

# Çizgeler & Huffman Encoding

Ege Üniversitesi Bilgisayar Mühendisliği

Veri Yapıları

Proje-4

Hüseyin YAŞAR 05-06-7657

Didem KAYALI 05-06-7669

Umut BENZER 05-06-7670 <http://www.ubenzer.com/>

Özlem GÜRSES 05-07-8496

**Teslim tarihi: 8 Ocak 2009**



# Programcı Katalogu

---

## 1.1-Platform ve Dil

Projenin geliştirilmesinde tüm grup üyeleri kendi bilgisayarlarını kullandığından birçok farklı platform kullanılmış olsa da, projenin birleştirilmesi ve beta testleri Umut Benzer'in bilgisayarında Windows Vista altında Eclipse 4.3 Ganymede IDE'sinde yapılmıştır. Projenin yazımında kullanılan dil JAVA 5 (JDK 1.6)dir.

## 1.2-Problem

Dosyadan bir çizge okumak, çizgeyi komşuluk matrisinde tutmak. Bu çizge üzerinde en kısa yol algoritması başta olmak üzere ödevde sözü edilen algoritmaları uygulayabilmek.

## 1.3-Sınıflar ve Metotlar

### Algorithms

Maindeki metot sayısını azaltmak ve aynı dosya üzerinde birden fazla kişinin aynı anda çalışmasına gerek kalmadan kod geliştirmeye imkân sağlamak için çizge algoritmaları bu sınıfa alınmıştır. Bu sınıftaki tüm metotlar *static* olup, bu sınıf bir nesne yaratmak için tasarlanmamıştır.

```
/**
 * @param: Ağırlıklı çizge
 * @param: Başlangıç düğüm numarası
 * @param: Bitiş düğüm numarası (-1: tüm düğümleri bul)
 * @return: En kısa yol sırasını içeren dizi
 * @see: http://www.cs.fit.edu/~ryan/java/programs/graph/Dijkstra-java.html
 * @author: UB
 */
public static int[] dijkstra(WeightedGraph G, int s, int f)
```

En kısa yolu bularak düğüm numaralarını döndürür. Bunlar daha sonra başka fonksiyonlar ile yazıya çevrilmiştir.

```
/**
 * @param: Uzaklık dizisi
 * @param: Gezilme durumu dizisi
 * @return: Gezilmeyen en yakın düğüm
 * @see: http://www.cs.fit.edu/~ryan/java/programs/graph/Dijkstra-java.html
 */
private static int minVertex(int[] dist, boolean[] v)
```

Gerekli bilgileri alarak gezilmeyen en "hafif" çizgiyi döndürür. Bu metot hazır alınmış ve değiştirilmemiştir.

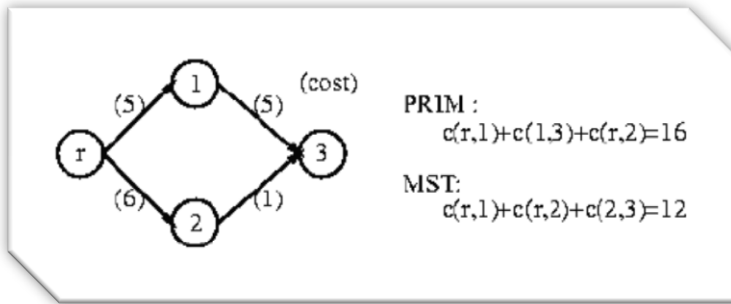
```
/**
 * @param: Ağırlıklı çizge
 * @return: Çizgenin yönlü olup olmadığı (komşuluk matrisi simetrik mi?)
 * @author: Hüss
 */
public static boolean isDirected(WeightedGraph G)
```

```

/**
 * @param: Ağırlıklı çizge
 * @param: Başlangıç düğüm numarası
 * @return: MST bulunduğundan sonra düğümler arası bağlantıları içeren dizi
 * @see: http://www.cs.fit.edu/~ryan/java/programs/graph/Prim-java.html
 * @author: Özlem
 */
public static int[] prim(WeightedGraph G, int s)

```

Prim algoritması ile minimum spanning tree bulunur. Dönen dizinin minimum spanning tree elde edilince kalan çizgileri tutar. Örneğin  $sonuc[0] = 1$  ise 0 numaralı düğüm birinci ile bağlantılıdır. (çizge yönsüz olduğu için tersi de geçerlidir.)



