

Bölüm 7: Yaklaşım Algoritmaları

Algoritmalar

Yaklaşım oranı

Gezgin satıcı (TSP)

En yakın komşu ve MST

2-opt yerel arama

İş çizelgeleme

Düğüm kapsama

Sırt çantası

Power Iteration

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

Öğrenme Çıktıları

Kavram

Yaklaşım (approximation) algoritmasının ne olduğunu, **yaklaşım oranını** ve sezgisel yöntemlerden farkını açıklamak.

TSP

Gezgin satıcı problemi için en yakın komşu, MST tabanlı 2-yaklaşım ve 2-opt yerel aramasını adım adım uygulamak.

Çizelgeleme ve Kapsama

İş çizelgeleme (liste, LPT, son tarihli iş sıralama) ve düğüm kapsama 2-yaklaşımını Java ile yazıp izlemek.

Sayısal Yakınsama

Power Iteration ile en büyük özdeğer ve özvektörün yinelemeli olarak nasıl yakınsandığını hesaplamak.

Yaklaşım Algoritmaları (Approximation Algorithms)

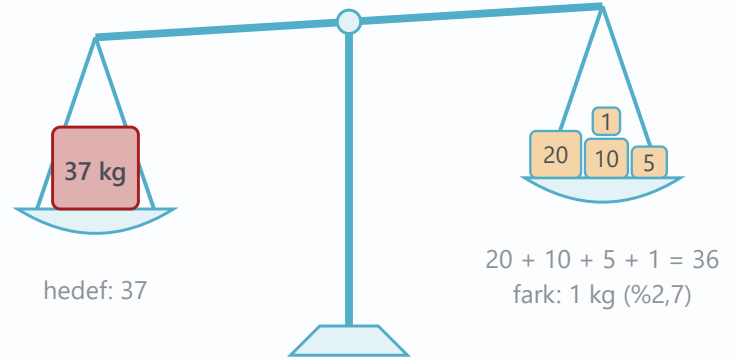
- Bazı problemlerin **kesin (optimal)** çözümünü bulmak çok zor ya da pratikte **imkânsız** olabilir: bilinen tüm kesin algoritmalar üstel zaman ister.
- Yaklaşım algoritmaları **mükemmel çözümü** aramak yerine, optimal çözüme **mümkün olduğunca yakın** çözümler üretir.
- **Polinom zamanda** çalışırlar: karmaşık problemler makul sürede çözülebilir.
- Çözümün optimal çözüme **belli bir oran ya da yüzde** içinde yakın olduğu çoğu zaman **kanıtlanabilir**.
- Hesaplama kaynakları (zaman, bellek) **kısıtlı** olduğunda tercih edilir.

Takas: **optimallikten** biraz vazgeçilir, karşılığında **verimlilik** ve çözüm kalitesi için bir **garanti** kazanılır.

GİRİŞ

Sezgi: Terazî Benzetmesi

- Terazinin bir kefesine bir ağırlık konduğunda, diğer kefeye **tam olarak aynı** ağırlığı koymak zor olabilir.
- Bu durumda, eldeki çeşitli **küçük ağırlıklar** kullanılarak büyük ağırlığa **olabildiğince yakın** bir denge kurulur.
- Elimizde yalnızca 20, 10, 5 ve 1 kg varsa ve her biri bir kez kullanılabilirse, 37 kg'a en yakın denge 36 kg'dır.



Yaklaşım algoritması da böyle çalışır: kesin dengeyi (optimumu) aramak yerine, **hızlıca** bulunan ve hatası **sınırlı** olan bir dengeyle yetinir.

GİRİŞ

Yaklaşım Oranı (Approximation Ratio)

C : algoritmanın bulduğu çözümün maliyeti, C^* : optimal çözümün maliyeti. Algoritma her girdi için

$$\max(C / C^*, C^* / C) \leq \rho(n)$$

koşulunu sağlıyorsa bir **$\rho(n)$ -yaklaşım algoritmasıdır** ($\rho \geq 1$; $\rho = 1$ ise algoritma optimaldir).

En küçükleme (minimization)

$C \leq \rho \cdot C^*$. Örnek: 2-yaklaşım bir düğüm kapsama algoritması, en fazla optimumun **iki katı** düğüm seçer.

En büyükleme (maximization)

$C \geq C^* / \rho$. Örnek: 2-yaklaşım bir sırt çantası algoritması, optimal değer en az **yarısını** garanti eder.

Oran bir **en kötü durum** garantisidir; pratikte çözümler çoğu zaman optimuma çok daha yakındır.

GİRİŞ

Yaklaşım Algoritması, Sezgisel ve Yaklaşım Şemaları

Yaklaşım	Garanti	Örnek
Kesin algoritma	Optimal; ama NP-zor problemlerde üstel zaman	Kaba kuvvet TSP: $(n-1)!/2$ tur
Yaklaşım algoritması	Polinom zaman + kanıtlanmış oran ρ	Düğüm kapsama: $\rho = 2$
PTAS	Her sabit $\epsilon > 0$ için $(1 + \epsilon)$ oranı, n 'de polinom (ϵ 'da üstel olabilir)	Öklid TSP (Arora, 1998)
FPTAS	$(1 + \epsilon)$ oranı, hem n 'de hem $1/\epsilon$ 'da polinom	Sırt çantası: $O(n^3/\epsilon)$
Sezgisel (heuristic)	Garanti yok; pratikte genellikle iyi	2-opt, en yakın komşu
Metasezgisel	Garanti yok; problemde bağımsız arama çerçevesi	Genetik algoritma, karınca kolonisi

Rastgele yuvarlama (randomized rounding) gibi rastgele yaklaşım algoritmaları da vardır; beklenen değerde bir oran garanti ederler (Bölüm 8).

GİRİŞ

Neden Kesin Çözüm Değil? P, NP ve NP-Zor

P

Polinom zamanda **çözülebilir** karar problemleri (ör. en kısa yol, MST).

NP

Önerilen bir çözümü polinom zamanda **doğrulanabilir** karar problemleri.

NP-zor (NP-hard)

NP'deki her problem buna polinom zamanda indirgenebilir; en az NP'deki en zor problem kadar zordur.

NP-tam (NP-complete)

Hem NP'de hem NP-zor. Örnekler: SAT, 3-SAT, düğüm kapsama, Hamilton çevrimi.

- TSP, düğüm kapsama, iş çizelgeleme ve sırt çantasının **en iyileme** sürümleri NP-zordur.
- $P \neq NP$ ise bunların hiçbirisi için polinom zamanlı **kesin** algoritma yoktur → **yaklaşım algoritmaları** gerekir.

GİRİŞ

Yaklaşım Algoritmalarının Uygulamaları

Gezgin satıcı (TSP)

Satıcının tüm müşterilere en kısa toplam yolla uğrayıp geri döndüğü rotaya karar vermek: lojistik, kargo, devre kartı delme.

Sırt çantası (knapsack)

Çantaya sığacak ve en büyük faydayı sağlayacak eşyaları seçmek: bütçe, yük ve portföy planlama.

İş (job) çizelgeleme

Birden fazla işin ya da sürecin makinelerde/işlemcilerde en verimli şekilde yürütülmesi.

Düğüm kapsama ve sayısal yöntemler

Ağlara en az sayıda izleme noktası yerleştirmek; büyük matrislerin özdeğerini yinelemeli yakınsamak (PageRank).

Yaklaşım Algoritmalarının Türleri

Basitleştirme

Problemin **daha basit bir sürümü** çözülür (ör. tam sayı kısıtı gevşetilir: LP gevşetmesi, MST ile TSP alt sınırı).

Açgözlü (greedy)

Her adımda **o an en iyi görünen** seçenek seçilerek ilerlenir (en yakın komşu, LPT, oran sıralı sırt çantası).

Yerel arama (local search)

Mevcut çözüm **küçük değişikliklerle** iyileştirilir; daha iyi komşu kalmayınca durulur (2-opt).

Primal–dual

LP'nin birincil ve eşlenik (dual) çözümleri birlikte büyütülür; kapsama problemlerinde sık kullanılır.

Rastgele

Rastgele yuvarlama ya da örnekleme; beklenen oran garantisi.

Metasezgisel

Benzetimli tavlama, genetik algoritma, karınca kolonisi.

Gezgin Satıcı Problemi (TSP)

- **En iyileme problemi:** satıcı, bir dizi şehri **en kısa toplam uzaklıkta** birer kez ziyaret edip başladığı şehre dönmek ister.
- **Kombinatorial problem:** çözüm, tüm şehirlerin **gezilme sırasıdır** (bir permütasyon).
- **NP-zor:** çok sayıda şehir için en iyi rotayı bulmak pratikte zordur.
- **Girdi:** n şehir (0, 1, 2, ... ya da A, B, C, ...) ve d_{ij} : i ve j şehirleri arasındaki uzaklık.
- Simetrik TSP: $d_{ij} = d_{ji}$; çizge kuramında: tam çizgedeki en hafif **Hamilton çevrimi**.

d	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

Örnek: 5 şehrin uzaklık matrisi
(sonraki slaytlarda kullanılır).

Problemin Çözümü

Tam çözüm

Tüm olası rotaları kontrol etmek (kaba kuvvet) ya da dinamik programlama (Held–Karp, $O(n^2 \cdot 2^n)$).

Yaklaşık çözümler

Sezgisel algoritmalar: en yakın komşu (nearest neighbor), MST tabanlı 2-yaklaşım, Christofides, genetik algoritmalar.

Optimizasyon araçları

Yerel arama (2-opt), benzetimli tavlama (simulated annealing), karınca kolonisi optimizasyonu (ant colony optimization).

n	$(n-1)!/2$ farklı tur
5	12
10	181 440
15	$\approx 4,4 \cdot 10^{10}$
20	$\approx 6,1 \cdot 10^{16}$
25	$\approx 3,1 \cdot 10^{23}$

Başlangıç şehri sabitlenir ve iki yön aynı sayılır. Saniyede 10^9 tur denense bile $n = 25$ için $\approx 10^7$ yıl gerekir.

En Yakın Komşu (Nearest Neighbor) Sezgiseli

1. Bir başlangıç şehri seç; onu ziyaret edildi olarak işaretle.
2. Bulunduğun şehirden **ziyaret edilmemiş en yakın** şehre git (açgözlü seçim).
3. Tüm şehirler ziyaret edilene kadar 2. adımı tekrarla.
4. Son şehirden **başlangıç şehrine dön** ve turu kapat.

Artıları

- + Çok basit; $O(n^2)$ zaman.
- + Genellikle optimumun %25 kadar üzerinde tur verir.
- + Yerel arama için iyi bir **başlangıç turu**.

Eksileri

- Sabit bir yaklaşım oranı **yoktur**; metrik TSP'de bile oran $\Theta(\log n)$ olabilir.
- Son dönüş kenarı çok uzun kalabilir.

En Yakın Komşu: Kodu Adım Adım İzle

```
1 int[] enYakinKomsu(int[][] d, int bas) {  
    ▶ int n = d.length, toplam = 0, simdiki = bas;  
    ▶ boolean[] ziyaret = new boolean[n];  
    ▶ int[] tur = new int[n + 1];  
    ▶ tur[0] = bas; ziyaret[bas] = true;  
  
    6 for (int k = 1; k < n; k++) {  
        7 int enYakin = -1, enAz = Integer.MAX_VALUE;  
        8 for (int j = 0; j < n; j++)  
            9 if (!ziyaret[j] && d[simdiki][j] < enAz) {  
                10 enAz = d[simdiki][j]; enYakin = j; }  
        11 // en yakın şehre git  
        12 tur[k] = enYakin; ziyaret[enYakin] = true;  
        13 toplam += enAz; simdiki = enYakin;  
        14 }  
        15 // başlangıca dön  
        16 tur[n] = bas; toplam += d[simdiki][bas];  
        17 return tur;  
        18 }
```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0

Başlangıç şehri 0: ziyaret[0] = true , tur = [0] , toplam = 0 .

Adım 1 / 11

DEĞİŞKENLER

k = - simdiki = 0 enYakin = -
enAz = - toplam = 0 tur = [0]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2   int n = d.length, toplam = 0, simdiki = bas;
3   boolean[] ziyaret = new boolean[n];
4   int[] tur = new int[n + 1];
5   tur[0] = bas; ziyaret[bas] = true;
6   for (int k = 1; k < n; k++) {
7     ▶ int enYakin = -1, enAz = Integer.MAX_VALUE;
8     ▶ for (int j = 0; j < n; j++)
9       ▶ if (!ziyaret[j] && d[simdiki][j] < enAz) {
10        ▶   enAz = d[simdiki][j]; enYakin = j; }
11   // en yakın şehre git
12   tur[k] = enYakin; ziyaret[enYakin] = true;
13   toplam += enAz; simdiki = enYakin;
14 }
15 // başlangıca dön
16 tur[n] = bas; toplam += d[simdiki][bas];
17 return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0

0. satır taranır, ziyaret edilmemişler: $d[0][1] = 10$, $d[0][2] = 15$, $d[0][3] = 20$, $d[0][4] = 25 \rightarrow$ en küçüğü **1 (10)**.

Adım 2 / 11

DEĞİŞKENLER

k = 1 simdiki = 0 enYakin = 1
 enAz = 10 toplam = 0 tur = [0]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         int enYakin = -1, enAz = Integer.MAX_VALUE;
8         for (int j = 0; j < n; j++)
9             if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                 enAz = d[simdiki][j]; enYakin = j; }
11         // en yakın şehre git
12         ▶ tur[k] = enYakin; ziyaret[enYakin] = true;
13         ▶ toplam += enAz; simdiki = enYakin;
14     }
15     // başlangıca dön
16     tur[n] = bas; toplam += d[simdiki][bas];
17     return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1

tur[1] = 1, toplam += 10 → 10; artık
simdiki = 1.

Adım 3 / 11

DEĞİŞKENLER

k = 1 simdiki = 1 enYakin = 1

enAz = 10 toplam = 10 tur = [0, 1]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         ▶ int enYakin = -1, enAz = Integer.MAX_VALUE;
8         ▶ for (int j = 0; j < n; j++)
9             ▶ if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                enAz = d[simdiki][j]; enYakin = j; }
11         // en yakın şehre git
12         tur[k] = enYakin; ziyaret[enYakin] = true;
13         toplam += enAz; simdiki = enYakin;
14     }
15     // başlangıca dön
16     tur[n] = bas; toplam += d[simdiki][bas];
17     return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1

1. satır taranır, ziyaret edilmemişler: $d[1][2] = 35$, $d[1][3] = 25$, $d[1][4] = 30$ → en küçüğü **3** (25).

Adım 4 / 11

DEĞİŞKENLER

k = 2

simdiki = 1

enYakin = 3

enAz = 25

toplam = 10

tur = [0, 1]

En Yakın Komşu: Kodu Adım Adım İzle

```
1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         int enYakin = -1, enAz = Integer.MAX_VALUE;
8         for (int j = 0; j < n; j++)
9             if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                 enAz = d[simdiki][j]; enYakin = j; }
11         // en yakın şehre git
12         ▶ tur[k] = enYakin; ziyaret[enYakin] = true;
13         ▶ toplam += enAz; simdiki = enYakin;
14     }
15     // başlangıca dön
16     tur[n] = bas; toplam += d[simdiki][bas];
17     return tur;
18 }
```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3

tur[2] = 3, toplam += 25 → 35; artık
simdiki = 3.

Adım 5 / 11

DEĞİŞKENLER

k = 2 simdiki = 3 enYakin = 3
enAz = 25 toplam = 35 tur = [0, 1, 3]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2   int n = d.length, toplam = 0, simdiki = bas;
3   boolean[] ziyaret = new boolean[n];
4   int[] tur = new int[n + 1];
5   tur[0] = bas; ziyaret[bas] = true;
6   for (int k = 1; k < n; k++) {
7     ▶ int enYakin = -1, enAz = Integer.MAX_VALUE;
8     ▶ for (int j = 0; j < n; j++)
9       ▶ if (!ziyaret[j] && d[simdiki][j] < enAz) {
10        ▶   enAz = d[simdiki][j]; enYakin = j; }
11   // en yakın şehre git
12   tur[k] = enYakin; ziyaret[enYakin] = true;
13   toplam += enAz; simdiki = enYakin;
14 }
15 // başlangıca dön
16 tur[n] = bas; toplam += d[simdiki][bas];
17 return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3

3. satır taranır, ziyaret edilmemişler: d[3][2] = 30, d[3][4] = 40 → en küçüğü **2** (30).

Adım 6 / 11

DEĞİŞKENLER

k = 3

simdiki = 3

enYakin = 2

enAz = 30

toplam = 35

tur = [0, 1, 3]

En Yakın Komşu: Kodu Adım Adım İzle

```
1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         int enYakin = -1, enAz = Integer.MAX_VALUE;
8         for (int j = 0; j < n; j++)
9             if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                 enAz = d[simdiki][j]; enYakin = j; }
11         // en yakın şehre git
12         ▶ tur[k] = enYakin; ziyaret[enYakin] = true;
13         ▶ toplam += enAz; simdiki = enYakin;
14     }
15     // başlangıca dön
16     tur[n] = bas; toplam += d[simdiki][bas];
17     return tur;
18 }
```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3 → 2

tur[3] = 2, toplam += 30 → 65; artık
simdiki = 2.

