GebzeYüksek Teknoloji Enstitüsü
Bilgisayar Mühendisliği Bölümü

	Rapor
TM # :	
DERS :	CSE 521
Başlık :	IDO Boğaz Hattı Projesi Optimal İskele Koordinatlarının Belirlenmesi
Anahtar Kelimeler :	Algoritma Tasarımı, Algoritma Analizi, Hırslı Sezgisel, Dinamik Programlama, Parçala Çöz, Kaba Kuvvet, Dallan Sınırla
Yazarlar :	Necmettin Çarkacı
Tarih :	29.04.2015
Yayın Dili :	Türkçe

Özet:

CSE 521 Algoritma Analizi dersi kapsamında verilen IDO Boğaz hattı iskelesi koordinatları için optimal çözümleri verecek beş farklı çözüm yöntemi ile (hırslı sezgisel, dinamik programlama, parçala çöz, kaba kuvvet, dallan sınırla) algoritmaları geliştirilmiş olup, bu algoritmaların uygulaması yapılarak farklı büyüklükteki veri setleri ile teorik ve pratik çalışma zamanı sonuçları analiz edilmiştir.

Anahtar Kelimeler

Algoritma Tasarımı, Algoritma Analizi, Hırslı Sezgisel, Dinamik Programlama, Parçala Çöz, Kaba Kuvvet, Dallan Sınırla

---

Abstract:

Keywords

Remote Eye Gaze Estimation, Appearance-Based Methods, Feature-Based Methods, Feature Extraction, Non-Linear Regression, Synthetic Image.

IDO Boğaz Hattı Projesi Optimal İskele Koordinatlarının Belirlenmesi

[Necmettin ÇARKACI]
[ncarkaci@gyte.edu.tr][1]

1. Sunuş / Introduction

CSE 521 Algoritma Analizi dersi kapsamında verilen IDO Boğaz hattı iskelesi koordinatları için optimal çözümleri verecek beş farklı çözüm yöntemi ile (hırslı sezgisel, dinamik programlama, parçala çöz, kaba kuvvet, dallan sınırla) algoritmaları geliştirilmiş olup, bu algoritmaların uygulaması yapılarak farklı büyüklükteki veri setleri ile teorik ve pratik çalışma zamanı sonuçları analiz edilmiştir.

Çalışmamızın sonraki aşamaları şu adımlardan oluşacaktır; problemin tanımlanacağı Bölüm 2 Problem Tanımı, çözüm yöntemlerinin ve geliştirilen algoritmaların açıklandığı teorik analizlerinin yapıldığı Bölüm 3 Çözüm Yöntemleri, deneysel analiz sonuçlarının açılandığı Bölüm 4 Deneysel Analiz ve Bölüm 5 Sonuç kısımlarından oluşacaktır.

2. Problem Tanımı/Problem Definition

IDO İstanbul boğazının Avrupa yakasına Beşiktaş'tan başlamak üzere kuzeye doğru bir dizi iskele kurmak istemektedir. Yaptığı araştırma sonucunda n farklı olası yer tespit etmiştir. Bu olası yerlerin Beşiktaş'a uzaklığı (artan sırada) $m_1, m_2, \ldots, m_n$ şeklindedir. Her bir konum için olası yolcu bilgileri analiz edilerek bir yıllık olası kazançlar $p_1, p_2, \ldots, p_n$ şeklinde hesaplanmıştır. Çok yakın yerlere iskele kurmamak adına, K belirli bir sayı olmak üzere, her iki iskele arasındaki mesafe en az K kadar olmak zorundadır.

Bu veriler ve kısıtlar altında beklenen IDO'nun bu proje kapsamında elde edebileceği kazancın en az fazla olacağı iskele konumlarının bulunmasıdır.

3. Çözüm Yöntemleri / Solution Techniques

Bölüm 2'de tanımlanan problem için problem çözümlerinde sık şekilde kullanılan beş adet (hırslı sezgisel, dinamik programlama, parçala çöz, kaba kuvvet, dallan sınırla) çözüm yöntemini kullanarak optimal çözümü oluşturacak algoritmalar oluşturulmuştur. Oluşturulan algoritmalar ve bu algoritmalara ait sözde kodlar ile çalışma zamanı analizleri aşağıdaki gibidir.

Çalışmamız kapsamında oluşturduğumuz algoritmalar için girdi olarak Tablo 1'de açıklanan değişkenler kullanılmış olup, algoritmaların çıktıları maksimum kazanç bilgisi ve bu kazancı sağlayacak iskele lokasyonlarını içerir listedir.

