

# Chap 4. The Greedy Approach

1. Minimum Spanning Trees
2. Dijkstra's Algorithm for Single-Source Shortest Paths
3. Scheduling
5. The Greedy Approach Vs Dynamic Programming:  
The Knapsack Problem

# Introduction

- Want to solve optimization problems
  - Using dynamic programming and greedy approach
- Dynamic Programming
  - A recursive property is used to divide an instance into smaller instances
- Greedy approach
  - Arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment. That is, each choice is locally optimal.

# Introduction

- 탐욕적인 알고리즘(**Greedy algorithm**)은 결정을 해야 할 때마다 그 순간에 가장 좋다고 생각되는 것을 해답으로 선택함으로써 최종적인 해답에 도달한다.
- 그 순간의 선택은 그 당시(**local**)에는 최적이다. 그러나 최적이라고 생각했던 해답들을 모아서 최종적인(**global**)해답을 만들었다고 해서, 그 해답이 궁극적으로 최적이라는 보장이 없다.
- 따라서 탐욕적인 알고리즘은 항상 최적의 해답을 주는지를 반드시 검증해야 한다.

# 탐욕적인 알고리즘 설계 절차

## 1. Selection procedure (선택과정)

- 현재 상태에서 가장 좋으리라고 생각되는 (greedy criterion) 해답을 찾아서 해답모음 (solution set)에 포함시킨다.

## 2. Feasibility check (적정성 점검)

- 새로 얻은 해답모음이 적절한지를 결정한다.

## 3. Solution check (해답 점검)

- 새로 얻은 해답모음이 최적의 해인지를 결정한다.


# Example: Change Problem

- Problem: 동전의 개수가 최소가 되도록 거스름 돈을 주는 문제
- Greedy Algorithm






```
While (there are more coins and the instance is not solved)
    Grab the largest remaining coin;           // selection procedure
    If (adding the coin makes the change
        exceed the amount owed)               // feasibility check
        reject the coin;
    else
        add the coin to the change;
    If (the total value of the change equals   // solution check
        the amount owed)
        the instance is solved;
}
```

# Example: Change Problem

**Coins**



**Amount owed: 36 cents**

Step	Total Change
1. Grab quarter	
2. Grab first dime	
3. Reject second dime	
4. Reject nickel	
5. Grab penny	











# Example: Change Problem

- 최적의 해를 얻지 못하는 경우
  - 12 cent 짜리 동전을 새로 발행했다고 하자.
  - 이 알고리즘을 적용하여 거스름돈을 주면,  
항상 동전의 개수는 최소가 된다는 보장이 없다.
  - 보기: 거스름돈 액수 = 16 cent
    - 탐욕알고리즘의 결과:  $12 \text{ cent} \times 1\text{개} = 12 \text{ cent}$ ,  
 $1 \text{ cent} \times 4\text{개} = 4 \text{ cent}$
    - 동전의 개수 = 5개  $\Rightarrow$  최적(optimal)이 아님!
    - 최적의 해:  $10 \text{ cent} \times 1\text{개}$ ,  $5 \text{ cent} \times 1\text{개}$ ,  $1 \text{ cent} \times 1\text{개}$ 가 되어  
동전의 개수는 3개가 된다.

# Example: Change Problem



Amount owed: 16 cents

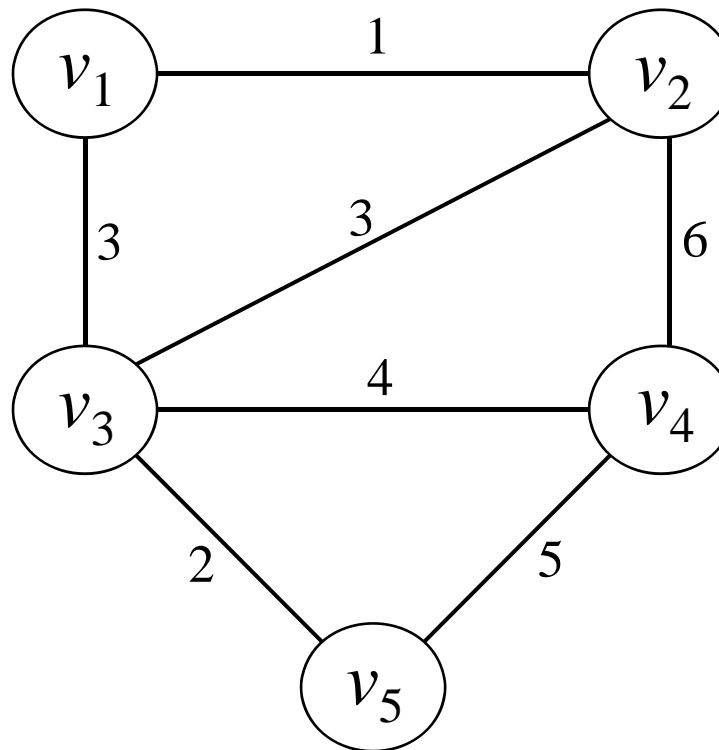
Step	Total Change				
1. Grab 12-cent coin					
2. Reject dime					
3. Reject nickel					
4. Grab four pennies					



# 그래프 용어

- 비방향성 그래프(undirected graph)  $G = (V, E)$ ,
  - $V$ 는 정점(vertex)의 집합
  - $E$ 는 이음선(edge)의 집합
- 경로(path)
- 연결된 그래프(connected graph) - 어떤 두 정점 사이에도 경로가 존재
- 부분그래프(subgraph)
- 가중치 포함 그래프(weighted graph)
- 순환경로(cycle)
- 순환적 그래프(cyclic graph), 비순환적 그래프(acyclic graph)
- 트리(tree) - 비순환적이며, 비방향성 그래프
- 뿌리 있는 트리(rooted tree) - 한 정점이 뿌리로 지정된 트리

