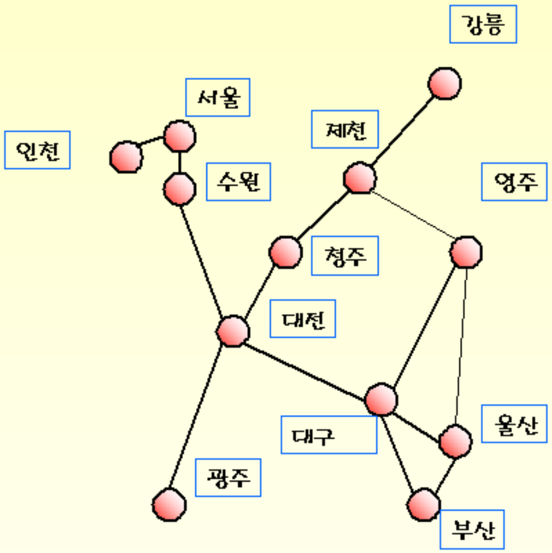
Chapt. 10

그래프(Graph)-1

그래프(Graph)

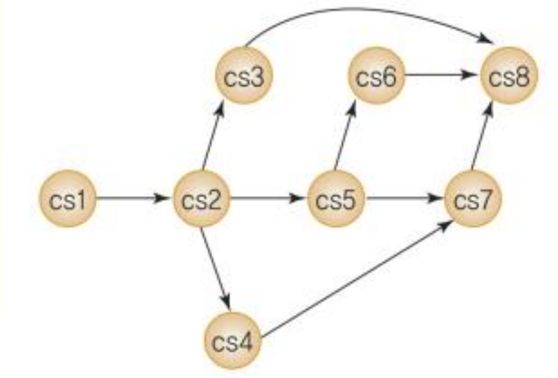
- **그래프** : 어떤 연관성에 의해 연결된 객체 간의 관계를 표현하는 자료구조

  [예] 전기회로, 업무 스케줄, 공정 관리, 네비게이션 맵, 순서도 등



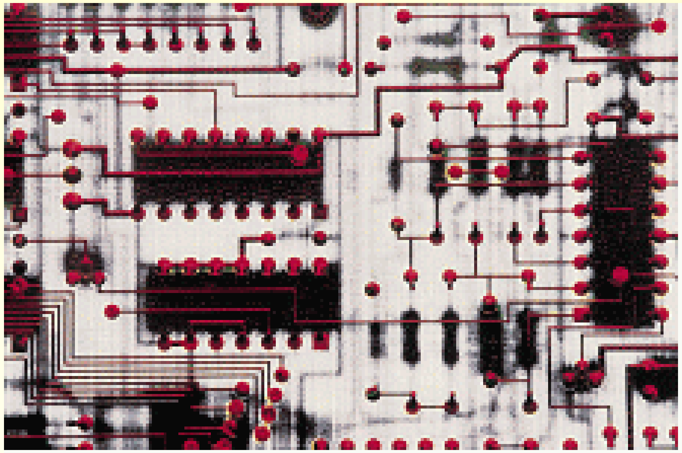
그래프(Graph)

- 그래프는 소프트웨어 분야에서 많이 사용하는 일반적인 자료구조에 속함
  : Tree도 그래프에 포함(cycle없는 그래프 : 노드 수 = 간선 수 - 1)
- 그래프 이론(graph theory) : 주어진 문제의 내용이 서로 연관이 있을 때, 내용들과 연관성 등을 그래프로 표현해서 문제 해결하려는 분야
  - 수학자 오일러에 의해 시작된 수학응용 분야



선이수 체계도

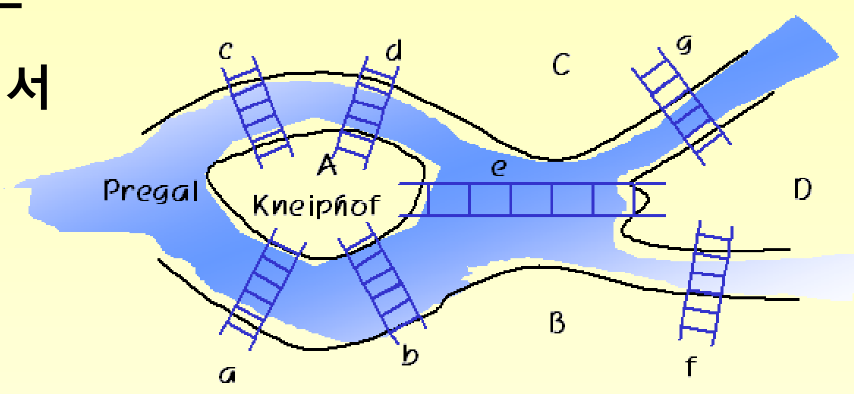
cs1 : c언어　　cs2 : 자료구조
cs3 : JAVA언어　cs4 : C#언어
cs5 : 알고리즘　cs6 : 운영체제



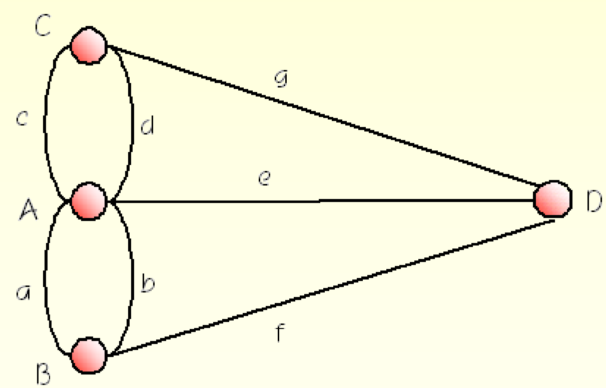
메인보드 회로 연결도

그래프 이론의 역사

- 1800년대 오일러(Euler)에 의하여 창안
- 오일러 문제 : 모든 다리를 1번씩 건너서 처음 출발했던 장소로 돌아오는 문제
  - 문제의 핵심을 그래프로 표현한다면?
    - 땅 : 정점(node)
    - 다리 : 간선(edge)
- 오일러 경로(Euler path)
  정점에 연결된 간선의 개수가 짝수이면 원래 지점으로 돌아올 수 있는 경로가 존재한다.
  ➔ 홀수 개이면 오일러 경로는 없음



(a) 모든 다리를 한번만 건너 돌아오는 경로 문제

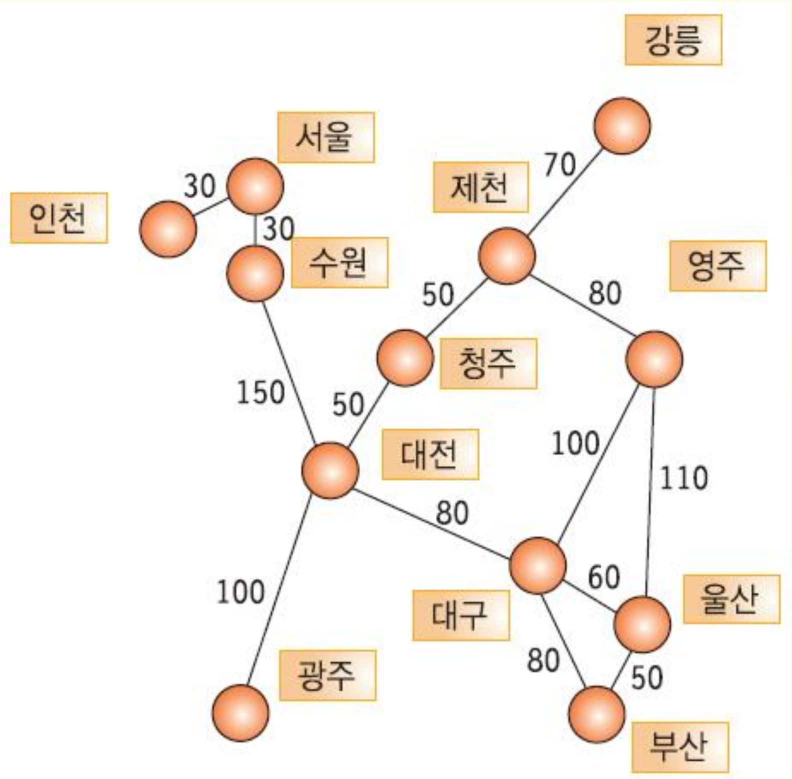


(b) 문제 (a)의 그래프 표현

그래프 용어

- 그래프는 G(V, E)로 표시
  - V : 정점(vertex)들의 집합
  - E : 간선(edge)들의 집합
  - 정점과 간선은 관련정보를 자체적으로 저장할 수 있음

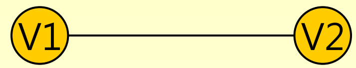
- 예제 그래프 : 지도(map)
  - 정점은 각 도시를 의미
  - 간선은 도시를 연결하는 도로를 의미 : 거리, 시간, 비용 등의 데이터로 표현



그래프의 종류

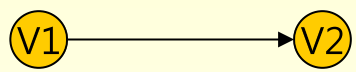
- 에지 종류에 따라 그래프는 2가지의 형태로 구분
  - **무방향 그래프(undirected graph)** : 방향성이 없는 간선을 통해서 양방향으로 갈수 있음. (V_1, V_2)와 같이 정점의 쌍으로 표현

    $(A, B) = (B, A)$

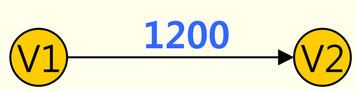


  - **방향 그래프(directed graph)** : 방향성이 존재하는 간선을 가진 그래프. 일방통행 도로처럼 한쪽 방향으로만 갈 수 있음을 의미. $\langle V_1, V_2 \rangle$로 표시

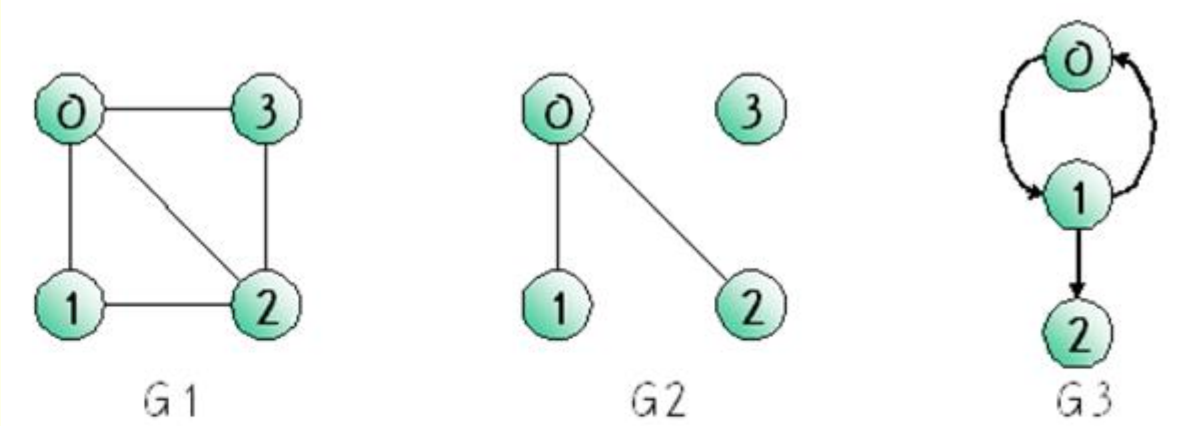
    $\langle V_1, V_2 \rangle \neq \langle V_1, V_2 \rangle$



- **가중치 그래프(weighted graph)** or 네트워크(network) : 간선에 가중치(비용, 거리 등)가 주어져 있는 그래프



그래프 표현의 예



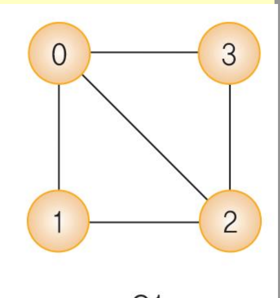
V(G1)= {0, 1, 2, 3}, E(G1)= {(0, 1), (0, 2), (0, 3), (1, 2), (2, 3)}

