Ch. 15 Graphs

- Graph traversals
  - Depth-first search
  - Breadth-first search
- Applications of graphs
  - Topological sorting
  - Spanning trees
  - Shortest paths

Definitions

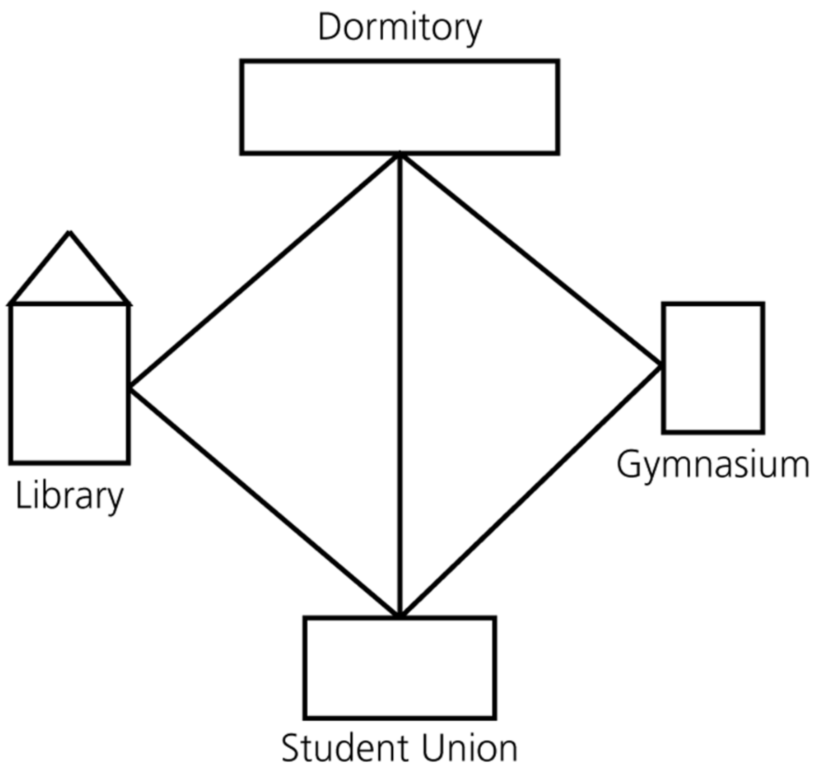
- Graph $G = (V, E)$

  where

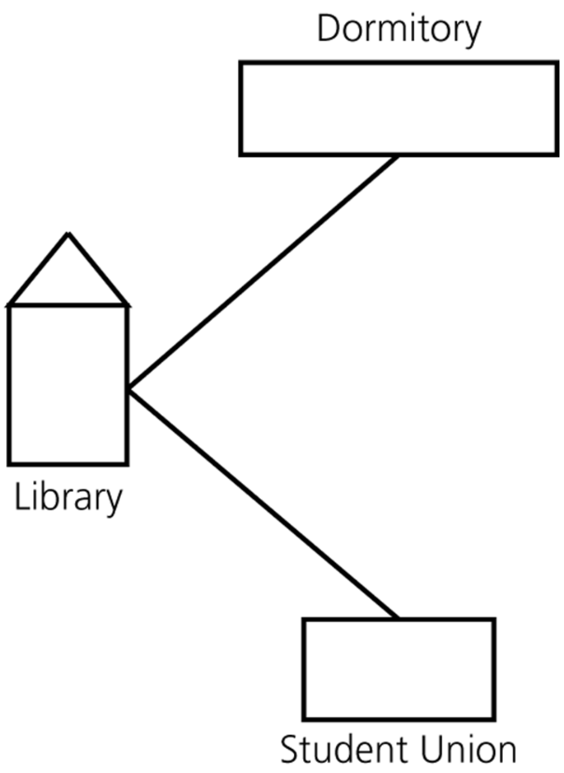
  V is the set of vertices (nodes) and E is the set of edges
- Subgraph
  - A subset of a graph's vertices and edges
- Two vertices are adjacent if they are joined by an edge

- Path
  - A seq. of connected edges
- Cycle
  - A path whose starting vertex and ending vertex are the same
- Simple path
  - A path that contains no cycle
- Simple cycle
  - A cycle that contains no cycle in it

- Connected graph
  - Every pair of distinct vertices has a path bet’n them
- Complete graph
  - Every pair of distinct vertices has an edge bet’n them
- Directed graph (digraph) : undirected graph
  - Whether edges have direction ?
- Weighted graph : unweighted graph
  - Whether edges have weights ?
- Adjacency, multigraph, loop

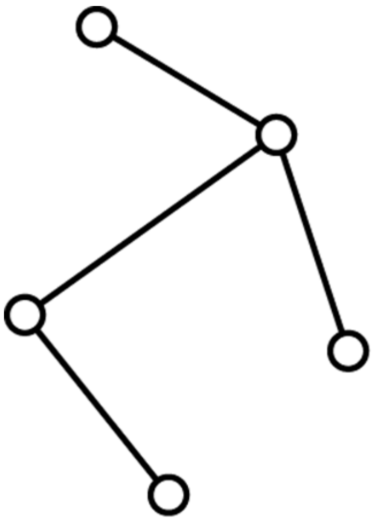


A campus map as a graph

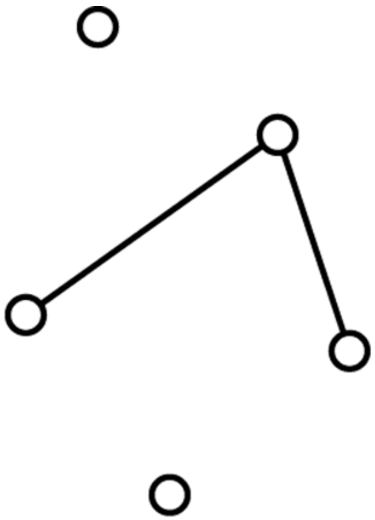


A subgraph

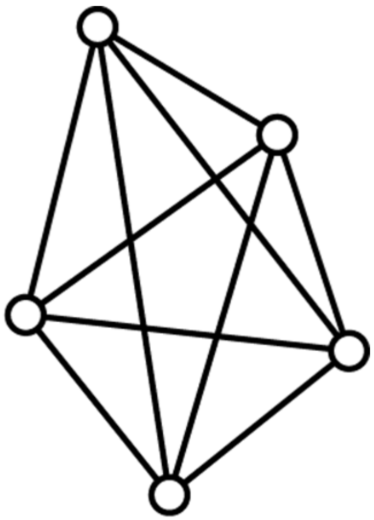
Graphs that are



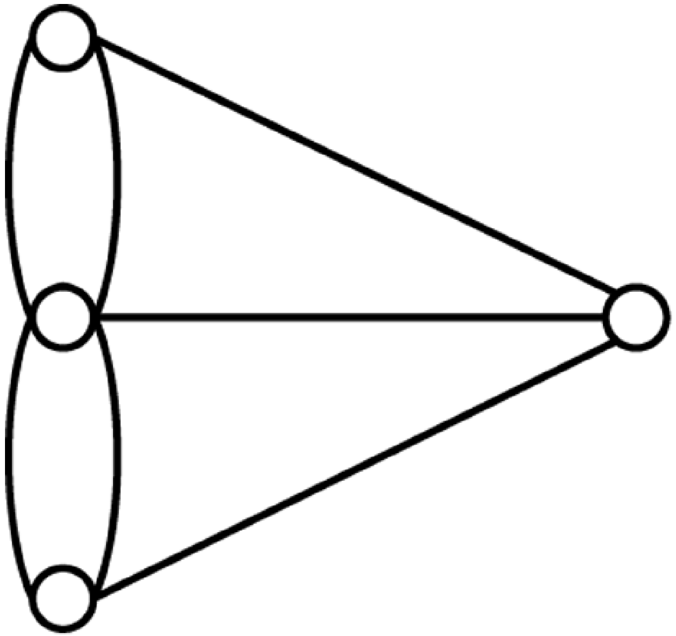
Connected



Disconnected



Complete

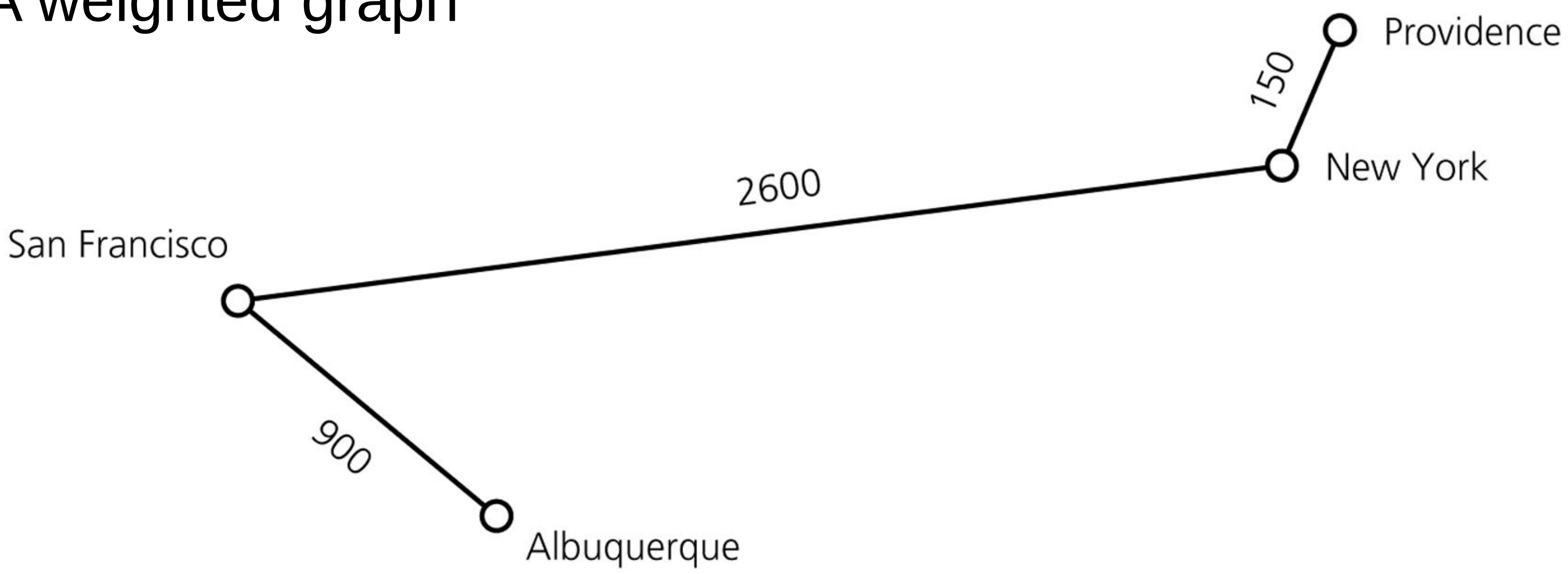


Multigraph

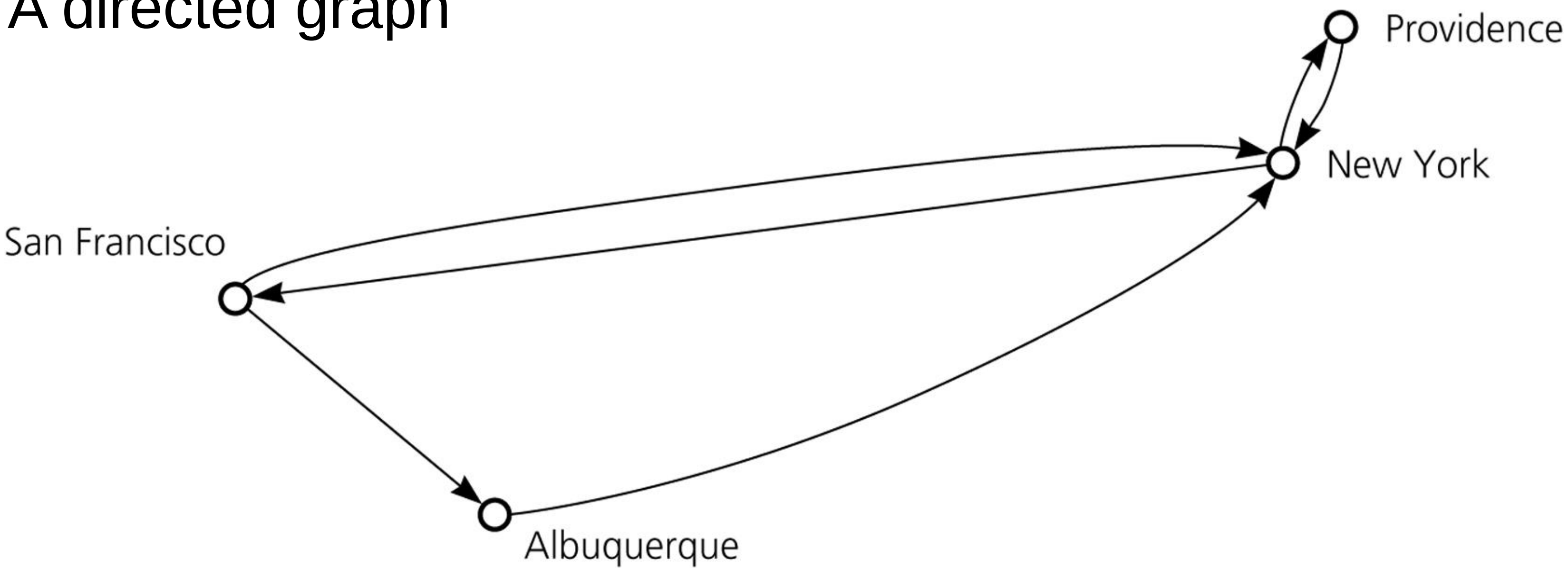
Loop

- ✓ Usually these are not considered as graphs.
- ✓ They are called as a multigraph and a graph w/ loops.

