Bölüm 17: Minimum Kapsayan Ağaçlar

C ile Veri Yapıları ve Algoritmalar

Kapsayan ağaç · Eş biçimlilik · Açgözlü yöntem · Kruskal · Union-find · Prim · Heap

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Tanımlamak

Kapsayan ağacı, ağaç/ağaç dışı kenarları ve derinlik/genişlik öncelikli kapsayan ağaçları açıklamak.

Saymak

Bir çizgenin kapsayan ağaçlarını üretme yollarını ve eş biçimlilik (izomorfizm) kavramını tartışmak.

Kruskal

Kenarları artan maliyetle seçip döngüyü **union-find** ile yakalayan Kruskal'ı C ile izlemek.

Prim

Tek öbeği büyüten Prim'i $O(n^2)$ dizilerle ve heap ile gerçeklemek.

Analiz

$O(e \log e)$, $O(n^2)$, $O(e \log n)$ sınırlarını karşılaştırıp yoğun/seyrek çizgeye göre seçmek.

Yaklaşım

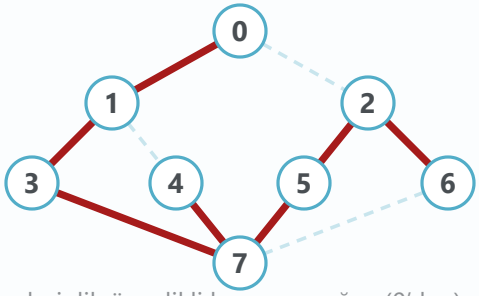
"Veri yapısını keşfetmek": min-heap ve kümeler algoritmayı nasıl hızlandırır?

Gezinme Kenarları İkiye Ayırır

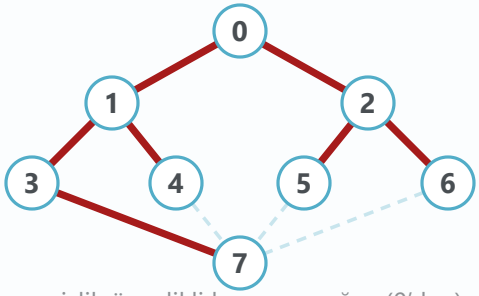
Bağlantılı bir çizgede DFS ya da BFS ile tüm düğümler ziyaret edilince kenarlar iki gruba ayrılır:

T: ağaç kenarları (tree)
İşlenen, yeni bir düğüme götüren kenarlar (kalın kırmızı). Tam n − 1 tanedir.

N: ağaç dışı kenarlar (non-tree)
İşlenmeyen kenarlar (kesikli). DFS'de geri (*back*), BFS'de çapraz (*cross*) kenar olarak da sınıflanır.



derinlik öncelikli kapsayan ağaç (0'dan)

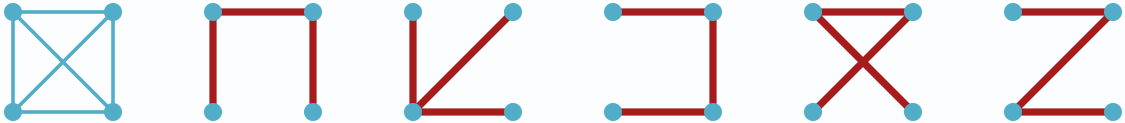


genişlik öncelikli kapsayan ağaç (0'dan)

DFS: N = {(0,2), (1,4), (6,7)} · BFS: N = {(4,7), (5,7), (6,7)}

Tanım: Kapsayan Ağaç (Spanning Tree)

Tanım: Bir kapsayan ağaç, tüm düğümleri birbirine bağlayan, çizgenin kenarlarından oluşan **ağaçtır**: n düğüm, **n − 1** kenar, döngü yok.



K4 a) b) c) d) e)

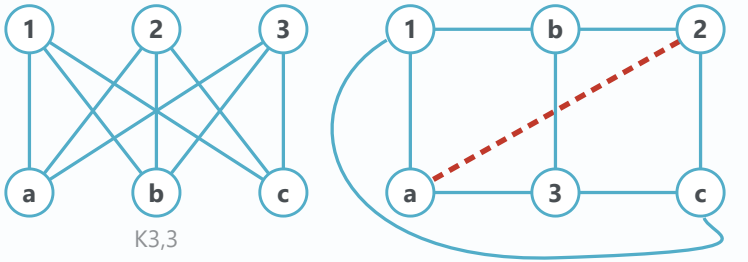
Eş biçimlilik (izomorfizm)

K4 tam çizgedir. a ve b **farklı** iki kapsayan ağaçtır (a bir yol, b bir yıldız); c, d, e ise a ile **eş biçimlidir**: a ≡ c ≡ d ≡ e.

Kaç tane?

K4'ün 16 kapsayan ağacı vardır (Cayley: n^{n-2} = 4^2 = 16): 12 yol biçimli + 4 yıldız. Ağırlıklar devreye girince "kaç tane?" sorusu yeniden ele alınır.

Eş Biçimlilik, Çizim ve Düzlemsellik



K3,3

başka bir çizimi: tek kesişme (kırmızı)

Aynı çizge ($K_{3,3}$), iki farklı çizim

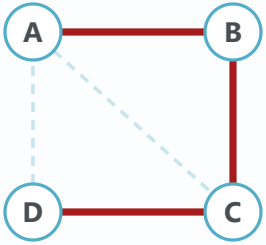
- Eş biçimliliği sınamak için DFS/BFS'den yararlanılır; ama genelde **zor** bir problemdir.
- Kenarları kesişmeden çizilebilen çizge **düzlemseldir** (*planar*). $K_{3,3}$ düzlemsel değildir: en az bir kesişme gerekir.
- Devre tasarımında: $K_{3,3}$ biçimli bir devre en az **iki katman** ister.

Hocadan: "Güzel bir 4. sınıf projesi olabilir: 3 boyutlu gösterimle a ≡ e gösterimini yapan program, algoritma... Hangi düğümler yer değiştirmeli?"

Hocadan: "Bunlar zor problemler... Master ya da doktora yapılırsa bunlarla uğraşılabilir. Bu arada "Master ya da doktora yapılmadıkça problem çözülemez." demiş oluyoruz."

Soru: Kaç Farklı Kapsayan Ağaç Vardır?

e kenar arasından n − 1 kenar **C(e, n − 1)** biçimde seçilir; her seçimde döngü olup olmadığı denetlenir ve eş biçimliler ayıklanır. Burada C(5, 3) = 10 seçimin 8'i ağaçtır.



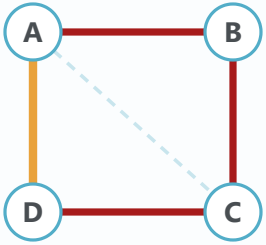
başlangıç ağacı T (3 kenar)

Önce **bir** kapsayan ağaç bulunur (ör. DFS ile).

Bu algoritma uzundur; bu yüzden tercih etmek mantıklı değildir. (Kesin sayı için Kirchhoff matris-ağaç teoremi kullanılır — ileri konu.)

Soru: Kaç Farklı Kapsayan Ağaç Vardır?

e kenar arasından n − 1 kenar **C(e, n − 1)** biçimde seçilir; her seçimde döngü olup olmadığı denetlenir ve eş biçimliler ayıklanır. Burada C(5, 3) = 10 seçimin 8'i ağaçtır.



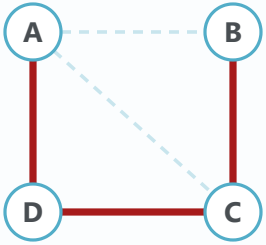
+ (D,A) → döngü A-B-C-D-A

Ağaç dışı bir kenar eklenir: tam **bir** döngü oluşur.

Bu algoritma uzundur; bu yüzden tercih etmek mantıklı değildir. (Kesin sayı için Kirchhoff matris-ağaç teoremi kullanılır — ileri konu.)

Soru: Kaç Farklı Kapsayan Ağaç Vardır?

e kenar arasından n − 1 kenar **C(e, n − 1)** biçimde seçilir; her seçimde döngü olup olmadığı denetlenir ve eş biçimliler ayıklanır. Burada C(5, 3) = 10 seçimin 8'i ağaçtır.



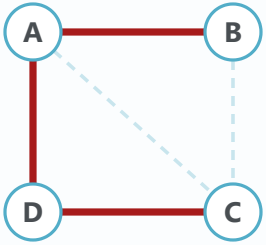
(A,B) çıkarıldı → yeni ağaç

Döngüdeki (A,B) çıkarılır → farklı bir kapsayan ağaç. Sonra geri eklenip bir sonraki çıkarılır.

Bu algoritma uzundur; bu yüzden tercih etmek mantıklı değildir. (Kesin sayı için Kirchhoff matris-ağaç teoremi kullanılır — ileri konu.)

Soru: Kaç Farklı Kapsayan Ağaç Vardır?

e kenar arasından n − 1 kenar **C(e, n − 1)** biçimde seçilir; her seçimde döngü olup olmadığı denetlenir ve eş biçimliler ayıklanır. Burada C(5, 3) = 10 seçimin 8'i ağaçtır.



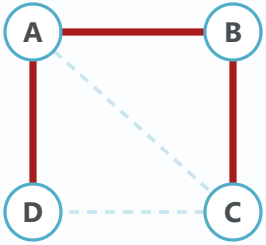
(B,C) çıkarıldı → yeni ağaç

Döngüdeki (B,C) çıkarılır → farklı bir kapsayan ağaç. Sonra geri eklenip bir sonraki çıkarılır.

Bu algoritma uzundur; bu yüzden tercih etmek mantıklı değildir. (Kesin sayı için Kirchhoff matris-ağaç teoremi kullanılır — ileri konu.)

Soru: Kaç Farklı Kapsayan Ağaç Vardır?

e kenar arasından n − 1 kenar **C(e, n − 1)** biçimde seçilir; her seçimde döngü olup olmadığı denetlenir ve eş biçimliler ayıklanır. Burada C(5, 3) = 10 seçimin 8'i ağaçtır.



