

WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI



Teoria Informacji i Kodowania

Sprawozdanie z laboratorium 3

Kompresja i dekompresja za pomocą kodowania Hoffmana

Osoba realizująca: Jakub Brych	Prowadzący: Dorobisz Jerzy
Grupa: WCY21IY4S1	Data ćwiczenia: 02.06.2023

Treści zadań

LAB 3

1. Rozszerzyć program z poprzednich laboratoriów o kompresję pliku na podstawie tabeli kodowej utworzonej na podstawie drzewa Huffmana. Plik nazwa.txt jest kompresowany do nazwa.kompresja
2. Utworzyć dekompresor. Plik nazwa.kompresja jest dekompresowany do nazwa.dekompresja.

L3_1

Do programu JB_L2_2 Zostały dodane trzy funkcje. Kompresja, kompresor i zapiszTabeleKodowa.

Kompresja

Jest wywoływana z poziomu main:

```
kompresja(we, nazwa);
```

Za argumenty przyjmując wskaźnik na plik wejściowy oraz nazwę pliku aby potem utworzyć skompresowany plik o tej samej nazwie ale z innym rozszerzeniem.

```
void kompresja(FILE *we, char nazwa_kompresja[100]) {
    vector<bool> boole;
    fseek(we, 0, SEEK_SET);
    int ch;
    while ((ch = fgetc(we)) != EOF) {
        for(SymbolKodowy* symbol : kolektorKodow){
            if(symbol->znak == ch){
                printf("%c", ch);
                for(int i=0; symbol->kod[i] != '\0'; i++){
                    if(symbol->kod[i]=='0'){
                        boole.push_back(false);
                    }else{
                        boole.push_back(true);
                    }
                }
            }
        }
    }

    char *rozszerzenie = "kompresja";
    zmienRozszerzenie(nazwa_kompresja, rozszerzenie);
    FILE* kompresja = fopen(nazwa_kompresja, "wb"); // Otwórz
    plik w trybie binarnym
    if (kompresja == nullptr) {
        cout << "Błąd otwarcia pliku do zapisu." << endl;
        return;
    }
}
```

```

    kompresor(kompresja, boole); // Wywołanie funkcji
    zapisującej do pliku

    fclose(kompresja); // Zamknięcie pliku
}

```

Na początku tworzony jest wektor na wartości boolean w który będziemy wpisywać zakodowany tekst. Następuje iteracja po całym pliku wejściowym. Dla każdego ze znaków w pliku iterujemy po wektorze kolektorKodow w celu znalezienia dla niego odpowiedniego kodu. Gdy znajdziemy, że symbol w kolektorze jest równy aktualnemu symbolowi pliku przechodzimy do zapisu w wektorze boole. W pętli for którą kończy warunek końca ciągu znakowego \0, wpisujemy do wektora boole true dla 1 i false dla 0.

Gdy mamy już utworzony pełny wektora z ciągiem kodowym, używam funkcji zmień rozszerzenie aby utworzyć plik nazwa.kompresja. Plik ten, wraz z wektorem boole przekazuje do funkcji kompresor.

Kompresor

```

void kompresor(FILE* wy, const vector<bool>& boole) {
    // Sprawdzenie, czy liczba bitów jest podzielna przez 8
    int dopelnien = 0;
    if (boole.size() % 8 != 0) {
        dopelnien = 8 - (boole.size() % 8);
    }

    // Zapisanie liczby nieznaczących bitów na początku pliku
    fputc(dopelnien, wy);
    printf("%d\n", dopelnien);

    // Zapisanie bitów w kolejnych bajtach
    unsigned char byte = 0;
    int bitIndex = 7;
    int counter = 0; // Licznik zapisanych bitów
    for (bool bit : boole) {
        byte |= (bit << bitIndex);
        bitIndex--;

        if (bitIndex < 0) {
            fputc(byte, wy);
            byte = 0;
            bitIndex = 7;
        }
        counter++;
    }

    // Sprawdzenie, czy zapisano wystarczającą liczbę
    // bitów
    if (counter == boole.size()) {
        // Uzupełnienie ostatniego bajtu zerami, jeśli
        // potrzeba
    }
}

