Zadanie 3 - Huffman compression

April 16, 2021

1 Compression with Huffman Trees

1.1 Paweł Kruczkiewicz

31.03.2021 r.

1.1.1 Treść

Zadanie polega na implementacji dwóch algorytmów kompresji:

1. statycznego algorytmu Huffmana (1 punkt)
2. dynamicznego algorytmu Huffmana (2 punkty)

Dla każdego z algorytmów należy wykonać następujące zadania:

1. Opracować format pliku przechowującego dane.
2. Zaimplementować algorytm kompresji i dekompresji danych dla tego formatu pliku.
3. Zmierzyć współczynnik kompresji (wyrażone w procentach: 1 - plik_skompresowany / plik_nieskompresowany) dla plików tekstowych o rozmiarach: 1kB, 10kB, 100kB, 1MB, dla różnych typów plików: plik tekstowy z portalu Guttenberga, plik źródłowy z Githubu, plik ze znakami losowanymi z rozkładu jednostajnego.
4. Zmierzyć czas kompresji i dekompresji dla plików z punktu 3 dla każdego algorytmu.

Zadanie dla chętnych: Zaimplementować dowolny algorytm ze zmiennym blokiem kompresji, który uzyska lepszy współczynnik kompresji na większości danych wejściowych, niż algorytmy Huffmana (+2 punkty).

1.2 Implementacje

1.2.1 Kod Huffmana statyczny

Tworzenie drzewa

[1]:
```
from bitarray import bitarray
from bitarray.util import ba2int
from queue import Queue
```

[2]:
```
from collections import deque
from math import inf



class InternalNode:
```

```
    def __init__(self, left, right, weight, parent=None, index=None):
        self.weight = weight
        self.left = left
        self.right = right
        self.parent = parent  # only for addaptive huffman tree
        self.index = index


class Leaf:
    def __init__(self, letter, weight, parent=None, index=None):
        self.weight = weight
        self.letter = letter
        self.parent = parent  # only for addaptive huffman tree
        self.index = index


def count_weights(text):   # counting number of occurences of a given character
    result = dict()
    for character in text:
        val = result.get(character, 0)
        result[character] = val + 1
    return result


def smallest_two_elems(deq1, deq2):
    result = []
    while len(result) < 2:
        smallest1 = Leaf("_", inf)
        smallest2 = Leaf("_", inf)
        if len(deq1):
            smallest1 = deq1[0]
        if len(deq2):
            smallest2 = deq2[0]

        if smallest1.weight < smallest2.weight:
            result.append(deq1.popleft())
        else:
            result.append(deq2.popleft())

    return tuple(result)


def static_huffman(letter_count):   # gives the Huffman tree for a given text
    n = len(letter_count)

    leaves = [Leaf(sign, weight) for sign, weight in letter_count.items()]
    leaves.sort(key=lambda x: x.weight)
```

```
    leaves = deque(leaves)  # making deque of leaves

    internal_nodes = deque()  # making empty deque for internal nodes

    for _ in range(n - 1):
        elem1, elem2 = smallest_two_elems(leaves, internal_nodes)
        internal_nodes.append(InternalNode(elem1, elem2, elem1.weight + elem2.
 ↪weight))

    return internal_nodes[-1]  # == root
```

Kodowanie

[3]:
```
#encoding format:
    # 32 bits for number of characters = n,
    # n times 8*(N_BYTES+K_BYTES) bits for character+its frequency
    # encoded text



# we can customise how much data is needed for metadata (this values are␣
 ↪minimum for tests included below)
N_BYTES = 2  # no |of bits for number of characters
K_BYTES = 3  # no of bits for frequency



def is_instance(obj, class_name):
    return obj.__class__.__name__ == class_name



def give_dict(node, result=None, acc=""):   #gives a mapping of characters to␣
 ↪code
    if is_instance(node, "Leaf"):
        result[node.letter] = acc

    if is_instance(node, "InternalNode"):
        if result is None:  result = dict()  # empty result dict declaration

        give_dict(node.left, result, acc + "0")
        give_dict(node.right, result, acc + "1")
        return result



def encode_prefix(letters_count):  # encoding only the meta information
    result = bitarray()

    dict_len_bytes = len(letters_count).to_bytes(length=N_BYTES,␣
 ↪byteorder='big', signed=False)
```

```
    result.frombytes(dict_len_bytes)  # appending the length

    for key, val in letters_count.items():
        result.frombytes(ord(key).to_bytes(length=N_BYTES, byteorder='big',␣
↪signed=False))  # encoding_key
        result.frombytes(val.to_bytes(length=K_BYTES, byteorder='big',␣
↪signed=False))  # encoding length of code

    return result


def encode(text):
    weight_dict = count_weights(text)
    huff_tree_root = static_huffman(weight_dict)
    code_dict = give_dict(huff_tree_root)

    metadata = encode_prefix(weight_dict)
    encoded_text = "".join([code_dict[letter] for letter in text])
    return metadata + bitarray(encoded_text)
```

