강의 내용

- 오늘 강의 내용 (10월 27일)
  - 중간고사 문제풀이
  - 9.3 그래프의 순회
  - 10.1 최소비용 신장트리 (가중치 그래프)
- 예습 (11월 3일) :
  - 10장 가중치 그래프 (계속)
- 숙제:
  - 연습문제(9장) : 1,2,4,5,6,7,18,21 번 풀어보기
  - 마감일 : 2009년 11월 10일(화)

9.3 그래프 순회

- 그래프 순회
  - 주어진 어떤 정점을 출발하여 체계적으로 그래프의 모든 정점들을 방문하는 것

- 그래프 순회의 종류
  - 깊이 우선 탐색(DFS)
  - 너비 우선 탐색(BFS)

9.3.1 깊이 우선 탐색(1)

- 깊이 우선 탐색(Depth First Search : DFS) 수행
  - (1) 정점 i를 방문한다.
  - (2) 정점 i에 인접한 정점 중에서 아직 방문하지 않은 정점이 있으면, 이 정점들을 모두 스택(stack)에 저장한다.
  - (3) 스택에서 정점을 삭제하여 새로운 i를 설정하고, 단계(1)을 수행한다.
  - (4) 스택이 공백이 되면 연산을 종료한다.
- 정점 방문 여부를 표시 : 배열 visited[n]을 이용하여 표현함.

$$\text{visited}[i] = \begin{cases} \text{true}, & \text{방문하였음} \\ \text{false}, & \text{방문하지 않았음} \end{cases}$$

9.3.1 깊이 우선 탐색(2)

```
DFS(i)
                                                    // i 는 시작 정점
  for (i←0; i<n; i ← i +1) do {
      visited[i] ← false;                           // 모든 정점을 방문 안 한 것으로 마크
  }
  stack ← createStack();                                        //방문할 정점을 저장하는 스택
  push(stack, i);                                   // 시작 정점 i 를 스택에 저장
  while (not isEmpty(stack)) do {                   // 스택이 공백이 될 때까지 반복 처리
       j ← pop(stack);
      if (visited[j] = false) then {                // 정점 j를 아직 방문하지 않았다면
          visit j;                                  // 직접 j를 방문하고
          visited[j] ← true;                        //  방문 한 것으로 마크
          for (each k ∈ adjacency(j)) do {                      // 정점 j에 인접한 정점 중에서
              if (visited[k] = false) then                      // 아직 방문하지 않은 정점들을
                  push(stack, k);                               // 스택에 저장
          }
      }
  }
end DFS()
```

9.3.1 깊이 우선 탐색(3)

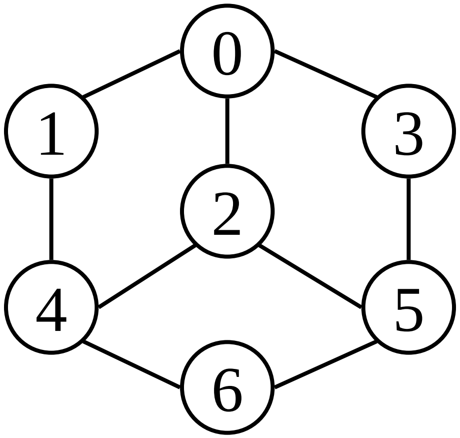


그림 9.10 탐색을 위한 그래프 G

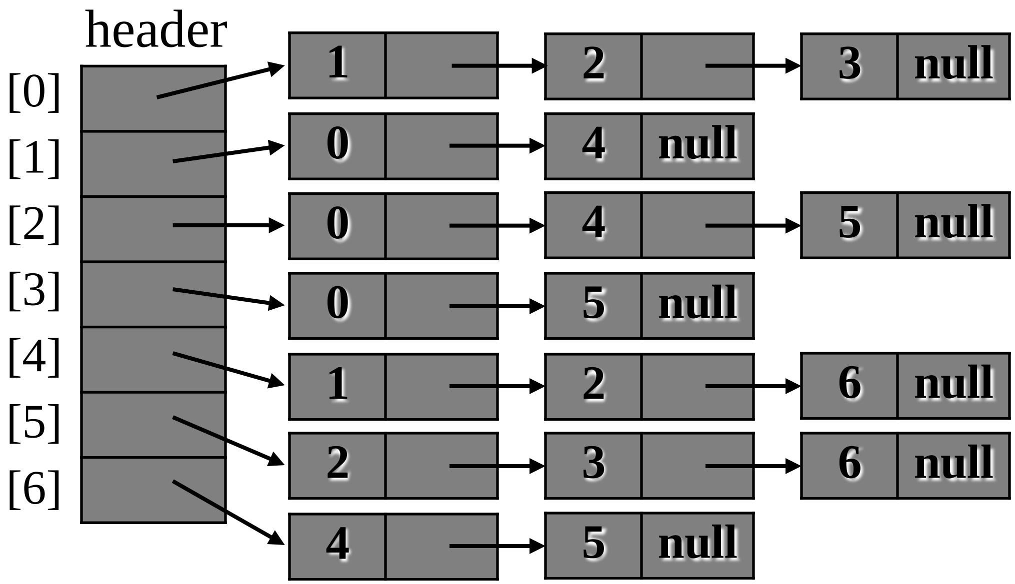
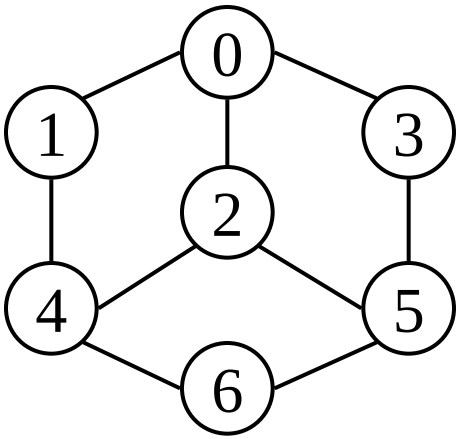


