



알고리즘 설계 – 3장

탐욕적인 방법
(Greedy Method)



목차

- 탐욕적인 방법의 소개
- 최소비용 신장 트리
 - Prim의 방법
 - Kruskal의 방법
- 단일 출발점 최단 경로 문제
- 0-1 배낭 채우기 문제



1. 탐욕적인 방법의 소개

- 결정을 해야 할 때마다 그 순간에 가장 좋다고 생각되는 것을 해답으로 선택함으로써 최종적인 해답에 도달
- 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 방법이 항상 최적의 해답을 주는지를 반드시 검증



탐욕적인 알고리즘의 설계 절차

- 선정 과정(**selection procedure**)
 - 현재 상태에서 가장 좋으리라고 생각되는(greedy) 해답을 찾아서 해답모음(solution set)에 포함
- 적정성 점검(**feasibility check**)
 - 새로 얻은 해답모음이 적절한지를 결정
- 해답 점검(**solution check**)
 - 새로 얻은 해답모음이 최적의 해인지를 결정



보기: 거스름돈 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 내림차순으로 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)!



최적의 해를 얻지 못하는 경우

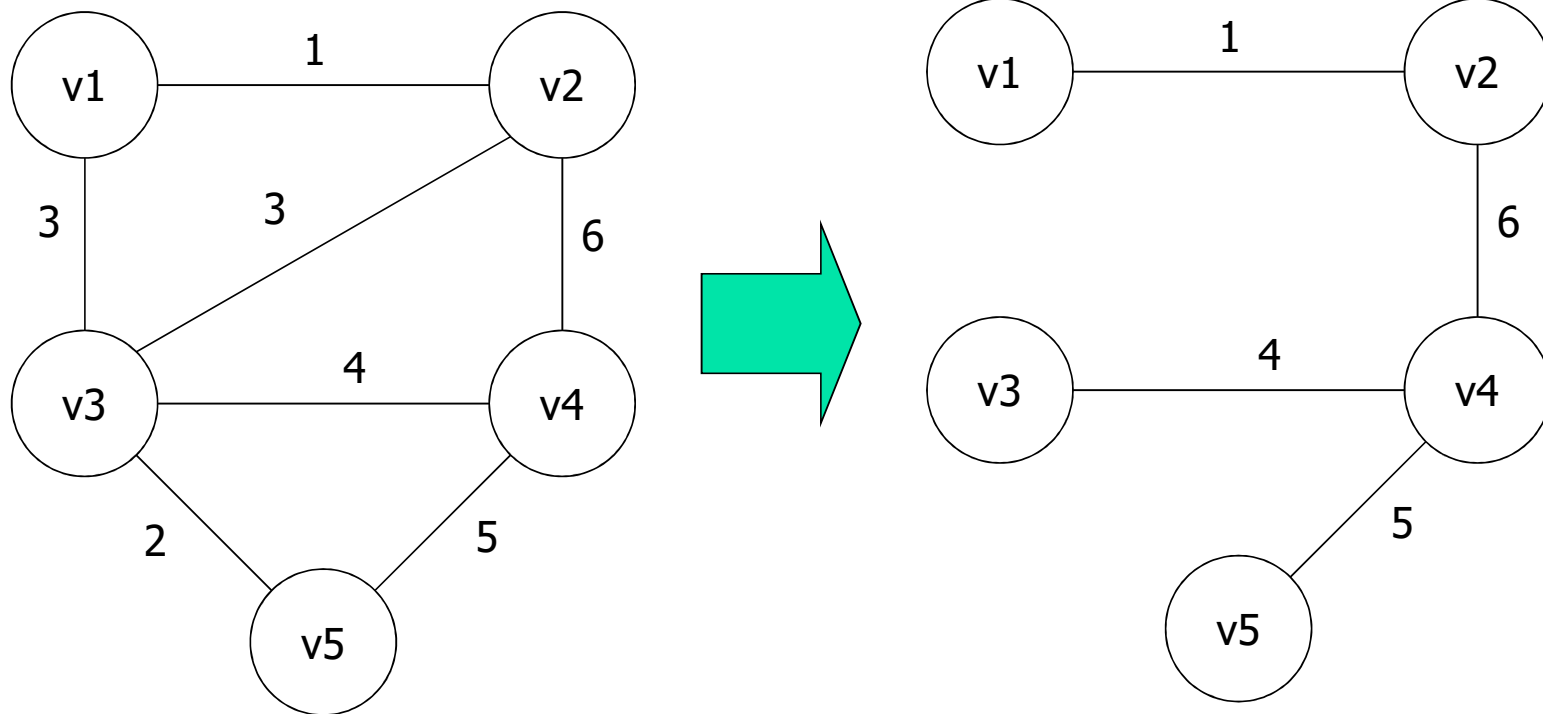
- 120원 짜리 동전을 새로 발행했다고 하자.
- 이 알고리즘을 적용하여 거스름돈을 주면, 항상 동전의 개수가 최소가 된다는 보장이 없다.
- 보기: 거스름돈 액수 = 160원
 - 탐욕 알고리즘의 결과:
 - $120\text{원} \times 1\text{개} = 120\text{원}, 10\text{원} \times 4\text{개} = 40\text{원}$
 - 동전의 개수 = 5개 \Rightarrow 최적(optimal)이 아님!
 - 최적의 해:
 - $100\text{원} \times 1\text{개}, 50\text{원} \times 1\text{개}, 10\text{원} \times 1\text{개}$ 가 되어 동전의 개수는 3개가 된다.