Tablo 1 Algoritmalar İçin Girdi Parametreleri

Değişken	Tip	Açıklama
`distanceList[1...n]`	Dizi	Problem tanımında açıklanan, iskelelerin Beşiktaş'a uzaklıklarını içerir sıralı liste
`profitList[1...n]`	Dizi	İskeleler için yıllık olası kazanç listesi
`limit`	Integer	Problem tanımında K olarak tanımlanan, seçilecek iskeler arasındaki en küçük mesafe
`rootStationIndex`	Integer	Başlangıç iskelesinin listeler içerisindeki indeksi

3.1 Hırslı Sezgisel Çözüm (Greedy Solution) :

3.1.1 Hırslı Sezgisel Çözüm Yaklaşımı

```
Greedy Algorithm ( a [ 1 .. N ] )
{
        solution = Ø

        for i = 1 to n
                x = select (a)
                is feasible ( solution, x )
                        solution = solution U {x}
        return solution
}
```

Algortima 1 Hırslı sezgisel algoritma genel yaklaşım[1]

Hırslı sezgisel algoritmalarda Algoritma 1'de tanımlanan genel çalışma prensibinde de görüleceği üzere göz önünde tutulması gereken üç durum mevcuttu ilk olarak seçilen durumun yerel en iyi seçim olması (local optimal) ve bu seçimin mümkün olması (feasible) ve yapılan seçimin daha sonra değiştirilmemesi (irrevocable)'dir.[1] Çalışmamızda karları büyükten küçüğe doğru sıralayarak karar anında algoritmanın kendisi için en iyi tercih olan en yüksek kar getiren durağa gitmesini sağlayan bir yol izledik. Bu yolu izlerken bu iskeleye gidilmesi için kısıt oluşturan iki iskele arası uzaklık şartının verilen mesafeden kısa olmama şartını algoritmamızın mümkün durum (feasibility) kontrolü için kullandık. Algoritma içerisinde verilen kararlar daha sonra değişime uğratılmayacağı bir yöntemle algoritma geliştirildi.

3.1.2 Hırslı Sezgisel Çözüm Sözde Kod

Hırslı sezgisel çözüm yöntemini kullanılarak problemin çözümü için geliştirdiğimiz sözde kod Algoritma 2'deki gibidir.

Hırslı Sezgisel Algoritma

```
// Tanım: En çok kar getirecek iskeleleri ve kazanç miktarı için hızlı sezgisel yaklaşım yöntemini kullanan çözüm sunar
// Girdi: Array DistanceList[l..n], ProfitList[l..n], Integer Limit, Integer Başlangıç iskelesinin indeksi
// Çıktı: En kazançlı iskele noktalarının indekslerini veren array ve toplam kazanç miktarı

1   define procedure greedySolution(distanceList[l..n], profitList[l..n], limit, sourceStationIndex):

      #initialize variables
2     stationIndexList   = []
3     stationSolution    = []
4     maxProfitSolution  = 0

      # Timsort worstcase θ(nlogn), avarage case θ(nlogn), bestcase θ(n)
5     sortedProfitList  =  sorted(profitList, reverse=True)

      # find and create station index list after sorting
6     for station in sortedProfitList:
7           stationIndexList.append(profitList.index(station))
8     end for

      # all profit from 1 to n
9     for stationIndex in stationIndexList:

            # feasibility check
10          if ((distanceList[stationIndex]- distanceList[sourceStationIndex]) > limit):

                  # is a feasibe add to solution set
11                stationSolution.append(stationIndex)
12                maxProfitSolution = maxProfitSolution+profitList[stationIndex]

13                sourceStationIndex = stationIndex
14          end if
15    end for

16    return maxProfitSolution, stationSolution
17  end procedure
```

Algoritma 2 Hırslı Sezgisel Algoritma

3.1.3 Hırslı Sezgisel Çözüm Açıklama

Algoritmanın 2-4 adımlarda algoritma kapsamında kullanılacak değişkenler için başlangıç değerleri atanmıştır. Bu değerler 0 ve boş listededir. Hırslı sezgisel algoritma çözüm yöntemi olarak iskeleler karı en yüksek olandan itibaren sıralandı. Sıralama işlemi 5. adımda yapıldı. Sıralama sonrası iskelelerin yerleri değiştiği için iskelelerin sıralı listedeki

yeni konumlarını 6-8 adımlardaki döngü ile bir listeye atandı. Daha sonra 9- 15 adımları arasında oluşturulmuş yeni listedeki iskele sıralamasına göre her bir iskele için uzaklık kontrolü yapılarak eğer uygunsa optimal iskele listesine eklendi. Uzaklığın uygun olup olmadığı (feasibility check) 10. adımda kontrol edildi. Mümkün sonuçlar için 11 adımda iskele optimal iskele listesine eklendi ve kazanç miktarına 12. adımda yeni iskele noktasını karı eklendi (adding to solution set). İterasyon sonlandığında elde edilen sonuç hırslı sezgisel algoritmanın optimal sonucu olarak çıktı olarak verildi.

