Chap 6. Graph

1. 개요

●목차

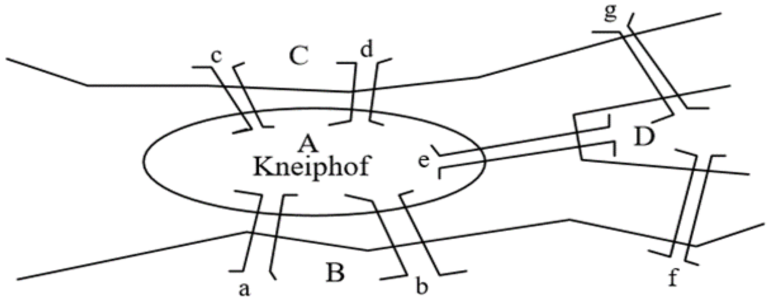
1)Graph 정의 및 표현
- Definitions
- Representation: Adjacency (Matrix, List)
  (인접행렬, 인접리스트)

2)그래프 기본 연산 (Elementary Graph Operation)
- Depth First search (깊이우선 탐색),
- Breadth First search(넓이 우선 탐색)

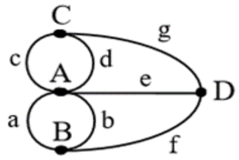
3)Minimum cost Spanning Tree (MST)
- Kruskal, Prim, Sollin

4)Shortest Path (single source all destination)

●그래프의 개요
- Koenigsberg 다리 문제
- 차수(degree) : 정점에 연결된 간선의 수



Koenigsberg 의 다리



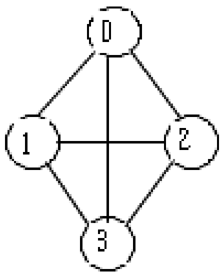
Euler 의 그래프

1.1 정의

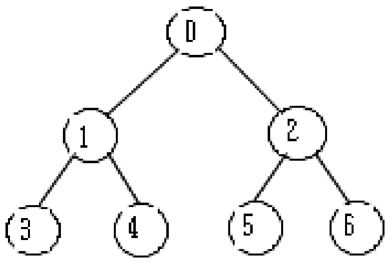
A **graph**, G, consists of two sets, a finite set of **vertices(**정점) and a finite set of **edges(**간선).

. **G = (V, E)**   **V: set of vertex**   **E: Set of edges**

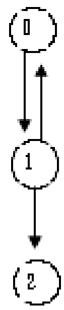
. Undirected Graph (무방향): (v1, v2) = (v2, v1)   G1, G2
. Directed Graph (방향) : <v1, v2> ≠ <v2, v1>   G3



G1



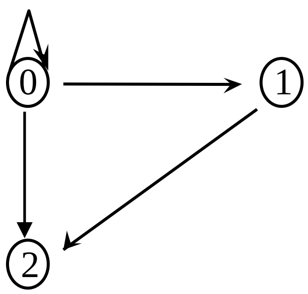
G2



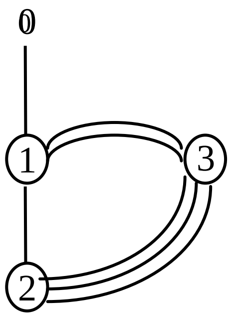
G3

V(G1) = {0,1,2,3}   E(G1) = {(0,1)(0,2)(0,3)(1,2)(1,3)(2,3)}
V(G2) = {0,1,2,3,4,5,6}   E(G2)={(0,1)(0,2)(1,3)(1,4)(2,5)(2,6)}
V(G3) ={0,1,2,}   E(G3) = {<0,1><1,0><1,2>}

- Restriction on a graph: **no self-loop,** **and** **no Multigraph**
  - Self Loop: 자기자신을 가리키는 간선
  - Multigraph(다중그래프): 두정점 사이에 여러 간선 있는 graph



자기 간선 가진 그래프



다중 그래프

■ Complete Graph 란?

the maximum number of edges 를 갖는 그래프 (모든 정점에서 다른 정점으로 간선이 존재하는 그래프)

(ex. G1 is a complete Graph,   G2, G3 is not complete graph)

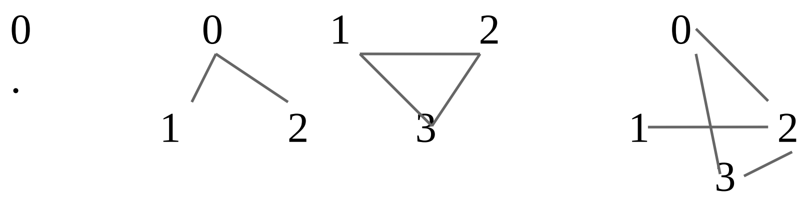
⇨ For undirected graph max number of edges **=> n(n-1)/2**
⇨ For directed graph, max number of edges **=> n(n-1)**

