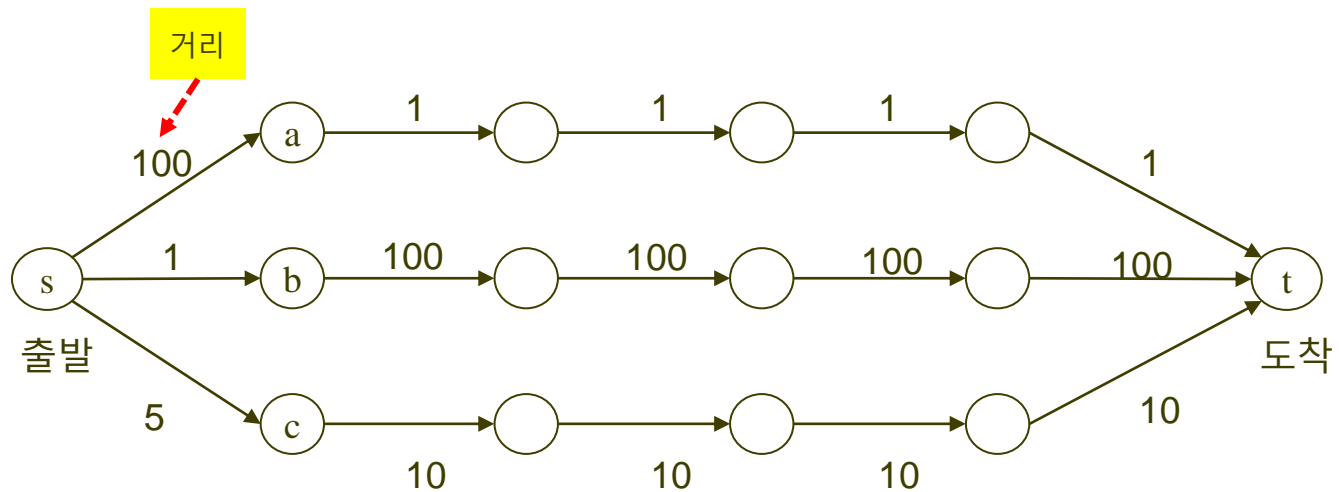


4장 탐욕적인 접근방법 (Greedy Algorithm)

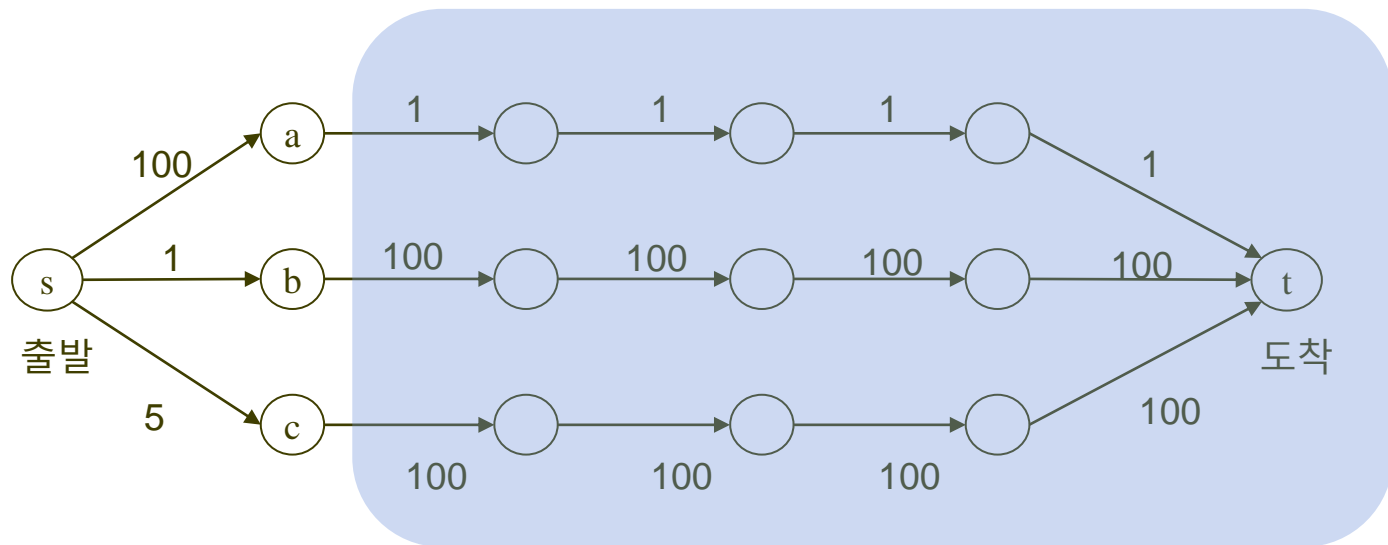
[문제]

출발지점에서 도착지점까지의 최단 경로를 찾는다.



[탐욕적 알고리즘]

출발지점에서 다음으로 직접 갈 수 있는 지점(노드) 중 가장 가까운 지점을 찾는다. 그 지점으로 이동한 후, 도착지점에 도달할 때 까지 같은 방법을 수행한다.



만일 출발지점에서 100, 1, 5 중 제일 가까운 b를 선택한다면, 최적해를 못 찾음

Greedy Algorithm

- 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. 선정과정(selection procedure)

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

2. 적정성점검(feasibility check)

새로 얻은 해답모음이 적절한지를 결정한다.

3. 해답점검(solution check)

새로 얻은 해답모음이 최적의 해인지를 결정한다.

보기: 거스름돈 문제

- 문제: 동전의 개수가 최소가 되도록 거스름 돈을 주는 문제
- 탐욕적인 알고리즘
 - ✓ 거스름돈을 x 라 하자.
 - ✓ 먼저, 가치가 가장 높은 동전부터 x 가 초과되지 않도록 계속 내준다.
 - ✓ 이 과정을 가치가 높은 동전부터 내림순으로 총액이 정확히 x 가 될 때까지 계속한다.
- 현재 우리나라에서 유통되고 있는 동전만을 가지고, 이 알고리즘을 적용하여 거스름돈을 주면, 항상 동전의 개수는 최소가 된다. 따라서 이 알고리즘은 최적(optimal)!

Coins



거스름돈: 36센트

Amount owed: 36 cents

Step

Total Change

1. Grab quarter



25 Cents

2. Grab first dime



25+10 Cents

3. Reject second dime



25+10+~~10~~ Cents

4. Reject nickel

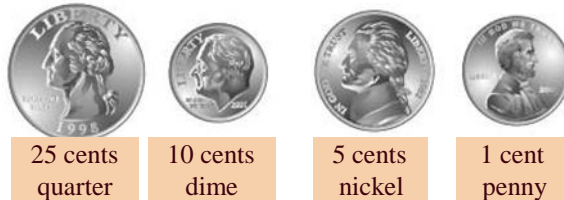


25+10+~~5~~ Cents

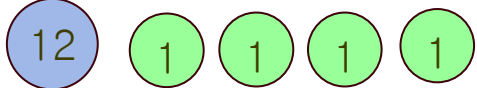
5. Grab penny



25+10+1 Cents



최적의 해를 얻지 못하는 경우

- 12원 짜리 동전을 새로 발행했다고 가정.
- 이 알고리즘을 적용하여 거스름돈을 주면, 항상 동전의 개수는 최소가 된다는 보장이 없다.
- 보기: 거스름돈 액수 = 16원
 - ✓ 탐욕알고리즘의 결과: 12원 \times 1개 = 12원, 1원 \times 4개 = 4원
 - ✓ 동전의 개수 = 5개 \Rightarrow 최적(optimal)이 아님! 
 - ✓ 최적의 해: 10원 \times 1개, 5원 \times 1개, 1원 \times 1개가 되어 동전의 개수는 3개가 된다.



Coins



Amount owed: 16 cents

Step

Total Change

1. Grab 12-cent coin



12 Cents

2. Reject dime



12+~~10~~ Cents

3. Reject nickel



12+~~5~~ Cents

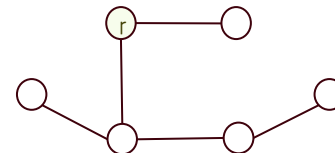
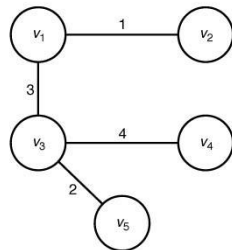
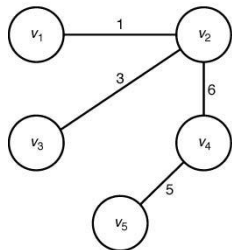
4. Grab four pennies



12+1+1+1+1 Cents

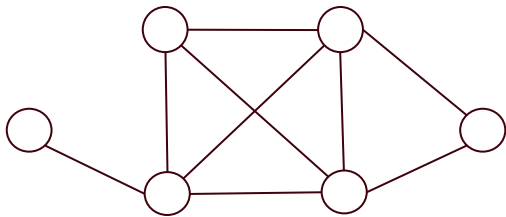
그래프 용어

- 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)
- 순환적그래프(cyclic graph), 비순환적그래프(acyclic graph).
- 트리(tree) - 비순환적이며, 비방향성 그래프.
- 뿌리 있는 트리(rooted tree) - 한 정점이 뿌리로 지정된 트리.

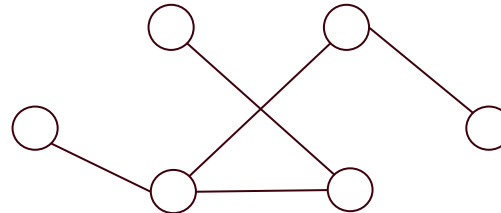
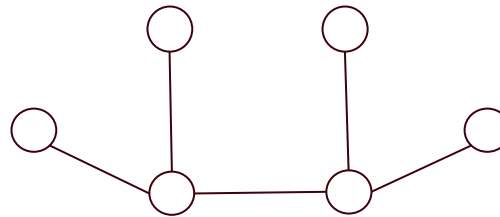


정의: 신장트리(spanning tree)

- 연결된, 비방향성 그래프 G 에서 순환경로를 제거하면서 연결된 부분그래프가 되도록 이음선을 제거하면 신장트리(spanning tree)가 된다.
- 따라서 신장트리는 G 안에 있는 모든 정점을 다 포함하면서 트리가 되는 연결된 부분그래프이다.



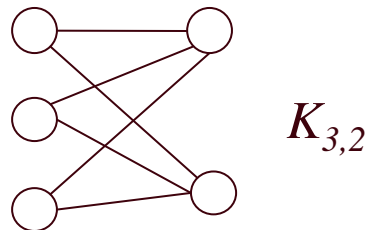
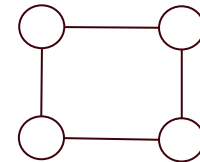
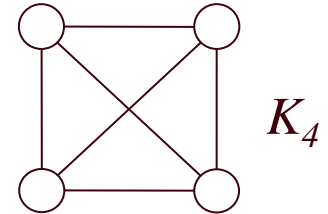
G



span: 밧줄로 매다

신장트리의 개수

- $t(G)$: connected graph의 spanning tree 개수
- 완전그래프(complete graph): 모든 노드간에 에지가 존재
 - ✓ 노드가 n 개인 complete graph K_n 의 에지 개수: $\binom{n}{2}$
 - ✓ 트리는 에지수가 $n-1$.
 - ✓ K_n 으로 부터 생성할 수 있는 에지 수가 $n-1$ 인 그래프의 개수: $\binom{n}{n-1} C_{n-1}$
- G 가 tree인 경우 $t(G)=1$
- G 가 cycle graph C_n 인 경우 $t(G)=n$
- G 가 K_n 인 경우 $t(G)=n^{n-2}$: Cayley's formula
- G 가 complete bipartite graph $K_{p,q}$ 인 경우 $t(G)=p^{q-1}q^{p-1}$

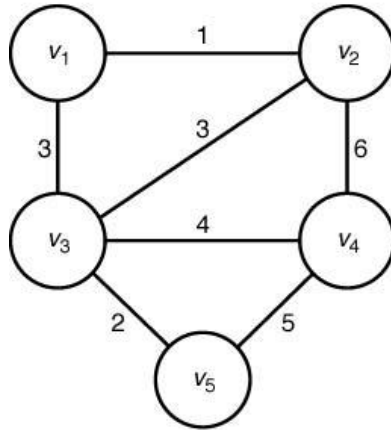


최소비용신장트리 (minimum spanning tree)

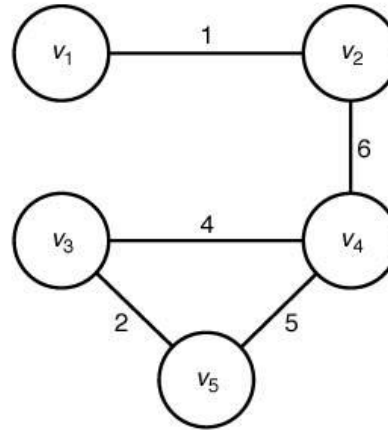
- 신장트리가 되는 G 의 부분그래프 중에서 가중치가 최소가 되는 부분그래프를 **최소비용신장트리(minimum spanning tree)**라고 한다. 여기서 최소의 가중치를 가진 부분그래프는 반드시 트리가 되어야 한다. 왜냐하면, 만약 트리가 아니라면, 분명히 순환경로(cycle)가 있을 것이고, 그렇게 되면 순환경로 상의 한 이음선을 제거하면 더 작은 비용의 신장트리가 되기 때문이다.
- 관찰: 모든 신장트리가 최소비용신장트리는 아니다.

span: 밧줄로 매다

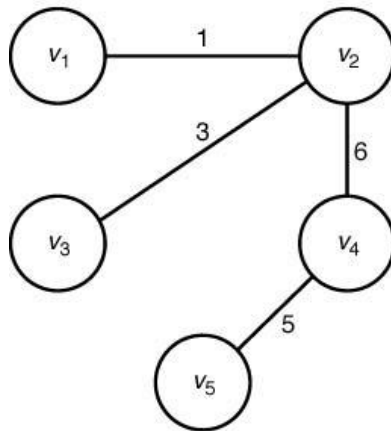
(a) A connected, weighted, undirected graph G .



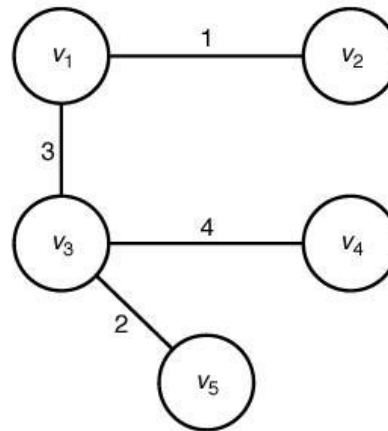
(b) If (v_4, v_5) were removed from this subgraph, the graph would remain connected.



(c) A spanning tree for G .

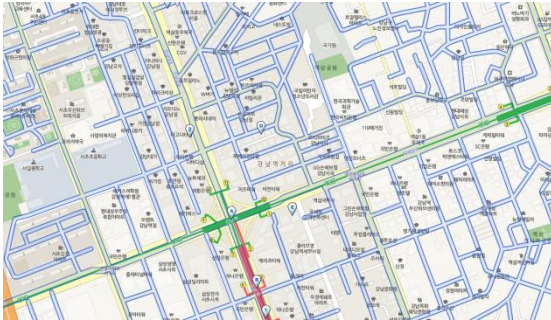


(d) A minimum spanning tree for G .