Adım 7 / 11

DEĞİŞKENLER

k = 3 simdiki = 2 enYakin = 2

enAz = 30 toplam = 65

tur = [0, 1, 3, 2]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         ▶ int enYakin = -1, enAz = Integer.MAX_VALUE;
8         ▶ for (int j = 0; j < n; j++)
9             ▶ if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                 enAz = d[simdiki][j]; enYakin = j; }
11     // en yakın şehre git
12     tur[k] = enYakin; ziyaret[enYakin] = true;
13     toplam += enAz; simdiki = enYakin;
14 }
15 // başlangıca dön
16 tur[n] = bas; toplam += d[simdiki][bas];
17 return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3 → 2

2. satır taranır, ziyaret edilmemişler: d[2][4] = 35 → en küçüğü 4 (35).

Adım 8 / 11

DEĞİŞKENLER

k = 4 simdiki = 2 enYakin = 4

enAz = 35 toplam = 65

tur = [0, 1, 3, 2]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         int enYakin = -1, enAz = Integer.MAX_VALUE;
8         for (int j = 0; j < n; j++)
9             if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                 enAz = d[simdiki][j]; enYakin = j; }
11         // en yakın şehre git
12         ▶ tur[k] = enYakin; ziyaret[enYakin] = true;
13         ▶ toplam += enAz; simdiki = enYakin;
14     }
15     // başlangıca dön
16     tur[n] = bas; toplam += d[simdiki][bas];
17     return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3 → 2 → 4

tur[4] = 4, toplam += 35 → 100; artık
simdiki = 4.

Adım 9 / 11

DEĞİŞKENLER

k = 4 simdiki = 4 enYakin = 4

enAz = 35 toplam = 100

tur = [0, 1, 3, 2, 4]

En Yakın Komşu: Kodu Adım Adım İzle

```

1 int[] enYakinKomsu(int[][] d, int bas) {
2     int n = d.length, toplam = 0, simdiki = bas;
3     boolean[] ziyaret = new boolean[n];
4     int[] tur = new int[n + 1];
5     tur[0] = bas; ziyaret[bas] = true;
6     for (int k = 1; k < n; k++) {
7         int enYakin = -1, enAz = Integer.MAX_VALUE;
8         for (int j = 0; j < n; j++)
9             if (!ziyaret[j] && d[simdiki][j] < enAz) {
10                 enAz = d[simdiki][j]; enYakin = j; }
11         // en yakın şehre git
12         tur[k] = enYakin; ziyaret[enYakin] = true;
13         toplam += enAz; simdiki = enYakin;
14     }
15     // başlangıca dön
16     ▶ tur[n] = bas; toplam += d[simdiki][bas];
17     return tur;
18 }

```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3 → 2 → 4 → 0

Tüm şehirler ziyaret edildi; başlangıca dönülür:

toplam += d[4][0] = 25 → **125**.

Adım 10 / 11

DEĞİŞKENLER

k = 5 simdiki = 4 enYakin = -

enAz = - toplam = 125

tur = [0, 1, 3, 2, 4, 0]

En Yakın Komşu: Kodu Adım Adım İzle

```
1 int[] enYakinKomsu(int[][] d, int bas) {  
2     int n = d.length, toplam = 0, simdiki = bas;  
3     boolean[] ziyaret = new boolean[n];  
4     int[] tur = new int[n + 1];  
5     tur[0] = bas; ziyaret[bas] = true;  
6     for (int k = 1; k < n; k++) {  
7         int enYakin = -1, enAz = Integer.MAX_VALUE;  
8         for (int j = 0; j < n; j++)  
9             if (!ziyaret[j] && d[simdiki][j] < enAz) {  
10             enAz = d[simdiki][j]; enYakin = j; }  
11         // en yakın şehre git  
12         tur[k] = enYakin; ziyaret[enYakin] = true;  
13         toplam += enAz; simdiki = enYakin;  
14     }  
15     // başlangıca dön  
16     tur[n] = bas; toplam += d[simdiki][bas];  
17     return tur;  
18 }
```

	0	1	2	3	4
0	0	10	15	20	25
1	10	0	35	25	30
2	15	35	0	30	35
3	20	25	30	0	40
4	25	30	35	40	0

tur: 0 → 1 → 3 → 2 → 4 → 0

Tur 0 → 1 → 3 → 2 → 4 → 0, uzunluk **125**. (Kaba kuvvetle doğrulandı: bu örnekte optimumdur.)

Adım 11 / 11

DEĞİŞKENLER

k = 5 simdiki = 4 enYakin = -

enAz = - toplam = 125

tur = [0, 1, 3, 2, 4, 0]

Kaba Kuvvetle Karşılaştırma: 12 Farklı Tur

Tur	Uzunluk
$0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 0$	125
$0 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 0$	125
$0 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 0$	125
$0 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 0$	125
$0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 0$	125
$0 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 3 \rightarrow 0$	125

Tur	Uzunluk
$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 0$	140
$0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 0$	140
$0 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 0$	140
$0 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 0$	140
$0 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 0$	140
$0 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 0$	140

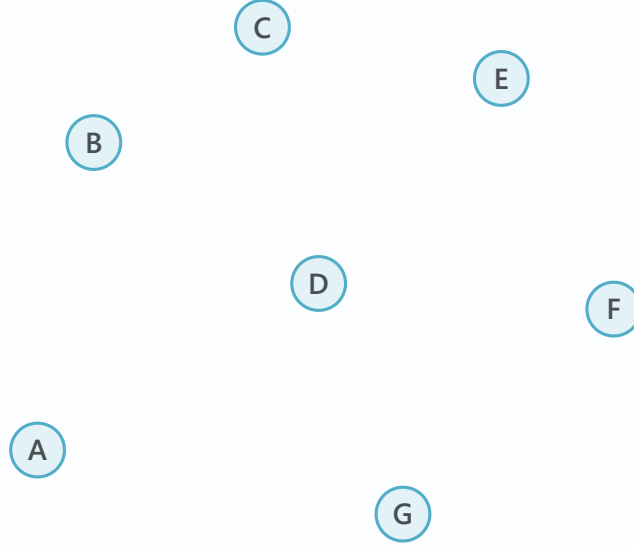
- $(5-1)!/2 = 12$ tur denendi: optimum **125**; en yakın komşu da 125 buldu (bu örnek için şanslı bir durum).
- Bu matris **üçgen eşitsizliğini** sağlamaz: $d_{12} = 35 > d_{10} + d_{02} = 10 + 15$. Yani örnek **metrik değildir**.

Üçgen Eşitsizliği ve Yaklaşım Sınırları

Metrik TSP: her i, j, k için $d_{ij} \leq d_{ik} + d_{kj}$ (doğrudan gitmek hiçbir zaman daha uzun değildir)

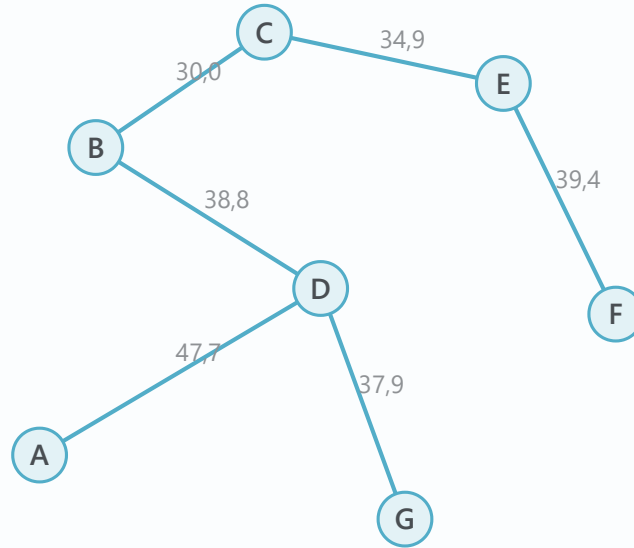
Durum	Oran	Açıklama
Genel TSP	yok	$P \neq NP$ ise hiçbir sabit p için polinom zamanlı p -yaklaşım yoktur: böyle bir algoritma Hamilton çevrimi problemini çözerdi.
Metrik TSP, MST tabanlı	2	MST + önce-kök (preorder) gezinme + kısayol; $O(n^2)$.
Metrik TSP, Christofides	1,5	MST + tek dereceli düğümlerde en küçük mükemmel eşleştirme + Euler turu (1976).
Öklid TSP	$1 + \epsilon$	Düzlemdeki noktalar için PTAS vardır (Arora, Mitchell).
En yakın komşu	$\Theta(\log n)$	Metrik durumda bile sabit oran garanti etmez.

Metrik TSP için MST Tabanlı 2-Yaklaşım



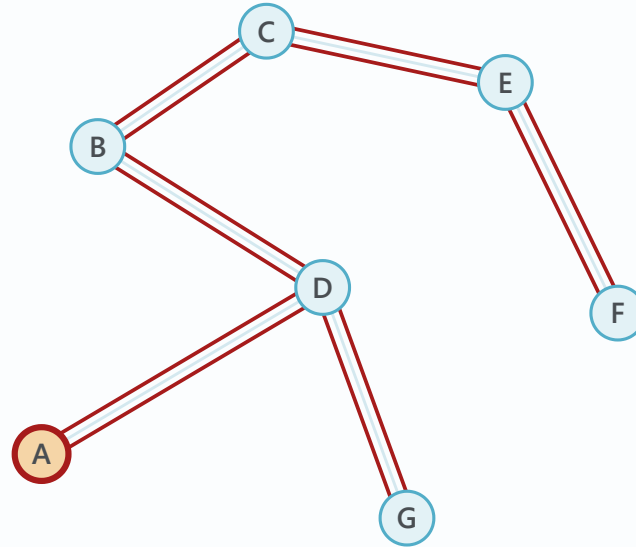
Metrik örnek: düzlemde 7 şehir; uzaklıklar Öklid uzaklığıdır (üçgen eşitsizliği sağlanır).

Metrik TSP için MST Tabanlı 2-Yaklaşım



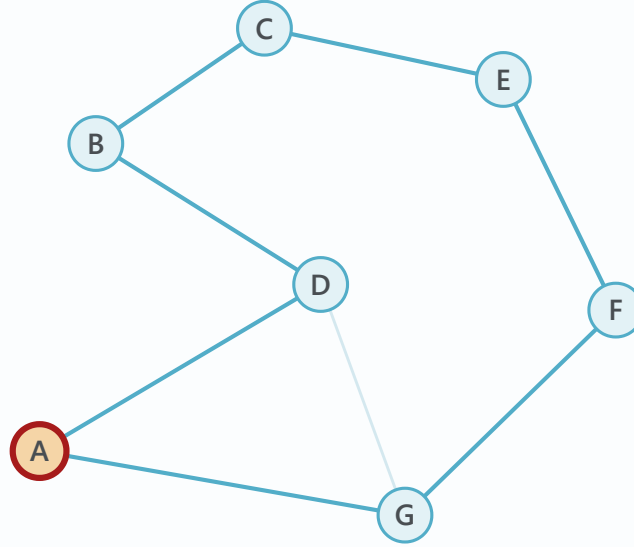
1) Prim ile **minimum kapsayan ağaç** (MST) kurulur: toplam ağırlık $w(T) = 228,8$. Bir turdan bir kenar silinince kapsayan ağaç kalır $\rightarrow w(T) \leq \mathbf{OPT}$.

Metrik TSP için MST Tabanlı 2-Yaklaşım



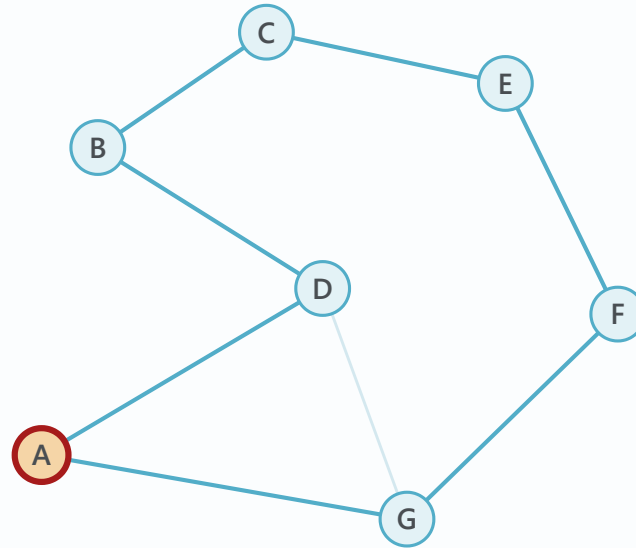
2) Ağaç A'dan derinlik öncelikli **iki kez** gezilir (her kenar bir gidiş bir dönüş): $A \rightarrow D \rightarrow B \rightarrow C \rightarrow E \rightarrow F \rightarrow E \rightarrow C \rightarrow B \rightarrow D \rightarrow G \rightarrow D \rightarrow A$. Uzunluk $2 \cdot w(T) = 457,6$.

Metrik TSP için MST Tabanlı 2-Yaklaşım



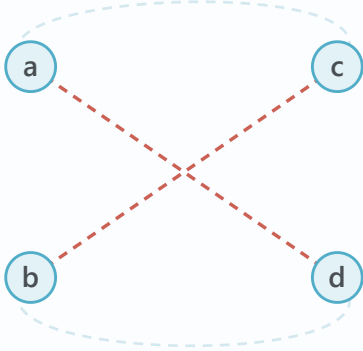
- 3) Tekrar eden şehirler atlanır (**kısayol**): önce-kök sırası $A \rightarrow D \rightarrow B \rightarrow C \rightarrow E \rightarrow F \rightarrow G \rightarrow A$.
Üçgen eşitsizliği sayesinde kısayol yolu uzatmaz: tur = 287,7.

Metrik TSP için MST Tabanlı 2-Yaklaşım

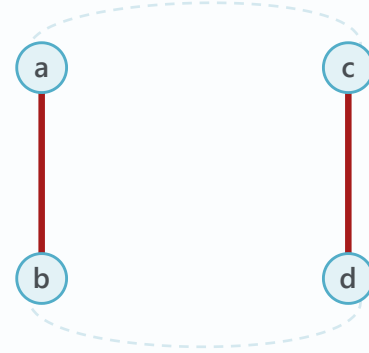


Sonuç: $\text{tur} \leq 2 \cdot w(T) \leq 2 \cdot \text{OPT}$. Burada $\text{tur} = 287,7$, optimum (kaba kuvvet, 360 tur) = 282,5
→ oran $1,02 \leq 2$.

Yerel Arama: 2-opt Hamlesi



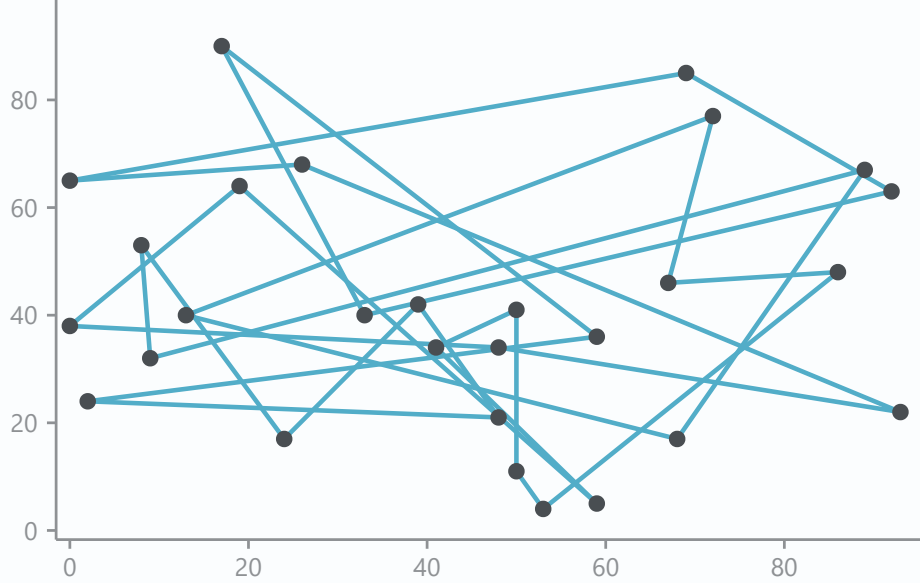
Önce: (a, b) ve (c, d) kenarları çaprazlanıyor.



Sonra: (a, c) ve (b, d); aradaki yol ters çevrildi.

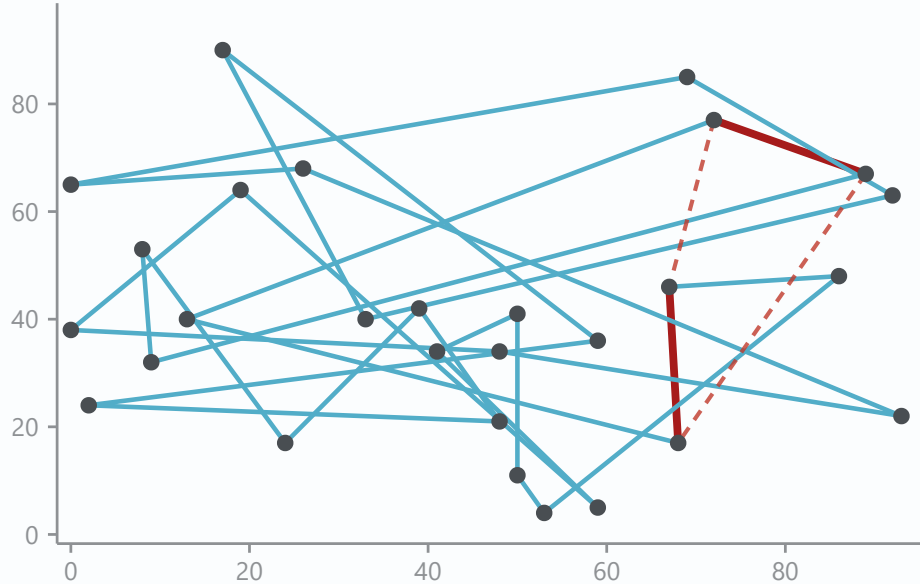
- Turdan iki kenar çıkar, uçları **çapraz** bağla: b ... c arasındaki alt yol **ters çevrilir**.
- Kazanç $\Delta = d(a,c) + d(b,d) - d(a,b) - d(c,d) < 0$ ise hamle yapılır. Öklid düzleminde her **kesişme** böyle kaldırılabilir.
- İyileştiren hamle kalmayınca tur **2-opt yerel optimumudur** (genel optimum olması gerekmez).

