Kompresja danych

Implementacja BWT oraz algorytmów dodatkowych

Jakub Machoń, Łukasz Banaśkiewicz, Kacper Szkudlarek

16 stycznia 2011

Streszczenie

Transformata Burrowsa-Wheelera to algorytm użyteczny przy bezstratnej kompresji danych. Dane po przetworzeniu tą transformacją dają się znacznie lepiej skompresować za pomocą klasycznych algorytmów kompresji. W ramach projektu powstała implementacja transformaty oraz kilku algorytmów drugiego kroku wykorzystywanych razem z BWT.

1 Wstęp

2 Implementacja

W ramach projektu zaimplementowane zostały algorytm transformaty Burrowsa-Wheelera a także zestaw popularnych algorytmów drugiego kroku. Każdy z algorytmów został opisany w kolejnych rozdziałach.

2.1 Transformata Burrowsa-Wheelera

Transformata Burrowsa-Wheelera nie jest metodą kompresji, a jedynie sposobem modyfikacji danych (położenia poszczególnych bajtów). Po modyfikacji dane wyjściowe zawierają zazwyczaj ciągi równych sobie bajtów umieszczonych po sobie. Tak ułożone dane lepiej poddają się kompresji. Algorytm transformaty Burrowsa-Wheelera zaimplementowano zgodnie z opisem w [3].

Strumień danych wejściowych dzielony jest na bloki o rozmiarze będącym parametrem algorytmu. Dane w każdym bloku sortowane są metodą sortowania przedrostków (ang. *Suffix Sorting*) - opis algorytmu sortowania znajduje się w kolejny akapicie. Wynikiem sortowania jest macierz indeksów sortowanych bajtów. Dane wyjściowe tworzone są poprzez przeglądanie posortowanej macierzy indeksów - jako wynik zastosowania transformaty należy zwrócić dane skonstruowane poprzez kolejne wypisywanie bajtów z pozycji *i-1*, gdzie *i* jest wartością zapisaną w macierzy indeksów. Dla *i=0* należy wypisać ostatni bajt z bloku. Do tak wypisanych danych należy dołączyć liczbę wskazującą pozycję na której znalazł się bajt o indeksie 1 z bloku wejściowego.

W implementacji Transformaty Burrowsa-Wheelera najbardziej problematycznym krokiem algorytmu jest sortowanie bajtów metodą sortowania przedrostków. Krok ten jest problematyczny ze względu na swoją złożoność czasową. Algorytm sortowania spowodował również najwięcej problemów podczas implementacji transformaty. W projekcie po dokonaniu przeglądów algorytmów sortowania przedrostkowego zdecydowano się zaimplementować algorytm Itoh'a (wg opisu w [2] (thesis.ps)).
Algorytm sortowania składa się z 3 kroków:

1. W pierwszym kroku należy podzielić dane na dwa typy: X i Y. Wykonujemy to przeglądając bajty w bloku i jeśli obecnie analizowany bajt jest leksykograficznie po następnym bajcie, wtedy zaznaczamy go jako należący do X. W przeciwnym przypadku jest on typu Y. Ostatni bajt z bloku należy porównywać z pierwszym bajtem tego samego bloku. (Rys. 1)

2. W drugim kroku sortujemy dane przez zliczanie, umieszczając w tablicy najpierw dane typu X, a później typu Y. Dane typu Y sortujemy zmodyfikowanym algorytmem Radix Sort, jeśli dla danego symbolu jest więcej elementów typu Y, niż 1. Modyfikacja algorytmu Radix Sort polega na tym, iż rozpoczyna on analizowanie danych od najbardziej znaczących bajtów, a nie tak jak w swej standardowej wersji - on najmniej znaczących. Dzięki temu poza przypadkami długich ciągów takich samych bajtów nie ma potrzeby analizowania całego bloku dla każdego sortowanego bajtu. Algorytm zaimplementowano iteracyjnie - z użyciem stosu umieszczonego na stercie.

3. W trzecim kroku następuje łączenie koszyków w następujący sposób - przeglądamy częściowo posortowany blok wejściowy (posortowane są tylko dane typu Y). Jeśli dla i będącego pozycją elementu z posortowanego bloku w pierwotnym bloku (wejściowym), bajt na pozycji *i-1* jest typu X, to należy go umieścić na pozycji, na której powinien się znajdować ze względu na swoją wartość i przesunąć wskaźnik dla tej wartości.

Niewątpliwą zaletą zastosowanego algorytmu jest fakt, iż występujące w kroku 3 łączenie koszyków ma złożoność liniową, co oznacza, że bajty z koszyka X zostały posortowane w czasie liniowym. Pewną wadą przyjętego rozwiązania jest to, iż w zależności od charakteru danych, spora ich część trafia do koszyka Y, który jest już sortowany algorytmem Radix Sort - podatnym na głęboką rekurencję.

