Bölüm 4.2: En Kısa Yol Algoritmaları

Algoritmalar

Gevşetme | Dijkstra | Bellman-Ford | Negatif döngü | Floyd-Warshall | A* arama | BFS ile ağırlıksız yol | Karşılaştırma

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavramlar

Tek kaynaklı, tüm çiftler ve tek çift en kısa yol problemlerini ayırt etmek; **gevşetme** (relaxation) işlemini ve önceki (π) dizisini açıklamak.

Dijkstra ve Bellman-Ford

İki algoritmayı bir örnek çizge üzerinde adım adım çalıştırmak, uzaklık ve π tablolarını doldurmak, Java kodunu izlemek.

Negatif kenar ve döngü

Dijkstra'nın negatif kenarda neden yanıldığını göstermek; Bellman-Ford ve Floyd-Warshall ile negatif döngüyü tespit etmek.

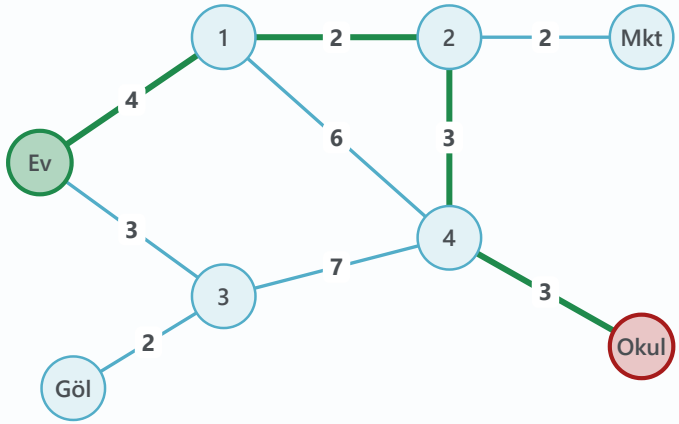
Floyd-Warshall ve A*

Tüm çiftler için D ve Π matrislerini k = 1…n güncellemek; A*'ın f = g + h ile hedefe yönelen aramasını uygulamak ve algoritmaları karşılaştırmak.

GİRİŞ

Yol Ağı Bir Çizgedir

- Kavşaklar **düğüm** (vertex), yollar **kenar** (edge), yol uzunlukları ya da süreleri **ağırlık** (weight) olur.
- Bir yolun **uzunluğu** = üzerindeki kenar ağırlıklarının toplamı.
- **En kısa yol** (shortest path): iki düğüm arasındaki toplam ağırlığı en küçük yol; uzunluğu δ(s, v) ile gösterilir.
- Örnek: Ev → Okul için Ev–1–2–4–Okul = 4 + 2 + 3 + 3 = **12**; daha az kenarlı Ev–1–4–Okul ise 13 eder.



Kenar sayısı en az olan yol, ağırlıklı çizgede en kısa yol **olmayabilir**. Navigasyon, ağ yönlendirme (OSPF, RIP), oyunlarda yol bulma ve lojistik bu problemi çözer.

GİRİŞ

En Kısa Yol Problem Türleri

Tek kaynaklı

(single-source) s'den **tüm** düğümlere. Dijkstra, Bellman-Ford, BFS.

Tüm çiftler

(all-pairs) her (u, v) çifti; sonuç n × n matris. Floyd-Warshall.

Tek çift

(single-pair) yalnızca s → t; hedefe yönelen arama. A*.

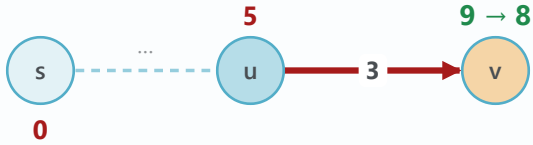
Algoritma	Ne bulur?	Kısıt / not
Dijkstra	kaynaktan tüm düğümlere	kenar ağırlıkları **negatif olmamalı**
Bellman-Ford	kaynaktan tüm düğümlere	negatif kenarda çalışır, negatif döngüyü bulur; Dijkstra'dan yavaş
Floyd-Warshall	tüm çiftler arası	negatif kenarda çalışır; negatif döngüyü köşegenden tespit eder
A*	başlangıçtan hedefe	sezgisel (heuristic) tahminle aramayı hızlandırır
BFS	kaynaktan tüm düğümlere	yalnızca ağırlıksız (ya da eşit ağırlıklı) çizgeler

Temel İşlem: Gevşetme (Relaxation)

Her düğüm için iki bilgi tutulur:

- `d[v]` : şimdiye kadar bulunan en kısa yol **tahmini** (başta ∞, kaynakta 0)
- `π[v]` : bu yolda v'den önceki düğüm (önceki, predecessor)

```
RELAX(u, v, w):
  eğer d[u] + w(u, v) < d[v]:
    d[v] = d[u] + w(u, v)
    π[v] = u
```



d[u] + w = 5 + 3 = 8 < 9 → d[v] = 8, π[v] = u

Bu bölümdeki tüm algoritmalar gevşetmeyi kullanır; fark, kenarları **hangi sırayla** ve **kaç kez** gevşettikleridir. d[v] yalnızca azalabilir ve hiçbir zaman gerçek δ(s, v)'nin altına inmez.

GİRİŞ

En Kısa Yolların Özellikleri

Optimal alt yapı

En kısa yolun her alt yolu da en kısa yoldur. $s \rightsquigarrow u \rightsquigarrow v$ en kısaysa $s \rightsquigarrow u$ parçası da en kısadır. Dinamik programlama ve açgözlü yaklaşım buna dayanır.

Üçgen eşitsizliği

Her (u, v) kenarı için $\delta(s, v) \leq \delta(s, u) + w(u, v)$. Gevşetme bu eşitsizliği bozan kenarları düzeltir.

Negatif döngü

Toplam ağırlığı negatif olan bir döngü kaynaktan ulaşılabiliyorsa, döngüde dolaştıkça yol kısalır: en kısa yol **tanımsızdır** $(-\infty)$.

Basit yol

Negatif döngü yoksa en kısa yol döngü içermez; en fazla $\mathbf{V - 1}$ kenarlıdır. Bellman-Ford'un $V - 1$ turu buradan gelir.

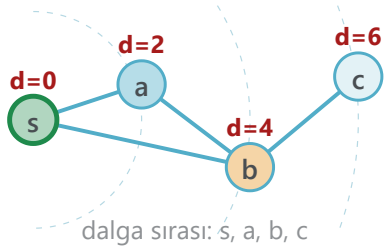
Yolu geri kurmak için π dizisi kullanılır: hedeften başlayıp $\pi[v]$ ile kaynağa kadar geri gidilir; elde edilen düğümler ters sırada yazılır. Tüm π kenarları birlikte bir **en kısa yol ağacı** oluşturur.

Dijkstra Algoritması

- **Kaynak** düğümden diğer tüm düğümlere olan en kısa yolları bulur (tek kaynaklı).
- 1956'da **Edsger W. Dijkstra** tarafından geliştirildi, 1959'da yayımlandı.
- Yalnızca **negatif olmayan** ağırlıklı kenarlardan oluşan çizgelerde doğru çalışır.
- **Açgözlü** (greedy): her adımda, henüz kesinleşmemiş düğümler arasından d değeri **en küçük** olanı seçer ve kesinleştirir.
- Seçilen düğümün komşuları gevşetilir.

Su dalgası benzetmesi

Kaynağa atılan bir taşın dalgası düğümlere uzaklık sırasıyla ulaşır: önce en yakın düğüm, sonra bir sonraki... Dijkstra düğümleri tam da bu sırayla **kesinleştirir**. Kesinleşen düğümlerin kümesine **S** denir.



dalga sırası: s, a, b, c

Dijkstra: Algoritma Adımları

Adım 1: Başlangıç düğümü seçilir, uzaklığı **0** atanır; diğer tüm düğümlerin uzaklığı **∞** olur.

İlklendirme: d[s] = 0, d[v] = ∞, π[v] = tanımsız, S = Ø.

Dijkstra: Algoritma Adımları

Adım 1: Başlangıç düğümü seçilir, uzaklığı **0** atanır; diğer tüm düğümlerin uzaklığı **∞** olur.

İlklendirme: d[s] = 0, d[v] = ∞, π[v] = tanımsız, S = Ø.

Adım 2: Seçilen düğümden başlayarak, henüz işlenmemiş **komşulara** olan uzaklıklar hesaplanır.

Komşu v için aday uzaklık: d[u] + w(u, v).

Dijkstra: Algoritma Adımları

Adım 1: Başlangıç düğümü seçilir, uzaklığı **0** atanır; diğer tüm düğümlerin uzaklığı **∞** olur.

İlklendirme: d[s] = 0, d[v] = ∞, π[v] = tanımsız, S = Ø.

Adım 2: Seçilen düğümden başlayarak, henüz işlenmemiş **komşulara** olan uzaklıklar hesaplanır.

Komşu v için aday uzaklık: d[u] + w(u, v).

Adım 3: Daha kısa bir yol bulunursa komşunun uzaklığı **güncellenir** (gevşetme).

d[u] + w(u, v) < d[v] ise d[v] ve π[v] değişir.

Dijkstra: Algoritma Adımları

Adım 1: Başlangıç düğümü seçilir, uzaklığı **0** atanır; diğer tüm düğümlerin uzaklığı **∞** olur.
İlklendirme: d[s] = 0, d[v] = ∞, π[v] = tanımsız, S = Ø.

Adım 2: Seçilen düğümden başlayarak, henüz işlenmemiş **komşulara** olan uzaklıklar hesaplanır.
Komşu v için aday uzaklık: d[u] + w(u, v).

Adım 3: Daha kısa bir yol bulunursa komşunun uzaklığı **güncellenir** (gevşetme).
d[u] + w(u, v) < d[v] ise d[v] ve π[v] değişir.

Adım 4: İşlenen düğüm **işaretlenir** (S'ye eklenir) ve uzaklığı en küçük işaretsiz düğüm **seçilir**.
Seçim, dizi ya da öncelikli kuyruk (priority queue) ile yapılır.

Dijkstra: Algoritma Adımları

Adım 1: Başlangıç düğümü seçilir, uzaklığı **0** atanır; diğer tüm düğümlerin uzaklığı **∞** olur.
İlklendirme: d[s] = 0, d[v] = ∞, π[v] = tanımsız, S = Ø.

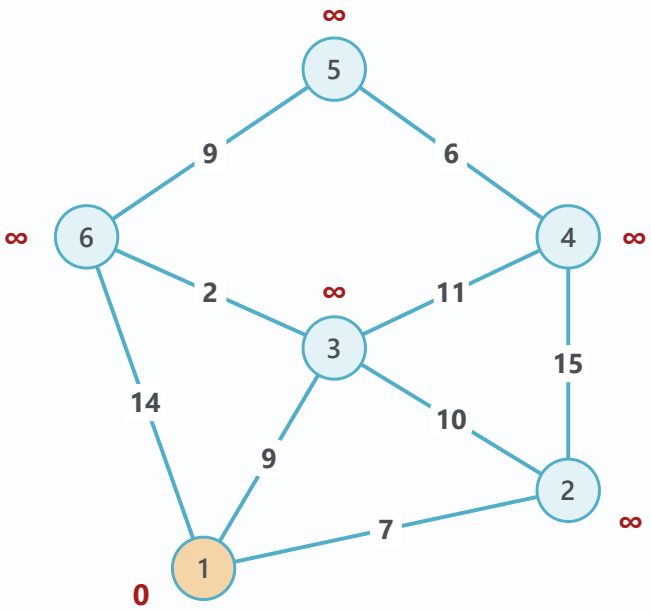
Adım 2: Seçilen düğümden başlayarak, henüz işlenmemiş **komşulara** olan uzaklıklar hesaplanır.
Komşu v için aday uzaklık: d[u] + w(u, v).

Adım 3: Daha kısa bir yol bulunursa komşunun uzaklığı **güncellenir** (gevşetme).
d[u] + w(u, v) < d[v] ise d[v] ve π[v] değişir.

Adım 4: İşlenen düğüm **işaretlenir** (S'ye eklenir) ve uzaklığı en küçük işaretsiz düğüm **seçilir**.
Seçim, dizi ya da öncelikli kuyruk (priority queue) ile yapılır.

Adım 5: Tüm düğümler işlenene kadar Adım 2–4 **tekrarlanır**.
Ulaşılamayan düğümlerin uzaklığı ∞ olarak kalır.

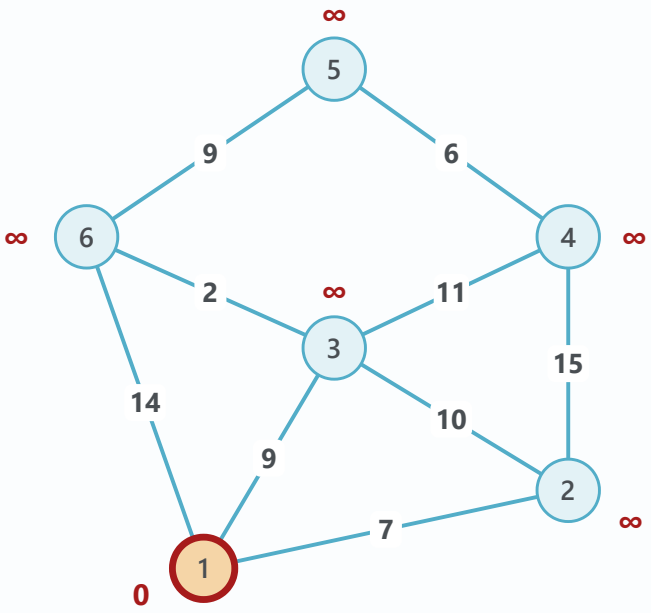
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	0	–
2	∞	–
3	∞	–
4	∞	–
5	∞	–
6	∞	–

İlklendirme: kaynak **1**, d[1] = 0, diğerleri ∞. Hiçbir düğüm kesinleşmedi.

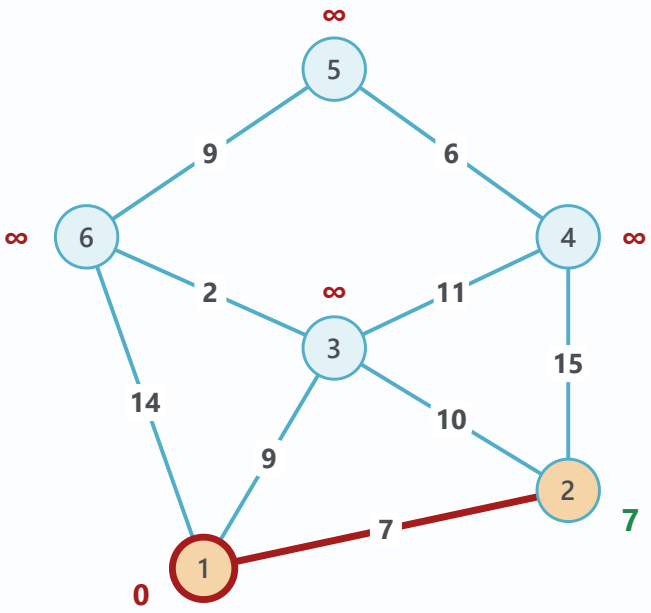
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	∞	–
3	∞	–
4	∞	–
5	∞	–
6	∞	–

Kesinleşmemişler içinde en küçük d: **1** (d = 0) seçilir ve kesinleşir.

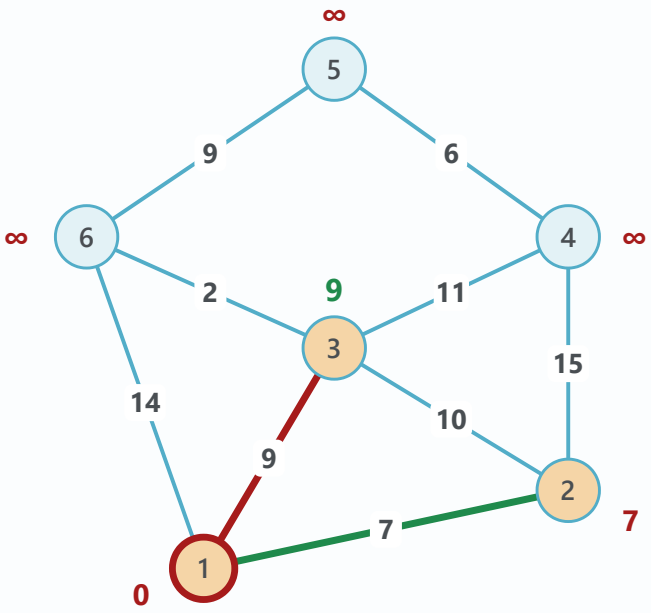
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	**1**
3	∞	–
4	∞	–
5	∞	–
6	∞	–

Kenar (1, 2): d[1] + 7 = 7 < ∞ → **d[2] = 7**, π[2] = 1.

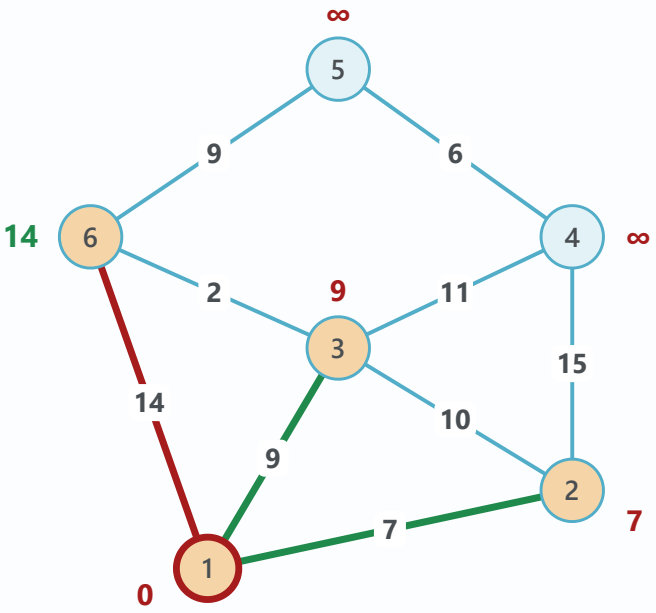
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	7	1
3	**9**	**1**
4	∞	–
5	∞	–
6	∞	–

Kenar (1, 3): d[1] + 9 = 9 < ∞ → **d[3] = 9**, π[3] = 1.

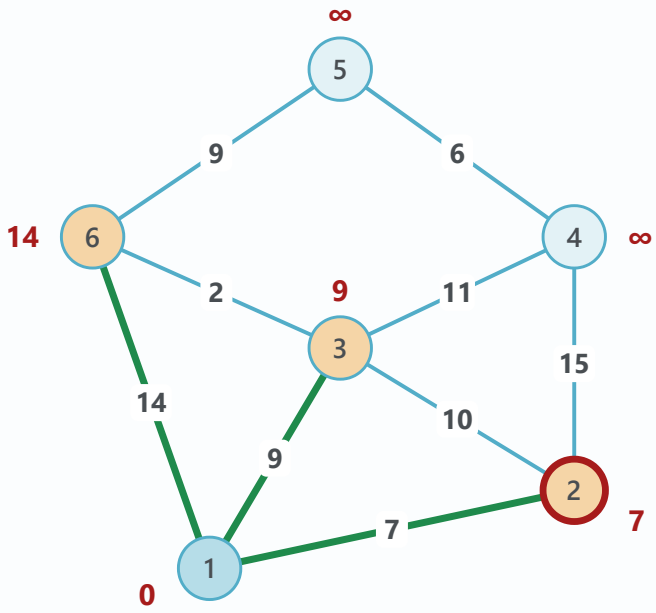
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	7	1
3	9	1
4	∞	–
5	∞	–
6	**14**	**1**

Kenar (1, 6): d[1] + 14 = 14 < ∞ → **d[6] = 14**, π[6] = 1.

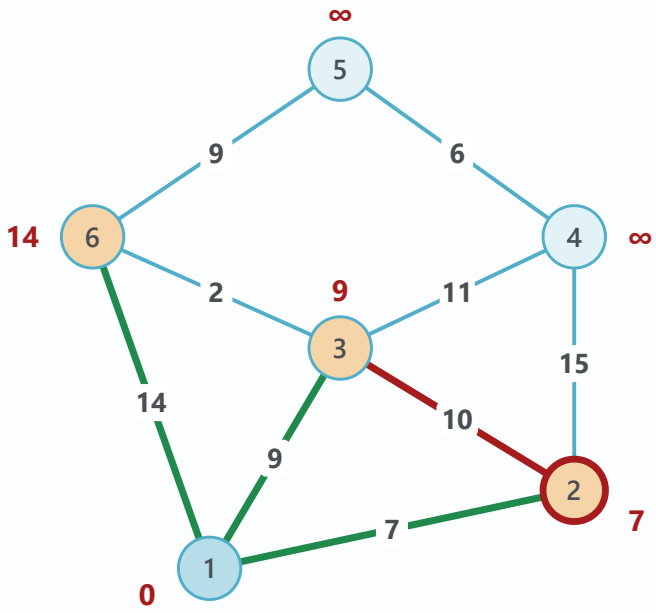
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	9	1
4	∞	–
5	∞	–
6	14	1

Kesinleşmemişler içinde en küçük d: **2** (d = 7) seçilir ve kesinleşir. Kesinleşmiş komşular (1) atlanır.

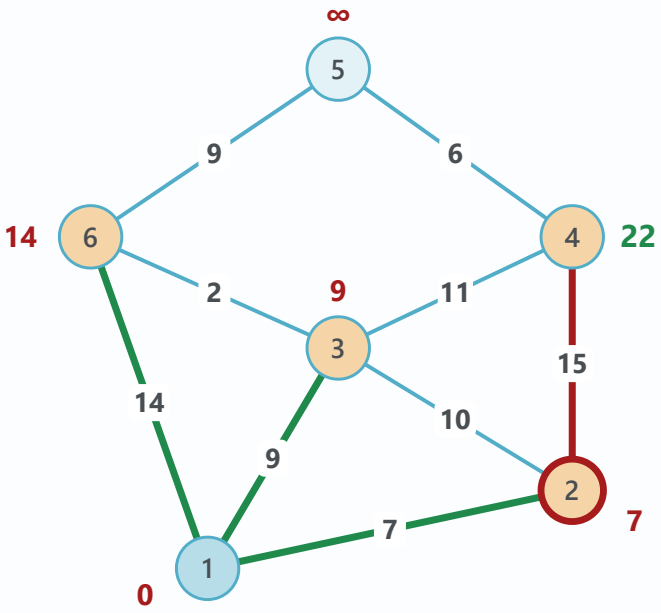
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	9	1
4	∞	–
5	∞	–
6	14	1

Kenar (2, 3): d[2] + 10 = 17 ≥ 9 → değişiklik yok.

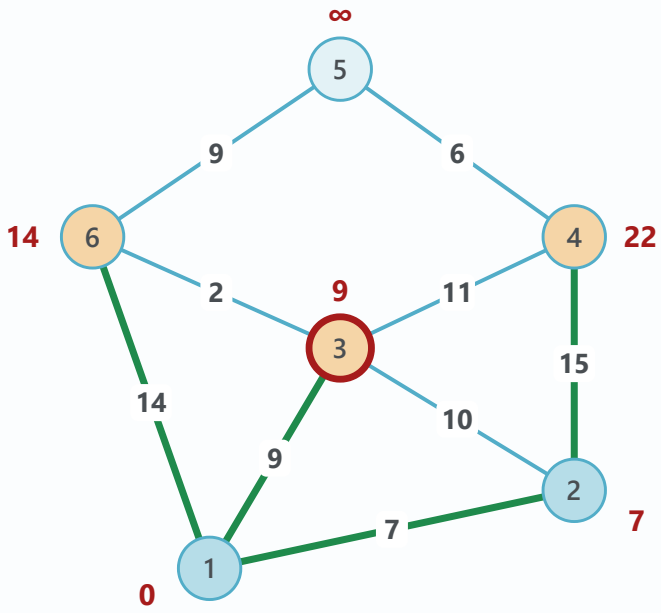
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	9	1
4	**22**	**2**
5	∞	–
6	14	1

Kenar (2, 4): d[2] + 15 = 22 < ∞ → **d[4] = 22**, π[4] = 2.

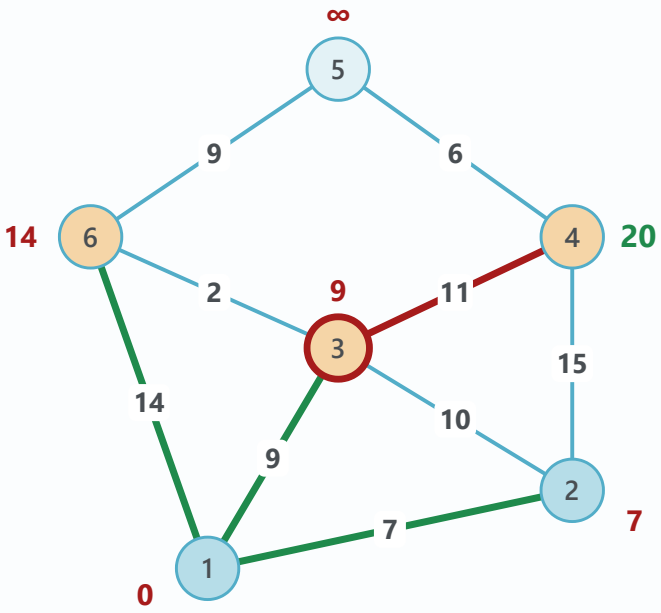
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	22	2
5	∞	–
6	14	1

Kesinleşmemişler içinde en küçük d: **3** (d = 9) seçilir ve kesinleşir. Kesinleşmiş komşular (1, 2) atlanır.

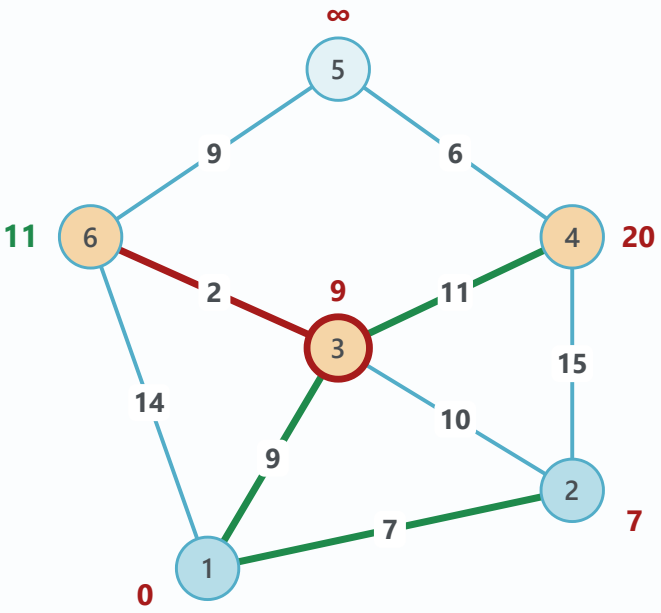
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	**3**
5	∞	–
6	14	1

Kenar (3, 4): d[3] + 11 = 20 < 22 → **d[4] = 20**, π[4] = 3.

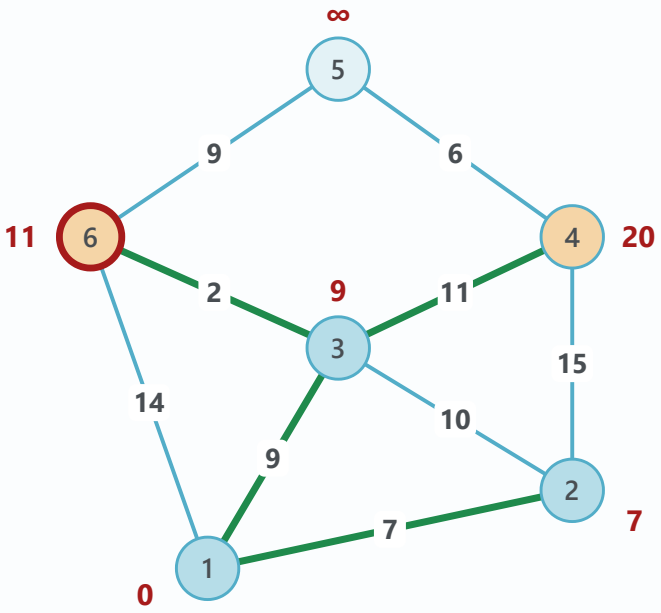
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	20	3
5	∞	–
6	**11**	**3**

Kenar (3, 6): d[3] + 2 = 11 < 14 → **d[6] = 11**, π[6] = 3.

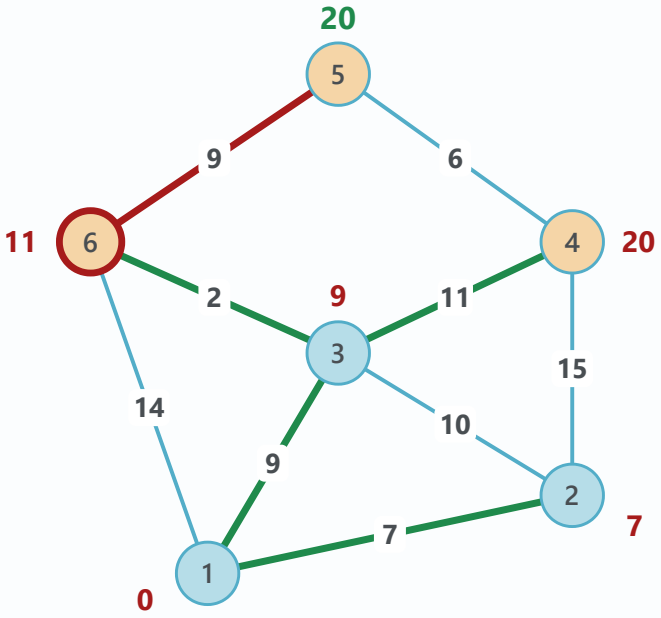
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	20	3
5	∞	–
6	**11**	3

Kesinleşmemişler içinde en küçük d: **6** (d = 11) seçilir ve kesinleşir. Kesinleşmiş komşular (1, 3) atlanır.