```

```

        if (dopelnien > 0) {
            byte |= 0;
            fputc(byte, wy);
            printf("Ostatni bajt (uzupelnienie): %d\n",
byte);
        }
        break;
    }
}
// Wypisanie zakodowanego ciągu bitów (z dopełnieniem)
int i = 0;
for (bool bit : boole) {
    putchar(bit ? '1' : '0');
    i++;
    if (i%8==0){
        printf(" ");
    }
}

printf("\nDlugosc ciagu kodowego: %d\n", i);

// Wypisanie liczby bitów uzupełnionych zerami
printf("Bitow dopelnien: %d \n", dopelnien);
}

```

Na samym początku sprawdzam czy liczba bitów jest podzielna przez 8 w celu zbadania czy i jeśli tak to, ile bitów dopełnień jest potrzebnych, aby zamknąć ostatni bajt. Liczbę tą zapisuje na początku pliku, aby dekompresor wiedział, kiedy przestać szukać porównań. Następnie następuje umieszczanie bitów w bajcie i wpisywanie go do pliku. W pętli iterujemy po zawartości wektora boole. Utworzyłem wcześniej zmienną unsigned char byte, w której będę zapisywać bity. Na zmiennej byte wykonujemy operacje OR i przesunięcia bitowego. Pobieramy bit z boole i dokonujemy przesunięcia bitowego o indeks (na początku 7). Następnie wynik ORujemy z dotychczasową wartością byte tym samym nadpisując go (nie tracąc poprzedniej zawartości). Po operacji dekrementowany jest index, tak aby w kolejnej iteracji przesunięcie było już mniejsze o to, ile bitów już wpisaliśmy. Za każdym razem następuje sprawdzenie czy nie zapelniliśmy już całego bajtu. Gdy tak się stanie, bajt danych jest zapisywany w pliku, zerowana jest zmienna byte oraz bitIndex ponownie jest ustawiana na 7.

Z każdą iteracją inkrementujemy zmienną counter.

Jeżeli licznik counter będzie równy wielkości wektora boole, oznacza to, że wpisaliśmy już wszystkie istotne bity i musimy przeprowadzić uzupełniania. W tym celu bierzący bajt jest ORowany z 0, co powoduje jedynie dopisanie zer by zamknąć bajt.

Wynikiem jest plik o takiej samej nazwie jak wejściowy, z rozszerzeniem .kompresja.

zapiszTabeleKodowa

```
void zapiszTabeleKodowa(char nazwa[100]) {
    char *rozzsz = "tabelakodowa";
    zmienRozszerzenie(nazwa, rozzsz);
    FILE *wy;
    wy = fopen(nazwa, "w");
    for (const SymbolKodowy* symbol : kolektorKodow) {
        fprintf(wy, "%c%s", symbol->znak, symbol->kod);
    }
}
```

Funkcja zapisuje do pliku z rozszerzeniem tabelakodowa zawartość kolektora kodów w formacie np: v00b010c011a10d1100w1101e11100. Będzie ona potrzebna do dekompresji.

Tabela kodowa mogłaby być również zapisywana w pliku kompresji.

L3_2

Program jest rozszerzeniem programu L3_1 o dodanie funkcji dekompresja, dekompresor oraz pobierzTabeleKodowa.

Wpierw w mainie sprawdzany jest warunek czy plik który wprowadził użytkownik jako parametr programu jest skompresowany:

```
if (strstr(nazwaPliku, ".kompresja") != nullptr) {
    dekompresja(nazwaPliku, we);
}
```

Jeśli tak, wykonuje się funkcja dekompresja z argumentami nazwy i wskaźnikiem na plik wejściowy.

Dekompresja

```
void dekompresja(char *nazwaPliku, FILE* we) {
    char *rozzsz = "tabelakodowa";
    char *rozzsz2 = "dekompresja";
    char *plikTabeli = nazwaPliku;
    char *plikDekompresji = nazwaPliku;
    zmienRozszerzenie(plikTabeli, rozzsz);
    pobierzTabeleKodowa(plikTabeli);
    zmienRozszerzenie(plikDekompresji, rozzsz2);
    dekompresor(we, nazwaPliku);
}
```

Wpierw tworzona jest nazwa pliku tabeli zawierająca tabele kodowa. Następnie program przypisuje jej wartości do tabeli wektora kolektor kodów za pomocą funkcji pobierzTabeleKodowa.

