```
        for (i=0; i<n; i++)
            marked[i] = false;
        for (i=0; i<n; i++)
            if (marked[i]!= true) {
                DFS(i);
                System.out.println ( "\n" );
            }
    }
```

메소드 CONCOMP에 의하여 〈그림 8.5〉의 그래프 G_4에 적용하여 실행하면

```
component 1 : (1, 2, 3, 4, 5)
component 2 : (6, 7)
```

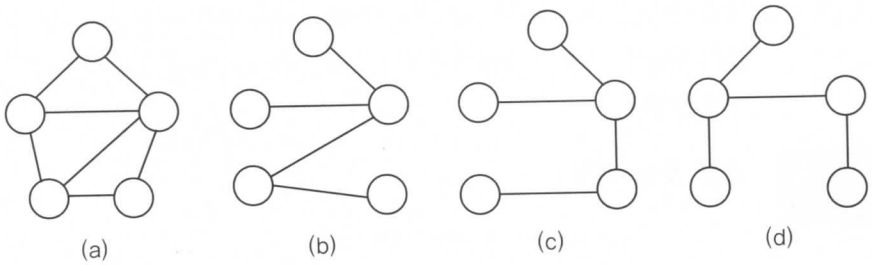
이라는 2개의 연결 요소가 결정된다.

8.4 그래프의 트리화

8.4.1 신장 트리

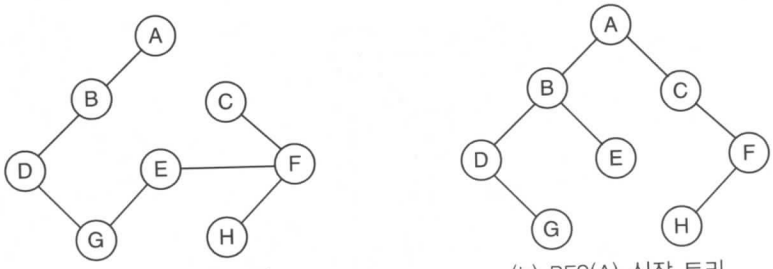
연결 그래프 $G=(V, E)$가 있을 때, 이것을 깊이 우선 검색이나 너비 우선 검색을 하면 집합 V에 있는 모든 정점을 방문하게 된다. 그러나 간선 집합 E는 검색 중에 운행된 간선과 그렇지 않은 간선으로 나눌 수 있다. 이 때, 검색 중에 운행된 간선의 집합을 E_1이라 하고, 운행되지 않은 간선의 집합을 E_2라 하면 새로운 집합 $T=(V, E_1)$을 구할 수 있는데, 집합 T는 G의 모든 정점과 검색 중에 운행된 간선만으로 이루어지는 트리를 형성한다. 이러한 트리를 신장 트리(spanning tree)라고 한다.

8.4.2



〈그림 8.14〉 그래프 (a)에 대한 3개의 신장 트리

하나의 연결 그래프는 검색 방법이나 검색시 출발하는 정점에 따라 각기 다른 신장 트리가 만들어지는데 〈그림 8.14〉는 (a)의 그래프에서 만들어질 수 있는 몇 가지 신장 트리를 (b), (c), (d)로 나타낸 것이다.



(a) DFS(A) 신장 트리 (b) BFS(A) 신장 트리

〈그림 8.15〉 〈그림 8.11〉의 그래프에 대한 신장 트리

신장 트리는 메소드 DFS나 BFS를 호출하여 만들 수 있는데, DFS를 호출하여 만든 신장 트리를 깊이 우선 신장 트리(depth first spanning tree)라 하고, BFS를 사용하여 만든 신장 트리를 너비 우선 신장 트리(breadth first spanning tree)라고 한다. 〈그림 8.15〉는 〈그림 8.11〉의 그래프에서 정점 A에서 검색을 시작했을 때 생성된 두 가지의 신장 트리이다.

신장 트리는 전기 회로망의 분석이나 또는 통신망 및 도로망의 운영 등 여러 분야에서 이용된다.

