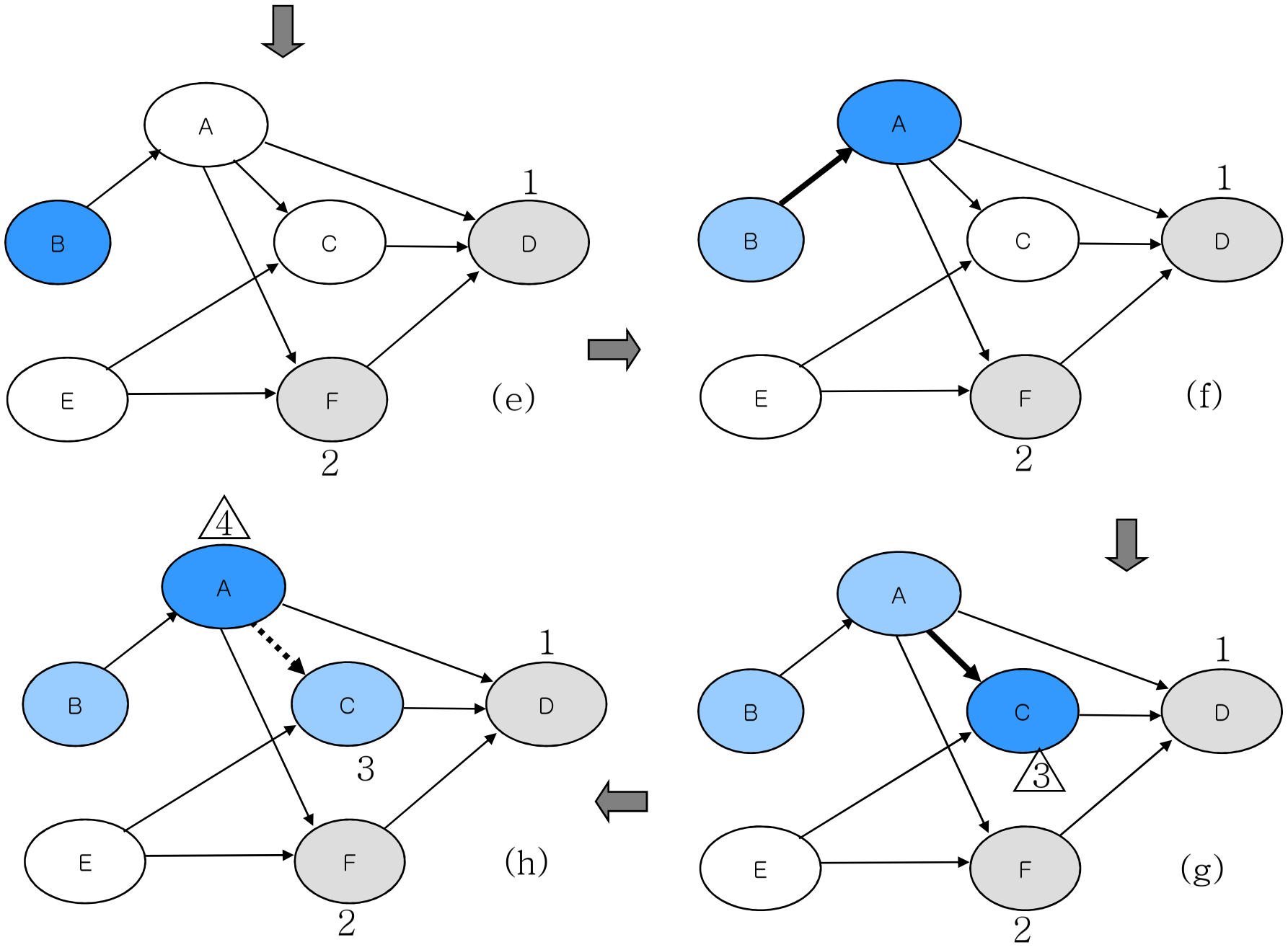
```
topologicalSort2(G)
{
    for each v∈V
        visited[v] ← NO;
    for each v∈V   ▷ The order of the vertices are irrelavant
        if (visited[v] = NO) then DFS-TS(v) ;
}
DFS-TS(v)
{
    visited[v] ← YES;
    for each x∈L(v)   ▷ L(v): v's adjacency list
        if (visited[x] = NO) then DFS-TS(x) ;
    Insert v in front of the list R
}
```