2-opt ile Turu İyileştirme



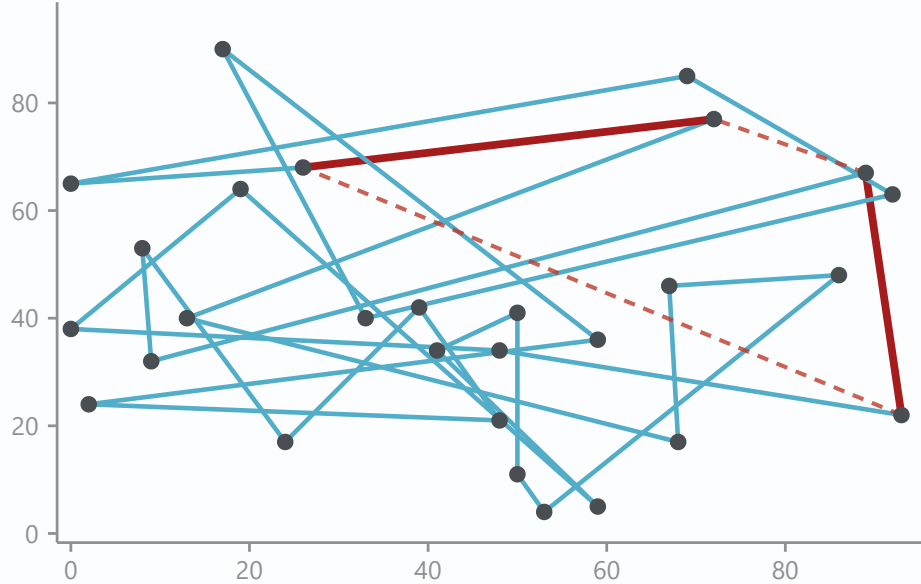
28 şehir, **rastgele** başlangıç turu: uzunluk **1269,8**. Kenarlar sürekli birbirini kesiyor.

2-opt ile Turu İyileştirme



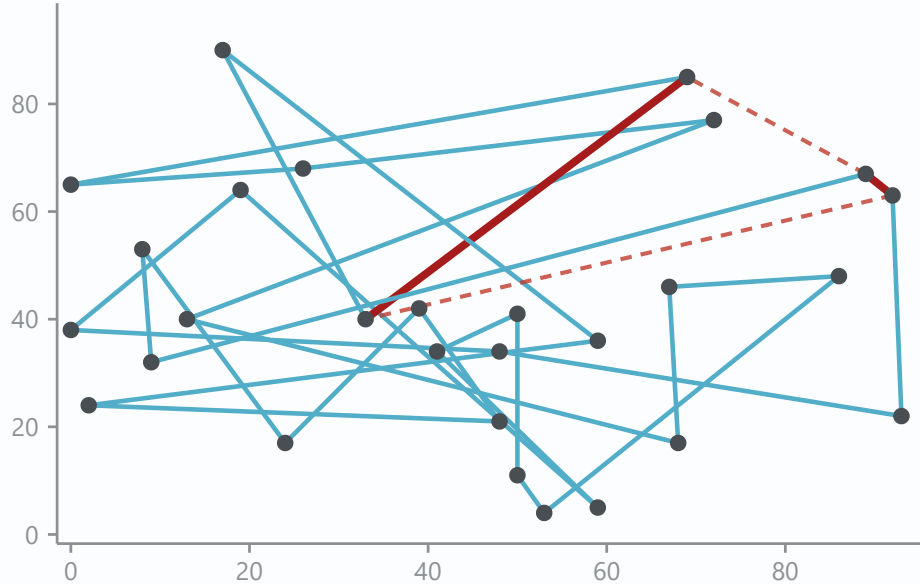
1. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -36,9 \rightarrow$ uzunluk **1232,9**.

2-opt ile Turu İyileştirme



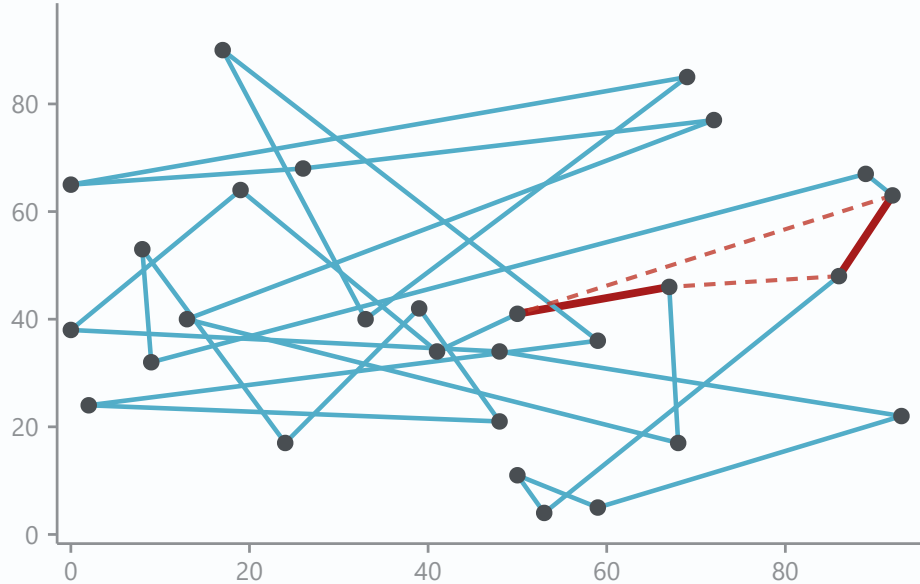
2. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -8,9$
→ uzunluk **1224,0**.

2-opt ile Turu İyileştirme



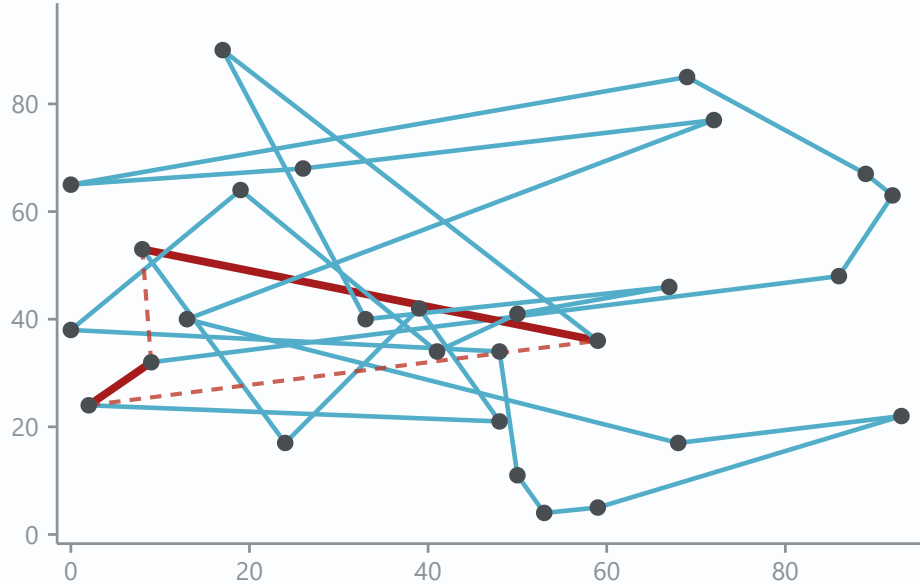
4. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -27,6 \rightarrow$ uzunluk **1187,3**.

2-opt ile Turu İyileştirme



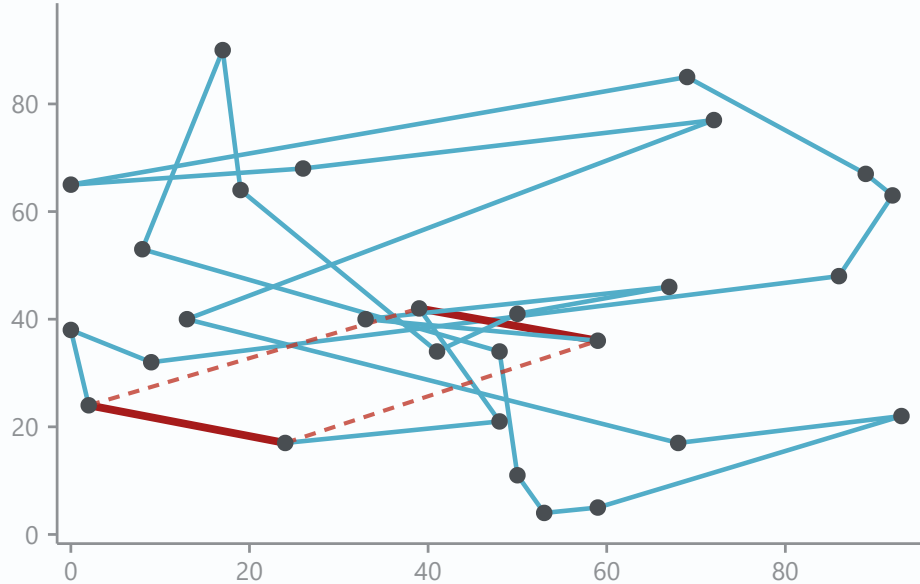
8. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -32,6 \rightarrow$ uzunluk **1111,7**.

2-opt ile Turu İyileştirme



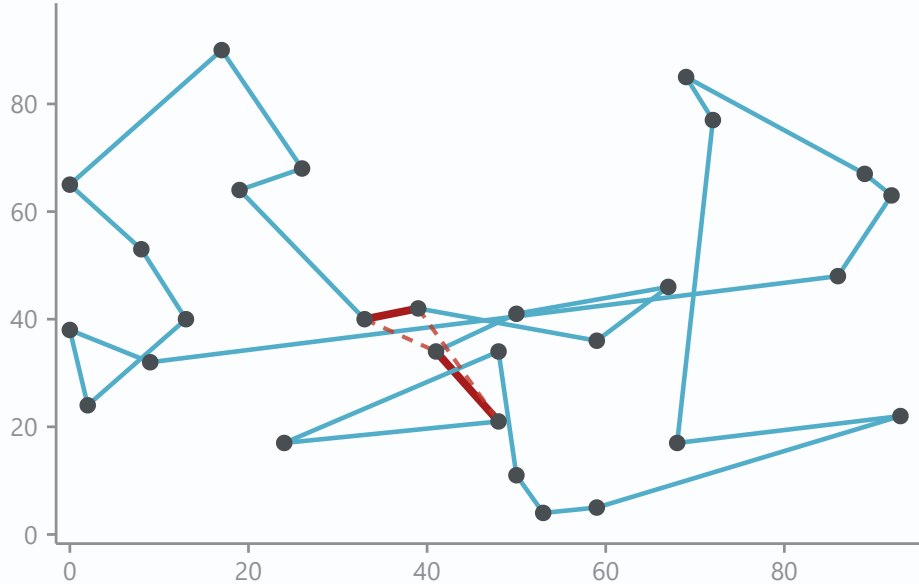
14. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -14,9 \rightarrow$ uzunluk **1005,2**.

2-opt ile Turu İyileştirme



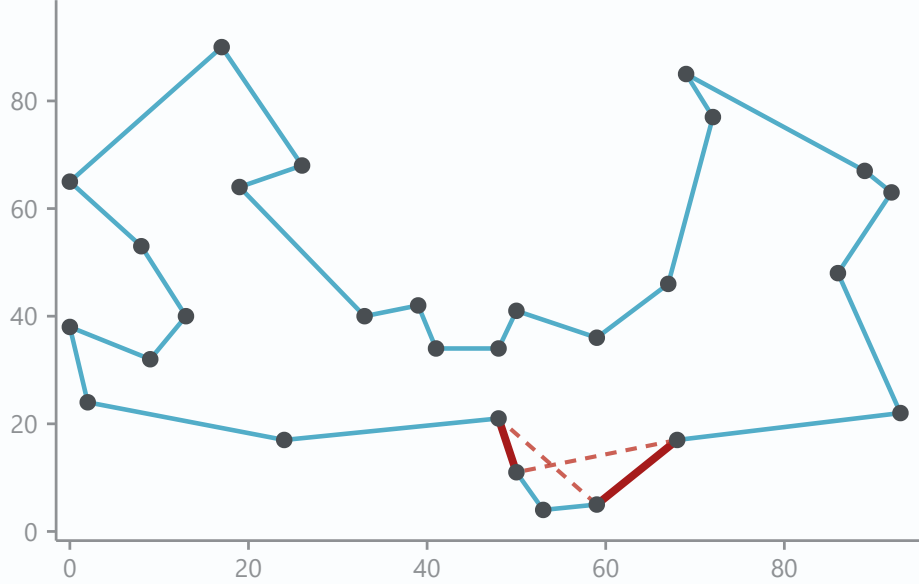
22. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -37,0 \rightarrow$ uzunluk **852,8**.

2-opt ile Turu İyileştirme



32. iyileştiren hamleden sonra: (a,b), (c,d) çıkarıldı (kesikli), (a,c), (b,d) eklendi (kalın). $\Delta = -11,8 \rightarrow$ uzunluk **595,9**.

2-opt ile Turu İyileştirme



48. ve son hamle: artık hiçbir 2-opt hamlesi turu kısaltmıyor → **yerel optimum**, uzunluk **430,8** (başlangıcın %34'ü). Kesişen kenar kalmadı.

2-opt Yerel Arama: Java

```
void ikiOpt(int[] tur, double[][] d) {
    int n = tur.length;
    boolean iyilesti = true;
    while (iyilesti) {
        iyilesti = false;
        for (int i = 0; i < n - 1; i++)
            for (int j = i + 2; j < n; j++) {
                int a = tur[i], b = tur[i + 1];
                int c = tur[j], e = tur[(j + 1) % n];
                double fark = d[a][c] + d[b][e]
                    - d[a][b] - d[c][e];
                if (fark < -1e-9) { // kısılıyor
                    tersCevir(tur, i + 1, j);
                    iyilesti = true;
                }
            }
    }
}
```

Bir tur tarama

$O(n^2)$ aday hamle; her hamle $O(1)$ 'de değerlendirilir, ters çevirme $O(n)$.

Yerel optimum

Döngü, hiçbir hamle kazanç sağlamayınca biter. Tur sayısı en kötü durumda üstel olabilir, pratikte azdır.

İyi başlangıç

Rastgele yerine en yakın komşu turuyla başlamak daha az hamle gerektirir.
Genelleştirme: 3-opt, Lin-Kernighan.

Yerel Optimumdan Kaçış: Metasezgiseller

Benzetimli tavlama

(Simulated annealing)
Kötüleştiren bir hamle de $e^{-\Delta/T}$ olasılıkla kabul edilir; "sıcaklık" T yavaşça düşürülür. Yerel optimumdan çıkabilir.

Genetik algoritma

Turlardan oluşan bir **popülasyon**; seçim, çaprazlama (crossover) ve mutasyonla nesiller boyunca iyileştirilir.

Karınca kolonisi

(Ant colony optimization)
Karıncalar kısa turların kenarlarına daha çok **feromon** bırakır; sonraki karıncalar feromonu yoğun kenarları tercih eder.

- Üçü de TSP, araç rotalama, iş çizelgeleme ve atama problemlerinde yaygındır.
- Hiçbiri yaklaşım oranı **garanti etmez**; parametre ayarı (sıcaklık, popülasyon, buharlaşma) sonucu belirgin biçimde etkiler.

İş Çizelgeleme (Job Scheduling)

- **En iyileme problemi:** belirli kaynaklarla (işçiler, makineler, işlemciler) işlerin **en verimli** şekilde planlanması.
- **Amaç:** işlerin tamamlanma süresini en aza indirmek ya da belirli ölçütlere (kâr, gecikme, son tarih) göre en iyilemek.

Makine çizelgeleme (en büyük yük)

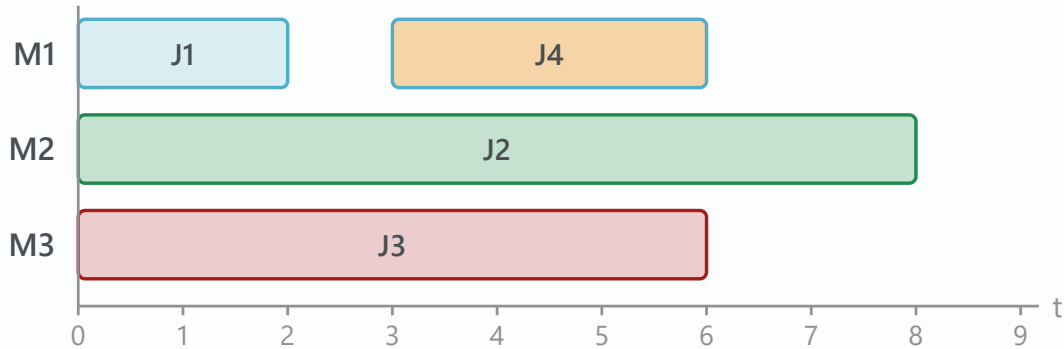
n iş, m özdeş makine; iş j 'nin süresi p_j . Son işin bittiği an $C_{\max}$ (en büyük yük, makespan) en küçük olsun. $m \geq 2$ için NP-zor.

