T.C.

TRAKYA ÜNİVERSİTESİ

FEN BİLİMLERİ ENSTİTÜSÜ

SIKIŞTIRILMIŞ METİN VERİLER İÇİNDE DİZGİ EŞLEME ALGORİTMALARININ KULLANILMASININ İNCELENMESİ VE YENİ BİR YAKLAŞIMIN GELİŞTİRİLMESİ

Halil Nusret BULUŞ

Doktora Tezi

Danışman: Yrd. Doç. Dr. Aydın CARUS

Edirne-2010

ÖZET

Bu tezin amacı, sıkıştırılmış dizgi eşleme yapmaya uygun yeni bir sıkıştırma algoritmasının ve sıkıştırılmış dizgi eşleme yaklaşımının geliştirilmesidir. Ayrıca mevcut sıkıştırma algoritmaları ile sıkıştırılmış veriler üzerinde dizgi eşleme yapan çalışmaların araştırılarak, sıkıştırılmış dizgi eşleme yöntem ve yaklaşımlarının kullandıkları sıkıştırma algoritmalarına göre sınıflandırılması da yapılmıştır.

Tezin birinci bölümünde, veri sıkıştırma, dizgi eşleme ve sıkıştırılmış dizgi eşleme konularından kısaca söz edilip, konular ile ilgili diğer genel terimlere açıklık getirilmiştir. Tezin ikinci bölümünde daha önceden üzerinde sıkıştırılmış dizgi eşleme yaklaşımı geliştirme çalışmaları yapılmış olan sıkıştırma algoritmaları ele alınmış ve bu algoritmaların örnekler ile açıklaması yapılmıştır. Tezin üçüncü bölümünde mevcut yapılmış olan sıkıştırılmış dizgi eşleme çalışmalarında kullanılmış olan dizgi eşleme yöntemleri ve yaklaşımları örneklerle açıklanmıştır. Tezin dördüncü bölümünde literatürde yer alan sıkıştırılmış dizgi eşleme çalışmaları ayrıntılı bir şekilde incelenmiştir. Tezin beşinci bölümünde, yeni geliştirilen sıkıştırılmış dizgi eşlemeye elverişli veri sıkıştırma algoritması ve bu algoritma için sıkıştırılmış dizgi eşleme yaklaşımı ayrıntılı olarak ele alınmıştır. Altıncı bölümde geliştirilmiş olan sıkıştırma algoritması ile mevcut diğer ilgili sıkıştırma algoritmalarının sıkıştırma oranları, sıkıştırma ve açma süreleri kıyaslanmıştır. Ayrıca mevcut bilinen külliyatlardan da yararlanılarak İngilizce ve Türkçe külliyatlar oluşturulmuştur. Bu külliyatlar üzerinde, geliştirilen sıkıştırılmış dizgi eşleme yaklaşımı ile diğer mevcut sıkıştırılmış dizgi eşleme yaklaşımlarının, dizgi eşleme süreleri bakımından kıyaslamaları yapılmıştır.

Bu tez 2010 yılında yapılmıştır ve 161 sayfadan oluşmaktadır.

Anahtar Kelimeler: Veri sıkıştırma, Dizgi eşleme, Sıkıştırılmış dizgi eşleme, Huffman Kodlama, Sözlük tabanlı kodlama, LZW

ABSTRACT

The purpose of this thesis is developing a new compression algorithm appropriate for compressed pattern matching and developing a new approach to compressed pattern matching. In addition to this, explanations of current compression algorithms and compressed pattern matching studies, the classification of compressed pattern matching methods and approaches according to the used compression algorithms take place in this thesis.

In the first chapter of this thesis, data compression, pattern matching and compressed pattern matching subjects are briefly explained and the definitions of the other terms related to these subjects are given. In the second chapter, compression algorithms which are studied previously in compressed pattern matching area are given and the explanation of these algorithms with examples are made. In the third chapter, pattern matching methods and approaches that are used in current compressed pattern matching studies are explained with some examples. In the fourth chapter, the compressed pattern matching studies in the literature are explained in details. In the fifth chapter, the new developed compression algorithm appropriate for compressed pattern matching and the compressed pattern matching approach for this compression algorithm are explained in details. In the sixth chapter, the compression ratios, compression and decompression times of the developed algorithm and current related algorithms are compared. In addition to this English and Turkish corpuses are made from well known corpuses. On these new corpuses, the developed compressed pattern matching approach and the current compressed pattern matching approaches are compared in compressed pattern matching times.

This thesis which consists of 161 pages was completed in 2010.

Key Words: Data Compression, Pattern Matching, Compressed Pattern Matching, Huffman Coding, Dictionary Based Compression, LZW

TEŞEKKÜR

Çalıştığımız alanda bilgi ve tecrübelerini benden esirgemeyen, her konuda her zaman yardımcı olan, ileriki çalışmalar için beni teşvik eden ve zor zamanlarımda sabrını esirgemeyen danışman hocam Sayın Yrd. Doç. Dr. Aydın CARUS'a,

Yaptıkları olumlu eleştiriler ve paylaştıkları değerli fikirler ile bu tezin yapılmasında katkıları olan Sayın Prof. Dr. Mesut RAZBONYALI, Sayın Yrd. Doç. Dr. Nurşen SUÇSUZ'a ve Sayın Yrd. Doç Dr. Altan MESUT'a,

Bu yoğun zamanlarda sabırlı yaklaşımları ve anlayışları için Sayın Bölüm Başkanım Yrd. Doç. Dr. Ercan BULUŞ'a ve değerli çalışma arkadaşlarıma,

Bugünlere gelmemdeki en büyük payın sahibi olan sevgili anneme, babama, ağabeyime, anneannem ve geçen ay kaybettiğim sevgili dedeme,

En zor zamanlarımda hep yanımda olan, sonsuz sabrı ve anlayışı ile bana desteğin en büyüğünü veren sevgili eşim Sinem BULUŞ'a teşekkürlerimi sunarım.

İÇİNDEKİLER

ÖZET ..... **i**
ABSTRACT ..... **ii**
TEŞEKKÜR ..... **iii**
İÇİNDEKİLER ..... **iv**
ŞEKİLLER LİSTESİ ..... **vi**
ÇİZELGELER LİSTESİ ..... **viii**
EKLER LİSTESİ ..... **x**
1. GİRİŞ **1**
2. VERİ SIKIŞTIRMA ALGORİTMALARI **7**
2.1. Olasılık Tabanlı Teknikler 7
2.1.1. Huffman kodlama 8
2.1.1.1. Huffman yaklaşımı 9
2.1.1.2. Shannon – Fano yaklaşımı 12
2.1.2. Kelime tabanlı Huffman kodları 16
2.1.2.1. Düz Huffman kodlama 17
2.1.2.2. Etiketlenmiş huffman kodu 18
2.1.3. Sonu etiketlenmiş yoğun kodlar 21
2.1.4. (s,c) – yoğun kodlar 25
2.1.5. Sınırlı önek bayt kodlama 28
2.2. Sözlük Tabanlı Teknikler 30
2.2.1. Statik sözlük yaklaşımı 31
2.2.2. Yarı-statik sözlük yaklaşımı 31
2.2.2.1. Bayt çiftleri (ikilileri) kodlama 32
2.2.3. Dinamik sıkıştırma yaklaşımı 34
2.2.3.1. LZ77 sıkıştırma algoritması 34
2.2.3.2. LZ78 sıkıştırma algoritması 39
2.2.3.3. LZW sıkıştırma algoritması 43
2.2.3.4. LZSS sıkıştırma algoritması 47
2.2.3.5. Geçiş uzunluğu kodlama sıkıştırma algoritması 51
2.3. Dönüşüm Teknikleri 52
2.3.1. BWT dönüşüm tekniği 53
3. DİZGİ EŞLEME ALGORİTMALARI **57**
3.1. Karp & Rabin Dizgi Eşleme Algoritması 57
3.2. Knuth – Morris – Pratt Dizgi Eşleme Algoritması 59

3.3. Boyer – Moore Dizgi Eşleme Algoritması 61
3.4. Shift – Or Dizgi Eşleme Algoritması 63
3.5. Horspool Dizgi Eşleme Algoritması 65
3.6. Bacward – Oracle Dizgi Eşleme Algoritması 65
4. SIKIŞTIRILMIŞ DİZGİ EŞLEME 68
4.1. Üzerinde Sıkıştırılmış Dizgi Eşleme Yapılan Algoritmalar 68
4.1.1. Genel teknikler üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları 68
4.1.2. Geçiş uzunluğu kodlama üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları 74
4.1.3. Bayt çiftleri kodlama üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları..... 76
4.1.4. Huffman kodlama üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları 81
4.1.5. Kelime tabanlı bayt kodlamalar üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları91
4.1.6. LZ ailesi üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları 94
4.1.6.1. LZ77 sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları... 94
4.1.6.2. LZ78 sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları... 96
4.1.6.3. LZW sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları..102
4.1.6.4. LZSS sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları .116
4.2. Üzerinde Sıkıştırılmış Dizgi Eşleme Yapılan Dönüşüm Teknikleri 118
4.2.1. BWT dönüşüm tekniği üzerinde yapılmış olan sıkıştırılmış dizgi eşleme çalışmaları..118
5. GELİŞTİRİLEN SIKIŞTIRILMIŞ DİZGİ EŞLEMEYE UYGUN SIKIŞTIRMA ALGORİTMASI.....121
5.1. Algoritma Detayları..... 121
5.2. Sıkıştırılmış Metni Açma Algoritması 136
5.3. Geliştirilen Algoritma İle Sıkıştırılmış Metinler Üzerinde Sıkıştırılmış Dizgi Eşleme143
6. GELİŞTİRİLEN ALGORİTMANIN DİĞER ALGORİTMALAR İLE KIYASLANMASI146
7. SONUÇLAR.....152
KAYNAKLAR.....154
EKLER 160
ÖZGEÇMİŞ.....162

ŞEKİLLER LİSTESİ

Şekil 2.1 Huffman Ağacı oluşturma adımları 10
Şekil 2.2 Huffman Ağacı 11
Şekil 2.3 Shannon – Fano ağacı 13
Şekil 2.4 Shannon – Fano ağacı oluşturma adımları 14
Şekil 2.5 Bayt kodlamalı Huffman ağacı 17
Şekil 2.6 Kelime tabanlı Huffman ağacı örneği 18
Şekil 2.7 Normal dağılım için PHC ve THC ağaçları 20
Şekil 2.8 Üssel dağılım için PHC ve THC ağaçları 21
Şekil 2.9 Örnek RPBC kodlama 28
Şekil 2.10 BPE'de sıkıştırma işlemi 33
Şekil 2.11 Dairesel kuyruk yapısı 39
Şekil 2.12 LZ78 sözlük ağacı 42
Şekil 2.13 Düz BWT dönüşümü 53
Şekil 2.14 BWT dönüşümü içeren bir sıkıştırma işleminin tüm aşamaları 56
Şekil 3.1 h2'nin h1'den kolay bir şekilde hesaplanması 58
Şekil 3.2 s<j durumu 62
Şekil 3.3 D1 ve D kaydırma kullanan BM 63
Şekil 3.4 $Ri0$ vektörü 63
Şekil 4.1 CS hiyerarşisi 70
Şekil 4.2 CS ve SLP'ler 72
Şekil 4.3 Örnek 4.2'deki evrim ağacı 72
Şekil 4.4 Boşluksuz kelime modeline göre Huffman Kodlama 82
Şekil 4.5 Yanlış eşleşme durumu 83
Şekil 4.6 "ro* rose is" dizgisi için genel arama şeması 88
Şekil 4.7 Arama otomatı 88
Şekil 4.8 Yanlış eşleşme örneği 90
Şekil 4.9 Bir katarın LZ faktorizasyonu 96
Şekil 4.10 Uzun ve kısa bloklar için önek (P) ve sonek (S) durumları 97
Şekil 4.11 LZ78 sıkıştırma algoritmasındaki blok yaklaşımı 100
Şekil 4.12 Sözlük ağacı 110
Şekil 4.13 Aktif blok olmayan durum 111

Şekil 4.14 Aktif blok olan durum 111
Şekil 4.15 Orijinal LZW ve 2 geçişli LZW örneği 113
Şekil 4.16 P=ababc'nin açılmış ST_p ağacı 115
Şekil 5.1 Geliştirilen sıkıştırma algoritması için akış şeması 136
Şekil 5.2 Geliştirilen açma algoritması 142
Şekil 5.3 Geliştirilen algoritma için sıkıştırılmış dizgi eşleme akış şeması 145

ÇİZELGELER LİSTESİ

Çizelge 2.1 Normal dağılım için Huffman tabloları 19
Çizelge 2.2 Üssel dağılım için kod tablosu 20
Çizelge 2.3 THC ve ETDC için kod kelimeleri yapısı 22
Çizelge 2.4 ETDC'de kod ataması 23
Çizelge 2.5 Normal dağılım için kod tablosu 24
Çizelge 2.6 Üssel dağılım için kod tablosu 24
Çizelge 2.7 (s,c)-yoğun kodda kod ataması 27
Çizelge 2.8 Örnek metin için üretilen LZ78 çıktıları 41
Çizelge 2.9 Girdi metnine göre LZW sıkıştırma algoritması kod çıktıları 44
Çizelge 2.10 Sıkıştırılmış girdi metninin açılması sonrası elde edilen sözlük 45
Çizelge 2.11 5 karakterlik katarlar 48
Çizelge 2.12 L=3 içim m, p, q değerleri ve önek – sonek tablosu 50
Çizelge 2.13 BWT dönüşüm tablosu 55
Çizelge 3.1 P dizgisi için π( ) değerleri 61
Çizelge 3.2 Örnek metin için Shift – Or algoritması tablosu 64
Çizelge 3.3 Boyer – Moore ve Horspool algoritmaları için kaydırma örneği 65
Çizelge 4.1 BPE'de geliştirilen algoritma için sıkıştırma oranları 79
Çizelge 4.2 Bit Huffman, Bayt Huffman, Compress ve Gzip tarafından gerçekleştirilen sıkıştırma başarım değerleri 82
Çizelge 4.3 Huffman, Bayt Huffman, 7 – bit Huffman, Compress ve Gzip için sıkıştırma ve açma zamanı 85
Çizelge 4.4 wsj metin dosyası için %99 doğrulukla arama zamanları 89
Çizelge 4.5 4-uzunluklu kodlamaya göre sembollerin kod karşılıkları 92
Çizelge 4.6 Üç metot arasında ortalama sıkıştırma oranları 99
Çizelge 4.7 Değişik algoritmalar için sıkıştırılmış dizgi eşleme zamanları 102
Çizelge 4.8 Arama zamanı (CPU zamanı) 105
Çizelge 4.9 Örnek 4.11 için önek bit haritaları 114
Çizelge 4.10 Sonek bit haritaları 115
Çizelge 4.11 Örnek 4.12 için önek bit haritaları 115
Çizelge 4.12 Belirgin nodların bit hariları 115
Çizelge 4.13 Sıkıştırılmış eşleme esnasında tampon durumu 118

Çizelge 5.1 LZW'de "Trakya_" kelimesi kodlanışı 128
Çizelge 5.2 Geliştirilen algoritmada "Trakya_" katarının ürettiği çıktılar 129
Çizelge 5.3 İlk kelime için örnek metinden sözlük oluşturma 130
Çizelge 5.4 Oluşturulan sözlük 131
Çizelge 5.5 Haritalama dizisi 132
Çizelge 5.6 2. Geçiş kodlama aşaması 132
Çizelge 5.7 Haritalama tablosu 139
Çizelge 6.1 Eng.txt dosyasını oluşturan dosyaların adı, büyüklükleri ve alındıkları yerler 147
Çizelge 6.2 Harf sayılarına göre gruplandırılmış dizgilerin ortalama sıkıştırılmış eşleme süreleri 147
Çizelge 6.3 Harf Sayılarına Göre Gruplandırılmış Dizgilerin Ortalama Sıkıştırılmış Eşleşme Zamanları 148
Çizelge 6.4 Sıkıştırma zamanı 148
Çizelge 6.5 Sıkıştırma zamanları 149
Çizelge 6.6 Sıkıştırılmış dosyaların açma zamanları 149
Çizelge 6.7 Sıkıştırılmış dosyaları açma zamanları 150
Çizelge 6.8 TWBCA algoritmasının sıkıştırma oranları 150
Çizelge 6.9 TWBCA algoritmasının sıkıştırma oranları 151

EKLER LİSTESİ

EK-A Kısaltmalar Listesi……160

1. GİRİŞ

Veri sıkıştırma konusunun bilgisayar bilimlerindeki tanımı veriyi daha öz ve daha kısa ifade etme çabasıdır. Son yıllardaki sayısal çoklu ortam evrimi, veri sıkıştırma konusunu oldukça önemli hale getirmiştir.

Veri sıkıştırma, veriyi daha hızlı iletmek veya daha az yer gerekecek şekilde saklamak için sıklıkla kullanılmaktadır. Ayrıca veri sıkıştırma algoritmaları birçok alanda kullanılmak üzere geliştirilmiş yazılımların içinde gömülmüş olarak yer almaktadır.

Veri sıkıştırma birçok uygulama için ortak bir ihtiyaç olduğu gibi, bilgisayar bilimlerinde de önemli ve aktif bir araştırma ve çalışma alanıdır. Veri sıkıştırma algoritma ve teknikleri olmadan, elektronik kütüphanecilik, video iletişim yöntemleri, sayısal televizyon yayıncılığı, mobil iletişim ve internet gibi daha bir çok pratikte önemli kullanıma sahip alanlardaki gelişmeler bu boyuta ulaşamazdı.

Veri sıkıştırma tekniklerinin ilgili olduğu ve kullanıldığı örnek alanlar:

- Faks, sesli posta ve telefon gibi kişisel iletişim alanları,
- Bellek yapıları, diskler ve teyp yapıları gibi bilgisayar sistemleri,
- Dağıtık bilgisayar sistemleri,
- Çoklu ortam gelişimi, görüntüleme ve sinyal işleme,
- Görüntü arşivleme ve video konferans,
- Sayısal televizyon ve uydu televizyonu.

Veri sıkıştırmayı zorunlu kılan problemler veri sıkıştırma üzerine birçok araştırmaya taban oluşturmaktadır. Uygulama alanının genişliğine kısmen bağlı olarak veri sıkıştırma birçok farklı disiplinin içinde kullanılmaktadır. Bu disiplinler hakkında yazılmış kitaplar bulunmaktadır. Veri sıkıştırma ile ilgili konular ve disiplinler:

- Bilgi teorisi,
- Kodlama teorisi,
- Bilgisayar ağları ve telekomünikasyon,

- Sayısal sinyal işleme,
- Görüntü işleme,
- Çoklu ortam,
- Steganografi,
- Bilgisayar güvenliği.

Veri sıkıştırma konusunda geçen *veri* terimi bilgisayar programlarının işlediği sayısal tipteki bilgi, *sıkıştırma* terimi de fazlalık olarak düşünülen veri kesiminin daha kısa kodlarla ifade edilmesi anlamına gelmektedir. *Veri sıkıştırma* terimi daha özel olarak verideki fazlalıklardan faydalanarak, bu fazlalıkları kodlamada en aza indirmek ya da başka bir deyişle veriyi kısa bir şekilde ifade etmek anlamında kullanılmaktadır [Pu, 2006].

Metin veriler bilgisayar sistemlerinde kullanılan veriler içerisinde önemli bir yer tutmaktadır. Kelime işlemci ve derleyiciler gibi birçok bilgisayar uygulaması, temel içerikleri metin karakteri olan verilerle işlem yapmaktadır. Bilgisayarlar sadece 1 ve 0 saklayıp işleyebilmektedir. Dolayısıyla her metin karakteri için bir ikili kod atanmıştır.

Bugünkü birçok bilgisayar ASCII kod kullanmaktadır. ASCII kod her karakterin 1 bayt ve her bir baytın 8 bit ile ifade edildiği sabit uzunluklu bir koddur. Sabit uzunluklu kod kullanmak verimlidir çünkü uygulama yazılımları için metin karakterlerini ele almayı kolaylaştırmaktadır.

Herhangi bir alfabe kullanılan rastgele ve doğal dille yazılmış bir metin içerisinde alfabedeki her karakterin yaklaşık olarak benzer frekansta geçmesi gerektiği düşünülmektedir. Aslında pratikte kullanılan veriler içindeki karakterler çok nadiren normal dağılım özelliği göstermektedir. Anlamlı metinlerden oluşan ve tipik bir doğal dil içeren metnin bazı karakterleri diğerlerinden daha fazla sayıda içerdiği basit bir karakter frekans analizi ile anlaşılabilir. Örneğin, İngilizce hazırlanmış bir metinde “E”, “T” ve “A” çok sık rastlanan karakterlerken, “Z” daha nadir görünen bir karakterdir. Veri sıkıştırma konusunda yer alan fazlalığı en aza indirgemek de burada ortaya çıkmaktadır. Çünkü ASCII kod bir karaktere sık geçen veya geçmeyen olmasına bakmadan 8 bit sabit uzunluklu kod atamaktadır. Fazlalığı en aza indirgemek karakterler için değişken uzunluklu kod kullanarak mümkündür. Kısa kodlar frekansı yüksek olan

karakterlere, uzun kodlar frekansı düşük olan karakterlere atanabilir. Bu verilen örnek de Shannon–Fano ve Huffman Kodlamanın temel mantığı olarak bilinmektedir.

Veri sıkıştırma alanında kullanılan bazı teknik terimler aşağıda açıklanmıştır.

- **Kodlayıcı:** Giriş veri kanalından gelen ham veriyi sıkıştırarak çıkış veri kanalına iletmektedir.
- **Kod Çözücü:** Kod çözücü veya dekoder kodlayıcının yaptığı işlemi tersi yönde yapmaktadır.
- **Uyarlanamaz Sıkıştırma Algoritması:** Sıkıştırılan veriye bağlı kalmaksızın işlemlerini, parametrelerini veya tablolarını değiştirmeyen algoritmadır. Belirli veriler için tasarlanmıştır ve tablodaki verilere uymayan farklı verilerde düşük sıkıştırma oranına sahiptirler.
- **Uyarlanabilir Sıkıştırma Algoritması:** Ham verinin durumuna göre işlemlerini ve parametrelerini uyarlar. Veri ele alındıkça ya da işlendikçe bu parametreler değişiklik göstermektedir.
- **Yarı Uyarlanabilir Sıkıştırma Algoritması:** Veri üzerinde iki geçiş yapan algoritmalardır. İlk geçişte veri hakkında istatistikler elde edilir, ikinci geçişte ilk geçiş sonuçlarına göre oluşturulmuş parametreler veya kodlar kullanılarak veri sıkıştırılır.
- **Yerel Uyarlanabilir Sıkıştırma Algoritması:** Girdi verisindeki yerel durumlara göre kendini uyarlar ve girdi verisinde ilerledikçe bu uyarlama devam eder.
- **Kayıplı/Kayıpsız Sıkıştırma:** Kayıplı sıkıştırma algoritmaları bazı bilgilerin kaybolmasına göz yumarak yüksek oranda sıkıştırma sağlar. Sıkıştırılmış veri açıldığı zaman orijinal verinin birebir aynısı değildir. Genel olarak video, ses ve resim dosyalarını sıkıştırmak için kullanılırlar. Eğer kayıp oranı küçük ise bunun çıplak göz ile fark edilmesi çok zordur. Buna zıt olarak metin veriler gibi tek bir bitin bile önemli olduğu verilerde sadece kayıpsız sıkıştırma algoritması kullanılabilmektedir. Kayıpsız sıkıştırma algoritması kullanılarak sıkıştırılmış bir veri açıldığı zaman orijinal veri ile birebir aynı olarak elde edilir.
- **Simetrik Sıkıştırma:** Kodlayıcı ve kod çözücü tamamen aynı algoritmayı kullanır ama yönleri farklıdır.
- **Asimetrik Sıkıştırma:** Kodlayıcı ya da kod çözücü diğerinden daha zor ve karmaşık işlemler yapmak zorunda kalır.

- **Sıkıştırma Performansı:** Bir sıkıştırma algoritmasının başarımının ifade edilmesinde birçok nicelik vardır bunlar;

  - **Sıkıştırma Oranı:**

Sıkıştırma Oranı=Çıktı Dosyası Büyüklüğü/Girdi Dosyası Büyüklüğü

Örneğin 0,6 olarak elde edilen oran, çıktı dosyasının girdi dosyasının %60'ı büyüklüğünde olduğunu, bunun da %40'lık bir fazlalık indirgeme oranına denk geldiğini belirtir. Sıkıştırma oranı aynı zamanda bpb olarak da adlandırılabilir. Bu da girdi dosyasındaki ortalama bir bitin kaç bit ile ifade edildiği anlamına gelir. Benzer şekilde bu terim resimler için bpp'dir.

Sıkıştırma oranına ilave olarak bit oranından da bahsetmek gerekir. Bit oranı bpb veya bpc için genel bir tanımdır. Bu sebeple bir algoritmanın temel amacı verilen girdiyi düşük bit oranı ile ifade edebilmektir.

  - **Sıkıştırma Faktörü:** Sıkıştırma oranının tersidir.

Sıkıştırma Faktörü=Girdi Dosyası Büyüklüğü/Çıktı Dosyası Büyüklüğü

1'den büyük değerler sıkıştırmayı, küçük değerler ise veri miktarını arttırmayı veya genişletmeyi ifade etmektedir. Faktör ne kadar büyük olursa sıkıştırma o derece iyi yapılmış demektir.

- **Olasılık Modeli:** Bu kavram istatistiksel veri sıkıştırma metotlarında önemlidir. Bazen sıkıştırma algoritması olasılık modeli ve kodlayıcı olmak üzere iki parçadan oluşur. Bir sonraki veri nesnesi (bit, bayt, piksel vb.) sıkıştırılmadan önce, model uygulanır ve veri nesnesinin olasılığının tahmin edilmesi sağlanır. Nesne ve olasılık kodlayıcıya gönderilir. Kodlayıcı nesneyi sıkıştırmak için tahmin edilen olasılığı kullanır.
- **Alfabe:** Alfabe terimi sıkıştırılacak dosya içerisindeki semboller kümesini ifade eder. Alfabe 0 ve 1 bitlerinden, 128 ASCII karakterinden ve 256 olası 8-bit kodlamadan ya da herhangi bir diğer sembolden oluşabilir [Salomon, 2002].

Verilere uygulanan veri sıkıştırma tekniklerinin yanı sıra verilere uygulanan önemli diğer bir konu da *dizgi eşleme*'dir. Dizgi eşleme veri içerisinde bilgiye ulaşmak için kullanılmaktadır ve bilginin var olduğu konumları arama işlemidir. Örneğin bir ansiklopedide herhangi bir konu üzerinde araştırma yapılıyorsa, tüm ansiklopediyi okumak veya gözden geçirmek yerine sadece ilgili aranan alanları ele almak yeterli olacaktır. Bu durumda öncelikle ansiklopedide ilgili alanların bulunması gerekmektedir. İşte bu arama ve bulma işlemi kabaca dizgi eşleme olarak bilinmektedir. Bütün ansiklopedi ele alınır ve konu ile alakalı bir dizgi o ansiklopedi içinde aranır. Dizginin bulunduğu yerler de konu ile alakalı kısımlar olacaktır.

Dizgi eşleme, dizginin tam olarak geçtiği yerleri kapsayacağı gibi belli bir miktar hata payı ile bulunmasına da izin verebilir. Dizginin hata payı içeren bulunma durumuna da *kısmi eşleşme* denmektedir.

Genel olarak, $n>m$ olmak üzere, m uzunluklu dizginin, n uzunluklu metin içerisinde bulunması amaçlanmaktadır. Bir dizgi sadece bir kelime gibi yalın bir katar olabileceği gibi birden fazla kelime içeren karmaşık bir halde de yer alabilir. Birçok durumda m çok büyüktür. Örneğin dizgi, genetikte çok uzun olan bir genoma karşılık gelebilir, görüntü işlemede sayısallaştırılmış bir resim milyonlarca karakter genişliğinde olabilir. Bu iki örnek de son derece uzun ve karmaşık dizgi yapılarına sahiptirler [Crochemore ve Rytter,1997].

Veri sıkıştırma ve dizgi eşlemenin metin veriler üzerinde yapılan önemli iki işlem olduğu oldukça açıktır. Bunların yanında son 15 – 20 yılda iki konuyu da bünyesinde barındıran yeni bir çalışma alanı ortaya çıkmıştır. Bu yeni alanın konusu veriyi sıkıştırdıktan sonra, sıkıştırılmış veri üzerinde açma işlemini gerçekleştirmeden ya da kısmi olarak açma gerçekleştirerek arama yapma işlemidir. Bu işleme de *sıkıştırılmış dizgi eşleme* adı verilmiştir.

Sıkıştırılmış dizgi eşlemeden ilk olarak Amir ve Benson'ın 1992 yılındaki çalışmasında bahsedilmiştir [Amir ve Benson, 1992]. Sıkıştırılmış dizgi eşleme, verilen bir sıkıştırılmış metin üzerinde metni açmadan dizgi eşlemenin yapıldığı durumdur. Verilen bir T metni ve aranan dizgi P olsun, T metnine karşılık gelen sıkıştırılmış metin

$Z = z_1 \dots z_n$ olarak ifade edilir. Sıkıştırılmış dizgi eşleme, *P* dizgisinin *T* içerisindeki tüm konumlarının sadece *P* ve *Z* kullanarak bulma işlemidir.

Sıkıştırılmış dizgi eşlemenin önemli olmasının nedeni ise günümüz veritabanları ve sayısal arşivlerin iki ana probleminin olmasıdır. Bu iki problem sırasıyla, metinsel veritabanlarının Giriş/Çıkış zamanını ve kapladıkları yeri azaltmak için verilerin sıkıştırılmış durumda saklanma gereksinimleri ile bu veritabanlarının etkili bir biçimde aranabilir olmasıdır. Bu iki birleştirilmiş problemin beraber çözülmesi basit değildir. 90'lardan önce tek çözüm metni açıp dizgi eşlemeyi açık metin üzerinde yapmak suretiyle gerçekleştirilir idi. Bu da zaman ve yer maliyeti olarak ortaya olumsuz sonuçlar çıkarmaktaydı [Karkkainen vd.,2000].

Günümüzde giderek artan veri miktarı ve artan internet kullanımı bu verileri saklamayı ve aynı zamanda iletişim hatlarından hızlı bir şekilde iletmeyi gerekli kılmaktadır. Verilerin saklanması ve iletişim hızı sorununa bir çözüm olarak görülen veri sıkıştırma algoritmalarının, aynı zamanda sıkıştırılmış veriler üzerinde açma işlemi yapmadan dizgi eşleme yapmaya uygun olması isteğini gündeme getirmiştir. Bu amaçla birçok sıkıştırılmış dizge eşleme yaklaşımı geliştirilmiştir.

2. VERİ SIKIŞTIRMA ALGORİTMALARI

Veri sıkıştırma yöntemleri, veriyi sıkıştırma tekniklerine göre *kayıplı veri sıkıştırma* ve *kayıpsız veri sıkıştırma* yöntemleri olmak üzere ikiye ayrılırlar. *Kayıplı veri sıkıştırma yöntemlerinde* sıkıştırılan veri açıldığında orijinal verinin birebir aynısı elde edilememektedir. Bunun nedeni de sıkıştırma esnasında verinin önemsiz ya da başka bir deyişle veri bütünlüğünü en az derecede etkileyen veri kümelerinin çıkartılmasıdır. Bazı veri kümeleri çıkartılıp geri kalan veriye kayıpsız sıkıştırma yöntemi uygulandığı için de açma esnasında sadece kayıpsız sıkıştırılmış kesimler tekrar orijinal veri ile aynı şekilde elde edilir. Bu yöntemler ortaya çıkacak orijinal dosya ve açılmış dosya farklarının göz ardı edilebileceği uygulamalarda kullanılabilmektedir. Bu tarz uygulamalara örnek olarak ses dosyaları ve görüntü dosyaları gibi dosyalar verilebilir.

Kayıpsız veri sıkıştırma yöntemleri ise sıkıştırılacak olan verinin bir baytının ve hatta bir bitinin bile önemli olduğu durumlarda kullanılır. Buna verilebilecek en güzel örnek metin verilerdir. Bu tip bir sıkıştırma yönteminde sıkıştırılma işlemine sokulan bir veri, sıkıştırıldıktan sonra açma işleminin de yapılmasıyla birebir elde edilmektedir.

Değişken uzunluklu kodlama olarak da bilinen *olasılık (veya istatistik) tabanlı kodlama* ve *sözlük tabanlı kodlama* olmak üzere iki tip kayıpsız sıkıştırma yöntemi mevcuttur. Olasılık tabanlı kodlamada, sıkıştırılmak istenen verinin bütünü içinde daha sık kullanılan sembollere daha az bit adedi ile ifade edilen kodlar atanır.

2.1. Olasılık Tabanlı Teknikler

Sıkıştırılmak istenen verinin tek tek tüm sembollerinin veya birkaç sembolün bir araya getirilmesi ile oluşturulan alt sembol kümelerinin olasılıklarının bulunması ve bu olasılık dağılımlarını temel alarak verinin tekrar kodlanmasına *olasılık tabanlı kodlama* ve buna dayalı tekniklere de *olasılık tabanlı teknikler* adı verilir.

ASCII kodlamasında her sembolü temsil etmek için 8-bit uzunluğunda bir kod kullanıldığından, bu kodlama türüne *sabit uzunluklu kodlama* denir. Eğer bir mesajın daha az bit ile ifade edilmesi isteniyorsa, o mesajı oluşturan birbirinden farklı sembollerin, farklı sayıda bit ile ifade edilmesi, yani *değişken uzunluklu kodlama* yapılması gerekir. Eğer mesajda daha sık kullanılan semboller daha az bit ile ifade edilirse, mesajın tamamı için sembol başına kullanılan bit sayılarının ortalaması (ortalama uzunluk) azaltılmış olur.

En çok kullanılan olasılık tabanlı kodlama tekniği *Huffman* ve *Aritmetik Kodlama*'dır [Mesut,2006].

2.1.1. Huffman kodlama

Huffman kodlama genellikle metin sıkıştırmada kullanılan başarılı bir sıkıştırma tekniğidir. Ele alınan normal bir metinde bazı karakterler diğerlerinden daha sık kullanılmaktadır. Örneğin İngilizce metinlerde E, A, O, T harfleri J, Q, X harflerinden çok daha sık kullanılmaktadır.

Huffman Kodlamanın temel özelliği çok sık geçen karakter veya semboller için standart 8 bit kullanılan ASCII kod yerine daha az sayıda bitle ifade edilen kodlar atamaktır. Bu sebeple farklı frekans dağılımlarındaki karakterler için harcanan bit sayısı oldukça düşecektir. Örnek 2.1'de örnek bir metin ve metin içerisindeki harflerin frekansları verilmiştir.

Örnek 2.1: T= BILL BEATS BEN (Daha basit kodlama için 2 boşluk karakteri göz ardı edilmiştir.)

Sembol Frekansları: B I L E A T S N

3 1 2 2 1 1 1 1

Frekansa Göre Sıralanmış Liste: B L E I A T S N

3 2 2 1 1 1 1 1

Sıkıştırma algoritmasının girdisi metindir. Çıktısı ise o metinden üretilmiş bit sırasıdır.

Daha az sıklıkla geçen karakterlere uzun kod ve sık geçen karakterlere kısa kod atanması işlemi gerçekleştirilmek istenmektedir. Bu kodlar bir 0 – 1 ağacında tutulmaktadır. Bir ağaçta sembollerin karşılıkları kökten sembolün bulunduğu yaprağa gidene kadar elde edilen 0 ve 1'lerin dizisidir. Kısa kod kelimeleri ağacın düşük seviyeli yapraklarında yer alırken, uzun kod kelimeleri ağacın yüksek seviyeli yapraklarında yer almaktadır. En uzun son iki kod kelimesi son bit hariç birbirine eşittir. Örnek 2.1'de B sembolü L sembolünden daha yüksek bir frekansa sahiptir. Bu sebeple kod kelimesi daha kısa olacaktır. Bunun yanında yine örnek 2.1'de S sembolünün kod kelimesi eğer 0000 ise N sembolününki de 0001 olacaktır.

Bir ikili ağacı oluşturmak için iki yaklaşım mevcuttur. Bir tanesi ağaç oluşturmaya en alt seviyeden başlayıp köke doğru ilerlemek suretiyle gerçekleştiren, bir diğeri de kökten alt yapraklara doğru inerek ağacı oluşturan yaklaşımdır. Birinci yaklaşım Huffman Kodlamada kullanılırken, diğer yaklaşım da Shannon – Fano Kodlamada kullanılmaktadır.

2.1.1.1. Huffman yaklaşımı

Huffman yaklaşımında öncelikle ağaç yaprakları olacak sembollerin listesi elde edilir. Semboller bir başka sembolle veya bir başka alt ağaçla birleştirilerek yeni alt ağaçları oluşturur. Bu iki elemanlı ve tekrar eden şekilde liste tüketilene kadar gerçekleştirilir.

Burada yapraktan köke doğru bir yaklaşım benimsendiği için öncelikle liste azalan sırada sıralanmış olmalıdır. Ufak değişikliklerle artan sırada da kullanılabilir. Ağaç oluşturma işlemi adım adım şu şekilde gerçekleşmektedir.

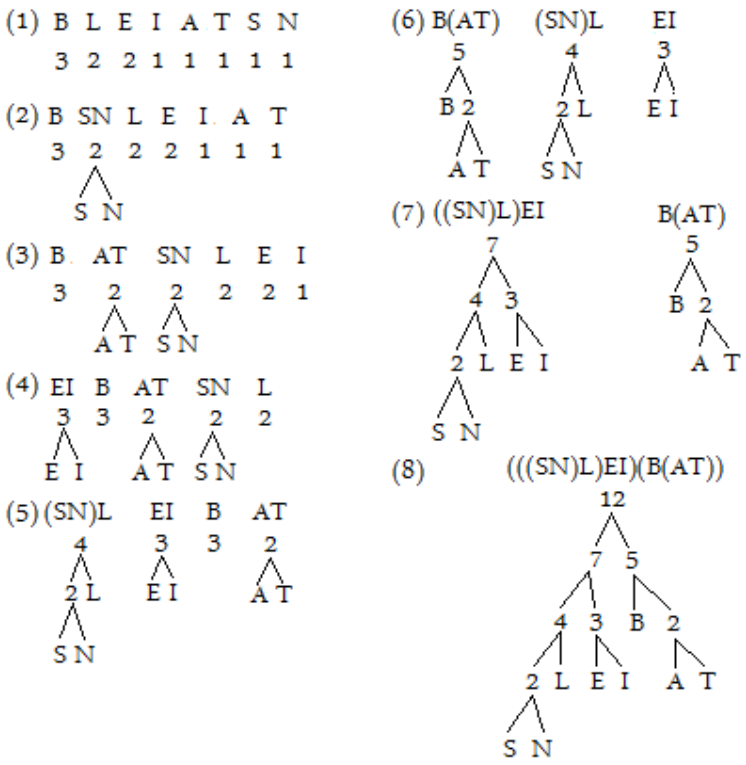
1- En az frekansa ya da olasılığa sahip iki eleman birleştirilir ve elemanların birleştirilmiş hali olarak kodlanan sembol, birleştirilen iki eleman silindikten sonra listeye eklenir.

2- Bu yeni eklenen elemanın frekansı ya da olasılığı içerdiği iki elemanın frekansları toplamına eşittir. Bundan sonra da listede bu yeni hesaplanan frekansa göre sıralanmış olarak tutulacaktır.

Şekil 2.1'de örnek 2.1'deki metin için Huffman Ağacı oluşturma adımları verilmiştir.

Örneğin şekil 2.1'de ilk adımda S ve N karakterleri birleştirilerek SN karakterini oluştururken, yeni oluşturulan birleşimin frekansı 2 olmuştur. İkinci adımda bu SN birleşimi 2. konumda yer almaktadır.

Burada birleştirilmiş sembolün frekansı 2 olduğu için birleşim katarı I harfinden hemen sonra, E harfinden hemen önce de yerleştirilebilirdi. Ancak yeni birleştirilmiş katarı daha kısa zamanda yeniden bir birleştirmeye sokmamak için bundan kaçınılmıştır.



Şekil 2.1 Huffman Ağacı oluşturma adımları

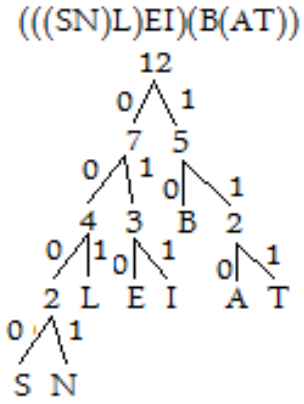
Alfabe $P = p_1, p_2 \dots p_n$ olasılıkları ve $S = s_1, s_2 \dots s_n$ azalan sırada karakterler dizisi olarak ifade edilirse, ikili ağaç oluşturmak için şu adımlar işlenir:

1- Eğer bir sembol varsa, hem ağaç kökü hem de yapraktır. Aksi halde en düşük olasılıklı (p_i ve p_j) iki sembol (s_i ve s_j) alınır.

2- Alfabeden s_i ve s_j silinir ve yeni birleştirilmiş (s_i,s_j) sembolü p_i+p_j olasılıkla alfabeye eklenir. Bu yeni sembol bir alt ağacın kökünü gösterir. Şu anda alfabe eskisinden bir tane az harfe sahiptir.

3- (s_i,s_j) sembolü azalan sırayı bozmayacak şekilde mümkün olan en yüksek yere yazılır.

Şekil 2.2'de örnek 2.1'deki metin için oluşturulmuş olan Huffman Ağacında kod atanması gösterilmektedir. İkili ağaç oluşturulduktan sonra şekil 2.2'de olduğu gibi sol dala 0 ve sağ dala 1 atamak oldukça kolaydır. Kenarların yanına konan 0 ve 1 değerleri *ağırlık* olarak adlandırılır. Bu 0 – 1 etiketlerine sahip ağaçlara da *ağırlıklı ağaçlar* denir. Bu şekilde elde edilen ağaç *Huffman Ağacı* olarak kabul edilir.



Şekil 2.2 Huffman Ağacı

Daha sonra kökten yaprağa her sembol için 0 ve 1 değerleri toplanır. Bir yaprağa ulaşıldığı zaman, 0 ve 1'lerin oluşturduğu grup o sembol için elde edilen kod kelimesidir. Bu şekilde elde edilen kodlara da *Huffman Kodları* denir.

Örneğin, E karakteri için 0 ve 1'leri toplarken öncelikle sol 0 dalına, arkasından sağ 1 dalına ve son olarak da yine sol 0 dalına bakılır. Böylece E sembolü için "010" Huffman Kodu elde edilir. Böylece (B, L, E, I, A, T, S, N) karakterleri için sırasıyla (10, 001, 010, 011, 110, 111, 0000, 0001) kodları hesaplanmış olur.

Huffman Sıkıştırma Algoritması gerçekleştirilirken öncelikle yapılması gereken adım Huffman Ağacı'nı oluşturmaktır. Daha sonra 0 ve 1'lerle etiketleme işlemi gerekleştirilir. Düz metinden okunan bir karakterin kod kelimesi bu ağaçlar yardımıyla elde edilir. Elde edilen kod kelimesi çıktı katarına eklenir ve yeni sembole geçilir.

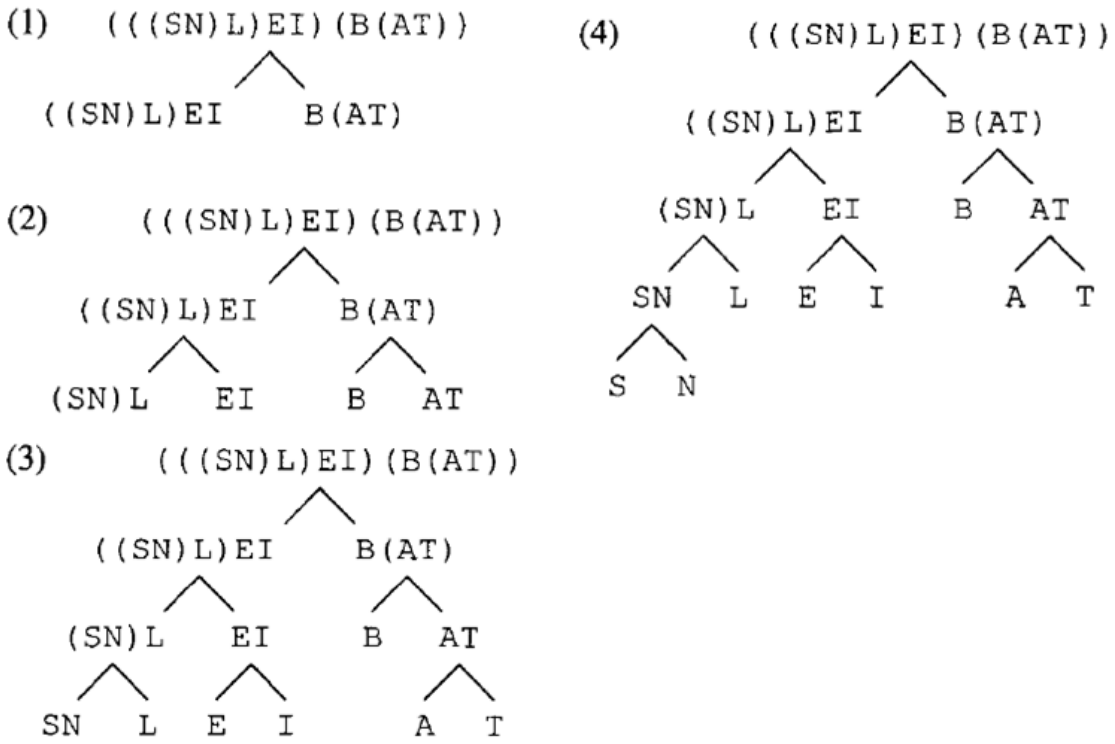
Huffman Sıkıştırma Algoritması ile sıkıştırılmış bir mesajın açılması işlemi ise her kod için eldeki kod kelimesine uygun Huffman Ağacı ilerlemesini içermektedir. Bu adımlar örnek 2.2'de gösterilmiştir.

Örnek 2.2: 0000010001 kod kelimesinin Huffman Ağacı kullanarak açılması esnasında bitler tek tek okunur. İlk adımda okuna bit 0 olduğu için kökten ilk sol dala dallanılır. İkinci adımda okunan bit yine 0 olduğu için yine sol dala dallanılır. Bu şekilde adımlar ilerlerken 4. adımda "S" karakterini içeren yaprağa ulaşılır. Çıktı olarak "S" üretilir. İşlem kalan bit dizisi için baştan başlatılır ve yine iki adım sonrasında "E" karakteri elde edilir. EOF anlamına gelen bir işaret görene kadar katar tüketilir.

2.1.1.2. Shannon – Fano yaklaşımı

Shannon – Fano yaklaşımı Huffman yaklaşımına oldukça benzemektedir. İki yaklaşım birbirinden sadece ikili ağacı oluştururken farklılık göstermektedir. Shannon – Fano yaklaşımında ikili ağaç kökten alt seviye yapraklara doğru oluşturulmaktadır. Örnek 2.3'te Shannon – Fano yaklaşımına göre ağaç oluşturmanın adımları verilmiştir.

Örnek 2.3: (((SN)L)EI)(B(AT)) metnindeki tüm karakterler kökü oluştursun. Parantezler de sembollerin nasıl birleştiğini belli etsin. Parantezler ayrıca kökten yaprağa doğru ağacı kurmaya da yararlar. Her iterasyonda nod iki parçaya bölünür ve her parça da sağ ve sol alt ağaçları belirtir.



