İSTANBUL TEKNİK ÜNİVERSİTESİ * FEN BİLİMLERİ ENSTİTÜSÜ

AĞAÇ YAPISININ LEMPEL-ZIV VERİ SIKIŞTIRMA ALGORİTMASINA UYARLANMASI

YÜKSEK LİSANS TEZİ

Müh. Tolga ULUS

Tezin Enstitüye Verildiği Tarih : 25 Ocak 1993
Tezin Savunulduğu Tarih : 11 Şubat 1993

Tez Danışmanı : Doç. Dr. Mithat UYSAL
Diğer Jüri Üyeleri : Prof. Dr. Galip TEPEHAN
Doç. Dr. Fikret BALTA

ŞUBAT 1993

ÖNSÖZ

Bu çalışmada, veri sıkıştırma yöntemlerinin tanıtımları yapılmış ve Lempel-Ziv yöntemi üzerinde yoğunlaşılmıştır. Eklerdeki bilgisayar programlarının hepsi IBM uyumlu kişisel bilgisayarlarda, DOS işletim sistemi altında, Pascal programlama dili ile yazılmıştır. Pascal derleyicisi olarak Turbo Pascal 5.5 seçilmiştir.

Türk alfabesindeki harflerin frekans dağılımının bulunmasında kullanılan Türkçe metinler değişik kaynaklardan sağlanmıştır. Kamil Adem, Yaman Başa, Osman Darcan, Zeynep Deleon, Hale Barry, Bekir Kara, Nilgün Kıran, Alin Oskanyan, Meltem Özturan, Yalçın Pekşen ve Pakize Yapa olmadan bu metinleri sağlamam olanaksızdı.

Bu çalışmada, bana yol gösteren danışmanım Prof.Dr.Metin Demiralp'e teşekkür ederim. Ayrıca, benden yardımlarını esirgemeyen çalışma arkadaşlarım, Boğaziçi Üniversitesi Meslek Yüksekokulu Teknik Programlar Bölümü Bilgisayar Programcılığı öğretim elemanlarına; en zor zamanlarımda bana destek olan arkadaşlarıma; eğitim ve öğrenim yaşamım boyunca bana rehber olan öğretmenlerime; son olarak da benim yaşama nedenim, azmim ve isteğim zayıfladığında yanımda olarak bana azim ve istek aşılayan aileme en içten teşekkürlerimi sunarım.

Tolga Ulus

İÇİNDEKİLER

ÖZET

Bu çalışmada, veri sıkıştırma algoritmalarından Lempel-Ziv algoritmasında kullanılan veri yapısı ikili ağaç seçilmiştir. Kullanılan bu veri yapısı sayesinde algoritma oldukça hız kazanmıştır.

Veri sıkıştırma, hem veri saklama, hem de iletişim alanında büyük kolaylıklar sağlamaktadır. Bu yolla, aynı ortamda daha fazla bilgi saklanabilmekte veya aynı bant genişliğinde daha fazla bilgi daha kısa sürede gönderilebilmektedir.

Metinlerde, karakter dağılımı, karakter tekrarı, çok kullanılan kalıplar gibi çeşitli gereksizlik tipleri vardır. Bir metni sıkıştırmak, bu metindeki gereksizliklerin belli bir oranda yokedilmesiyle olur. Varolan sıkıştırma yöntemleri, bu gereksizliklerin birini veya bir kısmını hedef alırlar. İdeal sıkıştırma ise, metindeki bütün gereksizliklerin tamamen yokedilmesiyle olur.

Yarım asır boyunca çeşitli sıkıştırma yöntemleri önerilmiştir. Bunlardan bir grubu, çeşitli karakteristiklerdeki metinler için özel olarak hazırlananlardır. Diğer bir kısmı ise statik olarak adlandırılırlar. Statik yöntemler metin üzerinden iki kere geçerler. İlkinde şifreleri oluşturur, ikincisinde ise bu şifreleri kullanarak sıkıştırılmış metni yaratırlar. Dinamik yöntemler ise, metin üzerinden sadece bir kez geçerek, hem şifrelemeyi oluşturur veya değiştirir, hem de sıkıştırılmış metni yaratırlar. Bunlar metnin değişen karakteristiğine adapte olduğundan statik yöntemlerden daha yüksek oranda sıkıştırma sağlarlar.

Dinamik yöntemlerden Lempel-Ziv, bir karakter katarı tablosunun tutulması ve idare edilmesini gerektirir. Bu tablo, sırasal dizi olarak tutulduğunda sıkıştırma algoritması oldukça uzun çalışmaktadır. Tablo ikili ağaç yapısında tutulduğu zaman, sıkıştırma diğerine oranla çok daha hızlı olmaktadır. Genişletmede ise, durum tam tersidir. Tablo, sırasal dizi olarak tutulduğunda daha hızlı genişletme sağlanmaktadır.

SUMMARY

APPLICATION OF TREE STRUCTURE TO LEMPEL-ZIV DATA COMPRESSION ALGORITHM

In this study, a variety of data compression techniques are presented. The study especially concentrates on Lempel-Ziv algotihm, one of the latest developments in the area. The binary tree structure is proposed that speeds up the algorithm over the classical array structure.

Data compression is the process of transforming a series of bytes into a new string that contains the same information but whose length is as small as possible. Data compression has important applications in the areas of information storage and data communication. Most computer applications need a large amount of data storage. Also, computer communication networks transfer large volumes of data over communication lines. Compressing data to be stored or transferred reduces the cost of storage and transmission. When files are stored in compressed form, the capacity of storage medium is approximately doubled. Similarly, when the amount of data to be transmitted is reduced, the capacity of communication channel is increased. Thus, data compression provides an economical and indispensible option to store and transmit the information.

Data compression may be achieved in either software or hardware. When it is hardware, a compressor is a hardware device that reads a series of bytes of data and converts it into a shorter string; and decompressor is a device that converts the compressed data back to its original form. When data compression is done in software, a compression algorithm does the former; and an expansion or decompression algorithm does the latter.

The logic that enables data compression is that the same information retained in the text after some bytes are replaced with shorter representations. In other words, data compression is achieved by removing the redundancy in the text. There are several types of redundancy found in the texts. These categories are not independent, but overlap to some extend. The major redundancy types are character distribution, character repetition, high-usage patterns and positional redundancy.

In typical texts, some bytes occur more frequently than others. Moreover, some bytes may never be used. Thus, a few bits on the average might be saved in the representation of bytes, using less number of bits for most frequently used bytes and more number of bits for least frequently used bytes.

In some files such as texts and business files, long repetitions of a single character occurs. These files can be encoded more compactly than by just repeating the character symbol. Examples are the space

characters in text files, strings of zeros in business files and especially the graphic files.

Certain sequences of characters reappear with relatively high frequency and can therefore be represented with relatively fewer bits for a saving of space. A period followed by two spaces appears very frequently in texts and could be encoded using less number of bits.

If certain characters appear consistently at a predictable place in each block of data, then they are at least partially redundant. For instance, in data base files, certain record fields may almost always have the same entry, such as "none" in it. Text, on the other hand, has no positional redundancy in it.

In general, data compression techniques can be divided into two categories, the nonreversible and the reversible. In nonreversible techniques, the size of the physical representation of data is reduced while a subset of the information is preserved. For example, in some data sets "leading zeros" or "trailing blanks" may be considered as irrelevant information. Note that these techniques are, by definition, dependent on the semantics of the data.

In reversible techniques, all of the information is considered to be relevant, and compression/decompression process recovers the orijinal data. The reversible procedures can further divided into two groups, semantic independent and semantic dependent techniques. Semantic independent techniques can be used on any data with varying degrees of effectiveness. They do not use any information regarding the information content of the data. On the other hand, semantic dependent techniques depend on the context and semantics of the data to provide for redundancy reduction.

A code is a mapping of source messages into codewords. Codes can be categorized as block-block, block-variable, variable-block or variable-block, where block-block indicates that the source messages and codeword are of fixed length and variable-variable codes map variable-length source messages into variable-length codewords.

The oldest and most widely used codes, ASCII and EBCDIC, are examples of block-block codes, mapping an alphabet of 64 (or 256) single characters onto 6-bit (or 8-bit) codewords. These are not discussed, since they do not provide compression. Rather the other codes fall into our interest area.

Data compression schemes are also classified as either static or dynamic. A static method is one in which the mapping from the set of messages to the set of codewords is fixed before transmission begins, so that a given message is represented by the same codeword every time it appears in the message ensemble. In these methods, the mapping is based on the the probabilities with which the source messages appear in the source ensemble. Messages that appear frequently are represented by short codewords; messages with smaller

probabilities map to longer codewords. These probabilities are determined before transmission begins.

A code is dynamic if the mapping from the set of messages to the set of codewords changes over time. These methods involve computing an approximation to the probabilities of occurrence, as the ensemble is being transmitted. The assignment of codewords to messages is based on the values of the relative frequencies of occurrence at each point in time. A message may be represented by a short codeword early in transmission because it occurs frequently at the beginning of the ensemble, even though its probability of occurrence over the total ensemble is low. Later, when the more probable messages begin to occur with higher frequency, the short codeword will be mapped to one of the higher probability messages, and it will be mapped to a longer codeword. Dynamic codes are also referred to in the literature as adaptive, in that they adapt to changes in ensemble characteristics over time.

All of the adaptive methods are one-pass methods; only one scan of the ensemble is required. Static methods require two passes: one pass to compute probabilities and determine the mapping, and a second pass for transmission. Thus, as long as the encoding and decoding times of an adaptive method are not substantially greater than those of a static method, the fact that an initial scan is not needed implies a speed improvement in the adaptive case. In addition, the mapping determined in the first pass of a static coding method must be transmitted along with compressed ensemble by the encoder to the decoder.

There are two dimensions along which the data compression schemes may be measured: algorithm complexity and amount of compression. When data compression is used in a data transmission application, the goal is speed. Speed of transmission depends on the number of bits sent, the time required for the encoder to generate the coded message, and the time required for the decoder to recover the original ensemble. In a data storage application, although the degree of compression is the primary concern, it is nonetheless necessary that the algorithm be efficient in order for the schemes to be practical. For a static scheme, there are three algorithms to analyse: the map construction algorithm, the encoding algorithm, and the decoding algorithm. For a dynamic method, there are just two algorithms: the encoding algorithm and the decoding algorithm.

Several common measures of compression have been suggested: average message length and compression ratio. Average message length is clearly the average length of a coded message. Compression ratio is defined in several ways. The ratio of the length of the coded message to the length of the original ensemble gives the most common meaning.

Semantic dependent data compression techniques are designed to respond to specific types of applications, one of which is image representation and processing. One of these methods is run-length encoding where successive occurrences of a symbol is replaced by a single symbol along with its frequency. The other is difference mapping, in which the image is represented as an array of

differences in brightness (or color) between adjacent pixels rather than the brightness values themselves.

Semantic dependent data compression is also of interest in business data processing. The runs of zeros and blanks can compressed using run-length encoding. The other methods may be dictionary substitution, selection of common phrases, selection of common suffixes and prefixes and selection of common character pairs.

Semantic independent data compression techniques are based on the probabilities of messages in the ensemble. Some are static codings that are Shannon-Fano coding, static Huffman coding, universal coding and arithmetic coding. The others are dynamic ones that are dynamic Huffman coding, Lempel-Ziv coding and BSTW coding.

Shannon-Fano coding has the advantage of simplicity. The code is constructed as follows: the source messages and their probabilities are listed in order of nonincreasing probability. This list is then divided in such a way as to form two groups of as nearly equal total probabilities as possible. Each message in the first group receives 0 as the first digit of its codeword; the messages in the second half have codewords beginning with 1. Each of these groups is then divided according to the same criterion, and additional code digits are appended. The process is continued until each subset contains only one message.

Huffman coding takes the probabilities of source letters and constructs a full binary tree. Initially, there is a set of singleton trees, each having the probability of a source letter. At each step, the method merges the two smallest probabilities into a tree whose probability is the total of two probabilities and whose root has two children that are two probabilities. The process is continued until one tree whose probability is 1 is left. The codewords are assign to each message following the path from root to leaves, such as right child meaning a 0 and left child meaning a 1.

The advantage of universal codes is that it does not require the exact probabilities with which the source messages appear. By mapping messages in order of decreasing probability to codewords in order of increasing length, the universality can be achieved. Another advantage to universal codes is that the codeword sets are fixed. It is not necessary to compute a codeword set based on the statistics of an ensemble. Since the ranking of source messages is the essential parameter in universal coding, a universal code may be thought as representing an enumeration of the source messages or as representing the integers. The most commonly used universal codes are Elias codes and Fibonacci codes.

Arithmetic coding represents a source ensemble by an interval between 0 and 1 on the real number line. Each symbol of the ensemble narrows this interval. As the interval becomes smaller, the number of bits needed to specify it grows. A high probability message narrows the interval less than a low probability message, so that high probability messages contribute fewer bits to the coded message.

Adaptive Huffman coding determines the mapping from source messages to codewords on the basis of a running estimate of the source message probabilities. The code is adaptive and requires only one pass over the data. The encoder learns the characteristics of the source. The decoder must learn along with the encoder by continually updating the Huffman tree in order to stay in syncronization with the encoder.

BSTW coding uses a list as an auxillary data structure and shorter encodings for words near the front of this list. The message list is

updated at each transmission using the move-to-front method. When a message is to be transmitted, the sender transmits its current position on the list, and then updates the list by moving the message to position 1 and shifting each of the other messages down one position. The receiver alter its list similarly. For the transmission of positions, the universal codes may be used.

Lempel-Ziv coding defines the set of source messages as the algorithm executes. The Lempel-Ziv algorithm parses the source ensemble into a collection of segments of gradually increasing length. At each step, the longest prefix of the remaining ensemble that matches an existing table entry is parsed off, along with the character following this prefix in the ensemble. The new source message which is the concatenation of existing table entry and character is added to the code table. In the implementation of table, if tree structure is used, the algorithm speeds up. Also, the space needed by the algorithm is decreased.

BÖLÜM 1. GİRİŞ

1.1. Model

Veri sıkıştırma yöntemleri gereksiz veri miktarını azaltarak, verinin saklanması veya iletiminin verimini artıran yöntemlerdir. Bunlar veriyi sıkıştırarak, veri saklama birimlerinde daha az yer tutmasını veya iletişim kanallarında daha fazla bilgi iletilmesini sağlarlar.

Sıkıştırıcı, bir dizi sembolü veya veri karakterlerini kabul eden ve daha kısa sembollere çeviren yazılım parçası veya donanım aygıtıdır. Sıkıştırma algoritması bir metini, yani semboller dizisini girdi olarak alır ve karşılık gelen sıkıştırılmış metini üretir. Sıkıştırma algoritması, sıkıştırıcının izlediği yöntemdir (Şekil 1-1).

Şekil 1-1. Sıkıştırıcı.

Genişletici, sıkıştırılmış metini alarak orijinal haline getiren bir yazılım parçası veya donanım aygıtıdır. Genişletme algoritması, sıkıştırılmış metini girdi olarak alır ve orijinal kaynak metini çıktı olarak üretir. Genişletme algoritması, genişleticinin izlediği yöntemdir (Şekil 1-2).

Şekil 1-2. Genişletici.

Örnek olarak "aaaaaa" bilgisinin sıkıştırılacağını varsayalım. Sıkıştırmanın daha sonra anlatılacağı gibi yöntemlerinden bir tanesi, bilgiyi "6a" olarak sıkıştırmaktır. Görüldüğü gibi orijinal bilgideki 6 sembolün yerine aynı bilgi, 2 sembolle temsil edilmiş, yani sıkıştırılmıştır.

"6a" bilgisini tekrar orijinal hali olan "aaaaaa" bilgisine genişletebilmek için, sıkıştırıcı ile aynı yöntemi kullanan bir genişleticiye ihtiyaç vardır. Yani genişletici, bilginin nasıl sıkıştırıldığını bilip, bunun tam tersi işlemini yaparak orijinal bilgiyi elde etmelidir. Bunun için, her sıkıştırıcının kendine özgü bir genişleticisi vardır. Farklı genişleticiler, bir sıkıştırıcının sıkıştırdığı orijinal bilgiyi elde edemezler.

1.2. Amaç

İkincil bellekte saklanan verilerin sıkıştırılmış olarak tutularak daha az yer kaplaması veya iletişim alanında verinin sıkıştırılmış olarak iletilerek oldukça değerli olan bant genişliğinin tasarruflu olarak kullanılması amaçlarıyla veri sıkıştırma teknikleri uygulanır. Veri sıkıştırmanın ikincil bir hizmeti olarak, gizlilik de sayılabilir. İkincil belleğe ya da iletişim hattına sızılarak elde edilen verinin üçüncü kişilere anlamsız gelmesi sağlanarak, maliyet kazancının yanı sıra gizlilik de ikinci bir avantaj olarak sayılabilir.

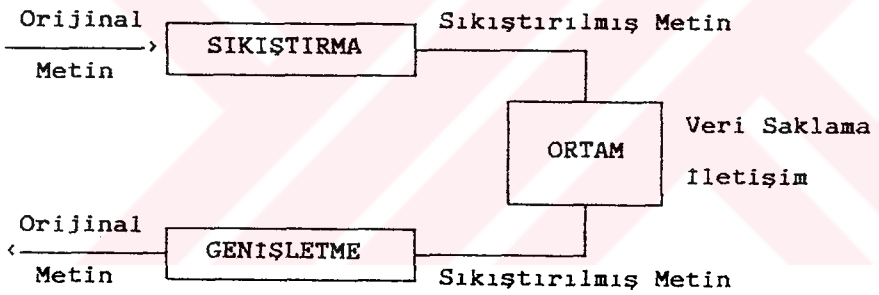
Veri sıkıştırma tekniklerini kullanmanın dezavantajları ise, veri sıkıştırma ve genişletmede harcanan zaman ile sıkıştırılmış veride olan herhangi bir hata durumunda veri kurtarmanın zorlaşması ve hatta bazı durumlarda olanaksız hale gelmesidir.

1.3. Uygulama

İletişim alanında veri sıkıştırma şu şekilde uygulanır : veriyi gönderen düğümde veri, sıkıştırıcı aracılığıyla sıkıştırılarak gönderilir; gönderilen düğümde sıkıştırılmış veri, genişletici aracılığıyla genişletilerek eski haline getirilir. İki yönlü veri iletişimi için, sıkıştıran ve genişleten birimler her iki düğümde de olmalıdır. Sıkıştırma ve genişletmede harcanan zaman, günümüzde

donanım hızı ile gözardı edilebilir, ama parazitli hatlar yüzünden sıkıştırılan veride oluşan hatalar oldukça önemlidir. Uydu aracılığıyla canlı olarak yayınlanan bir televizyon programındaki parazitler ya da resim donmaları bu hatalara örnek olarak gösterilebilir.

Veri saklamada ise veri sıkıştırma, genellikle büyük boyutlardaki verinin uzun vadeli saklanmasında kullanılır. Son yıllarda kalıcı bellekteki dosyaları sıkıştırılmış olarak saklayan işletim sistemleri de piyasaya çıkarılmıştır. Sıkıştırıcı, veriyi sıkıştırarak ikincil bellekte daha az yer kaplamasını sağlar. Veri gerektiği zaman, genişletici veriyi genişleterek kullanıma sunar. Sıkıştırma ve genişletme zamanı, dosyalar ikincil bellekde işletim sistemi tarafından sıkıştırılmış olarak tutulduğu zaman oldukça önemli olur. Hata ise, ikincil belleğin güvenilirliği ile ters orantılıdır (Şekil 1-3).



Şekil 1-3. Veri Sıkıştırma Modeli.

1.4. Bilgisayarda Bilgilerin Temsili

Bilgisayarlar ve iletişim sistemleri gibi sayısal ortamlarda, bilgiler sayısal olarak saklanır. Sayıların yanında, harfler ve bazı özel sembollerin sayısal ortamlarda temsili, ikili sayılarla yapılır. İkili veya iki tabanına göre olan sayı sistemi sadece 0 ve 1 sembollerinden oluşur. Bu sembollere ingilizce "ikili rakam" (binary digit) ibaresinin kısaltılmışı olan "bit" adı verilir. Bilgisayarın elektronik devrelerinde 0 biti düşük voltajla, 1 biti ise yüksek voltajla ifade edilir. İkili sistemde bir sayının değeri onluk sistemdeki gibi,

$$a_n \ 2^n + a_{n-1} \ 2^{n-1} + \ldots + a_1 \ 2^1 + a_0 \ 2^0$$

şeklinde ifade edilir. Örneğin, 1001 ikili sayısının onlu karşılığı,

$$1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 8 + 0 + 0 + 1 = 9$$

dur.

