Sprawozdanie na przedmiot Kompresja Danych, implementacja algorytmu LZW.

Bartosz Puszkarski

Grudzień 2019

Spis treści

1 Wstęp **3**

2 Omówienie teoretyczne algorytmu LZW **3**
2.1 Cechy metody LZW 3
2.2 Algorytm LZW 3

3 Projekt i działanie programu **4**
3.1 Projekt 4
3.2 Implementacja 5
3.2.1 Moduł główny - main.py 5
3.2.2 Algorytm LZW 5

4 Podsumowanie i obserwacje **7**

1 Wstęp

Celem projektu było zaimplementowanie dowolnego algorytmu kompresji bestratnej, które docelowym obszarem działań będą pliki tekstowe. Pozwolono nam na wybór języka i środowiska implementacji wybranej metody, wybór ograniczono do dwóch opcji - Matlab lub Python.
Po zapoznaniu się z materiałem przedstawionym na wykładzie oraz uzupełniając informacje w oparciu o źródła internetowe zdecydowałem się na implementację algorytmu bezstratnej kompresji słownikowej w wersji uproszczonej **LZW**.

2 Omówienie teoretyczne algorytmu LZW

Metoda LZW jest modyfikacją swojej poprzedniczki, metody LZ78. Dziedziczy po niej zasady działania. W metodzie tej, słownik inicjalizowany jest alfabetem kodowanego ciągu, a elementarne składniki zbioru (*elementy podstawowe*) są znane i uporządkowane w taki sposób, że każdy element podstawowy ma ustalony poprzednik i następnik.
Głównym powodem, dla którego wybrałem właśnie tę metodę jest fakt, że udało mi się ją przyswoić najszybciej i wydawała mi się najbardziej intuicyjna.

2.1 Cechy metody LZW

- Daje lepsze wyniki niż LZ78 (słownik szybko się zapełnia),
- Dane wyjściowe składają się jedynie z indeksów w słowniku,
- Możliwość wystąpienia przypadku szczególnego: kodowanie pozycji w słowniku, która nie została jeszcze zapełniona - wymaga odpowiedniej modyfikacji algorytmu.
- Najlepiej koduje krótkie sekwencje danych,

2.2 Algorytm LZW

Aby w najprostszy i najczytelniejszy sposób przedstawić działanie algorytmu posłużę się listą kroków dostarczoną nam podczas wykładu przez prof. Raka.

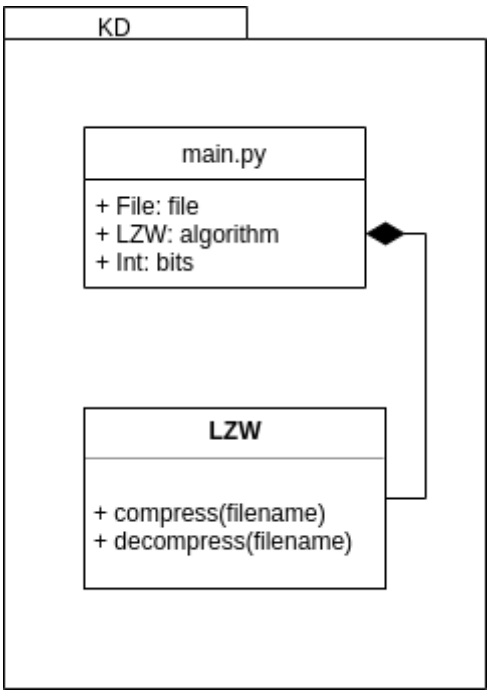
- Wypełnienie słownika alfabetem kodowanej informacji,
- Wczytanie znaku z wejścia,

- Jeśli znak połączony z ostatnio wczytanymi (o ile były) da frazę, która znajduje się w słowniku, wyemituj jej indeks, a do pamięci zapisz ostatni znak z wejścia, który nie pasował do frazy ze słownika.
- Dopisanie do słownika frazy powstałej z połączenia znaku w pamięci i wczytanych znaków.
- Powtarzanie tego procesu aż do wyczerpania strumienia wejściowego.

3 Projekt i działanie programu

3.1 Projekt

Program został skonstruowany w sposób modularny, gdzie moduł główny, w procesie realizacji celów projektowych, posługuje się klasą **LZW**, w której został zaimplementowany algorytm kompresji i dekompresji z wykorzystaniem metody LZW.



Rysunek 1: Podział modułowy programu.