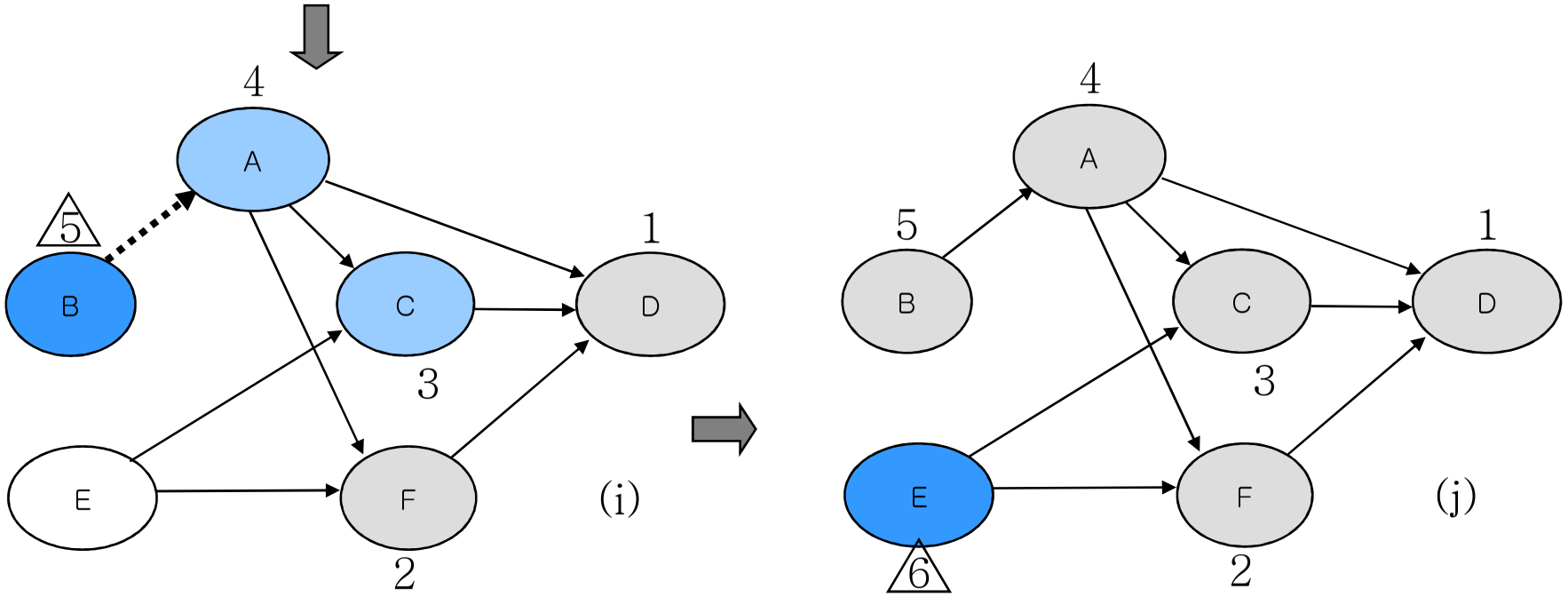
✓Running Time: $\Theta(|V|+|E|)$

✓After the algorithm, the list R has vertices topologically sorted

Topological Sort 2's Example




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


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

Shortest Paths

- For weighted digraph
  - We consider undirected graph as a digraph with two directed edges for one undirected edge
    - Undirected edge (u, v) is directed edges (u, v) and (v, u)
- Shortest path between two vertices
  - is a path of which the sum of edge weights is minimum

- Single source – shortest path
  - Shortest paths from single vertex to all other vertices
  - ➢ Dijkstra Algorithm
    - does not allow negative edges
  - ➢ Bellman-Ford Algorithm
    - allow negative edges
- All source – shortest path
  - Shortest paths for all pairs of vertices
  - ➢ Floyd-Warshall Algorithm

Dijkstra Algorithm

No negative edges

```
Dijkstra(G, r)
▷ G=(V, E): Graph
▷ r: Starting vertex
{
      S ← Φ ;                    ▷ S : A set of vertices
      for each u∈V
            d_u ← ∞ ;
      d_r ← 0 ;
      while (S≠V){               ▷ n times
            u ← extractMin(V-S, d) ;
            S ← S ∪ {u};
            for each v∈L(u) ▷ L(u) : A set of vertices connected from u
                  if (v∈V-S and d_v<d_u+w_u,v) then d_v ← d_u+w_u,v ;
      }
}
```

