Bezpieczeństwo systemów i usług informatycznych 2

Laboratorium 1

Paul Paczyński 254307

GRUPA PON 14:30
30.10.2023

Wprowadzenie

Zadanie laboratoryjne polegało na implementacji algorytmu Huffmana. Produktem końcowym ma być program wykonujący bezstratną kompresję danych używając tego algorytmu. Program ma poprawnie kompresować dane bity, oraz potrafić zamieniony ciąg odczytać ponownie.

Bezstratna kompresja polega na zakodowaniu informacji zawierającą zmniejszoną liczbę bitów niż jej oryginalny stan, gwarantując przy tym możliwość dokładnego odczytania informacji po jej kompresji.
Algorytm Huffmana polega na wprowadzeniu struktury zwanej słownikiem, przetrzymującą wszystkie znaki, które chociaż raz wystąpiły w pliku. W zależności od długości słownika, określamy ilość bitów, które będą potrzebne do zapisania jednego znaku po konwersji. Dla każdego znaku w słowniku przypisujemy mu liczbę, najczęściej jest to pozycja w strukturze. Np:

Słownik = [A, B, C, D]
Do zapisania każdego ze znaków wystarczą dwa bity, znaki można zakodować w następujący sposób:
A = 00 , B = 01, C = 10, D = 11
Dzięki tej metodzie użyjemy do zapisu jednego bajta, zamiast czterech. Samo takie zakodowanie wiadomości nie pozwoli nam na ponownie odczytanie, o ile nie znamy słownika. Problem został rozwiązany poprzez dodanie zawartości słownika przed ciąg bitów wiadomości, czyli ABCDx, gdzie x to bajt zakodowanej wiadomości. Nasuwa się tutaj jednak problem z rozpoznaniem kiedy kończy, a kiedy zaczyna się słownik. Dla prostoty pierwszy bajt zawiera informację o długości słownika.

Kolejnym problemem jest wyrównanie ilości bitów do pełnych bajtów. Największą ilość bitów jaką należy w skrajnym przypadku dodać to siedem. W wykonanym programie wstawiane są jedynki na sam koniec ciągu bitów. Ilość wstawionych dodatkowych bitów zawarta jest w pierwszych trzech bitach zaraz po końcu słownika.

Dzięki wyżej wymienionym rozwiązaniom, program czyta pierwszy bajt, dostając informację o długości słownika, zapisuje ten słownik, a następnie na podstawie zakodowanych bitów odtwarza informację oryginalną.

Implementacja

Funkcja tworzy tablice unikalnych znaków pojawiających się w tekście.

```
5  def get_alphabet(file_string):
6      dictionary = []
7      for symbol in file_string:
8          if symbol not in dictionary:
9              dictionary.append(symbol)
10     return dictionary
```

Funkcja sprawdza ile bitów potrzeba na zapisanie znaków. Następnie dla każdego symbolu znajduje jego indeks w alfabecie i odpowiednio dostawia go do końcowego stringa.

```
14 def encode_fixed_length(text,alphabet,):
15     # Ile bitów potrzeba na znak
16     code_length = (len(alphabet)-1).bit_length()
17     encoded_text = ''
18     for char in text:
19         char_index = alphabet.index(char)
20         char_code = format(char_index, '0' + str(code_length) + 'b')
21         encoded_text += char_code
22     return encoded_text
```

Jak dotąd wiadomość była stringiem udającym ciąg bitów, jednak w rzeczywistości był to string ze znakami ascii zer i jedynek. O ile sposób ten nie jest optymalny, to dla czytelności jest bardzo pomocny. Poniższa funkcja dostawia do wiadomości informację o długości dodatkowych bitów, oraz dostawia je na koniec. Następnie każdy „chunk", czyli ciąg znaków jedynek i zer konwertowany jest na znak i dołączany do wiadomości.

```
24  def change_to_bits(message):
25      coded_message = ""
26      first_bits = format(0,'03b') # Trzy bity początkowe, init 000
27      message = first_bits + message # Wstawienie trzy bity na początek
28      text_length = len(message)
29      additional_bits = 8 - (text_length % 8) # Obliczenie brakujących bitów
30      if additional_bits > 0:
31          message = format(additional_bits,'03b') + message[3:]
32      for i in range(0, text_length, 8): # Czytanie po bajcie
33          chunk = message[i:i + 8]
34          if len(chunk) < 8: # Dostawianie dodatkowych bitów
35              for i in range(additional_bits):
36                  chunk = chunk + "1"
37              coded_message += chr(int(chunk,2)) # Zamiana na znak
38          else:
39              coded_message += chr(int(chunk,2))
40      return coded_message
```

Pod koniec tworzona jest końcowa wiadomość w postaci:

{rozmiar słownika}{słownik}{wiadomość}

```
43  def start_compress(file_string):
44
45      alphabet = get_alphabet(file_string)
46      encoded_message = encode_fixed_length(file_string, alphabet)
47      return_value = chr(len(alphabet)) +  "".join(alphabet) + change_to_bits(encoded_message)
48      return return_value
```

Dekompresja korzysta z meta informacji zakodowanych w pliku. Najpierw odczytywana jest długość słownika z pierwszego bajta, następnie tworzony jest słownik. Funkcja create_binary z pozostałej części ciągu tworzy ciąg bitów. W nim odczytuje informację o dodatkowych jedynkach na końcu pliku, usuwając je, oraz 3 bity początkowe z tą informacją, zostawiając same potencjalne znaki. Pętla sprawdza kolejne bity, których ilość zależy od bit_for_sign, który można obliczyć z długości alfabetu.

```python
def decompress_text(text):
    alphabet_length  = ord(text[0])
    alphabet = list(text[1:alphabet_length+1])

    binary_string = create_binary(text[alphabet_length+1:])
    bit_for_sign = (alphabet_length-1).bit_length()
    final_string = ""

    for i in range(0,len(binary_string),bit_for_sign):
        chunk = binary_string[i:i+bit_for_sign]
        index = int(chunk,2)
        final_string = final_string + alphabet[index]
    return final_string


def create_binary(text):
    output_string=""
    for character in text:
        output_string = output_string + format(ord(character),'08b')
    filler = int(output_string[0:3],2)
    output_string = output_string[3:]
    output_string = output_string[:-filler]
    return output_string
```

Efekt końcowy:

```
WIADOMOSC:
Jakaś testowa wiadomość dla testów

Skompresowana wiadomsoc:
►Jakś tesowidmćló $&Î±"CqÂÎ¿?

Dekompresowana wiadomosc:
Jakaś testowa wiadomość dla testów
```