Kompresja danych

Implementacja BWT oraz algorytmów dodatkowych

Jakub Machoń, Łukasz Banaśkiewicz, Kacper Szkudlarek

17 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

Wg. definicji zawartej na portalu Wikipedia kompresja danych "polega na zmianie sposobu zapisu informacji tak, aby zmniejszyć redundancję i tym samym objętość zbioru." W dzisiejszym świecie zagadnienie kompresji staje się co raz ważniejsze. Ludzkość generuje co raz to większe zbiory danych, których archiwizacja staje się niezbędna. Dlatego właśnie kompresja jest prężnie rozwijającą się gałęzią nauki, której wymierne wyniki widzimy na co dzień. Można wyróżnić dwa podstawowe typy kompresji:

1. Kompresja stratna - nie pozwala ona na dokładne odtworzenie danych, jednak interesujące nas informacje zachowane są w stopniu pozwalającym na ich odczytanie, najlepszym przykładem tego typu kompresji są cyfrowe obrazy.

2. Kompresja bezstratna - pozwala na dokładne odtworzenie kompresowanych danych, dzięki czemu mogą one być ponownie przetwarzane.

Opisana w tym raporcie implementacja algorytmu kompresji BWT (ang. BWCA - Burrows-Wheeler Compression Algorithm) należy do tej drugiej kategorii. Ze względu na są prostotę działania i dobre efekty kompresji wykorzystywana jest m.in. w popularnym algorytmie kompresji BZIP2.

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.

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

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

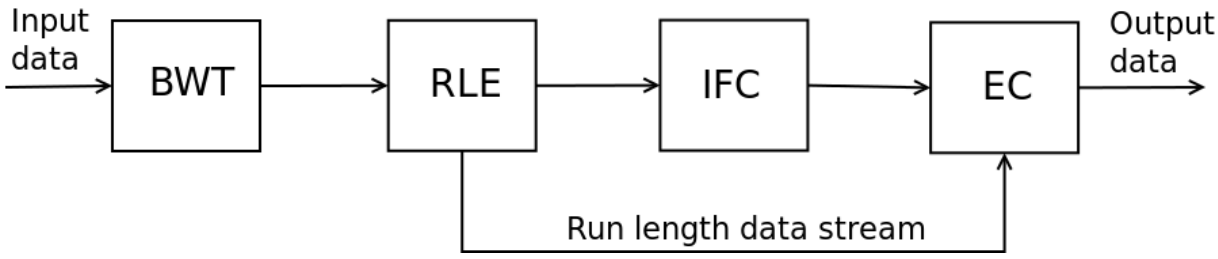
Rysunek 1: Przykład - podział danych wejściowych na typy

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 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.



Rysunek 3: Algorytm kompresji BWT z RLE2 i IFC

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].

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:
   $avg_i := \frac{(avg_{i-1}(window\_size-1))-index_{x_i})}{windows\_size}$
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

W ramach projektu została stworzona aplikacja przeprowadzająca testy algorytmu i wizualizująca wyniki kodowania. Jedną z jej części jest program napisany w języku C++, odpowiedzialny za uruchomienie właściwej aplikacji kodującej i pomiar czasu wykonania, którego wynik przekazywany jest do aplikacji języka Java, pozwalającej na wybór zestawu testów, parametrów algorytmów oraz prezentację i porównanie wydajności algorytmów. Wybór możliwy jest spośród 9 wariantów pracy stworzonego w ramach projektu kodera oraz algorytmu Bzip2 w standardowej implementacji dla systemu Linux. Dodatkowo dla naszego kodera istnieje możliwość ustawienia parametru — rozmiaru bloku BWT — od 1 do 8192 kB.

