

WOJSKOWA AKADEMIA TECHNICZNA



Sprawozdanie z przedmiotu:

Teoria informacji i kodowania

Temat: Kodowanie Huffmana

Prowadzący: *mgr inż. Małgorzata Pisula*

Autor: Mateusz Jasiński WCY20IY2S1

19.05.2022 r.

Opis teoretyczny:

Kodowanie Huffmana to bezstratna metoda kompresji wykorzystująca algorytm zachłanny. Jego działanie ma kilka etapów:

- **Tworzenie modelu źródła:**

Tworzenie modelu źródła polega na zliczeniu częstości występowania wszystkich symboli w pliku, a następnie posortowanie ich według liczby wystąpień. W przypadku takiej samej częstości sortowanie odbywa się według kodu ASCII.

- **Tworzenie drzewa kodowego:**

Za pomocą utworzonego modelu źródła tworzone jest drzewo kodowe, którego liści są kolejne symbole występujące w pliku. Każdy element drzewa składa się z symbolu (w postaci kodu ASCII) oraz liczby wystąpień. Budowanie drzewa polega na tworzeniu węzła (rodzica), którego częstość jest sumą dwóch węzłów (potomków) bez rodzica i o najmniejszej częstości. Proces ten jest powtarzany aż zostanie jeden węzeł.

- **Tworzenie tablicy kodowej:**

Tablica kodowa przechowuje kod każdego symbolu występującego w pliku. Tworzona jest na podstawie drzewa kodowego. Kod danego symbolu to droga z korzenia drzewa do liścia reprezentującego dany symbol. Przejście do lewego potomka oznacza dopisanie 0 do kodu, a przejście do prawego potomka 1.

- **Kompresja:**

Kompresowanie polega na odczytywaniu kolejnych symboli z pliku wejściowego, odnalezieniu kodu wczytanego symbolu w tablicy kodowej, a następnie zapisaniu go do pliku wyjściowego.

- **Dekompresja:**

Dekompresja to działanie odwrotne do kompresji. Można ją wykonać przy pomocy tablicy lub drzewa kodowego.

Używając tablicy odczytujemy z pliku skompresowanego kolejne bity, aż dostaniemy ciąg pokrywający się z kodem w tablicy kodowej.

Używając drzewa odczytujemy z pliku skompresowanego kolejne bity i w zależności od wartości tego bitu przechodzimy do prawego lub lewego potomka, aż napotkamy liść – węzeł bez potomków.

Implementacja w języku C:

Program został napisany za pomocą Dev C++.

Struktury:

```
struct HuffmanNodesList {
    int symbol, freq, left_symbol, right_symbol;
    struct HuffmanNodesList *next;
};
typedef struct HuffmanNodesList HuffmanNodesList;

struct HuffmanNode {
    int symbol, freq;
    struct HuffmanNode *left, *right;
};
typedef struct HuffmanNode HuffmanNode;

struct MinHeap
{
    int size, capacity;
    struct HuffmanNode** array;
};
typedef struct MinHeap MinHeap;

struct CodeTable
{
    int symbol, size;
    unsigned short code;
};
typedef struct CodeTable CodeTable;
```

Rys.1 Struktury programu

HuffmanNodesList:

Element listy węzłów drzewa kodowego, wykorzystywane do odtworzenia drzewa przy wczytywaniu z pliku.

HuffmanNode:

Węzeł drzewa kodowego. Przechowuje symbol (w ASCII) oraz liczbę jego wystąpień, jak również wskaźnik na lewego i prawego potomka.

MinHeap:

Struktura przechowująca informacje o węzłach z najmniejszymi liczbami wystąpień:

size – obecna wielkość struktury (liczba węzłów bez rodziców).

capacity – pojemność struktury.

array – wskaźniki na węzły.

CodeTable:

Element tablicy kodowej przechowuje symbol, wielkość kodu oraz kod.

Tworzenie modelu źródła:

Przed wywołaniem funkcji tworzenia modelu źródła rezerwuje sobie pamięć dla tablicy jednoelementowej (później będę ją powiększał w miarę potrzeb). Zadeklarowana tablica jest jednym z argumentów funkcji, która tworzy model źródła, kolejne to nazwa pliku, liczba wczytywanych bitów oraz zmienna przechowująca liczbę symboli w pliku (obecnie równa 0, ale będzie ona modyfikowana w funkcji, dlatego przekazuje jej oryginał przy pomocy wskaźnika).

```
if(i == 1) {
    printf("Plik wejściowy: ");
    scanf("%s", fileNameInput);
    //strcpy(fileNameInput, "test2.txt");
    HuffmanNode *huffmanModelArray = (HuffmanNode*) malloc(sizeof(HuffmanNode));
    huffmanModelArray = CreateDataModel(huffmanModelArray, fileNameInput, readBytesLength, &readCount);
}
```

Rys.2 Wywołanie funkcji tworzenia modelu źródła

```
HuffmanNode *CreateDataModel(HuffmanNode *huffmanModelArray, char *fileName, int readBytesLength, int *readCount) {
    int i, x;
    unsigned char buffer[1];
    FILE *fileHandle;
    if((fileHandle = fopen(fileName, "rb")) == NULL) {
        printf("Nie znaleziono pliku z danymi!");
        return NULL;
    }
    while(fread(buffer, sizeof(unsigned char), readBytesLength, fileHandle)) {
        x = 0;
        for(i=0; i<*readCount; i++) {
            if(huffmanModelArray[i].symbol == buffer[0]) {
                huffmanModelArray[i].freq++;
                x = 1;
                break;
            }
        }
        if(!x) {
            (*readCount)++;
            huffmanModelArray = (HuffmanNode*) realloc(huffmanModelArray, sizeof(HuffmanNode) * (*readCount));
            huffmanModelArray[(*readCount)-1].symbol = buffer[0];
            huffmanModelArray[(*readCount)-1].freq = 1;
        }
    }
    fclose(fileHandle);
    SortHuffmanModel(huffmanModelArray, *readCount);
    return huffmanModelArray;
}
```

Rys.3 Funkcja tworzenia modelu źródła

W funkcji otwieram plik z danymi w trybie czytania „rb” – read byte. Wczytując kolejne bajty zwiększam liczbę wystąpień, jeżeli wczytany symbol już wcześniej wystąpił lub dodaje do tablicy, jeżeli symbol pojawił się po raz pierwszy. Pętla kończy się gdy funkcja fread() zwróci 0, czyli wczyta 0 bajtów z pliku. Po tym zamykam plik z danymi i sortuje tablice modelu źródła po liczbie wystąpień przy pomocy funkcji (quicksort) SortHuffmanModel(). Po sortowaniu zwracam tablice modelu źródła do funkcji main().

```

int CompareHuffmanNodes(const void *item1, const void *item2) {
    HuffmanNode *node1 = (HuffmanNode *)item1;
    HuffmanNode *node2 = (HuffmanNode *)item2;
    int compareResult = (node1->freq - node2->freq);
    if(!compareResult)
        compareResult = (node1->symbol - node2->symbol);
    return -compareResult;
}

void SortHuffmanModel(HuffmanNode *huffmanModelArray, int modelArrayLength) {
    qsort(huffmanModelArray, modelArrayLength, sizeof(HuffmanNode), CompareHuffmanNodes);
}

```

Rys.4 Funkcja sortująca model źródła

Po udanym utworzeniu tablicy z modelem źródła zostaje ona wpisana do pliku wyjściowego o nazwie podanej przez użytkownika z dopiskiem „.model”. Robi to prosta funkcja WriteModelIntoFile().

```

printf("Plik wyjsciowy: ");
scanf("%s", fileName);
//strcpy(fileName, "wynik");
strcpy(fileNameOutput, fileName);
strcat(fileNameOutput, ".model");
WriteModelIntoFile(fileNameOutput, huffmanModelArray, readCount);
printf("\nModel zrodla zostal zapisany do pliku\n");

```

Rys.5 Wywołanie funkcji wpisującej model źródła do pliku

```

void WriteModelIntoFile(char *fileName, HuffmanNode *huffmanModelArray, int readCount) {
    FILE *fileHandle;
    int i;
    if((fileHandle = fopen(fileName, "wb")) == NULL)
        printf("Nie powiodlo sie otworzenie pliku do zapisu\n");
    else {
        fprintf(fileHandle, "%d\n", readCount);
        for(i=0; i<readCount; i++)
            fprintf(fileHandle, "%d:%d\n", huffmanModelArray[i].symbol, huffmanModelArray[i].freq);
        fclose(fileHandle);
    }
}

```

Rys.6 Funkcja wpisywania modelu źródła do pliku

Funkcja wpisująca model źródła do pliku jest bardzo prosta. Otwiera ona plik w trybie zapisu „wb” – write byte, a następnie wpisuje liczbę symboli w modelu i w pętli kolejne symbole oraz ich częstość występowania.

Tworzenie drzewa kodowego:

Tworzenie drzewa kodowego jest najbardziej skomplikowaną (według mnie) operacją w tym programie. Wykonuje ją funkcja CreateTree(), której argumentami są tablica modelu źródła oraz liczba symboli.

```
HuffmanNode* CreateTree(HuffmanNode *huffmanModelArray, int readCount) {
    int i=-1;
    HuffmanNode *left, *right, *parent;
    MinHeap *minHeap = createAndBuildMinHeap(huffmanModelArray, readCount);
    while (minHeap->size != 1) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);
        parent = newNode(i--, left->freq + right->freq);
        parent->left = left;
        parent->right = right;
        insertMinHeap(minHeap, parent);
    }
    return extractMin(minHeap);
}
```

Rys.7 Funkcja tworzenia drzewa kodowego

Na początku tworzony jest „minHeap”, czyli struktura przechowująca informacje o węzłach z najmniejszymi liczbami wystąpień.

```
MinHeap* createAndBuildMinHeap(HuffmanNode *huffmanModelArray, int readCount) {
    int i;
    MinHeap *minHeap = (MinHeap*)malloc(sizeof(MinHeap));
    minHeap->size = 0;
    minHeap->capacity = readCount;
    minHeap->array = (HuffmanNode**)malloc(minHeap->capacity * sizeof(HuffmanNode*));
    for(i=0; i<readCount; i++)
        minHeap->array[i] = newNode(huffmanModelArray[i].symbol, huffmanModelArray[i].freq);
    minHeap->size = readCount;
    int n = minHeap->size - 1;
    for(i=(n-1)/2; i>=0; i--)
        minHeapify(minHeap, i);
    return minHeap;
}
```

Rys.8. Funkcja tworząca „minHeap”

Na początku deklarowana jest struktura „minHeap”, która ma wielkość 0, pojemność równą liczbie symboli i tablice wielkości równej pojemności (liczbie symboli).