Şekil 2.3 Shannon – Fano ağacı

Örnek 2.3'te (((SN)L)EI)(B(AT)) katarının ((SN)L)EI ve B(AT) olarak ikiye ayrılacağı anlaşılmaktadır. İlk parça sol, ikinci parça da sağ alt ağacı oluşturur. Daha sonraki adımda her parça kendi içinde iki parçaya ayrılır. Bu bölünme her parçada tek bir sembol kalana kadar devam eder. Burada bilinmesi gereken bölünme için seçilecek olan orta noktadır. Bu durumda olasılıkların bilinmesi bu bölünme noktasının tayini için önemli rol oynamaktadır.

Örnek 2.4'te frekansları verilmiş olan harflere göre Shannon – Fano yaklaşımı ağacı oluşturulmuştur.

Örnek 2.4: S=(B, L, E, I, A, T, S, N) ve P=(3, 2, 2, 1, 1, 1, 1, 1) ise karakterler frekans toplamları farkı en az olacak şekilde ikiye bölünür. İlk parça (B, L) ve ikinci parça da (E, I, A, T, S, N) seçilirse ise frekans toplamları farkları:

$$|(2+3)-(2+1+1+1+1+1)|=2$$

olur. Bir sonraki bölünmede ise ilk parça B ve L, ikinci parça da (E, I) ve (A, T, S, N) olmak üzere ikiye ayrılır. Burada frekans farkları sırasıyla

$$|3-2|=1$$

ve

$$|(2+1)-(1+1+1+1)|=1$$

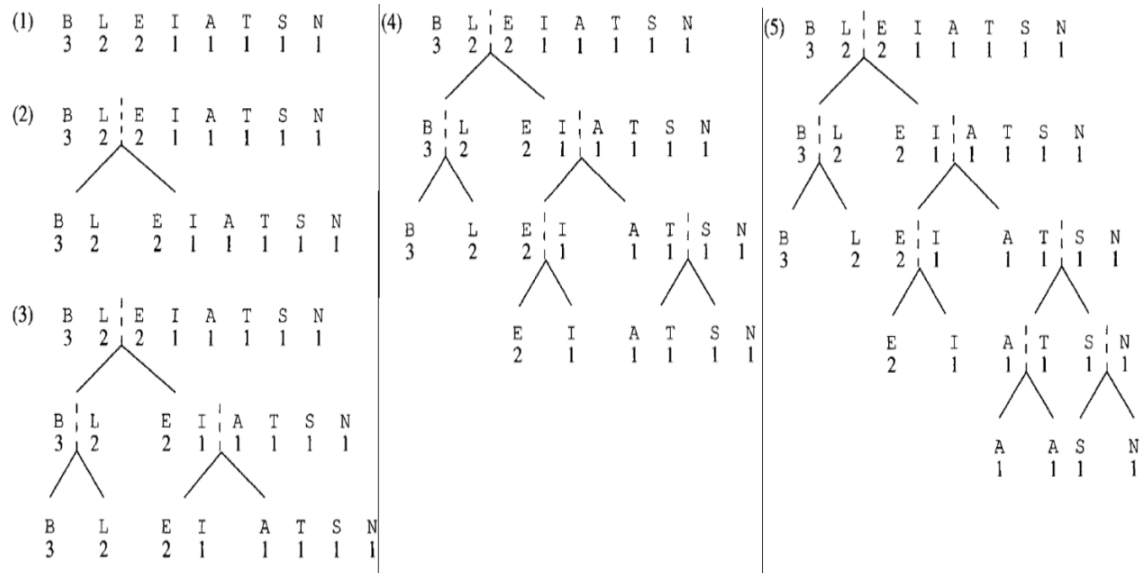
olur. Bu işlem her yaprakta bir sembol kalana kadar devam etmektedir.

Aslında amaç kod kelimesi üretmek olduğu için 0 ve 1 ile etiketleme işlemi bu bölünme adımlarına ilave edilir.

Sıkıştırma algoritması ise şu adımlardan oluşmaktadır:

1- Bir frekans tablosu oluşturulması
2- Tabloyu frekansa göre sıralama (Azalan sıra ile)
3- Yaklaşık eşit frekans toplamları olacak şekilde tablonun ikiye bölünmesi
4- Listenin üst kısmına 0, alt kısmına ise 1 atanması
5- Bölme (3) ve atama (5) adımlarının her yaprakta bir sembol olana kadar devam etmesi

Şekil 2.4'te örnek 2.4'te verilmiş olan harfler ve frekanslarına göre SF ağacı oluşturmanın adımları gösterilmiştir.



Şekil 2.4 Shannon – Fano ağacı oluşturma adımları

Örnek 2.5'te frekansları verilen harflerin oluşturulacak SF ağacında alacakları kod karşılıklarının belirlenmesi adımları gösterilmiştir.

Örnek 2.5: Sembol: A B C D E

Frekans: 15 7 6 6 5

Çözüm: İlk Bölme:

a- Tablo parçaların frekansları birbirine çok yakın olacak şekilde ikiye bölünür.

Sembol: A B | C D E

Frekans: 15 7 | 6 6 5

Toplam (22) | (17)

b- Her parçaya bir bit atanır. (Üst gruba 0, alt gruba 1 atanır.)

Sembol: A B | C D E

Frekans: 15 7 | 6 6 5

Toplam (22) | (17)

Kod Kelimesi: 0 | 1

İkinci Bölme: Her grup için yinelemeli tekrar et.

Sembol: A | B, C | D E

Frekans: 15 | 7, 6 | 6 5

Toplam: (15)|(7), (6)|(11)

Kod Kelimesi:00 | 01, 10 | 11

Üçüncü Bölme:

Sembol: A , B, C, D | E

Frekans: 15, 7, 6, 6 | 5

Kod Kelimesi:00, 01, 10, 110 | 111

Yinelemeli işlem bittiği zaman aşağıdaki kod kelimeleri elde edilmiş olur.

A	B	C	D	E
00	01	10	110	111

Açma işlemi Huffman Kodlama'daki açma işlemi ile aynıdır. Öncelikle SF ağacı oluşturulur. Daha sonra okunan kod kelimesi bir ağaç yaprağına götürene kadar ağaçta ilerlemeler yapılır. Yaprakta tek sembol bulununca o sembol çıktı olarak yazılır [Pu,2006].

2.1.2. Kelime tabanlı Huffman kodları

Huffman Kodları'nın geleneksel uygulaması karakter tabanlıdır. Karakterler alfabe sembolleri olarak ele alınır. Kelimeleri sembol olarak alan yaklaşımda sıkıştırma oranı daha da yükselmiştir. Karakter tabanlı Huffman kodlama orijinal metni %60 boyutuna indirirken, kelime tabanlı Huffman kodlamada bu oran %25-%30'a kadar düşmektedir. Şekil 2.5'te bayt kodlamalı Huffman Ağacına bir örnek verilmiştir.

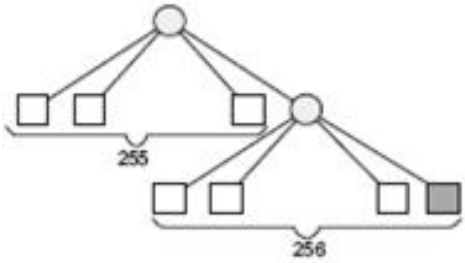
Bunun yanında kelimeleri temel semboller olarak kullanmak bilgi çıkarım sistemleri için de yararlı bir uygulama olmuştur. Çünkü bu yapıların da dayandığı temel sembol kelimelerdir.

Bu yaklaşımı kullanan iki tip Huffman kodu vardır. Bunlar *Düz Huffman Kod* ve *Etiketlenmiş Huffman Kod'*dur.

Huffman tarafından bulunan yaklaşım çoğunlukla bit kodlama esasına dayanmaktadır. Orijinal metindeki her sembol bitlerle ifade edilir. Ancak bunun yerine

kodlamayı bayt sırası şeklinde yapan yaklaşımlar da geliştirilmiştir. Bit yerine bayt kodlama kullanmak %5 gibi bir sıkıştırma oranı zafiyetine yol açsa da, açma ve sıkıştırılmış dizgi eşleme çok daha hızlı hale gelmektedir. Çünkü bayt üzerinde işlem yapmak, bitlerden bayt değerleri hesaplamaktan daha hızlıdır.

2^b boyutlu Huffman Ağacı oluşturmak, ikili Huffman Ağacı oluşturmaya benzemektedir. Tek fark, çocuk sayısı ilk aşamada belirlenmek zorundadır. [Martinez,2005]



Şekil 2.5 Bayt kodlamalı Huffman ağacı

2.1.2.1. Düz Huffman kodlama

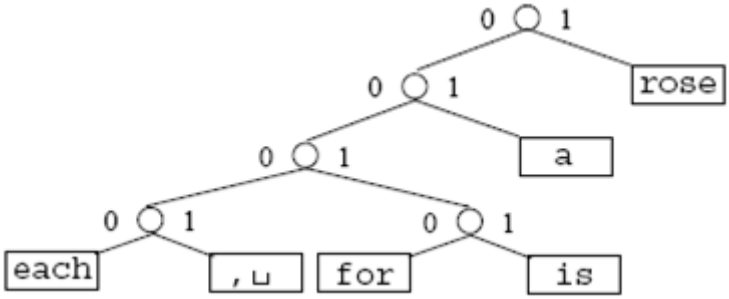
Yarı statik olan bu Huffman yönteminde semboller olarak karakterler yerine kelimeler kullanılmaktadır. İlk geçişte kelime frekansları bulunur ve kelimeler temel alınarak Huffman Ağaçları oluşturulur. İkinci geçişte ise sıkıştırma işlemi gerçekleştirilir.

Ağaç oluşturma işlemi Karakter Tabanlı Huffman Kodlama ile aynıdır. Aralarındaki tek fark olarak birinde karakterler işleme alınırken, diğerinde kelimeler aynı birer karaktermiş gibi işleme alınır. Örnek 2.6'da bu tip kodlamaya ve Şekil 2.6'da Huffman Ağacına bir örnek verilmiştir.

Örnek 2.6.:

Orijinal metin: for each rose, a rose is a rose

Sıkıştırılmış metin: 0010 0000 1 0001 01 1 0011 01 1



Şekil 2.6 Kelime tabanlı Huffman ağacı örneği

Kelime tabanlı sıkıştırma algoritması normal karakter tabanlı algoritmadan daha iyi sıkıştırma oranlarına sahiptir.

Metinler sadece kelimelerden değil ayraçlardan da oluştuğu için, bunları da hesaba katan bir formül düşünülmüştür. Ayraçlar için üretilen bir çözüm, hem karakterler hem de ayraçlar için ayrı alfabeler kullanmaktır. Eğer metnin hangi alfabeyle başladığı bilinirse, açma işlemi esnasında herhangi bir güçlük çıkmayacaktır.

Kelime tabanlı kodlama yönteminin bir çeşidi de "boşluksuz kelime modeli" kullanan yöntemdir. Bu yöntemde her kelimenin bir boşluk karakteri ile takip edildiği varsayılır ve kodlama esnasında bu boşluk ayrıca kodlanmaz. Bu model tek bir alfabe kullanmaktadır ve biraz da olsa sıkıştırma oranında iyileştirme yapmıştır. Örnek 2.6 "boşluksuz kelime modeli"ni kullanan bir örnektir. Şekil 2.6.'daki ağaçta da görüldüğü gibi kelimeler ağaca yerleştirilirken ve sıkıştırılmış metinde kodlanırken arkasındaki boşluklar kodlanmamıştır [Moura vd.,2000].

2.1.2.2. Etiketlenmiş huffman kodu

Kelime tabanlı Huffman uygulamalarından biri de Etiketlenmiş Huffman Kodlamadır. Bu kodlama daha çok sıkıştırılmış dizgi eşleme yapmak için kullanılan bir yöntemdir. Sıkıştırma oranı ise Kelime Tabanlı Düz Huffman Kodlamadan düşüktür.

Her baytta kelimeyi 7 bitle kodlamak esasına dayanmaktadır. Kalan 1 bit ise baytın kod kelimesi başlangıcı olup olmadığını işaretlemek için kullanılmaktadır.

Bu kodlamada tam baytlar kullanılır ancak bu tam baytların 7 biti kodlama işlemini yaptığı için sıkıştırılmış dosya yaklaşık %11 oranında büyümektedir. Ancak sıkıştırılmış dizgi eşleme performansı PHC'den daha iyidir.

Örnek 2.7.'de normal dağılıma sahip 17 kelimelik bir metnin var olduğu tasarlanmıştır. 17 kelime ve bu kelimelerin normal dağılıma göre aldıkları kod karşılıkları çizelge 2.1'de, üssel dağılıma göre aldıkları kod karşılıkları çizelge 2.2'de verilmiştir. Şekil 2.7'de normal dağılım için oluşturulmuş Huffman Ağacı ve şekil 2.8'de de üssel dağılım için oluşturulan Huffman Ağacı gösterilmiştir. Örneğin sadeliği açısından da bir baytın 3 bit ile temsil edildiği düşünülmüştür. Bu sebeple THC bir baytın sadece 2 bitini kodlama için kullanmıştır. [Martinez,2005]

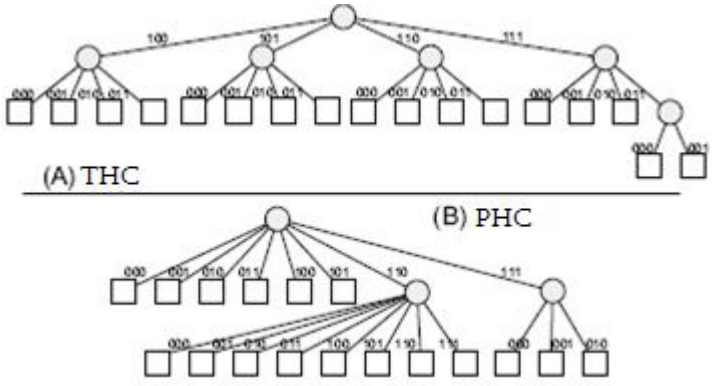
Örnek 2.7:

Çizelge 2.1 Normal dağılım için Huffman tabloları

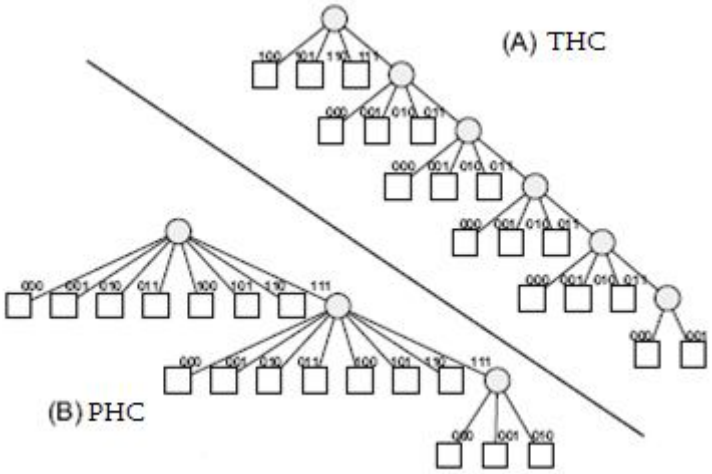
Kelime	Olasılık	PHC	THC
A	1/17	000	100 000
B	1/17	001	100 001
C	1/17	010	100 010
D	1/17	011	100 011
E	1/17	100	101 000
F	1/17	101	101 001
G	1/17	110 000	101 010
H	1/17	110 001	101 011
I	1/17	110 010	110 000
J	1/17	110 011	110 001
K	1/17	110 100	110 010
L	1/17	110 101	110 011
M	1/17	110 110	111 000
N	1/17	110 111	111 001
O	1/17	111 000	111 010
P	1/17	111 001	111 011 000
Q	1/17	111 010	111 011 001

Çizelge 2.2 Üssel dağılım için kod tablosu

Kelime	Olasılık	PHC	THC
A	1/2	000	100
B	1/4	001	101
C	1/8	010	110
D	1/16	011	111 000
E	1/32	100	111 001
F	1/64	101	111 010
G	1/128	110	111 011 000
H	1/256	111 000	111 011 001
I	1/512	111 001	111 011 010
J	1/1024	111 010	111 011 011 000
K	1/2048	111 011	111 011 011 001
L	1/4096	111 100	111 011 011 010
M	1/8192	111 101	111 011 011 011 000
N	1/16384	111 110	111 011 011 011 001
O	1/32768	111 111 000	111 011 011 011 010
P	1/65536	111 111 001	111 011 011 011 000
Q	1/65536	111 111 010	111 011 011 011 001



Şekil 2.7 Normal dağılım için PHC ve THC ağaçları



Şekil 2.8 Üssel dağılım için PHC ve THC ağaçları

2.1.3. Sonu etiketlenmiş yoğun kodlar

THC algoritması PHC algoritmasından iki yönüyle farklılık göstermektedir. Bunlardan birincisi sıkıştırılmış dizgi eşleme yapılabilme imkânına, diğeri de sıkıştırılmış veriye doğrudan erişim yapabilme imkânına sahip olmasıdır. Doğrudan erişim, içerdiği etiket biti sebebiyle bir kod kelimesinin nerede başlayıp nerede bittiğinin bilinmesi, dolayısıyla sıkıştırılmış metnin istenen kod kelimesine ulaşılması veya istenen kod kelimesinin açılabilmesi anlamına gelmektedir. THC, 7 bit kodlama ve 1 işaret biti kavramını kullandığı için bir bayt içindeki bitlerin tüm dizilimlerini içermez. İşaret biti kullanılması sıkıştırma oranının optimal olmadığı anlamına gelmektedir.

ETDC de PHC ve THC gibi yarı statik bir algoritmadır ancak tam anlamıyla Huffman Kodlama'ya dayanmamaktadır. Amacı THC'nin sıkıştırma oranını artırmak ancak bu artışı sağlarken de THC'nin iyi yönlerini de korumaktır. Çizelge 2.3'te THC ve ETDC için temsili kod karşılıkları ve bayt sayıları verilmiştir.

Çizelge 2.3 THC ve ETDC için kod kelimeleri yapısı

Bayt	THC	ETDC
1	1xxxxxxx	1xxxxxxx
2	1xxxxxxx 0xxxxxxx	0xxxxxxx 1xxxxxxx
3	1xxxxxxx 0xxxxxxx 0xxxxxxx	0xxxxxxx 0xxxxxxx 1xxxxxxx
…	…	…
N	1xxxxxxx 0xxxxxxx … 0xxxxxxx	0xxxxxxx…0xxxxxxx 1xxxxxxx

Bu iyi yönler:

- THC gibi her baytta bulunan işaret biti sayesinde kısmi açmalara izin verir.
- BM arama algoritması ile sıkıştırılmış dizgi eşleme yapmaya müsaittir.

Bunların yanında ETDC algoritmasında açma ve sıkıştırma işlemleri daha hızlıdır.

ETDC'de işaret biti ilk baytın başına değil, son baytın başına konur. İlk baytlarda işaret biti değeri 0 iken son baytta işaret biti değeri 1'dir. Bu değişimin şaşırtıcı sonuçları vardır. Bir sembole atanan değer, başka bir sembole atanan değerin öneki olamaz. Örneğin X ve Y iki kod kelimesi ve X, Y'den kısa olsun. X, Y'nin öneki olamaz çünkü X'in son baytında işaret biti 0'dır. Buna karşın Y'de ona karşılık gelen baytta bu değer 1'dir.

Önek olamama durumu sebebiyle kalan 7 bit için Huffman Kodu kullanmaya gerek yoktur. Bu 7 bitin tüm dizilimleri kullanılabilmektedir.

Kodlama aşaması Huffman Kodlama'dan hızlı ve kolaydır. Çünkü herhangi bir ağaç oluşturulmaz. 8 bit kullanılan bir kodlama şeması çizelge 2.4'teki gibi gerçekleştirilir:

- Sözlükteki kelimeler azalan sırada sıralanır.
- 128 – 255 değerleri arasındaki kod kelimeleri ilk 128 kelimeye atanır.
- 128 ile 16511 arası değerler sırasıyla iki baytlı kod kelimesi olarak atanır. Her kod kelimesinin ilk baytı 128'den küçük değer ve ikinci baytı da 128 – 255 arası değer alır.

- 16511'den yukarıdaki değerler 3 baytlı kod kelimesi olarak atanır.

Çizelge 2.4 ETDC'de kod ataması

Kelime Aralığı	Atanan Kod Kelimesi	Bayt Sayısı	Kelimeler
0	10000000	1	
1	10000001	1	
2	10000010	1	2^7
…	…	…	
2^7-1=127	11111111	1	
2^7=128	00000000:10000000	2	
129	00000000:10000001	2	
130	00000000:10000010	2	$_7\ 2^72$
…	…	…	
$2^72^7+2^7$-1=16511	01111111:11111111	2	
$2^72^7+2^7$=16512	00000000:00000000:10000000	3	
16513	00000000:00000000:10000001	3	
16514	00000000:00000000:10000010	3	$_3\ (2^7)$
…	…	…	
$(2^7)^3+(2^7)^2+2^7+1$	01111111:01111111:11111111	3	
…	…	…	…

Kelimelere karşılık gelen kod kelimelerini hesaplamak son derece basittir. Yapılması gereken tek şey sözlükteki kelimeleri frekanslarına göre azalan sırada sıralamak ve kodları sırayla atamaktır. Bu nedenle kod üretmek Huffman Kodlama'dan hızlı ve kolaydır.

Çizelge 2.5.'te 17 kelimelik bir metinde kod atanması örneği gösterilmiştir. Burada kelimelerin normal dağılım gösterdiği ve bir baytın 3 bitten oluştuğu varsayılmaktadır. Çizelge 2.6'da kelimelerin üssel dağılım gösterdiği durumlarda oluşacak kod atanması gösterilmiştir.

Çizelge 2.5 Normal dağılım için kod tablosu

Kelime	Olasılık	PHC	THC	ETDC
A	1/17	000	100 000	100
B	1/17	001	100 001	101
C	1/17	010	100 010	110
D	1/17	011	100 011	111
E	1/17	100	101 000	000 100
F	1/17	101	101 001	000 101
G	1/17	110 000	101 010	000 110
H	1/17	110 001	101 011	000 111
I	1/17	110 010	110 000	001 100
J	1/17	110 011	110 001	001 101
K	1/17	110 100	110 010	001 110
L	1/17	110 101	110 011	001 111
M	1/17	110 110	111 000	010 100
N	1/17	110 111	111 001	010 101
O	1/17	111 000	111 010	010 110
P	1/17	111 001	111 011 000	010 111
Q	1/17	111 010	111 011 001	011 100

Çizelge 2.6 Üssel dağılım için kod tablosu

Kelime	Olasılık	PHC	THC	ETDC
A	1/2	000	100	100
B	1/4	001	101	101
C	1/8	010	110	110
D	1/16	011	111 000	111
E	1/32	100	111 001	000 100
F	1/64	101	111 010	000 101
G	1/128	110	111 011 000	000 110
H	1/256	111 000	111 011 001	000 111
I	1/512	111 001	111 011 010	001 100
J	1/1024	111 010	111 011 011 000	001 101
K	1/2048	111 011	111 011 011 001	001 110
L	1/4096	111 100	111 011 011 010	001 111
M	1/8192	111 101	111 011 011 011 000	010 100
N	1/16384	111 110	111 011 011 011 001	010 101
O	1/32768	111 111 000	111 011 011 011 010	010 110
P	1/65536	111 111 001	111 011 011 011 000	010 111
Q	1/65536	111 111 010	111 011 011 011 001	011 100

Diğer tüm iki geçişli algoritmalar gibi metin sıkıştırıldıktan sonra, kullanılan sözlük de sıkıştırılmış dosyaya açma işleminde kullanılmak üzere eklenir. Ancak ayrıca

kod kelimelerinin de saklanmasına gerek yoktur. Sadece sıralı kelimelerin saklanması yeterlidir. Sözlük boyutu Huffman Ağacından da oldukça küçüktür.

Sıkıştırılmış metni açmanın ilk adımı sıkıştırılmış dosyadaki sözlük vektörünü elde etmektir. Sıkıştırılmış dosyadaki sözlük vektöründe de kelimeler sıralı olarak yer almaktadır. Dolayısıyla vektör elde edildikten sonra ayrıca kelime sıralamaya gerek yoktur.

sozluk[i] elemanına ulaşmak için, alınan bir *x* kod kelimesinin her baytı işlenir. Son baytın işaret biti 1 olduğu için 127'den büyük bir değerdir. Bu da kod kelimesi sonunu belirtir. İşlenen baytlardan elde edilen değerler, *i* indeksinin gösterdiği sözlük elemanına karşılık gelir [Martinez,2005].

2.1.4. (s,c) – yoğun kodlar

(s,c) – Yoğun Kodlar tekniği, ETDC tekniğini genelleştiren, iki geçişli bir tekniktir. ETDC bir kod kelimesini bitirmeyen baytlar için 0'dan 2^b-1'e kadar 2^{b-1} basamak kullanmaktadır (*b* kullanılan bit sayısını ifade etmektedir). Diğer 2^{b-1} basamak da kod kelimesini bitiren baytlar için kullanılmaktadır. Bunlardan ilkine *devam eden*, diğerine de *sonlandırıcı* adı verilir. Burada devam eden ve sonlandırıcı sayılarının optimal olup olmadığı sorusu ortaya çıkmaktadır. Verilen belirli bir kelime frekans dağılımı içeren metin kümesinde, devam eden ve sonlandırıcıların optimal sayısı bulunmak istenir. Bu da muhtemelen 2^{b-1}'den farklıdır. Bu mantık da ETDC ve (s,c)-DC arasındaki temel farktır. (s,c)-DC'de amaç kod kelimesinin ikiye ayrılabilip, arama işlemini sadece bir parçasında yapılabilecek şekilde üretilmesidir.

Örneğin, sıkıştırılacak bir metin dosyası sözlüğünde 5000 kelime olduğu ve kod kelimesi oluşturmak için baytların kullanıldığı (dolayısıyla b=8) varsayılsın.

Eğer ETDC kullanılmış ve sonlandırıcı ile devam eden bayt sayısı 128 olsaydı, bir bayttan 128 kod kelimesi üretilebilecek, bu üretilen 128 değer frekansı en yüksek olan ilk 128 kelimeye atanacaktır ve kalan diğer kelimeler de 2 baytlı kod kelimesine sahip olacaktır. Bu da;

$$128+128^2 = 16512$$

kelimenin bir ve iki bayt ile kodlanabileceği anlamına gelmektedir. Buradan yola çıkarak;

$$16512 - 5000 = 11512$$

kullanılmayan kod kelimesi kalacaktır.

Eğer sonlandırıcı değeri 230 (bu durumda devam eden değeri $256 - 230 = 26$) seçilmiş olursa,

$$230x230 = 6210$$

kelime 1 veya 2 bayt ile kodlanabilir olacaktır. Böylece 5000 kelime şu şekilde1 veya 2 bayt ile ifade edilebilecektir: en sık geçen 230 kelime 1 baytlı kodlara, kalan

$$5000 - 230 = 4770$$

kelime de 2 baytlı kodlara atanacaktır.

Sonuç olarak sonlandırıcı ve devam eden kelimelerin sayısının hesaplanmasında:

- Sözlük boyutu,
- Sözlüğün frekans dağılımı önemli rol oynamaktadır.

8 bitlik (b=8) semboller kullanılarak, kodlama işlemi aşağıdaki gibi ifade edilebilir (s-sonlandırıcı bayt sayısı, c- devam eden bayt sayısını vermektedir):

- 0'dan s-1'e kadar olan bir baytlık kod kelimeleri, sözlükteki ilk s tane kelimeye verilir.
- s'den s+sc-1'e kadar olan iki baytlık kod kelimeleri sırasıyla sözlük kelimelerine dağıtılır.
- s+sc'den s+sc+sc^2-1'e kadar olan kod kelimeleri de 3 bayt ile ifade edilir ve bu böyle devam eder.

Çizelge 2.7'de (s,c)-yoğun kodlar için kod atanması örneği verilmiştir.

Çizelge 2.7 (s,c)-yoğun kodda kod ataması

Kelime Aralığı	Atanan Kod Kelimesi	Bayt Sayısı	Kelimeler
0	[0]	1	
1	[1]	1	
2	[2]	1	s
…	…	…	
s-1	[s-1]	1	
s	[s][0]	2	
s+1	[s][1]	2	
s+2	[s][2]	2	
…	…	…	
s+s-1	[s][s-1]	2	sc
s+s	[s+1][0]	2	
s+s+1	[s+1][1]	2	
…	…	…	
s+sc-1	[sc-1][s-1]	2	
s+sc	[s][s][0]	3	
s+sc+1	[s][s][1]	3	
s+sc+2	[s][s][2]	3	sc^2
…	…	…	
$s+sc+sc^2-1$	[s+c-1][s+c-1][s-1]	3	
…	…	…	…

Burada *s* ve *c* değerlerinin seçimi önemlidir. Eğer *s* büyük seçilir ise, daha çok sık geçen kelime bir bayt ile ifade edilir. Ancak iki baytla ifade edilebilecek kelime sayısı azalacaktır. Eğer *s* küçük seçilirse, bu sefer çok sık geçen birçok kelime iki baytla ifade edilecek, bunun yanında iki baytla ifade edilebilecek kelime sayısı da artmış olacaktır.

s ve *c* değerleri elde edildikten sonra kod üretme kısmı oldukça kolaydır. Yine kod kelimesi veya frekansı sıkıştırılmış dosyada da tutmak gereksizdir. Sadece *s* değerini ve kelimeleri sıralı halde saklamak yeterli olacaktır.

Kodlama ve kod çözme aşamaları ETDC ile aynıdır. Sadece değişen nokta 128 yerine bir *s* değerini temel almaktır. [Martinez,2005]

2.1.5. Sınırlı önek bayt kodlama

Sınırlı Önek Bayt Kodlama her kod kelimesinin ilk baytını, kod kelimesinin uzunluğunu gösterecek şekilde kodlamaktadır. Ardından gelen baytlar arta kalan tüm kod uzayını kullanabilmektedir [Culpepper ve Moffat,2006].

(s,c)-DC'de belirli bir *s* değeri ve ona karşılık gelen bir bayt için uygun kod uzayı (S/(S+C)) vardır. İki bayt için belirlenen kod uzayı da (S/(S+C))'dir. Diğer baytlar için de bu aynı oran geçerlidir.

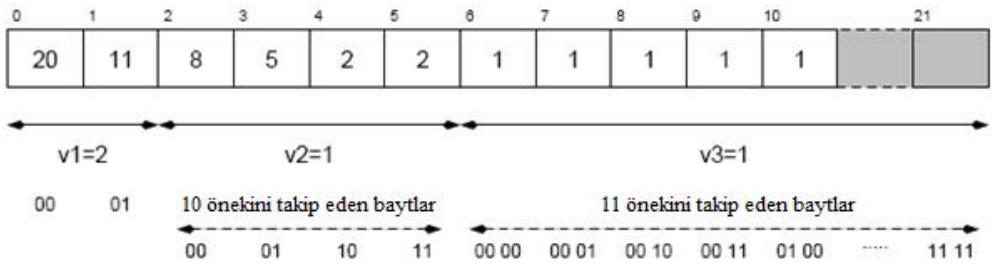
Bayt seviyeli Huffman Kodlamada ise kod kelimesi ancak 4 farklı bayt uzunlukta olabilmektedir. Bunlar 1 bayt, 2 bayt, 3 bayt ve 4 bayt uzunluklardır. Azalan olasılık dağılımlı *n* sembol dağılımında 0…(n-1) adet sembol 4 alt kümeye bölünmektedir. Eğer bu kümelerin boyutları sırasıyla h_1, h_2, h_3 ve h_4 olursa, $n = h_1 + h_2 + h_3 + h_4$ olur.

256 dereceli (radix-256, R=256) Huffman kodlama gibi, 4 sayıdan (v_1, v_2, v_3, v_4) oluşan bir düzenleme kullanılır. Ancak bu sayılar kod aralıklarını vermektedir.

$$v_1 + v_2 + v_3 + v_4 \leq R \qquad 2.1$$

olmaktadır. Her kod aralık değeri kendinden sonra gelecek bayt sayısını da belli etmektedir. Her aralığa belli sayıda bayt içeren kod kelimesi düşmektedir.

Örneğin; şekil 2.9'da R=4 ve n=11 ve (v_1, v_2, v_3)=(2,1,1). 11 ve 21 arası semboller kullanılmamaktadır.



Şekil 2.9 Örnek RPBC kodlama

Şekil 2.9'da R=4 için hesaplanmış ve n=11 sembollü bir katar ve karşılık gelen kod kelimeleri mevcuttur. Bu kodda, her kod kelimesindeki ilk iki bitlik birim, sonekteki iki bitlik birimlerin sayısını vermektedir. İki sembol bir bayt uzunlukta (v_1=2) kod kelimesine, dört sembol iki bayt uzunluğa (v_2=1) ve kalan beş sembol de 3 bayt uzunluğa sahiptir. Hali hazırda kodlanabilecek 11 birim boş kalmıştır [Culpepper ve Moffat,2005].

Örnek 2.8: T=Bim comes. Ben comes. Bim brings Ben broom. Ben brings Bim broom.

Örnek 2.8'deki cümlede R=8 olarak yani 3 bitle kodlanmaktır. Burada bir metnin n=5 sembollü katarı kullanılmıştır. Örneğin basitliği açısından noktalama işaretleri ve boşluk karakterleri göz ardı edilmiştir. v_1, v_2, v_3 ve v_4 4 eşit kod aralığını göstermektedir. Dizinin indeks değerleri rastgele seçilmiştir.

Sözlüğün sıralanmış halinde katar kelimeleri:

D[0]="Ben"

D[1]="comes"

D[11]="Bim"

D[23]="brings"

D[250]="broom"

Bu durumda kod aralığı dört eşit parçaya ayrıldığı için v=8/4=2'dir. Bu da ilk iki değer tek baytla simgelenir anlamına gelmektedir. İlk iki sözlük elemanın kodlanmış hali;

"Ben"=000

"comes"=001

Sonraki v aralığını (v_2) bir bayt takip edecek ve dolayısıyla iki bayt ile kodlanacaktır. Böyle bir kodlama 2x8=16 olası kodlamaya izin verecektir. Böylece:

"Bim"=010 111

Bir diğer v aralığı olan v_3'ü iki bay temsil edeceği için v_3=2x8x8=128 kodlamaya izin verecektir. Bu durumda:

"brings"=100 000 101

Son v aralığı v_4 ise en az kullanılan sözlük elemanlarını içerecektir. v_4=2 olacaktır ve onu takip eden 3 baytın her biri için mümkün 8 kodlama olacağından toplamda 2x8x8x8=1024 olası kodlama demektir.

" broom"=110 001 101 000

Sonuç olarak yukarıdaki örnekte v değerleri eşit (v=2) kabul edileceği için sözlüğün ilk 2 elemanı 1 baytla, sonraki 16 elemanı 2 baytla, sonraki 128 elemanı 3 baytla ve son 1024 elemanı da 4 baytla kodlanabilmektedir.

2.2. Sözlük Tabanlı Teknikler

Sözlük tabanlı teknikler, bir metinde sıkça yinelenen kalıpların belirlenmesi ve bu kalıplardan bir sözlük oluşturularak, her kalıbın sözlükteki sıra numarasının kodlanmasına dayalı tekniklerdir. Sözlük tabanlı teknikler 3 sınıfa ayrılmaktadır [Culpepper ve Moffat,2005].

- Statik Sözlük Yaklaşımı
- Yarı-Statik Sözlük Yaklaşımı
- Dinamik (Uyarlanır) Sözlük Yaklaşımı

Bu sınıflardan ilki olan statik sözlük yaklaşımında sıkıştırılacak veriler daha önceden var olan bir sözlük yardımı ile sıkıştırılmaktadır. Yarı-statik yaklaşımda metinden sözlük oluşturmak ve oluşturulan sözlük kullanılarak metni sıkıştırmak üzere

sıkıştırılacak metin üzerinde iki geçiş yapılmaktadır. Üçüncü sınıf olan dinamik yaklaşımda ise sıkıştırılacak veri üzerinde sadece bir geçiş yapılmaktadır. Bu geçişte hem sözlük oluşturulur hem de sıkıştırma işlemi gerçekleştirilir.

2.2.1. Statik sözlük yaklaşımı

Bu yaklaşımda sıkıştırılacak her veri için aynı sözlük kullanılmaktadır. Daha çok ASCII standardında kodlanmış metin veriler için kullanılmaktadır. Oluşturulan sözlük içerisinde genelde en çok kullanılan ikili, üçlü, dörtlü karakter grupları gibi gruplar mevcuttur. Bu sözlük elemanlarına da ASCII kodları içerisinde yer alan ama kullanılmayan karakterler atanmıştır.

Sözlük standart sözlük olduğu için sıkıştırılmış dosya içerisinde yer almaz. Sözlüğün sıkıştırılmış dosyada yer almaması sıkıştırma oranında önemli ölçüde kazanç sağlasa da her veri için aynı sözlük kullanıldığından ya da başka bir deyişle metne göre sözlük oluşturulmadığından sıkıştırma oranı dinamik algoritmalar kadar yüksek değildir.

2.2.2. Yarı-statik sözlük yaklaşımı

Bu yaklaşımda öncelikle sıkıştırılacak veri üzerinde veri hakkında bilgi toplamak ve sözlük oluşturmak için bir geçiş yapılmaktadır. İlk geçişte kullanılan algoritmanın özelliğine göre karakterler ya da karakter katarları tespit edilir. Bu katarlar daha kısa kodlarla ifade edilmek üzere sözlüğe yerleştirilir. Daha sonra metin sıkıştırılırken katarların sözlük karşılıkları kodlanır.

Açma aşamasında ise kod çözücünün sıkıştırılmış metindeki sözlük karşılıkları olarak kodlanmış kod kelimelerinin hangi katarlara karşılık geldiğini bilmesi gerekmektedir. Bunun için de kodlayıcı, kodlama aşamasında sözlükleri de sıkıştırılmış dosyaya kod çözücünün işine yarayacak şekilde eklemek zorundadır.

Yarı-statik sıkıştırma teknikleri kullanılan tekniklerde veriye özgü sözlük kullandığı için sıkıştırma oranı statik sözlük yaklaşımı kullanan tekniklerden daha iyidir. Ancak bu tekniklerde de kod çözücünün öğrenmesi gereken sözlüğün masrafı vardır.

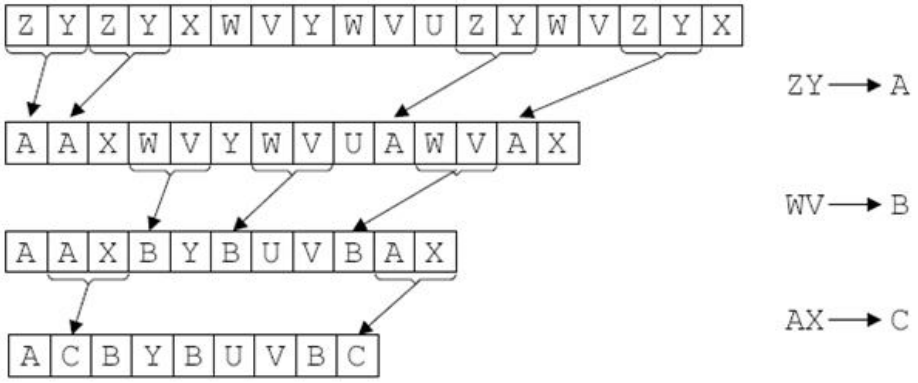
Ayrıca yarı-statik teknikler birden fazla geçiş yaptığı, sözlükler oluşturduğu için de statik tekniklerden sıkıştırma zamanı açısından daha yavaştır.

2.2.2.1. Bayt çiftleri (ikilileri) kodlama

Bayt çifti kodlama dizgi yer değiştirmesine dayanan basit bir sıkıştırma yöntemidir. Metinde en çok geçen ardışık ikilileri metinde geçmemiş olan bir karakterle kodlama esasına dayanmaktadır. Bu işlem 256 mümkün karakter de kullanılana ya da daha fazla sık geçen çift kalmayana kadar devam eder.

Her ardışık iki baytın yer değiştirmesinde, bir anahtar tablosu (yer değiştirme tablosu) yeni yer değiştiren çift eklenerek güncellenir. Bu yer değiştirme tablosu açma işlemi esnasında da gereklidir. Bu sebeple sıkıştırılmış dosya yer değiştirme tablosu ve sıkıştırılmış veri olarak iki parçadan oluşmaktadır.

Bu algoritma çok geçişlidir. Geçiş sayısı sınırlandırılmamıştır ve metin boyuna bağlıdır. Eğer bütün metin bellekte tutuluyorsa algoritma oldukça hızlı çalışmaktadır. Bu teknik, geniş dosyalarda kullanılmayan bayt sayısı az olduğundan, sıkıştırma işlemi istenilen sonucu üretmeden bitecektir. Şekil 2.10'da BPE'de sıkıştırma işlemi adımları gösterilmiştir.



Şekil 2.10 BPE'de sıkıştırma işlemi

Bu iki problem (bellek kullanımı ve az sayıda kullanılmayan bayt değeri) metni bloklara ayırıp, her bloğu ayrı ayrı sıkıştırarak giderilebilir. Bu da yer değiştirme tablosunun her blok için özgün olması demektir. Bunun sonucu olarak da sıkıştırma oranı düşecektir.

BPE'nin sağladığı 3 avantaj:

1- Çok düşük açma zamanı maliyeti,
2- Kısmi açma imkânı,
3- Bayt tabanlı olmasıdır.

Bu avantajlarda BPE'nin sıkıştırılmış dizgi eşlemede önemli bir rol oynamasını sağlamıştır. Bunun yanında sahip olduğu 2 dezavantaj da:

1- Çok yavaş sıkıştırma,
2- Düşük sıkıştırma oranıdır.

Bu problemleri aşmak için bazı iyileştirmeler tasarlanmıştır. Bu iyileştirmelerden biri çiftler tablosunu ilk blok için oluşturup, metnin tamamında kullanmaktır. Hızı oldukça artırmaktadır. Bir diğer yöntem de algoritmayı kelime tabanlı hale getirmektir [Martinez,2005].

2.2.3. Dinamik sıkıştırma yaklaşımı

Dinamik sıkıştırma yaklaşımları veriye özel sözlük kullanmaktadır. Sıkıştırma ve sözlük oluşturma tek geçişte gerçekleşmektedir. Sözlük sıkıştırılacak veriye özgü olduğu için sıkıştırma oranları oldukça yüksektir.

Bunun yanında sözlük sıkıştırma esnasında oluşturulduğu için verinin sıkıştırma adımlarında farklılıklar gösterir. Verinin başında bir karakter kümesinin oluşturduğu kod kümesi ile verinin sonlarına doğru gidildikçe aynı karakter kümesi tarafından oluşturulan kod kelimesi farklılık göstermektedir. Bu da sabit bir sözlüğün olmaması anlamına gelmektedir.

Sabit bir sözlük olmadığı için de kodlayıcı sıkıştırılmış dosyaya herhangi bir sözlük eklemez. Dolayısıyla sıkıştırılmış dosyada sözlük masrafı da yoktur ve kod çözücü kod kelimelerini işledikçe aynı sözlüğü açma esnasında oluşturmaktadır.

Dinamik sıkıştırma yaklaşımlarının bilinen en iyi örnekleri LZ ailesidir.

2.2.3.1. LZ77 sıkıştırma algoritması

LZ1 olarak da adlandırılan LZ77 sıkıştırma algoritmasının temel mantığı mevcut okunmuş olan metin parçasını sözlük olarak kullanmaktır. Kodlayıcı girdi katarına, içinde mevcut okunmuş veri olan bir pencere koyar ve semboller kodlandıkça bu pencere içerisindeki girdi metni sağdan sola kaydırılır. Bu nedenle bu yöntem *kayan pencere yöntemi* olarak da bilinir. Aşağıdaki örnekte pencere iki parçaya ayrılmıştır. Soldaki parça *arama tamponu* olarak adlandırılır. Bu soldaki pencere sözlük olarak kullanılan mevcut okunmuş metin parçasıdır. Sağdaki pencere ise *ileri tampon* olarak adlandırılmıştır ve kodlanacak veriyi içermektedir. Pratikte arama tamponu bin bayttan büyüktür. Buna karşı ileri tampon ancak onlar seviyesinde bayt büyüklüğündedir.

←kodlanmış metin …sir_sid_eastman_easily_t eases_sea_sick_seals…←okunacak metin

İleri tampondaki ilk “e” karakteri ile eşleşme bulmak amaçlı olarak arama tamponu geriye doğru (sağdan sola) aramaya başlanır. “easily” kelimesinin başında “e” karakteri ile bir eşleşme bulunur. Bu eşleşmenin arama tamponu sonuna uzaklığı 8’dir. Daha sonra iki tamponda da yer alan “e”yi de takip eden mümkün olan en fazla sayıda eşleşen karakter bulunmaya çalışılır. Bu durumda 3 sembol (eas) eşleşir ve dolayısıyla eşleşme uzunluğu 3’tür. Daha sonra geriye doğru aramalara daha uzun bir eşleşme bulmak için devam edilir. Bu adımda 16 karakter uzaklıkta “eastman” kelimesi içerisinde, aynı uzunlukta bir benzerlik daha vardır. En uzun eşleşme seçilir. Eşleşmelerin aynı boyda olması durumunda da en son bulunan için (16, 3, “e”) kodu hazırlanır.

İlk rastlanılan eşleşmeden ziyade son rastlanılanın izinin tutması kodlama işlemini basitleştirmektedir. Ancak ilk eşleşilenin tutulması ve programı daha karşık hale getirmenin de bazı avantajları vardır. Bu durumda en yakın uzaklık tutulmuş olur. Kod en uzun mesafeyi tutabilecek kadar yere sahip olduğu için bir avantaj olarak görülmese de LZ77’nin küçük adreslere kısa kod atayabilen bir Huffman veya herhangi bir istatistiksel metot ile takip edildiği durumlarda avantaj sağlamaktadır. Bu metot Bernd Herd tarafından sunulmuş ve LZH olarak adlandırılmıştır. Kısa uzaklıklar tutmak LZH’de daha iyi sıkıştırma oranı sağlamaktadır.

Bu durumda bir soru ortaya çıkmaktadır. Kod çözücü kodlayıcının pencereler arasındaki ilk eşleşmeyi mi yoksa son eşleşmeyi mi kodladığını açma esnasında nereden bilecektir? Doğru cevap kod çözücünün bunu bilemeyeceği ancak bilmesine de gerek olmadığıdır. Kod çözücü basit olarak üretilen kodu okumaktadır. Belirtilen uzaklıktaki eşleşmeler alınır ve bu eşleşmelerin ilk eşleşme mi son eşleşme mi olduğu sorusu göz ardı edilir.

Genelde LZ77 kod kelimesi üç parçadan oluşmaktadır: uzaklık, uzunluk ve ileri tampondaki bir sonraki sembol. Bu kod kelimesi çıktı katarına yazılır ve pencereler eşleşen katar için üç karakter, bir sonraki karakter için de bir karakter olmak üzere 4 karakter sağa kaydırılır. Örnek 2.9’da örnek bir metin için LZ77 sıkıştırma algoritması pencereleri gösterilmiştir.