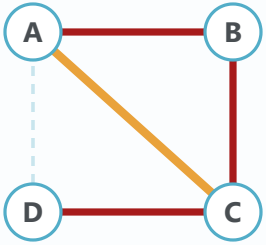
(C,D) çıkarıldı → yeni ağaç

Döngüdeki (C,D) çıkarılır → farklı bir kapsayan ağaç. Sonra geri eklenip bir sonraki çıkarılır.

Bu algoritma uzundur; bu yüzden tercih etmek mantıklı değildir. (Kesin sayı için Kirchhoff matris-ağaç teoremi kullanılır — ileri konu.)

Soru: Kaç Farklı Kapsayan Ağaç Vardır?

e kenar arasından n − 1 kenar **C(e, n − 1)** biçimde seçilir; her seçimde döngü olup olmadığı denetlenir ve eş biçimliler ayıklanır. Burada C(5, 3) = 10 seçimin 8'i ağaçtır.

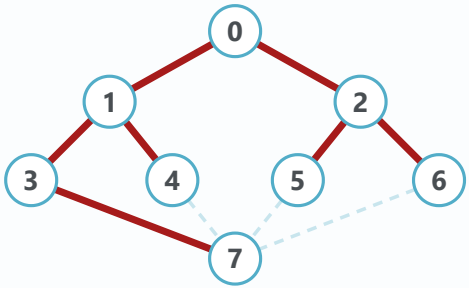


toplam 8 farklı ağaç

Bütün ağaç dışı kenarlar için tekrarlanır. Bu çizgede 8 kapsayan ağaç var; ama yöntem uzun ve tekrar üretir.

Bu algoritma uzundur; bu yüzden tercih etmek mantıklı değildir. (Kesin sayı için Kirchhoff matris-ağaç teoremi kullanılır — ileri konu.)

Kapsayan Ağacın Özellikleri

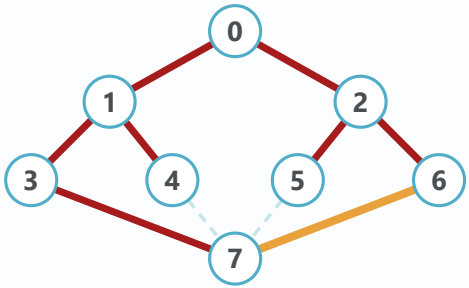


BFS kapsayan ağacı: 8 düğüm, 7 kenar (kalın kırmızı); ağaç dışı 3 kenar kesikli.

Adım 1 / 3

- Tam **n − 1** kenar, döngü yok, bağlantılı — üçünden ikisi üçüncüsünü gerektirir.
- Her düğüm çifti arasında **tek** yol vardır.
- Bir ağaç dışı kenar eklemek tek bir döngü oluşturur; döngüden herhangi bir kenarı çıkarmak yeni bir kapsayan ağaç verir (kapsayan ağaçları üretme yöntemi buna dayanır).
- Uygulama: ağdaki gereksiz bağlantıları atmak, yayın (*broadcast*) ağacı, elektrik devre denklemleri (Kirchhoff).

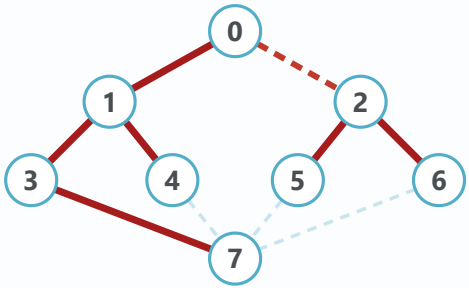
Kapsayan Ağacın Özellikleri



Ağaç dışı (6,7) eklenirse **tam bir döngü** oluşur: 6-2-0-1-3-7-6. Ağaç, ağaç olmaktan çıkar.

- Tam **n − 1** kenar, döngü yok, bağlantılı — üçünden ikisi üçüncüsünü gerektirir.
- Her düğüm çifti arasında **tek** yol vardır.
- Bir ağaç dışı kenar eklemek tek bir döngü oluşturur; döngüden herhangi bir kenarı çıkarmak yeni bir kapsayan ağaç verir (kapsayan ağaçları üretme yöntemi buna dayanır).
- Uygulama: ağdaki gereksiz bağlantıları atmak, yayın (*broadcast*) ağacı, elektrik devre denklemleri (Kirchhoff).

Kapsayan Ağacın Özellikleri



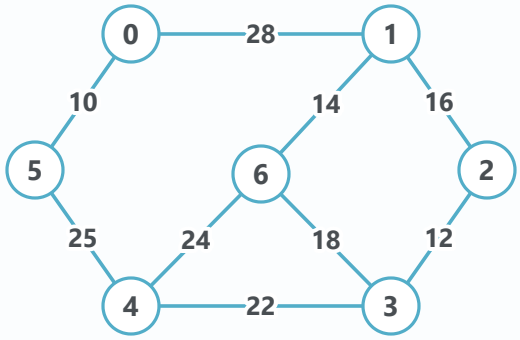
Herhangi bir ağaç kenarı, ör. (0,2), silinirse ağaç **ikiye bölünür**: {2, 5, 6} kopar. Kapsayan ağaç “en az kenarlı” bağlantılı alt çizgedir.

- Tam **n − 1** kenar, döngü yok, bağlantılı — üçünden ikisi üçüncüsünü gerektirir.
- Her düğüm çifti arasında **tek** yol vardır.
- Bir ağaç dışı kenar eklemek tek bir döngü oluşturur; döngüden herhangi bir kenarı çıkarmak yeni bir kapsayan ağaç verir (kapsayan ağaçları üretme yöntemi buna dayanır).
- Uygulama: ağdaki gereksiz bağlantıları atmak, yayın (*broadcast*) ağacı, elektrik devre denklemleri (Kirchhoff).

MİNİMUM MALİYET

Minimum Maliyetli Kapsayan Ağaç (MST)

- Yönsüz çizgede kenarlar **ağırlıklı** olabilir: G(V, E) yerine **G(V, E, w)**.
- Kapsayan ağaçlar içinde toplam kenar maliyeti **en az** olan(lar): minimum kapsayan ağaç.
- **Ör. a:** ağdaki kenar maliyetleri bilgisayarlar arası bağlantı maliyetidir.
- **Ör. b:** 7 köye elektrik/fiber optik: fizibilite hangi köyden hangisine kaç km kablo gerektiğini vermiş. Bağlantı olmayan köyler arası ∞ (yazılmıyor).



Örnek çizge

Hocadan: "Minimum kapsayan ağaçları bulurken kenar seçerken döngüden kaçınıyoruz... "Aşağı Tepecik Köyü'nden elektrik gelmezse Yukarı Tepecik'ten gelsin." Yok!"

MİNİMUM MALİYET

Algoritmalar ve Açgözlü (Greedy) Yaklaşım

Algoritmalar

Kruskal, Prim, Sollin (Borůvka); ayrıca *reverse-delete*. Bu bölümde ilk ikisi.

Programlama yöntemleri

Geri izleme (*backtracking*), açgözlü (*greedy*), dinamik programlama…

Greedy = ısrarcı

Her adımda o an için en iyi karar verilir ve **sonra değiştirilmez**; geri dönmez.

Üç kural

1) Döngü oluşturan kenar alınmaz. 2) En az maliyetli tercih edilir. 3) n düğümde tam **n − 1** kenar seçilir.

Neden doğru? (ek) *Kesit özelliği:* düğümleri ikiye bölen herhangi bir kesitte, kesiti geçen en ucuz kenar bir MST'de bulunur. Kruskal ve Prim her adımda böyle bir kenar seçer; bu yüzden açgözlü seçim hiç pişman olmaz.

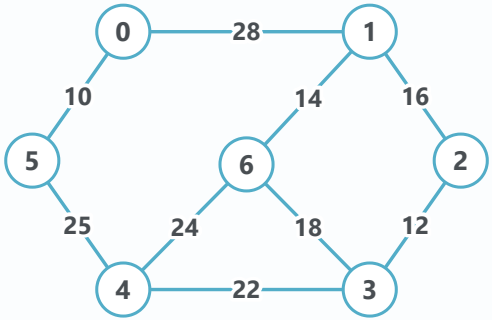
"Güzellik, fazlalıktan arınmışlıktır." — Michelangelo

Kruskal Algoritması (1956)

```
T = {};                    /* seçilen kenarlar */
while (|T| < n-1 && E ≠ ∅) {
    E'den en az maliyetli (v,w) kenarını seç;
    E = E - {(v,w)};
    if ((v,w) T içinde döngü oluşturmuyorsa)
        T = T ∪ {(v,w)};
    else
        (v,w)'yi bırak;
}
if (|T| < n-1)
    printf("minimum kapsayan ağaç yok\n");
```

- T: seçilen kenarlar, E: kalan kenarlar, |T|: seçilen kenar sayısı.
- Döngü n − 1 kenar seçilene kadar döner; henüz n − 1 seçilmemiş ama seçilebilecek kenar kalmamışsa da biter → çizge bağlantılı değildir.
- Kruskal'da seçilen kenarlar **ayrık öbekler** (orman) oluşturabilir; en sonda tek öbek olur.
- Joseph B. Kruskal, 1956; doğruluğunun ispatı vardır. "İspatı okununca daha iyi anlaşılır."

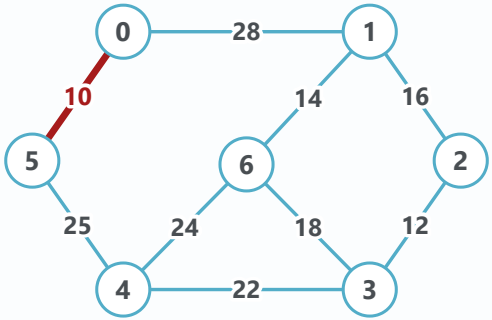
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	
(2,3)	12	
(1,6)	14	
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

Kenarlar maliyete göre sıralandı; her düğüm ayrı bir öbek.