## 예: 연결된 가중치 비방향그래프

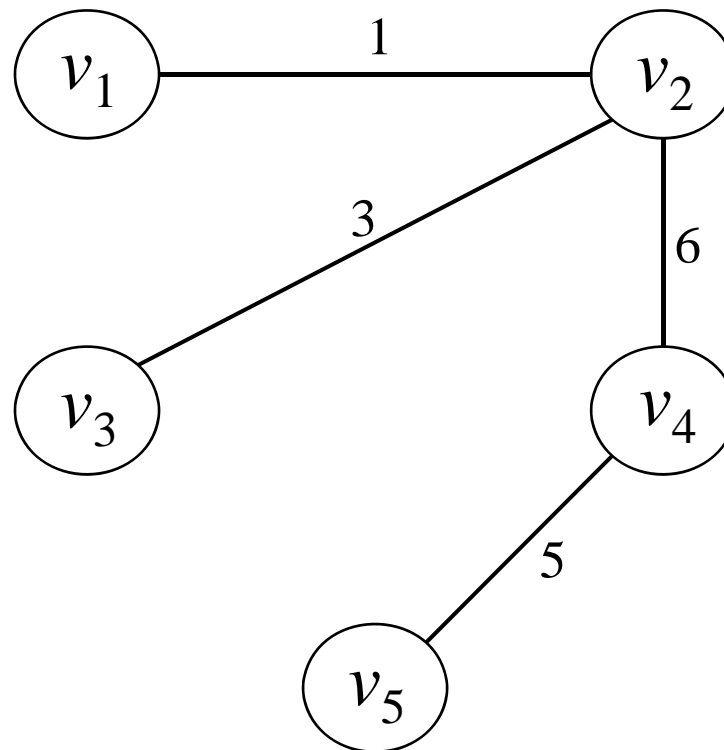


# Spanning Tree

## □ 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
- 따라서 신장트리는  $G$ 안에 있는 모든 정점을 다 포함하면서 트리가 되는 연결된 부분그래프

# Spanning Tree

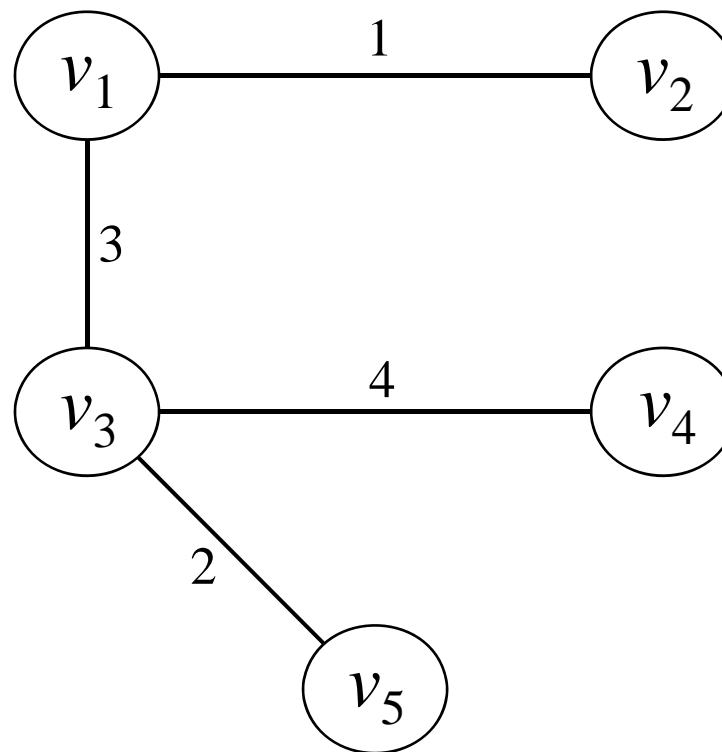


# Minimum Spanning Tree

## □ Minimum spanning tree (최소비용 신장트리)

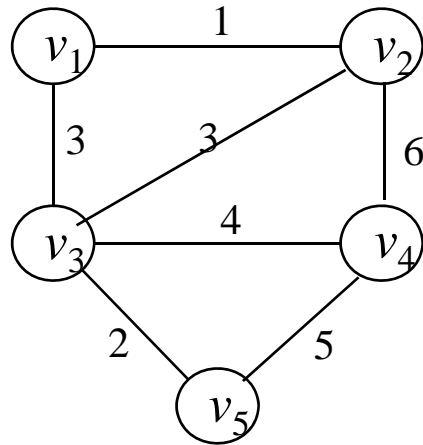
- A spanning tree with minimum weight
- 최소의 가중치를 가진 부분그래프는 반드시 트리가 되어야 한다.  
왜냐하면, 만약 트리가 아니라면, 분명히 순환경로(**cycle**)가 있을 것이고,  
그렇게 되면 순환경로 상의 한 이음선을 제거하면 더 작은 비용의  
신장트리가 되기 때문이다.
- 관찰: 모든 신장트리가 최소비용 신장트리는 아니다.

# Minimum Spanning Tree

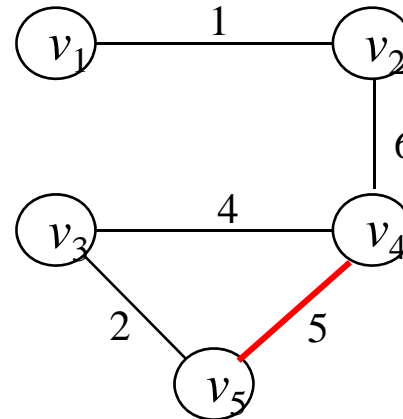


# Minimum Spanning Tree

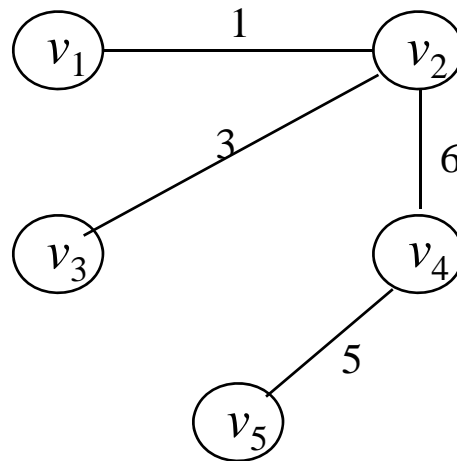
(a) A connected, weighted, undirected graph  $G$



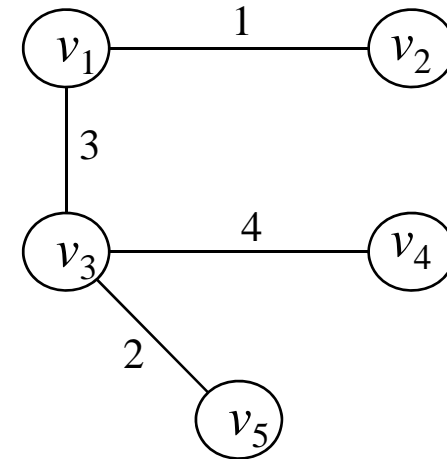
(b) If  $(v_4, v_5)$  were removed from this subgraph, the graph would remain connected.



(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.
  - 이유?

# Minimum Spanning Tree

## □ Greedy approach

- Problem: 비방향성 그래프  $G = (V, E)$ 가 주어졌을 때,  
 $F \subseteq E$ 를 만족하면서,  
 $(V, F)$ 가  $G$ 의 최소비용신장트리(MST)가 되는  $F$ 를 찾는 문제.

## ● Algorithm

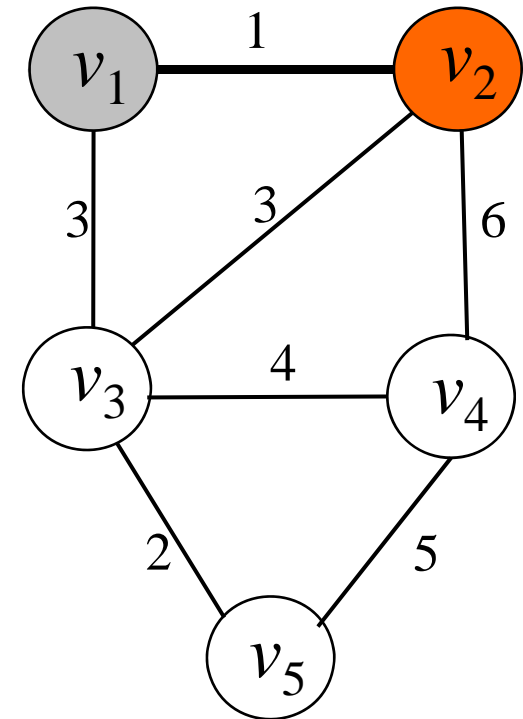
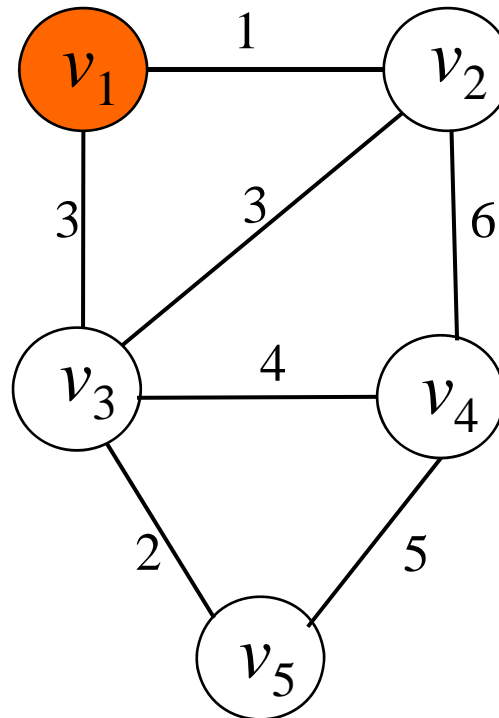
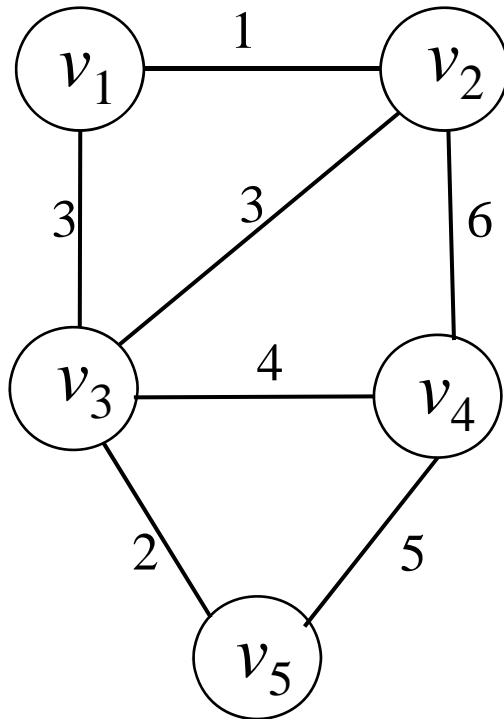
```
F =  $\Phi$ ;  
while (the instance is not solved) {  
    select an edge according to some locally // selection procedure  
    optimal consideration;  
  
    if (adding the edge to F dose not create a cycle)  
        add it; // feasibility procedure  
  
    if (T = (V, F) is a spanning tree) // solution check  
        the instance is solved;  
}
```