Örnek 2.9:… sir_sid_eastman_easily_tease s_sea_sick_seals …

Eğer geriye doğru arama sonucunda bir eşleşme bulunamaz ise, LZ77 kodlayıcısı 0 uzaklıklı, 0 uzunluklu ve eşleşmeyen sembolün yazıldığı bir kod kelimesi üretir. Bu da bir kod kelimesinin neden 3 parametre almak zorunda olduğunu açıklamaktadır. 0 uzunluklu ve 0 uzaklıklı kod kelimesi arama tamponun boş olduğu sıkıştırma işleminin ilk adımlarında sıkça üretilen bir çıktıdır. LZ77 sıkıştırma algoritmasının verilen örnek metnin ilk beş adımında aşağıdaki çıktı değerleri üretilmektedir.

	sir_sid_eastman_	→ (0,0,"s")
s	ir_sid_eastman_e	→ (0,0,"i")
si	r_sid_eastman_ea	→ (0,0,"r")
sir	_sid_eastman_eas	→ (0,0,"_")
sir_	sid_eastman_easi	→ (4,2,"d")

Bir sonraki adımda boşluk karakteri eşlenir ve "_e" katarı kodlanır.

sir_sid	_eastman_easily_	→ (4,1,"e")
sir_sid_e	astman_easily_te	→ (0,0,"a")

Buradan da açık olarak görüldüğü gibi tek bir sembolü ifade eden (0,0,..) gibi bir kod iyi bir sıkıştırma sağlamamaktadır. Bu kod kelimesinin kodlama maliyeti kolaylıkla hesaplanabilmektedir. Uzaklığın boyu $[\log_2 S]$'tir. Burada S arama tamponunun uzunluğudur. Pratikte, arama tamponu binlerce bayt uzunluğunda olabilir. Bu nedenle genelde uzaklık 10 – 12 bit şeklinde kodlanır. İkinci parametre olan uzunluğun boyu da benzer olarak $[\log_2(L-1)]$ olur. Burada L ileri tampon büyüklüğüdür. Pratikte ise ileri tamponun uzunluğu onlarla ifade edilen bayt boyundadır. Bu nedenle kapladığı alan sadece birkaç bittir. Sembol parametresinin uzunluğu da genel olarak 8 bittir. Ancak pratikte $[\log_2 A]$'dır. Burada A alfabe boyudur. Bir kod kelimesinin toplam uzunluğu

11+5+8=24

bit olmaktadır. Bu da tek bir 8 bitlik karakteri kodlamak için oldukça büyüktür.

Örnek 2.10'da "uzunluk" parametresinin ileri tampon boyundan neden büyük olabileceği gösterilmiştir.

Örnek 2.10:…Mr._ alf_eastman_easily_grows_alf alfa_in_his_ garden…

İleri tampondaki ilk "a" karakterini arama tamponunda eşleyen 5 tane "a" karakteri vardır. Ancak bu 5 eşleşmenin 2'sinde en uzun eşleşme sayısı olan 3 bulunmuştur. Kodlayıcı son bulduğu (en sol) eşleşmeyi (28,3,"a") kod kelimesi ile kodlayacaktır. Ancak, (3,4,"_") kod kelimesini oluşturur. İleri tampondaki 4 karakterlik "alfa" katarı, arama tamponunun son 3 karakteri ile ileri tamponun ilk karakteriyle eşleşme göstermektedir. Bunun sebebi kod çözücünün bu kodu doğrudan değiştirmeden ele almasıdır. Arama tamponunun 3. konumundan başlar ve tamponu sağa kaydırmak suretiyle bir sonraki 4 sembolü kopyalar. İlk üçü eski tampon içeriğinden, biri de bu yeni kodlanan üçlünün birincisinden olur. Aşağıdaki örnek durumu daha güzel açıklamaktadır.

… alf_eastman_easily_yells_A AAAAAAAAAA AAAAAH…

Kodlayıcı, ileri tampondaki 9 "A"yı, 10. A ile birlikte (1,9,A) şeklinde kodlar. Bunun nedeni eşleşmenin ileri tampon boyundan bir eksik kodlanabilmesidir.

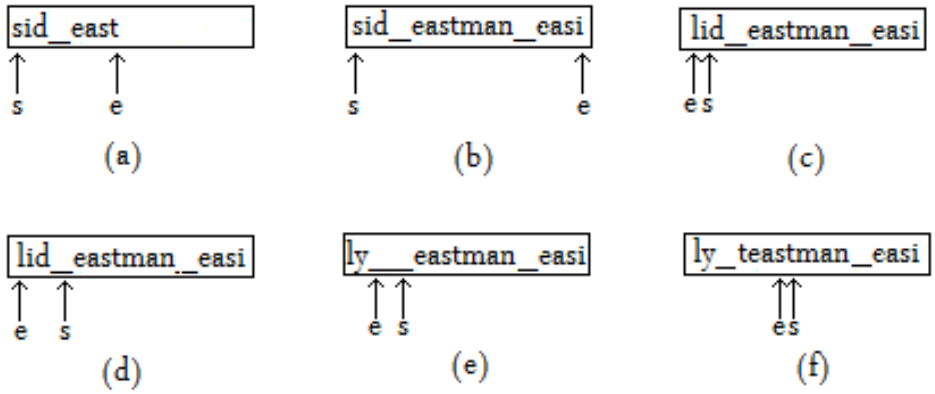
Kod çözücü, kodlayıcıdan daha basit bir mantıkla çalışmaktadır. Kodlayıcı penceresine eşit boyda bir tampon oluşturulur. Bir kod kelimesi ele alınır, tamponda eşleşme bulunr, eşleşme yazılır, üçüncü parametre sonuna eklenir, eşleşmiş katar ve üçüncü parametre tampona kaydırılır. Bu da LZ77 sıkıştırma algoritması veya türevlerinin bir kere sıkıştırılıp, çok defa açılması gereken dosyalarda kullanımının daha yararlı olduğunu göstermektedir.

İlk bakışta bu yöntemin girdi verisi hakkında herhangi bir varsayım yapmadığı görülür. Özellikle de sembol frekansları ile hiçbir işlem yapmamaktadır. Bu da LZ77 sıkıştırma algoritmasının sadece ileri tamponla girdi metninin birbirleri arasındaki eşleşme durumlarını incelemesi mantığına dayanmasının bir nedenidir. Daha önce

girilmiş yani tampon dışına kaymış metinle bir işlem yapılmamaktadır. Bu yöntem girdi metnindeki dizgilerin birbirlerine yakın olduğu varsayımını kullanır. Bu varsayımı doğrulayan verilerde sıkıştırma oranı yüksektir.

Temel LZ77 yöntemi üzerinde 1980'ler ve 1990'larda programcılar ve araştırmacılar tarafından birçok geliştirme ve iyileştirme gerçekleştirilmiştir. Bu iyileştirmelerden biri, değişken uzunluklu "uzaklık" ve "uzunluk" parametresi kullanmaktır. Başka bir iyileştirme de tampon boyutlarını artırmaktır. Bu yöntem daha iyi eşleşmeler bulsa da tamponlar içerisinde eşleşme olan alt katarları arama zamanını artırmaktadır. Geniş arama tamponu daha ileri seviye bir veri yapısını gerektirmektedir. Üçüncü geliştirme de kayan pencere kullanmaktır. En basit yaklaşım tüm metni eşleşme buldukça sola kaydırmaktır. Bundan daha hızlı bir yöntem doğrusal tamponu dairesel tamponla değiştirmektir. Kayan pencere mantığı iki işaretçiyi yer değiştirmekle gerçekleştirilmektedir. Başka bir yol da her kod kelimesi için ilave bit kullanmaktır. Bu üçüncü parametreyi elemek için işaret olarak kullanılır.

Dairesel tampon kullanılan teknikte yer alan ve tampon olarak kullanılan dairesel kuyruk temel bir veri yapısıdır. Dairesel kuyruk fiziksel olarak farklı şekilde kullanılan bir dizidir. Şekil 2.11 bir dairesel kuyruk örneğini göstermektedir. Burada sona ekleyip baştan silme yapan 16 baytlık bir dizi kullanılmıştır. *s* ve *e* işaretçileri her ilerlemede başlangıç ve sonu yeniden işaretlemektedir. (a)'da tamponda sadece 8 karakter vardır ve tamponun geri kalanı boştur. (b)'de 16 bayt da doludur ve *e* sonu göstermektedir. (c)'de ilk "s" karakteri silinerek "l" karakteri eklenmiştir. Burada dikkat edilmesi gereken şey *e* işaretçisinin *s* işaretçisinin soluna konumlanmış olmasıdır. (d)'de "id" ikilisi *s* işaretçisini kaydırmak suretiyle silinmiştir. Aslında hala dizide yer aldığı gözükmektedir. Ama işaretçi yer değiştirdiği için mantiksal olarak silinmiş olur. (e)'de "y_" ikilisi eklenmiştir ve *e* işaretçisi yer değiştirmiştir. (f)'de tampon "teas" ile bitip "tman" ile başlamaktadır. Dairesel kuyruğa yeni semboller ekleyerek işaretçileri yer değiştirmek, kuyruk içerisindeki veriyi kaydırmak demektir. Aslında tam olarak bir kaydırma yapılmamaktadır. [Salomon,2004]



Şekil 2.11 Dairesel kuyruk yapısı

2.2.3.2. LZ78 sıkıştırma algoritması

LZ77 sıkıştırma algoritmasının zayıf yanı küçük bir pencere kullanmasıdır. Bu da metnin ilerleyen kısımlarında değerli eşleşmelerin göz ardı edileceği anlamını taşımaktadır. Örneğin bir kitabın ilk bölümlerinde geçen 10 karakterlik bir bölüm, kitabın son bölümlerinde de geçiyorsa ve aradaki mesafe eğer arama tamponunun boyutundan büyük ise eşleşme bulunamayacaktır. Ancak eğer metindeki o 10 karakterlik katar bir sözlükte yer almış olursa eşleşmeler gözden kaçmayacak ve daha güzel sıkıştırma oranları elde edilmiş olacaktır. LZ77 sıkıştırma algoritmasının temel mantığı daha çok görüleni bulmaktan ziyade, son geçenleri bulmaktır.

LZ77 sıkıştırma algoritmasındaki ikinci zafiyet dizginin kısıtlı bir bölümünün eşleşme olarak nitelendirilebileceğidir. En uzun eşleşme ancak ileri tamponun boyuna yaklaşık olarak eşittir.

Bu problemleri aşmanın bir yolu, tampon boyutlarını büyütmek olsa da zaman açısından oldukça yüksek bir maliyeti vardır.

Ziv ve Lempel bu problemlerle karşılaştıkları için bir başka sözlük tabanlı sıkıştırma yöntemi geliştirdiler. Buna LZ78 adını vererek “Compression of Individual Sequences via Variable-Rate Coding” isimli makalelerinde 1978 yılında yayınladılar.

LZ78 sıkıştırma algoritması, LZ77 sıkıştırma algoritmasının pencere yaklaşımından sıyrılıp, potansiyel olarak sınırsız ve daha önceden geçmiş olan katarlardan oluşan bir sözlük yaklaşımı benimsemiştir. LZ77 sıkıştırma algoritmasına benzer olarak aynı anlama gelen parametreler içeren kod kelimeleri üretir. LZ78 sıkıştırma algoritmasının ürettiği kod kelimesi bir katara karşılık gelen kod ve bir karakter içeren bir kod kelimesine sahiptir. LZ77 sıkıştırma algoritmasından farklı olarak katar uzunluğu parametre olarak verilmez. Çünkü kod çözücü bu uzunluğu bilmektedir.

LZ78 sıkıştırma algoritmasında hem kodlayıcı hem de kod çözücü neredeyse boş bir sözlükle işe başlamaktadır. Sözlükte yer alan tek eleman *NULL* karakteridir. Her karakter okunduğunda, eldeki katara eklenir. Bu işlem katar sözlükte bir eşleşme gerçekleştirdiği sürece devam eder.

Eşleşme gerçekleşmediği anda sözlüğe ekleme işlemi yapılır. Son karakter okunana kadar sözlükte eşleşme gerçekleştirilmiştir. Son karakter okunduğunda bu eşleşme durumu ortadan kalkmıştır. İşlenen eldeki katar son eşleşen katar ve bir ilave karakter olarak ifade edilir. Bu da LZ77 sıkıştırma algoritmasının ürettiği çıktıdır.

Ancak bu noktada LZ78 sıkıştırma algoritması ilave bir adım kullanmaktadır. Yeni karakter içeren yeni katar sözlüğe eklenmektedir. Katar bir daha geçtiğinde daha uzun bir eşleşme ortaya çıkabilecektir.

Boş katar her zaman 0 indeksli sözlük elemanı ile eşleşecektir. Bu sebeple bir karakterle ilk defa karşılaşıldığında, 0 katarı artı bir karakter olarak kodlanacaktır. Karakter bir daha görüldüğünde katar parçası olarak değerlendirilecektir.

Kodlayıcı çıktısının bir örneği çizelge 2.8'de verilmiştir. LZ78 sıkıştırma algoritması neredeyse boş sözlükle işe başlar, bu sebeple ilk okunan "D" karakteri sözlükte herhangi bir eşleşemeye uymaz. Bu durumda kodlayıcı katar/karakter çifti olarak 0 ve "D" kodlar. Burada 0 indeksi boş katara ayrılmıştır.

Girdi Metni: DAD_DADA_DADDY_DADO…

Çizelge 2.8 Örnek metin için üretilen LZ78 çıktıları

Çıktı Katarı	Çıktı Karakteri	Kodlanan Katar
0	D	D
0	A	A
1	_	D_
1	A	DA
4	_	DA_
4	D	DAD
1	Y	DY
0	_	_
6	O	DADO

İlk İki karakter olan “D” ve “A” daha önceden görülmemişlerdir. Bu sebeple *0+karakter* çifti olarak kodlanacaktır. 1 numaralı indekse “D”, 2 numaralı indekse de “A” ilave edilir. Üçüncü karakter olan “D” okunduğunda sözlükte yer aldığı görülmektedir. Sonraki karakter olan “_” karakteri de okunur ve katara eklenir. “D_” sözlükte yer almadığı için üretilen kod, eşleşen “D” karakterinin indeksi ve “_” karakterini içeren *(1,”_”)* olur.

Kodlama devam ettikçe sözlük oluşur ve oldukça büyük parçaları içermeye başlar. Sözlükteki girdiler alfabetik sırada olacağı için girdiler oldukça hızlı olarak üretilebilir. Örnek metindeki 19 karakter okunup kodlandığında sözlük aşağıdaki gibi olur.

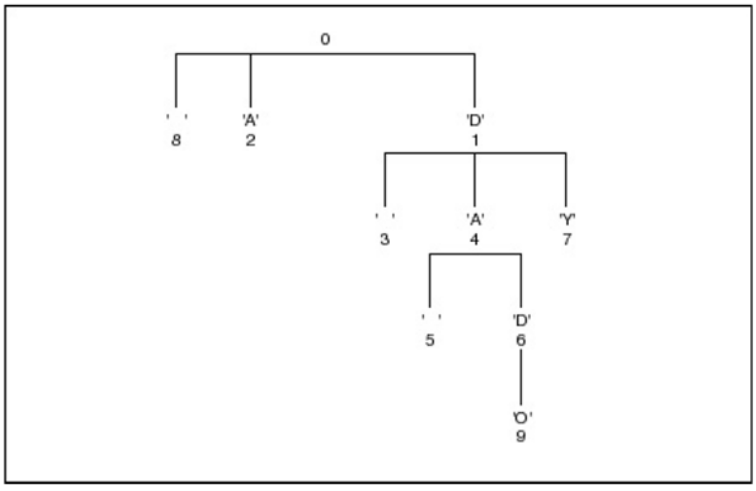
0“”
1“D”
2“A”
3“D_”
4“DA”
5“DA_”
6“DAD”
7“DY”
8“_”
9“DADO”

LZ77 sıkıştırma algoritmasında olduğu gibi LZ78 sıkıştırma algoritmasında da sözlük boyutu isteğe bağlı olarak seçilebilir. Ancak sözlük boyutunu seçerken dikkat edilmesi gereken iki husus mevcuttur. İlk olarak, kod kelimesi için kullanılacak olan bit sayısı düşünülmelidir. İkinci ve daha önemli olarak da sözlüğü yönetmek için gerekli olan CPU zamanı düşünülmelidir.

Teoride LZ78 sıkıştırma algoritması, sözlük boyutu büyüdükçe daha güzel oranlarda sıkıştırma yapmaktadır. Ancak bu durum sadece girdi metninin boyutu sonsuza yaklaşırken yani oldukça büyük olması durumunda gerçekleşmektedir. Büyük kod kelimeleri içeren küçük boyutlu metinler aynı derecede kötü etkilenmektedir.

LZ78 sıkıştırma algoritması ile ilgili olarak gerçek problem sözlüğü yönetmektir. Eğer sözlük indeksi olarak 16 bit uzunluklu kod kullanılırsa, *NULL* kodu da dâhil olmak üzere 65535 katar sözlükte yer alabilir. Katarlar son derece uzun olabilir. Hatta bir katarın mümkün 65536 alt katarını içerebilir.

Bu katarlar ağaçlar üzerinde tutulur. Ağaç kök nodunda *NULL* katarını tutmaktadır. *NULL* katarına eklenen her yeni karakter ağacın yeni bir dalıdır. LZ78 sıkıştırma algoritması için örnek bir ağaç şekil 2.12'de verilmiştir.



Şekil 2.12 LZ78 sözlük ağacı

Şekil 2.12'de gösterilen ağaç, 19 harflik örnek metin işlendikten sonra oluşmuş olan ağaçtır. Ağacı yönetmenin en büyük zorluğu da her noddan çıkan dallar sayesinde ağacın büyümesidir. 8-bit alfabeli ikili dosyalar sıkıştırılırken, her noddan çıkabilecek

256 mümkün dal mevcuttur. Bunları tutmak için de işaretçiler veya diziler kullanılabilir. Ancak hepsi dolu olamayacağı için oldukça büyük bir yer israfi söz konusudur. Bunun yerine ağaç dalı olarak değil, ağaç dalına referans veren indeksler listesi olarak tutulurlar. Bu belirgin ölçüde problemi çözse de oldukça yavaş bir yöntemdir.

Şekil 2.12'deki gibi bir ağaçta, var olan bir katarla sözlüğü karşılaştırmak son derece kolaydır. Normal olarak ağacı dolaşmak gibi bir işlemdir. Eğer bir ağaç nodunda katar tüketilirse orada bir eşleşme söz konusudur.

LZ78 sıkıştırma algoritmasının LZ77 sıkıştırma algoritmasına göre olumsuz bir yanı da kod çözücünün de aynı ağacı kod çözme esnasında oluşturmasının gerekmesidir. LZ77 sıkıştırma algoritmasında sözlük indeksi kendinden önce geçmiş bir katara işaret etmektedir. Ancak LZ78 sıkıştırma algoritmasında bu indeks bir sözlük ağacı indeksidir. Bu sebepten kodlayıcı gibi kod çözücü de aynı ağacı aynı mantıkla oluşturmalıdır.

Sözlük, boyutuna bağlı olarak kısa zamanda veya daha uzun zamanda dolacaktır. Eğer 16-bit kod kullanılıyorsa, sözlük 65535 katar işlendikten sonra dolacaktır.

Bu problemin üstesinden gelmek için birkaç yöntem mevcuttur. Ama en geçerli yöntem sözlük dolduktan sonra yeni nodlar eklemeyi bitirmektir. [Nelson ve Gailly,1995]

2.2.3.3. LZW sıkıştırma algoritması

LZW sıkıştırma algoritması Terry Welsch tarafindan "A Technique for High – Performance Data Compression" isimli makalede 1984 yılında sunulmuştur. LZW sıkıştırma algoritmasının kullanılmış olduğu en güzel örneklerden biri *UNIX Compress* programıdır.

LZW sıkıştırma algoritması, LZ78 sıkıştırma algoritması üzerindeki çıktı kod kelimesinde yer alan katar ve karakter ikilisine olan bağlılığı ortadan kaldırmaktadır.

LZW sıkıştırma algoritmasında kodlayıcı, katarlara karşılık gelen sözlük elemanlarının indeks değerlerini çıktı olarak üretmektedir.

Bunun için, LZW sıkıştırma algoritmasındaki en önemli değişiklik her sembolün en başta sözlükte yer almasıdır. Bu sebeple daha önce girdi metninde görülmemiş olsa bile karakterler kodlanabilir. LZW sıkıştırma algoritması her zaman bilinen katarlar için kod üretir ve her kod üretildiğinde sözlüğe yeni bir girdi eklenir.

Çizelge 2.9'da algoritmayı göstermek için algoritmanın bir metin üzerinde uygulanması gösterilmiştir. İlk geçişte "_W" katarının sözlükte olup olmadığı aranır. Sözlükte yer almadığı için "_" karakteri için kod çıktı olarak üretilir ve "_W" sözlüğe eklenir. Sözlükte 0 – 255 arası indeksler tek karakterlik sembollere ayrıldığı için sözlüğe ekleme 256 numaralı indeksten başlar. Üçüncü karakter "E" okunduğunda "WE" katarı sözlükte aranır ve çıktı olarak "W" üretilir. Aynı zamanda da "WE" katarı sözlüğe eklenir. İkinci kelimede "_W" katarı sözlüğün 256 numaralı indeksindeki elemanda bulunur. 256 değeri çıktı olarak üretilir ve "WE_" sözlüğe eklenir. Katar tüketilene kadar bu işlem devam eder.

Girdi Metni: "_WED_WE_WEE_WEB_WET_"

Çizelge 2.9 Girdi metnine göre LZW sıkıştırma algoritması kod çıktıları

Girdi Karakterleri	Çıktı Kodu	Yeni Kod Değeri ve Atanan Katar
_W	"_"	256="_W"
E	"W"	257="WE"
D	"E"	258="ED"
_	"D"	259="D_"
"WE"	256	260="_WE"
_	"E"	261="E_"
"WEE"	260	262="_WEE"
"_W"	261	263="E_W"
"EB"	257	264="WEB"
_	"B"	265="B_"
"WET"	260	266="_WET"
<EOF>	T	

Çizelge 2.9'da örnek metin için çıktı katar değerleri gösterilmektedir. Sözlük tablosu çok hızlı dolmaktadır. Çünkü her kod üretildiğinde sözlüğe yeni bir katar

eklenmektedir. Eğer çıktı için 9 bit kullanılırsa, 19 karakterlik katar için üretilen kod 13,5 karaktere düşecektir. Tabi ki yukarıdaki örnek, katar yer değiştirmelerini göstermek için seçilmiş özel bir örnektir. Normal metinlerde genellikle sıkıştırma oldukça büyük bir tablo oluşturulmadan başlamaz. Bu tablo da yüz civarı bayt okunmadan oluşturulamaz.

Açma algoritması içinde sıkıştırma algoritması mantığının hemen hemen aynısı kullanılmaktadır. Sıkıştırılmış dosyadan kod okunur ve orijinal metni elde etmek için bu kod kullanılır. LZW sıkıştırma algoritmasının en büyük artısı, açma algoritmasına eldeki sıkıştırılmış veriyi açmak için kullanılmak üzere herhangi bir sözlük transferinin gerekmemesidir. Girdi metni kullanılarak aynen sıkıştırma algoritmasında olduğu gibi sözlük oluşturulur. Sıkıştırma algoritması katarı ve bir kodun karakter içeriklerini kodu üretmeden önce verdiği için böyle bir işlem mümkündür. Bu sebeple de sıkıştırılmış metin içerisinde oldukça büyük yer tutan sözlük yer almaz.

Çzielge 2.10'da LZW sıkıştırma algoritması tarafından üretilen bir kod bulunmaktadır. Açma işlemi bittikten sonra elde edilen sözlük, sıkıştırma işlemi yapıldıktan sonra elde edilen sözlük ile aynıdır. Çıktı olarak elde edilen katar da, sıkıştırma işleminden önceki girdi metniyle aynıdır. Aynı zamanda ilk 256 kod tek karakterlik katarlara ayrılmıştır.

Girdi Metni: "_WED<256>E<260><261><257>B<260>T"

Çizelge 2.10 Sıkıştırılmış girdi metninin açılması sonrası elde edilen sözlük

Girdi/Yeni Kod	Eski Kod	Katar/Çıktı	Karakter	Yeni Tablo Girişi
"_"	""	"_"		
"W"	"_"	"W"	"W"	256="_W"
"E"	"W"	"E"	"E"	257="WE"
"D"	"E"	"D"	"D"	258="ED"
256	"D"	"_W"	"_"	259="D_"
"E"	256	"E"	"E"	260="_WE"
260	"E"	"_WE"	"_"	261="E_"
261	260	"E_"	"E"	262="_WEE"
257	261	"WE"	"W"	263="E_W"
"B"	257	"B"	"B"	264="WEB"
260	"B"	"_WE"	"_"	265=B_
"T"	260	"T"	"T"	266="_WET"

LZ78 sıkıştırma algoritmasındaki gibi LZW sıkıştırma algoritması sözlüğü de çok yollu ağaç şeklinde elde edilmektedir. Fakat LZW sıkıştırma algoritmasında depolanan sözlük verisi bir ağaç yapısında değildir. Bu ağaç yapısı bir sözlük veri yapısı altında tutulmaktadır.

```
struct sozluk{
  int kod_deger;
  int onceki_katar_kod;
  char karakter;
} Sozluk[Sozluk_Boyu];
```

Yukarıdaki kod parçası tüm sözlüğü tutmak için kullanılır. Veri yapısındaki her eleman bir nodu temsil eder. Nod üç nesne ile tanımlanır.

1- **kod_deger:** Sözlüğe eklenen katar için elde edilen değerdir. Kodlayıcı bundan sonra katarla karşılaştığında bu değeri kodlayacaktır.

2- **onceki_katar_kod:** LZ78 sıkıştırma algoritmasında, sözlükteki her katar kendinden bir karakter eksik başka bir katardan türetilmiştir. Bu alan da kataın türetildiği alt kataın indeksini tutulmaktadır.

3- **karakter:** Bu nod için belirli karakterdir. Örneğin eğer katar, daha önceki katar "TRAKY" ile kodlanıyorsa, karakter değeri "A" olacaktır. Bu nod da "TRAKYA"yı kodlayacaktır.

Burada ilk başta bir problem göze çarpmaktadır. O da hiçbir sözlük elemanının kendisinden sonra gelen elemana işaret eden işaretçiler barındırmamasıdır. Eğer işaretçi yoksa bir sözlük elemanından sonra gelene nasıl ulaşılır?

Cevap ağacın sözlük işaretçilerini nodların anahtarlanmış dizileri yardımı ile elde ediyor olmasıdır. Örnek olarak UNIX Compress programında, anahtar fonksiyonu *onceki_katar_kod* değeri ile *çocuk_karakter* değerini birleştirir ve nod listesinde 16-bit uzaklıkta bir değer elde eder. Hedef nod bulunduktan sonra, çakışmalar için kontrol

edilir. Çünkü bu nod ağaçtaki başka elemanlar tarafından da kontrol ediliyor olabilir. Genelde iki durumla karşılaşılır. Ya eldeki kodlara ait bir noda ulaşılır ya da kullanılabilecek boş bir nod bulur. [Nelson ve Gailly,1995]

2.2.3.4. LZSS sıkıştırma algoritması

LZ77 sıkıştırma algoritmasının bu sürümü 1982 yılında Storer ve Szymanski tarafından geliştirilmiştir. LZ77 sıkıştırma algoritması üzerindeki geliştirmeler aşağıdaki 3 işlemin gerçekleştirlmesiyle yapılmaktadır.

1- İleri tamponu bir dairesel kuyruk üzerinde tutmaktadır.
2- Arama tamponunu (sözlüğü) ikili arama ağacında tutmaktadır.
3- 3 yerine 2 parametreli kod kelimesi üretmektedir.

İkili arama ağacı, bir A ağaç nodunun her sol alt ağaç değeri A değerinden küçük ve her sağ alt ağaç değeri de A değerinden büyük olan ağaçtır. Buradaki ikili arama ağacı katarları tuttuğu için öncelikle iki katarın nasıl karşılaştırılacağı ve hangisinin büyük kabul edileceği bilinmelidir. Böyle bir karşılaştırmada kullanılan ölçüt alfabetik sıralamadır. Buna da sözlük sırası adı verilmektedir.

Bunun yanında, boşluk karakterinin ASCII kodu diğer karakterlerde daha küçük olduğu için " abc" katarı diğer katarlardan önce gelmektedir. Genel olarak bilgisayarın kullandığı karakter sırası küçükten büyüğe karakterlerin sıralanmasında kullanılmaktadır.

Örnek 2.11: Buradaki örnekte ikili arama ağacının sözlük aramasını nasıl hızlandıracağı gösterilmiştir. "sid_eastman_clumsily_teases_sea_sick_seals" girdi cümlesi olsun. Örneğin basit olması için pencere 16 bayt arama tamponu ve 5 bayt ileri tampon olmak üzere toplamda 21 bayt olarak kurulmuştur. 16+5 bayt okunduktan sonra kayan pencere;

sid_eastman_clum sily_ teases_sea_sick_seals

halini almaktadır. Burada “teases_sea_sick_seals” katarı işlenmeyi beklemektedir. Kodlayıcı arama tamponunu tarar, tabloya eklemek için on iki tane beş karakterlik katar üretir. Her birinin uzaklıkları çizelge 2.11’de verilmiştir.

Çizelge 2.11 5 karakterlik katarlar

Katar	Uzaklık
sid_e	16
id_ea	15
d_eas	14
_east	13
eastm	12
astma	11
stman	10
tman_	9
man_c	8
an_cl	7
n_clu	6
_clum	5

İleri tampondaki ilk harf “s”dir. Kodlayıcı ağaçta s ile başlayan katar arar. 16 ve 10 uzaklığında iki tane bulunur ve bunlardan ilki (16 karakter uzaklıkta olan) daha uzun bir eşleşme sağlamaktadır.

Örnekte uzunluk 2’dir ve iki parametreli kod kelimesi (16,2) olarak üretilir. Kodlayıcı daha sonra pencereyi 2 karakter kaydırmalıdır ve ağacı güncellemelidir.

si d_eastman_clumsi ly_te ases_sea_sick_seals

“sid_e” ve “id_ea” katarları silinip, “clums” ve “lumsi” katarları eklenmek suretiyle ağaç güncellenir. Eğer *k*-harfli bir katar eşleşmesi olursa, pencere *k* karakter kaydırılacak ve ağaçtan *k* katar silinip *k* yeni katar eklenecektir.

Silinecek *k* katar, kaydırmadan önceki ilk *k* katar ve eklenecek katarlar da kaydırmadan sonra en sonda yer alan *k* katardır. Her güncelleme eşit sayıda katar silip

eklediği için ağaç boyu hiç değişmez. Ancak ağacın şekli belirgin bir biçimde değişecektir.

LZSS'deki bir diğer LZ77 iyileştirmesi de kodlayıcı tarafından üretilen kod kelimesidir. Kod kelimesi sadece uzaklığı ve uzunluğu içermektedir. Eğer herhangi bir eşleşme bulunamazsa, (0,0,..) kod kelimesi yerine bir sonraki sembolü sıkıştırılmamış biçimde yazar. Sıkıştırılmamış kodları, üretilen kod kelimelerinden ayırmak için bir işaret biti kullanılmaktadır.

Arama tamponu binlerce bayt uzunluğunda olabilir ve uzaklık genelde 11-13 bit ile ifade edilmektedir. İleri tamponun uzunluğu da toplam uzunluk 16 bit yani 2 bayt olacak şekilde seçilmelidir. Örneğin arama tamponu 2 Kb ise ($=2^{11}$), ileri tampon 32 bayt ($=2^5$) olmalıdır. Uzaklık alanı 11 bit uzunluğunda ve uzunluk alanı da 5 bit uzunluğunda olmalıdır. Bu şekilde seçilen değerlerle kodlayıcı ya 2 bayt uzunluktaki kod kelimesi ya da 1 bayt uzunluktaki sıkıştırılmamış karakteri kodlayacaktır. İşaret bitlerinin nasıl kullanılacağı ya da neyi ifade edeceği hakkında bir sorun ortaya çıkacaktır. Buna uygun bir çözümde 8 çıktı değerini küçük bir tampona alarak herbiri için bir işaret biti içeren bir baytı, bu 8 çıktının önünde kodlamak olabilmektedir.

LZ77 sıkıştırma algoritması ve türevlerinin bir zayıf yönü, metnin sadece yakın kısımlarını sıkıştırmasıdır. Örneğin metin içerisinde "trakya" kelimesi normal olarak dağılmış olsun. Kelime ileri tampona kaydırıldığı zaman, arama tamponunda yer almıyor ise sıkıştırılamamaktadır.

Bir diğer zaafiyet de ileri tampon büyüklüğüdür. Bu büyüklük L ise en fazla L-1 uzunlukta eşleşme yapılabilmektedir. Eğer tampon büyütülür ise sıkıştırma oranı artacak fakat bunun yanında daha uzun eşleşmeler yapmak ve daha uzun tamponlarda eşleşme aramak kodlayıcının daha fazla zamanını alacaktır.

Frans Willems tarafından 1989 yılında geliştirilen bir yaklaşımda girdi metni ikili sembollerden oluşmaktadır. Bu durumda girdi metni *L* uzunluklu kelimelere ayrılır ve kayan pencere içinde yer alır. Kodlayıcı ileri tampondaki en solda yer alan kelimeyi eşlemeye çalışır. Her adımda sadece bir kelime eşlenir. Eğer bir eşleşme bulunursa, uzaklık 2 parçalı bir önek kodu olarak sıkıştırılmış veriye yazılır. Eğer eşleşme yok ise

özel bir kod üretilir. Bu kod eşleşmeyen kelimenin L sembolü tarafından takip edilir. Örnek 2.12 bu yaklaşıma bir örnek göstermektedir.

Örnek 2.12: *L=3* ve arama tampon uzunluğu $B = 2^L - 1 = 7$ olur ve virgüller kelime sınırlarını belirtmektedir.

← kodlanmış metin … 0100100 100,000,011,111,011,101,001 …←okunacak metin

Bu örnekte, en soldaki "100" kelimesinin arama tamponundaki ilk 3 bitle eşleştiği gözükmektedir. Buradaki uzaklık m=3'tür. Tampon bir kelime sola kaydırılır.

← …010 0100100 000,011,111,011,101,001,… …←okunacak metin

Şu anda işlenen "000" kelimesi için m=1'dir. Çünkü bu kelimedeki her bit bir solundaki konumda eşlenmektedir.

← …010010 0100000 011,111,011,101,001,… …←okunacak metin

"011" kelimesi ile herhangi bir eşleşme yoktur. Eşleşme olmadığı için *m* özel bir değer alır (8*). Bunun anlamı 8'e büyük veya eşit olmasıdır. Her *m, p* ve *q* tamsayılarını belirleyerek kodlanır. Eğer *m=8** ise *p L*'ye eşitlenir, değilse $2^p \leq m < 2^{p+1}$ olacak şekilde *m* değeri seçilir. *p* [0,L-1] aralığında yer alır. *q* tamsayısı ise [0,2^p-1] aralığında yer alan $q = m - 2^p$ olarak tanımalnır. Çizelge 2.12 L=3 için *m,p,q* değerleri ve önekleri göstermektedir.

Çizelge 2.12 L=3 içim m, p, q değerleri ve önek – sonek tablosu

M	p	q	Önek	Sonek	Uzunluk
1	0	0	00	--	2
2	1	0	01	0	3
3	1	1	01	1	3
4	2	0	10	00	4
5	2	1	10	01	4
6	2	2	10	10	4
7	2	3	10	11	4
8*	3	--	11	Kelime	5

p ve *q* değerleri bilindiği zaman m için bir önek kodu oluşturulur ve çıktı katarına yazılır. İki parçadan oluşmaktadır. Bunlar *p* ve *q*'nun ikili değerleri olan önek ve sonektir.

Örnekteki 7 kelimelik girdi metninin sıkıştırılmış karşılığı:

01|1,00,11|011,00,10|10,10|00,11|001,….

Yatay çizgiler önek ve soneki ayırmaktadır. "011" ve "001" kelimesi sıkıştırılmadan bırakılmıştır. [Salomon,2007]

2.2.3.5. Geçiş uzunluğu kodlama sıkıştırma algoritması

Eğer bir *d* nesnesi *n* defa ardışık olarak girdi verisinde yer alıyorsa, verinin bu tekrarlanan kısmı *nd* basit çifti ile yer değiştirir. Bir nesnenin *n* defa ardışık varoluşu *n* nesnesinin geçiş uzunluğu olarak adlandırılır ve *geçiş uzunluğu kodlama* ya da *RLE* olarak isimlendirilir. Her türlü veride kullanılabilmesine rağmen, daha çok görüntü sıkıştırmada kullanılmaktadır. Örnek 2.13'te bir örnek metin için RLE kod kelimesinin nasıl üretildiği adım adım anlatılmıştır.

Örnek 2.13: "2._abbcddaccb" metnini, "2._a2c2a2b" olarak sıkıştırmak doğru bir çözüm değildir. Kod çözücü karşılaştığı ilk "2" karakterinin orijinal metnin bir parçası olduğunu ve diğerlerinin de sırasıyla "b", "d", "c" karakterlerine karşılık geldiğini bilmelidir. Bunun yerine "2._a2bc2da2cb" şeklinde bir sıkıştırma yapmak da (aynı zamanda herhangi bir sıkıştırma da sağlamamaktadır) sorunu çözmeyecektir. Sorunun çözümü için bir yol, tekrarlamaları özel karakterle belirtmektir. Örneğin eğer "@" karakteri kaçış karakteri olarak kullanılırsa, metin "2._a@2bc@2da@2cb" halini almaktadır. Bu şekilde kodlanmış bir metin de eş anlamlılık meydana getirmeyecek şekilde açılabilmektedir. Aslında orijinal metinden de uzundur. Çünkü iki ardışık harf üç karakterle ifade edilmiş olmaktadır. Bu durumda en az üç ardışık harf bu şekilde tekrarlamalı yazılacak şekilde bir uyarlama yapılmak zorunluluğu sıkıştırma oranını eksi yönde ilerletmemek adına ortaya çıkmaktadır.

Sıkıştırma işlemi esnasında, ilk karakter okunduktan sonra sayaç 1 olur ve karakter kaydedilir. Ardışık karakterler kaydedilen karakter ile kıyaslanır. Eğer aynı ise tekrar sayacı artırılır. Farklı bir karakter okunduğu zaman, işlem tekrar sayacının değerine göre değişiklik gösterir. Eğer küçükse, kaydedilen karakter dosyaya yazılır ve yeni okunan karakter kaydedilir. Aksi halde, tekrar sayacı ve kaydedilmiş karakteri tarafından takip edilen bir "@" yazılır.

Açma işlemi de ileriye doğrudur. Bir "@" karakteri okunduğunda, *n* tekrar sayısı ve karakter doğrudan okunur ve karakter *n* defa çıktı katarına yazılır.

Bu metotta karşılaşılabilecek ana problemler:

1- Düz İngilizce metinlerde çok fazla tekrarlamalar yoktur. Birçok ikili tekrar vardır ancak üçlü tekrar çok nadiren görülmektedir. Tire veya yıldız işareti birçok kez tekrar edebilmektedir. Matematiksel metinlerde bazı rakamlar tekrar edebilmektedir.

2- "@" karakteri girdi katarının bir parçası olabilir. Bu durumda başka bir kaçış karakteri seçilmelidir. Ancak bazı durumlarda metin tüm mümkün alfabe karakterini kullanıyor olabilmektedir.

3- Tekrar sayısı çıktı metnine bayt olarak yazıldığı için bu değer en fazla 255 olabilir. Ancak 3 karakterden az olan tekrarlar göz ardı edileceği için 0 değeri ile 3 tekrar yapıldığı anlamı çıkartılırsa, bu durumda tekrar sayısı 255'ten 258'e kadar genişletilebilir.

N karakter içeren bir metnin L uzunluklu M tekrara sahipse, RLE tarafından üretilen sıkıştırılmış dosya boyutu $N - M(L-3)$ olur. Sıkıştırma faktörü de aşağıdaki formülle bulunur.[Salomon,2007]

$$\frac{N}{N - M(L-3)} \qquad 2.2$$

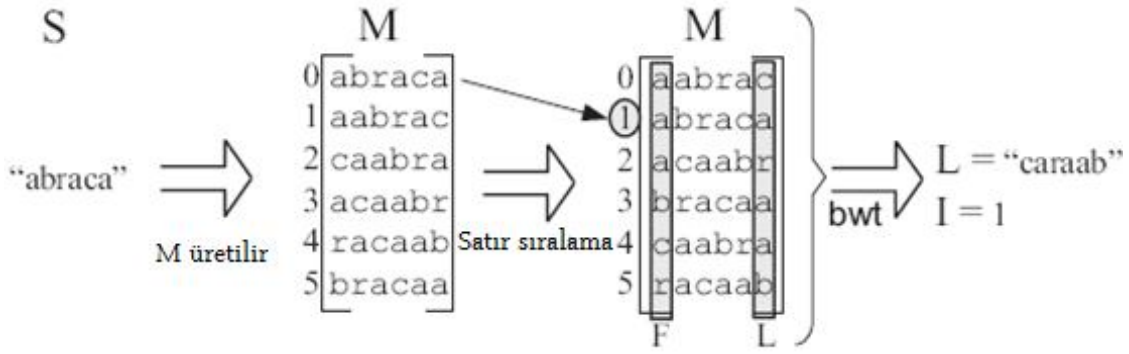
2.3. Dönüşüm Teknikleri

Metin dönüşüm teknikleri, metin içerisindeki veriyi değiştirmeden bir şekilden başka bir şekle dönüştürme yapan tekniklerdir. Metin sıkıştırma konusu içerisinde bu teknikler sıkıştırma yapmak için değil, metni daha iyi sıkıştırılabilecek hale sokmak için kullanılmaktadır. Veri sıkıştırma konusu içerisinde kullanılan en popüler metin dönüşüm tekniklerinden biri *Burrows – Wheeler Dönüşümü* adı verilen tekniktir.

2.3.1. BWT dönüşüm tekniği

Burrows – Wheeler Dönüşümü, Wheeler tarafından 1983 yılında tasarlanmış, ancak 1994 yılına kadar yayınlanmamıştır. Bu teknik bir sıkıştırma tekniği değil, orijinal metni daha iyi sıkışabilecek hale getiren bir dönüşüm tekniğidir. Şekil 2.13'te BWT tekniğine örnek verilmiştir.

BWT ters çevrilebilir bir algoritmadır. Başka bir deyişle BWT dönüşümü uygulanmış bir metnin, işlemler ters çevrilerek uygulandığında orijinali elde edilebilmektedir. Bir *S* katarı BWT dönüşümü uygulandıktan sonra yeni *TS* katarına dönüşmektedir. *TS, S* ile aynı bilgilere sahiptir (ve biraz fazlası) fakat bu bilgiler farklı sıradadır. Ters BWT dönüşümü ise orijinal *S* katarını tekrar elde etmektedir.



Şekil 2.13 Düz BWT dönüşümü

BWT dönüşümü hesaplanılırken, *N* uzunluklu bir *S* katarı ele alınır. Bir M_{NxN} matrisi oluşturulur. Bu matrisin ilk satırında *S* katarı yer alır. İkinci sırada ise $S \gg 1$ (S dairesel olarak bir sağa kaydırılır) katarı bulunmaktadır.

İkinci adımda matrisin tüm satırları alfabetik olarak sıralanır. Sıralama esnasında orijinal konumlarının izleri tutulur. Bu sıralanan satırlardan biri orijinal *S* katarını içermektedir. *I* değişkeni de *S*'yi barındıran bu satırın konumunu tutmaktadır.

Sıralamadan sonra *M* matrisinin ilk sütunu *F*, son sütunu da *L* ile gösterilir. Burada iki özellik göze çarpmaktadır:

1- *F*, *S*'deki tüm karakterleri içermektedir. Ancak bu karakterler alfabetik olarak sıralanmıştır.

2- *L*'deki *j* karakteri, *j* satırındaki katardan önce gelmektedir.

BWTdönüşüm tekniğinde üretilen sonuç *M*'nin *L* sütunundaki karakterler ve *I* değişkeni bileşkesidir. $BWT(S) \rightarrow (L, I)$ olarak ifade edilir. Şekil 2.13'te "abraca" katarı için BWT dönüşümü gösterilmektedir.

Ters BWT (IBWT) dönüşümü hesaplanırken, algoritma BWT dönüşüm tekniğinin (*L, I*) parametresini, BWT tekniği girdisini tekrar oluşturmak için kullanmaktadır. IBWT dönüşüm tekniği 3 aşamadan oluşmaktadır.

Birinci aşamada, *M* matrisinin *F* sütunu tekrar oluşturulur. Bu da *L* katarını alfabetik sıralama ile gerçekleştirilir. Örnek 2.14'te *L="caraab"* olduğu için, sıralanmış hali olan *F="aaabcr"* olur.

İkinci aşamada *F* ve *L*, *T* vektörünü hesaplamak için kullanılır. Bu vektör iki kataın karakter olarak karşılık gelenlerini belirler. Eğer *L[j]*, "c" karakterinin *L*'deki *k* numaralı geçişi ise, *F[i]* "c"nin *F*'deki *k* numaralı geçişini ifade edecek şekilde *T[j]=i* olur. Bu sebeple vektör *F* ve *L* arası karakter karşılık gelişlerini ifade eder.

Örnek 2.14: Çizelge 2.13'te "c"=L[0]'nin ilk geçişi *F*'de 4. konumdadır. Dolayısıyla T[0]=4 olur. Yine "a"=L[1] F[0]'dır. Bu sebeple T[1]=0 olur.

Çizelge 2.13 BWT dönüşüm tablosu

Konum	0	1	2	3	4	5
L=	c	a	r	a	a	b
F=	a	a	a	b	c	r
T=	4	0	5	1	2	3

T'nin tanımından $F[T[j]] = L[j]$ olur. Bu özellik *S* katarını tekrar elde etmeye yaradığı için önemlidir. Örnek 2.15'te *S* katarının tekrar elde edilme adımları gösterilmiştir.

Son aşamada *S* katarı *I* indeksi ve *L* ile *T* vektörleri yardımıyla elde edilir. *S*'yi elde etme işlemi:

$p \leftarrow I$; 2.3

$i \leftarrow 0$; 2.4

ile başlar. Daha sonra *N* iterasyon *S*'nin *N* elemanını elde etmek için gerçekleştirilir.

$S_{N-i-1} \leftarrow L[p]$; 2.5

$p \leftarrow T[p]$; 2.6

$i \leftarrow i + 1$; 2.7

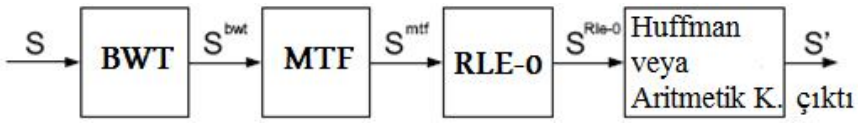
Örnek 2.15: *L="caraab", T=[4, 0, 5, 1, 2, 3]* ve *I=1* olsun. $p \leftarrow 1$ ve $i \leftarrow 0$ adımıyla işlem başlar. İlk iterasyonda $S_5 \leftarrow L[1] = 'a'$; $p \leftarrow T[1] = 0$; $i \leftarrow 1$ olur. İkinci iterasyonda $S_4 \leftarrow L[0] = 'c'$; $p \leftarrow T[0] = 4$ ve $i \leftarrow 2$ olur. Bu şekilde devam ederek *S* katarı sağdan sola elde edilir.

BWTdönüşüm tekniğinin neden etkili bir sıkıştırma yapmaya yardımcı olduğunu görmek için, "the" katarındaki *t* karakteri ve İngilizce bir metinde çok fazla sayıda "the" katarı geçtiği düşünülürse, *M*'deki sütunlar sıralandığı zaman, "he" ile başlayan sütunlar

bir arada olacaktır. Bunların da çok büyük bir kısmı *t* ile bitecektir. *L* metninin bir kesimi fazla sayıda *t* içerecektir. Bunun yanında "he"yi *s, T, S* gibi karakterler de izleyecektir. Bu sayede *L*'nin belirli kesimlerinde belirli harfler kümelenmiş olacaktır.

Böyle bir sıralama da *t* karakterinin yine *t* ile takip edilmesi olasılığını artırırken, başka bir harfle takip edilme olasılığını düşürmektedir. Bu da *öne alma* gibi algoritmaların kullandığı özelliktir. *MTF*, *t*'nin son geçtiği yere olan uzaklığı belirterek, başa bir *t* karakteri kopyalayan bir yöntemdir.