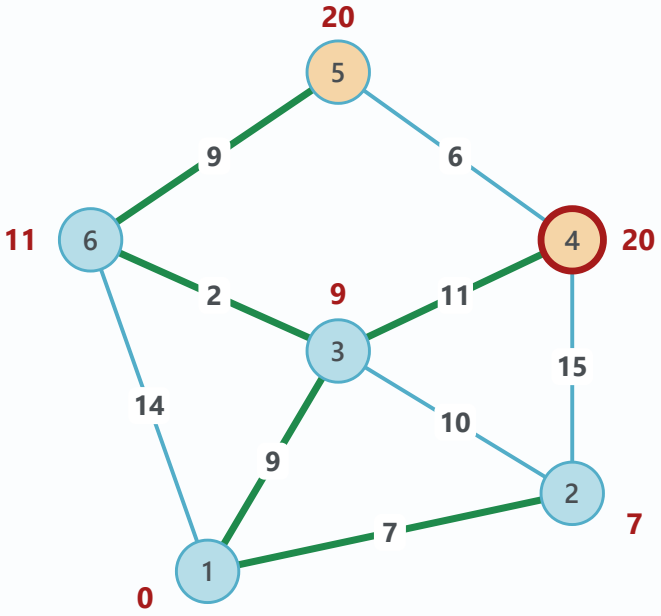
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	20	3
5	**20**	**6**
6	**11**	3

Kenar (6, 5): d[6] + 9 = 20 < ∞ → **d[5] = 20**, π[5] = 6.

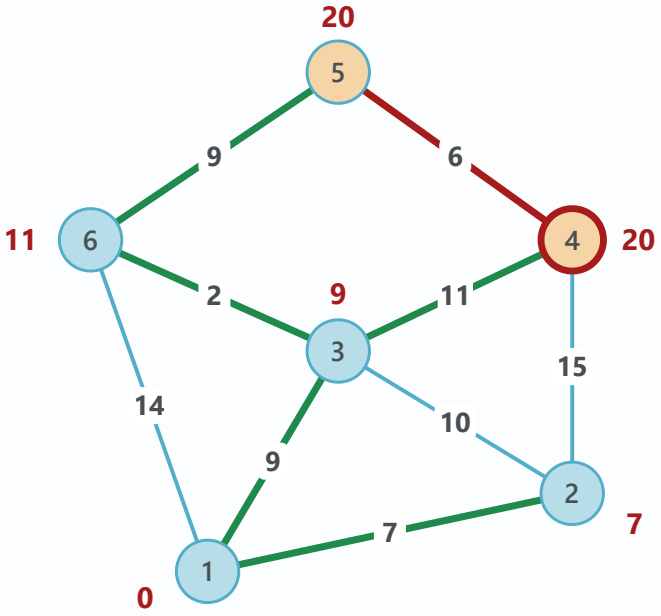
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	20	6
6	**11**	3

Kesinleşmemişler içinde en küçük d: **4** (d = 20) seçilir ve kesinleşir. (4 ve 5 eşit: 20; küçük numaralı 4 alındı.) Kesinleşmiş komşular (2, 3) atlanır.

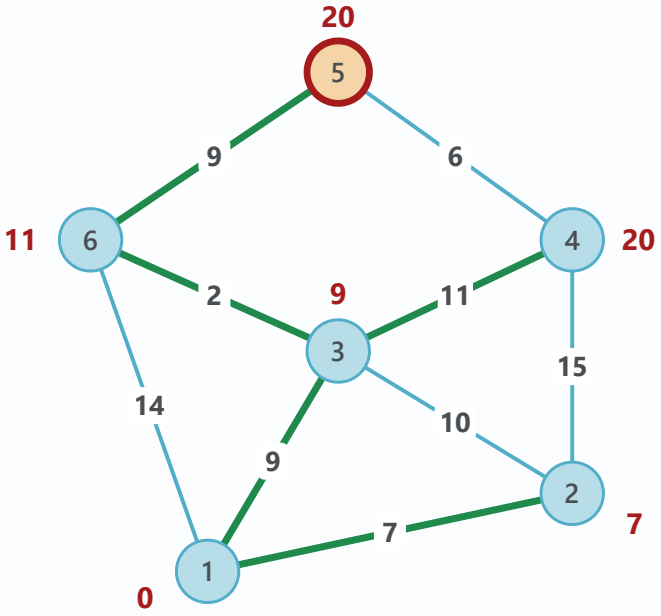
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	20	6
6	**11**	3

Kenar (4, 5): d[4] + 6 = 26 ≥ 20 → değişiklik yok.

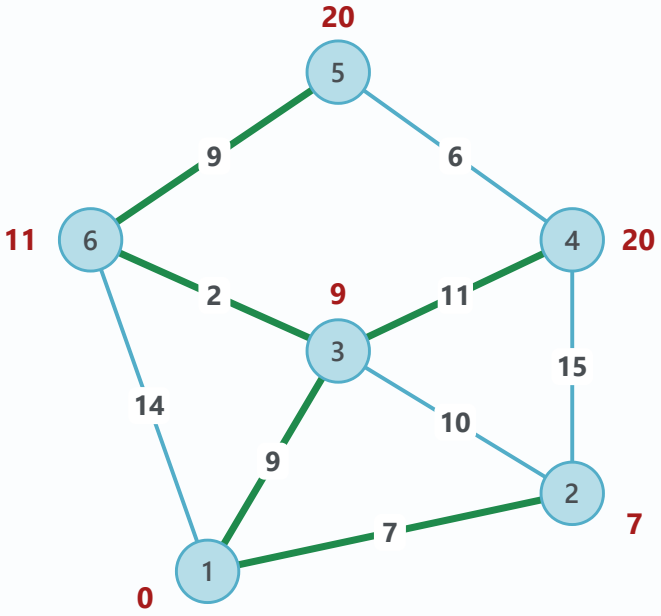
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	**20**	6
6	**11**	3

Kesinleşmemişler içinde en küçük d: **5** (d = 20) seçilir ve kesinleşir. Kesinleşmiş komşular (4, 6) atlanır.

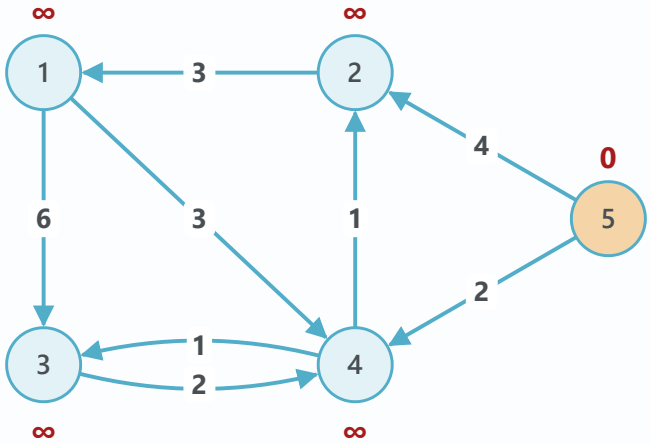
Dijkstra Örnek 1: Yönsüz Çizge, Kaynak 1



düğüm	d	π
1	**0**	–
2	**7**	1
3	**9**	1
4	**20**	3
5	**20**	6
6	**11**	3

Bitti: 1'den uzaklıklar 2:7, 3:9, 6:11, 4:20, 5:20. Yeşil kenarlar **en kısa yol ağacını** oluşturur (ör. 1 → 3 → 6 → 5).

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



Q

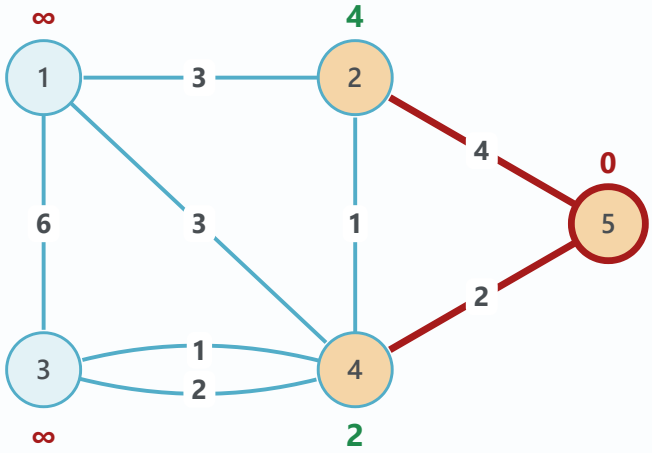
d	u	π
0	5	/
∞	1	/
∞	2	/
∞	3	/
∞	4	/

S

u	d	π

İlklendirme: kaynak **5**'in uzaklığı 0, diğerleri ∞. Tüm düğümler Q'da, S = ∅.

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



Q

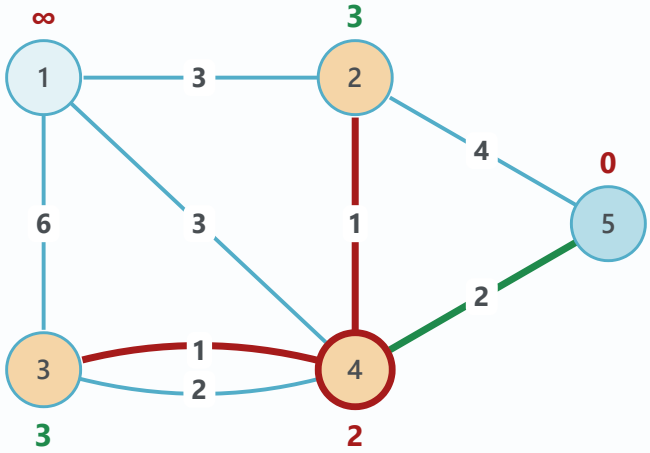
d	u	π
2	4	5
4	2	5
∞	1	/
∞	3	/

S

u	d	π
5	0	/

Adım 1: Q'dan **5** alınır, d = 0 ile S'ye konur. Gevşetme: (5,2): ∞ → **4**; (5,4): ∞ → **2**.

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



Q

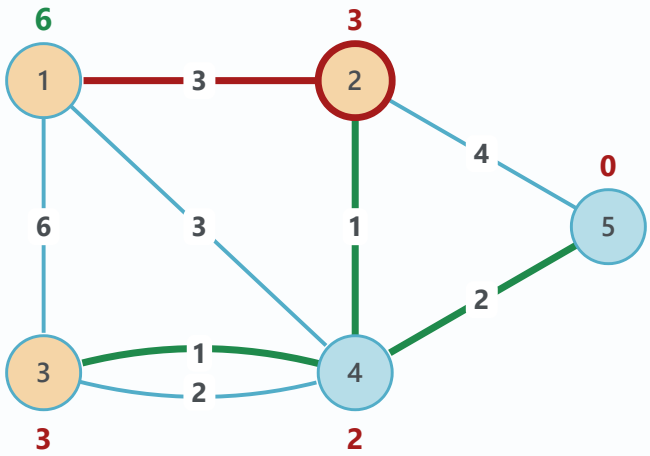
d	u	π
3	2	4
3	3	4
∞	1	/

S

u	d	π
5	0	/
4	2	5

Adım 2: Q'dan **4** alınır, d = 2 ile S'ye konur. Gevşetme: (4,2): 4 → **3**; (4,3): ∞ → **3**.

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



Q

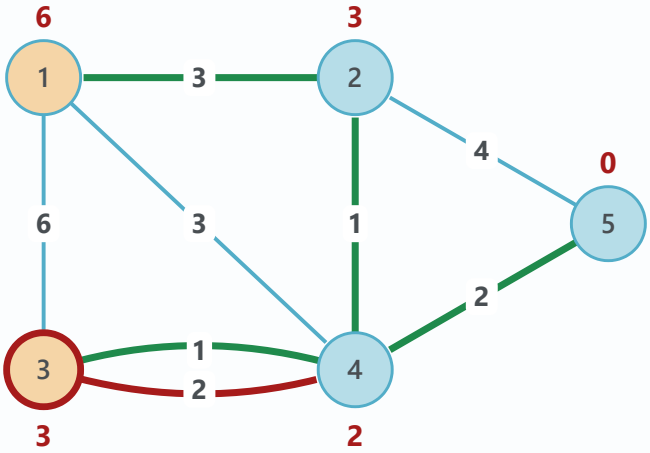
d	u	π
3	3	4
6	1	2

S

u	d	π
5	0	/
4	2	5
2	3	4

Adım 3: Q'dan **2** alınır, d = 3 ile S'ye konur. Gevşetme: (2,1): ∞ → **6**. (2 ve 3 eşit: 3; küçük numaralı 2 alındı.)

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



Q

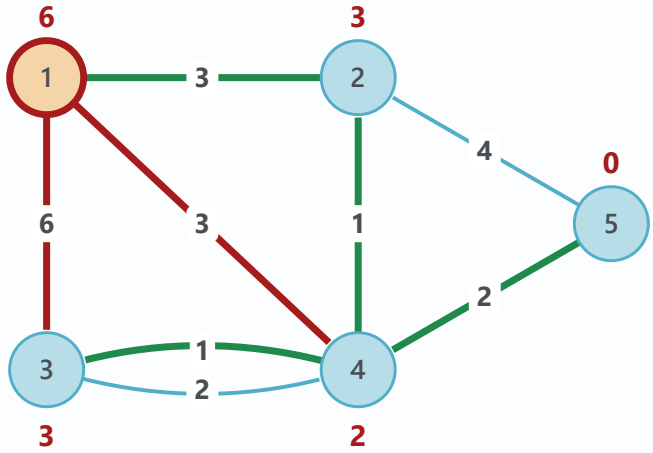
d	u	π
6	1	2

S

u	d	π
5	0	/
4	2	5
2	3	4
3	3	4

Adım 4: Q'dan **3** alınır, d = 3 ile S'ye konur. Gevşetme: (3,4): 4 zaten S'de. Daha kısa yol çıkmaz (gevşetme yok).

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



Q

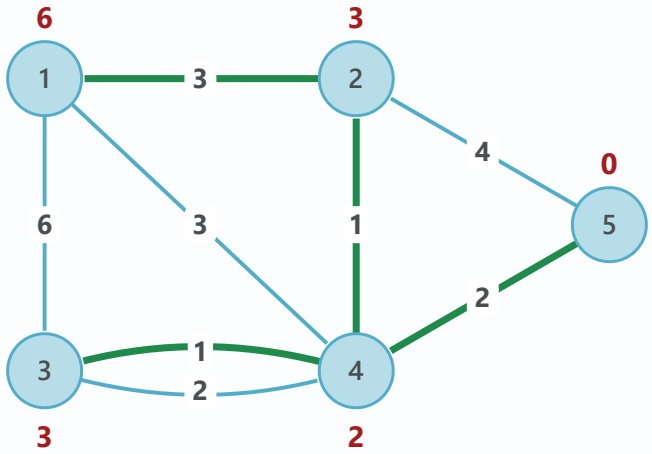
d	u	π

S

u	d	π
5	0	/
4	2	5
2	3	4
3	3	4
1	6	2

Adım 5: Q'dan **1** alınır, d = 6 ile S'ye konur. Gevşetme: (1,3): 3 zaten S'de; (1,4): 4 zaten S'de. Daha kısa yol çıkmaz (gevşetme yok).

Dijkstra Örnek 2: Yönlü Çizge, Q ve S Kümeleri



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

Son durum: 5'ten en kısa yollar. Yalnızca π kenarları (yeşil) kalır: 5→4→2→1 (6), 5→4→3 (3).

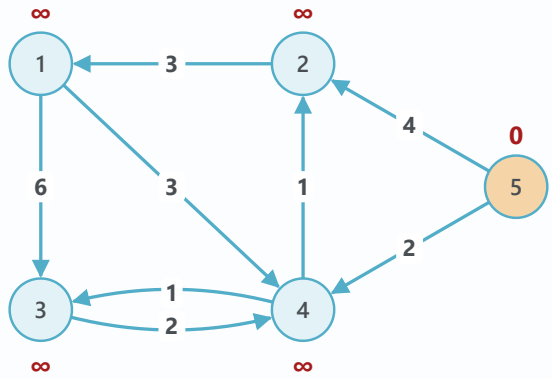
Dijkstra: Sözde Kod

```
DIJKSTRA(G, kaynak):
  mesafeler = [ ]; ziyaretEdildi = [ ]
  öncelikKuyruğu = { }
  her bir v için V içinde:
    mesafeler[v] = sonsuz
    ziyaretEdildi[v] = 0
  mesafeler[kaynak] = 0
  öncelikKuyruğu.ekle(kaynak, 0)
  döngü öncelikKuyruğu boş değil iken:
    u = öncelikKuyruğu.çıkar()   // en küçük mesafe
    eğer ziyaretEdildi[u] == 0:
      ziyaretEdildi[u] = 1
      her bir (u, v) için G[u] içinde:
        eğer mesafeler[u] + ağırlık(u, v) < mesafeler[v]:
          mesafeler[v] = mesafeler[u] + ağırlık(u, v)
          öncelikKuyruğu.ekle(v, mesafeler[v])
  döndür mesafeler
```

Tembel silme: bir düğüm kuyruğa birden çok kez girebilir; eski kayıt çıkarıldığında `ziyaretEdildi` kontrolü onu atlar. Yol için gevşetmede `önceki[v] = u` da saklanır.

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Diziler kurulur (düğümler 1..5, indis 0 kullanılmaz); uzaklik[5] = 0 ve (5, 0) kuyruğa eklenir.

Adım 1 / 27

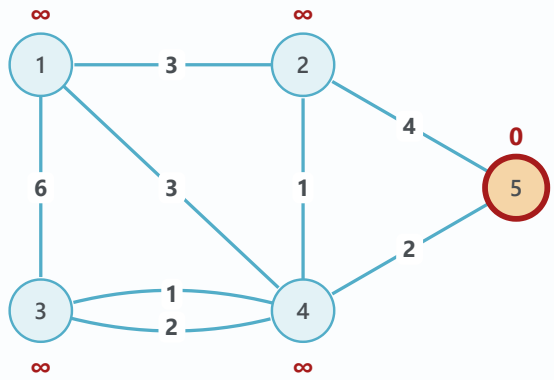
DEĞİŞKENLER

`u = –` `v = –` `w = –`

`kuyruk = [(5,0)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
▶      int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruktan en küçük uzaklıklı kayıt **(5, 0)** alınır: u = 5.

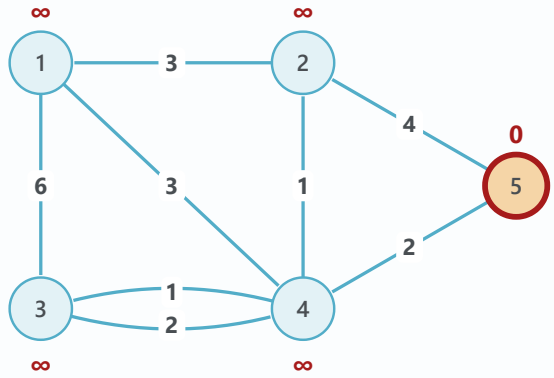
Adım 2 / 27

DEĞİŞKENLER

`u = 5` `v = –` `w = –` `kuyruk = []`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
▶      ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



5 kesinleşir: ziyaret[5] = true.

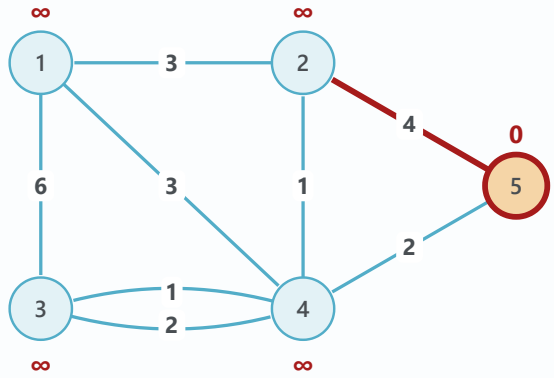
Adım 3 / 27

DEĞİŞKENLER

`u = 5` `v = –` `w = –` `kuyruk = []`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 5→2 (w = 4): 0 + 4 < ∞ → koşul **doğru**.

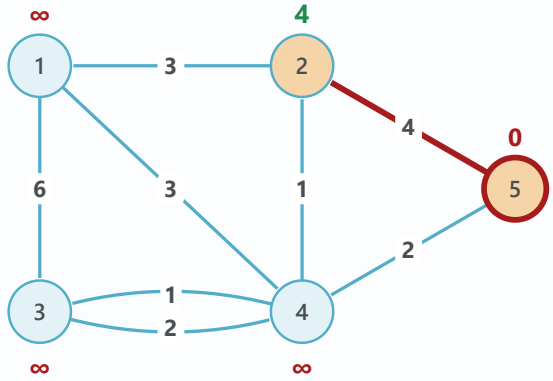
Adım 4 / 27

DEĞİŞKENLER

u = 5 | v = 2 | w = 4 | kuyruk = []

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
▶        kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



uzaklik[2] = 4, onceki[2] = 5; (2, 4) kuyruğa eklenir.

Adım 5 / 27

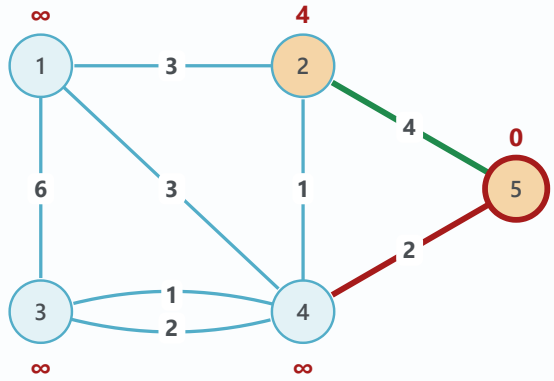
DEĞİŞKENLER

`u = 5` `v = 2` `w = 4`

`kuyruk = [(2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 5→4 (w = 2): 0 + 2 < ∞ → koşul **doğru**.

Adım 6 / 27

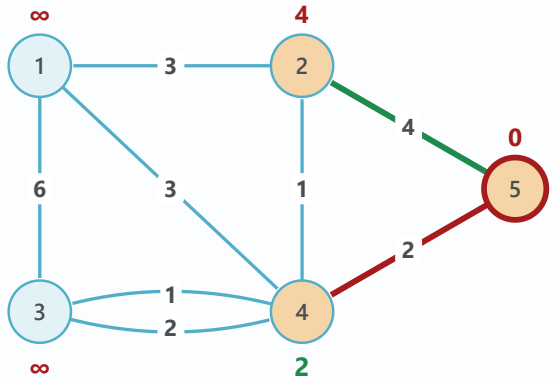
DEĞİŞKENLER

`u = 5` `v = 4` `w = 2`

`kuyruk = [(2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
▶        kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



uzaklik[4] = 2, onceki[4] = 5; (4, 2) kuyruğa eklenir.

Adım 7 / 27

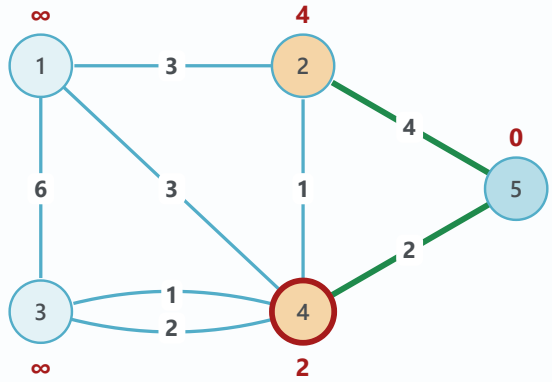
DEĞİŞKENLER

`u = 5` `v = 4` `w = 2`

`kuyruk = [(4,2), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
▶      int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruktan en küçük uzaklıklı kayıt **(4, 2)** alınır: u = 4.

Adım 8 / 27

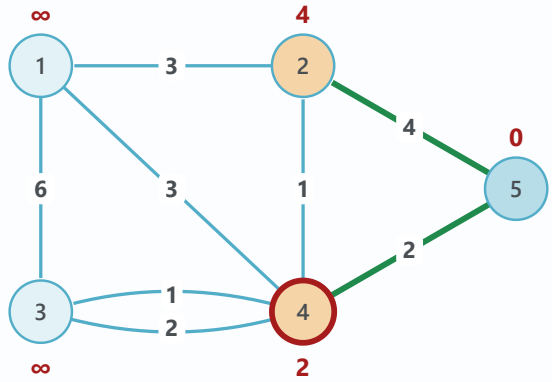
DEĞİŞKENLER

`u = 4` `v = –` `w = –`

`kuyruk = [(2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
▶      ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



4 kesinleşir: ziyaret[4] = true.

Adım 9 / 27

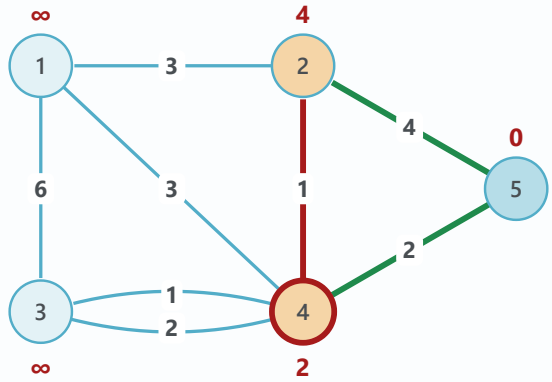
DEĞİŞKENLER

`u = 4` `v = –` `w = –`

`kuyruk = [(2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 4→2 (w = 1): 2 + 1 < 4 → koşul **doğru**.

Adım 10 / 27

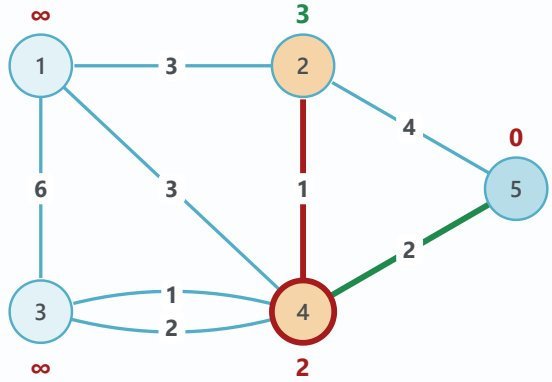
DEĞİŞKENLER

`u = 4` `v = 2` `w = 1`

`kuyruk = [(2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
▶        kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



uzaklik[2] = 3, onceki[2] = 4; (2, 3) kuyruğa eklenir.

Adım 11 / 27

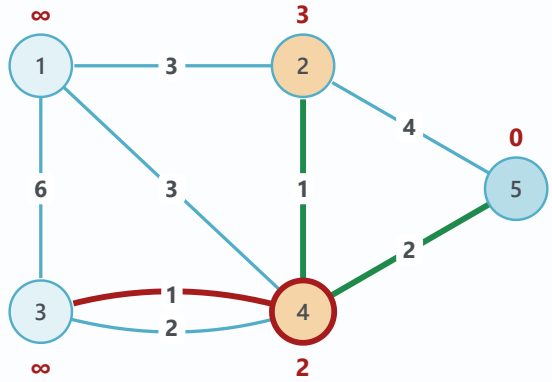
DEĞİŞKENLER

`u = 4` `v = 2` `w = 1`

`kuyruk = [(2,3), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 4→3 (w = 1): 2 + 1 < ∞ → koşul **doğru**.

Adım 12 / 27

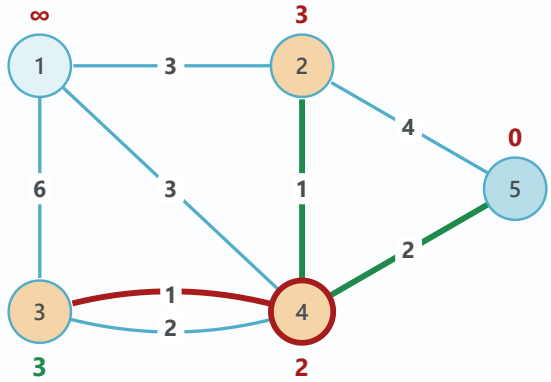
DEĞİŞKENLER

`u = 4` `v = 3` `w = 1`

`kuyruk = [(2,3), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
▶        kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



uzaklik[3] = 3, onceki[3] = 4; (3, 3) kuyruğa eklenir.

Adım 13 / 27

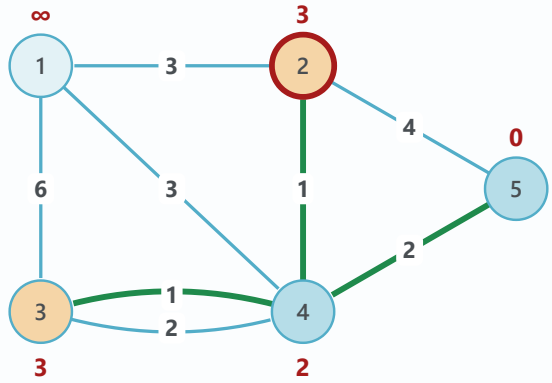
DEĞİŞKENLER

`u = 4` `v = 3` `w = 1`

`kuyruk = [(2,3), (3,3), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
▶      int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruktan en küçük uzaklıklı kayıt **(2, 3)** alınır: u = 2.

Adım 14 / 27

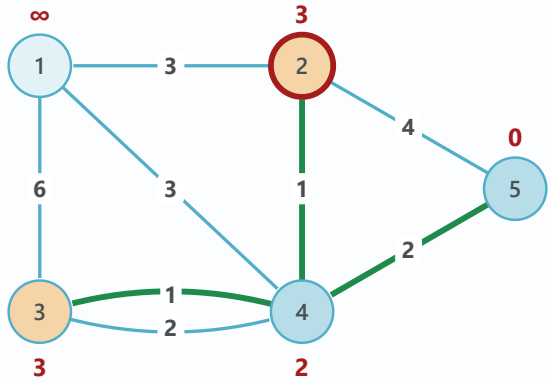
DEĞİŞKENLER

`u = 2` `v = –` `w = –`

`kuyruk = [(3,3), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
▶      ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



2 kesinleşir: ziyaret[2] = true.

