Chapter 6
Graphs

2024 Spring

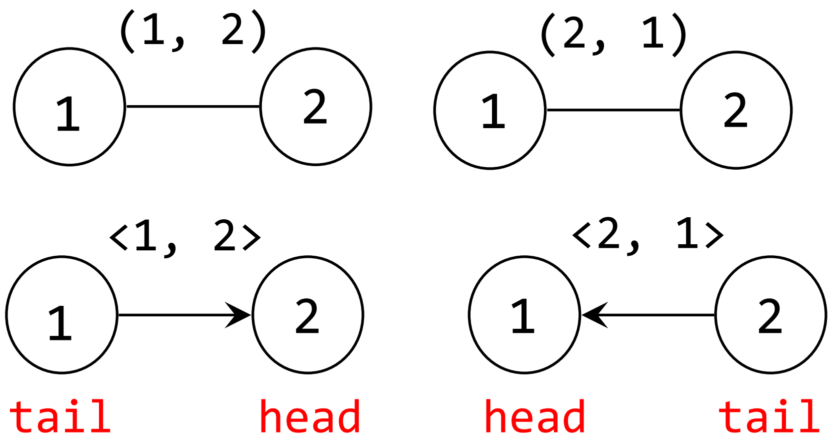
Ri Yu
Ajou University

Contents

Graph Abstract Data Type
Elementary Operations
Spanning Trees
Shortest Paths

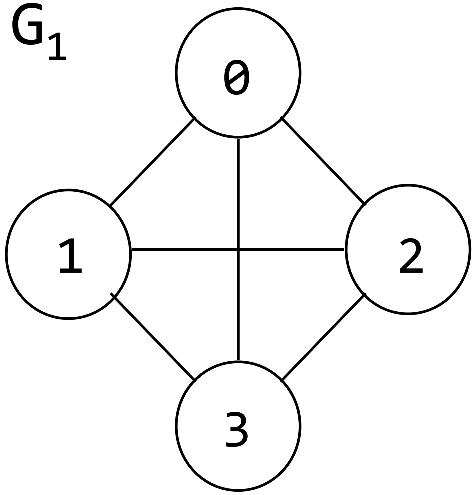
Definitions

- $G = (V, E)$, where
  - $V(G)$: set of vertices - finite and nonempty (정점)
  - $E(G)$: set of edges – finite and possibly empty (간선)
  - Restrictions
    - A graph may not have an edge from a vertex, I, back to itself
    - A graph may not have multiple occurrence of the same edge
- Undirected graph
  - unordered: $(u, v) = (v, u)$
- Directed graph (digraph)
  - ordered: $<u, v> \neq <v, u>$



Examples of Graph (1)

G_1



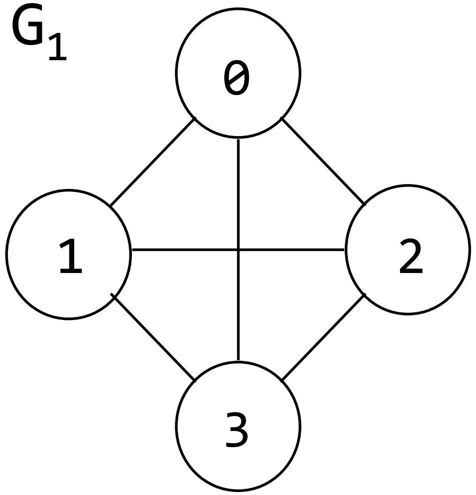
$V(G_1) = \{0, 1, 2, 3\}$
$E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$

G_2

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

Examples of Graph (1)

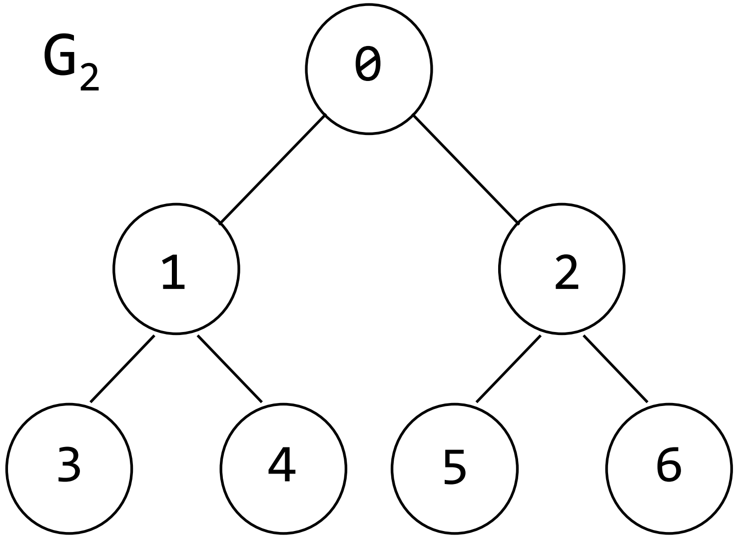
G_1



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

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

G_2

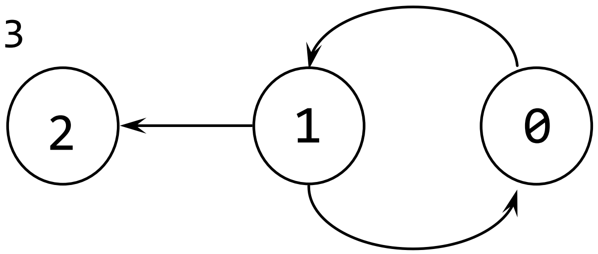


$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$

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

Examples of Graph (2)

G_3



$V(G_3) = \{0, 1, 2\}$
$E(G_3) = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,2\rangle\}$

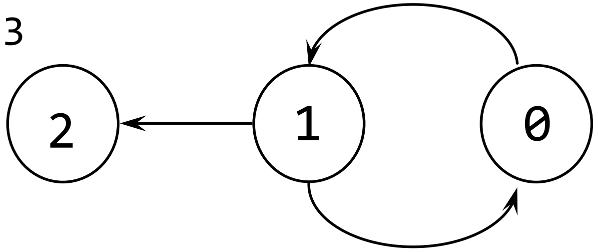
G_4

G_5 $V(G_5) = \{0, 1, 2, 3\}$
$E(G_5) = \{(0,1), (1,2), (1,3), (3,1), (3,2), (2,3), (3,2)\}$

$V(G_4) = \{0, 1, 2\}$
$E(G_4) = \{ \langle 0,0\rangle, \langle 0,2\rangle, \langle 1,0\rangle, \langle 2,1\rangle, \langle 2,0\rangle \}$

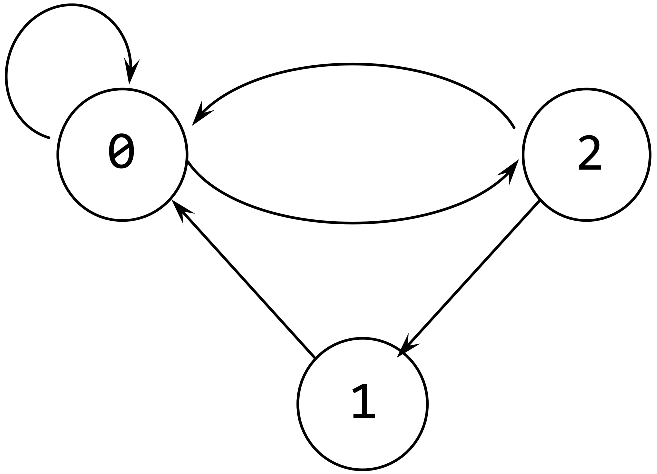
Examples of Graph (2)

G_3



$V(G_3) = \{0, 1, 2\}$
$E(G_3) = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,2\rangle\}$

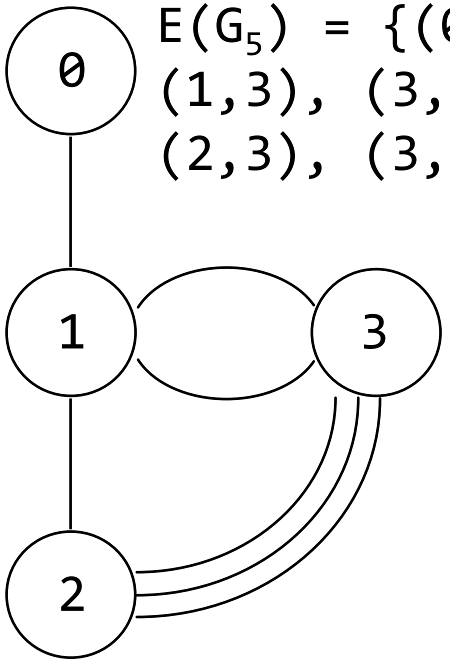
G_4



$V(G_4) = \{0, 1, 2\}$
$E(G_4) = \{ \langle 0,0\rangle, \langle 0,2\rangle, \langle 1,0\rangle, \langle 2,1\rangle, \langle 2,0\rangle \}$

G_5

$V(G_5) = \{0, 1, 2, 3\}$
$E(G_5) = \{(0,1), (1,2), (1,3), (3,1), (3,2), (2,3), (3,2)\}$



Terminology (1)

- Complete graph (완전 그래프)

  A graph that has the <u>maximum number of edges</u>

  →For an <u>undirected graph with *n* vertices</u>,

  the maximum number of the edges = $n(n\text{-}1)/2$

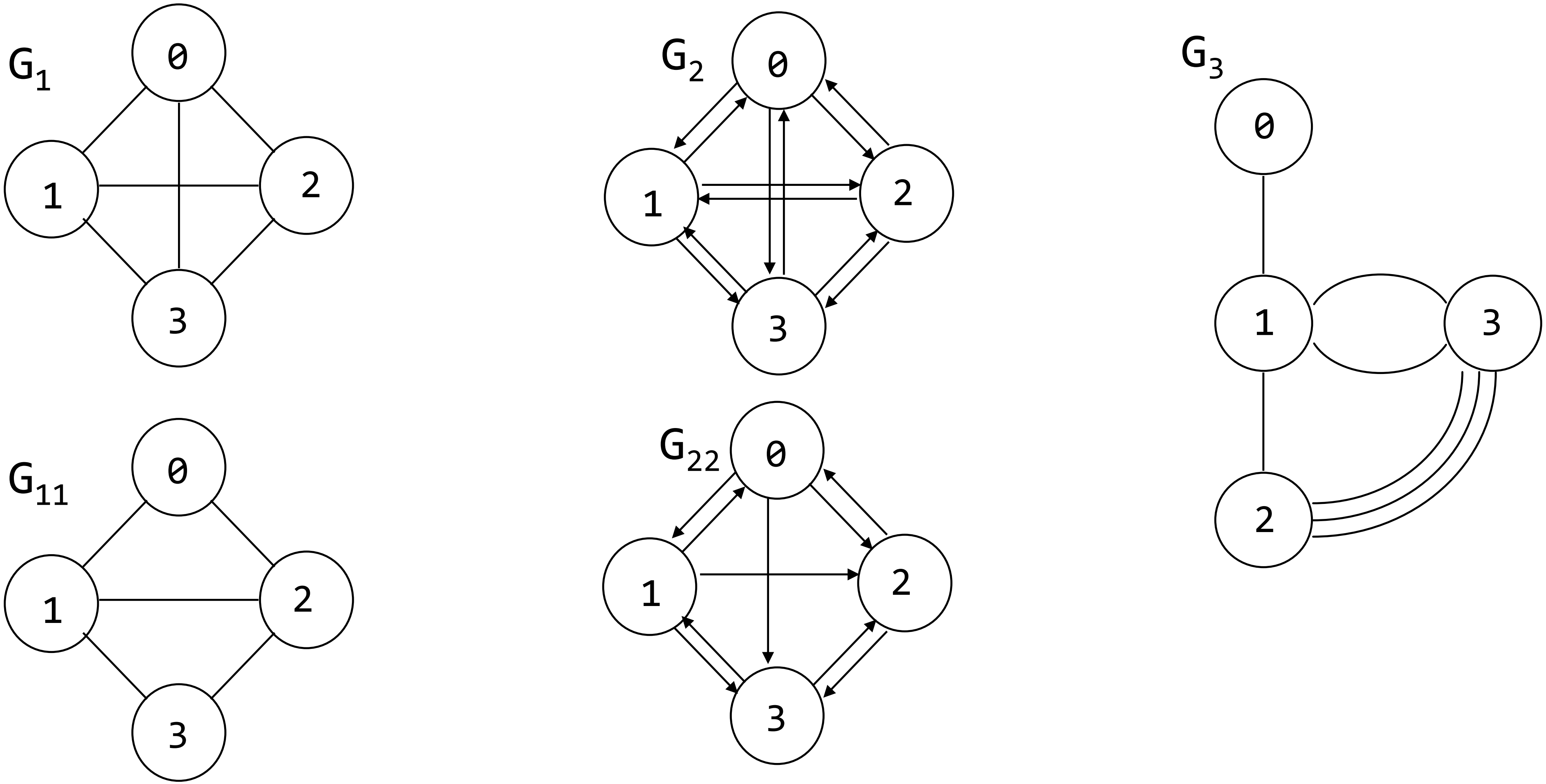
  →For a <u>directed graph with *n* vertices</u>,

  the maximum number of the edges = $n(n\text{-}1)$

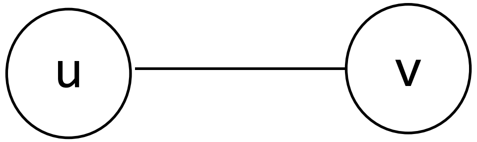
- Multigraph (다중그래프)

  A graph whose edges are unordered pairs of vertexes, and <u>the same pair of vertexes can be connected by multiple edges</u>

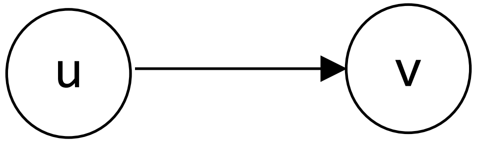
Terminology - Example



Terminology (2)



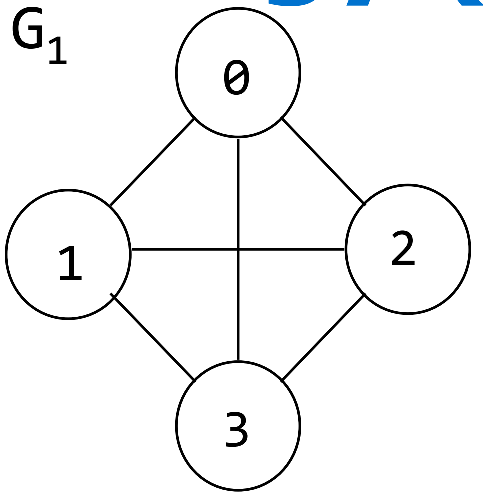
❖ If (u, v) is an edge of an undirected graph,
- The vertices u and v are adjacent (인접한)
- The edge (u, v) is incident (부속된) on u and v

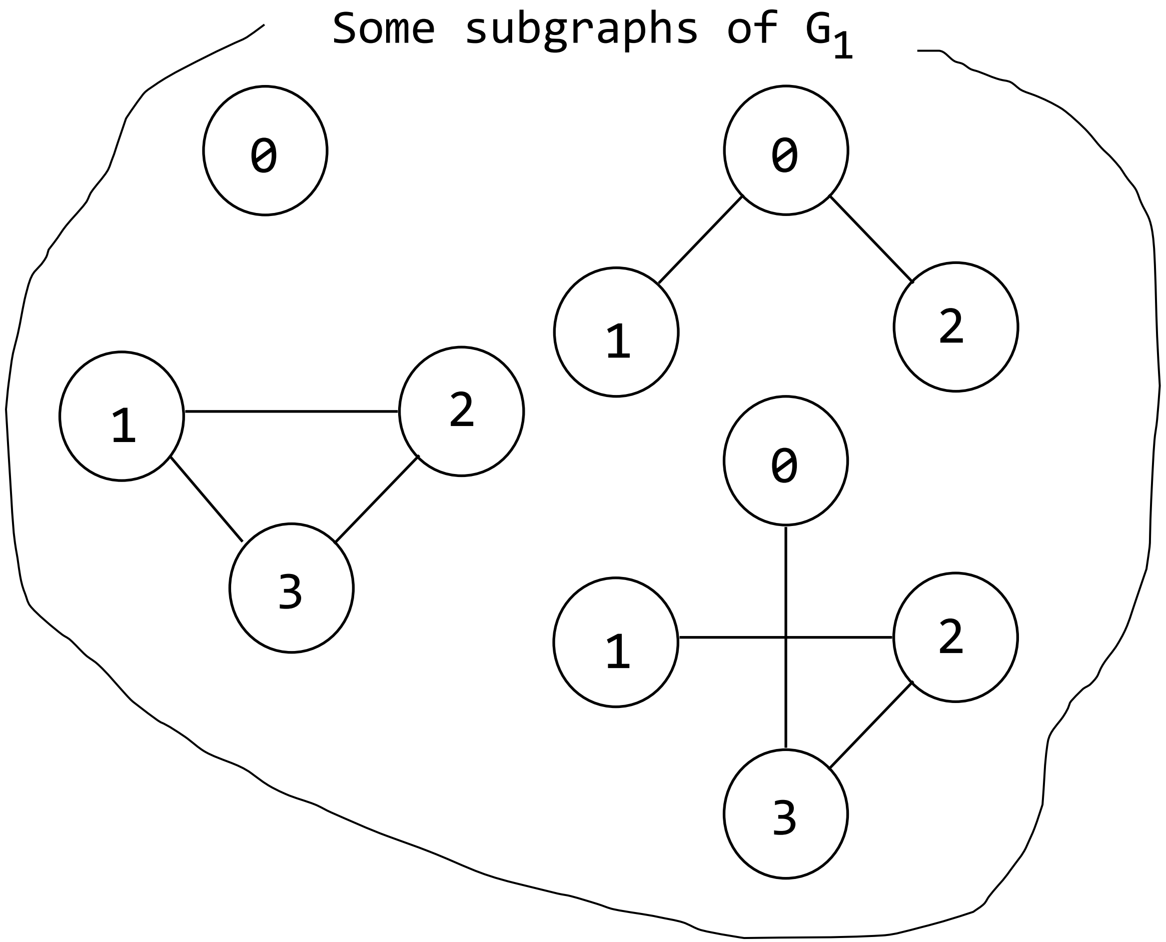


❖ If $<u, v>$ is a directed edge,
- The vertex u is adjacent to v
- The vertex v is adjacent from u
- The edge $<u, v>$ is incident on u and v

A subgraph of G is a graph G' such that
$V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Terminology (3)





Terminology (4)

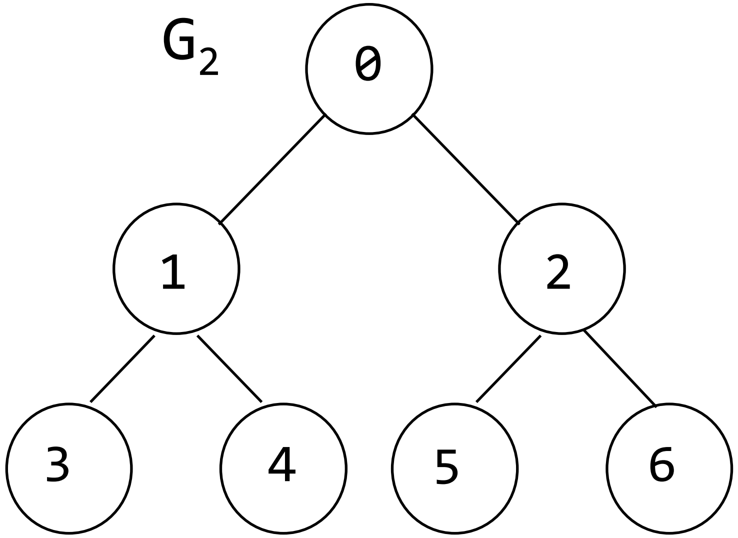
- A **path** from vertex u to v in graph G is a sequence of vertices, $\boldsymbol{u}$, i_1, i_2, …, i_k, $\boldsymbol{v}$ such that (u, i_1), (i_1, i_2), …, (i_k, v) are edges in an undirected graph
  - If G′ is a directed graph, the path consists of $<u, i_1>$, $< i_1, i_2 >$, …, $< i_k, v>$
  - The length of a path is the number of the edges on it

- A **simple path** is a path in which all vertices, except possibly the first and the last, are distinct

- A **cycle** is a simple path in which the first and the last vertices are the same

Terminology (5)

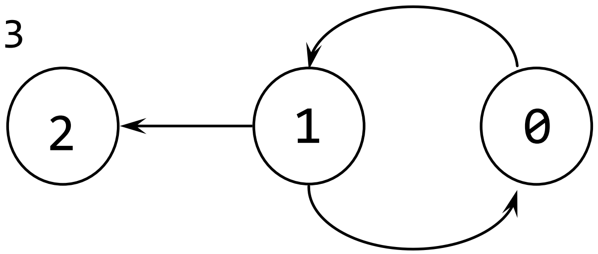
G_2



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

Simple path: 0, 2, 6
0, 1, 4
2, 5
…
0, 2, 6, 2, 6 ??

G_3



$V(G_3) = \{0, 1, 2\}$
$E(G_3) = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,2\rangle\}$

Simple path: 0, 1, 2
1, 0
1, 2

Cycle: 0, 1, 0
0, 1, 0, 1, 2 ??