8.4.2 최소 비용 신장 트리

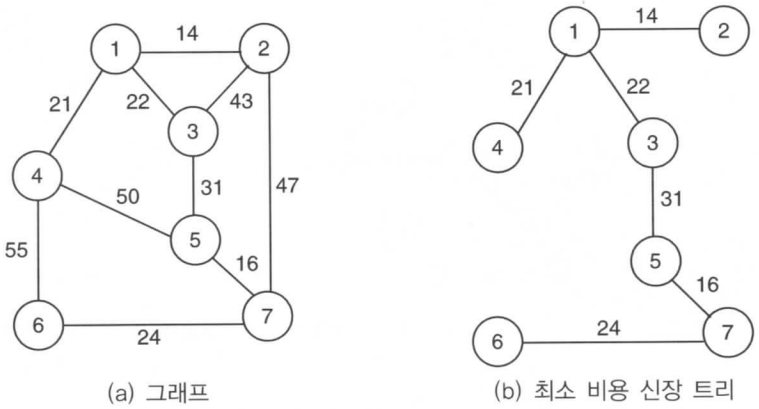
하나의 연결 요소로 구성되어 있는 그래프 G가 n개의 정점을 가질 경우 이것의 신장 트리는 $(n-1)$개의 간선을 갖는다.

응용 분야의 하나로 그래프의 각 간선에 가중값(weight)이 주어져 있는 경우를 생각해 보자. 여기에서 가중값은 통신망에 있어서 회선간의 비용이나 거리 등이 될 수 있다. 따라서, 그래프의 각 간선에 가중값이 부여되어 있다면 통신 비용이 가장 적게 들거나 거리가 가장 짧은 신장 트리를 생성하는 문제가 관심이 된다. 이 때 가장 적은 비용으로 생성된 트리를 최소 비용 신장 트리(minimum cost spanning tree)라고 한다.

최소 비용 신장 트리를 구하는 알고리즘에는 Prim의 알고리즘과 Kruskal의 알고리즘이 있는데, 먼저 Prim의 알고리즘(Prim's algorithm)에 대하여 살펴본다.

주어진 그래프 G에 대하여 최소 비용 신장 트리는 항상 유일하게 존재하는 것은 아니다.

〈그림 8.16〉은 그래프 G가 (a)와 같이 주어졌을 때 만들어진 최소 비용 신장 트리를 (b)로 보인 것이다. 〈그림 8.16〉을 예로 하여 Prim의 방법으로 어떻게 최소 비용 신장 트리가 생성되는가의 과정을 알아본다.

