Projektni Zadatak 1

Kompresija i zastita podataka

Projektni zadatak 1

Emilija Ristic 114/2018

Za generisanje fajl-a potrebno je u terminalu pokrenuti komadnu:

./creatingFileBase64.ps1

Za pokretanje samog programa potrebno pokrenuti komandu u terminalu:

python -X utf8 .\projektni_zadatak_1.py

Izracunavanje bajt-entropije binarnog fajla i implementiranje osnovnog algoritma za kompresiju

Kreiranje .txt fajla za potrebnu implementaciju

Korišćenjem PowerShell skripte koja koristi RNGCryptoServiceProvider kako bi generisala slučajne bajtove, kodira ih u BASE64 formatu i zapisuje u fajl.txt. Skripta će nam omogućiti da generisani fajl bude u rasponu od 1MB do 10MB.

```
[System.Reflection.Assembly]::LoadWithPartialName("System.Web") | Out-Null
$rand = New-Object System.Security.Cryptography.RNGCryptoServiceProvider
$bytes = New-Object Byte[] 1048576
$rand.GetBytes($bytes)
$base64string = [System.Web.HttpServerUtility]::UrlTokenEncode($bytes)
$base64string | Out-File -FilePath file.txt -Encoding ascii
```

Izracunavanje Entropije Fajla

Entropija je mera "nereda" koja nam daje informacije o uspešnosti mogućnosti kompresije našeg fajla. Veća entropija nam ukazuje da su naši podaci manje predvidivi.

Formula za izracunavanje entropije:

```
H(X) - entropija slucajne promenljiva X koja nam pokazuje prosecnu kolicinu informacija po jedinici podataka

p(xi) - verovatnoca da se odreceni znak pojavi (ucestalost svakog znaka u nasem fajlu)

log_2(p(xi)) - daje nam informaciju da izracunamo koliko informacija donosi svaki znak fajl-a

sumiranje ovih vrednosti nam daje ukupnu kolicinu informacija.
```

$$H(X) = -\sum i = 1np(xi) \cdot log2(p(xi))$$

```
import os
import math

def izracunajEntropiju(putanja_do_fajla):
    duzina = 0
    buff = {}

    with open(putanja_do_fajla, 'rb') as file:
        while True:
            content = file.read(1)
```

```
            if not content:
                break
            byte = ord(content)
            buff[byte] = buff.get(byte, 0) + 1
            duzina += 1

    H = 0
    for broj_ponavljanja in buff.values():
        if broj_ponavljanja:
            Hi = broj_ponavljanja / duzina
            H -= Hi * math.log2(Hi)

    return H

putanja_do_fajla = os.path.join('', 'file.txt')
entropy = izracunajEntropiju(putanja_do_fajla)
print("Entropija naseg fajl-a iznosi:", entropy)
```

```
output:
Entropija naseg fajl-a iznosi: 5.999981674117314
```

Rezultat naše entropije od približno 6 bita po bajtu nam govori da su podaci gotovo maksimalno slučajni i, u kontekstu kompresije, moguće je da možemo izvršiti minimalnu kompresiju našeg fajla.

