

Gospodarno kodiranje po Huffmanovem postopku

Poročilo projektne naloge

Jernej Sabadin



Mentorja: izr. prof. dr. Simon Dobrišek, as. dr. Klemen Grm

Predmet: Informacija in Kodi

Datum: 10. Maj 2023

Contents

| | | |
|-----|---|---|
| 1 | Naloge | 1 |
| 2 | Uvod | 1 |
| 3 | Huffmanov kod | 1 |
| 3.1 | Implementacija v okolju Python | 2 |
| 4 | Kodiranje datotek | 3 |
| 4.1 | Uspešnost koda | 5 |
| 5 | Dekodiranje .huf datotek | 5 |
| 6 | Vrednotenje rezultatov na izbranih datotekah | 5 |
| 6.1 | Smiselnost rezultatov Huffmanovega koda | 7 |
| 6.2 | Vpliv dolžine datoteke na kompresijska razmerja | 8 |
| 7 | Zaključek | 8 |
| 8 | Dodatno | 8 |
| 9 | Reference | 9 |

Ključne besede: Huffmanov kod, gospodarno kodiranje, Dekodiranje

1 Naloge

- Kodiranje in dekodiranje s Huffmanovim kodom
- Vrednotenje uspešnosti gospodarnega kodiranja

2 Uvod

Huffmanov kod je algoritem za stiskanje podatkov, ki je bil razvit leta 1952 s strani Davida A. Huffmana, ameriškega matematika in računalniškega znanstvenika. Huffmanov kod je učinkovit način za zmanjšanje velikosti podatkov z uporabo brezizgubne kompresije, kar pomeni, da je mogoče originalne podatke obnoviti brez izgube informacij.

Huffmanov kod temelji na ideji, da se pogostejsi simboli, kot so črke ali številke, zapišejo z manjšim številom bitov, medtem ko se redkejši simboli zapišejo z več biti. To se doseže z uporabo spremenljive dolžine kodiranja, kjer je dolžina koda za vsak simbol tesno povezana s frekvenco njegovega pojavljanja v izvorni datoteki.

3 Huffmanov kod

V nadaljevanju predstavimo Huffmanov algoritem za določanje kodnih zamenjav. Algoritem v grobem sledi spodnjim točkam:

- Izračun verjetnosti simbolov: Analiza nabora podatkov in določitev verjetnosti pojavljanja posameznih simbolov.
- Ustvarjanje vozlišč: Za vsak unikaten znak se ustvari vozlišče. To vozlišče vsebuje znak in njegovo verjetnost.
- Ustvarjanje prednostne čakalne vrste: vsa vozlišča se postavi v prednostno čakalno vrsto, razvrščeno po verjetnostih znakov.
- Grajenje Huffmanovega drevesa: Odstrani se dve vozlišči z najmanjšimi verjetnostmi iz prednostne čakalne vrste. Ustvari se novo vozlišče, ki ima ti dve vozlišči kot predhodnika in je njegova verjetnost vsota njunih verjetnosti. To novo vozlišče se vstavi nazaj v prednostno čakalno vrsto. Ta postopek se ponavlja, dokler ne ostane samo eno vozlišče. To zadnje vozlišče je koren Huffmanovega drevesa.
- Generiranje Huffmanovih kodnih zamenjav: Začne se pri korenu in se dodaja '0' za vsak rob, ki vodi do levega "otroka", in '1' za vsak rob, ki vodi do desnega "otroka". Ko se doseže listno vozlišče (vozlišče z znakom), pot od korena do lista pod Huffmanovo kodo za ta znak.

Natančneje postopek opišemo spodaj. Huffmanov algoritmom omogoča sestavljanje gospodarnega trenutnega koda za podani množici $A = \{x_1, x_2, \dots, x_a\}$ (množica znakov), $B = \{0, 1\}$ ($a > 2$) in podano porazdelitev verjetnosti p_1, p_2, \dots, p_a ($0 \leq p_i, \sum_{i=1}^a p_i = 1$).

- **1. korak**

- a) postavi $A_0 \leftarrow A$,
- b) znake v A_0 uredi tako, da je $p_a \leq \dots \leq p_2 \leq p_1$,
- c) znaku $x_{a-1} \in A_0$ pripisi znak $0 \in B$, znaku $x_a \in A_0$ pa znak $1 \in B$

- **2. korak**

za $j \leftarrow 1$ do $a - 2$ naredi:

- a) sestavi množico A_j tako, da združi zadnjih dva znaka množice A_{j-1} v en znak in mu prepiši verjetnost, ki jo dobimo kot vsoto verjetnosti združenih znakov. Množica A_j ima $a_j = a - j$ znakov,
- b) množico A_j uredi tako, da je $p_{a_j} \leq \dots \leq p_2 \leq p_1$,
- c) znaku $x_{a_j-1} \in A_j$ pripisi znak $0 \in B$, znaku $x_{a_j} \in A_j$ pa znak $1 \in B$

Konec-za

- **3. korak** Kodno zamenjavo za znak abecede vira x_i sestavimo tako, da vse znake abecede B , ki so se pojavljali z indeksom i , jemljemo po vrsti od konca proti začetku postopka.

Konec postopka

3.1 Implementacija v okolju Python

Izračun kodnih zamenjav v okolju python izvedemo z razredom Huffman.

```
class Huffman:
    def __init__(self):
        pass

    @staticmethod
    def huffman_algorithm(dictionary):
        # Run the Huffman algorithm
        priority_queue = Huffman.dict_to_priority_queue(dictionary)
        huffman_tree = Huffman.build_tree(priority_queue)
        codes = Huffman.build_code_dictionary(huffman_tree)

        return codes

    @staticmethod
    def dict_to_priority_queue(dct):
        # Convert the given dictionary to a priority queue
        queue = [(weight, [key, ""]) for key, weight in dct.items()]
        heapq.heapify(queue) # Sort by probability
        return queue

    @staticmethod
    def build_tree(queue):
        while len(queue) > 1:
            lo = heapq.heappop(queue) # Poping the element with lowest probability
            hi = heapq.heappop(queue) # Poping the element with highest probability
            for pair in lo[1:]:
                pair[1] = '1' + pair[1]
            for pair in hi[1:]:
                pair[1] = '0' + pair[1]
            # Combining signs with lowest probability, and add codes.
            # [lo[0] + hi[0]] assigns new probability , + lo[1:] + hi[1:] combines 2 signs
            heapq.heappush(queue, [lo[0] + hi[0]] + lo[1:] + hi[1:])
        return queue[0]

    @staticmethod
    def build_code_dictionary(tree):
        codes = defaultdict(str)
        for pair in tree[1:]:
            char, code = pair
            codes[char] = code
        return dict(codes)
```

