# *Sözlük Tabanlı Veri Sıkıştırma Projesi*

***Dictionary Based Coding Project***

Cumali TOPRAK
Mühendislik Fakültesi, Bilgisayar Mühendisliği
Kocaeli Üniversitesi
Kocaeli, Türkiye
cumalitoprakk@gmail.com

Mustafa ÇÖMEZ
Mühendislik Fakültesi, Bilgisayar Mühendisliği
Kocaeli Üniversitesi
Kocaeli, Türkiye
m.comez3737@gmail.com

***Özetçe*—Bu projede, bir .txt uzantılı dosyadaki verinin, LZ77 ve Deflate algoritmaları kullanılarak sıkıştırılması ve sıkıştırılan verinin başka .txt uzantılı dosyada sıkıştırılmış haliyle oluşturulması ve bu iki algoritmanın mukayese edilmesi amaçlanmaktadır. LZ77 algoritması eksiksiz olarak yapılmış olup dosyaya sıkıştırılan veriler yazdırılmıştır. Fakat deflate algoritmasında kullanılan blokları Huffman algoritması ile entegre edemediğimiz için sadece Huffman algoritması ile sıkıştırma işlemi yaptık.**

***Anahtar Kelimeler—Sıkıştırma algoritması, LZ77, Deflate.***

***Abstract* — In this project, it is aimed to compress the data in one file with .txt extension by using LZ77 and Deflate algorithms, and to create the compressed data as compressed in another .txt file and compare these two algorithms.LZ77 algorithm has been made completely and the data compressed to the file has been printed. However, since we could not integrate the blocks used in the deflate algorithm with the Huffman algorithm, we only compressed with the Huffman algorithm.**

***Keywords— Compression Algorithm, LZ77, Deflate.***

## I. GİRİŞ

Öncelikle projemiz C programlama dili ile geliştirilmiştir. Bizden de istendiği gibi sıkıştırılmak istenen metin.txt dosyasındaki tüm karakterler okunarak metin sıkıştırma işlemi yapılmaktadır. Sıkıştırılmış olan metin ayrı bir dosyada binary olarak kodlanmıştır.

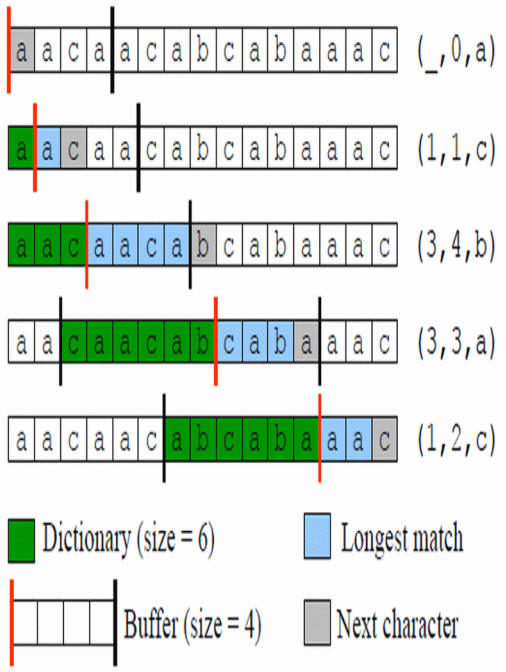
Dosya Sıkıştırma Nedir? Neden yapılır?

Bir veya birden fazla dosyayı kullanılan algoritmayla birlikte daha küçük boyuta indirgeme işlemine sıkıştırma denir. Sıkıştırma sayesinde dosyaların disk üzerinde kapladığı boyutu düşürülür. Olduğundan daha küçük boyuta indirgenen dosyalar daha kolay indirilebilir, paylaşışabilir hale gelirler. Böylece kullanıcılar hem zamandan hem veri miktarından tasarruf edebilirler.

## II. Kullanılan Yöntemler

### *A. LZ77 Sıkıştırma Algoritması*

LZ77 algoritmasının amacı, metin belgelerindeki tekrar eden bölümleri ortadan kaldırmak ve buna yönelik metin belgesini sıkıştırmaktır. LZ77 yaklaşımındaki sözlük, daha önce kodlanmış serinin bir parçasıdır. Algoritmadaki arama tamponun büyüklüğü daha önce kodlanmış serinin ne büyüklükte bir parçasında arama yapılacağını belirler. Arama tamponu büyüdükçe, sıkıştırma oranı artar, Fakat aynı zamanda sıkıştırma zamanı da artar. Biz de kendi oluşturduğumuz LZ77 algoritmasında 63 bitlik bir arama tamponu ve 4 bitlik ileri tampon kullandık. Kodumuzun detayları ilerleyen bölümlerde anlatılcaktır.