Sonuç olarak, bir karakterin ardışık tekrarlandığı küme, çıktıyı daha düşük karakter kümesi ile ifade etmeyi sağlamaktadır. *S* katarına BWT dönüşüm tekniği uygulandığı zaman, elde edilen *L* ve *I* çıktıları ile bir MTF kodlayıcı sayesinde *L*, *R* sayılar sırasına dönüştürülebilmektedir. Böyle üretilen bir *R* de *S*'den daha fazla sıkıştırılabilir durumdadır. Genelde *R*, Huffman tabanlı bir algoritma ya da aritmetik kodlama ile sıkıştırılmaktadır. Şekil 2.14'te BWT tekniği ile dönüştürülmüş metnin sıkıştırılmasındaki tüm adımlar gösterilmiştir.



Şekil 2.14 BWT dönüşümü içeren bir sıkıştırma işleminin tüm aşamaları

Açma işlemi için de sıkıştırılmış metne önce BWT'den sonraki adımlarda uygulanmış olan adımların tersleri uygulanmaktadır. Daha sonra elde edilen katara IBWT uygulanarak, metin orijinal hale dönüştürülür. [Martinez,2005]

3. DİZGİ EŞLEME ALGORİTMALARI

Dizgi eşleme algoritmaları, metin işlemenin yaygın olarak kullanıldığı alanlarda çok önemli bir yere sahiptir.

Boyutları çok büyük olan metinsel dosyalar üzerinde çalışıldığında bilgisayar kullanılması önemli performans artışı sağlamaktadır. Böyle büyük dosyalar içerisinde istenen verilerin kolayca aranabilmesi problemi, dizgi eşleme algoritmalarına duyulan ihtiyacı ön plana çıkarmaktadır. Bilgisayar teknolojisi ne kadar gelişmiş olursa olsun, veri boyutu arttıkça etkili sonuçların alınabilmesi için daha hızlı ve kabul edilebilir bellek büyüklüklerinde çalışan algoritmalara ihtiyaç duyulmaktadır.

Dizgi eşleme, metin içerisinde aranan dizginin en az bir ve genellikle bulunduğu tüm konumları bulmayı kapsamaktadır. Dizgi eşlemede, metnin hazırlanmasında kullanılan alfabe önem teşkil etmektedir. Dizgi eşleme algoritmalarının performansları kullanılan alfabeye göre farklılık gösterebilmektedir.[Say vd.,2002]

3.1. Karp & Rabin Dizgi Eşleme Algoritması

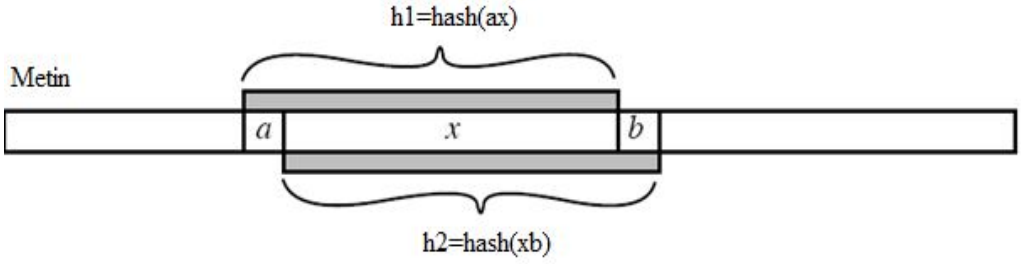
Karp & Rabin Dizgi Eşleme Algoritması içerisinde *anahtarlama* ve *parmak izi fonksiyonu* kullanmayı barındırmaktadır Öncelikle verilen iki nesne için parmak izini oluşturacak anahtar değerler hesaplanır. Eğer verilen iki x ve y nesnesinin *hash(x)* ve *hash(y)* ile verilen parmak izleri eşitse, iki nesnenin eşit olduğuna dair kuvvetli bir varsayım oluşmaktadır. Bu varsayım oluştuktan sonra gerek varsa daha derin bir inceleme yapılmaktadır. Parmak izlerinin iki önemli özelliği şunlardır;

- verimli bir şekilde hesaplanabilmelidir
- $x \neq y$ durumunda, $hash(x) = hash(y)$ olmamalıdır.

Anahtar fonksiyonu kullanmak Karp-Rabin dizgi eşleme algoritmasında kullanılan bir yaklaşımdır. İlk önce *m* uzunluklu bir dizginin parmak izi (*FP*) hesaplanır. Daha sonra metnin her *i* konumu için *metin[i+1...i+m]*'nin parmak izi (*FT*)

hesaplanır. Eğer *FP=FT* ise, *dizgi=metin[i+1…i+m]* olup olmadığı kontrol edilir. *i* konumunda *i+1* konumundaki anahtar fonksiyonunu kolay yoldan hesaplamanın bir yolu bulunmalıdır.

- *hash(metin[i+1…i+m])* kolayca *hash(text[i…i+m-1])*'den hesaplanmalıdır. Şekil 3.1 *h2*'yi *h1* kullanarak kolay yoldan hesaplamayı ifade etmektedir.



Şekil 3.1 h2'nin h1'den kolay bir şekilde hesaplanması

h2 kolay bir şekilde *h1*'den hesaplanmalıdır. *h2=f(a,b,h1), f*'in hesaplanması kolay olmalıdır.

Alfabenin $\{0,1\}$ elemanlarından oluştuğu ve her *m* uzunluklu *x* katarının bir ikili tamsayıyı ifade ettiği durumda, eğer *m* büyükse, sayı bir bellek hücresine sığamayacak kadar büyümektedir. Parmak izi fonksiyonu olarak da "*x mod Q*" örnek olarak alınabilir. *Q* mümkün olduğu kadar büyük bir asal sayı olarak seçilmelidir. O zaman parmak izi fonksiyonu aşağıdaki hali alır.

$$hash(x) = [x]_2 mod Q \qquad 3.1$$

Bu formülden yola çıkarak $[u]_2$, *u* ikili katarının *m* uzunluğu sayısıdır. *hash* fonksiyonundaki tüm parametrelerin uzunluğu *m*'dir. $g = 2^{m-1} mod Q$ olsun. O zaman *f* fonksiyonu aşağıdaki formül ile hesaplanır.

$$f(a,b,h) = 2(h - ag) + b \qquad 3.2$$

Bu özellikleri sağlayan herhangi bir anahtar fonksiyonu kullanılabilir. Anahtar fonksiyonu hesaplandıktan sonra, metnin üzerinde de anahtar fonksiyonun aynısı uygulanarak parmak izi değerleri elde edilir. Bu elde edilen değerlerde eşit sağlanan

noktalar gerekli olması durumunda daha detaylı bir eşleme yapılmalıdır.[Crochemore ve Rytter,1997]

3.2. Knuth – Morris – Pratt Dizgi Eşleme Algoritması

Düz ve sade bir algoritma yapısında olan KMP Dizgi Eşleme Algoritmasında, 1'den *n-m*'e kadar (*n*=metin uzunluğu, *m*=dizgi uzunluğu) kaydırmalar yapılır ve bu kaydırmalar sonucunda dizginin eşlenip eşlenilmediğine bakılır. KMP Dizgi Eşleme Algoritması'nın sıralı aramadan ya da deneme yanılma ile yapılan aramadan farkı, yanlış eşleşmenin olmayacağını garantileyen metin ve dizgi arasındaki kısmi eşleşmelerin bilgilerinden faydalanmasıdır.

Örneğin aşağıdaki metin ve dizgi ikilisinde, dizginin metnin en sol karakterine hizalanmıştır ve bir eşleşme bulmak amacıyla sağa doğru kaydırmalar yapılabilmektedir. Bu kaydırmaların nasıl olacağı metnin ve dizginin numaralanmış elemanları ile tarif edilmiştir.

T=xyxxyxyxyyxyxyxyyxyxyxxy

P=xyxyyxyxyxx

1. *P[1,..,3]*'ün *T[1,..3]* ile tam eşleştiği durum düşünüldüğünde burada bir yanlış eşleşme ortaya çıkmaktadır: *P[4]≠T[4]*. *P[1,..,3]=T[1,..3]* bilgisine dayanarak ve 3. konumdan itibaren dizgi ve metin sembollerini göz ardı ederek, potansiyel bir eşleşmenin nerede olabileceği çıkarımında bulunulmalıdır. Bu durumda, algoritma dizgiyi 2 pozisyon sağa kaydırmaktadır. Böylece *P[1]* ile *T[3]* aynı hizaya gelir. Bir sonraki karşılaştırma *P[2]* ve *T[4]* arasında gerçekleşmektedir.

2. *P[2]≠T[4]* olduğu için dizgi tekrar sağa kaydırılır. Böylece bir sonraki karşılaştırma *P[1]* ile *T[4]* arasında gerçekleşir.

3. Daha sonraki bir noktada, *P[1,…,10]* ile *T[6,…15]* eşleşmektedir. Burada yanlış eşleşme *P[11]≠T[16]* konumunda gerçekleşmektedir. Bir sonraki olası eşleşme içeren kaydırma 12. konuma kadardır. Bu sebeple dizgi sağa kaydırılır ve

P[1,…,11]=T[13,…23] eşleşmesi sorgulanır. Karşılaştırmalar, eşleşme olduğu sürece *P[4]=T[16], P[5]=T[17]* … şeklinde devam eder.

Buradaki bahsedilen kaydırmalar için geçerli bir kural tanımlanmalıdır. Bu tanımlar için de şu notasyonlar gereklidir: $S=s_1 \ldots s_k$ bir katardır. Her $s_1 \ldots s_i$, $1 \leq i \leq k$ katarı *s*'nin öneki olarak adlandırılır. Aynı zamanda, boş katar (sembol içermeyen) katar da *s*'nin önekidir. Eğer $s' \neq s$ ise o zaman s' uygun bir önektir. Benzer olarak, her $s_i \ldots s_k$, $1 \leq i \leq k$ katarı *s*'nin soneki olarak adlandırılır. Eğer $s' \neq s$ ise o zaman s' uygun bir sonektir.

P[1,…q] ile *T[i-q+1,…,i]* eşleşiyor ise ve bir yanlış eşleşme de $P[q+1] \neq T[i+1]$ konumunda gerçekleşirse, o zaman dizgi, *P[1,…q]*'nun en uzun uygun önekinin aynı zamanda *P[1,…q]*'nun uygun soneki, metnin *T[i]* konumundaki önekinin son karakteri ile hizalanacak şekilde sağa kaydırılır. Eğer $\pi(q)$, *P[1,…,π(q)]* en uzun uygun önek ve aynı zamanda *P[1,…,q]*'nun da soneki olduğu sayı ise, dizgi *P[1,..,π(q)], T[i-π(q)+1,..,i]* ile hizalanacak şekilde kaydırılır. KMP algoritması bu $\pi(q)$ değerlerini önişlemle hesaplar ve bir tabloda tutar. Örnek 3.1'de bu tablonun hesaplanmasına bir örnek verilmiştir.

Bu tablo şu şekilde hesaplanmaktadır: $\pi(1)$ her zaman 0'dır. $\pi(1,..,i)$'nin hesaplanmış olduğu ve $\pi(i+1)$'in hesaplanmakta olduğu düşünülsün. *P[1,…,π(i)], P[1,..,i]*'nin en uzun öneki aynı zamanda da sonekidir. $q = \pi(i)$ olsun. Eğer $P[i+1]=P[q+1]$, o zaman $\pi(i+1) = q+1$ olur. Aksi halde, $q = \pi(q)$ yapılır ve test tekrar edilir. $q = 0$ olana kadar devam eder. Bu noktada $\pi(i+1) = 0$ olur.

Özet olarak; KMP algoritması iki şekilde ilerler: (*P[1,…,q] ile [Ti-q+1,..,i]*'nin eşleştiği varsayılan durumlarda)

- Eğer $P[q+1]=T[q+1]$ ise, tam bir dizgi eşleşmesinin bulunduğu $q+1=m$ olmadığı sürece eşleşme uzunluğu genişletilir.
- Eğer $P[q+1] \neq T[i+1]$ ise, dizgi sağa kaydırılır.

Örnek 3.1.: P= x y x y y x y x y x x

q= 1 2 3 4 5 6 7 8 9 10 11

1≤q≤11 için; π(1) her zaman 0'dır. π(2) hesaplanırken "y" karakteri, dizginin herhangi bir öneki ile uyuşmamaktadır. Dolayısıyla 0'dır. π(3)'te ele alınan "x" karakteri ve kendisinden önce gelen karakterler dizginin sadece birinci elemanıyla önek durumu oluşturmaktadır. Dolayısıyla 1'dir. π(4)'te ele alınan "y" karakteri ve kendisinden önce gelen "x" karakteri hem dizginin öneki hem de o konumdaki sonekidir. Benzeşme sayısı 2'dir. Bu sebeple π(4)=2 olur. π(5)'te ele alınan "y" karakteri dizgi ve kendisinden önce gelen karakterler dizgi önekiyle benzeşme göstermezler. Dolayısıyla π(5)=0 olur. π(6)'da ele alınan "x" karakteri ve kendinden önce gelen karakterler ile dizgi önek benzeşme uzunluğu 1'dir. Dolayısıyla π(6)=1 olur. π(7) hesaplanırken ele alınan "y" karakteri ile kendisinden önce gelen "x" karakterinin oluşturduğu "xy" katarı dizgi öneki ile uyuşmaktadır. Bu durumda benzeşme uzunluğu 2'dir. π(7)=2 olur. Hesaplama yapmak için kullanılan daha kolay bir yol da q=7'de bir eşleşme olduğu ve q=6'da da eşleşme olduğunun hatırlanması, dolayısıyla π(6)'nın değerinin bir artırılması suretiyle π(7)'nin hesaplanabilmesidir. π(7)=2 olduğu için ve π(8)'de ele alınan "x" ile *P[1+ π(7)]* eşleştiği için π(7) bir artırılır. π(8)=3 olur. Aynı işlem π(9) için de geçerlidir. π(9)=4'tür. π(10) elde edilen benzeşme uzunluğu3, π(11)'de de bu uzunluk 1'dir. Çizelge 3.1'de verilen örnek için π() tablosu gösterilmiştir.

Çizelge 3.1 P dizgisi için π( ) değerleri

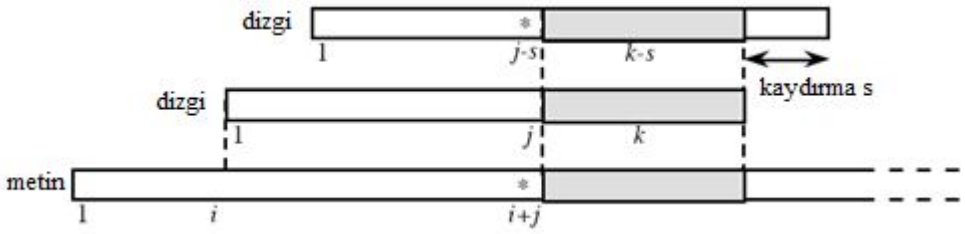
P	X	Y	X	Y	Y	X	Y	X	Y	X	X
q	1	2	3	4	5	6	7	8	9	10	11
π(q)	0	0	1	2	0	1	2	3	4	3	1

3.3. Boyer – Moore Dizgi Eşleme Algoritması

Boyer – Moore Dizgi Eşleme Algoritması dizgi karakterlerinin en sağındaki karakterden başlayarak metin ve dizgi üzerinde sağdan sola doğru arama işlemini gerçekleştirir. Uyuşmama durumunda (ya da tüm karakterlerin uyuşması durumunda) önceden hesaplanmış iki kaydırma fonksiyonu kullanmaktadır. Bunlar iyi-sonek (ya da uyuşma kaydırması) ve kötü-karakter (oluş kaydırması) olarak isimlendirilirler.

dizgi[j+1…m]=metin[i+j+1…i+m] ve *dizgi[j]≠metin[i+j]*

Algoritma tarafından elde edilen bilgi j değerinde saklanmaktadır. Aslında bir sonraki adımda j sabitlenmiş bir değere ayarlanacağından bu bilgi silinecektir. Amaç bunları kullanarak en büyük kaydırmayı gerçekleştirmektir.



Şekil 3.2 s<j durumu

Eğer metinde i ve $i+s$ arasında dizginin başlangıç konumu yoksa s kaydırması güvenlidir denir. Dizgi $i+s$ konumunda eşleşmeye başlarsa, o zaman aşağıdaki durumlar oluşur:

durum1(*j,s*): Her *k, s, t* için $j < k \leq m$, $s \geq k$ veya $dizgi[k-s] = dizgi[k]$

durum2(*j,s*): Eğer $s < j$ ise, $dizgi[j-s] \neq dizgi[j]$

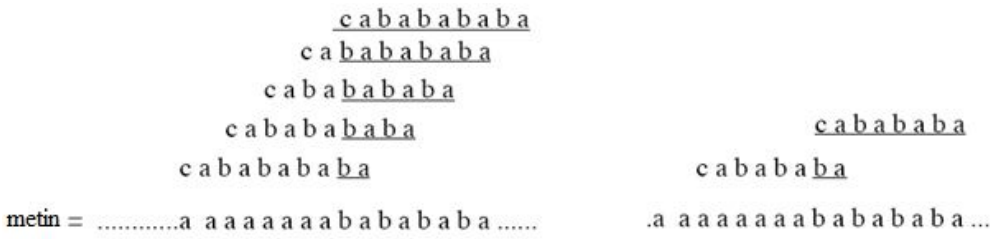
İki çeşit kaydırma tanımlanır. Her biri j *(<m)* pozisyonundaki dizgi soneki ile ilişkilidir ve kendi uzunluğu ile tanımlanmıştır.

$$D1[j] = min\{s > 0 : durum1(j,s)\} \qquad 3.3$$

$$D[j] = min\{s > 0 : durum1(j,s)\ ve\ durum2(j,s)\} \qquad 3.4$$

$D1[m] = D[m] = m - Sinir[m]$ tanımlanır. BM algoritmasında yanlış eşleşme durumunda bir konum kaydırmak yerine *D[j]* kadar kaydırma yapılır. Örnek 3.2'de *D1* ve *D* kullanılan kaydırma örnekleri verilmiştir.

Örnek 3.2: D yerine D1 kullanan benzer bir algoritmada P=cababababa ve T=aaaaaaaaaababababa ise:



Şekil 3.3 D1 ve D kaydırma kullanan BM

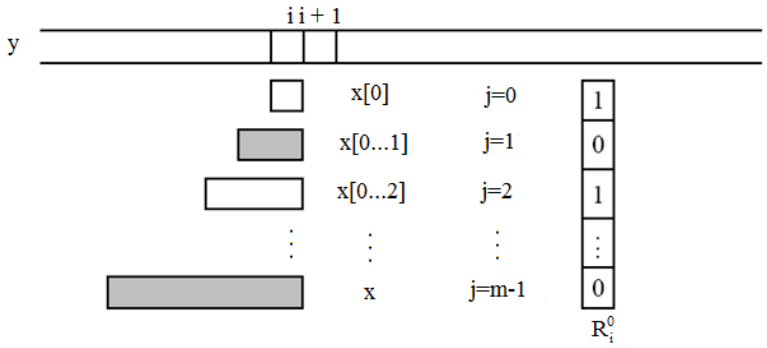
Şekil 3.3'te *D1–kaydırma* kullanan BM 30 karşılaştırma yaparken (sol), *D-kaydırma* kullanan BM 12 karşılaştırma (sağ) yapar

BM algoritması kaydırma uzunluğunu artırmak için başka bir yöntem daha kullanmaktadır. *metin[i+j]*'deki sembol dizginin herhangi bir yerinde geçmiyorsa, *s=j* uzunluğunda bir kaydırma yapılabilir. Örneğin $dizgi = a^{100}$ ve $metin = (a^{99}b)^{10}$ ise, her zaman 100 pozisyon kaydırma yapılır ve toplamda 10 karşılaştırmayla işlem biter. Aynı girdiler için normal BM 901 karşılaştırma yapar.[Crochemore ve Rytter,1997]

3.4. Shift – Or Dizgi Eşleme Algoritması

R^0, m uzunluklu bir bit dizisi olarak ifade edilir. R_i^0 vektörü *y[i]* metin karakteri işlendikten sonra R^0 vektörünün değeri olur ve *x* önekinin metnin *i* pozisyonu sonunda tüm eşleşmelerinin bilgilerini tutmaktadır. Şekil 3.4 R^0 vektörünü göstermektedir.

$$R_i^0[j] = \begin{cases} 0 & Eğer\ x[0 \dots j] = y[i-j \dots i] \\ 1 & Diğer\ durumlar \end{cases} \qquad 3.5$$



Şekil 3.4 $\boldsymbol{R_i^0}$ vektörü

R_{i+1}^0 vektörü R_i^0 'dan sonra aşağıdaki gibi hesaplanır. Her $R_i^0[j] = 0$ için:

$$R_{i+1}^0[j+1] = \begin{cases} 0 \; Eğer \; x[j+1] = y[i+1] \\ 1 \qquad Diğer \; durumlar \end{cases} \qquad 3.6$$

ve

$$R_{i+1}^0[0] = \begin{cases} 0 \; Eğer \; x[0] = y[i+1] \\ 1 \qquad Diğer \; durumlar \end{cases} \qquad 3.7$$

Eğer $R_{i+1}^0[m-1] = 0$ ise, tam bir eşleşme rapor edilir. R_i^0'dan R_{i+1}^0'e dönüşüm hızlı bir şekilde aşağıdaki gibi olur.

$0 \leq j \leq m-1$ için, eğer $x[j] = a$ ise $S_a[j] = 0$ olur.

S_a dizisi, x dizgisindeki a karakterlerinin yerini göstermektedir. Her S_a aramadan önce önişleme alınabilir. R_{i+1}^0'ın hesaplanması kaydırma ve veyalama olmak üzere iki işleme indirgenmiş olur. Örnek 3.3'te örnek bir metin için Shift – Or yaklaşımının adımları gösterilmiştir. [Crochemore ve Lecroq,1996]

$$R_{i+1}^0 = SHIFT(R_i^0) OR \; S_{y[i+1]} \qquad 3.8$$

Örnek 3.3: x=GATAA, y=CAGATAAGAGAA

Çizelge 3.2 Örnek metin için Shift – Or algoritması tablosu

S_A	S_C	S_G	S_T
1	1	0	1
0	1	1	1
1	1	1	0
0	1	1	1
0	1	1	1

	C	A	G	A	T	A	A	G	A	G	A	A
G	1	1	0	1	1	1	1	0	1	0	1	1
A	1	1	1	0	1	1	1	1	0	1	0	1
T	1	1	1	1	0	1	1	1	1	1	1	1
A	1	1	1	1	1	0	1	1	1	1	1	1
A	1	1	1	1	1	1	0	1	1	1	1	1

3.5. Horspool Dizgi Eşleme Algoritması

Boyer-Moore dizgi eşleme algoritmasında kullanılan kötü-karakter kaydırma fonksiyonu az sayıda sembol barındıran alfabeler kullanılan metinlerde çok etkili değildir, fakat alfabe dizgi uzunluğu ile karşılaştırıldığında daha geniş ise, sıklıkla ASCII tablosundan karakterler varsa ve dizgi eşleme sıradan bir metin editörü altında yapılıyorsa çok kullanışlı olabilmektedir. Kaydırma sadece Boyer-Moore algoritmasındaki kötü-karakter kaydırma fonksiyonunu, pencerenin en sağındaki karakteri kullanarak yapılmaktadır. Örnek 3.4'te Boyer – Moore ve Horspool tekniklerinin kaydırma adımları gösterilmiştir.[Ersin, 2008]

Örnek 3.4:

Çizelge 3.3 Boyer – Moore ve Horspool algoritmaları için kaydırma örneği

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**…**
a	b	c	a	b	d	a	a	C	b	a
b	c	a	a	b						
	b	c	a	a	b					
Boyer – Moore										

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**…**
a	b	c	a	b	d	a	a	c	b	a
b	c	a	a	b						
				b	c	a	a	b		
Horspool										

Örnek 3.4'te kıyaslanan çerçevedeki metin $t_0,...,t_4$=a b c a b'dir. Soneki "ab" eşleşmiş ancak c – a kıyaslaması bir yanlış eşleşmeye sebep olmuştur. BM'deki kötü karakter özelliği (a), "c" kötü karakterini kullanarak kaydırma mesafesini ayarlanır. Horspool ise metin çerçevesinin en sağ karakteri olan b'yi kullanmaktadır (b). Dizgi içerisindeki b karakteri, metinde bir b ile eşleşene kadar kaydırma yapılır. Dizginin sonundaki sayılmamaktadır.

3.6. Bacward – Oracle Dizgi Eşleme Algoritması

Boyer – Moore dizgi eşleme algoritması tipindeki algoritmalar dizgi soneklerini eşlemektedirler ama pencerede sağdan sola karakteri arayarak önek eşlemek ve sonra da kaydırma uzunluğunu genişletmek de mümkündür. BOM algoritması dizgiyi ters olarak

ele almaktadır ve ters dizginin sonek parçalarını kullanarak gerçekleştirilir. Bu veri yapısı bir kelimenin tüm soneklerini ve kelime içerisinde yer alan diğer kelimelerin birçoğunu tanımlayan bir otomattır. Ters kataın sonek parçalarını kullanan dizgi eşleme algoritması Backward Oracle Eşleme Algoritması olarak adlandırılmaktadır. BOM algoritmasının önişlem safhasında ters dizgi x^R'nin sonek parçaları yer alır. Dizginin faktörü olmayan kelimeleri de tanımasına rağmen, sonek tahminleri dizgi eşleme için de kullanılmaktadır. Çünkü sonek parçaları tarafından oluşturulan *m* uzunluğundan büyük ya da eşit kelimeler ters dizginin kendisidir.

BOM algoritmasının arama safhasında algoritma penceredeki karakterleri sağdan sola $O(x^R)$ otomatı ile q_0 durumundan başlayarak işler. O anki karakter için bir bağlantı kalmayana kadar devam eder. O anda metnin sonekiyle uyuşan en uzun dizgi öneki, $O(x^R)$'de alınan yolun uzunluğundan ve işlenen son durumdan küçüktür. Bu uzunluğun bilinmesiyle kaydırma sayısı hesaplanır. Örnek 3.5'te BOM algoritması kaydırma işlemi gösterilmiştir.

Örnek 3.5: İlk Deneme:

G C A T C G C A G A G A G T A T A C A G T A C G

* 8 7 2

G C A G A G A G

Kaydırma:5(8-3)

İkinci Deneme:

G C A T C G C A G A G A G T A T A C A G T A C G

8 7 6 5 4 3 2 1

G C A G A G A G

Kaydırma: 7(8-1)

Üçüncü Deneme:

G C A T C G C A G A G A G T A T A C A G T A C G

* 7 2 1

G C A G A G A G

Kaydırma: 7(8-1)

4. SIKIŞTIRILMIŞ DİZGİ EŞLEME

Sıkıştırılmış dizgi eşlemeden ilk olarak Amir ve Benson'ın çalışmasında bahsedilmiştir [Amir ve Benson, 1992]. Sıkıştırılmış dizgi eşleme verilen bir sıkıştırılmış metin üzerinde metni açmadan dizgi eşlemenin yapıldığı durumdur. Verilen bir T metni, bu metne karşılık gelen sıkıştırılmış metin $Z = z_1 \dots z_n$ ve dizgi P ise sıkıştırılmış dizgi eşleme P dizgisinin T içerisinde tüm varoluşlarını sadece P ve Z kullanarak bulma işlemidir.

Sıkıştırılmış dizgi eşleme metin veriler üzerinde yapılan önemli bir işlemdir. Sıkıştırılmış dizgi eşleme teknikleri bugünün metinsel veritabanlarında karşılaşılan iki problemin çözümü için ortaya atılmış yaklaşımlardır. Bu iki problem sırasıyla, metinsel veritabanlarının I/O zamanını ve kapladıkları yeri azaltmak için sıkıştırılmış durumda bulunmaları ve bu veritabanlarının etkili bir biçimde aranabilir olmasıdır. Bu iki birleştirilmiş istek beraber başarılması kolay şeyler değildir. 90'lardan önce tek çözüm olarak kullanılan teknik metni açıp aramayı açık metin üzerinde çalıştırmak suretiyle gerçekleştirilmiştir. Bu da zaman ve yer maliyeti olarak olumsuz sonuçlar ortaya çıkarmaktadır. [Karkkainen vd.,2000]

4.1. Üzerinde Sıkıştırılmış Dizgi Eşleme Yapılan Algoritmalar

4.1.1. Genel teknikler üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Kida, Shibata, Takeda, Shinohara ve Arikawa'nın 1999 yılında yaptıkları çalışma, sıkıştırılmış dizgi eşleme algoritması olarak birçok sözlük tabanlı metodu kapsamaktadır. Bu metotlar arasında LZ ailesi (LZ77, LZ78, LZSS, LZW), bayt çifti kodlama ve sabit sözlük yaklaşımları mevcuttur. Bu tekniklerin tamamını kapsayan sisteme "collage sistemi" adı verilmektedir.

CS, bir katarı *D* sözlük elemanı ve *S* sırası olmak şartıyla iki parametreli bir yapı biçiminde kodlayan sistemlerdir. Bu sistem içerisindeki algoritmaların içerdikleri temel işlemler birleştirme, yuvarlama ve tekrarlamadır.

Amir, Benson ve Farach'ın çalışması LZW üzerinde KMP otomatı karakteristiği gibi davranış göstermekteydi [Amir vd., 1994]. Yapılan bu çalışma ise onların fikrini tüm *CS*'ye genelleme amacına dayanmaktadır. Teknik olarak geliştirilen algoritma LZW için Amir, Benson ve Farach'ın geliştirdikleri algoritmayı genişletmektedir.

Bu genişletmeyi sağlamak için olası bir çözüm bit paralelliğini kullanmaktır. Bu yaklaşım verimli olmasına rağmen, daha genel durumlar için farklı yaklaşımlar ele alınmıştır. Buna ilaveten ele aldıkları yaklaşımlar çoklu dizgi eşlemeye de izin vermektedir.

Sabit sözlük yaklaşımı gibi bazı durumlarda, *D* kodlama esnasında *S* tarafından takip edilmektedir ya da *D, S*'den bağımsız olarak verilmektedir. Bu sebeple *D* önişleme tabi tutulmalıdır. Uyarlanabilir sözlük yaklaşımı gibi diğer durumlarda *S*'nin değişken değişken işlenmesi anında *D* artan bir şekilde yeniden oluşturulmaktadır. Dolayısıyla sıkıştırılmış dizgi eşleme için *D* her adımda işleme tabi tutulmalıdır.

Ekip bu çalışmada yukarıda anlatılan durumların detaylarını vermiş ve sonuç olarak da sıkıştırılmış dizgi eşlemenin birçok algoritma için genelleştirilmiş halini sunmuşlardır. [Kida vd, 1999]

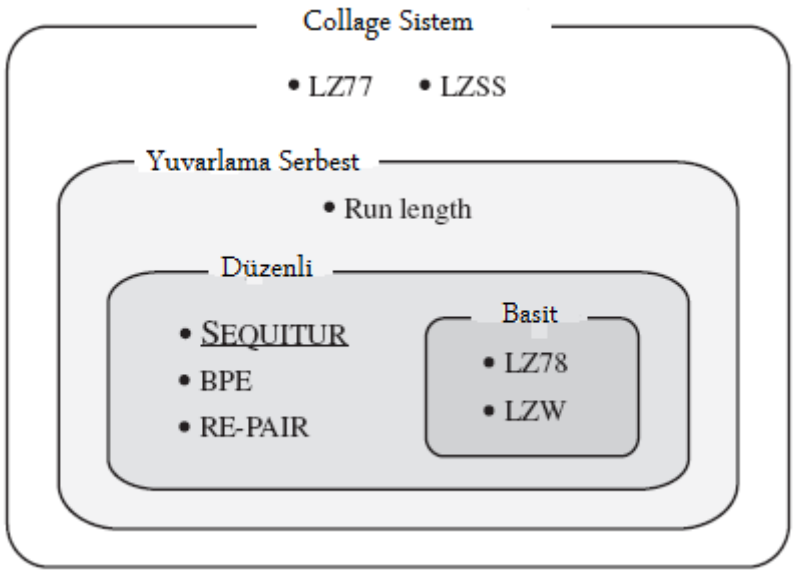
Shibata, Matsumoto, Takeda, Shinohara ve Arikawa 2000 yılında CS üzerinde Boyer – Moore dizgi eşleme tekniğine dayalı bir sıkıştırılmış dizgi eşleme çalışması yapmışlardır.

Bu çalışmada sıkıştırılmış dizgi eşleme birinci amaç olarak ele alınmıştır. İkinci amaç olarak da arama zamanını azaltılması hedeflenmiştir. Bu sebeple sıkıştırma oranından çok, arama zamanı konusuna değinilmiştir. İkinci hedefin birinci hedefi gerçekleştirmekten daha zor olduğu bu çalışmada savunulmuştur.

Çalışmada CS'nin detaylı bir tanımı yer almaktadır. Bu tanımdan sonra CS üzerinde dizgi eşleme, bu konuda yapılan eski çalışmalara da değinilerek tarif edilmiştir. Şekil 4.1 de CS hiyerarşisini göstermektedir.

Çalışmanın diğer bir bölümünde ise BM dizgi eşleme algoritmasının düz metinlerde uygulanışını dolayısıyla da BM'nin detaylı bir açıklaması yapılmıştır. Bu tekniğin CS'ye uyarlanması çalışmanın ana kısmını oluşturmaktadır. [Shibata vd., 2000]

Mitorai, Hirao, Matsumoto, Shinohara, Takeda ve Arikawa 2000 yılında *Sequitur* için bir sıkıştırılmış dizgi eşleme çalışması yapmışlardır. *Sequitur* girdi metninden katar hiyerarşisi çıkaran çok güçlü bir yazılımdır. Aynı zamanda da çok etkili bir sıkıştırma aracıdır.



Şekil 4.1 CS hiyerarşisi

Sequitur bir CS elemanı ve sözlük tabanlı sıkıştırma metodu olarak düşünüldüğünden sadece sıkıştırılmış dizgi eşleme için genel algoritmayı uygulamak gerekmektedir. *Sequitur* sonlu elemanlar sırasından bir gramer oluşturan ve bu oluşturulan grameri etkili bir sıkıştırma modeli olarak kullanan bir algoritmadır. *Sequitur*'un temel fikri bir kereden fazla geçen katarları kullanılmayan sembollerle yer değiştirmektir. Her tekrar gramer için bir kural oluşturur. Algoritmanın tüm işleyişinde aşağıdaki iki özellik gramer boyutunu azaltmak için ele alınır.

(P_1): Bitişik sembollerin hiçbir çifti gramerde birden fazla yer almaz.

(P_2): Her kural birden fazla kullanılır.

Birinci kural gramer kurallarının tek ve benzersiz olmasını, ikinci kural da her kuralın faydalı olduğunu belirtmektedir.

Örnek 4.1: T=abcdbcabcd, S başlangıç sembolü iken A ve C terminal olmayan sembollerdir.

$S \rightarrow CAC$ 4.1

$A \rightarrow bc$ 4.2

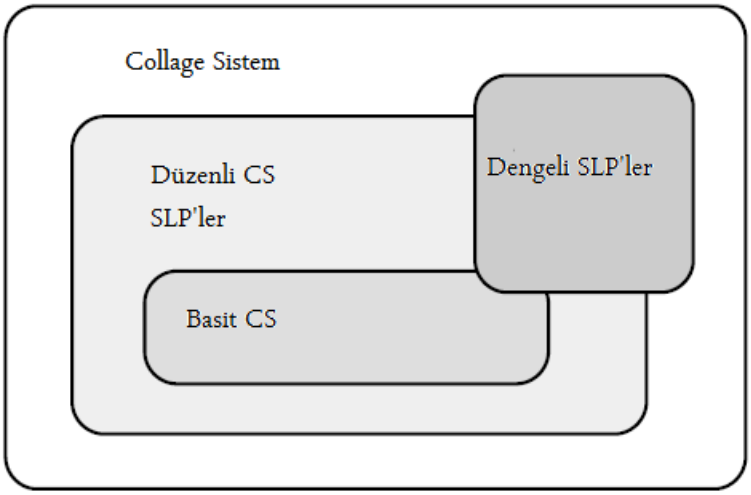
$C \rightarrow aAd$ 4.3

Yer değiştirmeler yapılarak kolaylıkla orijinal metin elde edilmektedir.

Sequitur üzerinde arama yapmak için KMP otomatı kullanılmıştır. CS üzerinde KMP ile arama yapmak daha önce yapılmış olan bir çalışmadır. *Sequitur* içinse küçük uyarlamalar gerekli olmaktadır. *D*, sözlüğün içerdiği *S* ile dönüşümlü çalışmaktadır. Bu sebeple de *D* adım adım *S*'nin işlenmesiyle açılmaktadır. Buna göre kullanılan atlama ve çıktı değerleri eşleşme arama esnasında birleştirilir. [Mitarai vd., 2001]

Hirao, Shinohara, Takeda ve Arikawa 2000 yılında Dengeli Düz Çizgi Programları için tam sıkıştırılmış dizgi eşleme yapan bir çalışma yaptılar.

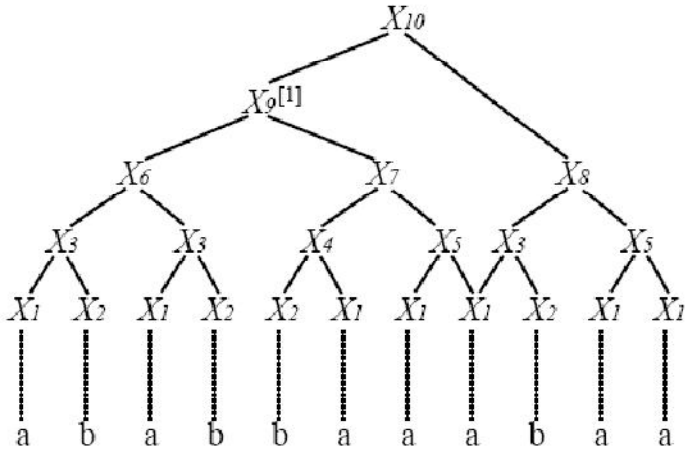
Bu çalışmada SLP ve onun bir çeşidi olarak ifade edilmiş katarlar üzerinde durulmuştur. SLP, bir katardan elde edilen Chomsky'nin normal formundaki içerik bağımsız gramerdir. Daha önceden geçen iki değişken birleşimine, bir değişken bir karakter birleşimine yapılan atamalardır. SLP ile ifade edilen bir katarın uzunluğu SLP'nin boyutuna bağlıdır. SLP'ler CS'nin merkezi alt sınıfıdır. Şekil 4.2 SLP'lerin CS içerisindeki konumunu göstermektedir.



Şekil 4.2 CS ve SLP'ler

Bu çalışmada daha hızlı dizgi eşlemenin yapılabildiği bir SLP türevi sunulmuştur. Burada, sadece aynı boydaki değişkenlerin son atanmaları hariç olmak şartıyla bileşimlerine izin verilmiştir. Buna da dengeli SLP adı verilmiştir. Çünkü evrim ağacı kök hariç tam ikili ağaç oluşturmaktadır. Bu sınırlamadan dolayı sıkıştırma oranı daha düşüktür. Ancak sıkıştırma işlemi normal SLP'den kolay ve hızlıdır.

Örnek 4.2: T=ababbaaabaa metni için dengeli SLP ağacı şekil 4.3'teki gibidir.



Şekil 4.3 Örnek 4.2'deki evrim ağacı

X_9 ve X_8 tam ikili ağaçları oluşturmaktadır. Bunları kullanmak daha hızlı sıkıştırılmış dizgi eşleme yapmayı sağlamaktadır.

Çalışmada, sıkıştırılmış dizgi eşleme için dizgi ile SLP ağacının kesişim noktaları bulunmaktadır. Dizgi, ağacın değişken sınırlarına göre ağaç içerisinde tam veya parçalı olarak gözükebilmektedir. Bu sebeple geçtiği yerlerin tamamı ve onlara karşılık gelen değişkenler hesaba katılmaktadır. Çalışmada dizgi eşleme matematiksel ifadelerle verilmiştir. [Hirao vd., 2000]

Kida, Matsumoto, Takeda, Shinohara ve Arikawa 2001 yılında daha önce CS için tek dizgi arayan algoritmalarını, çoklu dizgi eşleyen algoritma haline getirip sunmuşlardır. Sıkıştırılmış dizgi eşleme yapmak için yine KMP otomatını kullanmışlardır. Aynı zamanda LZW üzerinde de çoklu dizgi eşleyen bir yaklaşım sunmuşlardır. Buna ilave olarak, BM algoritmasının da çoklu dizgiler için bir uyarlaması çalışmalarında mevcuttur. [Kida vd., 2001]

Takeda, Shibata, Matsumoto, Kida, Shinohara, Fukamachi Shinohara ve Arikawa daha önceden yapmış oldukları sıkıştırılmış metinlerde dizgi eşleme algoritmalarına daha pratiksel yönden yaklaşan bir çalışma yapmışlardır. Bu çalışma daha önce yaptıkları çalışmanın tamamının birer tanımlarını içermektedir. [Takeda vd., 2001]

Inenaga, Shinohara ve Takeda 2004 yılında basit CS üzerinde sıkıştırılmış dizgi eşleme üzerine bir çalışma yapmışlardır. Basit CS, CS'nin LZ78 ve LZW'yi içeren bir alt sınıfıdır. Bu çalışmada basit SLP (SSLP) kullanılmıştır. [Inenaga vd., 2004]

Chen, Lu ve Jeffrey Ram 2004 yılında DNA sıraları üzerinde sıkıştırılmış dizgi eşleme yapan bir algoritma tasarlamışlardır.

DNA toplamda A, C, G ve T olmak üzere 4 farklı karakter içermektedir. Sıkıştırma mantığı her harfi 2 bit ile ifade etmeye dayanmaktadır. Sıkıştırılmış dizgi eşleme için de BM metoduna dayanan d-BM kullanılmaktadır.

Sıkıştırmada her harf 2 bit ile kodlanacağı için bir baytta 4 harf kodlanabilmektedir. Ancak kodlanmış uzunluğunun 4 ile bölümünden kalanının alınması, son baytta kaç bitin değerli olduğunu göstermektedir. Kodlama geri kalan boş bitlerin önemsiz bitlerle doldurulmasıyla oluşturulmaktadır. Daha sonra da *T*, DNA

sırası $\langle T', A\rangle$ ikilisi ile ifade edilmektedir. T' kodlanmış bayt sırasını, A da önemsiz hücre sayısını belirtmektedir. A değeri, $\{0,1,2,3\}$ kümesi içerisinden seçilmektedir.

Eğer yukarıdaki gibi bir sıkıştırma tekniği kullanılırsa, sıkıştırılmış dizgi ile sıkıştırılmış metin arasında bir hizalama problemi ortaya çıkabilmektedir.

Çözüm için P, DNA dizgisini 4 hizaya gelecek şekilde ifade etmek gerekmektedir. Her sıkıştırılmış dizgi $\langle P', A1, A2\rangle$ ile ifade edilmekte ve P' kodlanmış bayt sırasını, $A1$ önemsiz önek hücre sayısını, $A2$ de önemsiz sonek hücre sayısını vermektedir. $A1$ ve $A2$ $\{0,1,2,3\}$ kümesinden seçilmiş birer elemandır. Örnek 4.3 DNA için P dizgisinin 4 farklı hizaya geldiği durumları göstermektedir.

Örnek 4.3: P=ACTGA dizgisi $\langle P1', 0,3\rangle$, $\langle P2', 1,2\rangle$, $\langle P3', 2,1\rangle$ ve $\langle P4', 3,0\rangle$ olarak ifade edilmektedir.

$$P1' = 00\ 01\ 10\ 11\ 00\ xx\ xx\ xx$$

$$P2' = xx\ 00\ 01\ 10\ 11\ 00\ xx\ xx$$

$$P3' = xx\ xx\ 00\ 01\ 10\ 11\ 00\ xx$$

$$P4' = xx\ xx\ xx\ 00\ 01\ 10\ 11\ 00$$

Eşleşme sırasında kullanılan d-BM ise BM algoritmasının, üretilen 4 sıkıştırılmış dizgi için uygulanmasıyla oluşmaktadır.

4.1.2. Geçiş uzunluğu kodlama üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

RLE üzerinde yapılan ilk çalışmalardan biri Amir ve Benson tarafından 1992 yılında yapılmıştır [Amir ve Benson, 1992]. Hatta bu çalışma sıkıştırılmış dizgi eşleme konusunun öncü çalışmalarından biri olarak literatürde yerini almıştır.

Amir ve Benson çalışmasında sıkıştırılmış dizgi eşleme zamanının $O(|c(T)|)$ olması durumunda sıkıştırılmış dizgi eşlemenin optimal olacağını belirtmişlerdir. Sıkıştırılmış verinin boyutu veri içeriğine de bağlıdır. Bu sebepten en kötü durumda sıkıştırılmış dizgi eşleme zamanı $O(|T|)$ olmalıdır.

Tek boyutlu dizilerde RLE üzerine optimal çalışmalar daha önce de Amir, Landau ve Vishkin tarafından yapılmış, hatta yapılan bu çalışmalarda her satırın ve sütunun RLE sıkıştırmasının birleştirilmesi suretiyle elde edilen 2 boyutlu genellemesi sunulmuştur.

Amir ve Benson çalışmasında sırasıyla, çözmüş oldukları 2 boyutlu sıkıştırılmış dizgi eşleme probleminin doğru ve düzgün bir tanımını, geliştirdikleri algoritmalarının genel bir görüntüsü ile 2 boyutlu periyodiklik kavramının genel bir tekrarını, algoritmalarının dizgi önişlem adımlarını ve son bölümde de periyodik olmayan dizgiler için 2 boyutlu sıkıştırma problemi çözümünü sunmuşlardır.

Algoritmaları genel itibariyle diğer birçok algoritma gibi iki safhadan oluşmaktadır. Bunlar, dizgi önişlem safhası ve metin tarama safhasıdır.

Dizgi önişlem safhası: Dizgi, periyodiklik açısından analiz edilmektedir. Bu analizden dizginin periyot sınıfı ve iki aday kaynağın bağdaşabilirliği ile bunların üst üste gelmesiyle oluşabilecek yanlış eşleşmeleri gösteren tablo elde edilmektedir.

Metin tarama safhası: 3 adımdan oluşmaktadır.

1- **Sınırlandırma adımı:** Hazırlayıcı aday kaynakların seçildiği basit okuma yapılmaktadır.

2- **Bağdaştırma adımı:** Metin parçalarının aday kaynaklarının $\frac{m}{2} x \frac{m}{2}$ boyutlu bloklara ayrıldığı adımdır.

3- **Maskeyi kaldırma adımı:** Dizginin gerçek yerinin olup olmadığını belirleyen üçlü tarama adımıdır. [Amir ve Benson, 1992]

4.1.3. Bayt çiftleri kodlama üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Manber 1997 yılında yaptığı çalışmada dizgi eşleme algoritması üzerinde herhangi bir değişiklik yapmayıp, dizgi eşleme algoritmasından bağımsız ve üzerinde sıkıştırılmış dizgi eşleme yapılabilen bir sıkıştırma algoritması tasarlamıştır. Elde ettiği %30'luk sıkıştırma oranına ilave nerdeyse aynı oranda normal dizgi eşleme zamanından düşük sıkıştırılmış dizgi eşleme zamanı sonucuna ulaşmıştır. Bu çalışmaya kadar yapılan çalışmaların tamamı teorik iken, bu çalışmada algoritmanın pratikte uygulanması ile elde edilen sonuçlara da yer verilmiştir. Dizgi eşleme algoritmasından bağımsız olduğu için bu çalışmada sonuçları elde etmek için seçilen dizgi eşleme algoritması Boyer – Moore (BM) dizgi eşleme algoritmasıdır. BM algoritmasının hızlandırmak oldukça zor bir iştir. Dolayısıyla bit seviyeli algoritmalar ve Huffman Kodlama yerine, etkinliği ve verimliliği artırmak için bayt seviyeli kodlar kullanılmıştır. Ayrıca çalışmada algoritma detayları verilirken sıralı arama algoritması üzerinden gidilmiştir.