그림 9.11그래프 G에 대한 인접 리스트 표현

9.3.1 깊이 우선 탐색(4)



탐색을 위한 그래프 G

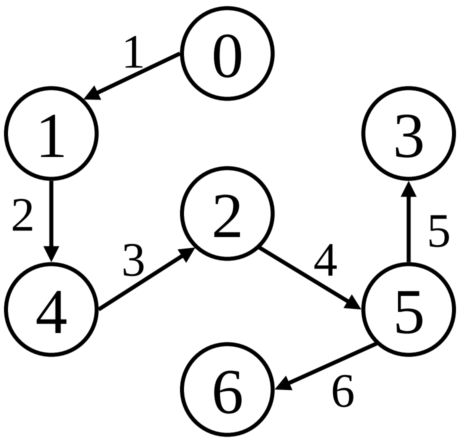


그림 9.12 그래프 G에 대한 깊이 우선 탐색 경로

(참고) G의 연결 요소 :
DFS()로 방문한 모든 정점들과 이 정점들에 부속한 G의 간선들을 모두 결합하면 곧 그래프 G의 연결요소가 된다.

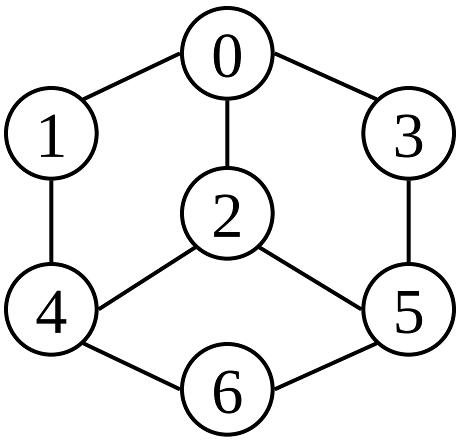
9.3.2 너비 우선 탐색(1)

- 너비 우선 탐색(Breadth First Search ; BFS) 수행
  (1) 정점 i를 방문한다.
  (2) 정점 i에 인접한 정점 중에서 아직 방문하지 않은 정점이 있으면, 이 정점들을 모두 큐(queue)에 저장한다.
  (3) 큐에서 정점을 삭제하여 새로운 i를 설정하고, 단계 (1)을 수행한다.
  (4) 큐가 공백이 되면 연산을 종료한다.
- 정점 방문 여부를 표시 방법
  - 배열 visited[n]을 이용

9.3.2 너비 우선 탐색(2)

```
BFS(i)                                                    // i는 시작 정점
   for (i←0; i<n; i ←i+1) do {
       visited[i] ← false;                               // 모든 정점을 방문 안 한 것으로 마크
   }
   visited[i] ← true;
   queue ← createQ();                                     // 방문할 정점을 저장하는 큐
   enqueue(queue, i);
   while (not isEmpty(queue)) do {
       j ← dequeue(queue);
       if (visited[j] = false) then {
           visit j;
           visited[j] ← true;
       }
       for (each k ∈ adjacency(j)) do {
           if (visited[k] = false) then {
               enqueue(queue, k);
           }
       }
   }
end BFS()
```

9.3.2 너비 우선 탐색(3)



