1 Giriş

Türk Dil Kurumu (TDK) sözlüklerinde "haberleşme" teriminin karşılıkları içinde bu kitabın konularıyla ilgili görünen tanımlar aşağıdadır;

- Bir yerden, bir kişiden, bir makineden bir başkasına, herhangi bir ortamdan yararlanarak bilgi gönderme.
- Telefon, telgraf, radyo gibi aygıtlarla yapılan bildirişim; bu yollardan yararlanarak yürütülen bilgi alışverişi.
- Kişiler veya kişiler ile teknik cihazlar arasındaki bilgi ve haber aktarımı.

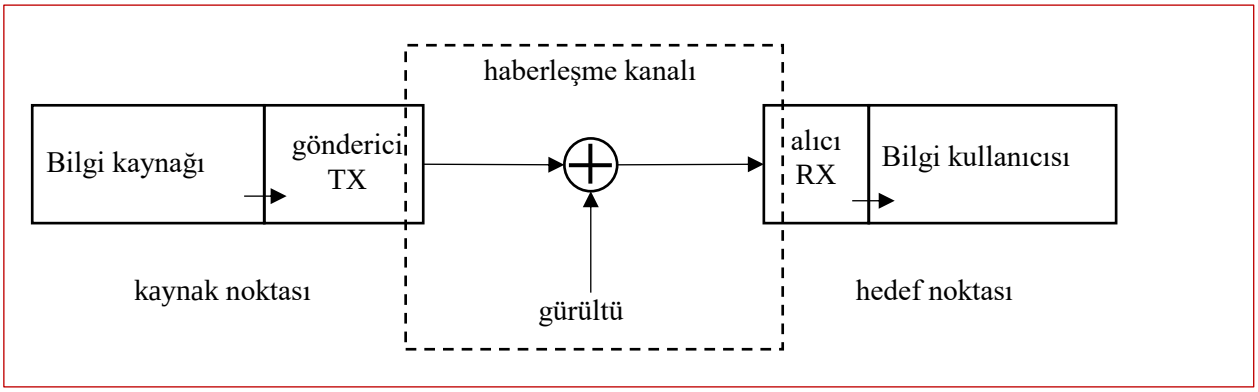
Yanlışlıkla birbiriyle eşanlamlı kullanılabileceğinden, *bilgi* ve *veri* olgularının bu kitapta kullanılış tanımlarının yapılması gerekir. Kitabımız elektronik haberleşme ile ilgili olduğundan;

Bilgi = asıl iletilmek istenen,

Veri = bilgiyi içinde barındıran

şeklinde özetleyelim. Özetin yeterli olmadığını bildiğimizden, bu bölümü iki kavramı birbirinden ayırıcı ve bunları matematiksel olarak birbirine bağlayıcı düşünce şekli oluşturmaya yönelik örneklerle ve tanımlarla donatalım.

Şekil 1.1 bir haberleşme sistemini özetlemektedir. Gönderici, kaynaktan aldığı bilgi taşıyan verileri elektromanyetik işaretlere çevirerek haberleşme kanalına aktarır. Alıcı ise, kanaldan aldığı işaretleri kullanıcının dilindeki veriye dönüştürür. Kaynak ve kullanıcı insan olmak zorunda değildir ancak gönderici ve alıcının ortak bir elektromanyetik işaret dili vardır. Elektronik haberleşme sistemlerinde çoğunlukla ilgilendiğimiz kısım, doğal olarak, verinin elektriksel işarete ve tekrar veri haline çevrildiği bloklar arasındaki kısımdır. Elektriksel işaret ve elektromanyetik işaret terimlerini birbirlerinin yerine geçecek şekilde kullanıyoruz. Çünkü Maxwell denklemlerine göre elektrik ve manyetik alanların değişimleri birbirini doğuran oluşumlardır. Yani değişken bir elektrik alanı değişken manyetik alanı doğuruyor, değişken manyetik alan da değişken elektrik alanına sebep oluyor. Birazdan göreceğimiz sebeplerden dolayı da, haberleşme açısından, bunların değişimi kaçınılmaz. Birisi varsa diğeri de vardır. Biz hemen her zaman elektrik alanını kontrol ediyoruz, manyetik alan ise kendiliğinden oluşuyor. Bunun sebebi, teknolojinin o şekilde gelişimi olabilir.



Şekil 1.1. Haberleşme sistemi özeti.

Burada dikkatimizi çekmesi gereken şey henüz bilgiden bahsetmemiş olmamızdır. O halde, bilgi ve veri kavramlarını ayırmamıza/birleştirmemize yarayacak basit örneklerle başlayalım.

"*Yarın güneş doğacak*" cümlesini ele alalım. Anlamı, hepimizin bildiği gibi "*küre şeklindeki dünyamızın bizim bulunduğumuz tarafı, kendi etrafındaki döngüsünün önemli bir kısmını tamamlayarak etrafında döndüğü güneş ismini verdiğimiz yıldıza doğru bakmaya meyledecek*" demektir. Biraz astronomi bilenler, bu cümleyi istediği kadar geliştirebilir, yanlışlarını düzeltip eksiklerini tamamlayabilir. İşin ilginç tarafı da "*yarın*" dediğimiz şeyin zaten bu döngü ile tanımlanmış olmasıdır, yani güneş ile dünya arasındaki ilişki bu şekilde olmasaydı, yarın diye birşey de olmazdı. "Hatta biz de olmazdık" diyenlere cevabımız, "biz varız ve bu cümlenin anlamı bu değildir" olurdu. Cümlede ima edilen edebi anlamı yok sayarsak, aslında bu cümle ile kimseye birşey kazandırmıyor, bir bilgi vermiyoruz. Yani ilettiğimiz mesajın bilgi değeri 0 (sıfır). Çünkü bu döngünün tekrar etme olasılığı 1 (bir). O halde, madem ki bu durum herkesin malumu, bunu bir cümle ile paylaşmaya gerek var mı? (Sadece haberleşme kitaplarında var.) Sonuçta veri var ama içeriğinde bilgi yok.

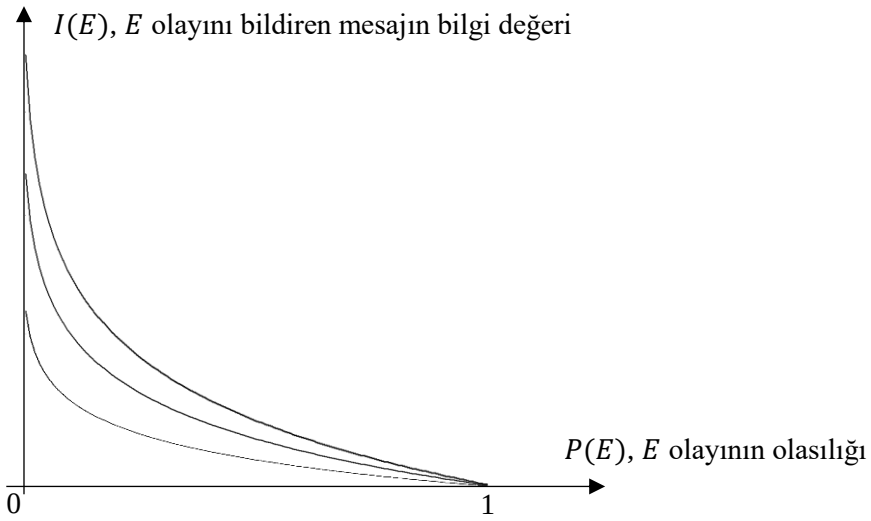
Şimdi de güvenilir bir kaynaktan gelen "*yarın borsa %25 düşecek*" cümlesini irdeleyelim. Borsaların bir günde %2'den fazla düşmesi olağan dışı bir durumdur. O halde bu cümle bize önemli bir bilgi aktarıyor. Yani cümlenin (verinin) bilgi değeri yüksektir (olasılığı düşüktür). Olayın olasılığı ile bilgi değerini ters olarak ilintilendirdik. Şekil 1.2 sözü dolandırmadan grafik olarak ifade ediyor. Şekilde karakteristikleri aynı olan 3 farklı grafik çizilmiş. Bunlar aslında 10, e ve 2 tabanlarına göre, $P(E)$ ve $I(E)$ sırasıyla olayın olasılığı ve cümlenin bilgi değeri olmak üzere,

$$I(E) = \frac{1}{log(P(E))} = -log(P(E)) \tag{1.1}$$

grafikleri. $I(E)$ değerine E olayının öz bilgisi diyoruz. Yine sözü dolandırmadan, $(0^+, 1^-)$ aralığında değerler alabilen olasılığı, $(\infty^-, 0^+)$ aralığındaki bilgi değerlerine haritalamak için kullanılmış diyelim. Yatay eksendeki 0 ve 1 dahil değildir, çünkü bunlar zaten birçok olay içinde ilgilendiğimiz olayın olasılığını belirtmiyor. Olasılık değerinin geçerli olabilmesi için olasılığı 0 olmayan karşıt bir olayın da var olması gerekir. 0 ve 1 bunu sağlamayan, ilgilenmediğimiz durumlar. Yarın güneşin doğmayacak olması gibi.

Özet olarak, bir olayı bildiren mesajın bilgi değerinin olması için toplamları 1 olan en az 2 olay olasılığı olması gerekir. Bu da demektir ki, bildirilen olayın ileride değişme olasılığı vardır. Yani ileride olayı bildiren mesaj da değişebilir. Bu durumda zamanla içeriği değişecek bir mesajdan bahsediyoruz. Mesajları elektriksel işaretlerle iletiyorsak, zamanla hiçbir özelliği değişmeyen elektriksel işaretlerle

iletişim yapılamaz diyebiliriz. Buna sabit voltaj dahil olduğu gibi, gerçekten peryodik işaretler de dahildir. Yani mesajlar elektriksel işaretlerin değişimi ile iletilir. Bu değişimlerin neler olabileceğine ileride değineceğiz.



Şekil 1.2. Bir olayın olasılığı ile olayı bildiren ifadenin bilgi değerinin ilintisi.

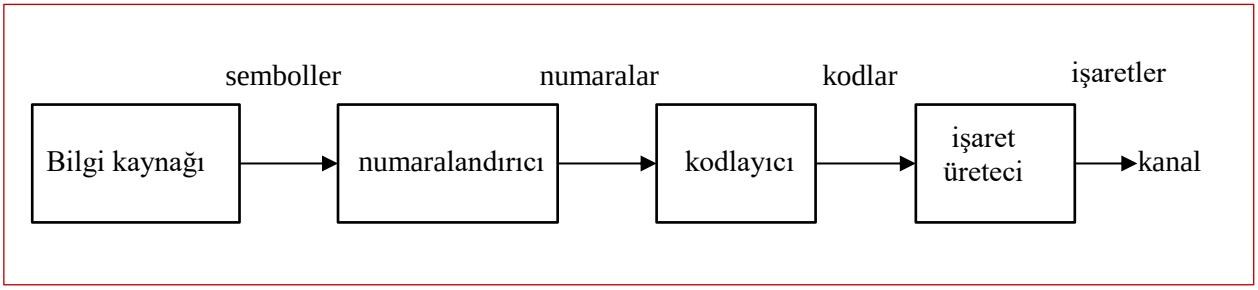
Elektrik-elektronik-haberleşme alanında bir miktar altyapımız olduğundan, logaritmayı hemen her zaman 2 tabanlı olarak görmeye/anlamaya alışkınız. O nedenle, aksi belirtilmediği sürece, bundan sonra $log(\cdot)$ yazsak ta $log_2(\cdot)$'yi ima ettiğimizi söyleyelim. Sebebini birazdan daha iyi anlayacağız.

Her mesajı/cümleyi bir diğerlerinden farklı bir sembol ile ifade edebiliriz, o nedenle bundan sonra kaynağın üretmesi olası her olayı bir sembol ile ifade edelim. Olası N adet (N sonlu pozitif sayı) olayı ifade edecek sembollerin sayısının da N olduğu açıktır. Bu olasılıklara $z = \{p_0, p_1, \cdots, p_{N-1}\}$ ($\sum_i p_i = 1$ olmak üzere), sembollere $A = \{a_0, a_1, \cdots, a_{N-1}\}$ alfabesi diyelim. Ayrıca $I = -log_2(u)$ olmak üzere $I = \{I_0, I_1, \cdots, I_{N-1}\}$ bilgi değeri setimizin de farkındayız. Örneğin sadece 2 adet olası durum var ise ve bu olayları temsil eden sembollere $A = \{a_0, a_1\}$ ya da $A = \{birinci, hayırdiğeri\}$ ya da $A = \{0, 1\}$ diyebiliriz. Sembollerin ne olduğuna bakılmaksızın bilgi değerlerimiz yine $I = \{I_0, I_1\}$ olacaktır.

Kodlama: Sembolleri, alıcı tarafına iletilmek üzere N adet birbirinden ayrılabilir işaret ile temsil ederiz, bu işaretleri de numaralandırırız. Sembolleri en baştan sıralı sayılardan seçmek büyük kolaylık sağlayabilir. Hatta numaralandırmada ikili sayı sistemini kullanmak daha da kolaylık sağlar. Çünkü, sayısal iletişim yapacaksak bu numaraları bir şekilde ikili sistemdeki sayılarla ifade etmek durumundayız. O halde, kaynak/gönderici tarafta yapılan işlemler Şekil 1.3 ile özetlenebilir. Şekildeki numaralandırıcı ile kodlayıcı beraber ele alınıp *kodlayıcı* ismi verilebilir, yapılan temsil sistemi değişikliğine de *kodlama* denilebilir. Özet olarak bizim açımızdan kodlama, olası her olayı ikili sayılardan oluşan bir dizi (string) ile temsil etmektir.

Kodlayıcı çıkışı elektriksel işaretlere çevrilmek üzere işaret üretecine gönderilir. Yani N adet olay N farklı işaret ile temsil edilir. Sürekli olarak temsil sistemi değişikliğinden bahsediyoruz. İşin özü de budur aslında. Ayrıca şu ana kadar ardışıl olayların birbirinden bağımsız olarak gerçekleştiğini varsaydık. Yani, "*vurdu*", "*gol oldu*" ve "*auta çıktı*" mesajları üç ayrı bağımsız olay gibi ele alındı. Ama biliyoruz ki "*vurdu gol oldu*" ve "*vurdu auta çıktı*" mesajları gibi de ele alınabilir. Yani olaylar önceki olaylara bağımlı olabilir. Bağımsız ise bu kaynağa *hafizasız kaynak* (memoryless source) diyoruz.

Kaynaktan üretilen sembollerin istatistiği daha önce üretilmiş sembollerin istatistiklerinden bir şekilde etkileniyor ise bu kaynaklara *hafızalı kaynak* (source with memory) denir. Quantum fiziğini de dahil edersek, her kaynak bir miktar hafızaya sahiptir diyebiliriz. Bu durumda en hafızasız dediğimiz zar atma olayı bile öncesinde gerçekleşen quantum olaylarına, örneğin zarın deformasyonuna bağlıdır sözüne kim itiraz edebilir. Yani hafıza, kaynağa hangi gözle baktığımıza bağlıdır.



Şekil 1.3. Veri kaynağı noktasında temsil değişiklikleri.

Genişletmeler: Herhangi bir sembol setindeki sembolleri yanyana koyarak daha fazla sayıda sembol elde ederiz. Örneğin $A = \{0, 1\}$ alfabesinin 3'lü genişletmesi $B = \{000, 001, \cdots, 111\}$ olur. Bunun 10'luk sayı sistemindeki rakamların yanyana koyulmasıyla istediğimiz sayıyı oluşturmaktan farkı yoktur. Bu şekilde, bir alfabenin *n*'inci genişletmesinden bahsedebiliriz; B'deki her sembol $A = \{a_k, k = 0, \dots r-1\}$'den alınan sembollerden oluşmak ve eşsiz/tek olmak üzere $B_i = b_0 b_1 \dots b_{n-1}, b_j = oneof\{a_k\}$. Bu şekilde genişletilmiş alfabedeki sembollerin olasılıkları toplamı da 1 olur. Yeni (sabit uzunlukta genişletilmiş) alfabedeki her sembolün olasılığı, $u = \{p_0, p_1, \dots, p_{r^n-1}\}$ olmak üzere, $p_i = \prod_j p_{b_j}$.

Örnek: $A = \{0, 1\}$ ve 3'üncü genişletme $B = \{000, 001, \cdots, 111\}$ ise

$$p(B_3) = p(011) = p(a_0)p(a_1)p(a_1) = p(0)p(1)p(1).$$

Genişletilmiş alfabedeki her sembol aynı genişletmeye/uzunluğa sahip olmak zorunda değildir. Örneğin $B = \{a, ab, caba, baca, \dots, bbb\}$ böyle bir alfabedir. Ancak, her sembol kaynaktan üretilmeli, kaynaktan üretilmeyen semboller de alfabede olmamalıdır. Bu şart sağlanınca hem orijinal hem de genişletilmiş alfabedeki sembollerin olasılıkları toplamı 1.0 olur. Bu şekildeki alfabelere *tam-alfabe* diyelim.

Örnek: Hilesiz bir madeni para ile yazı-tura atma olayında olasılıklar 0.5'tir. Yani yazı veya tura gelme olasılıkları eşit olduğuna göre bunları anlatacak ifadelerin bilgi değerleri de eşittir;

$$I(yazı) = I(tura) = -log_2(0.5) = 1.$$

Bu 1 birimlik bilgiyi ifade etmek ve iletmek için ikili sayı sisteminden 0 ve 1 sembollerini kullanalım. Yani, olayı (yazı-tura atma sonucunu) bildirmek için 1 bitlik bir sembol kullanıyoruz. Bilgi değerlerinin de 1 olması tesadüf müdür? Hayır, çünkü logaritmayı 2 tabanına göre aldık. İki olay üretebilen bir kaynakta oluşabilecek en yüksek ortalama bilgi değeri 1'dir. Zaten bit sayısı da $N_{bit} = 1, 2, \cdots$ gibi tamsayılar olabileceğinden olası en düşüğünü seçtik. Peki her sembolü 2 ya da daha çok bit ile temsil etme olanağımız var mıdır? Tabi ki. Örneğin $B = \{00, 011\}$ olabilirdi? Ama öncelikle bunun için güzel bir sebep bulmamız gerek, çünkü verimli bir temsil sistemi değil.