● If (0,1) is an **<u>edge</u>** in undirected graph, then **vertices 0** and **1** are **<u>ADJACENT(인접) 한다.</u>**
and the edge (0,1) is **<u>INCIDENT(부속)</u>** on vertices 0 and 1

ex) In Graph G2, vertices 3,4,0 are adjacent to vertex 1 and
edges (0,1), (1,3), (1,4) are incident on vertex 1

■ **Subgraph**: A subgraph of graph G is G', such that
$V(G') \subseteq V(G)$ & $E(G') \subseteq E(G)$

ex) in Graph G1, many **subgraphs**, such as



■ Cycle and Path

Cycle   path

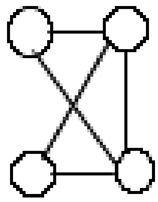
- Path(경로): 정점(간선)들의 연속
. 경로의 길이(length): 경로상의 간선 수
. Simple PATH(단순경로): 서로 다른 정점으로 구성된 경로

- CYCLE: (처음과 끝 정점이 같은 단순경로)

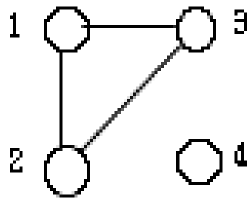
Cycle is a simple path in which the FIRST and LAST vertices are the same vertex

- CONNECTED: 연결(*connected*) 그래프, G

  vi, *vj* ∈ V(G) ⇒ *vi* 에서 *vj* 로의 경로 존재



Connected



Disconnected (G4)

- Connected **Component**: number of subgraphs

  ex) 위 Graph G4 has two components

  * <u>**Diff with Tree and Graph**</u>

  1) Tree is special case of graph
  2) Tree is a graph that is **connected**
  3) Tree is a graph that has **no cycle**

- **DEGREE**: number of edges incident to that vertex

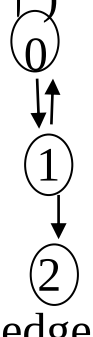
  . Undirected Graph:

  . Directed Graph: (indegree 내차수, outdegree 외차수)

  In G3,

  vertex 1: indegree 1

  outdegree 2 ➔ degree 3.



If d_i is degree of vertex ***i*** in G, with **n** vertices and ***e*** edges:

⇨ number of edges: for undirected graph $e = \left(\sum_{0}^{n-1} d_i \right) / 2$

<u>1.2 Graph Representation (3 가지 표현방법)</u>

1)인접행렬 (Adjacent matrix)

2)인접리스트 (Adjacent list)

3)인접다중리스트 (Adjacent multilist)

1) Adjacency Matrix(인접 행렬/매트릭스)

G1

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

. Undirected Graph: **Symmetric**
. Can save space by storing only upper/lower triangle of matrix
.Degree of any vertex=low sum(행의 합)

G3

	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

. Directed Graph: **Not symmetric**
. Degree of vertex: row sum -> outdegree
col sum -> indegree

- Sparse graphs 란? : 간선의 개수가 적은 그래프를 뜻함
- sparse graph 를 adjacency matrix 로 표현하면 memory waste 임. <u>adjacency list 가 적합함.</u>

2) Adjacent List (인접 리스트)

Replace **n** rows of adjacency matrix with **n** linked list, one for each vertex in G (<u>각 정점에 대해 **1** 개의 리스트 존재</u>.
n 행들을 n 개의 연결리스트로 표현)

headnodes vertex link

For G1:

Vertex0 → [1 |] → [2 |] → [3 |]

Vertex1 → [0 |] → [2 |] → [3 |]

Vertex2 → [0 |] → [1 |] → [3 |]

Vertex3 → [0 |] → [1 |] → [2 |]

For G3:

Vertex0 → [1 | NULL]

Vertex1 → [0 |] → [2 |]

Vertex2 [NULL]