Dane:	S	V	E	E	A	E	E	V
Typ:	Y	X	Y	X	Y	Y	Y	X
Indeks:	0	1	2	3	4	5	6	7

Rysunek 1: Przykład - podział danych wejściowych na typy

Podczas dekodowania danych (transformata odwrotna - przywracająca pierwotne ułożenie bajtów w bloku) stosowane jest sortowanie przez zliczanie - ponieważ podczas dekodowania wystarczy posortować bajty leksykograficznie (ściślej: tworzona jest tablica indeksów wskazujących na posortowane dane). W związku z tym dekodowanie danych jest dużo szybsze, niż kodowanie - algorytm sortowania przez zliczanie ma złożoność liniową

Rysunek 2: Przykład - sposób wstawiania bajtów typu X do posortowanej tablicy. W pogrubionej ramce znajduje się element powodujący wstawienie pogrubionego elementu typu X.

w stosunku do długości bloku. Pozostałe działania wykonywane podczas dekodowania również posiadają liniową złożoność.

W transformacie bardzo ważny jest algorytm sortowania, ponieważ błędne posortowanie danych powoduje przekłamania w odkodowanych danych.

2.2 RLE0 i RLE2

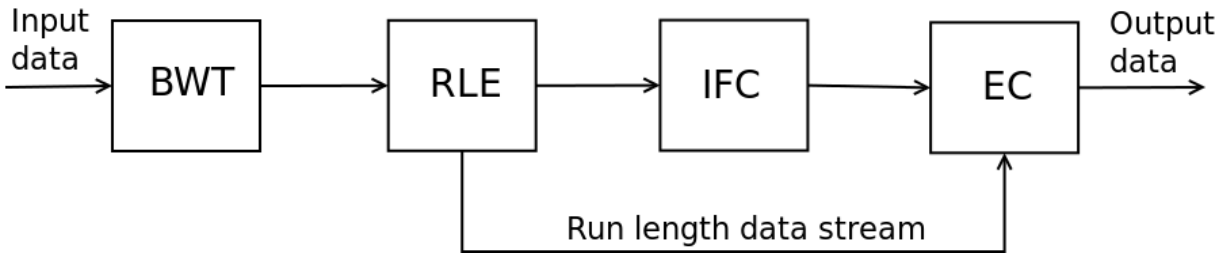
W projekcie użyte zostały dwie odmiany algorytmu kodowania długości serii - RLE0 i RLE2. Podstawową różnicą w prezentowanych wersjach algorytmów jest sposób i miejsce zapisu informacji o długości znalezionego ciągu danych.

2.2.1 RLE0

RLE0 [2] jest najprostszą i najczęścej stosowaną implementacją tego algorytmu. Algorytm analizuje plik wejściowy znak po znaku i każdy ciąg znaków, zapisuje w postaci znaku reprezentującego ten ciąg i jego długości. Algorytm dzięki swej prostocie jest szybki, jednak w niesprzyjających warunkach prowadzi do wydłużenia danych zamiast ich kompresji.

2.2.2 RLE2

Podstawową różnicą pomiędzy RLE0 i RLE2 [1] jest sposób przechowywania informacji o długości znalezionego ciągu. Algorytm analizuje dane wejściowe znak po znaku i w wypadku wykrycia ciągu o długości $n > 1$ zamienia taki ciąg na ciąg o długości dokładnie dwóch znaków. Natomiast do drugiej tablicy zapisywana jet rzeczywista długość ciągu. Wykorzystanie RLE2 razem z algorytmem IFC (2.3) zostało opisane w [1].



Rysunek 3: Algorytm kompresji BWT z RLE2 i IFC

2.3 Incremental Frequency Count

Algorytm IFC (ang. Incremental Frequency Count) został stworzony i przez Jurgena Abela. Algorytm powstał by zaoferować skuteczną metodę przetwarzania danych otrzymanych z BWT. Założeniem było stworzenie algorytmu szybkiego i dającego dobre efekty kompresji. Algorytm został opisany w [1]. Daje on czasy działania porównywalne z MTF, przy jednoczesnym wysokim stopniu kompresji na poziomie algorytmu Sebastiana Deorowicza - WFC (ang. Weighted Frequency Count) [2].

Zadaniem algorytmu drugiej fazy kompresji BWT jest takie przetworzenie struktury z symboli otrzymanych z BWT by była ona łatwa do skompresowania przy użyciu kodeków entropijnych. W celu obliczania wartości wyjściowych w algorytmie IFC każdemu symbolowi przyporządkowywany jest licznik. Wszystkie licznik przechowywane są w kolejności malejącej. Dla każdego odczytanego z wejścia symbolu na wyjście wypisywana jest wartość licznika odpowiadającego temu symbolowi, po czym następuje przeliczenie wartości licznika. Podstawową ideą jest tutaj inkrementacja adaptacyjna bazująca na wykorzystaniu informacji o elementach które już się pojawiły. Liczniki poszczególnych elementów alfabetu mogą być inkrementowane lub dekrementowane w zależności od częstości występowania danego elementu. Powoduje to, że element występujące często mają niższe indeksy. Wartości liczników są co jakiś czas normowane. Główny nacisk kładziony jest w algorytmie na obliczanie wartości inkrementacji liczników. Algorytm podzielony jest na pięć części:

