챕터 7. Greedy Aproach (탐욕 알고리즘) - (1) 탐욕법과 최소비용 신장트리 (프림 알고리즘)

탐욕 알고리즘 : The Greedy Approach (스크루지의 금모으기 - 고전 소설의 예화)

: 답을 하나씩 고르는데, 미리 정한 기준에 따라서 매번 '가장 좋아 보이는' 답을 선택한다.

명확한 정의를 하자면,

1. 최종 해답을 찾기 위해서 각 단계마다 하나의 답을 고름.

2. 각 단계에서 답을 고를 때 가장 좋아 보이는 답을 선택.

3. 최적화 문제에서,

    - 선택할 당시에는 최적의 답을 고르지만 (locally optimal),

    - 최종 해답이 반드시 최적임을 보장하지 않음(globally optimal not guaranteed)

동전의 거스름돈 문제

- 거스름돈이 870원인 경우,

- 동전의 개수가 최소가 되도록 동전을 선택하는 방법은?

거스름돈 문제 : 탐욕법으로 풀기

```
while (동전이 남아있고 문제가 미해결):
  가장 가치가 높은 동전을 선택한다.
  if (동전을 더하여 거스름돈의 총액이 거슬러주어야 할 액수를 초과):
    동전을 도로 집어넣는다.
  else:
    거스름돈에 동전을 포함시킨다.
  if (거스름돈의 총액이 거슬러주어야 할 액수와 같다)
    문제 해결.
```

거스름돈 알고리즘은 항상 최적해를 찾는가?

- 동전의 구성이 [500, 100, 50, 10, 5, 1]일 경우: Yes!

- 동전의 구성이 [500, 100, 80, 50, 10, 5, 1]일 경우: No!

예) 거스름돈 360원의 최적해는?

    - 탐욕 알고리즘의 해: [100, 100, 100, 50, 10]

    - 최적해: [100, 100, 80, 80]

-> 탐욕 알고리즘의 해가 항상 최적해를 갖지 않는다는걸 알 수 있다.

탐욕 알고리즘의 이모저모

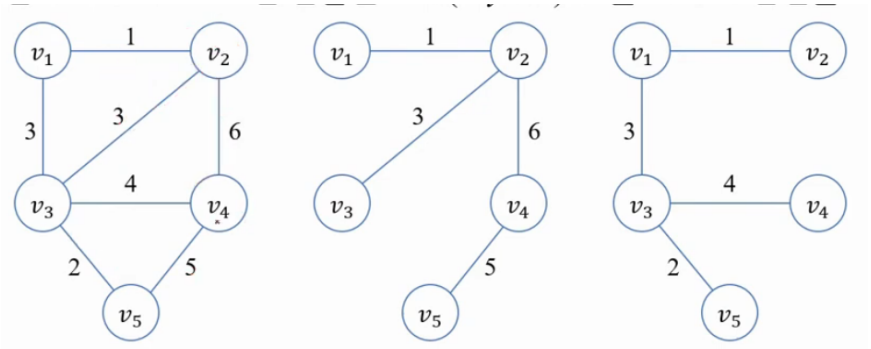
- 탐욕 알고리즘의 설계 전략: 공집합에서 시작
    - 선택 과정: 집합에 추가할 다음 최적의 원소를 고른다.
    - 적절성 검사: 새로운 집합이 해답으로 적절한지 검사한다.
    - 해답 점검 : 새로운 집합이 문제의 해답인지 판단한다.
- 탐욕 알고리즘의 장단점
    - 장점: 상대적으로 설계하기가 매우 쉽다.
    - 단점: 최적화 문제에서 반드시 정확성을 증명해야 함.

최소비용 신장트리 문제

- 문제: 주어진 그래프에서 최소비용 신장트리를 구하시오.
- 엄밀한 문제 정의
    - 주어진 그래프; G = (V, E) : connected, weighted, and undirected
        - 그래프 G는 모든 정점이 연결된 가중치가 있는 무방향 그래프
    - 신장트리(spanning tree): G의 부분 그래프 T = (V, F), F는 E에 속한다.
        - 그래프 G의 모든 정점을 연결하는 트리: 간선의 개수는 n - 1
    - 최소비용 신장트리(MST: Minimum cost Spanning Tree)
        - 모든 신장트리 T 중에서 가중치의 합이 최소가 되는 신장트리

최소비용 신장트리 문제의 이해

- 단순무식하게 풀기(Brute-Force)
    - 모든 신장트리를 찾아서 가중치의 합이 가장 작은 것을 선택
- 신장트리를 찾는 법
    - 간선의 개수가 n - 1 인 연결된 트리(acyclic)가 될 때까지 간선을 하나씩 제거



최소비용 신장트리: 욕심쟁이 방법(Greedy Approach)

- 1단계(초기화): 해답의 집합을 공집합으로 둔다.

