Algorytmy Tekstowe lab4

Norbert Wolniak

Useful functions

In [1]:
```python
import os
from datetime import datetime
```

In [2]:
```python
def str_to_bytes(code):
    zeros = 8 - (len(code) % 8)
    info_byte = format(zeros, "08b")
    b = bytearray()
    b.append(int(info_byte,2))
    for i in range(0, len(code), 8):
        b.append(int(code[i:i+8], 2))
    return bytes(b)

def str_from_bytes(bytes):
    zeros = bytes[0]
    code = ""
    for byte in bytes[1:-1]:
        code += format(byte, "08b")
    code += format(bytes[-1], "08b")[zeros:]
    return code
```

Static Huffman algorithm

In [3]:
```python
from array import array
```

In [4]:
```python
class Node:
    def __init__(self, char, weight, left=None, right=None):
        self.char = char
        self.weight = weight
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.weight < other.weight
```

In [5]:
```python
import queue
class StaticHuffmanTree:
    def __init__(self, file_to_read, file_to_write):
        text = None
        with open(file_to_read, encoding="utf-8", errors="replace") as f:
            text = f.read()
        self.file_to_write = file_to_write
        self.root = self.create_tree(text)
        self.huffman_code = {}
        self._create_huffman_code(self.root)
        self.encode(text)

    def create_tree(self, text):
        pq = queue.PriorityQueue()
        letter_counts = {}
        for char in text:
            if char in letter_counts:
                letter_counts[char] += 1
            else:
                letter_counts[char] = 1

        for char,count in letter_counts.items():
            pq.put(Node(char, count))

        while pq.qsize() > 1:
            left = pq.get()
            right = pq.get()
            pq.put(Node(None, left.weight + right.weight, left, right))

        return pq.get()

    def _create_huffman_code(self, node=None, sequence=""):
        if node is not None and node.left is None and node.right is None:
            self.huffman_code[node.char] = sequence
            return
```

```
            self._create_huffman_code(node.left, sequence + "0")
            self._create_huffman_code(node.right, sequence + "1")

    def encode(self, text):
        encoded_text = ""
        for letter in text:
            encoded_text += self.huffman_code[letter]
        with open(self.file_to_write, "wb") as f:
            f.write(str_to_bytes(encoded_text))

    def decode(self):
        with open(self.file_to_write, "rb") as f:
            encoded_text = str_from_bytes(f.read())
        decoded_text = ""
        i = 0
        text = ""
        while i < len(encoded_text):
            node = self.root
            while node.left is not None and node.right is not None:
                if encoded_text[i] == "0":
                    node = node.left
                else:
                    node = node.right
                i += 1
            decoded_text += node.char

        return decoded_text
```