Potem program uruchamia dekompresor z argumentem wskaźniku i nową nazwą z rozszerzeniem .dekompresja.

Dekompresor

```
void dekompresor(FILE* we, char nazwaDekompresji[100]) {
    //Bitowa zawartosc pliku wejscowego

    printf("Bitowa zawartosc pliku wejscowego:\n");
    int readByte;
    int count = 0;
    while ((readByte = fgetc(we)) != EOF) {
        bitset<8> bits(readByte);
        for (int i = 7; i >= 0; i--) {
            printf("%d", bits.test(i) ? 1 : 0);
            count++;
            if (count % 8 == 0) {
                printf(" ");
            }
        }
    }

    printf("\n");
    rewind(we);

    // Przygotowanie mapy odwzorowań
    map<string, char> odwzorowanie;
    for (SymbolKodowy *symbol: kolektorKodow) {
        string kod(symbol->kod, symbol->dlugosc);
        odwzorowanie[kod] = symbol->znak;
    }

    // Odczytanie liczby nieznaczących bitów
    int dopelnien = fgetc(we);
    printf("Odebrano %d nieznaczących bitow\n", dopelnien);

    // Przesunięcie wskaźnika na początek danych
    fseek(we, 1, SEEK_SET);

    // Odczytanie zawartości skompresowanych danych do wektora
    buffer
    vector<bool> buffer;
    int byte;
    while ((byte = fgetc(we)) != EOF) {
        for (int bitIndex = 7; bitIndex >= 0; bitIndex--) {
            bool b = (byte >> bitIndex) & 1;
            buffer.push_back(b);
        }
    }
    for(auto b: buffer){
        printf("%d", (int)b);
    }

    // Otwarcie pliku do zapisu zdekompresowanych danych
```

```

FILE *dekompresja = fopen(nazwaDekompresji, "w");
if (dekompresja == nullptr) {
    printf("Bład otwarcia pliku do zapisu.\n");
    return;
}

// Dekodowanie skompresowanej wiadomości
string biezacyKod;
char znak;
printf("Zdekodowane znaki: \n");

int counter = 0;
int licznikZnakow = 0;
int znaczaceBity = buffer.size() - dopelnien;

while (counter < znaczaceBity) {
    biezacyKod.push_back(buffer[counter] ? '1' : '0');

    if (odwzorowanie.find(biezacyKod) !=
odwzorowanie.end()) {
        znak = odwzorowanie.at(biezacyKod);
        printf("Counter: %d \nKod: %s\n", counter,
biezacyKod.c_str());

        fwrite(&znak, sizeof(char), 1, dekompresja);
        printf("Znaleziono znak: ");
        printf("%c\n", znak);
        licznikZnakow++;

        if (counter == znaczaceBity - 1) {
            break; // Przerwij pętlę po wczytaniu
ostatniego znaku
        }
        biezacyKod = "";
    }
    counter++;
}

printf("\n");

printf("Counter: %d\n", counter);
fclose(dekompresja);
fclose(we);
printf("\n");
}

```

Wpierw dla testów wypisywana jest zawartość binarna pliku wejściowego. Bitset<8> reprezentuje sekwencje o długości 8 bitów, tworząc obiekt bits z wartościami z readByte. Następnie w pętli za

pomocą `bits.test()` sprawdzane są bity na pozycji w bajcie i jeśli test zwraca `true` wypisywane jest 1 jeśli `false` to 0. Po 8 bitach printowana jest spacja.

Następnie tworzona jest mapa odwzorowanie przechowująca pary symbol-kod, iterując po wszystkich elementach kolektoraKodowego.

Następnie pobierany jest pierwszy bajt zawierający informacje o ilości bitów dopełnień. Zapisywany jest do zmiennej `dopelnien` a sam wskaźnik jest przesuwany na początek istotnych danych.

Kolejnym krokiem jest zapisanie skompresowanej zawartości do wektora `buffer`. Każdy kolejny bajt jest czytywany z pliku. Każdy bit jest pobierany i przypisywany do `bool b` (konieczne `AND 1` aby zachować tylko odczytany bit a nie liczbę całkowitą). Następnie ten bit jest umieszczany w buforze. Sam bufor na końcu jest wypisywany dla testów.