Adım 15 / 27

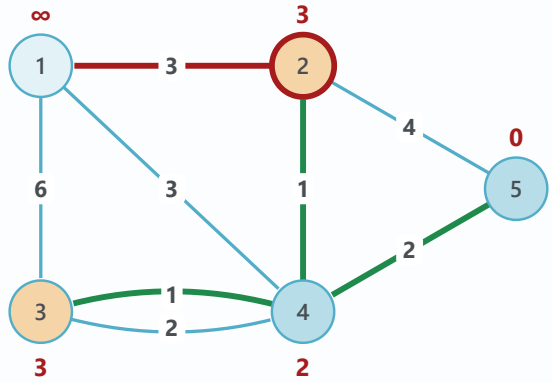
DEĞİŞKENLER

`u = 2` `v = –` `w = –`

`kuyruk = [(3,3), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 2→1 (w = 3): 3 + 3 < ∞ → koşul **doğru**.

Adım 16 / 27

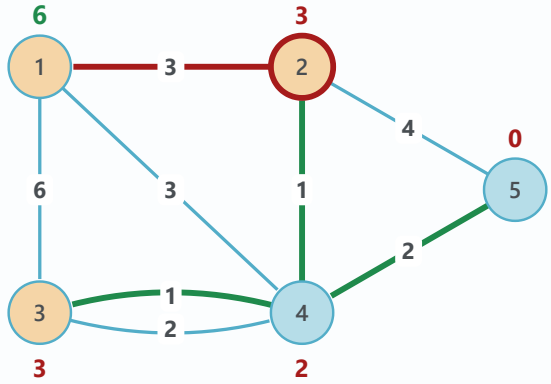
DEĞİŞKENLER

`u = 2` `v = 1` `w = 3`

`kuyruk = [(3,3), (2,4)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
▶        kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



uzaklik[1] = 6, onceki[1] = 2; (1, 6) kuyruğa eklenir.

Adım 17 / 27

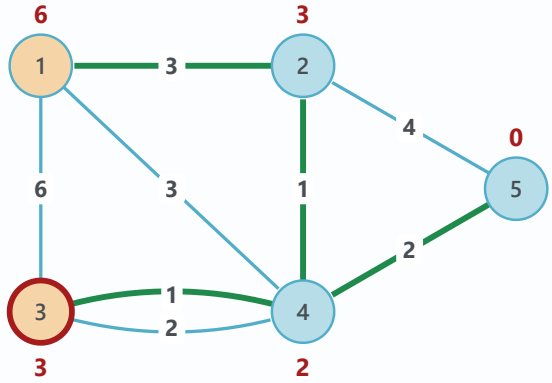
DEĞİŞKENLER

`u = 2` `v = 1` `w = 3`

`kuyruk = [(3,3), (2,4), (1,6)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
▶      int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruktan en küçük uzaklıklı kayıt **(3, 3)** alınır: u = 3.

Adım 18 / 27

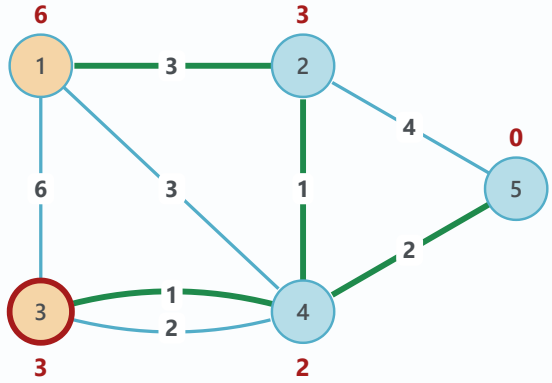
DEĞİŞKENLER

`u = 3` `v = –` `w = –`

`kuyruk = [(2,4), (1,6)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
▶      ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



3 kesinleşir: ziyaret[3] = true.

Adım 19 / 27

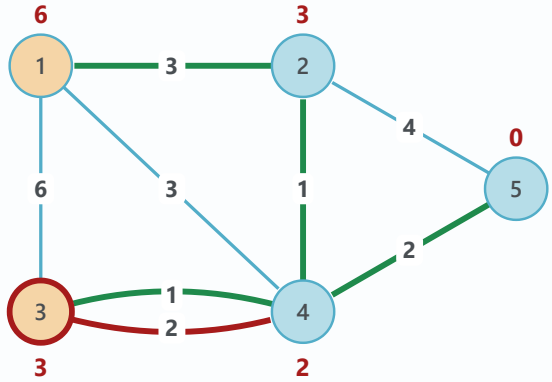
DEĞİŞKENLER

`u = 3` `v = –` `w = –`

`kuyruk = [(2,4), (1,6)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 3→4 (w = 2): ziyaret[4] = true → koşul **yanlış**.

Adım 20 / 27

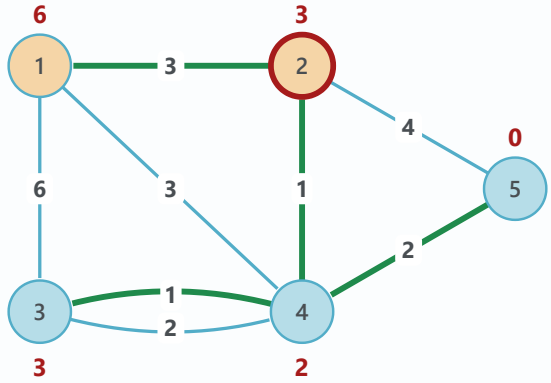
DEĞİŞKENLER

u = 3 | v = 4 | w = 2

kuyruk = [(2,4), (1,6)]

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
▶      int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruktan en küçük uzaklıklı kayıt **(2, 4)** alınır: u = 2.

Adım 21 / 27

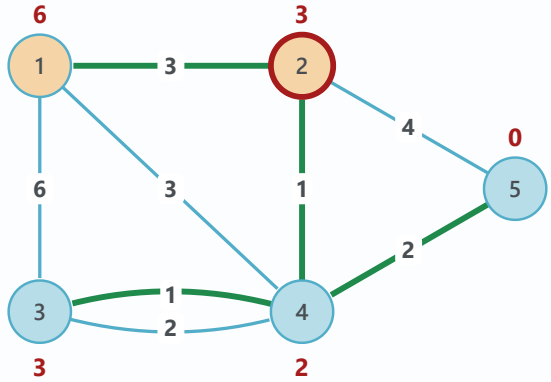
DEĞİŞKENLER

`u = 2` `v = –` `w = –`

`kuyruk = [(1,6)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
▶      if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



ziyaret[2] zaten true: bu, 2'nin eski (2, 4) kaydı → `continue` ile atlanır (tembel silme).

Adım 22 / 27

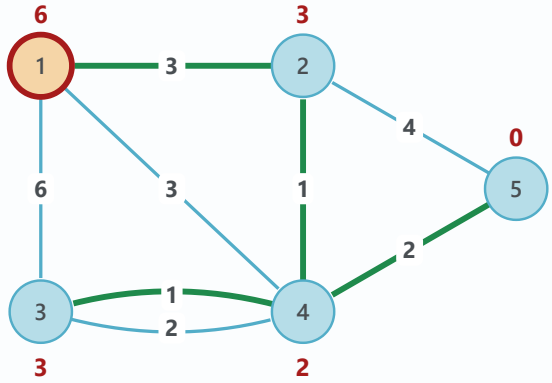
DEĞİŞKENLER

`u = 2` `v = –` `w = –`

`kuyruk = [(1,6)]`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
▶      int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruktan en küçük uzaklıklı kayıt **(1, 6)** alınır: u = 1.

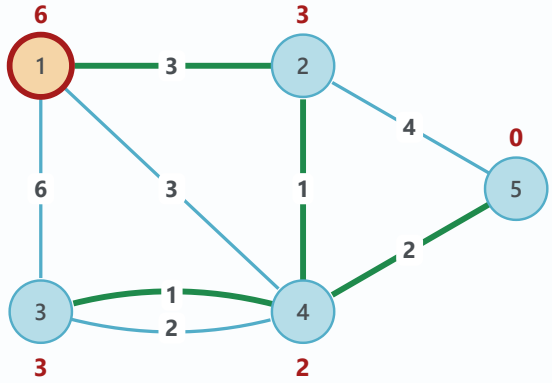
Adım 23 / 27

DEĞİŞKENLER

`u = 1` `v = –` `w = –` `kuyruk = []`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
▶      ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



1 kesinleşir: ziyaret[1] = true.

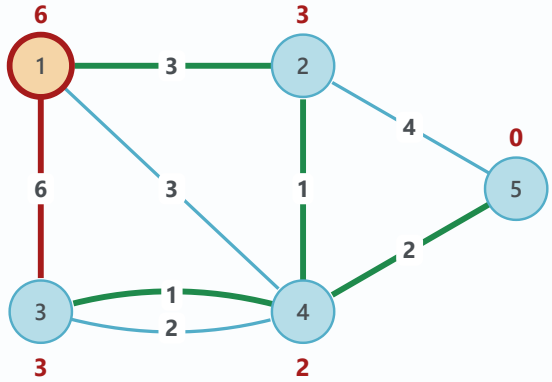
Adım 24 / 27

DEĞİŞKENLER

`u = 1` `v = –` `w = –` `kuyruk = []`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 1→3 (w = 6): ziyaret[3] = true → koşul **yanlış**.

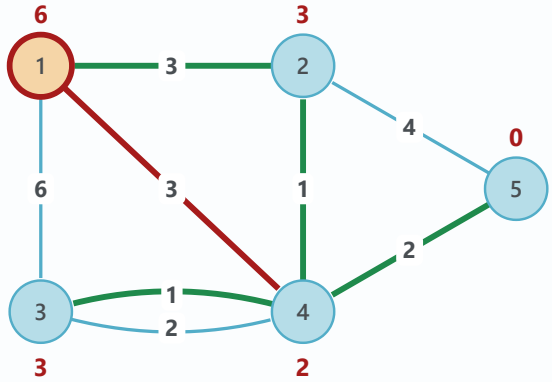
Adım 25 / 27

DEĞİŞKENLER

u = 1 | v = 3 | w = 6 | kuyruk = []

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
10   while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
▶        int v = k.hedef, w = k.agirlik;
▶        if (!ziyaret[v]
▶            && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kenar 1→4 (w = 3): ziyaret[4] = true → koşul **yanlış**.

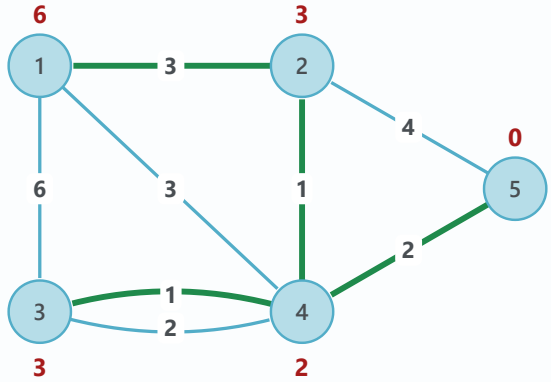
Adım 26 / 27

DEĞİŞKENLER

`u = 1` `v = 4` `w = 3` `kuyruk = []`

Dijkstra (Öncelikli Kuyruk): Kodu Adım Adım İzle

```
1  void dijkstra(int kaynak) {
2    int[] uzaklik = new int[V + 1];
3    int[] onceki = new int[V + 1];
4    boolean[] ziyaret = new boolean[V + 1];
5    Arrays.fill(uzaklik, SONSUZ);
6    uzaklik[kaynak] = 0;
7    PriorityQueue<int[]> kuyruk =
8      new PriorityQueue<>((a, b) -> a[1] - b[1]);
9    kuyruk.add(new int[] { kaynak, 0 });
▶    while (!kuyruk.isEmpty()) {
11     int u = kuyruk.poll()[0];
12     if (ziyaret[u]) continue; // eski kayıt
13     ziyaret[u] = true;
14     for (Kenar k : komsu.get(u)) {
15       int v = k.hedef, w = k.agirlik;
16       if (!ziyaret[v]
17           && uzaklik[u] + w < uzaklik[v]) {
18         uzaklik[v] = uzaklik[u] + w;
19         onceki[v] = u;
20         kuyruk.add(new int[] { v, uzaklik[v] });
21       }
22     }
23   }
24 }
```



Kuyruk boş: döngü biter. uzaklik = [6, 3, 3, 2, 0], onceki = [2, 4, 4, 5, –].

Adım 27 / 27

DEĞİŞKENLER

`u = –` `v = –` `w = –` `kuyruk = []`

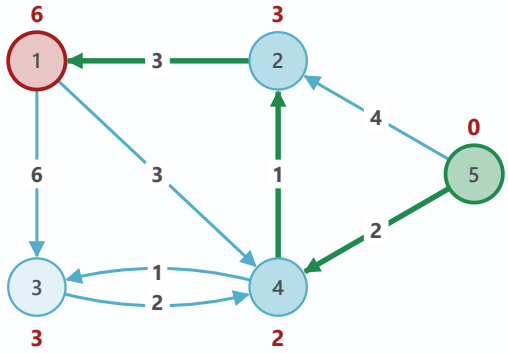
Yolu Geri Kurma: π (önceki) Dizisi

Örnek 2'nin sonucu: π = [2, 4, 4, 5, /]. Hedef 1 için geriye doğru:

1 ← π[1] = **2** ← π[2] = **4** ← π[4] = **5** (kaynak)

Ters çevrilince yol: **5** → **4** → **2** → **1**, uzunluk 2 + 1 + 3 = **6** = d[1].

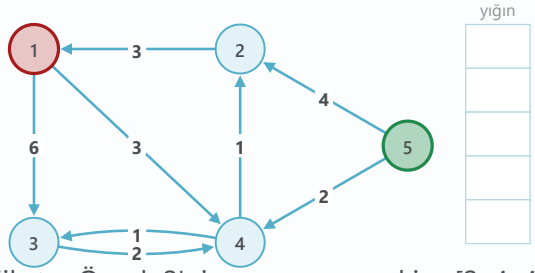
```
void yolYaz(int[] onceki, int kaynak, int hedef) {
  Deque<Integer> yigin = new ArrayDeque<>();
  for (int v = hedef; v != kaynak; v = onceki[v])
    yigin.push(v);          // hedeften geriye
  yigin.push(kaynak);
  while (!yigin.isEmpty())
    System.out.print(yigin.pop() + " ");
}
```



Yığın (stack) geriye doğru toplanan düğümleri doğru sıraya çevirir.

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
▶   Deque<Integer> yigin = new ArrayDeque<>();
3   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
6   while (!yigin.isEmpty())
7     System.out.print(yigin.pop() + " ");
8 }
```



Dijkstra Örnek 2'nin sonucu: onceki = [2, 4, 4, 5, /] (kaynak 5, hedef 1, d[1] = 6). Boş bir yığın oluşturuldu; yol hedeften geriye doğru toplanacak.

Adım 1 / 10

DEĞİŞKENLER

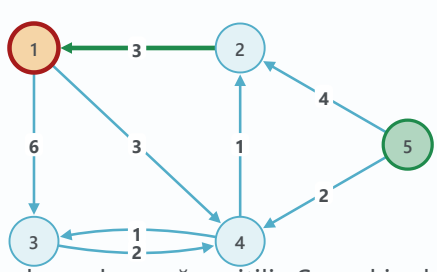
`v = –` `yigin = []`

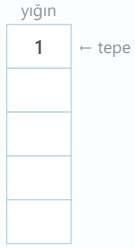
`onceki = [2, 4, 4, 5, /]`

ÇIKTI

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
▶   for (int v = hedef; v != kaynak; v = onceki[v])
▶     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
6   while (!yigin.isEmpty())
7     System.out.print(yigin.pop() + " ");
8 }
```





1 ≠ kaynak → yığına itilir. Sonraki adımda v = onceki[1] = 2.

Adım 2 / 10

DEĞİŞKENLER

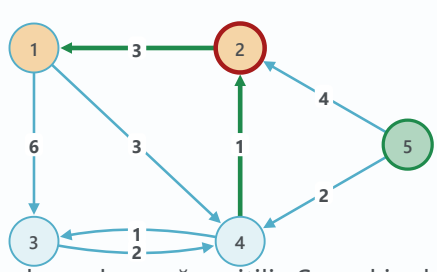
`v = 1` `yigin = [1]`

`onceki = [2, 4, 4, 5, /]`

ÇIKTI

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
▶   for (int v = hedef; v != kaynak; v = onceki[v])
▶     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
6   while (!yigin.isEmpty())
7     System.out.print(yigin.pop() + " ");
8 }
```





2 ≠ kaynak → yığına itilir. Sonraki adımda v = onceki[2] = 4.

Adım 3 / 10

DEĞİŞKENLER

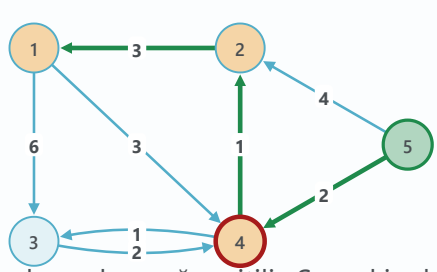
`v = 2` `yigin = [2, 1]`

`onceki = [2, 4, 4, 5, /]`

ÇIKTI

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
▶   for (int v = hedef; v != kaynak; v = onceki[v])
▶     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
6   while (!yigin.isEmpty())
7     System.out.print(yigin.pop() + " ");
8 }
```



yığın

4 ← tepe
2
1

4 ≠ kaynak → yığına itilir. Sonraki adımda v = onceki[4] = 5.

Adım 4 / 10

DEĞİŞKENLER

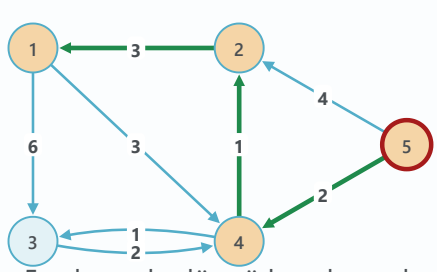
`v = 4` `yigin = [4, 2, 1]`

`onceki = [2, 4, 4, 5, /]`

ÇIKTI

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
▶   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
6   while (!yigin.isEmpty())
7     System.out.print(yigin.pop() + " ");
8 }
```



v = 5 = kaynak: döngü koşulu yanlış, döngü biter. Kaynağın kendisi henüz yığında değil.

Adım 5 / 10

DEĞİŞKENLER

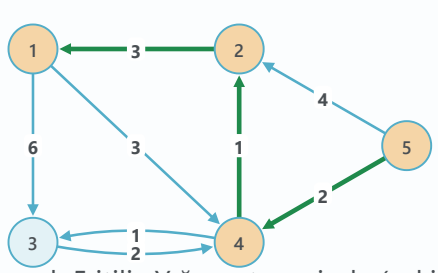
`v = 5` `yigin = [4, 2, 1]`

`onceki = [2, 4, 4, 5, /]`

ÇIKTI

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
3   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
▶   yigin.push(kaynak);
6   while (!yigin.isEmpty())
7     System.out.print(yigin.pop() + " ");
8 }
```



Kaynak 5 itilir. Yığının tepesinde (şekilde en üstte) kaynak, dibinde hedef var: sıra kendiliğinden düzeldi.

Adım 6 / 10

DEĞİŞKENLER

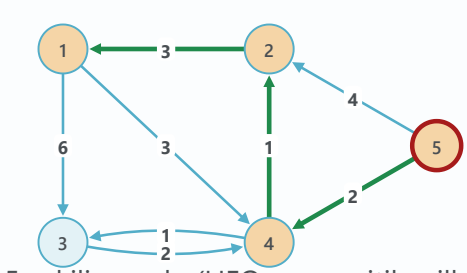
`v = –` `yigin = [5, 4, 2, 1]`

`onceki = [2, 4, 4, 5, /]`

ÇIKTI

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
3   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
▶   while (!yigin.isEmpty())
▶     System.out.print(yigin.pop() + " ");
8 }
```



5 çekilip yazılır (LIFO: en son itilen ilk çıkar).

Adım 7 / 10

DEĞİŞKENLER

`v = –` `yigin = [4, 2, 1]`

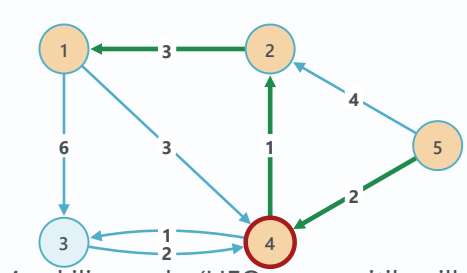
`onceki = [2, 4, 4, 5, /]`

ÇIKTI

```
5
```

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
3   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
▶   while (!yigin.isEmpty())
▶     System.out.print(yigin.pop() + " ");
8 }
```



4 çekilip yazılır (LIFO: en son itilen ilk çıkar).

Adım 8 / 10

DEĞİŞKENLER

`v = –` `yigin = [2, 1]`

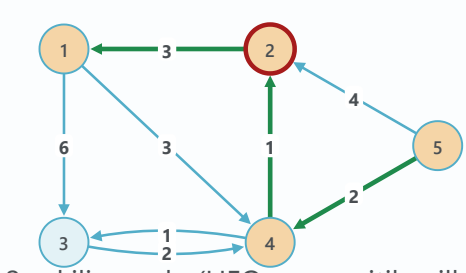
`onceki = [2, 4, 4, 5, /]`

ÇIKTI

```
5 4
```

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
3   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
▶   while (!yigin.isEmpty())
▶     System.out.print(yigin.pop() + " ");
8 }
```



2 çekilip yazılır (LIFO: en son itilen ilk çıkar).

Adım 9 / 10

DEĞİŞKENLER

`v = –` `yigin = [1]`

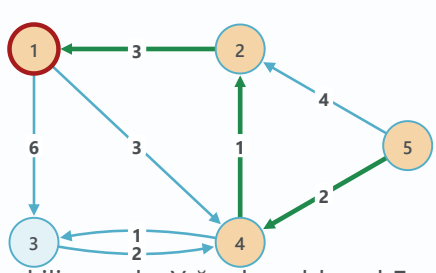
`onceki = [2, 4, 4, 5, /]`

ÇIKTI

```
5 4 2
```

Yolu Geri Kurma: `yolYaz(onceki, 5, 1)` İzleme

```
1 void yolYaz(int[] onceki, int kaynak, int hedef) {
2   Deque<Integer> yigin = new ArrayDeque<>();
3   for (int v = hedef; v != kaynak; v = onceki[v])
4     yigin.push(v);          // hedeften geriye
5   yigin.push(kaynak);
▶   while (!yigin.isEmpty())
▶     System.out.print(yigin.pop() + " ");
8 }
```



1 çekilip yazılır. Yığın boşaldı: yol **5** → **4** → **2** → **1**, uzunluk 2 + 1 + 3 = 6.

Adım 10 / 10

DEĞİŞKENLER

`v = –` `yigin = []`

`onceki = [2, 4, 4, 5, /]`

ÇIKTI

