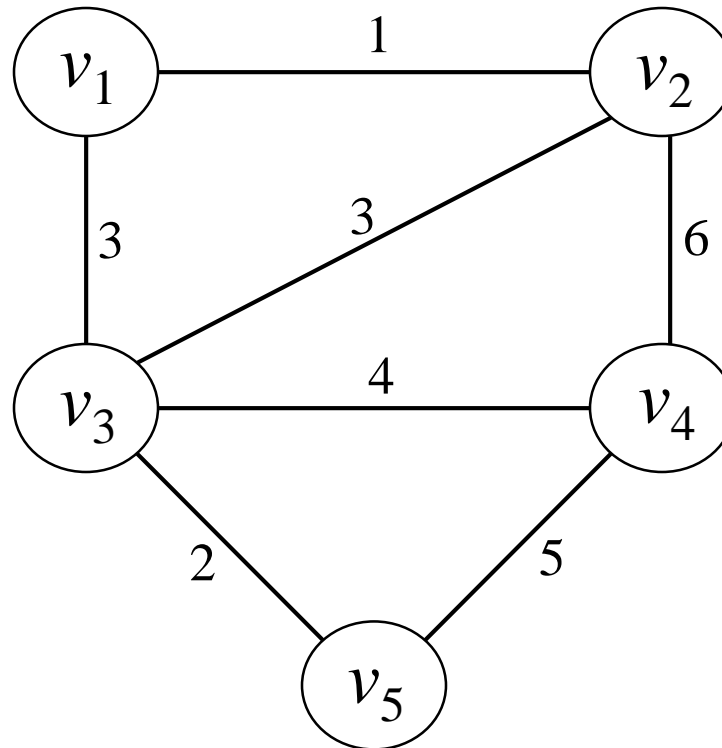


Graph algorithms II

1. Kruskal's Minimum Spanning Tree Algorithm
2. Floyd's All-Pairs Shortest Path Algorithm
3. Prim's Minimum Spanning Tree Algorithm
4. Dijkstra's Single Source Shortest Path Algorithm

연결된 가중치 비방향그래프



Minimum Spanning Tree

□ Spanning Tree (신장트리)

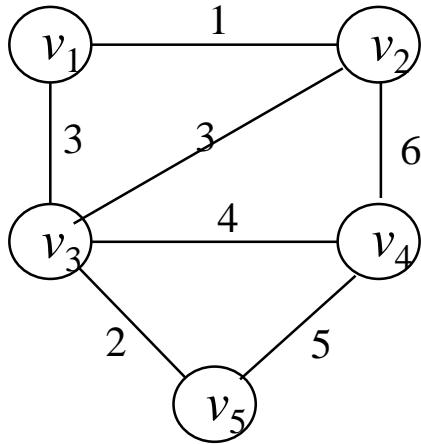
- A connected subgraph that contains all the vertices in G and is a tree
- 연결된 비방향성 그래프 G 에서 순환경로를 제거하면서 연결된 부분그래프가 되도록 이음선을 제거

□ Minimum spanning tree (최소비용 신장트리)

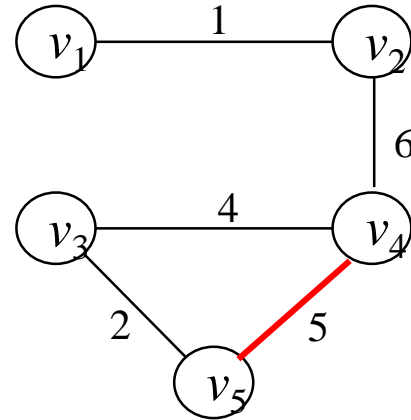
- A spanning tree with minimum weight in G
- 최소 비용의 연결된 부분그래프는 반드시 트리가 되어야 한다. 왜냐하면, 만약 트리가 아니라면, 분명히 순환경로(cycle)가 있을 것이고, 그렇게 되면 순환경로 상의 한 이음선을 제거하면 더 작은 비용의 연결된 부분그래프를 얻을 수 있기 때문이다.
- 관찰1: 모든 신장trie가 최소비용 신장trie는 아니다.
- 관찰2: 최소비용 신장trie는 유일하지 않을 수도 있다.

Minimum Spanning Tree

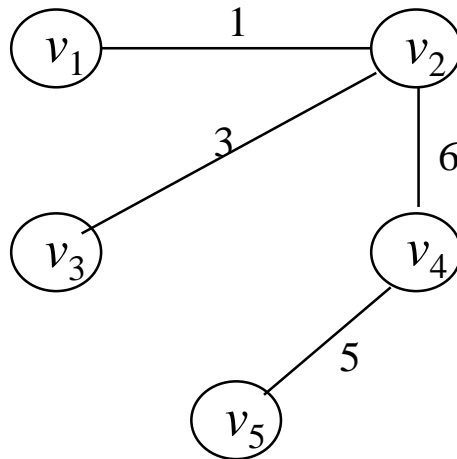
(a) A connected, weighted, undirected graph G



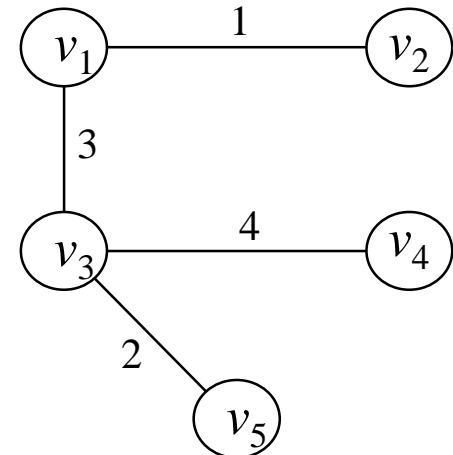
(b) If (v_4, v_5) were removed from this subgraph, the graph would remain connected.



(c) A spanning tree for G



(d) A minimum spanning tree for G



Minimum Spanning Tree

□ 최소비용신장트리의 적용 예

- 도로 건설 (road construction)
 - 도시들을 모두 연결하면서 도로의 길이가 최소가 되도록 하는 문제
- 통신 (telecommunications)
 - 전화선의 길이가 최소가 되도록 전화 케이블 망을 구성하는 문제
- 배관 (plumbing)
 - 파이프의 총 길이가 최소가 되도록 연결하는 문제

Minimum Spanning Tree

❏ Brute-force method

- 알고리즘

- 모든 신장트리를 다 고려해 보고,
그 중에서 최소비용이 드는 것을 고른다.

- 분석

- 이는 최악의 경우, 지수보다도 나쁘다.
- Complete graph의 신장트리는 $\Theta(n^{n-2})$ 개 존재함이 알려져 있다.

