Chapter 09

그래프 알고리즘

*** 학습목표**

- 그래프의 표현법을 익힌다.
- 너비우선탐색과 깊이우선탐색의 원리를 충분히 이해하도록 한다.
- 신장트리의 의미와 최소신장트리를 구하는 두 가지 알고리즘을 이해한다.
- 그래프의 특성에 따라 가장 적합한 최단경로 알고리즘을 선택할 수 있도록 한다.
- 위상정렬을 이해하고 DAG의 경우에 위상정렬을 이용해 최단경로를 구하는 방법을 이해한다.
- 강연결요소를 구하는 알고리즘을 이해하고 이 알고리즘의 정당성을 확신할 수 있도록 한다.
- 본문에서 소개하는 각 알고리즘의 수행시간을 분석할 수 있도록 한다.

Preview

수학은 패턴의 과학이다. 음악 역시 패턴들이다.
컴퓨터 과학은 추상화와 패턴의 형성에 깊은 관련이 있다.
컴퓨터 과학이 다른 분야들에 비해 특징적인 것은 지속적으로 차원이
급상승한다는 점이다. 미시적 관점에서
거시적 관점으로 도약하는 것이다.

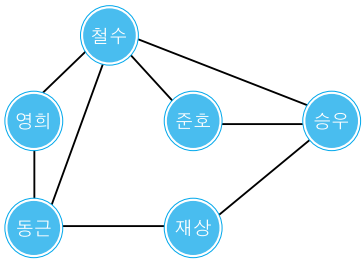
도널드 크누스

그래프는 현상이나 사물을 정점과 간선으로 표현하는 것으로, 정점은 대상이나 개체를 나타내고 간선은 이들 간의 관계를 나타낸다. 이 장에서는 컴퓨터 알고리즘에서 그래프를 표현하는 방법을 먼저 소개한 다음, 그래프에서 발생하는 여러 가지 문제를 해결하는 알고리즘들을 소개한다. 그래프의 정점들을 차례로 탐색하는 방법, 주어진 그래프로부터 최소신장트리 구하기, 싸이클이 없는 그래프에서 정점들의 순열을 구하는 위상정렬, 정점들간의 최단경로 구하기, 그래프에서 서로 강하게 연결된 부분 그래프들을 찾는 강연결요소 구하기 알고리즘들을 소개한다.

Section

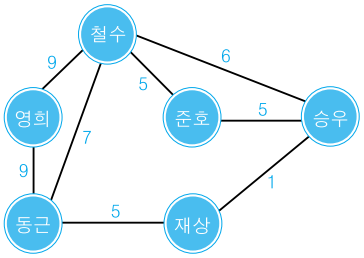
01 그래프

그래프는 현상이나 사물을 정점과 간선으로 표현하는 것으로, 정점(Vertex)은 대상이나 개체를 나타내고 간선(Edge)은 이들간의 관계를 나타낸다. 예를 들어, 사람들간의 친밀도를 나타내고 싶다고 하자. 각 사람을 하나씩의 정점으로 표시한다. 서로 친밀한 사람간에는 간선을 둔다. 그러면 [그림 9-1]과 같은 그래프가 된다. 각 간선은 간선으로 연결된 두 사람이 친밀함을 나타낸다. 간선은 두 도시를 연결하는 도로의 존재 여부, 두 집안간의 혼인 여부 등을 나타낼 수 있다.



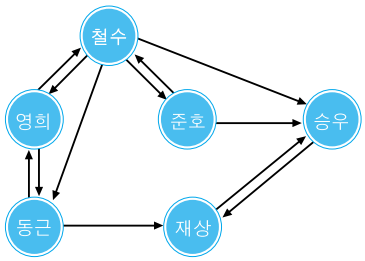
[그림 9-1] 사람들간의 친분 관계를 나타낸 그래프

[그림 9-2]는 [그림 9-1]을 조금 변형한 것이다. [그림 9-1]이 사람들간의 친밀도 존재 여부만 표기하는데 비해 [그림 9-2]는 친밀도의 크기까지 표시하고 있다. 즉, 간선에 가중치를 주어 친밀감의 정도를 보여준다. 이러한 가중치는 도시들간의 거리, 두 지점 사이에 묻힌 가스 파이프의 용량, 두 공항 사이의 비행시간 등을 나타낼 수 있다.



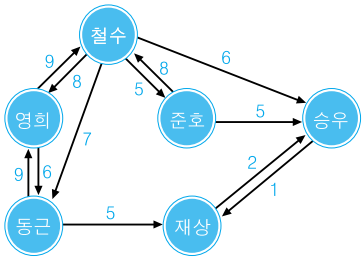
[그림 9-2] 친밀도를 가중치로 나타낸 친분 관계 그래프

[그림 9-3]은 간선이 방향을 갖는 예다. [그림 9-1]과 비교해 간선에 방향성의 개념이 들어간 것만 다르다. 즉, 각 사람이 다른 사람에게 애정을 가지고 있는가를 표시했다. 애정의 유무는 상호대칭적이지 않기 때문에 방향을 가진 간선으로 나타낼 수 있다. 방향을 가진 간선은 기업들간의 제품 공급 관계, 제품 생산 공정에서의 선후 관계 등을 나타낼 수 있다. [그림 9-1]과 같은 그래프로는 나타낼 수 없는 일방통행 도로의 존재를 나타낼 수도 있다. 이러한 그래프를 방향을 가진 그래프 또는 유향 그래프(Directed Graph)라고 한다. 반면, 간선의 방향이 없는 그래프를 무향 그래프(Undirected Graph) 또는 무방향 그래프라고 한다.



[그림 9-3] 방향을 고려한 친분 관계 그래프

[그림 9-4]는 [그림 9-3]의 방향을 가진 간선이 가중치를 갖는 경우다. 누가 누구를 얼마만큼 좋아하는지를 나타내고 있다. 서울과 LA간의 비행기 노선처럼 가는 데 걸리는 시간과 오는 데 걸리는 시간이 다른 경우를 나타낼 수도 있다.



[그림 9-4] 가중치를 가진 유향 그래프

이 장에서는 이러한 다양한 그래프를 이용해 문제를 해결하는 알고리즘들을 소개한다.

n개의 정점의 집합 V와 이들간에 존재하는 간선의 집합 E로 구성된 그래프 G를 보통 $G=(V, E)$로 표시한다.

Section

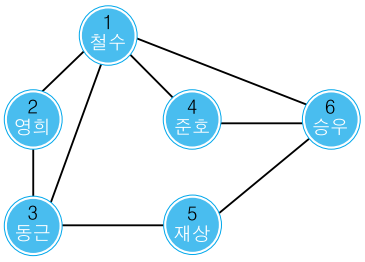
2 그래프의 표현

이 절에서는 컴퓨터에서 그래프를 표현하는 방법을 공부한다. 그래프의 표현법에는 크게 행렬을 이용하는 방법과 리스트를 이용하는 방법이 있다.

1 인접행렬을 이용한 방법

우선, 앞 절에서 살펴본 [그림 9-1]처럼 방향성이 없고, 가중치도 없는 단순 관계 그래프를 표현하는 방법을 알아보자.

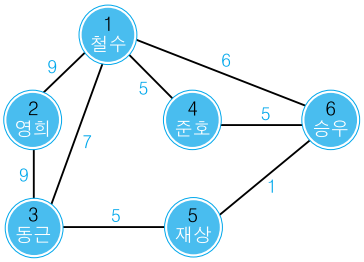
그래프 $G=(V, E)$에서 정점의 총수가 n이라 하자. 우선 $n \times n$ 행렬을 준비한다. 정점 i와 정점 j간에 간선이 있으면 행렬의 (i, j)원소와 (j, i)원소의 값을 1로 할당한다. 간선으로 연결된 두 정점은 인접하다(Adjacent)고 한다. 이런 식으로 모든 간선에 대해서 행렬의 해당 원소에 1을 할당하고, 나머지 원소에는 0을 할당한다. [그림 9-5]는 단순 관계 그래프를 인접행렬(Adjacency Matrix)로 표현한 것이다.



	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

[그림 9-5] 무방향 그래프와 이를 인접행렬로 표현한 예

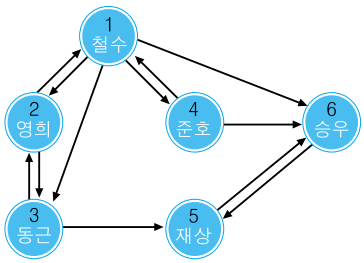
[그림 9-6]은 가중치가 있는 그래프를 인접행렬로 표현한 예다. 행렬의 각 원소에 1 대신 가중치를 저장한다는 점이 [그림 9-5]와 다르다.



	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

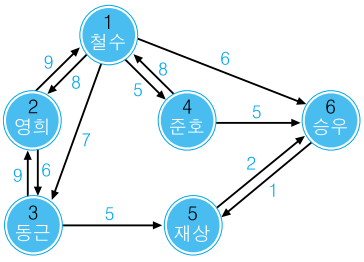
[그림 9-6] 가중치를 가진 무방향 그래프와 인접행렬 표현

[그림 9-7]과 [그림 9-8]은 유향 그래프를 인접행렬로 표현한 두 예다. 간선이 방향을 가지므로 인접행렬은 [그림 9-5], [그림 9-6]과 달리 대칭이 아니다.



	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

[그림 9-7] 유향 그래프와 인접행렬 표현



	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

[그림 9-8] 가중치를 가진 유향 그래프와 인접행렬 표현

행렬 표현법은 이해하기 쉽고 간선의 존재 여부를 즉각 알 수 있다는 장점이 있다. 정점 i와 정점 j의 인접 여부는 행렬의 (i, j)원소나 (j, i) 원소의 값만 보면 알 수 있기 때문이다. 대신 $n \times n$ 행렬이 필요하므로 n^2에 비례하는 공간이 필요하고, 행렬의 준비 과정에서 행렬

의 모든 원소를 채우는 데만 n^2에 비례하는 시간이 든다. 그러므로 $O(n^2)$ 미만의 시간이 소요되는 알고리즘이 필요한 경우에 행렬 표현법을 사용하면 행렬의 준비 과정에서만 $\Theta(n^2)$의 시간을 소모해버려 적절하지 않다.