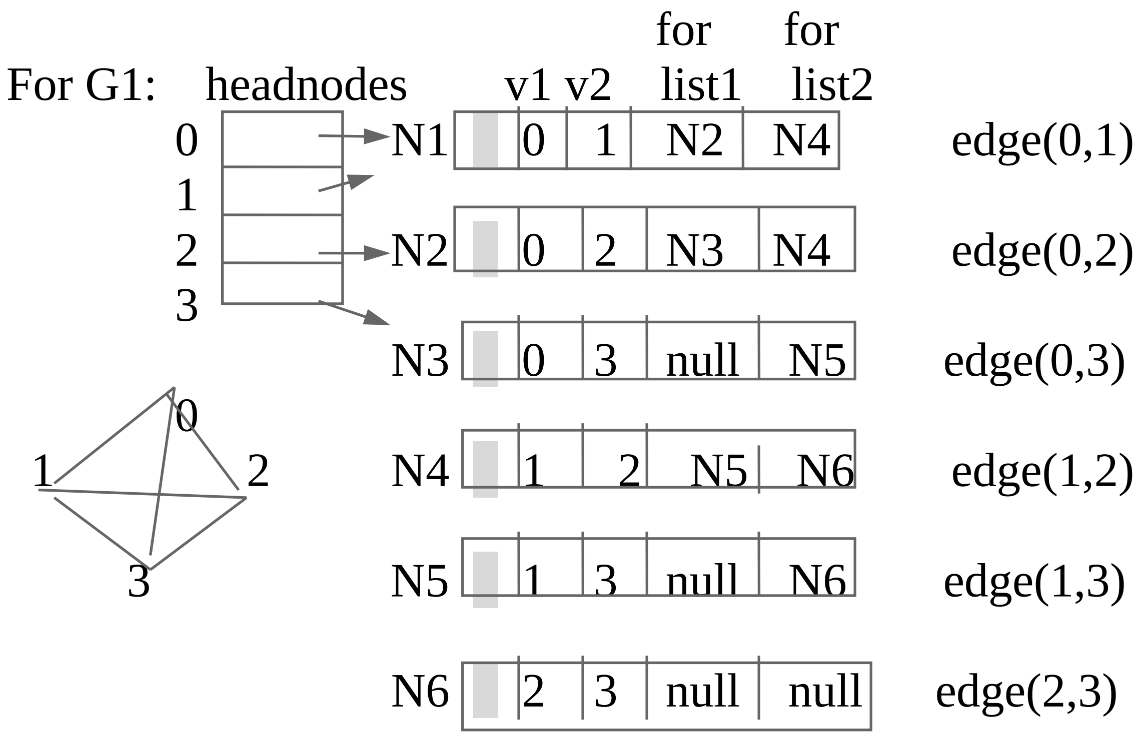
3) Adjacent Multilist (인접 다중리스트)

인접리스트에서는 각 간선이 두 번 표현되었음 (예를 들어 (1,0)의 경우 정점 1 에서 한번, 정점 0 에서 한번)
=> 그러나 어떤 경우에는 간선의 두 번째 정점,
(예 (0,1)의 경우 1) 을 mark 하여 이미 조사 하였음을 알 필요가 있을때 → Multilist 로 해결

– 새로운 노드 구조 (Multilist):

* Node 구조

marked	vertex1	vertex2	path1	path2



The lists are: vertex0: N1->N2->N3
vertex1: N1->N4->N5
vertex2: N2->N4->N6
vertex3: N3->N5->N6

● 가중치 간선 (weighted Edges)

\- 그래프의 간선에 가중치(weight)부여.

2. Elementary Graph Operations(그래프 기본연산)

1) DFS (Depth First Search): 깊이 우선탐색

. *visited*[*MAX_VERTICES*] : 배열 (초기치 = FALSE)

. *visited*[*i*] = TRUE : 정점 *i* 방문

2) 설계

```
class Node {
   private:
         int data;
         Node *link;
friend class Tree;
};
```

```
class Tree {
     private:   Node * root;
     public:
           Tree();
           void dfs(int v);
           void print-node(); };
```

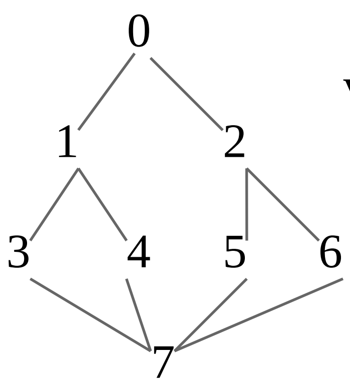
```
void DFS(int v)
{
      Node    *w;
      visited[v] = true;
      cout << v;
      for (w= graph[v]; w!=NULL; w=w->link)
           if (!visited[w->data])    DFS(w->data);
   } }
```

. 시작정점 v 방문 (visited[v]=true)

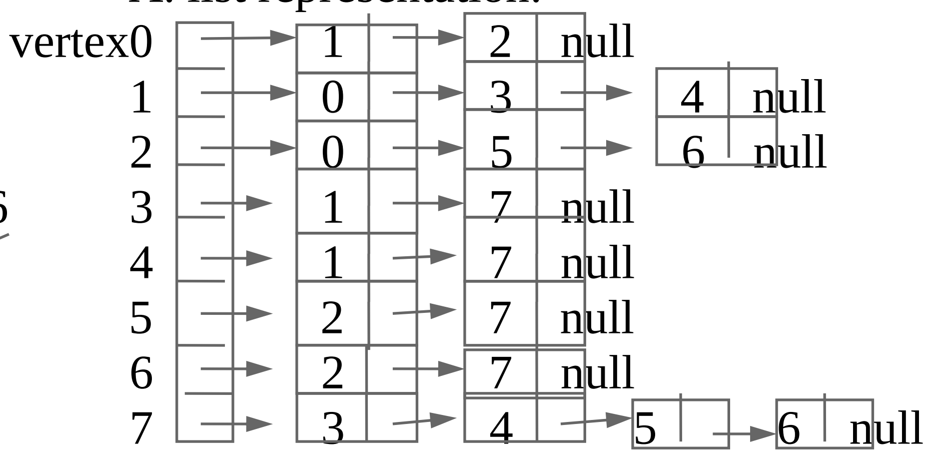
. For each vertex w adjacent to v do

if not visited[w] then DFS(w);