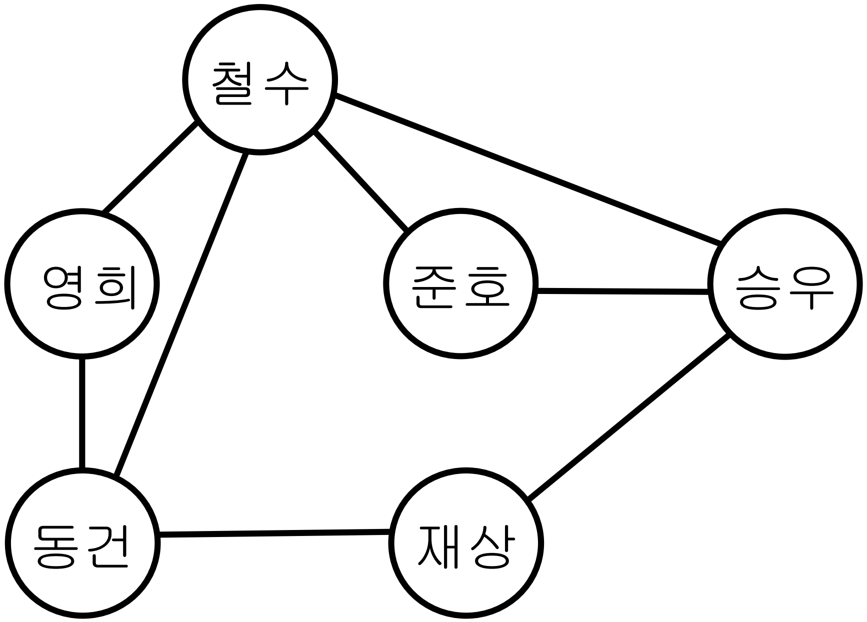
A weighted graph



A directed graph

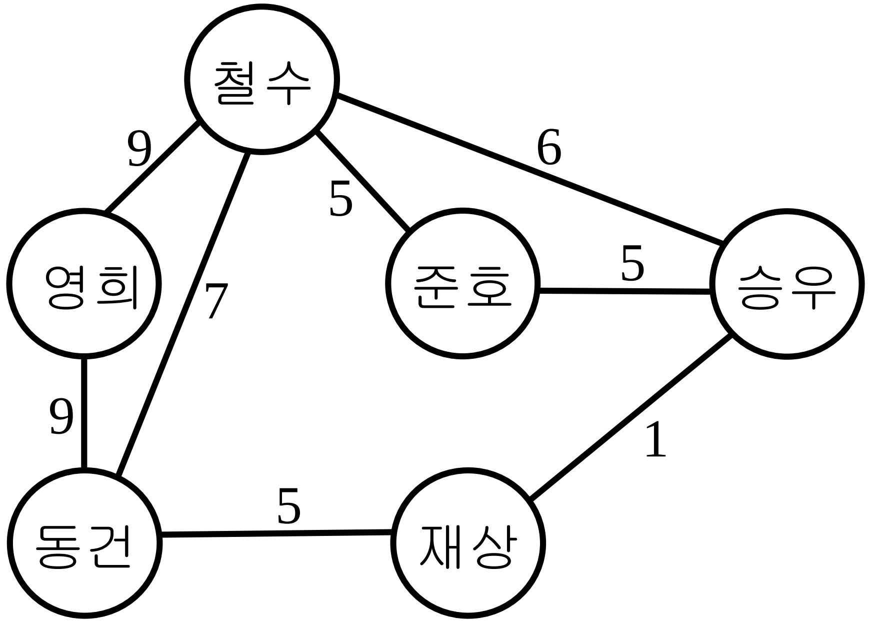


Graph의 예



사람들간의 친분 관계를 나타낸 그래프

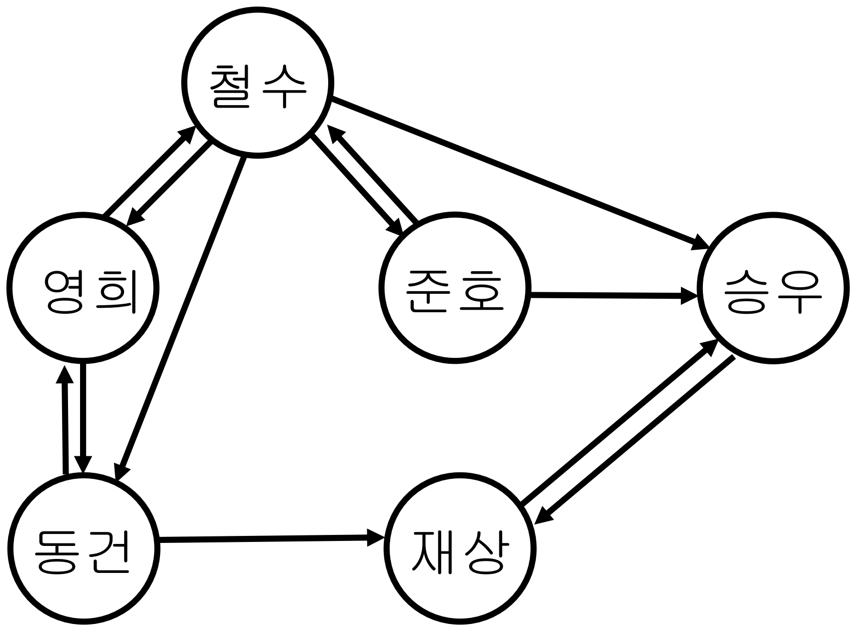
그래프의 예



친밀도를 가중치로 나타낸 친분관계 그래프

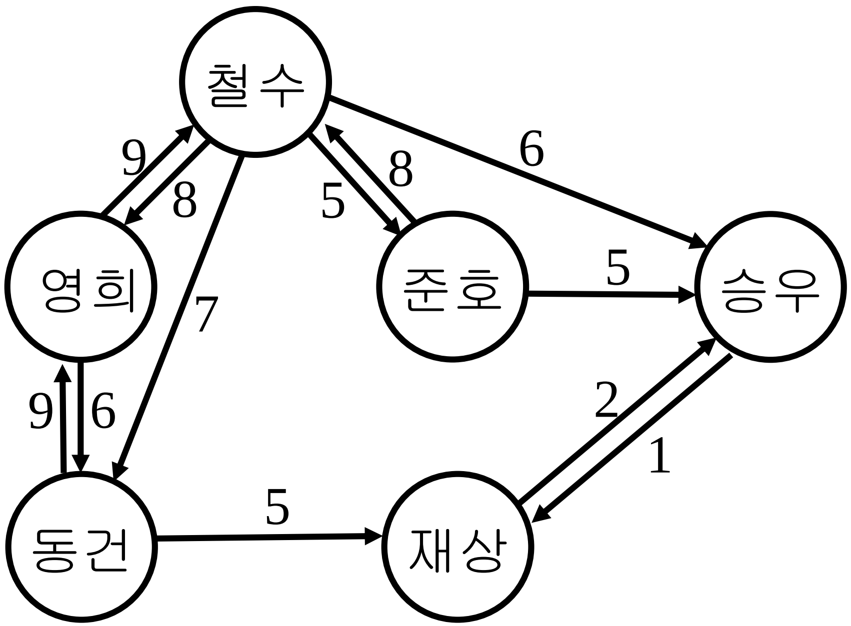
그래프의 예

Directed Graph (Digraph, 유향 그래프)



방향을 고려한 친분관계 그래프

그래프의 예



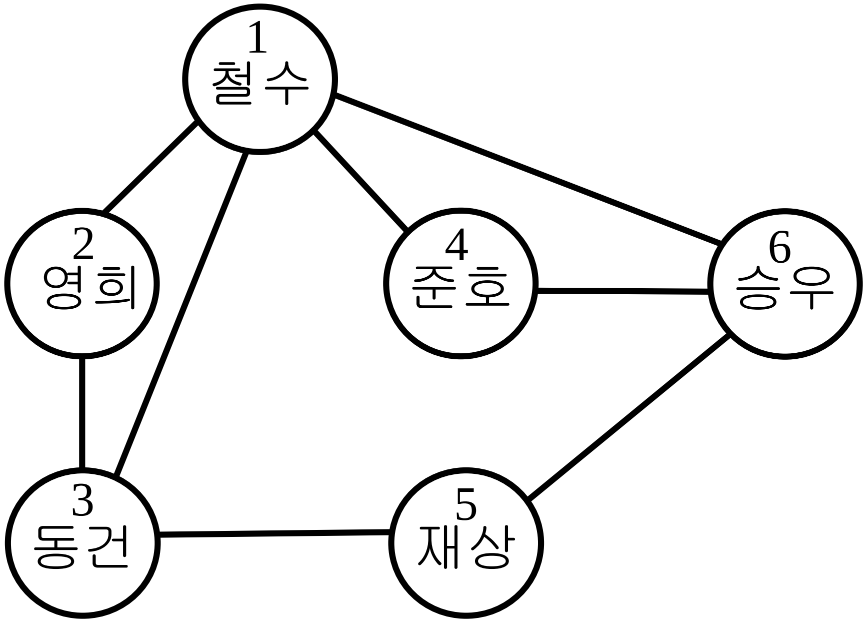
가중치를 가진 Digraph

Graph Representation

N: # of vetices

- Adjacency matrix
  - A graph is represented by an $N \times N$ matrix where *matrix*[*i*][*j*] is 1 if there is an edge bet'n vertex *i* and vertex *j*, and 0 otherwise.
  - In case of a digraph, *matrix*[*i*][*j*] is 1 if there is an edge from vertex *i* to vertex *j*.
  - In case of weighted graph, *matrix*[*i*][*j*] has the weight value instead of 1.