```
5 4 2 1
```

Dijkstra: Algoritma Karmaşıklığı

Maliyet, **en küçük uzaklıklı düğümü seçme** işleminin nasıl yapıldığına bağlıdır:

Seçim yapısı	Seçim	Gevşetme	Toplam	Uygun çizge
Dizi (doğrusal tarama)	$V \times O(V)$	$E \times O(1)$	$\mathbf{O(V^2)}$	yoğun ($E \approx V^2$)
İkili heap (öncelikli kuyruk)	$O(\log V)$	$O(\log V)$	$\mathbf{O((V + E) \log V)}$	seyrek ($E \ll V^2$)
Fibonacci heap	$O(\log V)$ amortize	$O(1)$ amortize	$\mathbf{O(E + V \log V)}$	kuramsal en iyi

Sayısal örnek

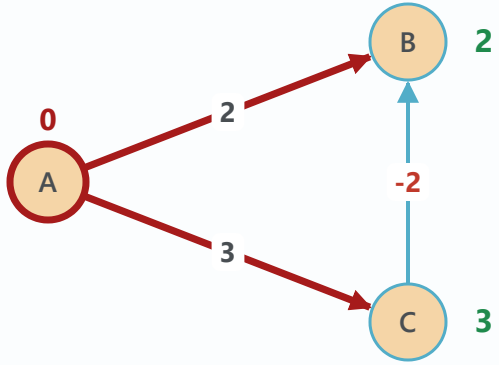
$V = 10^4$, $E = 5 \cdot 10^4$ (yol ağı): dizi $\approx 10^8$ işlem, heap $\approx 6 \cdot 10^4 \cdot 14 \approx 8{,}4 \cdot 10^5$ işlem.

Bellek

d, π ve ziyaret dizileri $O(V)$; tembel silmeli kuyrukta en fazla E kayıt → $O(V + E)$.

Dijkstra Neden Negatif Kenarda Yanılır?

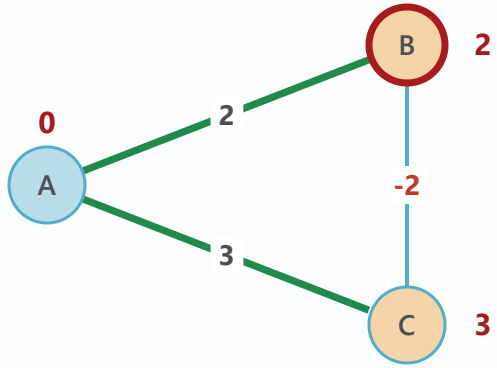
- Doğruluk kanıtı şuna dayanır: kesinleşen u için, S dışından geçen her yol en az d[u] uzunluğundadır; çünkü kenarlar **negatif değildir** ve yol yalnızca uzayabilir.
- Negatif kenar bu varsayımı bozar: sonradan işlenen bir düğüm, kesinleşmiş bir düğüme daha kısa yol açabilir.
- Tüm ağırlıklara sabit eklemek de **çözüm değildir**: çok kenarlı yollar daha fazla ceza alır, en kısa yol değişir.
- Negatif kenar varsa **Bellman-Ford** kullanılır.



A kesinleşir; gevşetme ile d[B] = 2, d[C] = 3.

Dijkstra Neden Negatif Kenarda Yanılır?

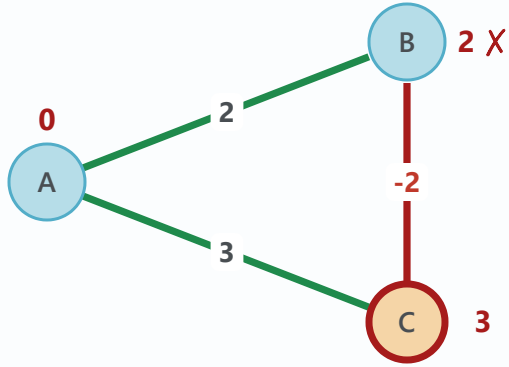
- Doğruluk kanıtı şuna dayanır: kesinleşen u için, S dışından geçen her yol en az d[u] uzunluğundadır; çünkü kenarlar **negatif değildir** ve yol yalnızca uzayabilir.
- Negatif kenar bu varsayımı bozar: sonradan işlenen bir düğüm, kesinleşmiş bir düğüme daha kısa yol açabilir.
- Tüm ağırlıklara sabit eklemek de **çözüm değildir**: çok kenarlı yollar daha fazla ceza alır, en kısa yol değişir.
- Negatif kenar varsa **Bellman-Ford** kullanılır.



En küçük d: B (2). Dijkstra B'yi **kesinleştirir**: “B'ye daha kısa yol olamaz” varsayımı.

Dijkstra Neden Negatif Kenarda Yanılır?

- Doğruluk kanıtı şuna dayanır: kesinleşen u için, S dışından geçen her yol en az d[u] uzunluğundadır; çünkü kenarlar **negatif değildir** ve yol yalnızca uzayabilir.
- Negatif kenar bu varsayımı bozar: sonradan işlenen bir düğüm, kesinleşmiş bir düğüme daha kısa yol açabilir.
- Tüm ağırlıklara sabit eklemek de **çözüm değildir**: çok kenarlı yollar daha fazla ceza alır, en kısa yol değişir.
- Negatif kenar varsa **Bellman-Ford** kullanılır.



C işlenir: d[C] + (−2) = 1 < 2, ama B zaten S'de. Sonuç d[B] = 2 **yanlış**; gerçek δ(A, B) = 1 (A → C → B).

Bellman-Ford Algoritması

- **Kaynak** düğümden diğer tüm düğümlere olan en kısa yolları bulur (tek kaynaklı).
- Richard **Bellman** (1958) ve Lester **Ford** Jr. (1956) adlarını taşır; E. F. Moore da 1959'da aynı yöntemi verdi.
- **Negatif ağırlıklı kenarları** işleyebilir.
- Kaynaktan ulaşılabilen **negatif döngüleri** tespit edebilir.
- Dijkstra'dan yavaştır: O(V · E).

Algoritma ilkeleri

Her düğüm için en kısa yol **tahminlerini** tutan bir dizi kullanır. Başta tüm tahminler ∞, kaynak 0'dır. Çizgedeki **tüm kenarlar** teker teker incelenir ve tahminler güncellenir; bu tarama V − 1 kez tekrarlanır. Düğüm seçimi yoktur, açgözlü değildir: bir **dinamik programlamadır**, i. tur en fazla i kenarlı en kısa yolları önceki turun sonuçlarından kurar.

Bellman-Ford: Algoritma Adımları

Adım 1: Başlangıç düğümüne uzaklık **0**, diğer düğümlere **∞** atanır.

π[v] = tanımsız.

Bellman-Ford: Algoritma Adımları

Adım 1: Başlangıç düğümüne uzaklık **0**, diğer düğümlere **∞** atanır.

π[v] = tanımsız.

Adım 2: Kenarlar **tek tek** incelenir, düğümler arası uzaklıklar gevşetme ile güncellenir; bu tur **V − 1** kez tekrarlanır.

Bir turda kenar sırası sabittir; sıra sonucu değil, kaç turda yakınsayacağını etkiler.

Bellman-Ford: Algoritma Adımları

Adım 1: Başlangıç düğümüne uzaklık **0**, diğer düğümlere **∞** atanır.

π[v] = tanımsız.

Adım 2: Kenarlar **tek tek** incelenir, düğümler arası uzaklıklar gevşetme ile güncellenir; bu tur **V − 1** kez tekrarlanır.

Bir turda kenar sırası sabittir; sıra sonucu değil, kaç turda yakınsayacağını etkiler.

Adım 3: Bir düğümün uzaklığı güncellenirse, sonraki incelemelerde onun **komşularının** uzaklıkları da güncellenebilir.

İyileşme, yol boyunca her turda en az bir kenar ilerler.

Bellman-Ford: Algoritma Adımları

Adım 1: Başlangıç düğümüne uzaklık **0**, diğer düğümlere **∞** atanır.

π[v] = tanımsız.

Adım 2: Kenarlar **tek tek** incelenir, düğümler arası uzaklıklar gevşetme ile güncellenir; bu tur **V − 1** kez tekrarlanır.

Bir turda kenar sırası sabittir; sıra sonucu değil, kaç turda yakınsayacağını etkiler.

Adım 3: Bir düğümün uzaklığı güncellenirse, sonraki incelemelerde onun **komşularının** uzaklıkları da güncellenebilir.

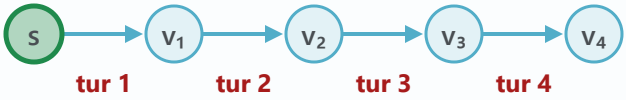
İyileşme, yol boyunca her turda en az bir kenar ilerler.

Adım 4: Negatif döngü kontrolü için tüm kenarlar **bir kez daha** incelenir; hâlâ gevşeyen kenar varsa negatif döngü vardır.

Döngü yoksa V − 1 turdan sonra hiçbir kenar gevşemez.

Neden V − 1 Tur? Karmaşıklık

- Negatif döngü yoksa her en kısa yol **en fazla V − 1 kenarlıdır**.
- **Değişmez:** i. turun sonunda, en fazla i kenarlı en kısa yolu olan her v için d[v] = δ(s, v).
- Tur i'de yolun i. kenarı mutlaka gevşetilir; V − 1 tur tüm yolları tamamlar.
- Her tur E kenarı inceler: toplam **O(V · E)** zaman, O(V) ek bellek.

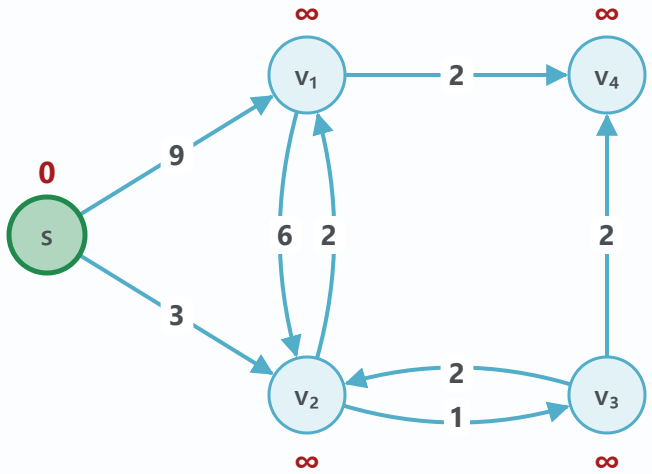


i. turun sonunda i kenarlı yol kesinleşir

Çizge	Zaman
Seyrek, E ≈ V	$O(V^2)$
Yoğun, E ≈ V^2	$O(V^3)$
En iyi durum (erken durma)	O(E)

Erken durma: bir turda hiçbir d değişmezse sonraki turlar da değiştirmez; döngüden çıkılabilir.

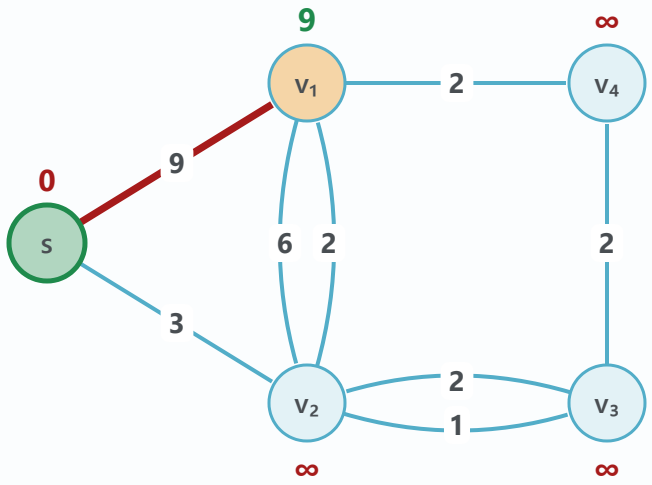
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

İlklendirme: d[s] = 0, diğerleri ∞. Kenarlar her turda yukarıdaki sırayla incelenir.

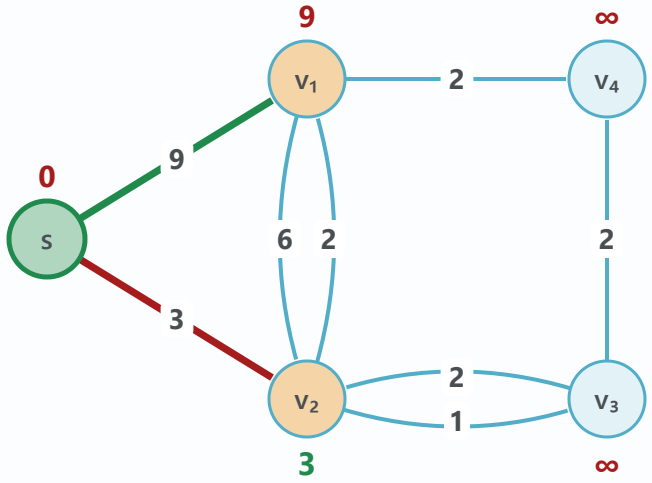
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v₁) 9 | (s,v₂) 3 | (v₁,v₂) 6 | (v₁,v₄) 2 | (v₂,v₁) 2 | (v₂,v₃) 1 | (v₃,v₂) 2 | (v₃,v₄) 2

Tur 1, kenar (s, v_1): 0 + 9 = 9 < ∞ → **d[v_1] = 9**.

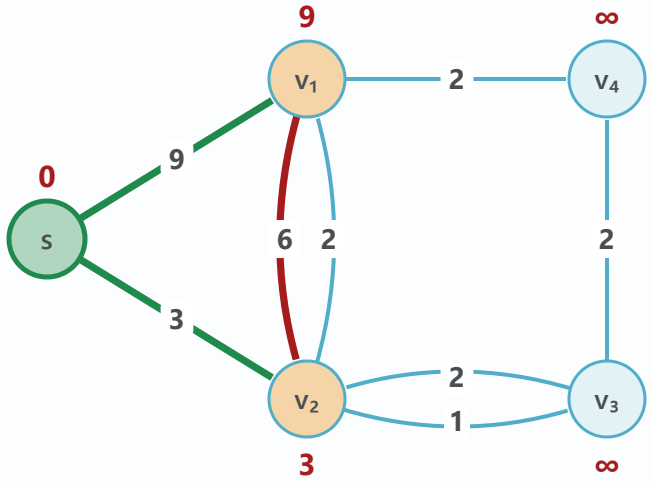
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v₁) 9 | (s,v₂) 3 | (v₁,v₂) 6 | (v₁,v₄) 2 | (v₂,v₁) 2 | (v₂,v₃) 1 | (v₃,v₂) 2 | (v₃,v₄) 2

Tur 1, kenar (s, v_2): 0 + 3 = 3 < ∞ → **d[v_2] = 3**.

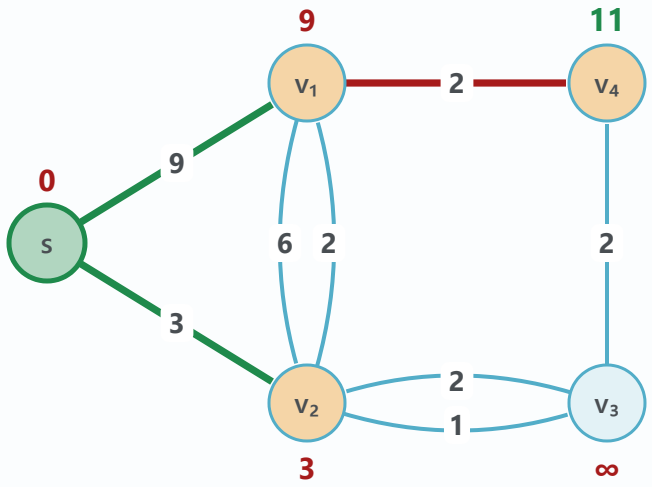
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v1) 9 | (s,v2) 3 | (v1,v2) 6 | (v1,v4) 2 | (v2,v1) 2 | (v2,v3) 1 | (v3,v2) 2 | (v3,v4) 2

Tur 1, kenar (v_1, v_2): 9 + 6 = 15 ≥ 3 → değişmez.

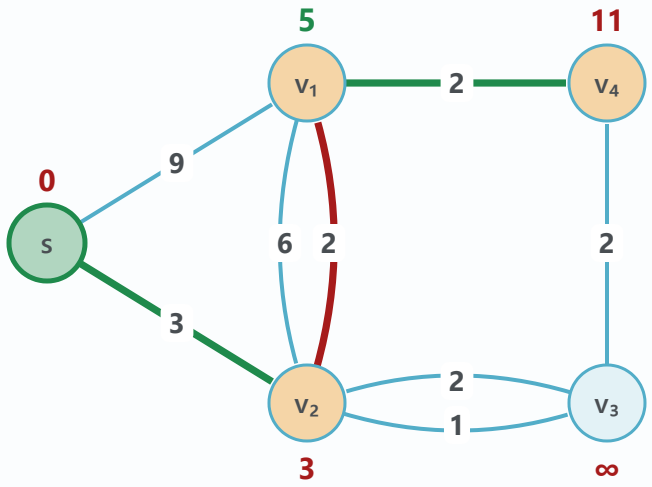
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



`(s,v₁) 9` `(s,v₂) 3` `(v₁,v₂) 6` `(v₁,v₄) 2` `(v₂,v₁) 2` `(v₂,v₃) 1` `(v₃,v₂) 2` `(v₃,v₄) 2`

Tur 1, kenar (v_1, v_4): 9 + 2 = 11 < ∞ → **d[v_4] = 11**.

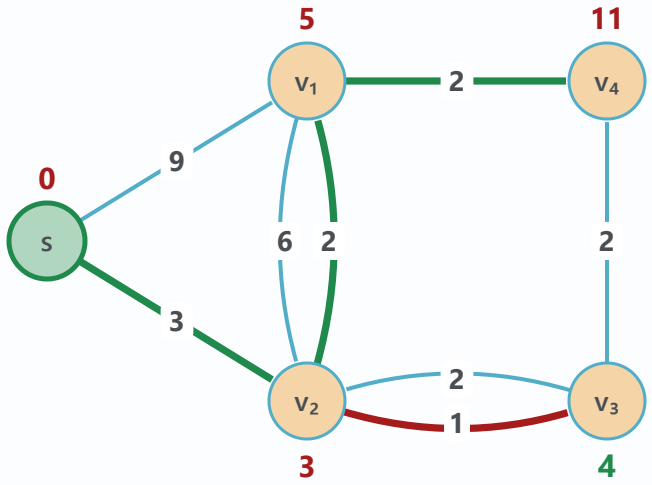
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Tur 1, kenar (v_2, v_1): 3 + 2 = 5 < 9 → **d[v_1] = 5**.

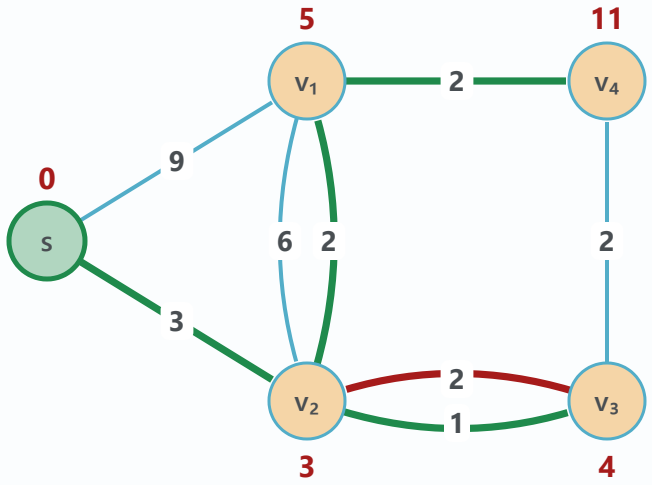
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



`(s,v₁) 9` `(s,v₂) 3` `(v₁,v₂) 6` `(v₁,v₄) 2` `(v₂,v₁) 2` `(v₂,v₃) 1` `(v₃,v₂) 2` `(v₃,v₄) 2`

Tur 1, kenar (v_2, v_3): 3 + 1 = 4 < ∞ → **d[v_3] = 4**.

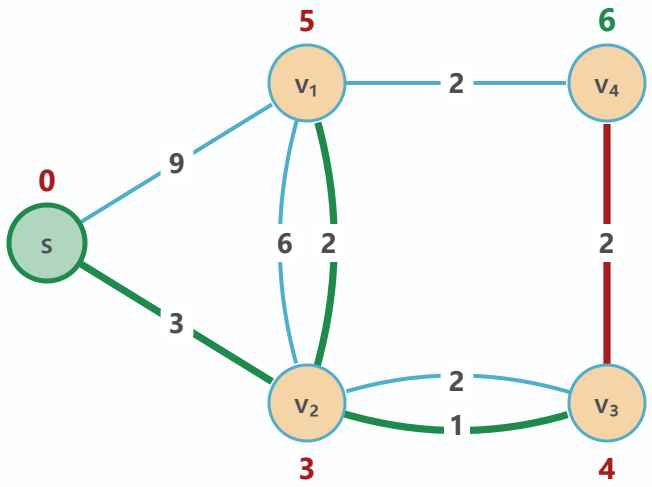
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Tur 1, kenar (v_3, v_2): 4 + 2 = 6 ≥ 3 → değişmez.

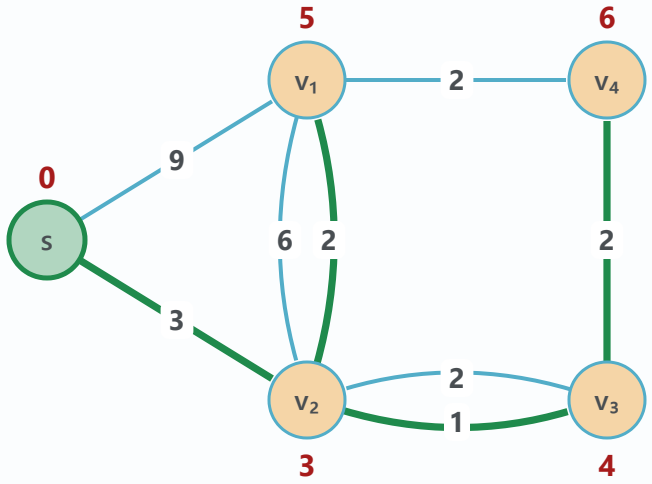
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



(s,v₁) 9 (s,v₂) 3 (v₁,v₂) 6 (v₁,v₄) 2 (v₂,v₁) 2 (v₂,v₃) 1 (v₃,v₂) 2 (v₃,v₄) 2

Tur 1, kenar (v_3, v_4): 4 + 2 = 6 < 11 → **d[v_4] = 6**.

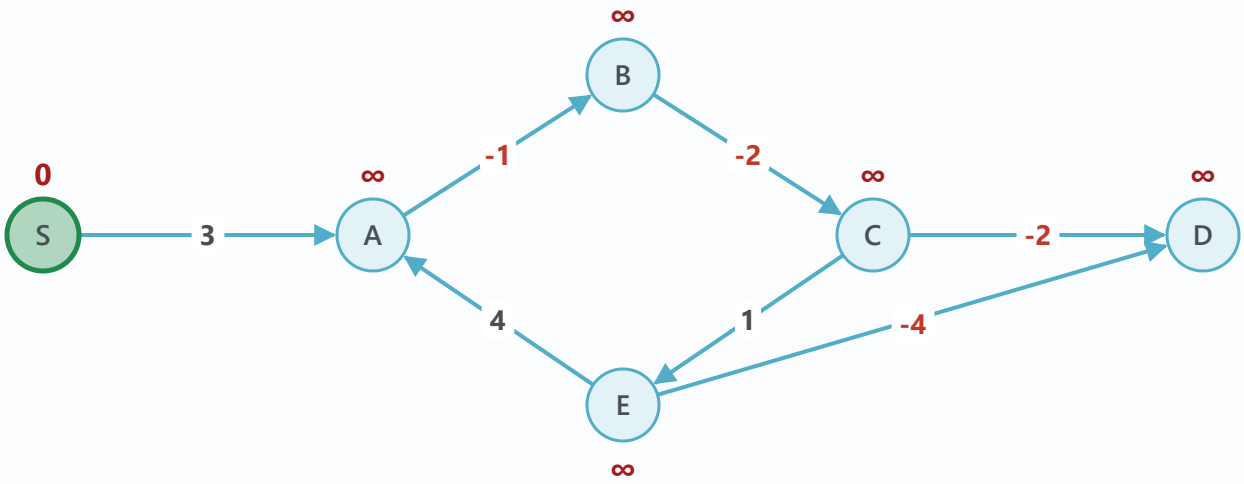
Bellman-Ford Örnek 1: Kenar Kenar Gevşetme



`(s,v₁) 9` `(s,v₂) 3` `(v₁,v₂) 6` `(v₁,v₄) 2` `(v₂,v₁) 2` `(v₂,v₃) 1` `(v₃,v₂) 2` `(v₃,v₄) 2`

Tur 2: sekiz kenarın hiçbiri gevşemez → **erken durma**. Sonuç: $v_1 = 5$, $v_2 = 3$, $v_3 = 4$, $v_4 = 6$ (bu sırada tek tur yetti).

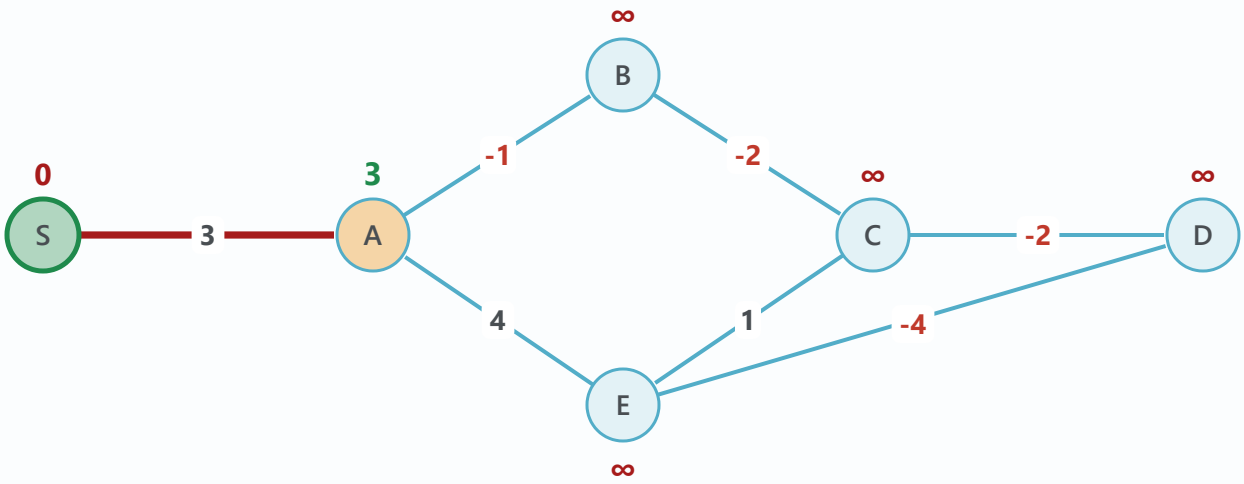
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Negatif kenarlı çizge, kaynak S. Kenarlar her turda **yukarıdaki sırayla** incelenir (sondan başa: en kötü sıra).

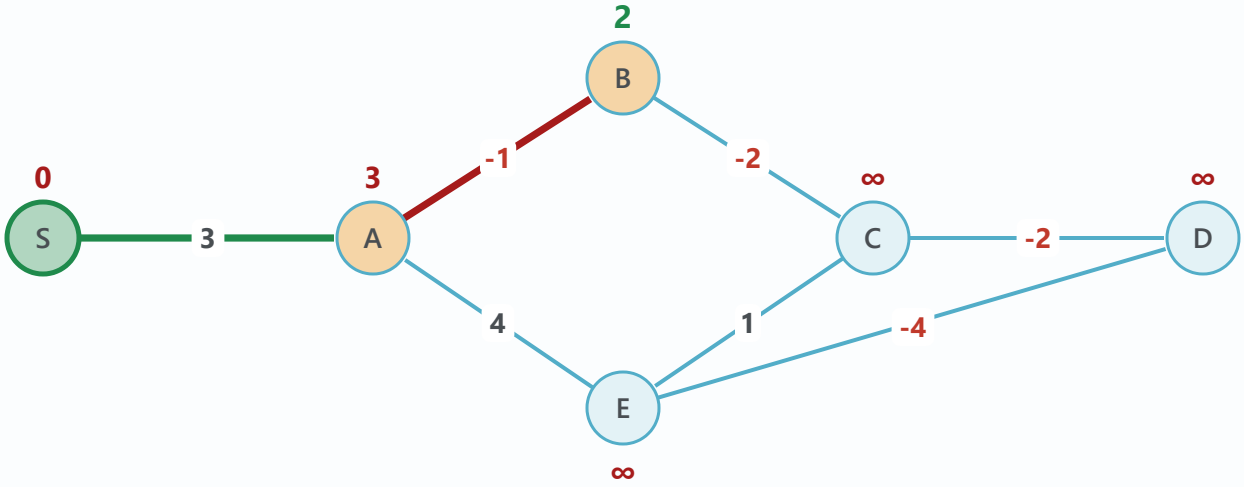
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Tur 1: yalnızca (S,A): ∞ → **3**. Diğer kenarların başı hâlâ ∞.

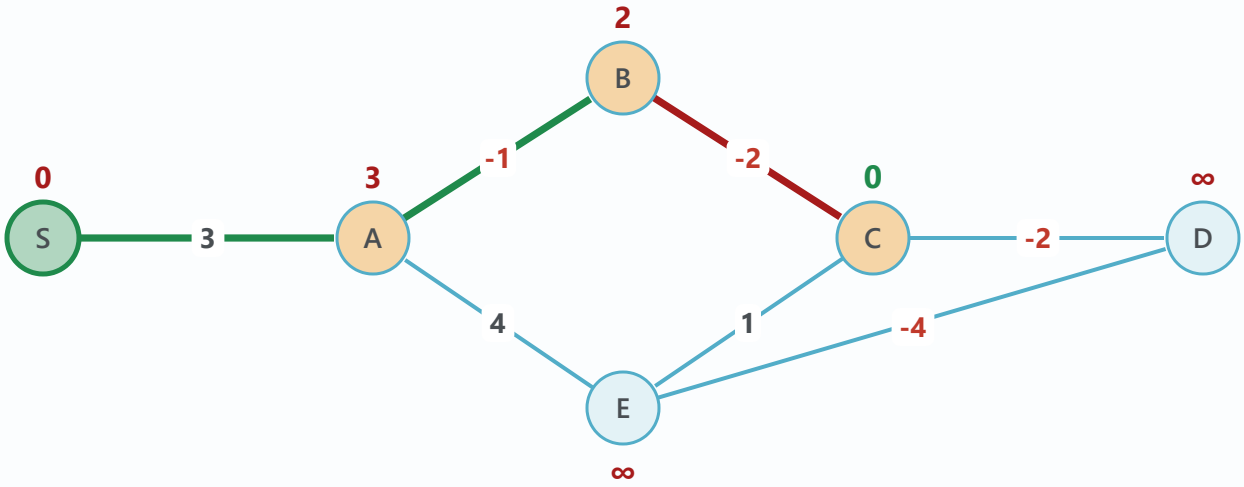
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Tur 2: yalnızca (A,B): ∞ → **2**.

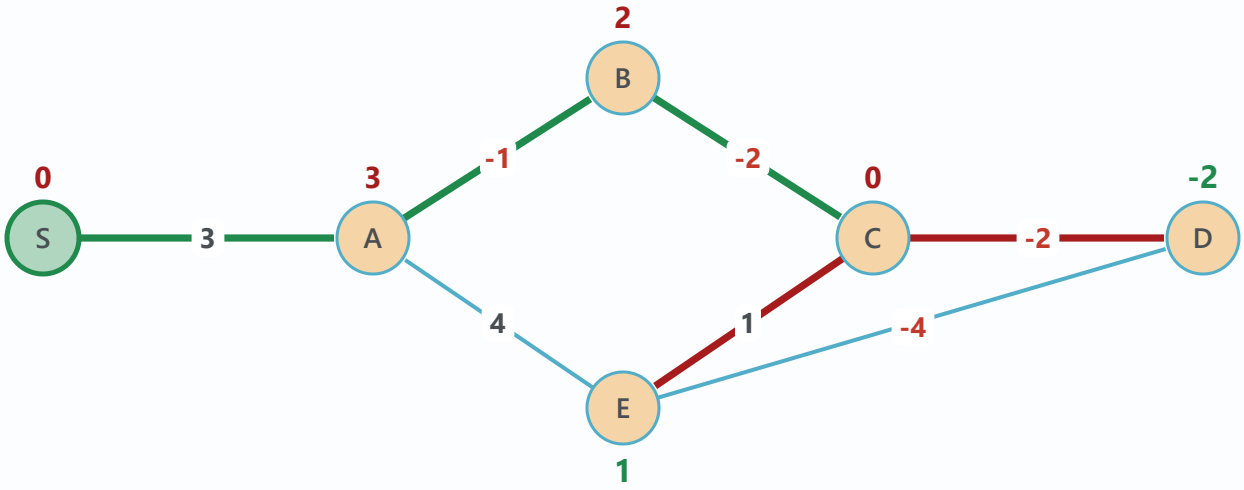
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Tur 3: yalnızca (B,C): ∞ → **0**.

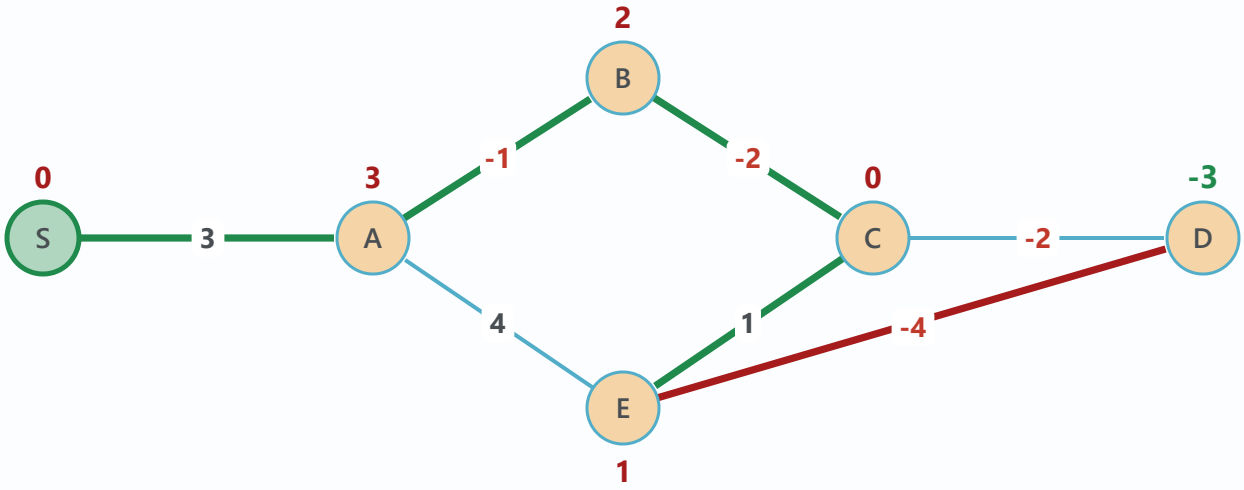
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Tur 4: yalnızca (C,D): ∞ → **-2**; (C,E): ∞ → **1**.

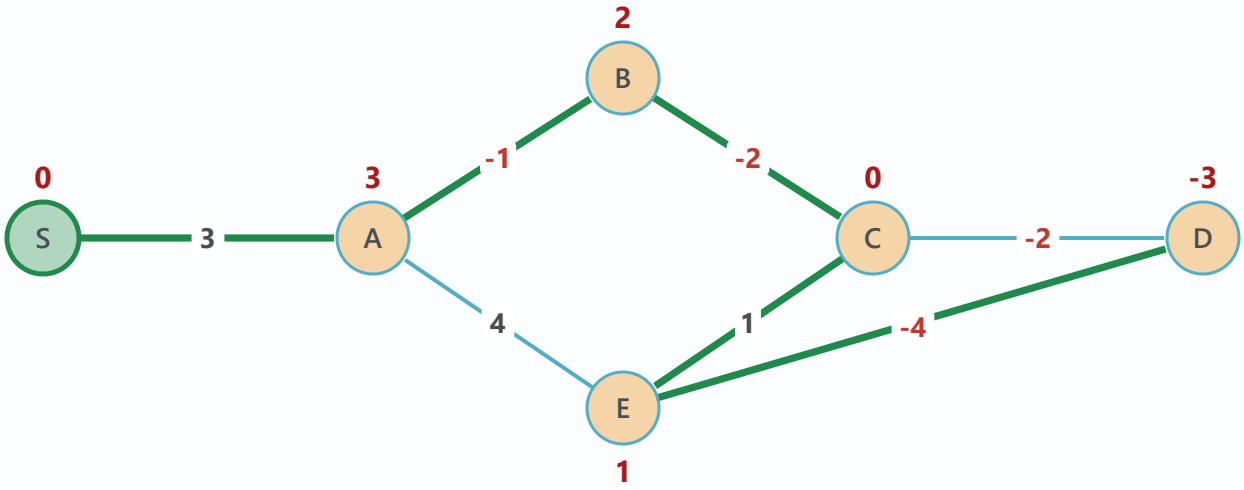
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Tur 5: yalnızca (E,D): -2 → **-3**. E üzerinden D'ye daha kısa yol: 1 + (−4) = −3.

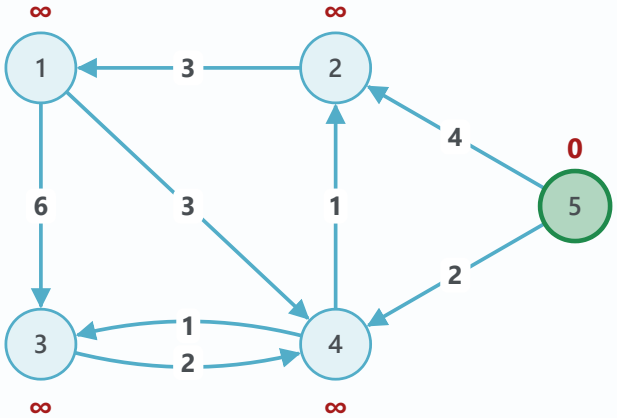
Bellman-Ford Örnek 2: Negatif Kenarlar



`(E,D) -4` `(E,A) 4` `(C,D) -2` `(C,E) 1` `(B,C) -2` `(A,B) -1` `(S,A) 3`

Denetim turu: hiçbir kenar gevşemez → negatif döngü yok (A→B→C→E→A = −1 −2 + 1 + 4 = +2). V − 1 = 5 turun hepsi gerekti.

Bellman-Ford Örnek 3: Dijkstra Örnek 2'deki Çizge

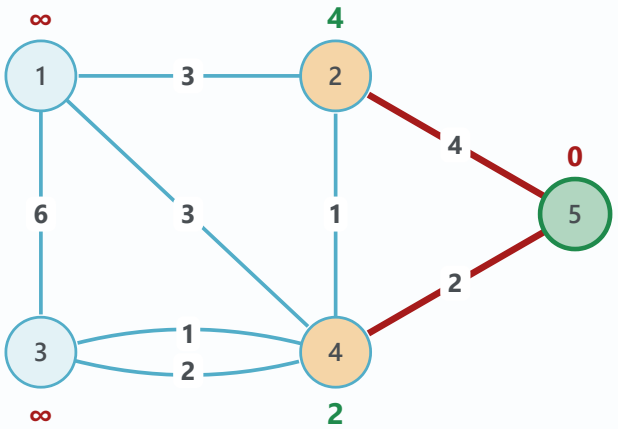


	1	2	3	4	5
d	∞	∞	∞	∞	0
π	/	/	/	/	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

İlklendirme: kaynak 5'in uzaklığı 0, diğerleri ∞. Kenar sırası: (1,3), (1,4), (2,1), (3,4), (4,2), (4,3), (5,2), (5,4).

Bellman-Ford Örnek 3: Dijkstra Örnek 2'deki Çizge



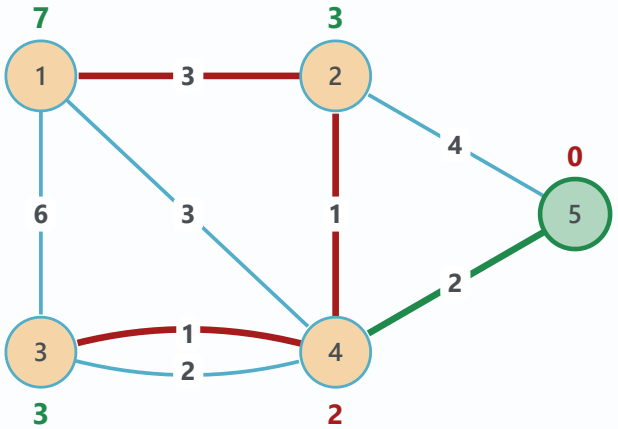
	1	2	3	4	5
d	∞	**4**	∞	**2**	0
π	/	**5**	/	**5**	/

(1,3) 6 (1,4) 3 (2,1) 3 (3,4) 2

(4,2) 1 (4,3) 1 (5,2) 4 (5,4) 2

Tur 1: (5,2): ∞ → **4**; (5,4): ∞ → **2**.

Bellman-Ford Örnek 3: Dijkstra Örnek 2'deki Çizge



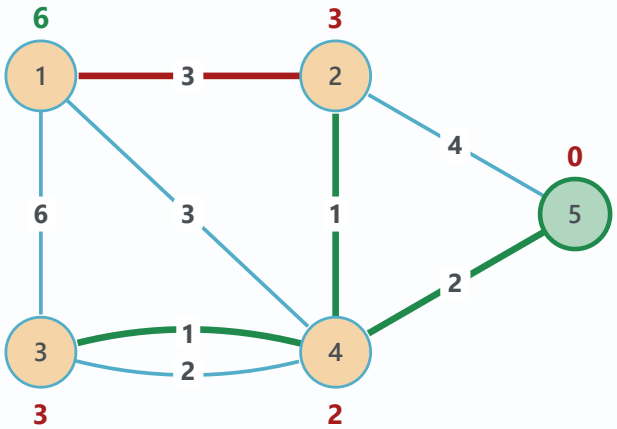
	1	2	3	4	5
d	7	3	3	2	0
π	2	4	4	5	/

(1,3) 6 (1,4) 3 (2,1) 3 (3,4) 2

(4,2) 1 (4,3) 1 (5,2) 4 (5,4) 2

Tur 2: (2,1): ∞ → **7**; (4,2): 4 → **3**; (4,3): ∞ → **3**. (4,2) kenarı 2'ye daha kısa yol buldu.

Bellman-Ford Örnek 3: Dijkstra Örnek 2'deki Çizge



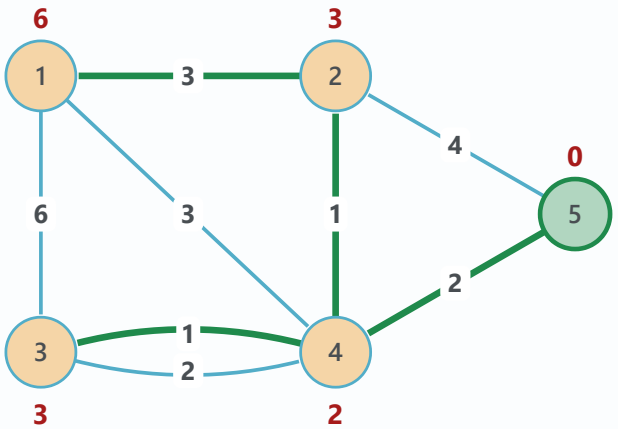
	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

(1,3) 6 (1,4) 3 (2,1) 3 (3,4) 2

(4,2) 1 (4,3) 1 (5,2) 4 (5,4) 2

Tur 3: (2,1): 7 → **6**. 2 bir önceki turda iyileştiği için (2,1) yeniden gevşer.

Bellman-Ford Örnek 3: Dijkstra Örnek 2'deki Çizge



	1	2	3	4	5
d	6	3	3	2	0
π	2	4	4	5	/

`(1,3) 6` `(1,4) 3` `(2,1) 3` `(3,4) 2`
`(4,2) 1` `(4,3) 1` `(5,2) 4` `(5,4) 2`

Tur 4: daha kısa yol bulunamadı (hiçbir kenar gevşemez).

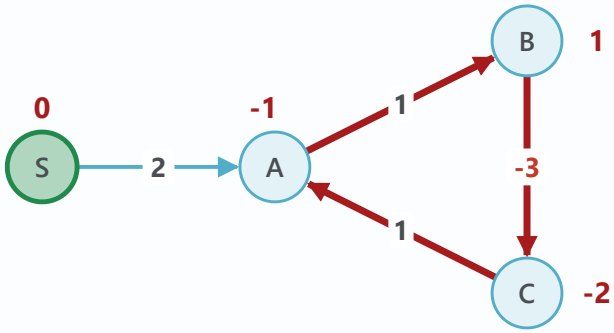
Negatif Döngü Kontrolü (V. Tur)

Her kenar için son bir kez gevşetme denenir: **d[v] > d[u] + w(u, v)** olan bir kenar kalırsa negatif döngü vardır. Örnek 3'ün sonucunda:

kenar	d[v]	d[u] + w(u, v)	sonuç
(1, 3)	d[3] = 3	d[1] + w = 6 + 6 = 12	3 ≯ 12 ✓
(1, 4)	d[4] = 2	d[1] + w = 6 + 3 = 9	2 ≯ 9 ✓
(2, 1)	d[1] = 6	d[2] + w = 3 + 3 = 6	6 ≯ 6 ✓
(3, 4)	d[4] = 2	d[3] + w = 3 + 2 = 5	2 ≯ 5 ✓
(4, 2)	d[2] = 3	d[4] + w = 2 + 1 = 3	3 ≯ 3 ✓
(4, 3)	d[3] = 3	d[4] + w = 2 + 1 = 3	3 ≯ 3 ✓
(5, 2)	d[2] = 3	d[5] + w = 0 + 4 = 4	3 ≯ 4 ✓
(5, 4)	d[4] = 2	d[5] + w = 0 + 2 = 2	2 ≯ 2 ✓

Hiçbir kenar gevşemiyor → **negatif döngü yok**; d = [6, 3, 3, 2, 0] kesindir.

Negatif Döngü Tespiti: Örnek



A → B → C → A döngüsü: 1 + (−3) + 1 = **−1**.
Kenar sırası: (S,A), (A,B), (B,C), (C,A).

tur	S	A	B	C
1	0	1	3	0
2	0	0	2	-1
3	0	-1	1	-2

- V − 1 = 3 tur sonunda d = [0, −1, 1, −2]; döngüdeki değerler her turda 1 azalıyor.
- Denetim: (A, B) için d[A] + 1 = 0 < d[B] = 1 → **negatif döngü var**.
- Döngüdeki ve döngüden ulaşılan düğümlerin en kısa yolu −∞'dur (tanımsız).

Bellman-Ford: Sözde Kod