Fig. 1: Huffmanov algoritem

Algoritem preizkusimo na enostavnem primeru, kjer imamo podano verjetnostno porazdelitev znakov.

```
def main():
    ##### NALOGA 1 #####
    print("##### NALOGA 1 #####\n")
    symbols_prob = {
        's1': 0.25,
        's2': 0.20,
        's3': 0.15,
        's4': 0.10,
        's5': 0.08,
        's6': 0.07,
        's7': 0.06,
        's8': 0.05,
        's9': 0.04,
    }
    huffman = Huffman()
    huffman_codes = huffman.huffman_algorithm(symbols_prob)
    print(huffman_codes)
```

Fig. 2: Enostaven primer za določitev kodnih zamenjav

Spodaj prikažemo kako se množice A_j z večanjem j skrajšujejo (iz vrstice v vrstico), sproti pa se gradijo kodne zamenjave prvotnih znakov z združevanjem vozlišč in dodajanjem znakov iz množice B . Sicer je slika slabo pregledna vendar je moč opaziti, da se iz vrstice v vrstico znaka z najmanjšima verjetnostima združita v skupen znak oziroma niz. Poleg tega pa se znakom doda vrednost 0 ali 1. Tako med tem ko združujemo znake tudi sestavljamo slovar kodnih zamenjav.

```
Initial Queue: [[0.04, ['s9', '']], [0.05, ['s4', '']], [0.06, ['s7', '']], [0.08, ['s6', '']], [0.08, ['s5', '']], [0.09, ['s3', '']], [0.12, ['s2', '']], [0.25, ['s1', '']]]
Lowest Probability: 0.04, Highest Probability: 0.05
Combined Array: [[s9, '1'], [s4, '0']]
Updated Queue: [[0.04, ['s9', '']], [0.05, ['s4', '0']], [0.09, ['s6', '1'], [s8, '01']], [0.25, ['s1', '1'], [s2, '02'], [s3, '03'], [s5, '05'], [s7, '07']]]
```

```
Lowest Probability: 0.04, Index Probability: 0.07
Combined Array: [[s9, '1'], [s4, '0'], [s6, '01'], [s8, '02']]
Updated Queue: [[0.04, ['s9', '']], [0.05, ['s4', '0'], [s6, '01'], [s8, '02']], [0.09, ['s5', '1'], [s7, '03'], [s3, '04'], [s1, '05'], [s2, '06'], [s0, '07']]]
```

```
Lowest Probability: 0.04, Index Probability: 0.08
Combined Array: [[s9, '1'], [s4, '0'], [s6, '01'], [s8, '02'], [s5, '05']]
Updated Queue: [[0.04, ['s9', '']], [0.05, ['s4', '0'], [s6, '01'], [s8, '02'], [s5, '05']], [0.09, ['s3', '1'], [s7, '03'], [s2, '04'], [s0, '05'], [s1, '06'], [s0, '07']]]
```

```
Lowest Probability: 0.04, Index Probability: 0.09
Combined Array: [[s9, '1'], [s4, '0'], [s6, '01'], [s8, '02'], [s5, '05'], [s3, '03']]
Updated Queue: [[0.04, ['s9', '']], [0.05, ['s4', '0'], [s6, '01'], [s8, '02'], [s5, '05'], [s3, '03']], [0.09, ['s2', '1'], [s7, '02'], [s0, '04'], [s1, '05'], [s0, '06'], [s0, '07']]]
```

```
Lowest Probability: 0.04, Index Probability: 0.10
Combined Array: [[s9, '1'], [s4, '0'], [s6, '01'], [s8, '02'], [s5, '05'], [s3, '03'], [s1, '01']]
Updated Queue: [[0.04, ['s9', '']], [0.05, ['s4', '0'], [s6, '01'], [s8, '02'], [s5, '05'], [s3, '03'], [s1, '01']], [0.09, ['s2', '1'], [s7, '02'], [s0, '04'], [s1, '05'], [s0, '06'], [s0, '07'], [s0, '08']]]
```

```
Lowest Probability: 0.04, Index Probability: 0.11
Combined Array: [[s9, '1'], [s4, '0'], [s6, '01'], [s8, '02'], [s5, '05'], [s3, '03'], [s1, '01'], [s0, '00']]
Updated Queue: [[0.04, ['s9', '']], [0.05, ['s4', '0'], [s6, '01'], [s8, '02'], [s5, '05'], [s3, '03'], [s1, '01'], [s0, '00']], [0.09, ['s2', '1'], [s7, '02'], [s0, '04'], [s1, '05'], [s0, '06'], [s0, '07'], [s0, '08'], [s0, '09']]]
```

Fig. 3: Grajenje huffmanovega slovarja kodnih zamenjav

Rešitev, ki jo dobimo je pričakovana. S tem enostavnim primerom se prepričamo, da algoritem deluje pravilno in ga lahko uporabimo za kodiranje binarnih datotek

```
('s2': '11', 's4': '101', 's7': '1001', 's6': '1000', 's1': '01', 's3': '001', 's5': '0001', 's9': '00000', 's8': '00000')
```

Fig. 4: Kodne zamenjave za enostaven primer

4 Kodiranje datotek

V nadaljevanju uporabimo funkcijo za določanje kodnih zamenjav tako, da binarno datoteko zakodiramo po Huffmanovem postopku. Pri tem za "zname" obravnavamo kar vse možne 8-bitne zluge, ki lahko sestavljajo poljubne binarne datoteke.

