Specyfikacja Implementacyjna Kompresora

Bartosz Dybowski Jakub Klenkiewicz

11.04.2023

**Streszczenie**

Dokument przedstawia szczegółowy opis projektu polegającego na napisaniu Kompresora w języku C oraz związaną z nim implementacją. Zawarte są w nim informacje o środowisku programu, użytych metodach, podziale na moduły i sposobie działania.

Spis treści

[1. Cel projektu 4](#_Toc132145808)

[2. Środowisko 4](#_Toc132145809)

[3. Diagram modułów 4](#_Toc132145810)

[4. Opis modułów 5](#_Toc132145811)

[5. Struktura pliku wyjściowego 5](#_Toc132145812)

[5.1 Nagłówek (save\_header) 5](#_Toc132145813)

[5.2 Skompresowany kod reprezentujący oryginalny przekaz 6](#_Toc132145814)

[(save\_code) 6](#_Toc132145815)

[5.3 Cyfra kontrolna 6](#_Toc132145816)

[6. Wykorzystane algorytmy 6](#_Toc132145817)

[7. Wykorzystane struktury 7](#_Toc132145818)

[8. Wykorzystane funkcje 8](#_Toc132145819)

[8.1 Moduł file 8](#_Toc132145820)

[8.2 Moduł misc 9](#_Toc132145821)

[8.3 Moduł print 9](#_Toc132145822)

[8.4 Moduł tree 9](#_Toc132145823)

[8.5 Main 10](#_Toc132145824)

[9. Wprowadzanie zmian i przeprowadzanie testów 10](#_Toc132145825)

# Cel projektu

Celem projektu jest wykonanie kompresora w języku C. Program ten bazuje na algorytmie Huffmana. Umożliwia on kompresję w trzech poziomach od wyboru: ośmiobitowym, dwunastobitowym i szesnastobitowym oraz dekompresję.

# Środowisko

|  |  |
| --- | --- |
| System operacyjny | Windows Subsystem for Linux, Ubuntu 20.04 |
| Język programowania | C17 (ISO/IEC 9899:2018) |
| Kompilator | gcc 10.2.1 |
| Środowisko programistyczne | Visual Studio Code i Visual Studio 2019 |

# Diagram modułów

Program składa się z siedmiu modułów – globals, file, misc, print, tree, getopt. Każdy z modułów składa się z dwóch plików - nagłówkowego .h oraz pliku źródłowego .c. Oprócz nich znajduje się również plik main.c, zawierający główną funkcję sterującą programem.

Obraz zawierający diagram

Opis wygenerowany automatycznie

Rys. 1 Diagram modułów

# Opis modułów

Plik main.c odpowiada za sterowanie całym programem i wywołuje potrzebne funkcje z odpowiednich modułów. Moduły programu i ich rola:

* **Moduł file** – znajdują się w nim wszystkie funkcje operujące na plikach
* **Moduł globals** – zawiera wszystkie zmienne globalne oraz struktury
* **Moduł misc** – zawiera funkcje, w których wykonywane są czynności pomocnicze
* **Moduł print** – znajdują się w nim wszystkie funkcje wypisujące
* **Moduł tree** – posiada funkcje, których zadaniem jest tworzenie drzewa
* **Moduł getopt** – moduł potrzebny do działania getopta na Windowsie

# Struktura pliku wyjściowego

## 5.1 Nagłówek (save\_header)

Inicjały

Informacja o zaszyfrowaniu (0 - nie lub 1 - tak)

Rodzaj kompresji (8,12,16)

„Ogonek”, czyli ile bitów z ostatniego zapisanego bajta jest istotnych (są rzeczywistym kodem, który należy zdekompresować, ogonek jest różny od zera dla długości kodu huffmana niepodzielnego przez 8)

Całe drzewo w formacie :

Indeks w oryginalnej tablicy używanej przy kompresji (dla 8,12 zajmuje zawsze 2 bajty, dla 16 zajmuje zawsze 3 bajty)

Ocena czy jest to liść (1) czy węzeł (0)

Jeżeli liść :

Bity jakim odpowiada liść (dla 8 jeden bajt, dla 12,16 dwa bajty)

Jeżeli nie liść :  
Indeks lewego dziecka

Indeks prawego dziecka (zajmują tyle samo co indeksy opisane wyżej)

## 5.2 Skompresowany kod reprezentujący oryginalny przekaz

## (save\_code)

Otwiera oryginalny plik, czyta znak, znajduje kod, dodaje kod do zmiennej tymczasowej i zapisuje ją do pliku wyjściowego jako znak, gdy zapełni się w niej wszystkie 8 bitów i tak, dopóki nie skończą się znaki w oryginalnym pliku.

## 5.3 Cyfra kontrolna

Cyfra kontrolna, zapisana na samym końcu, jest zmienną początkowo ustawioną na wartość równą 0, a następnie xorowana z każdym znakiem zawierającym kod (5.2) (bez nagłówka) zapisanym do pliku wyjściowego.