Bu kaynağın ikili genişletmesi $B = \{00, 01, 10, 11\}$ ve $z = \{0.25, 0.25, 0.25, 0.25\}$ olurdu. Sembollerden, örneğin 01, önce yazı ardından tura atılması anlamına gelirdi. Ancak unutmayalım; hafızasız bir kaynak varsayıyorsak, yazı-tura-yazı serisinin atılması hem 01 hem de 10 sembollerini gördük anlamına gelmeyecek. Yani o şekilde düşünmeyeceğiz. 01'i gördükten sonra geçmişe dair herşeyi unutacağız.

Ortalama Bilgi: Hafızasız kaynağın ürettiği sembollerin özbilgilerinin ağırlıklı ortalamasıdır. Yani,

$$I_{avg} = \sum_{i=0}^{N-1} p_i I_i = -\sum_{i=0}^{N-1} p_i log_2(p_i) \ [bit] \tag{1.2}$$

şeklinde yazılabilir.

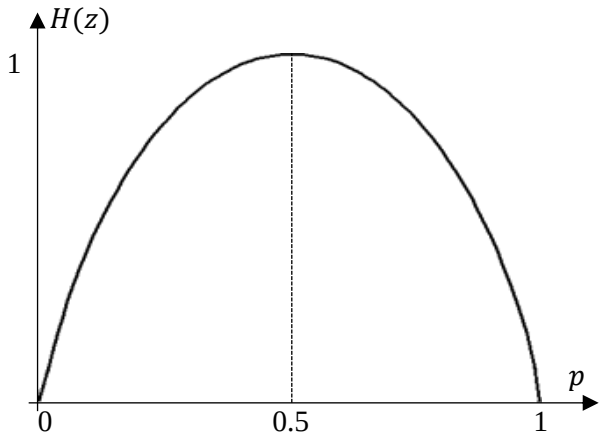
Denklem (1.2)'ye kaynağın *Entropi*si de diyoruz ve z olasılık seti olmak üzere $H(z)$ ile gösteriyoruz. Yani kaynağın entropisinin sadece sembol olasılıklarına bağlı olduğunu söylemiş oluyoruz. Entropinin birimi yoktur ancak logaritmayı 2 tabanı ile hesaplayınca sonuç bu sembolleri temsil edecek ikili dizilerin olabilecek en az ortalama bit sayısı olduğu için birim gibi bit yazdık. (not: entropi karmaşıklık demektir, bizde olasılıkların ne kadar düzensiz dağılmış olduğunu söyler).

Örnek: $z = \{p_0, p_1\}$ kaynağının entropisini bulalım.

$H(z) = -\sum_{i=0}^{N-1} p_i log_2(p_i) = -p_0 log_2(p_0) - p_1 log_2(p_1)$. $p_0 + p_1 = 1$ olduğunu bildiğimizden

$p = p_0 = 1 - p_1$ diyelim ve $H(z) = -plog_2(p) - (1-p)log_2(1-p)$ grafiğini çizelim.

Şekil 1.4'teki grafiğe ikili entropi (binary entropy) fonksiyonu ismi veriliyor. Entropinin (karmaşıklığın, karar verilemezliğin) en yüksek olduğu yer her iki olasılığın birbirine eşit olduğu (0.5) durumdur. Üç ya da daha fazla sembol içeren kaynakların entropi grafiklerini burada çizemememize rağmen en yüksek entopi değerlerinin olasılıkların eşit olduğu durumda gerçekleşeceğini görebiliriz.



Şekil 1.4. İkili kaynağın entropisinin olasılığa göre değişimi.

Örnek: Haberleşme dersinin harf notları $A = \{AA, BA, BB, CB, CC, DC, DD, FF\}$ ve notların olasılıkları (bir öğrencinin ilgili notu almış olma olasılığı) $z = \{0.05, 0.1, 0.15, 0.2, 0.2, 0.15, 0.05,$

0.1}'dir. Bu durumda bir öğrenci "*ben AA aldım*" deyince bize $I_{AA} = -log_2(0.05) = 4.32$ bitlik bilgi aktarmış olur. Peki dersi alan 90 öğrencinin notlarını öğrenirsek kaç bitlik bilgi almış oluruz?

$I_{toplam} = \sum_{ö=1}^{N_{öğrenci}} I_{öğrenci\_i}$ ya da $I_{toplam} = N_{öğrenci} \times I_{avg}$ ile bulabiliriz. Her öğrencinin aldığı notu bilmeye gerek olmadan

$I_{avg} = -\sum_{i=0}^{N-1} p_i log_2(p_i) = -0.05 \times log_2(0.05) - 0.1 \times log_2(0.1) - \cdots - 0.1 \times log_2(0.1) \cong 2.8464$ bit hesaplayıp $I_{toplam} = 90 \times 2.8464 \cong 256.1795$ bit şeklinde sonucu bulabiliriz. "Öğrencilerin notlarını bilmeden nasıl hesaplayabiliyoruz ki?" sorusunun cevabı zaten olasılıkların (z) bu notlarla hesaplanmış olması ya da notların bu olasılıklara göre verilmiş olmasıdır.

Her not için ortalamada 2.8464 bit iletilmesini nasıl sağlayabiliriz? Harfleri temsil eden ikili sembollerin uzunluğunu 3 bit yaparsak, yani $B = \{000, 001, 010, 011, 100, 101, 110, 111\}$ kodlarını kullanırsak ortalama 3 bit olur. Demek ki farklı bir kodlama yöntemi kullanmalıyız.

Not : (A, z) topluluğu (ensemble) hafızasız bilgi kaynağını tam olarak temsil eder.

1.1 Kodlama

Şekil 1.5 kaynaktan 5 farklı sembol üretilmesi durumu için birkaç farklı kodlama şekli öneriyor. Şimdilik olasılık kolonunu görmezden gelelim. Bunlardan Kod-1 dışındakiler değişken uzunlukta kodlar içeriyor. Şimdi bu örnek kodları inceleyerek hangi kodu kullanmanın avantajlı olacağına bakalım.

p_i	Sembol	Kod-1	Kod-2	Kod-3	Kod-4	Kod-5	Kod-6	Kod-7	
0.36	S_0	000	0	1	1	0	00	0	
0.18	S_1	001	1	10	01	01	01	100	
0.17	S_2	010	10	100	001	011	10	101	...
0.16	S_3	011	11	1000	0001	0111	110	110	
0.13	S_4	100	100	10000	00001	01111	111	111	

Şekil 1.5. Farklı kodlama önerileri/denemeleri.

Kod-1'in avantajı oldukça kolay üretilebilmesi ve alıcı tarafından da tanınmasıdır. Tabi ki alıcının hangi kodun hangi sembole karşılık geldiğini bilmesi (aynı alfabeyi kullanması) gereklidir. Ayrıca bu kodları oluşturan işaret dizilerinin birbiri ardına (seri halde) alıcıya gönderileceğini unutmayalım. Yani, örneğin 011 kodu gönderilirken önce 0'ı temsil eden işaret, ardından 1'i temsil eden işaret, ardından yine 0'ı temsil eden işaret belirli aralıklarla gönderilecek. Tabi ki herbir kodu temsil eden 5 farklı elektriksel işaret de kullanılabilir. Ancak, bu yöntemin sınırlarının farkında olalım; 2048 farklı sembol/kod varsa 2048 farklı işaret mi üreteceğiz? Alıcı bunları birbirinden nasıl ayıracak? Günümüzün yüksek iletim oranlı iletişim sistemlerinde kullanılan yöntem her ikisinin birleşimidir. Yani, semboller alt-sembollere (alt-kodlara) bölünerek birbirinden ayrılabilir sayıda farklı işaret ile seri olarak iletilir. Örnek olarak S_i sembolünü temsil eden kod 0011010111 olsun. Bu kod 2'şer bitlik alt-kodlara ayrılır, yani 2^2=4 farklı alt-kod oluşturulur ve 5 alt-sembolün herbiri seri olarak 4 işaretten

biri kullanılarak alıcıya gönderilir. Alt bölmelerin kaçar bit olacağı iletişim hattının karakteristiklerine göre belirlenir. Bu konulara ve ilgili örneklere ilerleyen bölümlerde yer vereceğiz.

Bitlerin (ve onları temsil eden işaretlerin) birbiri ardına gönderilmesinin alıcıda yarattığı zorluk, sembol sınırlarının belirgin olmayışıdır. Örneğin alıcının gördüğü bitler ...0101001110... olsun. Hangi bitten sonra yeni bir sembol başlamaktadır? Aralarında virgül vb. yoktur ki. Kod-1 bu soruna bir miktar çözüm getiriyor. Çünkü alıcı 3'er bitlik kodlar alacağını biliyor. Başlangıçtaki 3 bit doğru olarak çözümlendiğinde ardından gelen 3'lükler de doğru çözümlenir. Şekil 1.5'te verilen diğer örnek kodlarda ise bu durum yok, kodlar değişken uzunlukta. Yine de o kodların bazıları tekil olarak çözümlenebilmekte (özebir kodçözülen: uniquely decodable). Bununla ilgili birkaç tanım yapalım.

özebir kodçözülen: alıcı tarafından alınan her kodun, farklı uzunlukta dahi olsa, temsil ettiği sembol doğru olarak bulunabiliyorsa bu koda *özebir kodçözülen* denir.

anında kod: alıcı tarafından alınan *özebir kodçözülen* kodlar ilgili en son bit alındığında çözümlenebiliyorsa *anında kod* (instantaneous code)ismi alır.

Burada hem kod tablosuna (alfabe) hem de tablo içindeki ikili dizilere kod demek biraz karışıklı yaratıyor da olsa zamanla hangisinin kastedildiğinin anlaşılacağı ümit ediliyor.

Kod-2'nin özebir kodçözülen olmadığını görüyoruz. Alıcı ...0101001110... dizisiyle karşılaştığında sembol sınırlarını belirleyemiyor. Örneğin, 11 alıcı tarafında S_3'mü yoksa S_1S_1 olarak mı çözümlenecek? Bu durum Kod-2'yi seri iletimde kullanışsız hale getiriyor. Ancak Kod-3, -4 ve -5 doğru olarak çözümlenebiliyor;

Kod-3 : Son bit hariç diğer bitler 0 olduğundan, alıcı 1'i gördüğünde kodun bittiğini anlıyor.

Kod-4 : İlk bit hariç diğerleri 0 olduğundan, alıcı 1'i gördüğünde önceki kodun bitmiş olduğunu anlıyor. Yani kod özebir kodçözülen ama anında değil, sonraki sembolün ilk bitini görmesi gerekiyor. Bu şekilde oldukça gecikmiş olarak çözümlenebilir kodlar da var.

Kod-5 : Kod-4'ten bir farkı yok, sadece 0 yerine 1, 1 yerine 0 kullanılmış.

Kod-6 ise diğerlerinin sahip olduğu bitiş biti özelliğine sahip değilmiş gibi görünüyor. Ancak Kod-6 da *anında özebir kodçözülen* bir koddur. Kod-6'yı incelediğimizde hiçbir dizinin diğerlerinin başlangıcında olmadığını görüyoruz. Örneğin S_0'ı temsil eden 00 kodu diğerlerinin başlangıcında yok. Alıcı ikinci 0'ı aldığında sembolü temsil eden dizinin bittiğini anlıyor. Aynı durum diğer S_i'leri temsil eden diziler için de geçerli. Buna önekli (prefix) kod ismi veriliyor.

Kod-7'nin de Kod-6 gibi *anında özebir kodçözülen* bir kod olduğunu görüyoruz. Sadece kodların uzunlukları farklı.

Bu değişken uzunluklu kodları neden gördük? Çünkü amacımız gönderilecek bilgiyi en az bitle temsil etmek, böylece kanal kullanımın zamanından tasarruf etmek. Aslında yapmayı hedeflediğimiz şeyin adı *veri sıkıştırma*.

Veri Sıkıştırma: Asıl iletilmek istenen bilgiyi (bilgi dizisini) toplamda en az bitle temsil etmek. Buna neden bilgi sıkıştırma demedik? Çünkü bilgi sıkışmaz, ancak onu temsil eden veriyi sıkıştırabiliriz. Nereye kadar sıkıştırabiliriz? Olasılıkları kullanarak bilgi değerinin hesabını görmüştük. Gönderilecek toplam bit sayısı gönderilmek istenen toplam bilgiden küçük olamaz, eşit ya

da büyük olabilir. Aynı şekilde sembol başına ortalama bit sayısı da sembol başına ortalama bilgiden (I_{avg}) küçük olamaz.

O halde, Şekil 1.5'teki olasılıkları da kullanarak ortalama bilgiyi (entropi) ve kodların ortalama kod uzunluklarını hesaplayalım. Böylece hangisinin en verimli temsil olduğunu bulalım.

$z = \{0.36, 0.18, 0.17, 0.16, 0.13\}$ olmak üzere entropi

$H(z) = I_{avg} = -\sum_{i=0}^{4} p_i \, log(p_i) = -0.36 \times log_2(0.36) - \cdots \cong 2.2162$ bit hesaplanır.

Kod-1 : Hiç hesaplamadan L_{avg}=3 diyebiliriz. Zaten tüm diziler aynı uzunlukta.

Kod-2 : Kullanılamadığı için hesaba gerek yok. Hesaplansaydı 1.59<I_{avg} bulunurdu ki buradan bu kodlamayla bilgi kaybı yaşanacağını anlarız. Neden kullanılamayacağına bir açıklama daha. Günlük hayatta neden böyle bir kayıp yaşanmıyor? Örneğin 15 (onbeş) sayısını söylediğimizde nasıl oluyor da bilgi bize kayıpsız olarak aktarılıyor, dinleyici sayının bittiğini ve devamının gelmeyeceğini anlıyor? Çünkü konuşma dilinde farkında olmadan kullandığımız ayraçlar var. "onbeş" sözcüğünden sonra gelen kısa sessizlik ya da yeni kelime sayının bittiğini söylüyor.

Kod-3 : $L = \{1, 2, 3, 4, 5\}$ olmak üzere $L_{avg} = \sum_{i=0}^{4} p_i L_i = 0.36 \times 1 + 0.18 \times 2 + \cdots$ =2.52 bit hesaplanır. Yani, Kod-3 Kod-1'den iyidir ama I_{avg}'e hala uzaktır diyebiliriz.

Kod-4,-5 : L aynı olduğundan Kod-3 ile aynı.

Kod-6 : $L = \{2, 2, 2, 3, 3\}$ olmak üzere $L_{avg} = \sum_{i=0}^{4} p_i L_i = 0.36 \times 2 + 0.18 \times 2 + \cdots$ =2.29 bit bulunur. Yani Kod-6 diğerlerinden daha verimli, I_{avg}'ye de biraz daha yaklaşmış.

Kod-7 : $L = \{1, 3, 3, 3, 3\}$ olmak üzere $L_{avg} = 2.28$ bit hesaplanıyor. En verimlisi Kod-7 görünüyor.

Şekil 1.5'teki kodların nereden geldiğini açıklamadan verimliliklerini hesapladık. Doğal olarak, bilgi kaybı yaşanmaması için, *özebir kodçözülen* bir kod kullanmamız gerektiğini anlıyoruz. Ayrıca, bunu sağlamak için $L_{avg} \geq I_{avg}$ olması gerektiğini biliyoruz. Tabi ki dileğimiz, en verimli kodlama için, $L_{avg} = I_{avg}$ olmasıdır. Ancak bunu pratikte hiçbir zaman sağlayamayabiliriz.

En iyi (en küçük L_{avg}'i veren) kodları üretmenin algoritmik bir yöntemi var mıdır? Blok kodlar için cevabımız evet. *Blok kod*, kaynağın olasılıkları hesaplanabilir ayrık semboller ürettiği ve her sembole yine ayrıştırılabilir bir kodun atandığı kod tablolarına verilen isimdir. Burada, "demek ki blok olmayan kodlar da var" dememiz gerekiyor. Kod-7 blok kod üreten yöntemlerin içinde çok kullanılan *Huffman Kodlama* algoritması ile üretilmiştir.

1.2 Huffman Kodlama

Olasılıklar içinde en küçük olasılığa sahip iki sembolün birleştirilmesi ve her iki sembole farklı birer önek (0 ve 1) atanması ile çalışır. Böylelikle iki sembolün toplam olasılığına sahip yeni bir sembol oluşur, semboller de birbirinden önek ile ayrılır. Bu şekilde, yeni oluşan sembol tablosunda aynı işlem tekrarlandıkça değişken uzunluklu bir kod tablosu oluşur.