Özetle hırslı sezgisel algoritma çözüm yöntemi olarak iskeleler karı en yüksek olandan itibaren sıralandı. ilk olarak en yüksek karı getiren iskeleye ulaşıldı daha sonra bu iskeleden problem kısıtları gözönüne alınarak en yüksek karı getirecek diğer iskelelere ulaşılarak problem için bir çözüme ulaşıldı.

3.1.4 Hırslı Sezgisel Çözüm Teorik Analiz

Algoritmamız içerisinde sıralama işlemi (5. adım) için en kötü çalışma zamanı θ(nlogn), en iyi çalışma zamanı θ(n) ve ortalama çalışma zamanı θ(nlogn) olan python ve java programlama dillerinde varsayalı olarak kullanılan "Tim Sort"[2] algoritması kullanılmıştır. Sıralama sonrası iskelelerin sıralı listedeki konumunu elde edilmesi işlemi (6-8 adımlar arası) linear zamanda gerçekleşmiş olup, algoritmanın hırslı sezgisel çözüm bölümü (9-15 adımlar arası) linear zamanda gerçekleşmektedir. Değişkenlerin başlatılması (4-6. satırlar arası) ve sonucun döndürülmesi (17. satır) işlemleri sabit zamanda (constant time) gerçekleşmektedir.

Algoritmamızın çalışma zamanı;

T(n) = sortingTime + findIndexTime + greedySolutionTime + initialize Variables

T(n) = θ(nlogn) + θ(n) + θ(n) + θ(1)

T(n) = θ(nlogn)'dir.

3.1.5 Hırslı Sezgisel Çözüm Optimallık Kontrolü

Hırslı sezgisel çözüm yöntemi ile tasarladığımız algoritma bütün durumlar için optimal sonuç vermemektedir. Optimal sonucu vermediği durumlardan birisi aşağıdaki örnekte verilmiştir.

Şekil 1 İskeler için kazanç durumlarını gösterir örnek

Kazançları yukarıda şekil 1'de 400, 300 ve 500 olarak sıralanmış bir iskele kazanç durumunda hırslı sezgisel algoritma ilk olarak en fazla kazanç sağlayacağı son istasyon olan 500 kar getiren istasyonu seçecektir. Bu seçim yerel olarak optimal ve mümkün bir

seçimdir. Bununla birlikte 400 ve 300 adımlarına ulaşamayacaktır. 400 ve 300 adımlarına ulaşacak çözüm bizim hırslı sezgisel algoritmamızdan daha optimal sonuç üretebilecektir.

3.2 Dinamik Programlama (Dynamic Programming) :

Dinamik programlama yönteminde amacımız problem çözümü esnasında çözüm ağacında aynı parçaların tekrarlı şekilde çözümünü engelleyecek bir yöntem geliştirmektir. Bu durum genellikle algoritma geliştirme esnasında fazladan depolama alanı kullanılarak başarılabilmektedir. Çalışmamız da benzer yöntem izlenmiştir.

3.2.1 Dinamik Programlama Çözüm Yaklaşımı

İlk iskeleden başlayarak her bir iskele için o iskeleye ulaşılabilen durumların kazanç durumları hesaplandı ve en yüksek kazanç durumu o iskele için ayrı bir kazanç dizisine yazıldı. Aynı zamanda bu en yüksek kazancı getiren iskele lokasyonu da ayrı bir iskele lokasyon dizisinde tutuldu. Böylece ileride ki iskeleler için kazanç hesaplaması yapılırken bu iskeleye ulaşım sağlayan iskelelerin daha önce hesaplanıp diziye kaydettiğimiz kazanç verilerini kullanarak, tekrarlı şekilde aynı değerlerin hesaplanması önlenmiş oldu.

Problemimizin dinamik programlama yaklaşımı ile çözüm basamakları aşağıdaki gibidir.

a) **Karar dizisi (Desicion Sequence) :** Problemin çözümünde kullandığımız karar dizisi (decision sequence) probleme girdi olarak verilen iskelelerin her biridir. Aynı zamanda bu bize problemin çözümünde kullanacağımız alt problem sayısını da vermektedir.

b) **Karar (Decisions) :** Karar dizisindeki her bir iskele için verilecek karar o iskelenin optimal kazancı verecek iskele listesine eklenip eklenmeyeceğidir.

c) **Problemin sayısallaştırılması (Problem Enumaration) :**

Tanım : A(i, j) : i iskelesi ile j iskelesi arasında elde edilecek kazancı gösteren fonksiyon olsun. Öyle ki iskele sayısının n olduğu durumda $0 \leq i \leq j \leq n$

d) **Problemin özyineleme foknsiyonu :**

N adet iskelenin olduğu bir problemde başlangıç iskelesinden, k'nın k= 1…n olarak tanımlandığı durumda k. iskeleye ulaştıracak en kazançlı iskele listesini verecek fonksiyonumuz T (0,k) olarak tanımlayalım.