Prim algoritma yönlü algoritmalarda doğru sonuç üretmez. Bu yüzden programımızda digraphlar üzerinde Prim çalıştırılması engellenmiştir. Her ne kadar Edmond algoritmasını JAVA'ya aktarmaya çalışmışsak da başarılı olmadık. Yapabildiğimiz kadarı ayrıntılı açıklama satırlarıyla beraber kaynak kodunda bulunabilir.

```

/**
 * @param: Ağırlıklı çizge
 * @param: En kısa yol bilgilerini içeren dizi
 * @param: Başlangıç düğümü
 * @param: Bitiş düğümü
 * @return: Başlangıç ile bitiş arasındaki düğüm adları vektörü
 * @author: Özlem
 */
public static Vector<String> path(WeightedGraph G, int[] pred, int s, int e)

```

En kısa yol bilgisini, ağırlıklı çizgeyi başlangıç ve bitiş düğümlerini alarak bunları yazılmaya hazır bir metin vektörü haline dönüştürdükten sonra işlenmek üzere çağırıldığı fonksiyona gönderir.

```

/**
 * @param: Ağırlıklı çizge
 * @param: Başlangıç düğüm numarası
 * @return: BFS ile gezilince oluşacak düğüm sırası
 * @author: Didem
 */
public static int[] bfs(WeightedGraph w, int startNode)

```

Çizgenin BFS dolaşılması ile oluşan sıra bir dizi olarak çağırıldığı fonksiyona gönderilir.

## BListe

Alt yapı olarak bağlı liste kullanarak kuyruk yapısı tutan bir sınıftır. Banka kuyruğu ödevinden alınarak üzerinde oynamalar yapılmıştır.

```
public void ekle(int node) Kuyruğa yeni bir tam sayı değeri ekler.
```

```
public int cikar() Kuyruktan bir eleman çıkarır.
```

```
public boolean bosMu() Kuyruğun boş olup olmadığını denetler.
```

## BListeNode

Bağlı liste yapısında referans değeri tutmak için gerekli düğüm sınıfıdır. Başka kullanım amacı yoktur.

## Gorsel

Arayüz ile ilgili kodlamaların olduğu sınıftır. Bu yüzden metotlara tek tek değinilmeyecektir. Arayüz Eclipse Ganymede Visual Editör ile yaratılmıştır.

## Main

Dosyadan veri okuyan, arayüzü başlatan, arayüzden gelen verileri denetleyerek çizge algoritmalarının kullanılmasına uygun hala dönüştüren (ya da dönüştüremeyen) temel metottur.

```
/**
 * Dosyadan veri okur.
 * @author: UB
 * @link: http://www.roseindia.net/java/beginners/java-read-file-line-by-line.shtml
 */
public static void load()
```

İller ve maliyet dosyalarını okuyarak çizgeyi oluşturan metottur. Bu metot oldukça esnektir ve bir çok şekilde kodlanmış metinleri okuyabilir. Ayrıntılı bilgi metodun içindeki açıklamalarda mevcuttur.

```
/**
 * @author: Didem
 */
public static void main(String[] args)

/**
 * Ekrana yazdırılmaya uygun bir dizaynda maliyet dizisinin içindeki
 * bilgileri gönderir.
 *
 * @author Didem
 * @return String
 */
public static String maliyetDiziScreen()
```

```
/**
 * Düğümün içindeki bilgileri ekrana yazılmaya uygun formatta
 * gönderir.
 * @param: Düğüm no
 * @return: String
 * @author: Didem
 */
public static String getNodeDetails(int nid)
```

```
/**
 * @param: Başlangıç düğüm no
 * @return: String
 * @author: Özlem
 */
public static String bfs(int start)
```

Uygun veri kontrolü yaptıktan sonra BFS algoritmasını çalıştırır. Dönen sonucu yorumlayarak ekrana yazılabilecek şekilde çağrıldığı fonksiyona gönderir.

```
/**
 * @param: Başlangıç düğüm no
 * @return: String
 * @author: UB
 */
public static String mst(int start)
```

