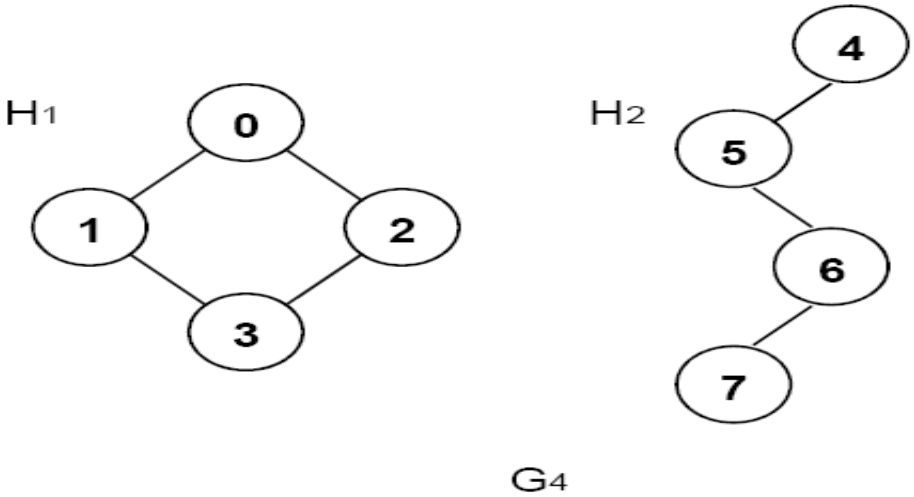
Graphs – 보충자료

Data Structures II

연결요소

- 무방향그래프(undirected graph) G에서, v_0부터 v_1 까지 경로가 있으면 v_0 와 v_1은 연결되었다고 한다.
- 서로 다른 정점들의 모든 쌍에 대해 경로가 있으면 그래프가 연결되었다고(connected) 한다.
- 연결요소(connected component) : 최대 연결 부분그래프 (a maximal connected subgraph)



H1 and H2 are two components of G4

연결요소에 관한 문제

- Problem 1: 무방향 그래프가 연결되었는지 결정하라
  - DFS(0) 또는 BFS(0)를 호출한 후 방문되지 않은 정점이 있는지 찾는다.
  - 인접리스트를 이용할 경우 O(n+e)

- Problem 2: 그래프의 연결요소들을 나열하라.
  - 아직 방문되지 않은 정점 v가 있을 때 DFS(v) 또는 BFS(v)를 호출하라

Determine the connected components in G

```
public void connected(void){
    int i;
   for (i=0; i<n; i++)
       visited[i] = false;
    for (i=0; i<n; i++)
       if (visited[i] != true){
           DFS(i);
           System.out.printfln("\n");
       }
}
```

Analysis : O(n+e) for adjacency lists

Minimum cost spanning tree

- 신장트리(spanning tree) : 그래프의 모든 정점을 연결하는 서브트리이다
- G가 n개의 정점을 가질 때, G의 신장트리는 n개의 정점과 n-1개의 간선으로 구성된다.
- 신장트리는 어떻게 찾을 수 있는가?

- 가중치 그래프에 대해,
  최소비용신장트리(minimum cost spanning tree)
  를 구하라

Minimum cost spanning tree (2)

가중치 무방향 그래프 G에 대해

- 신장트리의 비용(Cost) :
  : 신장트리의 간선들의 가중치의 합
- 최소비용신장트리 (Minimum cost spanning tree)
  : 가장 적은 비용의 신장트리

- 최소비용신장트리를 구하는 세 개의 알고리즘 (by *Greedy method*)
  -- Kruskal's algorithm
  -- Prim's algorithm
  -- Sollin's algorithm

Minimum cost spanning tree (3)

- 갈망법(Greedy method)
  - 최적의 해를 단계별로 구한다.
  - 각 단계에서는 몇 개의 판단기준에 따라 최상의 결정을 내린다

- 신장트리의 경우 최저비용을 기준으로 사용한다
  - 그래프 내에 있는 간선들만을 사용해야 한다
  - |V(G)|=n 일 때 정확히 n-1개의 간선들만을 사용해야 한다
  - 사이클을 생성하는 간선을 사용해서는 안 된다

Kruskal’s algorithm

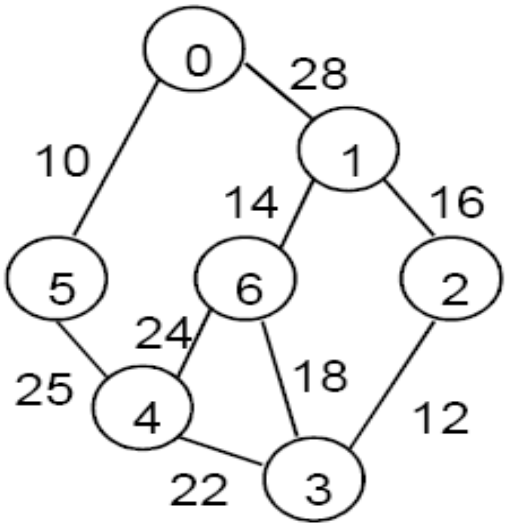
T = { }

- 간선들을 비용이 증가하는 순서로 정렬한다
- 이미 T에 있는 간선들과 함께 사이클을 형성하지 않으면 적은 비용을 가지는 간선들부터 차례로 T에 추가한다
- n개의 정점을 가지는 연결그래프에서는 정확히 n-1 개의 간선이 선택된다.

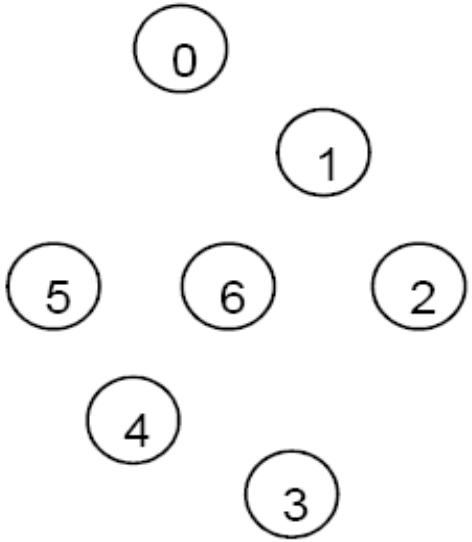
Kruskal's algorithm

```
T ={ };
/* E는 가중치에 의해 오름차순으로 정렬된 간선들의 집합*/
while (T contains less than n-1 edges and E is not empty){
    choose a least cost edge (v,w) from E;
    delete (v,w) from E;
    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)
    printf("No spanning tree\n");
```