O halde;
Amaç : T(0,k), $0 \leq k \leq n$ 'yı bulmak.
Özyineleme Fonksiyonu :

$$T(0,k) = \begin{cases} 0 & k = 0 \\ \max_{0 \leq i \leq k} \{ \quad T(0,i) + A(i,k) \quad \} & Otherwise \end{cases}$$

3.2.2 Dinamik Programlama Sözde Kod

Dinamik Algoritma

```
// Tanım: En çok kar getirecek iskeleleri ve kazanç miktarı için dinamik programlama yöntemini kullanan çözüm sunar
// Girdi: Array DistanceList[l..n], ProfitList[l..n], Integer Limit, Integer Başlangıç iskelesinin indeksi
// Çıktı: En kazançlı iskele noktalarının indekslerini veren array ve toplam kazanç miktarı

        define procedure  dynamicProgramming(distanceList[l..n],profitList[l..n], limit,
1       sourceStationIndex):

           # initialize variables
2          profitTable = []
3          pathTable   = []
4          stationList  = []

           # fill result table with profit values
5          for index in range(0,len(profitList)):
6             pathTable.insert(index, 0)
7             profitTable.insert(index,0)
8          end for

9          for currentNode to n:

10            for previousNode = currentNode decrease to -1): # decreasing range

11               if((distanceList[currentNode]-distanceList[previousNode]) > limit):

                    if ( (profitTable[previousNode]+profitList[currentNode]) >
12      profitTable[currentNode]):

                        # update profit value
                        profitTable[currentNode] =
13      profitTable[previousNode]+profitList[currentNode]

                        # update station path
14                      if (profitTable[previousNode] != 0):
15                         pathTable[currentNode] = previousNode
16                      else :
17                         pathTable[currentNode] = 0
18                      end if
19                   end if
20                end if
21             end for
22          end for
```

```
        # find max profit value
23      maxProfit = max(profitTable)

        # find station list from pathTable
24      stationIndex = profitTable.index(max(profitTable))
25      while(stationIndex!=0):
26          stationList.insert(0,stationIndex)

            # next station index
27          stationIndex = pathTable[stationIndex]
28        end while

29     return maxProfit, stationList

30   end procedure
```

Algoritma 3 Dinamik Programlama Algoritması

3.2.3 Dinamik Programlama Çözüm Açıklama

Dinamik programlama yöntemi kullanılarak oluşturulan Algoritma 3'de sözde kodu verilen yöntemde; 2-4 adımda harici depolama alanı olarak kullanılan, verilerin tutulacağı dinamik programlama tabloları oluşturulup başlangıç değerleri tanımlanmıştır. 9-22 adımlar arasında her bir iskele için kazanç değeri hesaplanmıştır. 23. adımda elde edilen kazanç değerlerinden maksimum olanı optimal kazanç değeri olarak elde edilmiştir. 24 – 28 adımlar arasında en yüksek kazanç değerine ait iskele lokasyonları 15 ve 17. adımlar arasında bulunarak dinamik programla yönteminde harici olarak tutulan tabloya eklenen değerlerle oluşturulan iskele tablosundan elde edilmiştir. 10 -21. adımlar arasında her bir iskele için o iskeleye uzaklığı limit olarak belirlenen değerden daha uzak olan iskelelerin (11. adım) kazanç değeri kullanılarak yeni kazanç değerleri elde edilmiş olup elde edilen kazanç değeri iskelenin kazanç tablosuna yazılmıştır. Kazanç tablosu eğer yeni durumda daha kazançlı bir iskele sıralaması varsa güncellenmiştir. Bu durum 12-19 adımlar arasında kontrol edilmiştir.

3.2.4 Dinamik Programlama Çözüm Teorik Analiz

Geliştirilen algoritmanın 4-6 satırları arasında değişkenlerin oluşturulması ve başlatılması ile 11,12,13,14,15,16,17,23,24 satırlarındaki atama ve şart kontrol işlemleri sabit zamanda (constant time) zaman harcamaktadır. Bu işlemler dışında 9-22 adımlar arasında çalışan iterasyon 10-21 satırları arasında çalışan ikinci bir iterasyon içermektedir. İçiçe bu iki iterasyon kazanç ve lokasyon tablolarını doldurmaktadır. Bu iki iterasyonun değeri n uzunlukla girdi içeren Algoritma 3'de tanımlanan bir algoritma için n^2'dir. Bunlara ek olarak geliştirilen algoritma kazanç tablosu için n uzunluğunda, lokasyon tablosu için n uzunluğunda olmak üzere iki adet n uzunluğunda fazladan depolama alanı kullanmıştır.

Fazladan kullanılan depolama alanı = $\underset{(n)}{\text{Kazanç Tablosu}} + \underset{(n)}{\text{İskele Listesi}} = 2n$'dir.

