1. 개요

●목차

1)트리 정의 및 표현
- Definitions
- Representation: Adjacency (Matrix, List)

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

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

4)Shortest Path (single source all destination)

1.1 정의

A graph, G, consists of two sets, a finite set of vertices and a finite set of edges.

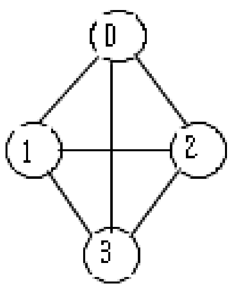
. **G = (V, E)** **V: set of vertex** **E: Set of edges**

. Undirected Graph (무방향): G1, G2

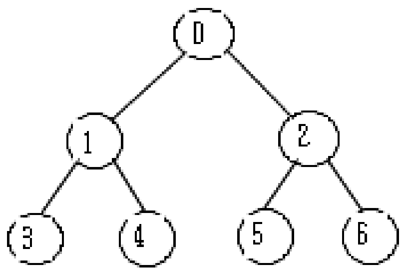
(v1, v2) = (v2, v1)

. Directed Graph (방향) : G3

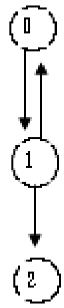
<v1, v2> ≠ <v2, v1>



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: **<u>no self-loop,</u>** **and** **<u>no Multigraph</u>**
  - self-loop: 자기 자신을 가리키는 간선
  - multigraph: 두 정점 사이에 여러 간선 있는 graph

- Complete Graph 란?

  the maximum number of edges 를 갖는 그래프

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

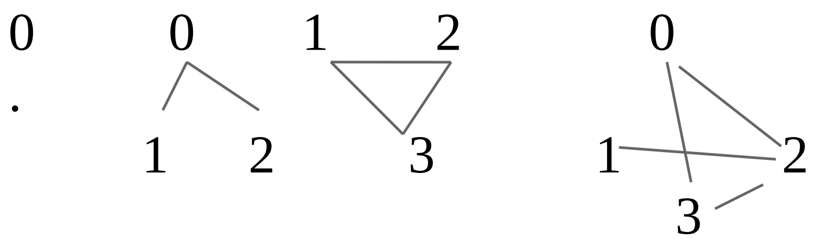
- ⇨ For undirected graph max number of edges **=> n(n-1)/2**
- ⇨ For directed graph, max number of edges **=> n(n-1)**

- If (0,1) is an edge in undirected graph, then vertices 0 and 1 are **ADJACENT**, and the edge (0,1) is **INCIDENT** on vertices 0 and 1

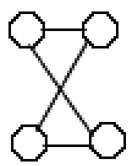
ex) In Graph G2, vertices 3,4,0 are adjacent to vertex 1 and edges (0,1), (1,3), (1,4) are incident on vertex 1

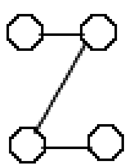
- Subgraph: A subgraph of graph G is G', such that $V(G') \subseteq V(G)$ & $E(G') \subseteq E(G)$

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



- Cycle and Path


Cycle


path

- Path(경로): 정점(간선)들의 연속

. 경로의 길이: 경로상의 간선 수

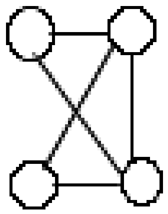
. Simple PATH(단순경로): 서로 다른 정점으로 구성된 경로

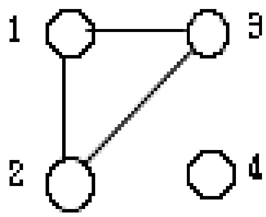
− CYCLE: (처음과 끝 정점이 같은 단순경로)

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

- CONNECTED: 연결(*connected*) 그래프, G

  $vi, vj \in V(G) \Rightarrow vi$ 에서 vj 로의 경로 존재





Connected Disconnected (G4)

- Connected Component: number of subgraphs

  ex) 위 Graph G4 has two components

  * Diff with Tree and Graph

  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

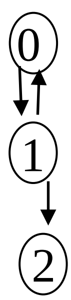
  . Undirected Graph:

  . Directed Graph: (indegree, outdegree)

  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 :

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

1.2 Graph Representation (3 가지 표현방법)

1)인접행렬 (Adjacent matrix)
2)인접리스트 (Adjacent list)
3)인접다중리스트 (Adjacent multilist)