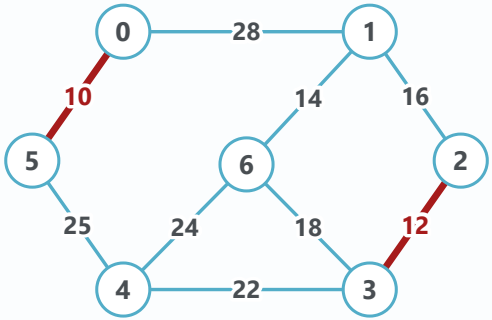
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	
(1,6)	14	
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(0,5) = 10: uçlar farklı öbeklerde → **ağaca eklendi**. Toplam 10.

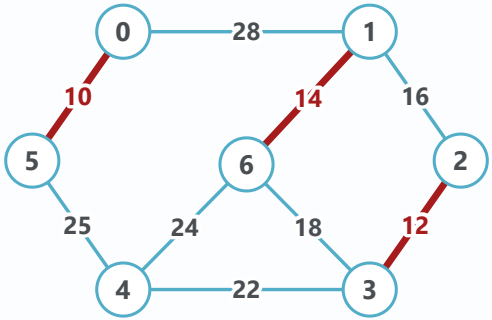
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(2,3) = 12: uçlar farklı öbeklerde → **ağaca eklendi**. Toplam 22.

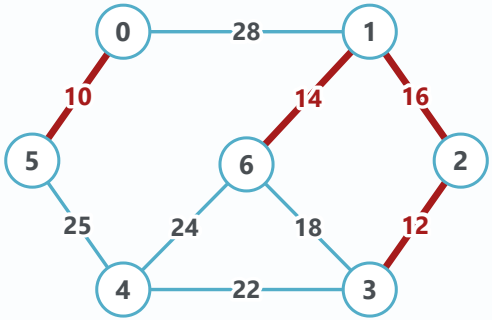
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	Ağaca ekle
(1,2)	16	
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(1,6) = 14: uçlar farklı öbeklerde → **ağaca eklendi**. Toplam 36.

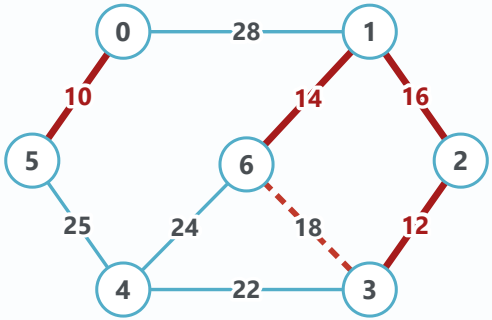
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	Ağaca ekle
(1,2)	16	Ağaca ekle
(3,6)	18	
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(1,2) = 16: uçlar farklı öbeklerde → **ağaca eklendi**. Toplam 52.

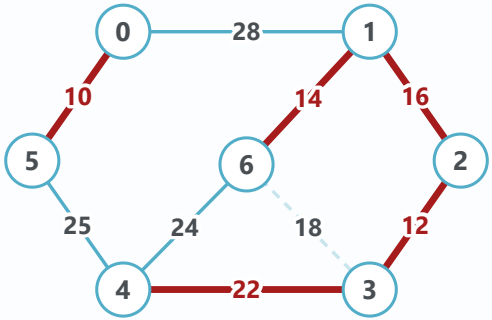
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	Ağaca ekle
(1,2)	16	Ağaca ekle
(3,6)	18	Döngü var, ekleme
(3,4)	22	
(4,6)	24	
(4,5)	25	
(0,1)	28	

(3,6) = 18: uçlar **aynı öbekte** → döngü oluşur, bırakılır.

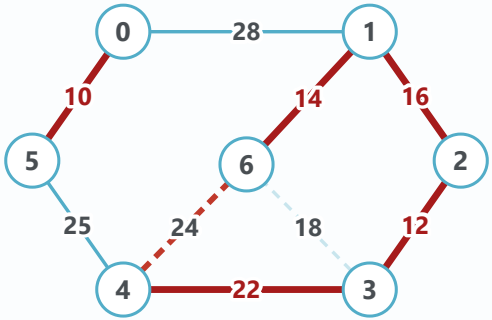
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	Ağaca ekle
(1,2)	16	Ağaca ekle
(3,6)	18	Döngü var, ekleme
(3,4)	22	Ağaca ekle
(4,6)	24	
(4,5)	25	
(0,1)	28	

(3,4) = 22: uçlar farklı öbeklerde → **ağaca eklendi**. Toplam 74.

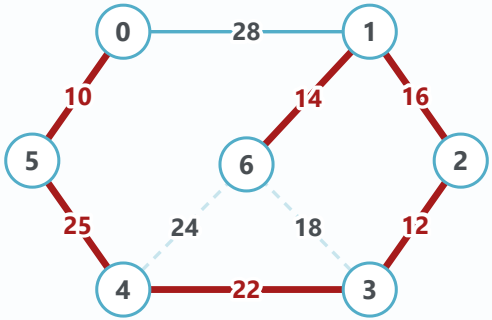
Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	Ağaca ekle
(1,2)	16	Ağaca ekle
(3,6)	18	Döngü var, ekleme
(3,4)	22	Ağaca ekle
(4,6)	24	Döngü var, ekleme
(4,5)	25	
(0,1)	28	

(4,6) = 24: uçlar **aynı öbekte** → döngü oluşur, bırakılır.

Kruskal Adım Adım



Kenar	w	İşlem
(0,5)	10	Ağaca ekle
(2,3)	12	Ağaca ekle
(1,6)	14	Ağaca ekle
(1,2)	16	Ağaca ekle
(3,6)	18	Döngü var, ekleme
(3,4)	22	Ağaca ekle
(4,6)	24	Döngü var, ekleme
(4,5)	25	Ağaca ekle
(0,1)	28	

(4,5) = 25: uçlar farklı öbeklerde → **ağaca eklendi**. Toplam 99. n − 1 = 6 kenar seçildi: **bitti** (28'e hiç bakılmadı).

Kruskal Nasıl Kodlanmaz?

- Gözlem: önce 2 bileşen oldu, sonra 3, sonra tekrar 2'ye düştü. "Yakaladık! Seçtiğim kenarları komşuluk listesinde tutarım."
- Seçilmeyen kenarlar bir dizide. **Kötü:** her seferinde en azı ara, sil, kalanları kaydır. **Daha iyi:** kenarları maliyete göre sıralı tut (uç1, uç2, maliyet).
- (0,5) geldi: listelerde aratırım, ne 0 var ne 5 → yeni liste. (3,2), (6,1) de yeni listelere. (1,2): farklı listelerde → döngü yok, iki listeyi birleştir.
- (3,6): ikisi de aynı listede → döngü! Doğru çalışır ama **sürekli bir arama maliyeti** var.

> **Hocadan:** "Döngüyü tespit ettik, ama yanlış algoritmayı anlatıyorum. Böyle kodlanmaz, yoksa puan alınamaz. "Böyle kodlanmaz!" diyorum, ama doğru çalışıyor, onu anlatmaya çalışıyorum."

> **Hocadan:** "Kruskal algoritmasının kodlamasını ödev olarak sorabilirim. Bakalım, gereken veri yapıları keşfedilebilecek mi? Onu merak ediyorum."

KRUSKAL

Kruskal Nasıl Kodlanır? İki Soru, İki Veri Yapısı

1) En az maliyetli kenar?

Min öncelikli kuyruk (min-heap). Kenarları tek tek eklemek O(e log e); diziyi aşağıdan yukarıya heap kurmayla heap'e çevirmek O(e). Ya da kenarlar bir kez sıralanır (`qsort`).

2) Döngü oluşturur mu?

Uçlar **iki ayrı kümenin** öğesiyse döngü olmaz: **union-find**. `kume_bul(3)` ile `kume_bul(2)` aynı kökü mü veriyor?

> **Hocadan:** "Arka planda hoca hangi veri yapılarını anlatmış da haberimiz yokmuş!! Biz de gittik, liste tutuyoruz... İlkokulda öğretilen kümeler burada nasıl da işimize yarıyormuş."

Maliyet: $|E|\cdot\lg|E| + |V|\cdot(\lg|E| + \Theta(1))$; aşağıdan yukarıya kurmayla $2|E| + |V|\cdot\lg|E|$ ($\Theta(1)$: ters Ackermann $\alpha \le \lg|E|$). En kötü durumda döngü veren kenarlar yüzünden $n - 1$ değil e kez *deleteMin* gerekebilir; güvenli üst sınır **O(e log e) = O(e log n)**.

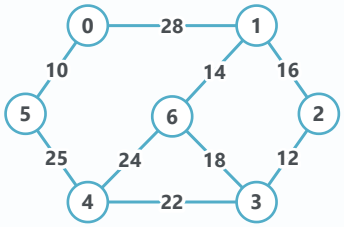
C: Kümeler (Union-Find) ve Karşılaştırma

```c
typedef struct { int u, v, w; } kenar;

int kume_bul(int ata[], int i)   /* kok: ata < 0 */
{
    while (ata[i] >= 0) i = ata[i];
    return i;
}
void birlestir(int ata[], int a, int b)   /* a, b kok */
{
    if (ata[a] <= ata[b]) {   /* a daha buyuk kume */
        ata[a] += ata[b];  ata[b] = a;
    } else {
        ata[b] += ata[a];  ata[a] = b;
    }
}
int kars(const void *x, const void *y)
{
    return ((const kenar *)x)->w - ((const kenar *)y)->w;
}
```

- `ata[i] < 0` : i köktür ve `-ata[i]` küme boyutudur (başta hepsi −1).
- **Ağırlıklı birleşim:** küçük küme büyüğün altına bağlanır → ağaç yüksekliği O(log n), `kume_bul` O(log n).
- Yol kısaltma (*path compression*) da eklenirse işlem başına amortize ≈ O(α(n)) — pratikte sabit.
- `kars` : `qsort` için maliyet karşılaştırması (küçük tam sayılar; taşma yok).