탐색을 위한 그래프 G

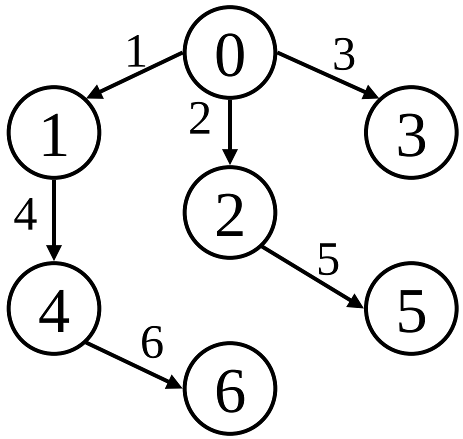


그림 9.13 그래프 G에 대한 너비 우선 탐색 경로

(참조) 너비 우선 순위로 만들어진 연결 요소는 정점 i에서 정점 j까지 도달할 수 있는 최소의 간선을 이용한 최단 경로를 표현함.

9.3.3 연결 요소(1)

- 연결 그래프인지의 여부를 판별하는 방법
  - DFS나 BFS 알고리즘 이용함.
  - 무방향 그래프 G에서 하나의 정점 i에서 시작하여 DFS(or BFS)로 방문한 노드 집합 V(DFS(G, i))가 V(G)와 같으면 G는 연결 그래프.

V(DFS(G, i)) = V(G) : 연결 그래프, 하나의 연결 요소
V(DFS(G, i))⊂ V(G) : 단절 그래프, 둘 이상의 연결 요소

- 연결 요소 찾기 방법
  - 정점 i에 대해 DFS (or BFS) 수행함.
  - 두개 이상의 연결 요소가 있는 경우에는 방문하지 않은 나머지 정점 j에 대해 DFS(or BFS) 반복 수행하면 연결 요소를 모두 찾을 수 있음.

9.3.3 연결 요소(2)

- 연결 요소를 찾는 알고리즘

```
dfsComponent(G, n)                    // G=(V,E), n은 G의 정점 수
  for (i←0; i <n; i ← i +1) do {
      visited[i] ← false;
  }
  for (i ←0; i <n; i ← i +1) do {
                                      // 모든 정점 0, 1, …, n-1에 대해 연결 요소 검사
      if (visited[i] = false) then {
          print("new component");
          DFS(i);               // 정점 i가 포함된 연결 요소를 탐색
      }
  }
end dfsComponent()
```

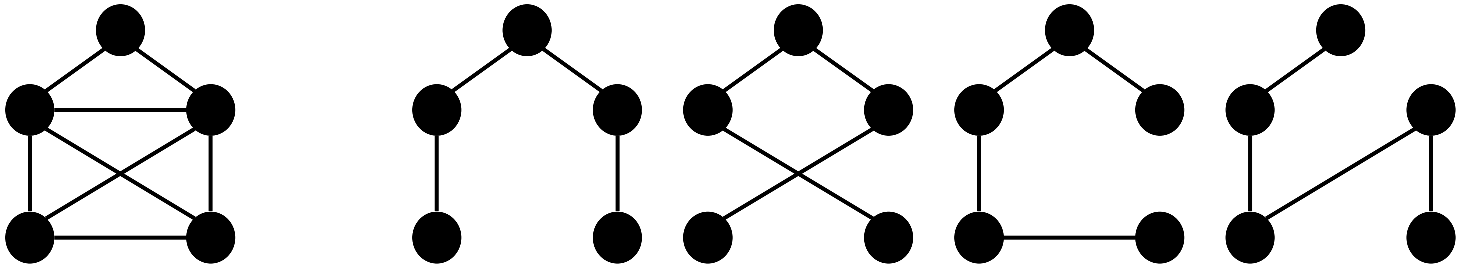
- (참조) DFS(i)를 BFS(i)로 대체해도 무방

9.3.4 신장 트리(Spanning Tree)(1)

- 트리 간선(tree edge)
  - G가 연결 그래프일 때
    - DFS나 BFS는 G의 모든 정점 방문할 때, G의 간선들은
      - 방문에 사용한 간선들(트리 간선)과
      - 그렇지 않은 간선들(비트리 간선)로 나뉨
  - 트리 간선을 구하는 방법
    - 방문에 사용된 간선 (j, k)의 집합을 T라 할 때
    - DFS와 BFS 알고리즘의 for속의 if-then 절에 명령문 T ← T∪{(j, k)} 삽입시켜 구할 수 있음.
    - T에 있는 간선들을 전부 결합시키면 그래프 G의 모든 정점들을 포함한 트리가 됨. 이러한 간선을 **트리 간선**이라 함

9.3.4 신장 트리(2)

- 신장 트리(spanning tree)
  - 그래프 G에서 E(G)에 있는 간선과 V(G)에 있는 모든 정점들로 구성된 트리를 말함.
  - DFS, BFS에 사용된 간선의 집합 T는 그래프 G의 신장 트리를 의미함.
  - 주어진 그래프 G에 대한 신장 트리는 유일하지 않음
  - 그림 9.14 연결 그래프 G와 신장 트리

