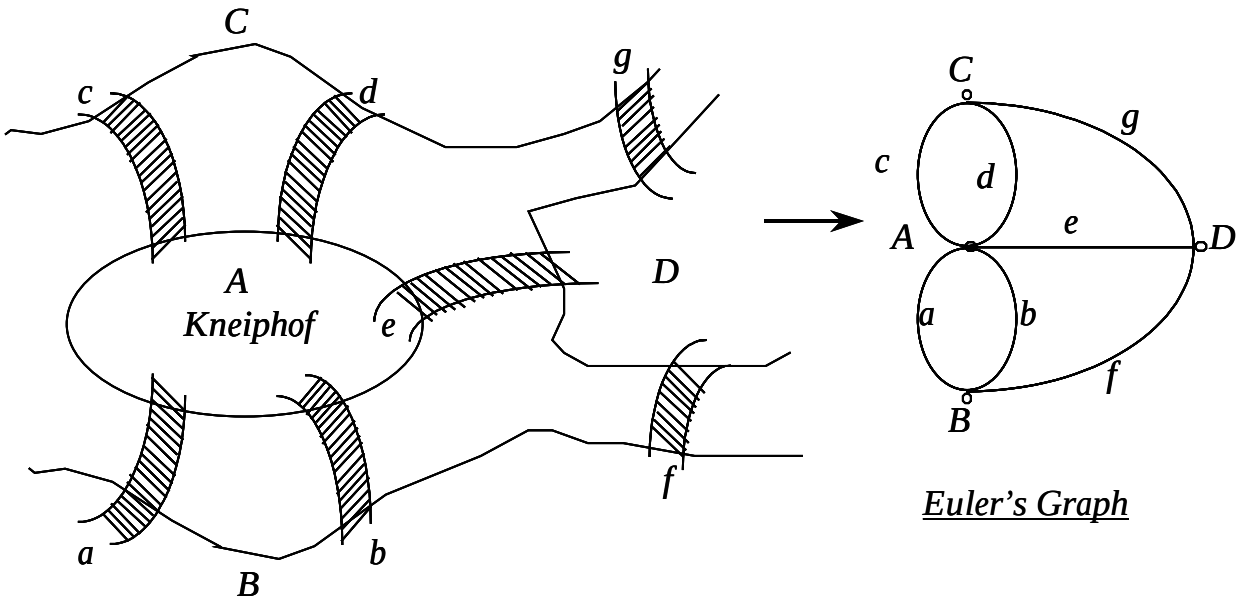
제 6 장 그래프

6.1 그래프 추상 데이터 타입

6.1.1 개 요

▶ **페이지 262, 그림 6.1 : Koenigsberg의 다리들**



□ 1736년 Euler가 Koenigsberg의 다리 문제 해결하기 위해 최초로 사용

- 그래프에 각 지역을 정점(vertex)으로 모든 다리를 간선(edge)으로 표시
  - 각 정점에 접한 간선의 수가 짝수인 경우에만 임의의 정점에서 출발하여 각 간선을 한번씩만 거치고 출발한 정점으로 되돌아 오는 길이 있다

→ 오일러 행로(Eulerian walk)

□ 응용

- 전기 회로의 분석, 최단 거리 탐색, 연구 계획의 분석, 화학 합성물들의 식별 등

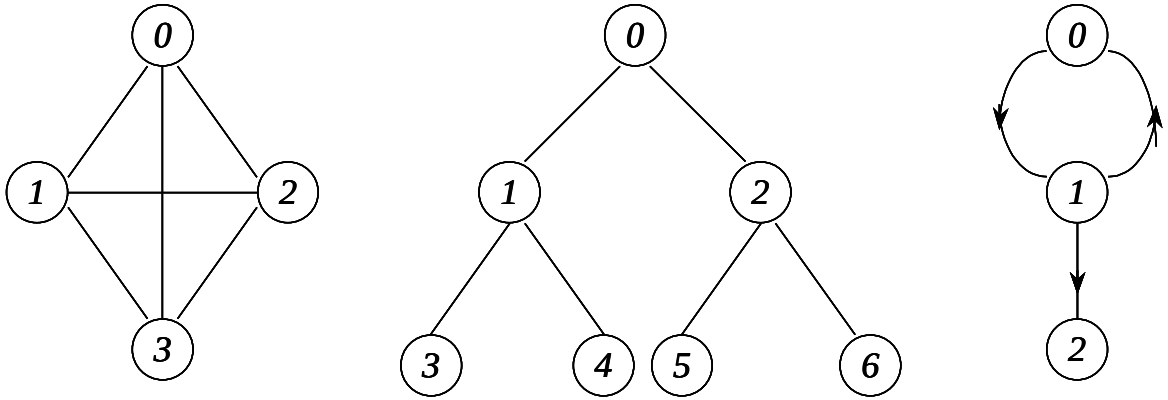
6.1.2 정 의

- 그래프, $G = (V, E)$
  - V : 공집합이 아닌 정점(vertex)들의 유한 집합
  - E : 정점 쌍, 간선(edge)들의 집합
  - → $V(G)$: 그래프 G의 정점들의 집합
  - → $E(G)$: 그래프 G의 간선들의 집합

- 무방향 그래프 (undirected graph)
  - $(u, v) = (v, u)$

- 방향 그래프 (directed graph)
  - $\langle u, v \rangle \neq \langle v, u \rangle$
  - 정점의 쌍 $\langle u, v \rangle$
    - u : 꼬리 (tail), v : 머리 (head)

▶ **페이지 263, 그림 6.2 : 세 개의 샘플 그래프**



G_1 G_2 G_3

$V(G_1)$ = { 0, 1, 2, 3 } $E(G_1)$ = { (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3) }

$V(G_2)$ = { 0, 1, 2, 3, 4, 5, 6 } $E(G_2)$ = { (0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6) }

$V(G_3)$ = { 0, 1, 2 } $E(G_3)$ = { <0, 1>, <1, 0>, <1, 2> }

□ 제한

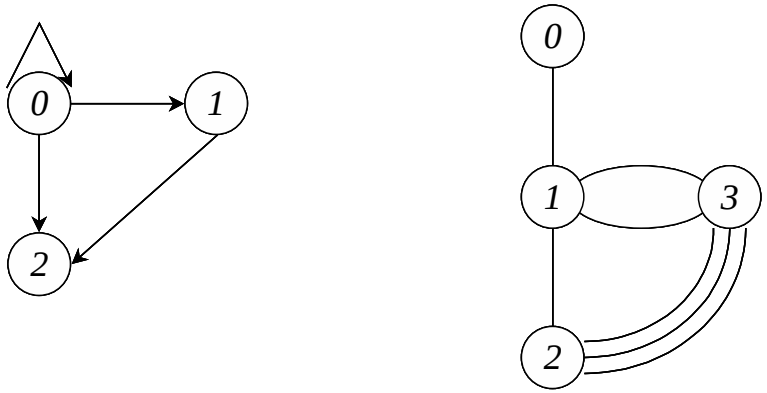
- 임의의 정점에서 자신으로 이어지는 간선을 가질 수 없다

  → 자기 간선 (self edge)

- 같은 간선을 중복해서 가질 수 없다

  → 다중 그래프 (multigraph)

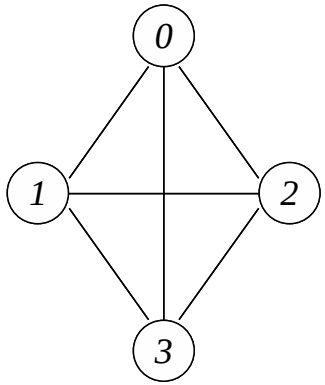
✧ **페이지 264, 그림 6.3 : 피드백 루프를 가지는 그래프와 다중 그래프의 예**



□ 완전 그래프 (Complete graph)

- 무방향 그래프
  - *n*개의 정점과 $n(n-1)/2$개의 간선을 가진 그래프
- 방향 그래프
  - *n*개의 정점과 $n(n-1)$개의 간선을 가진 그래프

✧ **페이지 263, 그림 6.2 : 세 개의 샘플 그래프 중 G_1**



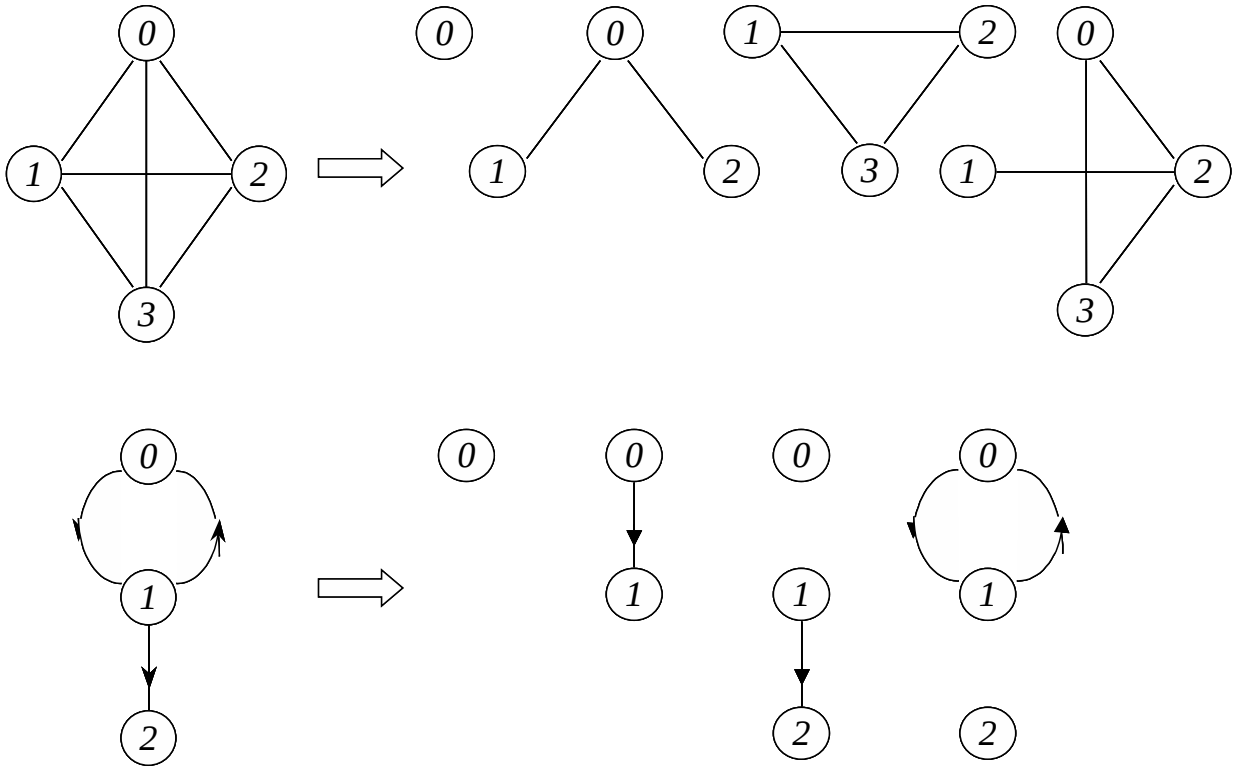
□ 인접과 부속

- (u, v)가 $E(G)$의 한 간선일 때
  - u와 v는 인접한다(adjacent)
  - 간선 (u, v)는 정점 u와 v에 부속된다(incident)
- <u, v>가 $E(G)$의 한 간선일 때
  - u adjacent to v and v adjacent from u
  - 간선 <u, v>는 정점 u와 v에 부속된다(incident)

□ G의 부분 그래프 (Subgraph), G'

- $V(G') \subseteq V(G)$ 이고 $E(G') \subseteq E(G)$

✧ **페이지 264, 그림 6.4 : G_1과 G_3의 부분 그래프**



□ 정점의 차수 (degree)

- 정점에 부속한 간선들의 수
- 방향 그래프에서의 진입 / 진출 차수
  - *v*의 진입 차수 (in-degree)
    : *v*가 머리인 간선들의 수
  - *v*의 진출 차수 (in-degree)
    : *v*가 꼬리인 간선들의 수
  - *n*개의 정점, *e*개의 간선을 가지는 무방향 그래프 *G*에서 정점 *i*의 차수를 d_i라 하면 $e = (1/2)\sum_{i=1}^{n} d_i$