Örnek : $A = \{s_0, s_1, s_2\}$ sembolleri, $z = \{0.7, 0.2, 0.1\}$ olasılıkları ve C_0={00, 01, 10} sabit uzunluklu kodlaması verildiğinde, en küçük iki olasılık s_1 ve s_2'ye ait olan 0.2 ve 0.1'dir. Birleştirilmiş sembole s_3 diyelim. Yeni setler $B = \{s_0, s_3\}$, $u = \{0.7, 0.3\}$ ve C_1={b, $\bar{b}$} olur. Yani, sembol sayımız 2 ise mecburen birisini 1 diğerini 0 ile kodlarız. Hangisine 0 ya da 1 atadığımızın önemi yoktur. Ancak biliyoruz ki s_3 aslında s_1 ya da s_2'den birisidir. Hangisi olduğunu belirtmek için bir bit daha kullanmamız gerekiyor. s_1 için c, s_2 için $\bar{c}$ kullanalım. Yine hangisinin 1 hangisinin 0 olduğunun önemi yoktur, yeter ki ayrıştıran bitler sağda (lsb) olsun. Böylece son durumda $A = \{s_0, s_1, s_2\}$ sembollerini temsilen C_1={b, $\bar{b}c$, $\bar{b}\bar{c}$} kodu ortaya çıkar. Örneğin C_1={0, 10, 11} ya da C_2={1, 00, 01} olası atamalardan ikisidir. b ve c'nin her türlü seçiminde üretilen kod *özebir kodçözülen* önekli kod olur. Msb'den başlayarak son bit alındığında hangi sembol olduğu anlaşılır.

Örnekte görüldüğü gibi olasılığı yüksek olan sembollere kısa kodlar atanır. Böylelikle ortalamada kod başına daha az bit kullanılmış olur. Huffman kod tablosu hazırlama işlemini biraz daha büyük bir sembol sayısı ile örnek yaparak detaylandıralım.

Örnek :

$A = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$ ve $z = \{0.2, 0.14, 0.12, 0.12, 0.1, 0.1, 0.08, 0.07, 0.05, 0.02\}$ veriliyor. Sabit uzunluklu kod kullanılsa B={0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001} kod tablosu oluşabilir ve L_{avg}=4 bit elde edilirdi. Entropi ise

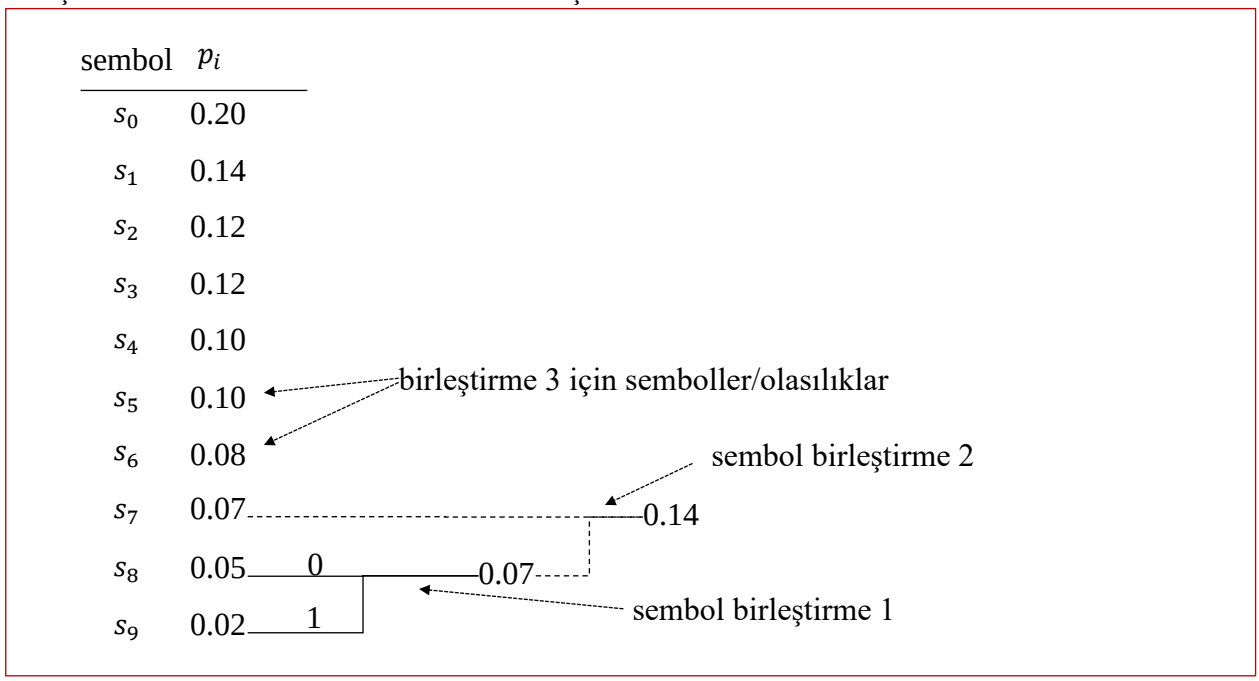
$H(z) = I_{avg} = -\sum_{i=0}^{9} p_i \, log(p_i) \cong 3.149$ bit bulunur. Yani normalde ortalama 3.14 bit ile temsil edilebilecek semboller 4 bit ile temsil ediliyor. Kodlama verimini

$$\eta = \frac{I_{avg}}{L_{avg}} \tag{1.3}$$

şeklinde tanımlarsak, sabit uzunluktaki kod için η =3.149/4 ≅ %79 bulunur. Değişken uzunluklu kod kullanarak verimliliği 0.2 (%20) puan arttırma şansımız vardır.

Şekil 1.6 örnekteki kaynağın sıralı olasılıklarını gösteriyor. Sıralı olduğu için en alttaki 2 olasılığın en düşük olasılıklar olduğunu görüyoruz. Bu iki sembolü birleştirilip olasılık toplamı birleşim yeri yanına yazılmış durumda. Birleştirilmiş iki sembolü birbirinden ayırdetmek için üsttekine 0 alttakine 1 önekleri atandı. Şimdi elimizde 9 adet sembol ve 9 olasılık var. Aynı işlem en düşük iki olasıkla tekrar edilmiş ve kesikli çizgilerle gösterilmiş. Bu ikisini birbirinden ayırmak için de üsttekine 0 alttakine (ve onun dallarındaki sembollere) 1 önekleri getirilecek. Böylece s_7'nin kodu 0, s_8 ve s_9'un kodları 10 ve 11 olacak. Bu işleme sadece 1 sembol kalana kadar devam edilince Şekil 1.7'daki Huffman ağacı elde edilir. Bu tek sembolün olasılığının 1 olmasına dikkat ediniz, aksi halde hesaplarda yanlış yapılmış demektir.

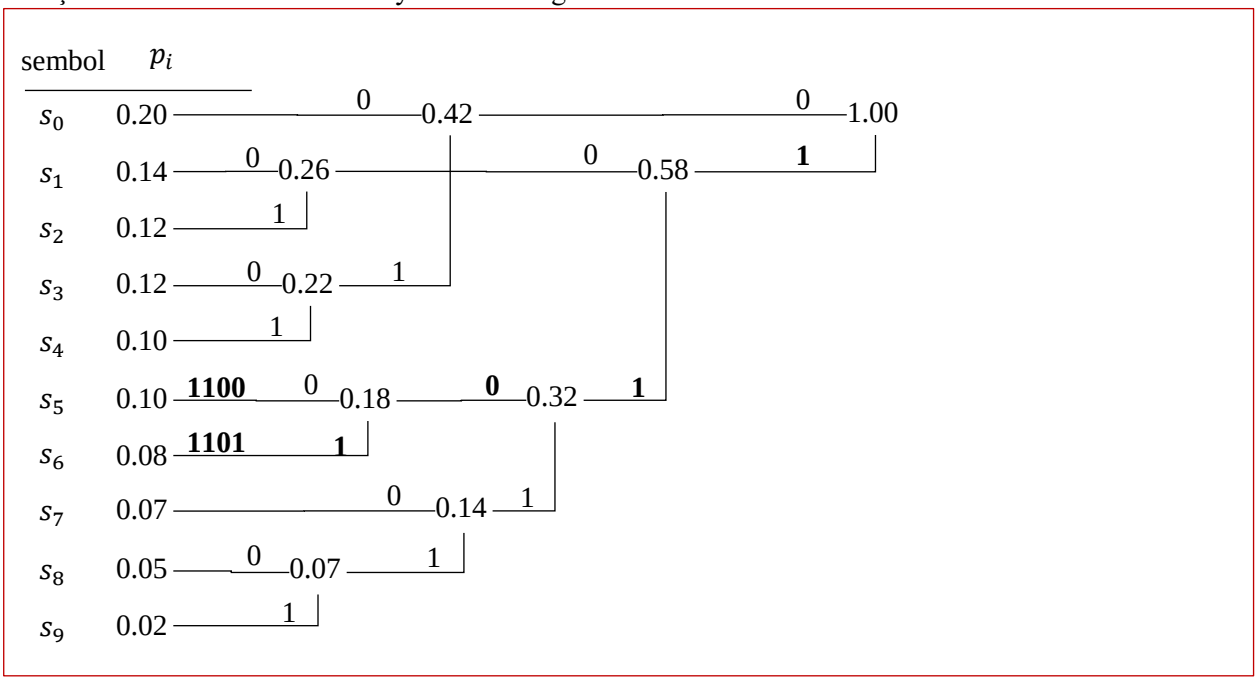
Şekil 1.6. Huffman kodlamada sembol birleştirme.



Daha pratik yöntem ise öncelikle olasılıkları birleştirip ağacı tamamlamak, bit atamalarını ise en sona bırakmaktır. Şekil 1.7'daki tamamlanmış Huffman ağacının ikili ağaç (binary tree) olduğuna dikkat ediniz. Sağdan sola doğru gidildiğinde (kökten yapraklara doğru) her kesişim yeri ikiye ayrılıyor. Bunlardan birisine 0 diğerine 1 dalı deyiniz. Bu şekilde tüm yapraklara kadar tüm dalları 0 yada şeklinde işaretleyiniz. Daha sonra, yaprakların kodunu bulmak için, kökten (msb) başlayarak yapraklara (lsb) doğru bit atamalarını birleştiriniz. Bu şekilde bulunmuş iki adet kod ve dalları üzerindeki bit atamaları kalın harflerle gösterilmiştir. Huffman ağacı oluştururken dikkat edilmesi gereken 3 nokta vardır;

- Daima en küçük iki olasılık birleştirilir. Eşit iki olasılık varsa hangisinin alındığın önemi yoktur.
- En sağdaki toplam olasılığın 1.00 olmasında dikkat edilir. 1.00 değilse bir yanlışlık yapılmış demektir.
- Sağdaki bit atamaları kodun daha önemli bitleridir. Yani kod, bit atamaları sağdan sola doğru en önemliden en önemsize doğru değişir. Birleştirirken buna dikkat edilir. Köke yakın bitler solda yer alır. Dikkat edilmezse özebir çözümlenemez bir kod ortaya çıkar.

Şekil 1.7. Huffman kodlama yöntemi örneği.



Şekil 1.7'daki örnek tamamlandığında z ={0.2, 0.14, 0.12, 0.12, 0.1, 0.1, 0.08, 0.07, 0.05, 0.02} olasılık setine sahip sembollerin C={00, 100, 101, 010, 011, 1100, 1101, 1110, 11110, 11111} kodları ile temsil edileceği bulunur. Başlangıçtaki sabit uzunluklu kodlamada her bir sembol 4 bit ile temsil ediliyordu. Şimdi ise bazıları 2, 3 veya 4 bit bazıları ise 5 bit ile temsil ediliyorlar. Kodlar kısaldığında temsil verimliliğinin artmasını anlayabiliyoruz ama 4 bitten 5 bite çıkan kodlar da var. Evet ama onların olasılıkları en düşük olanlar. Ortalamanın düşeceğini tahmin edebiliriz.

Kaynağın entropisini $H(z) \cong 3.149$ bit bulmuştuk. Yani, kodlamayı doğru yaptıysak temsil ortalamasını (L_{avg}) bu sayıya eşit ya da daha büyük ama sabit uzunluk olan 4'ten küçük bulacağımızı biliyoruz. Burada L_{avg}=3.19 bit bulunur.

Kodun özebir çözümlenen olduğunu kontrol edelim. En kolay yolu, kısa kodlardan başlayarak bir kodun diğerlerinin başlangıcında olmadığını görmektir. Burada olasılıkları büyükten küçüğe yazmanın faydasını görüyoruz. Böylece kodlar kısadan uzuna doğru sıralanır. Yani özebir çözümlenmeyi kontrol ederken sadece sağdakiler ile kontrol etmek yeterlidir, çünkü soldakiler zaten daha kısadır. Örneğin {1100, 1101, 1110, 11110, 11111} kodların hiçbirinin başlangıcında hemen soldaki 011 dizisinin olmadığını görüyoruz.

Kodlama verimliliğini $\eta = \frac{H(z)}{L_{avg}} = \frac{3.149}{3.19} \cong 0.987$ bulduk, bunun da sabit uzunluklu kodlama verimi olan $\frac{3.149}{4} \cong 0.787$'den çok daha iyi olduğunu gördük. Daha da iyi yapabilirmiydik? Genişletmeleri kullanarak verimi 1'e istediğimiz kadar yakınlaştırabiliriz. Bunu göstermek için daha basit bir örnek yapalım.

Örnek : z ={0.7, 0.2, 0.1} verilsin. Dağılımın entropisi

$H(z) = -\sum p_i \log(p_i) = -0.7log_2(0.7) - 0.2log_2(0.2) - 0.1log_2(0.1) \cong 1.157$ hesaplanır. Önceki örneğimizde bu dağılımı verimli temsil edebilecek Huffman kodlarının C_1={b, $\bar{b}c$, $\bar{b}\bar{c}$} olduğunu ve birisinin de C_1={0, 10, 11} olduğunu bulmuştuk. Buradan $L_{avg} = 1 \times 0.7 + 2 \times 0.2 + 2 \times$

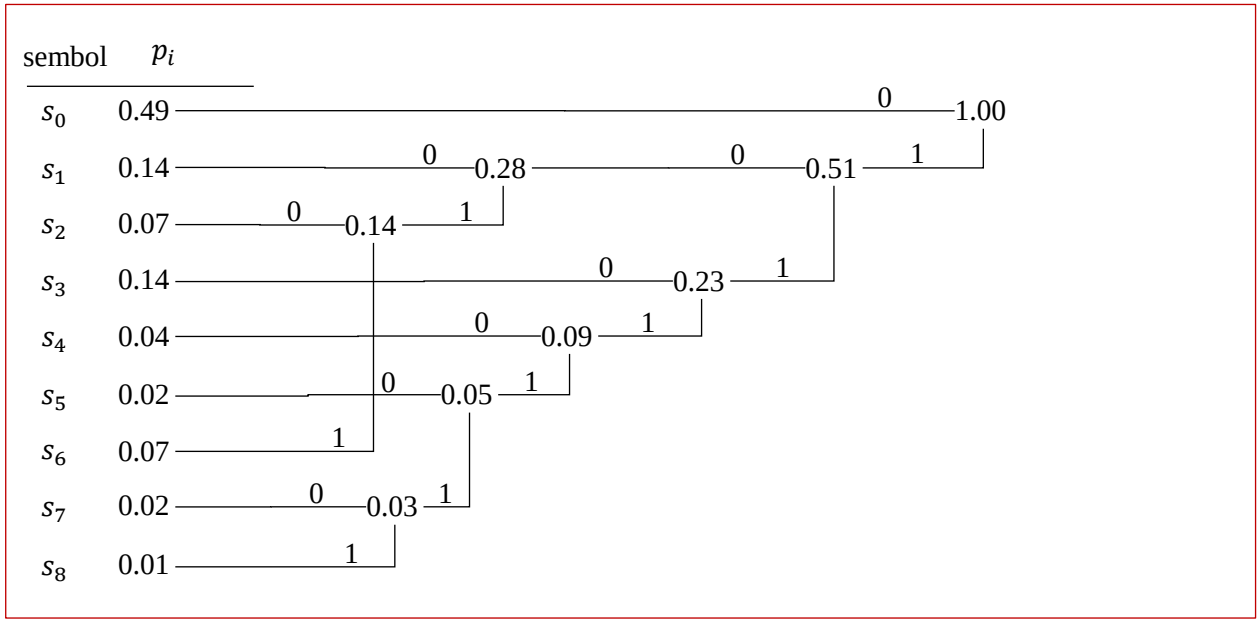
0.1= 1.3 bit bulunur. Aslında olası diğer kodların da ortalama kod uzunluğunun aynı olduğunu biliyoruz, çünkü kod uzunlukları aynı. Verimlilik ise $\eta = \frac{H(z)}{L_{avg}} \cong \frac{1.157}{1.3} \cong 0.89$ hesaplanır.

Şimdi de olasılık setini genişletelim, yani yeni kaynaktaki her bir sembol başlangıçtaki 2 sembolün birleşimden oluşsun. Buna ikinci genişletme (second extension) diyelim (not: ismin önemi yok, bazı kaynaklar buna birinci genişletme diyebilir). Bu durumda yeni kaynağımızın olasılık seti

u ={0.7x0.7, 0.7x0.2, 0.7x0.1, 0.2x0.7, 0.2x0.2, 0.2x0.1, 0.1x0.7, 0.1x0.2, 0.1x0.1}

u ={0.49, 0.14, 0.07, 0.14, 0.04, 0.02, 0.07, 0.02, 0.01} olur. Entropisi ise $H(u) \cong 2.314$ hesaplanır. Genişletilmemiş alfabe için L_{sabit}=2 bit iken, genişletilmiş alfabede L_{sabit2}=4 bit olur. Genişletilmiş alfabe için Huffman kodunu bulalım. Şekil 1.8 Huffman ağacını ve bit atamalarını gösteriyor. Ağaç üzerindeki bit atamalarını kökten yapraklara dizdiğimizde

E_2={0, 100, 1010, 110, 1110, 11110, 1011, 111110, 111111} kodu bulunur. Olasılıkları büyükten küçüğe sıralamadığımızdan kodlar da kısadan uzuna sıralanmadı (gerek görmedik).



Şekil 1.8. Genişletilmiş kod üzerinde Huffman ağacı.