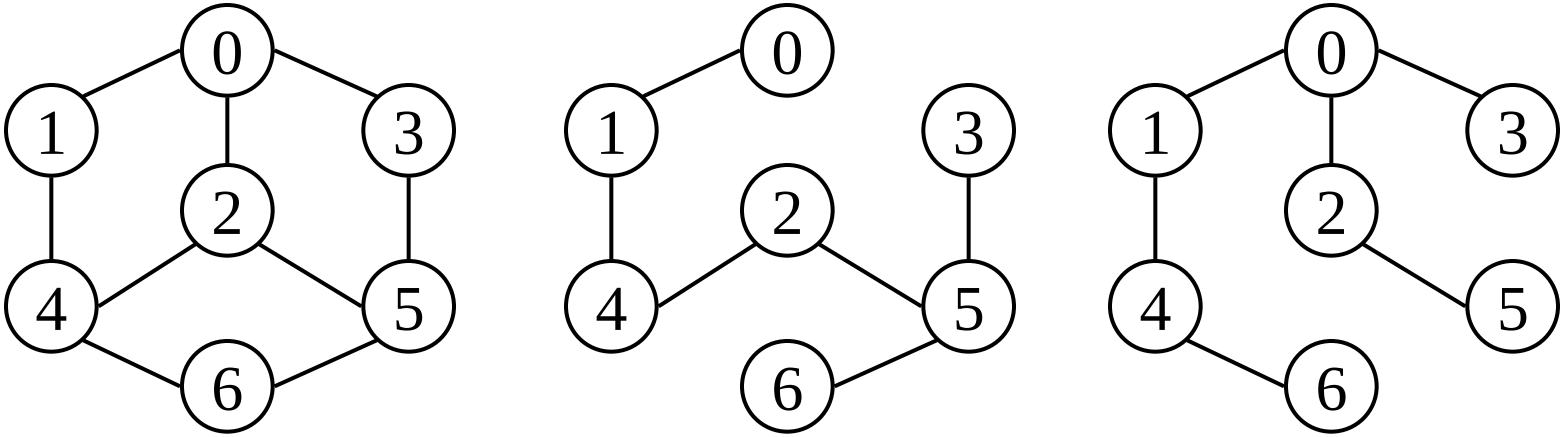


(a) 연결 그래프 G

(b) 신장 트리

9.3.4 신장 트리(3)

- 신장 트리의 종류
  - 깊이 우선 신장 트리(depth first spanning tree) : DFS 사용
  - 너비 우선 신장 트리(breadth first spanning tree) : BFS 사용

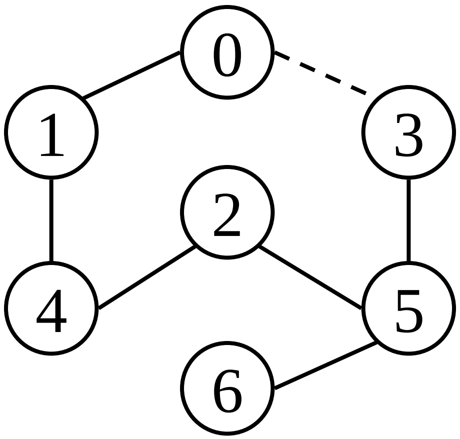


(a) 연결 그래프 G (b) DFS(0) 신장 트리 (c) BFS(0) 신장 트리

그림 9.15 연결 그래프 G와 신장 트리

9.3.4 신장 트리(4)

- 비트리 간선(nontree edge) 집합(NT)
  - 신장 트리에 사용되지 않은 간선들의 집합
  - NT에 있는 임의의 간선 (i, j)를 신장 트리 T에 첨가시키면 그 즉시 사이클이 만들어 진다.
  - 신장 트리에 비트리 간선을 첨가하면 더 이상 트리가 아님
  - 예) DFS(0) 신장 트리
    - 간선 (0, 3) 첨가 : 0, 1, 4, 2, 5, 3, 0으로 구성된 사이클 형성



9.3.4 신장 트리(5)

- 최소 연결 부분 그래프(minimal connected subgraph)
  - G의 부분 그래프 G' 중에서 다음 조건을 만족하는 그래프

    (1) V(G') = V(G)

    (2) E(G') ⊆ E(G)