Son tarihli iş sıralama

Her iş 1 birim sürer, bir **kârı** ve **son tarihi** (deadline) vardır. Son tarihinden önce biten işlerin toplam kârı en büyük olsun.

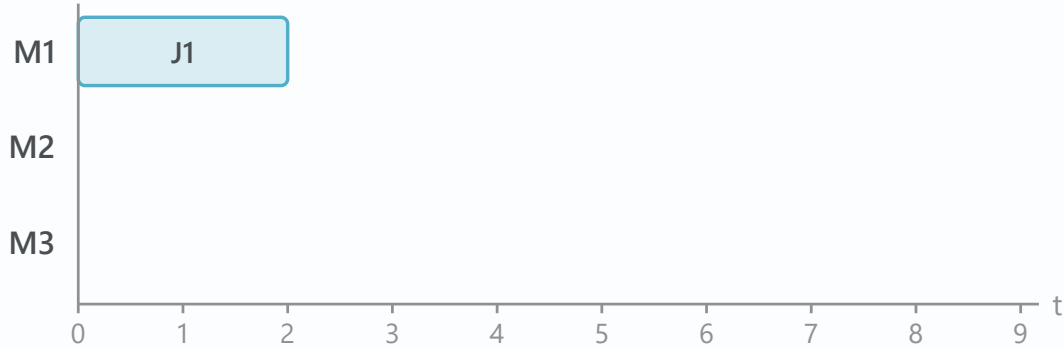
İlk problem için açgözlü **yaklaşım** algoritmaları (liste çizelgeleme, LPT), ikincisi için **optimal** bir açgözlü algoritma göreceğiz.

Makinelere İş Atama: Liste Çizelgeleme



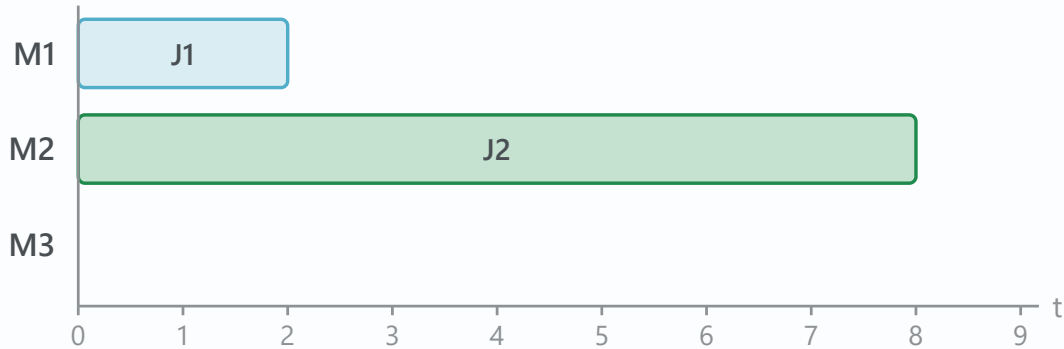
Üç özdeş makine M1–M3. İşler sırayla gelir: J1 (2), J2 (8), J3 (6) $t = 0$ 'da hazır; J4 (3) $t = 3$ 'te gelir. **Liste çizelgeleme** (Graham): sıradaki iş, **ilk boşalan** makineye atanır.

Makinelere İş Atama: Liste Çizelgeleme



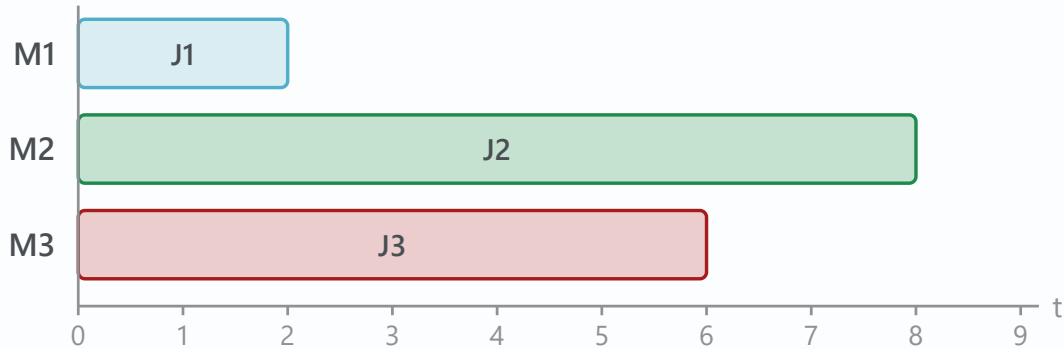
J1 (süre 2) → **M1**, $[0, 2]$ aralığında çalışır (tüm makineler boş; ilk makine seçilir).

Makinelere İş Atama: Liste Çizelgeleme



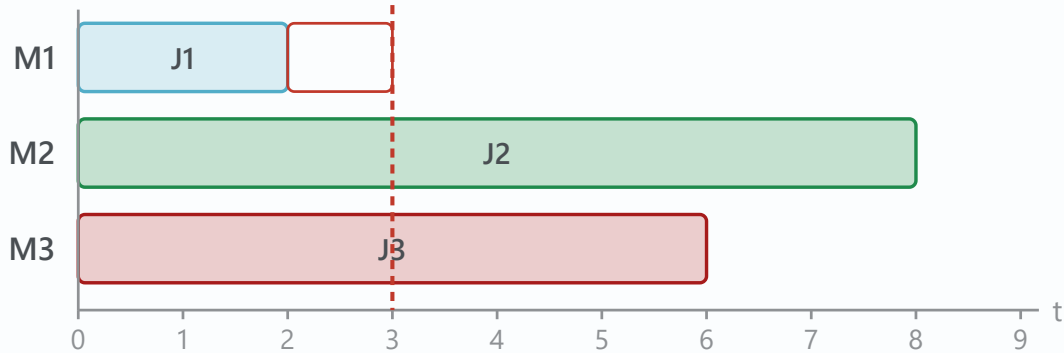
J2 (süre 8) → **M2**, $[0, 8]$ aralığında çalışır (M1 $t = 2$ anına kadar meşgul).

Makinelere İş Atama: Liste Çizelgeleme



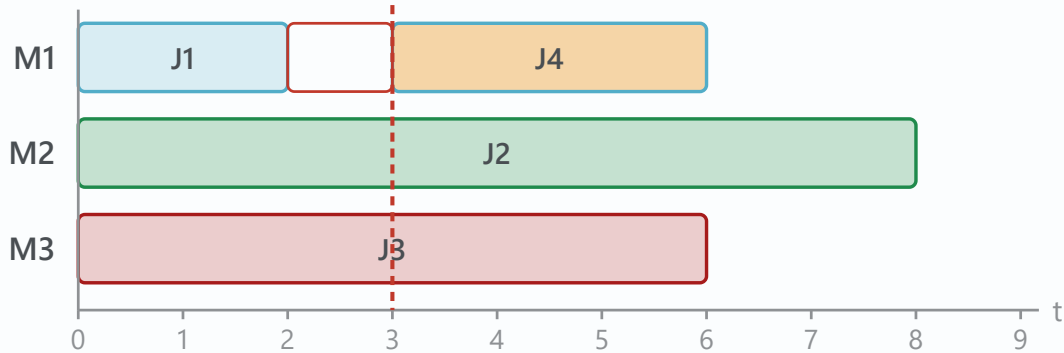
J3 (süre 6) → **M3**, [0, 6] aralığında çalışır (M1 t = 2 anına kadar meşgul; M2 t = 8 anına kadar meşgul).

Makinelere İş Atama: Liste Çizelgeleme



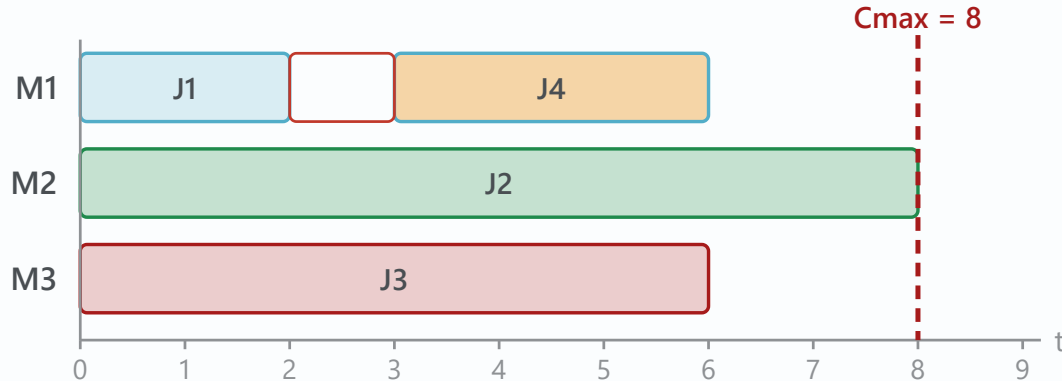
$t = 2$: M1 boşaldı ama bekleyen iş yok; makine $t = 3$ 'e kadar **boşta** kalır (taralı alan).

Makinelere İş Atama: Liste Çizelgeleme



J4 (süre 3) → **M1**, [3, 6] aralığında çalışır (M2 t = 8 anına kadar meşgul; M3 t = 6 anına kadar meşgul).

Makinelere İş Atama: Liste Çizelgeleme



Tüm işler atandı: $C_{\max} = 8$ (M2). Alt sınır: $OPT \geq \text{en uzun iş} = 8 \rightarrow$ bu çizelge **optimaldir**.
Genelde ise liste çizelgeleme yalnızca $(2 - 1/m)$ oranını garanti eder.

Liste Çizelgeleme: $(2 - 1/m)$ -Yaklaşım

İki alt sınır: $OPT \geq \sum p_j / m$ (iş yükü paylaştırılmalı) ve $OPT \geq \max p_j$.

1. En son biten iş j , s_j anında başlamış olsun.
2. s_j 'den önce **tüm makineler meşguldü** (yoksa j daha erken başlardı) $\rightarrow s_j \leq (\sum p - p_j) / m$.
3. $C_{\max} = s_j + p_j \leq \sum p / m + (1 - 1/m) \cdot p_j \leq OPT + (1 - 1/m) \cdot OPT$.

$C_{\max} \leq (2 - 1/m) \cdot OPT$; süre $O(n \log m)$ (en erken boşalan makine bir heap'te tutulur).

Sınır sıkıdır

$m = 2$, işler sırasıyla 1, 1, 2:

- Liste: $M1 = 1 + 2 = 3$, $M2 = 1 \rightarrow C_{\max} = 3$

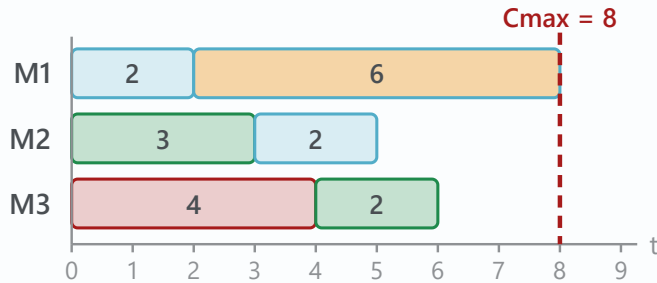
- Optimum: $\{1, 1\}$ ve $\{2\} \rightarrow 2$

Oran $3/2 = 2 - 1/2$.

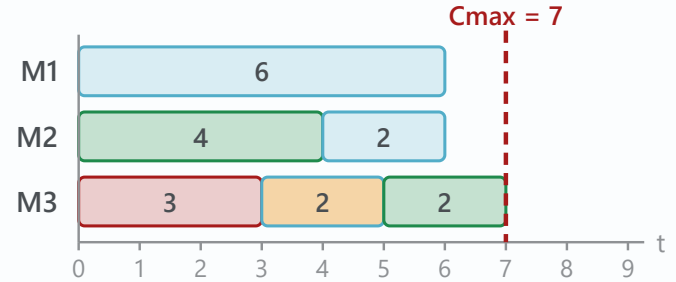
Genelde $m(m - 1)$ adet 1 birimlik iş ve sonda tek bir m birimlik iş oranı $2 - 1/m$ 'ye ulaştırır.

En Uzun İş Önce (LPT): 4/3-Yaklaşım

LPT (Longest Processing Time first): işleri **süreye göre azalan** sırala, sonra liste çizelgeleme uygula. İşler: 2, 3, 4, 6, 2, 2 ve $m = 3$.



Verilen sıra: $C_{\max} = 8$



LPT sırası 6, 4, 3, 2, 2, 2: $C_{\max} = 7 = \text{OPT} (\geq \lceil 19/3 \rceil = 7)$

- Uzun işler başa alınınca sona kalan kısa işler dengesizliği düzeltir: $C_{\max} \leq (4/3 - 1/(3m)) \cdot \text{OPT}$ (Graham, 1969). Süre $O(n \log n)$.
- Sıkı örnek ($m = 3$): 5, 5, 4, 4, 3, 3, 3 $\rightarrow$ LPT 11, optimum 9 ({5, 4}, {5, 4}, {3, 3, 3}).

Son Tarihli İş Sıralama (Job Sequencing)

- Tek makine; her iş **1 zaman birimi** sürer. İş, son tarihine kadar biterse kârı kazanılır.
- **Açgözlü kural:** işleri **kâra göre azalan** sırada ele al; her işi son tarihinden önceki **en geç boş zaman dilimine** (slot) yerleştir; boş dilim yoksa işi atla.
- En geç dilimi seçmek, erken dilimleri son tarihi daha yakın işlere bırakır.
- Bu problem için açgözlü kural **optimaldir** (matroid yapısı); süre $O(n^2)$, birleşim–bul (union–find) ile $\approx O(n \log n)$.

İş	Kâr	Son tarih
3	40	2
4	30	2
1	20	4
2	10	1

Dikkat: işleri **son tarihe** göre sıralamak yanlıştır. Örnek: (kâr 10, son tarih 1) ve (kâr 40, son tarih 1) → son tarih sırası 10 kazanır, kâr sırası 40.

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
    ► Arrays.sort(isler, (a, b) -> b.kar -  
      a.kar);  
3   int n = isler.length, toplamKar = 0;  
4   int[] dilim = new int[n];  
5   Arrays.fill(dilim, -1);          // -1: boş  
6   for (Is is : isler) {  
7       int t = Math.min(n, is.sonTarih) - 1;  
8       for (; t >= 0; t--)  
9           if (dilim[t] == -1) {  
10            dilim[t] = is.no;  
11            toplamKar += is.kar;  
12            break;  
13        }  
14    }  
15    return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

-1	-1	-1	-1
[0]	[1]	[2]	[3]
t=1	t=2	t=3	t=4

toplamKar = 0

İşler **kâra göre azalan** sıralanır: 3 (40), 4 (30), 1 (20), 2 (10). Bu örnekte tablo zaten sıralı.

Adım 1 / 13

DEĞİŞKENLER

is = - t = -

dilim = [-1, -1, -1, -1]

toplamKar = 0

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
        a.kar);  
3     ▶ int n = isler.length, toplamKar = 0;  
4     ▶ int[] dilim = new int[n];  
5     ▶ Arrays.fill(dilim, -1); // -1: boş  
6     for (Is is : isler) {  
7         int t = Math.min(n, is.sonTarih) - 1;  
8         for (; t >= 0; t--)  
9             if (dilim[t] == -1) {  
10                 dilim[t] = is.no;  
11                 toplamKar += is.kar;  
12                 break;  
13             }  
14     }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

-1	-1	-1	-1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 0

n = 4 dilim oluşturulur; hepsi -1 (boş).

dilim[t], t + 1. zaman dilimidir.

Adım 2 / 13

DEĞİŞKENLER

is = -

t = -

dilim = [-1, -1, -1, -1]

toplamKar = 0

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
        a.kar);  
3     int n = isler.length, toplamKar = 0;  
4     int[] dilim = new int[n];  
5     Arrays.fill(dilim, -1); // -1: boş  
6     for (Is is : isler) {  
7         int t = Math.min(n, is.sonTarih) - 1;  
8         for (; t >= 0; t--)  
9             if (dilim[t] == -1) {  
10                 dilim[t] = is.no;  
11                 toplamKar += is.kar;  
12                 break;  
13             }  
14     }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

-1	-1	-1	-1
[0]	[1]	[2]	[3]
t=1	t=2	t=3	t=4

toplamKar = 0

İş 3: son tarih 2 → en geç aday dilim **t =**
 $\min(4, 2) - 1 = 1$.

Adım 3 / 13

DEĞİŞKENLER

is = 3 (kâr 40, son 2) **t = 1**

dilim = [-1, -1, -1, -1]

toplamKar = 0

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
3     a.kar);  
4     int n = isler.length, toplamKar = 0;  
5     int[] dilim = new int[n];  
6     Arrays.fill(dilim, -1); // -1: boş  
7     for (Is is : isler) {  
8         int t = Math.min(n, is.sonTarih) - 1;  
9         for (; t >= 0; t--)  
10             if (dilim[t] == -1) {  
11                 dilim[t] = is.no;  
12                 toplamKar += is.kar;  
13                 break;  
14             }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

-1	3	-1	-1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 40

dilim[1] boş → iş 3 yerleştirilir, toplamKar = 40; break .

