Kodowanie słownikowe danych o strukturze bajtowej

Hubert Adamkiewicz

Styczeń 2021

1 Opis problemu

Zadanie polega na zrealizowaniu prostego kodera i dekodera wykorzystującego metodę LZ77 (dla ustalenia uwagi proszę stosować metodę w wariancie przedstawionym w materiałach wykładowych). Przyjmujemy następujące założenia:

1. Kodowane pliki mają strukturę bajtową (tzn. mogą zawierać dane z dowolnego problemu, w którym występuje co najwyżej **256** różnych „komunikatów").

2. Długość *bufora słownika* wynosi **256** bajtów, a *bufora wejściowego (look-ahead buffer)* **15** bajtów, tzn. pojedynczy *wskaźnik do słownika* ma rozmiar **2.5** bajta (**1** bajt reprezentujący położenie kopii fragmentu znalezionej w słowniku czyli *offset*, **0.5** bajta do zakodowania *długości kopii* oraz **1** bajt zawierający „komunikat" bezpośrednio po znalezionym fragmencie).

3. Rozwiązanie ma mieć postać DWÓCH gotowych do użycia funkcji/skryptów/aplikacji, tzn. *kodera* i *dekodera*.

4. Koder ma dwa argumenty wejściowe, tzn. (1) nazwa wraz z rozszerzeniem kodowanego pliku oraz (2) nazwa (wraz z dowolnie zdefiniowanym rozszerzeniem, na przykład **.xxx**) pliku zakodowanego, który ma być zapisany w tym samym folderze, co plik kodowany.

5. *Dekoder* ma również dwa argumenty wejściowe, tzn. (1) nazwa (wraz z zastosowanym rozszerzeniem, na przykład **.xxx**) dekodowanego pliku oraz (2) nazwa (wraz z zadanym rozszerzeniem) pliku rozkodowanego, który ma być zapisany w tym samym folderze, co plik dekodowany. Zrealizowane rozwiązania powinny być przetestowane na wybranych przykładach (np. plikach w formacie **.txt**).

2 Szczegóły implementacji

2.1 Koder

Funkcjonalność kodera została zaimplementowana w pliku *lz77_encoder.py*. Aby uruchomić program należy mieć zainstalowany w systemie interpreter języka Python w wersji przynajmniej 2.7 oraz użyć polecenia `python lz77_encoder.py plik_wejsciowy plik_wyjsciowy`. Jeżeli ścieżka do pliku wyjściowego nie będzie zawierała rozszerzenia, skrypt automatycznie przypisze rozszerzenie **.lz77**. Ponadto do polecenia można dodać parametr `-v` lub `--verbose`, który wyświetli w konsoli poszczególne kroki algorytmu LZ77 w formacie ⟨**offset, length, next_char**⟩ , gdzie:

- *offset* to pozycja najdłuższego znalezionego podciągu bufora look-ahead w buforze słownika (licząc od końca bufora słownika, czyli znaku najbliżej początku bufora look-ahead),
- *length* to długość znalezionego najdłuższego podciągu znaków, które należy przepisać z bufora słownika,
- *next_char* to kolejny znak do zakodowania po przepisaniu wcześniej zdefiniowanego podciągu znaków.

Koder po zakończeniu swojej pracy wyświetla także informacje dotyczące stopnia uzyskanej kompresji, procent kompresji oraz średnią bitową. Wartości te zostaną przytoczone w sekcji analizy wyników działania kodera.
Pewną niedogodnością korzystania z języka Python jest fakt, iż operuje on na bajtach, a nie bitach. Z tego powodu dane zapisywane są do plikach po 2 informacje jednocześnie, tak aby uzyskać całkowitą liczbę bitów do zapisu w zakodowanym pliku (w sekcji 1 zaznaczono, że pojedyncza informacja powinna mieć długość **2.5 bajta**).

2.2 Dekoder

Dekoder został zaimplementowany w pliku textitlz77_decoder.py. Działa on analogicznie do kodera, zatem uruchamiany jest poleceniem `python lz77_decoder.py plik_zakodowany plik_wyjsciowy` i tak samo jak koder posiada także opcjonalny parametr `--verbose` wyświetlający odczytane kroki algorytmu LZ77.
Dekoder zapisuje w pamięci zawartość pliku zakodowanego, a następnie pobiera z niego każdorazowo **2.5 bajta** danych, które dekoduje do jednego kroku algorytmu LZ77 i rozkodowany ciąg znaków dopisuje do tablicy danych wyjściowych. Na końcu pracy dekodera dane wyjściowe zapisane są do pliku o wskazanej w poleceniu ścieżce, co owocuje odtworzeniem pierwotnego pliku użytego jako wejście do kodera.

2.3 Zapisane przykłady działania skryptów

W katalogu z projektem znajdują się 3 podkatalogi zawierające zapisane przykłady użycia kodera i dekodera:

- *input* – zawiera pliki wejściowe dla kodera LZ77,
- *encoded* – zawiera pliki z katalogu *input* skompresowane przy użyciu kodera
- *decoded* – zawiera pliki z katalogu *encoded* zdekodowane za pomocą dekodera

Są to pliki opisane w kolejnym rozdziale, które posłużyły do przetestowania poprawności oraz skuteczności zaimplementowanych narzędzi.

