Bölüm 10: Dal Sınır Algoritmaları

Algoritmalar

Durum uzayı ağacı | Sınırlama ve budama | FIFO, LIFO, en düşük maliyet | 0/1 sırt çantası | Gezgin satıcı | Tam sayılı programlama | Maksimum klik | Paradigmaların karşılaştırması

Sercan KÜLCÜ · Giresun Üniversitesi · Bilgisayar Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Paradigma

Dal sınır yönteminin durum uzayı ağacı, dallanma, sınırlama ve budama adımlarını açıklamak; geri izlemeden farkını söylemek.

Sınır tasarımı

En büyükleme için üst, en küçükleme için alt sınır kurmak; sınırın geçerli (iyimser) olmasının neden şart olduğunu gerekçelendirmek.

Arama stratejileri

FIFO, LIFO ve en düşük maliyet (en iyi önce) stratejilerini aynı örnekte izlemek; bellek ve düğüm sayısı açısından karşılaştırmak; tasarım paradigmalarını birbirinden ayırt etmek.

Uygulamalar

0/1 sırt çantası, gezgin satıcı, tam sayılı doğrusal programlama ve maksimum klik problemlerini dal sınır ile adım adım çözmek.

GİRİŞ

En İyi Seçenekleri Ele Almak

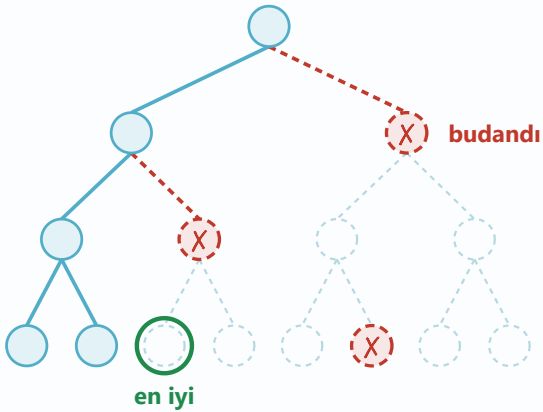
- Karmaşık problemleri çözmek, bazen bir **yol ayrımında** doğru seçeneği bulmaya benzer.
- **Dal sınır** (branch and bound) algoritmaları bu problemler için akıllı bir yaklaşım sunar: olası çözüm yollarını inceleyerek **en iyi seçeneği** bulmaya çalışır.
- Hedef: **kombinatoryal optimizasyon** problemlerinde **kesin** (optimal) çözüm. Tüm adayları denemek ise üstel ya da faktöriyel sürer.
- Fikir: bir dalın en fazla ne kadar iyi olabileceği **önceden tahmin edilir**; yeterince iyi değilse o dala hiç girilmez.

n	2^n alt küme	$(n-1)!$ tur
5	32	24
10	1 024	362 880
20	1 048 576	$\approx 1{,}2 \cdot 10^{17}$
40	$\approx 1{,}1 \cdot 10^{12}$	$\approx 2{,}0 \cdot 10^{46}$

Sırt çantası 2^n alt kümeyi, gezgin satıcı $(n-1)!$ turu dener: kaba kuvvet hızla imkânsızlaşır.

Dal Sınır Algoritmaları

- Dallanıp **budanan bir ağaç** gibi düşünülebilir.
- Önce problem daha küçük **alt problemlere** ayrılır (**dallanma**, branching).
- Her alt problem için ulaşılabilecek en iyi değer **tahmin edilir** (**sınır**, bound).
- Sınırı, o ana kadar bulunan en iyi çözümden daha iyi olmayan olasılıklar **elenir** (budama, pruning).
- İşlem kalan tüm alt problemler için tekrarlanır; gereksiz dallar budandıkça arama süresi kısalır.
- Gezgin satıcı, sırt çantası, çizelgeleme gibi birçok alanda kullanılır.



Budanan düğümün (X) alt ağacı hiç üretilmez.

GİRİŞ

Temel Kavramlar

Durum uzayı ağacı
Kök = hiçbir karar verilmemiş durum; her düzey bir karar (ör. "eşya i alınsın mı?"); yapraklar tam çözümler.

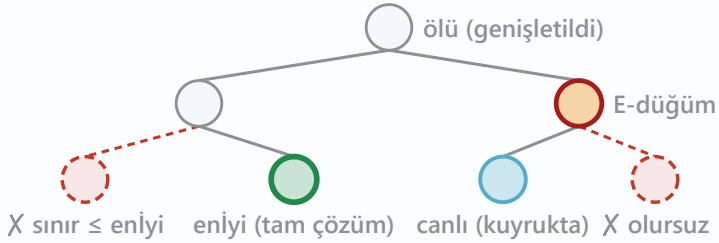
Dallanma
Bir düğümün alt problemlerini (çocuklarını) üretmek. Genişletilen düğüme **E-düğüm** (expanding node) denir.

Sınır
Düğümün altındaki **hiçbir** çözümün geçemeyeceği iyimser değer: en büyüklemede üst, en küçüklemede alt sınır.

Canlı düğüm
Üretilmiş ama henüz genişletilmemiş, umut vaat eden düğüm. Hepsi bir **kuyruk** yapısında bekler.

Ölü düğüm
Genişletilmiş, olursuz ya da sınırı yetersiz olduğu için budanmış düğüm; bir daha ele alınmaz.

enİyi (incumbent)
Şimdiye kadar bulunan en iyi tam çözümün değeri. Budama kararları bu değere göre verilir.

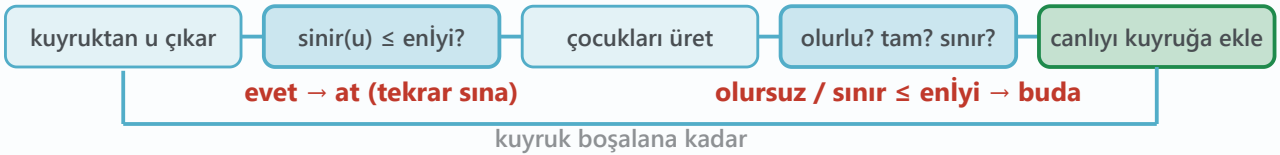


PARADİGMA

Genel Şema (En Büyükleme)

```
int enIyi = -∞;
kuyruk.ekle(kök);
while (!kuyruk.bosMu()) {
  Dugum u = kuyruk.cikar();   // strateji
  if (sinir(u) <= enIyi) continue;
  for (Dugum c : u.cocuklar()) {
    if (!olurluMu(c)) continue;
    if (tamCozumMu(c) && deger(c) > enIyi)
      enIyi = deger(c);       // yeni aday
    if (sinir(c) > enIyi)
      kuyruk.ekle(c);         // canlı düğüm
  }
}
return enIyi;
```

1. **Seç:** kuyruktan bir canlı düğüm çıkarılır. Hangisinin çıkacağını **strateji** belirler (FIFO, LIFO, en iyi sınır).
2. **Tekrar sına:** düğüm beklerken enİyi büyümüş olabilir; sınırı artık yetersizse atılır.
3. **Dallan:** çocuklar üretilir; olursuzlar elenir.
4. **Güncelle:** tam çözüm daha iyiyse enİyi yükseltilir.
5. **Sınırla:** yalnızca sınırı enİyi'yi geçebilen çocuklar kuyruğa konur.



En küçüklemede yönler ters çevrilir: alt sınır kullanılır, `alt(c) ≥ enIyi` ise budanır.

PARADİGMA

Sınır (Bound) Nasıl Seçilir?

Geçerli (iyimser) olmalı

En büyüklemede sınır(u) ≥ alt ağaçtaki her çözüm; en küçüklemede alt(u) ≤ her çözüm. Aksi hâlde optimum budanabilir ve sonuç **yanlış** olur.

Sıkı olmalı

Gerçek değere ne kadar yakınsa o kadar çok dal budanır. Sonsuz büyük bir "sınır" geçerlidir ama hiçbir şey budamaz.

Ucuz hesaplanmalı

Sınır her düğümde hesaplanır. Çok pahalı bir sınır, budadığı düğümlerden daha fazla zaman harcatabilir.

Gevşetme (relaxation) ile kurulur

Zor bir kısıt kaldırılır ve kolay problem çözülür: kesirli sırt çantası, LP gevşetmesi, en ucuz çıkış kenarları.

Başarı büyük ölçüde iyi bir sınır tahminine bağlıdır. En küçüklemede **alt sınır**, en iyi senaryoda elde edilebilecek minimum değerdir; zayıf bir alt sınır arama süresini uzatır.

PARADİGMA

Bir Düğüm Ne Zaman Budanır?

1. Olursuzluk

Düğüm bir kısıtı ihlal ediyor: ağırlık > kapasite, aynı şehre ikinci ziyaret, LP'nin çözümü yok. Altında hiçbir geçerli çözüm yoktur.

2. Sınır

sinir(u) ≤ enİyi (en büyükleme) ya da alt(u) ≥ enİyi (en küçükleme). Altındaki çözümler en iyiyi geçemez.

3. Tamlık

Düğüm tam bir çözüm (yaprak) ya da gevşetmenin çözümü zaten olurlu (ör. LP çözümü tam sayı). Değeri enİyi ile karşılaştırılır; dallanmaya gerek kalmaz.

Geri izleme yalnızca 1. kuralı (ve tamlığı) kullanır. Dal sınırı ondan ayıran şey **2. kuraldır**: düğüm geçerli olsa bile "yeterince iyi değilse" kesilir.

PARADİGMA

Arama Stratejileri: Sıradaki Canlı Düğüm Hangisi?

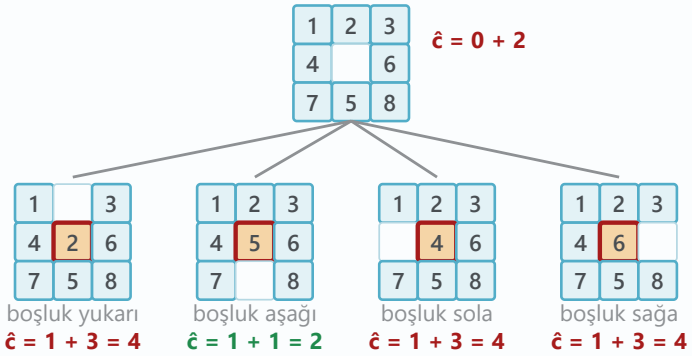
Strateji	Kuyruk	Nasıl gezer?	Artı / eksi
FIFO dal sınır	kuyruk (`ArrayDeque`)	düzey düzey, genişlik öncelikli (BFS)	sığ çözümleri erken bulur; canlı düğüm sayısı üstel büyüyebilir
LIFO dal sınır	yığın (ya da özyineleme)	derinlik öncelikli (DFS)	bellek O(derinlik); erken bir tam çözüm enİyi'yi hızla yükseltir
En düşük maliyet (LC, en iyi önce)	öncelikli kuyruk (`PriorityQueue`)	sınırı en umut verici düğüm önce	genellikle en az düğümü genişletir; kuyruk bellek tüketir, her işlem O(log k)

- Üç strateji de **aynı optimumu** bulur; değişen, üretilen düğüm sayısı ve bellektir.
- LC aramada maliyet fonksiyonu ĉ(u) = sınır değeri (ya da f(u) + ĝ(u) tahmini) olur; A* araması bu fikrin yol bulmadaki karşılığıdır.

PARADİGMA

En Düşük Maliyetli (LC) Arama: ĉ(x) = f(x) + ĝ(x)

- **f(x)**: köke uzaklık (buraya kadar ödenen maliyet); **ĝ(x)**: x'ten bir çözüme kalan maliyetin tahmini.
- Canlı düğümlerden ĉ'si en küçük olan genişletilir. ĝ = 0 seçilirse FIFO'ya, f = 0 seçilirse açgözlü derinlik aramasına yaklaşır.
- 8 bulmacasında ĝ = **yerinde olmayan taş sayısı**: her yanlış taş en az bir hamle ister, bu yüzden ĝ hiçbir zaman fazla tahmin etmez.
- Başlangıçta 5 ve 8 yanlış yerde (ĝ = 2). En küçük ĉ'li çocuk “boşluk aşağı”; onun çocuğu “boşluk sağa” hedefe ulaşır: **2 hamle**, yalnız 2 genişletme.



Hedef: 1 2 3 / 4 5 6 / 7 8 boşluk. Turuncu: hareket eden taş.

PARADİGMA

Geri İzleme ve Dal Sınır

Özellik	Geri izleme	Dal sınır
Problem türü	karar / arama: bir ya da tüm geçerli çözümler (n-vezir, Sudoku)	optimizasyon: en iyi çözüm (en büyük kazanç, en kısa tur)
Budama ölçütü	olurluluk: kısıt ihlal ediliyor mu?	olurluluk + **sınır**: enİyi'yi geçebilir mi?
Gezinme	her zaman DFS (özyineleme)	DFS, BFS ya da en iyi önce
Ek bilgi	yalnızca kısmi çözüm	kısmi çözüm + sınır fonksiyonu + enİyi
Bellek	O(derinlik)	DFS'te O(derinlik); BFS/LC'de kuyruk boyu kadar
En kötü durum	üstel	yine üstel; pratikte çok daha az düğüm

Sınır testi olmayan bir TSP ya da maksimum klik araması aslında geri izlemedir; tek bir sınır satırı eklenince dal sınıra dönüşür.

Avantajlar ve Dezavantajlar

Avantajlar

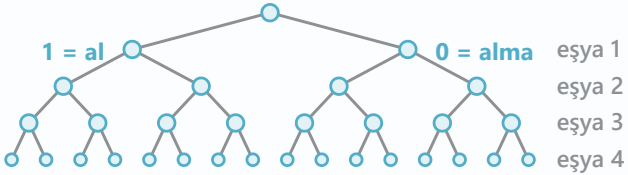
+ Karmaşık optimizasyon problemlerinde **kesin en iyi** çözümü bulur.
+ Arama alanını daraltır, gereksiz olasılıkları eler.
+ En iyi rotayı ya da seçimi bulmayı gerektiren problemlerde idealdir.
+ Çözümleri ağaç olarak temsil edilebilen çok çeşitli problemlere uygulanır.
+ Erken durdurulursa elde iyi bir çözüm ve onun sınırı (kalite garantisi) bulunur.

Dezavantajlar

– Çok fazla olasılık içeren durumlarda çözüm süresi artabilir: en kötü durumda **üstel** zaman.
– FIFO ve en iyi önce stratejilerinde bellek de üstel büyüyebilir.
– Başarı, **iyi bir sınır tahminine** bağlıdır; zayıf sınır neredeyse hiç budamaz.
– Probleme özgü sınır ve dallanma kuralı tasarlamak gerekir; uygulaması geri izlemeden karmaşıktır.

Örnek: 0/1 Sırt Çantası Problemi

- Gösterim: kazanç p_i, ağırlık w_i, kapasite W, karar $x_i \in \{0, 1\}$; $\Sigma\, w_i x_i \le W$ koşuluyla $\Sigma\, p_i x_i$ **en büyüklenir**.
- Eşyalar **p_i/w_i oranına göre azalan** sırada dizilir; sınır hesabı bu sırayı kullanır.
- Durum uzayı ağacı ikilidir: i. düzeyde sol dal "eşya i alındı" (1), sağ dal "alınmadı" (0). Tam ağaç $2^5 - 1 = 31$ düğüm.



Eşya	p_i	w_i	p_i/w_i
1	40	2	20
2	30	5	6
3	50	10	5
4	10	5	2

Kapasite W = 16.

Optimum: eşya 1 ve 3, kazanç **90** (ağırlık 12). Dal sınır bu sonuca 31 düğümün tamamını üretmeden ulaşır.

Üst Sınır: Kesirli Sırt Çantası Gevşetmesi

- Gevşetme: "eşya bölünemez" kısıtı kaldırılır. **Kesirli** sırt çantasını açgözlü yöntem en iyi şekilde çözer.
- u düğümünde (kazanç k, ağırlık a, düzey i): kalan eşyalar oran sırasıyla, sığdığı kadar **tam** eklenir; ilk sığmayan eşyanın **kesirli** parçası eklenir:

$$\text{sinir}(u) = k + \Sigma_{j=i+1}^{m-1} p_j + (W - a - \Sigma\, w_j) \cdot p_m / w_m \quad \text{(m: ilk sığmayan eşya)}$$

Neden geçerli?

Her 0/1 çözüm aynı zamanda bir kesirli çözümdür. Kesirli optimum ≥ 0/1 optimum, yani sınır asla gerçek en iyiyi küçük göstermez.

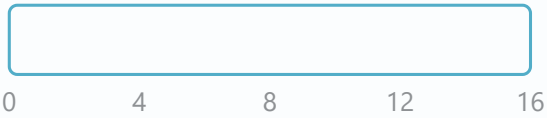
Kök için

40 + 30 = 70 (ağırlık 7); eşya 3 sığmaz, kalan 9 birim → 9 · 50/10 = 45. **sinir(kök) = 115**. a > W ise düğüm olursuzdur (sınır 0).

sinir(kök): Kodu Adım Adım İzle

```
▶ double sinir(Dugum u) {
▶   if (u.agirlik > W) return 0;   // olursuz
3   double sonuc = u.kazanc;
4   int toplamAg = u.agirlik;
5   int j = u.seviye + 1;
6   while (j < n && toplamAg + w[j] <= W) {
7     toplamAg += w[j];
8     sonuc += p[j];
9     j++;
10  }
11  if (j < n)                // kesirli parça
12    sonuc += (W - toplamAg) * (double) p[j] / w[j];
13  return sonuc;
14 }
```

kapasite W = 16



henüz eşya yok

Kök düğüm: düzey −1, kazanç 0, ağırlık 0. 0 ≤ 16 olduğundan düğüm **olurlu**.

