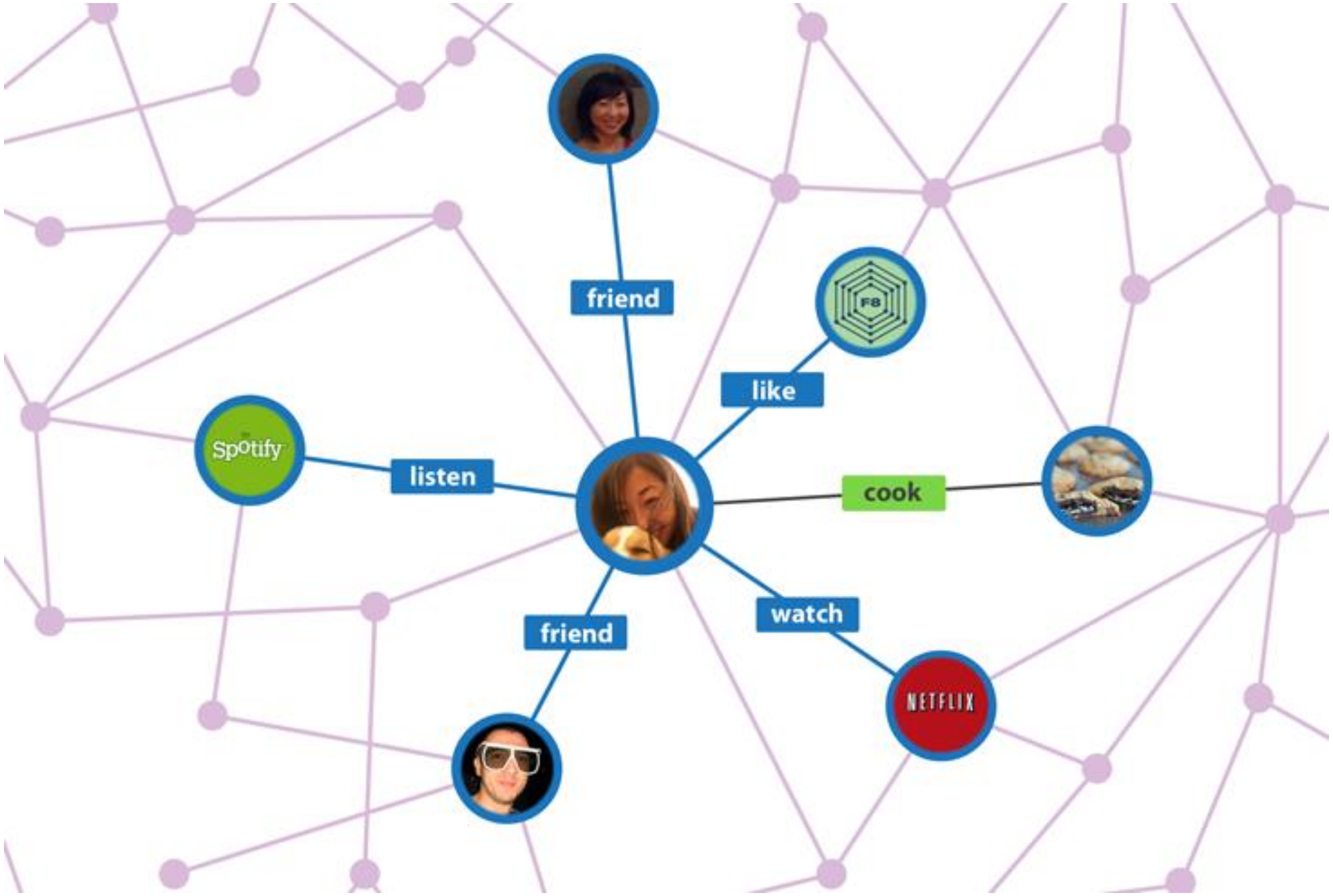
Graph

서승현 교수
Prof. Anes Seung-Hyun Seo
(seosh77@hanyang.ac.kr)

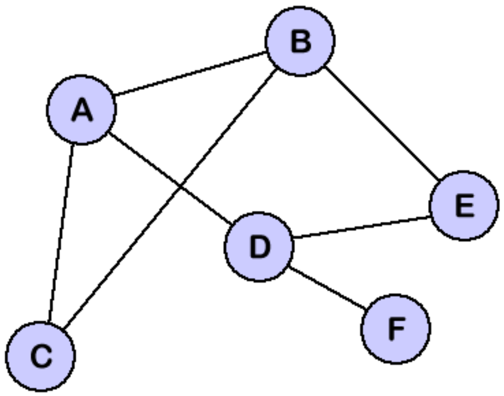
Division of Electrical Engineering
Hanyang University, ERICA Campus

Graph

- A more general nonlinear structure



Notation of Graph

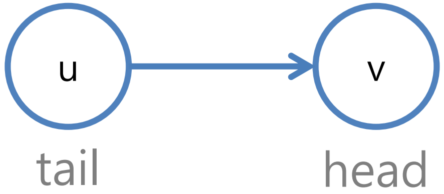


- **G = (V, E)**
  - V: set of vertices (=nodes)
  - E: set of edges
    - e.g., V = {A, B, C, D, E, F}
      E = { (A, B), (A, C), (A, D), (B, C), (B, E), (D, E), (D, F) }

- **Two kinds of edges**

Undirected edge = (u, v)

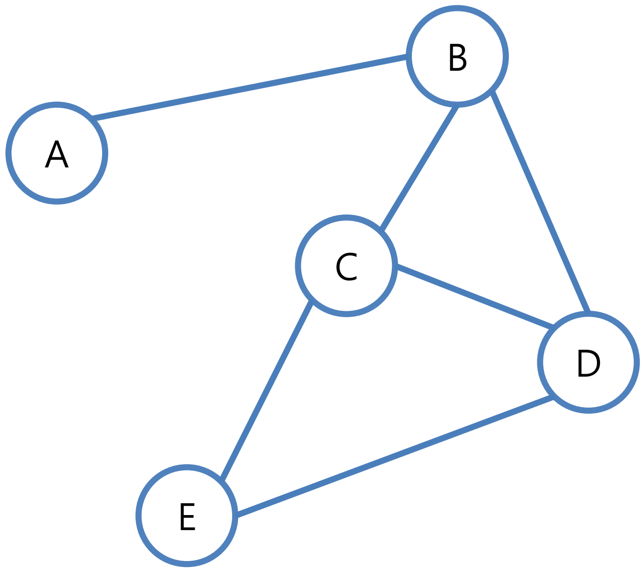
Directed edge = <u, v>



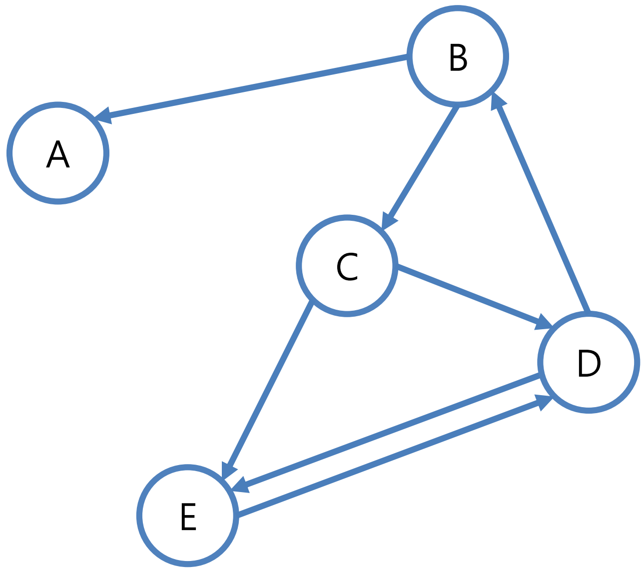
<v, u> ≠ <u, v>

Directed/undirected graphs

Undirected graph



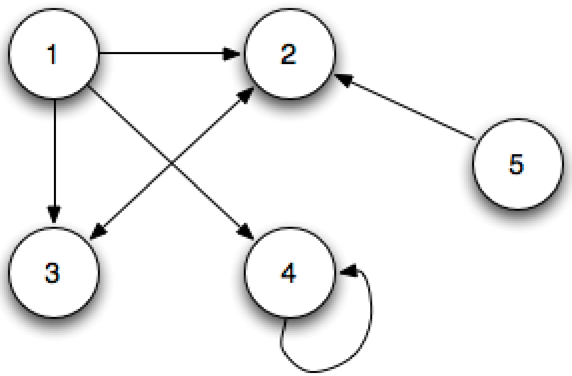
Directed graph
(=Digraph)



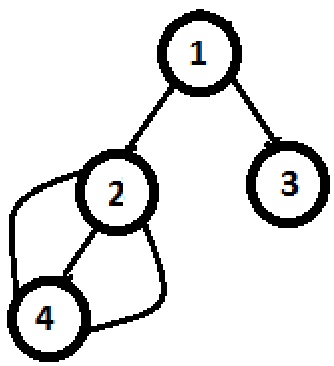
- Undirected graphs are special cases of directed graphs

Graph: Restrictions

- A Graph should not have a self-edge.
- There should be no more than one edge between two vertices.

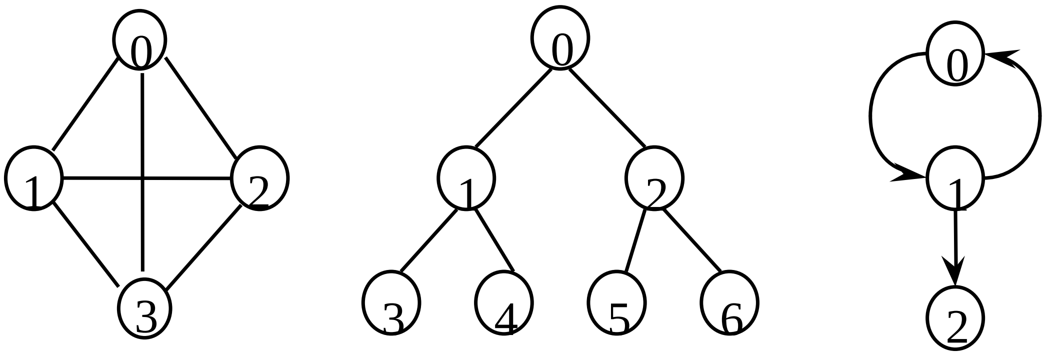


Self-edge



Multi-graph

Example of graphs



G_1 G_2 G_3

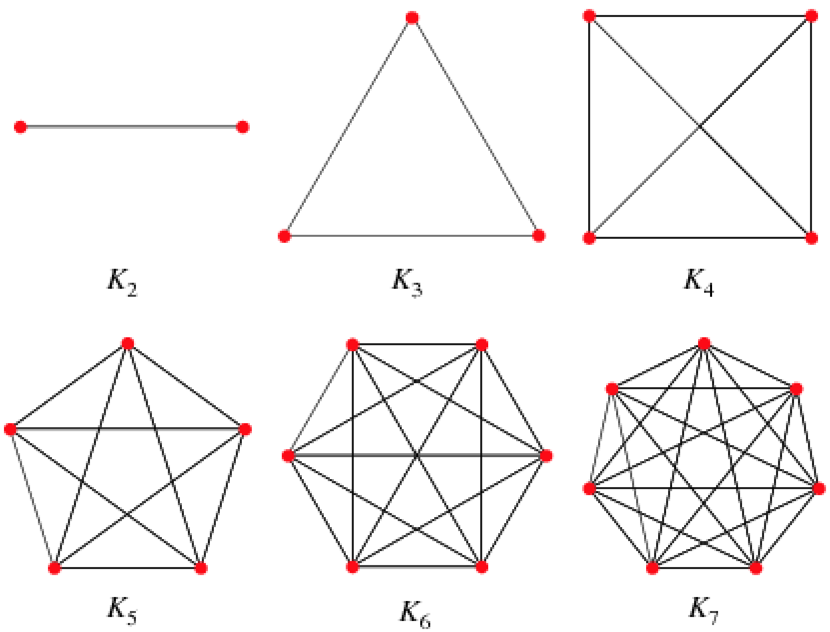
- $V(G_1)=\{0,1,2,3\}$
  $E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$

- $V(G_2)=\{0,1,2,3,4,5,6)$
  $E(G_2)=\{(0,1),(0,2),(1,3),(1,4),(2,5),(2,6)\}$

- $V(G_3)=\{0,1,2\}$
  $E(G_3)=\{<0,1>,<1,0>,<1,2>\}$

Complete graph

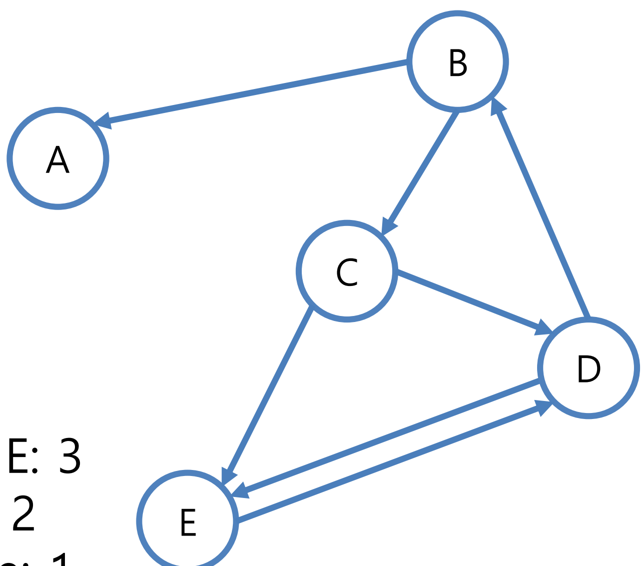
- **Every vertex is connected to the others.**
- **Maximum number of edges for n vertices.**
  - n(n-1)/2 for undirected graph, n(n-1) for directed graph
  - Why?



Degree

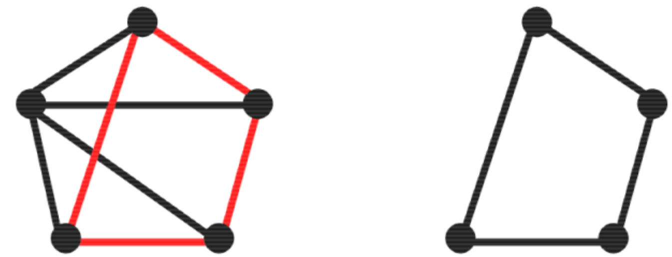
- **Degree of a node (= Degree of a vertex)**
  = # edges
- **In case of a digraph**
  - In-degree: # incoming edges
  - Out-degree: # Outgoing edges

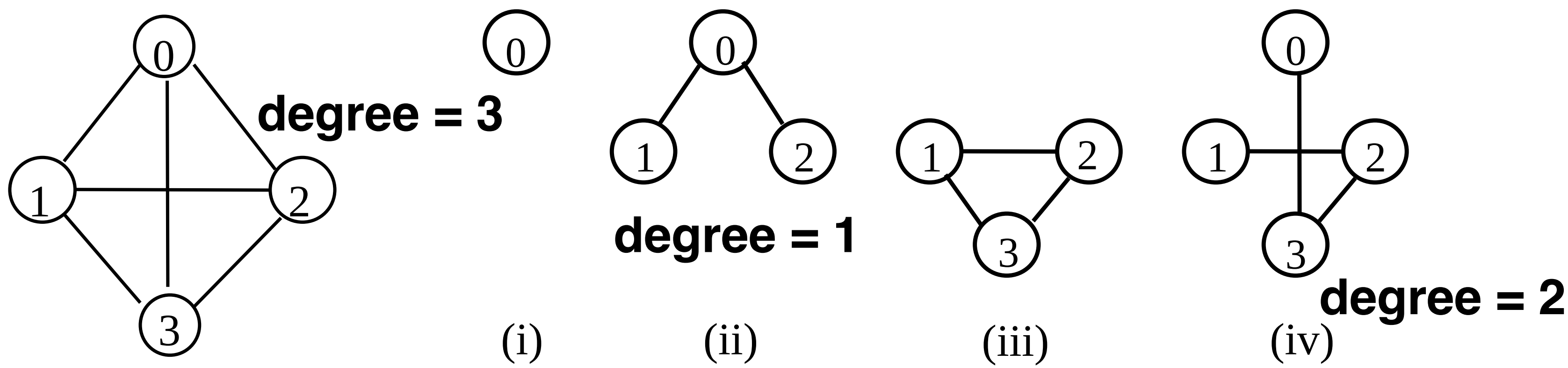
Degree of E: 3
In-degree: 2
Out-degree: 1



