제5강 그래프 알고리즘

김 승 주

정보보호연구소

성균관대학교 정보통신공학부

http://www.security.re.kr/

그래프

그래프

- 현상이나 사물을 정점(vertex)과 간선(edge)으로 표현한 것
- 그래프 G = (V, E)
  - V : 정점 집합
  - E : 간선 집합
- 두 정점이 간선으로 연결되어 있으면 인접하다고 한다
  - 인접 = adjacent
  - 간선은 두 정점의 관계를 나타낸다

그래프 용어 정의 (1/3)

- 경로(path): 간선에 의해 연속적으로 연결된 정점의 리스트 혹은 간선의 리스트.
- 단순경로(simple path): 동일 정점이 중복되어 나타나지 않는 경로.
- 연결그래프: 모든 정점간에 경로가 존재하는 그래프
- 연결요소(connected component): 그래프 내의 연결된 부분그래프

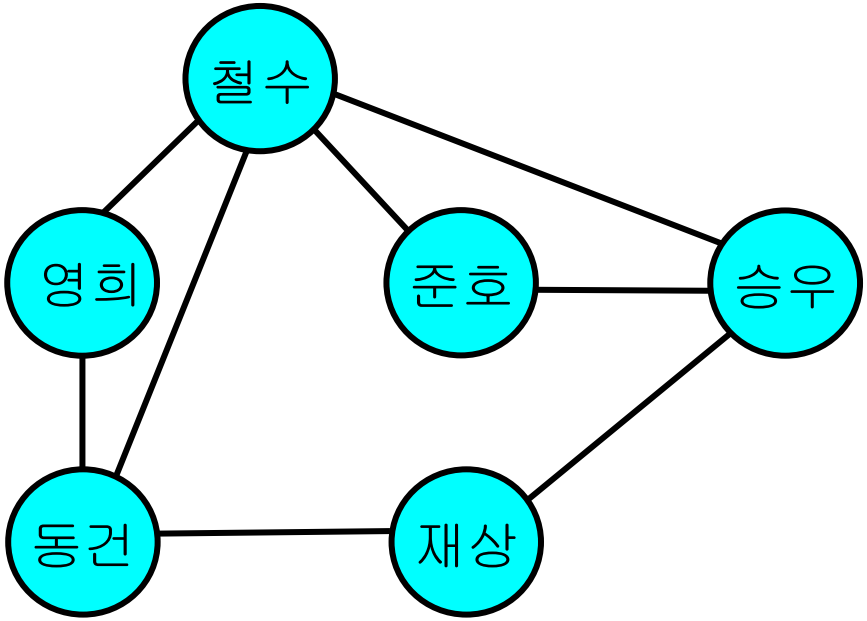
그래프 용어 정의 (2/3)

- 사이클(cycle): 첫째와 마지막 정점이 동일한 경로. 회로(circuit)라고도 함.
- 트리(tree): 사이클이 없는 연결된 그래프.
- 숲(forest): 여러 나무들로 이루어진 그래프.
- 신장트리(spanning tree): 그래프의 모든 정점을 포함하고 있는 트리.
- 완전 그래프(complete graph): 있을 수 있는 모든 간선을 갖는 그래프.

그래프 용어 정의 (3/3)

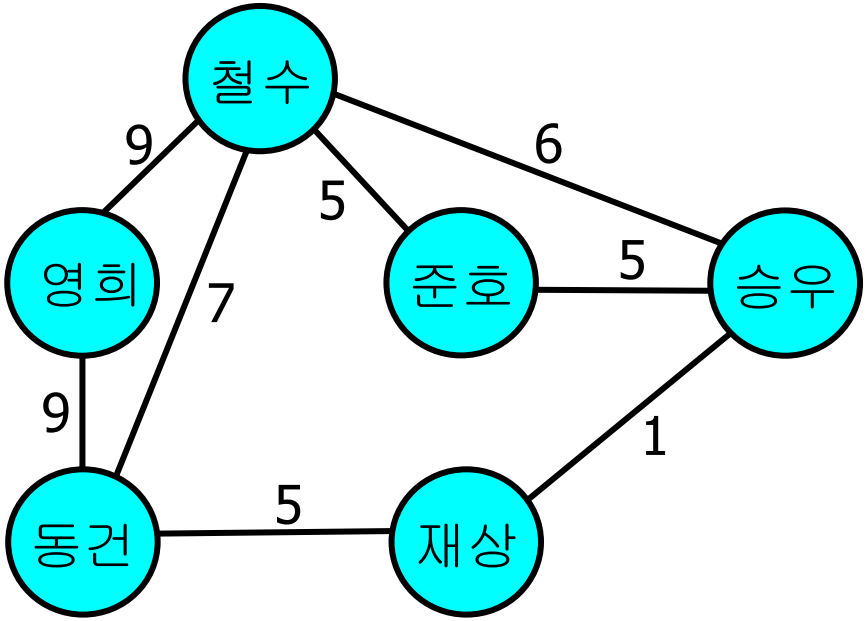
- 가중 그래프(weighted graph): 간선에 가중치가 주어진 그래프.
- 방향 그래프(directed graph): 간선에 방향이 부여된 그래프.
- 네트워크(network): 가중 방향 그래프

그래프의 예 (1/4)



사람들간의 친분 관계를 나타낸 그래프

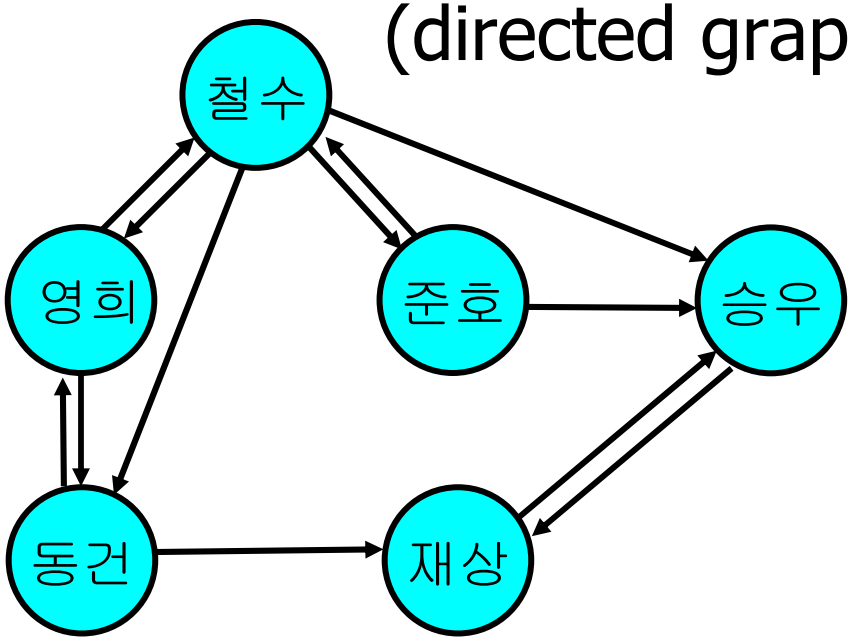
그래프의 예 (2/4)



친밀도를 가중치로 나타낸 친분관계 그래프

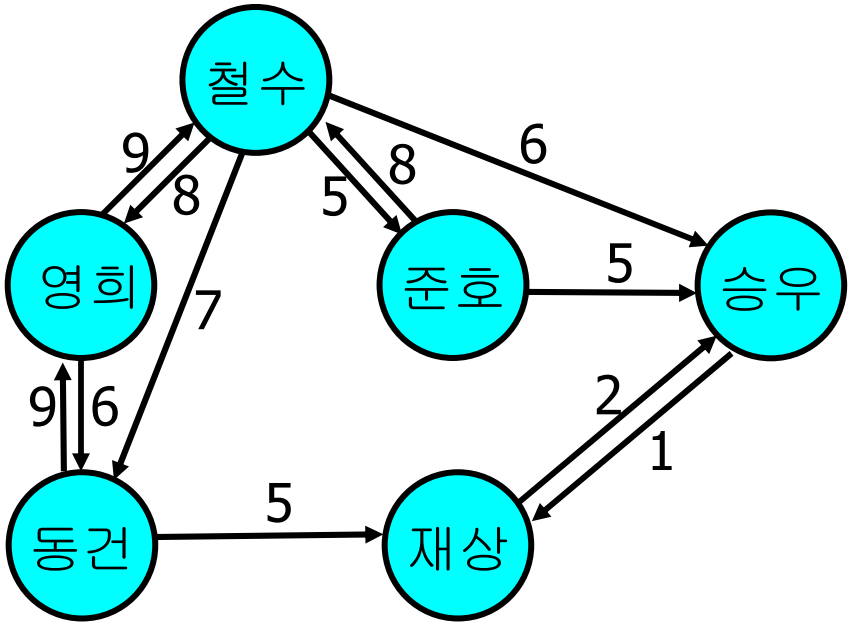
그래프의 예 (3/4)

유향 그래프
(directed graph = digraph)



방향을 고려한 친분관계 그래프

그래프의 예 (4/4)



가중치를 가진 유향 그래프

그래프의 표현

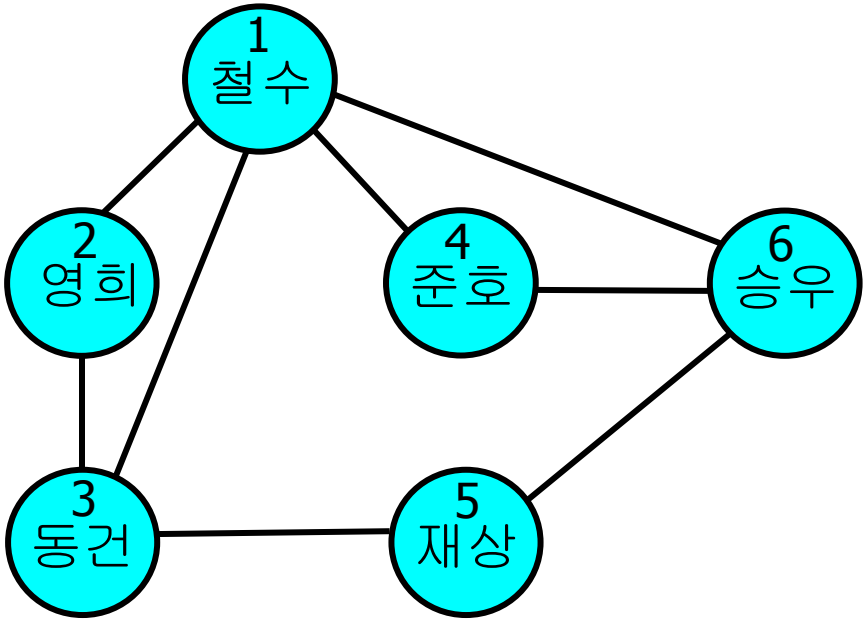
- 인접 행렬 표현법 – |V|*|V| 행렬. $O(|V|^2)$의 수행시간. 조밀한 그래프에 적절.

- 인접 리스트 표현법 – 연결 리스트. 성긴 그래프에 적절.

인접 행렬 표현법 (1/7)

- |V| X |V| 행렬로 표현
  - 원소 (i, j) = 1 : 정점 i 와 정점 j 사이에 간선이 있음
  - 원소 (i, j) = 0 : 정점 i 와 정점 j 사이에 간선이 없음
- 유향 그래프의 경우
  - 원소 (i, j)는 정점 i로부터 정점 j 로 연결되는 간선이 있는지를 나타냄
- 가중치 있는 그래프의 경우
  - 원소 (i, j)는 1 대신에 가중치를 가짐

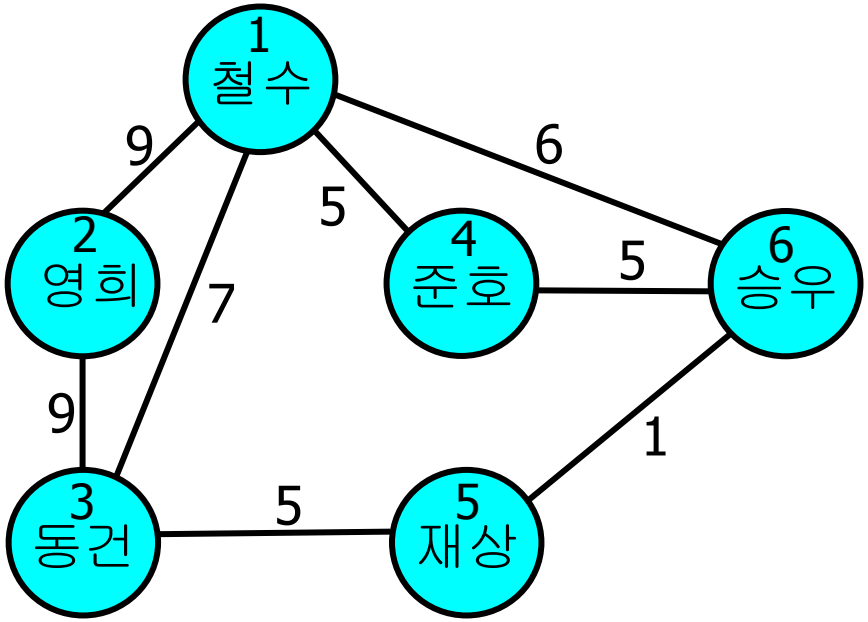
인접 행렬 표현법 (2/7)



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

무향 그래프의 예

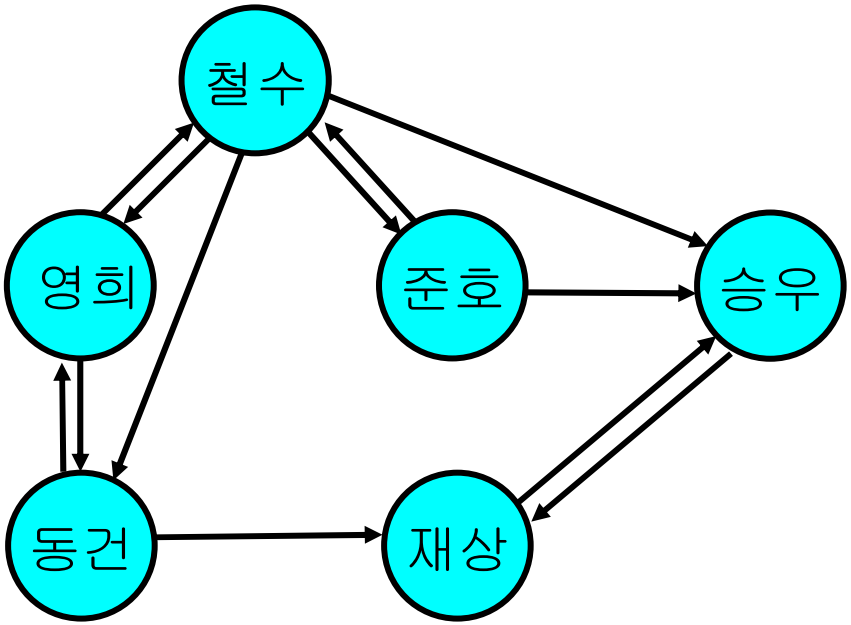
인접 행렬 표현법 (3/7)



	1	2	3	4	5	6
1	0	9	7	5	0	6
2	9	0	9	0	0	0
3	7	9	0	0	5	0
4	5	0	0	0	0	5
5	0	0	5	0	0	1
6	6	0	0	5	1	0

가중치 있는 무향 그래프의 예

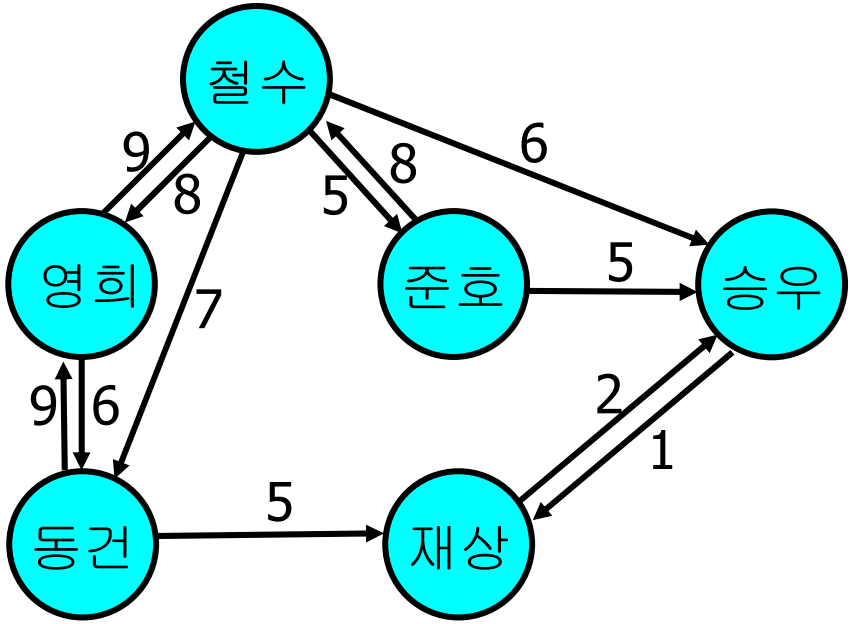
인접 행렬 표현법 (4/7)



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

유향 그래프의 예

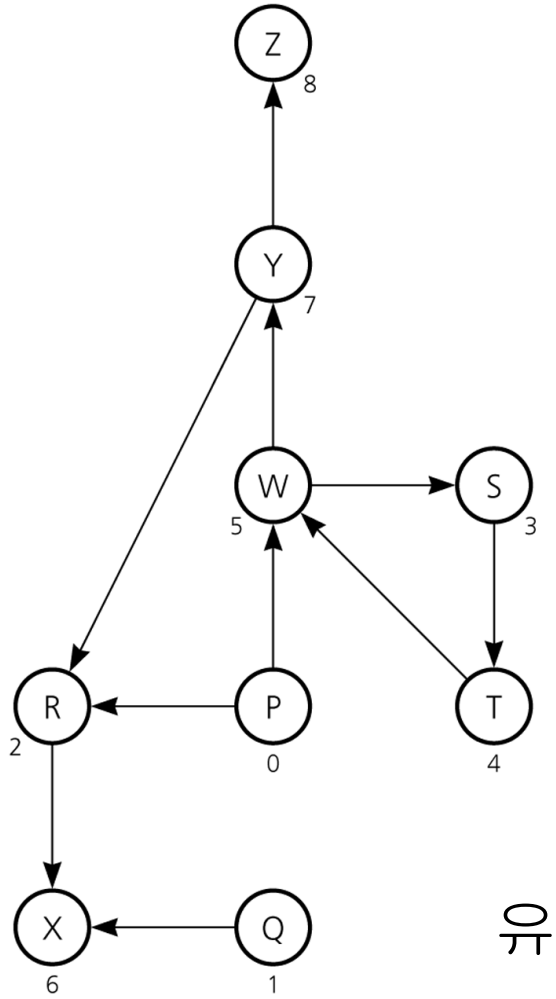
인접 행렬 표현법 (5/7)



	1	2	3	4	5	6
1	0	8	7	5	0	6
2	9	0	6	0	0	0
3	0	9	0	0	5	0
4	8	0	0	0	0	5
5	0	0	0	0	0	2
6	0	0	0	0	1	0

가중치 있는 유향 그래프의 예

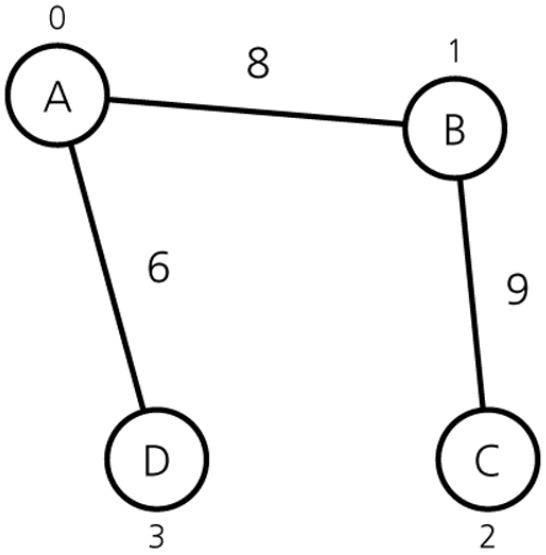
인접 행렬 표현법 (6/7)



		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

유향 그래프의 다른 예

인접 행렬 표현법 (7/7)



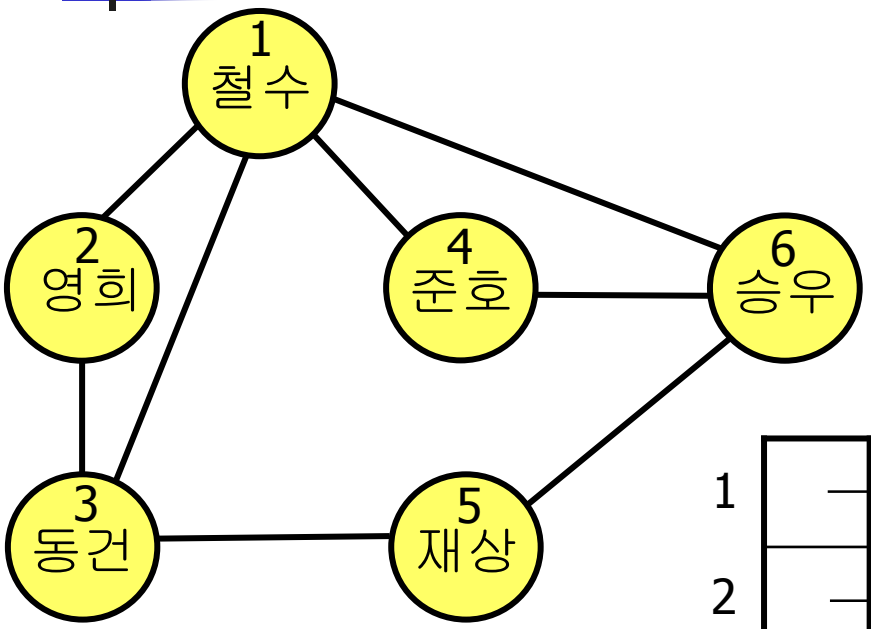
		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