1) Adjacency Matrix

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

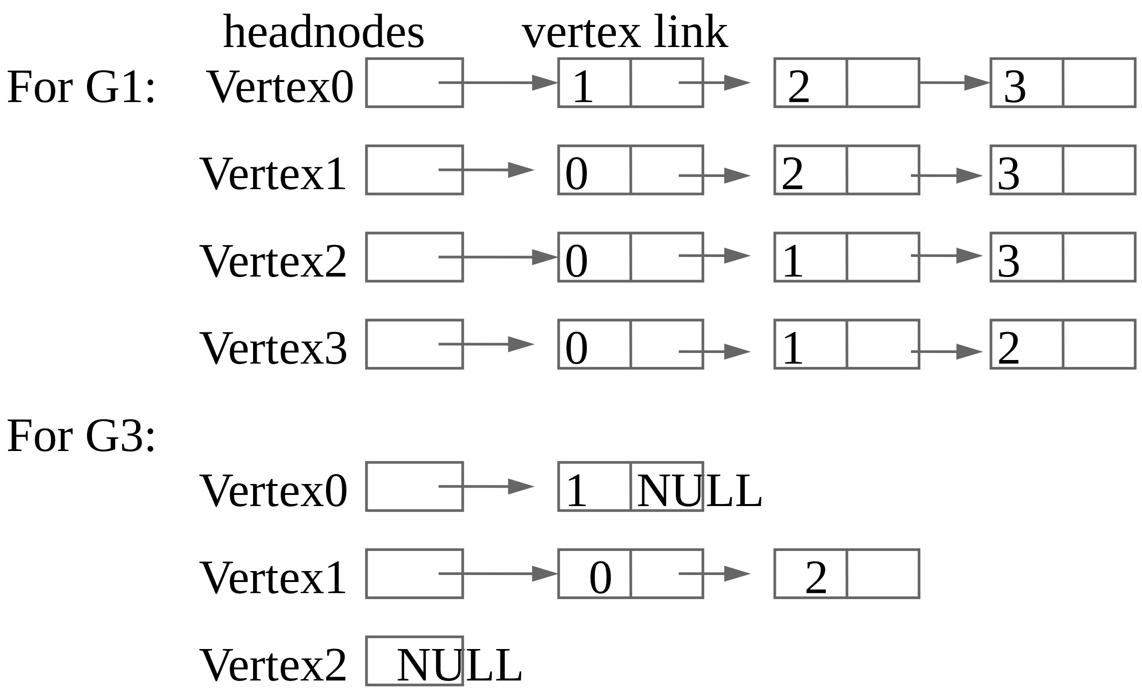
	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

. 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

Replace n rows of adjacency matrix with n linked list, one for each vertex in G (각 정점에 대해 **1** 개의 리스트 존재)



2. Elementary Graph Operations

1) DFS (Depth First Search): 깊이 우선탐색

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

. *visited*[*i*] = TRUE : 정점 *i* 방문

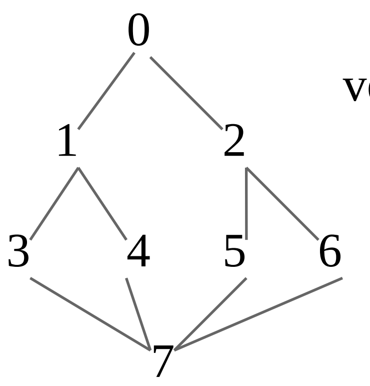
```
procedure DFS(int v)
 {
       Node    *w;
       visited[v] = true;
       cout << v;
        for (w= graph[v]; w!=NULL; w=w->link)
           if (!visited[w->vertex])    DFS(w->vertex);
      } }
```

. 시작정점 v 방문 (visited[v]=true)