Terminology (6)

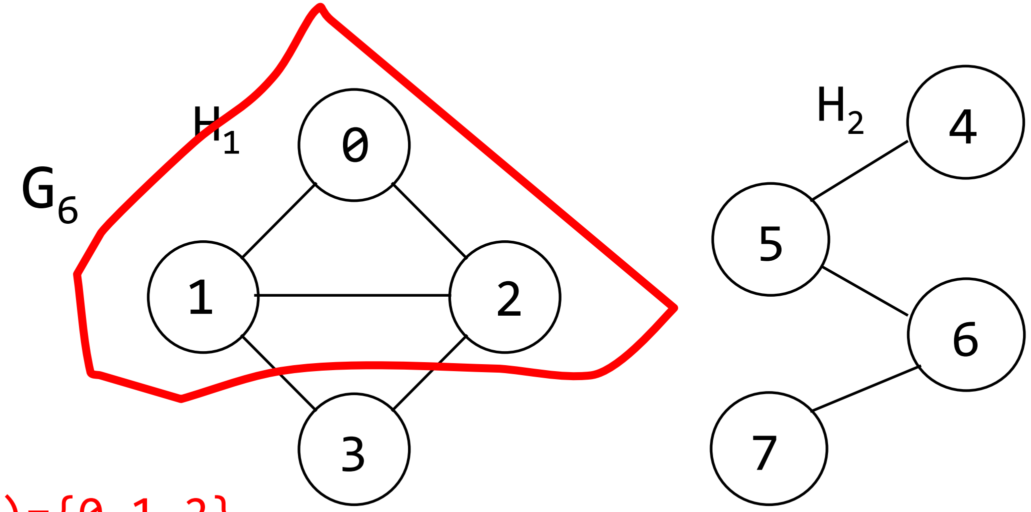
- In an undirected graph G, two vertices u and v are connected if there is a path in G from u and v

- A connected component or simply a component of an undirected graph is a maximal connected subgraph

- A tree is a graph that is connected and acyclic

Terminology (7)

$V(G_6) = \{0, 1, 2, 3, 4, 5, 6, 7\}$
$E(G_6) = \{(0,1), (0,2), (1,2), (1,3), (2,3), (4,5), (5,6), (6,7)\}$



Is 0 adjacent to 1?

Is 0 adjacent to 3?

Is 0 connected to 3?

Is 3 connected to 4?

$V(H_3)=\{0,1,2\}$
$E(H_3)=\{(0,1), (0,2), (1,2)\}$

Connected component?? No

Connected components

$V(H_1) = \{0, 1, 2, 3\}$
$E(H_1) = \{(0,1), (0,2), (1,2), (1,3), (2,3)\}$

$V(H_2) = \{4,5,6,7\}$
$E(H_2) = \{(4,5), (5,6), (6,7)\}$

Terminology (8)

- A directed graph is **strongly connected** if, for every pair of vertices, u and v in $V(G)$, there is a directed path from u and v and also from v to u

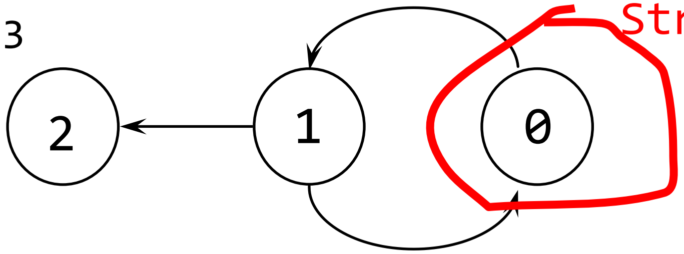
- A strongly connected component is a **maximal** subgraph that is strongly connected

Terminology (9)

$V(H_5)=\{0\}$, $E(H_5)=\{\}$

Strongly connected component?? No

G_3

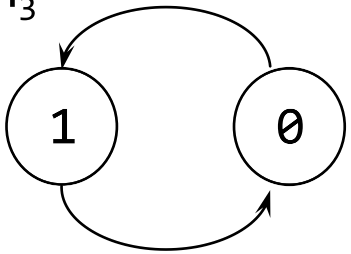


$V(G_3) = \{0, 1, 2\}$

$E(G_3) = \{\langle 0,1\rangle, \langle 1,0\rangle, \langle 1,2\rangle\}$

Strongly connected components

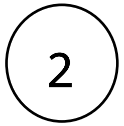
H_3



$V(H_3) = \{0, 1\}$

$E(H_3) = \{\langle 0,1\rangle, \langle 1,0\rangle\}$

H_4



$V(H_4) = \{2\}$

$E(H_4) = \{\}$

Terminology (10)

- The degree of a vertex is the number of edges incident to that vertex
- For a directed graph,

  the in-degree of a vertex v is defined as the number of edges that have v as the head

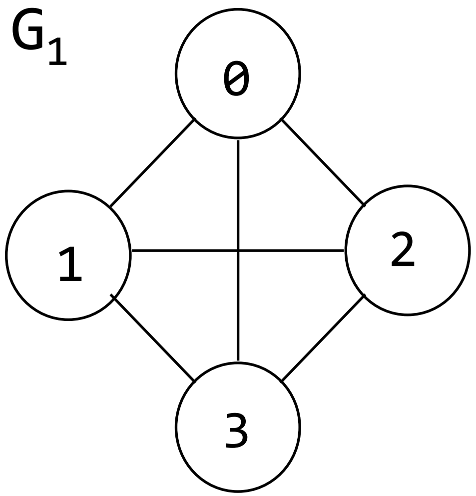
  the out-degree of a vertex v is defined as the number of edges that have v as the tail
- Property

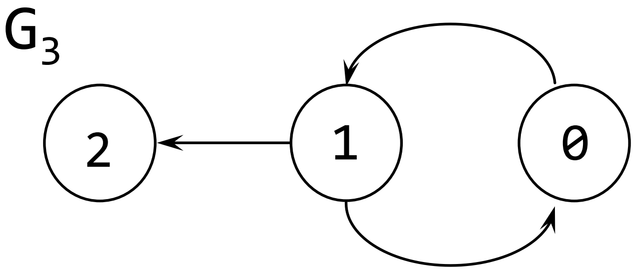
  e : the number of edges

  d_i : the degree of a vertex i in graph G

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

Terminology (11)





Degree of 0	
Degree of 1	
Degree of 3	

in-degree of 1	
out-degree of 1	
out-degree of 2	

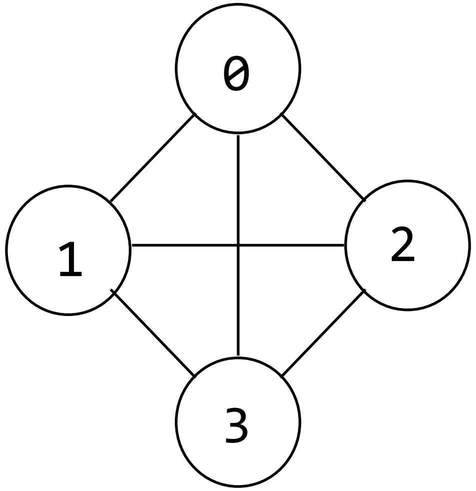
Graph representations

1. Adjacency Matrix
2. Adjacency Lists

Adjacency Matrix

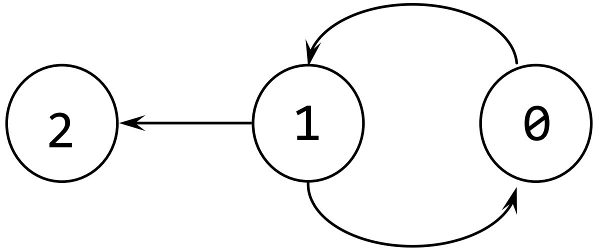
Two-dimensional $n \times n$ array, when the number of nodes is n

$$\texttt{a[i][j]} = \begin{cases} \texttt{1, if (i,j) is adjacent} \\ \texttt{0, otherwise} \end{cases}$$



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

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

Property of Adjacency Matrix

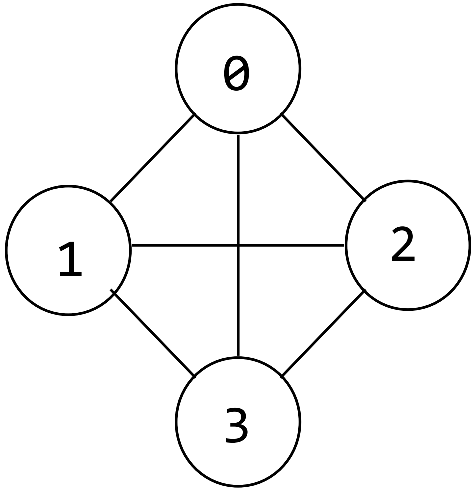
- The adjacency matrix for an undirected graph is symmetric
- The adjacency matrix for a digraph need not be symmetric
- For an undirected graph, the degree of any vertex *i* is its row sum
- For a directed graph,

  the row sum is the out-degree;

  the column sum is the in-degree

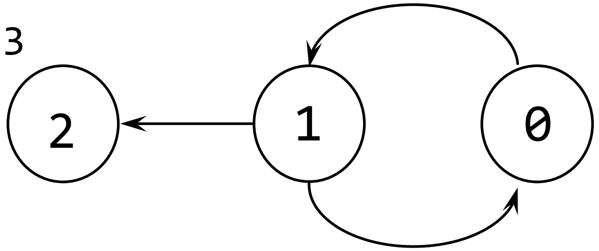
`<i,j> → row i, column j is set to 1`

G_1



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

G_3



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

In-degree of vertex 2 ?

out-degree of vertex 2 ?

0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역

(사당, 아주대)	(0, 1)
(사당, 강남)	(0, 2)
(강남, 의왕)	(2, 3)
(평촌, 의왕)	(4, 3)
(수지, 강남)	(5, 2)
(아주대, 강남)	(1, 2)
(명동, 서울역)	(6, 7)

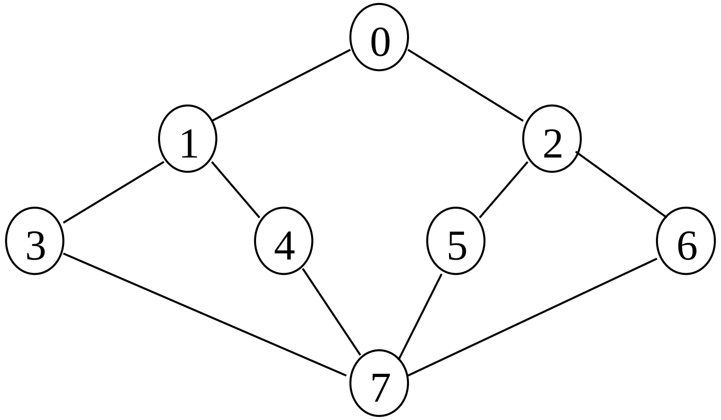
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]								
[1]								
[2]								
[3]								
[4]								
[5]								
[6]								
[7]								

0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역

(사당, 아주대)	(0, 1)
(사당, 강남)	(0, 2)
(강남, 의왕)	(2, 3)
(평촌, 의왕)	(4, 3)
(수지, 강남)	(5, 2)
(아주대, 강남)	(1, 2)
(명동, 서울역)	(6, 7)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
[0]	0	1	1	0	0	0	0	0
[1]	1	0	1	0	0	0	0	0
[2]	1	1	0	1	0	1	0	0
[3]	0	0	1	0	1	0	0	0
[4]	0	0	0	1	0	0	0	0
[5]	0	0	1	0	0	0	0	0
[6]	0	0	0	0	0	0	0	1
[7]	0	0	0	0	0	0	1	0

Waste of memory



Utilization = 20/64 = 31(%)

0	1	1	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	0	0	0	1	1	0
0	1	0	0	0	0	0	1
0	1	0	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1
0	0	0	1	1	1	1	0

Adjacency Lists

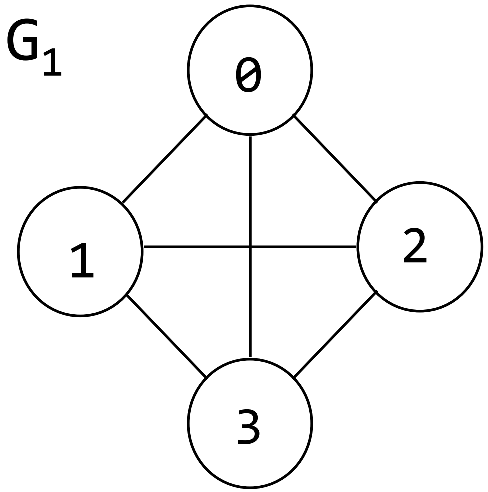
Replace n rows of adjacency matrix with n linked lists

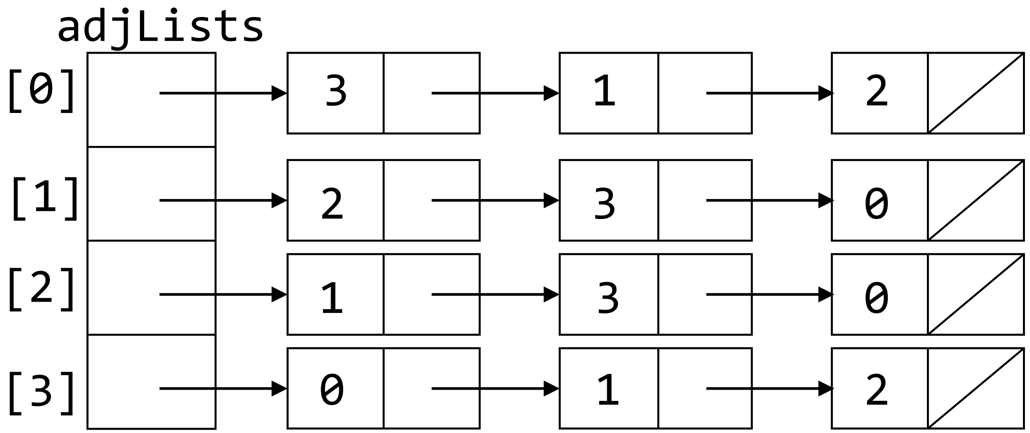
Every vertex i in G has one list

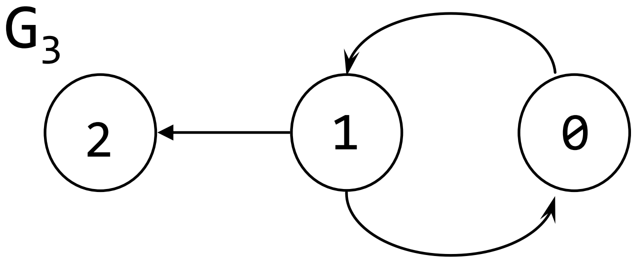
The nodes in chain i represent the vertices that are adjacent from i

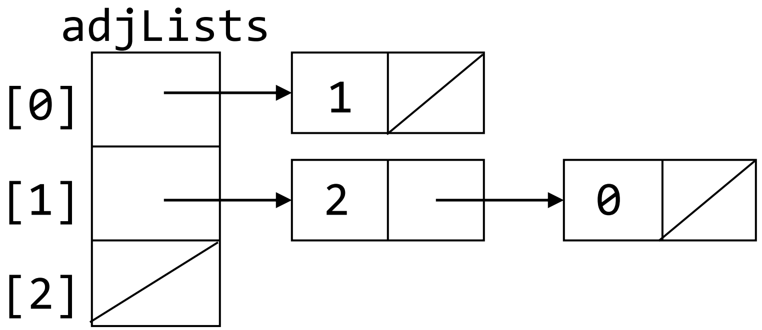
The vertices in each chain are not required to be ordered

Adjacency Lists - Examples









Adjacency Lists in C

```
typedef struct node *nodePointer;
typedef struct node {
    int vertex;
    nodePointer link;
};
nodePointer adjLists[MAX_NODES];
```

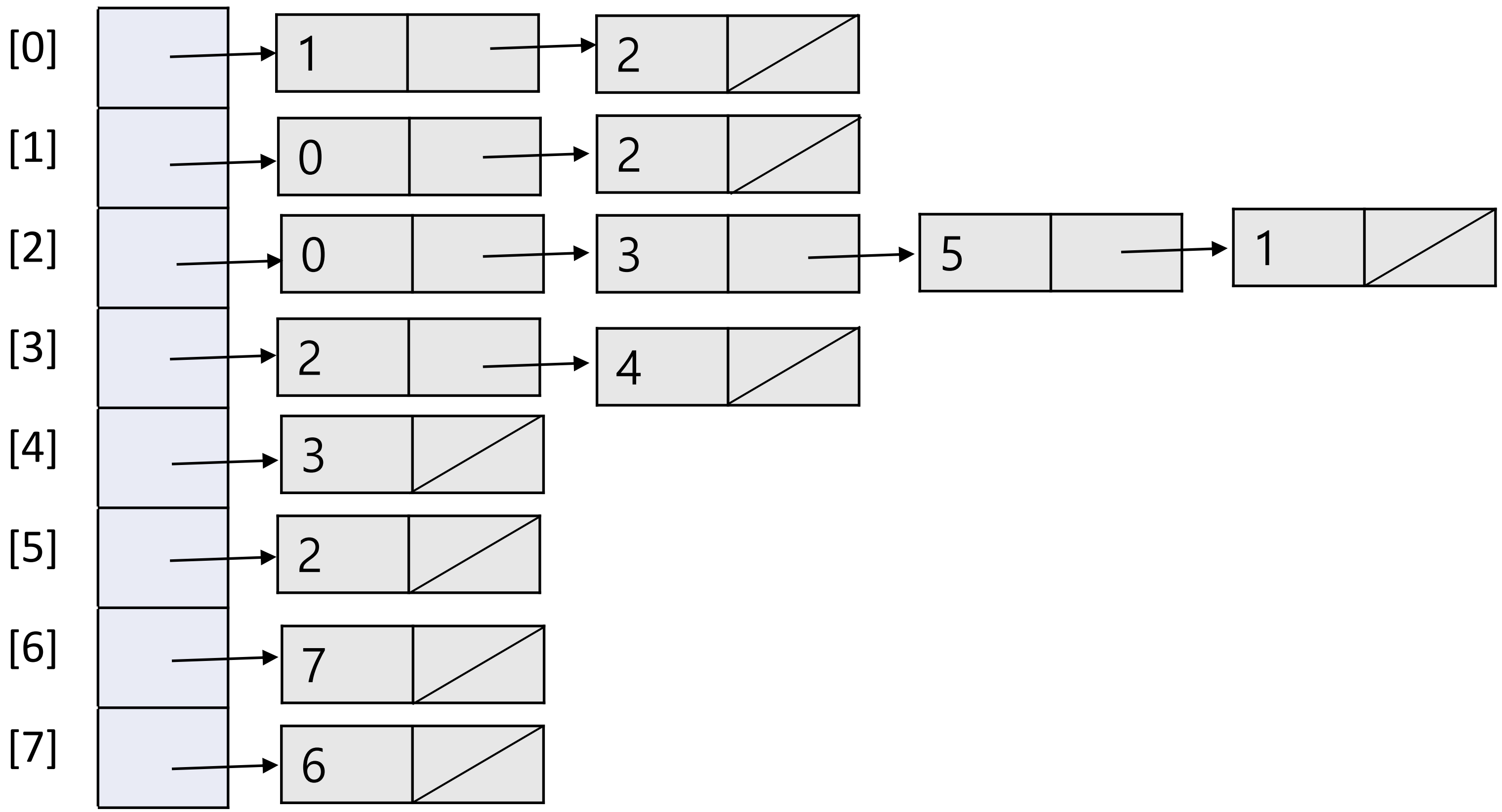
0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역

[0]	
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	
[7]	

(사당, 아주대)	(0, 1)
(사당, 강남)	(0, 2)
(강남, 의왕)	(2, 3)
(평촌, 의왕)	(4, 3)
(수지, 강남)	(5, 2)
(아주대, 강남)	(1, 2)
(명동, 서울역)	(6, 7)

0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역

(사당, 아주대)	(0, 1)
(사당, 강남)	(0, 2)
(강남, 의왕)	(2, 3)
(평촌, 의왕)	(4, 3)
(수지, 강남)	(5, 2)
(아주대, 강남)	(1, 2)
(명동, 서울역)	(6, 7)



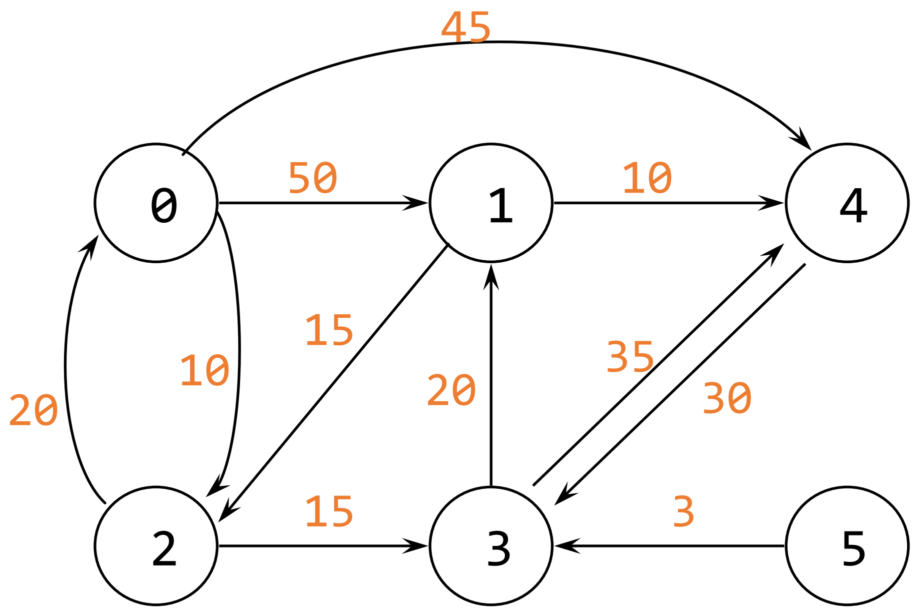
Weighted Edges (가중치 간선)

❖ Assign weights to edges of a graph

- Distance from one vertex to another, or
- Cost of going from one vertex to an adjacent vertex

❖ Modify representation to signify an edge with the weight of the edge

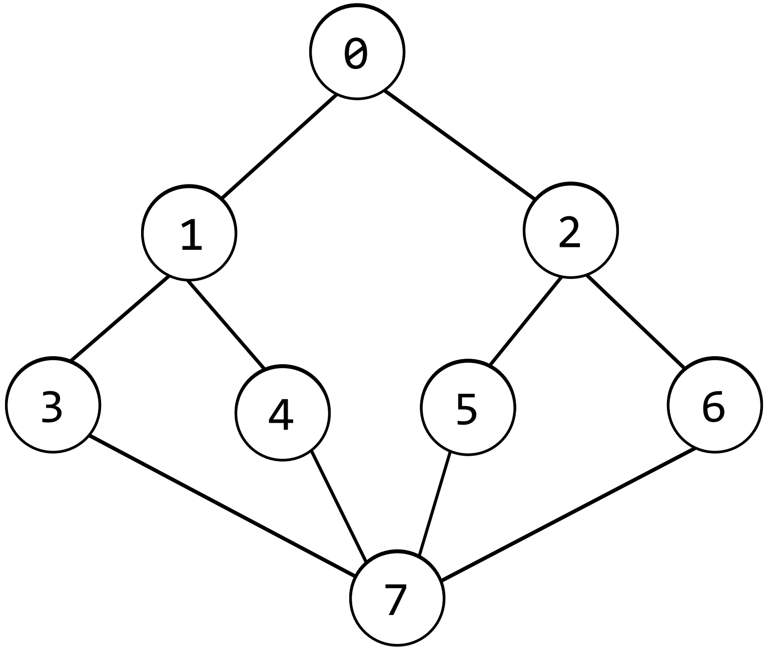
- for adjacency matrix : weight instead of 1
- for adjacency list : add weight field