    (3) G'는 연결 그래프

    (4) G'는 최소의 간선 수를 포함

- 신장 트리는 최소 연결 부분 그래프로서, n-1개의 간선을 가짐
  - n개의 정점을 가진 연결 그래프는 최소한 n-1개의 간선 필요
  - n-1개의 간선을 가진 연결 그래프는 트리임.
- 통신 네트워크 설계에 응용됨.
  - 정점이 도시, 간선이 도시간의 통신 링크를 나타냄.
  - 도시간 네트워크 설계에서 최소 링크 수(n-1)로 연결하는 방법

10장 가중치 그래프

10.1 최소 비용 신장 트리
10.2 최단 경로
10.3 위상 순서
10.4 임계 경로

10.1 최소 비용 신장 트리

- 가중치 그래프(Weighted Graph) 혹은 네트워크(Network)
  - 간선에 가중치가 부여된 그래프
- 최소 비용 신장 트리(minimum cost spanning tree)
  - 신장 트리 비용은 신장 트리를 구성하는 간선들의 가중치를 합한 것
  - 이 비용이 최소가 되는 신장 트리를 말함.
- 최소 비용 신장 트리를 구하는 알고리즘들
  - Kruskal 알고리즘
  - Prim 알고리즘
  - Sollin 알고리즘
- 갈망 기법(greedy method)
  - 최적의 해를 단계별로 구함
  - 각 단계에서 생성되는 중간 해법이 그 단계까지의 상황에서는 최적임.
- 신장 트리의 제한 조건
  - 가중치가 부여된 무방향 그래프
  - n – 1 (n= |V|)개의 간선만 사용하되 사이클을 생성하는 간선 사용 금지

10.1.1 Kruskal 알고리즘(1)

- 방법
  - 한번에 하나의 간선을 선택하되 비용이 가장 작은 간선을 택하여, 최소 비용 신장 트리 T에 추가함. n 개의 정점을 가진 그래프 G의 간선의 집합 E(G)로부터 n-1개의 간선을 선정하는 것임.
  - 비용이 가장 작은 간선을 선정하되, 이미 T에 포함된 간선들과 사이클이 만들어지는 것은 제외시킨다.
  - 비용이 같은 간선들은 임의의 순서로 하나씩 추가함.

- 구현
  - 최소 비용 간선 선택
    - 그래프 G의 간선들을 가중치에 따라 오름차순으로 정렬한 간선의 순차 리스트 유지함.
  - 사이클 검사
    - T에 추가로 포함될 정점들을 연결 요소별로 정점 그룹을 만들어 유지
    - 간선 (i, j)가 T에 포함되기 위해서는 사이클이 형성되지 않아야 함. 즉, 정점 i와 j가 각각 상이한 정점 그룹에 속해 있어야 함.

