5장. 그래프

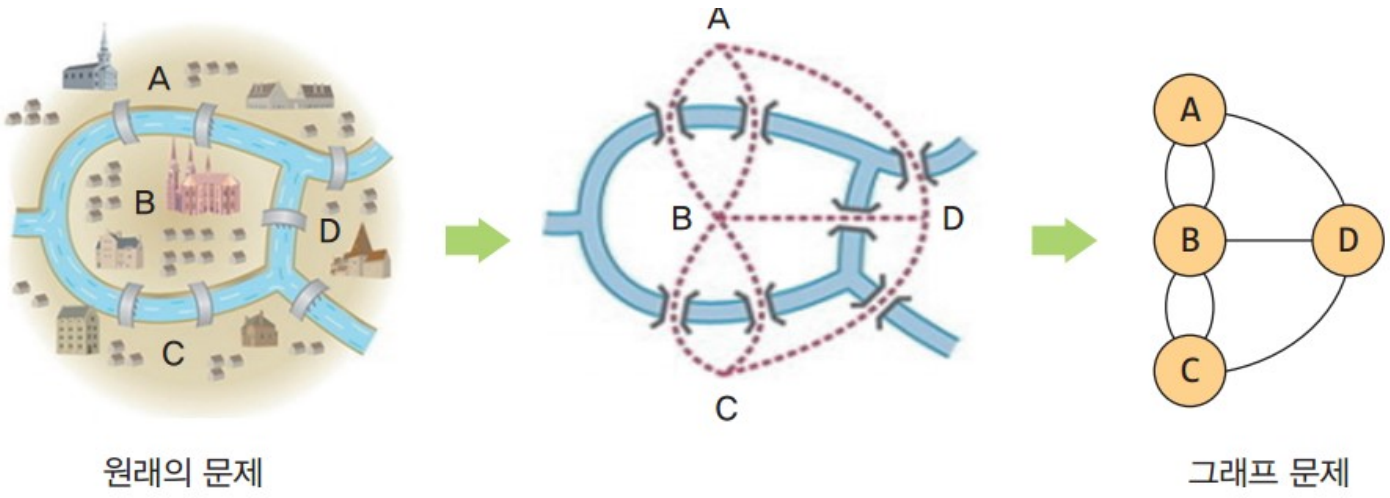
- 그래프의 의미
- 그래프의 표현
- 그래프 순회
- 신장 트리
- 그래프의 응용

학습목표

- 그래프의 의미
- 그래프의 표현
- 그래프 순회
- 신장 트리
- 그래프의 응용

그래프란?

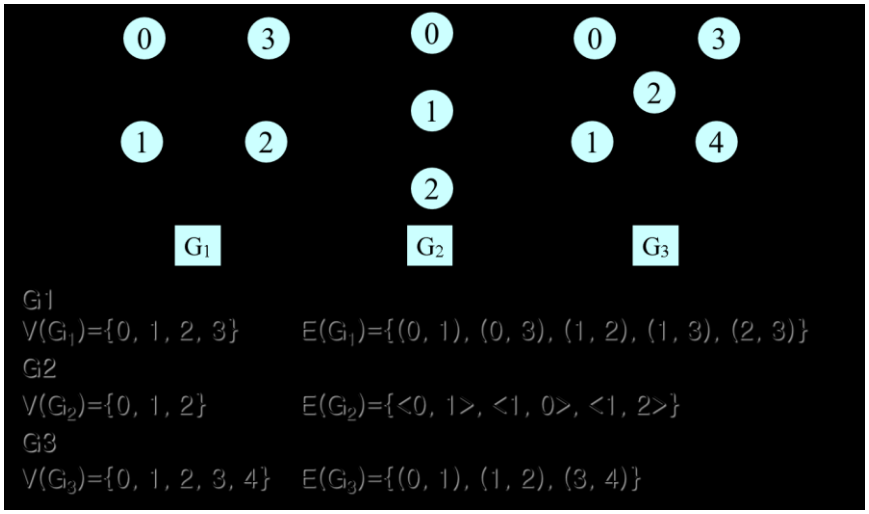
- **정보를 표현하는 점과 자료들의 관계를 표현한 연결선으로 정보의 표현을 구성한 비선형 자료구조.**
  - G = (V, E) 로 표현 //G : Graph. V : 정점(vertices), 노드(node), E : 간선(edge) , 링크(link):

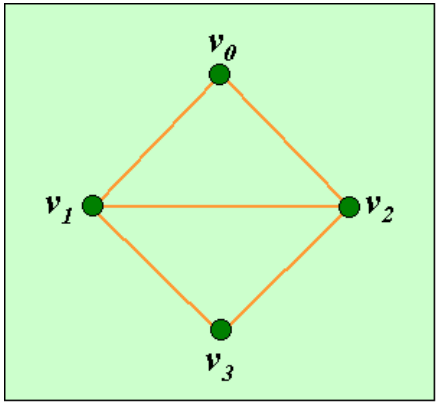


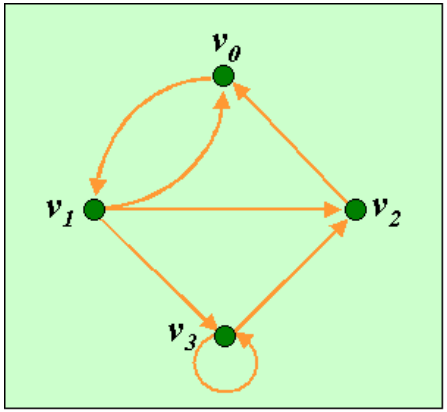
원래의 문제 그래프 문제

- 오일러(Euler)에 의해 그래프의 이론 정립.
  - Koenigsberg 다리 문제에서 시작
- 컴퓨터 과학 분야에서 가장 많이 사용되는 수학적 모델 중의 하나

- G= (V, E)
  - V : 공백이 아닌 노드 (node) 또는 정점(vertex)의 유한집합 // 정점은 정보를 표현
    - V(G) = {v_0, v_1 , … , v_{n-1}} //그래프에서 정점만 추출
  - E : 상이한 두 정점을 잇는 간선(edge, arc)의 유한집합 //연결선은 관계를 표현
    - E(G) = {e_0, e_1, … , e_{n-1}} //그래프에서 연결선만 추출
    - e_i 각각은 순서쌍(ordered pair)으로 표현 :
      - » v_0, v_2 를 연결하는 e_1이 무향선이면 (v_0, v_2)로, 유향선이면 <v_2, v_0>로 표시.