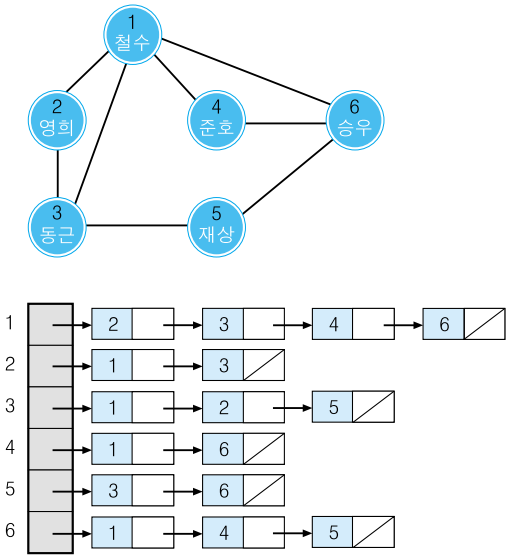
간선의 밀도가 아주 높은 그래프에서는 인접행렬 표현이 적합하다. 예를 들어, 전체 (i, j) 쌍에서 둘 중 하나 꼴로 간선이 있는 경우는 행렬을 사용하는 것이 효율적이다. 그렇지만 100만 개의 정점을 가진 그래프에서 간선이 200만 개밖에 없는 경우에 행렬 표현을 쓰면 시간과 공간이 많이 낭비된다. 행렬의 총 원소 수는 1조 개인데 이 중 고작 200만(또는 400만) 개만 1로 채워지고 나머지는 0이기 때문이다.

2 인접리스트를 이용한 방법

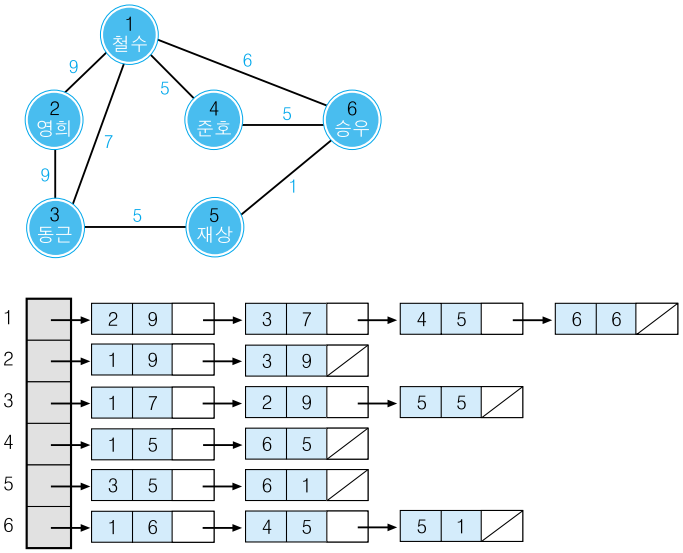
인접리스트(Adjacency List) 표현법은 각 정점에 인접한 정점들을 리스트로 표현하는 방법이다. 각 정점마다 리스트를 하나씩 만든다. 여기에 각 정점에 인접한 정점들을 연결 리스트(Linked List)로 매단다. 인접리스트 표현에서는 행렬 표현과 달리 존재하지 않는 간선은 표현상에 나타나지 않는다.

[그림 9-9]는 단순 관계 그래프를 인접리스트로 표현한 것이다. 간선 하나에 대하여 노드가 두 개씩 만들어진다. 각 노드는 〈정점 번호, 다음 정점의 포인터〉로 구성된다. 이와 같은 무방향 그래프를 위한 인접리스트 표현에서 필요한 총 노드 수는 존재하는 총 간선 수의 두 배다. 정점 i와 정점 j가 인접하면 정점 i의 연결 리스트에 정점 j가, 정점 j의 연결 리스트에 정점 i가 매달려 한 간선당 노드가 두 개씩 만들어지기 때문이다. 유향 그래프의 경우에는 간선 하나당 노드가 하나씩 존재한다.

[그림 9-10]은 가중치를 가진 그래프를 인접리스트로 표현한 예다. 각 노드에는 정점 번호, 다음 노드의 포인터와 함께 해당 간선의 가중치를 저장하는 부분이 추가된다. 즉, 각 노드는 〈정점 번호, 가중치, 다음 정점의 포인터〉로 구성된다.



[그림 9-9] 무방향 그래프와 인접리스트 표현



[그림 9-10] 가중치를 가진 그래프의 인접리스트 표현

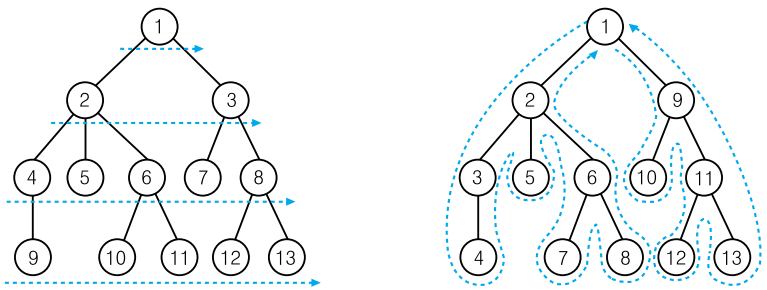
인접리스트는 공간이 간선의 총수에 비례하는 양만큼만 필요하므로 대체로 행렬 표현에 비해 공간의 낭비가 없다. 모든 가능한 정점 쌍에 비해 간선의 수가 적을 때 특히 유용하다. 그렇지만 거의 모든 정점 쌍에 대해 간선이 존재하는 경우에는 오히려 리스트를 만드는 데 필요한 오버헤드만 더 든다. 인접리스트는 정점 i와 정점 j간에 간선이 존재하는지를 알아볼 때 리스트에서 차례대로 훑어야 하므로 인접행렬 표현보다는 시간이 많이 걸린다. 특히 간선이 많은 경우에는 최악의 경우 n에 비례하는 시간이 들 수도 있다. 이 부분이 이를 이용하는 알고리즘의 성능에 치명적인 영향을 미칠 수 있다. 그래서 인접리스트 표현법은 간선의 밀도가 아주 높은 경우에는 그리 적합하지 않다. 그럴 때는 앞서 언급한대로 인접행렬 표현법을 쓰는 것이 좋다.

Section

03 너비우선탐색(BFS)과 깊이우선탐색(DFS)

그래프에서 모든 정점을 방문해야 할 때가 자주 있다. 모든 정점을 방문하는 방법은 다양하지만 대표적인 것이 이 절에서 소개할 너비우선탐색(BFS, Breadth-First Search)과 깊이우선탐색(DFS, Depth-First Search)이다. 두 방법은 매우 간단하지만 그래프 알고리즘에서 핵심적 위치를 차지한다. 그러므로 이 두 탐색 방법에 대해 아주 깊은 직관을 갖고 있을 필요가 있다.

일반적인 그래프를 대상으로 하는 두 탐색에 대한 설명에 앞서 직관적인 이해를 돕기 위해 트리를 대상으로 설명을 시작하자. 루트를 시작으로 탐색을 한다면 너비우선탐색(BFS)은 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. 이에 반해 깊이우선탐색(DFS)은 루트의 자식 정점을 하나 방문한 다음, 아래로 내려갈 수 있는 곳까지 내려간다. 더이상 내려갈 수가 없으면 위로 되돌아오다가 내려갈 곳이 있으면 즉각 내려간다. [그림 9-11]은 트리를 대상으로 한 너비우선탐색(BFS)과 깊이우선탐색(DFS)의 예다. BFS는 옆으로(너비로) 쭉 훑어나가고, DFS는 아래로(깊이로) 갈 수 있는 데까지 갔다가 막히면 되돌아와서 다시 내려간다. 너비우선탐색과 깊이우선탐색이라고 부르는 이유를 짐작할 수 있을 것이다.

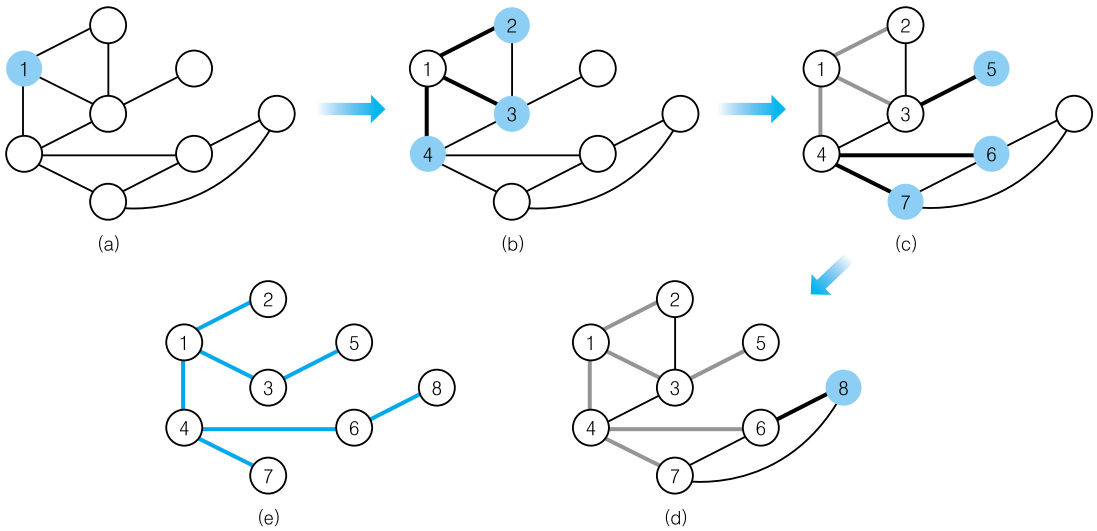


(a) BFS　　(b) DFS

[그림 9-11] 트리를 대상으로 한 BFS와 DFS

이제는 일반적인 그래프 $G=(V, E)$에 대해 BFS와 DFS를 해보자. [그림 9-12]는 무방향 그래프에 대해 BFS를 수행하는 예다.

- **그림 (a)**: 시작정점으로 정해진 정점을 방문한다(1로 표시했다).
- **그림 (b)**: (a)에서 방문한 정점에 인접한 정점을 모두 방문한다(각각 2, 3, 4로 표시했다).
- **그림 (c)**: 정점 2에 인접한 정점 중 방문하지 않은 정점은 없다. 정점 3에 인접한 정점 중 방문하지 않은 정점을 모두 방문한다. 하나밖에 없다(5로 표시했다). 정점 4에 인접한 정점 중 방문하지 않은 정점을 모두 방문한다. 두 개가 있다(각각 6, 7로 표시했다).
- **그림 (d)**: 정점 5에서 인접한 정점 중 방문하지 않은 정점은 없다. 정점 6에 인접한 정점중 방문하지 않은 정점을 모두 방문한다. 하나밖에 없다(8로 표시했다). 정점 7에 인접한 정점 중 방문하지 않은 정점은 없다.
- 마지막으로 정점 8에 인접한 정점 중 방문하지 않은 정점이 없어 더이상 갈 곳이 없으므로 끝낸다.



