Algorithm Analysis And Design

With Infinity Teams

Eng: Abdulrahman Hamdi

INFINITY TEAM
Since 2020

Table of contents

Eng: Abdulrahman Hamdi

06

Brute Force and Exhaustive Search

Eng: Abdulrahman Hamdi

Brute Force and Exhaustive Search Çerçevesi

• Selection Sort and Bubble Sort
- Selection Sort
- Bubble Sort

• Sequential Search and Brute-Force String Matching
- Sequential Search
- Brute-Force String Matching

• Exhaustive Search
- Traveling Salesman Problem (TSP)
- Knapsack Problem
- Assignment Problem

• Exercises and Summary
- Exercises for each topic
- Final summary of the content

Eng: Abdulrahman Hamdi

Brute-Force (Kaba Kuvvet) Algoritmaları

Bu derste göreceğimiz ilk algoritma dizaynı Brute-Force algoritmasıdır. Genel olarak bu algoritma türü, eldeki problemi tam ve kesin olarak çözebilmek adına herhangi bir zaman/hafıza kısıtlaması yapmadan olup olabilecek tüm çözüm yollarını dener ve içlerindeki işe yarayan algoritmayı bulur ve çıkartır. Daha genel olarak Brute-Force algoritmalar bir problemde zamanı ve hafızayı dikkate almaksızın aklımıza gelen ilk (çoğunlukla en ilkel) çözümdür.
Bu algoritmalar iki yönden önemlidir.

1-) Eldeki probleme kötü de olsa bir çözüm getirebilmesi (hiç yoktan iyidir)
2-) Eldeki probleme daha sonra geliştirebileceğimiz üst düzey/elit algoritmalarla kıyaslayabileceğimiz bir baz oluşturması. Yani daha sonra bulacağımız yöntemlerin zaman ve/veya hafıza açısından daha iyi olup olmadığını / bir avantaj sağlayıp sağlamadığını elimizdeki brüte force algoritmaya bakarak anlayabiliriz.

Eng: Abdulrahman Hamdi

Brute-Force (Kaba Kuvvet) Algoritmaları

• **Kaba Kuvvet**
– Bir problemi çözmek için en basit yaklaşım
– Genellikle problemin tanımına ve konseptine bağımlıdır
– Genellikle uygulaması en basit çözümdür

• **En temel örnekler**
– a^n hesaplanması
– n! Hesaplanması
– İki matrisin çarpımı
– Bir dizide bir elemanın aranması

• **Örnek:**
Ebob ve Ekok için Brute-Force algoritmalar bulalım. m ve n tam sayıları için → Ebob m veya n den daha büyük olamaz. O halde bunlardan küçüğünü buluruz , birer birer aşağı ineriz taki bulduğumuz sayı hem m ile hem de n ile bölünene kadar.

```
ebobBF(m, n)
// Girdi: m ve n tam sayıları
// Çıktı: m ve n sayılarının ebob'u

    eb ← min(m, n)

    while m mod eb ≠ 0 veya n mod eb ≠ 0:
     eb ← eb - 1

    return eb
```

Brute-Force (Kaba Kuvvet) Algoritmaları

```
ebobBF(m, n)
// Girdi: m ve n tam sayıları
// Çıktı: m ve n sayılarının ebob'u

eb ← min(m, n)

while m mod eb ≠ 0 veya n mod eb ≠ 0:
    eb ← eb - 1

return be
```

Burada veya ile bağlandığından yanlış olabilmesi için her iki tarafının da yanlış olması gerekir. Her iki taraf yanlış olduğunda
m mod eb = 0
n mod eb = 0
olur ki bu durumda zaten eb aradığımız ebob olur.

Eng: Abdulrahman Hamdi

Brute-Force (Kaba Kuvvet) Algoritmaları

```
ekokBF(m, n)
// Girdi: m ve n tam sayıları
// Çıktı: m ve n sayılarının ekok'u

    ek ← max(m, n)

    while ek mod m ≠ 0 veya ek mod n ≠ 0:
    ek ← ek + 1

    return ek
```

EKOK, m veya n'den daha küçük olamaz. Bu nedenle, büyük olan sayıdan başlayarak EKOK'u birer birer artırırız. Bulduğumuz sayı her ikisini de tam böldüğünde döngüden çıkarız. Yani, 'ek mod m = 0' ve 'ek mod n = 0' koşulu sağlandığında, bu değer aradığımız EKOK'tur.

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

Bir diziyi küçükten büyüğe sıralamak için kullanılan temel algoritmalardan olan Selection Sort ve Bubble Sort, Brute-Force algoritmalarına örnektir. Çünkü bu algoritmalar, basit gözlemlere dayanır ve uygulanmaları sırasında herhangi bir zaman kısıtı gözetmez.