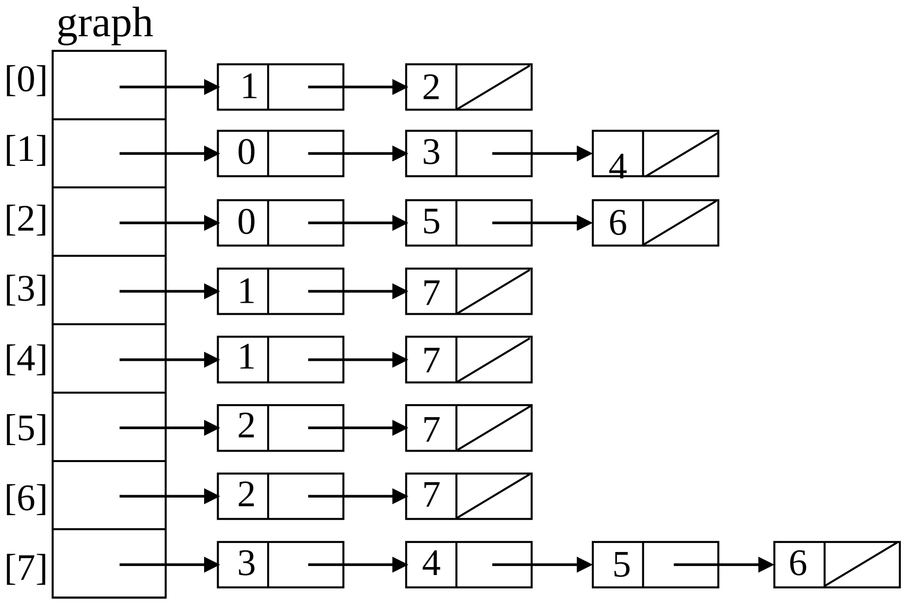


$$
\begin{array}{c|cccccc}
 & [0] & [1] & [2] & [3] & [4] & [5] \\
\hline
[0] & 0 & 50 & 10 & \infty & 45 & \infty \\
[1] & \infty & 0 & 15 & \infty & 10 & \infty \\
[2] & 20 & \infty & 0 & 15 & \infty & \infty \\
[3] & \infty & 20 & \infty & 0 & 35 & \infty \\
[4] & \infty & \infty & \infty & 30 & 0 & \infty \\
[5] & \infty & \infty & \infty & 3 & \infty & 0
\end{array}
$$

Graph Traversal

- ❖ Visit every vertex in a graph
  - DFS (Depth First Search) - similar to a preorder tree traversal
  - BFS (Breath First Search) - similar to a level-order tree traversal

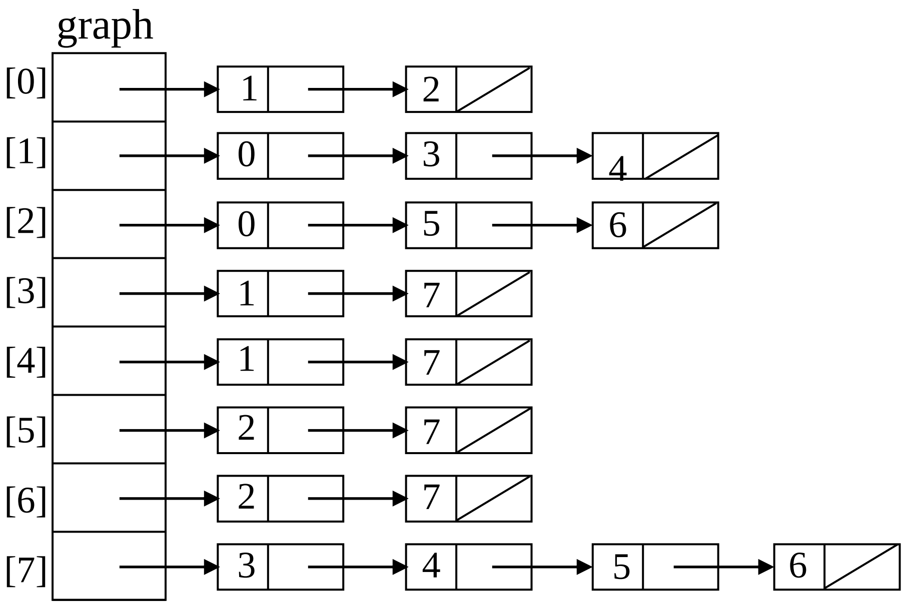




Depth First Search

```
typedef struct node *nodePointer;
typedef struct node {
    int vertex;
    nodePointer link;
};
nodePointer adjLists[MAX_NODES];
```

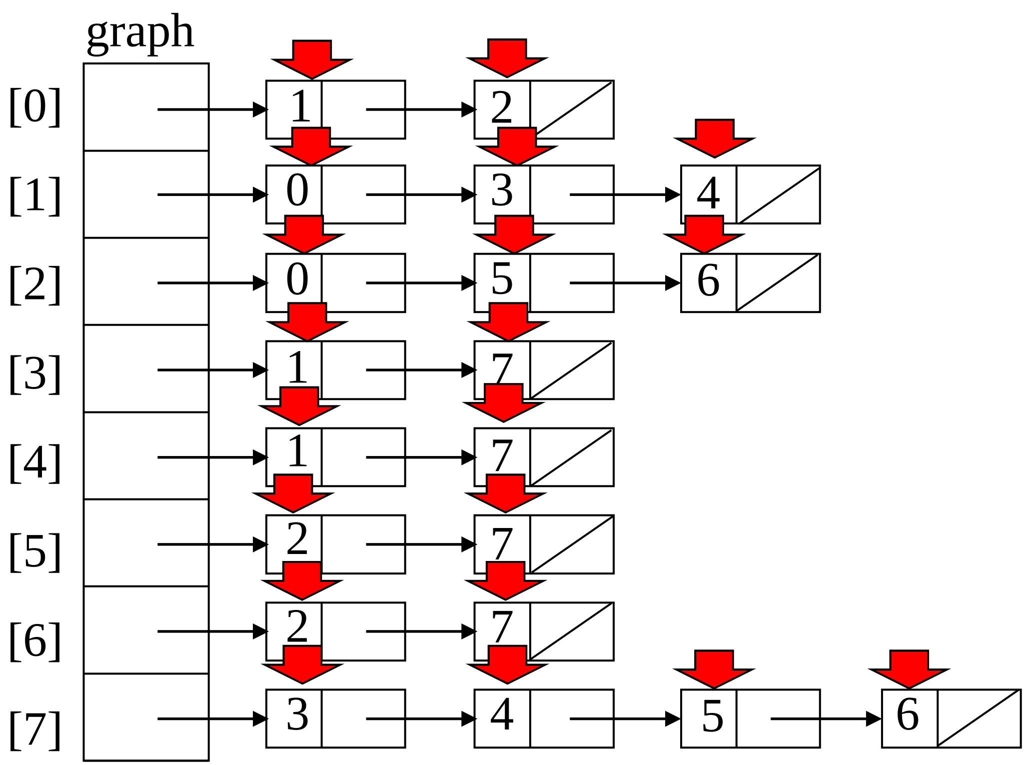
```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
nodePointer graph[MAX_VERTICES];

void dfs(int v)
{
/* depth first search 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);
   }
}
```



Example of DFS

```
void dfs(int v)
{
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d", v);
   for (w = graph[v]; w; w = w->link){
     if (!visited[w->vertex])
         dfs(w->vertex);
   }
}
```

`dfs(0);`



0 1 3 7 4 5 2 6

dfs(6)
dfs(2)
dfs(5)
dfs(7)
dfs(3)
dfs(1)
dfs(0)

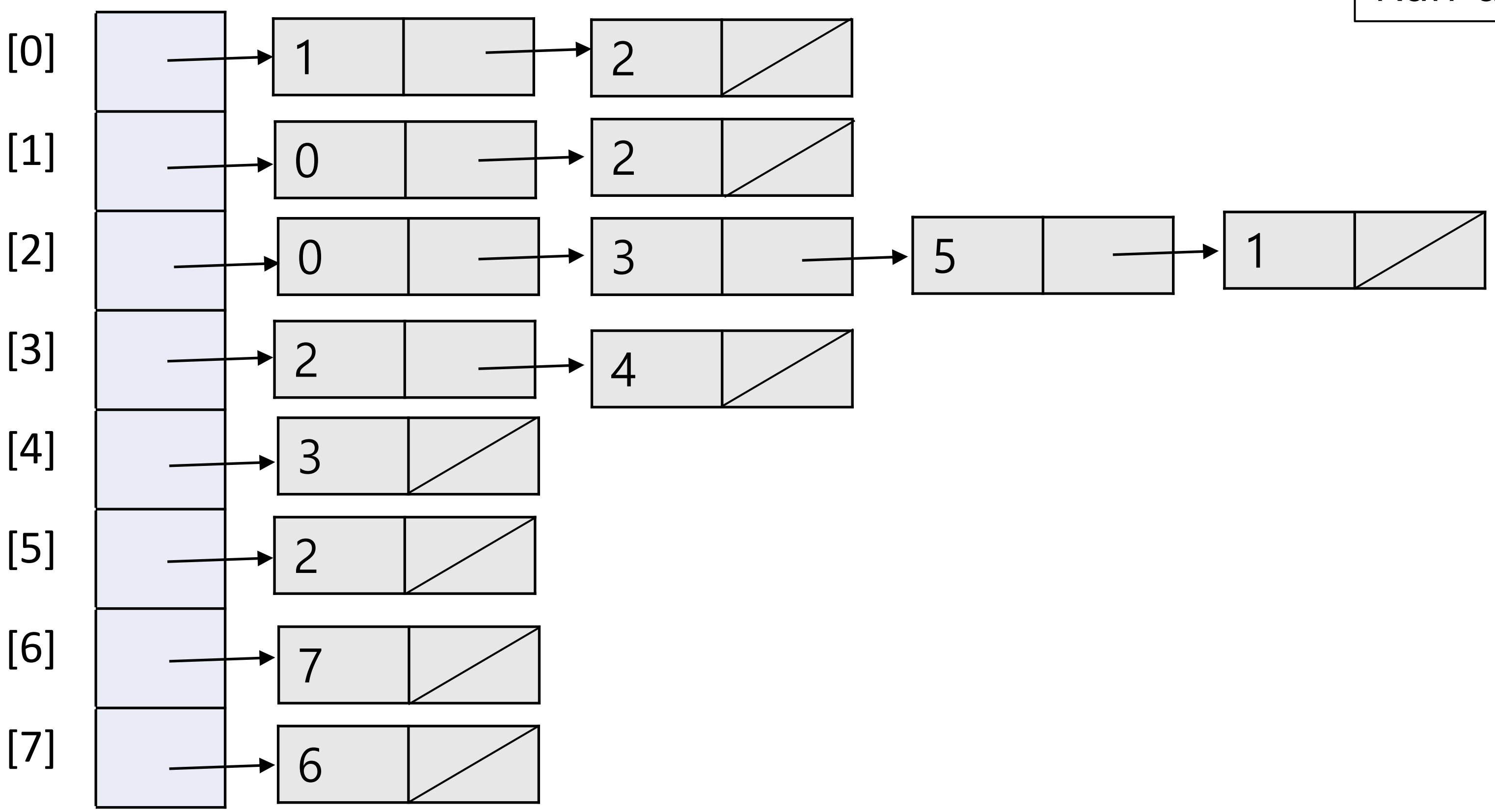
Function call stack

visited

[0]	1
[1]	1
[2]	1
[3]	1
[4]	1
[5]	1
[6]	1
[7]	1

0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역

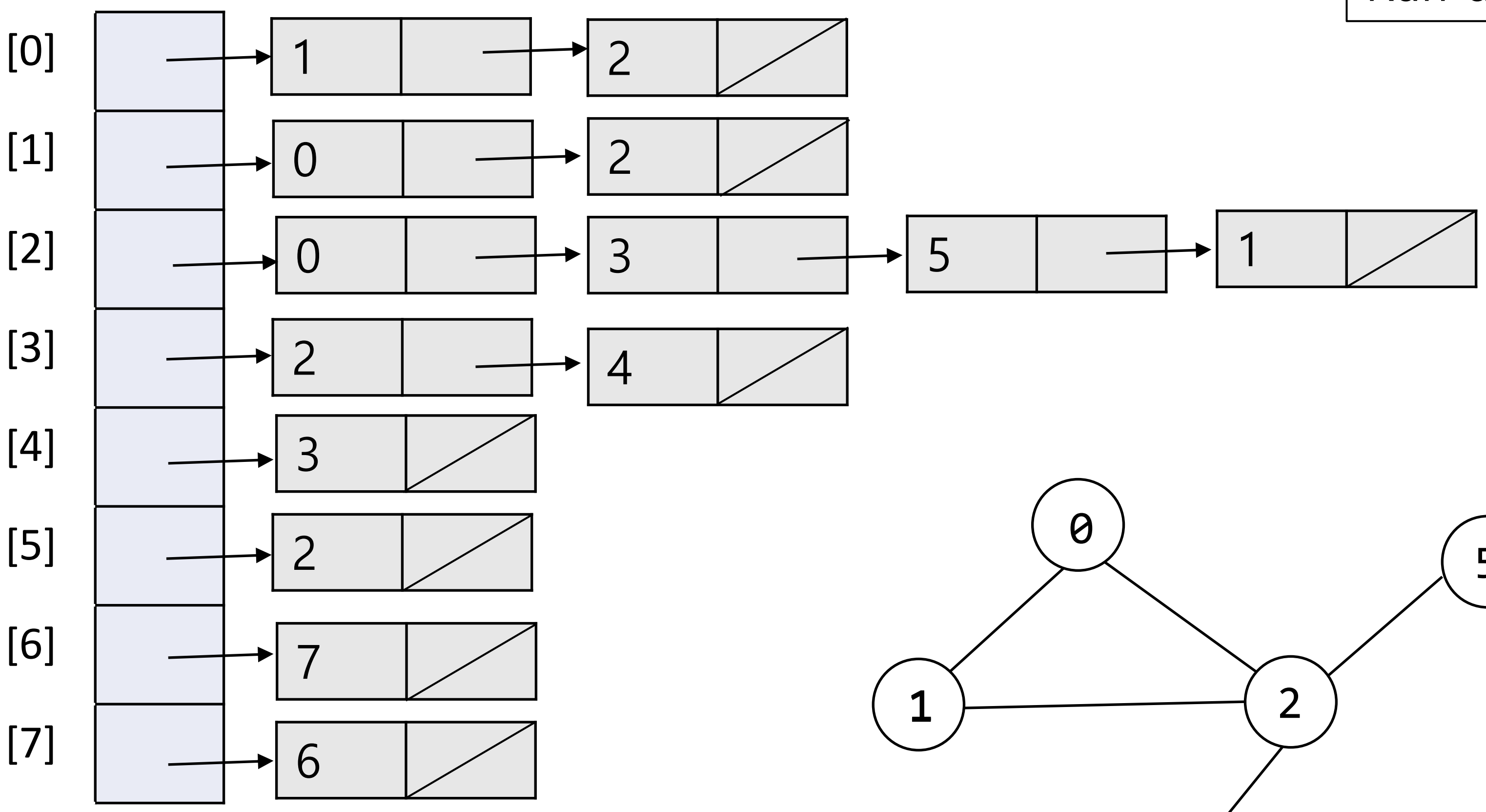
Run dfs(0) and show the result.

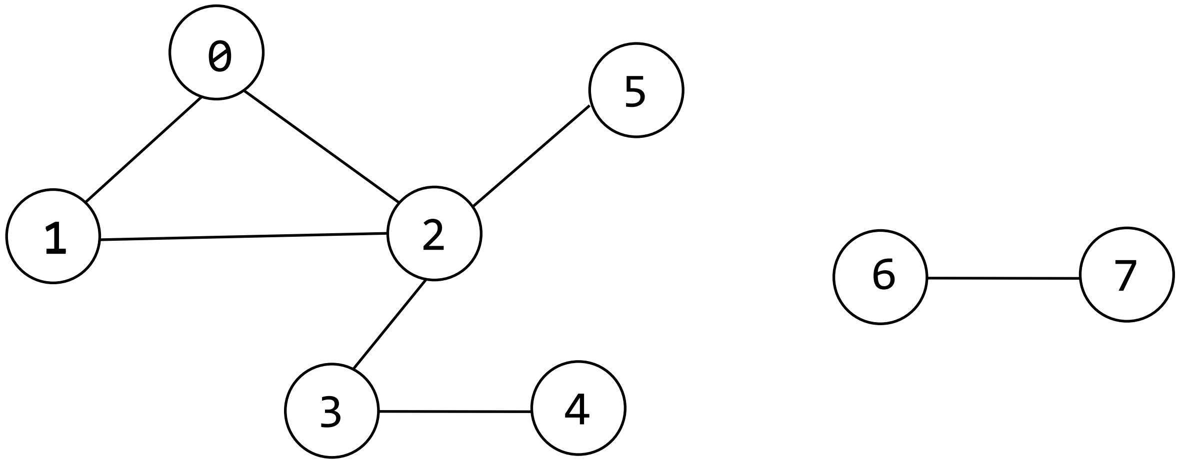



```
void dfs(int v)
{
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d", v);
   for (w = graph[v]; w; w = w->link){
     if (!visited[w->vertex])
        dfs(w->vertex);
   }
}
```

0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역

Run dfs(0) and show the result.





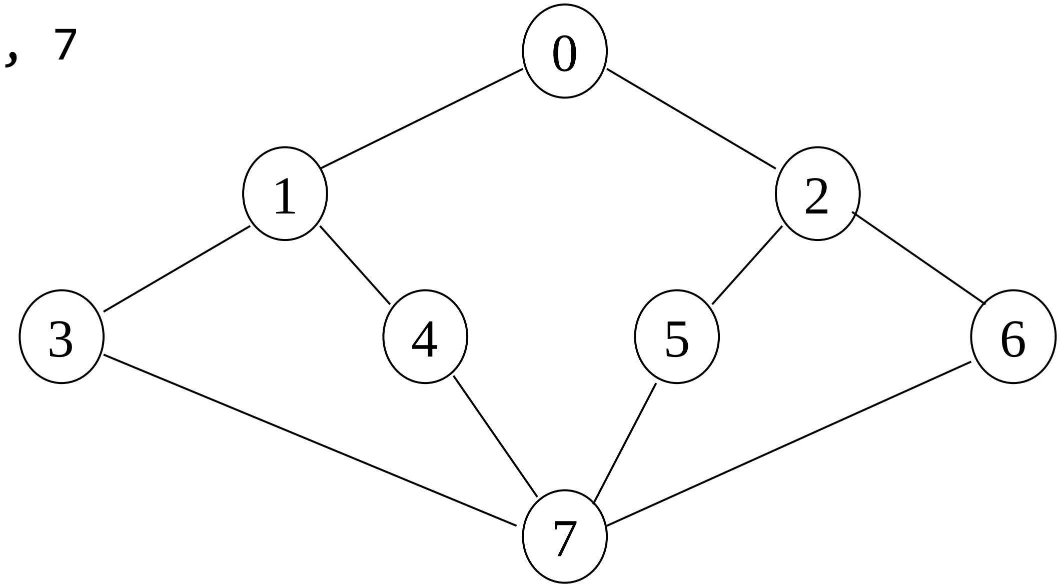

```
void dfs(int v)
{
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d", v);
   for (w = graph[v]; w; w = w->link){
     if (!visited[w->vertex])
        dfs(w->vertex);
   }
}
```

0 1 2 3 4 5

Breadth First Search

- BFS starts at vertex *v* and marks it as visited
- It then visits each of vertices on *v*'s adjacency list
- When all the vertices on *v*'s adjacency list are visited, all the unvisited vertices are visited that are adjacent to the first vertex on *v*'s adjacency list