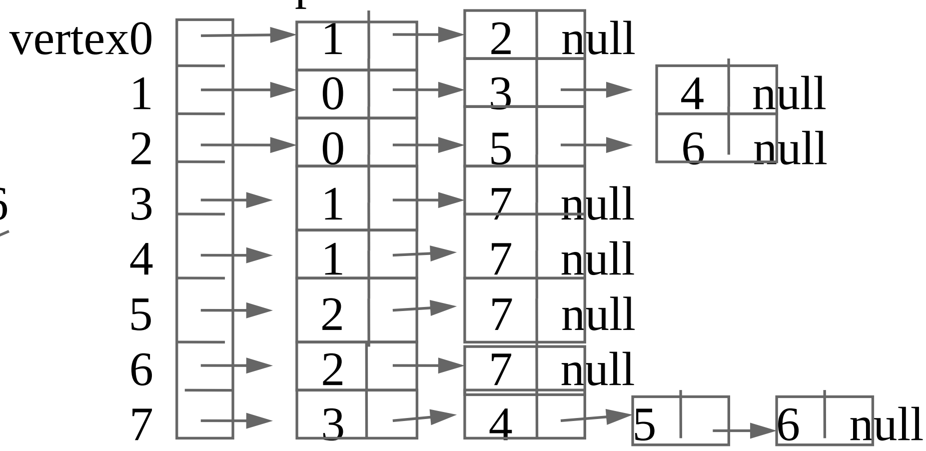
. For each vertex W adjacent to v do

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

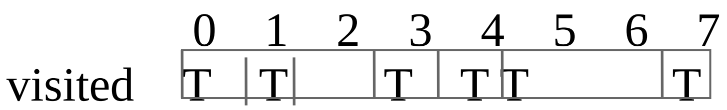
. 더이상 없으면 dfs 끝



A. list representation:



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




```
procedure dfs(…    ){
        visited[d] = true;
         print visited node;

        for (j = 0; j < n; j++)
             if (arr[d][j] == 1 && visited[j] == 0)
                    dfs(..);
}
```

2) BFS (Breadth First Search)

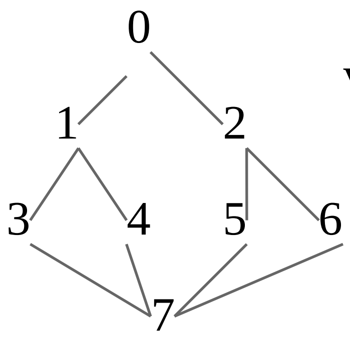
.Implement with Linked Queue

```
. procedure BFS(int v)    {
    Node p;        for all (visited[i]='f' )
    visited[v] = true;
    addq(Q, v);         cout << v;

    while (not empty (Q)) {
       v = dequeue(Q);
```

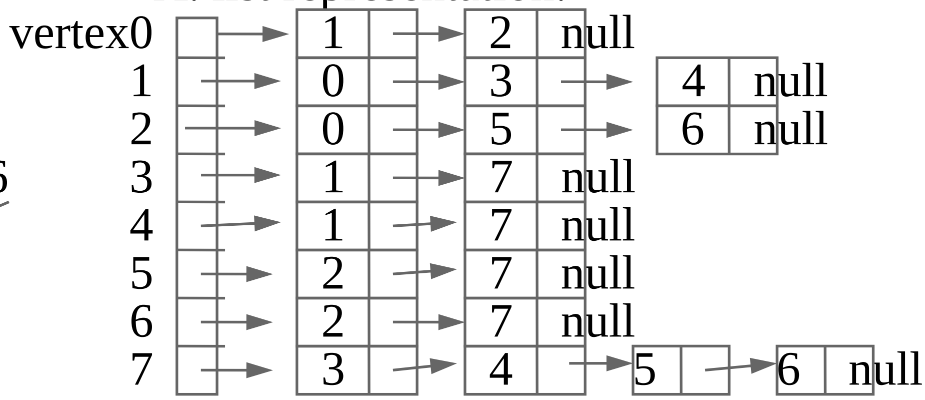
```
        for (p=graph[v]; p; p=p->next);//인접된 모든 노드에대해
            if (!visited[p->vertex]) {          //if not visited
                    enqueue(p->vertex);
                    visited[p] = true;
                    cout << p->vertex;
            }
        }
    }
```

```
int ADJM[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 };
```

```
while (flag) {
    deque(v);
    for (w=graph[v][v]; w<max; w++) {
    if(graph[v][w] !=0) && visited[w]!='t')
      {   addq(w);   visited[w]='t'; cout<<w}
          v++ }
```