2. 최소비용 신장 트리 (Minimum cost spanning tree)

- 신장 트리의 정의
 - 연결된, 비방향성 그래프 G 에서 순환경로를 제거하면서 연결된 부분 그래프가 되도록 에지를 제거하면 신장 트리(spanning tree)가 된다.
 - 따라서 신장 트리는 G 안에 있는 모든 정점을 다 포함하면서 트리가 되는 연결된 부분 그래프이다.

신장 트리의 예

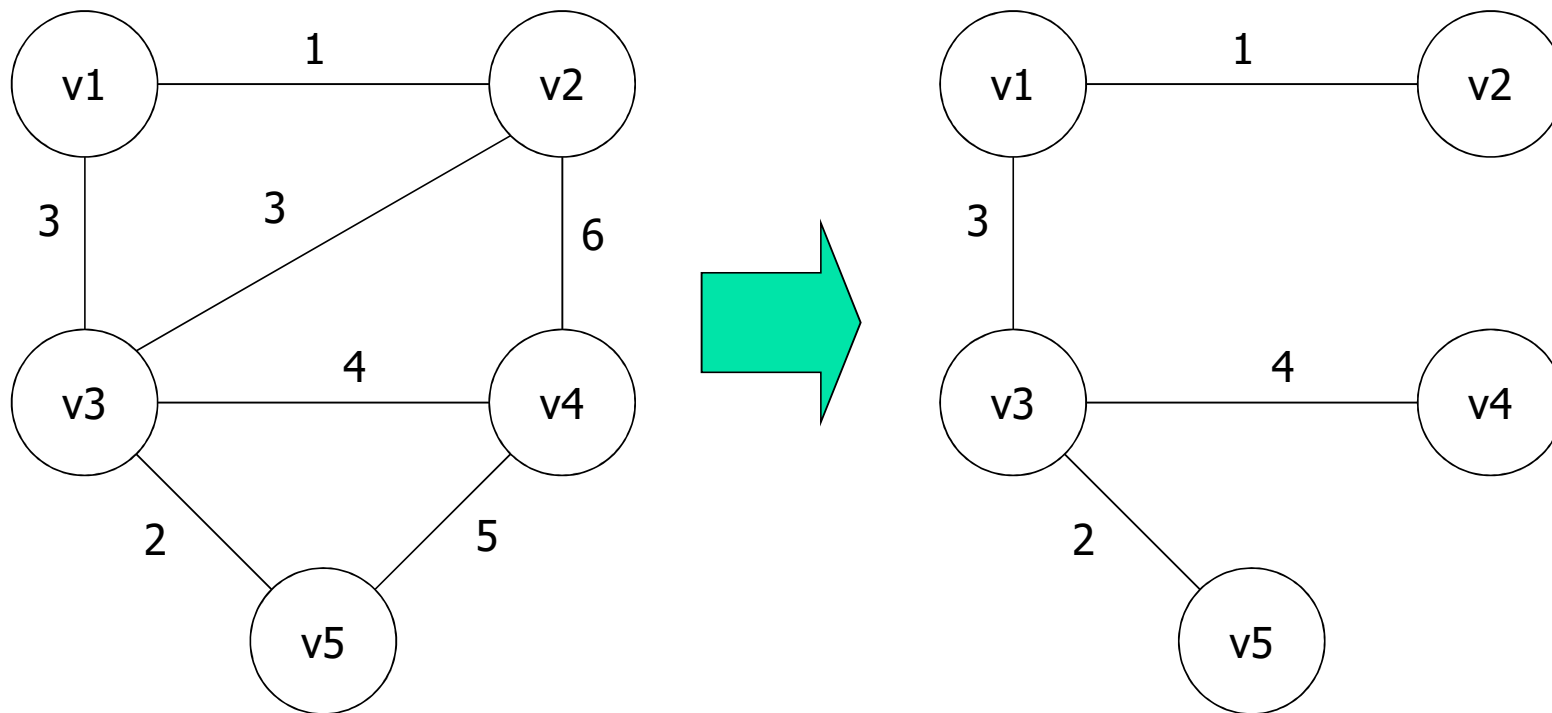




정의: 최소비용 신장 트리

- 신장트리가 되는 G 의 부분그래프 중에서 가중치가 최소가 되는 부분 그래프를 **최소비용 신장트리(minimum cost spanning tree)** 라고 한다.
- 최소의 가중치를 가진 부분그래프는 반드시 트리임.
 - 만약 트리가 아니라면, 분명히 순환경로(**cycle**)가 있을 것이고, 그렇게 되면 순환경로 상의 한 에지를 제거하면 더 작은 비용의 신장트리가 되기 때문이다.
- 관찰: 모든 신장트리가 최소비용 신장트리는 아니다.

최소비용 신장 트리의 예





최소비용 신장트리의 응용 분야

- 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

탐욕적인 접근 방법

- 문제: 비방향성 그래프 $G = (V, E)$ 가 주어졌을 때, $F \subseteq E$ 를 만족하면서, (V, F) 가 G 의 최소비용 신장트리(MST)가 되는 F 를 찾는 문제.
- 알고리즘:
 - $F := 0$;
 - 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복
 - 선택 절차: 적절한 최적해 선택절차에 따라서 하나의 에지를 선정
 - 적정성 점검: 선택한 에지를 F 에 추가시켜도 순환경로가 생기지 않으면, F 에 추가시킨다.
 - 해답 점검: $T = (V, F)$ 가 신장트리 이면, T 가 최소비용 신장트리 이다.

2.1 Prim의 알고리즘

- 개념

- $F := 0;$
- $Y := \{v1\};$
- 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복
 - 선정 절차/적정성 점검: $V - Y$ 에 속한 정점 중에서, Y 에 가장 가까운 정점 하나를 선정한다.
 - 선정한 정점을 Y 에 추가한다.
 - Y 로 이어지는 에지를 F 에 추가한다.
 - 해답 점검: $Y = V$ 가 되면, $T = (V, F)$ 가 최소비용 신장트리 이다.