10.1.1 Kruskal 알고리즘(2)

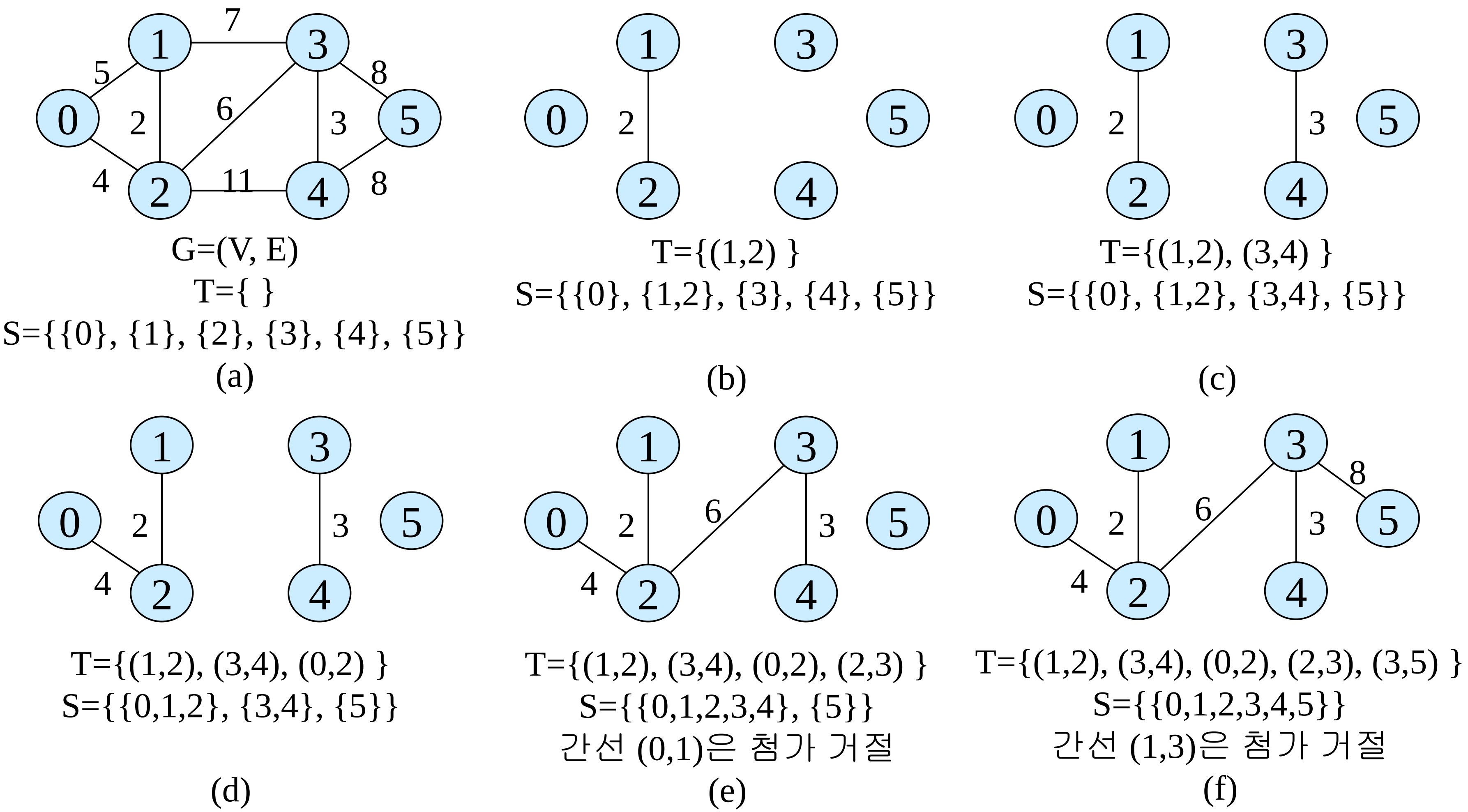


그림 10.1 Kruskal 알고리즘 수행 단계

10.1.1 Kruskal 알고리즘(3)

```
Kruskal(G,n)
                              //G=(E,V)이고 n=|V|, |V|는 정점 수
  T ← ∅;
  edgelist ← E(G);            // 그래프 G의 간선 리스트
  S0 ← {0}, S1 ← {1} , … , Sn-1 ←{n-1};
  while (|E(T)|<n-1 and |edgeList|>0) do {
                     // |E(T)|는 T에 포함된 간선 수, |edgeList|는 검사할 간선 수
    select least-cost (i, j) from edgeList;
    edgeList ← edgeList - {(i, j)};              // 간선 (i, j)를 edgeList에서 삭제
    if ({i, j} 가 동시에 Sk for any k에 속하지 않음) then {
      T ← T ∪ {(i, j)};                          // 간선 (i, j)를 T에 첨가
      Si ← Si ∪ Sj;                            // 간선이 부속된 두 정점 그룹을 합병
    }
  }
  if (|E(T)|<n-1) then {
    print ('no spanning tree');
  }
  return T;
end Kruskal()
```

10.1.2 Prim 알고리즘(1)

- 방법
  - 한번에 하나의 간선을 선택하여, 최소 비용 신장 트리 T를 구축함.
  - Kruskal 알고리즘과는 달리 구축 전 과정을 통해 하나의 트리만을 계속 확장해 나가는 방법임.

- 구현
  - 하나의 정점 u를 트리의 정점 집합 V(T)에 추가
  - V(T)의 정점들과 인접한 정점들 중 최소 비용 간선 (u, v)를 선택하여 T에 포함시키고, 새로 선정된 정점은 V(T)에 포함.
  - T가 n-1개의 간선을 포함할 때까지 반복한다. 즉, 모든 정점이 V(T)에 포함될 때까지 반복
  - 항상 새로 선택되는 간선 (u, v)은 u 또는 v 어느 하나만 T에 속하는 간선으로서 최소 비용을 가진 간선임. 즉, 사이클이 형성되지 않도록 선택해야 함.