A. list representation:



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

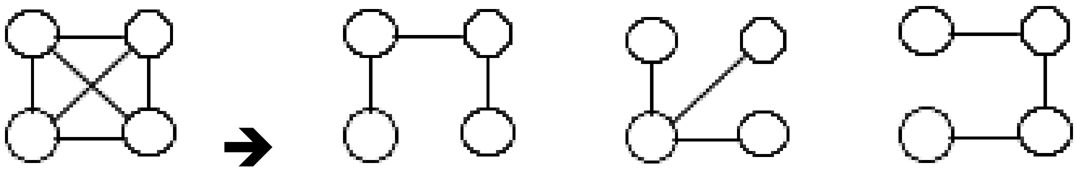
	0	1	2	3	4	5	6	7
visited	T	T	T	T	T			T

3) SPANNING TREES (신장트리)

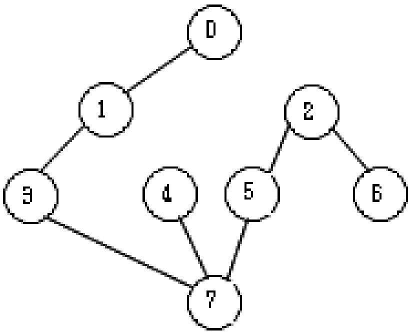
Definition: any tree that includes all the vertices in G

- G=(V,E), G’= (V’, E’)일 때, ST 는 (V⊆V’, E≤E’)
- Spanning Tree -> (n-1) edges (Cycle 없슴)

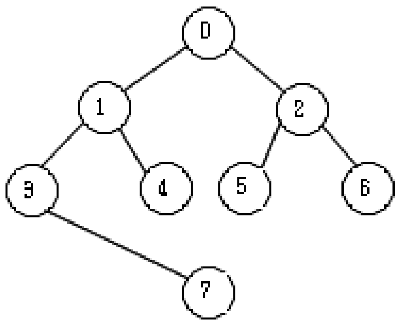
예) 하나의 연결 Graph 는 출발점이나 검색방법에 따라 각기 다른 신장 트리(Spanning Tree)가 만들어진다.



- 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



(DFS Spanning Tree)



(BFS Spanning Tree)

▶ 접근 방법: Greedy Method

- 최적의 해를 단계별로 구한다.
  . 각 단계에서, 판단기준에 따라 최상의 결정을 한다
- 한번 결정된 해는 번복 불가.

▶ cost가 제일 적은 신장트리(spanning tree) => MST

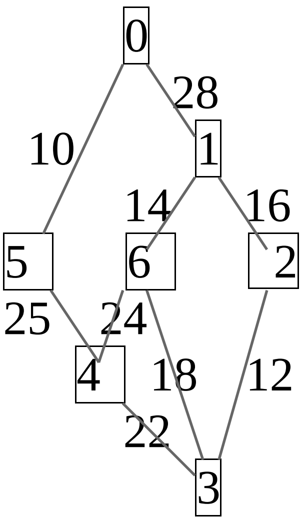
3. Minimum cost spanning trees(MST)

⇨ 대표적인 MST algorithms: Kruskal, Prim, Sollin

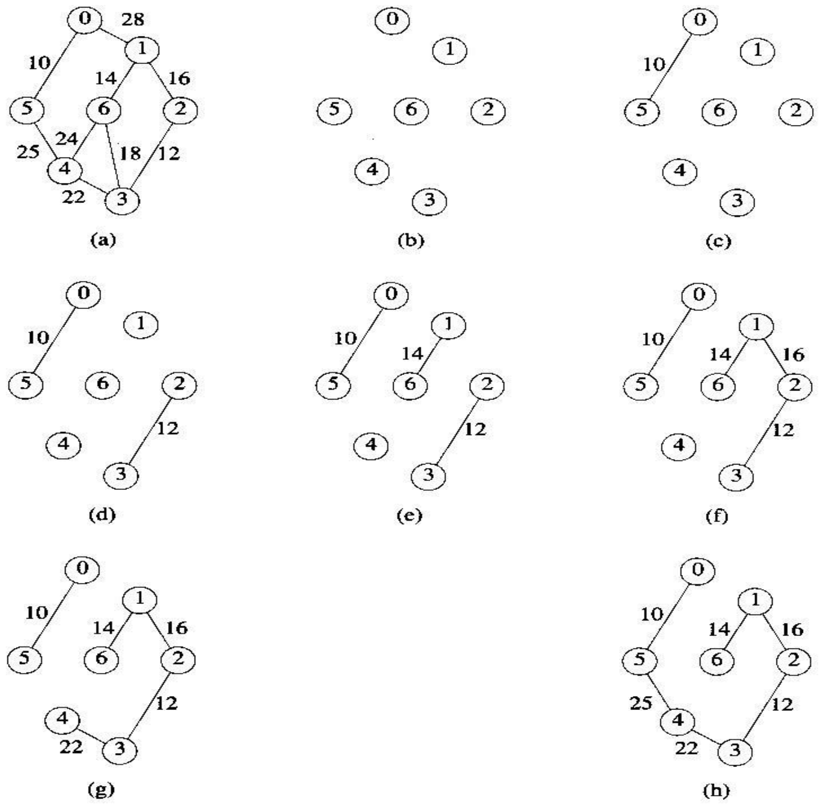
1) Kruskal’s Algorithm (Greedy method)