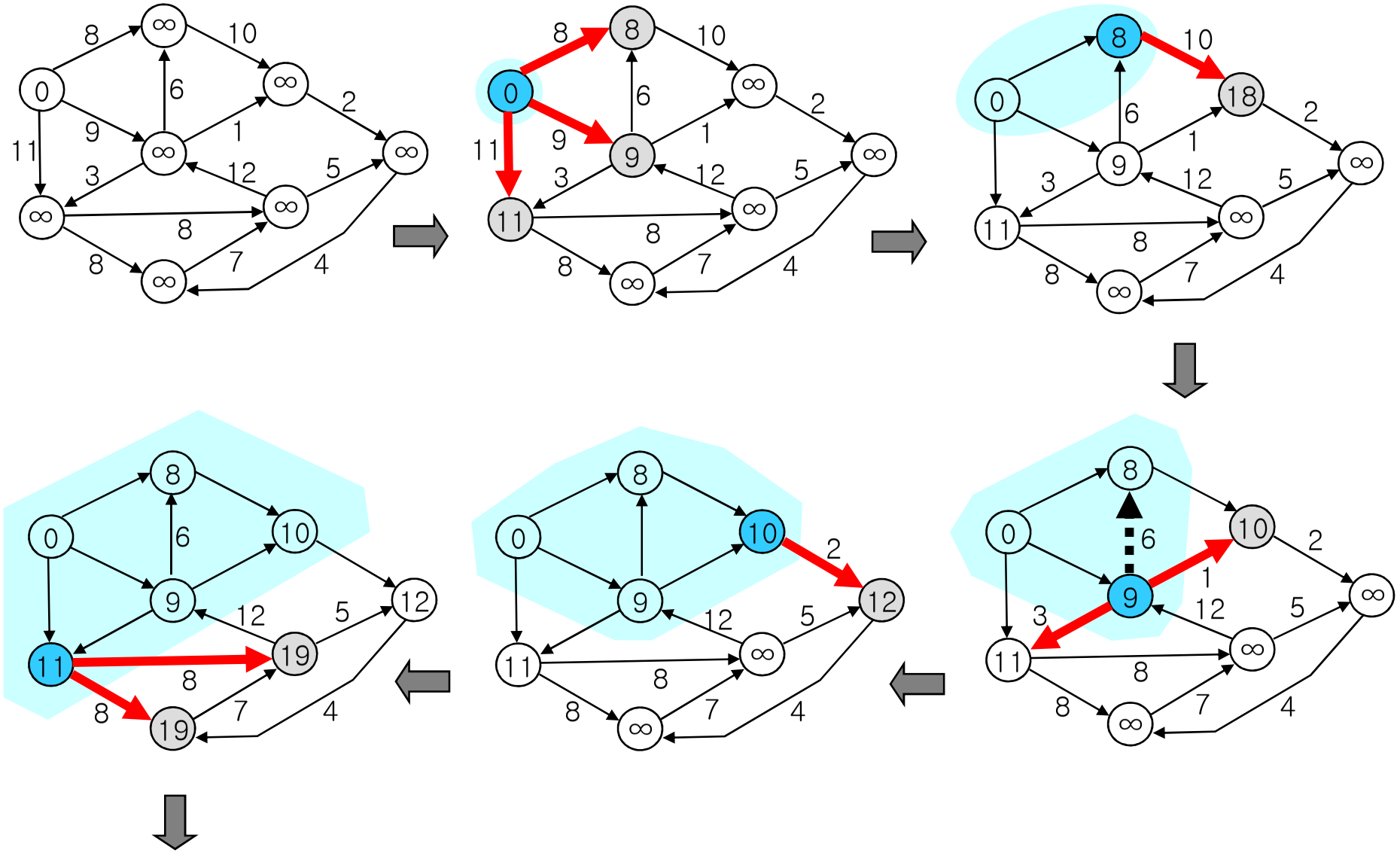
relaxation

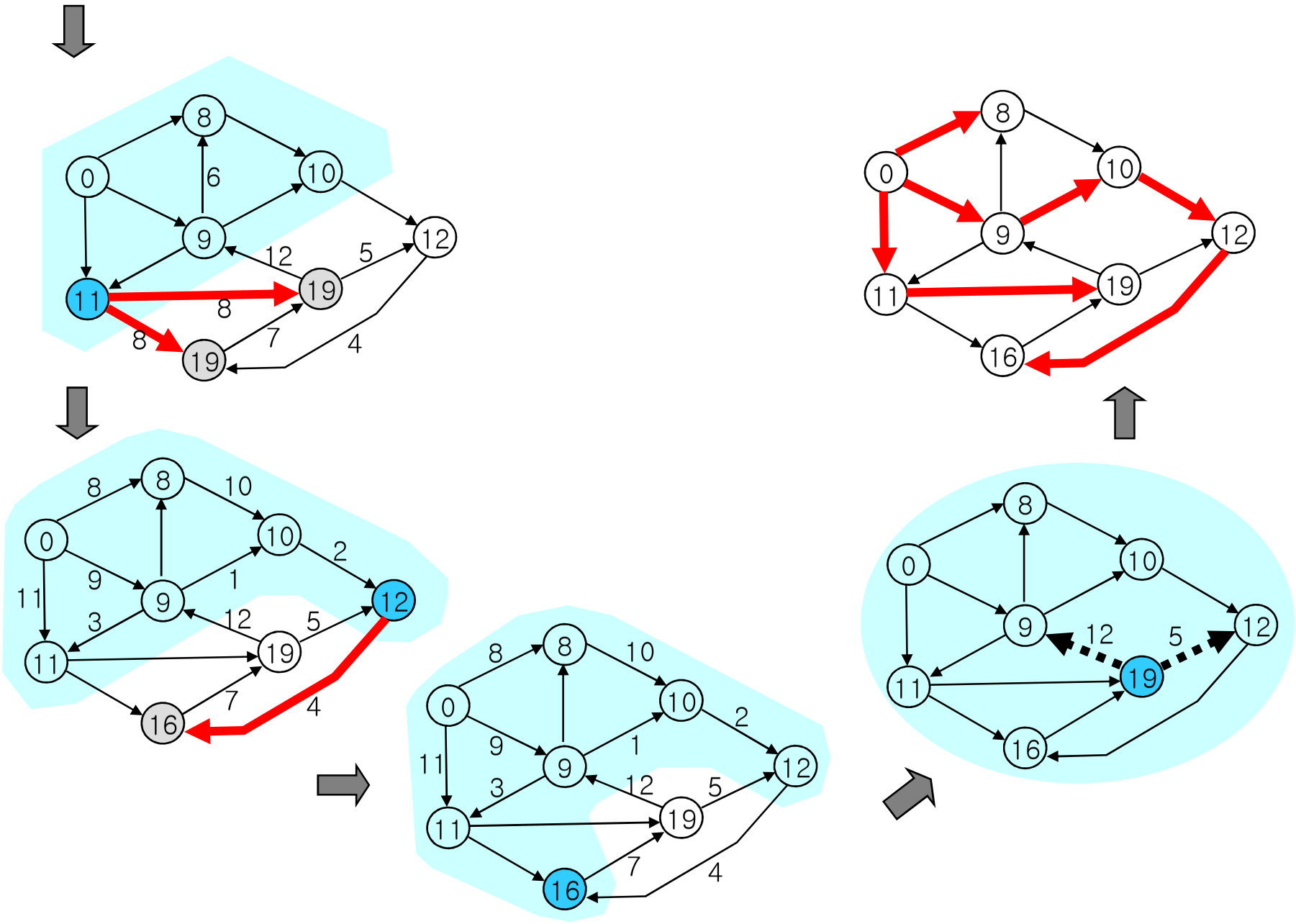
```
extractMin(Q, d)
{
      Return the vertex with smallest d in the set Q
}
```

✓Running Time: $O(|E|\log|V|)$

Using Heap

Dijkstra Algorithm's Example




0
8
10
6
9
12
5
11
19
8
8
19
7
4
16

Bellman-Ford Algorithm

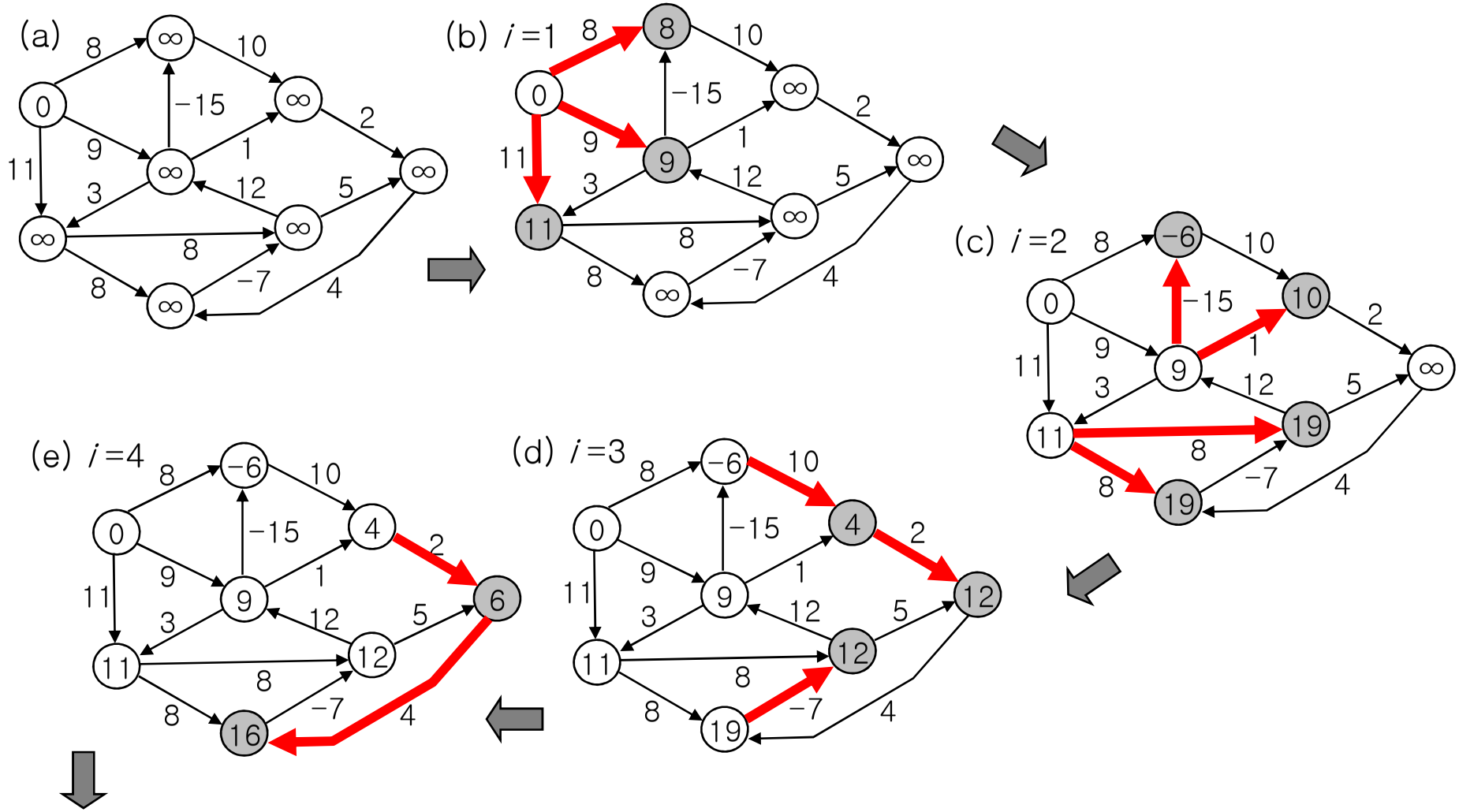
Allow negative edges

```
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} ;
}
```