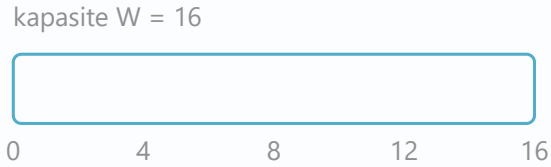
Adım 1 / 9

DEĞİŞKENLER

`u = A (kök)` `j = –` `toplamAg = –` `sonuc = –`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
▶    double sonuc = u.kazanc;
▶    int toplamAg = u.agirlik;
▶    int j = u.seviye + 1;
6    while (j < n && toplamAg + w[j] <= W) {
7      toplamAg += w[j];
8      sonuc += p[j];
9      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```



j = 0: sırada eşya 1

Sınır mevcut kazançtan başlar; `j` bir sonraki eşyayı gösterir (indis 0 = eşya 1).

Adım 2 / 9

DEĞİŞKENLER

`u = A (kök)` `j = 0` `toplamAg = 0` `sonuc = 0`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
6    while (j < n && toplamAg + w[j] <= W) {
7      toplamAg += w[j];
8      sonuc += p[j];
9      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```



0 + 2 ≤ 16 ?

0 + w[0] = 2 ≤ 16 → eşya 1 **tamamen** sığar.

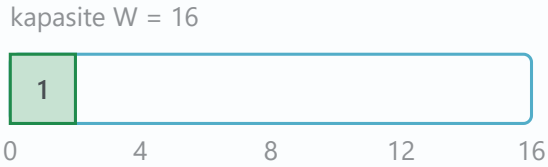
Adım 3 / 9

DEĞİŞKENLER

`u = A (kök)` `j = 0` `toplamAg = 0` `sonuc = 0`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
6    while (j < n && toplamAg + w[j] <= W) {
▶      toplamAg += w[j];
▶      sonuc += p[j];
▶      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```



eşya 1 eklendi

toplamAg = 2, sonuc = 40, j = 1.

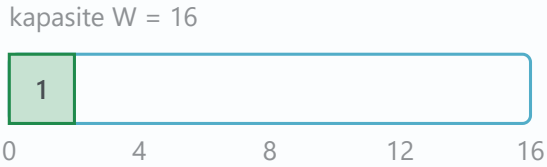
Adım 4 / 9

DEĞİŞKENLER

`u = A (kök)` `j = 1` `toplamAg = 2` `sonuc = 40`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
▶6   while (j < n && toplamAg + w[j] <= W) {
7      toplamAg += w[j];
8      sonuc += p[j];
9      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```



2 + 5 ≤ 16 ?

2 + w[1] = 7 ≤ 16 → eşya 2 de tamamen sığar.

Adım 5 / 9

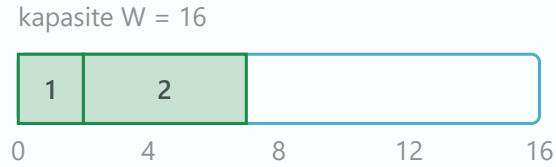
DEĞİŞKENLER

`u = A (kök)` `j = 1`

`toplamAg = 2` `sonuc = 40`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
6    while (j < n && toplamAg + w[j] <= W) {
▶      toplamAg += w[j];
▶      sonuc += p[j];
▶      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```



eşya 2 eklendi

toplamAg = 7, sonuc = 70, j = 2.

Adım 6 / 9

DEĞİŞKENLER

u = A (kök) | j = 2 | toplamAg = 7 | sonuc = 70

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
▶6   while (j < n && toplamAg + w[j] <= W) {
7      toplamAg += w[j];
8      sonuc += p[j];
9      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```

kapasite W = 16



7 + 10 = 17 > 16

7 + w[2] = 17 > 16 → eşya 3 tamamen sığmaz, döngü biter.

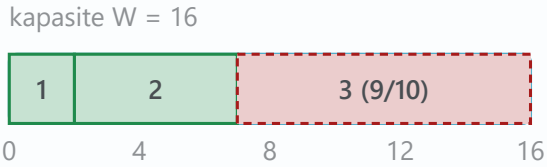
Adım 7 / 9

DEĞİŞKENLER

`u = A (kök)` `j = 2` `toplamAg = 7` `sonuc = 70`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
6    while (j < n && toplamAg + w[j] <= W) {
7      toplamAg += w[j];
8      sonuc += p[j];
9      j++;
10   }
▶    if (j < n)                  // kesirli parça
▶      sonuc += (W - toplamAg) * (double) p[j] / w[j];
13   return sonuc;
14 }
```



kalan 9 birim: eşya 3'ün 9/10'u

Kalan kapasite 9: eşya 3'ün **9/10'u** → 9 · 50/10 = 45 eklenir, sonuc = 115.

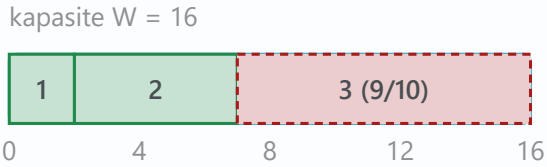
Adım 8 / 9

DEĞİŞKENLER

`u = A (kök)` `j = 2` `toplamAg = 7` `sonuc = 115`

sinir(kök): Kodu Adım Adım İzle