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

Her düğüm kendi kümesinin kökü (ata = −1). Kenarlar maliyete göre sıralanır: O(e log e).

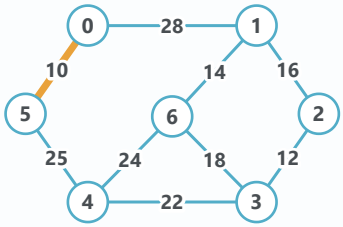
Adım 1 / 9

```
▶ for (i = 0; i < n; i++) ata[i] = -1;
▶ qsort(k, e, sizeof(kenar), kars);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = kume_bul(ata, k[i].u);
5     int b = kume_bul(ata, k[i].v);
6     if (a != b) {                  /* dongu yok */
7         T[t++] = k[i];
8         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`n = 7` `e = 9` `t = 0`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5	2-3	1-6	1-2	3-6	3-4	4-6	4-5	0-1
10	12	14	16	18	22	24	25	28

▲ i

i = 0: (0,5) = 10. `kume_bul` : 0 → kök 0, 5 → kök 5.

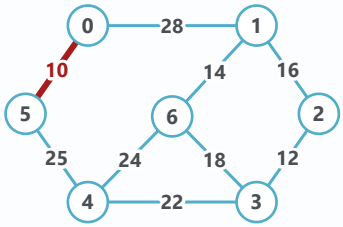
Adım 2 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
▶ for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶     int a = kume_bul(ata, k[i].u);
▶     int b = kume_bul(ata, k[i].v);
6     if (a != b) {                  /* dongu yok */
7         T[t++] = k[i];
8         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 0` `t = 0` `k[i] = (0,5,10)` `a = 0` `b = 5`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-2	-1	-1	-1	-1	0	-1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5	2-3	1-6	1-2	3-6	3-4	4-6	4-5	0-1
10	12	14	16	18	22	24	25	28

▲ i

0 ≠ 5 → döngü yok: T'ye eklenir, kümeler birleşir (boyutu büyük olan kök kalır). t = 1.

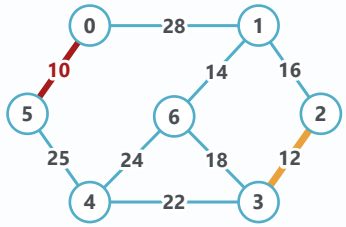
Adım 3 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = kume_bul(ata, k[i].u);
5     int b = kume_bul(ata, k[i].v);
▶     if (a != b) {                  /* dongu yok */
▶         T[t++] = k[i];
▶         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 0` `t = 1` `k[i] = (0,5,10)` `a = 0` `b = 5`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-2	-1	-1	-1	-1	0	-1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 1: (2,3) = 12. `kume_bul` : 2 → kök 2, 3 → kök 3.

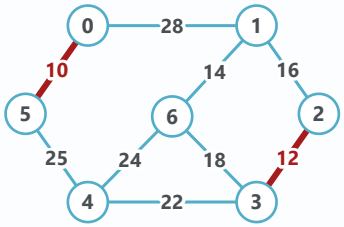
Adım 4 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
▶ for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶     int a = kume_bul(ata, k[i].u);
▶     int b = kume_bul(ata, k[i].v);
6     if (a != b) {                  /* dongu yok */
7         T[t++] = k[i];
8         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 1` `t = 1` `k[i] = (2,3,12)` `a = 2` `b = 3`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-2	-1	-2	2	-1	0	-1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

2 ≠ 3 → döngü yok: T'ye eklenir, kümeler birleşir (boyutu büyük olan kök kalır). t = 2.

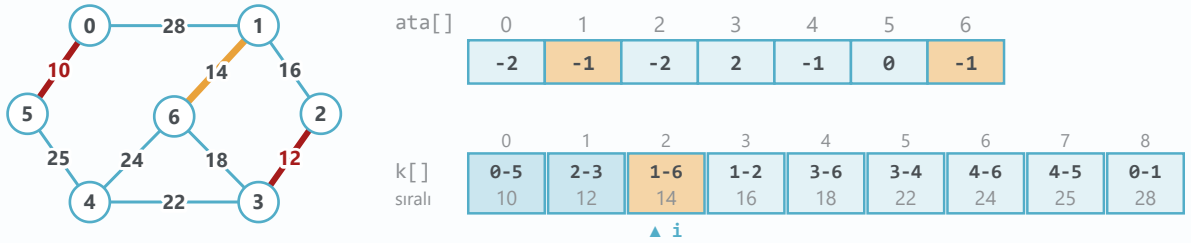
Adım 5 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = kume_bul(ata, k[i].u);
5      int b = kume_bul(ata, k[i].v);
▶      if (a != b) {                  /* dongu yok */
▶          T[t++] = k[i];
▶          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 1` `t = 2` `k[i] = (2,3,12)` `a = 2` `b = 3`

Kod İzleme: `kruskal()` ana döngüsü



i = 2: (1,6) = 14. `kume_bul` : 1 → kök 1, 6 → kök 6.

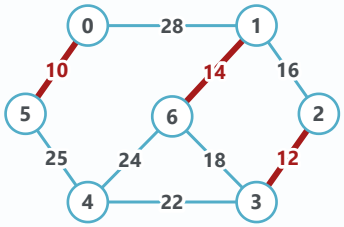
Adım 6 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
▶ for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶     int a = kume_bul(ata, k[i].u);
▶     int b = kume_bul(ata, k[i].v);
6     if (a != b) {                  /* dongu yok */
7         T[t++] = k[i];
8         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 2` `t = 2` `k[i] = (1,6,14)` `a = 1` `b = 6`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-2	-2	-2	2	-1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

1 ≠ 6 → döngü yok: T'ye eklenir, kümeler birleşir (boyutu büyük olan kök kalır). t = 3.

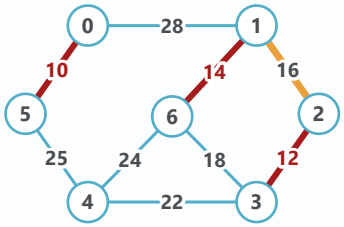
Adım 7 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = kume_bul(ata, k[i].u);
5      int b = kume_bul(ata, k[i].v);
▶      if (a != b) {                  /* dongu yok */
▶          T[t++] = k[i];
▶          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 2` `t = 3` `k[i] = (1,6,14)` `a = 1` `b = 6`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-2	-2	-2	2	-1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 3: (1,2) = 16. `kume_bul` : 1 → kök 1, 2 → kök 2.

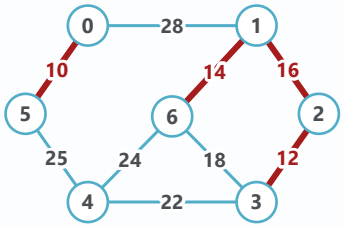
Adım 8 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
▶ for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶     int a = kume_bul(ata, k[i].u);
▶     int b = kume_bul(ata, k[i].v);
6     if (a != b) {                  /* dongu yok */
7         T[t++] = k[i];
8         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 3` `t = 3` `k[i] = (1,2,16)` `a = 1` `b = 2`

Kod İzleme: `kruskal()` ana döngüsü



ata[]

0	1	2	3	4	5	6
-2	-4	1	2	-1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

1 ≠ 2 → döngü yok: T'ye eklenir, kümeler birleşir (boyutu büyük olan kök kalır). t = 4.

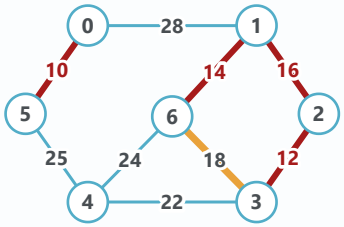
Adım 9 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = kume_bul(ata, k[i].u);
5      int b = kume_bul(ata, k[i].v);
▶      if (a != b) {                  /* dongu yok */
▶          T[t++] = k[i];
▶          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 3` `t = 4` `k[i] = (1,2,16)` `a = 1` `b = 2`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-4	1	2	-1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 4: (3,6) = 18. `kume_bul` : 3 → kök 1, 6 → kök 1.

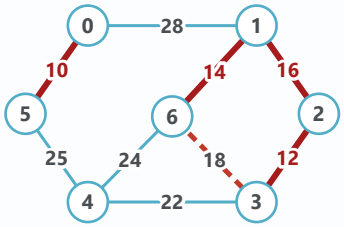
Adım 1 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = kume_bul(ata, k[i].u);
▶      int b = kume_bul(ata, k[i].v);
6      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 4` `t = 4` `k[i] = (3,6,18)` `a = 1` `b = 1`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-4	1	2	-1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

a = b = 1: iki uç aynı kümede → kenar **döngü oluşturur**, bırakılır.

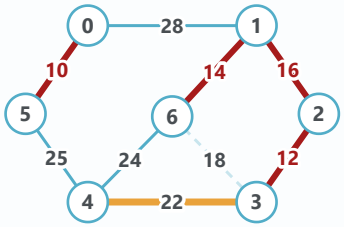
Adım 2 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = kume_bul(ata, k[i].u);
5      int b = kume_bul(ata, k[i].v);
▶      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 4` `t = 4` `k[i] = (3,6,18)` `a = 1` `b = 1`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-4	1	2	-1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
					▲ i			

i = 5: (3,4) = 22. `kume_bul` : 3 → kök 1, 4 → kök 4.

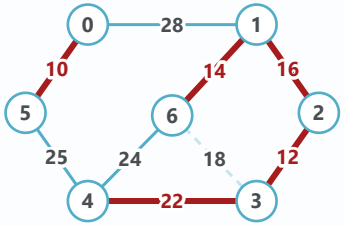
Adım 3 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = kume_bul(ata, k[i].u);
▶      int b = kume_bul(ata, k[i].v);
6      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 5` `t = 4` `k[i] = (3,4,22)` `a = 1` `b = 4`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-5	1	2	1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

1 ≠ 4 → döngü yok: T'ye eklenir, kümeler birleşir (boyutu büyük olan kök kalır). t = 5.

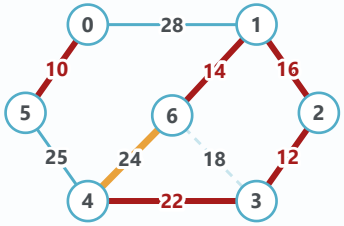
Adım 4 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = kume_bul(ata, k[i].u);
5     int b = kume_bul(ata, k[i].v);
▶     if (a != b) {                  /* dongu yok */
▶         T[t++] = k[i];
▶         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 5` `t = 5` `k[i] = (3,4,22)` `a = 1` `b = 4`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-5	1	2	1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

i = 6: (4,6) = 24. `kume_bul` : 4 → kök 1, 6 → kök 1.

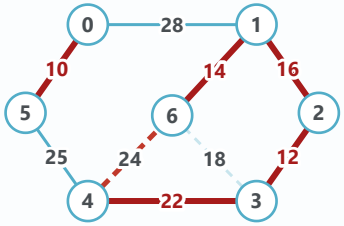
Adım 5 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = kume_bul(ata, k[i].u);
▶      int b = kume_bul(ata, k[i].v);
6      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 6` `t = 5` `k[i] = (4,6,24)` `a = 1` `b = 1`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-5	1	2	1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

a = b = 1: iki uç aynı kümede → kenar **döngü oluşturur**, bırakılır.

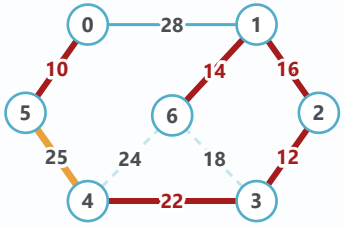
Adım 6 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
3  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = kume_bul(ata, k[i].u);
5      int b = kume_bul(ata, k[i].v);
▶      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 6` `t = 5` `k[i] = (4,6,24)` `a = 1` `b = 1`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
-2	-5	1	2	1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28
							▲ i	

i = 7: (4,5) = 25. `kume_bul` : 4 → kök 1, 5 → kök 0.

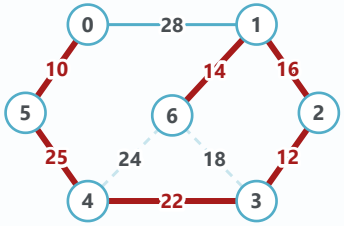
Adım 7 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
▶      int a = kume_bul(ata, k[i].u);
▶      int b = kume_bul(ata, k[i].v);
6      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 7` `t = 5` `k[i] = (4,5,25)` `a = 1` `b = 0`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
1	-7	1	2	1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

▲ i

1 ≠ 0 → döngü yok: T'ye eklenir, kümeler birleşir (boyutu büyük olan kök kalır). t = 6.

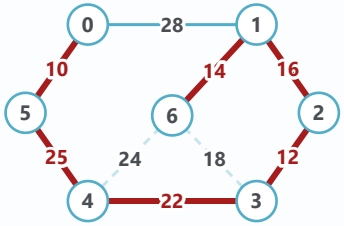
Adım 8 / 9