[4]:
```
example = encode("abracadabra")
print(example)
```

```
bitarray('0000000000000101000000000110000100000000000000000000010100000000011000
10000000000000000000000010000000000111001000000000000000000000001000000000011000
11000000000000000000000001000000000110010000000000000000000000000101111001100011
010111100')
```

Dekodowanie

[5]:
```
def decode(code):
    def split_bitarray():   # here we: 1. decode all informations from metadata;
↪ 2. return the encoded text solely
        n_bit = code[:8*N_BYTES]
        n = ba2int(n_bit)
        encoded_text_pointer = 8*N_BYTES + 8*n*(N_BYTES+K_BYTES)  # index of␣
↪the first bit of the encoded text

        char_count = dict()
        for i in range(8*N_BYTES, encoded_text_pointer, 8*(N_BYTES + K_BYTES)):
            character = chr(ba2int(code[i:i+8*N_BYTES], signed=False) )
            i += 8*N_BYTES
            count = ba2int(code[i:i+8*K_BYTES], signed=False)
            char_count[character] = count

        encoded_text = code[encoded_text_pointer:]
        return n, char_count, encoded_text
```

```
    def decode_text():  # the procedure of decoding code
        result = ''
        curr_node = huff_tree_root
        for bit in encoded_text:
            if is_instance(curr_node, "InternalNode"):
                if bit == 0:
                    curr_node = curr_node.left
                else:
                    curr_node = curr_node.right
                if is_instance(curr_node, "Leaf"):
                    result += curr_node.letter
                    curr_node = huff_tree_root

        return result

    n, char_count, encoded_text = split_bitarray()
    huff_tree_root = static_huffman(char_count)
    return decode_text()
```

[6]:
```
decode(example)
```

[6]: 'abracadabra'

