PROGRAMLAMA LABORATUVARI 2
2. PROJE

Yunus Emre GÜL
Bilgisayar Mühendisliği Bölümü
Kocaeli Üniversitesi
yunemregul@gmail.com

Özet

Bu doküman Programlama Laboratuvarı 2 dersi 2. Projesi için çözümümü açıklamaya yönelik oluşturulmuştur. Dökümanda projenin tanımı, çözüme yönelik yapılan araştırmalar, kullanılan yöntemler, proje sürecinde karşılaşılan problemler, proje hazırlanırken kullanılan geliştirme ortamı gibi programın oluşumunu açıklayan başlıklara yer verilmiştir. Doküman sonunda proje sonucu ve projemi hazırlarken kullandığım kaynaklar bulunmaktadır.

1. Proje Tanımı

1.1. Proje Tanımı

Projede LZ77 ve DEFLATE sıkıştırma algoritmalarını kullanarak belirli bir dosyayı sıkıştırmamız ve bu iki algoritma arasındaki performans farkını görmemiz beklenmektedir.

İki algoritma da kayıpsız sıkıştırma algoritmalarındandır. LZ77 algoritması kendi başına bir algoritma olmakla beraber, DEFLATE algoritması LZSS / LZ77 ve Huffman Kodlaması birleşiminden oluşmaktadır. Bu projede bizden LZSS algoritması kullanmamız istenmiştir. LZSS algoritması da LZ77 algoritmasının geliştirilmiş halidir.

Proje C ya da C++ dillerinden bir tanesi kullanılarak gerçekleştirilebilir.

1.2. İsterler

Proje tanımında belirtilen kurallar ve isterler aşağıdaki gibidir:

• Girdi olarak kullanılacak sıkıştırılmamış dosya ‘metin.txt’ şeklinde olacaktır.

• Sıkıştırılacak dosyadaki her bir karakter okunacaktır.

• En son her bir sıkıştırma algoritması için çıktı dosyası (metin dosyasının sıkıştırılmış hali) oluşturulacaktır.

• Kullandığınız sıkıştırma algoritmalarının performansları birbiri ile karşılaştırılacaktır.

2. Araştırmalar ve Yöntem

Projemi sağlayacağı kolaylıkları düşünerek C++ ile yapmaya karar verdim.

Araştırmalarıma LZ77 ve DEFLATE algoritmalarının ne olduğunu öğrenmekle başladım. LZ77 algoritmasının nasıl çalıştığını anlamak zor olmadı. Ancak DEFLATE algoritması üzerine neredeyse hiç düzgün kaynak bulamadım. Bulduklarım düzgünse de ben anlayamadım. Hangi kısmını anlayamadığıma detaylıca değineceğim.

LZ77 algoritması çalışma biçiminden dolayı ‘kayan pencere’ olarak da biliniyor. Pencere ‘arama tamponu’ ve ‘ileri tampon’ olmak üzere iki bölümden oluşuyor. Arama tamponu, metindeki şu ana kadar sıkıştırılan karakterlerin olduğu kısımdır. İleri tampon denen kısım da henüz sıkıştırılmamış, önümüzdeki karakterler anlamına gelir. Arama tamponu ‘sözlük’ olarak da bilinir.

İleri tampondaki her bir karaktere denk gelindiğinde şimdiye kadar denk gelinen karakterleri yani arama tamponunu gezip eşleşme var mı diye kontrol edilir. Eşleşme var ise en uzun olanı bulunur. Arama tamponuna eklenecek karakter ile en uzun eşleşme arasındaki mesafeye *offset* denir. Eşleşmenin uzunluğuna *length* denir. Bulunan *offset* ve *length* değerleri bir düğüm gibi, <*offset, length, eşleşme bitimindeki karakter*> şeklinde oluşturulur. Eğer eşleşme yoksa *offset* ve *length* değerleri 0 olacağından yapı <*0,0,karakter*> şeklinde olur. Oluşturulan bu düğüm gibi yapıya *token* adı verilir. Eğer *offset* değeri arama tamponunda bir yeri gösteriyorsa buna referans tokeni denmektedir.