[그림 9-12] BFS의 작동 예

그림에서 검은색으로 굵게 표시한 간선들은 각 정점을 처음으로 방문할 때 사용한 간선들이다. 그래프 G에서 이 간선들만 남기면 트리가 되는데 이 트리를 **너비우선트리**라 한다(그림 (e)).

위의 예와 같이 작동하는 BFS 알고리즘을 기술하면 [알고리즘 9-1]과 같다.

[알고리즘 9-1] BFS 알고리즘

```
BFS(G, s)
{
    for each v ∈ V − {s}
        visited[v] ← NO;
    visited[s] ← YES;                ▷ s : 시작정점
    enqueue(Q, s);                   ▷ Q: 큐
    while (Q ≠ ∅) {
        u ← dequeue(Q);
        for each v∈L(u)              ▷ L(u): 정점 u의 인접리스트
            if (visited[v] = NO) then {
                visited[v] ← YES;
                enqueue(Q, v);
            }
    }
}
```

먼저 시작정점을 제외한 모든 정점을 "방문하지 않았다"($visited[v] \leftarrow$ NO)로 표시한다. 큐의 맨 앞에 있는 정점을 빼내고 이에 인접한 정점 중 방문하지 않은 정점을 모두 "방문하였다"($visited[v] \leftarrow$ YES)로 표시하고 큐에 넣는다. BFS가 수행되는 동안 euqueue()와 dequeue()를 각각 정확히 $|V|$번씩 호출한다. 즉, 각 정점이 큐에 한 번씩 들어갔다가 나온다. BFS의 수행시간은 $\Theta(V+E)$이다.

이번에는 DFS를 살펴보자. [그림 9-13]은 무방향 그래프에 대해 DFS를 수행하는 예다.

- **그림 (a)**: 시작정점으로 정해진 정점을 방문한다(1로 표시했다).
- **그림 (b)**: 정점 1에 인접한 정점 중 하나를 방문한다(2로 표시했다).
- **그림 (c)**: 정점 2에 인접하면서 방문하지 않은 정점은 총 세 개다. 이 중 하나를 방문한다(3으로 표시했다).
- **그림 (d)**: 정점 3에 인접한 정점 중 방문하지 않은 정점은 두 개다. 이 중 하나를 방문한다(4로 표시했다).
- **그림 (e)**: 정점 4에 인접한 정점 중 방문하지 않은 정점은 하나뿐이다. 이를 방문한다

혼동의 염려가 없으면 정점의 수 $|V|$ 대신 V를, 간선의 수 $|E|$ 대신 E를 사용하기도한다. 즉, $\Theta(V+E)$는 $\Theta(|V|+|E|)$를 의미한다.

(5로 표시했다). 5에 인접한 정점 중 방문하지 않은 정점은 없다. 따라서 왔던 길로 되돌아간다. 정점 4, 3, 2로 되돌아가는 과정에 정점 4, 3에 인접한 정점 중 방문하지 않은 정점은 없다.

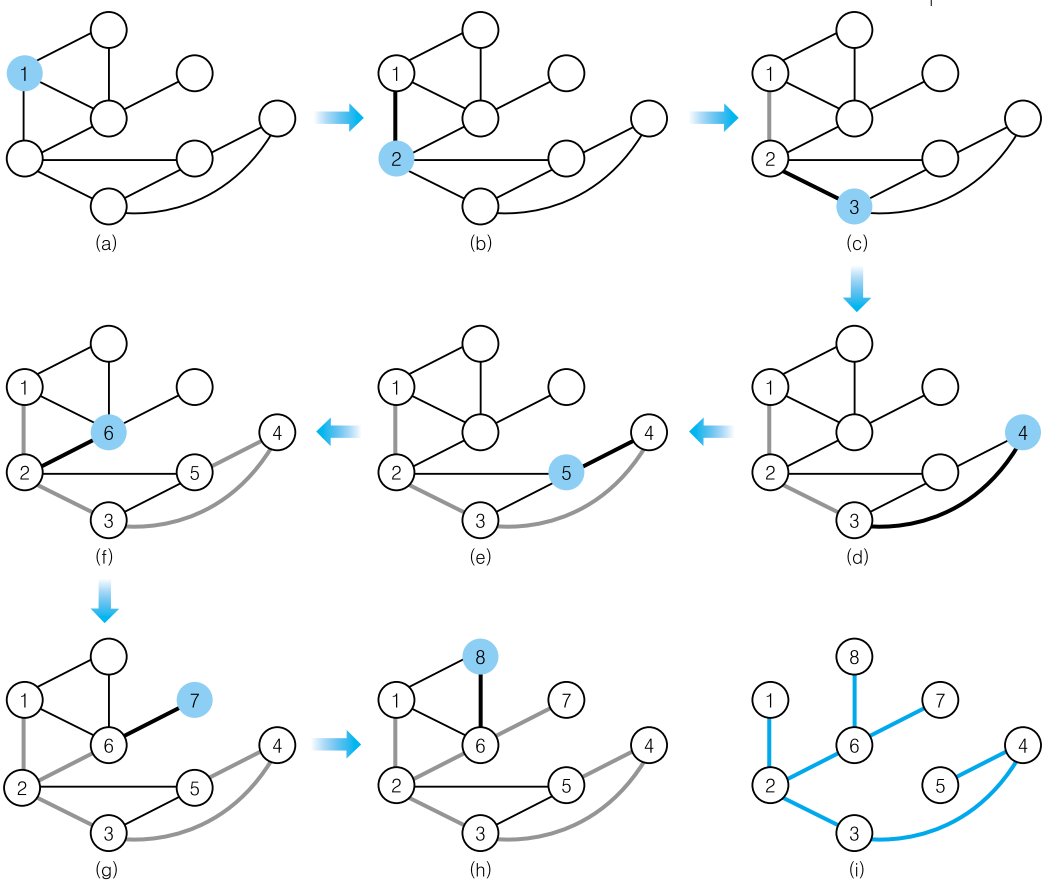
- **그림 (f)**: 정점 2로 되돌아오면 이에 인접한 정점 중 방문하지 않은 정점이 하나 있다. 이를 방문한다(6으로 표시했다).
- **그림 (g)**: 정점 6에 인접한 정점 중 방문하지 않은 정점은 두 개다. 이 중 하나를 방문한다(7로 표시했다).
- **그림 (h)**: 정점 7에 인접한 정점 중 방문하지 않은 정점은 없다. 정점 6으로 돌아간다. 정점 6에 인접한 정점 중 방문하지 않은 정점이 하나 있다. 이를 방문한다(8로 표시했다). 정점 8에서 인접한 정점 중 방문하지 않은 정점은 없다. 정점 6으로 돌아간다. 정점 6에서도 인접한 정점 중 방문하지 않은 정점은 없다. 정점 2로 돌아간다. 정점 2에서도 방문하지 않은 인접 정점이 남아있지 않다. 정점 1로 돌아간다. 정점 1에서도 방문하지 않은 인접 정점이 남아있지 않다. 시작정점(정점 1)에서 더이상 갈 곳이 없으므로 끝낸다.

그림에서 검은색으로 굵게 표시한 간선들은 각 정점을 처음으로 방문할 때 사용된 간선들이다. [그림 9-13]의 (i)는 [그림 9-12]의 (e)처럼 각 정점을 처음으로 방문할 때 사용한 간선들로 만들어진 트리다. 이 트리를 **깊이우선트리**라 한다. [그림 9-13]의 예와 같이 작동하는 DFS 알고리즘을 기술하면 [알고리즘 9-2]와 같다.

[알고리즘 9-2] DFS 알고리즘

```
DFS(G)
{
    for each v ∈ V
        visited[v] ← NO;
❶   for each v ∈ V
        if(visited[v] = NO) then aDFS(v);
}
aDFS(v)
{
    visited[v] ← YES;
    for each x ∈ L(v)          ▷ L(v): 정점 v의 인접리스트
        if (visited[x] = NO) then aDFS(x);
}
```

정점 v에 대해 aDFS(v)가 호출되면 먼저 정점 v를 "방문하였다"($visited[v] \leftarrow$ YES)로 표시하고 이와 인접한 정점 중 방문하지 않은 정점에 대해 각각 aDFS를 호출한다. 만약 정점 v에 인접한 정점 중 방문하지 않은 정점 x, y, z가 있더라도 x에 대해 aDFS를 호출하면 aDFS는 일단 깊이 들어갈 수 있는 데까지 들어가기 때문에 정점 y나 z가 다른 정점과 인접하면 방문될 수도 있다. 즉, aDFS(v)에서 x에 대해 aDFS를 끝내고 돌아오면 방문하지 않은 상태였던 정점 y나 z가 방문된 상태가 되어 이들에 대해 aDFS를 수행할 필요가 없게 되는 상황은 매우 흔하다. 궁극적으로 모든 정점에 대해 aDFS()가 한 번씩 호출된다. DFS의 수행시간은 $\Theta(V+E)$이다.



[그림 9-13] DFS의 작동 예

Section

04 최소신장트리

그래프 $G=(V, E)$의 신장트리는 정점 집합 V를 그대로 두고 간선을 $|V|-1$개만 남겨 트리가 되도록 만든 것이다. 임의의 그래프로부터 만들 수 있는 신장트리는 매우 다양하다. 앞 절에서 배운 너비우선트리와 깊이우선트리도 신장트리들이다. 간선들이 가중치를 갖는 그래프에서 간선 가중치의 합이 가장 작은 트리를 **최소신장트리**(Minimum Spanning Tree)라 한다. 이 절에서는 최소신장트리를 찾는 알고리즘들을 배운다.

1 프림 알고리즘

프림(Prim) 알고리즘은 집합 S를 공집합에서 시작하여 모든 정점을 포함할 때까지(즉, $S=V$가 될 때까지) 키워 나간다. 맨 처음 정점을 제외하고는 정점을 하나 더할 때마다 간선이 하나씩 확정된다. 프림 알고리즘을 집합 S를 중심으로 기술하면 [알고리즘 9-3]과 같다.

[알고리즘 9-3]에서 배열 $d[]$는 각 정점을 신장트리에 포함시키는 데 드는 최소비용을 구하기 위한 것으로 계속 수정해 나간다. 궁극적으로 $d[v]$는 각 정점 v를 신장트리에 연결하기 위해 드는 최소 비용을 갖게 되지만 알고리즘의 수행과정에서는 큰 값부터 시작하여 이 최소 비용을 향해 계속 감소한다. $tree[v]$는 현재까지의 계산 결과 정점 v를 신장트리에 연결시키는 비용이 가장 적게 드는 간선을 저장한다(실제로는 해당 간선에서 v의 맞은 편에 있는 정점을 저장한다). 알고리즘이 끝난 후 이것을 이용해서 최소신장트리를 만들 수 있다.