Sıkıştırma algoritmasının temeli basit dizgi yer değiştirmesi mantığına dayanmaktadır. Algoritmanın amacı, karakterlerin sık geçen çiftlerini bir baytla kodlanmış özel sembollerle yer değiştirmektir. Bir bayt 256 mümkün kodlamaya izin vermektedir fakat normal metinler daha azını kullanmaktadır. Örneğin ASCII kodlama 128 karakter kullanmaktadır ve çiftler için kullanmak üzere ilave 128 kod bırakmaktadır. Sıkıştırma her çiftin kendisi için ayrılmış özel baytla yer değiştirmesinden oluşmaktadır. Açma ise bu işlemlerin tersi sırada yapılması ile gerçekleştirilmektedir. İki işlem de oldukça hızlıdır. Ancak sıkıştırma oranı açısından LZ tabanlı algoritmalar ile kıyaslanamaz. Bunun yanında içerik bağımlı bu tekniklerde sıkıştırma mekanizmasının izini tutmadan dizgi eşleme yapmak, sıkıştırma durumunun okunan metne göre değişiklik gösterdiği için mümkün değildir. Bu da sıkıştırma oranındaki zaafın, dizgi eşleme hızı ile karşılanması anlamına gelmektedir.

Dizgi değiştirme metodu sıkıştırılmış veri üzerinde dizgi eşleme yapmak üzere bazı değişikliklere uğramaktadır. Bunun yanında bu algoritmada büyük bir problem göze çarpmaktadır.

Örnek olarak *wood* dizgisi arandığında, *wo* ve *od* ikililerinin de listeye dâhil olduğunu düşünülsün. *ij* nin kodlanan çifti, _ sembolünün de boşluğu sembolize ettiğini varsayılsın. *wood* dizgisi iki baytta *wo* ve *od* şeklinde olur. Fakat bu her zaman böyle olmak zorunda da değildir. Örneğin *_w*, *oo*, *d_* çiftleri de sık kullanılan çiftlerden olabilir. Hatta buna ilaveten *wood* kelimesi satırbaşı olabilir, daha geniş bir kelimenin (örn. *deadwood*, bu durumda çiftler *dw*, *oo*, *d_* şeklinde de olabilir.) parçası olabilir veya "." karakteri takip edebilir. Bunlar hep olasılıklar dâhilindedir. Bu birleşimlerin her biri farklı kodlamalara neden olacaktır.

wood kelimesinin kodlanmasındaki bu çok seçimliliğin nedeni, tüm sık kullanılan çiftler bilinse bile, *"w"* karakterinin bir çiftin ilk harfi mi, son harfi mi veya herhangi bir çifte ait mi olduğunu belirlemenin zor olmasıdır. Bu algoritmanın sorunun çözümü için dayandırılmak istendiği prensip, bir karakter grubu için muhtemelen çift kodlamalarının sayılarını kısıtlamaktır ve bu halde de iyi bir sıkıştırma sağlamaktır. Bir muhtemel yaklaşım sıkıştırmaya kelime başlarından başlamaktır. Diğer bir deyişle kelimeler ayrı ayrı sıkıştırılır ve diğer tüm semboller olduğu gibi bırakılır. Bu yaklaşım kısmen sorunu çözecektir. Fakat bu çözümün iki büyük zayıflığı vardır. Birincisi, tam kelimeleri aramayı kısıtlamasıdır. Bu durumda örneğin dizgi *"compression"* ise, metinde geçen *"decompression"* içinde bulunamayacaktır. İkincisi, boşluklar, virgüller ve noktalar çok sık rastlanan karakterler olduğu için, sıkıştırma anında onları göz ardı etmek oranlar açısından olumsuz etki ortaya çıkartacaktır.

Sorunun çözülmesi için üretilen değişik bir yaklaşım da, sık kullanılan çiftleri frekanslarına göre değil de, özel bir yolla seçmektir. Eklenilen temel prensip, hiçbir seçilen iki çiftin üst üste gelmemesidir. Diğer bir deyişle, bir karakter bir çiftte ikinci karakter durumunda ise diğer bir çiftte birinci karakter olmamalıdır.

İlk önce metni okuyup olası çiftlerin ve bu çiftlerin frekanslarının bulunduğu varsayılsın. *ab* çiftinin frekansını *f(ab)* ile gösterilmektedir. Bundan sonra bir *G=(V,E)* grafiği yapılmalıdır. *G*'nin düşey ekseni metindeki her farklı karakter için bir nokta olacak şekilde hazırlanmaktadır. *E* ise karakter çiftlerini ifade etmektedir. Kenar *ab*'nin ağırlığı *f(ab)* olur. Herhangi bir çift üst üste gelmeyecek şekilde en iyi kenarlar bulunmaya çalışılmaktadır. Düşey eksen V_1 ve V_2 olmak üzere iki kümeye ayrılır.

V_1'den V_2'ye kenar ağrılıkları toplamı en yüksek olacak şekilde kümeler seçilir. Ayrıca seçilen kenarların sayısını kısıtlamak gerekmektedir (kullanılmayan tahmini bayt sayısı 127'dir). V_1'den V_2'ye kenar ağrılıkları toplamı tam sıkıştırma oranını verecektir. Bu kenarların her biri, bir baytta kodlanan bir çifte denk gelmektedir. Bu da bir bayt kazanç sağlayacaktır.

Yukarıdaki problem sorunun çok açık bir formülüdür fakat şüphesiz bu aynı zamanda çok zor bir problemdir. Fakat diğer taraftan ilgilenilen grafikler çok büyük değildir. Önemli kenarlar sık kullanılan çiftleri ifade etmektedir ve metinler üzerine yapılan çözümleme çalışmalarında sık kullanılan çiftlerde o kadar da çok farklı karakter geçmediği görülmektedir. Bu sebepten dolayı, geniş grafikler için problem kontrol edilmesi zor olsa da, pratikte küçük grafikler için bu sorunu çözmek mümkündür. Bu problemi çözmek için birçok yaklaşım mevcuttur.

Son problem seçilen kenar sayısını sınırlandırmaktır. Bu sayı 127 olarak ayarlanmalıdır. Böylece 128 karakter ASCII kümesi ve bir tane işaretçi için kullanılabilmektedir. Bu işaretçiye metinde rastlanan nadir karakterler için ihtiyaç vardır. Bu durumda, sıkıştırılmış dosyada işaretçi ASCII olmayan karakter tarafından takip edilir. Bunun yanında dizgi eşleme için yeni satır sembolü sıkıştırılmamış formda bırakılır.

Çift seçimi esnasında limitler dâhilinde çift sayısı bulmak zordur. Bunun yerine öncelikle en iyi kümeleme, limit göz ardı ederek seçilir, daha sonra V_1 ve V_2'den en iyi 127 kenar seçilir.

Sıkıştırma algoritması iki adımlı bir algoritmadır. Birinci adımda dosya okunur ve en iyi çiftler belirlenir. İkinci adımda dosya tekrar okunur ve sıkıştırma gerçekleştirilir. Bu işlemi geliştirmek için, birinci adımda dosyadan alınan bir örnekleme uzayı kullanılmaktadır. Örneğin ilk bir milyon karakter ele alınır. Bu parça bir sonraki adım için bellekte tutulur. Diğer bir yöntem standart çiftler kullanmaktır. Bu da sıkıştırma işlemini yaparken zaman kazandıracak ancak birçok durumda daha az verimli sıkıştırma gerçekleştirecektir.

Bu algoritmada sıkıştırılmış dosya yaklaşık %30 kazanç sağlar. Eğer sıkıştırılmış dizgi eşleme gerekli değilse, çok verimli bir algoritma değildir. Çizelge 4.1'de sıkıştırma algoritmasının birkaç metin üzerinde vermiş olduğu sonuçlar gözükmektedir.

Çizelge 4.1 BPE'de geliştirilen algoritma için sıkıştırma oranları

Metin	**Orijinal Boyut**	**Sıkıştırma Kazancı (%)**	**Sıkıştırma Zamanı**	**Açma Zamanı**	**Çiftler için Zaman Yüzdesi**
Roget Thesaurus	1,38	31	10,3	0,9	14,6
CIA World Facts	2,42	28	11,4	1,7	13,2
King James' Bible	4,85	30	13,3	3,5	9,0
Wall Street Journal	15,84	33	23,9	11,2	6,7
Telecom Archieves	69,21	28	104,0	52,1	1,9

Sıkıştırılmış dizgi eşleme işlemi 5 safhadan oluşmaktadır. Birinci safhada, sıkıştırmada kullanılan sık çiftler okunur (sıkıştırılmış dosyanın başında yer alırlar) ve iki çevirme tablosu hazırlanır. Bir tablo tüm yeni karakterleri (128'den büyük değeri olması ile anlaşılırlar) kendi orijinal karakter çiftlerine haritalar. İkinci tablo tersidir; orijinal karakter çiftlerini yeni karakterlere haritalar.

İkinci safhada, dizgi ters tablo kullanılarak çevrilir (sıkıştırılır). Bu çevrim muhtemel ilk ve son karakter hariç diğerleri için tek alternatifli bir çevrimdir. İlk karakter V_2'ye veya son karakter V_1'e ait ise, dizgiden o an için atılır.

Üçüncü safhada herhangi bir dizgi eşleme algoritması sıkıştırılmış dosya üzerinde kullanılır ve tüm eşleşmeleri çıktı olarak dördüncü safhaya taşır.

Dördüncü safhada eşleşmeler çevrim tablosu kullanılarak açılır. Eğer dizginin ilk veya son karakteri atılmışsa, çıktılar da ona göre filtrelenir.

Beşinci safhada ise 4. safhadaki filtrelemeden geçen metin parçalarının tam hali ile ilk ve son karakterleri atılmış olan dizginin, o karakterleri eklenmiş hali kıyaslanır. Eşleşme olması durumunda bir eşleşmenin bulunduğu rapor edilir.

Bu beş safha çok iş yükü gibi gözükür ama iş yükünü belirleyen sadece 3. safhadır. Birinci safha sık kullanılan çiftlere bağlıdır ve bunlar en fazla 127 tanedir. İkinci safha dizginin uzunluğuna bağlıdır ki bu da normalde küçüktür. Dördüncü ve beşinci safhalar çıktı boyutuna bağlıdır ki bunlar da dizgiye bağlı olarak küçük olmalıdır. Bu sebeple çalışma zamanını büyük ölçüde belirleyen üçüncü safhadır.

Algoritmadaki en önemli nokta ise herhangi bir dizgi eşleme algoritmasının bu sıkıştırma algoritmasıyla bütünleşmiş biçimde çalışabilmesidir. [Manber, 1997]

Shibata, Kida, Fukamachi, Takeda, Shinohara, Shinohara, Arikawa 1999 yılında BPE üzerinde sıkıştırılmış dizgi eşleme yapan bir çalışmaya imza atmışlardır. BPE, basit bir evrensel dizgi yer değiştirme algoritmasıdır. Temel mantığı metinde çok sık geçen bir karakter çiftini, metinde geçmeyen bir karakter ile ifade etmektir. Bu işlem metinde kullanılmayan karakter kalmayana ya da sık geçen çift kalmayana kadar devam eder. Bu sebeple sıkıştırılmış metin yer değiştirme tablosu ve yer değiştirilmiş metin olmak üzere 2 parçadan oluşur.

Bu çalışmada BPE'nin sıkıştırılmış dizgi eşleme yapmaya müsait bir algoritma olduğu gösterilmiştir. Ayrıca geliştirilen algoritma *Compress* ve *Gzip* ile kıyaslanmıştır. Bununla beraber oldukça yavaş olan BPE sıkıştırma algoritmasını hızlandırmak için değişik uyarlamalar da yapmışlardır.

Sıkıştırılmış dizgi eşlemede, dizginin sıkıştırılmış kodlaması ile başka bir katarın sıkıştırılmış kodlaması benzer olabileceği için 2 yaklaşım ele almışlardır.

Birinci yaklaşımda işaretlenen eşleşmeler açılmaktadır. Bunlar herhangi bir dizgi eşleme algoritması kullanarak dizgi ile karşılaştırılmaktadır.

İkinci yaklaşımda KMP dizgi eşleme otomatı kullanılır. BPE sıkıştırma algoritması ile sıkıştırılmış metindeki karakterin birden fazla karakteri simgelediği durumlarda durum erişimlerinin bir serisi oluşur. Bu durum erişimlerinin ardışık

olanları için tek bir durum erişimi kullanılabilir. Bunları tutan tablo da çift boyutlu olarak tutulmaktadır. [Shibata vd., 1999]

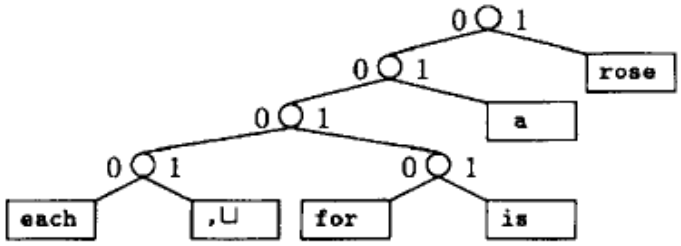
4.1.4. Huffman kodlama üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Moura, Navarro, Ziviani ve Yates 1998 yılında doğal dillerden oluşmuş metinler üzerinde hızlı açma ve sıkıştırma, aynı zamanda da etkili ve esnek sıkıştırılmış dizgi eşleme yapan bir algoritma sunmuşlardır. Sıkıştırma algoritmaları kelime tabanlı Huffman Kodlama esasına dayanmakta ve bit yerine bayt kodlama kullanmaktadır. *Compress* ve *Gzip* sırasıyla %40 ve %35 oranlarını yakalamışken, bu algoritma tipik İngilizce metinleri %30 oranında sıkıştırmaktadır. Ancak algoritmanın sıkıştırma zamanı *Compress* ile aynı iken *Gzip'*in neredeyse yarısı kadardır. Açma zamanları ise *Gzip'*ten küçük ve *Compress'*in de üçte biri kadardır.

Dizgi eşleme algoritması kelime merkezli Shift – Or ve BM tipli filtreye dayanmaktadır. Basit dizgiler aranırken *Agrep'*ten 2, karışık veya yaklaşık dizgiler için 8 kat hızlıdır.

Bu çalışmada yarı statik kelime tabanlı Huffman Kodlama kullanılmıştır. Metinler sadece kelimelerden oluşmamaktadır. Aynı zamanda ayraçlardan ve sonlandırıcı kelimelerden de oluşmaktadır. Dolayısıyla onları da içine alan bir model seçmek gerekmektedir. Bazı eski çalışmalarda bu iki eleman için farklı alfabeler kullanılmıştır. Metnin ayraç veya kelime ile başladığını bildikten sonra herhangi bir karışıklık ortaya çıkmamaktadır.

Bu çalışmada bunlar yerine "boşluksuz kelime modeli" adı verilen ve yukarıdaki modelin bir türevi olan model kullanılmıştır. Kodlama esasında bir kelimenin boşluk ile takip edildiği varsayılmaktadır. Bu model de sıkıştırma oranında küçük de olsa bir fark yaratmıştır. Şekil 4.4 kelime tabanlı Huffman kodlamanın boşluksuz kelime modeline örnek olarak verilmiştir. Bu örnekteki semboller {a, each, is, for, rose, ,_} ve frekansları da sırasıyla {2, 3, 1, 1, 1, 3, 1}'dir.



Orijinal metin: `for each rose, a rose is a rose`

Sıkıştırılmış metin: `0010 0000 1 0001 01 1 0011 01 1`

Şekil 4.4 Boşluksuz kelime modeline göre Huffman Kodlama

Verilen bir olasılık dağılımı için oluşturulan Huffman Ağacı oldukça geniştir. Dolayısıyla birçok uygulama için kuralsal ağaç kullanılmaktadır. Şekil 4.4'de gösterilen ağaç da bir kuralsal ağaçtır.

Bu çalışmada Huffman Ağacı 2 değil 256 dereceye sahiptir. Deneysel sonuçlar bit yerine bayt kullanmanın sıkıştırma oranında herhangi bir zararının olmadığını kanıtlamaktadır. Bununla beraber bayt Huffman kodunun açma zamanı bit Huffman kodunun ama zamanından oldukça iyidir.

Bayt Huffman Ağacı oluşturulurken dikkat edilmesi gereken bir nokta da ilk seviyede boş nod olmaması gerektiğidir. Bunun nedeni her seviye için 1 bayt ilave kodlama gerekmesi dolayısıyla da bir üst seviyede boş nod bırakılıp bir alt seviye nodlara eklenen kelimelerin kodlama esnasında 1 bayt daha az sıkıştırılması anlamına gelmektedir. Çizelge 4.2'de Bayt Huffman Kodlamanın diğer algoritmalarla kıyaslanma sonuçları verilmiştir.

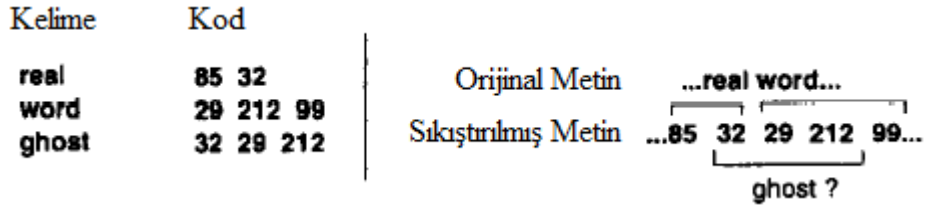
Çizelge 4.2 Bit Huffman, Bayt Huffman, Compress ve Gzip tarafından gerçekleştirilen sıkıştırma başarım değerleri

Metot	Dosyalar				
	ap	Wsj	doe	ziff	fr
Huffman (bit)	27,41	27,13	26,25	28,93	26,88
Bayt Huffman	31,16	30,60	30,19	32,90	30,14
Compress	43,80	42,94	41,08	41,56	38,54
Gzip	38,56	37,53	34,94	34,12	27,75

Sıkıştırılmış dizgi eşleme yaparken önce dizgi önişleme yapılmaktadır. Ağaç üzerinde gerekli ilerlemeler yapılarak dizginin sözlük kod karşılığı elde edilmekte ve daha sonra sıkıştırılmış metin bayt bayt okunmaktadır. Metin okunurken Huffman Ağaçları da gözden geçirilir. Eğer bir yaprağa ulaşılırsa, dizginin eşlendiği rapor edilir.

Son olarak ayraçlar ve sonlandırıcı kelimelerin nasıl işleneceği belirtilmelidir. Sıkıştırma esnasında bu kelimeler işaretlenir ve aramada bu işaretlemelerden faydalanılır.

Olası yanlış eşleşmeler de mümkündür. Örneğin "rose" kelimesinin kod karşılığı 47 131 8 olarak elde edilir ve bu kod doğrudan sıkıştırılmış metin içerisinde aranabilir. Ancak her eşleşme olan yerin doğrulanması gerekmektedir. Bu doğrulamanın yapılmasının gerekli olduğu şekil 4.5'te oldukça açık bir biçimde gösterilmektedir.



Şekil 4.5 Yanlış eşleşme durumu

Metni baştan işlemek yerine, metin sıkıştırılma esnasında aynı boyda küçük parçalara bölünmektedir. Bu parçalar blok olarak adlandırılır. Kodlar blok başlarına hizalanır. Böylece hiçbir kodun blok sınırlarını geçmemesi sağlanır. Bundan sonra yapılması gereken temel algoritmayı, eşlemeyi içeren blok başından uygulamak gerekir.

Sonuç olarak bu yapılan çalışma kelime tabanlı sıkıştırılmış dizgi eşleme işlemi gerçekleştiren bir algoritmadır. Yani başka bir deyişle kelimeyi bütün olarak arar, ancak kelime içerisinde geçebilecek katarları bulması beklenmemektedir. [Moura vd., 1998]

Moura, Navarro, Ziviani ve Yates yine 1998 yılındaki çalışmalarında sıkıştırılmış dizgi eşleme yapmak için başka bir teknik daha sunmuşlardır. Bu tekniğin önemli özelliklerinin basitlik, verimlilik ve esneklik olduğunu belirtmişlerdir. Elde ettikleri arama zamanı basit dizgi eşleme zamanlarına oldukça yakındır.

Bu çalışmada geliştirdikleri algoritma kelime tabanlı Huffman koduna dayanmaktadır. Her metin kelimesi tam bayt sırası olarak kodlanır ve Huffman Ağacı 2 yerine 128 dereceye sahiptir. Her bayt içerisinde 7 bit Huffman koduna, 1 bit de kelime başını işaretlemek için kullanılır. Bit kodlama yerine bayt kodlama sıkıştırma oranında herhangi bir zafiyete yol açmazken, işlem hızı açısından bit işlemeden daha hızlı olması da bir avantajdır.

Sıkıştırılmış dizgi eşleme algoritması öncelikle aynı Huffman Ağaçları kullanılarak dizgiyi sıkıştırır ve sıkıştırılmış dizgiyi, sıkıştırılmış metin içerisinde doğrudan arar. Arama işlemi metnin herhangi bir yerinden başlayabilir. Çünkü kod kelimesinin başlangıç baytı en yüksek biti ile işaretlenmiştir. Bu da dizgi eşleme aşamasında herhangi bir yöntemin kullanılmasını mümkün kılar.

Bu çalışmada kullanılan algoritmalar *cgrepboyer* isimli yazılım paketinde mevcuttur. *cgrepboyer* geniş metin grupları için tam ve yaklaşık sıkıştırılmış dizgi eşleme yapmaktadır.

Sıkıştırma algoritmasında yarı statik kelime tabanlı Huffman Kodlama kullanılmıştır. Birinci geçiş parametreleri (burada kelime frekansları) elde etmek amacıyla yapılır. Sıkıştırma işlemi ikinci geçişte yapılır. Ayrıca bu algoritmada "boşluksuz kelime modeli" de kullanılmıştır.

Moura et al'in çalışmasına atıfta bulunarak bayt Huffman sıkıştırma algoritmasında doğrudan dizgi eşleme yapılamayacağını belirtmiştir. Şekil 4.5'te de olası yanlış eşleme durumu gösterilmiştir.

Bu yanlış eşleme problemi her kelimenin başlangıcını işaretleyerek çözülmüştür. Eğer şekildeki 32 değerli baytın bir kelime başı olmadığı belli olursa, bu durumda bir eşleşme ortada gözükmeyecektir. Bu da birinci bit işaret biti olmak kaydıyla, 256 yerine 128 dereceli Huffman Ağacı kullanarak halledilebilmektedir. Bu bit kod kelimesi başındaki baytta 1 iken diğerlerinde 0 yapılırsa, işaretleme gerçekleşmiş olmaktadır.

Aynı örnekte "ghost" kelimesi için 32 29 12 değil 160 29 12 (160=128+32) aranır. Elde edilen kod bayt merkezli ve doğrudan aramaya izin verir bir kod halini almıştır.

Çizelge 4.2'de metotların sıkıştırma oranları verilmiştir. Sonuç olarak geliştirilen algoritma bayt – Huffman Koda %3 gibi bir üstünlük sağlarken Gzip ve Compress'in hala altındadır.

Çizelge 4.3 Huffman, Bayt Huffman, 7 – bit Huffman, Compress ve Gzip için sıkıştırma ve açma zamanı

Metot	Sıkıştırma					Açma				
	AP	WSJ	DOE	ZIFF	FR	AP	WSJ	DOE	ZIFF	FR
Huffman (1 bit)	490	526	360	518	440	170	185	121	174	151
Bayt Huffman	487	520	356	515	435	106	117	81	112	96
7 – bit Huffman	491	534	364	527	446	112	121	85	116	99
Compress	422	456	308	417	375	367	407	273	373	331
Gzip	1333	1526	970	1339	1048	147	161	105	139	111

Çizelge 4.3'te metotların açma ve sıkıştırma zamanları verilmiştir. Geliştirilen algoritma sıkıştırma zamanı açısından *Gzip*'ten 2-3 kat hızlı ve *Compress*'ten %17 yavaştır. Açma zamanı ise *Gzip*'ten %20, *Compress*'ten 3 kat hızlıdır. [Moura vd., 1998]

De Moura, Navarro, Ziviani ve Baeza – Yates 2000 yılında doğal dil içeren metinler için hızlı ve esnek sıkıştırılmış dizgi eşlemesi yapan bir çalışma yapmıştır. Basit kelimeler ve kelime grupları eşlemek için dizgiler sıkıştırılır ve sıkıştırılmış metni açmadan eşleme gerçekleştirilir. Karmaşık kelime veya kelime grupları için yaklaşık eşleme de basit gruplar için sıkıştırılmış dizgi eşlemeye yakın sonuçlar vermektedir. Bu çalışmada kelime temelli Huffman Kodlaması kullanılmıştır. İçerik olarak da üç eşleme yaklaşımı kullanılmıştır. İlk iki yaklaşım 1998 yılında yaptıkları iki çalışmaya dayanmaktadır.

Birinci algoritma Etiketlenmiş Huffman Kodu üzerinde herhangi bir dizgi eşleme algoritması uygulanmak suretiyle elde edilmiştir.

İkinci algoritma Düz Huffman Kodu üzerinde Shift – or dizgi eşleme algoritması kullanılarak dizgi eşleme yapılmıştır.

Üçüncü algoritma ise bu iki algoritmanın bir arada kullanıldığı bir yaklaşımdır. Başka bir deyişle THC sıkıştırma algoritması gibi doğrudan dizgi eşleme yapabildiği durumların yanı sıra, PHC sıkıştırma algoritması ile sıkıştırılmış veriler üzerinde de

uygulanabilmektedir. Bu algoritmalar aynı zamanda yaklaşık dizgi eşleme de yapabilmektedir.

Bu yaklaşımda öncelikle dizginin sıkıştırılmış karşılığı Huffman Ağacında bulunur. Daha sonra metin doğrudan aranır. Bulunan yerlerde PHC sıkıştırma algoritmasında kullanılan otomat kullanılır. Bu otomatın kullanılması için metin sıkıştırma esnasında aynı boyda küçük bloklara ayrılır. Eşlemenin olduğu blokta da temel algoritma uygulanır.[Moura vd, 2000]

Ziviani ve De Moura 2000 yılında metni ve metin çıkarım sisteminin indekslerini sıkıştıran metotlar üzerinde sıkıştırma yapmışlardır. Metin çıkarım sistemleri birçok bilgi çıkarımı sisteminin çekirdeğidir.

Önceleri IR sistemler üzerinde metin sıkıştırma uygulanamamaktaydı. Bunun nedeni sıkıştırılmış metne açmadan erişim yapılamıyor olmasıydı. Geliştirilen bazı teknikler sıkıştırılmış dizgi eşlemeye izin verirken aynı zamanda erişim zamanı ve metnin kapladıkları yeri küçülterek kazanç sağlamaktaydı. Bundan yola çıkarak bu çalışmada IR sistemlerde gerekli düzenlemeler yapıldıktan sonra IR sistemlerinin buna izin verip veremeyeceği araştırılmıştır.

Metin sıkıştırarak elde edilen kazançlar daha az yer gereksinimi, diskten okuma zamanı ya da transfer hızı azlığı ve de dizgi eşleme zamanı düşüklüğüdür. Metin sıkıştırma maliyeti olarak da sıkıştırma ve açma işlemleri maliyeti düşünülmektedir.

Etkili metin çıkarım teknikleri özelleştirilmiş indeksleme tekniğine gereksinim duymaktadır. Bunlardan basit ve popüler olanı “ters dosya”dır. “Ters dosya” belli bazı kelimelerin hangi metin grubunda geçtiğini işaretleyen vektörler tutmaktadır. İndeksin büyük kısmını da doküman numaraları kaplamaktadır.

Bu çalışmada sırasıyla metin sıkıştırma için mevcut yöntemlerin bazıları, mevcut yöntemlerin sıkıştırılmış dizgi eşlemeye uygun olup olmadığı, daha sonra da IR sistemler için yeni tekniklerin nasıl geliştirileceği anlatılmıştır.

IR sistemler için uygun olan sıkıştırma yöntemlerinden biri Huffman Kodlamadır. Genelde Huffman Kodlama, karakter temelli iken, IR sistemler için kelime temelli

olanların kullanılması daha uygun olmaktadır. Çünkü IR sistemlerin temelinde de kelimeler yatmaktadır. Kullanılan kelime tabanlı Huffman kodu aynı zamanda "boşluksuz kelime modeli"ni de içermektedir. Bunun yanında klasik Huffman Kodlamada yer alan bit kodlama yaklaşımının yerini bayt kodlamaya bırakması etkinliği artıracaktır.

Bit yerine bayt kodlama kullanan Huffman tekniğinin en önemli özelliği sıkıştırılmamış metin gibi sıkıştırılmış metnin de aranabilmesidir. Buna "doğrudan arama" denir.

Öncelikle sözlükten dizginin karşılığı bulunur. Daha sonra da dizginin karşılığı sıkıştırılmış metinde herhangi bir eşleme algoritması kullanılarak aranır.

Karşılaşılabilecek en büyük problem iki kod kelimesinin bazı durumlarda arka arkaya gelerek, orijinal metinde olmayan bir kodu oluşturması durumudur. Düz Huffman kodlarda mümkün iken, etiketli Huffman kodunda bu mümkün değildir.

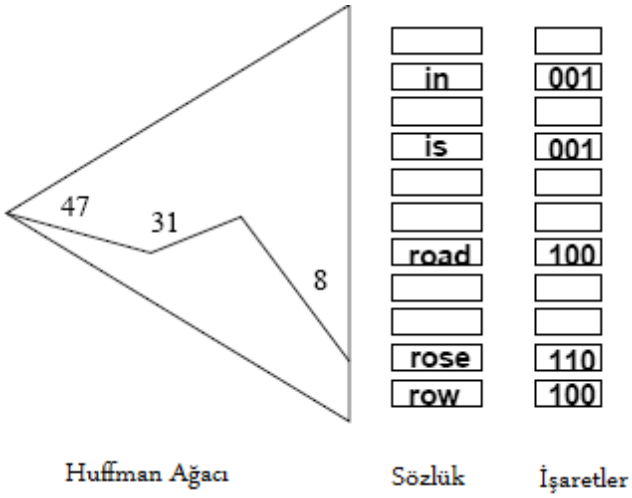
IR sistemler için önemli bir ihtiyaç da eşleşmedeki esnekliktir. Esnek (yaklaşık) eşleşme aranacak dizgide büyük/küçük harf birleşimleri gibi durumların göz ardı edildiği ve belli miktarda hataya izin veren bir tekniktir.

Bu çalışmada daha genel bir yaklaşım sunulmuştur. Bu yaklaşım aynı zamanda PHC sıkıştırma algoritmasında da kullanılabilmektedir.

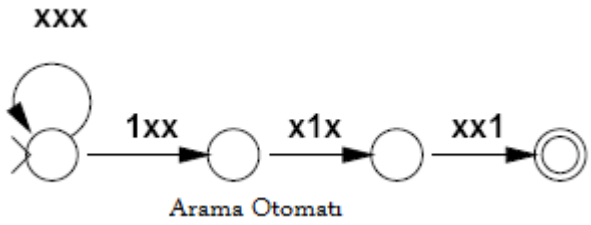
Tek bir kelime için dizgi eşleme işlemi, kelimenin kod karşılığının sözlükte ikili aranmasıyla başlamaktadır. Bulunduğu zaman kelimeye karşılık gelen yaprak işaretlenir. Daha sonra sıkıştırılmış metin bayt bayt okunur. Buna paralel sözlük de baştan başlayarak aynı adımlarla okunur. Bu işlem metni üretmeden açma yapmaya eşdeğerdir. Sözlükte işaretli yaprağa gelindiğinde eşleşme bulunmuştur.

Bu basit şema kolaylıkla karmaşık dizgilere de uygulanabilir. Karmaşık dizgilerin (kelime katarı halindeki dizgiler) her bir elemanı ayrı bir kelime olarak ele alınır. Eğer katar l elemana sahipse, her sözlük kelimesi için l bit uzunlukta bir maske ayarlanır. x kelimesinin i. biti, eğer x, katarın i. elemanı ile eşleşiyorsa işaretlenir. Katarın her i elemanı sırasıyla sözlükte aranır ve eşleştiği kelimelerin i. bitini işaretler. Şekil 4.6. "ro*

rose is” dizgisi için kelime başına bir hataya izin veren ve “ro*”ın da “ro” ile başlayan kelimeleri ifade ettiği durumlardaki maskeleri göstermektedir. Örneğin “rose” kelimesi dizginin 1. ve 2. konumu için doğrudur ve 3 elemanlı katar maskesini “110” olarak değiştirir.



Şekil 4.6 “ro* rose is” dizgisi için genel arama şeması



Şekil 4.7 Arama otomatı

Önişlem safhasından sonra Shift – Or otomatı eşleşme aramaları için etkili sonuç veren otomat olarak kullanılabilmektedir. Çizelge 4.4’te tam (k=0) ve yaklaşık (k=1,2,3) için, THC’de doğrudan ve otomatlı dizgi eşleme süreleri verilmiştir.[Ziviani vd., 2000]

Çizelge 4.4 wsj metin dosyası için %99 doğrulukla arama zamanları (sn)

Algoritma	k=0	k=1	k=2	k=3
Agrep	23.8 ±0.38	117.9±0.14	146.1±0.13	174.6±0.16
Doğrudan Arama	14.1±0.18	15.0±0.33	17.0±0.71	22.7±2.23
Otomat Arama	22.1±0.09	23.1±0.14	24.7±0.21	25.0±0.49

Shapira ve Daptardar 2004 yılında yaptıkları çalışmada KMP otomatı ile Huffman kodlama üzerinde bir sıkıştırılmış dizgi eşleme algoritması tasarlamışlardır.

Çalışmada hedeflerinde temel amaç Huffman Kodu üzerinde olası yanlış eşleşme sorununu, KMP otomatı üzerinde değişiklikler yaparak çözmek, dolayısıyla Huffman Kodlama ile sıkıştırılmış metinler üzerinde doğrudan sıkıştırılmış dizgi eşleme yapabilmektir.

Çalışmada, ilk yapılan iş dizgiyi de metin ile aynı ağaçları kullanarak sıkıştırmaktır. Daha sonra da eşleşmeyi bulmak için KMP tekniği kullanılmaktadır.

KMP algoritması temelinde bir yanlış eşleşme görüldüğü zaman, dizgi içerisinde işaretçinin ne kadar kaydırılacağının bilinmesi vardır. Bu kaydırma metin içerisindeki değil dizgi içerisindeki karakterlere bağlıdır. Sonuçta metindeki işaretçi asla azaltılmamaktadır. Bunun üstesinden gelmek için, metinde yanlış eşlemeye neden olan karakter ile bir sonraki karşılaşmayı gerçekleştirecek dizgi karakterinin indeksini elde etmek için kullanılan tabloyu oluşturma amacıyla dizgi önişleme alınır. Bu fikir sıkıştırılmış metinde sıkıştırılmış dizgiyi önişleme alarak kullanılmaktadır.

KMP – sonraki tablosu orijinal KMP tekniğindeki gibi sıkıştırılmış dizginin bitleri üzerinde oluşturulamaz. Bu kod kelimeleri sınırları içerisinde kıyaslamaya sebep olabilir. Bu kıyaslamaların elenmesi gerekmektedir.

Bunun bir çözümü KMP – sonraki tablosunu açık dizgiyi kullanıp, karşılık gelen bitlere uygulayarak gerçekleştirilebilir.

KMP – sonraki tablosu, kodlanmış metindeki tüm bitlerin kod kelimesi sınırlarını tutarak işleneceği şekilde düzenlenmelidir. Sıkıştırılmış dizgi önişleme şu şekilde olmaktadır; okunan kodlanmış metin soneki için bulunan en uzun kodlanmış dizgi öneki bulunur. Bunlar birinci kod kelimesi sınırına uyarlar. Bunun yanında ikili alfabe üzerinde çalışıldığı için, yanlış eşleşmeye neden olan karakteri de hesaba katarak algoritma genişletilebilir. Bir 1 biti yanlış eşleşmeye sebep oluyor ise, sıkıştırılmış dizgide karşılık gelen bit 0 olan yere kaydırılır. [Shapira ve Daptardar, 2006]

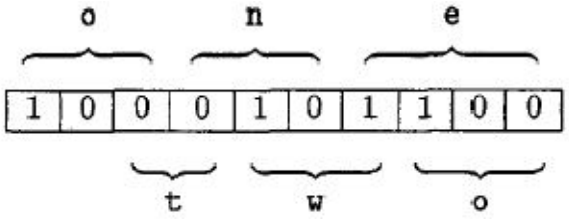
Klein ve Shapira 2005 yılında Huffman kodlama üzerinde yapılan sıkıştırılmış dizgi eşlemede ortaya çıkan yanlış eşleşmelerin sayısını azaltacak bir çalışma yapmışlardır. Huffman kodlamada yanlış eşleşmeye bir örnek örnek 4.3'te ve şekil 4.8'de verilmiştir.

Örnek 4.3:

Karakterler:	T	N	A	O	W	E	B	C
Kodlar:	00	010	011	100	101	1100	1101	1111

T=ONE $\varepsilon(T) = 1000101100$

P=TWO $\varepsilon(P) = 00101100$



Şekil 4.8 Yanlış eşleşme örneği

Eğer algoritma herhangi bir eşleşme bulamamış ise dizgi metinde kesinlikle yoktur denilebilir. Ancak dizgi sıkıştırılmış metinde var ise örnek 4.3.'ten de anlaşılacağı gibi kesinlikle var demek mümkün değildir. Bu eşleşmenin gerçek dizgi olup olmadığı en baştan açmadan anlaşılamaz ki bu da istenmeyen durumdur.

Böyle bir yanlış eşleşme problemine olası bir çözüm, metni eşit bloklara ayırmaktır. Her blok için giriş noktası belirlenir. Açma en son geçen giriş noktasından itibaren yapılmaktadır. Sıkıştırma oranı bir miktar düşmekte ancak bu yöntem baştan itibaren açmadan kurtulmayı sağlamaktadır. Bu çözüme alternatif olarak, çalışmada eğer dizgi *i* pozisyonunda geçiyor ise, belirli *K* bit geri gidip oradan itibaren açmak önerilmektedir. Eğer *i-K* pozisyonundaki bit bir kod kelimesi başlangıç biti değil ise hata en baştan yakalanmaktadır. *K* yeterince büyük seçilir ise, *i* bitinden önceki son bitlerin açılması doğru olacaktır.

i-K'dan başlayarak açma yapmaya bağlı olarak bir eşleşme olup olmadığı netleşmektedir.

Çalışmada sıkıştırılmış dizgi eşlemenin yanı sıra, yanlış eşleşme sayısı tahmini de yer almaktadır. Bunun için de iki modelden bahsedilmiştir. [Klein ve Shapira, 2005]

4.1.5. Kelime tabanlı bayt kodlamalar üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Brisaboa, Iglesias, Navarro ve Parama 2003 yılında üzerinde sıkıştırılmış dizgi eşleme de yapılabilen ETDC sıkıştırma algoritmasını sunmuşlardır. THC sıkıştırma algoritması benzeri bir algoritma olduğu için ETDC sıkıştırma algoritması üzerinde yanlış eşleşme olmadan dizgi eşleme yapmak mümkündür. Çalışmalarında Horspool ve Set-Horspool Algoritmaları ile sonuçları elde etmişlerdir. [Brisaboa vd., 2003]

Culpepper ve Moffat 2005 yılında bayt kodlamalarda dizgi eşleme üzerinde bir çalışma yapmışlardır. Bu çalışmada kelime tabanlı bayt kodlamalı Huffman, THC, ETDC, (s,c)-DC ve RPBC sıkıştırma algoritmalarında dizgi eşleme yapabilmek için, dizgi eşleme algoritmasında yapılacak değişikliklerden bahsedilmiştir.

Sonlandırıcı – devam eden bayt kodlarını kullanan algoritmalarda yapılması gerekli olan tek değişiklik yanlış eşleme filtresi koymaktır. Bu filtre eşleşme olan yerdeki baytı takip eden bayta bakar. Eğer devam eden bayt ise, bayt katarında kod kelimesinin sınırlarına uymayacağı için yanlış eşleşme rapor edilir. Eğer sonlandırıcı ise

o zaman bir eşleşme söz konusu olur. Diğer taraftan RPBC metodunda daha farklı bir yaklaşım kullanılmalıdır. Çünkü kod kelimesi kümesi uzunluğu ilk bayta bağlıdır ve sonraki bayta bakmaya elverişli değildir. RPBC kodları aranabilir yapmanın en kolay yolu metin – dizgi hizalamasını test ederek yanlış eşleşmenin hiç oluşmamasını sağlamaktır. Bunu yapmak için, arama işleminin dizgi kaydırma adımında kod kelimesi sınırlarına uymak gerekir. Ancak bu zahmetli istek şart değildir. Çünkü RPBC kodunun ilk baytı her kod kelimesinin kaç bayt uzunluğunda olacağını belirtmektedir. Bu da eş kodlanmayı ortadan kaldırmaktadır.

Sıralı aramada normalde bir bayt kaydırma olurken, bu kodlanan uzunluk belirten bayt sayesinde kaydırma bir bayttan fazla olabilmektedir. Bu da sıkıştırılmış dizgi eşleme için yapılmış bir değişikliktir.

Sembol atlama yaklaşımı, sonlandırıcı – devam eden bayt kodları için uygun değildir. Çünkü kod kelimesinin her baytını görmeden uzunluğunu hesaplamak mümkün değildir. Bu sebeple RPBC sıkıştırma algoritmasında sıralı arama, (s,c)-BC sıkıştırma algoritmasında sıralı aramadan daha az karşılaştırma yapmaktadır. Kod kelimesi sınırları içinde bir kaydırma yapılması imkânsızdır.

Benzer teknik tüm kaydırma içeren dizgi eşleme algoritmalarına uygulanabilir. Önek baytları ile sonek baytlarını ayırmak da uygun bir çözümdür. RPBC-PA olarak adlandırılan bu model ilave dizgi eşleme alternatifleri sunmaktadır.

Örnek 4.4:

Çizelge 4.5 4-uzunluklu kodlamaya göre sembollerin kod karşılıkları

Kelime	Sem.	Frek.	bc	phc	thc	dbc	scbc	rpbc
.\n	4	4	10 01	00	00 10	00	00	00
Bim	1	3	00	01	00 11	01	01	01
Ben	5	3	11 00	10	01 10 10	10 00	10	10
comes	3	2	10 00	11 00	01 10 11	10 01	11 00	11 00
brings	6	2	11 01	11 01	01 11 10	11 00	11 01	11 01
broom	7	2	10 10 00	11 10	01 11 11	11 01	11 10	11 10
(boşluk)	2	0	01	—	—	—	—	—

Çizelge 4.5'teki kod karşılıkları, "Fox in Socks" kitabındaki bir cümleye "boşluksuz kelime modeli" uygulanarak elde edilmiştir.

Çizelge 4.5'e göre 1, 3, 4, 5, 3, 4, 1, 6, 5 7, 4, 5, 6, 1, 7, 4 tam sayı sırası RPBC sıkıştırma algoritmasında aşağıdaki gibi ifade edilir.

Önek Baytları: 01, 11, 00, 10, 11, 00, 01, 11, 10, 11, 00, 10, 11, 01, 11, 00

Sonek Baytları: , 00, , , 00, , , 01, , 10, , , 01, , 10,

Burada dizgi eşleme işlemi iki aşamalı olacak şekilde değiştirilmelidir. İlk aşamada önek bayt sırası aranmaktadır. Bulunan yerlerin ikinci aşamada sonek baytları incelenmektedir.[Culpepper ve Moffat, 2006]

Carus, Buluş ve Mesut 2006 yılında sabit sözlükler kullanarak kelimeleri bayt tabanlı kodlayan bir algoritma sunmuşlardır. Sunulan algoritma ayrıca sıkıştırılmış dizgi eşlemeye de izin vermektedir. WordMatch adını verdikleri algoritma kelime tabanlı sıkıştırılmış dosyalardan kelime tabanlı dizgi eşleme yapmayı sağlamaktadır.

Kelime tabanlı kayıpsız sıkıştırma algoritması, İngilizcede en sık geçen kelimelerden oluşmuş sabit sözlük temeline dayanmaktadır. Kelimeler sahip oldukları harf sayılarına göre sözlükte bloklar halinde gruplanmıştır. Her sözlük için iki bayt kodlanmıştır. Birinci bayt blok indeksini, ikinci bayt da kelimenin blok içerisindeki indeksini vermektedir.

Sıkıştırma esnasında öncelikle kelime okunur, boşluk karakteri eklenir ve sözlükte aranır. Eğer kelime sözlükte bulunur ise blok numarası ve indeks kodlanır. Eğer kelime sözlükte bulunamazsa, boşluk karakteri düşürülür ve kelime iki kaçış karakteri arasına yazılır. Yapılan testlerde %50 civarında bir sıkıştırma oranı elde edilmiştir.

Bu algoritma ile sıkıştırılmış metinlerde dizgi eşleme işleminde öncelikle dizginin sözlükte yer alıp almadığı kontrol edilir. Eğer dizgi sözlükte yer alıyor ise blok ve indeks numarası, sıkıştırma esnasındaki gibi hesaplanır. Daha sonra sıkıştırılmış metinde ilk bayt okunur. Eğer bir kaçış karakteri okunmuş ise bir sonraki kaçış karakterine kadar atlanır. Eğer okunan bayt kaçış karakteri değilse bir bayt daha okunur

ve dizginin kodu ile bu iki bayt karşılaştırılır. Eğer dizgi sözlükte yer almıyor ise, dizgi kaçış karakterleri arasına yazılarak elde edilen kod kelimesi sıkıştırılmış metinde aranır. [Carus vd., 2006]

4.1.6. LZ ailesi üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

4.1.6.1. LZ77 sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Farach ve Thorup 1995 yılında sundukları çalışmada LZ77 (LZ1) sıkıştırma algoritması üzerinde sıkıştırılmış dizgi eşleme yapmayı sağlayan bir algoritma geliştirmişlerdir. Bu çalışmada sıkıştırılmış dizgi eşleme algoritmasının çalışma zamanının, basit bir açıp arama yapan algoritmadan küçük olması durumunda rekabet edebilir olduğunu belirtmişlerdir.

Yüksek entropili katarlarda "bilgi"nin katar içerisinde çok veya az dağıtılmış olduğu düşünülebilmektedir. Düşük entropili katarlar için bilgi belli kesimlere kümelenme eğilimi gösterebilmektedir. Geliştirdikleri algoritmada bu kümelenmenin avantajını sadece ilgili yerlerde kısmi açma yaparak kullanma prensibine dayanmaktadır.

Karp ve Rabin'in parmak izi uygulamasını vererek LZ1 sıkıştırma algoritması üzerinde rekabet edebilir bir zaman elde etmişlerdir. Sıkıştırılmış dizgi eşleme yapabilmek için LZ1 sıkıştırma algoritması üzerinde bazı dönüşümler gerçekleştirilmiştir. Bunlardan ilki $\sum$ alfabesinin $[1, |\sum|]$ içerisine haritalamak, diğeri de her (P_i, L_i, C_i) üçlüsünü(P_i, L_i) $(-C_i, 1)$, ikilileri ile yer değiştirmek olarak belirtilmiştir. $L_i = 0$ olan özel durumlarda(P_i, L_i, C_i) üçlüsü sadece $(-C_i, 1)$ ikilisi ile yer değiştirilmektedir. Böylece her belirli katar parçası pozitif uzunluğa sahip olur. Örnek olarak, "a^n" metni $a(-1,1)(0, n-1)$ sıkıştırılmış şeklini almaktadır.

