1. 개요

● 목차

1) 트리 정의 및 표현
   - Definitions
   - 3 Reresentation: Adjacency Matrix, Adjacency List, Adjacency M-list

2) 그래프 기본 연산 (Elementary Graph Operation)

   - Depth First search, Breadth First search

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

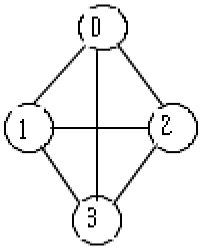
4) Shortest Path (single source all destination)

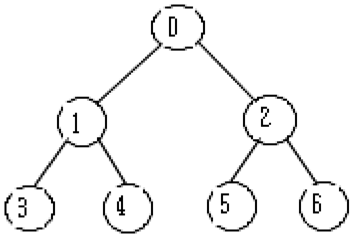
1.1 정의

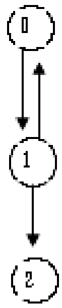
A graph, G, consists of two sets, a finite set of vertices and a finite set of edges.

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

. Undirected Graph (무방향): (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: **<u>no self-loop, and no Multigraph</u>**
  - self-loop : 자기 자신을 가리키는 간선
  - multigraph: 두 정점간에 여러 간선있는 graph

- Complete Graph 란?
  the maximum number of edges 를 갖는 그래프
  (ex. G1 is a complete Graph, G2, G3 is not complete graph)

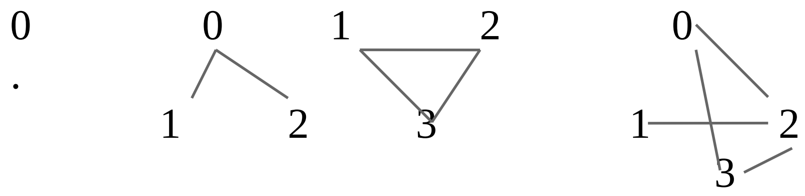
  ⇨ For undirected graph max number of edges **=> n(n-1)/2**
  ⇨ For directed graph, max number of edges **=> n(n-1)**

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



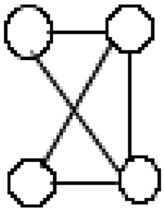
- Cycle and Path

Cycle path

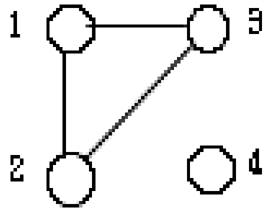
- Path(경로) : 정점(간선)들의 연속
  - 경로의 길이 : 경로상의 간선 수
  - Simple PATH(단순경로) : 서로 다른 정점으로 구성된 경로

- 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

- Connected Component: number of subgraphs
  ex) G4 has two components H1 and H2

  * 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
  (ex. G1, G3 is not Tree, G2 is tree)

- 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

G3

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

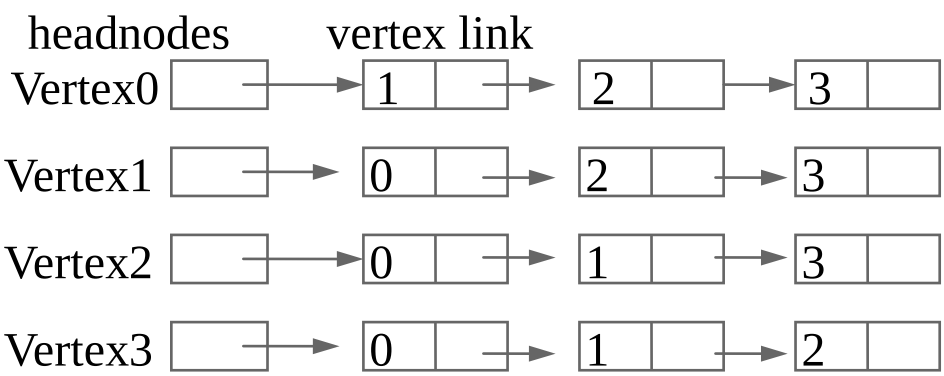
. Directed Graph : **Not symmetric**
. Degree of vertex: row sum -> outdegree
col sum -> indegree

- **Space complexity = $O(n^2)$**
- Sparse graphs 란?: 간선의 개수가 적은 그래프를 뜻함
- sparse graph 를 adjacency matrix 로 표현하면 memory waste 임. adjacency list 가 적합함.