Uygun veri kontrolü yaptıktan sonra MST algoritmasını çalıştırır. Dönen sonucu yorumlayarak ekrana yazılabilecek şekilde çağrıldığı fonksiyona gönderir.

```
/**
 * @param: Başlangıç düğüm no
 * @param: Bitiş düğüm no (-1 hepsi)
 * @return: String
 * @author: Özlem
 */
public static String dijkstra(int start, int finish)
```

Uygun veri kontrolü yaptıktan sonra en kısa yol algoritmasını çalıştırır. Dönen sonucu yorumlayarak ekrana yazılabilecek şekilde çağrıldığı fonksiyona gönderir.

```
/**
 * @param: String bir değer
 * @return: Stringin bir tam sayı olup olmadığı
 */
public static boolean isParsableToInt(String in)
```

Verilen stringin bir tam sayı olarak yorumlanıp yorumlanamayacağını gönderir. Şehir adı mı şehir numarası mı verildiğini tespit etmek için idealdir.

## WeightedGraph

Ağırlıklı çizmeyi tutmak için kullanılan nesnedir. Açık kaynak kod üzerinden gerekli ölçüde modifiye edilerek oluşturulmuş bir sınıftır.

```
/**
 * Referans değil, çizgenin tamamını bire bir kopyalamaya çalışan
 * bir metottur.
 * @deprecated
 * @author: UB
 */
public Object clone()
```

Edmonds algoritması çalışırken çizgede değişiklik yapması gerektiğinden çizmeyi kopyalama yoluna gidilmişti. Ancak Edmonds algoritmasını tam olarak yazamadığımızdan bu metot kullanılmamaktadır.

**public int** size() Çizgenin boyutunu gönderir.

**public void** setLabel(int vertex, String label) İlgili düğüme etiket atar.

**public** String getLabel(int vertex) İlgili düğümün etiketini gönderir.

**public void** addEdge(int source, int target, int w) İki düğüm arasına yol ekler.

**public void** removeEdge(int source, int target) İlgili yolu siler.

**public int** getWeight(int source, int target) İlgili yolun ağırlığını gönderir.

**public int[]** neighbors(int vertex) İlgili düğümün komşularını (giden) bir dizi içerisinde gönderir.

```
/** Bir köşeye gelen bağlantıları bulur.
 * @author Hüss
 * @param vertex number
 * @return incoming links
 */
public int[] incoming(int vertex)
```

İlgili düğümün komşularını (gelen) bir dizi içerisinde gönderir.

```
/**Etiketini köşe numarasını bulur. Bulamazsa -1 gönderir.
 * @author Hüss
 * @param vertex label
 * @return vertex number
 */
public int labelToVertex(String name)
```

Bu metot sayesinde programımızda ek özellik olarak şehir ismine göre de işlem yapabilmekteyiz.

**Edmonds algoritması için yazılmış, ancak algoritmanın karmaşıklığı nedeniyle tamamlanamamış metotlar proje içerisinde yer almaktadır, ancak raporda açıklanmayacaktır.**

## 1.4-Veri Yapıları

Programda çizge, bağlı liste, vektör ve dizi yapıları kullanılmıştır. Çizge komşuluk matrisinde tutulmuştur. Daha ayrıntılı bilgi metotlarda ve metotların içindeki açıklama satırlarında bulunabilir.

## 1.5-Dosya Özellikleri

**iller.txt** illerin adını (ve belki de sayısını) tutan dosyadır. Dosyanın en üst satırında dosyada kaç tane şehir olduğunun sayısı yazabilir de yazmayabilir de. Program her türlü dosyayı doğru okuyacaktır. Eğer şehir sayısı ile dosyada gerçekte olan şehirler tutmuyorsa şehir sayısı bilgisi ihmal edilir. Daha ayrıntılı bilgiyi programın açıklama satırlarında bulabilirsiniz.