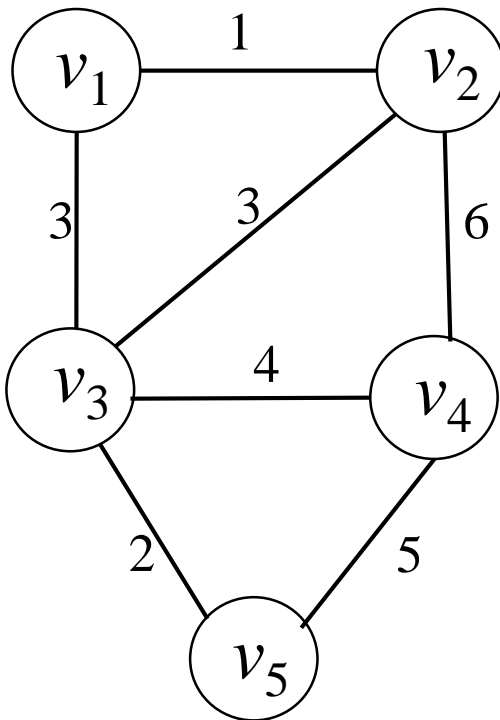
Kruskal's Algorithm

□ High-level Algorithm

```
 $F := \Phi;$  // initialize set of edges to empty  
create disjoint subsets of  $V$ , one for each  
vertex and containing only that vertex;  
sort the edges in  $E$  in nondecreasing order;  
While (the instance is not solved) {  
    select next edge; // selection procedure  
    if (the edge connects 2 vertices  
        in disjoint subsets) { // feasibility check  
        merge the subsets;  
        add the edge to  $F$ ;  
    }  
    if (all the subsets are merged) // solution check  
        the instance is solved;  
}
```

Kruskal's Algorithm

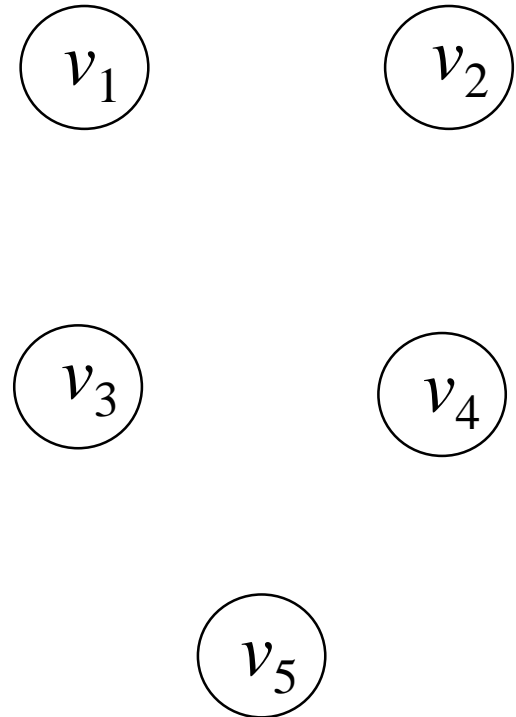
Determining a MST



1. Edges are sorted
by weight

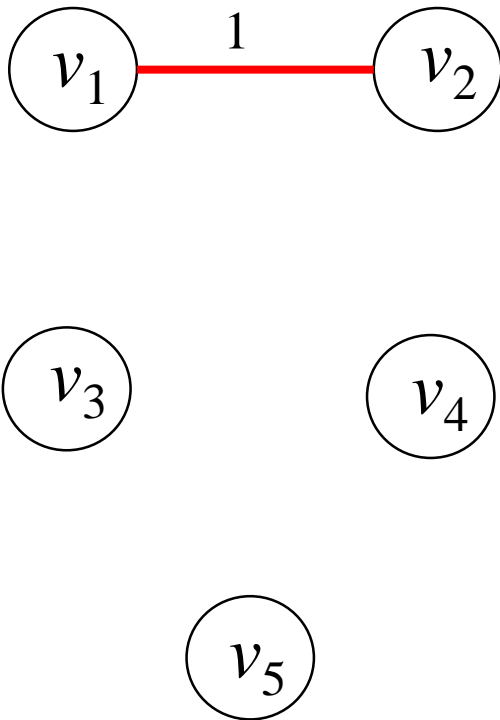
(v_1, v_2)	1
(v_3, v_5)	2
(v_1, v_3)	3
(v_2, v_3)	3
(v_3, v_4)	4
(v_4, v_5)	5
(v_2, v_4)	6