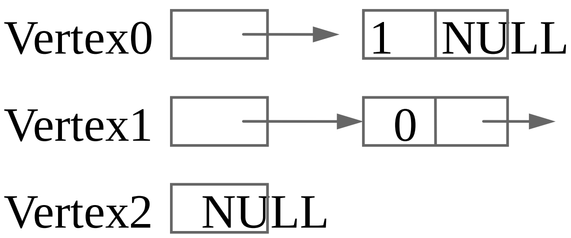
2) Adjacent List **space complexity = O(n+e)// e=edges**

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>)

For G1:



For G3:



3) Adjacent Multilist

인접리스트에서는 각 간선이 두 번 표현되었음 (예: (1,0))

=> Multilist 로 해결 가능

- Node structure

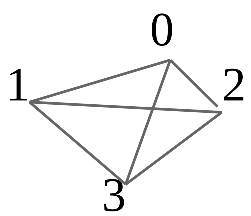
marked	vertex1	vertex2	path1	path2

```
typedef   struct   edge *edge-ptr;
typedef   struct   edge {
              shortint marked;
              int vertex1;
              int vertex2;
              edge-ptr path1;
              edge-ptr path2;     };
edge-ptr   graph[max];
```

For G1: headnodes

0
1
2
3

	marked	v1	v2	for list1	for list2	
N1		0	1	N2	N4	edge(0,1)
N2		0	2	N3	N4	edge(0,2)
N3		0	3	null	N5	edge(0,3)
N4		1	2	N5	N6	edge(1,2)
N5		1	3	null	N6	edge(1,3)
N6		2	3	null	null	edge(2,3)



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

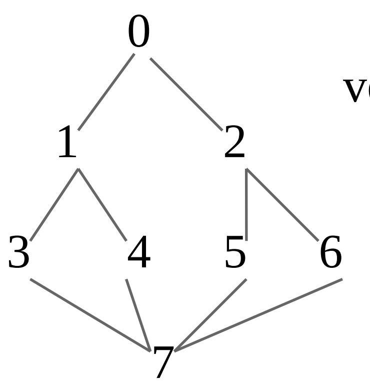
2. Elementary Graph Operations

1) DFS (Depth First Search): 깊이 우선탐색

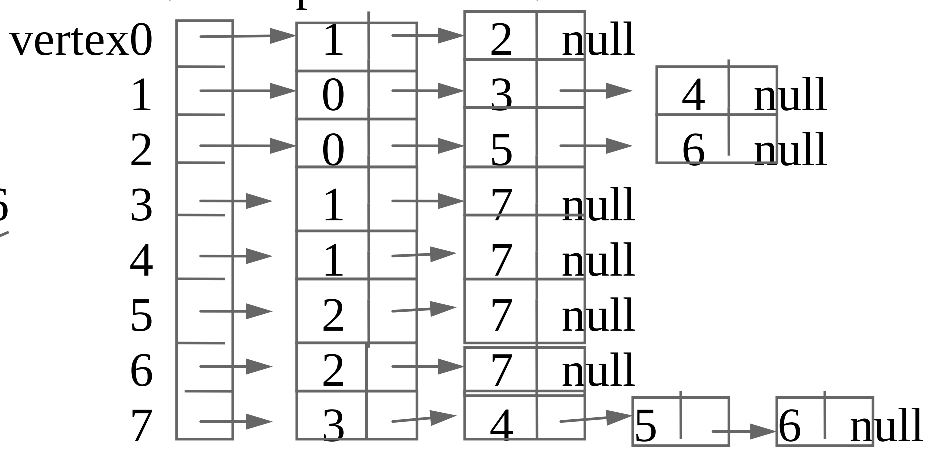
. use Adjacency linked list
. *visited*[*MAX_VERTICES*] : 배열 (초기치 = FALSE)
. *visited*[*i*] = TRUE : 정점 *i* 방문

