Implementacja kodów Huffmana oraz rozszerzonych kodów Huffmana

Martyna Siejba

14 czerwca 2020

1 Implementacja

1.1 Struktura programu

Program zaimplementowany został w języku C++ i jego kod źródłowy znajduje się w pięciu plikach:

1. `huffman_node.cpp` - zawiera pomocniczą klasę `HuffmanNode` reprezentującą wierzchołki drzewa budowanego przez algorytm;
2. `huffman.cpp` - zawiera implementację algorytmu: w konstruktorach klas `Distribution` oraz `CombinedDistribution` obliczane są prawdopodobieństwa symboli alfabetu na podstawie wejściowego tekstu, w konstruktorze klasy `HuffmanCodes` budowane jest drzewo i obliczane kody dla znaków z alfabetu, a metoda `HuffmanCodes::encode` implementuje kodowanie tekstu wejściowego z wykorzystaniem obliczonych już kodów;
3. `encode.cpp` - implementuje funkcję `main`, która dla danej nazwy plików i parametrów kodowania wypisuje na standardowe wyjście obliczony kod Huffmana;
4. `test.cpp` - implementuje pomocniczą klasę reprezentującą parametry testów wydajnościowych;
5. `run_test.cpp` - zawiera funkcję `main` umożliwiającą uruchomienie kodowania Huffmana dla podanych parametrów oraz wypisuje na standardowe wyjście informacje o stopniu kompresji, rozmiarze alfabetu oraz czasie działania algorytmu.

1.2 Algorytm

Główna część algorytmu podzielona została na dwie części zaimplementowane w dwóch różnych klasach (i pochodnych):

- obliczanie rozkładu prawdopodobieństw znaków lub ciągów znaków zaimplementowane jest w konstruktorach klas `Distribution` oraz jej pochodnej `CombinedDistribution` - ta część algorytmu zależy od trzech parametrów: rozmiaru bloku, sposobu wyliczania prawdopodobieństw ciągu znaków oraz opcji odrzucenia ciągów znaków niewystępujących w tekście;

- pozostała część algorytmu - zbudowanie drzewa dla kanonicznych kodów Huffmana i obliczenie kodów dla znaków lub ciągu znaków oraz kodowanie tekstu implementowane są odpowiednio w konstruktorze klasy `HuffmanCodes` i metodzie `HuffmanCodes::encode`.

1.2.1 Obliczanie rozkładu prawdopodobieństwa

Dla jasności w dalszej części opisu algorytmu określmy trzy parametry determinujące jego działanie:

- k (w kodzie często reprezentowany jako `k`) - liczba naturalna określająca rozmiar bloku;
- *combine* (w kodzie reprezentowany jako `combine` lub `cmb`) - wartość Boolowska określająca sposób obliczania prawdopodobieństwa ciągów znaków: jeśli jest prawdą oznacza, że prawdopodobieństwo ciągu znaków stanowi iloczyn prawdopodobieństw pojedynczych znaków, w przeciwnym przypadku prawdopodobieństwo to obliczanie jest na podstawie liczby wystąpień danego ciągu w tekście wejściowym;
- *existing_only* (w kodzie występujące jako `existing_only` lub `eo`) - wartość Boolowska, która, jeśli jest prawdziwa, oznacza, że ciągi niewystępujące w tekście nie występują w rozkładzie.

Rozpad prawdopodobieństwa dla ciągów znaków (w tym jednoelementowych) przechowywany jest w implementacji w jako członek `distr` klasy `Distribution` w postaci listy par ciągów znaków oraz liczb naturalnych, na których porządek odpowiada porządkowi prawdopodobieństw. Dla uproszczenia dalszej części opisu nazwijmy te liczby pseudoprawdopodobieństwami. Sam algorytm obliczania rozkładu zależy od ww. parametrów.

- Jeśli $k = 1$ wykonuje się pętla, która wczytuje kolejne znaki kodowanego tekstu i zlicza wystąpienia różnych znaków z wykorzystaniem tablicy haszującej, pseudoprawdopodobieństwem znaku jest w tym wypadku liczba wystąpień znaku z tekście.
- Jeśli $k > 1$ oraz $combine = false$ algorytm działa podobnie - algorytm wczytuje kolejne znaki tekstu wejściowego, trzymając w pamięci zawsze k ostatnich znaków i aktualizując w tablicy haszującej liczbę wystąpień ciągu znaków za każdym razem, kiedy wczytany jest nowy znak. Pseudoprawdopodobieństwo ciągu stanowi liczba wystąpień ciągu znaku w tekście.
- Jeśli $k > 1$, $combine = true$ oraz $existing\_only = false$ zostaje najpierw wykonany algorytm dla przypadku, kiedy $k = 1$. Następnie wywołana zostaje funkcja rekurencyjna `CombinedLettersDistribution::gen`, która generuje wszystkie ciągi występujących w tekście znaków długości k, a ich pseudoprawdopodobieństwem jest iloczyn wystąpień w tekście pojedynczych znaków ciągu.
- Jeśli $k > 1$, $combine = true$ oraz $existing\_only = true$ podczas wykonania pętli wczytującej kolejne znaki tekstu obliczane są zarówno wystąpienia pojedynczych znaków jak i ciągów długości k. Następnie dla

wszystkich znalezionych ciągów obliczane są pseudoprawdopodobieństwa, będące iloczynem wystąpień w tekście znaków ciągu. Można zauważyć, że lista par ciągów z pseudoprawdopodobieństwami dla danego tekstu tak obliczonych zawiera się w liście obliczonej z takimi samymi k i *combine*, ale $existing\_only = false$.

1.2.2 Obliczanie kodów Huffmana

Algorytm obliczania kodów działa niezależnie od parametrów determinujących sposób obliczania rozkładu prawdopodobieństwa, jedyną daną wejściową dla ciągu jest lista par ciągów znaków oraz pseudoprawdopodobieństw. Warto zauważyć, że pseudoprawdopodobieństwa, ponieważ pseudoprawdopodobieństwa są przeskalowanymi prawdopodobieństwami ich porządek a także porządek ich sum zostaje zachowany względem faktycznych prawdopodobieństw.

Pozostała część jest implementacją standardowego algorytmu budowania kodów Huffmana. Z wykorzystaniem kolejki priorytetowej budowane są drzewa binarne, w których liściach pamiętane są ciągi znaków. Kolejka wykorzystuje komparator określający kanoniczny porządek na drzewach. Początkowo do kolejki dodane są dla każdego występującego w rozkładzie ciągu znaków z wagą będącą pseudoprawdopodobieństwem danego ciągu jednowierzchołkowe drzewa o wartości w liściu będącej ciągiem. Następnie dopóki w kolejce jest więcej niż jedno drzewo, dwa najmniejsze drzewa według porządku kanonicznego (najmniejszej wagi, potem najdawniej tworzone drzewa) są usuwane z kolejki, łączone w nowe drzewo, które następnie zostaje dodane do kolejki. Po zbudowaniu drzewa wywołana zostaje prosta funkcja rekurencyjna przechodząca wgłąb po drzewie, budująca tablicę haszującą przyporządkowującą kody ciągom znaków.

1.2.3 Kodowanie tekstu

Kodowanie tekstu wejściowego zaimplementowane jest w metodzie `HuffmanCodes::encode`. Algorytm polega na wczytywaniu po znaku kolejnych bloków tekstu wejściowego długości k oraz dodawanie odpowiadających im kodów do zakodowanego tekstu. Algorytm koduje największy prefiks tekstu wejściowego o długości podzielnej przez k.

1.3 Złożoność obliczeniowa

Złożoność obliczeniowa całości algorytmu różni się w zależności od wartości parametrów. Niech N oznacza długość tekstu wejściowego, M długość wynikowego kodu, a A_i liczbę różnych ciągów długości i występujących w tekście oraz niech $A = A_1$.

- Kiedy $k > 1$, $combine = true$, $existing\_only = false$ przygotowanie rozkładu ma złożoność $O(N + A^k)$, obliczenie kodów $O(A^k * logA^k)$, a kodowanie $O(N+M)$, co daje całkowitą złożoność $O(N+M+A^k*k*logA)$.
- W pozostałych przypadkach otrzymujemy złożoność $O(N+M+A_k+A_k*logA_k)$.

1.4 Instrukcja użytkownika

W folderze zz kodem źródłowym znajduje się plik `Makefile`, więc żeby zbudować pliki wykonywalne wystarczy wywołać komendę

```
$ make
```

. Zbudowane zostaną dwa pliki wykonywalne `encode` oraz `run_test`. Oba pliki przyjmują takie same cztery parametry

1. nazwa pliku z tekstem wejściowym,
2. liczba całkowita określająca wartość k,
3. liczba całkowita (0 lub 1) określająca wartość *combine*,
4. liczba całkowita (0 lub 1) określająca wartość *existing_only*.

Odpalenie programu `encode` z odpowiednimi parametrami wypisze na standardowe wyjście kod Huffmana dla pliku, natomiast program `run_test` wypisuje na standardowe wyjście dane o kodowaniu: stopień kompresji, liczbę ciągów, dla których zostały obliczone kody oraz długość zakodowanej części tekstu wejściowego i zakodowanego tekstu. Możliwe jest też odpalenie programu `run_code` bez argumentów - uruchomione zostaną wtedy przykładowe teksty zdefiniowane w pliku `run_code.cpp`.

1.4.1 Przykłady użycia

Budowanie plików wykonywalnych:

```
$ make
rm -f *.o code1 code2
g++ -O2     -c -o encode.o encode.cpp
g++ -O2     -c -o huffman.o huffman.cpp
g++ -O2     -c -o huffman_node.o huffman_node.cpp
g++ -O2  -o encode encode.o huffman.o huffman_node.o
g++ -O2     -c -o run_test.o run_test.cpp
g++ -O2     -c -o test.o test.cpp
g++ -O2  -o run_test run_test.o test.o huffman.o huffman_node.o
```

Użycie programu `encode`:

```
$ ./encode short 2 1 0
01100010111110111100100001010010100101010000000 1
010010011111100110101101000011100000111101010 0
```

Użycie programu `run_test`:

```
$ ./run_test king_lear 2 1 0
Test: testy/king_lear, długość bloku: 2, łączone litery: 1,
tylko istniejące ciągi: 0
Rozmiar alfabetu: 3721
Długość zakodowanego tekstu: 68768
Szacowana długość oryginału: 170425
Stopień kompresji: 0.403509
Czas wykonania: 0.028679s
================================================================
```

2 Dane testowe

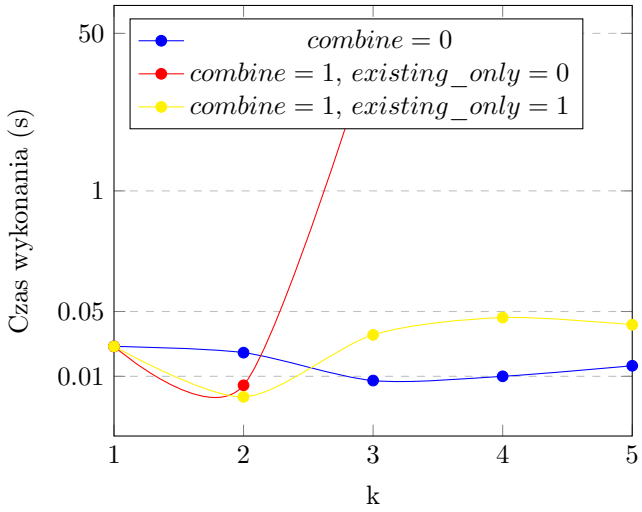
Pliki z danymi testowymi znajdują się w folderze `testy` i obejmują:

- `king_lear` - tekst w języku naturalnym,
- `king_lear_lc` - tekst w języku naturalnym pisany tylko małymi literami,
- `skos` - kod źródłowy strony internetowej,
- `huffman.cpp` - kod źródłowy programu,
- `pies.jpeg` - obrazek w postaci mapy bitowej,
- `kot.jpeg` - obrazek w postaci mapy bitowej.

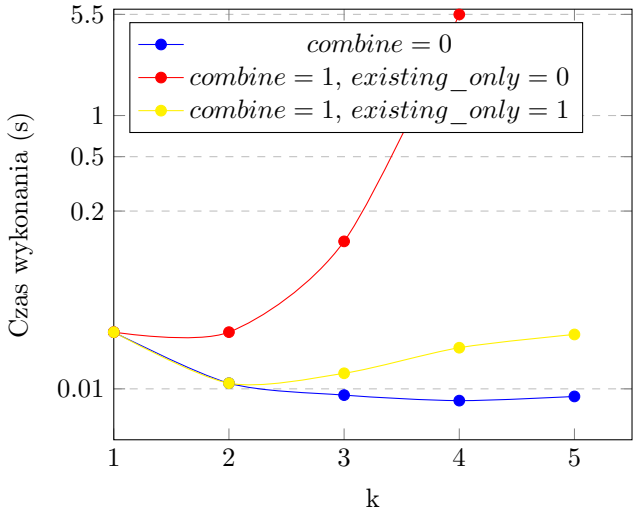
3 Wyniki

3.1 Wydajność kodowania w zależności od sposobu obliczania prawdopodobieństw ciągów znaków

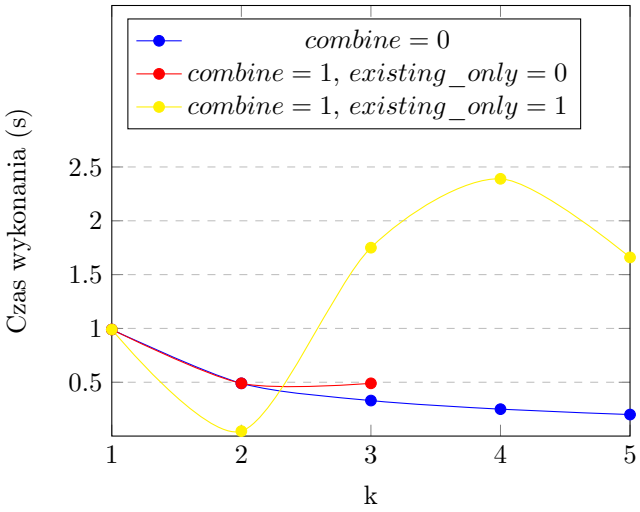
Średni (z trzech pomiarów) czas kodowania pliku `pies.jpeg` w zależności od parametrów



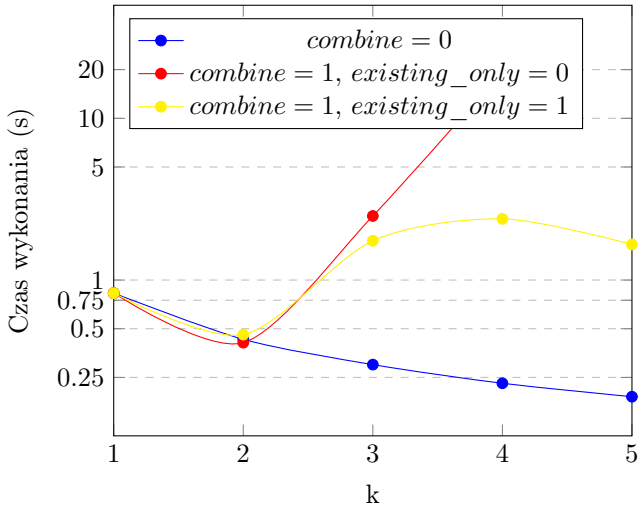
Średni (z trzech pomiarów) czas kodowania pliku `king_lear_lc` w zależności od parametrów



Stopień kompresji pliku `pies.jpeg` w zależności od parametrów



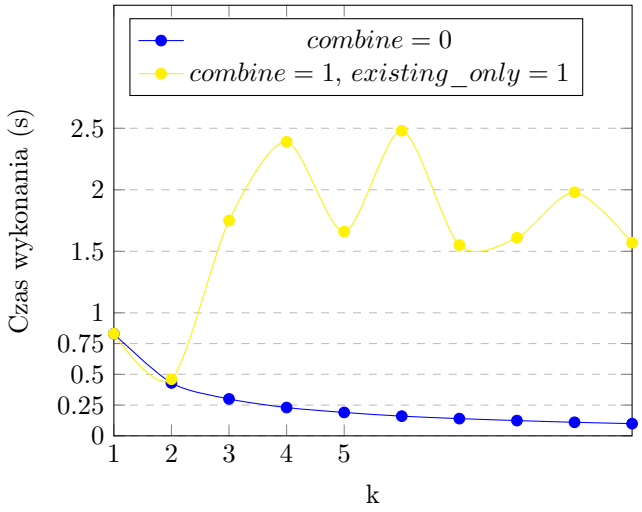
Stopień kompresji pliku `king_lear_lc` w zależności od parametrów



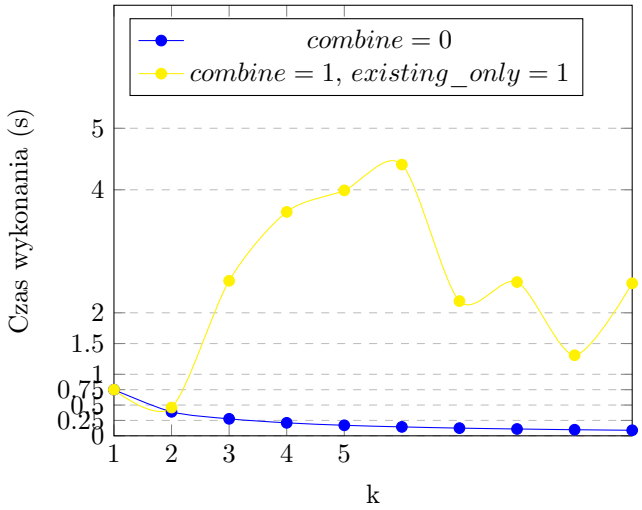
Jak sugerują wykresy zarówno dla obrazka w formie bitowej jak i tekstu w języku naturalnym najgorszym pod względem wydajności jak i stopnia kompresji sposobem obliczania rozkładu prawdopodobieństw dla bloku jest mnożenie prawdopodobieństw dla pojedynczych braków bez pomijania ciągów niewystępujących w tekście. Spadek wydajności wynika z wykładniczego wzrostu rozmiaru alfabetu względem długości bloku. Jest on więc bardziej widoczny na przykładzie pliku `pies.jpeg`, w którym rozmiar alfabetu wynosi 256 (dla pliku `king_lear_lc` jest to 39). Ponieważ obliczanie prawdopodobieństw w taki sposób jest niewydaje i z powodu ograniczeń sprzętowych na kolejnych wykresach nie będzie on reprezentowany.

3.2 Wydajność kodowania w zależności od długości bloku

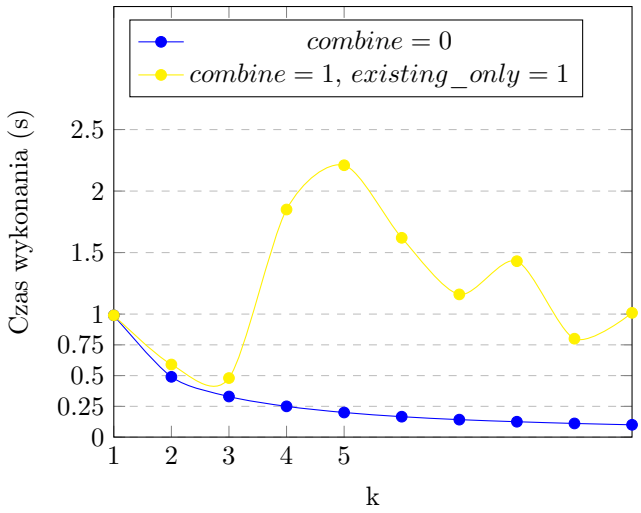
Stopień kompresji pliku `king_lear_lc` w zależności od parametrów



Stopień kompresji pliku `skos` w zależności od parametrów



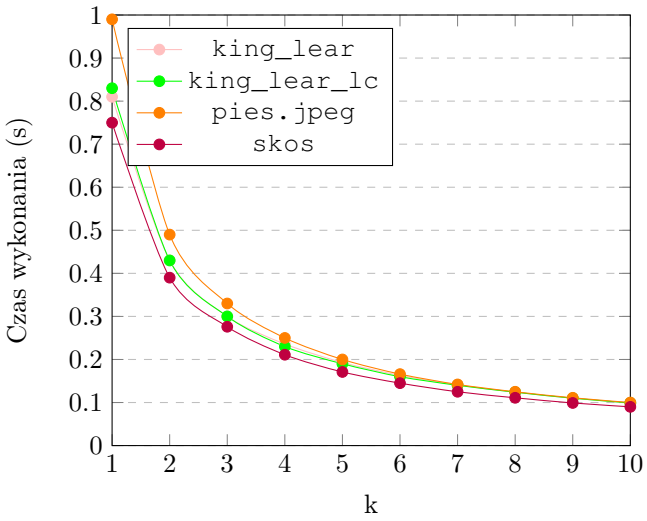
Stopień kompresji pliku `pies.jpeg` w zależności od parametrów



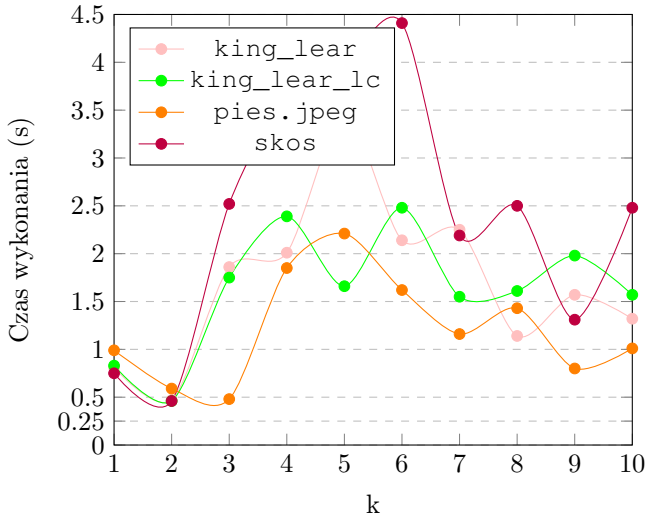
Wyniki pokazują, że pod względem stopnia kompresji najlepszym sposobem obliczania prawdopodobieństw ciągów znaków jest obliczenie ich na podstawie liczby wystąpień ciągu w tekście kodowanym.

3.3 Wydajność kodowania w zależności od typu danych

Stopień kompresji plików kiedy $combine = false$



Stopień kompresji plików kiedy $combine = true$, $existing_only = true$



Można zauważyć, że w przypadku obliczania prawdopodobieństw ciągów na podstawie liczby ich wystąpień w tekście, dla nieprzypadkowych danych stopień kompresji zachowuje się podobnie. W przypadku obliczania prawdopodobieństwa jako iloczynu prawdopodobieństw występujących w ciągu liter algorytm nie osiąga dobrych wyników.

4 Wnioski

Z wyników można wywnioskować, że najgorszym, zarówno pod względem wydajności jak i stopnia kompresji sposobem na obliczanie prawdopodobieństw ciągów znaków jest wymnażanie prawdopodobieństw pojedynczych liter. Pewnym usprawnieniem tego podejścia jest odrzucenie ciągów w ogóle niewystępujących

w kodowanym tekście - nieużywane ciągi nie wydłużają wtedy sztucznie kodów. Dodatkowo można zauważyć, że dla nieprzypadkowych danych algorytm opierający się na prawdopodobieństwach obliczanych na podstawie liczby wystąpień ciągu w tekście zachowuje się podobnie i ze wzrostem rozmiaru bloku maleje stopień kompresji.