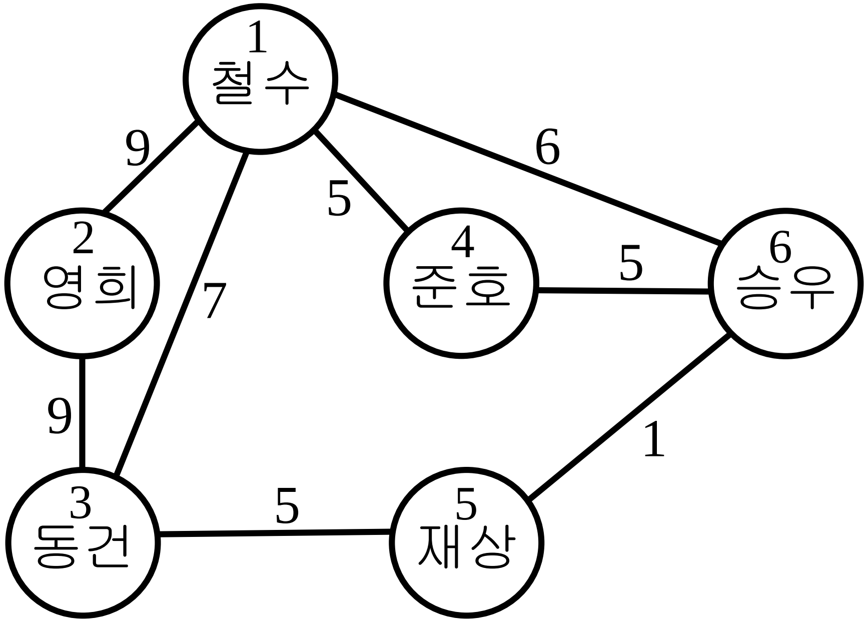
Adjacency Matrix



	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 Graph의 예

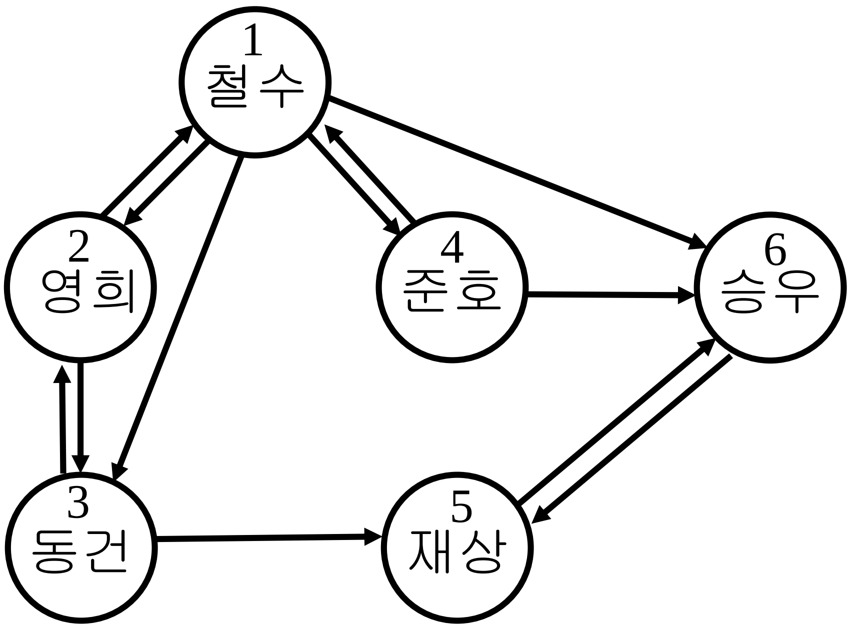
Adjacency Matrix



	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 Undirected Graph의 예

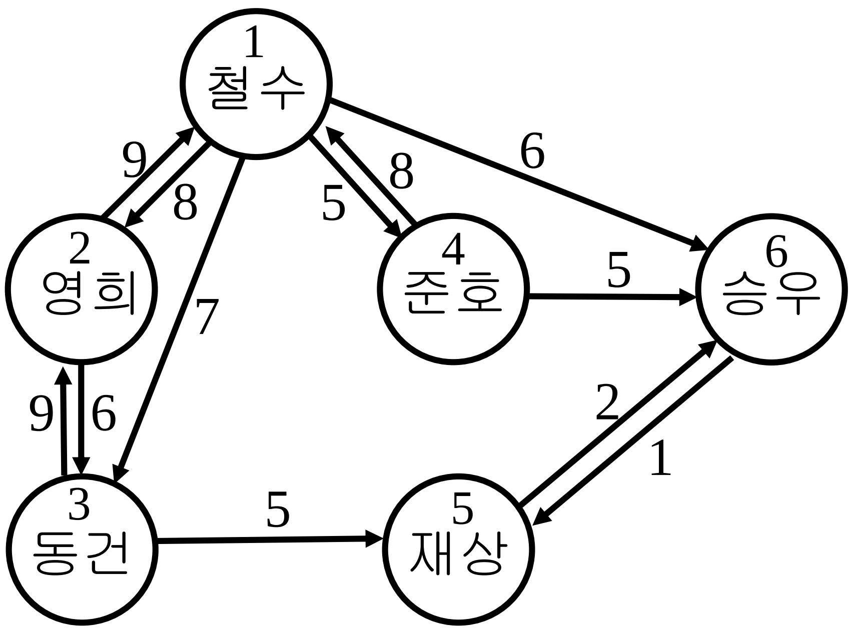
Adjacency Matrix



	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

Directed Graph의 예

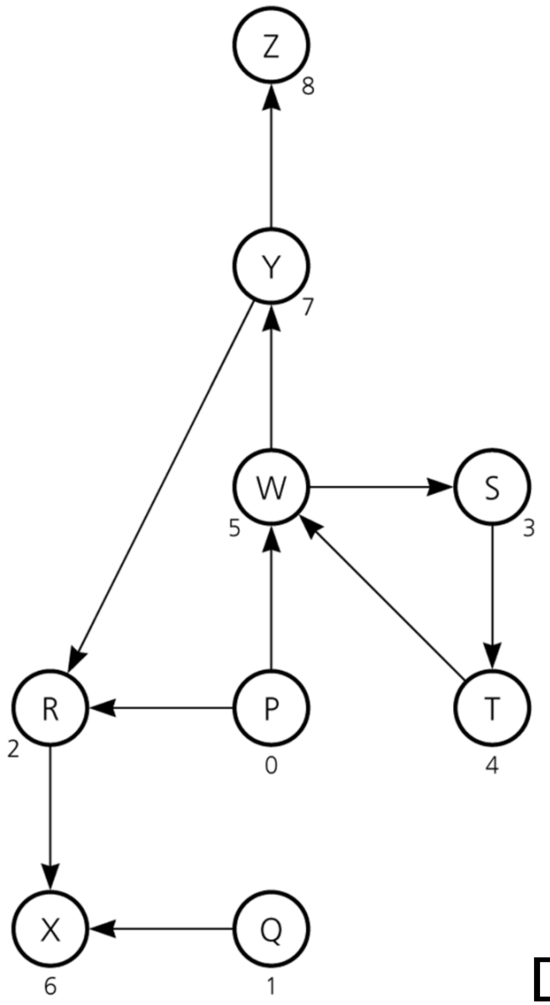
Adjacency Matrix



	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 Digraph의 예

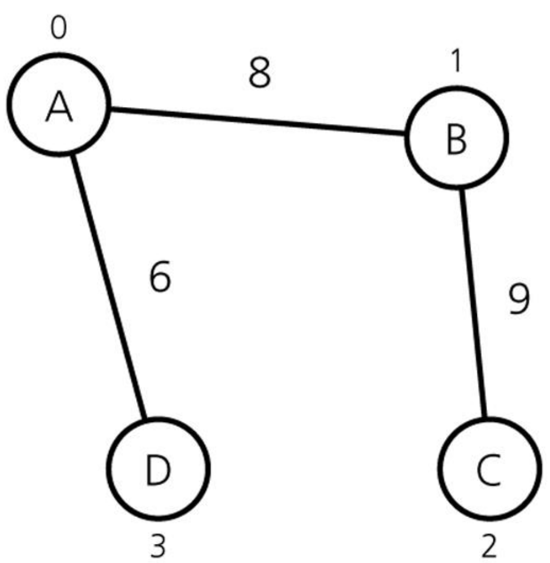
Adjacency Matrix



		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

Digraph의 다른 예

Adjacency Matrix

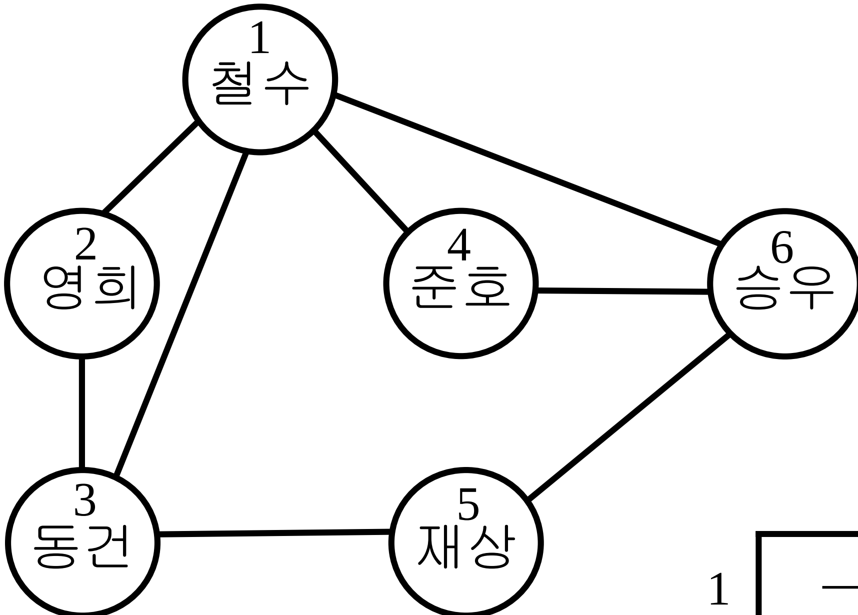


		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

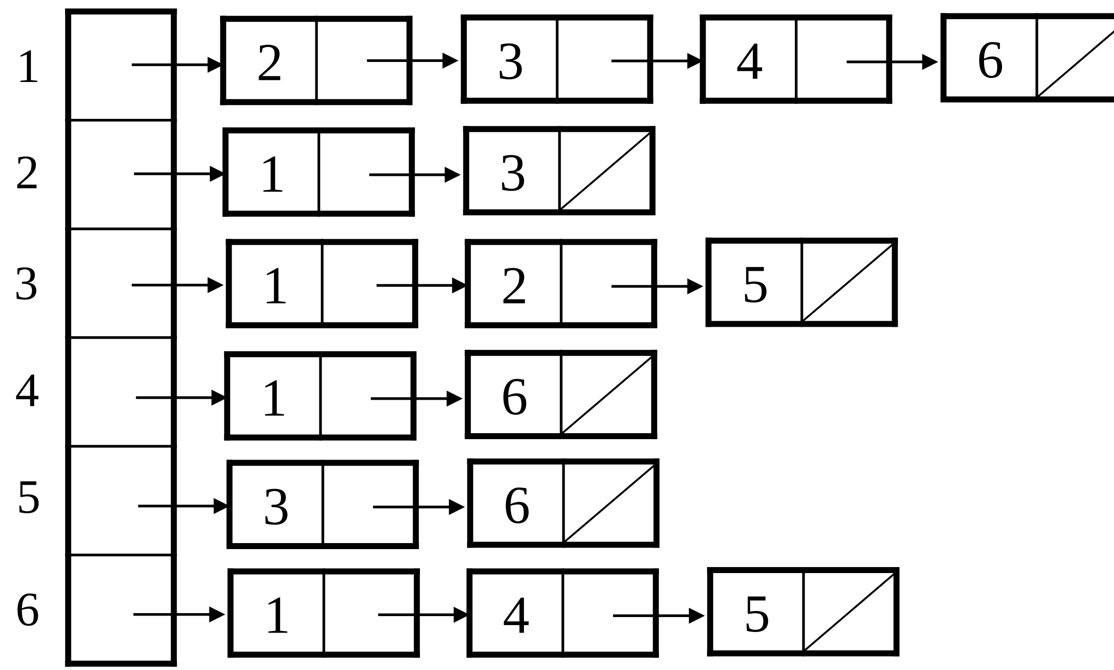
Weighted Graph의 다른 예

- Adjacency list
  - A graph is represented by N linked lists where list i is the list of vertices that is adjacent to vertex i.
  - In case of weighted graphs, the list also contains the weight values.

