****

**PROJEKT**

**Pozyskiwanie i odzyskiwanie danych elektronicznych**

**Kodowanie Huffmana**

**Jakub Kuśnierz**

**w61674**

**Informatyka, II semestr studiów magisterskich**

1. **Cel projektu**

Celem projektu było zaprojektowanie, wykonanie i przetestowanie aplikacji realizującej metodę bezstratnej kompresji Huffmana. Środowiskiem które zostało wybrane do wykonania całości został język programowania Java w wersji JDK 1.8. Wykorzystane zostały tylko funkcjonalności wbudowane w kod źródłowy języka, nie potrzebne były żadne biblioteki zewnętrzne. Użyto między innymi implementacji struktur danych jak mapa oraz kolejka priorytetowa oraz interfejsu Comparable służącego do porównywania obiektów. Programem w którym powstawał kod projektu był Intellij Idea produkowany przez firmę Jetbrains i wykorzystany w pełnej wersji na podstawie licencji udostępnianej dla studentów.

1. **Kodowanie Huffmana**

Kodowanie Huffmana jest jedną z metod kompresji bezstratnej. Zostało opracowane w 1952 roku przez Amerykanina Davida Huffmana. Algorytm nie jest doskonały, jednak używany ze względu na prostą w porównaniu do innych algorytmów implementacje oraz brak ograniczeń patentowych. Kodowanie Huffmana jest przykładem wykorzystania algorytmu zachłannego, którego działanie polega na wybraniu w każdym kroku najlepiej rokującego w danym momencie rozwiązania częściowego.

Kompresja Huffmana polega na zastępowaniu znaków kompresowanego pliku krótszymi kodami, a dokładniej ciągami bitów. Dzięki temu wynikowy ciąg bajtów jest znacznie krótszy niż ciąg wejściowy. Znaki są kodowane na 8 bitach, można na nich zapisać do 256 znaków. Każdy standardowy znak na przypisany swój numer w tablicy kodów ASCII. Rzadko kiedy jednak wszystkie z nich występują w pliku. Najczęściej są tam tylko litery, cyfry i kilka znaków specjalnych. Dzięki temu można wykorzystać mniej bitów i zaoszczędzić przez to dużo miejsca.

Kodowanie Huffmana polega na przypisaniu znakom liczb porządkowych, np. załóżmy, że nasz plik został zapisany za pomocą znaków alfabetu A={a, b, c, d, e). Alfabet ten składa się z 5 znaków jest to dużo mniej niż 256, nie potrzebujemy więc aż 8 bitów, wystarczą nam 3 bity, gdyż 23=8, za pomocą 3 bitów można zapisać 8 znaków. Załóżmy, że nasz plik ma dokładnie 1MB, czyli 1 048 576 bajtów. Każdy bajt jest kodowany na 8 bitach, czyli nieskompresowany plik zajmuje 8 \* 1 048 576 = 8 388 608 bitów. Jednak nam wystarczą po 3 bity na znak. Przypisujemy więc każdemu znakowi liczbę porządkową (zapisaną binarnie): a=000, b=001, c=010, d=011, e=100. Zamiast używać 8 bitowych kodów ASCII używamy własnych 3 bitowych kodów. Odnieśliśmy na tym duże korzyści: 1 048 576 \* 3 = 3 145 728 bitów, czyli 37,5% pliku wejściowego. Musimy zauważyć jednak, że tak dobry wynik osiągamy tylko dlatego, że nasz alfabet miał jedynie 5 znaków. Im większy alfabet, tym gorszy wynik. Przy alfabecie składającym się ze wszystkich 256 znaków będziemy potrzebowali 8 bitów, czyli tyle ile normalnie.

Główną wadą zastosowania algorytmu Huffmana jest konieczność przesłania całego drzewa aby możliwe było odtworzenie kodów jakie przypisaliśmy poszczególnym znakom. Bez niego dekompresja pliku staję się niemożliwa.

Istnieją równe modyfikację kodowania Huffmana m.in. algorytm dynamicznego kodowania Huffmana lub kodowanie po 2 znaki na raz. W projekcie została pokazana standardowa wersja algorytmu, oparta na statycznym kodowaniu.

Własności kodu Huffmana są następujące:

- jest kodem prefiksowym; oznacza to, że żadne słowo kodowe nie jest początkiem innego słowa

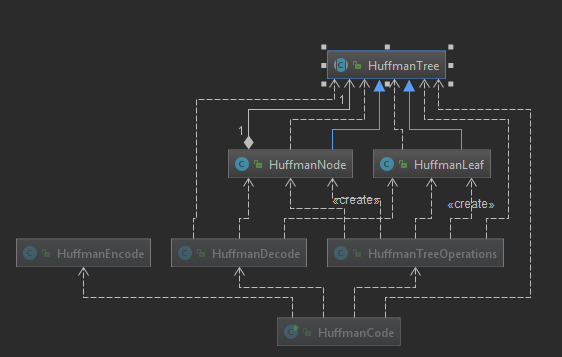
- średnia długość słowa kodowego jest najmniejsza spośród kodów prefiksowych

- słowa kodu dwóch najmniej prawdopodobnych symboli mają równą długość.