V(G2)= {0, 1, 2, 3}, E(G2)= {(0, 1), (0, 2)}

V(G3)= {0, 1, 2}, E(G3)= {<0, 1>, <1, 0>, <1, 2>}

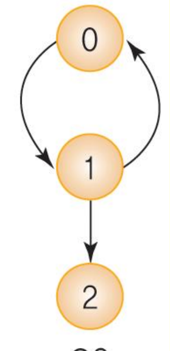
그래프 용어

- 인접한 정점(adjacent vertex) : edge에 의해 연결된 정점
  - G1에서 (0, 1), (0, 2), (0, 3), … , (2, 3)
  - G3에서 <0, 1>, <1, 0> <1, 2>



G1

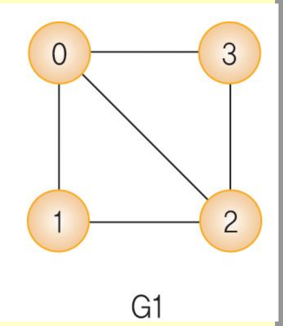
- **차수(degree)** : 정점에 연결된 다른 정점의 개수(간선 개수)
  - G1에서 V_1 의 차수 = 2
  - G3에서 V_1 의 진입차수 = 1, 진출차수 = 2



G3

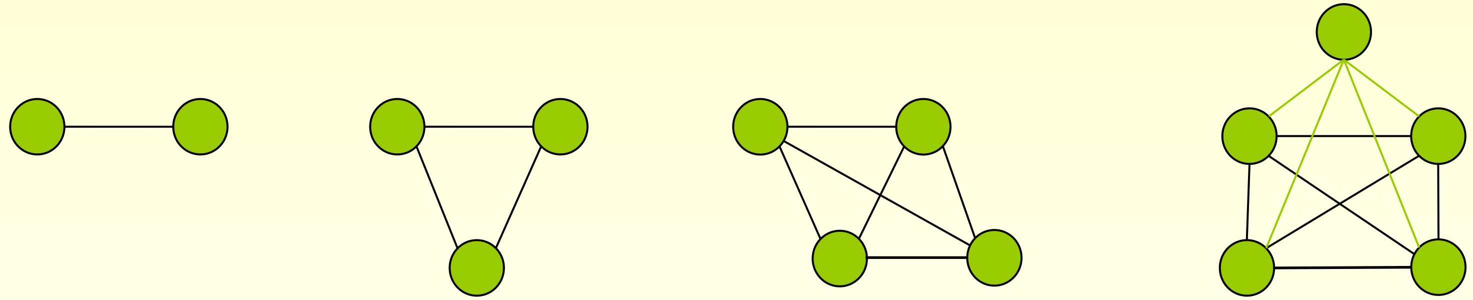
- **경로(path)** : 출발점에서부터 도착점까지의 정점의 나열
  - G1에서 0번 정점~3번 정점 사이의 단순(simple) 경로 : 0, 1, 2, 3
    사이클(cycle) : 0, 1, 2, 3, 0
  - G3에서 0번 정점~2번 정점 사이의 단순(simple) 경로 : 0, 1, 2
    0번 정점~2번 정점 사이의 사이클(cycle) : 0, 1, 0, 1, 2
    2번 정점~1번 정점 사이의 단순(simple) 경로 : 없음

그래프 용어



- **경로 길이 : 경로를 구성하는데 사용된 edge 개수**
  - **G1에서 0번 정점~3번 정점 사이의 경로 길이 : 0, 1, 2, 3 = 3**

- **완전 그래프 : 모든 정점들이 서로 연결되어 있는 그래프**
  - **특징 : 간선 개수 = (정점 개수-1)*(정점 개수) / 2**

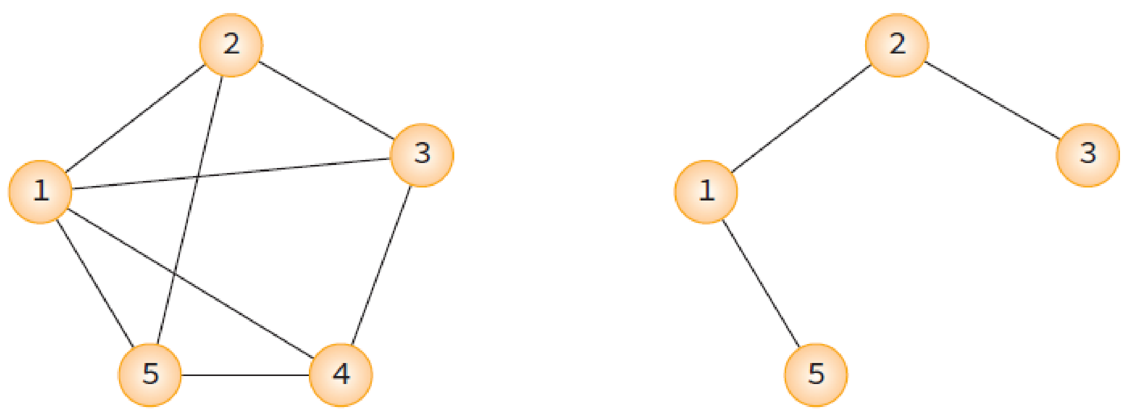


정점 2 간선 1	정점 3 간선 2	정점 4 간선 6	정점 5 간선 10

그래프 용어

- **부분 그래프(sub graph)**
  - 정점 집합 V와 간선 집합 E의 일부 데이터로 구성된(부분 집합) 그래프
  - <예> 그래프 G1의 부분 그래프 중의 하나

$$V(S) \subseteq V(G)$$
$$E(S) \subseteq E(G)$$



(a) 그래프　　(b) 부분 그래프

그래프 ADT

· 객체: 정점의 집합과 간선의 집합

· 연산

- create_graph() ::= 그래프를 생성한다.
- init(g) ::= 그래프 g를 초기화한다.
- insert_vertex(g,v) ::= 그래프 g에 정점 v를 삽입한다.
- insert_edge(g,u,v) ::= 그래프 g에 간선 (u,v)를 삽입한다.
- delete_vertex(g,v) ::= 그래프 g의 정점 v를 삭제한다.
- delete_edge(g,u,v) ::= 그래프 g의 간선 (u,v)를 삭제한다.
- is_empty(g) ::= 그래프 g가 공백 상태인지 확인한다.
- adjacent(v) ::= 정점 v에 인접한 정점들의 리스트를 반환한다.
- destroy_graph(g) ::= 그래프 g를 제거한다.

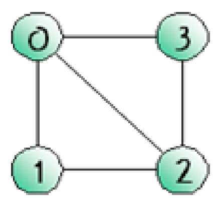
- 그래프에 정점을 추가하려면 insert_vertex() 연산을 사용하고, 간선을 추가하려면 insert_edge() 연산을 사용

그래프 표현 방법

- 그래프를 표현하는 2가지 방법
  - 인접행렬(adjacent matrix) : 2차원 배열 사용, 연결 여부를 간선으로 표현
  - 인접리스트(adjacency list) : 연결리스트를 사용, 연결 여부를 링크로 표현

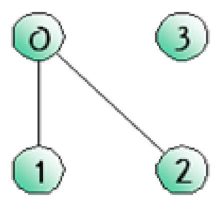
1) 인접행렬 방법

구성방법 : 간선 (i, j)가 그래프에 존재 ➔ 배열 M[i][j] = 1, 아니면 M[i][j] = 0



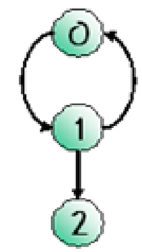
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

(a)



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

(b)



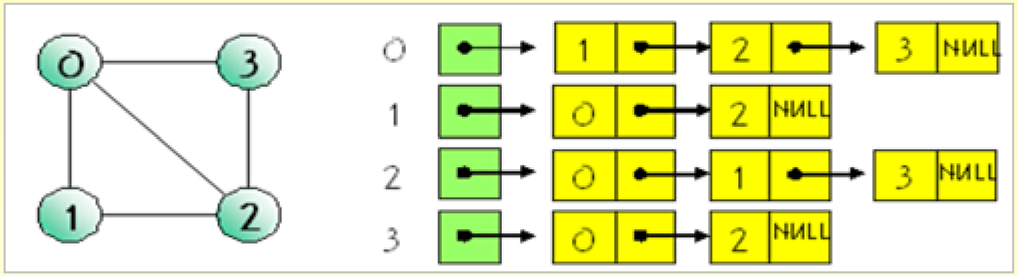
	0	1	2
0	0	1	0
1	1	0	1
2	0	0	0

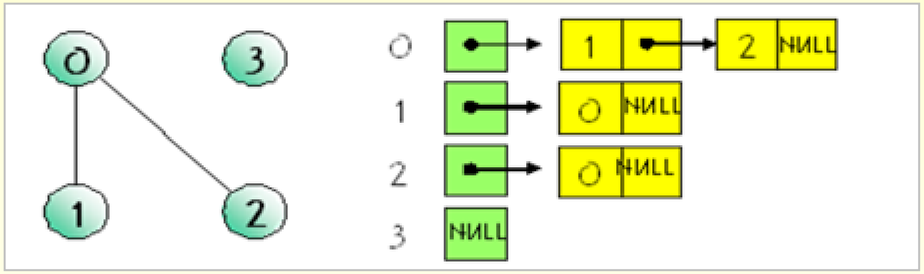
(c)

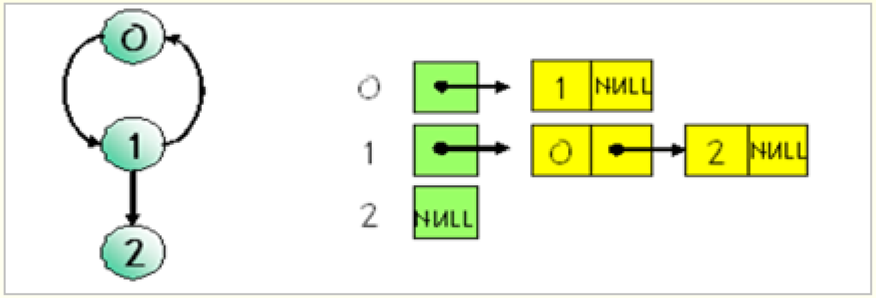
그래프 표현 방법(cont.)

2) 인접리스트 방법

- 각 정점에 인접한 정점들을 연결리스트로 표현



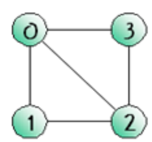




그래프 표현 방법 : 비교

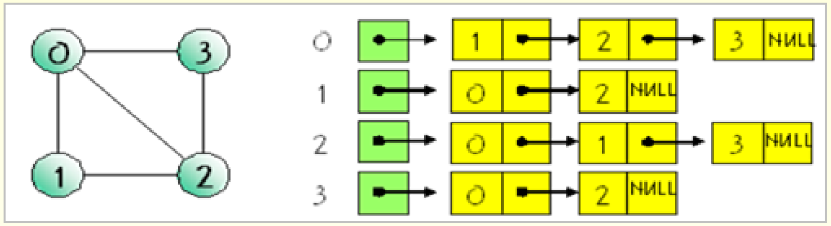
- **인접 행렬**
  - 배열을 사용하여 구현
  - 배열 요소는 정점 사이의 연결(edge의 유무)을 표현
  - 사용하기 편함
  - 처리과정의 시간성능이 약간 느림 : $O(n^2)$
  - 입력 : 배열 초기값으로 입력 가능

- **인접 리스트**
  - 연결리스트를 사용하여 구현
  - 각 노드는 정점에 연결(edge 유무)된 정점을 표현
  - 사용하기 약간 불편함
  - 처리과정의 시간성능이 약간 빠름 : $O(n+e)$
  - 입력 : 입력함수 사용해야 함



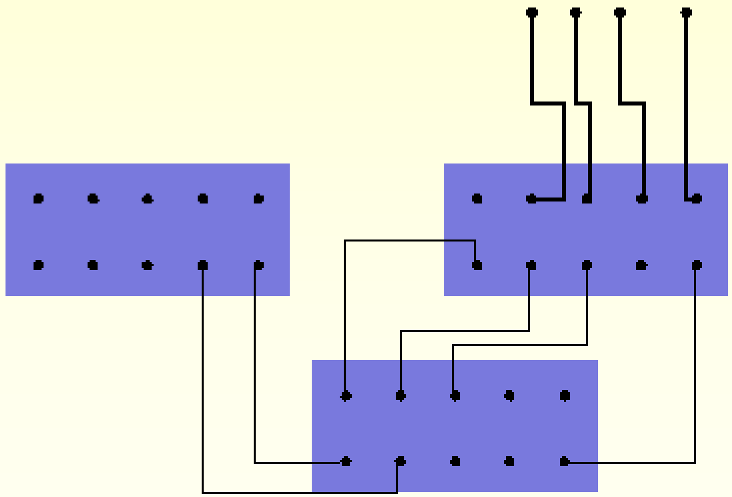
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