. 더이상 없으면 dfs 끝



A. list representation:



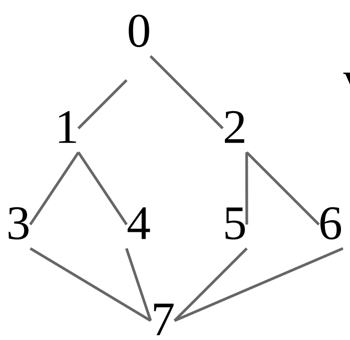
Start from V_0,: 0, 1, 3,7,4,5,2,6

	0	1	2	3	4	5	6	7
visited	T	T		T	T	T		T

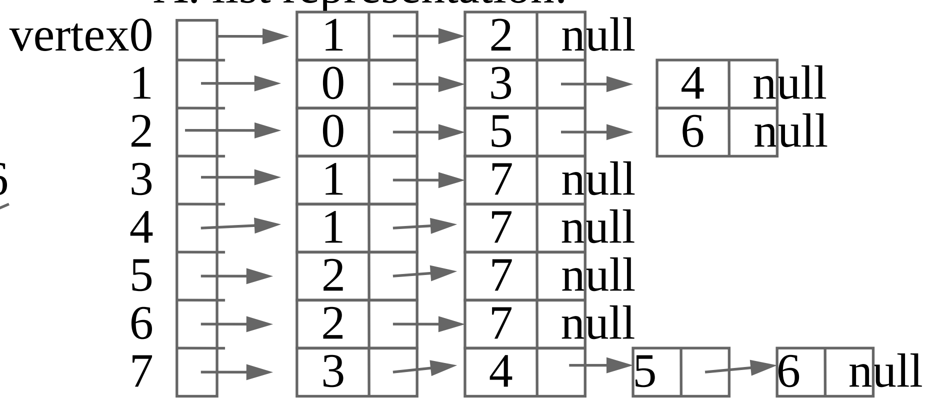
```
int ADM[size][size] ={
        0, 1, 1, 0, 0, 0, 0,
        1, 0, 0, 1, 1, 0, 0,
        1, 0, 0, 0, 0, 0, 1,
        0, 1, 0, 0, 0, 1, 0,
        0, 1, 0, 0, 0, 1, 0,
        0, 0, 0, 1, 1, 0, 1,
        0, 0, 1, 0, 0, 1, 0 };
```

2) BFS (Breadth First Search) . Implement with Linked Queue

```
void Graph::BFS(int v)    { //정점 v에서 시작, 넓이우선 탐색수행
 Node p;                           // 정점 방문시 visited[i]는 true 됨
 visited[v] = true;                    // Queue 사용
 addq(Q, v);        cout << v;

 while (not empty (Q)) {
    v = dequeue(Q);
    for (p=graph[v]; p; p=p->next); //인접된 모든 노드에 대해
       if (!visited[p->vertex]) {          //if not visited
              enqueue(p->vertex);
              visited[p] = true;
              cout << p->vertex;
       }}}}
```



A. list representation:



Start from V_0,: 0, 1, 2,3,4,5,6,7

	0	1	2	3	4	5	6	7
visited	T	T	T	T	T			T

3. SPANNING TREES (신장트리)

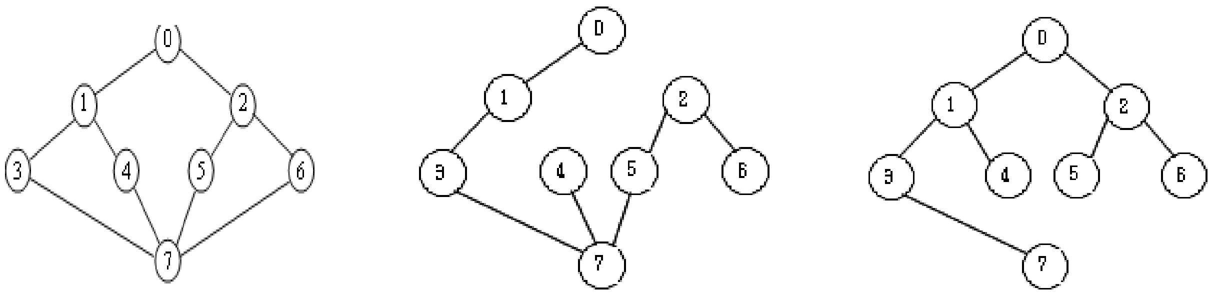
*용도 : 통신망, 교통망, 등 다양한 분야에 적용할 수 있음

- 신장트리의 제한조건:
  - 그래프내의 간선만 사용 (n 개의 정점. n-1 개의 간선)
  - Cycle 형성하는 간선 사용금지
  - 신장트리는 연결되어(connected) 있음.