Następnie tablica zapełniania jest węzłami, które przechowują wszystkie symbole występujące w modelu źródła.

```

HuffmanNode* newNode(int symbol, int freq) {
    HuffmanNode* temp = (HuffmanNode*)malloc(sizeof(HuffmanNode));
    temp->left = temp->right = NULL;
    temp->symbol = symbol;
    temp->freq = freq;
    return temp;
}

```

Rys.9 Funkcja tworzenia węzła

Funkcja tworzy węzeł, z przekazanym w argumencie symbolem i liczbą jego wystąpień. Dodatkowo ustawia potomków na NULL.

Po wypełnieniu tabeli zostaje ona posortowana za pomocą funkcji minHeapify(), która wykorzystuje sortowanie przez kopcowanie (heapsort).

```

void swapNode(HuffmanNode** a, HuffmanNode** b) {
    HuffmanNode* t = *a;
    *a = *b;
    *b = t;
}

void minHeapify(MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    if(left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;
    if(right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;
    if(smallest != idx) {
        swapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

```

Rys.10 Funkcja sortowania heapsort

Po tych wszystkich operacjach zwrócona zostaje struktura „minHeap” do funkcji CreateTree(). Teraz w pętli następuje tworzenie drzewa, powtarzane są następujące kroki do momentu, aż „minHeap” będzie miało jeden element:

- Wybierane zostają dwa węzły z najmniejszymi częstościami występowania za pomocą funkcji extractMin().

```

HuffmanNode* extractMin(MinHeap* minHeap) {
    HuffmanNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    minHeap->size--;
    minHeapify(minHeap, 0);
    return temp;
}

```

Rys.11 Funkcja wybierająca najmniejszy element z „minHeap”

- Stworzony zostaje nowy węzeł z sumą częstości występowania dwóch przed chwilą wybranych węzłów, oraz symbolem #n (n należy do naturalnych) reprezentowanym przez ujemną liczbę (np. -1 = #1, -2 = #2 itd.).
- Dwa wybrane węzły zostają przypisane do nowo powstałego węzła jako potomki.
- Nowo powstały węzeł zostaje dodany do struktury „minHeap” za pomocą funkcji insertMinHeap().

```
void insertMinHeap(MinHeap* minHeap, HuffmanNode* huffmanNode) {
    minHeap->size++;
    int i = minHeap->size - 1;
    while(i && huffmanNode->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = huffmanNode;
}
```

Rys.12 Funkcja dodająca element do tablicy w strukturze „minHeap”

Po zakończeniu działania pętli zwrócony zostaje ostatni element w tablicy struktury „minHeap”, czyli korzeń drzewa kodowego.

Utworzone drzewo zostaje wpisane do pliku z rozszerzeniem „.graf” za pomocą funkcji rekurencyjnej WriteTreeIntoFile().

```
void WriteTreeIntoFile(HuffmanNode *codeTree, int parent, FILE **fileHandle) {
    if(codeTree->symbol < 0)
        fprintf(*fileHandle, "%d:%d", -codeTree->symbol, codeTree->freq);
    else
        fprintf(*fileHandle, "%d:%d", codeTree->symbol, codeTree->freq);
    fprintf(*fileHandle, "\tchildLeft:");
    if(codeTree->left)
        if(codeTree->left->symbol < 0)
            fprintf(*fileHandle, "%d", -codeTree->left->symbol);
        else
            fprintf(*fileHandle, "%d", codeTree->left->symbol);
    else
        fprintf(*fileHandle, "-");
    fprintf(*fileHandle, "\tchildRight:");
    if(codeTree->right)
        if(codeTree->right->symbol < 0)
            fprintf(*fileHandle, "%d", -codeTree->right->symbol);
        else
            fprintf(*fileHandle, "%d", codeTree->right->symbol);
    else
        fprintf(*fileHandle, "-");
    fprintf(*fileHandle, "\tParent:");
    if(parent < 0)
        fprintf(*fileHandle, "%d\n", -parent);
    else if(parent == 0)
        fprintf(*fileHandle, "-\n");
    else
        fprintf(*fileHandle, "%d\n", codeTree->symbol);
    if(codeTree->left != NULL) WriteTreeIntoFile(codeTree->left, codeTree->symbol, fileHandle);
    if(codeTree->right != NULL) WriteTreeIntoFile(codeTree->right, codeTree->symbol, fileHandle);
}
```

Rys.13 Funkcja wpisująca drzewo kodowe do pliku

Funkcja ta wpisuje informacje o drzewie do pliku w kolejności pre-order, czyli obecnie odwiedzany węzeł, lewy węzeł, prawy węzeł. Informacje o obecnym węźle mają następujący szablon:

symbol:występowanie childLeft:symbol_lewego_potomka childRight:symbol_prawego_potomka Parent:symbol_rodzica

Np.

#8:40	childLeft:115	childRight:#7	Parent:-
115:15	childLeft:-	childRight:-	Parent:#8
#7:25	childLeft:#6	childRight:97	Parent:#8

Tworzenie tablicy kodowej:

Zgodnie z poleceniem, aby zakodować dane mieliśmy wczytać tabelę kodową z pliku, a nie użyć tej, której stworzyliśmy wcześniej, więc tworze tabelę kodową i zamiast do zmiennej wpisując ją od razu do pliku.

```
strcpy(fileNameOutput, fileName);
strcat(fileNameOutput, ".code");
if((fileHandle = fopen(fileNameOutput, "w")) == NULL)
    printf("Nie powiodło się otworzenie pliku do zapisu\n");
else {
    int *temp = (int*) malloc(sizeof(int) * readCount);
    fprintf(fileHandle, "%d\n", readCount);
    WriteCodeTableIntoFile(codeTree, temp, 0, fileHandle);
    free(temp);
    fclose(fileHandle);
    printf("tablica kodowa zostały zapisane do pliku\n");
}
```

Rys.14 Wywołanie funkcji wpisywania tabeli kodowej do pliku

Przed wywołaniem funkcji WriteCodeTableIntoFile() deklaruje tablicę temp o wielkości liczby występujących symboli (bo żaden kod na pewno nie będzie dłuższy niż liczba różnych symboli) oraz do pliku wpisując liczbę symboli, aby przy wczytywaniu wiedzieć ile danych potrzebuje odczytać.

```
void WriteCodeTableIntoFile(HuffmanNode *codeTree, int *temp, int x, FILE *fileHandle) {
    if(codeTree->left) {
        temp[x] = 0;
        WriteCodeTableIntoFile(codeTree->left, temp, x+1, fileHandle);
    }
    if(codeTree->right) {
        temp[x] = 1;
        WriteCodeTableIntoFile(codeTree->right, temp, x+1, fileHandle);
    }
    if(!(codeTree->left || codeTree->right)) {
        fprintf(fileHandle, "%d-", codeTree->symbol);
        int i;
        for(i=0; i<x; i++)
            fprintf(fileHandle, "%d", temp[i]);
        fprintf(fileHandle, "\n");
    }
}
```

Rys.15 Funkcja tworząca i wpisująca tabelę kodową do pliku

Tablica kodowa nie musi być w żaden sposób posortowana, więc wpisuje ją do pliku zaczynając od końcowych węzłów od lewej do prawej, czyli coś podobnego do kolejności

post-order z pominięciem bloków, które nie mają potomków. Na początku rekurencyjnie przechodzę na krańcowy lewy liść przy każdym przejściu dodając do tablicy „temp” 0. Gdy znajdę się na krańcu zostanie sprawdzone zostaną dwa pierwsze warunki if(), które się nie wykonają. Po tym przechodzimy do trzeciego warunku if(), który sprawdza czy węzeł w którym jesteśmy jest krańcowy, a jak jest to wypisuje jego symbol oraz zawartość tablicy temp do pliku. Po tym sprawdzeniu jako, że to rekurencja cofniemy się do wcześniejszego węzła i sprawdzony zostanie drugi warunek if(), który będzie poprawny, czyli przejdziemy w prawo i znowu zostanie wypisany potomek, znowu się cofniemy i będziemy sprawdzać kolejne węzły aż dojdziemy do ostatniego węzła po prawej stronie drzewa.

Po wpisaniu tablicy kodowej do pliku zamykam plik oraz zwalniam pamięć tablicy „temp”.

Teraz wczytuje tablice z pliku za pomocą funkcji ReadCodeTableFromFile().

```
CodeTable *ReadCodeTableFromFile(char *fileName, int *readCount) {
    FILE *fileHandle;
    CodeTable *codeTable;
    unsigned char buffer[1];
    int i, mode = 1;
    *readCount = 0;
    if((fileHandle = fopen(fileName, "rb")) == NULL) {
        printf("Nie znaleziono pliku z danymi!\n");
        return NULL;
    }
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle)) {
        if(buffer[0] == 13)
            break;
        *readCount = (*readCount) * 10 + buffer[0] - 48;
    }
    codeTable = (CodeTable*) malloc (sizeof(CodeTable)*(*readCount));
    for(i=0; i<*readCount; i++) {
        codeTable[i].symbol = 0;
        codeTable[i].code = 0;
        codeTable[i].size = 0;
    }
    i = 0;
    while(i < *readCount) {
        fread(buffer, sizeof(unsigned char), 1, fileHandle);
        if(buffer[0] == 10)
            continue;
        if(buffer[0] == 13) {
            i++;
            mode = 1;
            continue;
        }
        if(buffer[0] == '-') {
            mode = 0;
            continue;
        }
        if(mode)
            codeTable[i].symbol = codeTable[i].symbol * 10 + buffer[0] - 48;
        else {
            codeTable[i].size++;
            codeTable[i].code = codeTable[i].code << 1;
            codeTable[i].code += buffer[0] - 48;
        }
    }
    fclose(fileHandle);
    return codeTable;
}
```

Rys.16 Funkcja wczytywania tablicy kodowej z pliku

Funkcja ta wydaje się skomplikowana, ale jest w miarę prosta. Na początku otwieram plik i wczytuje kolejne znaki aż natrafię na symbol powrotu karetki (`\n`). Wiem, że wczytane symbole są cyframi, gdyż na początku tego pliku wpisałem wcześniej liczbę symboli. Po wczytaniu liczby symboli tworzę i zeruję tablice o wczytanej wielkości.