```
1  double sinir(Dugum u) {
2    if (u.agirlik > W) return 0;   // olursuz
3    double sonuc = u.kazanc;
4    int toplamAg = u.agirlik;
5    int j = u.seviye + 1;
6    while (j < n && toplamAg + w[j] <= W) {
7      toplamAg += w[j];
8      sonuc += p[j];
9      j++;
10   }
11   if (j < n)                 // kesirli parça
12     sonuc += (W - toplamAg) * (double) p[j] / w[j];
▶    return sonuc;
14 }
```



sinir(A) = 115

sinir(A) = 115: bu düğümün altındaki hiçbir çözüm 115'ten fazla kazandıramaz.

Adım 9 / 9

DEĞİŞKENLER

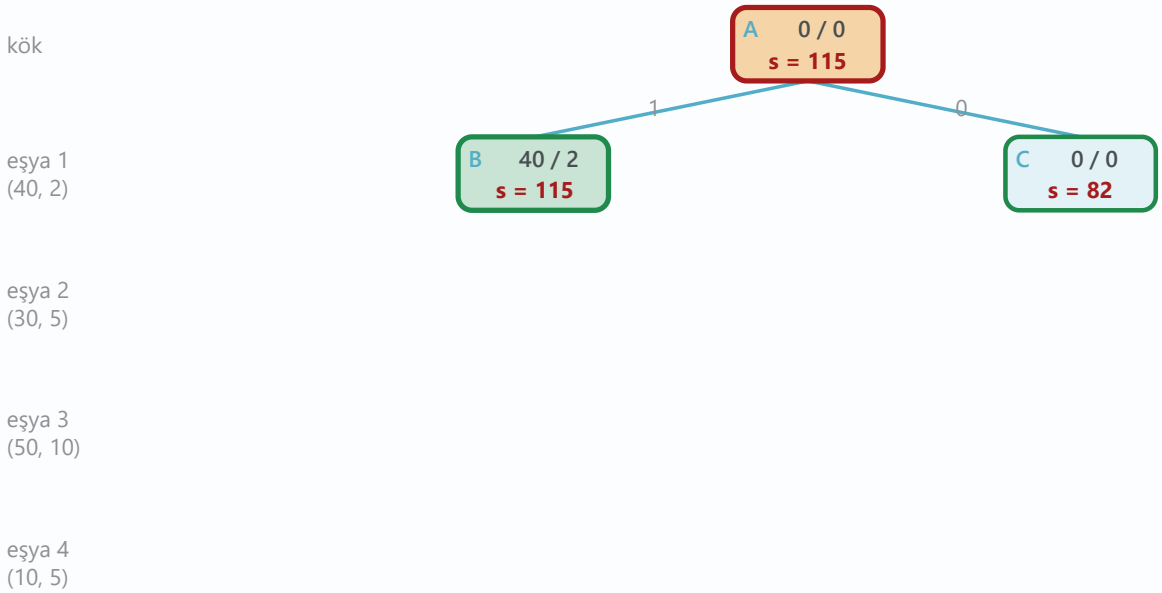
`u = A (kök)` `j = 2` `toplamAg = 7` `sonuc = 115`

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



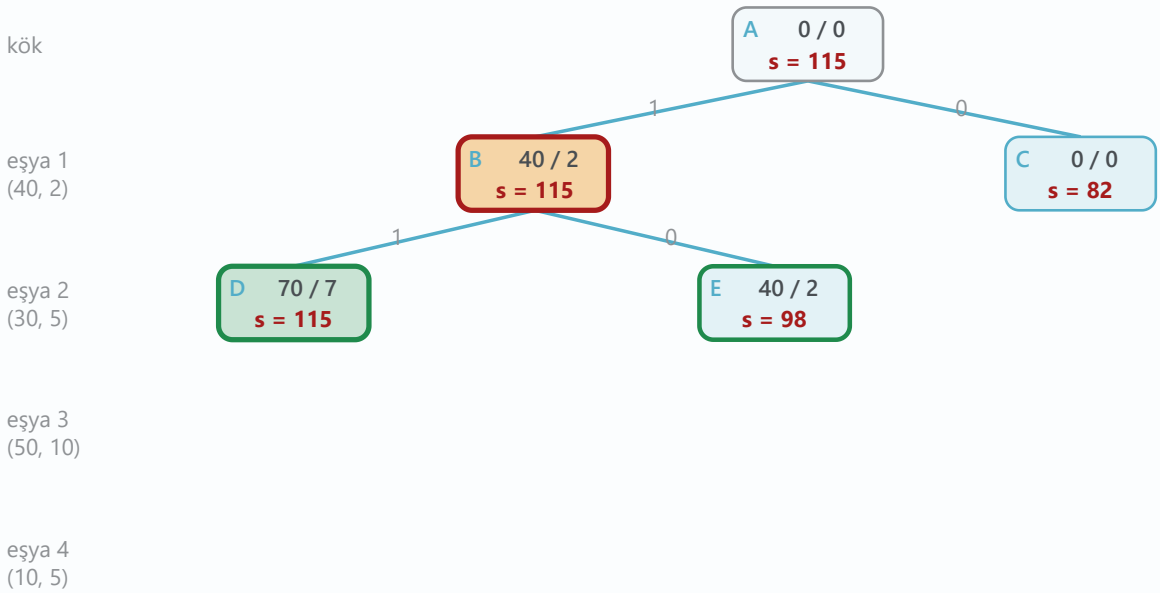
Kök **A**: hiçbir eşya seçilmedi (kazanç 0, ağırlık 0). Kesirli sınır **s = 115**; enİyi = 0. Kuyruk: A (115).

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



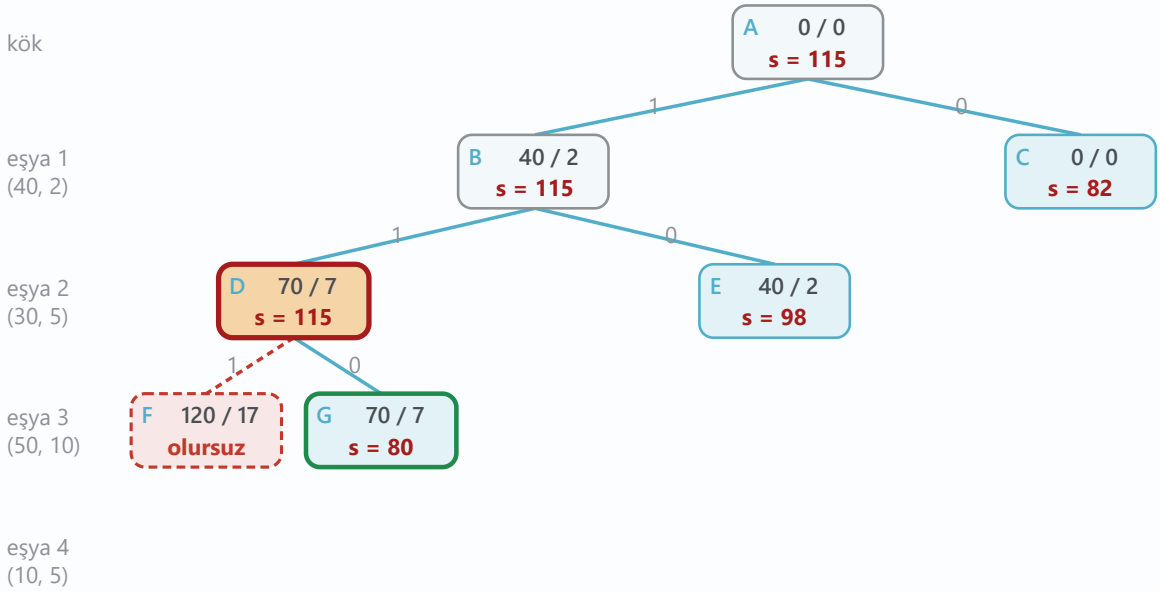
A çıkarıldı (en büyük sınır) ve dallandı. **B** (eşya 1 alındı): **enİyi = 40**, s = 115 → kuyruğa. **C** (eşya 1 yok): s = 82 → kuyruğa. Kuyruk: B (115), C (82).

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



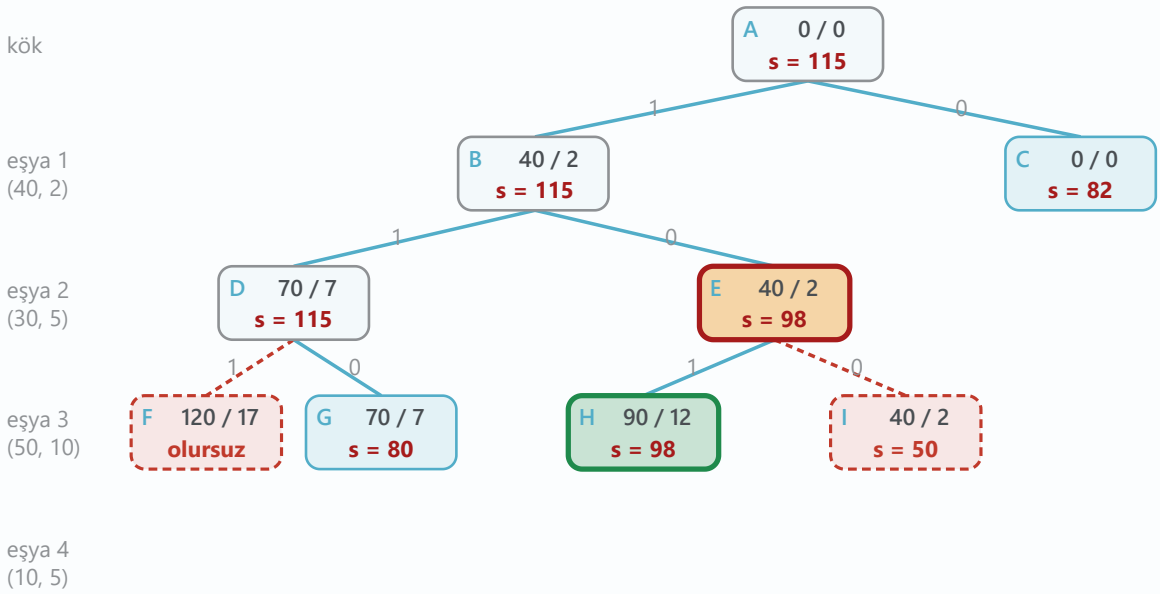
B çıkarıldı (en büyük sınır) ve dallandı. **D** (eşya 2 alındı): **enİyi = 70**, s = 115 → kuyruğa. **E** (eşya 2 yok): s = 98 → kuyruğa. Kuyruk: D (115), E (98), C (82).

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



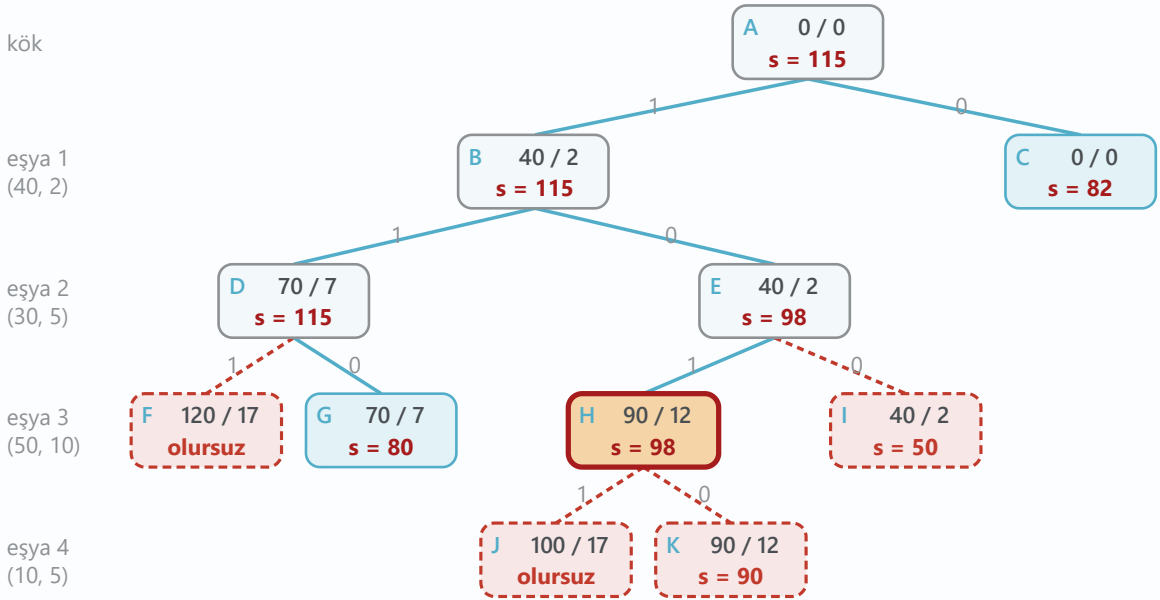
D çıkarıldı (en büyük sınır) ve dallandı. **F** (eşya 3 alındı): ağırlık 17 > 16 → **olursuz**. **G** (eşya 3 yok): s = 80 → kuyruğa. Kuyruk: E (98), C (82), G (80).

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



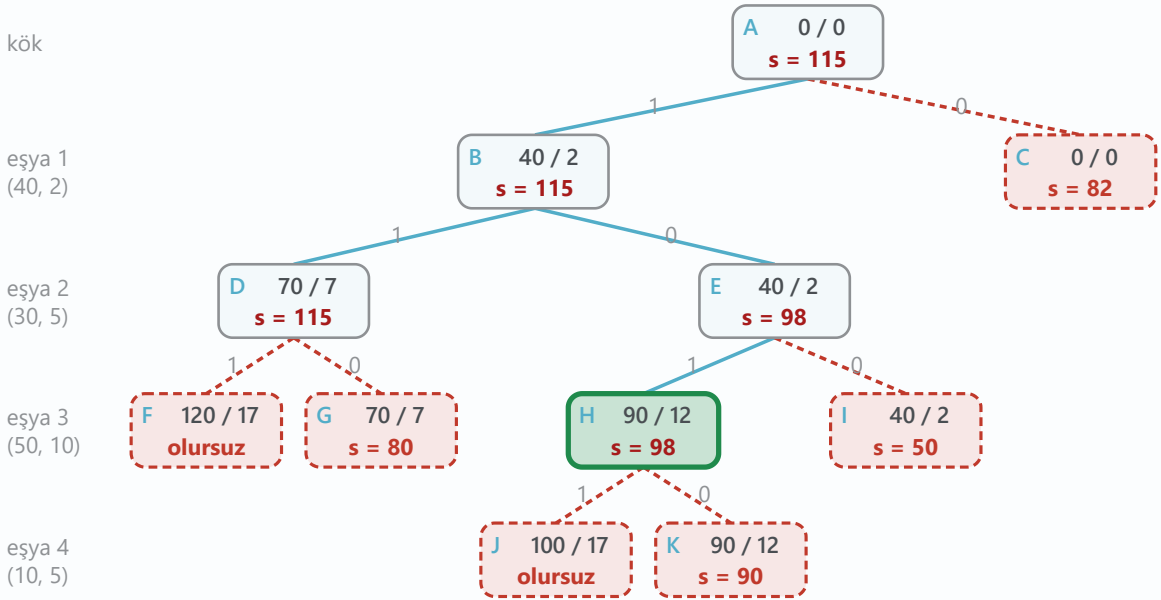
E çıkarıldı (en büyük sınır) ve dallandı. **H** (eşya 3 alındı): **enİyi = 90**, s = 98 → kuyruğa. **I** (eşya 3 yok): s = 50 ≤ 90 → budandı. Kuyruk: H (98), C (82), G (80).

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



H çıkarıldı (en büyük sınır) ve dallandı. **J** (eşya 4 alındı): ağırlık 17 > 16 → **olursuz**. **K** (eşya 4 yok): s = 90 ≤ 90 → budandı. Kuyruk: C (82), G (80).

En İyi Önce (LC) Dal Sınır: Durum Uzayı Ağacı



C çıkarıldı: s = 82 ≤ enİyi = 90 → budandı. **G** çıkarıldı: s = 80 ≤ enİyi = 90 → budandı. Kuyruk boş: **en iyi kazanç 90** (H: eşya 1 ve 3). Tam ağacın 31 düğümünden yalnızca 11'i üretildi.

En İyi Önce Dal Sınır: Java

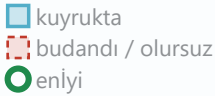
```java
int dalSinir() {
  PriorityQueue<Dugum> kuyruk = new PriorityQueue<>(
      (a, b) -> Double.compare(b.sinir, a.sinir));
  Dugum kok = new Dugum(-1, 0, 0);
  kok.sinir = sinir(kok);
  kuyruk.add(kok);  int enIyi = 0;
  while (!kuyruk.isEmpty()) {
    Dugum u = kuyruk.poll();           // en büyük sınır
    if (u.sinir <= enIyi || u.seviye == n - 1) continue;
    int i = u.seviye + 1;
    // sol çocuk: i. eşyayı al
    Dugum al = new Dugum(i, u.kazanc + p[i],
                         u.agirlik + w[i]);
    if (al.agirlik <= W && al.kazanc > enIyi)
      enIyi = al.kazanc;
    al.sinir = sinir(al);
    if (al.sinir > enIyi) kuyruk.add(al);
    // sağ çocuk: i. eşyayı alma
    Dugum alma = new Dugum(i, u.kazanc, u.agirlik);
    alma.sinir = sinir(alma);
    if (alma.sinir > enIyi) kuyruk.add(alma);
  }
  return enIyi;
}
```

- `Dugum` : düzey, kazanç, ağırlık, sınır (seçilen eşyalar için bir `boolean[]` ya da bit maskesi eklenebilir).
- Karşılaştırıcı ters: **en büyük sınır** önce çıkar.
- `poll` sonrası test, kuyrukta beklerken **eskiyen** düğümleri atar (örnekte C ve G).
- FIFO için `ArrayDeque` + `add()` / `poll()`, LIFO için `push()` / `pop()` yeterlidir; kodun geri kalanı aynıdır.
- En kötü durum $O(2^n)$ düğüm, düğüm başına O(n) sınır hesabı.

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
▶    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
▶    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
5      Dugum u = kuyruk.poll();          // ilk giren
6      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
7      for (int al = 1; al >= 0; al--) { // önce al
8        Dugum c = cocuk(u, al);         // sinir(c) dahil
9        if (c.agirlik <= W && c.kazanc > enIyi)
10         enIyi = c.kazanc;               // yeni aday
11       if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```





Aynı sırt çantası örneği (W = 16), bu kez **FIFO** kuyrukla. Kök A: kazanç 0, sınır 115; kuyrukta yalnız A.

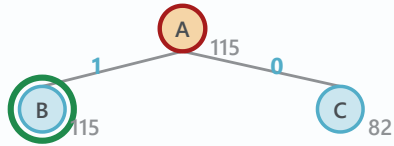
Adım 1 / 9

DEĞİŞKENLER

`u = –` `enIyi = 0` `kuyruk = [A]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
▶      for (int al = 1; al >= 0; al--) { // önce al
▶        Dugum c = cocuk(u, al);         // sinir(c) dahil
▶        if (c.agirlik <= W && c.kazanc > enIyi)
▶          enIyi = c.kazanc;               // yeni aday
▶        if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



kuyrukta
budandı / olursuz
enİyi

A kuyruğun başından çıkar ve dallanır: B (al: s = 115 → kuyruğa, **enİyi = 40**); C (alma: s = 82 → kuyruğa).

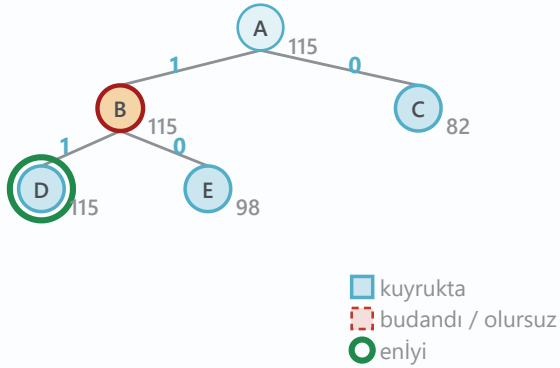
Adım 2 / 9

DEĞİŞKENLER

`u = A` `enIyi = 40` `kuyruk = [B, C]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
▶      for (int al = 1; al >= 0; al--) { // önce al
▶        Dugum c = cocuk(u, al);         // sinir(c) dahil
▶        if (c.agirlik <= W && c.kazanc > enIyi)
▶          enIyi = c.kazanc;               // yeni aday
▶        if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



B kuyruğun başından çıkar ve dallanır: D (al: s = 115 → kuyruğa, **enİyi = 70**); E (alma: s = 98 → kuyruğa).

Adım 3 / 9

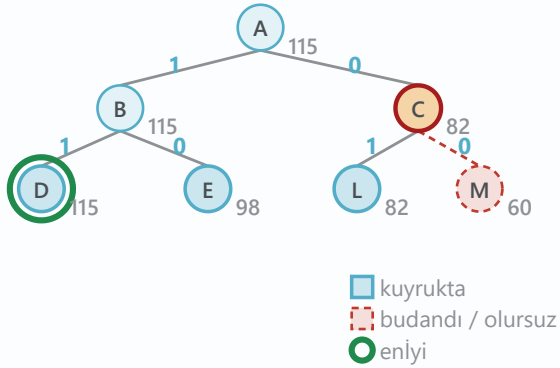
DEĞİŞKENLER

`u = B` `enIyi = 70`

`kuyruk = [C, D, E]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
▶      for (int al = 1; al >= 0; al--) { // önce al
▶        Dugum c = cocuk(u, al);         // sinir(c) dahil
▶        if (c.agirlik <= W && c.kazanc > enIyi)
▶          enIyi = c.kazanc;               // yeni aday
▶        if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



C kuyruğun başından çıkar ve dallanır: L (al: s = 82 → kuyruğa); M (alma: s = 60 ≤ 70, budandı).

Adım 4 / 9

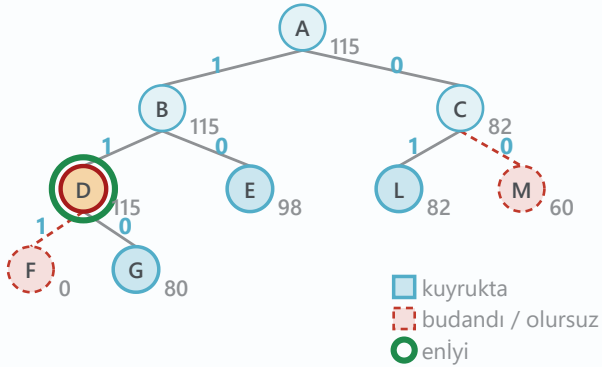
DEĞİŞKENLER

`u = C` `enIyi = 70`

`kuyruk = [D, E, L]`

FIFO Dal Sınır: Kod İzleme

```
 1 int fifoDalSinir() {
 2   ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
 3   kuyruk.add(kok);  int enIyi = 0;
 4   while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
▶      for (int al = 1; al >= 0; al--) { // önce al
▶        Dugum c = cocuk(u, al);         // sinir(c) dahil
▶        if (c.agirlik <= W && c.kazanc > enIyi)
▶          enIyi = c.kazanc;               // yeni aday
▶        if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



D kuyruğun başından çıkar ve dallanır: F (al: ağırlık 17 > 16, olursuz); G (alma: s = 80 → kuyruğa).

Adım 5 / 9

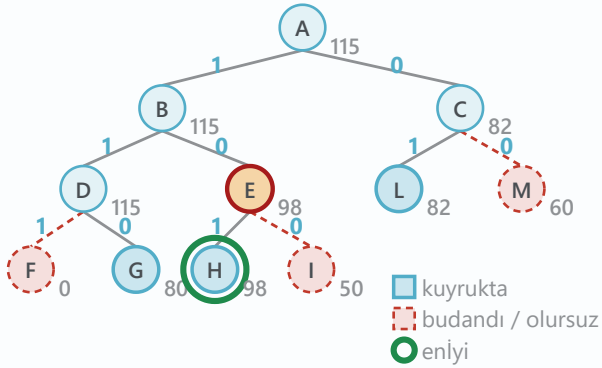
DEĞİŞKENLER

`u = D` `enIyi = 70`

`kuyruk = [E, L, G]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
▶      for (int al = 1; al >= 0; al--) { // önce al
▶        Dugum c = cocuk(u, al);         // sinir(c) dahil
▶        if (c.agirlik <= W && c.kazanc > enIyi)
▶          enIyi = c.kazanc;               // yeni aday
▶        if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



E kuyruğun başından çıkar ve dallanır: H (al: s = 98 → kuyruğa, **enİyi = 90**); I (alma: s = 50 ≤ 90, budandı).

Adım 6 / 9

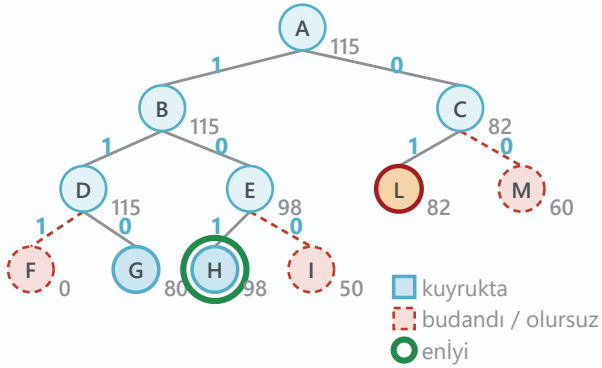
DEĞİŞKENLER

`u = E` `enIyi = 90`

`kuyruk = [L, G, H]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
7      for (int al = 1; al >= 0; al--) { // önce al
8        Dugum c = cocuk(u, al);         // sinir(c) dahil
9        if (c.agirlik <= W && c.kazanc > enIyi)
10         enIyi = c.kazanc;               // yeni aday
11       if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



L çıkar: sınırı 82 ≤ enİyi = 90 → **tekrar sınamada atılır** (kuyruğa girerken umut vardı, beklerken enİyi büyüdü).

