최소 스패닝 트리 - MST

MST - Minimum Spanning Tree

개요

사이클 없이 최소 비용으로 모든 그래프의 노드를 연결하는 알고리즘. 말만 들으면 이게 무슨 소리야? 싶은 이야기입니다.

> **사이클** : 그래프의 한 노드에서 출발하는데 다시 돌아오는 경로가 있는 경우

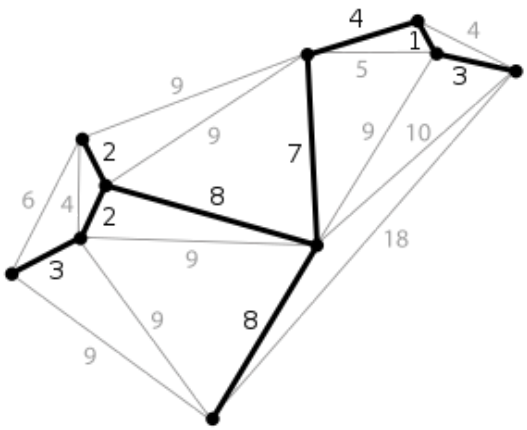
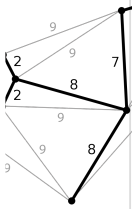


사진 출처

Minimum spanning tree - Wikipedia

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected,

𝕎 https://en.wikipedia.org/wiki/Minimum_spanning_tree



위 그림을 보면 빠르게 이해가 됩니다. 사이클 없이 노드들을 한번에 연결한 모습이죠. 이중 주목해야하는건 **<u>최소 비용</u>** 입니다. 알고리즘 이름에서 알 수 있듯 최소 비용이 메인이 되는 알고리즘입니다.

이 MST를 만들기 위해서 여러 알고리즘이 있겠지만 가장 대중적으로 알려진 알고리즘들 중 `크루스칼 알고리즘`과 `프림 알고리즘`을 알아보겠습니다.

크루스칼 알고리즘

크루스칼 알고리즘은 `Union-Find` 알고리즘을 우선 알고 있어야 합니다만, 이 과정만 넘기면 엄청 간단하게 알 수 있습니다.

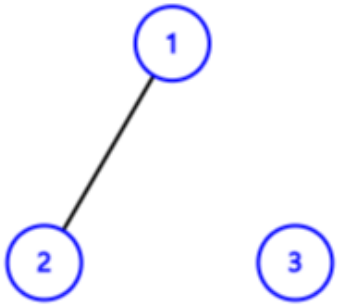
Union-Find : 합집합 찾기

그래프의 노드들이 있을때 **같은 집합에 있는가**를 알아보는 알고리즘으로 Disjoint-Set 알고리즘(또는 자료구조)를 사용하는 알고리즘입니다.

> 💡 아래 내용에서 **부모**는 **그룹**과 같은 의미입니다!

이 알고리즘은 총 2가지의 연산을 가진다고 생각하시면 됩니다. 예시를 보기 위해 1, 2, 3의 노드가 있다고 생각하고 배열을 하나 만들어 **자신의 부모를 배열에 저장**하고 있다고 하겠습니다.

노드	1	2	3
부모	1	2	3

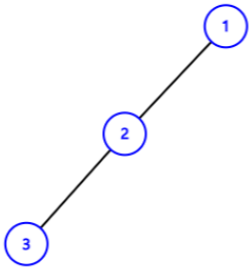


- Union (Merge)
  - 두 트리를 합치는 연산을 합니다. 두 트리에 있는 노드들이 가리키는 부모를 통일해 같은 집합으로 넣어주는 연산을 합니다.
  - 위 그림에서 1 과 2가 다른 부모를 가졌지만 이를 1 한쪽으로 몰아줍니다.
  - 결과 부모를 나타내는 배열은 아래와 같은 모습이 됩니다.

노드	1	2	3
부모	1	**1 (prev : 2)**	3

  - C++ 에서는 `union` 키워드가 있어서 주로 `merge` 로 이름짓는 편입니다.
- Find
  - 노드의 부모를 찾아주는 연산으로 부모 중에서 `루트 노드`를 찾아주는 연산을 합니다.
  - 과정 : 해당 노드의 부모 찾기 → 루트 노드가 나올때 까지 부모의 부모 찾기

- 만일 트리가 선형적으로 되어 있는 경우 이 연산의 시간복잡도는 최대 $O(N)$이 됩니다.