8 bitlik bellek birimine bayt denir. Bilgisayarlarda anlamı olan en küçük bilgi birimi baytdır. Her baytın bir sembole karşılık geldiği ASCII ve EBCDIC kodları artık standart hale gelmiştir. Bu kodlarda bir bayt ile ifade edilen 0 ve 255 arasındaki ikil sayılar bir sembole karşılık gelir. Bu sembollere karakter adı verilir.

1.5. Sıkıştırma

Bilgisayarlarda bilgiler genellikle bayt olarak saklanır. Bu yolla, birbirinden farklı 255 sembolün olduğu dosyalar saklanabilir. Eğer saklanan dosyalardaki birbirinden farklı sembol sayısı daha az ise sıkıştırma olasıdır. Örneğin dosyada bulunan birbirinden farklı sembol sayısı 32 ise, her sembol için 8 bit yani bir bayt yerine sadece 5 bit kullanmak yeterli olur. Örnek olarak, bir metinde sadece "a", "b", "c" ve "d" sembollerinin olduğunu varsayalım. Bu metin ASCII kodları ile saklandığı zaman, her karakter 8 bit olarak saklanır. Bu yolla 256 çeşit sembol saklanabilir. Oysa metinde sadece 4 sembol vardır. Bu yüzden her sembole 8 bit yerine, aşağıdaki gibi 2 bit atanabilir.

"a"	00
"b"	01
"c"	10
"d"	11

Böylece, 10 sembol uzunluğundaki "aabcdaabdd" karakter katarı, her bir karakter için 8 bit kullanılarak 80 bit uzunluğunda saklanırken, bu yolla her karakter için 2 bit kullanılarak 20 bit uzunluğunda saklanabilir.

Bazı dosyalarda bir ya da bir kaç karakter sık sık arka arkaya tekrarlanabilir. Örneğin, metin dosyalarında boşluk karakterinin arka arkaya sık tekrarlanması gibi. Bu gibi dosyalarda, arka arkaya gelen karakterler çok daha kısa bir şekilde saklanabilir. Bunu yaparken, arka arkaya gelen karakter sayısı sayılarak, karakterler yerine sadece sayısı ve o karakterden bir tane saklanır. Örneğin, 10 karakter uzunluğundaki "aaaaaaaaaa" karakter katarı 10 bayt olarak saklanırken, bu yolla 10 (tekrar sayısı) ve "a" olmak üzere 2 bayt olarak saklanabilir.

Çoğu sıkıştırma yöntemi sıkıştırılacak metini bir alfabeden seçilmiş karakter serisi olarak görür. Bu metinlerde, sık geçen karakterler sekizden az sayıda bit ile, nadir geçenlerde sekizden fazla bit ile değiştirilerek saklanırsa sıkıştırma sağlanabilir.

BÖLÜM 2. GENEL KAVRAMLAR

2.1. Giriş

Veri sıkıştırma genellikle şifreleme (veya kodlama) olarak anlaşılsa da, şifreleme gerekli verinin özel temsili için kullanılan genel bir terimdir. Enformasyon teorisi, verimli şifreleme ve sonucundaki iletişim hızı ve hata olasılığı çalışmaları olarak tanımlanabilir. Veri sıkıştırma, enformasyon teorisinin bir kolu olarak görülebilir. Amacı, iletilen veri miktarını en aza indirmektir.

Veri sıkıştırmanın bir özelliği, ASCII gibi belli bir koddaki karakter dizisini yine aynı bilgiyi taşıyan fakat mümkün olduğu kadar kısa yeni bir diziye çevirmektir. Veri sıkıştırma, veri iletimi ve veri saklama alanlarında önemli uygulamalara sahiptir. Bir çok veri işleme uygulamaları büyük oranlarda verinin saklanmasını gerektirir ve bilgisayarın kullanımı yeni alanlara doğru yayıldıkça bu uygulamaların sayısı devamlı olarak artmaktadır. Aynı zamanda bilgisayar iletişim ağlarının yaygınlaşması, iletişim hatlarında yüksek miktarda veri transferine neden olur. Saklanacak ya da iletilecek verinin sıkıştırılması saklama veya iletişim maliyetlerini azaltır. İletilecek verinin miktarı azaltıldığı zaman, sonuçta iletişim kanallarının kapasitesi artırılır. Benzer şekilde bir dosyanın orijinal boyutundan yarısına sıkıştırılması, saklama ortamı kapasitesini iki katına çıkarmak demektir. Bu yüzden, veriyi daha yüksek ve dolayısıyla daha hızlı bir saklama hiyerarşi düzeyinde saklamak ve bilgisayar sisteminin giriş/çıkış kanalları üzerindeki yükünü azaltmak mantıklı olacaktır.

2.2. Ana Kavramlar

Bu bölümde enformasyon teorisine kısa bir giriş yapılacaktır. Veri sıkıştırma yöntemlerinin değerlendirilmesi için gerekli tanımlar verilecektir.

Şifre (ya da kod), kaynak mesajların (A kaynak alfabesinden kelimelerin), kodkelimelere (B şifre alfabesindeki kelimelere) eşlenmesidir. Kaynak mesajlar, metinin parçalandığı temel birimlerdir. Bu temel birimler, kaynak alfabeden tek bir sembol veya sembol dizisi olabilirler. Örnek metin için kaynak mesajlar, {a,b,c,d,e,f,g, } kümesi sembolleri ya da bunların kombinasyonundan oluşan herhangi uzunluktaki semboller dizisi olabilir. Açıklama amacıyla şifre alfabesi, {0,1} kümesindeki semboller alınmıştır.

Şifreler, blok-blok, blok-değişken, değişken-blok veya değişken-değişken olarak sınıflandırılabilir. Blok-blok şifreler, kaynak mesajların ve şifre kelimelerin sabit uzunlukta olduklarını belirtirken, değişken-değişken şifreler, değişken uzunluktaki kaynak mesajları yine değişken uzunluktaki kodkelimelerine eşler. Tablo 2-1 ve Tablo 2-2 blok-blok ve değişken-değişken şifre örneklerini göstermektedir. Örnek metin, iki tablodaki gibi şifrelenirse, mesaj uzunlukları sırasıyla 120 ve 30 olur.

En eski ve yaygın şekilde kullanılan şifreler ASCII ve EBCDIC, 256 (veya 64) tek karakteri 8 (veya 6) bitlik kodkelimelerine eşleyen blok-blok şifre örnekleridir. Blok-blok şifreler, sıkıştırma sağlamadığından bu çalışmanın konusu değildir. Çalışmanın konusu blok-değişken, değişken-blok veya değişken-değişken şifreler olacaktır.

Değişken uzunlukta kaynak mesajlar olduğu zaman, kaynak metinin mesajlara nasıl ayrılacağı sorusu ortaya çıkar. Burada anlatılacak yöntemlerin çoğu kelime tanımlı tasarımlıdır. Yani, iletim ya da sıkıştırma yapılmadan önce, kaynak mesaj kümesi belirlenmiştir. Bu kaynak mesajlar, bir metin dosyası için alfabe harfleri, bir bilgisayar programı kaynak kodu için kullanılan programlama dilinin komutları olabilir. Serbest parçalama yöntemlerinde ise kaynak mesaj kümesi iletişim başladıktan sonra belirlenmeye başlanır.

Her kodkelimesi birbirinden ayrılabilen, yani kaynak mesajlarla kodkelimeler arasında birebir eşleme olan şifrelere belirgin şifre adı verilir. Her kodkelimesi, bir dizi kodkelimesi içinden ayrı olarak tanınabiliyorsa, belirgin şifre tek olarak deşifre edilebiliyor denir. Tablo 2-1 ve 2-2'deki şifreler belirgindir, fakat Tablo 2-2'deki şifre tek olarak deşifre edilemez. Örneğin, burada "11" olarak şifrelenmiş mesaj hem "ddddd", hem de "bbbbbb"

Tablo 2-1. Blok-blok şifre örneği.

Kaynak mesaj	Kodkelime
a	000
b	001
c	010
d	011
e	100
f	101
g	110
boşluk	111

Tablo 2-2. Değişken-değişken şifre örneği.

Kaynak mesaj	Kodkelime
aa	0
bbb	1
cccc	10
ddddd	11
eeeeee	100
fffffff	101
ggggggggg	110
boşluk	111

olarak deşifre edilebilir. Tek olarak deşifre edilebilir şifre, önek özelliğini taşıyorsa, yani hiçbir kodkelimesi bir diğerinin öneki değilse önek şifresi adını alır. Kodkelimeleri {1,10000,00} kümesinden oluşan şifre, tek olarak deşifre edilebilen bir şifre olduğu halde önek şifresi değildir. Önek şifreleri, anında deşifre edilebilirler, yani şifrelenmiş bir mesaj kodkelimelerine ileriye bakmadan ayrılabilir. {1,10000,00} kodkelimeleriyle kodlanmış bir mesajı açmak için ileriye bakmak gerekir. Örneğin, "1000000001" kodlanmış mesajının ilk kodkelimesi "1" dir. Fakat bu, mesajın son sembolü okunmadan anlaşılmaz (eğer sıfır sayısı tek rakam olsaydı, ilk kodkelimesi "100000" olurdu).

Bir önek şifresinde, eğer x bir kodkelimesinin önekiyse ve şifre alfabesindeki her q sembolü için xq ya bir kodkelimesi ya da bir kodkelimesinin öneki olduğu zaman, önek şifresi minimal olur. Örneğin, {00,01,10} minimal olmayan bir önek şifresi kodkelimeleridir. Bu örnekte, "1" in "10" kodkelimesinin öneki olması "11" in ya bir kodkelimesi ya da bir kodkelimesinin öneki olmasını gerektirir. Her ikisi de olmadığı için, bu önek şifresi minimal değildir. Bu minimal olma özelliği, şifrenin uzun olmasını

engeller. Bu örnekte, "10" kodkelimesi "1" ile değiştirilirse, daha kısa kodkelimelerle minimal önek kodu elde edilir.

2.3. Yöntemlerin Sınıflandırılması

Veri sıkıştırma yöntemleri mesaj ve kodkelimeleri uzunluklarından başka, statik veya dinamik olarak da iki grupta incelenebilir.

Statik yöntemler, mesaj kümesinden kodkelime kümesine eşlemenin iletişim (ya da sıkıştırma) başlamadan önce belirlendiği yöntemlerdir. Bunlarda, herhangi bir mesaj hep aynı kodkelimesiyle temsil edilir. Veri sıkıştırma klasiği olan Huffman [Huffman 1952] şifreleme statik bir yöntemdir. Huffman şifrelemesinde kodkelimelerin kaynak mesajlarla eşlenmesi, kaynak mesajların metindeki olasılıklarına göre yapılır. Metinde sık geçen (yüksek olasılıklı) mesajlar kısa kodkelimelere, nadir geçenler (düşük olasılıklı) uzun kodkelimelere atanır. Bu olasılıklar, sıkıştırma başlamadan önce metin bir kez okunduktan sonra belirlenir. Tablo 2-3, örnek metinin Huffman şifrelemesini göstermektedir. Bu şifrelemeyle metin, 117 bit ile temsil edilmektedir.

Tablo 2-3. Örnek metin için Huffman şifrelemesi.

Kaynak mesaj	Olasılık	Kodkelime
a	2/40	1001
b	3/40	1000
c	4/40	011
d	5/40	010
e	6/40	111
f	7/40	110
g	8/40	00
boşluk	5/40	101

Dinamik yöntemlerde ise, mesaj kümesinden kodkelime kümesine olan eşleme zamanla değişir. Örneğin dinamik Huffman şifrelemesinde, mesajların olasılıkları, o ana kadar olan geçiş sayılarına göre hesaplanır. Kodkelimelerin mesajlara eşlenmesi, herhangi bir zamandaki olasılık değerlerine göre devamlı olarak değişir. Bir mesaj iletişimin başında, metinin başlangıcında çok sayıda geçtiği için kısa kodkelimesiyle temsil edilebilir. Halbuki bu mesaj, bütün metinde çok düşük bir geçiş olasılığına sahip olabilir. İletişim

ilerledikçe, daha yüksek olasılıklı mesajlar geçtiği zaman, kısa kodkelimeleri yüksek olasılıklı mesajlara eşlenecek ve bu mesaj, metinin kalan kısmında nadir geçtiği için daha uzun kodkelimeleriyle temsil edilmeye başlanacaktır. Tablo 2-4, örnek metinin "aa bbb" önekine karşılık gelen dinamik Huffman şifrelemesi gösterir. Tabloda boşluk karakterinin frekansı, bütün metin boyunca "b" den büyük olduğu halde, bu noktada "b" daha yüksek frekansa sahiptir ve daha kısa kodkelimesiyle temsil edilmektedir.

Tablo 2-4. Örnek metinin "aa bbb" öneki için dinamik Huffman şifrelemesi.

Kaynak mesaj	Olasılık	Kodkelime
a	2/6	10
b	3/6	0
boşluk	1/6	11

Dinamik şifreler, metinin zamanla değişen karakteristiğine adapte oldukları için adaptif olarak da adlandırılırlar. Bazı adaptif yöntemler kaynak metindeki değişen kalıplara adapte olurlarken [Welch 1984], bazıları referans yerelliğine [Bentley 1986] adapte olurlar. Referans yerelliği, bir çok metinin ortak özelliği olarak görülen bazı kelime veya kalıpların kısa zaman aralıklarında sık geçmesi, sonra uzun süre kullanılmamasıdır.

Bütün adaptif yöntemler, tek-geçiş yöntemleridir, yani metinin sadece bir kez okunması yeterlidir. Statik yöntemler ise iki-geçiş gerektirir; biri mesajların metindeki olasılıklarını hesaplamak ve mesaj ile kodkelimelerin eşlemesini yapmak için, diğeri sıkıştırmayı yapmak için. Bu yüzden, adaptif yöntemlerin şifreleme ve deşifreleme süreleri statik yöntemlerinkinden çok daha büyük olmadıkça, bu yöntemler ilk okuma gerektirmediğinden zaman açısından iyileştirme sağlarlar. Ek olarak, statik şifrelemenin ilk geçişinde belirlenen eşleme, şifreleyiciden deşifreleyiciye gönderilmelidir. Bu eşleme, her dosya iletimi öncesi gönderilebilir veya bir eşleme üzerinde anlaşılıp bir kaç iletimde kullanılabilir. Tek-geçiş yöntemlerinde şifreleyici, iletim süresince eşlemeyi dinamik olarak tekrar tekrar tanımlar. Deşifreleyici de, kodkelimelerini aldıkça eşlemeyi tekrar tekrar tanımlar.

Bir yöntem, ne tamamen statik ne de tamamen dinamik olmayıp melez de olabilir. Basit melez bir şifrelemede, gönderici ve alıcı belli bir sayıda statik şifrenin bulunduğu birbirinin aynı şifre kitapları tutarlar. Her iletişim öncesi şifreleyici, şifre kitabındaki şifrelerden birini seçer ve seçtiği şifrenin adını ya da numarasını göndererek deşifreleyiciyi seçiminden haberdar eder. Deşifreleyici de, gelen kodkelimelerini seçilen şifreye göre deşifreler.

2.4. Veri Sıkıştırma Modeli

Veri sıkıştırma tekniklerinin birbirleriyle karşılaştırılmasında bazı kıstaslar gerekmektedir. Tekniklerin karşılaştırılmasında kullanılan Ölçülebilecek iki boyut vardır : algoritma karmaşıklığı ve sıkıştırma miktarı. Veri sıkıştırma veri iletiminde kullanıldığı zaman, amaç iletişimin en hızlı şekilde tamamlanmasıdır. İletişimin hızı da, gönderilen bit sayısına, şifreleyicinin mesajı şifrelemesi için gerekli zamana ve deşifreleyicinin orijinal mesajı elde etmesi için geçen zamana bağlıdır. Veri saklama uygulamalarında ise sıkıştırma derecesi ana kaygı olsa da, tekniğin kullanılması için algoritmanın da verimli olması önemlidir. Statik teknikler için üç algoritma vardır : eşleme bulma, şifreleme ve deşifreleme algoritmaları. Dinamik teknikler içinse algoritma sayısı ikiye düşer: şifreleme ve deşifreleme algoritmaları.

Çeşitli sıkıştırma ölçümleri önerilmiştir. Bunlar, gereksizlik [Shannon 1949], ortalama mesaj uzunluğu [Huffman 1952] ve sıkıştırma oranıdır [Davidson 1966; Rubin 1976; Ruth 1972]. Bu ölçülerin herbiri metin hakkında bazı varsayımlarda bulunur. Enformasyon teorisinde, genellikle kaynak mesajın bütün istatistiksel parametrelerinin tam olarak bilindiği varsayılır [Gilbert 1971]. En yaygın model, ayrık belleksiz kaynaktır. Bu kaynak, a1, a2, ... ,an sembollerinden oluşan sabit bir alfabeden seçilen sembol dizisidir. Bu semboller alfabeden, bazı sabit p(a1), ... , p(an) olasılıklarıyla istatistiksel açıdan bağımsız olarak rasgele seçilmiştir [Gallager 1968].

2.4.1. Gereksizlik ve Entropi

Veri sıkıştırıldığı zaman amaç, gereksizliği azaltarak sadece bilgiyi bırakmaktır. Bir kaynak mesaj x'in bilgi ölçüsü, -log p(x)'dır. Buradaki ve bundan sonra kullanılacak bütün logaritma fonksiyonlarında logaritma iki tabanına göredir. Burada p(x), x mesajının geçme olasılığıdır. p(x)=1 olduğu zaman, metinde sadece bu mesaj geçmesi gerektiğinden x hiç bilgi taşımaz. Benzer olarak p(x) küçük olduğu zaman, x mesajı daha az geçer ve daha çok bilgi taşır. Bütün kaynak alfabesinin ortalama bilgi miktarı, her kaynak harfinin bilgi miktarının geçme olasılığı kadar ağırlığını almakla bulunur. Böylece, $-\sum[p(x) \log p(x)]$ ifadesi elde edilir. Burada x, A kaynak mesaj alfabesinde bir semboldür. Bu ifade, şifrelenmiş mesaj için gerekli bit sayısına bir alt sınır koyan entropi kavramıdır [Shannon 1949]. Her mesajın kodkelimesinin uzunluğu, o mesajın bilgi miktarını taşımaya yetecek kadar olması gerektiğinden, bir şifrelenmiş mesajın boyutu entropiden küçük olamaz. Hiçbir sıkıştırma yöntemi, böyle bir modelde bir metini bilgi kaybı olmaksızın entropisinden daha az bite indiremez. Şifrelenmiş mesajın toplam bit sayısı en az, entropi ile metinin mesaj uzunluğunun çarpımı kadar olabilir.

Ek A, Pascal dilinde yazılmış entropi hesaplayan bir program içermektedir. Bu program, mesajları bayt olarak kabul ederek, verilen dosyadaki mesajların frekanslarını saymakta ve yukarıdaki entropi formülüne göre dosyanın entropisini hesaplamaktadır. Bu program C dilinde Thomas [Thomas 1991] tarafından yazılmıştır.

Bir metinin bilgi gereksizliği,

$$\sum[p(x)\ l(x)] - \sum[-p(x)\ \log p(x)]$$

ifadesi olarak tanımlanır. Burada l(x), x mesajını temsil eden kodkelimesinin uzunluğudur. $\sum[p(x)\ l(x)]$ ifadesi, kodkelime uzunluklarının, geçiş olasılığıyla ağırlığını yani, ortalama kodkelime uzunluğunu verir. $\sum[-p(x) \log p(x)]$ ise ortalama bilgi miktarıdır. Böylece gereksizliğin, ortalama kodkelime uzunluğu ile ortalama bilgi miktarı farkı olduğu bulunur. Eğer şifre, bir ayrık olasılık dağılımı için en az ortalama kodkelime uzunluğunu veriyorsa, bu şifreye minimum gereksizlik şifresi denir.

Statik olasılık modelleri bir çok metini iyi karakterize edemez. Örneğin, İngilizce'de "e" harfi "u" harfinden daha sık geçtiğinden, statik olasılık modeli "qe" nin "qu" dan daha sık geçmesini bekler. Oysa durum tam tersidir. Markov olasılık modelleri bu tür metinleri çok iyi karakterize eder. Markov modellerinde her harf geldiğinde bu harften sonra gelebilecek bütün harfler ve olasılıkları bulunur. Böylece, bir İngilizce metinde "q" harfinden sonra en yüksek olasılıkla "u" harfinin gelebileceğini Markov modeli tahmin edebilir [Gallager 1968; Jones 1988].