2. Disjoint sets are created

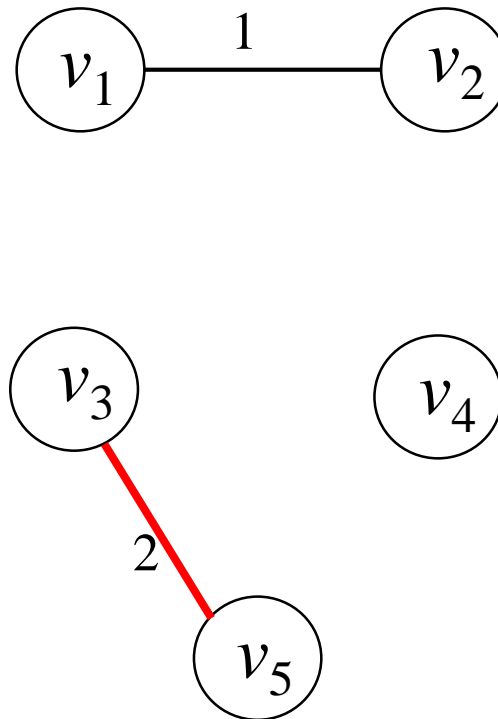


Kruskal's Algorithm

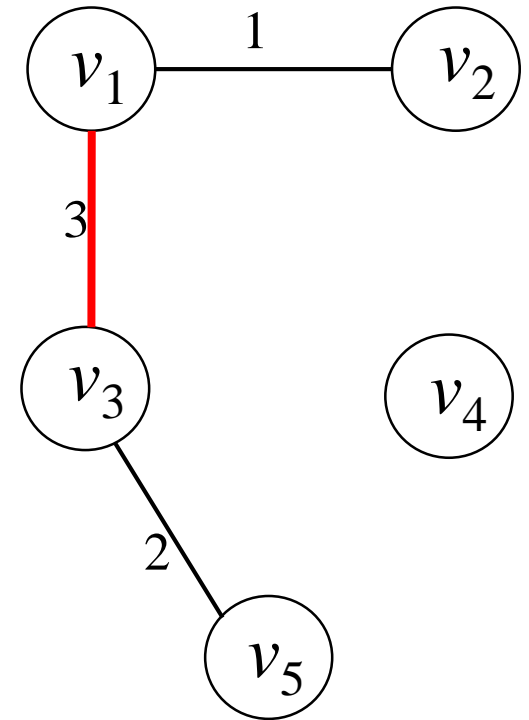
3. (v_1, v_2) is selected



4. (v_3, v_5) is selected

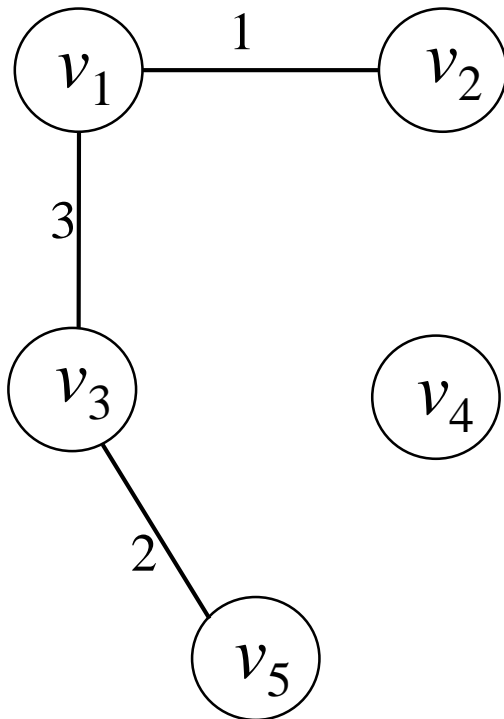


5. (v_1, v_3) is selected

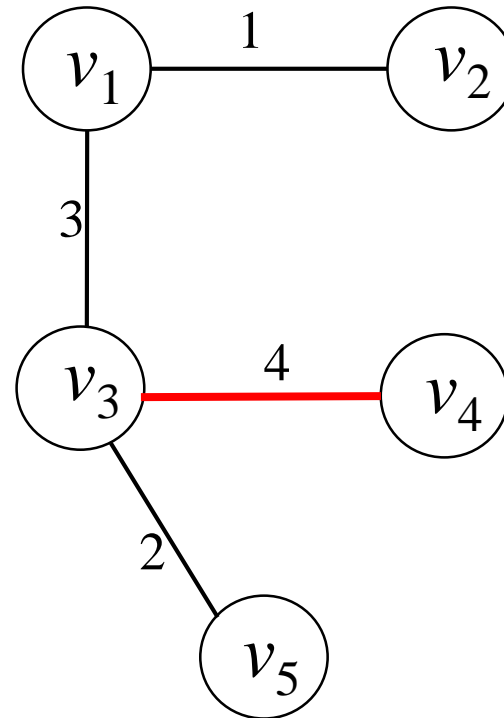


Kruskal's Algorithm

6. (v_2, v_3) is selected



7. (v_3, v_4) is selected



Kruskal's Algorithm

□ 서로소 집합 추상 데이터 타입 (disjoint set abstract data type)

index i ;

set_pointer p, q ;

- $\text{initial}(n)$: n 개의 서로소 부분집합을 초기화
(하나의 부분집합에 1에서 n 사이의 인덱스가 정확히 하나 포함됨)
- $p = \text{find}(i)$: 인덱스 i 가 포함된 집합의 포인터 p 를 넘겨줌
- $\text{merge}(p, q)$: 두 개의 집합을 가리키는 p 와 q 를 합병
- $\text{equal}(p, q)$: p 와 q 가 같은 집합을 가리키면 **true**를 넘겨줌

Kruskal's Algorithm

```
void kruskal(int n, int m, set_of_edges E, set_of_edges& F) {  
    index i, j;  
    set_pointer p, q;  
    edge e;  
  
    Sort the m edges in E by weight in nondecreasing order;  
    F =  $\Phi$ ;  
    initial(n);  
  
    while (number of edges in F is less than n-1) {  
        e = edges with least weight not yet considered;  
        i, j = indices of vertices connected by e;  
        p = find(i);  
        q = find(j);  
        if (!equal(p,q)) {  
            merge(p,q);  
            add e to F;  
        }  
    }  
}
```

Kruskal's Algorithm

□ Worst-Case Time-Complexity Analysis

- 단위연산: 비교문
- 입력크기: 정점의 수 n 과 이음선의 수 m
 1. 이음선 들을 정렬하는데 걸리는 시간: $\Theta(m \lg m)$
 2. 반복문 안에서 걸리는 시간: 루프를 m 번 수행한다. 서로소인 집합 자료구조(disjoint set data structure)를 사용하여 구현하고, find, equal, merge 같은 동작을 호출하는 횟수가 상수이면, m 개의 이음선 반복에 대한 시간복잡도는 $\Theta(m \lg n)$ 이다.
 3. n 개의 서로소인 집합(disjoint set)을 초기화하는데 걸리는 시간: $\Theta(n)$
- 그런데 여기서 $m \geq n - 1$ 이기 때문에, 위의 1과 2는 3을 지배하게 되므로, $W(m, n) = \Theta(m \lg m)$ 가 된다.
- 그러나, 최악의 경우에는 모든 정점이 다른 모든 정점과 연결이 될 수 있기 때문에, $m = \frac{n(n-1)}{2} \in \Theta(n^2)$ 가 된다. 그러므로, 최악의 경우의 시간복잡도는
$$W(m, n) \in \Theta(n^2 \lg n^2) = \Theta(2n^2 \lg n) = \Theta(n^2 \lg n)$$
- 최적여부의 검증(Optimality Proof)
 - Prim의 알고리즘의 경우와 비슷함. (교재 참조)

All-Pairs Shortest Path Problem

- 가중치가 있는 방향성 그래프에서 모든 정점에서 다른 모든 정점으로 가는 최단경로 구하는 문제

- 그래프에서 최단경로의 길이의 표현:

$$D^{(k)}[i][j] = v_i \text{에서 } v_k \text{까지의 정점들 만을 통해서 } v_i \text{에서 } v_j \text{로 가는 최단경로의 길이}$$

- 그래프의 인접 행렬(adjacent matrix) 표현: W

$$W[i][j] = \begin{cases} \text{이음선의 가중치} & v_i \text{에서 } v_j \text{로의 이음선이 있다면} \\ \infty & v_i \text{에서 } v_j \text{로의 이음선이 없다면} \\ 0 & i = j \text{ 이면} \end{cases}$$

동적계획식 설계전략 - 자료구조

□ 보기:

- W : 그래프의 인접행렬식 표현
- D : 각 정점들 사이의 최단 거리

$W[i][j]$	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

$D[i][j]$	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

여기서, $0 \leq k \leq 5$ 일 때, $D^{(k)}[2][5]$ 를 구해보자.