Çalışma zamanı = İç içe iterasyon (Lokasyon Tablosunun ve Kazanç Tablosunun Doldurulması) + Atama ve Şart işlemleri

$$T(n) = \theta(1) + \sum_{i=1}^{n} i \in \theta(n^2)$$

$T(n)_{worst} = \theta(n^2)$

Optimal kazanç listesi n'den daha az iskele kullanılarak oluşturulduğunda yada bazı iskelelere sınırlamalardan dolayı ulaşılamadığında çalışma zamanı o(n) olabilmekte. Bu nedenle $T(n) = O(n^2)$ 'dir.

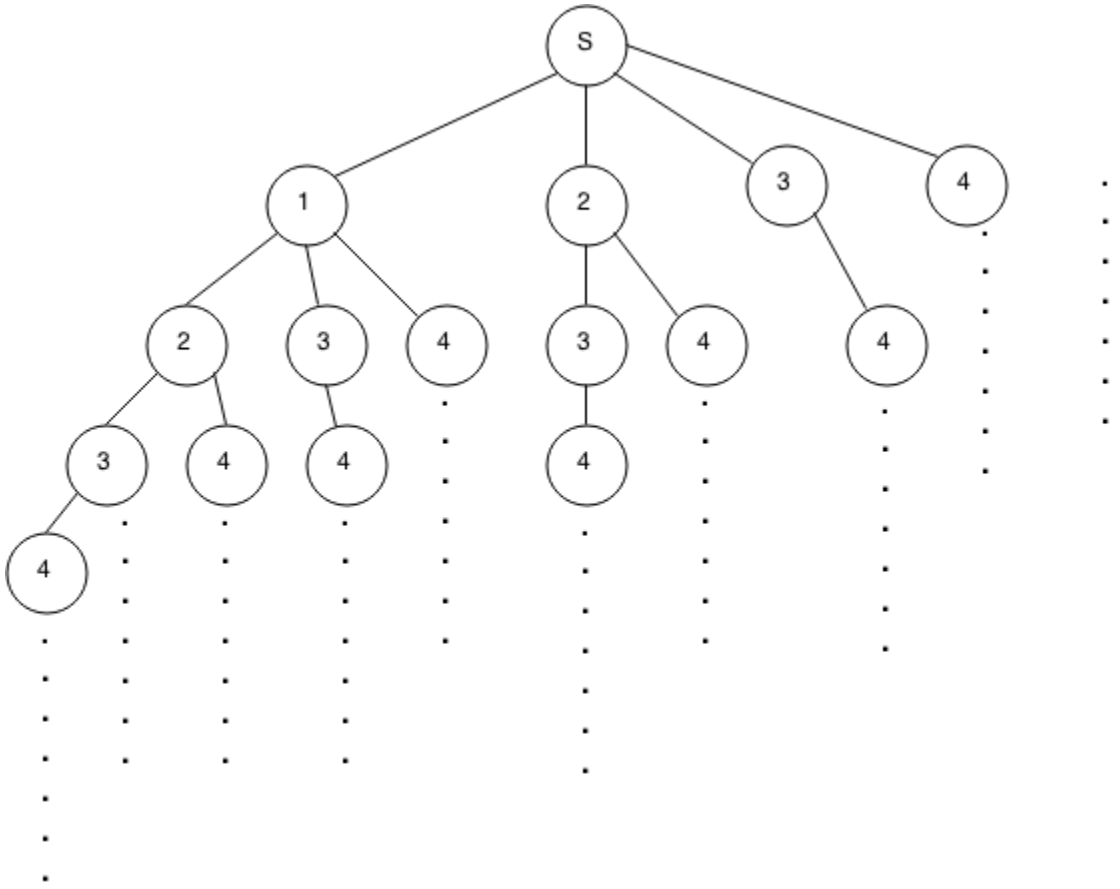
2.3 Parçala Çöz (Divide and Conqure) :

Problem yapısı itibariyle parçala çöz metodu ile çözülememektedir. Problem problem boyutu azaltılarak alt problemlere ayrılamamaktadır. Bununla birlikte problem parçala azalt (divide and decrease) yöntemi ile çözülebilmektedir. Bu yönteme uygun algoritma bölüm 3.4 kaba kuvvet bölümünde açıklanmıştır. Parçala çöz algoritması ile çözülememesinin nedeni problemin aynı yapıda daha küçük parçalara bölünememesidir. Problem parlara ayırıldığı durumlarda her bir parça öteki parçadaki değerlere ihtiyaç duymaktadır.

3.4 Kaba Kuvvet (Brute Force) :

Kaba kuvvet çözüm yaklaşımı problem ağacındaki her bir durumun kontrol edilmesine dayanır. Çalışmamız kapsamında yinelemeli olarak çalışan problem için her bir durumu kontrol eden kaba kuvvet yaklaşımına dayanan algoritma geliştirilmiştir.