Yeni alfabemiz (Şekil 1.8'deki) için kod uzunlukları L_u={1, 3, 4, 3, 4, 5, 4, 6, 6} yazılır ve buradan da L_{uavg}= 2.33 hesaplanır. Yani genişletilmiş Huffman kodlanmış alfabe için kodlama verimi

$\eta = \frac{H(u)}{L_{uavg}} = \frac{2.314}{2.33} \cong 0.9931$ olur. Görüldüğü gibi verimliliği yaklaşık 0.89'dan 0.99'a çıkardık. Verimliliği 1.00'a daha yakın yapmak için bir genişletme daha yapabiliriz (ikinci genişletmenin ikinci genişletmesi yada genişletilmemişin 3üncü genişletmesi). Ancak, verimlilik zaten 0.99'u aştığı için alacağımız sonuç göstereceğimiz çabaya değmeyebilir.

Bu örnekte birşey daha dikkatimizi çekiyor; $H(u) = 2H(z)$. Bu bir rastlantı değildir. Gerçekten de, $H(z)$ ve $H(u)$ sırasıyla genişletilmemiş ve genişletilmiş kaynakların entropileri ve n genişletme miktarı olmak üzere

$$H(u) = nH(z) \tag{1.4}$$

dir. Yani örnekte, genişletilmiş kaynağın entropisini $H(u) = -\sum p_j log(p_j)$ ile hesaplamak yerine $H(u) = 2H(z)$'den hesaplayabilirdik. Bunu, konudan sapmamak için, ispatlamadan söyledik.

Bazı kaynaklar için $L_{avg} = I_{avg}$ elde edebiliriz. Bunun olabilmesi için gerekli şartı, L_{avg} ve I_{avg} formüllerini karşılaştırarak görebiliriz. Bunlar

$H(z) = I_{avg} = -\sum_i p_i \, log(p_i)$ ve

$L_{avg} = \sum_i p_i L_i$

idi. Buradan görüyoruz ki her kodun uzunluğu, temsil ettiği sembolün bilgi değerine eşit ($L_i = I_i$) olursa $L_{avg} = I_{avg}$ elde edilir. Logaritmayı 2 tabanına göre aldığımıza göre sembollerin bilgi değerleri, x bir tamsayı olmak üzere, $I_i = 2^x$ olursa $I_i = -log_2(p_i)$'den sembol olasılıklarının $1/2^x$ olması gerektiğini görürüz. Böyle bir örnek z ={0.5, 0.25, 0.125, 0.125}'tir. Tabi ki pratikte kaynak sembol olasılıklarını kendimiz belirleyemeyiz. O nedenle, bu tip örnekler hesaplama kolaylığı sağlaması için sadece sınavlarda karşımıza çıkar :)

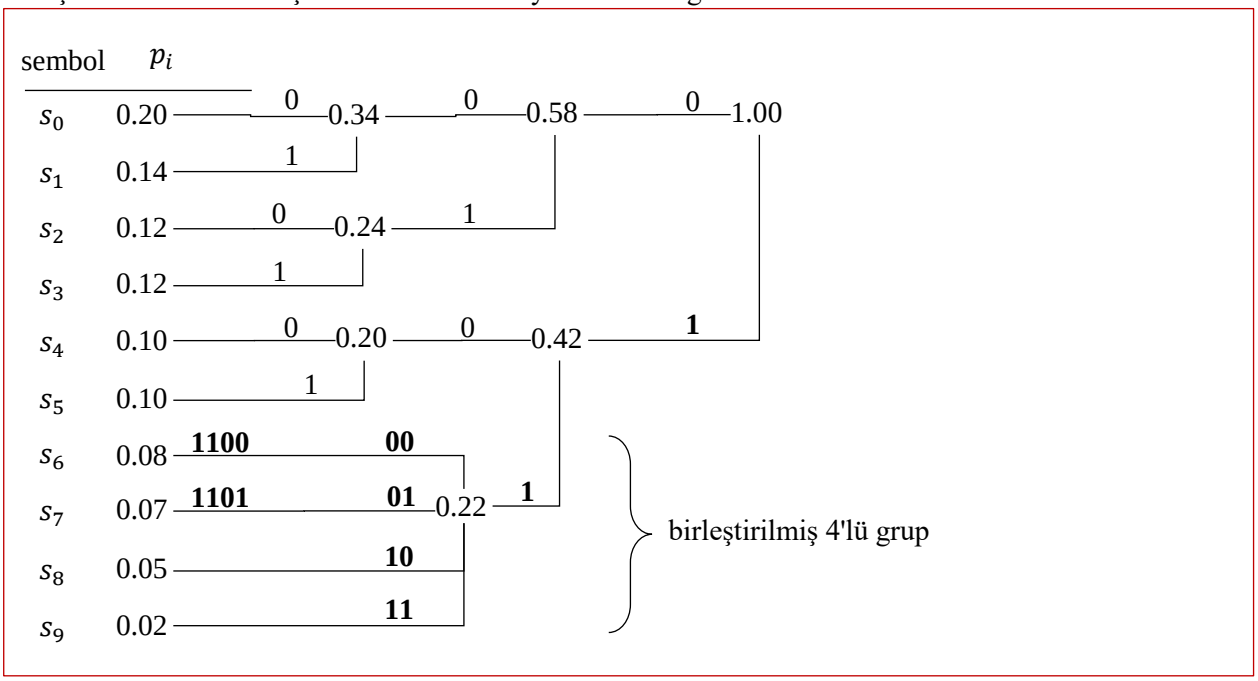
Huffman yöntemini ve Şekil 1.7 örneğini incelediğimizde 10 adet sembol için 9 adet birleştirme olabileceğini görüyoruz. N sembol sayısı olmak üzere $N - 1$ adet birleştirme gerekir. Bu arada, ikili kodlama yapabilmek için, üzerinde hesap yaptığımız ağaç dallarını da saklamamız gerekli olurdu. Bunun büyük N'ler için sorun olacağını, hatta aralarında çok küçük olasılık farkları olan ve sonuca (L_{avg}) çok az etki edecek şeyler için gereksiz çaba harcanacağını görebiliriz. İşlemleri daha basitleştirip yönetilebilir tutmak için *Kısaltılmış Huffman* yöntemi tercih edilebilir.

Kısaltılmış Huffman: Her basamakta sadece 2 adet sembolü birleştirmek yerine, bazı basamaklarda k bir pozitif tamsayı olmak üzere 2^k adet sembolü birleştirebiliriz. Bu yaklaşım ilk (olasılığı düşük olan semboller) basamaklarda önemli bir işlem tasarrufu sağlayabilir.

Şekil 1.7'daki örneğin içindeki en düşük olasılıklı 4 sembolün birleşimiyle oluşturulan Huffman ağacı Şekil 1.9'de gösteriliyor. Kısaltılmış ağaçta birleştirilen 4 sembolün kodlaması da 1'er bit yerine 2'şer bit atamasıyla yapılıyor. Bu durum şekilde en alttaki 4 dalın üzerine koyu harflerle yazılmış 2 bitlik atamalar ile ifade edilmiş. Bu gruba ait sembollerin iki tanesinin 4 bitlik kodları da koyu harflerle gösterilmiş.

Elde edilen alfabe TH ={000, 001, 010, 011, 100, 101, 1100, 1101, 1110, 1111} yine değişken kod uzunluklu elbette ama en kısa ve en uzun kodlar arasındaki uzunluk farkının Şekil 1.7'daki örneğe göre daha az olması kodlama verimin düştüğünü gösteriyor. Gerçekten de L_{avg_TH} =3.22 < L_{avg_huf} =3.19 bulunur. Yani kısaltılmış Huffman kodunun ortalama kod uzunluğu daha fazla. Verimliliği ise $\eta = \frac{H(z)}{L_{avg\_TH}} = \frac{3.149}{3.22} \cong 0.978$ (< 0.987) buluyoruz. Şekil 1.7'dan işlemlerin basitleştiğini görüyoruz ama badelini de verimlilikten kaybederek ödüyoruz.

Şekil 1.9. Kısaltılmış Huffman kodlama yöntemi örneği.



Tabi ki gerçek uygulamalarda Huffman kodlarını buradaki gibi el ile değil bilgisayar kodlarıyla üreteceğiz. Ancak çok sayıda sembol olduğunda bilgisayar zamanından tasarruf etme ihtiyacının ortaya çıkacağı açıktır, özellikle gerçek zamanlı uygulamalarda.

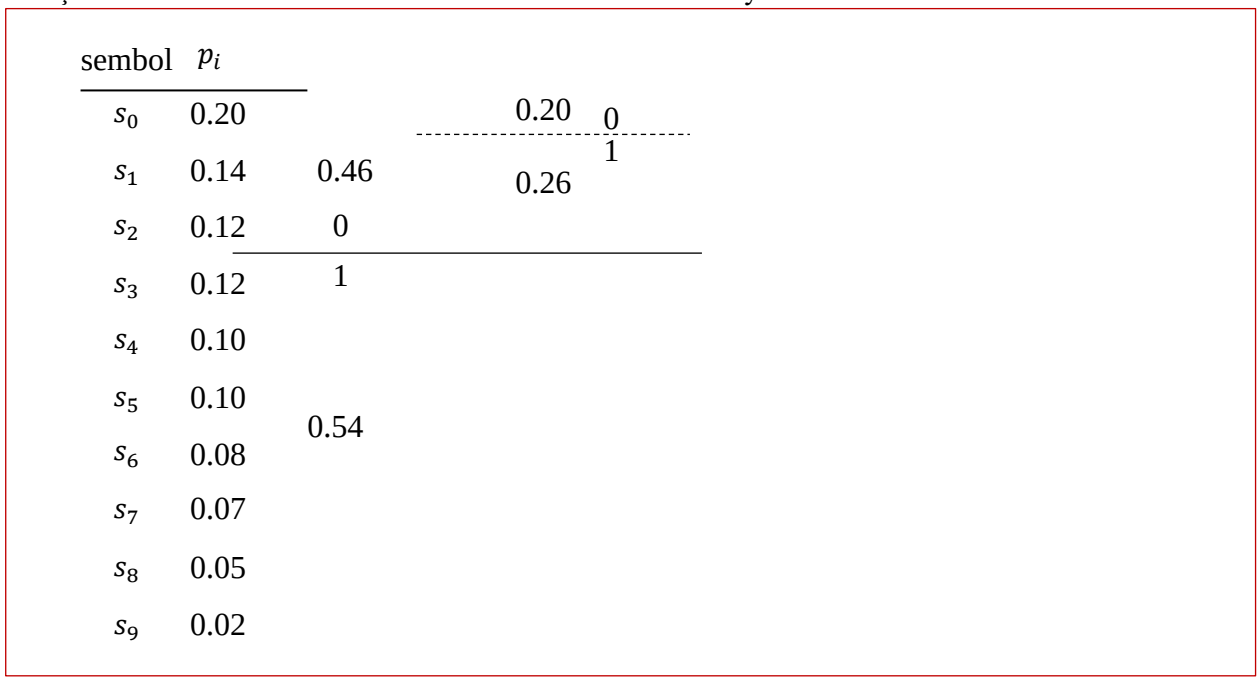
1.3 Shannon-Fano Kodlama

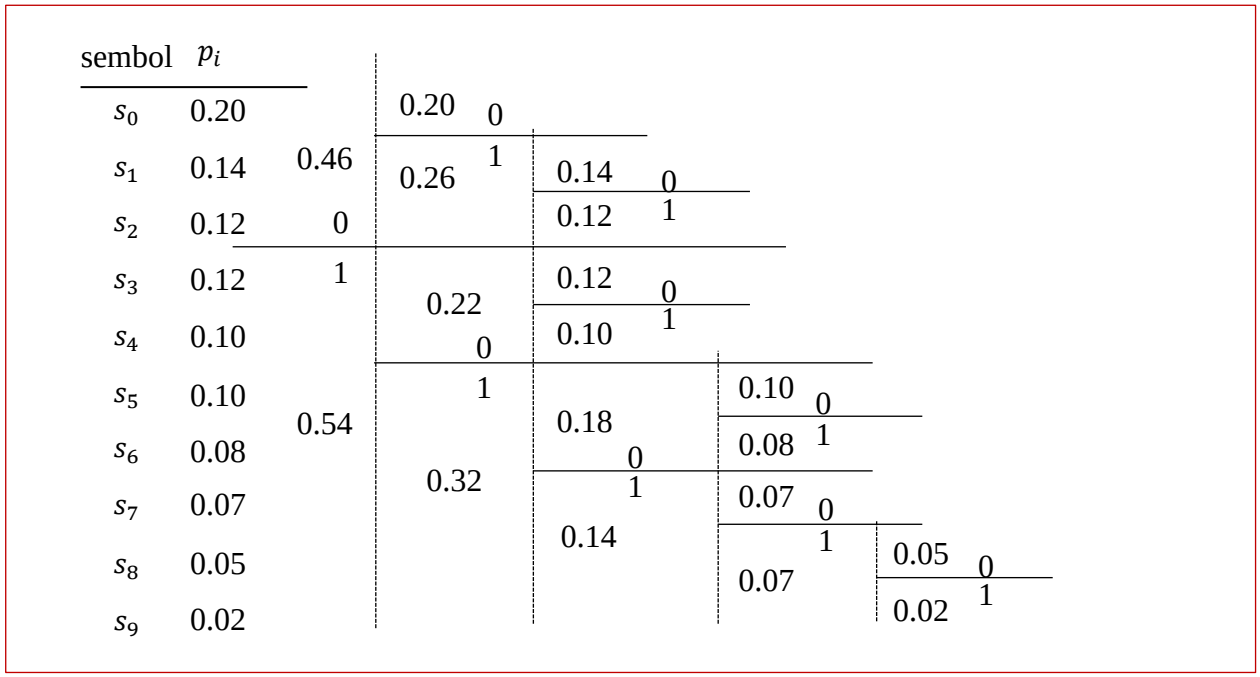
Olasılıkları büyükten küçüğe (ya da tersi) sıralanmış sembolleri, toplam olasılıkları eşit (ya da mümkün olduğunca yakın) iki gruba bölme ve gruplardaki kodlara 1 bitlik sonek atama yöntemi ile çalışır. Aynı gruptaki kodlara aynı sonek (1 ya da 0) atanır. Bu yöntemde, Huffman yönteminin aksine olasılıkları birleştirerek değil bölerek ve her bölmede en önemli bitten en önemsize doğru sonek atayarak kodlar üretilir.

Şekil 1.10, daha önce Huffman'ı açıklamak için kullandığımız, büyük olasılıktan küçüğe doğru sıralanmış olasılıklar içeren örnek üzerinde Shannon-Fano bölmesini açıklıyor. Toplam olasılık 1.00 olduğuna göre, ikiye böldüğümüzde her iki bölütte 0.5'e en yakın (aradaki fark en küçük) toplam grup olasılığı elde etmeye çalışıyoruz. Şekil 1.10'de görüldüğü gibi bu bölme {0.20, 0.14, 0.12}'yi içeren 0.46 olasılıklı bir grup ve aşağıda kalanları içeren 0.54 olasılıklı olasılıklı bir grup oluşturmakta. Gruplara b ve $\bar{b}$ en önemli bit değerlerini atıyoruz. Örneğin üstteki gruba 0, alttakine de 1 bit değerlerini atadık.

Şimdi her iki grubu da aynı yaklaşımla 2'ye bölüyoruz. Üstteki grubun ilk bölmesindeki toplam olasılık alt bölmesindeki toplam olasılığa en yakın olduğu bölütleme çizgisi kesikli olarak gösterilmiştir. Zaten 2 yerden bölme şansımız vardı. Her grupta sadece 1 olasılık kalıncaya kadar bu şekilde ikiye bölmeye ve bit atamaya devam ediyoruz ve sonuçta sembol sayısı kadar bölme elde ediyoruz (Şekil 1.11).

Şekil 1.10. Shannon-Fano kodlarının üretilmesinde bölme yöntemi.





Şekil 1.11. Shannon-Fano bölütlemeleri ve bit atamaları.

Tüm bölütlemeler tamamlanınca her bir sembolü temsil eden kod soldan sağa atadığımız bitlerin yanyana yazılmasıyla elde edilir. Böylece, örneğimiz için

SF={00, 010, 011, 100, 101, 1100, 1101, 1110, 11110, 11111} kod tablosu elde edilir. Kodların uzunluklarını aynı dağılımın Huffman ile kodlanmasında elde ettiğimiz C={00, 100, 101, 010, 011, 1100, 1101, 1110, 11110, 11111} kodlarının uzunluklarıyla karşılaştırdığımızda aynı olduğunu

görüyoruz. Çok farklı olmamakla beraber aynı çıkmasını beklemiyorduk. Örneğimizde tesadüfen aynı çıktı.

Şimdi de Huffman ve Shannon-Fano yöntemlerinin aynı sonucu vermeyeceği küçük bir örnek yapalım. Şekil 1.12 $z = \{0.36, 0.18, 0.17, 0.16, 0.13\}$ dağılımlı 5 sembol üreten kaynağın kodlamalarını gösteriyor. Huffman ikili ağacı ve Shannon-Fano bölütlemeleri alıştırma için öğrenciye bırakılmış.

sembol	p_i	Huffman	Shannon-Fano
s_0	0.36	0	00
s_1	0.18	100	01
s_2	0.17	101	10
s_3	0.16	110	110
s_4	0.13	101	101

Şekil 1.12. Aynı olasılıkların Huffman ve Shannon-Fano yöntemlerinde farklı kodlar üretmesi.