Shannon-Fano Algoritam

Shannon-Fano kodiranje je metoda za kompresiju podataka koja koristi princip razdvajanja skupa simbola metodom “odozgo nadole”. Početni skup delimo na dva skupa gde su skupovi podeljeni na osnovu učestalosti. Nakon toga primenjujemo kodiranje skupa simbola. Svaki od skupova se rekurzivno deli na manje skupove i koristi se kodiranje svakog skupa simbola. Na kraju dobijeni konačan kod za svaki simbol u tekstu gde svaki kod ima dužinu u zavisnosti od učestalosti simbola. Simboli koji su učestali dobiće kraće kodove, dok simboli koji nisu tako učestali dobiće duže kodove.

```
import math

class Simbol:
    def __init__(self, karakter, verovatnoca):
        # klasa simbol ima karakter koji se kodira
        # za svaki karakter računamo verovatnoću pojavljivanja karaktera i
        # code koji će nam služiti za kodiranje
        self.karakter = karakter
        self.verovatnoca = verovatnoca
        self.code = ""

def shannonFanoKodiranje(podaci):
    # broj ponavljanja određenog karaktera
    broj_ponavljanja = {}
    for karakter in podaci:
        broj_ponavljanja[karakter] = broj_ponavljanja.get(karakter, 0) + 1

    ukupan_broj_karaktera = len(podaci)

    # lista simboli = predstavlja jedan karakter sa njegovom verovatnoćom
    simboli = [Simbol(karakter, count / ukupan_broj_karaktera) for karakter, count in broj_ponavljanja.items()]

    # sortiranje simbola u opadajućem rasporedu prema njihovoj verovatnoći
    simboli.sort(key=lambda x: x.verovatnoca, reverse=True)

    def ponoviRekurzivno(simboli):
        if len(simboli) == 1:
            return

        # računamo ukupnu verovatnoću svih simbola u trenutnom skupu
        ukupna_verovatnoca = sum(sym.verovatnoca for sym in simboli)

        _verovatnoca = 0
```

```
        # razlika između ove dve vrste verovatnoća meri koliko su skupovi blizu po svojoj verovatnoći
        # Deljenje skupa na dva skupa i poređenje njihove verovatnoće nam govori o tome koliko će naše kodiranje biti efikasno
        min_diff = float('inf')

        split_index = 0

        for i in range(len(simboli) - 1):
            _verovatnoca += simboli[i].verovatnoca
            diff = abs((ukupna_verovatnoca - _verovatnoca) - _verovatnoca)
            if diff < min_diff:
                min_diff = diff
                split_index = i

        # dodeljujemo "0" i "1" kodovima u levoj i desnoj polovini
        for i in range(split_index + 1):
            simboli[i].code += "0"
        for i in range(split_index + 1, len(simboli)):
            simboli[i].code += "1"

        # ponavljamo rekurzivno za levu i desnu stranu skupa
        ponoviRekurzivno(simboli[:split_index + 1])
        ponoviRekurzivno(simboli[split_index + 1:])

    ponoviRekurzivno(simboli)

    # kreiranje rečnika simbola za kodove
    dict_kodova = {sym.karakter: sym.code for sym in simboli}

    print("dict_kodova", dict_kodova)
    encoded_podaci = ''.join(dict_kodova[karakter] for karakter in podaci)

    return encoded_podaci, dict_kodova

def sacuvajEncodedPodatke(input_file_path, output_file_path):
    # Čitaj txt fajl
    with open(input_file_path, 'r', encoding='utf-8') as file:
        podaci = file.read()

    # Primeni shannon fano kodiranje
    encoded_podaci, dict_kodova = shannonFanoKodiranje(podaci)

    with open(output_file_path, 'wb') as file:
        for karakter, code in dict_kodova.items():
            file.write(f"{ord(karakter)}:{code}\n".encode())  # Sačuvaj karaktere kao kodove
        file.write(b"\n")

        padded_encoded_podaci = encoded_podaci + '0' * ((8 - len(encoded_podaci) % 8) % 8)
        byte_array = bytearray()
        for i in range(0, len(padded_encoded_podaci), 8):
            byte = int(padded_encoded_podaci[i:i+8], 2)
            byte_array.append(byte)

        file.write(byte_array)

input_file_path = 'file.txt'
output_file_path = 'encoded_fajl_shannon_fano.bin'
sacuvajEncodedPodatke(input_file_path, output_file_path)
```