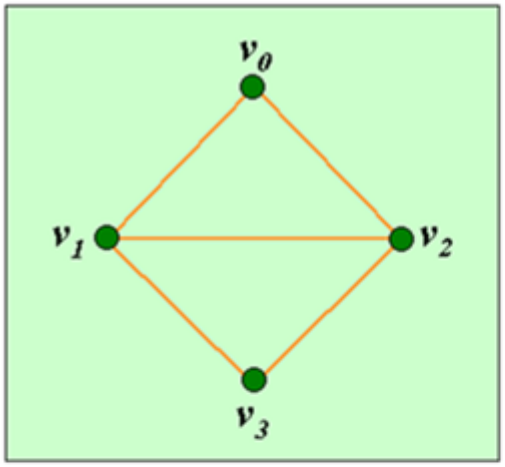
*. 방향성/무방향성의 표현 차이를 보세요

V : {v0, v1, v2, v3}로 동일

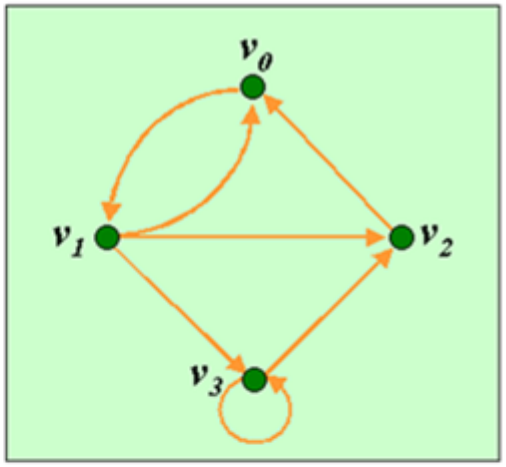
E : {(v0,v1),(v0,v2),(v1,v2),(v1,v3),(v2,v3)} // 왼쪽

E : {<v0, v1>,<v1, v0>,<v1, v2>,<v1, v3>,<v2, v0>,<v3, v2>,<v3, v3>} //오른쪽

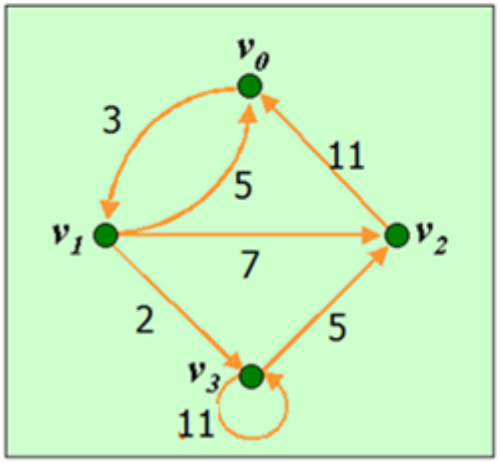
그래프 종류 및 관련 용어



무향그래프/단순그래프



유향 그래프/다중그래프



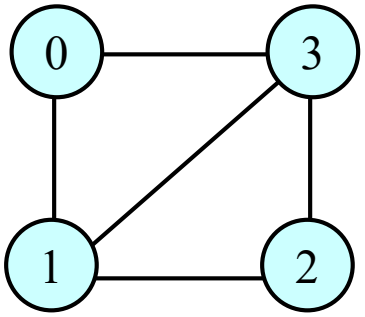
가중 그래프

- **용어**
  - 정점(Vertex, node)
  - 연결선(Edge, arc, link, 간선)
  - 무향 그래프(undirected graph) (v_0, v_1)….
  - 유향 그래프(directed graph) $<v_0, v_1>$, $<v_1, v_0>$….
  - 자가루프(self loop)
  - 사이클(Cycle)
  - 단순 그래프(simple graph, 자기 루프를 허용하지 않고, 두 정점 사이에 하나의 간선만 존재하는 그래프)
  - 다중 그래프(multigraph): 정점 사이에 복수 간선 허용그래프
  - 가중 그래프(Weighted Graph)

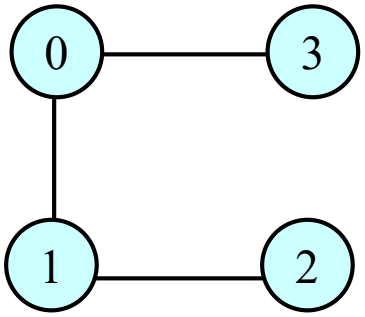
Q. 가중그래프에서 v_3 자가루프 비용이 1이라면 인접행렬은 어떻게 바뀌나?

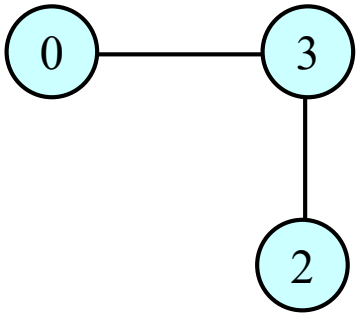
- 부분 그래프(부그래프, sub graph)
  - 어떤 그래프의 일부분으로 구성된 그래프
  - V(G')⊆V(G)이고, E(G')⊆E(G)인 그래프.