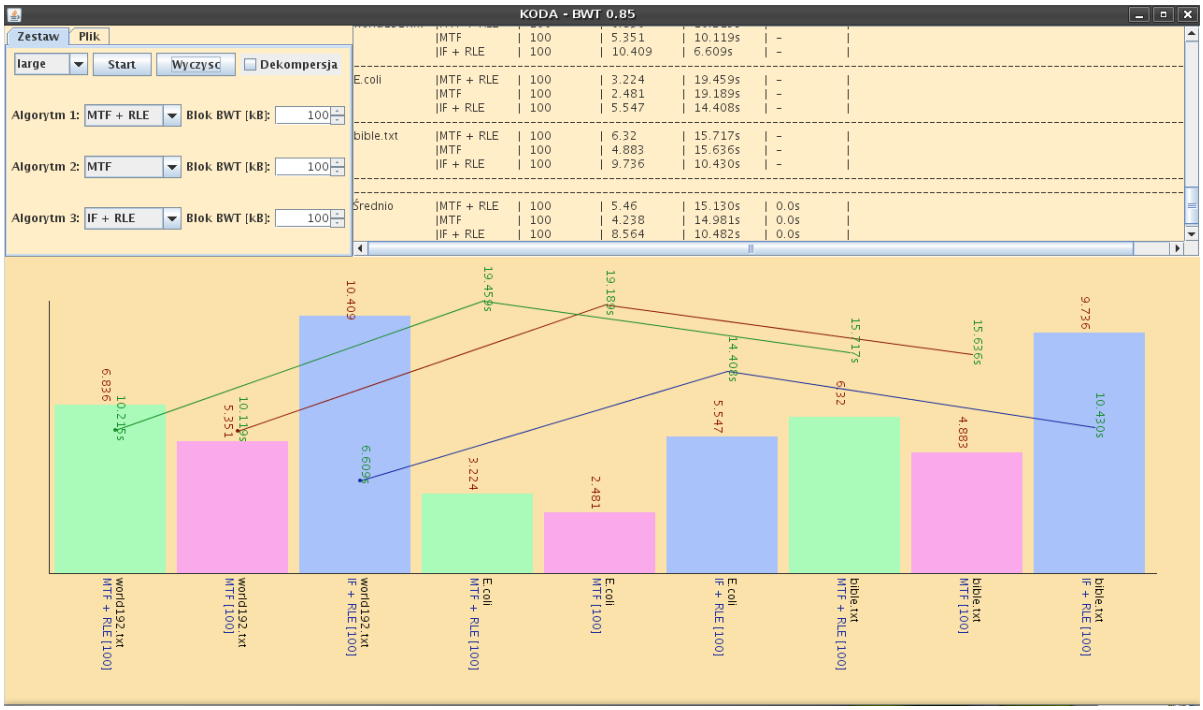
Główne okno aplikacji składa się z trzech części — w lewym górnym rogu znajdują się zakładki sterujące pracą aplikacji, w prawym górnym roku widoczna jest konsola tekstowa, a w dolnej części prezentowane są wykresy średniej bitowej oraz czasu kodowania kolejnych plików testowych dla wybranych algorytmów.

Zakładka *Plik* pozwala na przeprowadzenie pojedynczego testu. 3 przyciski pozwalają na wybór pliku, przeprowadzenie kompresji i dekompresji. Możliwy jest również wybór algorytmu i rozmiaru bloku BWT. Dodatkowo dla pliku przed kompresją, po kompresji oraz po dekompresji prezentowana jest nazwa, rozmiar oraz wyliczona suma CRC32, dzięki czemu można szybko stwierdzić, czy operacje zostały przeprowadzone prawidłowo. Po przeprowadzeniu kompresji jej czas i uzyskana średnia bitowa nanoszona jest na wykres. Informacje te można także przeczytać z konsoli.

Zakładka *Zestaw* pozwala na przeprowadzenie testu 3 wybranych algorytmów na podstawie plików zawartych w jednym z czterech zbiorów testowych. Dla każdego algorytmu istnieje możliwość indywidualnego ustawienia rozmiaru bloku BWT. W czasie przeprowadzania testu wykres oraz informacje na konsoli uaktualniane są na bieżąco. Po zakończeniu wszystkich operacji generowana jest tabela zawierająca zestawienie wyników i

wyliczenie średnich poszczególnych algorytmów dla wszystkich plików. Jeśli opcja Dekompresja zostanie zaznaczona przeprowadzona zostanie także procedura dekompresji i czasy jej wykonania zostaną zebrane w tabeli.

Na wykresie trzema kolorami przedstawione są rezultaty poszczególnych algorytmów. Wysokość słupka wyraża względną średnią bitową, natomiast wartości czasu kompresji łączone są linią tworząc wykres liniowy na każdego algorytmu z osobna. Dodatkowo pod słupkiem znajduje się informacja o nazwie kompresowanego pliku, wybranym algorytmie oraz rozmiarze bloku BWT.



Rysunek 4: Ekran główny aplikacji testującej

3.2 Testy wydajności

3.2.1 Dane testowe

Zbiór *artificl* zawiera cztery sztucznie wygenerowane pliki zawierające odpowiednio: jedną literę a *(a.txt)*, powtarzające się litery a *(aaa.txt)*, powtarzający się alfabet *(alphabet.txt)* oraz litery w kolejności pseudolosowej *(random.txt)*. Może być on pomocny w sprawdzeniu jak algorytmy zachowują się w najgorszym możliwym scenariuszu (losowe znaki, bardzo mały plik).