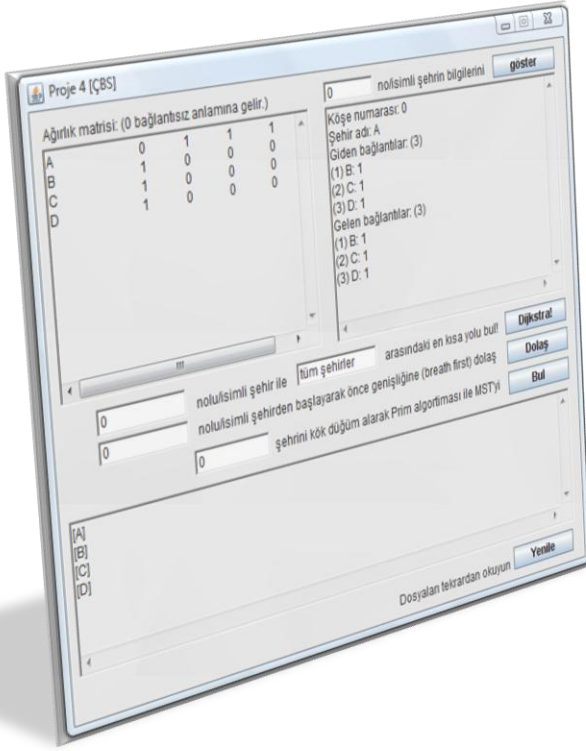
**maliyet.txt** iller arasındaki maliyetleri tutan dosyadır. Eğer iller arasında bağlantı yoksa -1 kullanılmıdır. Dosyada ılsayısı<sup>2</sup> tane sayısal (*bu projede integer*) değer olmalıdır. Daha az olduğu durumda program kalan değerleri -1 olarak değerlendirecektir. Aynı şekilde dosya okunmadığında da hiçbir il arasında bağlantı olmadığı düşünülecektir. Eğer ılsayısı<sup>2</sup>den fazla değer varsa sondaki değerler ihmal edilecektir. Değerlerin satırlara doğru yerleşmesinin ya da arada fazla boşluk olmasının bir anlamı yoktur. Aralarda geçersiz değerler varsa (*misal 1 2 3 SFGSG 45*) bu değerler -1 olarak değerlendirilecektir.

Dosyalar proje ile aynı klasörde bulunur.

## 1.6-Çalışma Süreleri

Projenin tamamlanması, karar verme süreci dahil olmak üzere toplamda yaklaşık 96 saat almıştır. Bu ortalama geçen süre içerisinde her bir çalışan kendi esnekliğine bağlı olarak günde en az 2 saatlik çalışma yapmıştır. Elimizde sağlıklı bir veri olmadığından ne yazık ki kişi bazında çalışma saatlerini net bir şekilde veremiyoruz. Ancak fonksiyon karmaşıklığı ve yazılan fonksiyon sayısı oranlanarak tahmini rakamlar elde edilebilir. (Projenin her fonksiyonunun yazarı belirtilmiştir.)

# Kullanıcı Katalogu



## 2.1-Programın İşletimi

Program tek pencerele bir arayüzden oluşmaktadır. Ne yazık ki çizgelerin grafiksel çizimi programda mevcut değildir. Programdaki tüm işlemleri hem şehir numarasına göre, hem de şehir adına göre yapabilirsiniz.

## 2.2-Kullanım Kılavuzu

Çökmez Bilişim Sistemleri'nden bir çizge analiz programı. Bizi tercih eden siz saygıdeğer kullanıcımıza teşekkür ederiz.

Program tek pencere üzerinde çalışmakta olduğundan oldukça basit ve kullanışlı bir arayüze sahiptir.

Programın sol üst köşesindeki kutuda dosyadan okunan çizgenin bilgileri gösterilir. Çizgenin bir düğümü hakkında daha fazla bilgi almak isterseniz, sağ üst kısımda bulunan kutucuğa düğümün ismini ya da numarasını yazarak göster düğmesine basmanız yeterli olacaktır.

ÇBS Çizge yazılımı en kısa yol, DFS ve MST bulmaya izin verecek şekilde tasarlanmıştır. Bu algoritmaları kullanmak isterseniz ilgili kutuları doldurup ilgili düğmeye basmanız yeterli olacaktır. Sonuçlar ekranın alt kısmında görünecektir.

Eğer çizgede değişiklik yaptıysanız programı yeniden başlatmak zorunda değilsiniz. Çökmez Bilişim Sistemlerinin ileri teknoloji yazılımlarıyla bu sorun da ortadan kalktı! Ekranın en altında bulunan yenile düğmesi ile programı yeniden başlatmaksızın dosyaları tekrar okuyabilirsiniz.