Adaptive Huffman algorithm

In [6]:

```
class NodeA:
    def __init__(self, char, weight, order, parent=None, left=None, right=None):
        self.char = char
        self.weight = weight
        self.order = order
        self.parent = parent
        self.left = left
        self.right = right


class AdaptiveHuffmanTree:
    def __init__(self, file_to_read, file_to_write):
        with open(file_to_read, encoding="utf-8", errors="replace") as f:
            text = f.read()
        self.alphabet_size = len(set(text))
        self.file_to_write = file_to_write
        self.encode(text)


    def encode(self, text):
        self.root = NodeA("#", 0, 2*self.alphabet_size + 1)
        self.nodes = {self.root.char: self.root}
        f = open(self.file_to_write, "wb")
        encoded_text = ""
        for char in text:
            if char in self.nodes:
                encoded_text += self._code(char, self.root)
                self._increment(self.nodes[char])
            else:
                NYT = self.nodes["#"]
                encoded_text += self._code(NYT.char, self.root) + format(ord(char), "08b")
                NYT.char = None

                new_node = NodeA(char, 1, NYT.order - 1, parent=NYT)
                new_NYT = NodeA("#", 0, NYT.order - 2, parent=NYT)

                self.nodes[char] = new_node

                NYT.left = new_NYT
                NYT.right = new_node

                self.nodes["#"] = new_NYT

                self._increment(self.nodes[char].parent)

        with open(self.file_to_write, "wb") as f:
            f.write(str_to_bytes(encoded_text))


    def decode(self):
        self.root = NodeA("#", 0, 2*self.alphabet_size + 1)
        self.nodes = {self.root.char: self.root}

        with open(self.file_to_write, "rb") as f:
            encoded_text = str_from_bytes(f.read())
        decoded_text = ""
```

```python
        curr_node = self.root
        i = 0
        while i < len(encoded_text):
            if curr_node.left is None and curr_node.right is None:
                if curr_node.char == "#":
                    char = chr(int(encoded_text[i:i+8],2))
                    i += 8

                else:
                    char = curr_node.char

                if char in self.nodes:
                    self._increment(self.nodes[char])
                else:
                    NYT = self.nodes["#"]
                    NYT.char = None

                    new_node = NodeA(char, 1, NYT.order - 1, parent=NYT)
                    new_NYT = NodeA("#", 0, NYT.order - 2, parent=NYT)

                    self.nodes[char] = new_node

                    NYT.left = new_NYT
                    NYT.right = new_node

                    self.nodes["#"] = new_NYT

                    self._increment(self.nodes[char].parent)

                decoded_text += char

                curr_node = self.root
            else:
                if i < len(encoded_text):
                    if encoded_text[i] == "0":
                        curr_node = curr_node.left
                    else:
                        curr_node = curr_node.right
                i += 1

        if curr_node.left is None and curr_node.right is None:
            decoded_text += curr_node.char

        return decoded_text

    def _increment(self, node):
        if node is None:
            return

        if self._is_highest_order(node):
            node.weight += 1
            self._increment(node.parent)
        else:
            tmp = self._get_highest_order(node)
            self._swap(node, self._get_highest_order(node))
            node.weight += 1
            self._increment(node.parent)

    def _is_highest_order(self, node):
        for other in self.nodes.values():
            if other.weight == node.weight and other.order > node.order:
                return False
        return True

    def _get_highest_order(self, node):
        weight = node.weight
        order = node.order
        highest_order_node = None
        for other in self.nodes.values():
            if other.weight == weight and other.order > order:
                highest_order_node = other
                order = highest_order_node.order
        return highest_order_node

    def _swap(self, node, other):
        other_parent = other.parent
        node_parent = node.parent
        node_order = node.order

        if other is other_parent.left:
            other_parent.left = node
        else:
            other_parent.right = node
        node.parent = other_parent
        node.order = other.order

        if node is node_parent.left:
            node_parent.left = other
        else:
```

```
            node_parent.right = other
        other.parent = node_parent
        other.order = node_order

    def _code(self, char, node, sequence=""):
        if node is not None and node.left is None and node.right is None:
            if node.char == char:
                return sequence
            else:
                return ""
        else:
            tmp = self._code(char, node.left, sequence + "0")
            if not tmp:
                tmp = self._code(char, node.right, sequence + "1")
            return tmp
```

Compress ratio function

In [7]:
```
def test_size(algorithm, file_to_read, file_to_write):
    if algorithm.__name__ == "StaticHuffmanTree":
        StaticHuffmanTree(file_to_read, file_to_write).decode()
    elif algorithm.__name__ == "AdaptiveHuffmanTree":
        AdaptiveHuffmanTree(file_to_read, file_to_write).decode()

    size_o = os.path.getsize(file_to_read)
    size_c = os.path.getsize(file_to_write)
    print("Algorithm : {}".format(algorithm.__name__))
    print("File name : {}".format(file_to_read))
    print("Uncompressed file size : {} bytes".format(size_o))
    print("Compressed file size : {} bytes".format(size_c))
    print('Compression ratio : {}\n\n'.format(round(1 - size_c / size_o, 2)))
```