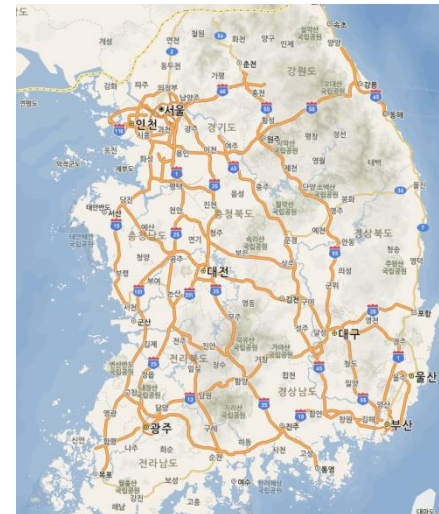


최소비용신장트리의 적용 예

- 도로건설
 - 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



from naver map



from naver map

무작정 알고리즘

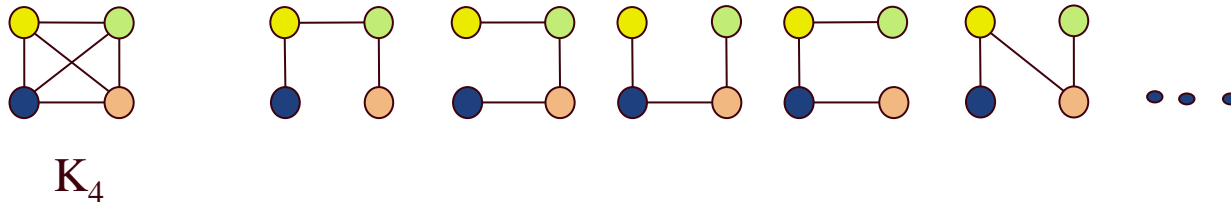
- 알고리즘

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

- 분석

- ✓ 이는 최악의 경우, 지수보다도 나쁘다.
- ✓ 이유? – spanning tree의 개수는 n 에 대해 최대 n^{n-2} .

- For $n=4$, 16 spanning trees.

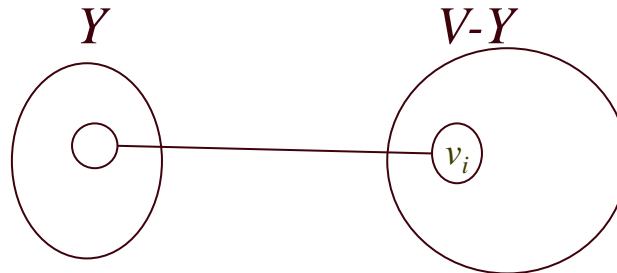


탐욕적인 알고리즘

- 문제: 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.
- 알고리즘:
 1. $F := \emptyset$;
 2. 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복
 - (a) 선정 절차:
적절한 최적해 선정절차에 따라서 하나의 이음선을 선정
 - (b) 적정성 점검:
선정한 이음선을 F 에 추가시켜도 사이클이 생기지 않으면,
 F 에 추가.
 - (c) 해답 점검: $T = (V, F)$ 가 신장트리이면, 사례해결. T 는 최소비용신장트리.

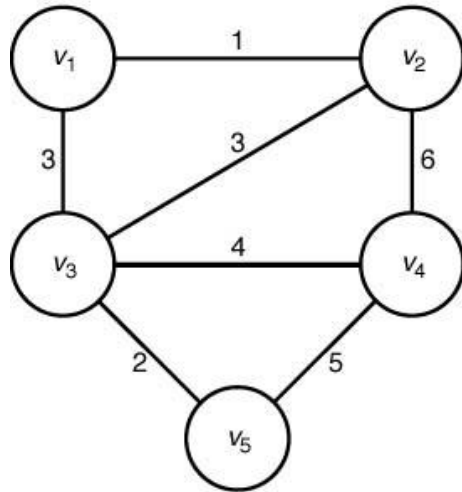
Prim의 알고리즘(1930)

1. $F := \emptyset$;
2. $Y := \{v_1\}$;
3. while(사례 미해결){
 - (a) 선정 절차/적정성 점검: $V - Y$ 에 속한 정점 중에서, Y 에 가장 가까운 정점 하나를 선정.
 - (b) 선정한 정점을 Y 에 추가.
 - (c) Y 로 이어지는 이음선을 F 에 추가.
 - if ($Y == V$)
 - (d) 해답 점검: $Y = V$ 가 되면, $T = (V, F)$ 가 최소비용신장트리}

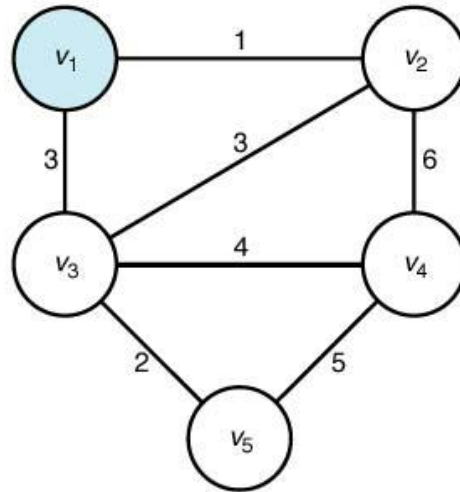


Prim's 알고리즘

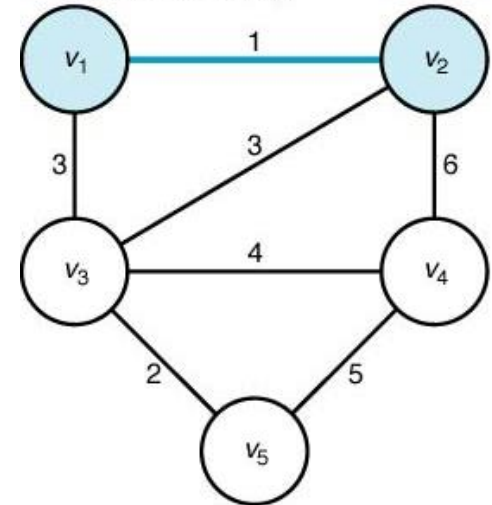
Determine a minimum spanning tree.



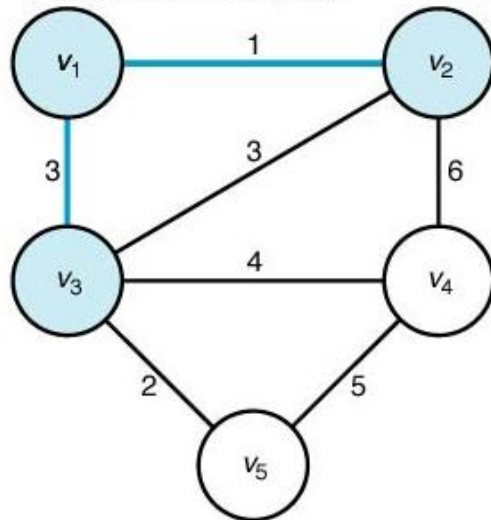
1. Vertex v_1 is selected first.



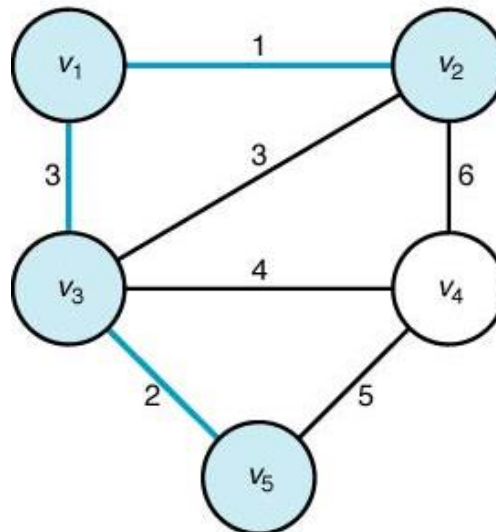
2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.



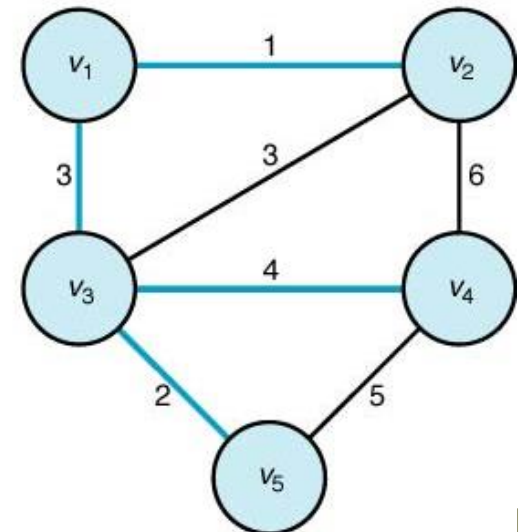
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$.



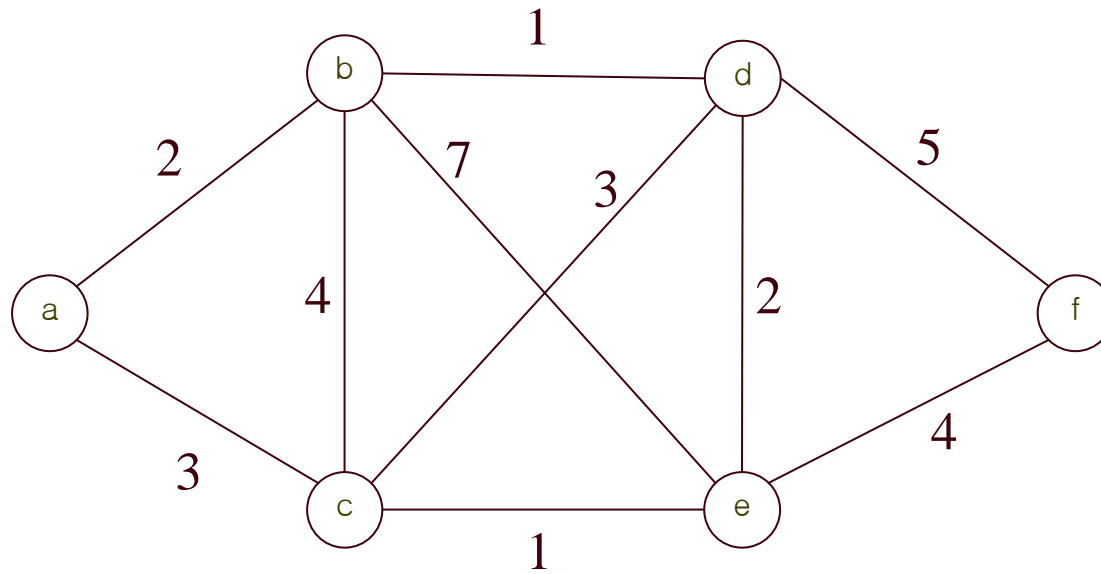
4. Vertex v_5 is selected because it is nearest to $\{v_1, v_2, v_3\}$.



5. Vertex v_4 is selected.



(ex)

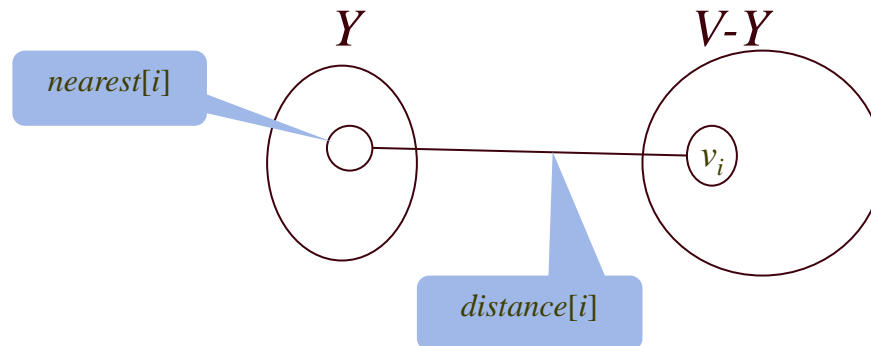


Prim의 알고리즘 (세부적)

- 그래프의 인접행렬식 표현

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

- $W[i][j]$ 는 대칭(symmetric)
- 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
 $\text{distance}[i] = v_i$ 와 $\text{nearest}[i]$ 를 잇는 이음선의 가중치

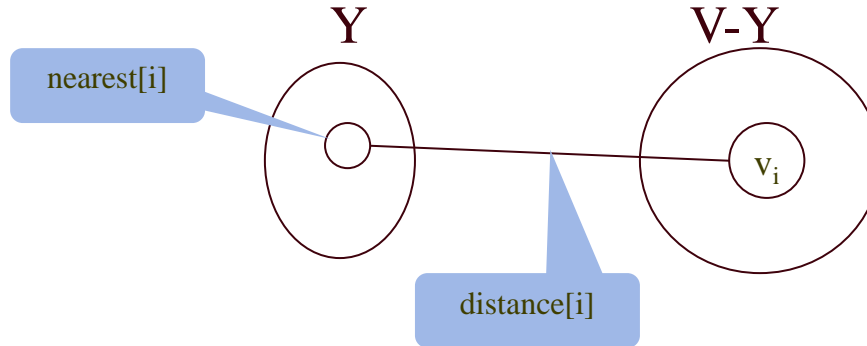


```

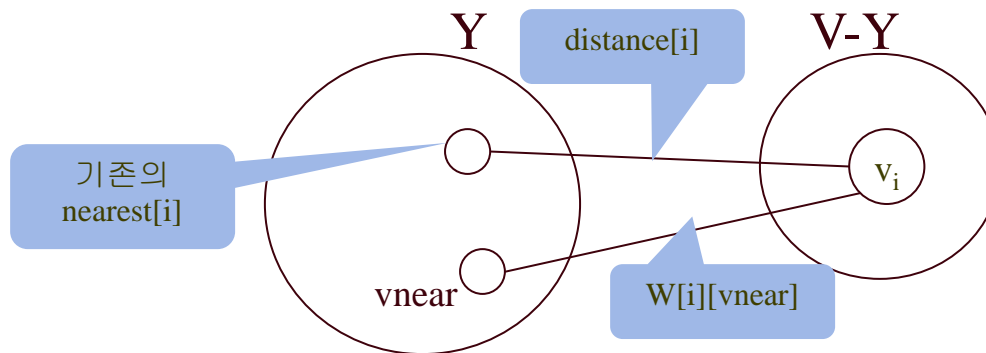
void prim(int n,                // 입력: 정점의 수
    const number W[][],        // 입력: 그래프의 인접행렬식 표현
    set_of_edges& F) {        // 출력: 그래프의 MST에 속한 이음선의 집합
    index i, vnear;  number min;  edge e;  index nearest[2..n]; number distance[2..n];

    F =  $\phi$ ;
    for(i=2; i <= n; i++) { // 초기화
        nearest[i] = 1;      // vi에서 가장 가까운 정점을 v1으로 초기화
        distance[i] = W[1][i]; // vi과 v1을 잇는 이음선의 가중치로 초기화
    }
    repeat(n-1 times) {      // n-1개의 정점을 Y에 추가한다
        min =  $\infty$ ;
        for(i=2; i <= n; i++) // 각 정점에 대해서
            if (0 <= distance[i] < min) { // distance[i]를 검사하여
                min = distance[i];        // 가장 가까이 있는 vnear을
                vnear = i;                 // 찾는다.
            }
        e = vnear와 nearest[vnear]를 잇는 이음선;
        e를 F에 추가;
        distance[vnear] = -1;             // 찾은 노드를 Y에 추가한다.
        for(i=2; i <= n; i++)
            if (W[i][vnear] < distance[i]) { // Y에 없는 각 노드에 대해서
                distance[i] = W[i][vnear];    // distance[i]를 갱신한다.
                nearest[i]=vnear;
            }
    }
}

```



- V-Y에 있는 노드 중 $distance[i]$ 를 최소로 하는 i 를 v_{near} 로 선정
- v_{near} 가 Y에 편입
- V-Y에 대해 $distance$ 배열을 재 계산: v_{near} 가 Y에 새로 편입되면서 $distance[i]$ 가 감소할 수 있는가 확인하는 절차 필요



```
if (W[i][v_near] < distance[i]){
    distance[i] = W[i][v_near];
    nearest[i]=v_near;}

```

Prim의 알고리즘 분석

- 분석

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

```
repeat(n-1 times) {  
    for(i=2; i <= n; i++)  
  
        for(i=2; i <= n; i++)  
  
}
```


최적여부의 검증(optimality proof)

- Prim의 알고리즘이 찾아낸 신장트리가 최소비용(minimal)인지를 검증해야 한다. 다시 말하면, Prim의 알고리즘이 최적(optimal)인지를 보여야 한다.
- **정의 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.
- **보조정리 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.

- 증명: F 가 유망하기 때문에 $F \subseteq F'$ 이면서 (V, F') 가 최소비용신장트리(MST)가 되는 이음선 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 순환경로를 하나 포함하게 되고, e 는 반드시 그 순환경로 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 순환경로 안에 반드시 존재하게 된다. 여기서 만약 $F' \cup \{e\}$ 에서 e' 를 제거하면, 그 순환경로는 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.

- 정리: Prim의 알고리즘은 항상 최소비용신장트리를 만들어 낸다.