10.1.2 Prim 알고리즘(2)

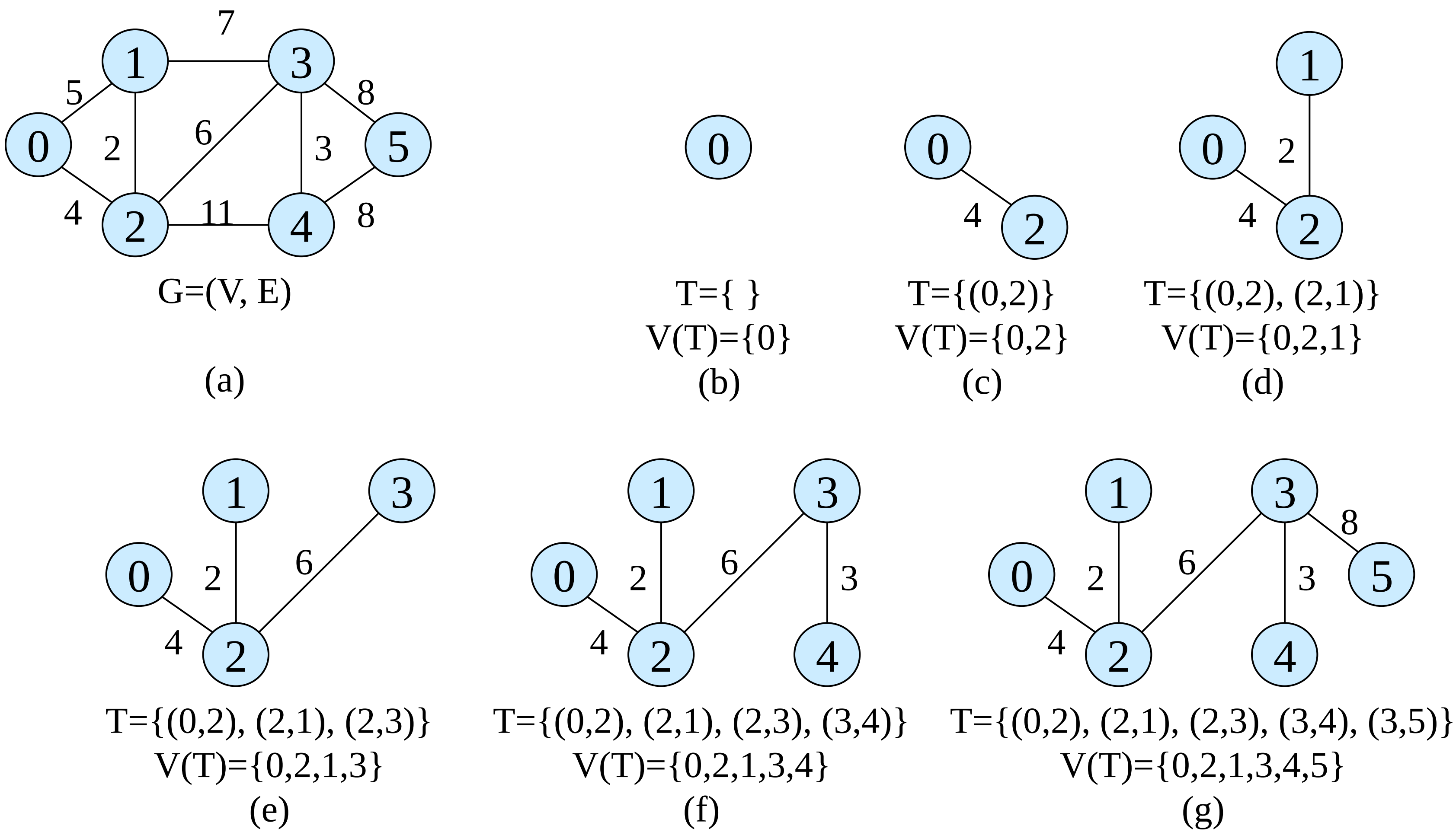


그림 10.3 Prim 알고리즘의 수행 단계

10.1.2 Prim 알고리즘(3)

```
Prim(G, i)                                  // i는 시작 정점
        T ← ∅;                              // 최소 비용 신장 트리
        V(T) = { i };                       // 신장 트리의 정점
        while (|T| < n-1) do {
            if (select least-cost (u, v) such that u ∈ V(T) and v ∉ V(T) then  {
                T ← T ∪ {(u, v)};
                V(T) ← V(T) ∪ {v};
            }
            else  {
                print("no spanning tree");
                return T;
            }
        }
        return T;
end Prim()
```

10.1.3 Sollin 알고리즘(1)

- 방법
  - 한 단계마다 여러 개의 간선을 선택하면서 최소 비용 신장 트리를 구축해 나가는 방법임.
  - 구축 과정 중에 두 개의 트리가 하나의 동일한 간선을 중복으로 선정할 경우, 하나의 간선만 사용하고 중복된 또 다른 간선은 사용하지 않음.

- 구현
  - 그래프의 각 정점 하나만을 포함하는 n개의 트리로 구성된 신장 포리스트(forest)에서부터 시작
  - 매번 포리스트에 있는 각 트리마다 하나의 간선을 선택
  - 선정된 간선들은 각각 두 개의 트리를 하나로 결합시키면서 신장 트리로 확장해 나감.
  - n-1개의 간선으로 된 하나의 트리가 만들어지거나, 더 이상 선정할 간선이 없을 때 종료함.