Definition: any **tree** that includes **<u>all the vertices</u>** in G
(신장트리는 G 의 최소부분 그래프(minimal subgraph) 이다.

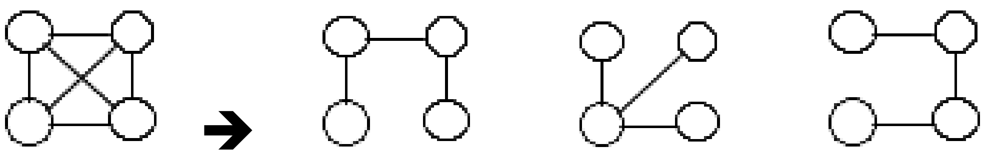
$$G = (V, E), \qquad V(G') = V(G) \quad \& \quad \mathbf{E(G') \subseteq E(G)}$$

- We can use DFS or BFS to create a spanning tree
  - when DFS is used => the result is DFS spanning tree
  - when BFS is used => the result is BFS spanning tree



(original graph) (DFS Spanning Tree) (BFS Spanning Tree)

- 하나의 연결그래프(Connected Graph)는 출발점이나 검색방법에 따라 각기 다른 신장 트리가 만들어진다.



3. Minimum cost spanning trees(MST)최소비용 신장트리

* cost가 제일 적은 신장트리(spanning tree)
* 대표적 MST 알고리즘: **kruskal, Prim, Sollin**
* 세 알고리즘 모두 다음 Greedy method(갈망법)사용
  - 최적의 해를 단계별로 구한다.
  - 각 단계에서, 판단기준에 따라 최상의 결정을 한다
    (신장트리에서는 최저비용을 기준으로)
  - 한번 결정된 해는 번복 불가.

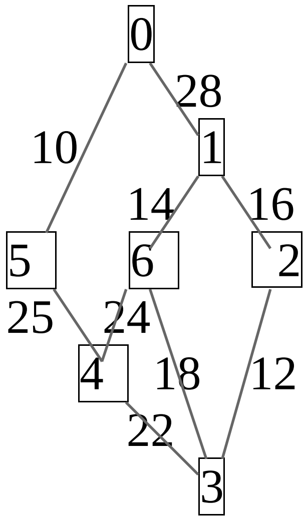
1) Kruskal’s Algorithm (Greedy method)

⇨ 그래프 G를 무방향 그래프라고 하면, Kruskal 알고리즘은 최소비용 신장트리를 생성한다.

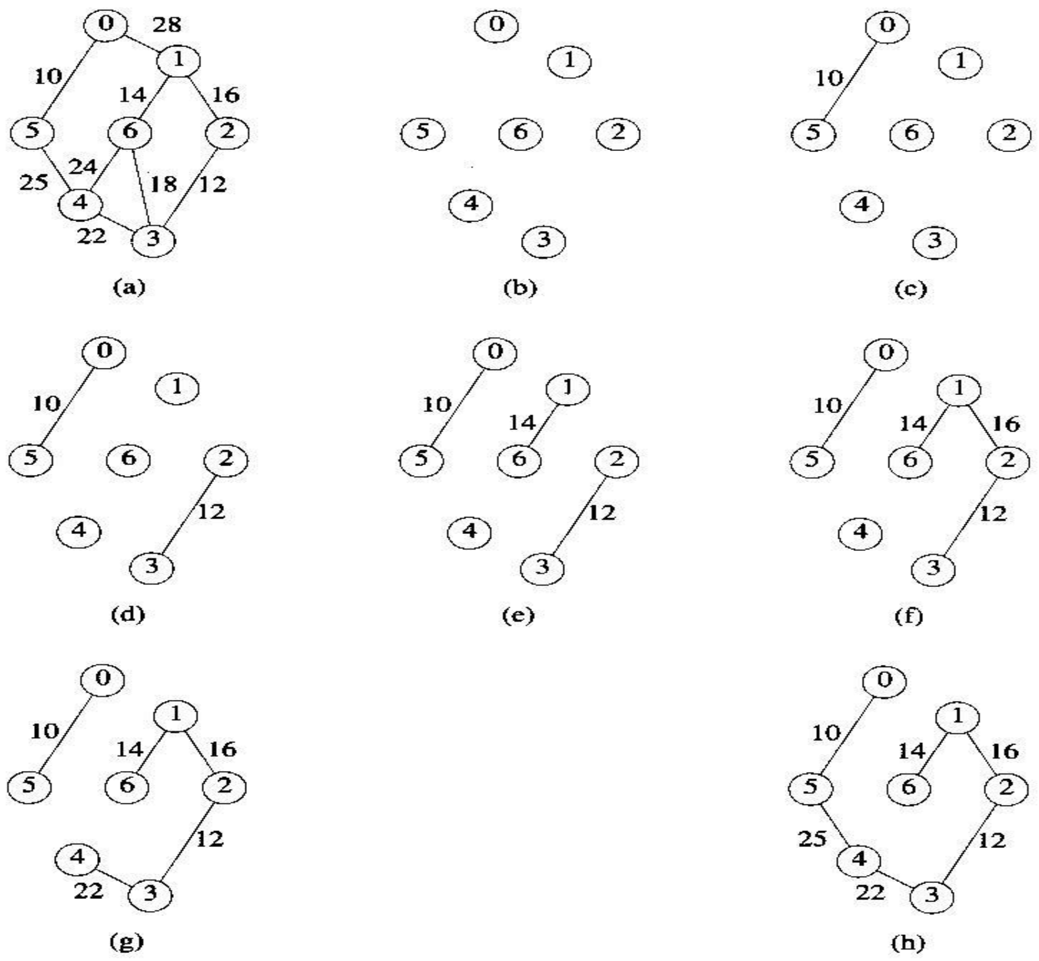
● 알고리즘:
- 한번에 한 간선씩 최소 비용 신장 트리를 구축
- MST T에 포함될 간선을 비용의 **크기 순**으로 선택
- T에 **n-1개의 간선이** 포함될 때까지 간선 추가

```
T = { };
while (T contains <n-1 edges) & (E is not empty) {
    choose a least cost edge (v,w) from E;   //최소비용 간선선택
    delete (v,w) from E;                     //간선집합 E에서 삭제
    if ((v,w) does not create a CYCLE in T   // T에서 사이클
            add(v,w) to T     => ACCEPT      // 만들지 않으면
    else    discard(v,w);     => REJECT
}
```