가중치 있는 그래프의 다른 예

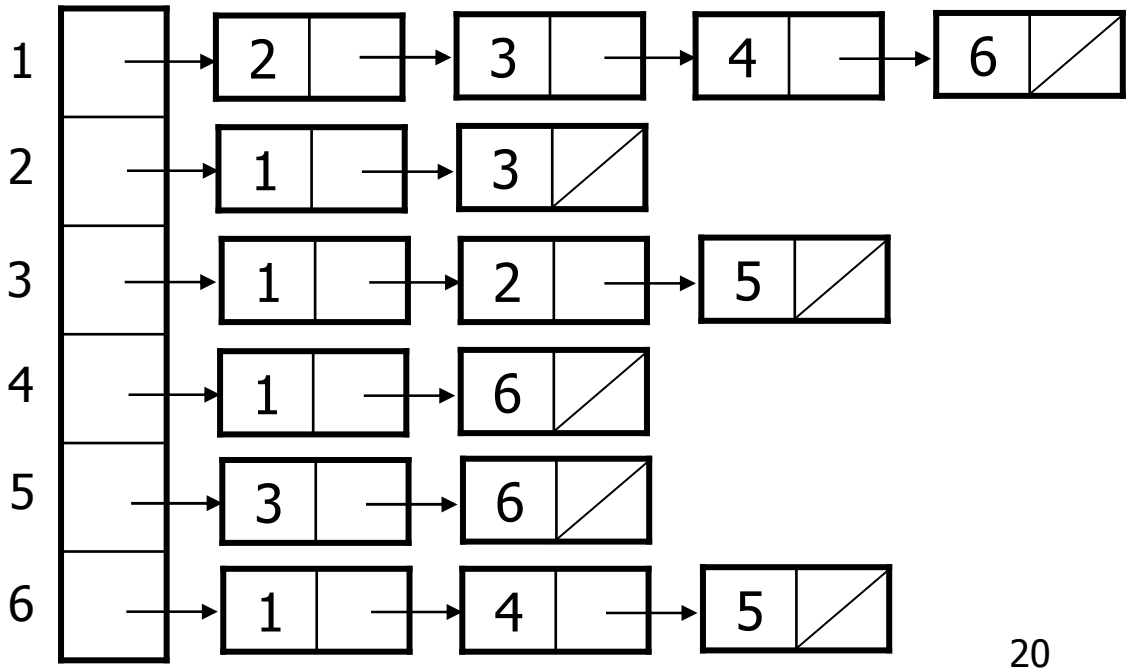
인접 리스트 표현법 (1/3)

- |V|개의 연결 리스트로 표현
- i번째 리스트는 정점 i에 인접한 정점들을 리스트로 연결해 놓음
- 가중치 있는 그래프의 경우
  - 리스트는 가중치도 보관한다

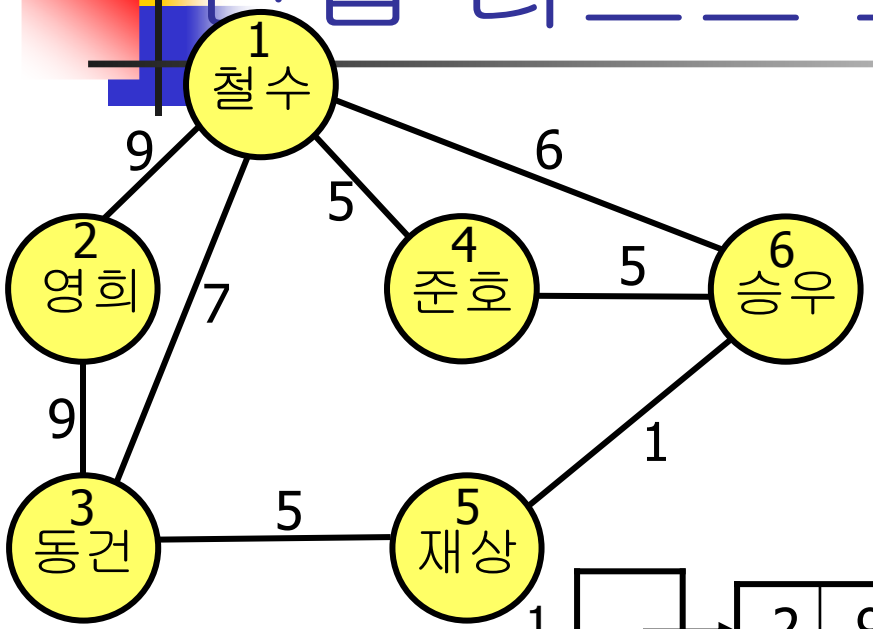
인접 리스트 표현법 (2/3)



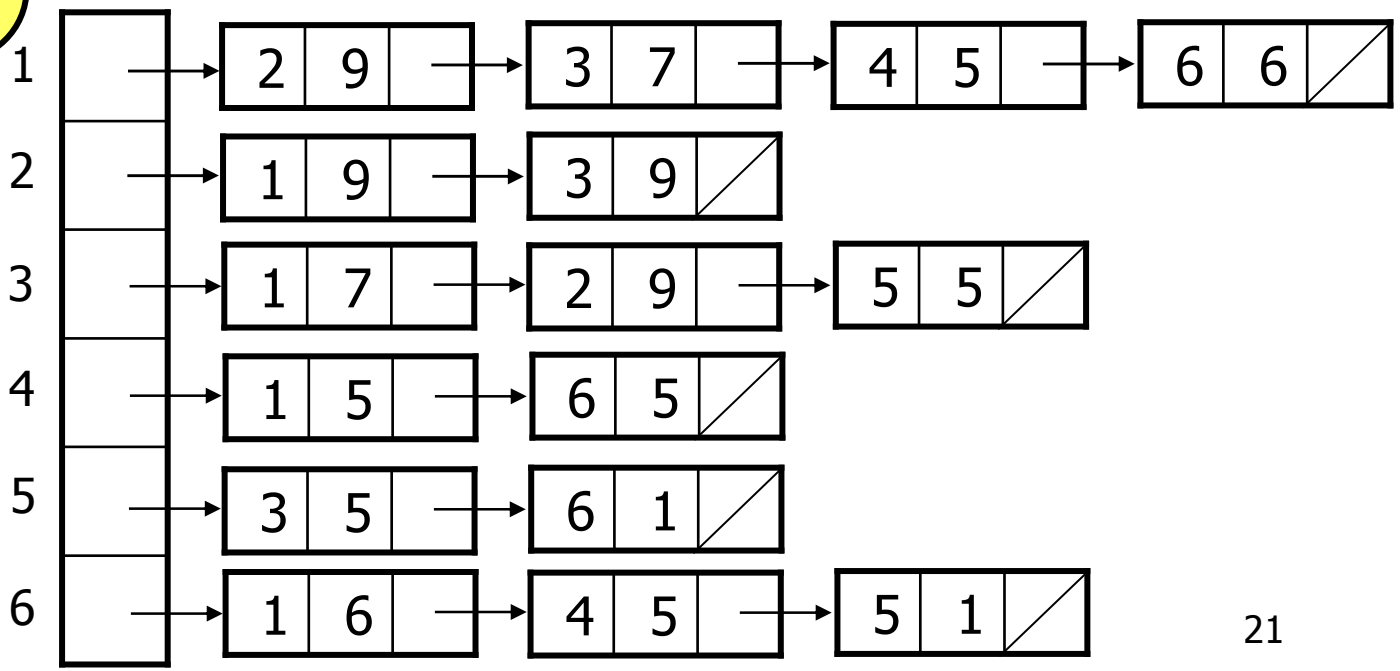
무향 그래프의 예

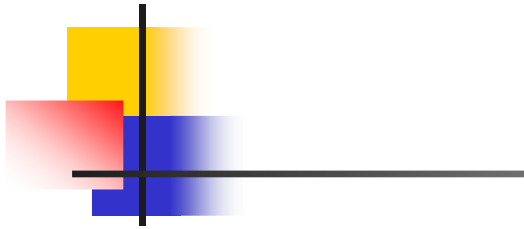


인접 리스트 표현법 (3/3)



가중치 있는 그래프의 예



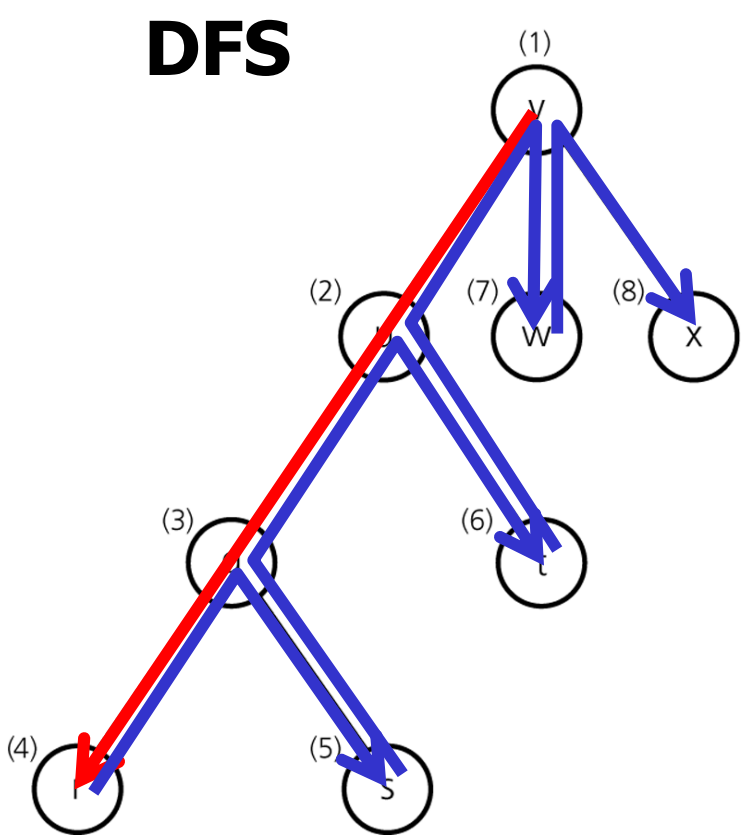


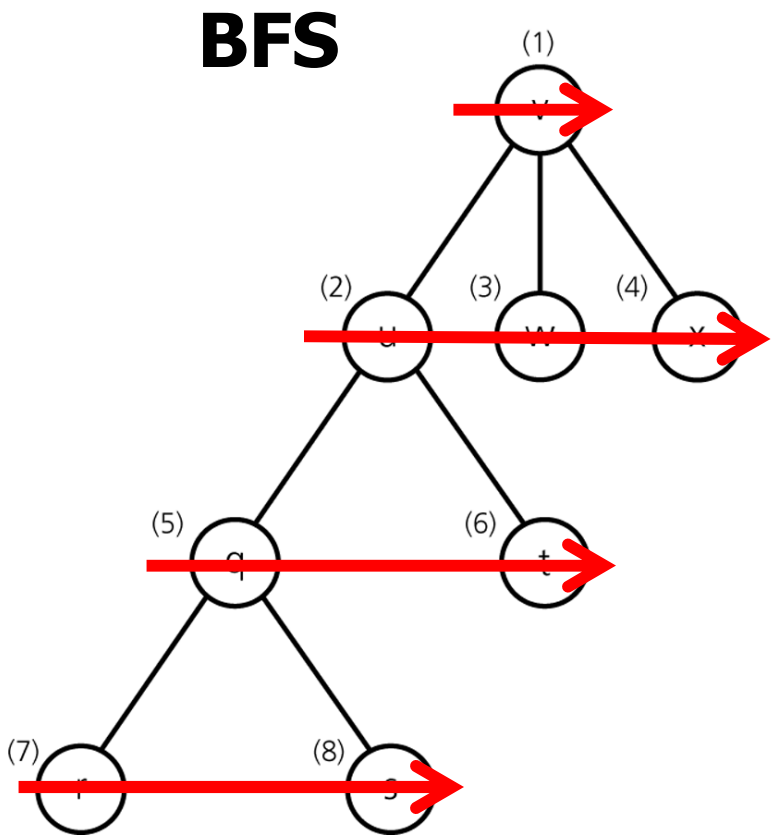
그래프의 순회

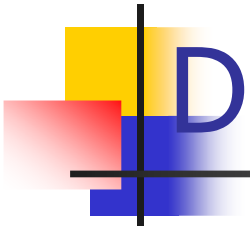
그래프 순회

- 그래프의 모든 정점을 탐색하는 대표적인 방법
  - 깊이 우선 탐색(DFS: Depth-First Search)
  - 너비 우선 탐색(BFS: Breadth-First Search)
- 탐색 과정에서 정점의 분류
  - 방문 정점: 이미 방문한 정점
  - 주변 정점: 방문 정점에 인접한 아직 방문하지 않은 정점
  - 미도달 정점: 방문 정점도 주변 정점도 아닌 전혀 접근하지 못한 정점
- 주변정점들 중 어떤 한 정점 x를 방문하여 방문 정점으로 만들고, x에 인접한 모든 미방문 정점을 주변정점으로 변경한다.

DFS v.s. BFS







DFS (1/4)

- 정점을 방문할 때 갈 수 있는 데까지 우선 가보다가 더 이상 진행할 수 없으면 왔던 길을 거슬러 올라가면서 아직 가보지 않은 길이 있으면 그 길을 따라 또 갈 수 있는 데까지 가 보는 방법.
  - 최근의 방문 정점에 인접한 주변 정점을 먼저 방문.
- 스택으로 구현하는 것이 적절.

DFS (2/4)

```
#define n 100    /* 정점의 개수  n = 100 */
struct node {
    int vertex;
    struct node *next ;
};
struct node *head[n];
```

<그림 6-7> 인접 리스트 자료 구조

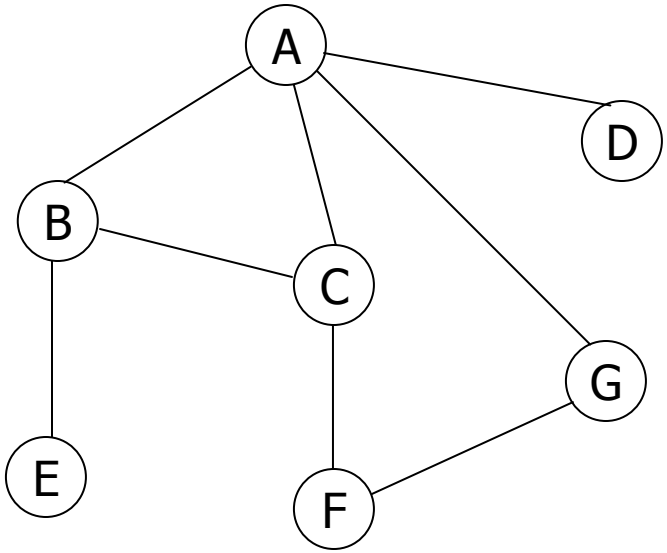
DFS (3/4)

```
int mark, flag[n] ;

dfs(G) {
/* 입력: 인접 리스트 표현의 그래프 G = (V, E)
   출력: 모든 정점에 대한 DFS 방문결과.  배열 flag에 방문순서 기록. */
1       int v;
2       mark = 0;
3       for (v = 0; v < n; v++)  /* flag의 초기화 */
4             flag[v] = 0;
5       for (v=0; v <n; v++)
6       if (flag[v] == 0) dfs_visit (v);
}
```

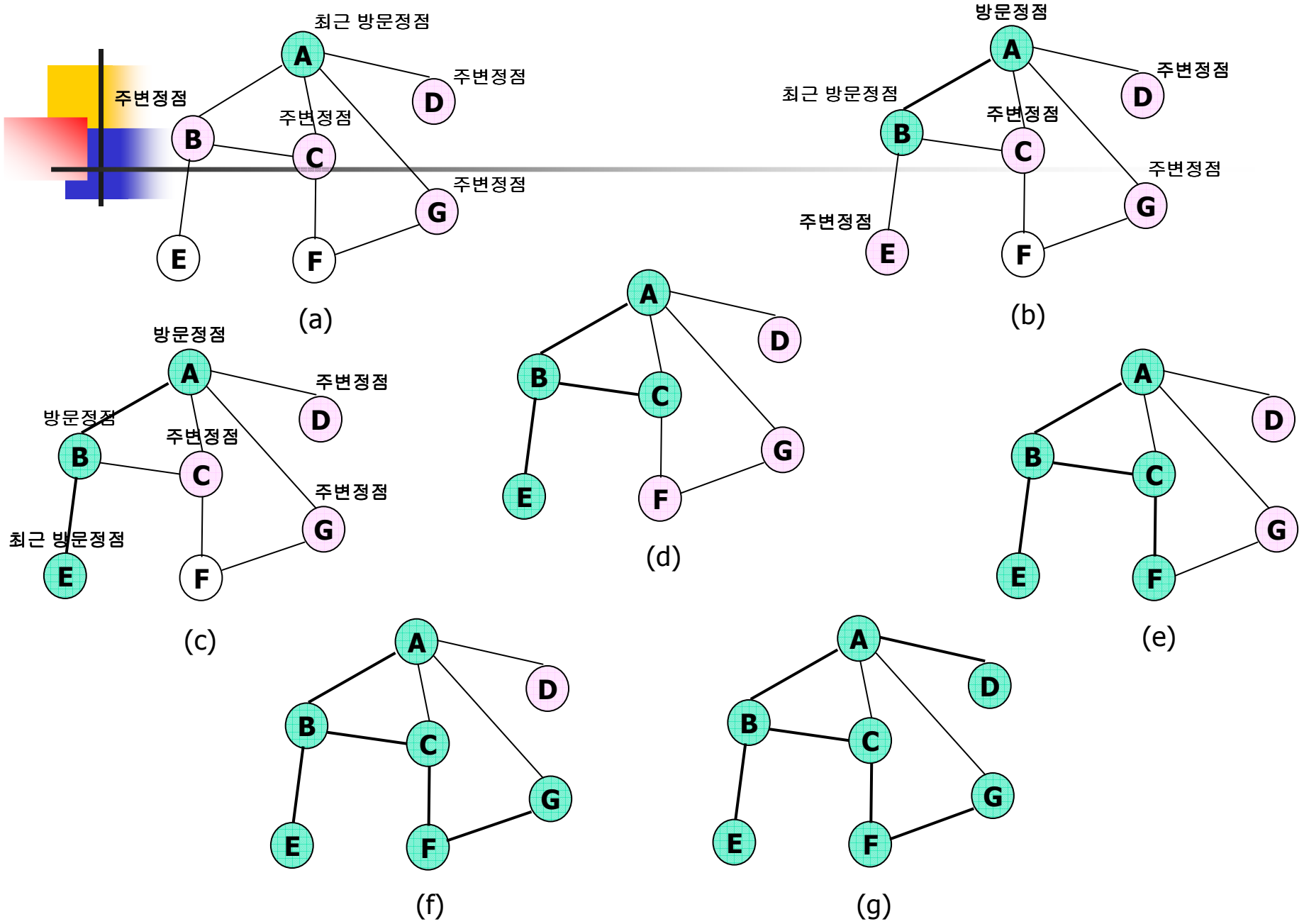
DFS (4/4)

```
dfs_visit(int v) {
/* 입력 : 정점 v
   출력 : v가 속한 연결성분의 순환적 깊이 우선 탐색 */

1       struct node *t ;
2       mark++; flag[v] = mark ;
3       t = head;
4       while(t != NULL)  {
5          if(flag[t->vertex] == 0)  dfs_visit(t->vertex) ;
6          t = t->next ;
7       }
}
```

DFS의 작동 예

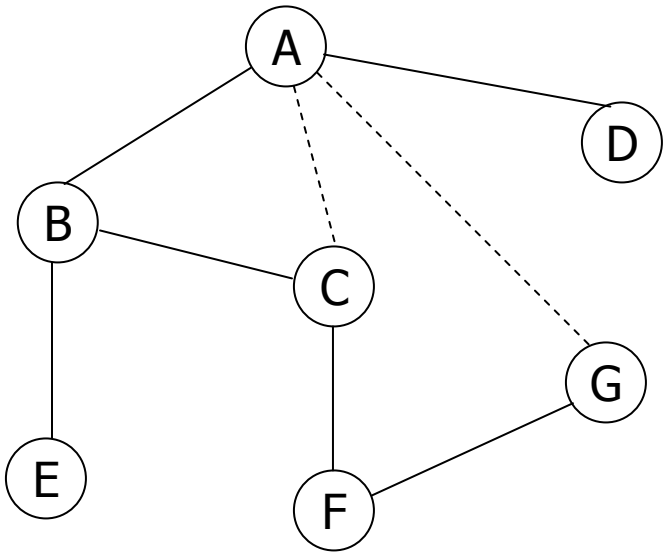


<그림 6-5> 그래프 예

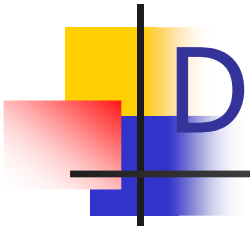


<그림 **6-6**> 깊이 우선 탐색 과정

깊이 우선 트리



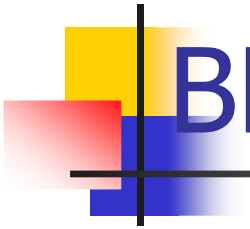
<그림 6-9> 깊이 우선 트리



DFS의 수행시간 분석

- 인접 리스트 표현 – $\Theta(|V|+|E|)$
- 인접 행렬 표현 – $O(|V|^2)$

BFS (1/3)



- 가장 오래된 주변정점을 먼저 방문. 즉, 거리 순으로 방문.
  - 예를 들어 루트를 시작으로 탐색을 한다면 먼저 루트의 자식을 차례로 방문한다. 다음으로 루트 자식의 자식, 즉 루트에서 두 개의 간선을 거쳐서 도달할 수 있는 정점을 방문한다. 다음으로 루트로부터 세 개의 간선을 거쳐서 도달하는 정점들 … 순으로 루트에서의 거리 순으로 방문한다.
- 큐로 구현하는 것이 적절
- 너비 우선 나무에서 시작 정점으로부터 해당 정점까지의 경로는 시작 정점에서 그 정점까지의 최단 경로에 해당.