Şekil-1: Algoritmanın uygulanışını gösteren şekil.

Kod Aşamaları:

```
char *dosya_oku(FILE *f, int *boyut)
{
    char *icerik;
    fseek(f, 0, SEEK_END);
    *boyut = ftell(f);
    icerik = malloc(*boyut);
    fseek(f, 0, SEEK_SET);
    fread(icerik, 1, *boyut, f);
    return icerik;
}
```

Şekil-2: Dosya_oku() fonksiyonun gösterimi.

Bu fonksiyon yardımı ile öncelikle metin.txt dosyası okunarak bir değişkenden saklanır. Bu metni içeren değişken daha sonra sıkıştırılmak üzere "kodlama()" fonksiyonuna gönderilir.

```
struct isaret
{
    uint8_t offset_len;
    char c;
};
```

Şekil-3: offset ve length kısımları sakladığımız struct yapısı.

Bu yapı sayesinde LZ77 ile oluşturduğumuz algoritmanın karakterlerinin bilgilerini tuttuk. Bu bilgileri daha sonra #define ile daha önce tanımladığımız makrolar sayesinde decompress edip sıkıstırılmıs metni elde edebileceğiz.

```
struct isaret *kodlama(char *yazi, int sinir, int *isaret_num)
{
    int kapasite = 1 << 3;

    int _isaret_num = 0;

    struct isaret i;

    char *ileritampon, *aramatampon;

    struct isaret *kodlu = malloc(kapasite * sizeof(struct isaret));

    for (ileritampon = yazi; ileritampon < yazi + sinir; ileritampon++)
    {
        aramatampon = ileritampon - OFFSETMASK;

        if (aramatampon < yazi)
            aramatampon = yazi;

        int maks_len = 0;
```

Şekil-4: Kodlama fonksiyonun gösterimi.

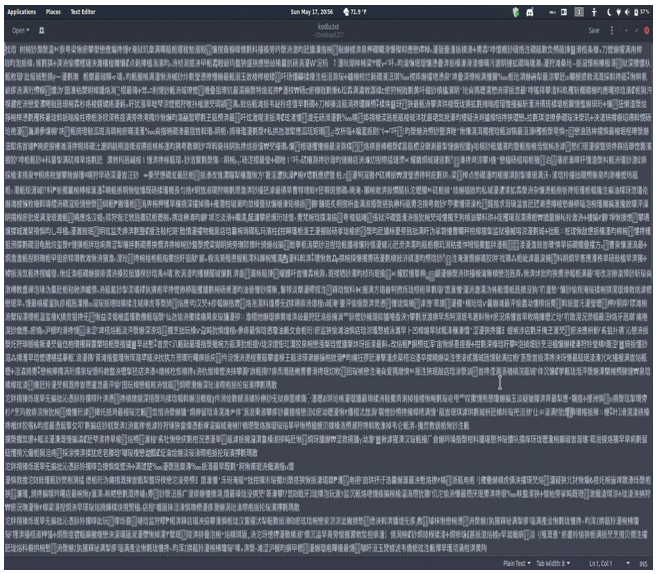
Bu fonksiyon ile dosyadan okuduğumuz karakteri sırası ile işleyerek daha önce görülme durumuna göre karakter bilgilerini ve bu karakterin daha önce nerde bulunduğunu bir üst bölümde anlattığımız struct yapısında sakladık. Bu fonksiyon çalışmasını bitirdiğinde kodlu.txt adlı dosya sıkıştırılmış dosya oluşturulacaktır.

```
char *cozum(struct isaret *isaretler, int isaret_num, int *cozulu_pcb)
{
    int kapasite = 1 << 3;
    *cozulu_pcb = 0;
    char *cozulu = malloc(sizeof(kapasite));
    int x;
    for (x = 0; x < isaret_num; x++)
    {
        int offset = GETOFFSET(isaretler[x].offset_len);
        int len = GETLENGTH(isaretler[x].offset_len);
        char c = isaretler[x].c;
        if (*cozulu_pcb + len + 1 > kapasite)
        {
            kapasite = kapasite << 1;
            cozulu = realloc(cozulu, kapasite);
        }
        if (len > 0)
        {
            memcopy_lz77(&cozulu[*cozulu_pcb], &cozulu[*cozulu_pcb - o
        }
```

Şekil-5: Cozum fonksiyonun gösterimi.