Subgraph

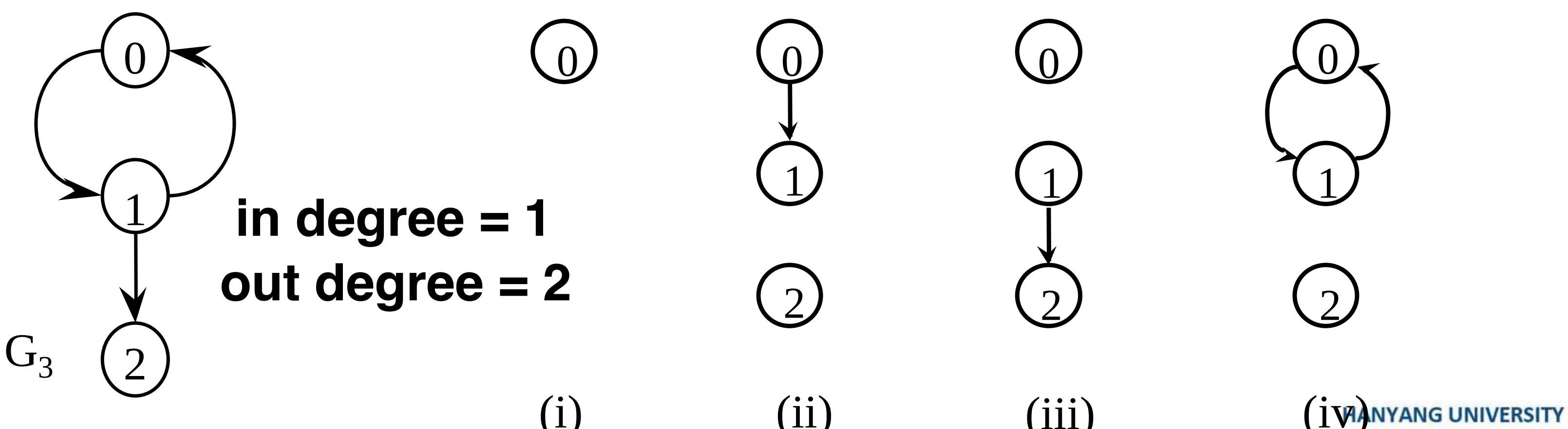
- Any subset of a graph is a subgraph





G_1

Subgraph of G_1



G_3

Subgraph of G_3

Other Terms

Adjacent

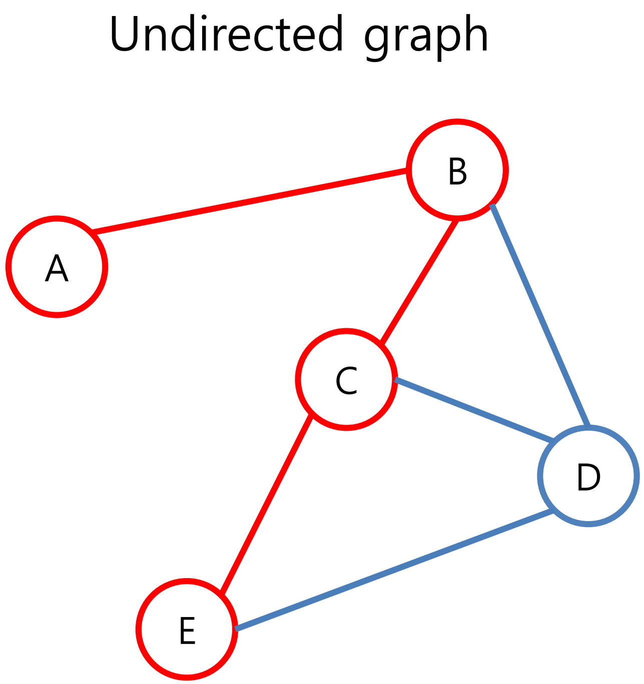
- Undirected: u and v are adjacent
- Directed: x is adjacent to y, y is adjacent from x

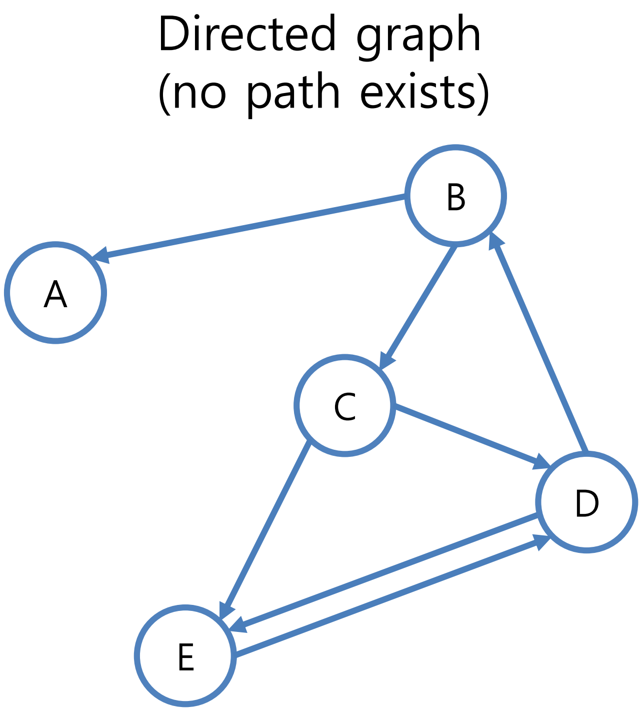
Incident

- Edge (u, v) is incident to u and v
- Edge (x, y) is incident to x and y

Path

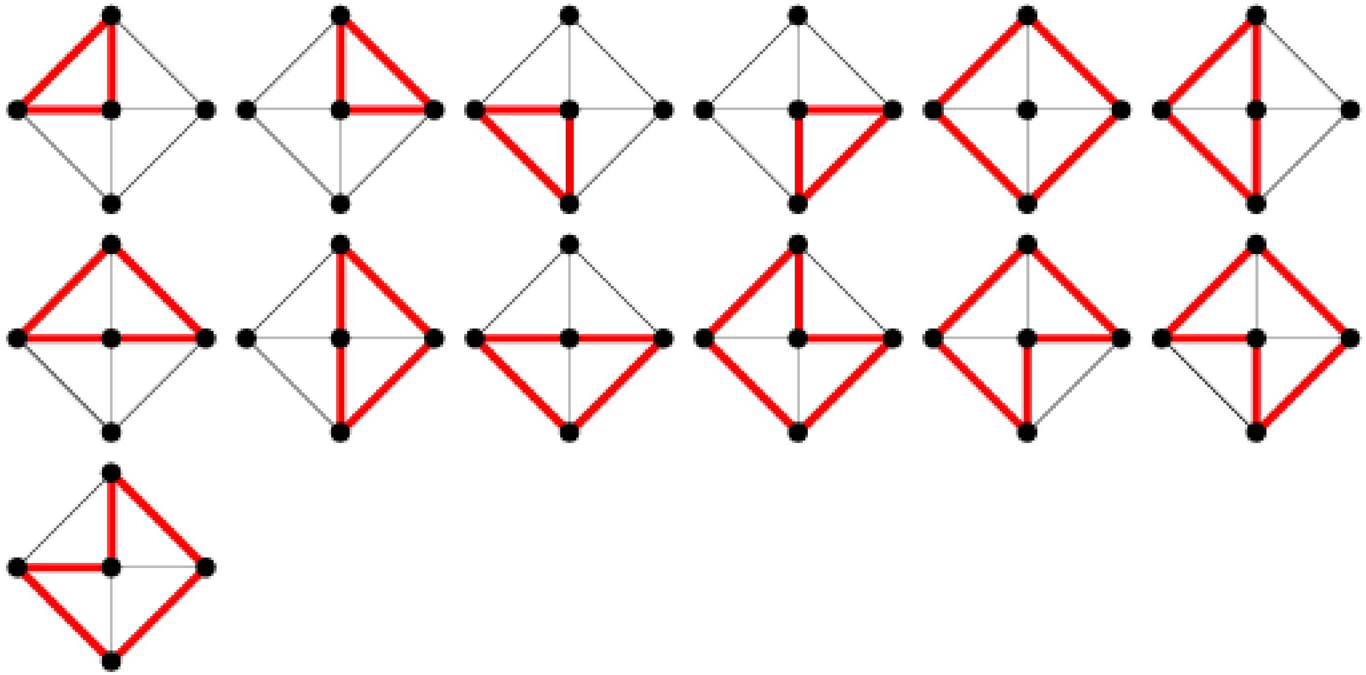
- **Path from A to E: a sequence of vertices A,B,C,E**
  - Not unique





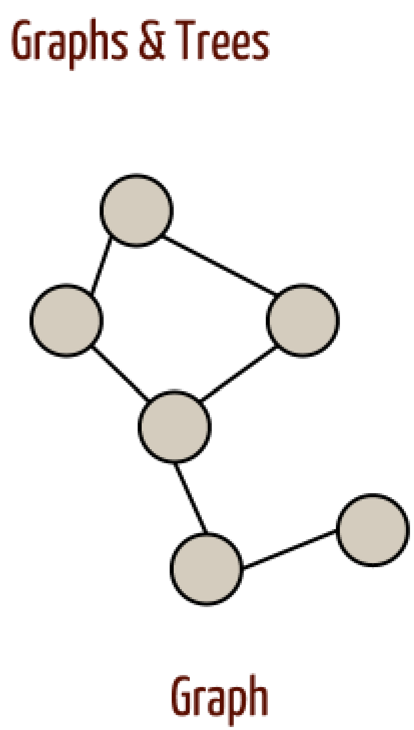
Cycle

- A path where the first and last vertices are the same



Tree is a special case of graph

- Tree = acyclic graph (graph without cycle)



Cicle이 있다

Cicle이 없다

Terms

- **Path: a sequence of vertices (from vertex u to vertex v)**
  - When (u,i_1), (i_1,i_2), ..., (i_k,v) are edges in E(G), a seqeunce of vertices u, i_1, i_2, ..., i_k, v
- **The Length of a path**
  - The number of edges on it
- **Simple path**
  - A path in which all vertices except possibly the first and last are distinct.

- **Simple directed path**
- **Cycle**
  - A simple path in which the first and last vertices are the same

Connected Component

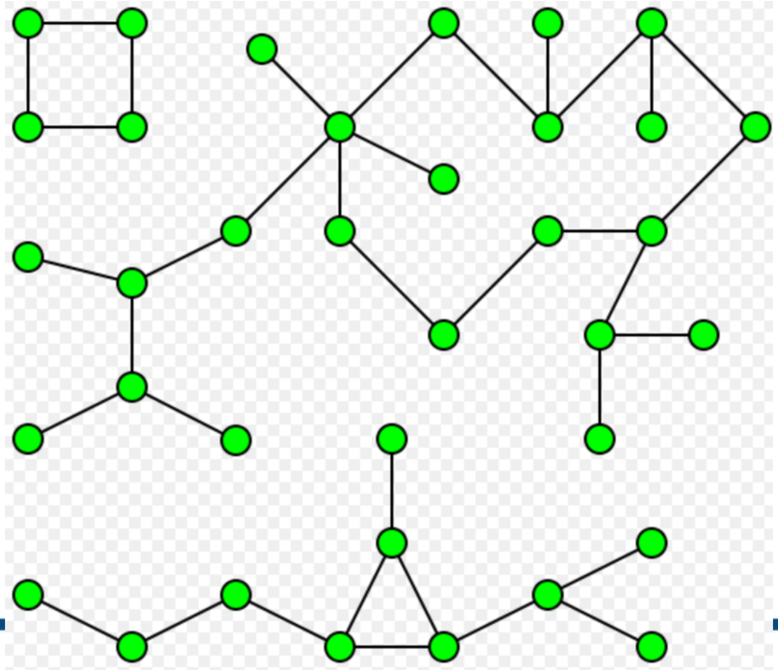
= maximal connected subgraph

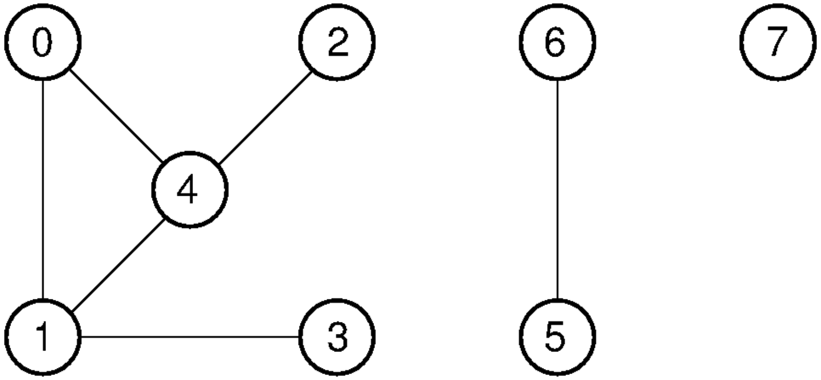
→ **All the elements are connected to each other, and there is no larger connected subgraph that contains all the elements.**

→ For example,

  → A vertex with no incident edges is itself a connected component.

  → A graph that is itself connected has exactly one connected component, consisting of the whole graph.

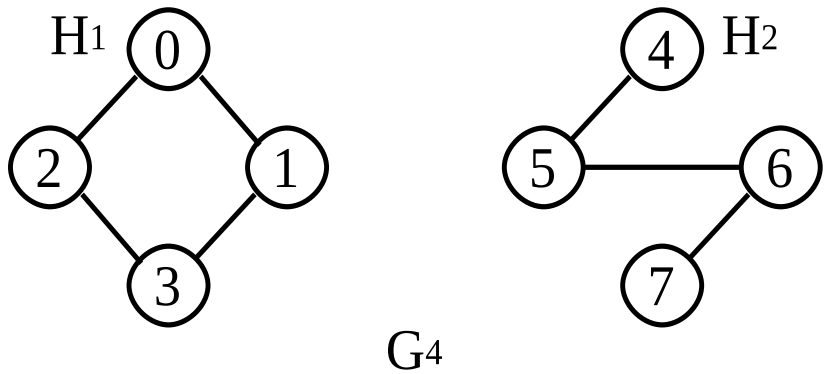




Terms

- **Connected component(or simply a component)**

  : a maximal connected subgraph



A graph with two connected components

- **Strongly connected**

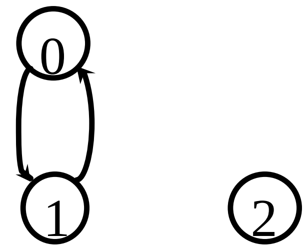
  : for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u.

- **Strongly connected component**

  : a maximal subgraph that is strongly connected

Terms

Strongly connected components of G_3



- **The degree of a vertex : the number of edges incident to that vertex**
- **The in-degree of a vertex v**
  - The number of edges for which v is the head
- **The out-degree of a vertex v**
  - The number of edges for which v is the tail
- **The number of edges**

$$e = (\sum_{0}^{n-1} d_i)/2$$

(a graph with n vertices and e edges, d_i=the degree of vertex i in a graph G)

- **Digraph : directed graph**

Abstract data type Graph

ADT Graph

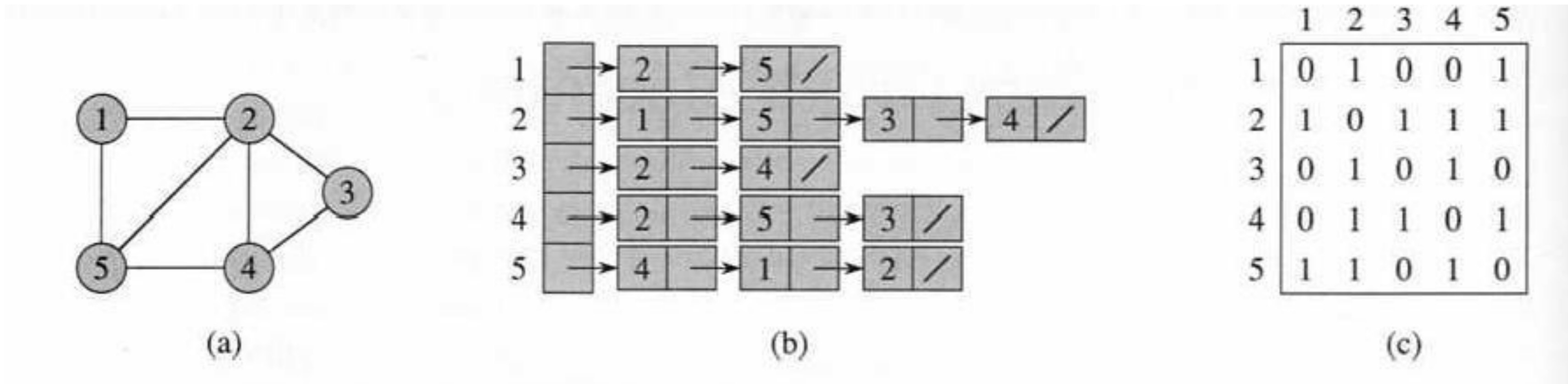
objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices.