Veride dört tip gereksizlik bulunabilir [Welch 1984]. Bu gereksizlik tipleri veride, yalnız başına ya da birkaçı birden bir grup halinde görülebilir. Veri sıkıştırma yöntemleri, verideki bu gereksizlikleri yok etmeye çalışırlar. Bazı yöntemler sadece bir gereksizlik üzerinde konsantre olurlarken, bazıları birkaçına birden adapte olabilirler. En iyi yöntem, bütün gereksizlikleri yok eden yöntemdir. Fakat bu, teoride mümkün olsa da, uygulamada aşırı bellek ve zaman istediğinden imkansızdır.

2.4.1.1. Karakter Dağılımı

Tipik bir karakter katarında, bazı karakterler diğerlerinden daha sık olarak kullanılır. Örneğin bir metin dosyasında, 8-bit ASCII kodlamasında 256 kodun yaklaşık 3/4'ü kullanılmaz. Böylelikle, 8 bitin yaklaşık 2 bitinden tasarruf edilebilir, bu da yaklaşık %25 zaman ve yer tasarrufu demektir.

Dosyalardaki karakter dağılımı, dosyanın özelliğine göre değişir. Metin dosyalarında harfler, ticari dosyalarda sayılar, diğerlerinde başka karakterler sık olarak geçebilir. Tablo 2-5, Snyderman ve

Tablo 2-5. İngilizce metinlerde her bin harfte görülen harf frekansı.

A - 82	J - 1	S - 61
B - 14	K - 4	T - 105
C - 28	L - 34	U - 25
D - 38	M - 25	V - 9
E - 131	N - 71	W - 15
F - 29	O - 80	X - 2
G - 20	P - 20	Y - 20
H - 53	Q - 1	Z - 0.7
I - 63	R - 68	

Hunt [Snyderman 1970] tarafından elde edilen, İngilizce metinlerde görülen harf frekans dağılımını göstermektedir. Tabloda görüldüğü gibi, "E" harfi İngilizce'de en sık kullanılan harftir.

Tablo 2-6. Türkçe metinlerde her bin harfte görülen harf frekansı.

A - 113	I - 53	R - 75
B - 27	İ - 84	S - 34
C - 10	J - 1	Ş - 16
Ç - 10	K - 42	T - 36
D - 44	L - 71	U - 28
E - 89	M - 44	Ü - 16
F - 5	N - 67	V - 10
G - 14	O - 30	Y - 30
Ğ - 10	Ö - 8	Z - 13
H - 9	P - 11	

Tablo 2-6 ise, Türkçe metinlerde rastlanılan harf frekans dağılımını göstermektedir. Bu tablo, çeşitli kaynaklardan sağlanan 2 megabayt uzunluğundaki Türkçe metinlerin, Ek B'deki Pascal bilgisayar programlama dilinde yazılan bir program tarafından işlenmesi ile oluşturulmuştur. Program, bütün metinlerin içinde toplandığı bir dosyayı girdi olarak almış ve içindeki bilgileri bayt olarak okuyarak, 0 ile 255 arasındaki baytların frekanslarını hesaplamıştır. Daha sonra bu bayt frekanslarından, Türkçe harfleri de göz önüne alarak, harf frekanslarının toplam harf sayısına oranını bulmuştur. Bunu yaparken İngiliz alfabesinde olup Türkçede olmayan "q", "w" ve "x" harfleri hesaplamalarda kullanılmamış; bunun yerine İngilizcede bulunmayan, Türkçedeki "Ç", "ç", "Ğ", "ğ", "İ", "ı", "Ö", "ö", "Ş", "ş", "Ü" ve "ü" harfleri kullanılmıştır. Ayrıca hesaplarken, "i" ve "ı" harflerinin büyükleri sırasıyla "İ" ve "I" olarak alınmıştır. Halbuki İngilizcede, "i" harfinin büyüğü "I" dır ve diğer harfler ise hiç yoktur.

2.4.1.2. Karakter Tekrarı

Bir karakter arka arkaya çok sayıda tekrar ettiği zaman mesaj, karakterin tekrarıyla daha sıkışık olarak şifrelenebilir. Arka arkaya karakter tekrarlarına, genellikle resim (imaj) dosyalarında rastlanılır. Metin dosyalarında, boşluk karakterinden başka karakterler pek arka arkaya görülmezler. Bununla beraber, ticari dosyalarda alfabetik alanlarda boşluk karakteri, sayısal alanlarda da sıfır karakterine arka arkaya rastlanılabilir.

2.4.1.3. Çok Kullanılan Kalıplar

Bazı karakter serileri, metinde oldukça sık geçebilirler. Bu yüzden bunlar, diğerlerine oranla daha az sayıda bit ile temsil edilerek zaman ve yer açısından tasarruf sağlanabilir. Örneğin, metin dosyalarında virgül ve boşluk karakter dizisi diğer ikili karakter dizilerinden daha sık geçer ve bu yüzden daha az sayıda bit kullanmak için tekrar şifrelenebilir. "ze" gibi çoğu harf çiftleri, tek harf olasılıklarından daha çok geçebilirler ve bu yüzden iki harf sembolünün şifrelendiğinden daha az sayıda bit ile tekrar şifrelenebilirler. Benzer olarak, "gc" gibi nadir geçen harf çiftleri, daha verimli bit kullanımı sağlamak için daha fazla sayıda bit ile şifrelenebilirler. Metinlerin bazı bölümlerinde, belli kelimeler çok sayıda kullanılır. Örneğin bu çalışmada, "şifre" ibaresi çok sık geçer. Bu metin sıkıştırılırken, "şifre" ibaresi kısa bir şifre ile temsil edilebilir.

2.4.1.4. Konumsal Gereksizlik

Eğer belli karakterler bir veri bloğunun bilinen bir yerinde devamlı olarak geçiyorsa, o zaman en azından kısmi olarak gereksizdir. Örneğin bazı ticari veri dosyalarında belli kayıt alanlarının değeri belli bir kümeden seçiliyor olabilir : cinsiyet alanının erkek ya da kadın, meslek alanının doktor, mühendis, avukat ve benzeri olması gibi.

2.4.2. Ortalama Mesaj Uzunluğu

Bir kodun verimliliğinin ölçüsü olarak Huffman [Huffman 1952], $\sum[p(x)\ \ l(x)]$ ifadesiyle verilen ortalama mesaj uzunluğunu kullanmıştır. Bu miktar aslında şifrelenmiş mesajın, yani kodkelimelerin ortalama uzunluğudur. Gereksizlik, ortalama kodkelime uzunluğu ile entropi arasındaki fark olduğundan ve entropi verilen bir olasılık dağılımına göre sabit olduğundan, ortalama kodkelime uzunluğunu en aza indirmek, gereksizliği en aza indirmek anlamına gelir.

Eğer belli bir olasılık dağılımı için entropi sonsuza doğru giderken, ortalama kodkelime uzunluğunun entropiye oranı 1'e

yaklaşırsa, o şifreye asimtotik olarak optimal şifre denir. Optimallik, ortalama kodkelime uzunluğunun teorik minimuma yaklaşmasını sağlar.

2.4.3. Sıkıştırma Oranı

Bir şifreleme tekniğinin sıkıştırma miktarı, sıkıştırma oranı ile ölçülebilir. Sıkıştırma oranı, çeşitli şekillerde tanımlanmıştır.

S = (ortalama mesaj uzunluğu / ortalama kodkelime uzunluğu)

şeklindeki tanımı en yaygın anlamını, yani orijinal mesajın şifrelenmiş mesaja olan kıyaslanmasını vermektedir [Cappellini 1985; Davidson 1966]. Örnek metinin 6 bitlik ASCII karakterlerden oluştuğunu varsayarsak, ortalama mesaj uzunluğu 6 bitdir. Huffman şifrelemesi bu metini 117 bit ile şifrelemiştir. Bu da ortalama kodkelime uzunluğunu 2.9 bit olarak verir. Burada sıkıştırma oranı 6/2.9, yani 2'den fazla olmuştur. Ya da başka bir deyişle, metin %50'sinden daha az uzunluktadır veya %50'den fazla bir sıkıştırma sağlanmıştır.

Sıkıştırma oranının başka bir tanımı da,

S = (K-R-OR)/S

olarak Rubin [Rubin 1976] tarafından verilmiştir. Bu tanımda K metin uzunluğu, R şifrelenmiş mesajın uzunluğu ve OR kod eşlemesinin uzunluğu, yani şifreleyicinin deşifreleyiciye şifre eşlemesini iletmesi için gerekli bit sayısıdır. OR miktarı, şifre eşlemesinin deşifreleyiciye iletmesini için gerektiren statik yöntemler için geçerlidir. Bu formüldeki amaç, iletimin veya saklanan dosyanın toplam boyutunu ölçmektir.

BÖLÜM 3. SIKIŞTIRMA YÖNTEMLERİ

3.1. Giriş

Sıkıştırma yöntemleri genel olarak iki gruba ayrılır. Birincisi, gürültülü, diğeri gürültüsüz yöntemlerdir. Gürültülü yöntemler, iletişim sırasında veride meydana gelebilecek hata ya da kayıplara tolerans gösteren yöntemlerdir. Bunlar genellikle parazitli iletişim hatlarında görüntü iletiminde kullanılabilirler. İletişim sırasında hattın parazitli olmasından dolayı sıkıştırılan görüntüde bazı bozulmalar meydana gelebilir. Bu yöntemler sıkıştırılmış görüntüyü orijinal görüntüye yakın bir görüntüye genişletirler. Gürültüsüz yöntemler ise, veri saklama gibi güvenilir ve kayba tahammülsüz ortamlarda kullanılır. Sıkıştırılmış metin genişletilirken orijinal metinin aynısı elde edilir. Bu çalışma, sadece gürültüsüz yöntemleri içine almaktadır. Gürültüsüz yöntemler de, içerik-bağımlı, statik ve dinamik olmak üzere üç başlık altında anlatılacaktır. Yöntemlerin açıklanmasında aşağıdaki örnek karakter dizisi kullanılacaktır.

"aa bbb cccc ddddd eeeeee fffffff gggggggg"

3.2. İçerik Bağımlı Yöntemler

Bu yöntemler, çeşitli uygulamalarda görülen bazı yerel gereksizlik tipleri için kullanılırlar. Bunlar ilginç ve kendi alanlarında değer taşıyan yöntemlerdir. En büyük dezavantajları, sınırlı sayıda uygulamalarda kullanılabilmeleridir. Programlandıkları tipte dosyalardaki gereksizliklere çok iyi adaptasyon sağladıklarından, bu tip dosyalarda genel sıkıştırma yöntemlerinden çok daha büyük başarı sağlarlar.

3.2.1. Resim İşleme

Resim verisinin sıkıştırılmasında kullanılan bir teknik, tekrar uzunluğu şifrelemesidir. Yaygın olarak kullanılan bu yöntemde, bir satırdaki

x(1),x(2),...,x(n)

dizisini, c(i)'nin koyuluk ya da renk ve 1(i)'nin de c(i) eşit koyuluk ya da renkteki arka arkaya gelen piksel sayısını temsil ettiği

[c(1),1(1)],[c(2),1(2)],...,[c(k),1(k)]

çiftler dizisine eşlenir. Yani, sıkıştırma anında arka arkaya gelen semboller sadece sembol ve tekrar sayısı ile değiştirilir. Genişletirken ise tersi işlem olan sayı kadar sembol eklenir.

Ek C ve D, Pascal dilinde yazılmış en küçük bilgi birim bayt olan tekrar uzunluğu sıkıştırması ve genişletmesi yapan bilgisayar programlarını göstermektedir. Sıkıştırma programı bir dosyayı girdi olarak almakta ve üçten fazla arka arkaya rastlanan karakterleri, sayı işareti, karakter sayısı ve karakterin kendisiyle değiştirmektedir. Metinde geçen sayı işaretlerini de, diğerlerinden ayırmak için arka arkaya iki tane olarak yazmaktadır. Üçten az arka arkaya rastlanan karakterleri aynen yazmaktadır. Bu metin, arka arkaya üç karakterden fazla dizi taşımayan ve sayı işaretinin çok olduğu dosyaların uzunluğunu artırır. Genişletme programı, sıkıştırma programının tam ters işlemini yapar. Uydu haritaları gibi resimlerde tekrar uzunluğu büyük oranda sıkıştırma sağlar [Gonzalez 1977].

Resim verisine özel diğer bir veri sıkıştırma yöntemi de, fark eşlemesidir. Bu yöntemde, koyuluk ya da renk yerine, bitişik iki piksel arasındaki parlaklık farkları dizisi resmi temsil eder. 256 renk ya da koyuluk düzeyi 8 bit ile belirtirken, bu yöntemle pikseller ortalama 3 bit ile temsil edilebilmektedir [Laeser 1986].

Diğer resim sıkıştırma yöntemleri Wilkins [Wilkins 1971] ve Cappellini [Cappellini 1985]'de bulunabilir.

3.2.2. Ticari Veriler

Ticari veri işlemede de veri sıkıştırma, veri işleme ve maliyet açısından önemlidir. Bu tip dosyalardaki yerel gereksizlik tipleri, sayısal alanlardaki sıfırlar dizisi karakter alanlardaki boşluk dizileri ve bazı alanların var olup olmamalarıdır. Sıfır ve boşluk dizilerini sıkıştırmada, tekrar uzunluğu şifrelemesi kullanılabilir. Mevcut biti yardımıyla boş olan alanların sıkıştırılması sağlanabilir [Ruth 1972]. Sözlük yerleştirme ise, banka hesabı tipi, cinsiyet, ay adı gibi sınırlı sayıda değer alabilen alanları bir kaç bit ile değiştirme sağlayan başka bir sıkıştırma yöntemidir [Reghbati 1981].

3.2.3. Metin dosyaları

Metin dosyalarına uygulanabilen bir çok yöntem vardır. Bunlardan biri, sözlük kullanarak metinde geçen bütün ya da bazı yaygın olarak kullanılan sözcükleri, sözlüğe gösterge ile şifrelemektir [Hahn 1974; Tropper 1982]. Yaygın kullanılan deyimler [Wagner 1973], ekler [Fraenkel 1983] ve karakter çiftlerinin [Cortesi 1982; Snyderman 1970] şifrelenmeleri de başka uygulamalardır.

3.3. Statik Yöntemler

Veri sıkıştırmanın klasiği sayılan Huffman şifrelemesi [Huffman 1952] 40 yıl önce geliştirildi. Huffman şifrelemesi, minimum gereksizlik şifrelerinin kurulmasına ilk çözümü getirmiştir. Uzun yıllar Huffman şifrelemeden daha iyi sıkıştırmayı bir şifrelemenin olamayacağı düşünülmüştür. Bundan daha önce, birbirinden ayrı olarak Shannon [Shannon 1949] ve Fano [Fano 1949] tarafından geliştirilen bir yöntem Huffman'dan daha kötü sonuçlar vermiştir. Huffman yöntemi, daha sonra geliştirilen yöntemlere kıyaslama için bir temel oluşturmuştur ve bu alanda yazılan makalelerin hemen tamamı tarafından referans olarak kullanılmıştır. Huffman ve Shannon-Fano şifrelemeleri, kaynak mesajların nasıl tanımlandığına bağlı olarak blok-değişken veya değişken-değişken olabilirler.

Başka bir statik şifreleme tekniği, tamsayıların ikili kodkelimelere eşlendiği evrensel şifrelemelerdir. Herhangi bir sınırlı alfabe numaralandırılabildiği için, bu şifrelendirme genel amaçlı bir kullanım sunar.

Aritmetik şifreleme ise, veri sıkıştırmaya diğer statik yöntemlerden farklı şekilde yaklaşım göstermiştir. Bu yöntem, kaynak mesajların kodkelimelere eşlendiği bir şifre yaratmaz. Bunun yerine, diğer yöntemlerin yaptığı gibi kodkelimelerin yanyana gelmesiyle oluşan şifre gibi olmayan, bir şifre dizisiyle kaynağı temsil eder. Aritmetik şifreleme, kaynağın entropisine oldukça yakın sonuçlar elde eder.

3.3.1. Shannon-Fano Şifrelemesi

Bu tekniğin en büyük avantajı basitliğidir. Bu yöntemde şifre oluşturma algoritması şöyledir: Kaynak mesajlar ve olasılıkları, olasılıkların artmayan sırasıyla listelenir. Bu liste, alt ve üst tarafta yaklaşık eşit toplamlı olasılıkların bulunduğu iki gruba bölünür. İlk grupta bulunan mesajlara kodkelimelerinin ilk rakamı olarak "0", ikinci gruptakilere de "1" verilir. Daha sonra bu iki grup, yine aynı kritere göre ikiye bölünerek ek şifre rakamları eklenir. Bu işlem, bütün gruplarda sadece bir mesaj kalana kadar sürdürülür. Kolayca görüldüğü gibi, Shannon-Fano şifrelemesi minimum gereksizlik şifresi elde eder.

Tablo 3-1, yöntemin basit bir olasılık dağılımı için uygulamasını göstermektedir. Burada herhangi bir x mesajının kodkelimesinin uzunluğu -log(x)'dir. Bu ancak, listenin devamlı iki eşit olasılıklı gruplara bölündüğü zaman gerçekleşebilir. Bu mümkün

Tablo 3-1. Bir Shannon-Fano şifrelemesi.

Kaynak mesaj	Olasılık	Kodkelime
m(1)	1/2	0
m(2)	1/4	10
m(3)	1/8	110
m(4)	1/16	1110
m(5)	1/32	11110
m(6)	1/64	111110
m(7)	1/64	111111

olamadığı zaman, bazı kodkelimelerin uzunluğu bunun bir fazlası olabilir. Shannon-Fano şifrelemesi, alt sınırı entropi, üst sınırı entropinin bir fazlası olan bir ortalama kodkelime uzunluğu elde eder. Tablo 3-2, örnek metin için Shannon-Fano şifrelemesini göstermektedir. Buna göre, örnek metin 117 bit ile temsil edilir. Bu örnekte olduğu gibi, genellikle Huffman şifreleme ile aynı ortalama kodkelime uzunluğu elde eder. Fakat, {0.35,0.17,0.17,0.16,0.15} mesaj olasılık kümesiyle Huffman şifrelemesinden daha kötü sonuçlar alır.

Tablo 3-2. Örnek metin için Shannon-Fano şifrelemesi.

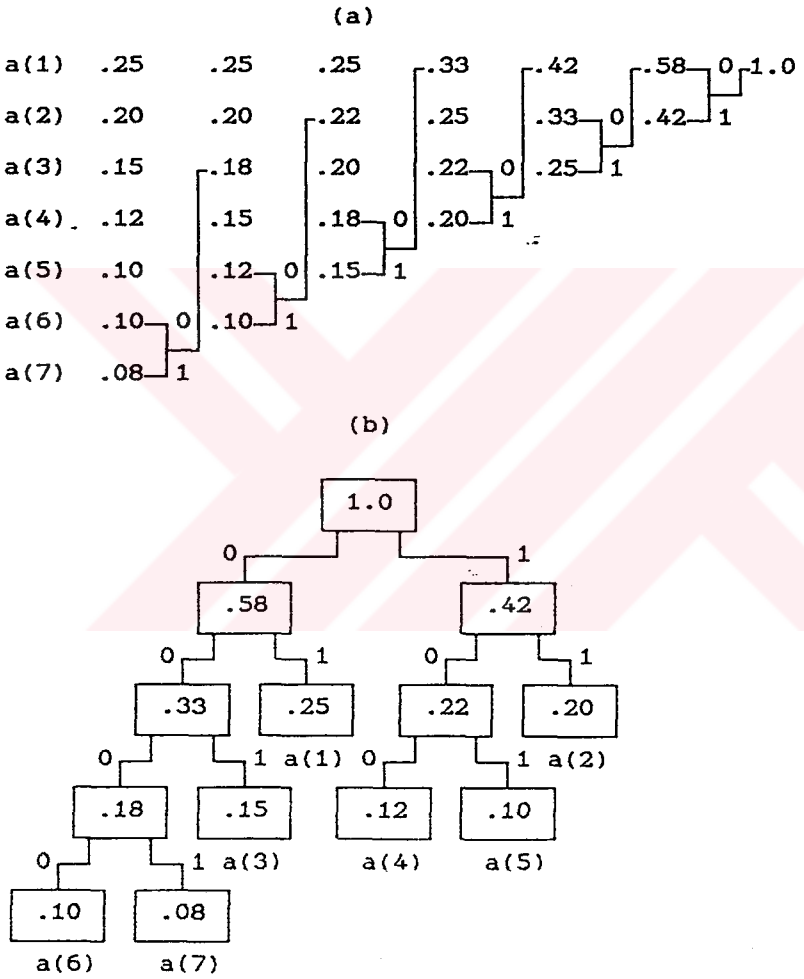
Kaynak mesaj	Olasılık	Kodkelime
g	8/40	00
f	7/40	010
e	6/40	011
d	5/40	100
boşluk	5/40	101
c	4/40	110
b	3/40	1110
a	2/40	1111

3.3.2. Huffman Şifrelemesi

Huffman yöntemi, {a(1),a(2),...,a(n)} pozitif ağırlıklar dizisi alır ve yaprakları ağırlık olan bir tam ikili ağaç kurar. Tam ikili ağaç, her düğümü ya iki çocuğu olan, ya da hiç çocuğu olmayan ağaçtır. Başlangıçta, listede her ağırlık için tekli ağaçlar kümesi vardır. Şifre oluşturma algoritmasının her adımında, ağırlıkları en küçük ağırlıklı iki ağaç birleştirilerek, ağırlığı bu ağaçların ağırlıkları toplamı olan ve kökü bu iki ağaç tarafından temsil edilen altağaçlara sahip yeni bir ağaç oluşturulur. Birleştirilen ağaçlar listeden çıkarılarak, yeni ağaç listeye eklenir. Bu işlem, listede sadece bir ağaç kalana kadar sürdürülür. Eğer herhangi bir anda en küçük ağırlık çiftini seçmede birden fazla yol varsa, bu çiftlerden herhangi biri seçilebilir. Listenin sıralı olması gerekmez, ama sıralı olması kolaylık yaratır.