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

    }
  }
```

. 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

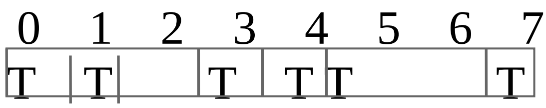


A. list representation:



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

visited

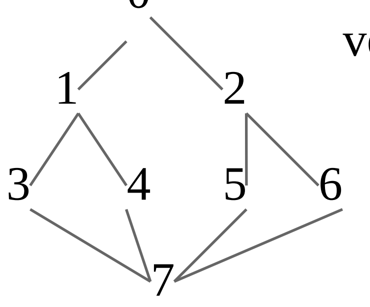


- Analysis: total time of DFS by A. List: O(e),
  by A. matrix: $O(n^2)$

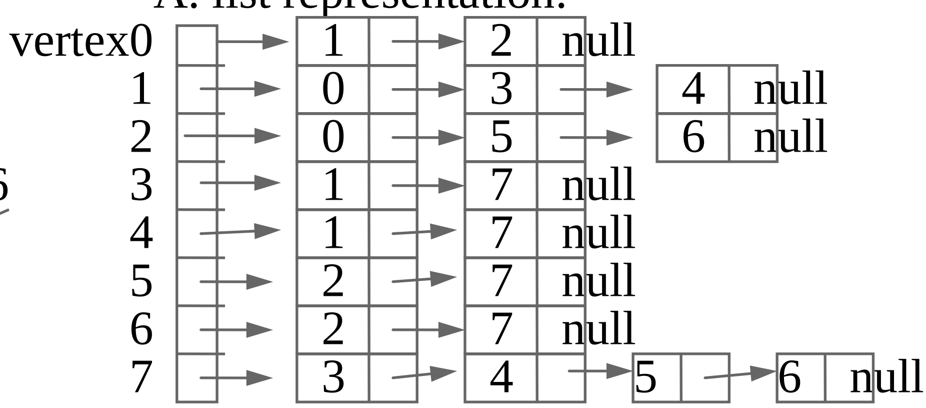
2) **BFS (Breadth First Search)** . Implement with Linked Queue

. **void Graph::BFS(int v) {**

```
Node p;

visited[v] = true;          addq(Q, v);
cout<< v;

while (front) {
    v = dequeue();
    for (p=graph[v]; p; p=p->next;   //인접된 모든 노드에 대해
        if (!visited[p->vertex]) {        //if not visited
                eneuque(p->vertext);
                visited[w] = 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

Q | 1 | 2 | …. | | |

- Analysis of BFS: same as DFS

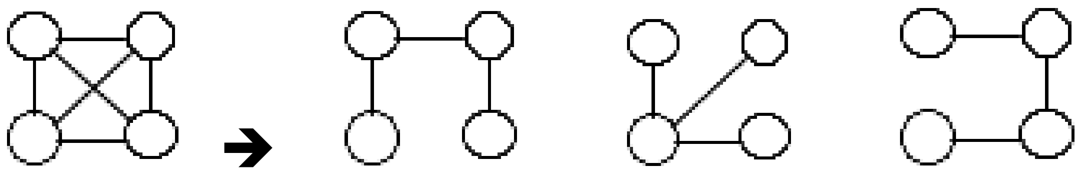
- Sample Code

```
class node {
        int vertex;          node *next;
        node(int num)    { vertex = num;      next = 0;}
    friend class Graph;
};

class Graph {
        private:
            node *graph[MaxVertices];
            bool    visited[MaxVertices];
            node    *front;      node *rear;
        public:
            Graph() {front = 0; rear = 0;}
            void initGraph();
            void insertGraph(int num1, int num2);
            void displayGraph();
            void enqueue(int v);
            int dequeue();
            void bfs(int v);
    };

void Graph::initGraph()
{    for (int i = 0; i < MaxVertices; i++) {
            graph[i] = 0;      visited[i] = false;      }      }

void Graph::enqueue(int v)
{      node *temp = new node(v);      ……………..
   }

int Graph::dequeue()
{      node *p;      int vertex;      .......................
   }
```

