Bölüm 4.3: Minimum Kapsayan Ağaç Algoritmaları

Algoritmalar

Kapsayan ağaç | Kesim ve döngü özelliği | Kruskal | Union-Find | Prim | Borůvka | Ters silme | Karşılaştırma

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavram

Kapsayan ağacı ve minimum kapsayan ağacı (MST) tanımlamak; V − 1 kenar, kesim ve döngü özelliklerini açıklamak.

Kruskal ve Union-Find

Kenarları sıralayıp döngü oluşturmayanları seçmek; döngü denetimini ayrık kümelerle (bul / birleştir) yapmak.

Prim

Ağacı bir düğümden büyütmek; anahtar ve ebeveyn dizilerini öncelikli kuyrukla adım adım izlemek.

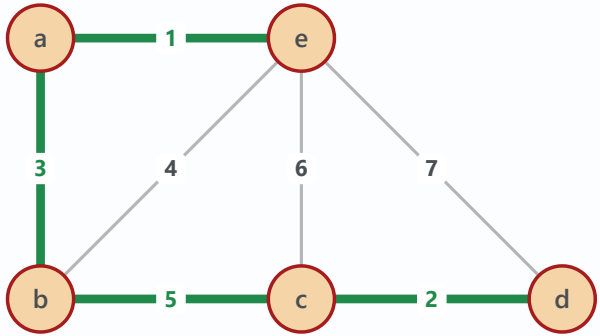
Borůvka, Ters Silme, Analiz

Bileşen tabanlı ve silme tabanlı yaklaşımları uygulamak; dört algoritmanın karmaşıklığını karşılaştırıp uygun olanı seçmek.

MST KAVRAMI

Kapsayan Ağaç ve Minimum Kapsayan Ağaç

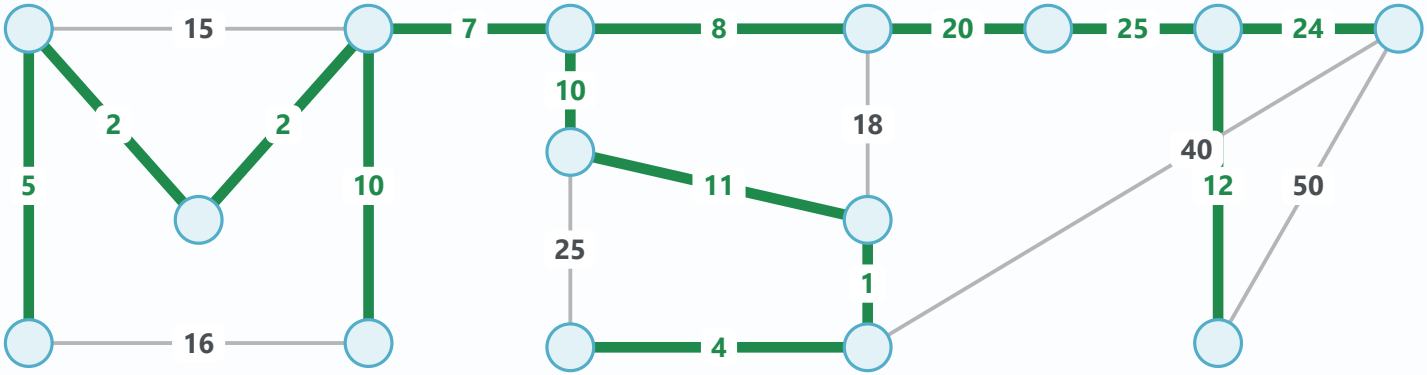
- G = (V, E): **bağlı**, **yönsüz** ve kenarları **ağırlıklı** bir çizge.
- **Kapsayan ağaç** (spanning tree): tüm düğümleri içeren, **bağlı** ve **döngüsüz** alt çizge. Her zaman tam **V − 1** kenarı vardır.
- **Minimum kapsayan ağaç** (Minimum Spanning Tree, MST): kenar ağırlıkları toplamı **en küçük** olan kapsayan ağaç.
- Ağaca bir kenar eklemek döngü oluşturur; bir kenar silmek ağacı ikiye böler.



Bir çizgenin çok sayıda kapsayan ağacı olabilir (n düğümlü tam çizgede n^{n-2} tane, Cayley formülü). Hepsini denemek yerine **açgözlü** (greedy) algoritmalar MST'yi doğrudan bulur. Yandaki çizgede MST ağırlığı w(T) = 1 + 2 + 3 + 5 = **11**.

MST KAVRAMI

Örnek: "MST" Çizgesi



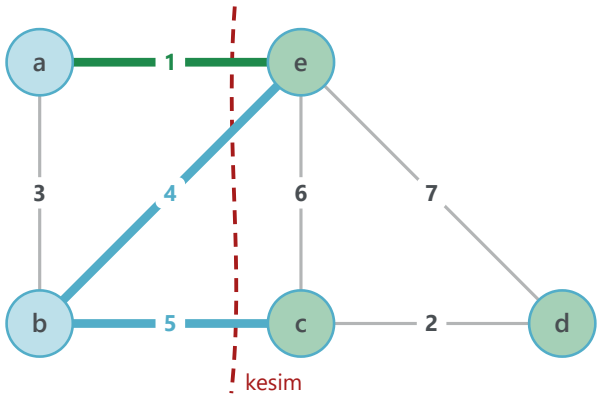
- 15 düğüm, 20 kenar. **Kalın** 14 kenar (= V − 1) minimum kapsayan ağacı oluşturur: toplam ağırlık **141**.
- İnce kenarlar dışarıda kalır: her biri, kalın kenarlarla bir **döngü** kapatır ve o döngünün **en ağır** kenarıdır (ör. 15, 16, 18, 25, 40, 50).
- Ağırlığı yüksek (20, 24, 25) kenarlar da seçilebilir: hiçbir döngünün **en ağırı** değildirler. Örneğin 40'ın kapattığı döngüde üçü de vardır; o döngünün en ağırı 40'tır.

MST KAVRAMI

Kesim Özelliği ve Döngü Özelliği

Kesim özelliği (cut property)

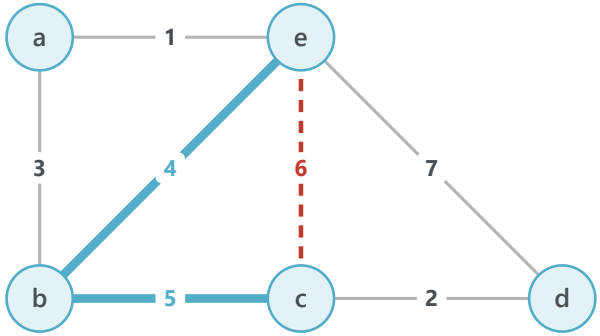
Düğümleri iki kümeye ayıran her **kesimde**, kesimi geçen **en hafif** kenar bir MST'ye aittir.



{a, b} | {c, d, e}: geçen kenarlar ae = 1, be = 4, bc = 5 → **ae** güvenli.

Döngü özelliği (cycle property)

Herhangi bir **döngünün en ağır** kenarı (tekse) hiçbir MST'de yer almaz.



b–c–e döngüsü: 4, 5, 6 → **ec = 6** asla seçilmez.

Kruskal ve Prim kesim özelliğini, ters silme döngü özelliğini kullanır. Tüm ağırlıklar farklıysa MST **tektir**.

MST KAVRAMI

Açgözlü MST Algoritmaları

Dört algoritmanın hepsi **açgözlüdür**: her adımda yerel olarak en iyi kenarı seçer (ya da en kötüsünü atar) ve bu seçim hiçbir zaman geri alınmaz.

Algoritma	Fikir	Yardımcı yapı	Zaman
Kruskal (1956)	Kenarları küçükten büyüğe sırala, döngü oluşturmayanları ekle	union-find	O(E log E)
Prim (1957)	Bir düğümden başla, ağacı en ucuz komşu kenarla büyüt	öncelikli kuyruk	O((V + E) log V)
Borůvka (1926)	Her bileşen kendi en ucuz kenarını aynı anda ekler	union-find	O(E log V)
Ters silme	Kenarları büyükten küçüğe dene, bağlılığı bozmayanı sil	BFS/DFS ya da union-find	O(E · (V + E))

Hepsi aynı ağırlıkta bir MST bulur; ağırlıklar farklıysa **aynı ağacı** bulurlar. Fark, kenarları hangi sırayla ele aldıkları ve gereken veri yapısındadır.

MST KAVRAMI

Nerede Kullanılır?

Ağ tasarımı
Şehirleri en az kabloyla bağlamak: elektrik, su, fiber, yol ağları. Borůvka'nın çıkış noktası Moravya'nın elektrik ağıdır.

Kümeleme
MST'den en ağır k − 1 kenar silinirse k küme kalır (tek bağlantılı kümeleme, single-linkage).

Yaklaşık algoritmalar
Metrik gezgin satıcı probleminde (TSP) MST tabanlı tur, en iyinin en fazla 2 katıdır.

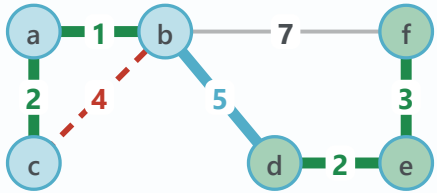
Görüntü işleme
Pikselleri düğüm, renk farkını ağırlık sayarak bölütleme (segmentation).

Devre tasarımı
Bacakları en kısa toplam iletkenle bağlamak (VLSI yönlendirme ön adımı).

Diğer çizge problemleri
Darboğaz yol (en büyük kenarı en küçük yol) MST üzerindedir; Steiner ağacı için temel yaklaşım.

Kruskal Algoritması

- Çizgedeki tüm düğümleri birbirine bağlayan, toplam ağırlığı en küçük ağacı oluşturur; **açgözlü** (greedy) bir yaklaşımdır.
- Joseph **Kruskal** tarafından geliştirilmiştir (1956).
- Başlangıçta her düğüm **ayrı bir ağaçtır** (orman); adım adım bu ağaçlar **birleştirilir**.
- Kenarlar **küçükten büyüğe** sıralanır; döngü oluşturmayan kenar seçilir, döngü oluşturan atlanır.



yeşil: seçilen · kırmızı: döngü (atla)
mavi: sıradaki, iki ağacı birleştirir

Güçlü yönler

+ **Seyrek** çizgelerde (E ≈ V) çok uygun.
+ Basit: kenar listesi + union-find yeterli.
+ Bağlı olmayan çizgede minimum kapsayan **orman** verir.

Zayıf yönler

– **Kenar sıralama** maliyeti: O(E log E).
– Yoğun çizgede ($E \approx V^2$) Prim daha hızlı olabilir.
– Döngü denetimi için ek yapı (union-find) gerekir.

Kruskal: Adımlar ve Sözde Kod

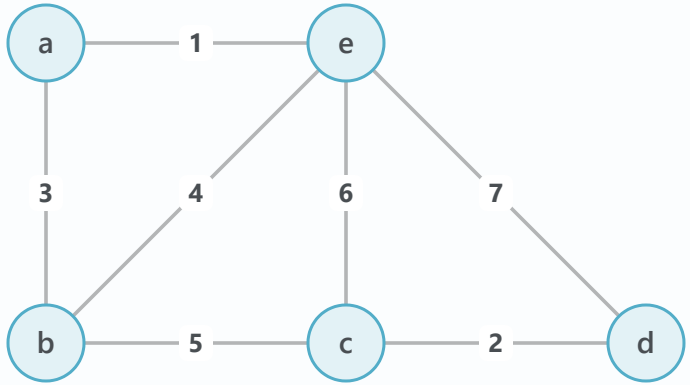
1. Çizgedeki tüm kenarlar ağırlıklarına göre **sıralanır**.
2. Sıralı kenarlar arasından sıradaki **en küçük** ağırlıklı kenar seçilir.
3. Kenar **farklı ağaçlara** ait iki düğümü birleştiriyorsa (döngü oluşturmuyorsa) ağaca eklenir ve iki ağaç birleştirilir.
4. Aynı ağacın iki düğümünü bağlıyorsa döngü oluşturur: **atlanır**.
5. V − 1 kenar seçilene (ya da kenarlar bitene) kadar 2–4 tekrarlanır.

```
KRUSKAL(G):
  MST = [ ]
  ayrikKume.oluştur(V)
  kenarlar.sırala(artana göre)
  her bir (u, v, ağırlık) için kenarlar içinde:
    eğer ayrikKume.bul(u) ≠ ayrikKume.bul(v):
      MST.ekle((u, v, ağırlık))
      ayrikKume.birleştir(u, v)
  döndür MST
```

"Aynı ağaçta mı?" sorusu **union-find** (ayrık kümeler) yapısıyla neredeyse sabit zamanda yanıtlanır.

Kruskal Adım Adım

ab	ae	bc	be	cd	ed	ec
3	1	5	4	2	7	6

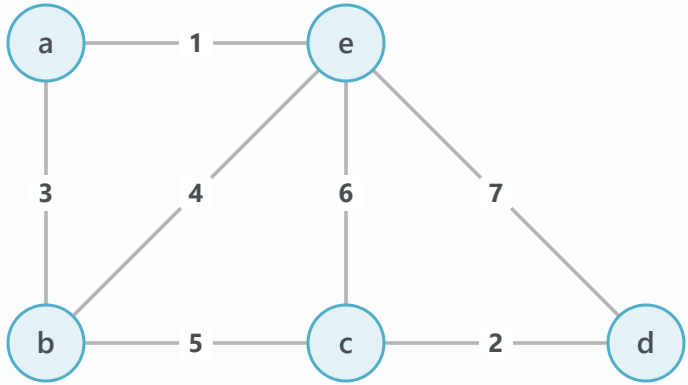


MST	—
w(T)	0
Ağaçlar	{a} {b} {c} {d} {e}

Başlangıç: her düğüm **ayrı bir ağaçtır** (5 ağaçlı orman). Kenar tablosu henüz sırasız.

Kruskal Adım Adım

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

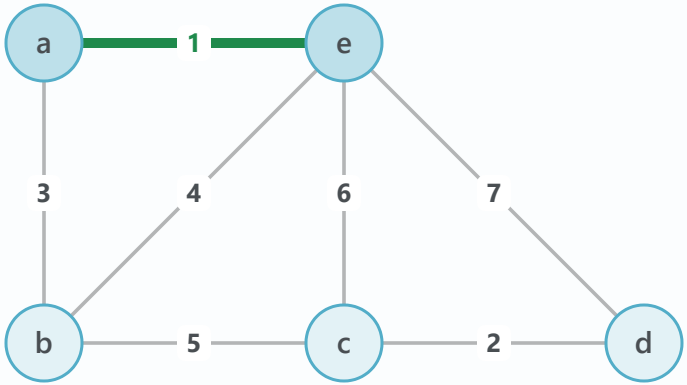


MST	—
w(T)	0
Ağaçlar	{a} {b} {c} {d} {e}

Adım 1: kenarlar **artan ağırlığa** göre sıralanır: ae, cd, ab, be, bc, ec, ed.

Kruskal Adım Adım

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

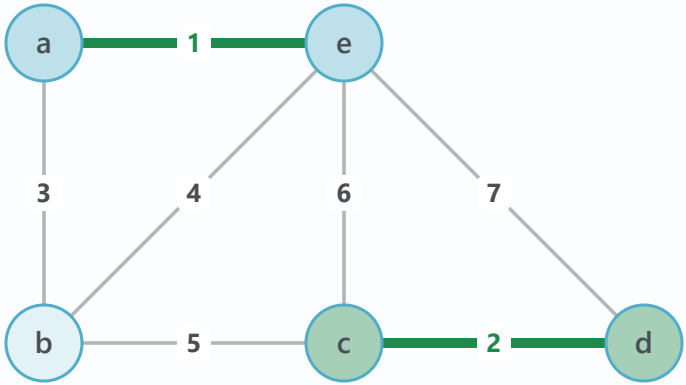


MST	ae
w(T)	1
Ağaçlar	{a, e} {b} {c} {d}

ae = 1: a ve e farklı ağaçlarda → kenar eklenir, iki ağaç birleşir.

Kruskal Adım Adım

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

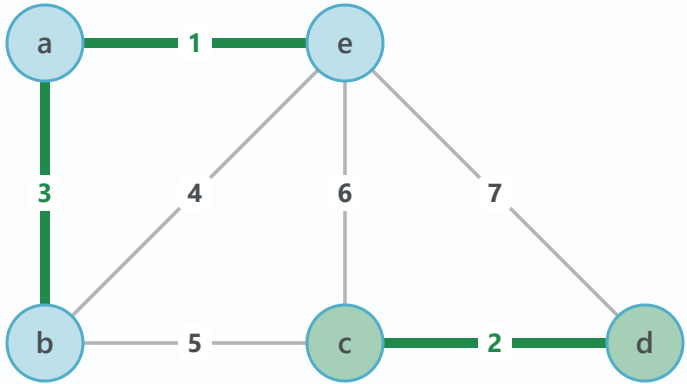


MST	ae, cd
w(T)	3
Ağaçlar	{a, e} {c, d} {b}

cd = 2: c ve d farklı ağaçlarda → kenar eklenir, iki ağaç birleşir.

Kruskal Adım Adım

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

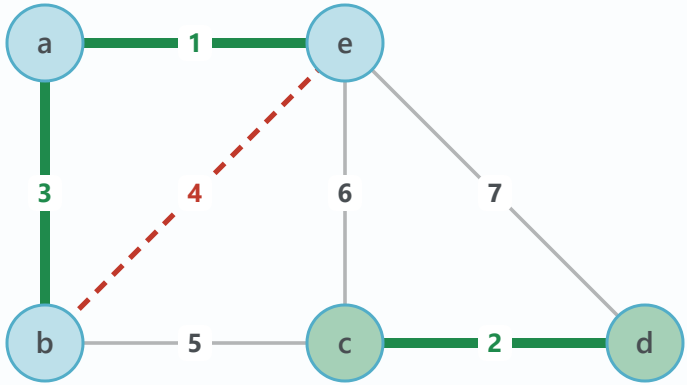


MST	ae, cd, ab
w(T)	6
Ağaçlar	{a, b, e} {c, d}

ab = 3: a ve b farklı ağaçlarda → kenar eklenir, iki ağaç birleşir.

Kruskal Adım Adım

ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

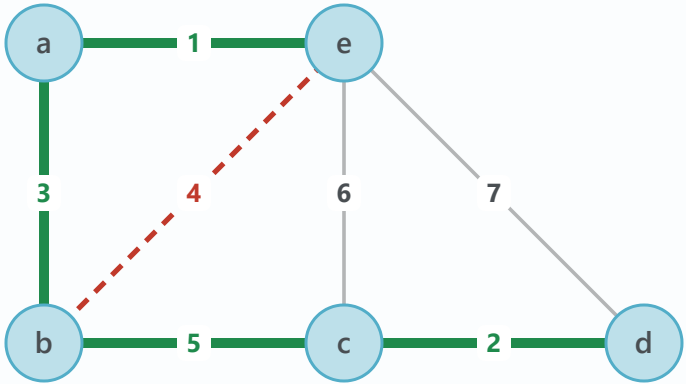


MST	ae, cd, ab
w(T)	6
Ağaçlar	{a, b, e} {c, d}

be = 4: b ve e zaten **aynı ağaçta** → eklenirse döngü oluşur, atlanır.

Kruskal Adım Adım

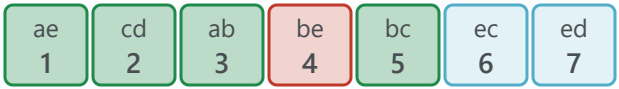
ae	cd	ab	be	bc	ec	ed
1	2	3	4	5	6	7

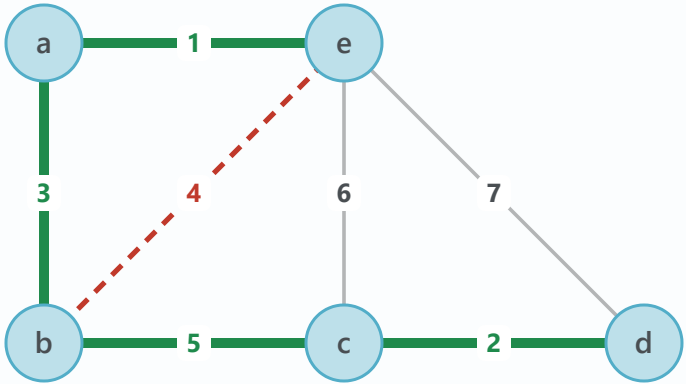


MST	ae, cd, ab, bc
w(T)	11
Ağaçlar	{a, b, c, d, e}

bc = 5: b ve c farklı ağaçlarda → kenar eklenir, iki ağaç birleşir.

Kruskal Adım Adım





MST	ae, cd, ab, bc
w(T)	11
Ağaçlar	tek ağaç

V − 1 = 4 kenar seçildi: **ağaç kuruldu**, w(T) = 11. ec ve ed'ye bakmaya gerek yoktur.

KRUSKAL

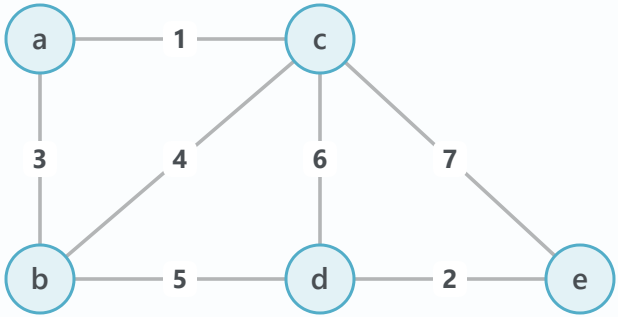
Union-Find (Ayrık Kümeler, Disjoint Set)

- Kesişmeyen kümeleri tutar. Her küme bir **ağaçtır**; kökü kümenin **temsilcisidir**. Ebeveyn dizisi `ata[]` : `ata[i]` i'nin ebeveyni, kökte `ata[r] = r`.
- **oluştur(V)**: her düğüm kendi kümesi → `ata[i] = i`.
- **bul(x)** (find): köke kadar çık, kökü döndür.
- **birleştir(x, y)** (union): bir kökü diğerinin altına bağla.
- Kruskal'da: `bul(u) == bul(v)` ise u ve v aynı ağaçtadır → kenar **döngü** kapatır.

```
int bul(int[] ata, int i) {
  if (ata[i] != i)
    ata[i] = bul(ata, ata[i]); // yol sıkıştırma
  return ata[i];
}
void birlestir(int[] ata, int x, int y) {
  ata[bul(ata, x)] = bul(ata, y);
}
```

İki iyileştirme: *yol kısaltma* (path compression) ve *rütbeye göre birleştirme* (union by rank). İkisi birlikte işlem başına amortize **O(α(V))** verir; α (ters Ackermann) pratikte ≤ 4'tür.

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

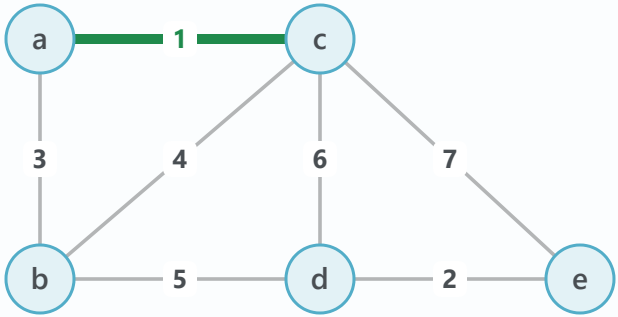


ata[ ]

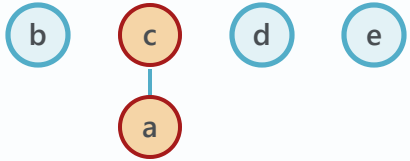
a	b	c	d	e
a	b	c	d	e

`oluştur(5)` : her düğüm kendi kümesinin kökü, `ata[i] = i` . Kenarlar: ac 1, de 2, ab 3, bc 4, bd 5, cd 6, ce 7.

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

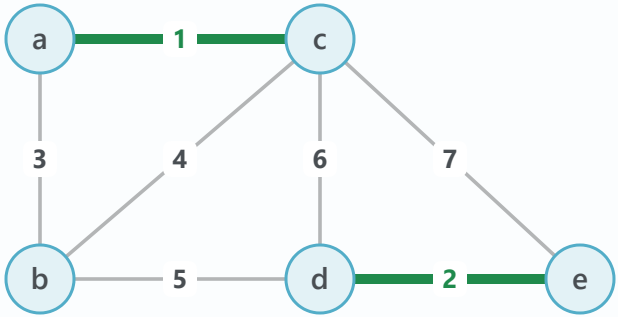


ata[ ]

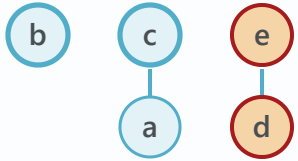
c	b	c	d	e
a	b	c	d	e

ac = 1: bul(a) = a, bul(c) = c → farklı → kenar MST'ye; `ata[a] = c`.

