Dokumentacja końcowa

Kompresor plików z wykorzystaniem algorytmu Huffmana

Łukasz Gumienniczuk, Paweł Rusak

11 kwietnia 2023

Spis treści

1 Cel projektu

Celem projektu było stworzenie programu, który kompresuje i dekompresuje pliki dowolnego formatu, traktując je jak pliki binarne przy pomocy algorytmu Huffmana w języku C11. Program posiada dwa tryby:

- Tryb kompresji – program kompresuje podany przez użytkownika plik i zapisuje wynik kompresji do podanego przez użytkownika pliku,
- Tryb dekompresji – program dekompresuje podany przez użytkownika skompresowany plik i zapisuje wynik dekompresji do podanego przez użytkownika pliku.

2 Flagi

Program obsługuje flagi zawarte w poniższej tabeli. Flagę lub flagi należy podać przy wywołaniu programu po pliku wejściowym i wyjściowym.
Przykładowe wywołanie programu wraz z flagami znajduję się w sekcji 4 Instrukcja użycia, a także zostaje wyświetlone po użyciu flagi `--help`.

Flaga	Opis flagi
–8	Program przeprowadzi kompresje na 8 bitach
–12	Program przeprowadzi kompresje na 12 bitach
–16	Program przeprowadzi kompresje na 16 bitach
–help	Program wyświetli instrukcje obsługi i zakończy działanie
–v	Program dodatkowo wyświetli słownik
–c	Program będzie działał w trybie kompresora
–x	Program będzie działał w trybie dekompresora

3 Kompilacja programu

Program należy skompilować w katalogu głównym projektu w środowisku Linux. Poniżej przedstawiamy możliwe komendy kompilacji:

- `make` – podstawowa kompilacja programu,
- `make test1` – komplikacja programu z testem na pliku .txt zawierającym treść „Pana Tadeusza” na 8 bitach,
- `make test2` – komplikacja programu z testem na pliku .txt zawierającym treść „Pana Tadeusza” na 12 bitach,
- `make test3` – komplikacja programu z testem na pliku .txt zawierającym treść „Pana Tadeusza” na 16 bitach,
- `make test4` – komplikacja programu z testem na pliku .jpeg przedstawiającym zdjęcie wiewiórki na 8 bitach,
- `make test5` – komplikacja programu z testem na pliku .jpeg przedstawiającym zdjęcie wiewiórki na 12 bitach,
- `make test6` – komplikacja programu z testem na pliku .jpeg przedstawiającym zdjęcie wiewiórki na 16 bitach.

4 Instrukcja użycia

Poniżej prezentujemy sposób uruchomienia skompilowanego programu:

```
./program in_file out_file <flags>
```

Przykłady:

- `./program PanTadeusz.txt compress.txt --12 --v`,
- `./program compress.txt rezultat.txt --x`,
- `./program zdjecie.jpg skompresowany_jpg.txt --8 --v --c`

Uwaga:

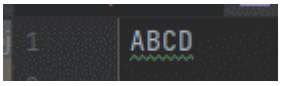
- Jeżeli program będzie wywołany bez flag `--c` oraz `--v` to automatycznie rozpozna czy plik wejściowy jest plikiem skompresowanym czy nieskompresowanym.
- Flag `--8`, `--12` oraz `--16` należy używać tylko przy kompresji.
- Brak flagi wybierającej długość sekwencji bitowej przy kompresji będzie skutkowało kompresją na 8 bitach.
- Jeżeli plik jest jednocześnie plikiem wejściowym i wyjściowym to plik ten zostanie nadpisany wynikiem działania programu.
- Pojawienie się flagi `--help` skutkuje wyświetleniem pomocy użytkownika i zakończeniem działania programu.
- Pojawienie się nieznanej flagi skutkuje przerwaniem działania programu.

5 Schemat pliku skompresowanego

1. Nagłówek – #LP#,
2. Stopień kompresji zapisany binarne,
3. Reszta1 – liczba bitów, które czytamy w ostatnim lub przedostatnim bajcie (w zależności czy reszta2 jest różna od 0),
4. Reszta2 – liczba bitów, czytana w końcowym bajcie, jeżeli reszta2 jest różna od zera. Otrzymujemy ją podczas dzielenia danych na 12 lub 16 bitów (w zależności od stopnia kompresji). W 8 nie występuje. *Nie jest kompresowana.* ,
5. Suma kontrolna – jeżeli jej *XOR* przez `DANE` jest równy 0 to znaczy, że plik jest poprawny,
6. Liczba słów w słowniku – zapisana na dwóch bajtach (gdyby liczba przekraczała 256),
7. ZNAK[8/12/16] – bity, które kodowaliśmy,
8. Długość kodu – zajmuje cały 1 bajt,
9. Kod – *Jeżeli jest to ostatni kod, to dopełniamy zerami do pełnego bajta.*

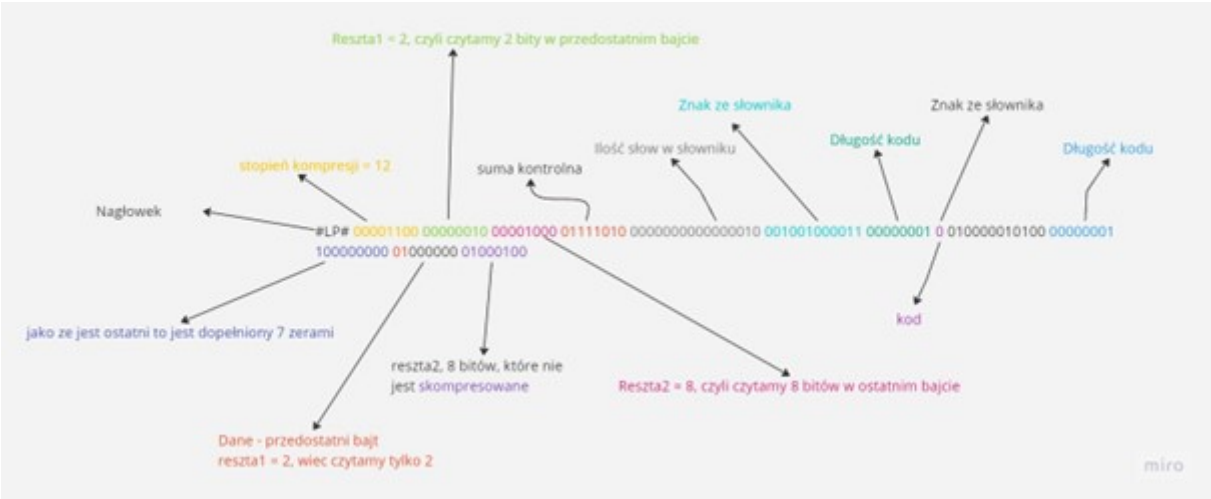
Przykład:

- Plik kompresowany:



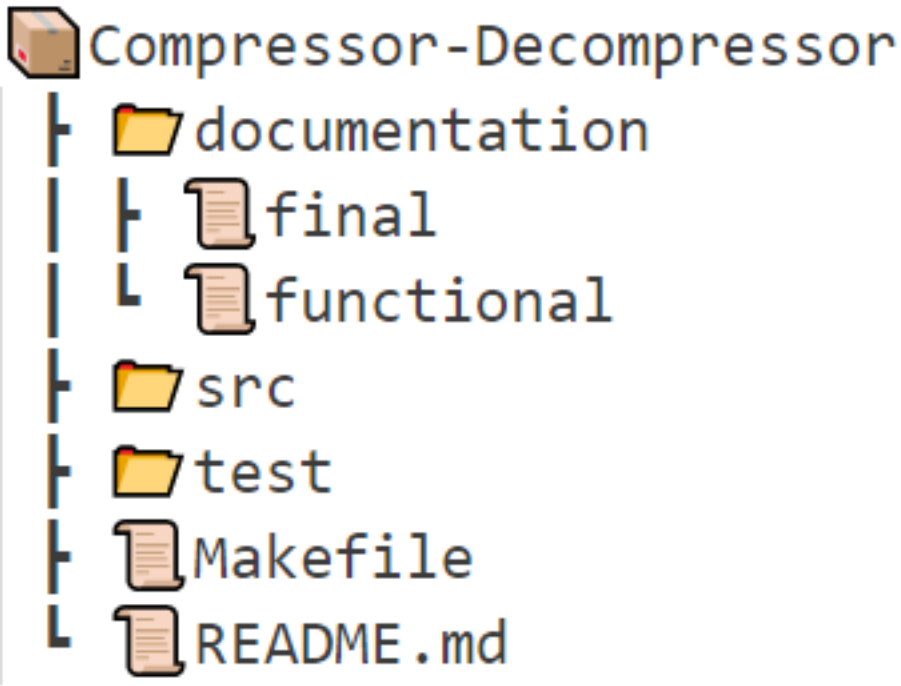
- Plik skompresowany:

Przykład pliku skompresowanego (zapis binarny):



6 Struktura programu

6.1 Struktura folderów

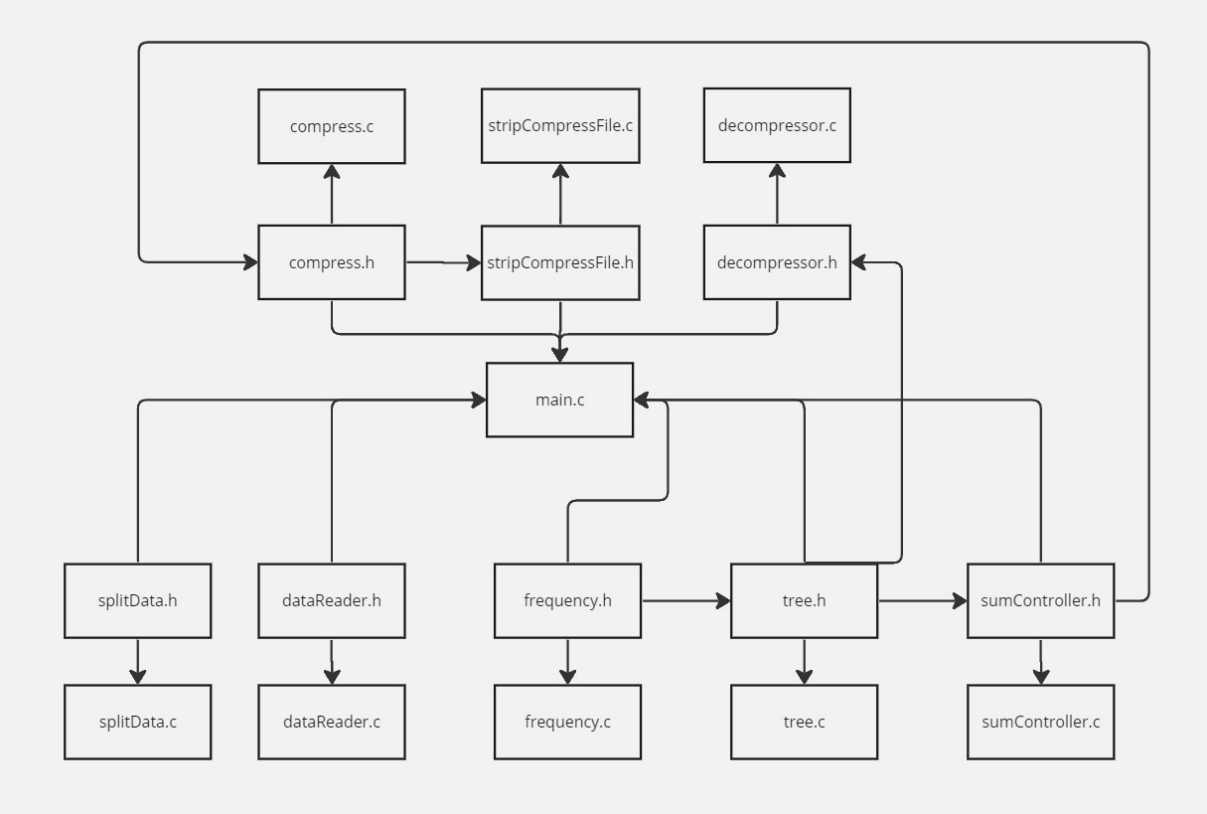


6.2 Struktura modułów

Program składa się z następujących modułów

- ogólnych:
  - `dataReader` - moduł zajmuję się pobraniem zawartości pliku,
  - `frequency` – moduł zajmuję się generowaniem statystyki występowania poszczególnych sekwencji bitowych,
  - `splitData` - moduł zajmuję się dzieleniem treści pliku na sekwencje bitowe;
- kompresujących:
  - `compress` - moduł zajmuję się zapisem wejściowego pliku za pomocą kodów binarnych,
  - `tree` – moduł zajmuję się utworzeniem binarnego kodu dla każdej sekwencji bitowej;
- dekompresujących:
  - `decompressor` – moduł zajmuję się odtworzeniem drzewa oraz dekompresją wraz z zapisem do pliku skompresowanego tekstu,
  - `stripCompressFile` - moduł zajmuję się odczytem słownika i treści pierwotnego pliku,
  - `sumController` - moduł zajmuję się odczytem sum kontrolnych ze skompresowanego pliku.

Każdy z nich składa się z pliku źródłowego *.c* i pliku nagłówkowego *.h*. Dodatkowo znajduję się również plik `main.c`, który steruje programem oraz zajmuje się obsługą flag.



Rysunek 1: Schemat modułów programu

7 Wykorzystane algorytmy

7.1 Algorytm kodowania Huffmana[1]

Algorytm kodowania Huffmana to metoda kompresji danych, która tworzy optymalne kody binarne dla symboli, tak aby minimalizować średnią długość bitową kodu. Algorytm składa się z kilku kroków:

1. Stworzenie listy symboli i ich częstości występowania w przetwarzanych danych.
2. Posortowanie listę w kolejności rosnącej według częstości występowania symboli.
3. Stworzenie drzewa binarnego, w którym liście odpowiadają symbolom, a ich głębokość odpowiada długości kodu dla danego symbolu. Węzły niebędące liśćmi reprezentują sumę częstości występowania dwóch symboli.
4. Przypisanie kodu binarnego do każdego liścia drzewa, tworząc kod Huffmana dla każdego symbolu. Kody binarne powinny być przydzielone w taki sposób, żeby sumaryczna długość kodów dla wszystkich symboli była minimalna.
5. Wykorzystanie utworzonych kodów Huffmana do zakodowania oryginalnych danych, zastępując każdy symbol jego kodem binarnym.
6. Zapisanie zakodowanych danych w pliku, wraz z dodatkowymi informacjami, takimi jak drzewo Huffmana lub tablica z kodami dla każdego symbolu, które będą potrzebne do odczytu i dekompresji.

7.2 Depth-first search (DFS)[2]

DFS jest algorytmem służącym do przechodzenia przez graf. Algorytm zaczyna swoje działanie w wierzchołku początkowym. Następnie przechodzi on do pierwszego wierzchołka, z którym jest połączony i w nim rozpoczyna swoje działanie, to jest znowu zaczyna się w pierwszym nieodwiedzonym wierzchołku. Jeżeli w danym wierzchołku wszystkie sąsiednie wierzchołki zostały już odwiedzone to algorytm wraca do pierwszego miejsca, w którym może znowu rozpocząć swoje działanie. Algorytm działa do momentu, aż wszystkie wierzchołki zostaną odwiedzone.

Algorytm został wykorzystany do wygenerowania kodów do kompresji dla poszczególnych sekwencji bitowych oraz zwolnienia pamięci zaalokowanej na odtworzenie drzewa przy dekompresji.

8 Wykorzystane struktury

Poniżej prezentujemy najważniejsze struktury, które wykorzystujemy w naszym programie.

- `struct frequency_t`

```
struct frequency_t {
    short bits;
    int frequency;
    int isTrue;
} frequency_t;
```

  Struktura jest wykorzystywana do przechowywania informacji o częstotliwości występowania danej sekwencji bitowej.

- `struct codes_t`

```
struct codes_t {
    short bits;
    char* code;
} codes_t;
```

  Struktura jest wykorzystywana do przechowywania binarnego kodu dla danej sekwencji bitowej.

- `Node_compress`

```
struct Node_compress {
    short bits;
    int frequency;
    struct Node_compress* left;
    struct Node_compress* right;
    bool been;
} Node_compress;
```

  Struktura jest wykorzystywana do stworzenia drzewa, dzięki któremu generowane są kody binarne służące do kompresji.

- `Node_decompress`

```
Node_decompress {
    short bits;
    struct Node_decompress* left;
    struct Node_decompress* right;
    bool is_leaf;
} Node_decompress;
```

  Struktura jest wykorzystywana do odtworzenia drzewa przy dekompresji.

9 Wykorzystane funkcje

Poniżej prezentujemy najważniejsze funkcje, które wykorzystujemy w naszym programie.

9.1 Funkcje ogólne

- `unsigned char* readData(char* fileName, int* size)` - funkcja zajmuję się odczytem pliku wejściowego,
- `short* splitData(unsigned char* data, int* size,int bitsToRead, unsigned char* rest, int* restBits)` - funkcja zajmuję się podziałem pliku na sekwencje 8, 12 lub 16 bitowe;

9.2 Funkcje kompresora

- `codes_t* get_codes(frequency_t* freqArray, int n)` – jest to główna funkcja generatora kodów, która wykorzystuje inne funkcje. Funkcja tworzy i zwraca kody binarne zgodnie z częstotliwością występowania sekwencji bitowych,
- `void make_tree(frequency_t* freqArray, struct Node_compress* leafs, struct Node_compress* nodes, int n, int* w)` – funkcja tworzy drzewo dla sekwencji bitowych,
- `void dfs(struct Node_compress* root, int tmp_code, struct Output_tmp* codes, int *index)` – funkcja generuje deterministyczne kody decymalne dla sekwecji bitowych,
- `void code_creator(struct Output_tmp *codes_second, struct codes_t *codes, int n)` – funkcja konwertuje kody decymalne na binarne;

9.3 Funkcje dekompresora

- `int headerCheck(char* data, int* size)` - funkcja sprawdza czy plik jest skompresowany,
- `codes_t* getDictionary(char* data, int* size, int compressionRatio, int* dictionarySize, int extraInfo)` - funkcja pobiera z pliku skompresowanego słownik i przetrzymuje go w strukturze,
- `void decoder(codes_t* codes, char* data, int n, int version, char* rest2, int restControl,FILE* out)` – jest to główna funkcja dekompresora. Funkcja odtwarza drzewo, po czym przetwarza i dekompresuje zdekompresowany plik korzystając z funkcji zapisujących do pliku.

10 Testy

10.1 Testy manualne

Nasz program przetestowaliśmy na wielu różnych plikach różnych formatów. Program w każdym przypadku działał poprawnie i plik skompresowany, a następnie zdekompresowany był identyczny.

10.2 Testy automatyczne

Program został przetestowany na pliku `.txt` oraz `.jpeg` na 8, 12 i 16 przy pomocy komend omówionych w sekcji 3. Kompilacja programu. Jest to skompresowanie i zdekompresowanie pliku, a następnie porównanie pliku pierwotnego z plikiem stworzonym przez program przy użyciu polecenia

```
@if cmp -s $(file1) $(file2);
```

W przypadku, gdy pliki są identyczne zostanie wyświetlony komunikat `files are the same`, a przeciwnym wypadku zostanie wyświetlony komunikat `files are different`.

11 Podział pracy

11.1 Moduł ogólny

- Stworzenie oraz sprawdzenie nagłówka – Łukasz Gumienniczuk,
- Odczyt pliku jako binarny – Łukasz Gumienniczuk,
- Obsługa flag – Paweł Rusak,
- Sumy kontrolne – Łukasz Gumienniczuk;

11.2 Moduł kompresora

- Stworzenie sekwencji bitowych – Łukasz Gumienniczuk,
- Wyznaczenie częstotliwości sekwencji bitowych – Łukasz Gumienniczuk,
- Stworzenie drzewa i kodów binarnych – Paweł Rusak,
- Kompresja pliku na podstawie pliku wejściowego i kodów binarnych – Łukasz Gumienniczuk;

11.3 Moduł dekompresora

- Odtworzenia słownika – Łukasz Gumienniczuk,
- Odtworzenie drzewa – Paweł Rusak,
- Dekompresja pliku na podstawie kodów binarnych i skompresowanej części pliku – Paweł Rusak,
- Zapis do pliku – Łukasz Gumienniczuk.

Literatura

[1] https://esezam.okno.pw.edu.pl/mod/book/view.php?id=70chapterid=1509, dostęp na 10.04.2023

[2] https://www.algorytm.edu.pl/grafy/przeszukiwanie-w-glab.html, dostęp na 10.04.2023