- $D^{(0)} = W$ 이고, $D^{(n)} = D$ 임은 분명하다. 따라서 D 를 구하기 위해서는 $D^{(0)}$ 를 가지고 $D^{(n)}$ 을 구할 수 있는 방법을 고안해 내어야 한다.

동적계획식 설계절차

1. Establish a recursive property

- $D^{(k-1)}$ 을 가지고 $D^{(k)}$ 를 계산할 수 있는 재귀 관계식을 정립

$$D^{(k)}[i][j] = \text{minimum}(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

경우1

경우2

경우 1: $\{v_1, v_2, \dots, v_k\}$ 의 정점들 만을 통해서 v_i 에서 v_j 로 가는 최단경로가 v_k 를 거치지 않는 경우.

$$\text{보기: } D^{(5)}[1][3] = D^{(4)}[1][3] = 3$$

경우 2: $\{v_1, v_2, \dots, v_k\}$ 의 정점들 만을 통해서 v_i 에서 v_j 로 가는 최단경로가 v_k 를 거치는 경우.

$$\text{보기: } D^{(2)}[5][3] = D^{(1)}[5][2] + D^{(1)}[2][3] = 4 + 3 = 7$$

$$\text{보기: } D^{(2)}[5][4]$$

2. 상향식으로 $k = 1$ 부터 n 까지 다음과 같이 이 과정을 반복하여 해를 구한다.

$$D^{(0)}, D^{(1)}, \dots, D^{(n)}$$

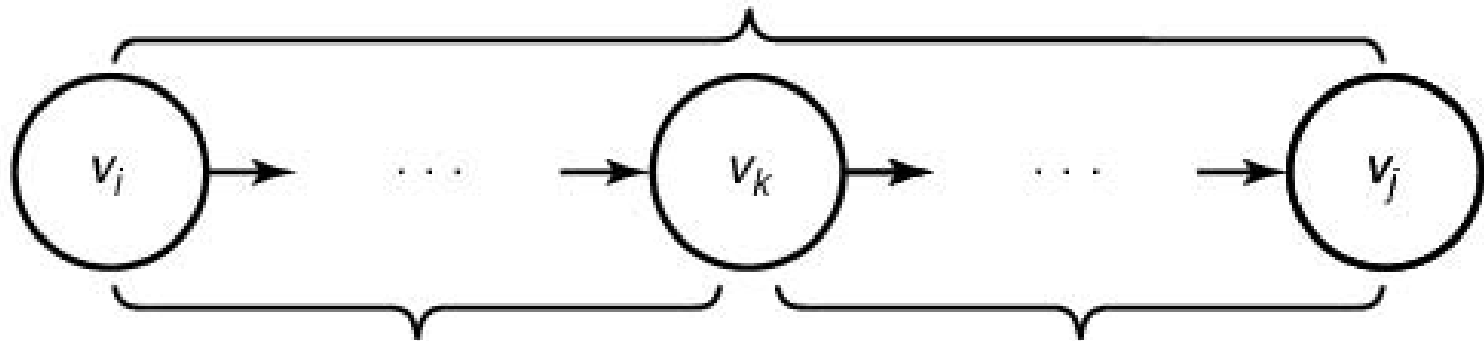
동적계획식 설계절차

$$D^{(k)}[i][j] = \text{minimum}(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$$

경우1

경우2

A shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \dots, v_k\}$



A shortest path from v_i to v_k using only vertices in $\{v_1, v_2, \dots, v_k\}$

A shortest path from v_k to v_j using only vertices in $\{v_1, v_2, \dots, v_k\}$

Floyd's Algorithm

□ 문제

- 가중치 포함 그래프의 각 정점에서 다른 모든 정점까지의 최단거리를 계산하고, 각각의 최단경로를 구하라.

- 입력

- 가중치 포함 방향성 그래프 W 와 그래프 정점의 수 n

- 출력

- 최단경로의 길이가 포함된 배열 D , 그리고 다음을 만족하는 배열 P

$$P[i][j] = \begin{cases} v_i \text{에서 } v_j \text{ 까지 가는 최단경로의 중간에 놓여 있는 정점이 최소한} \\ \text{하나는 있는 경우} \rightarrow \text{그 놓여 있는 정점 중에서 가장 큰 인덱스} \\ \\ \text{최단경로의 중간에 놓여 있는 정점이 없는 경우} \rightarrow 0 \end{cases}$$

Floyd's Algorithm

□ 알고리즘:

```
void floyd(int n, const number W[][], number D[][], index P[][]) {  
    index i, j, k;  
    for(i=1; i <= n; i++)  
        for(j=1; j <= n; j++)  
            P[i][j] = 0;  
  
    D = W;  
    for(k=1; k<= n; k++)  
        for(i=1; i <= n; i++)  
            for(j=1; j<=n; j++)  
                if (D[i][k] + D[k][j] < D[i][j]) {  
                    P[i][j] = k;  
                    D[i][j] = D[i][k] + D[k][j];  
                }  
}
```

Floyd's Algorithm

- 앞의 예를 가지고 D 와 P 를 구해 보시오.

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

최단경로의 출력

- 문제: 최단경로 상에 놓여 있는 정점을 출력하라.
- 알고리즘:

```
void path(index q,r) {  
    if (P[q][r] != 0) {  
        path(q,P[q][r]);  
        cout << " v" << P[q][r];  
        path(P[q][r],r);  
    }  
}
```

- 위의 P를 가지고 path(5,3)을 구해 보시오.

```
path(5,3) = 4  
    path(5,4) = 1  
        path(5,1) = 0  
            v1  
                path(1,4) = 0  
                    v4  
                        path(4,3) = 0
```

결과: v1 v4.