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

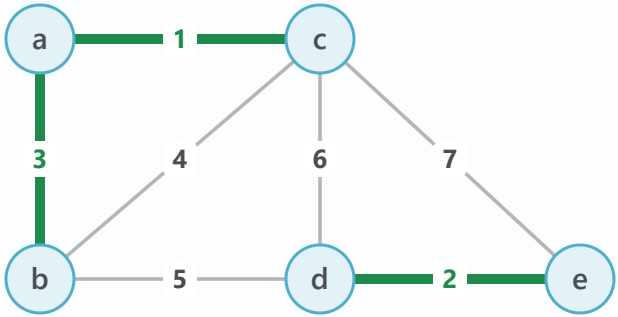


ata[ ]

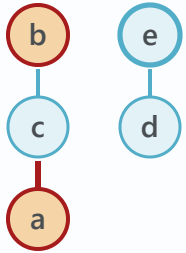
a	b	c	d	e
c	b	c	e	e

de = 2: bul(d) = d, bul(e) = e → farklı → kenar MST'ye; `ata[d] = e` .

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

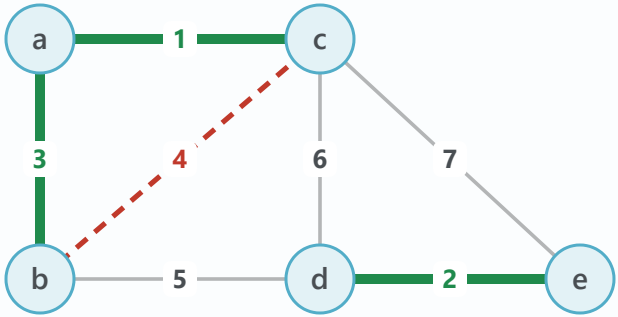


ata[ ]

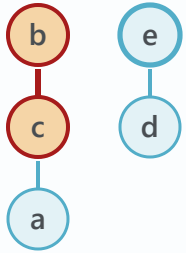
c	b	b	e	e
a	b	c	d	e

ab = 3: bul(a) = c, bul(b) = b → farklı → kenar MST'ye; `ata[c] = b`.

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

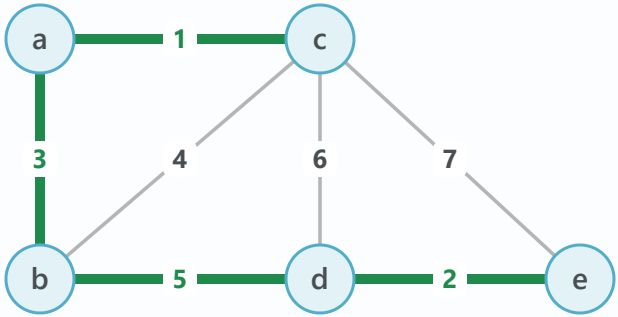


ata[ ]

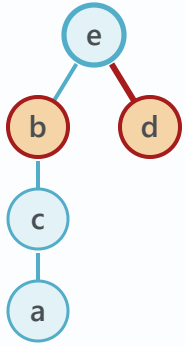
c	b	b	e	e
a	b	c	d	e

bc = 4: bul(b) = b (yol b), bul(c) = b → **aynı kök**: döngü, kenar atlanır.

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

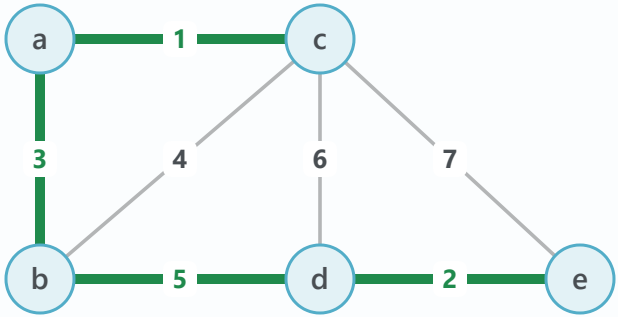


ata[ ]

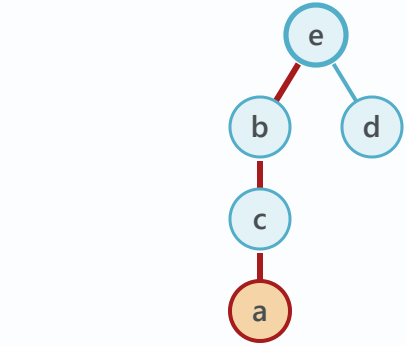
c	e	b	e	e
a	b	c	d	e

bd = 5: bul(b) = b, bul(d) = e → farklı → kenar MST'ye; `ata[b] = e`.

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)

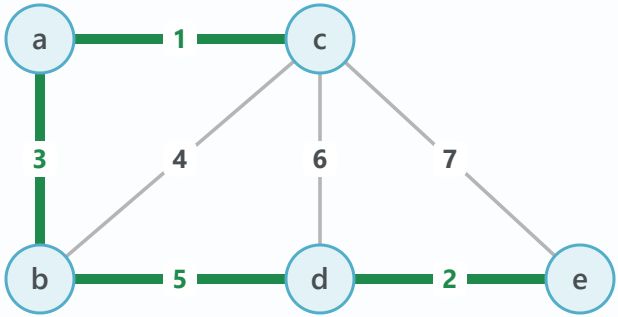


ata[ ]

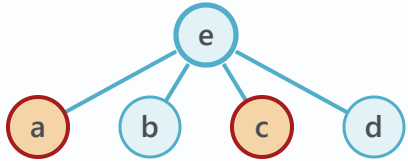
c	e	b	e	e
a	b	c	d	e

MST tamam (4 kenar, w = 11). Ama orman **uzun bir zincir** oldu: `bul(a)` a→c→b→e yolunu izler (3 adım).

Union-Find ile Döngü Denetimi



Ayrık küme ormanı (ok: ebeveyn)



ata[ ]

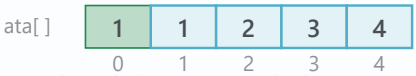
a	b	c	d	e
e	e	e	e	e

Yol kısaltma: `bul(a)` dönerken yoldaki her düğümü doğrudan köke bağlar (ata[a] = ata[c] = e). Sonraki aramalar 1 adım.

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
3     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
▶   ata[bul(ata, x)] = bul(ata, y);
8 }
```





Başta her öğe kendi kümesinin kökü: ata[i] = i.

`birlestir(0, 1)` : bul(0) = 0, bul(1) = 1 → ata[0] = 1; 0 artık 1'in altında.

Adım 1 / 12

DEĞİŞKENLER

`i = –`

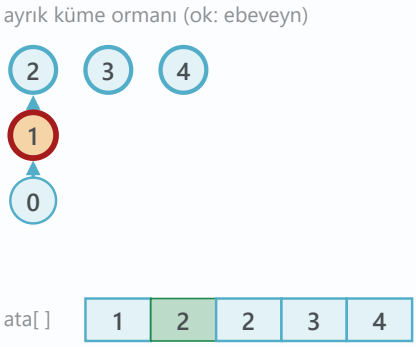
`ata = [1, 1, 2, 3, 4]`

ÇAĞRI YIĞINI

`birlestir(0, 1)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
3     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
▶   ata[bul(ata, x)] = bul(ata, y);
8 }
```



`birlestir(1, 2)` : bul(1) = 1, bul(2) = 2 → ata[1] = 2. Kök kökün altına bağlandı; zincir uzadı: 0 → 1 → 2.

Adım 2 / 12

DEĞİŞKENLER

`i = –`

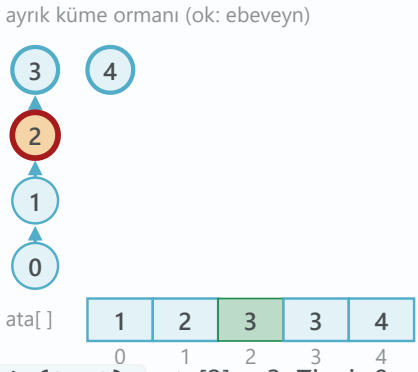
`ata = [1, 2, 2, 3, 4]`

ÇAĞRI YIĞINI

`birlestir(1, 2)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
3     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
▶   ata[bul(ata, x)] = bul(ata, y);
8 }
```



`birlestir(2, 3)` : ata[2] = 3. Zincir 0 → 1 → 2 → 3: bul(0) artık 3 adım sürer.

DEĞİŞKENLER

`i = –`

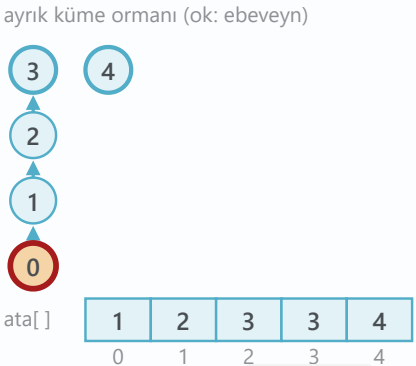
`ata = [1, 2, 3, 3, 4]`

ÇAĞRI YIĞINI

`birlestir(2, 3)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
▶   if (ata[i] != i)
▶     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```



ata[0] = 1 ≠ 0: 0 kök değil → `bul(1)` çağrılır.

DEĞİŞKENLER

`i = 0`

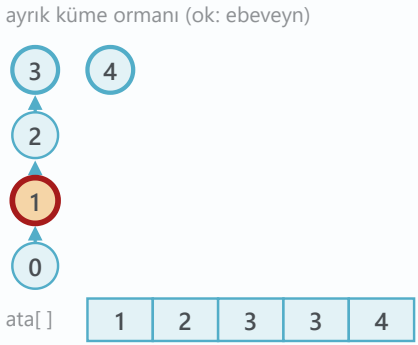
`ata = [1, 2, 3, 3, 4]`

ÇAĞRI YIĞINI

`bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
▶   if (ata[i] != i)
▶     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```



ata[1] = 2 ≠ 1: 1 kök değil → `bul(2)` çağrılır.

DEĞİŞKENLER

`i = 1`

`ata = [1, 2, 3, 3, 4]`

ÇAĞRI YIĞINI

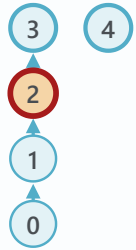
`bul(1)`

`bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
▶   if (ata[i] != i)
▶     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```

ayrık küme ormanı (ok: ebeveyn)



ata[ ]	1	2	3	3	4
	0	1	2	3	4

ata[2] = 3 ≠ 2: 2 kök değil → `bul(3)` çağrılır.

Adım 6 / 12

DEĞİŞKENLER

`i = 2`

`ata = [1, 2, 3, 3, 4]`

ÇAĞRI YIĞINI

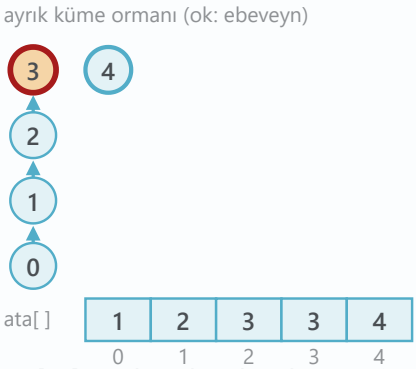
`bul(2)`

`bul(1)`

`bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
▶   if (ata[i] != i)
3     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
▶   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```



ata[3] = 3: 3 **kök**. Kök 3 döndürülür; özyineleme geri sarılmaya başlar.

Adım 7 / 12

DEĞİŞKENLER

`i = 3`

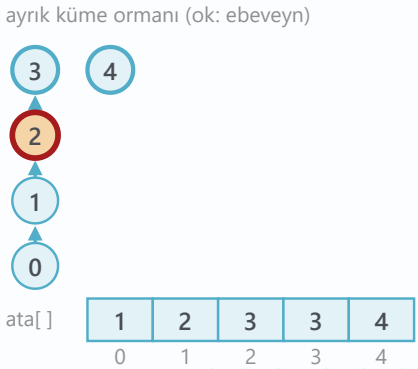
`ata = [1, 2, 3, 3, 4]`

ÇAĞRI YIĞINI

- `bul(3)`
- `bul(2)`
- `bul(1)`
- `bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
▶     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
▶   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```



Dönüşte ata[2] = 3 (zaten köke bağlıydı, değişmedi).

DEĞİŞKENLER

`i = 2`

`ata = [1, 2, 3, 3, 4]`

ÇAĞRI YIĞINI

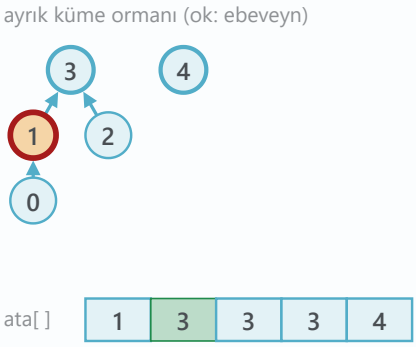
`bul(2)`

`bul(1)`

`bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
▶     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
▶   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```



Dönüşte ata[1] = 3: 1 artık doğrudan köke bağlı (2 → 3). **Yol sıkıştırma**.

DEĞİŞKENLER

`i = 1`

`ata = [1, 3, 3, 3, 4]`

ÇAĞRI YIĞINI

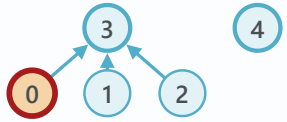
`bul(1)`

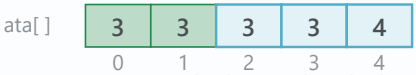
`bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
▶     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
▶   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```

ayrık küme ormanı (ok: ebeveyn)





Dönüşte ata[0] = 3: 0 artık doğrudan köke bağlı (1 → 3). **Yol sıkıştırma**.

Adım 10 / 12

DEĞİŞKENLER

`i = 0`

`ata = [3, 3, 3, 3, 4]`

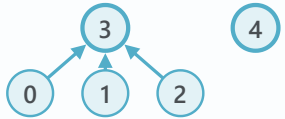
ÇAĞRI YIĞINI

`bul(0)`

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
3     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
▶   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
7   ata[bul(ata, x)] = bul(ata, y);
8 }
```

ayrık küme ormanı (ok: ebeveyn)



ata[ ]

3	3	3	3	4
0	1	2	3	4

`bul(0)` = 3. Yoldaki 0, 1, 2 düğümlerinin hepsi artık doğrudan 3'e bağlı: sonraki bul çağrıları tek adımda biter.

Adım 11 / 12

DEĞİŞKENLER

`i = –`

`ata = [3, 3, 3, 3, 4]`

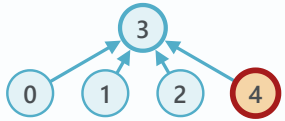
ÇAĞRI YIĞINI

boş

Union-Find: `bul` ve `birlestir` İzleme

```
1 int bul(int[] ata, int i) {
2   if (ata[i] != i)
3     ata[i] = bul(ata, ata[i]); // yol sıkıştırma
4   return ata[i];
5 }
6 void birlestir(int[] ata, int x, int y) {
▶   ata[bul(ata, x)] = bul(ata, y);
8 }
```

ayrık küme ormanı (ok: ebeveyn)



ata[ ]

3	3	3	3	3
0	1	2	3	4

`birlestir(4, 0)` : bul(4) = 4, bul(0) = 3 (tek adım!) → ata[4] = 3. Tüm öğeler tek kümede, kök 3.

Adım 12 / 12

DEĞİŞKENLER

`i = –`

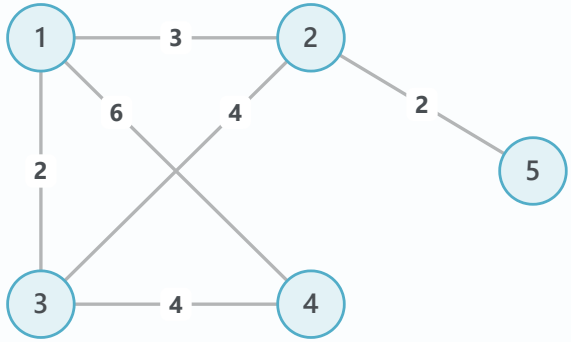
`ata = [3, 3, 3, 3, 3]`

ÇAĞRI YIĞINI

`birlestir(4, 0)`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
▶    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                // birleştir
14     }
15   }
16 }
```



Kenarlar sıralanır: (1,3)=2, (2,5)=2, (1,2)=3, (2,3)=4, (3,4)=4, (1,4)=6.

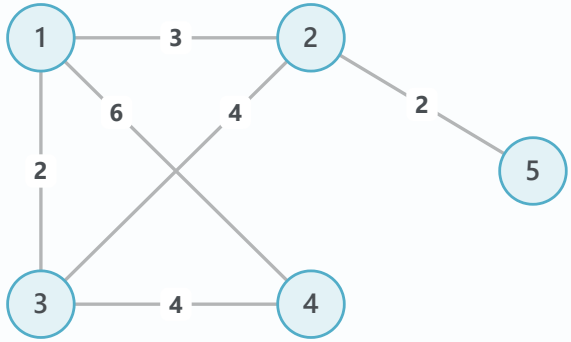
Adım 1 / 13

DEĞİŞKENLER

`V = 5` `k = 0` `agac = []` `w(T) = 0`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
▶    int[] ata = new int[V + 1];
▶    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                // birleştir
14     }
15   }
16 }
```



`ata[i] = i` : her düğüm ayrı bir ağaç (5 ağaç).

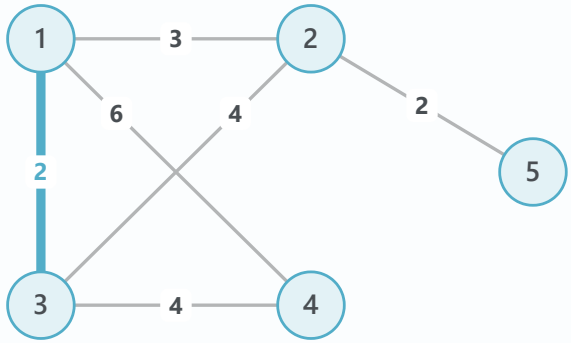
Adım 2 / 13

DEĞİŞKENLER

`V = 5` `ata = [1, 2, 3, 4, 5]` `k = 0` `agac = []` `w(T) = 0`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
▶      Kenar e = ke.get(k++);
▶      int x = bul(ata, e.u);
▶      int y = bul(ata, e.v);
11     if (x != y) {                  // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                  // birleştir
14     }
15   }
16 }
```



e = (1,3), ağırlık 2: `bul(1)` = 1, `bul(3)` = 3.

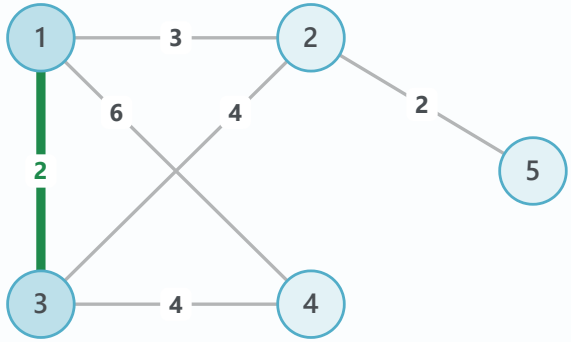
Adım 3 / 13

DEĞİŞKENLER

`V = 5` `ata = [1, 2, 3, 4, 5]` `k = 1`
`e = (1,3,2)` `x = 1` `y = 3` `agac = []`
`w(T) = 0`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
▶        agac.add(e);
▶        ata[x] = y;                // birleştir
14     }
15   }
16 }
```



x ≠ y → (1,3) MST'ye eklenir, `ata[1] = 3`. v1 ve v3 birleşir.

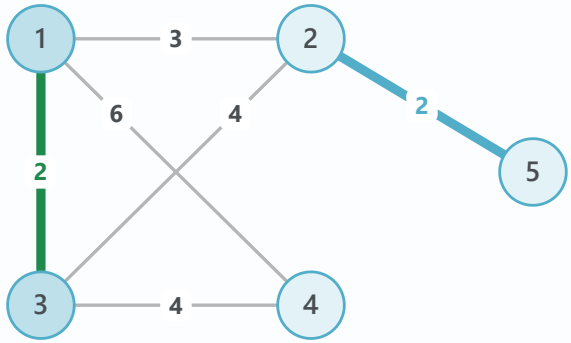
Adım 4 / 13

DEĞİŞKENLER

`V = 5` `ata = [3, 2, 3, 4, 5]` `k = 1` `e = (1,3,2)` `x = 1` `y = 3` `agac = [(1,3)]` `w(T) = 2`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
▶      Kenar e = ke.get(k++);
▶      int x = bul(ata, e.u);
▶      int y = bul(ata, e.v);
11     if (x != y) {                  // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                  // birleştir
14     }
15   }
16 }
```



e = (2,5), ağırlık 2: `bul(2)` = 2, `bul(5)` = 5.

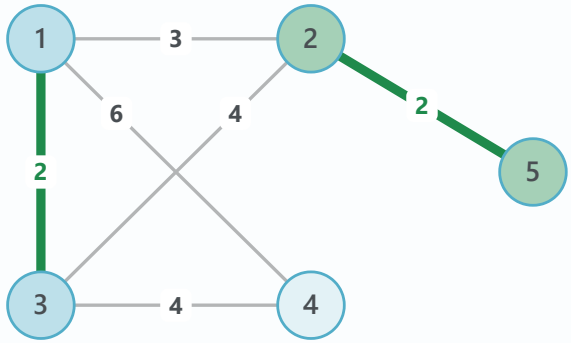
Adım 5 / 13

DEĞİŞKENLER

`V = 5` `ata = [3, 2, 3, 4, 5]` `k = 2` `e = (2,5,2)` `x = 2` `y = 5` `agac = [(1,3)]` `w(T) = 2`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
▶        agac.add(e);
▶        ata[x] = y;                // birleştir
14     }
15   }
16 }
```



x ≠ y → (2,5) MST'ye eklenir, `ata[2] = 5`. v2 ve v5 birleşir.

Adım 6 / 13

DEĞİŞKENLER

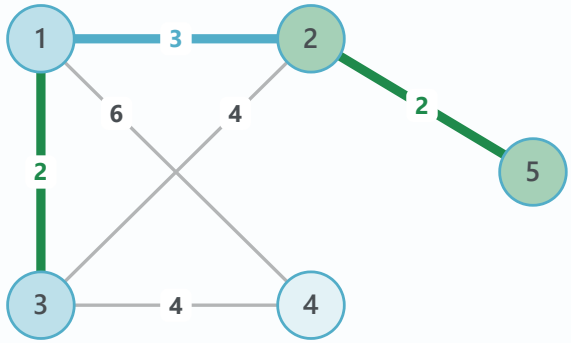
`V = 5` `ata = [3, 5, 3, 4, 5]` `k = 2`

`e = (2,5,2)` `x = 2` `y = 5`

`agac = [(1,3) (2,5)]` `w(T) = 4`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
▶      Kenar e = ke.get(k++);
▶      int x = bul(ata, e.u);
▶      int y = bul(ata, e.v);
11     if (x != y) {                  // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                  // birleştir
14     }
15   }
16 }
```



e = (1,2), ağırlık 3: `bul(1)` = 3, `bul(2)` = 5.

Adım 7 / 13

DEĞİŞKENLER

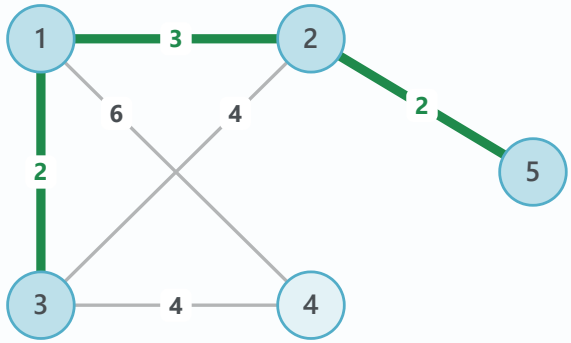
`V = 5` `ata = [3, 5, 3, 4, 5]` `k = 3`

`e = (1,2,3)` `x = 3` `y = 5`

`agac = [(1,3) (2,5)]` `w(T) = 4`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
▶        agac.add(e);
▶        ata[x] = y;                // birleştir
14     }
15   }
16 }
```



x ≠ y → (1,2) MST'ye eklenir, `ata[3] = 5`. Önceki iki ağaç birleşir.