## 2.3-Programın Kısıtlamaları

Program yönlü çizgeler için en küçük kapsayan ağaç hesaplayamamaktadır. (Ancak bu konuda çalışmalarımız devam etmektedir.)

Programımız çizgeleri grafik arayüzüne çizememektedir.

Şu an için programımıza dosya dışında yollardan çizge girişi yapılamamaktadır.

# Huffman Encoding

---

Huffman algoritması 1952 yılında David A. Huffman tarafından kurulmuş, verileri sıkıştırmaya yarayan algoritmadır. Verilerin kayıpsız sıkıştırılmasına olanak sağlar. Bu algoritma bir entropi kodlama algoritmasıdır.

Peki, 'entropi kodlama algoritması' ile ne denilmek istenmiştir?

Entropi bir bilgisayar bilimleri terimi olarak, bilgi içeriği birimidir: bir veriyi tutmak için aslında ne kadar bit gerektiğidir.

Entropi bazen bir *sürpriz* ölçüsüdür.

Önceden yüksek derecede tahmin edilebilen zincirlemeler hakiki bilginin az bir kısmını içerirler.

Örneğin: 11011011011011011011011

Bu sıralamada daha sonra ne geleceği tahmin edilebilir. Bütününüyle önceden tahmin edilemeyen n bit sıralaması n bitlik bir bilgiyi içerir.

Örneğin: 01000001110110011010010000

Bu sıralamada daha sonra ne geleceğini tahmin edemezsiniz. Verinin her zaman bir anlam taşıması gerekmediğine dikkat edilmesi gerekir.

## Hakiki Bilgi İçeriği

Kısmen tahmin edilebilen n bitlik bir bit zinciri n bitten daha az bilgiyi içerir.

Örnek #1: 111110101111111100101111101100

3 Bloğu: 111110101111111100101111101100

Örnek #2: 101111011111110111111011111100

Eşit olmayan olasılıklar:  $p(1) = 0.75$ ,  $p(0) = 0.25$

Örnek#3: "Dünya, savaşız ne kadar güzel olurdu..."

Karakter olasılıkları: a ve l ortak ama j ya da ğ ortak değil.

## Sabit ve değişken bit Genişlikleri

Bir Türkçe metni şifrelemek için yirmi dokuz büyük harf ve yirmi dokuz küçük harf ve bilimum noktalama işaretleri gereklidir. Biz onu tutmak için altmış dört karakter(6 bit) kullanırız.

## Huffman Algoritması Örneği

A = 0

B = 100

C = 1010

D = 1011

R = 11

ABRACADABRA = 01001101010010110100110

Bu, yirmi üç bitte on bir harftir.

Sabit genişlik şifrelemesi beş farklı harf için 3 bit gerektirmektedir ya da on bir harf için otuz üç bit gerektirmektedir. Şifrelenmiş bir bit kelimesinin şifresinin çözülebileceğini de unutmamak gerekir.

## Nasıl Çalışır?

Aşağıdaki örnekte 'A' ortak harftir.

ABRACADABRA:

5 A: A 1 bit uzunluğunda

2 R: R 2 bit uzunluğunda

2 B: B 3 bit uzunluğunda

1 C: C 4 bit uzunluğunda

1 D: D 4 bit uzunluğunda

## Huffman Encoding Yaratma

Her şifreleme ünitesi için sıklık frekansı yaratılır. –Ya da yüzde veya olasılık da kullanılabilir–

Bir ikili ağaç yaratılır. Ağacın elemanları frekansları tutar. Kökte, bütün frekansların toplamı tutulur.

Her düğümün çocukları frekansları düğümden küçük olan şifreleme üniteleridir.

## Örnek

### Aşama 1:

Frekansları aşağıdaki gibi olan kelimeler olduğunu varsayalım;

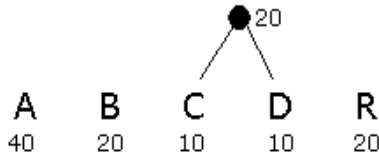
A: 40

B: 20

C: 10

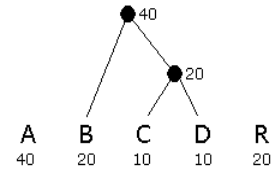
D: 10

R: 20



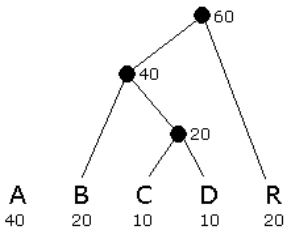
C ve D en frekansları en küçük olan harfler. Dolayısıyla onlar bağlanmıştır. C ve D artık beraber bir düğümdürler (C+D). Bu C+D düğümünün frekansı 20 olmuş olur.