• **Ne İşe Yararlar?**
Küçük boyutlu veri setlerinde kolayca uygulanabilir.

Öğretici algoritmalardır, temel sıralama mantığını anlamak için kullanılır.

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

• Seçimli Sıralama (Selection Sort)
Selection sort basitçe , en baştan , dizinin ilk elemanında başlayarak her adımda dizinin kalan kısmındaki en küçük elemanı seçen ve bu elemanı dizinin ilgili yerine koyan sıralama algoritması olarak karşımıza çıkar.

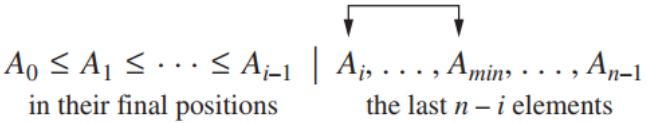
Örnek verecek olursak , 4-3-1-8-5 dizisini ele alalım. Bu dizinin en küçük elemanı 1'dir. O halde 1'i dizinin en başına alalım. Bu durumda dizi 1-3-4-8-5 olur. Bu aşamada birinci pozisyona en küçük elemanı koyduk. Şimdi ikinci pozisyona en küçük elemanı koyalım.

Dizinin 1'den sonraki kısmının en küçük elemanı 3'tür , o da zaten ikinci pozisyondadır , burada herhangi bir şey yapmamıza gerek yoktur. Aynı şey 4 için de geçerlidir. Onun da yeri , olması gereken yer olan üçüncü sıradadır çünkü tüm dizide en küçük 3. elemandır. Dördüncü pozisyonu iyileştirirken dördüncü ve beşinci pozisyonların en küçüğüne bakarız. Bu anlamda 5 en küçüktür. Bu yüzden onu 4. Pozisyona getiririz yani 8 ile yer değiştiririz. Son olarak dizi şu hali alır : 1-3-4-5-8 .
Dizinin adım adım nasıl değiştiğini de görelim:
4-3-1-8-5 → 1-3-4-8-5 → 1-3-4-5-8

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

• **Seçimli Sıralama (Selection Sort) daha özet:**
– Dizinin içerisindeki en küçük eleman bulunur.
– 1. sıradaki elemanla yer değiştirilir.
– En küçük bulma işlemi dizinin ikinci elemanından başlanılarak tekrar edilir.
– Bulunan en küçük değer 2.sıradaki elemanla yer değiştirir.
– Bu işlem dizinin son elemanına kadar devam eder.



n - 1 geçişten sonra liste sıralanmış olur.İşte bu algoritmanın sözde kodu (pseudocode); basitlik açısından, listenin bir dizi olarak uygulandığını varsayar:

```
selectionSort(A[0, ..., n-1])

// Girdi: n uzunluğunda bir A dizisi
// Çıktı: Elemanları küçükten büyüğe sıralanmış A dizisi

for i = 0'dan n-2'ye:
    min_indis ← i  // Varsayalım A[min_indis],
    //i. pozisyonda kendinin sağındakilerin en küçüğü olsun

    for j = i+1'den n-1'e:
        if A[j] < A[min_indis]:
            min_indis ← j

    A[min_indis] ile A[i]'yi yer değiştir.
```