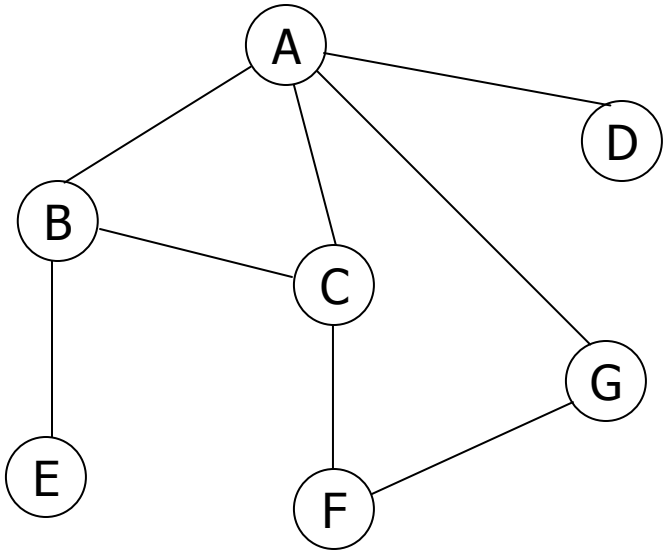
BFS (2/3)

```
int flag[n], mark ;

bfs(G) {
/* 입력 : 인접 리스트 표현의 그래프 G = (V, E)
   출력 : 모든 정점에 대한 BFS탐색 결과.  배열 flag에 방문 순서 기록. */
1       int v ;
2       mark = 0 ;
3       init_queue( ) ;
4       for (v = 0; v < n; v++)     /* flag의 초기화 */
5           flag[v] = 0 ;
6       for (v = 0; v < n; v++)
7           if (flag[v] == 0)  bfs_visit (v) ;
}
```

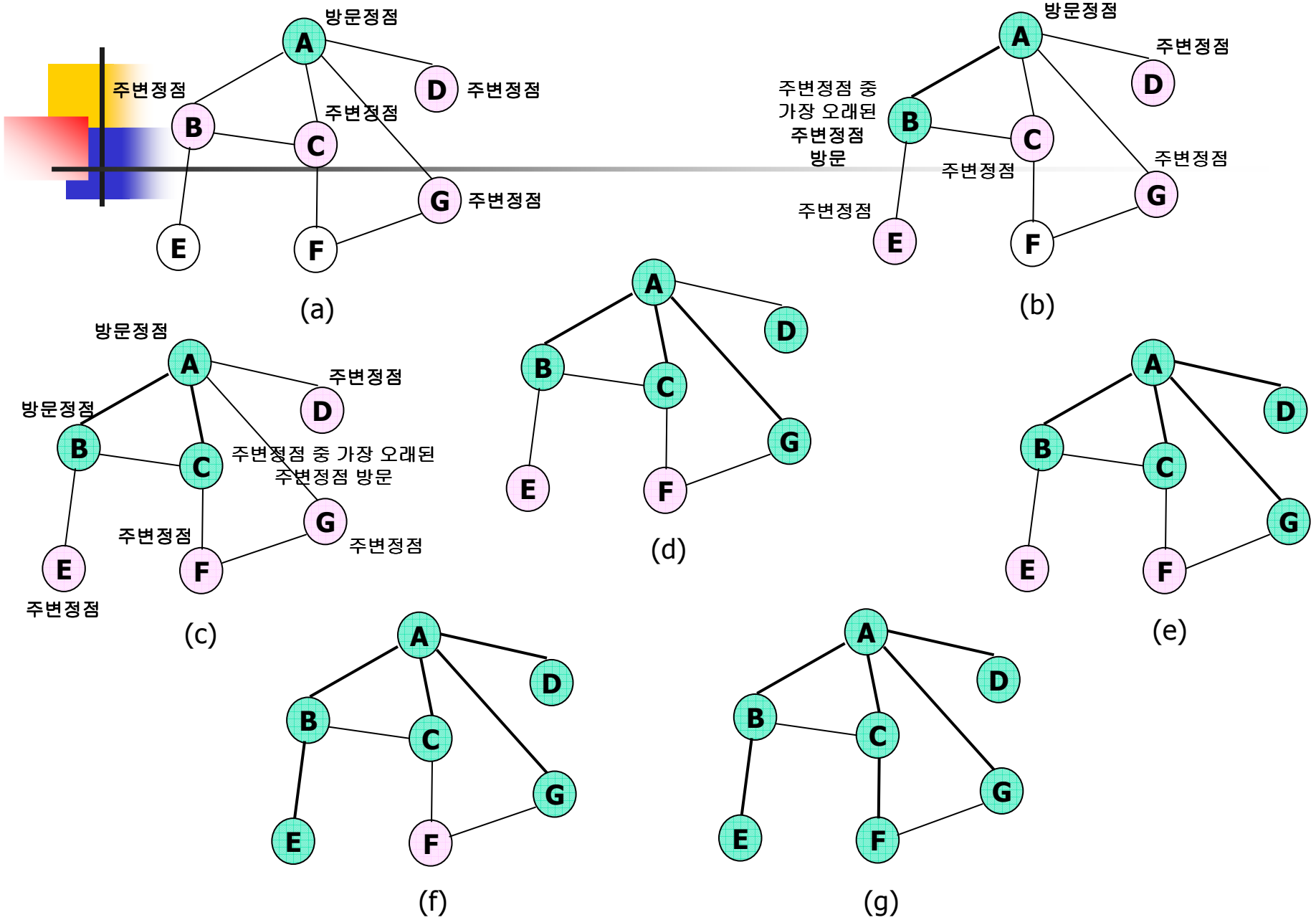
BFS (3/3)

```
bfs_visit(int v) {
/* 입력 : 정점 번호 v
   출력 : v가 속한 연결 성분의 너비 우선 방문 순서를 flag 배열에. */
1      struct node *t ;  int u ;
2      enqueue(v); flag[v] = -1 ;
3      while (!queue_empty( ) )  {
4          u = delqueue ( );
5          mark ++ ;  flag[u] = mark ;
6          for (t = head[u];  t != NULL;  t -> next)
7              if (flag[t->vertex] == 0)  {
8                  enqueue(t-> vertex) ;
9                  flag[t->vertex] =- 1 ;
10             }
11     }
 }
```

BFS의 작동 예

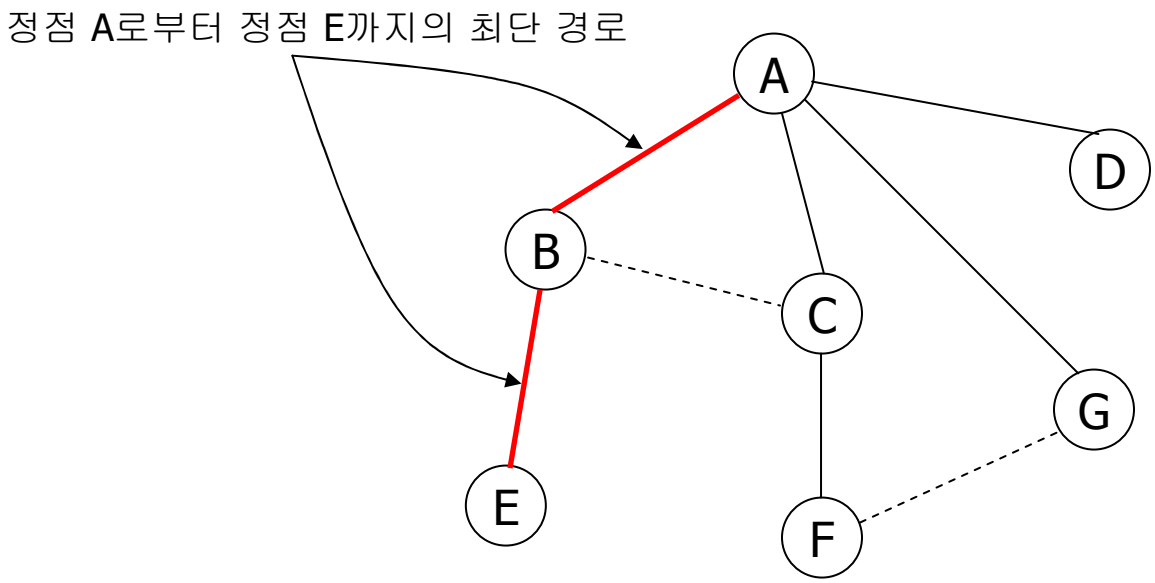


<그림 6-5> 그래프 예

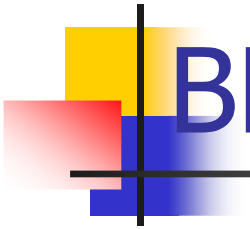


<그림 6-10> 너비 우선 탐색 과정

너비 우선 나무



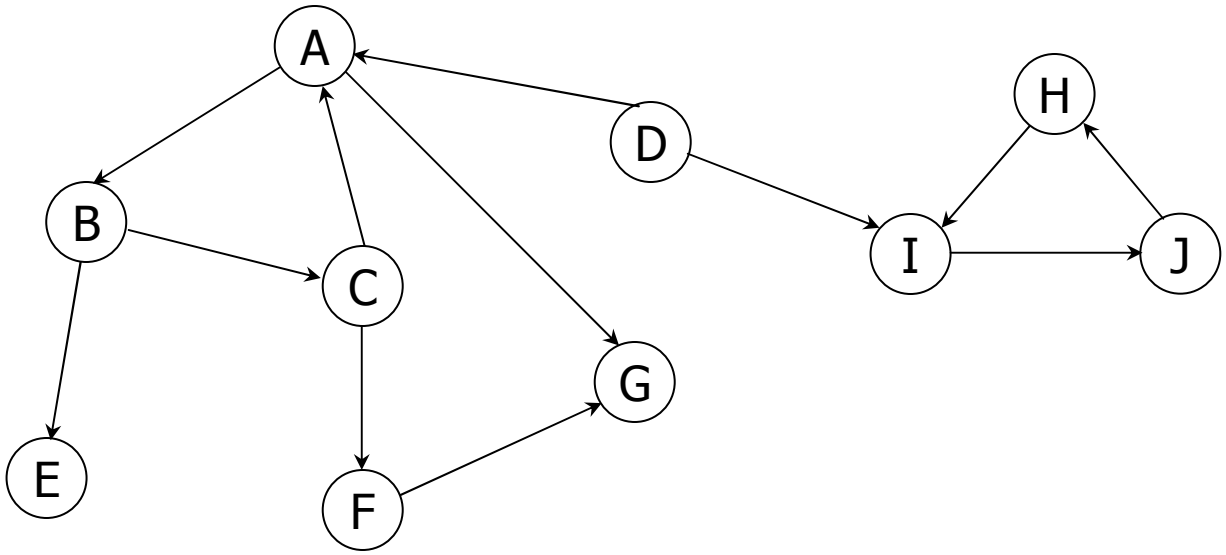
<그림 6-12> 너비 우선 트리



BFS의 수행시간 분석

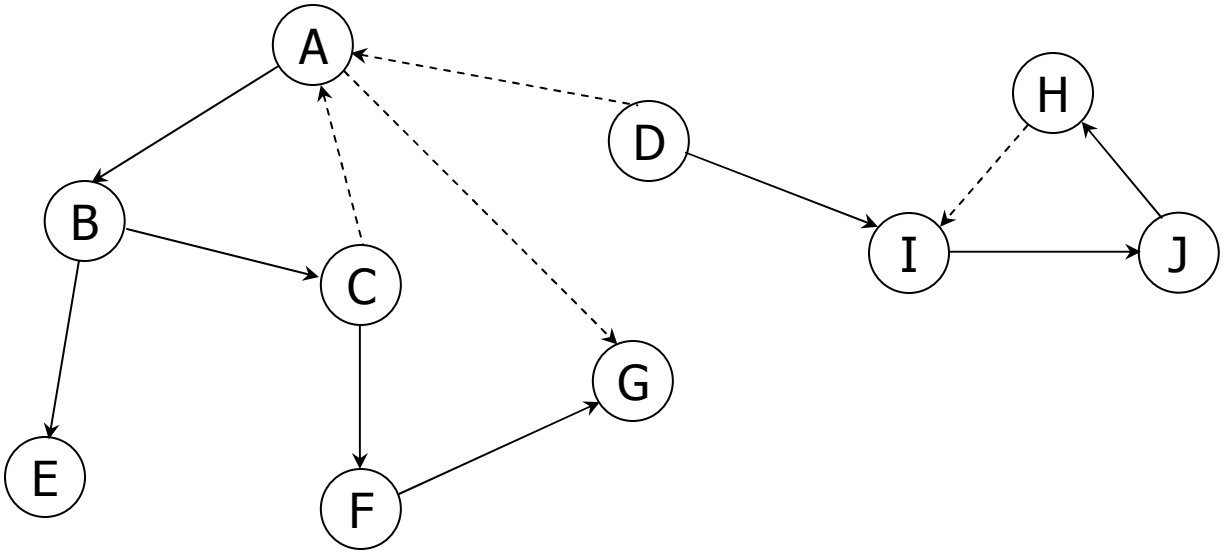
- 인접 리스트 표현 – $\Theta(|V|+|E|)$
- 인접 행렬 표현 – $O(|V|^2)$

방향 그래프

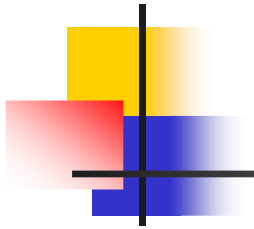


<그림 6-13> 방향 그래프

방향그래프의 깊이 우선 숲



<그림 6-14> 방향 그래프에 대한 깊이 우선 숲

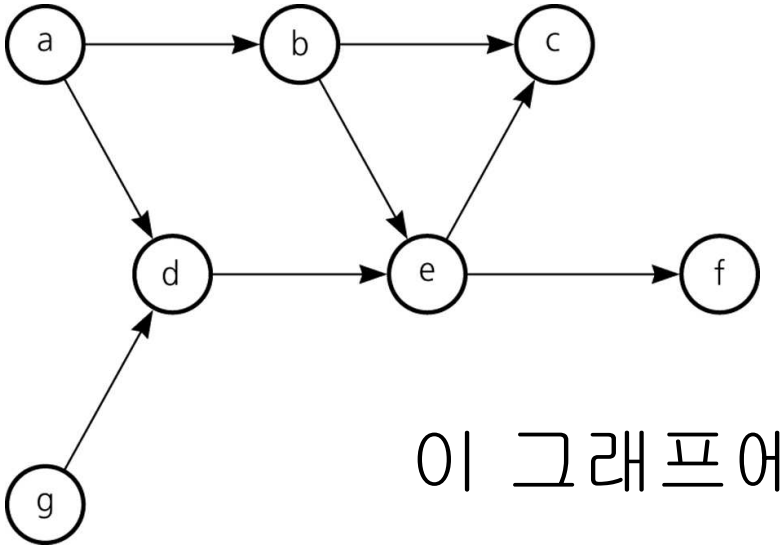


위상정렬

위상정렬 (Topological Sorting)

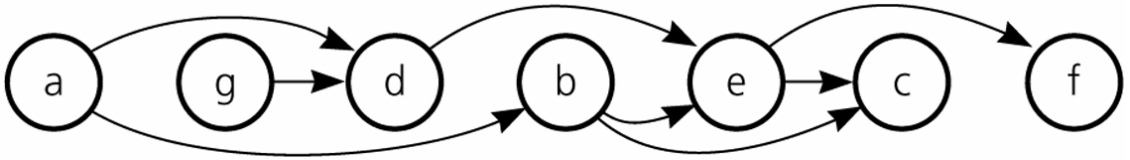
- 조건
  - 싸이클이 없는 유향 그래프 (Directed Acyclic Graph, DAG)
- 위상정렬
  - 모든 정점을 일렬로 나열하되
  - 정점 x에서 정점 y로 가는 간선이 있으면 x는 반드시 y보다 앞에 위치한다
  - 일반적으로 임의의 유향 그래프에 대해 복수의 위상 순서가 존재한다

위상정렬의 예

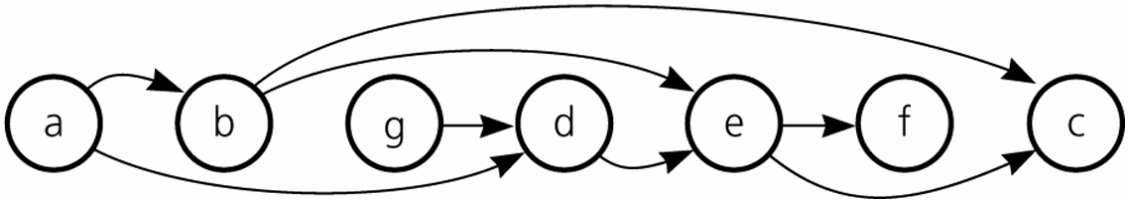


이 그래프에 대한 위상정렬의 예 2개

(a)



(b)



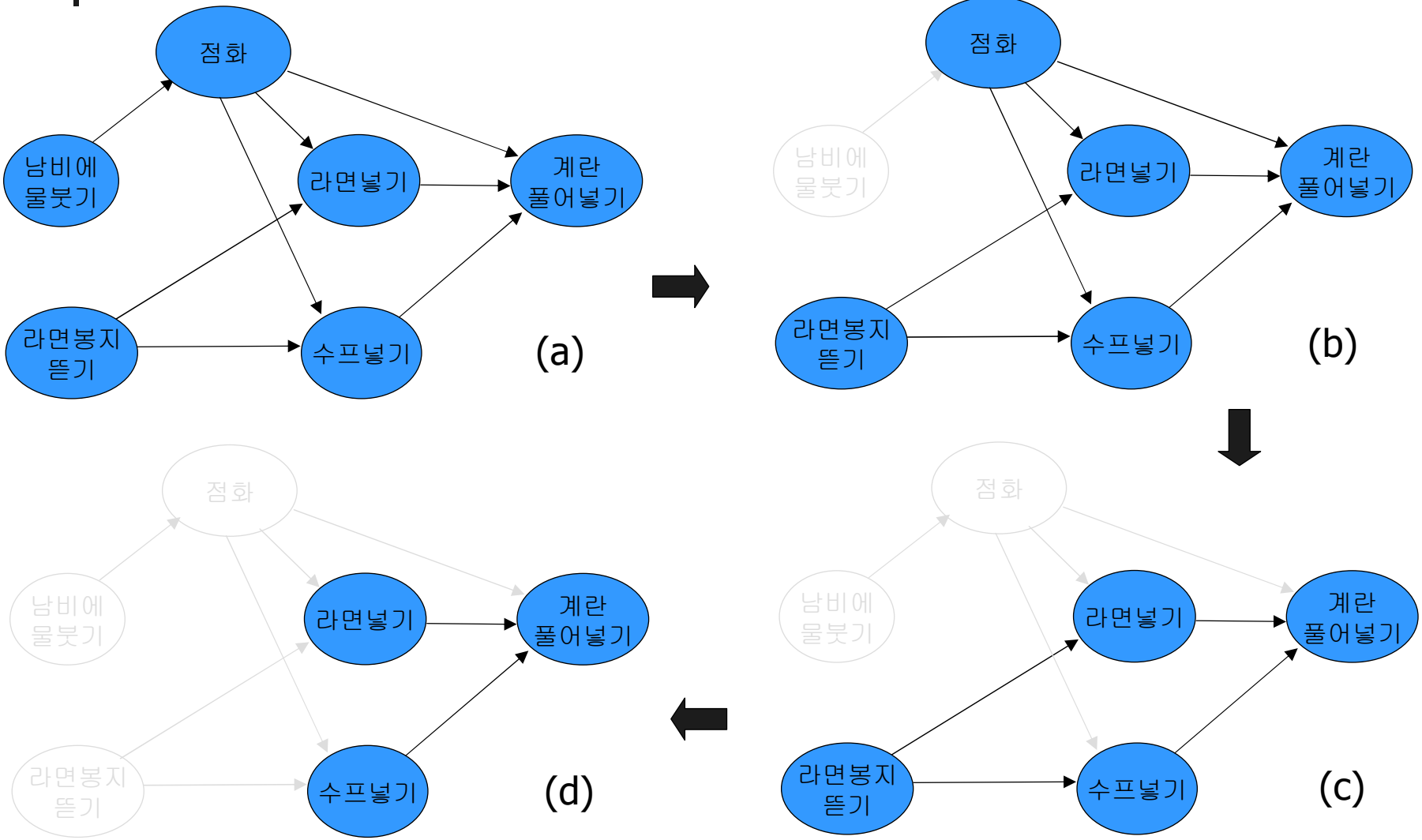
위상정렬 알고리즘 1

Algorithm 1 [Topological Sort]

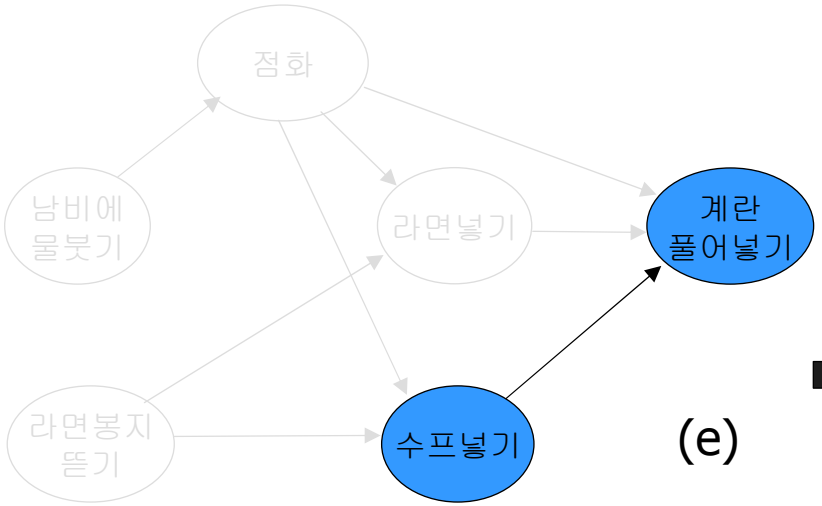
- Find a node v with in-degree zero; make v be the first element of the schedule.
- Delete v and its incident edges from the graph. Schedule recursively the remaining vertices.

Time : $\Theta(|V|+|E|)$

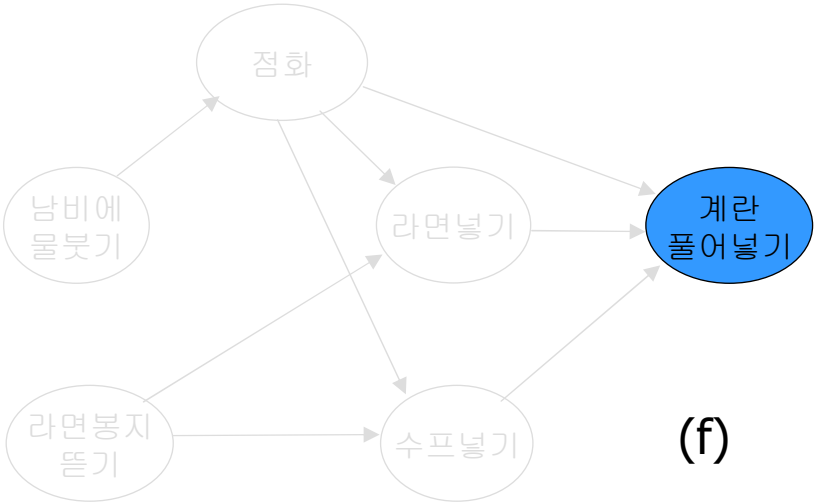
알고리즘 1의 작동 예 (1/2)



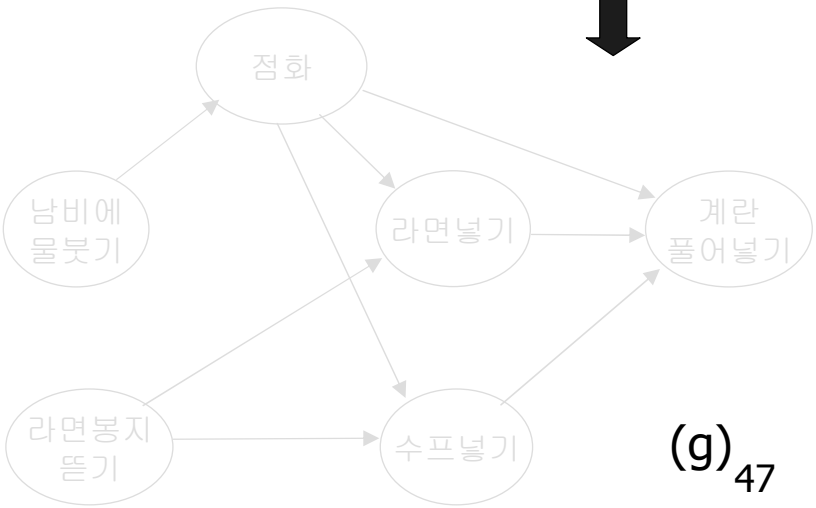
알고리즘 1의 작동 예 (2/2)



(e)



(f)



(g)

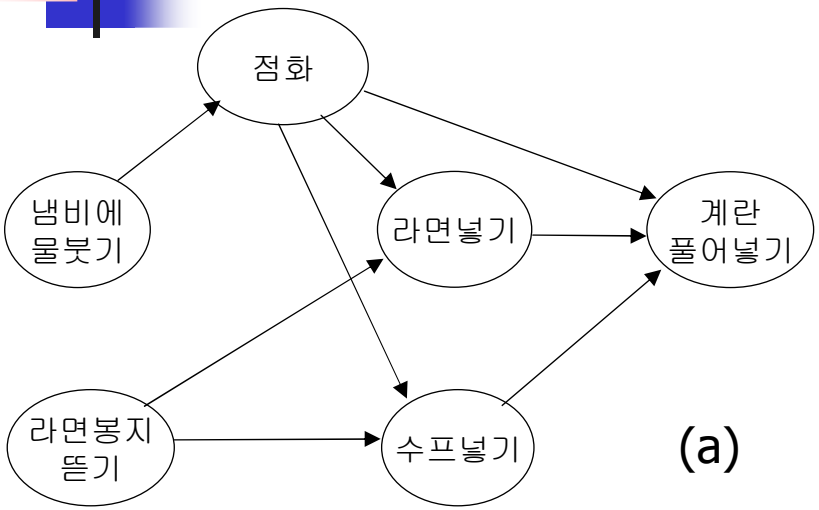
위상정렬 알고리즘 2

Algorithm 2 [Optimal Topological Sort]

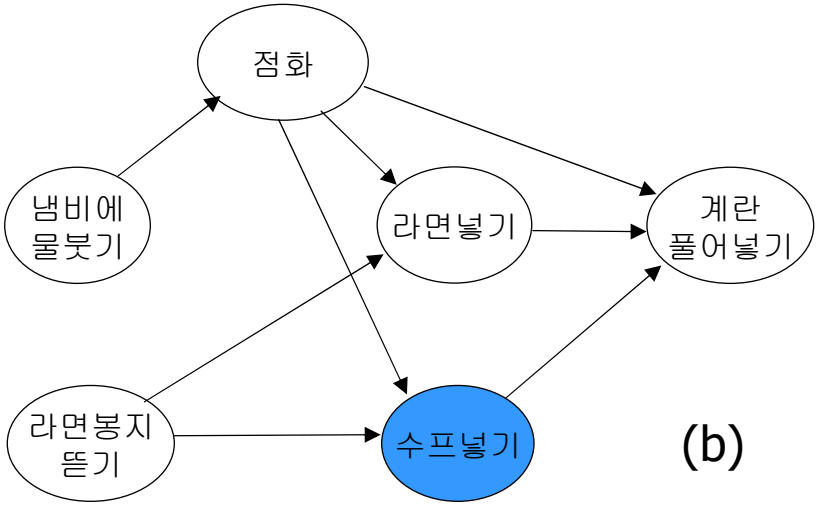
- Do a **DFS** on the graph starting from a vertex (of in-degree zero).
- Output the vertices in reverse order of their finish times.