3.4.1 Kaba Kuvvet Çözüm Yaklaşımı



Şekil 2 Kaba kuvvet algoritması çalışma ağacı

Şekil 2'de de görüleceği üzere elimizde n adet iskele bulunmakta bu iskeleler için her bir durumu deneyerek en büyük karı getirecek algoritma tablo'da geliştirilmiştir. Bu algoritma ilk olarak n adet karar noktasının her biri için problem kısıtını sağladığı durumlarda yinelemeli şekilde kendisini çağırarak en büyük karı getirecek iskele sıralamasını buluyor.

3.4.2 Kaba Kuvvet Çözümü Sözde Kod

Kaba Kuvvet Algoritması

// Tanım: En çok kar getirecek iskeleleri ve kazanç miktarı için kaba kuvvet yöntemini kullanan çözüm sunar
// Girdi: Array DistanceList[l..n], ProfitList[l..n], Integer Limit, Integer Başlangıç iskelesinin indeksi
// Çıktı: En kazançlı iskele noktalarının indekslerini veren array ve toplam kazanç miktarı

```
1   Define procedure bruteForce(distanceList, profitList, limit,
    rootStationIndex):

    # initialize variables
2   maxProfit    = 0
3   stationList  = []

    # for first visit list go to n item
4   for currentStationIndex = rootStationIndex+1 to n:

       # initialize current profit and station in every step
5      newProfit          = 0
6      newStationList     = []

       # feasibility check
7      if((distanceList[currentStationIndex]-
       distanceList[rootStationIndex]) > limit ):

          # call function for new sub list
8         subProfitValue,subStationList   =   bruteForce(distanceList,
          profitList, limit, currentStationIndex)

          # calculate new profit and new station list from sub routine
9         newProfit = profitList[currentStationIndex]+subProfitValue
10        newStationList.append(currentStationIndex)
11        newStationList = newStationList+subStationList
12
          # check new profit and update max profit and station list
13        if (newProfit > maxProfit):
14           maxProfit = newProfit
15           stationList = newStationList
16        End if
17     End if
18  End for

19  return maxProfit , stationList
20  End procedure
```

Algoritma 4 Kaba Kuvvet Algoritması

3.4.3 Kaba Kuvvet Çözüm Açıklama

Algoritmamızın 2-3 adımların değişkenler tanımlanmış ve başlangıç değerleri atanmıştır.4-18 adımları arasında yer alan iterasyonda problem için girdi olarak belirlenen her bir iskele için olası tüm bağlantılar denenerek bir maksimum kazanç değeri belirlenmekte. Bu maksimum kazanç değerlerinin en büyüğü 19 adımda problem sonucu olarak döndürülmektedir. 7. adımda eğer iskeleye ulaşılabilme durumu kontrol edilmekte, ulaşılabilir olduğu durumda fonksiyon o iskele için çağrılarak (8. adım) o adımdaki iskele için tüm olası durumlar hesaplanmaktadır. Algoritmanın çalışma şeması Şekil 2'de görüldüğü gibi bir çalışma ağacı oluşturmaktadır.

3.4.4 Kaba Kuvvet Çözümü Teorik Analiz

Geliştirilen algoritma özyinelemeli şekilde kendini çağırarak problem için öngörülen her bir durum için çalışmaktadır. Algoritmamız 4-18 adımları arasında problem uzunluğundaki iskele sayısı kadar kendini çağırmaktadır. Fonksiyon her kendini çağırdığında problem uzunluğu 1 azalmaktadır. Algoritmanın yinelemeli olarak n kere kendini her defasında problem boyutunu 1 azaltarak çalışma prensibi üzerinden aşağıdaki çalışma zamanı fonksiyonu yazılabilir.

Geliştirilen algoritmanın çalışma zamanı T(n) = T(n-1)+T(n-2)+T(n-3)+ ……..+T(0)'dir.

Buradan;
T(n) = T(n-1) + T(n-2) + T(n-3)+ ……..+ T(0)
T(n+1) = T(n) + T(n-1) + T(n-2) + ….....+ T(0) + T(1)'dir.

T(n+1)'den T(n) çıkarıldığında;
T(n+1) – T(n) = T(n) + T(1) => T(n) = 2T(n-1) + T(1)'dir. T(0) = 0, T(1) = 1

Buradan; T(n) = 2T(n-1)+1'dir.

Bu yinelemeli fonksiyonu çözmek için n'in ilk birkaç değerini yazdığımızda;

T(n) = 2T(n-1) + 1

T(0) = 0
T(1) = 1
T(2) = 3
T(3) = 7
T(4) = 15
T(5) = 31
T(6) = 63
….
….
….
T(n) = 2^n -1 olduğu görebiliriz.

Bu sonucu ispat için matematiksel indüksiyon metodunu kullanabiliriz.

n = 0 için T(0) $= 2^0 - 1 = 0$ (doğru)
n = 1 için T (1) $= 2^1 - 1 = 1$ (doğru)

n = k için doğru kabul edelim.