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

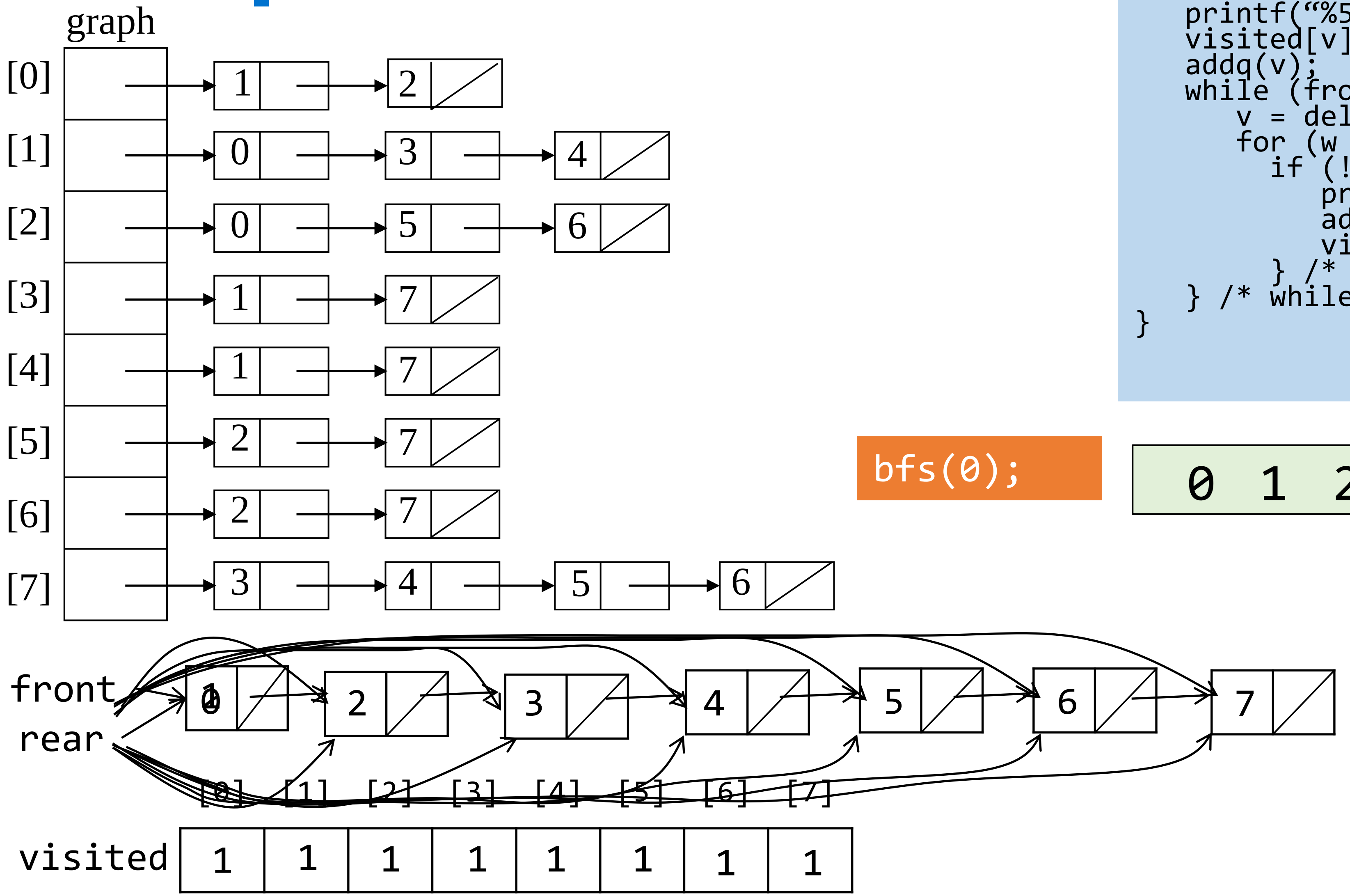


Breadth First Search

```
void bfs(int v)
{
   nodePointer w;
   front = rear = NULL;/* initialize queue */
   printf("%5d", v);
   visited[v] = TRUE;
   addq(v); /* p.159, Chapter 4 */
   while (front) {
      v = deleteq(); /* p.160, Chapter 4 */
      for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex]) {
           printf("%5d", w->vertex);
           addq(w->vertex);
           visited[w->vertex] = TRUE;
        } /* if */
   } /* while */
}
```

```
typedef struct node *queuePointer;
typedef struct node {
         int vertex;
         queuePointer link;
         };
queuePointer front, rear;
```

Example of BFS



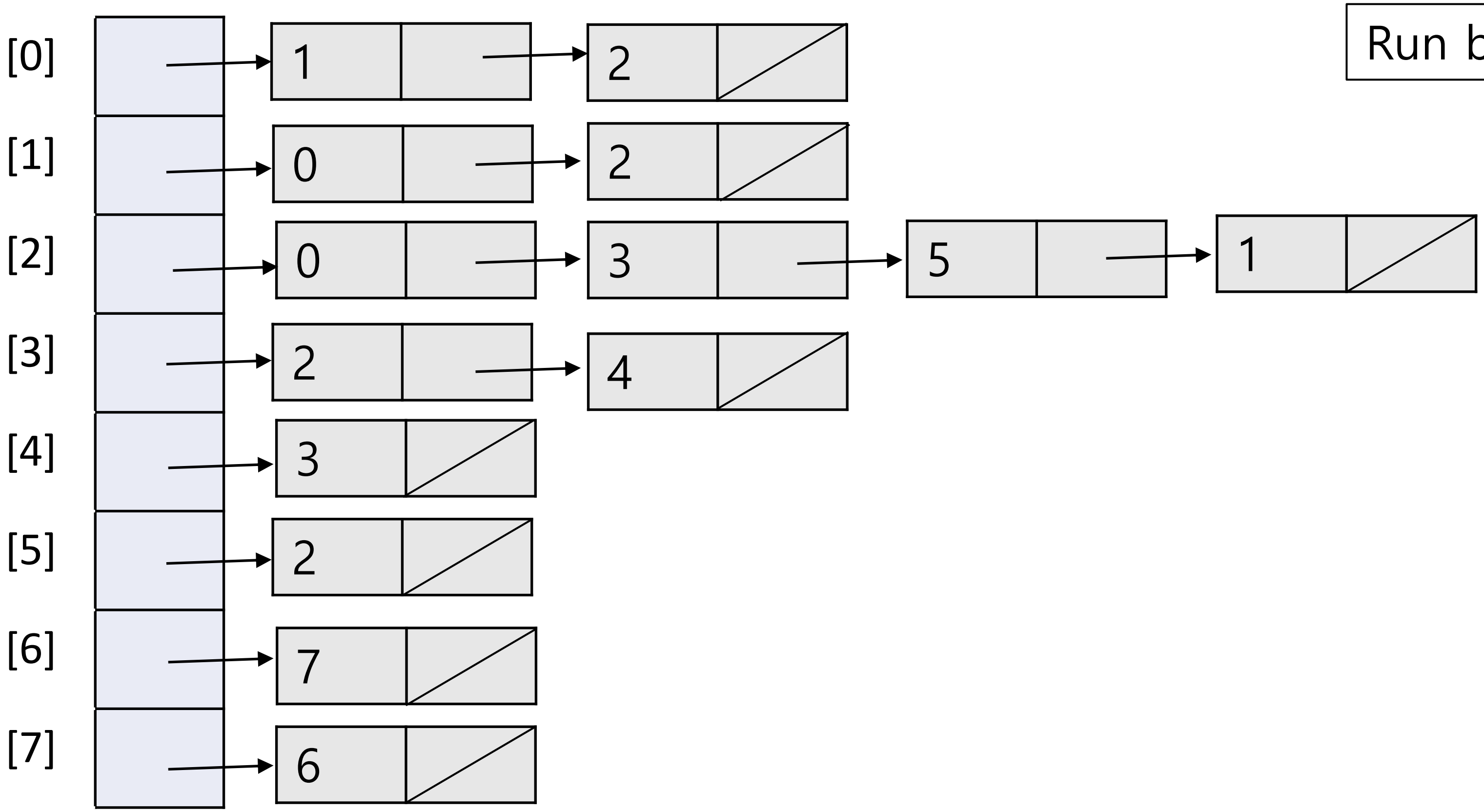

```
void bfs(int v)
{
    nodePointer w;
    front = rear = NULL;
    printf("%5d", v);
    visited[v] = TRUE;
    addq(v);
    while (front) {
        v = deleteq();
        for (w = graph[v]; w; w = w->link)
          if (!visited[w->vertex]) {
              printf("%5d", w->vertex);
              addq(w->vertex);
              visited[w->vertex] = TRUE;
          } /* if */
    } /* while */
}
```

bfs(0);

0 1 2 3 4 5 6 7

Run bfs(0) and show the result.

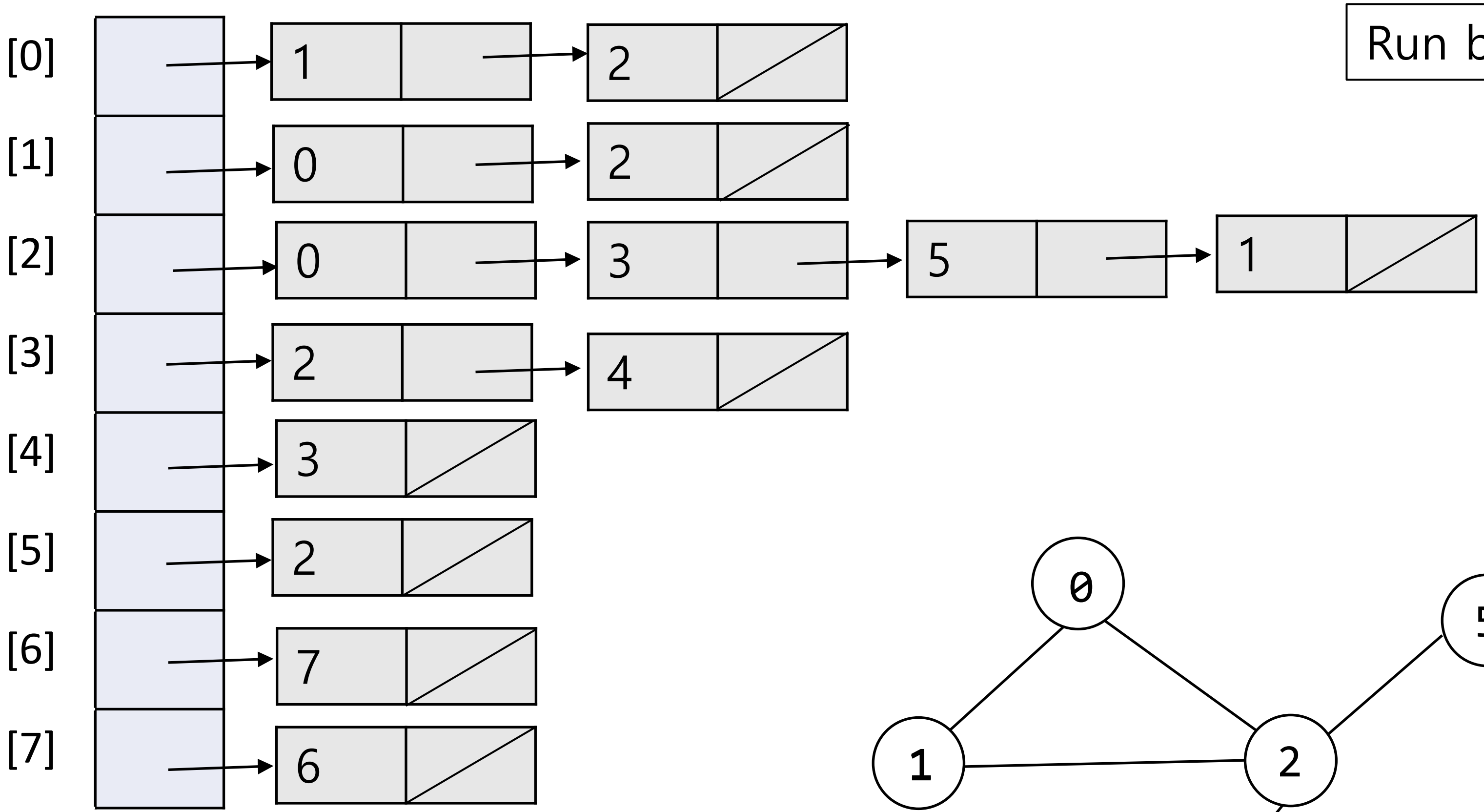
0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역



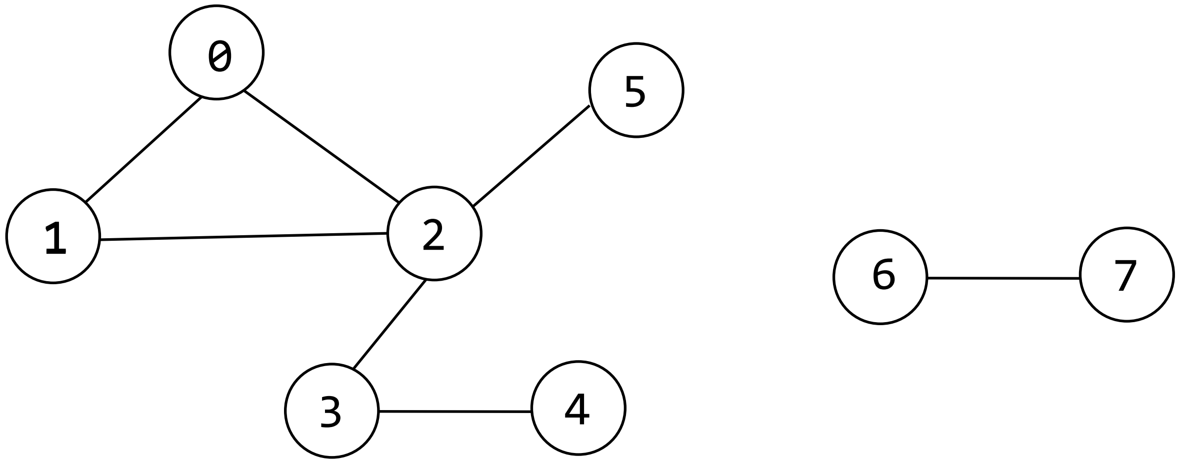

```
void bfs(int v)
{
   nodePointer w;
   front = rear = NULL;
   printf("%5d", v);
   visited[v] = TRUE;
   addq(v);
   while (front) {
      v = deleteq();
      for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex]) {
           printf("%5d", w->vertex);
           addq(w->vertex);
           visited[w->vertex] = TRUE;
        } /* if */
   } /* while */
}
```

Run bfs(0) and show the result.

0	사당
1	아주대
2	강남
3	의왕
4	평촌
5	수지
6	명동
7	서울역




```
void bfs(int v)
{
   nodePointer w;
   front = rear = NULL;
   printf(“%5d”, v);
   visited[v] = TRUE;
   addq(v);
   while (front) {
      v = deleteq();
      for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex]) {
           printf(“%5d”, w->vertex);
           addq(w->vertex);
           visited[w->vertex] = TRUE;
        } /* if */
   } /* while */
}
```



0 1 2 3 5 4

Connected Components (연결요소)

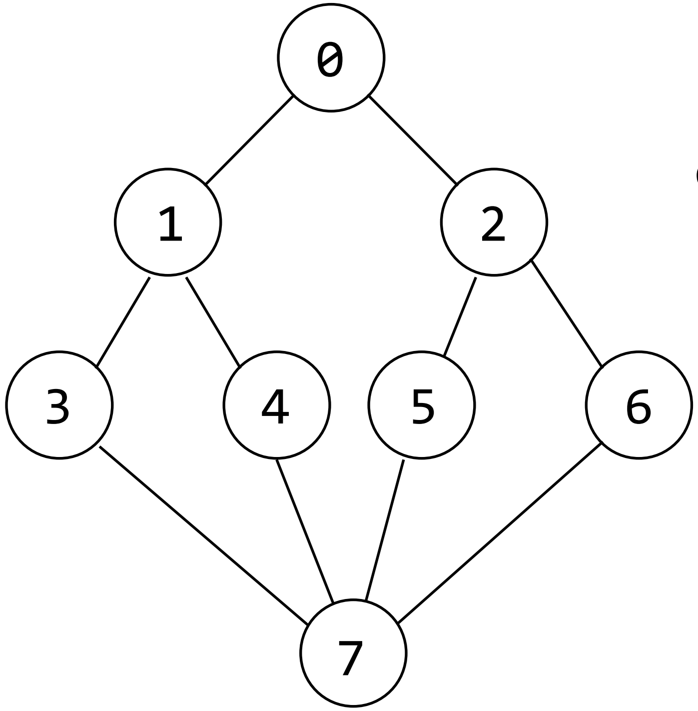
: a component of an undirected graph is a <u>maximal connected subgraph</u>

❖ To determine whether or not an undirected graph is connected
  – simply calling *dfs*(0) or *bfs*(0) and then determine if there are unvisited vertices

❖ To list the connected components of a graph
  – make repeated calls to either *dfs*(*v*) or *bfs*(*v*) where *v* is an unvisited vertex

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

Example of Connected Components



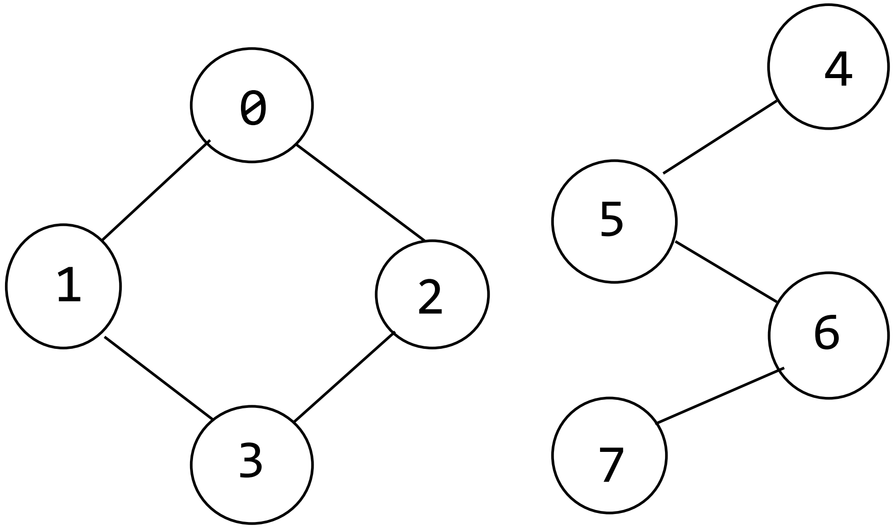
dfs(0);

0 1 3 7 4 5 2 6

dfs(0);
dfs(4);

0 1 3 2

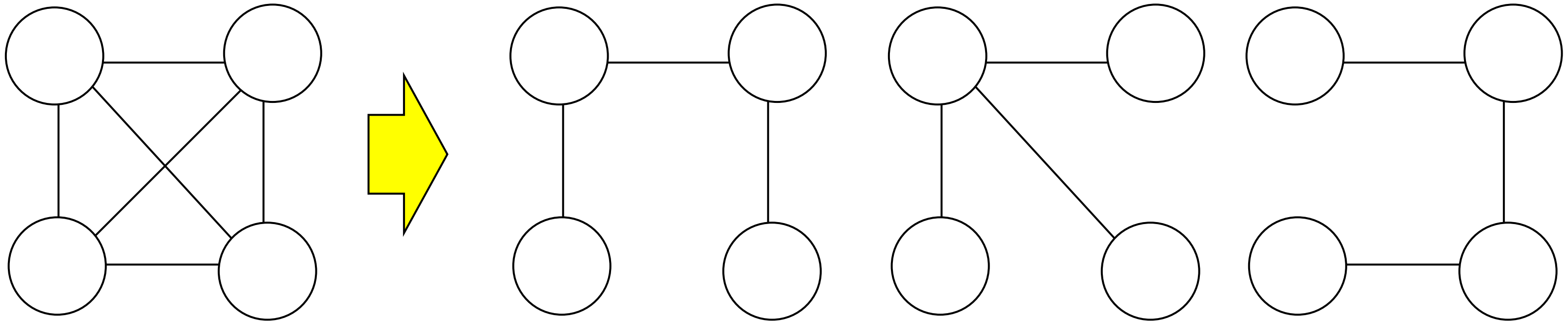
4 5 6 7



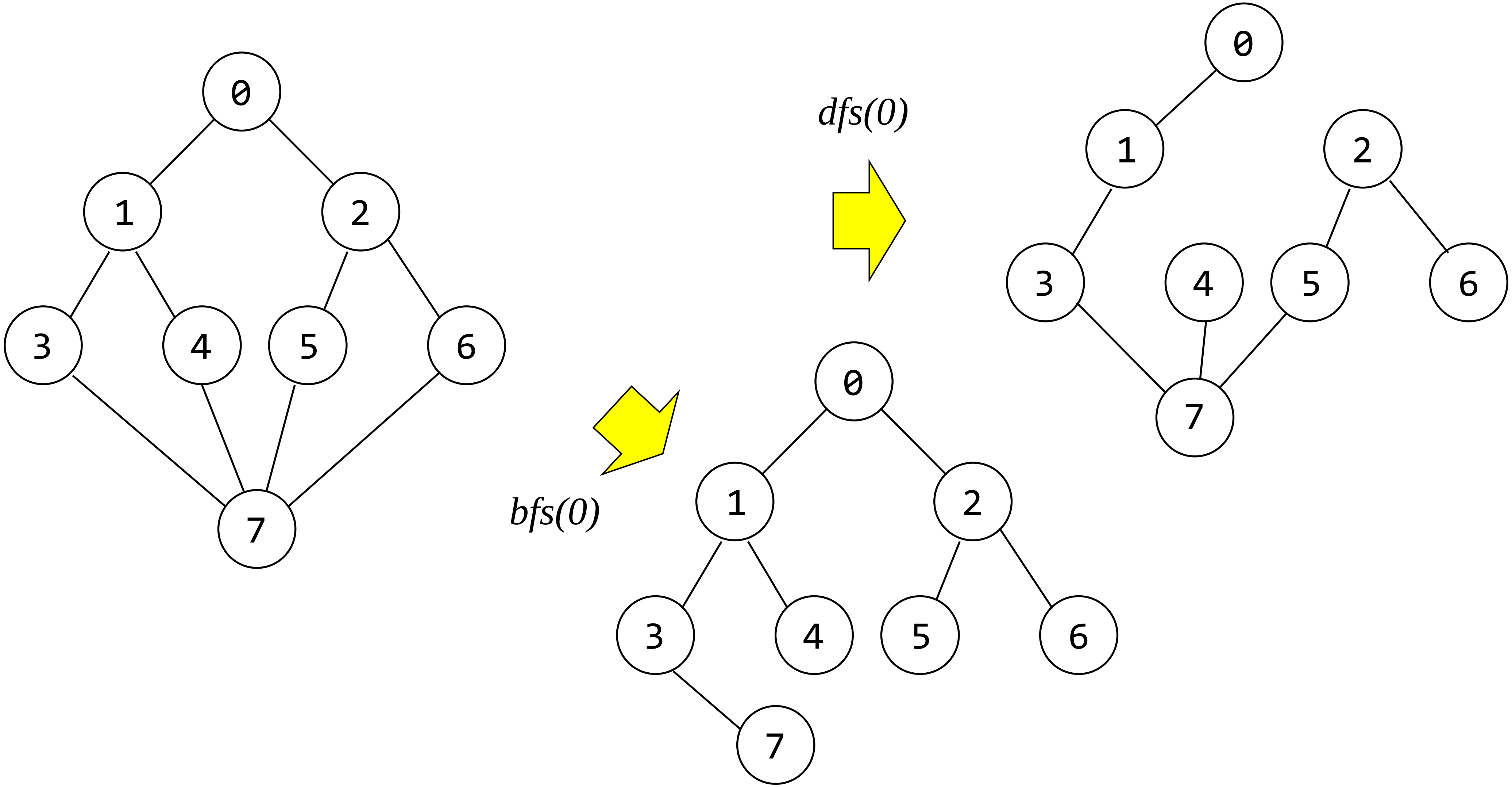
Spanning Trees (신장트리)