Time : $\Theta(|V|+|E|)$

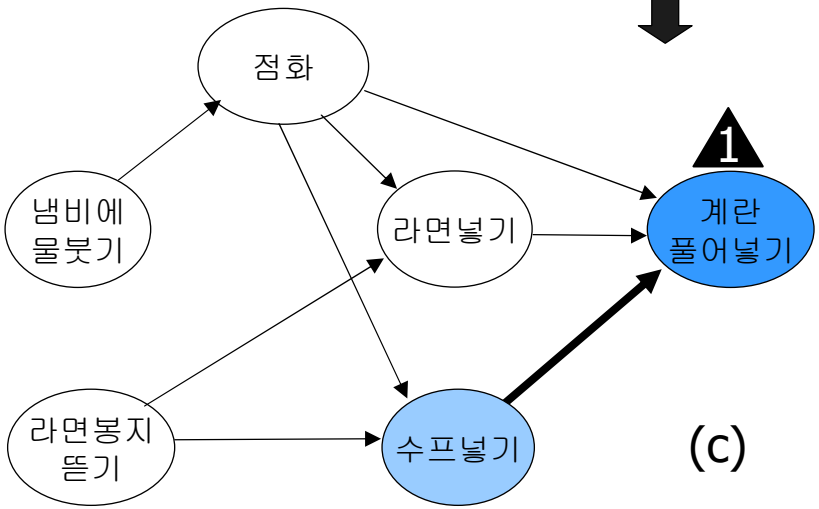
알고리즘 2의 작동 예 (1/3)



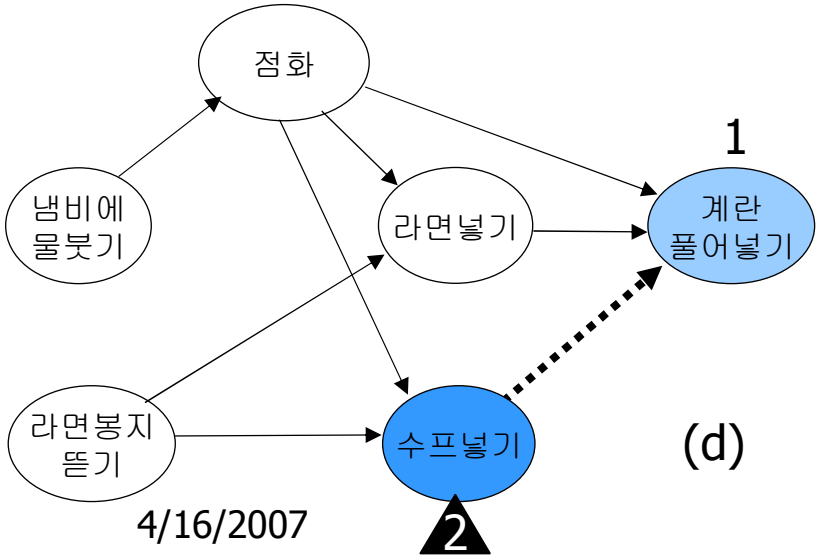
(a)



(b)

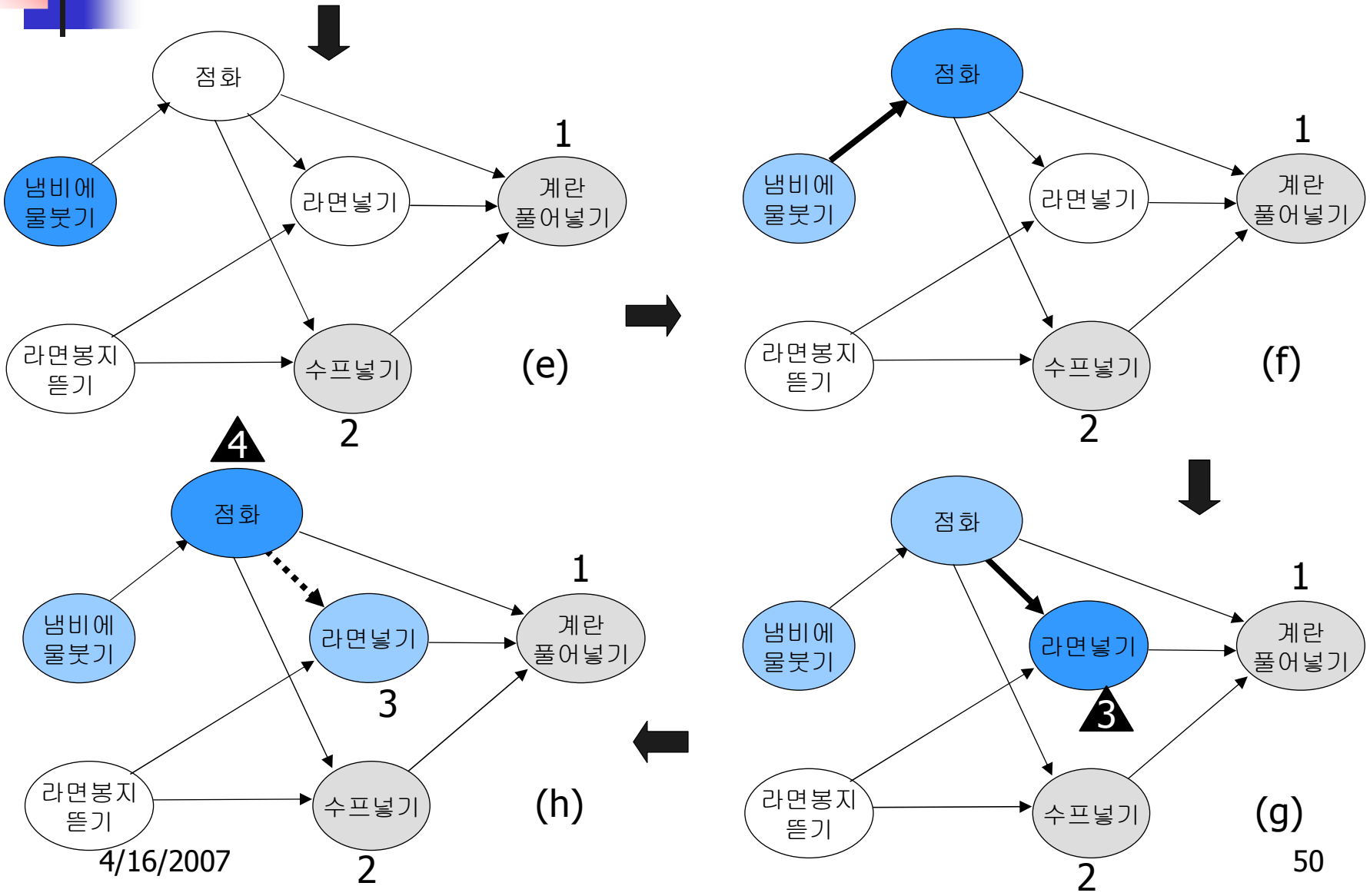


(c)

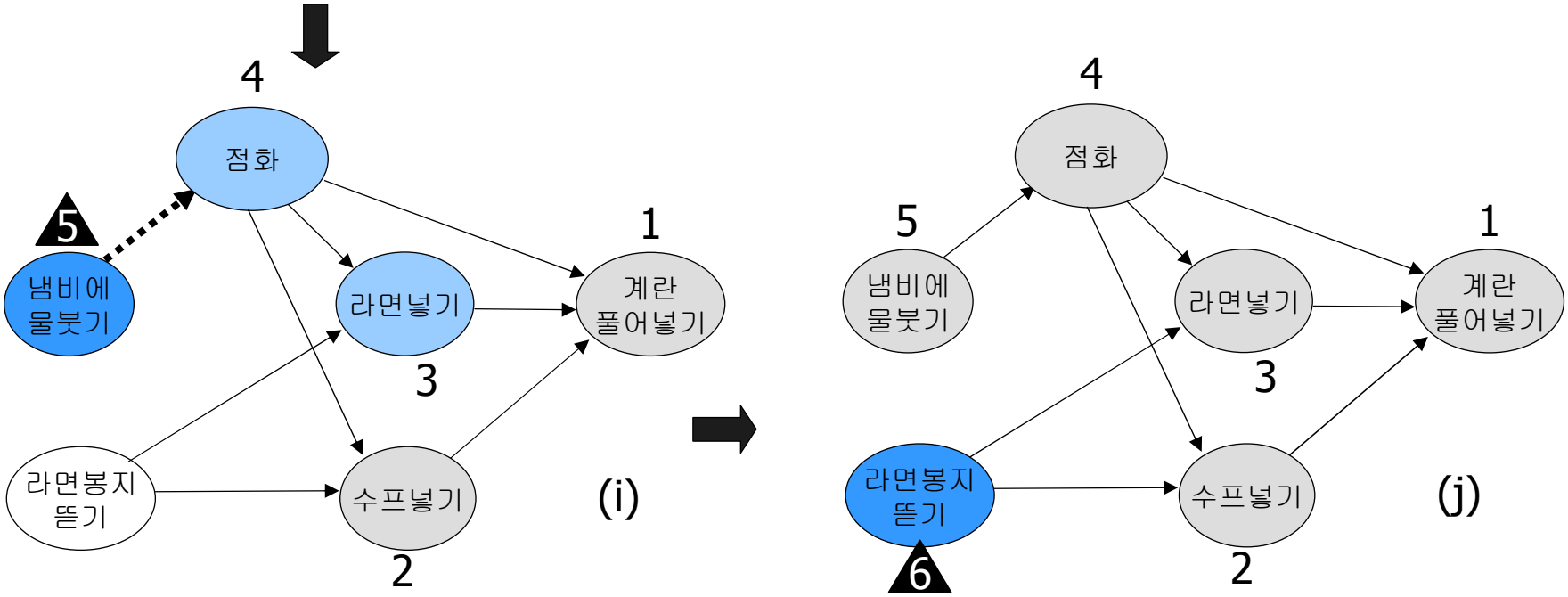


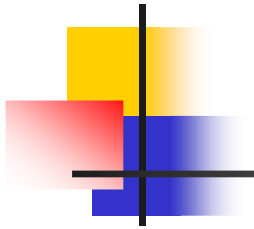
(d)

알고리즘 2의 작동 예 (2/3)



알고리즘 2의 작동 예 (3/3)





강연결요소

연결요소

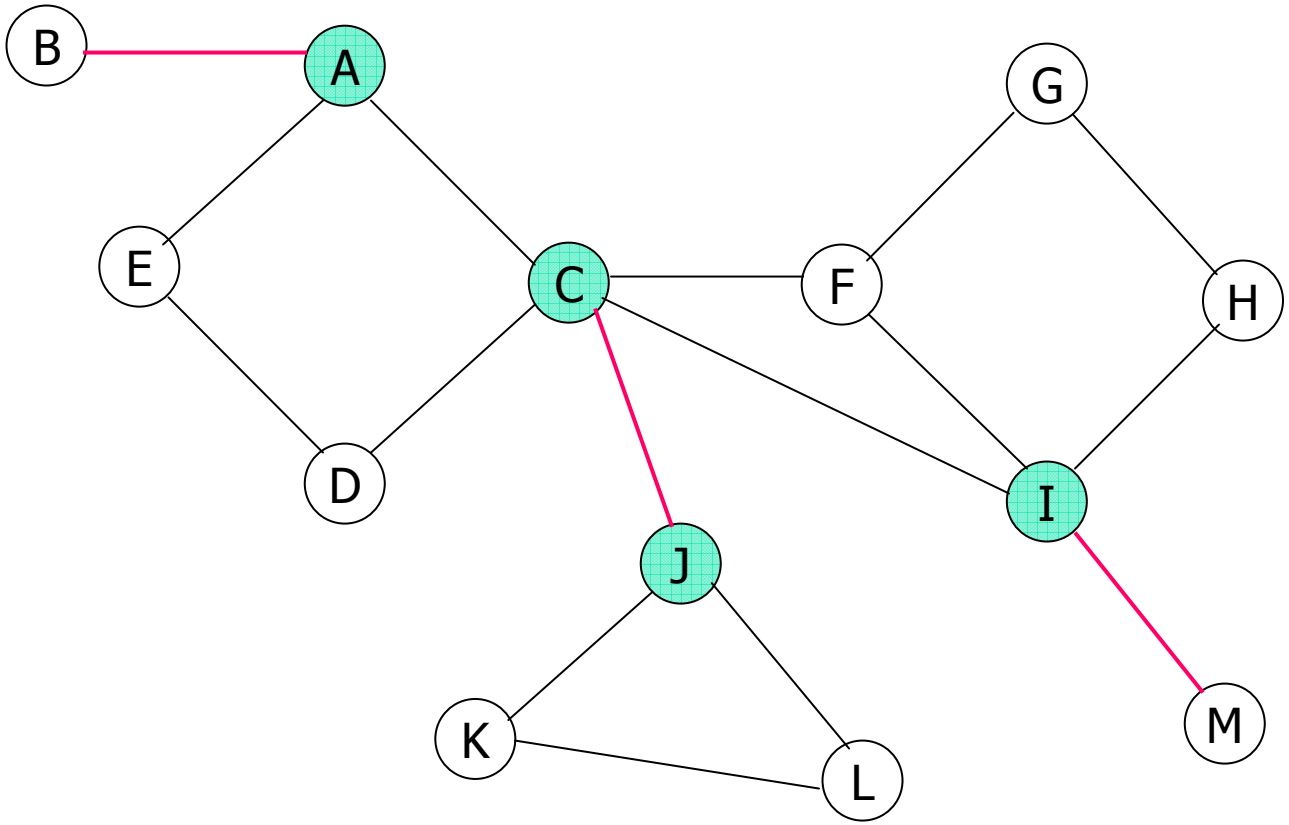
- 연결요소 **(connected component) :** 모든 정점쌍간에 경로가 존재하는 최대의 부분 그래프
  - 그래프 순회 알고리즘 또는 합-찾기 알고리즘에 의해 구할 수 있음.

- 二重連結 그래프 : 정점 쌍간을 연결하는 상이한 경로가 둘 이상인 그래프.

이중 연결 그래프 (1/2)

- 접합점 **(articulation point) :** 제거 시 그래프가 둘 이상의 부분으로 분할이 되는 정점.
  - 접합점이 없으면 이중 연결 그래프
- 다리 **(bridge) :** 제거하면 그래프가 둘 이상의 부분으로 분할이 되는 간선.
- 이중 연결 요소 : 모든 정점쌍간에 둘 이상의 경로가 존재하는 최대의 부분 그래프.
  - 모든 다리를 제거하면 이중 연결 요소가 구해진다.

이중 연결 그래프 (2/2)

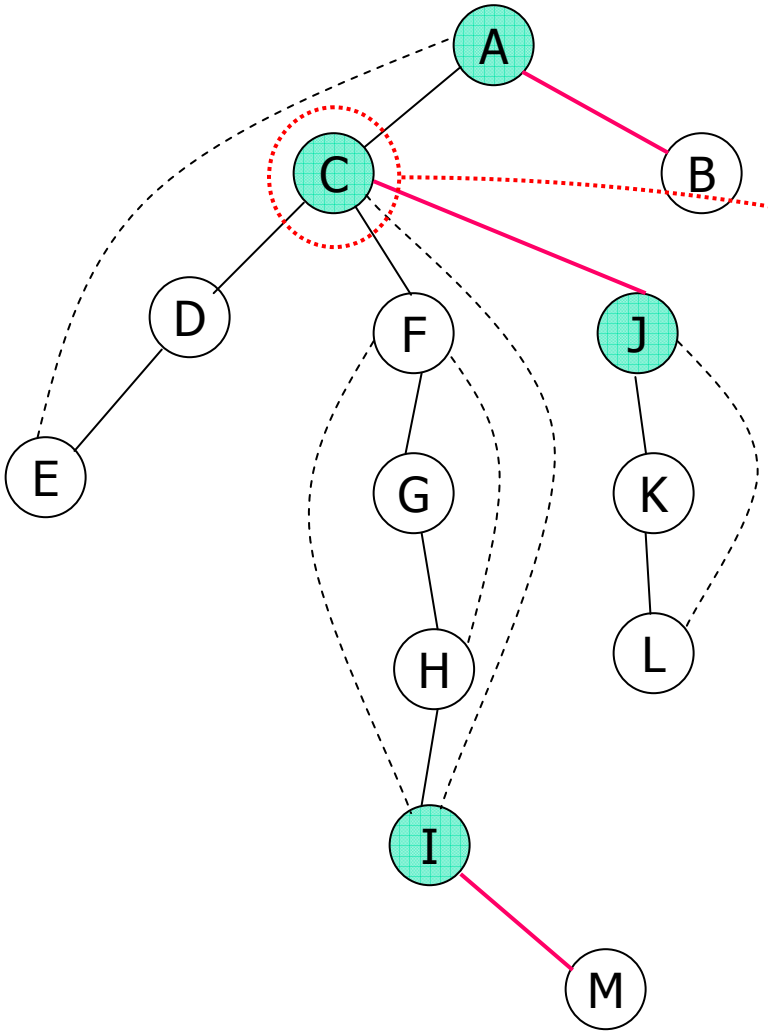


<그림 6-17> 접합점, 다리, 이중 연결 성분

DFS 나무에서 접합점의 조건

1. 뿌리에 둘 이상의 자식이 있으면 뿌리는 접합점이다.
2. 뿌리가 아닌 정점 **u**의 경우 **u**의 어떠한 한 자식 **v**라도 **v**와 그 자손에서 **u**의 조상으로 연결된 점선 링크가 없으면 **u**는 접합점이다.

DFS 나무에서 접합점의 조건



D 이하에서 **C** 위로 올라가는 점선 링크가 있지만 **F** 이하에서 **C** 위로 올라가는 점선 링크가 없으므로 **C**는 접합점이다.

<그림 6-18> **<그림 6-17>**에 대한 깊이 우선 트리

접합점 조사 알고리즘

```
int dfs_visit_ap(int v) {
/* 입력 : 정점 번호 v
   출력 : 접합점, 점선으로 연결된 최상위 정점의 방문 번호 */

1     int  min, m ;
2     struct node *t ;
3     mark++ ;  flag[v] = mark ;  min = mark ;
4     t = head[v]
5     while (t != NULL)  {
6          if (flag[t->vertex] == 0  {       /* 미방문 정점이면 */
7            m = dfs_visit_ap (t->vertex) ;      /* t->vertex 이하에서 점선으로 연결된
8                                                    최상위 정점의 방문번호 */
9            if (m < min)  min = m ;
10           if (m >= flag[v])  printf ("%c", name(v) ) ;      /* 접합점 */
11         } else     /* 이미 방문했던 정점이면, 즉 점선 링크이면 */
12           if (flag[t->vertex] < min)  min = flag[t->vertex] ;
13         t = t->next ;
14    }
15    return (min) ;
   }
```

다리일 필요충분조건

- 간선 (u,v)가 다리
- ⇔ (u,v)가 사이클에 포함되지 않는다.
- ⇔ 접합점 u에 부수된 간선으로서 v를 포함한 아래 정점들에서 u를 포함한 위쪽 정점들에 연결되는 점선 링크가 없다.

강연결요소

- 강연결요소 **(strongly connected component) :** 방향 그래프의 모든 정점쌍에 대해서 양방향으로 경로가 존재하는 최대의 부분 그래프

- 강연결 그래프 – 강연결 성분이 하나인 그래프

강연결요소 구하기 알고리즘

stronglyConnectedComponent(G)
{

1 그래프 G에 대해 DFS를 수행하여 각 정점 v의 완료시간 f[v]를 계산한다.

2 G의 모든 간선들의 방향을 뒤집어 G^R을 만든다.

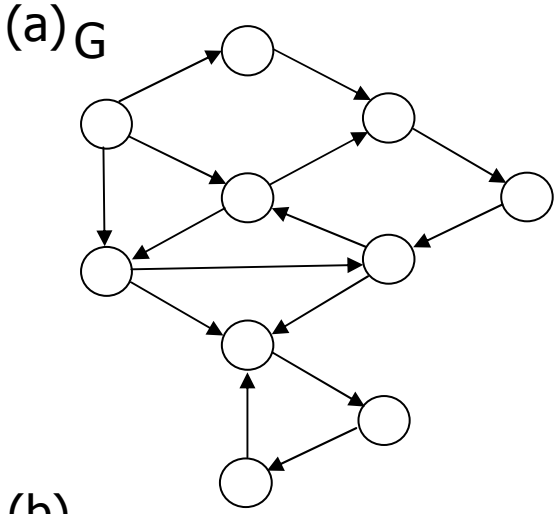
3 DFS(G^R)를 수행하되 DFS 알고리즘의 ❶행에서 시작점을 택할 때 1에서 구한 f[v]가 가장 큰 정점으로 잡는다.

