Maciej Szeptuch

Kompresja Danych 2015

Implementacja LZ78/LZW

Implementacja kompresji algorytmami LZ78/LZW (powstałe ciągi par/indeksów dodatkowo kodowane dynamicznym Huffmanem).

Użytkowanie

Obsługa programów z linii komend podobnie jak w przypadku linuksowych gzip/bzip2 itd.

```
lzw/lz78 [OPCJE]... [PLIK]
```

dostępne **OPCJE**:

- -c / --stdout Wypisywanie wyniku na standardowe wyjście
- -b / --bitsize Rozmiar słownika, maksymalna liczba bitów na indeks (15-31, domyślnie=20)
- -d / --decompress Rozpakuj podany plik
- -f / --force Nadpisz plik wynikowy
- -q / --quiet Wyłącz wypisywanie wszystkich informacji diagnostycznych
- -v / --verbose Włącz wypisywanie wszystkich możliwych informacji diagnostycznych (UWAGA: może tego być bardzo dużo)

Jeśli nie poda się **PLIK**u albo `-` - będzie kompresować standardowe wejście.

Algorytmy

Krótki przegląd wykorzystanych algorytmów.

LZ78

Słownikowa metoda bezstratnej kompresji danych, opracowana w 1978 roku przez Jacoba Ziva i Abrahama Lempela. Polega na zastępowaniu ciągów symboli parami: indeks do słownika przechowującego poprzednie ich wystąpienia oraz dodatkowa litera. Dzięki temu wielokrotnie powtarzające się ciągi symboli (np. te same słowa, czy frazy w tekście) są zastępowane o wiele krótszymi indeksami (liczbami). Dużą zaletą tej metody jest to, że potencjalnie bardzo dużego słownika w ogóle nie trzeba zapamiętywać – zostanie on odtworzony przez dekoder na podstawie zakodowanych danych. Jednak pewną wadą jest praktycznie jednakowa złożoność kodu kompresującego i dekompresującego.

LZW

Algorytm opisał w 1984 roku Terry A. Welch. Jest to modyfikacja **LZ78**.

Słownik jest na samym początku wypełniany wszystkimi symbolami alfabetu. Gwarantuje to, że zawsze da się znaleźć przynajmniej jednoliterowe dopasowanie. A to pozwala to na skrócenie wyjścia kodera - wypisywany jest wyłącznie indeks słowa w słowniku.

Dynamiczne kodowanie Huffmana

Opracowane w 1952 roku przez Davida Huffmana. Polega na utworzeniu słów kodowych (ciągów bitowych), których długość jest odwrotnie proporcjonalna do prawdopodobieństwa występowania danego symbolu. Tzn. im częściej dany symbol występuje w ciągu danych, tym mniej zajmie bitów. Dynamiczne kodowanie pozwala na kodowanie danych o nieznanej statystyce. Główną zaletą jest to, że nie ma potrzeby przesyłania drzewa kodów. Zamiast tego identyczną procedurę poprawiania drzewa muszą przeprowadzać zarówno koder, jak i dekoder. Do poprawiania drzewa zastosowałem algorytm **FGK** (Fallera-Gallera-Knutha) który zachowuje następujące niezmienniki:

- każdy wierzchołek drzewa(oprócz liści) ma zawsze dwóch potomków
- każdy wierzchołek przechowuje licznik: sumę liczby wystąpień symbli z danego poddrzewa. Dla liści poprostu liczba wystąpień danego symbolu
- przy przejściu drzewa wszerz od prawej do lewej liczniki tworzą ciąg nierosnący

Słownik

Głównym komponentem algorytmów LZ78/LZW jest słownik przechowujący poprzednie wystąpienia ciągów. Struktura którą zaimplementowałem, używa linowej pamięci ze względu na liczbę symboli w słowniku oraz pozwala na przeszukiwanie (znajdywanie ciągu przedłużonego o jedną literkę) w czasie stałym. W uproszczeniu słownik przechowuje elementy (**indeks**, **symbol**, **odpowiednie połączenia**…) gdzie indeks to indeks w słowniku reprezentujący prefiks danego ciągu bez ostatniego symbolu.

Szczegóły implementacji

Zaimplementowałem algorytmy LZ78, LZW, dynamicznego Huffmana wykorzystując FGK oraz obsługę bitowego wejścia/wyjścia. Algorytmy LZ78/LZW same w sobie nie są specjalnie skomplikowane, implementacja FGK również. Jedyne o czym można coś ciekawego powiedzieć to struktura danych służąca za słownik dla tych algorytmów..