자료 구조

- 인접행렬을 이용한 그래프의 구현, $W[i][j] =$
 - 에지의 가중치 (v_i 와 v_j 를 연결하는 에지가 존재할 경우)
 - ∞ (v_i 와 v_j 를 연결하는 에지가 없을 경우)
 - 0 ($i = j$ 인 경우)
- 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 정점
 - $\text{distance}[i]$
 - v_i 와 $\text{nearest}[i]$ 를 잇는 에지의 가중치

```

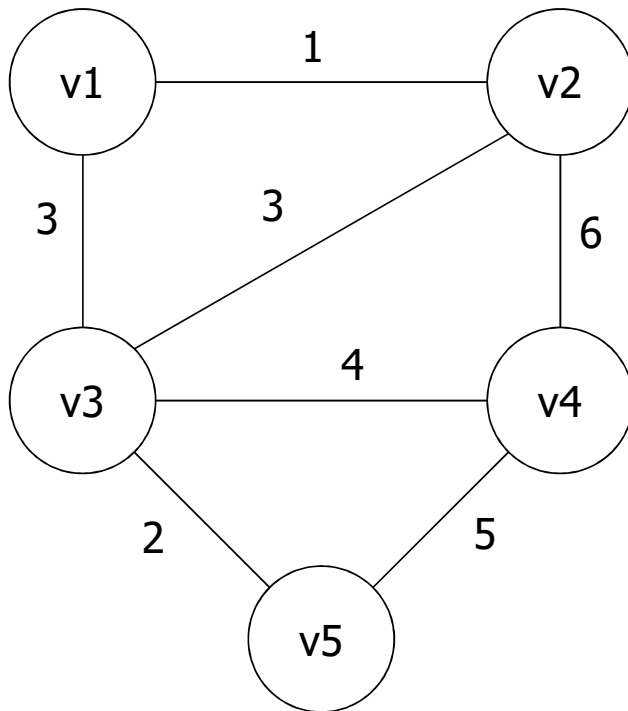
set_of_edges prim(int n, number W[][])
{
    index i, candidate, nearest[2..n];
    number min, distance[2..n];
    set_of_edges F;
    edge e;

    F =  $\emptyset$ ;
    for (i=2; i <= n; i++) {
        nearest[i] = 1;
        distance[i] = W[1][i];
    }
    repeat (n-1 times) {
        min =  $\infty$ ;
        for (i=2; i <= n; i++)
            if (0 <= distance[i] < min) {
                min = distance[i];
                candidate = i;
            }
        e = candidate와 nearest[candidate]를 잇는 에지;
        F = F  $\cup$  {e};
        distance[candidate] = -1;
        for (i=2; i <= n; i++)
            if (W[candidate][i] < distance[i]) {
                distance[i] = W[candidate][i];
                nearest[i] = candidate;
            }
    }
    return F;
}

```

// 초기화
 // v_i 에서 가장 가까운 정점을 v_1 으로 초기화
 // v_i 와 v_1 을 잇는 에지의 가중치로 초기화
 // n-1개의 정점을 Y에 추가한다
 // 각 정점에 대해서
 // distance[i]를 검사하여
 // 값이 가장 작은 정점 i를
 // candidate로 정한다.
 // 찾은 정점을 Y에 추가한다.
 // Y에 없는 각 정점에 대해서
 // distance[i]를 갱신한다.

알고리즘의 동작 과정



- $Y = \{v1\}$
 - $\text{distance}(2) = 1$
 - $\text{distance}(3) = 3$
 - $\text{distance}(4) = \infty$
 - $\text{distance}(5) = \infty$
- $Y = \{v1, v2\}, F = \{(1,2)\}$
 - $\text{distance}(3) = 3$
 - $\text{distance}(4) = 6, n(4) = 2$
 - $\text{distance}(5) = \infty$
- $Y = \{v1, v2, v3\}, F = \{(1,2), (1,3)\}$
 - $\text{distance}(4) = 4, n(4) = 3$
 - $\text{distance}(5) = 2, n(5) = 3$



알고리즘 분석

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

최적여부의 검증(Optimality Proof)

- Prim의 알고리즘의 결과가 최소비용 신장트리인지를 검증
- 정의 **4.1**: 비방향성 그래프 $G = (V, E)$ 가 주어지고, E 의 부분집합 F 에 MST가 되도록 에지를 추가할 수 있으면, F 는 **유망하다(promising)**라고 한다.
- 보조정리 **4.1**: $G = (V, E)$ 는 연결된 가중치 포함 비방향성 그래프. F 가 E 의 유망한 부분집합이라고 하고, Y 는 F 에 포함된 에지에 의해서 연결된 정점의 집합이라고 하자. 이때, Y 에 있는 어떤 정점과 $V - Y$ 에 있는 어떤 정점을 잇는 에지 중에서 가중치가 가장 작은 에지를 e 라고 하면, $F \cup \{e\}$ 는 유망하다.