Następnie zaczynam wczytywanie tabeli kodowej do utworzonej przed chwilą tablicy. Pierwszy warunek `if()` sprawdza czy wczytany symbol jest symbolem końca linii (`\n == 10`) i go pomija. Warunek ten jest potrzebny, gdyż przy bajtowym wczytywaniu z pliku na końcu każdej linii jest znak powrotu karetki (`\n`) i znak końca linii (`\0`), a oba te znaki nie są potrzebne w tablicy kodowej.

Wczytywanie tablicy podzielone jest na 2 części: wczytywanie symbolu (`mode=1`) i wczytywanie liczby wystąpień (`mode=0`). Znak powrotu karetki symbolizuje koniec wczytywania liczby wystąpień, a znak „-” koniec wczytywania symbolu.

Pętla zakończy się, gdy zostanie wczytane tyle wierszy, ile równa jest liczba wczytana na początku pliku. Po czym plik jest zamykany, a tabela kodowa zwracana do funkcji `main()`.

Kompresja:

Funkcja kompresji pobiera kod z tablicy kodowej i wpisuje zakodowane dane do pliku.

Na początku otwieram plik wejściowy (z danymi) i plik wyjściowy (do którego będę wpisywał zakodowane dane). W pętli wczytuje kolejne symbole z pliku wejściowego, następnie szukam jego kodu w tabeli kodowej i na końcu sprawdzam:

- Jeżeli wielkość bajtu do wpisania + wielkość kodu < 8, to przesuwam bajt o tyle bitów, ile ma kod, a następnie wpisuje dopisuje go do tego bajtu.
- Jeżeli wielkość bajtu do wpisania + wielkość kodu < 16, to przesuwam bajt, o 8 - jego obecną wielkość, i dopisuje lewą stronę kodu, aby dopełnić bajt, a następnie zapisuje go do pliku. Teraz przesuwam bajt o liczbę pozostałych w kodzie bitów i je dopisuje.
- Jeżeli oba powyższe były fałszem, to przesuwam bajt, o 8 - jego obecną wielkość, i dopisuje lewą stronę kodu, aby dopełnić bajt, a następnie zapisuje go do pliku. Teraz przesuwam bajt o 8 i dopisuje środkową część kodu, a następnie wpisuje go do pliku. Na koniec przesuwam bajt o liczbę pozostałych w kodzie bitów i je dopisuje.

Po zakończeniu pętli sprawdzam, czy nie został niezapełniony bajt, jeżeli tak to dopełniam go zerami i zapisuje do pliku.

```
int WriteCompressedFile(char *fileName, char *fileNameInput, int readCount, CodeTable *codeTable) {
    FILE *input, *output;
    if((input = fopen(fileNameInput, "rb")) == NULL) {
        printf("Nie znaleziono pliku z danymi!\n");
        return 1;
    }
    if((output = fopen(fileName, "wb")) == NULL) {
        printf("Nie udało się utworzyć pliku!\n");
        return 1;
    }
    unsigned char buffer[1], temp = 0;
    int i, x, temp_size = 0;
    while(fread(buffer, sizeof(unsigned char), 1, input)) {
        for(i=0; i<readCount; i++) {
            if(codeTable[i].symbol == buffer[0]) {
                if(temp_size + codeTable[i].size < 8) {
                    temp = temp << codeTable[i].size;
                    temp += codeTable[i].code;
                    temp_size += codeTable[i].size;
                } else if(temp_size + codeTable[i].size < 16) {
                    temp = temp << 8 - temp_size;
                    temp += codeTable[i].code >> codeTable[i].size + temp_size - 8;
                    fprintf(output, "%c", temp);
                    temp = codeTable[i].code;
                    temp_size = codeTable[i].size + temp_size - 8;
                } else {
                    temp = temp << 8 - temp_size;
                    temp += codeTable[i].code >> codeTable[i].size + temp_size - 8;
                    fprintf(output, "%c", temp);
                    temp = codeTable[i].code >> codeTable[i].size + temp_size - 16;
                    fprintf(output, "%c", temp);
                    temp = codeTable[i].code;
                    temp_size = codeTable[i].size + temp_size - 16;
                }
            }
        }
    }
    if(temp_size) {
        temp = temp << 8 - temp_size;
        fprintf(output, "%c", temp);
    }
    fclose(input);
    fclose(output);
    return 0;
}
```

Rys.17 Funkcja kompresji

Wczytanie drzewa kodowego z pliku:

Do wczytania drzewa kodowego z pliku służy funkcja ReadTreeFromFile().

```
HuffmanNode* ReadTreeFromFile(FILE *fileHandle) {
    HuffmanNodesList *list = CreateList(fileHandle);
    HuffmanNode *codeTree = (HuffmanNode*) malloc(sizeof(HuffmanNode));
    codeTree->symbol = list->symbol;
    codeTree->freq = list->freq;
    codeTree->left = list->left_symbol == '-' ? NULL : SearchNode(list, list->left_symbol);
    codeTree->right = list->right_symbol == '-' ? NULL : SearchNode(list, list->right_symbol);
    while(list != NULL) {
        HuffmanNodesList *tempList = list;
        list = list->next;
        free(tempList);
    }
    return codeTree;
}
```

Rys.18 Funkcja wczytania drzewa kodowego z pliku

Na początku tworzona jest lista, która zawiera wszystkie elementy drzewa.

```
HuffmanNodesList* CreateList(FILE *fileHandle) {
    HuffmanNodesList *head = CreateListNode(fileHandle);
    HuffmanNodesList *temp = head;
    while(!feof(fileHandle)) {
        temp->next = CreateListNode(fileHandle);
        if(temp->next != NULL)
            temp = temp->next;
    }
    return head;
}
```

Rys.19 Funkcja do tworzenia listy

```

HuffmanNodesList* CreateListNode(FILE *fileHandle) {
    unsigned char buffer[1];
    HuffmanNodesList *codeTree = (HuffmanNodesList*) malloc(sizeof(HuffmanNodesList));
    codeTree->next = NULL;
    if(fread(buffer, sizeof(unsigned char), 1, fileHandle) == 0)
        return NULL;
    if(buffer[0] == '#') {
        fread(buffer, sizeof(unsigned char), 1, fileHandle);
        codeTree->symbol = -(buffer[0] - 48);
    } else
        codeTree->symbol = buffer[0] - 48;
    //symbol
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle)) {
        if(buffer[0] == ':')
            break;
        if(codeTree->symbol > 0)
            codeTree->symbol = codeTree->symbol * 10 + buffer[0] - 48;
        else
            codeTree->symbol = codeTree->symbol * 10 - (buffer[0] - 48);
    }
    //frequency
    fread(buffer, sizeof(unsigned char), 1, fileHandle);
    codeTree->freq = buffer[0] - 48;
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle)) {
        if(buffer[0] == '\t')
            break;
        codeTree->freq = codeTree->freq * 10 + buffer[0] - 48;
    }
    //left
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle))
        if(buffer[0] == ':')
            break;
    fread(buffer, sizeof(unsigned char), 1, fileHandle);
    if(buffer[0] == '-')
        codeTree->left_symbol = 0;
    else if(buffer[0] == '#') {
        fread(buffer, sizeof(unsigned char), 1, fileHandle);
        codeTree->left_symbol = -(buffer[0] - 48);
    } else
        codeTree->left_symbol = buffer[0] - 48;
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle)) {
        if(buffer[0] == '\t')
            break;
        if(codeTree->left_symbol > 0)
            codeTree->left_symbol = codeTree->left_symbol * 10 + buffer[0] - 48;
        else
            codeTree->left_symbol = codeTree->left_symbol * 10 - (buffer[0] - 48);
    }
    //right
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle))
        if(buffer[0] == ':')
            break;
    fread(buffer, sizeof(unsigned char), 1, fileHandle);
    if(buffer[0] == '-')
        codeTree->right_symbol = 0;
    else if(buffer[0] == '#') {
        fread(buffer, sizeof(unsigned char), 1, fileHandle);
        codeTree->right_symbol = -(buffer[0] - 48);
    } else
        codeTree->right_symbol = buffer[0] - 48;
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle)) {
        if(buffer[0] == '\t')
            break;
        if(codeTree->right_symbol > 0)
            codeTree->right_symbol = codeTree->right_symbol * 10 + buffer[0] - 48;
        else
            codeTree->right_symbol = codeTree->right_symbol * 10 - (buffer[0] - 48);
    }
    //parent - nie potrzebny, pomijam
    while(fread(buffer, sizeof(unsigned char), 1, fileHandle))
        if(buffer[0] == '\n')
            break;
    return codeTree;
}

```

Rys.20 Funkcja do tworzenia elementu listy

Wczytanie elementu listy podzielone jest na kilka części:

- Utworzenie nowego elementu listy,
- Wczytanie symbolu,
- Wczytanie liczby wystąpień,
- Wczytanie symbolu lewego potomka,
- Wczytanie symbolu prawego potomka,
- Pominięcie wczytania symbolu rodzica, bo nie jest potrzebny.

Po wczytaniu całego wiersza zwracany jest element listy, który jest dodawany do listy.

Po utworzeniu listy budowane jest drzewo kodowe. Budowa odbywa się w kolejności pre-order i jest wykonywana rekurencyjnie za pomocą funkcji SearchNode().

```
HuffmanNode* SearchNode(HuffmanNodesList *list, int symbol) {
    HuffmanNodesList *tempList = list;
    while(tempList != NULL) {
        if(tempList->symbol == symbol) {
            HuffmanNode *codeTree = (HuffmanNode*) malloc(sizeof(HuffmanNode));
            codeTree->symbol = tempList->symbol;
            codeTree->freq = tempList->freq;
            codeTree->left = (tempList->left_symbol == 0)? NULL : SearchNode(list, tempList->left_symbol);
            codeTree->right = (tempList->right_symbol == 0)? NULL : SearchNode(list, tempList->right_symbol);
            return codeTree;
        } else {
            tempList = tempList->next;
        }
    }
    return NULL;
}
```

Rys.21 Funkcja do wyszukiwania węzła w liście

Funkcja ta szuka w liście elementu z takim samym symbolem, który został przekazany w argumencie. Po znalezieniu odpowiedniego elementu, tworzy węzeł drzewa i sprawdza czy ma on jakichś potomków, jak ma to ponownie zostaje uruchomiona funkcja SearchNode(). Gdy drzewo zostaje stworzone zwracany zostaje korzeń do funkcji ReadTreeFromFile(). Zostają stworzone dwa korzenie, lewej strony drzewa i prawej strony drzewa.