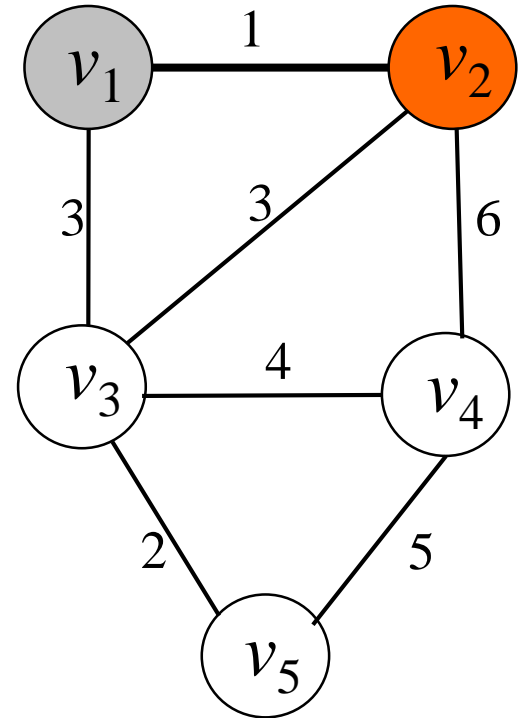
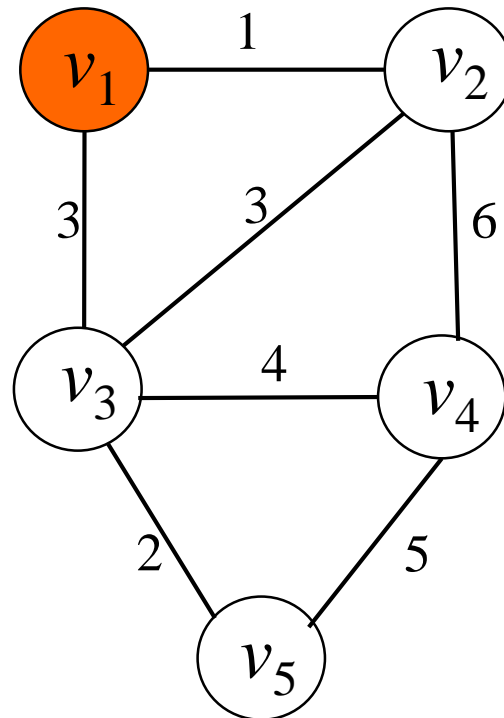
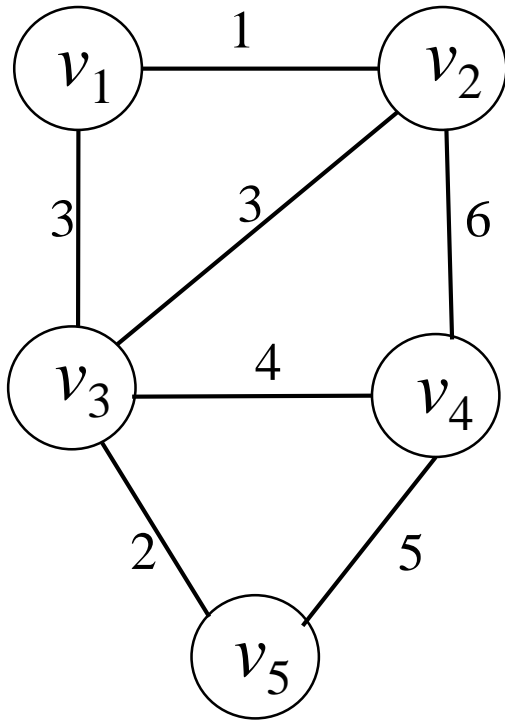
즉, v_5 에서 v_3 으로 가는 최단경로는 v_5, v_1, v_4, v_3 이다.

Prim's Algorithm

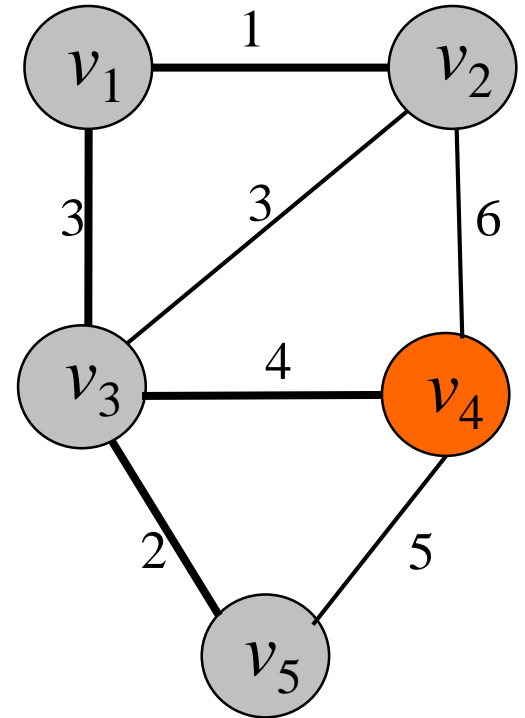
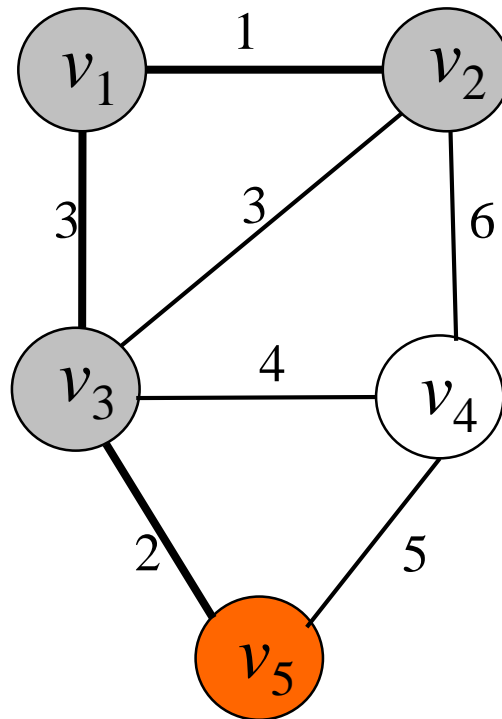
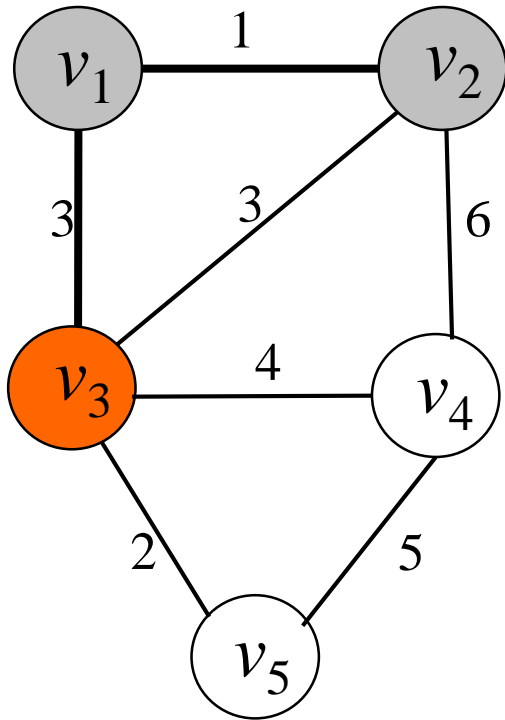
□ High-level Algorithm

```
 $F := \Phi;$  // initialize set of edges to empty  
 $Y := \{v_1\};$  // initialize set of vertices to  
// contain only the first one  
While (the instance is not solved) {  
    select a vertex in  $V - Y$  that is nearest to  $Y$ ; // selection procedure and  
// feasibility check  
    add the vertex to  $Y$ ;  
    add the edge to  $F$ ;  
    if ( $Y == V$ ) // solution check  
        the instance is solved;  
}
```