functions:

for all *graph* $\in$ *Graph*, v, v_1, and $v_2 \in$ *Vertices*

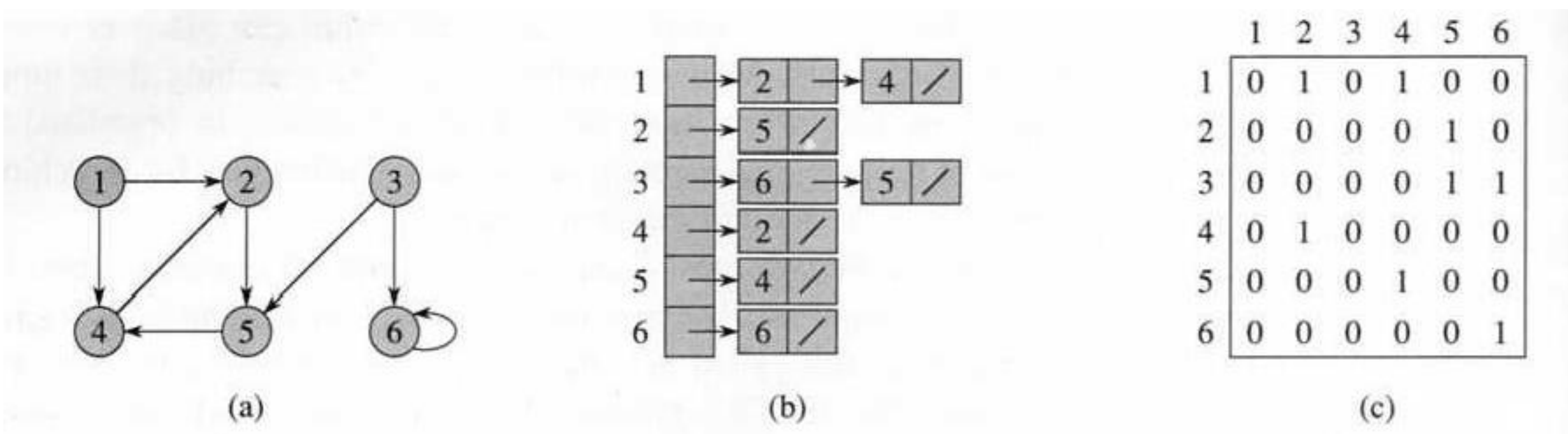
Graph Create()	::= return an empty graph
Graph InsertVertex(graph, v)	::= return a graph with v inserted v has no incident edges
Graph InsertEdge(graph, v_1, v_2)	::= return a graph with a new edge between v1 and v2
Graph DeleteVertex(graph, v)	::= return a graph in which v and all edges incident to it are removed
Graph DeleteEdge(graph, v_1, v_2)	::= return a graph in which the edge(v1, v2) is removed Leave the incident nodes in the graph
Boolean IsEmpty(graph)	::= if(graph = = empty graph) return TRUE else return FALSE
List Adjacent(graph, v)	::= return a list of all vertices that are adjacent to v

Graph representation- Adjacency Matrix vs. Adjacency List



(a) (b) (c)

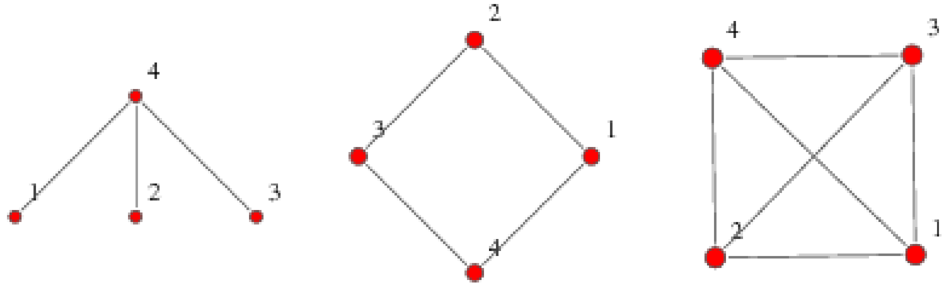
대각선을 중심으로 대칭이다.



(a) (b) (c)

Graph representation- Adjacency matrix

■ For undirected graph

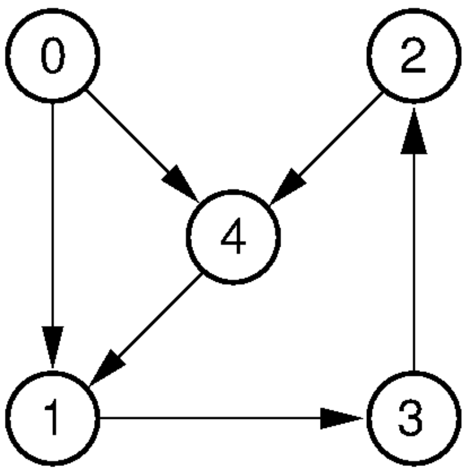


$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

▶ Symmetric, zero-diagonal

Graph representation- Adjacency matrix (cont'd)

■ For digraph



(a)

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

(b)

- ▸ Zero-diagonal, but not necessarily symmetric

Graph representation- Adjacency matrix (cont'd)

- **Adjacency Matrix**
  - G=(V,E): a graph with n vertices (n≥1)
  - Adjacency matrix : a two-dimensional n×n array
  - The edge $(v_i, v_j) \in E(G) \Rightarrow$ a[i][j]=1
  - The edge $(v_i, v_j) \notin E(G) \Rightarrow$ a[i][j]=0
  - The spaced needed to represent a graph using its adjacency matrix : n^2 bits

$$
\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}
$$

G_1

$$
\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 \end{array}
$$

G_3

$$
\begin{array}{c|cccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}
$$

G_4

Complete Graph

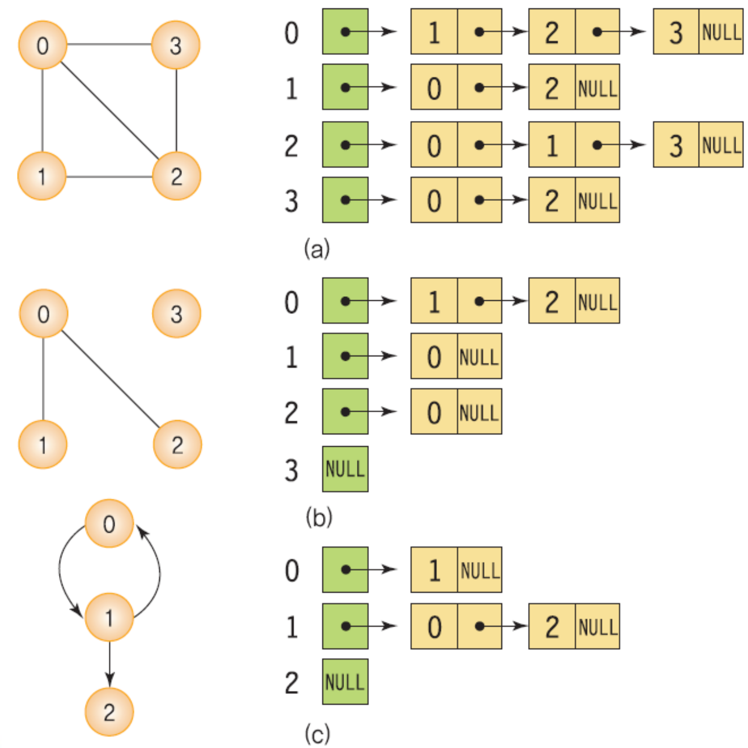
- Undirected graph: the degree of any vertex i is its row sum : $\sum_{j=0}^{n-1} a[i][j]$
- Directed graph: the row sum is the out-degree, and the column sum is the in-degree
- Time complexity of adjacency matrix : at least $O(n^2)$
- Time complexity of sparse graph : $O(e+n)$

Graph representation - Adjacency List

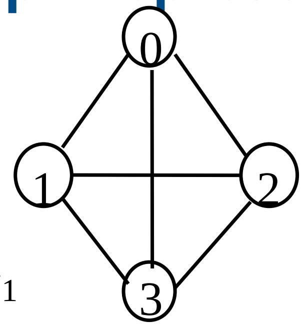
Adjacency Lists

- The n rows of the adjacency matrix are represented as n chains.
  - *For an undirected graph with n vertices* and *e edges*
    - The linked adjacency lists representation requires an array of size n head nodes and 2e list nodes.
  - For a directed graph
    - e list nodes.
- Use sequential lists : array node[]
  - node[i] = gives the starting point of the list for vertex i.
  - Node[n] = n + 2e − 1
  - The vertices adjacent from vertex i

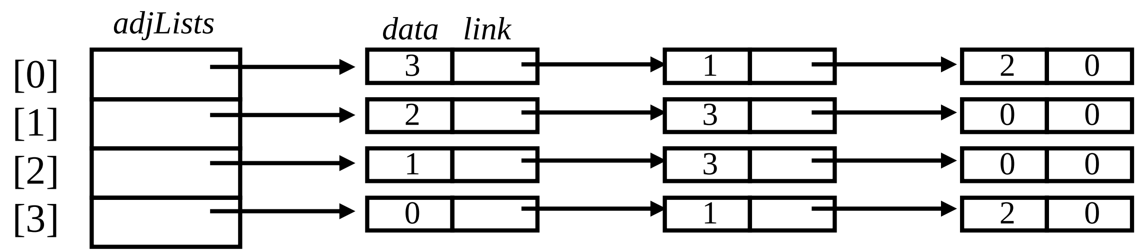
⇨ are stored in node[i], … , node[i+1]−1(0≤i≤n)



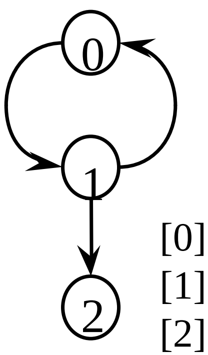
Graph representation - Adjacency List



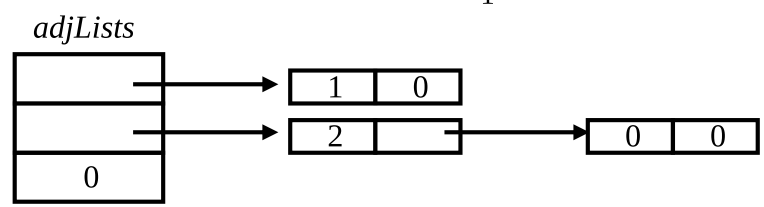
G_1



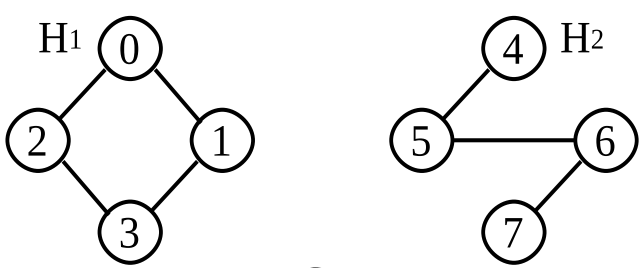
G_1



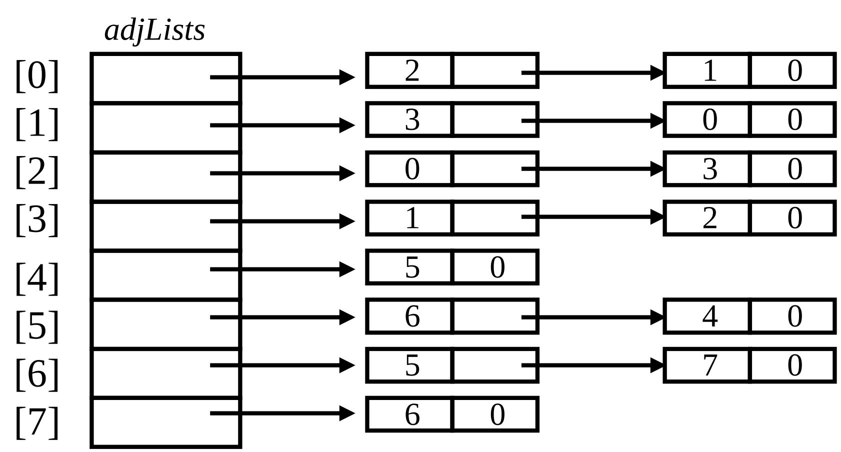
G_3



G_3



G_4



G_4

Weighted Edges

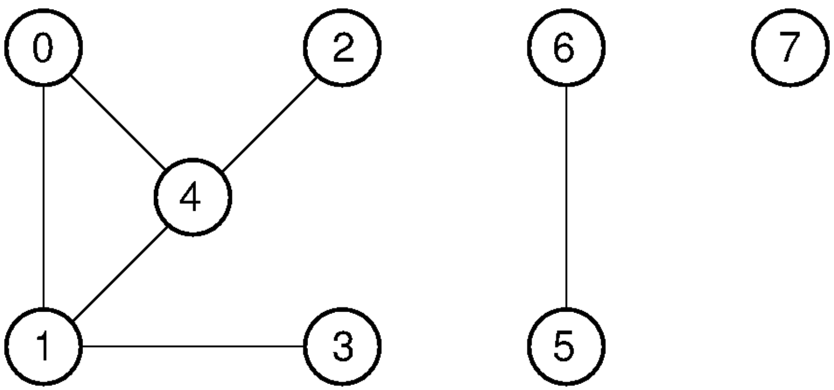
- **The edges of a graph have weights(가중치) assigned to them.**
  - Weights : the distance or the cost of going from one vertex to an adjacent vertex
- **Adjacency matrix : a[i][j] keeps the weights**
- **Adjacency list: the weight may be kept in the list nodes by including an additional field, weight**
- **Network : a graph with weighted edges**

Elementary Graph Operations

■ How can we traverse a graph?

- ▶ Assume that we use an "adjacency list"
- ▶ with a combination of a dynamic array + linked lists

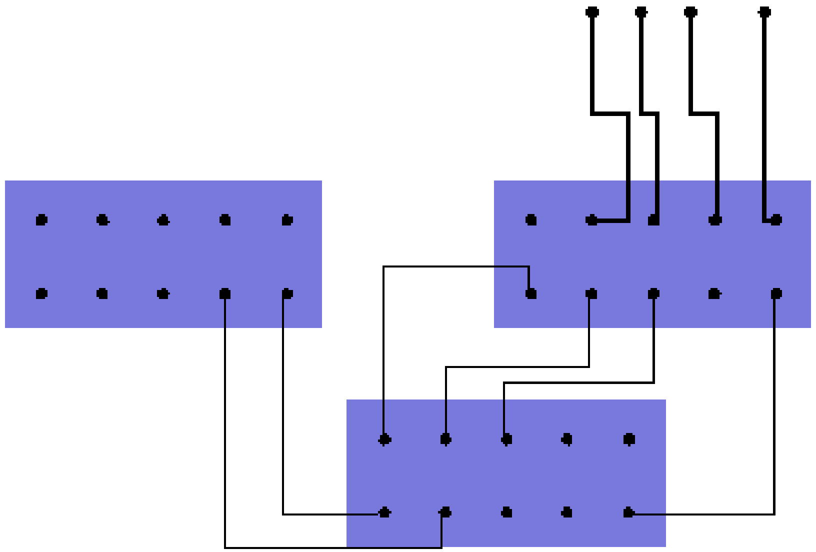
- ▶ We can only visit reachable vertices
  (= If there are more than one connected component, some vertices will not be reachable)



Elementary Graph Operations

■ Graph Search : depth first search and breadth first search

- ▶ Start from one vertex and visit all vertices one by one
- ▶ The most basic operations of the graph
- ▶ Many problems are solved by simply visiting the nodes of the graph.
- ▶ (Example)
  - Whether you can go from one city to another in a road network
  - Whether a specific terminal and another terminal are connected to each other in an electronic circuit



Elementary Graph Operations

- DFS: Depth-First Search
  - It is similar to a preorder tree traversal.
  - Visiting consists of printing the node's vertex field.
- **DFS: Depth-First Search (깊이-우선 탐색)**

(1) Visit the start vertex, v.

(2) Select an unvisited vertex, w, from v's adjacency list.

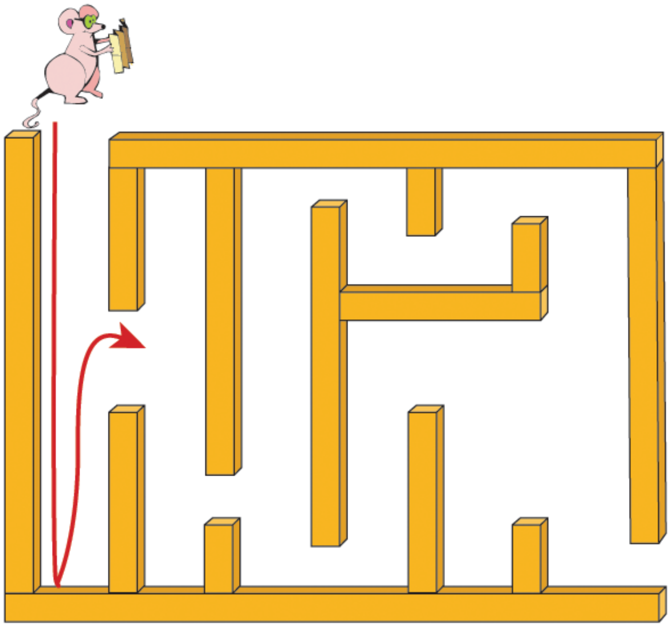
(3) Carry out a depth first search on w. We preserve our current position in v's adjacency list by placing it on a stack.