Time function

In [8]:
```
def time_test(algorithm, file_to_read, file_to_write):
    if algorithm.__name__ == "StaticHuffmanTree":
        #Encode
        encode_start = datetime.now()
        tree = StaticHuffmanTree(file_to_read, file_to_write)
        encode_time = (datetime.now() - encode_start).total_seconds()
        #Decode
        decode_start = datetime.now()
        tree.decode()
        decode_time = (datetime.now() - decode_start).total_seconds()
    elif algorithm.__name__ == "AdaptiveHuffmanTree":
        #Encode
        encode_start = datetime.now()
        tree = AdaptiveHuffmanTree(file_to_read, file_to_write)
        encode_time = (datetime.now() - encode_start).total_seconds()
        #Decode
        decode_start = datetime.now()
        tree.decode()
        decode_time = (datetime.now() - decode_start).total_seconds()

    print("Algorithm : {}".format(algorithm.__name__))
    print("Encode time : {} [s]".format(encode_time))
    print("Decode time : {} [s]\n".format(decode_time))

    return encode_time, decode_time
```

In [9]:
```
static_encode_times = []
adaptive_encode_times = []
static_decode_times = []
adaptive_decode_times = []
```

Test poprawności

In [10]:
```
print(StaticHuffmanTree("test.txt", "test_static.txt").decode())
```

```
abracadabra
```

In [11]:
```
print(AdaptiveHuffmanTree("test.txt", "test_static.txt").decode())
```

```
abracadabra
```

Random text files :

1 kB

In [12]:
```
test_size(StaticHuffmanTree, "1KB.txt", "1KB_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "1KB.txt", "1KB_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree, "1KB.txt", "1KB_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "1KB.txt", "1KB_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : 1KB.txt
Uncompressed file size : 1024 bytes
Compressed file size : 607 bytes
Compression ratio : 0.41


Algorithm : AdaptiveHuffmanTree
File name : 1KB.txt
Uncompressed file size : 1024 bytes
Compressed file size : 647 bytes
Compression ratio : 0.37


Algorithm : StaticHuffmanTree
Encode time : 0.001 [s]
Decode time : 0.00503 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 0.03997 [s]
Decode time : 0.030031 [s]
```

10 kB

In [13]:
```
test_size(StaticHuffmanTree, "10KB.txt", "10KB_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "10KB.txt", "10KB_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree, "10KB.txt", "10KB_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "10KB.txt", "10KB_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : 10KB.txt
Uncompressed file size : 10240 bytes
Compressed file size : 6088 bytes
Compression ratio : 0.41


Algorithm : AdaptiveHuffmanTree
File name : 10KB.txt
Uncompressed file size : 10240 bytes
Compressed file size : 6184 bytes
Compression ratio : 0.4


Algorithm : StaticHuffmanTree
Encode time : 0.008003 [s]
Decode time : 0.020996 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 0.348965 [s]
Decode time : 0.292033 [s]
```

100 kB

In [14]:
```
test_size(StaticHuffmanTree, "100KB.txt", "100KB_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "100KB.txt", "100KB_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree, "100KB.txt", "100KB_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "100KB.txt", "100KB_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : 100KB.txt
Uncompressed file size : 102400 bytes
Compressed file size : 60967 bytes
Compression ratio : 0.4


Algorithm : AdaptiveHuffmanTree
File name : 100KB.txt
Uncompressed file size : 102400 bytes
Compressed file size : 61530 bytes
Compression ratio : 0.4


Algorithm : StaticHuffmanTree
Encode time : 0.062002 [s]
Decode time : 0.190144 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 3.384165 [s]
Decode time : 2.450985 [s]
```

Uniform distribution ~ 147KB

In [15]:
```
import numpy as np
s = np.random.uniform(0,255,100000)
s = "".join([chr(int(r)) for r in s])
with open("uniform.txt", "w", encoding="utf-8", errors="surrogateescape") as f:
    f.write(s)
```