LZ77 algoritmasında sıkıştırma yaparken oluşan sorun her karakterin en az 2 byte’lik bir tokenle temsil edilmesinden doğar. Ancak referans tokeni olmayan tokenler için 1 byte’lik alan israf edilmiş olur. Bunu önlemek için LZ77 algoritması geliştirilerek LZSS algoritması elde edilmiştir.

LZSS algoritmasındaki fark LZ77’de ki gibi her karakteri en az 2 byte lik tokenler ile ifade etmektense tokenin başına bir bayrak *(flag)* eklenerek bu tokenin arama tamponundaki bir yere

referans mı yoksa sadece bir karakter olduğu mu belirlenir. Eğer sadece bir karakter ise bu tokenin boyutu 1 bitlik bir *flag* ve 1 bytelik bir karakter olmuş olur. Böylece LZ77'ye göre tasarruf edilmiş olunur.

DEFLATE algoritması hakkındaki araştırmalarıma değinecek olursam, DEFLATE algoritması LZ77 veya LZSS algoritmasının Huffman kodlaması ile birleştirilmesi şeklinde ifade ediliyor. Ancak sorun şu ki bu birleştirmenin ne şekilde olacağına yönelik tam bir açıklamaya denk gelmedim, bulduklarımı tam olarak anlayamadım. Bulduğum açıklamalar sıkıştırılacak verinin 'blok'lar halinde işlenmesi gerektiğinden bahsediyor. Her bloğun LZSS algoritması ve Huffman kodlamasının birlikte kullanılması ile sıkıştırıldığı söyleniyor. Bir kaynaktan direkt alıntı yapacak olursam:

"Her blok için oluşturulan Huffman ağacı bir önceki ve bir sonraki bloktan bağımsızdır. Sıkıştırılabilen blokların büyüklüğü değişkendir. Deflate algoritması Huffman ağacının etkili kodlama yapamayacak kadar büyüdüğünü gördüğünde, yeni bir Huffman ağacı oluşturmak için o bloğu sonlandırarak yeni bir blok başlatır. Sıkıştırılamayan blokların boyu ise 65.535 byte ile sınırlıdır. "

Altan Mesut, Doktora Tezi

Bu açıklamadan neredeyse hiç bir şey anlamadım. Öncesinde de bahsettiğim gibi bir blok nedir, sıkıştırılabilen, sıkıştırılamayan bloklar nedir, blokların sıkıştırma modu (1: sıkıştırılmamış, 2: Huffman ve LZSS ile sıkıştırılmış vs.) ne ifade ediyor? gibi daha nice sorular. Deflate ile ilgili bu kaynak gibi bir çok yabancı kaynağa da ulaştım ancak bulduklarım yine benzer şekilde detaylandırmadan kaba açıklamalar yapıyor. Belki konuyla ilgili eksik bilgiye sahibimdir, bu yüzden anlamamış olabilirim ancak yine de deflate işlemi için yapılan açıklamaların yeterince anlaşılır olduğunu düşünmüyorum.

Deflate üzerinde bu kadar zorluk yaşamam projenin bu kısmını yetiştirememeye yol açtı. Bu yüzden projemde sadece LZ77 ve LZSS algoritmalarının uygulaması ve karşılaştırması bulunmakta.

2.1. Karşılaşılan Problemler

Projede karşılaştığım bir problem *struct* ve *class* gibi yapıların içine koyulan verilerin boyutuna göre '*padding*' almasına dayanıyor. Örneğin Şekil 1'deki gibi bir *struct* yapımız olsun:

```
struct a
{
    uint16_t b;
    char c;
}
```

Şekil 1. Padding anlatımı için örnek yapı

Bu yapının kaplayacağı boyutu düşünecek olursak *uint16_t* tipi 2 byte alan kaplar, *char* tipi 1 byte alan kaplar. Normalde bu yapının toplamda 3 byte yer kaplaması gerektiğini düşünürken aslında öyle olmadığını ve bellekte düzgün saklanabilmesi için *padding* işlemi uygulandığını, yapının 4 byte ile temsil edildiğini öğrenmiş oldum. Bunun yol açtığı sorun örneğin bu yapıyı bir dosyaya yazacak olsam *padding*lerinden dolayı yapının her tanesi 4 byte lik alan kaplıyordu.

Bu problemi çözmek için yapının içindeki elemanları tek tek dosyaya yazdım böylece önce 2 byte daha sonra 1 byte ve toplamda 3 byte yazmış olarak problemi çözdüm.