Otwierany jest plik który będzie wynikiem dekompresji, tworzone są zmienne takie jak `counter`, `licznikznakow` oraz określana jest liczba znaczących bitów.

Następnie w pętli, dopóki `counter` jest mniejszy od liczby znaczących bitów wykonuje się przeszukiwanie. Dopisywane są kolejne bity do bieżącego kodu dopóki nie zostanie znalezione odwzorowanie. Do zmiennej `biezacyKod` dopisywane są wartości z bufora z pozycji określonej przez `counter`, który zwiększa się z każdą iteracją.

Za pomocą `find` program przeszukuje mapę odwzorowanie i jeśli został znaleziony element (gdy klucz nie jest znaleziony, zwracany jest iterator wskazujący na koniec mapy), znak jest wpisywany do pliku, na ekran, sprawdzamy czy nie jest to ostatni znak i zerujemy bieżący kod.

pobierzTabeleKodowa

```
void pobierzTabeleKodowa(const char* nazwaPliku) {
    kolektorKodow.clear();

    FILE* kody = fopen(nazwaPliku, "r");
    if (kody == NULL) {
        printf("Błąd otwarcia pliku %s\n", nazwaPliku);
        return;
    }

    char linia[256];
    while (fgets(linia, sizeof(linia), kody) != NULL) {
        SymbolKodowy* symbol = new SymbolKodowy;
        int a;
        sscanf(linia, "%d %s", &a, symbol->kod);
        symbol->znak = a;
        /* if(symbol->znak==0x0D) {
            continue;
        } */
        symbol->dlugosc = static_cast<int>(strlen(symbol->kod)); // Oblicz długość kodu
        kolektorKodow.push_back(symbol);
    }

    fclose(kody);
}
```










```

// Wyświetlanie tablicy kodowej
printf("Testowe wyświetlanie pobranej z pliku tabeli
kodowej:\n");
wypiszKodowanie();
}

```

Na początku otwierany jest plik z kodami o nazwie przekazanej jako argument. W pętli odcytujemy kolejne elementy pliku. Dla każdej nowej linii tworzony jest obiekt SymbolKodowy z odczytanymi wartościami za pomocą sscanf. Odczytywanie następuje w sposób %d jako decymalna reprezentacja znaku i %s jako string kodu (kod dla znaku). Obie wartości są przypisywane jako znak i symbol kodowy utworzonego w tej samej iteracji symbolu. Na koniec symbol jest wrzucany do globalnego kolektora kodów z którego korzysta dekompresor.