(4) Eventually, our search reaches a vertex, u, that has no unvisited vertices on its adjacency list. We remove a vertex from the stack and continue processing its adjacency list. Previously visited vertices are discarded; unvisited vertices are visited and placed on the stack.

(5) The search terminates when the stack is empty.

DFS: Depth-First Search

- **깊이 우선 탐색 (DFS: depth-first search)**
  - 한 방향으로 갈 수 있을 때까지 가다가 더 이상 갈 수 없게 되면 가장 가까운 갈림길로 돌아와서 이 곳으로부터 다른 방향으로 다시 탐색 진행
  - 되돌아가기 위해서는 **스택** 필요(순환함수 호출로 묵시적인 스택 이용 가능)



DFS: Depth-First Search

```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs (int v)
{/* DFS of a graph beginning at v */
      nodePointer w;
      visited[v] = TRUE;
      printf("%5d", v);
      for (w = graph[v]; w; w = w → link)
            if (!visited [w→vertex])
                  dfs (w→vertex);
}
```

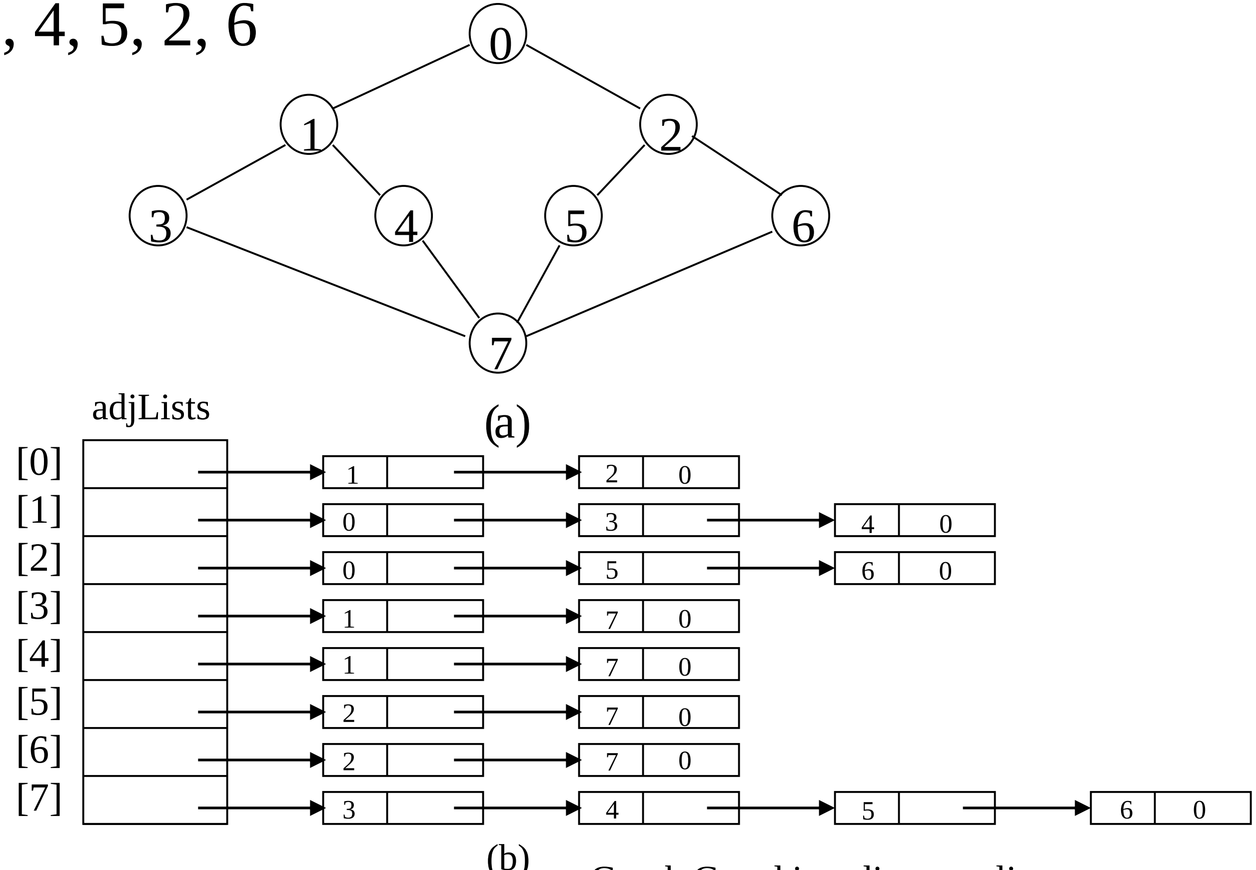
<Depth first search>

DFS: Depth-First Search

Ex 6.1)

– The vertices of G are visited in the following order:

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



Graph G and its adjacency lists

DFS: Depth-First Search

- Graph data type using adjacency matrix

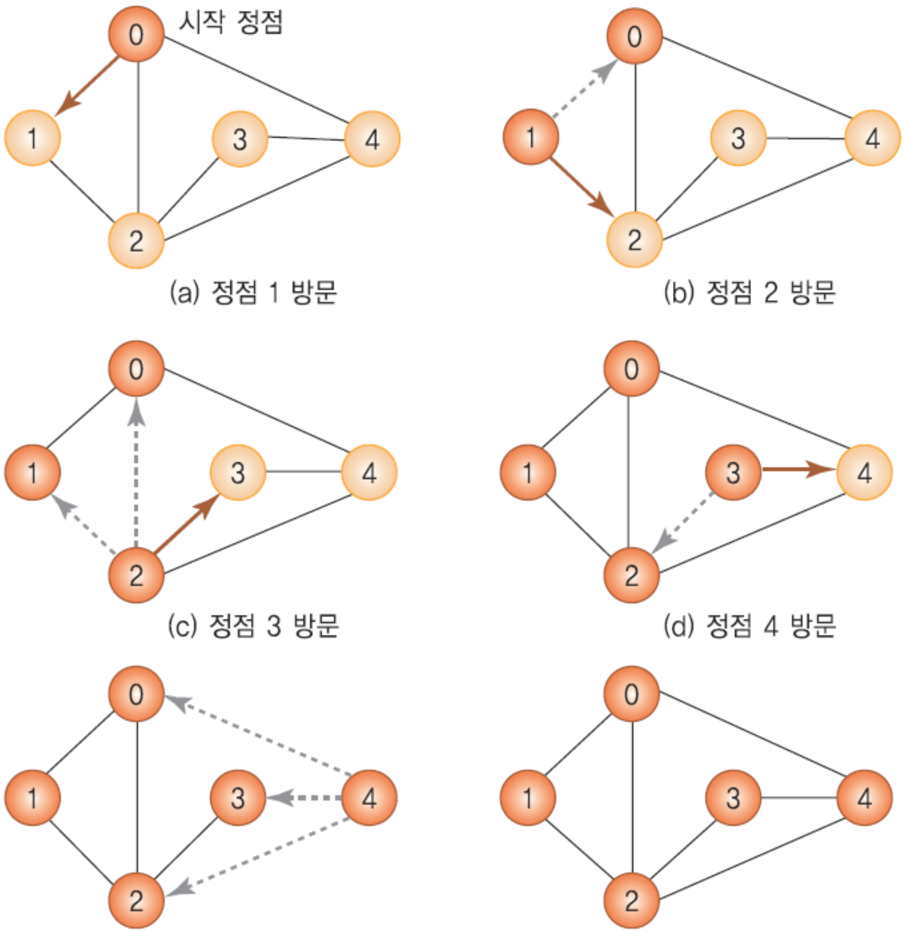
```
#define MAX_VERTICES 50
typedef struct GraphType {
        int n; // the number of vertices
        int adj_mat[MAX_VERTICES][MAX_VETICES];
    } GraphType;
```

- Graph data type using adjacency list

```
#define MAX_VERTICES 50
typedef struct GraphNode {
      int vertex;
      struct GraphNode *link;
  } GraphNode;
```

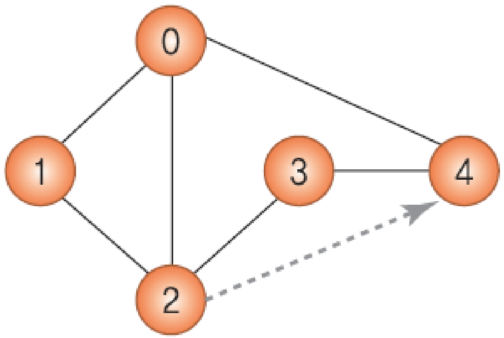
DFS: Depth-First Search

```
depth_first_search(v)
        v를 방문되었다고 표시;
        for all u ∈ (v에 인접한 정점) do
           if (u가 아직 방문되지 않았으면)then depth_first_search(u)
```

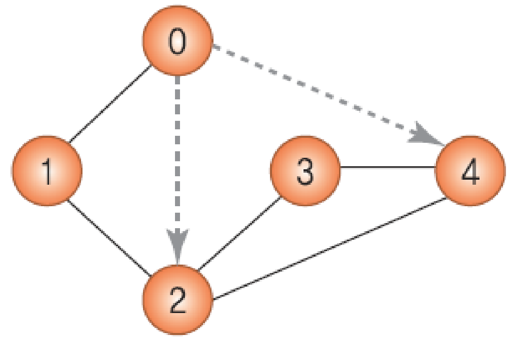


(a) 정점 1 방문
(b) 정점 2 방문
(c) 정점 3 방문
(d) 정점 4 방문
(e) 정점 3으로 backtracking
(f) 정점 2로 backtracking

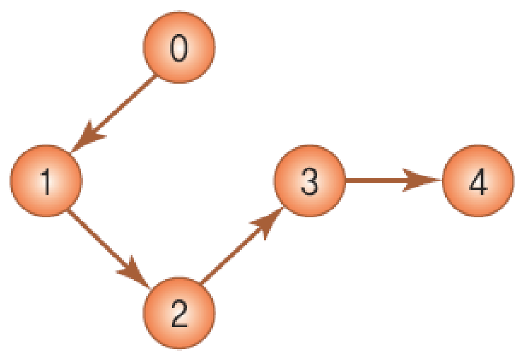
DFS: Depth-First Search



(g) 정점 1로 backtracking



(h) 정점 0으로 backtracking(탐색 종료)



(i) 탐색 결과(방문 순서 0, 1, 2, 3, 4)

DFS: Depth-First Search

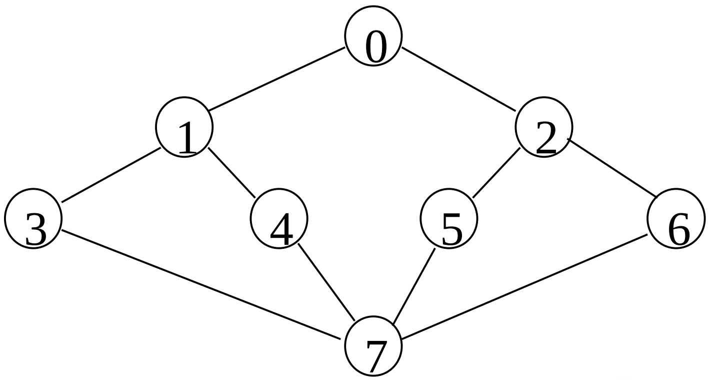
```
// 인접 행렬로 표현된 그래프에 대한 깊이 우선 탐색
int visited[MAX_VERTICES];
void dfs_mat(GraphType *g, int v)
{
   int w;
   visited[v] = TRUE;                // 정점 v의 방문 표시
   printf("%d ", v);                 // 방문한 정점 출력
   for(w=0; w<g->n; w++)             // 인접 정점 탐색
      if( g->adj_mat[v][w] && !visited[w] )
            dfs_mat(g, w);    //정점 w에서 DFS 새로시작
}
```

```
// 인접 리스트로 표현된 그래프에 대한 깊이 우선 탐색
int visited[MAX_VERTICES];
void dfs_list(GraphType *g, int v)
{
   GraphNode *w;
   visited[v] = TRUE;                // 정점 v의 방문 표시
   printf("%d ", v);                 // 방문한 정점 출력
   for(w=g->adj_list[v]; w; w=w->link)  // 인접 정점 탐색
     if(!visited[w->vertex])
            dfs_list(g, w->vertex); //정점 w->vertex에서 DFS 새로시작
}
```

BFS: Breadth-First Search

- **Breadth-First Search(너비 우선 탐색)**
  - Start at vertex v and mark it as visited.
  - Visit each of the vertices on v's adjacency list.
  - When we have visited all the vertices on v's adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on v's adjacency list.
- **To implement BFS, as we visit each vertex we place the vertex in a queue.**
- **Example 6.2**
  - The vertices of G are visited in the following order:

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



BFS: Breadth-First Search

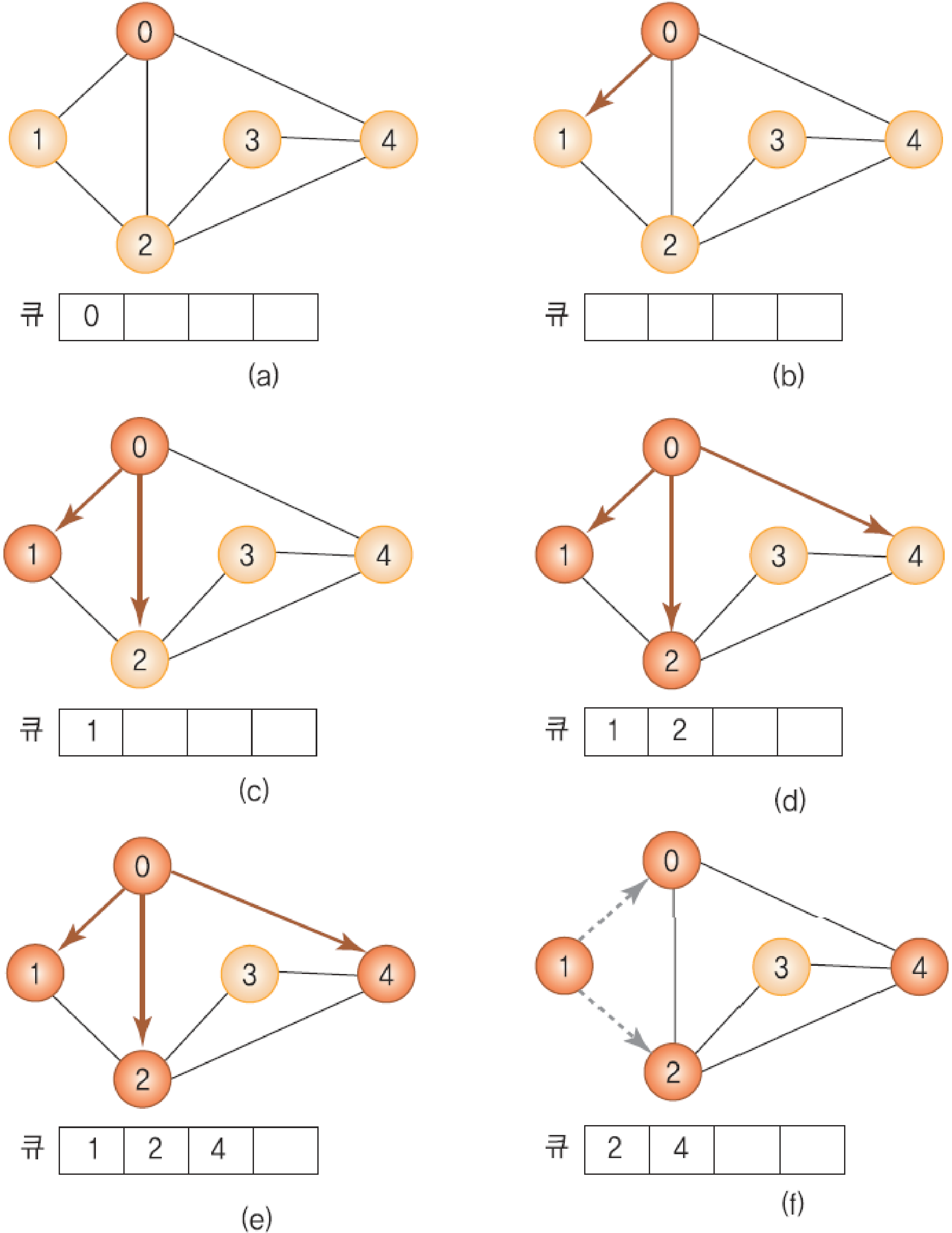
- **너비 우선 탐색(BFS: breadth-first search)**
  - 시작 정점으로부터 가까운 정점을 먼저 방문하고 멀리 떨어져 있는 정점을 나중에 방문하는 순회 방법
  - **큐를 사용하여 구현됨**