Adım 8 / 13

DEĞİŞKENLER

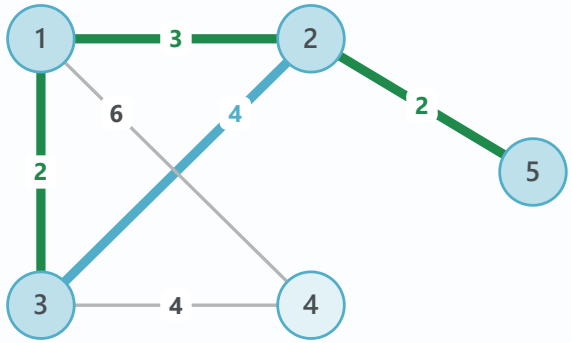
`V = 5` `ata = [3, 5, 5, 4, 5]` `k = 3`

`e = (1,2,3)` `x = 3` `y = 5`

`agac = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
▶      Kenar e = ke.get(k++);
▶      int x = bul(ata, e.u);
▶      int y = bul(ata, e.v);
11     if (x != y) {                  // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                  // birleştir
14     }
15   }
16 }
```



e = (2,3), ağırlık 4: `bul(2)` = 5, `bul(3)` = 5.

Adım 9 / 13

DEĞİŞKENLER

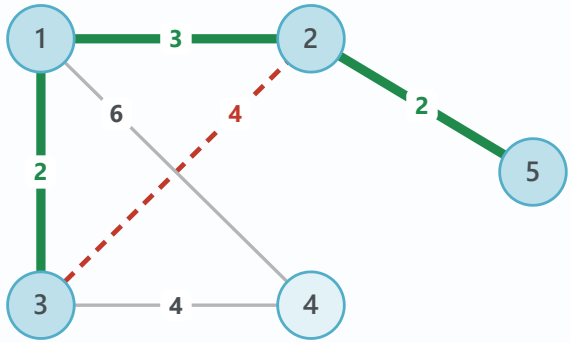
`V = 5` `ata = [3, 5, 5, 4, 5]` `k = 4`

`e = (2,3,4)` `x = 5` `y = 5`

`agac = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
▶      if (x != y) {                // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                // birleştir
14     }
15   }
16 }
```



x = y = 5: (2,3) iki ayrı ağacı birleştirmiyor → **döngü**, atlanır.

Adım 10 / 13

DEĞİŞKENLER

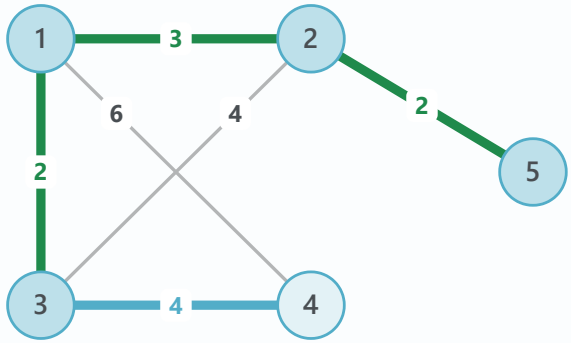
`V = 5` `ata = [3, 5, 5, 4, 5]` `k = 4`

`e = (2,3,4)` `x = 5` `y = 5`

`agac = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
▶      Kenar e = ke.get(k++);
▶      int x = bul(ata, e.u);
▶      int y = bul(ata, e.v);
11     if (x != y) {                  // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                  // birleştir
14     }
15   }
16 }
```



e = (3,4), ağırlık 4: `bul(3)` = 5, `bul(4)` = 4.

Adım 11 / 13

DEĞİŞKENLER

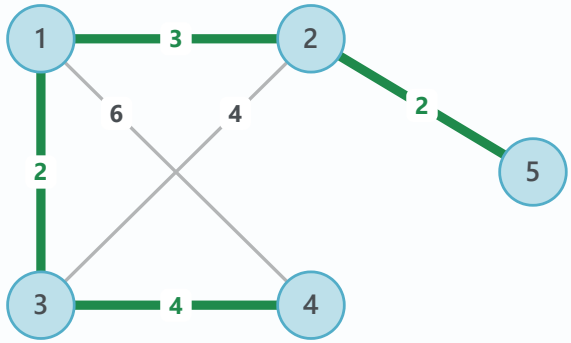
`V = 5` `ata = [3, 5, 5, 4, 5]` `k = 5`

`e = (3,4,4)` `x = 5` `y = 4`

`agac = [(1,3) (2,5) (1,2)]` `w(T) = 7`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
7    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
▶        agac.add(e);
▶        ata[x] = y;                // birleştir
14     }
15   }
16 }
```



x ≠ y → (3,4) MST'ye eklenir, `ata[5] = 4`. v4 ağaca katılır; w(T) = 11.

Adım 12 / 13

DEĞİŞKENLER

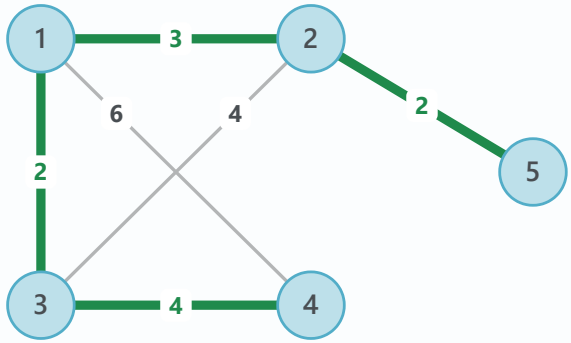
`V = 5` `ata = [3, 5, 5, 4, 4]` `k = 5`

`e = (3,4,4)` `x = 5` `y = 4`

`agac = [(1,3) (2,5) (1,2) (3,4)]` `w(T) = 11`

Kruskal: Kodu Adım Adım İzle

```
1  void kruskal(int V, List<Kenar> ke) {
2    Collections.sort(ke);          // artan
3    int[] ata = new int[V + 1];
4    for (int i = 1; i <= V; i++) ata[i] = i;
5    List<Kenar> agac = new ArrayList<>();
6    int k = 0;
▶    while (agac.size() < V - 1) {
8      Kenar e = ke.get(k++);
9      int x = bul(ata, e.u);
10     int y = bul(ata, e.v);
11     if (x != y) {                // ayrı ağaç
12       agac.add(e);
13       ata[x] = y;                // birleştir
14     }
15   }
16 }
```



`agac.size() = 4 = V − 1` : döngü biter. (1,4)=6 hiç incelenmez. **w(T) = 11**.

Adım 13 / 13

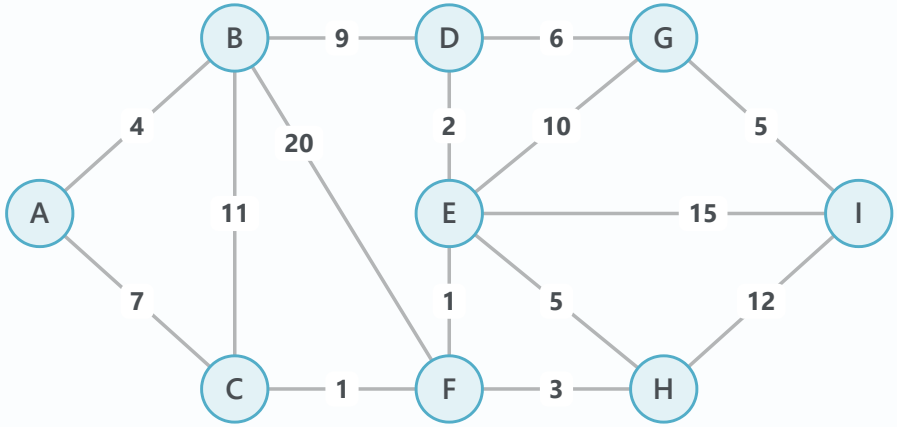
DEĞİŞKENLER

`V = 5` `ata = [3, 5, 5, 4, 4]` `k = 5`

`agac = [(1,3) (2,5) (1,2) (3,4)]` `w(T) = 11`

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

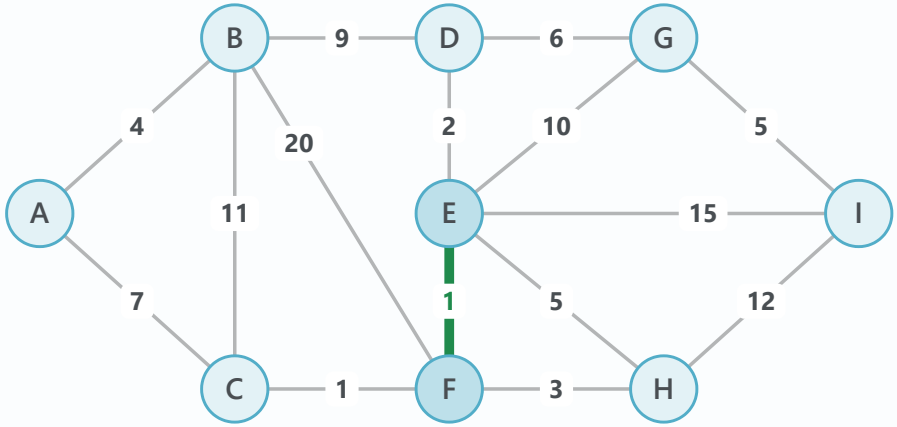


Kümeler: tek düğümlüler · Ağırlık: 0 · Bağlı: HAYIR

Kenarlar sıralı: 1, 1, 2, 3, 4, 5, 5, 6, 7, 9, 10, 11, 12, 15, 20. Henüz hiçbir kenar seçilmedi.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

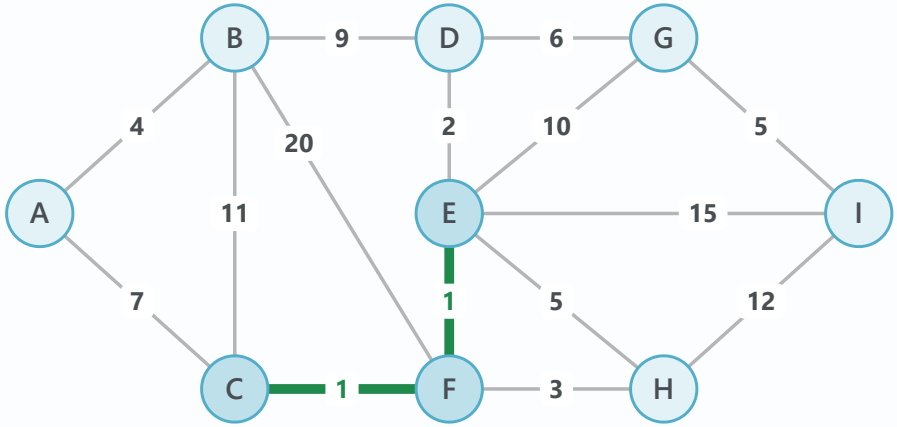


Kümeler: {E, F} · Ağırlık: 1 · Bağlı: HAYIR

EF = 1 eklendi (iki farklı küme). Toplam ağırlık 1.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

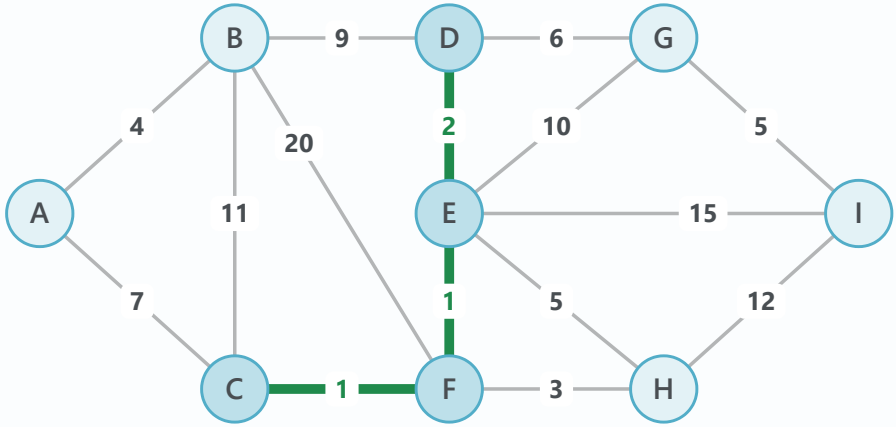


Kümeler: {C, E, F} · Ağırlık: 2 · Bağlı: HAYIR

CF = 1 eklendi (iki farklı küme). Toplam ağırlık 2.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

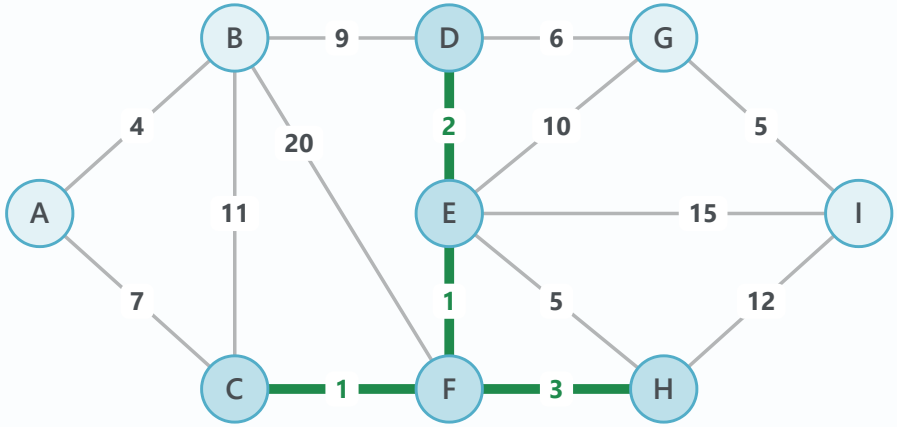


Kümeler: {C, D, E, F} · Ağırlık: 4 · Bağlı: HAYIR

DE = 2 eklendi (iki farklı küme). Toplam ağırlık 4.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

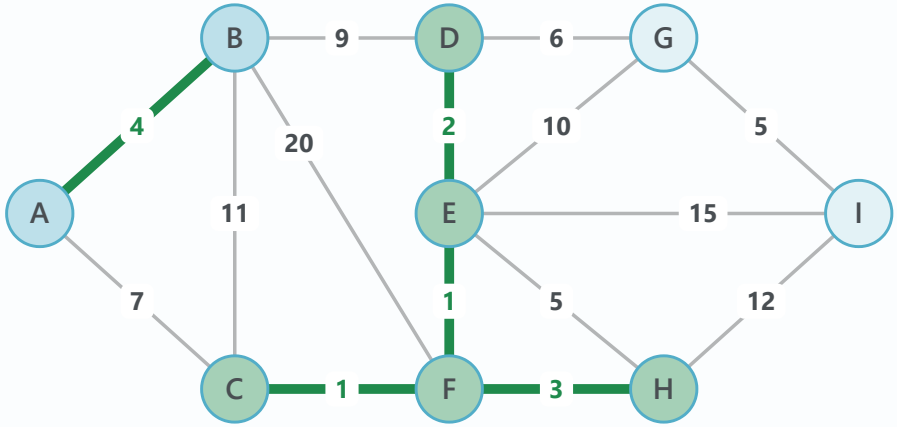


Kümeler: {C, D, E, F, H} · Ağırlık: 7 · Bağlı: HAYIR

FH = 3 eklendi (iki farklı küme). Toplam ağırlık 7.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

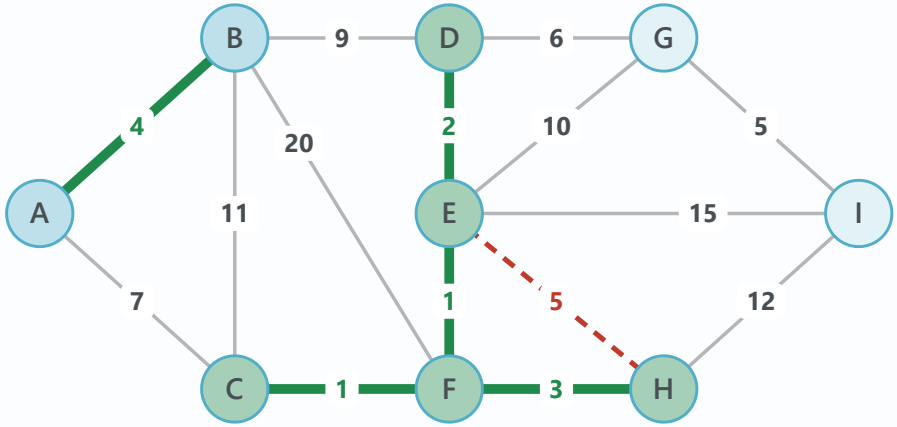


Kümeler: {C, D, E, F, H} {A, B} · Ağırlık: 11 · Bağlı: HAYIR

AB = 4 eklendi (iki farklı küme). Toplam ağırlık 11.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

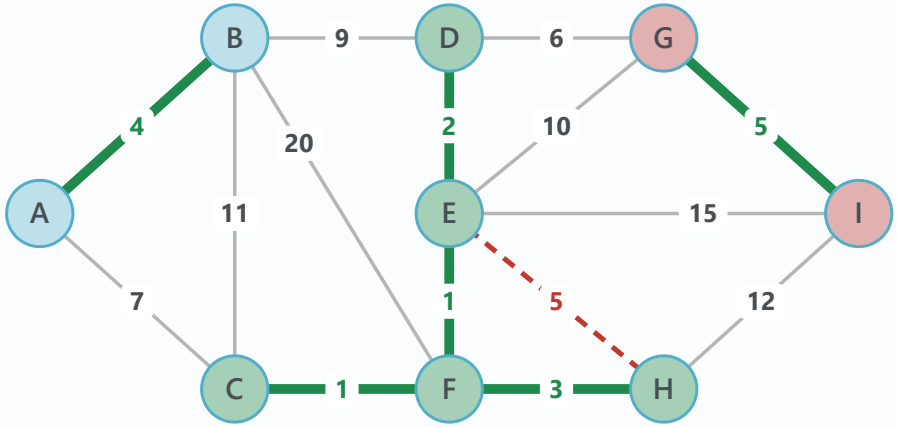


Kümeler: {C, D, E, F, H} {A, B} · Ağırlık: 11 · Bağlı: HAYIR

EH = 5: E ve H zaten aynı kümede (E–F–H yolu) → döngü, eklenmez. Toplam 11 kalır.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

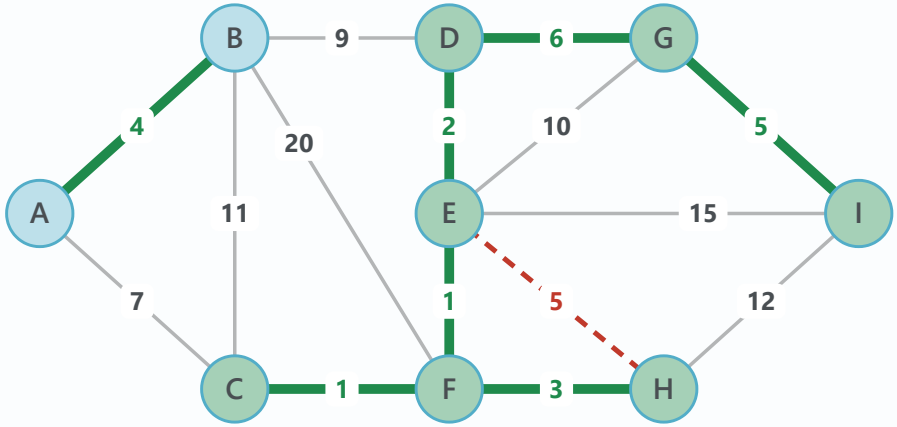


Kümeler: {C, D, E, F, H} {A, B} {G, I} · Ağırlık: 16 · Bağlı: HAYIR

GI = 5 eklendi (iki farklı küme). Toplam ağırlık 16.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20

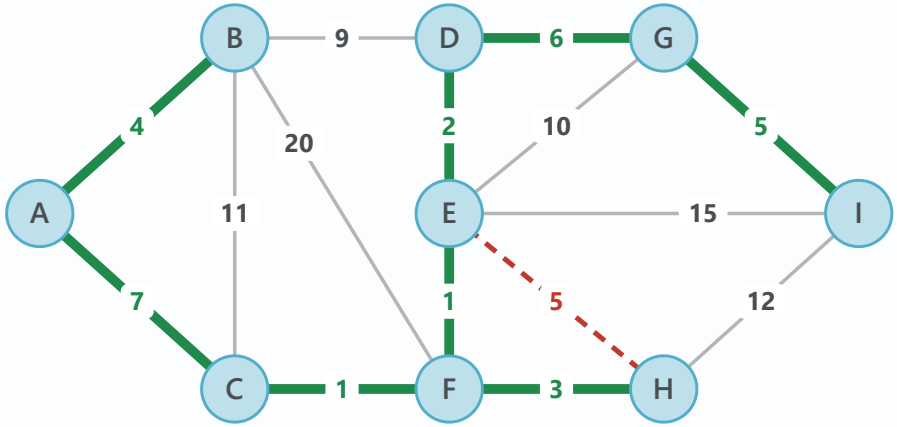


Kümeler: {C, D, E, F, G, H, I} {A, B} · Ağırlık: 22 · Bağlı: HAYIR

DG = 6 eklendi (iki farklı küme). Toplam ağırlık 22.

Kruskal Örneği: Dokuz Düğüm

EF	CF	DE	FH	AB	EH	GI	DG	AC	BD	EG	BC	HI	EI	BF
1	1	2	3	4	5	5	6	7	9	10	11	12	15	20



Kümeler: {A, B, C, D, E, F, G, H, I} · Ağırlık: 29 · Bağlı: EVET

AC = 7 eklendi (iki farklı küme). Toplam ağırlık 29. 8 = V − 1 kenar: **MST = 29**, çizge bağlı.

Kruskal: Karmaşıklık Analizi

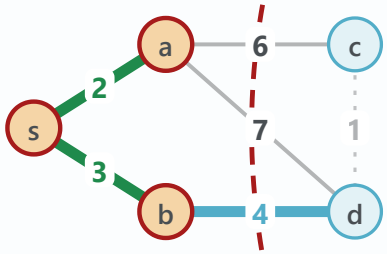
İşlem	Kaç kez	Maliyet
Kenarları ağırlığa göre sıralama	1	**O(E log E)**
`oluştur(V)`	1	O(V)
`bul` (her kenar için 2)	≤ 2E	O(E · α(V)) — yol kısaltma + rütbe O(E log V) — yalnızca rütbe
`birleştir`	V − 1	

- Toplam: **O(E log E)**; sıralama baskındır. $E \le V^2$ olduğundan $\log E \le 2 \log V$, yani $O(E \log E) = O(E \log V)$.
- Kenarlar zaten sıralıysa (ya da ağırlıklar küçük tam sayıysa, saymalı sıralama) süre neredeyse doğrusal: $O(E \cdot \alpha(V))$.
- Bellek: kenar listesi + ebeveyn dizisi (`ata`) → O(V + E).

Union-find işlemleri yalnızca rütbeye göre birleştirmeyle en kötü O(log V) sürer; yol kısaltma da eklenince işlem başına amortize maliyet **O(α(V))** olur. Her iki durumda da toplamı sıralama belirler.

Prim Algoritması

- Tüm düğümleri bağlayan en küçük ağırlıklı ağacı oluşturur; **açgözlü** bir algoritmadır.
- Robert C. **Prim** (1957); daha önce V. Jarník (1930) bulmuş, E. W. Dijkstra (1959) yeniden keşfetmiştir.
- Bir **başlangıç düğümü** seçilir; her adımda ağaçta olmayan düğümler arasından, ağaca **en ucuz kenarla** bağlanan düğüm ağaca eklenir.
- Her an **tek bir ağaç** vardır (Kruskal'daki ormandan farklı olarak).



kesimi geçen en ucuz kenar: 4

Güçlü yönler

+ **Yoğun** çizgelerde uygun: matrisle $O(V^2)$.
+ Kenar sıralaması gerekmez.
+ Komşuluk listesi/matrisi ile doğal çalışır.

Zayıf yönler

– Performans **veri yapısına bağlı** (dizi, ikili heap, Fibonacci heap).

– Bağlı olmayan çizgede yalnızca başlangıcın bileşenini kapsar.