증명: (수학적귀납법)

매번 반복이 수행된 후에 집합 F 가 유망하다는 것을 보이면 된다.

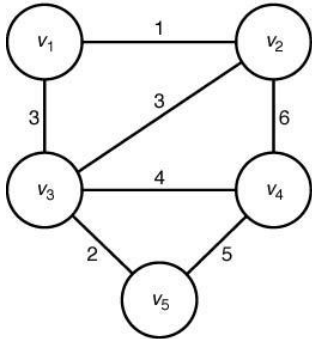
- ◆ 출발점: 공집합은 당연히 유망하다.
- ◆ 귀납가정: 어떤 주어진 반복이 이루어진 후, 그때까지 선정하였던 이음선의 집합인 F 가 유망하다고 가정한다
- ◆ 귀납절차: 집합 $F \cup \{e\}$ 가 유망하다는 것을 보이면 된다. 여기서 e 는 다음 단계의 반복 수행 시 선정된 이음선 이다. 그런데, 위의 보조정리 1에 의하여 $F \cup \{e\}$ 은 유망하다고 할 수 있다. 왜냐하면 이음선 e 는 Y 에 있는 어떤 정점을 $V - Y$ 에 있는 어떤 정점으로 잇는 이음선 중에서 최소의 가중치를 가지고 있기 때문이다.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

Kruskal의 알고리즘(1956)

1. $F := \phi$;
2. 서로소(素, disjoint)가 되는 V 의 부분집합 들을 만드는데, 각 부분집합 마다 하나의 정점만 가짐
3. E 안에 있는 이음선을 가중치의 비내림차순으로 정렬
4. while(답을 구하지 못했음){
 - (a) 선정 절차: 최소가중치를 갖고 있는 다음 이음선을 선정
 - (b) 적정성 점검: 만약 선정된 이음선이 두개의 서로소인 정점을 잇는다면, 먼저 그 부분집합을 하나의 집합으로 합하고, 그 다음에 그 이음선을 F 에 추가한다.
 - (c) 해답 점검: 만약 모든 부분집합이 하나의 집합으로 합하여 지면,
그 때 $T = (V, F)$ 가 최소비용신장트리.}

1. Edges are sorted by weight.

Determine a minimum spanning tree.



(v_1, v_2) 1

(v_3, v_5) 2

(v_1, v_3) 3

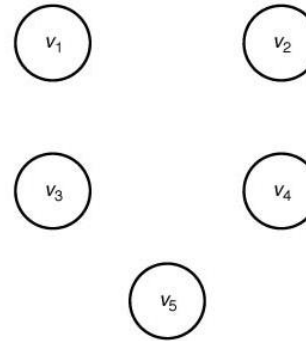
(v_2, v_3) 3

(v_3, v_4) 4

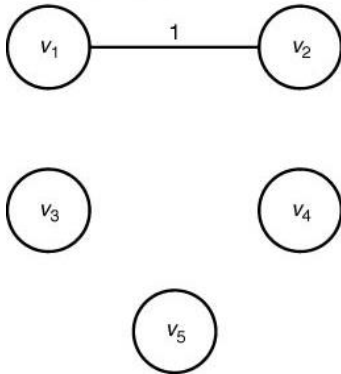
(v_4, v_5) 5

(v_2, v_4) 6

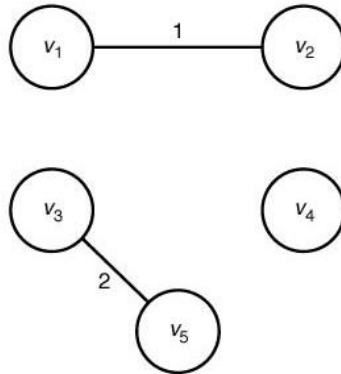
2. Disjoint set are created.



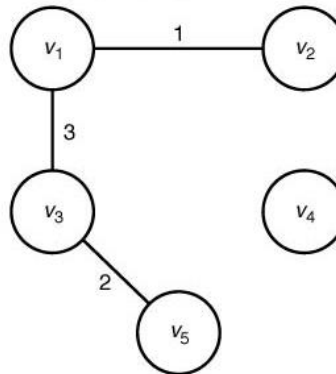
3. Edge (v_1, v_2) is selected.



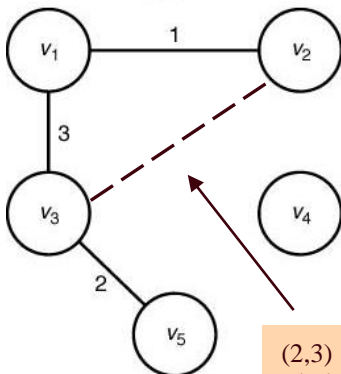
4. Edge (v_3, v_5) is selected.



5. Edge (v_1, v_3) is selected.

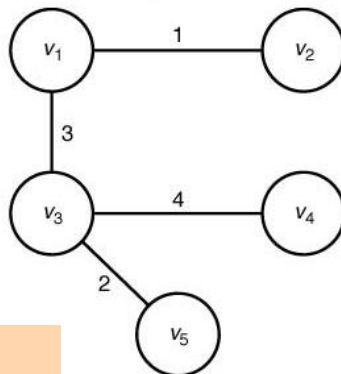


6. Edge (v_2, v_3) is selected.

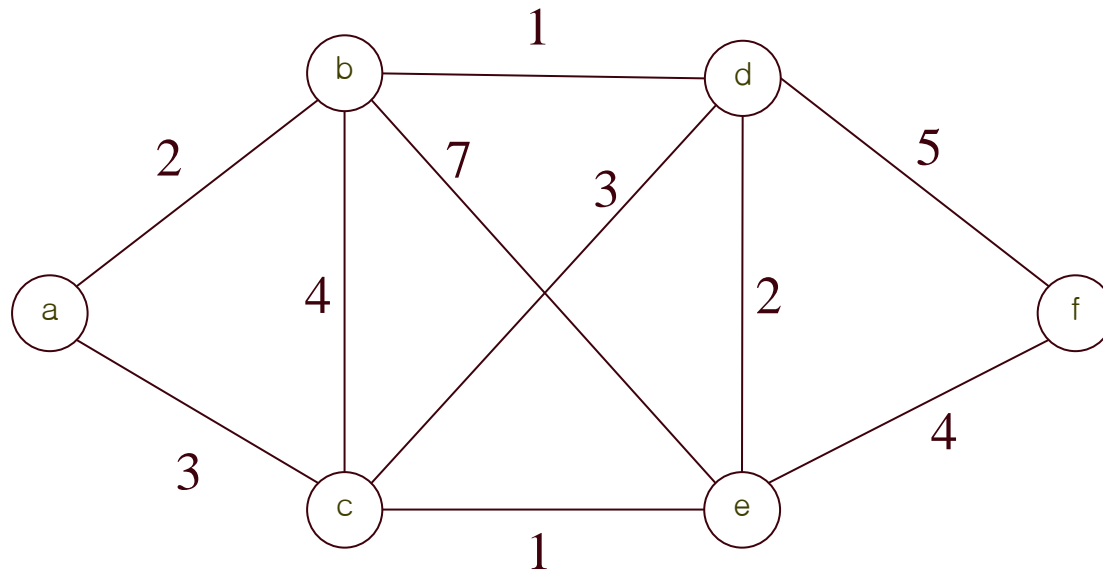


(2,3) 은
사이클을 만듦

7. Edge (v_3, v_4) is selected.



(ex)



```

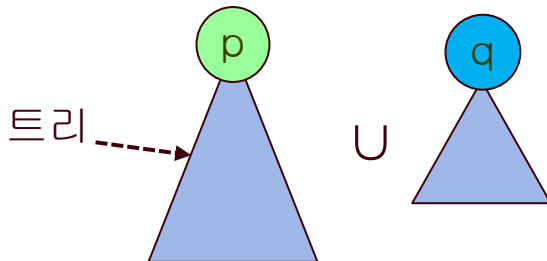
void kruskal(int n, int m,      // 입력: 정점의 수 n, 에지의 수 m
             set_of_edges E,  // 입력: 가중치를 포함한 이음선의 집합
             set_of_edges& F) { // 출력: MST를 이루는 이음선의 집합

    index i, j;
    set_pointer p, q;
    edge e;
    E에 속한 m개의 이음선을 가중치의 비내림차순으로 정렬;
    F =  $\phi$ ;
    initial(n);
    while (F에 속한 이음선의 개수가 n-1보다 작다) {
        e = 아직 점검하지 않은 최소의 가중치를 가진 이음선;
        (i, j) = e를 이루는 양쪽 정점의 인덱스;
        p = find(i);
        q = find(j);
        if (!equal(p,q)) {
            merge(p,q);
            e를 F에 추가;
        }
    }
}

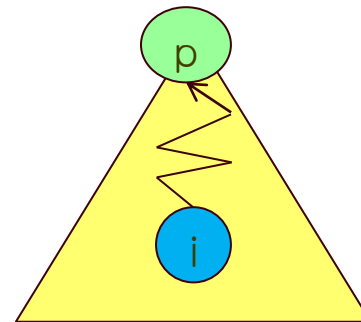
```

● 서로소 집합 추상 데이터타입(disjoint set abstract data type)

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



$\text{merge}(p, q) = \text{union}(p, q)$



$p = \text{find}(i)$

- Ackermann's function

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

$$A(2, 2) = A(1, A(2, 1))$$

$$A(2, 1) = A(1, A(2, 0))$$

$$A(2, 0) = A(1, 1)$$

$$A(1, 1) = A(0, 1)$$

$$A(0, 1) = 2$$

$$A(1, 2) = A(0, A(1, 1)) = A(0, 2)$$

$$A(0, 2) = 3$$

$$A(0, 3) = 4 \dots\dots$$

✓ Extremely fast growing function

✓ $A(4, 2)$ is an integer of 19,729 decimal digits.

✓ $A(4, 0) = 2^{2^2} - 3 = 13$

✓ $A(4, n) = 2^{2^{2^{\dots^{2^{n+2}}}}} - 3$

$$A(4, 1) = 2^{2^{2^2}} - 3 = 2^{2^4} - 3 = 2^{16} - 3 = 2^{10+6} - 3 = 1024 \times 64 - 3 = 65533$$

$$A(4, 2) = 2^{65536} - 3$$

$$((2^2)^2)^2 = 2^{2*2*2} = 2^8 = 256$$

- Ackermann 함수를 단순화 시켜 함수 F 생성. 함수 G 는 F 의 역함수

$$F(0) = 1,$$

$$F(i) = 2^{F(i-1)}, \text{ for } i > 0.$$

n	$F(n)$
0	1
1	2
2	4
3	16
4	65,536
5	$2^{65,536}$

n	$G(n)$
1	0
2	1
3~4	2
5~16	3
17~65,536	4
65,537~ $2^{65,536}$	5

- $G(n) = \min\{k \mid F(k) \geq n\}$,
✓ $G(100)=4$
- G grows extremely slowly.
- $G(n) \leq 5$ for all “practical” value of n , i.e., for all $n \leq 2^{65,536}$.
- ✓ n *union* and *find* operations take at most $O(nG(n))$ with path compression and weighting union heuristic.
- ✓ superlinear



Kruskal의 알고리즘의 분석

- ✓ 단위연산: find, equal, merge 내에 있는 포인터 이동 또는 데이터 비교문

- ✓ 입력크기: 정점의 수 n 과 에지의 수 m

1. 에지들을 정렬하는데 걸리는 시간: $\Theta(m \lg m)$

2. n 개의 disjoint set을 초기화하는데 걸리는 시간: $\Theta(n)$

3. 반복문 안에서 걸리는 시간: 루프를 최대 m 번 수행한다. disjoint set data structure를 사용하여 구현하고, 1회 반복에서 find, equal, merge(union) 같은 동작을 호출하는 횟수가 상수이면, m 번 반복에 대한 시간복잡도는 $\Theta(m \lg n)$ (or $mG(m)$)이다.

트리사용, weighting union heuristic, path compression 사용

```
void kruskal (.....)
{
    E에 속한 m개의 이음선을 가중치의 비내림차순으로 정렬;
    F =  $\phi$ ;
    initial(n);
    while (F에 속한 이음선의 개수가 n-1보다 작다) {
        . . . .
        p = find(i);
        q = find(j);
        if (!equal(p,q)) {
            merge(p,q);
            e를 F에 추가;
        }
    }
}
```

트리사용, weighting union heuristic 사용

- ✓ $m \geq n - 1$ 이기 때문에, 위의 1과 3은 2를 지배하게 되므로, $W(m, n) \in \Theta(m \lg m)$

Kruskal의 알고리즘의 분석

- ✓ 최악의 경우에는, 모든 정점이 다른 모든 정점과 연결이 될 수 있기 때문에,

$$m = \frac{n(n-1)}{2} \in \Theta(n^2)$$

그러므로,

$$W(m, n) \in \Theta(m \lg m) = \Theta(n^2 \lg n^2) = \Theta(2n^2 \lg n) = \Theta(n^2 \lg n)$$

- ✓ 최적여부의 검증(optimality proof)
 - - Prim의 알고리즘의 경우와 비슷함. (교재 참조)

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 $\Theta(n^2 \lg n)$	$\Theta(n \lg n)$	$\Theta(n^2 \lg n)$

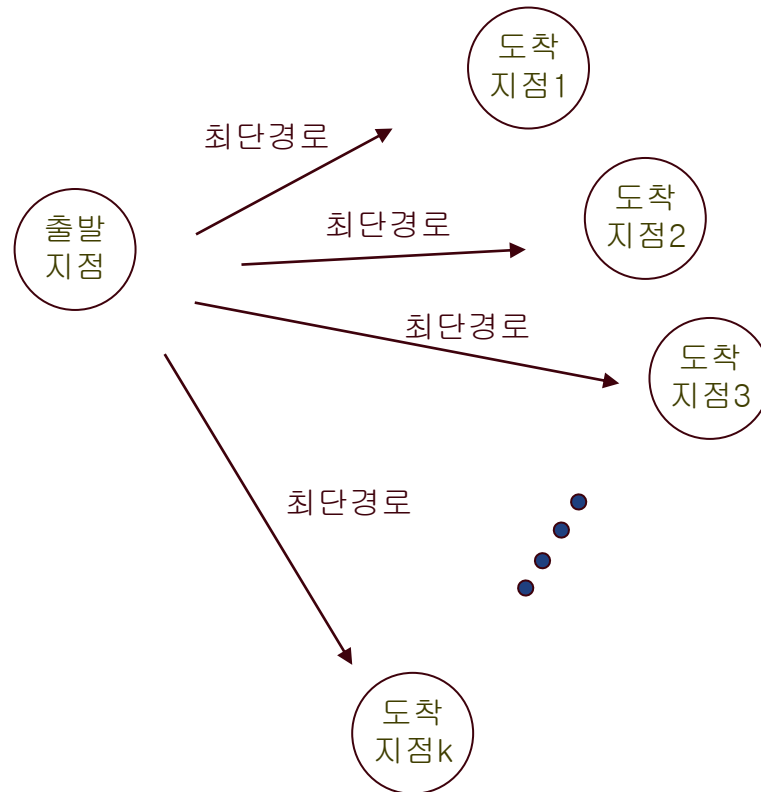
- 연결된 그래프에서의 m 은 $n-1 \leq m \leq \frac{n(n-1)}{2}$ 의 범위를 갖는다.
- sparse 그래프인 경우는 Kruskal 알고리즘이 효과적.
- dense인 경우는 Prim 알고리즘이 더 효과적

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	$\Theta(m + n \lg n)$	$\Theta(n \lg n)$	$\Theta(n^2)$