```
1 for (i = 0; i < n; i++) ata[i] = -1;
2 qsort(k, e, sizeof(kenar), kars);
3 for (i = 0, t = 0; i < e && t < n - 1; i++) {
4     int a = kume_bul(ata, k[i].u);
5     int b = kume_bul(ata, k[i].v);
▶     if (a != b) {                  /* dongu yok */
▶         T[t++] = k[i];
▶         birlestir(ata, a, b);
9     }                              /* yoksa birak */
10 }
11 if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 7` `t = 6` `k[i] = (4,5,25)` `a = 1` `b = 0`

Kod İzleme: `kruskal()` (devam)



ata[]

0	1	2	3	4	5	6
1	-7	1	2	1	0	1

k[] sıralı

0	1	2	3	4	5	6	7	8
0-5 10	2-3 12	1-6 14	1-2 16	3-6 18	3-4 22	4-6 24	4-5 25	0-1 28

t = n − 1 = 6 → döngü biter; (0,1) = 28'e hiç bakılmadı. MST maliyeti **10 + 12 + 14 + 16 + 22 + 25 = 99**.

Adım 9 / 9

```
1  for (i = 0; i < n; i++) ata[i] = -1;
2  qsort(k, e, sizeof(kenar), kars);
▶  for (i = 0, t = 0; i < e && t < n - 1; i++) {
4      int a = kume_bul(ata, k[i].u);
5      int b = kume_bul(ata, k[i].v);
6      if (a != b) {                  /* dongu yok */
7          T[t++] = k[i];
8          birlestir(ata, a, b);
9      }                              /* yoksa birak */
10 }
▶  if (t < n - 1) printf("agac yok\n");
```

DEĞİŞKENLER

`i = 8` `t = 6` `maliyet = 99`

Kruskal'ın Karmaşıklığı (union-find ile)

Adım	Maliyet	Açıklama
Sıralama / heap kurma	O(e log e) / O(e)	qsort ya da aşağıdan yukarıya heap kurma
En az kenarı almak	O(log e) × en çok e kez	min-heap deleteMin (sıralı dizide O(1))
kume_bul (2 kez / kenar)	O(log n) ağırlıklı; ≈ O(α(n)) yol kısaltmalı	en çok 2e çağrı
birlestir	O(1)	tam n − 1 kez
Toplam	**O(e log e) = O(e log n)**	$e \leq n^2 \rightarrow \log e \leq 2 \log n$

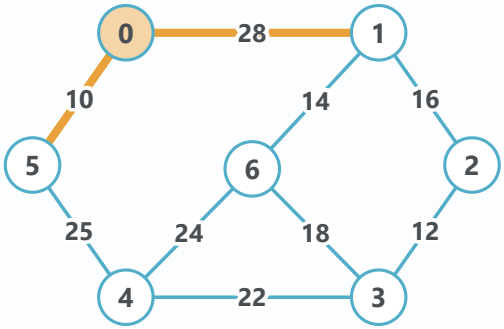
"Kötü" kodlama (listelerde arama + birleştirme) her kenarda O(n) arama yapar: O(e · n). Veri yapısı seçimi aynı algoritmayı bambaşka bir maliyete taşır.

Prim Algoritması

```
T  = {};
TV = {0};            /* ağaç düğümleri: 0'dan başla */
while (|T| < n-1) {
    u ∈ TV ve v ∉ TV olan en ucuz (u,v) kenarını seç;
    if (böyle kenar yok) break;
    TV = TV ∪ {v};
    T  = T  ∪ {(u,v)};
}
if (|T| < n-1)
    printf("minimum kapsayan ağaç yok\n");
```

- Jarník (1930), Prim (1957), Dijkstra (1959) → DJP / Prim-Jarník algoritması.
- Kruskal'da ayrık kenarlar seçilebiliyordu; Prim seçilen kenarları **tek öbek** hâlinde büyütür.
- Seçilebilecek kenarlar: bir ucu TV'de, diğer ucu TV'de **olmayan** kenarlar.
- Tek bir öbek giderek büyür; döngü denetimi gerekmez (v ∉ TV zaten garanti eder).

Prim Adım Adım

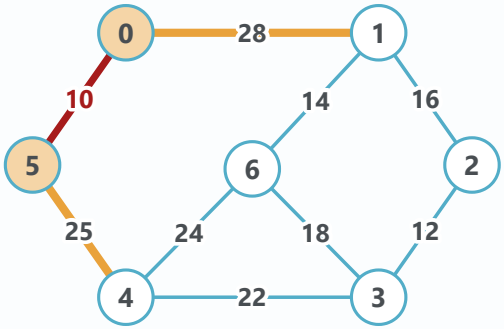


TV	T	Seçilebilecek
{0}	{}	(0,1), (0,5)

Aday kenarlar sarı. En ucuzu **(0,5) = 10** → 5 ağaca katılır.

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

Prim Adım Adım

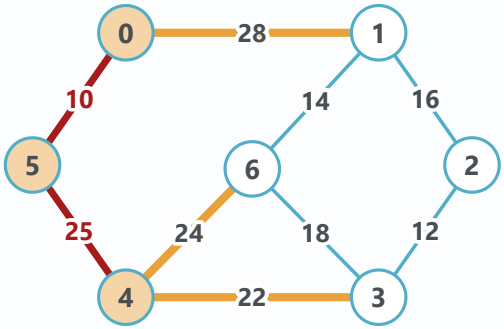


TV	T	Seçilebilecek
{0, 5}	{(0,5)}	(0,1), (5,4)

Aday kenarlar sarı. En ucuzu **(5,4) = 25** → 4 ağaca katılır.

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

Prim Adım Adım

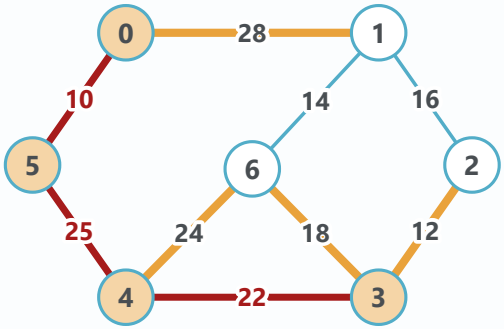


TV	T	Seçilebilecek
{0, 5, 4}	{(0,5), (5,4)}	(0,1), (4,3), (4,6)

Aday kenarlar sarı. En ucuzu **(4,3) = 22** → 3 ağaca katılır.

