Raport z laboratorium 3

Filip Nikolow

9 kwietnia 2021

1 Cel laboratorium

Celem laboratorium było zapoznanie się z algorytmami kompresji tekstu na przykładzie statycznego oraz dynamicznego algorytmu Huffmana. Jako zadanie dodatkowe należało zaimplementować dowolny algorytm ze zmiennym blokiem kompresji, który uzyska lepszy stopień kompresji od algorytmów Huffmana

2 Realizacja

Zaimplementowałem oba algorytmy Huffmana: statyczny oraz dynamiczny, a także w ramach zadania dodatkowego zaimplementowałem algorytm Lempela-Ziva-Welcha (LZW).

3 Realizacja poszczególnych poleceń

3.1 Polecenie 1 - format pliku

3.1.1 Statyczne kodowanie Huffmana

Zakodowany plik rozpoczyna się trzema bitami reprezentującymi liczbę zer (0-7) dopisanych na końcu pliku w ramach dopełnienia do pełnego bajtu. Następnie zakodowane jest drzewo Huffmana jako jego przeszukiwanie post-order:

- bit 0 oznacza węzeł wewnętrzny;
- bit 1 oznacza liść zawierający kodowany znak, kolejne 8 bitów to reprezentacja tego znaku w kodzie ASCII

Kodowanie drzewa jest zakończone pojedynczym bitem 0. Łatwo policzyć, że rozmiar reprezentacji liścia to 9 bitów, a węzła wewnętrznego 1 bit. Jak wiadomo, drzewo huffmana ma 2m-1 węzłów (gdzie m to rozmiar alfabetu), a zatem uzwględniając kończący bit zerowy, tak zakodowane drzewo huffmana zajmuje dokładnie 10*m bitów.
Po drzewie następuje ciąg bitów reprezentujących zakodowany tekst.

3.1.2 Dynamiczne kodowanie Huffmana

Podobnie jak przy kodowaniu statycznym, pierwsze 3 bity reprezentują liczbę uzupełniających zer na końcu pliku. Po nich następuje ciąg bitów odpowiadających zakodowanemu dynamicznie tekstowi.

3.1.3 LZW

Ten algorytm nie potrzebuje żadnych dodatkowych metadanych w nagłówku pliku, gdyż pojedyncze bloki kodowania są nie mniejsze niż 8 bitów a więc dekoder jest w stanie samodzielnie odrzucić zera z końca pliku. Cały plik jest więc kodem zwracanym przez encoder.

3.2 Polecenie 2 - implementacja

Tak jak wspomniałem we wstępie, zaimplementowałem trzy algorytmy kompresji i dekompresji - 2 obowiązkowe i jeden dodatkowy. Z ważniejszych szczegółów implementacji algorytmu LZW: jako podstawowy zbiór znaków przyjąłem 256 znaków ASCII, a testy przeprowadziłem z limitem rozmiaru pojedynczego bloku kodującego ustawionym na 16 bitów (w efekcie kodowane bloki zmieniały się od 9 do 16) - można go łatwo zmienić, jest to stała klasy LZW. Początkowo testy przeprowadzałem na maksymalnym bloku równym 12 bitów, jednakże taki słownik był nieco za mały do efektywnego kodowania języka naturalnego - nadal algorytm LZW wygrywał większość testów, lecz ze znacznie mniejszą przewagą w przypadku kompresjii książki.

3.3 Polecenia 3-4

Do testów pobrałem "Krzyżaków" w wersji angielskiej z portalu Guttenberga, oba zasugerowane na ćwiczeniach pliki źródłowe w języku C, oraz wygenerowałem za pomocą polecenia base64 plik z losowymi znakami ascii. Pliki źródłowe połączyłem w jeden, a następnie je oraz książke przepuściłem przez skrypt pythonowy podmieniający znaki unicode na najbardziej zbliżone znaki ASCII - wszystkie zaimplementowane algorytmy dzialaja tylko na znakach ASCII.

4 Benchmarki

Poniżej załączam wszystkie wymagane pomiary. Zmierzone czasy kompresji i dekompresji na wykresach po lewej stronie są wyrażone w sekundach (a zatem im mniej tym lepiej), natomiast współczynniki kompresji na wykresach po stronie prawej są obliczone wzorem:

$$1 - \frac{\text{plik\_skompresowany}}{\text{plik\_nieskompresowany}}$$

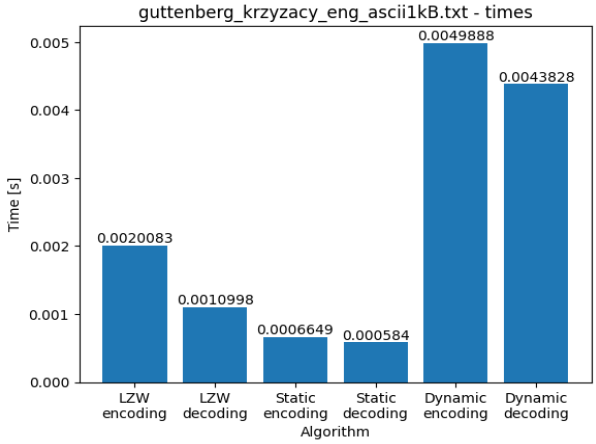
, a zatem im więcej tym lepiej. Porównując oba algorytmy Huffmana, zgodnie z oczekiwaniami algorytm dynamiczny osiąga minimalnie lepsze stopnie kompresji dla małych plików, natomiast przy plikach $\geqslant$10kB różnica się zaciera, a nawet zdarza się iż to kodowanie statyczne minimalnie wygrywa - są to jednak różnice raczej zaniedbywalne. Kodowanie dynamiczne za to zdecydowanie przegrywa jeśli chodzi o czasy kompresji i dekompresji - wina lezy po stronie funkcji transformującej drzewo huffmana podczas zczytywania tekstu, gdyż operacje wyszukiwania węzłów do zamiany oraz same zamiany są kosztownymi operacjami. Przechodząc do algorytmu LZW także wielkich zaskoczeń nie ma: dla małych plików algorytmy Huffmana i LZW mają zbliżone osiągi, natomiast w miarę wzrostu rozmiaru pliku, ujawnia się też przewaga kodowania zmiennymi blokami nad kodowaniem blokami stałymi. W efekcie na testach z książką oraz kodem źródłowym, LZW przegrywa tylko raz: na najmniejszym fragmencie książki. Z kolei po kompresji tekstu z losowymi znakami nie ma co spodziewać się zbyt wiele - oczekiwanym rezultatem jest stopień kompresji bliski 0. Tak się jednak nie dzieje, gdyż algorytmy huffmana osiągają ok. 25% stopień kompresji, a nawet i algorytm LZW dla większych plików jest w stanie lekko zmniejszyć ich rozmiar. Jak się nad tym zastanowić, to także nie jest niczym zaskakującym, gdyż funkcja użyta do generowania znaków, base64, nie używa pełnej palety kodu ASCII, a więc kodowanie Huffmana skraca długość pliku nie poprzez podmienienie kodów najczęściej występujących liter na najkrótsze, a po prostu poprzez usunięcie z kodu niewystępujących znaków. Z kolei algorytm LZW ma minimalny blok kodowania równy 8 (a efektywnie 9), a zatem jest w stanie kompresować tylko powtarzające się wzorce - na małych plikach ilość takich powtórzeń jest na tyle niewielka, że ich kodowanie nie jest w stanie przezwyciężyć dużego narzutu na pojedyncze kody. Przy większych plikach słownik LZW zapełnia się i znowu, ponieważ base64 generuje ograniczony alfabet, w końcu ilość powtarzających się wzorców jest na tyle duża, że możemy zaobserwować kompresję. Podsumowując, algorytm LZW wygrał 7 z 12 testów i w mojej opinii był najprostszym w kompletnej implementacji (wraz z formatem plików itp.). Bez