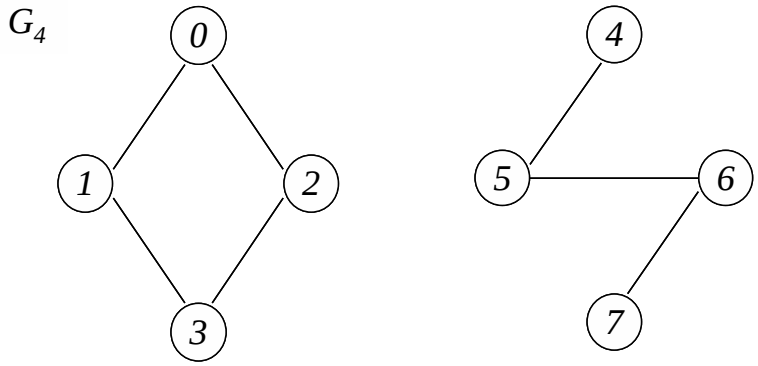
□ 경로(Path)

- $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$가 $E(G)$에 속한 간선들일 때 *G*에서의 정점 *u*로부터 *v*까지의 경로는 정점들의 열 $u, i_1, i_2, \ldots, i_k, v$
- 경로의 길이
  - 경로상의 간선의 수
- 단순 경로 (simple path)
  - 경로상의 처음과 마지막을 제외한 모든 정점을 다르다
- 사이클 (cycle)
  - 처음과 마지막 정점이 다른 단순 경로

□ 연결 그래프 (connected graph), G

- 정점 *u*와 *v*의 연결
  - *u*로부터 *v*까지의 경로 존재
- *V*(G)의 서로 다른 정점 *u*, *v*의 모든 쌍에 대해 *u*에서 *v*까지의 경로가 존재할 때 연결 그래프라 한다
- 연결 요소 (connected component)
  - 최대 연결 부분 그래프
- 트리
  - 사이클이 없는 연결 그래프

✧ **페이지 265, 그림 6.5 : 두 개의 연결 요소로 된 그래프**



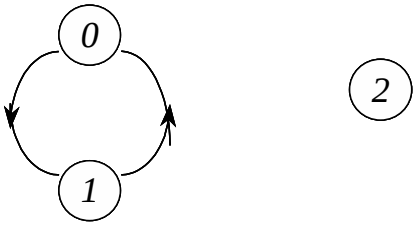
□ 강력 연결 그래프 (strongly connected graph)

- *V*(*G*)의 서로 다른 정점 *u*, *v*의 모든 쌍에 대해 *u*에서 *v*로, *v*에서 *u*로의 방향 경로가 존재

□ 강력 연결 요소 (strongly connected component)

- 강하게 연결된 최대 부분 그래프

✧ **페이지 266, 그림 6.6 : G_3의 강력 연결 요소**



G_3

▶ 페이지 266 ~ 267, 구조 6.1 : 그래프 추상 데이터 타입

structure *Graph*

objects: 공집합이 아닌 정점의 집합과 무방향 간선의 집합으로 각 간선은 정점의 쌍임

functions:

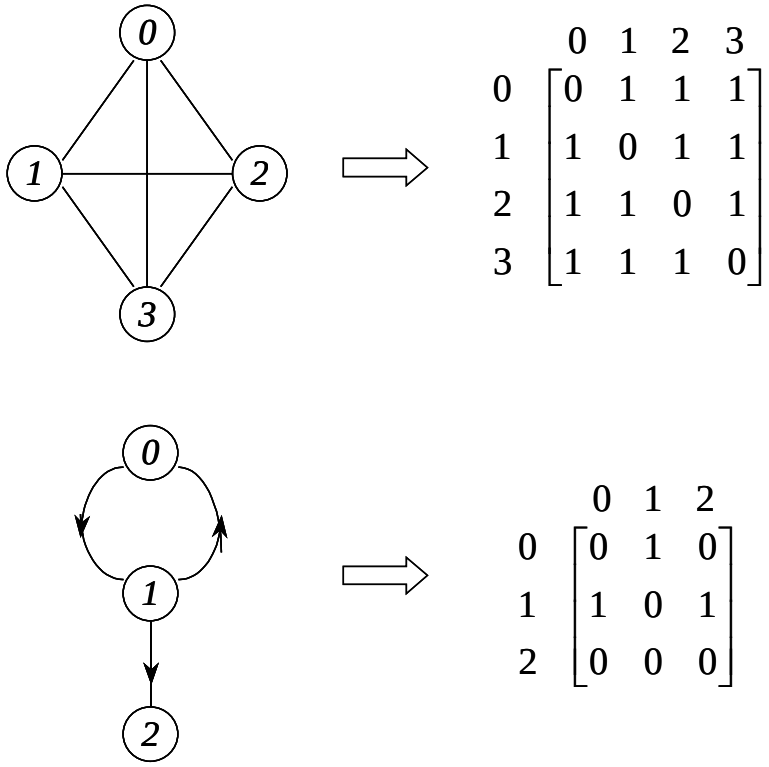
모든 *graph* $\in$ *Graph*, *v*, v_1, v_2 $\in$ *Vertices*에 대해서

Graph Create()	::= return 하나의 빈 그래프
Graph InsertVertex(*graph*, *v*)	::= return *v*를 삽입한 그래프 *v*는 부속한 간선을 갖지 않음
Graph InsertEdge(*graph*, v_1, v_2)	::= return v_1과 v_2 사이에 새 간선을 지닌 그래프
Graph DeleteVertex(*graph*, *v*)	::= return *v*와 *v*에 부속된 모든 간선들이 삭제된 그래프
Graph DeleteEdge(*graph*, v_1, v_2)	::= return 간선 (v_1, v_2)가 삭제된 그래프 부속한 노드들은 그래프에 그대로 둠
Boolean IsEmpty(*graph*)	::= if (*graph* == empty graph) return *TRUE* else return *FALSE*
List Adjacent(*graph*, *v*)	::= return *v*에 인접한 모든 정점들의 리스트

6.1.3 그래프 표현법

6.1.3.1 인접 행렬 (adjacency matrices)

▶ **페이지 268, 그림 6.7 : 인접 행렬**



☐ 저장 공간 = n^2

- 무방향 그래프의 인접 행렬은 대칭(symmetric)
  - 기억 장소가 반정도 절약 가능

☐ Nontrivial questions

- *G*에 있는 간선의 수는? / *G*가 연결되어 있는가?
  - $O(n^2)$
  - → $O(n + e)$에 수행이 가능해야 !!

6.1.3.2 인접 리스트

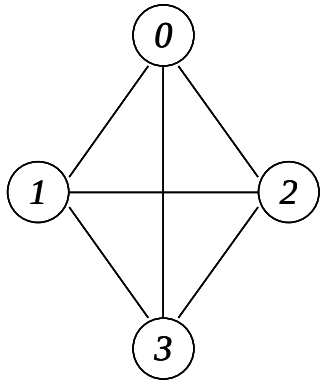
□ 인접 행렬의 *n*행들을 *n*개의 연결 리스트로 표현

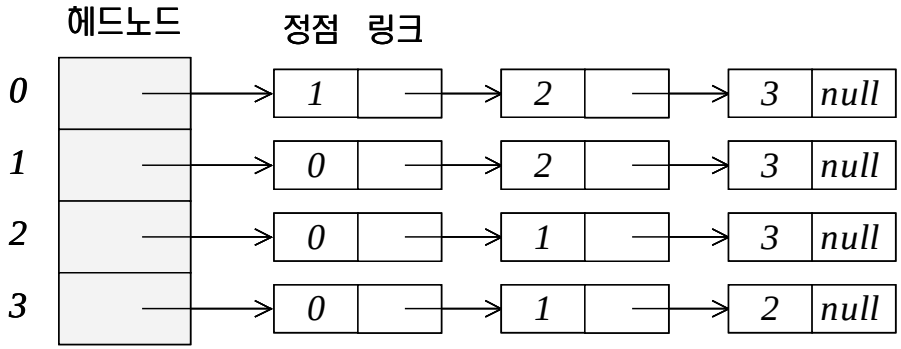
- 각 정점에 대해 한 개의 연결리스트 존재

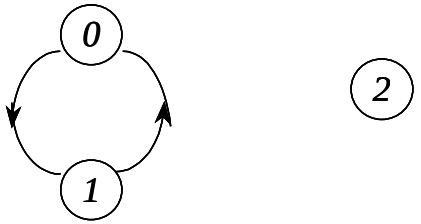
✧ **페이지 269, 인접 리스트를 위한 선언**

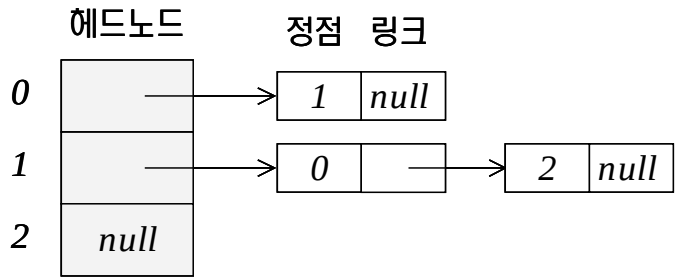
```
#define MAX_VERTICES 50   /* 정점의 최대 수 */
typedef struct node *node_ptr;
typdef struct node {
    int vertex;
    node_ptr link;
};
node_ptr graph[MAX_VERTICES];
int n = 0; /* 현재 사용중인 정점들 */
```

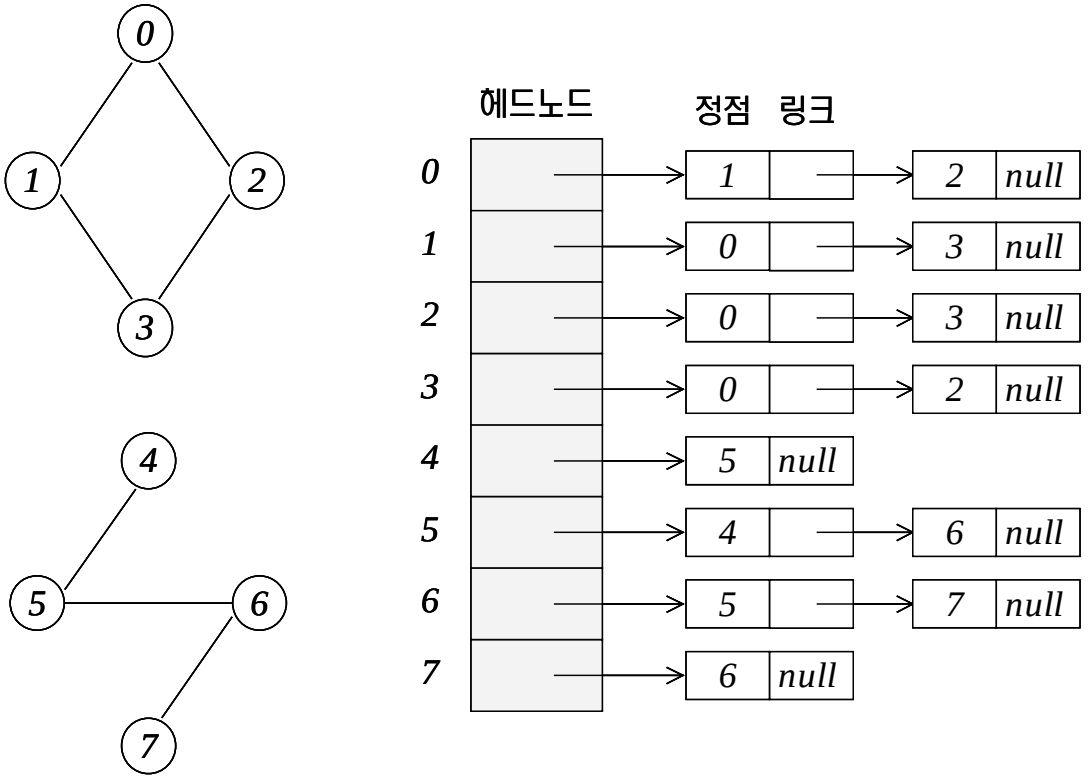
✧ **페이지 270, 그림 6.8 : 인접 리스트**






0
1
2


헤드노드
정점 링크
0
1
2
null
1
null
0
2
null


0
1
2
3
4
5
6
7
헤드노드
정점 링크
0 1 2 null
1 0 3 null
2 0 3 null
3 0 2 null
4 5 null
5 4 6 null
6 5 7 null
7 6 null

☐ 저장 공간

- 무방향 그래프 = *n*개의 헤드 노드 + 2*e*개의 리스트 노드
- 방향 그래프 = *n*개의 헤드 노드 + *e*개의 리스트 노드

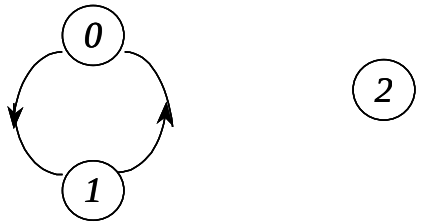
☐ Nontrivial questions

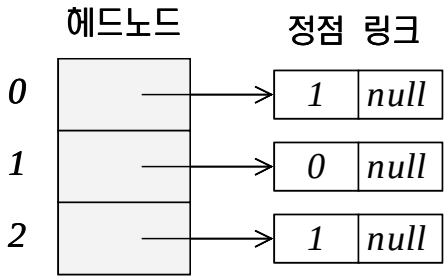
- O(*n* + *e*)
- 진입 차수를 구하려면 ?