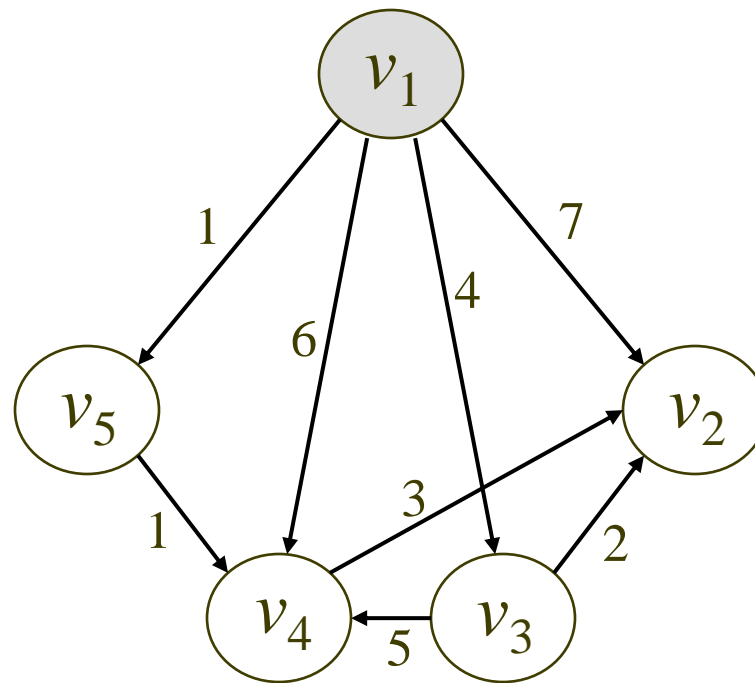
단일출발점 최단경로문제(single source shortest path problem) Dijkstra의 알고리즘(1959)



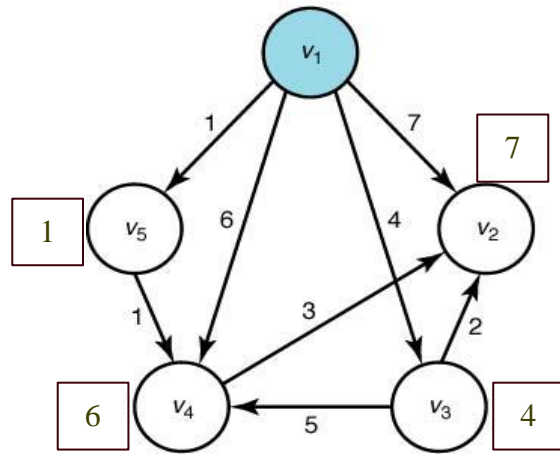
단일출발점 최단경로문제(single source shortest path problem) Dijkstra의 알고리즘(1959)

- 가중치가 있는 방향성 그래프에서 한 특정 정점에서 다른 모든 정점으로 가는 최단경로 구하는 문제.
- 알고리즘:
 1. $F := \phi$;
 2. $Y := \{v_1\}$;
 3. 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복
 - (a) 선정 절차/적정성 점검: $V - Y$ 에 속한 정점 중에서, v_1 에서 Y 에 속한 정점 만을 거쳐서 최단경로가 되는 정점 v 를 선정
 - (b) 그 정점 v 를 Y 에 추가.
 - (c) v 에서 F 로 이어지는 최단경로 상의 이음선을 F 에 추가.
 - (d) 해답 점검: $Y = V$ 가 되면, $T = (V, F)$ 가 최단경로를 나타내는 그래프

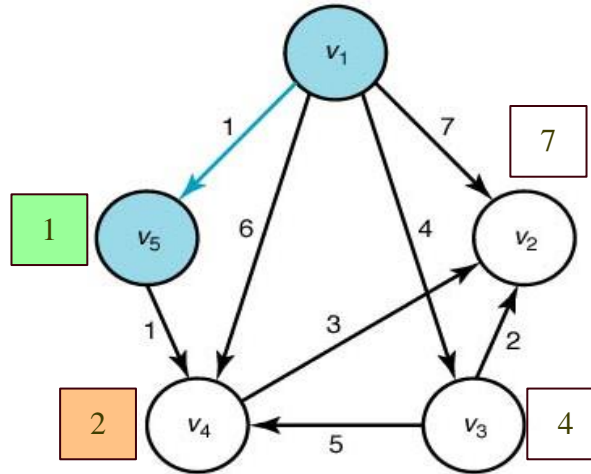
예



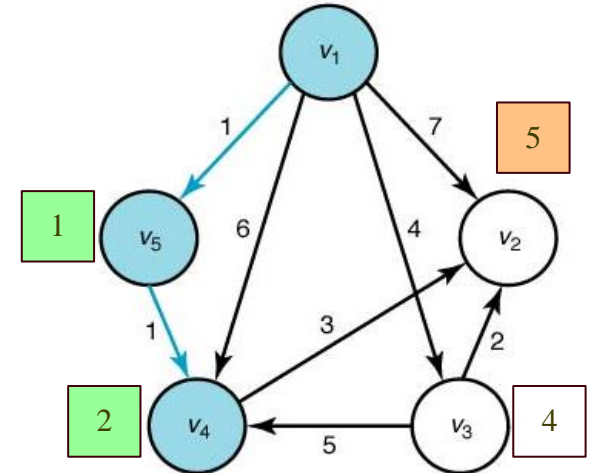
Compute shortest paths from v_1 .



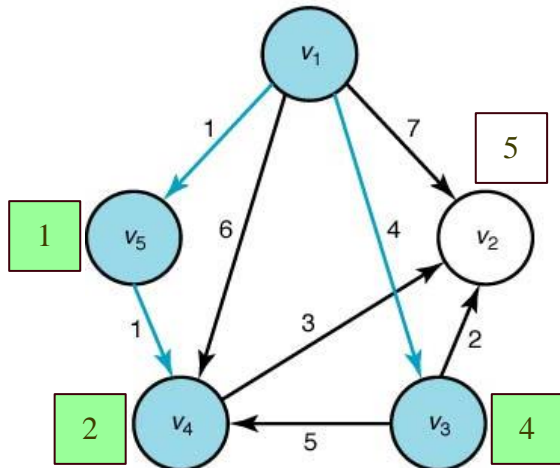
1. Vertex v_5 is selected because it is nearest to v_1 .



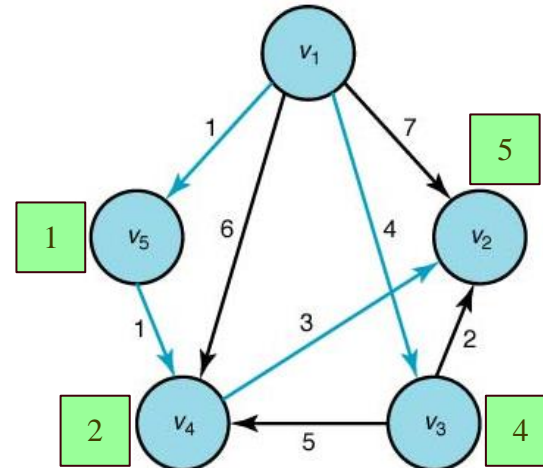
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



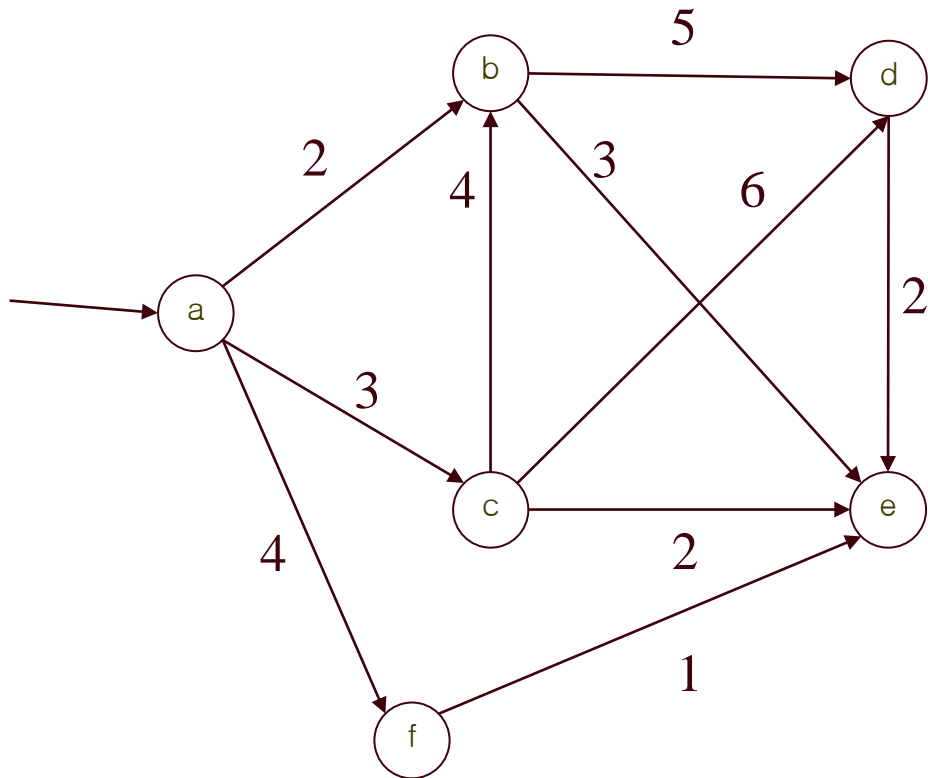
3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.

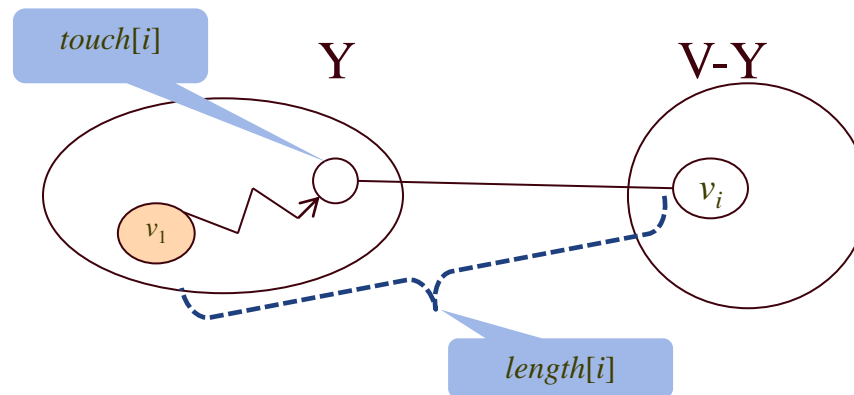


(ex)



Dijkstra의 알고리즘

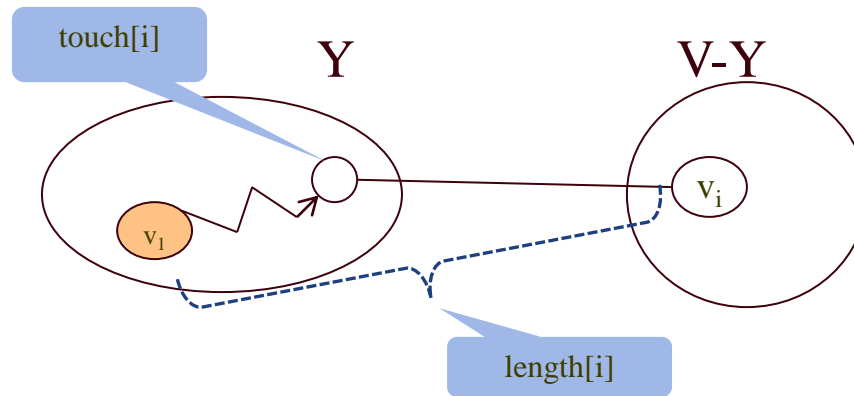
- 추가적으로 $touch[1..n]$ 과 $length[1..n]$ 배열 유지
 - ✓ $touch[i] = Y$ 에 속한 정점들만 중간에 거치도록 하여 v_1 에서 $v_i \in V - Y$ 로 가는 현재 최단경로상의 마지막 이음선을 $\langle v, v_i \rangle$ 라고 할 때, Y 에 속한 정점 v
 - ✓ $length[i] = Y$ 에 속한 정점들만 중간에 거치도록 하여 v_1 에서 v_i 로 가는 현재 최단경로의 길이



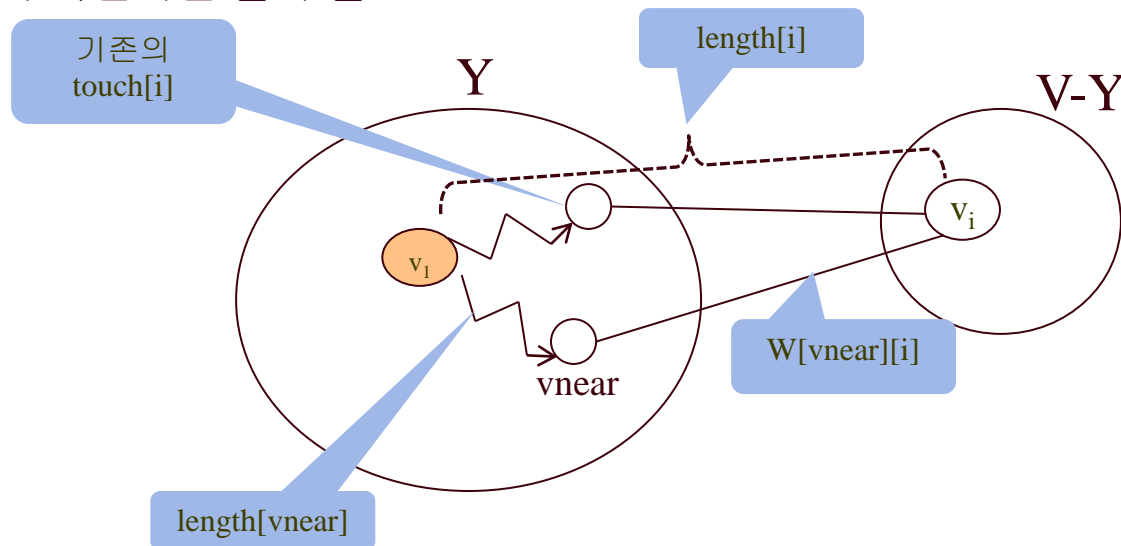
```

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++) {
        touch[i] = 1;
        length[i] = W[1][i];
    }
    repeat(n-1 times) {
        min =  $\infty$ ;
        for(i=2; i <= n; i++)
            if (0 <= length[i] < min) {
                min = length[i];
                vnear = i;
            }
        e = (touch[vnear], vnear): 이음선;
        e를 F에 추가;
        for(i=2; i <= n; i++)
            if (length[vnear] + W[vnear][i] < length[i]) {
                length[i] = length[vnear] + W[vnear][i];
                touch[i] = vnear;
            }
        length[vnear] = -1;
    }
}

```



- $V-Y$ 에 있는 노드 중 $length[i]$ 를 최소로 하는 i 를 $vnear$ 로 선정
- $vnear$ 가 Y 에 편입
- $V-Y$ 에 대해 $length$ 배열을 재 계산: $vnear$ 가 Y 에 새로 편입되면서 $length[i]$ 가 감소할 수 있는가 확인하는 절차 필요



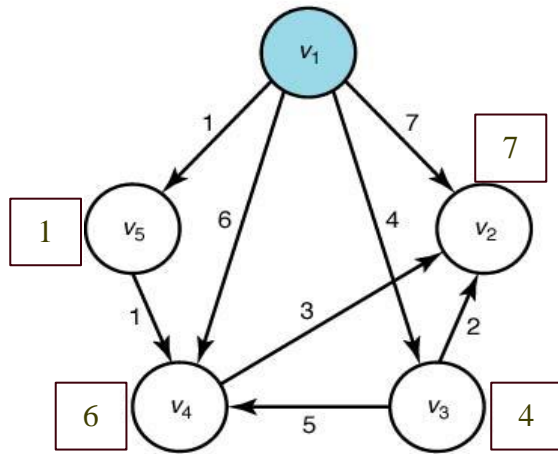
Dijkstra의 알고리즘 분석

- 분석

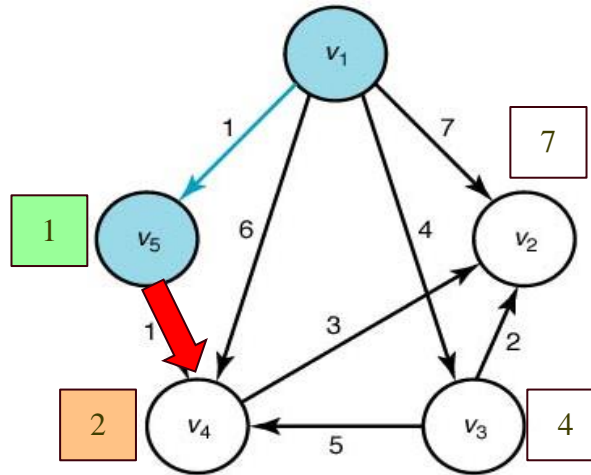
- ✓ $T(n) = 2(n-1)^2 \in \Theta(n^2)$.
- ✓ 힙(heap)으로 구현하면 $\Theta(m \lg n)$ 이고, 피보나찌 힙으로 구현하면 $\Theta(m + n \lg n)$ 이다.

```
void dijkstra(int n, const number W[][], set_of_edges& F)
    for(i=2; i <= n; i++)
        repeat(n-1 times)
            for(i=2; i <= n; i++)
                .....
            for(i=2; i <= n; i++)
```

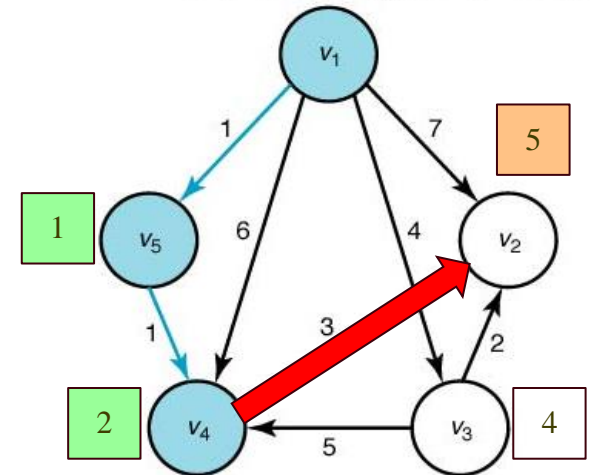
Compute shortest paths from v_1 .