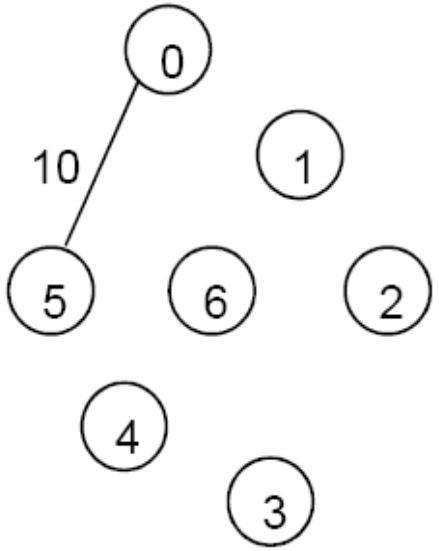
Example



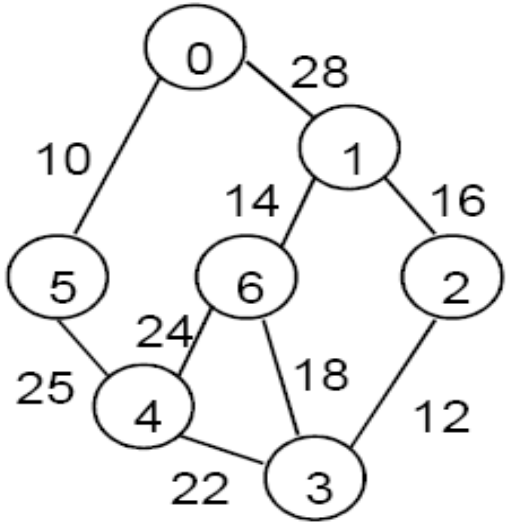
(a)



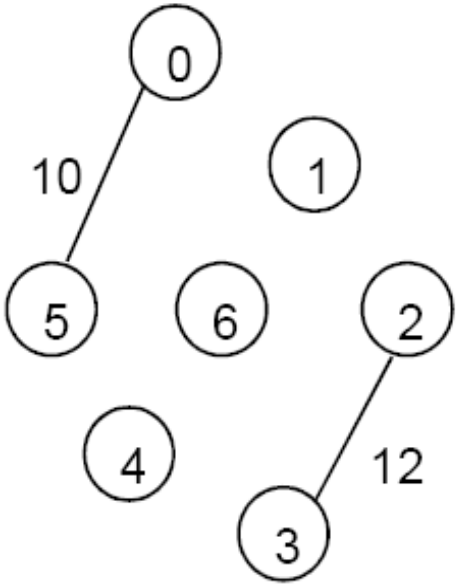
(b)



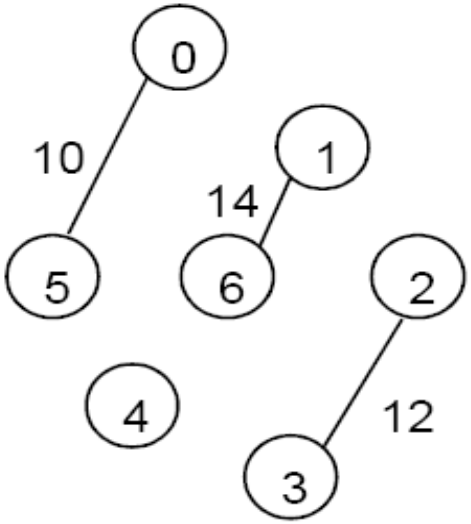
(c)


0
28
10
1
14
16
5
6
2
24
25
18
4
12
22
3

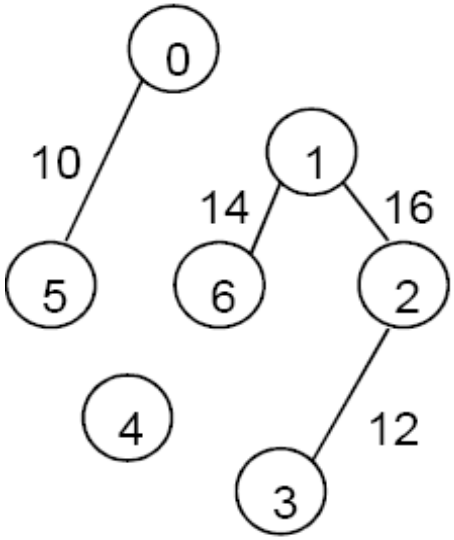
(a)


0
10
1
5
6
2
4
12
3

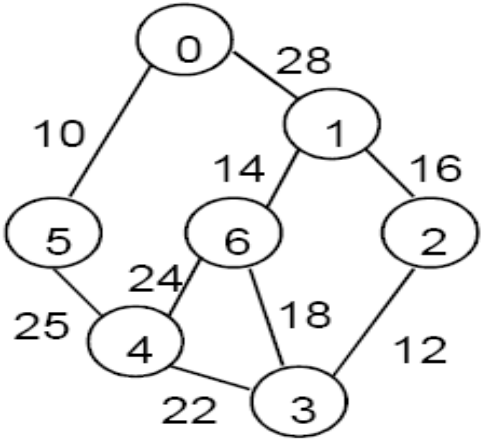
(d)


0
10
1
14
5
6
2
4
12
3

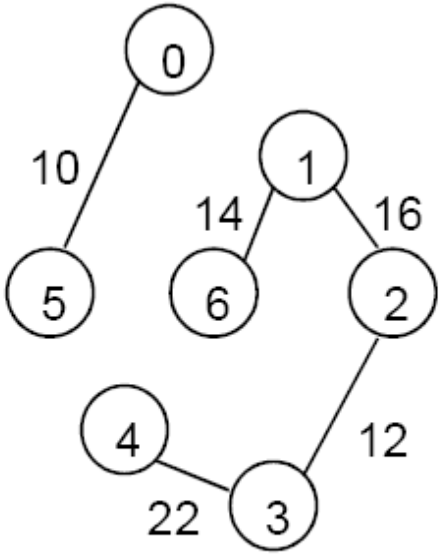
(e)


0
10
1
14
16
5
6
2
4
12
3

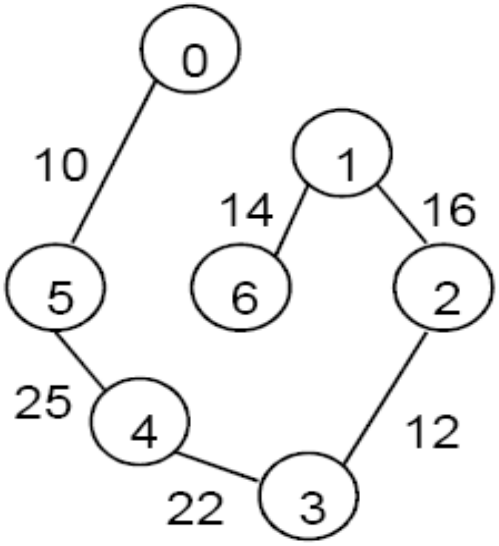
(f)