보조정리 4.1의 증명

- (V, F') 가 MST라고 가정. $F \subseteq F'$
- e : Y 와 $V - Y$ 를 연결하는 가중치가 가장 작은 에지
- If $(e \in F')$, $F \cup \{e\} \subseteq F'$ 이므로 $F \cup \{e\}$ 는 유망
- Otherwise ($e \notin F'$)
 - $F' \cup \{e\}$ 는 하나의 cycle을 포함.
 - Y 의 정점과 $V - Y$ 의 정점을 잇는 다른 에지 e' 존재
 - $F' \cup \{e\} - \{e'\}$: MST
 - (왜냐하면, e 는 Y 와 $V - Y$ 를 잇는 최소비용 에지)
 - $e' \notin F$ 이므로, $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$
 - 그러므로 $F \cup \{e\}$ 는 유망

최적여부의 검증(계속)

- 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 에지의 집합인 F 가 유망하다고 가정한다
 - 귀납절차: 집합 $F \cup \{e\}$ 가 유망하다는 것을 보이면 된다. 여기서 e 는 다음 단계의 반복 수행 시 선정된 에지 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 다른 정점으로 잇는 에지 중에서 최소의 가중치를 가지고 있기 때문이다.

2.2 Kruskal의 알고리즘

- 개념

- $F := 0;$
- 서로소(disjoint)가 되는 V 의 부분집합들을 만드는데, 각 부분집합마다 하나의 정점만 가지도록 한다
- E 안에 있는 에지를 가중치의 오름차순으로 정렬
- 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복
 - 선정 절차: **최소의 가중치**를 갖는 다음 에지를 선정
 - 적정성 점검: **만약 선정된 에지가 두 개의 서로소인 정점을 잇는다면**, 먼저 그 부분집합을 하나의 집합으로 합하고, 그 다음에 그 에지를 F 에 추가한다.
 - 해답 점검: 만약 모든 부분집합이 하나의 집합으로 합하여 지면, 그 때 $T = (V, F)$ 가 최소비용 신장트리이다.



서로 소인 집합의 구현

- index i ;
- set_pointer p, q ;
- 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
- $\text{union}(p, q)$: 두 개의 집합을 가리키는 p 와 q 를 합병
- $\text{equal}(p, q)$: p 와 q 가 같은 집합을 가리키면 **true**를 넘겨줌

알고리즘의 구현

```
set_of_edges kruskal(int n, int m, set_of_edges E) {  
    index i, j;  
    set_pointer p, q;  
    edge e;  
    set_of_edges F =  $\emptyset$ ;  
  
    E에 속한 m개의 에지들을 가중치의 오름차순으로 정렬;  
    initial(n);  
    while (F에 속한 에지의 개수 < n-1) {  
        e = 아직 점검하지 않은 최소의 가중치를 가진 에지;  
        i, j = e를 이루는 양쪽 정점의 인덱스;  
        p = find(i);    q = find(j);  
        if (!equal(p, q)) { union(p, q); F = F  $\cup$  {e}; }  
    }  
    return F;  
}
```

알고리즘 분석

- 단위연산: 비교문
- 입력크기: 정점의 수 n 과 에지의 수 e
 1. 에지 들을 정렬하는데 걸리는 시간: $\Theta(e \log e)$
 2. 반복문 안에서 걸리는 시간: 루프를 e 번 수행한다. 서로소는 집합 자료구조를 사용하여 구현하고, **find, equal, union** 같은 동작을 호출하는 횟수가 상수이면, e 번 반복에 대한 시간복잡도는 $\Theta(e \log e)$ 이다.
 3. n 개의 서로소인 집합을 초기화하는데 걸리는 시간: $\Theta(n)$
- $e \geq n - 1$ 이기 때문에, 1과 2는 3을 지배. $W(e, n) = \Theta(e \log e)$
- 최악의 경우: 모든 정점이 다른 모든 정점과 연결이 될 수 있기 때문에 $e = n(n-1)/2$ 가 된다. 그러므로, 시간복잡도는
$$W(e, n) \in \Theta(n^2 \log n^2) = \Theta(2n^2 \log n) = \Theta(n^2 \log n)$$
- 최적여부의 검증(Optimality Proof)
 - Prim의 알고리즘의 경우와 비슷함. (교재 참조)