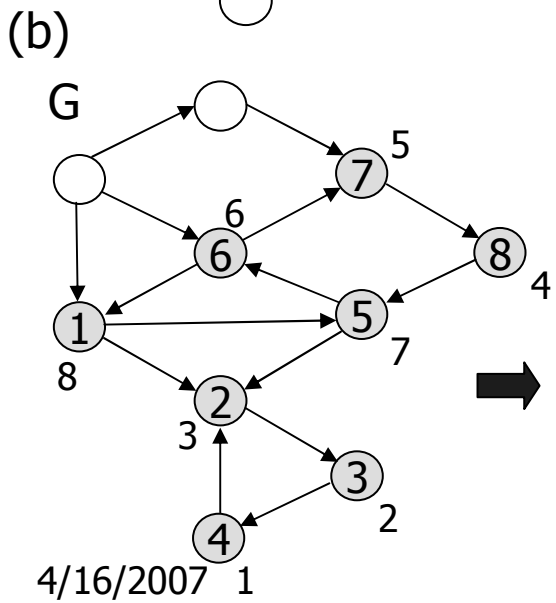
4 앞의 3에서 만들어진 분리된 트리들 각각을 강연결요소로 리턴한다.

}

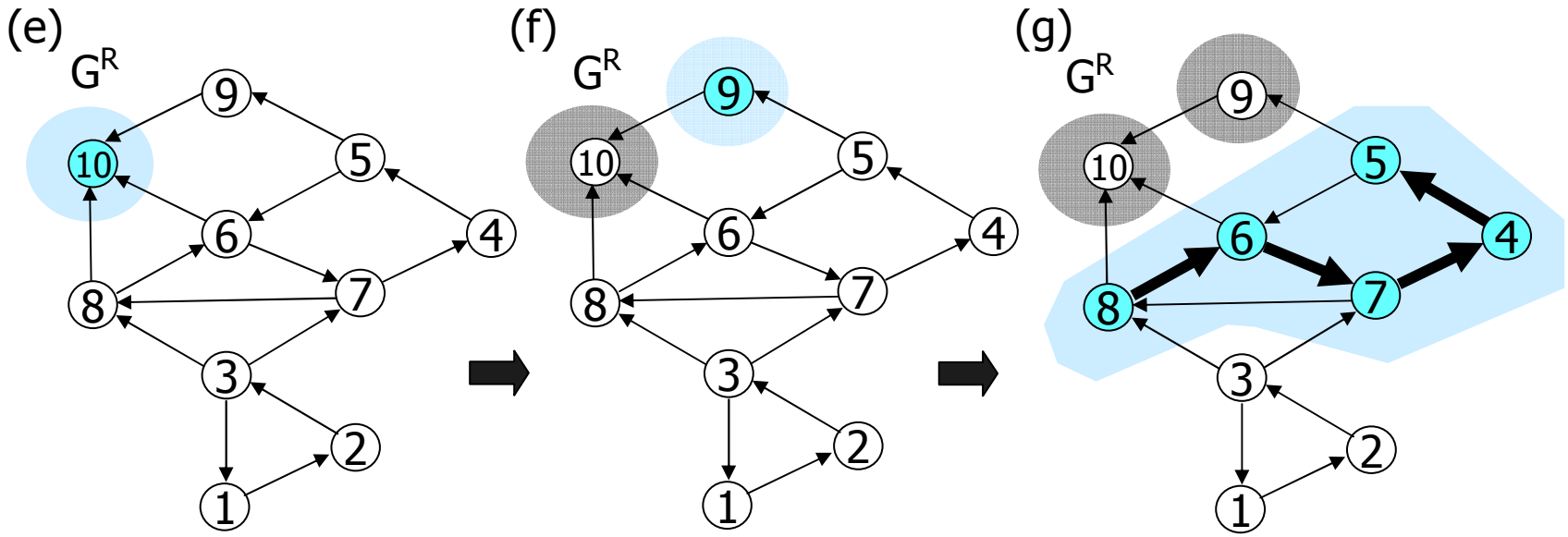
✓수행시간 : $\Theta(|V|+|E|)$

알고리즘의 작동 예 (1/3)

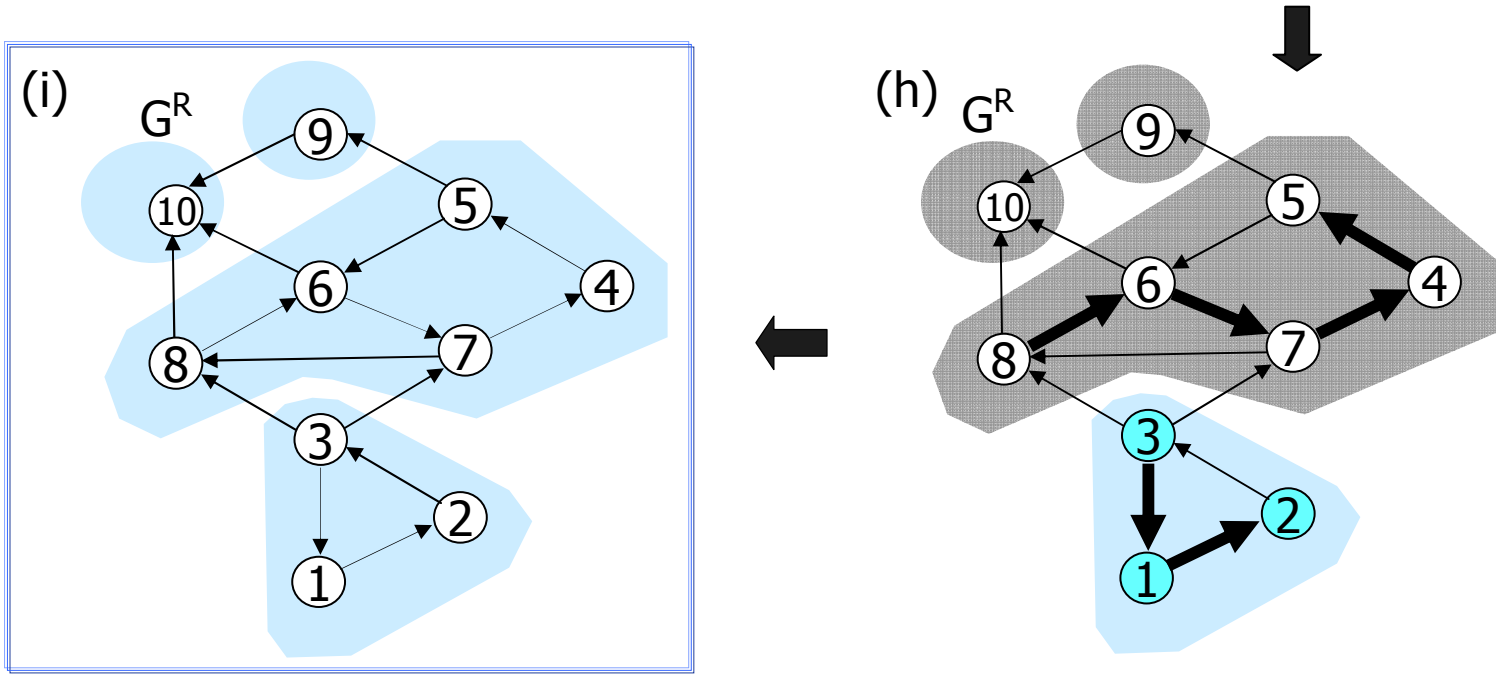




알고리즘의 작동 예 (2/3)



알고리즘의 작동 예 (3/3)



알고리즘의 수행시간 분석

- 정확성 - 두 가지를 증명
  1. 단계 4에서 같은 트리에 속하는 정점들은 같은 강연결요소에 속한다.
  2. 같은 강연결요소에 속하는 정점들은 단계 4에서 같은 트리에 속한다.

- $\Theta(|V|+|E|)$ 시간

최소 신장 트리

최소 신장 트리 (1/2)

- 조건
  - 무향 연결 그래프
    - **연결 그래프 (connected graph) :** 모든 정점 간에 경로가 존재하는 그래프
- 트리
  - 싸이클이 없는 연결 그래프
  - n개의 정점을 가진 트리는 항상 n-1개의 간선을 갖는다
- 그래프 G의 신장트리
  - G의 정점들과 간선들로만 구성된 트리

최소 신장 트리 (2/2)

- G의 최소신장트리 (MST: Minimum Spanning Tree) :
  - G의 신장트리들 중 간선의 합이 최소인 신장트리

- 크루스칼(Kruskal)의 방법과 프림(Prim)의 방법

프림 알고리즘

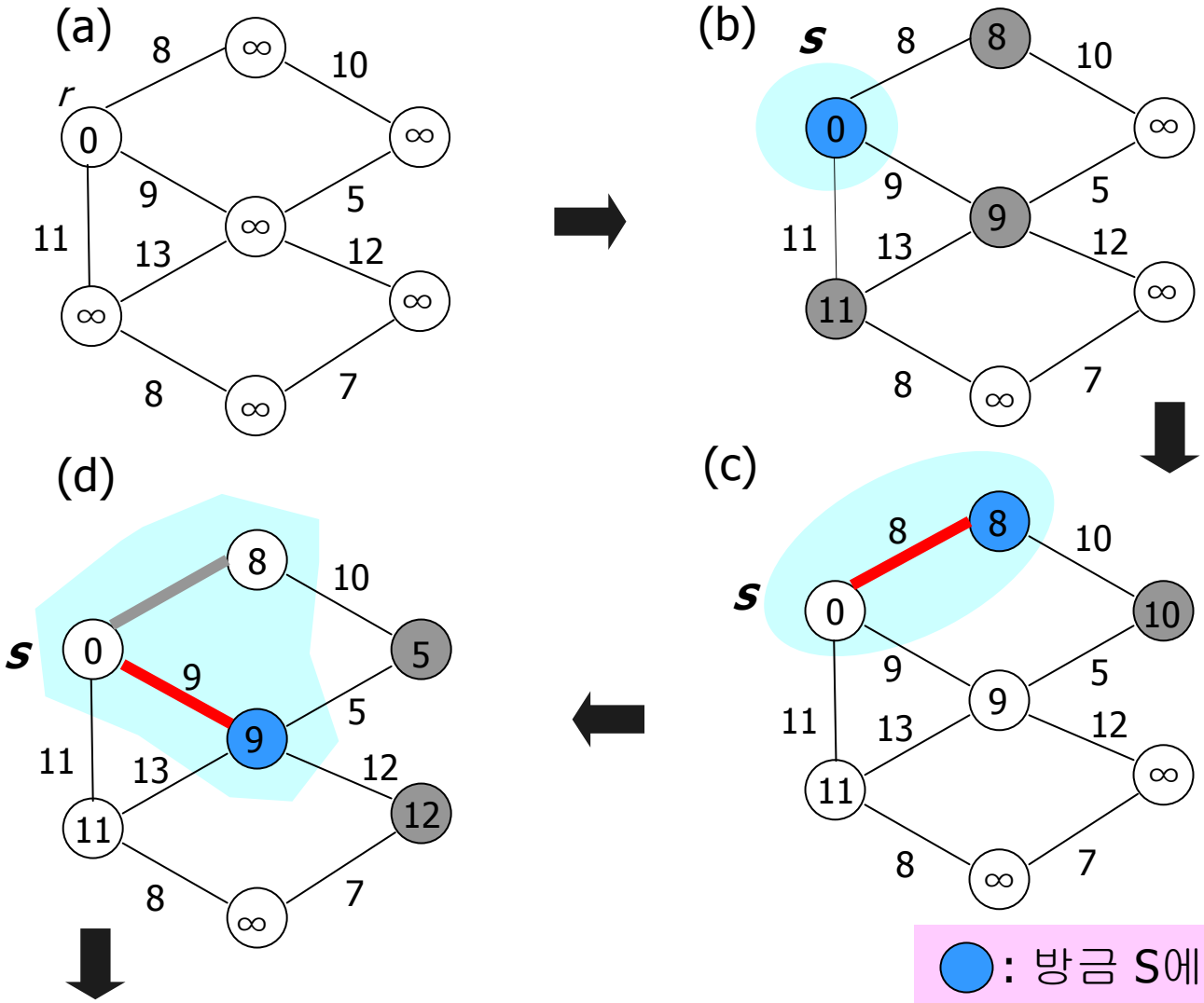
```
Prim (G, r)
{
      S ←Φ ;
      정점 r을 방문되었다고 표시하고, 집합 S에 포함시킨다;
      while (S≠V) {
            S에서 V-S를 연결하는 간선들 중 최소길이의 간선 (x, y) 를 찾는다;
                                                ▷ (x∈S, y∈V-S)
            정점 y를 방문되었다고 표시하고, 집합 S에 포함시킨다;
    }
}
```

✓수행시간: O(|E|log|V|)

힙 이용

- Prim 알고리즘은 greedy 알고리즘의 일종
- Greedy 알고리즘으로 최적해를 보장하는 드문 예

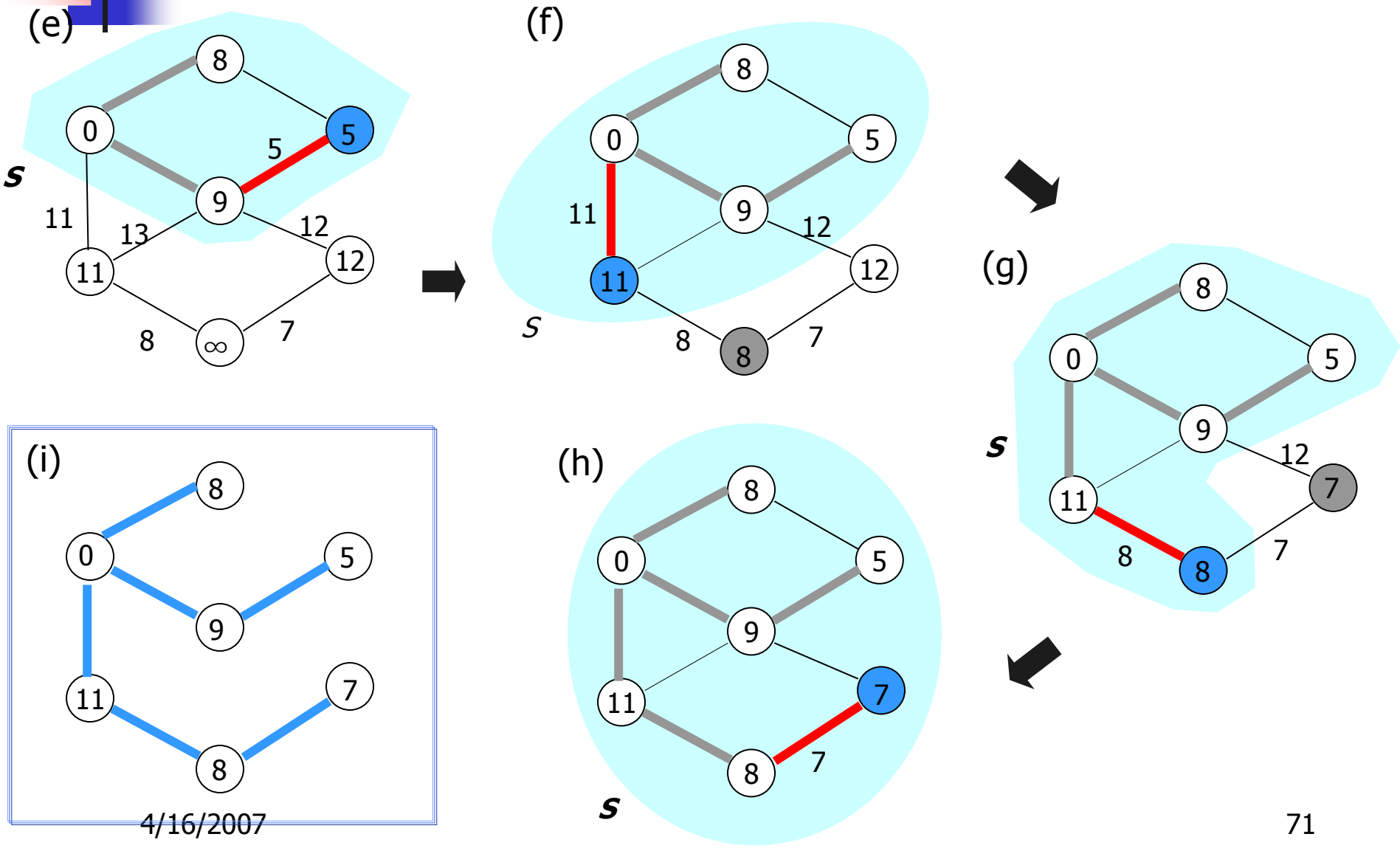
프림 알고리즘의 작동 예 (1/2)



: 방금 S에 포함된 정점

: 방금 이완이 일어난 정점

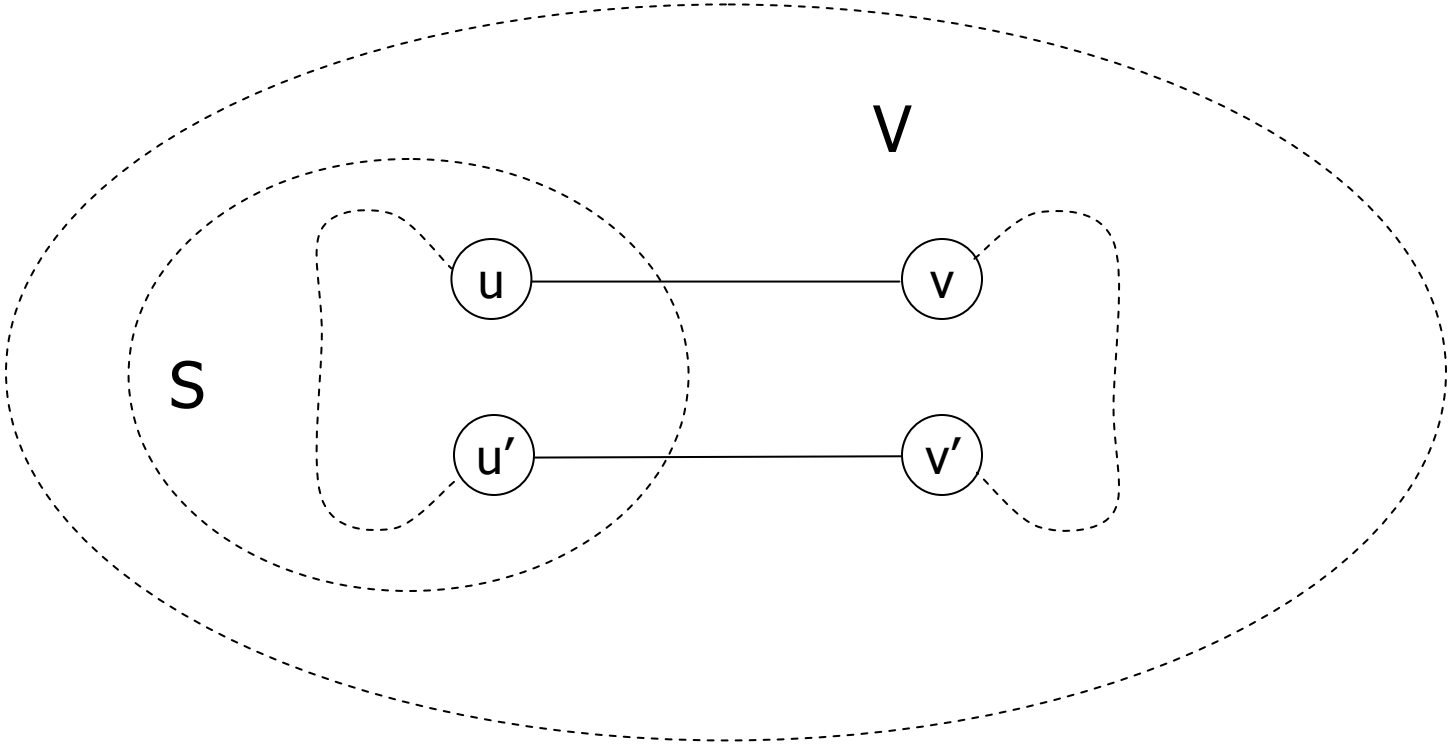
프림 알고리즘의 작동 예 (2/2)



정리 6.1 (1/2)

- 연결 가중 그래프 G = (V, E)에 대하여 S를 V의 진부분 집합이라 하자. (u, v)가 u ∈ S이고 v ∈ V-S인 최소 가중치 간선이라면 간선 (u, v)를 포함하는 최소 신장 트리가 존재한다.