Program ograniczony został jedynie do działania w trybie wsadowym. Aby zapewnić prostotę obsługi została także zaimplementowana pomoc.

```
->  kompresja-projekt python3 main.py -h
usage: main.py [-h] [-c] [-d] FILE BITS

Implementacja algorytmu kompresji słownikowej bezstratnej LZW, wykonana w
języku Python v3.6, jako projekt na Podstwy Kompresji Danych laboratoria.
Należy skorzystać tylko i wyłącznie z jednego trybu pracy programu na raz.

positional arguments:
  FILE        nazwa pliku, który chcemy skompresować / zdekompresować
  BITS        ilość bitów wykorzystywana do określenia maksymalnej pojemności
              słownika (nie zaleca się korzystania z liczb poniżej 8)

optional arguments:
  -h, --help  show this help message and exit
  -c          Tryb pracy - kompresja
  -d          Tryb pracy - dekompresja
```

3.2 Implementacja

3.2.1 Moduł główny - main.py

W module głównym **main.py** następuje użytych flag:

```
if args.decompress and args.compress:
    raise ValueError("Wybrano więcej niż jeden tryb pracy")
```

oraz wywołanie pożądanej funkcjonalności:

```
if args.compress:
    algorithm.compress(args.filename)
elif args.decompress:
    algorithm.decompress(args.filename)
```

3.2.2 Algorytm LZW

Algorytm został zaimplementowany w klasie **LZW** w obu trybach pracy - kompresji oraz dekompresji.
Implementacja głównej części algorytmu realizującej **kompresję**:

```
for symbol in data:
    recent_chain_with_next_symbol = recent_chain + symbol
    if recent_chain_with_next_symbol in self.dictionary:
```

```
        recent_chain = recent_chain_with_next_symbol
    else:
        compressed_data.append(self.dictionary[recent_chain])
        if(len(self.dictionary) <= self.max_table_size):
            self.dictionary[recent_chain_with_next_symbol] = self.dict_size
            self.dict_size += 1
        recent_chain = symbol
if recent_chain in self.dictionary:
    compressed_data.append(self.dictionary[recent_chain])
```

Oczywiście kompresja ta realizowana jest po uprzednim zainicjowaniu słownika zgodnie z zasadami metody:

```
self.dictionary = {chr(i): i for i in range(dict_size)}
```

Następnie skompresowane dane są zapisywane do pliku, o nazwie tożsamej nazwie pliku danych wejściowych z rozszerzeniem ***.lzw**, w postaci bitów zapisanych z kodowaniem hexadecymalnym:

```
self.compressed_data = compressed_data
output_file = open(filename.split('.')[0] + '.lzw', "wb")
for data in self.compressed_data:
    output_file.write(pack('>H', int(data)))
output_file.close()
```

Implementacja algorytmu **LZW** realizująca dekompresję podanego pliku w formacie ***.lzw**. (Program był testowany jedynie na pliku stworzonym przez przez swoją instancję - rezultat wywołania metody *compress()*).

```
for code in compressed_data:
    if not (code in dec_dictionary):
        dec_dictionary[code] = recent_chain + recent_chain[0]
    decompressed_data += dec_dictionary[code]
    if not(len(recent_chain)) == 0:
        dec_dictionary[next_code] =recent_chain + (dec_dictionary[code][0])
        next_code += 1
    recent_chain = dec_dictionary[code]
```

Tak jak w przypadku kompresji, tutaj także wyżej przedstawiona część realizowana jest w oparciu o wcześniej inicjowany słownik:

```
dec_dictionary = {i: chr(i)) for i in range(dec_dict_size)}
```

Ponieważ dekompresowane dane pochodzą z pliku, którego treść została zakodowana w systemie hexadecymalnym istnieje konieczność odkodowania danych wejściowych. Podstawą operacji jest odczytywanie par znaków z pliku wejściowego, dekodowanie ich i dodawanie do tablicy.

```
while True:
    rec = f.read(2)
    if len(rec) != 2:
        break
    (data, ) = unpack('>H', rec)
    compressed_data.append(data)
```

Otrzymane zdekompresowane dane zapisywane są w pliku o nazwie tożsamej z plikiem wejściowym z dodanym sufiksem **decompressed** w formacie ***.txt**:

```
out = filename.split('.')[0] + "_decompressed.txt"
output_file = open(out, 'w')
for data in decompressed_data:
    output_file.write(data)
output_file.close()
```

4 Podsumowanie i obserwacje

W trakcie testowania programu zauważyłem ciekawe zachowanie w przypadku podania za małej liczby bitów złóżących do określania wielkości - gdy podano liczbę mniejszą od 8, wielkość słownika była mniejsza niż 256 pozycji a plik wynikowy był 2x większych rozmiarów, niż plik źródłowy, który miał zostać skompresowany. Oczywiście wynik kompresji był nie do odkodowania, więc zakładam, że zwyczajnie był zły. Udało mi się osiągnąć kompresję często-powtarzającego się tekstu zawartego w pliku testowym na poziomie 50%.