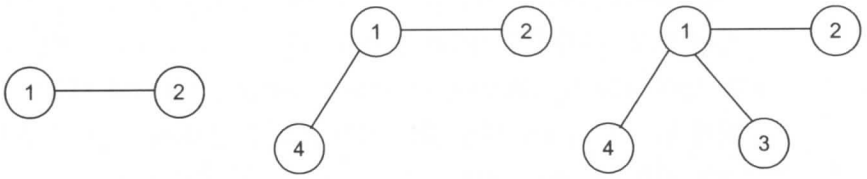


(a) 그래프 (b) 최소 비용 신장 트리

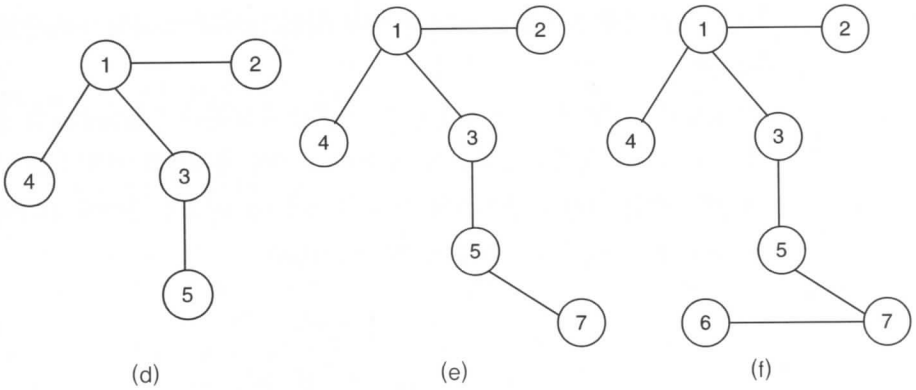
〈그림 8.16〉 그래프와 최소비용 신장 트리

먼저 임의의 정점 1을 선택한 후, 여기에 부속된 간선 중 가중값이 가장 적은 (1, 2)=14를 고르고, 다음에 정점 1과 2에 부속된 간선 중 선택되지 않은 것 중에서 가장 가중값이 적은 (1, 4)=21을 골라 연결시킨다. 다시 트리에 포함된 정점 1, 2, 4에 부속된 간선 중에서 선택되지 않은 가장 적은 (1,3)=22를 골라 연결하고, 그 다음에는 정점 1, 2, 4, 3에 부속된 간선 중에서 선택되지 않은 최소 비용의 간선 (3, 5)=31을 골라 연결한다. 이런 방법을 계속하면 그 다음에는 (5, 7)=16이, 또 그 다음에는 (7, 6)=24가 선택되는데, 이 때 모든 정점이 트리에 포함되었으므로 작업은 끝이난다. 이 과정을 나타내면 〈그림 8.17〉과 같다.

알고리즘 8.5



(a) (b) (c)



〈그림 8.17〉 최소 비용 신장 트리의 생성 과정

Prim의 알고리즘을 적용함에 있어서 처음 출발하는 정점이 어떤 것이라도 결과는 동일한 최소 비용 신장 트리가 만들어진다. 실행 과정에서 어떤 간선 (V, W)가 선택된 최소 비용의 간선이라 할지라도 이미 만들어진 중간 과정의 트리 내에 정점 V와 W가 모두 포함되어 있으면 이 간선은 제외되어야 한다. 만일 이것이 포함되면 사이클(cycle)이 형성되어 트리가 되지 않는다.

다음은 Prim의 알고리즘을 나타낸 것이다.

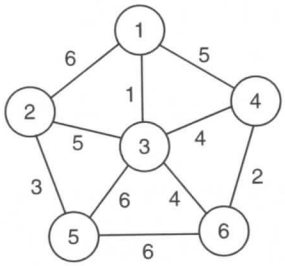
알고리즘 8.5

Prim의 최소 비용 신장 트리

```
// 그래프는 최소한 하나의 정점을 갖는다고 가정한다.
// T는 선택된 간선들의 집합이다. T를 공집합으로 초기화한다.
T = ∅;
// TV는 이미 트리에 포함된 정점들의 집합이다.
TV = {1};
// E는 그래프의 간선들의 집합이다.
while ((E!= ∅) && (|T| != n-1)) // |T|는 집합 T의 원소의 개수를 의미
{
        // (u,v)는 u ∈ TV와 v ∉ TV의 조건을 만족하는 최소 비용 간선이다.
        if ((u,v)가 존재하지 않는다)
                break;
        E = E - {(u,v)}; // 간선 (u,v)를 집합 E에서 삭제
        T = T ∪ {(u,v)}; // 간선 (u,v)를 집합 T에 추가
        TV = TV ∪ {v}; // 정점 v를 집합 TV에 추가
}
if(|T| == n-1)
        System.out.println("T는 최소 비용 신장 트리");
else
        System.out.println("그래프는 연결 그래프가 아님, 신장 트리를 갖지
        않음");
```

최소 비용 신장 트리를 만드는 또 다른 방법으로 Kruskal에 의하여 제안된 알고리즘이 있다.

이 방법은 그래프의 모든 간선을 그 가중값에 의하여 오름차순으로 정렬한 뒤, 가장 작은 간선부터 차례로 선택하여, 선택된 간선이 사이클을 이루지 않으면 신장 트리에 더한다. 이런 과정을 반복하여 신장 트리에 $(n-1)$개의 간선이 첨가되면, 즉, n개의 정점이 첨가되면 알고리즘의 수행은 정지된다.