정리 6.1 (2/2)



<그림 6-34> 최소 간선과 두 정점 집합

딸림정리 6.2

- G = (V, E)를 연결 가중 그래프라고 하자. B를 G에 대한 최소 신장 트리의 간선 집합 T의 부분 집합이라 하고, C를 숲 $G_B = (V, B)$에 있는 한 연결요소라 하자. 만약 (u, v)가 C와 다른 연결요소를 잇는 최소 간선이라면 $(u, v) \in T$인 T가 존재한다.
  - Kruskal의 알고리즘

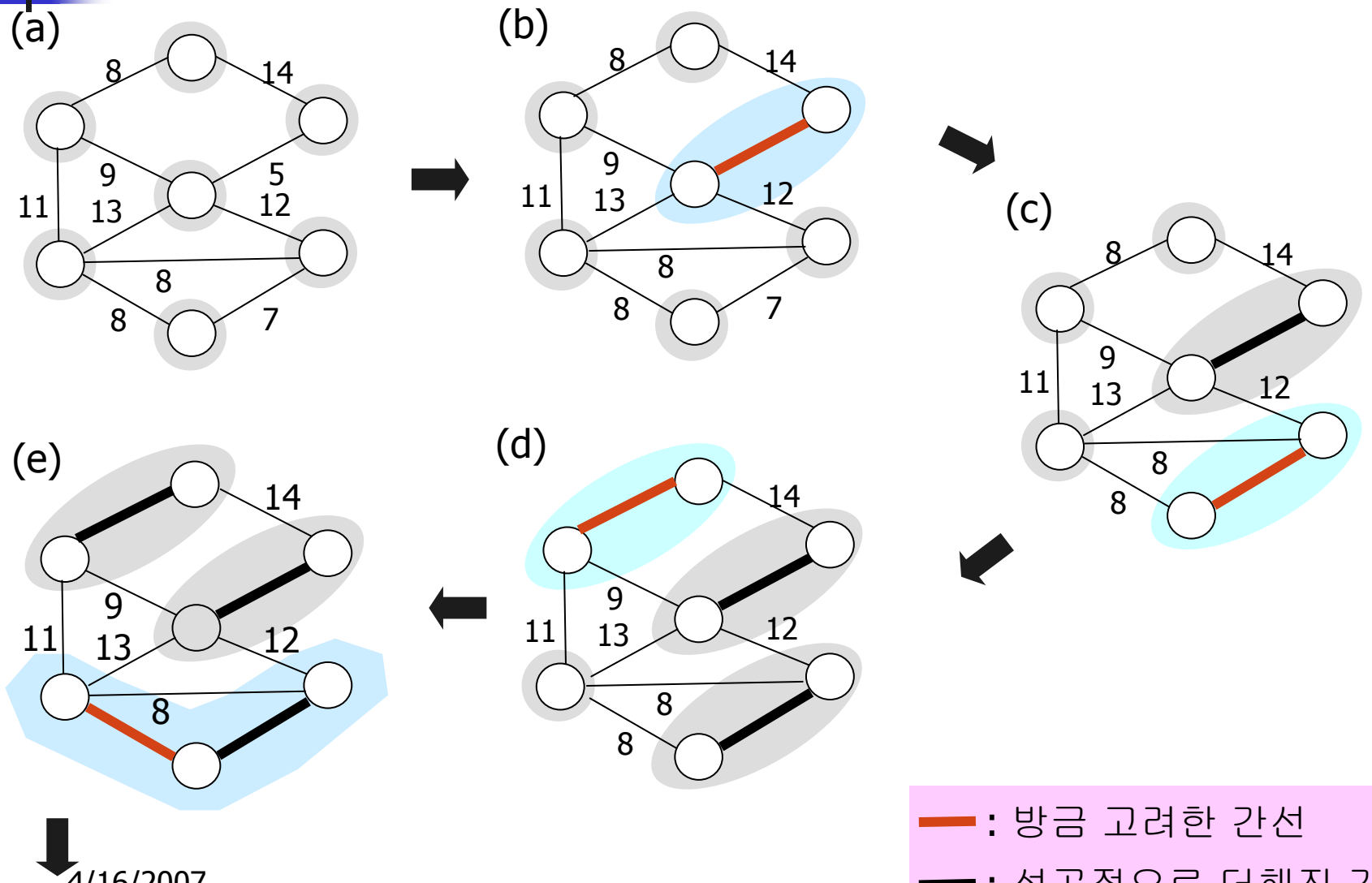
크루스칼 알고리즘

```
Kruskal (G, r)
{
    T ← Φ; ▷ T : 신장트리
    단 하나의 정점만으로 이루어진 n 개의 집합을 초기화한다;
    모든 간선을 가중치가 작은 순으로 정렬한다;
    while (T의 간선수 < n-1) {
            최소비용 간선 (u, v)를 제거한다;
            정점 u와 정점 v가 서로 다른 집합에 속하면 {
                    두 집합을 하나로 합친다;
                    T ← T∪{u, v)};
            }
    }
}
```

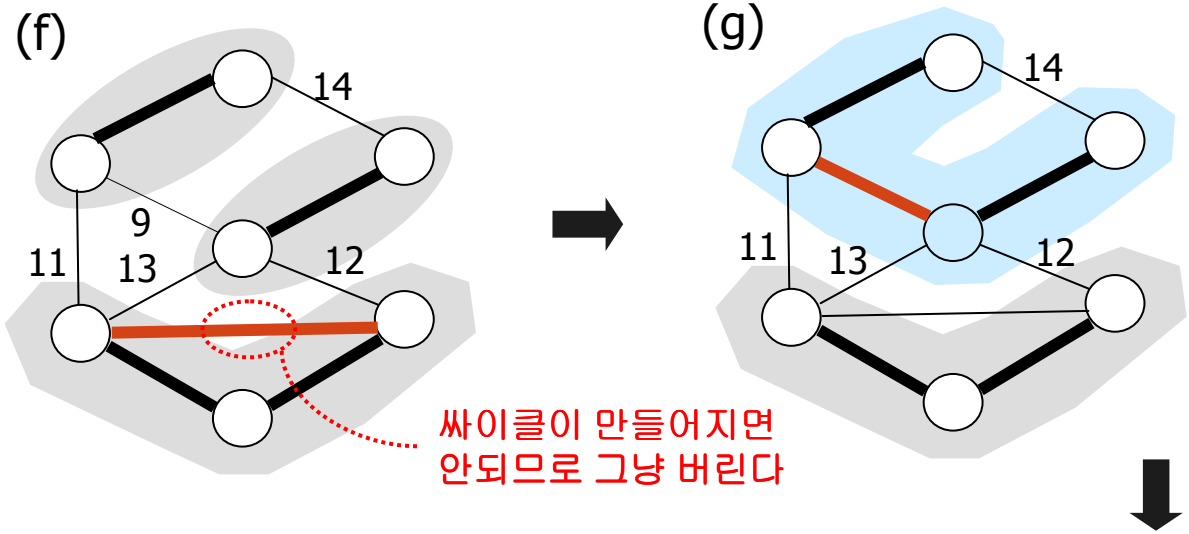
✓수행시간: O(|E|log|V|)

- 크루스칼 알고리즘은 greedy 알고리즘의 일종

크루스칼 알고리즘의 작동 예



크루스칼 알고리즘의 작동 예



프림 v.s. 크루스칼 (1/2)

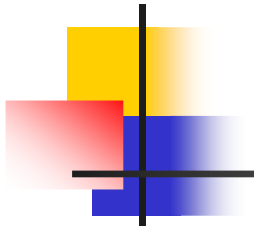
- **프림 알고리즘 :** 하나의 집합 S를 키워나가면서 하나의 신장트리를 만드는 방식
  - 집합 S에 정점을 하나씩 더해 나가지만 정점을 하나 더할 때마다 해당 정점을 트리에 연결하는 간선이 정해지므로 사실상 간선을 하나씩 더하는 것임

- **크루스칼 알고리즘 :** 여러 집합들을 합쳐가면서 최종적으로 하나의 신장트리를 만드는 방식
  - 두 개의 집합을 합칠 때마다 두 집합을 연결하는 하나의 간선이 정해지므로 역시 간선을 하나씩 더하는 것임

프림 v.s. 크루스칼 (2/2)

- 따라서 두 알고리즘 모두 n-1개의 간선을 더해가는 순서를 정해준다.

- 프림 알고리즘과 크루스칼 알고리즘은 이런 논리적 기반 하에서 최소(안전한) 간선을 차례로 더해 가는 것이다.



최단 경로

최단 경로 (1/2)

- 조건
  - 간선 가중치가 있는 유향 그래프
  - 무향 그래프는 각 간선에 대해 양쪽으로 유향 간선이 있는 유향 그래프로 생각할 수 있다
    - 즉, 무향 간선 (u, v)는 유향 간선 (u, v)와 (v, u)를 의미한다고 가정하면 된다
- 두 정점 사이의 최단경로
  - 두 정점 사이의 경로들 중 간선의 가중치 합이 최소인 경로
  - 간선 가중치의 합이 음인 싸이클이 있으면 문제가 정의되지 않는다

최단 경로 (2/2)

- 단일 시작점 최단경로
  - 단일 시작점으로부터 각 정점에 이르는 최단경로를 구한다
  - 다익스트라 알고리즘
    - 음의 가중치를 허용하지 않는 최단경로
  - 벨만-포드 알고리즘
    - 음의 가중치를 허용하는 최단경로
  - 싸이클이 없는 그래프의 최단경로
- 모든 쌍 최단경로
  - 모든 정점 쌍 사이의 최단경로를 모두 구한다
  - 플로이드-워샬 알고리즘

단일 시작점 최단 경로

- 음의 가중치를 갖는 간선이 없는 가중그래프에서 출발 정점 x에서 다른 모든 정점까지 가중치 합이 최소인 경로를 찾는 문제.

- 거리 D[v]: 출발점 s에서 현재까지 선택된 정점집합을 경유하여 정점 v에 이르는 최소 경로의 길이

Dijkstra 알고리즘 (1/2)

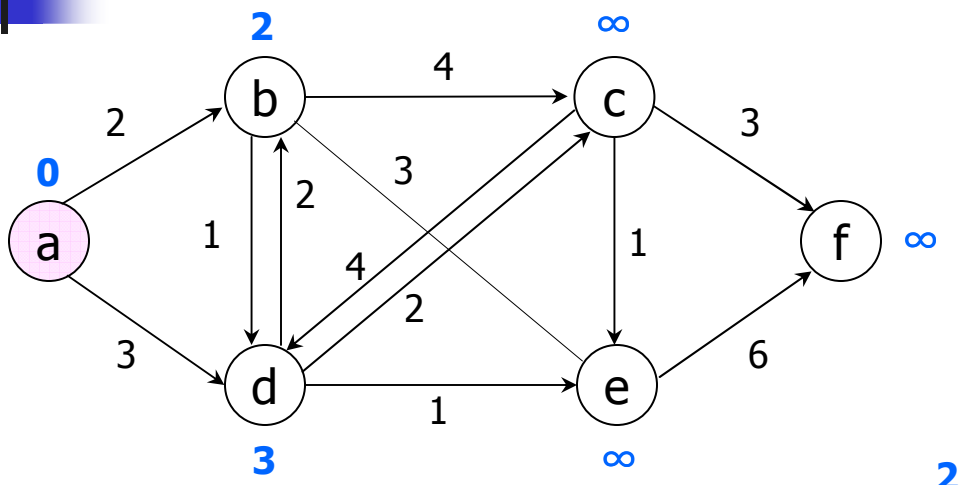
- Dijkstra 알고리즘 - 출발점에서 시작하여 거리가 최소인 정점을 선택해 나가면 최단 경로를 구할 수 있다는 greedy 알고리즘의 일종이다.
  - Dijkstra 알고리즘은 앞에서 배운 최소신장트리를 위한 프림 알고리즘과 원리가 거의 같다.

Dijkstra 알고리즘 (2/2)

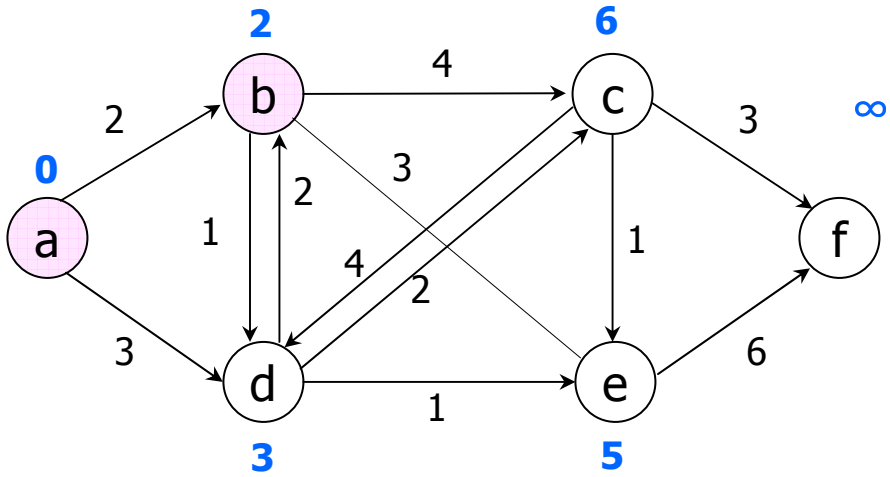
```
    Dijkstra(G, A, s) {
    /* 입력 : 그래프 G = (V, E), 인접 행렬 A, 시작 정점 s
       출력 : s로부터 다른 모든 정점까지의 최단 경로의 길이 */
1       U = {s};
2       for (모든 정점 v)
3           D[v] = A[s][v];
4       while (U != V)  {
5               D[w]가 최소인 정점 w∈V-U를 선택.
6               U = U∪{w};
7               for (w에 인접한 모든 정점 v)  {
8                   D[v] = min (D[v], D[w] +A[w][v];
                }
        }
    }
```

<그림 6-44> 최단 경로를 구하는 다익스트라 알고리즘

Dijkstra 알고리즘의 작동 예



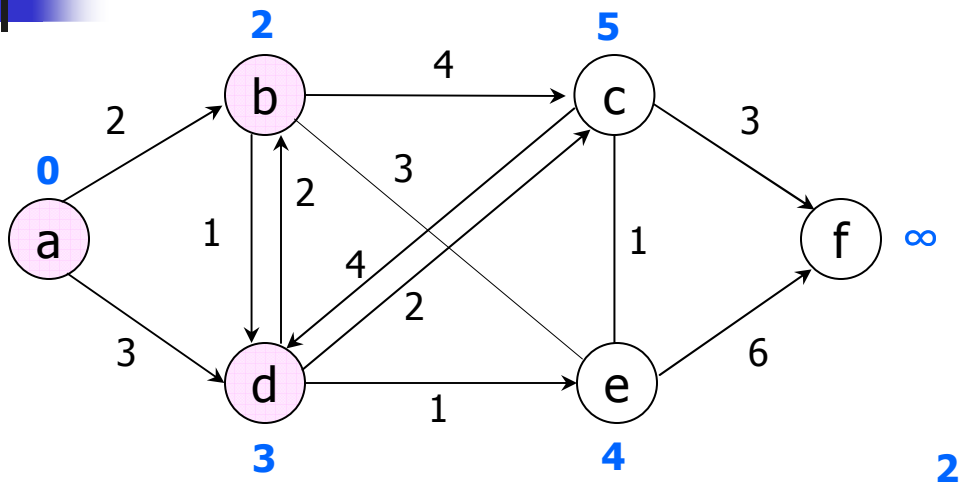
(a)



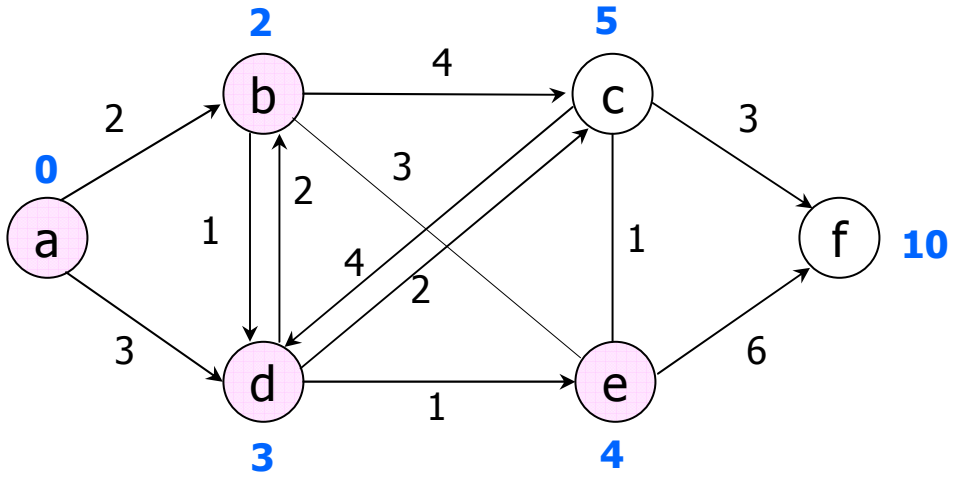
(b)

<그림 6-43> 다익스트라 방법의 적용 예

Dijkstra 알고리즘의 작동 예



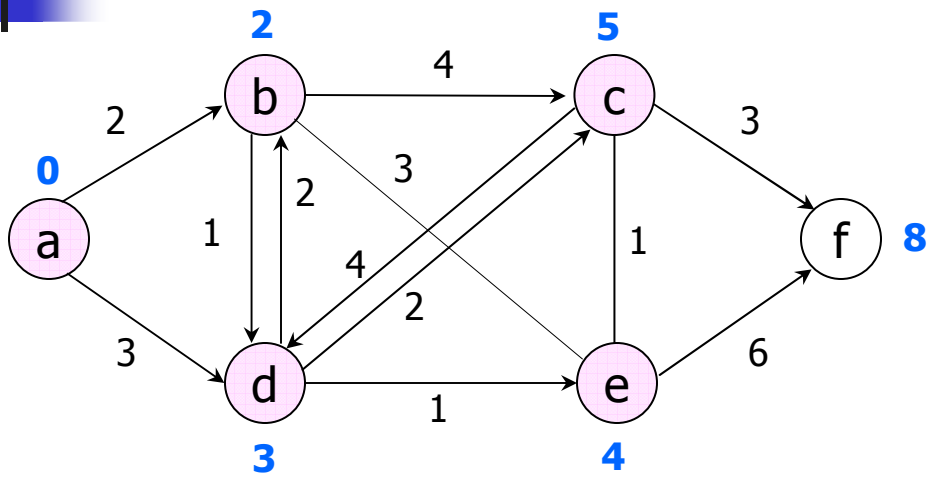
(c)



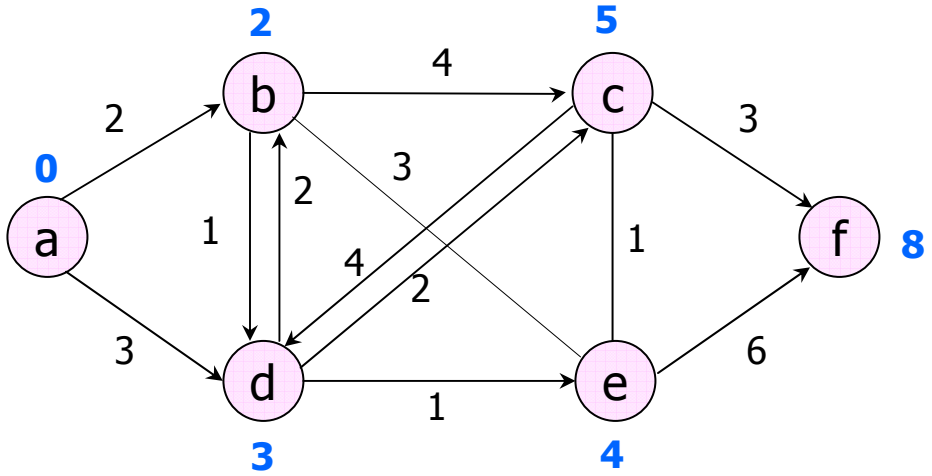
(d)

<그림 6-43> 다익스트라 방법의 적용 예

Dijkstra 알고리즘의 작동 예



(e)

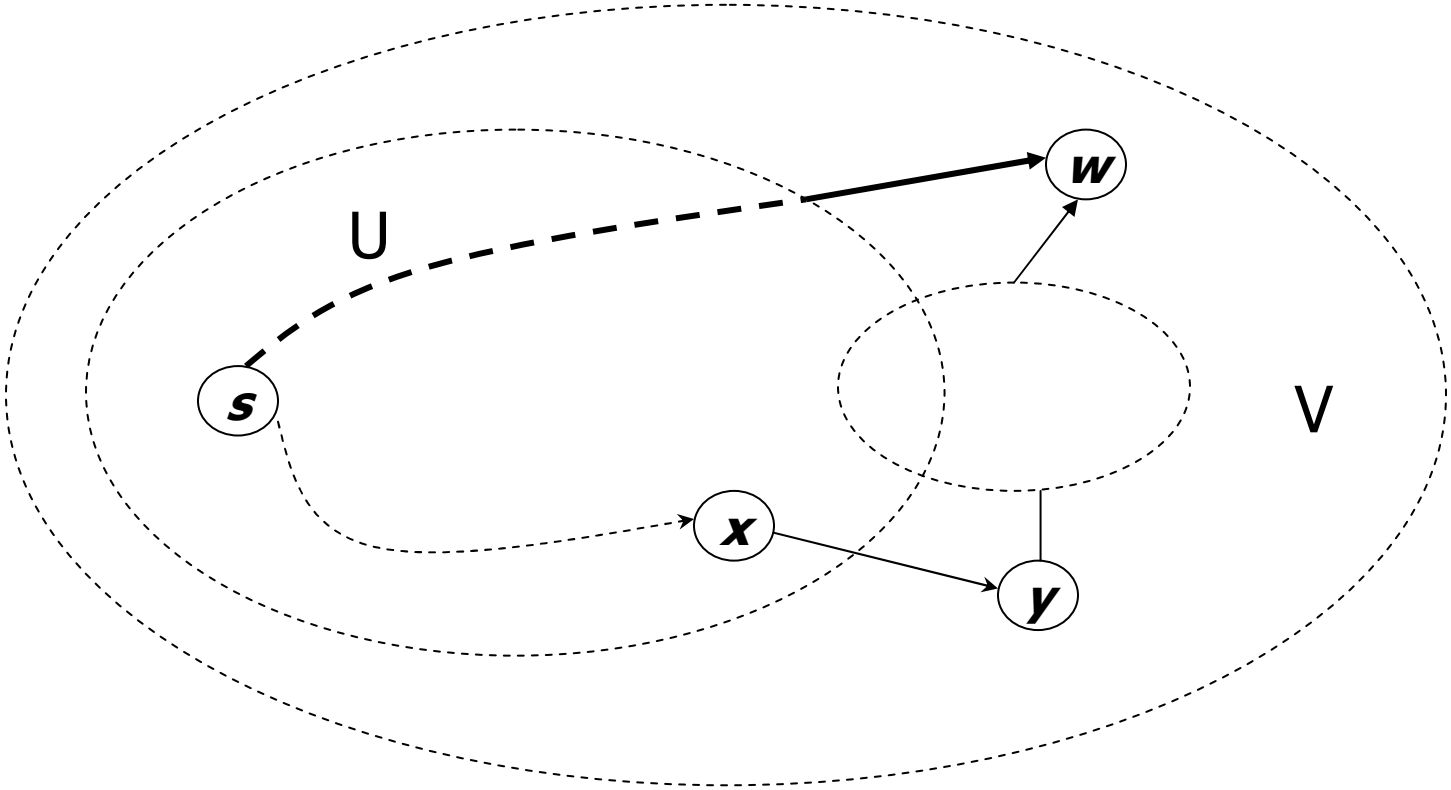


(f)

<그림 6-43> 다익스트라 방법의 적용 예

Dijkstra 알고리즘의 정확성

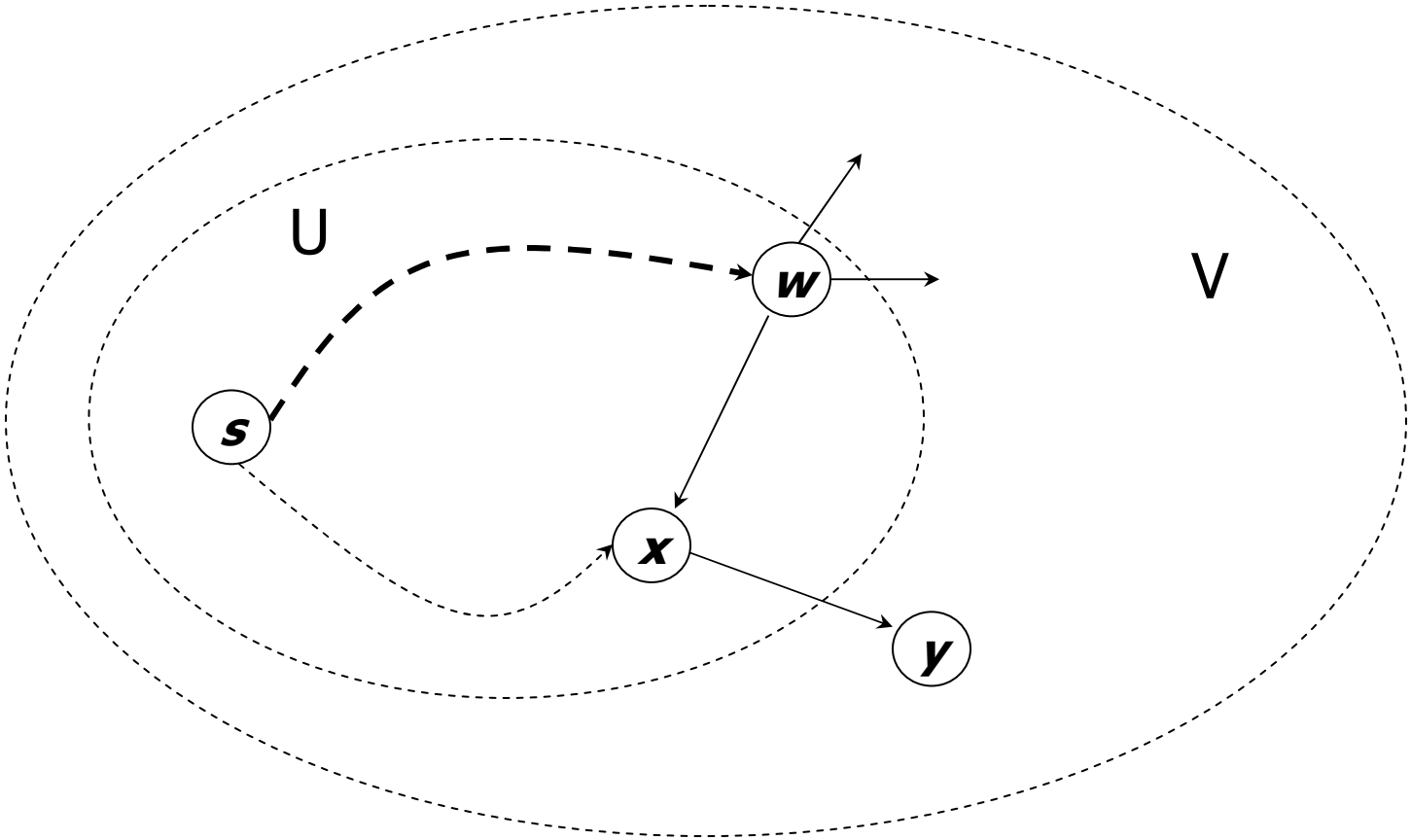
1. 미선택 정점 중에서 선택한 최소 거리 정점 w의 거리 D[w]는 s에서 w까지의 최단 경로의 길이이다.