Bu algoritma, her mesaja karşılık gelen kodkelime uzunluklarını belirler. Kodkelimelerinin seçiminde bir çok yol vardır. Bunların hepsi önek özelliğinde şifre oluşturur. Normal belirlemede, soldaki ağacın bağına "0" rakamı, sağdakine "1" rakamı verilir. Her mesaj

için kodkelimesi, kökten o harfi temsil eden yaprağa giden yoldaki rakamlardan oluşur. Şekil 3-1'deki olasılık listesi için Huffman kodkelimeleri {01,11,001,100,101,0000,0001} kümesi olur. Bu işlem minimum gereksizlik şifresi elde eder.



Şekil 3-1. Liste (a) ve ağaç (b) için Huffman işlemi.

Kodkelimelerini kurmada bir çok yol olduğundan dolayı, Huffman şifreleme her zaman aynı uzunlukta kodkelimeleri elde etmeyebilir. Örnek olarak, {0.4,0.2,0.2,0.1,0.1} mesaj olasılık kümesiyle, kodkelime uzunlukları {1,2,3,4,4} ve {2,2,2,3,3} olan şifreler çıkarılabilir. Fakat bu şifrelerin her ikisi de, aynı ortalama kodkelime uzunluğunu sağlarlar. Schwartz [Schwartz 1964a], yeni bir ağırlığı her zaman listede aynı ağırlıkda olanların üzerine yerleştiren ve listenin en altındaki iki ağırlığı birleştiren bir Huffman varyasyonu önermiştir. Böylelikle bulunan şifreler, maksimum uzunluktaki kodkelimelerin minimumuna ve minimum kodkelime uzunluğu toplamına sahiptir. Schwartz ile Kallick [Schwartz 1964b] ve Connel [Connel 1973], Huffman algoritmasının birbirini takip eden ikili şifreler üreten varyasyonunu önermişlerdir.

Daha önce belirtildiği gibi, Shannon-Fano yönteminin gereksizlik üst sınırı 1'dir. Huffman yönteminde bu, minimum kaynak mesaj olasılığı ile 0.086 toplamına eşittir [Gallager 1978]. Minimum kaynak mesaj olasılığının maksimum değeri 0.5'dir ve genellikle çok daha küçük, hatta sıfıra yakındır. Gereksizliğin tanımında, bu yöntemler için şifre eşlemenin iletilmesi gözardı edilmiştir. Fakat gerçek uygulamalarda şifre eşlemenin de iletilmesi gerekmektedir ve statik yöntemlerde hesaplanan entropi miktarında, şifre eşleme faktörünün de eklenmesi gerektiği akıldan çıkarılmamalıdır.

Ek E ve F, sırasıyla Huffman sıkıştırması ve genişletmesi yapan Pascal dilinde yazılmış iki programı içerir. Ek E'deki sıkıştırıcı program, bir metini girdi olarak alarak, ilk geçişte metindeki baytların frekanslarını bularak Huffman ağacını oluşturur. İkinci geçişte ise, metini sıkıştırarak, sıkıştırılmış metini kaynak metinindeki baytların frekansları ile beraber saklar. Ek F'deki genişletici program ise, önce baytların frekanslarından Huffman ağacını oluşturur ve ardından metini tek geçişte genişleterek orijinal metini oluşturur.

3.3.3. Melez Yöntemler

Eğer optimal şifrelemeden biraz daha kötü sonuçlar veren bir yöntem kullanılmak istenirse, şifre eşlemenin iletilmesi maliyeti, gönderici ve alıcı arasında üzerinde önceden anlaşılan bir şifre eşlemesi kullanılarak yok edilebilir. O anki mesajın Huffman şifrelemesi yerine, kullanılan şifre o anki mesajın karakteristiğine

uygun bir mesaj grubu için olan istatistiklere uygun olabilir. Yani, gönderici ve alıcı herbiri değişik gruptaki mesajların istatistiklerine uygun şifrelemeler içeren bir şifre kitabı kullanabilirler. Bu durumda gönderici, basitçe bu ortak şifrelerden hangisini kullandığını çok az bit ile göndererek şifre eşlemesini göndermekten kurtulabilir.

Ayrıca, bu melez teknik kullanılarak eşleme hesaplaması zamanı maliyeti de en aza iner. Yani, istatistiksel açıdan örnek bir dosya için eşleme bir kere yapılır ve bu, bütün o gruptaki dosyalar için kullanılabilir. Yalnız bu yöntemde örnek dosya seçiminin temsil ettiği grubu tam olarak karakterize edememesi ve herhangi bir dosyanın o grup dosyaların olasılık dağılımına uymaması gibi çeşitli riskler vardır.

3.3.4. Evrensel Şifrelemeler

Ortalama kodkelime uzunluğu üst sınırının aH+b ile sınırlı olduğu kaynak mesajların kodkelimelerine eşlendiği şifrelere evrensel şifreleme adı verilir. Yani evrensel şifreleme, optimal şifreden a kere daha uzun bir ortalama kodkelime uzunluğu elde eder. Bu şifrelemenin performansı a ve b sabitlerine bağlıdır. Eğer a sabiti 1 ise, o evrensel şifreleme optimaldir.

Evrensel şifrelemenin Huffman'a karşı üstünlüğü, kaynak mesajların tam olasılıklarının bilinmesinin gerekmemesidir. Bu şifrelemede yalnızca mesajların olasılık sırasının bilinmesi yeterlidir. Evrensellik, azalan olasılık sırasına göre dizilmiş mesajların, artan uzunluktaki kodkelimelere eşlenmesiyle sağlanır. Bu şifrelemenin iyiliği, mesajların olasılıkları değişse bile, olasılık sırası değişmedikçe kodkelimelerin sabit kalmasıdır.

Evrensel şifrelemenin ana parametresi kaynakların sıralanması ve numaralandırılması olduğu için, bu şifreleme tamsayıların şifrelendirilmesi olarak da düşünülebilir.

3.3.4.1. Elias Şifrelemesi

Elias [Elias 1975], tamsayılar kümesini ikili kodkelimelerine eşleyen bir dizi evrensel şifreleme tekniği bulmuştur. Bunlardan birincisi basittir, fakat optimal değildir. Bu şifreleme, bir x tamsayısını,

$$\lfloor \log x \rfloor$$

değeri kadar öneki sıfır ve x'in ikili değerine eşler. x'in ikili değeri en az sayıda bit ile ifade edilir, bu yüzden 1 ile başlar ve sayıyı önekden ayırır. Sonuçta, anında deşifre edilebilen bir şifre elde edilir. Burada, kodkelimenin uzunluğu başındaki sıfır sayısının iki katından bir fazladır ve ilk 1 rakamına ulaşıldığında kodkelimenin uzunluğu anlaşılır. Bu şifre minimum gereksizlik şifresi değildir, çünkü kodkelimesinin entropiye oranı entropi sonsuza giderken 2'dir. Tablo 3-3, bazı tamsayılar için birinci Elias şifresini vermektedir.

Tablo 3-3. Birinci Elias şifresi.

Tamsayı	Kodkelime
1	1
2	010
3	011
4	00100
5	00101
6	00110
7	00111
8	0001000
16	000010000
32	00000100000

İkinci Elias şifresi bir önceki Elias şifresini kullanır. Bu şifre x tamsayısını,

$$\lfloor \log x \rfloor + 1$$

değerinin ilk Elias şifresindeki karşılığı ile, başındaki "1" rakamı silinmiş durumda olan x'in ikili değerine eşler. Kodkelimenin uzunluğu,

$$\lfloor \log x \rfloor + 2 \lfloor \log(\lfloor \log x \rfloor + 1) \rfloor + 1$$

olur. Ortalama kodkelime uzunluğunun entropiye oranı 1 olduğundan bu şifre optimaldir. Tablo 3-4, bazı tamsayılar için ikinci Elias şifresini vermektedir. Tablo 3-5, örnek metinin ikinci Elias şifrelemesini göstermektedir. Bu şifrelemede metin 161 bitle temsil edilmiştir. Bu Huffman'a göre kötü bir sonuçtur.

Tablo 3-4. İkinci Elias şifresi.

Tamsayı	Kodkelime
1	1
2	0100
3	0101
4	01100
5	01101
6	01110
7	01111
8	00100000
16	001010000
32	0011000000

Tablo 3-5. Örnek metinin İkinci Elias şifrelemesi.

Kaynak mesaj	Frekans	Sıra	Kodkelime
g	8	1	1
f	7	2	0100
e	6	3	0101
d	5	4	01100
boşluk	5	5	01101
c	4	6	01110
b	3	7	01111
a	2	8	001000000

3.3.4.2. Fibonacci Şifrelemesi

Fibonacci sayılarını temel alan ikinci evrensel şifreleme Apostolico ve Fraenkel [Apostolico 1985] tarafından tanımlanmıştır. Fibonacci şifreleri optimal olmadığı halde, birbirinden ayrı mesaj sayısı az olduğu zaman Elias şifrelerinden daha iyi sonuç verir.

Fibonacci şifreleri dizisi, derecesi ikiden büyük Fibonacci sayılarını temel alır. İkinci dereceden Fibonacci sayıları, 1,1,2,3,5,8,13, .. dır. Derecesi m olan Fibonacci sayıları şöyle tanımlanmıştır: F(-m+1)'den F(0)'a kadar olan Fibonacci sayıları 1, diğerleri ise kendinden önceki m Fibonacci sayısının toplamıdır.

Burada sadece 2. dereceden Fibonacci şifresi anlatılmıştır. Diğer dereceden olan şifreler aynı yolla çıkarılabilir.

Fibonacci şifresinde her pozitif tamsayı aşağıdaki şekilde sadece bir ikili sayı tarafından temsil edilir:

$$\sum_{i=0}^{k}[d(i)F(i)]$$

Burada d(i), {0,1} kümesinden, F(i) de ikinci dereceden Fibonacci sayılarıdır. Dikkat edileceği gibi, bu temsillerde hiç yanyana "1" rakamı bulunmaz. Bu temsilin önek özelliği yoktur. Fibonacci şifresi, bu sayının rakamlarının ters sırada yazılımının sonuna "1" rakamı eklenmesiyle elde edilir. Tablo 3-6, bazı tamsayılar için Fibonacci şifresi verilmiştir. Bu kodkelimeleri önek özelliği taşır. Çünkü, her kodkelimesi iki ardışık "1" rakamı ile biter, bu da kodun başka hiçbir yerinde geçmez.

Tablo 3-6. Tamsayıların Fibonacci temsili ve Fibonacci şifresi.

Tamsayı	Fibonacci temsili							Kodkelime
1							1	11
2						1	0	011
3					1	0	0	0011
4					1	0	1	1011
5				1	0	0	0	00011
6				1	0	0	1	10011
7				1	0	1	0	01011
8			1	0	0	0	0	000011
16		1	0	0	1	0	0	0010011
32	1	0	1	0	1	0	0	00101011
	21	13	8	5	3	2	1	

Fibonacci şifresi evrenseldir ve aH+b ifadesinde a sabiti 2, b sabiti 3'dür. Daha yüksek dereceli Fibonacci şifreleri, kaynak mesaj sayısı çok ve olasılık dağılımı düzgün olduğu zaman, ikinci dereceli şifreden daha iyi sonuç verirler [Fraenkel 1985]. Tablo 3-7, örnek metin için Fibonacci şifrelemesini göstermektedir. Bu şifre ile metin, 153 bit ile temsil edilir. Bu Elias şifresinden iyi, fakat Huffman'dan daha kötüdür.

Tablo 3-7. Örnek metinin Fibonacci şifrelemesi.

Kaynak mesaj	Frekans	Sıra	Kodkelime
g	8	1	11
f	7	2	011
e	6	3	0011
d	5	4	1011
boşluk	5	5	00011
c	4	6	10011
b	3	7	01011
a	2	8	000011

3.3.5. Aritmetik Şifreleme

Bu yöntem, Abramson [Abramson 1963] tarafından kitabında sunulmuştur. Daha sonra bir çok kişi tarafından geliştirilmiştir [Rissanen 1976; Pasco 1976; Rubin 1979; Witten 1987].

Aritmetik şifrelemede kaynak metin, sayı doğrusundaki [0,1) aralığıyla belirlenir. Metindeki her sembol, bu aralığı daraltır. Aralık daraldıkça, onu belirten bit sayısı artar. Bu şifreleme, kaynağı temsil etmek için kaynak mesajların olasılıklarını kullanarak aralığı daraltır. Yüksek olasılıklı bir mesaj, aralığı düşük olasılıklı bir mesajdan daha az daraltır ve böylece şifrelenmiş mesaja daha az sayıda bit ekler. Yöntem, kaynak mesajların sıralı olamayan bir listesi ile başlar. Sayı doğrusu, kümülatif olasılıklara göre alt aralıklara bölünür.

Aritmetik şifreleme bir örnek ile şöyle açıklanabilir. {A,B,C,D,#} kaynak mesajları ve sırasıyla {0.2,0.4,0.1,0.2,0.1} olasılıklarıyla Tablo 3-8, sayı doğrusunun başlangıçtaki bölünüşü göstermektedir.

Tablo 3-8. Aritmetik şifreleme.

Kaynak mesaj	Olasılık	Aralık
A	0.2	[0,0.2)
B	0.4	[0.2,0.6)
C	0.1	[0.6,0.7)
D	0.2	[0.7,0.9)
#	0.1	[0.9,1)

"A" sembolü [0,1) aralığının ilk 1/5'i olan [0,0.2), "B" sonraki 2/5'i olan [0.2,0.6), "C" sonraki 1/10'u olan [0.6,0.7), "D" sonraki 1/5'i olan [0.7,0.9) ve "#" kalan 1/10'u olan [0.9,1) aralıkları ile temsil edilir. Şifreleme başladığı zaman, bütün metin [0,1) aralığıyla temsil edilir. "AADB#" metini için, ilk "A", aralığı [0,0.2)'ye ve ikinci "A", [0,0.04)'e düşürür. Sonraki "D", aralığı [0.028,0.036)'ya, "B", [0.0296,0.0328)'e ve "#", son aralık olan [0.03248,0.328)'e indirir. Bu son aralık ya da aralığın içindeki herhangi bir sayı "AADB#" metinini temsil eder.

Bu aralık azaltma işlemi, aralığın sol sınırı ve aralığın yeni boyutu değerlerinin belirlenmesidir. Bunlardan aralığın sol sınırı, önceki aralığın sol sınırına, o anki mesajın sol sınırıyla önceki aralığın boyutu çarpımının eklenmesiyle bulunur. Örnekte başlangıç aralık [0,1)'dir. İlk "A", aralığın sol sınırını 0 + 0 x 1, yani 0 yapar. Aralığın boyutunu ise, önceki aralığın boyutu ile o anki mesajın aralığının boyutunun çarpımı belirler. İlk "A" için bu 1 x 0.2, yani 0.2'dir. Yani ilk "A", aralığı [0,0.2) yapar.

Orijinal mesajı elde edebilmek için, deşifreleyici şifreleyicinin kullandığı kaynak modelini ve şifreleyicinin daha önce anlatıldığı gibi bulduğu son aralıkda bulunan tek bir sayı alır. Deşifreleme yapmak, iletilen sayının kaynak mesaj aralıklarından hangisinde olduğunun anlaşılması için bir dizi karşılaştırma yapmaktır. Verilen örnekte bu sayı 0.0325 olabilir. Deşifreleyici, bu sayıyı kullanarak şifreleyicinin hareketlerine benzer hareketler yapar. 0.0325 sayısı, A mesajının aralığı olan [0,0.2) aralığında olduğundan ilk mesaj "A"dır. Bu, aralığı [0,0.2)'ye düşürür. Deşifreleyici, bundan sonraki mesaj "A" ise aralığın [0,0.04), "B" ise [0.04,0.12), "C" ise [0.12,0.14), "D" ise [0.14,0.18) ve "#" ise [0.18,0.2) olacağını anlar. Sayı 0.0325 olduğundan, ikinci mesajın "A" olduğu anlaşılır. Bu işlem, bütün metin elde edilinceye kadar sürdürülür.

En son alt aralığın boyutu, o aralıkdaki bir sayıyı belirtmek için gerekli bit sayısını belirtir. [0,1) aralığının s boyutundaki bir alt aralığı, -log(s) sayıda bit ile temsil edilebilir. Son aralığın boyutu kaynak mesajların olasılıklarının çarpımlarına eşittir. Yani,

$$s = \prod_{i=1}^{n} [p(\text{kaynak mesaj } i)] \qquad n=\text{mesaj sayısı}$$

Bu ifade, tam olarak entropiye eşit olarak bulunur [Lelewer 1987]. Yani aritmetik şifreleme, diğer yöntemlerden daha iyi sonuçlar elde eder. Örnek metinin entropisi 115.7'dir. Yani, aritmetik şifreleme bu metini 116 bite sıkıştırır.

Aritmetik şifrelemenin uygulamasında bazı sorunlarla karşılaşılmaktadır. Bunların ilki, deşifreleyicinin nerede durması gerektiğini bilmemesidir. Gerçekten de 0 sayısı yukarıdaki örnekte aynı zamanda "A", "AA", "AAA", yani bir çok sayıda "A"nın yanyana geldiği mesajı belirtir. Bu sorunun çözümü, şifrelenmiş mesajla beraber orijinal mesaj uzunluğunun da gönderilmesidir. Başka bir çözüm de bir mesaj sembolünün mesaj sonu belirtmesidir. İkinci sorun ise, yukarıda anlatımdan en son aralık belli olana kadar, şifreleyicinin hiç mesaj gönderemediği gibi görünmektedir. Oysa durum böyle değildir. Aralık daraldıkça, aralık sınırlarını belirleyen sayıların solundaki rakamlar sabit kalmaktadır. Üçüncü sorun, üretilen sayının çok basamaklı olması ve bu sayıları işlemedeki zorluktur.

3.4. Dinamik Yöntemler

Bu yöntemler, veri üzerinden sadece bir kez geçerek, kaynak metinin değişen özelliklerine, devamlı olarak eşleme tablosunu değiştirerek adaptasyon sağlayan yöntemlerdir. Bu yöntemlerin kaynak noktası yine Huffman şifreleme olmuştur.

Adaptif Huffman şifreleme, Huffman yöntemine benzer olarak, Huffman ağacı kullanır. Aradaki fark ağacın devamlı olarak değişmesidir. Ağacın değiştirilme algoritması, yöntemin çeşitli varyasyonları olmasına yol açmıştır.