Po pomyślnym wczytaniu drzewa z pliku zostaje z pamięci zwolniona lista, a drzewo zwrócone do funkcji main().

Dekompresja:

Funkcja dekompresji polega na wczytywaniu kolejnych bitów i przechodzenia po odpowiednich węzłach drzewa, aż do napotkania takiego bez potomków.

Na początku tworzona jest tablica, która może przechować wszystkie odkodowane znaki, czyli licznosc z korzenia drzewa. Następnie tworzona jest kopia korzenia i rozpoczyna się pętla, która zakończy się po odczytaniu odpowiedniej liczby znaków.

W pętli:

- Jeżeli wczytany bajt jest już pusty, to wczytaj kolejny,
- Jeżeli bit jest równy 1 to przejdź do prawego potomka, a do lewego w przeciwnym wypadku,
- Jeżeli węzeł, w którym jesteśmy nie ma potomków to dodaj do tablicy symbol tego węzła.

```
int WriteDecompressedFile(char *fileNameInput, char *fileNameOutput, HuffmanNode *codeTree) {
    FILE *input, *output;
    if((input = fopen(fileNameInput, "rb")) == NULL) {
        printf("Nie powiodlo sie otworzenie pliku do odczytu\n");
        return 1;
    }
    unsigned char temp, buffer[0];
    int readCount = 0, tempSize = 0;
    int *decode = (int*) malloc(sizeof(int) * codeTree->freq);
    HuffmanNode *tempTree = codeTree;
    while(readCount < codeTree->freq) {
        if(tempSize == 0) {
            fread(buffer, sizeof(unsigned char), 1, input);
            temp = buffer[0];
            tempSize = 8;
        }
        if((temp >> tempSize - 1)%2)
            tempTree = tempTree->right;
        else
            tempTree = tempTree->left;
        tempSize--;
        if(tempTree->left == NULL && tempTree->right == NULL) {
            decode[readCount] = tempTree->symbol;
            readCount++;
            tempTree = codeTree;
        }
    }
    fclose(input);
    if((output = fopen(fileNameOutput, "wb")) == NULL) {
        printf("Nie powiodlo sie otworzenie pliku do zapisu\n");
        return 1;
    }
    for(readCount = 0; readCount < codeTree->freq; readCount++)
        fprintf(output, "%c", decode[readCount]);
    free(decode);
    fclose(output);
    return 0;
}
```

Rys.22 Funkcja dekodowania pliku

Po zakończeniu działania pętli odkodowującej dane, rozpoczyna się pętla wpisująca odkodowane dane do pliku.

Po wpisaniu całej zawartości tablicy „decode” zostaje ona zwolniona z pamięci, a funkcja zwraca wartość 0 jako potwierdzenie poprawnego odkodowania danych.

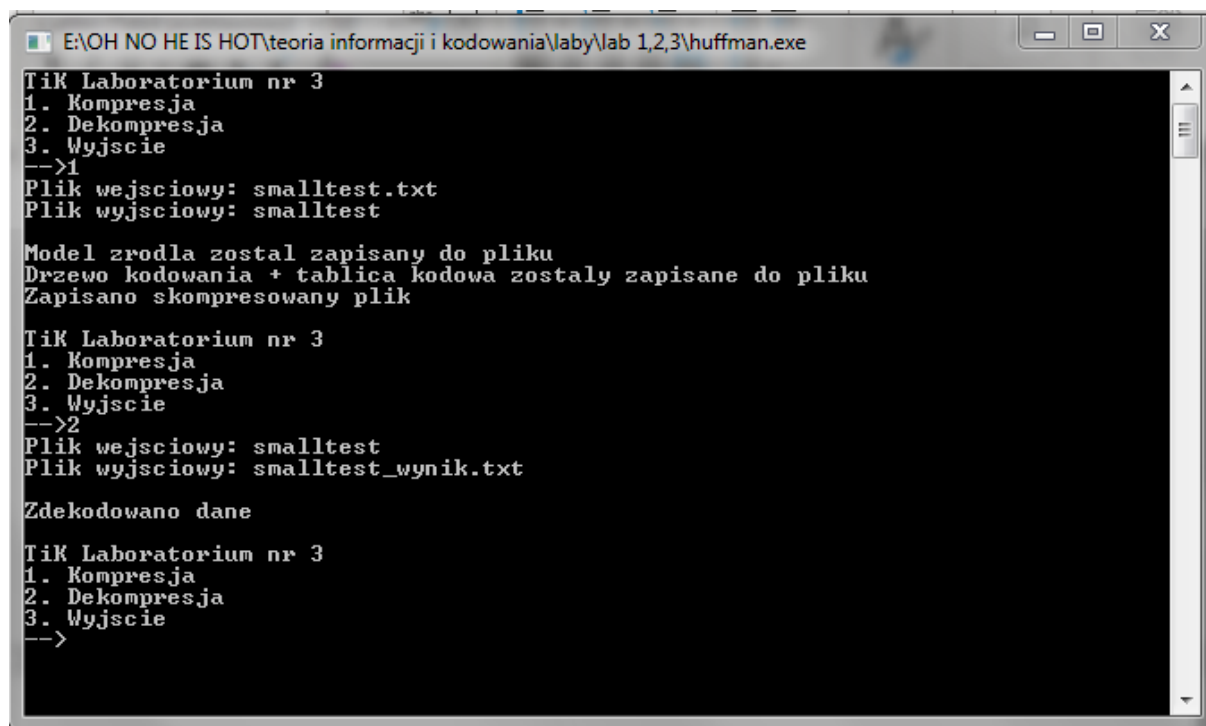
Działanie programu:

Program kompresuje i dekompresuje podany przez użytkownika plik.

Plik wprowadzony przez użytkownika musi znajdować się w tym samym folderze co program.

Do dekompresji potrzebny jest skompresowany plik, a dodatkowo plik o takiej samej nazwie z rozszerzeniem „.graf”, który zawiera drzewo kodowe wykorzystywane do dekodowania.

Działanie programu dla pliku smalltest.txt:



```
E:\OH NO HE IS HOT\teoria informacji i kodowania\lab\lab 1,2,3\huffman.exe
TiK Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->1
Plik wejściowy: smalltest.txt
Plik wyjściowy: smalltest







Model źródła został zapisany do pliku
Drzewo kodowania + tablica kodowa zostały zapisane do pliku
Zapisano skompresowany plik

TiK Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->2
Plik wejściowy: smalltest
Plik wyjściowy: smalltest_wynik.txt

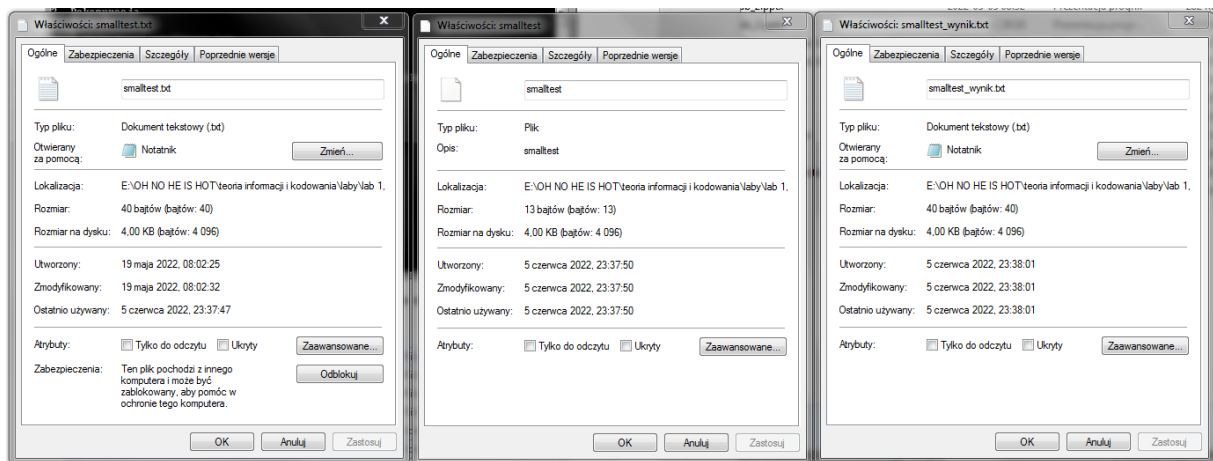
Zdekodowano dane

TiK Laboratorium nr 3
1. Kompresja
2. Dekompresja
3. Wyjście
-->
```

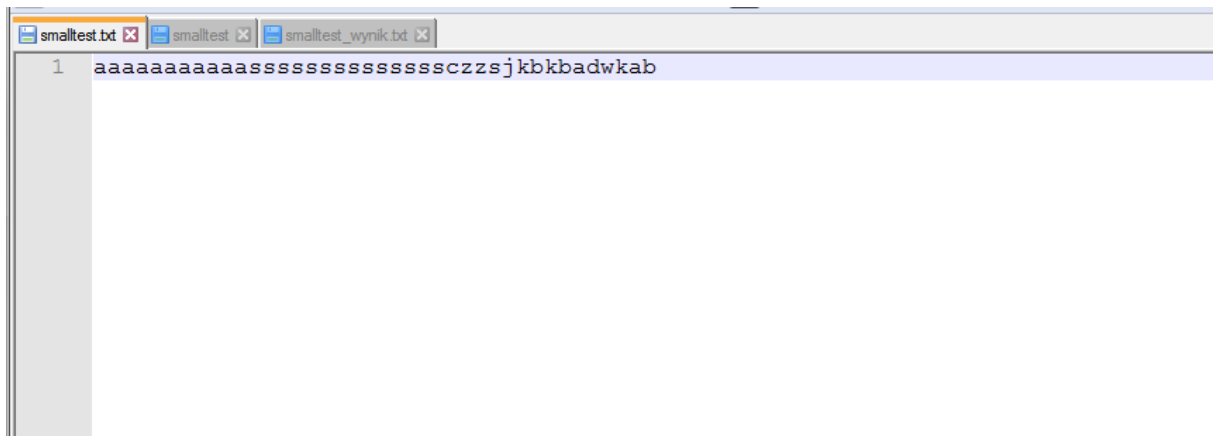
Rys.23 Konsola podczas działania programu dla smalltest.txt

	smalltest	2022-06-05 23:37	Plik	1 KB
	smalltest.code	2022-06-05 23:37	Plik CODE	1 KB
	smalltest.graf	2022-06-05 23:37	Plik GRAF	1 KB
	smalltest.model	2022-06-05 23:37	Plik MODEL	1 KB
	smalltest.txt	2022-05-19 08:02	Dokument tekstowy	1 KB
	smalltest_wynik.txt	2022-06-05 23:38	Dokument tekstowy	1 KB