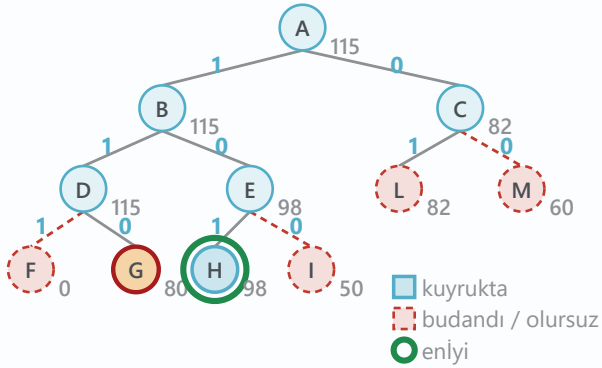
Adım 7 / 9

DEĞİŞKENLER

`u = L` `enIyi = 90` `kuyruk = [G, H]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
▶      Dugum u = kuyruk.poll();          // ilk giren
▶      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
7      for (int al = 1; al >= 0; al--) { // önce al
8        Dugum c = cocuk(u, al);         // sinir(c) dahil
9        if (c.agirlik <= W && c.kazanc > enIyi)
10         enIyi = c.kazanc;               // yeni aday
11       if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
14   return enIyi;
15 }
```



G çıkar: sınırı 80 ≤ enİyi = 90 → **tekrar sınamada atılır** (kuyruğa girerken umut vardı, beklerken enİyi büyüdü).

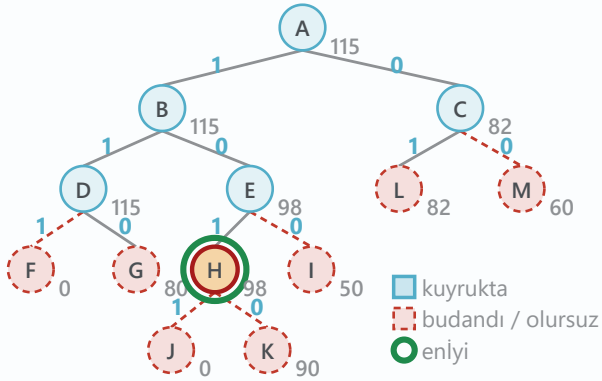
Adım 8 / 9

DEĞİŞKENLER

`u = G` `enIyi = 90` `kuyruk = [H]`

FIFO Dal Sınır: Kod İzleme

```
1  int fifoDalSinir() {
2    ArrayDeque<Dugum> kuyruk = new ArrayDeque<>();
3    kuyruk.add(kok);  int enIyi = 0;
4    while (!kuyruk.isEmpty()) {
5      Dugum u = kuyruk.poll();          // ilk giren
6      if (u.sinir <= enIyi || u.seviye == n - 1) continue;
7      for (int al = 1; al >= 0; al--) { // önce al
8        Dugum c = cocuk(u, al);         // sinir(c) dahil
▶        if (c.agirlik <= W && c.kazanc > enIyi)
▶          enIyi = c.kazanc;               // yeni aday
▶        if (c.sinir > enIyi) kuyruk.add(c);
12     }
13   }
▶    return enIyi;
15 }
```



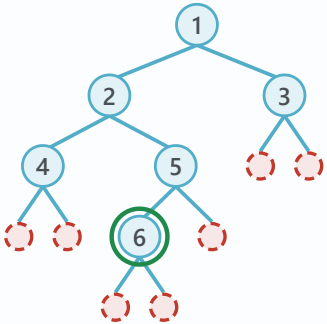
H kuyruğun başından çıkar ve dallanır: J (al: ağırlık 17 > 16, olursuz); K (alma: s = 90 ≤ 90, budandı). Kuyruk boş: **enİyi = 90** (eşya 1 ve 3). 13 düğüm üretildi, 6 genişletme, kuyruk en çok 3.

Adım 9 / 9

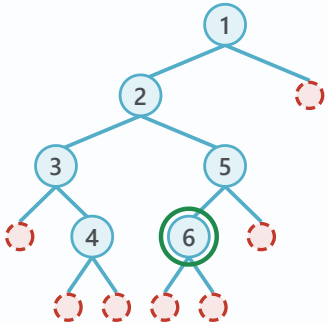
DEĞİŞKENLER

`u = H` `enIyi = 90` `kuyruk = []`

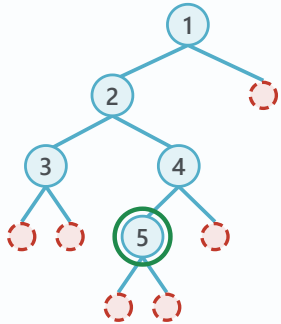
Aynı Örnekte Üç Strateji



FIFO: 13 düğüm, 6 genişletme



LIFO: 13 düğüm, 6 genişletme



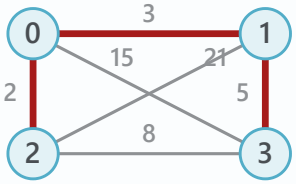
En iyi önce: 11 düğüm, 5 genişletme

Numaralar genişletme sırasıdır; küçük daireler üretilip genişletilmeyen düğümler (kırmızı kesikli: budanan), halka: optimum (90).

Yöntem	Üretilen düğüm	Genişletilen	En büyük kuyruk
Kaba kuvvet (tam ağaç)	31	15	—
FIFO dal sınır	13	6	3
LIFO dal sınır (önce "al")	13	6	3
En iyi önce (LC)	11	5	3

Gezgin Satıcı Problemi (TSP): Alt Sınır

- En küçükleme: aranan, en kısa tur; düğümlerin sınırı **alt sınırdır**.
- Düğüm = 0'dan başlayan kısmi yol; çocuklar = ziyaret edilmemiş bir sonraki şehir.
- **Alt sınır:** henüz **çıkılmamış** her şehirden (bulunulan şehir + ziyaret edilmemişler) en az bir kenarla çıkılmak zorundadır:



kalın: her şehrin en ucuz kenarı

2 + 3 + 2 + 5 = 12

	0	1	2	3	min
0	—	3	2	15	**2**
1	3	—	21	5	**3**
2	2	21	—	8	**2**
3	15	5	8	—	**5**

4 şehirli simetrik uzaklık matrisi d.

$$\text{alt(yol)} = \text{maliyet(yol)} + \text{enKucuk[son şehir]} + \Sigma_{v \text{ ziyaret edilmemiş}} \text{enKucuk[v]}$$

Kök: 2 + 3 + 2 + 5 = 12. Yol 0 → 1: 3 + 3 + (2 + 5) = 13. Her gerçek turun maliyeti bu toplamdan küçük olamaz, bu yüzden sınır geçerlidir.

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
▶ void tsp(int sehir, int maliyet, int sayac) {
2   if (sayac == n) {           // tur tamam
3     int tur = maliyet + d[sehir][0];
4     if (tur < enIyi) enIyi = tur;
5     return;
6   }
7   for (int s = 0; s < n; s++) {
8     if (ziyaret[s]) continue;
9     int yeni = maliyet + d[sehir][s];
10    ziyaret[s] = true;
11    if (altSinir(s, yeni) < enIyi) // sınır
12      tsp(s, yeni, sayac + 1);
13    ziyaret[s] = false;      // geri al
14  }
15 }
```

ÇAĞRI YIĞINI

`tsp(0, 0, 1)`

0

a=12

Şehir 0'dan başlanır. Kökün alt sınırı = 2 + 3 + 2 + 5 = **12**.

Adım 1 / 13

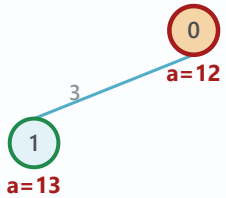
DEĞİŞKENLER

`sehir = 0` `maliyet = 0` `sayac = 1` `s = –`

`yeni = –` `alt = –` `enIyi = ∞`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
▶        tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



0 → 1: yeni = 0 + 3 = 3; alt = 13 < ∞ → **dallan**.

Adım 2 / 13

ÇAĞRI YIĞINI

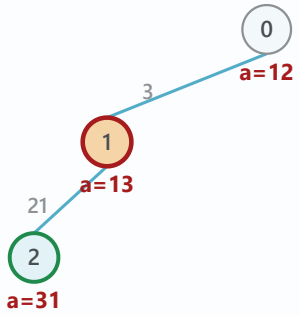
`tsp(0, 0, 1)`

DEĞİŞKENLER

`sehir = 0` `maliyet = 0` `sayac = 1` `s = 1` `yeni = 3` `alt = 13` `enIyi = ∞`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
▶        tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



1 → 2: yeni = 3 + 21 = 24; alt = 31 < ∞ → **dallan**.

Adım 3 / 13

ÇAĞRI YIĞINI

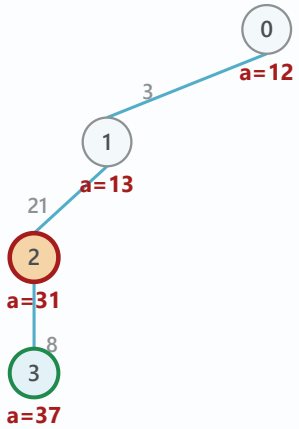
- `tsp(1, 3, 2)`
- `tsp(0, 0, 1)`

DEĞİŞKENLER

`sehir = 1` `maliyet = 3` `sayac = 2` `s = 2` `yeni = 24` `alt = 31` `enIyi = ∞`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
▶        tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



2 → 3: yeni = 24 + 8 = 32; alt = 37 < ∞ → **dallan**.

Adım 4 / 13

ÇAĞRI YIĞINI

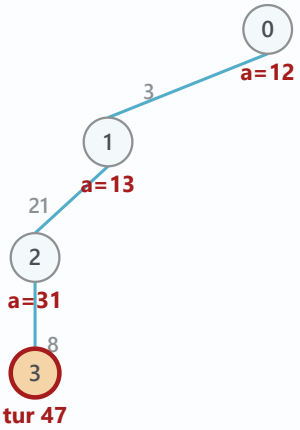
- `tsp(2, 24, 3)`
- `tsp(1, 3, 2)`
- `tsp(0, 0, 1)`

DEĞİŞKENLER

`sehir = 2` `maliyet = 24` `sayac = 3` `s = 3` `yeni = 32` `alt = 37` `enIyi = ∞`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
▶      int tur = maliyet + d[sehir][0];
▶      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
11     if (altSinir(s, yeni) < enIyi) // sınır
12       tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



Tur 0 → 1 → 2 → 3 → 0 = 32 + 15 = **47** < ∞ → enİyi = 47.

Adım 5 / 13

ÇAĞRI YIĞINI

- `tsp(3, 32, 4)`
- `tsp(2, 24, 3)`
- `tsp(1, 3, 2)`
- `tsp(0, 0, 1)`

DEĞİŞKENLER

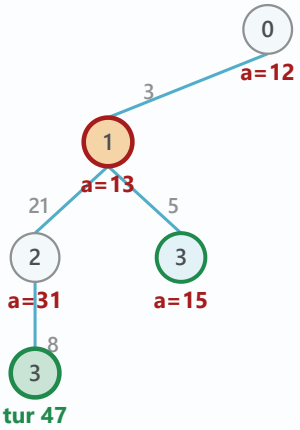
`sehir = 3` `maliyet = 32` `sayac = 4` `s = –`
`yeni = –` `alt = –` `enIyi = 47`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
▶        tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```

ÇAĞRI YIĞINI

- **tsp(1, 3, 2)**
- tsp(0, 0, 1)



1 → 3: yeni = 3 + 5 = 8; alt = 15 < 47 → **dallan**.

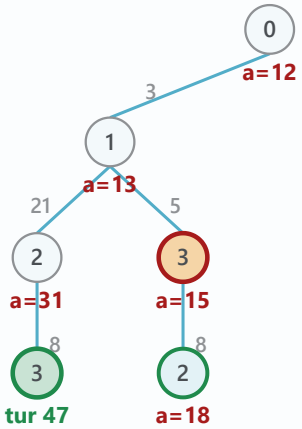
Adım 6 / 13

DEĞİŞKENLER

`sehir = 1` `maliyet = 3` `sayac = 2` `s = 3` `yeni = 8` `alt = 15` `enIyi = 47`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
▶        tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



3 → 2: yeni = 8 + 8 = 16; alt = 18 < 47 → **dallan**.

Adım 7 / 13

ÇAĞRI YIĞINI

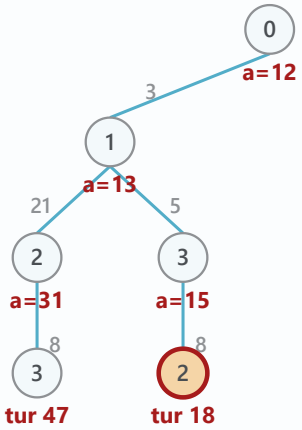
- `tsp(3, 8, 3)`
- `tsp(1, 3, 2)`
- `tsp(0, 0, 1)`

DEĞİŞKENLER

`sehir = 3` `maliyet = 8` `sayac = 3` `s = 2` `yeni = 16` `alt = 18` `enIyi = 47`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
▶      int tur = maliyet + d[sehir][0];
▶      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
11     if (altSinir(s, yeni) < enIyi) // sınır
12       tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



Tur 0 → 1 → 3 → 2 → 0 = 16 + 2 = **18** < 47 → enİyi = 18.

Adım 8 / 13

ÇAĞRI YIĞINI

- **tsp(2, 16, 4)**
- tsp(3, 8, 3)
- tsp(1, 3, 2)
- tsp(0, 0, 1)

DEĞİŞKENLER

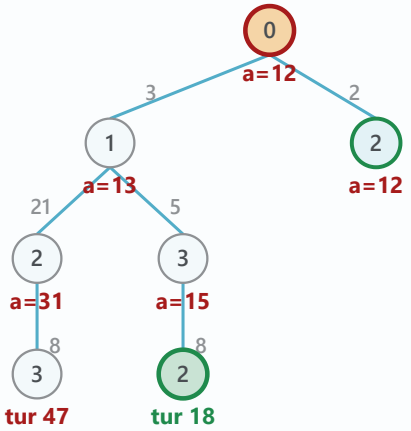
`sehir = 2` `maliyet = 16` `sayac = 4` `s = –`
`yeni = –` `alt = –` `enIyi = 18`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
▶        tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```

ÇAĞRI YIĞINI

`tsp(0, 0, 1)`



0 → 2: yeni = 0 + 2 = 2; alt = 12 < 18 → **dallan**.

Adım 9 / 13

DEĞİŞKENLER

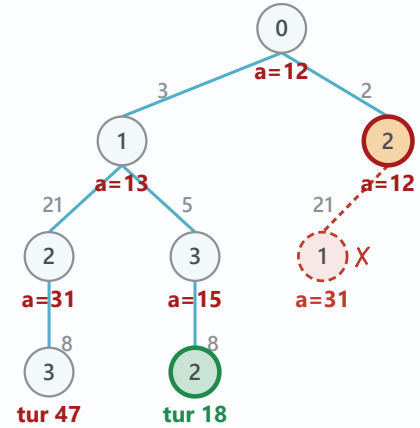
`sehir = 0` `maliyet = 0` `sayac = 1` `s = 2`
`yeni = 2` `alt = 12` `enIyi = 18`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
12       tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```

ÇAĞRI YIĞINI

- **tsp(2, 2, 2)**
- tsp(0, 0, 1)



2 → 1: yeni = 23; alt = 31 ≥ enİyi = 18 → **budandı**.

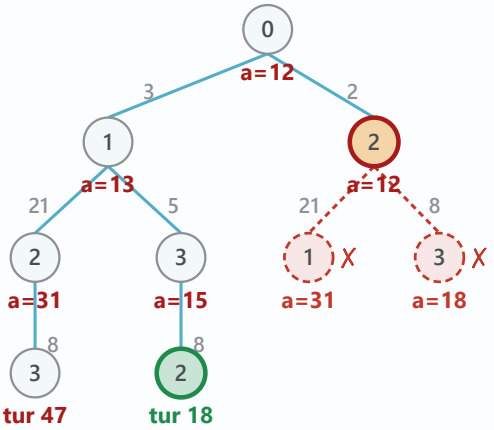
Adım 10 / 13

DEĞİŞKENLER

`sehir = 2` `maliyet = 2` `sayac = 2` `s = 1` `yeni = 23` `alt = 31` `enIyi = 18`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
12       tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```



2 → 3: yeni = 10; alt = 18 ≥ enİyi = 18 → **budandı**.

Adım 11 / 13

ÇAĞRI YIĞINI

- `tsp(2, 2, 2)`
- `tsp(0, 0, 1)`

DEĞİŞKENLER

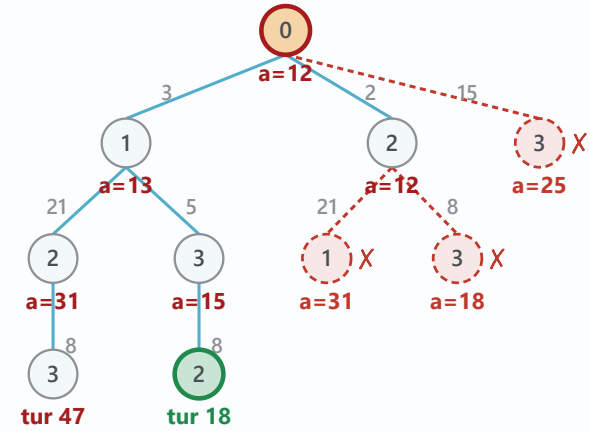
`sehir = 2` `maliyet = 2` `sayac = 2` `s = 3` `yeni = 10` `alt = 18` `enIyi = 18`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
▶      if (altSinir(s, yeni) < enIyi) // sınır
12       tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
15 }
```

ÇAĞRI YIĞINI

`tsp(0, 0, 1)`



0 → 3: yeni = 15; alt = 25 ≥ enİyi = 18 → **budandı**.

Adım 12 / 13

DEĞİŞKENLER

`sehir = 0` `maliyet = 0` `sayac = 1` `s = 3`

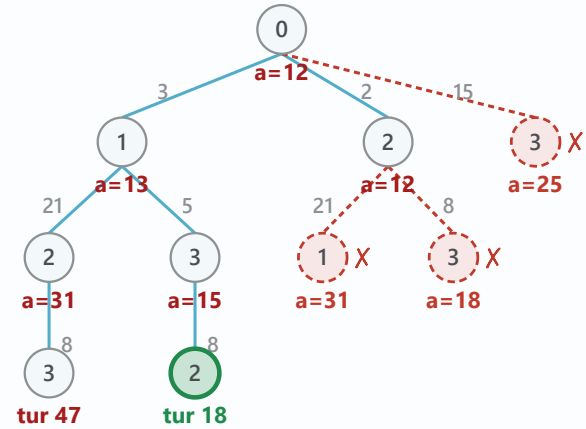
`yeni = 15` `alt = 25` `enIyi = 18`

tsp(0, 0, 1): Sınırlı Derinlik Öncelikli Arama