* 한번에 한 개의 간선을 MST T 에 추가.
* T 에 포함될 간선을 비용크기 순으로 선택 → SORT 필요.
* T 에 n-1 개의 간선이 포함될 때 까지 간선 추가.

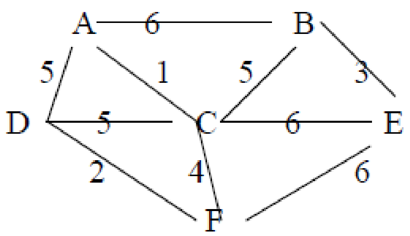
```
 Kruskal() {
    sort(); //
    T = { };    // T = MST
    while (T contains <n-1 edges) & (E is not empty) {
       choose a least cost edge (v,w) from E;
       delete (v,w) from E;
       if (v,w) does not create a CYCLE in T    // check cycle
                add(v,w) to T      => ACCEPT
       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



Ex)
```
Class Node {

      char ff;   int edges;    char ll;

  };
```



A 6 B	B 3 E
B 5 C	C 6 E
A 1 C	C 4 F
A 5 D	D 2 F
C 5 D	E 6 F

```
v[0].ff = 'A';  v[0].edges = 6;  v[0].ll = 'B';
v[1].ff = 'B';  v[1].edges = 5;  v[1].ll = 'C';
v[2].ff = 'A';  v[2].edges = 1;  v[2].ll = 'C';
v[3].ff = 'A';  v[3].edges = 5;  v[3].ll = 'D';
v[4].ff = 'C';  v[4].edges = 5;  v[4].ll = 'D';
…………..
```

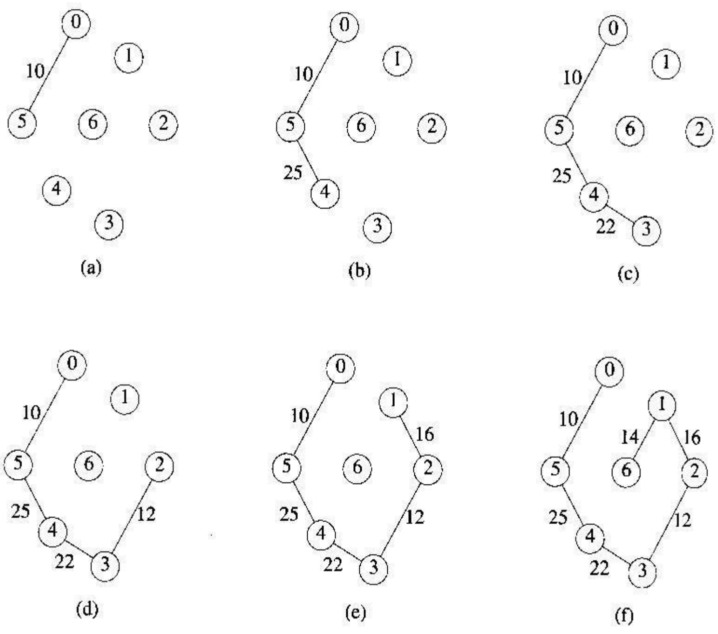
2) Prim's Algorithm

Kruskal's form a forest , but Prim's algorithm form a tree at each stage
- 각 단계에서 선택된 간선의 집합은 트리
- MST T에 n-1 개의 간선이 포함될 때까지 간선 추가
- 사이클 형성 않게, (U,V)중 하나만 T에 속한 것 선택

Algorithm : {start vertex v} // 시작 정점 필요