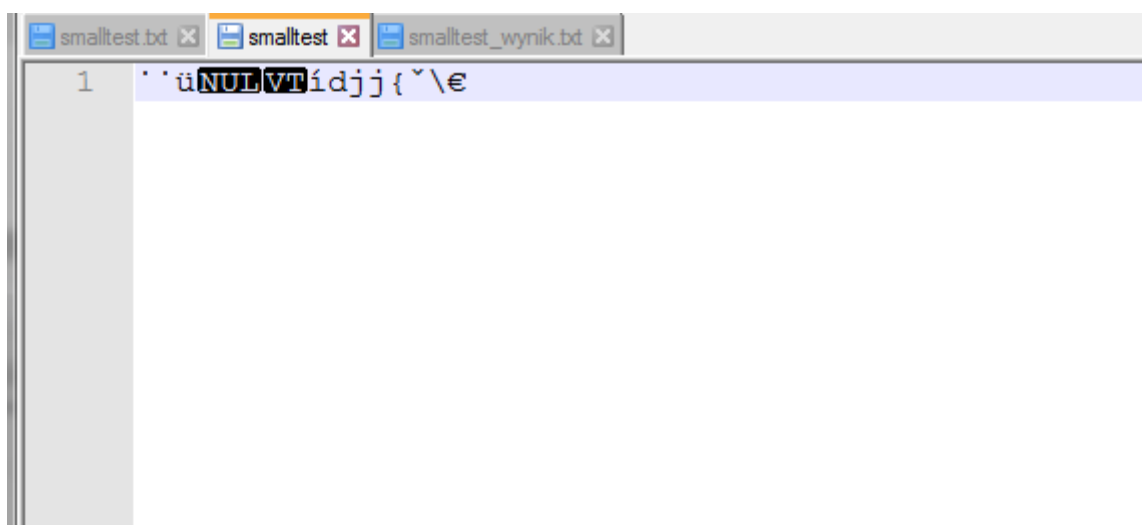
Rys.24 Pliki utworzone podczas działania programu dla smalltest.txt



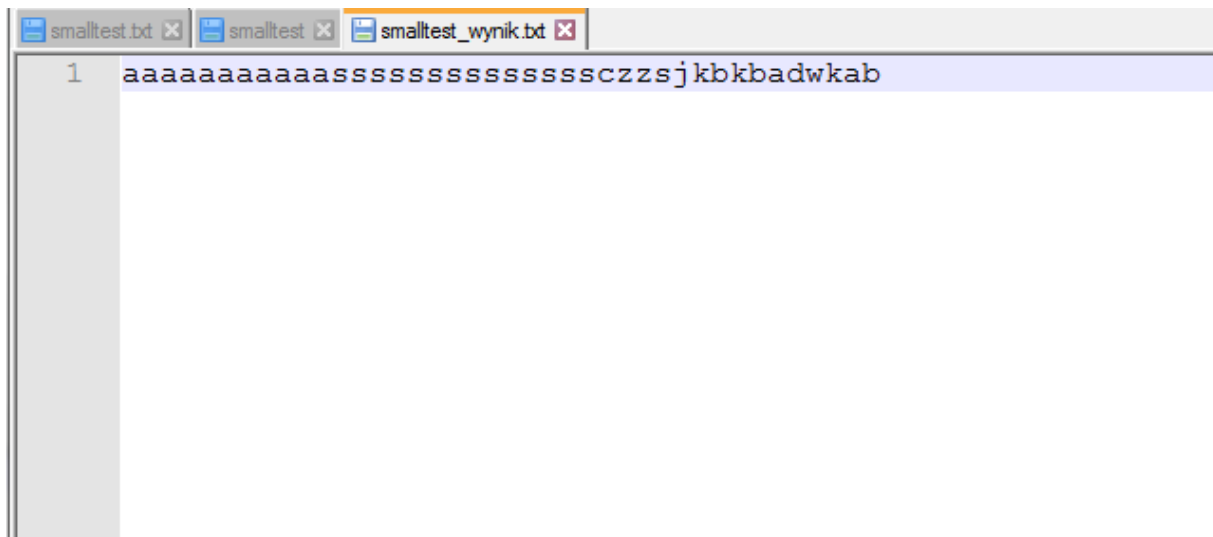
Rys.25 Wielkości utworzonych plików dla smalltest.txt



Rys.26 Zawartość pliku smalltest.txt

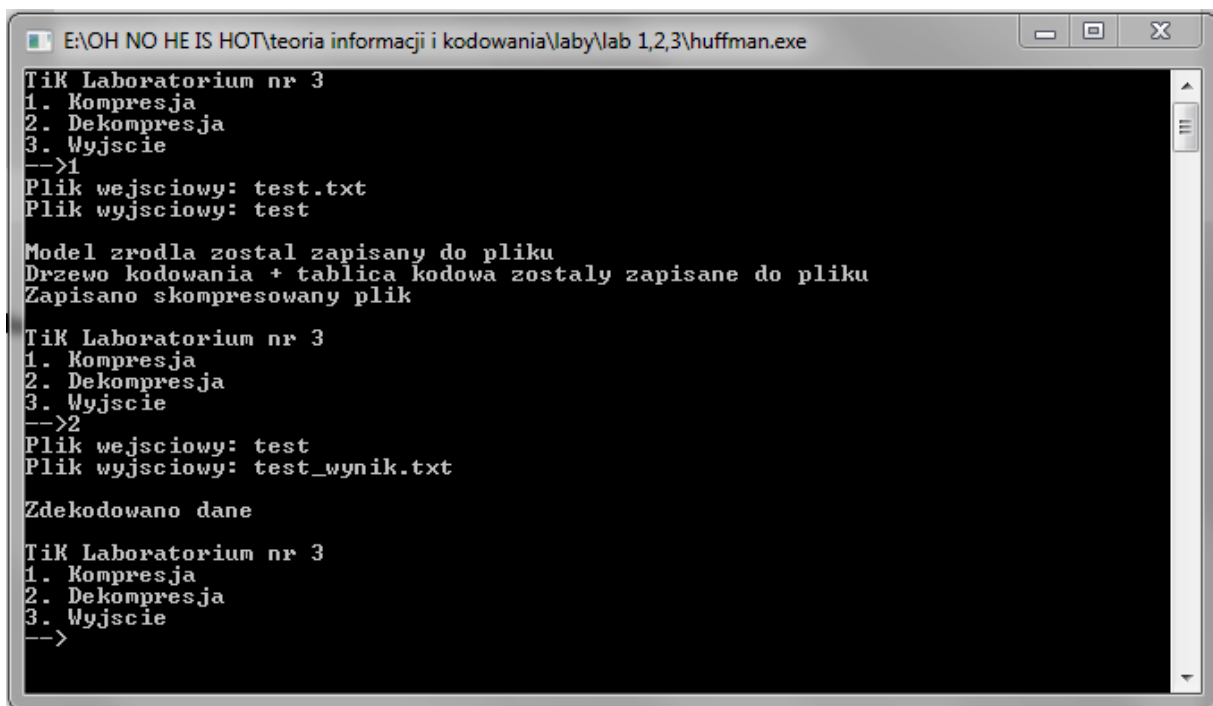


Rys.27 Zawartość skompresowanego pliku smalltest









Rys.28 Zawartość zdekompresowanego pliku smalltest_wynik.txt

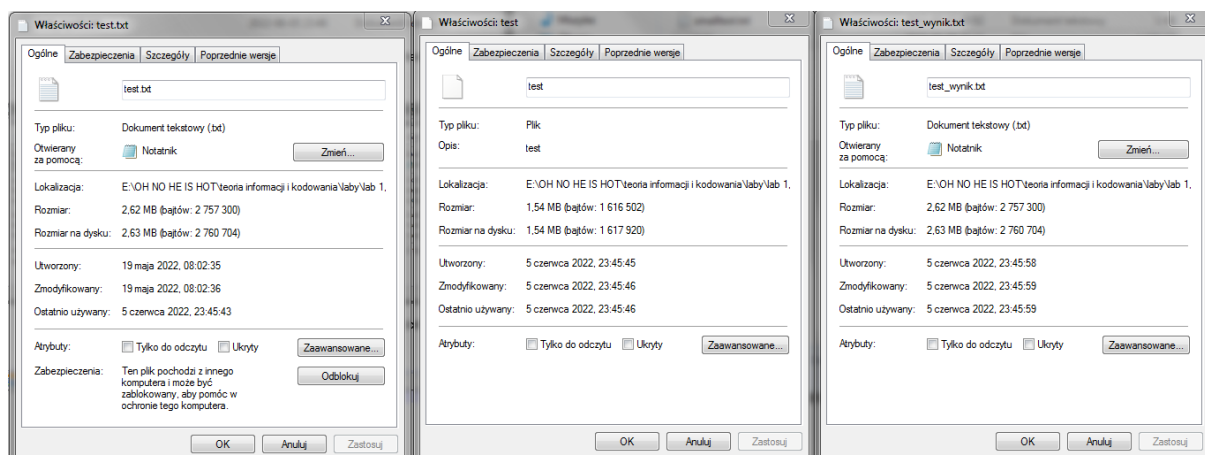
Działanie programu dla pliku test.txt:



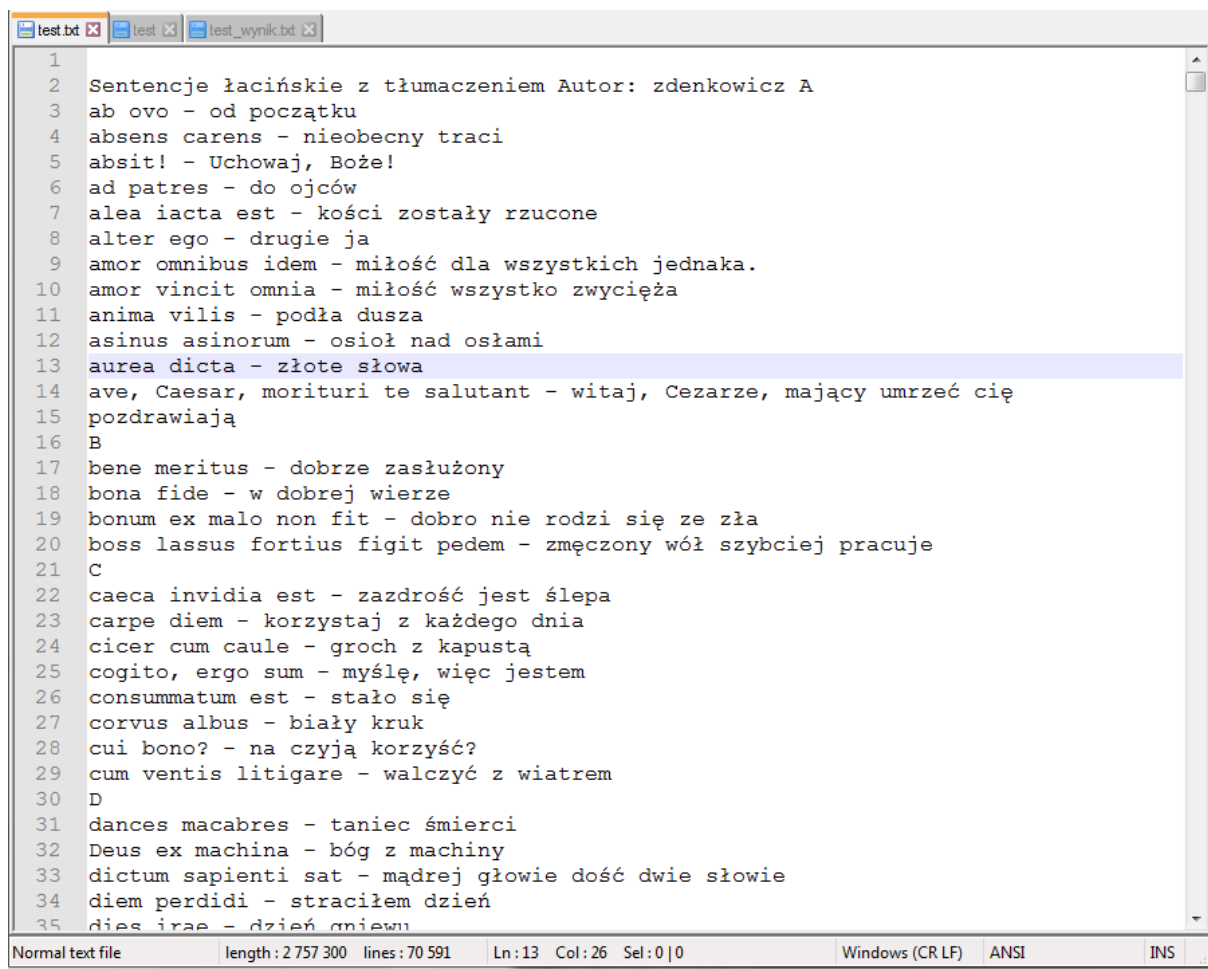
Rys.29 Konsola podczas działania programu dla test.txt

 test	2022-06-05 23:45	Plik	1 579 KB
 test.code	2022-06-05 23:45	Plik CODE	1 KB
 test.graf	2022-06-05 23:45	Plik GRAF	7 KB
 test.model	2022-06-05 23:45	Plik MODEL	1 KB
 test.txt	2022-05-19 08:02	Dokument tekstowy	2 693 KB
 test_wynik.txt	2022-06-05 23:46	Dokument tekstowy	2 693 KB

Rys.30 Pliki utworzone podczas działania programu dla test.txt



Rys.31 Wielkości utworzonych plików dla test.txt



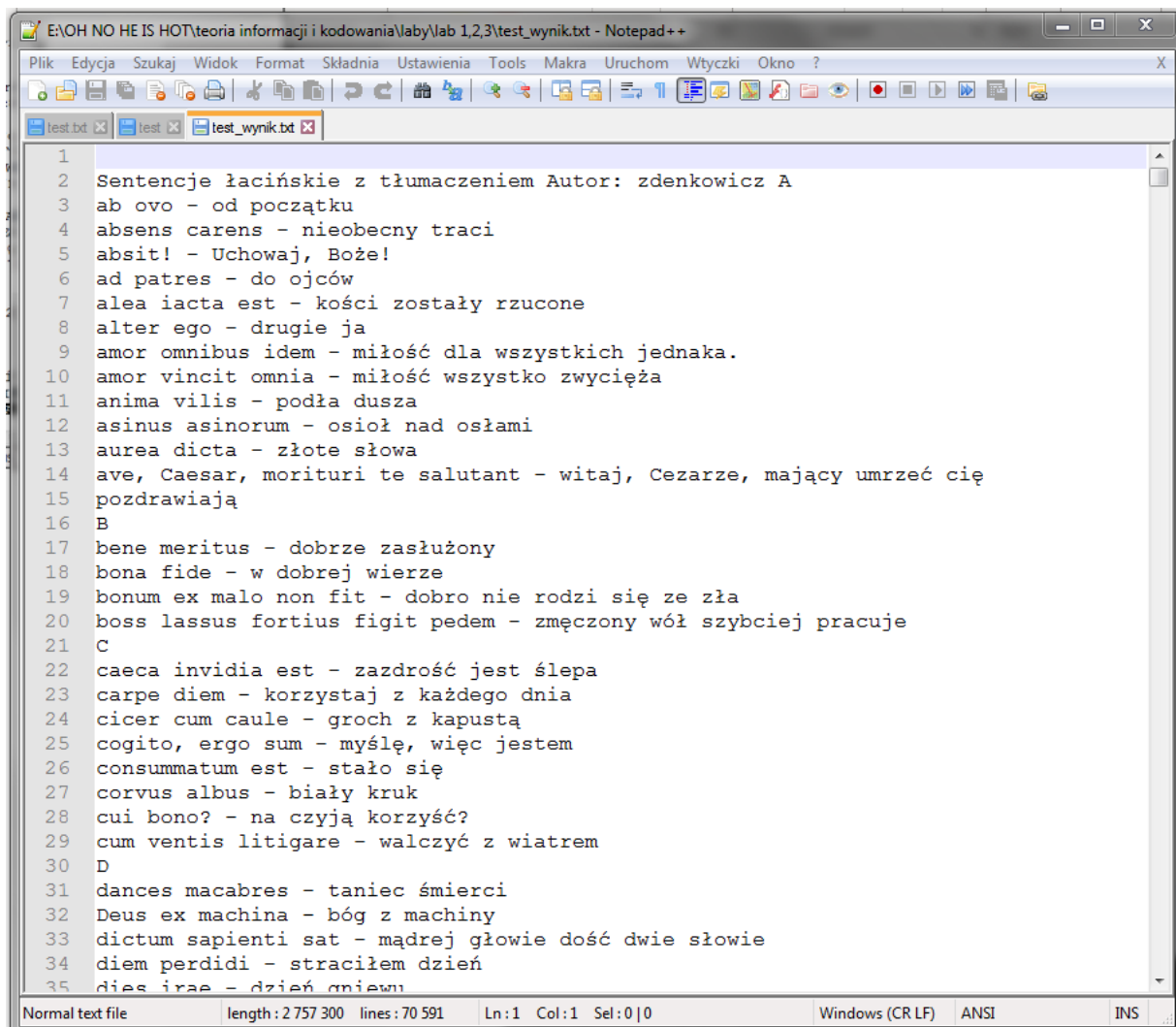
```
1
2 Sentencje łacińskie z tłumaczeniem Autor: zdenkowicz A
3 ab ovo - od początku
4 absens carens - nieobecny traci
5 absit! - Uchowaj, Boże!
6 ad patres - do ojców
7 alea iacta est - kości zostały rzucone
8 alter ego - drugie ja
9 amor omnibus idem - miłość dla wszystkich jednaka.
10 amor vincit omnia - miłość wszystko zwycięża
11 anima vilis - podła dusza
12 asinus asinorum - osioł nad osłami
13 aurea dicta - złote słowa
14 ave, Caesar, morituri te salutant - witaj, Cezarze, mający umrzeć cię
15 pozdrawiają
16 B
17 bene meritus - dobrze zasłużony
18 bona fide - w dobrej wierze
19 bonum ex malo non fit - dobro nie rodzi się ze zła
20 boss lassus fortius figit pedem - zmęczony wół szybciej pracuje
21 C
22 caeca invidia est - zazdrość jest ślepa
23 carpe diem - korzystaj z każdego dnia
24 cicer cum caule - groch z kapustą
25 cogito, ergo sum - myślę, więc jestem
26 consummatum est - stało się
27 corvus albus - biały kruk
28 cui bono? - na czyją korzyść?
29 cum ventis litigare - walczyć z wiatrem
30 D
31 dances macabres - taniec śmierci
32 Deus ex machina - bóg z maszyny
33 dictum sapienti sat - mądrej głowie dość dwie słowie
34 diem peridi - straciłem dzień
35 dies irae - dzień gniewu
```

Normal text file length: 2 757 300 lines: 70 591 Ln: 13 Col: 26 Sel: 0 | 0 Windows (CR LF) ANSI INS