Postopek kodiranja datotek podamo spodaj:

- 1. Določitev frekvenc pojavljanja posameznih simbolov in verjetnosti povezanih s temi frekven- cami
 - 2. Določitev kodnih zamenjav
 - 3. Zamenjava znakov v originalni datoteki s pri- padajočimi kodami iz 2. koraka
 - 4. Dodajanje $N \in \mathbb{N}$, $0 \leq N \leq 7$ ničel na konec kodiranega sporočila, da je dolžina v bitih deljiva z osem.
 - 5. Zapis glave v novo binarno datoteko. Vsebina glave sestavlja:
 - slovar v katerem so zapisani simboli in nji- hove kodne zamenjave
 - dolžina kodiranega sporočila v bitih; saj smo v primeru, da dolžina kodirane dototeke ni deljiva z osem, na konec dodali potrebno število ničel, da temu ne bo tako.
 - 6. Zapis kodiranih podatkov v novo binarno da- totoko

Glavo datoteke skupaj z odsekom kodiranega sporočila prikažemo spodaj.

Fig. 5: Glava kodirane datoteke skupaj z odsekom kodiranega sporočila

Omenjen postopek nadgradimo tako, da lahko poleg posameznih bajtov vhodne datoteke kodiramo tudi n-terice torej pare oz. trojice bajtov itd. Postopek je v glavnem zelo podoben zgorjnemu. Spremeni se pogled na datoteko z vidika, da obravnavamo pare, trojice itd. bajtov kot posamezne znake. S tem dosežemo povečanje učinkovitosti stiskanja datotek (poglavje 4.1).

Dodatna sprememba v algoritmu je dodajanje od 1 do $n - 1$ "lažnih" bajtov na konec datoteke, v primeru, da dolžina datoteke v bajtih ni deljiva z n . n predstavlja dolžino obravnavanih bajtov (1-terice,2-terice,...,n-terice bajtov), ki jih kodiramo. Dodani bajti so v literaturi poimenovani kot "Dummy". Mi si zanje izberemo znak "NULL" ($b' \backslash x00'$). To je znak, ki ga običajno ne najdemo v besedilnih datotekah.

Spodaj podamo kodo za določanje kodnih zamenjav

```

class Huffman:
    def __init__(self):
        pass

    @staticmethod
    def huffman_algorithm(dictionary):
        # Run the Huffman algorithm
        priority_queue = Huffman.dict_to_priority_queue(dictionary)
        huffman_tree = Huffman.build_tree(priority_queue)
        codes = Huffman.build_code_dictionary(huffman_tree)

        return codes

    @staticmethod
    def dict_to_priority_queue(dct):
        # Convert the given dictionary to a priority queue
        queue = [[weight, [key, ""]] for key, weight in dct.items()]
        heapq.heapify(queue) # Sort by probability
        return queue

    @staticmethod
    def build_tree(queue):
        while len(queue) > 1:
            lo = heapq.heappop(queue) # Poping the element with lowest probability
            hi = heapq.heappop(queue) # Poping the element with highest probability
            for pair in lo[1:]:
                pair[1] = '1' + pair[1]
            for pair in hi[1:]:
                pair[1] = '0' + pair[1]
            # Combining signs with lowest probability, and add codes.
            [lo[0] + hi[0]] assigns new probability , + lo[1:] + hi[1:] combines 2 signs
            heapq.heappush(queue, [lo[0] + hi[0] + lo[1:] + hi[1:]])
        return queue[0]

    @staticmethod
    def build_code_dictionary(tree):
        codes = defaultdict(str)
        for pair in tree[1:]:
            char, code = pair
            codes[char] = code
        return dict(codes)

    @staticmethod
    def calculate_frequencies(filepath, tuple_size):
        # Calculate the occurrences of bytes in a file.
        # Returns the dictionary
        freqs = defaultdict(int)
        N_padded_signs = 0
        with open(filepath, 'rb') as file:
            byte = file.read(tuple_size)
            while byte:
                # Check if the padding with dummy byte is needed
                if (len(byte) != tuple_size) & (len(byte) != 0):
                    # Dummy char is NULL ('\x00'), which is less likely to occur in text, audio or in images
                    N_padded_signs = (tuple_size - len(byte))
                    byte += (N_padded_signs) * (b'\x00')

                freqs[byte] += 1
                byte = file.read(tuple_size)

        return freqs,N_padded_signs

    @staticmethod
    def calculate_probabilities(freqs):
        # Calculates the probabilities of bytes in a file.
        # Returns the dictionary
        total = sum(freqs.values())
        probabilities = {key: freq / total for key, freq in freqs.items()}
        return probabilities

    @staticmethod
    def to_bitarray(bit_string):
        padding_needed = (8 - len(bit_string) % 8) % 8
        padded_bit_string = bit_string + '0' * padding_needed
        return padded_bit_string, padding_needed