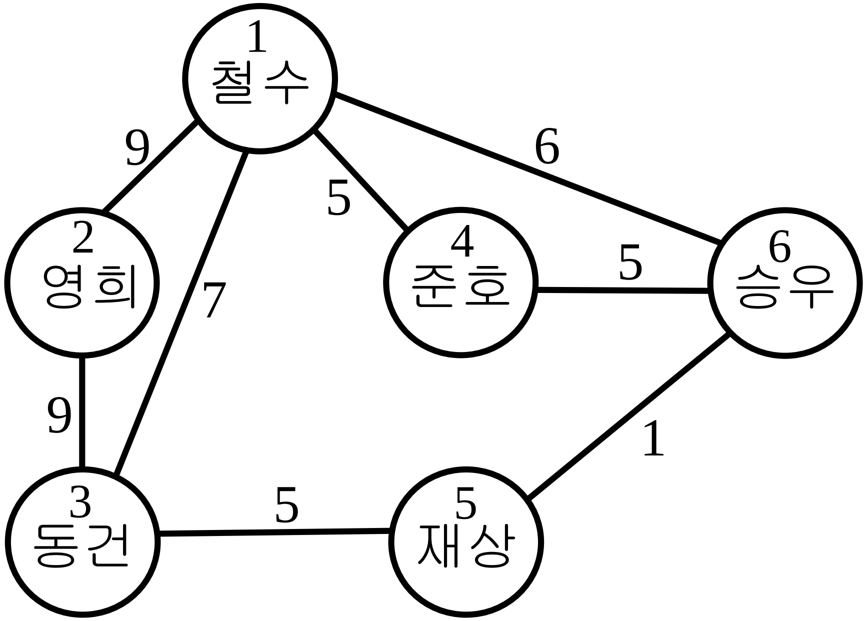
Adjacency List



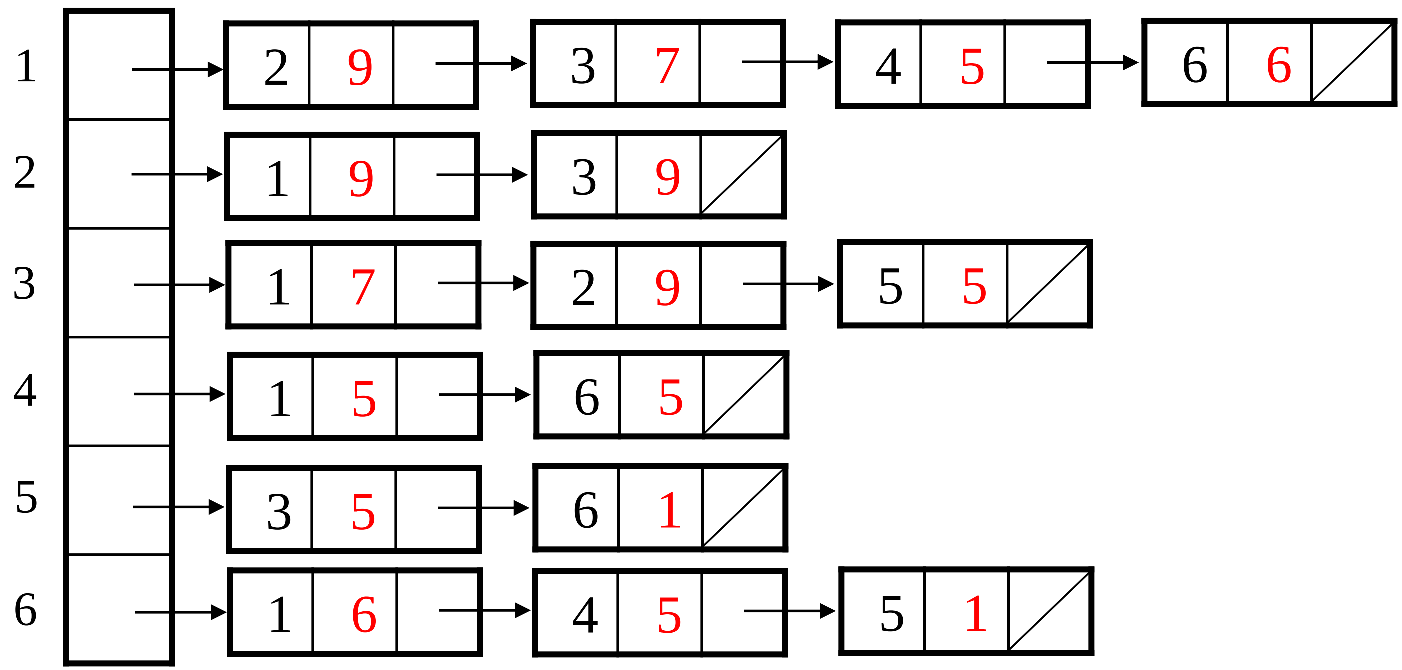
Undirected Graph의 예



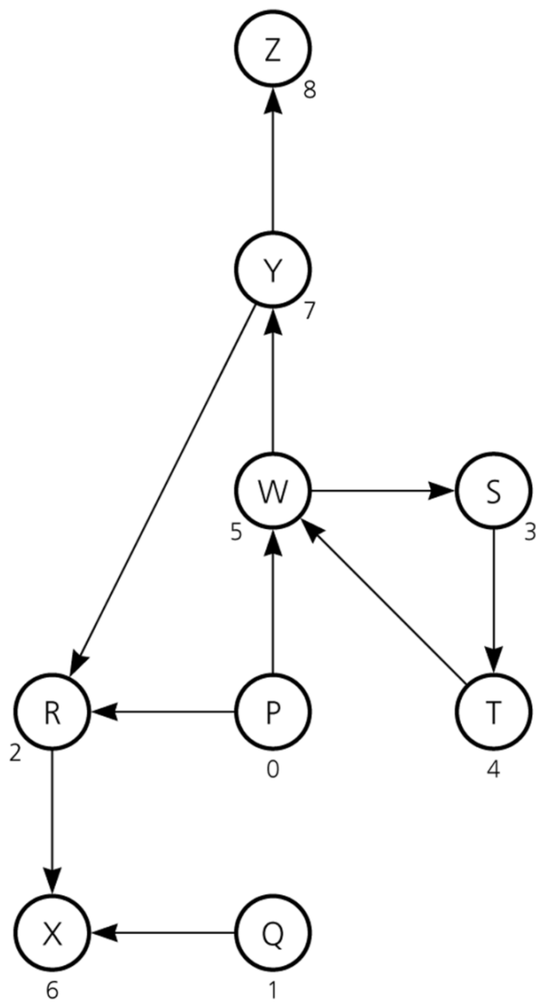
Adjacency List



Weighted Graph의 예

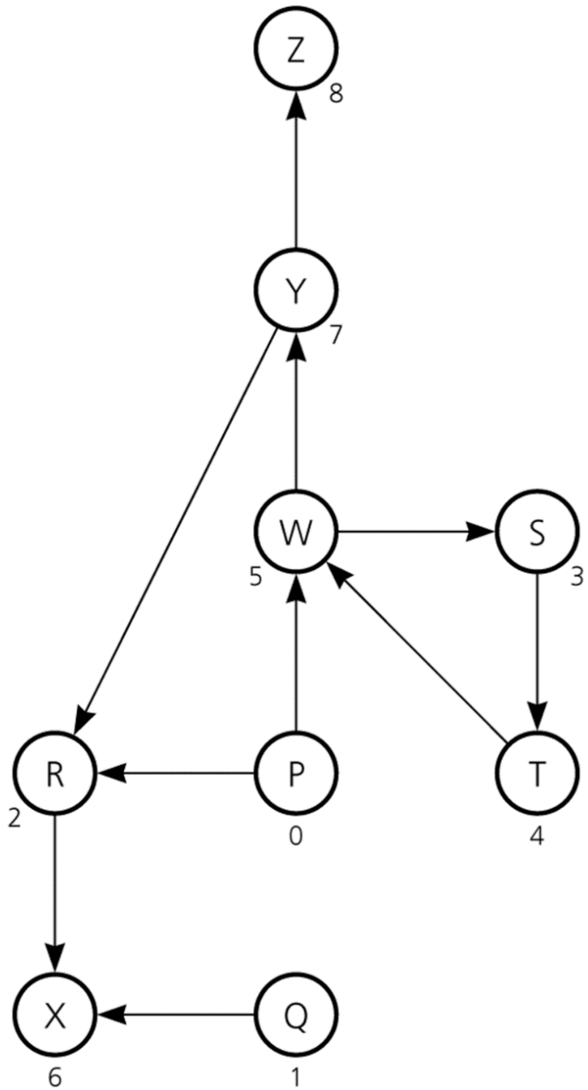


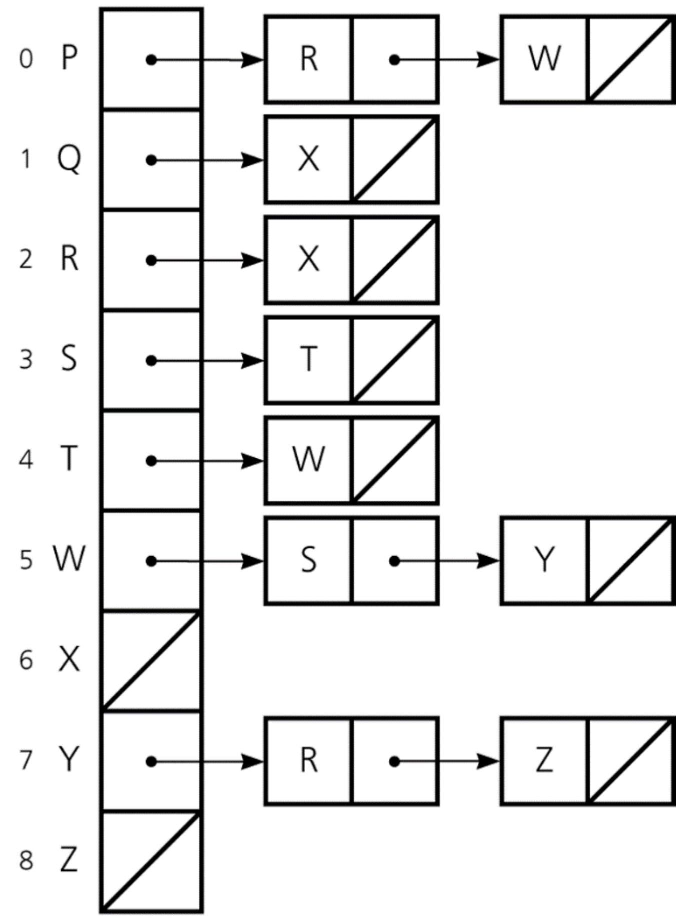
A Digraph and Its Adjacency Matrix



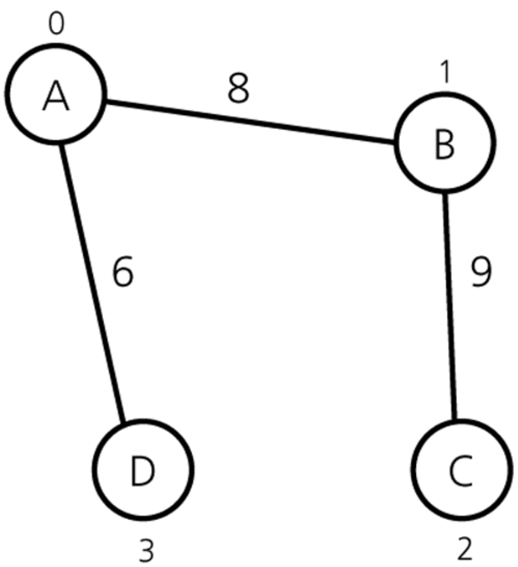
		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

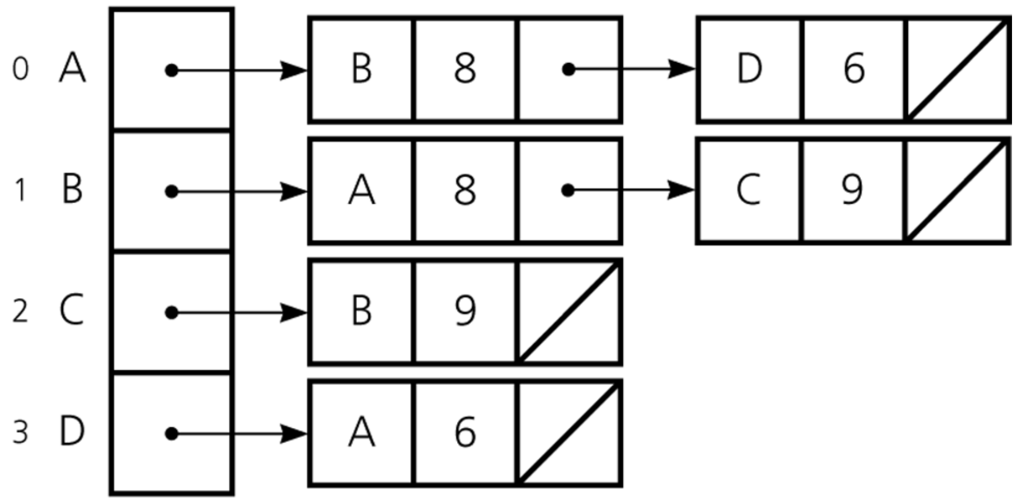
A Digraph and Its Adjacency List





A Weighted Digraph and Its Adjacency List





Graph Traversals

- Traversal in a graph G
  - starts at a vertex *u* and visits all vertices *v* for which there is a path bet’n *u* and *v*.
  - Two representatives
    - Depth-first search
    - Breadth-first search

Depth-First Search

```
DFS (v)
{
    Mark v as visited;
    for (each unvisited vertex u adjacent to v)
        DFS(u);
}
```

Non-Recursive Version of Depth-First Search

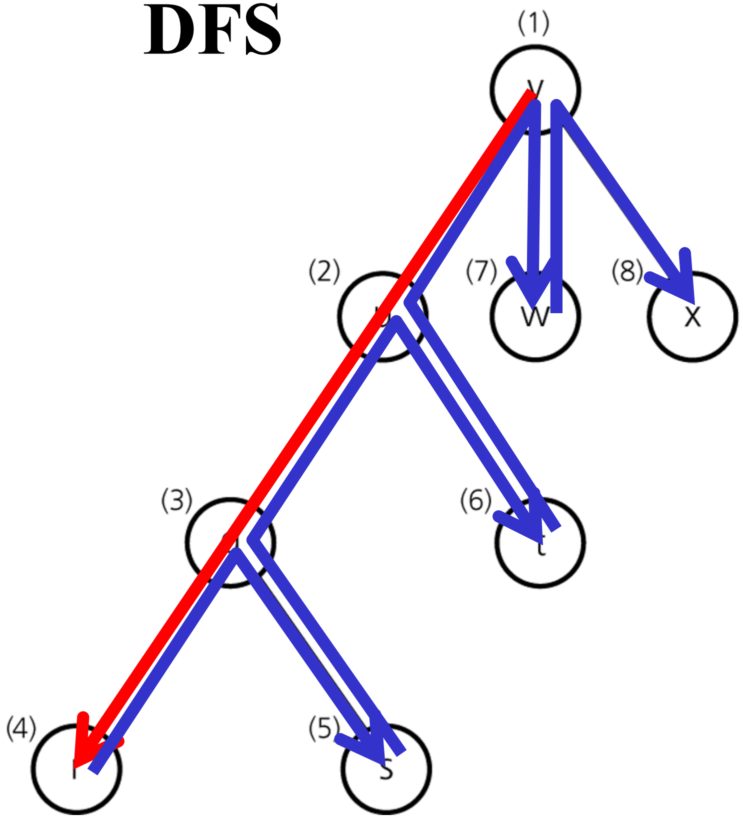
```
DFS(v)
{           // All vertices are marked UNVISITED in the beginning
            stack.push(v);
            mark[v] = VISITED;

            while (!stack.isEmpty( )) {
                        if (no unvisited vertices are adjacent to the stack-top vertex)
                                    stack.pop( );  // backtracking
                        else {
                                    Select an unvisited vertex w adjacent to the stack-top vertex;
                                    stack.push(w);
                                    mark[w] = VISITED;
                        }
            }
}
```

Breadth-First Search

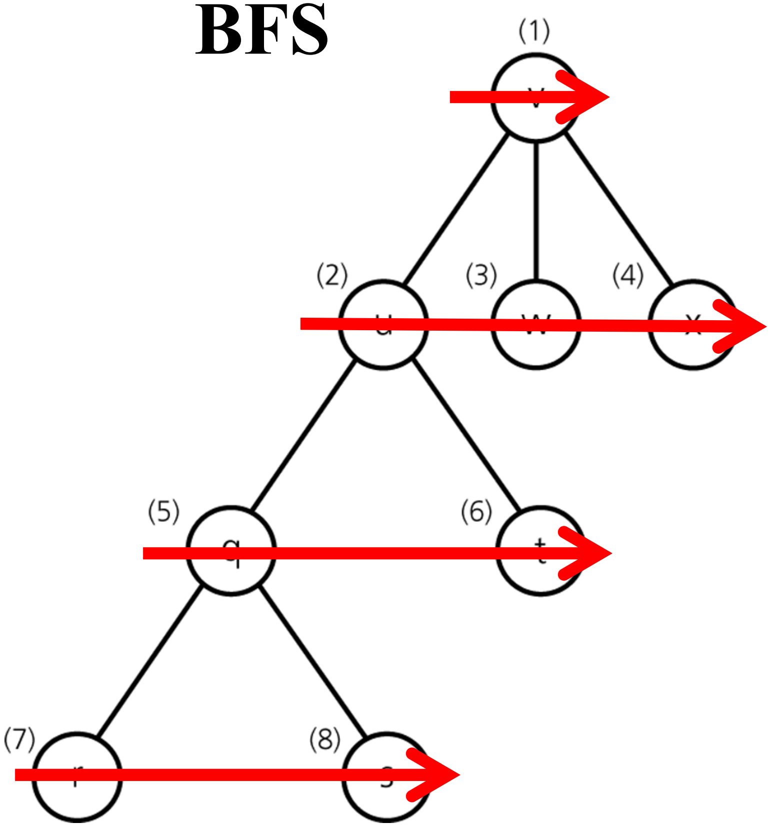
```
BFS(v)
{       // All vertices are marked UNVISITED in the beginning
        queue.enqueue(v);
        mark[v] = VISITED;
        while (!queue.isEmpty( )) {
                w = queue.dequeue( );
                for each unvisited vertex u adjacent to w
                {
                        queue.enqueue(u);
                        mark[u] = VISITED;
                }
        }
}
```

동일한 트리를 각각 DFS/BFS로 방문하기

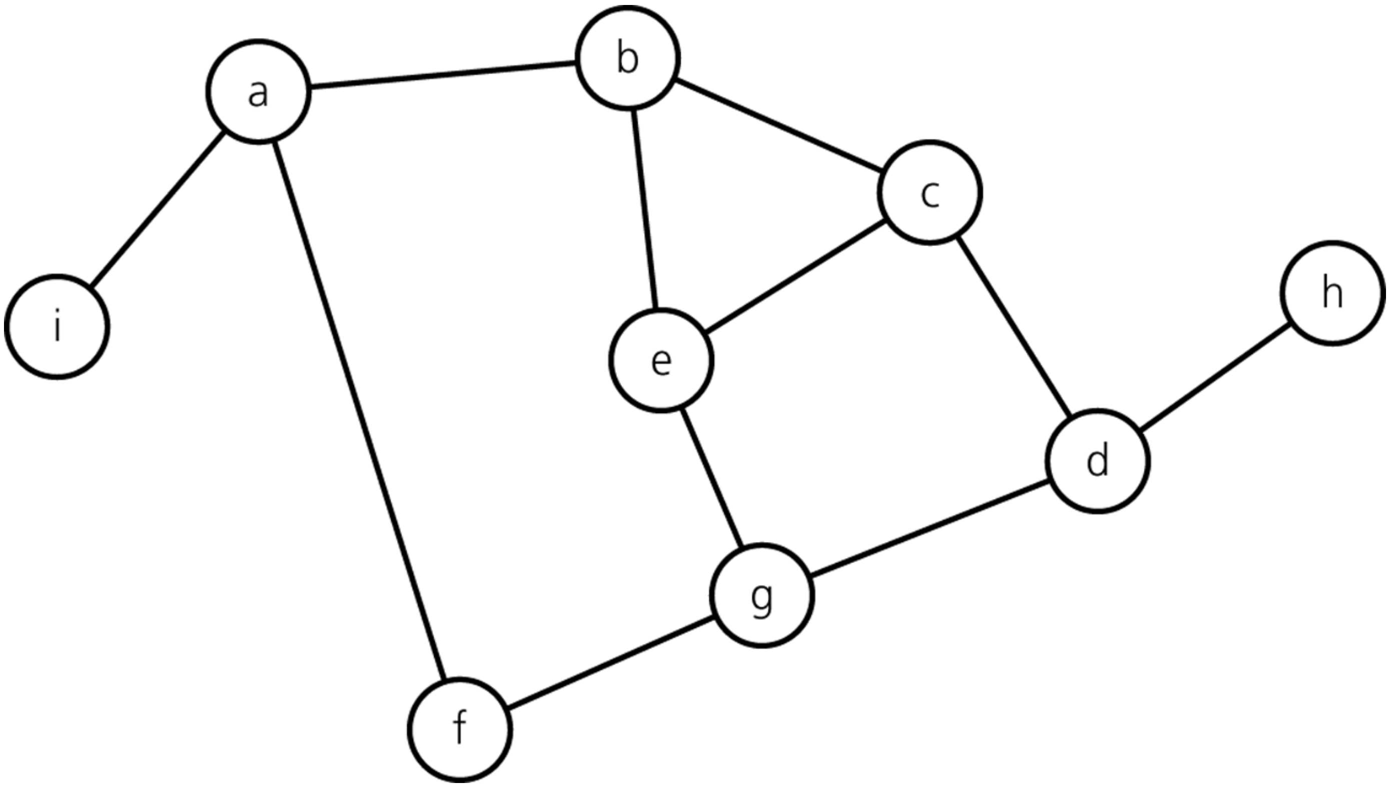
DFS



BFS



A Connected Graph with Cycles



The results of a depth-first traversal, beginning at vertex *a*, of the graph in the previous page