Prim: Adımlar, Anahtar ve Ebeveyn

1. Başlangıç düğümü seçilir ve bu düğüme ait tüm kenarlar **öncelikli kuyruğa** eklenir.
2. Kuyruktan, ağaçta olmayan bir düğüme giden **en küçük ağırlıklı** kenar seçilir.
3. Seçilen kenar ile bağlantılı yeni düğüm **ağaca eklenir**.
4. Yeni düğüme bağlı kenarlar kuyruğa eklenir; tüm düğümler ağaca girene kadar 2–4 tekrarlanır.

anahtar[v] (key)
v'yi ağaca bağlayan şimdiye kadarki **en ucuz kenarın** ağırlığı. Başta ∞, kaynakta 0.

ata[v]: ebeveyn (π, parent)
Bu en ucuz kenarın ağaçtaki ucu. Sonunda (ata[v], v) kenarları MST'yi verir.

Kuyruktan çıkan düğüm için, ağaçta olmayan her komşu v'de: `ağırlık(u, v) < anahtar[v]` ise anahtar ve ebeveyn güncellenir.

Prim: Sözde Kod

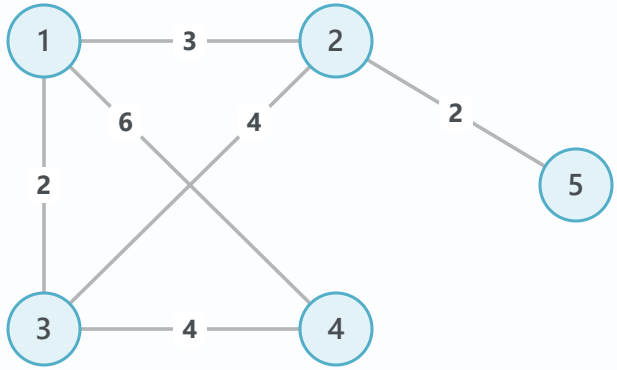
```
PRIM(G, kaynak):
  MST = [ ]
  mesafeler = [ ]
  ata = [ ]
  ziyaretEdildi = [ ]
  her bir v için V içinde:
    mesafeler[v] = ∞
    ziyaretEdildi[v] = 0
  mesafeler[kaynak] = 0
  öncelikKuyruğu.ekle(kaynak, 0)
```

```
döngü öncelikKuyruğu boş değil iken:
  u = öncelikKuyruğu.çıkar()
  eğer ziyaretEdildi[u] == 0:
    ziyaretEdildi[u] = 1
    her bir (u, v, ağırlık) için G[u] içinde:
      eğer ziyaretEdildi[v] == 0 ve
           ağırlık < mesafeler[v]:
        mesafeler[v] = ağırlık
        ata[v] = u
        öncelikKuyruğu.ekle(v, ağırlık)
her bir v için V içinde (kaynak hariç):
  MST.ekle((ata[v], v, mesafeler[v]))
döndür MST
```

Bu "tembel" (lazy) sürüm bir düğümü kuyruğa birden çok kez ekleyebilir; `ziyaretEdildi` denetimi eski kayıtları atar. Kuyruk en fazla E kayıt tutar → O(E log E).

PRIM

Prim Adım Adım: Öncelikli Kuyruk



key	u	π
0	1	/
∞	2	/
∞	3	/
∞	4	/
∞	5	/

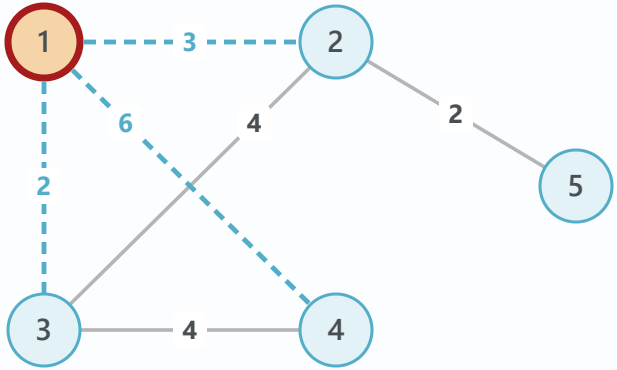
w(T) = 0

İlklendirme: kuyruk Q tüm düğümlerle başlatılır; `key(1) = 0`, diğerleri ∞, π = /.

Adım 1 / 7

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim Adım Adım: Öncelikli Kuyruk



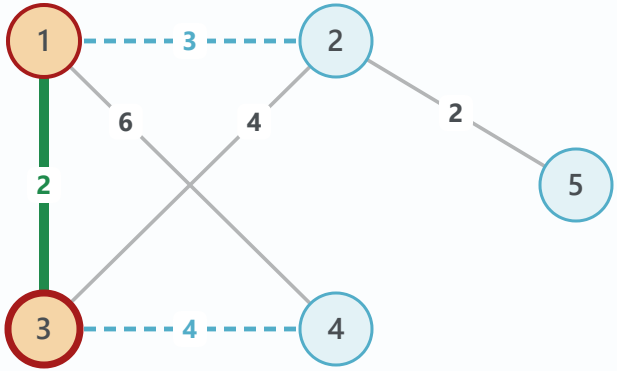
key	u	π
2	3	1
3	2	1
6	4	1
∞	5	/

w(T) = 0

Kuyruktan düğüm 1 çıkar. Komşular güncellenir: key(3) = 2 (1,3), key(2) = 3 (1,2), key(4) = 6 (1,4).

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim Adım Adım: Öncelikli Kuyruk



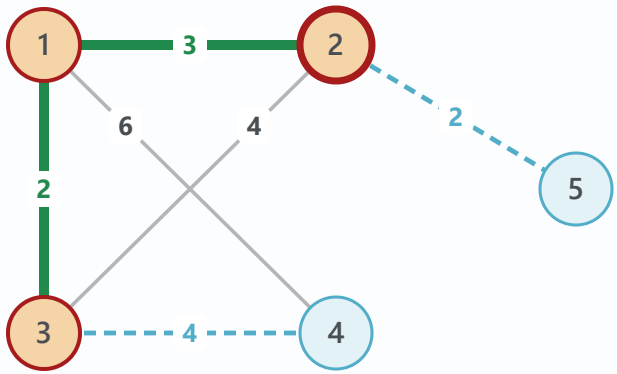
key	u	π
3	2	1
4	4	3
∞	5	/

w(T) = 2

Düğüm 3 çıkar, T'ye **(1,3)** eklenir. key(4) = 4 (3,4) olur; (2,3) = 4 > key(2) = 3 → değişmez.

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim Adım Adım: Öncelikli Kuyruk



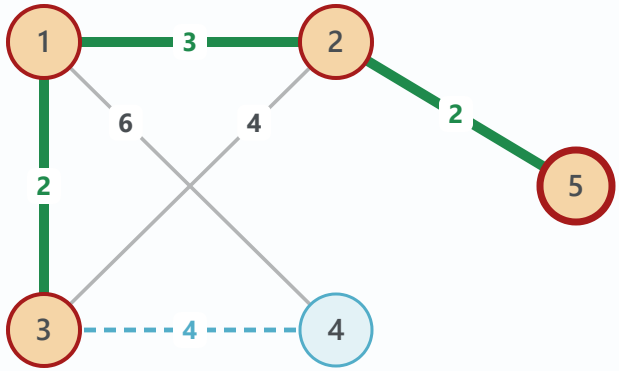
key	u	π
2	5	2
4	4	3

w(T) = 5

Düğüm 2 çıkar, T'ye **(1,2)** eklenir. key(5) = 2 (2,5).

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim Adım Adım: Öncelikli Kuyruk



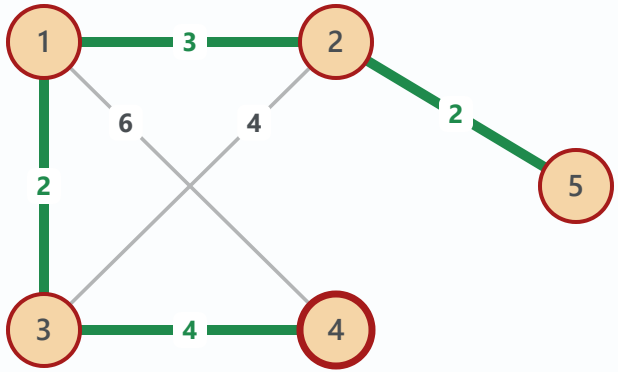
key	u	π
4	4	3

w(T) = 7

Düğüm 5 çıkar, T'ye **(2,5)** eklenir. Ağaçta olmayan komşusu yok: Q'da güncelleme yok.

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim Adım Adım: Öncelikli Kuyruk



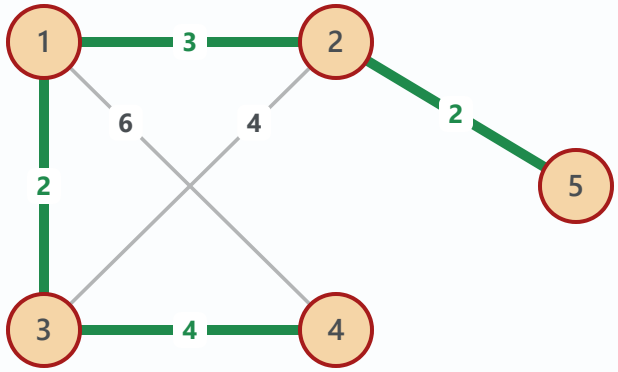
key	u	π
Q = ∅		

w(T) = 11

Düğüm 4 çıkar, T'ye **(3,4)** eklenir. Q'da güncelleme yok.

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim Adım Adım: Öncelikli Kuyruk



key	u	π
Q = ∅		

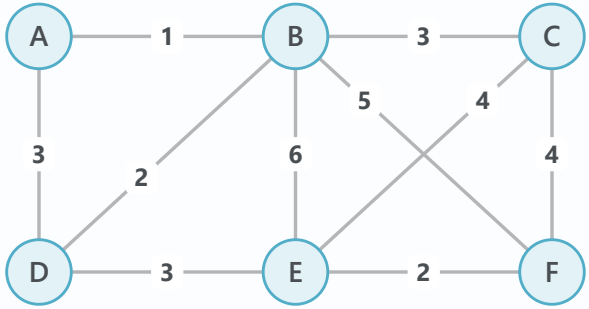
w(T) = 11

Son durum: Q = ∅. MST = (1,3), (1,2), (2,5), (3,4), **w(T) = 11** — Kruskal ile aynı ağaç.

Kesikli çizgi: π ile gösterilen, henüz kesinleşmemiş en ucuz bağlantı. Q tablosu key değerine göre sıralıdır.

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
▶    Arrays.fill(anahtar, Integer.MAX_VALUE);
▶    anahtar[kaynak] = 0;
▶    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



A–F çizgesi, kaynak **B** (indis 1). Tüm anahtarlar ∞; `anahtar[B] = 0`, `ata[B] = −1`.

Adım 1 / 15

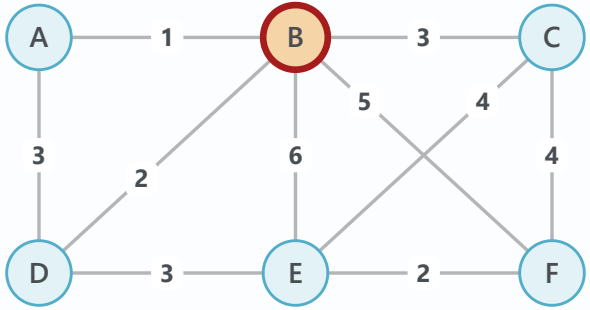
DEĞİŞKENLER

`kaynak = B` `anahtar = [∞, 0, ∞, ∞, ∞, ∞]`

`ata = [−, −, −, −, −, −]` `agacta = {}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = enKucukAnahtar(anahtar,
   agacta);
▶      agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



`enKucukAnahtar` → u = **B** (anahtar 0): kaynak ağaca girer.

Adım 2 / 15

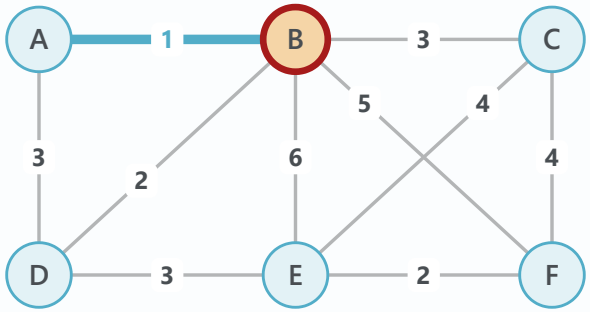
DEĞİŞKENLER

`s = 0` `u = B` `anahtar = [∞, 0, ∞, ∞, ∞, ∞]`

`ata = [-, -, -, -, -, -]` `agacta = {B}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = A: G[B][A] = 1 < anahtar[A] = ∞ → `anahtar[A] = 1` , `ata[A] = B` .

Adım 3 / 15

DEĞİŞKENLER

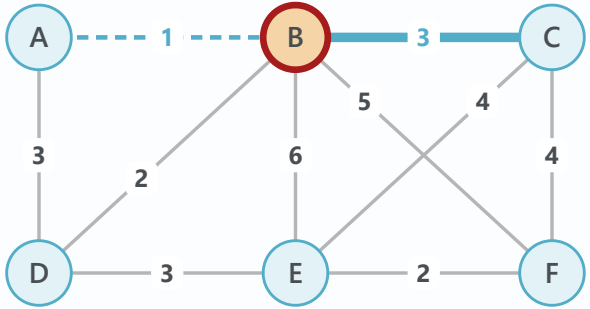
`s = 0` `u = B` `v = A`

`anahtar = [1, 0, ∞, ∞, ∞, ∞]`

`ata = [B, -, -, -, -, -]` `agacta = {B}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = C: G[B][C] = 3 < anahtar[C] = ∞ → `anahtar[C] = 3` , `ata[C] = B` .

Adım 4 / 15

DEĞİŞKENLER

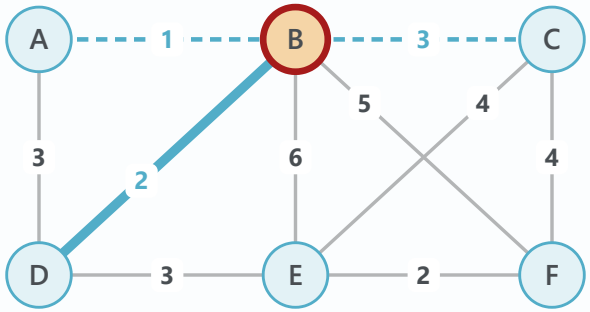
`s = 0` `u = B` `v = C`

`anahtar = [1, 0, 3, ∞, ∞, ∞]`

`ata = [B, –, B, –, –, –]` `agacta = {B}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = D: G[B][D] = 2 < anahtar[D] = ∞ → `anahtar[D] = 2` , `ata[D] = B` .

Adım 5 / 15

DEĞİŞKENLER

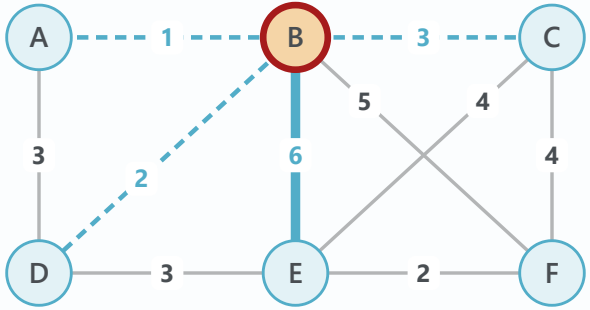
`s = 0` `u = B` `v = D`

`anahtar = [1, 0, 3, 2, ∞, ∞]`

`ata = [B, –, B, B, –, –]` `agacta = {B}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = E: G[B][E] = 6 < anahtar[E] = ∞ → `anahtar[E] = 6` , `ata[E] = B` .

Adım 6 / 15

DEĞİŞKENLER

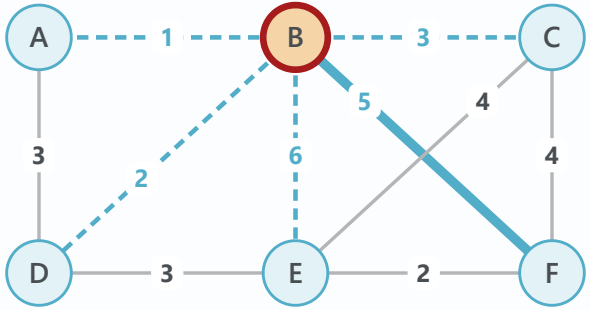
`s = 0` `u = B` `v = E`

`anahtar = [1, 0, 3, 2, 6, ∞]`

`ata = [B, –, B, B, B, –]` `agacta = {B}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = F: G[B][F] = 5 < anahtar[F] = ∞ → `anahtar[F] = 5` , `ata[F] = B` .

Adım 7 / 15

DEĞİŞKENLER

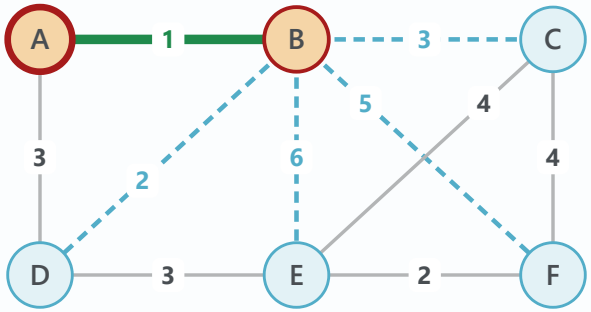
`s = 0` `u = B` `v = F`

`anahtar = [1, 0, 3, 2, 6, 5]`

`ata = [B, –, B, B, B, B]` `agacta = {B}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = enKucukAnahtar(anahtar,
   agacta);
▶      agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



`enKucukAnahtar` → u = **A** (anahtar 1); (B,A) ağaca girer. w(T) = 1. AD = 3 > anahtar[D] = 2: güncelleme yok.

Adım 8 / 15

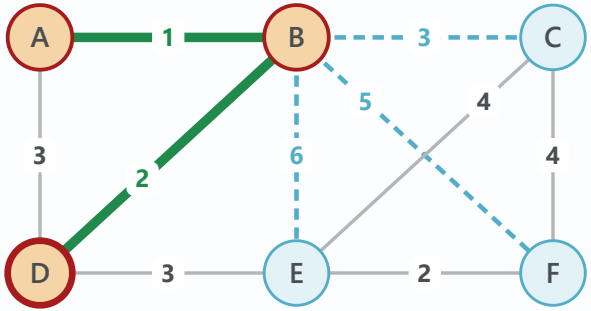
DEĞİŞKENLER

`s = 1` `u = A` `anahtar = [1, 0, 3, 2, 6, 5]`

`ata = [B, –, B, B, B, B]` `agacta = {B,A}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = enKucukAnahtar(anahtar,
   agacta);
▶      agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



`enKucukAnahtar` → u = **D** (anahtar 2); (B,D) ağaca girer. w(T) = 3.

Adım 9 / 15

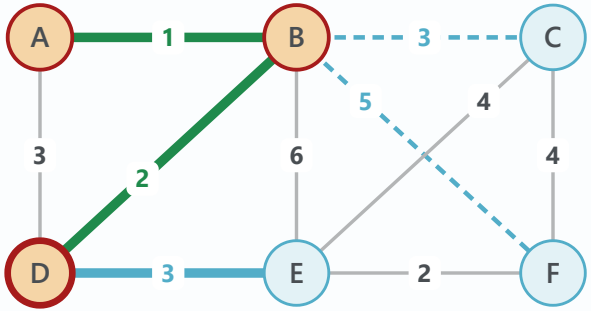
DEĞİŞKENLER

`s = 2` `u = D` `anahtar = [1, 0, 3, 2, 6, 5]`

`ata = [B, –, B, B, B, B]` `agacta = {B,A,D}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = E: G[D][E] = 3 < anahtar[E] = 6 → `anahtar[E] = 3` , `ata[E] = D` .

Adım 10 / 15

DEĞİŞKENLER

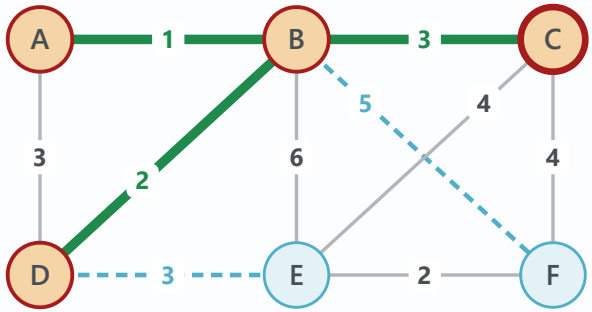
`s = 2` `u = D` `v = E`

`anahtar = [1, 0, 3, 2, 3, 5]`

`ata = [B, –, B, B, D, B]` `agacta = {B,A,D}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = enKucukAnahtar(anahtar,
   agacta);
▶      agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



`enKucukAnahtar` → u = **C** (anahtar 3); (B,C) ağaca girer. w(T) = 6. C ve E eşit (3): küçük indisli C.

Adım 11 / 15

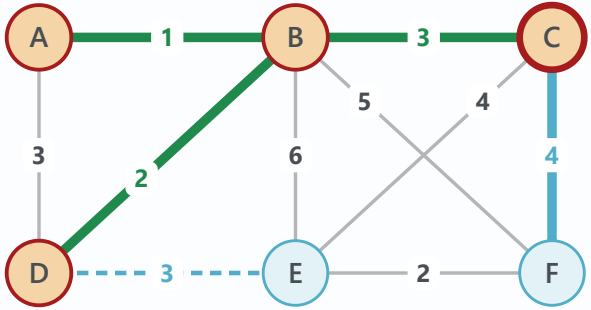
DEĞİŞKENLER

`s = 3` `u = C` `anahtar = [1, 0, 3, 2, 3, 5]`

`ata = [B, –, B, B, D, B]` `agacta = {B,A,D,C}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = F: G[C][F] = 4 < anahtar[F] = 5 → `anahtar[F] = 4` , `ata[F] = C` .

Adım 12 / 15

DEĞİŞKENLER

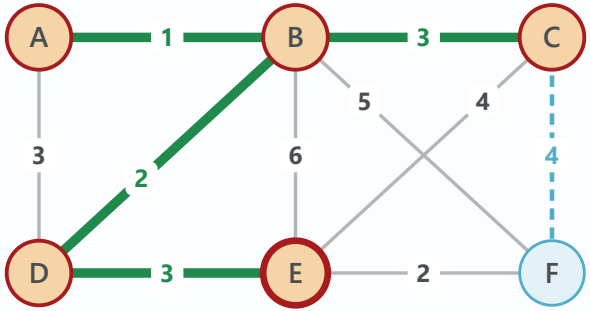
`s = 3` `u = C` `v = F`

`anahtar = [1, 0, 3, 2, 3, 4]`

`ata = [B, –, B, B, D, C]` `agacta = {B,A,D,C}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
▶      int u = enKucukAnahtar(anahtar,
   agacta);
▶      agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



`enKucukAnahtar` → u = **E** (anahtar 3); (D,E) ağaca girer. w(T) = 9.