Diğer adaptif yöntem Lempel-Ziv şifrelemesi, Lempel ve Ziv [Ziv 1977] tarafından bulunmuştur. Algoritma Welch [Welch 1984] tarafından geliştirilmiştir ve LZW algoritması adıyla anılmaya başlanmıştır. Değişken-blok tipinde şifrelemedir. Değişken uzunluktaki mesajları, sabit uzunlukta kodkelimelere eşler. Regan [Regan 1990], değişken-değişken tipinde bir varyasyonunu önermiştir.

3.4.1. Adaptif Huffman Şifrelemesi

Bu şifrelemede, o ana kadar geçen mesaj olasılıklarına göre kaynak mesajlar kodkelimelere eşlenir. Şifreleme, kaynağın yerel özelliklerine cevap vererek, yerel gereksizliği de bir anlamda yok etmeye çalışır. Bu bir anlamda, kaynak karakteristiğinin öğrenilmesi demektir. Deşifreleyici, şifreleyici ile eş hareketle Huffman ağacını aynı anda değiştirerek öğrenir. Bu şifrelemenin üstünlüğü, veri üzerinden sadece bir kez geçmesidir. Ayrıca adaptif olduğundan, eşleme tablosunun da iletilmesi gerekmemektedir.

Adaptif Huffman şifrelemesi, ilk olarak birbirinden ayrı olarak Faller [Faller 1973] ve Gallager [Gallager 1978] tarafından önerilmiş ve algoritma Knuth [Knuth 1985] tarafından geliştirilmiştir. Bu algoritmanın temelini Gallager 'ın tanımladığı kardeş özelliği oluşturur. Bir ikili şifre ağacı, eğer kök düğümü hariç bütün düğümlerinin kardeşi varsa ve düğümler yanındaki kardeş düğümüyle beraber, artmayan ağırlıklarla sıralanabiliyorsa kardeş özelliğine sahiptir. Gallager, bir ikili önek şifresinin sadece ve sadece şifre ağacı kardeş özelliğindeyse, Huffman şifresi olduğunu kanıtlamıştır. Bu yöntemde, şifreleyici ve deşifreleyici sürekli değişen Huffman şifre ağacı tutarlar. şifre ağacının yaprakları kaynak mesajları ve yaprakların ağırlıkları da mesajların geçme frekanslarını temsil ederler. Herhangi bir anda n olası mesajdan k tanesi geçer.

Başlangıçta, şifre ağacı 0-düğümü adı verilen tek yaprak düğümden oluşur. Bu 0-düğümü, o ana kadar kullanılmayan mesajları temsil eden özel bir düğümdür. Bir mesaj iletildiği anda, her iki taraf ağırlıkları artırmalı ve yeni ağırlıklara göre kardeş özelliğinin korunması için şifre ağacını tekrar hesaplanmalıdır. t sayıda mesaj iletildiği anda, bunlardan k tanesi birbirinden farklıysa, Huffman şifre ağacında k+1 yaprak vardır. Bunlardan k tanesi birbirinden farklı mesajlar için, 1 tanesi ise 0-düğümü, yani o ana kadar hiç kullanılmamış mesajlar içindir. Eğer bundan sonra gönderilen mesaj, daha önce kullanılmış olan k mesajdan biriyse, bu mesajın kodkelimesi gönderilir ve frekansı artırılarak şifre ağacı yeniden hesaplanır. Eğer daha önce kullanılmayan bir mesaj kullanılırsa, 0-düğümü ikiye ayrılarak biri yeni mesaj, diğeri 0-düğümü olan bir çift yaprak yaratılır ve ağaç tekrar hesaplanır. Bu durumda, 0-düğümünün kodkelimesi ve yeni mesaj gönderilir.

Vitter [Vitter 1987] bu algoritmaya ağacın tekrar hesaplanması üzerine bir yenilik getirmiştir. Buna göre, tekrar hesaplamada ağaç üzerindeki düzenleme işlemleri azalmış ve en uzun kodkelime minimum değerine indirilmiştir.

3.4.2. Lempel-Ziv Şifrelemesi

Bu yöntem, sabit bir kaynak mesaj setinden sabit kodkelime setine eşleme yapan klasik şifrelemelerden ayrılır. Algoritma icra edildikçe kaynak mesaj ve kodkelime kümelerinin değiştiği bu tip şifrelemelere serbest parçalamalı adı verilir. Lempel-Ziv şifrelemesi kaynak mesaj kümesini mesaj üzerinden geçerken tanımlar.

Lempel-Ziv yöntemi, sonlu bir alfabeden sembol katarlarını uzunluğu belli bir tamsayıyı aşmayan alt karakter katarları veya kelimelere ayıran bir kural ile bu alt karakter katarlarını sabit uzunluktaki tam olarak deşifre edilebilen kodkelimelerine eşleyen bir şifrelemeden oluşur [Ziv 1977]. Karakter katarları geçiş olasılıkları yaklaşık olarak birbirlerine eşit olacak şekilde seçilir. Bunun sonucu sık geçen sembollere uzun karakter katarlarında, az geçenlere ise kısalarında raslanır. Bu yaklaşım, sembol frekansı, karakter tekrarı ve sık kullanımlı kalıplar olarak ortaya çıkan gereksizlikleri hedef almada etkilidir. Tablo 3-9, küçük bir Lempel-Ziv şifre tablosunu göstermektedir. Bu tabloda 'z' harfi gibi düşük frekanslı harfler, tek başına sabit uzunluktaki kodkelimlerine eşlenmiştir. Boşluk ve sıfır gibi sık geçen semboller ise uzun karakter katarlarında görülürler. Etkili sıkıştırma uzun bir karakter katarı, tek bir 12 bit uzunluğundaki kodkelime ile temsil edildiğinde gerçekleştirilir.

Lempel-Ziv yöntemi, değişik kaynak metin karakteristikleri için şifrelemenin öğrenme ile birlikte olduğu bir sembol çözümleme stratejisidir. Tablo 3-9'de görüldüğü gibi, arka arkaya gelen sıfırlar ve boşluklar ögrenilmiştir.

Lempel-Ziv algoritması, kaynak metini uzunluğu giderek artan parçalara çözümler. Şifrelemenin her adımında, varolan tablodaki a mesajına eşit olan kalan metinin en uzun öneki, bu önekten sonra gelen c sembolüyle beraber çözümlenir. Yeni kaynak mesaj ac, şifre tablosuna eklenir. Bu yeni tablo elemanı, i'nin varolan tablo

Tablo 3-9. Lempel-Ziv şifre tablosu.

Sembol katarı	Şifre
"A"	1
"O"	2
"AT"	3
"AD"	4
"AN"	5
"ADI"	6
"O"	7
"OO"	8
"OOO"	9
" "	10
" "	11
" "	12
" "	13
"Z"	14

elemanının kodkelimesini ve c'nin eklenen sembolü gösterdiği (i,c) şeklinde şifrelenir. Tablo 3-10, örnek metinin Lempel-Ziv tablosunu göstermektedir. Şifrelenmiş mesaj, {(0,a),(1,boşluk),(0,b),(3,b),(0,boşluk),(0,c),(6,c),(6,boşluk),(0,d),(9,d),(10,boşluk),(0,e),(12,e),(13,e), (5,f),(0,f),(16,f),(17,f),(0,g),(19,g),(20,g),(20)} şeklindedir. Burada mesaj tablosu, Tablo 3-9'dekinden daha verimli şekilde gösterilmiştir. Karakter katarı, önek kodkelimesi ve karakter biçiminde gösterilerek, tablo elemanlarının sabit uzunlukta olması sağlanmıştır.

Tablo 3-10. Örnek mesaj için Lempel-Ziv şifresi.

Mesaj	Kodkelime
a	1
1boşluk	2
b	3
3b	4
boşluk	5
c	6
6c	7
6boşluk	8
d	9
9d	10
10boşluk	11
e	12
12e	13
13e	14
5f	15
f	16
16f	17
17f	18
g	19
19g	20
20g	21

Lempel-Ziv yöntemi, sabit uzunlukta kodkelimelerine eşleme yapar. Tablonun büyüklüğü kodkelimelerin uzunluğu tarafından belirlenir. Algoritmanın tanımından anlaşıldığı gibi, Lempel-Ziv şifrelemesi metinlerin başlangıç bölümlerinde oldukça verimsizdir. Örneğin, a-g karakterleri ve boşluk için 3 bitlik ve tablo indisleri için 5 bitlik kodkelimeleri kullanılırsa, örnek metin 173 bit ile şifrelenir ki, bu diğer yöntemlerden daha kötü sonuçlar verir. Bu yöntemin iyi sonuç vermesi için metinin yeterince uzun olması gerekir.

Eğer kodkelime uzunluğu yeterince değilse, Lempel-Ziv şifreleri bir süre sonra makul bir verimliliğe varır, kısa bir süre iyi bir sıkıştırma performansı yakalar, tablo dolduğunda da yeni mesajlar eklenemez ve daha fazla kazanç sağlayamaz. Eğer metinin karakteristiği zamanla değişirse, yöntem metinin yeni karakteristiğine adapte olamaz.

Lempel-Ziv şifresi asimtotik olarak optimaldir. Kaynak mesajın uzunluğu sonsuza giderken gereksizlik sıfıra yaklaşır. Algoritma başladığı zaman, her kaynak sembolü tek olarak şifrelenir. Bu, 8 bitlik kaynak sembolleri ve 12 bitlik kodkelimeleri için başlangıç şifrelemesinde %50'lere varan bir büyüme yapacaktır. Bu büyüme, bütün kaynak sembollerin tabloda bir eleman olarak tanımlanmalarıyla önlenebilir.

Lempel-Ziv yöntemi üzerinde çeşitli iyileştirmeler yapılmıştır [Storer 1982; Rodeh 1981; Rissanen 1983; Welch 1984]. Storer ve Szymanski veri tekrarlarının önüne geçen ve daha hızlı çalışan bir algoritma önermişlerdir. Rodeh, Pratt ve Even sonek ağaçlarına bağlı bir uyarlama ile algoritmayı hızlandırmışlardır. Welch karakter katarı tablosu saklanması ve erişiminde bazı yenilikler getirmiştir. Bu son uyarlama, UNIX compress [UNIX 1984] ve PKARC [PKARC 1987] sıkıştırma programlarında kullanılmıştır.

3.4.3. BSTW Şifrelemesi

Bu algoritma, Bentley, Sleator, Tarjan ve Wei [Bentley 1986] tarafından 1986 yılında önerilmiştir. Bu yöntem, verinin üzerinden sadece bir geçiş yapma avantajına sahiptir ve iki geçişli statik yöntemlere iletilen bit sayısı boyutu bakımından üstünlük gösterir.

BSTW algoritması, referans yerelliği avantajına da sahiptir. Algoritma, kendiliğinden düzenlenen bir listeyi veri yapısı olarak kullanır. Listenin önündeki semboller için daha kısa şifreler kullanılır. Listenin düzenlenmesinde öne taşıma stratejisi kullanılır.

BTSW algoritmasında diğer adaptif yöntemlerde olduğu gibi, gönderici ve alıcı şifrenin aynı şekilde temsil edilmesini sağlarlar ve mesaj listeleri her iletişimde öne taşıma yöntemini kullanarak güncellenir. Başlangıçta listelerde hiç eleman yoktur. Bir a mesajı iletileceği zaman, eğer a mesajı göndericinin listesinde varsa, gönderici onun listede o anki konumunu gönderir. Daha sonra a mesajını listenin birinci konumuna taşır ve diğerlerini bir konum aşağı kaydırır. Alıcı da listesini aynı şekilde değiştirir. Eğer a mesajı ilk kez iletiliyorsa, o ana kadar iletilen birbirinden farklı mesaj sayısı k ise, k+1 değeri iletilir. Sonra mesajın kendisi ilk ve son olmak üzere bir kez gönderilir. Son olarak alıcı ve gönderici a mesajını listelerinin başına eklerler. Örnek olarak, "abcadeabfd" mesajının iletimi "1a2b3c34d5e356f5" şeklinde olur.

Örnekte görüldüğü gibi, BSTW algoritması her kaynak mesajı bir kere iletir, daha sonra sadece listedeki konumlarını gönderir. Bu yüzden algoritmanın önemli bir özelliği listedeki konumları gösterecek tamsayıların akılcı bir şekilde temsil edilmesidir. Bunun için Elias şifreleri [Elias 1975] kullanılabilir.

Örnek metin BSTW algoritmasıyla "1a12boş1uk3b1124c11125d111126e1111127f1111118g1111111" şeklinde şifrelenir. Her kaynak mesaj için 3 bit ve liste konumları için Elias şifresi kullanılırsa, mesaj 81 bit ile ifade edilebilir.

BÖLÜM 4. LEMPEL-ZIV ALGORİTMASI

4.1. Giriş

Lempel-Ziv veri sıkıştırma yöntemi [Ziv 1977], ilk olarak 1977 yılında ve algoritmaya Welch'in katkısı [Welch 1984] ise 1984 yılında yayınlandı. Algoritma oldukça basittir. Yöntem temelde, değişken uzunluktaki karakter katarlarını tek bir şifre ile değiştirir. Metinin hiç bir ön analizini yapmaz. Bunun yerine, her yeni geçen karakter katarını bir tabloya yerleştirir. Sıkıştırma, bir karakter katarı tek bir şifre ile değiştirildiğinde gerçekleşir.

Algoritmanın ürettiği şifre herhangi bir uzunlukta olabilir. Fakat şifrenin uzunluğunun, tek bir karakterin bit sayısından daha fazla sayıda bitten oluşması gerekir. Eğer bir karakter ASCII şifresinde olduğu gibi 8 bit ile ifade edilirse, ilk 256 şifre standart karakter kümesi için kullanılır. Kalan şifreler, algoritma icra edildikçe karakter katarlarına atanır. Örneğin, 12 bit uzunluğunda şifreler kullanılırsa, 0-255 arasındaki şifreler standart karakter kümesine, kalan 256-4095 arası şifreler ise karakter katarları için kullanılır.

4.2 Sıkıştırma Algoritması

Algoritma Şekil 4-1'de gösterilmektedir. Bu algoritmada, o ana kadarki karakter katarının tabloda olup olmadığı kontrol edilmektedir. Karakter katarı tabloda yoksa katar tabloya yeni bir katar olarak eklenmekte, eğer varsa bir sonraki karakter de katara eklenmekte ve bu şekilde bütün metin işlenene kadar devam etmektedir.

Algoritma öncesi standart karakter kümesi tablonun ilk 256 elemanına atanır. Algoritmada KATAR değişkeninin değeri her zaman katar tablosunun bir elemanının değeriyle aynıdır. KATAR'ın şifresi

```
Standart karakter kümesini ilk 256 elemana ata
KATAR = okunan karakter
WHILE girdi karakterleri var DO
    KARAKTER = okunan karakter
    IF (KATAR + KARAKTER) tabloda var THEN
        KATAR = KATAR + KARAKTER
    ELSE
        KATAR'ın şifresini yaz
        (KATAR + KARAKTER) i katar tablosuna ekle
        KATAR = KARAKTER
    ENDIF
ENDWHILE
KATAR'ın şifresini yaz
```

Şekil 4-1. Lempel-Ziv sıkıştırma algoritması.

elemanın bu tablodaki değeridir. KARAKTER değişkeninin değeri ise, bir sonraki girdi karakteridir. Algoritma devamlı olarak metinin karakterlerini, KATAR+KARAKTER değeri tabloda bulunduğu sürece KATAR'a ekler. KATAR+KARAKTER değeri tabloda olmadığı zaman KATAR'ın şifresi, yani KATAR'ın değerine sahip tablo elemanının konumu çıktıya yazılır ve KATAR+KARAKTER değeri tabloya eklenir. Böylelikle algoritma yeni katarlar öğrenerek, daha sık geçen katarları daha az sayıda bit ile değiştirerek yüksek oranda sıkıştırma sağlar (Şekil 4-1).

Algoritmanın çalışmasını gösteren kısa bir karakter katarı Şekil 4-2'de gösterilmektedir. Örnek girdi karakter katarı "/"

Girdi Metin : "/KAÇ/KAL/KAN/KAR/KARA"

Girdi Karakterleri	Çıktı Şifreler	Katar Tablosu
/K	/	256 = /K
A	K	257 = KA
Ç	A	258 = AÇ
/	Ç	259 = Ç/
KA	256	260 = /KA
L	A	261 = AL
/	L	262 = L/
KAN	260	263 = /KAN
/	N	264 = N/
KAR	260	265 = /KAR
/	R	266 = R/
KARA	265	267 = /KARA
katar sonu	A	

Şekil 4-2. Örnek Lempel-Ziv sıkıştırma uygulaması.

karakterleriyle ayrılmış kısa Türkçe keliimelerden oluşmaktadır. Algoritma başladığı zaman, ilk döngüde "/K" katarının tablodaki varlığı araştırılır. Bulunamadığı için de "/"ye karşılık gelen şifre çıktı olarak verilir ve "/K" katarı tabloya eklenir. Standart karakter kümesi olan 256 karakter 0-255 arası şifrelere atandığı için, ilk katar 256 şifresine atanabilir. Daha sonra "A" karakteri okunur, "KA" katarı tabloya eklenir ve "K" çıktı olarak verilir. Bu işlem, ikinci kelimenin başlangıcında "/K" okunup tablonun 256. elemanı olduğu bulununcaya kadar böyle devam eder. O zaman, "/K" katarının tablodaki konumu 256 çıktı olarak verilir ve 3 karakter uzunluğundaki katar tabloya eklenir. Böylece ilk sıkıştırma sağlanmış olur. Algoritma girdi karakter katarı bitinceye kadar devam eder.

Örnekte görüldüğü gibi, karakter katarı tablosu oldukça hızlı büyümektedir. Çıktı 9 karakter ve 4 şifreden oluşmaktadır. Her karakter için 8 bit kullanıldığında 168 bit ile ifade edilen bilgi, algoritmada her şifre için 9 bit kullanılırsa 117 bite indirilebilmektedir. Örnek girdi, algoritmanın sıkıştırma performansını gösterme amacıyla seçildiği için, kısa bir karakter katarında iyi bir sıkıştırma sağlanmıştır. Normal yaşamda, metinlerde sıkıştırma daha geç başlar. Bu yüzden algoritma kısa metinlerde kötü performans gösterebilir.

4.3 Genişletme Algoritması

Genişletme algoritması, sıkıştırma algoritmasının tam tersidir. Girdi olarak sıkıştırma algoritmasının oluşturduğu şifreleri alır ve şifrelere karşılık gelen karakter katarlarını çıktı olarak verir. Bu algoritmada katar tablosunun sıkıştırıcıdan genişleticiye aktarılmasına gerek yoktur. Sıkıştırma sürecinde oluşturulan katar tablosu, genişletici tarafından aynen genişletme sürecinde oluşturulur.

Algoritma aynı sıkıştırmada olduğu gibi, her yeni şifreyi okuduğunda karakter katarı tablosuna yeni bir karakter katarı ekler. Tek yapması gereken her girdi şifresini bir karakter katarına çevirmek ve bunu çıktı olarak vermektir. Genişletme sırasında sorun çıkaran sadece bir durum vardır. Eğer (KATAR,KARAKTER)'den oluşan bir karakter katarı daha önceden tabloda varsa ve girdide KATAR,KARAKTER,KATAR,KARAKTER,KATAR dizisi ardışık olarak geliyorsa,

sıkıştırma algoritması genişleticinin daha tanımlamadığı bir şifreyi çıktı olarak yaratır. Bu, genişletme algoritmasının karşılaştığı tek tanımlanmamış şifre durumudur ve algoritmaya bir IF bloğu konarak çözümlenmiştir (Şekil 4-3).

```
Standart karakter kümesini ilk 256 elemana ata
ŞİFRE'yi oku
ŞİFRE'yi yaz
WHILE girdi şifreleri var DO
    YENİŞİFRE'yi oku
    IF YENİŞİFRE tabloda yok THEN
         KATAR = ŞİFRE'nin karakter kataı karşılığı
         KATAR = KATAR + KARAKTER
    ELSE
         KATAR = YENİŞİFRE'nin karakter kataı karşılığı
    ENDIF
    KATAR'ı yaz
    KARAKTER = KATAR'ın ilk karakteri
    ŞİFRE + KARAKTER'i tabloya ekle
    ŞİFRE = YENİŞİFRE
ENDWHILE
```

Şekil 4-3. Lempel-Ziv genişletme algoritması.

Şekil 4-4, Şekil 4-2'deki sıkıştırılmış şifreyi genişleten algoritmanın adımlarını göstermektedir. Bu şekildeki önemli nokta, genişletme katar tablosu ile sıkıştırma katar tablosunun sonnuçta aynı olmasıdır. Şekillerde görüldüğü gibi sıkıştırma algoritmasının girdisi ile genişletme algoritmasının çıktısı aynıdır ki bu da, bir sıkıştırma algoritmasından beklediğimiz normal sonuçtur.