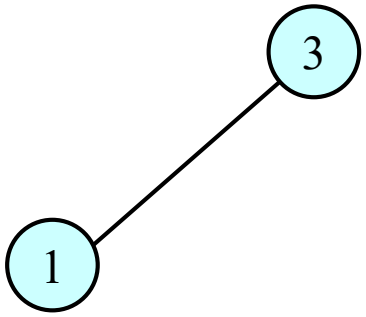
// G'는 그래프 G의 부분 그래프



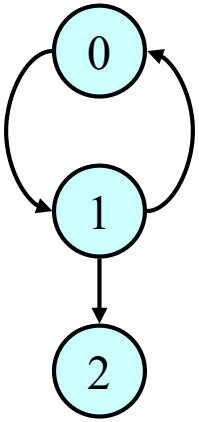
G_1



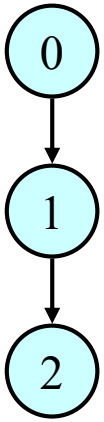


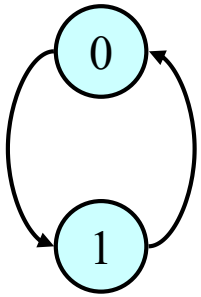


(부분 그래프들)



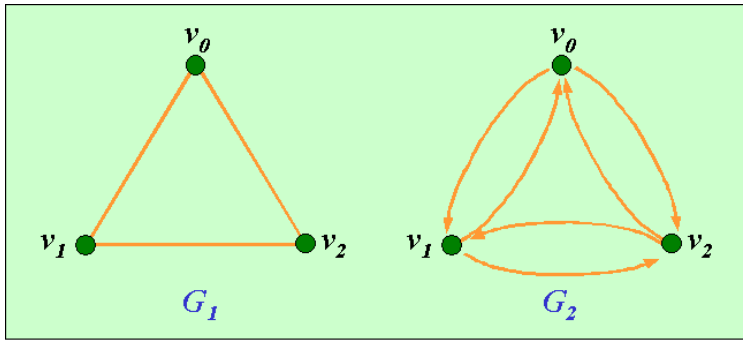
G_2





(부분 그래프들)

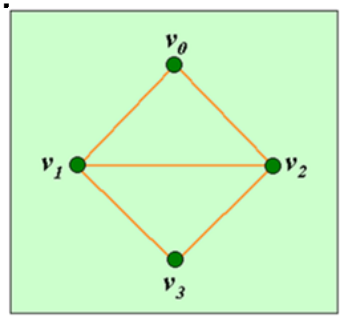
- 완전 그래프(complete graph) :모든 정점에서 모든 정점으로 연결선을 갖는 그래프)
- 인접(adjacent) 연결된 두 정점 관계를 이르는 말.
  - (vi, vj) 정점 vi와 vj는 인접.
  - <vi, vj> vj는 vi에서 vj 방향으로 인접.
- 부착/부속(incident on) 간선의 소속을 밝히는 말.
  - (vi, vj)는 vi 또는 vj에 부착.
  - <vi, vj>는 vi에 부착



- 차수(degree)
  - 무향 그래프에서의 차수 : 인접 정점의 수 또는 무방향 그래프에서 그 정점에 부속된 간선의 수.
  - 유향 그래프에서의 차수 :
    - 내향 차수(indegree) – 자신으로 향한 화살표 수.
    - 외향 차수(outdegree) - 밖으로 향한 화살표 수.
- 선행자(predecessor) – 유향 그래프에서 정점 v로 들어오는 간선의 정점.
  - <vi, v>의 경우 vi.
- 후행자(successor) – 유향 그래프에서 정점 v에서 나가는 간선의 정점.
  - <v, vk>의 경우 vk.

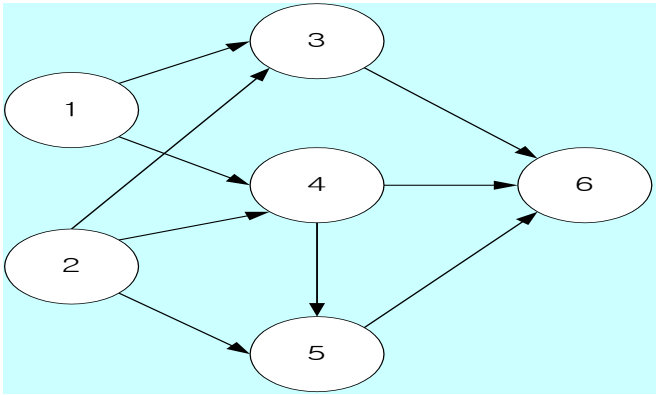
- 경로(Path) - 주어진 정점에서 목표 정점에 이르는 정점의 순서열.
  - 무향그래프 v0에서 v2경로 : v0, v2 | v0, v1, v2 | v0, v1, v3, v2

  cf) 유향 그래프에서도 해볼 것.



- 경로의 길이(path length) - 경로를 구성하는 간선의 수.

  예) (v0, v2) : 1 | (v0, v1) (v1, v2) : 2 | (v0, v1) (v1, v3) (v3, v2) : 3
- 단순 경로(simple path) - 처음과 마지막의 비교를 제외한 경로상의 정점이 모두 다른 경로. (시작과 끝은 같은 것을 허용)
  - 유향 그래프에서 v0, v1, v2, v0는 단순경로이나 v0, v1, v0, v1은 단순경로가 아님(v1이 중복).
- 연결 그래프(connected graph)-모든 정점들 사이에 경로가 존재하는 그래프
- 사이클(Cycle)
  - 첫 번째 정점과 마지막 정점이 동일한 단순 경로.
  - 사이클은 단순 경로 중 특별한 형태 중 하나.
- DAG(directed acyclic graph) –
  - 방향 그래프에서 사이클이 없는 그래프

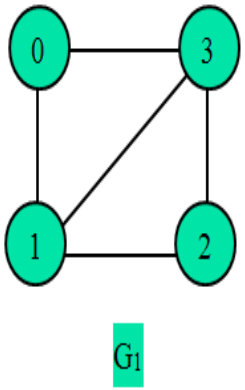


그래프의 표현

인접 행렬(adjacency matrix)로 표현

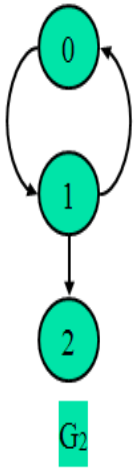
- 정점 수와 같은 크기의 정방 행렬(Matrix)로 표현. (행이 출발점 열이 도착점.)
- 가중치 그래프와 비가중치 그래프 구별
  - 비가중치 그래프
    - » 간선 (i, j)가 있으면: M[i][j] = 1, 또는 true 그렇지 않으면: M[i][j] = 0, 또는 false
  - 가중치 그래프의 경우
    - » 가중치 범위 참조하여 표현
  - 무향 그래프 경우 대칭 구조.
    - » (i, j)가 있으면 (j, i)도 존재.