```
BELLMAN_FORD(G, kaynak):
  mesafeler = [ ]
  her bir v için V içinde:
    mesafeler[v] = sonsuz
  mesafeler[kaynak] = 0
  döngü i = 1 den |V| - 1:
    her bir (u, v, ağırlık) için E içinde:
      eğer mesafeler[u] + ağırlık < mesafeler[v]:
        mesafeler[v] = mesafeler[u] + ağırlık
  her bir (u, v, ağırlık) için E içinde:
    eğer mesafeler[u] + ağırlık < mesafeler[v]:
      hata "Negatif ağırlıklı döngü var"
  döndür mesafeler
```

İki bölüm

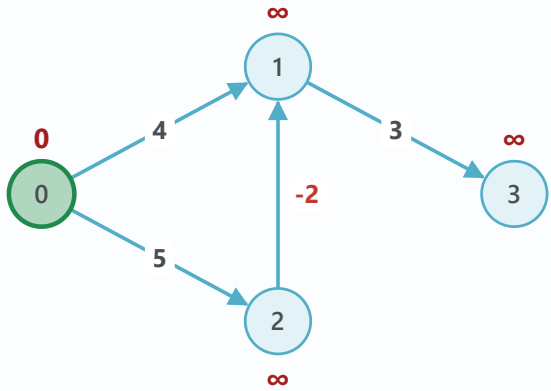
Dış döngü V − 1 tur; her turda **tüm** kenarlar. Ardından tek bir denetim turu.

Kenar listesi

Bellman-Ford düğüm seçmediği için çizgeyi bir kenar listesi E olarak tutmak yeterlidir.

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
▶    int[] uzaklik = new int[V];
▶    int[] onceki = new int[V];
▶    Arrays.fill(uzaklik, SONSUZ);
▶    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Kaynak 0. uzaklik = [0, ∞, ∞, ∞]. Kenar listesi: (2,1,−2), (0,1,4), (0,2,5), (1,3,3).

Adım 1 / 19

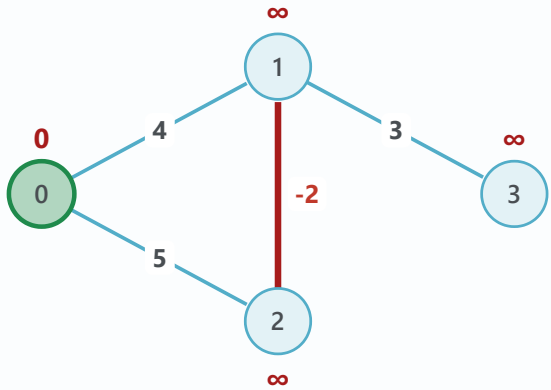
DEĞİŞKENLER

`i = –` `u = –` `v = –` `w = –`

`uzaklik = [0, ∞, ∞, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 1, kenar (2,1): uzaklik[2] = ∞ → koşul yanlış, atlanır.

Adım 2 / 19

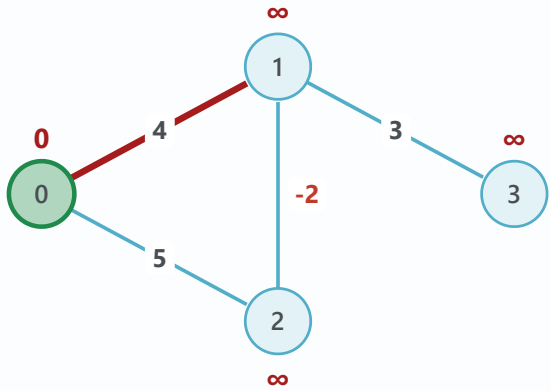
DEĞİŞKENLER

`i = 1` `u = 2` `v = 1` `w = -2`

`uzaklik = [0, ∞, ∞, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 1, kenar (0,1): 0 + (4) = 4 < ∞ → koşul doğru.

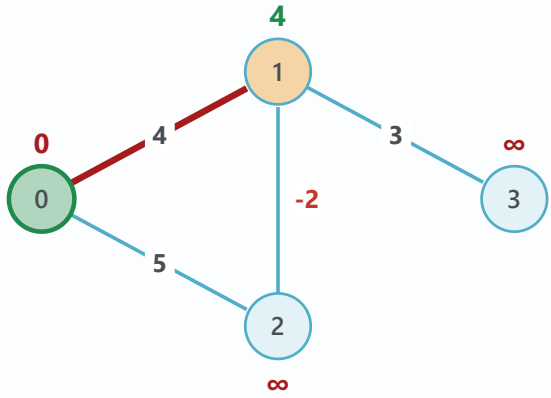
DEĞİŞKENLER

`i = 1` `u = 0` `v = 1` `w = 4`

`uzaklik = [0, ∞, ∞, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
▶          uzaklik[v] = uzaklik[u] + w;
▶          onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



uzaklik[1] = 4, onceki[1] = 0.

Adım 4 / 19

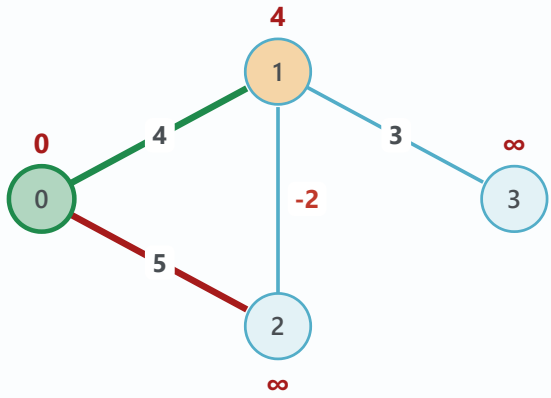
DEĞİŞKENLER

`i = 1` `u = 0` `v = 1` `w = 4`

`uzaklik = [0, 4, ∞, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 1, kenar (0,2): 0 + (5) = 5 < ∞ → koşul doğru.

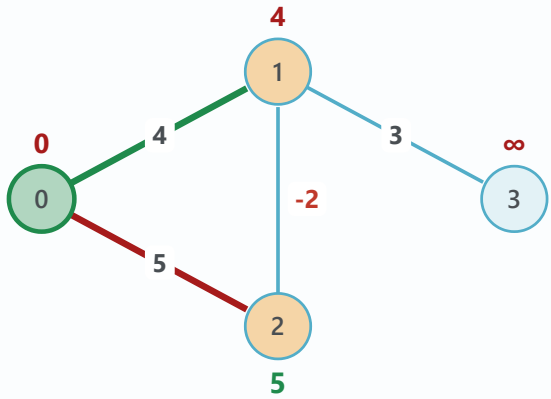
DEĞİŞKENLER

`i = 1` `u = 0` `v = 2` `w = 5`

`uzaklik = [0, 4, ∞, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
▶          uzaklik[v] = uzaklik[u] + w;
▶          onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



uzaklik[2] = 5, onceki[2] = 0.

Adım 6 / 19

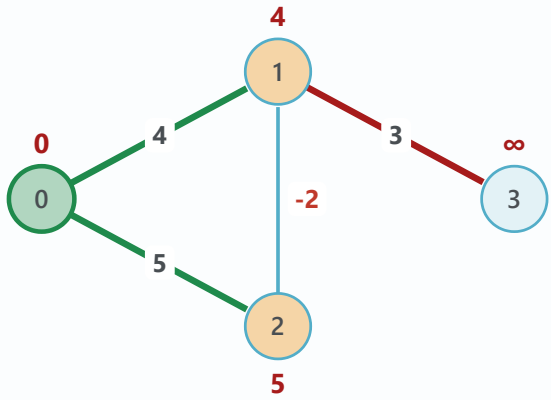
DEĞİŞKENLER

`i = 1` `u = 0` `v = 2` `w = 5`

`uzaklik = [0, 4, 5, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 1, kenar (1,3): 4 + (3) = 7 < ∞ → koşul doğru.

Adım 7 / 19

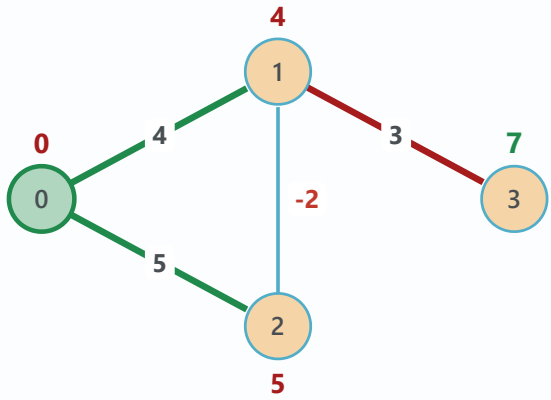
DEĞİŞKENLER

`i = 1` `u = 1` `v = 3` `w = 3`

`uzaklik = [0, 4, 5, ∞]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
▶          uzaklik[v] = uzaklik[u] + w;
▶          onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



uzaklik[3] = 7, onceki[3] = 1. Tur 1 bitti: (2,1) kenarı 2 henüz ∞ iken incelendiği için boşa geçti.

Adım 8 / 19

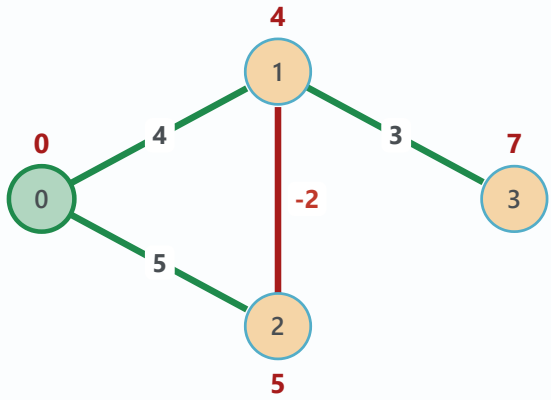
DEĞİŞKENLER

`i = 1` `u = 1` `v = 3` `w = 3`

`uzaklik = [0, 4, 5, 7]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 2, kenar (2,1): 5 + (-2) = 3 < 4 → koşul doğru.

Adım 9 / 19

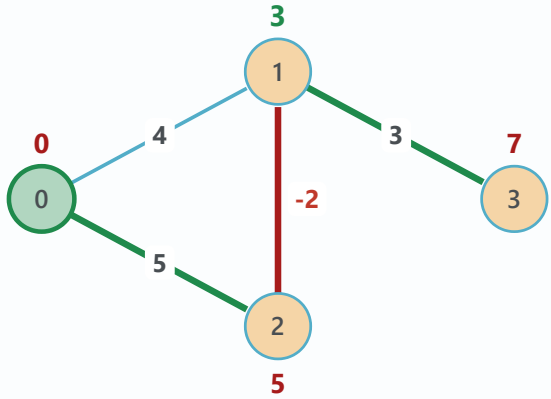
DEĞİŞKENLER

`i = 2` `u = 2` `v = 1` `w = -2`

`uzaklik = [0, 4, 5, 7]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
▶          uzaklik[v] = uzaklik[u] + w;
▶          onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



uzaklik[1] = 3, onceki[1] = 2.

Adım 10 / 19

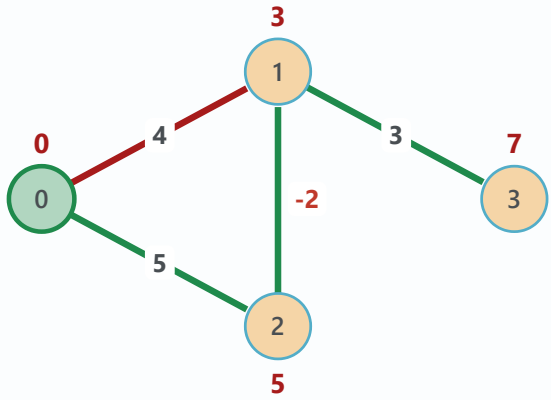
DEĞİŞKENLER

`i = 2` `u = 2` `v = 1` `w = -2`

`uzaklik = [0, 3, 5, 7]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 2, kenar (0,1): 0 + (4) = 4 ≥ 3 → değişiklik yok.

Adım 11 / 19

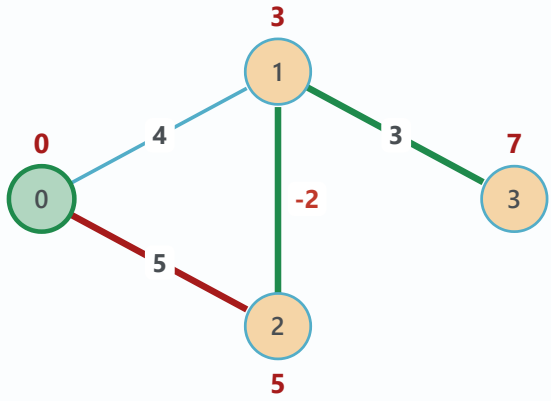
DEĞİŞKENLER

`i = 2` `u = 0` `v = 1` `w = 4`

`uzaklik = [0, 3, 5, 7]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 2, kenar (0,2): 0 + (5) = 5 ≥ 5 → değişiklik yok.

Adım 12 / 19

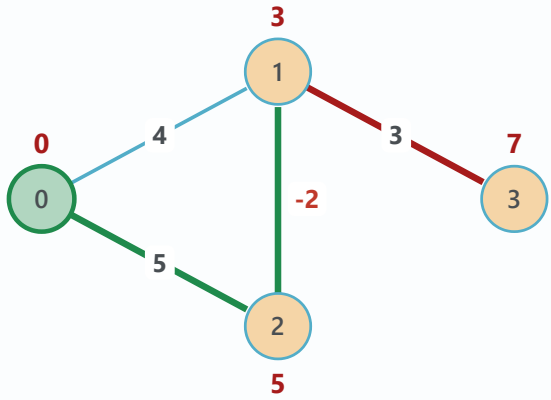
DEĞİŞKENLER

`i = 2` `u = 0` `v = 2` `w = 5`

`uzaklik = [0, 3, 5, 7]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 2, kenar (1,3): 3 + (3) = 6 < 7 → koşul doğru.

Adım 13 / 19

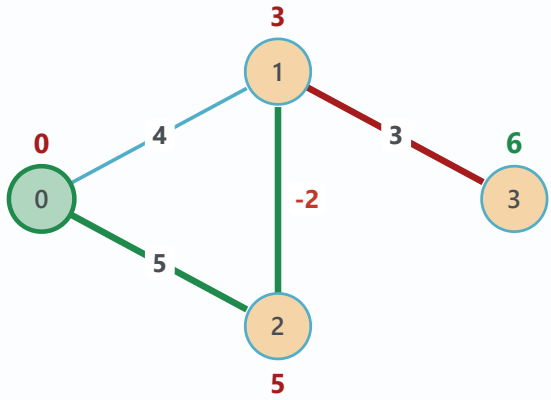
DEĞİŞKENLER

`i = 2` `u = 1` `v = 3` `w = 3`

`uzaklik = [0, 3, 5, 7]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
▶          uzaklik[v] = uzaklik[u] + w;
▶          onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



uzaklik[3] = 6, onceki[3] = 1. Tur 2 bitti: negatif kenar 1'i 3'e, o da 3'ü 6'ya indirdi.

Adım 14 / 19

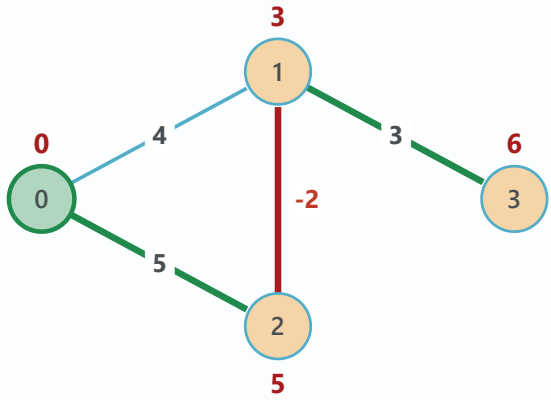
DEĞİŞKENLER

`i = 2` `u = 1` `v = 3` `w = 3`

`uzaklik = [0, 3, 5, 6]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 3, kenar (2,1): 5 + (-2) = 3 ≥ 3 → değişiklik yok.

Adım 15 / 19

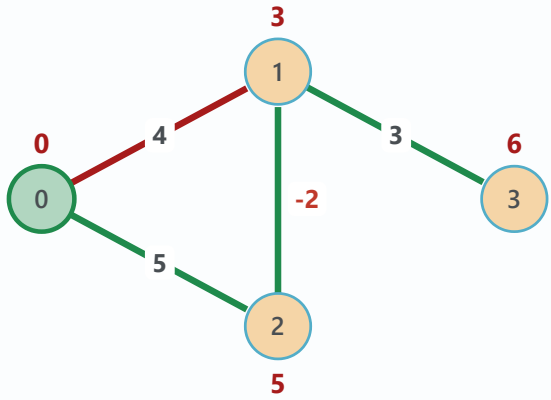
DEĞİŞKENLER

`i = 3` `u = 2` `v = 1` `w = -2`

`uzaklik = [0, 3, 5, 6]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 3, kenar (0,1): 0 + (4) = 4 ≥ 3 → değişiklik yok.

Adım 16 / 19

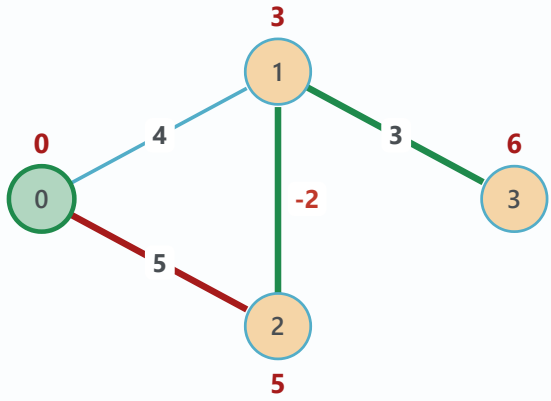
DEĞİŞKENLER

`i = 3` `u = 0` `v = 1` `w = 4`

`uzaklik = [0, 3, 5, 6]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 3, kenar (0,2): 0 + (5) = 5 ≥ 5 → değişiklik yok.

Adım 17 / 19

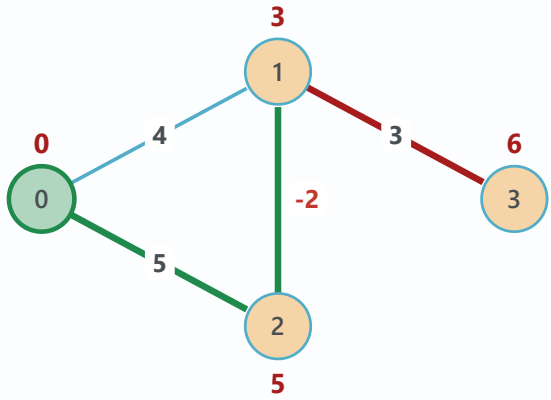
DEĞİŞKENLER

`i = 3` `u = 0` `v = 2` `w = 5`

`uzaklik = [0, 3, 5, 6]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
▶        int u = k.bas, v = k.son, w = k.agirlik;
▶        if (uzaklik[u] != SONSUZ
▶            && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
16   for (Kenar k : kenarlar) { // denetim
17     int u = k.bas, v = k.son, w = k.agirlik;
18     if (uzaklik[u] != SONSUZ
19         && uzaklik[u] + w < uzaklik[v])
20       throw new IllegalStateException(
21           "Negatif döngü");
22   }
23 }
```



Tur 3, kenar (1,3): 3 + (3) = 6 ≥ 6 → değişiklik yok.