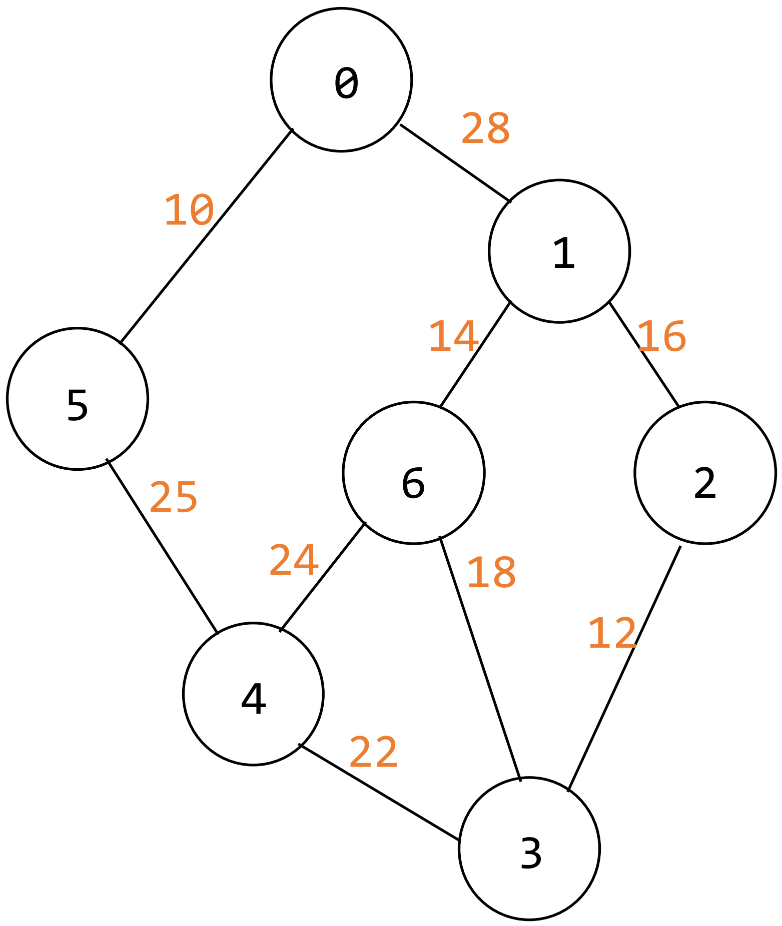
❖ If graph *G* is connected, *dfs()* or *bfs* implicitly partitions the edges in *G* into two sets:
- T : (for tree edges) set of edges used or traversed during the search
- N : (for nontree edges) set of remaining edges

❖ A spanning tree is any tree that consists solely of edges in *G* and that include all the vertices in *G*
1) if we add a nontree edge into a spanning tree → cycle
2) spanning tree is a minimal subgraph, G′, of G such that V(G)=V(G′) and G′ is connected



Example of DFS/BFS Spanning Trees




0
1
2
3
4
5
6
28
10
14
16
25
24
18
12
22

Minimum Cost Spanning Tree

- A spanning tree of least cost
  - Kruskal's algorithm
  - Prim's algorithm
  - Sollin's algorithm

- Applications
  - Network design: telephone, electrical, water, road, …

Minimum Cost Spanning Tree

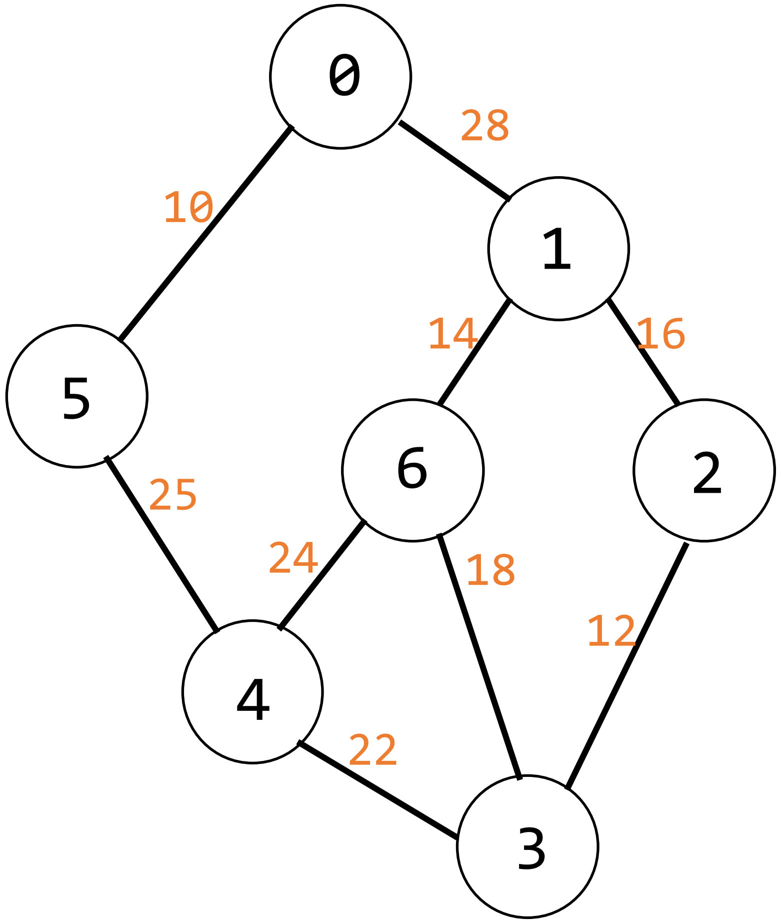
- Greedy method
  - Make the best decision, local optimum, at each stage using some criterion
  - When the algorithm terminates, we hope that the local optimum is equal to the global optimum.
- Spanning tree construction constraints
  - use only edges within the graph
  - use exactly *n*-1 edges
  - may not use edges that would produce a cycle

Kruskal’s Algorithm

```
T = {};
E = a set of every edges in the input graph
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\n”);
```

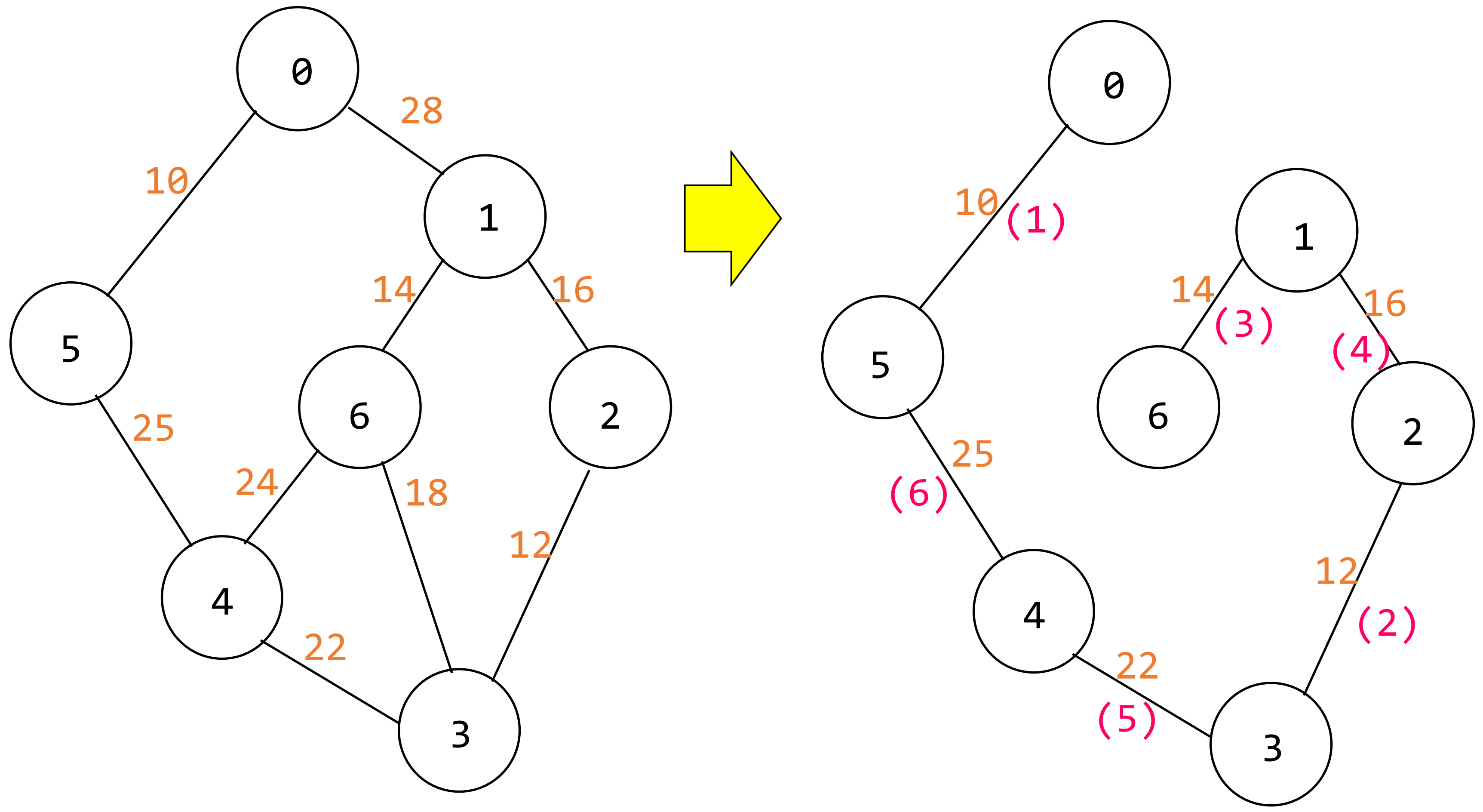
An Example of Kruskal’s Algorithm

```
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\n”);
```



T = {(0,5)(2,3)(1,6)(1,2)(3,4) (5,4)}

An Example of Kruskal's Algorithm

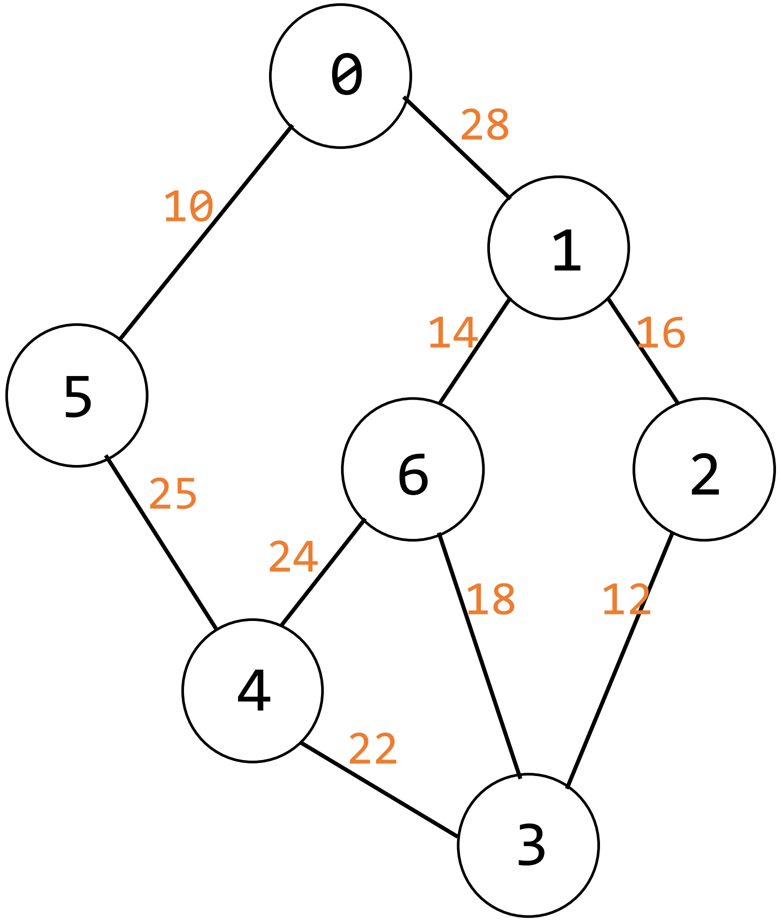


Prim’s Algorithm

```
T = {};
TV = {0}; /* start with vertex 0 and no edge*/
while (T contains fewer 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) to T;
}
if (T contains fewer than n-1 edges)
printf(“no spanning tree\n”);
```

An Example of Prim's Algorithm

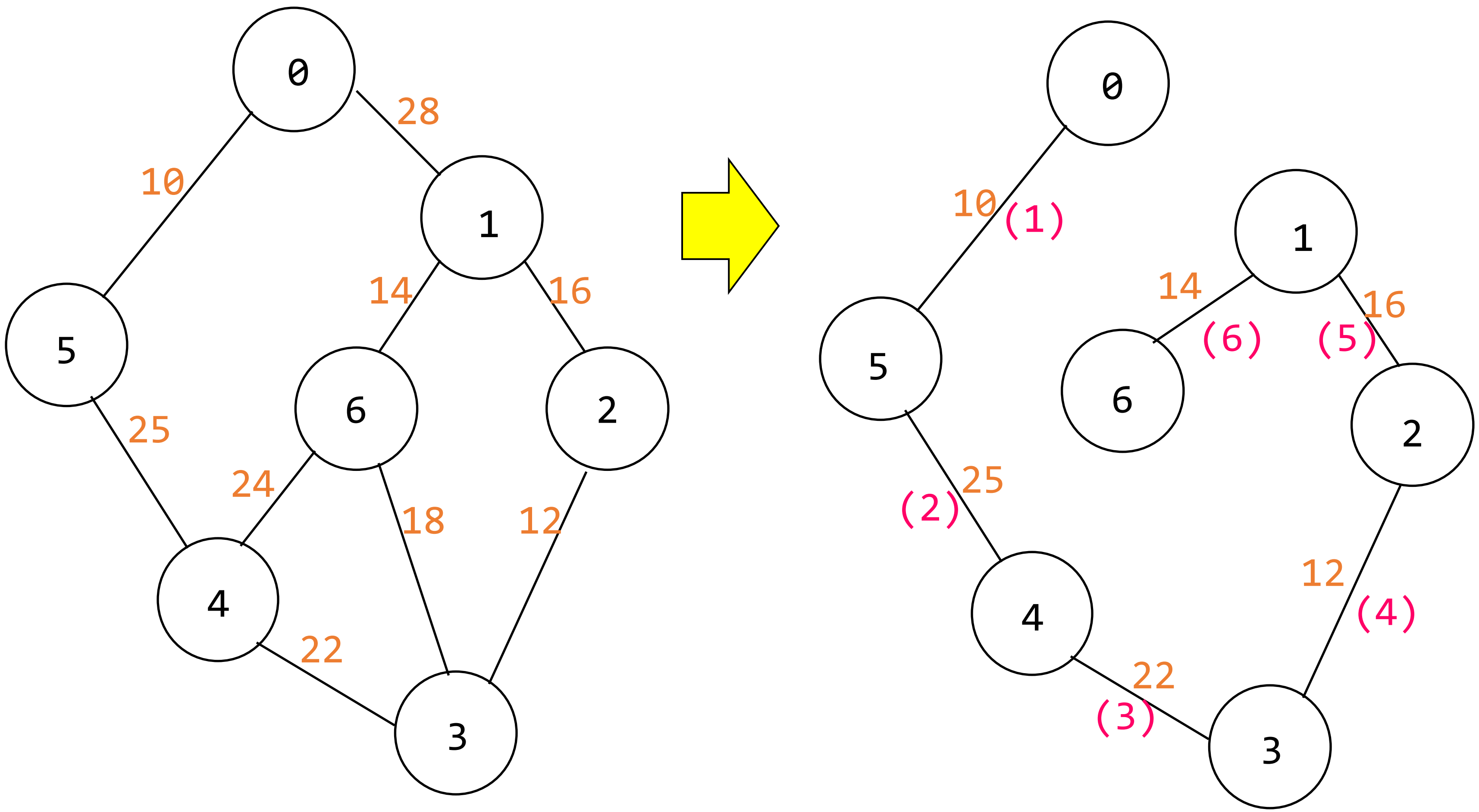
```
T = {};
TV = {0};
while (T contains fewer than n-1 edges)
{
   let (u,v) be a least cost edge such that u ∈ TV and v ∉
  TV;
   if (there is no such edge)
         break;
   add v to TV;
   add (u,v) to T;
}
if (T contains fewer than n-1 edges)
  printf("no spanning tree\n");
```



TV={ 0 5 4 3 2 1 6 }

T={(0,5) (5,4) (4,3) (3,2) (2,1) (1,6) }

An Example of Prim's Algorithm



Sollin's Algorithm

Start with a forest that has no edges

Select **several edges** for inclusion in T at each stage

1) Each component selects a least cost edge with which to connect to another component

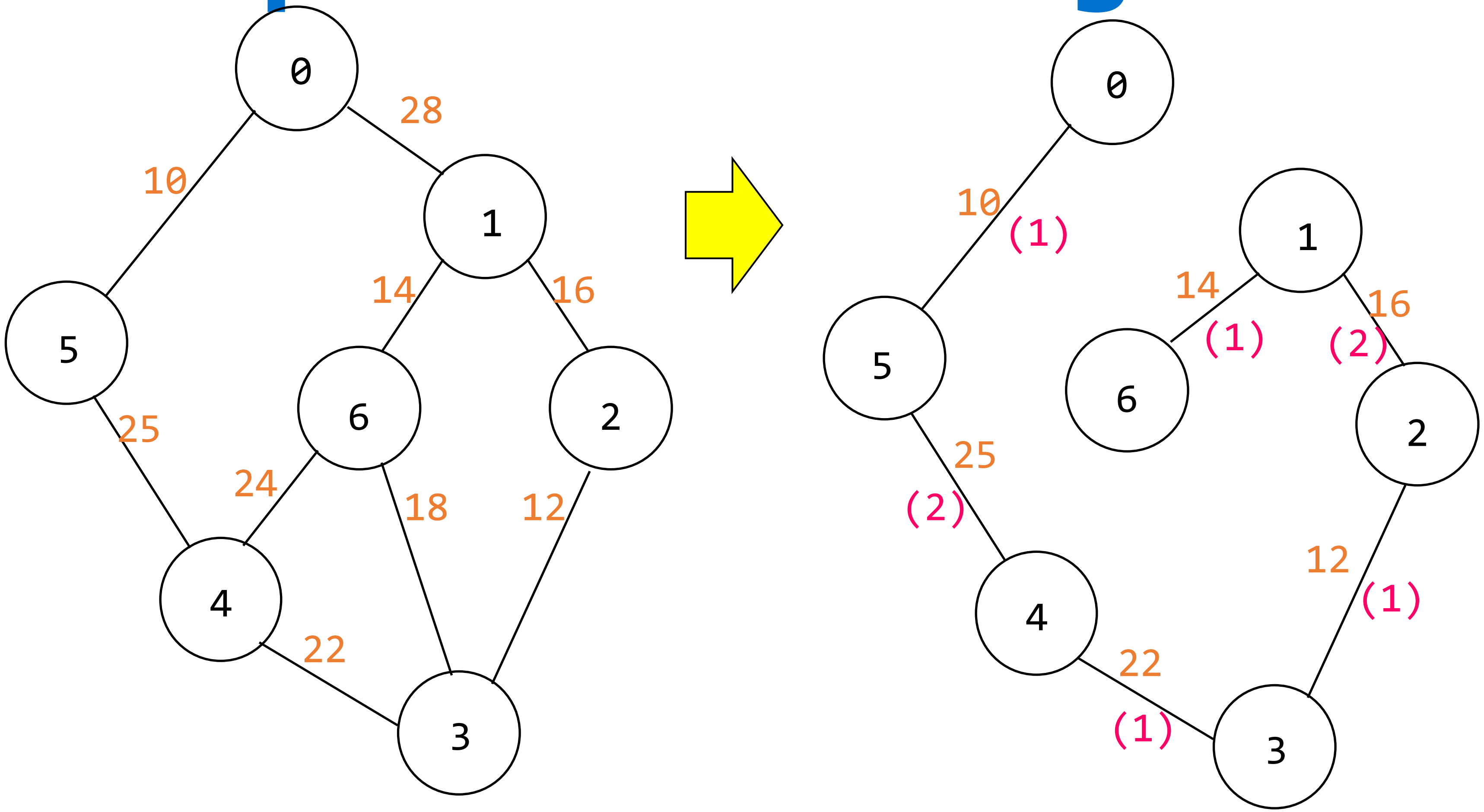
2) Duplicate selections are eliminated

- Cycles are possible when the graph has some edges that have the same cost

Terminate when

- There is only one tree at the end of a stage, or
- No edges remain for selection

An Example of Sollin's Algorithm



First stage: (0,5), (1,6), (2,3), (3,2), (4,3), (5,0), (6,1)

Second stage: (5,4), (1,2)

An Example of Sollin's Algorithm

Start with a forest that has no edges
Select several edges for inclusion in T at each stage

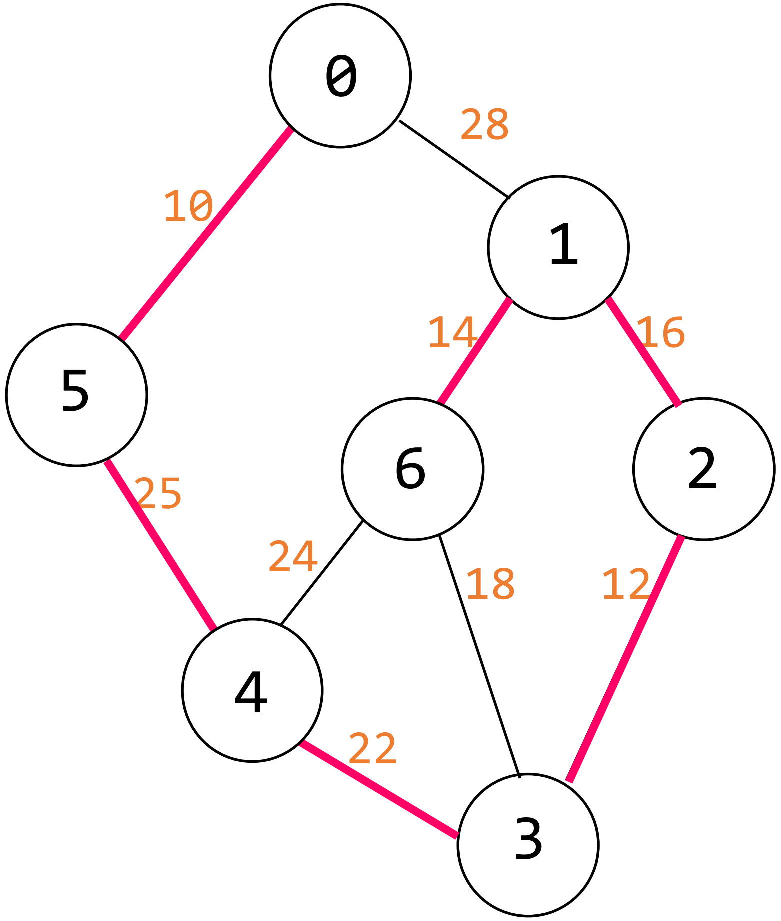
1) Each component selects a least cost edge with which to connect to another component
2) Duplicate selections are eliminated

