Kódolás és IT biztonság (VIHIBB01)
Laboratóriumi gyakorlat

Veszteségmentes és veszteséges tömörítés

Szerző:
Biczók Gergely
Lestyán Szilvia

2020. november 27.

Tartalomjegyzék

1. Labor célja

A labor gyakorlat során veszteséges és veszteségmentes tömörítési eljárásokkal fog megismerkedni. A cél, hogy mindkét típusú eljárás alapvető működését és erősségeit megértse, és azokat Python környezetben, gyakran használt könyvtárak segítségével, a való életből vett problémákon alkalmazza.

A munka során különböző fájlokat fog veszteségmentesen és veszteségesen tömöríteni a `zlib`[1] és a `opencv`[2] könyvtár segítségével.

2. Háttéranyag

A megnövekedett állomány méretek miatt az adatok hordozhatósága és tárolása (archiválása) rendkívül körülményessé vált. Korábban előfordult, hogy egy rendszer adatállományai több doboz hajlékonylemezre vagy később CD-re fértek csak rá. A problémát nem oldották meg a nagyobb háttértároló kapacitású egységek megjelenései sem, mivel mindig egyre több és egyre nagyobb fájl került a számítógépekre, azok fejlődésével. Ezzel párhuzamosan, a számítógép-hálózatok megjelenésével, felmerült az az igény is, hogy minél gyorsabban és hatékonyabban tudjunk adatokat mozgatni hálózati végpontok között. Hasonlóan a tároláshoz, a hálózati sávszélesség növekedésével az atviendő adat mennyisége is nőtt, tehát a probléma továbbra is jelen van. Mindkét esetben ésszerű megoldást kínál a tömörítés.

Az adathalmazok általában redundánsak, terjengősek, nem a lehető legrövidebbek, legtömörebbek. Sokszor ugyanazt az információt rövidebben is le lehet írni megfelelően kódolva: ez az adatok tömörítésének lehetősége. A tömörítés lényege, hogy az adatállományokban szereplő adatszekvenciák között lehetnek ismétlődések, és a tömörítő programok algoritmusai ezt használják ki. A tömörítő eljárások segítségével adatainkat olyan alakra hozhatjuk, amelynek kisebb az adatmennyisége, mint az eredetinek, gyakran csak töredéke. Kisebb helyet foglal az adathordozón, rövidebb idő alatt továbbítható a hálózaton. A felhasználáshoz azonban általában vissza kell alakítani az eredeti formátumra, ezt hívjuk kitömörítésnek vagy dekódolásnak.

[1] https://docs.python.org/3/library/zlib.html

[2] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html

2.1. Veszteségmentes tömörítés

A veszteségmentes tömörítés olyan kódolás, aminek eredményeként a létrejött kódolt (tömörített) jelhalmaz rövidebb, mint az eredeti, azaz kisebb az adatmennyisége, és a tömörített adathalmazból tökéletesen visszaállítható az eredeti dokumentum, nem veszítünk információt. Tehát a veszteségmentes tömörítés lehetővé teszi a tömörített adatból az eredeti adatok pontos rekonstrukcióját. Akkor alkalmazzuk, ha fontos, hogy az eredeti és a kitömörített adat bitről bitre megegyezzen. Tipikus példák a szövegfájlok, a futtatható állományok vagy adatbázisok. Néhány képformátum, köztük a PNG (vektorgrafikus képekre hatékony) vagy a GIF (bitmap képekre) is csak veszteségmentes tömörítést használ, míg egy TIFF fájl veszteséges és veszteségmentes tömörítést is tartalmazhat.

Tömörítési arány

Fontos megjegyezni, hogy a tömörítés hatékonysága nagyban függ mind a választott algoritmustól, mind a bemenettől. Olyan eset is előfordulhat, amikor a tömörített adat mérete nagyobb mint az eredetié: pl. ha fix méretű kódtáblát használunk, de a bemeneti adatban nagyon kicsi a redundancia. Egy algoritmus-bemenet páros tömörítési hatékonyságát jellemezhetjük a tömörítési aránnyal (r_c):

$$r_c = \frac{s_c}{s_s}$$

ahol s_c az állomány mérete tömörítés után, és s_s az állomány eredeti mérete.