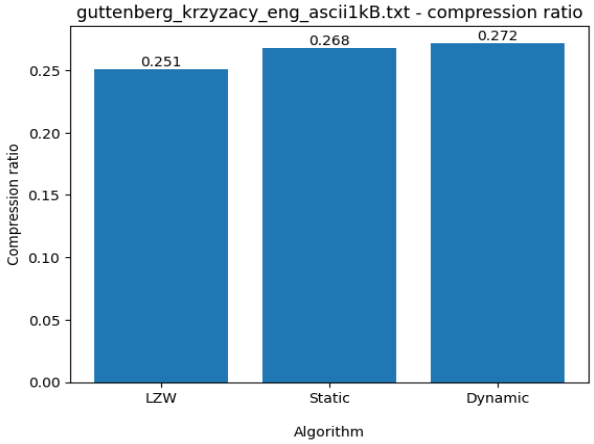
problemu mógłby wygrać 8 testów, gdyby jako podstawę kodowania przyjąć 128 znaków ASCII, a nie 256, ale byłby wtedy mniej uniwersalny.

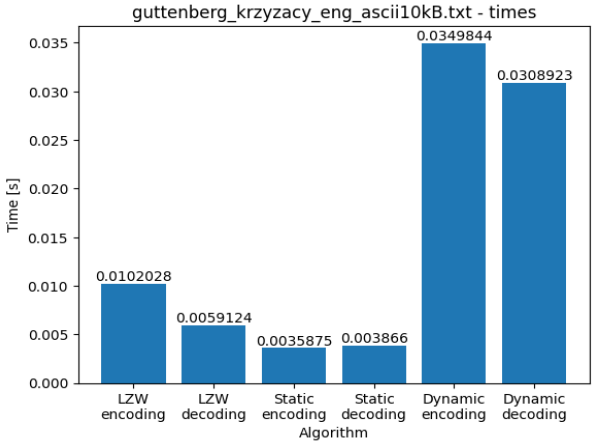
5 Szczegóły techniczne platformy testowej

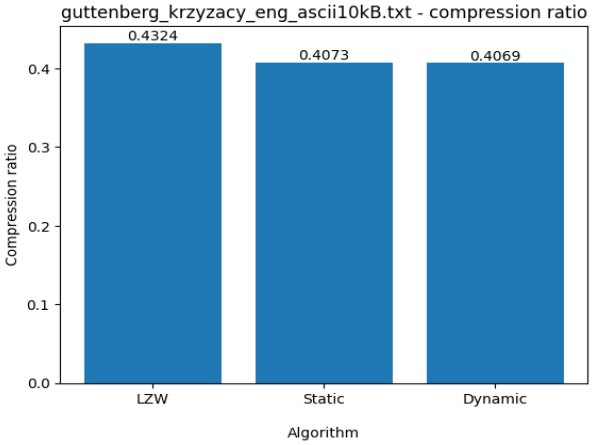
Wszystkie pomiary czasów zostały przeprowadzone na maszynie wirtualnej VirtualBox z systemem Ubuntu 20. Używana wersja języka Python to 3.8.5. Użyty procesor to AMD Ryzen 3900X. Poniższe pomiary czasów zostały powtórzone dziesięciokrotnie, a na wykresach zamieszczono średnie z tych pomiarów.

