Kruskal'ın Minimum Yayılan Ağaç Algoritması

Minimum Yayılan Ağaç Nedir?

Bağlı ve yönlendirilmemiş bir grafik verildiğinde , bu grafiğin yayılan ağacı , ağaç olan ve tüm köşeleri birbirine bağlayan bir altgraftır. Tek bir grafik birçok farklı yayılma ağacına sahip olabilir. Bir minimal yayılma (MST) bir ağırlıklı bağlanmış ve yönsüz grafik ya da en az ağırlıkça kapsayan ağaç daha az ağırlığa sahip bir kapsayan ağaç ya da her bir kapsayan ağacın ağırlığına eşittir. Yayılan ağacın ağırlığı, yayılan ağacın her bir kenarına verilen ağırlıkların toplamıdır.

Minimum yayılan bir ağacın kaç kenarı vardır?

Minimum yayılan bir ağacın (V - 1) kenarları vardır; burada V, verilen grafikteki köşe sayısıdır.

Minimum Yayılan Ağacın uygulamaları nelerdir?

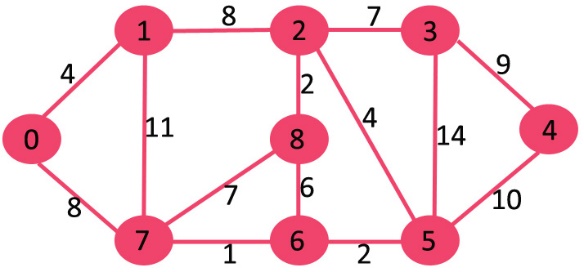
Kruskal'ın algoritmasını kullanarak MST'yi bulmak için adımlar

***1.****Tüm kenarları, ağırlıklarını azalmayacak şekilde sıralayın.****2.****En küçük kenarı seçin. Şimdiye kadar oluşan yayılan ağaçla bir döngü oluşturup oluşturmadığını kontrol edin. Döngü oluşmazsa, bu kenarı ekleyin. Aksi halde atın.****3.****Yayılan ağaçta (V-1) kenarlar olana kadar 2. adımı tekrarlayın.*

2. adım, döngüyü algılamak için [Union-Find algoritmasını](https://www.geeksforgeeks.org/union-find/) kullanır . Bu nedenle, aşağıdaki yayını önkoşul olarak okumanızı öneririz.  
[Birlik-Bul Algoritması | Set 1 (Bir Grafikte Döngüyü Algıla)](https://www.geeksforgeeks.org/union-find/)  
[Union-Find Algoritması | Set 2 (Sıra ve Yol Sıkıştırmasına Göre Birlik)](https://www.geeksforgeeks.org/union-find-algorithm-set-2-union-by-rank/)

Algoritma Açgözlü Bir Algoritmadır. Açgözlü Seçim, şimdiye kadar inşa edilen MST'de bir döngüye neden olmayan en küçük ağırlık kenarını seçmektir. Bir örnekle anlayalım: Aşağıdaki giriş grafiğini düşünün.

Algoritma Açgözlü Bir Algoritmadır. Açgözlü Seçim, şimdiye kadar inşa edilen MST'de bir döngüye neden olmayan en küçük ağırlık kenarını seçmektir. Bir örnekle anlayalım: Aşağıdaki giriş grafiğini düşünün.



Grafik 9 köşe ve 14 kenar içeriyor. Böylece, oluşturulan minimum yayılan ağaç (9 - 1) = 8 kenara sahip olacaktır.

Sıralamadan sonra:

Ağırlık Src Dest

1 7 6

2 8 2

2 6 5

4 0 1

4 2 5

6 8 6

7 2 3

7 7 8

8 0 7

8 1 2

9 3 4

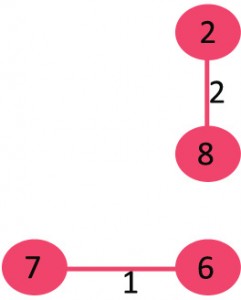
10 5 4

11 1 7

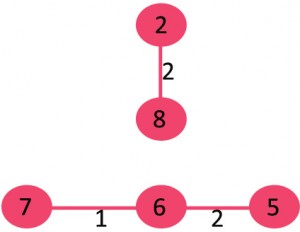
14 3 5

Şimdi sıralanan kenar listesinden tüm kenarları tek tek seçin  
**1.**Kenar 7-6'yı seçin: Hiçbir döngü oluşmaz, ekleyin.

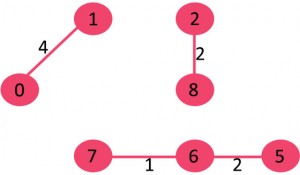
**2.**Çekme kenarı 8-2: Döngü oluşmaz, dahil edin.

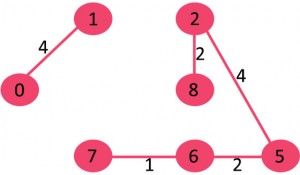


**3.**Seçim kenarı 6-5: Hiçbir döngü oluşmaz, dahil edin.



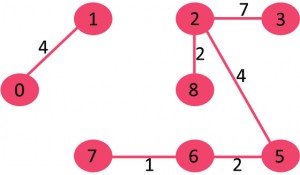
**4.**Kenar 0-1'i seçin: Hiçbir döngü oluşmaz, ekleyin.



**5.**Kenar 2-5'i seçin: Hiçbir döngü oluşmaz, ekleyin.  


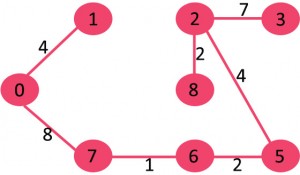
**6.**Kenar 8-6'yı seçin: Bu kenarı dahil etmek döngü ile sonuçlandığından atın.

**7.**Kenar 2-3'ü seçin: Hiçbir döngü oluşmaz, dahil edin.



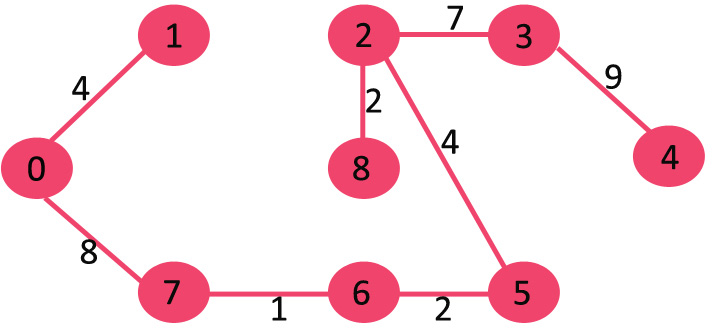
**8.**Kenar 7-8'i seçin: Bu kenarın dahil edilmesi döngü ile sonuçlandığından atın.

**9.**Kenar 0-7'yi seçin: Hiçbir döngü oluşmaz, dahil edin.



**10.**Kenar 1-2'yi seçin: Bu kenarın dahil edilmesi döngü ile sonuçlandığından atın.

**11.**Kenar 3-4'ü seçin: Hiçbir döngü oluşmaz, ekleyin.



Dahil edilen kenar sayısı eşit olduğundan (V - 1), algoritma burada durur.

KODLAMA

using namespace std;

class Edge

{

    public:

    int src, dest, weight;

};

//bağlı, yönlendirilmemiş bir yapıyı temsil eder

class Graph

{

    public:

    // V-> Number of vertices, E-> Number of edges

    int V, E;

Grafik bir kenar dizisi olarak temsil edililir

// Grafik yönlendirilmemiş

// src'den dest'e de dest'ten kenar

// to src. Her ikisi de burada 1 kenardır.

Edge\* edge;

};

Graph\* createGraph(int V, int E)

{

    Graph\* graph = new Graph;

    graph->V = V;

    graph->E = E;

    graph->edge = new Edge[E];

    return graph;

}

//union-find için bir alt kümeyi temsil eder.

class subset

{

    public:

    int parent;

    int rank;

};

// İ kümesini bulmak için yardımcı işlev

int find(subset subsets[], int i)

{

//root bulup yapması

if (subsets[i].parent != i)

        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;

}

// X ve Y kümesinin birleşimini yapar .

void Union(subset subsets[], int x, int y)

{

    int xroot = find(subsets, x);

    int yroot = find(subsets, y);

// yüksek kök altında küçük rütbe ağaçı takmak

if (subsets[xroot].rank < subsets[yroot].rank)

        subsets[xroot].parent = yroot;

    else if (subsets[xroot].rank > subsets[yroot].rank)

        subsets[yroot].parent = xroot;

//Eğer rütbeler aynıysa bir tane root olarak yapın

else

    {

        subsets[yroot].parent = xroot;

        subsets[xroot].rank++;

    }

}

// İki kenarı ağırlığına göre karşılaştır.

int myComp(const void\* a, const void\* b)

{

    Edge\* a1 = (Edge\*)a;

    Edge\* b1 = (Edge\*)b;

    return a1->weight > b1->weight;

}

// Kruskalı kullanarak MST’Yİ oluşturmak için işlev.

{

    int V = graph->V;

    Edge result[V]; // sonuçtaki mstyi saklar

    int e = 0; // dizin değişkeni ; sonuç için

    int i = 0; // dizin değişkeni ; sıralı kenar için

qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp); // azalmayan kenarları sırala //verilen grafiği değiştirip kopyasını oluşturmak

//subests oluşturmak için bellek ayırma

    subset \*subsets = new subset[( V \* sizeof(subset) )];

//Tek öğeyle V alt kümelerini oluşturma

{

        subsets[v].parent = v;

        subsets[v].rank = 0;

    }

// alıncak kenar sayısı V-1 e eşit.

 while (e < V - 1 && i < graph->E)

    {

// en küçük kenarı seçip artır

// sonraki için dizin

 Edge next\_edge = graph->edge[i++];

        int x = find(subsets, next\_edge.src);

        int y = find(subsets, next\_edge.dest);

// kenar döngüye dahil olmazsa

//sonuca dahil edip dizini artır

// diğer kenar için sonuç

  if (x != y)

        {

            result[e++] = next\_edge;

            Union(subsets, x, y);

        }

}

//sonuç içeriğini yazdır

    cout<<"Following are the edges in the constructed MST\n";

    for (i = 0; i < e; ++i)

        cout<<result[i].src<<" -- "<<result[i].dest<<" == "<<result[i].weight<<endl;

    return;

}

// sürücü kodu

int main()

{

int V = 4; // köşe sayısı

    int E = 5; // kenar sayısı

    Graph\* graph = createGraph(V, E);

// kenar ekle 0-1

graph->edge[0].src = 0;

    graph->edge[0].dest = 1;

    graph->edge[0].weight = 10;

// kenar ekle 0-2

    graph->edge[1].src = 0;

    graph->edge[1].dest = 2;

    graph->edge[1].weight = 6;

// kenar ekle 0-3

 graph->edge[2].src = 0;

    graph->edge[2].dest = 3;

    graph->edge[2].weight = 5;

// kenar ekle 1-3

  graph->edge[3].src = 1;

    graph->edge[3].dest = 3;

    graph->edge[3].weight = 15;

// kenar ekle 2-3

    graph->edge[4].src = 2;

    graph->edge[4].dest = 3;

    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;

}

İnşa edilen MST'nin kenarları aşağıdadır

2-3 == 4

0-3 == 5

0-1 == 10

**Zaman Karmaşıklığı:** O (ElogE) veya O (ElogV). Kenarların sıralanması O (ELogE) zamanını alır. Sıralamadan sonra tüm kenarları tekrarlıyoruz ve birleştirme algoritması uyguluyoruz. Bulma ve birleştirme işlemleri en az O (LogV) zaman alabilir. Dolayısıyla toplam karmaşıklık O (ELogE + ELogV) süresidir. E değeri en az O (V 2 ) olabilir, bu nedenle O (LogV) O (LogE) aynıdır. Bu nedenle, toplam zaman karmaşıklığı O (ElogE) veya O (ElogV)