Példa: futamhossz-kódolás

A legegyszerűbb veszteségmentes tömörítési eljárás a futamhossz-kódolás. Legegyszerűbb formájában és szöveges bemeneti fájl esetén számljuk az egymás után előforduló azonos karakterek számát, és a kimeneten az egyes karakterek és a hozzájuk tartozó számlálók szekvenciáját írjuk le. Tehát ha a bemenet a következő:

```
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWBWWWW
```

akkor a kimenet így néz ki:

```
12W1B12W3B24W1B4W
```

Ezt értelmezhetjük 12 fb "W", 1 "B", 12db "W", stb, sorozataként. Ebben az esetben (fix hosszúságú karakterkódolást feltételezve) a tömörörítési arány $r_c = \frac{18}{57} \approx 0.32$.

Fejlettebb algoritmusok

A való életben használt tömörítőprogramok hatékony eljárásokat alkalmaznak. Elemzik az állomány szerkezetét, és annak függvényében határozzák meg a használt tömörítési eljárásokat. Egy gyakran használt algoritmus a DEFLATE[3] , amely kihasználja mind az ismétlődő karaktersorozatokat az LZ77[4] algoritmuson keresztül, mind az információelméletileg optimális tömörítést, ami véletlen adatokra ad jó tömörítési arányt, a Huffman-kódolás[5] segítéségével. Ezt a két fázist egymás után alkalmazva, először kihasználva a futamhosszokat, majd ennek a kimenetét Huffman-kódolva kiváló tömörítési arány érhető el.

A DEFLATE algoritmusra épít több elterjedt tömörítőprogram is, például a `gzip`. Hasonló néven ismerik a program kimeneteként előálló fájlformátumot is, amely áll:

- egy 10 bájtos fejrész (header), melyben van egy mágikus szám (magic number), egy verziószám és egy időbélyeg
- Opcionális extra fejrészek, benne például az eredeti fájlnév,
- egy belső rész (body), benne a DEFLATE-tel tömörített felhasználói adat (payload)
- egy 8 bájtos lábrész (footer), benne egy CRC-32 checksum és az eredeti tömörítetlen adat hossza

2.2. Veszteséges tömörítés

A veszteséges tömörítés az adattömörítési algoritmusok egy osztálya, ami a veszteségmentes tömörítéssel ellentétben nem teszi lehetővé a tömörített adatból az eredeti adatok pontos rekonstrukcióját, ám egy „elég jó” rekonstrukciót igen. Gyakran használják az interneten is, a telefóniás és streamelési

[3] https://en.wikipedia.org/wiki/DEFLATE
[4] https://en.wikipedia.org/wiki/LZ77_and_LZ78
[5] https://en.wikipedia.org/wiki/Huffman_coding

alkalmazásokban. A veszteséges tömörítés olyan kódolás, aminek eredményeként létrejött kódolt (tömörített) jelhalmaz sokkal rövidebb, mint az eredeti, azaz kisebb az adatmennyisége, de a tömörített adathalmazból nem állítható tökéletesen vissza az eredeti, csak jó közelítéssel → információt vesztünk. A veszteséges módszerek használatának az az előnye a veszteségmentes módszerekhez képest, hogy sok esetben a veszteséges tömörítés sokkal kisebb fájlt képes előállítani, mint bármely veszteségmentes, és még így is kellően jó minőséget ér el. Veszteséges tömöritésre példák: JPEG, JPEG2000, Fraktál tömörítés.

JPEG

A JPEG – (Joint Photographic Experts Group) képek tárolására alkalmas fájlformátum. Főleg digitális fényképezőgépek által készitett digitális képeket tárolnak ebben a formátumban - amennyiben veszteséges formátumot használnak. Kiterjesztéseként a .jpeg, .jpg, ritkábban a .jpe használt.

Egy képen lévő információt veszteségesen tömöríti ez a formátum. Bár a tömörítés információveszteséggel jár, akár 10-100× kisebb fájlméret mellett is élvezhető a tömörített kép. Elsősorban fényképek, rajzok tárolására való. Grafikonok és egyéb hirtelen színátmenetű ábrák tárolására nem alkalmas, ezekre veszteségmentes tömörítésű formátum való (például PNG, GIF). A JPEG-ben nem képpontokat tárolnak le, hanem a képet transzformálják a frekvencia-tartományba a DCT-vel (diszkrét cosinus transzformáció).