<그림 6-45> 사실 1)의 증명

Dijkstra 알고리즘의 정확성

2. 거리 D의 조정이 정확하다.



Dijkstra 알고리즘의 실행시간

- O(|E|log|V|)

힙 이용

벨만-포드 알고리즘 (1/2)

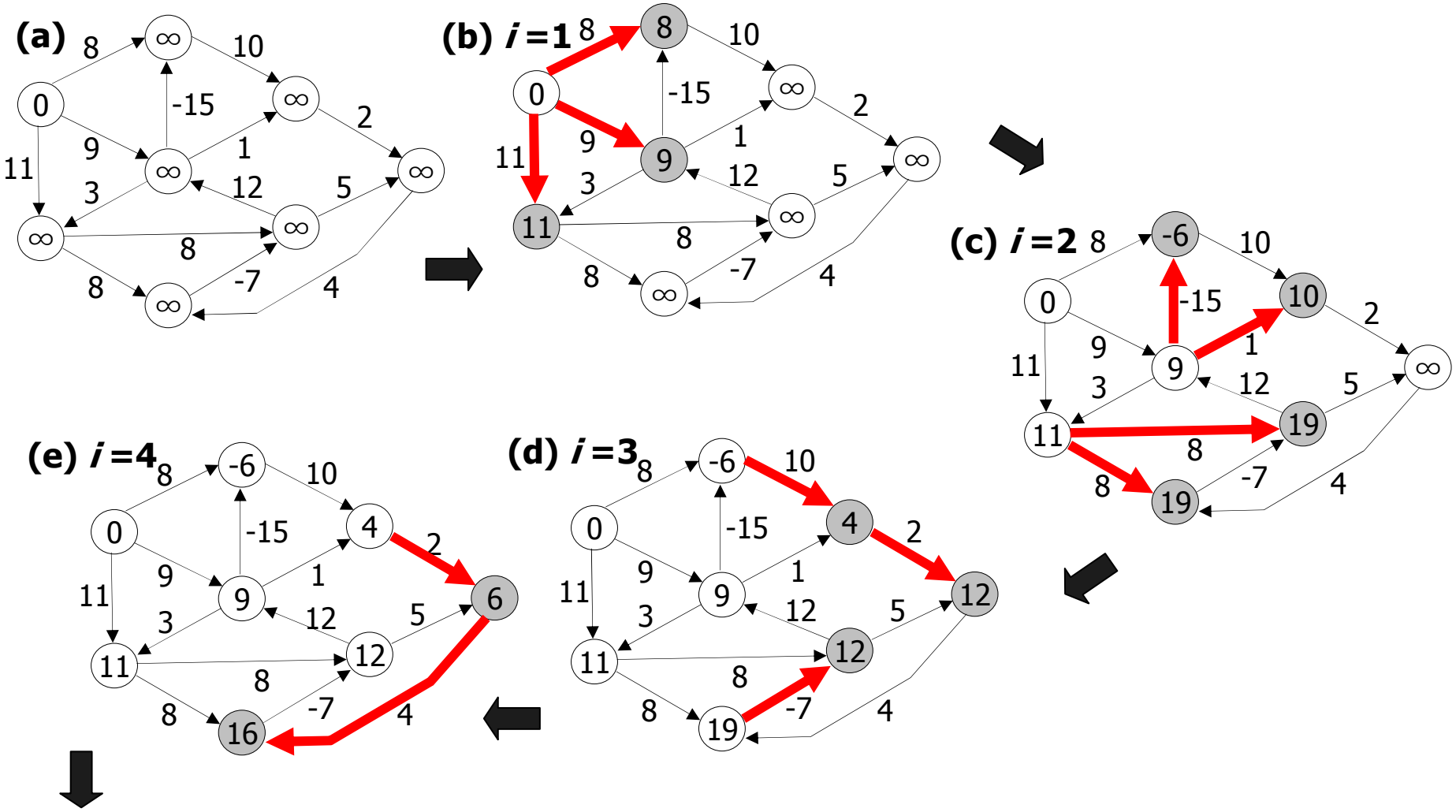
- Dijkstra 알고리즘은 간선의 가중치가 음이 되면 작동하지 않는다.
  - Dijkstra 알고리즘은 집합 S에 한번 포함된 정점에 대해서는 최단거리를 다시 계산하지 않으므로 이러한 예에 대해서는 제대로 작동하지 않는다.

- 벨만-포드(Bellman-Ford) 알고리즘은 입력 그래프 G = (V, E)에서 간선의 가중치가 음의 값을 허용하는 임의의 실수인 경우의 최단경로 알고리즘이다.

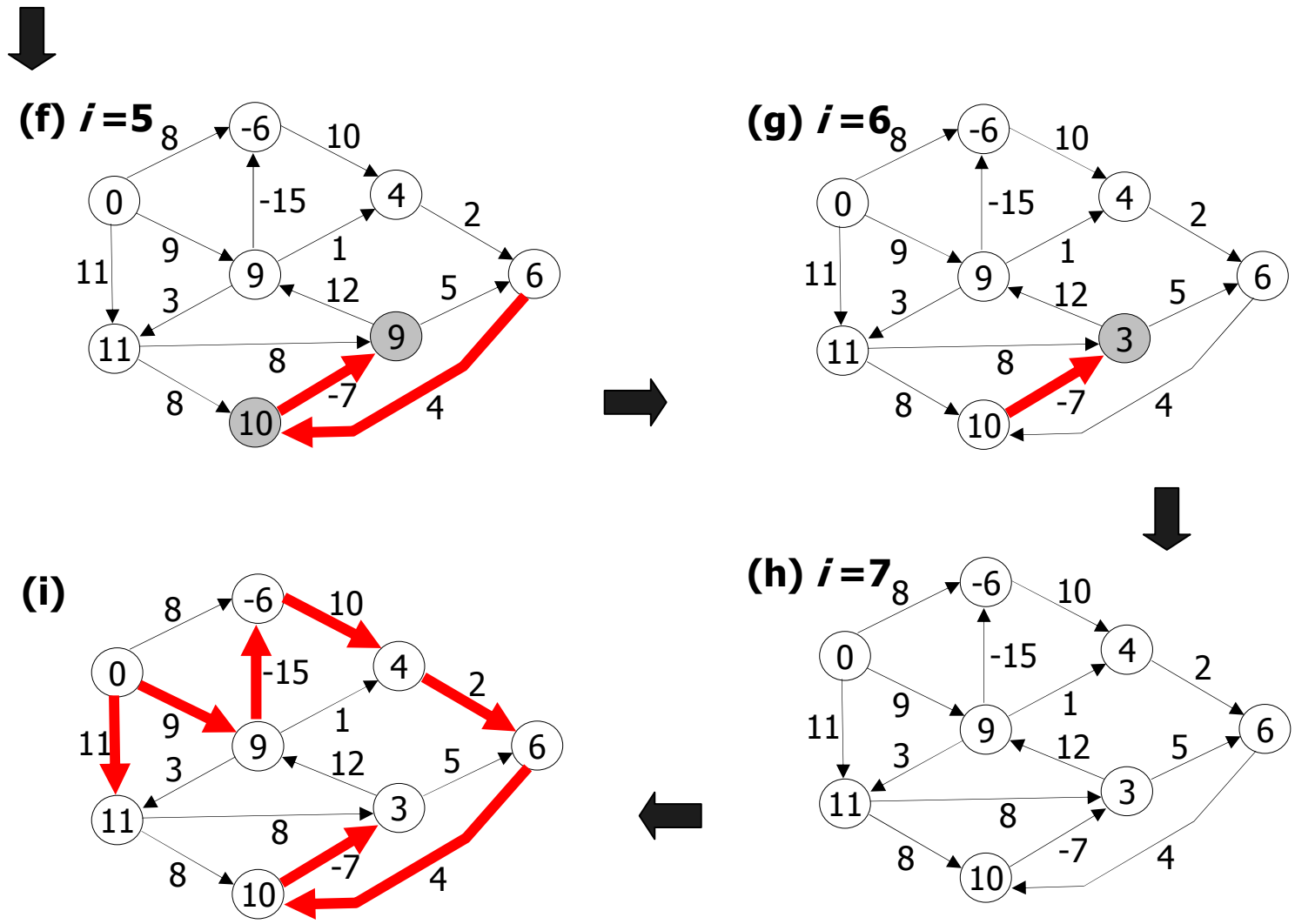
벨만-포드 알고리즘 (2/2)

- 벨만-포드 알고리즘은 간선을 최대 1개 사용하는 최단경로, 간선을 최대 2개 사용하는 최단경로, … 이런 식으로 간선을 최대 n-1개 사용하는 최단경로까지 구해나간다.

알고리즘의 작동 예 (1/2)



알고리즘의 작동 예 (2/2)



알고리즘의 실행시간

- 벨만-포드 알고리즘의 수행시간 : O(|V||E|)
- 벨만-포드 알고리즘을 사용해야 하는 곳에 다익스트라 알고리즘을 사용하면 제대로 해를 구하지 못한다. 반대로 다익스트라 알고리즘을 사용해 해를 구할 수 있는 경우에는 항상 벨만-포드 알고리즘을 써도 된다.
  - 그러나 벨만-포드 알고리즘은 O(|E|log|V|) 시간이 소요되는 다익스트라 알고리즘에 비해 시간이 많이 걸리므로 좋은 선택이 아니다.

모든 쌍 최단 경로 (1/3)

- 경로의 길이가 음인 사이클이 그래프에 존재하지 않는 것으로 가정.
  - 음의 사이클이 있으면 해당 사이클을 몇 번이고 반복해서 돌아 경로의 가중치 합을 무한정 낮출 수 있기 때문에 최단경로 문제 자체가 성립하지 않는다.
- 플로이드(-와샬) 알고리즘 - 동적 프로그래밍을 적용한 방법.
- 중간 정점 : 단순 경로 p의 첫 정점과 마지막 정점을 제외한 모든 정점

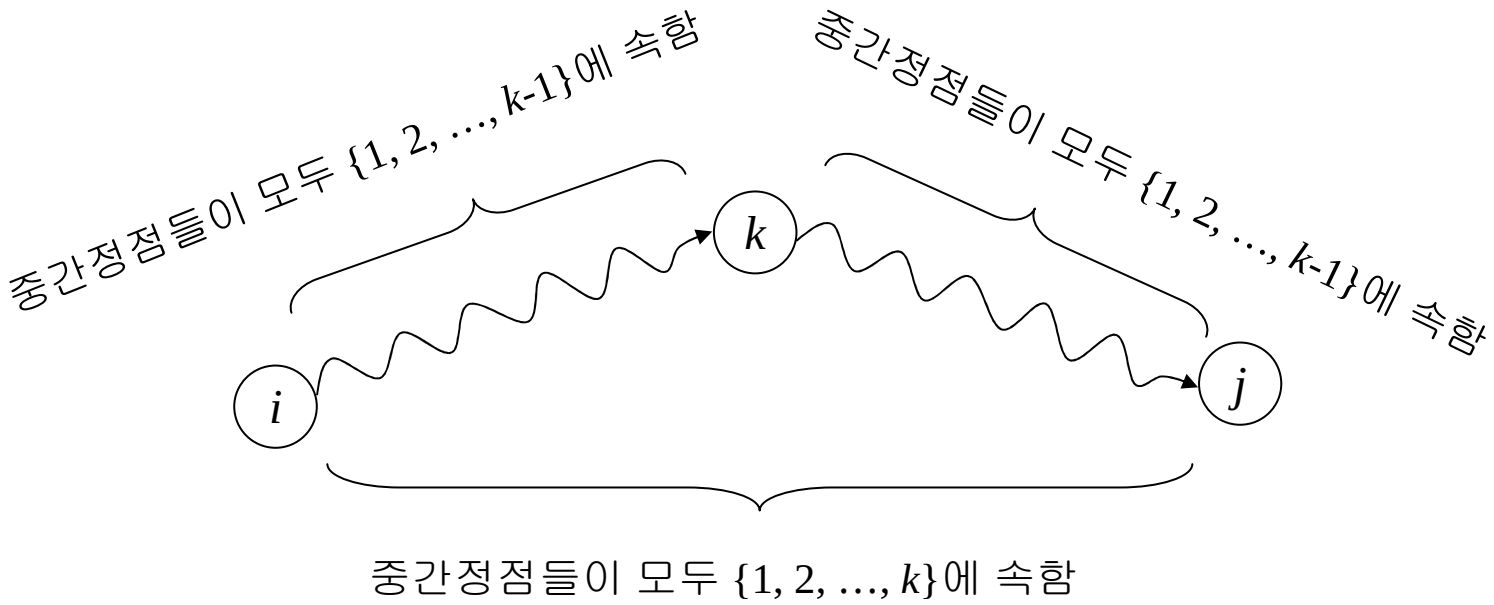
모든 쌍 최단 경로 (2/3)

- |V|*|V| 행렬 $D=(d_{ij})$
- d_{ij} – 정점 i에서 정점 j까지의 최단 경로의 길이
- $d_{ij}^{(k)}$ – 정점 번호가 k 이하인 정점만을 경유하는 정점 i에서 정점 j까지의 최단 경로의 길이

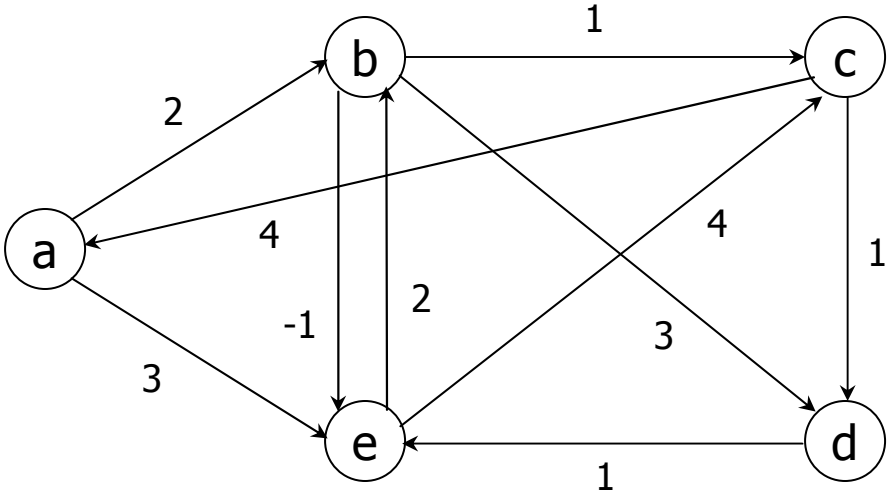
$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}), \; d_{ij}^{(0)}=w(i, j)$$

모든 쌍 최단 경로 (3/3)

- $d_{ij}^{(k)}$ 관련

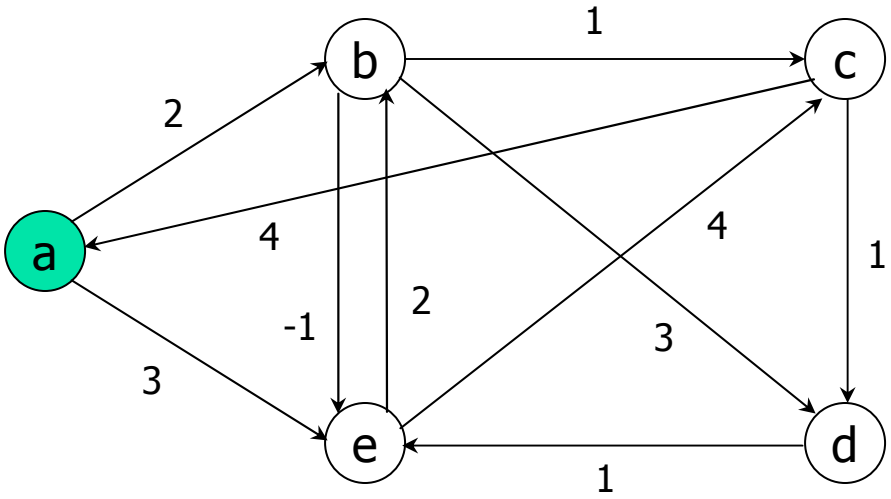


Floyd 알고리즘 (1/7)



$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & \infty & 3 \\ \infty & 0 & 1 & 3 & -1 \\ 4 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & 0 & 1 \\ \infty & 2 & 4 & \infty & 0 \end{bmatrix}$$

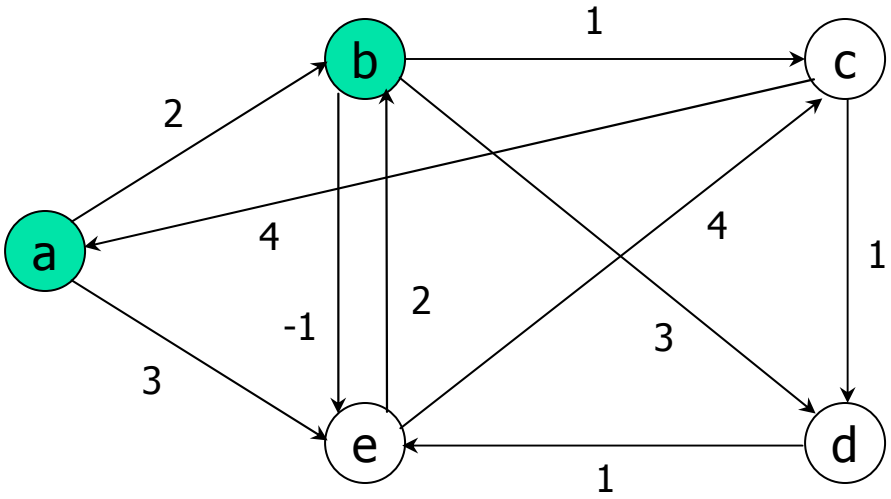
Floyd 알고리즘 (2/7)



$$D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & \infty & 3 \\ \infty & 0 & 1 & 3 & -1 \\ 4 & \underline{6} & 0 & 1 & \underline{7} \\ \infty & \infty & \infty & 0 & 1 \\ \infty & 2 & 4 & \infty & 0 \end{bmatrix}$$

$d_{35}^{(1)} = \min(d_{35}^{(0)}, d_{31}^{(0)}+d_{15}^{(0)})$
$= \min(\infty, 4+3)$
$= 7$

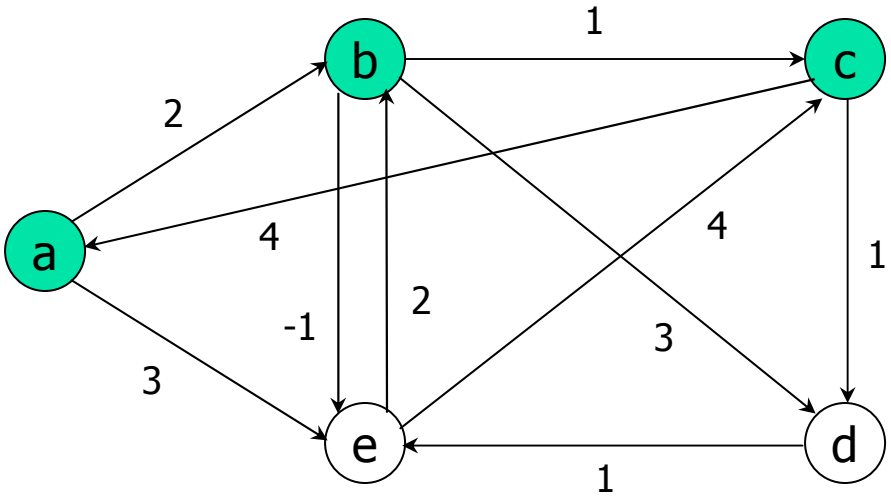
Floyd 알고리즘 (3/7)



$$D^{(2)} = \begin{bmatrix} 0 & 2 & \underline{3} & \underline{5} & \underline{1} \\ \infty & 0 & 1 & 3 & -1 \\ 4 & 6 & 0 & 1 & \underline{5} \\ \infty & \infty & \infty & 0 & 1 \\ \infty & 2 & 3 & \underline{5} & 0 \end{bmatrix}$$

$d_{35}^{(2)} = \min(d_{35}^{(1)}, d_{32}^{(1)}+d_{25}^{(1)})$
$= \min(7, 6\text{-}1)$
$= 5$

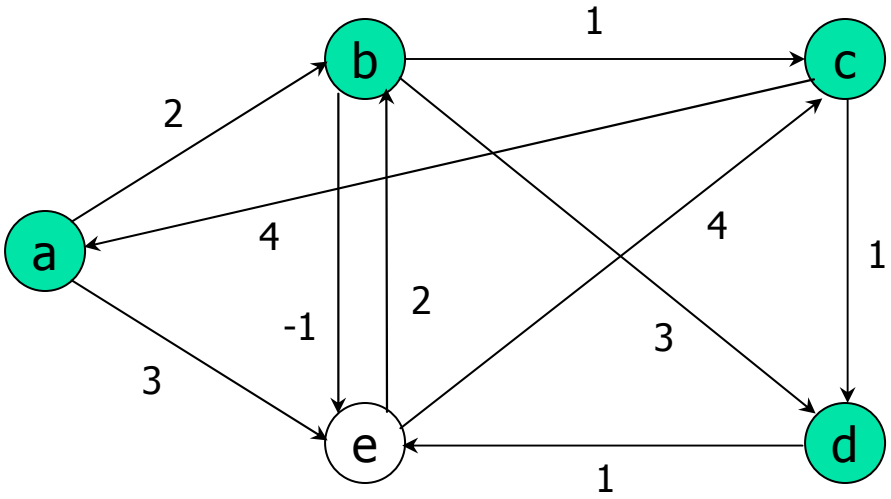
Floyd 알고리즘 (4/7)



$$D^{(3)} = \begin{bmatrix} 0 & 2 & 3 & \underline{4} & 1 \\ \underline{5} & 0 & 1 & \underline{2} & -1 \\ 4 & 6 & 0 & 1 & 5 \\ \infty & \infty & \infty & 0 & 1 \\ \underline{7} & 2 & 3 & \underline{4} & 0 \end{bmatrix}$$