(a)



그래프 탐색

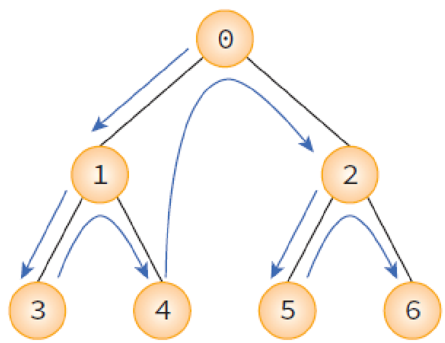
- 그래프 탐색은 주어진 그래프에서의 가장 기본적인 연산
  : 하나의 정점으로부터 시작하여 차례대로 모든 정점들을 한번씩 방문한다
- 단순히 그래프의 노드를 탐색하는 것으로 많은 문제들을 해결되나?
  (예) 하나의 정점에서 다른 정점으로 가는 경로가 있는지, 있다면 최소 경로는?
  (예) 전자 회로에서 특정 단자와 단자가 서로 연결되어 있는지?
  (예) 업무 스케줄링, 여행경로, 일정 조회, 예약시스템 등은 그래프로 표현 가능



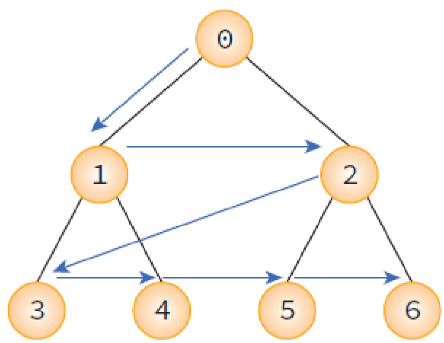
그래프 탐색(cont.)

- 깊이우선탐색(DFS: depth-first search)
  한 정점에 연결된 다른 정점으로 탐색을 시작(함수 call) ➔ 그 정점에 연결된 다른 정점으로 탐색(함수 call) ➔ 이 과정을 반복 ➔ 더 이상 탐색할 정점이 없으면?
  : 아직 탐색되지 않은 정점을 만날 때까지 되돌아가서(함수 return) 계속 진행

- 너비우선탐색(BFS: breadth-first search)
  시작 정점을 Queue에 삽입 ➔ Queue에서 삭제/출력하고 연결된 다른 정점들을 Queue에 저장 ➔ Queue에서 삭제/출력하고 연결된 정점들을 Queue에 저장 ➔ Queue가 빌 때까지 반복



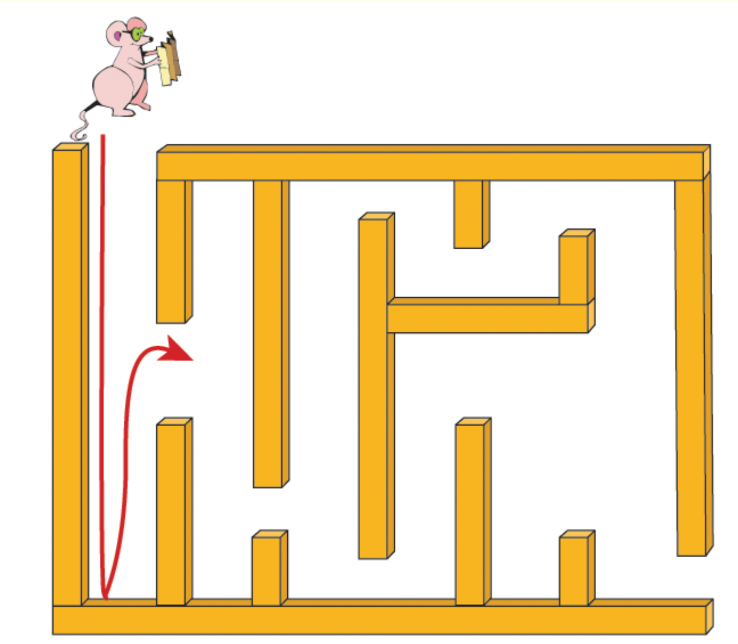
깊이 우선 탐색



너비 우선 탐색

그래프 탐색(cont.)

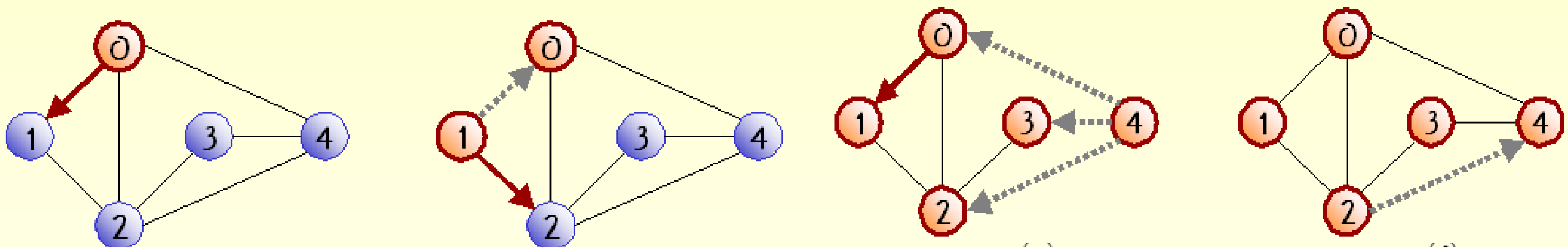
- 깊이 우선 탐색(DFS) : 한 방향으로 갈 수 있을 때까지 계속 탐색해가다가 더 이상 갈 수 없게 되면 다시 가장 가까운 갈림길로 돌아와서 그 곳에서 다른 방향으로 다시 탐색을 진행하는 방법
  - 진행해가기 : 함수 call
  - 되돌아가기 : 함수 return



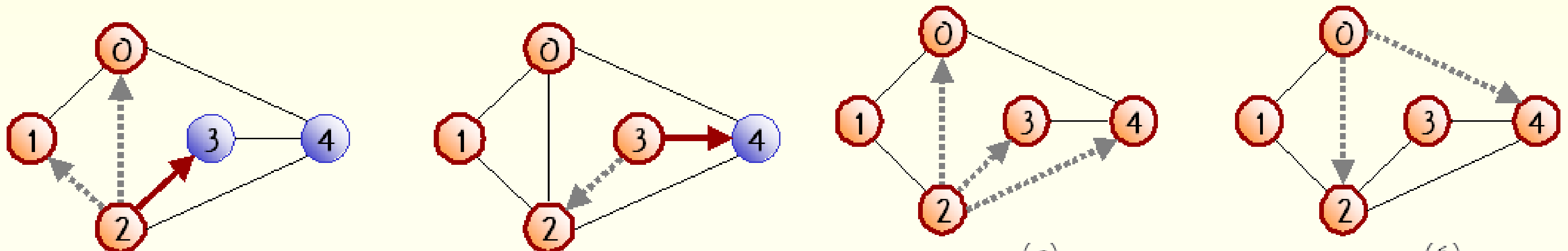
깊이우선탐색(DFS)

```
void  depth_first_search(정점 v)
{
   정점 v 방문 & 표시 ; // 방문기록 표시 & 출력
   for (v에 인접한 정점 u가 있으면) do
      if (u의 방문표시가 없으면) depth_first_search(u);
}
```

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



(a) (b) (e) (f)



(c) (d) (g) (h)

DFS 프로그램

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

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

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

DFS 프로그램

```
...
void dfs_mat(int g[], int v) {
     int w;  visited[v] = TRUE;
     printf("정점 %d -> ", v);
     for (w = 0; w<g->n; w++)
          if (g->adj_mat[v][w] && !visited[w])  dfs_mat(g, w);
}
void main()
{
    int  g[5][5] = { {0,1,1,0,1}, {1,0,1,0,0}, {1,1,0,1,1}, {0,0,1,0,1}, {1,0,1,1,0} };
    printf("\n 깊이 우선 탐색 \n");
    dfs_mat(g, 0);
    printf("종료 \n");
}
```

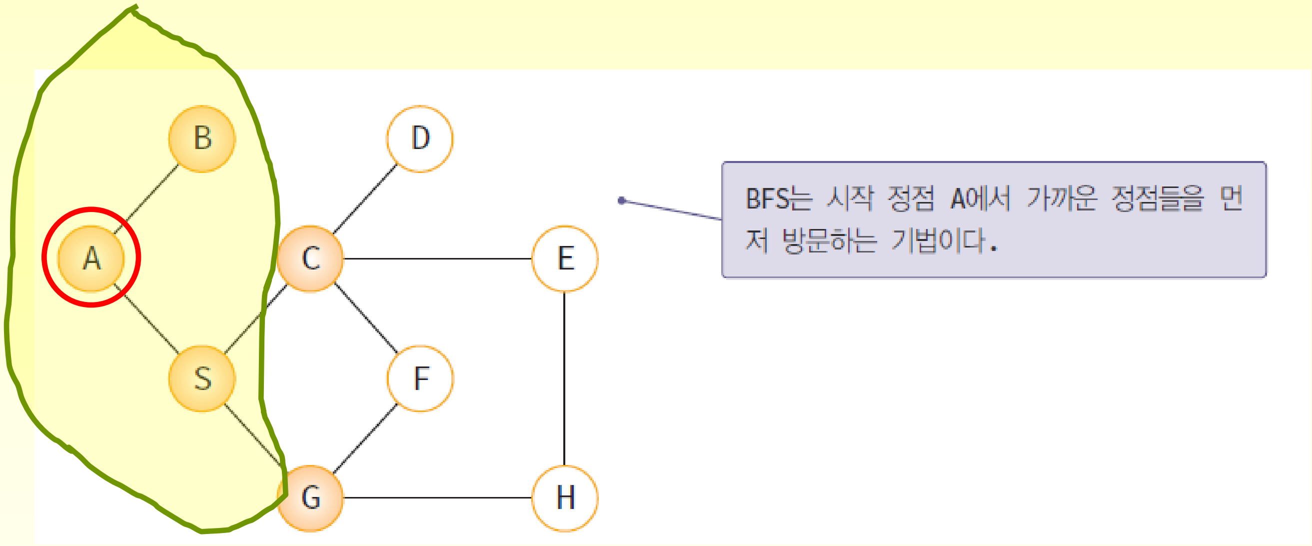
	0	1	2	3	4
0	0	1	1	0	1
1	1	0	1	0	0
2	1	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0

시간 성능 : O(n^2)

```
깊이 우선 탐색
정점 0 -> 정점 1 -> 정점 2 -> 정점 3 -> 정점 4 -> 종료
```

너비우선 탐색(BFS)