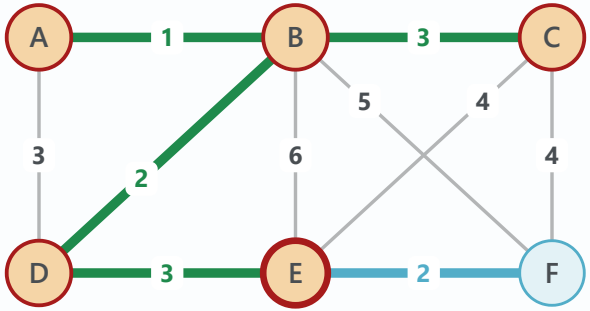
Adım 13 / 15

DEĞİŞKENLER

`s = 4` `u = E` `anahtar = [1, 0, 3, 2, 3, 4]` `ata = [B, –, B, B, D, C]` `agacta = {B,A,D,C,E}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
9    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
▶          anahtar[v] = G[u][v];
▶          ata[v] = u;
17       }
18   }
19 }
```



v = F: G[E][F] = 2 < anahtar[F] = 4 → `anahtar[F] = 2` , `ata[F] = E` .

Adım 14 / 15

DEĞİŞKENLER

`s = 4` `u = E` `v = F`

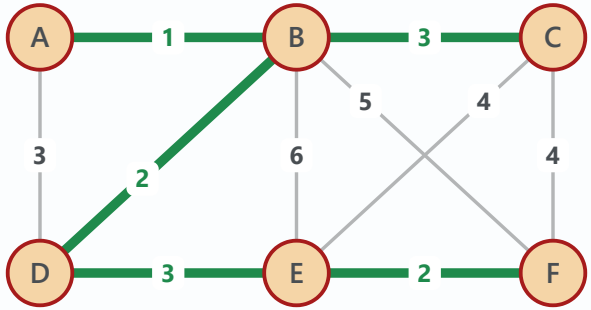
`anahtar = [1, 0, 3, 2, 3, 2]`

`ata = [B, –, B, B, D, E]`

`agacta = {B,A,D,C,E}`

Prim: Kodu Adım Adım İzle (Komşuluk Matrisi)

```
1  void prim(int[][] G, int kaynak) {
2    int V = G.length;
3    int[] anahtar = new int[V];
4    int[] ata = new int[V];
5    boolean[] agacta = new boolean[V];
6    Arrays.fill(anahtar, Integer.MAX_VALUE);
7    anahtar[kaynak] = 0;
8    ata[kaynak] = -1;
▶    for (int s = 0; s < V - 1; s++) {
10     int u = enKucukAnahtar(anahtar,
   agacta);
11     agacta[u] = true;
12     for (int v = 0; v < V; v++)
13       if (G[u][v] != 0 && !agacta[v]
14           && G[u][v] < anahtar[v]) {
15         anahtar[v] = G[u][v];
16         ata[v] = u;
17       }
18   }
19 }
```



Döngü V − 1 = 5 kez döndü; son düğüm F'nin ebeveyni E. MST = (ata[v], v): AB, BC, BD, DE, EF → **w(T) = 11**.

Adım 15 / 15

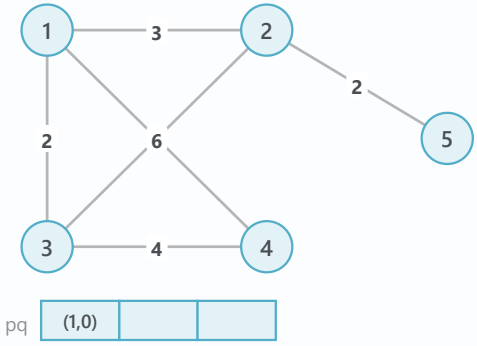
DEĞİŞKENLER

`anahtar = [1, 0, 3, 2, 3, 2]`

`ata = [B, −, B, B, D, E]` `w(T) = 11`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
▶    Arrays.fill(anahtar, SONSUZ);
▶    anahtar[kaynak] = 0;
▶    PriorityQueue<int[]> pq =
▶        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
▶    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
8      int u = pq.poll()[0];
9      if (agacta[u]) continue;  // eski
10     agacta[u] = true;
11     for (Kenar k : komsu.get(u))
12       if (!agacta[k.v] && k.w < anahtar[k.v]) {
13         anahtar[k.v] = k.w; ata[k.v] = u;
14         pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



Beş düğümlü yönsüz çizge, kaynak 1. anahtar[1] = 0, diğerleri ∞; kuyrukta tek kayıt (düğüm, anahtar) = (1,0). Ebeveyn dizisinde (ata[]) − henüz atanmadı demek.

Adım 1 / 8

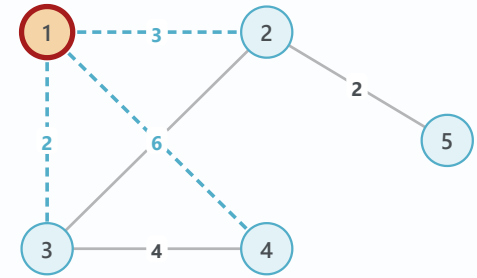
DEĞİŞKENLER

`u = –` `anahtar = [0, ∞, ∞, ∞, ∞]`

`ata = [–, –, –, –, –]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (agacta[u]) continue;  // eski
▶      agacta[u] = true;
▶      for (Kenar k : komsu.get(u))
▶        if (!agacta[k.v] && k.w < anahtar[k.v]) {
▶          anahtar[k.v] = k.w; ata[k.v] = u;
▶          pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



(1,0) en küçük kayıt: 1 ağaca girer. anahtar[2]: ∞ → 3, anahtar[3]: ∞ → 2, anahtar[4]: ∞ → 6; yeni kayıtlar kuyruğa eklendi.

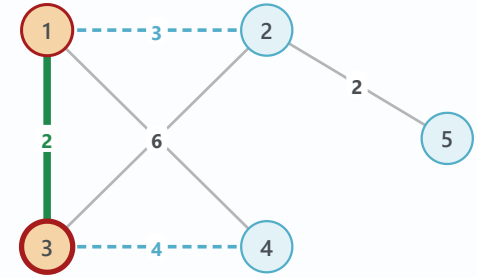
Adım 2 / 8

DEĞİŞKENLER

`u = 1` `anahtar = [0, 3, 2, 6, ∞]` `ata = [-, 1, 1, 1, -]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (agacta[u]) continue;  // eski
▶      agacta[u] = true;
▶      for (Kenar k : komsu.get(u))
▶        if (!agacta[k.v] && k.w < anahtar[k.v]) {
▶          anahtar[k.v] = k.w; ata[k.v] = u;
▶          pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



(3,2) en küçük kayıt: 3 ağaca girer (kenar 1–3, ağırlık 2). anahtar[4]: 6 → 4; yeni kayıtlar kuyruğa eklendi. 2'ye 4'lük kenar, mevcut 3'ten pahalı: güncelleme yok. (4,6) kaydı kuyrukta kalır.

Adım 3 / 8

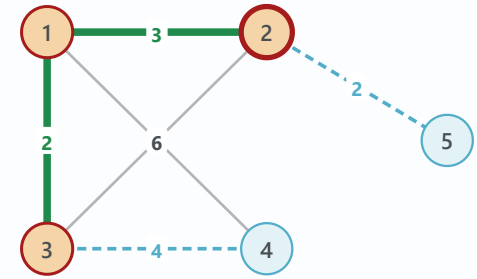
DEĞİŞKENLER

`u = 3` `anahtar = [0, 3, 2, 4, ∞]`

`ata = [–, 1, 1, 3, –]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (agacta[u]) continue;  // eski
▶      agacta[u] = true;
▶      for (Kenar k : komsu.get(u))
▶        if (!agacta[k.v] && k.w < anahtar[k.v]) {
▶          anahtar[k.v] = k.w; ata[k.v] = u;
▶          pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



(2,3) en küçük kayıt: 2 ağaca girer (kenar 1–2, ağırlık 3). anahtar[5]: ∞ → 2; yeni kayıtlar kuyruğa eklendi.

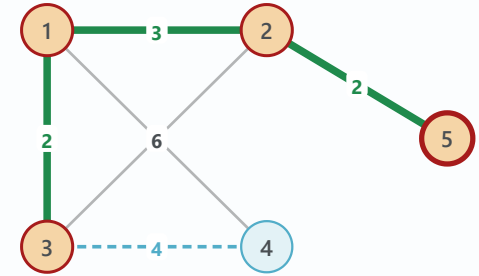
Adım 4 / 8

DEĞİŞKENLER

`u = 2` `anahtar = [0, 3, 2, 4, 2]` `ata = [-, 1, 1, 3, 2]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (agacta[u]) continue;  // eski
▶      agacta[u] = true;
▶      for (Kenar k : komsu.get(u))
▶        if (!agacta[k.v] && k.w < anahtar[k.v]) {
▶          anahtar[k.v] = k.w; ata[k.v] = u;
▶          pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



pq | (4,4) | (4,6) |

(5,2) en küçük kayıt: 5 ağaca girer (kenar 2–5, ağırlık 2). Ağaçta olmayan hiçbir komşusunun anahtarı küçülmedi.

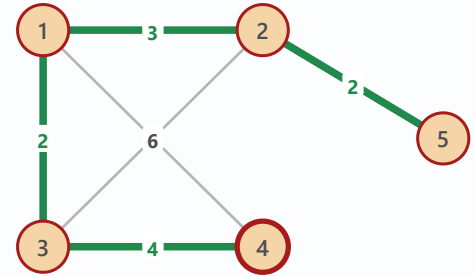
Adım 5 / 8

DEĞİŞKENLER

`u = 5` `anahtar = [0, 3, 2, 4, 2]` `ata = [-, 1, 1, 3, 2]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (agacta[u]) continue;  // eski
▶      agacta[u] = true;
▶      for (Kenar k : komsu.get(u))
▶        if (!agacta[k.v] && k.w < anahtar[k.v]) {
▶          anahtar[k.v] = k.w; ata[k.v] = u;
▶          pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



(4,4) en küçük kayıt: 4 ağaca girer (kenar 3–4, ağırlık 4). Ağaçta olmayan hiçbir komşusunun anahtarı küçülmedi.

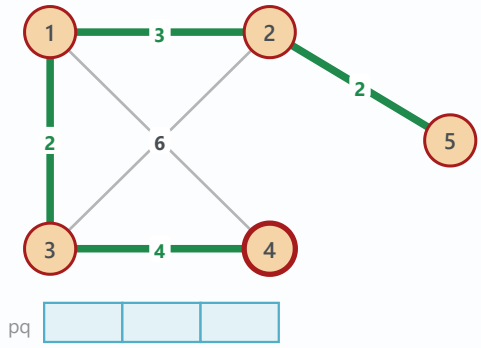
Adım 6 / 8

DEĞİŞKENLER

`u = 4` `anahtar = [0, 3, 2, 4, 2]` `ata = [-, 1, 1, 3, 2]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
7    while (!pq.isEmpty()) {
▶      int u = pq.poll()[0];
▶      if (agacta[u]) continue;  // eski
10     agacta[u] = true;
11     for (Kenar k : komsu.get(u))
12       if (!agacta[k.v] && k.w < anahtar[k.v]) {
13         anahtar[k.v] = k.w; ata[k.v] = u;
14         pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



(4,6) çıkarıldı ama 4 zaten ağaçta: bu **eski kayıt**, `continue` ile atlanır (tembel silme).

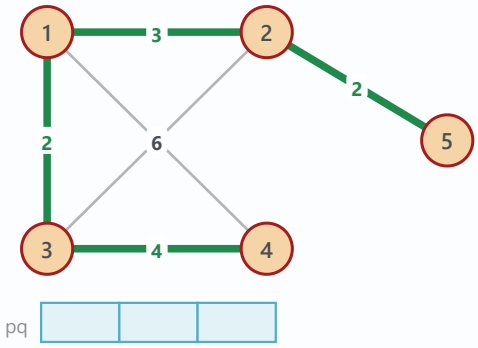
Adım 7 / 8

DEĞİŞKENLER

`u = 4` `anahtar = [0, 3, 2, 4, 2]` `ata = [-, 1, 1, 3, 2]`

Prim: Tembel Öncelikli Kuyruk Sürümünü İzle

```
1  void primPQ(int kaynak) {
2    Arrays.fill(anahtar, SONSUZ);
3    anahtar[kaynak] = 0;
4    PriorityQueue<int[]> pq =
5        new PriorityQueue<>((a, b) -> a[1] -
   b[1]);
6    pq.add(new int[] { kaynak, 0 });
▶    while (!pq.isEmpty()) {
8      int u = pq.poll()[0];
9      if (agacta[u]) continue;  // eski
10     agacta[u] = true;
11     for (Kenar k : komsu.get(u))
12       if (!agacta[k.v] && k.w < anahtar[k.v]) {
13         anahtar[k.v] = k.w; ata[k.v] = u;
14         pq.add(new int[] { k.v, k.w });
15       }
16   }
17 }
```



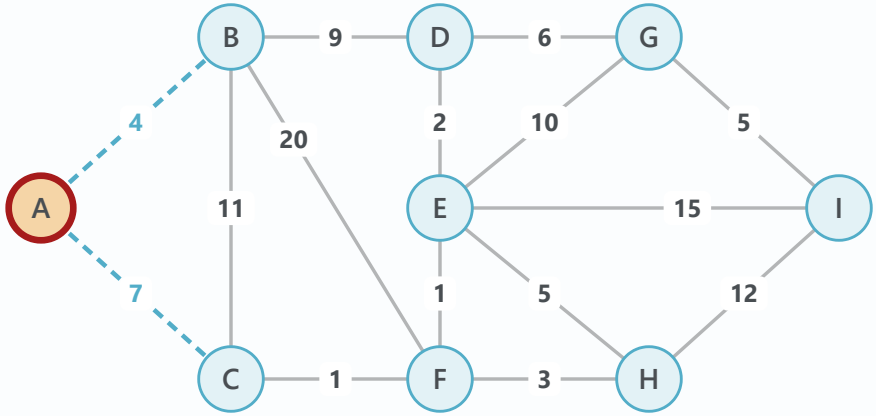
Kuyruk boşaldı. MST kenarları (ata[v], v): 1–3 (2), 1–2 (3), 2–5 (2), 3–4 (4); toplam **11**.

Adım 8 / 8

DEĞİŞKENLER

`u = –` `anahtar = [0, 3, 2, 4, 2]` `ata = [–, 1, 1, 3, 2]`

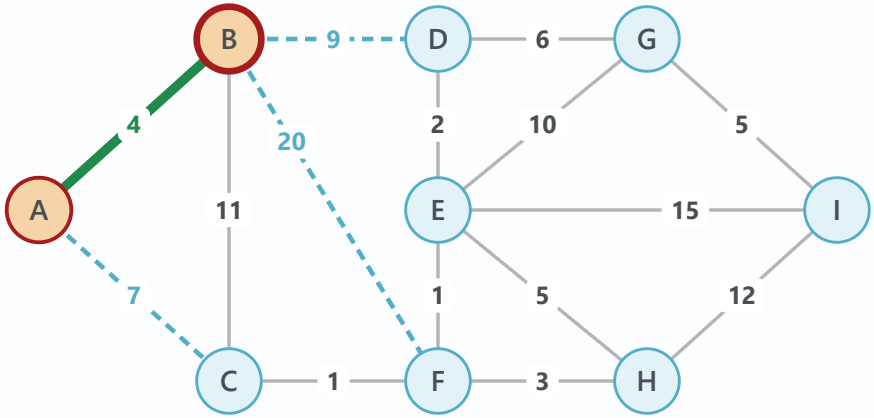
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	∞	∞	∞	∞	∞	∞
ebeveyn	–	A	A	–	–	–	–	–	–

Başlangıç **A**: komşuları B (4) ve C (7) aday olur.

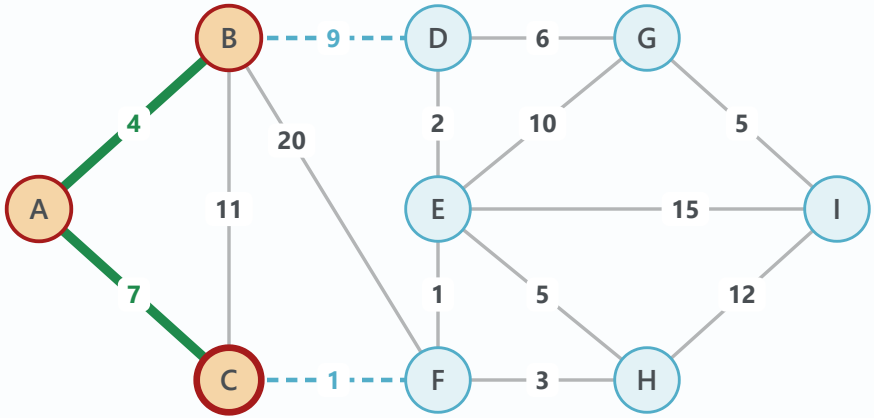
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	9	∞	20	∞	∞	∞
ebeveyn	–	A	A	B	–	B	–	–	–

B ağaca girer: (A,B) = 4, toplam 4. Güncellenen: D → 9, F → 20.

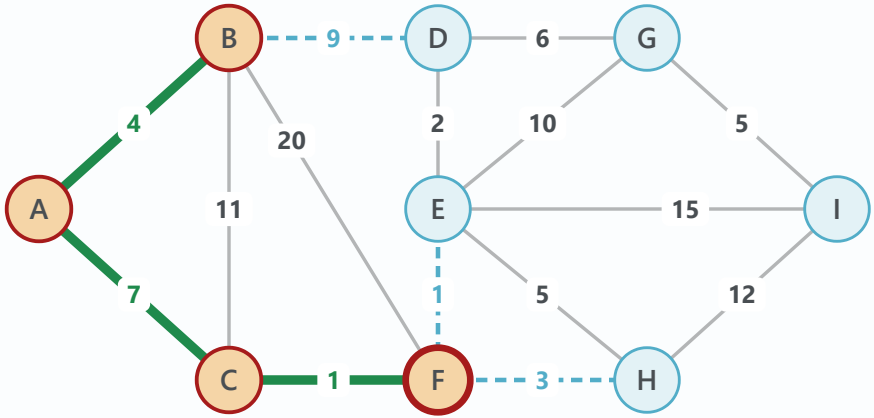
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	9	∞	1	∞	∞	∞
ebeveyn	–	A	A	B	–	C	–	–	–

C ağaca girer: (A,C) = 7, toplam 11. Güncellenen: F → 1.

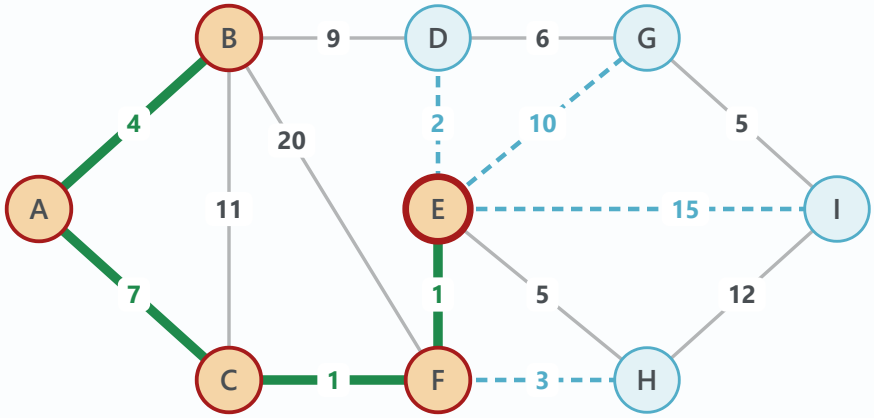
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	9	1	1	∞	3	∞
ebeveyn	–	A	A	B	F	C	–	F	–

F ağaca girer: (C,F) = 1, toplam 12. Güncellenen: E → 1, H → 3.

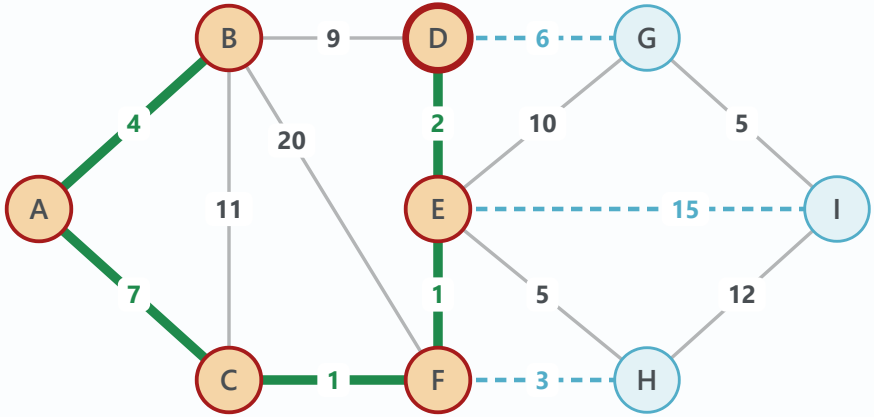
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	2	1	1	10	3	15
ebeveyn	–	A	A	E	F	C	E	F	E

E ağaca girer: (F,E) = 1, toplam 13. Güncellenen: D → 2, G → 10, I → 15.

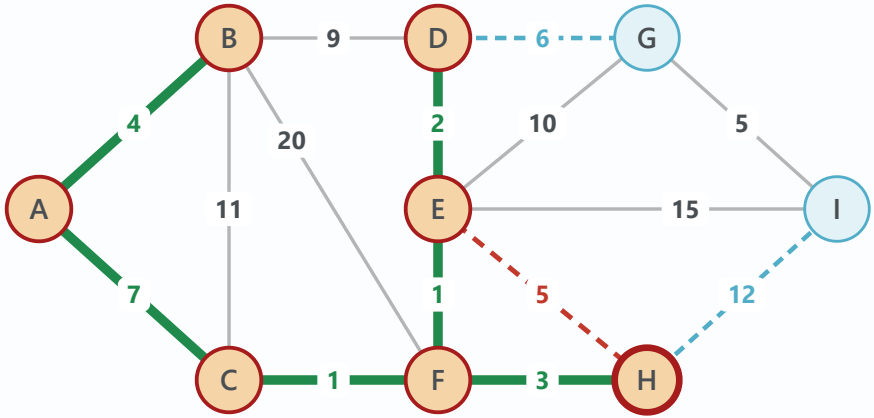
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	2	1	1	6	3	15
ebeveyn	–	A	A	E	F	C	D	F	E

D ağaca girer: (E,D) = 2, toplam 15. Güncellenen: G → 6.

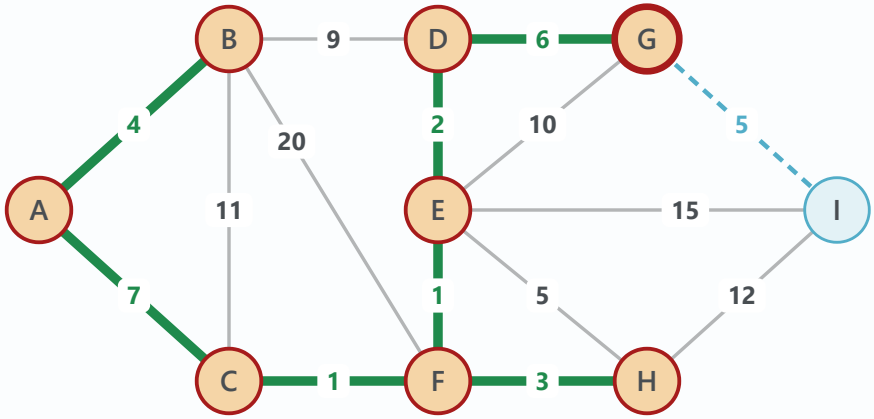
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	2	1	1	6	3	12
ebeveyn	–	A	A	E	F	C	D	F	H

H ağaca girer: (F,H) = 3, toplam 18. Güncellenen: I → 12. EH = 5 artık kullanılamaz: E ve H ikisi de ağaçta (döngü).

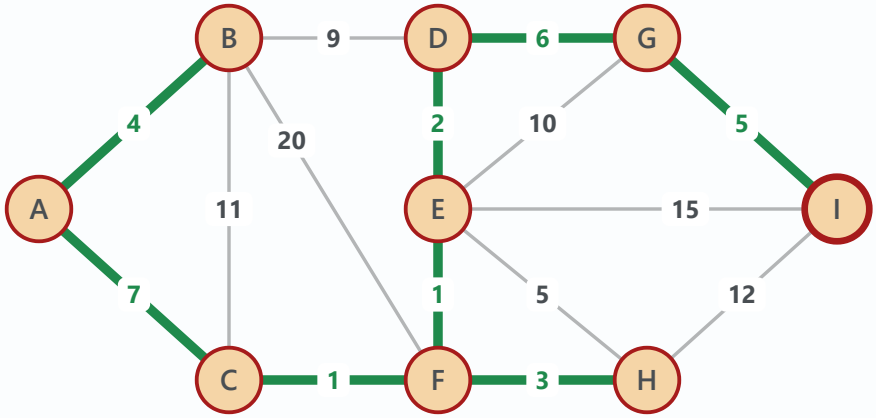
Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	2	1	1	6	3	5
ebeveyn	–	A	A	E	F	C	D	F	G

G ağaca girer: (D,G) = 6, toplam 24. Güncellenen: I → 5.

Prim Örneği: Dokuz Düğüm (A'dan Başla)



	A	B	C	D	E	F	G	H	I
anahtar	0	4	7	2	1	1	6	3	5
ebeveyn	–	A	A	E	F	C	D	F	G

I ağaca girer: (G,I) = 5, toplam 29. Tüm düğümler ağaçta: **MST = 29** (Kruskal ile aynı).

Prim: Karmaşıklık Analizi

Uygulama	en küçüğü çıkar	anahtar azalt	Toplam
Komşuluk matrisi + dizi taraması (yukarıdaki kod)	V × O(V)	O(1)	**$O(V^2)$**
Komşuluk listesi + ikili heap (binary heap)	V × O(log V)	E × O(log V)	**O((V + E) log V)**
Tembel sürüm (sözde kod): kuyrukta kenar kopyaları	≤ E ekleme/çıkarma, her biri O(log E)		O(E log E)
Fibonacci heap	O(log V) amortize	O(1) amortize	O(E + V log V)

Yoğun çizge ($E \approx V^2$)

Matris sürümü $O(V^2)$ en iyisidir; heap sürümü $O(V^2 \log V)$ olur.

Seyrek çizge ($E \approx V$)

Heap sürümü O(V log V); matris sürümü gereksiz yere $O(V^2)$.

Prim ile Dijkstra Arasındaki Fark

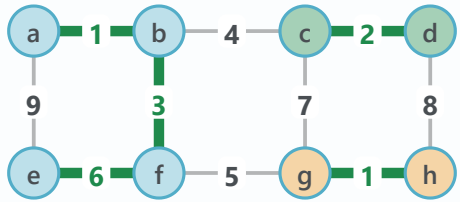
İki algoritmanın kodu neredeyse aynıdır: bir kaynaktan başlar, öncelikli kuyruktan en küçük anahtarlı düğümü çıkarır ve komşuları gevşetir. Tek fark **anahtarın anlamıdır**.

	Prim (MST)	**Dijkstra (en kısa yol)**
anahtar[v]	v'yi *ağaca* bağlayan en ucuz **tek kenar**	kaynaktan v'ye **toplam yol** uzunluğu
Güncelleme	`w(u,v) < anahtar[v]`	`uzaklik[u] + w(u,v) < uzaklik[v]`
Sonuç	toplam ağırlığı en küçük ağaç	en kısa yollar ağacı (SPT)
Kaynak	sonucu etkilemez (aynı ağırlık)	sonucu belirler
Negatif ağırlık	sorun değil	çalışmaz

Beş düğümlü örnekte 1'den en kısa yollar ağacı (1,4) = 6 kenarını kullanabilir; MST onun yerine (3,4) = 4'ü seçer.

Borůvka (Sollin) Algoritması

- Minimum kapsayan ağacı bulan **en eski** algoritma: Otakar **Borůvka**, 1926 (Moravya elektrik ağı). Diğer adı **Sollin** algoritması.
- Başlangıçta her düğüm **bağımsız bir bileşendir**.
- Her turda **her bileşen** kendisini dışarıya bağlayan **en ucuz kenarı** seçer; seçilenlerin hepsi birden eklenir.
- Bileşenler **union-find** ile birleştirilir; tek bileşen kalana kadar devam edilir. Bağlı çizgede $|V| - 1$ kenar üretir.



tur 1: her düğüm kendi en ucuz kenarını seçer → 8 bileşen 3 bileşene iner

Güçlü yönler

+ Bileşenlerin seçimleri birbirinden bağımsız: **paralel** çalışmaya çok uygun.
+ Tur sayısı en fazla $\lceil \log_2 V \rceil$.

Zayıf yönler

– Kruskal/Prim'den daha karmaşık kodlanır.

– Eşit ağırlıklarda **tutarlı bir eşitlik bozucu** (ör. kenar indisi) şart; yoksa döngü oluşabilir.

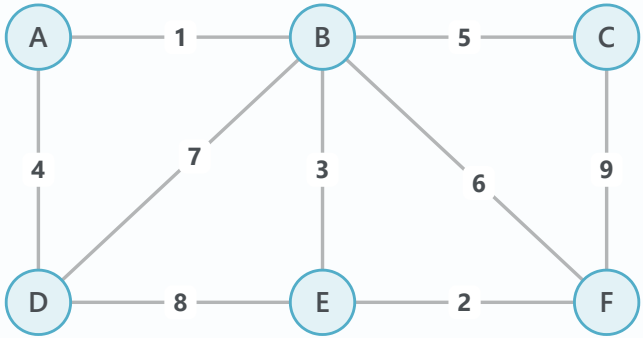
Borůvka: Adımlar ve Sözde Kod

1. Başlangıçta her düğüm tek başına bir bileşen (küme) sayılır.
2. Her bileşen için, onu **başka bir bileşene** bağlayan en küçük ağırlıklı kenar bulunur.
3. Bulunan kenarların hepsi eklenir; bağladıkları bileşenler birleştirilir (aynı kenar iki kez eklenmez).
4. Tek bileşen kalana kadar 2. ve 3. adımlar tekrarlanır.

Not: Borůvka kenarları **sıralamaz**; her turda tüm kenarları bir kez tarayıp bileşen başına en küçüğü tutar.