Prim's Algorithm



Prim's Algorithm



Prim's Algorithm

$$W[i][j] = \begin{cases} \text{이음선의 가중치} & v_i \text{에서 } v_j \text{로의 이음선이 있다면} \\ \infty & v_i \text{에서 } v_j \text{로의 이음선이 없다면} \\ 0 & i = j \text{ 이면} \end{cases}$$

- nearest[1..n]과 distance[1..n] 배열 유지

nearest[i] = Y에 속한 정점 중에서 v_i 에서
가장 가까운 정점의 인덱스

distance[1..n] = v_i 와 nearest[i]를 잇는
이음선의 가중치

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

Prim's Algorithm

```
void prim(int n, const number W[], set_of_edges& F) {
    index i, vnear; number min; edge e;
    index nearest[2..n]; number distance[2..n];

    F =  $\Phi$ ;
    for(i=2; i <= n; i++) {
        nearest[i] = 1;
        distance[i] = W[1][i];
    }

    repeat(n-1 times) {
        min = "infinite";
        for(i=2; i <= n; i++)
            if (0 <= distance[i] <= min) {
                min = distance[i];
                vnear = i;
            }
        e = edge connecting vertices indexed by vnear and nearest[vnear];
        add e to F;
        distance[vnear] = -1;
        for(i=2; i <= n; i++)
            if (W[i][vnear] < distance[i]) {
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
    }
}
```

// 초기화
// v_i 에서 가장 가까운 정점을 v_1 으로 초기화
// v_i 과 v_1 을 잇는 이음선의 가중치로 초기화

// n-1개의 정점을 Y에 추가한다

// 각 정점에 대해서
// distance[i]를 검사하여
// 가장 가까이 있는 vnear를
// 찾는다.

// 찾은 노드를 Y에 추가한다.

// Y에 없는 각 노드에 대해서
// distance[i]를 갱신한다.

Prim's Algorithm

□ Every-case Time Complexity Analysis

- 단위연산: repeat-루프 안에 있는 두 개의 for-루프 내부에 있는 명령문
- 입력크기: 마디의 개수, n
- 분석: repeat-루프가 $n-1$ 번 반복되므로
 - $T(n) = 2(n-1)(n-1) \in \Theta(n^2)$

Prim's Algorithm

□ 최적여부의 검증 (Optimality Proof)

Prim의 알고리즘이 찾아낸 신장트리가 최소비용(minimal)인지를 검증해야한다.

□ Definition 4.1

그래프 $G = (V, E)$ 가 주어져 있다. 임의의 이음선의 집합 $F \subseteq E$ 에 대해서 F 에 최소비용신장트리(MST)가 되도록 이음선을 추가할 수 있으면 F 는 유망하다(promising)라고 한다.

□ Lemma 4.1

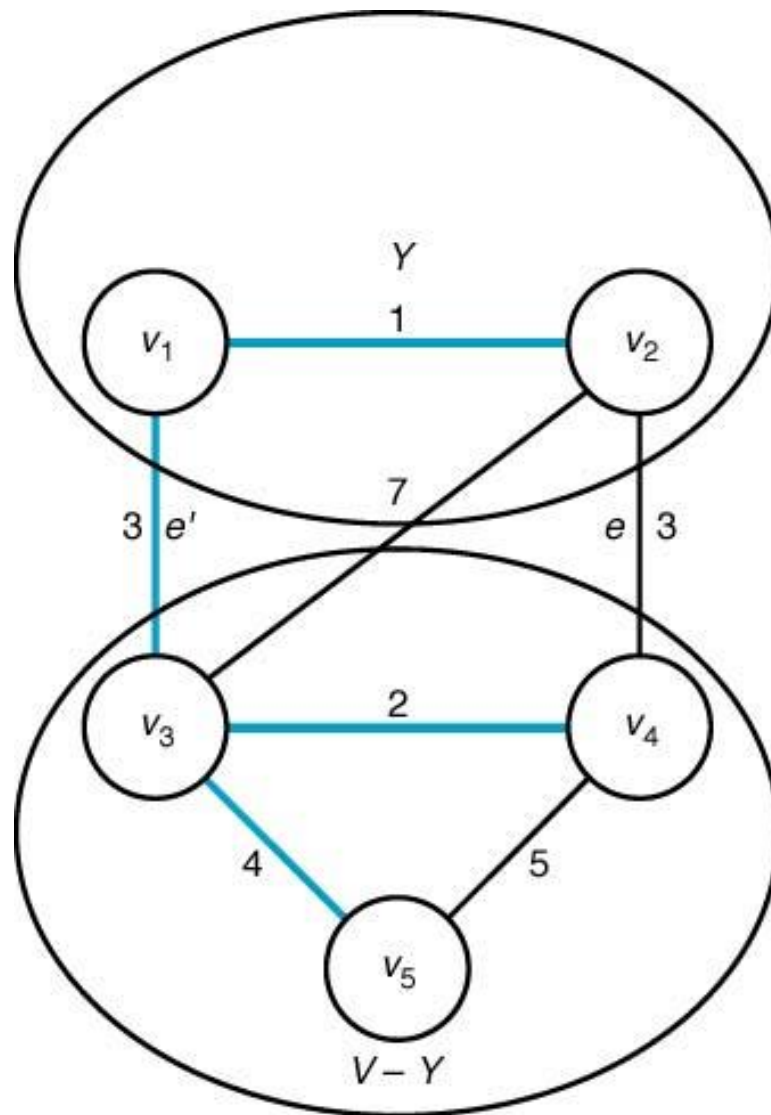
그래프 $G = (V, E)$ 가 주어져 있다. 이음선의 집합 F 가 유망하고, Y 는 F 안에 있는 이음선들에 의해 연결되어 있는 정점의 집합이라고 하자. 이때, Y 에 있는 정점과 $V - Y$ 에 있는 정점을 잇는 이음선 중에서 가중치가 가장 작은 이음선을 e 라고 하면, $F \cup \{e\}$ 는 유망하다.