1.2.2 Kod Huffmana dynamiczny

funkcje pomocnicze

[7]:
```
def get_node_code(curr_node):  # we go up the tree till we get parent
    result = bitarray()
    while curr_node is not None:
        dad = curr_node.parent
        if dad is not None and dad.left is curr_node:
            result.append(False)
        elif dad is not None and dad.right is curr_node:
            result.append(True)
        curr_node = dad

    return result[::-1]  # reversing since we go up the tree (normally we go␣
 ↪down)


def update_indexes(root):
    queue = Queue()
    queue.put(root)
    i = 0

    while not queue.empty():
```

```
        curr_node = queue.get()
        curr_node.index = i
        if is_instance(curr_node, "InternalNode"):
            queue.put(curr_node.right)
            queue.put(curr_node.left)
        i += 1

def increment(curr_node, nodes):  # go up the huffman tree, increment every␣
 ↪node and swap if necessary
    def swap_condition(curr, weight_leader):
        return curr.parent is not None and weight_leader. parent is not None␣
 ↪and \
               curr is not weight_leader.parent and curr.parent is not␣
 ↪weight_leader

    def swap(node_a, node_b):
        if node_a.parent is node_b.parent:
            if node_a.parent.left is node_a:
                node_a.parent.left = node_b
                node_a.parent.right = node_a
            else:
                node_a.parent.left = node_a
                node_a.parent.right = node_b
            return
        node_a_parent = node_a.parent
        node_b_parent = node_b.parent
        if node_a is node_a_parent.left:
            node_a_parent.left = node_b
        else:
            node_a_parent.right = node_b
        if node_b is node_b_parent.left:
            node_b_parent.left = node_a
        else:
            node_b_parent.right = node_a
        node_a.parent = node_b_parent
        node_b.parent = node_a_parent
        node_a.index, node_b.index = node_b.index, node_a.index

    while curr_node.parent is not None:
        leaders = nodes[curr_node.weight]
        leaders.sort(key=lambda x: x.index)
        i = 0
        while i < len(leaders) and leaders[i].index < curr_node.index:
            leader = leaders[i]
            if swap_condition(curr_node, leader):
                swap(curr_node, leader)
                break
```

```
            i += 1

        add_weight_and_update_nodes(nodes, curr_node)
        curr_node = curr_node.parent

    add_weight_and_update_nodes(nodes, curr_node)
    update_indexes(curr_node)


def add_weight_and_update_nodes(nodes, curr_node):
    old_weight_buddies = nodes[curr_node.weight]
    if curr_node in old_weight_buddies:
        old_weight_buddies.remove(curr_node)

    curr_node.weight += 1

    list_with_nodes_with_the_same_weight = nodes.setdefault(curr_node.weight,␣
↪[])
    list_with_nodes_with_the_same_weight.append(curr_node)


def copy_leaf(leaf_node):  # make new InternalNode out of a given Leaf
    new_node = InternalNode(None, None, leaf_node.weight, parent=leaf_node.
↪parent)
    if leaf_node.parent is not None:
        if leaf_node.parent.left is leaf_node:
            new_node.parent.left = new_node
        else:
            new_node.parent.right = new_node

    return new_node


def add_new_letter(zero_node, letter, leaves, nodes, root):  # making and␣
↪rearranging the tree
    new_internal = copy_leaf(zero_node)  # making internal node out of the zero␣
↪node and attaching it to parent

    new_leaf = Leaf(letter, 0, parent=new_internal)
    add_weight_and_update_nodes(nodes, new_leaf)
    new_internal.right = new_leaf
    leaves[letter] = new_leaf

    new_internal.left = zero_node
    zero_node.parent = new_internal

    if new_internal.parent is None:
```

```
        root = new_internal

    update_indexes(root)
    increment(new_internal, nodes)
    return new_internal
```

kodowanie

[8]:
```
# nyt means "not yet transmitted" - it's the "zero" node
def adaptive_huffman_encode(text):
    nyt_char = '\uE000'   # special character in UNICODE
    leaves = {nyt_char: Leaf(nyt_char, 0, parent=None, index=0)}
    encoded_text = bitarray()
    root = leaves[nyt_char]

    nodes = {0: [root]}

    for letter in text:
        if letter in leaves:
            curr_node = leaves[letter]
            node_code = get_node_code(curr_node)
            encoded_text += node_code

            increment(curr_node, nodes)
        else:
            zero_node = leaves[nyt_char]

            zero_node_code = get_node_code(zero_node)
            encoded_text += zero_node_code
            letter_bytes = ord(letter).to_bytes(length=N_BYTES,␣
↪byteorder='big', signed=False)
            encoded_text.frombytes(letter_bytes)

            new_internal = add_new_letter(zero_node, letter, leaves, nodes,␣
↪root)
            if new_internal.parent is None:  # new_internal is the factual root
                root = new_internal   # so we need to change it

    return encoded_text
```

[9]:
```
encoded_text = adaptive_huffman_encode('abracadabra')
print(encoded_text)
```

```
bitarray('0000000001100001000000000011000100000000000011100100100000000000110001
1011000000000000110010001101100')
```