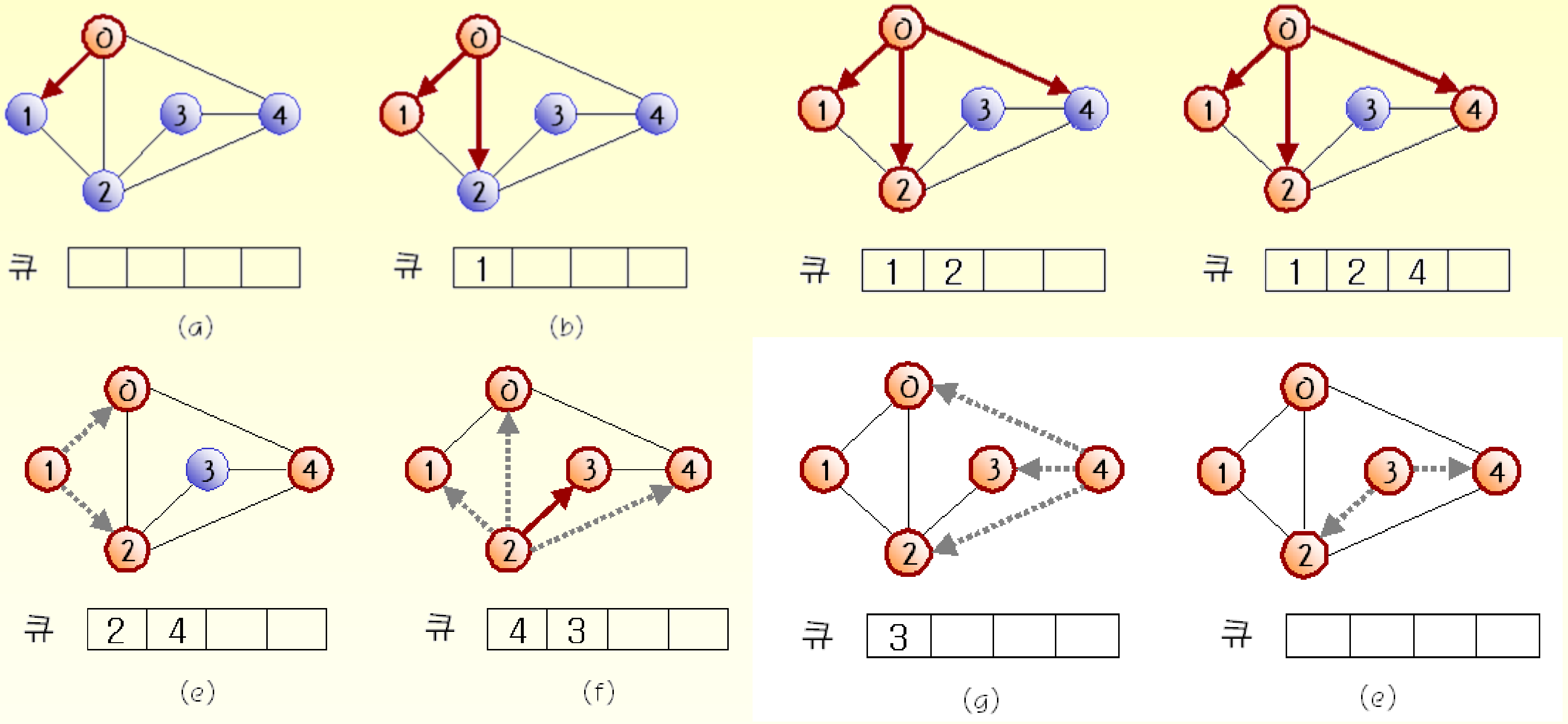
- 시작 정점으로부터 가까운 정점을 먼저 방문하고, 멀리 떨어진 정점을 나중에 방문하는 순회 방법큐를 사용하여 구현됨
  : 가까운 정점들을 Queue에 저장했다가 꺼내면서 방문하는 방법



너비우선탐색(BFS)

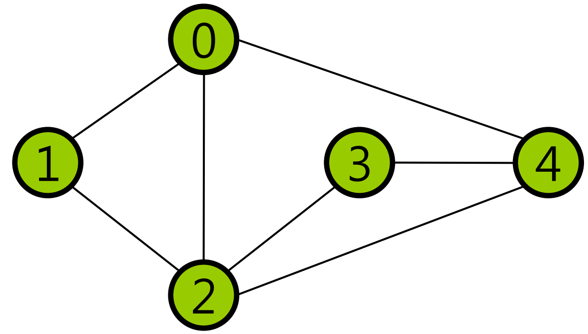
- 시작 정점에서 가까운 정점을 먼저 방문 ➔ 다음 순서의 정점 ➔ … ➔ 멀리 위치한 정점을 나중에 방문

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



BFS 알고리즘

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



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

BFS 프로그램

```
void  bfs_mat(int  g[], int v) {
      int w;    QueueType q;      init(&q);
      visited[v] = TRUE;          printf("정점 %d -> ", v);
      enqueue(&q, v);
      while(!is_empty(&q)){
            v = dequeue(&q);
            for(w=0; w<g->n; w++)
               if(g->adj_mat[v][w] && !visited[w]){
                    visited[w] = TRUE;
                    printf("%d ", w);
                    enqueue(&q, w);  }
      }
}
void main()
{
    int  g[5][5] = { {0,1,1,0,1}, {1,0,1,0,0}, {1,1,0,1,1}, {0,0,1,0,1}, {1,0,1,1,0} };
    printf("\n 너비 우선 탐색 \n");
    bfs_mat(g, 0);
    printf("종료 \n");
}
```

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

queue

시간 성능 : O(n^2)

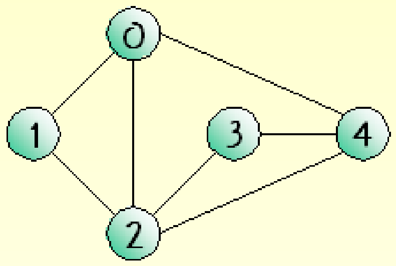
너비 우선 탐색
정점 0 -> 정점 1 -> 정점 2 -> 정점 3 -> 정점 4 -> 종료

Chapt. 11

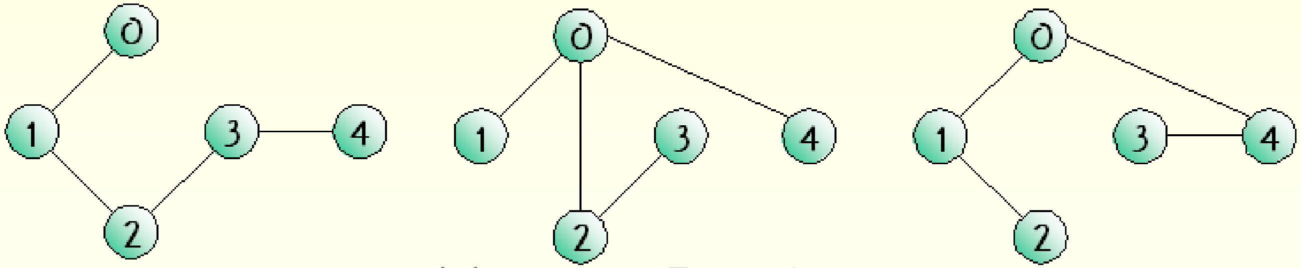
그래프(Graph)-2

신장(spanning) 트리

- 신장트리(spanning tree) : 그래프의 모든 정점을 포함하는 트리
- 조건 : 트리이므로 모든 정점들은 연결되어야 하고, 사이클은 없어야 함
  (n개 정점, n-1개 간선)



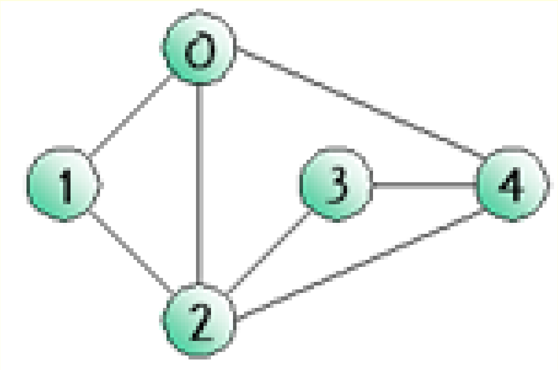
(a) 연결 그래프

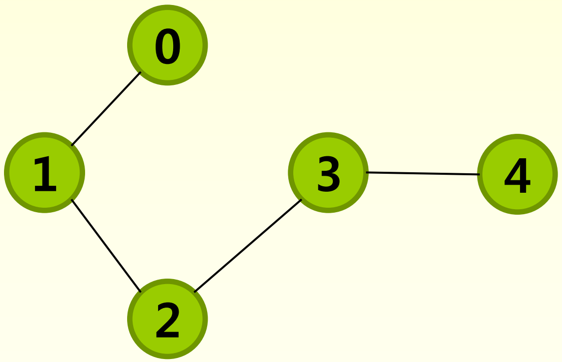


(b) 신장 트리중의 일부

신장트리 알고리즘

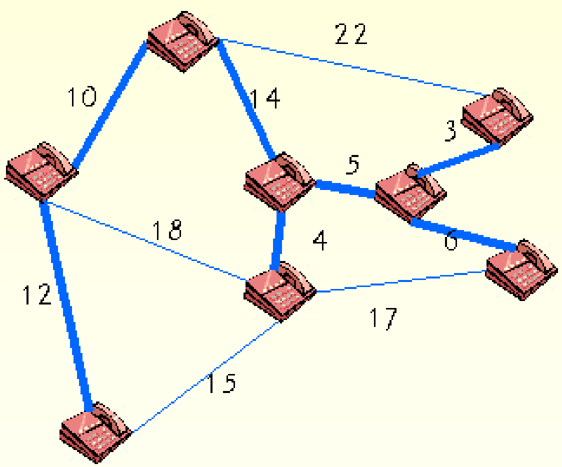
```
depth_first_search(v) {   // 깊이우선탐색 알고리즘으로 신장트리를 구성 가능
   v를 방문 표시;
   for all u ∈ (v에 인접한 정점) do
        if (u가 아직 방문되지 않았으면) { (v, u)를 신장트리의 간선으로 표시;
                                           depth_first_search(u);
                                         }
}
```

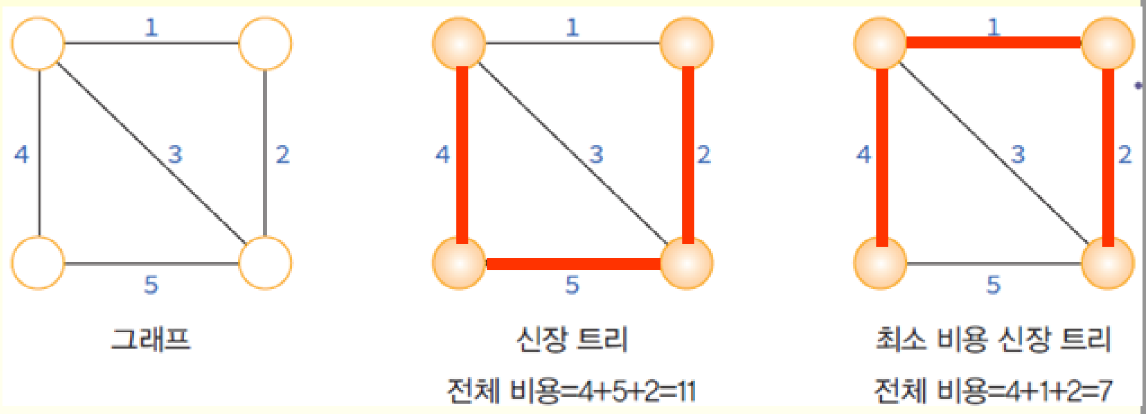




최소비용 신장트리(MST)

- 최소비용 신장트리(MST: minimum spanning tree)
  : 모든 정점들을 가장 적은 개수의 간선(비용)으로 연결하는 신장트리
- MST 응용분야
  - 도로 건설 - 도시들을 모두 연결하면서 도로의 길이가 최소가 되도록 구성
  - 전기 회로 - 단자들을 모두 연결하면서 전선의 길이가 가장 최소가 되도록 구성
  - 통신 - 전화선의 길이가 최소가 되도록 전화 케이블 망을 구성
  - 배관 - 파이프를 모두 연결하면서 파이프의 총 길이가 최소가 되도록 연결