Temel Sıralama Algoritmaları

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]   min ← j
    swap A[i] and A[min]
```

Örnek olarak, algoritmanın 89, 45, 68, 90, 29, 34, 17 listesi üzerindeki işlemi gösterilmektedir.Seçmeli sıralamanın (selection sort) analizi oldukça basittir. Girdi boyutu, eleman sayısı n ile belirlenir; temel işlem, A[j] < A[min] anahtar karşılaştırmasıdır.Bu işlemin kaç kez gerçekleştirileceği yalnızca dizinin boyutuna bağlıdır ve aşağıdaki toplam ile verilir:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

89	45	68	90	29	34	**17**
17	| 45	68	90	**29**	34	89
17	29	| 68	90	45	**34**	89
17	29	34	| 90	**45**	68	89
17	29	34	45	| 90	**68**	89
17	29	34	45	68	| 90	**89**
17	29	34	45	68	89	| 90

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

ALGORITHM *SelectionSort*$(A[0..n-1])$

//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

- **Temel işlemin gerçekleşme sayısı formülü:**

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}. \quad \Theta(n^2)$$

Böylece, seçmeli sıralama (selection sort) tüm girdiler için $\Theta(n^2)$ karmaşıklığına sahip bir algoritmadır. Ancak, anahtar değişimlerinin sayısının yalnızca $\Theta(n)$ olduğunu unutmayın; daha kesin bir ifadeyle, n - 1 (i döngüsünün her tekrarı için bir kez) değişim yapılır.Bu özellik, seçmeli sıralama algoritmasını birçok diğer sıralama algoritmasından olumlu bir şekilde ayırır.

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

• **Kabarcık Sıralama (Bubble Sort)**
Kısaca bubble sort , “sıralanmış bir dizide her eleman bir sonrakinden daha küçük olmak zorundadır” gözlemine dayanır. Bu haliyle bubble sort mantığı ile bir sıralama algoritması yazmak oldukça kolaydır. Alınan bir dizide her eleman bir sonraki elemandan küçük oluncaya kadar algoritmadaki loop döner ve her bir döngüde eğer bir eleman bir sonrakinden büyükse bu iki eleman yer değiştirilir

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \leq \cdots \leq A_{n-1}$$
in their final positions

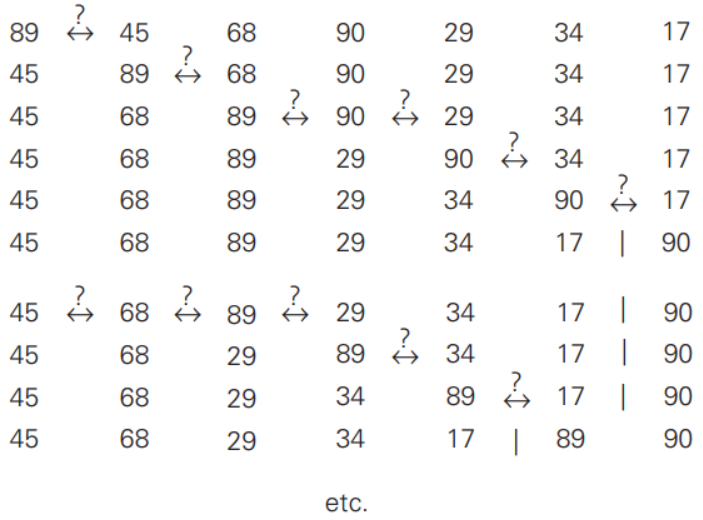
Here is pseudocode of this algorithm.

ALGORITHM *BubbleSort*$(A[0..n-1])$
//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

• **Kabarcık Sıralama (Bubble Sort)**
Algoritmanın 89, 45, 68, 90, 29, 34, 17 listesi üzerindeki işlemi örnek olarak gösterilmiştir.Yukarıda verilen kabarcık sıralaması (bubble sort) sürümü için anahtar karşılaştırmalarının sayısı, boyutu n olan tüm diziler için aynıdır. Bu sayı, seçmeli sıralama (selection sort) için olan toplama neredeyse eşit olan bir toplam ile elde edilir.



Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

• **Kabarcık Sıralama (Bubble Sort)**
Kabarcık sıralama (bubble sort) algoritmasının 89, 45, 68, 90, 29, 34, 17 listesi üzerindeki ilk iki geçişi gösterilmiştir.İki elemanın yer değiştirmesi yapıldıktan sonra yeni bir satır gösterilir. Dikey çubuğun sağındaki elemanlar nihai konumlarına ulaşmış olup, algoritmanın sonraki yinelemelerinde dikkate alınmaz.
• Temel İşlem :?
• Gerçekleşme Sayısı :?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Ancak, anahtar değişimlerinin sayısı girdiye bağlıdır. Azalan dizilerdeki en kötü durumda, anahtar değişimlerinin sayısı, anahtar karşılaştırmalarının sayısıyla aynı olur.

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Eng: Abdulrahman Hamdi

Temel Sıralama Algoritmaları

- **Kabarcık Sıralama (Bubble Sort)**

```
bubbleSort_V1(A[0, ..., n-1])

// Girdi: n uzunluğunda bir A dizisi
// Çıktı: Elemanları küçükten büyüğe sıralanmış A dizisi

while true:
    sıralı_ardışık ← 0
    // Bu, önce küçük sonra büyük olan sıralı ikilileri sayacak

    for i = 0'dan n-2'ye:
        if A[i+1] < A[i]:  // Sıralanmamış ikili tespiti
            temp ← A[i+1]
            A[i+1] ← A[i]
            A[i] ← temp
        else:
            sıralı_ardışık ← sıralı_ardışık + 1

    if sıralı_ardışık == n-1:  // Sıralanmış ise tüm ardışık ikililer
        break  // while'dan çıkmak için