Girdi YENİŞİFRE	ŞİFRE	KATAR çıktısı	KARAKTER	Katar Tablosu
/	/	/		
K	K	K	K	256 = /K
A	A	A	A	257 = KA
Ç	Ç	Ç	Ç	258 = AÇ
256	256	/K	/	259 = Ç/
A	A	A	A	260 = /KA
L	L	L	L	261 = AL
260	260	/KA	/	262 = L/
N	N	N	N	263 = /KAN
260	260	/KA	/	264 = N/
R	R	R	R	265 = /KAR
265	265	/KAR	/	266 = R/
A	A	A	A	267 = /KARA

Şekil 4-4. Örnek Lempel-Ziv genişletme uygulaması.

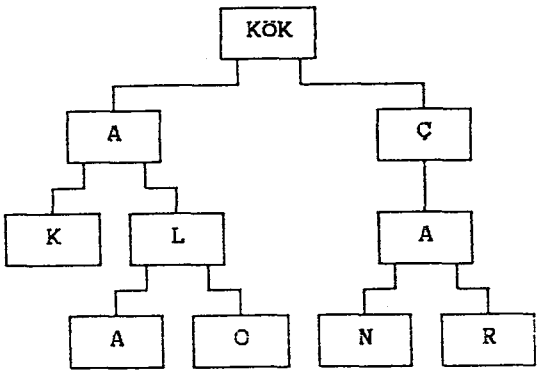
4.4 Karakter Katarı Tablosu

Bu algoritmalardaki mantık önceki bölümlerde anlatıldığı gibi oldukça basittir. Her iki algoritma da sadece bir kaç satırla ifade edilmiştir. Fakat algoritmaların bir bilgisayar diline uyarlanması gerektiğinde, karakter katarı tablosunun yönetimi dolayısıyla bu işlem oldukça zor olmaktadır. Ayrıca bu tablonuun üzerinde arama işlemi yapıldığından, arama işlemini hızlı kılacak bir veri yapısının da seçilmesi gereği görülmektedir.

Veri yapısı olarak dizi seçilebilir ve yeni elemanlar diziye sıra ile yerleştirilebilir. Fakat o zaman sıkıştırma algoritmasında arama zamanı oldukça büyük olur. Ek G ve H , veri yapısı dizi olan programlar içermektedir. Bu programlarda veri diziye sırasal olarak yerleştirilmektedir.

Welch [Welch 1984] veri yapısı olarak yine diziyi seçmiş, fakat elemanları diziye sırasal olarak değil, bir karıştırma fonksiyonu kullanarak yerleştirmiştir [Nelson 1989]. Bu da algoritmaya hız kazandırmıştır.

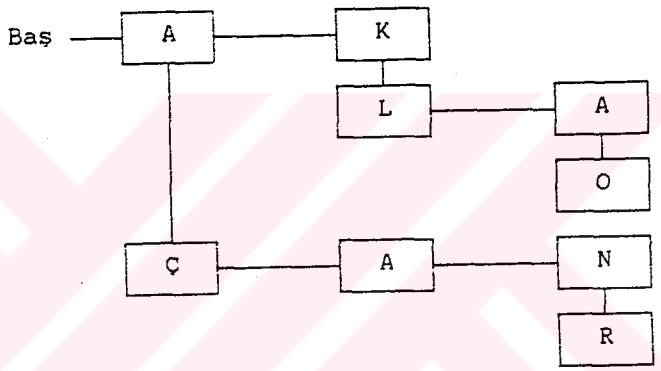
Bu çalışmada karakter katarı tablosu, çoklu ağaç yapısı olarak düşünülmüştür. Bu yapıda her düğüm bir karakter bilgisi saklar. Bu düğümden çıkan çok sayıdaki göstergeçler bir sonraki karakteri içeren düğümü işaret eder. Yalnız kök düğümde karakter bilgisi saklanmaz. Bu ağaçta kökten çıkan her uzunluktaki yollar karakter katarlarını oluşturur. Şekil 4-5'te örnek bir çoklu ağaç gösterilmektedir. Şekilde kökten çıkan yollardaki karakter dizileri {A,AK,AL,ALA,ALO,Ç,ÇA,ÇAN,ÇAR} birer karakter katarını oluştururlar.



Şekil 4-5. Çoklu ağaç yapısında karakter katarı tablosu.

Lempel-Ziv algoritmasında karakter katarı tablosuna eklenen katar daha önceden varolan bir katarın sonuna sadece bir karakter eklenerek elde edildiğinden, bu tür bir yapıda yeni bir katar ekleme, ağaca sadece yeni bir düğüm eklemekten ibarettir.

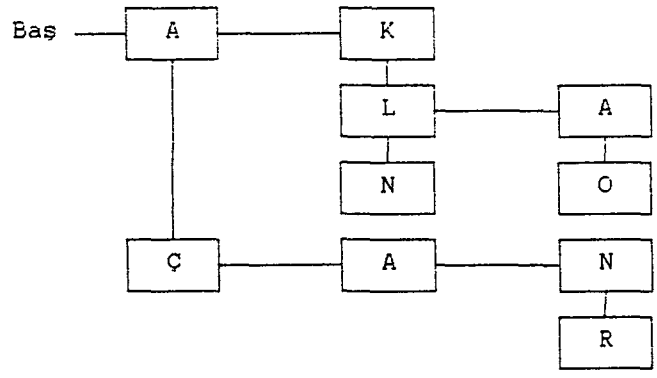
Bu yapı her düğümden çıkan çok sayıda göstergeçlerin ağacın ilk seviyedeki düğümlerinden itibaren kullanılmamaya başlanmasından dolayı bellek israfına yol açtığından ikili ağaç yapısı ile temsil edilebilir. İkili ağaç yapısında her düğümden iki göstergeç çıkar. Şekil 4-5'teki örnek çoklu ağaç ikili ağaç ile Şekil 4-6'da gösterilmiştir.



Şekil 4-6. İkili ağaç yapısında karakter katarı tablosu.

Bu şekilde yatay konumdaki göstergeç bir sonraki karakteri, düşey konumdaki göstergeç bir sonraki katarı göstermektedir. Burada A'dan çıkan yatay göstergeç sonraki karakterin K olduğunu göstermektedir. Oluşan katar AK'dır. K'dan çıkan düşey göstergeç kendinden önce gelen önek A'yı sabit tutarak katarın devamını L olarak göstermektedir. Yani, AL katarını oluşturmaktadır. Burada A ve L arasında hayali bir göstergeç vardır ve bu K düğümü vasıtasıyla bağlanmaktadır. Aynı yöntemle bütün karakter katarları {A,AK,AL,ALA,ALO,Ç,ÇA,ÇAN,ÇAR} olarak bulunur. Şekil 4-6'daki karakter katarı tablosuna AN karakter katarını eklemek yalnızca L düğümüne yatay göstergeçle N düğümünü bağlamaktır. Sonuç Şekil 4-7'de gösterilmektedir.

Ek I ve J, veri yapısı ağaç olarak seçen Lempel-Ziv şifrelemesi ve deşifrelemesi yapan programları içermektedir. Bu programlar ile G ve



Şekil 4-7. Karakter katarı tablosuna "AN" katarının eklenmesi.

H eklerindeki programlardaki Lempel-Ziv algoritmaları, tablonun idaresi haricinde aynıdır. Algoritmalar arasındaki fark sadece "intable" fonksiyonu ve "insert" yordamındadır. "intable" fonksiyonu bir karakter katarının karakter katarı tablosunda olup olmadığını kontrol eder. "insert" yordamı ise, karakter katarını tabloya veya ağaca ekler.

SONUÇ VE ÖNERİLER

Aşağıdaki tabloda çeşitli uzunluktaki dosyaların, sırasıyla G, H, I ve J eklerindeki programlar ile işlenme sürelerini içermektedir. Tabloda en çarpıcı nokta, dosya uzunluğu arttıkça tablo kullanan sıkıştırıcı programının süresinin üssel olarak artmasıdır. Diğer bir nokta ise tablo kullanan genişletici programının ağaç yapısı kullanan genişletici programından daha kısa çalışma süresi olmasıdır. Yani, sıkıştırıcı olarak ağaç yapısı kullanan program, genişletici olarak tablo kullanan program kullanıldığında en iyi performans elde edilmektedir.

Dosya Uzunluğu	Sıkıştırma Ağaç Yapısı	Sıkıştırma Tablo	Genişletme Ağaç Yapısı	Genişletme Tablo
199	0.65	0.60	0.54	0.54
3045	0.93	5.10	0.82	0.76
5463	1.20	9.44	0.98	0.93
10187	1.59	24.00	1.64	1.31
15822	2.41	69.86	1.86	1.64
21970	3.18	166.80	3.07	2.14
30110	4.88	430.83	7.90	3.02
56492	10.98	1568.94	14.56	6.31

KAYNAKLAR

ABRAMSON, N.,(1963), Information Theory and Coding, McGraw-Hill, New York.

APOSTOLICO, A., ve FRAENKEL, A.S.,(1985), Robust transmission of unbounded strings using Fibonacci representations. Tech. Rep. CS85-14, Dept. of Applied Mathematics, The Wiezmann Institute of Science, Rehovot, Israel.

BENTLEY, J.L., SLEATOR, D.D., TARJAN, R.E., ve WEI, V.K.,(1986), A locally adaptive data compression scheme, Comm. ACM, Vol.29, No.4, pp. 320-330.

CAPPELLINI, V.,(1985), Data Compression and Error Control Techniques with Applications, Academic Press, London.

CONNEL, J.B.,(1973), A Huffman-Shannon-Fano code, Proc. IEEE, Vol.61, No.7, pp. 1046-1047.

CORTESI, D.,(1982), An effective text-compression algorithm, BYTE, Vol.7, No.1, pp. 397-403.

DAVIDSON, L.D.,(1966), Theory of Adaptive Data Compression Edited by A.V.Balakrishinan, Academicc Press, New York, pp. 173-192.

ELIAS, P.,(1975), Universal codeword sets and representations of integers, IEEE Trans. Inf. Theory, Vol.21, No.2, pp. 194-203.

FALLER, N.,(1973), An adaptive system for data compression, In record of the 7th Asilomar Conference on Circuits, Systems and Computers, Naval Postgraduate School, Monterey, pp. 593-597.

FANO, R.M.,(1949), Transmission of Information, M.I.T. Press, Cambridge.

FRAENKEL, A.S., MOR, M., ve PERL, Y.,(1983), Is text compression by prefixes and suffixes practical?, Acta Inf., Vol.20, No.4, pp. 371-375.

FRAENKEL, A.S., ve KLEIN, S.T.,(1985), Robust universal complete codes as alternatives to Huffman

GALLAGER, R.G.,(1968), Information Theory and Reliable Communication, Wiley, New York.

GALLAGER, R.G.,(1978), Variations on a theme by Huffman, IEEE Trans. Inf. Theory, Vol.24, No.6, pp. 668-674.

GILBERT, E.N.,(1971), Codes based on inaccurate source probabilities, IEEE Trans. Inf. Theory, Vol.17, No.3, pp. 304-314.

GONZALEZ, R.C., ve WINTZ, P.,(1977), Digital Image Processing, Addison-Wesley, Reading.

HAHN, B.,(1974), A new technique for compression and storage of data, Comm. ACM, Vol.17, No.8, pp. 434-436.

HUFFMAN, D.A.,(1952), A method for the construction of minimum-redundancy codes, Proc. IRE, Vol.40, No.9, pp. 1098-1101.

JONES, D.W.,(1988), Application of splay trees to data compression, Comm. ACM, Vol.31, No.8, pp. 996-1007.

KNUTH, D.E.,(1985), Dynamic Huffman coding, J. Algorithms, Vol.6, No.2, pp. 163-180.

LAESER, R.P., MCLAUGHLIN, W.I., ve WOLFF, D.M.,(1986), Engineering Voyager 2's encounter with Uranus, Sci. Am., Vol.255, No.5, pp. 36-45.

LELEWER, D.A., ve DANIEL, S.H.,(1987), Data compression, ACM Comp. Surveys, Vol.19, No.3, pp. 261-296.

NELSON, R.M.,(1989), LZW Data Compression, Dr.Dobb's J., Oct., pp. 29-36, 86-87.

PASCO, R.,(1976), Source coding algorithms for fast data compression, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., Stanford.

PKARC FAST!,(1987), PKARC FAST File archival utility, Version 3.5, PKWARE Inc., Glendale.

REGAN, S.M.,(1990), LZW revisited, Dr. Dobb's J., Jun, pp. 126-127,167.

REGHBATI, H.K.,(1981), An overview of data compression techniques, Computer, Vol.14, No.4, pp. 71-75.

RISSANEN, J.J.,(1976), Generalized Kraft inequality and arithmetic coding, IBM J. Res. Dev., Vol.20, pp. 198-203.

RISSANEN, J.J.,(1983), A universal data compression system, IEEE Trans. Inf. Theory, Vol.29, No.5, pp. 656-664.

RODEH, M., PRATT, V.R., ve EVEN, S.,(1981), Linear algorithm for data compression via string matching, J. ACM, Vol.28, No.1, pp. 16-24.

RUBIN, F.,(1976), Experiments in text file compression, Comm. ACM, Vol.19, No.11, pp. 617-623.

RUBIN, F.,(1979), Arithmetic stream coding using fixed precision registers, IEEE Trans. Inf. Theory, Vol.25, No.6, pp. 672-675.

RUTH, S.S, ve KREUTZER, P.J.,(1972), Data compression for large business files, Datamation, Vol.18, No.9, pp. 62-66.

SCHWARTZ, E.S.,(1964a), An optimum encoding with minimum longest code and total number of digits, Inf. Control, Vol.7, No.1, pp. 37-44.

SCHWARTZ, E.S., ve KALLICK, B.,(1964b), Generating a canonical prefix encoding, Comm. ACM, Vol.7, No.3, pp. 166-169.

SHANNON, C.E., ve WEAVER, W.,(1949), The Mathematical Theory of Communication, University of Illinois Press, Urbana.

SNYDERMAN, M., ve HUNT, B.,(1970), The myriad virtues of text compaction, Datamation, Vol.16, No.12, pp. 36-40.

STORER, J.A., ve SZYMANSKI, T.G.,(1982), Data compression via textual substitution, J. ACM, Vol.29, No.4, pp. 928-951.

THOMAS, K.,(1991), Entropy, Dr.Dobb's J., Feb., pp. 32-34,110.

TROPPER, R.,(1982), Binary-coded text, A text-compression method, BYTE, Vol.7, No.4, pp. 398-413.

UNIX,(1984), UNIX User's Manual, Version 4.2 Berkeley Software Distribution, University of California, Berkeley.

VITTER, J.S.,(1987), Design and analysis of dynamic Huffman codes, J. ACM, Vol.34, No.4, pp. 825-845.

WAGNER, R.A.,(1973), Common phrases and minimum-space text storage, Comm. ACM, Vol.16, No.3, pp. 148-152.

WELCH, T.A.,(1984), A technique for high-performance data compression, Computer, Vol.17, No.6, pp. 8-19.

WILKINS, L.C., ve WINTZ, P.A.,(1971), Bibliograpy on data compression, picture properties and picture encoding, IEEE Trans. Inf. Theory, Vol.17, No.2, pp. 180-197.

WITTEN, I.H., NEAL, R.M., ve CLEARY, J.G,(1987), Arithmetic coding for data compression, Comm. ACM, Vol.30, No.6, pp. 520-540.

ZIV, J., ve LEMPEL, A.,(1977), A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory, Vol.23, No.3, pp. 337-343.

EKLER

EK A

```
program FileEntropy;
{$M 50000,0,655360}
    uses
        crt,dos;
    var
        f : file;
        fname :string;

    function entropy (var f : file) : real;
        const
            readbuffersize = 20000;
        var
            all : longint;
            sum : real;
            i : integer;
            freqtable : array [0..255] of longint;
            readbuffer : array [1..readbuffersize] of byte;
            readsize : word;

        begin
            for i:=0 to 255 do
                freqtable[i]:=0;
            repeat
                 blockread (f,readbuffer,readbuffersize,readsize);
                for i:=1 to readsize do
                    inc (freqtable[readbuffer[i]]);
            until (readsize < readbuffersize);
            sum:=0;
            all:=filesize (f);
            for i:=0 to 255 do
                begin
                    if (freqtable[i] <> 0) then

sum:=sum+freqtable[i]/all*ln(freqtable[i]/all)/ln(2);
                end;
            writeln ('possible size is : ',-sum*filesize
(f)/8:18:2);
            entropy:=-sum;
        end;

    function fileexist (fname : string) : boolean;
        var
            f : file;
        begin
            {$I-}
            assign (f,fname);
```

```
        reset (f);
        close (f);
        {$I+}
        fileexist:= (ioresult = 0);
    end;

begin
    fname:=paramstr(1);
    if (fname = '') then
        writeln ('Invalid syntax : "ENTROPY infile" required')
    else
        if fileexist (fname) then
            begin
                assign (f,fname);
                reset (f,1);
                writeln (entropy (f):10:8);
            end
        else
            writeln ('Error : infile does not exist');
end.
```

EK B

```
program FrequencyCount;
{$M 50000,0,655360}
    uses
        crt,dos;
    type
        letters = 'a'..'z';
    var
        f : file;
        g : text;
        fnamein : string;
        freqtable : array [0..255] of longint;
        lettertable : array [letters] of longint;

    procedure getfrequency (var f : file);
        const
            readbuffersize = 20000;
        var
            i : integer;
            readbuffer : array [1..readbuffersize] of byte;
            readsize : word;

        begin
            for i:=0 to 255 do
                freqtable[i]:=0;
            repeat
                 blockread (f,readbuffer,readbuffersize,readsize);
                 for i:=1 to readsize do
                     inc (freqtable[readbuffer[i]]);
             until (readsize < readbuffersize);
        end;

    procedure processtable;
        var
            ch : char;
            c, g, i, o, s, u : longint;
            sum : longint;
        begin
            for ch:='a' to 'h' do

lettertable[ch]:=freqtable[ord(ch)]+freqtable[ord(upcase(ch))];
             lettertable['i']:=freqtable[ord('i')]+freqtable[152];
            for ch:='j' to 'z' do

lettertable[ch]:=freqtable[ord(ch)]+freqtable[ord(upcase(ch))];
            lettertable['q']:=0;
            lettertable['w']:=0;
```

```
        lettertable['x']:=0;
        c:=freqtable[ord('Ç')]+freqtable[ord('ç')];
        g:=freqtable[ord('Ğ')]+freqtable[ord('ğ')];
        i:=freqtable[ord('I')]+freqtable[ord('ı')];
        o:=freqtable[ord('Ö')]+freqtable[ord('ö')];
        s:=freqtable[ord('Ş')]+freqtable[ord('ş')];
        u:=freqtable[ord('Ü')]+freqtable[ord('ü')];
        sum:=0;
        for ch:='a' to 'z' do
            sum:=lettertable[ch]+sum;
        sum:=sum+c+g+i+o+s+u;
        writeln (sum);
        for ch:='a' to 'z' do
            write (ch:2,':',lettertable[ch]/sum:7:5);
        write ('ç':2,':',c/sum:7:5);
        write ('ğ':2,':',g/sum:7:5);
        write ('ı':2,':',i/sum:7:5);
        write ('ö':2,':',o/sum:7:5);
        write ('ş':2,':',s/sum:7:5);
        write ('ü':2,':',u/sum:7:5);
    end;

function fileexist (fname : string) : boolean;
    var
        f : file;
    begin
        {$I-}
        assign (f,fname);
        reset (f);
        close (f);
        {$I+}
        fileexist:= (ioresult = 0);
    end;

begin
    fnamein:=paramstr (1);
    if (fnamein = '') then
        writeln ('Invalid syntax : "FREQ infile" required')
    else
        if fileexist (fnamein) then
            begin
                assign (f,fnamein);
                reset (f,1);
                getfrequency (f);
                processtable;
                close (f);
            end
        else
            writeln ('Error : infile does not exist');
end.
```