Zbiór *calgary* zawiera 18 plików różnego typu stanowiących *The Calgary Corpus*, standard dla badania wydajności bezstratnej kompresji w latach 90. ubiegłego wieku.

Zbiór *cantrbry* zawiera 11 plików stanowiących *The Canterbury Corpus*. Jest to zbiór utworzony w 1997 roku jako udoskonalenie poprzedniego zbioru. Pliki zostały tak dobrane, że ich rezultat przy wykorzystaniu ówczesnych algorytmów kompresji był reprezentatywny i stąd nadzieja, że będzie on taki także dla nowych metod kompresji.

Zbiór *large* zawiera 3 duże pliki (od 2 do 4,4 MB każdy) — kompletny genom bakterii E. Coli, jedna z wersji Biblii oraz opis krajów świata — *The CIA world fact book*.

3.2.2 Wyniki

Zbiór large Pierwszym przeprowadzonym testem był test wszystkich wersji algorytmu dla zbioru *large*. Rozmiar bloku BWT ustawiony został na wartość równą 4 MB, czyli nieco mniejszą niż rozmiar największego pliku.

Dla tych plików najlepszym okazał się referencyjny algorytm Bzip2, jednak pod względem wartości średniej bitowej niewiele gorszy okazał się algorytm MTF. Różnica wynosi jedynie 15 proc. Następne miejsce zajął algorytm MTF+RLE, a kolejne algorytmy wykorzystujące IFC. Dużo gorzej wypadły metody IF i DC.

Niestety pod względem czasu trwania kompresji nasz algorytm znacznie odbiegał od Bzip2. Najdłużej działała metoda, która uzyskała najlepszy wynik pod względem wartości średniej bitowej (MTF) i czas ten był aż 32 razy dłuższy od referencyjnego Bzip2.

Badanie wpływu rozmiaru bloku BWT Algorytm, który uzyskał najlepszą średnią bitową w poprzednim teście posłużył do wykonania testu wpływu wartości parametru rozmiaru bloku BWT na jakość kompresji. Zgodnie z przewidywaniami, im większa wartość bloku tym algorytm BWT lepiej radzi sobie z kompresją, ale wzrasta długość trwania kompresji. Ograniczeniem jest długość samego pliku poddawanego poszczególnym operacjom. W dalszych testach wartość tego parametru pozostała na poziomie 4 MB, gdyż tylko rozmiar jednego z testowanych plików przekracza tą wartość, a dalszy wzrost tego parametru poprawia średnią bitową dla tego pliku o mniej niż jeden promil.

Zbiór canterbry Po przeprowadzeniu testów na zbiorze *canterbry* dla 3 najlepszych algorytmów okazało się, że dla niektórych plików jeden z wariantów naszego algorytmu (MTF+RLE) może uzyskiwać mniejszą średnią bitową niż referencyjny Bzip2. Stało się tak dla pliku arkusza kalkulacyjnego *kennedy.xls*. Dodatkowo dla kilku innych plików uzyskaliśmy krótszy czas kompresji (*asyoulik.txt, xargs.1, grammar.lsp*).

Niestety, nasz algorytm dla niektórych plików uzyskuje czas kompresji rzędu kilku minut, mimo że nie są to duże pliki. Związane jest to z algorytmem sortowania użytym w BWT.

Zbiór calgaryS Po usunięciu ze zbioru calgary pliku *pic*, dla którego czas kompresji wynosił 10 minut, powstał zbiór *calgaryS*, dla którego przeprowadziliśmy testy wszystkich algorytmów. Średnie wyniki zostały zebrane w poniższej tabeli. Zamieszczone zostały także wykresy wartości średniej bitowej i czasu kompresji. Kolejność pod względem wartości średniej bitowej odpowiada kolejności dla zbioru *large*. W tym teście zostały dodatkowo zbadane czasy trwania dekompresji. Jedynie dla algorytmu RLE-2 + IFC czas dekompresji dwóch plików wyniósł 30 sekund, co spowodowało znaczący wzrost średniej. Pozostałym algorytmom dekompresja zajmowała najczęściej mniej niż jedną sekundę.