### Aşama 2:



Yeni durumda, en küçük frekanslar B, C+D ve R'nin frekanslarıdır. Herhangi ikisi öncelikle bağlanır.

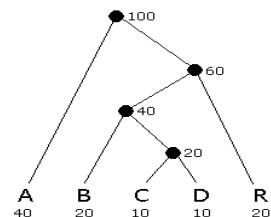
### Aşama 3:



A ve B+C+D değişkenleri 40 iken en küçük değişken R'dir.

Diğer ikisinden biri R ile bağlanır.

### Aşama 4:



En son kalan iki düğüm de bağlanırsa ağaç yandaki gibi olur.

### Aşama 5:

Sol dallara 0, sağ dallara 1 atanır. Her bir şifreleme kökten çıkan bir yoldur. Her yol bir yaprakta sonlanır.

### Benzersiz Prefix Özelliği

A = 0

B = 100

C = 1010

D = 1011

R = 11

Hiçbir bit kelimesi başka herhangi bir bit kelimesinin prefixi değildir.

Örneğin, E=01 derseydik, A(0) E'nin bir prefixi olurdu.

Benzersiz prefix özelliği vardır çünkü ikili ağaçlarda bir yaprak diğer düğümlerin yoluna çıkmaz.

## Pratik Noktalar

ABRACADABRA gibi küçük ve basit bir kelime için Huffman şifrelemesi kullanılması pratik değildir. Bunu çözmek için, bir kod tablosuna ihtiyaç duyulur. Eğer kod tablosunu bütün mesaja dâhil ederseniz, tüm iş ACSII mesajından büyük olur.

Huffman şifrelemesi basit bir veri sıkıştırma örneğidir: Verileri diğer şekillerde ihtiyaç duyulacağından daha az bitlerle ifade eder. Bu algoritma hem decode hem de encode özelliğine sahiptir. Ağaç yapısından yararlanılarak hem şifreleyebilir hem de şifrelenmiş bir kodu çözebilirsiniz.

## Algoritmanın İşleyişi

Huffman algoritmasının temelinde bir ağaç veri yapısı vardır. Aynı zamanda (daha etkin olması istendiğinde)kuyruk –öncelik kuyruğu- veri yapısı da kullanılmaktadır. Ancak etkinlikten feragat edebilen programcılar için öncelik kuyruğu kullanmak elzem değildir.

## Şifreleme

Şifrelenecek olan harflerin frekansları bulunur ve bir öncelik kuyruğuna atılır. (*Öncelik kuyruğunda sıralama küçükten büyüğe doğru yapılmalıdır.*) Kuyruktan iki harf çıkarılır, bu harflerin frekansı toplanır ve birleştirilir. Oluşan alt ağaç da öncelik kuyruğuna konulur. Tüm elemanlar birleştirilene kadar bu işlem tekrarlanır. İşlem yapılırken bir ikili ağaç yaratılmış olur. Kodlamaya geçildiğinde ise yukarıda da belirttiğimiz gibi ağacın sağ çocuklarına sıfır, sol çocuklarına ise bir değeri verilerek yaprakta bulunan harflere ulaşana kadar işlem tekrarlandığında her elemanın şifresi elde edilir.

Bu şekilde gidildiğinde fark edilir ki köke en yakın düğüm frekansı en yüksek olan yani en sık kullanılan elemanı tutan düğümdür. Sıkıştırmanın mantığı budur. Örneğin ASCII kodu her harf için sekiz bit ayırmıştır. Huffman şifrelemesinde elemanların tutulduğu bit sayısı sabit değil, değişkendir. Pratik bir çözüm olarak sıkıştırılacak metinde en çok kullanılan eleman en az sayıda bit ile tutulur ve böylece yerden kazanç sağlanmış olur. ASCII ile sekiz bit yer kaplayan bir harf Huffman algoritması ile sıkıştırıldığında iki bit kaplayabilir. Bu da sık tekrar eden yani frekansı yüksek olan elemanlar için toplamda çok büyük bir alanın salınması demektir ve her eleman için mutlaka bir şifre üretileceğinden verilerde kayıp engellenmiş olur.