```

Fig. 6: Določanje kodnih zamenjav

Funkcija za kodiranje datotek pa je naslednja:

```

@statimethod
def encode_file(filename_input, filename_output, tuple_size=1):
    freqs, N_padded_signs = Huffman.calculate_frequencies(filename_input, tuple_size)
    probabilities = Huffman.calculate_probabilities(freqs)
    huffman_codes = Huffman.huffman_algorithm(probabilities)

    # Encoding the huffman table with 1. order huffmans code.
    huffman_codes_bytes = str(huffman_codes).encode('utf-8')
    file = io.BytesIO(huffman_codes_bytes)
    freqs_, _ = Huffman.calculate_frequencies(file, 1)
    probabilities_ = Huffman.calculate_probabilities(freqs_)
    huffman_codes_of_huffman_codes = Huffman.huffman_algorithm(probabilities_)

    # Coding file with respect to the coding table
    coded_file_ = ''

    # Create a BytesIO object from the Huffman codes
    file = io.BytesIO(str(huffman_codes).encode('utf-8'))

    # Read from the BytesIO object as if it were a file
    byte = file.read(1)
    while byte:
        coded_file_ += huffman_codes_of_huffman_codes[byte]
        byte = file.read(1)

    compressed_huffman_codes, padding_ = Huffman.to_bytearray(coded_file_)

    # Compute the length of the compressed file in bits
    compressed_file_length_ = len(compressed_huffman_codes) - padding_
    #End of encoding of Huffman code table with 1. order huffman algorithm

    # Calculation of efficiency
    entropy = 0
    average_code_length = 0
    for key, p in probabilities.items():
        entropy += -p*np.log2(p) if p != 0 else 0
        average_code_length += p*len(huffman_codes[key])
    efficiency = entropy / average_code_length

    # Coding file with respect to the original huffman coding table
    coded_file = ''
    with open(filename_input, 'rb') as file:
        byte = file.read(tuple_size)
        while byte:
            # Padding with dummy NULL char if needed
            if (len(byte) != tuple_size) & (len(byte) != 0):
                byte += (N_padded_signs) * (b'\x00')
            coded_file += huffman_codes[byte]
            byte = file.read(tuple_size)

    compressed_data, padding = Huffman.to_bytearray(coded_file)

    # Compute the length of the compressed file in bits
    compressed_file_length = len(compressed_data) - padding

    with open(filename_output, 'wb') as file:
        # Concatenate all data into one bytes object
        file.write(b'CT:' + \
                  str(huffman_codes_of_huffman_codes).encode('utf-8') + \
                  b'\nNUMBER_OF_CODE_TABLE:' + \
                  str(compressed_file_length_).encode('utf-8') + \
                  b'\nNUMBER_OF_PADDED_BITS:' + \
                  str(padding_).encode('utf-8') + \
                  b'\nENCODED_CODE_TABLE:' + \
                  bitarray(compressed_huffman_codes) + \
                  b'\nNUMBER_OF_EFFECTIVE_BITS:' + \
                  str(compressed_file_length).encode('utf-8') + \
                  b'\nNUMBER_OF_PADDED_SIGNS:' + \
                  str(N_padded_signs).encode('utf-8') + \
                  b'\nENCODED_DATA:' + \
                  bitarray(compressed_data))

    return efficiency

```

Fig. 7: Funkcija za kodiranje datotek

Opaziti je mogoče, da smo originalni slovar kodnih zamenjav še enkrat kodirali s huffmanovim algoritmom prvega reda. Razlog za to opišemo spodaj.

Ko se velikost n poveča, se bo povečalo tudi število edinstvenih n-teric v vhodnih podatkih. To povzroči, da postane Huffmanova tabela kodnih zamenjav večja in bolj zapletena. To pa povzroči tudi večjo količino informacij v glavi, ki jih je treba shraniti poleg stisnjениh podatkov, kar prispeva k večji velikosti datoteke.

Na spodnjem primeru datoteke "besedilo.txt" prikažemo kaj se zgodi, ko dolžina teric bajtov narašča, slovarja kodnih zamenjav pa ne kodiramo s huffmanovim kodom 1. reda.

```
1-terica: Kodirani podatki brez kodne tabele cca: 584KB; Huffmannov slovar kodnih zamejnav.: 2KB  
Upresnosten koda za terice dolzine 1 znaza 99.112KB  
2-terica: Kodirani podatki brez kode tabele cca: 508KB; Huffmannov slovar kodnih zamejnav.: 32KB  
Upresnosten koda za terice dolzine 2 znaza 99.680KB  
3-terica: Kodirani podatki brez kode tabele cca: 460KB; Huffmannov slovar kodnih zamejnav.: 228KB  
Upresnosten koda za terice dolzine 3 znaza 99.716KB  
4-terica: Kodirani podatki brez kode tabele cca: 470KB; Huffmannov slovar kodnih zamejnav.: 895KB  
Upresnosten koda za terice dolzine 4 znaza 99.794KB
```

Fig. 8: Približne dolžine kodiranih sporočil skupaj z dolzinami glav (Preizkušanje na datoteki "besedilo.txt").

Opazimo, da lahko postane glava datoteke celo večja kakor samo kodirano sporočilo. Iz tega sledi, da vsebuje glava kar nekaj redundancy, ki jo lahko odpravimo s kodiranjem slovarja kodnih zamenjav s huffmanovim kodom 1. reda.

Novo glavo datoteke, ki jo pridobimo z omenjenim postopkom prikažemo spodaj.

Fig. 9: Glava datoteke

Struktura glave, ki jo pridobimo s kodo na sliki 7, je naslednja:

- 1. (**CT:**) Slovar kodnih zamenjav pridobljen s Huffmanovim kodom 1. reda na originalnem huffmanovem slovarju kodnih zamenjav
 - 2. (**NUM_BITS_CODE_TABLE:**) Dolžina kodiranega huffmanovega originalnega slovarja kodnih zamenjav v bitih
 - 3. (**NUM_PADDED_BITS:**) Število dodanih ničelnih bitov na konec kodiranega originalnega huffmanovega slovarja kodnih zamenjav v primeru, da dolžina kodiranega sporočila v bitih ni deljiva z 8.
 - 4. (**ENCODED_CODE_TABLE:**) S huffmanovim kodom 1. reda kodirana originalna kodna tabela
 - 5. (**NUMBER_OF_EFFECTIVE_BITS:**) Dolžina kodirane datoteke v bitih
 - 6. (**NUM_PADDED_SIGNS:**) Število dodanih "lažnih" ("dummy") znakov na konec datoteke, v primeru ko le ta v bytih ni deljiva z n .

Po glavi pa sledi kodirano sporočilo (**CODED DATA:**)

4.1 Uspešnost koda

Poleg kodiranja nam funkcija `encode_file()` izračuna tudi uspešnost gospodarnega koda [1] (N. Pavešič)

$$\eta = H/\bar{n} \quad (1)$$

kjer je \bar{n} povprečno število kodnih znakov na en znak iz abecede vira brez spomina

$$\bar{n} = \frac{\bar{n}_r}{r} \quad (2)$$

$$\bar{n}_r = \sum_{(x_1, \dots, x_r) \in A^r} P(x_1, \dots, x_r) n_r \quad (3)$$

kjer je n_r dolžina kodne zamenjave r-terice (bloka) znakov $(x_1, \dots, x_r) \in A_r$, vsota pa teče po vseh tistih $(x_1, \dots, x_r) \in A_r$, za katere je $P(x_1, \dots, x_r) > 0$.