□ 역인접 리스트 (inverse adjacency list)

◆ 각 정점에 대해 하나의 리스트, 리스트 노드들은 정점에 인접한 모든 정점에 대한 노드들로 구성

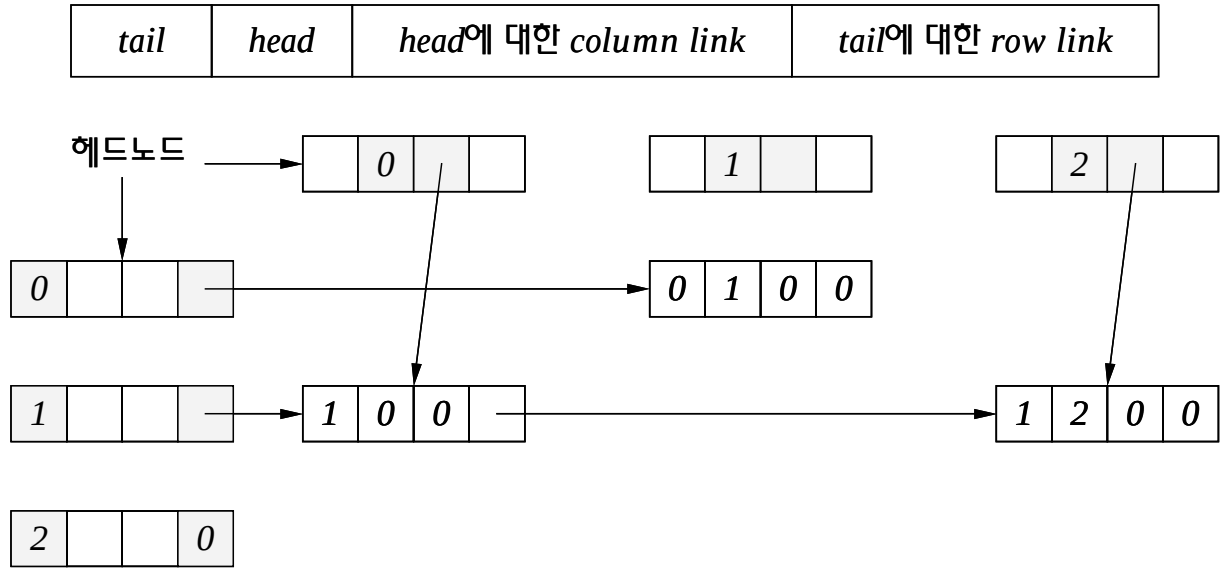
✧ **페이지 271, 그림 6.10 : 역인접 리스트**





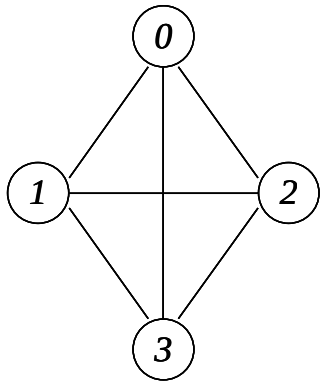
□ 희소 행렬 표현에서 사용된 노드 구조의 사용

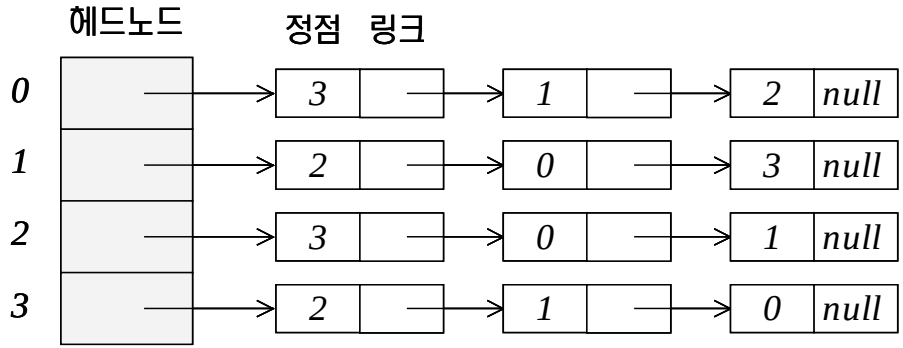
✧ **페이지 271, 그림 6.11 : 인접 리스트에 대한 또다른 노드 구조**
& 페이지 271, 그림 6.12 : 그래프의 직교 표현



□ 희소 행렬 표현에서 사용된 노드 구조의 사용

✧ **페이지 272, 그림 6.13 : G_4에 대한 오름차순이 아닌 인접 리스트**

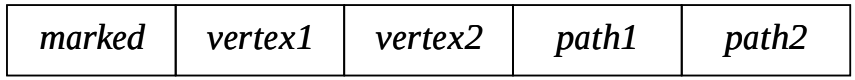




6.1.3.3 인접 다중 리스트

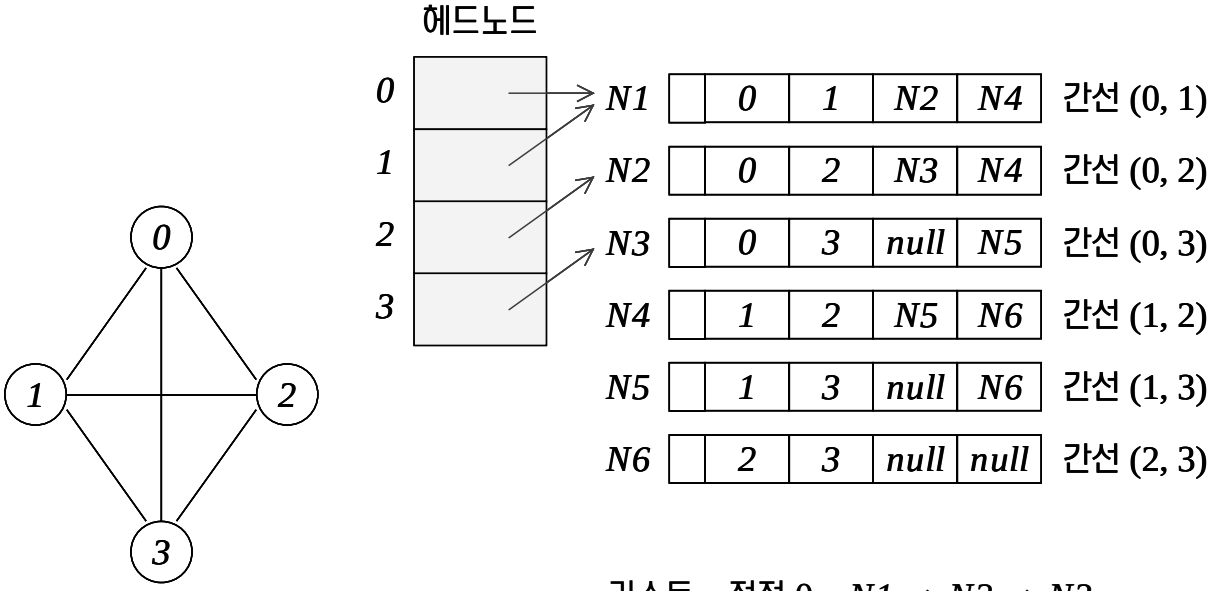
□ 모든 간선에 대해 하나의 노드만 있으나 두개의 리스트에 속한다

▶ **페이지 272, 그림 6.14 : 인접 다중 리스트를 위한 노드 구조**
& 페이지 272, 인접 다중 리스트를 위한 선언




```
typedef struct edge *edge_ptr;
typedef struct edge {
    short int marked;
    int vertex1;
    int vertex2;
    edge_ptr path1;
    edge_ptr path2;
};
edge_ptr graph[MAX_VERTICES];
```

▶ **페이지 273, 그림 6.15 : 인접 다중 리스트**

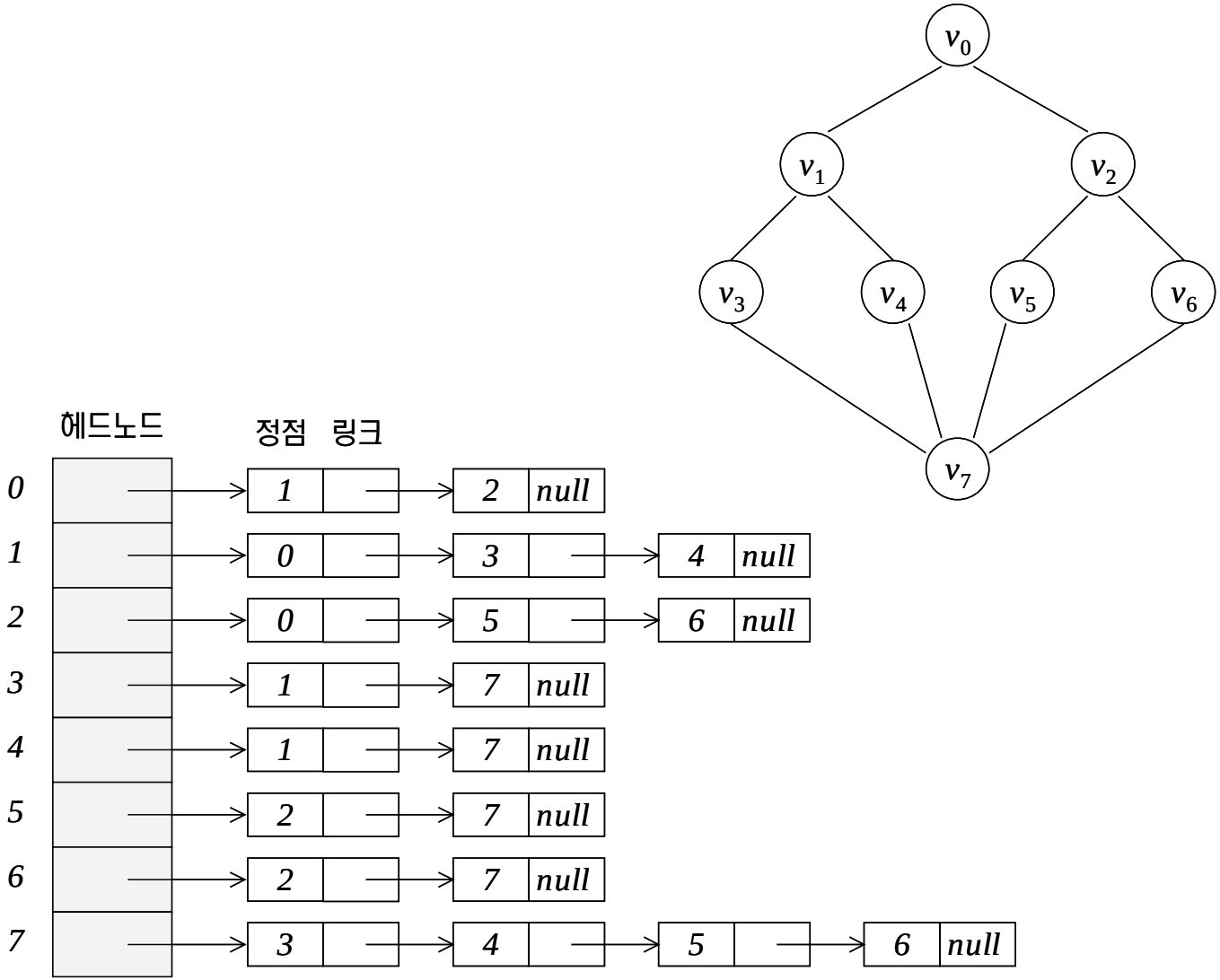


리스트 : 정점 0 : $N1 \rightarrow N2 \rightarrow N3$
정점 1 : $N1 \rightarrow N4 \rightarrow N5$
정점 2 : $N2 \rightarrow N4 \rightarrow N6$
정점 3 : $N3 \rightarrow N5 \rightarrow N6$

6.2 기본적인 그래프 연산

- 탐색
  - 한 정점 v에서 도달할 수 있는 모든 정점들을 방문
  - 깊이 우선 탐색 (depth first search; DFS)
  - 너비 우선 탐색 (breadth first search; BFS)