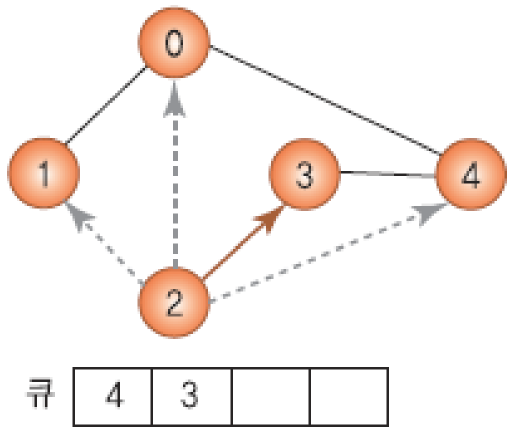
- **너비우선탐색 알고리즘**

```
breadth_first_search(v)
v를 방문되었다고 표시;
큐 Q에 정점 v를 삽입;
while (not is_empty(Q)) do
        큐 Q에서 정점 w를 삭제;
        for all u ∈ (w에 인접한 정점) do
                if (u가 아직 방문되지 않았으면) then  u를 큐 Q에 삽입;
                                                      u를 방문되었다고 표시;
```

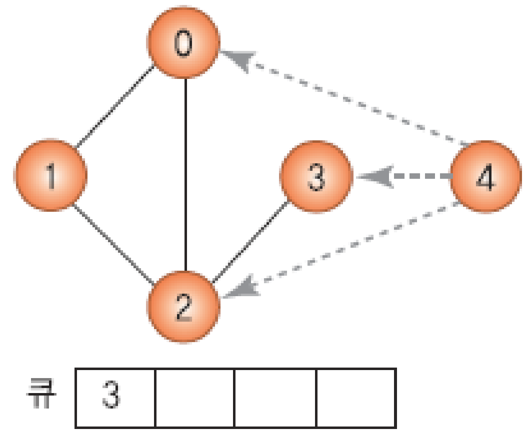
BFS: Breadth-First Search

```
void bfs(int v)
{/* BFS starting at v. the global array visited initialized to 0 */
    node_pointer w;
    queue_pointer front, rear;
    front = rear = NULL;  /* initialize queue*/
    printf("%5d", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front){
        v = deleteq(&front);
        for (w=graph[v]; w; w=w→link)
            if (!visited[w→vertex]) {
                printf("%5d", w→vertex);
                addq(&front, &rear, w→vertex);
                visited[w→vertex] = TRUE;}
```

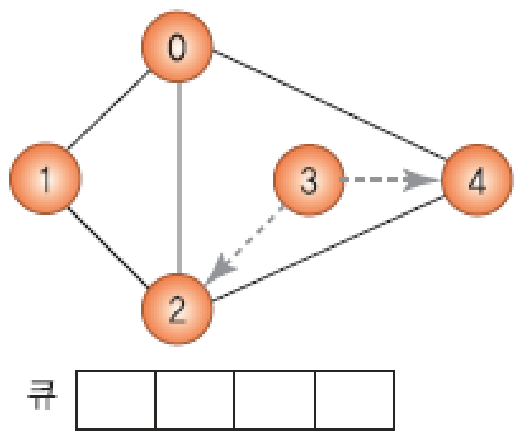

큐 0
(a)
큐
(b)
큐 1
(c)
큐 1 2
(d)
큐 1 2 4
(e)
큐 2 4
(f)



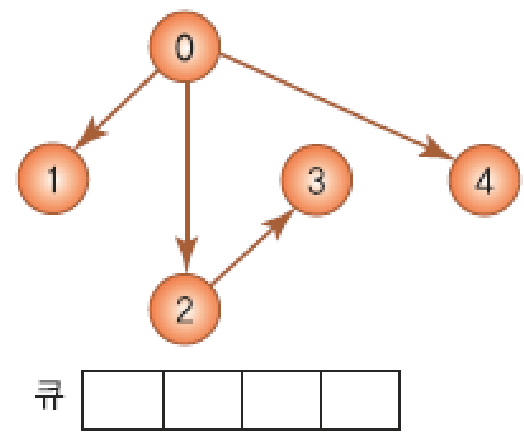
(g)



(h)



(i)



(j) 탐색 결과(방문 순서 0, 1, 2, 4, 3)

BFS program using adjacency matrix

```
void bfs_mat(GraphType *g, int v)
{       int w;
        QueueType q;
        init(&q);                    // 큐 초기화
        visited[v] = TRUE;           // 정점 v 방문 표시
        printf("%d ", v);            // 정점 출력
        enqueue(&q, v);              // 시작 정점을 큐에 저장
        while(!is_empty(&q)){
        v = dequeue(&q);             // 큐에 정점 추출
        for(w=0; w<g->n; w++)        // 인접 정점 탐색
                if(g->adj_mat[v][w] && !visited[w]){
                        visited[w] = TRUE; // 방문 표시
                        printf("%d ", w);  // 정점 출력
                        enqueue(&q, w);    // 방문한 정점을 큐에 저장
                }
        }
}
```

BFS program using adjacency list

```
void bfs_list(GraphType *g, int v)
{         GraphNode *w;
          QueueType q;
          init(&q);                    // 큐 초기화
          visited[v] = TRUE;           // 정점 v 방문 표시
          printf("%d ", v);            // 정점 v 출력
          enqueue(&q, v);              // 시작정점을 큐에 저장
          while(!is_empty(&q)){
                    v = dequeue(&q);                         // 큐에서 정점 추출
                    for(w=g->adj_list[v]; w; w = w->link) //인접 정점 탐색
                              if(!visited[w->vertex]){     // 미방문 정점 탐색
                              visited[w->vertex] = TRUE;   // 방문 표시
                              printf("%d ", w->vertex);    // 정점 출력
                              enqueue(&q, w->vertex); // 방문한 정점을 큐에 삽입
                    }
          }
}
```

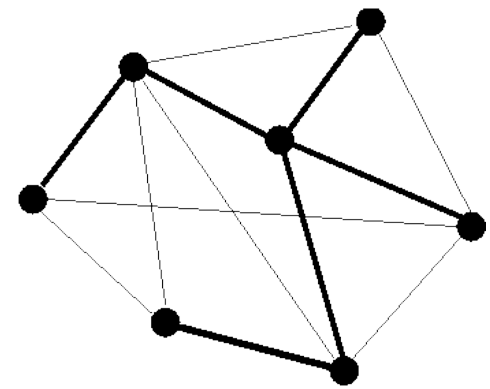
Connected component

- **The problem of determining whether or not an undirected graph is connected.**
  - We can implement this operation by simply calling either *DFS(v)* or *BFS(v)*

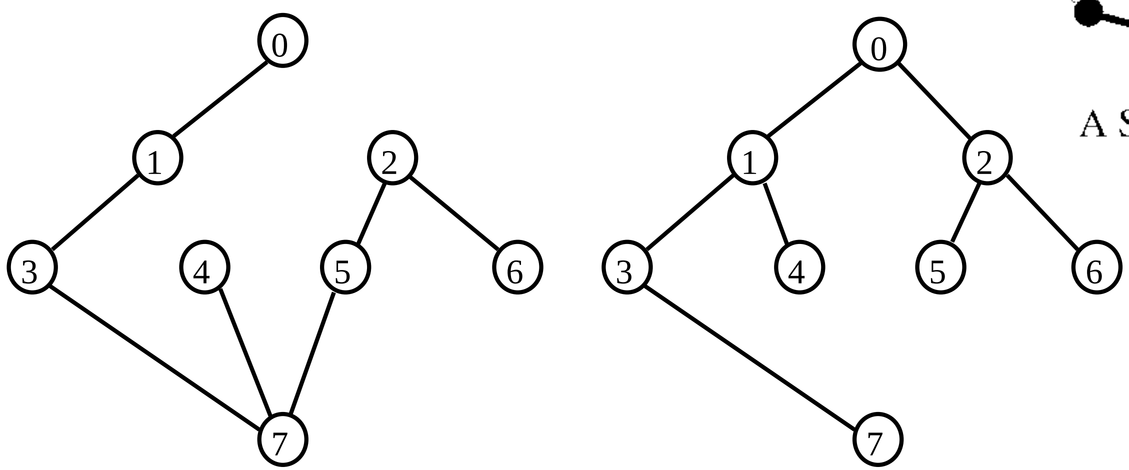
```
void connect(void)
{ /* determine the connected components of a graph */
    int i;
    for (i=0; i<n; i++)
        if (!visited[i]) {
         dfs(i);
         printf("\n");
         }
}
```

Spanning tree(1/2)

- When graph G is connected, a DFS or BFS starting at any vertex visits all the vertices in G.
- Spanning tree: a tree that is a subgraph of a graph, containing all the vertices.
  - depth-first spanning tree
  - breadth-first spanning tree



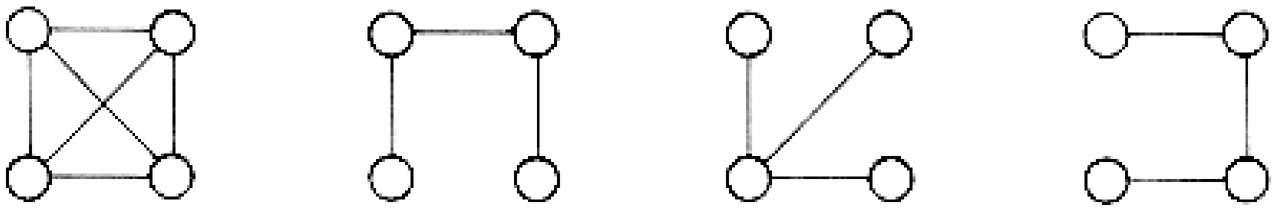
A Spanning Tree



(a) DFS(0) spanning tree (b) BFS(0) spanning tree

Spanning tree(2/2)

- **Spanning tree: any tree that consists solely of edges in G that includes all the vertices in G**

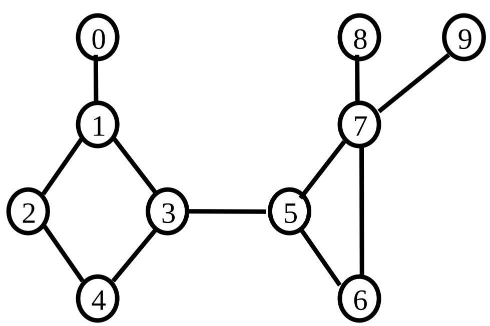


<A complete graph and three of its spanning trees>

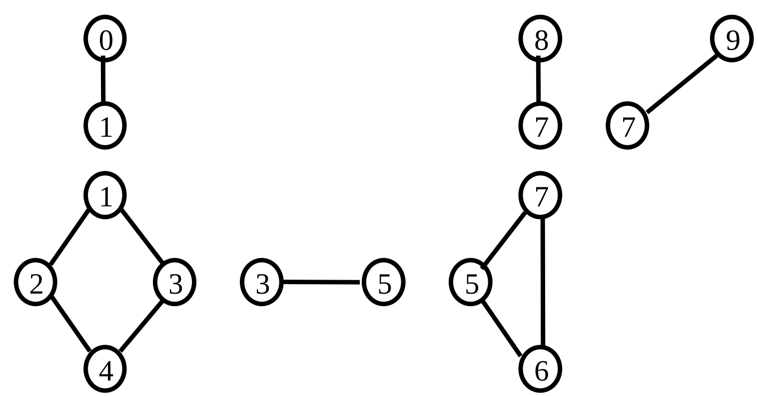
- **Spanning tree of G′ is a minimal subgraph of G such that**
  - V(G′) = V(G) and G′ is connected
  - Any connected graph with n vertices must have at least n-1 edges. All connected graphs with n-1 edges are trees

Biconnected Component (1/2)

- **Articulation point**
  - A vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph, G′, that has at least two connected components.
- **biconnected graph**
  - A connected graph that has no articulation points.
- **biconnected component**
  - maximal biconnected subgraph
  - “Maximal” means that G contains no other subgraph that is both biconnected and properly contains H.



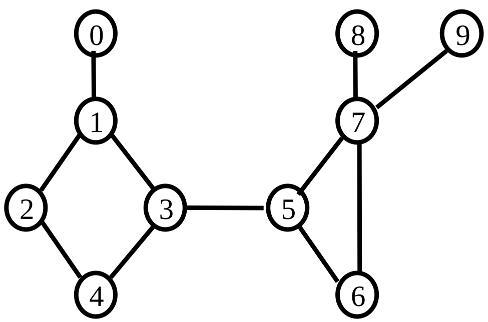
Connected graph



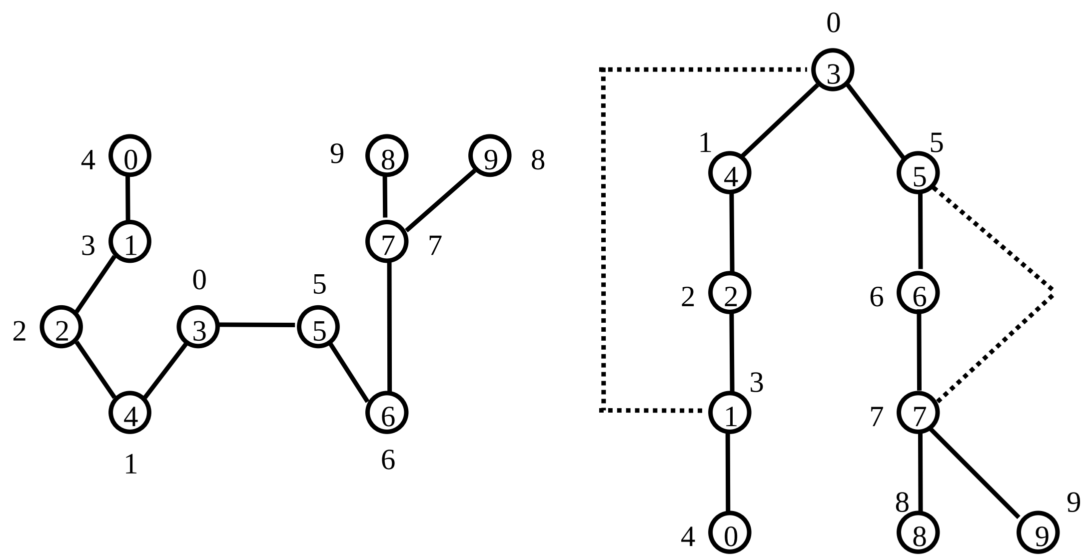
6 Biconnected components

Biconnected Component (2/2)

- By using any depth-first spanning tree of G, we find the biconnected components of a connected undirected graph.