return A
```

Aslında dizinin en başından başlayarak dizinin sonuna kadar ilerleyip eğer bir eleman bir sonrakinden büyük ise bu iki elemanı yer değiştirirsek , dizinin sonuna vardığımızda dizideki en büyük elemanın dizinin sonuna itildiğine şahit oluruz.

5-1-2-4-3 → 1-5-2-4-3 → 1-2-5-4-3 → 1-2-4-5-3 → 1-2-4-3-5 → 1-2-3-4-5

5 dizinin en sonuna geldi. Bu yüzden bir sonraki sefer tekrar her eleman bir sonrakinden küçük mü kontrolü yapmaya başladığımızda bu sefer dizinin en sonuna (n-1 . indise) gitmemize gerek yoktur.

Temel Sıralama Algoritmaları

• **Kabarcık Sıralama (Bubble Sort)**

Sonuç olarak dizinin en son indeksine en büyük elemanı , en sondan bir önceye en büyük ikinciyi , en sondan ikinciye en büyük üçüncüyü … getirerek sıralama yapılır. Bunun için bir for loop baştan sona kadar gider , bu loop dışta yer alıp her iterasyonda sondan kaçıncısının iyileştirilmesine yani o anki en büyüğün o pozisyona yerleştirilmesine karar verir. Başka bir for loop da (iç for loop) ikili kıyaslamaları yapar. Yani ardışık elemanlar arasındaki küçüklük-büyüklük durumunu test eder. Ayrıca bu for loop her başlatıldığında bir önceki gittiğinin bir eksiğine kadar gider, yani bir önceki çalıştırıldığından bir eksik pozisyona kadar ilerler. Çünkü dış for loop ile bir önceki iterasyonda o anki en sonuncu pozisyon iyileştirilmiştir.

```
bubbleSort_V2(A[0, ..., n-1])

// Orjinal bubble sort
// Girdi: n uzunluğunda bir A dizisi
// Çıktı: Elemanları küçükten büyüğe sıralanmış A dizisi

for i = 0'dan n-2'ye:
    for j = 0'dan n-2-i'ye:
        if A[j+1] < A[j]:
            temp ← A[j+1]
            A[j+1] ← A[j]
            A[j] ← temp
```

Şimdi bu algoritmanın zaman verimliliğini hesaplayalım.

i=0 olduğunda içteki loop n-1 kez çalışır

i=1 olduğunda içteki loop n-2 kez çalışır

i=2 olduğunda içteki loop n-3 kez çalışır

.

i=n-2 olduğunda içteki loop 0 kez çalışır

Toplam döngü sayısı = 0+1+.....+n-1 $= \frac{(n-1)*n}{2}$ = t(n)

Bu da aynı zamanda $O(n^2)$ ve $\Omega(n^2)$ 'dir. Dolayısıyla t (n) $\in \Theta(n^2)$ ' dir.

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

• **Sıralı Arama (Sequential Search)**
• Aranan elemanı dizi elemanları ile tek tek karşılaştırarak arama yapar
• Aranan elemanın dizi içerisinde (varsa) bulunduğu ilk indisi verir

• **Örnek:**
Klasik bir BF örneği de bir dizi içerisinde belirli bir değeri aradığımız arama probleminde kullandığımız sıralı arama ya da doğrusal (lineer) arama algoritmasıdır. Bu algoritmada dizi içerisinde en baştan başlayarak tek tek gezilir ve aradığımız değer bulunduğumuz hücrede var mı yok mu diye bakılır. Bu değer bulunduğu zaman diziden çıkılır.

```
sıralıArama(A[0, ..., n-1], k)

// Girdi: n uzunluğundaki A dizisi ve k
    değeri
// Çıktı: k, A'nın içinde ise k'nin olduğu
    indis, değilse -1

indis ← 0

while indis < n ve A[indis] ≠ k:
    indis ← indis + 1

if indis < n:
    return indis
else:
    return -1
```

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

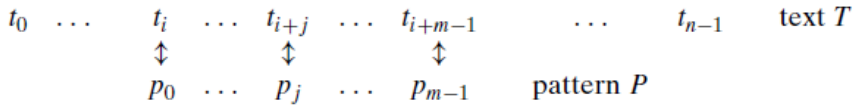
while kısmı “ve” ile bağlanmıştır. Bu yüzden sağı ya da solundaki herhangi bir ifade yanlış olduğunda while’dan çıkılır. Bu durumda indis = n olduğunda veya A[indis] = k olduğunda while’dan çıkılır. Aşağıdaki if , while’dan hangi nedenle çıkıldığını sorgulamak içindir. Eğer k’yi bulduğumuzdan ötürü A dizisinden çıkmışsak (yani A[indis]=k olmasından dolayı) indis n’den küçük kalmıştır. indis = n olmasından dolayı while’dan çıkmışsak k A dizisinde yoktur , o yüzden bu durumda return – 1 olur.

