

Minimum Spanning Tree (MST) - Kruskal'ın Algoritması

Elimizde kenarlarının ağırlığı olan yönsüz bir graph var. Bizim amacımız ise maliyeti minimum olan ve bütün noktaları içeren alt ağacı bulmak (maliyet kullanılan kenarların toplam ağırlığına eşittir.). Bu ağacımız MST olarak adlandırılır.

Önce MST'nin önemli özelliklerini tartışacağız ve daha sonra ise Kruskal'ın algoritmasına bağlı olarak en basit kodlama yöntemini göstereceğiz.

MST'nin Özellikleri

- Eğer bütün kenarların ağırlıkları farklıysa MST'miz özgündür. Diğer türlü verilen graphta birden fazla MST olabilir.
- The minimum spanning tree is also a skeleton with a minimum product of the weights of edges. (Proved it's easy enough to replace the weights of all edges on their logarithms)
- The minimum spanning tree is also a skeleton with a minimum weight of the heaviest edge. (This follows from the validity of Kruskal's algorithm)
- The skeleton of the maximum weight is sought is similar to the skeleton of minimum weight, enough to change the signs of all the ribs on the opposite and perform any of the minimum spanning tree algorithm.
- **KALAYCI düzeltmek, skeleton ne demek ben bilmemek araştırmak bulamamak**

Kruskal'ın Algoritması

Bu algoritma Kruskal tarafından 1956 yılında tanımlanmıştır.

Kruskal'ın algoritması öncelikle tüm noktaları farklı bir ağaca atar, ve sonra aşamalı olarak her tekrarda iki noktayı alır ve bunların içinde bulunduğu 2 ağacı birleştirir. Algoritma işleme başlamadan önce kenarlar ağırlıklarına göre artan sırada sıralanmıştır. Bu aşamada birleştirme işlemi başlar: kenarlar sırasıyla ele alınır ve bu kenarların birbirine bağladığı iki nokta eğer aynı ağaçta değilse kenarımız sonuca eklenir. En ağır kenar da işleme tabi tutulduktan sonra elimizde bütün noktaları içeren tek bir ağaç kalır ve bu ağacımız MST'dir.

En Basit Kodlanma Şekli

Bu kod yukarıda anlatıldığı şekilde kulanmıştır ve $O(M \log N + N^2)$ zamanda çalışır. Kenarlar $O(M \log N)$ işlemde sıralanır. tree_id arrayimiz bize noktaların hangi ağaca ait olduğunu gösterir. Böylece her bir kenar için ucundaki iki noktanın aynı ağaca ait olup olmadığını $O(1)$ işlemde anlarız. Son olarak da iki ağacın birleştirilmesi işlemi $O(N)$ zamanda yapılır ve toplam $N-1$ birleştirme işlemi yapmak zorunda olduğumuz için algoritmanın çalışma süresi $O(M \log N + N^2)$ 'dir.

```

int m;
vector <pair <int, pair <int, int>>> g (m); // kenar - nokta 1 ve 2

int cost = 0;
vector <pair <int, int>> res;

sort (g.begin (), g.end ()); // kenarları sırala
vector <int> tree_id (n);
for (int i = 0; i < n; ++ i) // her birini ayrı bir ağaca bağla
    tree_id [i] = i;
for (int i = 0; i < m; ++ i) // kenarları sırayla işle
{
    int a = g [i] .second.first, b = g [i] .second.second, l = g [i].first;
    if (tree_id [a] != tree_id [b]) //iki noktanın ağaçları farklıysa
    {
        cost += l; // sonuca kenarın ağırlığını
        res.push_back (make_pair (a, b)); // ve noktaları ekle
        int old_id = tree_id [b], new_id = tree_id [a];
        for (int j = 0; j < n; ++ j) // ağaçları birleştir
            if (tree_id [j] == old_id)
                tree_id [j] = new_id;
    }
}

```

Bu algoritmanın geliştirilmiş kodlanma şekli aşağıda verilmiştir.

Minimum Spanning Tree (MST) - Ayrık Kümeler Kullanılarak Yapılan Kruskal'ın Algoritması

Burada ayrık setler kullanılarak yapılan Kruskal'ın MST'sinin kodu verilecektir. Bu algoritma $O(M \log N)$ zamanda yapılmaktadır.

Tanım

Öncelikle kenarları azalmayan sırada sıralıyoruz. Bu işlemten sonra Ayrık Küme Birleşimi - Küme Oluşturma kullanarak toplam $O(N)$ zamanda kenarları ağaca ekliyoruz. Sıralamayı takip ederek her bir kenarın ucundaki iki noktanın aynı ağaçta olup olmadığına her seferinde $O(1)$ zamanda bakıyoruz (Toplamda $O(M)$). Son olarak iki ağacın birleşmesi de $O(1)$ zamanda olduğundan algoritmamızın toplam karmaşıklığı $O(M \log N + N + M) = O(M \log N)$.

Kodlama

Aşağıda randomlanmış Ayrık Küme Birleşimi kullanılmıştır.

```
vector<int> p (n);

int dsu_get (int v) { // noktanın hangi ağaca ait olduğu bulunurken ağaç aynı zamanda güncelleniyor
    return (v == p[v]) ? v : (p[v] = dsu_get (p[v]));
}

void dsu_unite (int a, int b) { // iki noktanın bağlı olduğu ağaçların birleştirilmesi
    a = dsu_get (a);
    b = dsu_get (b);
    if (rand() & 1)
        swap (a, b);
    if (a != b)
        p[a] = b;
}

// main kısmı

int m;
vector < pair < int, pair<int,int> > > g; // sırasıyla kenarın uzunluğu, 1. ve 2. nokta tutuluyor

// maliyeti hesaplama kısmı
int cost = 0;
vector < pair<int,int> > res;

sort (g.begin(), g.end()); // kenarları sırala
p.resize (n);
for (int i=0; i<n; ++i) // her bir nokta öncelikle ayrık bir küme
    p[i] = i;
for (int i=0; i<m; ++i) { // sırasıyla kenarlara bak
    int a = g[i].second.first, b = g[i].second.second, l = g[i].first;
    if (dsu_get(a) != dsu_get(b)) { // eğer iki ucu ayrı küme ise
        cost += l; // maliyeti ekle
        res.push_back (g[i].second); // kenarı sonuca al
        dsu_unite (a, b); // ve son olarak ağaçları birleştir
    }
}
```