    - 간선 집합 E의 부분 집합 F를 공집합으로 둔다.

- 2단계(선택): 최적의 원소 하나를 해답의 집합에 포함시킨다.

    - E에서 최적의 간선 하나를 추출해서 F에 포함시킨다.

    - 최적을 선택하는 방법?(간선을 선택하는 방법에 따라서) 프림 vs 크루스칼 으로 나누어진다.

- 3단계(검사): 해답의 집합이 최종이면 종료, 아니면 2단계를 반복한다.

    - 간선의 부분 집합 F의 원소 개수가 n - 1이면 최종 해답

최조비용 신장트리: 프림 알고리즘(Prim's Algorithm)

- 1단계(초기화) : 해답의 집합을 공집합으로 둔다.

    - F = Φ, Y = $\{v_1\}$: Y는 정점의 집합 V의 부분 집합

- 2단계(선택): 최적의 원소 하나를 해답의 집합에 포함시킨다.

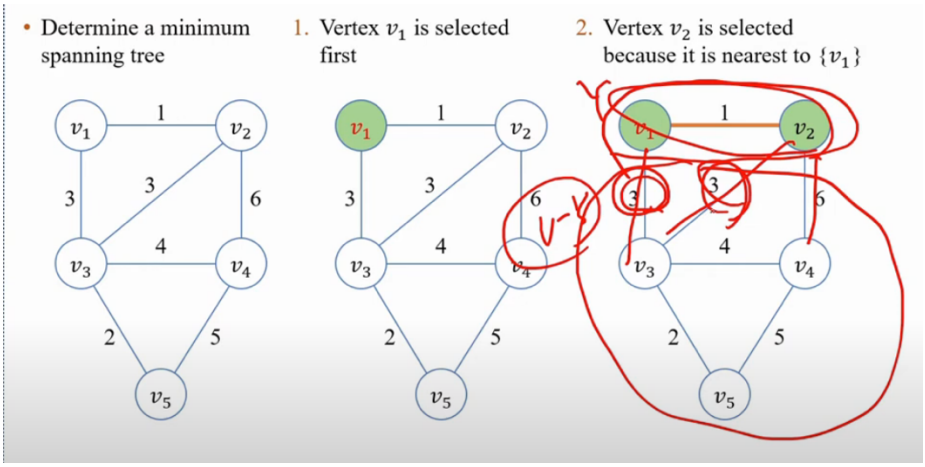
    - V - Y 집합에서 Y 집합에서 가장 가까운 정점 vnear를 선택

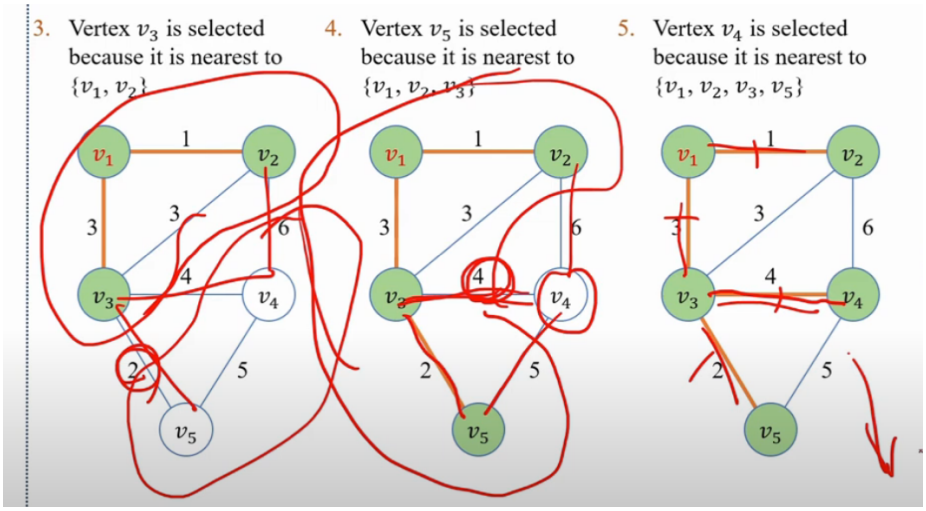
    - Y 집합에 vnear를 추가, F 집합에 (nearest(vear), vear)를 추가

- 3단계(검사) : 해답의 집합이 최종이면 종료, 아니면 2단계를 반복한다.