▶ 페이지 277, 그림 6.19 : 그래프와 인접 리스트



6.2.1 깊이 우선 탐색

□ 알고리즘

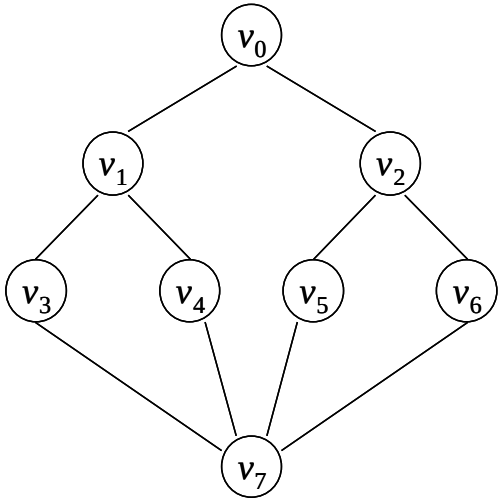
- 시작 정점 *v* 방문

→ *v*에 인접하면서 방문하지 않은 정점

→ *v*에 인접하면서 방문하지 않은 정점 *w* 택해 *w*를 시작점으로 깊이 우선 탐색 시작, 모든 인접 정점이 이미 방문된 정점 *u*에선 방문되지 않은 인접 정점 *w*를 갖고 제일 나중에 방문했던 정점으로 거슬러 올라가 정점 *w*로부터 다시 시작

→ 방문한 어떤 정점으로부터도 방문되지 않은 정점에 도달할 수 없을 때 탐색 끝남

▶ 페이지 276, 프로그램 6.1 : 깊이 우선 탐색




```
#define  FALSE  0
#define  TRUE   1
short int visited[MAX_VERTICES];
void dfs(int v)
{
/* 그래프의 정점 v에서 시작하는 깊이 우선 탐색 */
     node_pointer w;
     visited[v] = TRUE;
     printf("%5d", v);
     for (w = graph[v]; w; w = w->link)
          if (!visited[w->vertex])
               dfs(w->vertex);
}
```

□ 연산시간

- 인접리스트 사용시
  - 인접 리스트의 노드 1번씩 조사 (2*e*개) → O(*e*)
- 인접행렬 사용시
  - *v*에 인접한 정점들 결정 → O(*n*)
    방문해야 할 정점 *n*개 → O(n^2)

6.2.2 너비 우선 탐색

□ 알고리즘

- 시작 정점 *v* 방문

→ *v*에 인접한 모든 정점 방문

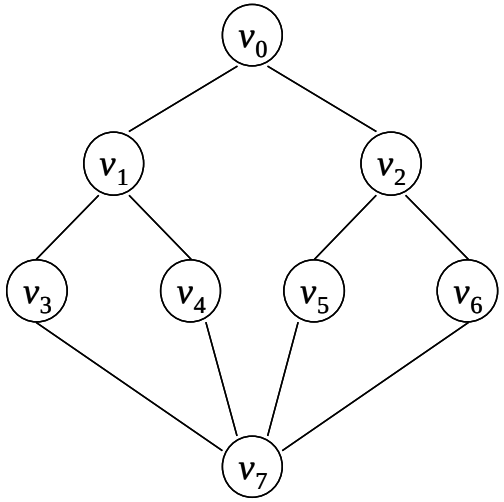
→ 이 정점들에 인접하여 방문하지 않은 정점 계속 방문

□ 동적 연결 큐의 사용

✧ **페이지 278, 큐의 정의문과 함수의 프로토타입**

```
typedef struct  queue *queue_pointer;
typedef struct  queue {
     int     vertex;
     queue_pointer   link;
};
void addq(queue_pointer *, queue_pointer *, int);
int   deleteq(queue_pointer *);
```

▶ 페이지 278 ~ 279, 프로그램 6.2 : 그래프의 너비 우선 탐색




```
void bfs(int v)
/* 정점 v에서 시작하는 너비 우선 탐색. 전역배열 visited는 0으로 초기화됨 */
{
    node_pointer w;  queue_pointer front, rear;
    front = rear = NULL;  /* 큐의 초기화 */
    printf("%5d", v);
    visited[v] = TRUE; addq(&front, &rear, v);
    while (front) {
        v = deleteq(&front);
        for (w = graph[v]; w; w = w->link)
            if (!visited[w->vertex]) {
                printf("%5d", w->vertex);
                addq(&front, &rear, w->vertex);
                visited[w->vertex] = TRUE;
            }
    }
}
```

□ 연산시간

- while 루프는 n번 반복
- 인접리스트 사용
  - d_i = degree(v_i)

  → 각 정점에 대한 while 루프의 비용 = d_i

  → $d_0 + d_1 + d_{n-1}$ = O(e)
- 인접행렬 사용
  - 각 정점에 대해 while 루프의 비용 = O(n)

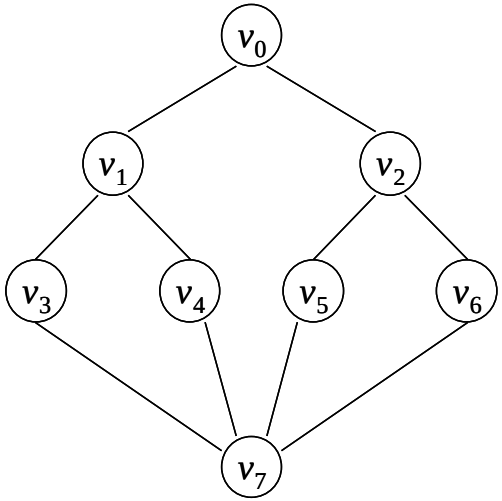
  → O(n^2)

6.2.3 연결 요소

- 무방향 그래프의 연결 여부를 결정하는 문제
  - bfs(0)나 dfs(0)를 호출한 후 방문이 안된 정점이 있는지 알아본다

- 주어진 그래프의 연결 요소(connected component)를 나열하는 문제
  - 아직 방문되지 않은 정점 *v*를 사용하여 bfs(*v*)나 dfs(*v*)를 계속해서 호출하여 찾아낸다

▶ **페이지 279 ~ 280, 프로그램 6.3 : 연결 요소**



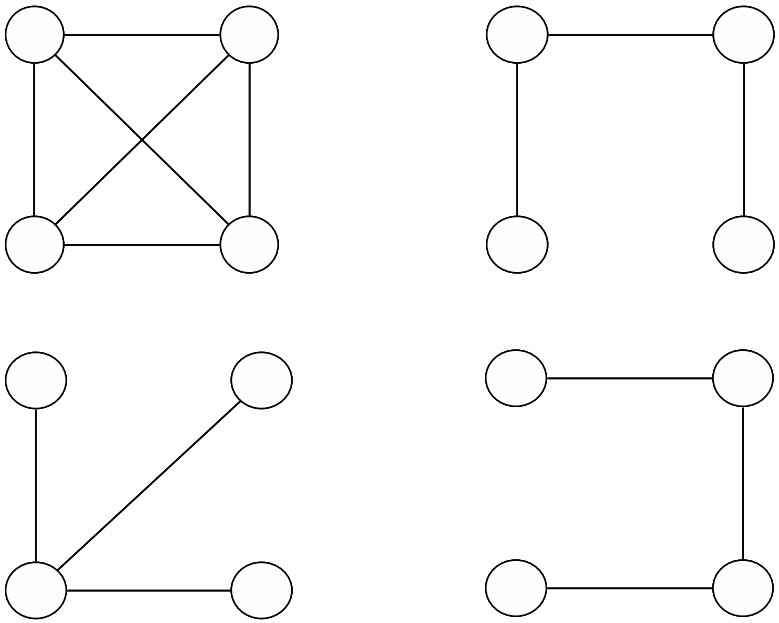

```
void connected(void)
{
/* 그래프의 연결 요소 결정 */
     int i;
     for (i = 0; i < n; i++)
     if (!visited[i])  {
          dfs(i);
          printf("\n");
     }
}
```

6.2.4 신장 트리

□ 신장 트리 (spanning trees)

- G의 간선들로만 구성되고 G의 모든 정점들이 포함된 트리

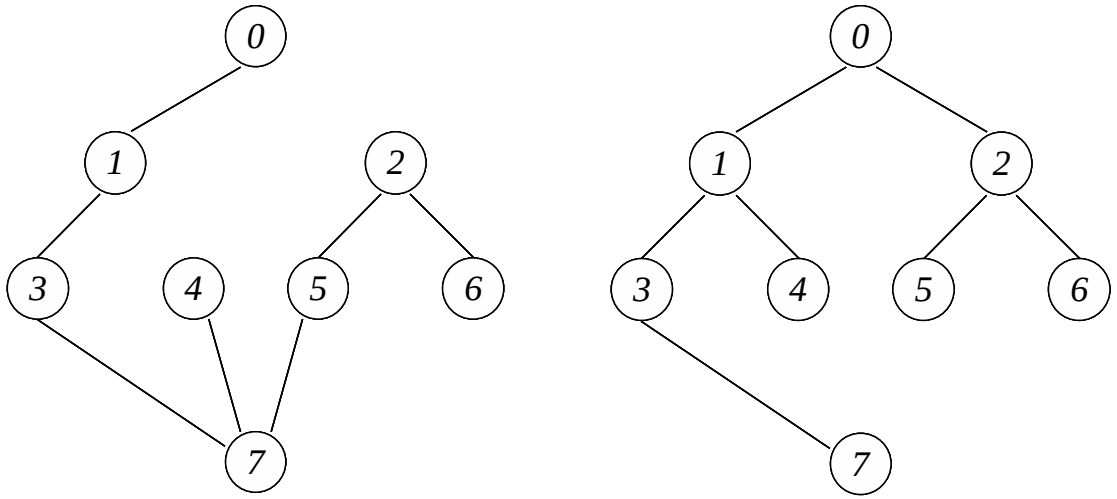
✧ **페이지 280, 그림 6.20 : 완전 그래프와 이 그래프의 세 신장 트리**



□ 그래프 G가 연결되어 있을 때, DFS나 BFS는 어떤 점에서 시작하든 G에 있는 모든 정점 방문

- 깊이 우선 신장 트리(depth first spanning tree)
  - DFS 호출하여 만들어진 신장 트리
- 너비 우선 신장 트리(breadth first spanning tree)
  - BFS 호출하여 만들어진 신장 트리

✧ **페이지 281, 그림 6.21 : dfs와 bfs 신장 트리**



dfs(0) 신장 트리 *bfs*(0) 신장 트리

□ 신장 트리의 특징

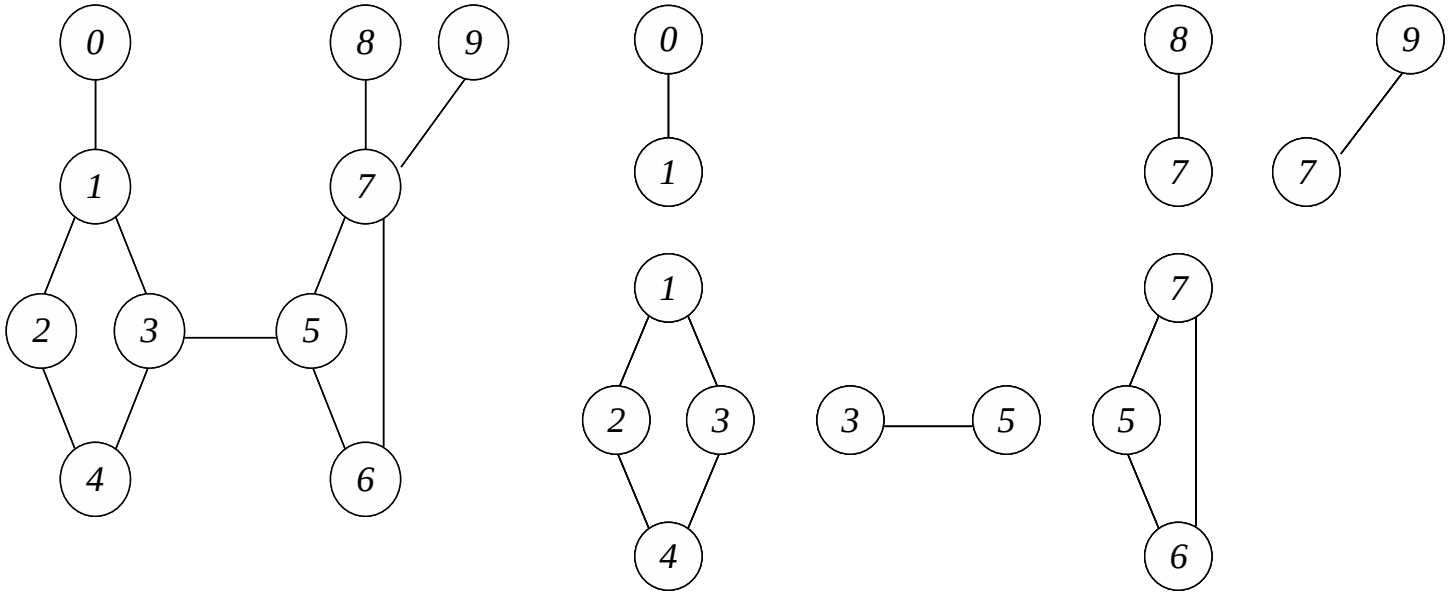
- 임의의 신장트리에 비트리 간선을 추가하면 사이클이 형성된다
  - ✓ 회로 등식의 생성
- 신장트리는 그래프 G의 최소 부분 그래프(minimal subgraph) G'로 $V(G') = V(G)$이고 G'은 연결되어 있다
  - ✓ 통신 네트워크 설계에 응용
- $|E(G')| = n - 1$ where $|V(G)| = n$