<코드로 보는 Union-Find>

```
#include <iostream>
using namespace std;
int parent[1000001];

int FindParent(int x){
    if(x == parent[x])
        return x;

    return FindParent(parent[x]);
}

void Merge(int x, int y){
    x = FindParent(x);
    y = FindParent(y);

    if(x == y)
        return;
    parent[x] = y;
}

int main(){
    for(int i = 0; i <= 1000000; i++)
        parent[i] = i;
}
```

<Find>

```
int FindParent(int x){
    if(x == parent[x])
        return x;

    return parent[x] = FindParent(parent[x]);
}
```

실제로 트리를 연결한다면 합쳐지는 트리가 기존 트리의 밑으로 들어가야 하지만 위에서 말한대로 그런 경우 시간복잡도가 너무 커지게 됩니다.

위와 같은 단점으로 인해서 Find의 return이 위와 같이 변경이 된 것입니다. 부모를 **찾는 과정에서 갱신도 함께** 함으로서 **처음 만나는 부모 = 루트 노드**가 되게끔 변경한 모습입니다.

> 위 함수의 시간복잡도는 $O(a(N))$이며 a는 '아커만 함수'라고 합니다.
> N이 2^{65536}일때 5 정도라고 합니다.

<Union - Merge>

```
void Merge(int x, int y){
    x = FindParent(x);
    y = FindParent(y);

    if(x == y)
        return;
    parent[x] = y;
}
```

부모를 찾아서 한쪽으로 연결하는 연산입니다.

트리의 깊이를 그대로 유지해야 하는 경우 아래 함수로 변경할 수 있습니다.

```
void Merge(int x, int y){
    x = FindParent(x);
    y = FindParent(y);

    if(x == y)
        return;

    if(level[x] > level[y])
        swap(x, y);

    else
        parent[x] = y;

    if(level[x] == level[y])
        level[y]++;
}
```

단 레벨을 생각하는 문제는 잘 나오지 않습니다. 또한 레벨을 고려하게 되면 find 함수도 변경, 탐색에서 시간 복잡도가 늘어나게 됩니다.

고생하셨습니다. Union-Find를 아셨으니 `크루스칼 알고리즘`을 금방 이해하실 수 있을겁니다! 과정은 아래처럼 간단하게 나타낼 수 있습니다.

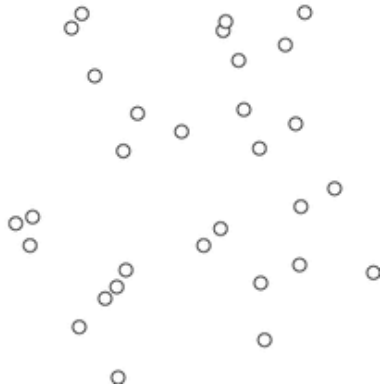
- 간선 배열을 일단 정렬
- 가장 짧은 간선부터 연결을 하는데 아래 과정의 연산을 함

    - 연결된 두 노드의 부모가 다르면 이 둘을 연결
    - 만일 부모가 같다면 연결 안함
- 이후 연결 횟수가 N-1이 되면 알고리즘을 종료

사실상 **Union-Find가 본체**라 크루스칼 알고리즘은 별게 없습니다. 그럼 아래 코드를 보겠습니다.

```
using namespace std;

pair<int, pair<int , int>> edge[100001];
int parents[10001];

int FindRoot(int x){
    if(x == parents[x])
        return x;
    return parents[x] = FindRoot(parents[x]);
}

void Merge(int x, int y){
    x = FindRoot(x);
    y = FindRoot(y);

    if(x == y)
        return;
    parents[x] = y;
}

int main(){
    // input...

    // 간선들을 정렬
    sort(edge, edge + e);

    // 연결될때 사용한 가중치를 저장할 변수
    int res = 0;

    // 간선 갯수만큼 크루스칼 알고리즘
    for(int i = 0; i < e; i++){
        if(FindRoot(edge[i].second.first) == FindRoot(edge[i].second.second))
            continue;

        // 부모가 다를시 연결
        Merge(edge[i].second.first, edge[i].second.second);
        res += edge[i].first;

        // 만일 연결된 수가 (노드 수-1) 이라면 알고리즘 종료
        edgenum++;
        if(edgenum == v - 1)
            break;
    }
}
```