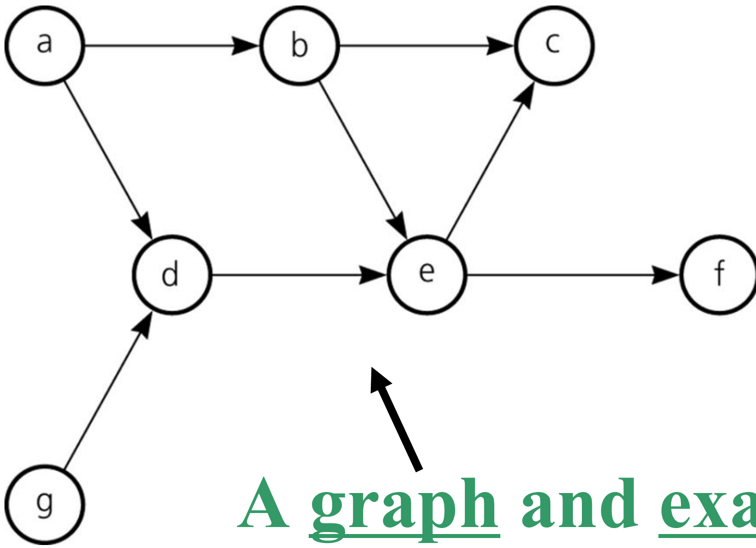
Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

The results of a breadth-first traversal, beginning at vertex *a*, of the graph in the previous page

Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

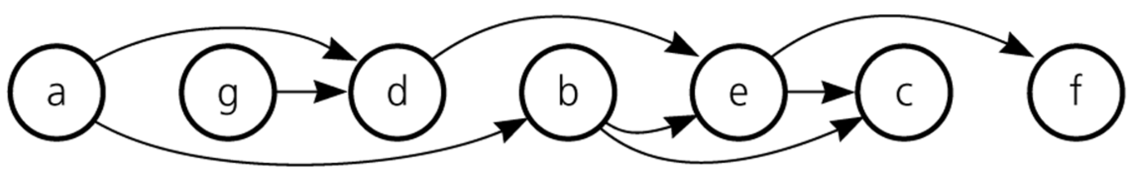
Topological Sorting

- Condition
  - Directed graph without cycles
- Topological order
  - An order of vertices in which vertex *x* precedes vertex *y* if there is an edge from *x* to *y*
  - Usually, there are many topological orders for a directed graph.

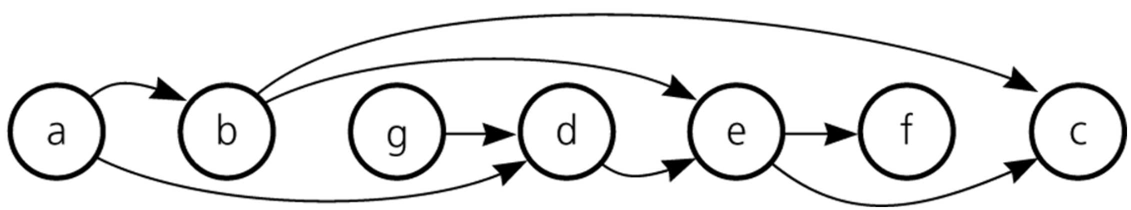


A <u>graph</u> and <u>examples</u> of topological orders

(a)

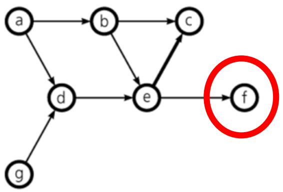


(b)

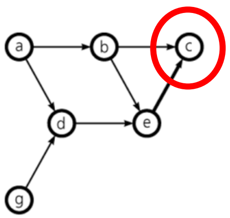


```
topologicalSort (G)
// G: graph
{
        for i = 1 to G.size( ) {
                Select a vertex v that has no successors;
                vertex[i] = v;
                Delete from G vertex v and the edges to v;
        }
        List vertex[G.size( )],…,vertex[1]; // a topological order
}
```

Graph **theGraph**	List **aList**

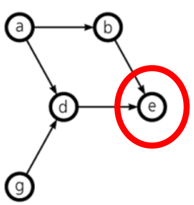


Remove f from **theGraph**;
add it to **aList**



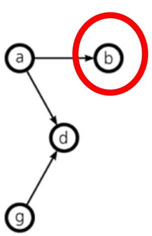
f

Remove c from **theGraph**;
add it to **aList**



c f

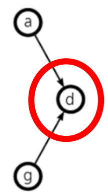
Remove e from **theGraph**;
add it to **aList**



e c f

Remove b from theGraph;
add it to aList

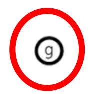
Graph **theGraph**	List **aList**



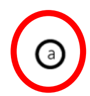
b e c f

Remove d from **theGraph**;
add it to **aList**

d b e c f



Remove g from **theGraph**;
add it to **aList**



g d b e c f

Remove a from **theGraph**;
add it to **aList**

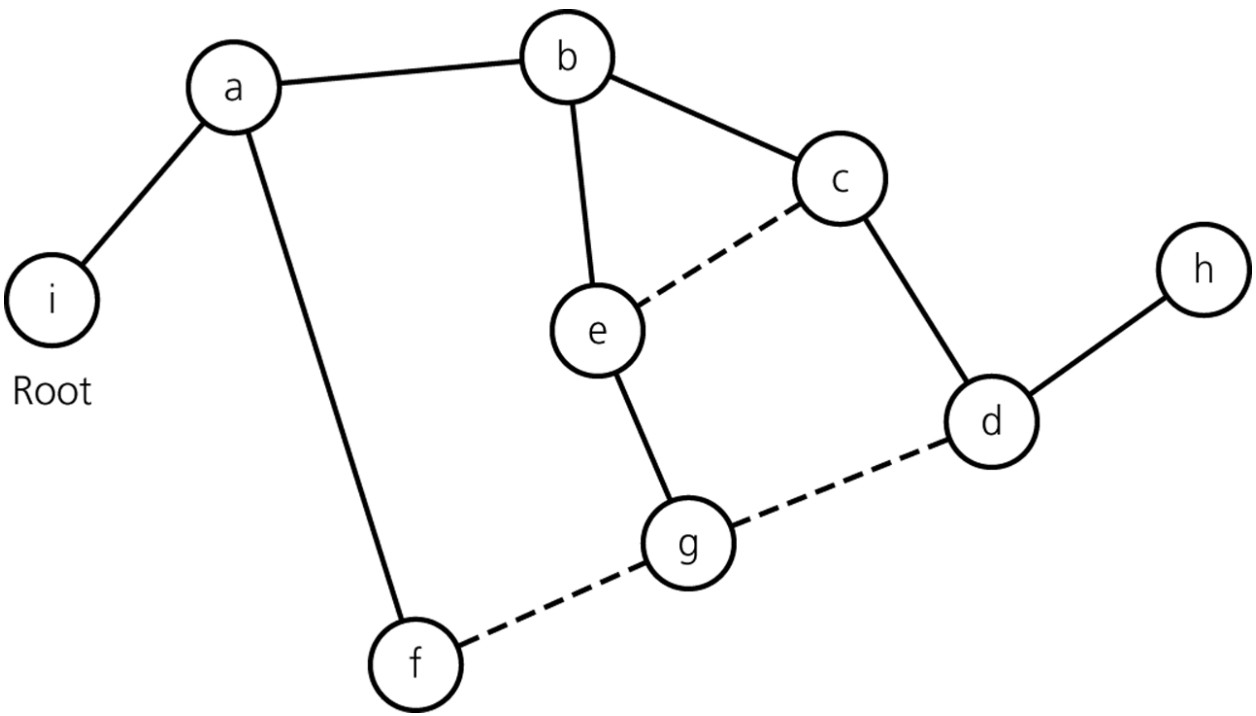
a g d b e c f

```
topologicalSort2 (G)
// G: graph
{
        for i = 1 to G.size( ) {
                Select a vertex v that has no predecessors;
                vertex[i] = v;
                Delete from G vertex v and the edges outgoing from v;
        }
        List vertex[1], …, vertex[G.size( )]; // a topological order
}
```

Spanning Trees

- Condition
  - Undirected connected graph
- Tree
  - A connected graph with no cycles
  - A tree with n vertices always has n-1 edges.
- Spanning tree of graph G
  - A tree as a subgraph of G that contains all of G's vertices

A Spanning Tree



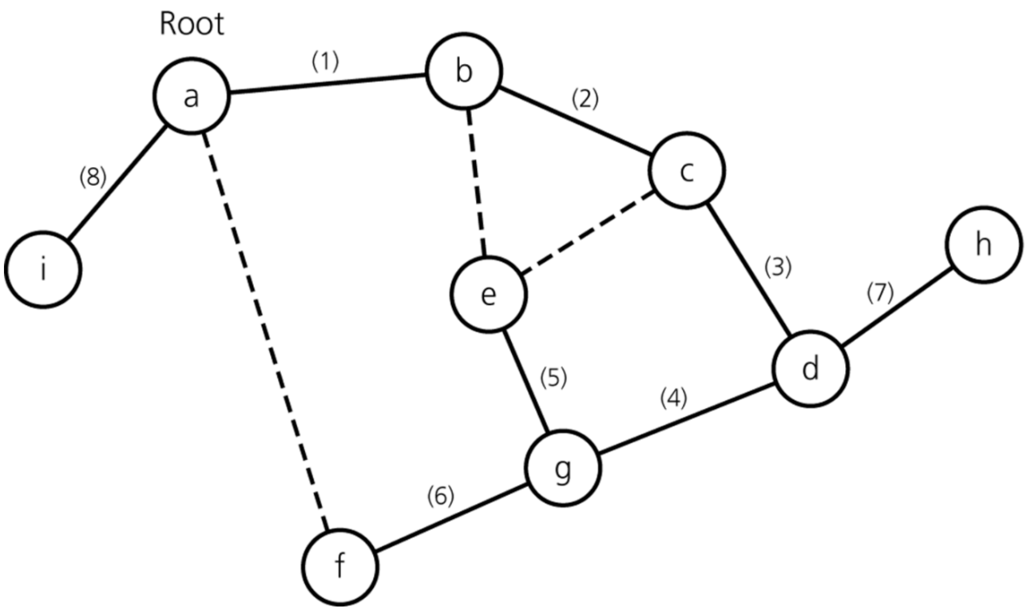
DFS/BFS Spanning Trees

```
DFSTree (v)
{
        Mark v as visited;
        for (each unvisited vertex u adjacent to v) {
                Mark the edge (u, v);
                DFSTree(u);
        }
}
```

✓ The edges marked in DFSTree() constitute a DFS spanning tree.

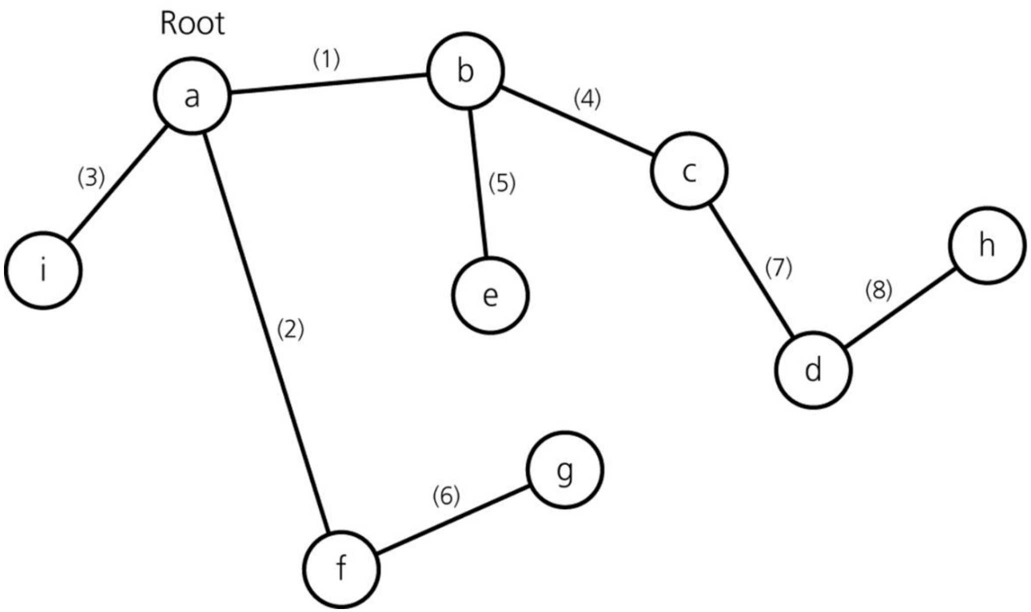
```
BFSTree(v)
{           // All vertices are marked UNVISITED in the beginning
            queue.enqueue(v);
            mark[v] = VISITED;
            while (!queue.isEmpty( )) {
                        w = queue.dequeue( );
                        for each vertex u adjacent to w
                        {
                                    queue.enqueue(u);
                                    mark[u] = VISITED;
                                    Mark the edge (w, u);
                        }
            }
}
```

✓ The edges marked in BFSTree() constitute a BFS spanning tree.