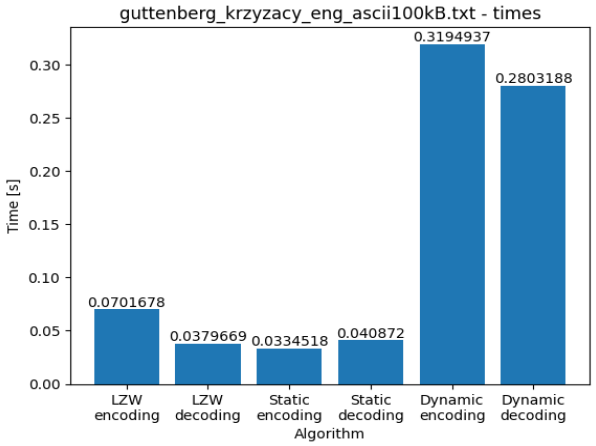
6 Benchmarki - wykresy

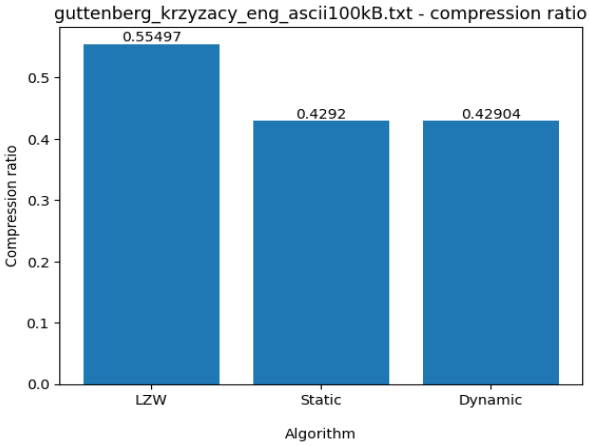


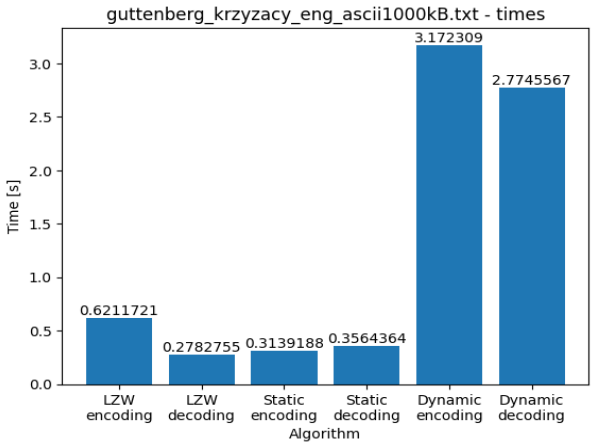


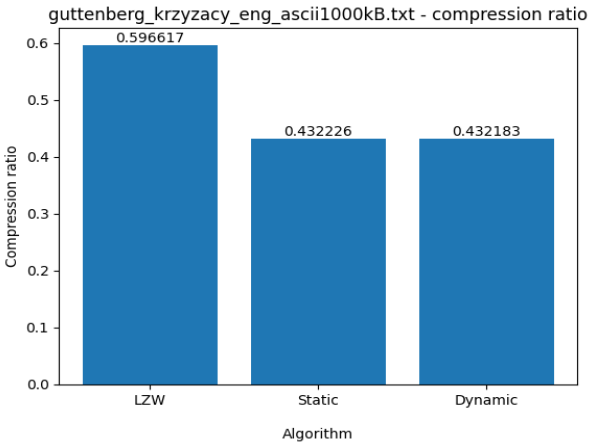


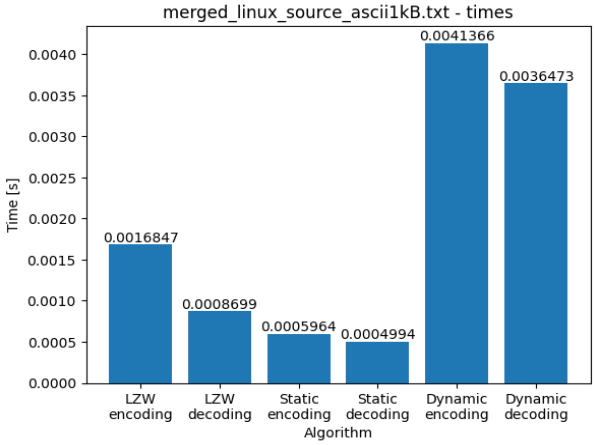


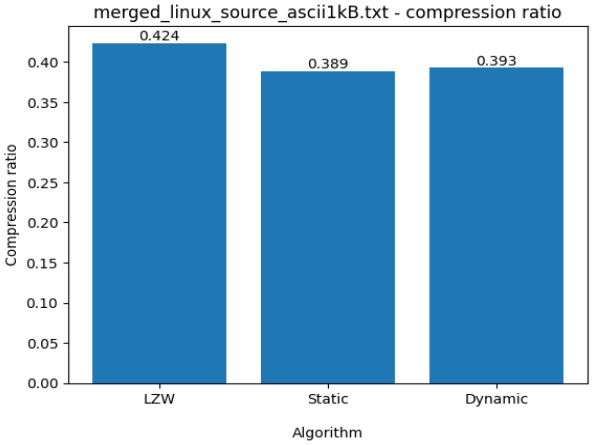


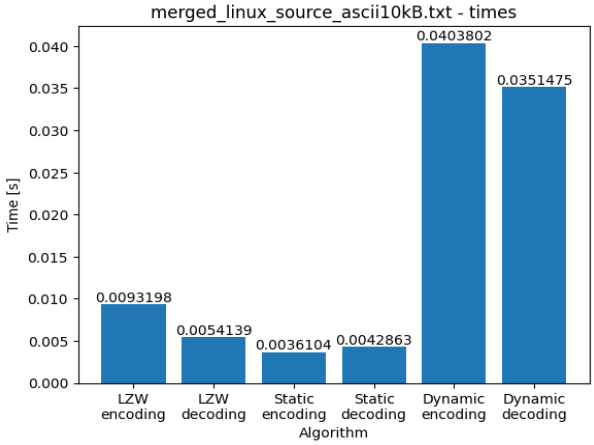


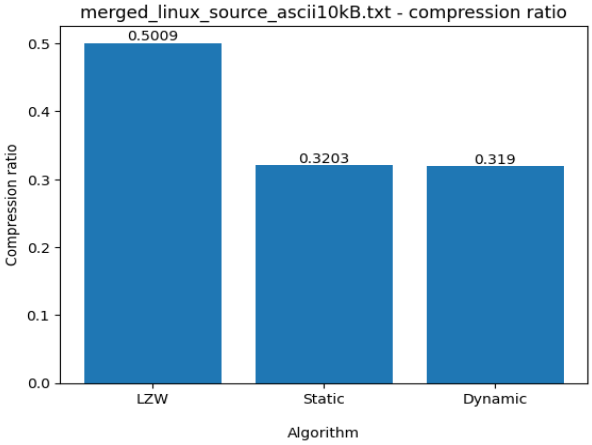

guttenberg_krzyzacy_eng_ascii1000kB.txt - times
Time [s]
0.6211721
0.2782755
0.3139188
0.3564364
3.172309
2.7745567
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm

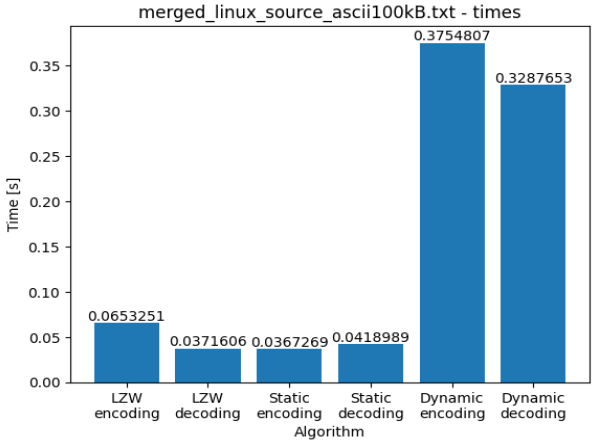

guttenberg_krzyzacy_eng_ascii1000kB.txt - compression ratio
Compression ratio
0.596617
0.432226
0.432183
LZW
Static
Dynamic
Algorithm