Bu fonksiyonla daha once kodlama adlı fonksiyonla isaret yapısını kullanarak sıkıstırdığımız dosyayı tekrar normal formuna döndürüyoruz. Burda bu kodlanmış veriyi açmak için işaret struct'ına atadığımız değerleri define ile tanımladığımız makrolar yardımı ile açıyoruz. Bu fonksiyon çalışmasını bitirdiği zaman elimizde Orijinal metin olan metin.txt dosyası ile eşdeğer bir dosya oluşur.

Bu fonksiyonlar dışında ara işlemleri yapan yardımcı fonksiyonlar daha bulunmaktadır. Fakat onun detaylarına burda girmeyeceğiz. LZ77 algoritması çalışmasını tamamladığı zaman dosyada sıkıştırılmış verinin gösterimi aşağıdaki gibi olur.



Şekil-5: Sıkıştırılmış metin.txt dosyasının kodlu.txt dosyasında sıkıştırılmış binary gösterimi.

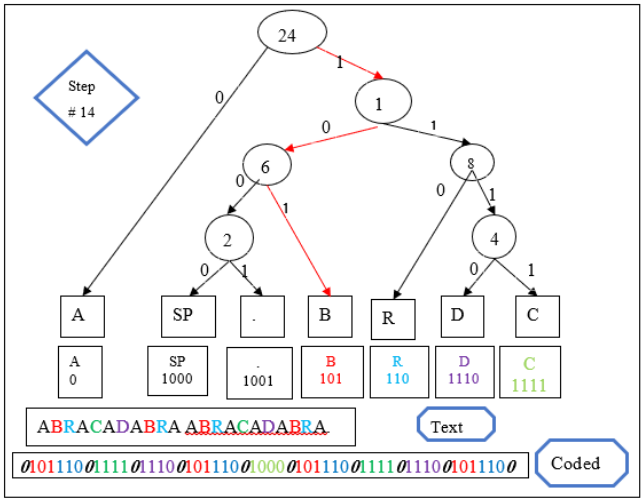
Bu dosyayı cozum() fonksiyonu yardımı ile algoritma çalışmasını tamamladığı zaman tekrar eski haline döndürülmüş olarak yaptık.

### *B. Huffman Algoritması*

Normal şartlarda bizden istenen Deflate algoritmasını daha önce özet kısmında da belirtiğimiz gibi blokları koda entegre edemediğimiz için biz de sadece Huffman algoritması ile sıkıştırma sonucu gösterdik. Bu yüzden Deflate algoritmasını tam olarak gerçekleyemedik. Burda yapmış olduğumuz Huffman kodlaması yardımı ile sıkıştırma sonuçlarının gösterimini anlatacağız.

Huffman Algoritması nedir? Nerelerde Kullanılır?

Bilgisayar bilimlerinde veri sıkıştırmak için kullanılan bir kodlama yöntemidir. Kayıpsız (lossless) olarak veriyir sıkıştırıp tekrar açmak için kullanılır. Huffman kodlamasının en büyük avantajlarından birisi kullanılan karakterlerin frekanslarına göre bir kodlama yapması ve bu sayede sık kullanılan karakterlerin daha az, nadir kullanılan karakterlerin ise daha fazla yer kaplamasını sağlamasıdır. Şayet bütün karakterlerin dağılımı eşitse yani aynı oranda tekrarlanıyorsa, bu durumda Huffman kodlaması aslında blok sıkıştırma algoritması (örneğin ASCII kodlama) ile aynı başarıya sahiptir. Ancak bu teorik durumun gerçekleşmesi imkansız olduğu için her zaman daha başarılı sonuçlar verir.




Şekil-5: Huffman algoritmasının şekil üzerinde gösterimi.

