Lab3_Piotr_Kica

April 16, 2021

Instrukcje Zadanie polega na implementacji dwóch algorytmów kompresji:

```
-statycznego algorytmu Huffmana (1 punkt)
```

```
-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_niesko

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 współczynn
```

Format pliku

- 8 bitów na początku określających ile jest pustych bitów na końcu
- zakodowany tekst
- puste bity (aby liczba bitów % 8 == 0)

Zapisywanie i wczytywanie, importy, funkcje pomocnicze…

[1]:
```
from bitarray import bitarray
from collections import Counter
from time import time
from bitarray.util import ba2int, int2ba
import os
from random import randrange

def get_compression_ratio(read_file, write_file):
    coded_size = os.path.getsize(write_file)
    original_size = os.path.getsize(read_file)
    return 1-coded_size / original_size

def write_compressed(filename, bitstring):
```

```python
    with open(filename, "wb+") as file:
        bitstring.tofile(file)

def read_compressed(filename):
    bitstring = bitarray()
    with open(f'{filename}', 'rb') as file:
        bitstring.fromfile(file)
    return bitstring

def generate_uniform_dist_file(filename, text_len):
    text = ""
    for i in range(1, text_len + 1):
        text += chr(randrange(48, 122))
        if (i % 50 == 0):
            text += "\n"

    with open(filename, "w+") as file:
        file.write(text)
```

Klasa node (współdzielona przez oba algorytmy)

[2]:
```python
class Node():
    def __init__(self, char=None, weight=0, order=0, left=None, right=None,␣
 ↪parent=None):
        self.left = left
        self.right = right
        self.weight = weight
        self.char = char
        self.order = order
        self.parent = parent

    def __str__(self):
        return f"'{self.char}' : {self.weight}"

    def is_leaf(self):
        if self.left is None and self.right is None:
            return True
        return False
```

0.0.1 Static huffman

[3]:
```python
class StaticHuffman:
    def __init__(self, text):
        self.text = text
        self.root = self.static_huffman()
        self.coded_chars_dict = self.get_coded_chars_dict()
```

```python
def static_huffman(self):
    letter_counts = Counter(self.text)
    nodes = []
    for letter, weight in letter_counts.items():
        nodes.append(Node(char=letter, weight=weight))
    internal_nodes = []
    leafs = sorted(nodes, key=lambda n: n.weight)
    while len(leafs) + len(internal_nodes) > 1:
        lightest_nodes = []
        if len(leafs) >= 2:
            lightest_nodes += leafs[:2]
        elif len(leafs) == 1:
            lightest_nodes.append(leafs[0])
        if len(internal_nodes) >= 2:
            lightest_nodes += internal_nodes[:2]
        elif len(internal_nodes) == 1:
            lightest_nodes.append(internal_nodes[0])

        lightest_nodes.sort(key=lambda n: n.weight)
        element_1, element_2 = lightest_nodes[0], lightest_nodes[1]

        new_internal_node = Node(weight=element_1.weight + element_2.weight)
        new_internal_node.left = element_1
        new_internal_node.right = element_2
        internal_nodes.append(new_internal_node)

        if len(leafs) > 0 and element_1 == leafs[0]:
            leafs.pop(0)
        else:
            internal_nodes.pop(0)
        if len(leafs) > 0 and element_2 == leafs[0]:
            leafs.pop(0)
        else:
            internal_nodes.pop(0)

    return internal_nodes[0]

def get_coded_chars_dict(self):
    def _code_chars(node, code):
        if node.is_leaf():
            coded_chars_dict[node.char] = code
            return
        if node.left:
            _code_chars(node.left, code + bitarray("0"))
        if node.right:
            _code_chars(node.right, code + bitarray("1"))
```

```
        coded_chars_dict = dict()
        _code_chars(self.root.left, bitarray("0"))
        _code_chars(self.root.right, bitarray("1"))

        return coded_chars_dict

    def encode(self):
        coded_text = bitarray()
        for s in self.text:
            coded_text += self.coded_chars_dict[s]

        end_bits = (8 - len(coded_text) % 8)
        coded_text = bitarray(f'{end_bits:08b}') + coded_text + bitarray(end_bits)

        return coded_text

    def decode(self, coded_text):
        decoded_text = ""
        tmp = self.root
        end_bits = ba2int(coded_text[:8])
        coded_text = coded_text[8:-end_bits]
        for char in coded_text:
            if char == False:
                tmp = tmp.left
            else:
                tmp = tmp.right
            if tmp.is_leaf():
                decoded_text += tmp.char
                tmp = self.root
        return decoded_text

def _time_static(read_file, save_file):
    with open(read_file, "r") as file:
        text = file.read()

    static_huffman = StaticHuffman(text)
    start = time()
    encoded = static_huffman.encode()
    end = time()

    compression_time = end-start
    write_compressed(save_file, encoded)
    read_encoded = read_compressed(save_file)

    start = time()
    decoded = static_huffman.decode(read_encoded)
```

```
    end = time()
    decompression_time = end-start
    assert text == decoded
    return compression_time, decompression_time
```