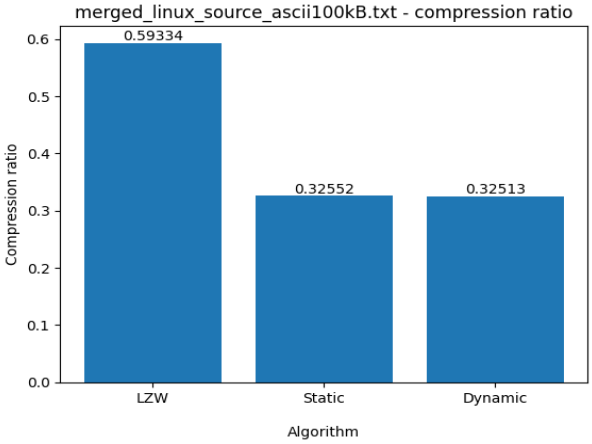

merged_linux_source_ascii1kB.txt - times
Time [s]
0.0016847
0.0008699
0.0005964
0.0004994
0.0041366
0.0036473
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm

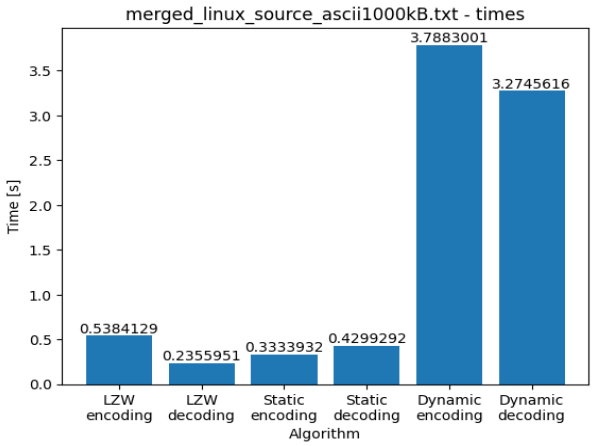

merged_linux_source_ascii1kB.txt - compression ratio
Compression ratio
0.424
0.389
0.393
LZW
Static
Dynamic
Algorithm

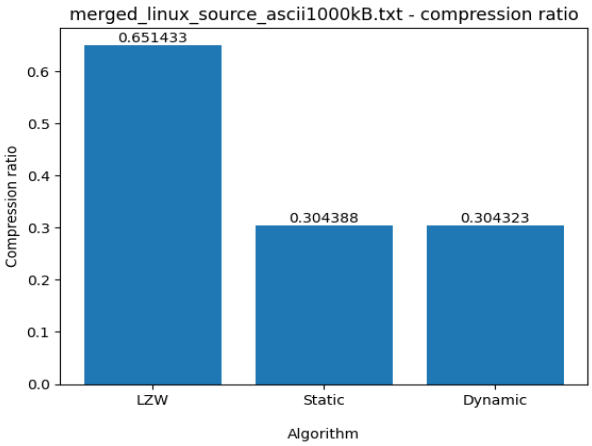

merged_linux_source_ascii10kB.txt - times
Time [s]
0.0093198
0.0054139
0.0036104
0.0042863
0.0403802
0.0351475
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm

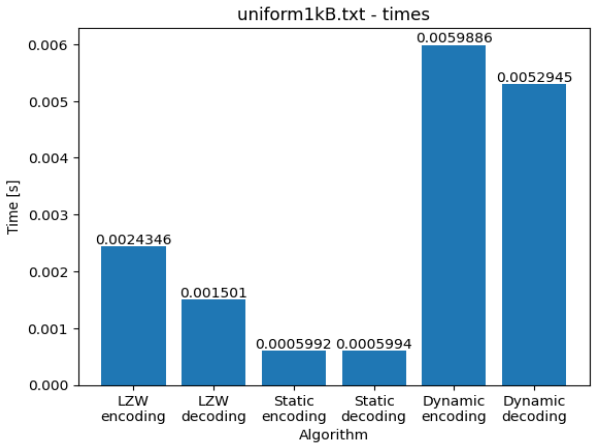

merged_linux_source_ascii10kB.txt - compression ratio
Compression ratio
0.5009
0.3203
0.319
LZW
Static
Dynamic
Algorithm

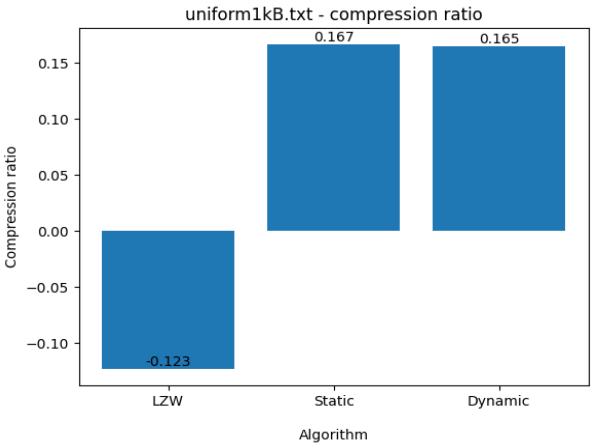

merged_linux_source_ascii100kB.txt - times
Time [s]
0.0653251
0.0371606
0.0367269
0.0418989
0.3754807
0.3287653
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm

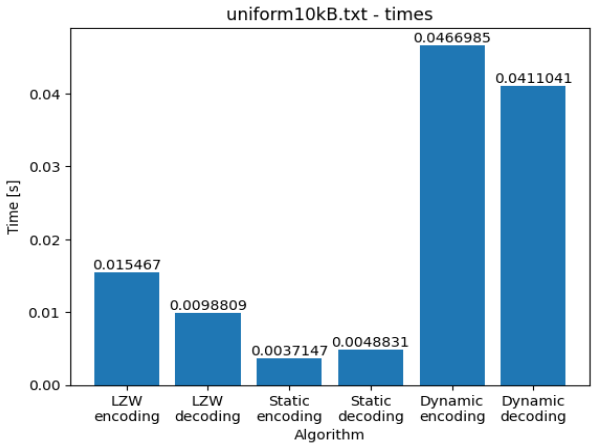

merged_linux_source_ascii100kB.txt - compression ratio
Compression ratio
0.59334
0.32552
0.32513
LZW
Static
Dynamic
Algorithm