1. **Diagram klas aplikacji**

Aplikacja składa się z dwóch głównych modułów. Jest to model do których należy klasa abstrakcyjna *HuffmanTree* implementująca interfejs *Comparable* oraz klasy reprezentujące liście i węzły drzewa dziedziczące po *HuffmanTree* czyli *HuffmanNode* i *HuffmanLeaf.* Zadaniem tych klas jest odwzorowanie drzewa wraz z możliwością przechowywania w nim potrzebnych informacji. Drugim głównym modułem są operację które możemy wykonywać na wprowadzonym tekście. Należy do nich zbudowanie drzewa, stworzenie mapy z przypisanymi kodami, wypisanie drzewa do celów poglądowych, zakodowanie oraz zdekodowanie tekstu. Całość funkcjonalności użyta została w głównej klasie aplikacji o nazwie HuffmanCode. Została w niej również stworzona metoda zamiany zakodowanego tekstu na nowe znaki z tabeli kodów ASCII ukazujący lepsze ukazanie naturalnemu odbiorcy zaoszczędzenie miejsca poprzez konwersje niż ciąg bitów.

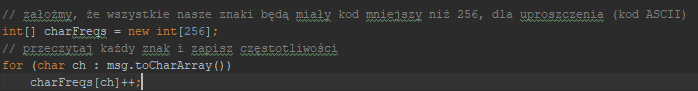
Diagram klas wraz z zachodzącymi relacjami wygenerowany za pomocą wykorzystywanego programu Intellij Idea został pokazany na zdjęciu poniżej.



1. **Opis wykonywanych kroków algorytmu**

Pierwszym krokiem działania algorytmu jest wczytanie tekstu, który chcemy poddać konwersji algorytmem Huffmana. Następnie przechodzimy w pętli po każdym znaku wczytanego tekstu zwiększając wystąpienie każdego znaku o 1 w momencie jego napotkania w tekście. Dla uproszczenia zakładamy, że wszystkie nasze znaki będą miały kod mniejszy niż 256 czyli będą należeć do tablicy kodów ASCII. Spowodowane to jest koniecznością sprecyzowania kodowania na potrzeby projektu. Początkowo standardy zawierały 127 znaków, które były wystarczające, obecnie najnowszy standard UNICODE 12 zawiera ich już ponad 110 tysięcy. Zawarte w tym są litery z różnych języków w tym chiński oraz emotikony. Znaki te powstają z połączenia innych znaków zapisywanych na 1 bajcie przez co mają większe rozmiary. Na zapisanie największego pojedynczego znaku a zasadzie emotikony potrzeba 11 bajtów. My jednak skupimy się na znakach zapisywanych na jednym bajcie, dlatego użyjemy tablicy kodów ASCII.

Poniżej przedstawiona została operacja przypisania częstotliwości do znaków według ich występowania w tekście.



Kolejnym krokiem gdy mamy już tablicę z częstotliwością występowania znaków jest zbudowanie tak zwanego drzewa Huffmana. Odbywa się to za pomocą specjalnie przygotowanej metody, która jako parametr wejściowy przyjmuję wcześniej stworzoną tablice.

W metodzie pierwsze zostaje zainicjalizowana kolejka priorytetowa, którą wykorzystamy do wykonania drzewa. Następnie są do niej dodawane liście, którymi są wszystkie znaki występujące co najmniej raz w początkowym tekście. Są one do kolejki według priorytetu. Czym więcej razy dany znak wystąpił tym wyżej będzie w kolejce. W tym momencie nasze drzewo składa się z samych liści, które są na jednym poziomie. Kolejnym krokiem, który będziemy wykonywać aż do pozostania nam w kolejce jednego węzła które będzie korzeniem głównym jest iteracja po drzewie. Podczas niej wybieramy dwa drzewa o najmniejszej częstotliwości występowania i łączymy je w jedno sumując częstotliwości, a następnie wstawiamy nowe drzewo do naszej kolejki w odpowiednie miejsce według nowej częstotliwości. Operacje tą wykonujemy aż do uzyskania korzenia naszego drzewa, które jest obiektem wynikowym zwracanym przez funkcję.

Kod metody służącej budowaniu drzewa został pokazany poniżej.