Entropija r-terice znakov $(x_1, \dots, x_r) \in A_r$ je v primeru, da je vir brez spomina, enaka

$$H(X_1, \dots, X_r) = rH, \quad (4)$$

Entropijo r-terice diskretnih naključnih spremenljivk izračunamo kot:

$$H(X_1, X_2, \dots, X_r) = -K \sum_{i_1} \cdots \sum_{i_r} p_{i_1 \dots i_r} \log_d p_{i_1 \dots i_r} \quad (5)$$

kjer so $K > 0$ poljubna konstanta, $d > 1$ osnova logaritma in $p_{i_1 \dots i_r} = P(X_1 = x_{i_1}, X_2 = x_{i_2}, \dots, X_r = x_{i_r})$ r-razsešna porazdelitev verjetnosti.

Entropija (H) je merilo povprečne količine informacij, ki jih vsebuje vsak simbol iz vira. V kontekstu informacijske teorije kvantificira negotovost ali naključnost nabora simbolov in njihove verjetnostne porazdelitve.

Entropija je torej ključna za razumevanje učinkovitosti koda, saj nam pove, koliko informacij v povprečju vsebuje vsak simbol iz vira. Če je entropija visoka, to pomeni, da je veliko negotovosti ali naključnosti v naboru simbolov, kar pomeni, da bomo potrebovali več bitov za njihovo kodiranje.

Uspešnost nam pove kako dobro se naša kodna shema (kodne zamenjave) prilega porazdelitvi verjetnosti vira. Če je uspešnost enaka 1, to pomeni, da je naša kodna shema optimalna, ker se vsak znak kodira z minimalno možno dolžino glede na njegovo verjetnost. H namreč predstavlja minimalno možno povprečno dolžino kode.

5 Dekodiranje .huf datotek

Datoteke dekodiramo tako, da najprej preberemo Huffmanov slovar kodnih zamenjav s katerim dekodiramo kodirano originalno kodno tabelo. S pomočjo dekodirane originalne kodne tabele nato pretvorimo podatke nazaj v izvorne. Pri tem upoštevamo število dodanih "lažnih" bajtov in dolžine kodiranega Huffmanovega slovarja kodnih zamenjav ter kodirane datoteke v bitih.

Koda za dekodiranje je podana spodaj.

```
@statmethod
def decode_file(filename_input, filename_output):
    # rest of the decode file code
    with open(filename_input, 'rb') as file:
        code_table_line = file.readline().decode('utf-8').strip()
        code_table_str = code_table_line.replace(":", "")
        huffman_encode_table = ast.literal_eval(code_table_str.strip())

    # Switch keys and values using a dictionary comprehension
    huffman_decode_table = {value: key for key, value in huffman_encode_table.items()}

    # Getting the number of bits of huffman encoded huffman table
    num_bits_line = file.readline().decode('utf-8').strip()
    num_padded_bits = int(num_bits_line.split(':')[1].strip())

    # Getting the number of padded zeros at the end of huffman encoded huffman table
    num_padded_bits_line = file.readline().decode('utf-8').strip()
    num_padded_bits = int(num_padded_bits_line.split(':')[1].strip())

    file.read(19) # Reading the 'ENCODED_CODE_TABLE'

    # Reading huffman table
    huffman_encoded_table = file.read(int((num_bits + num_padded_bits)/8))
    huffman_encoded_table_in_binary = ''.join(format(b, '08b') for b in huffman_encoded_table)
    if num_padded_bits > 0:
        decoded_data.extend(huffman_encoded_table_in_binary[-1*num_padded_bits:])
    else:
        decoded_data = huffman_encoded_table_in_binary[0:-1*num_padded_bits]
    decoded_data = bytarray(decoded_data)

    checker = ""
    for i in range(num_bits):
        checker += huffman_encoded_table_in_binary[i]
        if checker in huffman_decode_table:
            # Check if we are at the end of a file, where padding kicks in
            if i == num_bits - 1 & (num_padded_bits != 0):
                decoded_data.extend(huffman_decode_table[checker])
            else:
                decoded_data.append(huffman_decode_table[checker])
            checker = ""

    ##### FIX HERE .decode('utf-8').strip()
    byte_array_string = decoded_data.decode('utf-8').strip()
    huffman_decode_table = ast.literal_eval(byte_array_string.strip())

    # Switch Keys and Values using a dictionary comprehension
    huffman_decode_table = {value: key for key, value in huffman_decode_table.items()}

    file.read(26) # length of '\nNUMBER_OF_EFFECTIVE_BITS'

    num_bits_actual_code = int(file.readline().decode('utf-8').strip())
    N_padded_signs = file.readline().decode('utf-8').strip()
    N_padded_signs = int(N_padded_signs.split(':')[1].strip())

    file.read(11) # len of 'CODED_DATA'

    code = file.read()

    # Convert the input bytes to a binary string
    encoded_data = ''.join(format(b, '08b') for b in code)
    decoded_data = bytarray(decoded_data)
    checker = ""
    for i in range(num_bits_actual_code):
        checker += encoded_data[i]
        if checker in huffman_decode_table:
            # Check if we are at the end of a file, where padding kicks in
            if i == num_bits_actual_code - 1 & (N_padded_signs != 0):
                decoded_data.extend(huffman_decode_table[checker][0:-1*N_padded_signs])
            else:
                decoded_data.append(huffman_decode_table[checker])
            checker = ""
        else:
            decoded_data.append(huffman_decode_table[checker])
            checker = ""

    with open(filename_output, 'wb') as binary_file:
        binary_file.write(decoded_data)
```

Fig. 10: Koda za pretvorbo .huf datotek nazaj v originalno obliko

6 Vrednotenje rezultatov na izbranih datotekah

Rezultate vrednotimo na nekaj izbranih datotekah. Uporabimo besedilno datoteko, zazipano besedilno datoteko, 5 začetnih slik zbirke Kodak Dataset, govorni posnetek, posnetek zvokov narave in posnetek zvokov publike ljudi.

Spodaj prikažemo slike iz zbirke Kodak.



Fig. 11: Slike iz zbirke Kodak

Potem pa uporabimo še sliko iz risanke ter sliko papagaja



Fig. 12: Slika Miki miške ('cartoon.bmp') in papagaja ('slika.bmp')

Za preverjanje ali je dekodirnje pravilno uporabimo MD5 (Message-Digest algorithm 5), ki je široko uporabljen kriptografski izvleček (hash funkcija), ki ustvari 128-bitni zgoščen zapis iz danih podatkov. MD5 izvleček se pogosto uporablja za preverjanje integritete datotek, na primer za preverjanje, če je datoteka, ki jo prenašamo, enaka originalni datoteki na drugem mestu.