Şifre çözme ise çok basittir, verilen bir şifrenin başından başlanarak ağaçta ilerlenir. Kökten başlanır ve şifreye bakılır. Şifrede sıfır varsa kökten sola, bir varsa kökten sağa gidilir. Alt ağaçlar için de aynı mantık vardır. Yani şifre çözme, tıpkı şifreleme gibi *özyineleme* ile işler.

## Huffman Algoritmasının Java Örneği

```
public void readData() {
    openStream();
    try{
        int c= inFile.read();
        while(c!=-1){
            addFreqInHashtable(c);
            c=inFile.read();
        }
    } catch(IOException e){}
    filterHashtable(); //A-Z disindaki karakterlerin elenmesi
    sortByFrequency();
}

public void HuffmanTreeYarat() {
    readData();
    tree = new Node(); // agaci bosaltma

    while (sortedNodes.size()>1) {
        Node comNode = new Node((Node)sortedNodes.firstElement().(Node)sortedNodes.elementAt(1));

        //ilk iki dugumu siralanmis dugumlerden silme.
        sortedNodes.removeElementAt(0);
        sortedNodes.removeElementAt(0);

        boolean inserted = false;
        for (int i = 0; i<sortedNodes.size(); i++) {
            if(comNode.getWeight() <= ((Node)sortedNodes.elementAt(i)).getWeight()) {
                sortedNodes.insertElementAt(comNode, i);
                inserted = true;
                break;
            }
        }
        if(!inserted) {
            sortedNodes.addElement(comNode);
        }
    }
    if(sortedNodes.size()<1)
        return;

    tree = (Node)sortedNodes.firstElement();

    table = new Hashtable();
    String code = new String();
    encodingHashtable(tree, table, code);
    encoding_decoding_ornek("HELLO");
}

public void encodingHashtable(Node tree, Hashtable table, String code) {
    if(tree.isLeaf()) {
        Character c = new Character(tree.getLabel().charAt(0));
        table.put(c, cpde);
    } else {
        encodingHashtable(tree.getLeftNode(), table, code + "0");
        encodingHashtable(tree.getRightNode(), table, code + "1");
    }
}
```

```

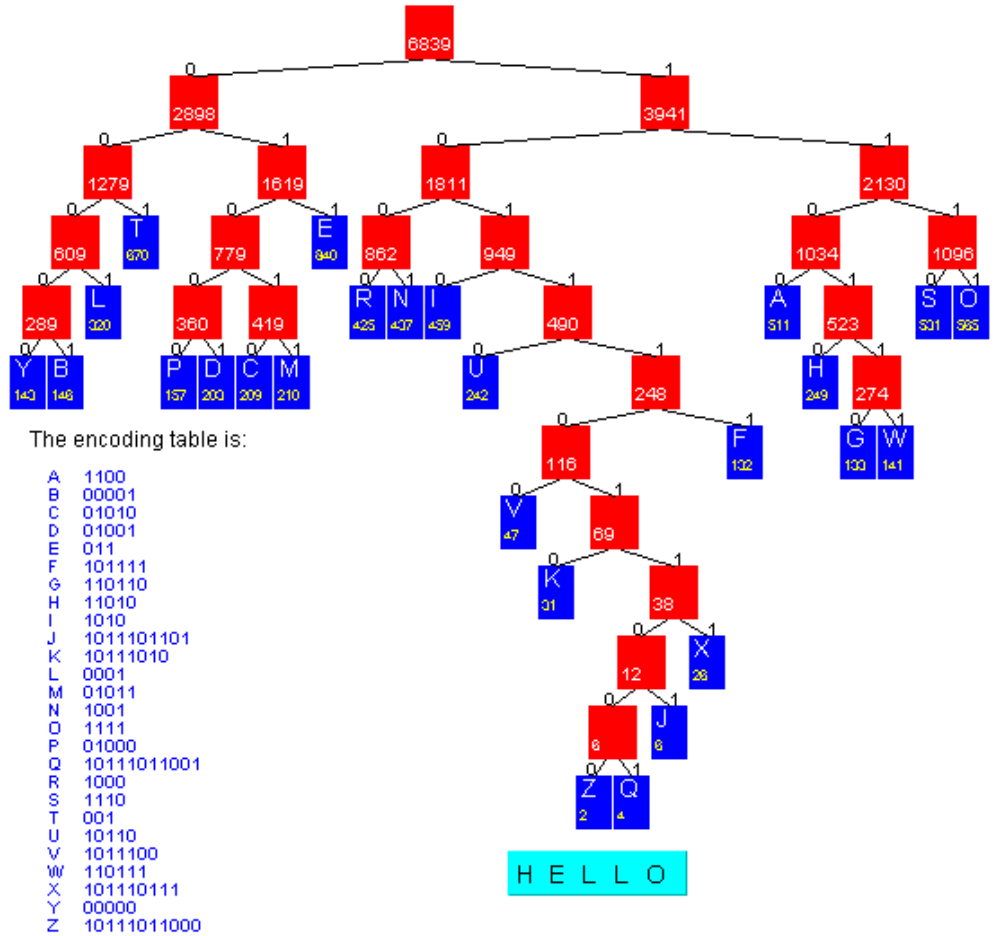
public void encoding_decoding_ornek(String str) {
    String code = new String();
    for(int i = 0; i<str.lengths(); i++) {
        Character c = new Character(str.charAt(i));
        code = code.concat((String)table.get(c) + "");
    }

    String Tokenizer st = new String Tokenizer(code);
    code = new String();
    while(st.hasMore Tokens()) code = code.concat(st.next Token());
    string result = new String();
    decode(code, tree, tree, result);
}

public void decode(String code, Node branch, Node tree, String result) {
    if(branch.isLeaf()) {
        if(code.lengths()<1) {
            result = result.concat(branch.getLabel());
            decode(code, tree, tree, result);
        } else {
            result = new String(result + branch.getLabel());
        }
    } else {
        if(code.startsWis("0")) {
            decode(code.substring(1), branch.getLeftNode(), tree, result);
        } else {
            decode(code.substring(1), branch.getRightNode(), tree, result);
        }
    }
}
}