# Prim's Algorithm

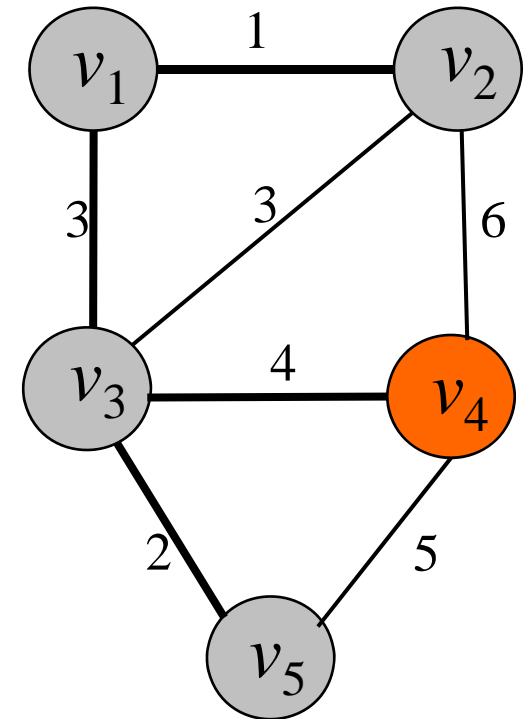
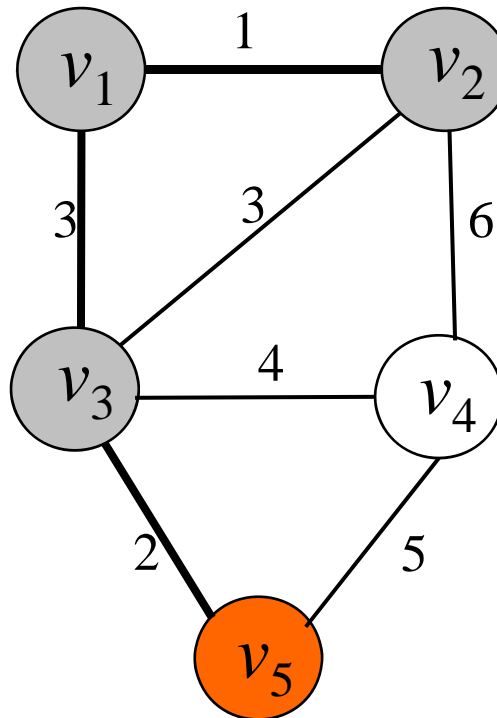
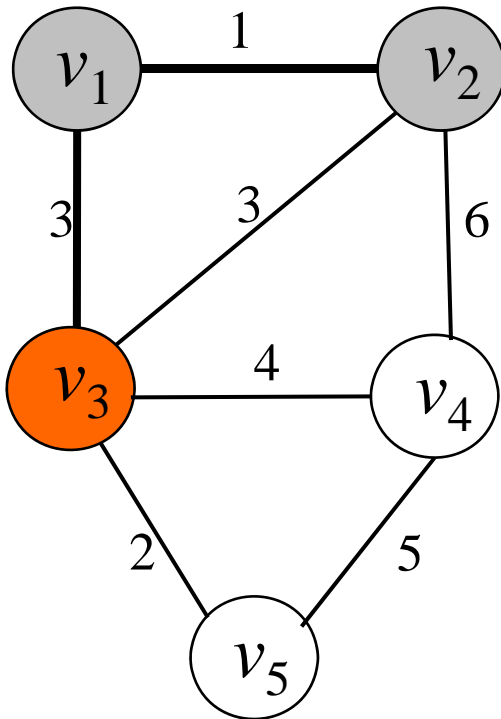
## □ High-level Algorithm

```
 $F := \Phi;$  // 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

- 그래프의 인접행렬식 표현

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

	1	2	3	4	5
1	0	1	3	$\infty$	$\infty$
2	1	0	3	6	$\infty$
3	3	3	0	4	2
4	$\infty$	6	4	0	5
5	$\infty$	$\infty$	2	5	0

# Prim's Algorithm

- 추가적으로  $\text{nearest}[1..n]$ 과  $\text{distance}[1..n]$  배열 유지

$\text{nearest}[i] = Y$ 에 속한 정점 중에서  $v_i$ 에서  
가장 가까운 정점의 인덱스

$\text{distance}[1..n] = v_i$ 와  $\text{nearest}[i]$ 를 잇는  
이음선의 가중치

# 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  
// vi에서 가장 가까운 정점을 v1으로 초기화  
// vi와 v1을 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 검증. 다시 말하면, Prim의 알고리즘이 최적(optimal)인지를 보여야 한다.

### ● Definition 4.1

비방향성 그래프  $G = (V, E)$ 가 주어지고,

만약  $E$ 의 부분집합  $F$ 에 MST가 되도록 이음선을 추가할 수 있으면,

$F$ 는 유망하다(promising)라고 한다.

### ● Lemma 4.1:

$G = (V, E)$ 는 연결되고, 가중치 포함 비방향성 그래프라고 하고,

$F$ 는  $E$ 의 유망한 부분집합이라고 하고,  $Y$ 는  $F$ 안에 있는 이음선 들에 의해서 연결이 되어 있는 정점의 집합이라고 하자.

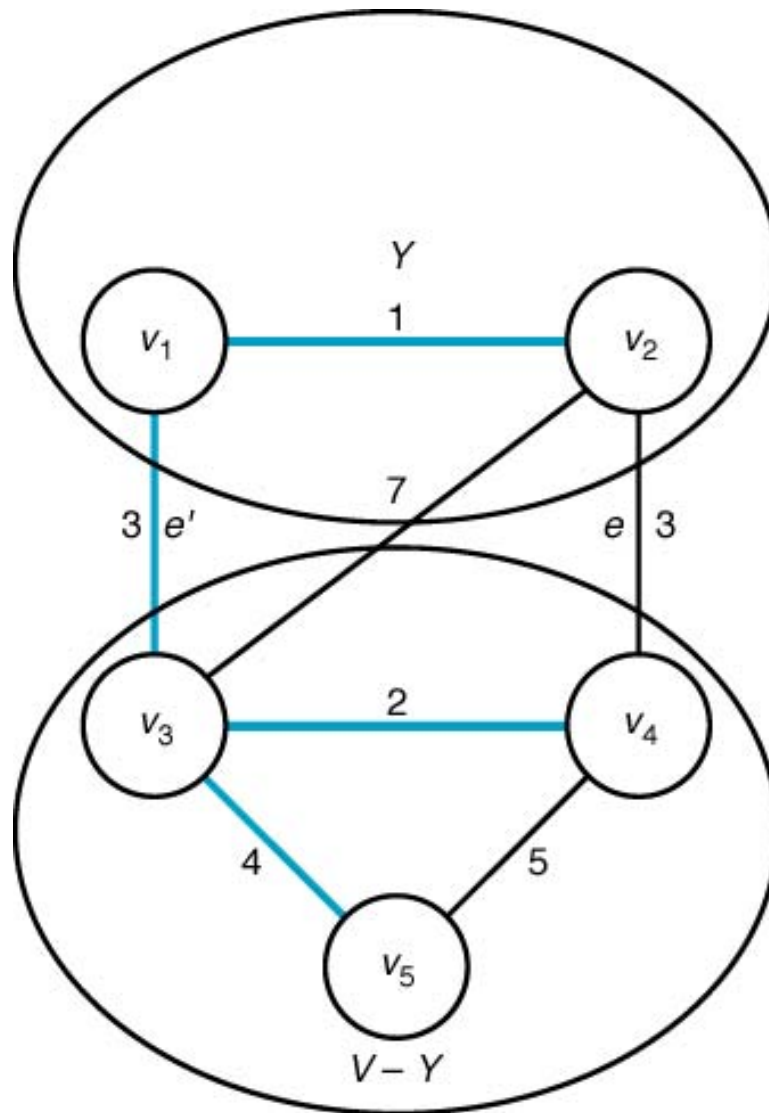
이때,  $Y$ 에 있는 어떤 정점과  $V - Y$ 에 있는 어떤 정점을 잇는 이음선 중에서 가중치가 가장 작은 이음선을  $e$ 라고 하면,  $F \cup \{e\}$ 는 유망하다.