<Q> 유향 그래프 경우도 같을까 ?



	[0]	[1]	[2]	[3]
[0]	0	1	0	1
[1]	1	0	1	1
[2]	0	1	0	1
[3]	1	1	1	0

a[4, 4]



	[0]	[1]	[2]
[0]	0	1	0
[1]	1	0	1
[2]	0	0	0

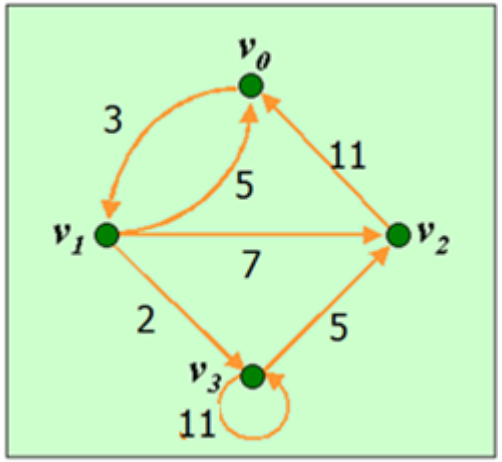
a[3, 3]

```
int G1 [4] [4] = {
                {0,1,0,1},
                {1,0,1,1},
                {0,1,0,1},
                {1,1,1,0}
};
int G2 [3] [3] = {
                {0,1,0},
                {1,0,1},
                {0,0,0}
};
```

그래프의 표현

❖ 가중치 그래프

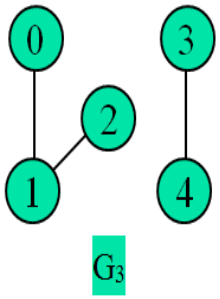
인접행렬



	V_0	V_1	V_2	V_3
V_0	0	3	∞	∞
V_1	5	0	7	2
V_2	11	∞	0	∞
V_3	∞	∞	5	11

Q. 왼쪽 가중치 그래프의 인접행렬 표현에서 간선이 없는 곳의 값은 얼마로 하면 좋을까?
또 아래 그래프를 인접행렬로 표현하시오.

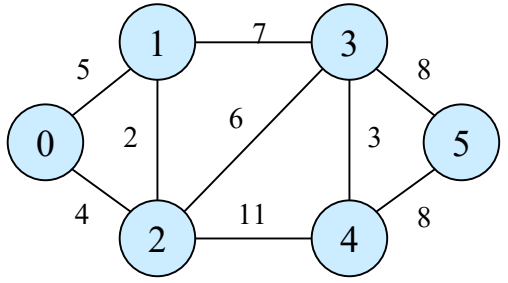
<참고 : 비가중치 그래프>



G_3

	[0]	[1]	[2]	[3]	[4]
[0]	0	1	0	0	0
[1]	1	0	1	0	0
[2]	0	1	0	0	0
[3]	0	0	0	0	1
[4]	0	0	0	1	0

a[5, 5]

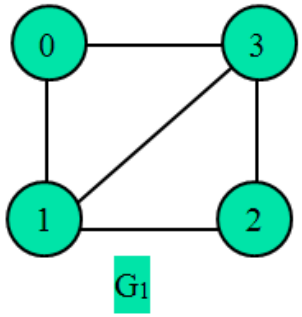


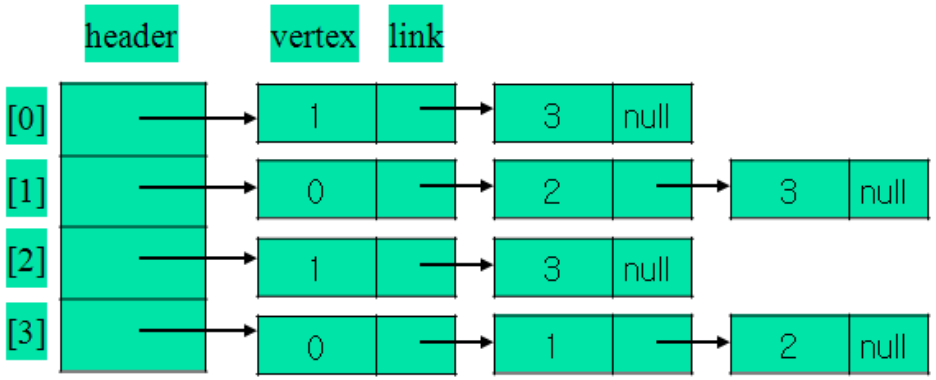
G=(V, E)

그래프의 표현

인접 리스트(adjacency list)로 표현

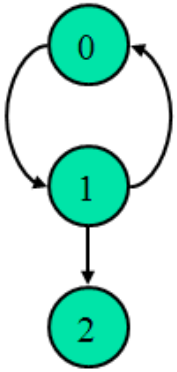
- 정점 만큼의 포인터(배열)를 이용하여 연결리스트로 표현

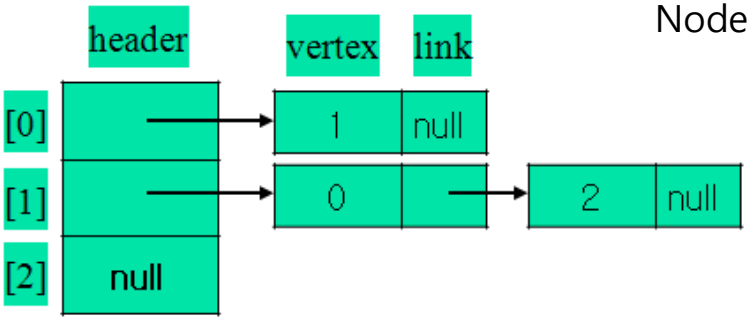




(a) G_1에 대한 인접 리스트

*. 가중치 그래프의 경우
Node의 data 부분을 (vertex/weight)로 표현.