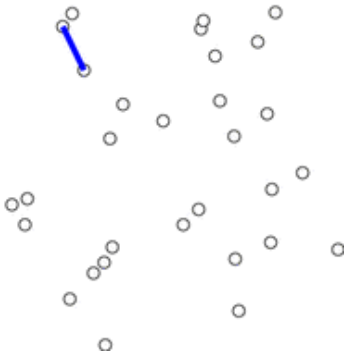
알고리즘 본체는 별게 없습니다! 이걸 그림으로 보면



이렇게 짧은 간선부터 차례차례 연결하는 모습을 볼 수 있습니다. 여기서 **부모(그룹)가 같다는건 사이클을 이룬다는 뜻**을 이용했다는걸 볼 수 있죠.

`크루스칼 알고리즘`은 간선을 위주로 하는 알고리즘이다 보니 Union-Find를 위와 같이 설계하면 $O(eloge)$ 입니다. (e : 간선의 수)

프림 알고리즘



이미 크루스칼 알고리즘을 봤으니 `프림 알고리즘`의 사진을 먼저 보겠습니다. 프림 알고리즘은 **시작점을 하나 잡고 연결지점을 확장하는 모습**입니다. 어차피 모든 노드가 연결되는 `최소 스패닝 트리`를 만드는 것이니 **시작지점은 아무거나** 잡아도 됩니다.

알고리즘의 흐름은 아래와 같습니다.

- 입력을 전부 받고 **우선순위 큐**를 선언해 줍니다. 그런 다음 시작 지점을 하나 정해 넣어줍니다.
  - 주로 1번 노드를 넣게 됩니다.
- 큐의 `top()` 원소와 연결된 노드들을 살펴봅니다.

    - 이미 방문한 노드면 넘어갑니다.
    - 아직 방문하지 않았다면 연결한 뒤 새 연결된 노드를 우선순위 큐에 삽입합니다.
- 큐가 빌 때까지 반복해 줍니다.

```
bool visit[10001];
vector<pair<int, int>> node[10001];

int main(){
    // input...

    priority_queue<pair<int, int>> q;

    // 시작지점 아무거나 잡기
    q.push({0, 1});

    // BFS 처럼 반복!
    while(!q.empty()){
        int curPos = q.top().second;
        int curCost = -q.top().first;
        q.pop();

        if(visit[curPos])
            continue;

        visit[curPos] = true;
        answer += curCost;

        for(int i = 0; i < node[curPos].size(); i++){
            int nextNode = node[curPos][i].first;
            int nextCost = node[curPos][i].second;

            q.push({-nextCost, nextNode});
        }
    }

    cout << answer << '\n';

}
```

시간 복잡도는 우선순위 큐를 사용할 때 $O(NlogN)$ 입니다!

두 알고리즘 비교

사실 큰 차이는 없지만 **사용하기 더 좋은 곳**은 각각 있습니다.

- 간선이 많은 경우 : `프림 알고리즘`
- 노드가 많은 경우 : `크루스칼 알고리즘`

추천 문제

1197번: 최소 스패닝 트리

첫째 줄에 정점의 개수 V(1 ≤ V ≤ 10,000)와 간선의 개수 E(1 ≤ E ≤ 100,000)가 주어진다. 다음 E개의 줄에는 각 간선에 대한 정보를 나타내는 세 정수 A, B, C가 주어진다. 이는 A번 정점과 B번 정점이 가중치 C인 간선으로

https://www.acmicpc.net/problem/1197

6497번: 전력난

입력은 여러 개의 테스트 케이스로 구분되어 있다. 각 테스트 케이스의 첫째 줄에는 집의 수 m과 길의 수 n이 주어진다. (1 ≤ m ≤ 200000, m-1 ≤ n ≤ 200000) 이어서 n개의 줄에 각 길에 대한 정보 x, y, z가 주어지는데, 이는 x번

https://www.acmicpc.net/problem/6497

16398번: 행성 연결

홍익 제국의 중심은 행성 T이다. 제국의 황제 윤석이는 행성 T에서 제국을 효과적으로 통치하기 위해서, N개의 행성 간에 플로우를 설치하려고 한다. 두 행성 간에 플로우를 설치하면 제국의 함선과 무역선들은 한 행성에서 다른 행성

https://www.acmicpc.net/problem/16398

보너스 문제

2887번: 행성 터널

https://www.acmicpc.net/problem/2887