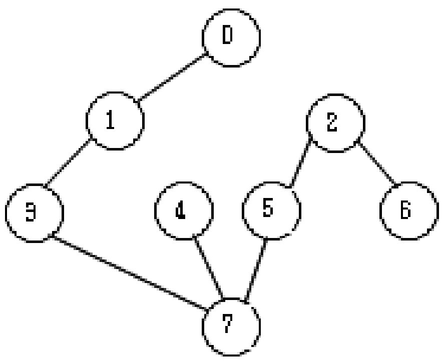
3) SPANNING TREES (신장트리)

Definition: any tree that includes all the vertices in G

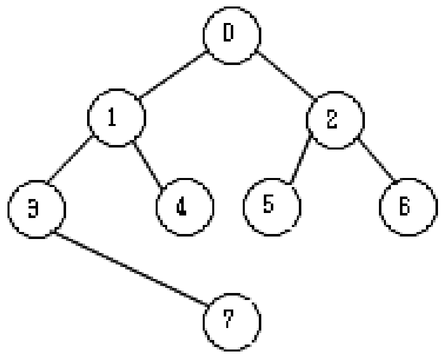
예) 하나의 연결 G 는 출발점이나 검색방법에 따라 각기 다른 신장 트리가 만들어진다.



- 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)

- **Greedy Method**

문제를 해결하는 각 단계에서 가장 최선의 방법을 결정하는 것으로 optimal solution 을 구할 수 있다. (best solution 이 아닐 수도 있다)

예) 현재, 11₵, 5₵, 1₵ 가 있는데 15₵ 만들기

➔ 1-11₵, 4-1₵ 로 만들 수 있다.

그러나 실제로는 3-5₵ 로 만들 것이다.

3. Minimum cost spanning trees(MST)

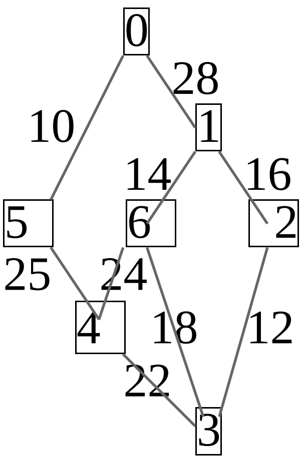
▶ cost가 제일 적은 신장트리

▶ Greedy Method 의 응용 예 ▶ 총 (n-1) edges 이다

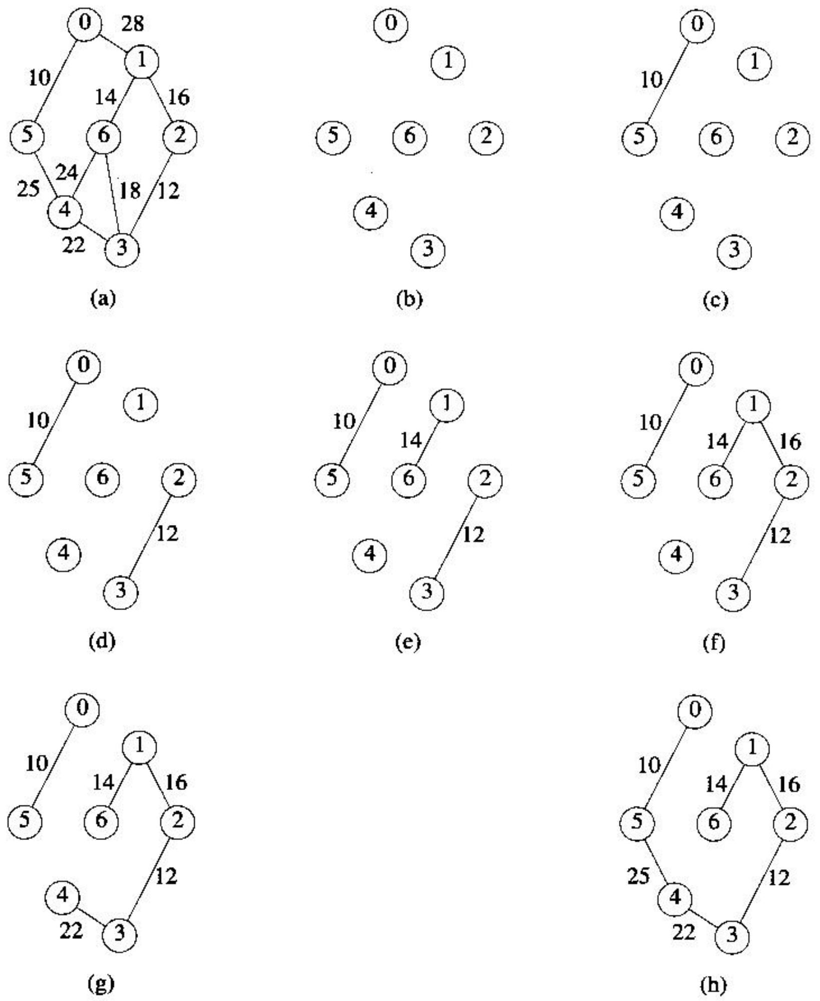
● 대표적인 MST algorithms
⇨ Kruskal's, Prim's, Sollin's