EK C

```
program RunLengthEncodingCompress;
    const
        numbersign = 0;
    type
        filetype = file of byte;
    var
        f,
        g : filetype;
        fnamein,
        fnameout : string;

    procedure writebyte (var g : filetype; abyte : byte);
        begin
            write (g,abyte);
        end;

    procedure compress;
        var
            repetition : integer;
            repwrite,
            prevbyte,
            currbyte : byte;

        procedure writesymbol;
            var
                i : integer;
            begin
                if (repetition > 3) then
                    begin
                        while (repetition > 0) do
                            begin
                                if (repetition > 255) then
                                    repwrite:=255
                                else
                                    repwrite:=repetition;
                                writebyte (g,numbersign);
                                writebyte (g,repwrite);
                                writebyte (g,prevbyte);
                                repetition:=repetition-255;
                            end;
                    end
                else
                    begin
                        if (prevbyte = numbersign) then
                            repetition:=repetition*2;
                            for i:=1 to repetition do
```

```
                                writebyte (g,prevbyte);
                        end;
                end;


        begin
                assign (f,fnamein);
                reset (f);
                assign (g,fnameout);
                rewrite (g);
                repetition:=1;
                read (f,prevbyte);
                while not eof (f) do
                    begin
                        read (f,currbyte);
                        if (currbyte = prevbyte) then
                            repetition:=repetition+1
                        else
                            begin
                                writesymbol;
                                prevbyte:=currbyte;
                                repetition:=1;
                            end;
                    end;
                writesymbol;
                close (f);
                close (g);
        end;

    function fileexist (fname : string) : boolean;
        var
            f : file;
        begin
            {$I-}
            assign (f,fname);
            reset (f);
            close (f);
            {$I+}
            fileexist:= (ioresult = 0);
        end;

    begin
        fnamein:=paramstr(1);
        fnameout:=paramstr(2);
        if (fnamein = '') or (fnameout = '') then
            writeln ('Invalid syntax : "RLE- infile outfile"
required')
        else
            if fileexist (fnamein) then
                compress
            else
                writeln ('Error : infile does not exist');
    end.
```

EK D

```
program RunLengthEncodingDecompress;
    const
        numbersign = 0;
    type
        filetype = file of byte;
    var
        f,
        g : filetype;
        fnamein,
        fnameout : string;

    procedure writebyte (var g : filetype; abyte : byte);
        begin
            write (g,abyte);
        end;

    procedure decompress;
        var
            repwrite,
            currbyte : byte;
            i : integer;
        begin
            assign (f,fnamein);
            reset (f);
            assign (g,fnameout);
            rewrite (g);
            while not eof (f) do
                begin
                    read (f,currbyte);
                    if (currbyte = numbersign) then
                        begin
                            read (f,currbyte);
                            if (currbyte = numbersign) then
                                writebyte (g,currbyte)
                            else
                                begin
                                    repwrite:=currbyte;
                                    read (f,currbyte);
                                    for i:=1 to repwrite do
                                        writebyte (g,currbyte)
                                end;
                        end
                    else
                        writebyte (g,currbyte);
                end;
            close (f);
```

```
            close (g);
        end;

    function fileexist (fname : string) : boolean;
        var
            f : file;
        begin
            {$I-}
            assign (f,fname);
            reset (f);
            close (f);
            {$I+}
            fileexist:= (ioresult = 0);
        end;

    begin
        fnamein:=paramstr(1);
        fnameout:=paramstr(2);
        if (fnamein = '') or (fnameout = '') then
            writeln ('Invalid syntax : "RLEX infile outfile"
required')
        else
            if fileexist (fnamein) then
                decompress
            else
                writeln ('Error : infile does not exist');
    end.
```

EK E

```
program HuffmanCompression (input,output);

{$M 35000,0,655360}
    uses
        dos,
        crt;
    const
        maxbits = 50;
        maxbitpos = 51;
        maxsymbols = 256;
        maxnodes = 512; {maxnodes equals 2*maxsymbols-1}
    type
        stype = (lson,rson);
        nodeptr = -1..maxnodes;
        bit = '0'..'1';
        codetype =
            record
                bits : array [1..maxbits] of bit;
                startpos : 1..maxbitpos;
            end;
        nodetype =
            record
                freq : word;
                byteinfo : byte;
                subright,
                subleft,
                father : nodeptr; {father is -1 if a node is in
rootnodes}
                sontype : stype;
            end;
        bytefile = file of byte;

    var
        frequencytable : array [0..maxsymbols-1] of word;
        f : file;
        g : file;
        abyte : byte;
        code : array [0..maxsymbols-1] of codetype;
        alphabet : array [0..maxsymbols-1] of byte;
        nodes : array [0..maxnodes-1] of nodetype;
        k : 1..maxbits;
        n,
        i : -1..maxsymbols;
        root,
        p,
        n1,
```

```
    n2,
    j,
    m : nodeptr;
    cd : codetype;
    symbol : char;
    small1,
    small2 : integer;
    a : string[30];
    h,m1,s,s1 : word;
    ch : char;
    maxfreq : longint;
    numberofsymbols : byte;
    fnamein,
    fnameout : string;

procedure initialize;
    begin
        for i:=0 to maxsymbols-1 do
            frequencytable[i]:=0;
    end;

procedure freqcount;
    const
        readbuffersize = 32000;
    var
        readbuffer : array [1..readbuffersize] of byte;
        readsize : word;
        i : integer;
    begin
        assign (f,fnamein);
        reset (f,1);
        repeat
            blockread (f,readbuffer,readbuffersize,readsize);
            for i:=1 to readsize do
                inc (frequencytable[readbuffer[i]]);
        until (readsize < readbuffersize);
    end;

procedure createtree;
    begin
        for j:=0 to maxnodes-1 do
            begin
                nodes[j].freq:=0;
                nodes[j].father:=-1;
                nodes[j].subright:=-1;
                nodes[j].subleft:=-1;
            end;
        for i:=0 to maxsymbols-1 do
            alphabet[i]:=0;
    n:=-1;
    maxfreq:=0;
    for i:=0 to maxsymbols-1 do
        if (frequencytable[i] <> 0) then
            begin
                if (frequencytable[i] > maxfreq) then
                    maxfreq:=frequencytable[i];
                n:=n+1;
                nodes[n].byteinfo:=i;
                nodes[n].freq:=frequencytable[i];
                alphabet[n]:=i;
            end;
    numberofsymbols:=n;      {0..n}
```

```
{we now build the trees}
for m:=n+1 to 2*n do
    begin
        {m is the next available node. Search all previous}
        {  nodes for the two root nodes n1 and n2 with    }
        {            smallest frequencies                 }
        n1:=0;
        n2:=0;
        small1:=maxint;
        small2:=maxint;
        for j:=0 to m-1 do
            if (nodes[j].father = -1) then {j is a root
node}
                if (nodes[j].freq < small1) then
                    begin
                        small2:=small1;
                        small1:=nodes[j].freq;
                        n2:=n1;
                        n1:=j;
                    end
                else
                    if (nodes[j].freq < small2) then
                        begin
                            small2:=nodes[j].freq;
                            n2:=j;
                        end;
        {set n1 to the left subtree of m and}
        {      n2 to the right subtree      }
        nodes[n1].father:=m;
        nodes[n1].sontype:=lson;
        nodes[n2].father:=m;
        nodes[n2].sontype:=rson;
        nodes[m].freq:=nodes[n1].freq+nodes[n2].freq;
        nodes[m].subleft:=n1;
        nodes[m].subright:=n2;
        root:=m;
    end;


    {extract the codes from the tree}
    for i:=0 to n do
        begin
        {initialize code[i]}
            cd.startpos:=maxbitpos;
            {travel up the tree}
            j:=i;
            while (nodes[j].father <> -1) do
                begin
                    if (nodes[j].sontype = lson) then
                        begin
                            cd.startpos:=cd.startpos-1;
                            cd.bits[cd.startpos]:='0';
                        end
                    else
                        begin
                            cd.startpos:=cd.startpos-1;
                            cd.bits[cd.startpos]:='1';
                        end;
                    j:=nodes[j].father;
                end;
            code[nodes[i].byteinfo]:=cd;
```

```
                end;
        end;

    procedure compress;
        const
            readbuffersize = 16000;
            writebuffersize = 16000;
        var
            readbuffer : array [1..readbuffersize] of byte;
            readsize : word;
            writebuffer : array [1..writebuffersize] of byte;
            writesize : word;

        procedure writebyte (var g : file; abyte : byte);
            begin
                inc (writesize);
                writebuffer[writesize]:=abyte;
                if (writesize = writebuffersize) then
                    begin
                        blockwrite(g,writebuffer,writesize);
                        writesize:=0;
                    end;
            end;

        procedure writeword (var g : file; aword : word);
            var
                abyte : byte;
            begin
                abyte:=aword div 256;
                writebyte (g,abyte);
                abyte:=aword - abyte * 256;
                writebyte (g,abyte);
            end;

        procedure writeheader;
            begin
                for i:=0 to 255 do
                    writeword (g,frequencytable[i]);
            end;

        procedure writefile;
            var
                i : integer;
                k : byte;

            procedure putfile;
                var
                    j : byte;
                begin
                    for j:=code[readbuffer[i]].startpos to maxbits
do
                        begin
                            abyte:=abyte shl 1;
                            if (code[readbuffer[i]].bits[j] = '1')
then
                                abyte:=abyte or 1;
                            inc(k);
                            if (k = 9) then
                                begin
                                    writebyte(g,abyte);
                                    k:=1;
                                    abyte:=0;
```

```
                                end;
                            end;
                        end;

                    begin
                        assign (f,fnamein);
                        reset (f,1);
                        abyte:=0;
                        k:=1;
                        repeat
                            blockread
(f,readbuffer,readbuffersize,readsize);
                            for i:=1 to readsize do
                                putfile;
                        until (readsize < readbuffersize);
                        if (k <> 1) then
                            begin
                                abyte:=abyte shl (9-k);
                                writebyte(g,abyte);
                            end;
                        close (f);
                    end;

                begin
                    writesize:=0;
                    assign (g,fnameout);
                    rewrite (g,1);
                    writeheader;
                    writefile;
                    blockwrite(g,writebuffer,writesize);
                    close (g);
                end;

            procedure encode;
                begin
                    freqcount;
                    createtree;
                    writeln ('numberofsymbols :',numberofsymbols+1);
                    writeln ('max freq : ',maxfreq);
                    compress;
                end;

                procedure printresults;
                    var
                        fsize : longint;
                    begin
                        fsize:=0;

                        {print results}
                        for i:=0 to n do
                            begin
                                fsize:=fsize+nodes[i].freq*(maxbits+1-
code[nodes[i].byteinfo].startpos);
                            write                                (alphabet[i],'
',frequencytable[alphabet[i]],' ');
                            for  k:=code[nodes[i].byteinfo].startpos  to
maxbits do
                                write (code[nodes[i].byteinfo].bits[k]);
                            ch := readkey;
                            writeln;
                        end;
                    writeln(fsize div 8);
```

```
            end;

    function fileexist (fname : string) : boolean;
        var
            f : file;
        begin
            {$I-}
            assign (f,fname);
            reset (f);
            close (f);
            {$I+}
            fileexist:= (ioresult = 0);
        end;

    begin
         fnamein:=paramstr(1);
         fnameout:=paramstr(2);
         if (fnameout = '') or (fnamein = '') then
            writeln ('Invalid syntax : "HUFF- infile outfile"
required')
         else
             if fileexist (fnamein) then
                 begin
                     initialize;
                     encode;
{                    printresults;}
                 end
             else
                 writeln ('Error : infile does not exist');
    end.
```

EK F

```
program HuffmanDecompression (input,output);
{$M 35000,0,655360}
    uses
        dos,
        crt;
    const
        maxbits = 50;
        maxbitpos = 51;
        maxsymbols = 256;
        maxnodes = 512; {maxnodes equals 2*maxsymbols-1}
    type
        stype = (lson,rson);
        nodeptr = -1..maxnodes;
        bit = '0'..'1';
        codetype =
            record
                bits : array [1..maxbits] of bit;
                startpos : 1..maxbitpos;
            end;
        nodetype =
            record
                freq : word;
                byteinfo : byte;
                subright,
                subleft,
                father : nodeptr; {father is -1 if a node is in
rootnodes}
                sontype : stype;
            end;
        bytefile = file of byte;

    var
        frequencytable : array [0..maxsymbols-1] of word;
        f : file;
        g : file;
        abyte : byte;
        code : array [0..maxsymbols-1] of codetype;
        alphabet : array [0..maxsymbols-1] of byte;
        nodes : array [0..maxnodes-1] of nodetype;
        k : 1..maxbits;
        n,
        i : -1..maxsymbols;
        root,
        p,
        n1,
        n2,
```

```
        j,
        m : nodeptr;
        cd : codetype;
        symbol : char;
        small1,
        small2 : integer;
        a : string[30];
        h,m1,s,s1 : word;
        ch : char;
        maxfreq : longint;
        numberofsymbols : byte;
        fname : string[30];
        first : boolean;

    procedure createtree;
        begin
            for j:=0 to maxnodes-1 do
                begin
                    nodes[j].freq:=0;
                    nodes[j].father:=-1;
                    nodes[j].subright:=-1;
                    nodes[j].subleft:=-1;
                end;
            for i:=0 to maxsymbols-1 do
                alphabet[i]:=0;
        n:=-1;
        maxfreq:=0;
        for i:=0 to maxsymbols-1 do
            if (frequencytable[i] <> 0) then
                begin
                    if (frequencytable[i] > maxfreq) then
                        maxfreq:=frequencytable[i];
                    n:=n+1;
                    nodes[n].byteinfo:=i;
                    nodes[n].freq:=frequencytable[i];
                    alphabet[n]:=i;
                end;
        numberofsymbols:=n;      {0..n}

        {we now build the trees}
        for m:=n+1 to 2*n do
            begin
                {m is the next available node. Search all previous}
                {  nodes for the two root nodes n1 and n2 with    }
                {            smallest frequencies                 }
                n1:=0;
                n2:=0;
                small1:=maxint;
                small2:=maxint;
                for j:=0 to m-1 do
                    if  (nodes[j].father  =  -1)  then  {j  is  a  root
node}
                        if (nodes[j].freq < small1) then
                            begin
                                small2:=small1;
                                small1:=nodes[j].freq;
                                n2:=n1;
                                n1:=j;
                            end
                        else
                            if (nodes[j].freq < small2) then
                                begin
```

```
                                    small2:=nodes[j].freq;
                                    n2:=j;
                                end;
                    {set n1 to the left subtree of m and}
                    {       n2 to the right subtree       }
                    nodes[n1].father:=m;
                    nodes[n1].sontype:=lson;
                    nodes[n2].father:=m;
                    nodes[n2].sontype:=rson;
                    nodes[m].freq:=nodes[n1].freq+nodes[n2].freq;
                    nodes[m].subleft:=n1;
                    nodes[m].subright:=n2;
                    root:=m;
                end;


                {extract the codes from the tree}
                for i:=0 to n do
                    begin
                    {initialize code[i]}
                        cd.startpos:=maxbitpos;
                        {travel up the tree}
                        j:=i;
                        while (nodes[j].father <> -1) do
                            begin
                                if (nodes[j].sontype = lson) then
                                    begin
                                        cd.startpos:=cd.startpos-1;
                                        cd.bits[cd.startpos]:='0';
                                    end
                                else
                                    begin
                                        cd.startpos:=cd.startpos-1;
                                        cd.bits[cd.startpos]:='1';
                                    end;
                                j:=nodes[j].father;
                            end;
                        code[nodes[i].byteinfo]:=cd;
                    end;
            end;

        procedure decompress;
            const
                readbuffersize = 16000;
                writebuffersize = 16000;
            var
                readbuffer : array [1..readbuffersize] of byte;
                readsize : word;
                writebuffer : array [1..writebuffersize] of byte;
                writesize : word;

            procedure writebyte (var g : file; abyte : byte);
                begin
                    inc (writesize);
                    writebuffer[writesize]:=abyte;
                    if (writesize = writebuffersize) then
                        begin
                            blockwrite(g,writebuffer,writesize);
                            writesize:=0;
                        end;
                end;
```

```
procedure writefile;
    var
        i : integer;
        k : byte;

    procedure putfile;
        begin
            writeln ('Enter a code ');
            for i:=1 to 30 do
                a[i]:=' ';
            readln (a);
            i:=0;
            while (a[i] <> ' ') do
                begin
                    i:=i+1;
                    if (nodes[p].subleft = -1) and
                       (nodes[p].subright = -1) then
                       begin
                            write (chr (alphabet[p]));
                           p:=root;
                       end;
                    if (a[i] = '0') then
                        p:=nodes[p].subleft
                    else
                        p:=nodes[p].subright;
                end;
        end;

    begin
        abyte:=0;
        k:=1;
        first:=true;
        p:=root;
        repeat
            blockread
(f,readbuffer,readbuffersize,readsize);
            if first then
                begin
                    for i:=1 to 256 do
                            frequencytable[i-
1]:=256*readbuffer[(i-1)*2+1]
                                                +readbuffer[(i-
1)*2+2];
                    createtree;
                    first:=false;
                    i:=513;
                    while (i <= readsize) do
                        putfile;
                end
            else
                begin
                    i:=1;
                    while (i <= readsize) do
                        putfile;
                end;
         until (readsize < readbuffersize);
    end;

begin
    writesize:=0;
    assign (f,fname);
    reset (f,1);
```

```
            assign (g,'huf.in');
            rewrite (g,1);
            writefile;
             blockwrite(g,writebuffer,writesize);
            close (g);
            close (f);
        end;

    procedure decode;
        begin
            decompress;
{            writeln ('Enter a code ');
            for i:=1 to 30 do
                a[i]:=' ';
            readln (a);
            i:=0;
            p:=root;
            while (a[i] <> ' ') do
                begin
                    i:=i+1;
                    if (nodes[p].subleft = -1) and
                       (nodes[p].subright = -1) then
                       begin
                            write (chr (alphabet[p]));
                           p:=root;
                       end;
                    if (a[i] = '0') then
                        p:=nodes[p].subleft
                    else
                        p:=nodes[p].subright;
                end; }
        end;

        procedure printresults;
            var
                fsize : longint;
            begin
                fsize:=0;

                {print results}
                for i:=0 to n do
                    begin
                         fsize:=fsize+nodes[i].freq*(maxbits+1-
code[nodes[i].byteinfo].startpos);
                        write                              (alphabet[i],'
',frequencytable[alphabet[i]],' ');
                        for  k:=code[nodes[i].byteinfo].startpos  to
maxbits do
                             write (code[nodes[i].byteinfo].bits[k]);
                        ch := readkey;
                        writeln;
                    end;
                 writeln(fsize div 8);
            end;

    begin
         fname:=paramstr(1);
        decode;
        printresults;
    end.
```

EK G

```
program LZWTableCompress (input,output);
{$M 50000,0,655360}
    uses
        dos, crt;

    const
        maxbyte = 255;
        maxsymbols = 4096;

    type
        tabletype =
            record
                headinfo : integer;
                byteinfo : byte;
            end;

    var
        table : array [0..maxsymbols] of tabletype;
        nextavailcode : word;
        fnamein,
        fnameout : string;

    procedure initialize;
        var
            i : byte;
        begin
            for i:=0 to 255 do
                begin
                    table[i].headinfo:=-1;
                    table[i].byteinfo:=i;
                end;
        end;

    function intable (var index:word; strng:word; abyte:byte) :
boolean;
        var
            found : boolean;
        begin
            found:=false;
            while (index < nextavailcode) and not (found) do
                begin
                    if (table[index].byteinfo = abyte)
                       and (table[index].headinfo = strng) then
                        found:=true;
                    inc (index);
                end;
```

```
        dec (index);
        if (found) then
            intable:=true
        else
            intable:=false;
    end;

procedure insert (strng:word; abyte:byte);
    begin
        table[nextavailcode].headinfo:=strng;
        table[nextavailcode].byteinfo:=abyte;
        inc (nextavailcode);
    end;

procedure lzwcompress;
    const
        readbuffersize = 20000;
        writebuffersize = 20000;
    var
        f : file;
        abyte : byte;
        g : file;
        readbuffer : array [1..readbuffersize] of byte;
        readsize : word;
        writebuffer : array [1..writebuffersize] of byte;
        writesize : word;
        strng,
        index,
        i : word;
        ch : char;
        first,
        toggle : boolean;
        firstbyte,
        secondbyte,
        remainbyte : byte;

    procedure writebyte (var g : file; abyte : byte);
        begin
            inc (writesize);
            writebuffer[writesize]:=abyte;
            if (writesize = writebuffersize) then
                begin
                    blockwrite (g,writebuffer,writesize);
                    writesize:=0;
                end;
        end;

    procedure writecode (var g : file; aword : word);
        begin
            firstbyte:=aword shr 8;
            secondbyte:=aword;
            if toggle then
                begin
                    writebyte (g,secondbyte);
                    remainbyte:=firstbyte shl 4;
                end
            else
                begin
                    remainbyte:=remainbyte or firstbyte;
                    writebyte (g,remainbyte);
                    writebyte (g,secondbyte);
                end;
```

```
            toggle:=not toggle;
        end;

    procedure putfile;
        begin
            if (first) then
                begin
                    strng:=readbuffer[i];
                    first:=false;
                end
            else   begin
             if intable (index,strng,readbuffer[i]) then
                begin
                    strng:=index;
                end
            else
                begin
                    writecode (g,strng);
                    if nextavailcode < maxsymbols then
                        insert (strng,readbuffer[i]);
                    strng:=readbuffer[i];
                    index:=256;
                end;    end;
        end;

    begin
        index:=0;
        toggle:=true;
        first:=true;
        writesize:=0;
        assign (g,fnameout);
        rewrite (g,1);
        assign (f,fnamein);
        reset (f,1);
        abyte:=0;
        repeat
             blockread (f,readbuffer,readbuffersize,readsize);
            for i:=1 to readsize do
                putfile;
         until (readsize < readbuffersize);
        close (f);
        writecode (g,strng);
        if not (toggle) then
            writebyte (g,remainbyte);
         blockwrite (g,writebuffer,writesize);
        close (g);
    end;

function fileexist (fname : string) : boolean;
    var
        f : file;
    begin
        {$I-}
        assign (f,fname);
        reset (f);
        close (f);
        {$I+}
         fileexist:= (ioresult = 0);
    end;

begin
     fnamein:=paramstr(1);
```

```
        fnameout:=paramstr(2);
        if (fnameout = '') or (fnamein = '') then
           writeln ('Invalid syntax : "LZWT- infile outfile"
required')
        else
            if fileexist (fnamein) then
               begin
                   nextavailcode:=256;
                   initialize;
                   lzwcompress;
               end
            else
                writeln ('Error : infile does not exist');
    end.
```