```
BORUVKA(G):
  MST = [ ]
  bileşenSayısı = |V|
  döngü bileşenSayısı > 1 iken:
    enUcuz = [ ]  // bileşen başına
    her bir e = (u, v, ağırlık) için E içinde:
      cu = ayrikKume.bul(u)
      cv = ayrikKume.bul(v)
      eğer cu ≠ cv:
        enUcuz[cu] = min(enUcuz[cu], e)
        enUcuz[cv] = min(enUcuz[cv], e)
    her bir (u, v, ağırlık) için enUcuz içinde:
      eğer ayrikKume.bul(u) ≠ ayrikKume.bul(v):
        MST.ekle((u, v, ağırlık))
        ayrikKume.birleştir(u, v)
        bileşenSayısı -= 1
  döndür MST
```

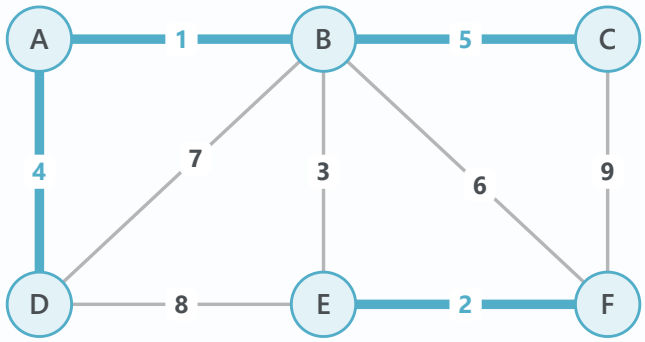
Borůvka Adım Adım: Altı Düğüm



Başlangıç: her düğüm ayrı bir bileşen (6 bileşen).

Kruskal kenarları 1, 2, 3, 4, 5 sırasıyla tek tek ekler; Borůvka'da ise tur 1'de dört kenar **aynı anda** eklenir. Sonuç ağacı aynıdır.

Borůvka Adım Adım: Altı Düğüm

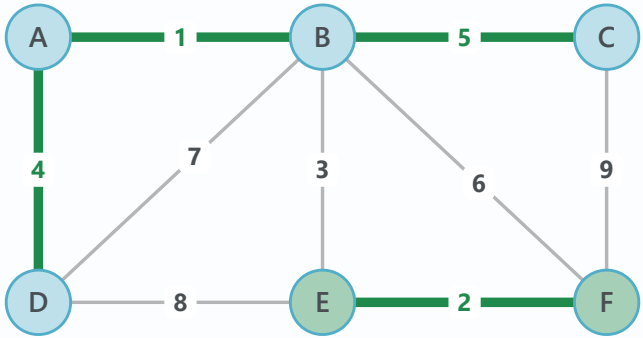


Bileşen	En ucuz
{A}	AB 1
{B}	AB 1
{C}	BC 5
{D}	AD 4
{E}	EF 2
{F}	EF 2

Tur 1: her bileşen kendi en ucuz dış kenarını seçer (mavi). AB ve EF iki bileşen tarafından seçilir ama birer kez eklenir.

Kruskal kenarları 1, 2, 3, 4, 5 sırasıyla tek tek ekler; Borůvka'da ise tur 1'de dört kenar **aynı anda** eklenir. Sonuç ağacı aynıdır.

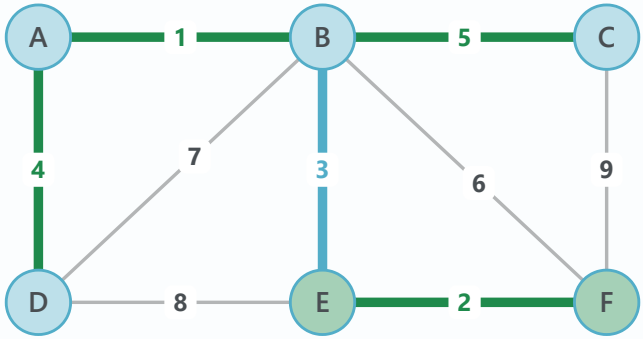
Borůvka Adım Adım: Altı Düğüm



Tur 1 sonu: 4 farklı kenar eklendi (AB, BC, AD, EF), 2 bileşen kaldı: {A, B, C, D} {E, F}. Toplam 12.

Kruskal kenarları 1, 2, 3, 4, 5 sırasıyla tek tek ekler; Borůvka'da ise tur 1'de dört kenar **aynı anda** eklenir. Sonuç ağacı aynıdır.

Borůvka Adım Adım: Altı Düğüm

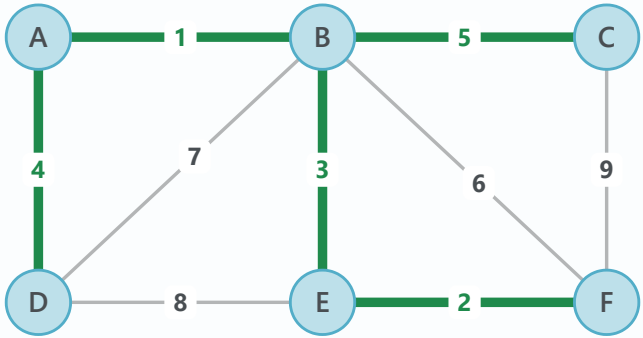


Bileşen	En ucuz
{A,B,C,D}	BE 3
{E,F}	BE 3

Tur 2: her bileşen kendi en ucuz dış kenarını seçer (mavi). İki bileşen de aynı kenarı seçer.

Kruskal kenarları 1, 2, 3, 4, 5 sırasıyla tek tek ekler; Borůvka'da ise tur 1'de dört kenar **aynı anda** eklenir. Sonuç ağacı aynıdır.

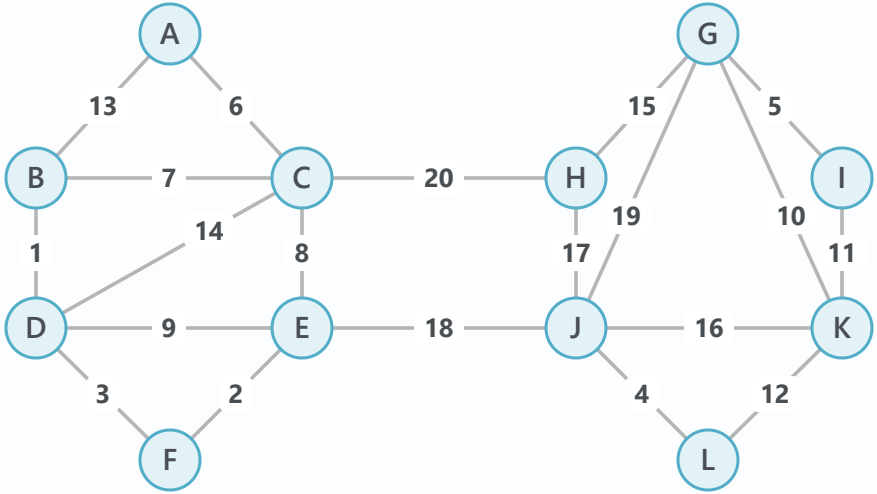
Borůvka Adım Adım: Altı Düğüm



Tur 2 sonu: 1 farklı kenar eklendi (BE), 1 bileşen kaldı: {A, B, C, D, E, F}. Toplam 15. **MST = 15** (AB, BC, AD, EF, BE).

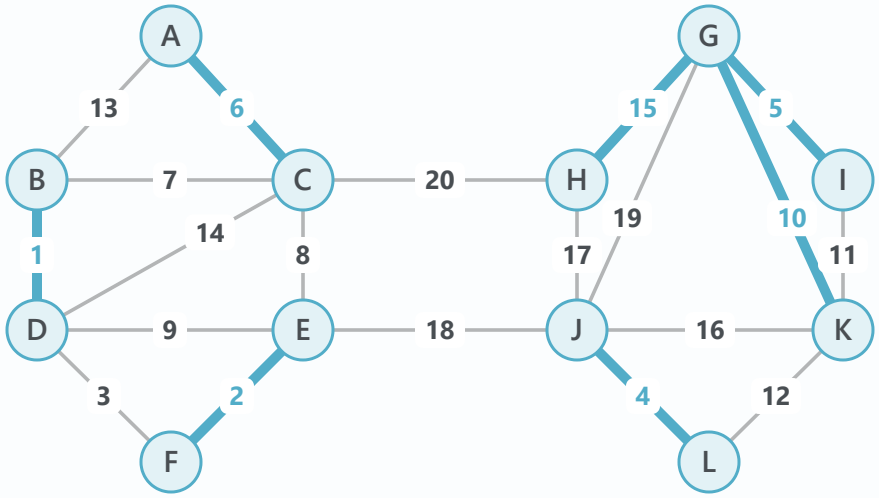
Kruskal kenarları 1, 2, 3, 4, 5 sırasıyla tek tek ekler; Borůvka'da ise tur 1'de dört kenar **aynı anda** eklenir. Sonuç ağacı aynıdır.

Borůvka Örneği: On İki Düğüm



Başlangıç: her düğüm ayrı bir bileşen (12 bileşen).

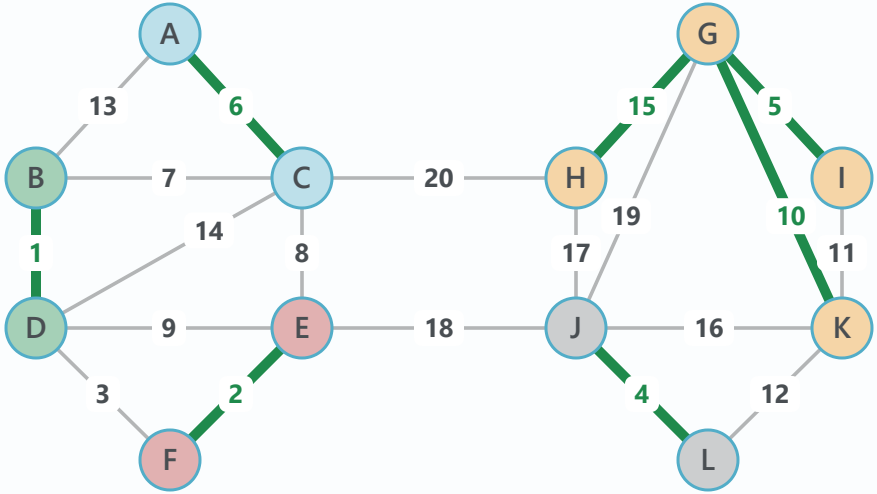
Borůvka Örneği: On İki Düğüm



Bileşen	En ucuz
{A}	AC 6
{B}	BD 1
{C}	AC 6
{D}	BD 1
{E}	EF 2
{F}	EF 2
{G}	GI 5
{H}	GH 15
{I}	GI 5
{J}	JL 4
{K}	GK 10
{L}	JL 4

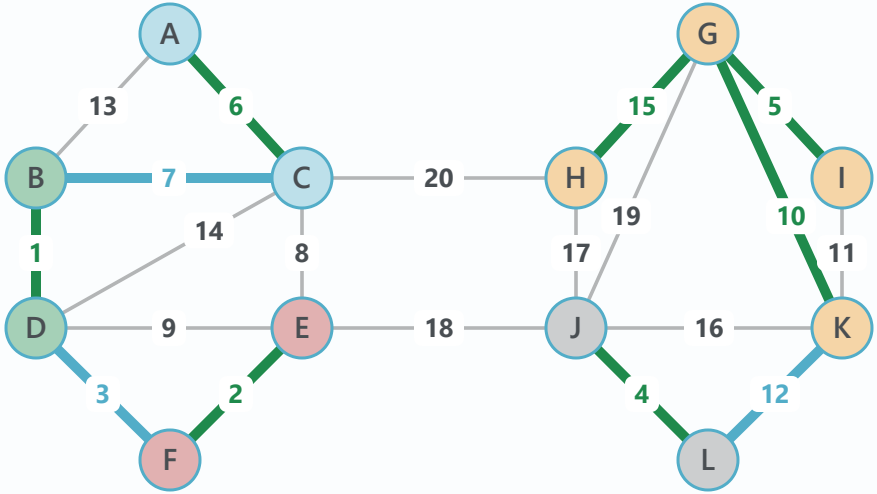
Tur 1: her bileşenin en ucuz dış kenarı (mavi).

Borůvka Örneği: On İki Düğüm



Tur 1 sonu: 7 farklı kenar eklendi (AC, BD, EF, GI, GH, JL, GK), 5 bileşen kaldı: {G, H, I, K} {A, C} {B, D} {E, F} {J, L}. Toplam 43.

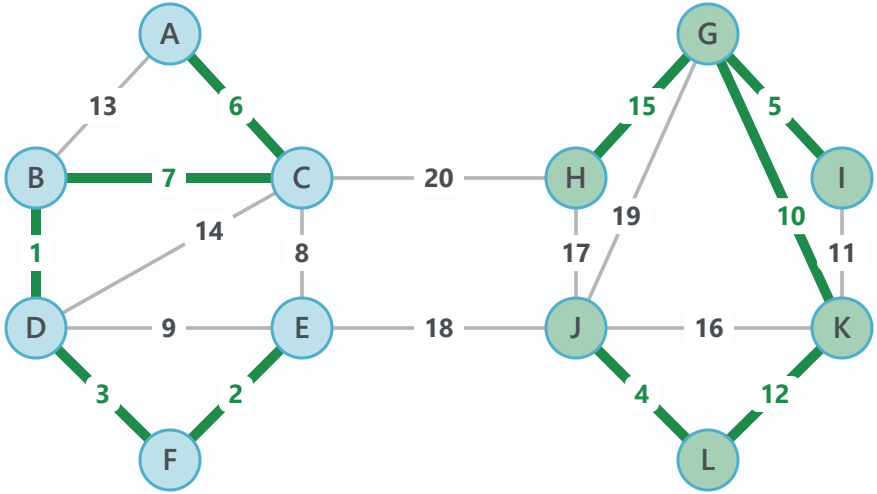
Borůvka Örneği: On İki Düğüm



Bileşen	En ucuz
{A,C}	BC 7
{B,D}	DF 3
{E,F}	DF 3
{G,H,I,K}	KL 12
{J,L}	KL 12

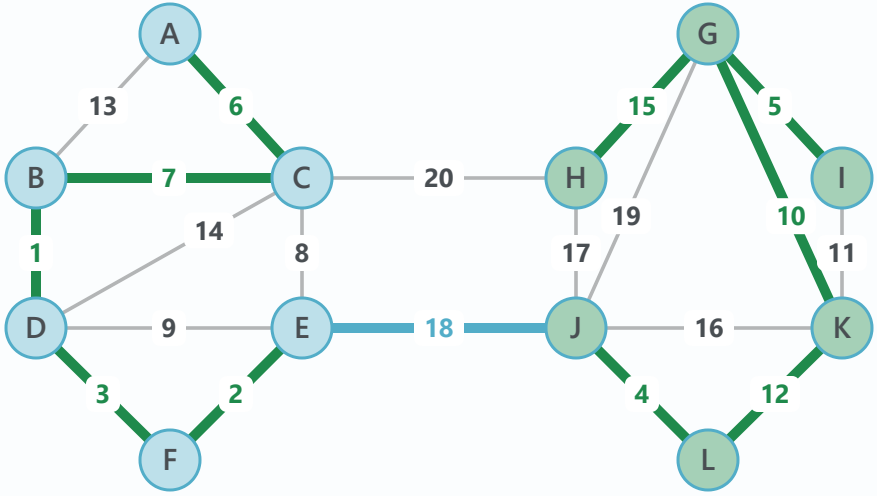
Tur 2: her bileşenin en ucuz dış kenarı (mavi).

Borůvka Örneği: On İki Düğüm



Tur 2 sonu: 3 farklı kenar eklendi (BC, DF, KL), 2 bileşen kaldı: {A, B, C, D, E, F} {G, H, I, J, K, L}. Toplam 65.

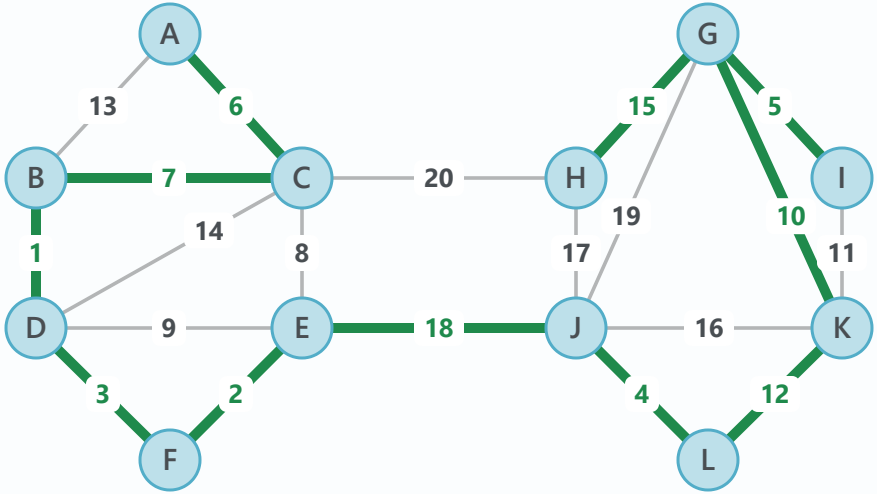
Borůvka Örneği: On İki Düğüm



Bileşen	En ucuz
{A,B,C,D,E,F}	EJ 18
{G,H,I,J,K,L}	EJ 18

Tur 3: her bileşenin en ucuz dış kenarı (mavi).

Borůvka Örneği: On İki Düğüm



Tur 3 sonu: 1 farklı kenar eklendi (EJ), 1 bileşen kaldı: {A, B, C, D, E, F, G, H, I, J, K, L}. **MST = 83**, yalnızca 3 turda.

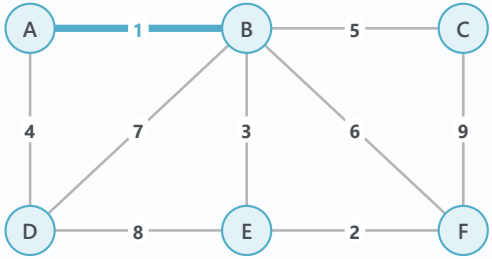
Borůvka: Java Kodu ve Karmaşıklık

```
while (bilesenSayisi > 1) {
  Kenar[] enUcuz = new Kenar[V];
  for (Kenar e : kenarlar) {          // 1) en ucuzlar
    int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
    if (k1 == k2) continue;           // aynı bileşen
    if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
      enUcuz[k1] = e;
    if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
      enUcuz[k2] = e;
  }
  for (Kenar e : enUcuz) {            // 2) hepsini ekle
    if (e == null) continue;
    int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
    if (k1 == k2) continue;           // zaten eklendi
    agac.add(e);
    ata[k1] = k2;                     // birleştir
    bilesenSayisi--;
  }
}
```

- Her turda her bileşen **en az bir** başka bileşenle birleşir → bileşen sayısı en az **yarıya** iner.
- Tur sayısı ≤ $\lceil \log_2 V \rceil$; her tur tüm kenarları bir kez tarar: O(E).
- Toplam: **O(E log V)**. Sıralama yapılmaz; "ağırlık sıralaması" gerekmez.
- `e.w <` karşılaştırması eşitlikte ilk (küçük indisli) kenarı tutar: bu **tutarlı eşitlik bozucu** döngüyü önler.

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
▶      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
▶        enUcuz[k1] = e;
▶      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
▶        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
13     int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;            // birleştir
17     bilesenSayisi--;
18   }
19 }
```



Altı bileşen (her düğüm tek başına); kenarlar listedeki sırayla taranır. Kenar AB(1): enUcuz[A] ve enUcuz[B] = AB(1).