The DFS spanning tree algorithm visits vertices in this order: a, b, c, d, g, e, f, h, i. Numbers indicate the order in which the algorithm marks edges.

A DFS spanning tree rooted at vertex ***a***



The BFS spanning tree algorithm visits vertices in this order: a, b, f, i, c, e, g, d, h. Numbers indicate the order in which the algorithm marks edges.

A BFS spanning tree rooted at vertex ***a***

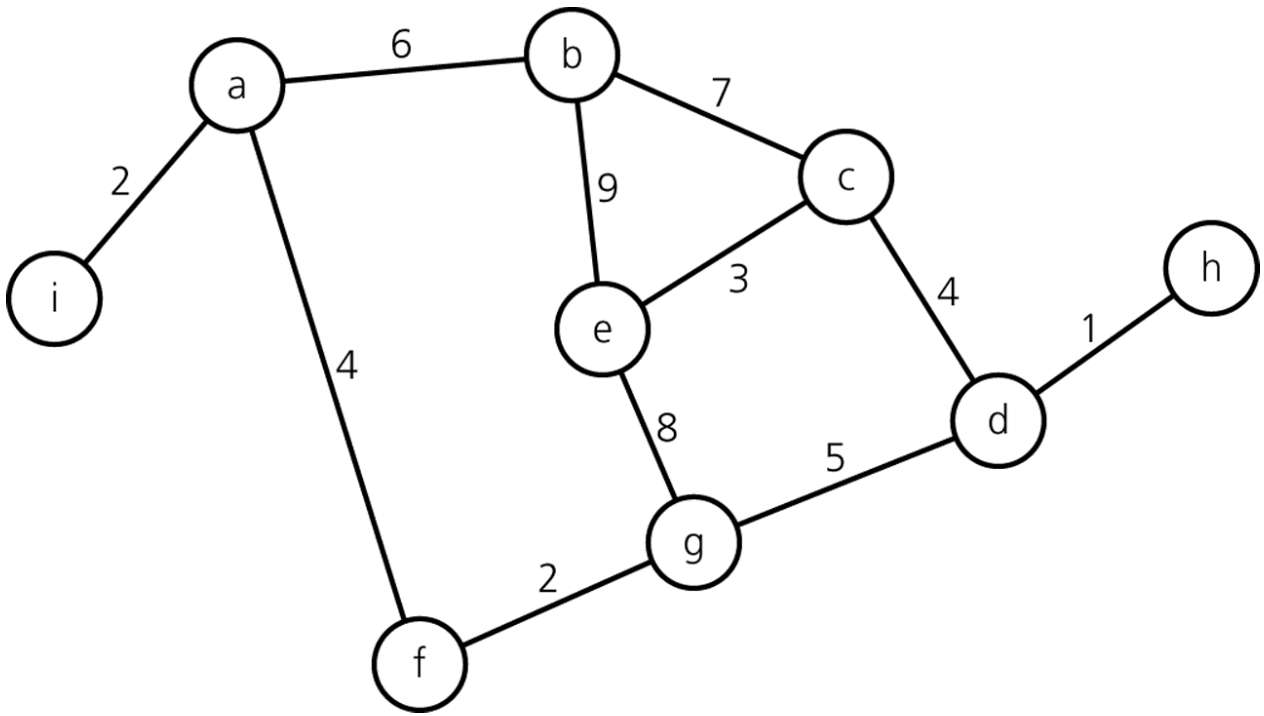
Minimum Spanning Trees

- Condition
  - Weighted undirected graph
- The cost of a spanning tree
  - The sum of the edge weights in the spanning tree
- Minimum spanning tree
  - A spanning tree with the minimum cost

Prim Algorithm

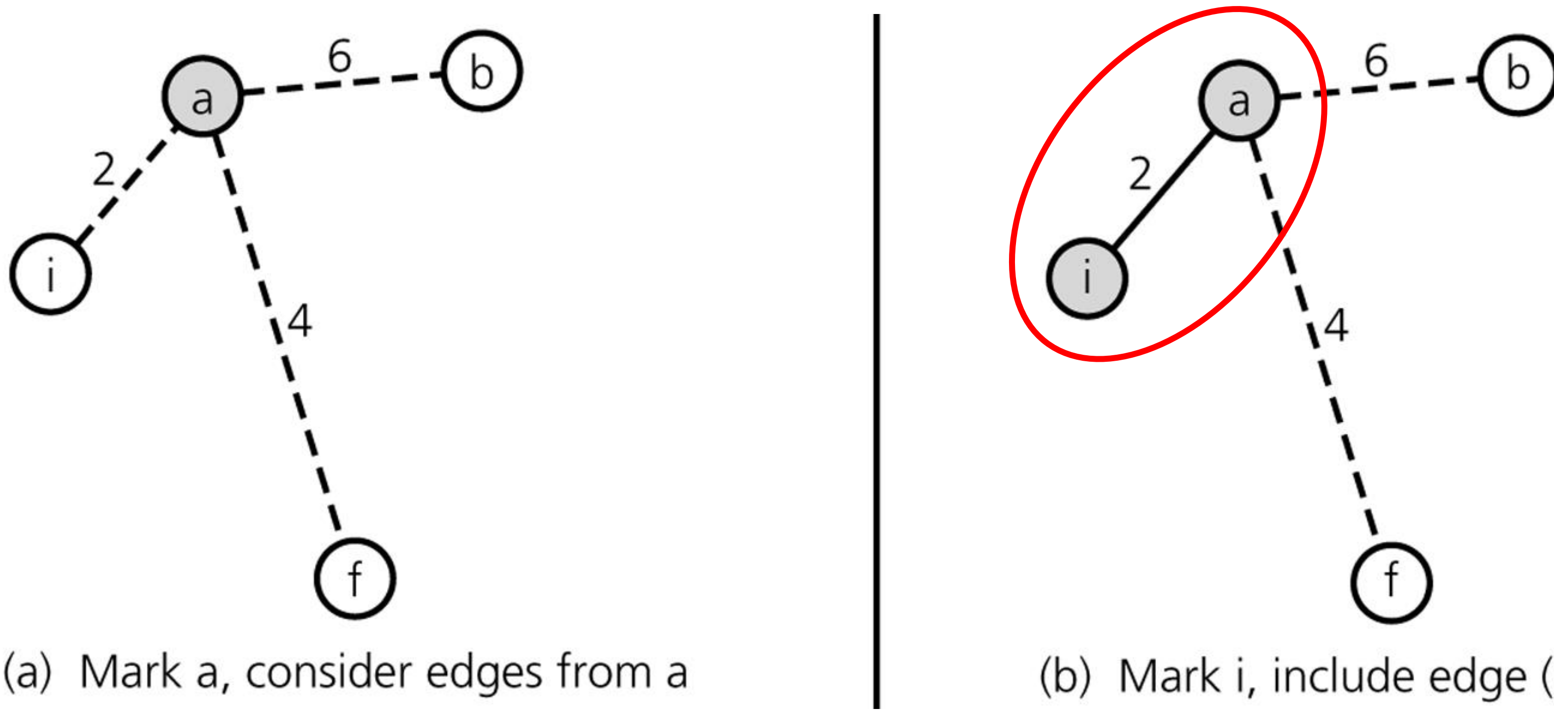
```
PrimAlgorithm (v)
{
        Mark v as visited and include it in the m.s.t.;
        while (there are unvisited vertices) {
                Find a least-cost edge (x,u) from a visited vertex x
                                        to an unvisited vertex u;
                Mark u as visited;
                Add the vertex u and the edge (x,u) to the m.s.t.;
        }
}
```

- ✓ Prim's algorithm is an example of greedy algorithm.
- ✓ It is a rare case that a greedy algorithm guarantees global optimality.



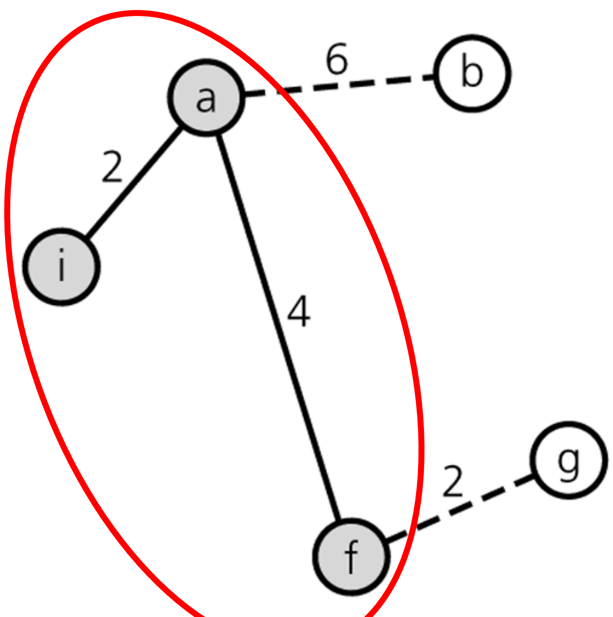
A weighted, connected, undirected graph

Prim Algorithm의 작동 예 1

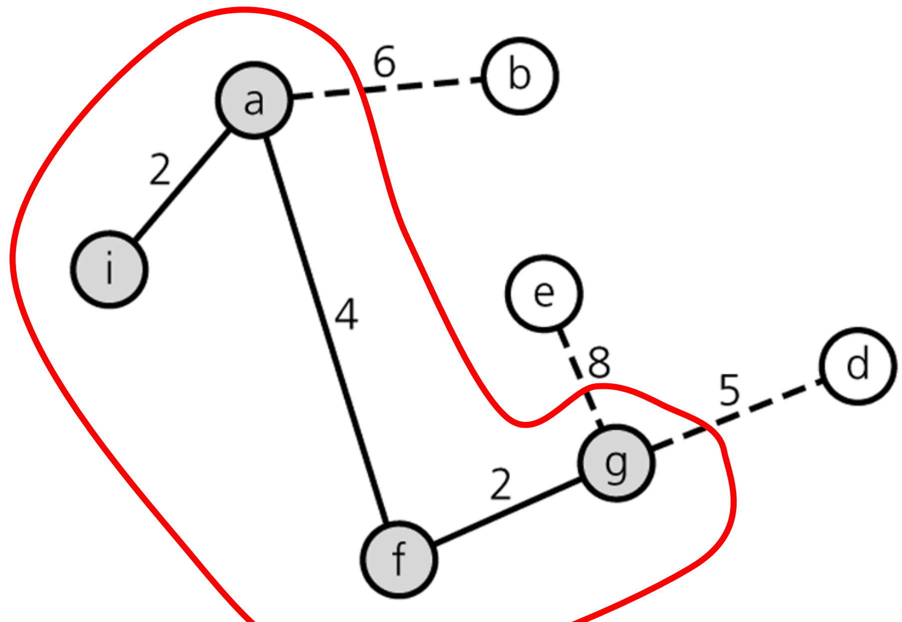


(a) Mark a, consider edges from a

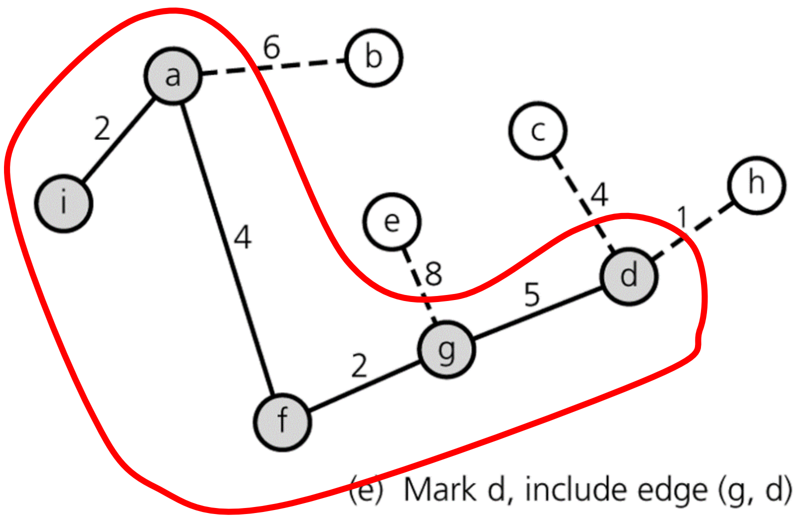
(b) Mark i, include edge (a, i)



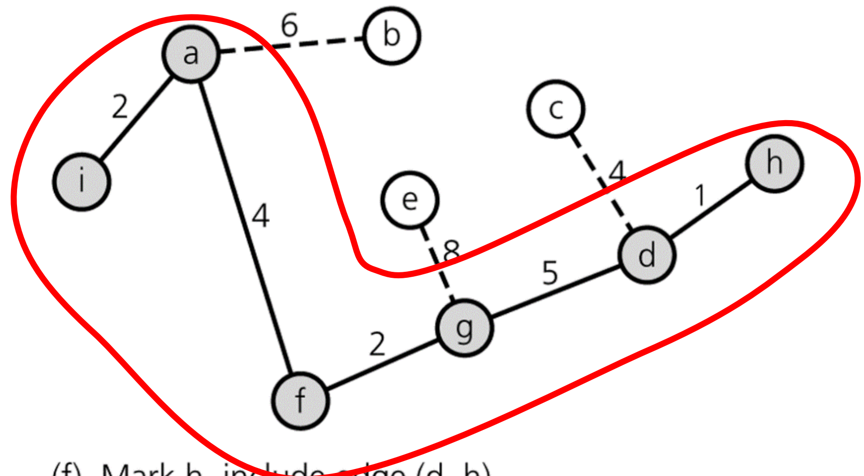
(c) Mark f, include edge (a, f)