Funkcija md_5hash izgleda tako

```
def md5_hash(data):
    md5 = hashlib.md5()
    md5.update(data)
    return md5.hexdigest()
```

Fig. 13: Koda za izračun izvlečkov MD5

Koda bo javila napako v primeru, če sta MD5 izvlečka različna.

Z spodnjo kodo izračunamo uspešnosti in kompre-
sija razmerja med kodirano in originalno datoteko.

Fig. 14: Koda za klic kodiranj in dekodiranj

Rezultate uspešnosti in kompresijska razmerja med kodirano in originalno datoteko prikažemo na spodnji tabeli.

| Datoteka | n-Terica | η % | Kompresijsko razmerje |
|---------------------------|----------|-------------|--------------------------|
| besedilo.txt (0.99 MB) | 1-Terica | 99.11 | 0.58 |
| | 2-Terica | 99.68 | 0.52 |
| | 3-Terica | 99.72 | 0.54 |
| | 4-Terica | 99.79 | 0.76 |
| besedilo.zip (427 KB) | 1-Terica | 99.97 | 1.01 |
| | 2-Terica | 99.83 | 2.81 |
| | 3-Terica | 99.72 | 5.70 |
| | 4-Terica | 99.63 | 4.78 |
| kodim01.bmp (1,12 MB) | 1-Terica | 99.68 | 0.92 |
| | 2-Terica | 99.79 | 1.07 |
| | 3-Terica | 99.76 | 0.74 |
| | 4-Terica | 99.79 | 3.23 |
| kodim02.bmp (1,12 MB) | 1-Terica | 99.56 | 0.86 |
| | 2-Terica | 99.74 | 0.92 |
| | 3-Terica | 99.72 | 0.59 |
| | 4-Terica | 99.84 | 1.91 |
| kodim03.bmp (1,12 MB) | 1-Terica | 99.53 | 0.94 |
| | 2-Terica | 99.78 | 1.19 |
| | 3-Terica | 99.76 | 0.93 |
| | 4-Terica | 99.84 | 2.23 |
| kodim04.bmp (1,12 MB) | 1-Terica | 99.64 | 0.94 |
| | 2-Terica | 99.80 | 1.20 |
| | 3-Terica | 99.78 | 0.93 |
| | 4-Terica | 99.80 | 3.13 |
| kodim05.bmp (1,12 MB) | 1-Terica | 99.76 | 0.94 |
| | 2-Terica | 99.79 | 1.28 |
| | 3-Terica | 99.77 | 1.36 |
| | 4-Terica | 99.76 | 3.65 |
| kodim01.png (719 KB) | 1-Terica | 99.93 | 1.01 |
| | 2-Terica | 99.79 | 2.11 |
| | 3-Terica | 99.85 | 5.74 |
| | 4-Terica | 99.51 | 4.84 |
| cartoon.bmp (530 KB) | 1-Terica | 93.09 | 0.26 |
| | 2-Terica | 96.31 | 0.19 |
| | 3-Terica | 98.00 | 0.26 |
| | 4-Terica | 98.47 | 0.23 |
| slika.bmp (1.8 MB) | 1-Terica | 99.74 | 0.95 |
| | 2-Terica | 99.80 | 1.07 |
| | 3-Terica | 99.83 | 2.28 |
| | 4-Terica | 99.81 | 3.44 |
| govor.wav (181 KB) | 1-Terica | 99.23 | 0.72 |
| | 2-Terica | 99.63 | 1.01 |
| | 3-Terica | 99.85 | 2.52 |
| | 4-Terica | 99.53 | 2.80 |
| narava.wav (622 KB) | 1-Terica | 99.65 | 0.72 |
| | 2-Terica | 99.56 | 0.61 |
| | 3-Terica | 99.77 | 2.40 |
| | 4-Terica | 99.65 | 3.63 |
| publika.wav (648 KB) | 1-Terica | 99.75 | 0.87 |
| | 2-Terica | 99.75 | 0.98 |
| | 3-Terica | 99.76 | 4.11 |
| | 4-Terica | 99.54 | 4.85 |
| govor.mp3 (36,1 KB) | 1-Terica | 99.67 | 1.11 |
| | 2-Terica | 99.65 | 5.91 |
| | 3-Terica | 99.41 | 5.18 |
| | 4-Terica | 99.59 | 4.39 |

Tab. 1: Rezultati huffmanovega koda na različnih datotekah

Na zbirki Kodak Dataset dobimo najboljšo kompresijsko razmerje na sliki "kodim02.bmp" najslabšo pa na sliki "kodim05.bmp".



Fig. 15: Slika kodim02.bmp na levi in slika kodim05.bmp na desni

Učinkovitost Huffmanovega kodiranja je odvisna od porazdelitve simbolov v podatkih. Na splošno velja, da bolj kot so podatki ponavljajoči in predvidljivi, boljše je doseženo kompresijsko razmerje.

Barve na levi sliki so bolj enotne, z velikimi površinami enakih ali podobnih barv (rdeča vrata). To vodi do večje verjetnosti ponavljanja vrednosti slikovnih elementov, zaradi česar je slika primernejša za Huffmanovo kodiranje. Posledično bo kompresijsko razmerje boljše za takšno sliko kakor za sliko na desni.

Slika skupine motoristov z različnimi barvami ima bolj raznolike vrednosti slikovnih elementov, saj so po celotni sliki razporejene različne barve in tekture. To zmanjša ponavljanje vrednosti slikovnih elementov, zaradi česar je Huffmanovo kodiranje manj učinkovito pri stiskanju slike. Posledično je doseženo kompresijsko razmerje za to sliko nižje v primerjavi s sliko z rdečimi vrti.

Izmed slik dosežemo najboljšo kompresijsko razmerje na sliki 'cartoon.bmp', kar je tudi pričakovano, saj vsebuje le nekaj barv izmed katerih so ene bolj, druge manj pogoste. Slika je razmeroma kratke dolžine (530KB) vendar pa učinkovitost kljub temu z večanjem teric narašča, kar pomeni, da so ocene verjetnostnih porazdelitev dovolj natančne saj je v sliki prisotnih le nekaj barv.