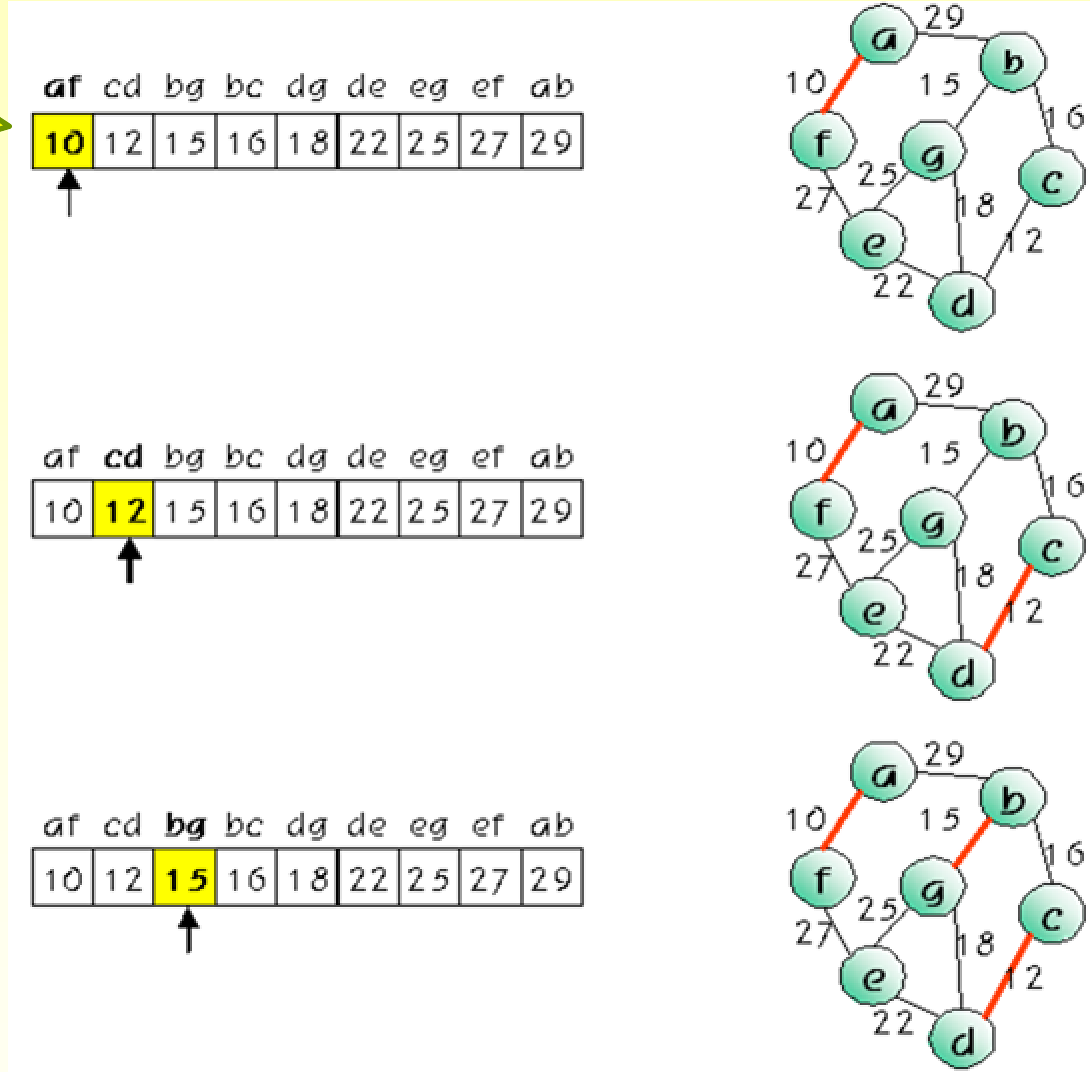
MST 알고리즘

- 2가지의 대표적인 알고리즘
  - Kruskal 알고리즘
  - Prim 알고리즘
- 탐욕적인 방법(greedy method)을 적용
  - 알고리즘 설계방법 중의 한가지 방법
  - 어떤 상황에서 결정할 때마다 그 순간 최선이라고 생각되는 것을 선택 → 최종 해답에 도달할 때까지 반복

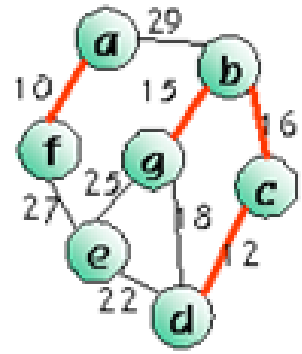
    - 주위할 점 : 선택할 때마다 최적의 답을 구하는데 문제없는 선택인지를 검사해야 함
  - Kruskal, Prim 알고리즘은 최소비용 신장트리를 구하는 최적 방법임이 증명되었음

Kruskal 알고리즘의 동작

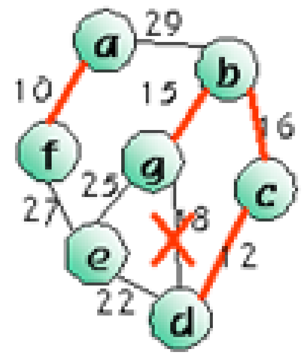
모든
간선
정렬



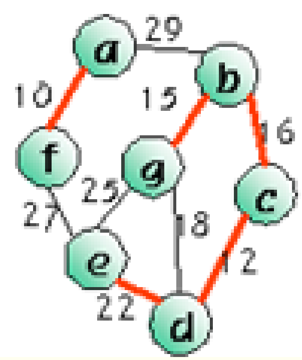
af	cd	bg	**bc**	dg	de	eg	ef	ab
10	12	15	**16**	18	22	25	27	29

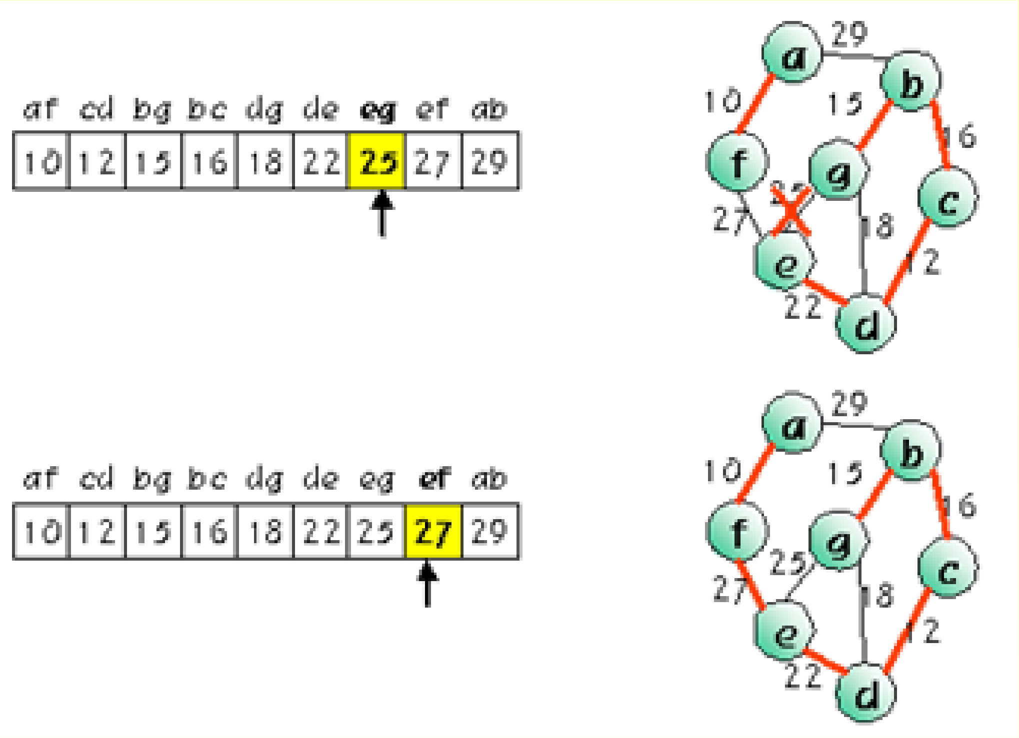


af	cd	bg	bc	**dg**	de	eg	ef	ab
10	12	15	16	**18**	22	25	27	29

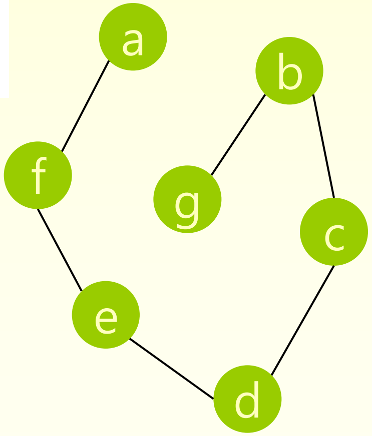


af	cd	bg	bc	dg	**de**	eg	ef	ab
10	12	15	16	18	**22**	25	27	29





최소비용 = 10+27+22+12+16+15

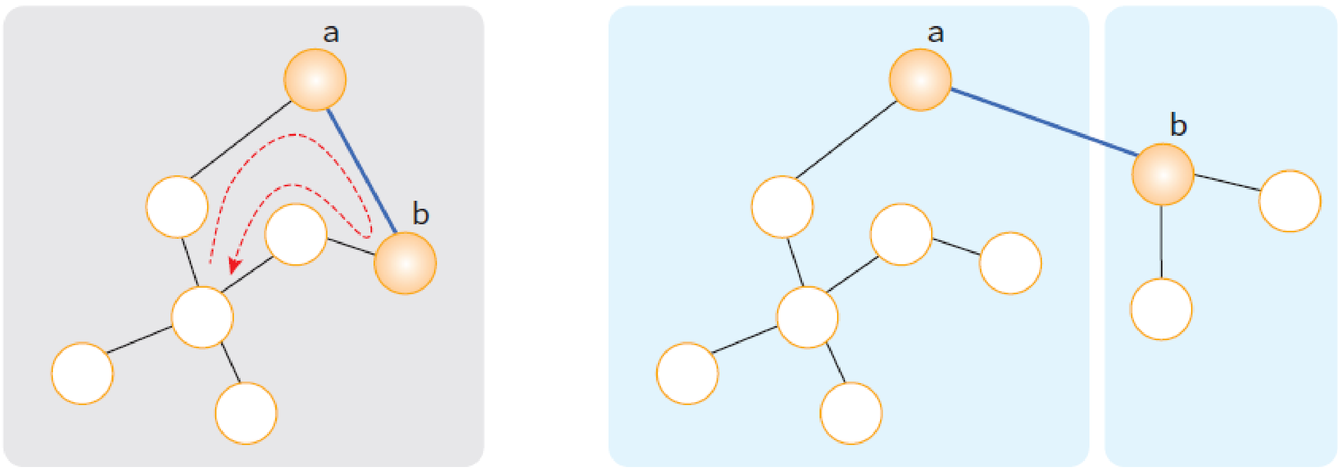


kruskal 알고리즘의 구현

- **union-find** 알고리즘이 필요
  - 집합들을 결합하고, 집합의 원소가 어떤 집합에 속하는지를 판단하는 알고리즘
  - 여러가지 방법으로 구현이 가능하다
  - Kruskal 알고리즘에서는 트리에 간선을 추가할 경우 발생하는 사이클 검사에 사용됨