6.2.5 이중 결합 요소와 단절점

□ 이준 결합 요소

- 단절점 (articulation point)
  - 그래프의 한 정점과 이에 부속한 모든 간선들을 같이 삭제하면 최소한 두 개의 연결 요소를 가지게 되는 정점
- 이중 결합 그래프 (biconnected graph) = 단절점이 없는 연결 그래프
- 이중 결합 요소 (biconnected component) = 최대 이중 결합 부분 그래프

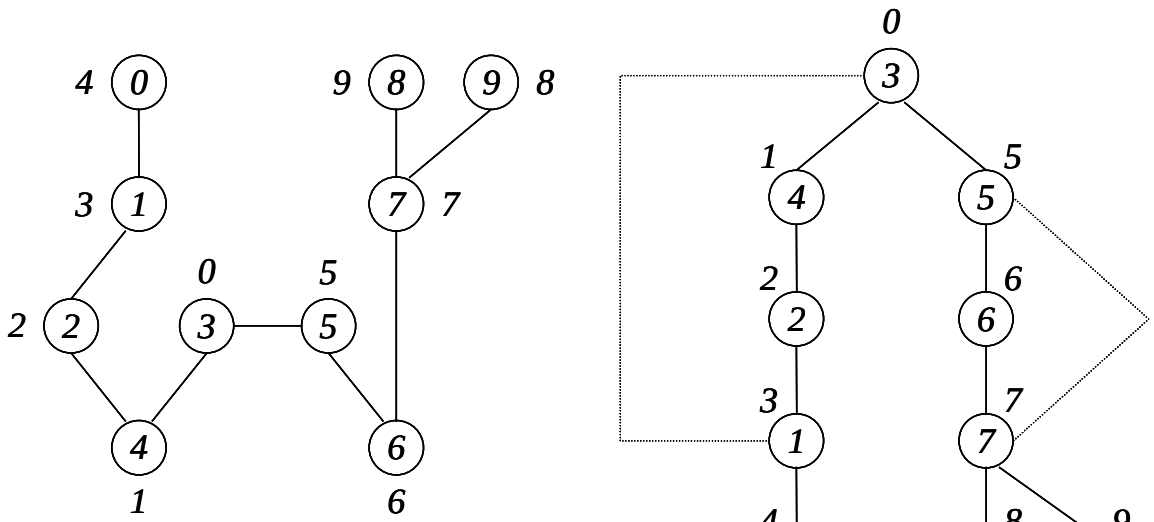
✧ **페이지 283, 그림 6.22 : 연결 리스트와 그 이중결합 요소들**



□ 비트리 간선

- 백 간선 (back edge), (*u*, *v*)
  - *u*가 *v*의 조상이거나 *v*가 *u*의 조상
- 교차 간선 (cross edge)
  - 백 간선이 아닌 비트리 간선
  - 어떤 그래프도 깊이 우선 신장 트리에 대해 교차 간선을 가질 수 없다

✧ **페이지 284, 그림 6.23 : 깊이 우선 신장 트리**



깊이 우선 신장 트리

☐ dfn & low

- *dfn*(*w*) = 깊이 우선 번호
- *low*(*w*) = min { *dfn*(*w*), min { *low*(*x*) | *x* is a child of *w* },
  min { *dfn*(*x*) | (*w*, *x*) is a back edge } }

✧ **페이지 284, 그림 6.24 : 루트를 3으로 하는 dfs 신장 트리의 dfn 값과 low 값**

정점	0	1	2	3	4	5	6	7	8	9
dfn	*4*	*3*	*2*	*0*	*1*	*5*	*6*	*7*	*9*	*8*
low	*4*	*0*	*0*	*0*	*0*	*5*	*5*	*5*	*9*	*8*

☐ 단절점, *u*

- 두 개 이상의 자식을 가지는 신장 트리의 루트이거나 *low*(*w*) ≥ *dfn*(*u*)를 만족하는 자식 *w*를 가진다

▶ **페이지 285, 선언문**

& 페이지 285 ~ 286, 프로그램 6.4 : dfn과 low의 결정

& 페이지 286, 프로그램 6.5 : dfn과 low의 초기화

& 페이지 286 ~ 287, 프로그램 6.6 : 그래프의 이중결합 요소

```
#define MIN2(x, y) ((x) < (y) ? (x) : (y))
short int dfn[MAX_VERTICES];
short int low[MAX_VERTICES];
int num;

void init(void)
{
    int i;
    for(i = 0; i < n; i++) {
        visited[i] = FALSE;
        dfn[i] = low[i] = -1;
    }
    num = 0;
}
```

```
void dfnlow(int u, int v)
{
    node_ptr ptr;
    int w;
    dfn[u] = low[u] = num++;
    for (ptr = graph[u]; ptr; ptr = ptr->link) {
        w = ptr->vertex;
        if(dfn[w] < 0) {
            dfnlow(w, u);
            low[u] = MIN2(low[u],low[w]);
        }
        else if (w != v)
            low[u] = MIN2(low[u], dfn[w]);
    }
}
```

```
void bicon(int u, int v)
{
    node_ptr ptr;    int w, x, y;
    dfn[u] = low[u] = num++;
    for(ptr = graph[u];ptr; ptr = ptr->link) {
        w = ptr->vertex;
        if(v != w && dfn[w] < dfn[u]) add(&top, u, w);
        if(dfn[w] < 0) {
            bicon(w, u); low[u] = MIN2(low[u], low[w]);
            if(low[w] >= dfn[u]) {
                printf("new biconnected component: ");
                do {
                    delete(&top, &x, &y); printf("  <%d,%d>", x, y);
                } while(!((x == u) && (y == w)));
                printf("\n");
            }
        }
        else if (w != v) low[u] = MIN2(low[u], dfn[w]);
    }
}
```

6.3 최소 비용 신장 트리

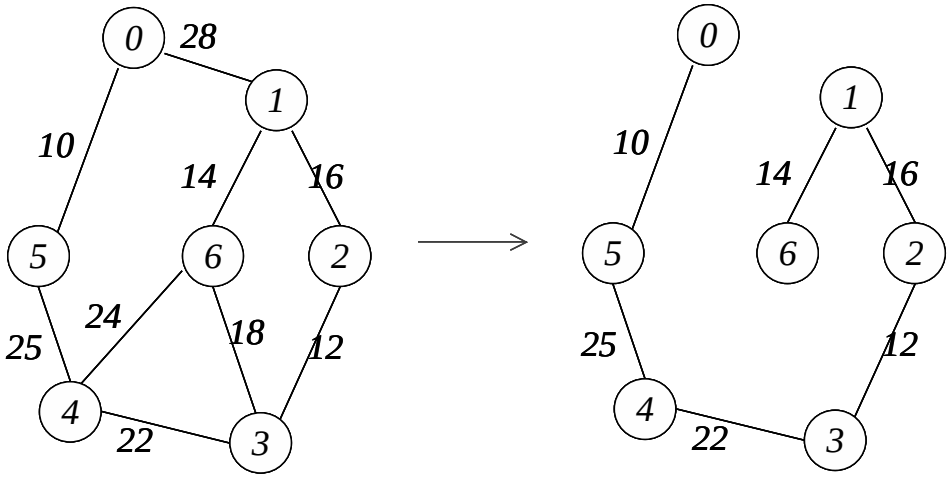
□ 최소 비용 신장 트리

- 최저의 비용을 가지는 신장 트리
- 만족해야 하는 제한 조건
  - 그래프 내에 있는 간선들만을 사용
  - 정확하게 $n - 1$개의 간선만을 사용
  - 사이클을 생성하는 간선을 사용하지 않아야 한다

□ 알고리즘

- greedy method
  - 최적의 해를 단계별로 구한다
- Kruskal 알고리즘
- Prim 알고리즘
- Sollin 알고리즘

▶ 최소 비용 신장 트리의 예



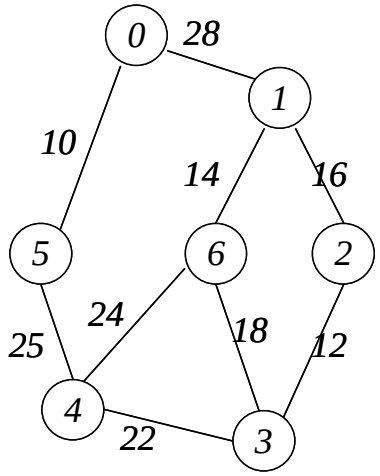
6.3.1 Kruskal 알고리즘

□ 알고리즘

- 간선들을 하나씩 결정하여 마지막으로 최소 비용 신장 트리 *T*를 만든다
- 간선들은 비용이 작은 순으로 *T*에 포함하되, 이미 *T*속에 있는 간선과 비교해서 사이클을 형성하지 않으면 *T*에 포함한다

▶ **페이지 291, 프로그램 6.7 : Kruskal 알고리즘**

& 페이지 290, 그림 6.25 : Kruskal 알고리즘의 각 단계



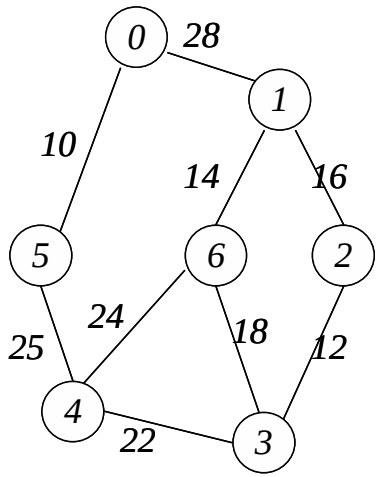

```
T = { };
while (T가 n – 1개 미만의 간선 포함 && E가 비어있지 않음) {
    E에서 최저 비용 간선 (v, w) 선택;
    E에서 (v, w)를 삭제;
    if ((v, w)가 T에서 사이클을 형성하지 않음)
        (v, w)를 T에 추가;
    else
        (v, w)를 거부;
}
if (T가 n – 1개 보다 적은 간선을 포함)
    printf("No spanning tree\n");
```

6.3.2 Prim 알고리즘

□ 알고리즘의 각 단계에서 선택된 간선의 집합이 트리를 이룬다

▶ **페이지 293, 프로그램 6.8 : Prim 알고리즘**

& 페이지 294, 그림 6.27 : Prim 알고리즘의 각 단계




```
T = { };
TV = {0}; /* 정점 0으로 시작. 간선은 비어있음.*/
while (T의 간선수가 n – 1보다 적음) {
    u ∈ TV이고 v ∉ TV인 최저 비용 간선을 (u, v)라 함;
    if (그런 간선이 없음)
        break;
    v를 TV에 추가;
    (u, v)를 T에 추가;
}
if (T의 간선수가 n – 1 보다 적음)
    printf("No spanning tree\n");
```

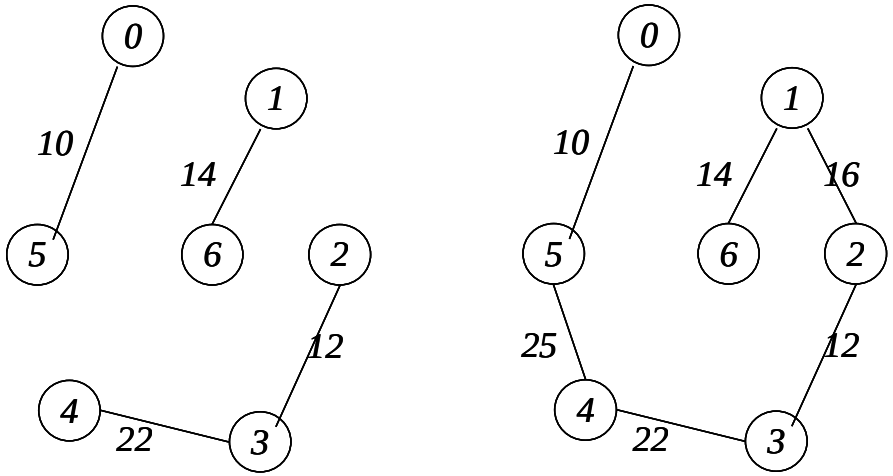
6.3.3 Sollin 알고리즘

□ 알고리즘

- 각 단계에서 *T*에 포함될 간선을 여러개 선택
- 한 단계의 시작점에서 선택된 간선들은 *n*개의 모든 그래프 정점들을 포함하면서 신장 포리스트를 이룬다
- 한 단계 동안 포리스트내에 있는 각 트리에 대해 하나의 간선을 선택