Kaynağın entropisi $H(z) = 2.216$ bit bulunuyor. Ortalama kod uzunlukları ise $L_{avg\_Huf} = 2.28$ bit ve $L_{avg\_SF} = 2.29$ bit hesaplanıyor. Huffman kodlamanın az da olsa daha verimli bir kod ürettiği söylenebilir. Gerçekte de özebir çözümlü blok kodlamalar arasındaki ilişkinin

$$H(z) \leq L_{avg\_Huf} \leq L_{avg\_SF} \leq L_{sabit} \tag{1.5}$$

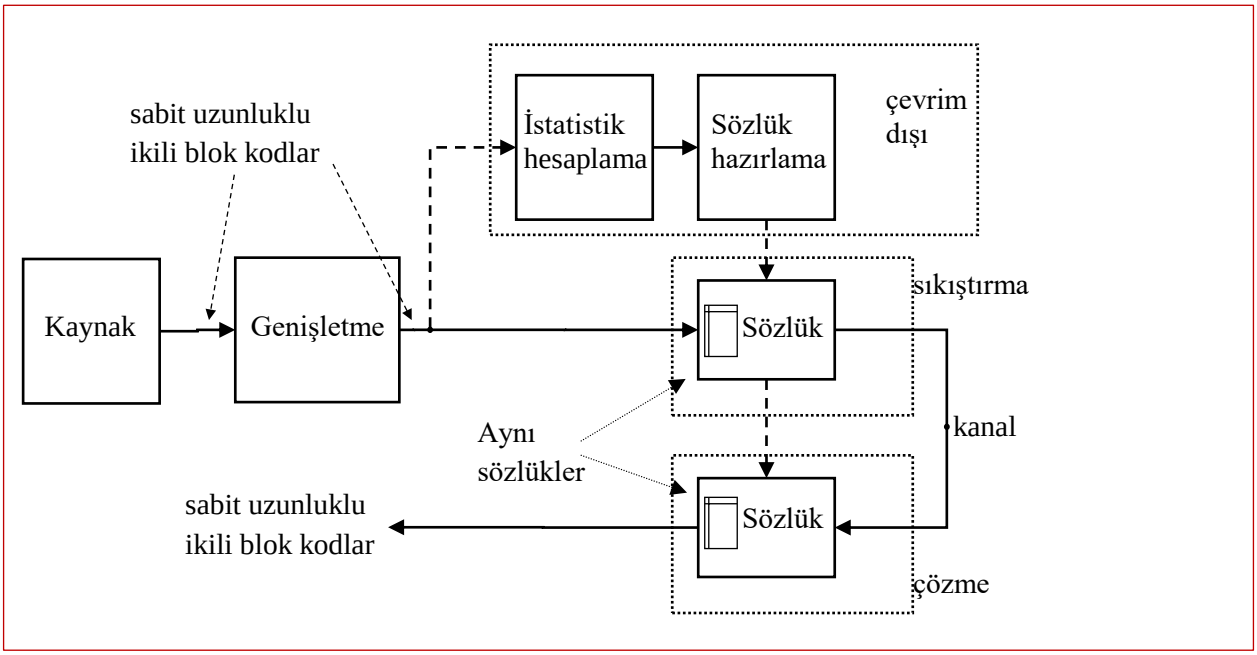
olduğu ispatlanabilir. Denklem (1.5)'teki karşılaştırma kısaltılmış Huffman kodlamasını içermemekte. Bu yöntemin de L_{avg_Huf} ile L_{sabit} arasında bir sonuç vereceği kolayca tahmin edilebilir.

Şu ana kadar aslında kod üretme olan işleme kodlama diyerek bir yanlış yaptık. Doğrusu, kodlama kelimesinin kaynaktan çıkan sembollere karşı gelen kodların kod tablosundan bulunarak çıkışa verilmesi işlemine karşılık gelmesi olmalıydı. Kod tablosunu ürettiğimiz Huffman yada Shannon-Fano gibi yöntemlere ise kod üretme denmeliydi. Daha da anlamlı bir terim ise *sözlük oluşturma* (dictionary creation) olur. Şimdi, bu öğrendiklerimizin daha anlamlı hale geleceği sözlük tabanlı veri sıkıştırma konusuna girelim.

1.4 Sözlük Tabanlı Veri Sıkıştırma

Kaynaktan çıkan sembollerin sabit uzunluklu kodlandığını ya da kaynaktan zaten sabit uzunluklu (genişletilmiş de olabilir) blok kodların çıktığını varsayalım. Bu noktada, öncelikle kaynak (ya da sembol) istatistiklerinin hesaplanması ve ardından bu istatistiklere dayanarak kod dönüştürme tablolarının oluşturulması gerekir. İstatistikler (sembollerin olasılıkları) önceden biliniyor da olabilir.

Kod dönüştürme tablosunun elde edilmesi ardından kaynaktan üretilen her sabit uzunluklu sembol için dönüştürme tablosundan karşılığı bulunarak çıkışa verilmesi işlemine *sözlük tabanlı veri sıkıştırma* denir.



Şekil 1.13. Sözlük tabanlı veri sıkıştırma akış şeması.

Sıkıştırılmış (ya da değişken uzunluklu kodlanmış) veri çıkışa (kanala ya da olası bir sonraki sayısal bir sonraki katmana) verilir. Çıkış bir veri saklama ortamı da olabilir. Benzeri şekilde kaynak da bir veri saklama ortamı olabilir. Bilgisayarın sabit diskindeki bir dosyanın sıkıştırılarak geri saklanmasını göz önüne getiriniz. Sıkıştırma sırasında kullanılan sözlük (örneğin Huffman kod tablosu) geri çözme sırasında da gereklidir. Yani, sözlüğün de saklanması (ya da iletilmesi) gerekir. Geri çözme yani asıl sabit uzunluklu sayısal sembolleri geri elde edilmesi, yine sözlükten arama ve bulunan kodun sabit uzunluklu karşılığını çıkışa verme (look-up) şeklinde gerçeklenir.

İstatistikleri bilinmeyen kaynaklar için öncelikle istatistiklerin bulunması gerekir. Bu işlem veri sıkıştırma değildir, ancak sözlük tabanlı sıkıştırma için gerekli bir adımdır. Ama çoğu zaman bu işlemin, kaynak değişmedikçe, bir defa yapılması yeterlidir. O nedenle Şekil 1.13'deki istatistik hesaplama ve sözlük hazırlama blokları çevrimdışı (offline) olarak gösterilmiştir.

Her ne kadar sembol kaynağını sonsuz sayıda ardışıl sembol üreten bir blok olarak düşünsek te, pratikte çoğunlukla karşılaşılan şey, sıkıştırılacak sonlu büyüklükte bir dosyadır. Yani, tüm dosya taranarak istatistikler oldukça net şekilde hesaplanabilir. Ancak, her kaynak böyle değildir. Örneğin mikrofondan alınan sayısallaştırılmış işareti gözönüne alalım ve ses işaretinin neredeyse gerçek zamanlı olarak sıkıştırılıp alıcıya gönderileceğini varsayalım. Burada yapılması pratik olan şey, işaretin kısa zaman aralıklarında (milisaniyeler) istatistiklerini hesaplayıp her zaman aralığı için yeni bir sözlük hazırlamak ve sıkıştırmayı gerçekleştirmektir. Tüm sesin (konuşmanın) bitmesini bekleyip ondan sonra sıkıştırmaya çalışmak ihtiyacı görmeyen anlamsız bir yaklaşımdır. Yine de, ilk kısa zaman aralığı için hesaplanan istatistiklerin tüm ses işareti boyunca geçerli, ya da daha önceden benzeri ses işaretleriyle hazırlanmış bir sözlüğün güncel, olduğunu varsayarak, üretilen sözlük tüm

işaret boyunca kullanılabilir. Bu, sıkıştırmada en verimli sonuçları üretmese de problemi çözmede yeterlidir. Bir diğer yaklaşım da istatistikleri devamlı olarak yeni verilerle güncel tutarak, ses işaretinin henüz alınmayan kısımlarının güncel istatistiklere uyumlu olmasını beklemektir.

Veri sıkıştırmada hedeflenen sonuç, aynı bilginin daha az veri ile temsil edebilmek, dolayısıyla veriyi saklamak için daha az saklama alanı kullanabilmek ya da veriyi iletirken daha az bant genişliği kullanabilmektir. Belki de tasarruf edilen bant genişliğinin iletişimde faydası olacak başka bir amaç için kullanılması mümkündür. Çünkü, bir bakış açısıyla, sıkıştırma sırasında atılan kısım tekrarlardır (redundancy).

Sıkıştırılan veriyi tekrar bir sıkıştırma işleminden geçirerek daha da kısa bir veri elde etme yanılgısına düşmemek lazımdır. İletilen veya saklanan sadece sıkışmış veri değildir, asıl veriyi geri elde etmek için gerekli diğer veriler (örneğin sözlük) de saklanır ya da iletilir. Uç nokta olarak, bir dosyayı sıkıştıra sıkıştıra 1 bit haline getirirsek, yanında sözlük olarak orijinal dosyayı da göndermemiz gerektiğini düşünelim. Tüm mesele, veri sıkıştırmada ve çözmede aynı yorumun yapılabilmesini sağlamak, bunun için de ortak bir çözümleme tablosu (örneğin sözlük) kullanmaktır. Herkesin kullanabileceği, iletilmesine ihtiyaç olmadan bilebileceği bir sözlüğü bir kez ve son kez üretmek mümkün değildir, mümkün olsaydı bile inanılmaz büyük olurdu. Ancak, burada hala geliştirmeye açık bir alan var. Verinin kayıpsız sıkıştırılabileceği en uç noktanın entropi olduğunu biliyoruz. Entropiyi ise sembollerin olasılıklarından hesaplıyoruz. Yani sembollerimizi istediğimiz gibi seçerek entropinin farklı çıkmasını sağlayabileceğimizi zannedebiliriz. Hafızasız ikili kaynaklar için bunun mümkün olmadığını hissedebiliyoruz. O halde, kaynakların hafızasız olduğu varsayımını terketmemiz gerekir. Zaten gelişmeler de bu yöndedir. Örneğin ses ve video sıkıştırmadaki modern yöntemler ses ve resim çerçevelerinin önceki çerçevelerle olan ilişkisini (temporal correlation) kullanmaktadır.

Şu ana kadar hedefimiz kaynakta üretilen verilerin sıkıştırıldıktan sonra hiçbir hatası olmadan geri kazanılması idi. Yani sıkıştırmaya giren her bitin çözme sonunda kayıpsız şekilde aynen geri elde edilmesi amaçlanıyordu. Takibeden bölümlerde anlatılan *aritmetik kodlama* ve *Lempel-Ziv* kodlama yöntemleri de aynı amacı güden ama başlangıçta bir sözlük hazırlamayan ya da sözlüğün sıkıştırma sırasında hazırlandığı yöntemlerdir. Bunlarda üretilen semboller önceki sembollere bağlı olduğu için sıkıştırma algoritması çıkışına hafızalı kaynak gözüyle bakabiliriz.

1.5 Aritmetik Kodlama

Aritmetik kodlamadaki yaklaşım, birçok sembolden oluşan bir girdi bloğunu bir gerçel sayı ekseninde bir sayı ile temsil etmektir (floating point). Bu yöntemde, her sembol için bir kod üretilen sembol-kod tablosu bulmaya ihtiyaç yoktur.

Huffman ve Shannon-Fano kodlama örneklerini çözerken temsilde bir sıkıntı olduğunu farketmiş olmalısınız. Sembol olasıkları ancak bilgi değerleri bir tamsayı olduğunda verimli bir şekilde temsil edilebiliyor, aksi halde entropiden uzaklaşılıyor. Aritmetik kodlama bu problemi önemli ölçüde aşmaktadır. 1987'de J.G. Witten tarafından tekrar gündeme getirilmeden çok önceleri prensipleri bilinen aritmetik kodlamanın bu konudaki başarısı gerçel sayıları temsil etmek ve bilgisayar programlarında kullanabilmek için reel sayılara ihtiyaç olmamasındandır. Ancak yöntemin çalışmasını

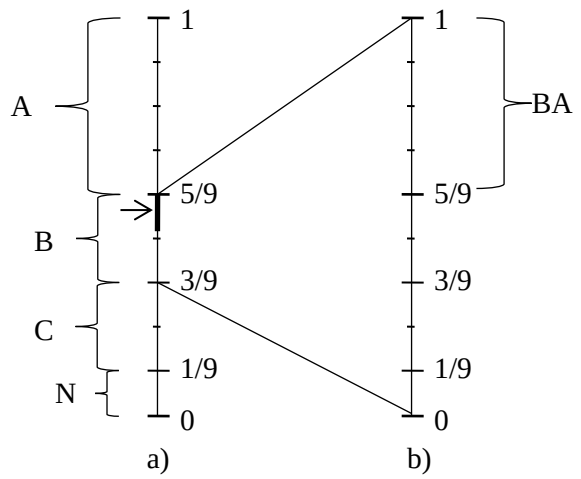
gerçel sayılarla açıklamak ve örneklemek çok daha kolaydır. O nedenle gerçel sayıları kullanarak bir örnek yapalım.

Örnek: Sıkıştırılacak verimiz S ={N, C, B, A} alfabesindeki sembollerden oluşan "BABACANCA" kelimesi olsun. Bu durumda, z ={1/9, 2/9, 2/9, 4/9} olmak üzere (S, z) topluluğu bu kaynağı tam olarak tanımlar. Aritmetik kodlamanın yaklaşımı, bu kelimenin (0,1) aralığında bir sayı ile temsil edilmesidir. Kelime ne kadar uzunsa, bu sayının da çözünürlüğünün o kadar yüksek olması beklenebilir. Bu semboller (0,1) aralığını gösteren eksen üzerinde kendi olasılıklarını belirten genişlikte aralıklarla yerleştirildiğinde (sıra önemli değil), alfabeden rastgele seçilen bir sembol ilgili aralıktaki herhangi bir sayı ile temsil edilebilir. Örneğin, Şekil 1.14a'da gösterildiği gibi B sembolü (3/9,5/9) aralığındaki herhangi bir sayı ile temsil edilir. Ok ile gösterilen 0.5 sayısı gayet uygundur. Bu şekilde kodlandığında, kod çözülürken (3/9, 5/9) aralığında herhangi bir sayı ile karşılaşıldığında kodlanan sembolün B olduğu anlaşılır.

B sembolü ardından A sembolü varsa, yani kelimemiz BA ise, B aralığı (0, 1) aralığındaymış gibi düşünülüp bu aralık içinde A aralığı belirlenir. Yapılan (3/9, 5/9)→(0, 1) haritalaması ile eşdeğerdir ve Şekil 1.14b'de gösterilmektedir. BA aralığı Şekil 1.14a üzerinde kalın çızgi ile gösterilmiştir. Bu aralıktan herhangi bir sayı BA kelimesini temsil eder. Bu aralığın alt sınırı $\frac{3}{9} + \left(\frac{5}{9} - \frac{3}{9}\right)\frac{5}{9}$, üst sınırı $\frac{5}{9}$ ve genişliği $\left(\frac{5}{9} - \frac{3}{9}\right)\left(1 - \frac{5}{9}\right) = \frac{2}{9} \times \frac{4}{9}$, yani her iki sembolün genişliklerinin çarpımı, olmaktadır.

B ve A'dan sonraki semboller için de (BACANCA) bu işlemler tekrarlanınca aralığın giderek daralacağı açıktır. Tüm semboller için işlemler tamamlanınca son üretilen aralık içindeki herhangi bir sayı kelimemizi ya da veri bloğumuzu temsil eder. Şekil 1.15a bu işlemlerin algoritmik akışını göstermektedir. Algoritma "BABACANCA" dizisi üzerinde S ={N, C, B, A} seti ve z ={1/9, 2/9, 2/9, 4/9} olasılıkları ile çalıştırıldığında L=0.504218425577384 ve H=0.504228998069330 değerlerini üretmektedir. Bu ikisi arasında verilen herhangi bir sayı (örneğin 0.50421) "BABACANCA" dizisini/bloğunu/kelimesini temsil etmektedir. Bu sayı ve (S, z) ikilisi beraberce saklanır ya da iletilirler.

Şekil 1.14. (0, 1) gerçel sayı aralığının aritmetik kodlama için bölütlenmesi. a) Tüm karakterlerin



aralıkları ve BA kelimesinin kalın çizgi ile gösterilmiş aralığı b) BA kelimesi için B aralığının genişletilmesi.

Şekil 1.15b ise bir gerçel sayı ve bloktaki sembol sayısı verildiğinde blok içindeki karakterleri sırasıyla bulan algoritmik akışı göstermekte. Kaç karakter üretileceği bilgisi kod çözme algoritmasına verilmez ise, algoritma BABACANCA kelimesinden sonra da semboller üretmeye devam eder.

```
L = 0.0
H = 1.0
Dizinin_Her_Karakteri(k)_İçin
  R = H - L
  H = L + R * H(z(k))
  L = L + R * L(z(k))
Sıradaki_Karakter
Çıktı = H_ve_L_arasında_bir_sayı
```

a)

```
X = kodlanmış_sayı
Tüm_Karakterler_Üretilinceye_Kadar
  X'ten_karakteri_(C)_üret
  R = H(X) - L(X)
  X = (X - L(z(C))) / R
Sıradaki_Karakter
```

b)

Şekil 1.15. Aritmetik kodlamanın algoritması. a) Kodlama. b) Kod çözme.

Bu algoritmaların normal bir bilgisayara gerçel sayılar kullanarak uygulanmasındaki sıkıntı, bilgisayarlarda gerçel sayıların tam olarak temsil edilememesindedir. Gerçele en yakın temsil sistemi kayan nokta (floating point) sayı temsil sistemidir (örneğin IEEE Standard 754). Burada sayılar sınırlı sayıda bit ile temsil edilebilmektedir (bilgisayarlardaki her sayı gibi). Bunun üstünde bir çözünürlük gerektiren sayılarda kesme ya da yuvarlama (truncation, round-off) hataları oluşmaktadır. Şekil 1.15a'daki algoritmanın belirsiz bir sembol sayısından sonra yuvarlama yapmaya başlayacağı açıktır. Yani blok büyüklükleri gerçek uygulamalarda kullanılamayacak kadar küçük olmaya zorlanmaktadır.