If T contains fewer than n-1 edges, then “ “No spanning tree”;



cost	edge	action
10	(0,5)	accept
12	(2,3)	accept
14	(1,6)	accept
16	(1,2)	accept
18	(3,6)	reject => cycle
22	(3,4)	accept
24	(4,6)	reject => cycle
25	(4,5)	accept
28	stop	already (n-1) edges added



* Prim’s algorithm form **a tree** at each stage, but Kruskal’s form a **forest**

2) Prim's Algorithm

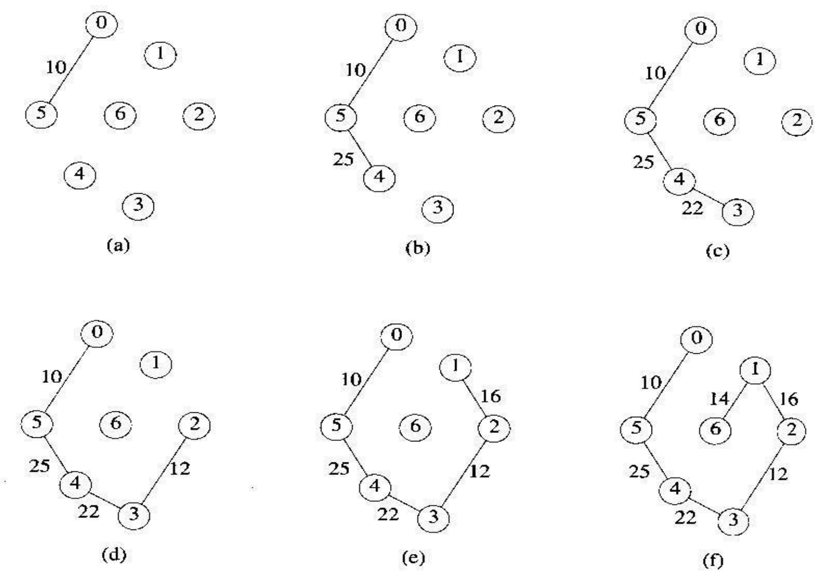
알고리즘:

- 한번에 한 간선씩 최소 비용 신장 트리를 구축
- 하나의 정점으로 된 트리 T 에서 시작
- 각 단계에서 선택된 간선의 집합은 트리
- T 에 n-1 개의 간선이 포함될 때까지 간선 추가
- 사이클 형성 않도록, 각 단계에서 간선 (U,V)선택시
  U 또는 V 중 하나만 T 에 속한 것 선택.