▶ **페이지 293, 프로그램 6.8 : Prim 알고리즘**

& 페이지 294, 그림 6.27 : Prim 알고리즘의 각 단계

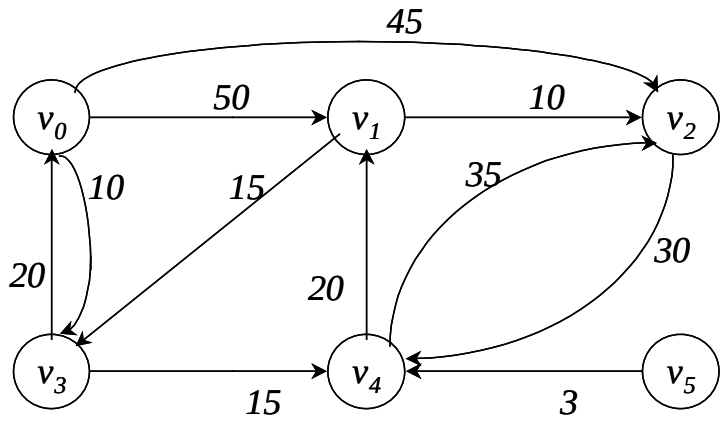


6.4 최단 경로와 이행적 폐쇄

- 단일 출발점 / 모든 종점
  - 양수 간선 비용
    - Dijkstra 알고리즘
  - 일반적인 가중치
    - BellmanFord 알고리즘

6.4.1 Dijkstra 알고리즘

▶ **페이지 297, 그림 6.29 : 그래프와 v_0에서의 최단 경로**



□ 알고리즘의 설계

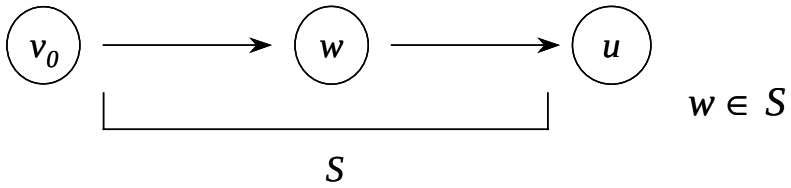
v_0 : 출발점

S : 최단 경로가 이미 찾아진 정점들의 집합

w : 임의의 정점

distance[w] = S에 있지 않은 w에 대해 v_0에서 출발하여 S에 있는 정점들만을 거쳐 w에 도달하는 최단경로

- 경로는 길이의 크기 순서로 생성된다
- 다음 최단 경로가 정점 u로 가는 것이라면 v_0에서 시작하여 u에서 끝나며 그 경로는 S에 있는 정점들만을 통과하게 된다
  –

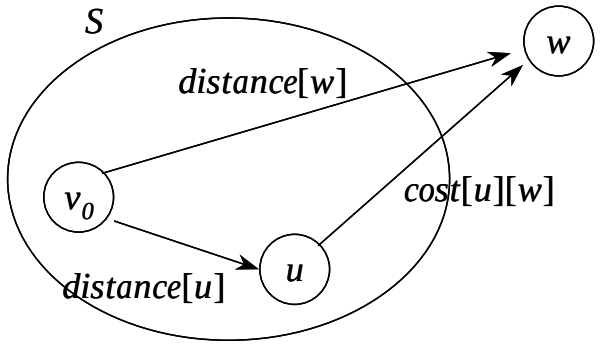


  – 최단 경로는 경로 길이의 크기 순서로 구해지므로
- 생성된 다음 경로의 종착점은 S에 없는 모든 정점 중에서 *distance*[u]가 최소인 그런 정점 u가 되어야 한다

- v_0에서 u로 가는 최단 경로를 생성하고 나면 정점 u는 S의 원소가 된다
  - 이때 *distance*[*u*]는 줄어들 수 있다
  - v_0에서 u를 거쳐 w로 가는 경로가 u를 거치지 않은 경로보다 짧은 경우 u에서 w로 가는 경로가 어떤 정점 (중간 정점)도 포함하지 않으면 이 경로의 길이는 *distance*[*u*] + *length*(<*u*, *w*>)

  → min{*distance*[*w*], *distance*[*u*] + *cost*[*u*][*w*]}

  –

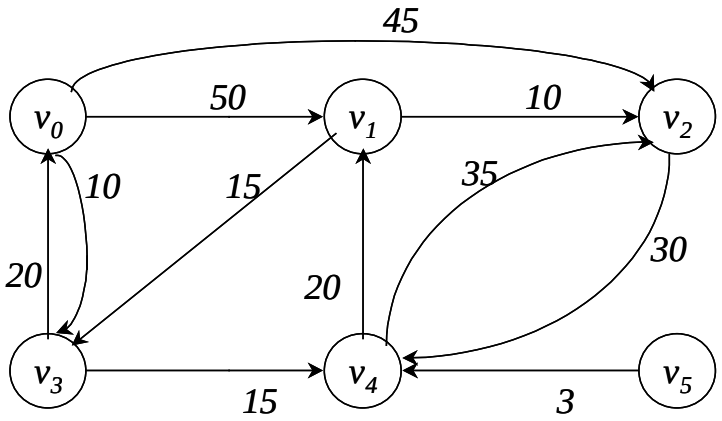


□ 가정

- *n*개의 정점에 0 부터 n – 1까지의 번호 부여
- 배열 *found*
  - 정점 *i*가 *S*에 속하지 않으면 *found*[*i*] = FALSE
    속하면 *found*[*i*] = TRUE
- *cost*[*i*][*j*]는 간선 <*i*, *j*>의 가중치를 갖는 비용 인접 행렬로 표현

▶ **페이지 298, 프로그램 6.9 : 최단 경로 알고리즘을 위한 선언문**

```
#define MAX_VERTICES 6
int cost[][MAX_VERTICES] = {
    {   0,   50,   10, 1000,   45, 1000},
    {1000,    0,   15, 1000,   10, 1000},
    {  20, 1000,    0,   15, 1000, 1000},
    {1000,   20, 1000,    0,   35, 1000},
    {1000, 1000,   30, 1000,    0, 1000},
    {1000, 1000, 1000,    3, 1000,    0}};
int distance[MAX_VERTICES];
short int found[MAX_VERTICES];
int n = MAX_VERTICES;
```



▶ 페이지 299, 프로그램 6.10 : 하나의 출발점을 갖는 최단 경로들

& 페이지 299 ~ 300, 프로그램 6.11 : 최저 비용 간선의 선택

```
void shortestpath(int v, int cost[][MAX_VERTICES],
    int distance[], int n, short int found[])
{
     int i, u, w;
     for (i = 0; i < n; i++) {
          found[i] = FALSE; distance[i] = cost[v][i];
     }
     found[v] = TRUE; distance[v] = 0;
     for (i = 0; i < n – 2; i++) {
          u = choose(distance, n, found);
          found[u] = TRUE;
          for (w = 0; w < n; w++)
               if (!found[w])
                    if (distance[u] + cost[u][w] < distance[w])
                         distance[w] = distance[u] + cost[u][w];
     }
}
```

```
int choose(int distance[], int n, int found[])
{
/* 아직 조사되지 않은 정점 중에서 distance가 최소인 것을 찾음 */
    int i, min, minpos;
    min = INT_MAX;
    minpos = -1;
    for (i = 0; i < n; i++)
        if (distance[i] < min && !found[i]) {
            min = distance[i];
            minpos = i;
        }
    return minpos;
}
```

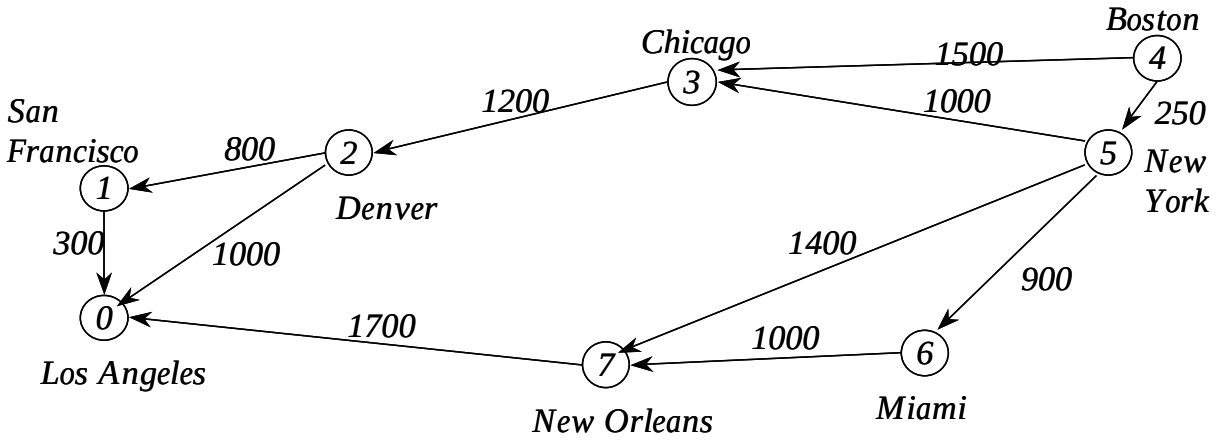
✍ 알고리즘 분석

- 인접 행렬 표현 사용
  - G에 있는 간선들을 결정하는 데 $O(n^2)$의 시간

  → 경로 알고리즘의 시간 복잡도 = $O(n^2)$

▶ **페이지 300, 그림 6.30 : 항공 노선의 다이그래프**

& 페이지 302, 그림 6.31 : 항공 노선 다이그래프에 대한 shortestpath의 작동



	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

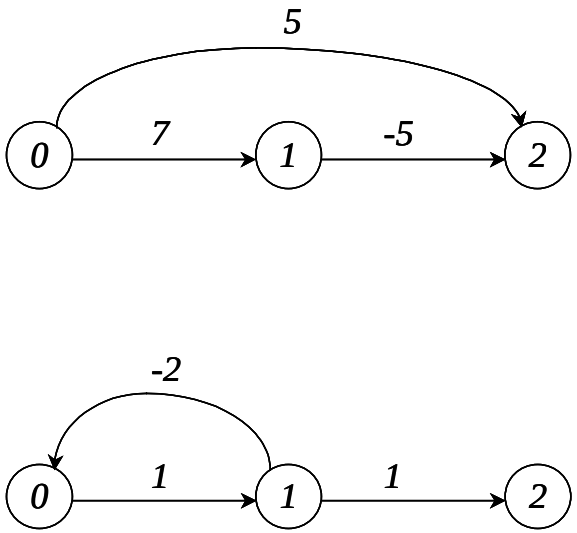
반복	S	Vertex selected	Distance							
			LA	SF	DEN	CHI	BOST	NY	MIA	NO
			[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	--	----	+∞	+∞	+∞	1500	0	250	+∞	+∞
1	{4}	5	+∞	+∞	+∞	1250	0	250	1150	1650
2	{4,5}	6	+∞	+∞	+∞	1250	0	250	1150	1650
3	{4,5,6}	3	+∞	+∞	2450	1250	0	250	1150	1650
4	{4,5,6,3}	7	3350	+∞	2450	1250	0	250	1150	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}									

6.4.2 BellmanFord 알고리즘

☐ Dijkstra 알고리즘

- 간선이 음의 길이를 가지는 경우 옳지 않다

✧ **페이지 302, 그림 6.32 : 음의 사이클을 갖는 그래프**

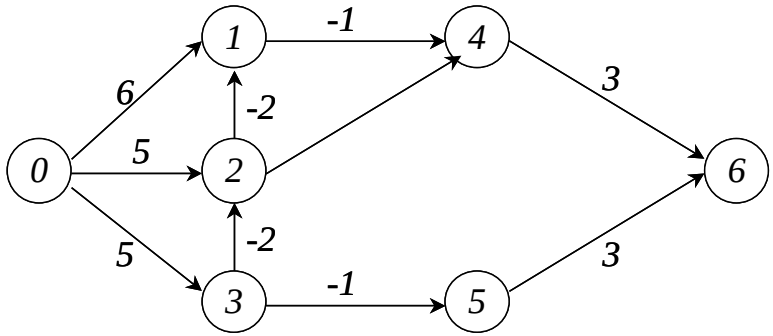


☐ BellmanFord 알고리즘

- 가정
  - 음의 길이 사이클이 존재하지 않는다
- $dist^l[u]$
  - l개의 간선을 포함하는 시발점 v로부터 u까지의 최단 경로의 길이
  - $dist^k[u] = \min\{ dist^{k-1}[u], \min\{ dist^{k-1}[i] + length[i][u]\}\}$