```
1  void tsp(int sehir, int maliyet, int sayac) {
2    if (sayac == n) {            // tur tamam
3      int tur = maliyet + d[sehir][0];
4      if (tur < enIyi) enIyi = tur;
5      return;
6    }
7    for (int s = 0; s < n; s++) {
8      if (ziyaret[s]) continue;
9      int yeni = maliyet + d[sehir][s];
10     ziyaret[s] = true;
11     if (altSinir(s, yeni) < enIyi) // sınır
12       tsp(s, yeni, sayac + 1);
13     ziyaret[s] = false;       // geri al
14   }
▶ }
```

ÇAĞRI YIĞINI

boş



Bitti: en kısa tur **0 → 1 → 3 → 2 → 0 = 18**; 16 yerine 10 düğüm.

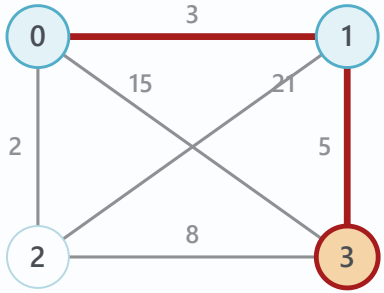
Adım 13 / 13

DEĞİŞKENLER

`sehir = 0` `maliyet = 0` `sayac = 1` `s = –` `yeni = –` `alt = –` `enIyi = 18`

altSinir(3, 8): Kod İzleme

```
1 int altSinir(int son, int maliyet) {
▶   int alt = maliyet + enKucuk[son];
3   for (int v = 0; v < n; v++)
4     if (!ziyaret[v])       // gidilmemiş
5       alt += enKucuk[v];
6   return alt;
7 }
```



v	enKucuk	ziy.
0	2	✓
1	3	✓
2	2	—
3	5	✓

yol 0 → 1 → 3, maliyet 8

TSP izlemesindeki 1 → 3 adımı: yol 0 → 1 → 3, maliyet 3 + 5 = 8; ziyaret = {0, 1, 3}. Son şehir 3'ten bir kez daha çıkılacak: alt = 8 + enKucuk[3] = 8 + 5 = **13**.

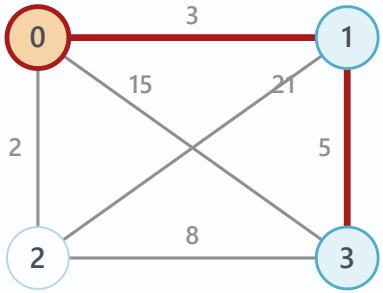
Adım 1 / 6

DEĞİŞKENLER

`son = 3` `maliyet = 8` `v = –` `alt = 13`

altSinir(3, 8): Kod İzleme

```
1 int altSinir(int son, int maliyet) {
2   int alt = maliyet + enKucuk[son];
▶   for (int v = 0; v < n; v++)
▶     if (!ziyaret[v])       // gidilmemiş
5       alt += enKucuk[v];
6   return alt;
7 }
```



v	enKucuk	ziy.
0	2	✓
1	3	✓
2	2	—
3	5	✓

yol 0 → 1 → 3, maliyet 8

v = 0: ziyaret edildi (yolda) → katkı yok, çıkış kenarı zaten maliyette.

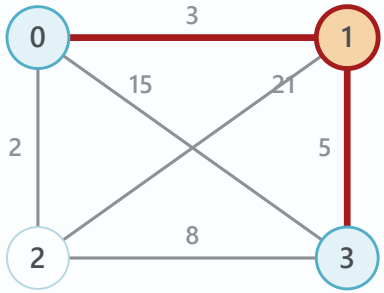
Adım 2 / 6

DEĞİŞKENLER

`son = 3` `maliyet = 8` `v = 0` `alt = 13`

altSinir(3, 8): Kod İzleme

```
1 int altSinir(int son, int maliyet) {
2   int alt = maliyet + enKucuk[son];
▶   for (int v = 0; v < n; v++)
▶     if (!ziyaret[v])       // gidilmemiş
5       alt += enKucuk[v];
6   return alt;
7 }
```



v	enKucuk	ziy.
0	2	✓
1	3	✓
2	2	—
3	5	✓

yol 0 → 1 → 3, maliyet 8

v = 1: ziyaret edildi (yolda) → katkı yok, çıkış kenarı zaten maliyette.

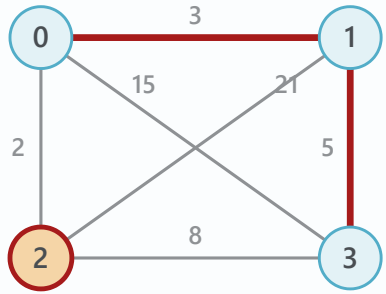
Adım 3 / 6

DEĞİŞKENLER

`son = 3` `maliyet = 8` `v = 1` `alt = 13`

altSinir(3, 8): Kod İzleme

```
1 int altSinir(int son, int maliyet) {
2   int alt = maliyet + enKucuk[son];
3   for (int v = 0; v < n; v++)
▶     if (!ziyaret[v])       // gidilmemiş
▶       alt += enKucuk[v];
6   return alt;
7 }
```



v	enKucuk	ziy.
0	2	✓
1	3	✓
2	2	—
3	5	✓

yol 0 → 1 → 3, maliyet 8

v = 2: henüz gidilmedi → en az enKucuk[2] = 2 ile çıkılmalı: alt = **15**.

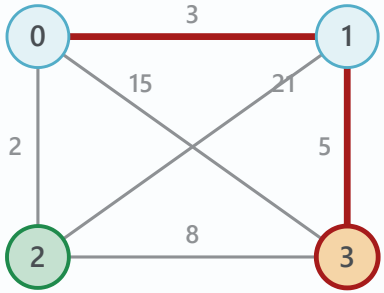
Adım 4 / 6

DEĞİŞKENLER

`son = 3` `maliyet = 8` `v = 2` `alt = 15`

altSinir(3, 8): Kod İzleme

```
1 int altSinir(int son, int maliyet) {
2   int alt = maliyet + enKucuk[son];
▶   for (int v = 0; v < n; v++)
▶     if (!ziyaret[v])       // gidilmemiş
5       alt += enKucuk[v];
6   return alt;
7 }
```



v	enKucuk	ziy.
0	2	✓
1	3	✓
2	2	—
3	5	✓

yol 0 → 1 → 3, maliyet 8

v = 3: ziyaret edildi (yolda) → katkı yok; 3'ün çıkışı başta eklendi.

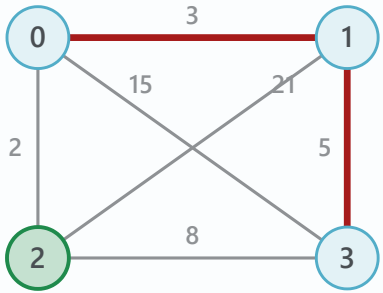
Adım 5 / 6

DEĞİŞKENLER

`son = 3` `maliyet = 8` `v = 3` `alt = 15`

altSinir(3, 8): Kod İzleme

```
1 int altSinir(int son, int maliyet) {
2   int alt = maliyet + enKucuk[son];
3   for (int v = 0; v < n; v++)
4     if (!ziyaret[v])       // gidilmemiş
5       alt += enKucuk[v];
▶   return alt;
7 }
```



v	enKucuk	ziy.
0	2	✓
1	3	✓
2	2	—
3	5	✓

yol 0 → 1 → 3, maliyet 8

alt = **15** döner. Bu daldaki her tur ≥ 15; gerçek tur 0 → 1 → 3 → 2 → 0 = 18. 15 < enİyi = 47 olduğundan TSP izlemesinde dala girildi.

Adım 6 / 6

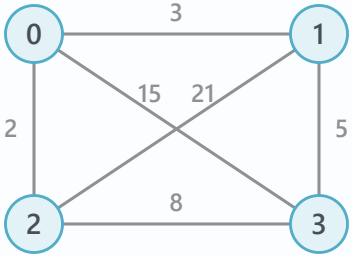
DEĞİŞKENLER

`son = 3` `maliyet = 8` `v = –` `alt = 15`

Daha Sıkı Bir Alt Sınır: İki En Kısa Kenar

- Turda her şehre **bir kenarla girilir, bir kenarla çıkılır**: her şehir tam iki tur kenarına değer.
- Şehrin bu iki kenarı, ona değen en kısa iki kenardan kısa olamaz. Tüm şehirler için toplanınca her kenar iki kez sayılır:

$$\text{tur} \geq \tfrac{1}{2} \cdot \Sigma_v \text{ (v'nin en kısa iki kenarı)}$$



v en küçük iki toplam

0 2 + 3 = 5

1 3 + 5 = 8

2 2 + 8 = 10

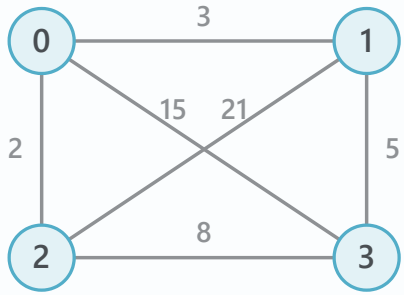
3 5 + 8 = 13

36 / 2 = 18

Aynı matriste “en ucuz çıkış kenarı” sınırı 12 veriyordu; bu sınır 18'dir ve optimum tur 0 → 1 → 3 → 2 → 0 = 3 + 5 + 8 + 2 = 18'e eşittir. Kökte sınır = optimum olduğu için ilk tam tur bulunur bulunmaz kalan bütün dallar budanır.

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
▶    int toplam = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // en küçük iki
5      for (int u = 0; u < n; u++) {
6        if (u == v) continue;
7        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = — m2 = — toplam = 0

Simetrik 4 şehirli matris. toplam = 0; her şehir için en kısa iki kenar aranacak.

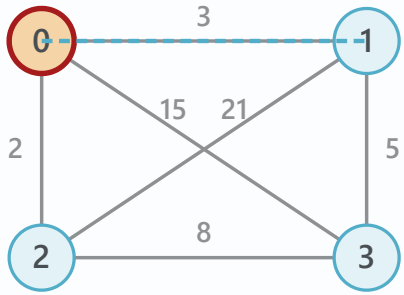
Adım 1 / 8

DEĞİŞKENLER

`v = –` `u = –` `m1 = –` `m2 = –` `toplam = 0`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // en küçük iki
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 3 m2 = ∞ toplam = 0

v = 0, u = 1: d = 3 < m1 = ∞ → m1 = 3 (eski m1 m2'ye kayar).

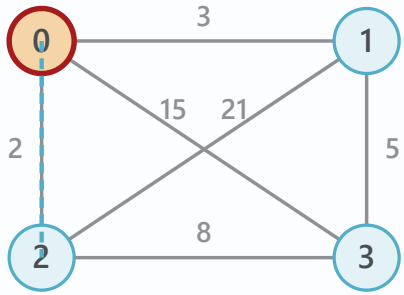
Adım 2 / 8

DEĞİŞKENLER

`v = 0` `u = 1` `m1 = 3` `m2 = ∞` `toplam = 0`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // en küçük iki
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 2 m2 = 3 toplam = 0

u = 2: d = 2 < m1 = 3 → m2 = 3, m1 = 2. İki en küçük kayar gibi güncellenir.

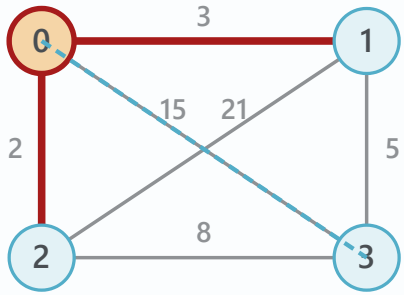
Adım 3 / 8

DEĞİŞKENLER

`v = 0` `u = 2` `m1 = 2` `m2 = 3` `toplam = 0`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // en küçük iki
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 2 m2 = 3 toplam = 5

u = 3: 15 ikisinden de büyük, değişmez. toplam += 2 + 3 = **5**.

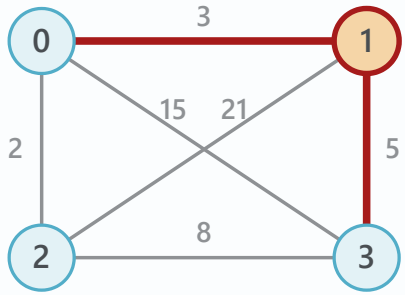
Adım 4 / 8

DEĞİŞKENLER

`v = 0` `u = 3` `m1 = 2` `m2 = 3` `toplam = 5`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // en küçük iki
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 3 m2 = 5 toplam = 13

v = 1: kenarlar 3, 21, 5 → m1 = 3, m2 = 5. toplam = **13**.

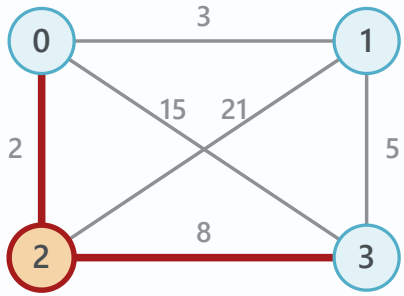
Adım 5 / 8

DEĞİŞKENLER

`v = 1` `u = 3` `m1 = 3` `m2 = 5` `toplam = 13`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // en küçük iki
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 2 m2 = 8 toplam = 23

v = 2: kenarlar 2, 21, 8 → 2 + 8. toplam = **23**.

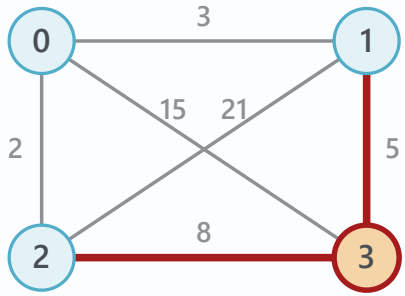
Adım 6 / 8

DEĞİŞKENLER

`v = 2` `u = 3` `m1 = 2` `m2 = 8` `toplam = 23`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
▶    for (int v = 0; v < n; v++) {
▶      int m1 = INF, m2 = INF;     // en küçük iki
▶      for (int u = 0; u < n; u++) {
▶        if (u == v) continue;
▶        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
▶        else if (d[v][u] < m2) m2 = d[v][u];
9      }
▶      toplam += m1 + m2;
11   }
12   return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 5 m2 = 8 toplam = 36

v = 3: kenarlar 15, 5, 8 → 5 + 8. toplam = **36**.

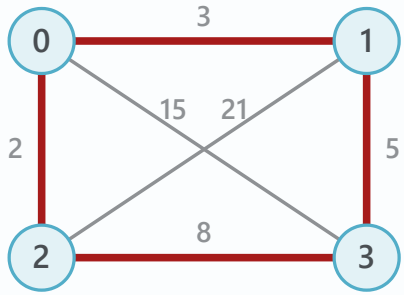
Adım 7 / 8

DEĞİŞKENLER

`v = 3` `u = 2` `m1 = 5` `m2 = 8` `toplam = 36`

ikiKenarSinir(): Kod İzleme