First stage:

T={(0,5)(1,6)(2,3)(4,3)(6,1)}

Second stage:

T = { (0,5) (1,6) (2,3) (4,3) (5,4) (1,2) }



Shortest Paths

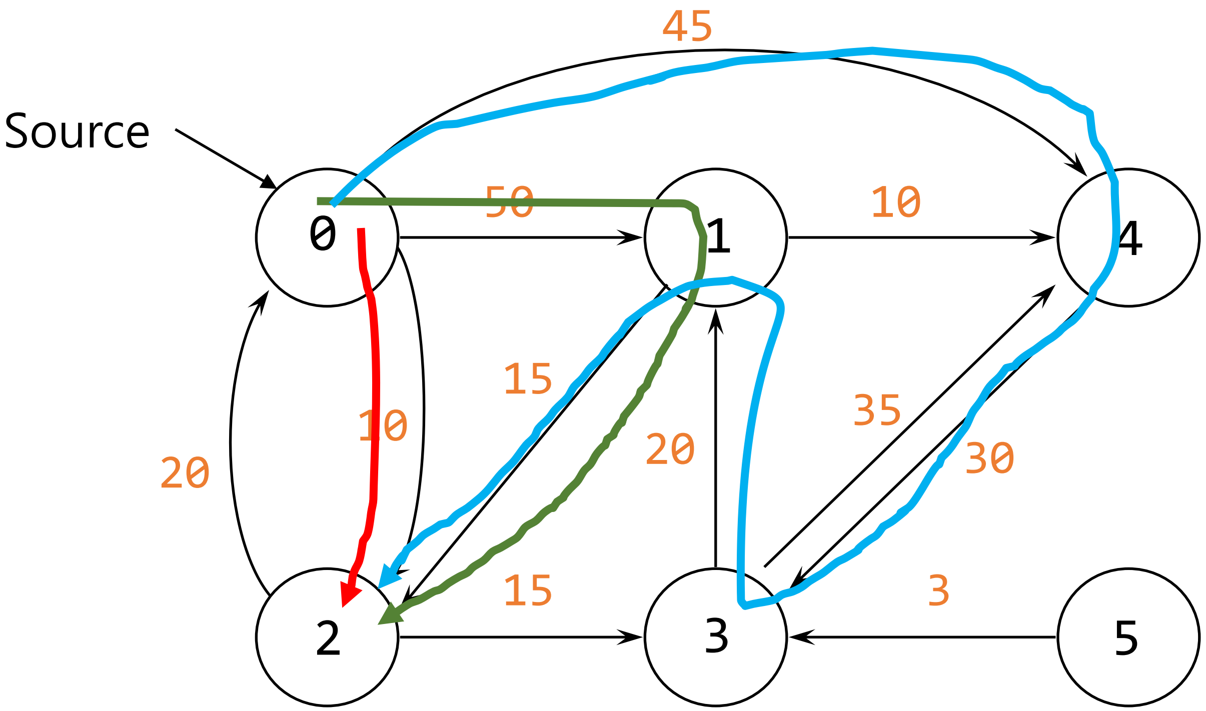
Single source -> all destinations

- Dijkstra algorithm
- Bellman-Ford algorithm

What is the shortest path?

Let's find all paths from 0 to 2.

The length of a path is now defined to be the **sum of the weights** of the edges on that path, rather than the number of edges.



Path	Length
1) 0 2	10
2) 0 2 3	25
3) 0 2 3 1	45
4) 0 4	45

Shortest paths from 0

Shortest Path Observations

v_0 : source vertex

S : set of vertices, including v_0, whose shortest paths have been found

w : any vertex ($\notin S$)

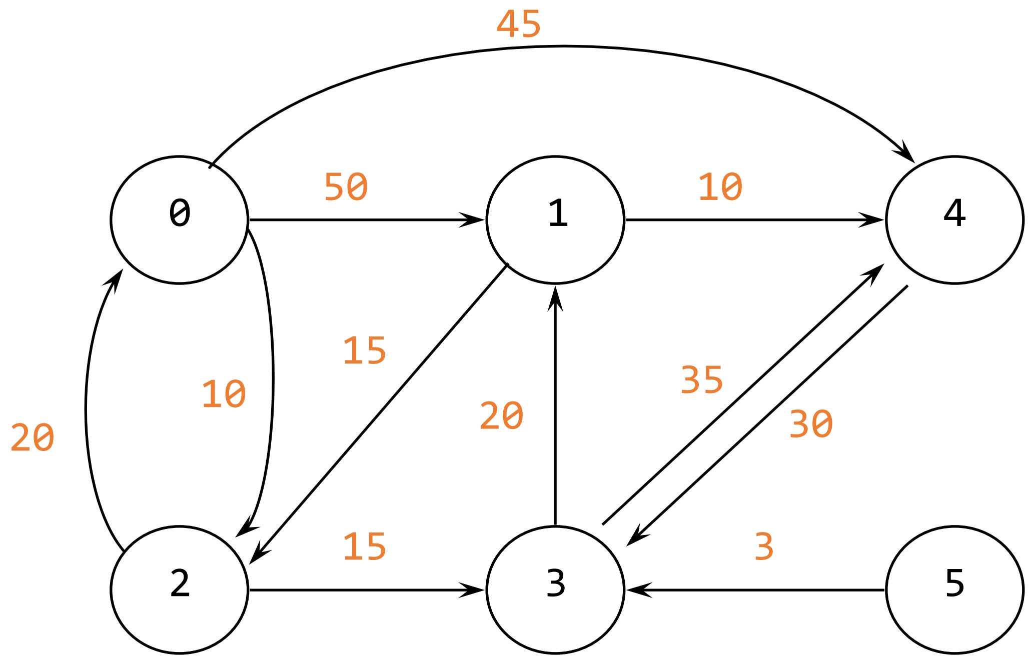
distance[w] : the length of the shortest path starting from v_0, going through vertices only in S, and ending in w

1) If the next shortest path from v_0 is to u, the path from v_0 to u goes through only those vertices that are in S
2) Vertex u is chosen so that it has the minimum distance, distance[u], among all the vertices not in S
3) Shortest path from v_0 to w can be changed by adding u to S

$\rightarrow$ distance [w] = distance[u] + cost(<u, w>)

Implementation of Dijkstra's Algorithm in C (1)

```
#define MAX_VERTICES 6
int cost[][MAX_VERTICES] =
 {{   0,   50,   10, 1000,   45, 1000},
  {1000,    0,   15, 1000,   10, 1000},
  {  20, 1000,    0,   15, 1000, 1000},
  {1000,   20, 1000,    0,   35, 1000},
  {1000, 1000,   30, 1000,    0, 1000},
  {1000, 1000, 1000,    3, 1000,    0}};
int distance[MAX_VERTICES];
short int found[MAX_VERTICES];
int n = MAX_VERTICES;
```



Implementation of Dijkstra’s Algorithm in C (2)

```
void shortestPath(int v, int cost[][MAX_VERTICES],
     int distance[], int n, short int found[])
{
   int i, u, w;
   for (i = 0; i < n; i++) {
       found[i] = FALSE;
       distance[i] = cost[v][i];
   }
   found[v] = TRUE;   distance[v] = 0;
   for (i = 0; i < n-2; i++) {
       u = choose(distance, n, found);
       found[u] = TRUE;
       for (w = 0; w < n; w++)
           if (!found[w])
              if (distance[u]+cost[u][w] < distance[w])
                    distance[w] = distance[u]+cost[u][w];
   } /* for i */
}
```

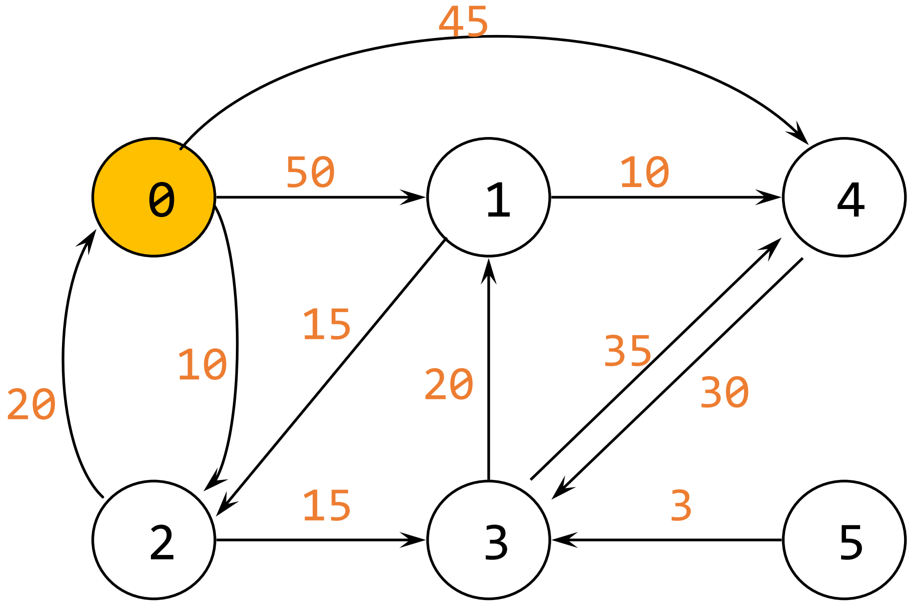
- v : source vertex
- cost[][] : adjacency matrix of a digraph
- distance[i] : the shortest path from vertex v to i
- found[i] : 0 if the shortest path from i has not been found
  1 otherwise ($i \in S$)

Implementation of Dijkstra’s Algorithm in C (3)

```
int choose(int distance[], int n, int found[])
{
  /* find smallest distance not yet checked */
   int i, min, minpos;
   min = INT_MAX;
   minpos = -1;
   for (i = 0; i < n; i++)
       if (distance[i] < min && !found[i]) {
          min = distance[i];
          minpos = i;
       }
   return minpos;
}
```

Dijkstra’s Algorithm Example 1

Initial condition



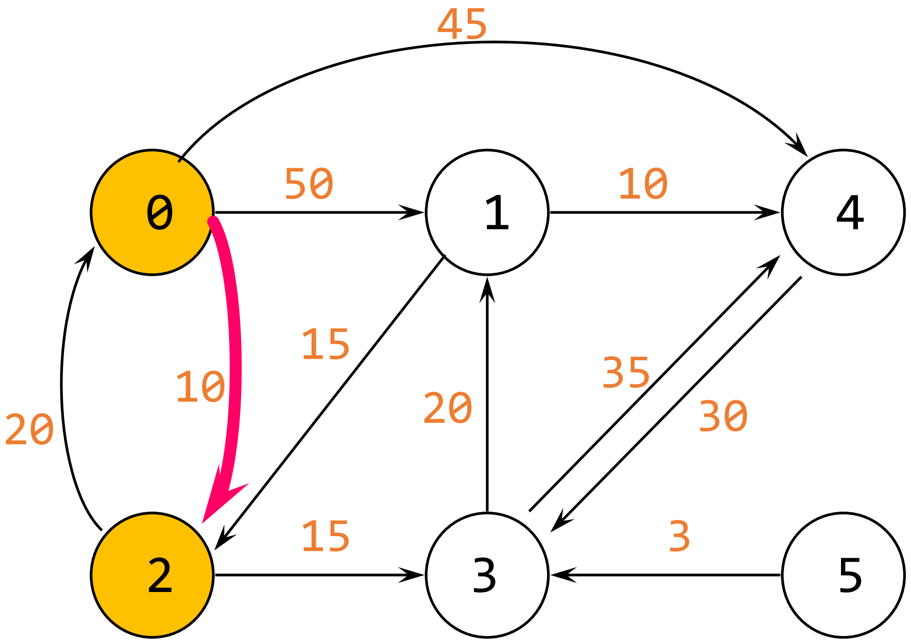
S = {0}

Cost adjacency matrix

$$
\begin{array}{c|cccccc}
 & [0] & [1] & [2] & [3] & [4] & [5] \\
\hline
[0] & 0 & 50 & 10 & \infty & 45 & \infty \\
[1] & \infty & 0 & 15 & \infty & 10 & \infty \\
[2] & 20 & \infty & 0 & 15 & \infty & \infty \\
[3] & \infty & 20 & \infty & 0 & 35 & \infty \\
[4] & \infty & \infty & \infty & 30 & 0 & \infty \\
[5] & \infty & \infty & \infty & 3 & \infty & 0
\end{array}
$$

distance

Dijkstra’s Algorithm Example 1 – 1st Step

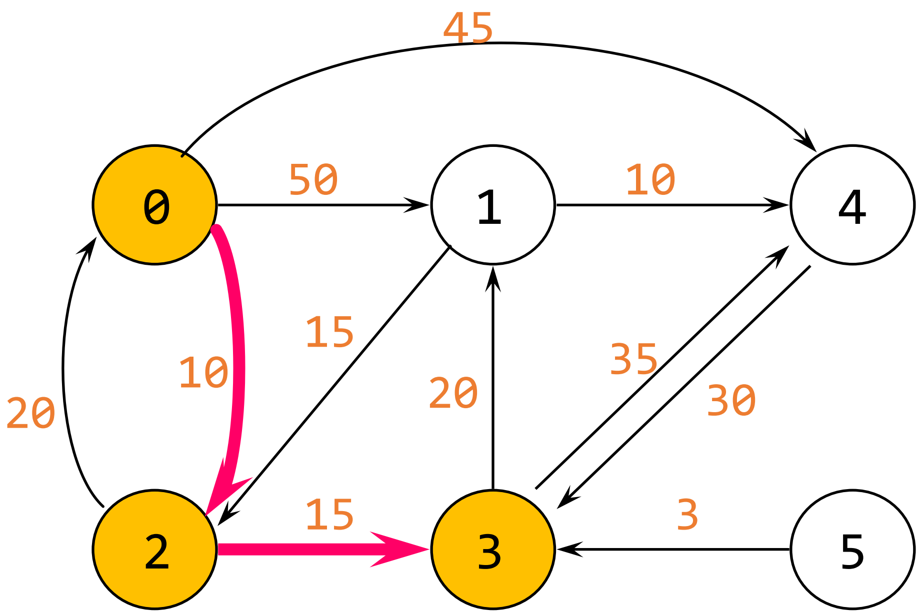


$$
\begin{array}{c|cccccc}
 & [0] & [1] & [2] & [3] & [4] & [5] \\
\hline
[0] & 0 & 50 & 10 & \infty & 45 & \infty \\
[1] & \infty & 0 & 15 & \infty & 10 & \infty \\
[2] & 20 & \infty & 0 & 15 & \infty & \infty \\
[3] & \infty & 20 & \infty & 0 & 35 & \infty \\
[4] & \infty & \infty & \infty & 30 & 0 & \infty \\
[5] & \infty & \infty & \infty & 3 & \infty & 0
\end{array}
$$

S = {0, 2}

0→2: (0, 2), 10

Dijkstra's Algorithm Example 1 – 2nd Step



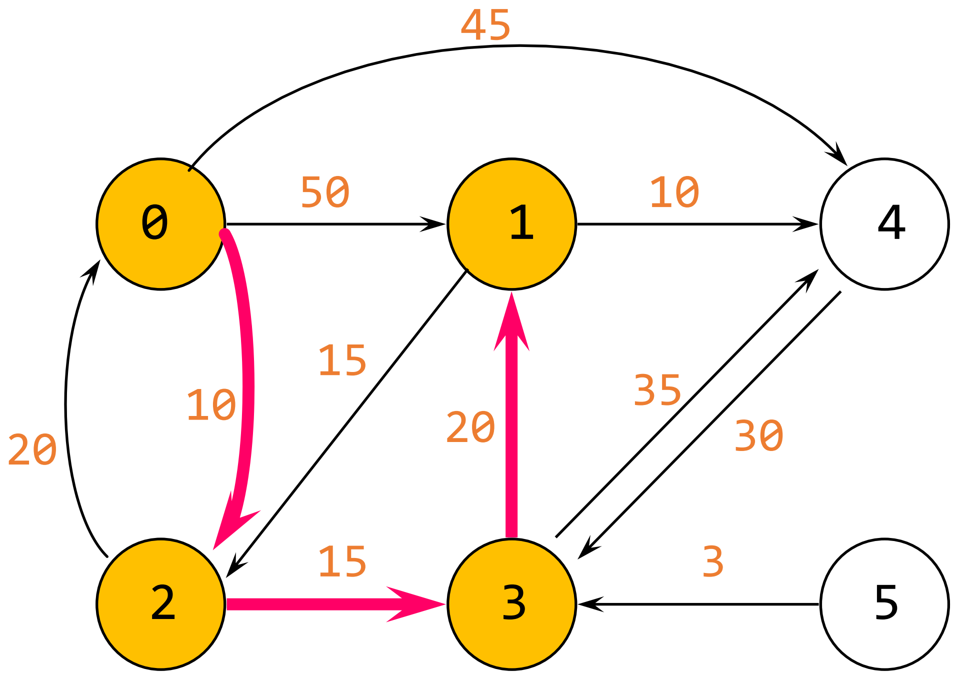
$$
\begin{array}{c|cccccc}
 & [0] & [1] & [2] & [3] & [4] & [5] \\
\hline
[0] & 0 & 45 & 10 & 25 & 45 & \infty \\
[1] & \infty & 0 & 15 & \infty & 10 & \infty \\
[2] & 20 & \infty & 0 & 15 & \infty & \infty \\
[3] & \infty & 20 & \infty & 0 & 35 & \infty \\
[4] & \infty & \infty & \infty & 30 & 0 & \infty \\
[5] & \infty & \infty & \infty & 3 & \infty & 0
\end{array}
$$

S = {0, 2, 3}

0→2: (0, 2), 10

0→3: (0, 2, 3), 25

Dijkstra's Algorithm Example 1 – 3rd Step



	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	45	10	25	45	∞
[1]	∞	0	15	∞	10	∞
[2]	20	∞	0	15	∞	∞
[3]	∞	20	∞	0	35	∞
[4]	∞	∞	∞	30	0	∞
[5]	∞	∞	∞	3	∞	0

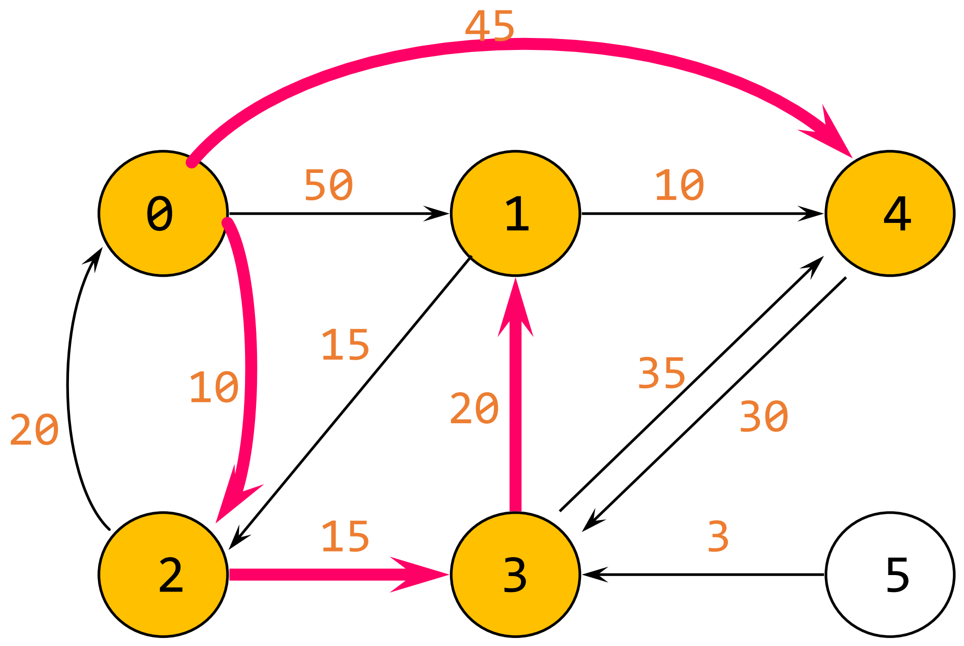
S = {0, 2, 3, 1}

0→2: (0, 2), 10

0→3: (0, 2, 3), 25

0→1: (0, 2, 3, 1), 45

Dijkstra’s Algorithm Example 1 – 4th Step



$$
\begin{array}{c|c}
 & \begin{matrix} [0] & [1] & [2] & [3] & [4] & [5] \end{matrix} \\
\begin{matrix} [0] \\ [1] \\ [2] \\ [3] \\ [4] \\ [5] \end{matrix} &
\begin{bmatrix}
0 & 45 & 10 & 25 & 45 & \infty \\
\infty & 0 & 15 & \infty & 10 & \infty \\
20 & \infty & 0 & 15 & \infty & \infty \\
\infty & 20 & \infty & 0 & 35 & \infty \\
\infty & \infty & \infty & 30 & 0 & \infty \\
\infty & \infty & \infty & 3 & \infty & 0
\end{bmatrix}
\end{array}
$$