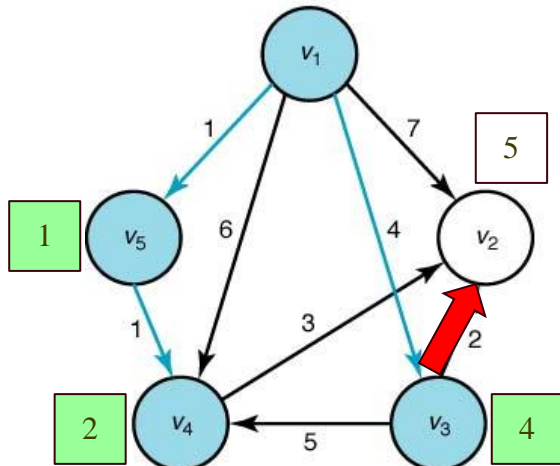
1. Vertex v_5 is selected because it is nearest to v_1 .



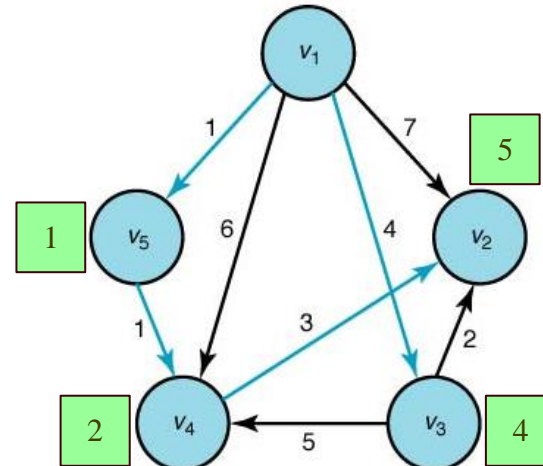
2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.



3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



update를 확인해야 하는 아크


```

void dijkstra(int n, const number W[][], set_of_edges& F) {
    for(i=2; i <= n; i++) {
        ----
    }
    repeat(n-1 times) {
        for(i=2; i <= n; i++)
            if (0 <= length[i] < min) {
                find shortest and vnear

            }
        for(i=2; i <= n; i++)
            if (length[vnear] + W[vnear][i] < length[i]) {
                update length and touch
            }
    }
}

```

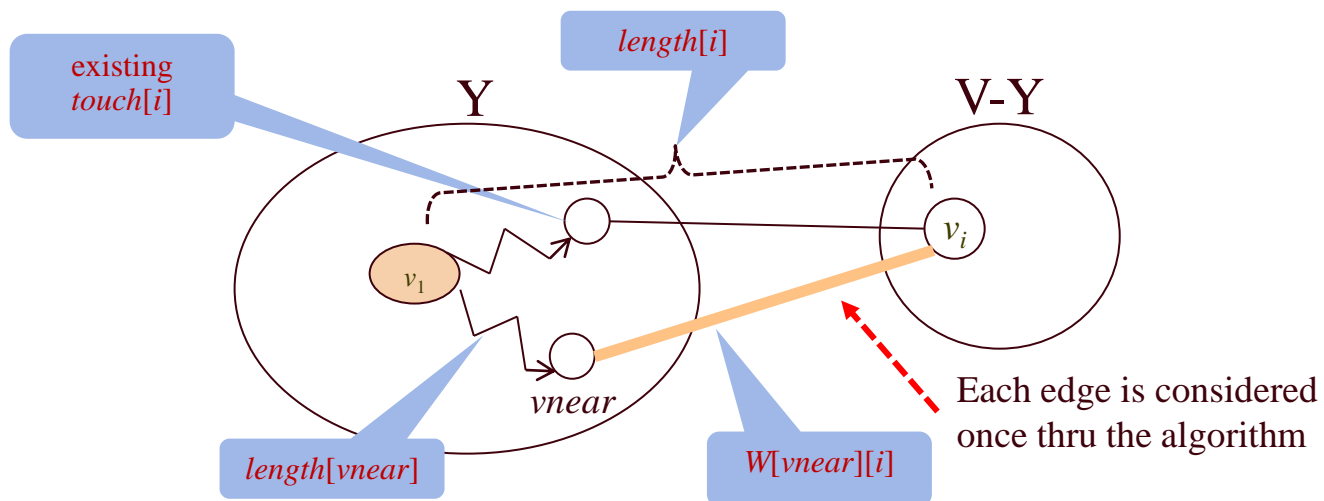
$O(n \lg n)$: n 개를 갖고 있는 min heap

✓ Consider only the edges connected to vnear.

✓ Each edge is considered once.

✓ 총 m 번의 확인 및 변경 작업 가능

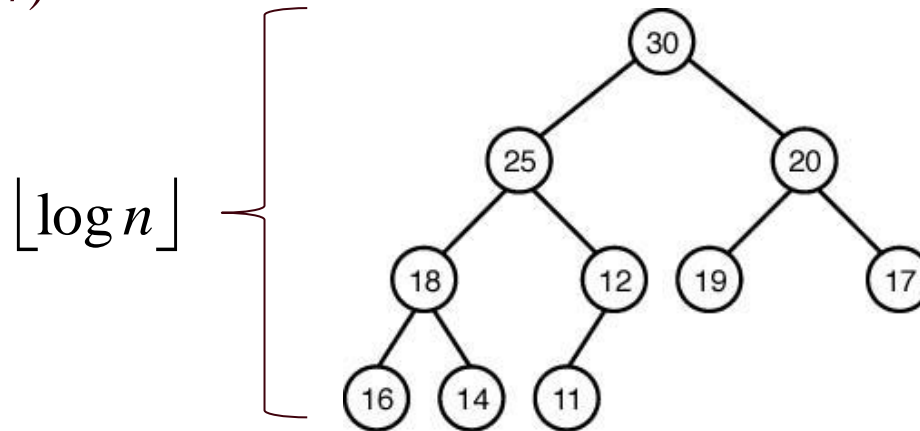
✓ 정확히는 “총에지수- v_1 에 연결된 에지수” 만큼 작업 가능



복잡도가 $\Theta(m \lg n)$ 인 이유 설명

- ✓ 그림에서 박스로 표현된 length 배열을 heap으로 구현 – 데이터 개수 $n-1$
- ✓ Min length 찾기: 데이터 n 개의 min heap에서 min 찾기를 $n-1$ 회
 - $O(n \lg n)$
- ✓ length[] update:
 - m 에 비례하는 횟수의 heap operation 가능
 - vnear의 편입에 따라 연결된 에지를 이용한 length 배열의 update 확인작업
 - 데이터의 개수가 n 개인 heap operation은 $\lg n$ 시간 필요
 - $\Theta(m \lg n)$
- ✓ $m \geq n-1$ 이므로 Dijkstra알고리즘의 복잡도는 $\Theta(m \lg n)$

- MaxHeap (Ch 7)



- Characteristics of Heap

1. Find a Max – $O(1)$
2. Remove the Max and rebuilding – $O(\log n)$
3. Add(delete or modify) a data - $O(\log n)$

- Suitable to maintain a Max of Queue – priority queue

Dijkstra의 알고리즘 분석

- 최적여부의 검증(optimality proof)
 - ✓ Prim의 알고리즘의 경우와 비슷함.

Huffman Code

1. Fixed-length binary code
2. Variable-length binary code
3. Optimal binary code
4. 전치코드: prefix code

허프만코드는 최적 이진 코드를 만듦

- Fixed-length binary code

- ✓ 코드의 길이가 고정

- (예) a:00, b:01, c:11 등

- Variable-length binary code

- ✓ 코드의 길이가 변함

- (예) a:10, b:0, c:11 등

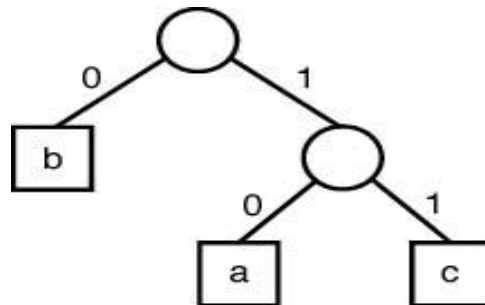
- a를 00으로 표기하면 b와 구분할 수 없음.

- Optimal binary code

- ✓ 주어진 파일에 있는 문자들을 이진코드로 표현하는데 필요한 비트의 개수가 최소가 되는 코드

- Prefix code

- ✓ 한 문자의 코드워드가 다른 문자의 코드워드의 앞부분이 될 수 없다.
(예) a:10, b:0, c:11 등
- ✓ 앞으로 읽을 비트를 확인하지 않아도 코드를 해석할 수 있음.



- ✓ abc의 prefix: a, ab, abc

(예) 파일 A = bbbaac

코드	c1	c2	c3
a	00	00	10
b	01	01	0
c	10	1	11
총 비트 수	12	11	9

c1: 010101000010 $2 \times 6 = 12$ (prefix 코드)

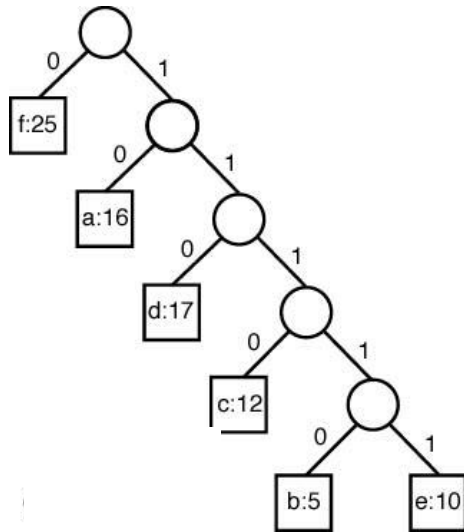
c2: 01010100001 $2 \times 3 + 2 \times 2 + 1 = 11$ (prefix 코드)

c3: 000101011 $1 \times 3 + 2 \times 2 + 2 = 9$ (prefix 코드: Huffman 코드)

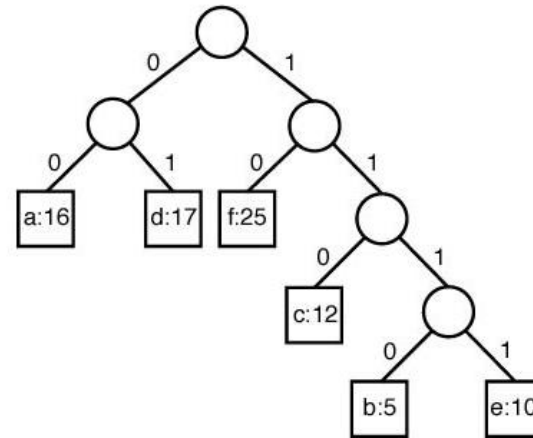
* c3가 파일 A에 대해서는 최적코드

$$\text{총비트수 } bits(T) = \sum_{i=1}^n frequency(v_i) \times depth(v_i)$$

(A)



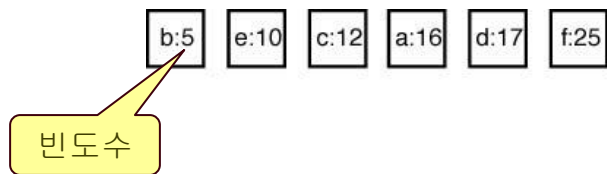
(B)



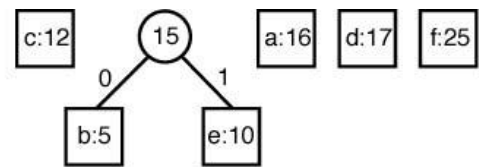
prefix 코드의 예. (B)는 Huffman code

● Huffman 코드 구축 방법

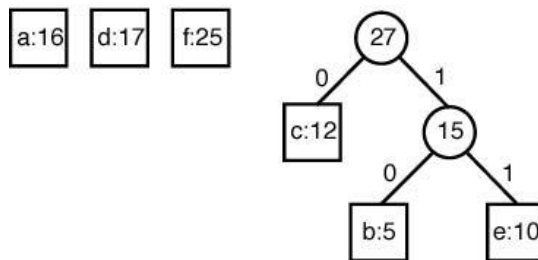
- (1) 빈도수를 데이터로 갖는 n 개의 노드를 생성
- (2) 빈도수의 합이 최소가 되는 노드를 merge 시켜 이진트리로 구축
- (3) 모든 노드가 하나의 이진트리가 될 때까지 단계(2)를 반복



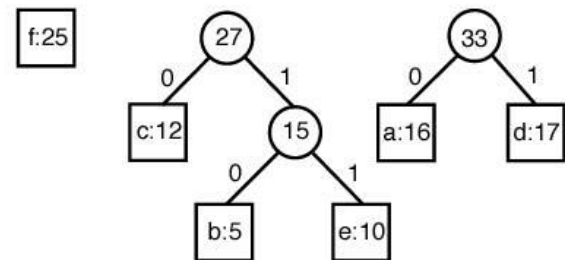
(0)



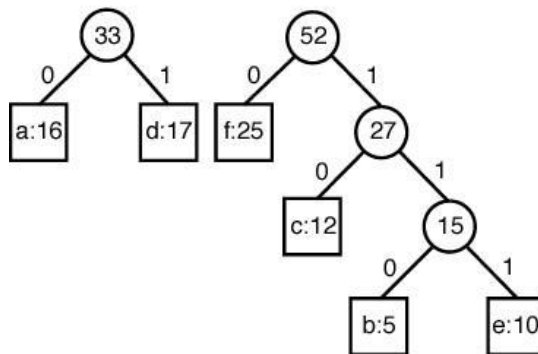
(1)



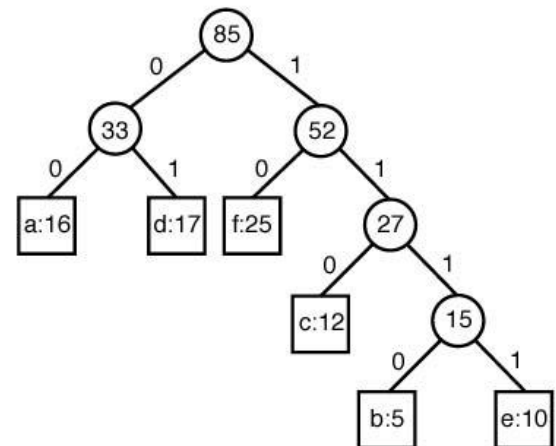
(2)



(3)



(4)



(5)

Huffman 코드 구축 단계

● Huffman code 생성 알고리즘

- 우선순위 큐(Priority Queue) 사용 - Heap
- 자료 구조

```
struct nodetype {  
    char symbol;  
    int frequency;  
    nodetype* left;  
    nodetype* right;  
};
```

● 초기

- ✓ 우선순위 큐 PQ에서 nodetype 레코드를 가리키는 포인터 n 개를 생성.
- ✓ PQ의 각 포인터 p 에 대해

$p \rightarrow \text{symbol} = \text{문자}$

$p \rightarrow \text{frequency} = \text{문자의 빈도수}$

$p \rightarrow \text{left} = p \rightarrow \text{right} = \text{NULL}$

- 빈도수가 작을수록 우선순위가 높다

```
for (i=1; i <= n-1; i++) {  
    remove(PQ, p);  
    remove(PQ, q);  
    r = new nodetype;  
    r->left = p;  
    r->right = q;  
    r->frequency = p->frequency + q->frequency;  
    insert(PQ, r);  
}  
remove(PQ, r);  
return r;
```

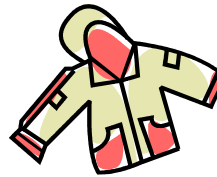
- `remove(PQ, r)` : 우선순위큐에서 최대 우선순위 데이터 `r`을 제거
- $\Theta(n \lg n)$

- Huffman 알고리즘은 최적 이진 코드를 만든다.