→ 모든 u에 대해 $dist^{n-1}[u]$를 구한다

✧ **BellmanFord 알고리즘**




```
void BellmanFord(int n, int v)
{
    int i, k;
    for (i = 0; i < n; i++) dist[i] = lengh[v][i];
    for (k = 2; k <= n – 1; k++) {
        for (each u such that u != v and u has at least one incoming edge)
            for (each <i, u> in the graph)
                if (dist[u] > dist[i] + length[u][i]) dist[i] = dist[i] + length[i][u];
    }
}
```

6.4.3 모든 쌍의 최단 경로

- $i \neq j$인 모든 정점의 쌍 v_i와 v_j간의 최단 경로를 구한다

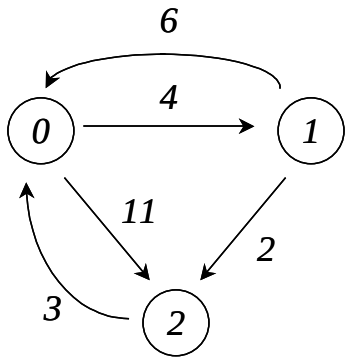
- 간단한 알고리즘
  - *n*개의 독립적인 단일 시발점/모든 종점 문제로 처리
    - shortestpath의 *n*번 적용
      : $O(n^3)$ 또는 $O(n^2 \log n + ne)$
    - bellmanford의 *n*번 적용
      : $O(n^4)$ 또는 $O(n^2 e)$
  - → All-pairs 알고리즘

□ 알고리즘의 기술

- 비용 인접 행렬 (cost adjacency matrix) 표현
  - $cost[i][j] = 0$, where $i = j$
  - $cost[i][j] = \infty$, where $\langle i, j \rangle \notin E(G)$, $i \neq j$
- $A^k[i][j]$
  - k보다 큰 인덱스를 가지는 정점을 중간에 통과하지 않고 i에서 j로 가는 최단경로의 비용
- $A^{n-1}[i][j]$
  - i에서 j로 가는 최단 경로의 비용
- $A^{-1}[i][j] = cost[i][j]$
- A^{k-1} 를 생성했으면 A^k의 생성이 가능
  - i에서 j까지의 최단 경로가 인덱스 k보다 큰 정점을 통과하지 않는 경우
    : $A^k[i][j] = A^{k-1}[i][j]$
  - i에서 j로 가는 최단 경로가 인덱스가 k인 정점 통과하는 경우
    : 이 경로는 i에서 k로, k에서 j로 가는 경로로 구성되며
    $k-1$ 보다 큰 인덱스를 갖는 정점을 통과하지 않고
    i에서 k로, k에서 j로의 최단 경로이다
    : $A^k[i][j] = A^{k-1}[i][k] + A^{k-1}[k][j]$
- $A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, \quad k \geq 0$
  $A^{-1}[i][j] = cost[i][j]$

▶ **페이지 303, 그림 6.33 : 방향 그래프와 비용 행렬**

& 페이지 304, 그림 6.34 : allcosts가 생성한 A^k 행렬들



A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^1	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

▶ 페이지 303, 프로그램 6.12 : 모든 쌍의 최단 경로를 구하는 함수

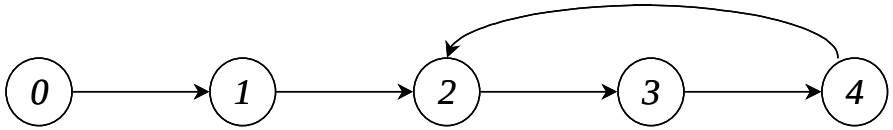
```
void allcosts(int cost[][MAX_VERTICES], int distance[][MAX_VERTICES], int n)
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            distance[i][j] = cost[i][j];
    for (k = 0; k < n; k++)
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (distance[i][k]+distance[k][j] < distance[i][j])
                    distance[i][j] = distance[i][k] + distance[k][j];
}
```

6.4.4 이행적 폐쇄

- 이행적 폐쇄 행렬 A^+
  - $A^+[i][j] = 1$, i에서 j로의 경로의 길이가 0보다 클 때
  - $A^+[i][j] = 0$, 그렇지 않으면

- 반사 이행적 폐쇄 행렬 A*
  - $A^*[i][j] = 1$, i에서 j로의 경로의 길이가 0이거나 0보다 클 때
  - $A^*[i][j] = 0$, 그렇지 않으면

▶ **페이지 305, 그림 6.35 : 그래프 *G*와 인접 행렬 *A*, A^+, A^***



$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

A^+

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

A^*

6.5 작업 네트워크

☐ AOV 네트워크

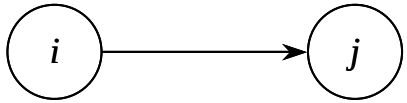
- activity on vertex network
- 정점이 작업을 나타내고 그 간선이 작업 간의 우선 관계를 나타내는 방향 그래프

☐ AOE 네트워크

- 방향 간선 : 프로젝트에서 수행되어야 할 작업
- 정점 : 사건

6.5.1 AOV 네트워크

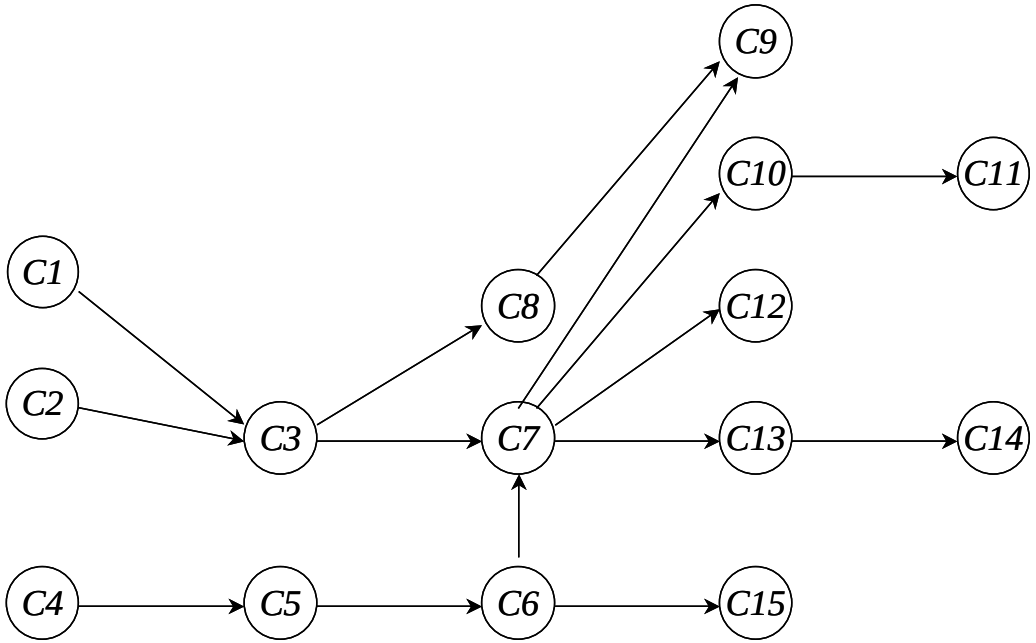
- 선행 관계(precedence relation)
  - AOV 네트워크 *G*에서 만약 *i*에서 *j*까지의 방향 경로가 존재하면 정점 *i*는 정점 *j*의 선행자(predecessor) 이다
    - 만약 <*i*, *j*>가 *G*의 간선이면 *i*는 *j*의 즉각 선행자(immediate predecessor)이다
  - 만약 *i*가 *j*의 선행자라면 *j*는 *i*의 후속자(successor)이다
    - *i*가 *j*의 즉각 선행자면 *j*는 *i*의 즉각 후속자(immediate successor)이다



immediate predecessor *immediate successor*

▶ **페이지 309, 그림 6.38 : AOV 네트워크**

과목 번호	과목명	선수 과목
C1	프로그래밍 I	없음
C2	이산 수학	없음
C3	자료 구조	*C1, C2*
C4	수학 I	없음
C5	수학 II	*C4*
C6	선형 대수	*C5*
C7	알고리즘 분석	*C3, C6*
C8	어셈블리어	*C3*
C9	운영 체제	*C7, C8*
C10	프로그래밍 언어론	*C7*
C11	컴파일러 설계	*C10*
C12	인공 지능	*C7*
C13	계산 이론	*C7*
C14	병렬 알고리즘	*C13*
C15	수치 해석	*C5*


C9
C10
C11
C1
C8
C12
C2
C3
C7
C13
C14
C4
C5
C6
C15

□ 이행적(transitive)

- 모든 세 쌍 i, j, k에 대해 $i \cdot j$이고 $j \cdot k \rightarrow i \cdot k$가 성립하면 관계 ·는 이행적(transitive)이다

□ 비반사적(irreflexive)

- 집합 S의 어떤 원소 x에 대해서도 $x \times x$인 경우가 성립되지 않을 때
  그 집합 S에 대해 비 반사적이다

□ 부분 순서(partial order)

- 이행적이면서 비반사적 관계

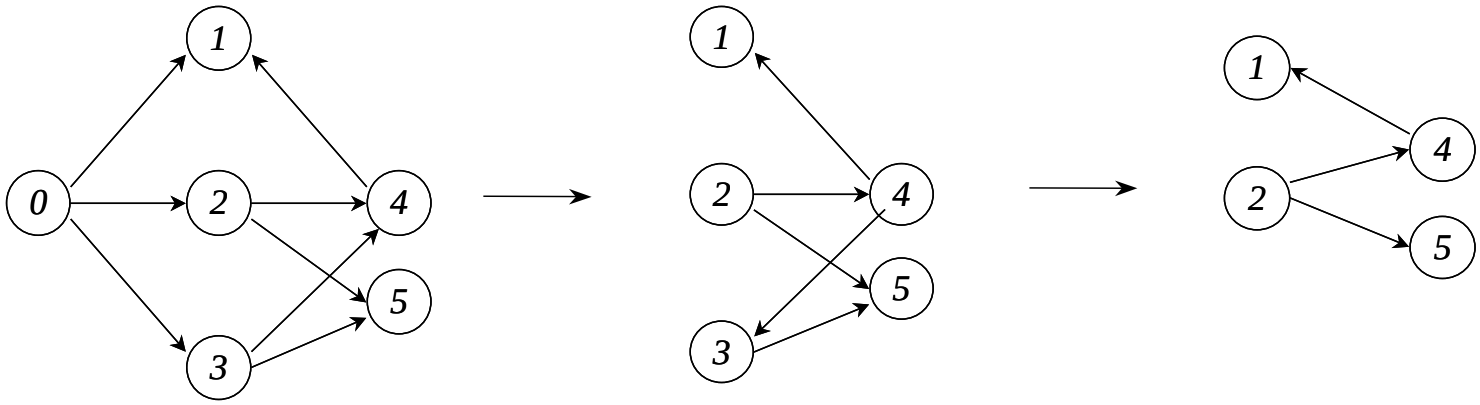
□ 위상 순서 (topological order)

- 네트워크 내에서 정점 i가 정점 j의 선행자일 때 i가 j 앞에 있는 선형 순서 (linear order)
- ✓ C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

▶ 페이지 310, 프로그램 6.13 : 위상 정렬

& 페이지 310, 그림 6.39 : AOV 네트워크에 대한 프로그램 시뮬레이션

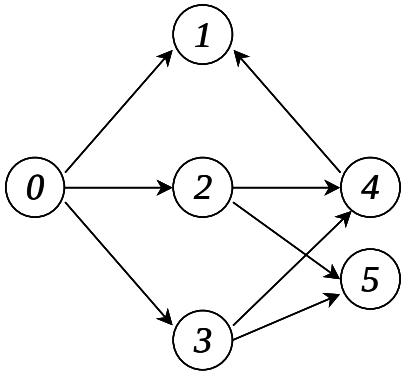
```
for (i = 0; i < n; i++) {
    if (every vertex has a predecessor) {
        fprintf(stderr,"network has a cycle.\n");
        exit(1);
    }
    pick a vertex v that has no predecessors;
    output v;
    delete v and all edges leading out of v from the network;
}
```