Prim's Algorithm

Lemma 4.1의 증명

- F 가 유망하기 때문에 $F \subseteq F'$ 이면서 (V, F) 가 MST가 되는 이음선의 집합 F' 가 반드시 존재한다.
- 만일 $e \in F'$ 라면, $F \cup \{e\} \subseteq F'$ 가 되고, 따라서 $F \cup \{e\}$ 도 유망하다.
- 이제 $e \notin F'$ 라고 가정하자. $e=uv$ 라 하고 $u \in Y$, $v \in V - Y$ 라 하자.
 - F' 은 신장트리이기 때문에 u 와 v 간에 F' 의 이음선만을 이용한 단일경로 γ 를 포함한다. 따라서 $F' \cup \{e\}$ 는 u 와 v 를 연결하는 두 가지 경로(e 와 γ)를 갖게 된다.
 - 경로 γ 는 Y 에 있는 정점 u 에서 $V-Y$ 에 있는 정점 v 로 가는 경로이므로 V 에서 $V-Y$ 로 건너는 이음선 $e' = u'v' \in F'$ 을 포함하고 있다.
 - 이제 $F'' = F' \cup \{e\} - \{e'\}$ (즉 $F' \cup \{e\}$ 에서 e' 를 제거)를 생각해 보자. F'' 은 정점 u 와 v 를 연결하는 경로를 하나만 갖게 되고 신장트리가 된다. 그런데 e 는 Y 에 있는 정점과 $V - Y$ 에 있는 정점을 연결하는 이음선 중 가중치가 최소인 이음선이므로 e' 의 가중치보다 작거나 같다. 만일 작다면 F' 이 MST라는 가정에 모순이므로 같아야 한다. 즉 $F'' = F' \cup \{e\} - \{e'\}$ 은 또다른 MST이다.
 - e' 는 F 안에 속할 수 없으므로(F 는 Y 의 정점만을 연결한 집합), $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$ 가 되고, 따라서 $F \cup \{e\}$ 는 유망하다.

Prim's Algorithm



Prim's Algorithm

- Theorem 4.1 (최적여부의 검증 (Optimality Proof))
Prim의 알고리즘은 항상 최소비용신장트리를 만들어낸다.

증명: (수학적귀납법)

매번 반복이 수행된 후에 집합 F 가 유망하다는 것을 보이면 된다.

- 출발점: 공집합은 당연히 유망하다.
- Prim의 알고리즘을 k 번 수행하며 만든 이음선의 집합 F 가 유망하다고 가정하자.
- $k+1$ 번째 선정된 이음선을 e 라 할 때 집합 $F \cup \{e\}$ 가 유망하다는 것을 보이면 된다. 그런데 Lemma 4.1에 의하여 $F \cup \{e\}$ 은 유망하다. 이음선 e 는 Y 에 있는 어떤 정점을 $V - Y$ 에 있는 어떤 정점으로 잇는 이음선 중에서 최소의 가중치를 가지고 있기 때문이다.

Minimum Spanning Tree

- 두 알고리즘 시간 복잡도 비교

연결된 그래프에서의 m 은 $n-1 \leq m \leq \frac{n(n-1)}{2}$ 의 범위를 갖는다.

	$W(m,n)$	sparse graph $m=\Theta(n)$	dense graph $m=\Theta(n^2)$
Prim	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Kruskal	$\Theta(m \lg m)$	$\Theta(n \lg n)$	$\Theta(n^2 \lg n)$

- 알고리즘의 시간복잡도는 그 알고리즘을 구현하는데 사용하는 자료구조에 좌우되는 경우도 있다.

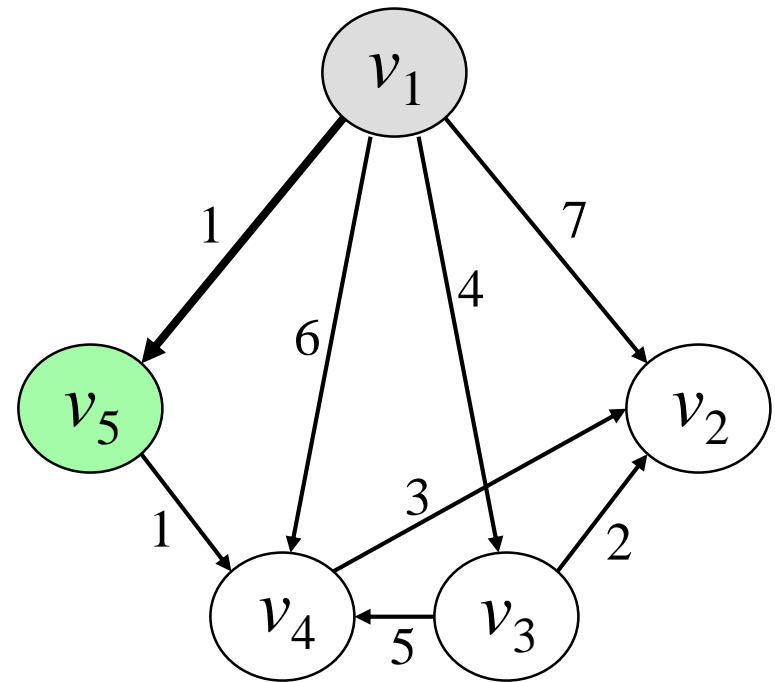
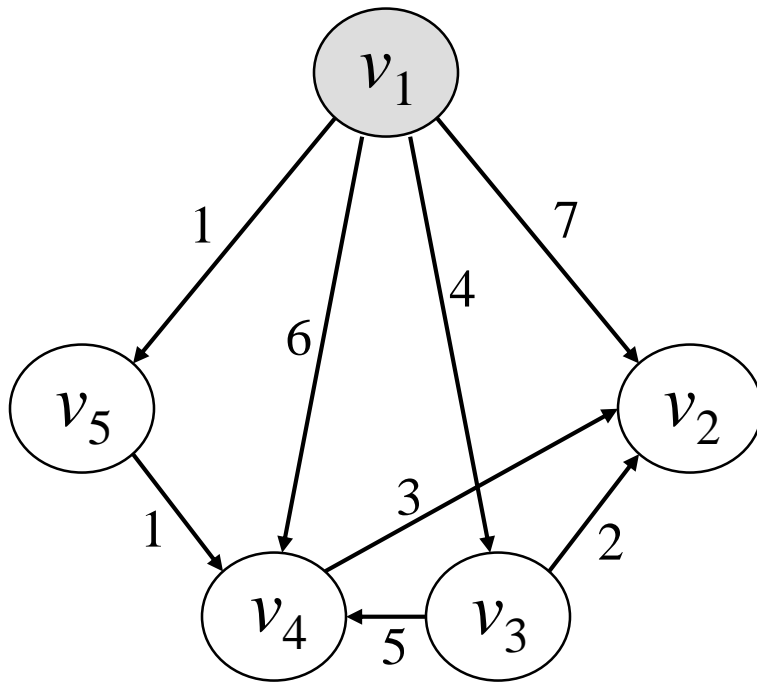
Prim 의 알고리즘	$W(m,n)$	sparse graph $m=\Theta(n)$	dense graph $m=\Theta(n^2)$
Heap	$\Theta(m \lg n)$	$\Theta(n \lg n)$	$\Theta(n^2 \lg n)$
Fibonacci heap	$\Theta(m + n \lg n)$	$\Theta(n \lg n)$	$\Theta(n^2)$