(ex) a:10 b:4 c:7 d:3 e:2 f: 5

0-1 Knapsack Problem

- Item partitioning is not allowed.



0-1 배낭 채우기 문제 (0-1 Knapsack Problem)

- 문제: $S = \{item_1, item_2, \dots, item_n\}$

$w_i = item_i$ 의 무게

$p_i = item_i$ 의 가치

$W =$ 배낭에 넣을 수 있는 총 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.

$$\begin{array}{ll} \text{MAX} & \sum_{item_i \in A} p_i \\ \text{subject to} & \sum_{item_i \in A} w_i \leq W \end{array}$$

or

$$\begin{array}{ll} \text{MAX} & \sum_{i=1}^n p_i x_i \\ \text{subject to} & \sum_{i=1}^n w_i x_i \leq W \\ & x_i = 0 \text{ or } 1, \text{ for } i = 1, n \end{array}$$

- 무작정 알고리즘

- ✓ n 개의 물건에 대해서 모든 부분집합을 다 고려한다.
- ✓ 그러나 불행하게도 크기가 n 인 집합의 부분집합의 수는 2^n 개 이다.

$$A = \{a, b, c\}$$

$$\text{Power set of } A = 2^A$$

$$= \{ \phi = \{\}, \{a\}, \{b\}, \{c\}, \{a,b\}, \{b,c\}, \{a,c\}, \{a,b,c\} \}$$

✓ Power set= 멱집합

탐욕적 알고리즘 1

- 가장 비싼 물건부터 우선적으로 채운다.
- 최적이 아님.
- (이유) $W = 30\text{kg}$
 - ✓ 탐욕적인 방법: $item_1 \Rightarrow 25\text{kg} \Rightarrow 10\text{만원}$
 - ✓ 최적인 해답: $item_2 + item_3 \Rightarrow 20\text{kg} \Rightarrow 18\text{만원}$

품목	무게	값
$item_1$	25kg	10만원
$item_2$	10kg	9만원
$item_3$	10kg	9만원

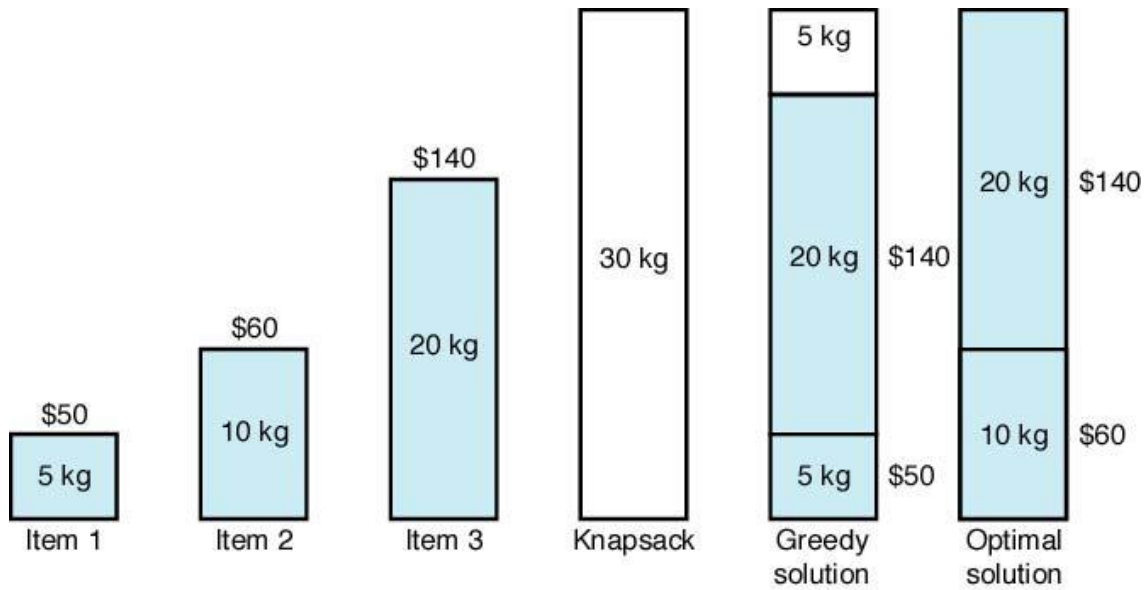
탐욕적 알고리즘 2

- 가장 가벼운 물건부터 우선적으로 채운다.
- 최적이 아님.
- (이유) 가벼운 물건이 무게에 비해 값어치가 없을 때 손해

탐욕적 알고리즘 3

- 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다.
- 최적이지 아님!
- (이유) $W = 30\text{kg}$
 - ✓ 탐욕적인 방법: $item_1 + item_3 \Rightarrow 25\text{kg} \Rightarrow 190\text{만원}$
 - ✓ 최적인 해답: $item_2 + item_3 \Rightarrow 30\text{kg} \Rightarrow 200\text{만원}$

품목	무게	값	무게당값
$item_1$	5kg	50만원	10만원/kg
$item_2$	10kg	60만원	6만원/kg
$item_3$	20kg	140만원	7만원/kg



배낭 빈틈없이 채우기 문제 (fractional knapsack problem)

- greedy algorithm
- 물건의 일부분을 잘라서 담을 수 있다.
- 탐욕적인 접근방법으로 최적해를 구하는 알고리즘을 만들 수 있다.
 - ✓ 단위무게당 값어치가 큰 것부터 넣고, 더 이상 넣을 수 없으면 배낭에 남은 무게 만큼 잘라서 넣는다.
- $item_1 + item_3 + item_2 \times 1/2 \Rightarrow 30\text{kg} \Rightarrow 220\text{만원}$

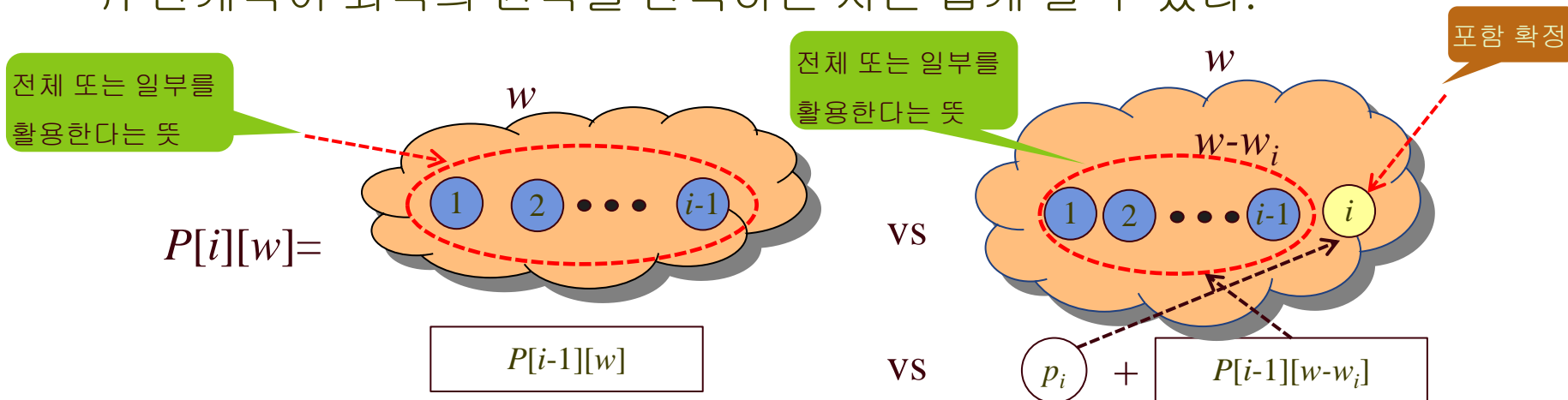
품목	무게	값	무게당값
$item_1$	5kg	50만원	10만원/kg
$item_2$	10kg	60만원	6만원/kg
$item_3$	20kg	140만원	7만원/kg

0-1 배낭채우기 문제의 동적계획적인 접근방법

- $i > 0$ 이고 $w > 0$ 일 때, 전체 무게가 w 가 넘지 않도록 i 번째 까지의 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 최적의 원칙을 만족하는 지는 쉽게 알 수 있다.



i 번째 item을 넣지 않을 것인가 또는 넣을 것인가?

0-1 배낭채우기 문제의 동적계획적인 접근방법

- 그러면 어떻게 $P[n][W]$ 값을 구할 수 있을까?

다음과 같은 2차원 배열을 만든 후, 각 항을 계산하여 채워 넣으면 된다:

- ✓ $\text{int } P[0..n][0..W]$. 여기서 $P[0][w] = 0, P[i][0] = 0$
- ✓ 계산해야 할 항목의 수는 $nW \in \Theta(nW)$

- 최대 이익은

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

```

knapsack(p,w,n,W) {
  for(w=0 to W) P[0,w]=0
  for(i=0 to n) P[i,0]=0

  for(i=1 to n)
    for(w =0 to W)
      if( $w_i \leq w$ )
         $P[i,w] = \max \{ P[i-1, w], p_i + P[i-1, w - w_i] \}$ 
      else
         $P[i,w] = P[i-1, w];$ 

  return P[n,W]
}

```

(예) $W=10$

품목	무게	값
$item_1$	5kg	10만원
$item_2$	4kg	40만원
$item_3$	6kg	30만원
$item_4$	3kg	50만원

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

$P[i,w]$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

최대 가치

$\max\{10, 40+0\}$

$P[1,5]$

$p_2 + P[1,1]$

$\max\{70, 50+40\}$

$P[3,10]$

$p_4 + P[3,7]$

0-1 배낭채우기 문제의 동적계획적인 접근방법

- 여기서 n 과 W 와는 아무런 상관관계가 없다. 따라서, $W = n!$ 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. 즉, $P[n-1][W]$ 와 $P[n-1][W-w_n]$ 두 항만 계산하면 된다.

$(n-1)^{\text{st}}$ row	$P[n-1][W-w_n]$	$P[n-1][W]$
n^{th} row			$P[n][W]$

이런 식으로 $n = 1$ 이나 $w \leq 0$ 일 때 까지 계속해 나가면 된다.

0-1 배낭채우기 문제의 동적계획적인 접근방법

위의 예에서 $P[3][30]$ 을 계산해 보자. 개량 알고리즘은 다음과 같이 7개 항만 계산하는데 비해서, 이전 알고리즘은 $3 \times 30 = 90$ 항을 계산해야 한다. $W=30$.

$$P[3][30] = \max(P[2][30], 140 + P[2][10]) = \max(110, 140 + 60) = 200$$

$$P[2][30] = \max(P[1][30], 60 + P[1][20]) = \max(50, 60 + 50) = 110$$

$$P[2][10] = \max(P[1][10], 60 + P[1][0]) = \max(50, 60 + 0) = 60$$

$$P[1][0] = 0$$

$$P[1][10] = 50$$

$$P[1][20] = 50$$

$$P[1][30] = 50$$

품목	무게	값	무게당 값
$item_1$	5kg	50만원	10만원/kg
$item_2$	10kg	60만원	6만원/kg
$item_3$	20kg	140만원	7만원/kg

P	0	10	20	30
1	0	50	50	50
2		60		110
3				200

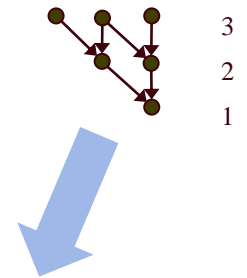
0-1 배낭채우기 문제의 동적계획적인 접근방법

- 그러면 개량 알고리즘의 최악의 경우 수행시간을 계산해 보자. $(n - i)$ 번째 행에서 기껏해야 2^i 항을 계산하므로, 총 계산하는 항 수는 $1 + 2 + 2^2 + \dots + 2^{n-1} = 2^n - 1$ 이 된다. 따라서 $\Theta(2^n)$ 가 된다.

- 만일 $n=W+1$, 모든 i 에 대해 $w_i=1$ 이면 계산하는 엔트리의 총 수는

$$1+2+3+\dots+n = n(n+1)/2 = (W+1)(n+1)/2 \in \Theta(nW)$$

그러므로 P 행렬의 엔트리 수 $\in O(nW)$.



- 위의 두 가지 경우를 합하면 최악의 경우의 수행시간은

$O(\min(2^n, nW))$ 이다.

	0	1	...	W-2	W-1	W
1						
...						
n-2				x	x	x
n-1					x	x
n						x

● 분할정복 방법으로 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.

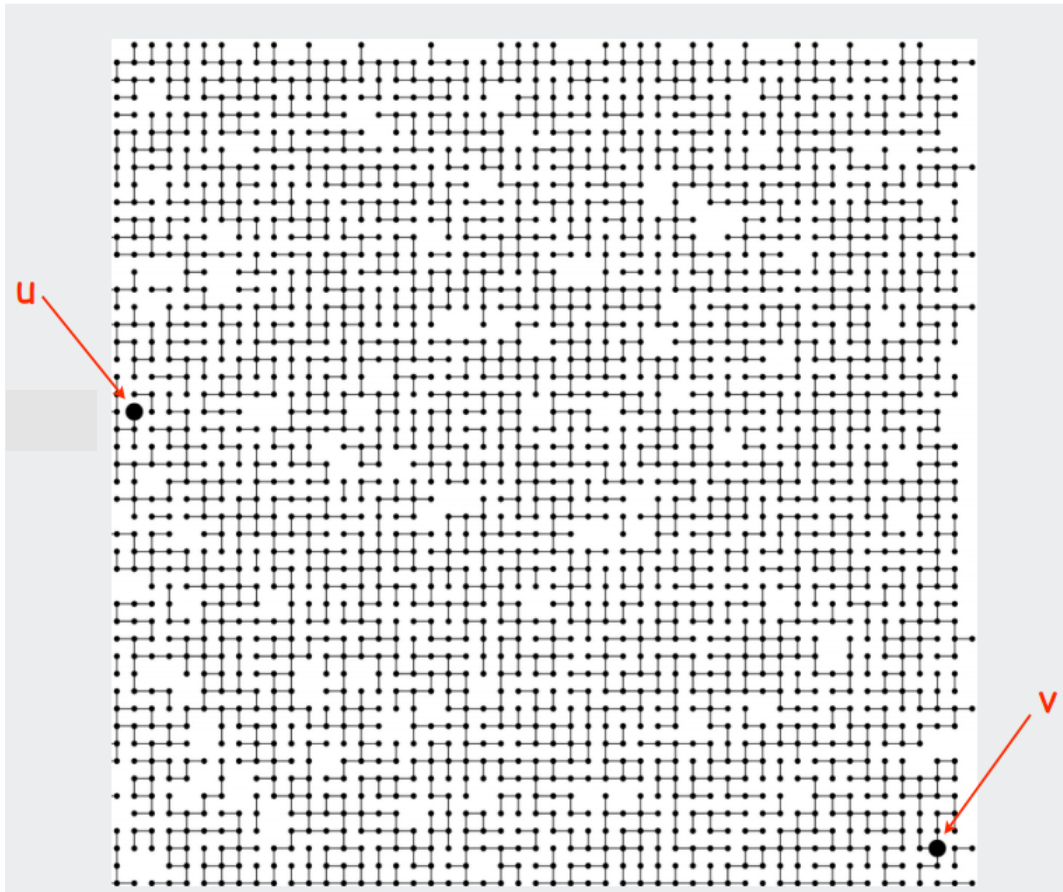
– NP문제 (9장)

탐욕적인 방법과 동적계획법의 비교

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

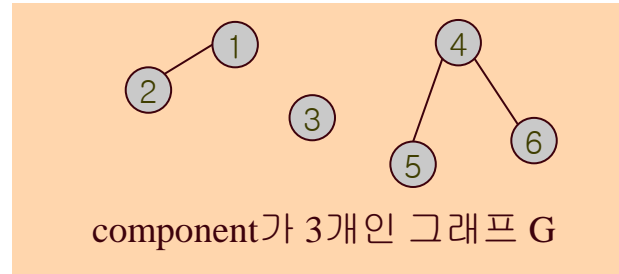
상호배타적 집합의 처리 (Disjoint Sets Algorithm)

- 응용
 - ✓ 인터넷 상의 웹페이지 관계
 - ✓ alias 처리
 - ✓ 컴퓨터 네트워크



- u 와 v 가 같은 component에 있는가?