Testy

Name	Date modified	Type	Size
 small3.dekompresja	19.06.2023 01:27	DEKOMPRESJA File	1 KB
 drzewo_tabela_kodowa.txt	19.06.2023 01:26	Dokument tekstowy	4 KB
 small3.kompresja	19.06.2023 01:26	KOMPRESJA File	1 KB
 small3.modelSort	19.06.2023 01:26	MODELSORT File	1 KB
 small3.tabelakodowa	19.06.2023 01:26	TABELAKODOWA ...	1 KB
 a.exe	19.06.2023 01:26	Application	209 KB
 JB_L3_2.cpp	18.06.2023 23:53	CPP File	13 KB

```
small3.dekompresja
Plik  Edytuj  Wyświetl

a1
b2c3
d4e5f6
g7h8i9j10
k11112m13n14

o15p16 r16 s17 t18 u 19

w

2

0
```

```
small3.dekompresja  small3.txt
Plik  Edytuj  Wyświetl

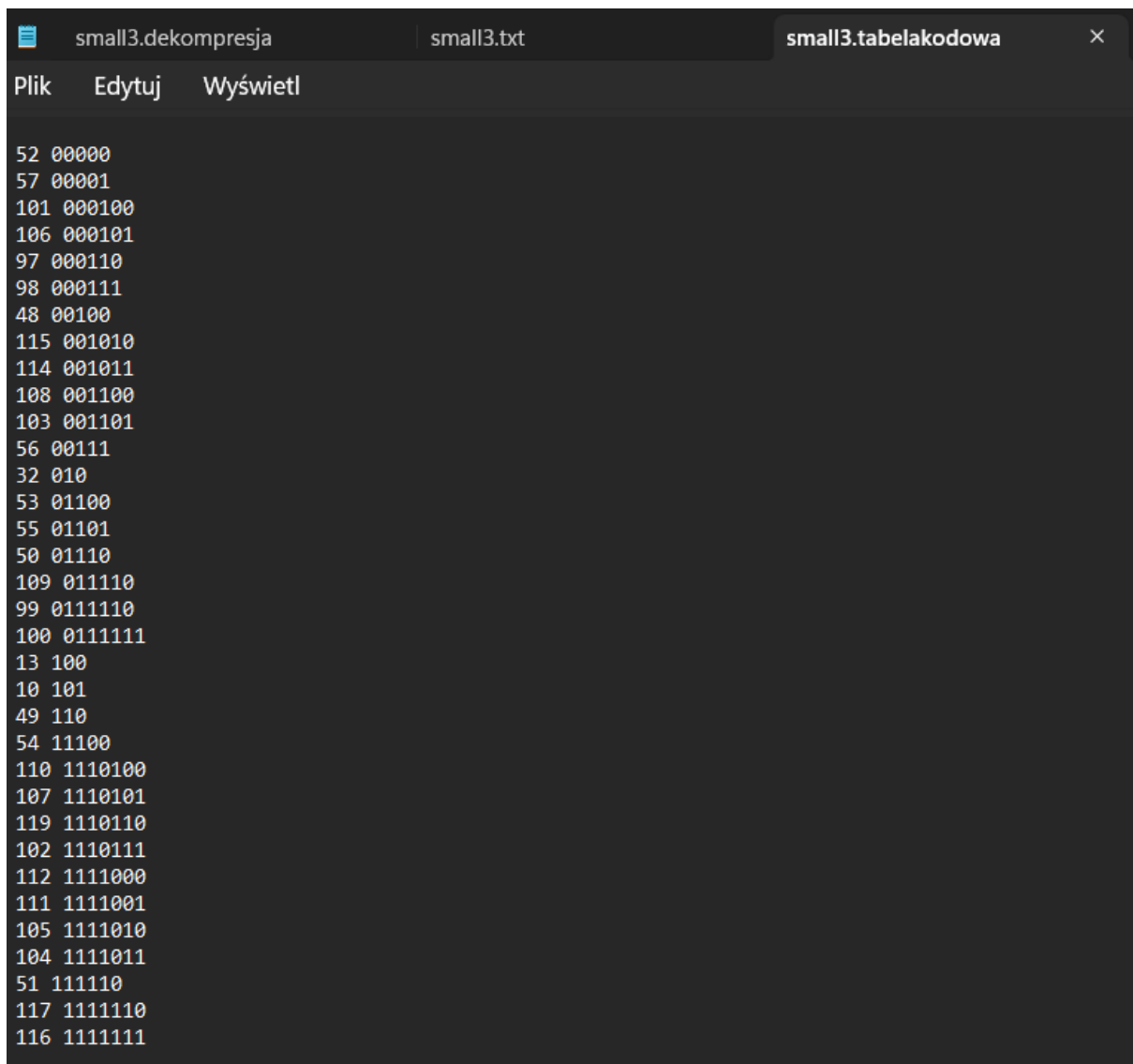
a1
b2c3
d4e5f6
g7h8i9j10
k11112m13n14

o15p16 r16 s17 t18 u 19

w

2

0
```



The image shows a code editor window with three tabs: "small3.dekompresja", "small3.txt", and "small3.tabelakodowa". The "small3.tabelakodowa" tab is active. The editor contains a list of binary strings, each preceded by a decimal index. The strings are arranged in a single column, with some lines having leading spaces. The indices range from 52 down to 116, with some gaps. The binary strings are of varying lengths, from 4 to 11 bits.

```
52 00000
57 00001
101 000100
106 000101
97 000110
98 000111
48 00100
115 001010
114 001011
108 001100
103 001101
56 00111
32 010
53 01100
55 01101
50 01110
109 011110
99 0111110
100 0111111
13 100
10 101
49 110
54 11100
110 1110100
107 1110101
119 1110110
102 1110111
112 1111000
111 1111001
105 1111010
104 1111011
51 111110
117 1111110
116 1111111
```