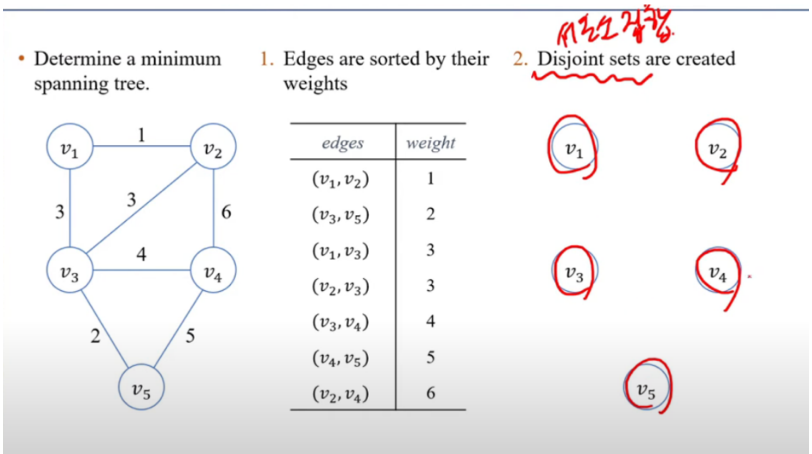
챕터 7. Greedy Aproach (탐욕 알고리즘) - (2) 서로소 집합과 크루스칼 알고리즘

최소비용 신장트리: 크루스칼 알고리즘(Kruskal's Algorithm)

- 1단계(초기화): 해답의 집합을 공집합으로 둔다. F = Φ
    - V의 **서로소 집합(disjoint set)**을 생성한다.
    - E를 비내림차순으로 **정렬**한다.

- 2단계(선택): 최적의 원소 하나를 해답의 집합에 포함시킨다.
    - 정렬된 E 집합에서 간선 e = (i, j)를 선택
    - 두 정점 i, j가 속한 집합 p, q를 찾아서 (**Find**),
      p, q가 같으면 e를 버리고, 다르면 F에 e를 포함한 후, p,q 를 합친다 (Union).

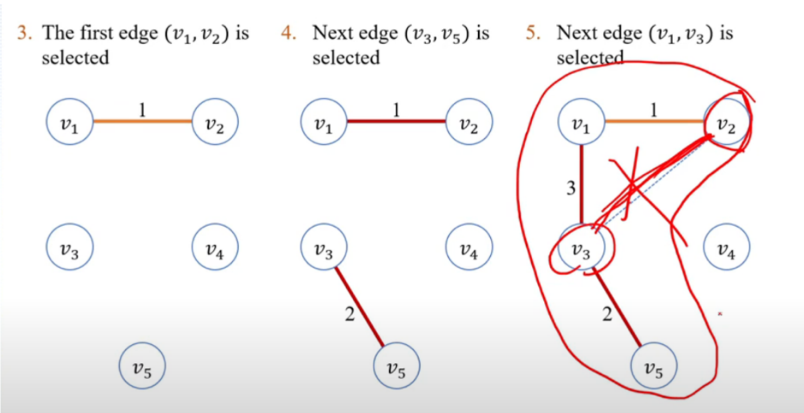
    => Cycle 탐지 - (Cycle을 형성해버리면 Graph가 되기 때문에, 우리가 목적한 MST(tree)랑은 달라진다.)



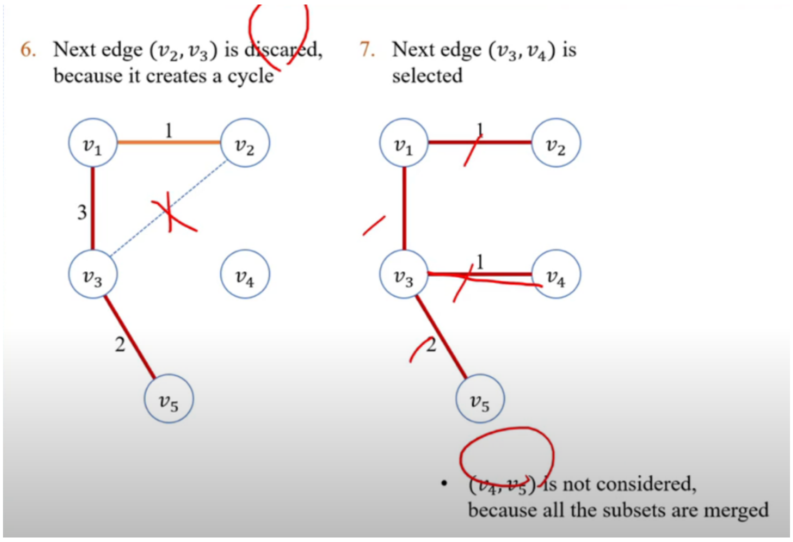
edges	weight
(v_1, v_2)	1
(v_3, v_5)	2
(v_1, v_3)	3
(v_2, v_3)	3
(v_3, v_4)	4
(v_4, v_5)	5
(v_2, v_4)	6

1. Edge의 weight(가중치)가 작은 것부터 차례대로 정렬한다.

2. 서로소 부분집합을 create(구축)한다. ==> *참고: 자료구조로 치면 여러개의 tree가 모여있는 forest구조이다.