a와 b가 같은 집합에 속함

a와 b가 다른 집합에 속함

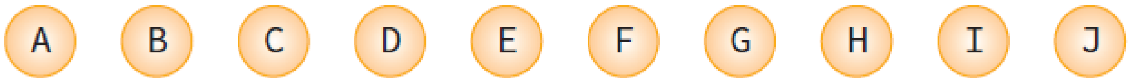


(a) 사이클 형성

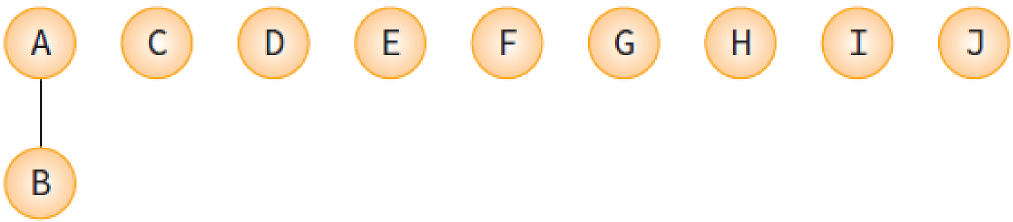
(b) 사이클 형성되지 않음

Kruskal의 MST 알고리즘

- union-find 방법

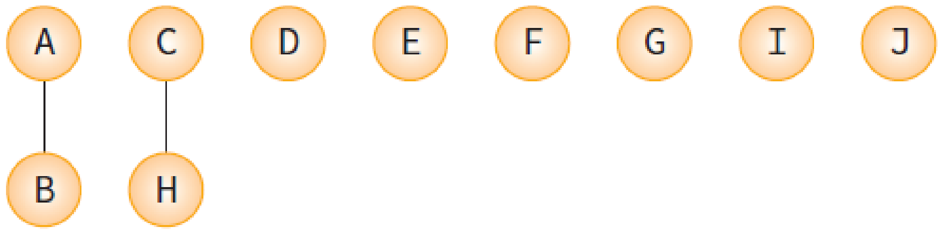


A	B	C	D	E	F	G	H	I	J
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1



A	B	C	D	E	F	G	H	I	J
-1	0	-1	-1	-1	-1	-1	-1	-1	-1

Kruskal의 MST 알고리즘

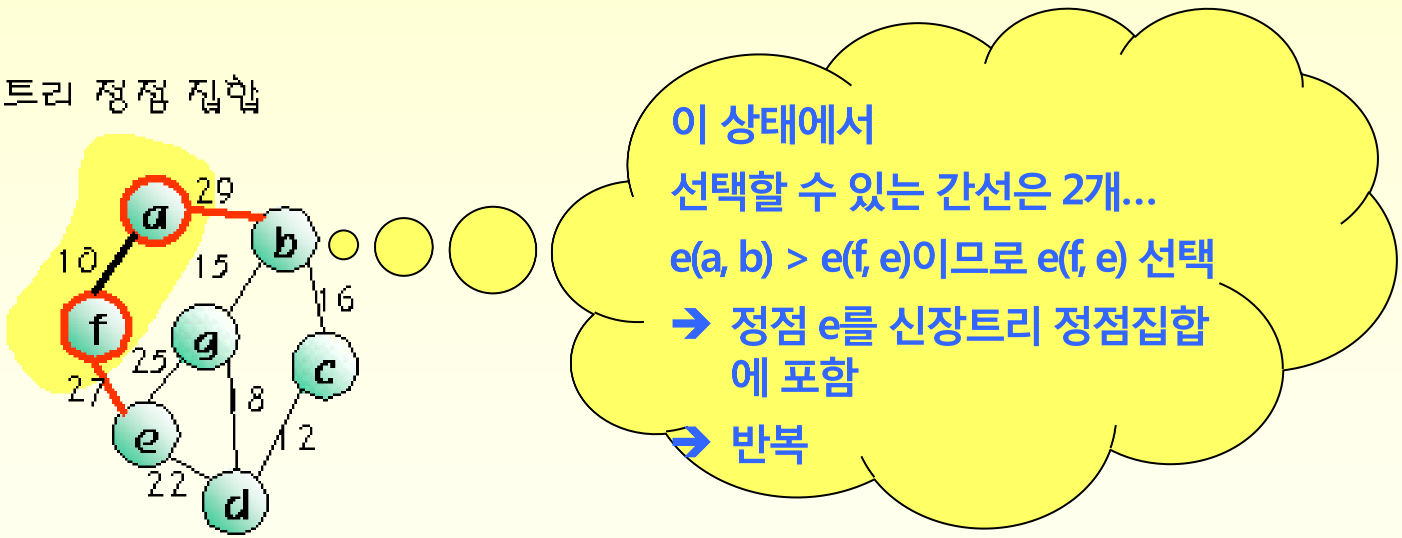


A	B	C	D	E	F	G	H	I	J
−1	0	−1	−1	−1	−1	−1	2	−1	−1

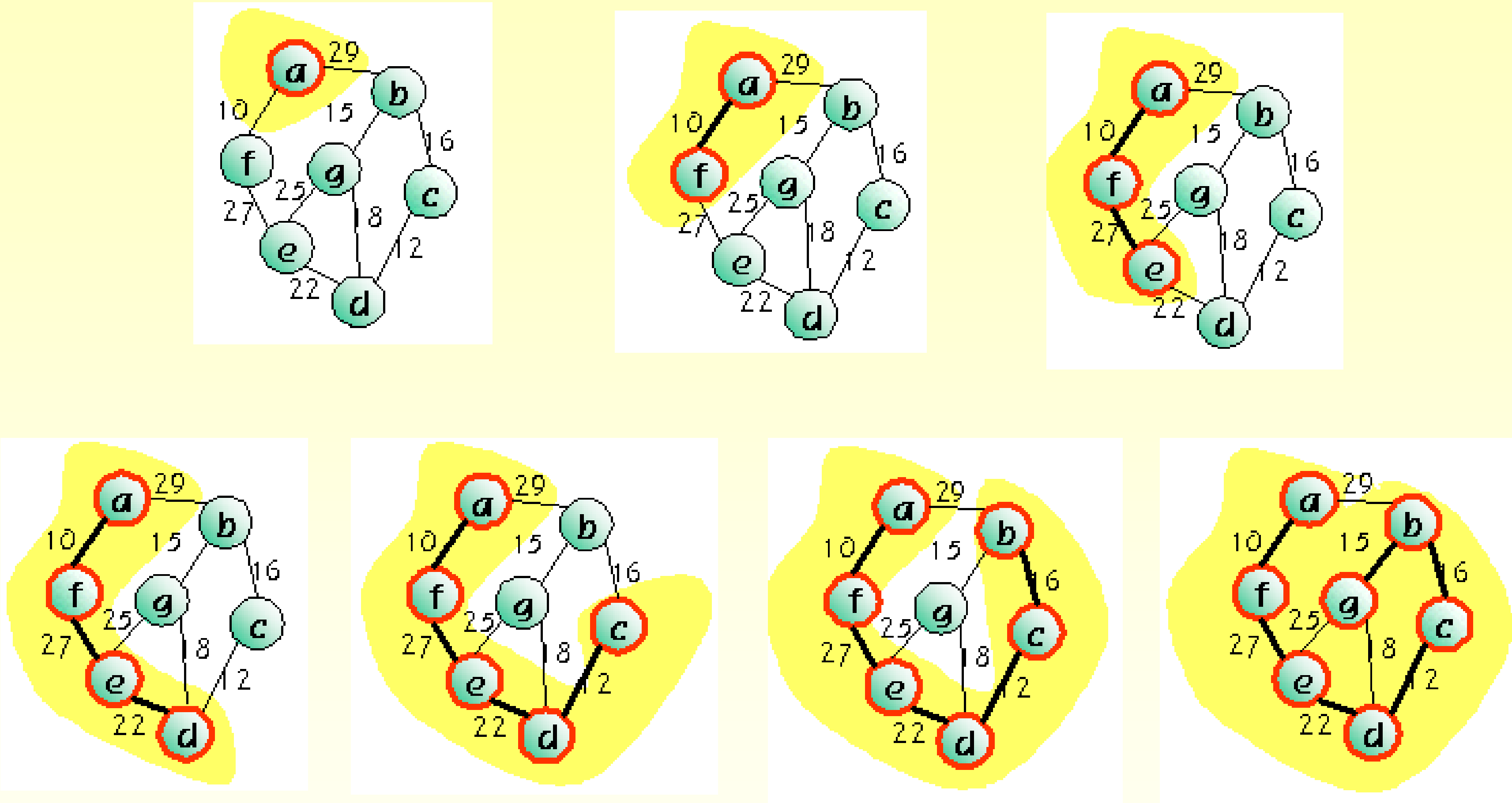
프로그램 11-7 참조

Prim 알고리즘

- 시작 정점에서 출발하여 신장트리 부분집합을 단계적으로 확장하는 방법
  - 첫 단계에서는 시작 정점만을 신장 트리 집합에 포함시켜 놓음
- 만들어진 신장트리 집합에 인접한 정점들 중에서 최소 길이의 간선으로 연결된 정점을 선택하여 트리를 확장 ➔ 반복
- 신장트리의 간선 개수가 n-1개가 될 때까지 반복



Prim의 MST 알고리즘 진행과정



Prim의 MST 알고리즘

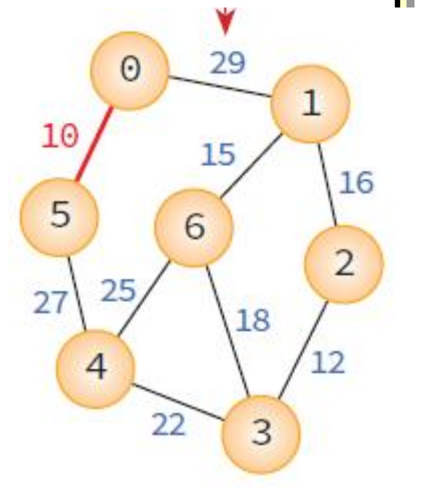
```
// 최소 비용 신장 트리를 구하는 Prim의 알고리즘
// 입력: 네트워크 G=(V, E), s는 시작 정점
// 출력: 최소 비용 신장 트리를 이루는 정점들의 집합
Prim(G, s)

    for each u∈V do
        dist[u] ← ∞
    dist[s] ← 0
    우선 순위큐 Q에 모든 정점을 삽입(우선순위는 dist[])
    for i ← 0 to n-1 do
        u ← delete_min(Q)
        화면에 u를 출력
        for each v∈(u의 인접 정점)
            if( v∈Q and weight[u][v]<dist[v] )
                then dist[v] ← weight[u][v]
```

Prim의 MST 프로그램

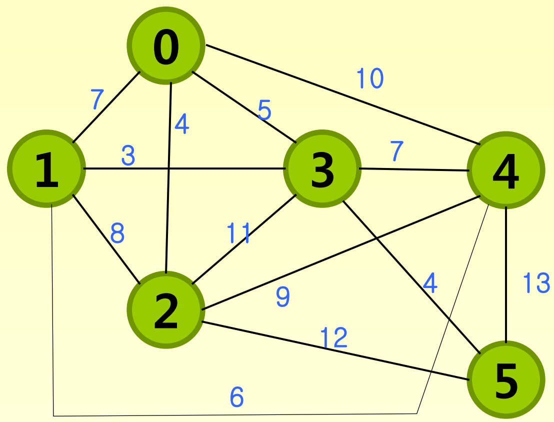
```
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0
#define MAX_VERTICES 100
#define INF 1000L
typedef struct GraphType {
		int n;	int weight[MAX_VERTICES][MAX_VERTICES];
} GraphType;
int selected[MAX_VERTICES];		int distance[MAX_VERTICES];
int get_min_vertex(int n)
{
	int v, i;
	for (i = 0; i <n; i++)
		if (!selected[i]) {  v = i;   break;  }
	for (i = 0; i < n; i++)
		if (!selected[i] && (distance[i] < distance[v])) v = i;
	return (v);
}
```

```
void prim(GraphType* g, int s)  {
     int i, u, v;
     for (u = 0; u<g->n; u++)  distance[u] = INF;
     distance[s] = 0;
     for (i = 0; i<g->n; i++) {
          u = get_min_vertex(g->n);
          selected[u] = TRUE;
          if (distance[u] == INF) return;
          printf("정점 %d 추가\n", u);
          for (v = 0; v<g->n; v++)
               if (g->weight[u][v] != INF)
                    if (!selected[v] && g->weight[u][v]< distance[v])
                         distance[v] = g->weight[u][v];
    }
}
void main() {
  GraphType g = { 7, {{ 0, 29, INF, INF, INF, 10, INF }, { 29,  0, 16, INF, INF, INF, 15 },
                       { INF, 16, 0, 12, INF, INF, INF }, { INF, INF, 12, 0, 22, INF, 18 },
                     { INF, INF, INF, 22, 0, 27, 25 },  { 10, INF, INF, INF, 27, 0, INF },
                     { INF, 15, INF, 18, 25, INF, 0 } } };
   prim(&g, 0);
}
```

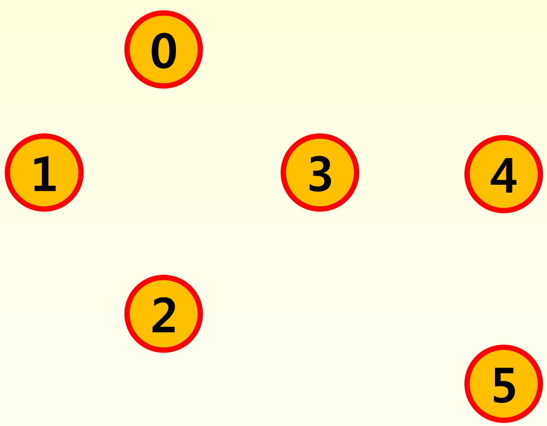
```
정점 0 추가
정점 5 추가
정점 4 추가
정점 3 추가
정점 2 추가
정점 1 추가
정점 6 추가
```