Huffmanov Algoritam

Primjenjujemo Huffmanov algoritam kompresije koji ima pristup “odozdo nadole”. Za razliku od Shannon-Fano algoritma gde smo simbole delili na osnovu trenutne verovatnoće, Huffman koristi stablo i prioritetni red za spajanje simbola sa najmanjim ponavljanjima. Huffmanov algoritam daje bolje rezultate jer gradi optimalno stablo za kodiranje, dok Shannon-Fano može biti manje precizan.

```
class Simbol:
    def __init__(self, karakter, frekvencija):
        # karakter koji se kodira
        # broj ponavljanja datog karaktera u taksu
```

```
        # kod koji ce biti dodeljen karakteru tokom Huffmanovog kodiranja
        # pointeri levo,desno za stablno koje gradimo

        self.karakter = karakter
        self.frekvencija = frekvencija
        self.code = ""
        self.levo = None
        self.desno = None

def huffmanKodiranje(podaci):
    # racunamo broj pojavljivanja svakog simbola u ulaznim podacima
    frekvencija = {}
    for karakter in podaci:
        if karakter in frekvencija:
            frekvencija[karakter] += 1
        else:
            frekvencija[karakter] = 1

    # kreiramo listu objekata za svaki karakter
    simboli = [Simbol(karakter, frekvencija) for karakter, frekvencija in frekvencija.items()]

    # sortiranje simbola na osnovu broja pojavljivanja tako da simbol
    # sa najmanjim brojem ponavljanja bude prvi
    simboli.sort(key=lambda x: x.frekvencija)

    # sve dok imamo vise od jednog simbola potrebno je da uklanjamo
    # one simbole koji imaju najmanji broj ponavljanja
    # na taj nacin kreiramo stablo i zbir ponavljanja postavje novi cvor
    # dodajemo novi cvor u listu i ponavljamo sortiranje
    while len(simboli) > 1:
        levo = simboli.pop(0)
        desno = simboli.pop(0)

        novi_cvor = Simbol(None, levo.frekvencija + desno.frekvencija)
        novi_cvor.levo = levo
        novi_cvor.desno = desno

        simboli.append(novi_cvor)
        simboli.sort(key=lambda x: x.frekvencija)

    # dodeljujemo kodove svakom simbolu
    # simboli koji se nalaze levo u stablu se kodiraju sa "0",
    # dok se simboli koji se nalaze desno u stablu kodiraju sa "1"

    def dodeliKodove(cvor, kod=""):
        if cvor is None:
            return
        if cvor.karakter is not None:
            cvor.code = kod
            dict_kodova[cvor.karakter] = kod
        dodeliKodove(cvor.levo, kod + "0")
        dodeliKodove(cvor.desno, kod + "1")

    dict_kodova = {}

    dodeliKodove(simboli[0])

    encoded_podaci = ''.join(dict_kodova[karakter] for karakter in podaci)

    return encoded_podaci, dict_kodova

def sacuvajEncodedPodatke(input_file_path, output_file_path):
    with open(input_file_path, 'r', encoding='utf-8') as file:
        podaci = file.read()

    encoded_podaci, dict_kodova = huffmanKodiranje(podaci)

    with open(output_file_path, 'wb') as file:
        for karakter, code in dict_kodova.items():
            file.write(f"{ord(karakter)}:{code}\n".encode())  # Sačuvaj karaktere kao kodove
        file.write(b"\n")
```

```
        padded_encoded_podaci = encoded_podaci + '0' * ((8 - len(encoded_podaci) % 8) % 8)
        byte_array = bytearray()
        for i in range(0, len(padded_encoded_podaci), 8):
            byte = int(padded_encoded_podaci[i:i+8], 2)
            byte_array.append(byte)

        file.write(byte_array)

input_file_path = 'file.txt'
output_file_path = 'encoded_fajl_huff.bin'
sacuvajEncodedPodatke(input_file_path, output_file_path)
```

LZ77

LZ77 algoritam kompresije podataka koristi principe pretraživanja i pronalaženja ponovljenih segmenata u tekstu kako bi smanjio veličinu podataka, koristeći klizni prozor za praćenje karaktera i optimizaciju kompresije.

```
# koristi klizni prozor za pronalazenje ponovljenih sekvenci
# i kodira ih (distanca, duzina, sledeci_karakter)
def lz77_compress(podaci, velicina_prozora=255, lookahead_buff=15):
    kompresovani_podaci = []
    i = 0
    n = len(podaci)

    while i < n:
        duzina_ponavljanja = 0
        distanca = 0
        start = max(0, i - velicina_prozora)
        for j in range(start, i):
            duzina = 0
            while (i + duzina < n) and (podaci[j + duzina] == podaci[i + duzina]) and (duzina < lookahead_buff):
                duzina += 1
            if duzina > duzina_ponavljanja:
                duzina_ponavljanja = duzina
                distanca = i - j

        if duzina_ponavljanja > 0:
            if i + duzina_ponavljanja < n:
                kompresovani_podaci.append((distanca, duzina_ponavljanja, podaci[i + duzina_ponavljanja]))
            else:
                kompresovani_podaci.append((distanca, duzina_ponavljanja, 0))  # Koristi 0 ako nema više karaktera
            i += duzina_ponavljanja + 1
        else:
            kompresovani_podaci.append((0, 0, podaci[i]))
            i += 1

    return kompresovani_podaci
```