```
sıralıArama(A[0, ..., n-1], k)

// Girdi: n uzunluğundaki A dizisi ve k
    değeri
// Çıktı: k, A'nın içinde ise k'nin olduğu
    indis, değilse -1

indis ← 0

while indis < n ve A[indis] ≠ k:
    indis ← indis + 1

if indis < n:
    return indis
else:
    return -1
```

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

- **Kaba Kuvvet String Eşleme**
- n uzunluğundaki T metni içerisinde m uzunluğundaki P örüntüsünü arama



ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Bilgisayar bilimlerinde en sık karşılaşılan problemlerden biri bir stringin içinde başka bir stringi arama problemidir. Örneğin ismailoğlu stringi içerisinde ‘ailo’ stringini aramak ya da TCCGATTCGG nükleotit dizisi içinde ‘CGA’ stringini aramak bu problemin bir türüdür. Bu probleme string match problemi (string eşleşmesi problemi) denir. Bu problem basitçe , tıpkı bir önceki örnekteki gibi bir dizinin içinde bir değer arama problemi gibi çözülebilir. Yani aranan string boyutu kadar arama yaptığımız string içinde tek tek gezeriz.

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

• **Kaba Kuvvet String Eşleme**

T C C G A T T C G G

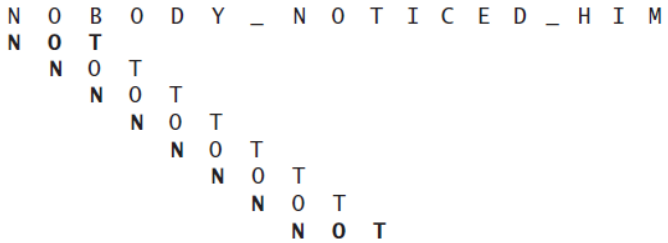
Burada aradığımız 3'lüyü bulmak için en baştan başlayarak 3 er 3 er diziye bakarız. Öncelikle TCC üçlüsünü alırız ve CGA ya eşit mi diye bakarız. Daha sonra CCG üçlüsünü alırız ve CGA ya eşit mi diye bakarız... Bu şekilde ilerleyerek doğru olan eşleşene kadar devam ederiz. Eğer bulamazsak en son olarak CGG üçlüsüne bakarız ve eğer o da doğru değilse döngüden çıkarız.

```
stringMatchBF(A[0, ..., n-1], B[0, ..., m-1])
// Girdi: n uzunluğunda A dizisi, m uzunluğunda B
    dizisi
// Çıktı: B, A'nın içinde ise A'da B'nin başladığı
    indis, değilse -1

    for i = 0'dan n-m'ye:
    j ← 0
    while j < m ve A[i+j] = B[j]:
        j ← j + 1

        if j = m:  // Tüm A[i+j] ile B[j]'ler
    eşleşmiş demektir
        return i  // Eşleşmenin başladığı yer

    return -1  // Eşleşme olmadığında en son -1
    dönülsün
```

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

• **Kaba Kuvvet String Eşleme**



Kötü durum

– Aranan örüntünün bulunamaması

• Kaydırmanın örüntünün son karakteri karşılaştırıldıktan sonra yapılabilmesi

– Her seferinde (n-m+1 kez) 3. karakter karşılaştırmasında uyumsuzluk çıkması

• Bu durumda sınıfı : O(nm)

– Ortalama durum:

• Bir doğal dil için birkaç karşılaştırmada uyumsuzluk yakalanır

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

• **Polinom Hesaplama**

Brute-Force algoritmaları ile ilgili bir başka uygulama alanı polinom hesaplamalarıdır. Burada bir $P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \dots + a_nx^n$ polinomu tanımlı iken yani elimizde $a_0, a_1, a_2, \dots, a_n$ katsayıları varken bir x değeri için amacımız P(x)'i hesaplamaktır. Basitçe P(x) , n+1 adet toplamdan oluşur. Bu yüzden P(x)'i hesaplamak için akla gelen ilk şey bir for loop açıp n+1 adet toplamı toplaya toplaya gitmektir.

```
polinomHesapla(P=[a0 , a1, … , an] , x)
//Girdi: a0 , a1, … , an katsayılarından oluşan P dizisi ; x değeri
//Çıktı: x değeri polinomda yerine koyulduğunda ortaya çıkan toplam
t ← P[0]  // Yani a0 değeri oluyor bu
for i=1'den n'ye
        c ← 1   // x'in üstleri buraya gelecek
    for j=1'den i'ye
            c ← c*x
    t ← t + P[i]*c
return t
```

Eng: Abdulrahman Hamdi

Kaba Kuvvet Arama Algoritmaları

- **Polinom Hesaplama**

  Bu algoritmanın zaman verimliliğini hesaplayalım.

  i=1 için içerdeki loop 1 kez çalışır/döner.

  i=2 için içerdeki loop 2 kez çalışır/döner.

  .

  .

  .

  i=n için içerdeki loop n kez çalışır/döner.

  İç loopun toplam çalışma sayısı $1+2+\ldots+n = \frac{n\cdot(n+1)}{2}$ olup $t(n) = (n^2 + n)\cdot 1/2$ ' dir. O halde t $(n) \in O(n^2)$, t $(n) \in \Omega(n^2)$ ve dolayısıyla t $(n) \in \Theta(n^2)$ ' dir.

Kaba Kuvvet Arama Algoritmaları

- **Polinom Hesaplama**

Yani polinom hesaplama süresi üssel olarak artar. Bu algoritma tam olarak bir Brute-Force örneğidir. Zamanı, işlem fazlalığını dikkate almaksızın direkt sonuca gitmeyi hedefler. Oysa bu algoritma aynı değeri birden fazla kez hesaplar. Örneğin i=4 olduğunda x^4 'ü hesaplamak için x*x*x*x ' i hesaplar. Fakat bir önceki loopta i=3 iken x^3 'ü hesaplamak için x*x*x bulunmuş idi. İşte eğer x^i yi , bir loopu i kez döndürerek hesaplamak yerine , daha önce bulduğumuz x^{i-1} ile x'i çarparak sonuca ulaşacak olursak gereksiz hesaplardan kurtulmuş oluruz.