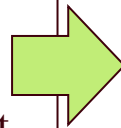
1. linked list 사용 방법
2. array 사용 방법
3. tree(forest) 사용 방법
 - a. 단순 방법
 - b. 무게(weight)고려 방법
 - c. path compression 방법



→ {1,2}, {3},
{4,5,6}

문제:

- ✓ 노드 1은 어느 component에 있나?
- ✓ 노드 1과 3은 연결되어 있나?
- ✓ 노드 1과 3이 속해 있는 component를 하나의 component로 만들어라



해결방법:

- ✓ find(1)
- ✓ find(1) == find(3)?
- ✓ union(1,3)

- 상호배타적(disjoint): 집합간에 교집합은 ϕ

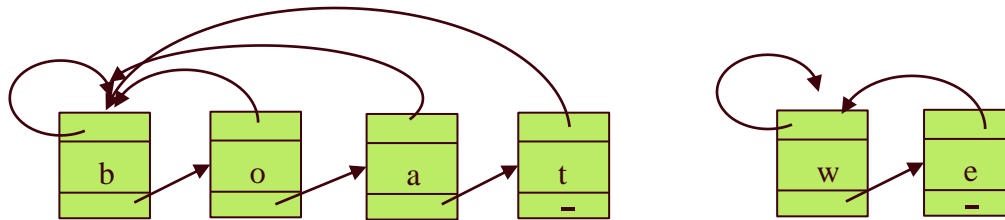
- Operations

- ✓ `make_set(v)`: make a set with an element v
- ✓ `find(v)`: returns a name of the set containing v
- ✓ `union(u, v)`: 원소 u 와 원소 v 를 갖고 있는 집합을 하나로 합친다. 이미 같은 집합에 속해 있을 수 있다.

- 문제

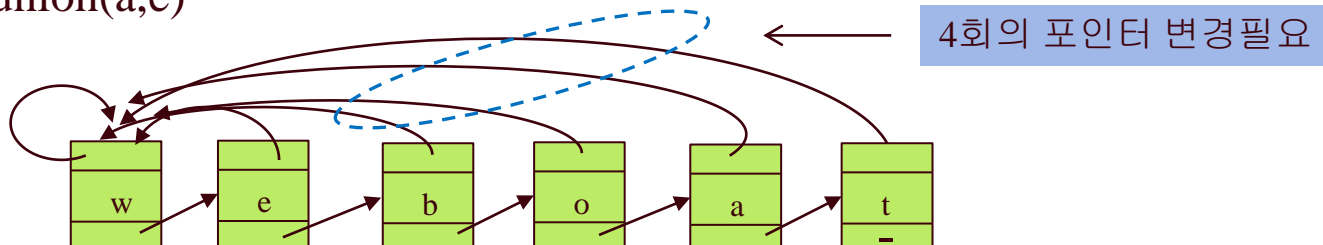
- ✓ `find`, `union`의 명령어가 다수 있을 때 효율적인 처리가 가능한 자료구조와 알고리즘을 개발한다.

1. linked list를 사용하는 방법



- set의 이름은 최초 원소값
- 각 원소는 set이름으로 가는 포인터와 next 포인터를 갖는다.
- make_set(v): $O(1)$
- find(v): $O(1)$
- union(u,v): u가 속해 있는 리스트의 set이름으로 가는 포인터를 v가 속해있는 set의 이름으로 바꾸어 준다 $u \rightarrow v$. (반대도 가능)

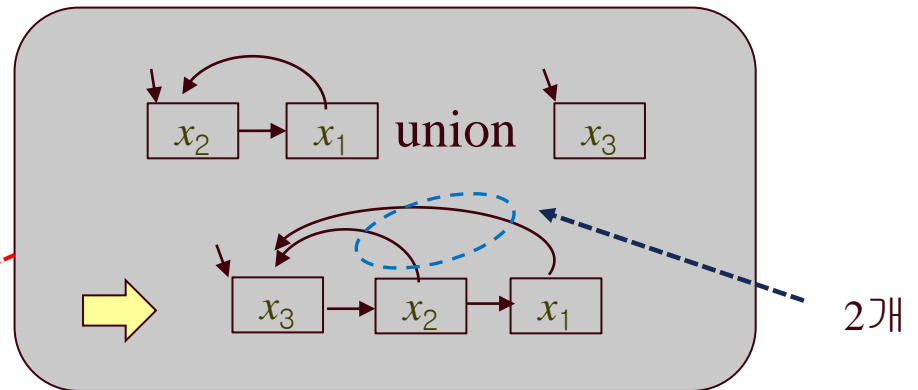
(예) union(a,e)



commands	업데이트되는 포인터의 개수
make_set(x_1)	1
make_set(x_2)	1
.....	...
make_set(x_n)	1
union(x_1, x_2)	1
union(x_2, x_3)	2
.....	...
union(x_{n-1}, x_n)	$n-1$

$O(n^2)$

k 크기의 리스트가 1크기의
리스트에 연결되는 현상

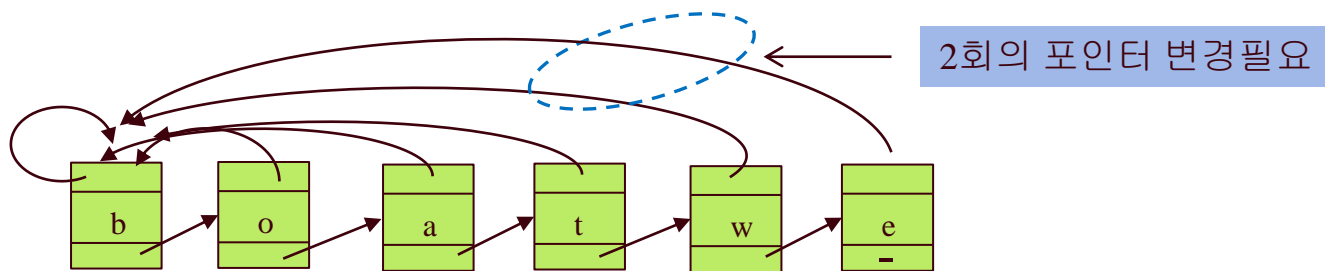
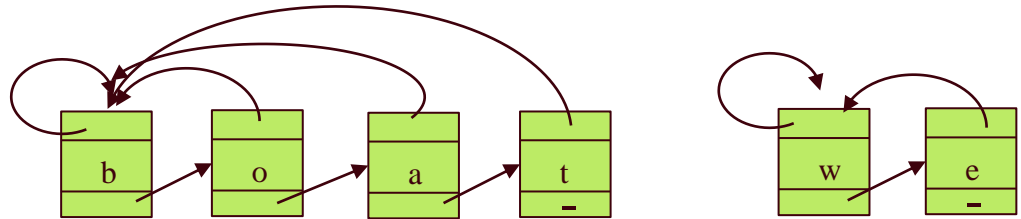


- n 개의 데이터에 m 번의 union은 $O(m^2)$ 소요 ($m < n$)

- 개선점:

리스트의 크기를 별도로 저장하고, 크기가 작은 리스트를 큰 리스트에 연결한다. weighting-union heuristic

(예) union(a,e)



weighting-union heuristic 사용 시

- n 개의 원소가 있을 때, 개선된 방법으로 $n-1$ 번의 union을 수행하는데 $O(n \lg n)$ step 소요

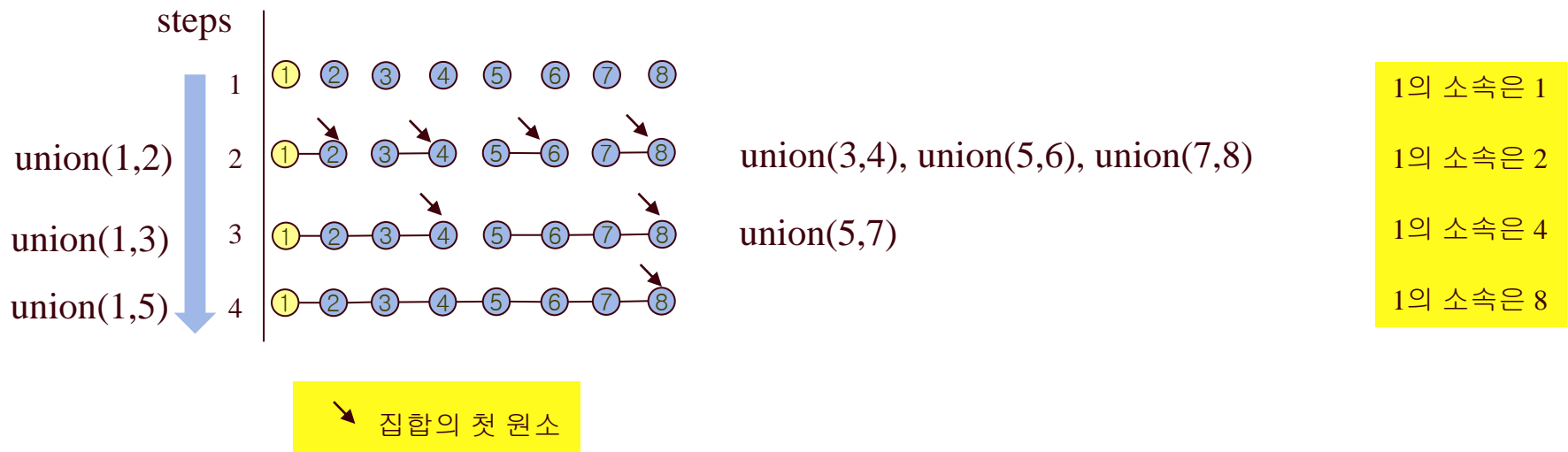
(proof)

- ✓ A가 B가 합쳐져 B가 되었을 때, 새로 만들어진 B에 대해 $|B| \geq 2|A|$, 여기서 $|A|$ 는 리스트 A의 크기
- ✓ 따라서 어떠한 원소도 $\lg n$ 번 보다 더 많이 이동할 수 없다.
- ✓ 한 원소에 대해 $O(\lg n)$
- ✓ 따라서 총 $O(n \lg n)$

weighting-union heuristic 사용 시

● 8개의 데이터가 있을 때 하나의 특정 노드가 소속을 최대로 바꾸는 경우

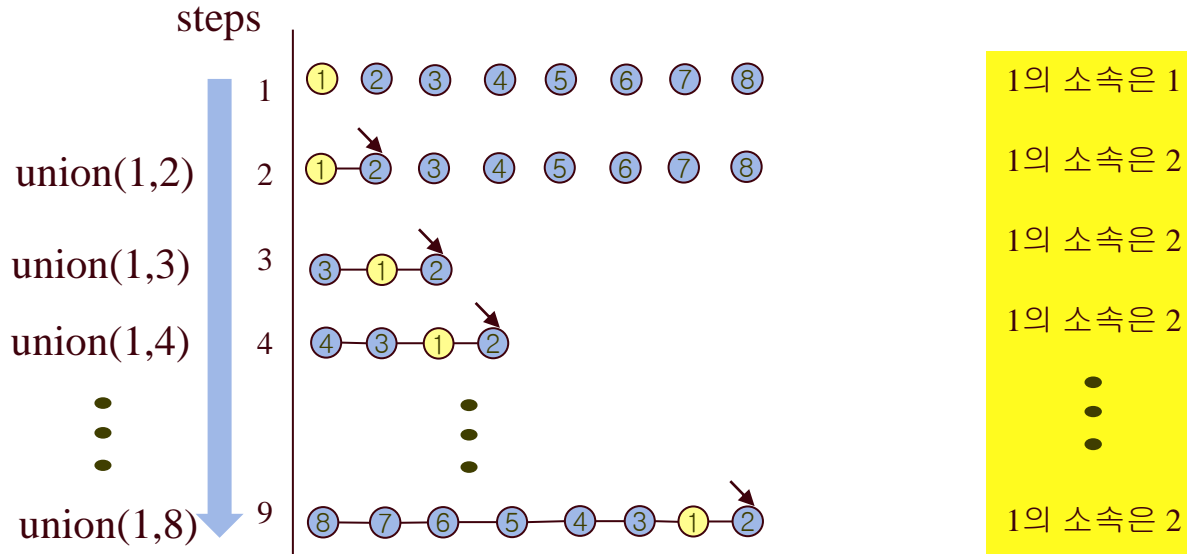
✓ 크기가 동일한 집합의 union 인 경우는, 앞집합을 뒤집합에 결합



✓ 1번 원소는 최대 3회 포인터가 바뀐다.

weighting-union heuristic 사용 시

$\text{union}(1,2), \text{union}(1,3), \text{union}(1,4), \dots, \text{union}(1,8)$ 이면 1번 원소는 단 1회의 포인터 변경 발생



2. array를 사용하는 방법

i	1	2	3	4	5	6
set_id[i]	1	2	3	1	2	2

{1,4}, {2,5,6}, {3}

- find(x) : set_id[x]만 확인하면 됨
- union(x, y): set_id[y]를 갖고 있는 원소들의 set_id로 set_id[x]를 변경

$x \rightarrow y$

- union(1,2)

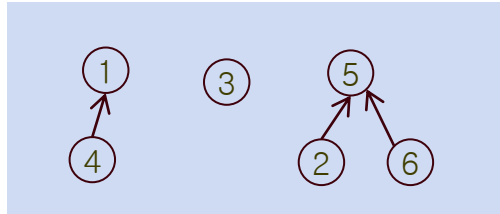
i	1	2	3	4	5	6
set_id[i]	2	2	3	2	2	2

{2,5,6,1,4}, {3}

- ✓ 리스트를 사용할 때와 마찬가지로 weighting-union heuristic을 사용하면 시간 단축
- ✓ 리스트와 같은 시간 소요

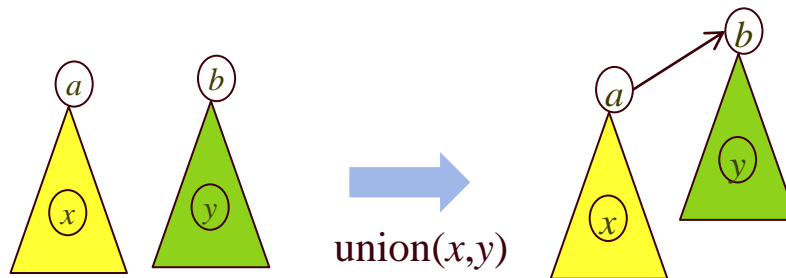
3. tree를 사용하는 방법

(a) 단순한 방법

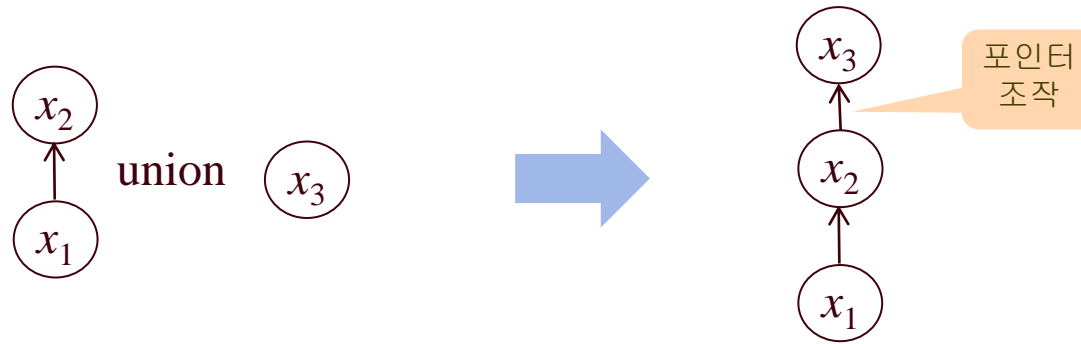


$\{1,4\}, \{3\}, \{2,5,6\}$

- 하나의 component는 하나의 트리로 표현
- 전체 그래프가 전체(모든) 집합을 나타냄
- root원소가 집합의 이름을 나타냄
- 각 원소는 parent로 향하는 포인터를 갖는다.
- $\text{find}(x)$: x 에서 출발하여 parent로 향하는 포인터를 따라 root로 이동하여, root값을 반환
- $\text{union}(x,y)$: $\text{find}(x)$ 원소를 $\text{find}(y)$ 의 child node로 만든다.