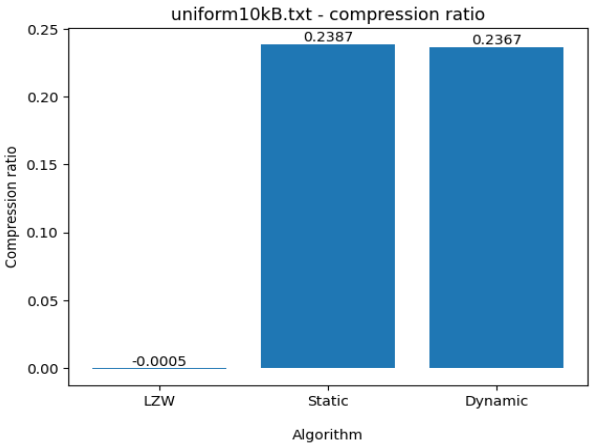

merged_linux_source_ascii1000kB.txt - times
Time [s]
0.5384129
0.2355951
0.3333932
0.4299292
3.7883001
3.2745616
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm

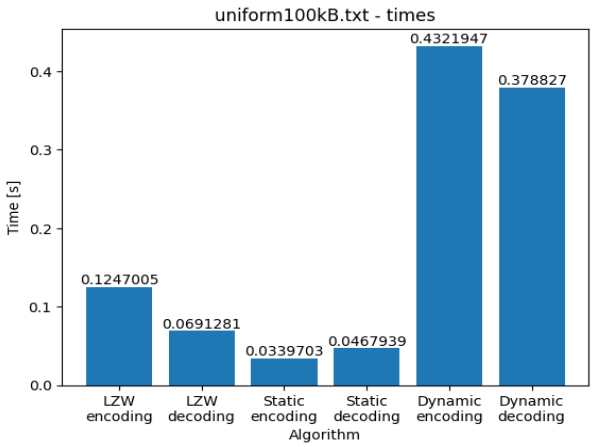

merged_linux_source_ascii1000kB.txt - compression ratio
Compression ratio
0.651433
0.304388
0.304323
LZW
Static
Dynamic
Algorithm

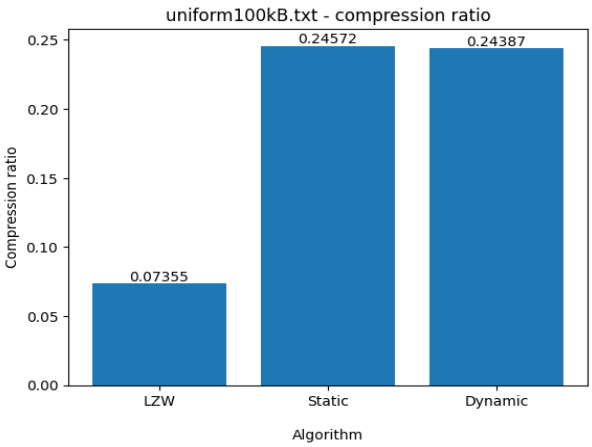

uniform1kB.txt - times
Time [s]
0.0024346
0.001501
0.0005992
0.0005994
0.0059886
0.0052945
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm

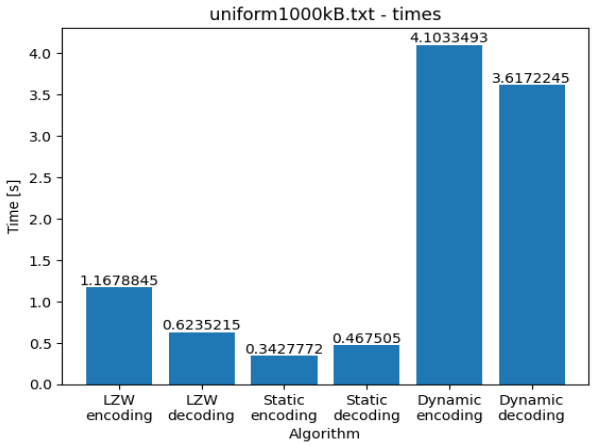

uniform1kB.txt - compression ratio
Compression ratio
0.167
0.165
-0.123
LZW
Static
Dynamic
Algorithm

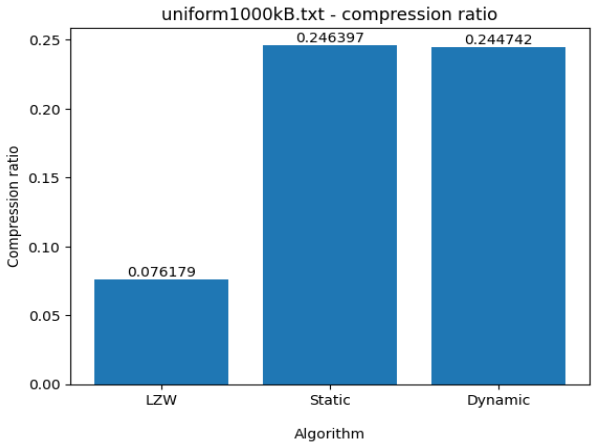

uniform10kB.txt - times
Time [s]
0.015467
0.0098809
0.0037147
0.0048831
0.0466985
0.0411041
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm


uniform10kB.txt - compression ratio
Compression ratio
0.2387
0.2367
-0.0005
LZW
Static
Dynamic
Algorithm


uniform100kB.txt - times
Time [s]
0.1247005
0.0691281
0.0339703
0.0467939
0.4321947
0.378827
LZW encoding
LZW decoding
Static encoding
Static decoding
Dynamic encoding
Dynamic decoding
Algorithm


uniform100kB.txt - compression ratio
Compression ratio
0.24572
0.24387
0.07355
LZW
Static
Dynamic
Algorithm