[알고리즘 9-3] 프림 알고리즘 (버전 1)

```
Prim(G, r)
▷ G=(V, E): 주어진 그래프
▷ r: 시작 정점
{
        S ← ø;                          ▷ S: 정점 집합
    ❶ for each u∈V
            d[u] ← ∞;
        d[r] ← 0;
    ❷ while (S≠V) {                     ▷ n회 순환한다.
        ❸ u ← extractMin(V−S, d);
          S ← S∪{u};
        ❹ for each v∈L(u)               ▷ L(u): u의 인접 정점 집합
            ❺ if (v∈V−S and w(u, v)<d[v]) then {
                ❻ d[v] ← w(u, v);
                ❼ tree[v] ← u;
              }
    }
}

extractMin(Q, d[])
{
    집합 Q에서 d값이 가장 작은 정점 u를 리턴한다;
}
```

프림 알고리즘을 처음 접하는 독자라면 앞의 설명만으로 재빨리 이해하기는 힘들 것이다. [그림 9-14]를 보면서 설명하자. 그림은 7개의 정점이 가중치를 가진 간선들로 연결되어 있는 그래프다.

- **그림 (a)**: 시작 정점 r의 연결비용을 0으로, 나머지 정점들의 연결비용을 ∞로 초기화한 이다.

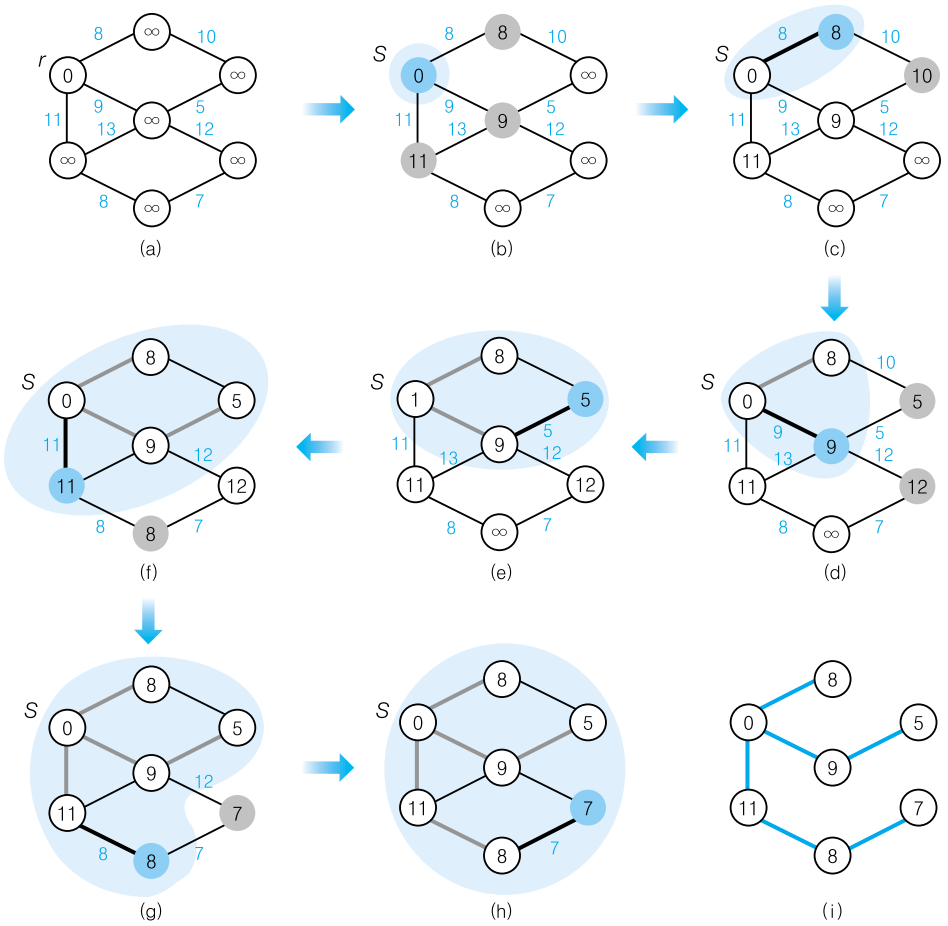
$n = |V|$

- **그림 (b)**: 먼저 정점 r을 집합 S의 첫 번째 원소로 포함시킨다. 이제 집합 S 바깥의 정점들 중 정점 r에 연결된 정점들을 살핀다. 정점 r과 이 정점들을 연결하는 비용은 각각 8, 9, 11이다. 이 비용을 각 정점에 기록해둔다. 그림에서 음영으로 표시된 영역이 집합 S를 나타낸다. 집합 S는 (a)에서 공집합으로 시작하여 원소가 하나씩 늘어나 (h)에서 최종적으로 n개가 됨을 볼 수 있다. 각 정점에 기록된 값은 해당 정점을 집합 S에 있는 정점과 연결하는 데(집합 S에 포함시키는 데) 드는 최소비용이다. 그림 (b)에서 집합 S는 정점 r만으로 구성되어 있으므로 이와 연결된 세 정점은 각각 8, 9, 11의 값을 갖고, 아직 집합 S와 연결 관계가 없는 나머지 정점들은 모두 ∞의 값을 갖는다.
- **그림 (c)**: 집합 S에 연결하는 데 가장 최소비용이 드는 정점을 새로이 집합 S에 포함시킨다. 8의 비용이 드는 정점이다. 이 정점을 집합 S에 포함시키고 나니 한 정점을 집합 S에 연결하는 비용이 ∞에서 10으로 바뀌었다. 집합 S의 바깥에서 회색으로 표시한 정점들은 이와 같이 방금 집합 S가 확장되면서 이에 연결하는 최소 비용이 바뀐 정점들을 나타낸다. 이 최소 비용의 변경은 [알고리즘 9-3]의 ❻에서 수행된다. 이제 집합 S 바깥의 정점들 중 연결비용이 가장 낮은 것의 비용은 9이다.
- **그림 (d)**: 연결비용이 9인 정점을 새로이 집합 S에 포함시킨다. 이로 인해 한 정점의 연결비용이 ∞에서 12로 바뀌고, 한 정점은 연결비용이 10에서 5로 바뀌었다. 정점의 연결비용이 ∞에서 바뀌는 것은 이 정점이 최초로 집합 S에 연결할 수 있게 되었을 때다. 정점의 연결비용이 ∞가 아닌 수에서 바뀌는 것은 이미 연결은 할 수 있는 상태가 되었지만 더 좋은 연결 방법이 발견되었을 때다. 이런 식으로 정점의 연결비용은 여러 번 바뀔 수도 있다. 이제 나머지 정점 중 연결비용이 최소인 것은 비용이 5이다.
- **그림 (e)**: 이 정점을 새로이 집합 S에 포함시킨다. 이 경우에는 정점이 S에 추가되면서 다른 정점의 연결비용에 미치는 영향은 없다.
- **그림 (f)**: 연결비용 11인 정점이 S에 추가된다. 이로 인해 한 정점의 연결비용이 ∞에서 8로 바뀌었다.
- **그림 (g)**: 연결비용 8인 정점이 S에 추가된다. 이로 인해 한 정점의 연결비용이 12에서 7로 바뀌었다. 이것도 연결비용이 두 번째 바뀐 것이다.
- **그림 (h)**: 마지막 남은 정점이 7의 비용으로 연결되면서 최소신장트리가 완성되었다.

최종적인 최소신장트리는 (i)와 같다. 그림의 표기를 다시 정리하면, 음영으로 표시된 영역이 집합 S를 나타낸다. 집합 S 안에서 별색으로 표시된 것은 방금 집합 S로 들어간 정점이다. 집합 S의 바깥에서 회색으로 표시된 것은 방금 집합 S로 들어간 정점과 관계되어 최소연결비용(d값)에 변동이 생긴 정점이다.

방금 집합 S로 들어간 정점에 의해서 변동이 생기는 경우도 있고, (e)처럼 변동이 전혀 생기지 않는 경우도 있다. 어떤 정점은 변동이 여러 번 생기기도 한다. 그림 (c)에서 $\infty \rightsquigarrow 10$으로 바뀐 정점은 (d)에서 다시 5로 바뀌었다. (d)에서 $\infty \rightsquigarrow 12$로 바뀌었다가 (g)에서 다시 7로 바뀐 예도 볼 수 있다. 이렇게 값이 바뀌는 것을 **이완**(Relaxation)이라고 한다. 굵은 선으로 표시된 간선들이 최소신장트리를 구성하는 간선들이다. 정점이 S로 하나씩 들어갈 때마다 간선 하나가 같이 굵어짐을 볼 수 있다. 알고리즘의 ❼에 대응된다. 즉, 해당 정점을 집합 S로 집어넣는 데 기여한 간선이고 해당 정점의 연결비용을 마지막으로 바꾼 간선이다. 이것은 해당 정점을 최소신장트리에 연결시키는 간선이다.

영어에서 실제치와 추정치(또는 근사치)의 괴리가 크면 긴장도가 높다고 표현한다. 값이 바뀌면 이 긴장도가 낮아지므로 "이완"이라는 용어를 쓴다.



[그림 9-14] 프림 알고리즘의 작동 예

프림 알고리즘을 [알고리즘 9-4]와 같이 기술할 수도 있다. 앞의 버전은 집합 S를 중심으로 기술한 것이고, 다음 버전에서는 집합 S를 제외한 나머지, 즉 $Q=V-S$를 중심으로 기술하였다. 이 변화 이외에는 완전히 똑같은 알고리즘이다. 직관적으로는 집합 S를 바로 다루는 첫 번째 버전이 이해하기에 좀 나을지 모르나 구현하는 데는 두 번째 버전이 좀 나을 것이다.

[알고리즘 9-4] 프림 알고리즘 (버전 2)

```
Prim(G, r)
▷ G=(V, E): 주어진 그래프
▷ r: 시작 정점
{
    Q ← V;                                  ▷ Q: S에 속하지 않은 정점 집합
  ❶ for each u ∈ Q
        d[u] ← ∞;
    d[r] ← 0;
  ❷ while (Q ≠ ∅) {                         ▷ n회 순환된다.
      ❸ u ← deleteMin(Q, d);
      ❹ for each v ∈ L(u)                   ▷ L(u): u의 인접 정점 집합
          ❺ if (v ∈ Q and w(u, v) < d[v]) then {
              ❻ d[v] ← w(u, v);
                tree[v] ← u;
            }
    }
}

deleteMin(Q, d[ ])
{
  집합 Q에서 d값이 가장 작은 정점 u를 리턴하고, u를 집합 Q에서 제거한다;
}
```

첫 번째 프림 알고리즘에서 집합 S가 공집합에서 시작하여 모든 정점을 포함할 때까지(즉, $S=V$가 될 때까지) 커져가는 과정은, 두 번째 프림 알고리즘에서 집합 Q가 모든 정점의 집합 V로부터 시작하여 공집합으로 줄어가는 것과 일치한다.