0
28
10
1
14
16
5
6
2
24
25
18
4
12
22
3

(a)


0
10
1
14
16
5
6
2
4
12
22
3

(g)


0
10
1
14
16
5
6
2
25
4
12
22
3

(h)

Analysis of Kruskal's algorithm

- 간선들의 집합 E에서 최소비용의 간선을 선택할 때,
  min heap을 사용하면
  - heap을 구성할 때: O(e)
  - 최소비용 간선을 찾아서 삭제하는데 O(loge)

- 한 간선이 T에 있는 간선들과 함께 사이클을 형성하지 않는것을 확인할 때
  - union-find set operation: less than O(loge)
  - T의 각 연결요소를 요소 내의 정점들의 집합으로 간주

  총 시간 복잡도: O(e loge)

Set Representation by Trees

S1 = {0, 6, 7, 8}
S2 = {1, 4, 9}
S3 = {2, 3, 5}

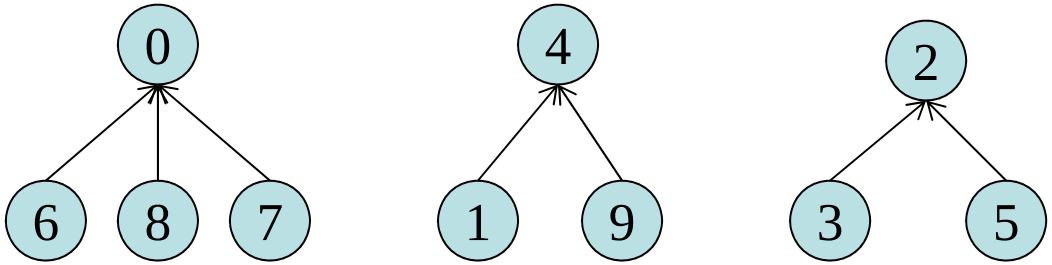
} Pairwise disjoint

Assume that the elements of the set are the numbers 0,1, …, n-1

<u>Two basic operations</u>

1. Union of two disjoint sets : UNION(S_i, S_j)
   UNION (S1, S2) = { 0, 6, 7, 8, 1, 4, 9 }
2. Find the set containing the element i : FIND(i)
   FIND(3) = S_3
   FIND(8) = S_1

Possible Representation



S_1 S_2 S_3

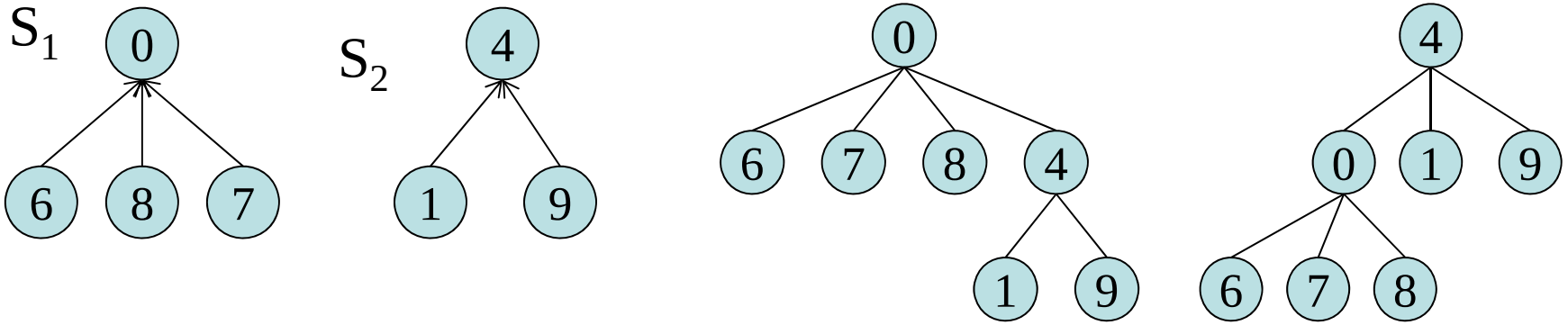
- Array Representation

0	1	2	3	4	5	6	7	8	9	i
-1	4	-1	2	-1	2	0	0	0	4	parent(i)

Set Name : root label

$S_1 \rightarrow 0$ $S_2 \rightarrow 4$ $S_3 \rightarrow 2$

Union and Find




```
void union( int i, int j ) {
      parent[i] = j
}

int find( int i ){
    for ( ; parent[i] ≥ 0; i = parent[i] )
          ;
    return i;
}
```

Analysis of union & find

Initially, $S_i = \{i\}$, $0 \le i < n$

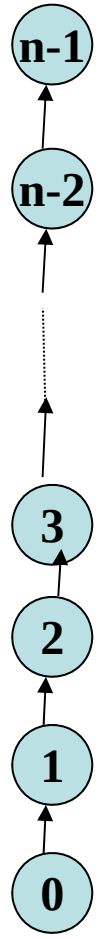
Process Sequence :

Union(0,1), find(0)
Union(1,2), find(0)
:
Union(n – 2, n – 1), find(0)

Union(i, i+1) : O(1)
find (0) : O(i) for the element in level i

Totally,

$$\sum_{i=1}^{i=n-1} i = \mathbf{O}(n^2)$$



Weighting Rule for union(i, j)

- Weighting Rule for union(i, j)

  Let #i be the number of elements in the set i

  Let #j be the number of elements in the set j

  If #i < #j, then make j the parent of i

  If #i ≥ #j, then make i the parent of j

- Modification of Representation :

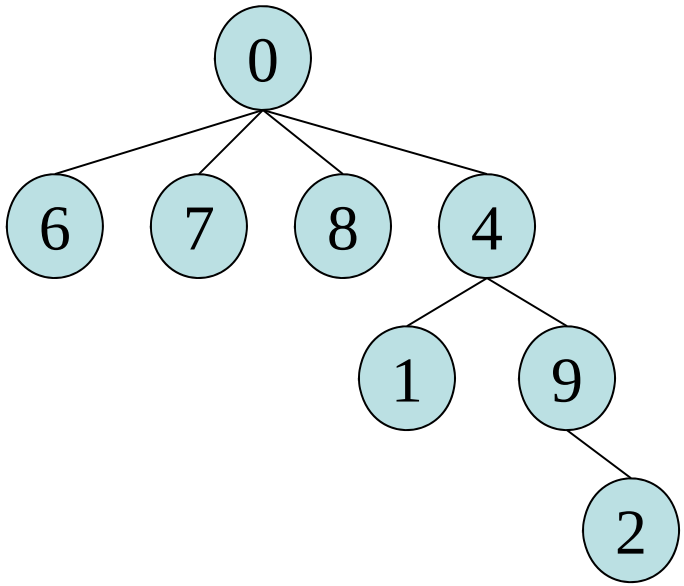
  Each root has the negative of the number of elements of the corresponding set.