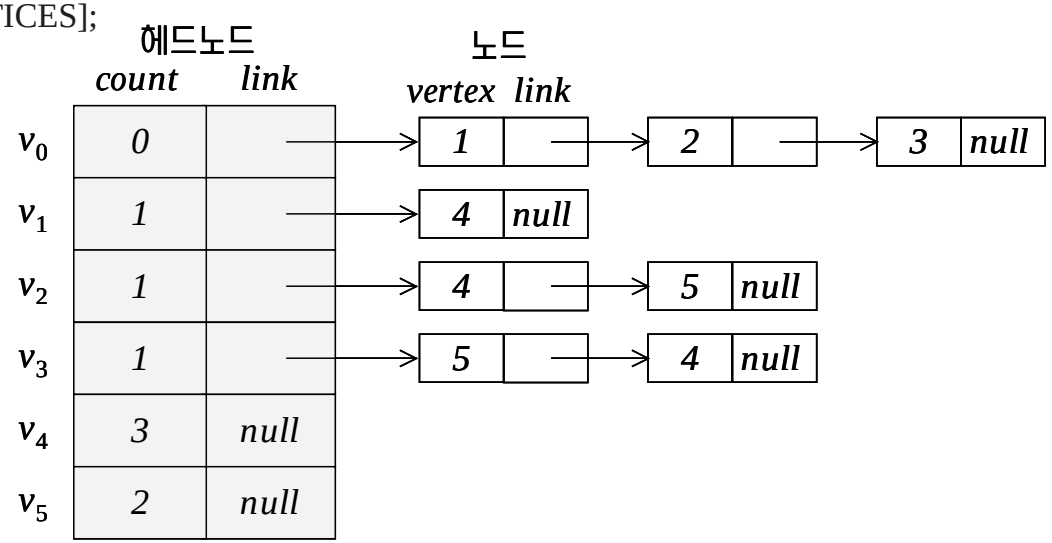


▶ **페이지 311, 네트워크의 인접 리스트 표현**

& 페이지 313, 그림 6.40 : 인접 리스트 표현

```
typedef struct node *node_ptr;
typedef struct node {
    int vertex;
    node_ptr link;
};
typedef struct {
    int count;
    node_ptr link;
} hdnodes;
hdnodes graph[MAX_VERTICES];
```





▶ 페이지 312 ~ 313, 위상 정렬

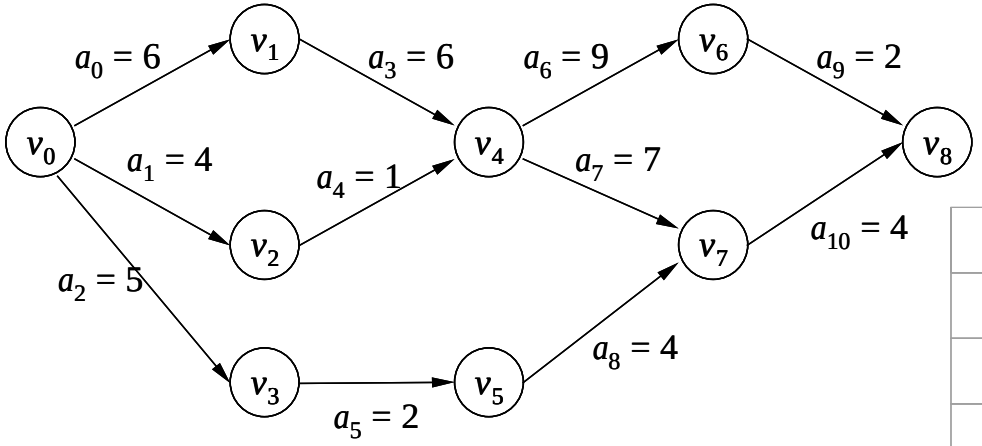
```
void topsort(hdnodes graph[], int n)
{
    int i, j, k, top; node_ptr ptr;
    top = -1;
    for (i = 0; i < n; i++)
        if (!graph[i].count) { graph[i].count = top; top = i; }
    for (i = 0; i < n; i++)
        if (top == -1) { fprintf(stderr,"\nNetwork has a cycle.\n"); exit(1); }
        else {
            j = top; /* unstack a vertex */
            top = graph[top].count;
            print("v%d, ", j);
            for (ptr = graph[j].link; ptr; ptr = ptr->link) {
                k = ptr->vertex;
                graph[k].count--;
                if (!graph[k].count) {    graph[k].count = top; top = k; }
            }
        }
}
```

6.5.2 AOE 네트워크

☐ AOE 네트워크

- activity on edge network
- 방향 간선이 수행되어야 할 작업을 나타내고
  정점이 작업의 완료를 알리는 사건을 나타낸다

▶ **페이지 314, 그림 6.41 : AOE 네트워크**

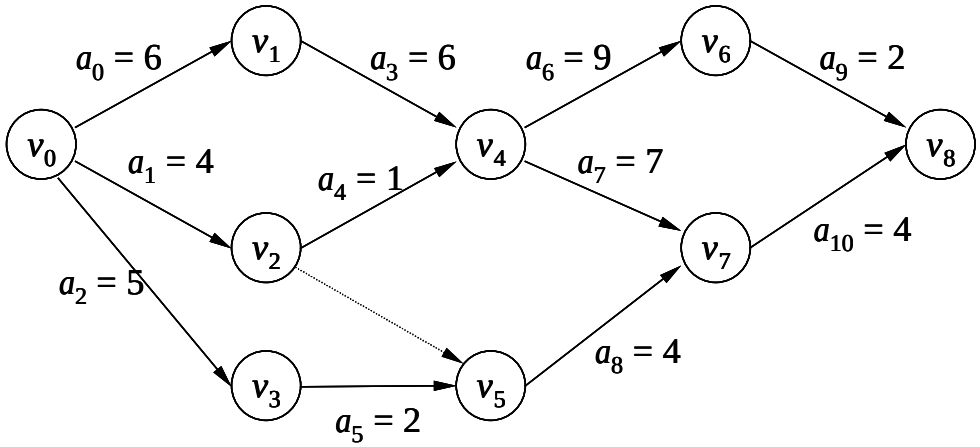


	설명
v_0	프로젝트 시작
v_1	작업 a_0의 완료
v_4	작업 a_3와 a_4의 완료
v_7	작업 a_7와 a_8의 완료
v_8	프로젝트의 완료

□ 가상 작업(dummy activity)

- 수행 시간이 0인 작업
- 작업들에 대해 추가의 제약 조건이 필요한 경우에 사용

✧ **가상 작업**



□ 임계 경로 (critical path)

- 최장 길이의 경로
- 하나의 네트워크는 둘 이상의 임계경로를 가질 수 있다
  - v_0, v_1, v_4, v_7, v_8과 v_0, v_1, v_4, v_6, v_8

□ 가장 이른 시간(earliest time)과 가장 늦은 시간 (latest time)

- earliest time, $e(i)$
  - 시작 정점 0에서 정점 i까지의 최장 경로 길이
- latest time, $l(i)$
  - 프로젝트 기간을 지연시키지 않으면서 가장 늦게 시작할 수 있는 시간

⇒ 임계 작업 (critical activities)

- $e(i) = l(i)$인 작업
- criticality = $l(i) - e(i)$

⇒ 임계 작업의 식별

- AOE 네트워크의 모든 작업들에 대한 $e(i)$와 $l(i)$를 계산

→ 모든 비임계 작업들을 AOE 네트워크에서 삭제한 후, 시작 정점에서 종료 정점까지의 모든 경로를 생성

☐ $e(i)$, $l(i)$ 계산

- $ee[j]$: earliest event time
- $le[j]$: latest event time

→ $ee[j] = \max\{ee[i] + \text{duration of } <i, j>\}$
$le[j] = \min\{le[i] - \text{duration of } <j, i>\}$
$e(i) = ee(k)$
$l(i) = le[l] - \text{duration activity } a_i$

▶ **페이지 317, 그림 6.42 : 위상 정렬로부터 *earliest* 값의 계산**

earliest	*[0]*	*[1]*	*[2]*	*[3]*	*[4]*	*[5]*	*[6]*	*[7]*	*[8]*	*stack*
initial	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*[0]*
output v_0	*0*	*6*	*4*	*5*	*0*	*0*	*0*	*0*	*0*	*[3,2,1]*
output v_3	*0*	*6*	*4*	*5*	*0*	*7*	*0*	*0*	*0*	*[5,2,1]*
output v_5	*0*	*6*	*4*	*5*	*0*	*7*	*0*	*11*	*0*	*[2,1]*
output v_2	*0*	*6*	*4*	*5*	*5*	*7*	*0*	*11*	*0*	*[1]*
output v_1	*0*	*6*	*4*	*5*	*7*	*7*	*0*	*11*	*0*	*[4]*
output v_4	*0*	*6*	*4*	*5*	*7*	*7*	*16*	*14*	*0*	*[7,6]*
output v_7	*0*	*6*	*4*	*5*	*7*	*7*	*16*	*14*	*18*	*[6]*
output v_6	*0*	*6*	*4*	*5*	*7*	*7*	*16*	*14*	*18*	*[8]*
output v_8										

▶ **페이지 319, 그림 6.43 : AOE 네트워크에 대한 *latest* 값의 계산**

latest	*[0]*	*[1]*	*[2]*	*[3]*	*[4]*	*[5]*	*[6]*	*[7]*	*[8]*	*stack*
initial	*18*	*18*	*18*	*18*	*18*	*18*	*18*	*18*	*18*	*[8]*
output v_8	*18*	*18*	*18*	*18*	*18*	*18*	*16*	*14*	*18*	*[7,6]*
output v_7	*18*	*18*	*18*	*18*	*7*	*10*	*16*	*14*	*18*	*[5,6]*
output v_5	*18*	*18*	*18*	*18*	*7*	*10*	*16*	*14*	*18*	*[3,6]*
output v_3	*3*	*18*	*18*	*8*	*7*	*10*	*16*	*14*	*18*	*[6]*
output v_6	*3*	*18*	*18*	*8*	*7*	*10*	*16*	*14*	*18*	*[4]*
output v_4	*3*	*6*	*6*	*8*	*7*	*10*	*16*	*14*	*18*	*[2,1]*
output v_2	*2*	*6*	*6*	*8*	*7*	*10*	*16*	*14*	*18*	*[1]*
output v_1	*0*	*6*	*6*	*8*	*7*	*10*	*16*	*14*	*18*	*[0]*

$latest[8] = earliest[8] = 18$

$latest[6] = min\{latest[8] - 2\} = 16$

$latest[7] = min\{latest[8] - 4\} = 14$

$latest[4] = min\{latest[6] - 9; latest[7] - 7\} = 0$

$latest[1] = min\{latest[4] - 1\} = 6$

$latest[2] = min\{latest[4] - 1\} = 6$

$latest[5] = min\{latest[7] - 4\} = 10$

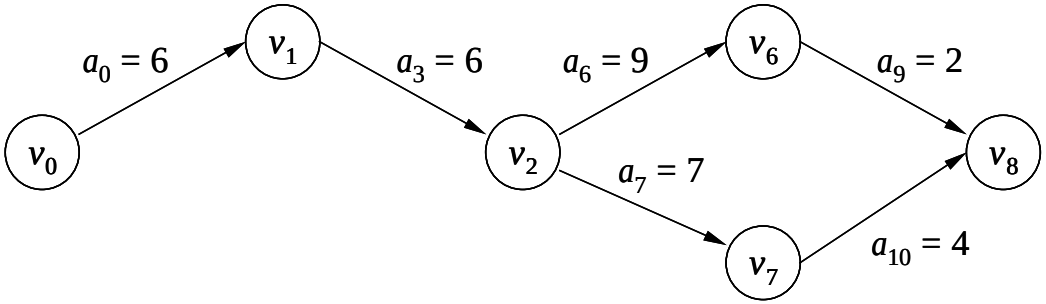
$latest[3] = min\{latest[5] - 2\} = 8$

$latest[0] = min\{latest[1] - 6; latest[2] - 4; latest[3] - 5\} = 0$

□ 네트워크에서 모든 비 임계 작업을 제거한 방향 그래프

- 이 그래프 상의 경로가 아닌 것이 원래 그래프 상의 임계 경로인 것은 없다

✧ **페이지 320, 그림 6.45 : 비임계 작업 삭제 후의 그래프**



□ TOPOLOGICAL-ORDER 알고리즘은 네트워크와 방향 사이클만 탐지할 수 있다

✧ **페이지 321, 그림 6.45 : 도달 불가능한 작업을 지닌 AOE 네트워크**