Algorytmy tekstowe – Algorytmy kompresji

Tomasz Gargula

1 Wstęp

W raporcie opiszę i porównam implementacje statycznego i dynamicznego drzewa Huffmana. Przy implementacji skorzystałem między innymi z bibliotek: `bitarray` oraz `collections.deque`. Ćwiczenie wykonałem, korzystając z paradygmatu obiektowego.

2 Implementacja algorytmów

W następnych podsekcjach opiszę implementację poszczególnych części programu.

2.1 Generator

Aby uzyskać pliki z jednostajnym rozkładem liter, skorzystałem z poniższego programu.

```

```

2.2 Funkcje pomocnicze

Poniżej zamieszczone są funkcje pomocnicze, które przydały się przy implementacji algorytmów Huffmana. Dodatkowo definiuję zmienne globalne `BITS_PER_LETTER` oraz `BITS_PER_SIZE`, które określają liczbę bitów przeznaczanych na przechowywanie znaków (16 bitów pozwala na kodowanie polskich znaków diakrytycznych) oraz na przechowywanie rozmiaru drzewa. Określenie rozmiaru struktury jest konieczne, ponieważ translacja do kodu binarnego powoduje dopisanie bitów do pełnych bajtów. Dopisywane są wtedy zera do zakodowanej struktury. Może więc zdarzyć się sytuacja, że po odkodowaniu otrzymamy jedną lub dwie litery więcej.

```python
BITS_PER_LETTER = 16
BITS_PER_SIZE = 64

def extract_min(leafs, nodes):
    if not leafs:
        return nodes.pop()
    if not nodes:
        return leafs.pop()
    return nodes.pop() if nodes[-1].weight < leafs[-1].weight else leafs.pop()


def count_letters(text):
    dictionary = {}
    for letter in text:
        dictionary[letter] = dictionary.get(letter, 0) + 1
    return dictionary


def utf8(letter):
    return bin(ord(letter))[2:].zfill(BITS_PER_LETTER)
```

2.3 Węzły

Poniżej zamieszczam implementację węzłów wykorzystywanych w strukturach drzew Huffmana. Warto zauważyć, że określenie dzieci przy tworzeniu węzła, rejestruje węzeł automatycznie jako rodzica. Metoda `_save`, podobnie jak w innych strkuturach, służy do wizualizacji drzewa przy pomocy biblioteki `pydot`.

```
class Node:
    def __init__(self, weight, left=None, right=None):
        self.left = left
        self.right = right
        self.weight = weight
        self.parent = None
        self.visited = False
        if left: left.parent = self
        if right: right.parent = self

    def postorder(self):
        return self.left.postorder() + self.right.postorder() + [self]

    def _save(self, G, parent=None, i=None):
        node = pydot.Node(str(self), label=str(self.weight), shape="circle")

        G.add_node(node)
        if parent:
            edge = pydot.Edge(parent, node, label=str(i))
            G.add_edge(edge)
        for i, child in enumerate((self.left, self.right)):
            child._save(G, node, i)
```

2.4 Liście

Zdecydowałem się wyodrębnić osobną klasę reprezentującą liście, która dodatkowo posiada metodę `code()`, która zwraca aktualną reprezentację bitową litery znajdującej się w liściu.

```
class Leaf(Node):
    def __init__(self, weight, letter):
        super().__init__(weight)
        self.letter = letter

    def postorder(self):
        return [self]

    def code(self):
        pointer = self
        code = []
        while pointer.parent is not None:
            parent = pointer.parent
            if parent.left == pointer:
                code.append("0")
            if parent.right == pointer:
                code.append("1")
            pointer = parent
        return ''.join(reversed(code))

    def _save(self, G, parent, i):
        node = pydot.Node(
            str(self), label=f'{self.letter}: {self.weight}', shape="square")
        G.add_node(node)
```

```
        edge = pydot.Edge(parent, node, label=str(i))
        G.add_edge(edge)
```

2.5 Lista

Dynamiczne drzewo Huffmana wykorzystuje zależność, że jeśli T jest kompletnym ważonym drzewem binarnym z p liśćmi z dodatnimi wagami, a waga rodzica jest równa sumie wag dzieci, to węzły mogą być ułożone w ciąg niemalejący $(w(x_1), w(x_2), w(x_3), ..., w(x_n)$ taki, że dla każdego i x_{2i} oraz x_{2i+1} są bliźniakami tj. mają tego samego rodzica. Gdy przetwarzamy nową literę, to zwiększamy jej wagę. Aby utrzymać poprawność struktury potrzebny jest szybki dostęp do następnego węzła. Aby to zapewnić, skorzystałem ze wzorca `Decorator`, obudowując zwykłą pythonową listę i dodając do każdego węzła informację o tym, na którym miejscu w liście się znajduje.

```
'''Decorator class for List that saves info about the index'''
class List:
    def __init__(self, *nodes):
        self.structure = []
        self.extend(*nodes)

    def __getitem__(self, index):
        return self.structure[index]

    def __setitem__(self, index, node):
        self.structure[index] = node
        node.index = index

    def append(self, node):
        self.structure.append(node)
        node.index = len(self.structure) - 1

    def extend(self, *nodes):
        for node in nodes:
            self.append(node)

    def pop(self):
        node = self.structure.pop()
        node.index = None
```