```
1  double ikiKenarSinir() {
2    int toplam = 0;
3    for (int v = 0; v < n; v++) {
4      int m1 = INF, m2 = INF;     // en küçük iki
5      for (int u = 0; u < n; u++) {
6        if (u == v) continue;
7        if (d[v][u] < m1) { m2 = m1; m1 = d[v][u];
   }
8        else if (d[v][u] < m2) m2 = d[v][u];
9      }
10     toplam += m1 + m2;
11   }
▶    return toplam / 2.0;  // her kenar iki kez
13 }
```



m1 = 5 m2 = 8 toplam = 36

36 / 2 = **18** döner. Kalın kenarlar tam da optimum turu (0-1-3-2-0) oluşturur; bu yüzden sınır sıkıdır. Kısmi yolda aynı fikir, yoldaki kenarlar sabitlenerek uygulanır.

Adım 8 / 8

DEĞİŞKENLER

`toplam = 36` `sonuç = 18.0`

İndirgenmiş Maliyet Matrisi ile Alt Sınır

- Köşegen ∞ yapılır. Her **satırdan** en küçüğü, sonra her **sütundan** en küçüğü çıkarılır.
- Her tur her satırdan ve her sütundan tam bir hücre kullanır; çıkarılan toplam her turun maliyetinden düşülmüştür → **alt sınır**.
- Örnek: satırlar 2 + 3 + 2 + 5 = 12, sütun 3'ten 2 → kök sınırı **14**.
- **Dallanma** i → j kenarıyla: i satırı ve j sütunu ∞, (j, i) ∞ yapılır, matris yeniden indirgenir. Çocuğun sınırı = ebeveyn + indirgenmiş c[i][j] + yeni indirgeme.

Kökten dal	sınır
0 → 1	14 + 1 + 3 = 18
0 → 2	14 + 0 + 4 = 18
0 → 3	14 + 11 + 3 = 28

	0	1	2	3
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4
3	10	0	3	∞

İndirgenmiş matris (toplam 14). Her satırda ve sütunda en az bir 0 var.

0 → 3 dalı 28 ile, optimum 18 bulunduğu anda budanır. Bu yöntem asimetrik (d[i][j] ≠ d[j][i]) TSP'de de geçerlidir (Little ve diğ., 1963).

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
▶    int toplam = 0;
3    for (int i = 0; i < n; i++) {        // satırlar
4      int mn = satirMin(m, i);
5      if (mn > 0 && mn < INF) {
6        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // sütunlar
10     int mn = sutunMin(m, j);
11     if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```

	0	1	2	3
0	∞	3	2	15
1	3	∞	21	5
2	2	21	∞	8
3	15	5	8	∞

toplam = 0

Köşegeni ∞ olan 4 × 4 uzaklık matrisi; toplam = 0.

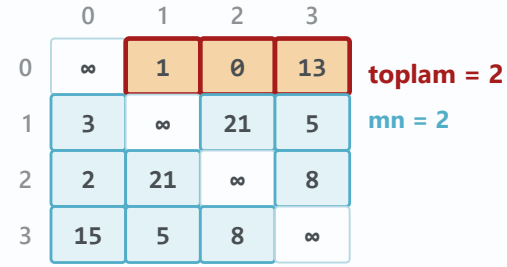
Adım 1 / 8

DEĞİŞKENLER

`i = –` `j = –` `mn = –` `toplam = 0`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
▶    for (int i = 0; i < n; i++) {        // satırlar
▶      int mn = satirMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // sütunlar
10     int mn = sutunMin(m, j);
11     if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```



Satır 0: en küçük 2 → satırdan 2 çıkar: 1, 0, 13. toplam = 2.

Adım 2 / 8

DEĞİŞKENLER

`i = 0` `mn = 2` `toplam = 2`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
▶    for (int i = 0; i < n; i++) {        // satırlar
▶      int mn = satirMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // sütunlar
10     int mn = sutunMin(m, j);
11     if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```

	0	1	2	3
0	∞	1	0	13
1	0	∞	18	2
2	2	21	∞	8
3	15	5	8	∞

toplam = 5

mn = 3

Satır 1: en küçük 3 → 0, 18, 2. toplam = 5.

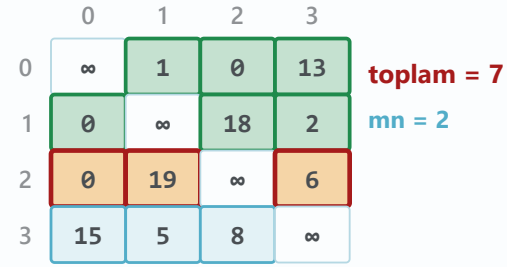
Adım 3 / 8

DEĞİŞKENLER

`i = 1` `mn = 3` `toplam = 5`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
▶    for (int i = 0; i < n; i++) {        // satırlar
▶      int mn = satirMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // sütunlar
10     int mn = sutunMin(m, j);
11     if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```



Satır 2: en küçük 2 → 0, 19, 6. toplam = 7.

Adım 4 / 8

DEĞİŞKENLER

`i = 2` `mn = 2` `toplam = 7`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
▶    for (int i = 0; i < n; i++) {        // satırlar
▶      int mn = satirMin(m, i);
▶      if (mn > 0 && mn < INF) {
▶        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // sütunlar
10     int mn = sutunMin(m, j);
11     if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```

	0	1	2	3
0	∞	1	0	13
1	0	∞	18	2
2	0	19	∞	6
3	10	0	3	∞

toplam = 12

mn = 5

Satır 3: en küçük 5 → 10, 0, 3. Satır indirgemesi bitti: toplam = 12 (en ucuz çıkış sınırının aynısı).

Adım 5 / 8

DEĞİŞKENLER

`i = 3` `mn = 5` `toplam = 12`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
3    for (int i = 0; i < n; i++) {        // satırlar
4      int mn = satirMin(m, i);
5      if (mn > 0 && mn < INF) {
6        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
▶    for (int j = 0; j < n; j++) {        // sütunlar
▶      int mn = sutunMin(m, j);
▶      if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```

	0	1	2	3
0	∞	1	0	13
1	0	∞	18	2
2	0	19	∞	6
3	10	0	3	∞

toplam = 12

mn = 0

Sütun 0: en küçük 0 → değişiklik yok. Sütun 1 ve 2 de zaten 0 içerir.

Adım 6 / 8

DEĞİŞKENLER

`j = 0` `mn = 0` `toplam = 12`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
3    for (int i = 0; i < n; i++) {        // satırlar
4      int mn = satirMin(m, i);
5      if (mn > 0 && mn < INF) {
6        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
▶    for (int j = 0; j < n; j++) {        // sütunlar
▶      int mn = sutunMin(m, j);
▶      if (mn > 0 && mn < INF) {
▶        sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
15   return toplam;
16 }
```

	0	1	2	3
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4
3	10	0	3	∞

toplam = 14

mn = 2

Sütun 3: değerler 13, 2, 6 → en küçük 2; çıkarılır: 11, 0, 4. toplam = **14**. Sütun adımı, "her şehre bir kez girilir" bilgisini de sınıra katar.

Adım 7 / 8

DEĞİŞKENLER

`j = 3` `mn = 2` `toplam = 14`

indirge(): Satır ve Sütun İndirgemesi

```
1  int indirge(int[][] m) {
2    int toplam = 0;
3    for (int i = 0; i < n; i++) {        // satırlar
4      int mn = satirMin(m, i);
5      if (mn > 0 && mn < INF) {
6        satirdanCikar(m, i, mn); toplam += mn;
7      }
8    }
9    for (int j = 0; j < n; j++) {        // sütunlar
10     int mn = sutunMin(m, j);
11     if (mn > 0 && mn < INF) {
12       sutundanCikar(m, j, mn); toplam += mn;
13     }
14   }
▶    return toplam;
16 }
```

	0	1	2	3
0	∞	1	0	11
1	0	∞	18	0
2	0	19	∞	4
3	10	0	3	∞

toplam = 14

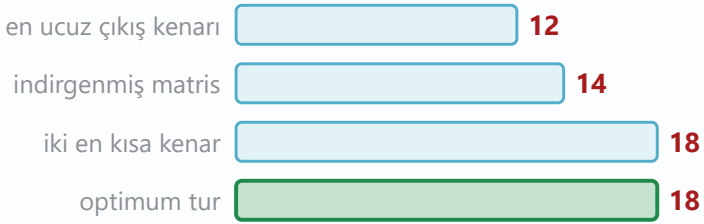
Sınır 14 döner: 12'den sıkı, iki kenar sınırından (18) gevşek. Matris, çocuk düğümlerin sınırı için saklanır.

Adım 8 / 8

DEĞİŞKENLER

`toplam = 14`

Üç TSP Sınırı: Sıkılık ve Maliyet



Sınır	Düğüm başına maliyet	Neyi kullanır?	Not
En ucuz çıkış kenarı	O(n) (önceden enKucuk[])	her şehirden bir çıkış	en ucuz, en gevşek
İndirgenmiş matris	$O(n^2)$	her şehre bir giriş + bir çıkış	asimetrik TSP'de de geçerli
İki en kısa kenar	$O(n^2)$	her şehre değen iki kenar	simetrik TSP için

Sıkı sınır daha çok budar ama her düğümde daha pahalıdır. Daha da sıkı sınırlar (1-ağaç, LP gevşetmesi) büyük örneklerde kullanılır; seçim, budanan düğüm sayısı ile düğüm başına iş arasındaki dengeye göre yapılır.

Daha Büyük Örnek: 6 Şehir

	0	1	2	3	4	5	min
0	—	29	20	21	16	31	**16**
1	29	—	15	29	28	40	**15**
2	20	15	—	15	14	25	**14**
3	21	29	15	—	12	18	**12**
4	16	28	14	12	—	22	**12**
5	31	40	25	18	22	—	**18**

- Budamasız geri izleme tüm (n−1)! = 120 turu dener; `altSinir` testiyle aynı sonuç çok daha az düğümle bulunur.
- Kök alt sınırı: 16 + 15 + 14 + 12 + 12 + 18 = **87**; optimum **115**.
- En iyi tur: **0** → **1** → **2** → **3** → **5** → **4** → **0** = 29 + 15 + 15 + 18 + 22 + 16 = 115.

Sürüm	Üretilen düğüm	tsp çağrısı
Budamasız (geri izleme)	326	326
En ucuz çıkış sınırı	210	130

Daha sıkı sınırlar

Simetrik TSP'de her şehrin **iki** en kısa kenarının toplamının yarısı; ya da satır/sütun **indirgenmiş maliyet matrisi** (Little ve diğ., 1963). Sıkı sınır daha çok budar ama her düğümde daha pahalıdır.

Atama Problemi: n Kişi, n İş

- c[i][j]: i. kişinin j. işi yapma maliyeti. Her kişiye farklı bir iş verilerek toplam en küçüklenir; n! olası atama vardır.
- Durum uzayı: i. düzeyde i. kişiye boş bir iş seçilir.
- **Alt sınır:** kalan her kişi en az kendi satırının en küçüğünü öder (iş çakışmaları yok sayılır):

$$\text{alt} = \text{yeni} + \text{kalanMin}[i+1],\quad \text{kalanMin}[i] = \Sigma_{r \geq i} \min_j c[r][j]$$

alt ≥ enİyi ise dal kesilir. Satır en küçükleri 4, 3, 2 → kalanMin = [9, 5, 2, 0]: kökün sınırı 9.

kişi \ iş	0	1	2	min
0	5	8	4	**4**
1	7	3	6	**3**
2	2	9	5	**2**

Atama problemi polinom zamanda da çözülür (Macar yöntemi, $O(n^3)$). Dal sınır burada yöntemi küçük ve kolay izlenir bir örnekte göstermek içindir; ek kısıtlı (ör. bazı işler birlikte yapılamaz) türevlerinde gerçekten gerekir.

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                      // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
7      if (kullanildi[j]) continue;
8      int yeni = maliyet + c[i][j];
9      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                      // sınır
11     kullanildi[j] = true;
12     ata(i + 1, yeni);
13     kullanildi[j] = false;           // geri al
14   }
15 }
```

	iş 0	iş 1	iş 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

enİyi = ∞

kalanMin = [9, 5, 2, 0]; enİyi = ∞. ata(0, 0) çağrılır: kişi 0 için işler sırayla denenir.

Adım 1 / 10

DEĞİŞKENLER

`i = 0` `j = –` `yeni = –` `alt = –` `enİyi = ∞`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                        // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                        // sınır
▶      kullanildi[j] = true;
▶      ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```

	iş 0	iş 1	iş 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

enİyi = ∞

alt = 10

k0 → iş 0: yeni = 5, alt = 5 + kalanMin[1] = 10 < ∞ → dallan.

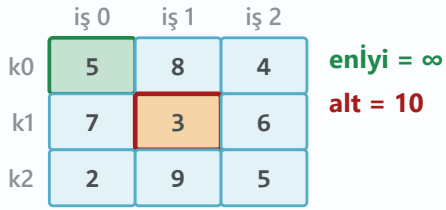
Adım 2 / 10

DEĞİŞKENLER

`i = 0` `j = 0` `yeni = 5` `alt = 10` `enİyi = ∞`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                        // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                        // sınır
▶      kullanildi[j] = true;
▶      ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```



k1: iş 0 dolu, atlanır. İş 1: yeni = 8, alt = 8 + 2 = 10 → dallan.

Adım 3 / 10

DEĞİŞKENLER

`i = 1` `j = 1` `yeni = 8` `alt = 10` `enİyi = ∞`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
▶    if (i == n) {                        // tam atama
▶      enIyi = Math.min(enIyi, maliyet);
▶      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                        // sınır
▶      kullanildi[j] = true;
▶      ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```

	iş 0	iş 1	iş 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

enİyi = 13

k2: yalnız iş 2 boş: yeni = 13 → i = 3, tam atama: enİyi = **13**.

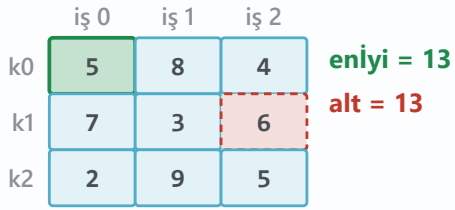
Adım 4 / 10

DEĞİŞKENLER

`i = 3` `maliyet = 13` `enİyi = 13`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                      // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
▶        continue;                      // sınır
11     kullanildi[j] = true;
12     ata(i + 1, yeni);
▶      kullanildi[j] = false;           // geri al
14   }
15 }
```



Geri dönülür. k1 → iş 2: yeni = 11, alt = 11 + 2 = 13 ≥ 13 → **kesilir**.

Adım 5 / 10

DEĞİŞKENLER

`i = 1` `j = 2` `yeni = 11` `alt = 13` `enİyi = 13`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                        // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
▶        continue;                        // sınır
11     kullanildi[j] = true;
12     ata(i + 1, yeni);
▶      kullanildi[j] = false;             // geri al
14   }
15 }
```

	iş 0	iş 1	iş 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

enİyi = 13

alt = 13

k0 → iş 1: yeni = 8, alt = 8 + 5 = 13 ≥ 13 → kesilir; altındaki 2 atama hiç denenmez.

Adım 6 / 10

DEĞİŞKENLER

`i = 0` `j = 1` `yeni = 8` `alt = 13` `enİyi = 13`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                        // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                        // sınır
▶      kullanildi[j] = true;
▶      ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```

	iş 0	iş 1	iş 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

enİyi = 13

alt = 9

k0 → iş 2: yeni = 4, alt = 9 < 13 → dallan.

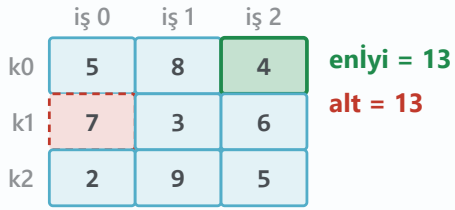
Adım 7 / 10

DEĞİŞKENLER

`i = 0` `j = 2` `yeni = 4` `alt = 9` `enİyi = 13`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                        // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
▶        continue;                        // sınır
11     kullanildi[j] = true;
12     ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```



k1 → iş 0: yeni = 11, alt = 13 ≥ 13 → kesilir.

Adım 8 / 10

DEĞİŞKENLER

`i = 1` `j = 0` `yeni = 11` `alt = 13` `enİyi = 13`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
2    if (i == n) {                        // tam atama
3      enIyi = Math.min(enIyi, maliyet);
4      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                        // sınır
▶      kullanildi[j] = true;
▶      ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```

	iş 0	iş 1	iş 2
k0	5	8	4
k1	7	3	6
k2	2	9	5

enİyi = 13

alt = 9

k1 → iş 1: yeni = 7, alt = 9 → dallan.

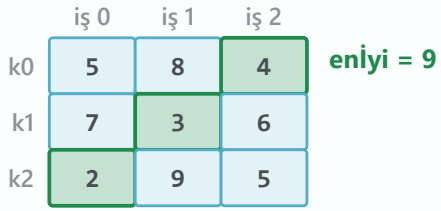
Adım 9 / 10

DEĞİŞKENLER

`i = 1` `j = 1` `yeni = 7` `alt = 9` `enİyi = 13`

ata(0, 0): Kod İzleme