Eğer u, dizginin geçtiği en düşük konum ise;

$P = T[u, u + |P| - 1]$ ve $U_i \in [u, u + |P|]$'dir. $i = 0, \ldots, N$ için en uzun dizgi alt katarı P_i^+ , $T[U_i, u]$'nun öneki ve en uzun dizgi alt katarı P_i^-, $T[0, U_i + L_i - 1]$'nin sonekidir. Bu durumda dizgi eşleme P'nin $P_i^- P_{i+1}^+$'nin bir alt katarı olup olmadığı sorusu ortaya çıkar Tüm metni açmak yerine sadece bu formdaki işaretli yerleri kısmi olarak açıp aramak mümkün olmaktadır. Ancak tüm $P_i^{+/-}$'lerin karakterlerini tutmak çok pahalıya mal olur. Dolayısıyla her $P_i^{+/-}$ dizgi içerisinde başlangıç ve bitiş gösteren işaretçiler olarak ifade edilir.

Özet olarak Farach ve Thorup çalışmasında sıkıştırılmış dizgi eşleme algoritmasını işaretli kısımları kısmi açma yöntemiyle gerçekleştirmişlerdir. [Farach ve Thorup, 1995]

Daha sonra Gasieniec, Karpinski, Plandowski ve Rytter 1996 yılında LZ1 sıkıştırma algoritması üzerinde sıkıştırılmış dizgi eşleme konusunda bir çalışma yapmışlardır. Sıkıştırılmış dizgi eşlemenin yanında bazı başka konuları da ele almışlardır. Bu konular palindromları, periyodları ve katarların karelerinin hesaplanması olarak belirtilmiştir.

Sıkıştırılmış dizgi eşleme yapabilmek için öncelikle sıkıştırılmış metni parçalara bölme (faktorizasyon) yapılır. *w* katarı aşağıdaki şekilde parçalara ayrılmaktadır. Örnek bir katarın parçalara bölünmesi de örnek 4.5'te gösterilmiştir.

$w = c_1 f_1 c_2 f_2 \ldots . f_k c_{k+1}$, $c_1 = w[1]$ ve her $1 \leq i \leq k$, $c_i \in \sum$ için f_i, $f_i c_{i+1} \ldots . f_k c_{k+1}$'in en uzun önekidir. f_i, [p,q] aralığı olarak tanımlanır. $f_i = w[p, q]$ olur.

Örnek 4.5: aababbabbaababbabba#

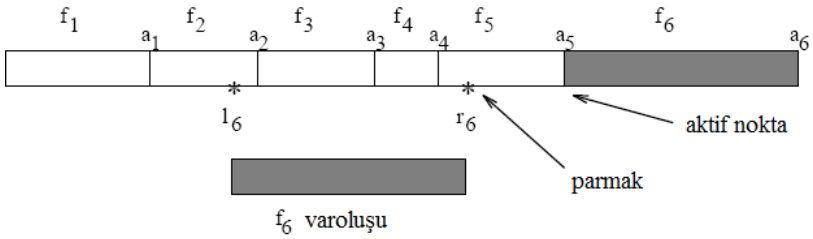
$c_1 f_1 c_2 f_2 c_3 f_3 c_4 f_4 c_5$= a a b ab b abb a ababbabba#

LZ(aababbabbaababbabba) = a[1,1]b[1,2]b[4,6]a[2,10]#

Terminal semboller başa alınarak aşağıdaki değişiklikler yapılır. Terminal sembol pozisyonları kelime içerisinde -1, -2 … şeklinde devam eder. Terminal sembole her atıf o konum içerisindeki aralık ifadesi ile değiştirilir.

ab[-2,-2][1,1][-1,-1][1,2][-1,-1][4,6][-2,-2][2,10]#

Lempel – Ziv kodu katarların aralık belirten doğal bir faktorizasyonudur. Her faktörün de son pozisyonu aktif nokta olarak adlandırılır. Sadece aktif noktaları bilmek yeterli değildir. Bunun yanında faktörün daha önce geçtiği yerleri belirten parmak göstergeci de tanımlanmalıdır. Bu işlemler şekil 4.9'da gösterilmiştir.



Şekil 4.9 Bir katarın LZ faktorizasyonu

Tanımlanan veri yapıları her f_i faktörü için solküme(i) ve sağküme(i) içermektedir. Bu veri yapıları da dizginin geçtiği yerleri, palindromları ve kareleri hesaplamaya yarar. [Gasieniec vd., 1996]

4.1.6.2. LZ78 sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Navarro ve Raffinot 1999 yılında metnin blok sırası halinde olduğu durumlar için bir sıkıştırılmış dizgi eşleme algoritması geliştirmişlerdir. Bu blok yapısı da Lempel – Ziv sıkıştırma ailesinin karakteristik özelliğidir.

Geliştirdikleri algoritmayı önce LZ77 sıkıştırma algoritmasına uygulamışlardır. LZ78 sıkıştırma algoritmasına, algoritmanın uygulanması ile daha etkili sonuç veren bir algoritma olduğu ortaya çıkmıştır. Daha sonra LZ77 ile LZ78 sıkıştırma algoritmalarının birleşimi şeklinde davranan algoritma sunmuşlardır. Bu algoritma

LZ77 sıkıştırma algoritması gibi hızlı sıkıştırıp, LZ78 sıkıştırma algoritması gibi hızlı dizgi eşleme zamanına sahiptir.

Yaklaşımları basittir ve bit paralelliğine dayanmaktadır. Bit paralelliği bir makine kelimesinin (gelen olarak 32 veya 64 bit) işlemci tarafından bütün bitlerinin aynı zamanda işlenmesi esasına dayanmaktadır.

Dizgi eşleme tekniklerinin temel tanımı, metnin karakter yerine bloklar halinde verildiği zaman verilebilmektedir. Bloklar ya 1 karakterden ya da önceden var olan blokların birleşmesiyle oluşmaktadır. Blok blok metin işlenir. Her yeni B bloğu için, B'nin bir tanımı çıkarılır. Şekil 4.10 blokların önek ve sonek olma durumlarını göstermektedir.

$D(B) = (L, O, S, P, M)$

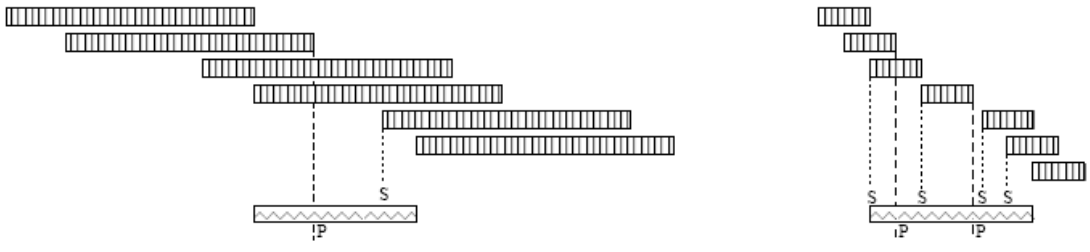
$L = |B|$, B'nin karakter olarak uzunluğu

$O = Offs(B)$, B'nin işlendiği zamana kadar olan metin uzunluğu

$S = Suff(B)$, B'nin tam geçtiği ya da dizgi sonekinin B önekiyle uyuştuğu konum

$P = Preff(B)$, B'nin tam geçtiği ya da dizgi önekinin B sonekiyle uyuştuğu konum

$M = Matches(B)$, dizginin geçtiği blok konumları



Şekil 4.10 Uzun ve kısa bloklar için önek (P) ve sonek (S) durumları

D(B) tanımı iki şekilde elde edilir.

a- Blok tek bir karakterdir ve tanım doğrudan elde edilir.

b- Blok önceden geçmiş blokların birleşimidir. Tanım önceki blokların tanımları incelenerek elde edilir.

Yeni bir blok tanımı elde edildiği zaman, bu tanım dizgi eşleme durumunu güncellemek için kullanılır. Blok işlenmesi tamamlanmış olur ve bir sonraki bloğa geçilir.

Çalışmalarında sunulan birleşik yaklaşım ise dizgi eşlemede LZ77 sıkıştırma algoritmasının en kötü yanı olan blokların kendi kendini çağırmasını engellemek amacıyla geliştirilmiştir. Geliştirilen algoritma LZ77 ile LZ78 sıkıştırma algoritmaları arası bir algoritmadır. Örnek 4.6'da yeni yaklaşıma bir örnek ve çizelge 4.6'da da yeni algoritmanın diğer algoritmalarla kıyaslanması verilmiştir.

T metninin $t_1 \ldots t_i$ öneki $Z = b_1 \ldots b_c$ olarak sıkıştırılmış kabul edildiğinde, $t_{i+1} \ldots t_u$'nun en uzun sıkıştırılmış $b_r \ldots b_{r+h}$ v öneki aranır. Eğer birden fazla alternatif var ise minimum blok sayısı olan alınır. *(r,h)* bloğu kodlanır. *v* boş ise $(0, t_{i+1})$ olarak kodlanır.

Örnek 4.6: Metin = ananas

Okunan ilk karakter=a, Sözlük boş olduğu için 1 numaralı sözlük indeksine a yazılır.

Çıktı=(0,a)

Sözlük = 1-a

Okunan karakter = n, Sözlükte yer almaz. Sözlüğün sıradaki elemanına eklenir. Çıktı(0,n)

Sözlük= 1-a, 2-n

Okunan karakter = a, Sözlükte 1 numaralı indekste. Bir karakter daha okunur.

Okunan karakter = n, Sözlükte 2 numaralı indekste bulunur. Çıktı olarak ilk bulunan indeks ve iki indeks arası fark yazılır. Sözlüğe “an” eklenir.

Çıktı = (1,1)

Sözlük= 1-a, 2-n, 3-an

Okunan karakter = a, Sözlükte 1 numaralı indekste. Bir karakter daha okunur.

Okunan karakter = s, Sözlükte yer almaz.

Çıktı = (1,0)

“as” ikilisi sözlükte yer almadığı için çıktı olarak a’nın indeksi ve 0 üretilir. Kalan “s” karakteri ise sözlükte yer almadığı için çıktı olarak (0,s) üretilir [Navarro ve Raffinot, 1999].

Sözlük= 1-a, 2-n, 3-an, 4-s

Sıkıştırılmış metin = (0,a)(0,n)(1,1)(1,0)(0,s)

Çizelge 4.6 Üç metot arasında ortalama sıkıştırma oranları

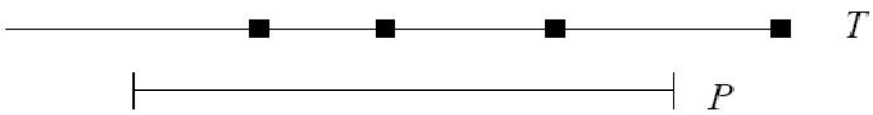
Dosya	Boyut (Kb)	LZ77(%)	LZ78(%)	Hybrid(%)
English	1324	29,67	36,15	29,28
DNA	1390	28,03	25,30	29,08
book1	751	34,10	40,70	35,62
book2	597	29,33	40,21	30,44
paper1	52	32,33	46,20	34,29
paper2	80	32,68	43,00	34,80
paper3	45	35,10	45,50	38,12
paper4	13	37,60	47,95	41,07
paper5	12	39,85	50,79	41,74
paper6	37	33,60	47,72	35,69
Progc	39	32,21	47,99	34,16
Progl	70	22,45	39,10	23,30
Progp	48	21,34	40,36	22,46

Navarro ve Tarhio 2000 yılında LZ78 ve LZW sıkıştırma algoritmaları üzerinde Boyer – Moore dizgi eşleme yaklaşımı kullanarak sıkıştırılmış dizgi eşleme çalışması yapmışlardır.

Bu çalışmaya kadar BM tekniği sıkıştırılmış dizgi eşlemeler için kullanılmamıştır. Bu uygulamayla daha önceden var olan uygulamaların arama zamanında %30 önüne geçmişlerdir.

Çalışmada öncelikle LZ78 ve LZW sıkıştırma algoritmalarının, daha sonra da BM tekniğinin detaylı bir tanımı verilmiştir. LZW sıkıştırma algoritması ayrı bir teknik değil de LZ78 sıkıştırma algoritmasının bir çeşidi olarak ele alınmıştır.

Şekil 4.11'de LZ78 sıkıştırma tekniğine kuramsal pencere yaklaşımı ile örnek verilmiştir. Siyah kutular *b=(s,c)* bloğunun belirgin karakterini, düz çizgi ise iç karakterlerini simgelemektedir. Bir metin daha önceden işaret edilmiş blokların tekrar sıralanmasıyla elde edilir. (*s*, *s* tarafından işaret edilen blok ve onun öncesi … gibi)



Şekil 4.11 LZ78 sıkıştırma algoritmasındaki blok yaklaşımı

Bu duruma BM tekniğini uygulamak maliyetlidir. Çünkü blok içi karakterlere ulaşmak gereklidir. Bloğun son karakterine *i* uzaklıktaki bir karakter için zincirde *i* blok geri gidilmelidir.

Bu sebeple önce belirgin karakter düşünülerek işe başlanır. Kaydırmaları maksimize etmek için, en sağdan en sola doğru gidilir.

$$B(i,j) = \min\left(\{i\} \cup \{i-j, 1 \le j \le i \wedge P_j = c\}\right) \qquad 4.4$$

formülü penceredeki *i* konumundaki *c* karakteri için en yüksek güvenli kaydırmayı vermektedir. Eğer pencerenin o konumunda dizgi var ise bu kaydırma 0 olacaktır. Belirgin karakterin 0 kaydırma vermediği durumlarda, pencere kaydırılır. Aksi halde iç karakterler de düşünülmek zorunda kalınır.

Algoritma sıkıştırılmış veriyi blok blok okumak sureti ile uygulanabilmektedir. 0 ya da daha fazla bloğu pencereye okuyup önceki işlemleri kaydırma yapabildiği sürece tekrarlamak mümkündür. Her okunan blok için, son karakteri, işaret ettiği blok, sıkıştırılmış verideki konumu ve uzunluğu (bu son iki eleman sıkıştırılmış dosyada yoktur ama okurken hesaplanabilir) saklanır.

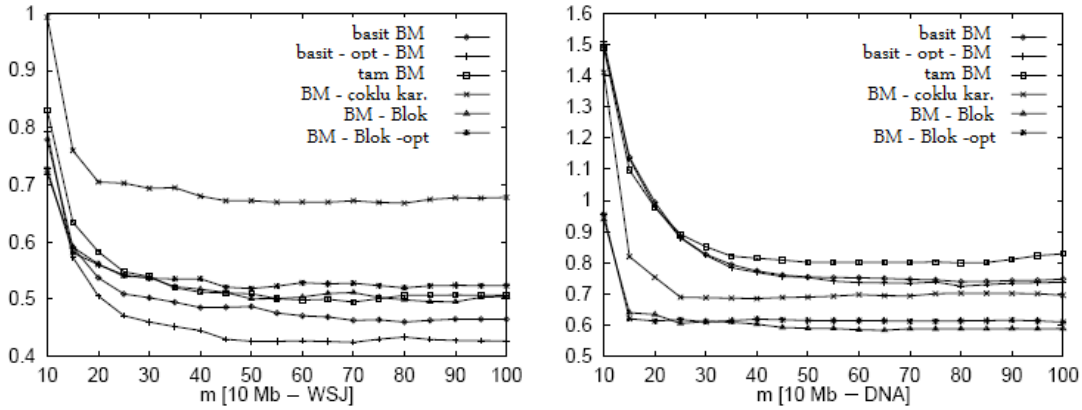
Bu durum, yukarıdaki algoritmanın bloğun dizginin tüm karakterleri ile eşlenip eşlenmediğine bakmak için bloğu açması durumunda, dizginin blok içerisinde geçmesi suretiyle mümkündür. Bu da dizginin blok uzunluğundan çok küçük olmadığı durumlarda etkilidir.

Bir nevi "döngü atlama" özelliği eklenerek algoritma üzerinde küçük bir iyileştirme yapılabilmektedir. Bu iyileştirme, yeterince blok okunana kadar kaydırmayı ertelemek yerine her okunan bloğun belirgin karakterini kullanarak kaydırma yapmaktadır. Pratikte bu, belirgin karakterleri soldan sağa sırada düşünmek anlamına gelmektedir. Çok ve küçük kaydırma kullanılır ve daha önceden saklanmış karakterleri daha az sıralar. Bu çalışmada orijinal sürüm "basit BM" ve iyileştirilmiş sürüm "basit – opt – BM" olarak adlandırılmıştır.

Bu basit metot geniş alfabeler için etkiliyken DNA gibi dar alfabe içeren metinler için etkili kaydırmalar üretmekte başarısızdır. Bir karakter yerine q karakter grubu kullanılarak kaydırma yapan çoklu karakter tekniği bu durumlarda kullanılabilir.

Çalışmada bu konu ve tam blok kaydırma üzerinde de durulmuştur. Çizelge 4.7'de bu konuda yapılan testlerin sonuçları verilmiştir. [Navarro ve Tarhio, 2000]

Çizelge 4.7 Değişik algoritmalar için sıkıştırılmış dizgi eşleme zamanları (sn)



Navarro ve Tarhio 2005 yılında daha önce yaptıkları çalışmaya paralel olarak *LZgrep* adını verdikleri ve sıkıştırılmış dizgi eşleme yapan bir yazılım üretmişlerdir. BM tekniği üzerinde çeşitli uyarlamalar yaparak, LZ78 ve LZW sıkıştırma algoritmaları ile kodlanmış olan metin üzerinde belirgin karakterleri kullanarak kaydırma yapar hale getirmişlerdir. *LZgrep* bir LZW sıkıştırma algoritması kodlayıcısı olan Unix'in *Compress* algoritması ile sıkıştırılmış dosyalar üzerinde *grep* gibi doğrudan dizgi eşleme yapabilen bir araçtır. Ayrıca *LZgrep* Unix'in *gzip* algoritması ile sıkıştırılmış dosyalarda da mevcut olan yöntemlerden daha hızlı olarak açarak arama yapmaktadır. [Navarro ve Tarhio, 2005]

4.1.6.3. LZW sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Amir, Benson ve Farach 1994 yılında LZW sıkıştırma algoritması ile sıkıştırılmış metinler üzerinde sıkıştırılmış dizgi eşleme yapan bir çalışma yapmışlardır. Yapılan bu çalışma, uyarlanabilir algoritmalar üzerine yapılan ve neredeyse en uygun olarak bilinen ilk çalışmadır. Elde ettikleri değerler, "aç ve ara" algoritma seçeneklerinden daha küçüktür.

Ayrıca bu çalışmada sıkıştırılmış dizgi eşleme algoritmalarının geliştirilmesinde kullanılacak olan yeni bir ölçüt de ortaya koymuşlardır. Bu ölçüt de "*ilave yer*"dir. Bazı

uygulamalarda sınırlı miktarda ilave yer uygundur. Bu sebeple sıkıştırılmış dosyaya ek olarak kullanılan *ilave yer* de oldukça önemlidir.

Algoritmaları diğer birçok dizgi eşleme algoritmasında olduğu gibi 2 aşamadan oluşmaktadır. Bunlar sırasıyla, dizgi önişlem ve metin tarama aşamalarıdır.

Birinci aşamada dizginin bazı durumlara cevap verecek şekilde önişleme alınması gerekmektedir. Bu durumlar;

S_1 ve S_2'nin arka arkaya gelen bloklar olduğu kabul edildiğinde,

- S_1'in dizgi öneki ve S_2'nin dizgi iç katarı olduğu,
- S_1'in dizgi öneki ve S_2'nin dizgi soneki olduğu,
- S_1'in dizgi iç katarı olduğu durumlardır.

İkinci aşama olan metin tarama aşaması ise iki bileşenden meydana gelmektedir. Bileşenler sırasıyla, sözlük oluşturma ve güncelleme bileşeni ile dizgi eşleme bileşenidir.

Sözlük oluşturulurken önek işareti, sonek işareti, iç katar işaretlerinden uygun olan konulur. Buna ilave olarak, her nod katar içerisindeki ilk sembolü de tutmaktadır. Önek işareti karşılık gelen önekin uzunluğunu, sonek işareti karşılık gelen sonekin uzunluğunu, iç katar işareti ise bu nodun gösterdiği alt katarın indislerini tutmaktadır.

Dizgi eşleme parçası bir önceki blokla biten en uzun dizgi önekinin izini tutar ve bir sonraki bloğun bu öneki genişletip genişletmediğini sorgular. Bu genişletmeler sonunda dizginin tamamen tüketildiği yerler tam eşleşme olarak işaretlenmiştir.

Amir, Benson ve Farach bu çalışmasında dizgi eşleme bölümünde KMP otomatını kullanarak sonuçları elde etmişlerdir. [Amir vd., 1994]

Kida, Takeda, Shinohara, Miyazaki, ve Arikawa 1998 yılında LZW sıkıştırma algoritması üzerine sıkıştırılmış dizgi eşleme yapan bir algoritma sunmuşlardır. Geliştirdikleri bu algoritma KMP otomatının hareketlerinin karakterinde davranan iki

fonksiyon olarak görülebilmektedir. Bu görünüm de algoritmayı sadeleştirmekte ve çoklu sıkıştırılmış dizgi eşleme işlemine yardımcı olmaktadır.

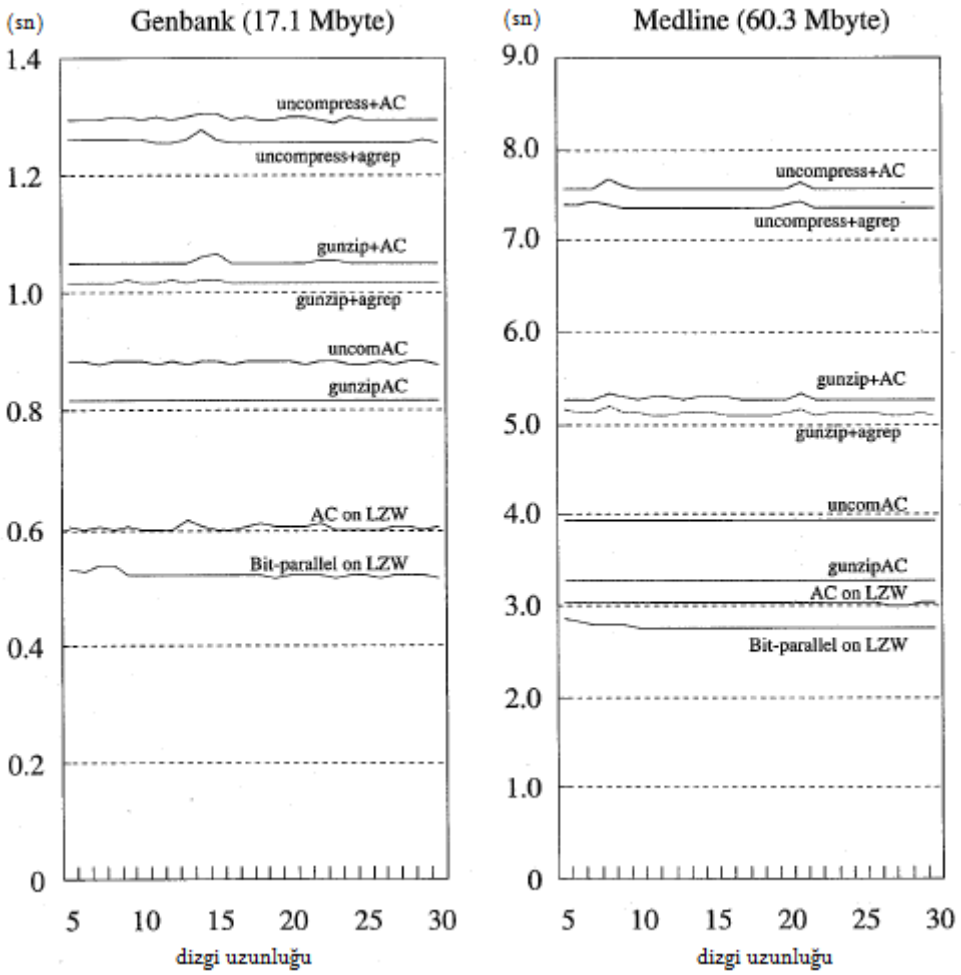
Ayrıca bu çalışmada bit paralelliğini kullanan başka bir algoritma daha sunulmuştur. Yine bu çalışmada LZW sıkıştırma algoritması, Aho – Corasick dizgi eşleme algoritması ve genelleştirilmiş sonek ağacı kullanılmıştır.

Geliştirdikleri algoritmanın dayandığı prensip LZW sıskıştırma algoritması ile sıkıştırılmış metin üzerinde AC otomatını kullanarak, bu otomatın düz metinlerdeki gibi dizgi eşleme yapmasını sağlamaktır.

Çalışmanın genelinde Amir, Benson ve Farch'ın yukarıda anlatılan algoritmalarına atıfta bulunarak, o çalışmadan farklı olarak dizgi eşlemeyi çoklu dizgiler için gerçekleştirmişlerdir. Bu çalışma çoklu sıkıştırılmış dizgi eşleme yapan ilk algoritma olması yönünden önem taşımaktadır.

Bit paralelliği yaklaşımının normal metinler ve LZW sıkıştırma algoritması ile sıkıştırılmış metinler için ayrı ayrı tanımını vererek, yaptıkları testlerde bu yaklaşımı da değerlendirmişlerdir. Yapmış oldukları testlerin bazı sonuçları çizelge 4.8'de mevcuttur. [Kida vd., 1998]

Çizelge 4.8 Arama zamanı (CPU zamanı)



Barcaccia, Cresti, De Agostine 1998 yılında yaptıkları çalışmada LZW sıkıştırma algoritması üzerinde kimlik sezgiseli kullanarak sıkıştırılmış dizgi eşleme yapmışlardır. LZW sıkıştırma algoritmasındaki standart *bir sonraki karakteri işleme* sezgiseline dayanmaktadır.

Standart LZW sıkıştırma algoritmasında kullanılan *bir sonraki karakteri işleme sezgiseli*, alt katarları, eldeki katarı soldan sağa okumak suretiyle oluşturulmaktadır. Başlangıç olarak sözlük alfabe karakterlerini içermektedir. Sözlüğe ekleme işlenmemiş katarın önekinin sözlük elemanlarından hiçbirine uymadığı zaman gerçekleşir. Örnek 4.7'de standart LZW sıkıştırma algoritmasının çözümlemesi gösterilmiştir.

Örnek 4.7: Metin: abababaaaaa

Çözümleme: a, b, ab, aba, a, aa, a

Sözlük: a, b, ab, ba, aba, abaa, aa, aaa

Kod: 1, 2, 3, 5, 1, 7, 1

Sıkıştırılmış dizgi eşleme esnasında dizgi, sözlüğe her eleman eklendiğinde güncellenmektedir. Ancak bu ekleme her zaman var olan bir sözlük elemanına tek karakter eklenmesiyle oluşur. ID sezgiseli bu özelliği kullanmamaktadır.

ID sezgiseli katar soldan sağa okundukça aynı zamanda alt katarları da öğrenmektedir. Sözlük yine başlangıçta alfabe elemanları ile doludur. ID güncelleme sezgiseli sözlüğe son eşleşen ve o an eşleşeni birleştirerek eklemektedir. O anda eşleşen sözlük işaretçisi ile gösterilir ve okuma yeni karakterden devam eder. Bu sebeple iki ardışık katarın birleşimi bir sözlük elemanıdır. Örnek 4.8'de ID sezgiseli ile çözümlemeye örmek verilmiştir.

Örnek 4.8: Metin: abababababbabb

Çözümleme: a, b, ab, ab, abab, ab, b, abb, babb

Sözlük: a, b, ab, bab, abab, ababab, "ababab", abb, babb, abbbabb

Kod: 1, 2, 3, 3, 5, 3, 2, 8, 9

Beşinci adımda "abab" elemanı sözlükte bulunduğu için yerine boş kayıt eklenir.

Sıkıştırılmış dizgi eşleme safhasında diğer LZW sıkıştırma algoritmalarında olduğu gibi dizgi ve katar arasında önek, sonek veya iç katar olma durumları değerlendirilir. Bu değerlendirmeler sonucunda bulunan eşleşmeler işaretlenir. Metin ilerledikçe sözlük ağacı gelişeceği için güncellemeler gerçekleştirilir.

Bu çalışmada geliştirilen tekniğin, daha önce LZW sıkıştırma algoritması üzerinde geliştirilen tekniklerden farklı kısmı dizgi eşleme aşaması değil, sözlük ağaçlarını oluşturma aşamasıdır. [Barcaccia vd., 1998]

Kida, Takeda, Shinohara ve Arikawa 1999 yılında yaptıkları çalışmada LZW sıkıştırma algoritması ile sıkıştırılmış metinler üzerinde sıkıştırılmış dizgi eşleme yapmak için Shift – And yaklaşımını kullanmışlardır.

Shift – And metodu birçok uygulamada klasik dizgi eşleme yapmak için oldukça verimlidir. Metot oldukça basit ve dizginin makine kelimesinden kısa olduğu durumlarda oldukça hızlıdır. Bu çalışmada makine kelimesi uzunluğu 32'den küçük olarak kabul edilmiştir.

Shift – And algoritması kısaca şu esaslara dayanmaktadır.

$P = P[1:m]$, m uzunluklu dizgi

$T = T[1:N]$, N uzunluklu metin

$k = 0, 1, \dots, N$ için

$$R_k = \{1 \leq i \leq m | i \leq k \; ve \; P[1:i] = T[k-i+1:k]\} \qquad 4.5$$

olsun. Herhangi bir $a \in \sum$ için

$$M(a) = \{1 \leq i \leq m | P[i] = a\}, \qquad 4.6$$

$$f: 2^{\{1,2,\dots,m\}} x \sum \rightarrow 2^{\{1,2,\dots,m\}}, \qquad 4.7$$

$S \subseteq \{1, \dots, m\}$, olduğu yerlerde

$$f(S, a) = ((S \oplus 1) \cup \{1\}) \cap M(a) \qquad 4.8$$

olur.

Bu fonksiyon kullanılarak $k = 1,2, \dots, m$ için R_k değerleri hesaplanır.

1- $R_0 = \emptyset$

2- $R_{k+1} = f(R_k, T[k+1])$, $(k \geq 0)$

$k = 1,2, \ldots, N$ için algoritma metnin *k.* karakterini okur. R_k değerini hesaplar. Eğer $m \in R_k$ ise $T[k-m+1:k]$ olur ki bu da dizginin metnin $k-m+1$ pozisyonunda eşlendiği anlamını taşımaktadır.

$m \leq 32$ olduğu zaman R_k kümeleri ve *M(a), m*-bit tamsayı olarak ifade edilebilir. R_k tamsayıları hesaplanır.

1- $R_0 = 0$

2- $R_{k+1} = \big((R_k \ll 1) + 1\big) \& M(T[k+1]), (k \geq 0)$

Burada '«' bit kaydırmayı, '&' ise mantıksal çarpımı ifade eder. Eğer

$$R_k \& 2^{m-1} \neq 0 \qquad 4.9$$

ise dizgi eşlenmiştir.

Örnek 4.9: T=abababbabcababc ve P=ababc

	a	b	a	b	a	b	b	a	b	c	a	b	a	b	c
a	0	1	0	1	0	1	0	0	1	0	0	1	0	1	0	0
b	0	0	1	0	1	0	1	0	0	1	0	0	1	0	1	0
a	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0	0
b	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	0
c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1*

Örnek 4.9'da Shift – And karakteristiği gösterilmektedir. * işareti o pozisyonda bir dizginin eşlendiğini belirtilmektedir.

LZW sıkıştırma algoritması üzerinde uyarlanması şu mantığa dayanmaktadır. R_k değerleri hesaplanırken metin yerine *D* sözlük elemanları kullanılır. Sözlük elemanlarının birbirinden türetilmiş olması ile bu hesaplamalar ilerletilir. [Kida vd., 1999]

Gasieniec ve Rytter 1999 yılında LZW sıkıştırma algoritması üzerinde bir sıkıştırılmış dizgi eşleme çalışması yapmışlardır. Bu çalışmada iki verimli sıkıştırılmış dizgi eşleme yapan yaklaşım sunulmuştur.

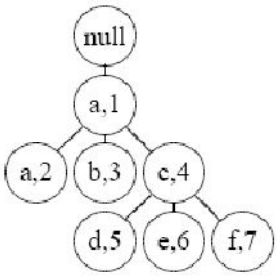
Birinci yaklaşıma *sıralı yaklaşım* adını vermişlerdir. Sıralı ve klasik KMP algoritmasını temel alan bir yaklaşımdır. Bu yaklaşımda faktörlerin (kod kelimelerinin alt kelimelerinin) keyfi seçilmiş ancak aynı uzunluklu metin ve dizgi parçaları (muhtemelen birden çok kod kelimesi içeren) ile kıyaslamaları yapılmaktadır. Bu kıyaslamalara atomik kıyaslamalar adı verilmektedir.

Her faktör için 3 eleman tutulur. *B* faktörünün adı, dizgi sonekiyle uyuşan konumu ve eşleşme uzunluğu tutulmaktadır. Bazı atomik kıyaslamalar içerisinde kalan kod kelimeleri *S*'nin parçası olarak daha sonraki atomik kıyaslamalarda aktif yer almamaktadır. Onun yerine doğrudan *B*'nin kayıtlı parçaları tutulur ve daha sonraki kıyaslamalar için de bu değerler kullanılır.

İkinci yaklaşım *paralel yaklaşım*dır. Arama safhası iki bölümden oluşur. Aday sayısının azaltılması ve kıyaslama bu bölümlerdir. Kod kelimesinde yalnızca bir aday kalana kadar tek tek elenir. Daha sonra da kullanılan veri yapıları sayesinde eşleme araması gerçekleştirilir. [Gasieniec ve Rytter, 1999]

Ho ve Yen 2002 yılında LZW sıkıştırma algoritması ile sıkıştırılmış metinler üzerine sıkıştırılmış dizgi eşleme yapan bir algoritma tasarlamışlardır.

Sözlük yaklaşımı kullanan tekniklerde girdi metni sözlük içeriğine göre bloklara ayrılmaktadır. Örneğin; "ababc" metni "a", "bab", "abc" bloklarına ayrılırsa, sıkıştırılmış metin de "a", "bab", "abc"ye atanmış kodlara eşit olacaktır. Dizgi de "aba" olursa, metin içerisinde birden fazla bloğa dağılmış halde bulunduğu görülmektedir. Bundan önceki çalışmalarda sıkıştırılmış dizgi eşleme işlemi blok içeriğini bilmeye dayanmaktadır. Bu çalışmada ise sıkıştırılmış dizgi eşleme işlemi gerek duyulan blokların açılması işlemine dayanmaktadır. Eğer bloğun açılması işlemi esnasında eşleşme imkânsız hale gelirse işlem kesilir.



Şekil 4.12 Sözlük ağacı

Bu çalışmada sözlük ağacı oluşturulurken sözlük elemanı 3 kısımdan oluşmaktadır. Şekil 4.12'de sözlük ağacı örneği gösterilmiştir. Bunlar *karakter, indeks* ve *aile*dir. *Karakter* eklenen karakteri, *indeks* de kökten ağacın o dalına kadar olan ve otomatik biçimde artan bir sayıyı tutar ve *aile* de kendinden önce gelen aileye işaret eden işaretçidir. Başlangıç durumunda sözlük boştur. Her yeni karakter okunduğunda sözlükte var mı diye bakılır. Yoksa *aile* işaretçisi ile eklenir.

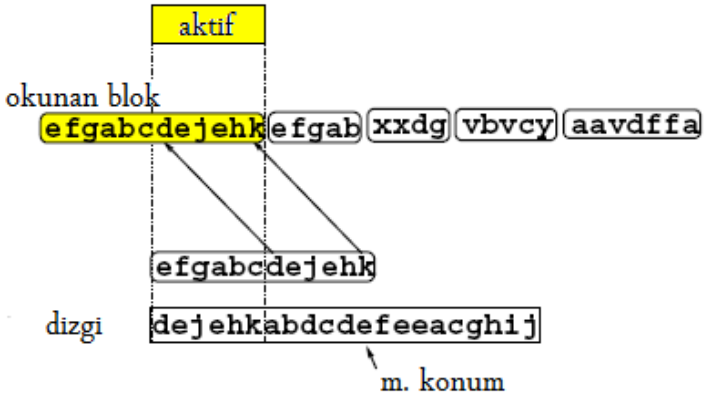
Sıkıştırılmış dizgi eşleme metnin blok blok okunması esasına dayanmaktadır. Önceki çalışmalarda sıkıştırılmış dizgi eşleme, her bloğun açılması, sözlükten okunması ve eşleşme bulunmaya çalışması olarak gerçekleştirilmiştir. İçeriği "zzz" olan bir blok, "aaa" olan bir dizgi ile eşlenilmeye çalışırsa ilk elde edilen karakter z olur. Metin [.][.][z] halini alır. Derinliği de 3 olduğu için daha fazla açmaya gerek kalmaz ve başka bloğa geçilir.

Bu da bir eşleşmenin imkânsız hale gelmesi demektir. Bu fikirden yola çıkarak, sözlük yapısı her bloğun uzunluğu elde edilebilecek şekilde düzenlenebilir.

Açma aşamasında sözlük yeniden oluşturulduğu için aşama değiştirilmeden bırakılır. Ancak ağaçlar açma esnasında yapraktan köke doğru açıldığı için sıkıştırılmış dizgi eşleme aşamasında bazı değişiklikler yapmak gerekmektedir. Bu da dizgiyi ve metni ters çeviren bir otomat kullanmak olarak ortaya çıkmaktadır.

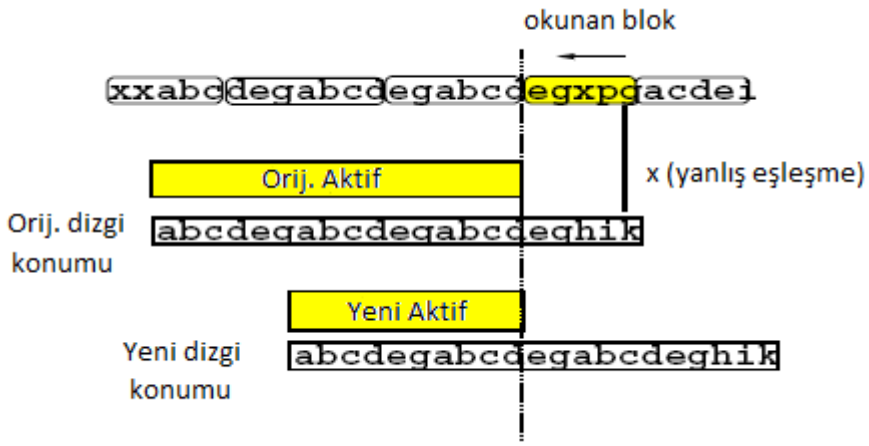
Sıkıştırılmış dizgi eşleme aşamasında bazı durumlarla karşılaşılır. Bunlardan birincisi aktif bloğun yani işlenen bloğun kısmi bir eşleşme içermediği durumdur. Şekil 4.13'te bu durum gösterilmiştir. Bu durumda aktif blok olmadığı için blok sonekiyle uyuşan en uzun dizgi öneki bulunur. En uzun önek (P') bulunduğunda işlem durur.

Eğer $P \neq P'$ ise, P' kısmi eşleşme ve blok da aktif blok olarak adlandırılır. İşlemin bir sonraki adımında dizginin kalanı ele alınır.



Şekil 4.13 Aktif blok olmayan durum

Bir diğer durum da şekil 4.14'teki durumdur. Şekilde taralı kısım var olan aktif blokları göstermektedir. İşlenen blok da "egxpg" dizgisini içerir. O anki blok ve dizgi arasında yanlış eşleşme olduğundan, bloklar değiştirilir. Bu değiştirmede KMP otomatı kullanılır.



Şekil 4.14 Aktif blok olan durum

Okunan blok işlenirken yanlış eşleşme olmadığı durumlar düşünülürse;

1- Dizgi tüketilmiş ve tam bir eşleşme oluşmuştur.
2- Blok tüketilmiştir ve tüketilen blok aktif bloklar listesine eklenir.

İşlenen blok ve dizgi uzunlukları göz önüne alınırsa;

- Blok dizgiden kısa olabilir. Bu durumda tam bir eşleşme bulmak imkânsızdır. Dolayısıyla dizginin en uzun önekinin bulunmasını gerektirir.

Dizgi bloktan kısa veya eşit olabilir. Dizgi blok içerisinde bulunabilir. [Ho ve Yen, 2002]

Tao ve Mukherjee'nin 2004 yılında yaptıkları çalışma LZW sıkıştırma algoritması üzerinde sıkıştırılmış dizgi eşleme yapmaktadır. Çalışma Amir'in 1996 yılındaki algoritmasına dayandırılmış ancak farklı olarak dizginin ilk geçtiği yeri değil, tüm geçtiği yerleri bulmuştur. Ayrıca çoklu dizgi aramayı da önceki çalışmaya eklemiştir. [Tao ve Mukherjee, 2004]

Zhang, Tao, Satya ve Mukherjee 2004 yılında LZW sıkıştırma algoritması üzerinde bir sıkıştırılmış dizgi eşleme algoritması sunmuşlardır Sundukları çalışmada metin çıkarım sistemlerini konu olarak alınmıştır.

TRS'de anahtar kelimeler, metin veritabanından toplanmaktadır ve ters indeks yapısı oluşturularak o yapı içerisinde saklanmaktadır. Bir sorgu verildiği zaman arama motoru ters indekste kelimeyi bulur. Sonra da hedef doküman seçilir ve işlenir.

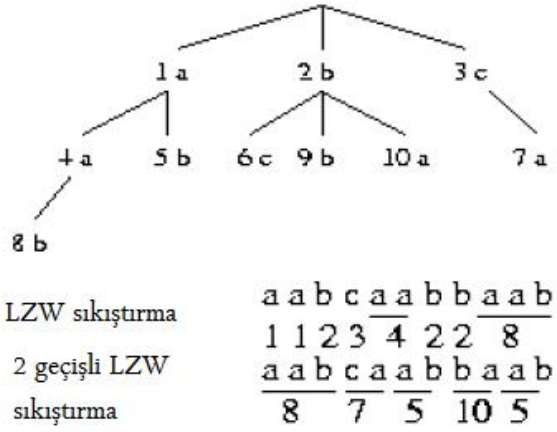
Çok büyük metin grupları için sıkıştırılmış metnin bir parçasına doğrudan erişme ya da hem sıkıştırılmış eşleme hem de metin çıkarım işlemi yapmak zordur.

Bu çalışmada LZW sıkıştırma algoritması üzerinde önemli değişiklikler yapılmıştır. Algoritma iki geçişli ve ikinci geçişte sıkıştırma yapan bir hale getirilmiştir. Birinci geçişte LZW ağaçları normal LZW sıkıştırma algoritmasında oluşturulduğu gibi oluşturulurken, ikinci geçişte ağaçlar kullanılarak sıkıştırma yapılmaktadır. Ağaçlar da sıkıştırılmış metin dosyasına eklenir. Alternatif olarak büyük sözlüklerden oluşturulmuş standart ağaçlar kullanılabilmektedir. Bu da tüm işlemi bir geçişe indirgemek anlamına gelmektedir. Örnek 4.10'da bu yaklaşıma bir örnek verilmiştir.

Sözlük boyu kodlamada kullanılacak bit sayısına göre değişmektedir. Dosya ne kadar büyükse, ağaç maliyetinin sıkıştırılmış dosya büyüklüğüne oranı da o kadar

azalacaktır. Elde edilen sonuçlar LZW sıkıştırma algoritmasına oldukça yakındır. Aralarda etiketlemeler yaparak ya da blokların başlangıç konumları belirtilerek doğrudan erişim yapılabilmektedir.

Örnek 4.10: T=aabcaabbaab



Şekil 4.15 Orijinal LZW ve 2 geçişli LZW örneği

Örnek 4.10'da "aab" dizgisi 3 kere geçmektedir. Sırasıyla "112", "42" ve "8"olarak kodlanmıştır. Eğer kodlamada 12 bit kullanıyor ise kodlamada sırasıyla 36, 24 ve 12 bit kullanılmış olur.

Bu örnekteki metin eğer 2 geçişli algoritma kullanılarak sıkıştırmaya tabi tutulursa, çıktı katarı "8.7.5.10.5" olarak kodlanacaktır. Sıkıştırma aşamasının her noktasında aynı ağaç kullanılmış olacaktır. Bu da herhangi bir noktaya herhangi bir anda erişimi mümkün kılacaktır.

Erişim için etiketler koymak gerekecektir. Bu da sıkıştırma oranını negatif yönde çok da fazla etkilemeyecektir.

Çalışmada sıkıştırılmış dizgi eşleme üzerinde çok fazla durulmamış ancak metnin herhangi bir yerine doğrudan erişilebileceğinden bahsedilmiştir. Ayrıca yapılan testlerde ağaçlar 12, 13 ve 14 bit kodlamaya göre kurulmuştur. [Zhang vd., 2004]

Tao ve Mukherjee'nin 2005 yılında yaptıkları çalışmada LZW sıkıştırma algoritması üzerinde sıkıştırılmış dizgi eşleme yapmışlardır. Sıkıştırılmış dizgi eşleme

Aho – Corasick algoritması ile gerçekleştirilmiştir. LZW ağaçları belirgin açma yapmadan yani okunan ağaçtaki katarı açılmış dosyaya eklemeden oluşturulabildiği için LZW ağaçlarından bir durum erişim tablosu oluşturulmaktadır. Daha sonra da çoklu eşleme için AC otomatı kullanılmaktadır. Algoritma Kida tarafından üretilen benzer algoritma ile kıyaslanmıştır. Elde edilen sonuçlar çok fazla sayıda dizgi kullanılmayan durumlar için algoritmanın en iyi algoritma olduğunu göstermektedir. Dolayısıyla genel dizgi eşleme uygulamaları için tercih edilebilir bir algoritmadır. [Tao ve Mukherjee, 2005]

Lee ve Huang 2008 yılında LZW sıkıştırma algoritması ile sıkıştırılmış metinlerde sıkıştırılmış dizgi eşleme yapan bir yaklaşım sunmuşlardır. Amir – Benson – Farach'ın sunmuş oldukları algoritmayı bit haritası temelli olarak uygulamışlardır. Ayrıca ABF algoritmasını var olan tüm dizgileri bulacak şekilde geliştirmişlerdir.

Yaklaşımlarında öncelikle 2 bit haritası tablosu kullanmışlardır. Birinci tablo dizgi önekleri, ikinci tablo da dizgi sonekleri için oluşturulmuştur. Bit haritası uzunluğu *m* dizgi uzunluğuna eşittir. *(i-1)*. öneke karşılık gelen bit haritası A_i ile ifade edilir. Eğer önekin o anında yine bir dizgi öneki içeriyorsa o da işaretlenir. Aynı işlemler sonekler için de gerçekleştirilir. Bit haritaları B_k ile ifade edilir. $R = A_i \otimes B_k$ işlemi (AND) yapılır. Eğer $i > 1$ olursa sınırlar içerisinde bir dizgi eşlemesi vardır.

Örnek 4.11: P=abcab ise S_1=abca ve S_2=bcab olur. $P_x = 4$, $S_x = 4$ ve R=01001 bulunur. Bu örnekte dizgi S_1'in ilk konumunda eşleşir. *R*'de 2 tane 1 olması S_1S_2 içinde iki yerde eşleşme olduğunu gösterir [Lee ve Huang, 2008].