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

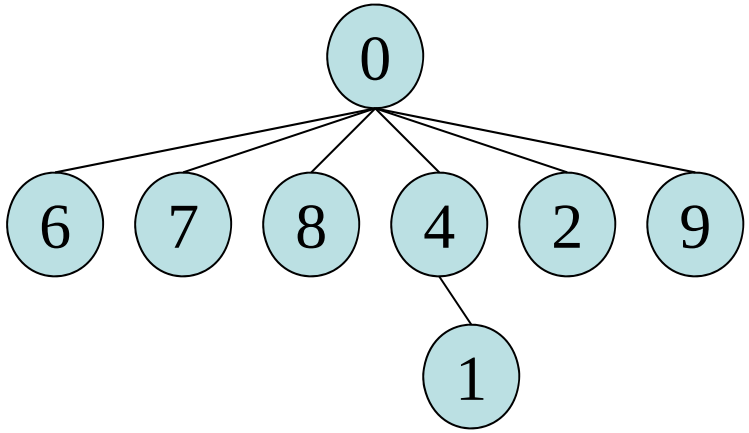
Collapsing Rule for Find(i)

If *j* is a node in the path from *i* to its root,
then make *j* a child of the root.

Find(2) = 0



collapsingFind(2) = 0



0	1	2	3	4	5	6	7	8	9	10
-8	4	0	-3	0	3	0	0	0	0	3

WeightedUnion & CollapsingFind

다음의 메소드들을 완성하시오

WeightedUnion(i,j){ }

CollapsingFind(i){ }

Prim's algorithm

- 하나의 정점으로 된 트리 T에서 시작한다
- 최저비용간선 (u,v)을 구해 T∪{(u,v)} 도 트리가 되면 그 간선을 T에 추가한다
- T가 n-1개의 간선을 가질 때까지 간선의 추가 단계를 반복한다

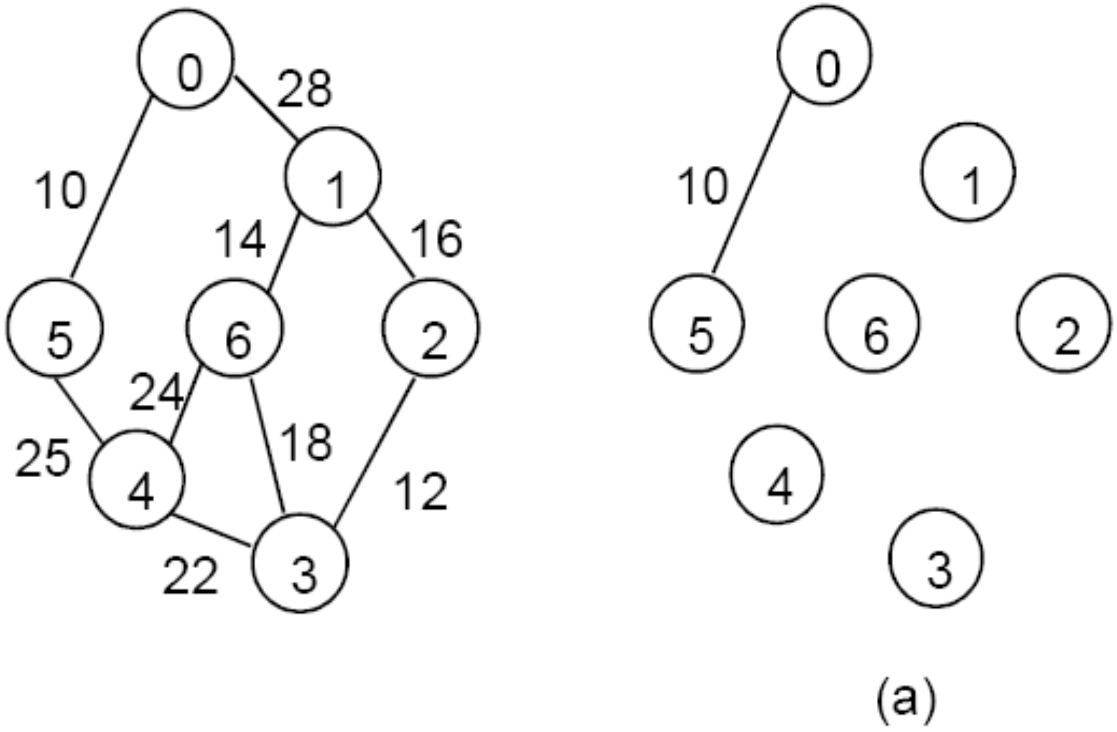
- 알고리즘의 각 단계에서 선택된 간선들의 집합은
  - Prim's algorithm에서는 ??을 형성
  - Kruskal's algorithm에서는 ??을 형성

Prim's algorithm

```
T={ };   /* the set of tree edges */
TV={0};  /* the set of tree vertices */
while (T contains fewer than n-1 edges){
    let (u,v) be a least cost edge such that u ∈TV and v ∉TV;
    if (there is no such edge)
       break;
    add v to TV;
    add (u,v) to T;
}
if (T contains fewer than n-1 edges)
    printf("No spanning tree\n");
```