# Wykorzystane algorytmy

Do stworzenia kompresora potrzebne było użycie **kodowania Huffmana.** Polega na określeniu częstości występowania każdego symbolu, a następnie na tej podstawie tworzone jest tzw. **Drzewo Huffmana**. Dzięki temu odczytujemy kod zero-jedynkowy z drzewa i przypisujemy go każdemu symbolowi.

Obraz zawierający diagram, wykres

Opis wygenerowany automatycznie

Rys. 2 Drzewo Huffmana

# Wykorzystane struktury

Wszystkie struktury wykorzystywane w programie:

* **struct node\_t:**

typedef struct {

    short char\_id; // sekwencja bitow badz ojciec

    int sum; // ilosc wystapien w tekscie

    int father\_id; // ojciec

    int left;//lewe dziecko

    int right;//prawe dziecko

    char bit; // (0,1) do tworzenia slownika

}node\_t;

* **struct keyy\_t:**

typedef struct {

    short char\_id;

    char\* code;

}keyy\_t;

* **struct branch\_t:**

typedef struct {

    int node\_id; // indeks z tablicy nodes

    int left; // lewe dziecko

    int right; // prawe dziecko

    unsigned short char\_id; // znak

}branch\_t;

* **struct min\_t:**

typedef struct {

    int val;

    int ind;

}min\_t;

* **union data:**

union data {

    short buf;

    struct {

        char A;

        char B;

    }D;

};

* **union big\_data:**

union big\_data {

    int buf;

    struct {

        char A;

        char B;

        char C;

        char D;

    }E;

};

# Wykorzystane funkcje

## Moduł file

* **unsigned char read\_byte\_from\_file(FILE\* fp)** – funkcja czytająca bajty z pliku
* **FILE\* open\_file(int argc, char\*\* argv, int argv\_num, char\* flag)** – funkcja używana do otwierania pliku
* **int valid\_initials(FILE\* fp)** – sprawdzanie poprawności inicjałów
* **unsigned short read\_two\_bytes(FILE\* fp)** – czytanie dwóch bajtów
* **int read\_three\_bytes(FILE\* fp)** – czytanie trzech bajtów
* **void write\_two\_bytes\_to\_file(FILE\* fp, short x)** – wpisywanie dwóch bajtów do pliku
* **void write\_three\_bytes\_to\_file(FILE\* fp, int x)** – wpisywanie trzech bajtów do pliku
* **char get\_control\_number(FILE\* fp)** – pobiera cyfrę kontrolną z pliku
* **void save\_initials(FILE\* fp)** – wpisywanie inicjałów do pliku
* **int get\_file\_size(FILE\* fp)** – liczy ilość znaków w pliku
* **void write\_byte\_to\_file(FILE\* fp, char x)** – wpisywanie bajtu do pliku

## Moduł misc

* **int get\_unique\_chars(int count[], int n)** - funkcja zliczająca ilość unikalnych znaków występujących w pliku wejściowym
* **char\* add\_zeros\_to\_left(char\* arr, int n)** – funkcja pomocnicza do poprawnego wypisywania wszystkich bitow bajtu
* **int get\_max\_code\_len(node\_t\* nodes, int unique\_chars)** – funkcja znajdująca najdłuższy kod
* **void reverse\_string(char\*\* arr, int strlen)** – funkcja odwracająca kod przypisany do znaku
* **void fill\_char\_arr(char\*\* arr, int n, char ch)** – wypełnienie całej tablicy znaków jednym rodzajem znaku (używane do „zerowania” tablicy znakiem ‘\0’)
* **int bin\_string\_to\_dec(char\* arr, int n)** – zamiana kodu binarnego na dziesiętny
* **int get\_bin\_len(int x)** – dlugosc liczby zapisanej dziesietnie w zapisie binarnym
* **char\* dec\_to\_bin\_string(unsigned char x, int n)** – zamiana kodu dziesiętnego na binarny
* **unsigned char get\_pass\_var(char\* pass)** – tworzenie zmiennej zxorowanej z każdą literą hasła po której następnie będzie szyfrowany plik
* **int get\_total\_code\_len(node\_t\* nodes, keyy\_t\* dict, int\* count, int unique\_chars)** – całkowita długość skompresowanego kodu huffmana
* **int get\_tail(node\_t\* nodes, keyy\_t\* dict, int\* count, int unique\_chars, int COMPRESS\_MODE)** – funkcja znajduje ogon, czyli pozostałe bity z ostatniego bajta
* **void xor\_char\_pass(unsigned char\* ch, unsigned char pass\_var)** – zxorowanie przekazanego znaku z zmienna pass\_var uzyskana z funkcji get\_pass\_var(char\* pass)