# Prim's Algorithm

## □ Optimality Proof (Lemma 4.1)

- $F$ 가 유망하기 때문에  $F \subseteq F'$ 이면서  $(V, F')$ 가 최소비용신장트리(MST)가 되는 이음선의 집합  $F'$ 가 반드시 존재한다.
- 경우 1: 만일  $e \in F'$ 라면,  $F \cup \{e\} \subseteq F'$ 가 되고, 따라서  $F \cup \{e\}$ 도 유망하다.
- 경우 2: 만일  $e \notin F'$ 라면,  $(V, F')$ 는 신장트리이기 때문에,  $F' \cup \{e\}$ 는 반드시 순환 경로를 하나 포함하게 되고,  $e$ 는 반드시 그 순환 경로 가운데 한 이음선이 된다.
  - 그러면  $Y$ 에 있는 한 정점에서  $V - Y$ 에 있는 한 정점을 연결하는 어떤 다른 이음선  $e' \in F'$ 가 그 순환 경로 안에 반드시 존재하게 된다.
  - 여기서 만약  $F' \cup \{e\}$ 에서  $e'$ 를 제거하면, 그 순환 경로는 없어지게 되며, 다시 신장트리가 된다. 그런데  $e$ 는  $Y$ 에 있는 한 정점에서  $V - Y$ 에 있는 한 정점을 연결하는 최소의 가중치(weight)를 가진 이음선이기 때문에,  $e$ 의 가중치는 반드시  $e'$ 의 가중치 보다 작거나 같아야 한다. (실제로 반드시 같게 된다.)
  - 그러면  $F' \cup \{e\} - \{e'\}$ 는 최소비용신장트리(MST)이다.
  - 결론적으로  $e'$ 는  $F$ 안에 절대로 속할 수 없으므로 ( $F$ 안에 있는 이음선들은  $Y$ 안에 있는 정점들만을 연결함을 기억하라),  $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$ 가 되고, 따라서  $F \cup \{e\}$  유망하다.

# Prim's Algorithm



# Prim's Algorithm

## □ Theorem 4.1

- 정리: Prim의 알고리즘은 항상 최소비용신장트리를 만들어 낸다.

- 증명: (수학적귀납법)

매번 반복이 수행된 후에 집합  $F$ 가 유망하다는 것을 보이면 된다.

- ◆ 출발점: 공집합은 당연히 유망하다.
- ◆ 귀납가정: 어떤 주어진 반복이 이루어진 후, 그때까지 선정하였던 이음선의 집합인  $F$ 가 유망하다고 가정한다
- ◆ 귀납절차: 집합  $F \cup \{e\}$ 가 유망하다는 것을 보이면 된다.  
여기서  $e$ 는 다음 단계의 반복 수행 시 선정된 이음선 이다.  
그런데, 위의 보조정리 1에 의하여  $F \cup \{e\}$ 은 유망하다고 할 수 있다.  
왜냐하면 이음선  $e$ 는  $Y$ 에 있는 어떤 정점을  $V - Y$ 에 있는 어떤 정점으로 잇는 이음선 중에서 최소의 가중치를 가지고 있기 때문이다.

증명 끝.