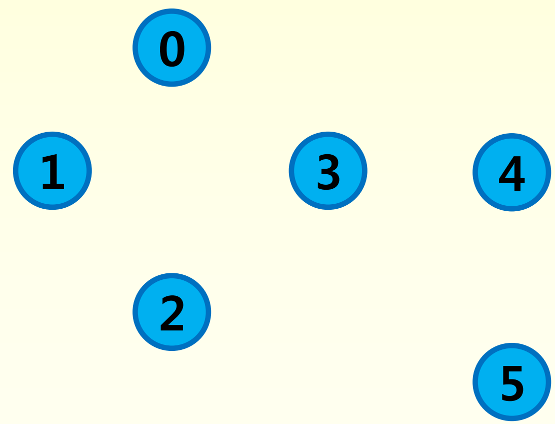


MST 구성과정 비교 : Kruskal vs. Prim



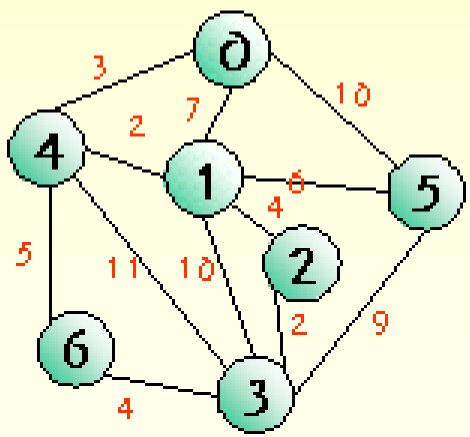
	0	1	2	3	4	5
0	0	7	4	5	10	999
1	7	0	8	3	999	999
2	4	8	0	11	9	12
3	5	3	11	0	7	4
4	10	999	9	7	0	13
5	999	999	12	4	13	0





최단경로 찾기

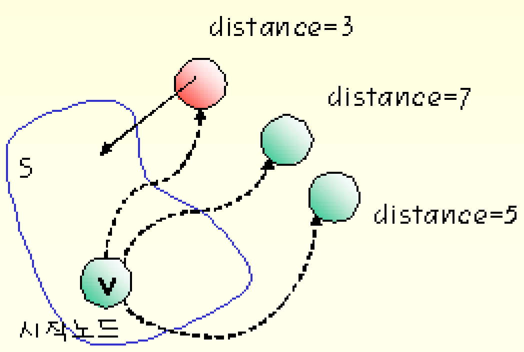
- 최단 경로(shortest path) 문제 적용 : 네트워크에서 정점 i와 정점 j를 연결하는 경로 중에서 간선들의 가중치 합이 최소가 되는 경로를 찾는 문제
- 간선의 가중치 : 비용, 거리, 시간 등의 표현

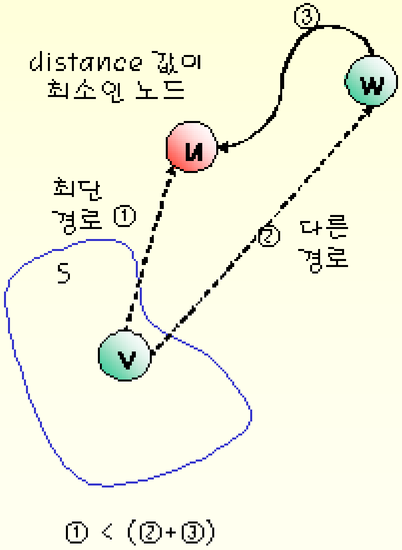


	0	1	2	3	4	5	6
0	0	7	∞	∞	3	10	∞
1	7	0	4	10	2	6	∞
2	∞	4	0	2	∞	∞	∞
3	∞	10	2	0	11	9	4
4	3	2	∞	11	0	∞	5
5	10	6	∞	9	∞	0	∞
6	∞	∞	∞	4	5	∞	0

Dijkstra의 최단경로 알고리즘

- 하나의 시작 정점(출발점)으로부터 모든 다른 정점까지의 최단 경로를 찾는 알고리즘
- 집합 S : 시작 정점 v 부터의 최단경로가 이미 발견된 정점들의 집합
- distance 배열 : 인접행렬을 검사하여 집합 S에서 최단 경로를 가진 정점만을 거쳐서 각 정점까지 가는 최단 경로의 길이(매번 업데이트 됨)
- 매 단계에서 distance 값이 최소인 정점을 S에 추가 ➔ 최단경로 유지





Dijkstra의 최단 경로 알고리즘

- 매 단계에서 새로운 정점이 S에 추가되면 distance값을 갱신한다.

```
distance[w] = min(distance[w],distance[u]+weight[u][w])
```

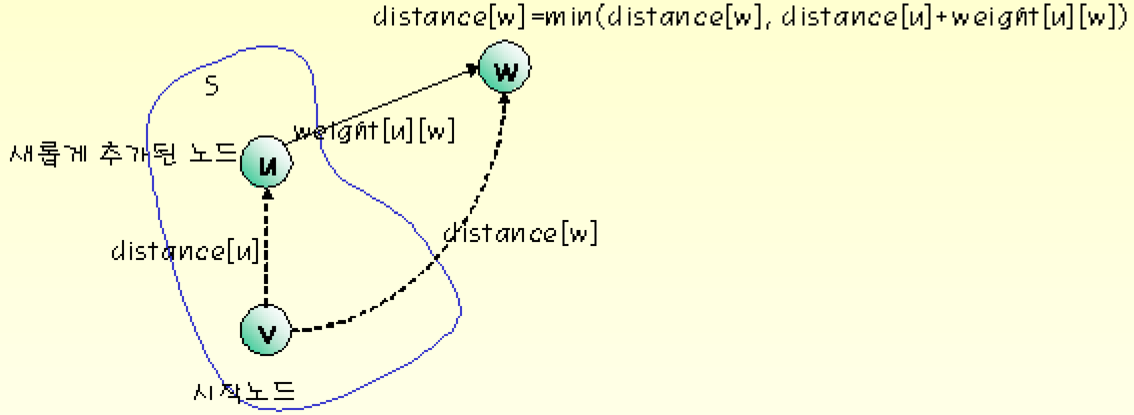
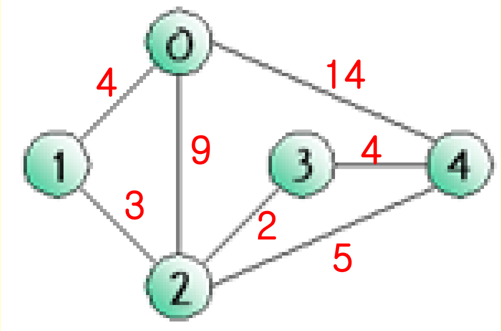


그림 10.34 최단 경로 알고리즘에서의 distance값 갱신

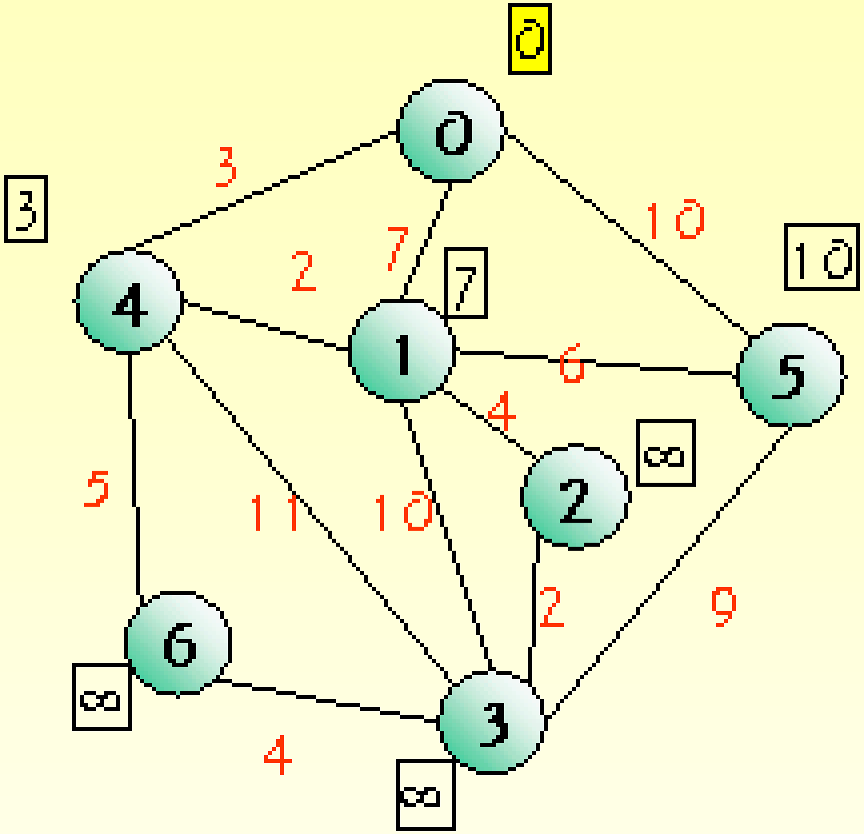
Dijkstra의 최단 경로 알고리즘

// 입력: 가중치 그래프 G(가중치는 양수)
// 출력: distance 배열(v에서 모든 정점까지 최단 거리)

```
shortest_path(G, v)   // v는 출발점
{
  S←{v}
  for (모든 정점 w∈G )
      distance[w]←weight[v][w];
  while (모든 정점이 S에 포함되지 않으면)
  {    u←집합 S에 속하지 않는 정점 중에서 최소 distance 정점;
      S←S∪{u}
      for  (u에 인접하고 S에 있는 각 정점 z)
          if (distance[u]+weight[u][z] <  distance[z])
                 distance[z] ← distance[u]+weight[u][z];  // 더 짧은 경로로 대체
  }
}
```