프림 알고리즘의 수행시간을 분석해보자. 우선 ❶의 for 루프는 정확히 n회 반복된다($n=|V|$, 혼동의 염려가 없으면 편의상 $|V|$ 대신에 V를 사용하기도 한다). 각각의 for 루프에서 단순 할당 작업을 하므로 상수 시간이 소모되어 ❶의 for 루프는 총 $O(V)$ 시간을 소요한다. 결과적으로 보면 프림 알고리즘이 $O(V)$보다 시간이 더 소요되므로 이 for 루프는 점근적 시간에 영향을 미치지 않는다. ❷의 while 루프는 정확히 n회 반복된다. 이 while 루프에 ❹의 for 루프가 중첩되어 있다. 여기서 좀 주의를 기울여야 한다. 두 개의 루프가 중첩되어 있지만 ❹의 for 루프는 u와 인접한 간선을 훑어보는데 어떻게 되든 한 간선은 두 번만 본다(임의의 간선 (u, v)는 알고리즘을 통틀어 ❺에서 단 두 번만 사용된다). 그러므로 ❹의 for 루프는 while 루프를 통틀어 $2|E|$번 돌 뿐이다. 프림 알고리즘의 수행시간은 ❸의 deleteMin()이 ❷의 while 루프가 반복될 때마다 수행되는 시간과 ❺, ❻이 ❹의 for 루프 내에서 수행되는 시간 중 하나가 좌우하게 된다. ❻에서의 d값 변경은 단순한 배열 원소값 변경으로는 힘든데 그 이유는 뒤이어 설명한다.

deleteMin()은 d값이 가장 작은 정점을 고른 다음 제거하는 것으로, 어떠한 자료구조를 쓰느냐에 따라 시간이 달라진다. 가장 원시적인 배열을 사용한다고 해보자. 즉, $d[]$를 배열로 볼 수 있다. 배열을 정렬된 배열로 유지한다면 가장 작은 원소를 추출하는 데는 상수 시간이 든다. 단, ❻에서 $d[v]$ 값의 변동이 생길 때 이를 수정하기 위해서는 배열을 한 칸씩 줄줄이 이동할 필요가 생기므로 최대 $O(V)$ 시간이 소요된다. 이러한 수정은 최악의 경우 $|E|$번 일어날 수 있으므로 이것이 수행시간을 좌우하여 전체 시간은 $O(VE)$가 된다. 만약 배열을 정렬되지 않은 형태로 유지한다면 이번에는 가장 작은 원소를 추출하는 데 $O(V)$ 시간이 소요된다. 대신 $d[v]$ 값의 변동은 상수 시간에 해결된다. 따라서 이 경우에는 ❸의 deleteMin()에 소요되는 시간이 좌우하여 $O(V^2)$이 된다. 이 정도의 시간은 아무래도 좀 불만스럽다.

d값의 관리를 위해 최소힙을 사용하면 시간을 개선할 수 있다. d값들이 힙으로 구성되어 있다면 ❸의 deleteMin()은 $O(\log V)$ 시간에 가능하다. while 루프가 n회 반복되므로 이와 관계된 비용은 $O(V\log V)$이다. 그런데 ❻에서 d값의 변동이 생기면 힙을 조정해주어야 한다. 힙에서 임의의 원소값이 변해서 이를 반영하여 조정하는 데는 $O(\log V)$ 시간이 소요된다. 이러한 수정은 ❻에서 최악의 경우 $|E|$번 일어날 수 있으므로 이와 관련된 비

용은 총 $O(E\log V)$가 된다. 힙정렬에서 배웠듯이, 최초에 힙을 만드는 데는 $O(V)$이면 충분하므로 프림 알고리즘의 수행시간은 $O(E\log V)$가 된다.

2. 크루스칼 알고리즘

크루스칼(Kruskal) 알고리즘은 싸이클을 만들지 않는 범위에서 최소 비용 간선을 하나씩 더해가면서 최소신장트리를 만든다. n개의 정점으로 트리를 만드는 데는 $n-1$개의 간선이 필요하므로, 알고리즘은 최초에는 간선이 하나도 없는 상태에서 시작하여 $n-1$개의 간선을 더하면 끝난다. 크루스칼 알고리즘은 프림 알고리즘처럼 하나의 트리를 키워나가는 방식이 아니고, 임의의 시점에 최소 비용의 간선을 더하므로 여러 개의 트리가 산재하게 된다. 최초에는 n개의 트리로 시작한다. 즉, 간선이 하나도 없는 상태의 n개의 정점은 각각 정점 하나만으로 이루어진 n개의 트리로 볼 수 있다. 하나의 간선을 더할 때마다 두 개의 트리가 하나의 트리로 합쳐진다. $n-1$개의 간선을 더하고 나면 모든 트리가 합쳐져서 하나의 트리가 된다. [알고리즘 9-5]에 크루스칼 알고리즘을 기술하였다.

[알고리즘 9-5] 크루스칼 알고리즘

```
Kruskal(G)
{
    T ← ∅;                          ▷ T: 신장트리
  ❶ 단 하나의 정점만으로 이루어진 n개의 집합을 초기화한다;
  ❷ 모든 간선을 가중치의 크기순으로 정렬하여 배열 A[1...E]에 저장한다;
  ❸ while (T의 간선수 < n-1) {
      ❹ A에서 최소비용의 간선 (u, v)를 제거한다;
      ❺ if (정점 u와 v가 다른 집합에 속함) then {
          ❻ T ← T ∪ {(u, v)};
          ❼ 정점 u와 v가 속한 두 집합을 하나로 합친다;
        }
    }
}
```

[그림 9-15]는 크루스칼 알고리즘의 작동 예다. 음영으로 표시된 영역 하나가 하나의 집합에 해당한다. 그림 (a)에서 각각의 정점만으로 이루어진 n개의 집합을 볼 수 있다. 이것부터 시작해 한 번에 하나씩의 간선을 통해 집합을 합쳐나간다. 궁극적으로 $n-1$번 합치면 (h)처럼 하나의 집합이 된다. 이제 그림을 설명해보자.

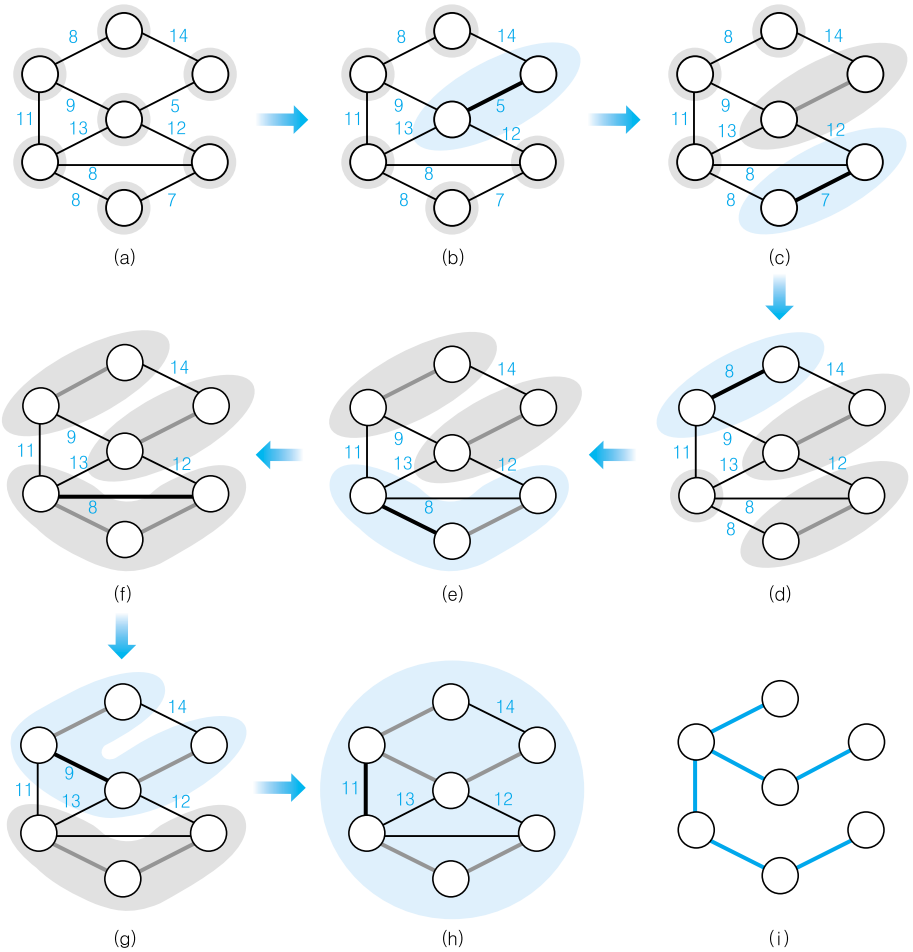
- **그림 (b)**: 그림 (a)에서 가장 작은 간선은 5이다. 이 간선을 택하고 이로 인해 각각 원소 하나씩을 가진 두 집합이 합쳐서 두 개의 원소를 가진 집합이 되었다. 그림에서 별색의 음영으로 표시된 집합이다. 앞으로 이 간선은 다시는 보지 않는다. 알고리즘에서 사실상 제거한다. 나머지 간선 중 가장 작은 것은 가중치가 7이다.
- **그림 (c)**: 앞의 가중치 7인 간선으로 두 집합이 하나로 합쳐진다. 이 간선은 알고리즘에서 제거한다. 이제 남은 간선 중 가장 작은 것은 가중치가 8인데 3개가 있다. 이 중 하나를 임의로 선택한다.
- **그림 (d)**: 앞의 가중치 8인 간선으로 두 집합이 하나로 합쳐진다. 이제 남은 간선 중 가장 작은 것은 가중치가 8인데 2개가 있다. 이 중 하나를 임의로 선택한다.
- **그림 (e)**: 다시 두 집합이 하나로 합쳐진다. 이제 남은 간선 중 가장 작은 것은 가중치가 8이다.
- **그림 (f)**: 이 간선의 경우 양 끝점이 같은 집합에 속한다. 즉, 이 간선을 집어넣으면 기존의 집합에서 싸이클이 만들어진다. 신장트리를 만들어가는 과정이므로 싸이클이 만들어지면 안된다. 따라서 이 간선은 그냥 버린다.
- **그림 (g)**: 다음으로 가중치 9를 가진 간선이 더해진다.
- **그림 (h)**: 마지막으로 가중치 11을 가진 간선이 더해지면서 모든 정점이 하나의 집합으로 통합된다.