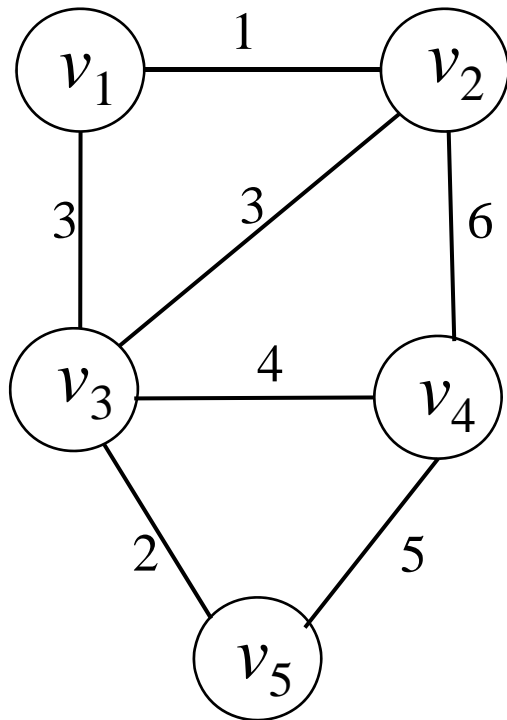
# Kruskal's Algorithm

## □ 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

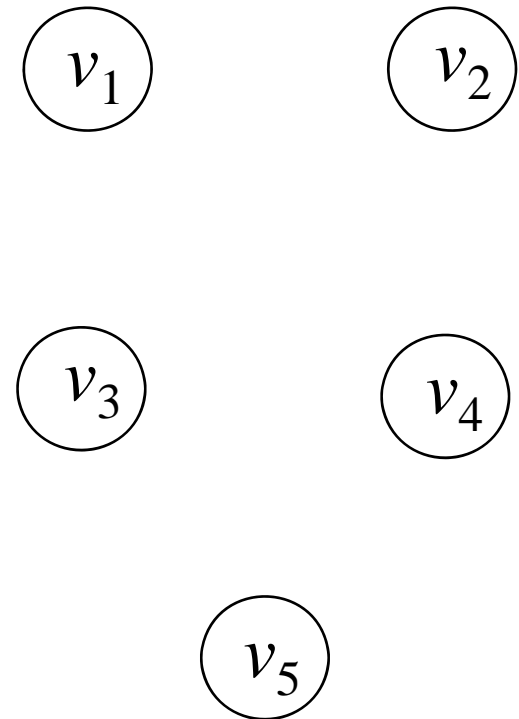
Determining a MST



1. Edges are sorted  
by weight

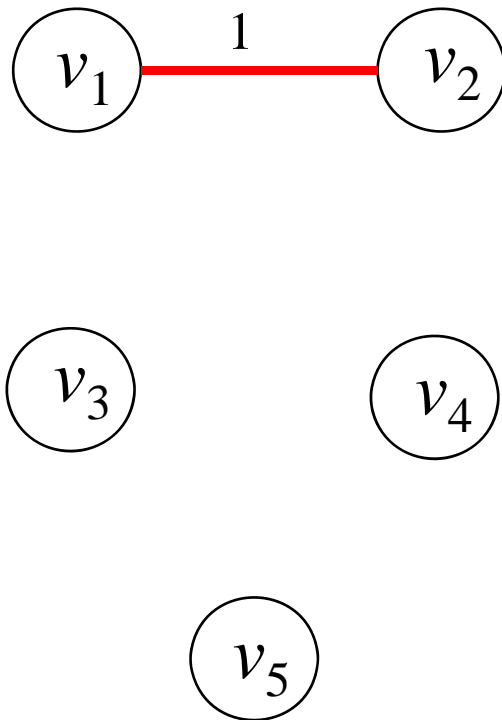
$(v_1, v_2)$	1
$(v_3, v_5)$	2
$(v_1, v_3)$	3
$(v_2, v_3)$	3
$(v_3, v_4)$	4
$(v_4, v_5)$	5
$(v_2, v_4)$	6

2. Disjoint sets are created

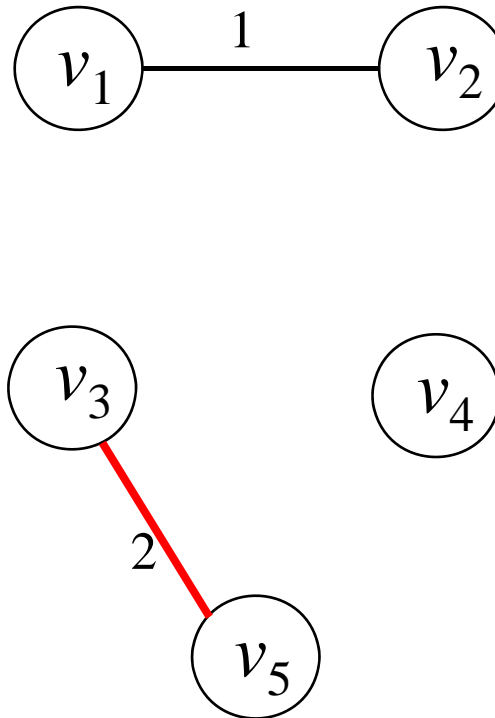


# Kruskal's Algorithm

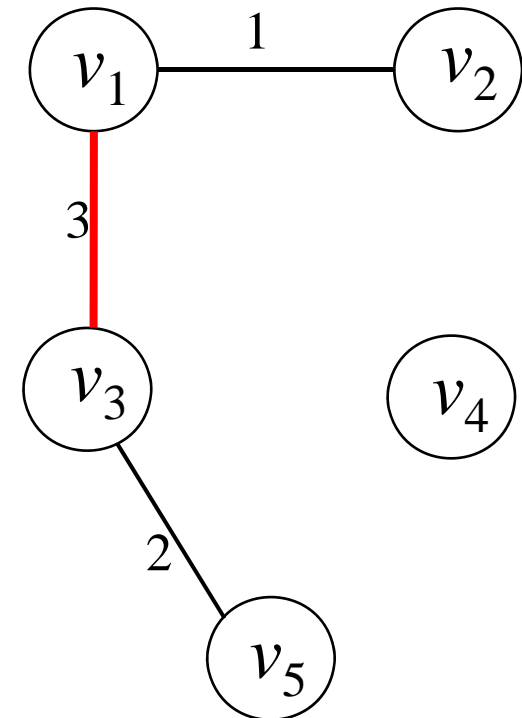
3.  $(v_1, v_2)$  is selected



4.  $(v_3, v_5)$  is selected



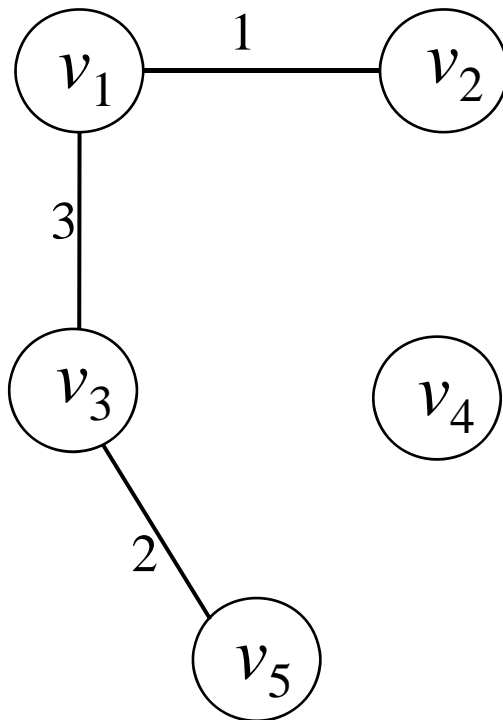
5.  $(v_1, v_3)$  is selected



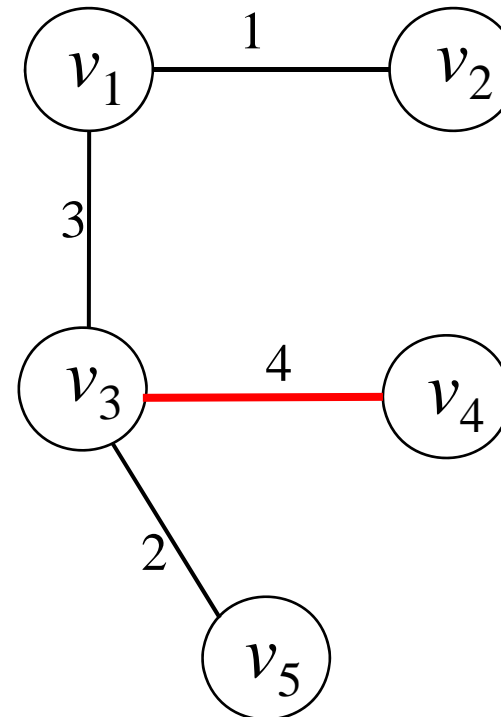


# Kruskal's Algorithm

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  $\Theta(m \lg m)$ 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)

# Minimum Spanning Tree

## 두 알고리즘의 비교

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

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

# Minimum Spanning Tree

## □ 토론 사항

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

Prim 의 알고리즘	$W(m,n)$	sparse graph	dense graph
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

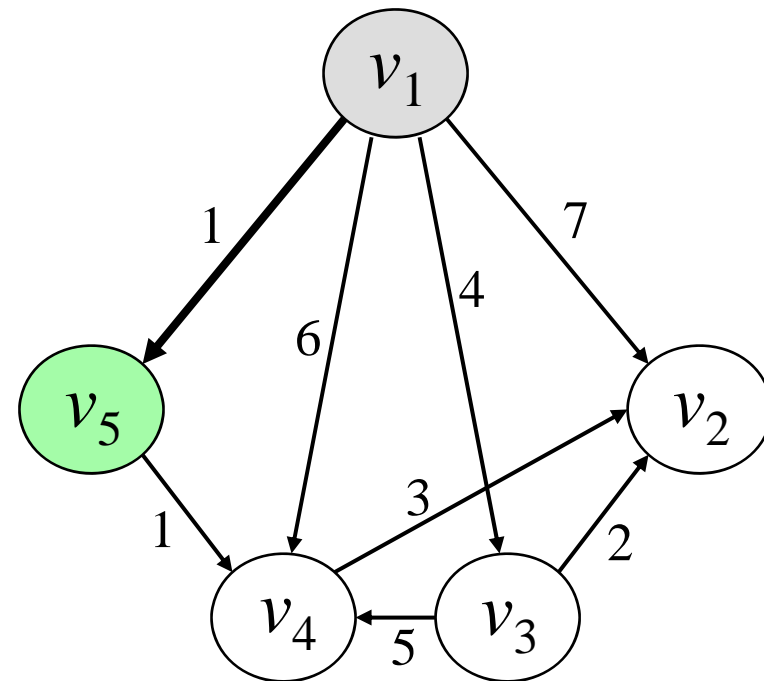
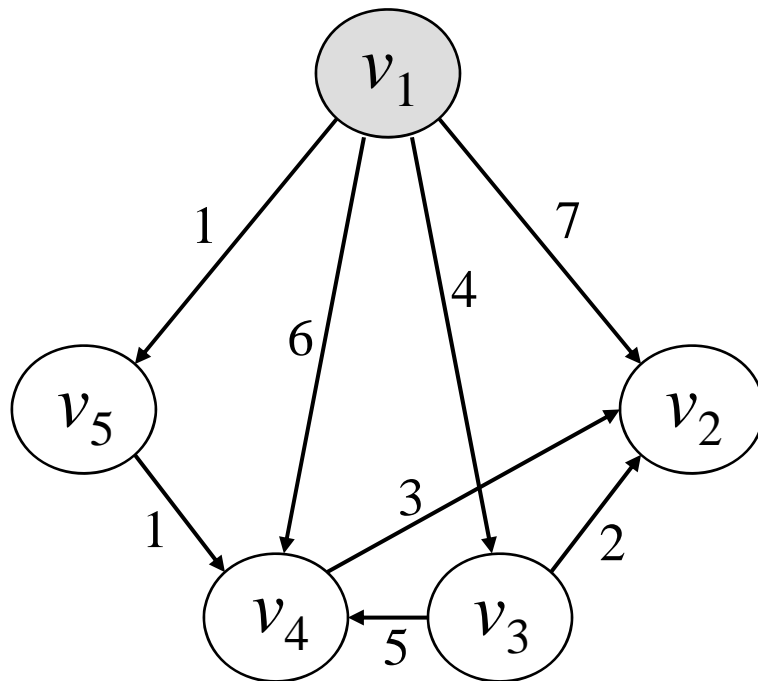
- 가중치가 있는 방향성 그래프에서 한 특정 정점에서 다른 모든 정점으로 가는 최단경로 구하는 문제
- 시작점  $v_1$
- 알고리즘