1. Wypisanie wartości odpowiedniego licznika na wyjście.
2. Obliczenie różnicy dla średniej wartości liczników dla poprzedniego i obecnego znaku. Wartości średnie obliczane są w przesuwnym oknie wg. wzoru:
   $avg_i := \frac{(avg_{i-1}\dot{(}window\_size-1))-index_{x_i})}{windows\_size}$
3. Obliczenie wartości inkrementacji na podstawie różnicy średnich i wartości inkrementacji poprzedniego elementu:
   $inc_i := inc_{i-1} - \frac{inc_{i-1}dif_i}{q}$, gdzie q jest czynnikiem skalującym inkrementacji.
4. Przeskalowanie liczników jeżeli któryś z nich przekroczył wartość graniczną.
5. Ponowne posortowanie liczników. Sprowadza się to do odpowiedniego przesunięcia licznika elementu, który ostatnio pojawił się na wejściu.

2.4 Distance Codding

Algorytm kodowania dystansowego (ang. Distance Coding - DC) [2] został zaprezentowany w 2000 roku przez Edgara Bindera. Nie ma żadnej oficjalnej publikacji na jego temat. Został on zaproponowany na jednej z grup dyskusyjnych o kompresji danych.

Dla każdego symbolu wejściowego x_i^{bwt} algorytm znajduje odległość do jego następnego wystąpienia i ją zapisuje na wyjście. Jeżeli jest to ostanie wystąpienie elementu w ciągu wejściowym to na wyjście wypisywane jest 0. Do poprawnego odczytania o ponownego zdekodowania zbioru wyjściowego niezbędna jest znajomość pozycji początkowych wszystkich elementów alfabetu występujących w analizowanym ciągu.

2.5 Move To Front

Move To Front (MTF) [2] jest to prosta transformacja strumieniowa, której celem jest zmniejszenie entropii w kodowanym ciągu danych. Algorytm ten jest zalecany przez twórców BWT jako algorytm drugiego stopnia, najlepiej współpracujący z BWT. Algorytm operuje na danych wejściowych i liście L zawierającej wszystkie elementy alfabetu. Początkowo lista L jest posortowana w pewien ustalony sposób. Algorytm odczytuje z wejścia kolejne symbole ciągu wejściowego i dla każdego z nich na wyjście wyprowadzana jest pozycja odpowiadająca temu elementowi w liście L. Następnie lista jest modyfikowana tak, że element, który wystąpił jako ostatni wędruje na pierwszą pozycję. Taka modyfikacja powstanie ciągów powtarzających się elementów na wyjściu. Np. wszystkie ciągi jednakowych elementów zostaną zastąpione przez ciągi zer i jednej dodatkowej liczby stanowiącej pozycję początkową znaku. Do zdekodowania otrzymanego ciagu wyjściowego potrzebna jest jedynie wiedza na temat początkowego posortowania listy L zawierającej alfabet.

2.6 Invesrion Frequencies

Algorytm IF (ang. Inversion Frequencies) [2] został zaproponowany przez Arnavut i Magliveras w roku 1997. Jest on stosowany jako zamiennik MTF (2.5) w algorytmie kompresji BWT. Algorytm ten jako wyjście podaje sekwencje liczb z przedziału $[0; n-1]$, gdzie n jest ilością elementów w ciągu wejściowym. Dla każdego elementu a_i wykorzystywanego alfabetu algorytm skanuje sekwencje wejściową. W momencie pierwszego napotkania aktualnie analizowanego znaku a_i algorytm wypisuje jego pozycję. Dla każdego kolejnego wystąpienia na wyjściu zapisywana jest liczba będąca ilością znaków większych niż analizowany element a_i, która pojawiła się od jego ostatniego wystąpienia. Powstała w ten sposób sekwencja nie jest niestety wystarczająca do odkodowania danych wejściowych. Drugim niezbędnym elementem jest lista zawierająca ilość wystąpień każdego ze znaków w analizowanym ciągu.

3 Testy

3.1 Aplikacja testująca

3.2 Testy wydajności

3.2.1 Dane testowe

3.2.2 Wyniki

Literatura

[1] Jurgen Abel. Incremental frequency count - a post bwt-stage for the burrows-wheeler compresion algorithm. *SOFTWARE - PRACTISE AND EXPERIENCE*, 2006.

[2] Sebastian Deorowicz. Second step algorithm in the burrows-wheeler compression algorithm. *SOFTWARE - PRACTISE AND EXPERIENCE*, 2001.

[3] Artur Przelaskowski. *Kompresja danych*. Wydawnictwo BTC, 2005.