T(k) = 2^k - 1 doğru olduğunu varsayalım.

n = k+1 için doğruluğunu test edelim.

T(k+1) = 2.(2^k -1) + 1 => T(k+1) = 2^{k+1} - 1 doğrudur.

Problemin çözümü için geliştirilmiş kaba kuvvet algoritmasının çalışma zamanı en iyi durumda $\theta(1)$, en kötü durumda $\theta(2^n)$'dir. $T(n) = \theta(2^n)$

3.5 Dallan Sınırla (Branch and Bound) :

Dallan sınırla algoritmasını 3.4 bölümde tasarlanan ve Şekil 2'de çalışma ağacı gösterilen kaba kuvvet algoritmasının durum uzayı küçültülerek gerçekleştirilmiştir. Algoritmamız için deptgh-first, breath-first ve best-first yöntemlerinden depth-firts arama seçilmiştir. Depth-first yöntemi ile problem ağacının ilk kollarında optimum sonuç elde edileceği için daha sonra yapılacak aramalar bu yaklaşımla kolayca sınırlandırılabilecektir.

3.5.1 Kaba Kuvvet Çözümü Sözde Kod

Dallan Sınırla Upper Bound Algoritması

```
// Tanım: Dallan sınırla algoritması için upper bound değeri bulur
// Girdi: Array DistanceList[l..n], ProfitList[l..n], Integer Limit, Integer Başlangıç iskelesinin indeksi
// Çıktı: Olabilecek kazanç durumu
1    def procedure upperBound (distanceList, profitList, limit, sourceStationIndex):

         #initialize variables
2        upperBoundVal = 0

3        for node = sourceStationIndex to n:

4            upperBoundVal = upperBoundVal + profitList[node]

5        return upperBoundVal
```

Algoritma 5 Upper Bound Algoritması

Dallan Sınırla Algoritması

```
// Tanım: En çok kar getirecek iskeleleri ve kazanç miktarı için dallan sınırla yöntemini kullanan çözüm sunar
// Girdi: Array DistanceList[l..n], ProfitList[l..n], Integer Limit, Integer Başlangıç iskelesinin indeksi
// Çıktı: En kazançlı iskele noktalarının indekslerini veren array ve toplam kazanç miktarı

1     def procedure branchAndBoundSolution(distanceList, profitList, limit, rootStationIndex):

          # initialize variables
2         bestSolution          = 0
3         upperBoundVal      = 0
4         bestStationList       = []


          # our best initial solution will be the greedy solution
5         bestSolution, bestStationList = greedySolution(distanceList, profitList, limit, rootStationIndex)

          # for first visit list go to n item
6         for currentStationIndex = rootStationIndex+1 to n:

              # initialize current profit and station in every step
7             newProfit           = 0
8             newStationList     = []

              # feasibility check
9             if ((distanceList[currentStationIndex]-distanceList[rootStationIndex]) > limit ):

10                upperBoundVal   = upperBound(distanceList, profitList, limit, currentStationIndex)
11                if (upperBoundVal > bestSolution) :


                      # call function for new sub list which root station is currentStationIndex
12                    subProfitValue, subStationList = branchAndBoundSolution(distanceList, profitList, limit,
                      currentStationIndex)

                      # calculate new profit and new station list from sub routine
13                    newProfit      = profitList[currentStationIndex]+subProfitValue
14                    newStationList.append(currentStationIndex)
15                    newStationList = newStationList+subStationList

                      # check new profit and update max profit and station list
16                    if (newProfit > bestSolution):

17                        bestSolution    = newProfit
18                        bestStationList     = newStationList

19                else:
20                    print "pruning because of bounding"
21            else:
22                print "pruning because of feasibility"
23        return bestSolution , bestStationList
```

Algoritma 6 Dallan Sınırla Algoritması

3.5.2 Dallan Sınırla Algoritması Açıklama

Dallan sınırla algoritmasında kaba kuvvet algoritmasıyla benzer yöntem kullanılmıştır. Bununla birlikte 10. adımda Algoritma 5'te tanımlanan upper boun fonksiyonu ile bir üst sınır değeri belirlenmekte ve 11. adımda bu değere göre fonksiyon sınırlandırılmaktadır.

Dallan sınırla upperBound algoritmasında 3-5 adımlar arasında algoritmaya girdi olarak verilen başlangıç iskelesine göre ondan sonraki iskelelerin toplam kazanç durumunu geri döndürmektedir.