```
F := 0;
Y := {v1};
While (the instance is not solved)
    select a vertex v from V - Y, that has a shortest path // selection procedure
    from v1, 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;

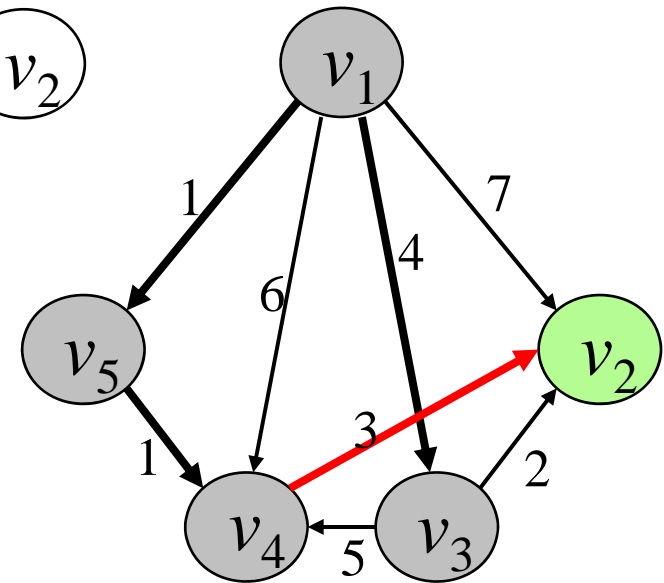
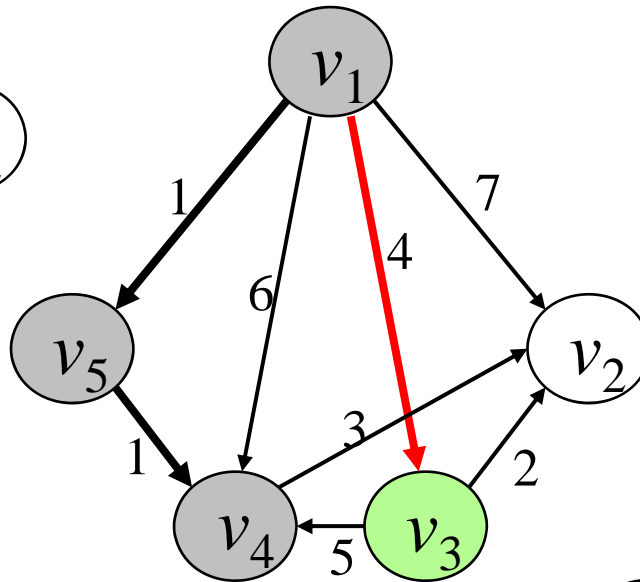
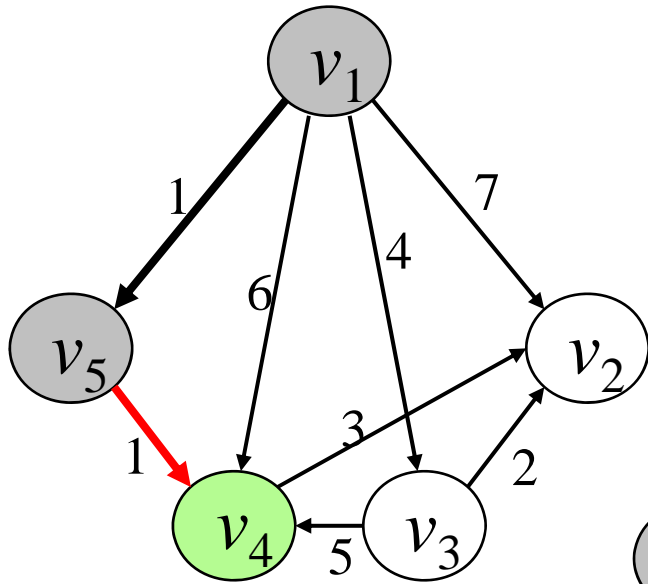
    if (Y == V)                                              // solution check
        the instance is solved;
```

# Dijkstra's Algorithm





# Dijkstra's Algorithm



# Dijkstra's Algorithm

- Define touch & length

- touch[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
- length[i] = length of the current shortest path from  $v_1$  to  $v_i$  using only vertices in  $Y$  as intermediates

# Dijkstra's Algorithm

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

    F =  $\Phi$ ;
    for(i=2; i <= n; i++) {          // For all vertices, initialize v1 to be the last
        touch[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 touch[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];
                touch[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.

# Scheduling

- Goal of scheduling
  - Minimize the total time they spend both waiting and being served  
(time in the system)
  - 2 examples
    - A hair stylist – customers for different treatments (serving times)
    - Schedule with deadlines
      - Each job has the same amount of time to complete, but has a deadline

# Scheduling

- Minimizing total time in the system

- Ex 4.2 There are 3 jobs:  $t_1 = 5$ ,  $t_2 = 10$ ,  $t_3 = 4$

Job

- 1 5 (service time)
- 2 5 (wait for job 1) + 10 (service time)
- 3 5 (wait for job 1) + 10 (wait for job 2) + 4 (service time)

■ Schedule	Total time in the system	
[1, 2, 3]	$5 + (5+10) + (5+10+4)$	= 39
[1, 3, 2]	$5 + (5+4) + (5+4+10)$	= 33
[2, 1, 3]	$10 + (10+5) + (10+5+4)$	= 44
[2, 3, 1]	$10 + (10+4) + (10+4+5)$	= 43
[3, 1, 2]	$4 + (4+5) + (4+5+10)$	= 32
[3, 2, 1]	$4 + (4+10) + (4+10+5)$	= 37

➔ Intuition: Execute the shortest jobs first

# Scheduling

## □ Algorithm