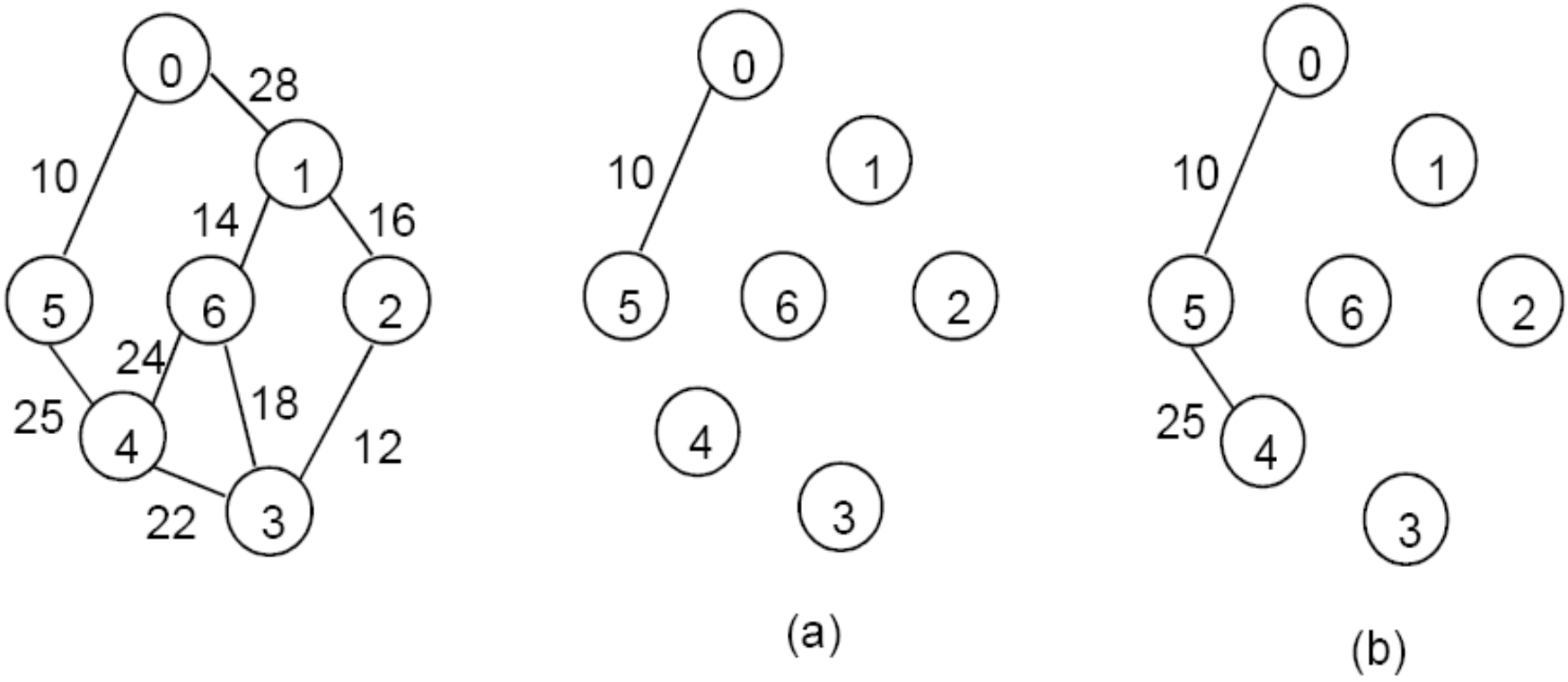
Make sure that the added edge does not form a cycle

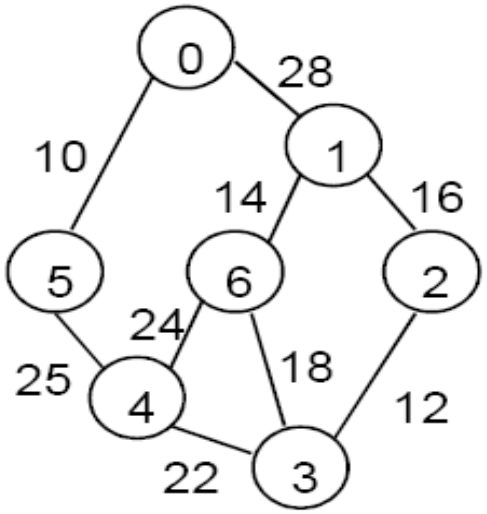
Stages in Prim's algorithm

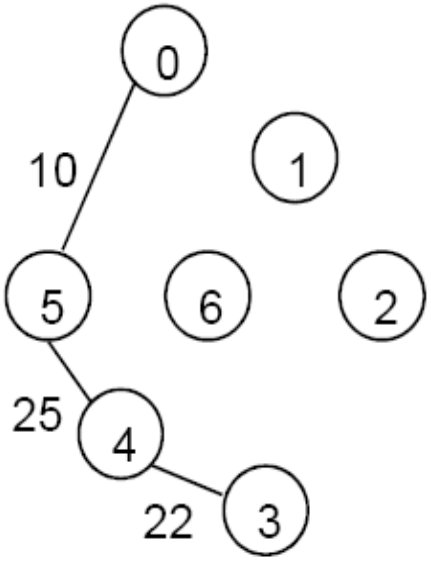


(a)

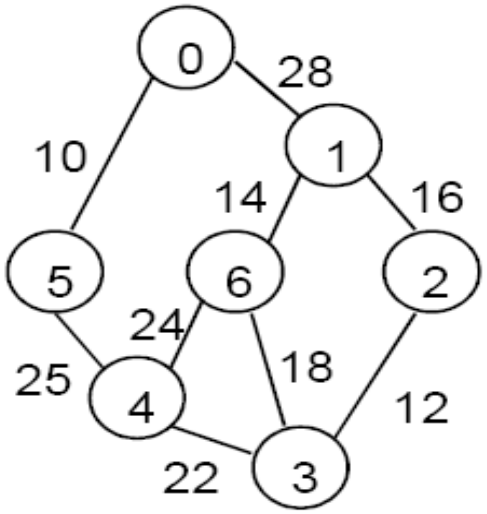
Stages in Prim's algorithm

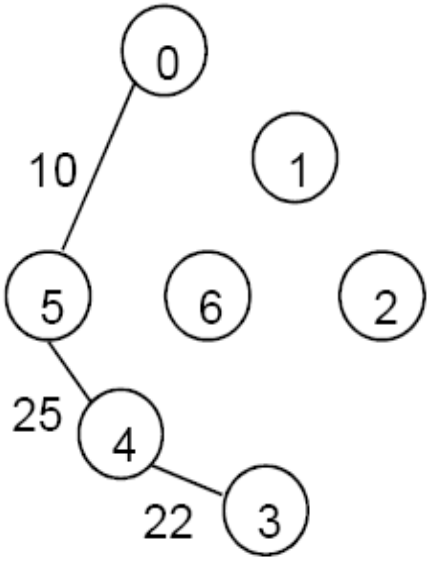




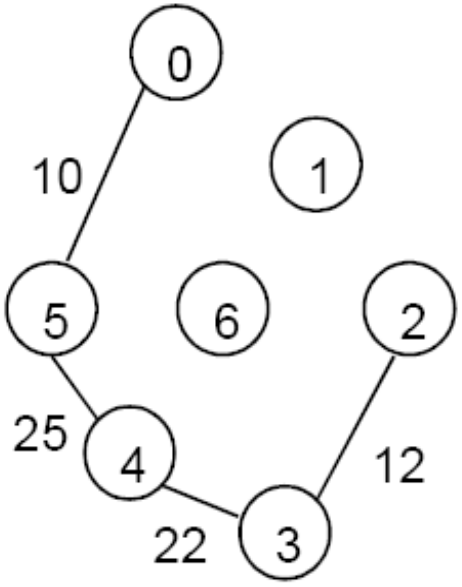


(c)