Implementacja słownika

Słownik składa się z elementów **<indeks prefiksu**, **symbol**, **następnik**, **lewe_dziecko**, **prawe_dziecko>** które reprezentują ciągi. Następnik to indeks pierwszego ciągu którego dany jest prefiksem. Lewe i prawe dziecko definują proste drzewo binarne (niezbalansowane, ale można jak ktoś bardzo chce) tych elementów które mają taki sam prefiks ale inny symbol. Żeby dodać nowy symbol do danego ciągu wystarczy teraz przejść do następnika i zgodnie z symbolem przechodzić odpowiednio do prawego lub lewego dziecka

aż go nie znajdziemy (lub dodać go w odpowiednim miejscu). Jako że rozmiar alfabetu jest stały czas takiego przejścia też jest stały - w najgorszym przypadku musimy przejść wszystkie inne symbole niż ten którego szukamy, ale dzięki temu że jest to drzewo to w średnim przypadku wykonamy ich znacznie mniej. Z ciekawostek implementacyjnych - w związku z tym że w LZW pusty słownik to dla nas tak naprawdę słownik z całym alfabetem, dane w tej strukturze są zapisane tak, że mam osobne tablice na symbol, indeks itd. a nie jak normalnie tablice krotek. Sprawia to, że "wyczyszczenie" słownika sprowadza się do zmiany rozmiaru wszystkich tabel i wyzerowania jednej z nich (do czego można użyć memset/bzero w C/C++) co jest wydajniejsze niż przechodzenie po tablicy struktur i ręczne ustawianie wartości i zdecydowanie wydajniejsze niż wyczyszczenie całej struktury i dodanie tych początkowych symboli od nowa.

Dane testowe

LZ78/LZW w teorii powinny dobrze sprawdzać się w warunkach kiedy w danych występuje dużo powtarzających się ciągów. Dużo powtarzających się ciągów na pewno występuje w tekstach, oraz wydaje się że w obrazkach (te same kolory). W związku z tym, korzystając ze stron z testami http://prize.hutter1.net/ oraz http://www.maximumcompression.com/ wybrałem kawałek angielskiej wikipedii, tekst w języku angielskim, logi serwera www. Dla testów sprawdziłem także jak poradzą sobie ze słownikiem języka angielskiego oraz obrazkiem BMP.

Wyniki eksperymentów

Wykresy i wyniki w pliku doc/wykresy.pdf

Algorytmy zachowują się podobnie - w końcu wiele się nie różnią. Zwiększanie rozmiaru słownika pozwala zazwyczaj osiągnąć lepszy stopień kompresji zazwyczaj o paredziesiąt procent. Zwiększanie rozmiaru pozwala także trochę przyspieszyć działanie algorytmu przynajmniej do momentu w który koszty skakania po pamięci nie przekroczą kosztów czyszczenia słownika. Dzieje się tak we wszystkich przypadkach poza jednym, w trakcie kompresji słownika (english.dic) większy limit psuje stopień kompresji o parę procent. Dzieje się tak zapewne dlatego, że w tym przypadku słownik zawiera słowa występujące w tym pliku i ich prefiksy - żadne się nie powtarza i są oddzielone znakami nowej lini - a kiedy jest mały i często czyszczony pozwala także na zapamiętanie niektórych sufiksów co może i jak widać poprawia nieznacznie stopień kompresji.

Dekompresja w przypadku zarówno LZ78 i LZW jest około dwóch razy wolniejsza - dzieje się tak dlatego że aby wypisać słowo odpowiadające danemu kodowi trzeba przejść po strukturze słownika wstecz - otrzymuje się słowo zapisane od tyłu, więc trzeba je jeszcze odwrócić aby wypisać. Możnaby to trochę przyspieszyć trzymając tymczasowy cache słów odpowiadających kodom ale to kosztowałoby też więcej pamięci.

Z różnic pomiędzy algorytmami: LZW jest oczywiście bardziej efektywny, w testach średnio o około 8% bez większej zmiany w czasie kompresji, za to z

około 10% pogorszeniem czasu dekompresji. Z paru testów jakie wykonałem wynika, że najprawdopodobniej jest to związane z wczytywaniem bit po bicie - jak wyrówna się rozmiary wypisywanych indeksów do pełnych bajtów (kosztem stopnia kompresji) czas działania dekompresji staje się bardzo zbliżony (mimo tego że rozmiar wejścia jest wtedy większy).

Co do dobrze kompresujących się danych: jak można się było spodziewać kompresowanie logów serwera www dało najlepsze rezultaty (plik wynikowy około 9% oryginalnego pliku dla lzw oraz 13% dla lz78), wszystko się tam praktycznie co chwile powtarza. Natomiast najgorszy wynik, ale nadal całkiem przyzwoity, uzyskany został dla słownika (około 40% dla lzw oraz 50% dla lz78), w którym niewiele się powtarza.

Źródła

- http://pl.wikipedia.org/wiki/LZ78
- http://pl.wikipedia.org/wiki/LZW
- http://pl.wikipedia.org/wiki/Kodowanie_Huffmana
- http://warp.povusers.org/EfficientLZW/
- http://prize.hutter1.net/
- http://mattmahoney.net/dc/enwik8.zip - brakujący plik enwiki8 z testdata (100MB)
- http://www.maximumcompression.com/