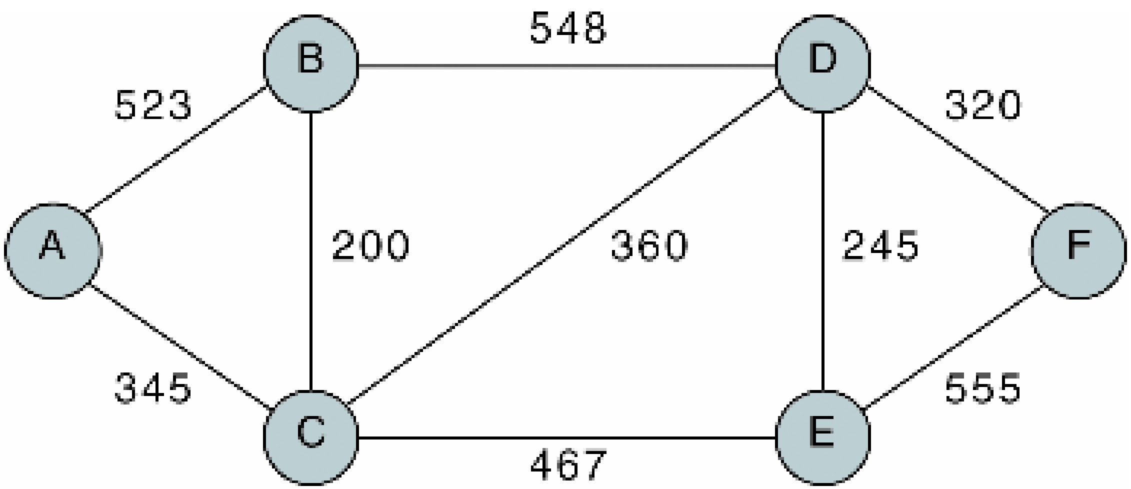
Connected graph



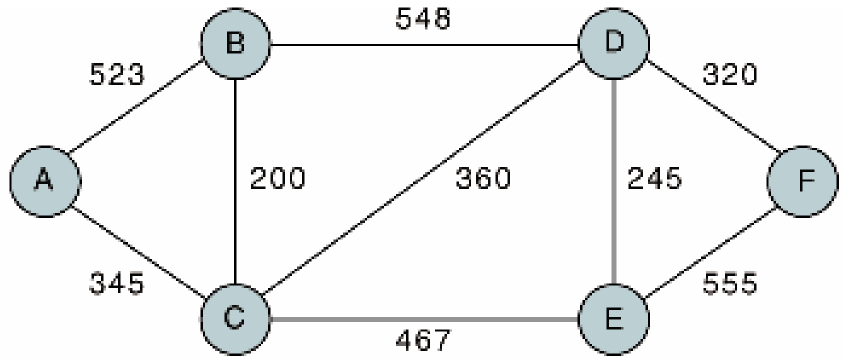
Biconnected components

Weighted Graph (1/2)

- **Weighted graph (network): a graph whose edges are weighted**



Weighted Graph (2/2)

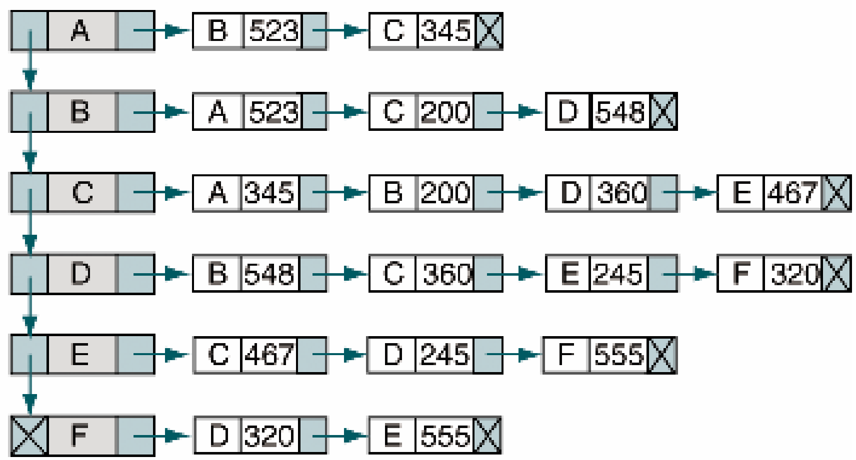


A
B
C
D
E
F

Vertex vector

	A	B	C	D	E	F
A	0	523	345	0	0	0
B	523	0	200	548	0	0
C	345	200	0	360	467	0
D	0	548	360	0	245	320
E	0	0	467	245	0	555
F	0	0	0	320	555	0

Adjacency matrix

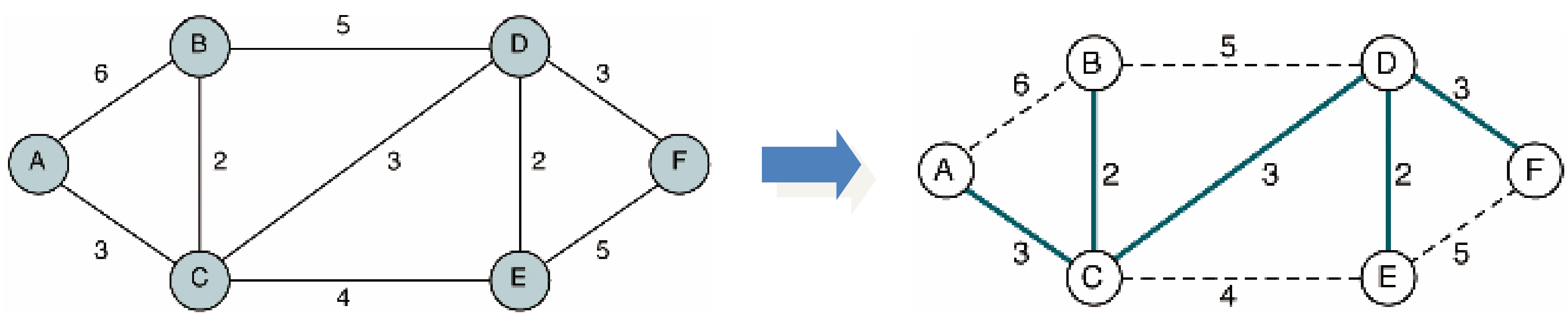


Vertex list

Adjacency list

Minimum Cost Spanning Trees (1/2)

- **Minimum (cost) spanning tree: spanning tree of least cost (sum of weights)**
  - Every vertices are included
  - Total edge weight is minimum possible



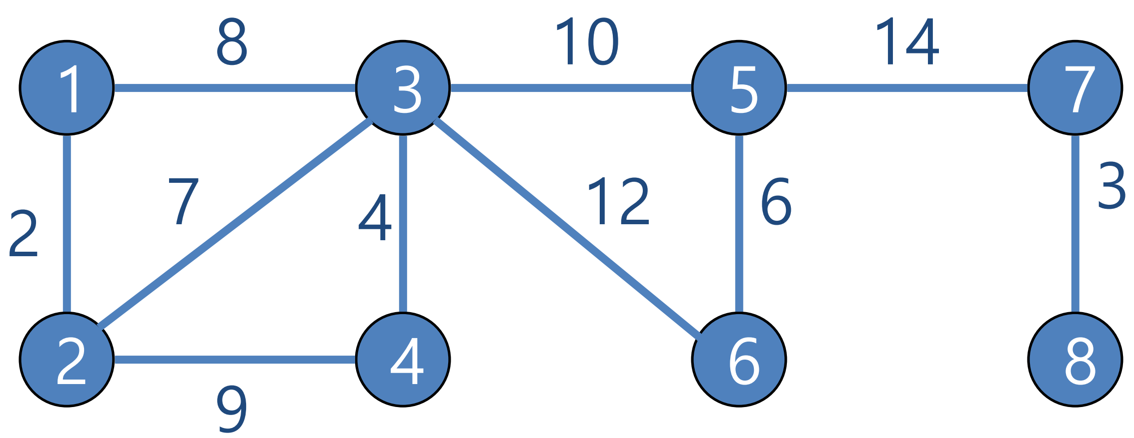
Minimum Cost Spanning Trees (2/2)

- **Minimum spanning tree algorithms**
  - **Kruskal's algorithm**
  - **Prim's algorithm**
  - Sollin's algorithm

- **For spanning trees, we use a least cost criterion.**
  - Must use only edges within the graph.
  - Must use exactly n-1 edges.
  - May not use edges that would produce a cycle.

Kruskal's Algorithm(1/3)

- Build a minimum cost spanning tree T by adding edges to T one at a time.
- Check edges in nondecreasing order of weights.
- An edge is added to T if it does not form a cycle with the edges that are already in T.

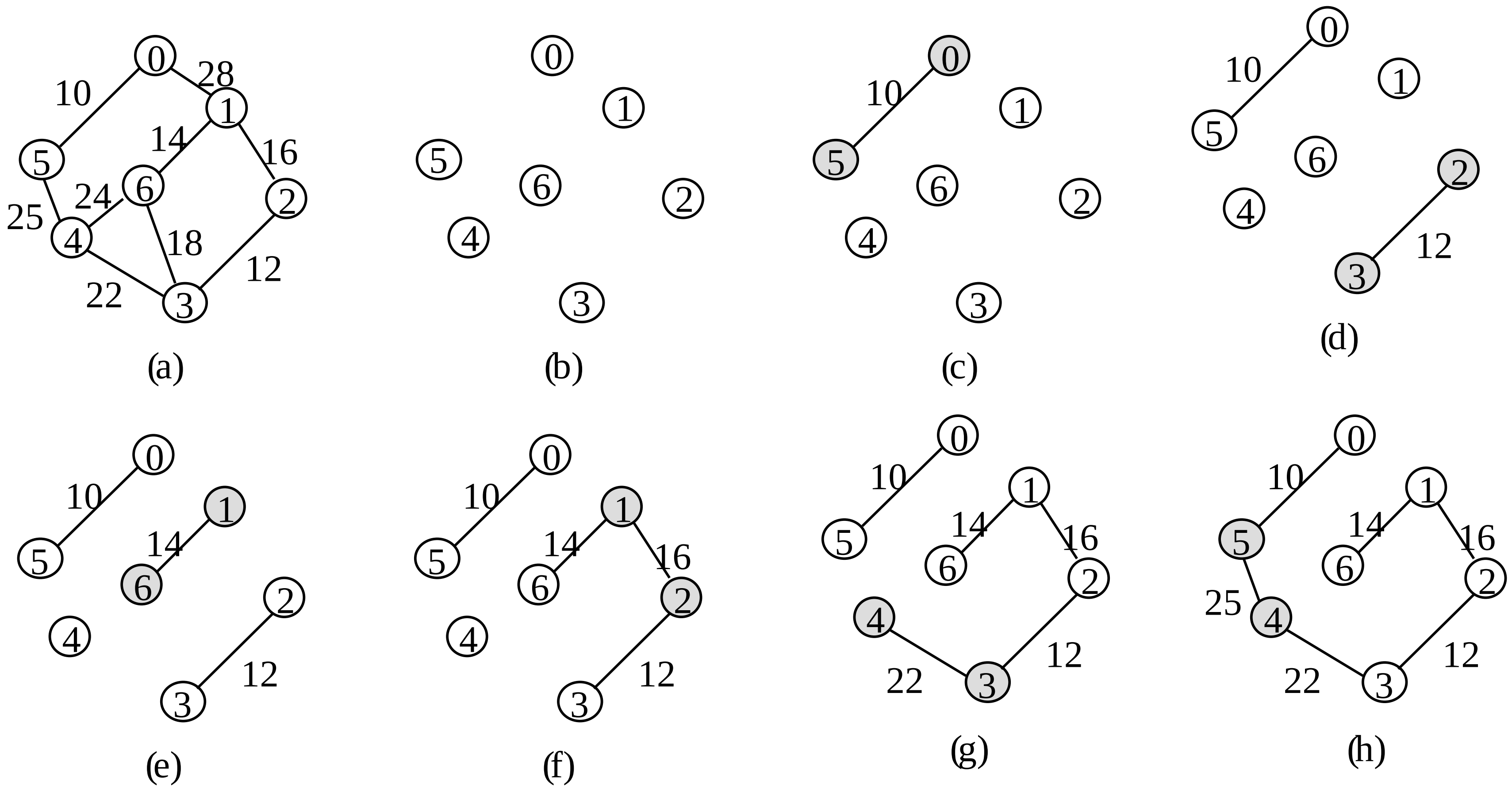


Total cost:

[order of selection]
1. (1, 2)
2. (7, 8)
3. (3, 4)
4. (5, 6)
5. (2, 3)
6. (1, 3) not selected
7. (2, 4) not selected
8. (3, 5)
9. (3, 6) not selected
10. (5, 7)

Kruskal's Algorithm(2/3)

- Ex 6.3



<Stages in Kruskal's algorithm>

Kruskal's Algorithm(3/3)

```
T = { };
while ((T contains less than n-1 edges) && (E is not empty)) {
        choose a least cost edge (v,w) from E ;
        delete  (v,w)  from E ;
    if ((v,w) does not create a cycle in T)
          add (v, w) to T;
    else
          discard (v,w);
}
if (T contains fewer than n-1 edges)
          printf ("No spanning tree");
```

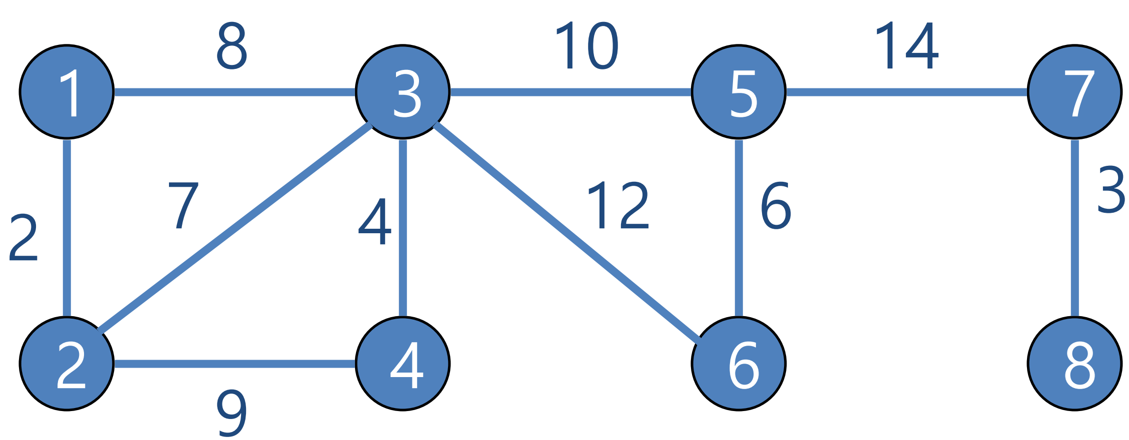
Kruskal algorithm

■ Theorem 6.1

- Let G be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree.

Prim Algorithm (1/6)

- **Grow the tree by adding a vertex at a time**
  - Select an edge with minimum weight, among those connected to the tree
  - You can start with an arbitrary vertex



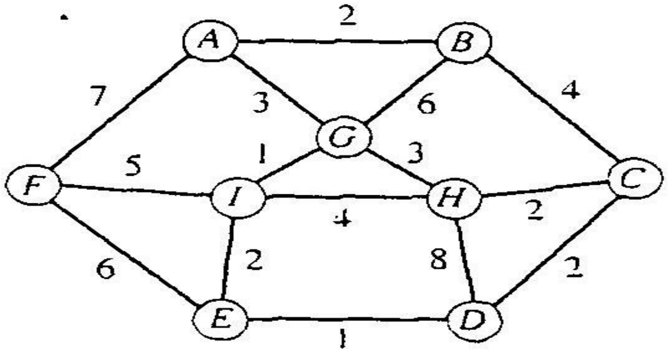
Total cost:

[order of selection]
0. initial vertex: 5
1. (5, 6)
2. (3, 5)
3. (3, 4)
4. (2, 3)
5. (1, 2)
6. (5, 7)
7. (7, 8)

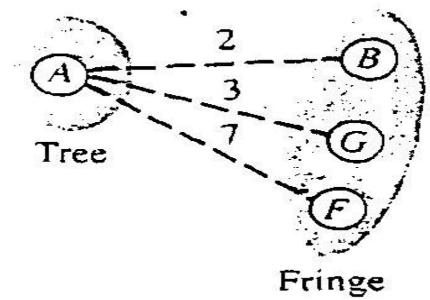
Prim Algorithm (2/6)

- Starting with T = {}, add an edge into T in stages.
- At each stage, …
  - Partitioned edges into
    - T: tree edges
    - V: edges not included in T
  - Select the least cost edge (t, v) such that
    - $t \in T$ and $v \in V$
  - Repeat n-1 times
- Then, Prim's algorithm produces a MCST (Minimum Cost Spanning Trees)

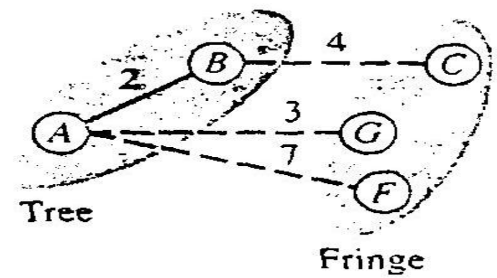
Prim Algorithm (3/6)



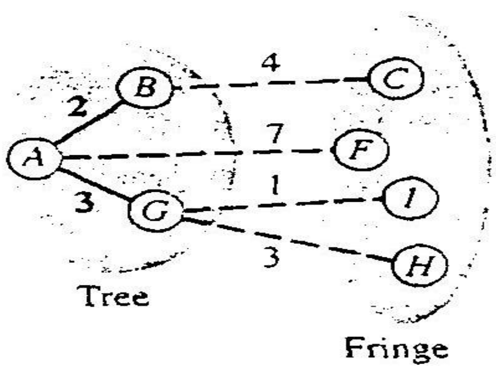
(a) A weighted graph



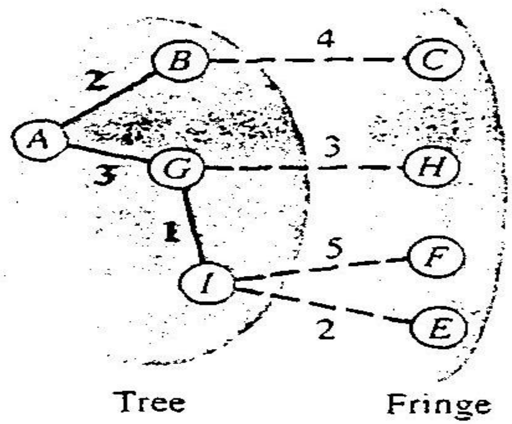
(b) After selection of the starting vertex



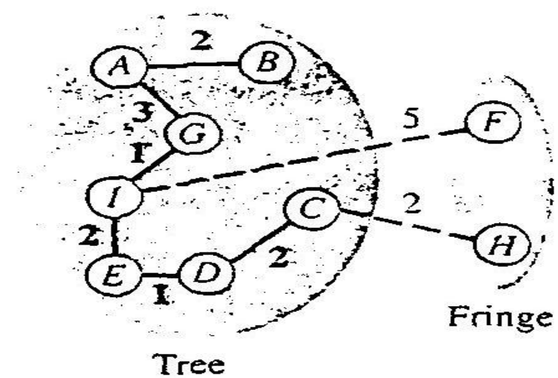
(c) BG was considered but did not replace AG as a candidate.