1) Kruskal's Algorithm (Greedy method)

```
T = { };
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)
              add(v,w) to T      => ACCEPT
    else      discard(v,w);      => REJECT
}
If T contains fewer than n-1 edges than
      printf("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



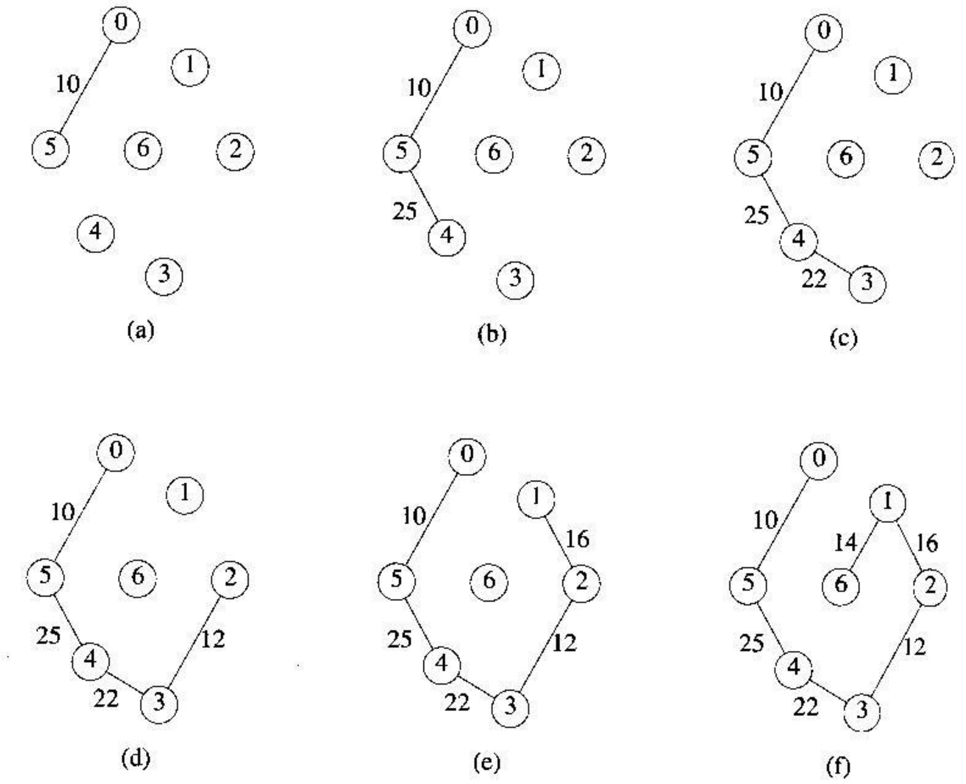
2) Prim's Algorithm- 각 단계에서 선택된 간선의 집합 = 트리

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

Algorithm : {can start with any vertex}