Za besedilno datoteko opazimo, da dobimo najboljše rezultate v smislu kompresijskega razmerja za 2-terice. Za 3-terice postane glava datoteke kljub temu, da kodiramo tudi huffmanovo kodno tabelo velika. Vendar so rezultati vseeno boljši kakor za 1-terice. Učinkovitost η se za besedilno datoteko z večanjem teric povečuje.

Ko se dolžina teric poveča, izvorna abeceda postane večja in bolj zapletena, verjetnostna porazdelitev teh teric pa zajame več konteksta iz izvirnih podatkov, povprečno število kodnih znakov na en znak iz abecede vira \bar{n} se zmanjša. Po Shannonovem izreku se gospodarnost koda s podaljševanjem blokov znakov, ki jim prirejamo kodne zamenjave, povečuje.

Na primer, če bi kodirali besedilo v angleščini, bi z uporabo posameznih črk morda ugootovili, da so nekatere črke (kot so 'e' in 't') pogostejše kot druge. Toda z uporabo skupin znakov ali "teric" bi lahko odkrili še več struktur, kot so pogoste kombinacije črk (npr. 'th', 'ing', itd.), ki bi jih lahko še učinkoviteje kodirali. Entropija na znak bi bila za 2-terice nižja kakor za 1-terice. Verjetnost pojava določenega zaporedja simbolov postane bolj predvidljiva in Huffmanov algoritem

to izkoristi.

S tem pristopom se izvorna abeceda sicer poveča (ker imamo več možnih "teric" ali skupin znakov), vendar lahko kodiranje bolje uporabi informacije o verjetnostni porazdelitvi znakov v podatkih, kar vodi do boljše učinkovitosti pri stiskanju podatkov.

Seveda pa se lahko pri prekratkih dolzinah datotek zgodi naslednje. Ko povečujemo dolžino teric, v bistvu segmentiramo naš izvorni podatkovni niz na večje kose. Ker je datoteka premajhna nastopa veliko različnih teric, veliko katerih se v besedilu morda pojavi le nekajkrat, huffmanov algoritem pa tako deluje slabše. Manjša datoteka tako omogoča slabše ocene frekvenc simbolov in razkrivanje manj ponavljajočih se vzorcev, ki bi se lahko učinkovito stisnili. To pomeni, da Huffmanov algoritem, ki je najučinkovitejši, ko so velike razlike v frekvencah simbolov, morda ne bo mogel učinkovito stisniti podatkov z dolgimi tericami, kar vodi do zmanjšane učinkovitosti.

Izmed zvočnih datotek dosežemo največjo kompresijo na datoteki z zvoki narave, saj je sestavljena iz zvokov, izmed katerih so nekateri veliko bolj pogosti.

Najslabšo kompresijo pa dosežemo na posnetku iz nekakšne nogometne tekme, saj vsebuje veliko različnih zvokov in očitno nobeden ne prevladuje.

Kodiranje preizkusimo tudi na brezizgubno kompresiranih .png datotekah. Pri tem se kompresirana datoteka celo rahlo poveča, saj so datoteke že kompresirane in vsebujejo zanemarljivo malo redundancy, ki jo algoritom lahko izkoristi.

Podobno dobimo za govorno mp3 datoteko (kompresirano z izgubno kompresijo). Kompresirana datoteka s huffmanovim kodom se rahlo poveča.

6.1 Smiselnost rezultatov Huffmanovega koda

Pri računanju Huffmanovih kodov za n -terice daljših dolžin opazimo, da učinkovitost z večanjem n ne narašča vedno, kot bi po Shannonovem izreku o gospodarnem kodiranju sprva predvideli. To se namreč zgodi zato, ker so datoteke premajhne in ne moremo zagotoviti zanesljive ocene porazdelitve znakov. Za statistično signifikantno oceno namreč velja naslednje

$$(dolžina\ datoteke) > 10 \times (nabor\ znakov) \quad (6)$$

Zato mora za datoteke veljati

- **Huffman za 1-terice bajtov:**

nabor $2^8 = 256$, min. meja dolžine datoteke za smiselno oceno ≈ 2.5 KB

- **Huffman za 2-terice bajtov:**

nabor $2^{16} = 65536$, min. meja dolžine datoteke za smiselno oceno ≈ 655 KB

- **Huffman za 3-terice bajtov:**

nabor $2^{24} \approx 16.8 \times 10^6$, min. meja dolžine datoteke za smiselno oceno ≈ 168 MB

Sedaj si lahko trend naraščanja učinkovitosti bolje razložimo. Vidimo, da dobimo pri vseh datotekah statistično

signifikantno oceno porazdelitve verjetnosti za 1-terice. Statistično signifikantne ocene porazdelitve verjetnosti za 2-terice pa ne moremo oceniti pri zvočnih datotekah, zazipani besedilni datoteki ter sliki 'cartoon.bmp'. Za preostale datoteke lahko dobimo statistično signifikantno oceno verjetnostnih porazdelitev za 1-terice in 2-terice. Za nobeno datoteko pa nemoremo pridobiti statistično signifikatne ocene verjetnostne porazdelitve za terice dolžin enakih ali daljših kakor 3.

6.2 Vpliv dolžine datoteke na kompresijska razmerja

Za namen tega eksperimenta uporabimo datoteko "besedilo.txt". Opazujemo kako dolžina originalne datoteke vpliva na kompresijska razmerja. Pri tem jemljemo $\frac{1}{5}, \frac{2}{5}, \dots, \frac{5}{5}$ izvirne datoteke.