Çizelge 4.9 Örnek 4.11 için önek bit haritaları

P_x	Önek	Bit Haritası #	Bit Haritası
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*abc*	4	00010
4	*abca*	5	01001

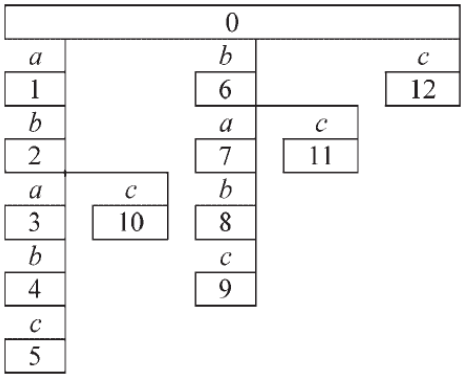
Çizelge 4.10 Sonek bit haritaları

S_x	Sonek	Bit Haritası #	Bit Haritası
1	*b*	1	00001
2	*ab*	2	00010
3	*cab*	3	00100
4	*bcab*	4	01001

Örnek 4.12: P=ababc

Çizelge 4.11 Örnek 4.12 için önek bit haritaları

P_x	Önek	Bit Haritası #	Bit Haritası
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*aba*	4	01010
4	*abab*	5	00101



Şekil 4.16 P=ababc'nin açılmış ST_p ağacı

Çizelge 4.12 Belirgin nodların bit hariları

Belirgin Nod	Bit Haritası
0	00000
2	10100
5	10000
6	01010
9	01000
10	00100
11	00010
12	00001

4.1.6.4. LZSS sıkıştırma algoritması üzerinde yapılmış sıkıştırılmış dizgi eşleme çalışmaları

Klein ve Shapira 2000 yılında LZSS sıkıştırma algoritması kullanılarak sıkıştırılmış metin veriler üzerinde sıkıştırılmış dizgi eşleme yapan bir algoritma tasarlamışlardır.

LZSS'de metin tek karakterler ya da önceden geçmiş katarları gösteren işaretçiler olarak kodlanmaktadır. Bu kodlanan işaretçiler *(off, len)* ikilisi şeklinde ifade edilirler. *off* bulunulan konumdan bir önceki geçiş yerine kadar olan karakter uzaklığı, *len* ise eşleşen katarın uzunluğunu göstermektedir. Örnek 4.13'te Standart LZSS sıkıştırma algoritmasına bir örnek verilmiştir.

Örnek 4.13: T= acdeabceabcdeaeab ise ε(T)= acdeabc(4,4)(9,3)(7,3) olur. Eğer P= bcdeaea dizgisi aranırsa, *T*'nin içerisinde *P* karakterlerinin *P*'dekinden farklı sıralarda yer aldığı gözükür. Dolayısıyla sıkıştırılmış dizgi eşleme algoritmaları daha sonra eşleşme olasılığına karşı işaretçiler tutmalıdır. Ancak bu işlem oldukça maliyetlidir.

Bunun bir çözümü, *P*'nin her karakteri için son geçiş ve son en uzun geçiş parametreleri tutulması olabilir. Örnek 4.13'te 2. konumdaki "c" ele alınsın. Mevcut konum da (4,4) ikilisinin hemen ardı olsun. O ana kadar T'nin ilk 11 karakteri işlenmiştir. T=acdeabceabc halindedir. Tek "c" karakteri için son geçtiği yer 11. konumdur. En uzun eşleşme ise 2. konumdaki "cd"dir. Bu değerler tutulur ve metin üzerinde ilerledikçe bu değerlerin tutulduğu tablo güncellenir.

Buna rağmen bu algoritma *P* dizgisinin geçtiği tüm konumları bulamaz.

Örnek 4.14: Eğer T=cabeabecabdfabd ise ε(T)=cabe(3,3)(7,3)df(4,3) olur. P=fabd olursa, "a" ile başlayan alt katarın 2. konumda geçitiği yer işaretlenecek ama (3,3) ile ifade edildiği 5. konumdaki ikili tarafından bir sonraki adımda silinecektir. Bunun yanında (7,3) ikilisi de "ab" içeren tam dizgiye sahip olacaktır. Dolayısıyla ne 9. ne de 13. konumda (4,3) ikilisini kullanarak "ab" geçişleri hesaplanamayacaktır. 12. konumdaki eşleşme gözden kaçırılacaktır.

Bir karakter katarı işlenirken, *(off,len)* işaretçileri ile metnin ilerleyen bölümlerinde o katara işaret edilip edilemeyeceği bilinemez. Bu da işaretçilerin yerini değiştirme fikrini doğurmuştur. İşaretçileri işaret ettikleri katarın yanına almak ve geriye doğru uzaklık gösteren parametre yerine ileri doğru uzaklık gösteren parametre kullanmak sorunun çözümü için düşünülen bir adımdır.

Örnek 4.15: T= dabcdeabcb LZSS sıkıştırma algoritması ile ε(T)=dabcde(5,3)b olurken, algoritmanın değiştirilmiş halinde ε(T)=d(5,3)abcdeb şeklini alır. Ancak sıkıştırılmış dizgi eşleme işlemi için *(off,len)* ikilisinin hangi katara denk geldiği hemen bilinmelidir. Bu sebeple işaretçi, işaret ettiği katarın hemen arkasına yazılmaktadır. O zaman ε(T)=dabc(2,3)deb halini alır.

Ancak üst üste gelen katarlar yüzünden bu yöntem de başarısız olmaktadır.

Örnek 4.16: T=abcdeabcddea ve LZSS sıkıştırma algoritması ile sıkıştırılmış karşılığı ε(T)=abcde(5,4)(6,3) olur. Yukarıdaki fikirle metin ε(T)=abcd(1,4)e olarak kodlanmaya başlar. *T*'deki "dea" katarına ulaşıldığı zaman (3,3) ikilisini *T*'nin 6. konumundaki "a"nın yanına koymalıdır. Ama bu "a" metinde daha önce de geçmiş ve dolayısıyla (1,4) çifti ile kodlanmıştır. Bu da (3,3) çiftinin nereye konacağını belirsizleştirir.

Bir ihtimal üst üste gelen işaretçilerden kaçınmaktır. Bu durumda en uzun eşleşmeler yeri geldiğinde göz ardı edilebilir. ε(T)=abcd(2,3)ea(3,3). Bu da sıkıştırma oranında önemli kayıp nedeni olacaktır.

Buna alternatif sunulan bir çözüm olarak gerekli olduğu durumlarda *(off,len)* ikilisi *(off,len,slide)* üçlüsü ile değiştirilebilir.

Burada *slide,* orijinal *(off,len)* ikilisinin kaydırılacağı karakter uzunluğunu göstermektedir. ε(T)=abcd(1,4)e(3,3,1) halini alır. Burada (3,3,1) üçlüsü, orijinal (3,3) ikilisinin "e"yi takip eden karakterden sonra konulması gerektiğini belirtir. Bu durumda "dea"ya işaret eder.

Sıkıştırılmış dizgi eşleme için son uyarlama ile sıkıştırılmış metinde bir işaretçi görülürse, içeriği okunmalıdır. Eğer bu içerikte dizgi ile eşleşen *len* uzunluğunda bir alt

katar var ise, dizgi ile daha sonra da karşılaşılabileceği, bunun da ilgili alt kataın ve konumlarının hatırlanma zorunluluğu olduğunun unutulmaması gerekmektedir. Pratikte bunun için bir dairesel tampon kullanılmaktadır.

Sıkıştırılmış metin sıra ile okunmaya başlar. Buna paralel olarak da tampon da okunur. Okunan eleman tek bir karakter, bir ikili veya bir üçlü olabilir.

Bir işaretçi ile karşılaşıldığı zaman ilgili karakterler tampona alınır. Tamponun başına da konumları eklenir. Bu konumlar ve değerler yardımı ile sıkıştırılmış dizgi eşleme işlemi gerçekleştirilir.

Çizelge 4.13 T=xbcyxbcabcdeabcdddea ve P=ddea örnek metinleri için üretilmiştir. Son uyarlama ile sıkıştırılmış hali ε(T)=xbc(1,3)yabcd(1,4)e(3,3,1) olur. [Klein ve Shapira, 2000]

Çizelge 4.13 Sıkıştırılmış eşleme esnasında tampon durumu

okunan	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
tampon								a			d	e	a			d	d	e	a
sonraki					8								16						

4.2. Üzerinde Sıkıştırılmış Dizgi Eşleme Yapılan Dönüşüm Teknikleri

4.2.1. BWT dönüşüm tekniği üzerinde yapılmış olan sıkıştırılmış dizgi eşleme çalışmaları

Adjeroh, Mukherjee, Bell, Powell ve Zhang 2001 yılında yaptıkları çalışmada BWT dönüşüm tekniği kullanılarak sıkıştırılmış metinler üzerinde tam ve *k*- dereceli yaklaşık dizgi eşleme yapan algoritma sunmuşlardır.

Çalışmada öncelikle BWT dönüşüm tekniğinin bir tanımı ve örneği verilmiştir. Daha sonra ise BWT dönüşüm tekniği üzerinde sıkıştırılmış dizgi eşlemenin genel yaklaşımı, yaklaşık dizgi eşleme ve deneysel sonuçlar çalışmada yer almıştır.

BWT dönüşüm tekniği tabanlı sıkıştırma genelde 4 aşamadan oluşmaktadır.

1- BWT dönüşümleri
2- Daha iyi sıkıştırma için öne alma kodlama
3- MTF tarafından üretilen çıktının RLE kodlanması
4- RLE çıktısı üzerinde entropi kodlama metotları (Huffman ve aritmetik kodlama gibi) ile değişken uzunluklu kodlama

Bu çalışmada üretilen yaklaşım, BWT dönüşüm tekniğinin girdi metinleri üzerinde dönüşüm işleminin parçası olarak yaptığı anlamsal sıralamayı gözlemlemektir. Kod çözücü sıralanmış içerik açısından sadece kısıtlı bilgiye sahiptir ama iki sembol arası olası bir eşleşmeyi bulmak için bunu genişletmek mümkündür. Daha sonra bu eşleşen kısım açılarak dizgi eşleşmenin devam edip etmediği görülmektedir. Çalışmada tanımlar *F*, *L* ve *V* dizileri üzerinde verilmiştir ve MTF öncesi BWT dönüşüm tekniği çıktısı ele alınmıştır.

T metin katarı üzerindeki ikililer (bi-gram) elde edilir. *F* ve *L* sırasıyla *T* ve *P* dizgisinin bi-gramlarını (Q_2^T ve Q_2^P) içerir. Bu bi-gramlar en az iki amaç için kullanılabilir.

- **Ön Filtreleme:** Olası bir eşleşme için $Q_2^T \cap Q_2^P$ kümesi içindeki bi-gramlar düşünülür. Eğer kesişim boş ise, dizginin metinde geçmediği ve daha fazla açma yapmaya gerek olmadığı anlamını taşır.
- **Yaklaşık Dizgi Eşleme:** Bi-gram yaklaşımı *q*-gram yaklaşımı olarak genişletilebilir ve $Q_q^T \cap Q_q^P$ işlemi yapılır. İkinci aşamada bu kesişim *k*-dereceli yaklaşık eşleme için yorumlanır.

Örnek 4.17: T=abraca, P=rac olursa L=caraab ve F=aaabcr elde edilir. F ve L kullanılarak $Q_2^T = \{ac, ab, br, ca, ra\}$ ve $Q_2^P = \{ac,\ ra\}$ elde edilir.

İkisinin kesişiminde de $\{ra, ac\}$ kümesinin kesişim kümesi içerisinde yer aldığı görülmektedir. Tam dizgi eşleme için "ac" elenecektir ve bu sebeple T içinde "ra" geçen yerler aranacaktır. Çünkü her eşleşme "ra"yı içermelidir. P=abr olursa kesişim $\{ab, br\}$ olacaktır ve F içerisinde "a" ile başlayan potansiyel başlama noktaları elenecektir.

Bu çalışmadaki yaklaşık dizgi eşleme yaklaşımının iki aşaması mevcuttur. Birinci aşamada uygun boyutlu *q*-gramlar kullanarak filtreleme işlemi yapılır. Sonuçta elde edilen potansiyel eşleşmelere konumlanılır. İkinci aşamada ise bu eşleşmeler doğrulanır. Bu doğrulama adımı genelde yavaştır ama metnin küçük bir kısmını içerir. Dolayısıyla tam performans bulunan potansiyel eşleşme sayısına bağlıdır. [Adjeroh vd., 2002]

5. GELİŞTİRİLEN SIKIŞTIRILMIŞ DİZGİ EŞLEMEYE UYGUN SIKIŞTIRMA ALGORİTMASI

Bu tezde geliştirilen algoritma yarı statik bir algoritmadır ve geliştirilen algoritmanın ismi *"Etiketlenmiş Kelime Tabanlı Sıkıştırma Algoritması"*dır. Algoritma ilk geçişinde sözlük oluşturur. İkinci geçişte ise sıkıştırma işlemi gerçekleştirilir. Geliştirilen algoritma yarı statik olduğu için kod çözücünün açma esnasında kullanılan sözlüğü de bilmesi gerekmektedir. Dolayısıyla sıkıştırma aşamasında kullanılan sözlük de sıkıştırılmış dosya içine yazılmaktadır. Sıkıştırılmış dosya sıkıştırılmış metin ve sözlük olmak üzere iki ayrı parçadan oluşmaktadır.

Bununla beraber geliştirilen algoritma sıkıştırılmış dizgi eşleme yapmaya da izin vermektedir. Verilen dizgi, öncelikle sıkıştırılmış verinin içindeki sözlük kullanılarak sıkıştırılır ve verinin ikinci parçası olan sıkıştırılmış veri kısmında aranır. Bulunan konumlar tam dizgi eşleme sonuçlarıdır. Ancak dizgi eşleme sadece kelime tabanlı olarak yapılabilmektedir. Kelime tabanlı yapılması birden fazla kelimenin oluşturduğu karmaşık dizgilerin aranmasını engellememektedir. İlave olarak sıkıştırılmış dizgi eşleme yaparken herhangi bir kısmi açma yapmak gerekmemektedir. Dolayısıyla sıkıştırılmış dizgi eşleme zamanı oldukça düşüktür. Ayrıca sıkıştırılmış dizgi eşleme yaparken dizgi eşleme algoritmasının başarımını düşürecek bir dezavantaj oluşturmamaktadır.

5.1. Algoritma Detayları

Algoritmayı daha detaylı olarak incelersek, temelde LZW sıkıştırma algoritması sözlük oluşturma yapısına benzediği görülür. LZW sıkıştırma algoritmasındaki gibi daha önceden geçmiş olan karakter grubuna bir karakter eklenip, sözlüğe eklenmesi esasına dayanmaktadır. Ancak sözlüğe eklemede kelime sonları önceden belirlenmiş olan ayraç karakterler ile belirlenmektedir. Bu yaklaşım ile de belirlenmiş olan ayraç karakterlerle biten kelimeler belirlendiğinden, algoritma kelime tabanlı sıkıştırılmış dizgi eşleme için elverişli olmaktadır.

Algoritmanın yarı statik olmasının sıkıştırılmış dizgi eşlemenin yanında sağladığı bir diğer avantaj da sıkıştırılmış bir veri üzerinde istenilen bir konumdan kısmi açma yapabilmesidir. Oluşturulan sözlüğün ve bu sözlüğe göre kodlanan karakter gruplarına karşılık gelen özel kodlar sayesinde metin istenen bir yerinden istenildiği kadar açılabilmektedir.

Sıkıştırma algoritmasında kullanılan sözlüklerin oluşturulması LZW sıkıştırma algoritmasındakine benzerlik gösterse de temelde farklıdır.

Önce benzerlikleri belirtmek gerekirse, sözlüklerde var olan karakter gruplarına bir karakter eklenerek yeni sözlük elemanının oluşması en büyük benzerliktir.

Başka bir benzerlik de karakter gruplarının sözlük oluşturma esnasında ağaç veri yapılarının kullanılmasıdır. Ağaç veri yapısını kullanmak sözlükte, aranacak katarın yerini belirlemede en etkin yöntemlerden biridir. İkinci geçişte ise ele alınan katarın yine ağaç üzerinde arandıktan sonra elde edilen kod karşılığı kodlanmak suretiyle katar sıkıştırılmış olarak elde edilmektedir.

Algoritmanın LZW sıkıştırma algoritmasından farklılaştığı ilk nokta kod kelimesi oluşturulurken kullanılan bit sayısıdır. LZW sıkıştırma algoritması kodları 9 – 10 – 12 bit gibi isteğe bağlı bit sayısı kullanan değerler ile oluşturulurken, geliştirilen algoritma 2 bayt dolayısıyla 16 bit kodlama esasına standart bir şekilde dayandırılmıştır. 10 bit kodlama kullanılan LZW sıkıştırma algoritmasında 2^{10} olası sözlük öğesi kodlanırken, geliştirilen algoritmada bu sayı 2^{16} bit yani 65536'dır. Ama 65536 sözlük yerinin hepsini kullanmak mümkün olmayacaktır. Bunun nedeni sıkıştırılmayan karakterlerin sıkışan karakterlerden ayrılması için kullanılabilecek bazı sözlük yerlerinin göz ardı edilmesidir.

Algoritmada kullanılacak mümkün sözlük öğesi sayısı, metin içerisinde geçen farklı karakter sayısına bağlıdır. Örneğin bir metinde 100 farklı karakter geçiyor ise, kullanılabilecek sözlük öğesi sayısı

$$2^8.2^8 - 100.2^8 = 39936$$

olur.

Kodlama esnasında ilk geçişte bu farklı karakter sayısı bulunmaktadır. Aynı zamanda bu geçişte karakterler indeksi 0'dan başlayan bir dizide haritalanır.

Örneğin metnin ilk karakteri *a* ise *mapping[0]= 'a'* yapılır. İkinci karakter *b* ise *mapping[1]= 'b'* olur. Üçüncü ele alınan karakter tekrar *a* ise dizide daha önceden yer aldığı için herhangi bir işlem yapılmaz ve bir sonraki karaktere geçilir. Bu karakterin de *c* olduğu düşünülürse, *mapping[2]= 'c'* yapılır.

Bu işlemin metin sonuna kadar yapılması ile elde edilen en büyük indeks değeri metinde yer alan farklı karakter sayısını vermektedir. Ayrıca daha sonra sıkıştırma esnasında bir karakterin bu dizi içerisindeki indeksi kullanılacağından bu diziye erişimin kolay olması için ters haritalama dizisi de kullanılmaktadır. Bu kullanılan ikinci dizide hangi harfin hangi indekse karşılık geldiği belirtilir.

Örneğin *mapping[0]= 'a'* ise *a*'nın ASCII değeri olan 97 numaralı indekste haritalama dizisinin indeksi tutulur. *inv_mapping[97]= '0'* olur. Dolayısıyla okunan bir karakterin ASCII kodu kullanılarak önce ters haritalama dizisinden haritalama indeksi bulunur ve o indeks kodlanır. Bu işlem de sıkıştırılamayan karakterler için haritalama dizisinin dolaşılmasını engellemekte ve dolayısıyla hız ve işlem maliyetini oldukça düşürmektedir.

Kodlama esnasında orijinal karakterler yerine indeksler kullanıldığı için bu indeksleri kod çözücünün de bilmesi gerekmektedir. Dolayısıyla haritalama dizisindeki karakterler, başlarında metin içerisinde kaç tane farklı sembol olduğunu belirten bayt ile birlikte sırasıyla sıkıştırılmış verilerin başına yazılmaktadır.

Eğer *n* tane farklı karakter varsa, oluşturulan ilk sözlük yani dizilerin dosyaya yazılmasıyla oluşan yer maliyeti 1 bayt *n* sayısını belirtmek üzere ve de arkasından gelen *n* tane karakterin *n* bayt maliyeti olmak üzere toplamda *n+1* bayt ilk sözlük maliyeti (M_1) vardır.

$$M_1=n+1 \qquad 5.1$$

Geliştirilen algoritmanın ikinci farkı kelime sonlarını temel almasıdır. Bunun için kelime sonlarını belirtmesini istediğimiz ayraçlar (boşluk, enter vb.) belirlenmektedir.

Kodlayıcı ilk geçişte bunları göz önüne alarak sözlük ağacını oluşturur. Eğer ele alınan katar sözlükte var ise bir karakter daha okunur. Bu okunan karakterle beraber katar sözlükte yok ise hepsi sözlüğün sıradaki elemanına eklenir. Eğer eldeki katarın son harfi bu ayraçlardan biri ise ve katar da sözlükte var ise yeni karakter oku – katara ekle kuralı işlenmemektedir. Yeni bir karakter okunmaz ve sözlüğe herhangi bir ekleme yapılmaz. O anda kodlayıcının bulunduğu yer bir kelime sonudur ve ağaca ekleme işleminin orada durdurulması gerekmektedir. Dolayısıyla bir sonraki kelimenin başına konumlanır ve en baştan gerekli işlemler buradan başlar. Örnek 5.1'de kelime sonunda sözlüğe eklemenin yapılmamasına bir örnek verilmiştir.

Örnek 5.1: T="Trakya_" (Burada "_" karakteri boşluğu simgelemektedir) Örneğin açıklama basitliği açısından sözlük öğelerinin ağaçlarda değil *Dict[39936][40]* isimli dizide tutulduğu varsayılsın. İndeks numaraları rastgele seçilmiştir.

Dict[45]= "Trak",

Dict[178]= "Traky",

Dict[1020]= "Trakya",

Öncelikle "T" karakterinden başlayarak eldeki katarlar sözlüklerde aranır. En son eşleşen katar olan "Trakya" bulunur. Ancak "Trakya_" sözlükte yoktur. Dolayısıyla sözlüğün sıradaki indeksine eklenir.

Dict[5150]= "Trakya_"

Bu katar sözlüğe eklendikten sonra metnin herhangi bir yerinde bir "Trakya_" daha geçtiği varsayıldığında, kodlayıcı sözlüğün 5150 indeksinde "Trakya_"a rastlayacaktır. Normalde bir karakter daha okunup, sözlüğe o şekliyle eklenmesi gerekirken, "_" karakterinden bir kelime sonu olduğu anlaşılır ve herhangi bir ekleme yapılmaksızın sıradaki kelimeye geçilir.

Kodlamanın ilk geçişinde elde edilen karakterler ve sözlük ağaçları ikinci geçişte sıkıştırma amaçlı kullanılmaktadır. Sıkıştırma prensibi eldeki katarın oluşturulmuş sözlük ağacında yer alan en büyük parçalarla ifade edilmesidir. Bu durumda dikkat

edilmesi gereken husus sıkıştırılmış katar ile sıkıştırılamamış karakterlerin fark edilmesidir. Sıkıştırılamayan her karakter haritalama dizisindeki indeksi ile kodlanır. Katarlar ise iki baytla kodlanırken, ilk bayt değerinin farklı karakter sayısından küçük olmaması gerekmektedir. İkinci bayt için herhangi bir sınırlama yoktur. Örnek 5.2'de eş anlamlı kodlamaya örnek verilmiştir.

Örnek 5.2: T="Trakya_Üniversitesi_..." ve n=100

Örnek 5.1'deki verilerden de yararlanarak, öncelikle "Trakya_" katarı karşılığı olan 5150 indeksi iki bayt ile ifade edilecek halde yazılmalıdır. Yazılırken kullanılan formül ise (*N* de sözlük indeksini göstermek üzere);

İlk bayt değeri=*N*/256 5.2

İkinci bayt değeri=*N mod* 256 5.3

olarak bulunur.

Buradan yola çıkarak;

İlk bayt değeri=5150/256=20

İkinci bayt değeri=5150%256=30 bulunur.

"Trakya_" katarı yerine çıktı ikilisi olarak 20,30 bayt değerlerine sahip karakterler çıktı dosyasına yazılır.

Ancak burada karşılaşılacak en büyük sorun ise herhangi bir işaretleme konulmaması, dolayısıyla da sıkıştırılmış katar ile sıkıştırılmamış karakterin karışması durumudur. Örneğin;

mapping[20]='Ü'

mapping[30]='n'

Bu durumda “Trakya_Üniversitesi_...” kataın sıkıştırılmış karşılığı ”20,30,20,30…” şeklinde başlamaktadır. Bu da kod çözücünün açma esnasında yanlış yorumlar yapmasına sebep olacaktır. Dolayısıyla ilk bayt üzerinde o ikilinin sıkıştırılmış katara karşılık geldiğini belli edecek bir değişiklik yapmak gerekmektedir. Bu da ilk bayt değerini farklı karakter sayısından 256’ya kadar bir değerle ifade etmek anlamına gelmektedir.

Örneğin *n*=100 için yukarıdaki denklemler:

İlk bayt değeri=*N*/256+n 5.4

İkinci bayt değeri=*N mod* 256 5.5

olarak değiştirilir.

Elde edilen ilk bayt değerine eklenen n sayısı haritalama dizisinin en yüksek indeksidir. 100 farklı karakter için 0-99 arası değerler doludur. Birinci baytın alabileceği her bir değer için ikinci bayt 256 farklı değer alabilmektedir. O halde kodlamada kullanılabilecek ağaç sayısı (*K*);

$$K = (256 - n)x256 \quad 5.6$$

formülü ile bulunmaktadır.

Bu durumda yukarıdaki örnek “120,30,20,30” şeklinde kodlanacak ve eş kodlama ortadan kalkacaktır.

Ancak iki bayt ile ifade edilecek en yüksek değerin 65535 olduğu bilindiğine göre ve ilk geçişte karakter sayısı tam bilinmeden eklemeler yapıldığına göre sözlük indeksini 65535 noktasından geriye doğru vermek algoritmayı yukarıdaki işlemlerden kurtaracaktır. Bu durumda indekslerin bayt karşılıklarını hesaplarken *n* sayısı ile toplamak gerekmeyecek, indeks sayısı doğrudan eklenmiş halini verecektir.

Geliştirilen sıkıştırma algoritmasının LZW sıkıştırma algoritmasından farklılıklarından bir tanesi de başlangıç durumunda oluşan ağaçların karakter sayılarıdır. Örneğin LZW sıkıştırma algoritması herhangi bir alfabe içermemekte ve gördüğü ilk

harften itibaren ağaca eklemektedir. Bu da örneğin 12 bit kodlamada bir karakter için 8 bit yerine 12 bit kodlanması anlamına gelmektedir.

Geliştirilen bu algoritmada ise ağaçlara eklemeler iki karakteri geçtiği anda yapılmaya başlanmaktadır. Bunun iki nedeni vardır. Birinci neden iki karakterden küçük sözlük elemanlarının, iki bayt kodlamalı bu algoritmada herhangi bir kodlama kazancına neden olmamasıdır. İkinci ve daha önemli olan sebep ise kısıtlı sözlük elemanlarının tekliler ve ikili karakter gruplarıyla doldurulmasının engellenmesidir. Örneğin "Trakya_" katarının adım adım incelendiğinde, "T" ve "Tr" katarlarının sözlüklerde yer almaması sağlanmalıdır. Sözlüğe ilk eklenen katar "Tra" olmalıdır.

Bunu sağlamak için öncelikle sözlüğe eklenmesi gereken ikililer işaretlenmelidir. Örneğin "Trakya_" kelimesi ilk kez geçtiğinde sözlükte "Tr" aranacak ancak bulunamayacağı için sözlüğe eklenmesi gerekecektir. Onun yerine ilk kez geçtiğinde "Tr" ikilisi işaretlenmeli, kelime ikinci kez geçtiğinde "Tr" işaretli olacağı için kelimenin daha önce geçtiği anlaşılmalı ve "Tra" ağaçlarda uygun yere yerleştirilmelidir.

Bu iki karakter işaretleme için iki boyutlu bir dizi kullanılabilir. "Tr" ikilisi ilk defa geçtiği zaman (T=84, r=114) *kontrol_dizi[84][114]=1* yapılır. Böylece dizi değerinin 1 olup olmadığını kontrol ederek ikilinin geçip geçmediği anlaşılmaktadır.

Buradan bir genelleme yapılmak istenirse herhangi bir *k* uzunluklu katarın herhangi bir öneki başka bir katar ile önek uyuşması yoksa, *(k-1)* tekrarda sözlüğe yazılması anlamına çıkarılabilir. Örnek 5.3'te bu genellemeye bir örnek verilmiştir.

Örnek 5.3: "Trakya_" (k=7) kelimesine benzeyen herhangi bir kelime yoksa ilk tekrarda "Tr" işaretlenir. İkinci kez kelime metinde geçtiğinde önce *kontrol_dizi[T][r]==1* kontrolü yapılır. İşaretlenmiş olduğu görüldükten sonra "Tra" sözlüğe eklenir. Üçüncü kez geçtiğinde ise "Tr"nin işaretli olduğu ve "Tra"nın ağaçta yer aldığı görülür. "Trak" sözlüğe eklenir. Bu şekilde 6. kelime geçişinde 7 harfli "Trakya_" ağaçlarda tamamen yer alır.

Bununla beraber arada bir de “Trakyalı_” kelimesi geçerse o zaman, önekler benzeştiği için bir ekleme de bu kelime üzerinden yapılacağından 5. geçişte 7 harfli “Trakya_” sözlüğe eklenmektedir.

İlk geçişte hem sözlük oluşturulduğu hem de karakter sayıları hesaplanması ve haritalama yapıldığı için ağaçlar alabilecekleri en büyük indeksten geriye doğru sıralanır. Bunun nedeni karakter sayısı bulunmadan eklemelerin başlamasıdır. Örneğin ağaca ilk eklenen katarın indeks değeri 65535, ikincinin 65534 olur. Bu şekilde de indeks (256-n)x256 değerine ulaşana kadar düşürülür. Her farklı karakter görüldüğünden n değeri bir artırılmaktadır.

Geliştirilen algoritmanın LZW sıkıştırma algoritması temelinden farklılaşması konusunda eklenmesi gereken son adım da ağaca eklendikten sonra eklenen katarın tamamen tampon bölgeden boşaltılması ve tampon bölgeye yeni iki karakter alınmasıdır. Örnek 5.4’te ve çizelge 5.1’de “Trakya_” kelimesinin LZW sıkıştırma algoritmasında kodlanması adım adım gösterilmiştir.

Örnek 5.4: “Trakya_” kelimesi LZW sıkıştırma algoritmasında çizelge 5.1’deki gibi kodlanır;

Çizelge 5.1 LZW’de “Trakya_” kelimesi kodlanışı

Girdi Karakterleri	Çıktı Kodu	Yeni Kod Değeri ve Atanan Katar
Tr	“T”	256= “Tr”
a	“r“	257= “ra”
k	“a”	258= “ak”
y	“k”	259= “ky”
a	“y”	260= “ya”
_	“a”	261= “a_”

Oysa yeni geliştirilen bu algoritmada ise kodlama çizelge 5.2’deki gibi olmaktadır.

Çizelge 5.2 Geliştirilen algoritmada “Trakya_” katarının ürettiği çıktılar

Girdi Karakterleri	Çıktı Kodları	İşaretlenen Dizi Elemanı
Tr	“T”, “r”	Kontrol_dizi[84][114]=1
ak	“a”, “k”	Kontrol_dizi[97][107]=1
ya	“y”, “a”	Kontrol_dizi[121][97]=1
_	“_”	

Kodlama aşamasında sözlük elemanlarının indeks kodları ile karşılık gelen katarların yer değiştirdiği ikinci geçiş nispeten daha kolay bir aşamadır. Bu aşamada sırasıyla ilk karakterden itibaren metin tampon belleğe okunur. Tampon bellekteki katar sözlükte eşleşmeyene kadar bu işlem devam eder. Eşleşmeyen katar bulunduğunda son eşleşenin indeks değeri formül 5.4 ve formül 5.5 ile hesaplanmış ikili bayt sırasına dönüştürülür ve sıkıştırılmış dosyaya yazılır. Dosyaya yazma işleminden sonra son eşleşenin bulunduğu tampon boşaltılır ve sıradaki karakterden işlem yeniden yapılır. Eğer tampon içerisinde bir karakterlik bir katar varsa o karakterin haritalama dizisinden indeks karşılığı tek bayt halinde kodlanır. Eğer tampon içerisinde iki baytlık bir karakter dizisi varsa o zaman ilk karakterin haritalama dizisinden karşılığı kodlanır, tampondaki ikinci karakter birinci karakter üzerine kaydırılır ve yeni bir karakter eklenir.

Bunun yanında kodlama esnasında ayrıca kelime sonu kontrolü yapmaya gerek yoktur. Çünkü ağaca ekleme fonksiyonları ayraç gördüğü zaman işlemi bırakmıştır ve ayraçtan sonraki karakterin eklemesi yapılmamıştır. Bu nedenle en alt yapraklarda her zaman ayraçlar yer almaktadır. Bu da kelime sonlarının otomatik olarak ve kontrol edilmeksizin ağaçların son yapraklarında bitmesi anlamına gelmektedir.

Kodlama safhasının iki aşaması da (sözlük oluşturma ve kod karşılıklarını yazma) örnek 5.5’te ve çizelge 5.3’te adım adım gösterilmiştir. Örnekte“_” karakteri kelime ayracı olarak kabul edilen boşluk karakteri yerine kullanılmıştır.

Örnek 5.5: T=”kartal_kalkar_dal_sarkar_dal_sarkar_kartal_kalkar%”

1. Aşama (Sözlük oluşturma)

Çizelge 5.3 İlk kelime için örnek metinden sözlük oluşturma

Okunan Karakter	Tampon	İşaretlenen İkili	Haritalama dizisi	Sözlükte var mı?	Sözlük
k	k	--	Mapping[0]= ‘k’	--	--
a	ka	[k][a]	Mapping[1]= ‘a’	--	--
r	r	--	Mapping[2]= ‘r’	--	--
t	rt	[r][t]	Mapping[3]= ‘t’	--	--
a	a	--	--	--	--
l	al	[a][l]	Mapping[4]= ‘l’	--	--
_	_	--	Mapping[5]= ‘_’	--	--

k	k	--	--	--	--
a	ka	[k][a]=işaretli	--	--	--
l	kal	--	--	--	65535-kal
k	k	--	--	--	--
a	ka	[k][a]=işaretli	--	--	--
r	kar	--	--	--	65534-kar
_	_	--	--	--	--
d	d	--	Mapping[6]= ‘d’	--	--
a	da	[d][a]	--	--	--
l	l	--	--	--	--
_	l_	[l][_]	--	--	--
s	s	--	Mapping[7]= ‘s’	--	--
a	sa	[s][a]	--	--	--
r	r	--	--	--	--
k	rk	[r][k]	--	--	--
a	a	--	--	--	--
r	ar	[a][r]	--	--	--
_	_	--	--	--	--
d	d	--	--	--	--
a	da	[d][a]=işaretli	--	--	--
l	dal	--	--	--	65533-dal
_	_	--	--	--	--
s	s	--	--	--	--
a	sa	[s][a]=işaretli	--	--	--
r	sar	--	--	--	5532-sar
k	k	--	--	--	--

a	ka	[k][a]=işaretli	--	--	--
r	kar	--	--	Evet(65534)	--
_	kar_	--	--	--	65531-kar_
k	k	--	--	--	--
a	ka	[k][a]=işaretli	--	--	--
r	kar	--	--	Evet(65534)	--
t	kart	--	--	--	65530-kart
a	a	--	--	--	--
l	al	[a][l]=işaretli	--	--	--
_	al_	--	--	--	65529-al_
k	k	--	--	--	--
a	ka	[k][a]=işaretli	--	--	--
l	kal	--	--	Evet(65535)	--
k	kalk	--	--	--	65528-kalk
a	a	--	--	--	--
r	ar	[a][r]=işaretli	--	--	--
%	ar%	--	Mapping[8]= '%'	--	65527-ar%

Örnek 5.5'teki metin ilk geçişte işlendikten sonra elde edilen sözlük ve haritalama dizisi sırasıyala çizelge 5.4 ve çizelge 5.5'te verilmiştir.

Çizelge 5.4 Oluşturulan sözlük

İndeks	Katar
65535	kal
65534	kar
65533	dal
65532	sar
65531	kar_
65530	kart
65529	al_
65528	kalk
65527	ar%

Çizelge 5.5 Haritalama dizisi

Dizi İndeksi	Eleman
0	k
1	a
2	r
3	t
4	l
5	_
6	d
7	s
8	%

2. Aşama (Kod Karşılıkları Hesaplama)

Bu aşamada metnin başına konumlanılır ve sırasıyla karakterler tampona okunarak sözlükte aranır. Sözlükte bulundukça okuma işlemi devam eder. Çizelge 5.6'da ikinci aşamanın adımları gösterilmiştir.

Çizelge 5.6 2. Geçiş kodlama aşaması

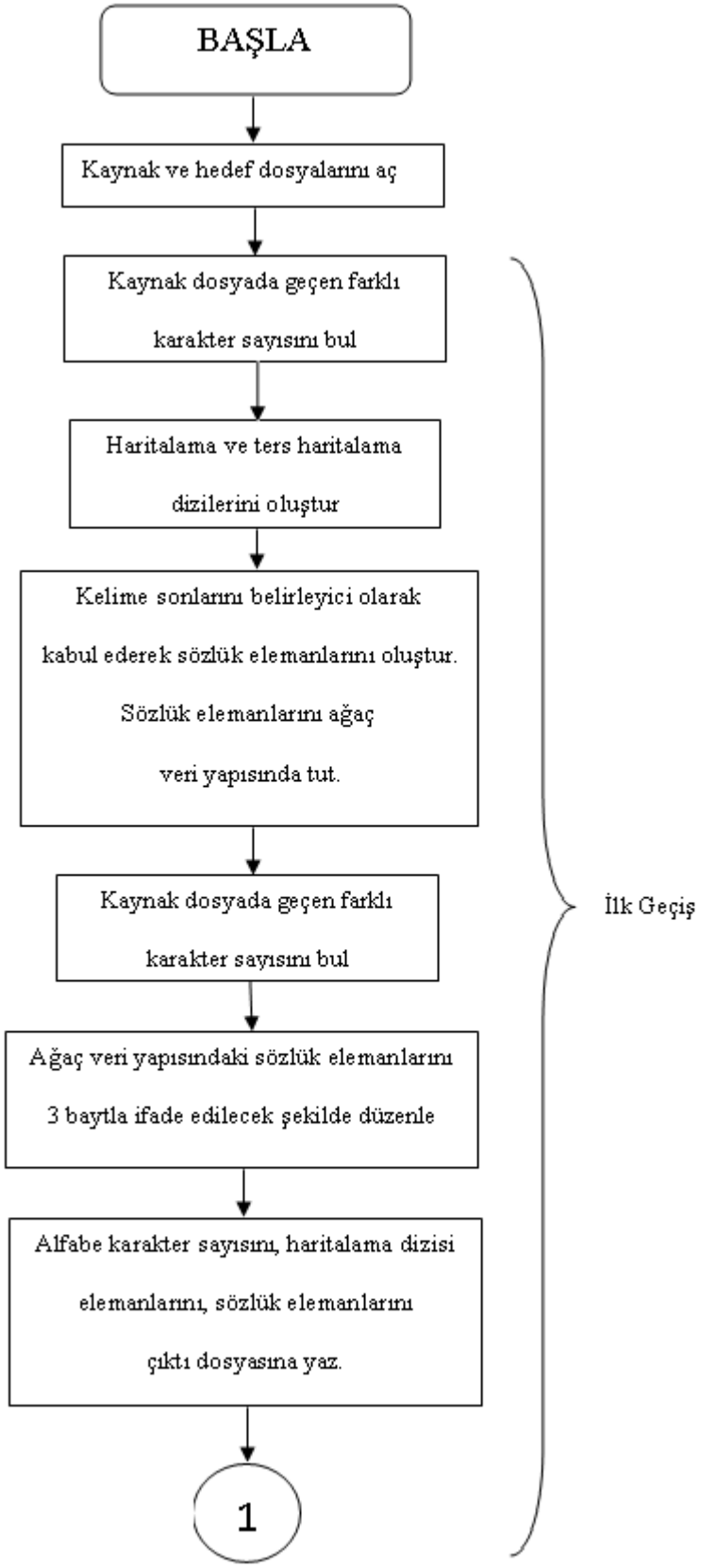
Okunan Karakter	Tampon	İndeks	Çıktı
k	k	--	--
a	ka	--	--
r	kar	65534	--
t	kart	65530	255,250
a	a	--	--
l	l	--	--
_	al_	65529	255,249

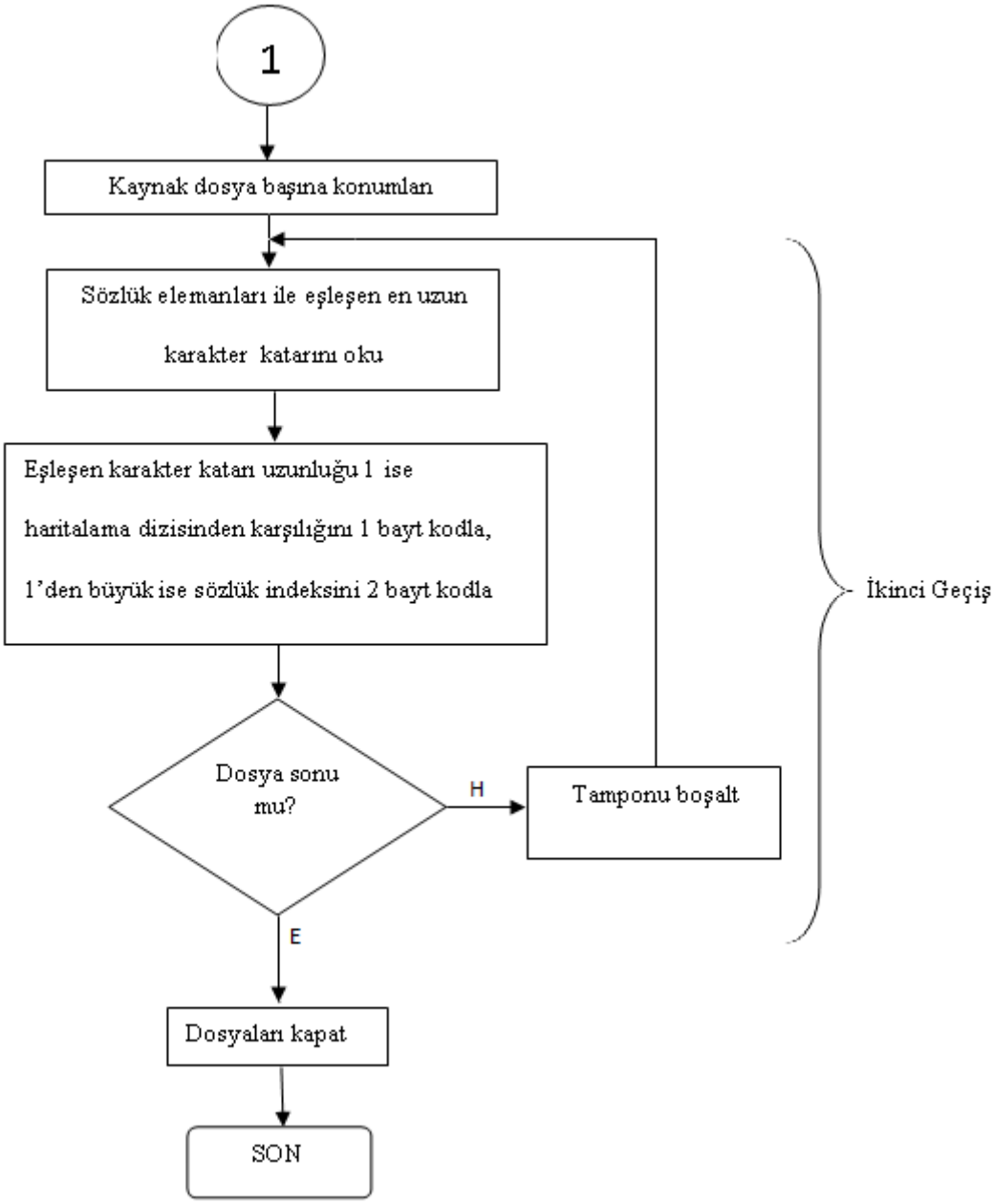
k	k	--	--
a	ka	--	--
l	kal	65535	--

k	kalk	65528	255,248
a	a	--	--
r	ar	--	--
_	ar_	--	1
d	r_d	--	2
a	_da	--	5
l	dal	65533	255,253
_	_	--	5
s	s	--	--
a	sa	--	--
r	sar	65532	255,252
k	k	--	--
a	ka	--	--
r	kar	65534	--
_	kar_	65531	255,251
s	s	--	--
a	sa	--	--
r	sar	65532	255,252
k	k	--	--
a	ka	--	--
r	kar	65534	--
_	kar_	65531	255,251
d	d	--	--
a	da	--	--
l	dal	65533	255,253
_	_	--	--
s	s	--	--
a	sa	--	--
r	sar	65532	255,252
k	k	--	--
a	ka	--	--
r	kar	65534	--
_	kar_	65531	255,251
k	k	--	--
a	ka	--	--
r	kar	65534	--
t	kart	65530	255,250
a	a	--	--

l	al	--	--
_	al_	65529	255,249
k	k	--	--
a	ka	--	--
l	kal	65535	--
k	kalk	65528	255,248
a	a	--	--
r	ar	--	--
%	ar%	65527	255,247

Yukarıdaki örnek “kartal_kalkar_dal_sarkar_dal_sarkar_kartal_kalkar%” metni, “255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247” halini almıştır.


BAŞLA
Kaynak ve hedef dosyalarını aç
Kaynak dosyada geçen farklı karakter sayısını bul
Haritalama ve ters haritalama dizilerini oluştur
Kelime sonlarını belirleyici olarak kabul ederek sözlük elemanlarını oluştur. Sözlük elemanlarını ağaç veri yapısında tut.
Kaynak dosyada geçen farklı karakter sayısını bul
Ağaç veri yapısındaki sözlük elemanlarını 3 baytla ifade edilecek şekilde düzenle
Alfabe karakter sayısını, haritalama dizisi elemanlarını, sözlük elemanlarını çıktı dosyasına yaz.
1
İlk Geçiş



Şekil 5.1 Geliştirilen sıkıştırma algoritması için akış şeması

5.2. Sıkıştırılmış Metni Açma Algoritması

Açma işlemini tarif etmeden önce sözlüklerin sıkıştırılmış dosya başına nasıl ekleneceğinden bahsetmek gerekmektedir. Farklı karakter sayısı bilindikten sonra kaç tane sözlük elemanı olacağının nasıl hesaplanacağından 5.1 numaralı bölümde bahsedilmiştir. Normal metinlerde yaklaşık olarak 35000 – 42000 sözlük elemanı

vardır. Bu sözlük elemanlarının içlerindeki katarlar da en az 3 karakterdir. Eğer ortalama uzunluk 5 karakter olarak düşünülürse, sözlük maliyeti 5x40000=200000 bayt olacaktır. Bu da oldukça yüksek bir rakamdır. Bunun yanında sözlük elemanlarının hepsini 3'er bayt ile kodlamak mümkündür.

Eğer bir sözlük elemanı hiçbir katara benzeşmeden doğrudan sözlüğe ekleniyorsa 3 karaktere sahiptir. Örnek 5.6'da 3 karaktere sahip sözlük öğesi gösterilmiştir. Dolayısıyla sözlük yapısı içerisindeki 3 karakterin de kodu farklı harf sayısından küçüktür.

Örnek 5.6: Örnek 5.5'ten;

65535 – kal – 0,1,4

65534 – kar – 0,1,2

65533 – dal – 6,1,4 olduğu görülür.

3 harften daha uzun katara sahip olan sözlük elemanları ise şu şekilde kodlanmaktadır:

65531 – kar_ - 0,1,2,5

65530 – kart – 0,1,2,3

65528 – kalk – 0,1,4,0

Daha uzun katarlar için bu tanım daha da uzun olacaktır. Bunun yerine her katarın kendisinden önce sözlüğe girmiş olan katarlardan birine bir harf eklenerek oluşması mantığından yola çıkarak sözlükler daha kısa kodlanabilmektedir. Sözlük elemanında yer alan katar kodlanırken sahip olduğu 3 baytın ilk ikisi önek benzeşmesi olduğu sözlük elemanının indeksini belirtmektedir. Sıkıştırma aşamasında olduğu gibi o indeks iki baytla bu bölümde saklanmaktadır. Kalan üçüncü bayt ise eklenen harfin haritalanmış dizideki karşılığıdır. Örnek 5.6 tekrar düzenlenecek olursa;

65531 – kar_ - 255,254(65534 - kar),5

65530 – kart – 255,254(65534 - kar),3

65528 – kalk – 255,255 (65535 - kal),0

halini almaktadır. Son hali elde edilmiş sözlük de sıkıştırılmış dosya başına yazılmaktadır. Buradaki sözlük masrafi da:

$$M_2 = 3xSözlük\ Eleman\ Sayısı = 3x(256 - n)x256 \qquad 5.7$$

Sıkıştırılmış dosya başına yazılan sözlüklerin toplam masrafı da:

$$M = M_1 + M_2 = (n + 1) + 3xSözlük\ Eleman\ Sayısı \qquad 5.8$$

olur.