(b) G_2에 대한 인접 리스트

❖ Graph ADT

데이터: 정점과 간선의 집합

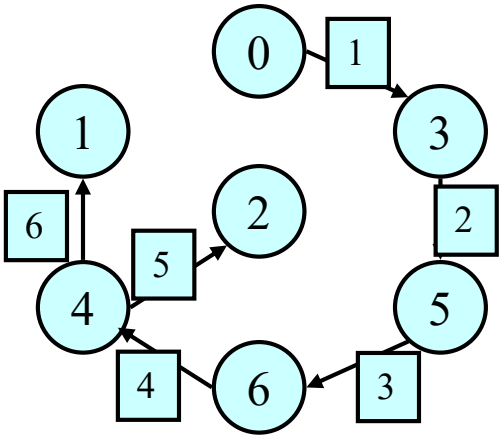
연산

- isEmpty(): 그래프가 공백 상태인지 확인한다.
- countVertex(): 정점의 수를 반환한다.
- countEdge(): 간선의 수를 반환한다.
- getEdge(u,v): 정점 u에서 정점 v로 연결된 간선을 반환한다.
- degree(v): 정점 v의 차수를 반환한다.
- adjacent(v): 정점 v에 인접한 모든 정점의 집합을 반환한다.
- insertVertex(v): 그래프에 정점 v를 삽입한다.
- insertEdge(u,v): 그래프에 간선 (u,v)를 삽입한다.
- deleteVertex(v): 그래프의 정점 v를 삭제한다.
- deleteEdge(u,v): 그래프의 간선 (u,v)를 삭제한다.

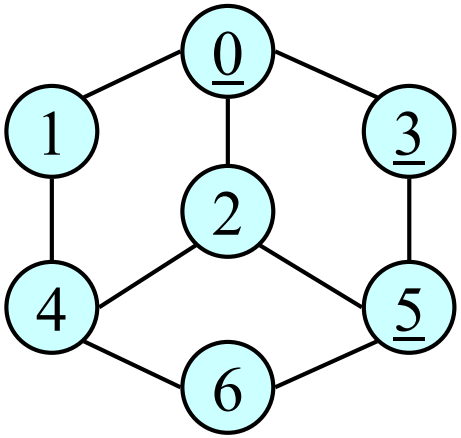
그래프의 순회

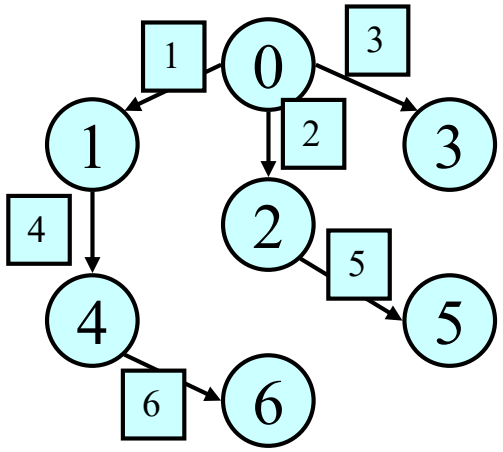
- **순회 :** 주어진 정점을 출발하여 체계적으로 그래프의 모든 정점들을 방문하는 것.
  - 깊이 우선 탐색 : 스택을 이용.
    - 준비작업으로 스택과 방문 표시용 배열을 준비하고, 시작 정점을 스택에 넣고 시작.
    - 스택이 공백이 아닌 동안 아래를 반복 수행
    - 1) 스택 상단의 정점 i를 꺼내 이미 방문했으면 버리고, 방문하지 않았으면 방문한다.
    - 2) 정점 i 인접 정점 중 아직 방문하지 않은 정점을 스택에 저장(순서 고려).
  - 너비 우선 탐색 : 큐를 이용. 위 깊이 우선 알고리즘과 행태는 동일.

<공통> 큰 번호부터 삽입 시 다른 순회 순서가 형성됨.



깊이 우선 탐색 경로
(단, 큰 번호 우선)
방문순서 : 0-3-5-6-4-2-1

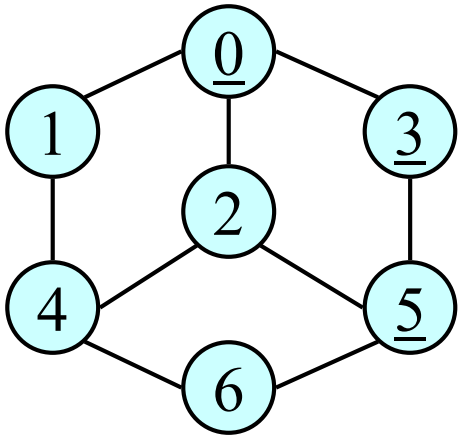




너비 우선 탐색 경로
(단, 작은 번호 우선)
방문순서 : 0-1-2-3-4-5-6

깊이 우선 탐색

❖ 그래프의 인접행렬 표현



	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	0	0	1	0	0
2	1	0	0	0	1	1	0
3	1	0	0	0	0	1	0
4	0	1	1	0	0	0	1
5	0	0	1	1	0	0	1
6	0	0	0	0	1	1	0

```
#define    NON     7    /* Num of Node 그래프를 구성하고 있는 노드 수 정의*/

int G1 [NON] [NON] = {        /* 그래프 구성 */
            {0,1,1,1,0,0,0},
            {1,0,0,0,1,0,0},
            {1,0,0,0,1,1,0},
            {1,0,0,0,0,1,0},
            {0,1,1,0,0,0,1},
            {0,0,1,1,0,0,1},
            {0,0,0,0,1,1,0}};
```

출발점을 5로 하고
작은 수를
우선 탐색하는 조건이라면...