최종적인 최소신장트리는 (i)와 같다. (h)에서 볼 수 있듯이 가중치 12, 13, 14를 가진 세 개의 간선은 살펴보지 않고 알고리즘이 끝났다. 크루스칼 알고리즘이 간선을 차례로 보지만 작은 순서로 보므로 이렇게 상당한 수의 간선을 보지 않은 채 끝날 수 있다.

그림의 표기를 다시 한 번 정리하면, 음영으로 표시된 영역은 하나씩의 집합을 나타낸다. 이 중 별색 음영으로 표기된 것은 하나의 간선을 통해 방금 하나로 합쳐진 집합이다. 검은색으로 굵게 표시된 간선은 방금 취급한 간선이다. 이로 인해 두 집합이 하나로 합쳐질 수도 있고 (b, c, d, e, g, h), 싸이클을 만들어 그냥 버려질 수도 있다(f). 회색으로 굵게 표시된 간선들은 싸이클을 만들지 않고 성공적으로 더해진 간선들이다. 이 간선들이 궁극적으

로 최소신장트리를 만든다. 그림에서는 이미 한 번 취급되어 더이상 고려하지 않는 간선들은 가중치 값을 보이지 않게 하고, 방금 취급된 간선과 앞으로 고려할 간선들의 가중치만 남겼다.

그림의 각 집합은 하나씩의 부분 트리에 해당한다. 따라서 두 집합이 하나로 합쳐지는 것은 두 부분 트리가 하나로 합쳐지는 것이다. 프림 알고리즘이 하나의 트리(=집합)에 초점을 맞추어 키워나가는 데 비해 크루스칼 알고리즘은 부분적 트리(=집합)들을 조각조각 만들고 합쳐간다.



[그림 9-15] 크루스칼 알고리즘의 작동 예

[알고리즘 9-5]의 크루스칼 알고리즘의 수행시간을 분석해보자. 우선 ❷에서 모든 간선을 정렬하는 데 $O(E\log E)=O(E\log V)$가 소요된다. ❸의 while 루프는 최소 $|V|-1$회, 최대 $|E|$회 반복된다. while 루프 안을 보자. ❹는 정렬된 배열에서 맨 앞부터 하나씩 제거해나가면 되므로 상수 시간이 든다. ❻은 간선 하나를 단순히 더하는 것이라 상수 시간이 든다. ❺와 ❼은 집합을 관리하는 작업들(두 정점이 다른 집합에 속하는지 확인하는 작업과 두 집합을 하나로 합치는 작업)로 여러 가지 구현 방법이 있다. 7장에서 배운 집합의 처리 방법을 이용하면 최대 $O(E)$번의 Find-Set()과 Union()을 수행하게 된다. 여기에 ❶에서 $|V|$번의 Make-Set()이 필요하다. 랭크를 이용한 Union과 경로압축을 이용한 Find-Set을 사용하면 이들을 모두 합친 시간은 $O((V+E)\log^* V)$가 된다. $\log^* V$는 거의 상수로 간주할 수 있으므로 결국 크루스칼 알고리즘의 시간을 좌우하는 것은 우습게도 ❷의 정렬 시간이다. 따라서 총 수행시간은 $O(E\log V)$가 된다.

❸ 안전성 정리

앞서 소개한 두 알고리즘의 작동 방법만 알고자 하는 독자는 이 부분을 읽지 않아도 좋다. 하지만 소개한 두 알고리즘이 옳음을 확신할 수 있는 이유를 알고 싶으면 읽어보기 바란다. 여기서는 두 알고리즘의 이론적 정당성을 제공한다.

앞서 소개한 두 알고리즘은 사실 한 가지 원리를 바탕으로 한다. 프림 알고리즘은 하나의 집합 S를 키워나가면서 하나의 신장트리를 만드는 방식이고, 크루스칼 알고리즘은 여러 집합들을 합쳐가면서 최종적으로 하나의 신장트리를 만드는 방식이다. 프림 알고리즘은 집합 S에 정점을 하나씩 더해 나가지만 정점을 하나 더할 때마다 해당 정점을 트리에 연결하는 간선이 정해지므로 사실상 간선을 하나씩 더하는 것이다. 크루스칼 알고리즘은 두 개의 집합을 합칠 때마다 두 집합을 연결하는 하나의 간선이 정해지므로 역시 간선을 하나씩 더하는 것이다. 따라서 두 알고리즘 모두 $n-1$개의 간선을 더해가는 순서를 나름대로 정해준다. 여기서 다루는 정리는 이렇게 더해나가는 간선이 궁극적으로 최소신장트리에 이르는 길을 놓치지 않아 해당 간선을 일단 최소신장트리에 포함시키는 것이 안전하다는 것이다.

간선이 가중치를 갖는 무방향 그래프 $G=(V, E)$의 정점들이 임의의 두 집합 S와 $V-S$로 나누어져 있다고 하자. 간선의 두 끝점이 한쪽은 S에, 한쪽은 $V-S$에 속한다면 이 간선은 S와 $V-S$ 사이의 **교차간선**이라 한다. S와 $V-S$사이의 교차간선들 중 가장 가벼운 간선(들)을 **최소 교차간선**이라 하자.

정리 01 안전성 정리

간선이 가중치를 갖는 무방향 그래프 $G=(V, E)$의 정점들이 임의의 두 집합 S와 $V-S$로 나누어져 있다고 하자. 간선 (u, v)가 S와 $V-S$사이의 최소 교차간선이면 (u, v)를 포함하는 그래프 G의 최소신장트리가 반드시 존재한다.

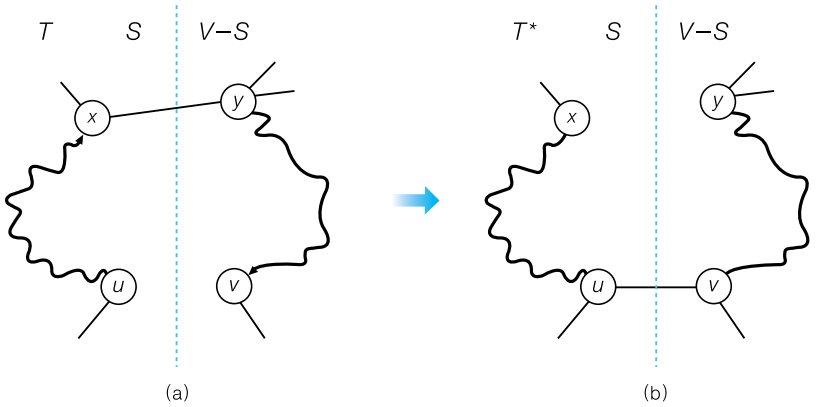
[증명] 임의의 최소신장트리 T가 간선 (u, v)를 포함하고 있지 않다고 하자. 트리 T는 연결 그래프이므로 정점 u에서 정점 v에 이르는 경로가 반드시 존재하고 이것은 u에서 정점 v에 이르는 유일한 경로다. 만약 경로가 2개 이상이라면 u와 v를 포함하는 싸이클이 만들어져 트리가 될 수 없다. 이 경로는 S에서 시작해서 $V-S$에 이르는(또는 그 반대) 경로이므로 이 경로상에는 S와 $V-S$사이의 교차간선이 적어도 하나 존재한다. 이런 교차간선 하나를 (x, y)라 하자. [그림 9-16](a)에 이러한 상황을 그려 놓았다. 그림에서 모든 실선은(곡선 포함) 최소신장트리 T를 구성하는 간선을 나타낸다. 화살표는 경로임을 표시하기 위해 사용했을 뿐이고 실제로는 일련의 간선들을 나타낸다. 그림에 집합 S와 집합 $V-S$의 교차간선은 (x, y) 이외에도 복수 개 있을 수 있다. (x, y)는 그 중 임의로 하나를 잡은 것이다.

트리 T에 간선 (u, v)를 더하면 싸이클이 만들어진다. 이 싸이클은 앞의 $u \rightsquigarrow v$ 경로에 간선 (u, v)가 더해진 싸이클이다. 트리 T에서 (x, y)를 제거하고 (u, v)를 포함시키면 다른 신장 트리 T^*가 만들어진다([그림 9-16](b)). (u, v)가 S와 $V-S$ 사이의 최소 교차간선이므로 $w(u, v) \leq w(x, y)$이다. 그러므로 (x, y)를 제거하고 (u, v)를 새로 넣은 신장트리 T^*는 $w(T^*) \leq w(T)$를 만족한다(이때 $w(u, v) < w(x, y)$인 경우는 있을 수가 없다. 왜냐하면 T가 최소신장트리라고 하였으므로 $w(u, v) < w(x, y)$라면 $w(T^*) < w(T)$가 되어 T가 최소신장트리라는 조건에 모순이기 때문이다. $w(u, v) = w(x, y)$라면 $w(T^*) = w(T)$가 되어 T^*도 최소신장트리다).

즉, (u, v)를 포함하고 있지 않은 최소신장트리가 존재한다면 반드시 이를 포함하는 최소신장트리로 바꿀 수 있다. ■

프림 알고리즘에서 이미 구성된 집합 S에서의 최소신장트리에 $V-S$ 사이를 연결하는 최소 간선을 더하는 것은 **안전하다**고 말할 수 있다. 크루스칼 알고리즘에서 임의의 간선 (u, v)를 더하는 시점에 (u, v)는 그 때까지 고려되지 않은 모든 간선들 중 최소 간선이다. 정점 u가 속한 집합을 S라 하면, 크루스칼 알고리즘의 작동 원리상 간선 (u, v)는 집합 S와 나머지 집합 $V-S$의 최소 교차간선이다. 그러므로 간선 (u, v)를 더하는 것은 **안전하다**. 프

림 알고리즘과 크루스칼 알고리즘은 이런 논리적 기반 하에서 안전한 간선을 차례로 더해 가는 것이다.