[4]:
```
def timer(read_file, method):
    print(f"Test {read_file}, metoda={method}:")
    save_file = "test_" + read_file
    if method in ["static", "Static"]:
        compression_time, decompression_time = _time_static(read_file,␣
 ↪save_file)
    else:
        compression_time, decompression_time = _time_dynamic(read_file,␣
 ↪save_file)
    print(f"Czas kompresji {method}: {compression_time}s")
    print(f"Czas dekompresji {method}: {decompression_time}s")
    print(f"Współczynnik kompresji {method}: {get_compression_ratio(read_file,␣
 ↪save_file)}%\n")
```

[5]:
```
timer("unix_long_file.c","static")
```

```
Test unix_long_file.c, metoda=static:
Czas kompresji static: 0.10260605812072754s
Czas dekompresji static: 0.7621722221374512s
Współczynnik kompresji static: 0.3235074811601987%
```

0.0.2 Dynamic huffman

Format pliku

- 8 bitów na początku określających ile jest pustych bitów na końcu
- zakodowany tekst
- puste bity (aby liczba bitów % 8 == 0)

[6]:
```
class DynamicHuffman:
    def __init__(self):
        self.order = 100
        NYT = Node(weight=0, order=self.order + 1, char='NYT')
        self.root = NYT
        self.NYT = NYT
        self.leaves_dict = {'NYT': NYT}
        self.weight_dict = dict()
        self.weight_dict[0] = {NYT}
        self.weight_dict[1] = set()

    def create_new_node(self, char):
        NYT = self.NYT
```

```
        left_node = Node(char='NYT', weight=0, order=self.order - 1, parent=NYT)
        right_node = Node(char=char, weight=1, order=self.order, parent=NYT)
        self.order -= 2
        NYT.char = None
        NYT.left = left_node
        NYT.right = right_node
        self.NYT = left_node
        self.weight_dict[0].add(left_node)
        self.weight_dict[1].add(right_node)
        self.leaves_dict[char] = right_node
        self.leaves_dict['NYT'] = left_node
        self.increment_and_switch(NYT)

    def increment_and_switch(self, node):
        while node != self.root:
            node = node.parent
            max_order_node = max(self.weight_dict[node.weight], key=lambda n: n.
↪order)
            if max_order_node != node:
                node.order, max_order_node.order = max_order_node.order, node.
↪order
                if node.parent == max_order_node.parent:
                    if node.parent.left == node:
                        node.parent.right = node
                        node.parent.left = max_order_node
                    else:
                        node.parent.right = max_order_node
                        node.parent.left = node
                else:
                    if node.parent.left == node:
                        node.parent.left = max_order_node
                    else:
                        node.parent.right = max_order_node

                    max_order_node_parent = max_order_node.parent
                    max_order_node.parent = node.parent
                    if max_order_node_parent.left == max_order_node:
                        max_order_node_parent.left = node
                    else:
                        max_order_node_parent.right = node
                    node.parent = max_order_node_parent

            self.weight_dict[node.weight].remove(node)
            node.weight += 1
            if node.weight not in self.weight_dict:
                self.weight_dict[node.weight] = set()
            self.weight_dict[node.weight].add(node)
```

```python
    def get_char_code(self, char):
        node = self.leaves_dict[char]
        char_code = bitarray()
        while node != self.root:
            if node.parent.left == node:
                char_code.append(0)
            else:
                char_code.append(1)
            node = node.parent
        char_code.reverse()
        return char_code

def decode_dynamic(coded_text):
    dynamic_tree = DynamicHuffman()
    decoded_text = ""
    current_node = dynamic_tree.root
    end_bits = ba2int(coded_text[:8])
    coded_text = coded_text[8:-end_bits]
    index = 0
    while index < len(coded_text):
        while not current_node.is_leaf():
            if coded_text[index] == False:
                current_node = current_node.left
            else:
                current_node = current_node.right
            index += 1

        if current_node.char != 'NYT':
            char_decoded = current_node.char
            node = dynamic_tree.leaves_dict[char_decoded]
            dynamic_tree.increment_and_switch(node)
        else:
            char_coded = coded_text[index:index + 8]
            char_decoded = char_coded.tobytes().decode("utf-8")
            dynamic_tree.create_new_node(char_decoded)
            index += 8
        current_node = dynamic_tree.root
        decoded_text += char_decoded

    return decoded_text

def encode_dynamic(text):
    def _encode_char(char):
        if char in tree.leaves_dict:
            coded_char = tree.get_char_code(char)
            node = tree.leaves_dict[char]
```

```
            tree.increment_and_switch(node)
        else:
            char_code = tree.get_char_code('NYT')
            char_code.frombytes(char.encode("utf-8"))
            coded_char = char_code
            tree.create_new_node(char)
        return coded_char

    tree = DynamicHuffman()
    coded_text = bitarray()
    for char in text:
        coded_text += _encode_char(char)

    end_bits = (8 - len(coded_text) % 8)
    coded_text = bitarray(f'{end_bits:08b}') + coded_text + bitarray(end_bits)
    return coded_text

def _time_dynamic(read_file, save_file):
    with open(read_file, "r") as file:
        text = file.read()

    start = time()
    encoded = encode_dynamic(text)
    end = time()
    compression_time = end-start

    write_compressed(save_file, encoded)
    read_encoded = read_compressed(save_file)

    start = time()
    decoded = decode_dynamic(read_encoded)
    end = time()
    decompression_time = end-start
    assert text == decoded
    return compression_time, decompression_time
```