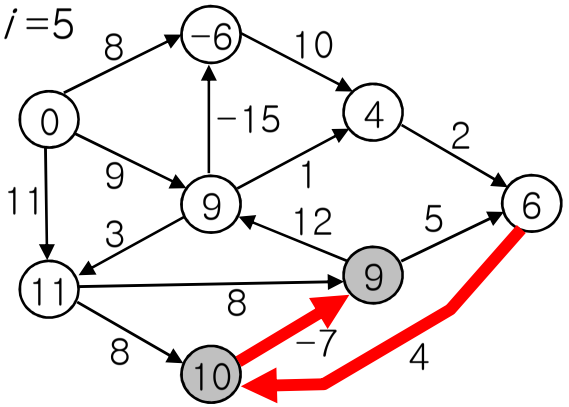
relaxation

✓Running Time: $\Theta(|E||V|)$

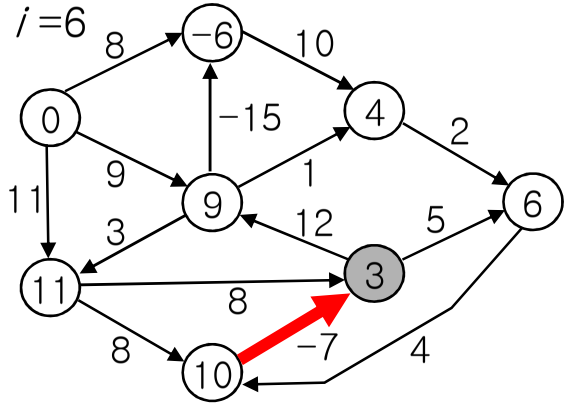
Bellman-Ford Algorithm's Example



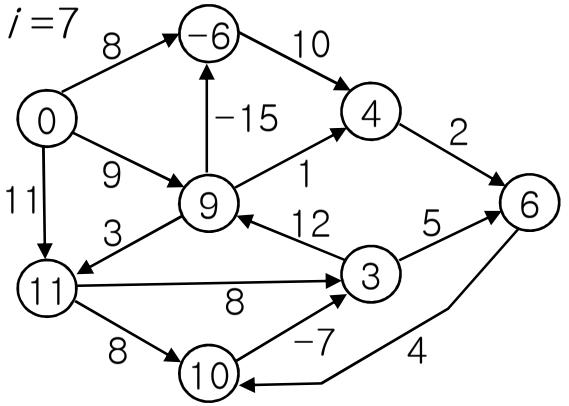
(f) $i=5$


8
−6
10
0
−15
4
2
9
1
11
9
12
5
6
3
11
9
8
8
−7
10
4

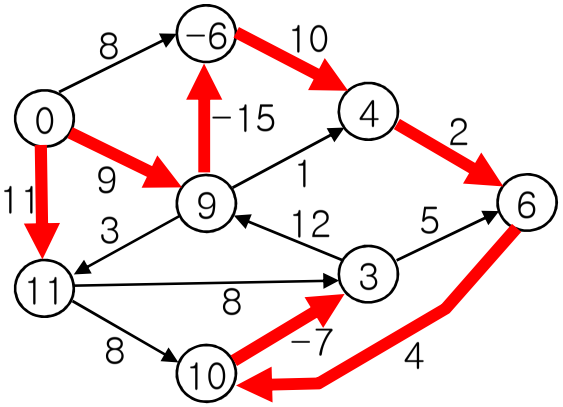
(g) $i=6$


8
−6
10
0
−15
4
2
9
1
11
9
12
5
6
3
11
3
8
8
−7
10
4

(h) $i=7$


8
−6
10
0
−15
4
2
9
1
11
9
12
5
6
3
11
3
8
8
−7
10
4

(i)


8
−6
10
0
−15
4
2
9
1
11
9
12
5
6
3
11
3
8
8
−7
10
4

Bellman-Ford Algorithm in terms of DP

- d_t^k : Shortest distance from *r* to *t* through at most *k* edges
- Objective: d_t^{n-1}

✓ Recurrence Relation

$$\begin{cases} d_v^k = \min_{\text{for all edges } (u, v)} \{d_u^{k-1} + w_{u,v}\}, & k > 0 \\ d_r^0 = 0 \\ d_t^0 = \infty, \quad t \neq r \end{cases}$$

Floyd-Warshall Algorithm

- Shortest distance for all possible pairs of vertices
- Application
  - Road Atlas
  - Navigation System
  - Network Communication