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

Prim Adım Adım

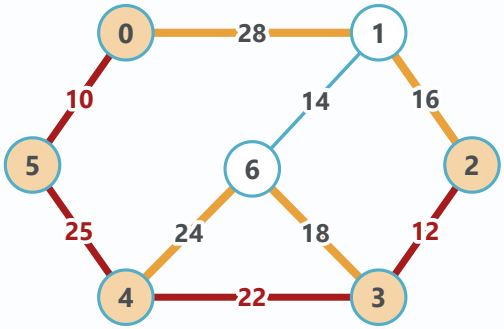


TV	T	Seçilebilecek
{0, 5, 4, 3}	{(0,5), (5,4), (4,3)}	(0,1), (3,2), (3,6), (4,6)

Aday kenarlar sarı. En ucuzu **(3,2) = 12** → 2 ağaca katılır.

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

Prim Adım Adım

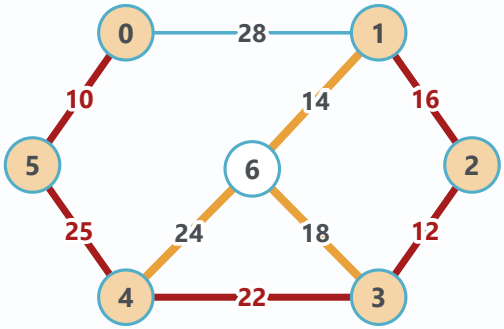


TV	T	Seçilebilecek
{0, 5, 4, 3, 2}	{(0,5), (5,4), (4,3), (3,2)}	(0,1), (2,1), (3,6), (4,6)

Aday kenarlar sarı. En ucuzu **(2,1) = 16** → 1 ağaca katılır.

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

Prim Adım Adım

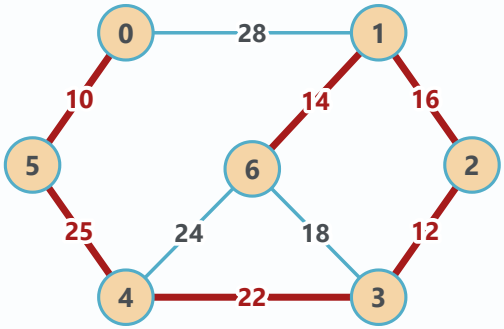


TV	T	Seçilebilecek
{0, 5, 4, 3, 2, 1}	{(0,5), (5,4), (4,3), (3,2), (2,1)}	(1,6), (3,6), (4,6)

Aday kenarlar sarı. En ucuzu **(1,6) = 14** → 6 ağaca katılır.

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

Prim Adım Adım



TV	T	Seçilebilecek
{0, 5, 4, 3, 2, 1, 6}	{(0,5), (5,4), (4,3), (3,2), (2,1), (1,6)}	—

TV tüm düğümleri içeriyor: 6 kenar, toplam **99** — Kruskal ile aynı ağaç (ağırlıklar farklı olduğundan MST tektir).

Adım 7 / 7

Son adımlarda (0,1) kenarı “seçilebilecek” gibi görünse de iki ucu da TV’de olduğundan aday değildir.

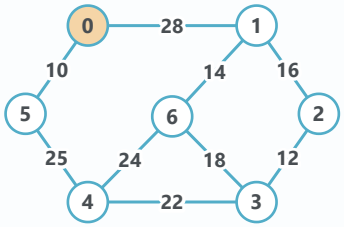
Prim'i Kodlamak: Asıl Zorluk

- En az maliyetli aday kenar **tek hamlede** alınmalı.
- Bir kenar seçilince aday kümesine yeni kenarlar **eklenir**, bazıları **çıkar** (iki ucu da TV'ye düşenler): A kümesi B olur.
- 0 seçildiğinde yalnızca (0,1) ve (0,5)'i gören bir veri yapısı olmalı; diğerlerine bakmamalı bile.
- Çözüm (ek): her $v \notin TV$ için `enaz[v]` = TV'ye en ucuz bağlantı. Dizi ile $O(n^2)$; min-heap ile $O(e \log n)$.

> **Hocadan:** "Bunun çözümü bulunabiliyorsa veri yapıları insanı bir noktaya taşımıştır... "Zaten Prim yazmış, ben de kodlarım." denirse karmaşıklık birden büyür."

> **Hocadan:** "Hayatında 'algoritma' kelimesini duymamış çizge kuramcısı birine bu problem verilirse o böyle çözer. Bilgisayar mühendisi ise arka plandaki veri yapısının ne olması gerektiğini keşfeder."

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	0	0	0	0
enaz	0	28	∞	∞	∞	10	∞
yakin	0	0	0	0	0	0	0

TV = {0}. `enaz[v]` = w[0][v]: 0'a bağlı 1 (28) ve 5 (10); diğerleri ∞. `yakin[v]` = 0.

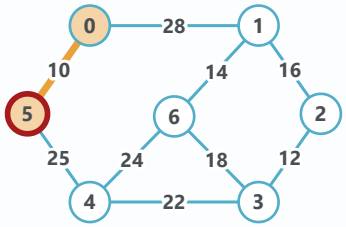
Adım 1 / 13

```
▶ secildi[0] = 1;                    /* TV = {0} */
  for (v = 0; v < n; v++)
▶     enaz[v] = w[0][v], yakin[v] = 0;
  for (k = 1; k < n; k++) {
      u = en_yakin(enaz, secildi, n);  /* O(n) */
      if (u < 0) break;                /* agac yok */
      secildi[u] = 1;  toplam += enaz[u];
      for (v = 0; v < n; v++)
          if (!secildi[v] && w[u][v] < enaz[v])
              enaz[v] = w[u][v], yakin[v] = u;
  }
```

DEĞİŞKENLER

`k = –` `u = –` `toplam = 0`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	0	0	1	0
enaz	0	28	∞	∞	∞	10	∞
yakin	0	0	0	0	0	0	0

En küçük `enaz` : **u = 5** (10), kenar (0,5) ağaca girer. toplam = 10.

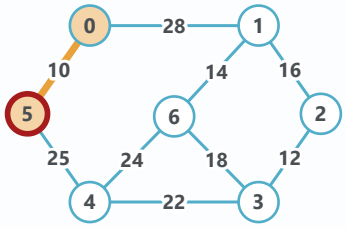
Adım 2 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
▶      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 1` `u = 5` `toplam = 10`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	0	0	1	0
enaz	0	28	∞	∞	25	10	∞
yakin	0	0	0	0	5	0	0

5 üzerinden daha ucuz bağlantılar: enaz[4] = 25 (yakin = 5).

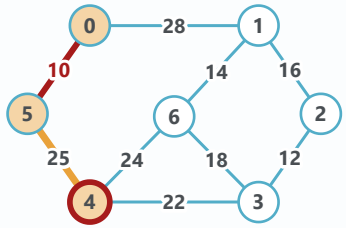
Adım 3 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
7      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
▶          if (!secildi[v] && w[u][v] < enaz[v])
▶              enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 1` `u = 5` `toplam = 10`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	0	1	1	0
enaz	0	28	∞	∞	25	10	∞
yakin	0	0	0	0	5	0	0

En küçük `enaz` : **u = 4** (25), kenar (5,4) ağaca girer. toplam = 35.

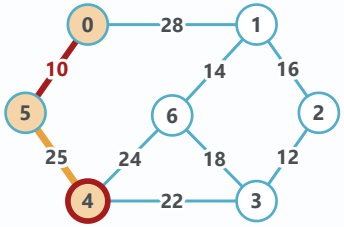
Adım 4 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
▶      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 2` `u = 4` `toplam = 35`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	0	1	1	0
enaz	0	28	∞	22	25	10	24
yakin	0	0	0	4	5	0	4

4 üzerinden daha ucuz bağlantılar: enaz[3] = 22, enaz[6] = 24 (yakin = 4).

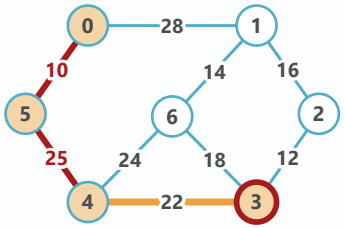
Adım 5 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
7      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
▶          if (!secildi[v] && w[u][v] < enaz[v])
▶              enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 2` `u = 4` `toplam = 35`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	1	1	1	0
enaz	0	28	∞	22	25	10	24
yakin	0	0	0	4	5	0	4

En küçük `enaz` : **u = 3** (22), kenar (4,3) ağaca girer. toplam = 57.

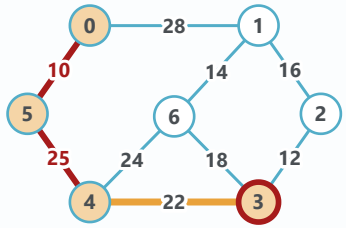
Adım 6 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
▶      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 3` `u = 3` `toplam = 57`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	0	1	1	1	0
enaz	0	28	12	22	25	10	18
yakin	0	0	3	4	5	0	3

3 üzerinden daha ucuz bağlantılar: enaz[2] = 12, enaz[6] = 18 (yakin = 3).

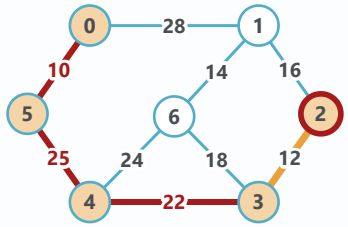
Adım 7 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
7      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
▶          if (!secildi[v] && w[u][v] < enaz[v])
▶              enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 3` `u = 3` `toplam = 57`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	1	1	1	1	0
enaz	0	28	12	22	25	10	18
yakin	0	0	3	4	5	0	3

En küçük `enaz` : **u = 2** (12), kenar (3,2) ağaca girer. toplam = 69.

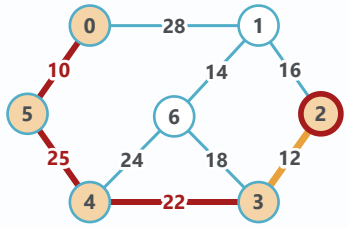
Adım 8 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
▶      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 4` `u = 2` `toplam = 69`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	0	1	1	1	1	0
enaz	0	16	12	22	25	10	18
yakin	0	2	3	4	5	0	3

2 üzerinden daha ucuz bağlantılar: enaz[1] = 16 (yakin = 2).

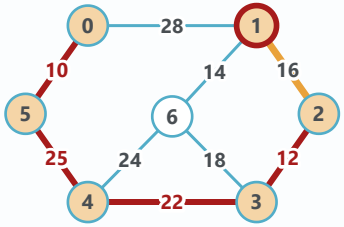
Adım 9 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
5      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
7      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
▶          if (!secildi[v] && w[u][v] < enaz[v])
▶              enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 4` `u = 2` `toplam = 69`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	1	1	1	1	1	0
enaz	0	16	12	22	25	10	18
yakin	0	2	3	4	5	0	3

En küçük `enaz` : **u = 1** (16), kenar (2,1) ağaca girer. toplam = 85.

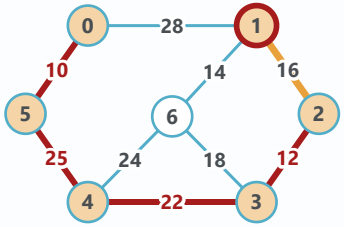
Adım 10 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
▶      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 5` `u = 1` `toplam = 85`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	1	1	1	1	1	0
enaz	0	16	12	22	25	10	14
yakin	0	2	3	4	5	0	1

1 üzerinden daha ucuz bağlantılar: enaz[6] = 14 (yakin = 1).

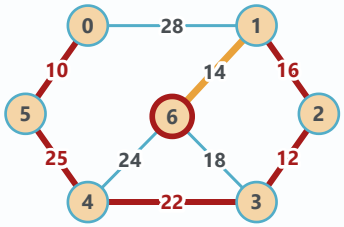
Adım 11 / 13

```
 1 secildi[0] = 1;                    /* TV = {0} */
 2 for (v = 0; v < n; v++)
 3     enaz[v] = w[0][v], yakin[v] = 0;
 4 for (k = 1; k < n; k++) {
 5     u = en_yakin(enaz, secildi, n);  /* O(n) */
 6     if (u < 0) break;                /* agac yok */
 7     secildi[u] = 1;  toplam += enaz[u];
 8     for (v = 0; v < n; v++)
▶          if (!secildi[v] && w[u][v] < enaz[v])
▶              enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 5` `u = 1` `toplam = 85`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	1	1	1	1	1	1
enaz	0	16	12	22	25	10	14
yakin	0	2	3	4	5	0	1

En küçük `enaz` : **u = 6** (14), kenar (1,6) ağaca girer. toplam = 99.

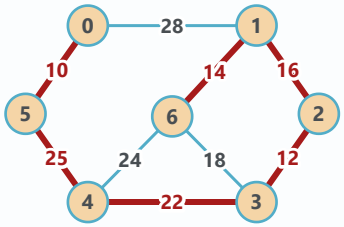
Adım 12 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
4  for (k = 1; k < n; k++) {
▶      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
▶      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 6` `u = 6` `toplam = 99`

Kod İzleme: `prim()` (enaz/yakin dizileri)



	0	1	2	3	4	5	6
secildi	1	1	1	1	1	1	1
enaz	0	16	12	22	25	10	14
yakin	0	2	3	4	5	0	1

n − 1 düğüm eklendi: toplam **99**. Her tur O(n) arama + O(n) güncelleme → **O(n²)**; yoğun çizgede idealdir.

Adım 13 / 13

```
1  secildi[0] = 1;                    /* TV = {0} */
2  for (v = 0; v < n; v++)
3      enaz[v] = w[0][v], yakin[v] = 0;
▶  for (k = 1; k < n; k++) {
5      u = en_yakin(enaz, secildi, n);  /* O(n) */
6      if (u < 0) break;                /* agac yok */
7      secildi[u] = 1;  toplam += enaz[u];
8      for (v = 0; v < n; v++)
9          if (!secildi[v] && w[u][v] < enaz[v])
10             enaz[v] = w[u][v], yakin[v] = u;
11 }
```

DEĞİŞKENLER

`k = 7` `u = −` `toplam = 99`

Prim'i Heap ile Hızlandırmak (ek konu)

```
/* komsuluk listesi + min-heap (anahtar = enaz) */
heap_ekle(H, 0, 0);
while (!heap_bos(H)) {
    u = heap_min_al(H);          /* O(log n) */
    if (secildi[u]) continue;    /* bayat kayit */
    secildi[u] = 1;
    for (p = baslik[u]; p; p = p->bag)
        if (!secildi[p->dugum] &&
            p->agirlik < enaz[p->dugum]) {
            enaz[p->dugum] = p->agirlik;
            yakin[p->dugum] = u;
            heap_ekle(H, p->dugum, p->agirlik);
        }
}
```

Uygulama	Zaman	Uygun
Dizi (yukarıdaki)	$O(n^2)$	yoğun çizge ($e \approx n^2$)
İkili heap + liste	$O(e \log n)$	seyrek çizge
Fibonacci heap	$O(e + n \log n)$	kuramsal
Kruskal	$O(e \log e)$	seyrek, kenar listesi hazır

Araştırma sorusuna cevap: yoğun çizgede dizili Prim ($O(n^2)$), seyrek çizgede Kruskal ya da heap'li Prim daha etkilidir.

PRİM

Problem Çözmede Yaklaşım

> **Hocadan:** “Bu problem nasıl çözülmüş? Literatürde hangi veri yapıları önerilmiş? Dünya bunu nasıl çözüyor, ben nasıl çözeyim?”

> **Hocadan:** ““Algoritmaya çok iyi hâkimiz.” Değil! Hâlâ üzerinde çalışıyoruz. Problem çözülürken neyin nerede kullanılacağı: işte veri yapıları burada devreye giriyor.”

“*To follow the path: look to the master, follow the master, walk with the master, see through the master, become the master.*”

Yani önce ustanın (literatürün) çözümü öğrenilir, sonra özgün çözüm üretilir.

PRİM

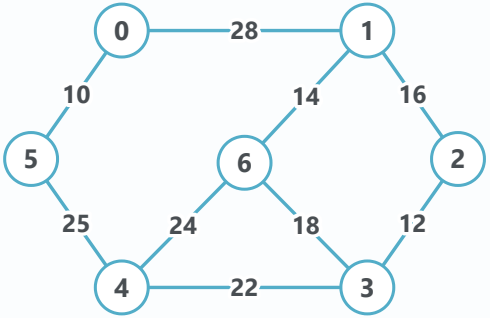
Ödev: Kruskal'ı Veri Yapısıyla Kodlayın

"Haftaya herkes Kruskal Algoritması'nı anlattığım şekilde kodlayıp getiriyor." Beklenen: **min-heap** + **union-find**, girdi kenar listesi, çıktı ağaç kenarları ve toplam maliyet.

- Kenar yapısı `kenar {u, v, w}`; heap'i diziden aşağıdan yukarıya kurun (O(e)).
- `kume_bul`'a yol kısaltma ekleyin; `birlestir` ağırlıklı olsun.
- Çizge bağlantılı değilse "minimum kapsayan ağaç yok" yazsın (|T| < n − 1).
- 7 köy örneğinde 99 bulduğunu, (3,6) ve (4,6)'nın reddedildiğini gösteren bir çıktı üretin.

▶ **Kontrol listesi**

Diğer MST Algoritmaları: Sollin (Borůvka), Reverse-Delete

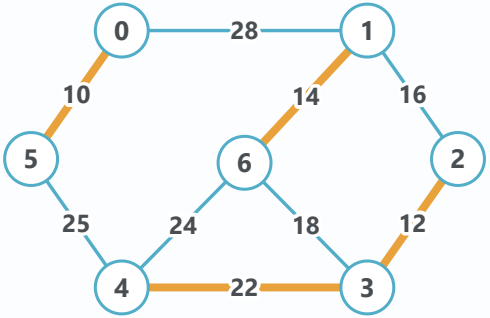


Başta her düğüm ayrı bir bileşen.

Adım 1 / 3

- **Sollin / Borůvka (1926):** her turda *her bileşen* en ucuz dış kenarını seçer; bileşen sayısı en az yarıya iner → en çok log n tur, **O(e log n)**. Paralelleştirmeye uygundur.
- **Reverse-delete:** kenarlar azalan maliyetle denenir; kenarı silmek çizgeyi koparmıyorsa kenar silinir. Kruskal'ın tersi.
- Hepsi aynı kesit/döngü özelliklerine dayanır ve (farklı ağırlıklarda) **aynı** ağacı bulur.

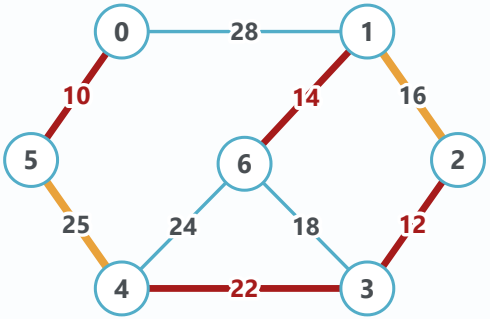
Diğer MST Algoritmaları: Sollin (Borůvka), Reverse-Delete



Tur 1: her bileşen kendisinden çıkan **en ucuz** kenarı seçer (sarı): (0,5)=10, (1,6)=14, (2,3)=12, (3,4)=22. Kalan bileşen: 3.

- **Sollin / Borůvka (1926):** her turda *her bileşen* en ucuz dış kenarını seçer; bileşen sayısı en az yarıya iner → en çok log n tur, **O(e log n)**. Paralelleştirmeye uygundur.
- **Reverse-delete:** kenarlar azalan maliyetle denenir; kenarı silmek çizgeyi koparmıyorsa kenar silinir. Kruskal'ın tersi.
- Hepsi aynı kesit/döngü özelliklerine dayanır ve (farklı ağırlıklarda) **aynı** ağacı bulur.

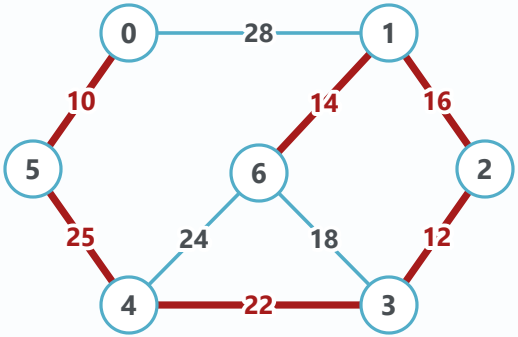
Diğer MST Algoritmaları: Sollin (Borůvka), Reverse-Delete



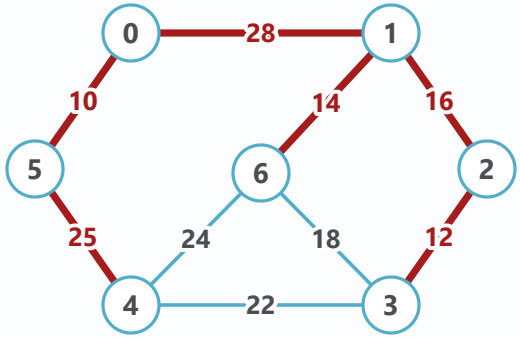
Tur 2: her bileşen kendisinden çıkan **en ucuz** kenarı seçer (sarı): (4,5)=25, (1,2)=16. Kalan bileşen: 1. Toplam **99**.

- **Sollin / Borůvka (1926):** her turda *her bileşen* en ucuz dış kenarını seçer; bileşen sayısı en az yarıya iner → en çok log n tur, **O(e log n)**. Paralelleştirmeye uygundur.
- **Reverse-delete:** kenarlar azalan maliyetle denenir; kenarı silmek çizgeyi koparmıyorsa kenar silinir. Kruskal'ın tersi.
- Hepsi aynı kesit/döngü özelliklerine dayanır ve (farklı ağırlıklarda) **aynı** ağacı bulur.

MST ile En Kısa Yol Ağacı Aynı Şey Değildir



MST (toplam 99)



0'dan en kısa yol ağacı (toplam 105)

MST

Toplam kenar maliyetini en aza indirir; kökü yoktur. 0'dan 1'e ağaç üzerinden uzaklık: 85.

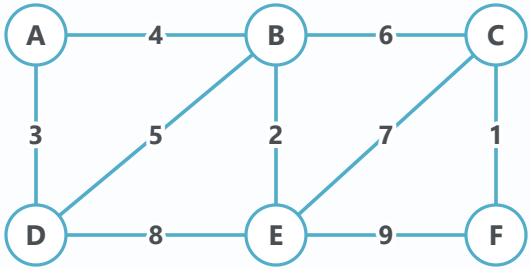
En kısa yol ağacı

Her düğümün 0'a **uzaklığını** en aza indirir: d(1) = 28, ama toplam kenar maliyeti 105 > 99.

Kruskal ile Prim'in Karşılaştırması

	Kruskal	**Prim**
Büyüyen yapı	orman (ayrık öbekler birleşir)	tek ağaç (TV)
Seçim ölçütü	tüm kenarlar arasında en ucuzu	TV'den çıkan en ucuz kenar
Döngü denetimi	union-find gerekir	gerekmez ($v \notin TV$)
Veri yapısı	sıralı dizi / min-heap + kümeler	enaz[] dizisi / min-heap
Zaman	O(e log e)	$O(n^2)$ ya da O(e log n)
İyi olduğu yer	seyrek çizge, kenar listesi	yoğun çizge, matris
Bağlantılı olmayan çizge	minimum kapsayan **orman** verir	yalnızca 0'ın bileşenini kapsar

Sınav Tarzı Soru: Kruskal ve Prim'i İşletin



Kruskal'ın kodlanması ödev olarak verildi; aşağıdaki soru sınavlarda sorulabilecek tarzdadır.

1. Kruskal ile MST'yi bulun; her kenar için "ekle / döngü" kararını yazın.
2. Prim'i A'dan başlatıp seçilen kenar sırasını yazın.
3. Union-find `ata[]` dizisinin son hâlini verin (ağırlıklı birleşim).

▶ **Çözüm**

Laboratuvar: Kruskal ve Prim

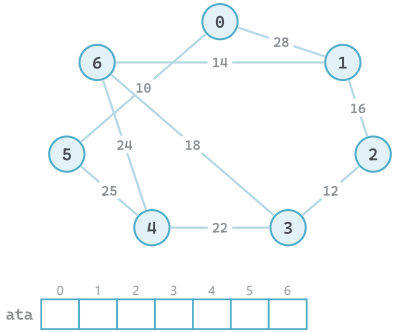
Kruskal: 7 düğüm, 9 kenar

Girdi (ağırlıklı kenarlar, ör. 0-1:28) `0-1:28, 0-5:10, 1-2:16, 1-6:14, 2-3:12, 3-4:22, 3-6:18, 4-5`

Uygula

KOD: kruskal

```
 1 for (i = 0; i < n; i++) ata[i] = -1;
 2 qsort(k, e, sizeof(kenar), kars);
 3 for (i = 0, t = 0;
 4      i < e && t < n - 1; i++) {
 5     int a = kume_bul(ata, k[i].u);
 6     int b = kume_bul(ata, k[i].v);
 7     if (a != b) { /* dongu yok */
 8         T[t++] = k[i];
 9         birlestir(ata, a, b);
10     } /* yoksa birak */
11 }
12 if (t < n - 1) printf("agac yok\n");
```



toplam 0

ÇIKTI (henüz yok)

Kruskal: kenarlar küçükten büyüğe denenir; uçları farklı kümedeyse ağaca alınır, aynı kümedeyse döngü oluşturacağı için bırakılır. Kümeler union-find ile tutulur.

ÖZET

Karmaşıklık Özeti

Algoritma	Yapı	Zaman	Not
Tüm alt kümeleri dene	—	$C(e, n-1) \times O(n)$	üstel, pratik değil
Kruskal (kötü kodlama)	listeler + arama	$O(e \cdot n)$	“böyle kodlamayın”
Kruskal	sıralama/min-heap + union-find	$O(e \log e)$	orman büyür, birleşir
Prim	diziler (enaz, yakin)	$O(n^2)$	yoğun çizge
Prim	ikili heap + komşuluk listesi	$O(e \log n)$	seyrek çizge
DFS/BFS kapsayan ağacı	yığın / kuyruk	$O(n + e)$	ağırlıksız

ÖZET

Sık Yapılan Hatalar

1. Kruskal'da döngüyü "iki uç da daha önce seçilmiş mi?" diye denetlemek (yanlış: farklı öbeklerde olabilirler).
2. Prim'de aday kümesinden iki ucu da TV'de olan kenarları çıkarmamak.
3. MST ile en kısa yol ağacını karıştırmak: MST toplam maliyeti, en kısa yol tek tek uzaklıkları küçültür.
4. Bağlantılı olmayan çizgede $|T| < n - 1$ kontrolünü unutmak.
5. Union-find'da kök olmayan düğümleri birleştirmek (önce `kume_bul`!).
6. ∞ için INT_MAX kullanıp toplamda taşma yaşamak.

Alıştırmalar

1. 7 köy örneğinde (3,6) kenarının maliyeti 18 yerine 11 olsaydı MST ve maliyeti ne olurdu?
2. Kruskal ve Prim aynı çizgede hangi koşulda farklı ağaçlar verebilir?
3. 8 düğümlü örnek çizgede 7'den başlayan DFS ve BFS kapsayan ağaçlarının kenarlarını yazın.
4. K_4'ün kaç kapsayan ağacı vardır? Kaçı yol, kaçı yıldız biçimlidir?
5. Kenar ağırlıkları 1..10 arası tam sayıysa Kruskal'ın sıralama adımı nasıl O(e)'ye iner?
6. Prim'in $O(n^2)$ sürümü neden seyrek çizgede kötüdür?

▶ **Cevaplar / ipuçları**

ÖZET

Bölüm Özeti

Kapsayan ağaç

n − 1 kenarlı, döngüsüz, tüm düğümleri bağlayan alt çizge; DFS/BFS birer tane üretir.

Minimum maliyet

Toplam ağırlığı en küçük kapsayan ağaç; açgözlü seçim doğru sonuç verir (kesit özelliği).

Kruskal

Kenarlar artan maliyetle denenir; döngü union-find ile yakalanır: O(e log e).

Prim

Tek öbek en ucuz kenarla büyür: diziyle $O(n^2)$, heap ile O(e log n).

Veri yapısı

Aynı fikir, farklı yapı: liste araması O(e·n) → kümeler O(e log e).

Seçim

Seyrek çizgede Kruskal ya da heap'li Prim; yoğun çizgede dizili Prim.

ÖZET

Kaynaklar

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2. baskı, §6.3 Minimum Cost Spanning Trees.
- T. H. Cormen ve ark., *Introduction to Algorithms*, 4. baskı, Bölüm 19 (Disjoint Sets) ve Bölüm 21 (Minimum Spanning Trees).
- R. Sedgewick, *Algorithms in C, Part 5*, Bölüm 20 (Minimum Spanning Trees).

← Bölüm 16: Kümeler ve Union-Find

Ders sayfası

Bölüm 18: En Kısa Yol ve Aktivite Ağları

İlgili: Çizgeler · Kümeler ve Union-Find

Son

Bölüm 17: Minimum Kapsayan Ağaçlar

Sonraki bölüm: Bölüm 18: En Kısa Yol ve Aktivite Ağları · Ders sayfası