EK H

```
program LZWTableDecompress (input,output);
{$M 10000,0,655360}
    uses
        dos, crt;

    const
        maxbyte = 255;
        maxsymbols = 4096;

    type
        tabletype =
            record
                headinfo : integer;
                byteinfo : byte;
            end;

    var
        table : array [0..maxsymbols] of tabletype;
        nextavailcode : word;
        fnamein,
        fnameout : string;

    procedure initialize;
        var
            i : byte;
        begin
            for i:=0 to 255 do
                begin
                    table[i].headinfo:=-1;
                    table[i].byteinfo:=i;
                end;
        end;

    procedure insert (strng:word; abyte:byte);
        begin
            table[nextavailcode].headinfo:=strng;
            table[nextavailcode].byteinfo:=abyte;
            inc (nextavailcode);
        end;

    procedure lzwdecompress;
        const
            readbuffersize = 3000;     { This should be a multiple of
three }
            writebuffersize = 3000;    { This should be a multiple of
three }
```

```
var
    f : file;
    abyte : byte;
    g : file;
    outcode : word;
     readbuffer : array [1..readbuffersize] of byte;
    readsize : word;
     writebuffer : array [1..writebuffersize] of byte;
    writesize : word;
    i,
    j : integer;
    ch : char;
    first,
    toggle : boolean;
    firstbyte,
    secondbyte,
    remainbyte : byte;
    oldcode,
    newcode : word;

 procedure writebyte (var g : file; abyte : byte);
    begin
        inc (writesize);
        writebuffer[writesize]:=abyte;
        if (writesize = writebuffersize) then
            begin
                blockwrite (g,writebuffer,writesize);
                writesize:=0;
            end;
    end;

 procedure convertfrombyte (var newcode : word; var j :
integer);
    begin
        if toggle then
            begin
                newcode:=(readbuffer[j+1] shr 4);
                newcode:=newcode shl 8;
                newcode:=newcode or readbuffer[j];
                j:=j+2;
            end
        else
            begin
                newcode:=(readbuffer[j-1] and $0F);
                newcode:=newcode shl 8;
                newcode:=newcode or readbuffer[j];
                inc (j);
            end;
        toggle:=not toggle;
    end;

 procedure putfile;

     procedure writestring (p : tabletype);
        begin
            if (p.headinfo = -1) then
                firstbyte:=p.byteinfo
            else
                writestring (table[p.headinfo]);
            writebyte (g,p.byteinfo);
        end;
```

```
        begin
            if (first) then
                begin
                    abyte:=newcode;
                    writebyte (g,abyte);
                    oldcode:=newcode;
                    first:=false;
                end
            else
                begin
                    if (newcode > nextavailcode-1) then
                        begin
                            writestring (table[oldcode]);
                            writebyte (g,firstbyte);
                        end
                    else
                        writestring (table[newcode]);
                    if nextavailcode < maxsymbols then
                        insert (oldcode,firstbyte);
                    oldcode:=newcode;
                end;
        end;

    begin
        toggle:=true;
        first:=true;
        writesize:=0;
        assign (g,fnameout);
        rewrite (g,1);
        assign (f,fnamein);
        reset (f,1);
        abyte:=0;
        repeat
            blockread (f,readbuffer,readbuffersize,readsize);
            j:=1;
            while (j <= readsize) do
                begin
                    convertfrombyte (newcode,j);
                    putfile;
                end;
        until (readsize < readbuffersize);
        close (f);
        blockwrite (g,writebuffer,writesize);
        close (g);
    end;

function fileexist (fname : string) : boolean;
    var
        f : file;
    begin
        {$I-}
        assign (f,fname);
        reset (f);
        close (f);
        {$I+}
        fileexist:= (ioresult = 0);
    end;

begin
    fnamein:=paramstr(1);
    fnameout:=paramstr(2);
    if (fnameout = '') or (fnamein = '') then
```

```
            writeln ('Invalid syntax : "LZWTX infile outfile"
required')
        else
            if fileexist (fnamein) then
                begin
                    nextavailcode:=256;
                    initialize;
                    lzwdecompress;
                end
            else
                writeln ('Error : infile does not exist');
    end.
```

EK I

```
program LZWTreeCompress (input,output);
{$M 50000,0,655360}
    uses
        dos, crt;

    const
        maxbyte = 255;
        maxsymbols = 4096;

    type
        nodeptr = ^nodetype;
        nodetype =
            record
                byteinfo : byte;
                code : word;
                nextbyte,
                nextcode : nodeptr;
            end;

    var
        start : array [0..maxbyte] of nodeptr;
        head : nodeptr;
        nextavailcode : word;
        fnamein,
        fnameout : string;

    function getnode (i : byte): nodeptr;
        var
            p : nodeptr;
        begin
            new (p);
            p^.byteinfo:=i;
            p^.code:=nextavailcode;
            inc (nextavailcode);
            p^.nextcode:=nil;
            p^.nextbyte:=nil;
            getnode:=p;
        end;

    procedure initialize;
        var
            p,
            r : nodeptr;
            i : byte;
        begin
            new (r);
```

```
            head:=r;
            for i:=0 to 255 do
                begin
                    start[i]:=r;
                    p:=r;
                    p^.code:=i;
                    p^.byteinfo:=i;
                    p^.nextbyte:=nil;
                    new (r);
                    p^.nextcode:=r;
                end;
            dispose (r);
            p^.nextcode:=nil;
        end;

    procedure displaytable (p : nodeptr; level : byte);
        var
            i : byte;
            ch : char;
        begin
            while (p <> nil) do
                begin
                    for i:=1 to level*5 do
                        write (' ');
                    writeln (chr(p^.byteinfo):4,p^.code:4);
                    ch:=readkey;
                    if (p^.nextbyte <> nil) then
                        displaytable (p^.nextbyte, level+1);
                    p:=p^.nextcode;
                end;
        end;

    function intable (var p:nodeptr; abyte:byte) : boolean;
        begin
            if (p = nil) then
                intable:=false
            else
                begin
                    while (p^.nextcode <> nil) and
(p^.nextcode^.byteinfo <= abyte) do
                        p:=p^.nextcode;
                    if (p^.byteinfo = abyte) then
                        intable:=true
                    else
                        intable:=false;
                end;
        end;

    procedure insert (var p,pp:nodeptr; abyte:byte);
        var
            r : nodeptr;
        begin
            r:=getnode (abyte);
            if (p = nil) then
                pp^.nextbyte:=r
            else
                begin
                    if (abyte < p^.byteinfo) then
                        begin
                            pp^.nextbyte:=r;
                            r^.nextcode:=p;
                        end
```

```
                else
                    begin
                        r^.nextcode:=p^.nextcode;
                        p^.nextcode:=r;
                    end;
            end;
    end;

procedure lzwcompress;
    const
        readbuffersize = 20000;
        writebuffersize = 20000;
    var
        f : file;
        abyte : byte;
        g : file;
        outcode : word;
        currnode,
        prevnode : nodeptr;
        readbuffer : array [1..readbuffersize] of byte;
        readsize : word;
        writebuffer : array [1..writebuffersize] of byte;
        writesize : word;
        i : integer;
        toggle : boolean;
        firstbyte,
        secondbyte,
        remainbyte : byte;

    procedure writebyte (var g : file; abyte : byte);
        begin
            inc (writesize);
            writebuffer[writesize]:=abyte;
            if (writesize = writebuffersize) then
                begin
                    blockwrite (g,writebuffer,writesize);
                    writesize:=0;
                end;
        end;

    procedure writecode (var g : file; aword : word);
        begin
            firstbyte:=aword shr 8;
            secondbyte:=aword;
            if toggle then
                begin
                    writebyte (g,secondbyte);
                    remainbyte:=firstbyte shl 4;
                end
            else
                begin
                    remainbyte:=remainbyte or firstbyte;
                    writebyte (g,remainbyte);
                    writebyte (g,secondbyte);
                end;
            toggle:=not toggle;
        end;

    procedure putfile;
        begin
            if intable (currnode,readbuffer[i]) then
                begin
```

```
                        outcode:=currnode^.code;
                        prevnode:=currnode;
                        currnode:=currnode^.nextbyte;
                    end
                else
                    begin
                        writecode (g,outcode);
                        if nextavailcode < maxsymbols then
                            insert
(currnode,prevnode,readbuffer[i]);
                        currnode:=start[readbuffer[i]];
                        outcode:=currnode^.code;
                        prevnode:=currnode;
                        currnode:=currnode^.nextbyte;
                    end;
            end;

        begin
            toggle:=true;
            writesize:=0;
            assign (g,fnameout);
            rewrite (g,1);
            currnode:=head;
            prevnode:=head;
            assign (f,fnamein);
            reset (f,1);
            abyte:=0;
            repeat
                blockread (f,readbuffer,readbuffersize,readsize);
                for i:=1 to readsize do
                    putfile;
            until (readsize < readbuffersize);
            close (f);
            writecode (g,outcode);
            if not (toggle) then
                writebyte (g,remainbyte);
            blockwrite (g,writebuffer,writesize);
            close (g);
        end;

    function fileexist (fname : string) : boolean;
        var
            f : file;
        begin
            {$I-}
            assign (f,fname);
            reset (f);
            close (f);
            {$I+}
            fileexist:= (ioresult = 0);
        end;

    begin
        fnamein:=paramstr(1);
        fnameout:=paramstr(2);
        if (fnameout = '') or (fnamein = '') then
            writeln ('Invalid syntax : "LZW- infile outfile"
required')
        else
            if fileexist (fnamein) then
                begin
                    nextavailcode:=256;
```

```
                initialize;
        {        displaytable (head, 0);}
                lzwcompress;
        {        displaytable (head, 0);}
            end
        else
            writeln ('Error : infile does not exist');
end.
```

EK J

```
program LZWTreeDecompress (input,output);
{$M 10000,0,655360}
    uses
        dos, crt;

    const
        maxbyte = 255;
        maxsymbols = 4096;

    type
        nodeptr = ^nodetype;
        nodetype =
            record
                byteinfo : byte;
                code : word;
                nextbyte,
                prevbyte,
                nextcode,
                prevcode : nodeptr;
            end;

    var
        start : array [0..16000] of nodeptr;
        head : nodeptr;
        nextavailcode : word;
        fnamein,
        fnameout : string;

    function getnode (i : byte): nodeptr;
        var
            p : nodeptr;
        begin
            new (p);
            p^.byteinfo:=i;
            p^.code:=nextavailcode;
            inc (nextavailcode);
            p^.nextcode:=nil;
            p^.nextbyte:=nil;
            p^.prevcode:=nil;
            p^.prevbyte:=nil;
            getnode:=p;
        end;

    procedure initialize;
        var
            p,
```

```
        r : nodeptr;
        i : byte;
    begin
        new (r);
        head:=r;
        for i:=0 to 255 do
            begin
                start[i]:=r;
                p:=r;
                p^.code:=i;
                p^.prevbyte:=nil;
                p^.prevcode:=start[i-1];
                p^.byteinfo:=i;
                p^.nextbyte:=nil;
                new (r);
                p^.nextcode:=r;
            end;
        start[0]^.prevcode:=nil;
        dispose (r);
        p^.nextcode:=nil;
    end;

procedure displaytable (p : nodeptr; level : byte);
    var
        i : byte;
        ch : char;
    begin
        while (p <> nil) do
            begin
                for i:=1 to level*5 do
                    write (' ');
                writeln (chr(p^.byteinfo):4,p^.code:4);
                ch:=readkey;
                if (p^.nextbyte <> nil) then
                    displaytable (p^.nextbyte, level+1);
                p:=p^.nextcode;
            end;
    end;

procedure insert (var p,pp:nodeptr; abyte:byte);
    var
        r : nodeptr;
    begin
        r:=getnode (abyte);
        if (p = nil) then
            begin
                pp^.nextbyte:=r;
                r^.prevbyte:=pp;
            end
        else
            begin
                if (abyte < p^.byteinfo) then
                    begin
                        r^.nextcode:=p;
                        r^.prevbyte:=pp;
                        pp^.nextbyte:=r;
                        p^.prevcode:=r;
                    end
                else
                    begin
                        r^.nextcode:=p^.nextcode;
                        p^.nextcode:=r;
```

```
                                r^.prevbyte:=pp;
                                r^.prevcode:=p;
                                if (r^.nextcode <> nil) then
                                    r^.nextcode^.prevcode:=r;
                            end;
                    end;
                start[nextavailcode-1]:=r
            end;

        procedure lzwdecompress;
            const
                readbuffersize = 3000;     { This should be a multiple of
three }
                writebuffersize = 3000;    { This should be a multiple of
three }
            var
                f : file;
                abyte : byte;
                g : file;
                outcode : word;
                currnode,
                prevnode : nodeptr;
                readbuffer : array [1..readbuffersize] of byte;
                readsize : word;
                writebuffer : array [1..writebuffersize] of byte;
                writesize : word;
                i,
                j : integer;
                ch : char;
                first,
                toggle : boolean;
                firstbyte,
                secondbyte,
                remainbyte : byte;
                oldcode,
                newcode : word;

            procedure writebyte (var g : file; abyte : byte);
                begin
                    inc (writesize);
                    writebuffer[writesize]:=abyte;
                    if (writesize = writebuffersize) then
                        begin
                            blockwrite (g,writebuffer,writesize);
                            writesize:=0;
                        end;
                end;

            procedure convertfrombyte (var newcode : word; var j :
integer);
                begin
                    if toggle then
                        begin
                            newcode:=(readbuffer[j+1] shr 4);
                            newcode:=newcode shl 8;
                            newcode:=newcode or readbuffer[j];
                            j:=j+2;
                        end
                    else
                        begin
                            newcode:=(readbuffer[j-1] and $0F);
                            newcode:=newcode shl 8;
```

Anabilim Dalı : Mühendislik Bilimleri
Programı : Sistem Analizi
Çalışmayı Yapan : Tolga Ulus
Danışman : Doç. Dr. Mithat Uysal
Dönemi : 1992 - 1993 Kış

AĞAÇ YAPISININ LEMPEL-ZIV VERİ SIKIŞTIRMA ALGORITMASINA UYARLANMASI

Anahtar Kelimeler: Veri sıkıştırma, şifreleme, veri gereksizliği, adaptif şifreleme, Lempel-Ziv şifreleme, ikili ağaç.

Özet: Veri sıkıştırma, hem veri saklama, hem de iletişim alanlarında zaman ve yer açısından büyük avantajlar sağlamaktadır. Bu çalışmada, önce veri sıkıştırma algoritmaları incelenmiş ve ardından adaptif veri sıkıştırma algoritmalarından Lempel-Ziv yöntemi üzerinde yoğunlaşılmıştır. Bu algoritmada, dizi veri yapısı yerine ikili ağaç veri yapısı önerilmiştir. Bu yapıyla algoritmanın daha hızlı veri sıkıştırdığı görülmüştür.

APPLICATION OF TREE STRUCTURE TO LEMPEL-ZIV DATA COMPRESSION ALGORITHM

Keywords: Data compression, coding, data redundancy, adaptive coding, Lempel-Ziv codes, binary tree.

Abstract: Data compression has many advantages in information storage and data communication areas from the point of storage space and transmission speed. This study first examines the data compression algorithms and then focuses on Lempel-Ziv algorithm, one of the adaptive algorithms in the area. For this algorithm, the binary tree data structure over array data structure is proposed. It is observed that this modification speeds up the algorithm.

```
                    newcode:=newcode or readbuffer[j];
                    inc (j);
                end;
            toggle:=not toggle;
        end;

    procedure putfile;

        procedure writestring (p : nodeptr);
            begin
                if (p^.prevbyte = nil) then
                    firstbyte:=p^.byteinfo
                else
                    writestring (p^.prevbyte);
                writebyte (g,p^.byteinfo);
            end;

        begin
            if (first) then
                begin
                    abyte:=newcode;
                    writebyte (g,abyte);
                    oldcode:=newcode;
                    prevnode:=start[oldcode];
                    first:=false;
                end
            else
                begin
                    if (newcode > nextavailcode-1) then
                        begin
                            writestring (start[oldcode]);
                            writebyte (g,firstbyte);
                        end
                    else
                        writestring (start[newcode]);
                    prevnode:=start[oldcode];
                    if (prevnode^.nextbyte = nil) then
                        currnode:=nil
                    else
                        begin
                            currnode:=prevnode^.nextbyte;
                            while (currnode^.nextcode <> nil)
and (currnode^.nextcode^.byteinfo <= firstbyte) do
                                currnode:=currnode^.nextcode;
                        end;
                    insert (currnode,prevnode,firstbyte);
                    oldcode:=newcode;
                end;
        end;

    begin
        toggle:=true;
        first:=true;
        writesize:=0;
        assign (g,fnameout);
        rewrite (g,1);
        currnode:=head;
        prevnode:=head;
        assign (f,fnamein);
        reset (f,1);
        abyte:=0;
        repeat
```

```
                blockread (f,readbuffer,readbuffersize,readsize);
                j:=1;
                while (j <= readsize) do
                    begin
                        convertfrombyte (newcode,j);
                        putfile;
                    end;
            until (readsize < readbuffersize);
            close (f);
            blockwrite (g,writebuffer,writesize);
            close (g);
        end;

    function fileexist (fname : string) : boolean;
        var
            f : file;
        begin
            {$I-}
            assign (f,fname);
            reset (f);
            close (f);
            {$I+}
            fileexist:= (ioresult = 0);
        end;

    begin
        fnamein:=paramstr(1);
        fnameout:=paramstr(2);
        if (fnameout = '') or (fnamein = '') then
            writeln ('Invalid syntax : "LZWX infile outfile"
required')
        else
            if fileexist (fnamein) then
                begin
                    nextavailcode:=256;
                    initialize;
            {        displaytable (head, 0);}
                    lzwdecompress;
            {        displaytable (head, 0);}
                end
            else
                writeln ('Error : infile does not exist');
    end.
```

ÖZGEÇMİŞ

Tolga Ulus, 1965 yılında İskenderun'da doğdu. İlkokul ve ortaokulu Ankara'da okudu. 1983 yılında Ankara Lisesi'ni bitirdi. 1988 yılında Boğaziçi Üniversitesi Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümü'nü bitirerek Bilgisayar Mühendisi ünvanını kazandı. Aynı yıl, Boğaziçi Üniversitesi Meslek Yüksekokulu Teknik Programlar Bölümü Bilgisayar Programcılığı Programı'nda Öğretim Görevlisi olarak göreve başladı. Halen bu görevini sürdürmektedir. Kendisi IEEE Computer Society öğrenci üyesidir.