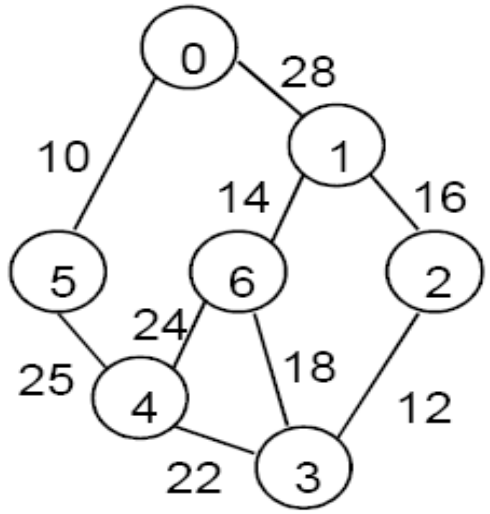

0
28
10
1
14
16
5
6
2
24
25
18
4
12
22
3

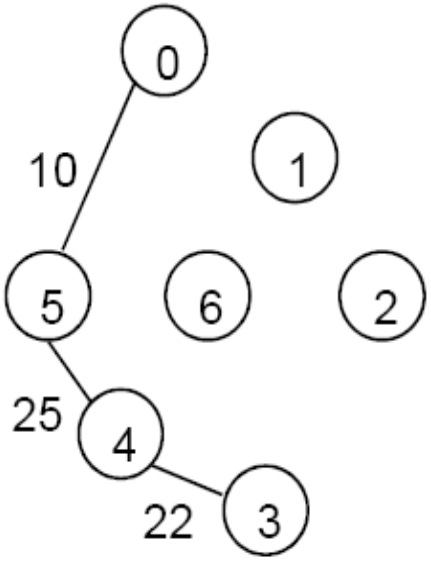

0
10
1
5
6
2
25
4
22
3

(c)

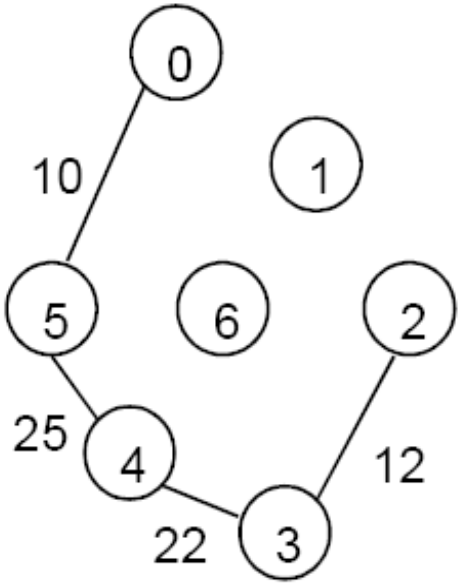

0
10
1
5
6
2
25
4
12
22
3

(d)

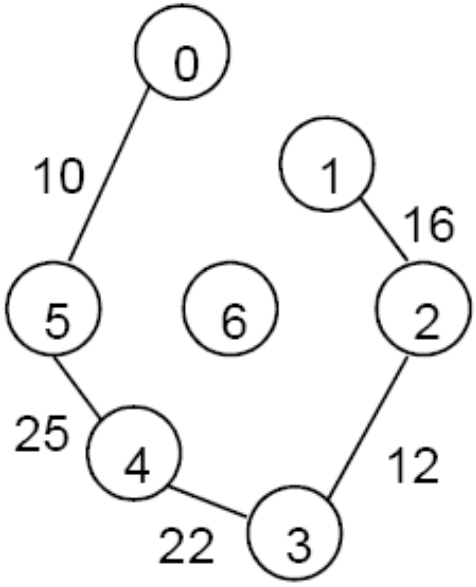

0
28
10
1
14
16
5
6
2
24
25
18
12
4
22
3


0
10
1
5
6
2
25
4
22
3

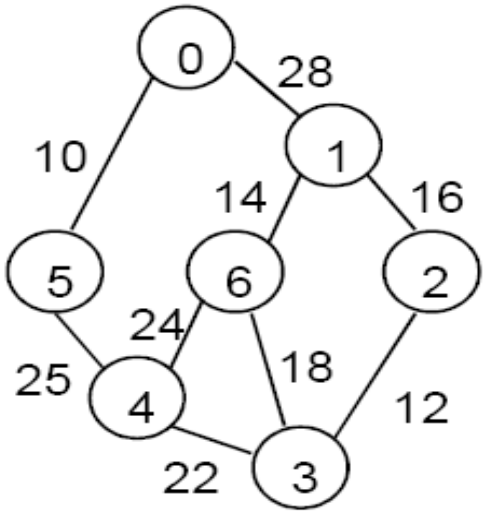
(c)

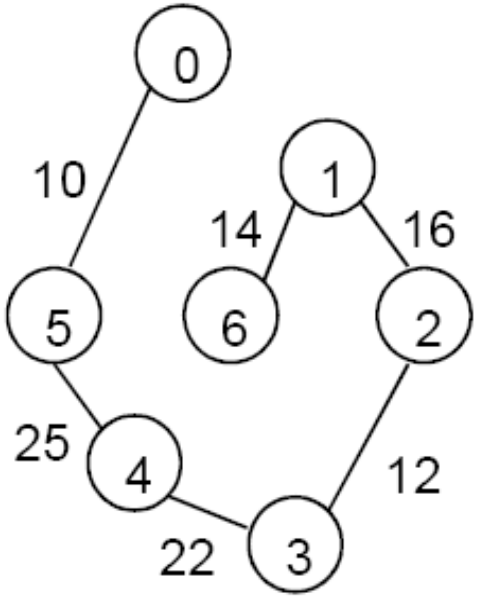

0
10
1
5
6
2
25
4
12
22
3

(d)


0
10
1
16
5
6
2
25
4
12
22
3

(e)





(f)

$O(n^2)$이거나

그보다 빠른 구현

Activity On Vertex network

- Activity on vertex (AOV) network
  - 방향그래프
  - 정점은 작업(tasks 또는 activities)을 나타낸다
  - 간선은 작업 사이의 선행관계를 나타낸다