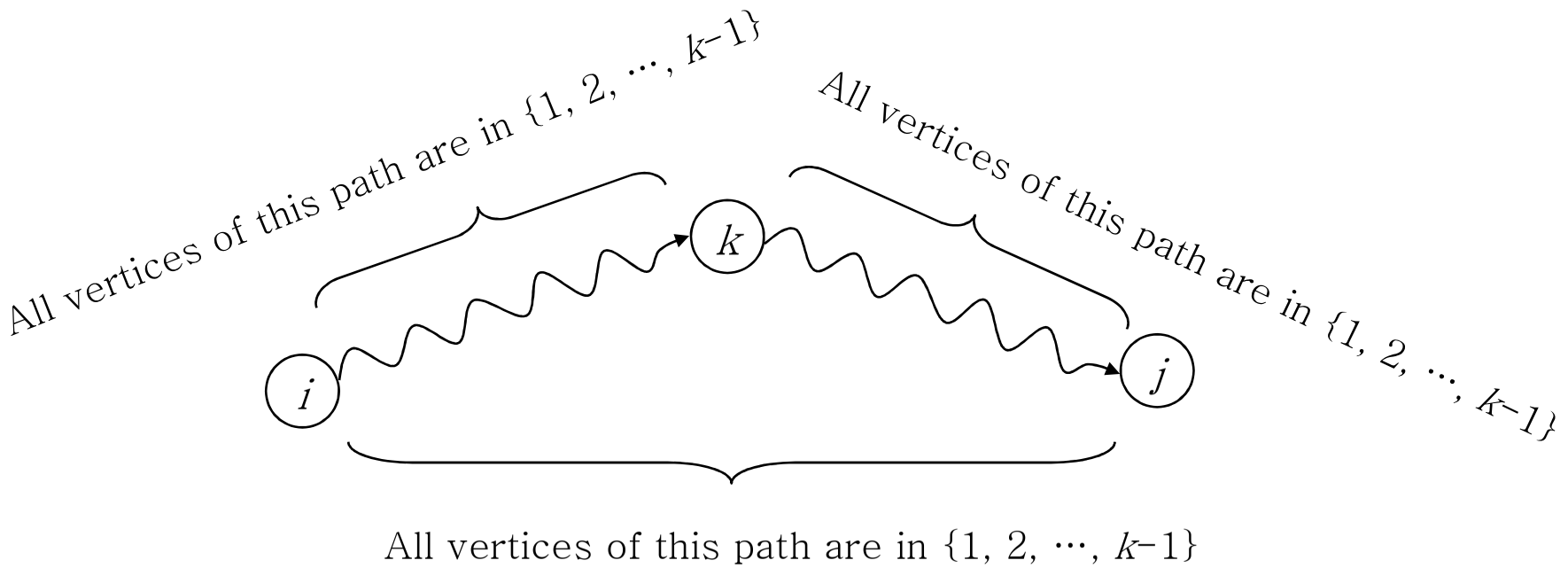
Floyd-Warshall Algorithm

```
FloydWarshall(G)
{
    for i ← 1 to n
        for j ← 1 to n
            d^0_ij ← w_ij ;
    for k ← 1 to n                  ▷ Intermediate vertices {1, 2, …, k}
        for i ← 1 to n              ▷ i : Starting vertex
            for j ← 1 to n          ▷ j : Ending vertex
                d^k_ij ← min {d^(k-1)_ij , d^(k-1)_ik + d^(k-1)_kj};
}
```

✓ d^k_{ij} : Shortest path from i to j through vertices $\{1, 2, \ldots, k\}$

✓Running Time: $\Theta(|V|^3)$

d^{k}_{ij}


All vertices of this path are in {1, 2, ⋯, k−1}
All vertices of this path are in {1, 2, ⋯, k−1}
k
i
j
All vertices of this path are in {1, 2, ⋯, k−1}

Shortest Path of Graph with no cycles

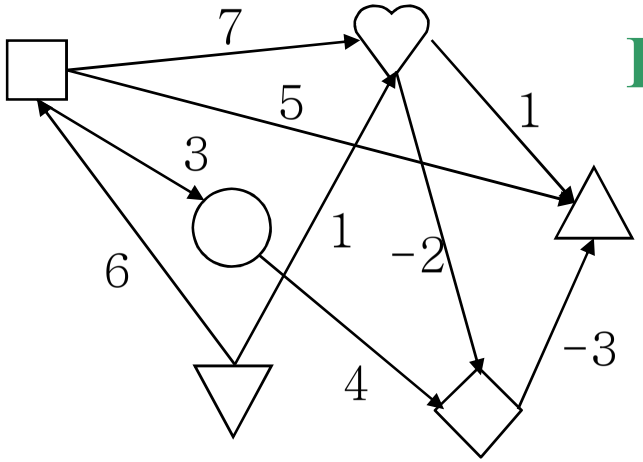
- Directed graph with no cycle = DAG
  - DAG: Directed Acyclic Graph
- The shortest path in DAG can be obtained in linear time

```
DAG-ShortestPath(G, r)
{
    for each u∈V
            d_u ← ∞;
    d_r ← 0;
    Topological sort G's vertices
    for each u∈V (in topological order)
        for each v∈L(u) ▷ L(u) : Adjacent vertices of u
            if (d_u + w_{u,v} < d_v) then d_v ← d_u + w_{u,v} ;
}
```