〈그림 8.18〉 가중값 그래프

〈표 8.2〉 최소 비용 신장 트리의 생성 과정

정렬된 간선	가중값	실행	신장 트리(T)	서브 트리의 수
-	-	-	① ② ③ ④ ⑤ ⑥	6
(1, 3)	1	T에 더함	① ② ③ ④ ⑤ ⑥	5
(4, 6)	2	〃	① ② ③ ④ ⑤ ⑥	4
(2, 5)	3	〃	① ② ③ ④ ⑤ ⑥	3
(3, 6)	4	〃	① ② ③ ④ ⑤ ⑥	2
(3, 4)	4	제외	(cycle 형성)	
(2, 3)	5	T에 더함	① ② ③ ④ ⑤ ⑥	1
(1, 4)	5	terminate		
(1, 2)	6			
(3, 5)	6			
(5, 6)	6			

이와 같은 방법에 따라 알고리즘을 기술하면 다음과 같다.

알고리즘 8.6

Kruskal의 최소 비용 신장 트리

```
public void KRUSKAL(void)
/* E는 가중값에 의하여 오름차순으로 정렬된 간선들의 집합이다. */
{
  T = {};
  while (T contains less than n-1 edges && E is not empty) {
        choose a lowest edge (v,w) from E;
        DELETE (v,w) from E;   /* E에서 최소 비용의 간선을 선택한다. */
        if ((v,w) does not create a cycle in T)
              add (v,w) to T;
        else
              discard(v,w);
  }
  if (T contains fewer than n-1 edges)
        System.out.println("No spanning tree\n");
  else
        System.out.println("Spanning tree\n");
}
```

KRUSKAL 알고리즘에서 E는 그래프 G의 모든 간선들의 집합이다. 이 집합에 대한 연산은 최소의 가중값을 갖는 간선을 선택하여 삭제하는 것이다. 이러한 연산은 간선들이 순차적으로 저장되어 있을 때 효과적으로 수행된다. 미리 정렬되어 있지 않다면 히프 정렬(heap sort)을 이용하여 간선들을 $O(e \log e)$ 시간에 최소 비용 간선을 뽑아낼 수 있다. 여기서 e는 간선의 수이다.

8.5 그래프의 응용

8.5.1 최단 경로의 검색

방향 그래프(directed graph) G의 간선에 가중값이 부여되어 있을 때, 어떤 하나의 정점에서 모든 다른 정점에 이르는 최단 경로(shortest path)를 찾는 문제를 생각해 보자.

이런 문제는 정점들이 도시를 나타내고, 간선들이 도시와 도시 사이의 도로를 나타내는 데 사용되며, 간선에 부여되는 가중값은 두 도시 사이의 거리나 소요 시간 또는 유류 사용량 등이 될 수 있다. 이 때 현실적으로 부딪히는 문제로서는 어떤 두 도시 사이에 경로가 존재하는가의 여부와 만일 경로가 여러 개일 경우라면 가장 거

〈그림 8.24〉의 방향 그래프에 대한 A, A^+, A^*를 나타내면 〈그림 8.25〉와 같다.

〈그림 8.24〉 방향 그래프 G

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

(a) G의 인접 행렬 (b) G의 A^+ 행렬 (c) G의 A^* 행렬

〈그림 8.25〉 그래프 G의 여러 행렬

〈그림 8.25〉의 A^* 행렬에서 0이 아닌 요소는 경로가 존재함을 나타내고 있으므로 이것을 통하여 방향 그래프에서 임의의 두 정점간에 경로가 존재하는지의 여부를 확인할 수 있는 것이다.

8.5.4 위상 정렬

하나의 큰 과제는 작은 여러 개의 작업(activity)들로 나누어질 수 있는데, 하나의 과제를 완료하려면 작은 작업들을 일정한 순서에 따라 연속적으로 수행함으로써 끝나게 된다.

예를 들어, 전산학과의 교과 과정이 〈표 8.4〉와 같다고 하고, 정해진 모든 교과목을 이수함으로써 학사 학위를 받는다고 한다면 교과 과정의 이수가 하나의 과제(project)이고, 각각의 과목 이수가 작업(activity)이 되는 셈이다.