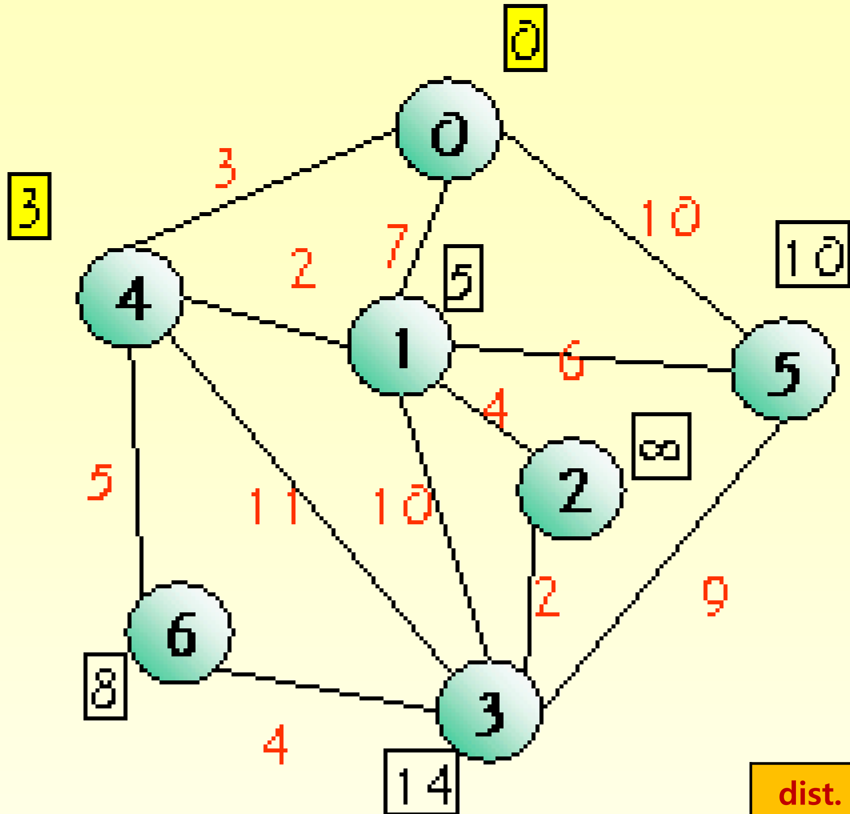


dist.	v_0	v_1	v_2	v_3	v_4
V_0 ~	0	4	9	999	14



S={0}

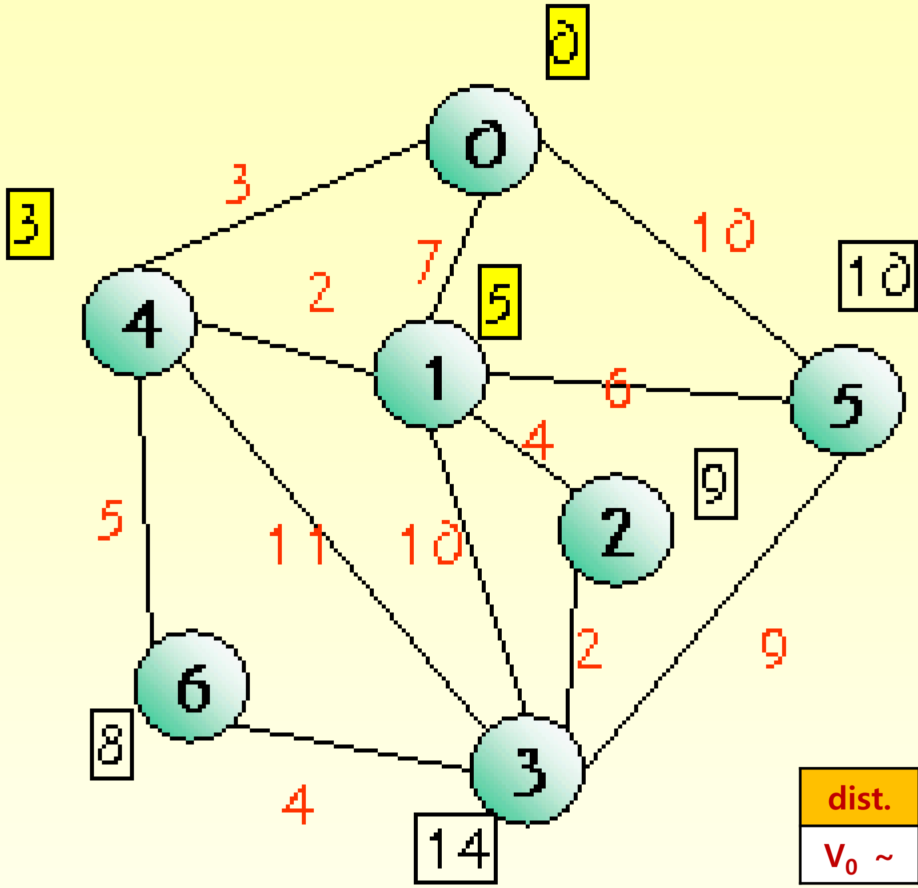
	0	1	2	3	4	5	6
distance[]=	0	7	∞	∞	3	10	∞



dist.	v_0	v_1	v_2	v_3	v_4	v_5	v_6
V_0 ~	0	7	999	999	3	10	999

S={0, 4}

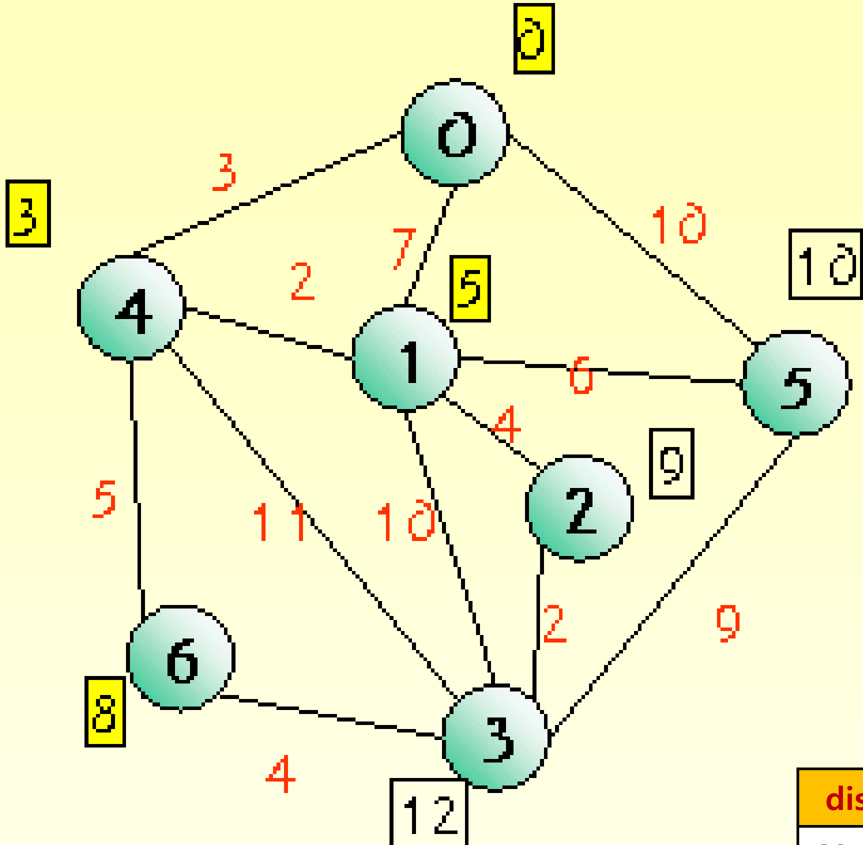
	0	1	2	3	4	5	6
distance[]=	0	5	∞	14	3	10	8



dist.	v_0	v_1	v_2	v_3	v_4	v_5	v_6
V_0 ~	0	5	999	14	3	10	8

S={0, 4, 1}

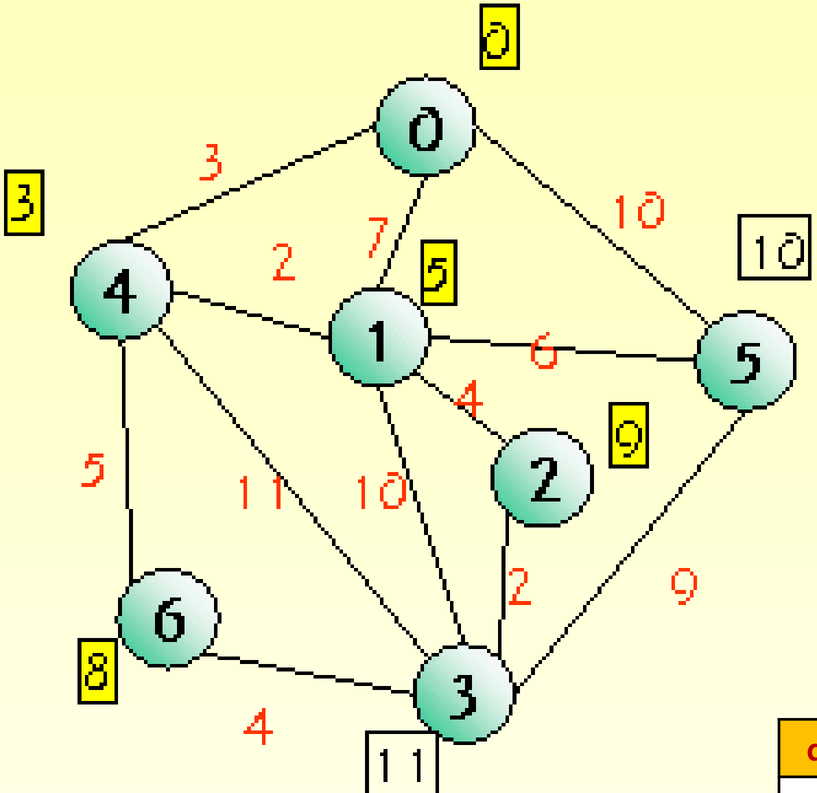
	0	1	2	3	4	5	6
distance[]=	0	5	9	14	3	10	8



dist.	v_0	v_1	v_2	v_3	v_4	v_5	v_6
V_0 ~	0	5	9	14	3	10	8

S={0, 4, 1, 6}

	0	1	2	3	4	5	6
distance[]=	0	5	9	12	3	10	8

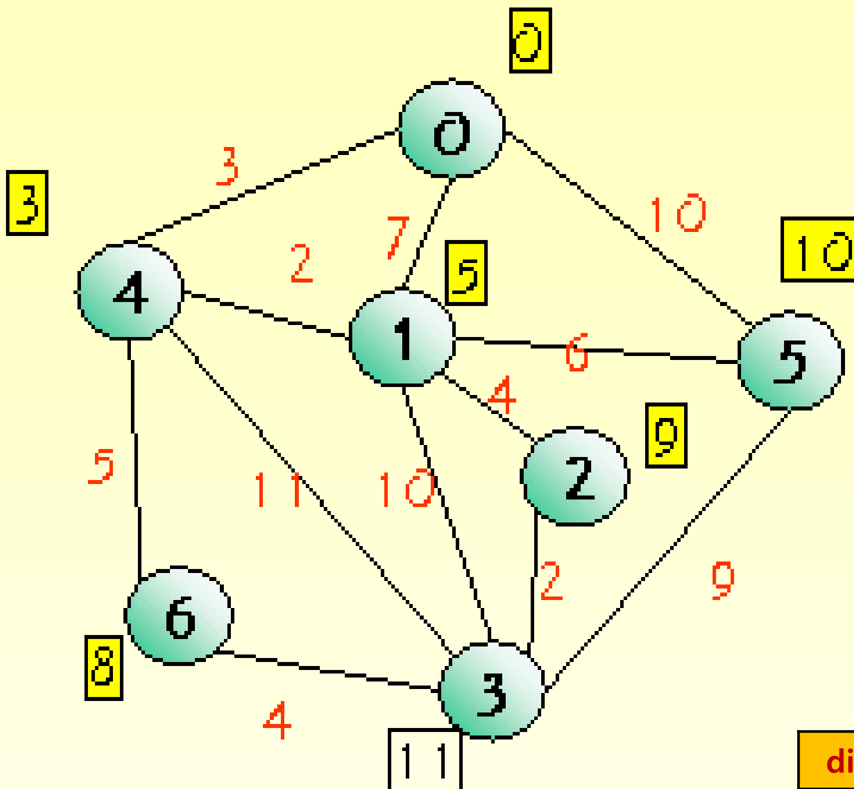


dist.	v_0	v_1	v_2	v_3	v_4	v_5	v_6
V_0 ~	0	5	9	12	3	10	8

S={0, 4, 1, 6, 2}

distance[]=

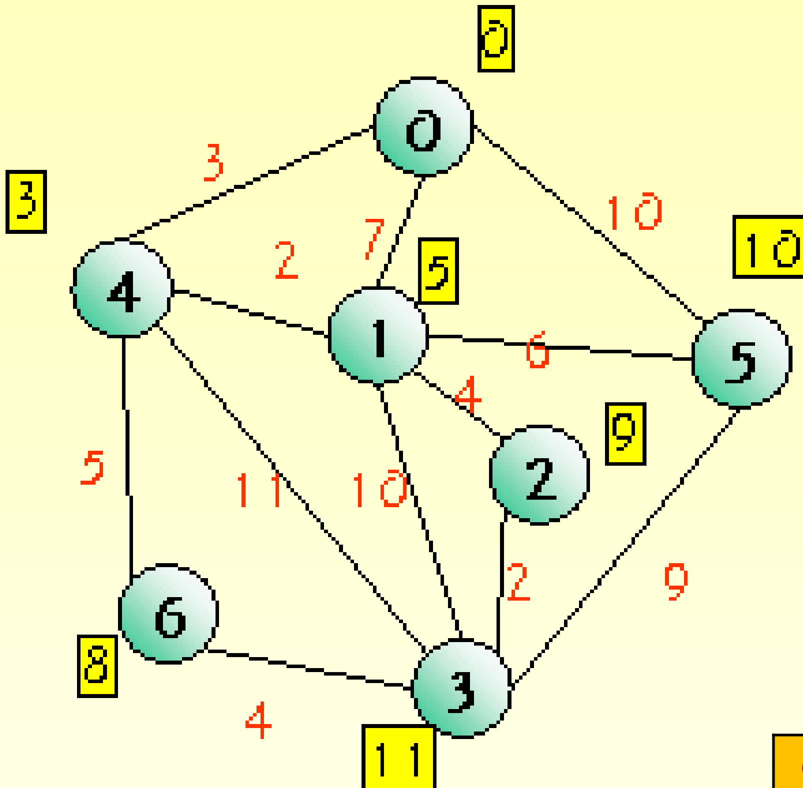
0	1	2	3	4	5	6
0	5	9	11	3	10	8



dist.	v_0	v_1	v_2	v_3	v_4	v_5	v_6
V_0 ~	0	5	9	11	3	10	8

S={0, 4, 1, 6, 2, 5}

	0	1	2	3	4	5	6
distance[]=	0	5	9	11	3	10	8



dist.	v_0	v_1	v_2	v_3	v_4	v_5	v_6
V_0 ~	0	5	9	11	3	10	8

S={0, 4, 1, 6, 2, 5, 3}

distance[]=

0	1	2	3	4	5	6
0	5	9	11	3	10	8