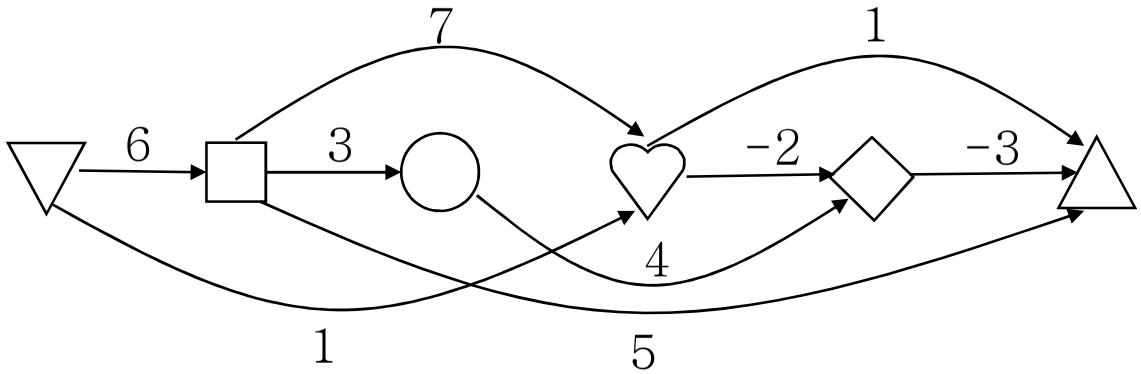
✓Running Time: $\Theta(|V|+|E|)$

DAG-ShortestPath's Example

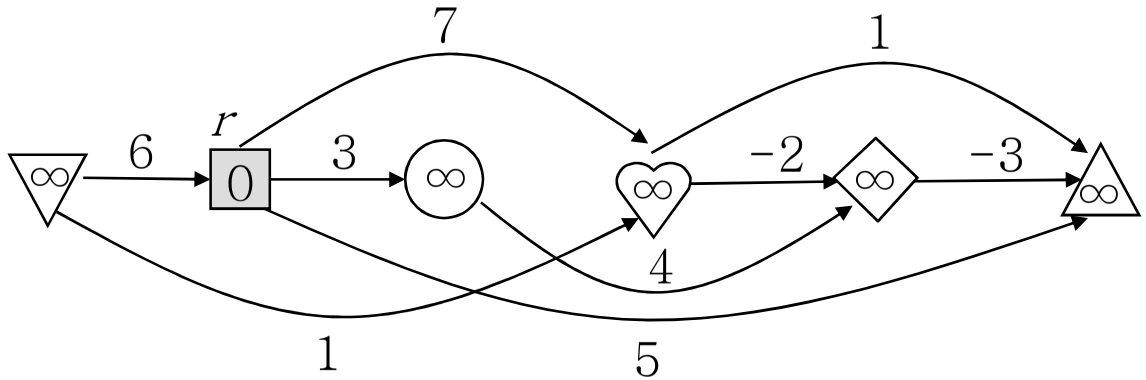
(a)



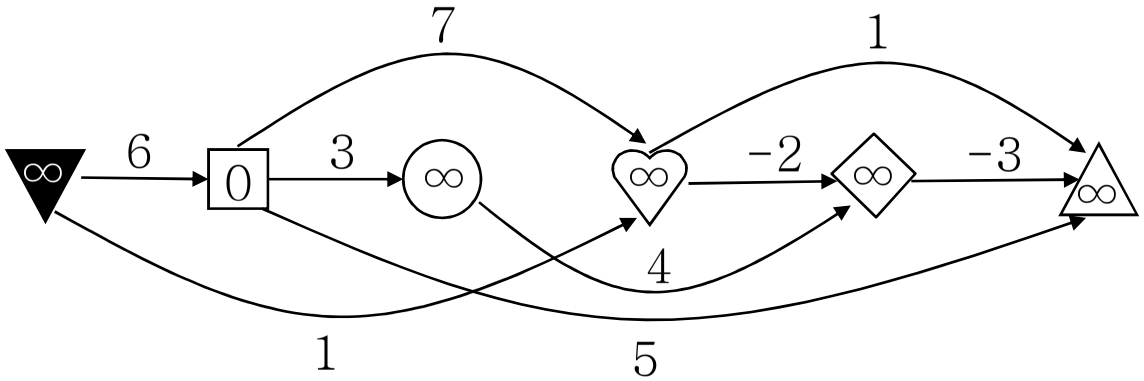
(b)



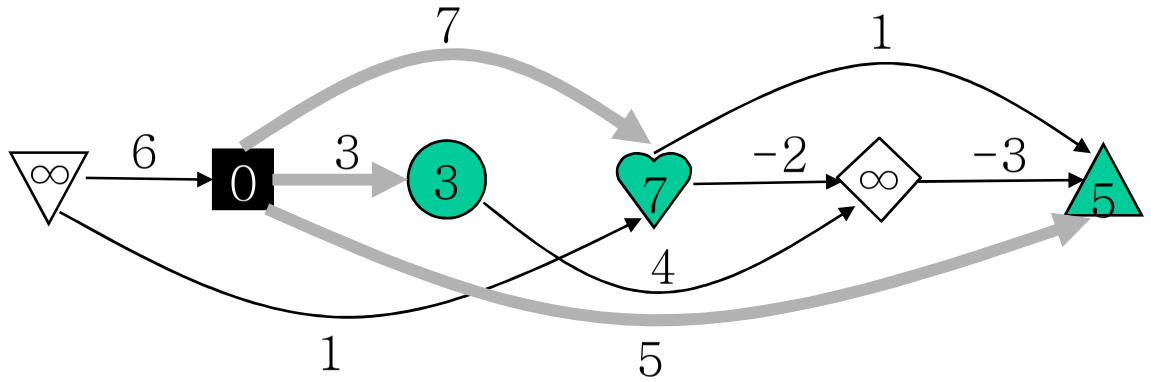
(c)



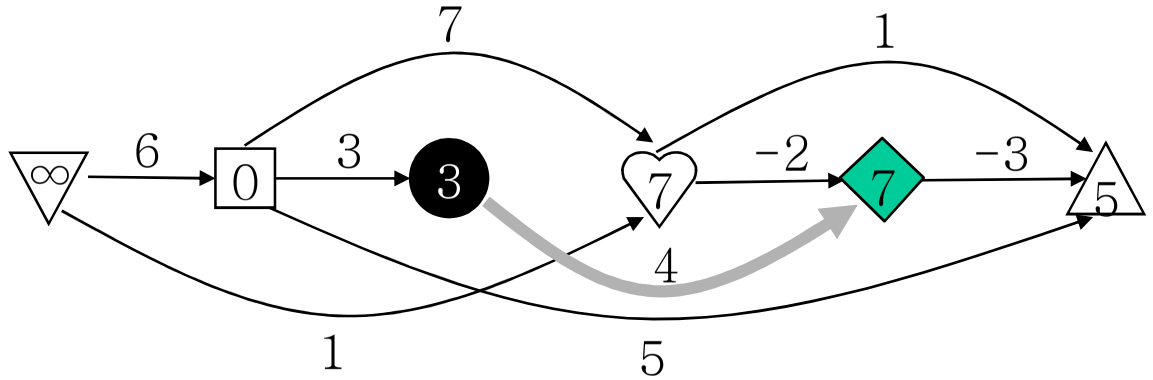
(d)