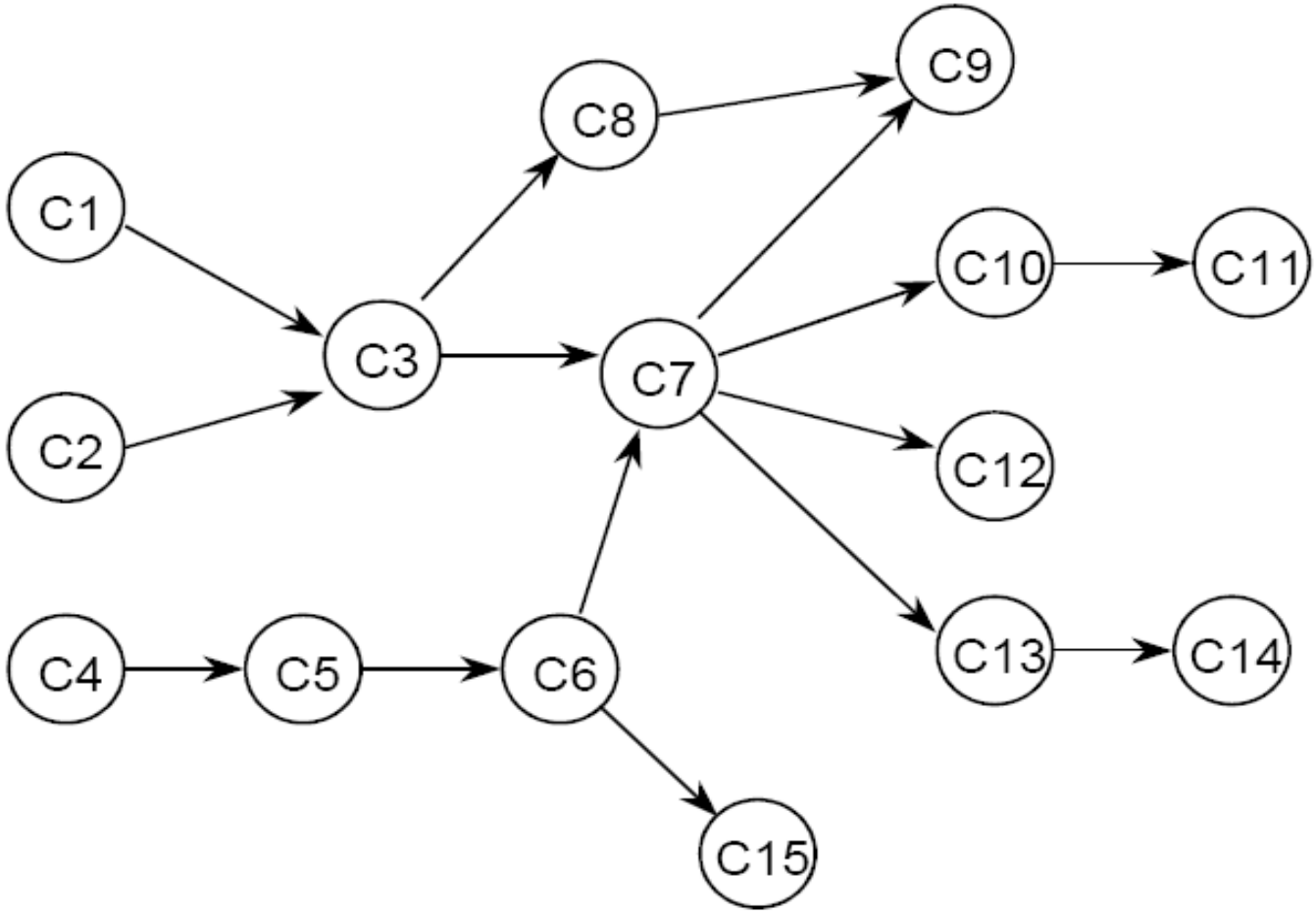
- AOV network G에서 정점 i로부터 정점 j로 방향경로가 존재하면, 정점 i는 정점 j의 선행자(predecessor)라 한다
- $<i,j> \in E(G)$이면, 정점 i는 정점 j의 직속선행자(immediate predecessor)이다
- 정점 i는 정점 j의 선행자이면, j는 i의 후속자(successor)이다

Activity Networks (AOV)

course number	course name	prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

courses needed for a computer science degree at a hypothetical university

AOV network



AOV network representing courses as vertices and edges as prerequisites

Topological order (위상순서)

- 위상순서(Topological order) :
  - 그래프의 정점들의 선형 순서
  - 임의의 두 정점 i, j에 대해, 네트워크에서 i가 j의 선행자이면 위상순서에서도 i가 j앞에 있는 선형순서이다

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

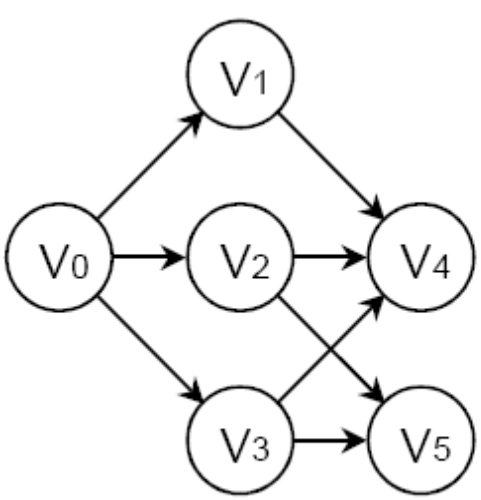
Topological sort (위상정렬)

1) 선행자를 가지지 않는 정점을 구하여 순서에 추가

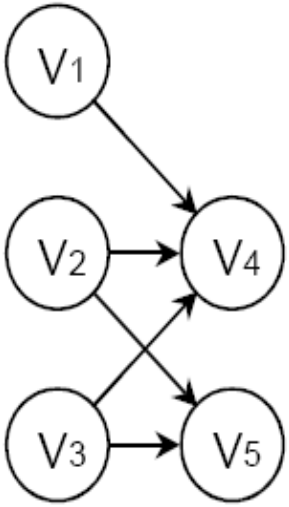
2) 그 정점과 그것으로부터 나오는 모든 간선들을 네트워크에서 삭제한다

- 다음이 만족할 때까지 위의 1,2과정을 반복
  - 모든 정점들이 나열되었거나
  - 모든 남아있는 정점들이 선행자를 가진다 -> 네트워크가 사이클을 가진다