Adım 4 / 13

DEĞİŞKENLER

is = 3 (kâr 40, son 2)

t = 1

dilim = [-1, 3, -1, -1]

toplamKar = 40

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
   a.kar);  
3     int n = isler.length, toplamKar = 0;  
4     int[] dilim = new int[n];  
5     Arrays.fill(dilim, -1);    // -1: boş  
6     for (Is is : isler) {  
7         int t = Math.min(n, is.sonTarih) - 1;  
8         for (; t >= 0; t--)  
9             if (dilim[t] == -1) {  
10                 dilim[t] = is.no;  
11                 toplamKar += is.kar;  
12                 break;  
13             }  
14     }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

-1	3	-1	-1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 40

İş 4: son tarih 2 → en geç aday dilim $t = \min(4, 2) - 1 = 1$.

Adım 5 / 13

DEĞİŞKENLER

 $is = 4$ (kâr 30, son 2) $t = 1$ $dilim = [-1, 3, -1, -1]$ $toplamKar = 40$

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {
2     Arrays.sort(isler, (a, b) -> b.kar -
3         a.kar);
4     int n = isler.length, toplamKar = 0;
5     int[] dilim = new int[n];
6     Arrays.fill(dilim, -1); // -1: boş
7     for (Is is : isler) {
8         int t = Math.min(n, is.sonTarih) - 1;
9         for (; t >= 0; t--)
10             if (dilim[t] == -1) {
11                 dilim[t] = is.no;
12                 toplamKar += is.kar;
13                 break;
14             }
15     }
16     return dilim;
17 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

-1	3	-1	-1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 40

dilim[1] = 3 dolu → bir önceki dilime bak:

t = 0 .

Adım 6 / 13

DEĞİŞKENLER

is = 4 (kâr 30, son 2)

t = 0

dilim = [-1, 3, -1, -1]

toplamKar = 40

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
   a.kar);  
3     int n = isler.length, toplamKar = 0;  
4     int[] dilim = new int[n];  
5     Arrays.fill(dilim, -1);          // -1: boş  
6     for (Is is : isler) {  
7         int t = Math.min(n, is.sonTarih) - 1;  
8         for (; t >= 0; t--)  
9             if (dilim[t] == -1) {  
10                 dilim[t] = is.no;  
11                 toplamKar += is.kar;  
12                 break;  
13     }  
14 }  
15 return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	-1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 70

dilim[0] boş → iş 4 yerleştirilir, toplamKar = 70; break .

Adım 7 / 13

DEĞİŞKENLER

is = 4 (kâr 30, son 2)

t = 0

dilim = [4, 3, -1, -1]

toplamKar = 70

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
3         a.kar);  
4     int n = isler.length, toplamKar = 0;  
5     int[] dilim = new int[n];  
6     Arrays.fill(dilim, -1); // -1: boş  
7  
8     for (Is is : isler) {  
9         int t = Math.min(n, is.sonTarih) - 1;  
10        for (; t >= 0; t--)  
11            if (dilim[t] == -1) {  
12                dilim[t] = is.no;  
13                toplamKar += is.kar;  
14                break;  
15            }  
16    }  
17    return dilim;  
18 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	-1
[0]	[1]	[2]	[3]
t=1	t=2	t=3	t=4

toplamKar = 70

İş 1: son tarih 4 → en geç aday dilim $t = \min(4, 4) - 1 = 3$.

Adım 8 / 13

DEĞİŞKENLER

is = 1 (kâr 20, son 4)

t = 3

dilim = [4, 3, -1, -1]

toplamKar = 70

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
3     a.kar);  
4     int n = isler.length, toplamKar = 0;  
5     int[] dilim = new int[n];  
6     Arrays.fill(dilim, -1); // -1: boş  
7     for (Is is : isler) {  
8         int t = Math.min(n, is.sonTarih) - 1;  
9         for (; t >= 0; t--)  
10             if (dilim[t] == -1) {  
11                 dilim[t] = is.no;  
12                 toplamKar += is.kar;  
13                 break;  
14             }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 90

dilim[3] boş → iş 1 yerleştirilir, toplamKar = 90; break .

Adım 9 / 13

DEĞİŞKENLER

is = 1 (kâr 20, son 4)

t = 3

dilim = [4, 3, -1, 1]

toplamKar = 90

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
   a.kar);  
3     int n = isler.length, toplamKar = 0;  
4     int[] dilim = new int[n];  
5     Arrays.fill(dilim, -1);        // -1: boş  
6     for (Is is : isler) {  
7         int t = Math.min(n, is.sonTarih) - 1;  
8         for (; t >= 0; t--)  
9             if (dilim[t] == -1) {  
10                 dilim[t] = is.no;  
11                 toplamKar += is.kar;  
12                 break;  
13             }  
14     }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 90

İş 2: son tarih 1 → en geç aday dilim **t =**
 $\min(4, 1) - 1 = 0$.

Adım 10 / 13

DEĞİŞKENLER

is = 2 (kâr 10, son 1)

t = 0

dilim = [4, 3, -1, 1]

toplamKar = 90

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
3         a.kar);  
4     int n = isler.length, toplamKar = 0;  
5     int[] dilim = new int[n];  
6     Arrays.fill(dilim, -1); // -1: boş  
7     for (Is is : isler) {  
8         int t = Math.min(n, is.sonTarih) - 1;  
9         for (; t >= 0; t--)  
10            if (dilim[t] == -1) {  
11                dilim[t] = is.no;  
12                toplamKar += is.kar;  
13                break;  
14            }  
15     return dilim;  
16 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 90

dilim[0] = 4 dolu → bir önceki dilime bak:

t = -1 .

Adım 11 / 13

DEĞİŞKENLER

is = 2 (kâr 10, son 1)

t = -1

dilim = [4, 3, -1, 1]

toplamKar = 90

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {
2     Arrays.sort(isler, (a, b) -> b.kar -
3         a.kar);
4     int n = isler.length, toplamKar = 0;
5     int[] dilim = new int[n];
6     Arrays.fill(dilim, -1); // -1: boş
7     for (Is is : isler) {
8         int t = Math.min(n, is.sonTarih) - 1;
9         for (; t >= 0; t--)
10             if (dilim[t] == -1) {
11                 dilim[t] = is.no;
12                 toplamKar += is.kar;
13                 break;
14             }
15     }
16     return dilim;
17 }
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	1
[0] t=1	[1] t=2	[2] t=3	[3] t=4

toplamKar = 90

t = -1 : son tarihi 1 olan iş 2 için boş dilim kalmadı → iş **yapılamaz** (kâr 0).

Adım 12 / 13

DEĞİŞKENLER

is = 2 (kâr 10, son 1)

t = -1

dilim = [4, 3, -1, 1]

toplamKar = 90

Son Tarihli İş Sıralama: Kodu Adım Adım İzle

```
1 int[] isSiralama(Is[] isler) {  
2     Arrays.sort(isler, (a, b) -> b.kar -  
3     a.kar);  
4     int n = isler.length, toplamKar = 0;  
5     int[] dilim = new int[n];  
6     Arrays.fill(dilim, -1); // -1: boş  
7     for (Is is : isler) {  
8         int t = Math.min(n, is.sonTarih) - 1;  
9         for (; t >= 0; t--)  
10             if (dilim[t] == -1) {  
11                 dilim[t] = is.no;  
12                 toplamKar += is.kar;  
13                 break;  
14             }  
15     }  
16     return dilim;  
}
```

İş	Kâr	Son t.
3	40	2
4	30	2
1	20	4
2	10	1

dilim (zaman dilimi)

4	3	-1	1
[0]	[1]	[2]	[3]
t=1	t=2	t=3	t=4

toplamKar = 90

Sonuç `dilim` = [4, 3, -1, 1]: Önce iş 4, sonra iş 3, t = 4'te iş 1. Toplam kâr = 40 + 30 + 20 + 0 = **90**.

Adım 13 / 13

DEĞİŞKENLER

is = -

t = -

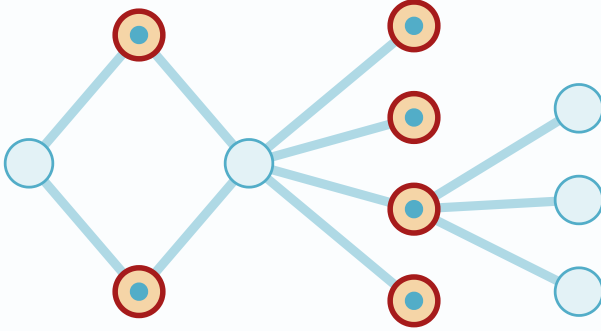
dilim = [4, 3, -1, 1]

toplamKar = 90

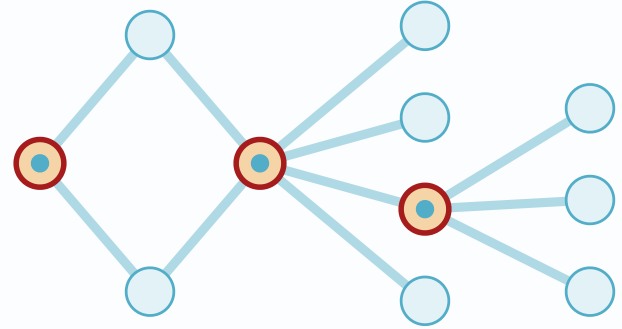
DÜĞÜM KAPSAMA

Düğüm Kapsama (Vertex Cover)

- **Kapsama problemi:** bir çizgenin **tüm kenarlarını** kapsayacak en az sayıda düğümü seçmek. Bir kenar, uçlarından **en az biri** seçilmişse kapsanır.
- **Amaç:** tüm kenarları en az düğümle kapsayan $C \subseteq V$ kümesini bulmak. Karar sürümü ("k düğümlük kapsama var mı?") **NP-tamdır**.



Geçerli bir kapsama, boyut 6 (her kenarı bir "gözcü" görüyor)

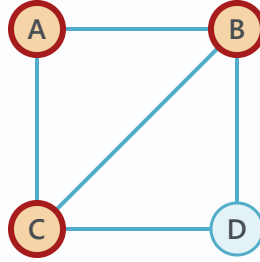


Aynı çizgede en küçük kapsama, boyut 3

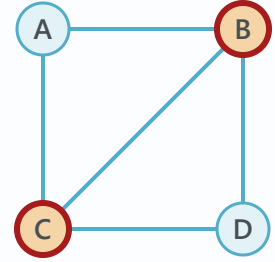
DÜĞÜM KAPSAMA

Örnek: Dört Düğümlü Çizge

- **Düğüm**ler: A, B, C, D
- **Kenar**lar: (A, B), (A, C), (B, C), (B, D), (C, D)
- {A, B, C} bir düğüm kapsama kümesidir: her kenarın en az bir ucu kümede.
- Ama **en küçük** değildir: {B, C} de beş kenarın hepsini kapsar.
- Tek düğüm yetmez (B, (A, C)'yi kapsamaz) → optimum **2**.



{A, B, C}: geçerli, boyut 3



{B, C}: en küçük, boyut 2

Düğüm Kapsama: Çözüm Yaklaşımları

Tam çözüm

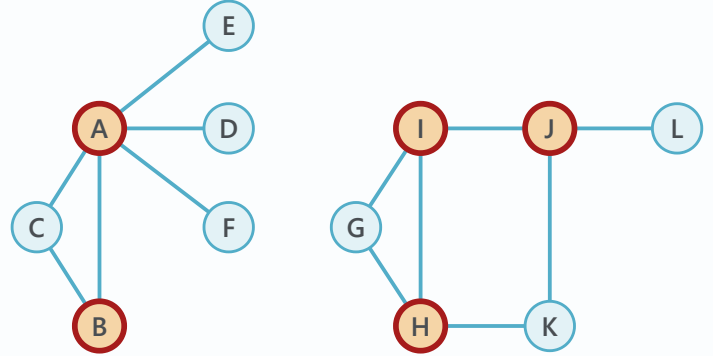
Tüm olası düğüm kümelerini kontrol etmek:
 2^n alt küme $\times$ $O(m)$ denetim.

Yaklaşık çözümler

Açgözlü algoritmalar (en yüksek dereceli düğüm), kenar seçmeli 2-yaklaşım algoritması.

Optimizasyon araçları

Doğrusal programlama (LP gevşetmesi + yuvarlama), dal sınır (branch and bound).



En küçük kapsama {A, B, H, I, J} (5 düğüm)

Düğüm Kapsama Algoritması

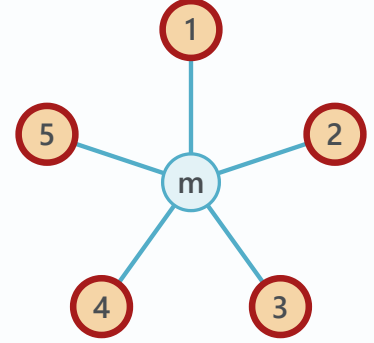
Saf fikir:

1. Başlangıçta boş bir kapsama kümesi oluştur.
2. Kapsanmamış bir kenar seç; uçlarından birini kümeye ekle.
3. Tüm kenarlar kapsanana kadar tekrarla.

Uçlardan **yalnızca birini** eklemek garanti vermez. Doğru

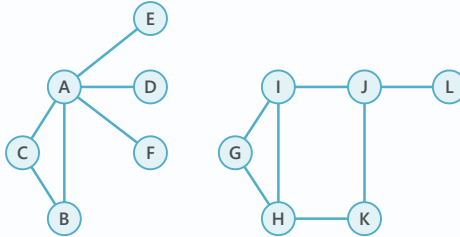
2-yaklaşım:

1. Kapsanmamış bir (u, v) kenarı seç; **u ve v'nin ikisini de** ekle.
2. u ya da v'ye değen tüm kenarları kapsanmış say; kenar kalmayana kadar tekrarla.



Her kenarda yaprak ucu seçilirse 5 düğüm; optimum $\{m\}$, 1 düğüm. n yapraklı yıldızda oran n olur.

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



Kapsama kümesi boş. Kenarlar sırayla gezilecek: (A,B), (A,C), ..., (J,L).

Adım 1 / 15

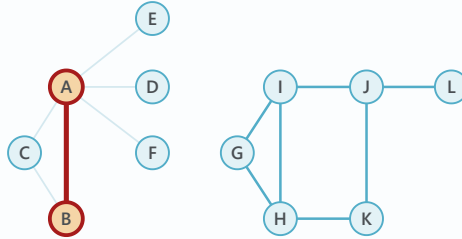
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {
2     Set<Character> kapsama = new HashSet<>();
3     for (char[] k : kenarlar) {
4         char u = k[0], v = k[1];
5         if (!kapsama.contains(u) && !kapsama.contains(v)) {
6             kapsama.add(u); kapsama.add(v); } // iki uç
7     }
8     return kapsama;
9 }
```

DEĞİŞKENLER

k = -

kapsama = {}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(A, B): iki uç da kapsamada değil → **A ve B** eklenir. Bu kenarlar bir **eşleştirme** (matching) oluşturur.

Adım 2 / 15

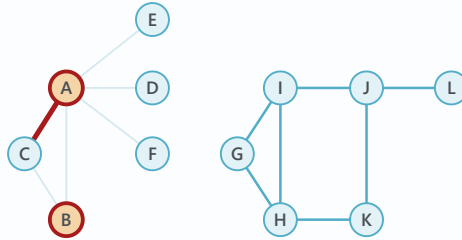
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {
2   Set<Character> kapsama = new HashSet<>();
3   for (char[] k : kenarlar) {
4     ▶ char u = k[0], v = k[1];
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {
6     ▶   kapsama.add(u); kapsama.add(v); } // iki uç
7   }
8   return kapsama;
9 }
```

DEĞİŞKENLER

k = (A,B)

kapsama = {A, B}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(A, C): A zaten kapsamada → kenar kapsanmış, atlanır.

Adım 3 / 15

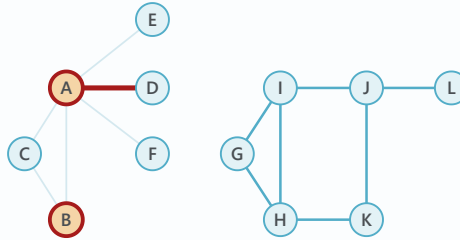
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {
2   Set<Character> kapsama = new HashSet<>();
3   for (char[] k : kenarlar) {
4     ▶ char u = k[0], v = k[1];
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {
6       kapsama.add(u); kapsama.add(v); } // iki uç
7   }
8   return kapsama;
9 }
```

DEĞİŞKENLER

k = (A, C)

kapsama = {A, B}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(A, D): A zaten kapsamada → kenar kapsanmış, atlanır.

Adım 4 / 15

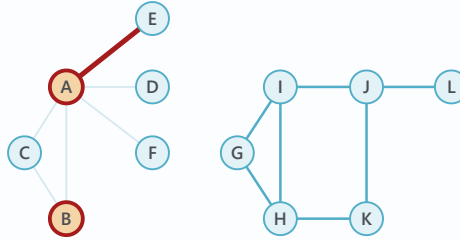
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     ▶ char u = k[0], v = k[1];  
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7   }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (A, D)

kapsama = {A, B}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(A, E): A zaten kapsamada → kenar kapsanmış, atlanır.

Adım 5 / 15

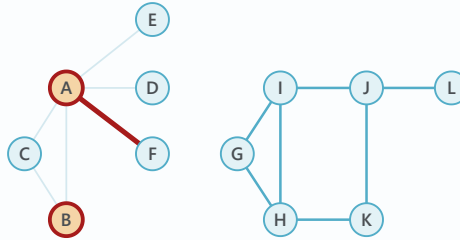
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2     Set<Character> kapsama = new HashSet<>();  
3     for (char[] k : kenarlar) {  
4         ▶ char u = k[0], v = k[1];  
5         ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6             kapsama.add(u); kapsama.add(v); } // iki uç  
7         }  
8     return kapsama;  
9 }
```

DEĞİŞKENLER

k = (A, E)

kapsama = {A, B}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(A, F): A zaten kapsamada → kenar kapsanmış, atlanır.

Adım 6 / 15

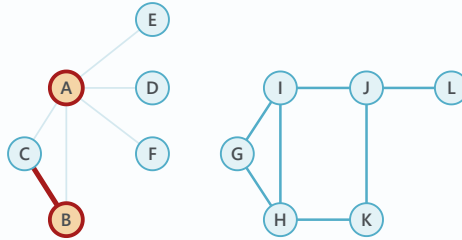
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     ▶ char u = k[0], v = k[1];  
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (A, F)

kapsama = {A, B}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(B, C): B zaten kapsamada → kenar kapsanmış, atlanır.

Adım 7 / 15

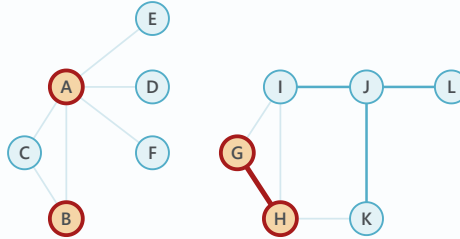
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     ▶ char u = k[0], v = k[1];  
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (B, C)

kapsama = {A, B}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(G, H): iki uç da kapsamada değil → **G ve H** eklenir. Bu kenarlar bir **eşleştirme** (matching) oluşturur.

Adım 8 / 15

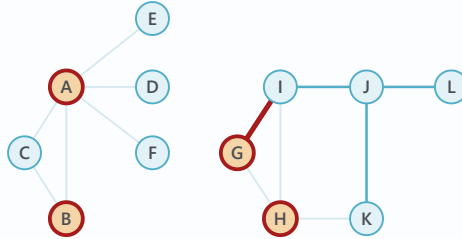
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     ▶ char u = k[0], v = k[1];  
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       ▶ kapsama.add(u); kapsama.add(v); } // iki uç  
7   }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (G,H)

kapsama = {A, B, G, H}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(G, I): G zaten kapsamada → kenar kapsanmış, atlanır.

Adım 9 / 15

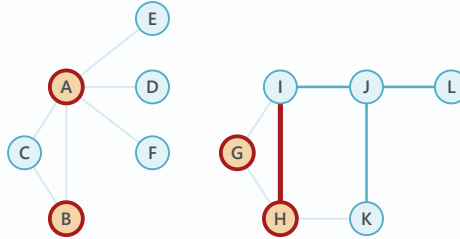
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     char u = k[0], v = k[1];  
5     if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (G, I)

kapsama = {A, B, G, H}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(H, I): H zaten kapsamada → kenar kapsanmış, atlanır.

Adım 10 / 15

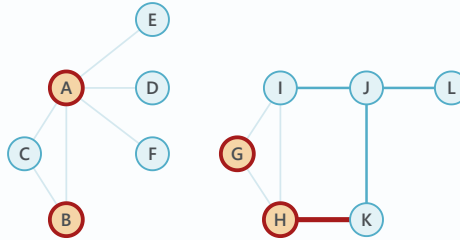
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     ▶ char u = k[0], v = k[1];  
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (H, I)

kapsama = {A, B, G, H}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(H, K): H zaten kapsamada → kenar kapsanmış, atlanır.

Adım 11 / 15

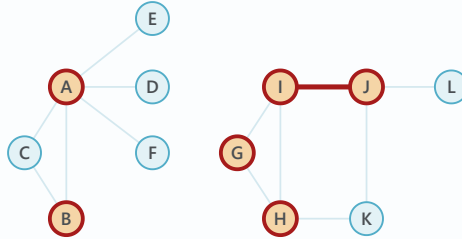
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     ▶ char u = k[0], v = k[1];  
5     ▶ if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

$k = (H, K)$

$kapsama = \{A, B, G, H\}$

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(I, J): iki uç da kapsamada değil → **I ve J** eklenir. Bu kenarlar bir **eşleştirme** (matching) oluşturur.

Adım 12 / 15

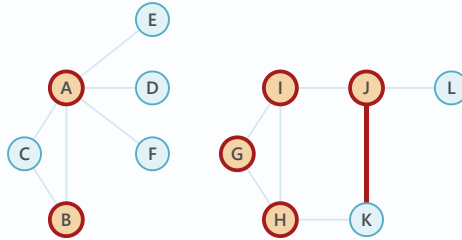
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     char u = k[0], v = k[1];  
5     if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7   }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (I, J)

kapsama = {A, B, G, H, I, J}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(J, K): J zaten kapsamada → kenar kapsanmış, atlanır.

Adım 13 / 15

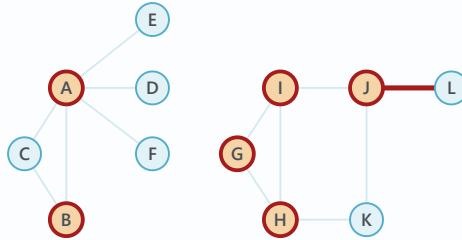
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     char u = k[0], v = k[1];  
5     if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (J, K)

kapsama = {A, B, G, H, I, J}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



(J, L): J zaten kapsamada → kenar kapsanmış, atlanır.

Adım 14 / 15

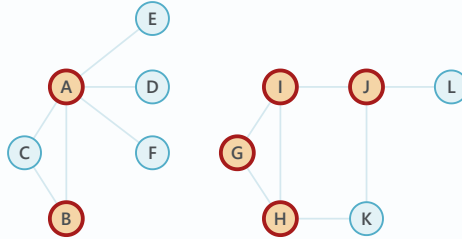
```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     char u = k[0], v = k[1];  
5     if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7     }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = (J,L)

kapsama = {A, B, G, H, I, J}

Düğüm Kapsama 2-Yaklaşımı: Kodu Adım Adım İzle



Sonuç {A, B, G, H, I, J}: 6 düğüm. Optimum 5 ({A, B, H, I, J}) → oran $6/5 = 1,2 \leq 2$.

Adım 15 / 15

```
1 Set<Character> dugumKapsama(char[][] kenarlar) {  
2   Set<Character> kapsama = new HashSet<>();  
3   for (char[] k : kenarlar) {  
4     char u = k[0], v = k[1];  
5     if (!kapsama.contains(u) && !kapsama.contains(v)) {  
6       kapsama.add(u); kapsama.add(v); } // iki uç  
7   }  
8   return kapsama;  
9 }
```

DEĞİŞKENLER

k = -

kapsama = {A, B, G, H, I, J}

Neden En Fazla $2 \times \text{Optimum}$?

1. Algoritmanın seçtiği kenarlar M kümesi olsun. Seçilen her kenarın iki ucu da eklendiği için M 'deki kenarların **ortak ucu yoktur** (M bir eşleştirmedir).
2. Her kapsama, M 'deki her kenarı kapsamak için **farklı** bir düğüm içermek zorundadır $\rightarrow$ **$\text{OPT} \geq |M|$** .
3. Algoritma tam olarak **$|C| = 2 |M|$** düğüm seçer $\rightarrow |C| \leq 2 \cdot \text{OPT}$.

Süre

$O(V + E)$: her kenar bir kez incelenir.

Açgözlü derece

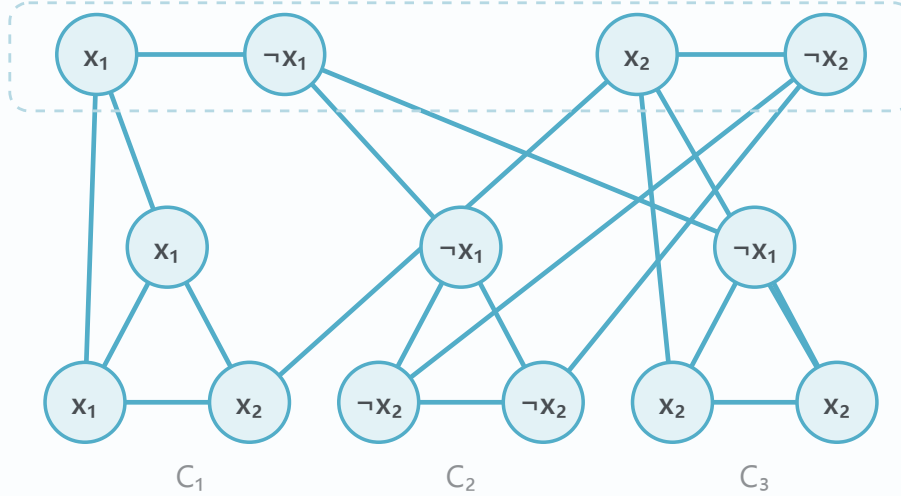
En yüksek dereceli düğümü seçmek sezgiseldir ama oranı yalnızca $O(\log n)$ 'dir.

LP yuvarlama

$x_u + x_v \geq 1$ gevşetmesini çöz, $x_v \geq 1/2$ olanları seç $\rightarrow$ yine 2-yaklaşım.

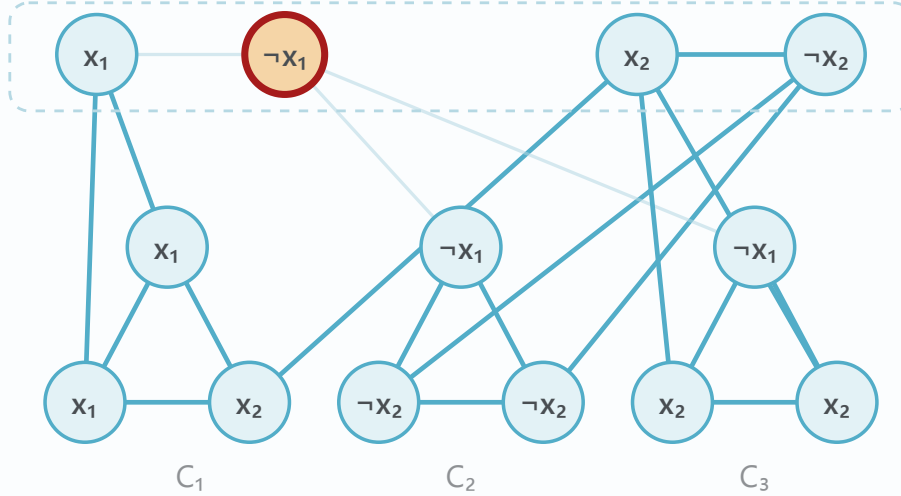
2'den belirgin biçimde iyi bir sabit oran bilinmiyor; $2 - \epsilon$ oranının (Benzersiz Oyunlar Varsayımı altında) imkânsız olduğu gösterilmiştir.

NP-Tamlık: 3-SAT'tan Düğüm Kapsamaya İndirgeme



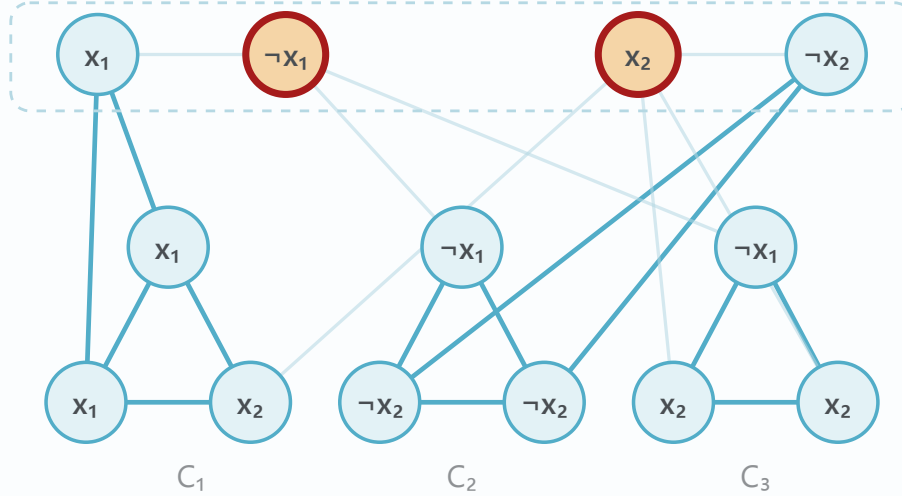
$\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$. Her değişken için bir kenar (üst kutu), her tümce için bir üçgen; tümcedeki her literal aynı adlı değişken düğümüne bağlanır. $l = 2$ değişken, $m = 3$ tümce $\rightarrow k = l + 2m = \mathbf{8}$.

NP-Tamlık: 3-SAT'tan Düğüm Kapsamaya İndirgeme



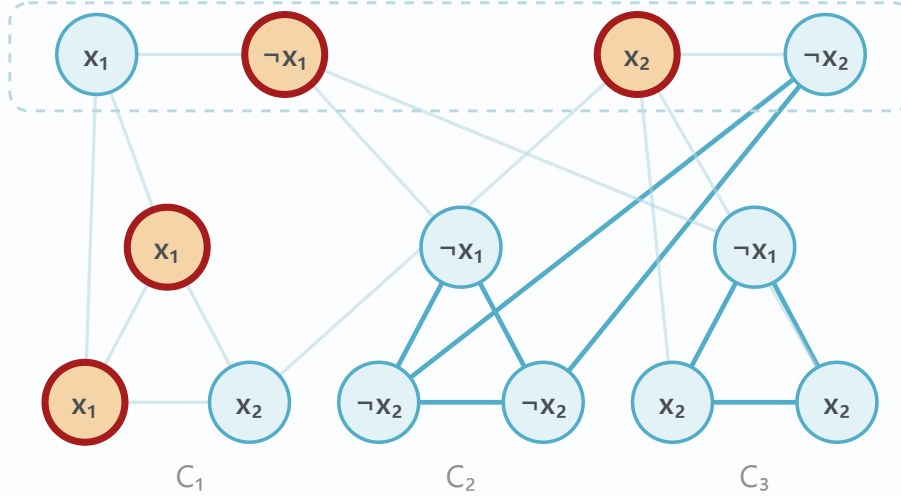
Atama $x_1 = \text{yanlış}$: değişken kenarından $\neg x_1$ seçilir. Kapsanan kenarlar soluklaşır.

NP-Tamlık: 3-SAT'tan Düğüm Kapsamaya İndirgeme



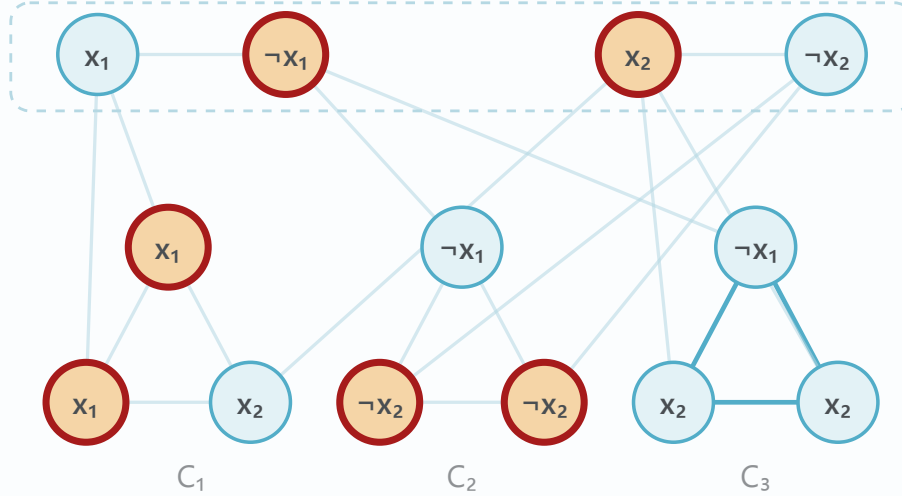
Atama $x_2 = \text{doğru}$: **x_2** seçilir. Her değişken kenarı tam olarak bir düğümlle kapsandı ($l = 2$ düğüm).

NP-Tamlık: 3-SAT'tan Düğüm Kapsamaya İndirgeme



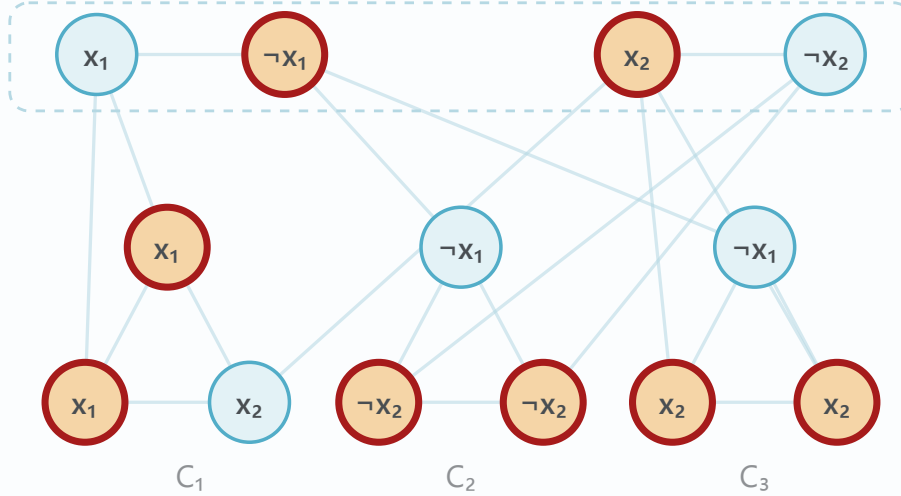
C_1 'de doğru literal x_2 (bağlantı kenarı zaten kapsanmış) dışarıda bırakılır; diğer iki düğüm (x_1, x_1) eklenir.

NP-Tamlık: 3-SAT'tan Düğüm Kapsamaya İndirgeme



C_2 'de doğru literal $\neg x_1$ dışarıda; $\neg x_2$ ve $\neg x_2$ düğümleri eklenir.

NP-Tamlık: 3-SAT'tan Düğüm Kapsamaya İndirgeme



C_3 'te doğru literal $\neg x_1$ dışarıda; x_2 ve x_2 eklenir. **8 düğüm, 20 kenarın hepsi kapsandı:** φ sağlanabilir $\Leftrightarrow k = l + 2m$ boyutlu kapsama var.

Sırt Çantası: Oran Sıralı Açgözlü Yaklaşım

Eşya	Değer	Ağırlık	Oran	Kapasite 10
6	20	7	2,86	✓ alındı (ağırlık 7)
3	8	4	2,00	✗ sığmaz
7	15	8	1,88	✗ sığmaz
2	5	3	1,67	✓ alındı (ağırlık 10)
5	10	6	1,67	✗ sığmaz
1	3	2	1,50	✗ sığmaz
4	6	5	1,20	✗ sığmaz

Açgözlü toplam değer 25; DP ile optimum da 25 (eşya 2 + 6).

- Eşyaları **değer/ağırlık** oranına göre azalan sırala; sığanı al. $O(n \log n)$.
- Kesirli sırt çantasında bu kural **optimaldir**; 0/1 sürümde ise **garanti yoktur**.
- Kötü örnek: $W = 10$, (değer 2, ağırlık 1) ve (değer 10, ağırlık 10). Açgözlü 2 alır, optimum 10: oran W kadar büyüyebilir.
- İyileştirilmiş kural**: $\max(\text{açgözlü değer, sığan en değerli tek eşya}) \rightarrow \mathbf{2\text{-yaklaşım}}$ (optimumun en az yarısı).
- Daha iyisi: değerleri ölçekleyip DP uygulayan **FPTAS**, $(1 - \epsilon) \cdot \text{OPT}$, $O(n^3/\epsilon)$.

Power Iteration (Kuvvet Yinelemesi) Algoritması

- Bir matrisin **en büyük** (mutlak değeri en büyük, baskın) **özdeğerini** λ_1 ve buna ait bir **özvektörü** v_1 yaklaşık olarak hesaplayan **yinelemeli** bir yöntemdir.
- Tüm özdeğerleri kesin olarak hesaplamak ($O(n^3)$) yerine, yalnızca baskın özdeğer ve özvektörün **yaklaşık bir tahminini** verir.
- Her adım yalnızca bir **matris-vektör çarpımıdır** ($O(n^2)$, seyrek matriste $O(\text{sıfır olmayan öge})$): **büyük boyutlu** matrislerde etkilidir.

Burada iki kavram bir arada: sonuç kesin değer yerine bir **yaklaşımdır** (tahmin) ve yinelemeler gerçek değere **yakınsar** (converge). Google'ın PageRank'i bu yöntemin en ünlü uygulamasıdır.

Power Iteration: Adımlar ve Neden Çalışır?

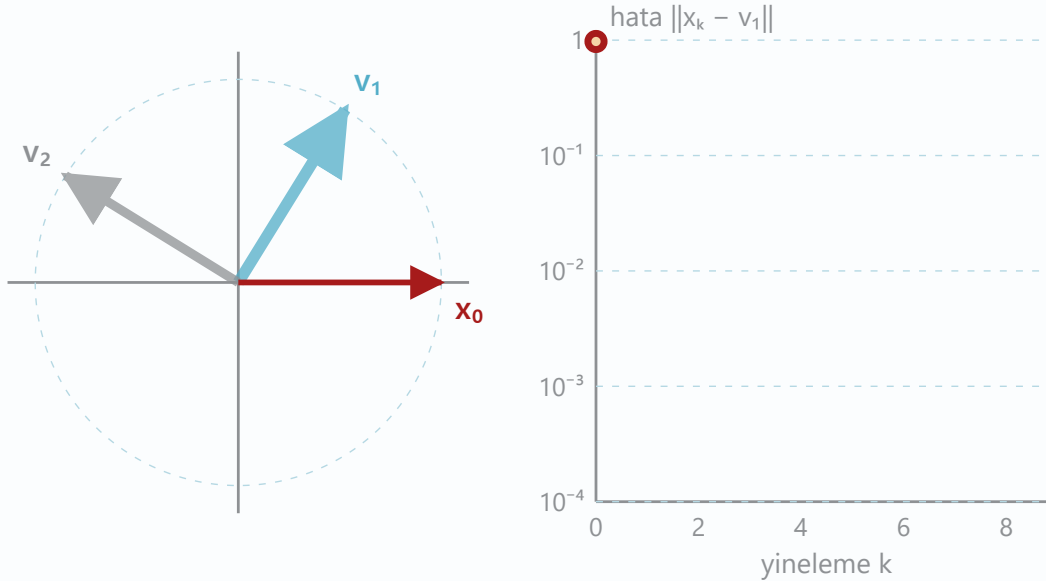
1. Bir A matrisi belirlenir ve **rastgele** (sıfır olmayan) bir başlangıç vektörü x_0 seçilir.
2. Vektör matrisle çarpılarak yeni vektör elde edilir:
 $y = A \cdot x_k$.
3. Yeni vektörün **normu** (uzunluğu) 1'e normalleştirilir: $x_{k+1} = y / \|y\|$.
4. Bu işlem istenen sayıda ya da değişim küçülene kadar tekrarlanır.
5. $x_k \rightarrow v_1$; özdeğer tahmini **Rayleigh bölümü** $\lambda \approx x_k^T A x_k$.

Neden? $x_0 = c_1 v_1 + c_2 v_2 + \dots$ ise

$$A^k x_0 = \lambda_1^k [c_1 v_1 + c_2 (\lambda_2 / \lambda_1)^k v_2 + \dots]$$

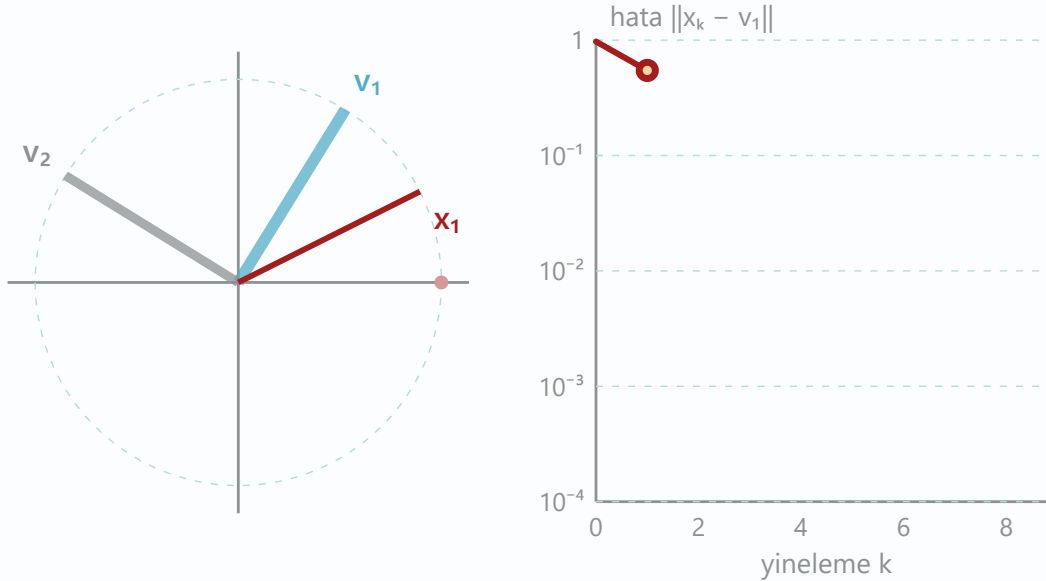
- $|\lambda_2 / \lambda_1| < 1$ olduğundan diğer bileşenler **geometrik** olarak söner.
- Hata her adımda $\approx |\lambda_2 / \lambda_1|$ katına iner: oran 1'e yakınsa yakınsama yavaştır.
- Koşul: $c_1 \neq 0$ (x_0, v_1 'e dik olmamalı) ve $|\lambda_1| > |\lambda_2|$.

Power Iteration Adım Adım



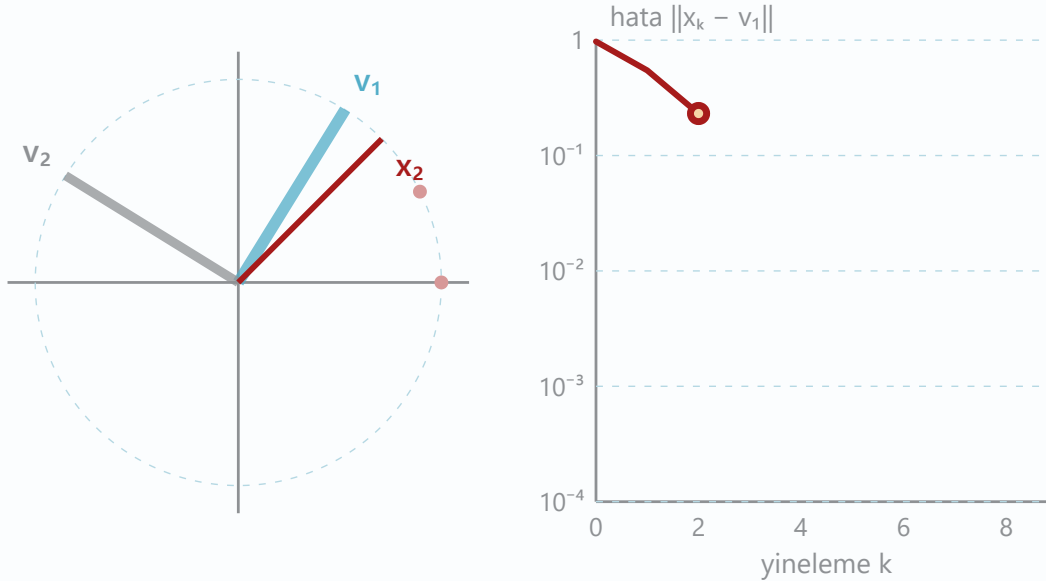
$A = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}$, özdeğerler $\lambda_1 = (5 + \sqrt{5})/2 \approx 3,618$ ve $\lambda_2 \approx 1,382$. Başlangıç $x_0 = (1, 0)$;
hata $||x_0 - v_1|| = 0,974$.

Power Iteration Adım Adım



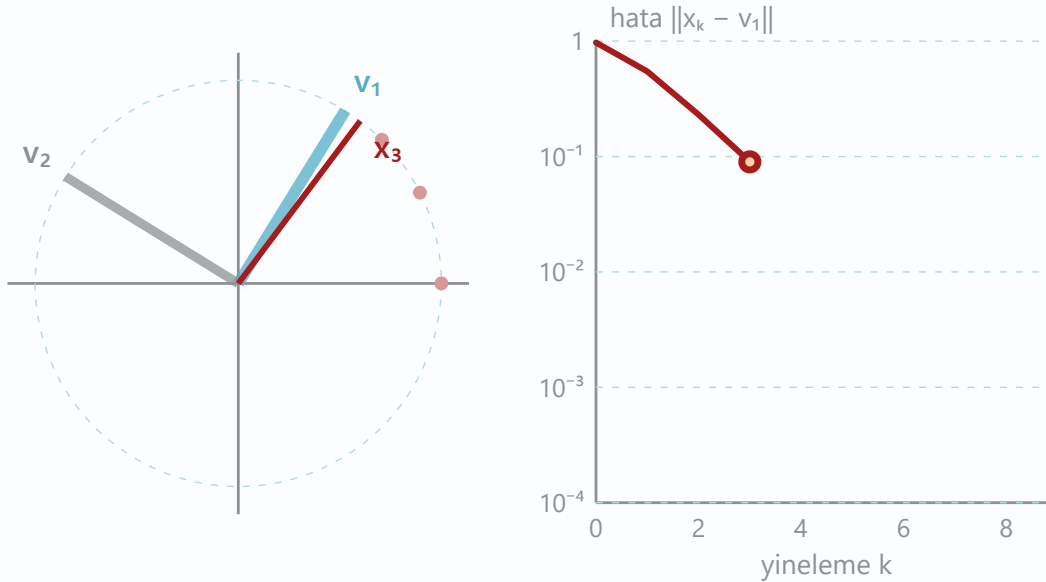
$k = 1$: $A \cdot x = (2,000, 1,000)$, normalleştir $\rightarrow x = (0,894, 0,447)$; $\lambda \approx 3,000$, hata 0,5465.

Power Iteration Adım Adım



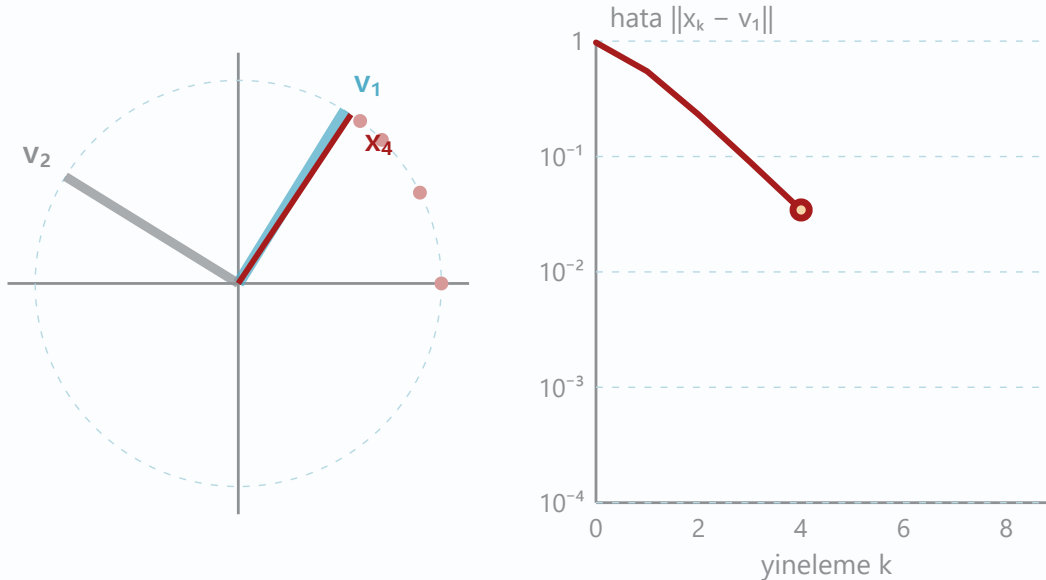
$k = 2$: $A \cdot x = (2,236, 2,236)$, normalleştir $\rightarrow x = (0,707, 0,707)$; $\lambda \approx 3,500$, hata $0,2313$.

Power Iteration Adım Adım



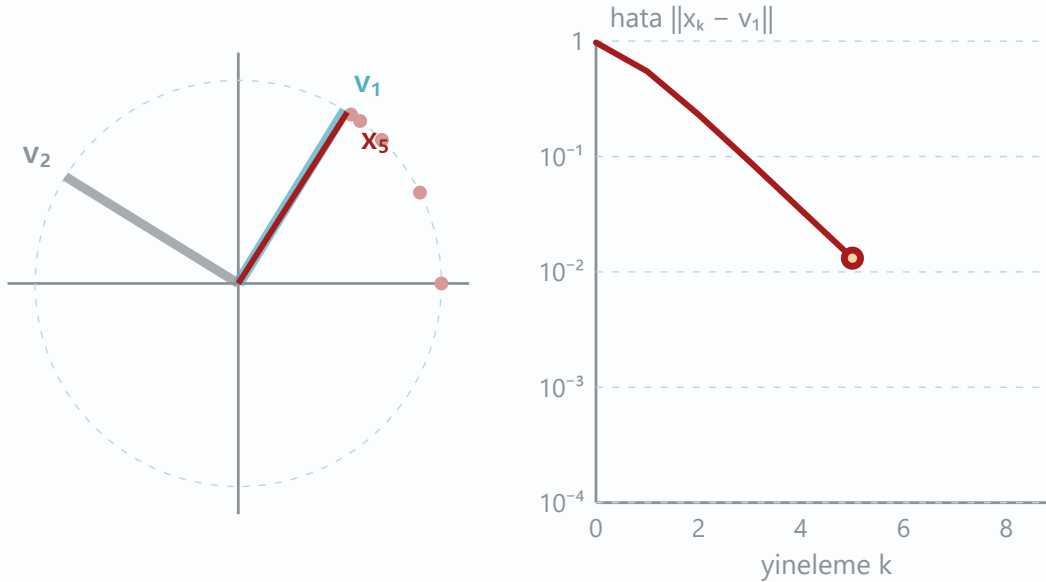
$k = 3$: $A \cdot x = (2,121, 2,828)$, normalleştir $\rightarrow x = (0,600, 0,800)$; $\lambda \approx 3,600$, hata 0,0899.

Power Iteration Adım Adım



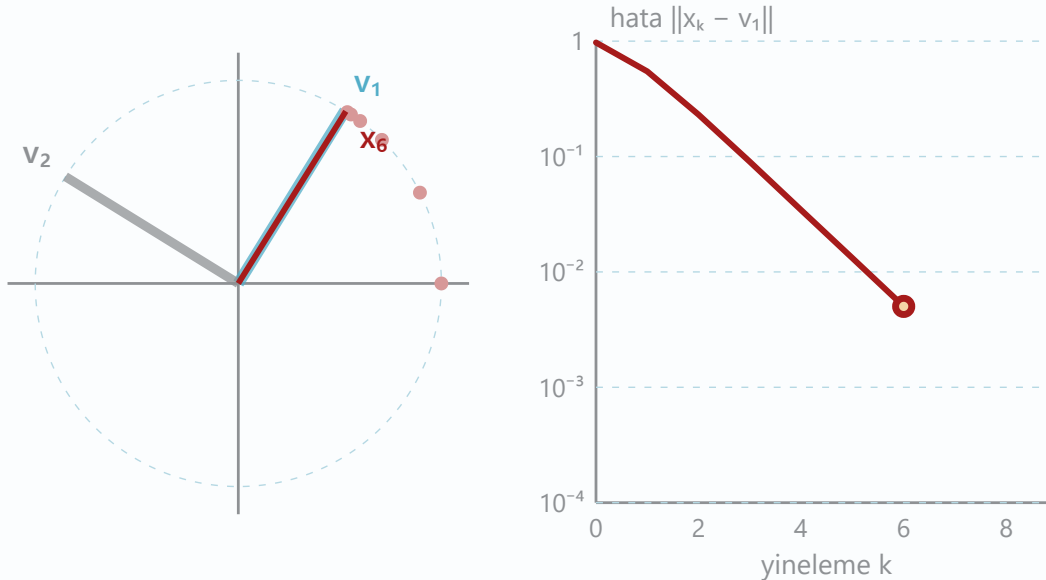
$k = 4$: $x = (0,5547, 0,8321)$; $\lambda \approx 3,61538$; hata $3,4 \cdot 10^{-2}$. Her adımda $\approx \lambda_2/\lambda_1 \approx 0,38$ katına iner.

Power Iteration Adım Adım



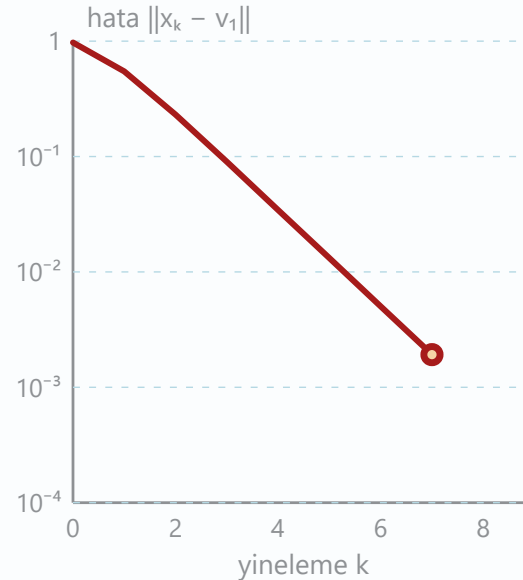
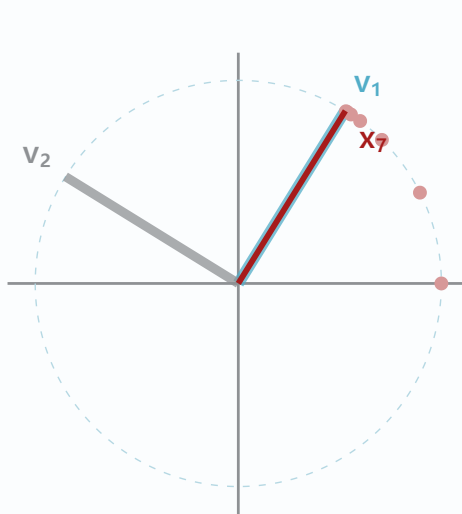
$k = 5$: $x = (0,5369, 0,8437)$; $\lambda \approx 3,61765$; hata $1,3 \cdot 10^{-2}$. Her adımda $\approx \lambda_2/\lambda_1 \approx 0,38$ katına iner.

Power Iteration Adım Adım



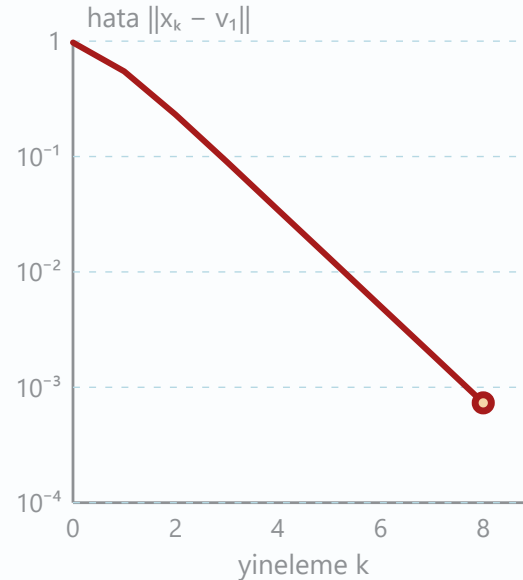
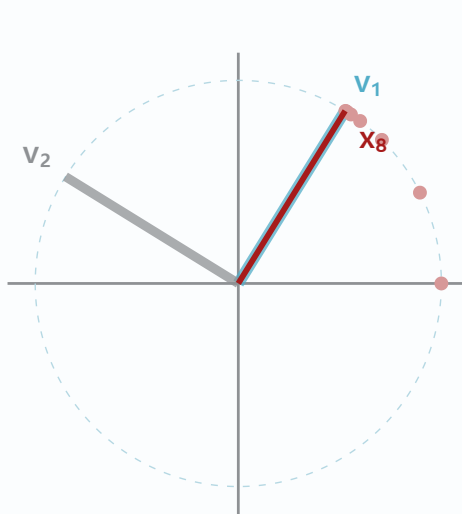
$k = 6$: $x = (0,5300, 0,8480)$; $\lambda \approx 3,61798$; hata $5,0 \cdot 10^{-3}$. Her adımda $\approx \lambda_2/\lambda_1 \approx 0,38$ katına iner.

Power Iteration Adım Adım



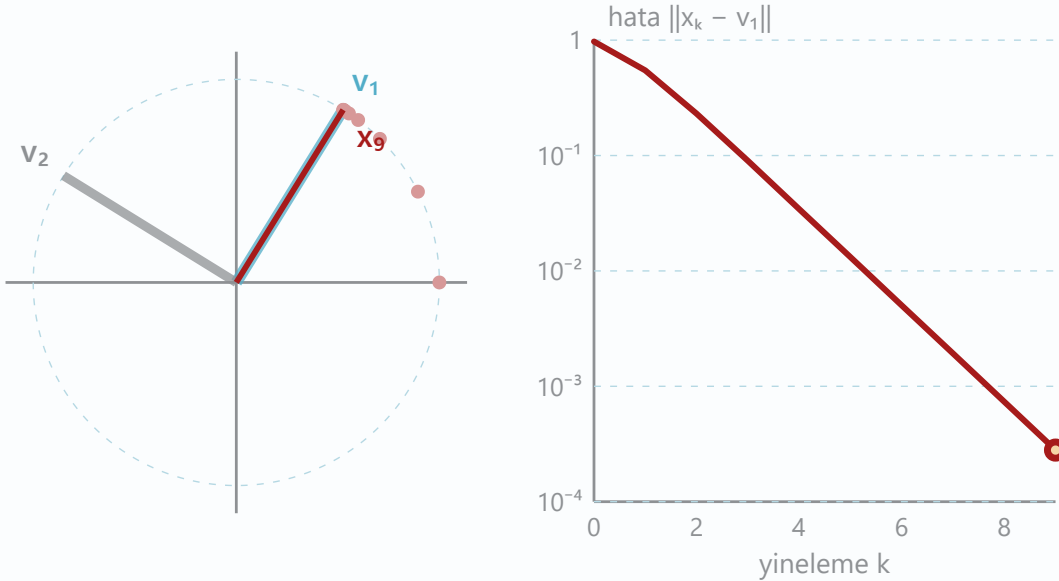
$k = 7$: $x = (0,5274, 0,8496)$; $\lambda \approx 3,61803$; hata $1,9 \cdot 10^{-3}$. Her adımda $\approx \lambda_2/\lambda_1 \approx 0,38$ katına iner.

Power Iteration Adım Adım



$k = 8$: $x = (0,5264, 0,8503)$; $\lambda \approx 3,61803$; hata $7,3 \cdot 10^{-4}$. Her adımda $\approx \lambda_2/\lambda_1 \approx 0,38$ katına iner.

Power Iteration Adım Adım



$k = 9$: $x \approx (0,5260, 0,8505) \approx v_1 = (0,5257; 0,8507)$, $\lambda \approx 3,618034$; hata $\approx 2,8 \cdot 10^{-4}$. Hata grafiği log ölçekte bir doğru: **doğrusal (geometrik) yakınsama**.

POWER ITERATION

Power Iteration: Java ve Bir 3×3 Örnek

```
double[] kuvvet(double[][] a, int adim) {  
    int n = a.length;  
    double[] x = new double[n];  
    Arrays.fill(x, 1.0);    //  $x_0 \neq 0$   
    for (int k = 0; k < adim; k++) {  
        double[] y = new double[n];  
        for (int i = 0; i < n; i++)    //  $y = A \cdot x$   
            for (int j = 0; j < n; j++)  
                y[i] += a[i][j] * x[j];  
        double norm = 0;  
        for (double yi : y) norm += yi * yi;  
        norm = Math.sqrt(norm);  
        for (int i = 0; i < n; i++)    //  $\|x\| = 1$   
            x[i] = y[i] / norm;  
    }  
    return x;  
}
```

Örnek matris (satırlar):

$(0,511 \ 0,32 \ 0,12) \cdot (0,211 \ 0,62 \ 0,12) \cdot$
 $(0,311 \ 0,12 \ 0,16); x_0 = (1, 1, 1)$

k	x_1	x_2	x_3	$\ A \cdot x\ $
1	0,6474	0,6474	0,4023	1,4690
2	0,6533	0,6533	0,3827	0,8974
3	0,6539	0,6539	0,3807	0,9005
4	0,6539	0,6539	0,3805	0,9008
5	0,6539	0,6539	0,3805	0,9008
6	0,6539	0,6539	0,3805	0,9008
8	0,6539	0,6539	0,3805	0,9008

$\|A \cdot x\| \rightarrow \lambda_1 = 0,9008; v_1 \propto (0,6539; 0,6539;$
 $0,3805). \lambda_2/\lambda_1 = 0,3/0,9008 \approx 0,33 \rightarrow$ hızlı
yakınsama.

Power Iteration: Sınırlar ve Uygulamalar

Güçlü yanları

- + Çok basit; yalnızca matris–vektör çarpımı.
- + Seyrek ve **dev** matrislerde (milyarlarca satır) uygulanabilir.
- + Bellek $O(n)$: matris değiştirilmez.

Sınırları

- Yalnızca **baskın** özdeğeri bulur.
- $|\lambda_2| \approx |\lambda_1|$ ise çok yavaş; $|\lambda_1| = |\lambda_2|$ ise yakınsamayabilir.
- x_0, v_1 'e dik seçilirse ($c_1 = 0$) kuramsal olarak başarısız olur.

- **PageRank**: web bağlantı matrisinin baskın özvektörü sayfaların önemini verir.
- **Türevler**: ters yineleme (en küçük özdeğer), kaydırmalı yineleme, Lanczos/Arnoldi; PCA'da ilk temel bileşen.

ÖZET

Özet

Problem	Algoritma	Oran	Süre
TSP (genel)	en yakın komşu (sezgisel)	garanti yok	$O(n^2)$
Metrik TSP	MST + önce-kök + kısayol / Christofides	2 / 1,5	$O(n^2) / O(n^3)$
TSP	2-opt yerel arama	garanti yok	$O(n^2) / \text{tarama}$
Makine çizelgeleme	liste çizelgeleme / LPT	$2 - 1/m / 4/3 - 1/(3m)$	$O(n \log m) / O(n \log n)$
Son tarihli iş sıralama	kâr sıralı, en geç boş dilim	1 (optimal)	$O(n^2)$
Düğüm kapsama	kenar seç, iki ucu ekle	2	$O(V + E)$
0/1 sırt çantası	max(oran sıralı, en iyi tek eşya) / FPTAS	$2 / 1 + \epsilon$	$O(n \log n) / O(n^3/\epsilon)$
Baskın özdeğer	Power Iteration	$\text{hata} \propto \lambda_2/\lambda_1 ^k$	$O(n^2) / \text{adım}$

Etkileşimli Simülatörler

NP-Tamlık ve Hesaplanamazlık

P, NP, NP-zor ve NP-tam sınıfları, polinom zamanlı indirgeme, Cook–Levin teoremi ve yaklaşım algoritmalarının neden gerektiği; kısa doğru/yanlış sınıflandırma alıştırmalarıyla.

Gezgin Satıcı Problemi

Rastgele şehirler üretin; hızlı en yakın komşu sezgiselini kaba kuvvetle bulunan optimal turla karşılaştırın ve faktöriyel büyümeyi gözlemleyin.

ÖZET

Sık Yapılan Hatalar

Sezgisel = yaklaşım algoritması sanmak

En yakın komşu ve 2-opt iyi sonuç verir ama **oran garantisi** yoktur; yaklaşım algoritması kanıtlı bir ρ ister.

Oranı ters yazmak

En küçüklemeye $C \leq \rho \cdot C^*$, en büyüklemeye $C \geq C^*/\rho$. ρ her zaman ≥ 1 'dir.

Kapsamada tek uç eklemek

Kenarın yalnızca bir ucunu eklemek yıldız çizgede n kat kötü olabilir; 2-yaklaşım **iki ucu da** ekler.

Üçgen eşitsizliğini unutmak

MST tabanlı 2-yaklaşımın kısayol adımı yalnızca **metrik** TSP'de yolu uzatmaz.

İşleri son tarihe göre sıralamak

İş sıralamada açgözlü kural **kâra göre azalan** sıradır; son tarih yalnızca dilim seçiminde kullanılır.

Normalleştirmeyi atlamak

Power Iteration'da x her adımda normalleştirilmezse sayılar taşar ($|\lambda_1| > 1$) ya da sıfıra iner.

ÖZET

Örnek Problemler

1. Bir en küçükleme probleminde algoritma 30, optimum 20 buluyor. Bu girdi için oran nedir?
2. 5 şehirlik matriste (TSP slaytı) en yakın komşuyu **2** numaralı şehirden başlatın: tur ve uzunluk?
3. 1–2–3–4–5–6 yolunda kenarlar (1,2), (2,3), ..., (5,6) sırasıyla işlenirse 2-yaklaşım kaç düğüm seçer? Optimum?
4. $m = 2$, işler 3, 3, 2, 2, 2: liste çizelgeleme ve LPT $C_{\max}$ 'ı ve optimum nedir?
5. İşler (kâr, son tarih): a(100, 2), b(19, 1), c(27, 2), d(25, 1), e(15, 3). Açgözlü sıralama ve toplam kâr?
6. $W = 10$, eşyalar (2, 1) ve (10, 10) (değer, ağırlık): oran sıralı açgözlü ve iyileştirilmiş kural ne bulur?

► Cevapları göster

ÖZET

Alıştırmalar

1. $a(0,0)$, $b(4,3)$, $c(4,0)$, $d(0,3)$ ve turda (a,b) , (c,d) kenarları var. 2-opt kazancı Δ nedir?
2. $m = 3$ için liste çizelgelemenin $5/3$ oranına ulaştığı bir iş listesi kurun.
3. K_5 tam çizgesinde 2-yaklaşım kaç düğüm seçer? Optimum kaçtır?
4. $A = [[2, 0], [0, 1]]$, $x_0 = (1, 1)$. x_k 'nin ikinci bileşeni birincisinin 10^{-3} 'ünden kaç adımda küçük olur?
5. İşleri son tarihe göre sıralayan bir açgözlü kod düşünün. Bunun yanlış sonuç verdiği bir örnek bulun.
6. 2-yaklaşımı, seçilen eşleştirmeyi de döndürecek biçimde Java ile yazın.

► Cevaplar / ipuçları

Kaynaklar

- Cormen, Leiserson, Rivest ve Stein, *Introduction to Algorithms*, 4. baskı, 2022 (Bölüm 35: Approximation Algorithms).
- V. V. Vazirani, *Approximation Algorithms*, Springer, 2001.
- D. P. Williamson ve D. B. Shmoys, *The Design of Approximation Algorithms*, Cambridge University Press, 2011.
- R. L. Graham, "Bounds on Multiprocessing Timing Anomalies", *SIAM Journal on Applied Mathematics* 17(2), 1969.
- N. Christofides, "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem", Rapor 388, Carnegie Mellon Üniversitesi, 1976.
- M. Sipser, *Introduction to the Theory of Computation*, 3. baskı, 2012 (3-SAT → düğüm kapsama indirgemesi).
- G. H. Golub ve C. F. Van Loan, *Matrix Computations*, 4. baskı, 2013 (Power Iteration).

Son

Bölüm 7: Yaklaşım Algoritmaları

Sonraki bölüm: Bölüm 8: Rastgele Algoritmalar · [Terim sözlüğü](#)