7 Kody modułów

7.1 Statyczne kodowanie Huffmana

```
from collections import deque
from bitarray import bitarray
from bitarray.util import ba2int, int2ba
from pretty_print import PrettyPrint
from node import Node


class StaticHuffman(PrettyPrint):

    def __init__(self, text):
        super().__init__()
        self.text = text

    def count_letter_frequency(self):
        atom = 1 / len(self.text)
        frequency = dict()
        for c in self.text:
            val = frequency.get(c, 0)
            frequency[c] = val + atom
        return frequency

    def get_two_min(self, A, B):
        res = []
        while len(res) < 2:
            a, b = 1, 1
            if A:
                a = A[0].weight
            if B:
                b = B[0].weight
            if a < b:
                res.append(A.popleft())
            else:
                res.append(B.popleft())
        return res[0], res[1]
```

```
35
36      def static_huffman(self, letter_frequency):
37          trees = deque()
38          leaves = [Node(label, frequency) for label, frequency in letter_frequency.items()]
39          if len(leaves) == 1:
40              return leaves[0]
41          leaves.sort(key=lambda node: node.weight)
42          leaves = deque(leaves)
43          while leaves or len(trees) > 1:
44              N1, N2 = self.get_two_min(trees, leaves)
45              trees.append(N1.join(N2))
46          return trees[0]
47
48      def calculate_huffman_tree(self):
49          self.root = self.static_huffman(self.count_letter_frequency())
50          return self.root
51
52
53  class StaticCompressor:
54      ''' This class can take any class with a 'calculate_huffman_tree()' method
55      which returns a prefix-free tree root (using nodes from the 'Node' import)
56      Probably not a very useful feature :)'''
57
58      # File encoding metadata
59      TRAILING_ZEROS_SIZE = 3
60
61      def __init__(self, filename, huffman_class=None):
62          self.filename = filename
63          self.huffman_tree_builder = huffman_class
64
65      def encode(self, text, display_tree=False):
66          '''Encodes 'text' to a file named self.filename\n
67          current formatting is:
68          bits[:3] - trailing zeros count (automatically appended zeros at the end of file)
69          then the tree is encoded followed by a single 0 bit, then the text in huffman code
70          followed by 0-7 trailing zeros. Something like this:
71          [trailing_zeros, tree, '0', text]'''
72
73          def tree_encoder(node, code=''):
74              '''Does post-order travelsal of the huffman tree
75              When it finds an inner node, '0' is printed
76              When it finds a leaf, it prints '1' followed by 8-bit ascii code
77              The tree size is 2m-1 where m is the alphabet size
78              hence the post-order travelsal takes (2m-1) + 8*m = (10m - 1) bits of space'''
79              if node.left:
80                  tree_encoder(node.left, code + '0')
81              if node.right:
82                  tree_encoder(node.right, code + '1')
83              if node.label:
84                  bits.append(1)
85                  bits.frombytes(node.label.encode())
86                  header_bit_count[0] += 9
87                  ascii_to_bin[node.label] = code
```

```
            else:
                bits.append(0)
                header_bit_count[0] += 1

        bits = bitarray()
        ascii_to_bin = dict()
        header_bit_count = [self.TRAILING_ZEROS_SIZE]
        H = self.huffman_tree_builder(text)
        tree_encoder(H.calculate_huffman_tree())
        bits.append(0)
        if display_tree:
            H.pretty_print()
        for c in text:
            bits.extend(bitarray(ascii_to_bin[c]))
        tmp = bitarray()
        length = (len(bits) + self.TRAILING_ZEROS_SIZE)
        tmp.extend(int2ba((8 - (length % 8)) % 8,
                          self.TRAILING_ZEROS_SIZE))   # trailing zeros count
        tmp.extend(bits)
        with open(self.filename, "wb") as f:
            tmp.tofile(f)

    def decode(self):
        '''Decodes file 'self.filename' using formatting explained in the 'encode()' docstring'''

        def tree_decoder():
            stack = []
            i = self.TRAILING_ZEROS_SIZE
            while bits[i] is not False or len(stack) != 1:
                if bits[i]:
                    i += 1
                    label = bits[i:i + 8].tobytes().decode()
                    i += 8
                    stack.append(Node(label))
                else:
                    i += 1
                    right = stack.pop()
                    left = stack.pop()
                    stack.append(Node(left=left, right=right))
            return stack[0], i + 1

        bits = bitarray()
        with open(self.filename, "rb") as f:
            bits.fromfile(f)
        trailing_zeros_count = ba2int(bits[:self.TRAILING_ZEROS_SIZE])
        node, header_bit_count = tree_decoder()
        root = node
        # In case there is only one letter in the tree:
        if node.left is None:
            node.left = node
            node.right = node
        charlist = []
        for i in range(header_bit_count, len(bits) - trailing_zeros_count):
```

```
            if node.label is not None:
                charlist.append(node.label)
                node = root
            if bits[i]:
                node = node.right
            else:
                node = node.left

        charlist.append(node.label)
        return ''.join(charlist)


if __name__ == '__main__':
    text = "abracadabra"
    S = StaticCompressor('compressed.static', StaticHuffman)
    S.encode(text, True)
    print(S.decode())
```

7.2 Dynamiczne kodowanie Huffmana

```
from bitarray import bitarray
from bitarray.util import ba2int, int2ba
from pretty_print import PrettyPrint
from node import Node


class ExtendedNode(Node):

    def __init__(self, index, label=None, weight=None, left=None, right=None, parent=None):
        super().__init__(label, weight, left, right)
        self.parent = parent
        self.index = index

    def swap_child_link(self, new_node, old_node):
        if self.left == old_node:
            self.left = new_node
        else:
            self.right = new_node

    def swap_index(self, node, lis):
        lis[self.index], lis[node.index] = lis[node.index], lis[self.index]
        self.index, node.index = node.index, self.index

    def swap(self, node, lis):
        self.swap_index(node, lis)
        if self.parent:
            self.parent.swap_child_link(node, self)
        if node.parent:
            node.parent.swap_child_link(self, node)
        node.parent, self.parent = self.parent, node.parent