```

Kodun ürettiği ağaç yapısı ve şifreleme tablosu şu yandaki gibi olacaktır.



# Yararlanılan Kaynaklar

---

## Çizgeler

<http://www.cs.fit.edu/~ryan/java/programs/graph/Dijkstra-java.html>  
<http://www.cs.fit.edu/~ryan/java/programs/graph/Prim-java.html>  
<http://www.roseindia.net/java/beginners/java-read-file-line-by-line.shtml>  
<http://www.ce.rit.edu/~sjyeec/dmst.html>  
[http://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree)  
[http://en.wikipedia.org/wiki/Prim%27s\\_algorithm](http://en.wikipedia.org/wiki/Prim%27s_algorithm)  
[http://en.wikipedia.org/wiki/Chu%E2%80%93Liu/Edmonds\\_algorithm](http://en.wikipedia.org/wiki/Chu%E2%80%93Liu/Edmonds_algorithm)  
[http://en.wikipedia.org/wiki/Depth-first\\_search](http://en.wikipedia.org/wiki/Depth-first_search)  
[http://en.wikipedia.org/wiki/Breadth-first\\_search](http://en.wikipedia.org/wiki/Breadth-first_search)

## Huffman

<http://www.cs.auckland.ac.nz/software/AlgAnim/huffman.html>  
<http://www.dspguide.com/ch27/3.htm>  
<http://www.cs.auckland.ac.nz/software/AlgAnim/huffman.html>  
[http://en.wikipedia.org/wiki/Huffman\\_coding](http://en.wikipedia.org/wiki/Huffman_coding)  
[http://tr.wikipedia.org/wiki/Huffman\\_kodu](http://tr.wikipedia.org/wiki/Huffman_kodu)  
<http://www.huffmancoding.com/david-huffman/huffman-algorithm>  
<http://www.britannica.com/EBchecked/topic/274828/Huffman-encoding>

Kaynaklara ulaşmamızın anahtarı  arama motoruna teşekkürü borç biliriz.