```
Sort the jobs by service time in nondecreasing order;
While (the instance is not solved) {
    schedule the next job;                // selection procedure and
                                          // feasibility check
    if (there are no more jobs)          // solution check
        the instance is solved;
}
```

## □ Time complexity

- $W(n) = \Theta(n \lg n)$

# Scheduling

## □ Theorem 4.3

The only schedule that minimizes the total time in the system is one that schedules jobs in nondecreasing order by service time

- Proof (by contradiction)

If they are not scheduled in nondecreasing order,

then for at least one  $i$  where  $1 \leq i \leq n-1$ ,  $t_i > t_{i+1}$

We can rearrange our original schedule by changing  $i$ -th,  $(i+1)$ -st

$$T' = T + t_{i+1} - t_i$$

$T \leq$  total time in the original schedule

$T' \leq$  total time in the rearranged schedule

Because  $t_i > t_{i+1}$ ,  $T' < T \rightarrow \text{Contradict!!}$



# Scheduling

## □ Multiple-Server Scheduling Problem

- $m$  servers
- Order the jobs by service time in nondecreasing order
  - server 1   jobs 1,  $(1+m)$ ,  $(1+2m)$ ,  $(1+3m)$ , .....
  - server 2   jobs 2,  $(2+m)$ ,  $(2+2m)$ ,  $(2+3m)$ , .....
  - :
  - server  $i$    jobs  $i$ ,  $(i+m)$ ,  $(i+2m)$ ,  $(i+3m)$ , .....
  - :
  - server  $m$    jobs  $m$ ,  $(m+m)$ ,  $(m+2m)$ ,  $(m+3m)$ , .....
- The jobs end up being processed in the following order
  - 1, 2, 3, ...,  $m$ ,  $1+m$ ,  $2+m$ , ...,  $m+m$ ,  $1+2m$ , ...

# Scheduling

## □ Scheduling with Deadlines

### ● Ex 4.3

Job	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

■ Schedule	Total Profit
[1, 3]	$30 + 25 = 55$
[2, 1]	$35 + 30 = 65$
[2, 3]	$35 + 25 = 60$
[3, 1]	$25 + 30 = 55$
[4, 1]	$40 + 30 = 70$
[4, 3]	$40 + 25 = 65$

# Scheduling

- Feasible sequence
  - all the jobs in the sequence start by their deadlines
- Feasible set
  - if there exists at least one feasible sequence for the jobs in the set
- Optimal sequence
  - a feasible sequence with maximum total profit
- Optimal set of jobs
  - the set of jobs in the optimal sequence

# Scheduling

## □ Algorithm

```
sort the jobs in nondecreasing order by profit;

S =  $\Phi$ ;

while (the instance is not solved) {
    select the next job;                // selection procedure and
    if (S is feasible with this job added) // feasibility check
        add this job to S;
    if (there are no more jobs)         // solution check
        the instance is solved;
}
```

# Scheduling

## □ Ex 4.4

Job	Deadline	Profit
1	3	40
2	1	35
3	1	30
4	3	25
5	1	20
6	3	15
7	2	10

1. S is set to  $\emptyset$
2. S is set to {1} because the sequence [1] is feasible
3. S is set to {1,2} because the sequence [2,1] is feasible
4. {1,2,3} is rejected because there is no feasible sequence for this set
5. S is set to {1,2,4} because the sequence [2,1,4] is feasible
6. {1,2,4,5} is rejected because there is no feasible sequence for this set
7. {1,2,4,6} is rejected because there is no feasible sequence for this set
8. {1,2,4,7} is rejected because there is no feasible sequence for this set

# Scheduling

## □ Lemma 4.3

- Let  $S$  be a set of jobs. Then  $S$  is feasible iff the sequence obtained by ordering the jobs in  $S$  according to nondecreasing deadlines is feasible
- Proof:

(충분) Suppose  $S$  is feasible & there is at least one feasible sequence

Suppose  $[.., x, y, ..]$  and  $y$  has a smaller deadline than  $x$

So the new sequence  $[.., y, x, ..]$  is feasible

(필요) Of course.  $S$  is feasible if the ordered sequence is feasible

## □ Ex 4.5

- To determine whether  $\{1, 2, 4, 7\}$  is feasible,  
Lemma 4.3 says we need to only check the feasibility of the sequence  
 $[2, 7, 1, 4]$   
(1) (2) (3) (3) -- deadlines
- Because job 4 is not scheduled by its deadline, the sequence is not feasible.  
By Lemma 4.3, the set is not feasible.

# Scheduling

## □ Algorithm 4.4: Scheduling with Deadlines

Problem: determine the schedule with maximum total profit

Inputs:  $n$  (the number of jobs);

array of integers  $\text{deadline}[1..n]$  (in non-decreasing order)

Output: an optimal sequence  $J$  for the jobs