```

```
32      def get_code(self):
33          code = []
34          node = self
35          while node.parent is not None:
36              if node.parent.left == node:
37                  code.append(False)
38              else:
39                  code.append(True)
40              node = node.parent
41          return reversed(code)
42
43
44  class DynamicHuffmanCompressor(PrettyPrint):
45      '''This class realizes dynamic huffman encoding and decoding'''
46
47      def __init__(self, filename):
48          super().__init__()
49          self.filename = filename
50          self.nodelist = []
51
52      def increment(self, node):
53          '''Takes care of weight increases and swaps if required'''
54          swap_index = node.index
55          while swap_index >= 0 and node.weight == self.nodelist[swap_index].weight:
56              swap_index -= 1
57          to_swap = self.nodelist[swap_index + 1]
58          if to_swap != node and to_swap != node.parent:
59              self.nodelist[swap_index + 1].swap(node, self.nodelist)
60          node.weight += 1
61          if node.parent:
62              self.increment(node.parent)
63
64      def create_inner_node(self, empty_node, new_leaf):
65          '''Creates new inner node, used when a new letter is met'''
66          new_inner_node = ExtendedNode(len(self.nodelist),
67                                        None,
68                                        0,
69                                        left=empty_node,
70                                        right=new_leaf,
71                                        parent=empty_node.parent)
72          if empty_node.parent:
73              empty_node.parent.swap_child_link(new_inner_node, empty_node)
74          new_leaf.parent = new_inner_node
75          empty_node.parent = new_inner_node
76          self.nodelist.append(new_inner_node)
77          new_inner_node.swap_index(empty_node, self.nodelist)
78          return new_inner_node
79
80      def add_letter(self, c, empty_node):
81          '''Adds a new letter to the tree'''
82          new_leaf = ExtendedNode(len(self.nodelist), c, 1)
83          self.nodelist.append(new_leaf)
84          new_inner_node = self.create_inner_node(empty_node, new_leaf)
```

```
85          self.increment(new_inner_node)
86          return new_leaf
87
88      def encode(self, text):
89          '''Dynamically encodes text as binary; every time a new letter is met, binary code of
90          the empty_node (representing all not yet met letters) is appended to binary code followed
91          by the ascii code of the letter; otherwise if a letter has already been in the tree, its
92          current code is appended and its weight is increased which also means that tree update
93          (increasing parents weights and swaping some nodes) may be required'''
94          empty_node = ExtendedNode(0, "##", 0)
95          nodes = {"##": empty_node}
96          self.nodelist = [empty_node]
97          bits = bitarray()
98          for c in text:
99              if c in nodes:
100                 bits.extend(nodes[c].get_code())
101                 self.increment(nodes[c])
102             else:
103                 bits.extend(empty_node.get_code())
104                 bits.frombytes(c.encode())
105                 nodes[c] = self.add_letter(c, empty_node)
106
107         self.root = self.nodelist[0]
108         trailing_zeros = (8 - ((len(bits) + 3) % 8)) % 8
109         tmp = bitarray(int2ba(trailing_zeros, 3))
110         tmp.extend(bits)
111         with open(self.filename, "wb") as f:
112             tmp.tofile(f)
113
114     def decode(self):
115         """Decodes binary file encoded by the 'encode(text)' method. Similarly to the encoding,
116         here the huffman tree is also dynamically adjusted while reading the binary file. Hence
117         it is not required to explicitly save huffman tree in the file thus allowing potentially
118         slighlty better compression ratio"""
119         empty_node = ExtendedNode(0, "##", 0)
120         nodes = {"##": empty_node}
121         self.nodelist = [empty_node]
122         bits = bitarray()
123         with open(self.filename, "rb") as f:
124             bits.fromfile(f)
125         trailing_zeros = ba2int(bits[:3])
126         charlist = []
127         i = 3
128         while i < len(bits) - trailing_zeros:
129             node = self.nodelist[0]
130             # nodes always have either both children or none, so checking for one is sufficient
131             while node.left:
132                 if bits[i]:
133                     node = node.right
134                 else:
135                     node = node.left
136                 i += 1
137             if node == empty_node:
```

```
                letter = bits[i:i + 8].tobytes().decode()
                i += 8
                nodes[letter] = self.add_letter(letter, empty_node)
            else:
                letter = node.label
                self.increment(nodes[letter])
            charlist.append(letter)
        return ''.join(charlist)


if __name__ == '__main__':
    text = "abracadabra"
    D = DynamicHuffmanCompressor('compressed.dynamic')
    D.encode(text)
    print(D.decode())
```

7.3 Lempel-Ziv-Welch

```
from bitarray import bitarray
from bitarray.util import ba2int, int2ba
from math import log2, ceil


class LZW:
    """Lempel-Ziv-Welch algorithm implementation with variable compression block
    The compression block dynamically changes as the dictionary fills. Default variability
    range is from 9 to 16 (first character is encoded with 8 bits).
    Changing block size can yield much better compression ratio,
    especially on natural language texts"""

    BLOCK_SIZE = 16
    ASCII_LEN = 256

    def __init__(self, filename):
        self.filename = filename
        self.MAX_LEN = int(2**self.BLOCK_SIZE)

    def encode(self, text):
        base_table = dict()
        for i in range(self.ASCII_LEN):
            base_table[chr(i)] = int2ba(i, self.BLOCK_SIZE)
        bits = bitarray()
        table = base_table.copy()
        table_len = self.ASCII_LEN
        string = text[0]
        for i in range(1, len(text)):
            char = text[i]
            if string + char in table:
                string += char
            else:
                bits.extend(table[string][-ceil(log2(table_len)):])
```

```
                if table_len == self.MAX_LEN:
                    table = base_table.copy()
                    table_len = self.ASCII_LEN
                table[string + char] = int2ba(table_len, self.BLOCK_SIZE)
                table_len += 1
                string = char
        bits.extend(table[string][-ceil(log2(table_len)):])
        with open(self.filename, "wb") as f:
            bits.tofile(f)

    def decode(self):
        table = [None] * self.MAX_LEN
        for i in range(self.ASCII_LEN):
            table[i] = chr(i)
        bits = bitarray()
        with open(self.filename, "rb") as f:
            bits.fromfile(f)
        table_len = self.ASCII_LEN
        start = ceil(log2(table_len))
        string = table[ba2int(bits[:start])]
        text = string
        # If the character count is not a multiply of BLOCK_SIZE, there were some
        # automatically appended zeros at the end of file to fill whole byte, we skip these
        # It's not actually needed to work, but is a nice thing to expose
        actual_size = len(bits) - (len(bits) % self.BLOCK_SIZE)
        i = start
        while i < actual_size:
            if table_len == self.MAX_LEN:
                table_len = self.ASCII_LEN
            jump = ceil(log2(table_len + 1))
            code = ba2int(bits[i:i + jump])
            if code >= table_len:
                entry = string + string[0]
            else:
                entry = table[code]
            text += entry
            table[table_len] = string + entry[0]
            table_len += 1
            string = entry
            i += jump
        return text


if __name__ == '__main__':
    text = 'abracadabra'
    C = LZW('compressed.lzw')
    # C.encode(text)
    print(C.decode())
```