[그림 9-16] 최소 교차간선으로 최소신장트리를 바꾼 상황

Section

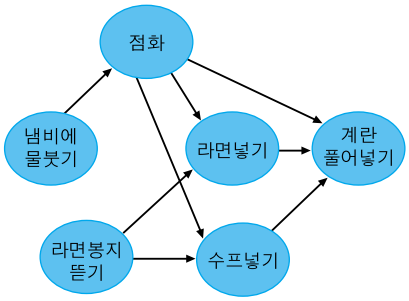
05 위상정렬

완수해야 할 여러 가지 일이 있고 이들간에 상호 선후관계가 있다. 한 사람이 라면을 끓이려고 할 때 어떤 순서로 일을 해야 하는가? [그림 9-17]은 라면을 맛있게 끓이기 위해 필요한 작업의 선후관계를 나타낸 것이다. 두 작업간에 간선이 있으면 이들의 순서는 반드시 유지되어야 작업이 제대로 된다. 예를 들면, 라면을 넣는 것은 점화를 한 후여야 한다. 계란을 넣는 것은 라면을 넣은 다음이어야 한다. 계란을 넣는 것은 수프를 넣은 다음이어야 한다. 이런 식이다. 한 사람이 한 번에 한 가지 작업밖에 할 수 없으므로 위 선후관계에 기초해서 아래의 순서로 라면 끓이기를 할 수 있다.

① 냄비에 물붓기
② 점화
③ 라면봉지 뜯기
④ 라면넣기
⑤ 수프넣기
⑥ 계란 풀어넣기

또한 냄비에 물붓기와 라면봉지 뜯기는 어느 것을 먼저 해도 상관없으므로, 다음과 같이 해도 된다.

① 라면봉지 뜯기
② 냄비에 물붓기
③ 점화
④ 라면넣기
⑤ 수프넣기
⑥ 계란 풀어넣기



[그림 9-17] 라면 끓이기 작업에서의 선후관계

라면넣기와 수프넣기도 선후관계가 없으므로 다음과 같은 순서도 무방하다.

① 라면봉지 뜯기
② 냄비에 물붓기
③ 점화
④ 수프넣기
⑤ 라면넣기
⑥ 계란 풀어넣기

위의 세 가지 순서에 공통적인 것은 [그림 9-17]에서 작업 i와 작업 j사이에 간선 (i, j)가 존재한다면 작업 i는 반드시 작업 j보다 먼저 수행된다는 사실이다. 그림의 모든 간선에 대해서 이 성질만 만족하면 어떤 순서라도 좋다. 이러한 성질을 만족하는 정렬을 **위상정렬**(Topological Sorting)이라고 한다.

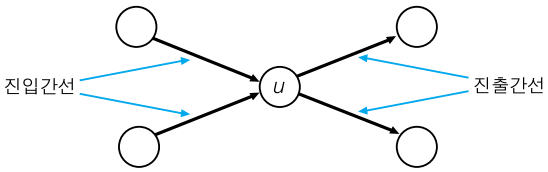
위상정렬은 싸이클이 없는 유향 그래프 $G=(V, E)$에서 V의 모든 정점을 정렬하되 다음 성질을 만족해야 한다.

- 간선 (i, j)가 존재하면 정렬 결과에서 정점 i는 반드시 정점 j보다 앞에 나열되어야 한다.

만약 그래프에 싸이클이 있다면 위의 성질은 결코 만족될 수 없으므로 위상정렬을 할 수 없다. [알고리즘 9-6]은 위상정렬 알고리즘을 기술한 것이다. 알고리즘에서 정점 u의 **진입간선**이란 [그림 9-18]과 같이 u에 연결된 간선들 중 u로 향하는 간선을 말하고, **진출간선**이란 u에 연결된 간선들 중 u로부터 나가는 간선을 말한다.

[알고리즘 9-6] 위상정렬 알고리즘 1

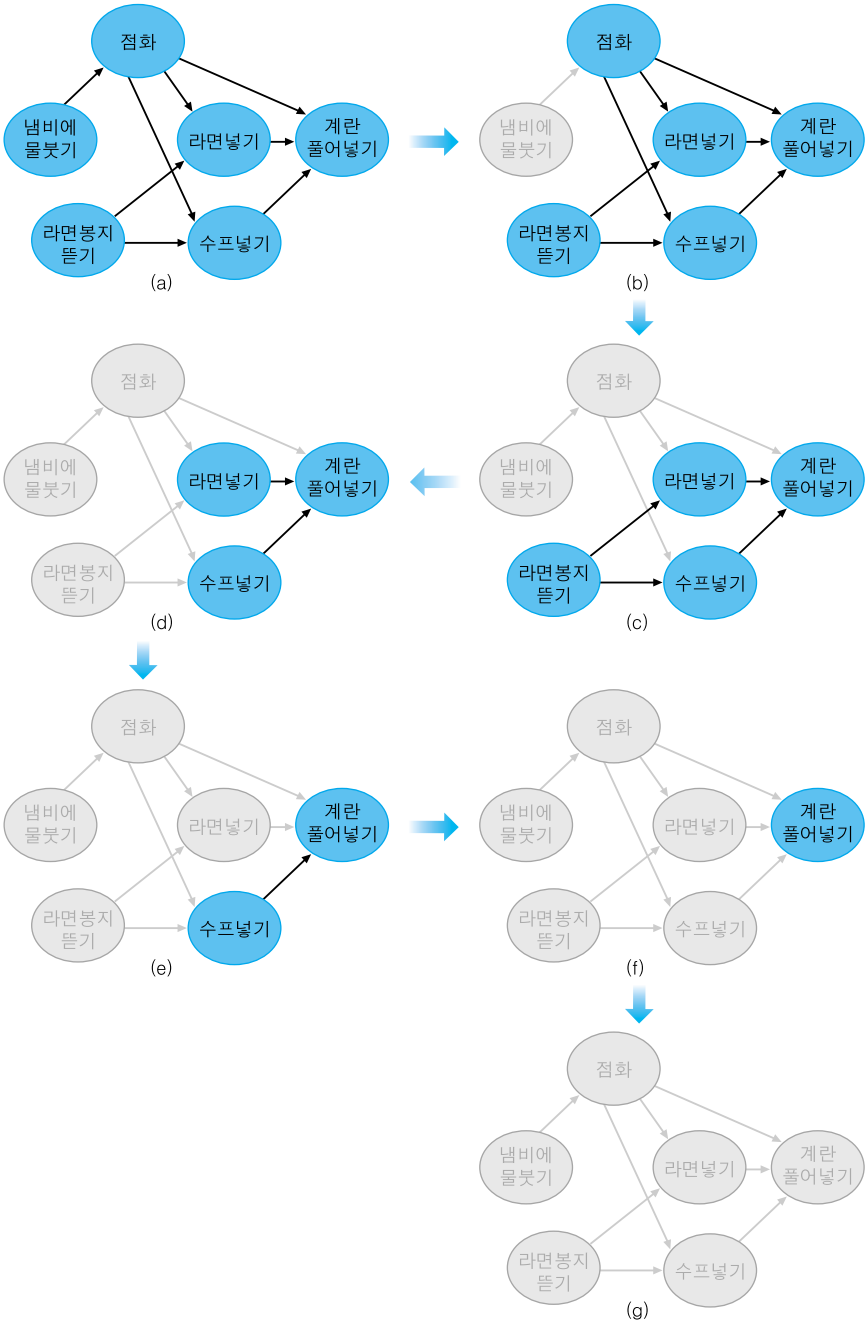
```
topologicalSort1(G, v)
{
    for i ← 1 to n {
        ❶ 진입간선이 없는 정점 u를 선택한다;
          A[i] ← u;
        ❷ 정점 u와, u의 진출간선들을 모두 제거한다;
    }
    ▷ 이 시점에 배열 A[1...n]에는 정점들이 위상정렬되어 있다.
}
```



[그림 9-18] 정점 u의 진입간선과 진출간선

[그림 9-19]는 [그림 9-17]의 그래프를 입력으로 위상정렬 알고리즘을 수행하는 예다. 과정을 차례로 설명해보자.

- **그림 (a)**: 진입간선이 없는 정점은 "냄비에 물붓기"와 "라면봉지 뜯기"이다.
- **그림 (b)**: 이 중 임의로 "냄비에 물붓기"를 택하고 이 정점과 진출간선을 모두 제거한다. 이제는 "점화"와 "라면봉지 뜯기"가 진입간선이 없다.
- **그림 (c)**: 이 중 "점화"를 택하고 이 정점과 진출간선을 모두 제거한다. 이제는 진입간선이 없는 정점은 "라면봉지 뜯기"뿐이다.
- **그림 (d)**: "라면봉지 뜯기"를 선택하고 이 정점과 진출간선을 모두 제거한다. 이제 진입간선이 없는 정점은 "라면넣기"와 "수프넣기"이다.
- **그림 (e)**: 두 정점 중 "라면넣기"를 선택하고 이 정점과 진출간선을 모두 제거한다. 이제 진입간선이 없는 정점은 "수프넣기"뿐이다.
- **그림 (f)**: "수프넣기"를 선택하고 이 정점과 진출간선을 제거한다. 이제 정점이 하나만 남았다. 당연히 진입간선은 없다.
- **그림 (g)**: 이 정점을 제거한다.



[그림 9-19] 위상정렬 알고리즘 1의 작동 예

여기서 제거된 정점의 순서가 하나의 위상정렬을 이루고, 이것은 이 절의 시작 부분에서 예를 든 세 가지 위상정렬 중 첫 번째에 해당한다.

위상정렬 알고리즘의 수행시간을 분석해보자. for 루프는 n번 반복된다. 매 반복시마다 한 개의 정점이 선택되고 해당 정점에 연결된 진출간선이 모두 제거된다. 그러므로 각 간선은 단 한 번씩만 취급된다. 그러므로 총 수행시간은 $\Theta(V+E)$이다. 이를 달성하기 위해서는 자료구조와 관련한 약간의 기술이 필요한데 자세한 것은 부연설명을 참고하기 바란다.

[알고리즘 9-7]에 위상정렬 알고리즘을 하나 더 소개한다. 이것은 DFS를 이용한 것으로 앞의 알고리즘보다 더 많이 사용된다. 앞의 첫 번째 위상정렬 알고리즘은 좀더 직관적인 접근법을 택한 것이고, 두 번째는 DFS의 성질을 적절히 이용한 고급스런 알고리즘이라 할 수 있다. DFS를 거의 그대로 이용하면서 약간의 요소만 더 추가되어 두 번째 알고리즘이 더 구현하기 간단하다. DFS와 다른 점은 함수의 맨 끝에(❶) 작업이 끝난 정점을 연결 리스트로 관리하는 부분이다. 알고리즘이 끝나고 나서 연결 리스트 R에 매달린 순서가 위상정렬된 순서다.