Algorithm 1: {start vertex **v**}

```
T = { };            //Prime MST      TV={0} //start with vertex 0//
 for (i=1 to max)                          //COST= adjacency matrix
      lowcost[i]= COST[v][i];        //v= starting vertex

 while   (T contains fewer than n-1 edges)   {
     Find (u,v) be a least cost edge such that   u∈TV & v∉TV;
     if   (there is no such edge)     break;
          else   add   v to TV;   add   (u,v) to T;
     let u to be a new v, and continue
   }
  if   (T contains fewer than n-1 edges) print "No spanning Tree";
```

ex) starting vertex '0'



3) Sollin's Algorithm

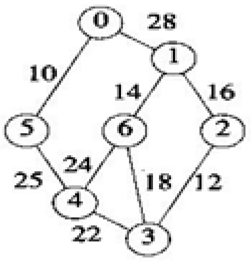
각 단계별로, T에 포함될 간선을 여러개 선택

(i) 그래프의 모든 *n* 정점을 포함하는 신장 트리 구성

(ii) forest 내의 각 트리에 대해 하나의 간선 선택,

(최소비용선택)

Ex)



0 1 5 6 2	=>	0 : (0,5) 1 : (1,6) 2: (2,3) 3: (3,2) 4: (4,3) 5: (5,0) 6: (6,1)	=>	0 1 5 6 2 4 3 forest	=>	(0,5) (1,6) (2,3) (4,3)
4 3						

Step1: List vertex Step2: select EDGE for each vertex

Step3: Select next edges for each trees

. Tree{0,5}: => (1,0),(4,5)중에서 (4,5)선택, cost is 25 이 최소

. Tree{1,6}: => (1,2), (6,3)중에서 (1,2)선택, cost is 16 이 최소

. Tree{2,3,4}:=> (2,1),(3,6)(4,6) 중에서 (2,1)선택, cost is 16 이 최소

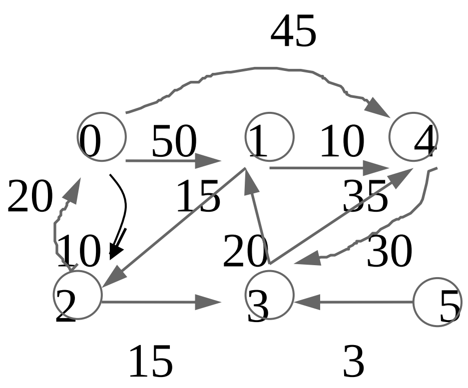
⇨ 결과

0
5 1
6 2
4 3

4. Shortest Path (최단경로)

1) Single Source All Destination (단일 출발점-> 모든 도착지)

- *v*0(*source*)에서 G 의 다른 모든 정점(도착지)까지의 최단경로



path	length
1) v0 v2	10
2) v0 v2 v3	25
3) v0 v2 v3 v1	45
4) v0 v4	45

< shortest path from v0 to v1,v2,v3,v4>

<no path from v0 to v5>

* *found*[*i*] : if *found*[*i*]=TRUE *v*i 까지의 최단경로 발견

distance[*i*] : *v*0 에서 S 내의 정점만을 거친 *v*i 까지의 최단거리 (S=최단경로가 발견된 정점의 집합)

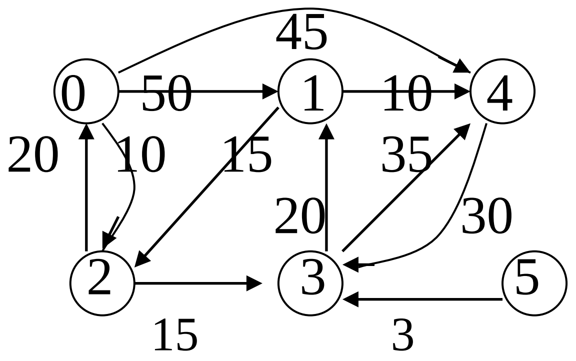
- 초기치 : *distance*[*i*] = *cost*[0][*i*]
- *cost*[*i*][*j*] : <*i*, *j*>의 가중치

* 그래프 : 비용 인접 행렬(*cost adjacency matrix*)로 표현

```
void initCostMatrix(int cost[][])    { //initialize
int I, j;
for (i = 0; i < 8; i++)
   for (j = 0; j < 8; j++)
          if (i == j)    cost[i][j] = 0;
          else           cost[i][j] = 999;
}
// Enter distance value from data file.
```

- Algo (Shortest path)   by Dijkstra's algorithm

```
Void Shortestpath (int v, int cost[][], int dist[], int n, bool found[]) {
        int i,u,w;
        for (i=0; i<n; i++)    {                          . O(n)
              found[i] = false;                           . found all FALSE
              distance[i] = cost[v,i]; }                  . initial value assign

        found[v]=true;                     // start vertex mark
        distance[v]=0;                     // start vertex 0

        for (i=0; i<n-1; i++)    {
            u = choose(distance, n, found);             // find min value node
            found[u]= true;                              // mark that node

            for (w =0; w<n; w++) {          // and replace if revised value
              if (!found[w]) {                 // if not marked
                if (distance[u]+cost[u,w] <distance[w])    //is smaller than org
                    distance[w] = distance[u] + cost[u,w];      // value
            }}
          Print " Distance ->", dist[ ];    print 1 row for the vertex
} }

int choose( )
{      int i, min; // min=max_value 로  초기화

      for (i = 0; i < n; i++)
              if (dist[i] < min && !found[i]) {
                      min = distance[i]
              }
      return i;
}
```