5->2->0->1->4->6->3

필요 자료구조와 기능

<< 그래프 >>

```
#define     NON      7

 int G1 [NON] [NON] = {
              {0,1,1,1,0,0,0},
              {1,0,0,0,1,0,0},
              {1,0,0,0,1,1,0},
              {1,0,0,0,0,1,0},
              {0,1,1,0,0,0,1},
              {0,0,1,1,0,0,1},
              {0,0,0,0,1,1,0}};
```

<< 방문 유무 표시 >>

```
int visited[NON] = {0,0,0,0,0,0,0};
//방문 안된 상태
```

<< 스택 구성>>

```
struct   Stack {
    int   data[NON*NON];
    int   top;
};
```

<< 스택에 출발점 넣기 >>

```
stack.top = 0;
stack.data[stack.top] = 5;
```

```
int haveNodeToVisit( int visited[NON] )/*방문할 노드가 남았는지 확인*/
{
    int i;
    for(i=0; i<NON; i++)
        if (visited[i] == 0)
            return 1;   /*방문할 노드가 남았음을 알림*/
    return 0;           /*모두 방문했음을 알림*/
}
```

```
int popStack( struct Stack * stack )       /*스택에서 하나를 추출함*/
{
    int node;
    if (stack->top < 0)
        node = -1;     //데이타가 없음을 표시. 즉 스택이 공백(empty) 상태임을 표시
    else {
        node = stack->data[stack->top];
        (stack->top)--;
    }
    return node;
}
```

```
void pushStack( struct Stack * stack, int i )      /*자식 노드를 스택에 저장*/
{
    (stack->top)++;
    stack->data[ stack->top ] = i;
}
```

```
int isNotVisited( int currentNode, int visited[NON] )      /*방문 대상인지 확인*/
{
    if ( visited[currentNode] )
        return 0;    /*방문했음을 알림*/
    else
        return 1;    /*방문안했음을 알림*/
}
```

```
void checkVisited( int currentNode, int visited[NON] )        /*현재 노드 방문 처리*/
{
    visited[currentNode] = 1;  /*방문했음을 표시*/
    printf("₩n %d ", currentNode);
}
```

```
void printStack( struct Stack stack )
{
    int i;
    printf("₩n₩n[STACK] : ");
    for(i=0; i<=stack.top; i++) {
        printf("%3d", stack.data[i]);
    }
}
```

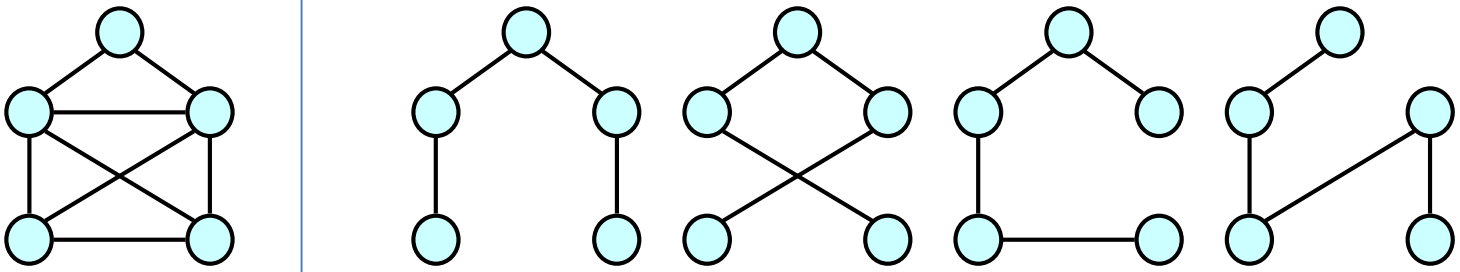
깊이 우선 탐색 수행

```
/*스택에 출발점 넣기*/
stack.top = 0;
stack.data[stack.top] = 5;

/*아래 행동을 모두 방문하거나 스택에 자료가 없을 때까지 반복*/
/* 1) 스택에서 자료를 꺼내 방문하지 안했으면 방문처리하고, 2) 방문하지 않은 자식들을 막내부터 스택에 넣는다. */
while ( (stack.top >= 0) && haveNodeToVisit( visited ) ) {
    currentNode = popStack( &stack );  /*스택에서 방문할 노드를 얻어냄*/
    if (currentNode < 0 ){              /*이 조건이 참이라면 프로그램에 에러가 있다는 증거*/
        printf("₩n₩n error #01₩n₩n");
        exit( 1);
    }
    if ( isNotVisited( currentNode, visited ) ) {
        checkVisited( currentNode, visited ); /*현재 노드 방문 처리*/
        /*방문하지 않은 자식들을 막내부터 스택에 넣는다.*/
        for(int i = NON - 1; i >= 0; i--) {
            if (( G1[i][currentNode] == 1 ) && (!visited[i]) ) {
                pushStack( &stack, i );
            }
        }
    }
}
```

```
[STACK] :   5
 5

[STACK] :   6  3  2
 2

[STACK] :   6  3  4  0
 0

[STACK] :   6  3  4  3  1
 1

[STACK] :   6  3  4  3  4
 4

[STACK] :   6  3  4  3  6
 6

[STACK] :   6  3  4  3
 3
```

신장 트리(Spanning Tree)

- **그래프 G에서 E(G)의 일부와 V(G)의 모든 정점들로 구성하되, 모든 정점이 연결되어 있으며, 사이클이 없는 그래프**
  - 주어진 그래프 G에 대한 신장트리는 유일하지 않음.



  - Q : 너비 우선, 깊이 우선 탐색의 결과도 신장트리 일까?

그래프의 응용 : 최소 비용 신장트리

- **최소비용 신장 트리**
  - **가중치 그래프에서** 비용이 가장 적게 드는 간선들을 이용하여 구성된 신장 트리.

- **최소 비용 신장 트리를 만드는 방법**
  - Kruskal 알고리즘
  - Prim 알고리즘
  - Sollin 알고리즘

Kruskal 알고리즘

- 모든 간선을 비용이 가장 작은 간선부터 차례로 선택하여, 이미 방문한 간선들과 사이클을 형성하면 버리고, 그렇지 않으면 신장 트리 형성에 이용.