2.6 Prototyp drzewa Huffmana

Statyczne i dynamiczne drzewo Huffmana mają kilka metod wspólnych lub działających na tej samej zasadzie. Korzystając więc z mechanizmu dziedziczenia, zdefiniowałem klasę HuffmanTree.

```
class HuffmanTree:
    def __init__(self, root):
        self.root = root
        self.codes, self.letters = self._get_encodings()

    def save(self, filename='out.png'):
        G = pydot.Dot(graph_type="graph")
        self.root._save(G)
        G.write_png(filename)

    def _get_encodings(self):
        codes, letters = {}, {}
        pointer = self.root
        code = []
        while pointer:
            if pointer.left is None:
```

```
                encoded = ''.join(code)
                codes[pointer.letter] = encoded
                letters[encoded] = pointer.letter
                pointer.visited = True
                code.pop()
                pointer = pointer.parent
            else:
                if not pointer.left.visited:
                    code.append('0')
                    pointer = pointer.left
                elif not pointer.right.visited:
                    code.append('1')
                    pointer = pointer.right
                else:
                    pointer.visited = True
                    code = code[:-1]
                    pointer = pointer.parent
        return codes, letters

    def decode_size(self, binary):
        return int(binary[:BITS_PER_SIZE].to01(), 2)
```

2.7 Statyczne drzewo Huffmana

Poniżej zamieściłem implementację statycznego drzewa Huffmana.

```
class StaticHuffmanTree(HuffmanTree):
    def __init__(self, text=None):
        if text is not None:
            super().__init__(self._build(count_letters(text)))
            self.encoded = self.encode(text).tobytes()
        else:
            self.root = None

    def get_encoded(self):
        return self.encoded

    def _build(self, dictionary):
        leaves = [Leaf(weight, letter) for letter, weight in
                  sorted([item for item in dictionary.items()], key=lambda item: -item[1])]
        nodes = deque()
        if len(leaves) == 1:
            nodes.append(leaves.pop())
        while len(leaves) + len(nodes) > 1:
            node1 = extract_min(leaves, nodes)
            node2 = extract_min(leaves, nodes)
            nodes.appendleft(Node(node1.weight + node2.weight, node1, node2))
        return nodes.pop()

    def _encode_tree(self):
        return bitarray(''.join(
            ['1' + utf8(node.letter) if isinstance(node, Leaf) else '0' for node in
             ↪  self.root.postorder()]
        ) + '0')

    def _encode_size(self, text):
        return bitarray(bin(len(text))[2:].zfill(BITS_PER_SIZE))

```

```
    def _encode_text(self, text):
        return bitarray(''.join([self.codes[letter] for letter in text]))

    def encode(self, text):
        return self._encode_size(text) + self._encode_tree() + self._encode_text(text)

    def decode_tree(self, binary):
        stack = []
        j = 0
        while j < len(binary):
            j += 1
            if binary[j-1]:
                stack.append(Leaf(None, chr(int(binary[j:j+BITS_PER_LETTER].to01(), 2))))
                j += BITS_PER_LETTER
            else:
                if len(stack) == 1:
                    self.root = stack.pop()
                    self.codes, self.letters = self._get_encodings()
                    return j
                node1, node2 = stack.pop(), stack.pop()
                stack.append(Node(None, node2, node1))

    def decode_text(self, binary, size):
        i, j = 0, 1
        result = []
        while i < len(binary) and (size is None or len(result) < size):
            while binary[i:j].to01() not in self.letters:
                j += 1
            result += self.letters[binary[i:j].to01()]
            i = j
        return ''.join(result)

    def decode(self, binary):
        size = self.decode_size(binary)
        text_begin = self.decode_tree(binary[BITS_PER_SIZE:])
        return self.decode_text(binary[BITS_PER_SIZE + text_begin:], size)
```

2.8 Dynamiczne drzewo Huffmana

Poniżej znajduje się implementacja dynamicznego drzewa Huffmana.

```
class DynamicHuffmanTree(HuffmanTree):
    def __init__(self, stream=[]):
        self.root = Leaf(0, None)
        self.NYT = self.root
        self.pointers = {}
        self.encoded = bitarray()
        self.nodes = List(self.NYT)
        for letter in stream:
            self.add(letter)

    def get_encoded(self):
        return bitarray(bin(self.root.weight)[2:].zfill(BITS_PER_SIZE)) + self.encoded

    def decode(self, binary):
        pointer = self.root
        decoded = []
        skipping = 0
```

```
18
19          size = decode_size(binary)
20          print(size)
21
22          binary = binary[BITS_PER_SIZE:]
23
24          for i, bit in enumerate(binary):
25              skipping = skipping - 1 if skipping > 0 else 0
26              if not size:
27                  break
28              if not skipping:
29                  if isinstance(pointer, Leaf):
30                      size -= 1
31                      if pointer is self.NYT:
32                          letter = chr(int(binary[i:i+BITS_PER_LETTER].to01(), 2))
33                          print(binary[i:i+BITS_PER_LETTER])
34                          skipping = BITS_PER_LETTER
35                          self.add(letter)
36                          decoded.append(letter)
37                          pointer = self.root
38                          continue
39                      else:
40                          decoded.append(pointer.letter)
41                          print(pointer.letter)
42                          self.add(pointer.letter)
43                          pointer = self.root
44
45                  pointer = pointer.right if bit else pointer.left
46
47          skipping = skipping - 1 if skipping > 0 else 0
48          if not skipping:
49              if isinstance(pointer, Leaf):
50                  if pointer is self.NYT:
51                      letter = chr(int(binary[i:i+BITS_PER_LETTER].to01(), 2))
52                      skipping = BITS_PER_LETTER
53                      self.add(letter)
54                      decoded.append(letter)
55                      pointer = self.root
56                  else:
57                      decoded.append(pointer.letter)
58                      self.add(pointer.letter)
59                      pointer = self.root
60
61          return ''.join(decoded)
62
63      def next(self, node):
64          return self.nodes[node.index - 1] if node.index > 0 else None
65
66      def add(self, letter):
67          pointer = self.root
68          if letter in self.pointers:  # has been transferred
69              leaf = self.pointers[letter]
70              self.encoded.extend(leaf.code())
71              p = leaf
72              next = self.next(p)
73              while next is not None and not (isinstance(p, Leaf) ^ isinstance(next, Leaf)) and
                ↪  p.weight == next.weight:
```

```
                self._swap(p, next)
                next = self.next(p)
            if p.parent.left is self.NYT:
                leaf = p
                p = p.parent
            else:
                leaf = None

        else:  # not yet transferred (NYT)
            self.encoded.extend(self.NYT.code() + utf8(letter))
            parent = self.NYT.parent
            self.nodes.pop()
            leaf = Leaf(0, letter)
            node = Node(0, self.NYT, leaf)
            self.nodes.extend(node, leaf, self.NYT)
            if self.NYT is self.root:
                self.root = node
            if parent is not None:
                parent.left = node
            node.parent = parent
            self.pointers[letter] = leaf
            p = node

        self._update(p, leaf)

    def _swap(self, node1, node2):
        index1 = node1.index
        index2 = node2.index
        self.nodes[index1], self.nodes[index2] = self.nodes[index2], self.nodes[index1]
        parent1 = node1.parent
        parent2 = node2.parent
        if parent1.left is node1 and parent2.left is node2:
            parent1.left, parent2.left = parent2.left, parent1.left
        elif parent1.left is node1:
            parent1.left, parent2.right = parent2.right, parent1.left
        elif parent2.left is node2:
            parent1.right, parent2.left = parent2.left, parent1.right
        else:
            parent1.right, parent2.right = parent2.right, parent1.right
        node1.parent = parent2
        node2.parent = parent1

    def _update(self, node, leaf):
        parent = node.parent
        while node is not None:

            next = self.next(node)
            while next is not None and isinstance(next, Leaf) and next.weight == node.weight +
            ↪  1:
                self._swap(node, next)
                next = self.next(node)
            node.weight += 1
            node = parent
            if node is not None:
                parent = node.parent

        if leaf is not None:
```

```
            next = self.next(leaf)
            while next is not None and not isinstance(next, Leaf) and next.weight ==
            ↪  leaf.weight:
                self._swap(leaf, next)
                next = self.next(leaf)
            leaf.weight += 1
```

2.9 Testy

W celu sprawdzenia, czy algorytmy są w stanie odkodować zakodowany tekst napisałem test, który zamieszczam poniżej. W tym celu skorzystałem z możliwości biblioteki `pytest`

```
import pytest
from os import listdir
from bitarray import bitarray

from huffman.huffman import StaticHuffmanTree, DynamicHuffmanTree


def test_decompress():
    for filename in listdir("data"):
        encodedfile = ''.join(filename.split('.')[:-1] + [".bin"])
        with open(f"data/{filename}", "r") as f:
            text = f.read()

        for HuffmanTree in (StaticHuffmanTree, DynamicHuffmanTree,):

            with open(f"encoded/{HuffmanTree.__name__}/{encodedfile}", "rb") as f:
                array = bitarray()
                array.fromfile(f)
                decoded = HuffmanTree().decode(array)

            assert text == decoded
```

2.10 Benchmarki

Poniżej zamieszczam kod odpowiedzialny za wykonanie pomiarów i sporządzenie wykresów.

```
from os import listdir
from timeit import default_timer as timer
import matplotlib
from matplotlib import pyplot as plt
import numpy as np

from huffman.huffman import DynamicHuffmanTree, StaticHuffmanTree


def benchmark(f, *args):
    start = timer()
    result = f(*args)
    return timer() - start, result


if __name__ == '__main__':

    data = ([], [], [])

    for filename in sorted(listdir("data")):
```

```
        data[0].append(filename)

        for i, HuffmanTree in enumerate((StaticHuffmanTree, DynamicHuffmanTree)):
            file = filename

            with open(f"data/{file}", "r") as f:
                elapsed, HT = benchmark(HuffmanTree, f.read())


            data[i+1].append(elapsed)

            file = ''.join(file.split('.')[:-1] + [".bin"])

            # since we have built the tree, we can save it to file
            with open(f"encoded/{HuffmanTree.__name__}/{file}", "wb") as f:
                f.write(HT.get_encoded())


    width = 0.35

    independent = data[0][:-3], data[1][:-3], data[2][:-3]
    dependent = data[0][-3:], data[1][-3:], data[2][-3:]

    x = np.arange(len(independent[0]))

    fig, ax = plt.subplots()
    rects1 = ax.bar(x - width / 2, independent[1], width, label='static')
    rects2 = ax.bar(x + width / 2, independent[2], width, label='dynamic')
    ax.set_title('Static and dynamic huffman encoding comparison')
    ax.set_ylabel('Time [s]')
    ax.set_xticks(x)
    ax.set_yscale('log')
    ax.set_xticklabels(independent[0], rotation="vertical", fontsize="x-small")
    ax.legend()

    for rect, label in zip(rects1, independent[1]):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height, round(label,3), ha='center', va='bottom', fontsize="x-small")

    for rect, label in zip(rects2, independent[2]):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height, round(label,3), ha='center', va='bottom', fontsize="x-small")

    fig.tight_layout()

    plt.show()

    x = np.arange(len(dependent[0]))

    fig, ax = plt.subplots()
    rects1 = ax.bar(x - width / 2, dependent[1], width, label='static')
    rects2 = ax.bar(x + width / 2, dependent[2], width, label='dynamic')
    ax.set_title('Static and dynamic huffman encoding comparison')
    ax.set_ylabel('Time [s]')
```

```
    ax.set_xticks(x)
    ax.set_yscale('log')
    ax.set_xticklabels(dependent[0], rotation="vertical", fontsize="x-small")
    ax.legend()

    for rect, label in zip(rects1, dependent[1]):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height, round(label,3), ha='center', va='bottom', fontsize="x-small")

    for rect, label in zip(rects2, dependent[2]):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height, round(label,3), ha='center', va='bottom', fontsize="x-small")

    fig.tight_layout()

    plt.show()
```

2.11 Wyliczenie współczynników kompresji

Poniżej zamieszczam kod odpowiedzialny za wylieczenie współczynników kompresji i sporządzenie wykresów.

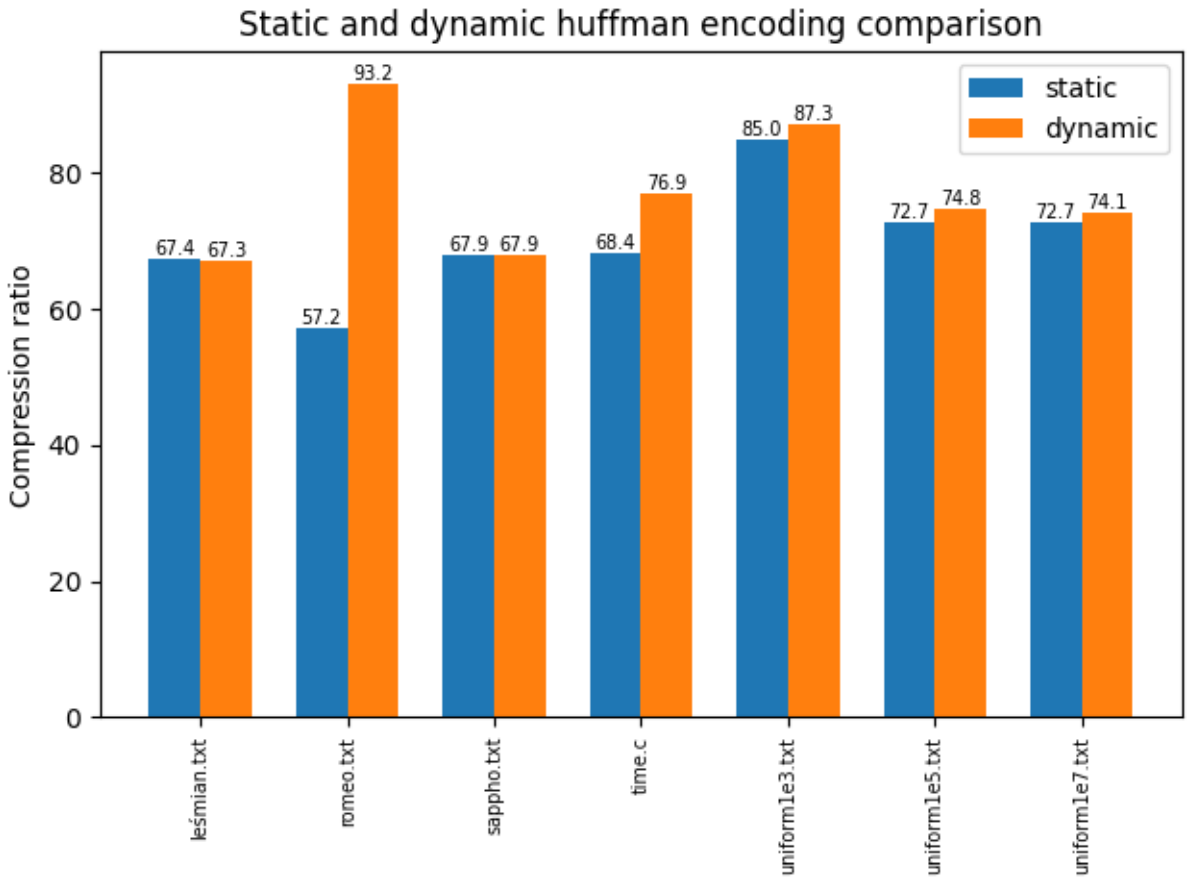
```
import os
import matplotlib
from matplotlib import pyplot as plt
import numpy as np

if __name__ == '__main__':

    data = ([], [], [])

    for filename in sorted(os.listdir("data")):

        original = os.path.getsize(f"data/{filename}")

        encoded = ''.join(filename.split('.')[:-1] + [".bin"])

        static = os.path.getsize(f"encoded/StaticHuffmanTree/{encoded}")

        dynamic = os.path.getsize(f"encoded/DynamicHuffmanTree/{encoded}")

        data[0].append(filename)
        data[1].append(static / original * 100)
        data[2].append(dynamic / original * 100)

    x = np.arange(len(data[0]))  # the label locations
    width = 0.35  # the width of the bars

    fig, ax = plt.subplots()
    rects1 = ax.bar(x - width / 2, data[1], width, label='static')
    rects2 = ax.bar(x + width / 2, data[2], width, label='dynamic')
    ax.set_title('Static and dynamic huffman encoding comparison')
    ax.set_ylabel('Compression ratio')
    ax.set_xticks(x)
    ax.set_xticklabels(data[0], rotation="vertical", fontsize="x-small")
    ax.legend()

```

```
    for rect, label in zip(rects1, data[1]):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height, round(label,1), ha='center', va='bottom', fontsize="x-small")

    for rect, label in zip(rects2, data[2]):
        height = rect.get_height()
        ax.text(rect.get_x() + rect.get_width() / 2,
                height, round(label,1), ha='center', va='bottom', fontsize="x-small")

    fig.tight_layout()

    plt.show()
```

3 Porównanie współczynnika kompresji

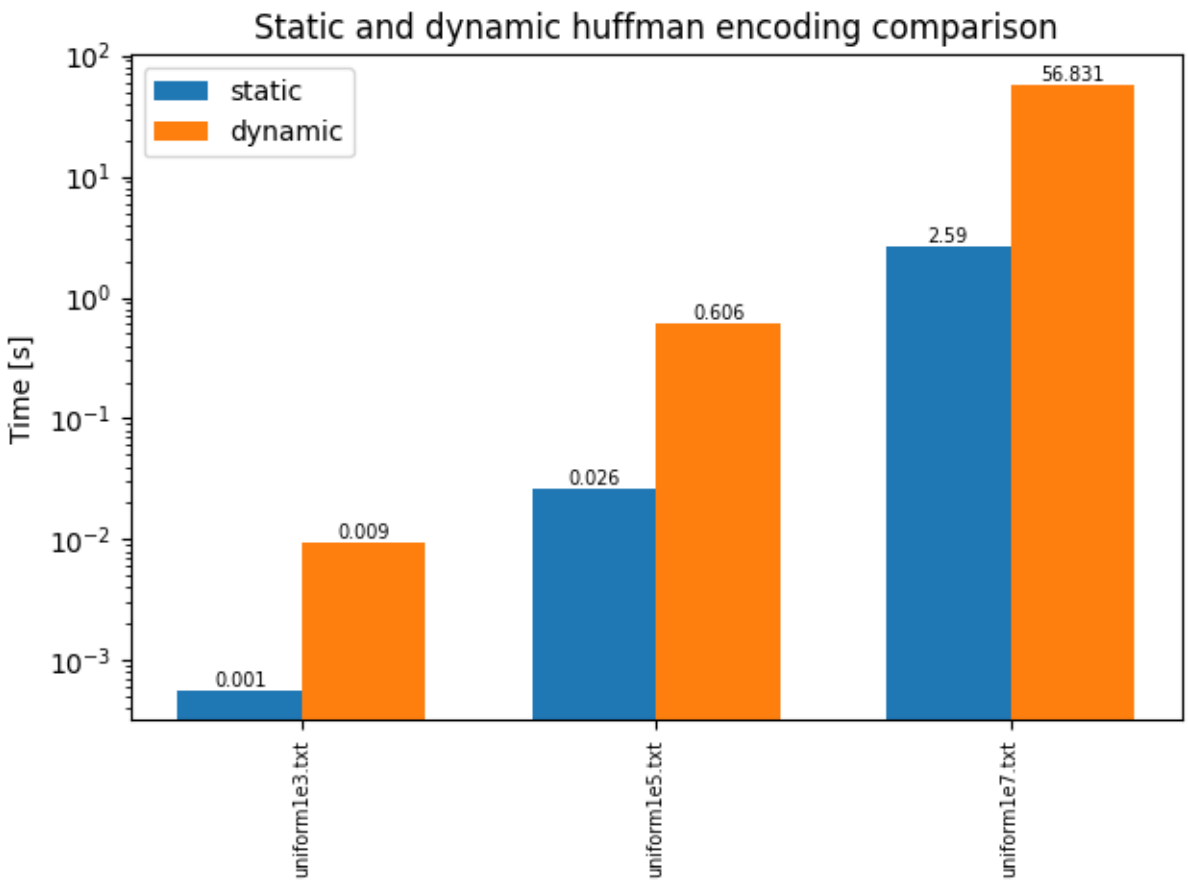
Poniżej zamieszczam porównanie współczynników kkompresji dla obydwu algorytmów wyliczanych jako

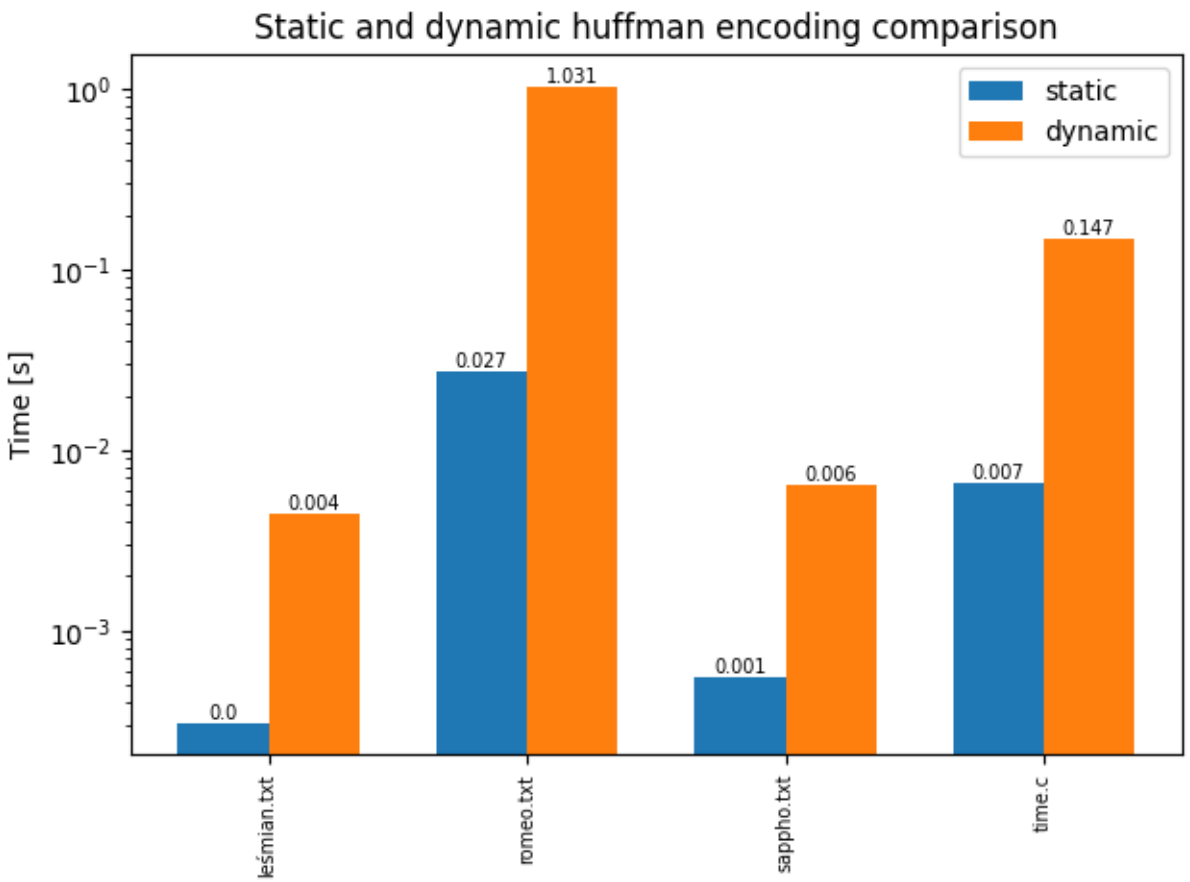
$$\frac{rozmiar\ pliku\ skompresowanego}{rozmiar\ pliku\ oryginalnego} \cdot 100\%$$



4 Porównanie czasów budowy struktur (kompresji)

Poniżej zamieszczam wykresy przedstawiające czasy wykonania algorytmów. Na drugim wykresie dane zostały wygenerowane przez zamieszczony wyżej generator dla kolejno: 1e3, 1e5 oraz 1e7 znaków.

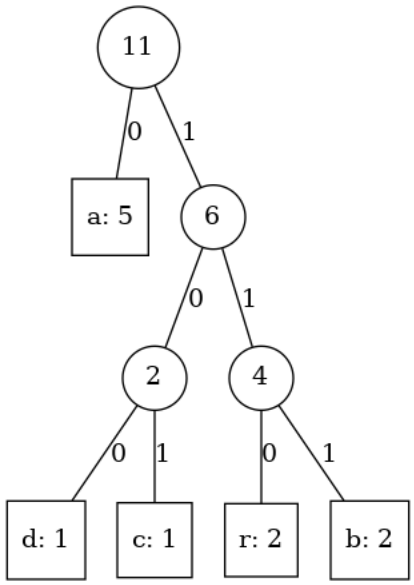

Static and dynamic huffman encoding comparison
static
dynamic
Time [s]
10^2
10^1
10^0
10^{-1}
10^{-2}
10^{-3}
0.001
0.009
0.026
0.606
2.59
56.831
uniform1e3.txt
uniform1e5.txt
uniform1e7.txt