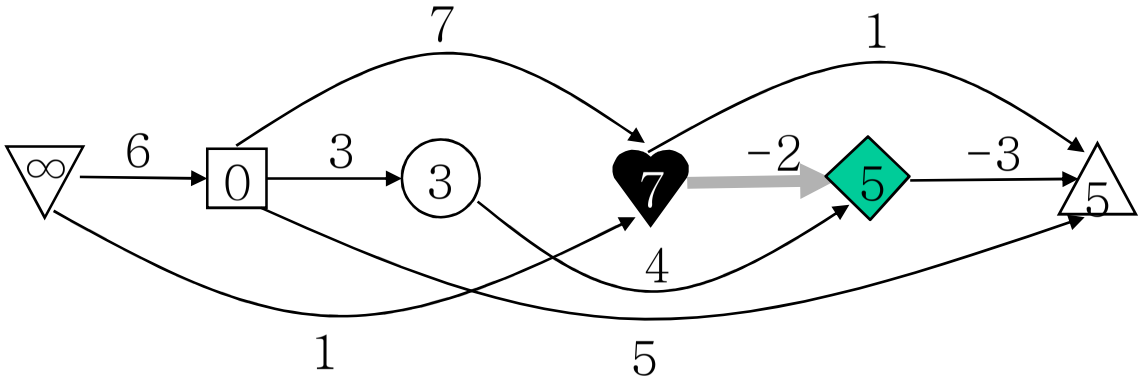
(e)



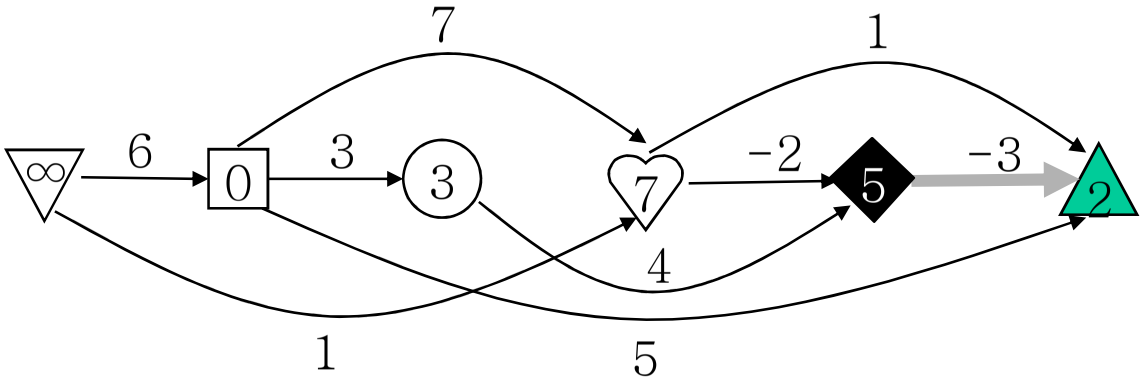
(f)



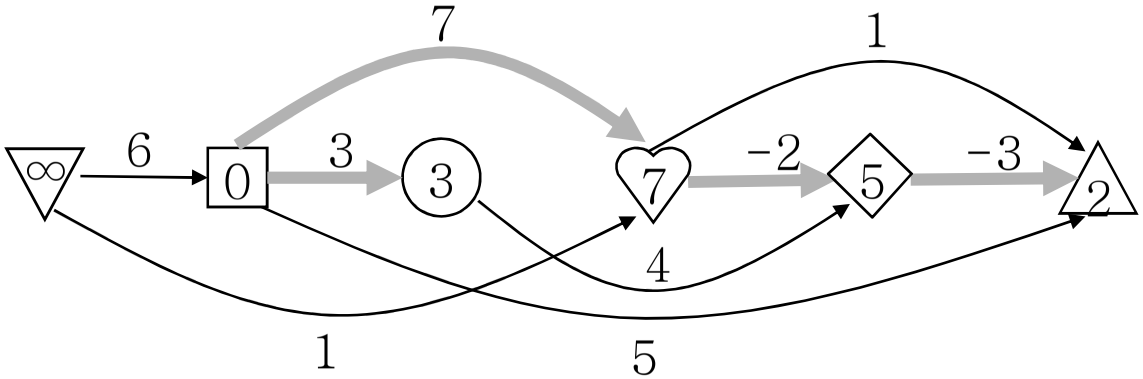
(g)



(h)



(i)