```
void schedule (int n, const int deadline[],
               sequence_of_integers& j) {
    index i;
    sequence_of_integer K;
    J = [1];
    for (i=2; i<=n; i++){
        K=J with i added according to nondecreasing values of deadline[i];
        if (K is feasible)
            J = K;
    }
}
```

# Scheduling

□ Ex 4.6

Job	Deadline
1	3
2	1
3	1
4	3
5	1
6	3
7	2

1. J is set to [1]
2. K is set to [2,1] and is determined to be feasible  
J is set to [2,1] because K is feasible
3. K is set to [2,3,1] and is rejected because it is not feasible
4. K is set to [2,1,4] and is determined to be feasible  
J is set to [2,1,4] because K is feasible
5. K is set to [2,5,1,4] and is rejected because it is not feasible
6. K is set to [2,1,6,4] and is rejected because it is not feasible
7. K is set to [2,7,1,4] and is rejected because it is not feasible



# Scheduling

## □ Worst-Case Time Complexity Analysis

- Basic operation: a comparison operation
- Input size:  $n$ , the number of jobs
- Analysis
  - It takes a time of  $\Theta(n \lg n)$  to sort the jobs
  - In each iteration of the for  $i$  loop,  
do at most  $i-1$  comparisons to add the  $i$ -th job to  $K$ ,  
and at most  $i$  comparisons to check if  $K$  is feasible
  - Therefore, the worst case is

$$\sum_{i=2}^n [(i-1) + i] = n^2 - 1 \in \Theta(n^2)$$

- Because this time dominates the sorting time,  $W(n) \in \Theta(n^2)$

# 탐욕적인 방법과 동적계획법의 비교

탐욕적인 접근방법	동적계획법
최적화 문제를 푸는데 적합	최적화 문제를 푸는데 적합
알고리즘이 존재할 경우 보통 더 효율적	때로는 불필요하게 복잡
단일출발점 최단경로 문제: $\Theta(n^2)$	단일출발점 최단경로 문제: $\Theta(n^3)$
알고리즘이 최적인지를 증명해야 함	최적화 원칙이 적용되는지를 점검해 보기만 하면 됨
배낭 빈틈없이 채우기 문제는 풀지만, 0-1 배낭 채우기 문제는 풀지 못함	0-1 배낭 채우기 문제를 푼다

# The Knapsack Problem

## □ Problem:

$$S = \{item_1, item_2, \dots, item_n\},$$

$w_i = item_i$ 의 무게

$p_i = item_i$ 의 가치

$W$  = 배낭에 넣을 수 있는 최대 무게

라고 할 때,  $\sum_{item_i \in A} w_i \leq W$ 를 만족하면서

$\sum_{item_i \in A} p_i$ 가 최대가 되도록

$A \subseteq S$ 가 되는  $A$ 를 결정하는 문제이다.

## □ 무작정 (탐욕적) 알고리즘

- $n$ 개의 물건에 대해서 모든 부분 집합을 다 고려한다.
- 그러나 불행하게도 크기가  $n$ 인 집합의 부분집합의 수는  $2^n$ 개이다.

→ Proof? (2 ways)

# The Knapsack Problem

## □ The 0-1 Knapsack Problem (1)

- 가장 비싼 물건부터 우선적으로 채운다.
- 애석하게도 이 알고리즘은 최적이지 않다!

- 왜 아닌지 보기:  $W = 30$  lb

품목	무게	값
$item_1$	25 lb	\$10
$item_2$	10 lb	\$9
$item_3$	10 lb	\$9

- 탐욕적인 방법:  $item_1 \Rightarrow 25 \text{ lb} \Rightarrow \$10$
- 최적인 해답:  $item_2 + item_3 \Rightarrow 20 \text{ lb} \Rightarrow \$18$

# The Knapsack Problem

## □ The 0-1 Knapsack Problem (2)

- 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다.
- 그래도 최적이지 않다!

- 왜 아닌지 보기:  $W = 30$  lb

품목	무게	값	값어치
$item_1$	5 lb	\$50	\$10/lb
$item_2$	10 lb	\$60	\$6/lb
$item_3$	20 lb	\$140	\$7/lb

- 탐욕적인 방법:  $item_1 + item_3 \Rightarrow 25$  lb  $\Rightarrow$  \$190
- 최적인 해답:  $item_2 + item_3 \Rightarrow 30$  lb  $\Rightarrow$  \$200

# The Knapsack Problem

## □ The Fractional Knapsack Problem

- 물건의 일부분을 잘라서 담을 수 있다.
- 탐욕적인 접근방법으로 최적해를 구하는 알고리즘을 만들 수 있다.
- $item_1 + item_3 + (5/10) * item_2 = \$50 + \$140 + (5/10)*\$60$   
 $\Rightarrow \$220 \quad (30 \text{ lb})$
- Optimal!

# The Knapsack Problem

## □ Dynamic Programming Approach (0-1 Knapsack Problem)

- $i > 0$  이고  $w > 0$  일 때, 전체 무게가  $w$ 가 넘지 않도록  $i$ 번째까지의 항목 중에서 얻어진 최고의 이익(optimal profit)을  $P[i][w]$ 라고 하면,

$$P[i][w] = \begin{cases} \text{maximum}(P[i-1][w], p_i + P[i-1][w - w_i]) & (\text{if } w_i \leq w) \\ P[i-1][w] & (\text{if } w_i > w) \end{cases}$$

여기서  $P[i-1][w]$ 는  $i$ 번째 항목을 포함시키지 않는 경우의 최고 이익이고,  $p_i + P[i-1][w - w_i]$ 는  $i$ 번째 항목을 포함시키는 경우의 최고 이익이다. 위의 재귀 관계식이 최적화 원칙을 만족하는지는 쉽게 알 수 있다.

- 그러면 어떻게 최대 이익  $P[n][W]$ 값을 구할 수 있을까?
  - int  $P[0..n][0..W]$ 의 2차원 배열을 만든 후, 각 항을 계산하여 넣는다
  - 여기서  $P[0][w] = 0, P[i][0] = 0$ 으로 놓으면 되므로, 계산해야 할 항목의 수는  $nW \in \Theta(nW)$

# The Knapsack Problem

## □ Refinement of Dynamic Programming

- 여기서  $n$ 과  $W$ 와는 아무런 상관관계가 없다.  
만일 (임의적으로)  $W = n!$ 이라고 한다면, 수행시간은  $\Theta(n \times n!)$ 이 된다.  
그렇게 되면 이 알고리즘은 앞에서 얘기한 무작정 알고리즘보다도 나을게 하나도 없다.
- 그럼 이 알고리즘을 최악의 경우에  $\Theta(2^n)$  시간에 수행될 수 있도록, 즉 무작정 알고리즘 보다 느리지 않고, 때로는 훨씬 빠르게 수행될 수 있도록 개량할 수 있을까?
  - 착안점은  $P[n][W]$ 를 계산하기 위해서  $(n-1)$ 번째 행을 모두 계산할 필요가 없다는데 있다.



# The Knapsack Problem

- $P[n][W]$ 는 아래 식으로 표현할 수 있다

$$P[n][W] = \begin{cases} \text{maximum}(P[n-1][W], p_n + P[n-1][W - w_n]) & (\text{if } w_n \leq W) \\ P[n-1][W] & (\text{if } w_n > W) \end{cases}$$

- 따라서 (n-1)번째 행에서는  $P[n-1][W]$ 와  $P[n-1][W-w_n]$  항만 필요
- i-번째 행에 어떤 항목이 필요한지를 결정한 후에,  
다시 (i-1)번째 행에 필요한 항목을 결정
  - $P[i][w]$ 는  $P[i-1][w]$ 와  $P[i-1][w-w_i]$ 로 계산
- 이런 식으로  $n = 1$ 이나  $w \leq 0$ 일 때까지 계속해 나가면 된다.

# The Knapsack Problem

## □ Ex 4.7

- $W=30$  lb

품 목	무 게	값
$item_1$	5 lb	\$50
$item_2$	10 lb	\$60
$item_3$	20 lb	\$140

- We need  $P[3][W] = P[3][30]$ 
  - To compute  $P[3][30] \rightarrow \max(P[3-1][30], p_3 + P[3-1][30-w_3])$   
 $= \max(P[2][30], p_3 + P[2][10])$
  - To compute  $P[2][30] \rightarrow \max(P[2-1][30], p_2 + P[2-1][30-w_2])$   
 $= \max(P[1][30], p_2 + P[1][20])$
  - To compute  $P[2][10] \rightarrow \max(P[2-1][10], p_2 + P[2-1][10-w_2])$   
 $= \max(P[1][10], p_2 + P[1][0])$

# The Knapsack Problem

- Compute row 1

- $$P[1][w] = \begin{cases} \max(P[0][w], \$50 + P[0][w-5]) & (\text{if } w_1 = 5 \leq w) \\ P[0][w] & (\text{if } w_1 = 5 > w) \end{cases}$$
$$= \begin{cases} \$50 & (\text{if } w_1 = 5 \leq w) \\ \$0 & (\text{if } w_1 = 5 > w) \end{cases}$$

- Therefore

- $$P[1][0] = \$0; P[1][10] = \$50; P[1][20] = \$50; P[1][30] = \$50$$

- Compute row 2

- $$P[2][10] = \begin{cases} \max(P[1][10], \$60 + P[1][0]) & (\text{if } w_2 = 10 \leq 10) \\ P[1][10] & (\text{if } w_2 = 10 > 10) \end{cases}$$
$$= \$60$$

- $$P[2][30] = \begin{cases} \max(P[1][30], \$60 + P[1][20]) & (\text{if } w_2 = 10 \leq 30) \\ P[1][30] & (\text{if } w_2 = 10 > 30) \end{cases}$$
$$= \$60 + \$30 = \$110$$

# The Knapsack Problem

- Compute row 3

- $$P[3][30] = \begin{cases} \max(P[2][30], \$140 + P[2][10]) & (\text{if } w_3 = 20 \leq 30) \\ P[1][10] & (\text{if } w_3 = 20 > 30) \end{cases}$$
$$= \$140 + \$60 = \$200$$

- The modified algorithm – compute only 7 entries
- The original algorithm - compute  $3 \times 30 = 90$  entries

# The Knapsack Problem

- Efficiency in the worst case
  - Compute at most  $2^i$  entries in the  $(n - i)$ -th row
    - Therefore the total number is  $1 + 2 + 2^2 + \dots + 2^{n-1} = 2^n - 1$ .
    - 따라서 최악의 경우의 수행시간은  $\Theta(2^n)$
  - The number of entries computed is in  $O(nW)$ 
    - What about the number of the modified algorithm ?
    - If  $n = W+1$ , and  $w_i = 1$  for all  $i$ ,  
then the total number of entries is about
$$1 + 2 + 3 + \dots + n = n(n+1) / 2 = (W+1)(n+1) / 2$$
    - For arbitrary large values of  $n$  and  $W$ ,  $\Theta(nW)$
  - Combining these 2 results, the worst case is in  $O(\min(2^n, nW))$

# The Knapsack Problem

- 분할정복 방법으로도 이 알고리즘을 설계할 수도 있고,  
그 최악의 경우 수행시간은  $\Theta(2^n)$ 이다.
- 아직 아무도 이 문제의 최악의 경우 수행시간이  
지수(**exponential**)보다 나은 알고리즘을 발견하지 못했고,  
아직 아무도 그러한 알고리즘은 없다라고 증명한 사람도 없다.  
- **NP**문제