Rys.32 Zawartość pliku test.txt

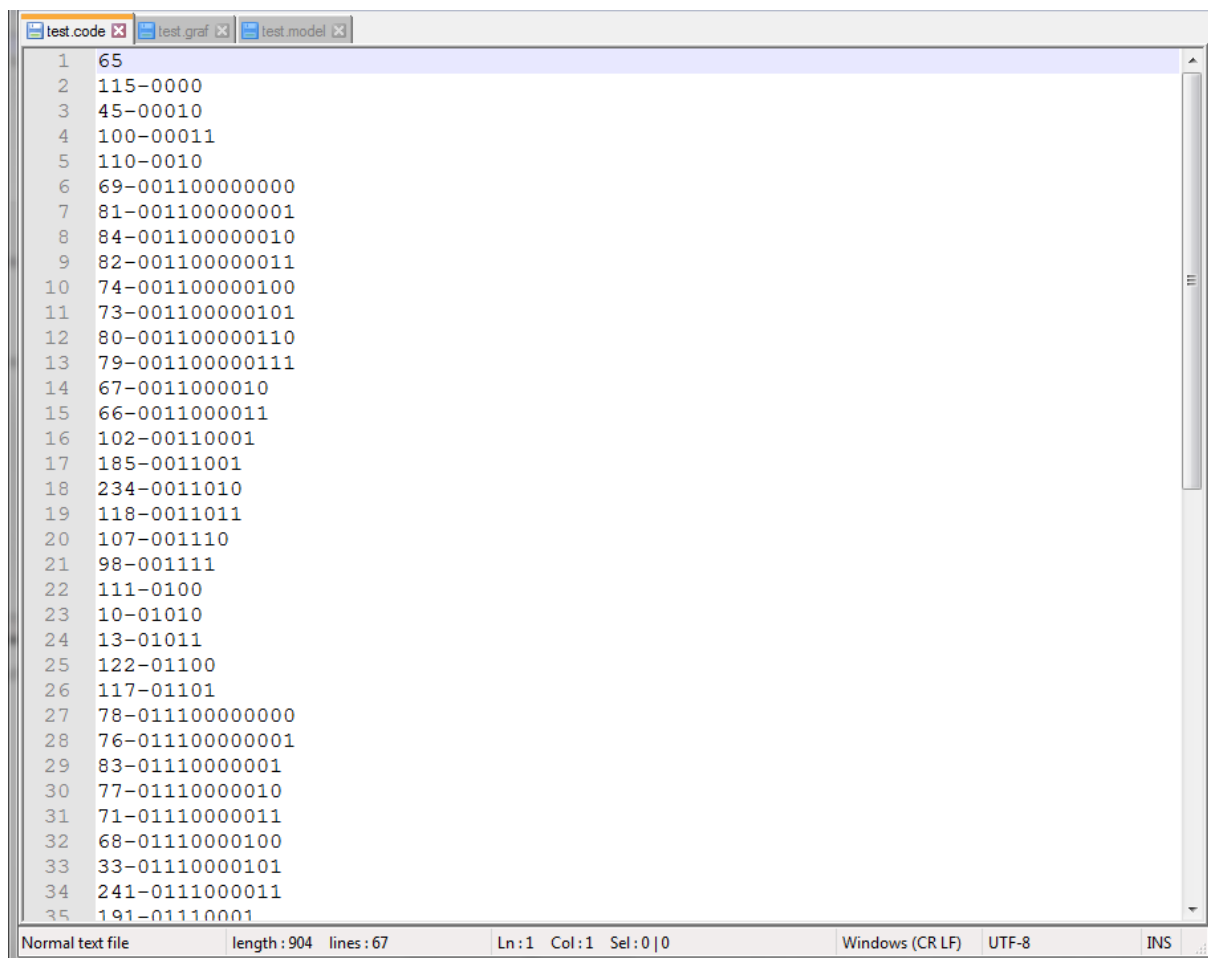
```
E:\OH NO HE IS HOT\teoria informacji i kodowania\lab\lab 1,2,3\test - Notepad++
Plik Edycja Szukaj Widok Format Składnia Ustawienia Tools Makra Uruchom Wtyczki Okno ?
test.bt test test_wynik.bt
1 ZsSOSYNdZB,čl,wleüčERSE/ÉWznÓž,-RSDC1Éx~•Ö-~>~'~";~+=+vsV~<0,I9,,DC4TzDLBüyMwČl-Px
2 Ns'}'X,t5āSUBō,ZL]ç.vJ*>7·ÍdVTI9REç'/ž%Eeó'}NžX~ETX&
3 tds{řE[SO+~\~Q$Í7a~YÍETBÍNAK~fiI+KØ2`ÜEYÖĐÓN0µA|`>ý| (Ø9NFD-P RSMVT SOHásl`šBEL=ęP::š
4 *4?ŠEČ ÍqB~j=
5 )žž(žThl3~eóšL~ō&EāñE5JHTÇ÷EF+n)~^ Ůō<ōfVN...~ōNULšRÉTXBS7i
6 cýžwLRSJŠ,Fžā)žSŮōEM1dūdta=-ńj0šōřčLñ7āř,ESC~+tŠI%>é{fnDššBōřč~Gú**rq"ACKā~e:ESC
7 •NAK"óúšwzōó'd7'Zž,,
8 "Łm~RÝNAK
9 Ąa~ć-ŠŠ
10 ',j{Bšžn:jwSUBrōōý= GSñš"W,nciÉCM>čeu{jemáoĀšütNĚ~`Fždĭ•-žTōšµ8JµSO řSYNPř?DC4wp_
11 jUSř_āeNAKESC?FS;Řž:jō`NAKjčž+~"QTÚW,ruřrōt'e]Mú
12 •NAKdL"švčLš÷řTijř9•ĚeEOTŮ\
13 NšçEMESC=N~RÝā1ē3uhřq•«ETXvH6...ňLĀŮDC4W dāl'E(tčhōāETBžSYN~YŮ]ōŠITř;žBš"E(žÝĚi|nĭ
14 ēBē5jµšµCANĀūĚšqo@āA+~"l9Ź"eNAK:FSG5dšruSUBažš VT;šōSOH«QžkóEōē=QPcřjĭ_ę/^PµCANç8~
15 +ĐōōEĪ4YŇ^ŮRřč6šLČĚ6÷E4ōó%J,svŮhñjışsā>÷dĀŠ~ĐQ+:~tĭl
16 i-]µN|SĪf~Ůye...ESC~::~<šETXwRSç%~Iā-@Ů~ś<~,~NRžý"ēĚLM?8" yŮluzt~N|6šř,žřESCāsl,E*ç:
17 EM=ī"č9n"W\EččEM-QŮÍžēĭmSYNžē^DC4@oETX~] >ŭo/ō,~ōĪI)~^ĐUScĭTē'ZIZG0)@?āř;šG0,"ž
18 Ō!éš~dDC4ōixĚ{š«Q~I ESut|{•«Eç7dn@dšš...NAKē~úūřŮ?/ř?ōž+š<žúž\ānšĭpōEš"]DCĪē3ī>ōšrWĪ
19 žµJ`eEOTŮ]`ŮnšdŇNkhāVswLšX?KĪdř>MVTčřQRžčā4u{VTō{"ESC=NĐňACKwā-@d"Ź-ID~šDLōā~K
20 -Y.š|QĪLšōAESCō#Rçtž.šdĀ÷ ŮĀjōYtnq/m~ŮQRŮLžž~ESC?RSslžRSČZ<+@I:~/
21 *ā~DC4SOĒā+~čX_ī=šřw~o~tōĪR~ō",vvJQ{řE;"ō-ĚŮESú*^ōd>CĪRŮLgōĭ<Q~xSUBg tŮQRŮĀ!
22 ůš""ŮŮ~žrō/AŮ^Īč~µDC2kōYwENOŮ9"Tž(ōšGSGšē<ńjš~"Pin ýSTX(@~RřčKōŇŮ/+JPĪLSOZ%
23 ~š~ENO-SYNŮ (@~ACK~8~ńCANŮ5jš+žž7EKňōš'-D...u÷sstT~ō6dtž/SETBtVTQ!T'|]DĪEŠOH,ht``Q:
24 āō,āĀ""=EķCANĭtSO>ýčQř?āĚ~-(š~ō<ō/E=bSOĪó;±@QšNAKčgū-Đēū>ĚŮETX=EtFDC4w~DC2řēcd'ĭ
25 *QŮ.šžžVTQ7]bžd(yexĚ"šĀ)đ,š"ēnōČ]9ŠI~mjšē-]DC3
26 (yex;eNAK/Ě(SGŠšā.šfĭřŮŠ#~ZššqēŮWóĭ...CĚ.āŮ (@~YEōšRŮ-ETX)E?EMtā3OACKY9GSBōM'Ů
27 ōūū!TWYčēōŮĚ~Āēōžn-ō%QĭōGSNáoštāQáo(šES~'PāwXdúč-2zōřymýESCēē>lSYNč@~=ř/ĀžL12:
28 _š#ŮĀ~n~'Pi'žŮPin"mLš~ETX&
29 š-DRSāŮ)FŮčbō'/
30 Íš-XčĭNAK:FSG5{'_+řESC$N%<Eō`ōjōRSNř~...sYNAKōdLčŮ"ž,ūž86EMz'ĭx
31 3`~vETX|š+SEMšžūā~QĀĭxĀACKpĭš|0ETB7KFF` (@~]8Ě?ó'ŮĚdVt3~ĭV-<US`AESCžēSUB?STXšhčĭV
32 +Sç:ACKMzžāōw~řšgāž'ç""Od`xn(«'š/ACK~ščāFFč-Zq.ř?_zšĐ'šžXŁCŮNAK;Og5j0SUBµ:>EMĪŮ
33 ĪĭlŮžĚ"ūžtMř|.Lā»,xĀřID~řd@Y'žwwBEL} %ju65dLĚā»,xĀōóO(žx;ĭNĭřdšršµCANGSZāBS<AC
34 5ĚENōřřnACKē""OW ruřud>ā'Ů>ĀcdFV@ř+ LĀFSāŮlDC3-CFMDC2w^~ī!ACKDē'ōĚ
```