dekodowanie

[10]:
```
def adaptive_huffman_decode(code):
    nyt_char = '\uE000'  # special character in UNICODE
    leaves = {nyt_char: Leaf(nyt_char, 0, parent=None)}
    decoded_text = ""
    root = leaves[nyt_char]

    nodes = {0: [root]}
    i = 0
    while i < len(code):
        curr_node = root
        while is_instance(curr_node, "InternalNode"):
            bit = code[i]
            if bit == 0:
                curr_node = curr_node.left
            else:
                curr_node = curr_node.right
            i += 1

        if curr_node.letter != nyt_char:  # the letter was previously in text
            letter = curr_node.letter
            decoded_text += letter
            increment(curr_node, nodes)
        else:                              # the first encounter of the letter␣
↪(curr_node is zero_node)
            letter = chr(ba2int(code[i:i + 8 * N_BYTES], signed=False))
            i += 8 * N_BYTES
            decoded_text += letter

            zero_node = leaves[nyt_char]

            new_internal = add_new_letter(zero_node, letter, leaves, nodes,␣
↪root)
            if new_internal.parent is None:  # new_internal is the factual root
                root = new_internal  # so we need to change it

    return decoded_text
```

[11]:
```
decoded_text = adaptive_huffman_decode(encoded_text)
print(decoded_text)
```

```
abracadabra
```

1.3 Testy

W folderze `testy` znajduja się pliki z tekstem do zakodowania

[12]:
```
from time import time
import os
```

```
def test(tests_dir_name):
    def unit_test(path, encode_function, decode_function, name):
        with open(file_path, "r", encoding="utf8") as file:
            text = file.read()

        start_encode = time()
        code = encode_function(text)
        end_encode = time()

        start_decode = time()
        decode_function(code)
        end_decode = time()

        result_path = os.path.splitext(path)[0] + "_result.txt"
        with open(result_path, "wb") as file:
            code.tofile(file)

        compressed_size = os.path.getsize(result_path)
        uncompressed_size = os.path.getsize(path)

        print(f'{name}')
        print(f'{file_path}:')
        print(f'encoding:  {round(end_encode - start_encode, 4)} [s]')
        print(f'decoding:  {round(end_decode - start_decode, 4)} [s]')
        print(f'compresion rate: {100 - 100*compressed_size/uncompressed_size}␣
↪%')
        print()

        os.remove(result_path)   # clean up after ourselves


    curr_dir = os.getcwd()
    os.chdir(tests_dir_name)
    test_files_list = os.listdir(".")
    try:
        for file_path in sorted(test_files_list, key=os.path.getsize):
            unit_test(file_path, encode, decode, "static huffman")
            unit_test(file_path, adaptive_huffman_encode,␣
↪adaptive_huffman_decode, "adaptive huffman")
    finally:
        os.chdir(curr_dir)
```

[ ]:
```
test("testy")
```

```
static huffman
linux_1KB.txt:
encoding:  0.0007 [s]
decoding:  0.0025 [s]
compresion rate: -1.001251564455572 %

adaptive huffman
linux_1KB.txt:
encoding:  0.321 [s]
decoding:  0.296 [s]
compresion rate: 20.525657071339168 %

static huffman
random_10kB.txt:
encoding:  0.006 [s]
decoding:  0.018 [s]
compresion rate: -4.030000000000001 %

adaptive huffman
random_10kB.txt:
encoding:  28.8668 [s]
decoding:  28.6287 [s]
compresion rate: 20.760000000000005 %

static huffman
linux_10KB.txt:
encoding:  0.002 [s]
decoding:  0.016 [s]
compresion rate: 29.36986843406865 %

adaptive huffman
linux_10KB.txt:
encoding:  5.4178 [s]
decoding:  5.4602 [s]
compresion rate: 31.605500049460872 %

static huffman
The_old_ones_100KB.txt:
encoding:  0.022 [s]
decoding:  0.131 [s]
compresion rate: 43.52047523981107 %

adaptive huffman
The_old_ones_100KB.txt:
encoding:  57.5675 [s]
decoding:  56.5762 [s]
compresion rate: 43.71427180211326 %
```

```
static huffman
Kritik_der_reinen_Vernunft_1MB.txt:
encoding:  0.229 [s]
decoding:  1.4586 [s]
compresion rate: 46.399579992550144 %

adaptive huffman
Kritik_der_reinen_Vernunft_1MB.txt:
encoding:  679.5731 [s]
decoding:  732.7438 [s]
compresion rate: 46.422359726213024 %

static huffman
random_1MB.txt:
encoding:  0.2191 [s]
decoding:  1.518 [s]
compresion rate: 38.25 %
```

[ ]: