

# Algorytmy kompresji

Weronika Ormaniec

18.04.2020 r.

```
[1]: from static_huffman import SHuffman
      from dynamic_huffman import DHuffman
      import os
```

Implementacje obu algorytmów znajdują się w plikach *static\_huffman.py* i *dynamic\_huffman.py*.

```
[2]: def compare_files(file1, file2):
      with open(file1, "r") as f1:
          data1 = f1.read()
      with open(file2, "r") as f2:
          data2 = f2.read()
      print(f'Files equal: {data1 == data2}')

      def compression_ratio(file1, file2):
          print(f'Compression ratio: {100 * (1 - os.stat(file1).st_size/os.stat(file2).
          →st_size)}%')
```

## 1 Statyczne kodowanie Huffmana

Dla statycznego kodowania Huffmana opracowano następujący format pliku: begin

- W pierwszej kolejności do pliku kodowane jest drzewo z pomocą rekurencyjnej funkcji, która, idąc od korzenia, dla każdego wierzchołka koduje najpierw jego lewe dziecko, a następnie jego prawe dziecko. Jeśli dziecko jest węzłem wewnętrznym jest oznaczane przez 0. Jeśli dziecko jest liściem oznaczane jest przez 1, po którym na 8 bitach zakodowany jest kod ASCII znaku kodowanego przez dany liść.
- Następnie zapisywany jest zakodowany tekst.
- Na koniec otrzymany kod jest dopełniany tak, aby jego długość była podzielna przez 8 (tak, aby dało się też potem to odkodować) poprzez dodanie na początek fragmentu  $0^{8-\text{liczba\_bitow}\%8-1}1$ .

### 1kB

```
[43]: s = SHuffman("data/1000B", file=True)
```

```
[44]: %%timeit  
s.compress()
```

792  $\mu$ s  $\pm$  30.2  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 1000 loops each)

```
[45]: compression_ratio("sh_compressed", "data/1000B")
```

Compression ratio: 42.472582253240276%

```
[46]: s = SHuffman()
```

```
[47]: %%timeit  
s.decompress()
```

2.95 ms  $\pm$  696  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
[48]: compare_files("sh_decompressed.txt", "data/1000B")
```

Files equal: True

### 10kB

```
[37]: s = SHuffman("data/10000B", file=True)
```

```
[38]: %%timeit  
s.compress()
```

2.49 ms  $\pm$  112  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
[39]: compression_ratio("sh_compressed", "data/10000B")
```

Compression ratio: 46.12777833150603%

```
[40]: s = SHuffman()
```

```
[41]: %%timeit  
s.decompress()
```

19.9 ms  $\pm$  1.85 ms per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
[42]: compare_files("sh_decompressed.txt", "data/10000B")
```

Files equal: True

### 100kB

```
[31]: s = SHuffman("data/100000B", file=True)
```

```
[32]: %%timeit  
s.compress()
```

19.6 ms  $\pm$  2.37 ms per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
[33]: compression_ratio("sh_compressed", "data/100000B")
```

Compression ratio: 46.503074822338064%

```
[34]: s = SHuffman()
```

```
[35]: %%timeit  
s.decompress()
```

196 ms  $\pm$  6.83 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[36]: compare_files("sh_decompressed.txt", "data/100000B")
```

Files equal: True

### 1MB

```
[23]: s = SHuffman("data/1000000B", file=True)
```

```
[24]: %%timeit  
s.compress()
```

178 ms  $\pm$  1.53 ms per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

```
[30]: compression_ratio("sh_compressed", "data/1000000B")
```

Compression ratio: 46.55068274693511%

```
[26]: s = SHuffman()
```

```
[27]: %%timeit  
s.decompress()
```

2.73 s  $\pm$  212 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[28]: compare_files("sh_decompressed.txt", "data/1000000B")
```

Files equal: True

## 2 Dynamiczne kodowanie Huffmana

Dla dynamicznego kodowania Huffmana do pliku zapisano tylko otrzymany kod tekstu uzupełniony tak, aby liczba jego bitów była podzielna przez 8 (sposób uzupełnienia taki sam, jak dla statycznego kodowania Huffmana).

**1kB**

```
[3]: s = DHuffman("data/1000B", file=True)
```

```
[4]: %%timeit  
s.compress()
```

10 ms  $\pm$  489  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
[5]: compression_ratio("dh_compressed", "data/1000B")
```

Compression ratio: 42.871385842472584%

```
[6]: s = DHuffman()
```

```
[7]: %%timeit  
s.decompress()
```

9.43 ms  $\pm$  226  $\mu$ s per loop (mean  $\pm$  std. dev. of 7 runs, 100 loops each)

```
[8]: compare_files("dh_decompressed.txt", "data/1000B")
```

Files equal: True

### 10kB

```
[9]: s = DHuffman("data/10000B", file=True)
```

```
[10]: %%timeit  
s.compress()
```

87.1 ms  $\pm$  1.33 ms per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

```
[11]: compression_ratio("dh_compressed", "data/10000B")
```

Compression ratio: 46.11781122296422%

```
[12]: s = DHuffman()
```

```
[13]: %%timeit  
s.decompress()
```

98.4 ms  $\pm$  13.2 ms per loop (mean  $\pm$  std. dev. of 7 runs, 10 loops each)

```
[14]: compare_files("dh_decompressed.txt", "data/10000B")
```

Files equal: True

### 100kB

```
[15]: s = DHuffman("data/100000B", file=True)
```

```
[16]: %%timeit  
s.compress()
```

855 ms  $\pm$  8.85 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[17]: compression_ratio("dh_compressed", "data/100000B")
```

Compression ratio: 46.5000847195782%

```
[18]: s = DHuffman()
```

```
[19]: %%timeit  
s.decompress()
```

875 ms  $\pm$  57.6 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[20]: compare_files("dh_decompressed.txt", "data/100000B")
```

Files equal: True

**1MB**

```
[21]: s = DHuffman("data/1000000B", file=True)
```

```
[22]: %%timeit  
s.compress()
```

8.93 s  $\pm$  240 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[23]: compression_ratio("dh_compressed", "data/1000000B")
```

Compression ratio: 46.5483903119705%

```
[24]: s = DHuffman()
```

```
[25]: %%timeit  
s.decompress()
```

8.76 s  $\pm$  537 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)

```
[26]: compare_files("dh_decompressed.txt", "data/1000000B")
```

Files equal: True