Adım 18 / 19

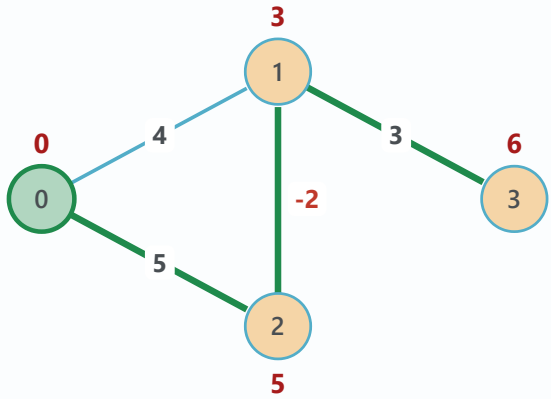
DEĞİŞKENLER

`i = 3` `u = 1` `v = 3` `w = 3`

`uzaklik = [0, 3, 5, 6]`

Bellman-Ford: Kodu Adım Adım İzle

```
1  void bellmanFord(int kaynak) {
2    int[] uzaklik = new int[V];
3    int[] onceki = new int[V];
4    Arrays.fill(uzaklik, SONSUZ);
5    uzaklik[kaynak] = 0;
6    for (int i = 1; i < V; i++) {
7      for (Kenar k : kenarlar) {
8        int u = k.bas, v = k.son, w = k.agirlik;
9        if (uzaklik[u] != SONSUZ
10           && uzaklik[u] + w < uzaklik[v]) {
11         uzaklik[v] = uzaklik[u] + w;
12         onceki[v] = u;
13       }
14     }
15   }
▶  for (Kenar k : kenarlar) { // denetim
▶    int u = k.bas, v = k.son, w = k.agirlik;
▶    if (uzaklik[u] != SONSUZ
▶        && uzaklik[u] + w < uzaklik[v])
20     throw new IllegalStateException(
21         "Negatif döngü");
22   }
23 }
```



Tur 3 hiçbir şeyi değiştirmedi. Denetim turu (satır 16–20): hiçbir kenar gevşemez, istisna fırlatılmaz. Sonuç [0, 3, 5, 6].

Adım 19 / 19

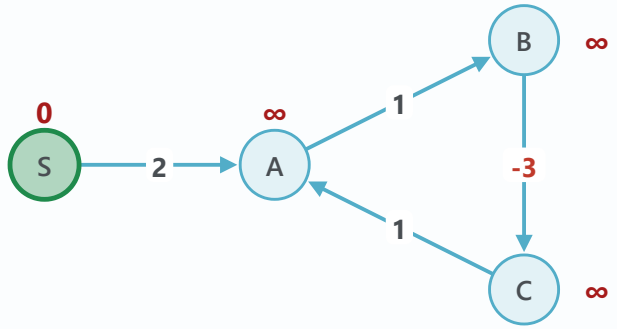
DEĞİŞKENLER

`i = –` `u = 1` `v = 3` `w = 3`

`uzaklik = [0, 3, 5, 6]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
▶ for (int i = 1; i < V; i++) {
2   for (Kenar k : kenarlar) {
3     int u = k.bas, v = k.son, w = k.agirlik;
4     if (uzaklik[u] != SONSUZ
5         && uzaklik[u] + w < uzaklik[v]) {
6       uzaklik[v] = uzaklik[u] + w;
7       onceki[v] = u;
8     }
9   }
10 }
11 for (Kenar k : kenarlar) { // denetim
12   int u = k.bas, v = k.son, w = k.agirlik;
13   if (uzaklik[u] != SONSUZ
14       && uzaklik[u] + w < uzaklik[v])
15     throw new IllegalStateException(
16         "Negatif döngü");
17 }
```



başlangıç

S → A (2) ve A → B → C → A döngüsü: 1 + (−3) + 1 = −1. uzaklik = [S, A, B, C] = [0, ∞, ∞, ∞]; kenarlar SA, AB, BC, CA sırasıyla gevşetilecek.

Adım 1 / 9

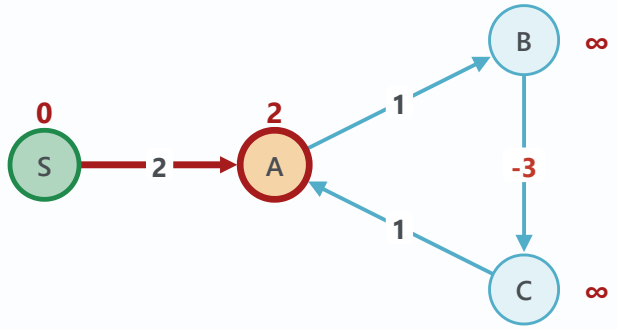
DEĞİŞKENLER

`i = –` `u = –` `v = –` `w = –`

`uzaklik = [0, ∞, ∞, ∞]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
1  for (int i = 1; i < V; i++) {
2    for (Kenar k : kenarlar) {
▶      int u = k.bas, v = k.son, w = k.agirlik;
▶      if (uzaklik[u] != SONSUZ
▶          && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
8      }
9    }
10 }
11 for (Kenar k : kenarlar) { // denetim
12   int u = k.bas, v = k.son, w = k.agirlik;
13   if (uzaklik[u] != SONSUZ
14       && uzaklik[u] + w < uzaklik[v])
15     throw new IllegalStateException(
16         "Negatif döngü");
17 }
```



tur 1

Tur 1, kenar SA: 0 + (2) = 2 daha küçük → uzaklik[A] = 2.

Adım 2 / 9

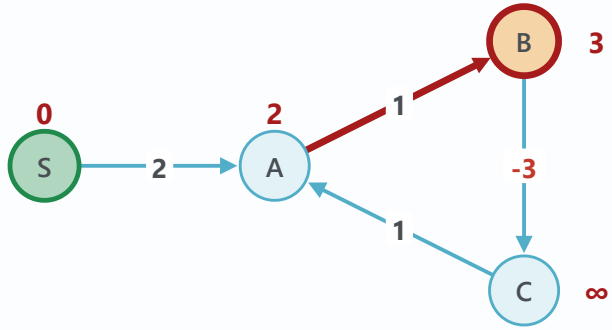
DEĞİŞKENLER

`i = 1` `u = S` `v = A` `w = 2`

`uzaklik = [0, 2, ∞, ∞]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
1  for (int i = 1; i < V; i++) {
2    for (Kenar k : kenarlar) {
▶      int u = k.bas, v = k.son, w = k.agirlik;
▶      if (uzaklik[u] != SONSUZ
▶          && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
8      }
9    }
10 }
11 for (Kenar k : kenarlar) { // denetim
12   int u = k.bas, v = k.son, w = k.agirlik;
13   if (uzaklik[u] != SONSUZ
14       && uzaklik[u] + w < uzaklik[v])
15     throw new IllegalStateException(
16         "Negatif döngü");
17 }
```



tur 1

Tur 1, kenar AB: 2 + (1) = 3 daha küçük → uzaklik[B] = 3.

Adım 3 / 9

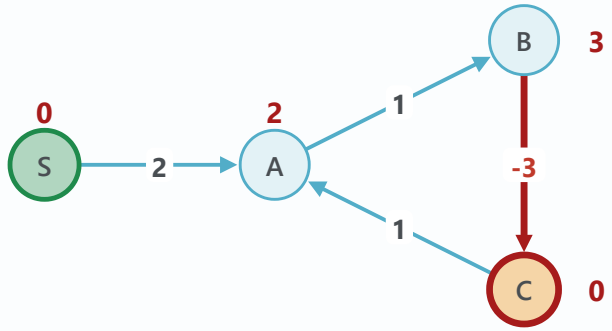
DEĞİŞKENLER

`i = 1` `u = A` `v = B` `w = 1`

`uzaklik = [0, 2, 3, ∞]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
1  for (int i = 1; i < V; i++) {
2    for (Kenar k : kenarlar) {
▶      int u = k.bas, v = k.son, w = k.agirlik;
▶      if (uzaklik[u] != SONSUZ
▶          && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
8      }
9    }
10 }
11 for (Kenar k : kenarlar) { // denetim
12   int u = k.bas, v = k.son, w = k.agirlik;
13   if (uzaklik[u] != SONSUZ
14       && uzaklik[u] + w < uzaklik[v])
15     throw new IllegalStateException(
16         "Negatif döngü");
17 }
```



tur 1

Tur 1, kenar BC: 3 + (-3) = 0 daha küçük → uzaklik[C] = 0.

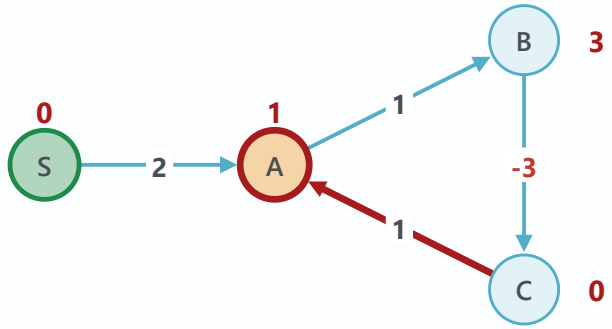
DEĞİŞKENLER

`i = 1` `u = B` `v = C` `w = -3`

`uzaklik = [0, 2, 3, 0]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
1  for (int i = 1; i < V; i++) {
2    for (Kenar k : kenarlar) {
▶      int u = k.bas, v = k.son, w = k.agirlik;
▶      if (uzaklik[u] != SONSUZ
▶          && uzaklik[u] + w < uzaklik[v]) {
▶        uzaklik[v] = uzaklik[u] + w;
▶        onceki[v] = u;
8      }
9    }
10 }
11 for (Kenar k : kenarlar) { // denetim
12   int u = k.bas, v = k.son, w = k.agirlik;
13   if (uzaklik[u] != SONSUZ
14       && uzaklik[u] + w < uzaklik[v])
15     throw new IllegalStateException(
16         "Negatif döngü");
17 }
```



tur 1

Tur 1, kenar CA: 0 + (1) = 1 daha küçük → uzaklik[A] = 1. A, döngü üzerinden dolaşınca 2'den 1'e indi.

Adım 5 / 9

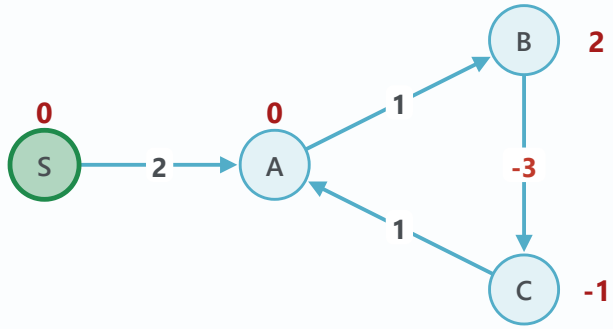
DEĞİŞKENLER

`i = 1` `u = C` `v = A` `w = 1`

`uzaklik = [0, 1, 3, 0]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
for (int i = 1; i < V; i++) {
  for (Kenar k : kenarlar) {
    int u = k.bas, v = k.son, w = k.agirlik;
    if (uzaklik[u] != SONSUZ
        && uzaklik[u] + w < uzaklik[v]) {
      uzaklik[v] = uzaklik[u] + w;
      onceki[v] = u;
    }
  }
}
for (Kenar k : kenarlar) { // denetim
  int u = k.bas, v = k.son, w = k.agirlik;
  if (uzaklik[u] != SONSUZ
      && uzaklik[u] + w < uzaklik[v])
    throw new IllegalStateException(
        "Negatif döngü");
}
```



tur 2

Tur 2 bitti: dört kenardan üçü yine gevşedi (B = 2, C = -1, A = 0). Döngüdeki değerler her turda 1 azalıyor.

Adım 6 / 9

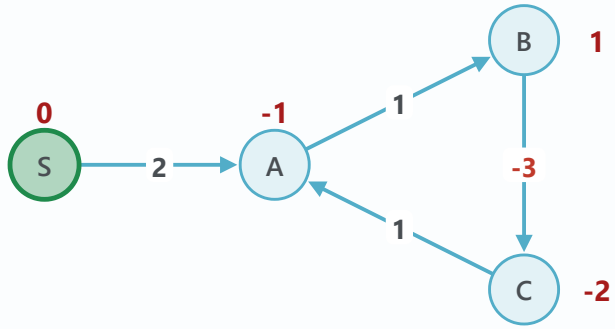
DEĞİŞKENLER

`i = 2` `u = C` `v = A` `w = 1`

`uzaklik = [0, 0, 2, -1]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
for (int i = 1; i < V; i++) {
  for (Kenar k : kenarlar) {
    int u = k.bas, v = k.son, w = k.agirlik;
    if (uzaklik[u] != SONSUZ
        && uzaklik[u] + w < uzaklik[v]) {
      uzaklik[v] = uzaklik[u] + w;
      onceki[v] = u;
    }
  }
}
for (Kenar k : kenarlar) { // denetim
  int u = k.bas, v = k.son, w = k.agirlik;
  if (uzaklik[u] != SONSUZ
      && uzaklik[u] + w < uzaklik[v])
    throw new IllegalStateException(
        "Negatif döngü");
}
```



tur 3

Tur 3 bitti: dört kenardan üçü yine gevşedi (B = 1, C = -2, A = -1). Döngüdeki değerler her turda 1 azalıyor.

Adım 7 / 9

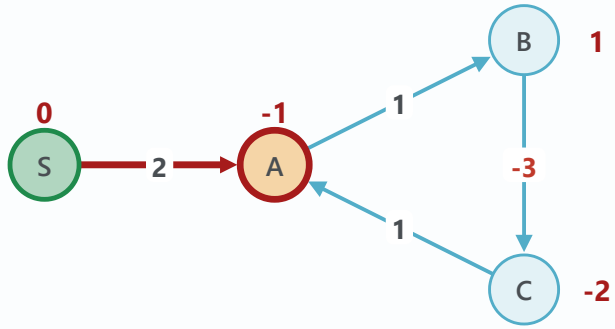
DEĞİŞKENLER

`i = 3` `u = C` `v = A` `w = 1`

`uzaklik = [0, -1, 1, -2]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
1  for (int i = 1; i < V; i++) {
2    for (Kenar k : kenarlar) {
3      int u = k.bas, v = k.son, w = k.agirlik;
4      if (uzaklik[u] != SONSUZ
5          && uzaklik[u] + w < uzaklik[v]) {
6        uzaklik[v] = uzaklik[u] + w;
7        onceki[v] = u;
8      }
9    }
10 }
▶ for (Kenar k : kenarlar) { // denetim
▶   int u = k.bas, v = k.son, w = k.agirlik;
▶   if (uzaklik[u] != SONSUZ
▶       && uzaklik[u] + w < uzaklik[v])
15     throw new IllegalStateException(
16         "Negatif döngü");
17 }
```



denetim turu

V − 1 = 3 tur bitti; denetim turu başlar. Kenar SA: 0 + 2 = 2 < -1 değil, sorun yok.

Adım 8 / 9

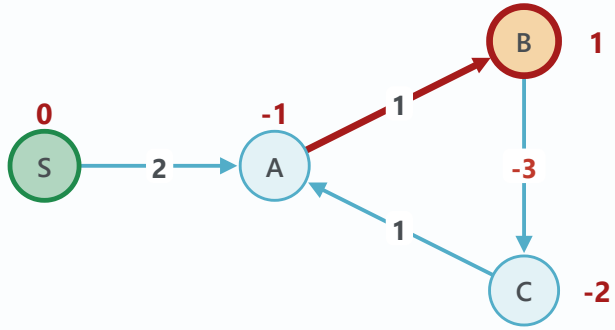
DEĞİŞKENLER

`i = –` `u = S` `v = A` `w = 2`

`uzaklik = [0, −1, 1, −2]`

Bellman-Ford: Negatif Döngü Denetimi İzleme

```
1  for (int i = 1; i < V; i++) {
2    for (Kenar k : kenarlar) {
3      int u = k.bas, v = k.son, w = k.agirlik;
4      if (uzaklik[u] != SONSUZ
5          && uzaklik[u] + w < uzaklik[v]) {
6        uzaklik[v] = uzaklik[u] + w;
7        onceki[v] = u;
8      }
9    }
10 }
11 for (Kenar k : kenarlar) { // denetim
▶    int u = k.bas, v = k.son, w = k.agirlik;
▶    if (uzaklik[u] != SONSUZ
▶        && uzaklik[u] + w < uzaklik[v])
▶      throw new IllegalStateException(
▶          "Negatif döngü");
17 }
```



denetim turu **negatif döngü!**

Denetim, kenar AB: -1 + 1 = 0 < 1 → V – 1 turdan sonra hâlâ gevşiyor: **negatif döngü var**, istisna fırlatılır.

Adım 9 / 9

DEĞİŞKENLER

`i = –` `u = A` `v = B` `w = 1`

`uzaklik = [0, -1, 1, -2]`

Floyd-Warshall Algoritması

- **Tüm düğüm çiftleri** arasındaki en kısa yolları bulur (all-pairs).
- Bernard Roy (1959), Stephen **Warshall** (1962, geçişli kapanış) ve Robert **Floyd** (1962, en kısa yol) tarafından birbirinden bağımsız verilmiştir.
- **Negatif ağırlıklı kenarlarla** çalışır; negatif döngüyü **tespit eder** (o durumda en kısa yol tanımsızdır).
- **Dinamik programlama** kullanır; sonuçları bir V × V matriste (D) tutar.

Tek kaynak mı, tüm çiftler mi?

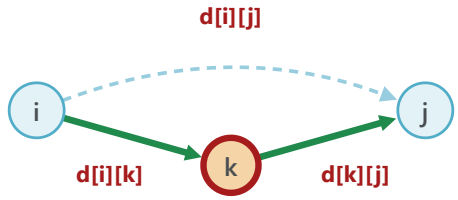
Bellman-Ford ve Dijkstra **tek kaynaktan** diğer düğümlere en kısa yolları bulur. Floyd-Warshall ise **tüm çiftler** arasındaki en kısa yolları tek seferde, üç iç içe döngüyle hesaplar. Kod çok kısadır; yoğun çizgelerde pratiktir.

Floyd-Warshall: Adımlar ve Yineleme Bağıntısı

1. **Adım 1:** D matrisi kenar ağırlıklarıyla doldurulur: D[i][i] = 0; doğrudan kenar yoksa ∞.
2. **Adım 2:** Ara düğüm k = 1, 2, …, n sırayla ele alınır; her (i, j) çifti için…
3. **Adım 3:** k üzerinden geçen yolun uzunluğu D[i][k] + D[k][j] hesaplanır ve mevcut D[i][j] ile karşılaştırılır.
4. **Adım 4:** Daha kısaysa yeni değer matrise yazılır (Π matrisi de güncellenir).

Alt problem

$D^{(k)}[i][j]$: ara düğümleri yalnızca {1, …, k} kümesinden seçilen en kısa i ⇝ j yolunun uzunluğu.



$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j])$

k ya yolda yoktur ya da yol k'den bir kez geçer. $D^{(n)}$ sonuçtur.

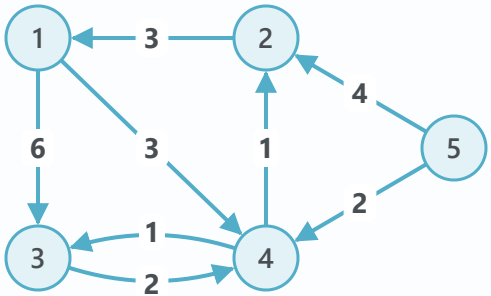
Floyd-Warshall: Karmaşıklık

- Zaman **$\Theta(V^3)$**: 3 iç içe for döngüsü, her biri V kez.
  - ilk (en dış) döngü tüm **ara** düğümleri (k) gezer,
  - ikinci döngü tüm **kaynak** düğümleri (i) gezer,
  - son döngü tüm **hedef** düğümleri (j) gezer.
- Bellek **$\Theta(V^2)$**: D yerinde (in-place) güncellenebilir; k. satır ve sütun k turunda değişmez.

Tüm çiftler için	Zaman
Floyd-Warshall	$O(V^3)$
V × Dijkstra (dizi)	$O(V^3)$
V × Dijkstra (heap)	$O(V(V + E) \log V)$
V × Bellman-Ford	$O(V^2 E)$

Döngü sırası önemlidir: **k en dışta** olmalıdır. k içeride olursa sonuç yanlış çıkar.

Floyd-Warshall Örneği: D ve Π Matrisleri



$D^{(0)}$

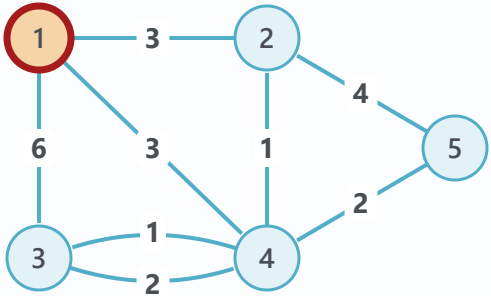
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	∞	∞	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

$\Pi^{(0)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	/	/	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

k = 0 (ilklendirme): D kenar ağırlıklarıyla doldurulur, köşegen 0, kenar yoksa ∞. Π[i][j] = i (doğrudan kenar varsa), yoksa /.

Floyd-Warshall Örneği: D ve Π Matrisleri



$D^{(1)}$

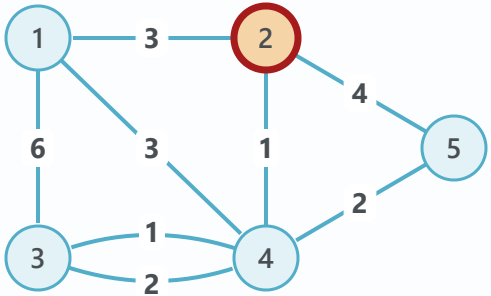
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	**9**	**6**	∞
3	∞	∞	0	2	∞
4	∞	1	1	0	∞
5	∞	4	∞	2	0

$\Pi^{(1)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	**1**	**1**	/
3	/	/	/	3	/
4	/	4	4	/	/
5	/	5	/	5	/

k = 1: düğüm 1 üzerinden **2** ⇝ **3** (3 + 6 = 9) ve **2** ⇝ **4** (3 + 3 = 6) yolları bulundu.

Floyd-Warshall Örneği: D ve Π Matrisleri



$D^{(2)}$

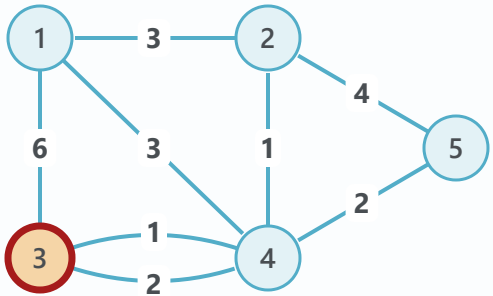
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	**4**	1	1	0	∞
5	**7**	4	**13**	2	0

$\Pi^{(2)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	**2**	4	4	/	/
5	**2**	5	**1**	5	/

k = 2: düğüm 2 üzerinden **4** ⇝ **1** (1 + 3 = 4), **5** ⇝ **1** (4 + 3 = 7) ve **5** ⇝ **3** (4 + 9 = 13) bulundu.

Floyd-Warshall Örneği: D ve Π Matrisleri



$D^{(3)}$

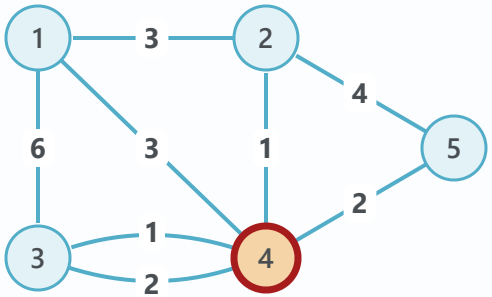
	1	2	3	4	5
1	0	∞	6	3	∞
2	3	0	9	6	∞
3	∞	∞	0	2	∞
4	4	1	1	0	∞
5	7	4	13	2	0

$\Pi^{(3)}$

	1	2	3	4	5
1	/	/	1	1	/
2	2	/	1	1	/
3	/	/	/	3	/
4	2	4	4	/	/
5	2	5	1	5	/

k = 3: düğüm 3 üzerinden kısa yol bulunamadı (3'ten yalnızca 4'e kenar var, hiçbir değer iyileşmez).

Floyd-Warshall Örneği: D ve Π Matrisleri



$D^{(4)}$

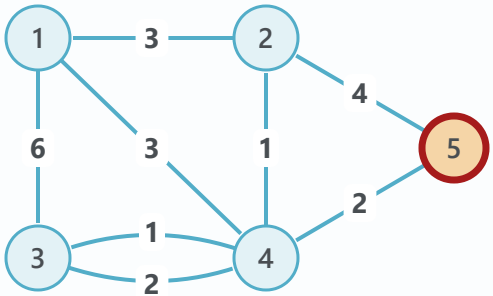
	1	2	3	4	5
1	0	**4**	**4**	3	∞
2	3	0	**7**	6	∞
3	**6**	**3**	0	2	∞
4	4	1	1	0	∞
5	**6**	**3**	**3**	2	0

$\Pi^{(4)}$

	1	2	3	4	5
1	/	**4**	**4**	1	/
2	2	/	**4**	1	/
3	**2**	**4**	/	3	/
4	2	4	4	/	/
5	**2**	**4**	**4**	5	/

k = 4: düğüm 4 üzerinden 1⇝2, 1⇝3, 2⇝3, 3⇝1, 3⇝2, 5⇝1, 5⇝2, 5⇝3 kısaldı (8 hücre). Ör. 5⇝3: 2 + 1 = 3 < 13.

Floyd-Warshall Örneği: D ve Π Matrisleri



$D^{(5)}$

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

$\Pi^{(5)}$

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

k = 5: düğüm 5 üzerinden kısa yol yok (5'e gelen kenar olmadığından D[i][5] = ∞). Son matrisler hazır.

Son Durum ve Yolun Okunması

D (uzaklıklar)

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

Π (önceki)

	1	2	3	4	5
1	/	4	4	1	/
2	2	/	4	1	/
3	2	4	/	3	/
4	2	4	4	/	/
5	2	4	4	5	/

- Π[i][j]: i'den j'ye en kısa yolda **j'den önceki** düğüm.
- 5 ⇝ 1: Π[5][1] = 2, Π[5][2] = 4, Π[5][4] = 5 → **5** → **4** → **2** → **1**, D[5][1] = 6.
- 1 ⇝ 2: Π[1][2] = 4, Π[1][4] = 1 → **1** → **4** → **2**, D[1][2] = 4.
- 5. sütun ∞: 5'e hiçbir düğümden ulaşılamaz.
- 5. satır, Dijkstra ve Bellman-Ford'un kaynak 5 için bulduğu [6, 3, 3, 2, 0] ile aynıdır.

Floyd-Warshall: Sözde Kod

1. İlklendirme

```
FLOYD_WARSHALL(G):
  mesafeler = [ ][ ]
  her bir u için V içinde:
    her bir v için V içinde:
      eğer u == v:
        mesafeler[u][v] = 0
      yoksa eğer (u, v) E içinde:
        mesafeler[u][v] = ağırlık(u, v)
      yoksa:
        mesafeler[u][v] = sonsuz
```

2. Üç iç içe döngü ve denetim (m = uzaklıklar)

```
  her bir k için V içinde:
    her bir i için V içinde:
      her bir j için V içinde:
        eğer m[i][k] + m[k][j] < m[i][j]:
          m[i][j] = m[i][k] + m[k][j]
  her bir i için V içinde:
    eğer mesafeler[i][i] < 0:
      hata "Negatif ağırlıklı döngü var"
  döndür mesafeler
```

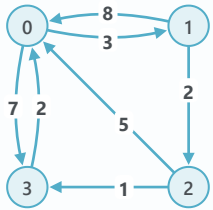
Floyd-Warshall: Java

```java
void floydWarshall(int[][] d) {  // V x V
  for (int k = 0; k < V; k++)       // ara
    for (int i = 0; i < V; i++)     // kaynak
      for (int j = 0; j < V; j++)   // hedef
        if (d[i][k] != SONSUZ
            && d[k][j] != SONSUZ
            && d[i][k] + d[k][j] < d[i][j]) {
          d[i][j] = d[i][k] + d[k][j];
          onceki[i][j] = onceki[k][j];
        }
  for (int i = 0; i < V; i++)
    if (d[i][i] < 0)
      throw new IllegalStateException(
          "Negatif döngü");
}
```

- Matris **yerinde** güncellenir; ayrı $D^{(k)}$ kopyası gerekmez.
- SONSUZ denetimi, `Integer.MAX_VALUE` toplamının taşmasını (overflow) önler.
- `onceki[i][j] = onceki[k][j]` : yeni yolda j'den önceki düğüm, k ⇝ j yolundakiyle aynıdır.
- Negatif döngü: bir düğümün **kendine** uzaklığı negatif çıkarsa (D[i][i] < 0).