S = {0, 2, 3, 1, <u>4</u>}

0→2: (0, 2), 10

0→3: (0, 2, 3), 25

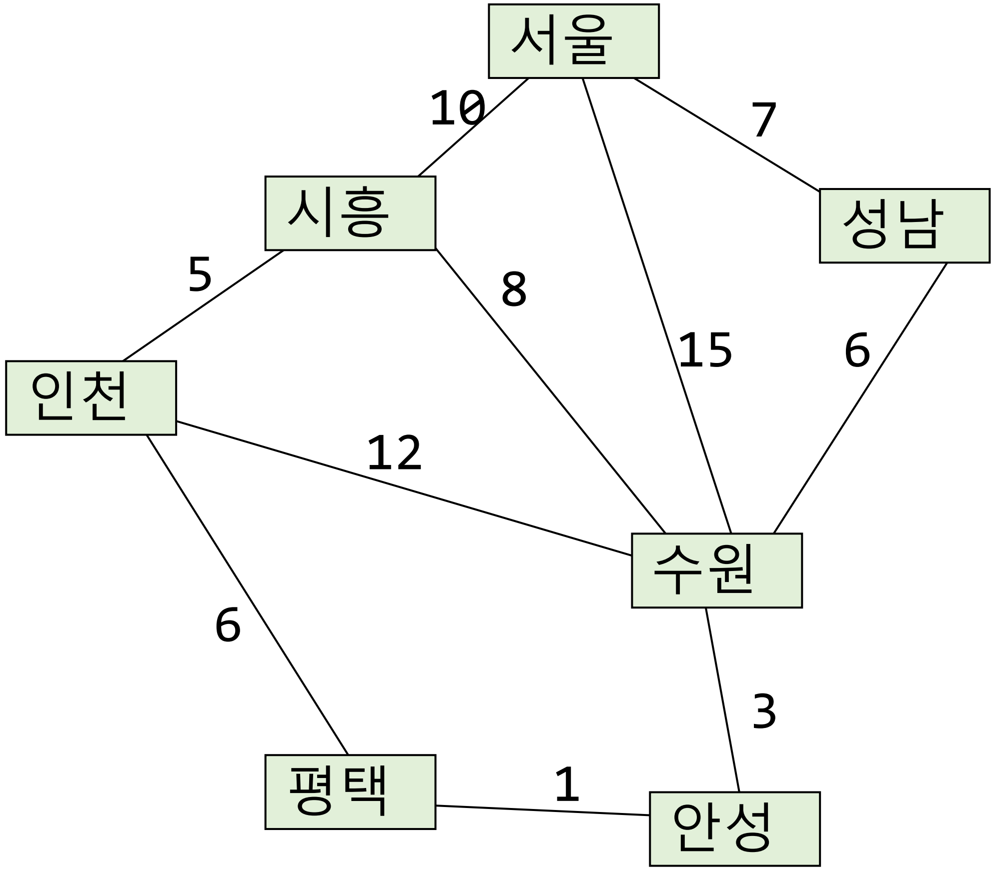
0→1: (0, 2, 3, 1), 45

$\boxed{0 \to 4: (0, 4), 45}$

Implementation of Dijkstra's Algorithm in C (4)

```
void shortestPath(int v, int cost[][MAX_VERTICES],
     int distance[], int n, short int found[], int pi[])     /* pi: an array of predecessors for each vertex */
{
   int i, u, w;
   for (i = 0; i < n; i++) {
       found[i] = FALSE;
       distance[i] = cost[v][i];
       if (distance[i] < MAX_DISTANCE) pi[i]=v; else pi[i]=-1;
   }
   found[v] = TRUE;   distance[v] = 0;
   for (i = 0; i < n-2; i++) {
       u = choose(distance, n, found);
       found[u] = TRUE;
       for (w = 0; w < n; w++)
           if (!found[w])
              if (distance[u]+cost[u][w] < distance[w]){
                     distance[w] = distance[u]+cost[u][w]; pi[w]=u;
              }
   } /* for i */
}
```

Dijkstra’s Algorithm: Example 2

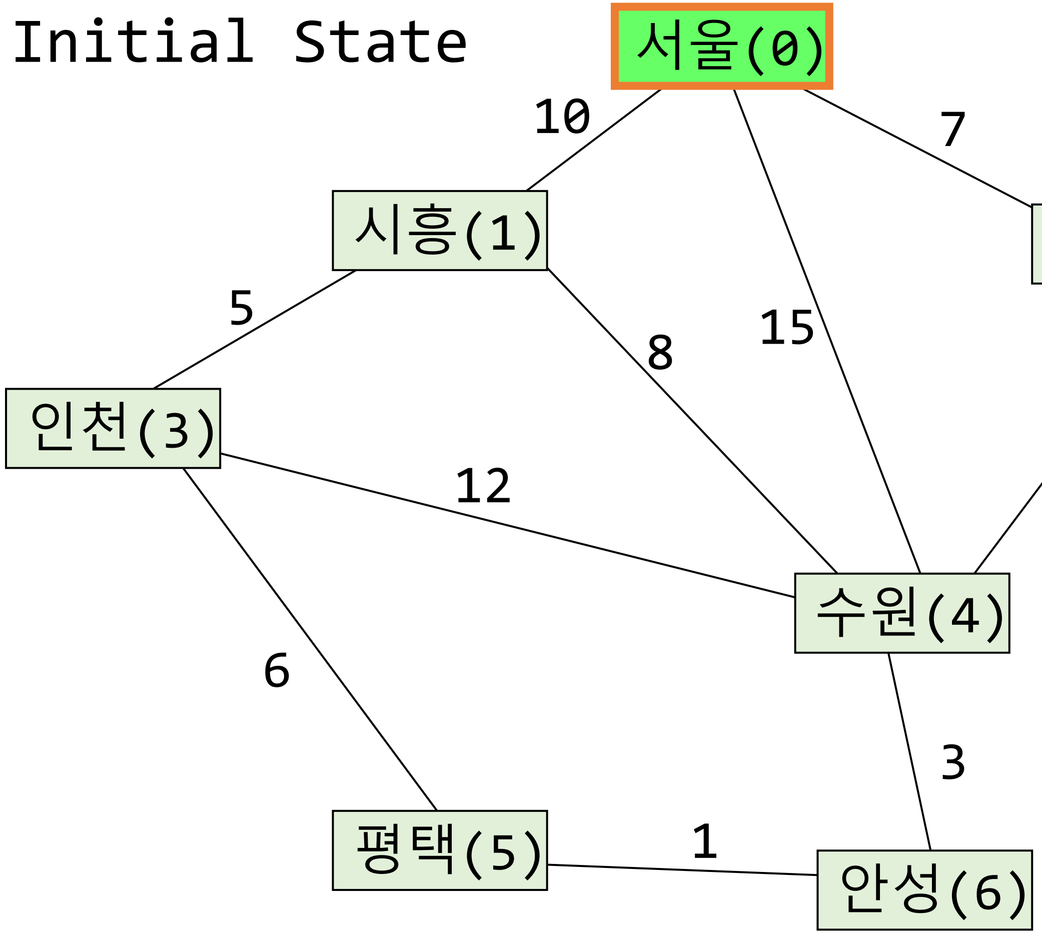


서울:0, 시흥:1, 성남:2, 인천:3, 수원:4, 평택:5, 안성:6

Cost adjacency matrix

$$\begin{bmatrix} 0 & 10 & 7 & 1000 & 15 & 1000 & 1000 \\ 10 & 0 & 1000 & 5 & 8 & 1000 & 1000 \\ 7 & 1000 & 0 & 1000 & 6 & 1000 & 1000 \\ 1000 & 5 & 1000 & 0 & 12 & 6 & 1000 \\ 15 & 8 & 6 & 12 & 0 & 1000 & 3 \\ 1000 & 1000 & 1000 & 6 & 1000 & 0 & 1 \\ 1000 & 1000 & 1000 & 1000 & 3 & 1 & 0 \end{bmatrix}$$

Initial State



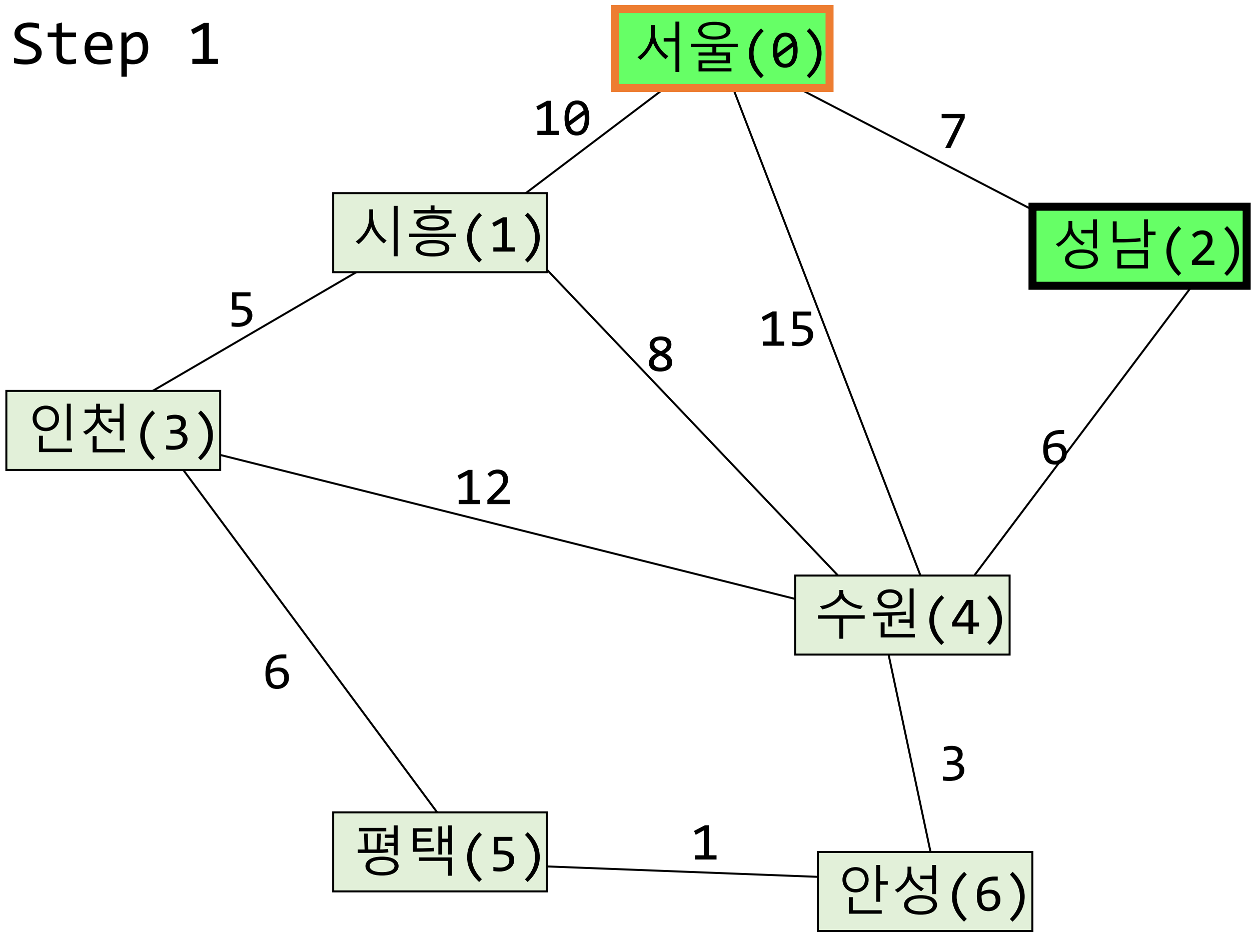
	distance	found
[0]	0	1
[1]	10	0
[2]	7	0
[3]	1000	0
[4]	15	0
[5]	1000	0
[6]	1000	0

S={서울(0)}

pi

[0]	[1]	[2]	[3]	[4]	[5]	[6]
0	0	0	-1	0	-1	-1

Step 1



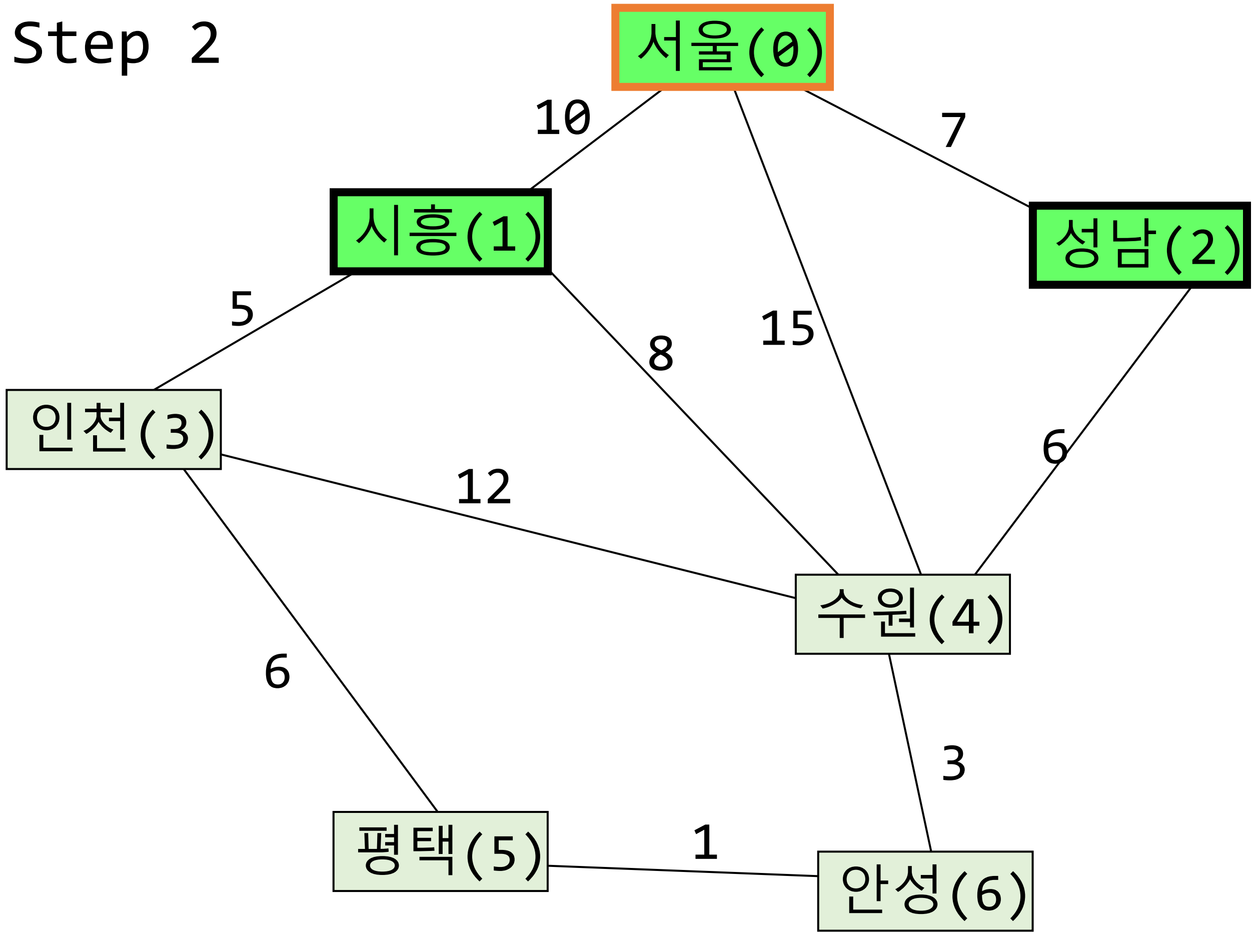
	distance	found
[0]	0	1
[1]	10	0
[2]	7	1
[3]	1000	0
[4]	15 13	0
[5]	1000	0
[6]	1000	0

(서울, 성남)+(성남, 수원)

S={서울(0), **성남(2)**}

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
pi	0	0	0	-1	2	-1	-1

Step 2



	distance
[0]	0
[1]	10
[2]	7
[3]	1000 15
[4]	13
[5]	1000
[6]	1000

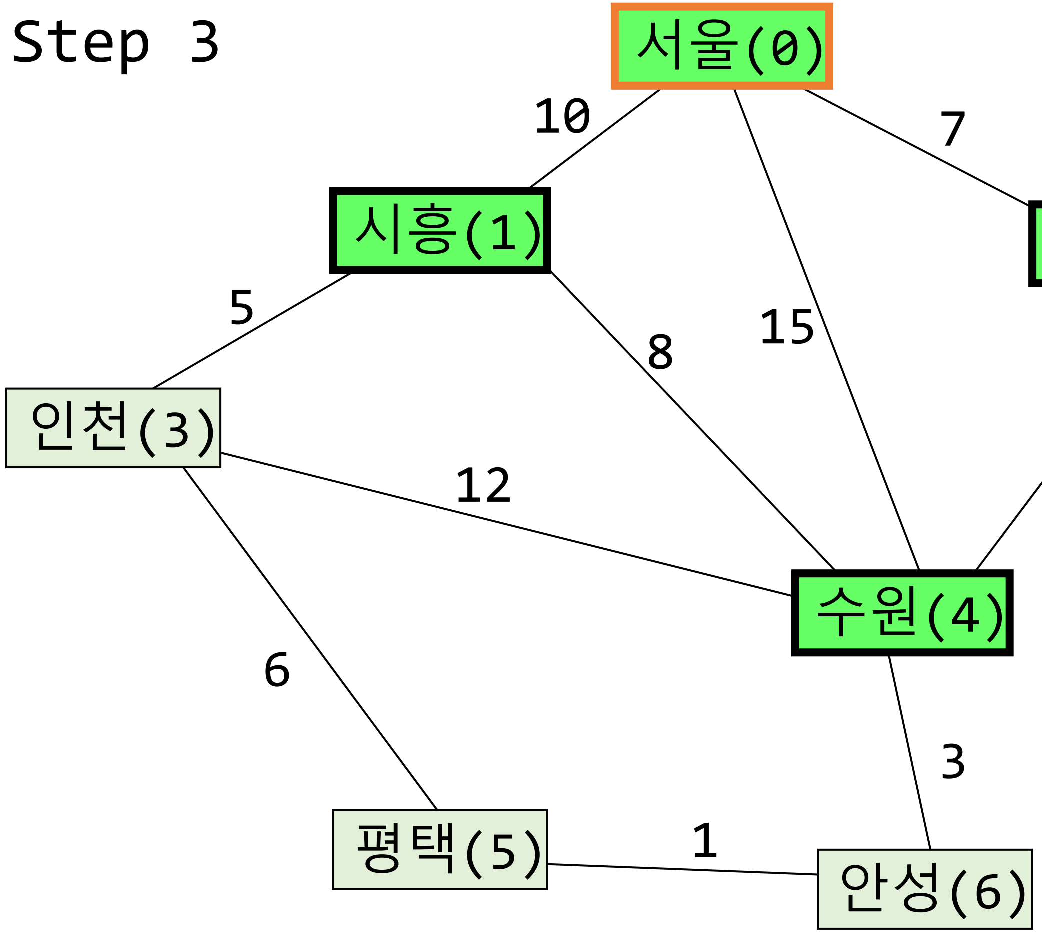
	found
[0]	1
[1]	1
[2]	1
[3]	0
[4]	0
[5]	0
[6]	0

(서울, 시흥)+(시흥,인천)

S={서울(0), 성남(2), **시흥(1)**}

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
pi	0	0	0	1	2	-1	-1

Step 3



	distance
[0]	0
[1]	10
[2]	7
[3]	15
[4]	13
[5]	1000
[6]	1000 16

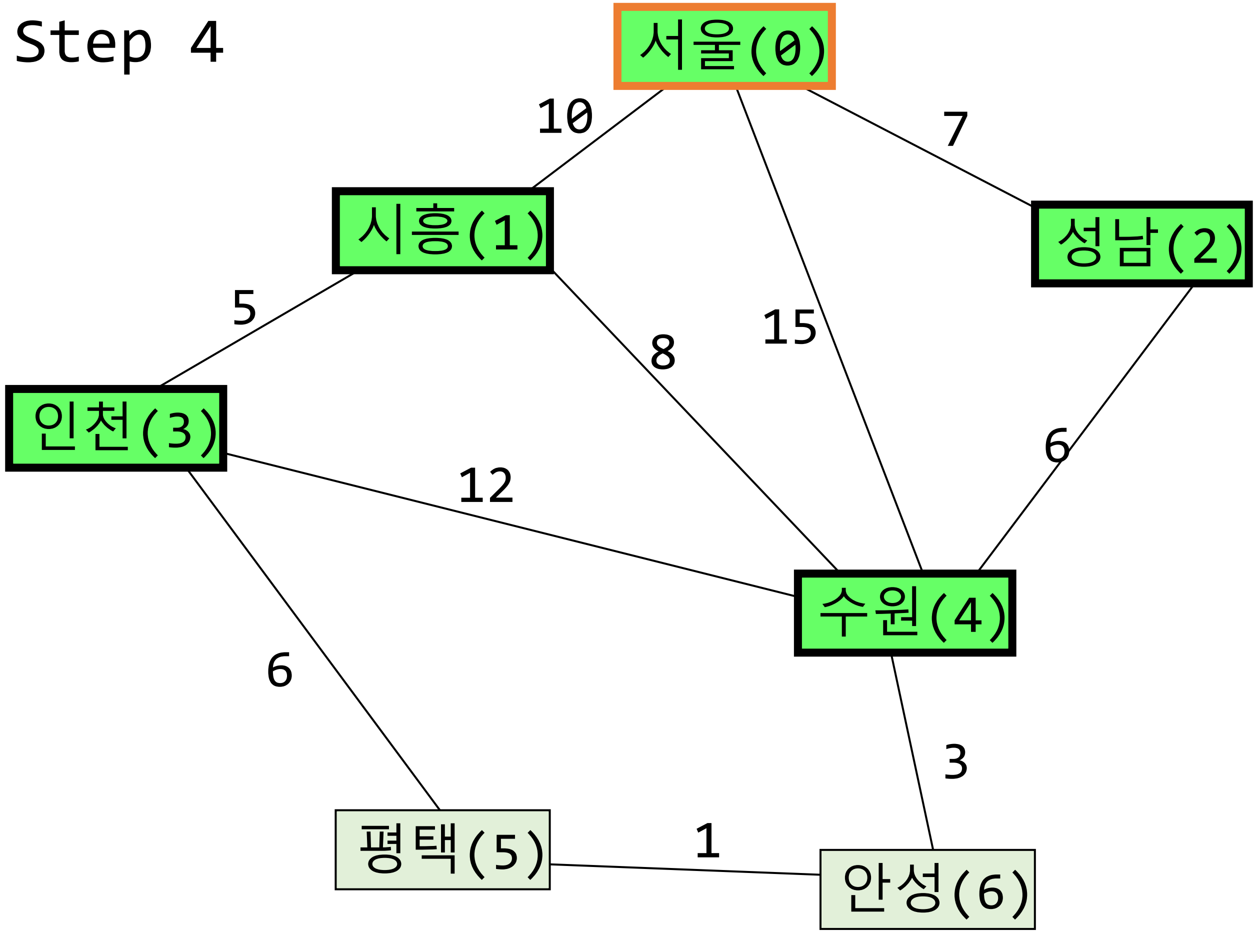
	found
[0]	1
[1]	1
[2]	1
[3]	0
[4]	1
[5]	0
[6]	0