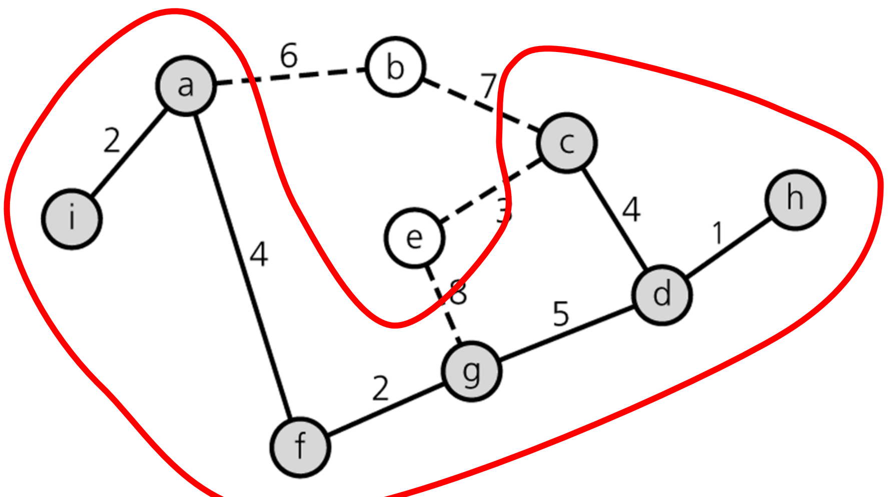
(d) Mark g, include edge (f, g)



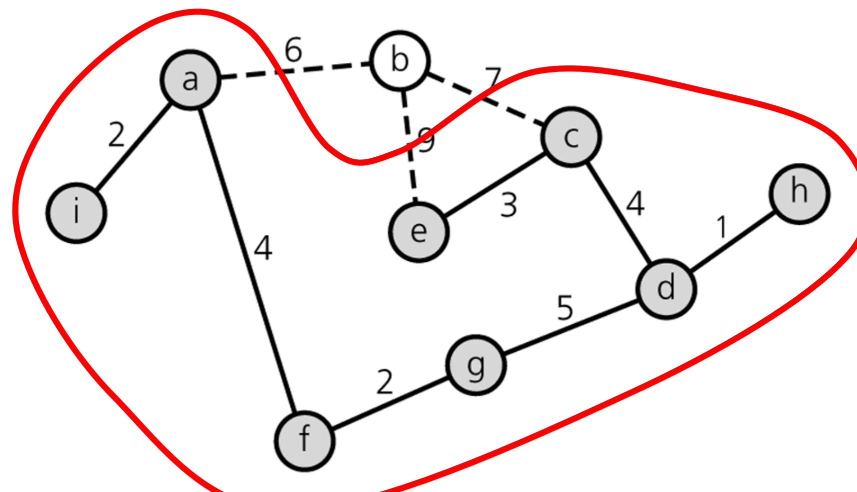
(e) Mark d, include edge (g, d)



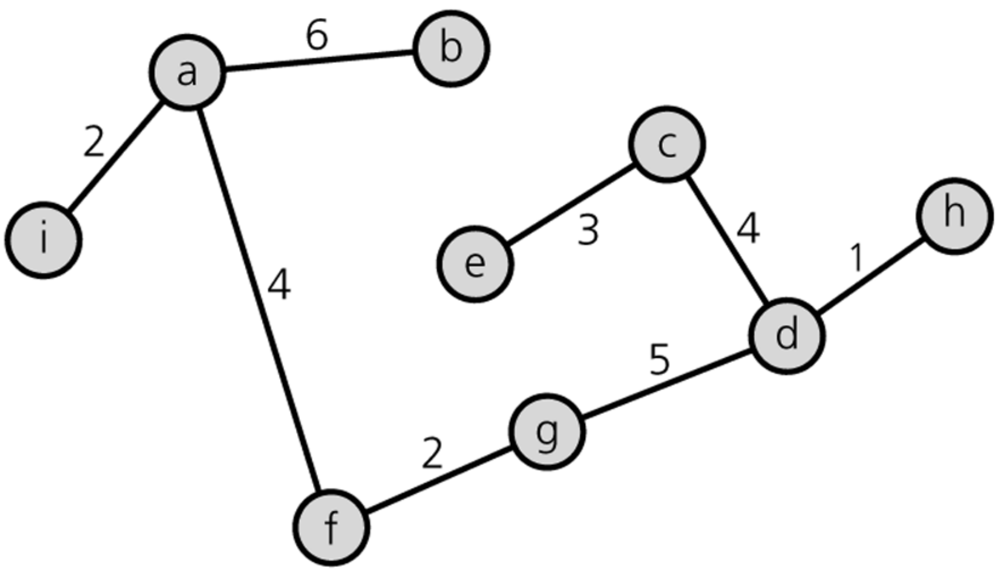
(f) Mark h, include edge (d, h)



(g) Mark c, include edge (d, c)

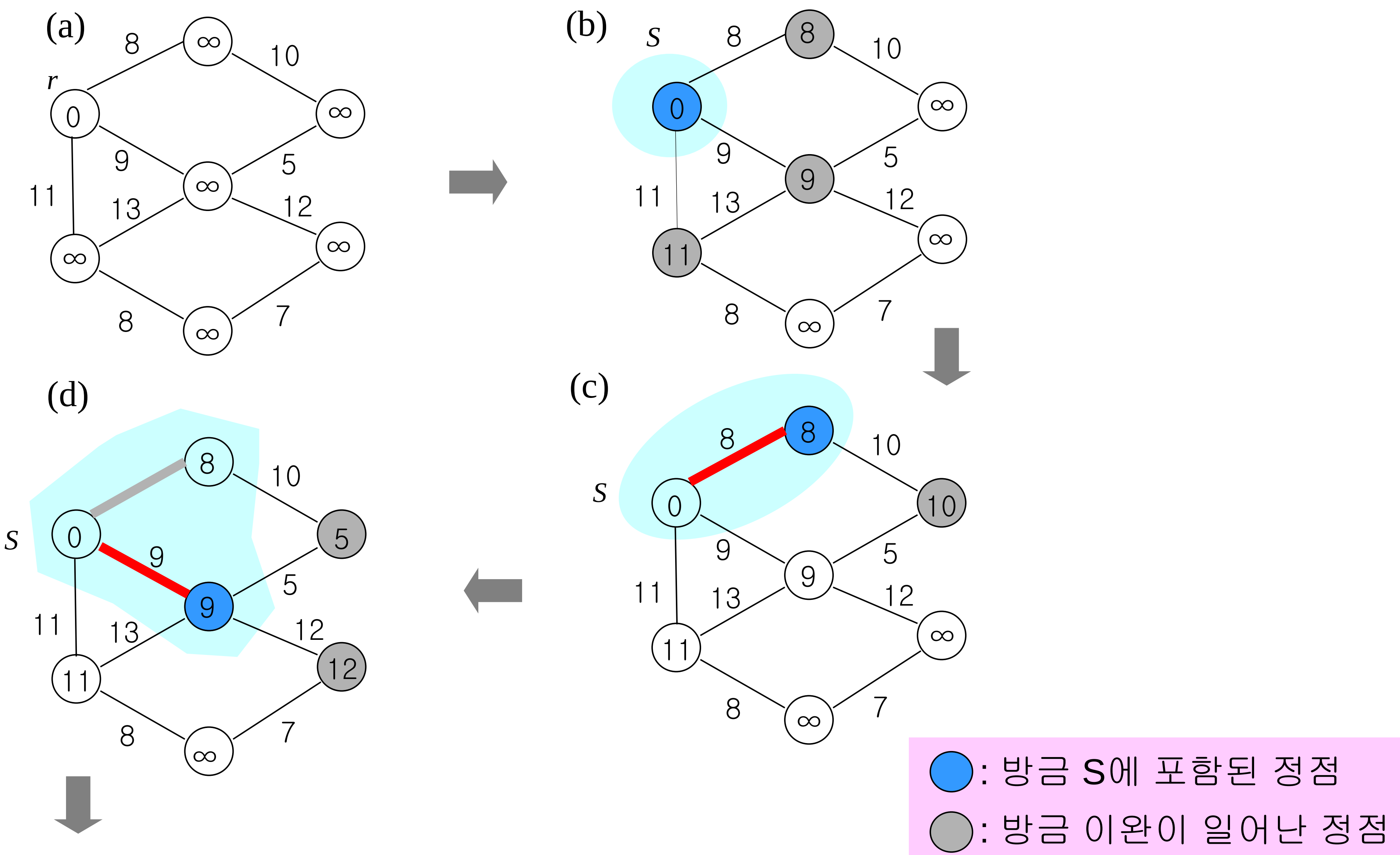


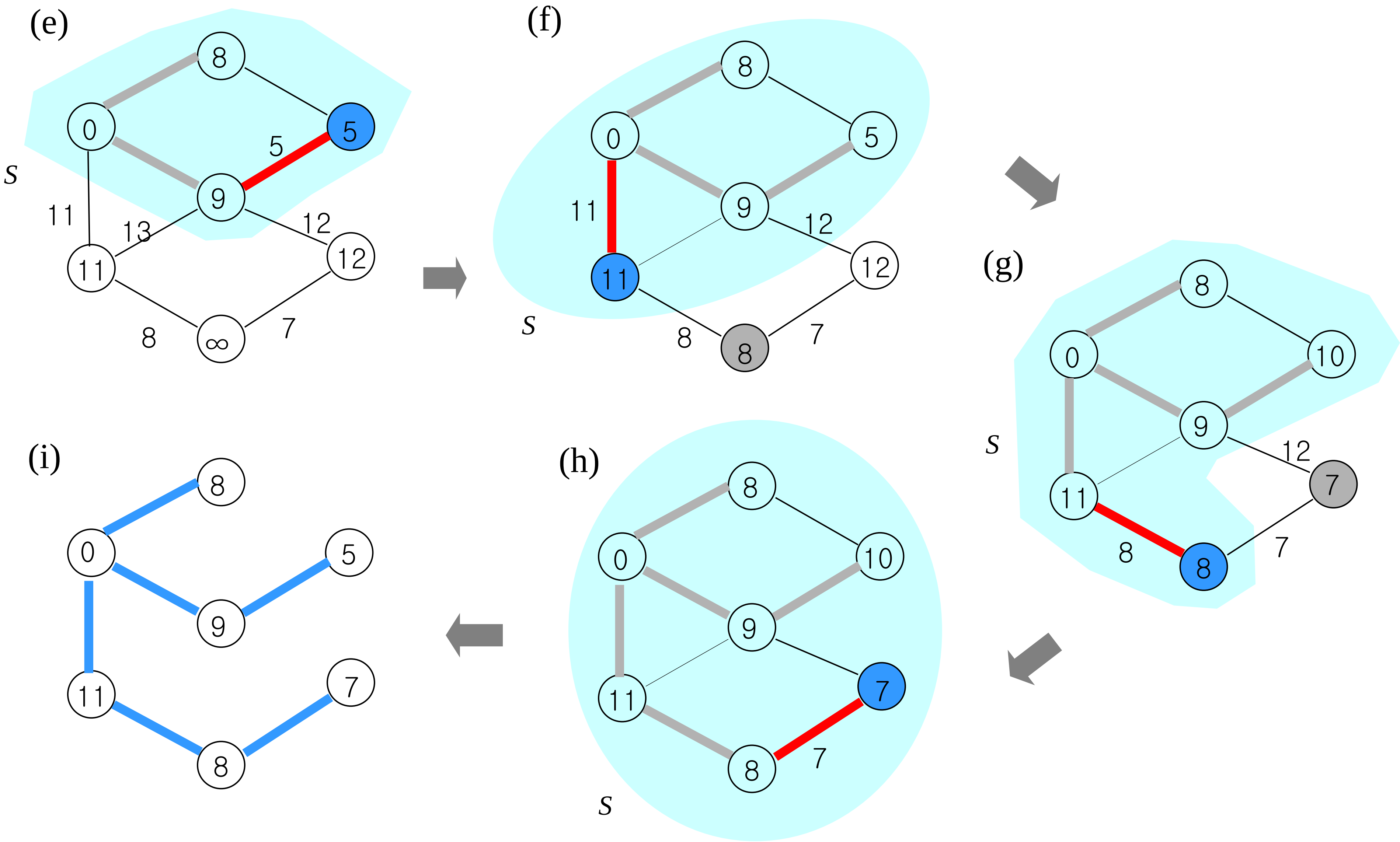
(h) Mark e, include edge (c, e)



(i) Mark b, include edge (a, b)

Prim Algorithm의 작동 예 2




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

Shortest Paths

- Condition
  - Weighted digraph
  - Undirected graph can be also thought to be a digraph in which undirected edge (u,v) means two directed edges (u,v) and (v,u).
- Shortest path bet’n two vertices
  - A path that has the smallest sum of edge weights

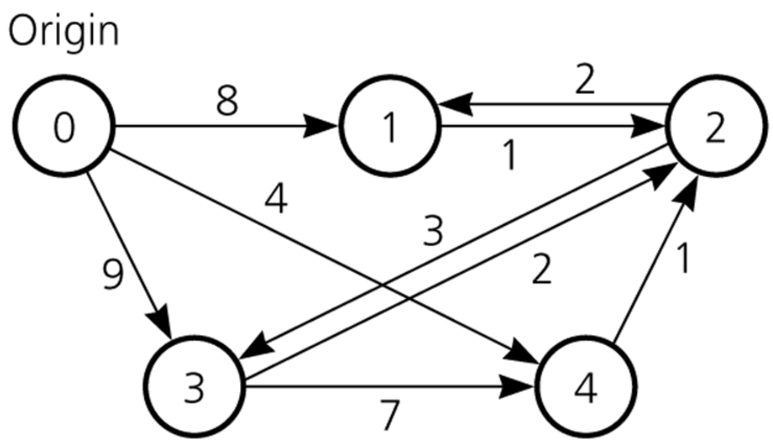
Dijkstra Algorithm

```
DijkstraAlgorithm (G, s) // s: the starting vertex
{
        S = {s}  ;
        for v = 1 to G.size( )
                distance[v] = weight(s,v);
        for i = 2 to G.size( ) {
                Find the smallest distance[v] s.t. v is not in S;
                S = S ∪ {v};
                for each vertex u not in S {
                        if (distance[u] > distance[v] + weight(v,u))
                                distance[u] = distance[v] + weight(v,u);
                }
        }
}
```

Relaxation!