Karşılaştığım bir diğer problem de LZSS algoritmasını uygularken bayrak *(flag)* verisinin boyutunun nasıl 1 bit olacağıydı. Bildiğim kadarı ile 1 biti temsil eden bir veri tipi yoktu ancak daha sonra C++ 'da bitfields isimli bir özellik olduğunu keşfettim. Bu özellik sayesinde flag değerini 1 bit olarak sakladım (Şekil 2).

```
class lzss_token
{
    public:
        unsigned flag : 1;
        uint16_t offset_length;
        char c;

        lzss_token();
```

Şekil 2. Bitfields özelliği kullanarak veri boyutunu istenen bitte belirlemek

LZSS uygularken karşılaştığım büyük bir problem de dosyalara bit şeklinde yazım yapılamamasıydı. Yani bunun anlamı *flag* değerini kendi başına yazamıyordum. Bunu çözmek için *flag* değerlerini 1 byte lik bölümler halinde tutup o şekilde yazdım. Örneğin normalde flagden hemen sonra tokenin referans gösterdiği *offset* ve *length* ya da bir karakter gelmesi gerekir. Benim çözüm yöntemimde 8 tane flag ardından 8 tane ilgili token geliyor. Böylece flagleri ve tokenleri 'chunk' lar halinde tutuyorum, flag eri yazmak mümkünleşiyor.

2.2. Kazanımlar

Bu projeyle birlikte daha önce hiç tecrübem olmayan sıkıştırma algoritmalarından bazılarını deneyimlemiş oldum. Ayrıca daha önce hiç bir

projemde C++ dilini kullanmamıştım bu projeyle kullanmış ve kendimi geliştirmiş oldum.

Sıkıştırma algoritmalarının nasıl çalıştığına yönelik fikir edindim. Sıkıştırmadan doğan problemleri ve bunlara nasıl optimizasyonlar getirildiğini (LZSS) öğrenmiş oldum.

3. Geliştirme Ortamı

Projemi Linux sistemde, Visual Studio Code üzerinde geliştirdim. Projemin gelişimini ve versiyonlarını takip edebilmek için de Git versiyon kontrol sistemi kullandım.

4. Kod Bilgisi

4.1. Akış Diyagramı

Kısım ektedir. [1](akis_diyagrami.jpg)

4.2. Projenin Derlenmesi ve Çalıştırılması

Projenin hem Windows hem de Linux için derlenmiş hali dosyalarda mevcuttur. Yine de kendiniz derlemek isterseniz g++ kullanarak derlemek için aşağıdaki komutu komut satırına girebilirsiniz:

g++ main.cpp lz77.cpp lzss.cpp -o program

Ardından ‘program’ isimli dosya çalıştırılabilir. Çalıştırırken dikkat edilmesi gereken programın olduğu klasörde metin.txt isimli bir dosya olmalıdır. Program çalıştırıldığında metin.txt dosyasını LZ77 ve LZSS algoritmaları ile sıkıştırarak lz77_encoded.bin ve lzss_encoded.bin çıktı dosyalarını oluşturacaktır.

Çalışmasına dair fotoğraflar ekte bulunabilir. [2] (readme.pdf)

4.3. Sonuç

Projenin gerektirdiği DEFLATE algoritmasını bölüm 2’de de belirttiğim gibi tamamlayamadım. Projemde sadece LZ77 ve LZSS algoritmalarının uygulanışı ve karşılaştırılması görülebilir.

Kaynakça

1. LZ77 algoritmasını anlamak: https://sites.google.com/site/datacompressionguide/lz77 https://ysar.net/algoritma/lz77.html https://en.wikipedia.org/wiki/LZ77_and_LZ78 https://towardsdatascience.com/how-data-compression-works-exploring-lz77-3a2c2e06c097
2. LZSS algoritmasını anlamak: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymanski http://michael.dipperstein.com/lzss/
3. DEFLATE algoritması hakkında bilgi: https://zlib.net/feldspar.html https://www.quora.com/How-does-the-DEFLATE-compression-algorithm-work
4. LZ77, LZSS, DEFLATE hakkında genel bilgi edinmek: http://altanmesut.trakya.edu.tr/pubs/DR_Tez.pdf
5. Çeşitli diğer problemler: https://stackoverflow.com/