두 알고리즘의 비교

	$W(e, n)$	sparse graph	dense graph
Prim	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Kruskal	$\Theta(e \log e), \Theta(n^2 \log n)$	$\Theta(e \log e)$	$\Theta(n^2 \log n)$

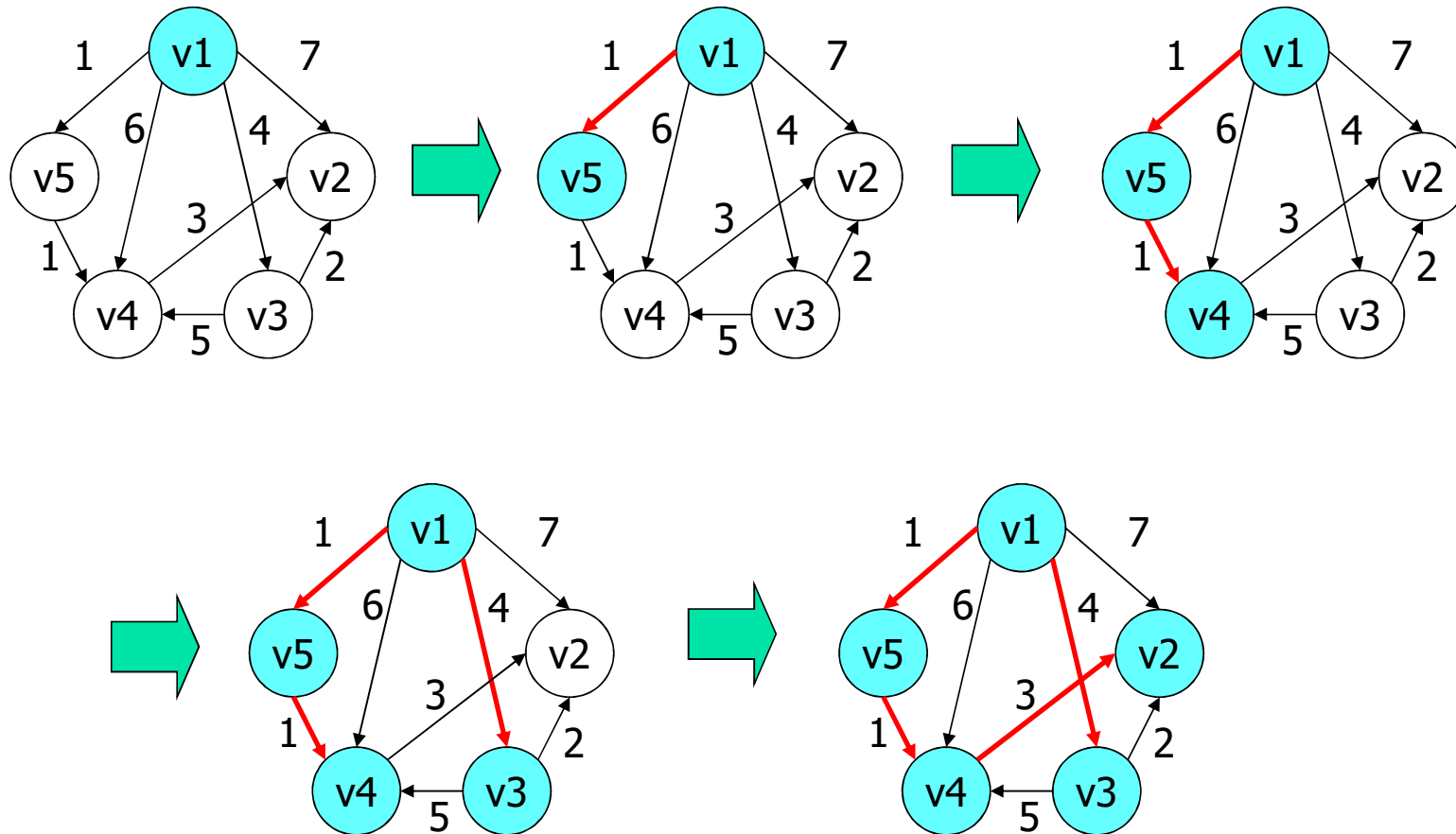
연결된 그래프에서의 e 는 $n - 1 \leq e \leq n(n-1)/2$ 의 범위를 갖는다.