3 Wyniki i wnioski

Testy działania kodera i dekodera zostały przeprowadzone głównie na plikach wykorzystywanych podczas wykładów. Jako że na zajęciach skupialiśmy się głównie na obrazach i dźwiękach, do bazy plików wejściowych dołączone zostały również pliki tekstowe: "Lorem ipsum" (5 pierwszych akapitów), tekst powieści Adama Mickiewicza pt. "Pan Tadeusz, czyli ostatni zajazd na Litwie" oraz tekst dotyczący pojęcia *strony obliczeniowej* w tłumaczeniu tekstów (źródło: `http://tlumaczeniamickiewicz.pl/o_stronie_obliczeniowej_2.pdf`).

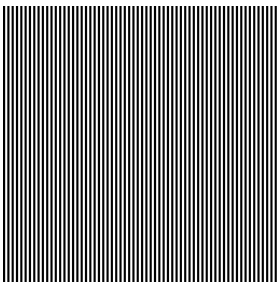
3.1 Kompresja plików tekstowych

1. **Tekst powieści "Pan Tadeusz"** - wyniki kompresji:
   - CR (stopień kompresji) = 1.16203778
   - CP (procent kompresji) = 13.94%
   - BR (średnia bitowa) = 6.8845
2. **"Lorem ipsum"** - wyniki kompresji:
   - CR = 1.29839572
   - CP = 22.98%
   - BR = 6.1614
3. **"O stronie obliczeniowej"** - wyniki kompresji:
   - CR = 1.28762770
   - CP = 22.34%
   - BR = 6.2130

Kompresja tekstów przynosi zatem zysk w postaci około 20%-owego zmniejszenia rozmiaru zakodowanego pliku, co na przykład w przypadku tekstu "Pana Tadeusza" pozwoliło na oszczędność w postaci **65.68kB** powierzchni dyskowej lub transferu danych.

Rysunek 1: SUNFLOW.bmp

Rysunek 2: BOATS.bmp



Rysunek 3: img1.bmp

3.2 Kompresja obrazów (bitmap 8-bitowych)

1. **SUNFLOW.bmp** - wyniki kompresji:
   - CR = 0.86708754
   - CP = -15.33%
   - BR = 9.2263
2. **BOATS.bmp** - wyniki kompresji:
   - CR = 0.94227315
   - CP = -6.13%
   - BR = 8.4901
3. **img1.bmp** - wyniki kompresji:
   - CR = 5.44677024
   - CP = 81.64%
   - BR = 1.4688

Analiza wyników kompresji obrazów algorytmem LZ77 wykazuje, że zakodowane pliki graficzne zazwyczaj mają większy rozmiar niż pliki wejściowe. Prawdopodobnie jest to spowodowane tym, że analizując obraz piksel po pikselu rzadko powtarza się dłuższa sekwencja takich samych wartości następujących po sobie bądź parametry kodera (długość bufora słownika i bufora wejściowego) są nieefektywne do kodowania tego typu informacji.
Warto natomiast zauważyć, że w sytuacji, gdy obraz zawiera powtarzalne wartości (tak jak w pliku *img1.bmp*), to algorytm bardzo efektywnie je zapisuje zmniejszając rozmiar pliku zakodowanego ponad pięciokrotnie. Niestety, w praktyce takie bitmapy pojawiają się niezwykle rzadko, choć algorytm ten może znaleźć zastosowanie w kodowaniu obrazów zawierających kody kreskowe bądź kody QR.

3.3 Kompresja sygnału audio (32-bit float, mono, 44100Hz)

Co prawda implementacja została przygotowana do pracy z plikami zawierających 8-bitowe informacje o wartościach całkowitych, jednak działa ona bezpośrednio na danych w postaci binarnej. Postanowiłem zatem sprawdzić zachowanie zaimplementowanego kodera LZ77 w pracy z plikami przeznaczonymi dla poprzedniego projektu.

1. **icing.wav** - wyniki kompresji:
   - CR = 0.76259250
   - CP = -31.13%
   - BR = 10.4905
2. **"music2.wav"** - wyniki kompresji:
   - CR = 0.73465301
   - CP = -36.12%
   - BR = 10.8895
3. **"song.wav"** - wyniki kompresji:
   - CR = 0.76278970
   - CP = -31.10%
   - BR = 10.4878

Wyniki kompresji są tragiczne - zakodowane pliki audio są o ponad 30% większe niż oryginały. Zdecydowanie nie jest to sposób na kompresję tego typu sygnałów w tym formacie. Trudno jednak dziwić się takim wynikom - algorytm został zaprojektowany do pracy z 8-bitowymi informacjami, a ponadto, jak wynika z poprzednich doświadczeń, głównie z tekstami.
Warto jednak zaznaczyć, że z uwagi na działanie algorytmu bezpośrednio na danych binarnych zdekodowane pliki są zgodne z oryginalnymi mimo niekompatybilności dotyczącej formatu.

4 Podsumowanie

Zaimplementowana wersja kodera LZ77 z buforem słownika o rozmiarze **256 bajtów** oraz **15-bajtowym** buforem wejściowym dobrze radzi sobie z kompresją plików tekstowych oraz bitmap o dużej powtarzalności następujących po sobie sekwencji jasności. Nie nadaje się ona natomiast do kompresji obrazów fotorealistycznych bądź fotografii oraz wszelakich sygnałów audio. Najprawdopodobniej można jednak dostosować algorytm kompresji LZ77 do pracy z takimi plikami poprzez manipulację wielkością obu buforów oraz zwiększeniem pojedynczej kodowanej informacji.