Dijkstra's Algorithm

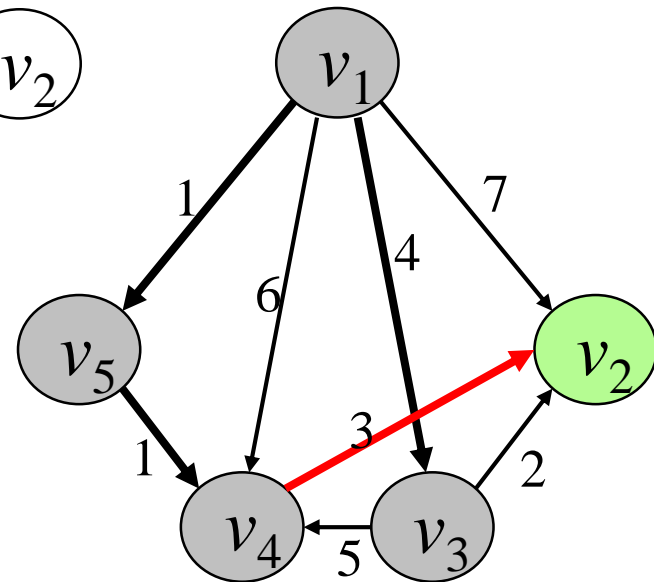
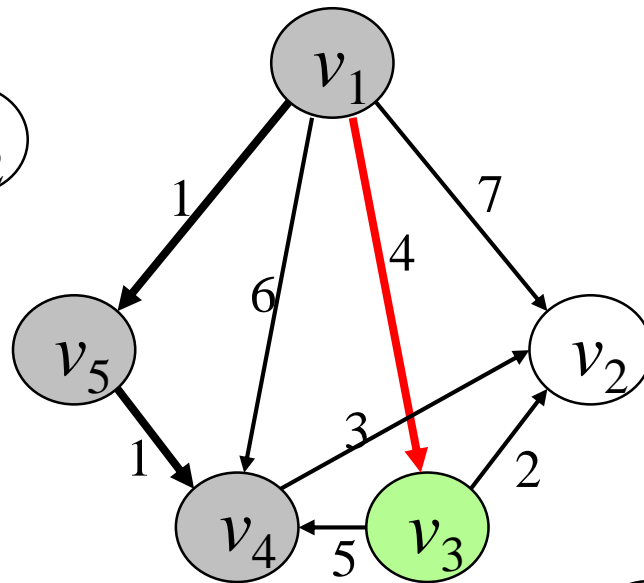
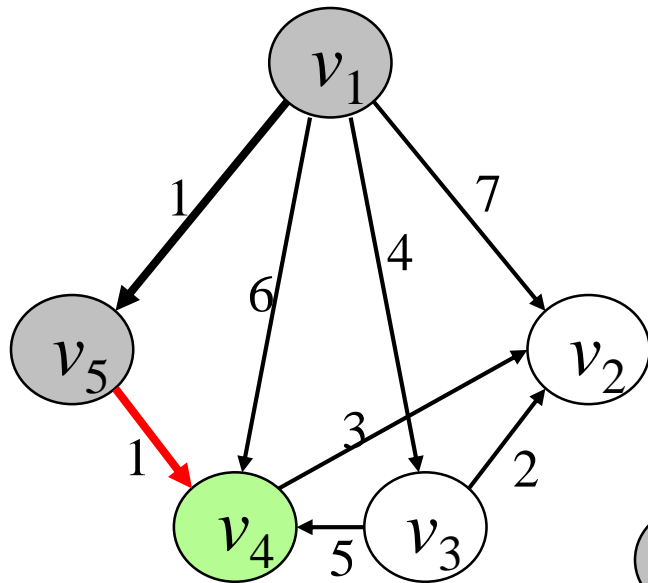
- 가중치가 있는 방향성 그래프에서 한 특정 정점에서 다른 모든 정점으로 가는 최단경로 구하는 문제
- 시작점 v_1
- 알고리즘

```
 $F := 0;$   
 $Y := \{v_1\};$   
While (the instance is not solved)  
    select a vertex  $v$  from  $V - Y$ , that has a shortest path // selection procedure  
    from  $v_1$ , using only vertices in  $Y$  as intermediate; // and feasibility check  
  
    add the new vertex  $v$  to  $Y$ ;  
    add the edge (on the shortest) that touches  $v$  to  $F$ ;  
  
    if ( $Y == V$ ) // solution check  
        the instance is solved;
```

Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm

□ Define nearest & length

- $\text{nearest}[i]$ = index of vertex v in Y such that the edge $\langle v, v_i \rangle$ is the last edge on the current shortest path from v_1 to v_i using only vertices in Y as intermediates
- $\text{length}[i]$ = length of the current shortest path from v_1 to v_i using only vertices in Y as intermediates.
(Prim 알고리즘에서 $\text{distance}[i]$ 와 같은 역할)

Dijkstra's Algorithm

```
void dijkstra (int n, const number W[][], set_of_edges& F) {
    index i, vnear; edge e;
    index nearest[2..n]; number length[2..n];

    F =  $\Phi$ ;
    for(i=2; i <= n; i++) {           // For all vertices, initialize v1 to be the last
        nearest[i] = 1;                // vertex on the current shortest path from v1,
        length[i] = W[1][i];           // and initialize length of that path to be the
    }                                  // weight on the edge from v1.

    repeat(n-1 times) {                // Add all n-1 vertices to Y.
        min = "infinite";
        for(i=2; i <= n; i++)          // Check each vertex for having shortest path.
            if (0 <= length[i] <= min) {
                min = length[i];
                vnear = i;
            }
        e = edge from vertex indexed by nearest[vnear]
            to vertex indexed by vnear;
        add e to F;
        for(i=2; i <= n; i++)
            if (length[vnear] + W[vnear][i] < length[i]) {
                length[i] = length[vnear] + W[vnear][i];
                nearest[i] = vnear;      // For each vertex not in Y, update its shortest
            }                           // path. Add vertex indexed by vnear to Y.
        length[vnear] = -1;
    }
}
```

Dijkstra's Algorithm

□ 분석

- $T(n) = 2(n-1)^2 \in \Theta(n^2)$.

□ 최적여부의 검증(Optimality Proof)

- Prim의 알고리즘의 경우와 비슷함.