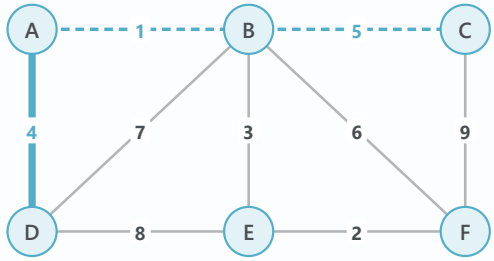
Adım 1 / 14

DEĞİŞKENLER

`e = AB(1)` `ata = [A, B, C, D, E, F]` `enUcuz = [AB, AB, –, –, –, –]` `bilesenSayisi = 6`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
▶      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
▶        enUcuz[k1] = e;
▶      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
▶        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
13     int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;            // birleştir
17     bilesenSayisi--;
18   }
19 }
```



Kenar BC(5): enUcuz[C] = BC(5). Kenar AD(4): enUcuz[D] = AD(4).

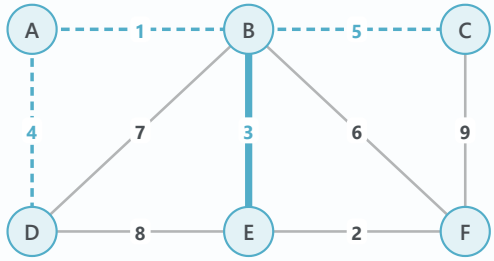
Adım 2 / 14

DEĞİŞKENLER

`e = AD(4)` `ata = [A, B, C, D, E, F]` `enUcuz = [AB, AB, BC, AD, –, –]` `bilesenSayisi = 6`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
▶      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
▶        enUcuz[k1] = e;
▶      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
▶        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
13     int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;            // birleştir
17     bilesenSayisi--;
18   }
19 }
```



Kenar BE(3): enUcuz[E] = BE(3).

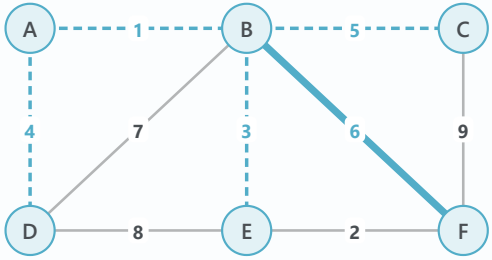
Adım 3 / 14

DEĞİŞKENLER

`e = BE(3)` `ata = [A, B, C, D, E, F]` `enUcuz = [AB, AB, BC, AD, BE, –]` `bilesenSayisi = 6`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
▶      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
▶        enUcuz[k1] = e;
▶      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
▶        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
13     int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;            // birleştir
17     bilesenSayisi--;
18   }
19 }
```



BD(7) hiçbir bileşenin en ucuzunu geçemedi. Kenar BF(6): enUcuz[F] = BF(6).

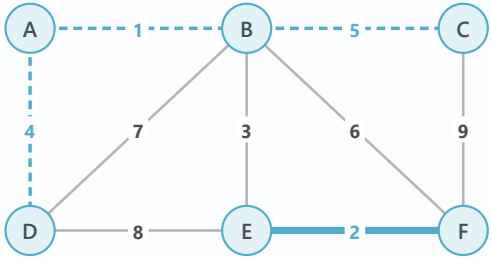
Adım 4 / 14

DEĞİŞKENLER

`e = BF(6)` `ata = [A, B, C, D, E, F]` `enUcuz = [AB, AB, BC, AD, BE, BF]` `bilesenSayisi = 6`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
▶      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
▶        enUcuz[k1] = e;
▶      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
▶        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
13     int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;            // birleştir
17     bilesenSayisi--;
18   }
19 }
```



CF(9) ve DE(8) daha ucuzların gerisinde kaldı. Kenar EF(2): enUcuz[E] ve enUcuz[F] = EF(2). E için 3'ten, F için 6'dan ucuz.

Adım 5 / 14

DEĞİŞKENLER

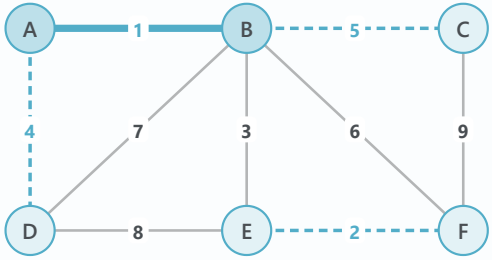
`e = EF(2)` `ata = [A, B, C, D, E, F]`

`enUcuz = [AB, AB, BC, AD, EF, EF]`

`bilesenSayisi = 6`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
▶      agac.add(e);
▶      ata[k1] = k2;           // birleştir
▶      bilesenSayisi--;
18   }
19 }
```



AB(1) ağaca eklenir; bul ile kökler A ve B → ata[A] = B. Bileşen sayısı 5.

Adım 6 / 14

DEĞİŞKENLER

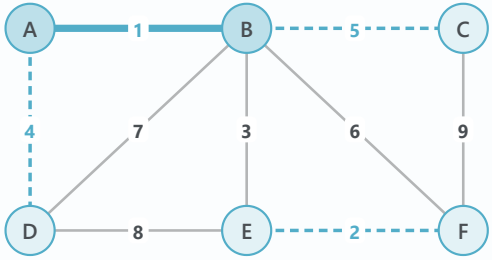
`e = AB(1)` `ata = [B, B, C, D, E, F]`

`enUcuz = [AB, AB, BC, AD, EF, EF]`

`bilesenSayisi = 5`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
▶      if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;           // birleştir
17     bilesenSayisi--;
18   }
19 }
```



enUcuz[B] = AB(1) ama iki ucu artık aynı bileşende: zaten eklendi, atlanır.

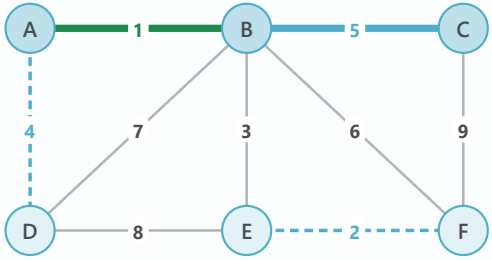
Adım 7 / 14

DEĞİŞKENLER

`e = AB(1)` `ata = [B, B, C, D, E, F]` `enUcuz = [AB, AB, BC, AD, EF, EF]` `bilesenSayisi = 5`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
▶      agac.add(e);
▶      ata[k1] = k2;            // birleştir
▶      bilesenSayisi--;
18   }
19 }
```



BC(5) ağaca eklenir; bul ile kökler B ve C → ata[B] = C. Bileşen sayısı 4.

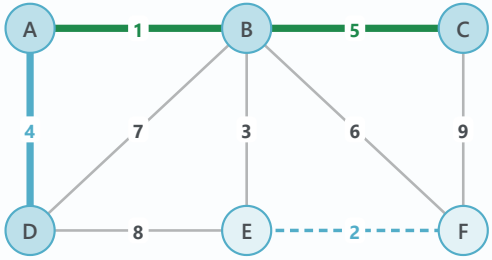
Adım 8 / 14

DEĞİŞKENLER

`e = BC(5)` `ata = [B, C, C, D, E, F]` `enUcuz = [AB, AB, BC, AD, EF, EF]` `bilesenSayisi = 4`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
▶      agac.add(e);
▶      ata[k1] = k2;            // birleştir
▶      bilesenSayisi--;
18   }
19 }
```



AD(4) ağaca eklenir; bul ile kökler C ve D → ata[C] = D. Bileşen sayısı 3.

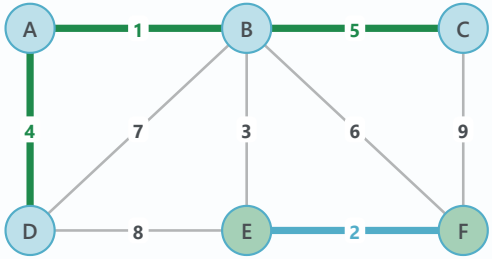
Adım 9 / 14

DEĞİŞKENLER

`e = AD(4)` `ata = [C, C, D, D, E, F]` `enUcuz = [AB, AB, BC, AD, EF, EF]` `bilesenSayisi = 3`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
▶      agac.add(e);
▶      ata[k1] = k2;            // birleştir
▶      bilesenSayisi--;
18   }
19 }
```



EF(2) ağaca eklenir; bul ile kökler E ve F → ata[E] = F. Bileşen sayısı 2.

Adım 10 / 14

DEĞİŞKENLER

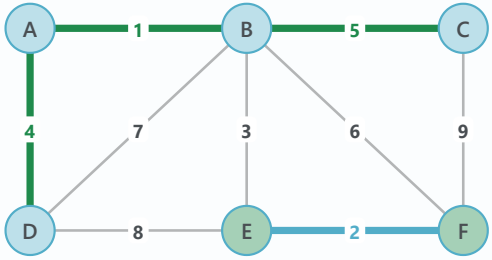
`e = EF(2)` `ata = [C, C, D, D, F, F]`

`enUcuz = [AB, AB, BC, AD, EF, EF]`

`bilesenSayisi = 2`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
▶      if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;           // birleştir
17     bilesenSayisi--;
18   }
19 }
```



enUcuz[F] = EF(2) ama iki ucu artık aynı bileşende: zaten eklendi, atlanır.

Adım 11 / 14

DEĞİŞKENLER

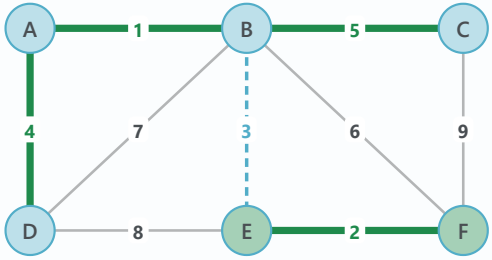
`e = EF(2)` `ata = [C, C, D, D, F, F]`

`enUcuz = [AB, AB, BC, AD, EF, EF]`

`bilesenSayisi = 2`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
▶    Kenar[] enUcuz = new Kenar[V];
▶    for (Kenar e : kenarlar) {      // 1) en ucuzlar
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
▶      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
▶        enUcuz[k1] = e;
▶      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
▶        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
13     int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;            // birleştir
17     bilesenSayisi--;
18   }
19 }
```



Tur 2: AB, BC, AD, BD, EF bileşen içinde kalır (continue). Bileşenler arası BE(3), BF(6), CF(9), DE(8) içinden en ucuzu BE(3): iki kökün enUcuz'u da BE.

Adım 12 / 14

DEĞİŞKENLER

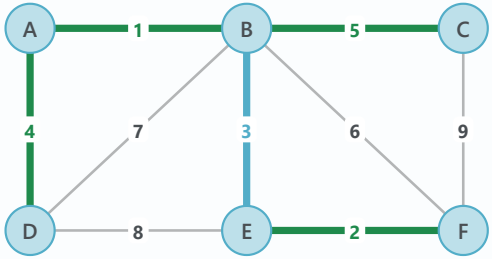
`e = –` `ata = [D, D, D, D, F, F]`

`enUcuz = [–, –, –, BE, –, BE]`

`bilesenSayisi = 2`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
12     if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
14     if (k1 == k2) continue;
▶      agac.add(e);
▶      ata[k1] = k2;            // birleştir
▶      bilesenSayisi--;
18   }
19 }
```



BE(3) ağaca eklenir; bul ile kökler D ve F → ata[D] = F. Bileşen sayısı 1. Tek bileşen kaldı.

Adım 13 / 14

DEĞİŞKENLER

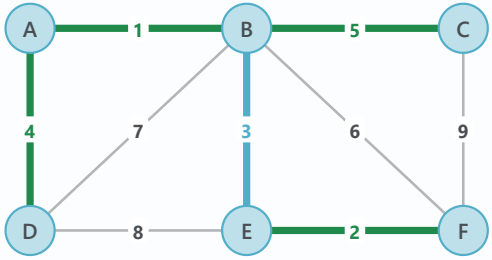
`e = BE(3)` `ata = [D, D, D, F, F, F]`

`enUcuz = [–, –, –, BE, –, BE]`

`bilesenSayisi = 1`

Borůvka: Kodu Adım Adım İzle

```
1  while (bilesenSayisi > 1) {
2    Kenar[] enUcuz = new Kenar[V];
3    for (Kenar e : kenarlar) {      // 1) en ucuzlar
4      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
5      if (k1 == k2) continue;
6      if (enUcuz[k1] == null || e.w < enUcuz[k1].w)
7        enUcuz[k1] = e;
8      if (enUcuz[k2] == null || e.w < enUcuz[k2].w)
9        enUcuz[k2] = e;
10   }
11   for (Kenar e : enUcuz) {        // 2) hepsini ekle
▶      if (e == null) continue;
▶      int k1 = bul(ata, e.u), k2 = bul(ata, e.v);
▶      if (k1 == k2) continue;
15     agac.add(e);
16     ata[k1] = k2;           // birleştir
17     bilesenSayisi--;
18   }
19 }
```



enUcuz[F] = BE(3) ama iki ucu artık aynı bileşende: zaten eklendi, atlanır. `bilesenSayisi` = 1: döngü biter, MST ağırlığı **15**.

Adım 14 / 14

DEĞİŞKENLER

`e = BE(3)` `ata = [D, F, D, F, F, F]`

`enUcuz = [–, –, –, BE, –, BE]`

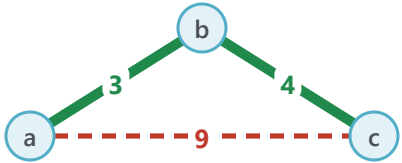
`bilesenSayisi = 1`

Ters Silme (Reverse-Delete) Algoritması

- Minimum kapsayan ağacı bulan **açgözlü** bir algoritmadır (Kruskal, 1956 makalesinde önerdi).
- Kenarları ağırlıklarına göre **azalan** şekilde sıralar.
- En ağır kenardan başlayarak, çizgenin **bağlılığını bozmayan** her kenarı siler.
- **Döngü özelliğine** dayanır: silinen kenar, bir döngünün en ağır kenarıdır.

Kruskal'ın aynası

Kruskal **küçük** kenarları **eklerken**, ters silme **büyük** kenarları **silmeye** çalışır. Biri boş çizgeden, öteki tam çizgeden başlar.



Kullanım

Daha çok **teorik/öğretici** değeri vardır; her silmede bağlılık testi gerektiği için pratikte genellikle yavaştır.

Ters Silme: Adımlar, Sözde Kod, Karmaşıklık

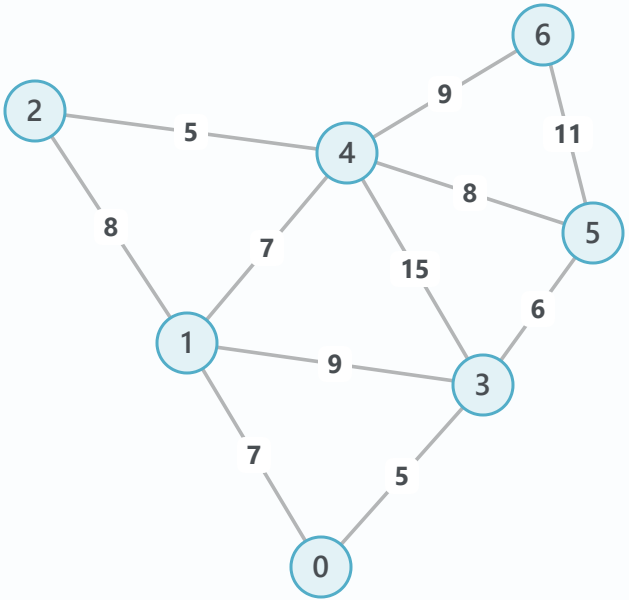
1. Tüm kenarlar ağırlığına göre **azalan** şekilde sıralanır.
2. Sıradaki en ağır kenar silinir.
3. Çizge artık **bağlı değilse** silinen kenar geri eklenir.
4. Tüm kenarlar işlenene kadar 2. ve 3. adımlar tekrarlanır.

Karmaşıklık: her kenar için bir BFS/DFS (ya da union-find ile yeniden kurma) → **O(E · (V + E))**. Sıralamanın O(E log E) maliyeti bunun yanında küçük kalır.

```
TERS_SİLME(G):
  MST = tüm kenarlar
  kenarlar.sırala(azalana göre)
  her bir (u, v, ağırlık) için kenarlar içinde:
    MST.sil((u, v, ağırlık))
    ayrikKume.sıfırla()
    her bir (x, y) için MST içinde:
      ayrikKume.birleştir(x, y)
    kök = ayrikKume.bul(V[0])
    eğer bir v için ayrikKume.bul(v) ≠ kök:
      MST.ekle((u, v, ağırlık))  // geri koy
  döndür MST
```

Bağlılık testi tüm düğümlere bakar; kökten kopuk **en az bir** düğüm varsa silinen kenar **bir kez** geri konur.

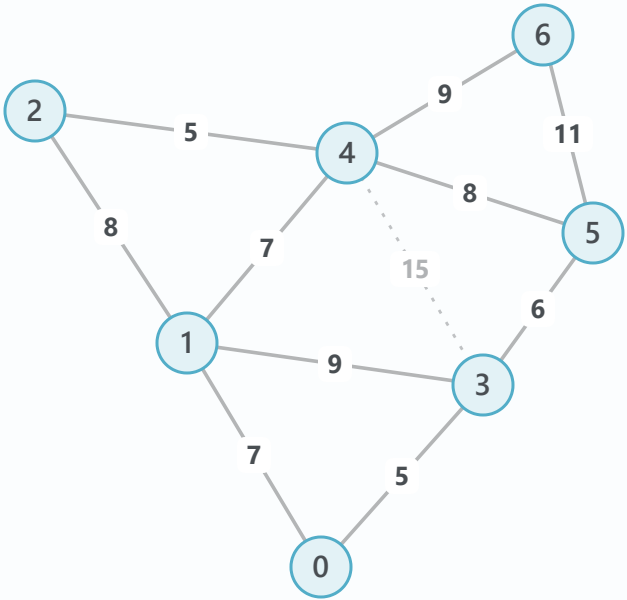
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
3–4	15	
5–6	11	
1–3	9	
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

Kenarlar azalan sırada: 15, 11, 9, 9, 8, 8, 7, 7, 6, 5, 5. Toplam ağırlık 90.

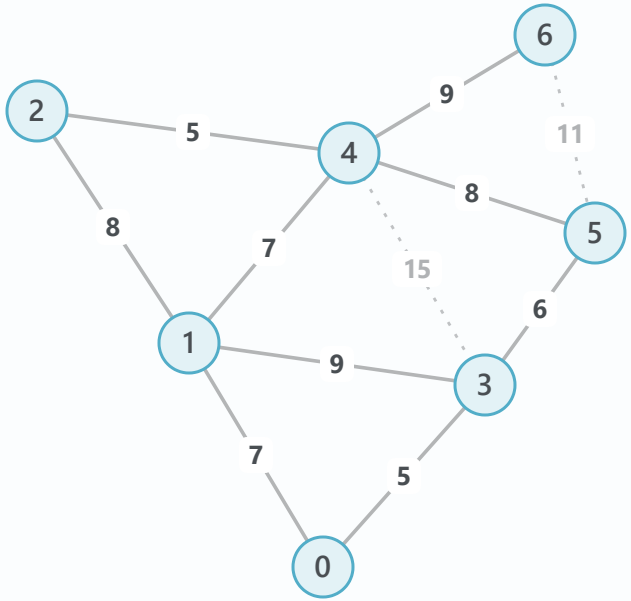
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
5–6	11	
1–3	9	
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

3–4 = 15 silinir: çizge hâlâ bağlı (kenar bir döngünün en ağırıdır).

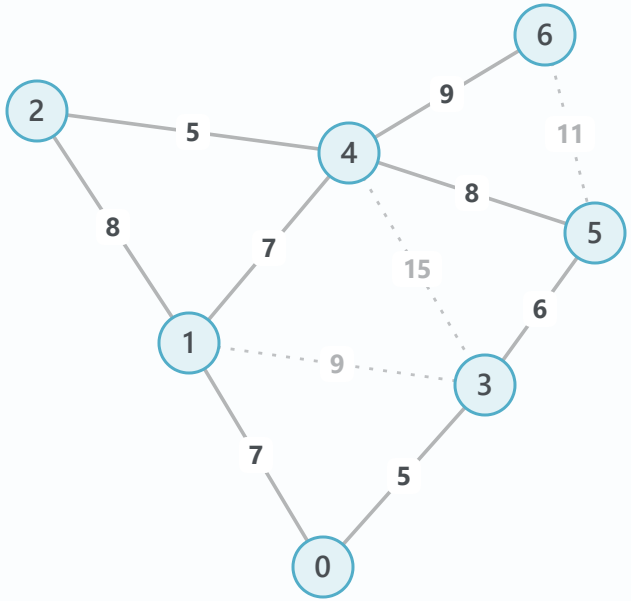
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
~~5–6~~	~~11~~	~~silindi~~
1–3	9	
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

5–6 = 11 silinir: çizge hâlâ bağlı (kenar bir döngünün en ağırıdır).

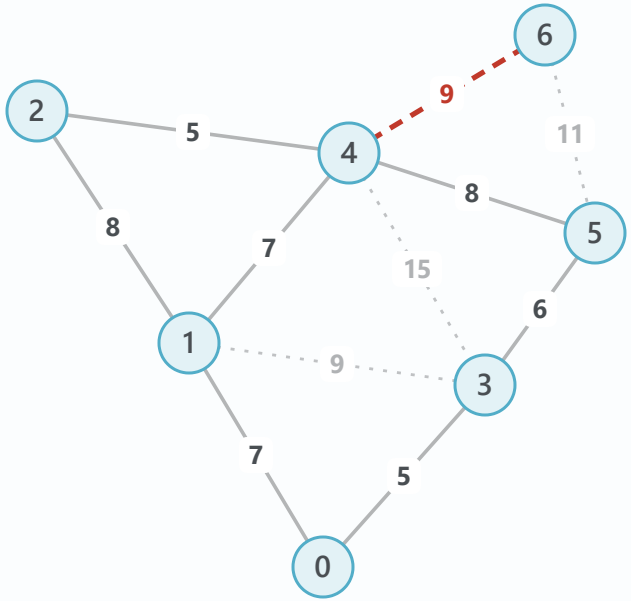
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
~~5–6~~	~~11~~	~~silindi~~
~~1–3~~	~~9~~	~~silindi~~
4–6	9	
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

1–3 = 9 silinir: çizge hâlâ bağlı (kenar bir döngünün en ağırıdır).

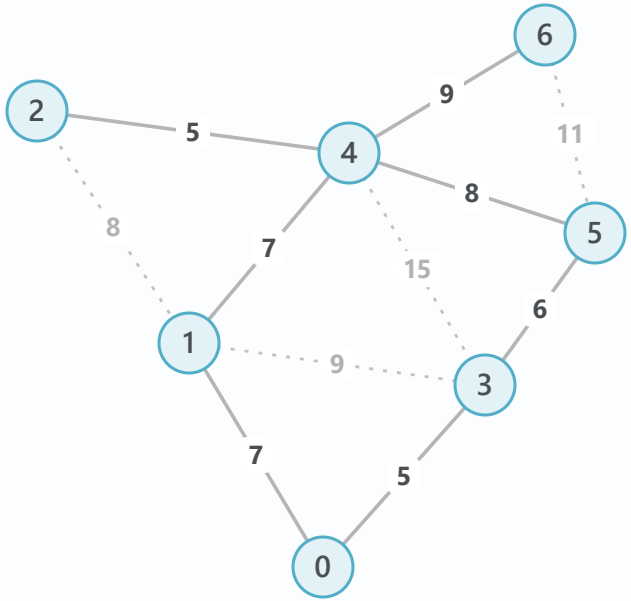
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
~~5–6~~	~~11~~	~~silindi~~
~~1–3~~	~~9~~	~~silindi~~
4–6	**9**	**kaldı**
1–2	8	
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

4–6 = 9 silinirse düğüm 6 yalnız kalır → bağlılık bozulur, kenar **geri eklenir**.

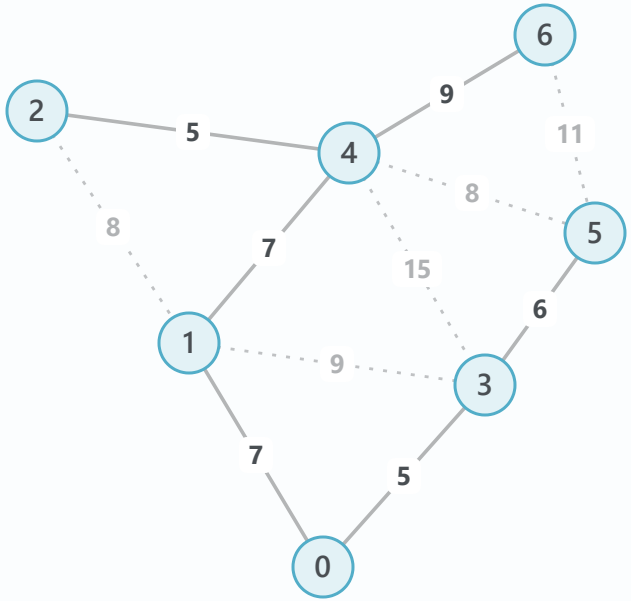
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
~~5–6~~	~~11~~	~~silindi~~
~~1–3~~	~~9~~	~~silindi~~
4–6	**9**	**kaldı**
~~1–2~~	~~8~~	~~silindi~~
4–5	8	
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

1–2 = 8 silinir: çizge hâlâ bağlı (kenar bir döngünün en ağırıdır).