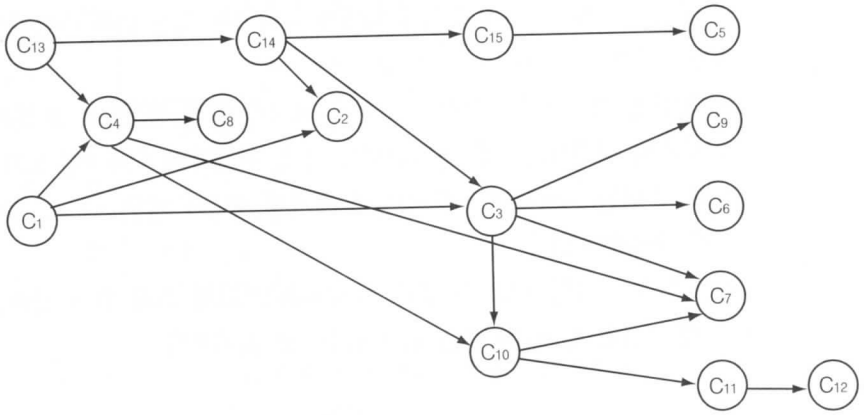
이수해야 할 과목 중에는 선수 과목에 관계없이 수강하여 이수할 수 있는 것이 있는가 하면, 어떤 과목은 다른 과목들을 선수 과목으로 이수한 후에야 수강할 수 있는 것들도 있다. 따라서 교과 과정의 각 과목 상호간의 선후 관계가 생기게 되는데, 과목을 정점으로 나타내고, 선후 관계를 방향 간선으로 나타내면 〈그림 8.26〉과 같은 방향 그래프로 표현할 수 있다.

〈그림 8.26〉과 같이 작업이 정점을 나타내고, 간선이 작업간의 선후 관계를 나타내는 방향 그래프를 특별히 정점 작업 네트워크(activity on vertex network) 또는 AOV-네트워크라고 한다. 또 여기에서 정점 i로부터 j까지의 방향 경로가 존재하면 i는 j의

〈표 8.4〉 전산학과 교과 과정

교과목 번호	교과목 이름	선수 과목 번호
C_1	컴퓨터 입문	-
C_2	수치 해석	C_1, C_{14}
C_3	자료 구조	C_1, C_{14}
C_4	어셈블리어	C_1, C_{13}
C_5	오토마타 이론	C_{15}
C_6	인공 지능	C_3
C_7	컴퓨터 그래픽스	C_3, C_4, C_{10}
C_8	이산 수학	C_4
C_9	알고리즘 분석	C_3
C_{10}	프로그래밍 언어	C_3, C_4
C_{11}	컴파일러 구조	C_{10}
C_{12}	운영 체제	C_{11}
C_{13}	해석학 I	-
C_{14}	해석학 II	C_{13}
C_{15}	선형 대수	C_{14}

선행자(predecessor)가 되고, j는 i의 후속자(successor)가 되며, 간선 $\langle i, j \rangle$에 있어서 i는 j의 즉각 선행자이고, j는 i의 즉각 후속자이다. 〈그림 8.26〉에서 C_{13}은 C_{14}, C_{15}, C_5의 선행자이고, C_5는 C_{13}, C_{14}, C_{15}의 후속자이다. 또 C_3은 C_{14}나 C_1의 즉각 후속자이면서 C_9, C_6, C_7, C_{10} 등의 즉각 선행자이다.



〈그림 8.26〉 교과 과정 방향 그래프

방향 그래프로 표시된 것이 AOV-네트워크가 되려면 이행적이면서 비반사적(irreflexive)이라야 하는데, 비반사적이려면 그래프 내에 사이클이 존재하지 않는 비사이클 그래프(acyclic graph)이어야 한다. 이행적이면서 비반사적 선후 관계를 부분 순서(partial order)라고 한다.

이제 학사 학위를 취득하기 위하여 어떤 순서로 각 교과목을 이수하여야 하는지 살펴보자. 한 번에 한 과목씩만 수강이 가능하다고 가정하면 먼저 수강할 수 있는 과목은 선수 과목이 없는 C_1 또는 C_{13} 중의 어느 하나이다. C_{13}을 먼저 이수한다면 그 다음에는 C_1, C_{14} 중의 어느 하나를 수강할 수 있다. 이 때, C_1과 C_{14}를 이수한다면 그 다음에 수강할 수 있는 것은 선수 과목의 이수가 끝난 C_2, C_4 중의 어느 하나가 될 수 있으므로 이 중 어느 한 과목을 이수한다. 이와 같은 방법으로 모든 과목을 이수한다면, 그 순서는

$C_{13}, C_1, C_4, C_8, C_{14}, C_2, C_3, C_{10}, C_{11}, C_{12}, C_7, C_6, C_9, C_{15}, C_5$

또는

$C_1, C_{13}, C_4, C_8, C_{14}, C_{15}, C_5, C_2, C_3, C_{10}, C_7, C_{11}, C_{12}, C_6, C_9$

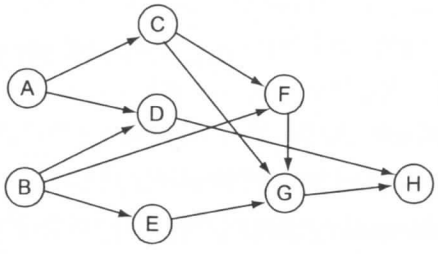
가 될 수 있고, 이외에도 여러 가지 순서 리스트가 만들어질 수 있다.

AOV-네트워크에서 만들어지는 이러한 순서 리스트를 위상 순서(topological order)라 하고, 이렇게 정렬하는 것을 위상 정렬(topological sort)이라고 한다.

이제 위상 정렬을 위한 알고리즘을 생각해 보자. 앞에서 설명한 방법에 의하면 선행자를 갖지 않는 정점이 출력 대상이 된다. 일단 어떤 정점이 출력되면 그 정점에 인접한 모든 간선을 그 정점과 함께 제거해 나아가는 일을 반복 수행하여 모든 정점이 출력된 시점에서 끝난다.

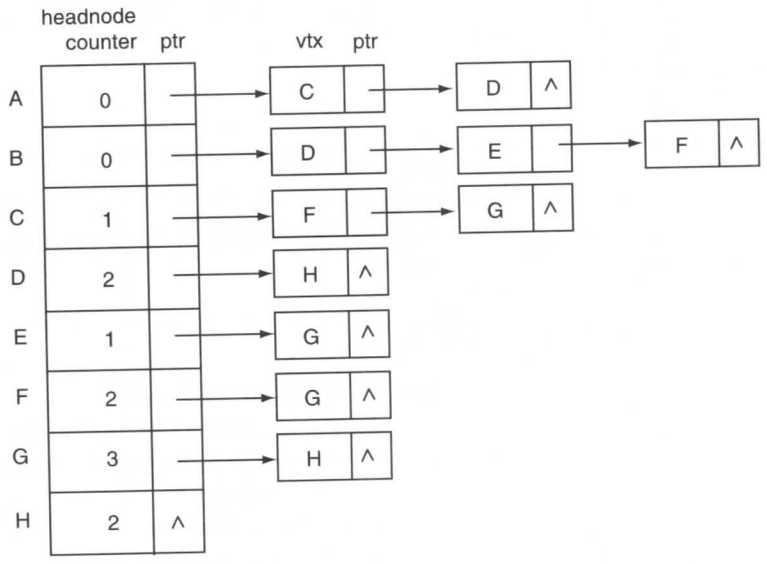
이 방법을 적용하기 위해서는 방향 그래프를 인접 리스트로 표현하고, 헤드 노드에 그 노드의 선행자의 수를 유지하도록 하여, 선행자의 수가 0인 노드를 출력 대상으로 하고, 출력된 노드에 인접한 간선이 제거될 때마다 해당 노드의 선행자의 수를 감소시켜 나아가야 한다.

예를 들어, 〈그림 8.27〉과 같은 AOV-네크워크에 대한 위상 정렬을 하고자 한다면 이것을 〈그림 8.28〉과 같은 인접 리스트로 표현한다.