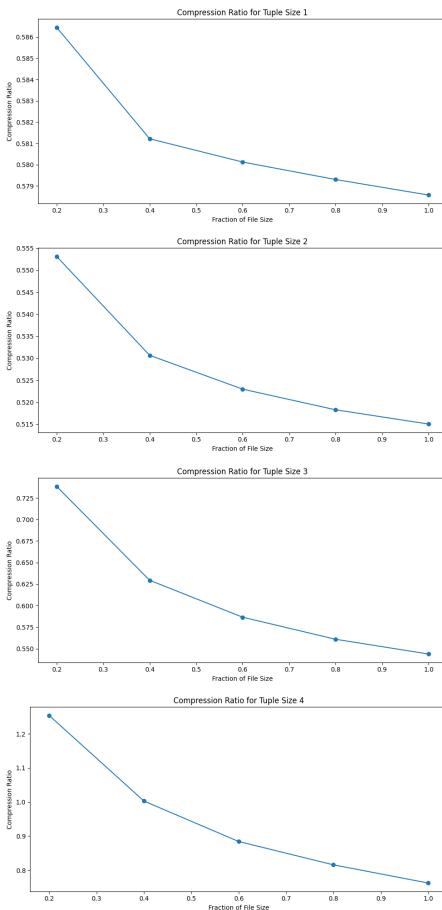


Fig. 16: Kompresijsko razmerje v odvisnosti od dolžine datoteke (celotna datoteka ima približno 1MB)

Očitno je, da se kompresijsko razmerje z večanjem dolžine besedilne datoteke izboljšuje. Večja datoteka omogoča bolše ocene frekvenc simbolov in razkrivanje več ponavljajočih se vzorcev, ki se lahko učinkovito stisnejo. Spremembe v velikosti stisnjениh datotek se z večanjem dolžine originalne datoteke bolj poznajo pri večjih tericah. Npr. za 4-terice pri $\frac{1}{5}$ datoteki ne dosežeme nobene kompresije, še huje, kodirana datoteka je celo večja od izvirne. Pri polni dolžini datoteke pa dosežemo očitno bolše kompresijsko razmerje.

To se zgodi zato, ker večje datoteke omogočajo boljšo oceno verjetnosti simbolov, ki temelji na daljših zaporedjih (tericah), kar je pomembno za vire s spominom, kot je besedilna datoteka. Upoštevanje daljših teric v kodiranju Huffmana omogoča boljše izkoristiti spominske lastnosti vira, kar vodi do boljše kompresije.

Rezultate smo pridobili s spodnjo kodo

```

# Testiranje kako velikost datoteke vpliva na kompresijsko razmerje
# Dodatek za spreminjanje velikosti podatkov
fractions = [0.2, 0.4, 0.6, 0.8, 1.0] # Dolžine Zelenje frakcije dolžine datoteke
with open('datoteka/besedilo.txt', 'rb') as f:
    data_full = f.read() # branje celotne datoteke
full_size = len(data_full) # Dolžina celotne datoteke

# Slovarji za shranjevanje učinkovitosti in razmerja stiskanja
efficiencies = {1: [], 2: [], 3: [], 4: []}
compression_ratios = {1: [], 2: [], 3: [], 4: []}

for fraction in fractions:
    data_size = int(full_size * fraction) # Izračun dolžine podatkov
    data = data_full[:data_size] # Izberi ustrezne kolikino podatkov

    # Tu zapisite podatke v začetno datoteko
    with open('temp.txt', 'wb') as f:
        f.write(data)

    for t in [1, 2, 3, 4]:
        filename_input_encoding = 'temp.txt'
        filename_output_encoding = 'datoteka/generated_datoteke/' + 'temp' + '_encoded' + '-' + str(t) + '.txt'
        filename_input_decoding = 'datoteka/generated_datoteke/' + 'temp' + '_decoded' + '-' + str(t) + '.txt'
        filename_output_decoding = 'filename_output_encoding'

        # Uporabite začasno datoteko namesto pravne datoteko
        Huffman.encode_file(filename_input_encoding, filename_output_encoding, tuple_size=t)
        Huffman.decode_file(filename_input_decoding, filename_output_decoding)

        # Preverjanje velikosti datotek
        file_size_original = os.path.getsize('temp.txt') # Velikost originalne datoteke
        file_size_encoded = os.path.getsize(filename_output_encoding) # Velikost kodirane datoteke
        compression_ratio = file_size_encoded / file_size_original

        # Shranjevanje učinkovitosti in razmerja stiskanja
        efficiencies[t].append(efficiency)
        compression_ratios[t].append(compression_ratio)
os.remove('temp.txt')

# Plotting the data
for t in [1, 2, 3, 4]:
    plt.figure(figsize=(10, 5))
    plt.title(f'Compression Ratio for Tuple Size {t}')
    plt.xlabel('Fraction of File Size')
    plt.ylabel('Compression Ratio')
    plt.plot(fractions, compression_ratios[t], 'o-')
    plt.tight_layout()
    plt.show()

```

Fig. 17: Kompresijsko razmerje v odvisnosti od dolžine datoteke (celotna datoteka ima približno 1MB)

7 Zaključek

Čez poglavja smo se poglobili v študijo Huffmannovega kodiranja, algoritma za stiskanje podatkov, in njegove implementacije v Pythonu. Zasnovali smo in izvedli kodiranje in dekodiranje binarnih datotek ter vrednotili rezultate kompresije.

Spoznali smo, da Huffmannovo kodiranje, čeprav je enostavno za razumevanje in implementacijo, zahteva statistično signifikantno oceno verjetnosti znakov v naboru podatkov za optimalno učinkovitost. Ugotovili smo, kako dolžina in narava podatkovnih nizov vplivata na stopnjo stiskanja.

Pomembno spoznanje je, da se z večanjem teric uspešnost izboljšuje, vendar se z večanjem teric eksponentno podaljšuje tudi časa, ki sta potrebna za kodiranje in dekodiranje sporočil, kar je seveda slabo.

S Huffmanom prvega reda smo kodirali tudi huffmanovo kodno tabelo, kar zmanjša velikost datoteke, vendar spet za ceno časa potrebnega za kodiranje in dekodiranje.

8 Dodatno

V kodi smo zaradi dolgega kodiranja in dekodiranja podali izračun za 1-terice, 2-terice in 3-terice. Za rezultate z 4-tericami se enostavno doda "4" v vrsticah kode 329 351 in 283.

9 Reference

- 1 N. Pavešić, Informacija in Kodi, Založba FE in FRI 2010
- 2 S. Dobrišek: Informacija in Kodi, gradivo za predavanja, FE 2023
- 3 K. Grm: Informacija in Kodi, gradivo za laboratorijske vaje, FE