In [16]:
```
test_size(StaticHuffmanTree, "uniform.txt", "uniform_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "uniform.txt", "uniform_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree,  "uniform.txt", "uniform_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "uniform.txt", "uniform_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : uniform.txt
Uncompressed file size : 150366 bytes
Compressed file size : 99854 bytes
Compression ratio : 0.34


Algorithm : AdaptiveHuffmanTree
File name : uniform.txt
Uncompressed file size : 150366 bytes
Compressed file size : 100215 bytes
Compression ratio : 0.33


Algorithm : StaticHuffmanTree
Encode time : 0.099992 [s]
Decode time : 0.307996 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 36.838307 [s]
Decode time : 24.795063 [s]
```

Linux 506KB

In [17]:
```
test_size(StaticHuffmanTree, "linux2.txt", "linux2_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "linux2.txt", "linux2_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree,  "linux2.txt", "linux2_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "linux2.txt", "linux2_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : linux2.txt
Uncompressed file size : 517486 bytes
Compressed file size : 352192 bytes
Compression ratio : 0.32


Algorithm : AdaptiveHuffmanTree
File name : linux2.txt
Uncompressed file size : 517486 bytes
Compressed file size : 354418 bytes
Compression ratio : 0.32


Algorithm : StaticHuffmanTree
Encode time : 0.657031 [s]
Decode time : 1.386391 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 55.123501 [s]
Decode time : 28.580729 [s]
```

Linux 672KB

In [18]:
```
test_size(StaticHuffmanTree, "linux1.txt", "linux1_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "linux1.txt", "linux1_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree,  "linux1.txt", "linux1_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "linux1.txt", "linux1_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : linux1.txt
Uncompressed file size : 688096 bytes
Compressed file size : 450918 bytes
Compression ratio : 0.34


Algorithm : AdaptiveHuffmanTree
File name : linux1.txt
Uncompressed file size : 688096 bytes
Compressed file size : 463847 bytes
Compression ratio : 0.33


Algorithm : StaticHuffmanTree
Encode time : 0.912028 [s]
Decode time : 1.876092 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 75.53816 [s]
Decode time : 31.5014 [s]
```

Book 844kB

In [19]:
```
test_size(StaticHuffmanTree, "book.txt", "book_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "book.txt", "book_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree, "book.txt", "book_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "book.txt", "book_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : book.txt
Uncompressed file size : 260308 bytes
Compressed file size : 140117 bytes
Compression ratio : 0.46


Algorithm : AdaptiveHuffmanTree
File name : book.txt
Uncompressed file size : 260308 bytes
Compressed file size : 146828 bytes
Compression ratio : 0.44


Algorithm : StaticHuffmanTree
Encode time : 0.193997 [s]
Decode time : 0.493004 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 25.150145 [s]
Decode time : 9.590814 [s]
```

1 MB

In [20]:
```
test_size(StaticHuffmanTree, "1MB.txt", "1MB_static_compressed.bin")
test_size(AdaptiveHuffmanTree, "1MB.txt", "1MB_adaptive_compressed.bin")

time = time_test(StaticHuffmanTree, "1MB.txt", "1MB_static_compressed.bin")
static_encode_times.append(time[0])
adaptive_encode_times.append(time[1])

time = time_test(AdaptiveHuffmanTree, "1MB.txt", "1MB_adaptive_compressed.bin")
adaptive_encode_times.append(time[0])
adaptive_decode_times.append(time[1])
```

```
Algorithm : StaticHuffmanTree
File name : 1MB.txt
Uncompressed file size : 1048576 bytes
Compressed file size : 624869 bytes
Compression ratio : 0.4


Algorithm : AdaptiveHuffmanTree
File name : 1MB.txt
Uncompressed file size : 1048576 bytes
Compressed file size : 630119 bytes
Compression ratio : 0.4


Algorithm : StaticHuffmanTree
Encode time : 1.357966 [s]
Decode time : 2.590673 [s]

Algorithm : AdaptiveHuffmanTree
Encode time : 32.681079 [s]
Decode time : 23.803335 [s]
```