〈그림 8.27〉 AOV - 네트워크

〈그림 8.28〉의 인접 리스트에서 헤드 노드의 counter가 0인 노드 A와 B를 스택에 넣는다. 스택에서 B를 꺼내어 출력하고, 여기에 인접한 간선을 제거하면 D, E, F의 counter는 1씩 감소되어 각각 1, 0, 1이 되므로 counter가 0인 E를 스택에 넣는다. 다시 스택에서 E를 꺼내어 출력하고, 여기에 인접한 간선을 제거하면 G의 counter가 1이 감소되어 2가 된다. 이번에 또 스택에서 A를 꺼내어 출력하고 A에 인접한 간선을 제거하면 C와 D의 counter는 모두 0이 되므로 C와 D를 스택에 넣는다. 다시 스택에서 D를 꺼내어 출력하고, 여기에 인접한 간선을 제거하면 H의 counter는 1이 된다. 다시 스택에서 C를 꺼내어 출력하고, 여기에 인접한 간선을 제거하면 F와 G의 counter가 각각 0, 1이 되므로 F를 스택에 넣은 후, 스택에서 F를 꺼내어 출력하고, 여기에 인접한 간선을 제거하면 G의 counter는 0이 되어 G가 스택에 들어간다. 스택에서 G를 꺼내어 출력하고, 여기에 인접한 간선을 제거하면 H의 counter가 0이 되어 스택에 들어간다. 이 때 H를 꺼내어 출력하면 수행은 종료된다.



〈그림 8.28〉 〈그림 8.27〉의 인접 리스트

이 과정을 순차적으로 나타내면 〈그림 8.29〉와 같다.
〈그림 8.29〉에 의하여 구해진 위상 순서는 B, E, A, D, C, F, G, H가 된다. 이 과정에 따른 알고리즘을 기술하면 다음과 같다.

알고리즘 8.9

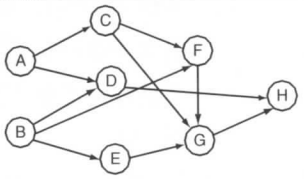
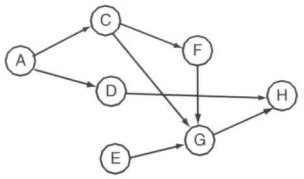
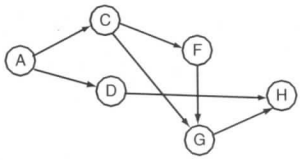
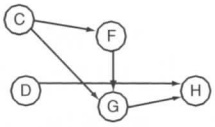
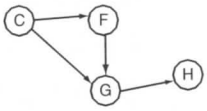
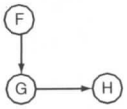
위상 정렬

```
public class node {
      int vertex;
      node link;
}
public class adjacencylists {
        int counter;
        node link;
}

public void TOPOLSORT(adjacencylists [] list, int n)
/* n개의 정점을 갖는 AOV-Network를 위상 정렬한다. */
{
   int i, j, k, top = -1;
   node ptr;
   for (i=0; i<n; i++)
       if (list[i].counter ==0 ) {
           list[i].counter = top;
           top = i;
       }


   for (i=0; i<n; i++)
      if (top == -1) {
         System.out.println( "Network has a cycle.\n" );
         exit(1);
      }
     else {
         j = top;
         top = list[top].counter;
         System.out.println("vertex"+j);
         for (ptr=list[j].link; ptr! = null; ptr=ptr.link) {
             k = ptr.vertex;
             list[k].counter--;
             if (list[k].counter == 0) {
                 list[k].counter = top;
                 top = k;
             }
      }
}
```

이 알고리즘은 while 루프가 n번 반복되므로 $O(n)$이 되는데, 루프 내에서 각 정점은 그 정점의 진출 차수만큼 시행되므로 전체 시간은

$$O\left(\left(\sum_{i=0}^{n-1} d_i\right)+n\right)=O(e+n)$$

이 된다. 여기서 d_i는 정점 i의 진출 차수이고 e는 간선의 수이다.

스택	출력	그래프
A, B	-	
A ↓ A,E	B	
A ↓ A	E	
null ↓ C,D	A	
C ↓ C	D	
null ↓ F	C	
null ↓ G	F	
null ↓ H	G	
null	H	

〈그림 8.29〉 위상 정렬의 수행 과정