- ✓ Dijkstra's algorithm is an example of greedy algorithm.
- ✓ It is another rare case that a greedy algorithm guarantees global optimality.

(a)

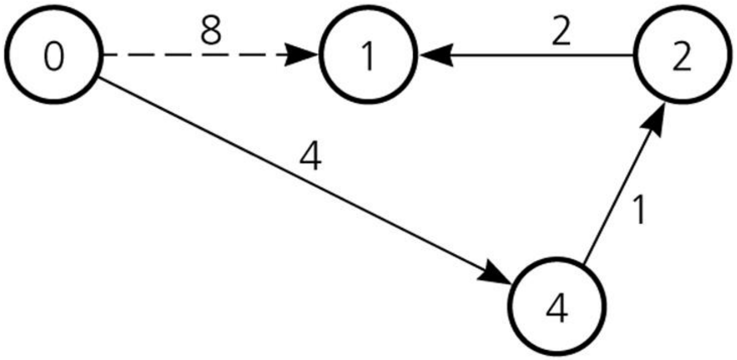


(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

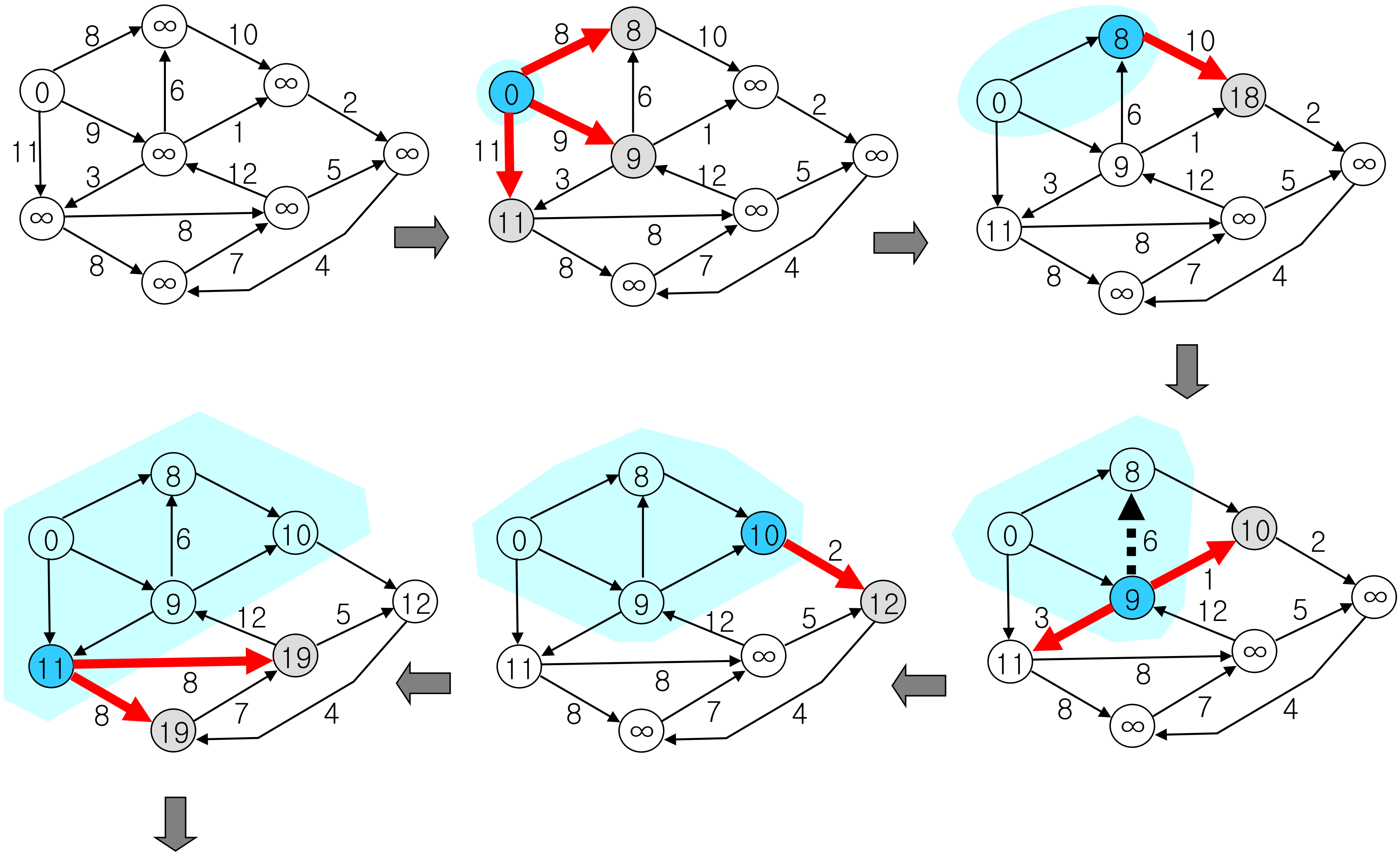
A weighted directed graph and its adjacency matrix

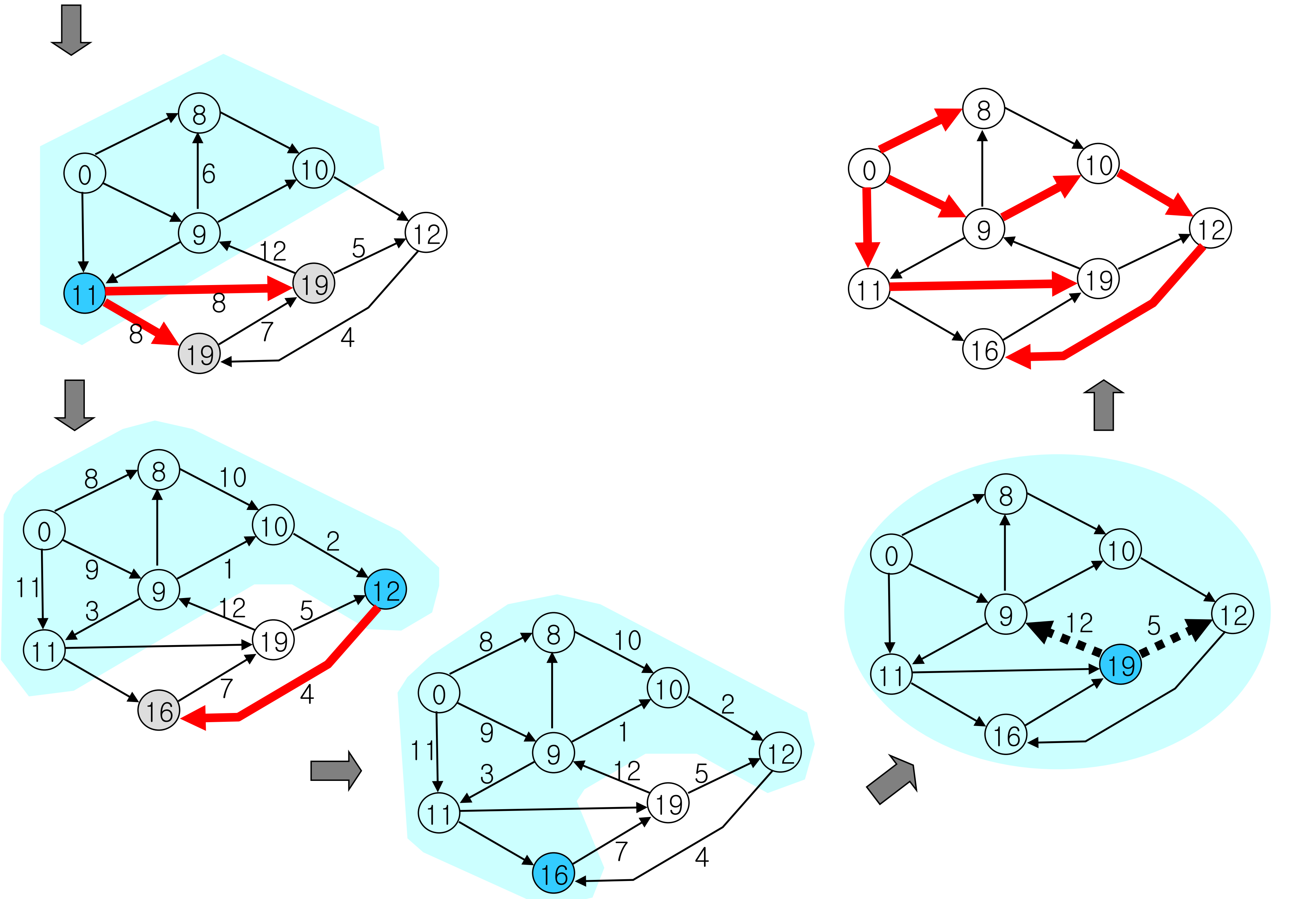
An example relaxation



The path 0–4–2–1 is shorter than 0–1

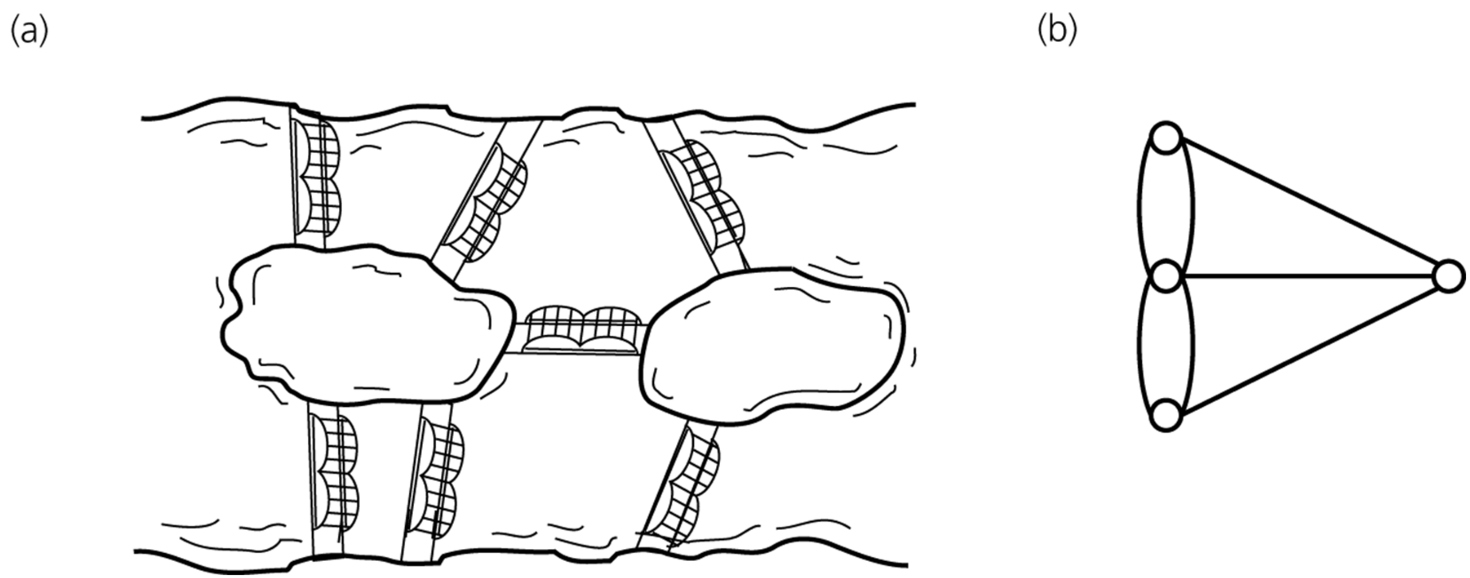
Dijkstra Algorithm의 작동 예





0 8 10 9 12 11 19 19
6 12 5 8 8 7 4
0 8 10 9 12 11 19 16
8 10 9 1 2 11 3 12 5 7 4
0 8 10 9 12 11 19 16
8 10 9 1 2 11 3 12 5 7 4
0 8 10 9 12 11 19 16
12 5
0 8 10 9 12 11 19 16

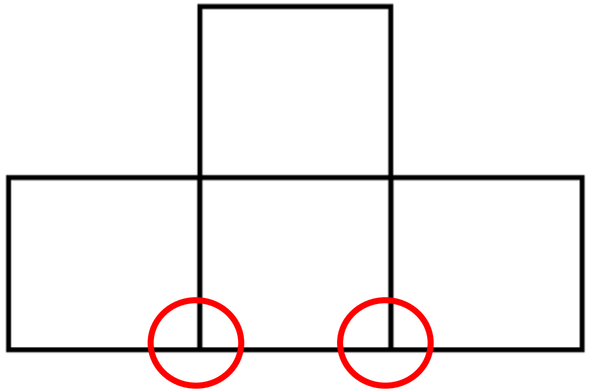
Euler Circuit : The Oldest Graph Application



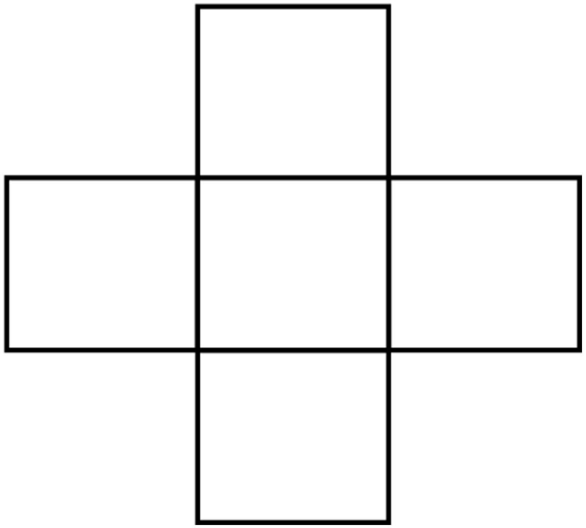
- ✓ Is there a cycle that uses every edge exactly once?
- ✓ To have an Euler circuit, every vertex must have an even degree.

Pencil and paper drawings

(a)



(b)



Other Graph Problems

- Traveling salesman problem
- Graph planarity
- Graph coloring
- Graph bisection
- Max flow
- VLSI circuit placement & routing
- Too many more …