Açma aşamasında kod çözücü öncelikle sıkıştırılmış dosyanın ilk baytını okur. O bayt farklı harf sayısını (n) verecektir. Dolayısıyla kod çözücü bir dizi açar ve n tane ardışık baytı bu dizinin ardışık elemanlarına yazar. Haritalama dizisi elde edilmiştir. Daha sonra yine n sayısı kullanılarak sözlük ağaçlarının sayısının hesaplanması gerçekleştirilir. Hesaplanan bu değer kadar bellekte yer açılır. Sıkıştırılmış verinin sıradaki 3 baytı alınır ve o açılan yerdeki 65535 numaralı indekse yazılır. Daha sonra sıradaki 3 bayt alınır. Burada eğer alınan 3 baytın birincisi n'den büyük ise bir referans söz konusudur. İki bayt ele alınarak sözlük indeksi hesaplanır, ayrılmış bellek bölgesine indeksin gösterdiği katar ve ilave olarak da üçüncü baytın ifade ettiği karakter yazılır. Bu işlem sözlük sayısına ulaşılıncaya kadar devam eder. İşlem bittikten sonra tüm sözlük doğrudan erişimli bir diziye çıkartılmış olur.

Daha sonra sıkıştırılmış metin işlenmeye başlanır. Bir bayt okunur. Eğer bayt değeri n'den küçük ise doğrudan haritalama dizisindeki o harfin karşılığı dosyaya yazılır. Eğer okunan bayt değeri n'den büyük ise, bu bir bayt daha okunması gerektiğini belirtir. Okunan iki baytın açılan sözlükteki karşılığı bulunur ve dosyaya yazılır.

Açma işlemi doğrudan erişimli diziler kullandığı için sözlük tekrar oluşturulduktan sonra zaman açısından çok da maliyetli değildir. Örnek 5.7'de metin

çok kısa olduğu için sözlük ağaçlarının tamamı dolmamıştır. Böyle durumlarda 1-2 bayt da kullanılan sözlük eleman sayısını belirlemek için başa yazılır.

Örnek 5.7:

Sıkıştırılmış Metin: "9, k, a, r, t, l, _, d, s, %, 9, 0, 1, 4, 0, 1, 2, 6, 1, 4, 7, 1, 2, 255, 254, 5, 255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247"

Adım 1: İlk bayt okunur. Okunan bayt değeri 9 olduğu için sırasıyla okunan 9 karakter bir diziye aktarılır. Böylece haritalama tablosu elde edilir. Çizelge 5.7 elde edilen haritalama tablosunu göstermektedir.

Çizelge 5.7 Haritalama tablosu

İndeks	Harf
0	k
1	a
2	r
3	t
4	l
5	_
6	d
7	s
8	%

Sıkıştırılmış Metin: "9, 0, 1, 4, 0, 1, 2, 6, 1, 4, 7, 1, 2, 255, 254, 5, 255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247"

Adım 2: Elde edilen ilk bayt değerinden ağaç sayısı hesaplanır ve bellekte o kadar yer tahsisi yapılır. Daha sonra 3 bayt okunarak, bayt değerlerinin haritalama dizisindeki karşılıkları bulunarak ilk sözlük elemanına yerleştirilir.

Okunan Bayt Dizisi= 0, 1, 4

İndeks	Katar
65535	0,1,4 - kal

Sıkıştırılmış Metin: “ 0, 1, 2, 6, 1, 4, 7, 1, 2, 255, 254, 5, 255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

Adım 3: Sonraki 3 bayt okunur. Sözlüğün sıradaki indeksine yerleştirilir.

İndeks	Katar
65535	0,1,4 - kal
65534	0,1,2 - kar

Sıkıştırılmış Metin: “6, 1, 4, 7, 1, 2, 255, 254, 5, 255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

Adım 4: Sonraki 3 bayt okunur. Sözlüğün sıradaki indeksine yerleştirilir.

İndeks	Katar
65535	0,1,4 - kal
65534	0,1,2 - kar
65533	6,1,4 - dal

Sıkıştırılmış Metin: “7, 1, 2, 255, 254, 5, 255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

Adım 5: Sonraki 3 bayt okunur. Sözlüğün sıradaki indeksine yerleştirilir.

İndeks	Katar
65535	0,1,4 - kal
65534	0,1,2 - kar
65533	6,1,4 - dal
65532	7,1,2 - sar

Sıkıştırılmış Metin: “255, 254, 5, 255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

Adım 6: Sonraki 3 bayt okunur. Sözlüğün sıradaki indeksine yerleştirilir. Ancak burada okunan baytlardan ilki n değerinden büyüktür. Dolayısıyla ilk bayt 256 ile çarpılır ve ikinci bayt ile toplanır. Elde edilen değerdeki sözlük elemanına üçüncü bayt eklenir. Değer=255*256+254=65534

İndeks	Katar
65535	0,1,4 - kal
65534	0,1,2 - kar
65533	6,1,4 - dal
65532	7,1,2 - sar
65531	65534,5 – kar_

Sıkıştırılmış Metin: “255, 254, 3, 1, 4, 5, 255, 255, 0, 1, 2, 8, 255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

Bu şekilde 4 adım daha giderek tüm sözlük elde edilir. Daha sonra ise okunan bayt değerleri sözlükte aranır.

Sıkıştırılmış Metin: “255, 250, 255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

İlk okunan değer (255) n değerinden büyük olduğu için bir bayt daha okunur. İndeks ilk baytın 256 ilke çarpılıp, ikinci baytın bu sonuca eklenmesiyle (65530) elde edilir. O indeksteki katar çıktı dosyasına yazılır.

Çıktı=”kart”

Sıkıştırılmış Metin: “255, 249, 255, 248, 1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

İki kere bu işlem yapıldığında aşağıdaki çıktı dosyası elde edilir.

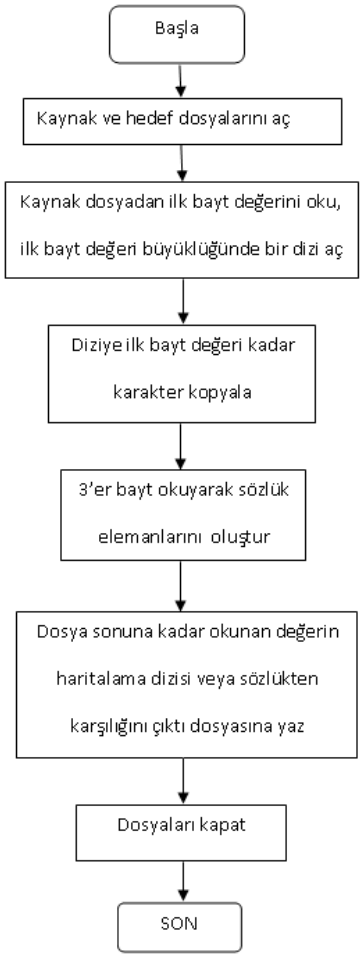
Çıktı=”kart(65530)al_(65529)kalk(65528)”

Sıkıştırılmış Metin: “1, 2, 3, 255, 253, 5, 255, 252, 255, 251, 255, 253, 255, 252, 255, 251, 255, 250, 255, 249, 255, 248, 255, 247”

Şimdi okunan değer 1’dir. n’den küçük olduğu için karakter doğrudan haritalama dizisinden elde edilir.

Çıktı=”kart(65530)al_(65529)kalk(65528)a(1)”

Bu işlemler sıkıştırılmış metin tüketilene kadar devam eder.



Şekil 5.2 TWBCA açma algoritması

5.3. Geliştirilen Algoritma İle Sıkıştırılmış Metinler Üzerinde Sıkıştırılmış Dizgi Eşleme

Geliştirilen algoritma sıkıştırılmış dizgi eşleme yapmaya son derece müsaittir. Yarı statik olduğu için belirli sözlüklere sahiptir ve her kelime bu sözlükler ile sıkıştırılmaktadır. Dinamik algoritmaların aksine aynı sözlükler kullanıldığı için metnin başında bir kelime nasıl sıkıştırılıyorsa, devamında da aynı şekilde sıkıştırılmaktadır.

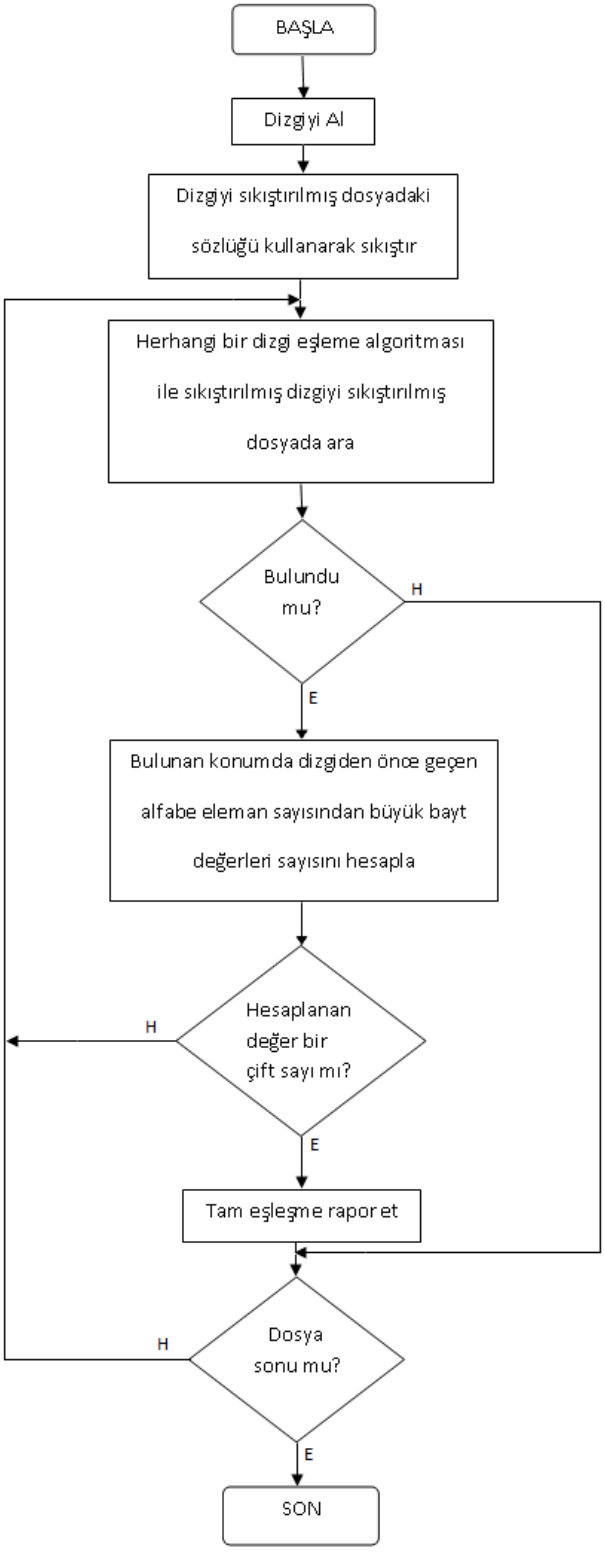
Yine sözlükler belirli ve de her sözlük elemanı sabit olduğu için bir kelimenin sıkışmış hali başka bir kelimenin sıkışmış hali ile aynı olamamaktadır. Bu iki özellik de sıkıştırılmış dizgi eşlemeyi mümkün kılmaktadır.

Yukarıdaki örnekte "kartal_" kelimesi "255,250,255,249" bayt sırası halinde kodlanmıştır. Dolayısıyla eşlenmek istenen dizgi "kartal_" ise aynı şekilde kodlanacağından metin içerisinde herhangi bir dizgi eşleme algoritması kullanılarak bulunabilir.

Ancak sıkıştırılmış dizgi eşleme yaparken dikkat edilmesi gereken bir husus vardır. Örneğin "Trakya_" kelimesinin kodlanmış hali "101 26 167 42" olsun. Bu durumda "101 26" ikilisi ile biten bir kelime ile "167 42" ikilisi ile başlayan kelimeler arka arkaya gelirse bir yanlış eşleşme söz konusu olacaktır. Aslında teoride mümkün olan bu durum pratikte mümkün olmayacaktır. Çünkü "101 26" ikilisi "Trakya_" kelimesinin önekidir ve içerisinde ayraç içermemektedir. Çünkü kendisinden sonra gelen bir ikili daha vardır ve ayraç onun içerisinde gizlidir. Ayraç içermediği için böyle bir kelime sonu olamayacağı yani bir kelimenin böyle bitemeyeceği açıkça gözükmektedir. Dolayısıyla teoride var olan bu durum pratikte var olamayacaktır.

Bir kelimenin öneki ayraç içermediği için başka bir kelimenin soneki olamayacağından yanlış eşleşme ancak bayt çiftlerinin bölünmesi ile gerçekleşebilir. Örneğin kodu "101 167 20 167 95" olan bir kelime arandığı zaman "101" ile biten kelime arkasından "167 20 167 95" kod karşılığı olan kelime gelirse bir yanlış eşleşme ortaya çıkar. 101 değerli baytın önündeki bayt bilinmediği için o ikili bir ayraç içerebilir. 167 95 de bir ayraç içerdiği için bu teoride de pratikte de mümkündür.

Burada 101 değerli baytın önündeki bayt değerinin n sayısından büyük olması gerekmektedir. Bu algoritmanın bir özelliğidir. Bir ikilinin ilk baytı asla n sayısından küçük olamamaktadır. Buradan yola çıkarak, 101'i içeren ikilinin önünde de bir ikili varsa ikinci baytı ya n'den büyük veya eşittir ya da n'den küçüktür. Her iki durumda da 101'in önünde tek sayıda n'den büyük değer mevcuttur. Oysaki bir kelime bir ikili ile başlıyor ise önünde bayt çiftini bölmediği için muhakkak çift sayıda n'den büyük değer vardır. Dolayısıyla konulacak bu kontrol ile kataın gerçekten yer alıp almadığı ortaya çıkacaktır.



Şekil 5.3 TWBCA için sıkıştırılmış dizgi eşleme akış şeması

6. GELİŞTİRİLEN ALGORİTMANIN DİĞER ALGORİTMALAR İLE KIYASLANMASI

Bu bölümde geliştirilmiş olan algoritmanın diğer sıkıştırılmış dizgi eşlemeye izin veren sıkıştırma algoritmaları ile kıyaslamalarına yer verilmiştir.

Kıyaslama işleminde kullanılan diğer algoritmalar *ETDC*, *Compress* ve *Gzip*'tir. ETDC sıkıştırma algoritması kelime tabanlı sıkıştırılmış dizgi eşleme yapan bir algoritmadır. ETDC sıkıştırma algoritmasının sıkıştırma aşamasındaki adımları ve bu adımlarda yapılan işlemler detayları ile bölüm 2'de verilmiştir. ETDC sıkıştırma algoritması kelime tabanlı sıkıştırma yaptığı için sıkıştırılmış dizgi eşleme de geliştirilen algoritma gibi kelime tabanlıdır.

Kıyaslamada kullanılan diğer sıkıştırma algoritması *Compress* UNIX işletim sistemlerinin standart sıkıştırma programıdır. LZW sıkıştırma algoritmasını temel olarak almıştır. *Compress* ile sıkıştırılmış dosyalarda sıkıştırılmış dizgi eşleme yapabilmek için detayları bölüm 4'te verilmiş olan *Lzgrep* isimli yazılım kullanılmaktadır.

Bu sıkıştırma algoritmalarının yanı sıra *Gzip* sıkıştırma algoritması da kıyaslamalar için kullanılmıştır. *Gzip* de standart bir UNIX işletim sistemi veri sıkıştırma yazılımıdır. Yine compress veri sıkıştırma yazılımı ile sıkıştırılmış verilerde *Lzgrep* ile dizgi eşleme yapılabildiği gibi aynı *Lzgrep* isimli yazılım gzip yazılımı ile sıkıştırılmış verilerde de sıkıştırılmış dizgi eşleme yapılmaktadır.

Bu bölümde kullanılan metin dosyalarından olan *odtu.txt* metin dosyası Say, Zeyrek, Oflazer ve Özge'nin 2002 yılında "Eleventh International Conference of Turkish Linguistics" isimli konferansta sundukları "Development of a Corpus and a Treebank for Present-day Written Turkish" isimli çalışmalarından alınmıştır. *odtu.txt* metin dosyası "The Metu Corpus" metin dosyasının XML biçimli bazı karakter ve kelimelerin atılmış olduğu basitleştirilmiş bir şeklidir [Say vd., 2002].

Testlerde kullanılan *bible.txt, world192.txt* isimli metin dosyaları "Catenbery Corpus"tan, *dickens.txt* isimli metin dosyası "Silesia Corpus"tan, *britannica.txt* isimli metin dosyası da "Project Gutenberg"den alınmıştır.

Eng.txt isimli metin dosyası ise çizelge 6.1'de gösterilmiş olan 5 farklı metin dosyasının birleştirilmesinden elde edilmiştir.

Çizelge 6.1 Eng.txt dosyasını oluşturan dosyaların adı, büyüklükleri ve alındıkları yerler

Dosya Adı	**Boyut (bayt)**	**Tanım**	**Alındığı Yer**
1musk12.txt	1,349,141	The Three Musketeers by Alexandre Dumas père	Gutenberg (Etext-No:1257)
alice29.txt	152,089	Alice's Adventures in Wonderland by Lewis Carroll	Canterbury Corpus
asyoulik.txt	125,179	As You Like It by William Shakespeare	Canterbury Corpus
bible.txt	4,047,392	The King James version of the bible	Canterbury Corpus
dickens.txt	10,192,446	Collected works of Charles Dickens	Silesia Corpus

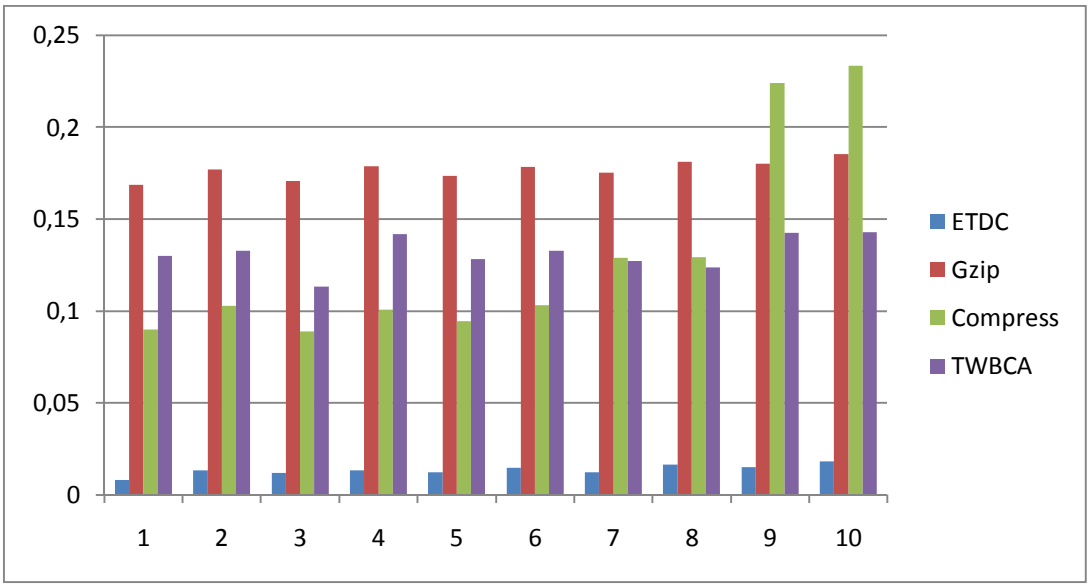
Çizelge 6.2 belirtilen algoritmaların sıkıştırılmış dizgi eşleme zamanlarını göstermektedir.

Çizelge 6.2 Harf sayılarına göre gruplandırılmış dizgilerin ortalama sıkıştırılmış eşleme süreleri (sn)

		ETDC	Gzip	Compress	TWBCA
15 Harfli 75 Dizgi	Eng.txt	0.0081	0.1688	0.0899	0.13
	odtu.txt	0.0136	0.177	0.1028	0.1329
12 Harfli 100 Dizgi	Eng.txt	0.0119	0.1709	0.0889	0.1135
	odtu.txt	0.0136	0.1786	0.1009	0.1419
9 Harfli 100 Dizgi	Eng.txt	0.0124	0.1736	0.0944	0.1283
	odtu.txt	0.015	0.1785	0.1033	0.1328
6 Harfli 100 Dizgi	Eng.txt	0.0125	0.1754	0.1291	0.1274
	odtu.txt	0.0167	0.1813	0.1292	0.1239
3 Harfli 100 Dizgi	Eng.txt	0.0151	0.1803	0.224	0.1424
	odtu.txt	0.0182	0.1855	0.2333	0.1428

Çizelge 6.3'te çizelge 6.2'de iki farklı dosya ve farklı harf sayıları içeren dizgiler için yapılan 10 denemenin grafiksel oranları gösterilmiştir. Bu grafikteki dikey eksen süreyi, yatay eksen de çizelge 6.2 ile ilişkili deneme numarasını göstermektedir.

Çizelge 6.3 Harf Sayılarına Göre Gruplandırılmış Dizgilerin Ortalama Sıkıştırılmış Eşleşme Zamanları



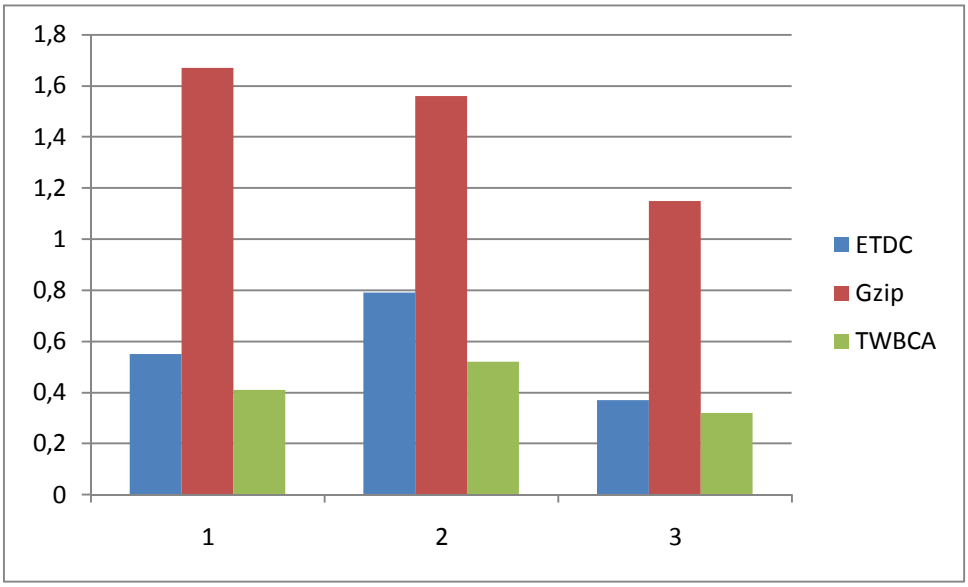
Çizelge 6.2'de değişik sayıda dizgi içeren dizgi gruplarının 4 algoritma üzerinde elde ettikleri sıkıştırılmış eşlenme süreleri verilmiştir. Verilen süreler belli harf sayılarına göre gruplandırılmış dizgiler üzerinden elde edilmiştir.

Çizelge 6.3'e göre *TWBCA* sıkıştırılmış dizgi eşleme algoritması her harf grubu için *Gzip* algoritmasından daha iyi sonuçlar elde etmiştir. Bununla beraber *TWBCA* algoritmasının kısa kelimeler için (3 ve 6 harfli) elde etmiş olduğu süreler *Compress* Algoritmasından az da olsa düşüktür. Ancak daha uzun harfe sahip olan kelimelerin eşlenmesinde *TWBCA* Algoritması *Compress* Algoritması'nın gerisinde kalmıştır. Ancak ETDC algoritması her harf sayısı grubu için diğer tüm algoritmaların altında süreler üretmiştir.

Çizelge 6.4 Sıkıştırma zamanı (sn)

	ETDC	Gzip	TWBCA
Eng.txt	0.55	1.67	0.41
odtu.txt	0.79	1.56	0.52
dickens.txt	0.37	1.15	0.32

Çizelge 6.5 Sıkıştırma zamanları

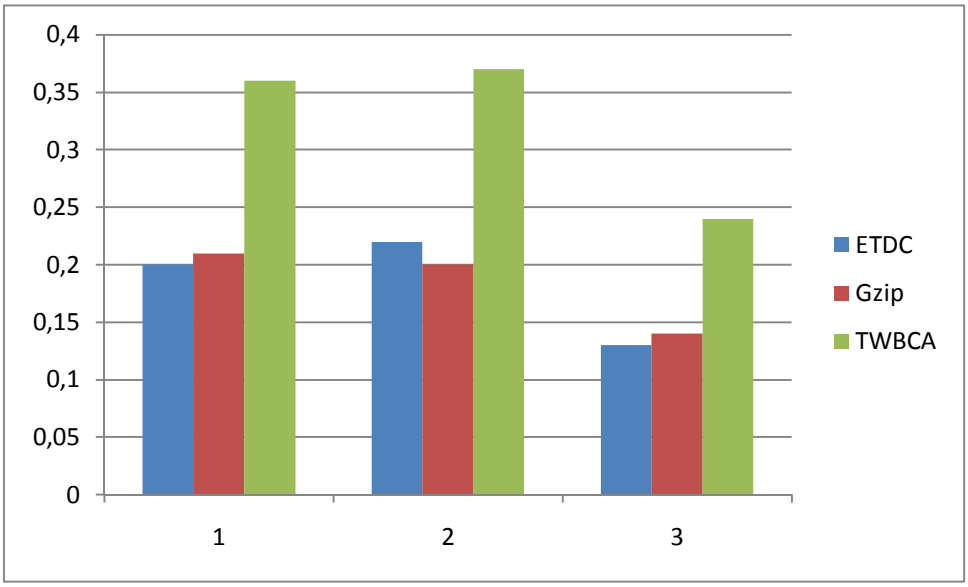


Çizelge 6.4'te *ETDC, Gzip* ve *TWBCA* sıkıştırma algoritmalarının örnek metin dosyalarını sıkıştırma zamanı verilmiştir. Çizelgeden de anlaşılacağı gibi *TWBCA* sıkıştırma algoritmasının veri sıkıştırma zamanı diğer algoritmalardan düşüktür. Bu da sıkıştırma işleminin hızlı olması gerektiği durumlarda oldukça önemlidir. Çizelge 6.5 ise çizelge 6.4'ün grafiksel gösterimini sunmaktadır. Bu çizelgede yer alan dikey eksen açma süresini, yatay eksen de grafiğin çizelge 6.4'teki 3 farklı metin verisi için oluşturulduğunu göstermektedir.

Çizelge 6.6 Sıkıştırılmış dosyaların açma zamanları (sn)

	ETDC	Gzip	TWBCA
Eng.txt	0.20	0.21	0.36
odtu.txt	0.22	0.20	0.37
dickens.txt	0.13	0.14	0.24

Çizelge 6.7 Sıkıştırılmış dosyaları açma zamanları (sn)

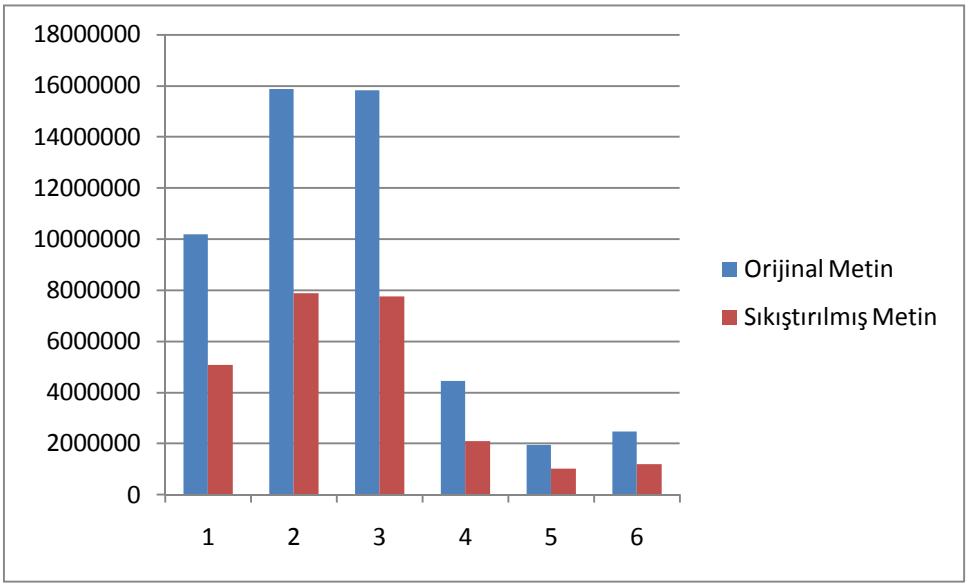


Çizelge 6.6'da verilen sıkıştırılmış dosyaların açma işlemi esnasında harcanan zaman gösterilmiştir. Çizelgeye göre *TWBCA* diğer algoritmalardan gözle görülür bir biçimde yavaş açma işlemi yapmaktadır. Bunun nedeni sözlüğün sıkıştırılmış dosyaya yazılırken kendi içinde elemanlarının birbirine referans olarak gösterilmiş biçimde yazılmış olması ve sözlüğün açma esnasında bu referansları takip ederek bir sözlük elemanı için birden fazla erişim yapmasıdır.

Çizelge 6.8 TWBCA algoritmasının sıkıştırma oranları

Dosya Adı	Orijinal Boyut (Bayt)	Sıkıştırılmış Boyut (Bayt)	Sıkıştırma Oranı (%)
dickens.txt	10192446	5087067	49,910
Eng.txt	15866246	7893201	49,748
odtu.txt	15828594	7761656	49,036
bible.txt	4445255	2105545	47,366
britannica.txt	1945731	1029203	52,895
world192.txt	2473400	1193380	48,249

Çizelge 6.9 TWBCA algoritmasının sıkıştırma oranları



Çizelge 6.8'de *TWBCA* Algoritmasının sıkıştırma algoritmalarının testlerinde kullanılmakta olan bazı önemli metinler üzerindeki sıkıştırma oranları verilmiştir. Algoritma genel itibariyle %50 civarında sıkıştırma sağlamaktadır. Algoritma bu oranı yakalarken kullanılan diller karşısında herhangi bir zafiyet göstermemektedir. Örneğin *odtu.txt* Türkçe dili kullanılmış bir metin dosyası iken, diğerlerinde kullanılan dil İngilizcedir. Bunun nedeni de *TWBCA* sıkıştırma algoritmasının iki geçişli bir algoritma olması ve ilk geçişte işlenen metne özel sözlük oluşturmasıdır.

Çizelge 6.9'da çizelge 6.8'in grafiksel gösterimi verilmiştir. Dikey eksen dosyaların bayt sayısını, yatay eksen de dosyanın çizelge 6.8'deki satır numarasını yani sırasını vermektedir.

Testlerin gerçekleştirildiği bilgisayar ortamında Intel (R) Core(TM)2Duo T8300 2.4 Ghz işlemci, 3 Gb RAM ve Ubuntu 10.0.4 işletim sistemi bulunmaktadır.

Ayrıca bu tezin yazımında, külliyatların ve diğer verilerin saklanması ve iletiminin sağlanmasında, yeni bir algoritmanın tasarlanması, kodlanması ve diğer algoritmalarla testlerinin yapılmasında kullanılan tüm donanım birimleri Trakya Üniversitesi Bilimsel Araştırma Projesi Fonu'ndan karşılanmıştır.

7. SONUÇLAR

Sıkıştırılmış dizgi eşleme işlemini başarılı kılan iki etmen vardır. Bunlar sırasıyla doğru eşleyebilme ve hızlı eşleyebilmedir. 90'lı yıllardan önce metinler üzerinde dizgi eşleme sadece açık metin dosyalarında yapılırken, 90'lı yılların başından itibaren hızlı ve doğru sıkıştırılmış dizgi eşleyen birçok algoritma geliştirilmiş veya mevcut sıkıştırma algoritmalarına sıkıştırılmış dizgi eşleme işlemi yapabilen yazılımlar eklenmiştir. Bu geliştirilen yazılımların ve algoritmaların daha çok sözlük tabanlı algoritmalar ekseninde toplandığı gözlemlenmiştir.

Buna paralel olarak bu çalışmada üretilen algoritma da sözlük tabanlı ve yarı statik sözlük yaklaşımı kullanan bir algoritmadır. Algoritmanın yarı statik olması da üzerinde etkin bir şekilde dizgi eşleme yapabilmeyi sağlamaktadır.

Yapılan testler sonucunda geliştirilen algoritmanın ortalama bir sıkıştırma oranına sahip olduğunu göstermektedir. Yine testler sonucunda geliştirilen algoritma üzerinde sıkıştırılmış dizgi eşleme zamanlarının yine ortalama bir seviyede olduğu görülmüştür.

TWBCA algoritması, sıkıştırma algoritmaları etkinlik ölçütlerinden olan hızlı sıkıştırma kriterini başarılı bir şekilde gerçekleştirmektedir. Ancak buna ters olarak açma işleminde bir zafiyet söz konusu olmuştur. Bunun nedeni olarak da sıkıştırılmış sözlüğün açılması için geçen sürenin de açma işlemine eklenmesi gösterilmiştir.

TWBCA algoritması bu zafiyet açısından büyük dosyalarda sözlük maliyetinin sıkıştırılmış metin dosyasına oranının daha az olacağı gerçeğinden yola çıkarak, sözlüğü sıkıştırmadan sıkıştırılmış dosya başına yazmak suretiyle iyileştirilebilmektedir.

Algoritmanın avantajlarından bir tanesi sabit sözlük kullanmaması yani yarı statik olması ve kullanılan dosyaya özgü sözlük oluşturması nedeniyle dil bağımsız olmasıdır.

Geliştirilen sıkıştırma algoritması TWBCA'nın diğer bir avantajı da kullanılan karakterlerin haritalanması ve dolayısıyla kullanılmayan karakterlerin kodlarının boş kalmasıdır. Bu işlem de geliştirilen sıkıştırma algoritmasının kullanılmayan karakter olduğu sürece sıkıştırma yapması dolayısıyla metin verisi haricinde diğer verilerde de

kullanılabilmesine olanak sağlamaktadır. Ancak algoritmanın ortaya çıkaracağı sıkıştırma oranının dosyada kullanılmayan karakter sayısı ile orantılı olduğu dolayısıyla da görüntü ve ses dosyası gibi dosyalarda oranın düşük olacağı dikkatlerden kaçmamaktadır.

KAYNAKLAR

Adjeroh D., Mukherjee A., Bell T., Powell M., Zhang N., "Pattern Matching in BWT – Compressed Text", Data Compression Conference, Proceedings. DCC 2002

Amir A., Benson G., "Efficient Two Dimensional Compressed Matching", Data Compression Conference, 1992

Amir A., Benson G., Farach M., "Let Sleeping Files Lie: Pattern Matching in Z – Compressed Files", Symposium on Discrete Algorithms, Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms, P: 705 – 714, 1994

Barcaccia P., Cresti A., De Agostino S., "Pattern Matching in Text Compressed with the ID Heuristic", DCC, Proceedings of the Conference on Data Compression,P:113, 1998

Brisaboa N. R., Iglesias E. L., Navarro G., Parama J. R., "An Efficient Compression Code for Text Databases", Lecture Notes In Computer Science, Proceedings of the 25th European conference on IR research, Pisa, Italy, SECTION: Architectural issues and efficiency issues in IR, P: 468-481, 2003

Carus A., Buluş H. N., Mesut A., "WordMatch: Word Based String Matching over Compressed Texts", UNITECH 07, Gabrovo, Bulgaristan

Chen L., Lu S., Ram J., "Compressed Pattern Matching in DNA Sequences", CSB, Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference, P: 62 – 68, 2004

Crochemore M., Lecroq T., "Pattern Matching and Text Compression Algorithms", ACM Computing Surveys (CSUR) Volume 28 , Issue 1, P: 39 – 41, 1996

Crochemore M., Rytter W., "Text Algorithms", Oxford University Press, USA, 1997

Culpepper J. S., Moffat A., "Enhanced Byte Codes with Restricted Prefix Properties", String Processing and Information Retrieval, Springer Berlin / Heidelberg, 2005

Culpepper J. S., Moffat A., “Phrase – based Pattern Matching in Compressed Text”, String Processing and Information Retrieval, Springer Berlin / Heidelberg, 2006

Ersin A., “Dizgi Eşleme Algoritmalarının İncelenmesi ve Yeni Bir Dizgi Eşleme Algoritması” Yüksek Lisans Tezi, Trakya Üniversitesi, 2008

Farach M., Thorup M., “String Matching in Lempel – Ziv Compressed Strings”, Annual ACM Symposium on Theory of Computing, Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, Las Vegas, Nevada, United States, P: 703 – 712, 1995

Gasieniec L., Karpinski M., Plandowski W., Rytter W., “Randomized Efficient Algorithms For Compressed Strings: the Finger – Print Approach”, Combinatorial Pattern Matching, P:39-49, Springer Berlin / Heidelberg, 1996

Gasieniec L., Rytter W., “Almost Optimal Fully LZW – Compressed Pattern Matching”, DCC, Proceedings of the Conference on Data Compression, P: 316, 1999

Hirao M., Shinohara A., Takeda M., Arikawa S., “Fully Compressed Pattern Matching Algorithm for Balanced Straight – Line Programs”, SPIRE, Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00), P: 132, 2000

Ho M. H., Yen H. C., “A Dictionary – based Compressed Pattern Matching Algorithm”, COMPSAC, Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment, P: 873 – 878, 2002

Inenaga S., Shinohara A., Takeda M., “A Fully Compressed Pattern Matching Algorithm for Simple Collage Systems”, The Prague Stringology Conference, 2004

Karkkainen J., Navarro G., Ukkonen E., ”Approximate String Matching Over Ziv – Lempel Compressed Text”, Combinatorial Pattern Matching, Springer Berlin / Heidelberg, 2000

Kida T., Takeda M., Shinohara A., Miyazaki M., Arikawa S., "Multiple Pattern Matching in LZW Compressed Text", DCC, Proceedings of the Conference on Data Compression, P:103, 1998

Kida T., Shibata Y., Takeda M., Shinohara A., Arikawa S., "A Unifying Framework for Compressed Pattern Mtaching", SPIRE, Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware, P: 89, 1999

Kida T., Takeda M., Shinohara A., Arikawa S., "Shift – And Approach to Pattern Matching in LZW Compressed Text", Lecture Notes In Computer Science; Vol. 1645, Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching, P: 1 – 13, 1999

Kida T., Matsumoto T., Takeda M., Shinoharai A., Arikawa S., "Multiple Pattern Matching Algorithms on Collage System", Lecture Notes In Computer Science; Vol. 2089, Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, P: 193 – 206, 2001

Klein S. T., Shapira D., "A New Compression Method for Compressed Matching", DCC, Proceedings of the Conference on Data Compression, P: 400, 2000

Klein S. T., Shapira D., "Pattern Matching in Huffman Encoded Texts", Information Processing and Management: an International Journal, Volume 41 , Issue 4 , P: 829 – 841, 2005

Lee T. H., Huang N. L., "An Efficient Pattern Matching Scheme in LZW Compressed Sequences", Security and Communication Networks Volume 1 Issue 4**,** P: 325 – 335, 2008

Manber U., "A Text Compression Scheme that Allows Fast Searching Directly in the Compressed File", ACM Transactions on Information Systems (TOIS), Volume 15 , Issue 2 (April 1997), P: 124 – 136, 1997

Martinez A. F., “New Compression Codes for Text Databases”, Phd Thesis, University of A Coruna , 2005

Mesut A., “Veri Sıkıştırmada Yeni Yöntemler”, Doktora Tezi, Trakya Üniversitesi, 2006

Mitarai S., Hirao M., Matsumoto T., Shinohara A., Takeda M., Arikawa S., “Compressed Pattern Matching for Sequitur”, DCC, Proceedings of the Data Compression Conference, P: 469, 2001

Moura E. S., Navarro G., Ziviani N., Baeza – Yates R., “Fast Searching on Compressed Text Allowing Errors”, Annual ACM Conference on Research and Development in Information Retrieval, Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval, 1998

Moura E. S., Navarro G., Ziviani N., Baeza – Yates R., “Direct Pattern matching on Compressed Text”, SPIRE, 1998

Moura E. S., Navarro G., Ziviani N., Baeza – Yates R., “Fast and Flexible Word Searching on Compressed Text”, ACM Transactions on Information Systems (TOIS) , Volume 18 , Issue 2 , 113-139, 2000

Navarro G., Raffinot M., “A General Practical Approach to Pattern Matching over Ziv – Lempel Compressed Text”, Lecture Notes In Computer Science; Vol. 1645, Proceedings of the 10th Annual Symposium on Combinatorial Pattern, P:14-36, 1999

Navarro G., Tarhio J., “Boyer – Moore String Matching over Ziv – Lempel Compressed Text”, Lecture Notes In Computer Science; Vol. 1848, Proceedings of the 11th Annual Symposium on Combinatorial Pattern, P:166-180, 2000

Navarro G., Tarhio J., “LZgrep: A Boyer – Moore String Matching Tool for Ziv – Lempel Compressed Text”, Software—Practice & Experience, Volume 35 , Issue 12 , P: 1107 – 1130, 2005

Nelson M., Gailly J. L., “The Data Compression Book – Second Edition”, M&T Books, 1995

Pu, I. M., “Fundamental Data Compression”, Butterworth-Heinemann, 2006

Salomon D., “A Guide to Data Compression Methods”, Springer, 2002

Salomon D., “Data Compression – Third Edition”, Springer, 2004

Say, B., Zeyrek, D., Oflazer, K., Ozge, U., “ Development of a Corpus and a Treebank for Present-day Written Turkish”, Eleventh International Conference of Turkish Linguistics, pp: 183-192, 2002

Shapira D., Daptardar A., “Adapting the Knuth – Morris – Pratt Algorithm for Pattern matching in Huffman Encoded Texts”, Information Processing and Management: an International Journal, Volume 42 , Issue 2, 2006

Shibata Y., Kida T., Fukamachi S., Takeda M., Shinohara A., Shinohara T., Arikawa S., “Byte Pair Encoding: A Text Compression Scheme That Accelerates Pattern Matching”, 1999

Shibata Y., Matsumoto T., Takeda M., Shinohara A., Arikawa S., “A Boyer – Moore Type Algorithm For Compressed Pattern Matching”, Lecture Notes In Computer Science; Vol. 1848, Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching, P: 181 – 194, 2000

Takeda M., Shibata Y., Matsumoto T., Kida T., Shinohara A., Fukamachi S., Shinohara T., Arikawa S., “Speeding Up String Matching by Text Compression: The Dawn of a New Era”, Transactions of Information Processing Society of Japan, VOL.42;NO.3;P:370-384, 2001

Tao T., Mukherjee A., “LZW Based Compressed Pattern Matching”, DCC, Proceedings of the Conference on Data Compression, P: 568, 2004

Tao T., Mukherjee A., “Pattern Matching in LZW Compressed Files”, IEEE Transactions on Computers, Volume 54 , Issue 8 Ps: 929 – 938, 2005

Zhang N., Tao T., Satya R. V., Mukherjee A., “Modified LZW Algorithm for Efficient Compressed Text Retrieval”, ITCC, Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2 - Volume 2, P:224,2004

Ziviani N., de Moura E. S., Navarro G., Baeza – Yates R., “Compression: A Key for Next Generation Text Retrieval Systems”, Computer, vol. 33, no. 11, pp. 37-44, 2000

EKLER

EK-A. KISALTMALAR LİSTESİ

KISALTMALAR	İNGİLİZCE	TÜRKÇE
ABF	Amir – Benson – Farach	
AC	Aho - Corasick	
ASCII	American Standard Code for Information Interchange	Bilgi Değişimi İçin Amerikan Standart Kodlama Sistemi
BC	Byte Coding	Bayt Kodlama
BF	Brute – Force	Deneme/Yanılma
BM	Boyer – Moore Pattern Matching Algorithm	Boyer – Moore Dizgi Eşleme Algoritması
bpb	Bit per bit	Bit başına bit
bpc	Bit per character	Karakter başına bit
BPE	Byte Pair Encoding	Bayt Çifti Kodlama
bpp	Bit per pixel	Piksel başına bit
BWT	Burrows Wheeler Transform	Burrows Wheeler Dönüşümü
CPM	Compressed Pattern Matching	Sıkıştırılmış Dizgi Eşleme
CPU	Central Processing Unit	Merkezi İşlem Birimi
CS	Collage System	Collage Sistemi
DNA	Deoksiribonucleic Asid	Deoksiribonükleik Asit
EOF	End – of – File	Dosya Sonu
ETDC	End Tagged Dense Code	Sonu Etiketlenmiş Yoğun Kodlar
IBWT	Inverse BWT	Ters BWT
ID	Identity	Kimlik
IR	Information Retrieval	Bilgi çıkarımı
KMP	Knuth Morris Pratt	
KR	Karp – Rabin	
LZ	Lempel Ziv	

LZ1	Lempel Ziv 1	
LZ77	Lempel Ziv 77	
LZ78	Lempel Ziv 78	
LZH	Lempel – Ziv – Herd	
LZSS	Lempel–Ziv–Storer – Szymanski	
LZW	Lempel Ziv Welch	
MTF	Move – to – front	Öne Alma
PHC	Plain Huffman Code	Düz Huffman Kodu
RLE	Run-Length Encoding	Geçiş Uzunluğu Kodlama
RPBC	Restricted Prefix Byte Coding	Sınırlı Önek Bayt Kodlama
(s,c)-DC	(s,c) – Dense Codes	(s,c)-Yoğun Kod
SF	Shannon – Fano	
SLP	Straight – Line Program	Düz Doğru Program
SSLP	Simple Straight – Line Program	Basit Düz Doğru Program
THC	Tagged Huffman Code	Etiketlenmiş Huffman Kodu
TRS	Text Retrieval Systems	Metin Çıkarım Sistemleri
TWBCA	Tagged Word Based Compression Algorithm	Etiketlenmiş Kelime Tabanlı Sıkıştırma Algoritması
VLC	Variable Length Coding	Değişken Uzunluklu Kodlama

ÖZGEÇMİŞ

23 Ağustos 1981'de Kırklareli İli Vize İlçesi'ne bağlı Sergen Kasabasında doğdu. İlköğrenimini Vize Cumhuriyet İlkokulunda, orta ve lise öğrenimini Lüleburgaz Anadolu Lisesinde tamamladı. 1999 yılında Trakya Üniversitesi Mühendislik – Mimarlık Fakültesi Bilgisayar Mühendisliği Bölümünde okumaya hak kazandı. 2003 yılında mezun olarak Eylül 2003'te Trakya Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalında yüksek lisansa başladı. Ocak 2004 tarihinde Trakya Üniversitesi Bilgisayar Mühendisliği Bölümünde araştırma görevlisi olarak çalışmaya başladı. Ocak 2006 tarihinde yüksek lisans programından mezun olarak, yine Ocak 2006'da Trakya Üniversitesi Fen Bilimleri Enstitüsü Bilgisayar Mühendisliği Anabilim Dalında doktora programına başladı. Ocak 2009 tarihinde Namık Kemal Üniversitesi Çorlu Mühendislik Fakültesi Bilgisayar Mühendisliği Bölümünde araştırma görevliliğine başladı. Bu görevini görevlendirmeye tabi olarak Trakya Üniversitesi Bilgisayar Mühendisliğinde sürdürmektedir. Kasım 2007 tarihinden beri evlidir.