< 문제 그래프의 인접 행렬 표현 >

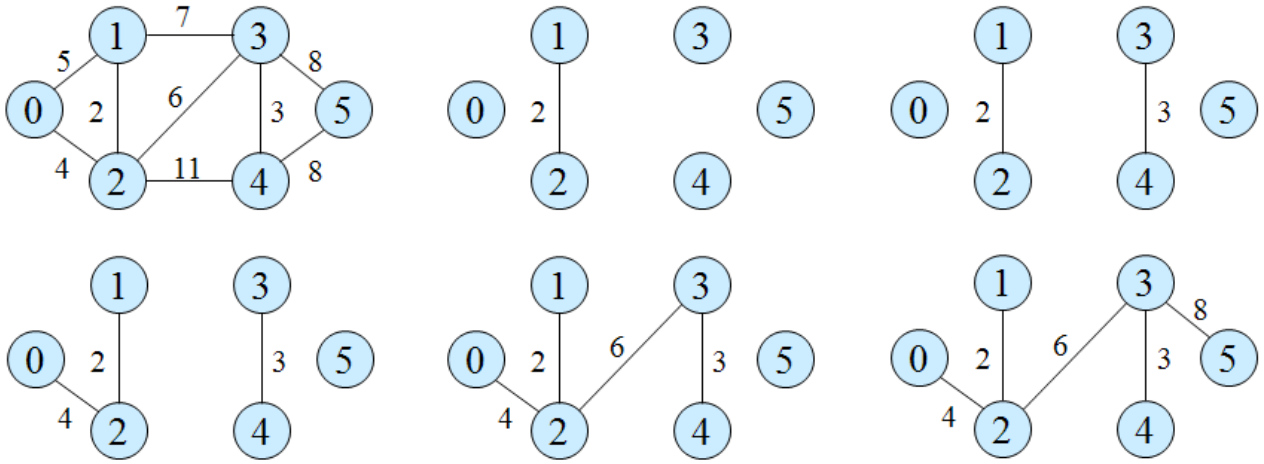
0	**5**	**4**	**∞**	**∞**	**∞**
5	0	2	7	∞	∞
4	2	0	6	11	
∞	7	6	0	3	8
∞	∞	11	3	0	8
∞	∞	∞	8	8	0

간선	(1,2)	(3,4)	(0,2)	(0,1)	(2,3)	(1,3)	(3,5)	(4,5)	(2,4)
비용	2	3	4	5	6	7	8	8	11

간선	(1,2)	(3,4)	(0,2)	(0,1)	(2,3)	(1,3)	(3,5)	(4,5)	(2,4)
선택	O	O	O	X	O	X	O		

사이클 형성

완성

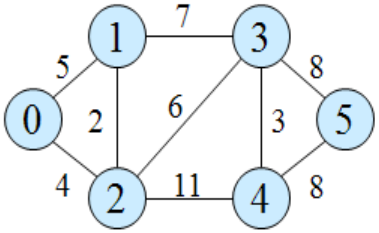


Kruskal 알고리즘 수행 단계

Prim 알고리즘

- 주어진 출발 정점에서 시작하여 하나의 신장 트리로 성장시키되, 현재 선택된 노드와 간선으로 구성된 부분 그래프에 사이클을 형성하지 않는 가장 적은 비용의 간선을 선택해 나감으로 신장 트리를 구성.

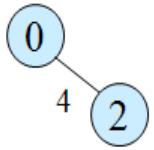
E(T) : 신장 트리의 간선의 집합
V(T) : 구성 신장 트리 정점



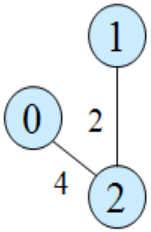
G=(V, E)

(a)

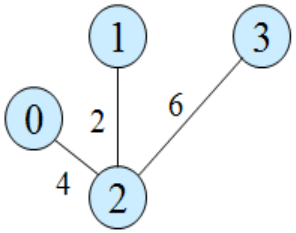
E(T)={ }
V(T)={0}
(b)



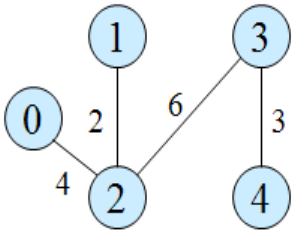
E(T)={(0,2)}
V(T)={0,2}
(c)



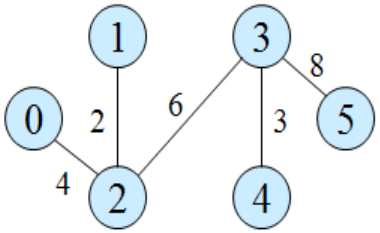
E(T)={(0,2), (2,1)}
V(T)={0,2,1}
(d)



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



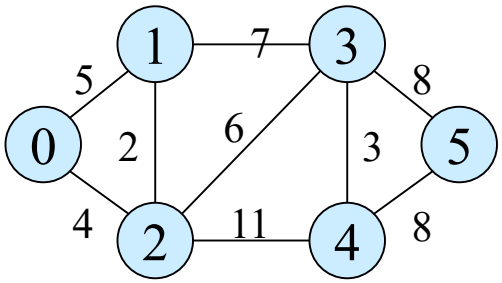
E(T)={(0,2), (2,1), (2,3), (3,4)}
V(T)={0,2,1,3,4}
(f)



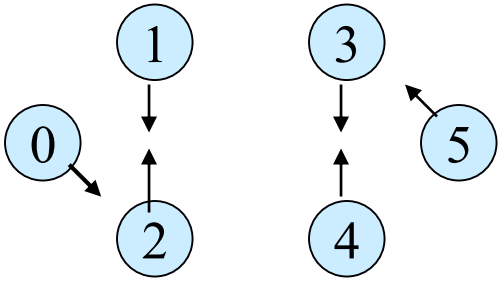
E(T)={(0,2), (2,1), (2,3), (3,4), (3,5)}
V(T)={0,2,1,3,4,5}
(g)