● $\text{union}(x_1, x_3)$



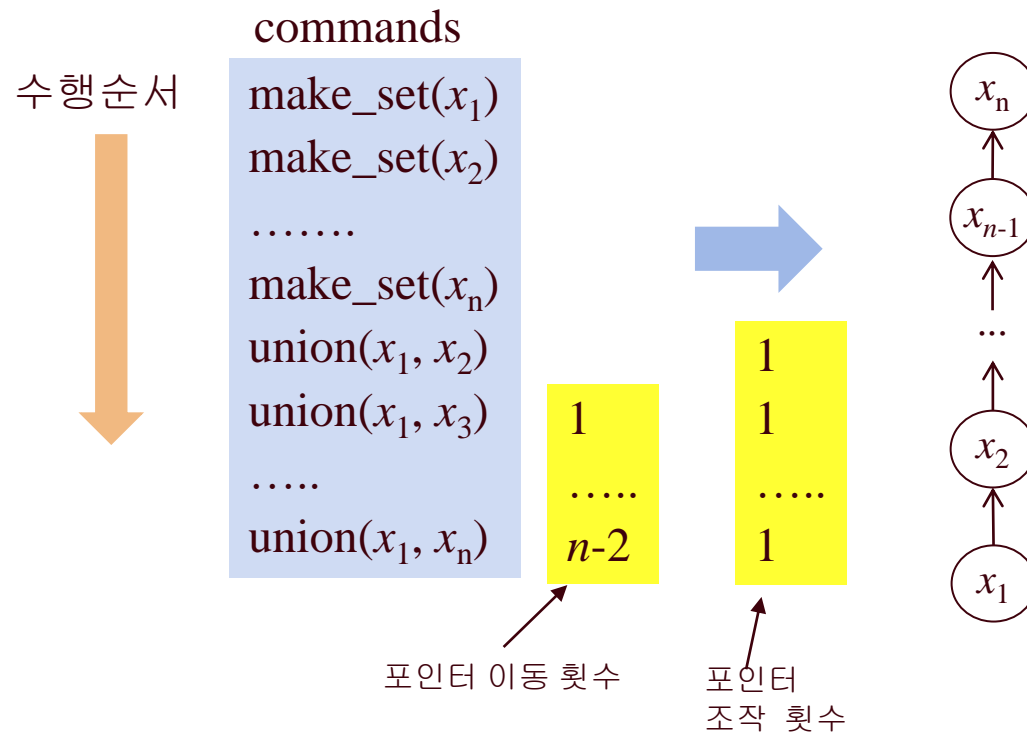
✓ $\text{union}(x_1, x_3)$ 은 $\text{find}(x_1)$, $\text{find}(x_3)$ 을 포함

↑
포인터 이동 1회 필요

↑
포인터 이동 0회 필요

✓ $\text{union}(x_1, x_3)$ 을 완료하기 위해 한 번의 포인터 조작 필요

최종 생성 트리(집합 모양)

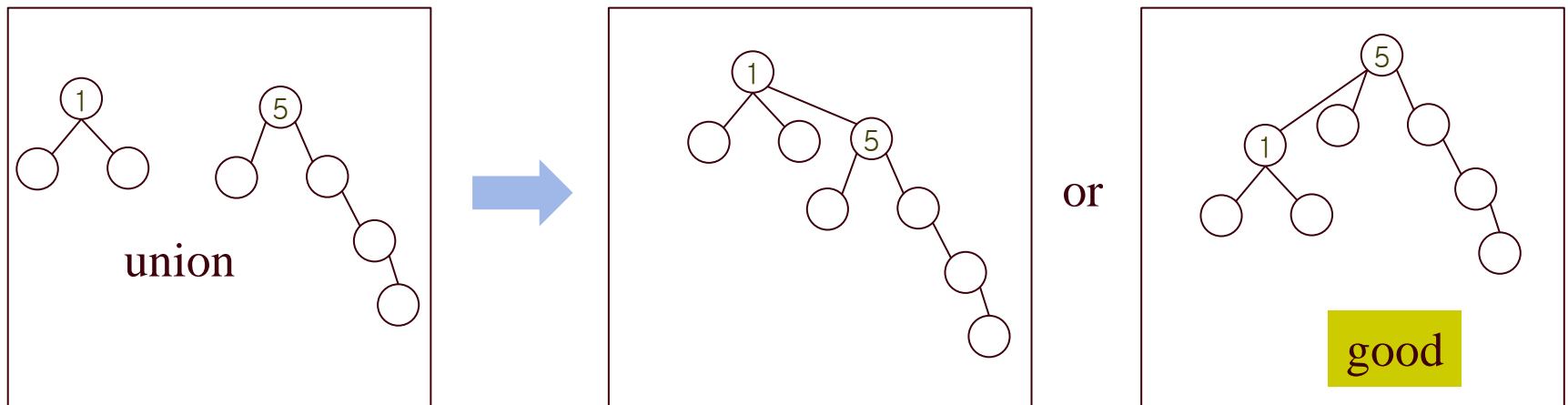


- 포인터 이동 횟수 = $1 + 2 + 3 + \dots + (n-2)$
- $O(n^2)$ 시간 필요

(b) 트리의 무게를 고려한 방법

- 개선점

- ✓ 트리가 갖고 있는 원소의 수(weight)를 저장한다.
- ✓ 트리의 weight가 작은 것을 weight가 큰 트리에 연결한다.
- ✓ 트리의 높이(height)를 이용하여 트리의 rank 정의. rank를 활용해도 같은 효과



- ✓ 결과 트리의 높이가 낮은 것이 유리
- ✓ find 를 빨리 수행하는 것이 검색하는데 바람직
- ✓ 이 방법을 사용할 경우 n 개의 원소를 갖고 있을 때, forest에 있는 어떠한 트리도 높이가 $\lg n$ 을 넘을 수 없다.
- ✓ $O(n)$ union-find operation은 최대 $O(n \lg n)$ 시간 소요

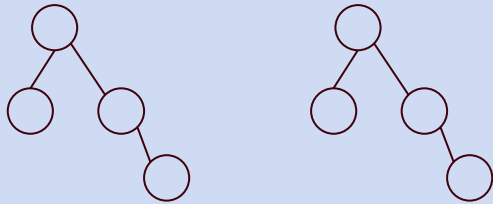
n 개의 singleton disjoint set에 대한 m 번의 union, find가 있을 경우

[예] 8개의 disjoint set에서 출발하여 다수의 union을 할 때 만들 수 있는 최고 높이의 트리 – 무게 고려

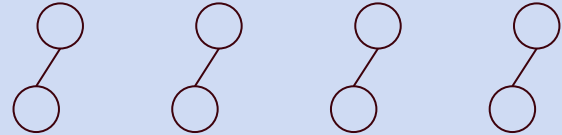
초기



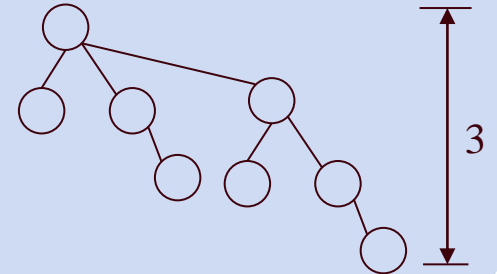
2회 union



4회 union



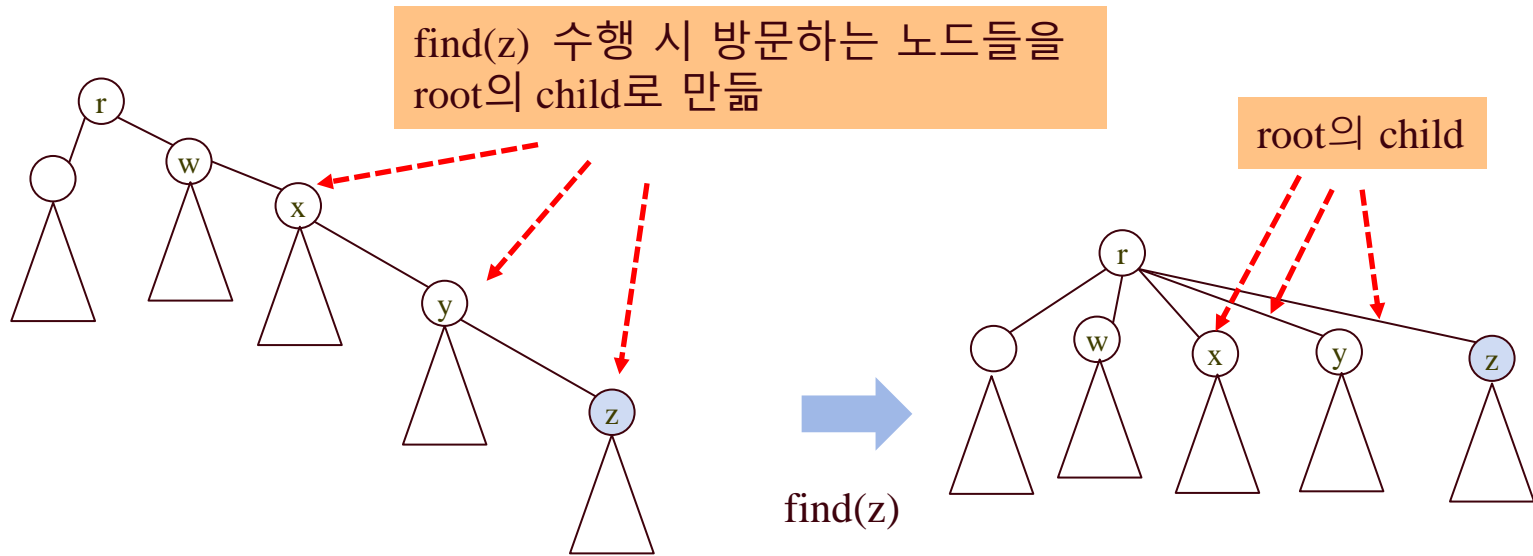
1회 union



- n 개의 원소가 있을 때 최종 트리최대높이 = $\lg n$
- 따라서 m 회의 union, find의 시간은 $O(m \lg n)$ 이다.

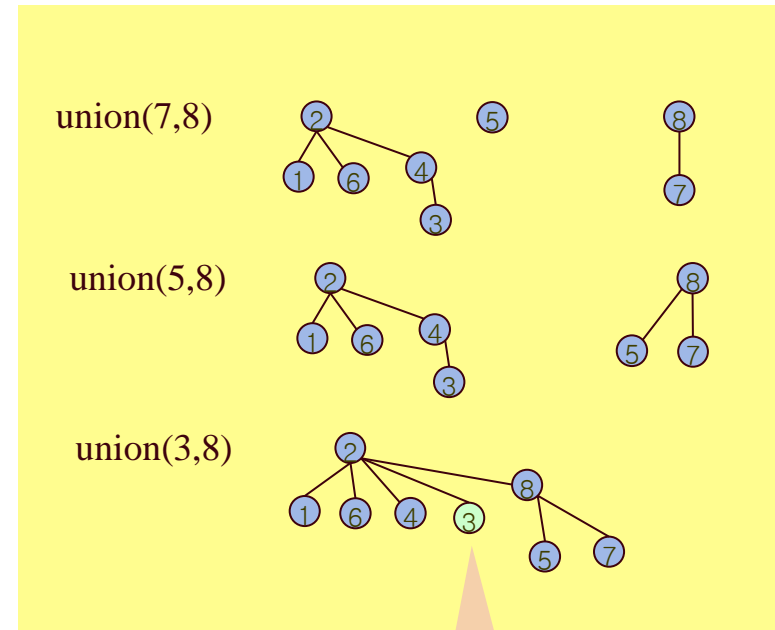
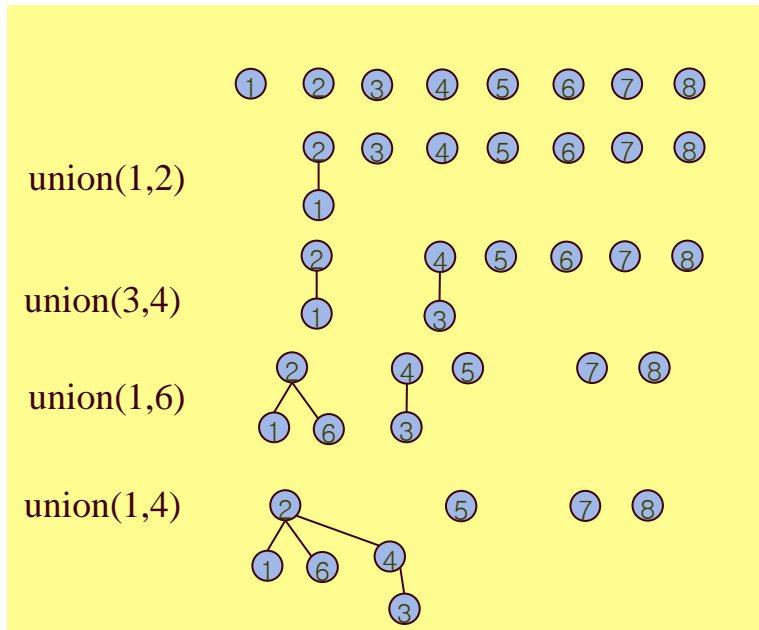
(c) path compression 방법

find(z) 명령어가 수행될 경우



- union with weighting-union and path compression

union(1,2) union(3,4) union(1,6) union(1,4) union(7,8) union(5,8) union(3,8)



path
compression
수행 후

- union(3,8)은 find(3)을 포함.
- find(3)에 의해 노드 3은 root의 child 노드가 된다

$$F(0) = 1,$$

$$F(i) = 2^{F(i-1)}, \text{ for } i > 0.$$

n	$F(n)$
0	1
1	2
2	4
3	16
4	65,536
5	$2^{65,536}$

n	$G(n)$
1	0
2	1
3~4	2
5~16	3
17~65,536	4
65,537~ $2^{65,536}$	5

- $G(n) = \min\{ k \mid F(k) \geq n \}$
 - ✓ $G(100)=4$
- G grows extremely slowly.
- $G(n) \leq 5$ for all “practical” value of n , i.e., for all $n \leq 2^{65536}$.
- ✓ n *union* and *find* operations take at most $O(nG(n))$ with path compression and weighting union heuristic.
- ✓ superlinear

Kruskal의 알고리즘의 분석

- ✓ 단위연산: find, equal, merge 내에 있는 포인터 이동 또는 데이터 비교문

- ✓ 입력크기: 정점의 수 n 과 에지의 수 m

1. 에지들을 정렬하는데 걸리는 시간: $\Theta(m \lg m)$

2. n 개의 disjoint set을 초기화하는데 걸리는 시간: $\Theta(n)$

3. 반복문 안에서 걸리는 시간: 루프를 최대 m 번 수행한다. disjoint set data structure를 사용하여 구현하고, 1회 반복에서 find, equal, merge(union) 같은 동작을 호출하는 횟수가 상수이면, m 번 반복에 대한 시간복잡도는 $\Theta(m \lg n)$ (or $mG(m)$)이다.

트리사용, weighting union heuristic, path compression 사용

```
void kruskal (.....)
{
    E에 속한 m개의 이음선을 가중치의 비내림차순으로 정렬;
    F =  $\phi$ ;
    initial(n);
    while (F에 속한 이음선의 개수가 n-1보다 작다) {
        . . . .
        p = find(i);
        q = find(j);
        if (!equal(p,q)) {
            merge(p,q);
            e를 F에 추가;
        }
    }
}
```

트리사용, weighting union heuristic 사용

- ✓ $m \geq n - 1$ 이기 때문에, 위의 1과 3은 2를 지배하게 되므로, $W(m, n) \in \Theta(m \lg m)$



4장 끝

수고하셨습니다.