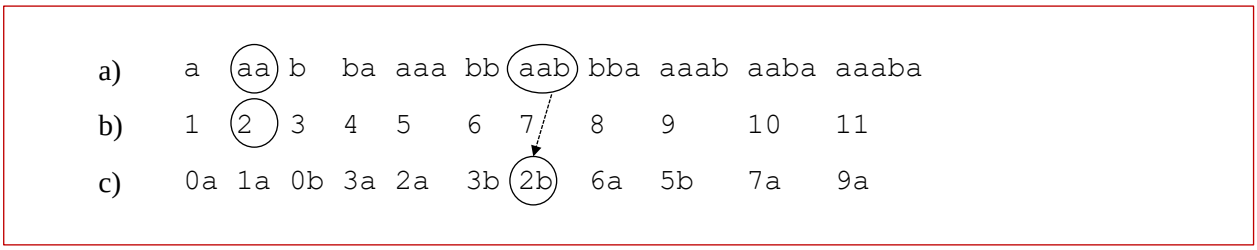
Aritmetik kodlama, esasları 1960'lı yıllardan beri bilinmesine rağmen bilgisayarlarda yüksek hassasiyetlerde sayılar saklanması/kullanılması mümkün olmadığından, 1987'de J. G. Witten'in gerçekte hassas sayılara ihtiyaç olmadığını gösterinceye kadar kullanılamamıştır. Witten'in aritmetik

kodlama algoritması tam sayıları (ikili) kullanır ve hafızada tutmaya ihtiyaç duyduğu basamak sayısı çok azdır.

1.6 Lempel-Ziv Kodlama

Sembol-kod tablosunu (sözlük) oluşturmayı kodlama işlemine paralel olarak yürüten Lempel-Ziv yaklaşımının en büyük avantajı istatistik hesaplamaması, bu nedenle de büyük dosyalarda, bahsedilen diğer yöntemlere göre, oldukça hızlı çalışmasıdır. Lempel-Ziv algoritması deyince LZ77 ve LZ78 yöntemleri akla gelmektedir. Bu iki yöntem aslında önemli ölçüde farklıdır. Ortak yanları ise girdi dosyasını okurken hazırlanan sözlüğün gerçekte sözlükte zaten var olan kelimeleri referans alması ve farklar için ilaveler getirmesidir. Örneğin sözlükte 15 numaralı yerde "ekleme" şeklinde bir kelime varsa, "eklemek" kelimesi de {15, k} şeklinde ifade edilir ve bu kelime için de sözlükte bir yer verilir. Ancak tüm kelimeler, örneğin "ekleme" de dahil, sözlükte daha önce yer almış kelimelere ilave yapılarak üretilir. Yani sözlüğün her elemanı, önceki kelimelere bir bağlantı ve eklenen sembol ile ifade edilir. Veri başlangıcında verimsiz bir yöntem olarak görülebilecek bu yaklaşım, girdi verisi ilerledikçe oldukça uzun kelimelerin kısa bağlantılarla ifade edilmesiyle iyi bir sıkıştırma sağlar.

Örnek: İkili sembollerden oluşan "aaabbaaaabbaabbbaaaabaabaaaaba" verilsin. İkili sembolleri 0 ve 1 yerine a ve b şeklinde yazdık ki birazdan ekleyeceğimiz diğer sayılarla karışmasın. Girdi sembolleri okundukça sözlük taranır ve henüz sözlükte olmayan bir dizi görüldüğünde {referans, sembol} ikilisi ile sözlüğe eklenir. Bu durumda girdi dizisini daha önce karşılaşılmayan alt-diziler şeklinde düşünmek yöntemin anlaşılmasını daha da kolaylaştırır. Yani örnek girdimiz "a", "aab", "ba", "aaa", "bb", "aab", "bba", "aaab", "aaba", "aaaba" şeklinde alt-dizilere ayrılır (Şekil 1.16a). Ayrılan alt-diziler numaralandırılır. Şekil 1.16b bu numaralandırmayı göstermektedir. Diziyi öncesinde karşılaşılmayan alt-dizilere ayırma işleminden çok kolaymış gibi bahsettik. Aslında bu, önceden bulunan alt-dizileri yeni dizi ile karşılaştırmak için taramak demektir ve yöntemin en çok zaman alan kısmıdır. Hızlandırmak için çeşitli arama yöntemleri devreye sokulabilir.



Şekil 1.16. a) Örnek dizinin alt-dizilere ayrılması. b) Numaralandırılması. c) Çıktı dizisi

Artık her bir alt-dizi, daha önce karşılaşılan alt-dizi referansı ve ilave edilen sembolü ile temsil edilebilir. Örneğin "aaba" alt-dizisi 7a ile temsil edilir. Burada 7, "aab" alt-dizisinin numarasıdır, a ise ilave edilen semboldür, yani aslında "aaba" alt-dizisi de 7a temsili ile sözlüğe eklenmiş olur. Tabi ki "aab" ikili dizisi yerine 7 sayısının yazılmasının sıkıştırmaya ne faydası olacağı argümanı öne sürülebilir. Örneğimizdeki gibi kısa dizilerde bir faydası olmaz. Ancak, girdi dizisi birkaç milyon sembolden oluştuğunda, alt-dizilerin boyları da uzar ve verimli bir temsil sağlanmaya başlanır. Özetle,

bu algoritma küçük dosyaları sıkıştırmakta bir işe yaramaz, çıktı dosyası girdiden büyük olur. Büyük dosyalarda ise Shannon limitine yaklaşmaya başlar. Aynı zamanda, bir istatistik hesaplanmadığından, bahsedilen diğer algoritmalara göre oldukça hızlıdır. Günümüzde sıklıkla kullanılan "zip" programlarının popülaritesi bu yüzdendir.

Şekil 1.16c tüm referansları yazılmış diziyi gösteriyor. "0a1a0b3a2a3b2b6a5b7a9a" dizisi sıkıştırılmış çıktı dizisidir. 0 (sıfır) referans numarası herhangi bir başlangıç dizisi olmadığını gösteriyor.

Sıkıştırılmış diziyi açma ve orijinal dosyayı üretme işlemi {referans, sembol} ikililerinden sözlüğü ve alt-diziyi üretme şeklindedir. Örneğin, 0a herhangi bir alt-diziyi referans almadığından "a" alt-dizisi, 1 yer numarası ile, sözlüğe ve çıktıya eklenir. Benzeri şekilde 1a görüldüğünde 1 nolu alt-diziye a eklenip "aa" oluşturulur ve sözlüğe eklenip çıktıya verilir.

Buraya kadar anlatılan sıkıştırma yöntemi LZ78'tir. LZ77 ise bir sözlük oluşturmak yerine girdi dosyasının o ana kadar okunan kısmını sözlük olarak kullanır. Öncesinde karşılaşılmayan her yeni alt-dizi için dosya da (okunan kısımda) geriye doğru bir referans (offset), sembol sayısı ve yeni sembol den oluşan {referans, sayı, sembol} üçlüsü üretilir. Örneğin, dosyamızın şu ana kadar okunan kısmı "aaabbaaaabbaabbb" ve son okunan alt-dizi ise "aaaaba" olsun. Alt-dizinin "aaaab" kısmı önceden görülen ve 6'ıncı (geriye doğru 11'inci) sembolden başlayan 5 sembollük dizi ile aynıdır. Bu durumda, {11, 5, a} üçlüsü üretilir ve çıktıya verilir. Burada, "aaaaba" dizisinin alt-dizileri olan "a", "aa", "aaa", "aaaa" ve "aaaab" için birşey üretilmediğine, çünkü bu dizilerin daha önce görüldüğüne dikkat edelim. Yani okunup yeni diziye eklenen her karakter için geriye doğru bir arama yapılıyor ve bulunmadığı durumda bir 3'lü üretiliyor. Büyük dosyalarda bu arama işlemi zaman alıcı olacağından dosyanın en son 2^B kadarlık kısmı (B, referans için ayrılan bit sayısı olmak üzere) hafızada tutulup bunun içinde arama yapılır. 2^B'ye arama penceresi ismi verilir.

LZR, LZSS, LZB, LZH, LZW, LZC, LZT, LZMW, LZJ, LZFG, LZS ve varsa burada adı geçmeyen diğer LZxx algoritmaları LZ77 ve LZ78'in değişik uyarlamaları (variants) olup fikir olarak benzerdirler. "Neden bu kadar çok uyarlama var?" sorusunun cevabı patent çatışmalarıdır.

1.7 Kayıplı Kodlama

Sıkıştırılmış veriden açtığımızda orijinal veri ile aynı olmaması durumuna, kayıplı sıkıştırmanın yüksek getirileri nedeniyle razı oluruz. Buraya kadar gördüğümüz kayıpsız sıkıştırma yöntemlerinin limiti kaynak entropisidir. Günlük hayatta kullandığımız verilerin çoğunluğu için, eğer başlangıçta zaten kabul edilebilir bir şekilde kodlandıysa, ortalama kod uzunluğu entropinin 2 ya da en fazla 3 katını geçmez. Yani, sıkıştırma ile en iyi ihtimalle veri miktarını 1/3'üne indirebiliriz. Ancak, taşınan bilgide %1'lik bir kayba razı olunması karşılığında sıkışmış verinin orijinalin 1/30'u kadar olabileceği söylense, atılacak bilginin önemine göre razı olabiliriz. Kayıplı sıkıştırma algoritmaları da bu yaklaşımla oldukça geniş kullanım alanı bulmaktadır. Örneğin internetten taşınan ve web sayfasında gösterilen resimlerin %99'u bu sınıfa girmektedir. Biraz renk bozulmasını, resim üzerinde ufak tefek hatalar olmasını umursamayız. Tam bir genelleme yapmak mümkün olmasa da, resim ve ses gibi örnekleme (sampling) kullanılarak üretilen verilerde zaten belli bir miktar hata olduğunu kabul edersek, kayıplı sıkıştırmanın bu tür verilere uygun olacağını söyleyebiliriz. Karşılığında, 3 saniyede

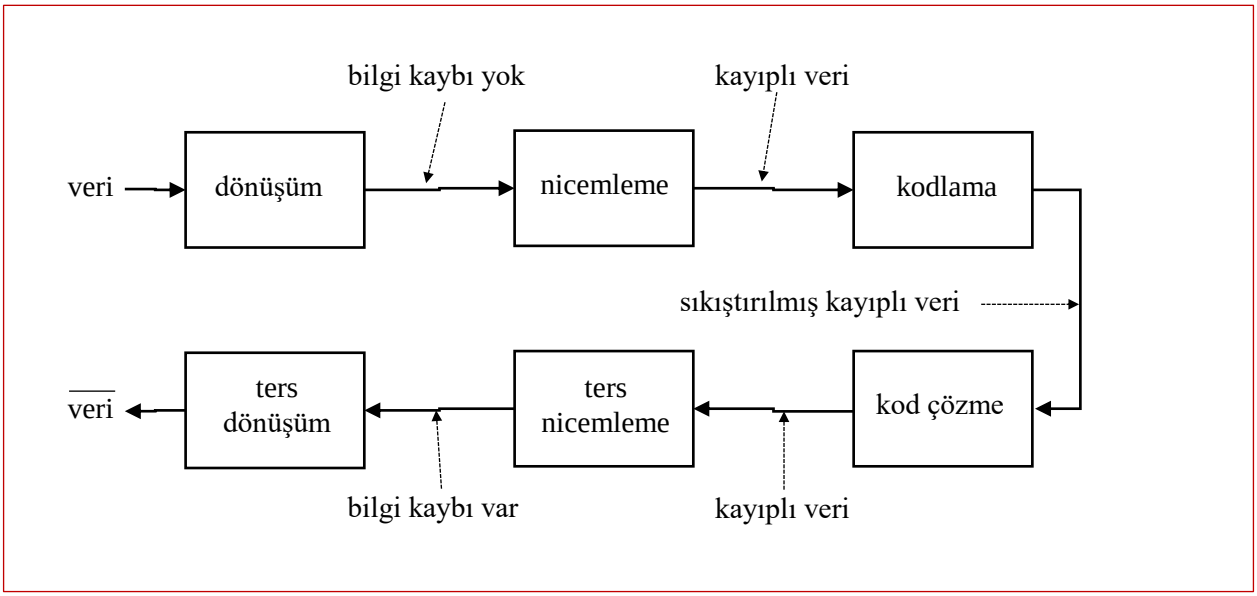
ileteceğimiz bir resmi, biraz daha bozulmayla, 100 ms'de iletebiliriz. Her resim için aynı yaklaşımı göstermek ya da her kayba razı olmak zorunda değiliz. Özetle, attığımız taş ürküttüğümüz kurbağaya değmelidir. Örneğin, oldukça yüksek harcamalar ile Mars'a gönderdiğimiz bir aracın gönderdiği resimlere kayıplı sıkıştırma uygulayıp saklamak ve disk alanından birkaç GB kazanmayı hedeflemek, mantıksızdır. Benzeri sınırlama, tıbbi görüntüleme teknikleri ile üretilen resimlerde de karşımıza çıkar. Hatta, bunun gibi insan hayatını doğrudan ilgilendiren durumlarda kayıplı sıkıştırma kullanmak yasal olmayabilir de. Ayrıca, kapasitesi sınırlı bir kanaldan gerçek zamanlı veri iletmek istediğimizde iletilecek veriyi kapasite sınırları içine indirmek durumundayız. Örneğin, bir telefon konuşmasında sesteki gecikmenin birkaç yüz milisaniyeyi geçmemesi hedeflenir ve bozulmaya razı olunur, aksi halde rahatsız edici bir iletişim ortaya çıkar.

Günümüzde, taşınmak istenen (internet üzerinden olabilir) resim, ses, video gibi verilerin neredeyse tamamı eğlence amaçlı olduğundan, kayıplı sıkıştırma yöntemlerinin yaygınlığını kaçınılmazdır. Hatta bazı durumlarda alıcıda sıkıştırılmış verinin bile kullanılmasına gerek olmayabilir. Örneğin canlı bir futbol müsabakası yayınında anlık bir resim düşünelim. Burada herkesin gözü oyunculardadır ve toptadır. Kimse yerdeki çimenleri ve arka plandaki seyircileri incelemez. Televizyon alıcısının bu kısımları sentetik verilerle doldurması kimsenin dikkatini çekmez. Yani taşınan veri orijinal resimde hesaplanabilecek entropiden oldukça küçük olabilir ve gerçeklik algısını bozmadığı sürece kimsenin umurunda olmaz. Bu yapılıyor demiyorum ama benzeri şeylerin yapıldığı bir gerçek.

Kayıplı veri sıkıştırmada tekrarlılığı azaltma amacına ek olarak amaç uygulama için önemsiz veriyi atmak ta hedeflenir. Çoğunlukla, veriden atılacak şeyleri veriye bakarak görmek/belirlemek oldukça zordur. O nedenle, veri tekrarlılığının daha kolay belirlenebileceği bir forma dönüştürülür. Dönüştürülmüş veriden önemsiz/gereksiz olduğu düşünülen kısımlar atılır ve geride kalan veri kayıpsız sıkıştırma teknikleri de dahil olmak üzere çeşitli yöntemlerle kodlanır. Şekil 1.17'de gösterildiği gibi dönüştürülmüş veri üzerinde kodlama yapıldığı için bu yaklaşıma Dönüşüm Kodlama (Transform Coding) ismi verilir.

Veri sıkıştırma işlemleri sayısal sistemlerde sınırlı temsil yeteneği olan sayı tipleriyle (integer, IEEE-754 vb) gerçekleştirildiği için işlemlerden gelen doğal bir kayıp (yuvarlama) vardır. Teoride böyle bir kaybı olmayan dönüşüm ve ters dönüşüm çiftleri sayısal olarak uygulandığında yuvarlama hataları üretilmesi kaçınılmazdır. Ancak sıkıştırma kapsamında "kayıp" kelimesi ile nicemleme ile oluşan, bilerek atılan ya da temsil değişikliğinden doğan kayıp kastedilmektedir.

Şekil 1.17'de asıl kaybın nicemleme işleminde olduğu gösterilmiştir. Çünkü nicemleme işleminin tam olarak tersi yoktur. Bunu Nicemleme ve Örnekleme bölümünde daha detaylı anlatıyoruz ama burada basit bir örnek verelim. 3.141592 sayısını noktadan sonra 2 hanesini koruyarak tamsayı ile temsil etmek istediğimizi varsayalım. Sayıyı 100 ile çarparak noktadan sonrasını atalım ve 314 elde edelim. Burada bir nicemleme yaptık. Ters Nicemleme ile gösterilen blokta 314 sayısı 100'e bölünerek 3.14 elde edilir. Ancak orijinal 3.141592 sayısı geri elde edilemez. İşte bilerek ve razı olarak yaptığımız kayıp budur. Bir noktaya daha değinmek gerekir. Yukarıdaki örnekte sadece nicemleme değil aynı zamanda nicemleme ve haritalama (mapping) yaptık. Kodlama bloğunun sadece sıkıştırma yaptığını varsayarak, bütünlüğü bozmamak için, haritalama fonksiyonu aslında bir çeşit kodlama olduğu halde nicemlemeye dahil ettik. Yani ters nicemleme fonksiyonu sadece ters haritalama yapıyor, nicemlemenin tersi yok.



Şekil 1.17. Dönüşüm kodlama ile kayıplı sıkıştırma ve çözme akışı.