```
T = { };      //Prime MST, TV={0}->   start with vertex 0 //v
For (i=1 to max)
            lowcost[i]= COST[v][i];    //v= start vertex
while    (T contains fewer than n-1 edges)    {
    Find (u,v) be a least cost edge from lowcost[];
    Print (v, u)            // print least cost edge and mark
    if    (there is no such edge)      break;
    else       add u 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";
  print total value;
```

ex) starting vertex '0'



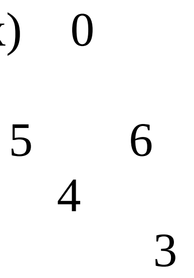
(a) (b) (c) (d) (e) (f)

3) Sollin's Algorithm 각 단계별, T에 포함될 간선을 여러개 선택

(i) 그래프의 모든 *n* 정점을 포함하는 신장트리 구성

(ii) forest 내의 각 트리에 대해 하나의 간선 선택,
(최소비용선택)

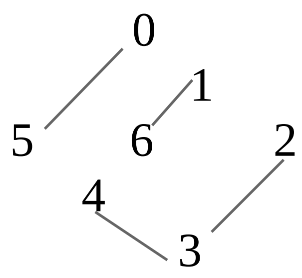
ex)



=>

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

=>



forest

=>

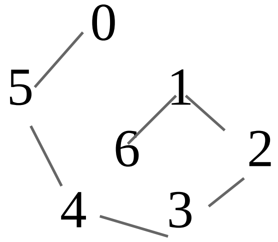
(0,5)
(1,6)
(2,3)
(4,3)

. Tree{0,5}:=> (1,0),(4,5), will select (4,5) since cost is 25 (minimum)

. Tree{1,6}: => (1,2), (6,3), will select (1,2) since cost is 16 (min)

. Tree{2,3,4}:=> (2,1),(3,6)(4,6), will select (2,1) since cost is 16 (min)

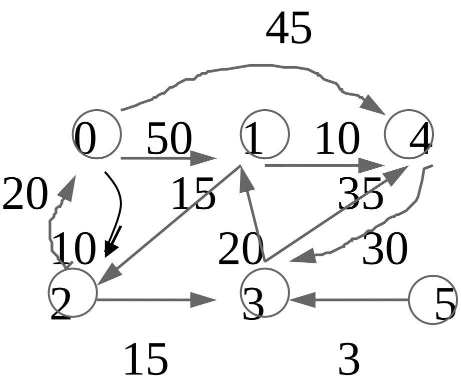
⇨ 결과



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)

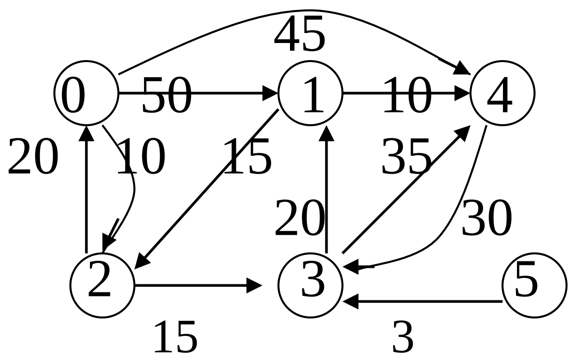
- 초기치 : *distance*[*i*] = *cost*[0][*i*]
- *cost*[*i*][*j*] : <*i*, *j*>의 가중치

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

      for (w =0; w<max; 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 ->", distance[ ];    print final distance
} }

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

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



	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*

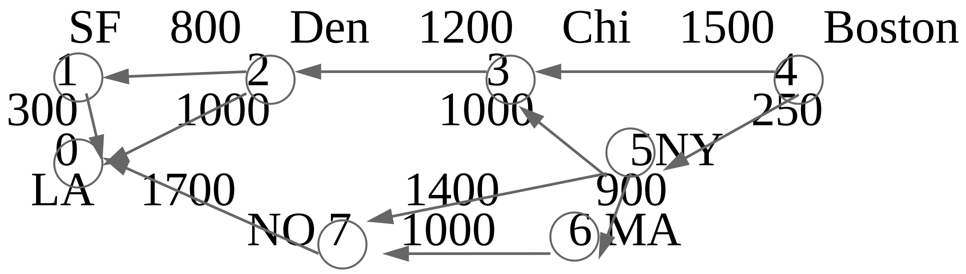
Vertex	0	1	2	3	4	5
Distance	0	45	10	25	45	999
S	1	1	1	1	1	1

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