Rys.33 Zawartość skompresowanego pliku test



```
1
2 Sentencje łacińskie z tłumaczeniem Autor: zdenkowicz A
3 ab ovo - od początku
4 absens carens - nieobecny traci
5 absit! - Uchowaj, Boże!
6 ad patres - do ojców
7 alea iacta est - kości zostały rzucone
8 alter ego - drugie ja
9 amor omnibus idem - miłość dla wszystkich jednaka.
10 amor vincit omnia - miłość wszystko zwycięża
11 anima vilis - podła dusza
12 asinus asinorum - osioł nad osłami
13 aurea dicta - złote słowa
14 ave, Caesar, morituri te salutant - witaj, Cezarze, mający umrzeć cię
15 pozdrawiają
16 B
17 bene meritus - dobrze zasłużony
18 bona fide - w dobrej wierze
19 bonum ex malo non fit - dobro nie rodzi się ze zła
20 boss lassus fortius figit pedem - zmęczony wół szybciej pracuje
21 C
22 caeca invidia est - zazdrość jest ślepa
23 carpe diem - korzystaj z każdego dnia
24 cicer cum caule - groch z kapustą
25 cogito, ergo sum - myślę, więc jestem
26 consummatum est - stało się
27 corvus albus - biały kruk
28 cui bono? - na czyją korzyść?
29 cum ventis litigare - walczyć z wiatrem
30 D
31 dances macabres - taniec śmierci
32 Deus ex machina - bóg z maszyny
33 dictum sapienti sat - mądrej głowie dość dwie słowie
34 diem perdidi - straciłem dzień
35 dies irae - dzień gniewu
```

Rys.34 Zawartość zdekompresowanego pliku test_wynik.txt



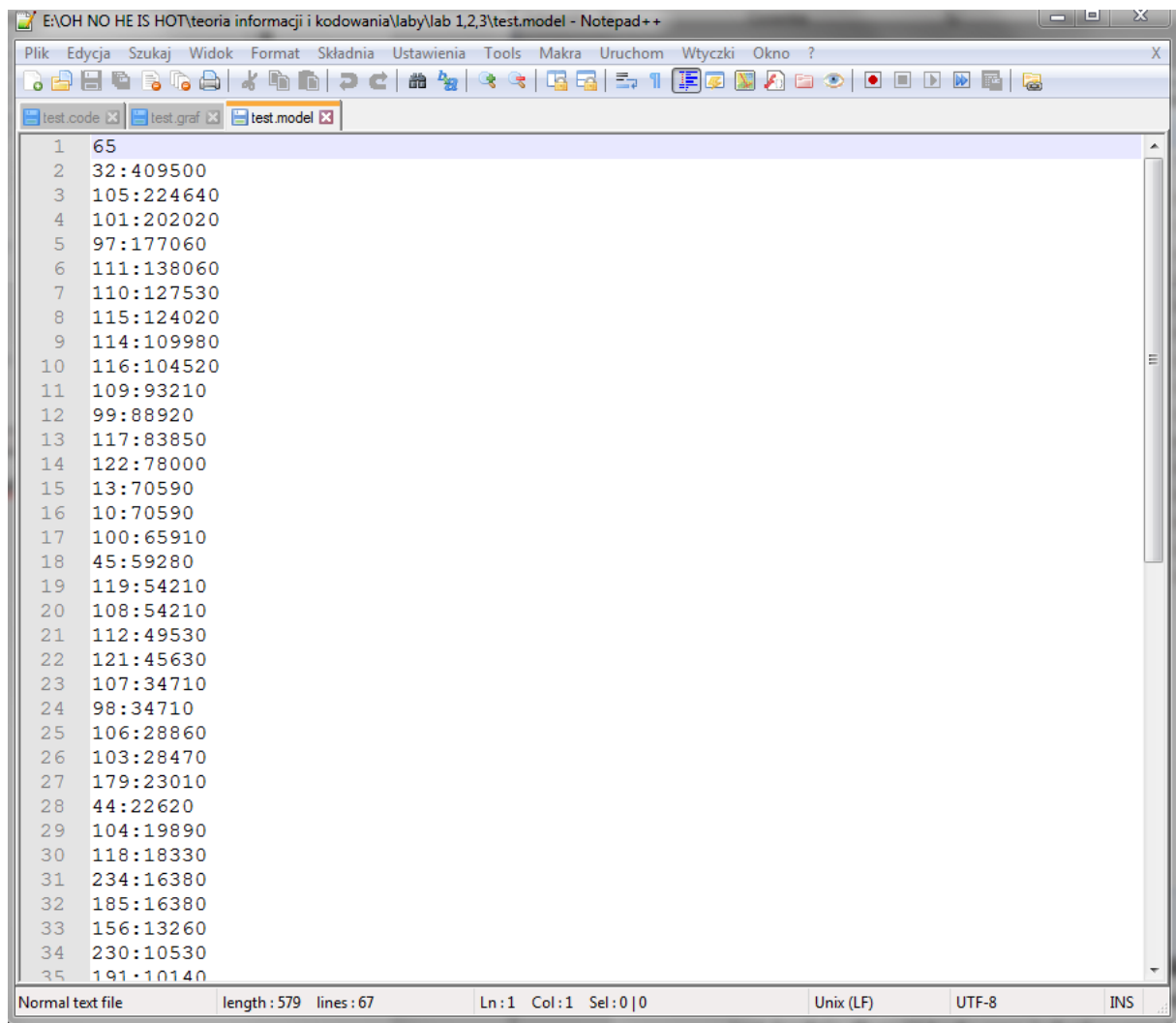
```
1 65
2 115-0000
3 45-00010
4 100-00011
5 110-0010
6 69-001100000000
7 81-001100000001
8 84-001100000010
9 82-001100000011
10 74-001100000100
11 73-001100000101
12 80-001100000110
13 79-001100000111
14 67-0011000010
15 66-0011000011
16 102-00110001
17 185-0011001
18 234-0011010
19 118-0011011
20 107-001110
21 98-001111
22 111-0100
23 10-01010
24 13-01011
25 122-01100
26 117-01101
27 78-011100000000
28 76-011100000001
29 83-01110000001
30 77-01110000010
31 71-01110000011
32 68-01110000100
33 33-01110000101
34 241-0111000011
35 191-011110001
```

Normal text file length: 904 lines: 67 Ln: 1 Col: 1 Sel: 0 | 0 Windows (CR LF) UTF-8 INS

Rys.35 Zawartość pliku test.code

```
test_code x test.graf x test.model x
1 #64:2757300 childLeft:#62 childRight:#63 Parent:-
2 #62:1126320 childLeft:#58 childRight:#59 Parent:#64
3 #58:512460 childLeft:#51 childRight:#52 Parent:#62
4 #51:249210 childLeft:115 childRight:#43 Parent:#58
5 115:124020 childLeft:- childRight:- Parent:#51
6 #43:125190 childLeft:45 childRight:100 Parent:#51
7 45:59280 childLeft:- childRight:- Parent:#43
8 100:65910 childLeft:- childRight:- Parent:#43
9 #52:263250 childLeft:110 childRight:#44 Parent:#58
10 110:127530 childLeft:- childRight:- Parent:#52
11 #44:135720 childLeft:#37 childRight:#38 Parent:#52
12 #37:66300 childLeft:#31 childRight:#32 Parent:#44
13 #31:31590 childLeft:#27 childRight:185 Parent:#37
14 #27:15210 childLeft:#23 childRight:102 Parent:#31
15 #23:7020 childLeft:#17 childRight:#19 Parent:#27
16 #17:3120 childLeft:#11 childRight:#12 Parent:#23
17 #11:1560 childLeft:#3 childRight:#2 Parent:#17
18 #3:780 childLeft:69 childRight:81 Parent:#11
19 69:390 childLeft:- childRight:- Parent:#3
20 81:390 childLeft:- childRight:- Parent:#3
21 #2:780 childLeft:84 childRight:82 Parent:#11
22 84:390 childLeft:- childRight:- Parent:#2
23 82:390 childLeft:- childRight:- Parent:#2
24 #12:1560 childLeft:#6 childRight:#4 Parent:#17
25 #6:780 childLeft:74 childRight:73 Parent:#12
26 74:390 childLeft:- childRight:- Parent:#6
27 73:390 childLeft:- childRight:- Parent:#6
28 #4:780 childLeft:80 childRight:79 Parent:#12
29 80:390 childLeft:- childRight:- Parent:#4
30 79:390 childLeft:- childRight:- Parent:#4
31 #19:3900 childLeft:67 childRight:66 Parent:#23
32 67:1950 childLeft:- childRight:- Parent:#19
33 66:1950 childLeft:- childRight:- Parent:#19
34 102:8190 childLeft:- childRight:- Parent:#27
35 185:16380 childLeft:- childRight:- Parent:#31
Normal text file length: 6171 lines: 130 Ln: 1 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS
```

Rys.36 Zawartość pliku test.graf



```
1 65
2 32:409500
3 105:224640
4 101:202020
5 97:177060
6 111:138060
7 110:127530
8 115:124020
9 114:109980
10 116:104520
11 109:93210
12 99:88920
13 117:83850
14 122:78000
15 13:70590
16 10:70590
17 100:65910
18 45:59280
19 119:54210
20 108:54210
21 112:49530
22 121:45630
23 107:34710
24 98:34710
25 106:28860
26 103:28470
27 179:23010
28 44:22620
29 104:19890
30 118:18330
31 234:16380
32 185:16380
33 156:13260
34 230:10530
35 191:10140
```

Rys.37 Zawartość pliku test.model

Podsumowanie:

Tab.1 Porównanie działania programu

Rozmiar pliku wejściowego	Liczba różnych symboli	Rozmiar pliku skompresowanego
103 B	2	19 B
40 B	9	13 B
203 B	27	108 B
19 143 B	34	10 074 B
2 757 300 B	65	1 616 502 B

Program kompresuje zarówno duże jak i małe pliki bez względu jakie symbole mają w sobie.

Jak widać w Tabeli 1 im większy plik wejściowy tym więcej miejsca jesteśmy w stanie zaoszczędzić dzięki kompresji. Dużą rolę w stopniu skompresowania pliku odgrywa liczba różnych symboli w pliku wejściowym. Im więcej różnorodności tym większy będzie skompresowany plik. Dzieje się tak, ponieważ przy większej liczbie symboli tworzone będą dłuższe kody je reprezentujące, czyli średnia długość kodu będzie większa. Widać to po przykładowych danych w Tabeli 1. Plik o wielkości 103 bajtów z 2 różnymi symbolami został skompresowany do 19 bajtów, czyli do niecałych 20% swojej oryginalnej wielkości. Pliki z około 30 różnymi symbolami zmniejszone zostały do około połowy swojej oryginalnej wielkości, za to plik z 65 symbolami zmniejszył swoją wielkość do około 58%.