Kayıplı veri sıkıştırma yöntemlerinde kullanılabilecek dönüşümlerde aranan özellikler şunlardır;

- Basitlik
- Donanım ile gerçekleştirebilme olanağı
- Hızlı algoritmaların varlığı
- Atılabilecek bilgiyi temsil eden veriyi ayrıştırmada işe yararlık.

İstenen özellikler açısından bakıldığında aşağıdaki dönüşümler öne çıkmaktadır;

- Fourier dönüşümü
- Kosinüs dönüşümü
- Dalgacık (Wavelet) dönüşümleri
- Karhunen-Loeve dönüşümü (Hotelling ya da Eigenvalue dönüşümü olarak da anılır)
- Walsh-Hadamard dönüşümü
- Slant ve Haar dönüşümleri

Bunlardan ilk 3'ü oldukça yaygın şekilde uygulanmakta ve kullanılmaktadır. Burada, doğrusal işlemlerle gerçekleştirilebilen dönüşümlere genel bir bakış atalım ve yaygın bir örneği açıklayalım.

Kaynak verimizin x_n (n=0, 1, ..., N-1) sayı dizisinden (gerçek ya da sanal) oluştuğunu varsayalım. Dönüştürülmüş verimiz ise $y_k = \sum_{n=0}^{N-1} f_{kn} x_n$ (k=0, 1, ..., K-1) ağırlıklı doğrusal toplamından oluşsun. Eğer x_n'leri y_k'lerden tekrar geri elde etmek istiyorsak $K \geq N$ olmalıdır. f_{kn} ağırlıklarına dönüşümün çekirdeği ismi verilir. Dönüşümü

$X = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}$ ve $F = \begin{bmatrix} f_{0,0} & f_{0,1} & \cdots & f_{0,N-1} \\ f_{1,0} & f_{1,1} & \cdots & f_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ f_{K-1,0} & f_{K-1,1} & \cdots & f_{K-1,N-1} \end{bmatrix}$ olmak üzere

$$Y = FX \tag{1.6}$$

matris formunda yazabiliriz. Dönüştürülmüş veriden asıl veriyi geri elde etmek için ise

$$X = GY \tag{1.7}$$

kullanılır. Eğer F tersi alınabilir ve $G = F^{-1}$ ise X tam olarak geri elde edilebilir, aksi halde G'ye yaklaşık ters dönüşüm ismi verilir. Tersinin alınabilmesi için $rank(F) = K = N$ olmalıdır. Bu durumda $rank(G) = N$ olmaktadır. Bu da F ve G'in satırlarının dik (orthogonal) olmasıyla mümkündür. Diklik konusunu İşaret, Doğrusallık, Enerji bölümünde irdeliyoruz.

Dönüşüm çekirdeği $e^{-j2\pi kn/N}$ ve ters dönüşüm çekirdeği de $e^{j2\pi kn/N}$ ise buna Kesikli Fourier Dönüşümü (DFT: Discrete Fourier Transform) denir. DFT için Fourier bölümüne bakınız. F , $f_{k,n} = e^{-j2\pi kn/N}$ katsayılarından oluşur. Sayısal resim ve sayısal hareketli görüntü gibi 2 ve 3 boyutlu kesikli veriler için 2 ve 3 boyutlu (2B ve 3B) dönüşümler kullanılabilir. Örneğin 2B Fourier dönüşümü

$$y_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{m,n} e^{-j2\pi\left(\frac{kn}{N} + \frac{lm}{M}\right)} \tag{1.8}$$

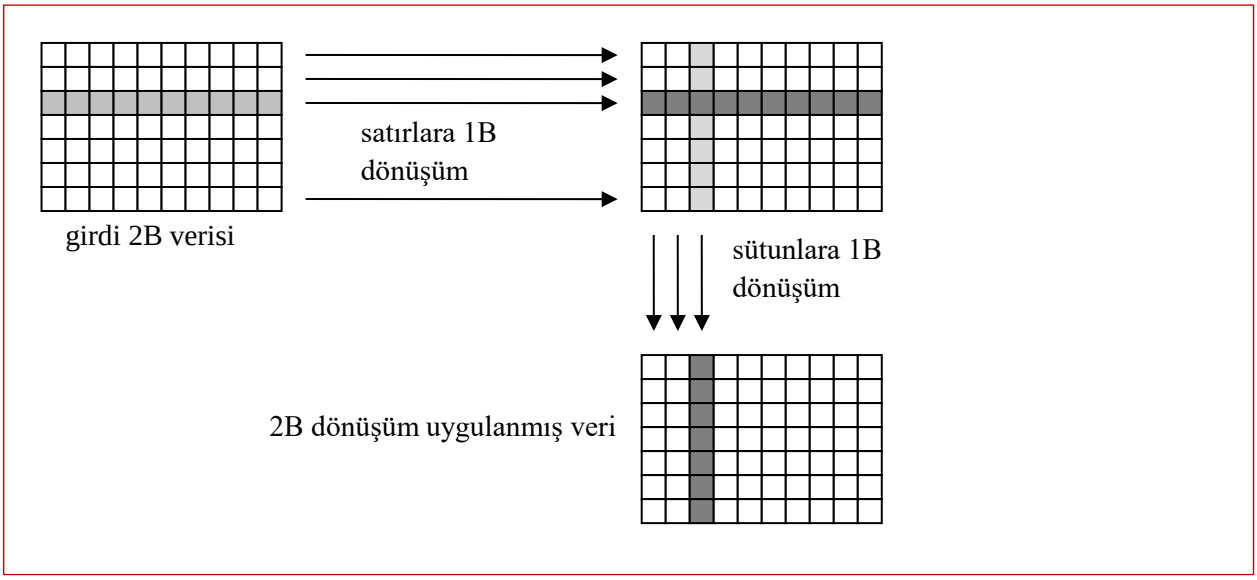
şeklinde tanımlanır. Dönüşüm çekirdeği $f_{k,n,l,m} = e^{-j2\pi\left(\frac{kn}{N}\right)} e^{-j2\pi\left(\frac{lm}{M}\right)}$ şeklinde $f_{k,n,N}$ ve $f_{l,m,M}$'ye ayrıştırılabildiğinden, dönüşüm

$$y_{k,l} = \sum_{m=0}^{M-1} \left(\sum_{n=0}^{N-1} x_{m,n} f_{k,n,N} \right) f_{l,m,M} \tag{1.9}$$

şeklinde 1 boyutlu (1B) dönüşümlerle gerçekleştirilebilir. Bu durumda dönüşümün matris formunda yazılımı, F_1 ve F_2 1B dönüşümlerin matris formundaki çekirdekleri olmak üzere,

$$Y = F_1 X F_2 \tag{1.10}$$

şeklinde yapılır. Eğer X matrisi, $N \times M$ büyüklüğünde sayısal bir resim gibi 2B bir veriyi temsil ediyorsa denklem (1.9)'un anlamı şudur; Resmin tüm satırları/sütunları üzerinde 1B dönüşüm uygulanır, elde edilen matrisin tüm sütunları/satırları üzerinde 1B dönüşüm uygulanır ve 2B dönüşüm sonucu elde edilir (Şekil 1.18). Bu şekildeki dönüşümlere ayrıştırılabilir dönüşüm ismi verilir. Ayrıştırılabilir dönüşümlerin her boyuttaki dönüşüm çekirdeği aynı ise bu dönüşümlere simetrik dönüşüm denir. Ayrıştırılabilir simetrik 2B dönüşüm özel durumunda eğer girdi verilerinde $N = M$ ise $X = GYG$ olur. Doğal olarak $K = L = N = M$ olup satır ve sütun 1B dönüşümler birbirinin aynı olur.



Şekil 1.18. 2B dönüşümün 1B dönüşümlerle gerçeklenmesi.

Kesikli Kosinus Dönüşümü'nün (DCT: Discrete Cosine Transform) ise birbirinden biraz farklı en az 5 tanımı vardır. Burada JPEG (Joint Photographic Experts Group) kayıplı sıkıştırma yönteminin temel (baseline) yaklaşımında kullanılan kosinüs dönüşümüne değinelim. Bu 2B dönüşüm

$$\alpha(s) = \begin{cases} \frac{1}{\sqrt{N}} & ,s = 0 \\ \frac{2}{\sqrt{N}} & s \neq 0 \end{cases}$$ olmak üzere

$$y_{k,l} = \sum_{m=0}^{M-1} \alpha(m) \left(\sum_{n=0}^{N-1} \alpha(n) x_{m,n} cos\big((2n+1)k\pi/2N\big) \right) cos\big((2m+1)l\pi/2M\big) \qquad (1.11)$$

şeklinde ayrıştırılabilir simetrik bir şekilde tanımlıdır. Yani 1B dönönüşümlerle gerçekleştirilebilir. Şekil 1.19, bir sayısal resme 2B kosinüs dönüşümü uygulanıp katsayıların sadece end üşük frekanslı (her iki boyutta) dörtte birini alarak ters dönüşüm uygulanmasıyla elde edilen resmi göstermektedir. Elbette resimde özellikle detaylar ve keskin kenarlarda hatalar oluşmaktadır ancak bu basit örnekle bile önemli miktarda veri sıkıştırma gerçeklenebileceği görülmektedir.

Hatırlanması gereken bir diğer nokta, DCT'nin birbirinden farklı en az 5 tanımı olduğudur. Denklem (1.11)'de verilen bunlardan sadece birisidir. Sonraki bölümde özetlediğimiz JPEG kayıplı resim sıkıştırma yönteminde

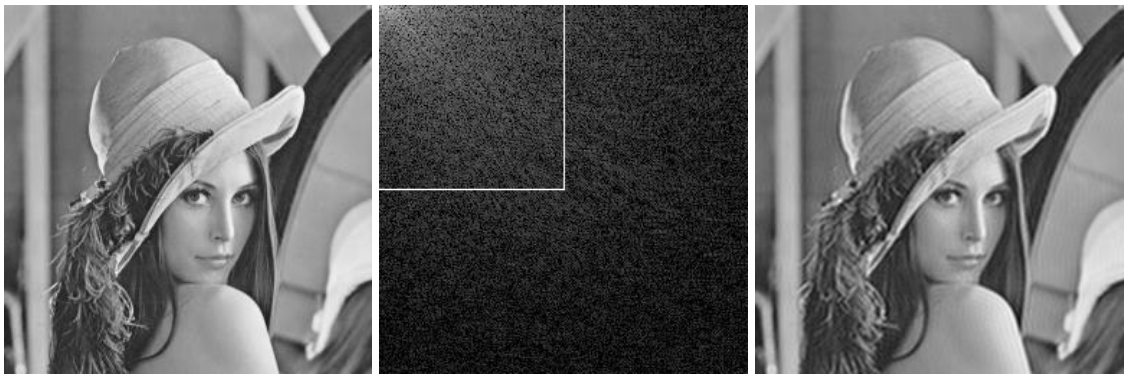
$$y_k = \sum_{n=0}^{N-1} x_n cos\left(\left(n + \frac{1}{2}\right) k\pi/N\right) \qquad (1.12)$$

DCT formülü kullanılmaktadır.

1.7.1 JPEG

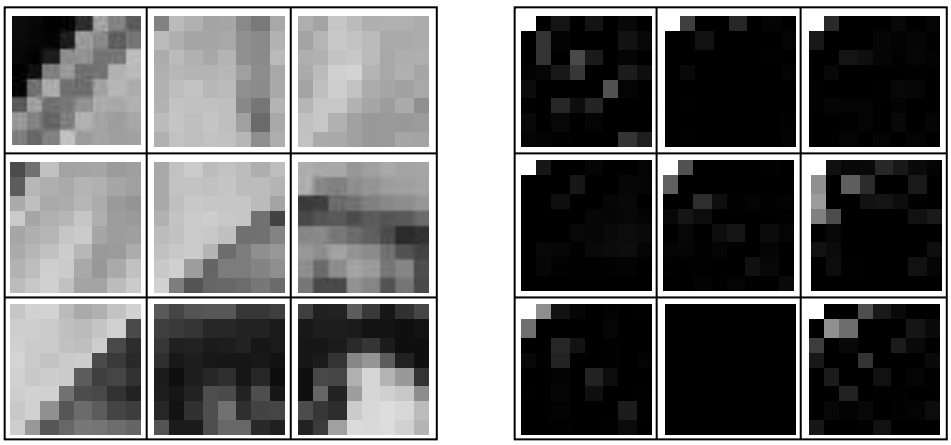
JPEG standardı da sayısal resim sıkıştırmaya benzeri şekilde yaklaşmaktadır. Tüm resimden, b pikselleri temsil etmek üzere kullanılan bit sayısı olmak üzere (örneğin 8 bit), 2^{b-1} (örneğin 128)

çıkarılır. Resim 8x8'lik alt-resimlere ayırır. Herbirine 2B kosinüs dönüşümü uygulanır. Şekil 1.20 8x8'lik bloklardan 9 adedini ve kosinüs dönüşümlerini göstermektedir. Dönüşüm bloklarının çoğunda enerjinin sol-üst köşeye (düşük frekanslar) toplandığı görülmektedir. Doğal resimlerde komşu pikseller yüksek benzerlik taşıdığından bu durum oldukça normaldir. Benzerliğin düşük olduğu durumlar kenarları (yüksek farkları) temsil ettiğinden bu yüksek frekansları temsil eden bu katsayılar atıldığında bozulmalar oluşur. JPEG, yüksek frekanslı katsayıları doğrudan atmak yerine değerleri yüksek olan katsayıları tutup diğerlerini atmak için zig-zag tarama yapmaktadır.



a) b) c)

Şekil 1.19. Kosinüs Dönüşümü ile kayıplı veri sıkıştırma örneği. a) Orijinal resim b) Kosinüs dönüşümü, ters dönüşümde kullanılan kısım işaretlenmiş c) Ters dönüşüm sonucu



a) b)

Şekil 1.20. a) 8x8 boyutlarında resim blokları b) Blokların kosinüs dönüşümleri sonucu oluşan katsayılar (katsayıları resim haline getirmek için büyüklükleri (0, 255) aralığına ölçeklenmiştir.

8x8'lik dönüşüm bloğunun elemanları öncelikle bir normalizasyon matrisinin elemanlarına bölünür. Şekil 1.21 standart bir normalizasyon matrisini göstermektedir. Dönüşüm bloğunun (64 adet gerçek sayı) herbir elemanı normalizasyon matrisinin karşıgelen elemanına bölünür ve tamsayıya yuvarlanır. Normalizasyon matrisin değerleri standardizasyon çalışmaları sırasında deneyler

sonucunda belirlenmiştir. Şekil 1.22'de örnek olarak verilen katsayılara normalizasyon uygulanarak elde edilen değerler Şekil 1.23'te verilmiştir.

16	11	10	16	24	40	51	61
13	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Şekil 1.21. JPEG standardıyla önerilen normalizasyon matrisi değerleri deneylerle hesaplanmıştır.

344,12	85,935	-27,117	-60,316	51,375	-67,483	5,4144	-0,9025
-51,894	-20,105	24,176	-8,2341	-7,1179	-3,3003	-7,7824	6,6240
-8,8309	-9,0513	-5,9634	-0,4184	-10,926	-7,4485	2,2045	-13,435
-15,807	12,277	-8,7681	-24,146	1,1885	-5,0784	-8,8987	5,5736
-3,6250	-10,469	-17,855	0,4644	-21,875	-0,5659	-0,3162	-2,1043
-4,7567	1,3709	-11,813	-3,8462	-5,8313	-7,7358	-2,0873	-6,2885
-2,7012	-7,8226	-3,7955	-17,866	-3,1863	-13,250	-5,7866	2,7547
-0,7145	1,8562	-1,4233	-7,1339	-5,2303	-5,1786	-2,0782	4,4869

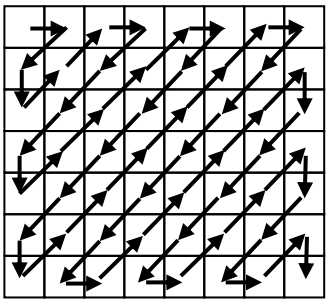
Şekil 1.22. Kosinüs dönüşümü uygulanmış 8x8 bloklardan birisinin değerleri.

21	7	-2	-3	2	-1	0	0
-3	-1	1	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Şekil 1.23. Şekil 1.22'deki değerler normalizasyon matrisinin karşı gelen elemanlarına bölünmüş ve tam sayıya yuvarlanmış.

Normalizasyon uygulanmış tamsayılardan oluşan bloğun elemanları çoğunlukla sol-üst köşeye toplandığından Şekil 1.24'te gösterilen zig-zag tarama ile tek boyutlu dizi haline getirildiğinde genel eğilim sayıların giderek küçülmesidir. Dizinin son kısımlarındaki büyük bir bölümü sıfır olduğundan kodlanmasına gerek yoktur. Bu nedenle sıfır olmayan son elemandan sonra geri kalanların sıfır olduğunu belirtmek üzere özel bir kod yerleştirilir. Normalizasyon matrisi elemanlarına bölme ve

yuvarlama işlemi bir nicemleme işlemidir. Tabi ki yuvarlanmış sayılardan orijinal değerleri geri elde etmek mümkün değildir. JPEG'deki kayıp buradan kaynaklanmaktadır. Oransal olarak kayıpların büyük çoğunluğu yüksek frekanslı katsayılarda oluşmakta, bunlar ise resimdeki keskin kenarlara ve ani değişimlere karşı gelmektedir.



Şekil 1.24. Zig-Zag tarama ile normalize edilmiş katsayıların serileştirilmesi.