[7]:
```
timer("unix_long_file.c","dynamic")
```

```
Test unix_long_file.c, metoda=dynamic:
Czas kompresji dynamic: 6.398504972457886s
Czas dekompresji dynamic: 6.916891574859619s
Współczynnik kompresji dynamic: 0.24465261701830798%
```

0.0.3 Testy

[8]:
```
timer("1kb.txt","static")
timer("1kb.txt","dynamic")
```

```
Test 1kb.txt, metoda=static:
Czas kompresji static: 0.0s
Czas dekompresji static: 0.0008783340454101562s
Współczynnik kompresji static: 0.42559523809523814%

Test 1kb.txt, metoda=dynamic:
Czas kompresji dynamic: 0.01200413703918457s
Czas dekompresji dynamic: 0.008991003036499023s
Współczynnik kompresji dynamic: 0.28422619047619047%
```

[9]:
```
timer("10kb.txt","static")
timer("10kb.txt","dynamic")
```

```
Test 10kb.txt, metoda=static:
Czas kompresji static: 0.0029976367950439453s
Czas dekompresji static: 0.025757551193237305s
Współczynnik kompresji static: 0.436028659160696%

Test 10kb.txt, metoda=dynamic:
Czas kompresji dynamic: 0.09113383293151855s
Czas dekompresji dynamic: 0.07482624053955078s
Współczynnik kompresji dynamic: 0.32016376663254864%
```

[10]:
```
timer("100kb.txt","static")
timer("100kb.txt","dynamic")
```

```
Test 100kb.txt, metoda=static:
Czas kompresji static: 0.013973236083984375s
Czas dekompresji static: 0.0910348892211914s
Współczynnik kompresji static: 0.43565721941464475%

Test 100kb.txt, metoda=dynamic:
Czas kompresji dynamic: 0.9847598075866699s
Czas dekompresji dynamic: 1.2185966968536377s
Współczynnik kompresji dynamic: 0.32605333412142523%
```

[11]:
```
timer("1mb.txt","static")
timer("1mb.txt","dynamic")
```

```
Test 1mb.txt, metoda=static:
```

```
Czas kompresji static: 0.12251019477844238s
Czas dekompresji static: 0.906287431716919s
Współczynnik kompresji static: 0.4356254736365128%

Test 1mb.txt, metoda=dynamic:
Czas kompresji dynamic: 9.316588640213013s
Czas dekompresji dynamic: 9.04110312461853s
Współczynnik kompresji dynamic: 0.3340348362096579%
```

[12]:
```
timer("guttenberg.txt","static")
timer("guttenberg.txt","dynamic")
```

```
Test guttenberg.txt, metoda=static:
Czas kompresji static: 0.026180744171142578s
Czas dekompresji static: 0.15308427810668945s
Współczynnik kompresji static: 0.4399711166543795%

Test guttenberg.txt, metoda=dynamic:
Czas kompresji dynamic: 1.637868881225586s
Czas dekompresji dynamic: 1.5693025588989258s
Współczynnik kompresji dynamic: 0.32944687383229476%
```

[13]:
```
timer("unix_long_file.c","static")
timer("unix_long_file.c","dynamic")
```

```
Test unix_long_file.c, metoda=static:
Czas kompresji static: 0.07562088966369629s
Czas dekompresji static: 0.6662788391113281s
Współczynnik kompresji static: 0.3235074811601987%

Test unix_long_file.c, metoda=dynamic:
Czas kompresji dynamic: 6.490269184112549s
Czas dekompresji dynamic: 6.0875020027160645s
Współczynnik kompresji dynamic: 0.24465261701830798%
```

[14]:
```
generate_uniform_dist_file("random.txt", 100000)
timer("random.txt","static")
timer("random.txt","dynamic")
```

```
Test random.txt, metoda=static:
Czas kompresji static: 0.02000880241394043s
Czas dekompresji static: 0.11568617820739746s
Współczynnik kompresji static: 0.22967307692307692%

Test random.txt, metoda=dynamic:
```

```
Czas kompresji dynamic: 0.9722821712493896s
Czas dekompresji dynamic: 0.9192969799041748s
Współczynnik kompresji dynamic: 0.2263846153846154%
```

0.0.4 Wnioski

Statyczny huffman wypada lepiej pod względem czasowym i współczynnika kompresji. Jednak nie zawsze da się zastosować go więc trzeba użyć dynamicznego huffmana.

Dla rozkładu jednostajnego współczynniki kompresji są niemal równe, ale dynamiczny wypada 10x gorzej czasowo.