```
1  void ata(int i, int maliyet) {
▶    if (i == n) {                        // tam atama
▶      enIyi = Math.min(enIyi, maliyet);
▶      return;
5    }
▶    for (int j = 0; j < n; j++) {
▶      if (kullanildi[j]) continue;
▶      int yeni = maliyet + c[i][j];
▶      if (yeni + kalanMin[i + 1] >= enIyi)
10       continue;                        // sınır
▶      kullanildi[j] = true;
▶      ata(i + 1, yeni);
13     kullanildi[j] = false;             // geri al
14   }
15 }
```



k2 → iş 0: yeni = 9 → tam atama, enİyi = **9** (4 + 3 + 2). Sınıra eşit olduğu için daha iyisi yoktur. 7 çağrı; budamasız arama 3! = 6 atamanın hepsini (16 çağrı) gezerdi.

Adım 10 / 10

DEĞİŞKENLER

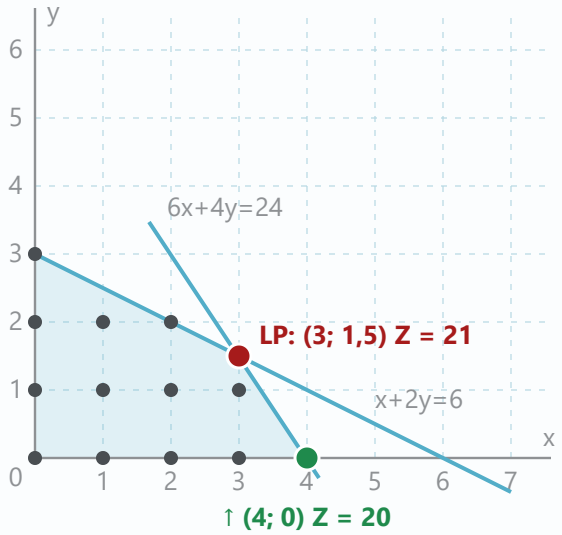
`i = 3` `maliyet = 9` `enİyi = 9`

Tam sayılı Doğrusal Programlama (ILP)

- Doğrusal programın tüm değişkenleri **tam sayı** olmalı. Dal sınırın en yaygın endüstriyel kullanımı (Land ve Doig, 1960).
- **Sınır:** tam sayı koşulunu kaldır, **LP gevşetmesini** çöz. LP optimumu, ILP optimumu için üst sınırdır.
- **Dallanma:** kesirli bir değişken (y = 1,5) için iki alt problem: y ≤ 1 ve y ≥ 2. Aradaki kesirli bölge atılır, hiçbir tam sayı nokta kaybolmaz.

Örnek: en büyükle Z = 5x + 4y; 6x + 4y ≤ 24, x + 2y ≤ 6, x, y ≥ 0 tam sayı.

İkinci örnek: Z = 3x + 2y; x + y ≤ 4, x ≤ 3, y ≤ 2. Kaba kuvvet 0 ≤ x ≤ 3, 0 ≤ y ≤ 2 ızgarasındaki 12 noktanın hepsini dener → (3, 1), Z = 11. Bu örnekte LP gevşetmesi de (3, 1) verir: dal sınır **kökte** biter.



Gölgeli: LP olurlu bölgesi; noktalar: tam sayı çözümler.

ILP: LP Gevşetmeleriyle Dallanma

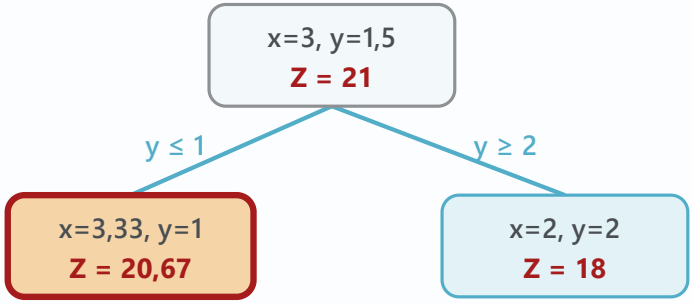
x=3, y=1,5
Z = 21

Kök: tam sayı koşulu gevşetilir (LP gevşetmesi). LP optimumu (3; 1,5), **Z = 21** → üst sınır. y kesirli → y üzerinde dallan.

Adım 1 / 5

LP optimumu (3; 1,5) yuvarlanırsa (3; 1) → Z = 19 ya da (3; 2) → olursuz olur. Gerçek optimum (4; 0) yuvarlamayla bulunamaz.

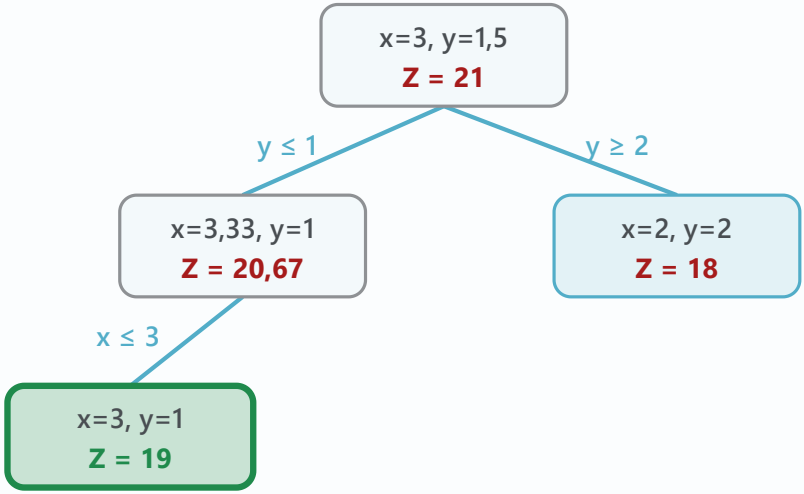
ILP: LP Gevşetmeleriyle Dallanma



Dal y ≤ 1: LP optimumu (3,33; 1), Z = 20,67. x kesirli → x ≤ 3 ve x ≥ 4 dalları açılır.

LP optimumu (3; 1,5) yuvarlanırsa (3; 1) → Z = 19 ya da (3; 2) → olursuz olur. Gerçek optimum (4; 0) yuvarlamayla bulunamaz.

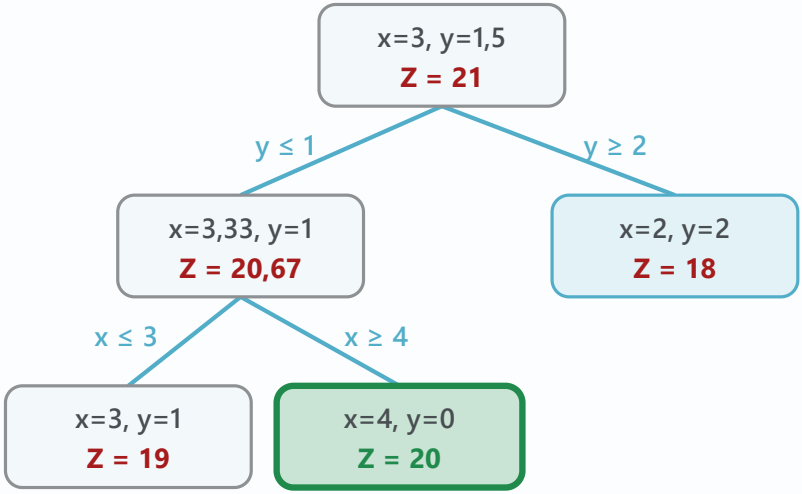
ILP: LP Gevşetmeleriyle Dallanma



Dal x ≤ 3: LP çözümü (3; 1) **tam sayı** → ilk aday çözüm, enİyi = 19.

LP optimumu (3; 1,5) yuvarlanırsa (3; 1) → Z = 19 ya da (3; 2) → olursuz olur. Gerçek optimum (4; 0) yuvarlamayla bulunamaz.

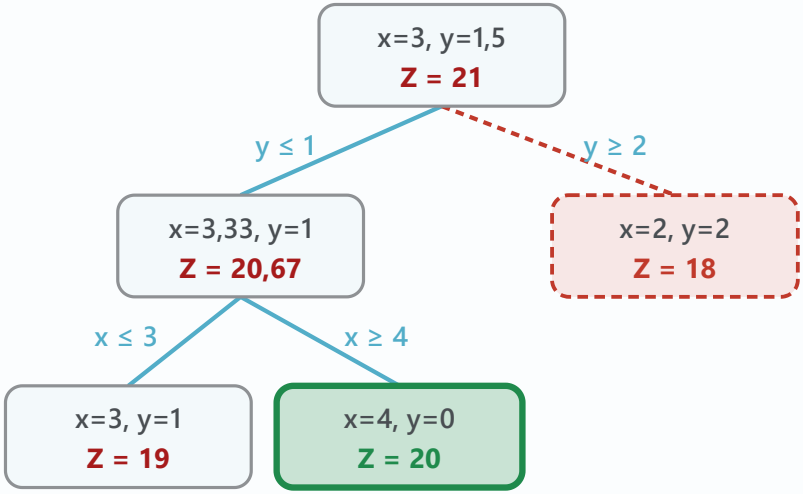
ILP: LP Gevşetmeleriyle Dallanma



Dal x ≥ 4: LP çözümü (4; 0) tam sayı, Z = 20 > 19 → **enİyi = 20**.

LP optimumu (3; 1,5) yuvarlanırsa (3; 1) → Z = 19 ya da (3; 2) → olursuz olur. Gerçek optimum (4; 0) yuvarlamayla bulunamaz.

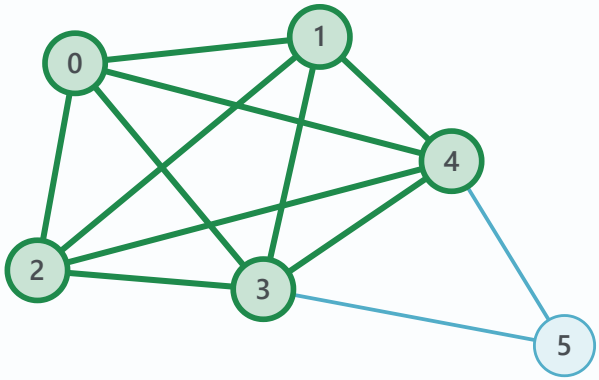
ILP: LP Gevşetmeleriyle Dallanma



Dal y ≥ 2: LP sınırı Z = 18 ≤ 20 → **budandı**. Optimum: x = 4, y = 0, **Z = 20**.

LP optimumu (3; 1,5) yuvarlanırsa (3; 1) → Z = 19 ya da (3; 2) → olursuz olur. Gerçek optimum (4; 0) yuvarlamayla bulunamaz.

Maksimum Klik (Maximum Clique)



6 düğümlü örnek çizge; en büyük klik {0, 1, 2, 3, 4}.

- **Klik:** her iki düğümü komşu olan düğüm kümesi.
- **Sınır:** |K| + |aday| ≤ |enİyi| ise mevcut klik ne kadar büyütülse de enİyi'yi geçemez.

```
void klik(List<Integer> K, int bas) {
  if (K.size() > enIyi.size())
    enIyi = new ArrayList<>(K);
  List<Integer> aday = new ArrayList<>();
  for (int v = bas; v < n; v++)
    if (uygun(v, K)) aday.add(v);  // K'ye komşu
  if (K.size() + aday.size() <= enIyi.size())
    return;                        // sınır
  for (int v : aday) {
    K.add(v);
    klik(K, v + 1);
    K.remove(K.size() - 1);          // geri al
  }
}
```

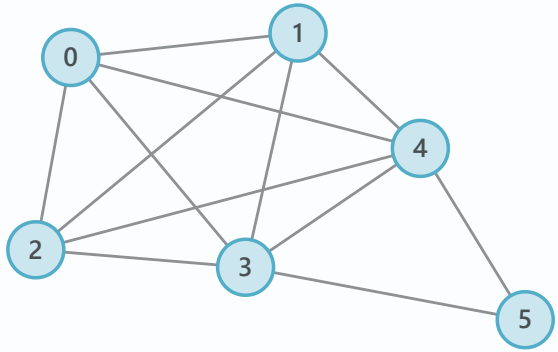
Sürüm	klik çağrısı
Sınırsız geri izleme	36
|K| + |aday| sınırıyla	17

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
 7   if (K.size() + aday.size() <= enIyi.size())
 8     return;                        // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
12     K.remove(K.size() - 1);        // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- `klik({})`
- `main()`



|K| + |aday| = 0 + 6 = 6 ; |enİyi| = 0

Altı düğümlü çizge; K boş, enİyi boş. Adaylar 0–5'in hepsi: 0 + 6 > 0, sınır geçilir; ilk aday 0 eklenir ve derine inilir.

Adım 1 / 10

DEĞİŞKENLER

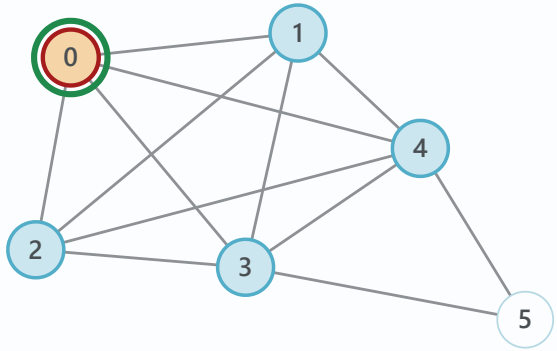
`K = {}` `bas = 0` `aday = {0, 1, 2, 3, 4, 5}` `enIyi = {}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
 7   if (K.size() + aday.size() <= enIyi.size())
 8     return;                        // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
12     K.remove(K.size() - 1);        // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- `klik({0})`
- `klik({})`
- `main()`



|K| + |aday| = 1 + 4 = 5 ; |enİyi| = 1

K = {0} enİyi'den büyük → **enİyi = {0}**. K'nin hepsine komşu adaylar {1, 2, 3, 4}; 1 + 4 > 1 → devam, ilk aday 1 eklenir.

Adım 2 / 10

DEĞİŞKENLER

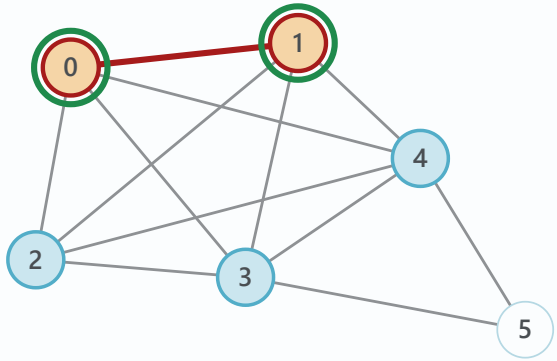
`K = {0}` `bas = 1` `aday = {1, 2, 3, 4}` `enIyi = {0}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
 7   if (K.size() + aday.size() <= enIyi.size())
 8     return;                        // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
12     K.remove(K.size() - 1);        // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- `klik({0, 1})`
- `klik({0})`
- `klik({})`
- `main()`



|K| + |aday| = 2 + 3 = 5 ; |enİyi| = 2

K = {0, 1} enİyi'den büyük → **enİyi = {0, 1}**. K'nin hepsine komşu adaylar {2, 3, 4}; 2 + 3 > 2 → devam, ilk aday 2 eklenir.

Adım 3 / 10

DEĞİŞKENLER

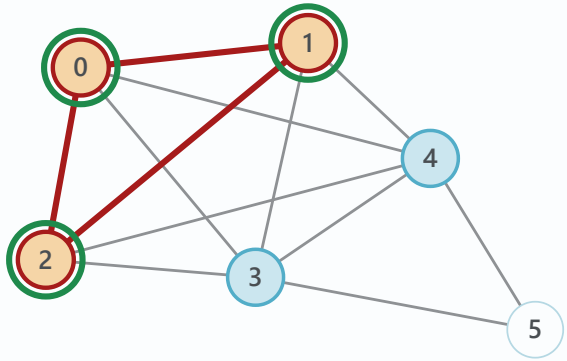
`K = {0, 1}` `bas = 2` `aday = {2, 3, 4}` `enIyi = {0, 1}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
 7   if (K.size() + aday.size() <= enIyi.size())
 8     return;                        // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
12     K.remove(K.size() - 1);        // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- **klik({0, 1, 2})**
- klik({0, 1})
- klik({0})
- klik({})
- main()



|K| + |aday| = 3 + 2 = 5 ; |enİyi| = 3

K = {0, 1, 2} enİyi'den büyük → **enİyi = {0, 1, 2}**. K'nin hepsine komşu adaylar {3, 4}; 3 + 2 > 3 → devam, ilk aday 3 eklenir.

Adım 4 / 10

DEĞİŞKENLER

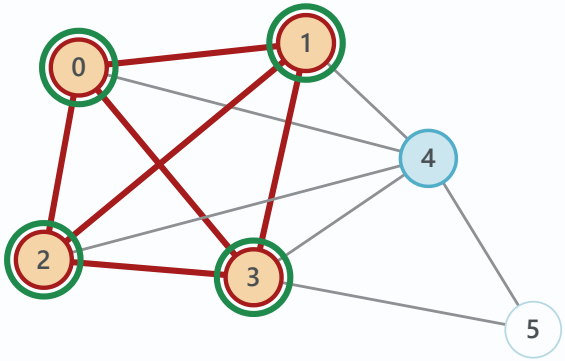
`K = {0, 1, 2}` `bas = 3` `aday = {3, 4}` `enIyi = {0, 1, 2}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
   if (K.size() > enIyi.size())
     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
     if (uygun(v, K)) aday.add(v);  // K'ye komşu
 7   if (K.size() + aday.size() <= enIyi.size())
 8     return;                      // sınır
   for (int v : aday) {
     K.add(v);
     klik(K, v + 1);
12     K.remove(K.size() - 1);      // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- **klik({0, 1, 2, 3})**
- klik({0, 1, 2})
- klik({0, 1})
- klik({0})
- klik({})
- main()



|K| + |aday| = 4 + 1 = 5 ; |enİyi| = 4

K = {0, 1, 2, 3} enİyi'den büyük → **enİyi = {0, 1, 2, 3}**. K'nin hepsine komşu adaylar {4}; 4 + 1 > 4 → devam, ilk aday 4 eklenir.

Adım 5 / 10

DEĞİŞKENLER

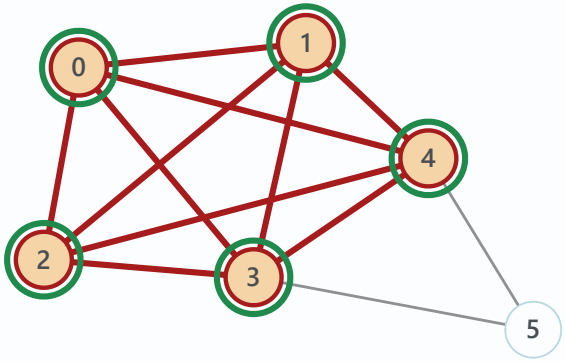
`K = {0, 1, 2, 3}` `bas = 4` `aday = {4}` `enIyi = {0, 1, 2, 3}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
   if (K.size() > enIyi.size())
     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
   if (K.size() + aday.size() <= enIyi.size())
     return;                        // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
12     K.remove(K.size() - 1);        // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- **klik({0, 1, 2, 3, 4})**
- klik({0, 1, 2, 3})
- klik({0, 1, 2})
- klik({0, 1})
- klik({0})
- klik({})
- main()



|K| + |aday| = 5 + 0 = 5 ; |enİyi| = 5

K = {0, 1, 2, 3, 4}: **enİyi = {0, 1, 2, 3, 4}** (5 düğüm). Aday kalmadı: 5 + 0 ≤ 5 → `return`.

Adım 6 / 10

DEĞİŞKENLER

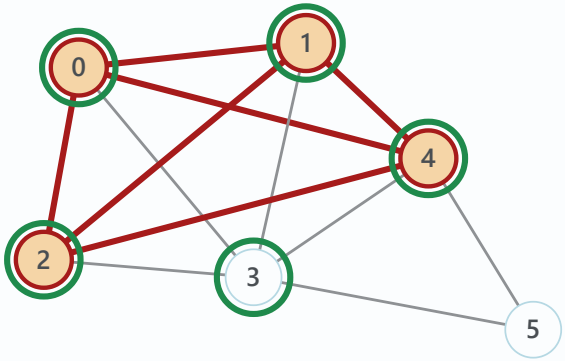
`K = {0, 1, 2, 3, 4}` `bas = 5` `aday = {}` `enIyi = {0, 1, 2, 3, 4}`

klik({}, 0): Sınırlı Arama İzleme

```
1  void klik(List<Integer> K, int bas) {
2    if (K.size() > enIyi.size())
3      enIyi = new ArrayList<>(K);
4    List<Integer> aday = new ArrayList<>();
5    for (int v = bas; v < n; v++)
6      if (uygun(v, K)) aday.add(v);  // K'ye komşu
▶    if (K.size() + aday.size() <= enIyi.size())
▶      return;                       // sınır
9    for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
▶      K.remove(K.size() - 1);       // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- `klik({0, 1, 2, 4})`
- `klik({0, 1, 2})`
- `klik({0, 1})`
- `klik({0})`
- `klik({})`
- `main()`



|K| + |aday| = 4 + 0 = 4 ; |enİyi| = 5

Geri alınır, K = {0, 1, 2, 4} denenir: aday {}, 4 + 0 = 4 ≤ 5 → **sınırla budandı**; bu klik 5'i geçemez.

Adım 7 / 10

DEĞİŞKENLER

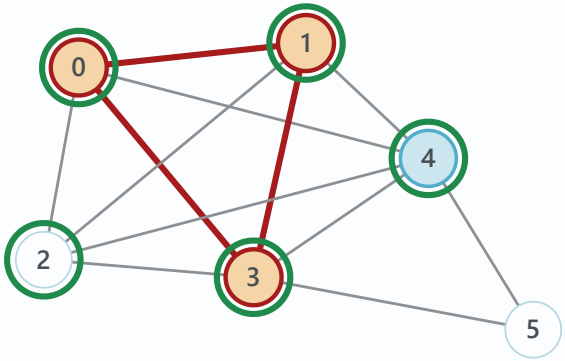
`K = {0, 1, 2, 4}` `bas = 5` `aday = {}` `enIyi = {0, 1, 2, 3, 4}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
 ▶   if (K.size() + aday.size() <= enIyi.size())
 ▶     return;                       // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
 ▶     K.remove(K.size() - 1);       // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- `klik({0, 1, 3})`
- `klik({0, 1})`
- `klik({0})`
- `klik({})`
- `main()`



|K| + |aday| = 3 + 1 = 4 ; |enİyi| = 5

Geri alınır, K = {0, 1, 3} denenir: aday {4}, 3 + 1 = 4 ≤ 5 → **sınırla budandı**; bu klik 5'i geçemez.

Adım 8 / 10

DEĞİŞKENLER

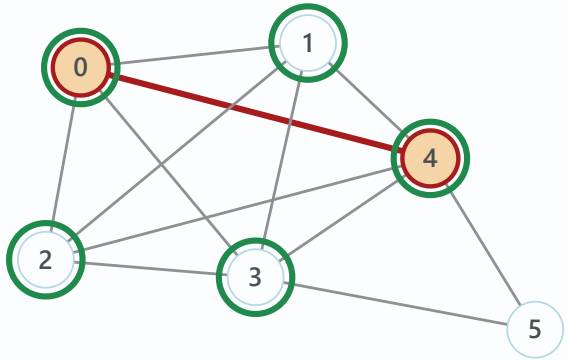
`K = {0, 1, 3}` `bas = 4` `aday = {4}` `enIyi = {0, 1, 2, 3, 4}`

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
▶7   if (K.size() + aday.size() <= enIyi.size())
▶8     return;                       // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
▶12    K.remove(K.size() - 1);       // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- **klik({0, 4})**
- klik({0})
- klik({})
- main()



|K| + |aday| = 2 + 0 = 2 ; |enİyi| = 5

Aynı biçimde {0, 1, 4}: 3 + 0; {0, 2}: 2 + 2; {0, 3}: 2 + 1; {0, 4}: 2 + 0 — hepsi ≤ 5, budandı.

Adım 9 / 10

DEĞİŞKENLER

K = {0, 4} | bas = 5 | aday = {}

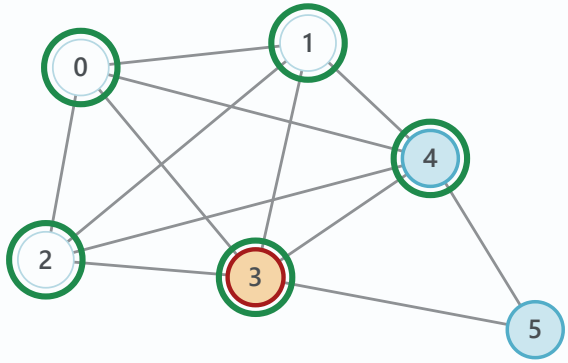
enIyi = {0, 1, 2, 3, 4}

klik({}, 0): Sınırlı Arama İzleme

```
 1 void klik(List<Integer> K, int bas) {
 2   if (K.size() > enIyi.size())
 3     enIyi = new ArrayList<>(K);
 4   List<Integer> aday = new ArrayList<>();
 5   for (int v = bas; v < n; v++)
 6     if (uygun(v, K)) aday.add(v);  // K'ye komşu
▶7   if (K.size() + aday.size() <= enIyi.size())
▶8     return;                       // sınır
 9   for (int v : aday) {
10     K.add(v);
11     klik(K, v + 1);
12     K.remove(K.size() - 1);       // geri al
13   }
14 }
```

ÇAĞRI YIĞINI

- `klik({3})`
- `klik({})`
- `main()`



|K| + |aday| = 1 + 2 = 3 ; |enİyi| = 5

Kökte kalan adaylar 1–5 de tek tek denenir ({1}: 4, {2}: 3, {3}: 3, {4}: 2, {5}: 1); hepsi ≤ 5 → budandı. Sonuç **{0, 1, 2, 3, 4}**, 17 çağrı (sınırsız 36).

Adım 10 / 10

DEĞİŞKENLER

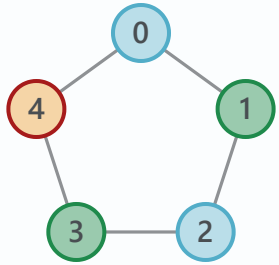
`K = {3}` `bas = 4` `aday = {4, 5}`

`enIyi = {0, 1, 2, 3, 4}`

DİĞER UYGULAMALAR

Klik İçin Daha Sıkı Sınır: Boyama

- Bir klikteki her düğüm çifti komşudur; bu yüzden klikteki düğümlerin hepsi **farklı renk** alır.
- Adaylar k renkle boyanabiliyorsa adaylardan en çok k düğüm kliğe girebilir: **|K| + renk(aday) ≤ |enİyi|** ise budanır.
- Açgözlü boyama O(aday^2) sürer ve en iyi boyama olmasa da geçerli bir üst sınır verir (fazla renk kullanmak sınırı yalnızca gevşetir).
- Örnek: adaylar 5 düğümlü bir halka, K boş, enİyi = 3. Eski sınır 0 + 5 > 3 → dal açılır; renk sınırı 0 + 3 ≤ 3 → **budanır** (halkadaki en büyük klik 2'dir).



3 renk:

{0, 2} {1, 3} {4}

|aday| = 5

renk sınırı = 3

Pratikte en hızlı tam klik algoritmalarının çoğu (Tomita'nın MCQ / MCS ailesi) bu fikre dayanır: adayları renk sırasıyla dener ve renk sınırıyla budar.

DİĞER UYGULAMALAR

Dal Sınırla Çözülen Diğer Problemler

Hamilton yolu

Her düğümü tam bir kez ziyaret eden yol; ağırlıklı sürümü TSP'ye çok benzer sınırlarla çözülür.

Boole sağlanabilirlik (SAT)

Değişkenlere doğruluk değeri atanır; çelişen kısmi atamalar budanır (DPLL). En büyük SAT için sınır: sağlanabilecek en fazla tümce.

Çizelgeleme

En büyük yük (makespan) ya da gecikme en küçüklenir; sınır: kalan işlerin en iyi durumdaki yükü.

Kutu paketleme

Eşyalar en az sayıda kutuya yerleştirilir; alt sınır ⌈Σ boyut / kutu kapasitesi⌉.

Maksimum kapsama

En fazla öğeyi kapsayan k küme seçilir; sınır: kalan en büyük kümelerin ekleyebileceği öğe sayısı.

Atama problemi

n iş, n kişiye en düşük maliyetle atanır; alt sınır: atanmamış her satırın en küçük maliyeti.

Maksimum akış gibi polinom zamanlı algoritması olan problemlerde dal sınıra gerek yoktur; yöntem NP-zor problemler için anlamlıdır.

DİĞER UYGULAMALAR

Dal Sınır, A* ve Derinlik Öncelikli Dal Sınır

Özellik	En iyi önce dal sınır (LC)	A*	Derinlik öncelikli dal sınır
Seçim	en iyi sınır (ĉ)	en küçük f = g + h	yığın / özyineleme
Sınır / tahmin	geçerli (iyimser) sınır	kabul edilebilir h (fazla tahmin etmez)	geçerli sınır + enİyi
Hedef	optimizasyon (en iyi yaprak)	başlangıçtan hedefe en kısa yol	optimizasyon
Bellek	canlı düğümler: üstel olabilir	açık liste: üstel olabilir	O(derinlik)
İlk tam çözüm	genellikle geç ama iyi	bulunan ilk hedef optimumdur	erken (her an bir çözüm var)

- A*, ağırlıklı çizgede yol arayan bir LC dal sınırdır: h(x), x'ten hedefe kalan maliyetin alt sınırıdır (8 bulmacasında yanlış taş sayısı).
- Bellek bittiğinde derinlik öncelikli sürüm ya da IDA* (sınır eşiğini yineleyerek artıran derinlik araması) kullanılır.

DEĞERLENDİRME

Tasarım Paradigmaları: Birikimli Karşılaştırma

Paradigma	Karar	Alt problemler / gezinme	Ne zaman?	Örnek
Kaba kuvvet	tüm adayları üret, sonra test et	tam arama uzayı	uzay çok küçükse	8^8 vezir yerleşimi
Böl ve fethet	sabit bölme kuralı; böl, çöz, birleştir	bağımsız; yukarıdan aşağı	alt problemler örtüşmüyorsa	birleştirme sıralaması, Strassen
Açgözlü	o an en iyi görüneni seç; geri dönme	seçimden sonra tek alt problem	açgözlü seçim özelliği kanıtlıysa	Dijkstra, Huffman, Kruskal
Dinamik programlama	tüm seçenekleri dene, en iyisini sakla	örtüşen; çoğunlukla aşağıdan yukarı	optimal alt yapı, polinom sayıda alt problem	Fibonacci, 0/1 sırt çantası, LCS
Geri izleme	kısmi çözümü genişlet; geçersizse geri al	durum uzayı ağacında DFS	kısıt sağlama, tüm çözümler	n-vezir, Sudoku
Dal sınır	geri izleme + sınırla budama	DFS, BFS ya da en iyi önce	NP-zor eniyileme, kesin sonuç	0/1 sırt çantası, TSP

Soru sırası: alt problemler bağımsız mı? → **böl ve fethet**. Örtüşüyor mu? → **dinamik programlama**. Yerel en iyi seçim küresel en iyiye götürüyor mu? → **açgözlü**. Hiçbiri yoksa ve problem NP-zorsa: kesin sonuç için **geri izleme / dal sınır**, polinom sürede garantili sonuç için **yaklaşım algoritması**.

Özet

Konu	Hatırlanacaklar
Paradigma	durum uzayı ağacı; seç → dallan → sınırla → buda; enİyi en iyi tam çözümü tutar
Sınır	geçerli (iyimser), sıkı, ucuz; gevşetme ile kurulur. En büyüklemede üst, en küçüklemede alt sınır
Budama	olursuzluk, sınır ≤ enİyi (ya da alt ≥ enİyi), tam çözüm / tam sayı LP
Stratejiler	FIFO (BFS, kuyruk), LIFO (DFS, yığın, az bellek), LC / en iyi önce (öncelikli kuyruk, az düğüm)
Sırt çantası	kesirli sırt çantası üst sınırı; örnekte sınır(kök) = 115, optimum 90; 31 yerine 11 düğüm
TSP	alt sınır = maliyet + çıkılmamış şehirlerin en ucuz çıkış kenarları; 4 şehirde 16 → 10 düğüm
ILP / klik	LP gevşetmesi + kesirli değişkende dallanma; klikte |K| + |aday| ≤ |enİyi| sınırı
Karmaşıklık	en kötü durumda üstel; pratik başarı sınırın kalitesine bağlı
Paradigmalar	bağımsız → böl ve fethet; örtüşen → DP; açgözlü seçim → açgözlü; NP-zor kesin → geri izleme / dal sınır

ÖZET

Etkileşimli Simülatörler

Gezgin Satıcı Problemi

En fazla 10 rastgele şehirde hızlı en yakın komşu sezgiseli ile tüm turları deneyen kaba kuvvetin karşılaştırması.

ÖZET

Sık Yapılan Hatalar

Geçersiz (kötümser) sınır

Sırt çantasında yalnızca tam sığan eşyaları toplamak üst sınır değildir; optimumu içeren dal budanabilir.

Yönü karıştırmak

En büyüklemede sinir ≤ enİyi, en küçüklemede alt ≥ enİyi ise budanır. Tersini yazmak ya optimumu kaçırır ya hiç budamaz.

Kuyruktan çıkınca tekrar sınamamak

Düğüm eklendiğinde umut vaat ediyordu; beklerken enİyi büyüdüyse artık gereksizdir (örnekte C ve G).

Eşyaları sıralamamak

Kesirli sınır, eşyalar p_i/w_i'ye göre azalan sırada değilse geçerli bir üst sınır vermez.

enİyi'yi yalnızca yapraklarda güncellemek

Sırt çantasında her olurlu düğüm bir çözümdür; erken güncelleme daha çok budama sağlar.

Geri izleme ile aynı sanmak

Sınır testi olmayan kod (ör. budamasız TSP araması) dal sınır değildir; yalnızca tüm ağacı gezer.

ÖZET

Örnek Problemler

Sırt çantası: (p, w) = (20, 2), (30, 5), (35, 7), (12, 3); W = 9 (oranlar 10, 6, 5, 4).

1. Kökün kesirli üst sınırını hesaplayın.
2. Optimum kazanç nedir, hangi eşyalar alınır?
3. "Eşya 1 alınmadı" düğümünün (0, 0) sınırı nedir? Optimum bulunduktan sonra budanır mı?
4. 4 şehirli TSP örneğinde 0 → 3 → 1 yolunun alt sınırı nedir?
5. Maksimum klikte K = {3, 4}, aday = {5}, |enİyi| = 5 iken ne olur?
6. Belleği çok kısıtlı bir sistemde hangi arama stratejisi seçilmelidir?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. Sırt çantası kodunu FIFO kuyrukla yazın; bölümdeki örnekte kaç düğüm üretilir?
2. `tsp` metoduna en iyi turu kaydeden `int[] enIyiYol` ekleyin.
3. 4 şehirli örneğin kökünde "iki en kısa kenar toplamının yarısı" sınırını hesaplayın.
4. Maliyet matrisi [[9, 2, 7], [6, 4, 3], [5, 8, 1]] olan atama probleminin kök alt sınırı ve optimumu nedir?
5. Kesirli sırt çantası sınırının her zaman geçerli olduğunu kanıtlayın.
6. İkinci ILP örneği ($Z = 3x + 2y$; $x + y \le 4$, $x \le 3$, $y \le 2$) LP gevşetmeli dal sınırla kaç düğümde çözülür?

▶ **Cevaplar / ipuçları**

Kaynaklar

- E. Horowitz, S. Sahni ve S. Rajasekaran, *Computer Algorithms*, 2. baskı, Silicon Press, 2008 (Bölüm 8: Branch-and-Bound).
- R. Neapolitan, *Foundations of Algorithms*, 5. baskı, Jones & Bartlett, 2015 (Bölüm 6).
- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3. baskı, Pearson, 2012 (Bölüm 12.2).
- A. H. Land ve A. G. Doig, "An Automatic Method of Solving Discrete Programming Problems", *Econometrica* 28(3), 1960.
- J. D. C. Little, K. G. Murty, D. W. Sweeney ve C. Karel, "An Algorithm for the Traveling Salesman Problem", *Operations Research* 11(6), 1963.
- R. Carraghan ve P. M. Pardalos, "An Exact Algorithm for the Maximum Clique Problem", *Operations Research Letters* 9(6), 1990.

Son

Bölüm 10: Dal Sınır Algoritmaları

Sonraki bölüm: Bölüm 11: Yaklaşım Algoritmaları · Terim sözlüğü