## Moduł print

* **void print\_progress(char msg[], double current, double max\_val)** – wypisuje procentowy postęp kolejnych etapów kompresji
* **void print\_decompression\_success(char control\_number)** – wypisuje informacje czy udało się zdekompresować plik
* **void print\_short\_as\_bits(short x, int bits)** – wypisywanie shorta jako bity
* **void print\_key(keyy\_t key, int max\_code\_len)** – wypisywanie kodu dla danego znaku
* **void print\_bin(char\* bin, int n)** – wypisywanie binarne
* **void print\_bin\_as\_bits(char\* bin, int n, int bits)** – wypisanie podanej ilosci znakow lancucha
* **void print\_help()** – funkcja wypisująca zawartość helpa

## Moduł tree

* **void prepare\_nodes\_arr(node\_t\*\* nodes, int count[], int n)** - wypełnianie początkowych wartości zmiennych dla każdego elementu node\_t
* **void move\_occuring\_nodes\_left(node\_t\*\* nodes, int\* right, int unique\_chars, int MAX\_VAL)** - funkcja przesuwająca wszystkie elementy tablicy, dla których znak wystąpił chociaż raz po kolei nadpisując NIEISTOTNE elementy (czyli takie, dla których znak nie wystąpił)
* **void move\_to\_child(branch\_t\* tree, branch\_t\* tmp, char\* ch\_bin, int\* bin\_ind, int\* tmp\_ind)** -
* **branch\_t\* read\_tree\_info(FILE\* fp, int\* tree\_size, int COMPRESSION\_MODE)** – odczytywanie informacji do odtworzenia drzewa
* **node\_t\* create\_tree(int\* count, int\* right, int unique\_chars, int MAX\_VAL)** – tworzenie drzewa
* **void prepare\_new\_node(node\_t\*\* nodes, int right, min\_t min1, min\_t min2)** - funkcja wypełnia wartości zmiennych nowego węzła będącego ojcem już dwóch poprzednio stworzonych
* **void reverse\_tree(branch\_t\*\* tree, int tree\_size)** – odwracanie drzewa
* **int count\_fatherless(node\_t\* nodes, int right)** - funkcja zlicza węzły, którym nie przypisano innego węzła jako ojca (node\_id = -1 przy tworzeniu nowego elementu)
* **min\_t mini1\_all(node\_t\* nodes, int right)** – funkcja znajdująca index oraz wartość spośród wszystkich węzłów
* **min\_t mini1\_fatherless(node\_t\* nodes, int right)** - funkcja znajdująca index oraz wartość najmniejszej wartości w tablicy w zakresie [0,right)
* **min\_t mini2(node\_t\* nodes, int right, min\_t min1)** - funkcja znajdująca index oraz wartość drugiej najmniejszej wartości w tablicy w zakresie [0,right)
* **void fill\_bit\_values(node\_t\* nodes, min\_t min1, min\_t min2)** – funkcja przypisująca 0 i 1 przydatna do stworzenia słownika

## Main

* **int\* count\_occurences(FILE\* fp, int fsize, int dict\_size) –** zliczanie wystąpień unikalnych ciągów bitowych (np. dla trybu 8 poszczegolnych liter)
* **keyy\_t\* create\_dict(node\_t\* nodes, int unique\_chars, int MAX\_VAL)** – funkcja tworząca słownik
* **void save\_header(FILE\* fp, node\_t\* nodes, char tail, int fsize, int right, int COMPRESSION\_MODE)** – funkcja tworząca nagłówek w pliku wyjściowym
* **void save\_code(FILE\* in, FILE\* out, keyy\_t\* dict, int fsize, int unique\_chars, int COMPRESSION\_MODE)** – funkcja zapisująca skompresowany kod huffmana do pliku wyjsciowego
* **void compress(FILE\* in, FILE\* out, int COMPRESSION\_MODE, char\* pass)** – funkcja wykonująca kompresję
* **branch\_t\* read\_header(FILE\* fp, int\* tree\_size, int\* tail, unsigned char\* control\_number, int\* DECOMPRESSION\_MODE)** – funkcja czytająca nagłówek
* **void read\_code(FILE\* fp, FILE\* out, branch\_t\* tree, int tree\_size, int tail, char control\_number, int DECOMPRESSION\_MODE)** – funkcja odczytująca plik wyjściowy
* **void decompress(FILE\* in, FILE\* out)** – funkcja wykonująca dekompresję

# Wprowadzanie zmian i przeprowadzanie testów

Wykonywane zmiany były pokazywane i przekazywane między sobą w zależności od tego, kto miał jakie zadanie.

Testowanie programu przeprowadzone jest w sposób półautomatyczny. Wprowadzamy różnego rodzaju pliki i sprawdzamy poprawność działania programu. Sprawdzamy także, czy program wypisuje komunikaty o nieprawidłowościach, celowo popełniając błędy podczas użycia. Wykorzystujemy również do tego odpowiednie formuły z pliku makefile:

* **test8** – test kompresji ośmiobitowej
* **test12** – test kompresji dwunastobitowej
* **test16** – test kompresji szesnastobitowej

Każdy z tych testów wykonuje skompresowanie, a następnie zdekompresowanie podanego pliku testowego o nazwie test.txt