3. Edge (v1, v2)를 선택한다. 4. Edge (v3, v5)를 선택한다. 5. Edge (v1, v3)를 선택한다.



6. Edge (v2, v3)를 선택한다.   ==> Cycle을 형성하기에 추가를 하지 않는다.

7. Edge (v3, v4)를 선택한다.   ==> 종료조건, n이 4개가 되었다.

사이클 탐지를 어떻게 하지?

- 서로소 집합 (Disjoint Set)

    - 교집합이 공집합인 두 집합 A, B는 서로소 집합. A ∩ B = Φ

- Union-find 알고리즘

    - 서로소 집합 자료구조를 이용해서

    - 두 개의 원소가 같은 집합에 속하는 지를 판단할 수 있는 알고리즘.

- 전체집합 U = { A, B, C, D, E }

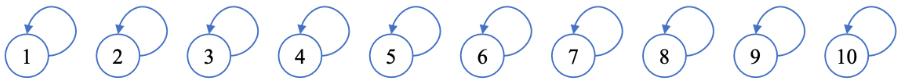
```
for i in U:
    initial(i);    {A}    {B}    {C}    {D}    {E}    (disjoint sets)
                           ↑      ↑
   p = find(B);            p      q
   q = find(C);

   merge(p, q);    {A}   {B,C}          {D}    {E}
                           ↑                    ↑
   p = find(C);            p                    q    equal(C, E);
   q = find(E);                                        returns false;

   merge(p, q);    {A}   {B,C,E}        {D}
                          ↑   ↑
   p = find(C);           p   q
   q = find(E);                                      equal(C, E);
                                                       returns true;
```

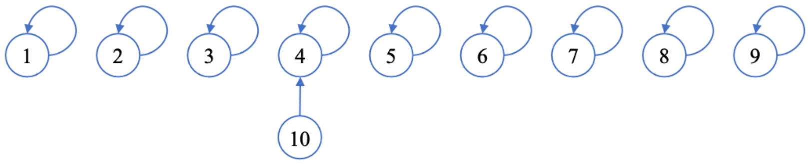
-> find할 때 같은 집합이 되면 true를 호출한다. (종료한다)

```
initial(10);
```



1	2	3	4	5	6	7	8	9	10
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

```
merge(find(4), find(10);
```

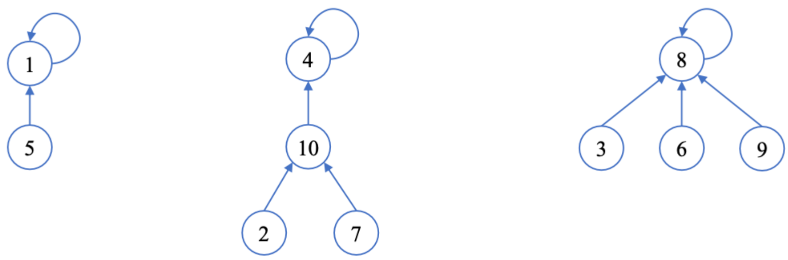


1	2	3	4	5	6	7	8	9	4
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

1. 자기 자신을 가르키는 tree들이 나열되어 있으니까, forest라고도 할 수 있다.

2. merge(find(4), find(10))호출 시 각각 자기들의 집합 4와 10이 묶여지는걸 볼 수 있다.

```
After several union and find:
```



1	10	8	4	1	8	10	8	8	4
$U[1]$	$U[2]$	$U[3]$	$U[4]$	$U[5]$	$U[6]$	$U[7]$	$U[8]$	$U[9]$	$U[10]$

1. merge하면은 tree의 루트노드를 다른 루트노드에 연결하면, 하나의 같은 집합이 될 수 있다.

2. find하면은 tree 노드의 루트노드까지의 이어진 노드들을 통해 루트노드를 반환한다.