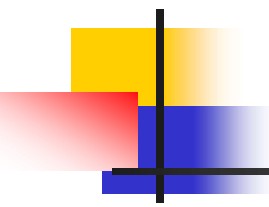
3. 단일 출발점 최소 경로(Dijkstra)

- 가중치가 있는 방향성 그래프에서 한 특정 정점에서 다른 모든 정점으로 가는 최단경로를 구하는 문제.
- 알고리즘:

- $F := 0;$
- $Y := \{v_1\};$
- 최종해답을 얻지 못하는 동안 다음 절차를 계속 반복
 - 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
 - 정점 v 를 Y 에 추가
 - v 에서 F 로 이어지는 최단경로 상의 에지를 F 에 추가
 - 해답 점검:
 - $Y = V$ 가 되면, $T = (V, E)$ 가 최단경로를 나타내는 그래프이다.

예제 그래프





Dijkstra 알고리즘의 분석

- Prim의 알고리즘의 경우와 비슷함
 - $T(n) \in \Theta(n^2)$
- 최적 여부의 검증도 Prim의 알고리즘과 비슷함.

탐욕적인 방법과 동적 프로그래밍의 비교

탐욕적인 접근 방법	동적 프로그래밍
■ 최적화 문제를 푸는데 적합	■ 최적화 문제를 푸는데 적합
■ 알고리즘이 존재할 경우 보통 더 효율적	■ 때로는 불필요하게 복잡
■ 알고리즘이 최적인지를 증명	■ 최적화 원칙이 적용되는지를 점검
■ 단일 출발점 최단 경로 문제: $\Theta(n^2)$	■ 단일 출발점 최단 경로 문제: $\Theta(n^3)$
■ 배낭 빈틈없이 채우기 문제는 풀지만, 0-1 배낭 채우기 문제는 풀지 못함	■ 0-1 배낭 채우기 문제는 푼다

4. 0-1 배낭 채우기 문제(0-1 Knapsack Problem)

- 문제: $S = \{\text{item}_1, \text{item}_2, \dots, \text{item}_n\}$
 - $w_i = \text{item}_i$ 의 무게
 - $p_i = \text{item}_i$ 의 가치
 - $W =$ 배낭에 넣을 수 있는 총 무게라고 할 때,
 - $\sum_{\text{item } i \in A} w_i \leq W$ 를 만족하면서 $\sum_{\text{item } i \in A} p_i$ 가 최대가 되도록 $A \subseteq S$ 인 A 를 구하는 문제.
- 무작정 알고리즘
 - n 개의 물건에 대해서 모든 부분집합을 다 고려한다.
 - 크기가 n 인 집합의 부분집합의 수는 2^n 개 이다.



0-1 배낭 채우기 문제: 탐욕적 알고리즘(1)

- 가장 비싼 물건부터 우선적으로 채운다.
- 애석하게도 이 알고리즘은 최적이지 않다!
- 왜 아닌지 보기:
 - 문제 정의
 - $W = 30\text{kg}$
 - item1: 무게 25kg, 값 10만원
 - item2: 무게 10kg, 값 9만원
 - item3: 무게 10kg, 값 9만원
 - 탐욕적인 방법: item1 \Rightarrow 25kg \Rightarrow 10만원
 - 최적의 해: item2 + item3 \Rightarrow 20kg \Rightarrow 18만원

0-1 배낭 채우기 문제: 탐욕적 알고리즘(2)

- 무게 당 가치가 가장 높은 물건부터 우선적으로 채운다.
- 그래도 최적이지 아니다!
- 왜 아닌지 보기:
 - 문제 정의
 - $W = 30\text{kg}$
 - item1: 무게 5kg, 값 50만원, 가치 10만원/kg
 - item2: 무게 10kg, 값 60만원, 가치 6만원/kg
 - item3: 무게 20kg, 값 140만원, 가치 7만원/kg
 - 탐욕적인 방법: item1 + item3 $\Rightarrow 25\text{kg} \Rightarrow 190\text{만원}$
 - 최적의 해: item2 + item3 $\Rightarrow 30\text{kg} \Rightarrow 200\text{만원}$



배낭 빈틈없이 채우기 문제

- 물건의 일부분을 잘라서 담을 수 있다.
- 탐욕적인 접근방법으로 최적의 해를 구하는 알고리즘을 만들 수 있다.
- $\text{item1} + \text{item3} + \text{item2} / 2 \Rightarrow 30\text{kg} \Rightarrow 220\text{만원}$
 - 최적의 해!