PSNR

A csúcsjel-zaj arány (PSNR) a jel maximális teljesítménye és a jel zaja közötti arány. A mérnökök általában a PSNR-t használják a tömörített képek minőségének mérésére. Az egyes képelemek (pixel) színértékkel rendelkeznek, amely megváltozhat, amikor egy képet tömörít, majd nem tömörít. A jeleknek széles dinamikai tartománya lehet, tehát a PSNR általában decibelben fejeződik ki, ami egy logaritmikus skála. Kiszámitása:

$$\mathrm{PSNR} = 10 \cdot \log_{10}(255^2/\mathrm{MSE})$$
$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

ahol 255 a maximális szürkeegyensúly eltérés lehet egy képben 8 biten tárolva, Y_i és $\hat{Y}_i$ pedig a két összehasonlitandó kép egyes pixelei. Legyen a $\mathrm{PSNR}_1 =$

35dB és a $PSNR_2 = 39dB$ két referenciaérték. $PSNR_1$ egy közepesen jó képet eredményez, $PSNR_2$ pedig már jónak nevezhető.

Színterek

A színterek a színek ábrázolására használható virtuális térbeli koordinátarendszerek, ahol az egyes színek tulajdonságait azok koordinátái fejezik ki. Ezek általában: egy színezeti, egy világossági és egy színtelítettségi jellemzők. A színtérben az ábrázolható színek valamilyen rend szerint kerülnek elhelyezésre (például az alapján, hogy a színtér alapszíneinek milyen arányú keverésével állíthatók elő), és a pozíciójukat meghatározó koordinátákkal kerülnek azonosításra (például az RGB színtérben a (255,0,128) koordinátán a maximális piros, nulla zöld, és a maximális felének megfelelő kék komponensek összeadásából keletkező szín található). A python 3 dimenziós mátrixokként ábrázolja őket, ami egy 3 dimenziós numpy tömbnek felel meg: [magasság, szélesség,szín]. Legegyszerűbben úgy képzeljük el, hogy 3 sima mátrixot teszünk egymásra, ahol az egyes szintek az alapszineknek felelnek meg.

A YUV és az YCbCr a videóban használt RGB jelek különbségkódolása, azaz a piros (Red), a zöld (Green) és a kék (Blue) alapszinek különbségei. A YUV a régi analóg videóban található. A YCbCr a digitális video formátumokban található. Y az RGB súlyozott összege, az UV és a CbCr egyaránt súlyozott különbségek az RB és Y között. A súlyokat úgy tervezték, hogy az UV és CbCr semleges színeknél nulla vagy állandó legyen (fekete, szürke és fehér szín, ahol R = G = B) .Ez azt jelenti, hogy a fényerő-információt nagyrészt az Y-csatornán szállítják, a színinformációkat pedig a másik kettőben hordozzák. A videó átvitel és a tömörítés kihasználhatja ezt. A színinformáció almintába kerülhet (alacsonyabb felbontással és kisebb sávszélességet használva), a képminőség minimális veszteségével. Példa itt: this.

3. Felhasznált programkönyvtárak

3.1. zlib

A `zlib` a DEFLATE algoritmus absztrakciója programkönyvtár formában. Mind a `gzip` fájlformátumot, mind egy könnyúsúlyú adatfolyam formátumot

is támogat. `Azlib` adatfolyam, a DEFALTE algoritmus és a gzip fájlformátum is szabványosításra került: az RFC 1950, RFC 1951 és az RFC 1952 definiálja őket.

A `zlib` veszteségmentes tömörítési könyvtár alapvetően C-nyelven íródott, és rengeteg hardveren és operációs rendszeren endelkezésre áll, és maguk a `zlib` által definiált formátumok is hordozhatók platformokon át. Fontos megjegyezni, hogy i) szinte sohasem növeli meg tömörítés során az adatmennyiséget, és ii) általa felhasznált memória terület mérete nem függ a bemenettől, és csökkenthető is a tömörítési ráta rovására.

A mérés során a Python `zlib` könyvtárat fogják használni.

- Hivatalos dokumentáció:
  https://docs.python.org/3/library/zlib.html
- Tutorial (különös tekintettel a fájl tömörítésre (`zlib.compress()`) és kitömörítésre (`zlib.decompress()`):
  https://stackabuse.com/python-zlib-library-tutorial/

3.2. Opencv

Az OpenCV (Open Source Computer Vision Library) számitógépes látással kapcsolatos funkciók (pythonban opencv-python-ként installáljuk és cv2-ként hivjuk), transzformációk, függvényeket tartalmazó, nagyon széles körben elterjedt könyvtár. Felhasználási területeiből néhányat emlitve: 2D és 3D feature toolkit, arcfelismerés, ember-számitógép interakció (HCI), mozgáskövetés stb. Mi ennek a hatalmas könyvtárnak csak egy kis részével fogunk megismerkedni a gyakorlat alatt. A helyes értelmezéshez él alkalmazáshoz szükségünk lesz még a *numpy* és *math* könyvtárakra is. Alább néhány segédanyag:
https://www.tutorialspoint.com/numpy/numpy_introduction.htm
https://opencv-python-tutroals.readthedocs.io/
https://www.tutorialsteacher.com/python/math-module