(d) After AG is selected and fringe and candidates are updated



(e) IF has replaced AF as a candidate.



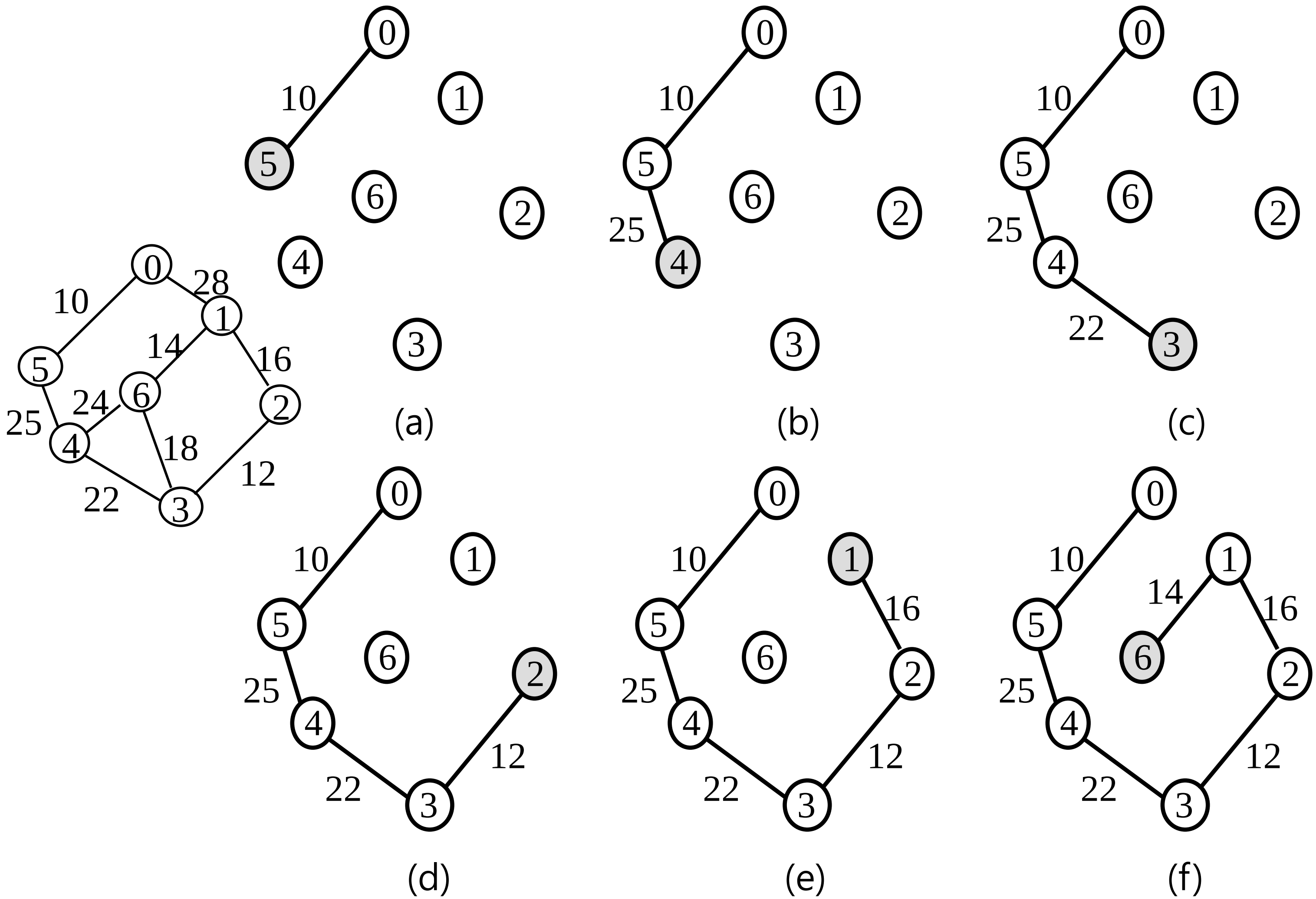
(f) After several more passes: The two candidate edges will be put in the tree.

Prim Algorithm (4/6)

```
T = {  } ;  /* T is set of tree edges */
TV = { 0 } ; /*TV is set of tree vertices */
/* start with vertex 0 and no edges */
while (T contains less than n-1 edges)  {
    let (u ,v) be a least cost edge such that u ∈ TV and v ∉ TV;
    if (there is no such edge )
      break;
    add v to TV;
    add (u , v) or (v ,u) to T ;
}
if(T contains fewer than n -1 edges)
    printf(“No spanning tree₩n”);
```

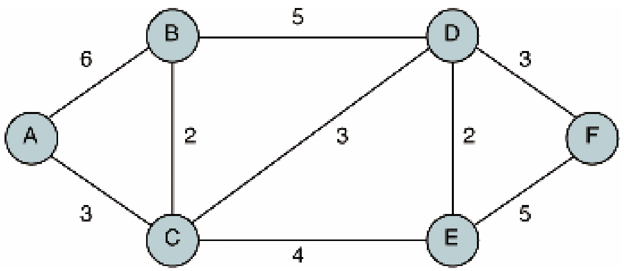
< Prim algorithm >

Prim Algorithm (5/6)

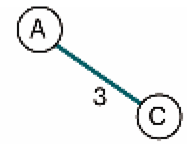


<Stages in Prim’s algorithm>

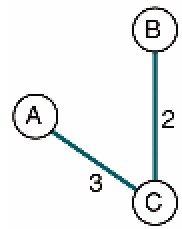
Prim Algorithm (6/6)



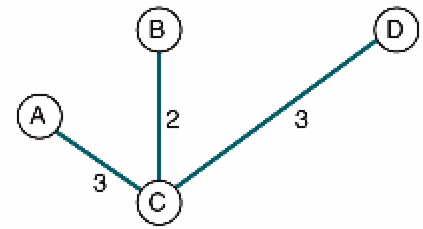
(a) Insert first vertex



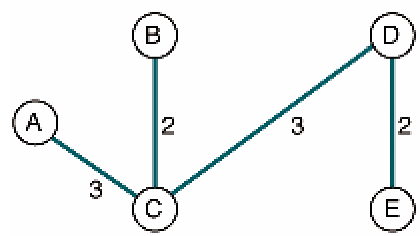
(b) Insert edge AC



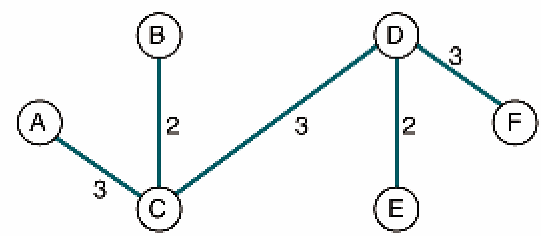
(c) Insert edge BC



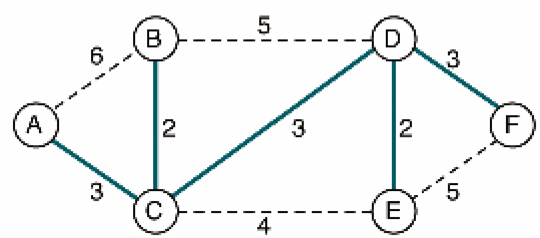
(d) Insert edge CD



(e) Insert edge DE



(f) Insert edge DF



(g) Final tree in the graph

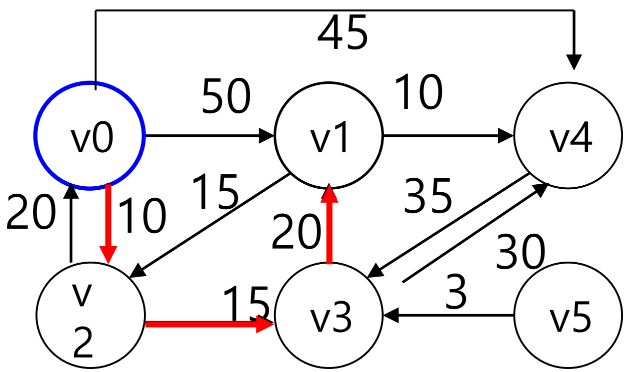
Greedy Algorithm

- **Greedy method**
  : Solves a problem by selecting greedy choices
  - We construct an optimal solution(the locally optimal choice) in stages.
  - At each stage, we make a decision that is the best decision using some criterion at this time.
  - Since we cannot change this decision later, we make sure that the decision will result in a feasible solution.

- **Kruskal's and Prim's algorithms are greedy algorithms**

Shortest Path Algorithm (1/5)

- Given a (directed) graph G=(V,E), determine a shortest path from v0 to each of the remaining vertices of G

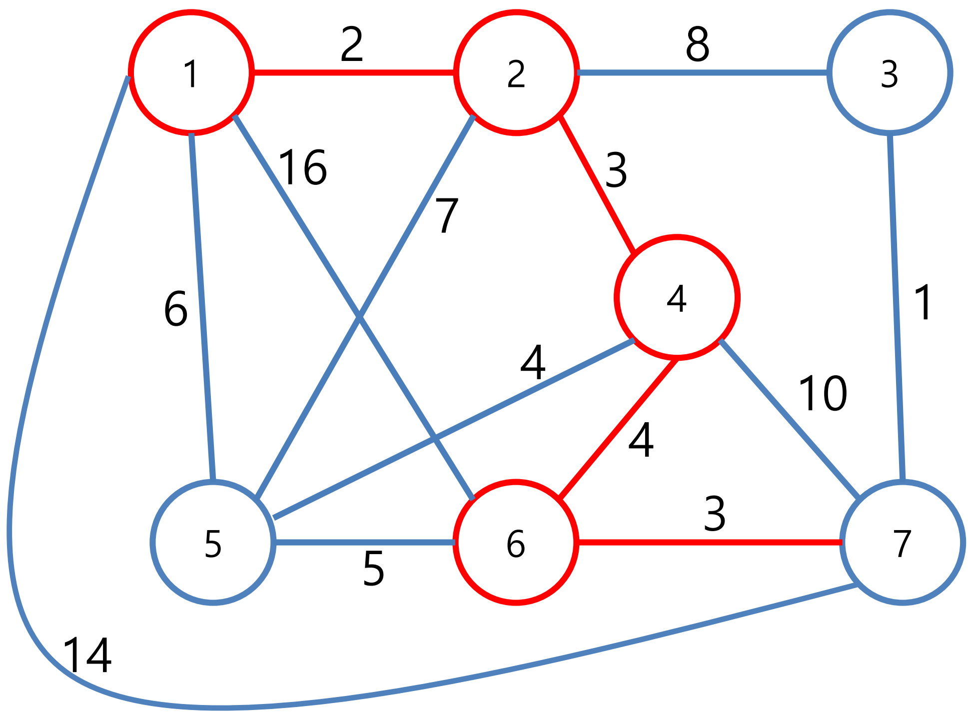


destination	path	Length
v2	v0v2	10
v3	v0v2v3	25
v1	v0v2v3v1	45
v4	v0v4	45
v5	No path	infinite

shortest paths starting from v0

Shortest Path Algorithm (2/5)

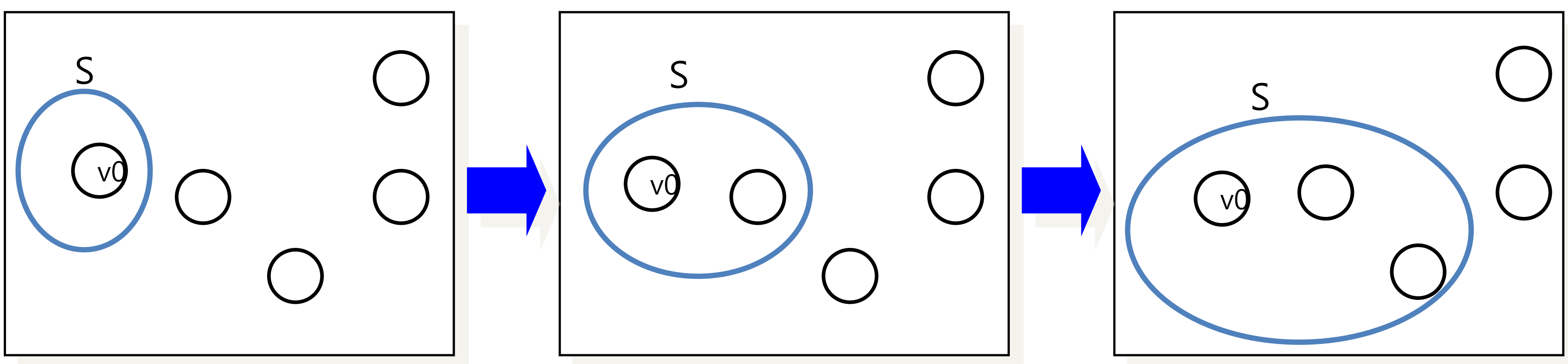
- Finding the shortest path from 1 to 7
- Greedy algorithm does not work.



2+3+4+3=12 > 11=2+8+1

Shortest Path Algorithm(3/5)

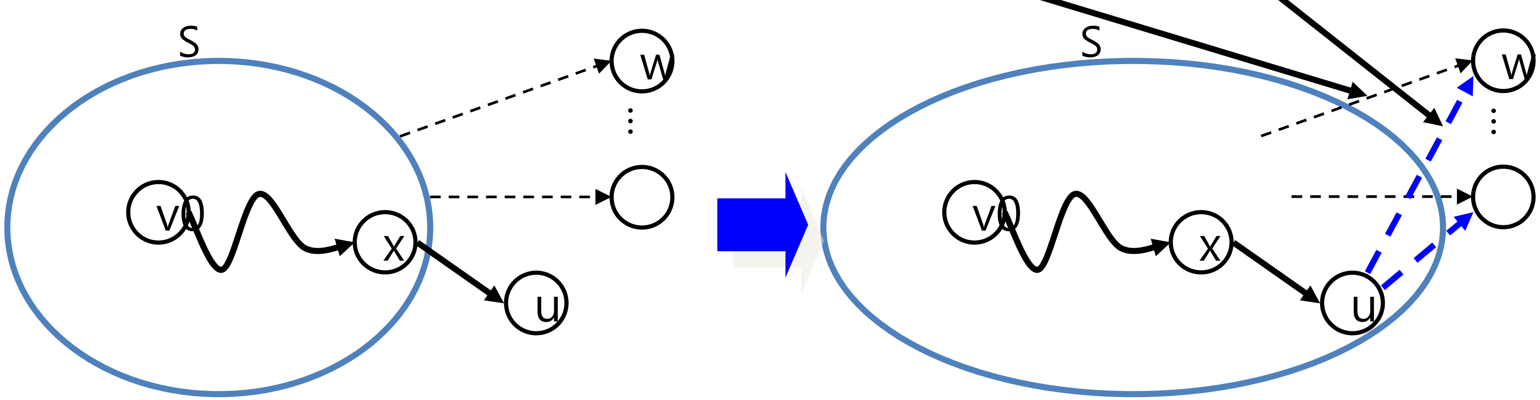
- **Idea: starting from a source vertex v0, find the shortest paths to other vertices <u>in stages</u>**
  - Let S: set of vertices to which the shortest path from v0 is found.
    - Initially, S = { v0 }
  - Add a vertex into S at each stage



Shortest Path Algorithm(4/5)

■ At each stage …

- ▶ Add a vertex u whose ***distance*** is **minimum**
  - ***distance***[w] = length of shortest path from v0 to w, **going through vertices in S**
  - Initially, ***distance***[w] = ***cost***[v0, w] (edge weight)
- ▶ After adding u, adjust distance of vertices not in S
  - ***distance***[w] = min(***distance***[w], ***distance***[u] + ***cost***[u,w])



Shortest Path Algorithm(5/5)

- **How about finding shortest paths for all the destinations? (from a certain starting node)**