Nakon što smo obavili kompresiju podataka primenom algoritma LZ77, potrebno je izvršiti dekompresiju. Originalni podaci se obnavljaju korišćenjem informacija o distanci i dužini ponavljanja iz kompresovanih podataka.

```
# rekonstruise originalni tekst na osnovu (distanca, duzina, sledeci_karakter)
def lz77_decompress(kompresovani_podaci):
    dekompresovani_podaci = []
    for distanca, duzina, next_karakter in kompresovani_podaci:
        start = len(dekompresovani_podaci) - distanca
        for i in range(duzina):
            dekompresovani_podaci.append(dekompresovani_podaci[start + i])
        dekompresovani_podaci.append(next_karakter)
    return bytes(dekompresovani_podaci)
```

LZW

LZW algoritam za kompresiju koristi dinamički rečnik koji se gradi tokom same kompresije podataka. Na početku se implementira rečnik sa pojedinačnim karakterima. Kada naiđe na sekvencu koja se ne nalazi u rečniku, dodaje je i zamenjuje njenu sekvencu sa kodom.

Razlika između prethodno pomenutog LZ77 i LZW algoritma je u načinu kodiranja podataka. LZ77 koristi prozor za pretragu ponavljajućih karaktera, dok LZW gradi dinamički rečnik tokom same kompresije.

```
def lzwKomp(podaci):
    recnik = {bytes([i]): i for i in range(256)}
    preth = bytes()
    rezultat_kompresije = []
    code = 256

# dinamicki azuriranje recnika i generisanje kompresovanih podataka
    for karakter in podaci:
        sekvenca_karaktera = preth + bytes([karakter])
         # proveravamo da li nova sekvanca karaktera postoji u recniku, ako postoji
         # potrebno je da azuriramo prethnodnu
        if sekvenca_karaktera in recnik:
            preth = sekvenca_karaktera
        # ako sekvenca karaktera ne postoji dodajemo kod prethodnika
        # u rezulata, sekvenca se dodaje u recnik sa novim kodom i kod uvecavamo za 1
        # azurairamo prethodnika sa na treuntni karakter
        else:
            rezultat_kompresije.append(recnik[preth])
            recnik[sekvenca_karaktera] = code
            code += 1
            preth = bytes([karakter])

    if preth:
        rezultat_kompresije.append(recnik[preth])

    return rezultat_kompresije
```

Potrebno je da odradimo dekompresiju kompresovanog fajla kako bi rekonstruisali originalne sekvence podataka koje su bile kompresovane.

Novi podaci se dodaju u recnik kako bi kasnije mogli da ih prepoznamo i koristimo za dekodiranje.

```
def lzwDec(kompresovani_podaci):
    # kreirano recnik i mapiramo karaktere
    recnik = {i: bytes([i]) for i in range(256)}
    # prethodnik je pocetna sekvenca koja se postavlja na prvi kod
    preth = bytes([kompresovani_podaci[0]])
    rezultat_dekompresije = [preth]
    code = 256

    for trenutni in kompresovani_podaci[1:]:
        # ako treutna sekvenca postoji u recniku,
        if trenutni in recnik:
            sekvenca = recnik[trenutni]
        elif trenutni == code:
            sekvenca = preth + preth[0:1]
        else:
            raise ValueError(f'Bad compressed trenutni: {trenutni}')

        rezultat_dekompresije.append(sekvenca)
        recnik[code] = preth + sekvenca[0:1]
        code += 1
        preth = sekvenca

    return b''.join(rezultat_dekompresije)
```