위상정렬 알고리즘의 수행시간 분석

앞에서 설명한 위상정렬을 위한 [알고리즘 9-6]의 수행시간 $\Theta(V+E)$는 진입간선이 없는 정점을 찾는 것이 상수 시간에 이루어진다고 해야 가능하다. 이것은 자명하지가 않다.

우선 그래프의 표현을 인접행렬로 하면 행렬의 준비 과정에서 $\Theta(V^2)$시간이 소요되므로 대부분의 그래프에 대해 $\Theta(V+E)$를 넘어버려 인접리스트를 사용하기로 한다. 일반적으로 유향 그래프의 인접리스트에서 간선 (u, v)는 정점 u의 연결 리스트에만 포함시킨다. 즉, 인접리스트에서 간선 (u, v)는 정점 u의 진출간선으로만 관리된다. 이 경우 진입간선이 없는 정점을 찾기 위해서는 모든 정점의 진출간선 리스트를 다 보아야 한다. 이것 하나를 찾는데 최악의 경우 $\Theta(V+E)$ 시간이 들어 앞의 시간을 맞출 수 없다. 이 문제는 다음의 방법으로 해결할 수 있다.

알고리즘을 시작할 때 배열 $C[1...n]$에 각 정점의 진입간선 수를 기록한다. 인접리스트를 한 번 훑으면서 간선 (u, v)를 만나면 $C[v]$를 1 증가시키면 된다. 모두 훑고 나면 $C[1...n]$에는 각 정점의 진입간선 수가 기록된다. $C[1...n]$를 한 번 훑어 0의 값을 가진 정점들을 리스트로 만들어둔다. ❷에서 정점 u의 진출간선 (u, v)를 지울 때 $C[v]$를 1 감소시킨다. 그러면 $C[1...n]$은 항상 현재의 진입간선 수를 갖고 있게 된다. 이 과정에서 정점 v의 $C[v]$ 값이 0에 이르면 앞의 진입간선 수가 0인 정점 리스트에 삽입한다. 이렇게 관리하면서 ❶에서 진입간선이 없는 정점을 필요로 할 때마다 이 리스트에서 하나씩 빼주면 된다.

따라서 진입간선이 없는 정점을 찾는 것은 상수 시간에 이루어지고, 이것을 가능하게 하기 위한 앞의 모든 준비 과정은 $\Theta(V+E)$의 시간이 소요된다.

[알고리즘 9-7] 위상정렬 알고리즘 2

```
topologicalSort2(G)
{
    for each v∈V
        visited[v] = NO;
    for each v∈V            ▷ 정점의 순서는 무관
        if (visited[v] = NO) then DFS-TS(v);
}
DFS-TS(v)
{
    visited[v] = YES;
    for each x∈L(v)
        if (visited[v] = NO) then DFS-TS(x);
    ❶ 연결 리스트 R의 맨 앞에 정점 v를 삽입한다;
}
```

이 알고리즘의 수행시간은 DFS에 상수 시간이면 충분한 ❶의 연결 리스트 삽입을 더한 것뿐이므로 DFS의 수행시간과 같다. 그러므로 $\Theta(V+E)$이다.

[그림 9-20]은 앞의 라면 끓이기 작업을 위상정렬 알고리즘 2를 이용해서 수행해본 예다.

- **그림 (a)**: 주어진 그래프다. 여기서 첫 번째 DFS-TS를 위해 아무 정점이나 택한다.
- **그림 (b)**: "수프넣기"를 시작 정점으로 하여 DFS-TS를 시작하였다.
- **그림 (c)**: "계란 풀어넣기"를 방문한다. 이 정점에서 더 할 일이 없다. 즉, 완료되었다. 연결 리스트에 "계란 풀어넣기"를 삽입하고 첫 번째 완료된 정점이란 의미로 ❶을 표시하였다. 알고리즘에서 ❶의 삽입이 처음으로 수행되었다는 의미다.
- **그림 (d)**: "계란 풀어넣기" 정점이 완료되었으므로 "수프넣기" 정점으로 되돌아온다(Backtracking). "수프넣기" 정점에서도 더 할 일이 없으므로 연결 리스트에 삽입하고 완료된 정점이란 의미로 ❷를 표시하였다. 그림 (b)에서 시작한 "수프넣기"를 시작 정점으로 한 DFS-TS가 끝났다.
- **그림 (e)**: 방문하지 않은 정점 중 하나를 임의로 선택하여 DFS-TS를 다시 시작한다. "냄비에 물붓기" 정점이 시작정점으로 선택되었다.

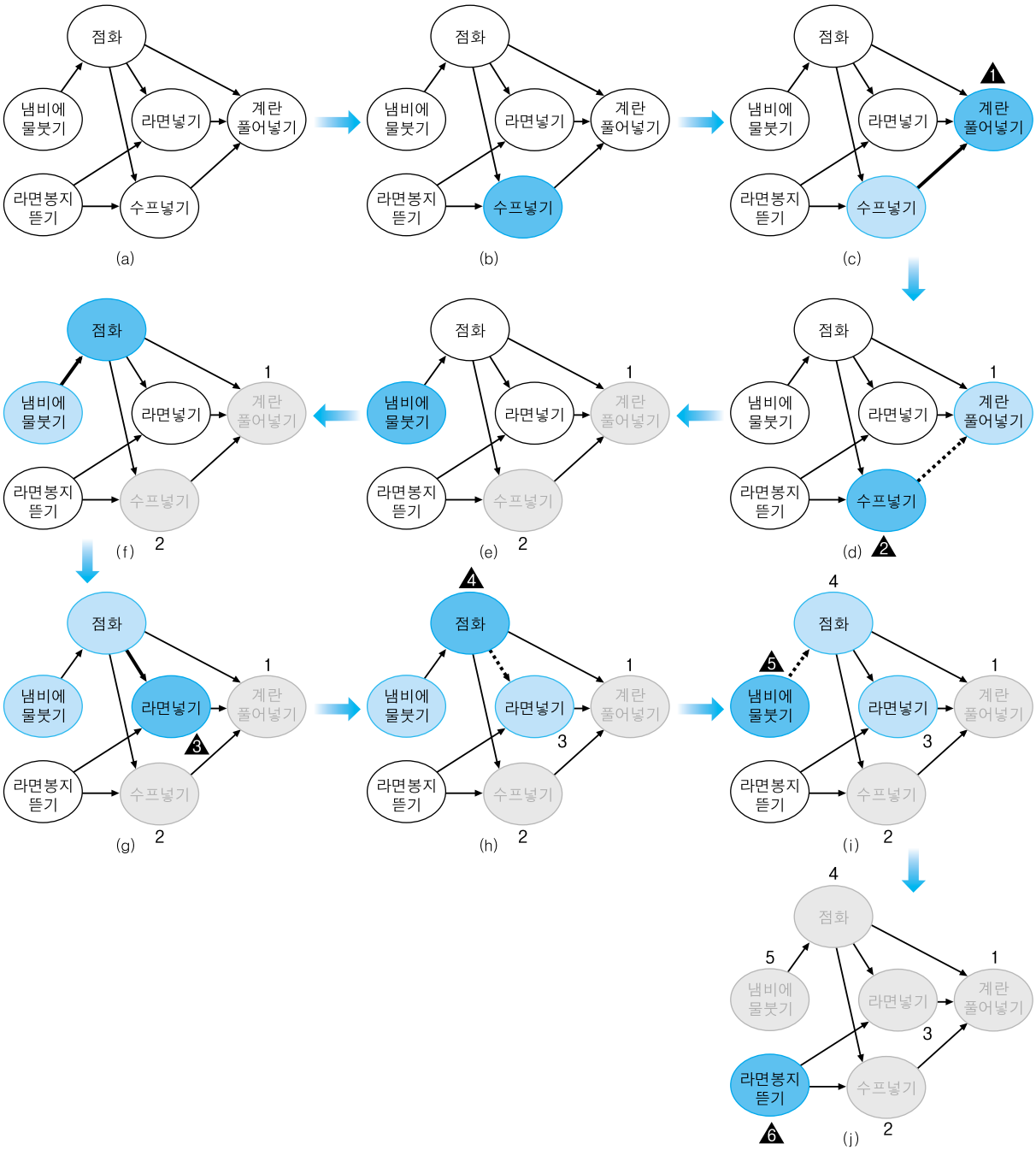
- **그림 (f)**: "점화" 정점을 방문한다. "점화" 정점의 진출간선이 방문하지 않은 "라면넣기" 정점으로 연결되어 있으므로 "점화" 정점은 아직 완료되지 않았다.
- **그림 (g)**: "라면넣기" 정점을 방문한다. "라면넣기" 정점에서는 더 할 일이 없으므로 연결 리스트에 삽입하고 완료된 정점이란 의미로 ❸을 표시하였다.
- **그림 (h)**: "점화" 정점으로 되돌아온다. "점화" 정점에서도 더 할 일이 없으므로 연결 리스트에 삽입하고 완료된 정점이란 의미로 ❹를 표시하였다.
- **그림 (i)**: "냄비에 물붓기" 정점으로 되돌아온다. 이 정점에서도 더 할 일이 없으므로 연결 리스트에 삽입하고 완료된 정점이란 의미로 ❺를 표시하였다. 그림 (e)에서 시작한 "냄비에 물붓기" 정점을 시작으로 한 DFS-TS가 끝났다.
- **그림 (j)**: 남은 정점이 "라면봉지 뜯기" 하나밖에 없으므로 이를 시작으로 DFS-TS를 수행하여 바로 완료된다. 연결 리스트에 삽입하고 완료 순서 ❻을 표시하였다.

그림에서 검정색 실선으로 굵게 표시한 간선은 관련 정점을 처음으로 방문할 때 사용된 것이고(DFS의 전진), 검정색 점선으로 굵게 표시한 간선은 화살표의 반대 방향으로 백트래킹할 때 사용된 것이다(DFS의 후진). 별색으로 표시된 정점들은 현재의 DFS-TS에서 방문한 정점들이다. 이 중 짙은 별색으로 표시한 정점이 해당 단계에서 관심의 대상인 정점이다. 옅은 회색으로 표시한 정점들은 이미 처리가 끝난 정점들이다.

최종적으로 연결 리스트 R에는 그림에서 표시된 완료순서의 역순으로 정점들이 저장되어 있다. 이 순서가 위상정렬 순서이다. 즉, 다음과 같은 순서가 된다.

① 라면봉지 뜯기
② 냄비에 물붓기
③ 점화
④ 라면넣기
⑤ 수프넣기
⑥ 계란 풀어넣기

이것은 이 절의 시작 부분에서 예를 든 세 가지 위상정렬 중 두 번째와 일치한다.



[그림 9-20] 위상정렬 알고리즘 2의 작동 과정을 보여주는 예