7.4 Moduły pomocnicze

7.4.1 Node

```
class Node:
    node_count = 0

    def __init__(self, label=None, weight=None, left=None, right=None):
        self.ind = Node.node_count
        Node.node_count += 1
        self.label = label
        self.weight = weight
        self.left = left
        self.right = right

    def join(self, node):
        return Node(None, self.weight + node.weight, self, node)

    def __repr__(self):
        if self.weight:
            return "Index: " + str(self.ind) + "\nLabel: " + str(
                self.label) + "\nWeight: " + str(round(self.weight, 2))
        else:
            return "Index: " + str(self.ind) + "\nLabel: " + str(self.label)
```

7.4.2 Pretty Print

```
import pydot
import tempfile
from PIL import Image


class PrettyPrint:
    '''Base class for printing trees using pydot's write method
    Tree node needs to have 'left' and 'right' attributes'''

    def __init__(self):
        self.root = None

    def pretty_print(self, name=None, display=True):
        '''Prints a tree as an image\n
        name - if provided, saves tree image to that filename,
        display - wheter to display the image'''

        def dfs_helper(node, graph):
            for c in [node.left, node.right]:
                if c:
                    graph.add_edge(pydot.Edge(str(node), str(c)))
                    dfs_helper(c, graph)

        if self.root is None:
            print("Root is None!")
```

```
            return

        graph = pydot.Dot(graph_type='graph')
        dfs_helper(self.root, graph)
        if name is None:
            fout = tempfile.NamedTemporaryFile(suffix=".png")
            name = fout.name
        else:
            name += "_tree.png"
        graph.write(name, format="png")
        if display:
            Image.open(name).show()
```

7.4.3 Benchmark

```
from lempel_ziv_welch import LZW
from static_huffman import StaticHuffman, StaticCompressor
from dynamic_huffman import DynamicHuffmanCompressor
from time import time
import matplotlib.pyplot as plt
from os.path import getsize


def benchmark(funclist, datlist, labels, repeats=None):
    if repeats is None:
        repeats = 1
    xs = []
    ys = []
    for dat in datlist:
        times = dict()
        for func, label in zip(funclist, labels):
            for _ in range(repeats):
                t_enc, t_dec = func(dat)
                key1, key2 = label + '\nencoding', label + '\ndecoding'
                times[key1] = times.get(key1, 0) + (t_enc / repeats)
                times[key2] = times.get(key2, 0) + (t_dec / repeats)
        xs.append(times.keys())
        ys.append(times.values())
    return xs, ys


def compression_ratio_benchmark(funclist, datlist, labels, repeats=1):
    xs = []
    ys = []
    for dat in datlist:
        times = dict()
        for func, label in zip(funclist, labels):
            for _ in range(repeats):
                ratio = func(dat)
                key = label + '\n'
                times[key] = times.get(key, 0) + (ratio) / repeats
```

```
37          xs.append(times.keys())
38          ys.append(times.values())
39      return xs, ys
40
41
42  def plot_factory(xs, ys, titles, axis_labels=None, save_to_file=False, display=True):
43      for i in range(len(xs)):
44          plt.bar(xs[i], ys[i])
45          plt.title(titles[i])
46          for j, v in enumerate(ys[i]):
47              plt.annotate(str(round(v, 7)), xy=(j, v), ha='center', va='bottom')
48          if axis_labels is not None:
49              plt.xlabel(axis_labels['x'])
50              plt.ylabel(axis_labels['y'])
51
52          if save_to_file:
53              plt.savefig("plots/" + titles[i] + ".png", bbox_inches='tight')
54          if display:
55              plt.show()
56
57
58  def universal_compression_wrapper(source_filename, compressed_filename, C):
59      uncompressed = getsize('../sources/' + source_filename)
60      with open('../sources/' + source_filename, "r") as f:
61          text = f.read()
62      C.encode(text)
63      compressed = getsize(compressed_filename)
64      return 1 - (compressed / uncompressed)
65
66
67  def LZW_compression_benchmark_wrapper(source_filename):
68      filename = 'compressed.lzw'
69      C = LZW(filename)
70      return universal_compression_wrapper(source_filename, filename, C)
71
72
73  def static_compression_benchmark_wrapper(source_filename):
74      filename = 'compressed.static'
75      C = StaticCompressor(filename, StaticHuffman)
76      return universal_compression_wrapper(source_filename, filename, C)
77
78
79  def dynamic_compression_benchmark_wrapper(source_filename):
80      filename = 'compressed.dynamic'
81      C = DynamicHuffmanCompressor(filename)
82      return universal_compression_wrapper(source_filename, filename, C)
83
84
85  def universal_wrapper(source_filename, C):
86      with open('../sources/' + source_filename, "r") as f:
87          text = f.read()
88      t_enc1 = time()
89      C.encode(text)
```

```
    t_enc2 = time()
    t_dec1 = time()
    C.decode()
    t_dec2 = time()
    return t_enc2 - t_enc1, t_dec2 - t_dec1


def LZW_benchmark_wrapper(source_filename):
    C = LZW('compressed.lzw')
    return universal_wrapper(source_filename, C)


def static_benchmark_wrapper(source_filename):
    C = StaticCompressor('compressed.static', StaticHuffman)
    return universal_wrapper(source_filename, C)


def dynamic_benchmark_wrapper(source_filename):
    C = DynamicHuffmanCompressor('compressed.dynamic')
    return universal_wrapper(source_filename, C)


if __name__ == '__main__':
    labels = ["LZW", "Static", "Dynamic"]
    datlist = [
        name + str(i) + 'kB.txt'
        for i in [1, 10, 100, 1000]
        for name in ['guttenberg_krzyzacy_eng_ascii', 'merged_linux_source_ascii', 'uniform']
    ]
    # time benchmarks
    funclist = [LZW_benchmark_wrapper, static_benchmark_wrapper, dynamic_benchmark_wrapper]
    xs, ys = benchmark(funclist, datlist, labels, 10)
    # compression ratio benchmarks
    funclist = [
        LZW_compression_benchmark_wrapper, static_compression_benchmark_wrapper,
        dynamic_compression_benchmark_wrapper
    ]
    xs2, ys2 = compression_ratio_benchmark(funclist, datlist, labels)
    # Plotting
    titles = [s + " - times" for s in datlist]
    plot_factory(xs, ys, titles, {'x': 'Algorithm', 'y': 'Time [s]'}, True)
    titles = [s + " - compression ratio" for s in datlist]
    plot_factory(xs2, ys2, titles, {'x': 'Algorithm', 'y': 'Compression ratio'}, True)
```