3.5.3 Dallan Sınırla Algoritması Üst ve Alt Sınır Fonksiyonları

Aramamız da iki adet sınırlandırma yöntemi kullandık. Bunlardan ilki problemimiz için geçerli olan mümkünlük şartıydı. Problemimizde birbirlerine uzaklıkları belli bir K değerinin altında olan iskeleler aynı anda çözüm kümesinde yer alamıyordu. Biz bu problem kısıtını sınırlama yöntemlerinden ilki olarak belirledik. İkinci sınırlama yöntemimiz problem için geçerli olan üst sınır upper bound değeri oldu. Problemimiz için üst sınır upper bound fonksiyonu Algoritma 5'de tanımlanmıştır. Bu algoritmaya göre verilen iskeleden sonraki olası en büyük kazanç değerini geri döndürmektedir.
Algoritmamız için başlangıçta iyi çözüm olarak bestSolution (5. adım) değeri 3.1 hırslı sezgisel bölümünde oluşturulan algoritma ile elde ettiğimiz değeri atadık. Bu problemimiz için alt sınırı teşkil etti. Daha sonra derinlemesine analiz yöntemi kullanılarak optimum sonuç elde edilmeye çalışılmaktadır. Her bir adımda o adım için upper bound'a bakıldı bu değer en iyi çözüm (best solution) daha büyükse daha iyi değere ulaşmak için algoritma çalıştırılmaya devam etti.

4. Deneysel Analiz / Conclusion

Tablo 2 Algoritma Çalışma Zamanı Tablosu

Algoritma	**Çalışma Zamanı**	**Girdi Boyutu**
Hırslı Sezgisel	**θ(nlogn)**	**N**
Dinamik Programlam	**$O(n^2)$**	**N**
Parça Çöz	**-**	**-**
Kaba Kuvvet	**2^n**	**N**
Dalla Sınırla		**N**

Geliştirilen algoritmaların çalışma zamanları Tablo 2'de gösterildiği gibidir.

Probelem Girdi Boyutu	Hırslı Sezgisel	Dinamik Programlam	Parça Çöz	Kaba Kuvvet	Dalla Sınırla
N = 10	2.3	68.3	-	0.0	0.0
N = 20	4.7	0.0		2.3	0.0
N = 40	9.4	0.0			0.1

5. Sonuçlar / Conclusion

CSE 521 Algoritma analizi dersi kapsamında IDO İstanbul şehir hatları tarafından Beşiktaş'tan başlamak üzere boğaz kurulmak istenen iskelelerin en çok kazanç sağlayacak iskele lokasyon listenin bulunması amaçlı optimizasyon problemi için beş adet çözüm yöntemi için algoritma geliştirilmiştir. Bu çözüm yöntemleri hırslı sezgisel, dinamik programlama, parçala çöz, kaba kuvvet ve dallan sınırla yöntemleridir. Bu yöntemler içerisinde hırslı sezgisel algoritma her zaman optimal çözümü vermemektedir. Bununla birlikte büyük boyutlu veri setleri için hızlı çalışmaktadır. Parçala çöz yöntemi için problemi çözebilecek uygun bir algoritma geliştirilememiştir. Bunun nedeni problemin problemin boyutunu küçültecek alt problemlere ayırılamaması olmuştır. Bununla birlikte parçala azalt (divide and decrease) algoritması bu problemin çözümünde uygulanmıştır. Dinamik programlama, kaba kuvvet, ve dallan sınırla yöntemleri ile problemin çözümü için optimum sonucu verecek algoritmalar oluşturulmuştur. Kaba kuvvet algoritması küçük boyutlu veri setlerinde sonucu hızlı bir şekilde verebilirken problem boyutu büyüdükçe çalışma zamanı üstsel şekilde arttığından, problem boyutundaki küçük artışlar çalışma zamanında büyük değişimlere neden olmuştur. Buna karşı dallan sınırla algoritmasıyla kaba kuvvet algoritmasındaki gereksiz hesaplamalar azaltılarak problemin çalışma zamanı azaltılmıştır. Dinamik programla yöntemi ile de problemin çözümü için fazladan depolama alanları kullanılarak problemin çalışma zamanı belirgin ölçüde azaltılmıştır. Tüm bunları sıra problemde kısıtlarından birisi olan iki iskele arasında olabilecek en küçük uzaklığı tanımlayan K değişkenin değeri de problemin çalışma zamanına önemli ölçüde etki ettiği gözlenmiştir. K değerinin küçülmesi problem için çalışma kümesi arttırmakta büyümesi azaltmakta olduğu gözlenmiştir. Problemin çözümüne ilişkin ilgili geliştirilen algoritmalar python programlama dili ortamında uygulanmış, çalışma zamanı çıktıları için karşılaştırma tablosu oluşturulmuştur.

5. Kaynaklar / References

[1] Sevilgen, Erdoğan, "Algoritma Analizi Ders Sunuları", 2015, Gebze Teknik Üniversitesi, Bilgisayar Mühendisliği Bölümü

[2] Timsort, http://svn.python.org/projects/python/trunk/Objects/listsort.txt, Son erişim tarihi :29.04.2015