Floyd-Warshall: Kodu Adım Adım İzle

```
▶ void floydWarshall(int[][] d) {  // V x V
2   for (int k = 0; k < V; k++)        // ara
3     for (int i = 0; i < V; i++)      // kaynak
4       for (int j = 0; j < V; j++)    // hedef
5         if (d[i][k] != SONSUZ
6             && d[k][j] != SONSUZ
7             && d[i][k] + d[k][j] < d[i][j]) {
8           d[i][j] = d[i][k] + d[k][j];
9           onceki[i][j] = onceki[k][j];
10        }
11  for (int i = 0; i < V; i++)
12    if (d[i][i] < 0)
13      throw new IllegalStateException(
14          "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	∞
2	5	∞	0	1
3	2	∞	∞	0

onceki

	0	1	2	3
0	/	0	/	0
1	1	/	1	/
2	2	/	/	2
3	3	/	/	/

Başlangıç: d kenar ağırlıklarıyla dolu (köşegen 0, kenar yoksa ∞); onceki[i][j] = i. Sağdaki matris d, altındaki onceki.

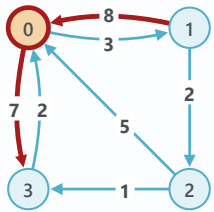
Adım 1 / 13

DEĞİŞKENLER

`k = –` `i = –` `j = –`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // ara
3      for (int i = 0; i < V; i++)      // kaynak
4        for (int j = 0; j < V; j++)    // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	15
2	5	∞	0	1
3	2	∞	∞	0

onceki

	0	1	2	3
0	/	0	/	0
1	1	/	1	0
2	2	/	/	2
3	3	/	/	/

gri: k = 0 satırı ve sütunu

k = 0 ara düğüm: d[1][0] + d[0][3] = 8 + 7 = 15 < ∞ → d[1][3] = 15, onceki[1][3] = onceki[0][3] = 0.

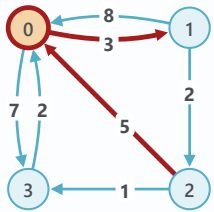
Adım 2 / 13

DEĞİŞKENLER

`k = 0` `i = 1` `j = 3` `d[i][j] = 15`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)        // ara
3      for (int i = 0; i < V; i++)      // kaynak
4        for (int j = 0; j < V; j++)    // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	15
2	5	8	0	1
3	2	∞	∞	0

onceki

	0	1	2	3
0	/	0	/	0
1	1	/	1	0
2	2	0	/	2
3	3	/	/	/

gri: k = 0 satırı ve sütunu

k = 0 ara düğüm: d[2][0] + d[0][1] = 5 + 3 = 8 < ∞ → d[2][1] = 8, onceki[2][1] = onceki[0][1] = 0.

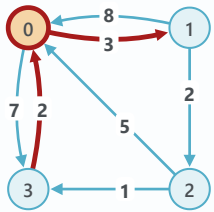
Adım 3 / 13

DEĞİŞKENLER

`k = 0` `i = 2` `j = 1` `d[i][j] = 8`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	∞	7
1	8	0	2	15
2	5	8	0	1
3	2	5	∞	0

onceki

	0	1	2	3
0	/	0	/	0
1	1	/	1	0
2	2	0	/	2
3	3	0	/	/

gri: k = 0 satırı ve sütunu

k = 0 ara düğüm: d[3][0] + d[0][1] = 2 + 3 = 5 < ∞ → d[3][1] = 5, onceki[3][1] = onceki[0][1] = 0.

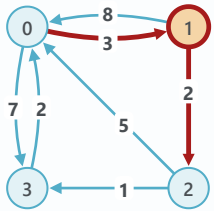
Adım 4 / 13

DEĞİŞKENLER

`k = 0` `i = 3` `j = 1` `d[i][j] = 5`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	7
1	8	0	2	15
2	5	8	0	1
3	2	5	∞	0

onceki

	0	1	2	3
0	/	0	1	0
1	1	/	1	0
2	2	0	/	2
3	3	0	/	/

gri: k = 1 satırı ve sütunu

k = 1 ara düğüm: d[0][1] + d[1][2] = 3 + 2 = 5 < ∞ → d[0][2] = 5, onceki[0][2] = onceki[1][2] = 1.

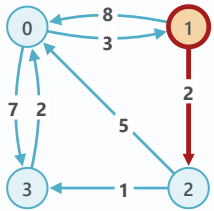
Adım 5 / 13

DEĞİŞKENLER

`k = 1` `i = 0` `j = 2` `d[i][j] = 5`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	7
1	8	0	2	15
2	5	8	0	1
3	2	5	**7**	0

onceki

	0	1	2	3
0	/	0	1	0
1	1	/	1	0
2	2	0	/	2
3	3	0	**1**	/

gri: k = 1 satırı ve sütunu

k = 1 ara düğüm: d[3][1] + d[1][2] = 5 + 2 = 7 < ∞ → d[3][2] = 7, onceki[3][2] = onceki[1][2] = 1.

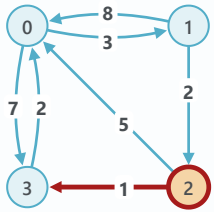
Adım 6 / 13

DEĞİŞKENLER

`k = 1` `i = 3` `j = 2` `d[i][j] = 7`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	8	0	2	15
2	5	8	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	1	/	1	0
2	2	0	/	2
3	3	0	1	/

gri: k = 2 satırı ve sütunu

k = 2 ara düğüm: d[0][2] + d[2][3] = 5 + 1 = 6 < 7 → d[0][3] = 6, onceki[0][3] = onceki[2][3] = 2.

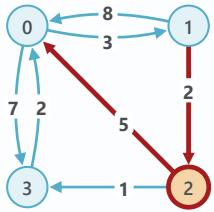
Adım 7 / 13

DEĞİŞKENLER

`k = 2` `i = 0` `j = 3` `d[i][j] = 6`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	7	0	2	15
2	5	8	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	2	/	1	0
2	2	0	/	2
3	3	0	1	/

gri: k = 2 satırı ve sütunu

k = 2 ara düğüm: d[1][2] + d[2][0] = 2 + 5 = 7 < 8 → d[1][0] = 7, onceki[1][0] = onceki[2][0] = 2.

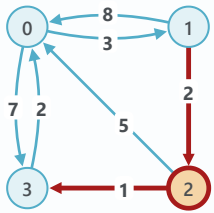
Adım 8 / 13

DEĞİŞKENLER

`k = 2` `i = 1` `j = 0` `d[i][j] = 7`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	7	0	2	3
2	5	8	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	2	/	1	2
2	2	0	/	2
3	3	0	1	/

gri: k = 2 satırı ve sütunu

k = 2 ara düğüm: d[1][2] + d[2][3] = 2 + 1 = 3 < 15 → d[1][3] = 3, onceki[1][3] = onceki[2][3] = 2.

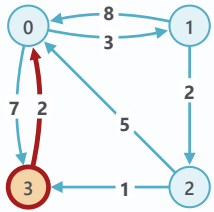
Adım 9 / 13

DEĞİŞKENLER

`k = 2` `i = 1` `j = 3` `d[i][j] = 3`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	5	8	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	2	0	/	2
3	3	0	1	/

gri: k = 3 satırı ve sütunu

k = 3 ara düğüm: d[1][3] + d[3][0] = 3 + 2 = 5 < 7 → d[1][0] = 5, onceki[1][0] = onceki[3][0] = 3.

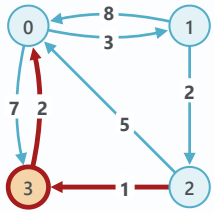
Adım 10 / 13

DEĞİŞKENLER

`k = 3` `i = 1` `j = 0` `d[i][j] = 5`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	8	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

gri: k = 3 satırı ve sütunu

k = 3 ara düğüm: d[2][3] + d[3][0] = 1 + 2 = 3 < 5 → d[2][0] = 3, onceki[2][0] = onceki[3][0] = 3.

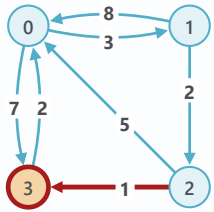
Adım 11 / 13

DEĞİŞKENLER

`k = 3` `i = 2` `j = 0` `d[i][j] = 3`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
▶          if (d[i][k] != SONSUZ
▶              && d[k][j] != SONSUZ
▶              && d[i][k] + d[k][j] < d[i][j]) {
▶            d[i][j] = d[i][k] + d[k][j];
▶            onceki[i][j] = onceki[k][j];
10         }
11   for (int i = 0; i < V; i++)
12     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

gri: k = 3 satırı ve sütunu

k = 3 ara düğüm: d[2][3] + d[3][1] = 1 + 5 = 6 < 8 → d[2][1] = 6, onceki[2][1] = onceki[3][1] = 0.

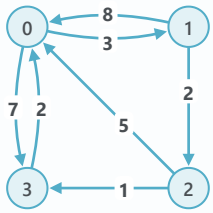
Adım 12 / 13

DEĞİŞKENLER

`k = 3` `i = 2` `j = 1` `d[i][j] = 6`

Floyd-Warshall: Kodu Adım Adım İzle

```
1  void floydWarshall(int[][] d) {  // V x V
2    for (int k = 0; k < V; k++)       // ara
3      for (int i = 0; i < V; i++)     // kaynak
4        for (int j = 0; j < V; j++)   // hedef
5          if (d[i][k] != SONSUZ
6              && d[k][j] != SONSUZ
7              && d[i][k] + d[k][j] < d[i][j]) {
8            d[i][j] = d[i][k] + d[k][j];
9            onceki[i][j] = onceki[k][j];
10         }
▶   for (int i = 0; i < V; i++)
▶     if (d[i][i] < 0)
13       throw new IllegalStateException(
14           "Negatif döngü");
15 }
```



d

	0	1	2	3
0	0	3	5	6
1	5	0	2	3
2	3	6	0	1
3	2	5	7	0

onceki

	0	1	2	3
0	/	0	1	2
1	3	/	1	2
2	3	0	/	2
3	3	0	1	/

Dört tur bitti. Köşegende negatif değer yok → negatif döngü yok. Örnek yol: onceki[1][3] = 2, onceki[1][2] = 1 → 1 → 2 → 3, d[1][3] = 3.

Adım 13 / 13

DEĞİŞKENLER

k = – i = – j = –

A* (A Yıldız) Arama Algoritması

- İki nokta (başlangıç → hedef) arasındaki en kısa yolu bulan bir **arama** algoritmasıdır.
- 1968'de Peter **Hart**, Nils **Nilsson** ve Bertram **Raphael** tarafından geliştirildi.
- Dijkstra'nın (düzgün maliyetli arama) gerçek maliyetini, açgözlü **en iyi ilk arama** (Greedy Best-First Search) yaklaşımının hedef tahminiyle birleştirir.
- Düzgün çalışması için iyi bir **tahmin fonksiyonu** (sezgisel, heuristic) gerekir.
- Düğüm sayısı arttıkça zaman ve bellek maliyeti hızla artar.

Algoritma ilkeleri

Her düğüm için bir **tahmin** h(n) kullanılır: düğümün hedefe olan tahmini uzaklığı. Her adımda **açık listedeki** düğümler arasından gerçek maliyet ile tahmini maliyetin toplamı en küçük olan seçilir. Böylece algoritma hedefe doğru ilerlerken en az maliyetli yolu da korur.

A* ARAMA

A* Nasıl Çalışır? f(n) = g(n) + h(n)

g(n): gerçek maliyet
Başlangıç düğümünden n'ye kadar **bulunan** yolun maliyeti ("g-maliyet"). Dijkstra'daki d[n] ile aynıdır.

h(n): sezgisel
n'den hedefe **tahmini** maliyet ("h-maliyet"). Kuş uçuşu uzaklık, Manhattan uzaklığı gibi.

f(n) = g(n) + h(n)
n üzerinden geçen bir çözümün tahmini toplam maliyeti. Açık listeden en küçük f'li düğüm genişletilir.

Seçim ölçütü	Algoritma	Davranış
yalnızca g(n)	**Dijkstra**	her yöne eşit yayılır; doğru ama çok düğüm açar
yalnızca h(n)	**Açgözlü en iyi ilk**	hedefe koşar; hızlı ama en kısa yolu garanti etmez
g(n) + h(n)	**A***	hedefe yönelir; h uygunsa en kısa yolu garanti eder

Sezgisel tahminlere göre hedefe **yakın görünen** düğümler önceliklendirilir; ama g sayesinde pahalı bir yoldan gelen düğüm geri planda kalır.

A*: Algoritma Adımları

1. Başlangıç düğümünün g değeri 0 yapılır ve düğüm **açık listeye** (open list, öncelikli kuyruk) eklenir. Diğer düğümlerin g değeri ∞'dur.

Açık liste: keşfedilmiş ama henüz genişletilmemiş düğümler.

A*: Algoritma Adımları

1. Başlangıç düğümünün g değeri 0 yapılır ve düğüm **açık listeye** (open list, öncelikli kuyruk) eklenir. Diğer düğümlerin g değeri ∞'dur.

Açık liste: keşfedilmiş ama henüz genişletilmemiş düğümler.

2. Açık listeden **f = g + h** değeri en küçük düğüm çıkarılır ve **kapalı listeye** (closed list) konur.

Seçim ölçütü g ya da h tek başına değil, toplamları f = g + h'dir.

A*: Algoritma Adımları

1. Başlangıç düğümünün g değeri 0 yapılır ve düğüm **açık listeye** (open list, öncelikli kuyruk) eklenir. Diğer düğümlerin g değeri ∞'dur.

Açık liste: keşfedilmiş ama henüz genişletilmemiş düğümler.

2. Açık listeden **f = g + h** değeri en küçük düğüm çıkarılır ve **kapalı listeye** (closed list) konur.

Seçim ölçütü g ya da h tek başına değil, toplamları f = g + h'dir.

3. Çıkarılan düğüm **hedefse** çözüm bulunmuştur: yol, önceki işaretçileriyle geri kurulur.

Hedef, açık listeye eklendiği anda değil, çıkarıldığı anda kesinleşir.

A*: Algoritma Adımları

1. Başlangıç düğümünün g değeri 0 yapılır ve düğüm **açık listeye** (open list, öncelikli kuyruk) eklenir. Diğer düğümlerin g değeri ∞'dur.

Açık liste: keşfedilmiş ama henüz genişletilmemiş düğümler.

2. Açık listeden **f = g + h** değeri en küçük düğüm çıkarılır ve **kapalı listeye** (closed list) konur.

Seçim ölçütü g ya da h tek başına değil, toplamları f = g + h'dir.

3. Çıkarılan düğüm **hedefse** çözüm bulunmuştur: yol, önceki işaretçileriyle geri kurulur.

Hedef, açık listeye eklendiği anda değil, çıkarıldığı anda kesinleşir.

4. Değilse her komşu için g' = g(u) + w(u, v) hesaplanır; g' daha küçükse g, f ve önceki değerleri **güncellenir**, komşu açık listeye eklenir.

Bu, Dijkstra'daki gevşetmenin aynısıdır.

A*: Algoritma Adımları

1. Başlangıç düğümünün g değeri 0 yapılır ve düğüm **açık listeye** (open list, öncelikli kuyruk) eklenir. Diğer düğümlerin g değeri ∞'dur.

Açık liste: keşfedilmiş ama henüz genişletilmemiş düğümler.

2. Açık listeden **f = g + h** değeri en küçük düğüm çıkarılır ve **kapalı listeye** (closed list) konur.

Seçim ölçütü g ya da h tek başına değil, toplamları f = g + h'dir.

3. Çıkarılan düğüm **hedefse** çözüm bulunmuştur: yol, önceki işaretçileriyle geri kurulur.

Hedef, açık listeye eklendiği anda değil, çıkarıldığı anda kesinleşir.

4. Değilse her komşu için g' = g(u) + w(u, v) hesaplanır; g' daha küçükse g, f ve önceki değerleri **güncellenir**, komşu açık listeye eklenir.

Bu, Dijkstra'daki gevşetmenin aynısıdır.

5. 2. adıma dönülür. Açık liste boşalırsa hedefe **yol yoktur**.

Seçim yalnızca son düğümün komşuları arasından değil, açık listenin tamamından yapılır.

Sezgisel Fonksiyonun Özellikleri

Kabul edilebilir (admissible)

$h(n) \le h^*(n)$: gerçek kalan maliyeti **asla aşmaz**. Bu durumda A* hedefi çıkardığında bulduğu yol en kısadır.

Tutarlı (consistent)

$h(u) \le w(u, v) + h(v)$ ve $h(\text{hedef}) = 0$. Tutarlı h ile bir düğüm kapalı listeye girdiğinde kesinleşir (yeniden açılmaz).

Izgarada sezgiseller

4 yön: Manhattan $|\Delta x| + |\Delta y|$. 8 yön: oktil (octile) $14 \cdot \min + 10 \cdot (\max - \min)$. Serbest düzlem: Öklid uzaklığı.

Uç durumlar

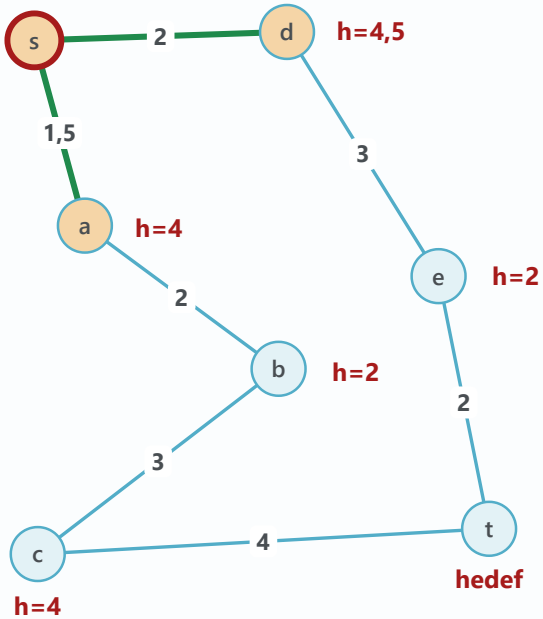
$h = 0 \rightarrow$ A* = Dijkstra. $h = h^*$ (mükemmel) → yalnızca en kısa yol üzerindeki düğümler açılır. $h > h^* \rightarrow$ hızlı ama en kısa yol garanti edilmez.

A*: Algoritma Karmaşıklığı

- Arama ağacı açısından en kötü durumda zaman ve bellek **$O(b^d)$**:
  - **b**: dallanma faktörü (bir düğümün ortalama komşu sayısı), **d**: hedefin derinliği (çözümdeki adım sayısı).
- Sonlu bir çizgede tutarlı h ile A*, Dijkstra gibi en fazla **$O((V + E) \log V)$** iş yapar.
- Sezgisel ne kadar iyiyse o kadar az düğüm açılır: **mükemmel** tahminde (h = h*) yalnızca yol üzerindeki d düğüm genişletilir.
- Zayıf nokta **bellektir**: açık ve kapalı listeler büyür; IDA*, SMA* gibi türevler bellek sınırlıdır.

$O(b^d)$ mükemmel tahminin değil, en **kötü** durumun (h işe yaramaz, ör. h = 0) sınırıdır; mükemmel tahminde arama yol uzunluğuyla doğrusaldır.

A* Örnek: s'den t'ye (h değerleri düğüm yanında)



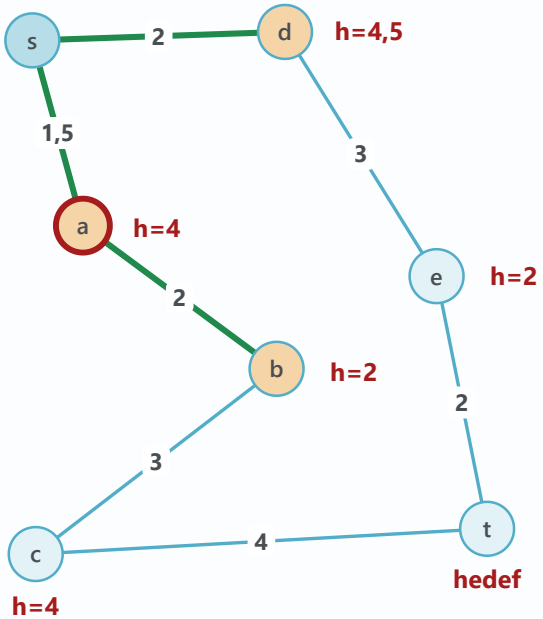
Açık liste

düğüm	g	h	f
a	1,5	4	**5,5**
d	2	4,5	6,5

Kapalı: s

s genişletilir: komşular a (g = 1,5) ve d (g = 2) açılır. f(a) = 1,5 + 4 = **5,5**, f(d) = 2 + 4,5 = 6,5.

A* Örnek: s'den t'ye (h değerleri düğüm yanında)



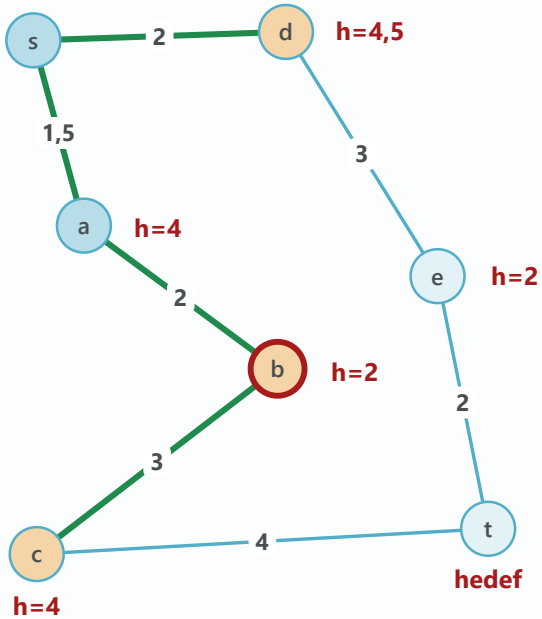
Açık liste

düğüm	g	h	f
b	3,5	2	**5,5**
d	2	4,5	6,5

Kapalı: s, a

En küçük f: a (5,5). a genişletilir → b: g = 3,5, f(b) = 3,5 + 2 = **5,5**.

A* Örnek: s'den t'ye (h değerleri düğüm yanında)



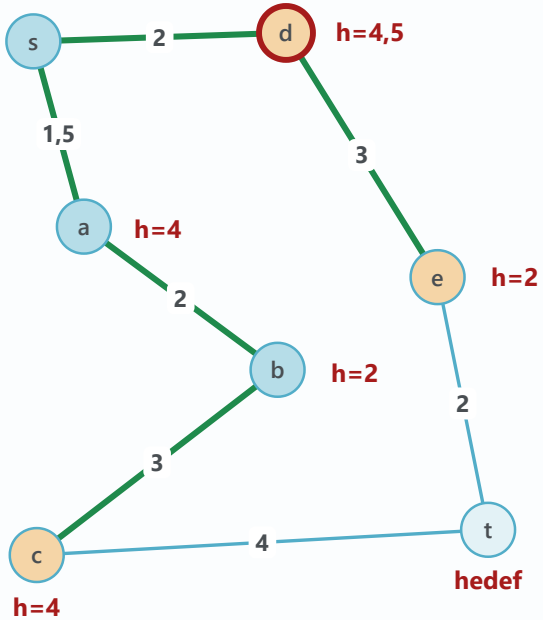
Açık liste

düğüm	g	h	f
d	2	4,5	**6,5**
c	6,5	4	10,5

Kapalı: s, a, b

b genişletilir (5,5 < 6,5) → c: g = 6,5, f(c) = 6,5 + 4 = 10,5. Bu yol pahalılaşıyor.

A* Örnek: s'den t'ye (h değerleri düğüm yanında)



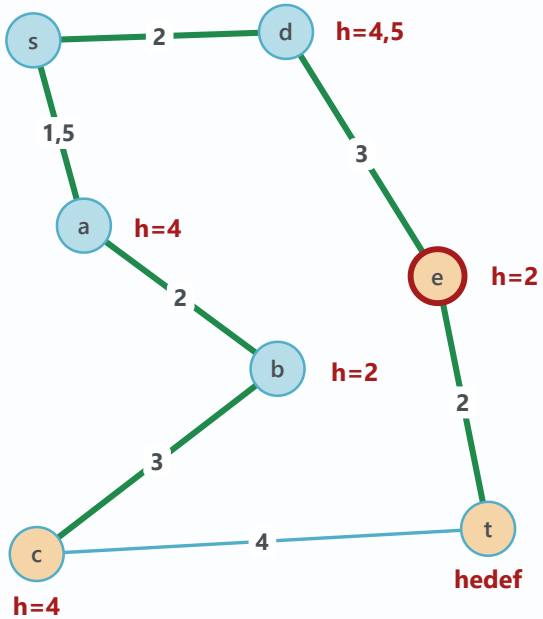
Açık liste

düğüm	g	h	f
e	5	2	**7**
c	6,5	4	10,5

Kapalı: s, a, b, d

Artık en küçük f: d (6,5). Arama diğer kola geçer → e: g = 5, f(e) = 5 + 2 = **7**.

A* Örnek: s'den t'ye (h değerleri düğüm yanında)



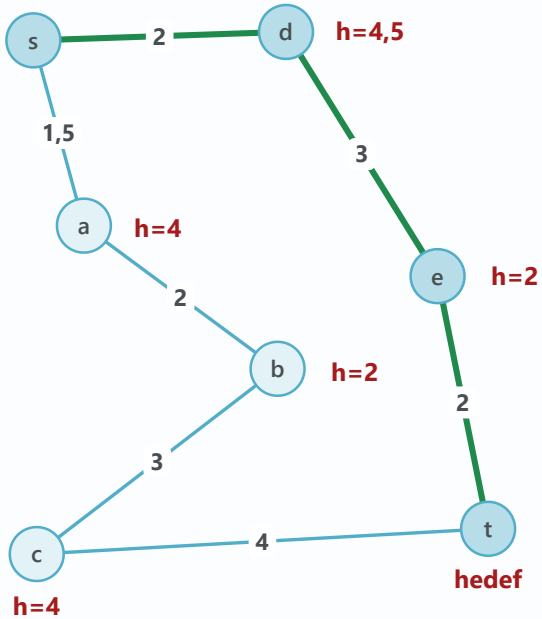
Açık liste

düğüm	g	h	f
t	7	0	**7**
c	6,5	4	10,5

Kapalı: s, a, b, d, e

e genişletilir → t: g = 5 + 2 = 7, f(t) = 7 + 0 = **7** < f(c) = 10,5.

A* Örnek: s'den t'ye (h değerleri düğüm yanında)



Açık liste

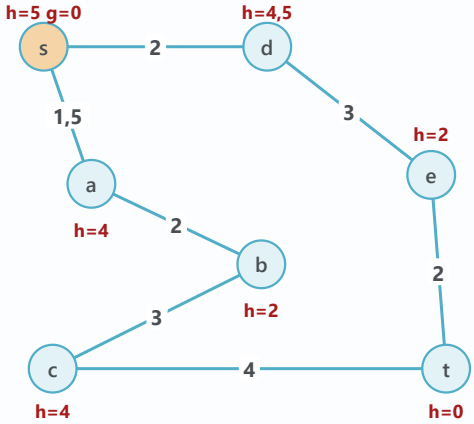
düğüm	g	h	f
c	6,5	4	**10,5**

Kapalı: s, a, b, d, e, t

t açık listeden çıkarıldı: **hedef**. En kısa yol s → d → e → t, maliyet 7. c hiç genişletilmedi.

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
▶    Arrays.fill(g, SONSUZ); g[s] = 0;
▶    Queue<Integer> acik = new PriorityQueue<>(
▶      Comparator.comparingDouble(v -> g[v] + h[v]));
▶    acik.add(s);
6    while (!acik.isEmpty()) {
7      int u = acik.poll();       // en küçük f
8      if (u == t) return g[t];   // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



Yönsüz çizge, hedef t. Düğüm yanındaki h değerleri hedefe tahmini uzaklıktır. g(s) = 0; açık listede yalnızca s var (parantezde f = g + h).

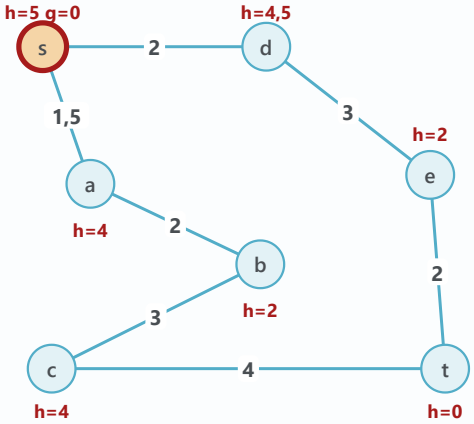
Adım 1 / 12

DEĞİŞKENLER

`u = –` `g(u) = –` `açık = [s(5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
▶      int u = acik.poll();      // en küçük f
▶      if (u == t) return g[t];  // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



Açık listede f'si en küçük olan s çıkarılır (f = 0 + 5 = 5). Hedef değil, komşuları incelenecek.

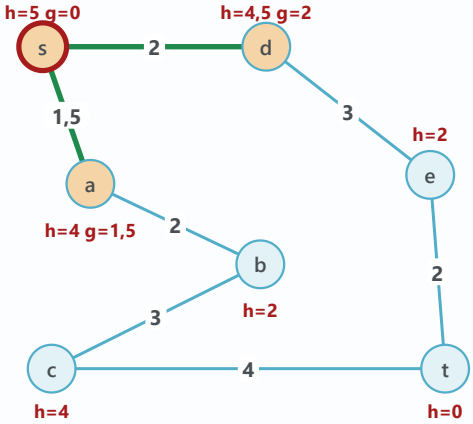
Adım 2 / 12

DEĞİŞKENLER

`u = s` `g(u) = 0` `açık = []`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
7      int u = acik.poll();       // en küçük f
8      if (u == t) return g[t];   // hedef
▶      for (Kenar k : komsu[u])
▶        if (g[u] + k.w < g[k.v]) {
▶          g[k.v] = g[u] + k.w; ata[k.v] = u;
▶          acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



g(a) = 0 + 1,5 = 1,5, f = 5,5; g(d) = 0 + 2 = 2, f = 6,5.

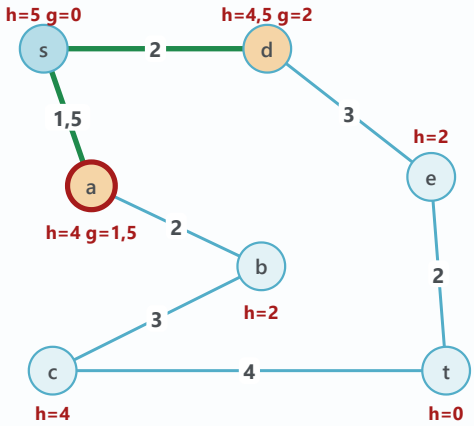
Adım 3 / 12

DEĞİŞKENLER

`u = s` `g(u) = 0` `açık = [a(5,5), d(6,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
▶      int u = acik.poll();       // en küçük f
▶      if (u == t) return g[t];   // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



Açık listede f'si en küçük olan a çıkarılır (f = 1,5 + 4 = 5,5). Hedef değil, komşuları incelenecek.

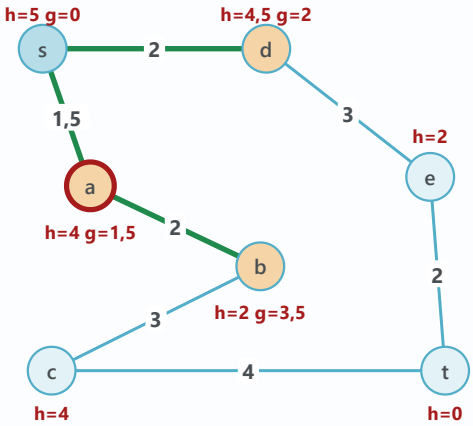
Adım 4 / 12

DEĞİŞKENLER

`u = a` `g(u) = 1,5` `açık = [d(6,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
7      int u = acik.poll();       // en küçük f
8      if (u == t) return g[t];   // hedef
▶      for (Kenar k : komsu[u])
▶        if (g[u] + k.w < g[k.v]) {
▶          g[k.v] = g[u] + k.w; ata[k.v] = u;
▶          acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



g(b) = 1,5 + 2 = 3,5, f = 5,5.

Adım 5 / 12

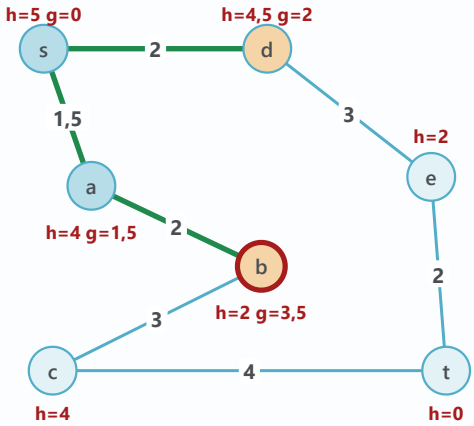
DEĞİŞKENLER

`u = a` `g(u) = 1,5`

`açık = [b(5,5), d(6,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
▶      int u = acik.poll();       // en küçük f
▶      if (u == t) return g[t];   // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



Açık listede f'si en küçük olan b çıkarılır (f = 3,5 + 2 = 5,5). Hedef değil, komşuları incelenecek.

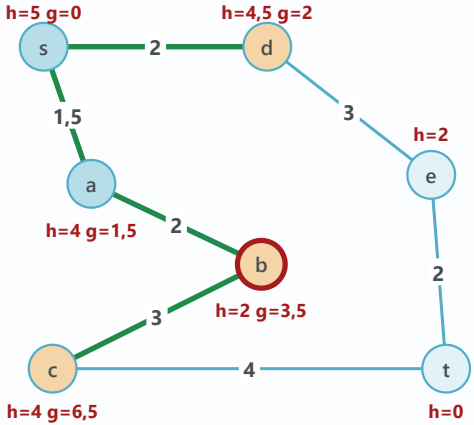
Adım 6 / 12

DEĞİŞKENLER

`u = b` `g(u) = 3,5` `açık = [d(6,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
7      int u = acik.poll();      // en küçük f
8      if (u == t) return g[t];  // hedef
▶      for (Kenar k : komsu[u])
▶        if (g[u] + k.w < g[k.v]) {
▶          g[k.v] = g[u] + k.w; ata[k.v] = u;
▶          acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



g(c) = 3,5 + 3 = 6,5, f = 10,5. c'nin f'si (10,5) büyük; sıradaki seçim f'si 6,5 olan d olacak.

Adım 7 / 12

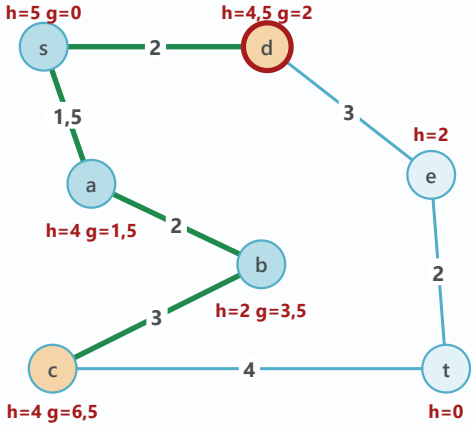
DEĞİŞKENLER

`u = b` `g(u) = 3,5`

`açık = [d(6,5), c(10,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
▶      int u = acik.poll();       // en küçük f
▶      if (u == t) return g[t];   // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



Açık listede f'si en küçük olan d çıkarılır (f = 2 + 4,5 = 6,5). Hedef değil, komşuları incelenecek.

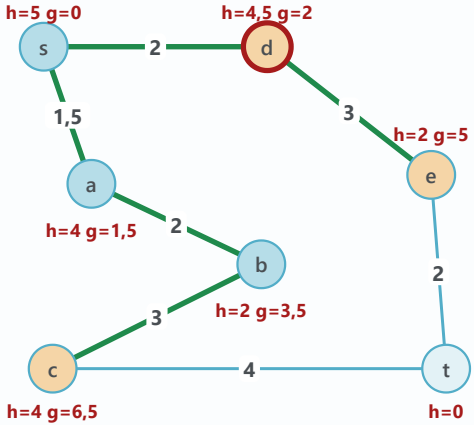
Adım 8 / 12

DEĞİŞKENLER

`u = d` `g(u) = 2` `açık = [c(10,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
7      int u = acik.poll();       // en küçük f
8      if (u == t) return g[t];   // hedef
▶      for (Kenar k : komsu[u])
▶        if (g[u] + k.w < g[k.v]) {
▶          g[k.v] = g[u] + k.w; ata[k.v] = u;
▶          acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



g(e) = 2 + 3 = 5, f = 7.

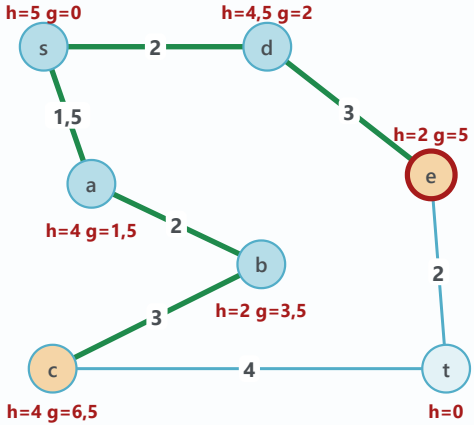
Adım 9 / 12

DEĞİŞKENLER

`u = d` `g(u) = 2` `açık = [e(7), c(10,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
▶      int u = acik.poll();      // en küçük f
▶      if (u == t) return g[t];  // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



Açık listede f'si en küçük olan e çıkarılır (f = 5 + 2 = 7). Hedef değil, komşuları incelenecek.

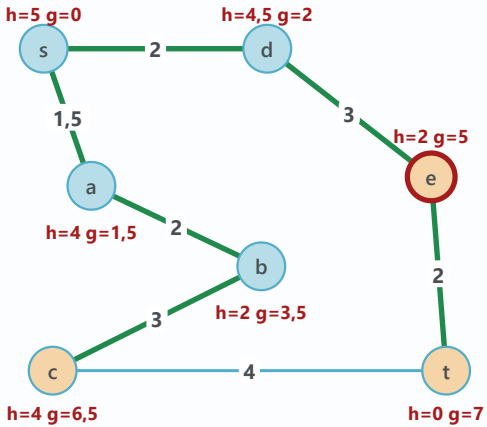
Adım 10 / 12

DEĞİŞKENLER

`u = e` `g(u) = 5` `açık = [c(10,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
7      int u = acik.poll();      // en küçük f
8      if (u == t) return g[t];  // hedef
▶      for (Kenar k : komsu[u])
▶        if (g[u] + k.w < g[k.v]) {
▶          g[k.v] = g[u] + k.w; ata[k.v] = u;
▶          acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



g(t) = 5 + 2 = 7, f = 7.

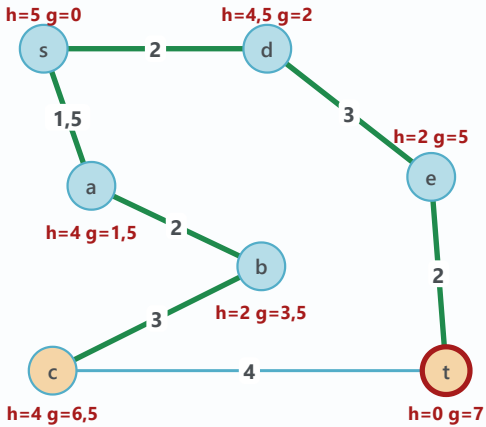
Adım 11 / 12

DEĞİŞKENLER

`u = e` `g(u) = 5` `açık = [t(7), c(10,5)]`

A*: Kodu Adım Adım İzle