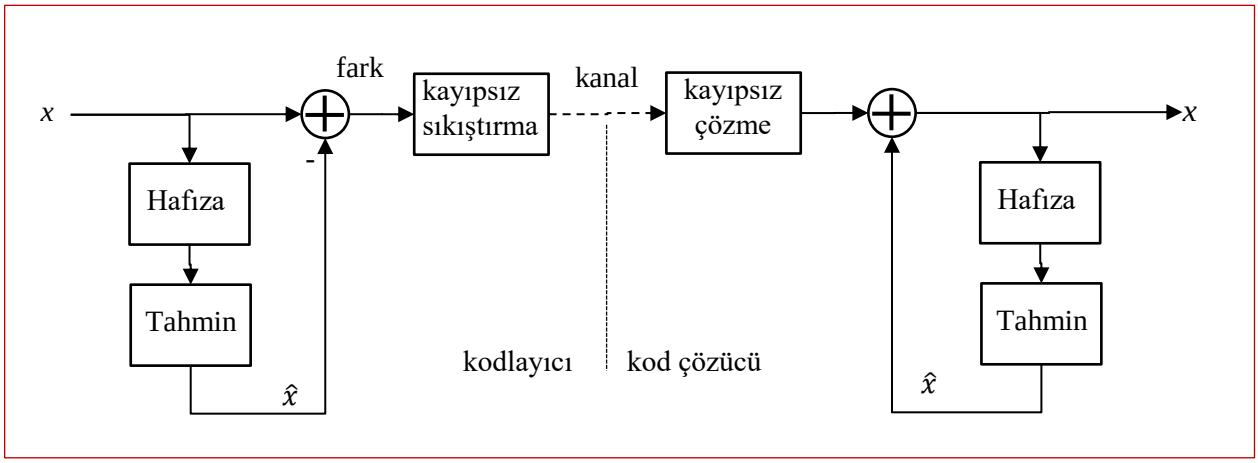
Şekil 1.23'teki normalize edilmiş bloğun zig-zag taranması ile elde edilmiş dizi {21, 7, -3, 0, -1, -2, -3, 1, 0, -1, 0, 0, 0, 0, 2, -1, BS} olup BS blok sonunu belirten özel bir koddur. Resim bu şekildeki serileştirilmiş dizilerden geri elde edileceği zaman, dönüşüm bloğu oluşturulurken 8x8'lik blok sol-üst kısmından itibaren dizi elemanları ile doldurulur ve BS'den sonrası sıfır olarak bırakılır. Tüm bloklar üzerinde uygulanan kosinüs dönüşümü, normalizasyon ve zig-zag tarama ile elde edilen dizilerden oluşan veri akışı kayıpsız kodlama yöntemleri ile sıkıştırılır ve JPEG dosya formatında kaydedilir.

JPEG ile kayıplı sıkıştırma işlemi 8x8'lik bloklar ile gerçekleştirildiğinden tekrar geri açma sonrasında bloklar komşuluğunda bir uyumsuzluk oluşmakta, dikkatli incelendiğinde blok etkisi gözlenebilmektedir. Bloklar ile işlem yapıldığında bu etki neredeyse kaçınılmazdır ve etkiyi azaltmak için ilave işlemler (süzgeçten geçirme vb) yapmak gerekir.

Buraya kadar anlatılanlar 1992'de hazırlanan JPEG standardını referans almaktadır. Daha sonra geliştirilen JPEG-2000 de temelde aynı yaklaşımı gözetmekle beraber, daha yüksek sıkıştırma, daha iyi çözünürlük ve dinamik aralık, aşamalı kodlama (progressive coding), seçenekli kayıpsız sıkıştırma ve esnek dosya formatını da dahil ederek daha esnek bir yapı sunmaktadır.

1.7.2 Farksal Öngörü Kodlama

Farksal öngörü kodlama (differential predictive coding) yada sadece öngörü kodlama doğrudan veri sıkıştırmayı hedeflemiyor, ama yaptığı iş itibarıyla veri sıkıştırma yöntemleri içinde bulunmayı hak ediyor. Yöntemin hedefi, önceki verilerden olası yeni veriyi tahmin etmek ve gerçekleşen ile tahmin edilen arasındaki farkın küçük olmasını bekleyerek bu farkı kodlamak. Şekil 1.25 bu yaklaşımı özetlemektedir.



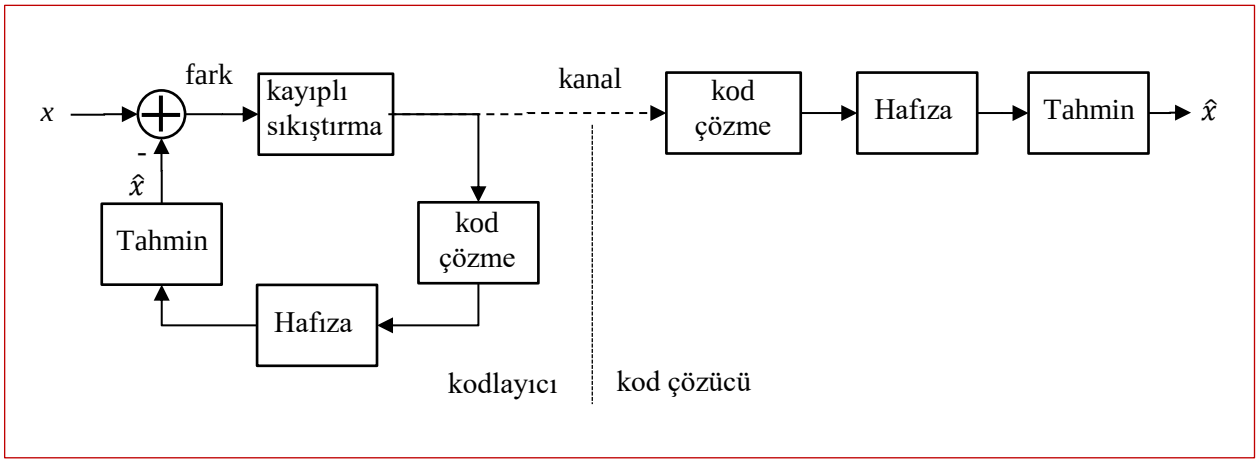
Şekil 1.25. Kayıpsız farksal öngörü kodlayıcı.

Tahmin/öngörü sonucu üretilen veri ile gerçekleşen veri arasındaki fark kodlandığı ve kod çözme aşamasında da bu işlemin tersi kayıpsız olarak gerçekleştirilebildiği için, yöntem kayıpsızdır. Fark verisinin ardışıl örnekleri arasındaki istatistiksel bağımlılık (korelasyon) oldukça azaltılmış olduğu gibi girdi verisinin karakteristiğine bağlı olarak daha az bitle temsil edilebilir. Yani, kayıpsız sıkıştırma bloğuna giren verideki sembol sayısı daha azdır. Bu durum, özellikle ardışıl x değerleri arasındaki benzerliğin yüksek olduğu uygulamalarda önemli bir veri sıkıştırma sağlar. Örneğin, aşırı örneklenmiş bir işaretin ardışıl örnekleri için oldukça kullanışlı olabilir. O nedenle bu yaklaşıma Örnekleme bölümünde biraz daha değindik.

En küçük hafıza miktarı sadece önceki 1 sembolü saklayabilendir. Böylece fark, ardışıl iki sembolün pozitif ya da negatif yöndeki farkı olmaktadır. Eğer bu fark sadece 1 bit ile temsil edilirse (örneğin 0 farkın pozitif ve 1 negatif yönde olduğunu söylerse), yani farkın tam olarak temsil edilememe ihtimali varsa, yöntem kayıplı olur ve orijinal x işareti geri elde edilemez. Yöntemin ismi de farksal darbe kod modülasyonu (DPCM: Differential Pulse Code Modulation) olur. Tabi ki x girdisi 1 bit ise (ikili seri akış) kayıp oluşmaz, DPCM'in kayıpsız özel durumu oluşur. Genel olarak, Şekil 1.25'deki kayıpsız sıkıştırma bloğunu kayıplı sıkıştırma ile değiştirir isek kayıplı farksal öngörü kodlayıcı elde ederiz. Bu aslında, tüm olası farkların temsil edilememesi ile eşdeğerdir.

Farkın küçük olabilmesi için, yani daha doğru bir tahmin yapabilmek için, sadece önceki 1 örnek yerine önceki n örnek kullanılabilir. Kayıplı kodlayıcı kullanılması durumunda önceki örnek (ya da örnekler) ile tahmin etmek yerine kodlanmış ve kod çözülmüş örnekler ile tahmin yapılması daha mantıklıdır (Şekil 1.26). Böylelikle alıcıda kod çözme sırasında hata birikmesinin önüne geçilmiş olunur (hata yine olur, ama birikmez).

Giriş bölümünde, henüz hazır olmadığımız bir durumda, bu kadar detaya neden girdik? Çünkü sıradaki bölümde anlatılan MPEG farksal kodlamayı kullanıyor.



Şekil 1.26. Kayıplı farksal öngörü kodlayıcı.

1.7.3 MPEG

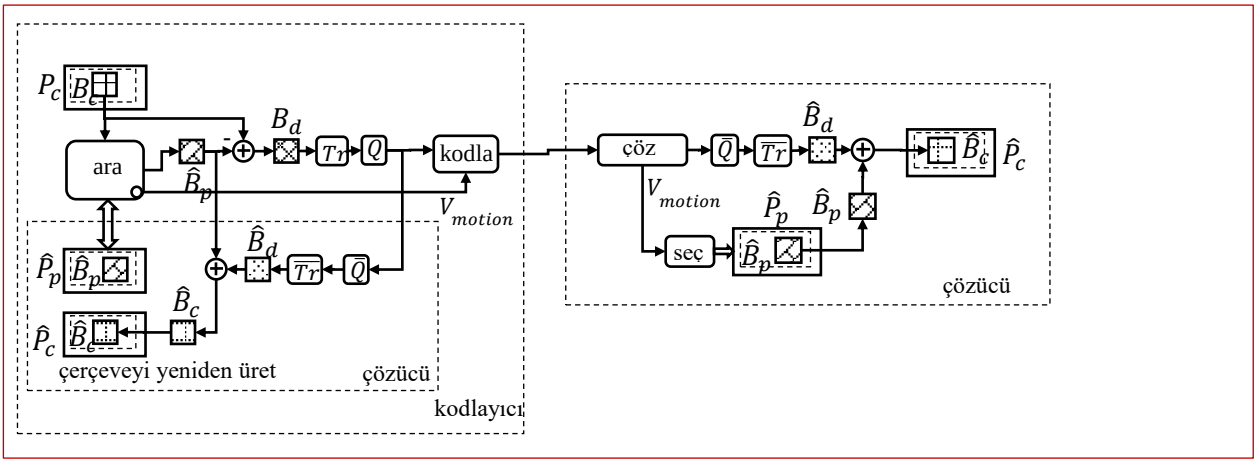
Kayıplı sıkıştırma haberleşme kitaplarında yeterince yer alamayacak kadar geniş bir konudur. Yine de, aynı JPEG'de yaptığımız gibi, kayıplı video sıkıştırma içeren MPEG (Motion Picture Experts Group) konusuna, öğrenciye bir fikir verme açısından, anahatlarıyla değinmek gerekir. Ne var ki, konu hakkında bir şekilde uzmanlaşmak isteyenlerin sayfalar dolusu standartları okuyup özümsemesi, nedenleri ve tarihçesi hakkında fikir sahibi olması kaçınılmazdır. Çünkü, artık sıkıştırmanın genel olarak veri içindeki tekrarlar/benzeşimler/korelasyondan faydalanmak olduğunu anlamamız yanında, uyumluluk ve standardizasyon konularının da önemli bir yer tuttuğunu öğrendik.

Video akışının çerçeve (frame) adı verilen ardışıl resimlerin insan gözünü/beynini yanıltıcı ve sanki hareketli bir sahneymiş etkisi yaratıcı bir hızla gösterilmesi olduğunu söyleyebiliriz. Öyleyse ardışıl gösterilen resimlerin ayrı ayrı JPEG usulü sıkıştırılması ve gönderilmesi de bir video akışıdır. Ancak burada, biraz geride kalmış bu yöntem yerine, henüz yaygınlaşan ve bahsedilen yaklaşıma göre önemli üstünlükleri olan nispeten yeni bir standardın anahatlarına değinmek daha anlamlıdır. Bu yaklaşımın adı Yüksek Verimli Video Kodlamadır (HEVC : High Efficiency Video Coding).

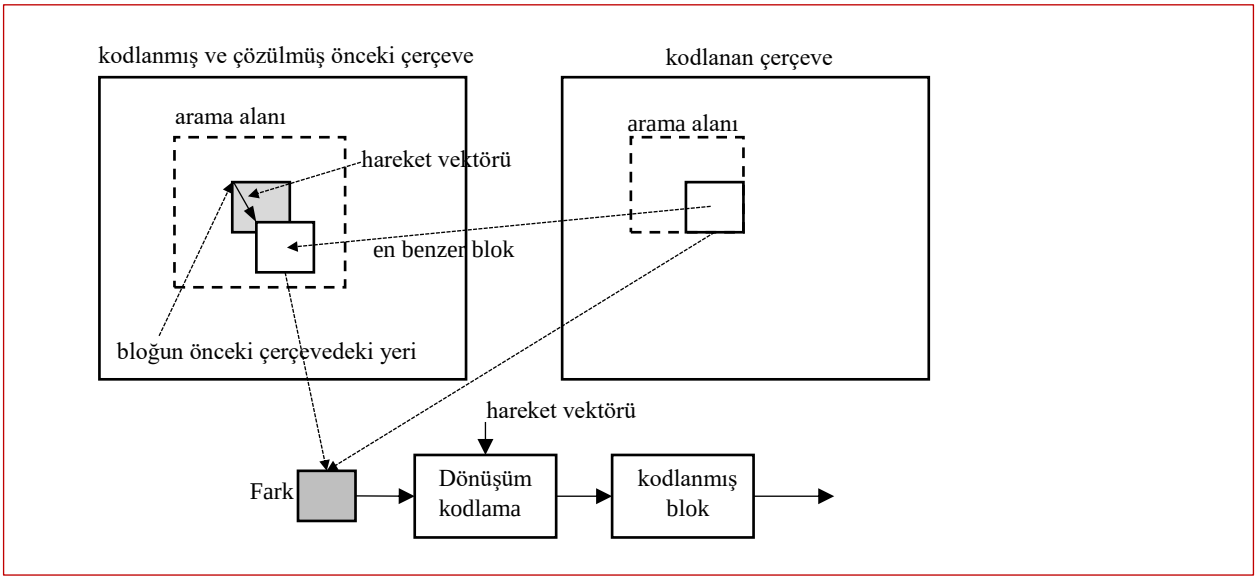
HEVC, JPEG'deki gibi blok içindeki korelasyondan faydalanmanın yanında, komşu bloklar, hatta önceki çerçevelerdeki bloklar ile olan benzerliklerden de faydalanır. Yani sadece uzamsal (spatial) benzerlikler iyi bir sıkıştırma için yeterli olmamış, zamansal (temporal) benzerlikler de kullanılmaya çalışılmıştır. Aslında her iki benzerliğe de yabancı değiliz. HEVC bunu yaparken şu anki teknolojinin işlem hızını da göz önüne almak zorundadır, çünkü yaklaşım oldukça işlem yoğundur. Ayrıca, en basit anlamda önceki çerçevelerle olan benzerliklere bakılacaksa, önceki çerçevelerin de hafızada tutulması gerekmektedir. Yani sadece işlem yoğun değil, aynı zamanda hafıza kullanımı da yoğundur.

HEVC'i şu ana kadar gördüğümüz sıkıştırma yöntemlerinin yüksek veri miktarları üzerinde bir bileşimi olarak düşünebiliriz. Bunu iyice basitleştirilmiş olan Şekil 1.27'den görebiliriz.

JPEG gibi, bloklar üzerinde işlemler, HEVC için de geçerli olmaya devam etmektedir. Ancak bloklar sadece 8x8 değil, 16x16, 32x32, 16x32 vb olabilmektedir. Büyük bloklar dönüşüm kodlamanın verimini arttırmakta, ancak daha yüksek işlem gücü gerektirmektedir. En çok işlem gücü gerektiren işlemlerden birisi de hareket bulma (motion vector prediction) işlemidir. Özet olarak bu işlem halihazırda kodlanacak bloğun daha önce kodlanıp gönderilen ve halihazırdaki çerçevenin kodlanmış/gönderilmiş kısmında en benzerini bulma işlemidir ve Şekil 1.27'te *ara* bloğu ile gösterilmiştir. İşlemin detayı Şekil 1.28'da anlatılmıştır.



Şekil 1.27. HEVC yaklaşımı.



Şekil 1.28. HEVC için hareket bulma.

MPEG ile kayıplı farksal öngörü kodlayıcı arasındaki benzerlik dikkatimizi çekiyor. Aradaki fark, MPEG kodlayıcısının kodlanmış ve kodu çözülmüş önceki n bloğun tümünü kullanarak bir tahmin yapmak yerine bunlardan sadece en benzeri ile tahmin yapmasıdır. Böylece tahmin için gerekli işlem yükü en benzeri arama işlem yüküne dönüşür.

MPEG-HEVC ile ilgili sayfalar dolusu dökümanı burada anlatmamız/açıklamamız anlamlı değildir. Burada, haberleşme sistemlerinde karşılaşılan iletilecek verilerin önemli bir çoğunluğunun bu gibi veriler olduğu için, sadece bir fikir verme amacıyla anahatlarına değindik. Veri sıkıştırma konusu ayrıca birçok kaynak kitabı kapsayacak bir konudur.

Giriş bölümünü kapatırken de aynı şekilde haberleşme sistemlerinde iletimi yapılan verilerin tümüne ve elde edilişlerine değinmemizin mümkün ve anlamlı olmadığına dikkat çekelim. Bütünlüğü bozmamak adına burada oldukça kısa şekilde anlatılan yöntemleri öğrenmek isteyenlerin o yöntemlere adanmış ilgili kaynaklara başvurması esastır.