==> 간단한데 유용한 자료구조

```
typedef struct edge {
    int u, v, w;
} edge_t;

struct edge_compare {
    bool operator()(edge_t e1, edge_t e2) {
        if (e1.w > e2.w) return true;
        else return false;
    }
};

typedef vector<edge_t> set_of_edges;
typedef priority_queue<edge_t, vector<edge_t>, edge_compare> PriorityQueue;


// sort the m edges in E by weight in nondecreasing order;
for (edge_t e: E)
    PQ.push(e);
```

```
vector<int> dset;

void dset_init(int n) {
    dset.resize(n + 1);
    for (int i = 1; i <= n; i++)
        dset[i] = i;
}

int dset_find(int i) {
    while (dset[i] != i)
        i = dset[i];
    return i;
}

void dset_merge(int p, int q) {
    dset[p] = q;
}
```

Algorithm 4.2: Kruskal's Algorithm

```
void kruskal(int n, int m, set_of_edges& E, set_of_edges& F) {
    int p, q;
    edge_t e;
    PriorityQueue PQ;

    sort the m edges in E by weight in nondecreasing order;
    F.clear(); // F = ∅;
    dset_init(n);
    while (number of edges in F is less than n - 1) {
         e = PQ.top(); PQ.pop(); // edge with least weight not yet considered;
        p = dset_find(e.u);
        q = dset_find(e.v);
        if (p != q) {
            dset_merge(p, q);
            F.push_back(e); // add e to F
        }
    }
}
```

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <queue> // priority_queue를 위한 헤더
using namespace std;

typedef struct edge {
    int u, v, w;
} edge_t;

struct edge_compare {
    bool operator()(edge_t e1, edge_t e2) {
        return e1.w > e2.w; // 최소 힙
    }
};

typedef vector<edge_t> set_of_edges;
typedef priority_queue<edge_t, vector<edge_t>, edge_compare> PriorityQueue;

vector<int> dset;

void kruskal(int n, int m, set_of_edges& E, set_of_edges& F);
void dset_init(int n);
int dset_find(int i);
void dset_merge(int p, int q);

int main () {
    int n, k; // 정점 수 n, 간선 수 k
    cin >> n >> k;

    set_of_edges E;
    for (int i = 0; i < k; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        E.push_back({u, v, w});
    }

    set_of_edges F; // MST에 포함될 간선들

    kruskal(n, k, E, F);

    for (edge_t e : F) {
        cout << e.u << " " << e.v << " " << e.w << '\n';
    }

    return 0;
}

void kruskal(int n, int m, set_of_edges& E, set_of_edges& F) {
    int p, q;
    edge_t e;
    PriorityQueue PQ;

    // 모든 간선을 우선순위 큐에 삽입
    for (edge_t edge : E)
        PQ.push(edge);

    dset_init(n);
    int count = 0;

    while (!PQ.empty() && count < n - 1) {
        e = PQ.top(); PQ.pop();

        p = dset_find(e.u);
        q = dset_find(e.v);

        if (p != q) {
            dset_merge(p, q);
            F.push_back(e);
            count++;
        }
    }
}

void dset_init(int n) {
    dset.resize(n + 1); // 1-based 인덱싱
    for (int i = 1; i <= n; i++)
        dset[i] = i;
}

int dset_find(int i) {
    if (dset[i] == i)
        return i;
    return dset[i] = dset_find(dset[i]); // 경로 압축
}

void dset_merge(int p, int q) {
    int root_p = dset_find(p);
    int root_q = dset_find(q);
    if (root_p != root_q)
        dset[root_q] = root_p;
}
```

시간복잡도 분석

단위연산: 비교 명령문

입력크기: n (정점의 개수), m(간선의 개수)

고려사항(3가지)

1. 간선을 정렬하는데 걸리는 시간

-> Θ(m lg m)

2. while 루프에서 걸리는 시간, 서로소 집합을 조작하는데 걸리는 시간이 이 루프의 시간복잡도를 좌우한다.

(왜냐하면, 나머지는 모두 상수이기 때문이다.) 최악의 경우, while 루프를 빠져나가기 전에 간선을 모두 고려하는데, 이는 루프를 m번 반복한다는 뜻이 된다.

-> $W(m) \in \Theta(m \lg m)$

3. n개의 서로소 집합을 초기화하는데 걸리는 시간, 앞에서 서로소 집합 데이터 구조의 구현방법을 사용하면, 초기화하는 시간복잡도는 다음과 같다.

-> $T(n) \in \Theta(n)$

m >= n - 1 이므로, 서로소 집합을 정렬하고 조작하는 시간이 초기화시간을 지배한다. 따라서 다음과 같다.

-> $W(m ,n) \in \Theta(m \lg m)$

최악의 경우가 n의 값과는 상관이 없는 것처럼 보일지도 모른다. 그러나 최악의 경우 모든 마디는 다른 모든 ㅁ나디와 연결되기 때문에 다음과 같다.

-> $m = n(n-1) / 2 \in \Theta(n^2)$

그러므로 최악의 경우에 다음과 같이 쓸 수도 있다.

-> $w(m, n) \in \Theta(n^2 \lg n^2) = \Theta(n^2\ 2\lg n) = \Theta(n^2 \lg n)$

크루스칼 알고리즘은 프림 알고리즘과 비교할 때 최악의 경우에 대해서 위의 두 가지 표현을 모두 사용하는 게 좋다.