10.1.3 Sollin 알고리즘(2)

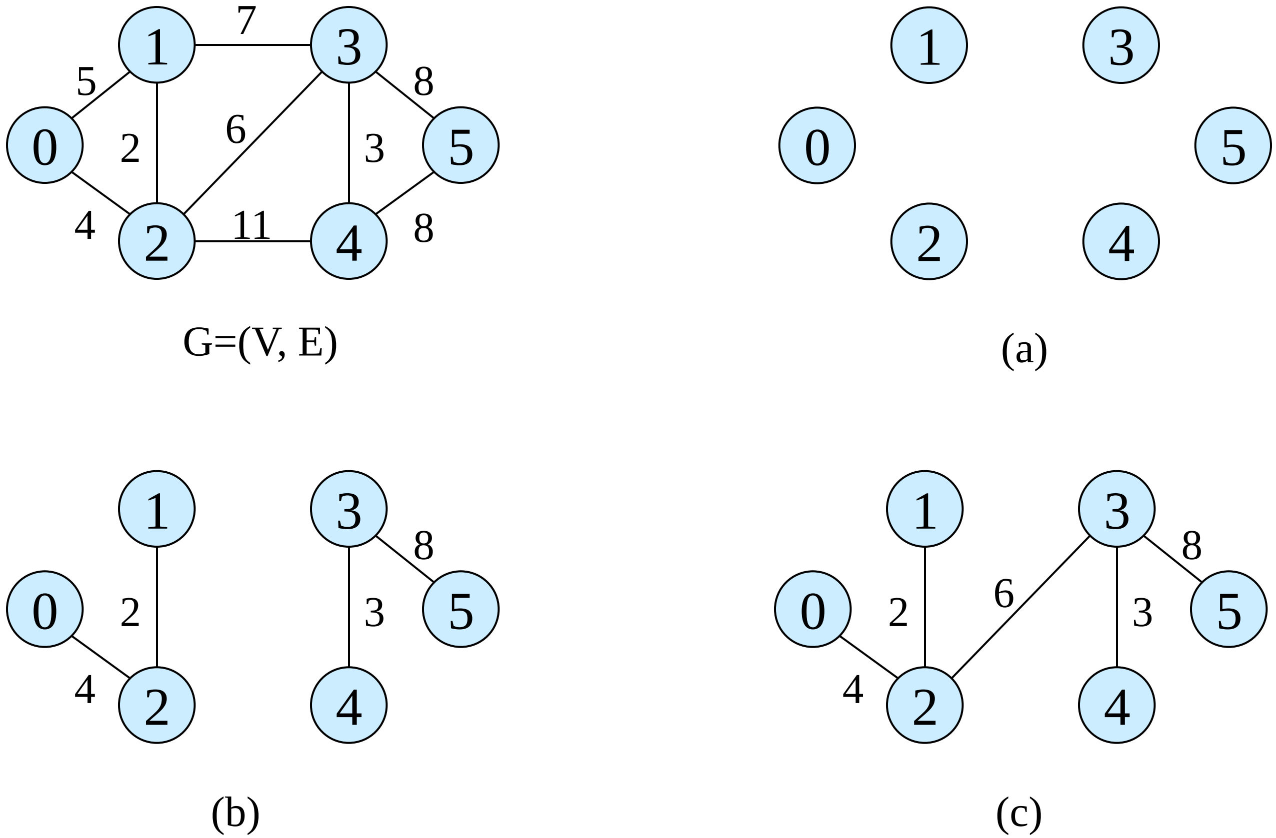


그림 10.4 Sollin 알고리즘의 수행 단계

10.1.3 Sollin 알고리즘(3)

```
Sollin(G, n)
                                                   // G = (V, E), n = |V|
  S0 ← {0}; S1 ← {1}; , ... , Sn-1 ← {n-1};        // n개의 노드 그룹으로 초기화
  T ← ∅;                                           // 최소 비용 신장 트리
  List ← ∅;                                        // 연산 단계에서 선정된 간선
  while (|T| < n-1 and Edges ≠ ∅ ) do {
      for (each Si) do {
          select least-cost (u, v) from Edges such that u∈Si and v∉Si;
      if ((u, v)∉List) then List ← List ∪ {(u, v)};    // 중복 간선은 제거
  }
  while (List ≠ ∅) do {                              // List가 공백이 될 때까지
      remove (u, v) from List;
      if ({u, v} ⊄ Su or {u, v} ⊄ Sv) then {   // Su와 Sv는 각각 정점 u와 v가 포함된 트리
          T ← T ∪ {(u, v)};
          Su ← Su ∪ Sv;
          Edges ← Edges – {(u, v)}; }}}
  if ((|T| < n-1) then {
      print("no spanning tree"); }
  return T;
end Sollin()
```