5 Wizualizacje

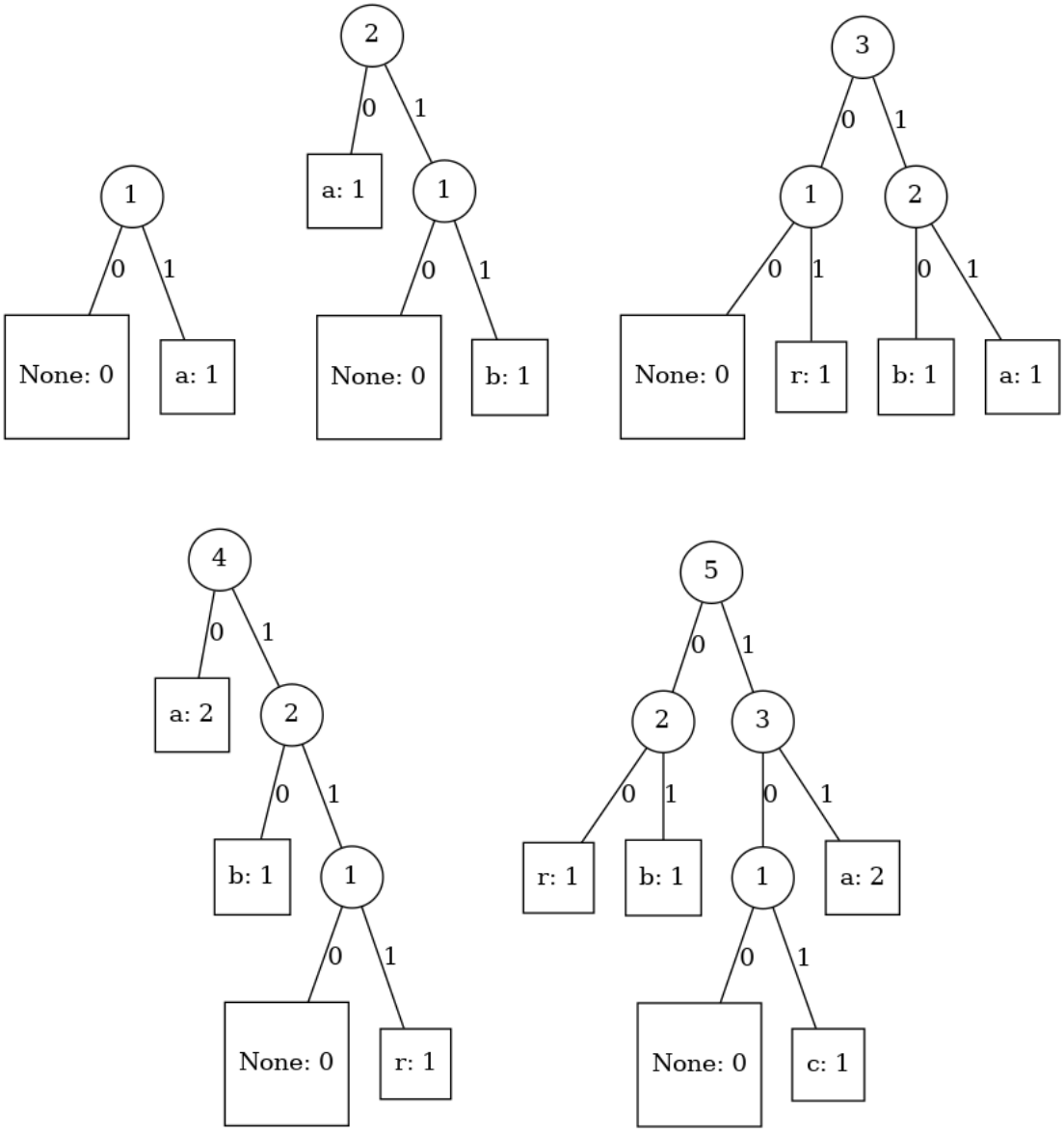
Wykonałem test dla słowa ”abracadabra” i zapisałem struktury w postaci zwizualizowanej.