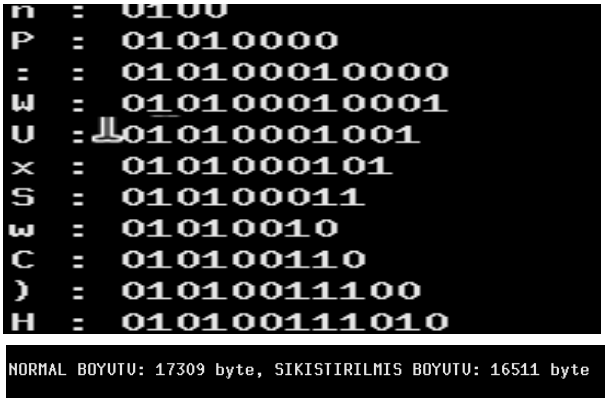
Kod Aşamaları:

Öncelikle metin.txt dosyası yukarda daha önce tanımladığımız yardımcı fonksiyon yardımı ile karakter karakter okunur. Okunan bu metinin daha sonra karakter karakter frequency (sıklık) tespiti yapılır.

```
for(i=0; i<metin_boyutu; i++)
{
    if(charFrequency[(int)kaynak_metin[i]]==0)
    {
        usedChar[(int)kaynak_metin[i]]=kaynak_metin[i];
        flag++;
    }
    charFrequency[(int)kaynak_metin[i]]+=1;
}
```

Şekil-6: Karakterlerin sıklık tespitini yapan kaynak kod gösterimi.

Sonra gerçekleştirilen bu aşamadan sonra bu her bir karakteri ve metin içerisinde geçme sıklığını Huffman algoritması yardımı ile bitlerle ifade ederek, metin içerisinde çok fazla geçen karakterleri daha az bitle daha çok geçen karakterleri daha çok bitle ifade eder. Huffman algoritması çalışmasını bitirdiği zaman Orijinal toplam karakter boyutu(byte cinsinden) ve Huffman kodlama ile yapılacağı zaman toplam byte sayısı ile birlikte ekrana bastırılır. Bunun yanınnda karakterler ve bunların kaç bitle ifade edildikleri de ekrana bastırılır. Bu sıkıştırma işlemini dosyaya yansıtamadık. Çünkü her karakter için farklı bitlerle ifade edildiğinden bu yapı için hangi veri tipini kullancağımızı çözemedik. Huffman algoritması çalışmasını tamamladığı zaman oluşacak çıktı aşağıdaki gibidir.




Şekil-7: Huffman algoritmasının sonucunun gösterimi.

## III. Programın Çalıştırılması

Uygulamayı çalıştırmak için öncelikle kaynak kodumumuz bulunduğu klasöre "metin.txt" dosyasının ekleyip derleyip çalıştırdığımız zaman LZ77 algoritması sıkıştırdığı metni kodlu.txt dosyasında saklar. Fakat Huffan algoritmasının sonucu ise direkt ekrana yazdırılır.

## IV. GERÇEKLEŞTİRİLEN TESTLER VE DEĞERLENDİRME

Proje test edilmiş olup eğer gerekli dosyanın verilmesi taktirde yukarda belirtiğimiz işlemler sorunsuz olarak gerçekleşecektir. Bu uygulama linux ve windows işletim sistemlerinde test edilmiştir.

## V. ÖZET VE SONUÇLAR

Özetle, projede belirtilen LZ77 algoritması sorunsuz olarak çalışmakta olup gerekli koşulları sağlamaktadır. Deflate algoritması ise bloklarda yaşadığımız problem sebebi ile sadece Huffman algoritması yapılmıştır. Gerekli çalışmaların ve değerlendirilmelerin yapılması sonucu Deflate algoritması tamamlanabilir.

## KAYNAKÇA

[1]http://bilgisayarkavramlari.sadievrenseker.com/2009/02/25/huffman-kodlamasi-huffman-encoding/
[2]https://www.youtube.com/watch?v=co4_ahEDCho
[3] https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
[4] https://en.wikipedia.org/wiki/LZ77_and_LZ78
[5] https://www.slideshare.net/veysiertekin/lz77-lempelziv-algorithm
[6]https://www.cs.helsinki.fi/u/tpkarkka/opetus/12k/dct/lecture07.pdf