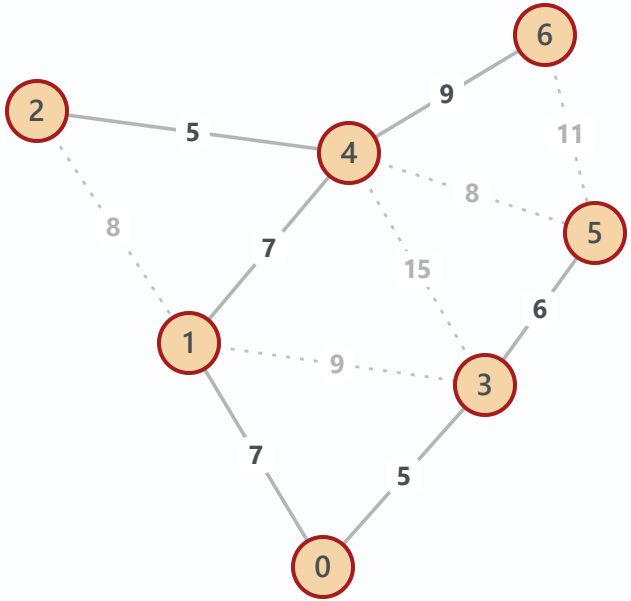
Ters Silme Adım Adım: Yedi Düğüm



Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
~~5–6~~	~~11~~	~~silindi~~
~~1–3~~	~~9~~	~~silindi~~
4–6	**9**	**kaldı**
~~1–2~~	~~8~~	~~silindi~~
~~4–5~~	~~8~~	~~silindi~~
1–4	7	
0–1	7	
3–5	6	
2–4	5	
0–3	5	

4–5 = 8 silinir: çizge hâlâ bağlı (kenar bir döngünün en ağırıdır).

Ters Silme Adım Adım: Yedi Düğüm

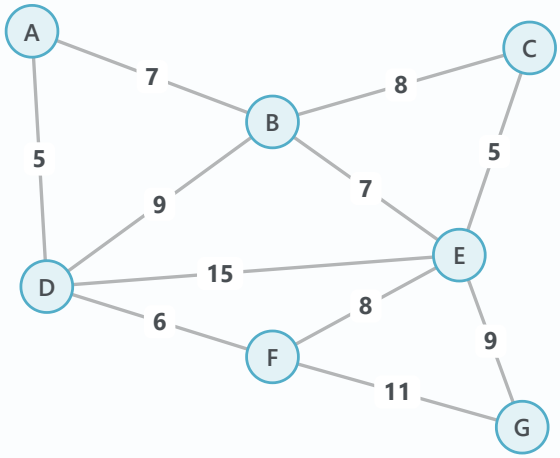


Kenar	w	Durum
~~3–4~~	~~15~~	~~silindi~~
~~5–6~~	~~11~~	~~silindi~~
~~1–3~~	~~9~~	~~silindi~~
4–6	**9**	**kaldı**
~~1–2~~	~~8~~	~~silindi~~
~~4–5~~	~~8~~	~~silindi~~
1–4	**7**	**kaldı**
0–1	**7**	**kaldı**
3–5	**6**	**kaldı**
2–4	**5**	**kaldı**
0–3	**5**	**kaldı**

Kalan 6 kenar (7, 7, 6, 5, 5 ve 9) birer **köprüdür**: hiçbiri silinemez. 6 = V − 1 kenar, **MST = 39**.

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
▶    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
6      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



Kenarlar azalan sıralanır: DE 15, FG 11, BD 9, EG 9, BC 8, EF 8, AB 7, BE 7, DF 6, AD 5, CE 5.

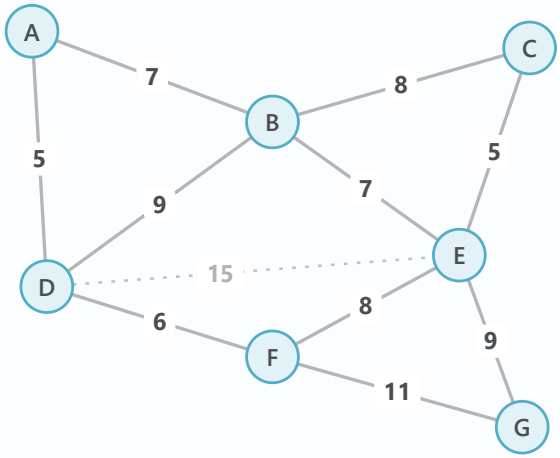
Adım 1 / 13

DEĞİŞKENLER

`i = 0` `boyut = 11` `toplam = 90`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
▶      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



En yüksek ağırlıklı DE incelenir: **DE = 15** silindi, `bagliMi` = true → silinmiş kalır.

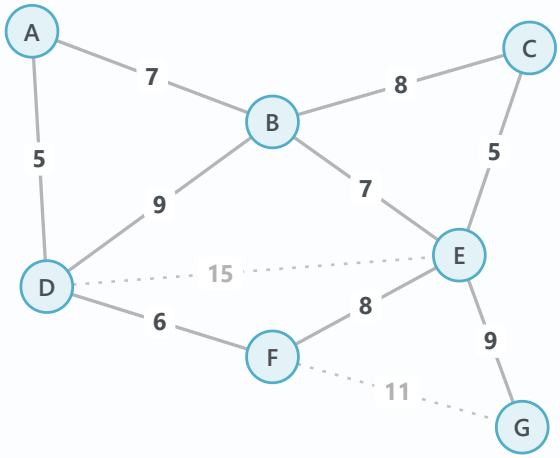
Adım 2 / 13

DEĞİŞKENLER

`i = 0` `e = DE(15)` `bagliMi = true` `boyut = 10` `toplam = 75`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
▶      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



FG çıkarılırsa bağlılık bozuluyor mu? **FG = 11** silindi, `bagliMi` = true → silinmiş kalır.

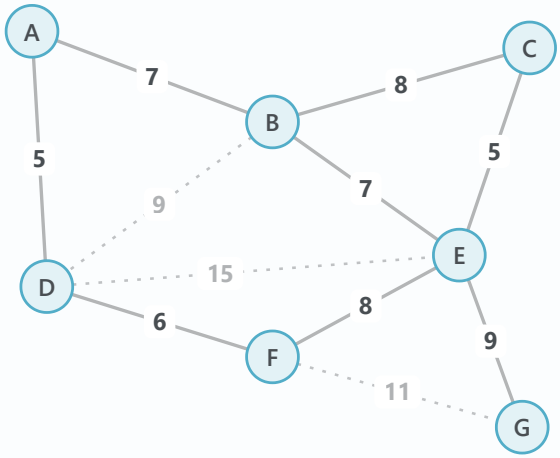
Adım 3 / 13

DEĞİŞKENLER

`i = 0` `e = FG(11)` `bagliMi = true` `boyut = 9` `toplam = 64`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
▶      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



Bir sonraki en ağır kenar BD: **BD = 9** silindi, `bagliMi` = true → silinmiş kalır.

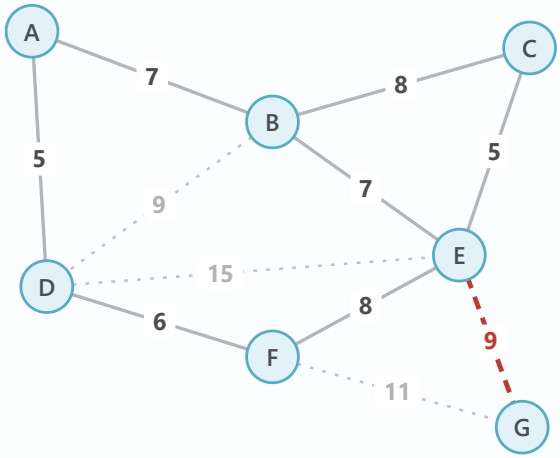
Adım 4 / 13

DEĞİŞKENLER

`i = 0` `e = BD(9)` `bagliMi = true` `boyut = 8` `toplam = 55`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
▶        ke.add(i, e);              // geri koy
▶        i++;
9      }
10   }
11 }
```



EG = 9 silinirse G yalnız kalır (FG zaten silindi) → `bagliMi` = false, EG geri konur; `i` = 1.

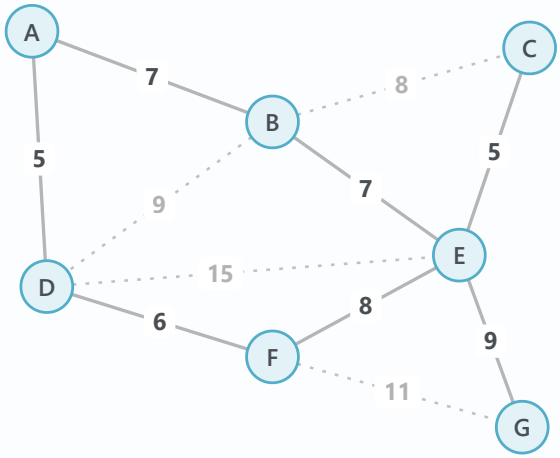
Adım 5 / 13

DEĞİŞKENLER

`i = 1` `e = EG(9)` `bagliMi = false` `boyut = 8` `toplam = 55`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
▶      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



BC = 8 silindi, `bagliMi` = true → silinmiş kalır.

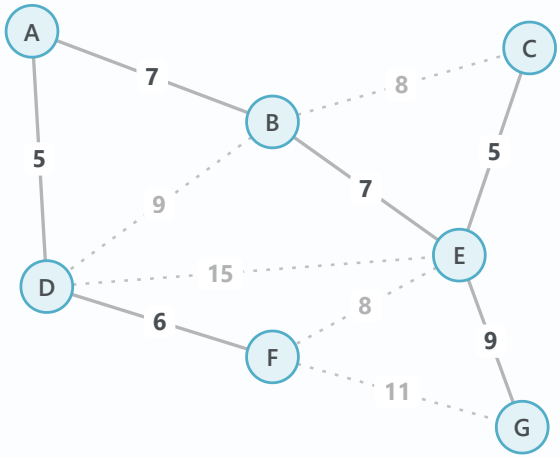
Adım 6 / 13

DEĞİŞKENLER

`i = 1` `e = BC(8)` `bagliMi = true` `boyut = 7` `toplam = 47`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
▶      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



EF = 8 silindi, `bagliMi` = true → silinmiş kalır.

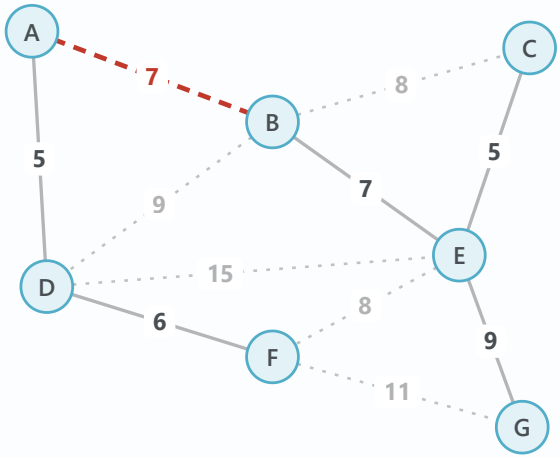
Adım 7 / 13

DEĞİŞKENLER

`i = 1` `e = EF(8)` `bagliMi = true` `boyut = 6` `toplam = 39`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
▶        ke.add(i, e);              // geri koy
▶        i++;
9      }
10   }
11 }
```



AB = 7 silinirse {A, D, F} kopar → geri konur, `i` = 2.

Adım 8 / 13

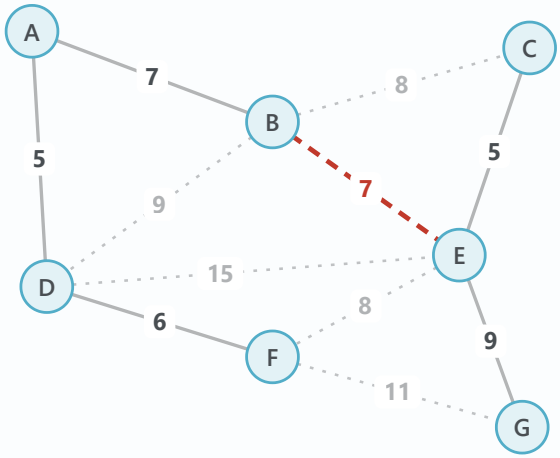
DEĞİŞKENLER

`i = 2` `e = AB(7)` `bagliMi = false`

`boyut = 6` `toplam = 39`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
▶        ke.add(i, e);              // geri koy
▶        i++;
9      }
10   }
11 }
```



BE = 7 silinirse {C, E, G} kopar → geri konur, `i` = 3.

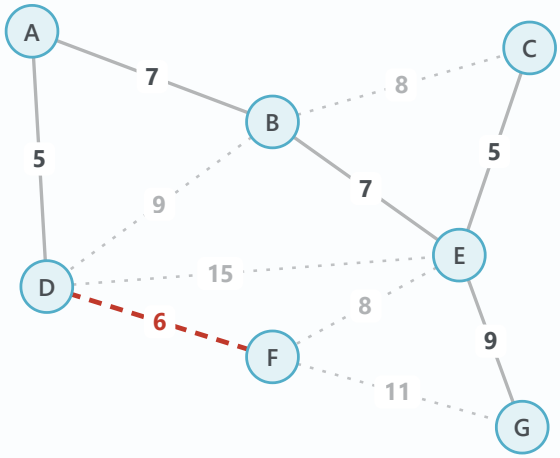
Adım 9 / 13

DEĞİŞKENLER

`i = 3` `e = BE(7)` `bagliMi = false` `boyut = 6` `toplam = 39`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
▶        ke.add(i, e);              // geri koy
▶        i++;
9      }
10   }
11 }
```



DF = 6 silinirse {F} kopar → geri konur, `i` = 4.

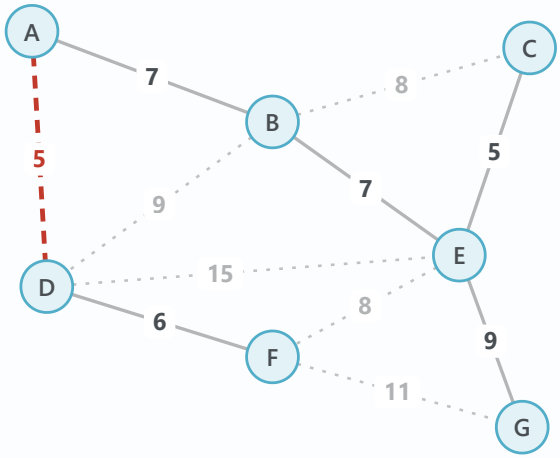
Adım 10 / 13

DEĞİŞKENLER

`i = 4` `e = DF(6)` `bagliMi = false` `boyut = 6` `toplam = 39`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
▶        ke.add(i, e);              // geri koy
▶        i++;
9      }
10   }
11 }
```



AD = 5 silinirse {D, F} kopar → geri konur, `i` = 5.

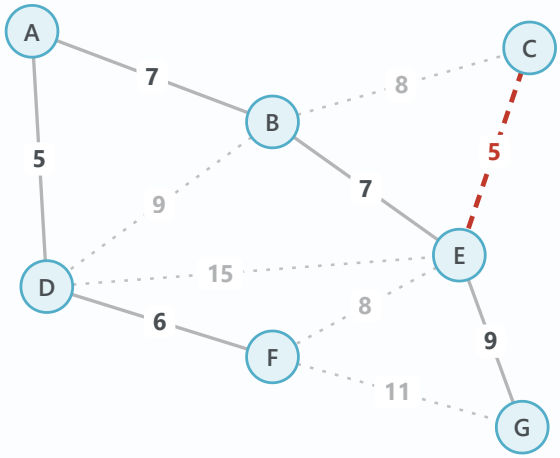
Adım 11 / 13

DEĞİŞKENLER

`i = 5` `e = AD(5)` `bagliMi = false` `boyut = 6` `toplam = 39`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
4    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
▶      if (!bagliMi(V, ke)) {
▶        ke.add(i, e);              // geri koy
▶        i++;
9      }
10   }
11 }
```



CE = 5 silinirse {C} kopar → geri konur, `i` = 6.

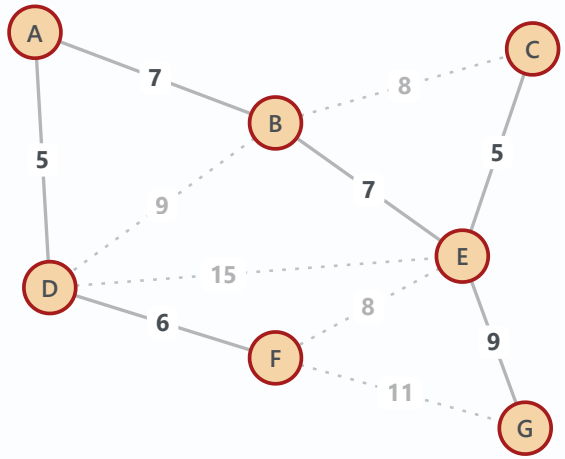
Adım 12 / 13

DEĞİŞKENLER

`i = 6` `e = CE(5)` `bagliMi = false` `boyut = 6` `toplam = 39`

Ters Silme: Kodu Adım Adım İzle

```
1  void tersSil(int V, List<Kenar> ke) {
2    ke.sort((a, b) -> b.w - a.w); // azalan
3    int i = 0;
▶    while (i < ke.size()) {
5      Kenar e = ke.remove(i);      // geçici sil
6      if (!bagliMi(V, ke)) {
7        ke.add(i, e);              // geri koy
8        i++;
9      }
10   }
11 }
```



`i = boyut = 6` : döngü biter. Silinecek başka kenar yok; kalan ağaç AB, AD, BE, CE, DF, EG → **w = 39**.

Adım 13 / 13

DEĞİŞKENLER

`i = 6` `boyut = 6` `toplam = 39`

KARŞILAŞTIRMA

Dört Algoritmanın Karşılaştırması

	Kruskal	Prim	Borůvka	Ters silme
Bakış	kenar odaklı	düğüm odaklı	bileşen odaklı	kenar odaklı (silme)
Ara durum	orman	tek ağaç	orman	bağlı çizge
Dayandığı özellik	kesim	kesim	kesim	döngü
Veri yapısı	sıralı kenarlar + union-find	öncelikli kuyruk / dizi	union-find	BFS/DFS
Zaman	O(E log E)	$O(V^2)$ ya da O((V + E) log V)	O(E log V)	O(E · (V + E))
En uygun	seyrek çizge	yoğun çizge	paralel / dağıtık	öğretim, teori
Bağlı olmayan çizge	orman verir	bir bileşen	orman verir	bileşenleri korur

Dokuz düğümlü çizgede Kruskal ve Prim aynı ağacı (29) bulur; ekleme **sırası** farklıdır: Kruskal 1, 1, 2, 3, 4, 5, 6, 7 — Prim 4, 7, 1, 1, 2, 3, 6, 5.

Özet

Konu	Hatırlanacaklar
MST	bağlı, yönsüz, ağırlıklı çizgede toplam ağırlığı en küçük kapsayan ağaç; V − 1 kenar
Özellikler	kesimin en hafif kenarı güvenli; döngünün en ağır kenarı dışarıda; farklı ağırlıklar → tek MST
Kruskal	kenarlar sıralanır; `bul(u) ≠ bul(v)` ise kenar eklenir, ağaçlar birleşir — O(E log E)
Union-Find	`ata[]` ormanı; yol kısaltma + rütbe → işlem başına O(α(V))
Prim	tek ağacı büyüt; anahtar = ağaca en ucuz kenar — $O(V^2)$ / O((V + E) log V)
Borůvka	her bileşen en ucuz dış kenarını aynı anda ekler; ≤ $\log_2 V$ tur — O(E log V)
Ters silme	azalan sırada sil, bağlılık bozulursa geri koy — O(E · (V + E))

ÖZET

Etkileşimli Simülatörler

Minimum Kapsayan Ağaç

2–15 düğümlü rastgele, bağlı ve ağırlıklı çizgelerde (düğümler sürüklenebilir) Prim (düğüm 0'dan) ve Kruskal adım adım: eklenen ve döngü yüzünden atlanan kenarlar günlükte listelenir, sonunda toplam ağırlık gösterilir.

Prim ve Kruskal

Aynı çizgede iki algoritmanın bulduğu toplam ağırlık aynıdır, kenar ekleme sırası farklıdır. Ağırlıkları eşit kenarlar varsa iki ağaç farklı olabilir.

ÖZET

Sık Yapılan Hatalar

MST'yi en kısa yol ağacı sanmak

MST toplam ağırlığı küçültür; iki düğüm arasındaki MST yolu en kısa yol olmak zorunda değildir.

Döngüyü komşulukla denetlemek

Kruskal'da "u ile v doğrudan bağlı mı?" yetmez; **aynı bileşende mi** sorusu (bul) sorulmalıdır.

Prim'de uzaklık toplamak

`anahtar[u] + w` yazmak Dijkstra'dır. Prim'de anahtar yalnızca **tek kenarın** ağırlığıdır.

Negatif ağırlıktan korkmak

MST algoritmaları negatif ağırlıkla da doğru çalışır (yalnızca karşılaştırma yapılır); sorun Dijkstra'ya özgüdür.

Bağlı olmayan çizge

MST yoktur, minimum kapsayan **orman** vardır. `while (agac.size() < V - 1)` kenarlar bitince dizi sınırını aşar.

Yönlü çizgeye uygulamak

Bu algoritmalar yönsüz çizge içindir. Yönlü karşılığı (minimum kapsayan dallanma) Chu–Liu/Edmonds algoritmasıdır.

ÖZET

Örnek Problemler

Çizge: **AB = 2, AC = 3, BC = 1, BD = 4, CD = 5, CE = 6, DE = 7** (yönsüz).

1. Kruskal'ın incelediği kenarları sırasıyla yazın; hangileri reddedilir? w(T) = ?
2. Prim'i A'dan çalıştırın: düğümler hangi sırayla ağaca girer? İlk adımdan sonra anahtar dizisi nedir?
3. Borůvka kaç turda biter? Tur 1'de hangi kenarlar seçilir?
4. Ters silme hangi kenarları siler?
5. 100 düğümlü bağlı bir çizgenin MST'sinde kaç kenar vardır?
6. Tüm ağırlıklara 5 eklenirse MST değişir mi? Toplam ne olur?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. Tüm ağırlıklar farklıysa MST'nin tek olduğunu kesim özelliğiyle gösterin.
2. **En büyük** kapsayan ağacı bulmak için Kruskal'ı nasıl değiştirirsiniz?
3. Dokuz düğümlü örneğin MST'sini kullanarak düğümleri **3 kümeye** ayırın.
4. 0..5, `ata[bul(x)] = bul(y)` : birleştir (0,1), (2,3), (1,3), (4,5), (3,5). Son `ata` ? `bul(0)` neyi değiştirir?
5. V = 1000: Prim $O(V^2)$ ve O(E log V) sürümlerini E = 5 000 ve 500 000 için karşılaştırın.
6. Kruskal'ı bağlı olmayan çizgede orman döndürecek biçimde düzeltin.

▶ **Cevaplar / ipuçları**

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022 (Bölüm 19: ayrık kümeler, Bölüm 21: MST).
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Bölüm 4.3: Minimum Spanning Trees).
- J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem", *Proc. AMS* 7(1), 1956.
- R. C. Prim, "Shortest Connection Networks and Some Generalizations", *Bell System Technical Journal* 36(6), 1957.
- O. Borůvka, "O jistém problému minimálním", *Práce Moravské přírodovědecké společnosti* 3, 1926.
- R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm", *Journal of the ACM* 22(2), 1975.

Son

Bölüm 4.3: Minimum Kapsayan Ağaç Algoritmaları

Sonraki bölüm: Bölüm 4.4: Ağ Akışı · Terim sözlüğü