```
polinomHesaplaEfektif (P=[a0 ,a1,… ,an] , x)
//Girdi: P dizisi , x değeri
//Çıktı: x değerinin polinomda alacağı değer
t ← P[0]
c ← 1
for i=1'den n'ye
     c ← c*x
     t ← t + P[i]*c
return t
```

Bu algoritmanın zaman verimliliği t(n) = n ' dir. O halde bu algoritma θ (n) 'e aittir.

Etraflı Arama Algoritmaları

• Etraflı Arama (Exhaustive Search)
– Kombinasyonel problemlere yönelik kaba kuvvet (brute-force) yaklaşımıdır. Problem alanındaki tüm olası elemanlar üretilir, belirlenen kısıtları sağlayanlar seçilir ve en uygun olanı bulunur. Bu yöntem genellikle optimizasyon problemlerinde kullanılır ve yol uzunluğu, atama maliyeti gibi ölçütleri en iyi şekilde optimize etmeye çalışır. Ancak, bu yaklaşımın uygulanması için belirli kombinatoryal nesneleri üretebilen algoritmalara ihtiyaç duyulur.Exhaustive search.

Özellikle şu üç önemli problemde uygulanır:
Gezgin Satıcı Problemi – Minimum toplam mesafeyi kapsayan rotanın bulunması.
Sırt Çantası Problemi – Ağırlık ve değer kısıtlarına göre en iyi eşyanın seçilmesi.
Atama Problemi – Görevlerin en düşük maliyetle kişilere atanması.

Daha özet:
– Kombinasyonel çözümler içerisinden belli bir özelliğe sahip olanı arama için kullanılan bir kaba kuvvet çözümü
– Her ihtimal denenerek arama yapılır
• Gezgin Satıcı Problemi (Traveling Salesman)
• Sırt Çantası Problemi (Knapsack Problem)
• İşe Alma Problemi (Assignment Problem)

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• **Knapstack (Sırt Çantası) Problemi**
Knapstack problemi , bir hırsızın belirli bir taşıma limiti olan çantasına toplam değeri en fazla olan ürünleri koyması problemidir. Bu durum aslında çok daha genişletilebilir. Genel olarak , belirli bir kısıt altında (bütçe , ağırlık vb. olabilir) , belirli bir sayıdaki ürün içerisinden toplam kârımızı maximize edecek şekilde yaptığımız her seçim aslında bir knapstack problemi olarak düşünülebilir.

Not: Bizim burada dikkate aldığımız knapstack problemi klasik knapstack problemidir. Her ürün ya bir kez alınır ya da hiç alınmaz. Bu 0/1 knapstack problemi olarak da bilinir. Bundan başka çoklu (multiple) knapstack problemi de vardır , burada bir ürün birden fazla kez alınabilir (yatırım hissesi gibi).

Eng: Abdulrahman Hamdi

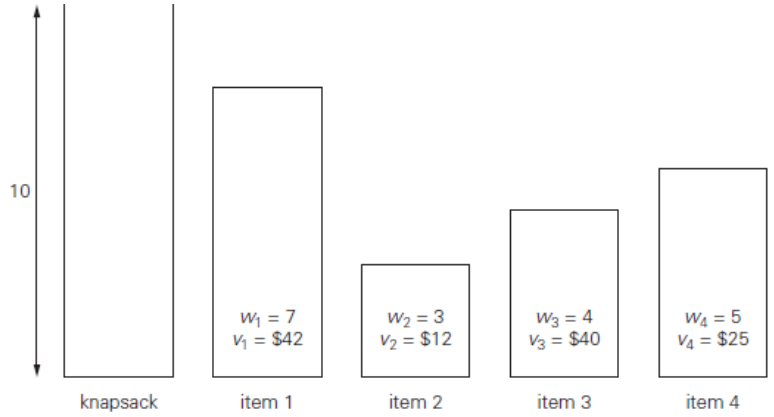
Etraflı Arama Algoritmaları

• **Knapstack (Sırt Çantası) Problemi**

Bir hırsızın çalacağı nesnelere karar verme problemi

– Çantasına sığmalı

– Değeri mümkün olduğunca yüksek olmalı



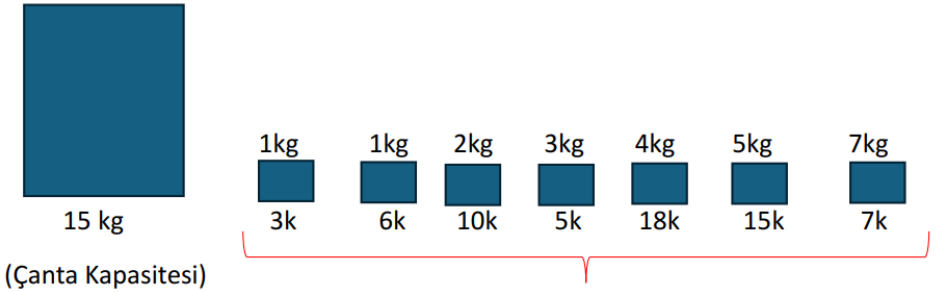
Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• **Knapstack (Sırt Çantası) Problemi**

Diyelim ki elimizde 15 kg'lık bir çanta yani en fazla 15 kg taşıyabilecek bir çanta ve farklı ağırlıklara ve farklı değerlere sahip 7 eşya olsun.

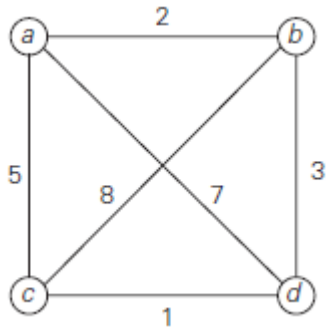


Bu problemin kesin çözümü Brute-Force bir algoritma ile bulunabilir. Bu algoritmada basitçe üstteki eşyaların hepsini içeren bir kümenin var olduğu varsayılır veve bu kümeninher bir alt kümesi alınır , bu alt kümelerden toplam ağırlığı 15'i geçenler atılır. Daha sonra toplam ağırlığı 15 ve altı olan alt kümeler içerisinde toplam değeri en yüksek olan çözüm olur. Fakat buradaki ana problem alt küme sayısının üssel olarak büyümesidir. Hatırlarsak n elemanlı bir kümenin toplam 2n adet alt kümesi olur. Bu yüzden eğer yukarıda anlatılan şekilde bir Brute-Force algoritma oluşturulur ise bu algoritma büyük n değerleri için yani çok fazla eşya olduğu durumlarda çok aşırı yavaş kalır. Bunun yerine bir suboptimal (tam optimal – kesin olmayan) bir algoritma tercih edilir. Örneğin 'Genetik Algoritma' .

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• **Traveling Salesman Problemi (TSP) (Gezgin Satıcı Problemi)**
Bir satıcının – Aralarındaki mesafeler bilinen
şehirleri
• Her şehirden bir kez geçerek başladığı şehre en kısa yoldan dönmesi
• Hamiltonian Circuit
• (n-1)!/2



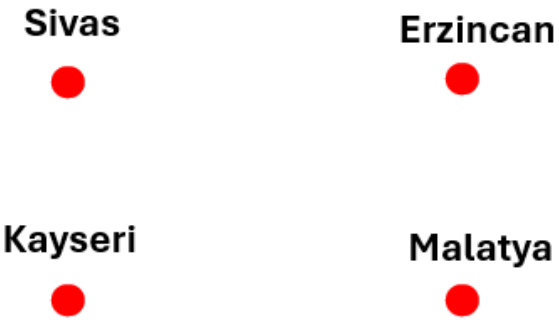
Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• **Traveling Salesman Problemi (TSP) (Gezgin Satıcı Problemi)**

TSP , bir satıcının bulunduğu şehirden (en kısa yolu izleyerek) diğer tüm şehirleri bir kez ziyaret etmesi ve daha sonra bulunduğu şehre tekrar geri dönmesi problemidir. Örnek olması bakımından diyelim ki Sivas'ta bulunan bir satıcı Kayseri , Malatya ve Erzincan şehirlerine uğrayıp , tekrar Sivas'a geri dönmek istiyor. Bu durumda satıcının izleyebileceği en kısa yol ne olur?



Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• Traveling Salesman Problemi (TSP) (Gezgin Satıcı Problemi)

Mümkün olan tüm yollar:

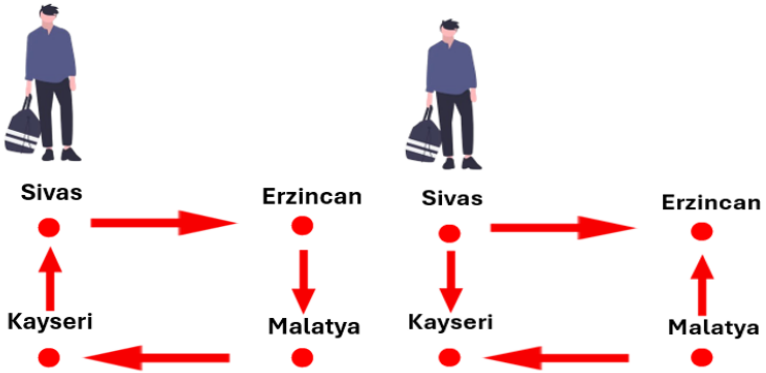
1 -) Sivas – Erzincan – Malatya - Kayseri - Sivas
2 -) Sivas – Kayseri – Malatya – Erzincan – Sivas
3 -) Sivas – Erzincan – Kayseri – Malatya – Sivas
4 -) Sivas – Malatya – Kayseri – Erzincan – Sivas
5 -) Sivas – Malatya – Erzincan – Kayseri – Sivas
6 -) Sivas – Kayseri – Erzincan – Malatya - Sivas

Bu şekilde Erzincan , Malatya ve Kayserinin kendi aralarında farklı sıralanması ile farklı yollar elde edilir. Bu şekilde de olup olabilecek tüm yollar üretilmiş olur. Fakat dikkat edilirse bu yollar birbirinden tamamen farklı değildir. Örneğin Sivas – Erzincan – Malatya – Kayseri – Sivas yolu ile Sivas – Kayseri – Malatya – Erzincan – Sivas yolu aslında birbirinin aynısıdır. Yani toplam yol aynıdır sadece ziyaret ediş sırası farklıdır.

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• **Traveling Salesman Problemi (TSP) (Gezgin Satıcı Problemi)**



İkisinin de toplam uzunluk (yol uzunluğu) aynı

O halde 3. ve 4. yollar ile 5. ve 6. yollar da birbirleriyle aynıdır. Bu yüzden TSP'de üretilebilecek tüm yollar $(toplam\ şehir\ sayısı)!/2$ ' dir. Yine de herhangi bir faktöriyeli ikiye bölmek onu çok fazla küçültmez. Bu yüzden $(toplam\ şehir\ sayısı)!/2$ yine de çok çok büyük bir sayıdır. Yani TSP'nin Brute-Force çözümü şehir sayısı çok iken çok mümkün değildir. Genel olarak optimal çözümü 2^nadet çözümden biri olan knapstack problemi ile (n-1)!/2 adet çözümden biri olan traveling salesman problemi non-polynomial (NP) türde problemlerdir. (Knapstack için n toplam eşya sayısı , TSP için n toplam şehir sayısını verir. Burada bir çıkarmamızın nedeni başlangıç şehrini sabit tuttuğumuz sıralamada permütasyona dahil etmediğimiz içindir.)

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• İşe Alma Problemi (Assignment Problem)
Atama problemi, n kişiyi n işe atarken toplam maliyeti en aza indirmeyi amaçlayan bir optimizasyon problemidir. Her i. kişinin j. işe atanması durumunda ortaya çıkacak maliyet C[i, j] olarak bilinir. Amaç, her satırdan (kişiden) ve her sütundan (işten) bir öğe seçerek toplam maliyeti minimize eden atamayı bulmaktır.Bu problem, doğrudan sezgisel yöntemlerle çözülemeyecek kadar karmaşıktır.

Örneğin, her satırdaki en küçük değeri seçmek geçerli bir çözüm sağlamaz çünkü aynı sütuna denk gelebilirler.Tükeniş arama yöntemi, tüm olası kişi-iş atamalarını (n! permütasyonlarını) oluşturarak, her biri için toplam maliyeti hesaplayıp en düşük olanı seçmeyi gerektirir. Bu yöntem hesaplama açısından pahalı olsa da kesin çözüm sağlar.
Örneğin, verilen maliyet matrisi için ⟨2, 3, 4, 1⟩ seçimi, Kişi 1 → İş 2, Kişi 2 → İş 3, Kişi 3 → İş 4, Kişi 4 → İş 1 şeklinde bir atama anlamına gelir. Tüm permütasyonlar oluşturularak en düşük maliyetli atama belirlenir.

Eng: Abdulrahman Hamdi

Etraflı Arama Algoritmaları

• **İşe Alma Problemi (Assignment Problem)**

n aday n pozisyon için işe başvuruyor

– Her aday her farklı pozisyon için farklı maaş talep ediyor

– Her pozisyona 1 kişi alınacak

– Her aday işe alınacak

– Toplam maaş maliyeti minimum olacak

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.

Algorithm Analysis And Design

Thanks for listening!

Eng: Abdulrahman Hamdi