```
1  double aYildiz(int s, int t) {
2    Arrays.fill(g, SONSUZ); g[s] = 0;
3    Queue<Integer> acik = new PriorityQueue<>(
4      Comparator.comparingDouble(v -> g[v] + h[v]));
5    acik.add(s);
6    while (!acik.isEmpty()) {
▶      int u = acik.poll();      // en küçük f
▶      if (u == t) return g[t];  // hedef
9      for (Kenar k : komsu[u])
10       if (g[u] + k.w < g[k.v]) {
11         g[k.v] = g[u] + k.w; ata[k.v] = u;
12         acik.remove(k.v); acik.add(k.v);
13       }
14   }
15   return SONSUZ;
16 }
```



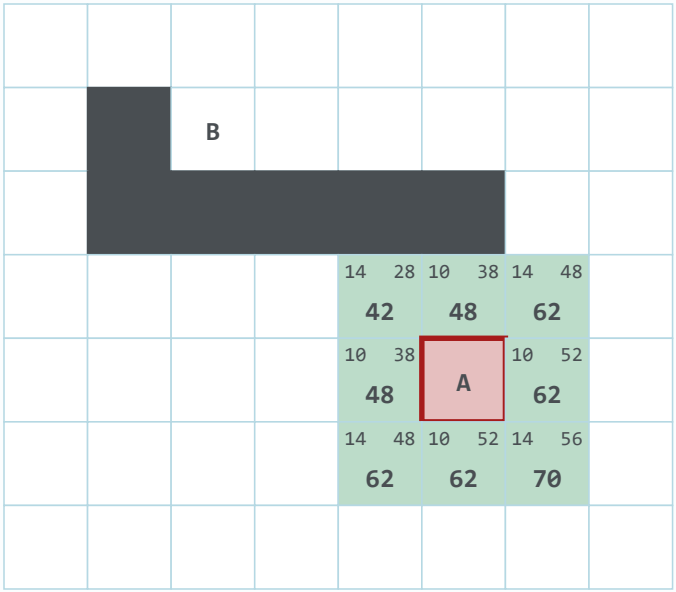
En küçük f'li düğüm t (f = 7): hedef açık listeden çıktı → `return g[t]` = **7**. Ebeveyn dizisinden yol: s → d → e → t; c hiç açılmadı.

Adım 12 / 12

DEĞİŞKENLER

`u = t` `g(u) = 7` `açık = [c(10,5)]`

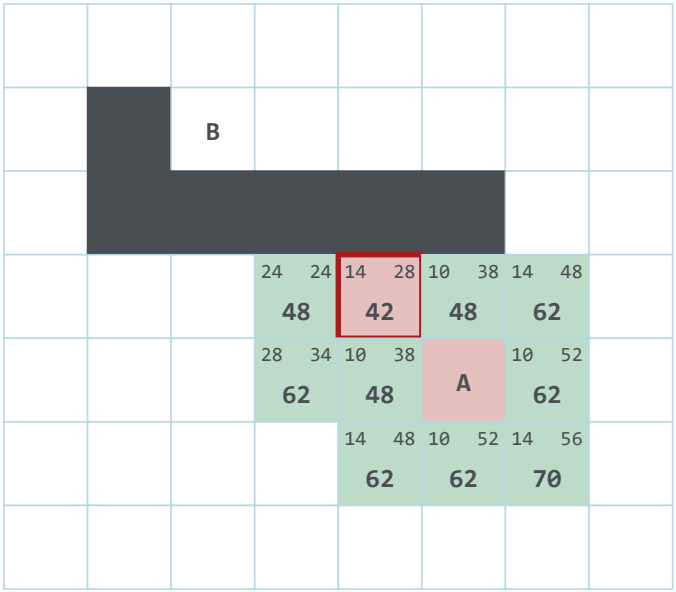
A* Örnek: Izgarada Duvarın Etrafından



açık kapalı yol duvar sol üst g · sağ üst h · orta f

A genişletilir: 8 komşu açılır. Düz adım 10, çapraz 14; h = oktil uzaklık (duvarlar yok sayılır). En küçük f = 42.

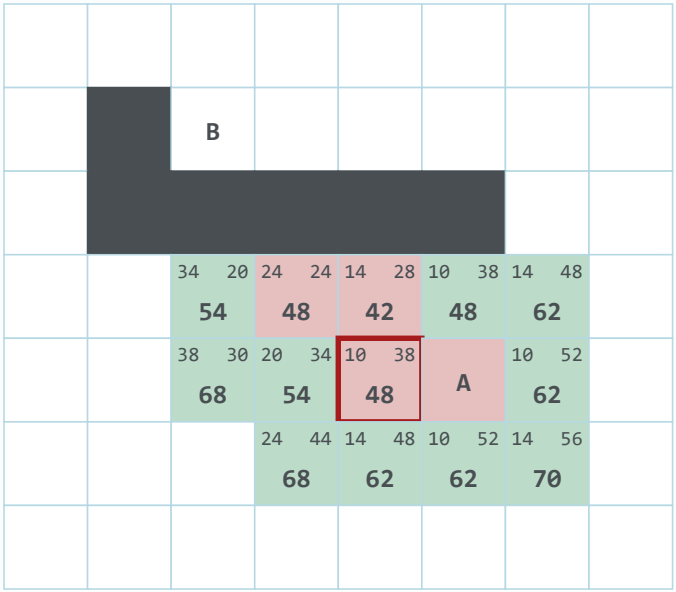
A* Örnek: Izgarada Duvarın Etrafından



açık kapalı yol duvar sol üst g · sağ üst h · orta f

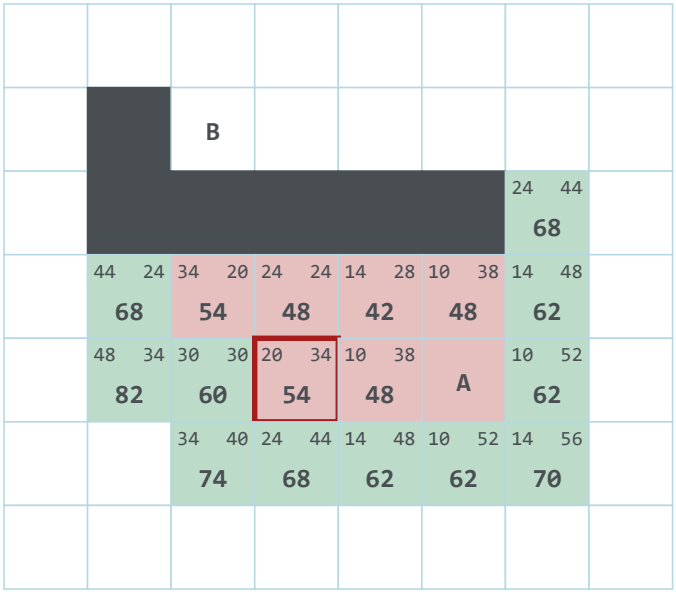
2. genişletme: en küçük f = 42 olan hücre (turuncu çerçeve) kapalı listeye geçer, komşuları açılır/güncellenir.

A* Örnek: Izgarada Duvarın Etrafından



4. genişletme: en küçük f = 48 olan hücre (turuncu çerçeve) kapalı listeye geçer, komşuları açılır/güncellenir.

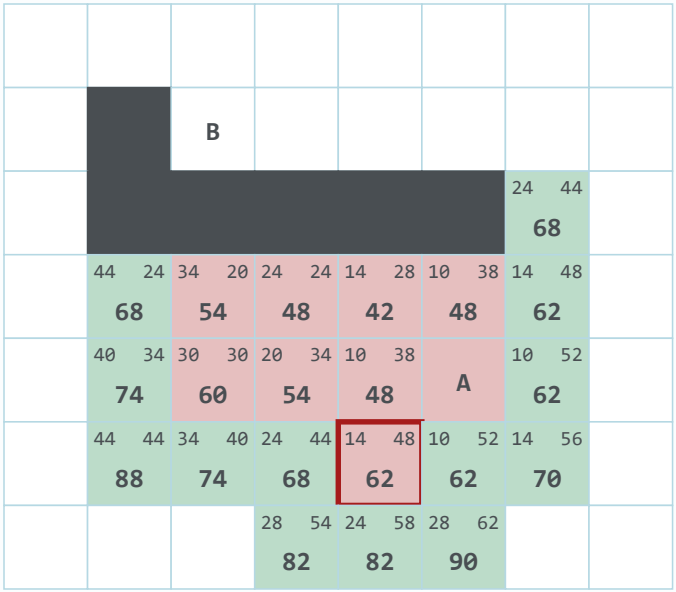
A* Örnek: Izgarada Duvarın Etrafından



açık kapalı yol duvar sol üst g · sağ üst h · orta f

7. genişletme: en küçük f = 54 olan hücre (turuncu çerçeve) kapalı listeye geçer, komşuları açılır/güncellenir. Duvarın altındaki hücreler f = 42…62 ile önce denenir.

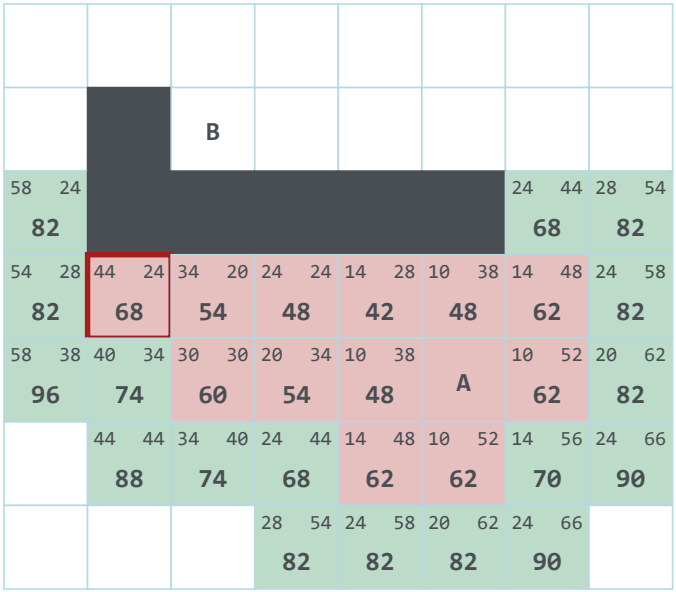
A* Örnek: Izgarada Duvarın Etrafından



açık kapalı yol duvar sol üst g · sağ üst h · orta f

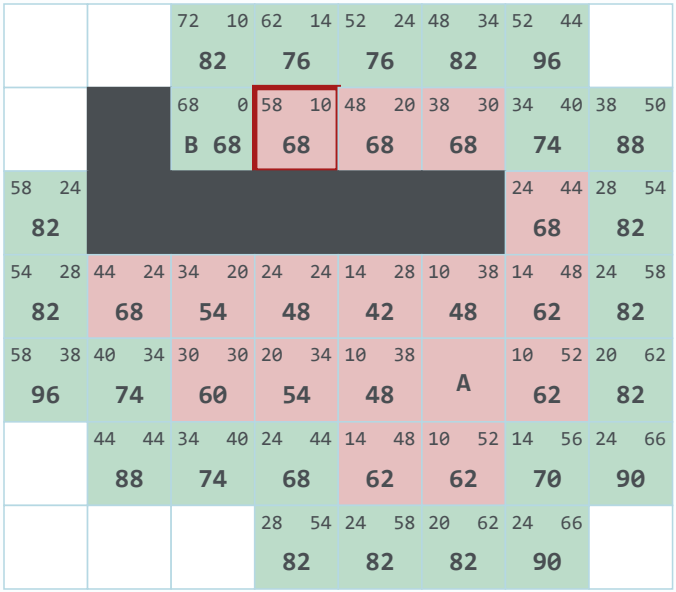
9. genişletme: en küçük f = 62 olan hücre (turuncu çerçeve) kapalı listeye geçer, komşuları açılır/güncellenir.

A* Örnek: Izgarada Duvarın Etrafından



13. genişletme: en küçük f = 68 olan hücre (turuncu çerçeve) kapalı listeye geçer, komşuları açılır/güncellenir. Alt yol tıkalı: f = 68 değerine çıkıldı, arama duvarın iki yanına yayılıyor.

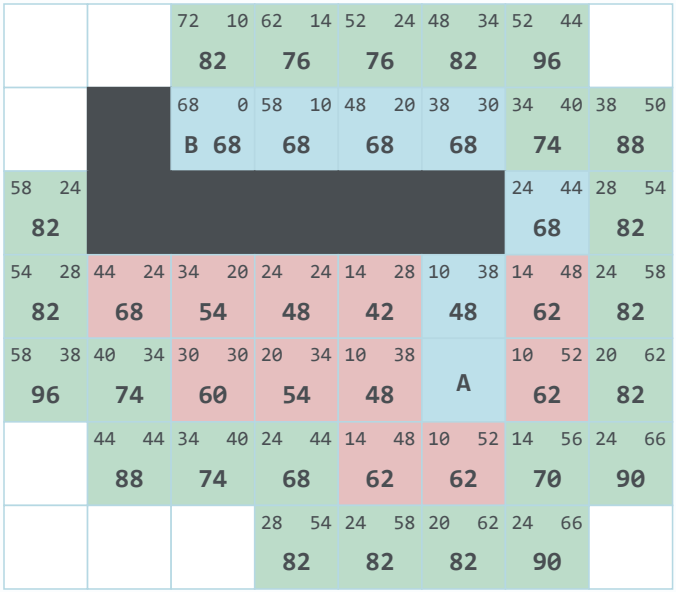
A* Örnek: Izgarada Duvarın Etrafından



açık kapalı yol duvar sol üst g · sağ üst h · orta f

17. genişletme (f = 68): duvarın sağından dolaşan yol B'ye ulaştı; B açık listede f = 68.

A* Örnek: Izgarada Duvarın Etrafından



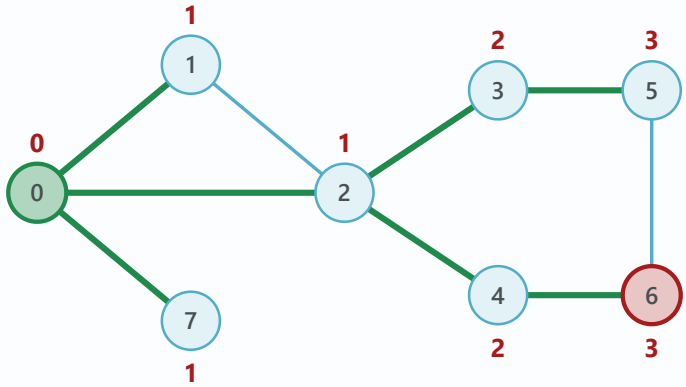
açık kapalı yol duvar sol üst g · sağ üst h · orta f

B çıkarıldı: yol A → yukarı → sağ üst çapraz → sol üst çapraz → duvar üstünden üç adım sola, maliyet **68** (2 çapraz × 14 + 4 düz × 10). 18 hücre genişletildi.

İLGİLİ YÖNTEMLER

BFS: Ağırlıksız Çizgede En Kısa Yol

- Tüm kenarlar eşit (ağırlık 1) ise en kısa yol = en az kenarlı yol.
- **Genişlik öncelikli arama** (BFS) düğümleri kaynağa uzaklık sırasıyla, katman katman ziyaret eder: d[v] = d[u] + 1.
- Kuyruk (FIFO), Dijkstra'nın öncelikli kuyruğun yerini alır: **O(V + E)**.
- Örnek (yandaki çizge): 0 → 6 en kısa yolu **0 → 2 → 4 → 6** (3 kenar).
- **İki yönlü arama** (bidirectional): BFS hem kaynaktan hem hedeften başlatılır, iki dalga buluşunca durulur; yaklaşık $2 \cdot b^{d/2}$ düğüm açılır.



Sayılar: BFS düzeyi (kenar sayısı). Yeşil: BFS ağacı.

KARŞILAŞTIRMA

En Kısa Yol Algoritmalarının Karşılaştırması

Algoritma	Problem	Negatif kenar	Negatif döngü	Zaman	Yöntem
BFS	tek kaynak, ağırlıksız	—	—	$O(V + E)$	katman katman
Dijkstra	tek kaynak	hayır	hayır	$O(V^2)$ / $O((V + E) \log V)$	açgözlü
Bellman-Ford	tek kaynak	evet	tespit eder	$O(V \cdot E)$	tekrarlı gevşetme (DP)
Floyd-Warshall	tüm çiftler	evet	tespit eder	$\Theta(V^3)$	dinamik programlama
A*	tek çift	hayır	hayır	en kötü $O(b^d)$	sezgisel + açgözlü

Ağırlıksız mı?
BFS yeterli.

Ağırlıklar ≥ 0?
Dijkstra; hedef tek ve iyi h varsa A*.

Negatif kenar?
Bellman-Ford.

Tüm çiftler, küçük V?
Floyd-Warshall.

Özet

Konu	Hatırlanacaklar
Gevşetme	d[u] + w < d[v] ise d[v] ve π[v] güncellenir; tüm algoritmaların ortak işlemi
Dijkstra	en küçük d'li düğümü kesinleştir, komşularını gevşet; ağırlıklar ≥ 0; O((V + E) log V)
Bellman-Ford	tüm kenarları V − 1 tur gevşet; V. turda gevşeyen kenar = negatif döngü; O(V · E)
Floyd-Warshall	D[i][j] = min(D[i][j], D[i][k] + D[k][j]), k en dışta; D[i][i] < 0 → negatif döngü; $\Theta(V^3)$
A*	f = g + h; kabul edilebilir h ile en kısa yol; h = 0 → Dijkstra
BFS	ağırlıksız çizgede en az kenarlı yol; O(V + E)
Yol	π (Π) işaretçileriyle hedeften geriye; en kısa yol ağacı

ÖZET

Etkileşimli Simülatörler

En Kısa Yol Algoritmaları

Rastgele bağlı, ağırlıklı çizgelerde Dijkstra, Bellman-Ford ve Floyd-Warshall A düğümünden son düğüme çalıştırılır, adım günlükleri karşılaştırılır. Düğümler sürüklenebilir.

A* Yol Bulma

Duvarları değiştirilebilen bir ızgarada A*: keşfedilen hücreler ve bulunan en kısa yol (4 yön, Manhattan sezgiseli).

ÖZET

Sık Yapılan Hatalar

Negatif kenarda Dijkstra

Kesinleşmiş bir düğüme sonradan daha kısa yol açılabilir; sonuç sessizce yanlış olur. Negatif kenarda Bellman-Ford gerekir.

∞ + w taşması

`Integer.MAX_VALUE + w` negatife döner ve "daha kısa" görünür. Bu yüzden önce `d[u] != SONSUZ` denetlenir.

Floyd-Warshall döngü sırası

k ara düğüm döngüsü **en dışta** olmalı; i, j, k sırası yanlış sonuç verir.

Tur sayısı

Bellman-Ford'da V − 1 tur gerekebilir (Örnek 2); iş tek turda bitmeyebilir. Denetim turu ayrıdır.

A*'da hedefte durmak

Hedef açık listeye **eklendiğinde** değil, listeden **çıkarıldığında** durulur. Fazla tahmin eden h en kısa yolu kaçırır.

Yönsüz kenarda negatif ağırlık

Yönsüz negatif kenar u → v → u ile kendi başına negatif döngüdür.

ÖZET

Örnek Problemler

Yönlü çizge: A→B 4, A→C 2, C→B 1, B→D 5, C→D 8, C→E 10, D→E 3; kaynak A.

1. Dijkstra düğümleri hangi sırayla kesinleştirir? Son d değerleri nedir?
2. A'dan E'ye en kısa yolu π dizisiyle yazın.
3. Kenarlar alfabetik sırayla (A→B, A→C, B→D, C→B, C→D, C→E, D→E) incelenirse Bellman-Ford'un her turunda kaç gevşetme olur?
4. 1→2 (4), 2→3 (1), 1→3 (7), 3→1 (−2) çizgesinin Floyd-Warshall son D matrisi nedir? Negatif döngü var mı?
5. V = 1000, E = 5000 ve negatif kenarlar var. Hangi algoritma, yaklaşık kaç işlem?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. Dijkstra Örnek 1'deki çizgede kaynağı **5** alarak tüm uzaklıkları bulun.
2. `dijkstra(kaynak, hedef)`: hedef kuyruktan çıkınca duran sürümü yazın.
3. Her düğüm için **en kısa yol sayısını** da hesaplayan Dijkstra yazın.
4. Bellman-Ford'a erken durma ekleyin; en iyi durumda karmaşıklık nedir?
5. Floyd-Warshall'ı boolean matrisle **geçişli kapanış** (Warshall) için uyarlayın.
6. Manhattan uzaklığı 4 yönlü ızgarada neden kabul edilebilir, 8 yönlüde neden değildir?

▶ **Cevaplar / ipuçları**

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022 (Bölüm 22–23).
- Sedgewick ve Wayne, *Algorithms*, 4. baskı, 2011 (Bölüm 4.4: Shortest Paths).
- E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", *Numerische Mathematik* 1, 1959.
- R. Bellman, "On a Routing Problem", *Quarterly of Applied Mathematics* 16, 1958.
- R. W. Floyd, "Algorithm 97: Shortest Path", *Communications of the ACM* 5(6), 1962; S. Warshall, "A Theorem on Boolean Matrices", *Journal of the ACM* 9(1), 1962.
- P. E. Hart, N. J. Nilsson ve B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. SSC* 4(2), 1968.

Son

Bölüm 4.2: En Kısa Yol Algoritmaları

Önceki bölüm: Bölüm 4.1: Çizge Gezinme Algoritmaları · Sonraki bölüm: Bölüm 4.3: Minimum Kapsayan Ağaç Algoritmaları · Terim sözlüğü