(서울, 수원)+(수원,안성)

S={서울(0), 성남(2), 시흥(1), 수원(4)}

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
pi	0	0	0	1	2	-1	4

Step 4



	distance
[0]	0
[1]	10
[2]	7
[3]	15
[4]	13
[5]	~~1000~~ 21
[6]	16

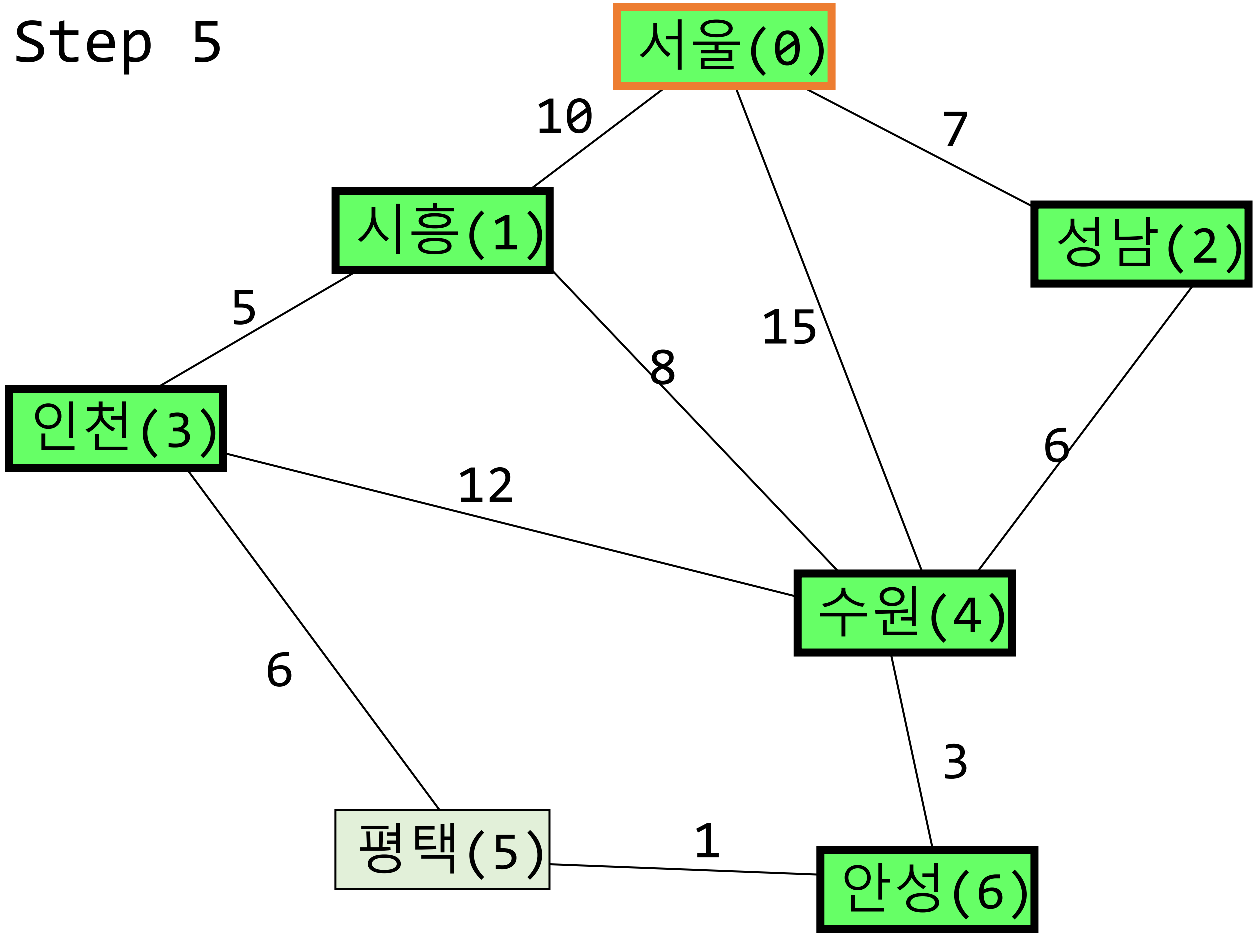
	found
[0]	1
[1]	1
[2]	1
[3]	1
[4]	1
[5]	0
[6]	0

(서울, 인천)+(인천,평택)

S={서울(0), 성남(2), 시흥(1), 수원(4), **인천(3)** }

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
pi	0	0	0	1	2	3	4

Step 5



	distance	found
[0]	0	1
[1]	10	1
[2]	7	1
[3]	15	1
[4]	13	1
[5]	21 17	0
[6]	16	1

(서울, 안성)+(안성,평택)

S={서울(0), 성남(2), 시흥(1), 수원(4), 인천(3), **안성(6)** }

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
pi	0	0	0	1	2	**6**	4

How to determine the path

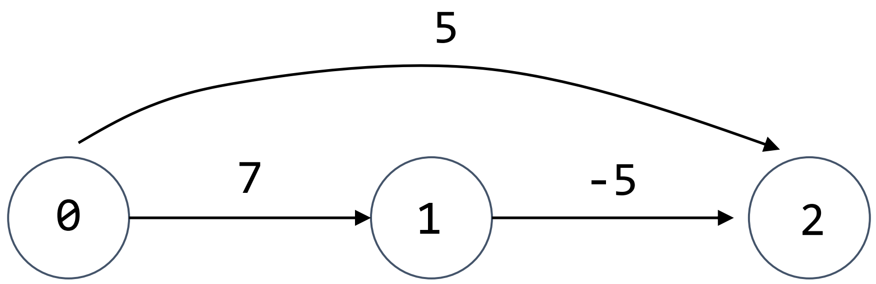
S={서울(0), 성남(2), 시흥(1), 수원(4), 인천(3), 안성(6), 평택(5) }

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
pi	0	0	0	1	2	6	4

서울 → 평택 서울(0) 성남 (2) 수원 (4) 안성(6) 평택(5)

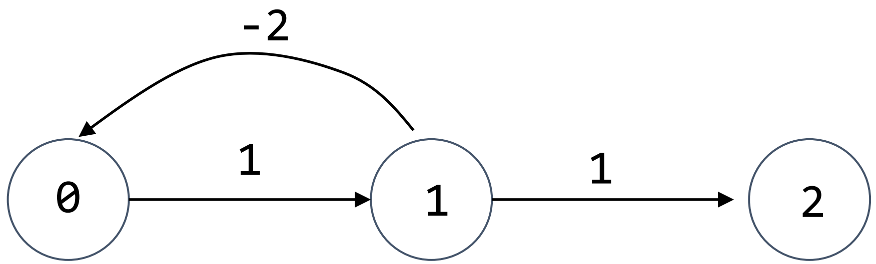
서울 → 인천 ?

Negative Edge Cases



Shortest path from 0 to 2 ? 0,1,2

Dikstra's algorithm cannot find this shortest path



Cost of the shortest path between 0 and 2 ?

0,1,0,1,0,1,0,1,…,2 → -∞

There is no shortest path!!

Bellman-Ford's algorithm: Overview

Allows negative weights

If there is a negative cycle, returns "a negative cycle exists"

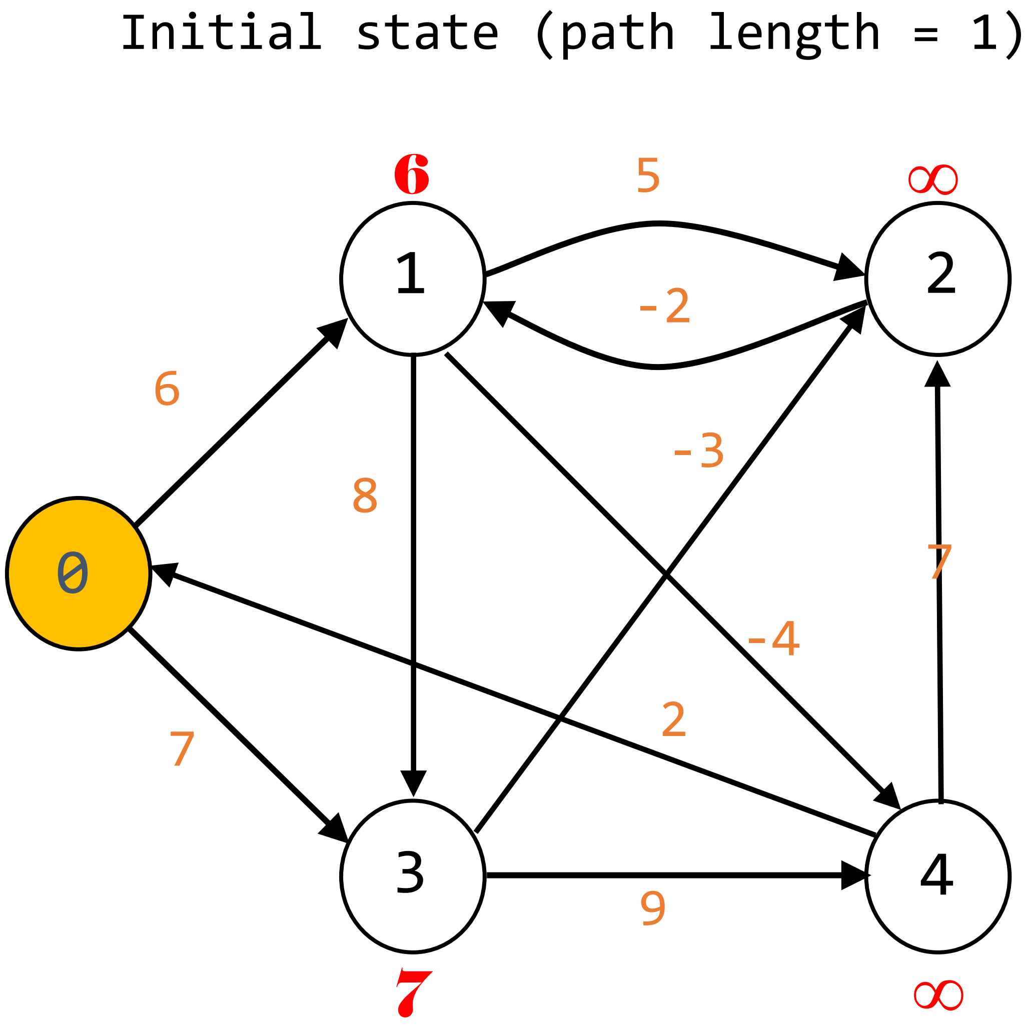
The idea:

- There is a shortest path from s to any other vertex that does not contain a negative cycle
- The maximal number of edges in such a path with no cycles is $|V|-1$, because it can have at most $|V|$ nodes on the path if there is no cycle
  $\Rightarrow$ it is enough to check paths of up to $|V|-1$ edges.

Bellman-Ford Algorithm

```
void BellmanFord(int n, int v)
{ /* single source all destination shortest paths with negative edge lengths. */
  for (int i=0; i<n; i++)
    dist[i] = length[v][i];  /* initialize dist */

  for (int k =2; k <= n-1; k++)
     for (each u such that u != v and u has at least one incoming edge)
        for (each <i, u> in the edge)
            if (dist[u] > dist[i] + length[i][u])
                 dist[u] = dist[i] + length[i][u];
  for each edge (u,w) in E
     if (dist(w) > dist(u)+length[u][w])
          return FALSE;                        /* negative cycle detection */
}
```

Bellman-Ford's algorithm: Example

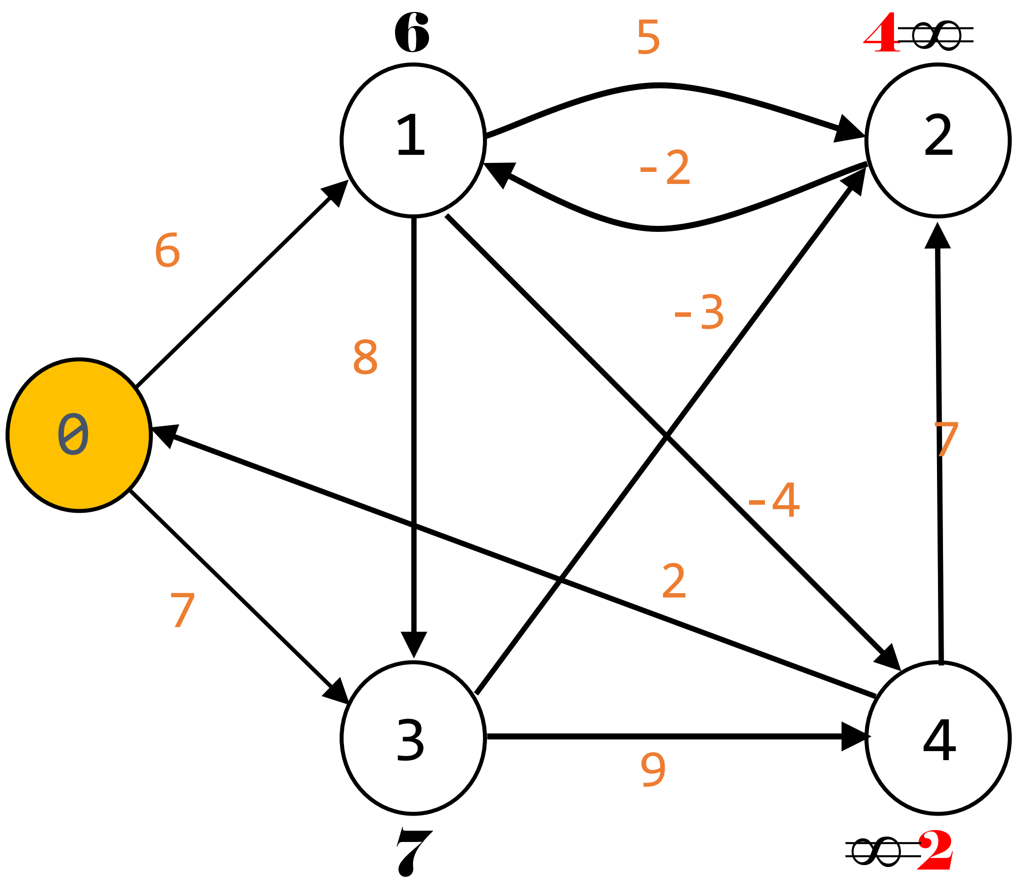


dist[1]= 6 (0,1)
dist[2]= ∞
dist[3]= 7 (0,3)
dist[4]= ∞

	[0]	[1]	[2]	[3]	[4]
[0]	0	6	∞	7	∞
[1]	∞	0	5	8	−4
[2]	∞	−2	0	∞	∞
[3]	∞	∞	−3	0	9
[4]	2	∞	7	∞	0

Path length = 2

```
for each <i,u>,
  min{dist[u], dist[i] + length[i][u]}
```



dist[1]= 6 (0,1)
  min{6, ∞+(-2)}

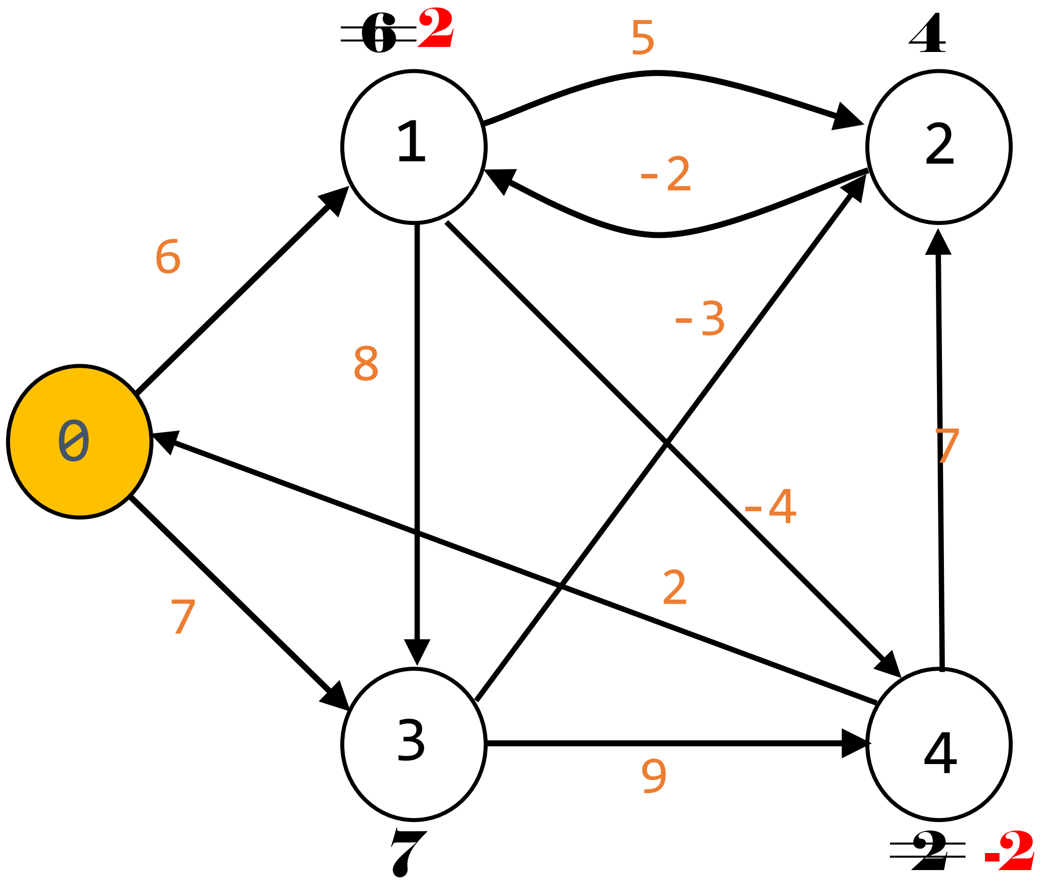
dist[2]= ∞ (0,3,2)
  min{∞, 6+5, 7+(-3), ∞+7}

dist[3]= 7 (0,3)
  min{7, 6+8}

dist[4]= ∞ (0,1,4)
  min{∞, 6+(-4), 7+9}

Path length = 3

```
for each <i,u>,
  min{dist[u], dist[i] + length[i][u]}
```



dist[1]= 6 (0,3,2,1)
  min{6, 4+(-2)}

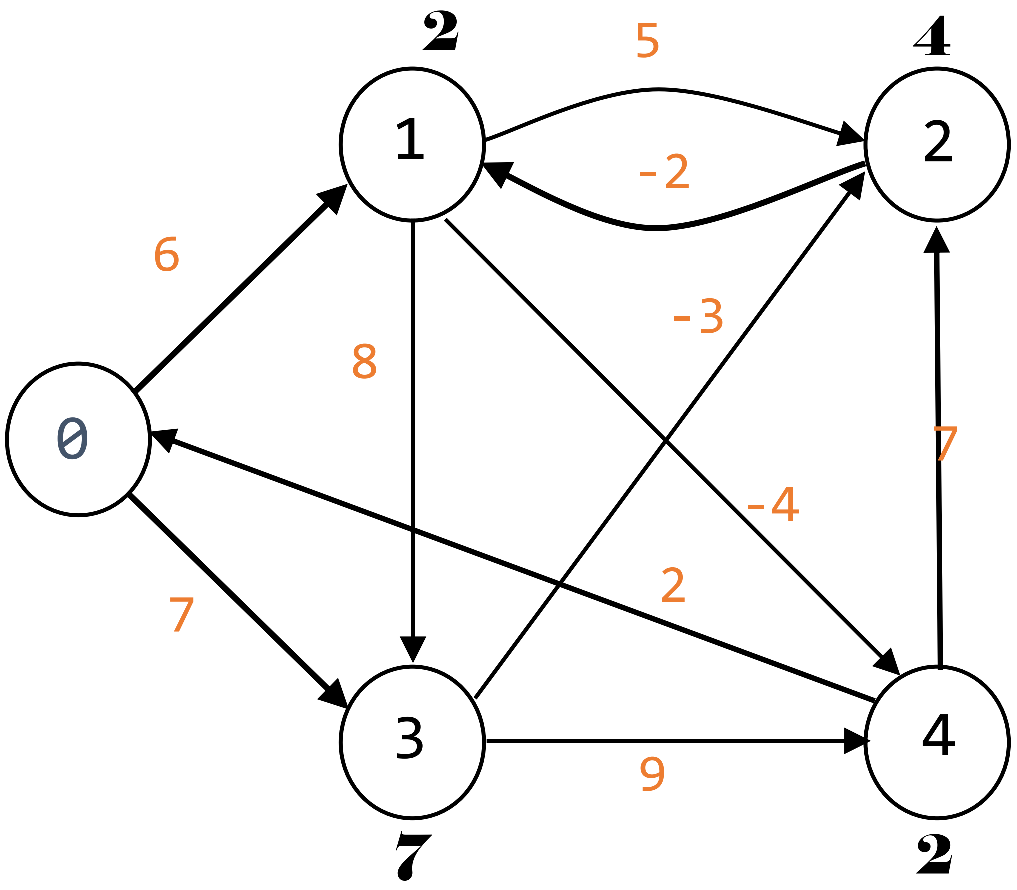
dist[2]= 4 (0,3,2)
  min{4, 6+5, 7+(-3), 2+7}

dist[3]= 7 (0,3)
  min{7, 2+8}

dist[4]= -2 (0,3,2,1,4)
  min{2, 2+(-4), 7+9}

Path length = 4

for each <i,u>,
min{dist[u], dist[i] + length[i][u]}



dist[1]= 2 (0,3,2,1)
min{2, 4+(-2)}

dist[2]= 4 (0,3,2)
min{4, 2+5, 7+(-3), 2+7}

dist[3]= 7 (0,3)
min{7, 2+8}

dist[4]= -2 (0,3,2,1,4)
min{-2, 2+(-4), 7+9}

Next Topic

Sorting