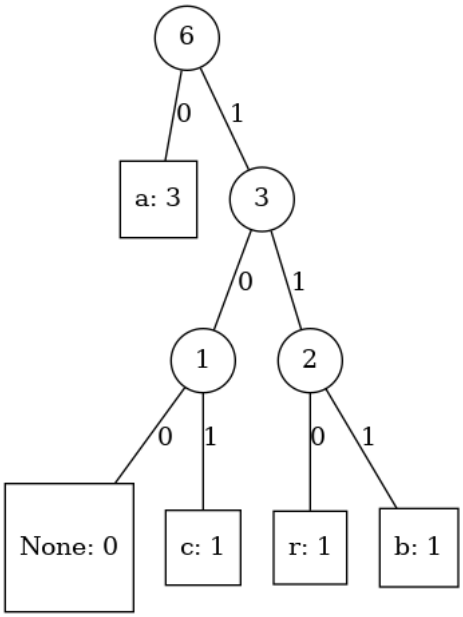
5.1 Statyczne drzewo Huffmana

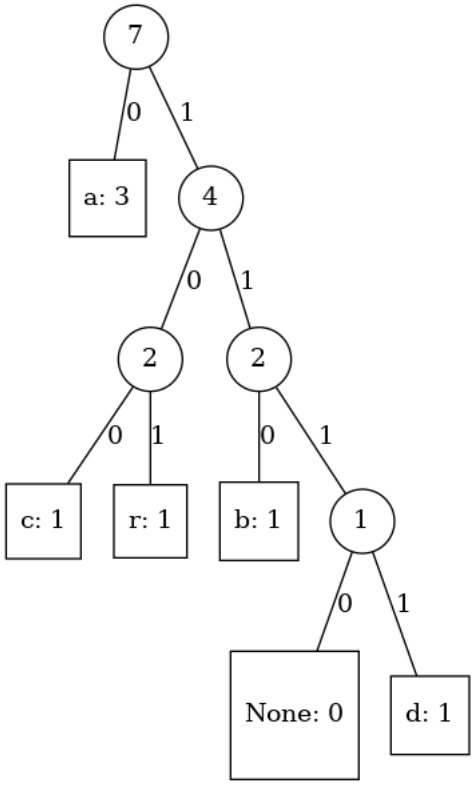


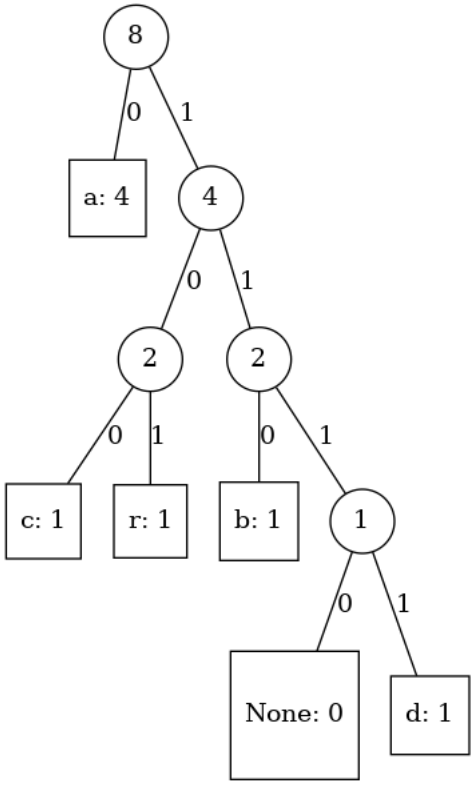
5.2 Dynamiczne drzewo Huffmana

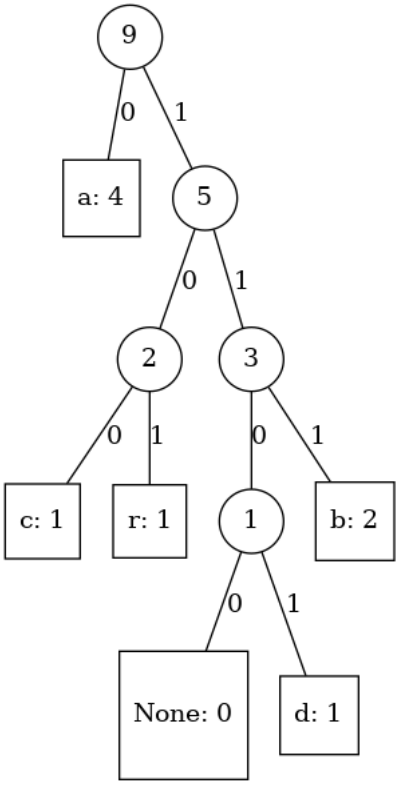
Na kolejnych grafach będzie widać, w jaki sposób powstaje drzewo Huffmana po dodaniu każdej kolejnej litery.

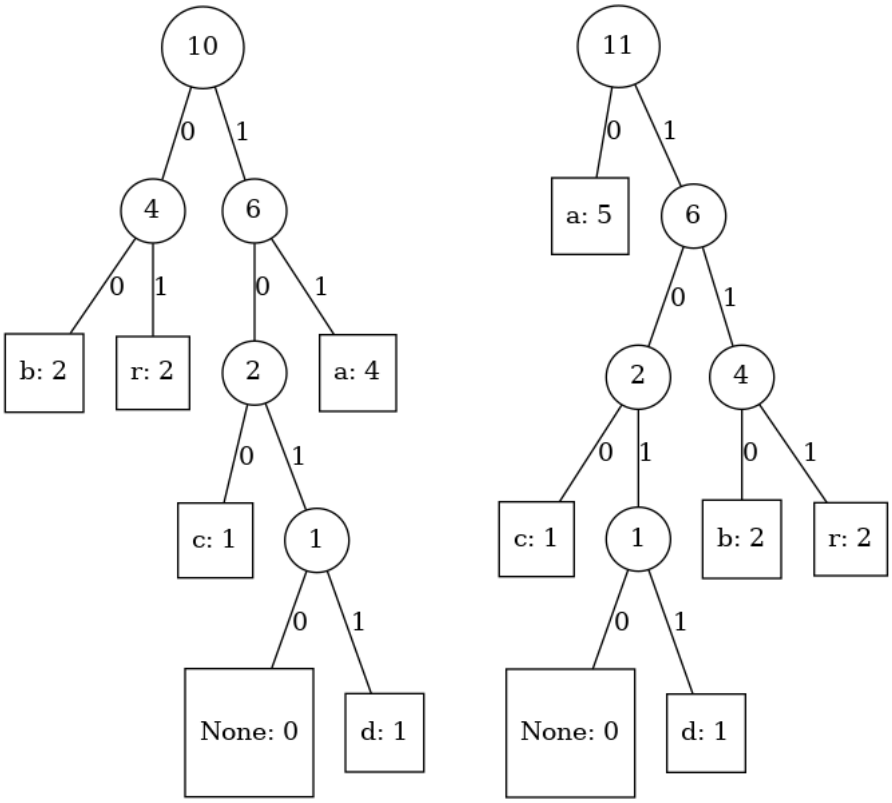



6
0
1
a: 3
3
0
1
1
2
0
1
0
1
None: 0
c: 1
r: 1
b: 1


7
0
1
a: 3
4
0
1
2
2
0
1
0
1
c: 1
r: 1
b: 1
1
0
1
None: 0
d: 1


8
0
1
a: 4
4
0
1
2
2
0
1
0
1
c: 1
r: 1
b: 1
1
0
1
None: 0
d: 1


9
0
1
a: 4
5
0
1
2
3
0
1
0
1
c: 1
r: 1
1
b: 2
0
1
None: 0
d: 1



Zauważmy, że w każdym momencie zachowany jest warunek z lematu opisanego przy implementacji klasy List.

6 Wnioski

Obydwa algorytmy dobrze spełniają swoją rolę tj. współczynnik kompresji znajduje się w granicach $70-75\%$ w zależności od danych. Kompresja za pomocą dynamicznego drzewa Huffmana trwa dłużej, jednak jest bardziej elastyczna – dodanie kolejnej litery jest możliwe bez określania od nowa kodowania dla wszystkich liter.