```
T = { };
TV={0} //start with vertex 0//
while  (T contains fewer than n-1 edges)  {
    let (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;
}
if  (T contains fewer than n-1 edges)
            printf ("No spanning Tree");
```

ex) starting vertex ‘0’



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 1 5 6 2 4 3 =>

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)

. 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)

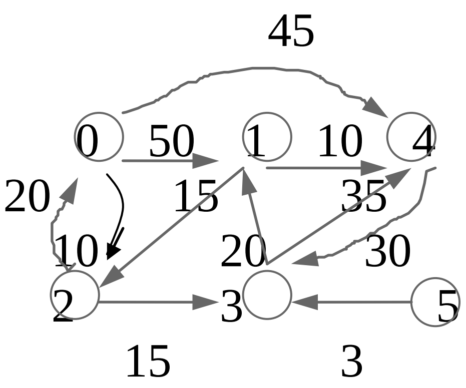
. Tree{2,3,4}:=> (2,1),(3,6)(4,6), will select (2,1) since cost is 16 (min)

⇨ 결과 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)

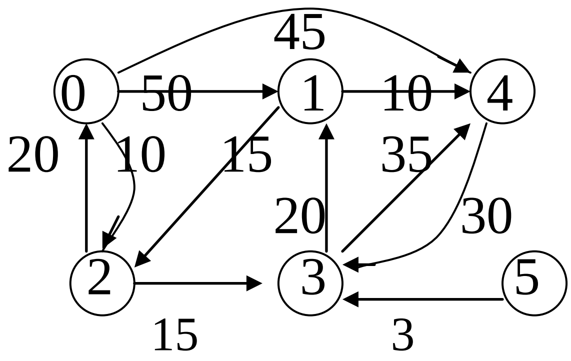
- 초기치 : *distance*[*i*] = *cost*[0][*i*]
- *cost*[*i*][*j*] : <*i*, *j*>의 가중치

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

```
void initCostMatrix(int cost[][8])    {
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] = 10000;
}
```

- 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-2; 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
} }

int choose(int dist[], int n, bool found[])
{       int i, min, minpos;

        min = INT_MAX;
        minpos = -1;

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



	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. $S = S \cup \{v3\} = \{v0, v2, v3\}$

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. $S = S \cup \{v1\} = \{v0, v1, v2, v3\}$

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

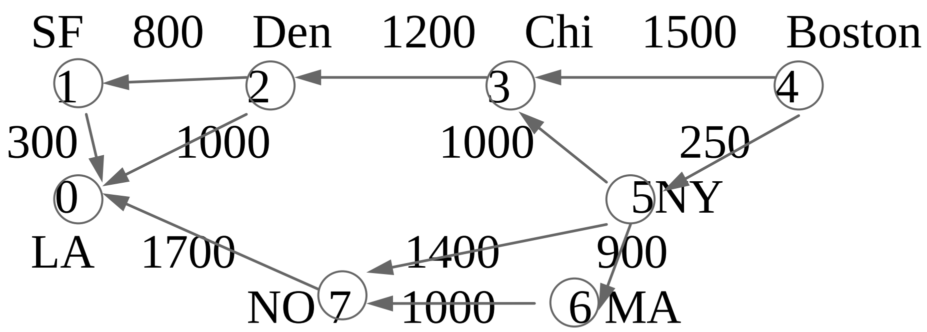
5. $S = S \cup \{v4\}$

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. $S = S \cup \{v5\}$

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 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 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**	**<u>250</u>**	**∞**	**∞**
1	**4**	**5**	**∞**	**∞**	**∞**	**1250**	**0**	**250**	**<u>1150</u>**	**1650**
2	**4,5**	**6**	**∞**	**∞**	**∞**	**<u>1250</u>**	**0**	**250**	**1150**	**1650**
3	**4,5,6**	**3**	**∞**	**∞**	**2450**	**1250**	**0**	**250**	**1150**	**<u>1650</u>**
4	**4,5,6,3**	**7**	**3350**	**∞**	**<u>2450</u>**	**1250**	**0**	**250**	**1150**	**1650**
5	**4,5,6,3,7**	**2**	**3350**	**<u>3250</u>**	**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}									

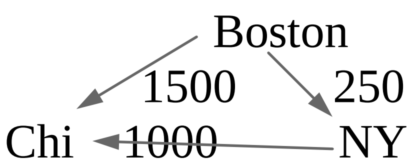
* 250 + 1000 < 1500

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