Knapsack problem의 종류

- 0-1 knapsack problem
 - 각 item당 배낭에 들어갈 수 있는 수는 기껏해야 1
- Bounded knapsack problem
 - item_i 가 배낭에 들어갈 수 있는 수 $\in \{0, 1, \dots, c_i\}$
- Unbounded knapsack problem
 - 각 item당 배낭에 들어갈 수 있는 수가 무제한

0-1 배낭 채우기 문제: 동적 프로그래밍(1)

- $i > 0$ 이고 $w > 0$ 일 때, 전체 무게가 w 를 넘지 않도록 i 번째까지의 항목 중에서 얻어진 최고의 이익 $P[i][w] =$
 - $\text{maximum}(P[i-1][w], p_i + P[i-1][w-w_i])$ ($w_i \leq w$)
 - $P[i-1][w]$ ($w_i > w$)
- $P[i-1][w]$: i 번째 항목을 포함시키지 않는 경우의 최고이익
- $p_i + P[i-1][w-w_i]$: i 번째 항목을 포함시키는 경우의 최고 이익
- 위의 재귀 관계식은 최적화 원칙을 만족함.

0-1 배낭 채우기 문제: 동적 프로그래밍(2)

- 최종 해 $P[n][W]$ 를 구하는 방법
 - 2차원 배열 $\text{int } P[0..n][0..W]$ 정의
 - 각 항을 차례대로 계산.
 - 단, $P[0][w] = 0, P[i][0] = 0$
- 시간복잡성
 - 계산해야 할 항목의 수 = $nW \in \Theta(nW)$
- 주의
 - 여기서 n 과 W 와는 아무런 상관관계가 없다.
 - $W = n!$ 이라고 한다면 수행시간은 $\Theta(n \times n!)$ 이 된다.
 - 무작정 알고리즘보다 뭐가 좋지?

0-1 배낭 채우기 문제: 분할 정복법

■ 분할 정복법

- $P[n][W]$ 를 계산하기 위해서 $(n-1)$ 번째 항을 모두 계산할 필요 없음!
- $P[n-1][W]$ 와 $P[n-1][W-w_n]$ 두 항만 계산하면 된다.
- 이런 식으로 $n = 1$ 이나 $w \leq 0$ 일 때 까지 계속

■ 예

- $P[3][30] = \max(P[2][30], 140 + P[2][10]) = 200$
- $P[2][30] = \max(P[1][30], 60 + P[1][20]) = 110$
- $P[2][10] = \max(P[1][10], 60 + P[1][0]) = 60$
- $P[1][30] = \max(P[0][30], 50 + P[0][25]) = 50$
- $P[1][20] = P[1][10] = 50$
- $P[1][0] = 0$

item1: 무게 5kg, 값 50만원
item2: 무게 10kg, 값 60만원
item3: 무게 20kg, 값 140만원

7개의 항만 계산 ($nW = 90$)

0-1 배낭 채우기 문제:복잡성

- 분할 정복법의 분석
 - $(n - i)$ 번째 항에서 기껏해야 2^i 항을 계산
 - 계산하는 총 항의 수 = $\Theta(2^n)$
$$1 + 2 + 2^2 + \dots + 2^{n-1} = 2^n - 1$$
 - 최악의 경우의 수행시간은 $O(\text{minimum}(2^n, nW))$ 이다.
 - 분할 정복법으로 접근할 경우: $O(2^n)$
 - 동적 프로그래밍의 경우: nW 항이 추가! ($W \approx n$ 일 때 사용 가능)
- 아직 아무도 이 문제의 최악의 경우 실행시간이 지수(exponential)보다 나은 알고리즘을 발견하지 못했고, 그러한 알고리즘이 없다라고 증명한 사람도 없다.

NP문제