    - Y = V : Y 집합이 V 집합의 모든 원소를 포함하면 종료

1. Y와 V - Y 집합에서 가장 가까운 정점 vnear중, 최적의 vnear를 선택한다.





MST의 최소 cost는 10인 최소비용신장트리를 만든 것.

(증명해야 하지만 수학적으로 증명하는건 어렵기 때문에 넘어감).

그럼 prim 알고리즘을 어떻게 구현할까?

- W[i][j]: 인접행렬 (간선의 가중치)

- nearest[i]: Y 집합에서 v_i에 가장 가까운 정점의 인덱스

- distance[i]: v_i와 nearest[i]의 정점을 연결하는 간선의 가중치

W	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

	i	2	3	4	5	*e*
init:	nearest[i]	1	1	1	1	
	distance[i]	1	3	∞	∞	
step 1:	nearest[i]	1	1	2	1	(2, 1, 1)
	distance[i]	-1	3	6	∞	
step 2:	nearest[i]	1	1	3	3	(3, 1, 3)
	distance[i]	-1	-1	4	2	
step 3:	nearest[i]	1	1	3	3	(5, 3, 2)
	distance[i]	-1	-1	4	-1	
step 4:	nearest[i]	1	1	3	3	(4, 3, 4)
	distance[i]	-1	-1	-1	-1	

1. vertex의 길이 n, 2차원 행렬 W, edge의 집합 F를 받는다.

2. F의 공집합을 F.clear()로 표현한다.

3. for문을 보면 i의 인덱스는 2부터 시작하는 걸 볼 수 있다. why? v_1은 스타트 vertex이기 때문에, 이미 처음에 Y에 포함시켰으니까 그런 것,

4. 또한 for문의 내용을 보면 nearest[i] = 1;로 스타트 vertex를 초기화해주는 것을 볼 수 있다.

ALGORITHM 4.1: Prim's Algorithm

```
void prim(int n, matrix_t& W, set_of_edges& F)
{
    int vnear, min;
    vector<int> nearest(n + 1), distance(n + 1);

    F.clear(); // F = ∅;
    for (int i = 2; i <= n; i++) {
        nearest[i] = 1;
        distance[i] = W[1][i];
    }
```

ALGORITHM 4.1: Prim's Algorithm (continued)

```
    repeat (n - 1 times) {
        min = ∞;
        for (int i = 2; i <= n; i++)
            if (0 <= distance[i] && 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 (int i = 2; i <= n; i++)
            if (distance[i] > W[i][vnear]) {
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
    }
}
```

```
#define INF 0xffff

typedef vector<vector<int>> matrix_t;
typedef vector<pair<int, int>> set_of_edges;
typedef pair<int, int> edge_t;


// e = edge connecting vertices indexed by vnear and nearest[vnear];
// add e to F;
F.push_back(make_pair(vnear, nearest[vnear]));


set_of_edges F;
prim(n, W, F);
for (edge_t e: F) {
    u = e.first; v = e.second;
    cout << u << " " << v << " " << W[u][v] << endl;
}
```