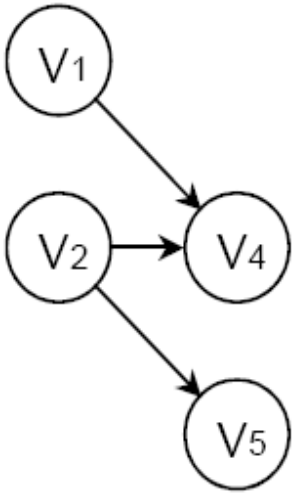
Example



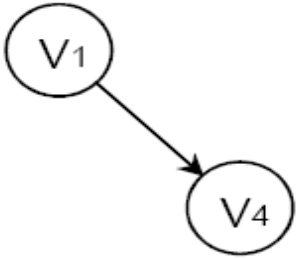
(a) initial



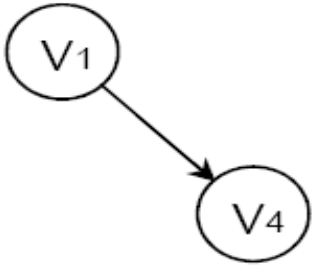
(b) V_0



(c) V_3



(d) V_2



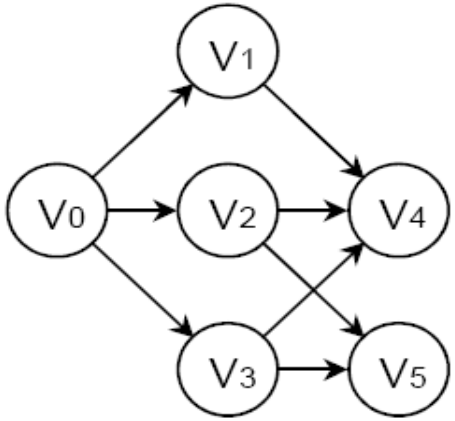
(e) V_5

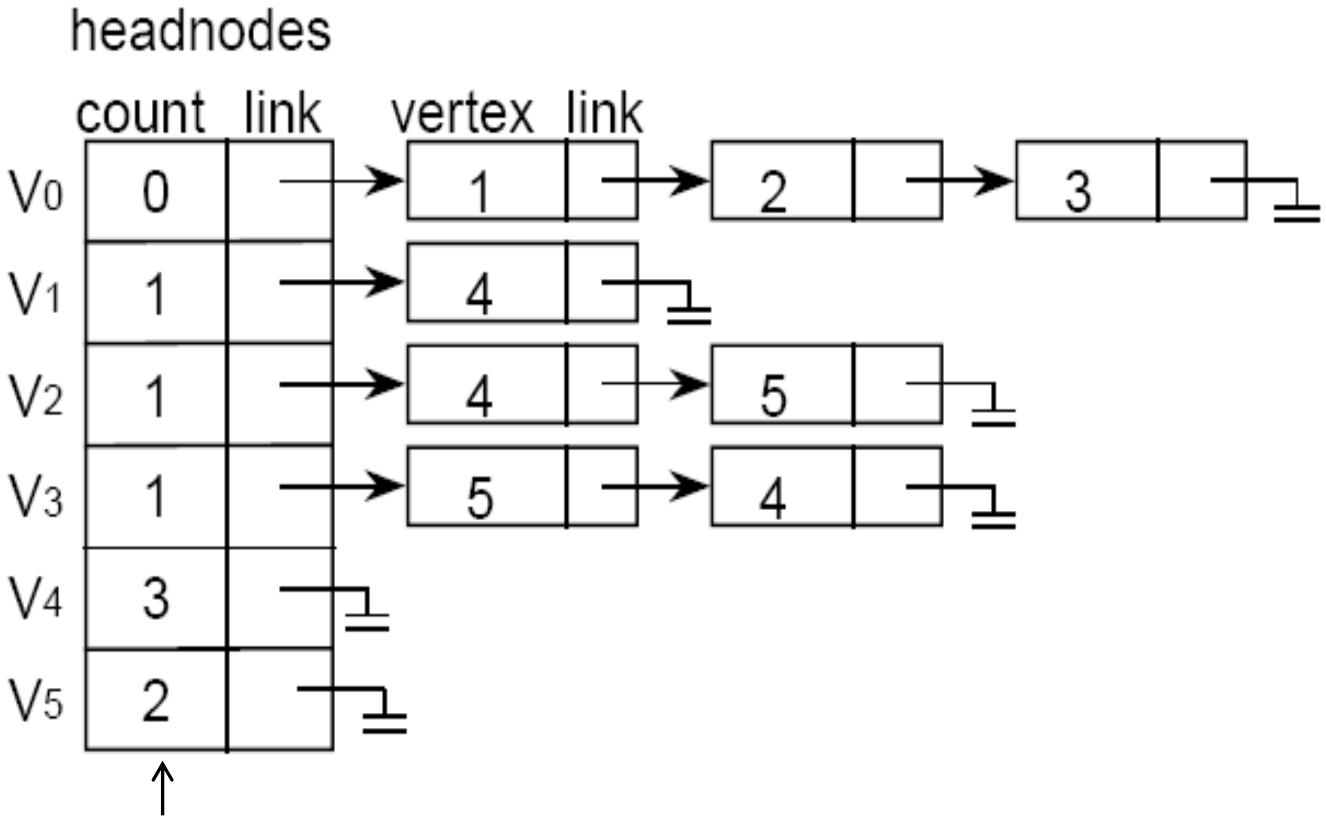
(f) V_1

(g) V_4

Topological sort

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





in-degree of the vertex

Implementations

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

public class adjlist {
        int count;      /* in-degree of the vertex */
        node link;
};

adjlist a[MAX_VERTICES];
```

```
public void topsort(adjlist [ ] a, int n){
    int i, j, k, top=-1;
    node ptr;
    for (i=0; i<n; i++){
        if (a[i].count == 0){
            a[i].count = top;   top = i;
        }   /* utilize a[ ].count as the space for a stack */
    }
    for (i=0; i<n; i++)
        if (top == -1){
            System.out.println("Network has a cycle\n");
             exit(1);
        } else {
              /* in the next page */
        }
}
```

```
j=top;     /*unstack a vertex  */
 top = a[top].count;
 System.out.println("vertex"+j);
 for (ptr=a[j].link; ptr; ptr=ptr->link){
 /* decrease the count of the successor vertices of j */
     k = ptr->vertex;
     a[k].count--;
     if (a[k].count==0) {
     /* add vertex k to the stack */
          a[k].count = top;
          top = k;
     }
}
```

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

d_i : out-degree of vertex i