- **Dijkstra's algorithm**
  → Find all the shortest paths incrementally
  - Only works for nonnegative weights.
  - Only care about the minimum weight of the previous step when we are finding those with length k
  - $\min(a \rightarrow b) = \min_{\{c\}} (\min(a \rightarrow c) + (c \rightarrow b))$

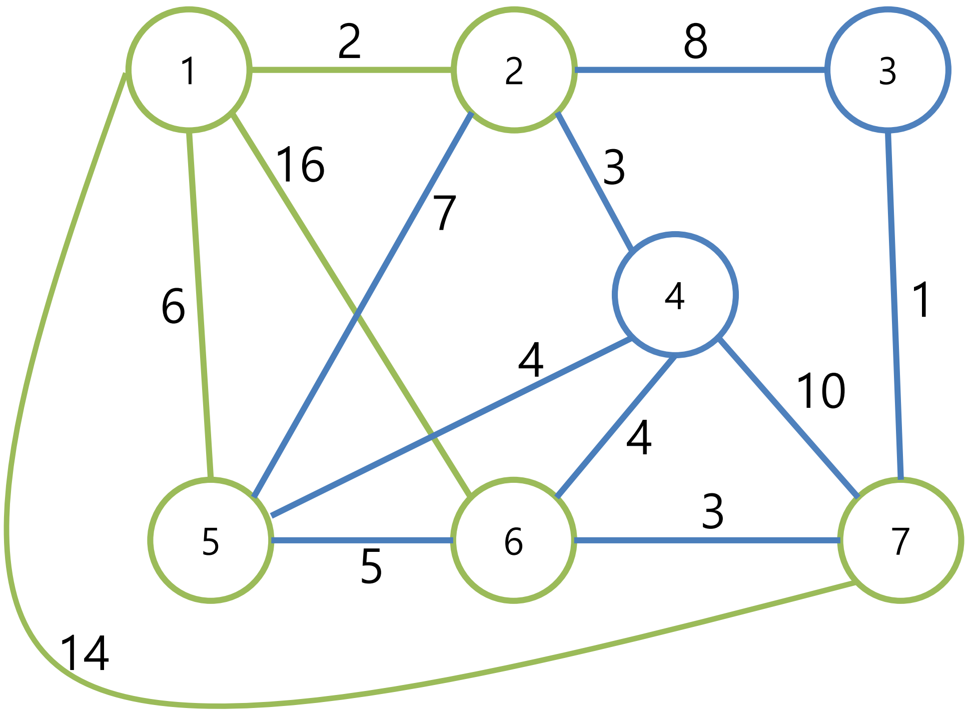
Dijkstra's algorithm(1/8)

- **Dijkstra's algorithm (source vertex: v)**

```
int i, u, w;
for(i = 0; i < n; i++){
    found[i] = FALSE;                // if found[i] == TRUE, i is in S
    distance[i] = cost[v][i];        // v : source vertex
}
found[v] = TRUE;                     // initially S = { v }
distance[v] = 0;
for(i = 0; i < n – 2; i++){
    u = choose(distance, n, found); // find a vertex with minimum distance
    found[u] = TRUE;                 // add u into S
    for(w = 0; w < n; w++){          // adjust distances to the vertices not in S
      if(!found[w] && distance[u]+cost[u][w] < distance[w])
         distance[w] = distance[u]+cost[u][w];
    }
}
```

Dijkstra's algorithm(2/8)

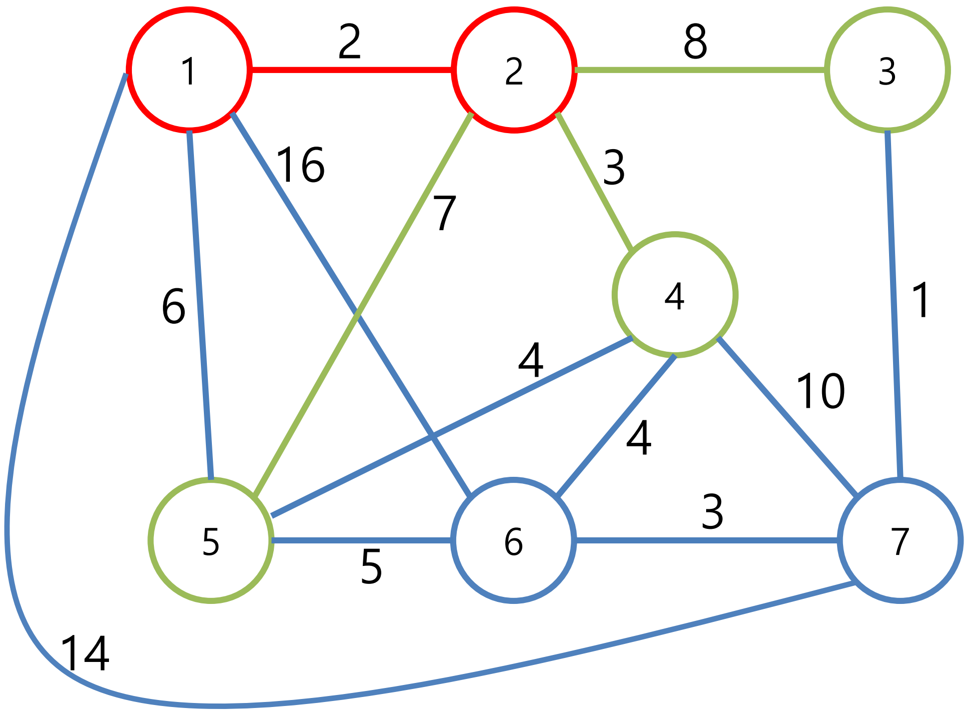
■ **Source node: 1**



dst.	path	cost
1	-	-
2	1→2	2
3	-	-
4	-	-
5	1→5	6
6	1→6	16
7	1→7	14

Dijkstra's algorithm(3/8)

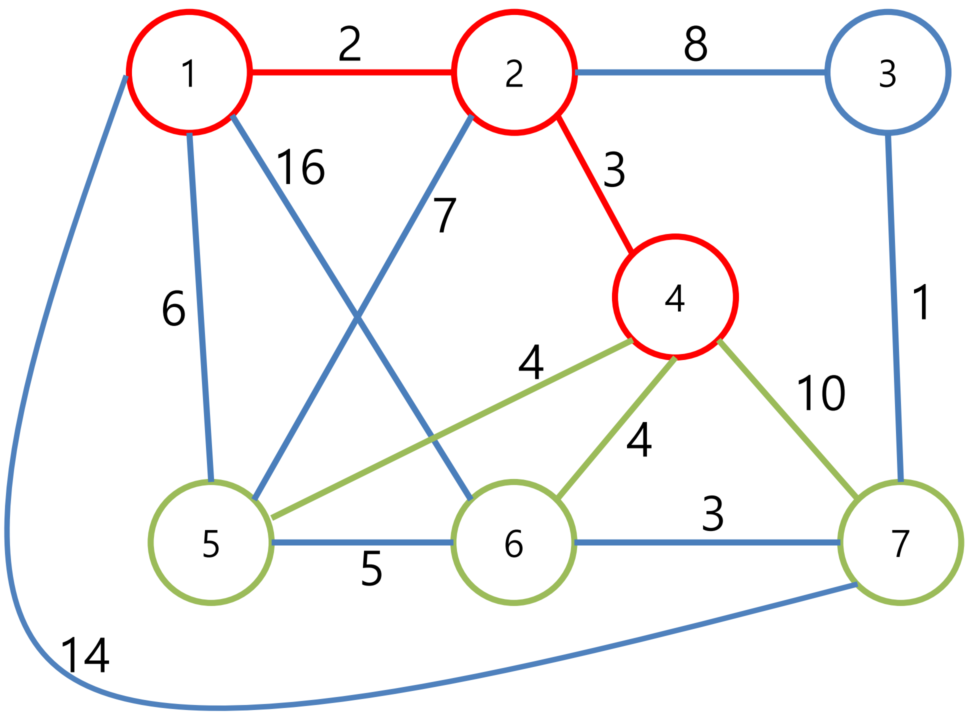
■ **Update based on 1→2 (minimum weight)**



dst.	path	cost
1	-	-
2	1→2	2
3	1→2→3	10
4	1→2→4	5
5	1→5	6
6	1→6	16
7	1→7	14

Dijkstra's algorithm(4/8)

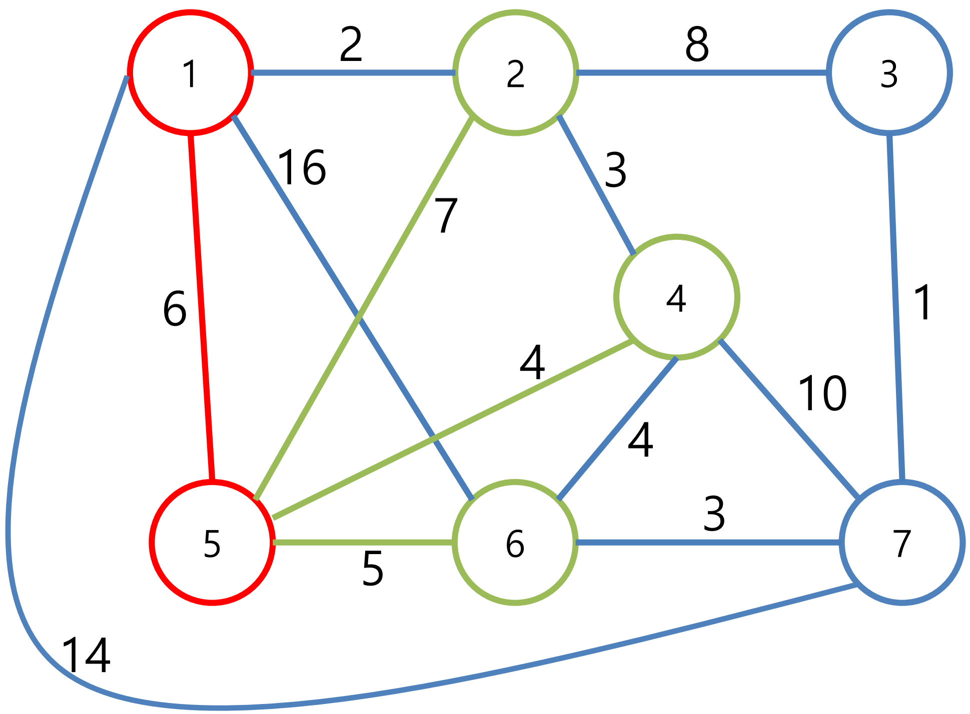
■ **Update based on 1→2→4 (the next minimum weight)**



dst.	path	cost
1	-	-
2	1→2	2
3	1→2→3	10
4	1→2→4	5
5	1→5	6
6	1→2→4→6	9
7	1→7	14

Dijkstra's algorithm(5/8)

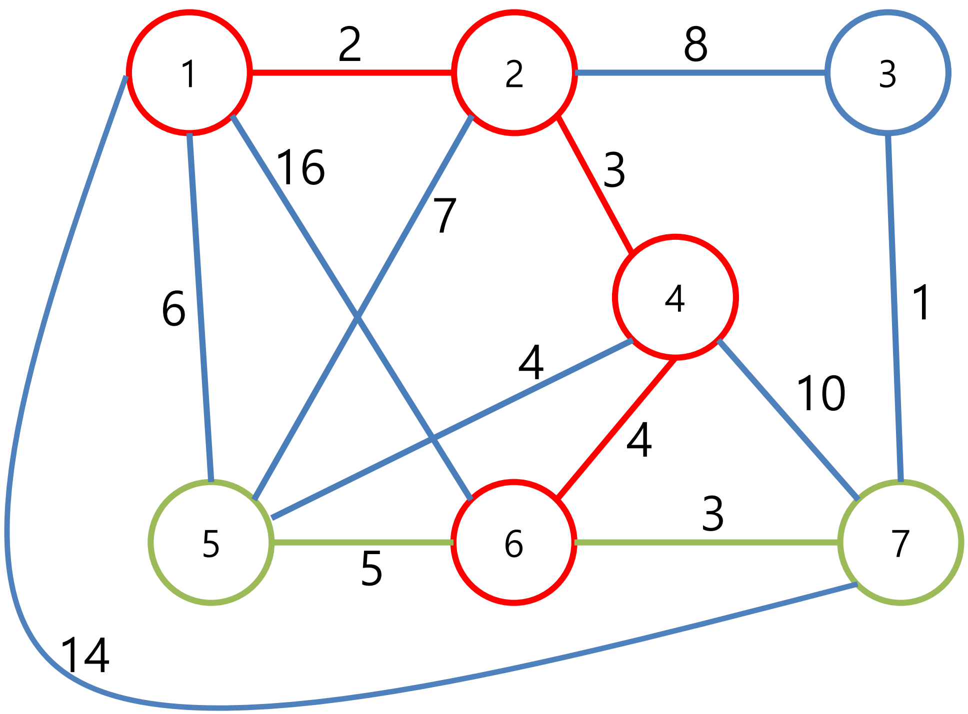
■ **Update based on 1→5 (the next minimum weight)**



dst.	path	cost
1	-	-
2	1→2	2
3	1→2→3	10
4	1→2→4	5
5	1→5	6
6	1→2→4→6	9
7	1→7	14

Dijkstra's algorithm(6/8)

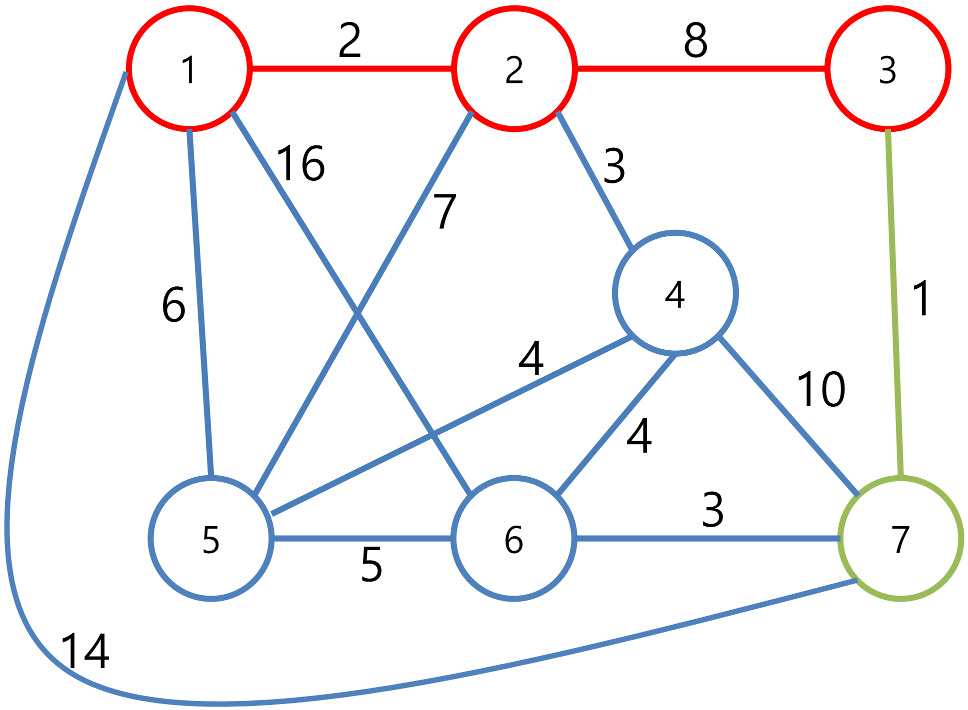
- Update based on 1→2→4→6 (the next minimum weight)



dst.	path	cost
1	-	-
2	1→2	2
3	1→2→3	10
4	1→2→4	5
5	1→5	6
6	1→2→4→6	9
7	1→2→4→6→7	12

Dijkstra's algorithm(7/8)

■ **Update based on 1→2→3 (the next minimum weight)**



dst.	path	cost
1	-	-
2	1→2	2
3	1→2→3	10
4	1→2→4	5
5	1→5	6
6	1→2→4→6	9
7	1→2→3→7	11

Dijkstra's algorithm(8/8)

- **No need to maintain the entire path for the each destination**
  - Only the last vertex before this destination is sufficient
  - Why?

dst.	path	cost
1	-	-
2	1→2	2
3	1→2→3	10
4	1→2→4	5
5	1→5	6
6	1→2→4→6	9
7	1→2→3→7	11

dst.	last	cost
1	-	-
2	1	2
3	2	10
4	2	5
5	1	6
6	4	9
7	3	11

Dynamic programming

- **Dynamic programming**
  - A problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems.
    - Each sub-problem has an optimal answer.
  - Difference from divide-and-conquer
    - Overlapping sub-problems

- **Divide-and-conquer**
  - solves a problem by splitting it recursively into smaller problems until all of the remaining problems are trivial

- **Dijkstra's algorithm is a dynamic programming**

Graph Applications

- Navigation, segmentation, etc.