	0	1	2	3	4	5
0		50	10		45	
1			15		10	
2	20			15		
3		20			35	
4				30		
5				3		

비용 인접 행렬 - *cost*

Vertex	0	1	2	3	4	5
Distance	0	50	10	999	45	999
S	1	0	0	0	0	0

1. S = {*v*0} : 초기는 공백

distance(1) = 50
distance(2) = 10 <= *min*
distance(3) = 999
distance(4) = 45
distance(5) = 999

Vertex	0	1	2	3	4	5
distance	0	50	10	999	45	999
S	1	0	1	0	0	0

2. S = S ∪ {*v*2} = {*v*0, *v*2}

distance(1)<- min{distance(1), distance(2)+(*v*2,*v*1,999)} 50
distance(3)<-min{distance(3), distance(2)+(*v*2,*v*3,15)} 25 <= *min*
distance(4)<- min{distance(4), distance(2)+(*v*2,*v*4,999)} 45
distance(5)<- min{distance(5), distance(2)+(*v*2,*v*5,999)} 999

vertex	0	1	2	3	4	5
distance	0	50	10	25	45	999
S	1	0	1	1	0	0

3. S = S ∪ {v3} = {v0, v2, v3}

distance(1)<- min{distance(1), distance(3)+*v*3,*v*1,20}}   45 <= *min*

distance(4)<- min{distance(4), distance(3)+(*v*3,*v*4,35)}   45

distance(5)<- min{distance(5), distance(3)+(*v*3,*v*5,999)}   999

Vertex	0	1	2	3	4	5
Distance	0	45	10	25	45	999
S	1	1	1	1	0	0

4. S = S ∪ {*v*1} = {*v*0, *v*1, *v*2, *v*3}

distance(4)<-min{distance(4), distance(1)+(*v*1,*v*4,10)}   45<=   *min*

distance(5)<- min{distance(5), distance(1)+(*v*1,*v*5,999)}   999

Vertex	0	1	2	3	4	5
distance	0	45	10	25	45	999
S	1	1	1	1	1	0

5. S = S ∪ {*v*4}

distance(5) <-min{distance(5), distance(4)+(*v*4,*v*5,999)}   999 <=   *min*

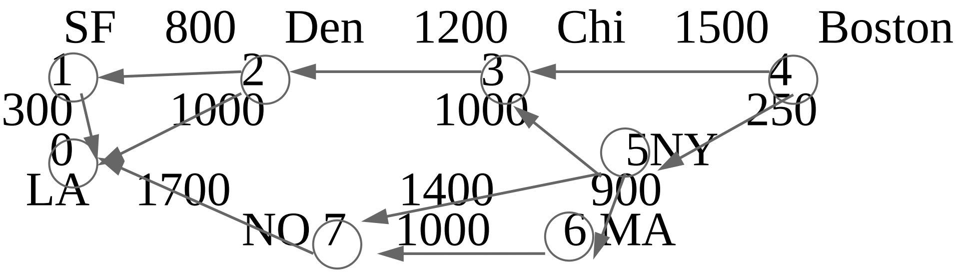
Vertex	0	1	2	3	4	5
Distance	0	45	10	25	45	999
S	1	1	1	1	1	1

6. S = S ∪ {*v*5}

Final Solution:

Distance	0	45	10	25	45	999

Ex2)



Matrix (cost[v,I])	0	1	2	3	4	5	6	7
0	0	∞	∞	∞	∞	∞	∞	∞
1	300	0						
2	1000	800	0					
3			1200	0				
4				**1500**	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

- Found:

F	F	F	F	F	F	F	F

false initially

- Distance:

0	0	0	1500	0	250	∞	∞

1st iteration

Iteration	visited	vertex selected	Distance LA [0]	SF [1]	DEN [2]	CHI [3]	BOS [4]	NY [5]	MA [6]	NO [7]
Initial	-	-	∞	∞	∞	**1500**	**0**	**250**	∞	∞
1	**4**	**5**	∞	∞	∞	**1250**	**0**	**250**	**1150**	**1650**
2	**4,5**	**6**	∞	∞	∞	**1250**	**0**	**250**	**1150**	**1650**
3	**4,5,6**	**3**	∞	∞	**2450**	**1250**	**0**	**250**	**1150**	**1650**
4	**4,5,6,3**	**7**	**3350**	∞	**2450**	**1250**	**0**	**250**	**1150**	**1650**
5	**4,5,6,3,7**	**2**	**3350**	**3250**	**2450**	**1250**	**0**	**250**	**1150**	**1650**
6	**4,5,6,3,7,2**	**1**	**3350**	**3250**	**2450**	**1250**	**0**	**250**	**1150**	**1650**
	{4,5,6,3,7,2,1}									

* 250 + 1000 < 1500

if (distance[u] + cost[u,w] <distance[w]) // is smaller than org

distance[w] = distance[u] + cost[u,w]; // value

1250 250 + 1000

therefore CHI has been changed to 1250 thereafter

but this do not change