Strongly Connected Components

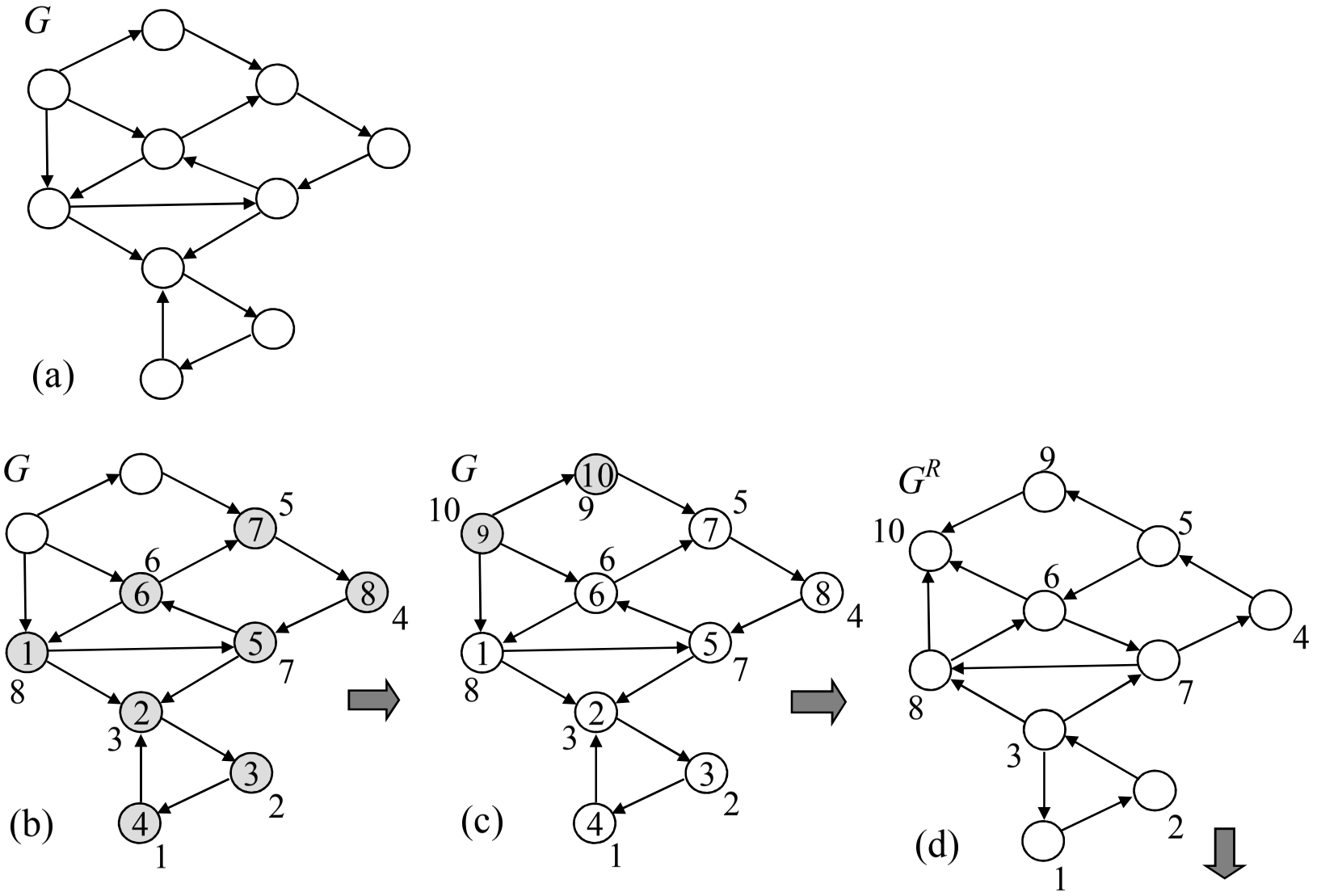
- Strongly Connected
  - For all pairs of vertices, there are bidirectional paths, the graph is strongly connected
  - If a subgraph is strongly connected, it is strongly connected component of the graph

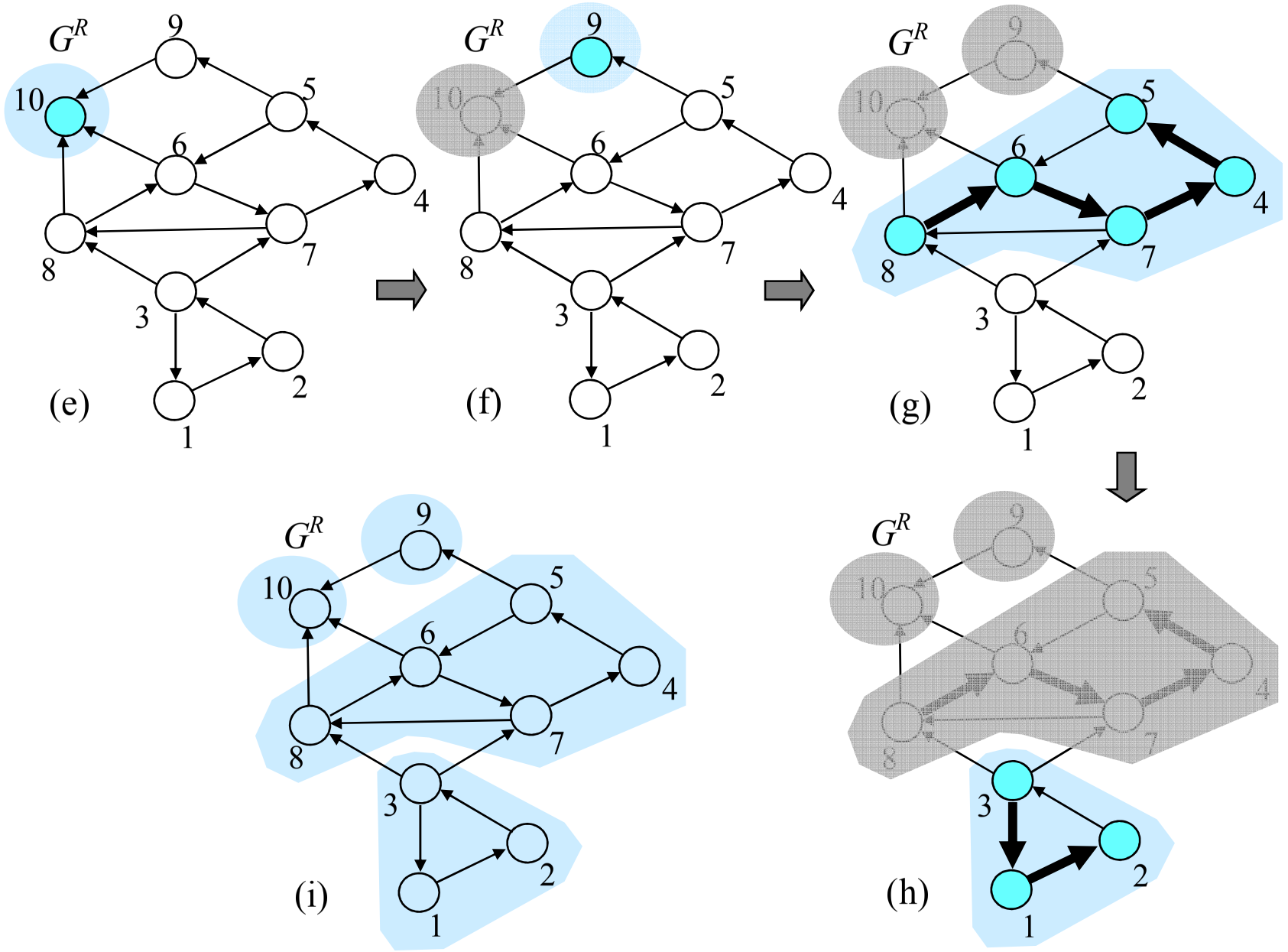
SCCs Algorithm

stronglyConnectedComponent(G)
{
1. Run DFS on the graph. Record the finish time f_v for each vertex v
2. Make G^R
3. Run DFS on G^R. The starting point is a vertex of which f_v is maximum
4. Return the resulting forest as strongly connected components
}

✓Running Time: $\Theta(|V|+|E|)$

stronglyConnectedComponent's Example




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