4. Feladatok

4.1. Vezetett feladat

Adott egy szövegfájl, amely Jane Austen: Pride and Prejudice c. művét tartalmazza angol nyelven (`pride_and_prejudice.txt`). Tömörítse be a zlib könyvtár segítségével. A kiegészítendő program-szkeletont a `zip_todo.py` fájl tartalmazza. Figyelmesen olvassa el a kommenteket.
Olvassa be a bemeneti fájlt:

```
# read the content of the input file into a variable
called original_data
    with open(ifile_name, 'rb') as f:
        original_data = f.read()
```

Végezze el a tömörítést a megfelelő szinttel:

```
    #main compression part
    compressed_data = zlib.compress(original_data, int(level))
```

Számítsa ki a tömörítési arányt:

```
    #calculate compression ratio
    compress_ratio = (float(len(original_data)) - float(len(compressed_data)))
    / float(len(original_data))
    print('Compressed: %f %%' % (100.0 * compress_ratio))
```

Végül írja ki a tömörített adatokat egy fájlba:

```
    # write out the encoded text to the output file
    with open(ofile_name, "wb") as f:
        f.write(compressed_data)
```

Ha kész a tömörítési és a tömörítési arány kiszámolási logika, futtassa 1-es tömörítési szinttel. Mekkora a tömörítési arány, százalékban kifejezve, 2 tizedesjegy pontossággal (**Moodle 1a**)?
Most írja meg a kitömörítés kódját is.
Olvassa be a bemeneti, tömörített fájlt:

```
# read the content of the compressed input file into a
variable called input_string
    with open(ifile_name, 'rb') as f:
        compressed_data = f.read()
```

Végezze el a kitömörítést:

```
# main decompression part
decompressed_data = zlib.decompress(compressed_data)
```

Végül írja ki a kitömörített adatokat egy új fájlba:

```
# write out the decoded text to the output file
with open(ofile_name, "wb") as f:
    f.write(decompressed_data)
```

Ha kész a kitömörítési logika, tömörítse ki az előzőekben betömörített fájlt, és a kimenet méretét hasonlítsa össze az eredeti `pride_and_prejudice.txt`-vel. Mekkora a méretbeli különbség bájtban számolva (**Moodle 1b**)?

4.2. Önálló feladat

Tegyük fel, hogy egy óriási ingyenes e-book weboldalt üzemeltet, és szűkös az elérhető tárhely. Hogy ez ne okozzon gondot, egy e-book mérete maximálva van 275000 bájtban. Mekkora az a legkisebb tömörítési szint, amivel tömörítve fel tudja tölteni Jane Austen művét (**Moodle 2**)?

4.3. Önálló feladat

Próbálja meg a `parrot.png` képfájlt tömöríteni az előző feladatokban kiegészített tömörítő program segítségével. Mit tapasztal? Mekkora a tömörítési arány százalékban a legerősebb tömörítés esetén, 2 tizedesjegy pontossággal (**Moodle 3a**)?

Most veszteséges képtömörítést alkalmazunk ugyanerre a fájlra. A kódrészletben az eredeti kép beolvasása mentse el (új név alatt!) újra a képet, de `.jpg` formátumban, és adja meg a JPEG tömörítés paraméterét úgy, hogy a legjobb minőségű képet kapjuk. Figyelmesen olvassa el a megjegyzéseket a kódban!

Ezután egészítse ki a `psnr_todo.py` skeletont, és számoljon PSNR-t a két kép között és az eredményt adja be a moodle-ba 2 tizedesjegy pontossággal (**Moodle 3b**). Egy kis segítség: itt.

4.4. Önálló feladat

A feladatban szín tér (color space) konverziókat kell végrehajtani; a bementi kép ismét a `parrot.png`. Egészítse ki a kódot a hiányzó helyeken. Először keresse meg a helyes paramétert, majd konvertáljon vele. Figyelem, az opencv nem RGB-ként, hanem BGR-ként értelmezi a képeket! Csökkentse a Cb és Cr szinteket 32 faktorral, azaz így 8 bit helyett 3 biten fognak elhelyezkedni az értékek. Majd konvertálja vissza a képet BGR-be és figyelje meg, hogy mennyivel kevesebb szín található a képen. Ezután számoljon PSNR-t az eredeti és a leskálázott kép között. Az eredményt adja be moodle-ba, 2 tizedesjegy pontossággal (**Moodle 4**).