$d_{35}^{(3)} = \min(d_{35}^{(2)}, d_{33}^{(2)}+d_{35}^{(2)})$
$= \min(5, 0+5)$
$= 5$

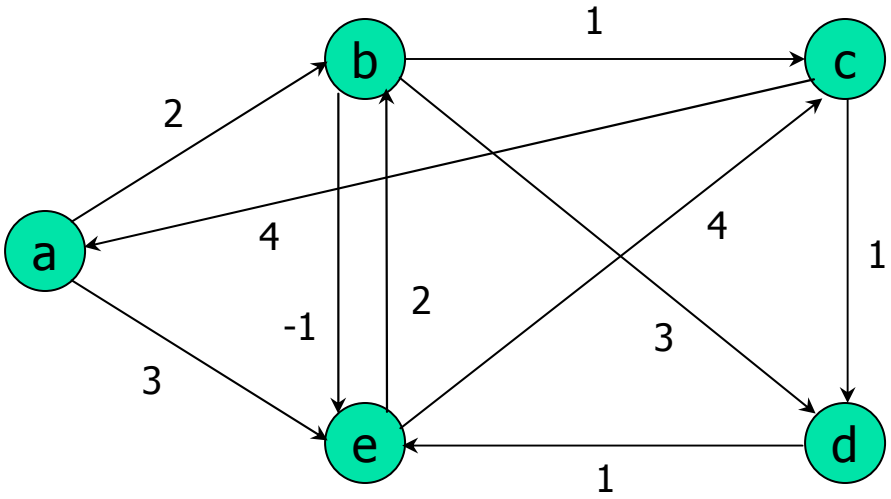
Floyd 알고리즘 (5/7)



$$D^{(4)} = \begin{bmatrix} 0 & 2 & 3 & 4 & 1 \\ 5 & 0 & 1 & 2 & -1 \\ 4 & 6 & 0 & 1 & \underline{2} \\ \infty & \infty & \infty & 0 & 1 \\ 7 & 2 & 3 & 4 & 0 \end{bmatrix}$$

$d_{35}^{(4)} = \min(d_{35}^{(3)}, d_{34}^{(3)}+d_{45}^{(3)})$
$= \min(5, 1+1)$
$= 2$

Floyd 알고리즘 (6/7)



$$D^{(5)} = \begin{bmatrix} 0 & 2 & 3 & 4 & 1 \\ 5 & 0 & 1 & 2 & -1 \\ 4 & \underline{4} & 0 & 1 & 2 \\ \underline{8} & \underline{3} & \underline{4} & 0 & 1 \\ 7 & 2 & 3 & 4 & 0 \end{bmatrix}$$

$d_{35}^{(5)} = \min(d_{35}^{(4)}, d_{35}^{(4)}+d_{55}^{(4)})$
$= \min(2, 2+0)$
$= 2$

Floyd 알고리즘 (7/7)

```
Floyd(A, n) {
/* 입력 : 인접 행렬 A[1..n][1..n],  n = |V|
   출력 : 모든 정점쌍 간의 거리 행렬 D */
        int  i,  j,  k;
1       for (i = 1;  i <= n;  i++)
2           for (j = 1;  j <= n;  j++)
3               D[i][j] = A[i][j];
4       for (k = 1;  k <= n;  k++)
5           for (i = 1;  i <= n;  i++)
6               for (j = 1; j <= n;  j++)
7                   if (D[i][j] > D[i][k] + D[k][j])
8                       D[i][j] = D[i][k] + D[k][j];
  }
```

✓수행시간 : $\Theta(|V|^3)$

<그림 6-48> 모든 쌍 최단 경로 길이를 구하는 플로이드 알고리즘

수정된 Floyd 알고리즘 (1/2)

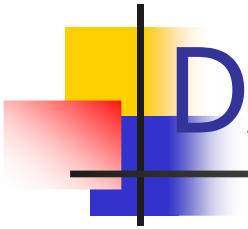
```
FloyPath(A, n) {
/* 입력 : 인접비용 행렬 A[1..n][1..n],  n = |V|
   출력 : 모든 정점쌍 간의 거리 행렬 D, 최단 선행 정점 P */

        int  i,  j,  k;
1       for (i = 1;  i <= n;  i++)
2           for (j = 1;  j <= n;  j++)  {
3               D[i][j] = A[i][j];
4               P[i][j] = 0;
5           }
6       for (k = 1;  k <= n;  k++)
7           for (i = 1;  i <= n;  i++)
8               for (j = 1;  j <= n;  j++)
9                   if (D[i][j] > D[i][k] + D[k][j])  {
10                      D[i][j] = D[i][k] + D[k][j];
11                      P[i][j] = k ;
12                  }
   }
```

<그림 6-49> 경로 자체도 구하는 수정된 플로이드 알고리즘

수정된 Floyd 알고리즘 (2/2)

```
PrintPath(int i, int j) {
        int k;
1       if ((k = P[i][j]) != 0)  {
2               PrintPath (i, k);
3               printf ("%d", k);
4               PrintPath (k, j);
5       }
    }
```

<그림 6-50> 모든 쌍 최단 경로의 인쇄



DAG의 최단 경로

- 싸이클이 없는 유향 그래프를 DAG라 한다
  - DAG : Directed Acyclic Graph

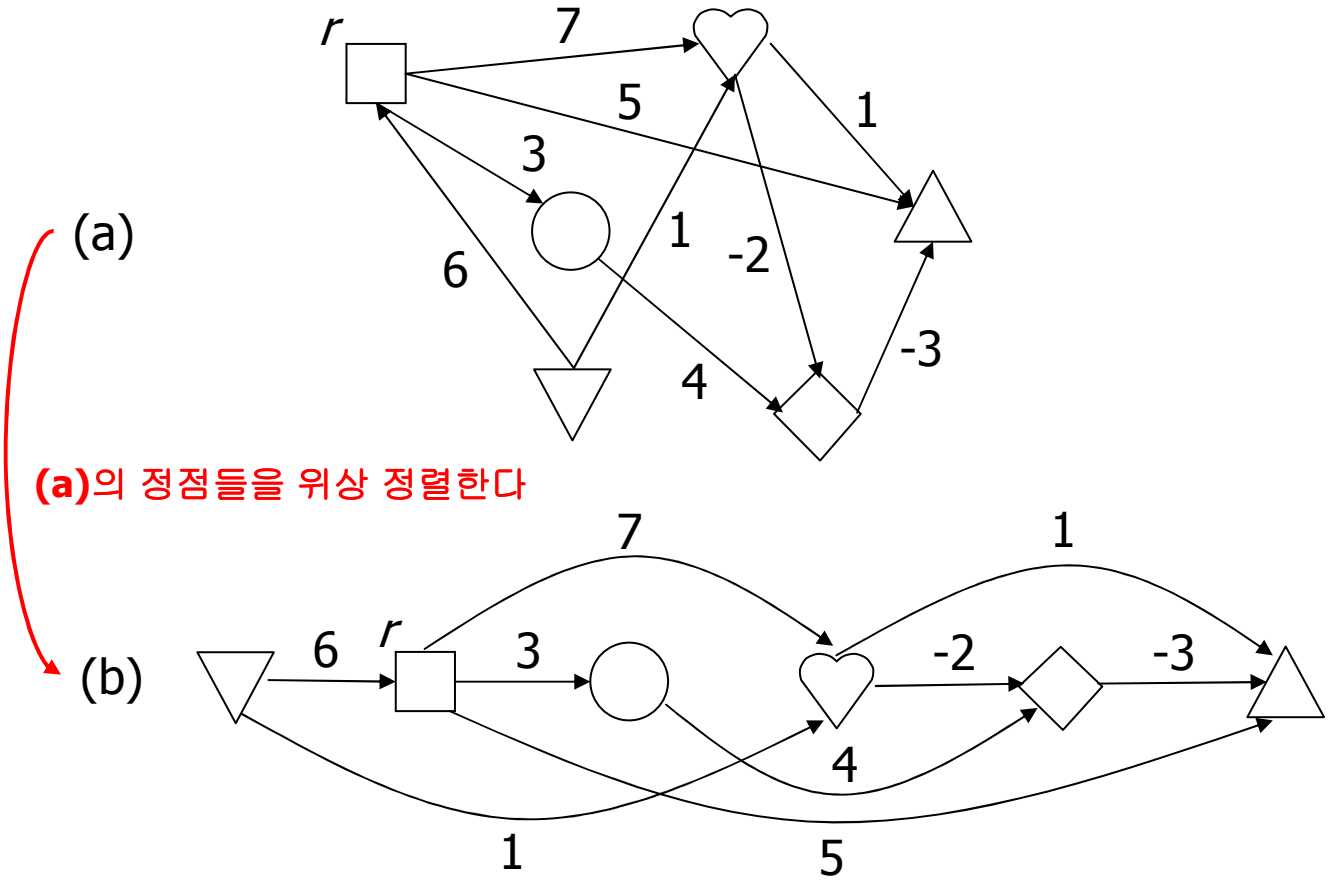
- DAG에서의 최단경로는 선형시간에 간단히 구할 수 있다

DAG의 최단경로 알고리즘

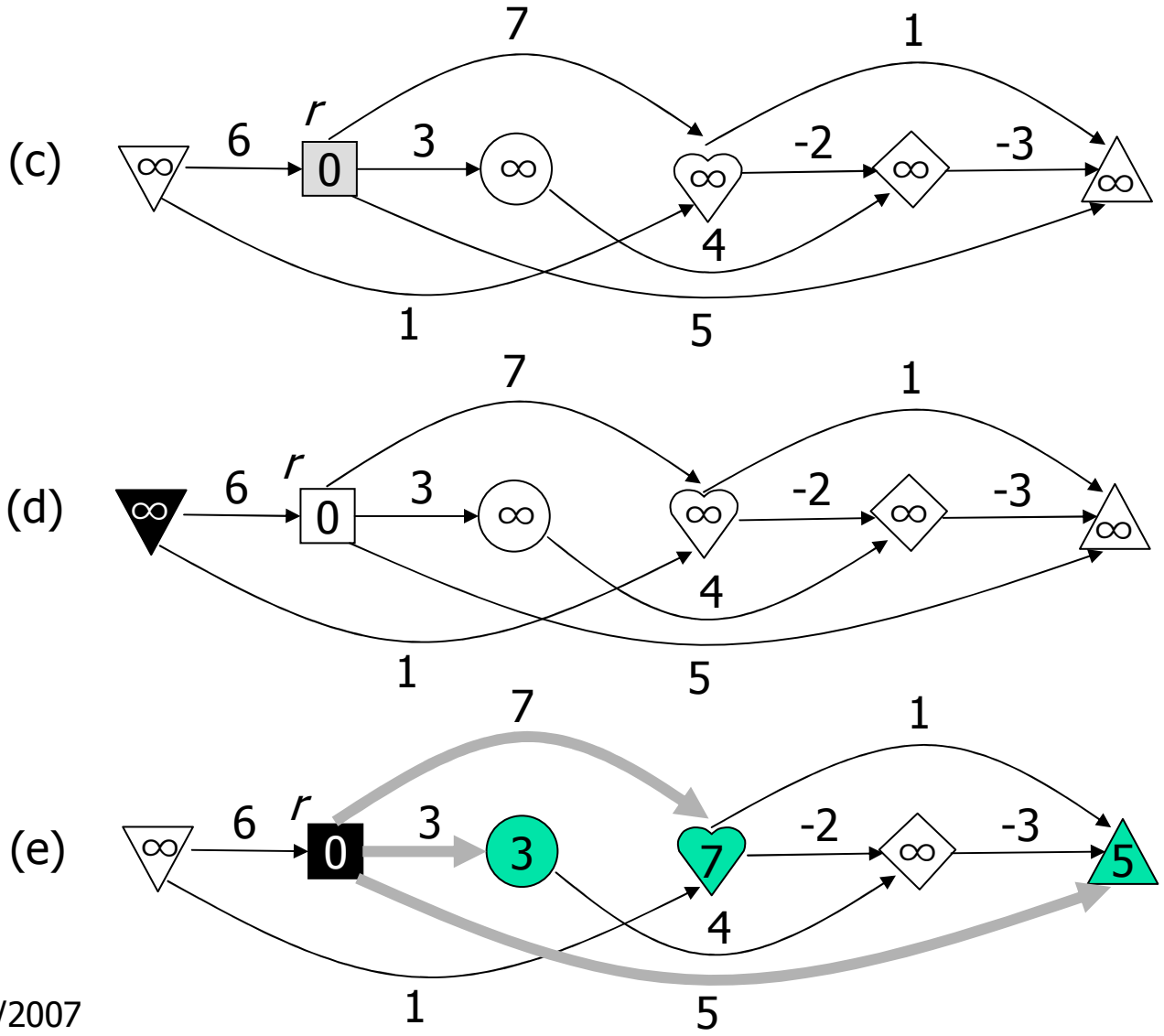
```
DAG-ShortestPath(G, r)
{
    for each u∈V
        d[u] ← ∞;
    d[r] ← 0;
    G의 정점들을 위상정렬한다;
    for each u∈V (위상정렬 순서로)
        for each v∈L(u) ▷ L(u) : 정점 u로부터 연결된 정점들의 집합
            if (d[u] + w[u, v] < d[v]) then {
                d[u] ← d[u] + w[u, v] ;
                prev[v] ← u;
            }
}
```

✓수행시간: Θ(|V|+|E|)

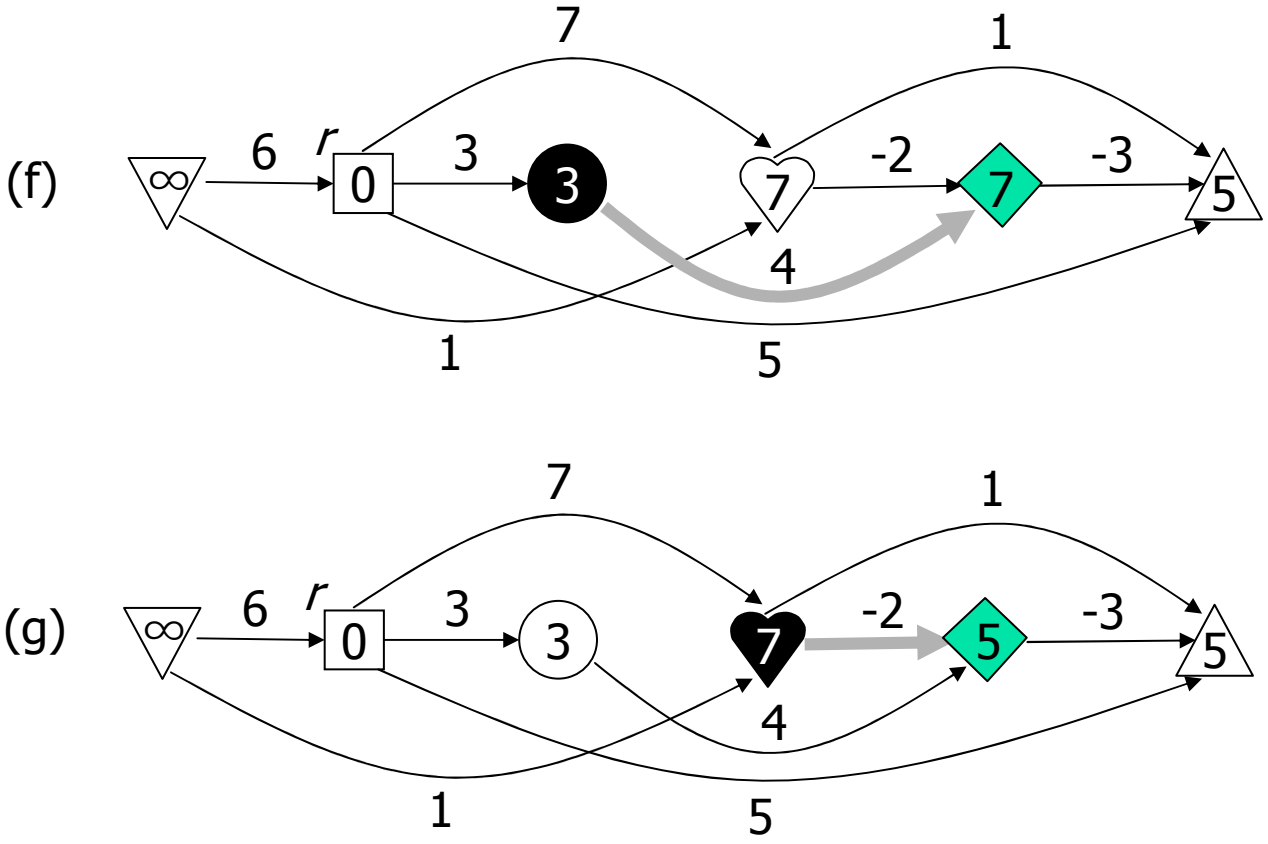
DAG-ShortestPath의 작동 예



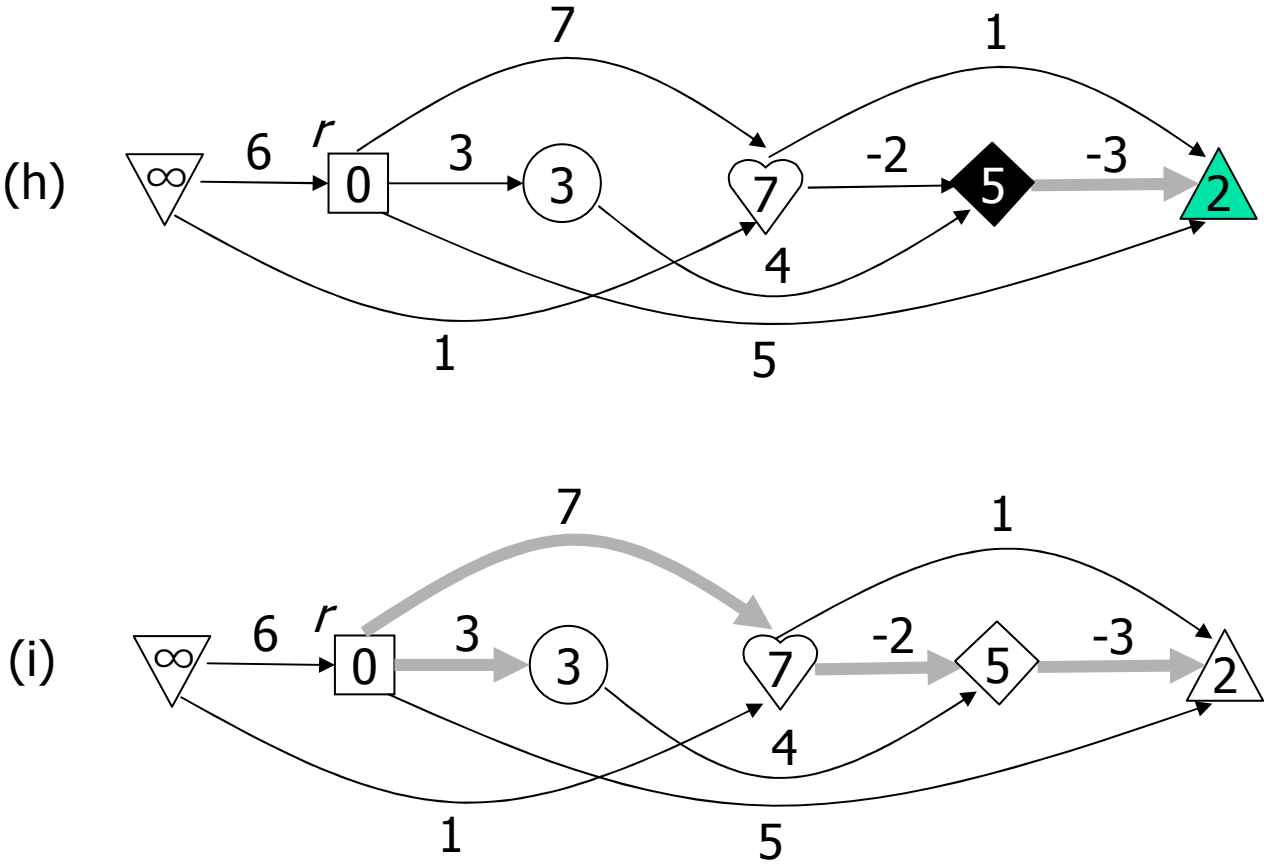
DAG-ShortestPath의 작동 예

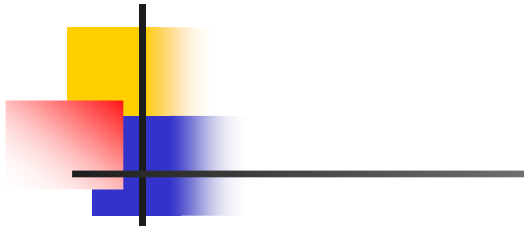


DAG-ShortestPath의 작동 예

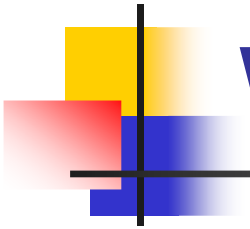


DAG-ShortestPath의 작동 예



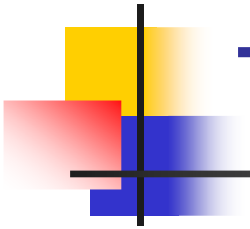


[참고] Searching & AI



What is Search For?

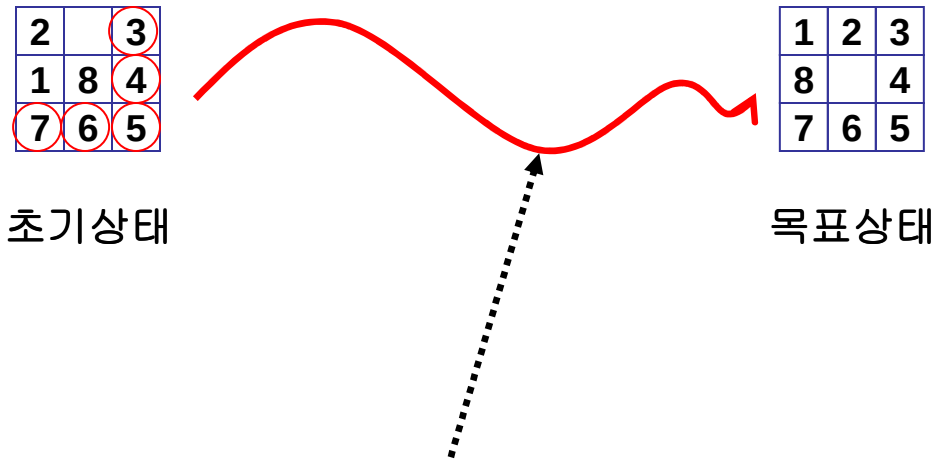
- All AI is search!
  - Not totally true (obviously) but more true than you might think.
  - Finding a good/best solution to a problem amongst many possible solutions.
- Ways of methodically deciding which solutions are better than others.



Types of Searches

- Blind searches consider alternative actions without any bias
  - **Breadth-first**
  - **Depth-first**
  - Iterative deepening
  - Uniform cost
- Heuristic searches use a heuristic to inform their choice of action
  - Greedy
  - A^*

Example: The 8-puzzle



Example: The 8-puzzle