Sollin 알고리즘

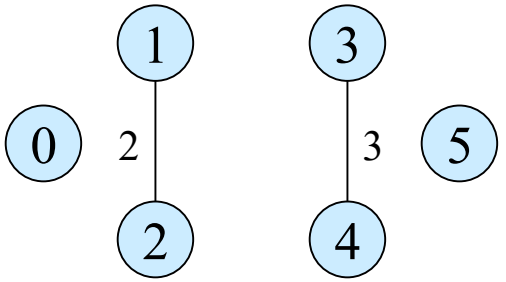
- 초기에 간선이 없다고 생각하고 각 단계에서 부분 그래프의 그룹마다 부착(incident on)된 간선 중 사이클을 형성하지 않는 가장 적은 비용의 간선을 선택하며, 통합해 나감으로, 최소 비용 신장 트리를 구성



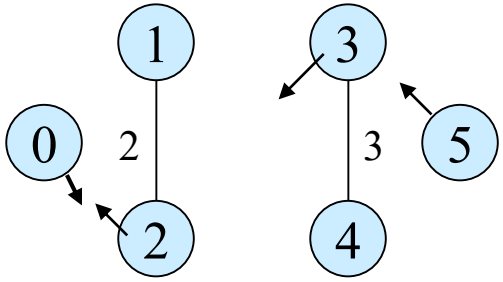
G=(V, E)



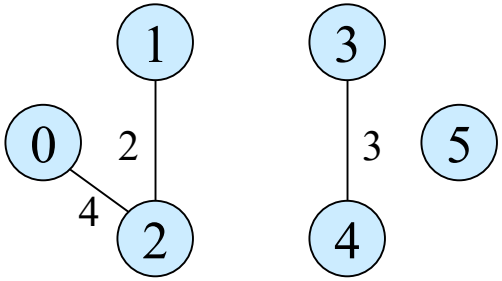
(a)



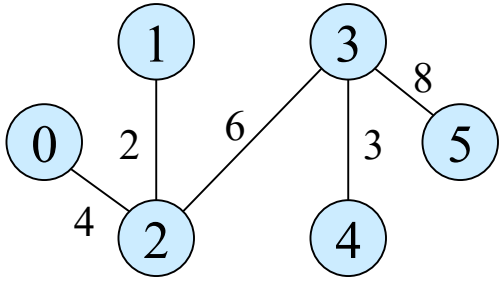
(b)



(c)



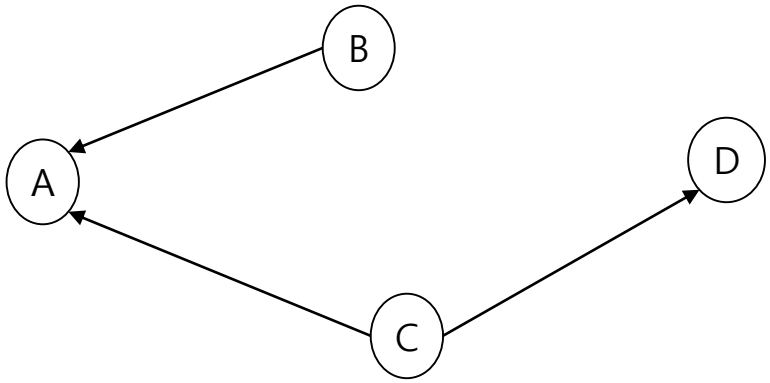
(d)



(-)

위상 정렬(Topological Sort)

- **부분 순서 관계가 있는 원소들에 대해, 정렬의 결과가 그 순서 관계를 지키면서 정렬하는 방법.**
  - V(G)={A, B, C, D} E(G)={<B, A>, <C, D>, <C, A>} 일 때 위상 정렬.



	A	B	C	D
A	0	0	0	0
B	1	0	0	0
C	1	0	0	1
D	0	0	0	0

<참고> 행렬 값 0, 1은
관계가 있고 없음을 표현한다.

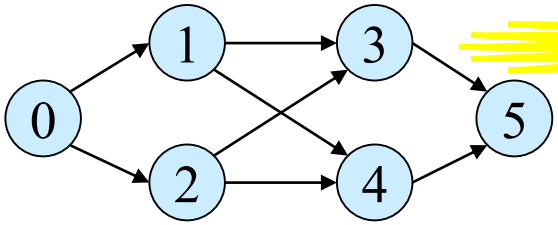
  - 결과 : (B, C, A, D), (B, C, D, A),
    (C, B, A, D), (C, B, D, A), (C, D, B, A)

//2 줄로 표현한 이유가 무엇일까?

위상정렬 알고리즘

AOV(activity on vertex) network
작업간의 선후 관계를 나타내는 방향 그래프
정점 : 작업을 표시
간선 : 작업들 간의 선후 관계 표시

```
topologicalSort(AOVnetwork, n)
   // G=(V, E), n=|V|
   for (i←0; i<n; i←i+1) do {
         select u with no predecessor;      // u∈V, indegree=0
         if (there is no such u) then return;
         print(u);
         remove u and all arcs incident from u;
   }
end topologicalSort
```



모든 결과를 출력하려면 아래와 같은 처리가 필요

< 출력 결과 >
0 1 2 3 4 5
0 1 2 4 3 5
0 2 1 3 4 5
0 2 1 4 3 5

	0	1	2	3	4	5
0	0	1	1	0	0	0
1	0	0	0	1	1	0
2	0	0	0	1	1	0
3	0	0	0	0	0	5
4	0	0	0	0	0	5
5	0	0	0	0	0	0

초기 그래프

	1	2	3	4	5
1	0	0	1	1	0
2	0	0	1	1	0
3	0	0	0	0	5
4	0	0	0	0	5
5	0	0	0	0	0

remove u and all arcs incident from u; //0을 선택한 경우
이 다음은 1, 2 중 하나를 선택할 수 있다.