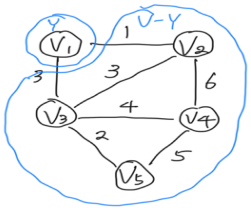
```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <stdio.h>
#define INF 0xffff
using namespace std;

typedef vector<vector<int> > matrix_t;
typedef vector<pair<int, int> > set_of_edges;
typedef pair<int, int> edge_t;


void prim(int n, matrix_t& W, set_of_edges& F);
int main()
{
    set_of_edges F;
    int row, col;
    int n,m;
    cin >> n >> m;
    matrix_t W(n+1,vector<int>(n+1,INF));

    while (m--)
    {
        cin >> row >> col;
        cin >> W[row][col];
        W[col][row] = W[row][col];
    }
    for (int i = 1; i <= n; i++)
        W[i][i] = 0;
    prim(n, W, F);

    for (edge_t e:F) {
        row = e.first; col = e.second;
        cout << row << " " << col << " " << W[row][col] << endl;
    }

    return 0;
}
void prim(int n, matrix_t& W, set_of_edges& F) {
    int vnear, min;
    vector<int>nearest(n + 1), distance(n + 1);
    F.clear();
    for(int i=2;i<=n;i++)
    {
        nearest[i]=1;
        distance[i]=W[1][i];
    }
    for (int j = 1; j < n; j++)
    {
        for (int i = 2; i <= n; i++)
        {
            if (i != n)
                cout << nearest[i] << " ";
            else
                cout << nearest[i] << endl;
        }
        min = INF;
        for (int i = 2; i <= n; i++)
        {
            if (0 <= distance[i] && distance[i] < min)
            {
                min = distance[i];
                vnear = i;
            }
        }

        F.push_back(make_pair(vnear, nearest[vnear]));

        distance[vnear] = -1;
        for (int i = 2; i <= n; i++)
        {
            if (distance[i] > W[i][vnear])
            {
                distance[i] = W[i][vnear];
                nearest[i] = vnear;
            }
        }

    }
    for (int i = 2; i <= n; i++)
    {
        if (i != n)
            cout << nearest[i] << " ";
        else
            cout << nearest[i] << endl;
    }


}
```

matrix W, n=5 (→ n: vertex 수), k=7 (→ k: 간선의 수)

인접행렬 W[i][j] (간선의 가중치)

W	1	2	3	4	5
1	INF	1	3	INF	INF
2	1	INF	3	6	INF
3	3	3	INF	4	2
4	INF	6	4	INF	5
5	INF	INF	2	5	INF

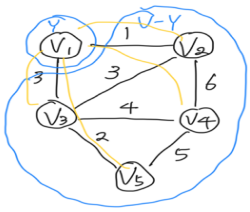
(우방향 그래프이기 때문.)



→초기 그래프 상태.

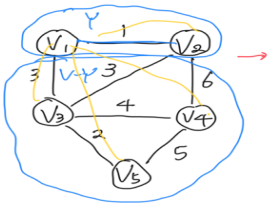
nearest[i] : Y집합에서 Vi에 가장 가까운 정점의 인덱스
distance[i] : Vi와 nearest[i]의 정점을 연결하는 간선의 가중치.

	1	2	3	4	5	e
init:	nearest[i]	1	1	1	1	→ i에다 Y집합과 연결되거나, 가까운 정점 number.
	distance[i]	1	3	∞	∞	→ Y집합에서 연결되는 i의 가중치 값.
STEP 1:	nearest[i]	1	1	2	1	2 1 1
	distance[i]	-1	3	6	∞	
STEP2:	nearest[i]	1	1	3	3	3 1 3
	distance[i]	-1	-1	4	2	
STEP3:	nearest[i]	1	1	3	3	5 3 2
	distance[i]	-1	-1	4	-1	
STEP4:	nearest[i]	1	1	3	3	4 3 4
	distance[i]	-1	-1	-1	-1	



→초기 그래프 상태.
↳위 표에서 init 단계

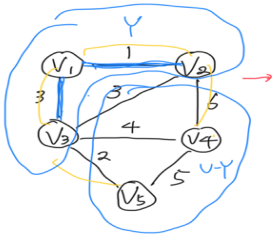
i	2	3	4	5
nearest[i]	1	1	1	1
distance[i]	1	3	∞	∞



→ STEP 1

i	2	3	4	5
nearest[i]	1	1	2	1
distance[i]	-1	3	6	∞

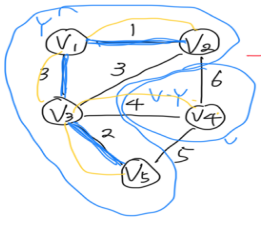
2 1



→ STEP 2

i	2	3	4	5
nearest[i]	1	1	3	3
distance[i]	-1	-1	4	2

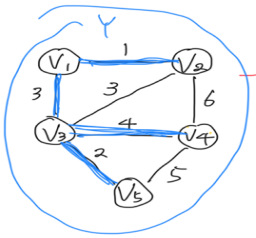
3 1



→ STEP 3

i	2	3	4	5
nearest[i]	1	1	3	3
distance[i]	-1	-1	4	-1

5 3



→ STEP 4 Y = V

i	2	3	4	5
nearest[i]	1	1	3	3
distance[i]	-1	-1	-1	-1

4 3