Zbiór artificl Przeprowadzony został także test dla zbioru plików sztucznych. Algorytm MTF uzyskał najlepsze czasy kompresji dla pliku zawierającego pojedynczą literę

large				
plik	**algorytm**	**blok [kB]**	**śr. bitowa**	**kompr. [s]**
world192.txt	Bzip2	4000	**1.583**	**0.935s**
	MTF + RLE	4000	1.804	29.421s
	MTF	4000	1.957	29.694s
	IF + RLE	4000	2.712	26.746s
	IF	4000	4.981	26.459s
	DC + RLE	4000	8.927	25.512s
	DC	4000	6.487	24.891s
	RLE-2 + IFC	4000	1.949	34.557s
	RLE-0 + IFC	4000	1.985	26.821s
	IFC	4000	2.569	26.734s
E.coli	Bzip2	4000	**2.157**	**1.664s**
	MTF + RLE	4000	2.829	66.329s
	MTF	4000	2.209	67.245s
	IF + RLE	4000	4.897	61.990s
	IF	4000	5.328	63.581s
	DC + RLE	4000	9.895	59.992s
	DC	4000	7.263	58.907s
	RLE-2 + IFC	4000	3.225	60.267s
	RLE-0 + IFC	4000	3.825	59.656s
	IFC	4000	2.17	60.523s
bible.txt	Bzip2	4000	**1.671**	**1.312s**
	MTF + RLE	4000	2.095	29.769s
	MTF	4000	2.084	29.665s
	IF + RLE	4000	3.131	25.330s
	IF	4000	5.072	25.481s
	DC + RLE	4000	9.186	23.188s
	DC	4000	6.697	22.796s
	RLE-2 + IFC	4000	2.25	24.448s
	RLE-0 + IFC	4000	2.367	23.67s
	IFC	4000	2.348	23.464s
Średnio	Bzip2	4000	**1.803**	**1.303s**
	MTF + RLE	4000	2.242	41.839s
	MTF	4000	2.083	42.201s
	IF + RLE	4000	3.58	38.22s
	IF	4000	5.127	38.507s
	DC + RLE	4000	9.336	36.230s
	DC	4000	6.815	35.531s
	RLE-2 + IFC	4000	2.474	39.757s
	RLE-0 + IFC	4000	2.725	36.514s
	IFC	4000	2.362	36.907s

Rysunek 5: Wyniki testu dla zbioru large

large				
plik	**algorytm**	**blok [kB]**	**śr. bitowa**	**kompr. [s]**
world192.txt	MTF	40	2.758	**23.256s**
	MTF	400	2.188	26.302s
	MTF	3000	**1.957**	29.650s
	MTF	4000	**1.957**	28.948s
	MTF	5000	**1.957**	29.858s
	MTF	8000	**1.957**	29.283s
E.coli	MTF	40	2.218	**19.382s**
	MTF	400	2.214	40.156s
	MTF	3000	2.21	57.802s
	MTF	4000	2.209	65.588s
	MTF	5000	**2.208**	64.417s
	MTF	8000	**2.208**	66.350s
bible.txt	MTF	40	2.476	**25.154s**
	MTF	400	2.19	27.529s
	MTF	3000	2.09	29.576s
	MTF	4000	**2.084**	29.504s
	MTF	5000	**2.084**	29.740s
	MTF	8000	**2.084**	29.747s
Średnio	MTF	40	2.484	**22.597s**
	MTF	400	2.197	31.329s
	MTF	3000	2.085	39.9s
	MTF	4000	**2.083**	41.346s
	MTF	5000	**2.083**	41.338s
	MTF	8000	**2.083**	41.793s

Rysunek 6: Wyniki testu parametru rozmiaru bloku BWT dla zbioru large

canterbry				
plik	**algorytm**	**blok [kB]**	**śr. bitowa**	**kompr. [s]**
asyoulik.txt	Bzip2	4000	**2.528**	0.77s
	MTF + RLE	4000	3.346	0.600s
	MTF	4000	2.902	0.582s
ptt5	Bzip2	4000	**0.775**	0.53s
	MTF + RLE	4000	1.047	636.420s
	MTF	4000	1.597	638.811s
xargs.1	Bzip2	4000	**3.334**	0.6s
	MTF + RLE	4000	5.166	0.37s
	MTF	4000	4.421	0.34s
kennedy.xls	Bzip2	4000	1.012	0.253s
	MTF + RLE	4000	**0.888**	2.599s
	MTF	4000	2.265	2.637s
sum	Bzip2	4000	**2.7**	0.17s
	MTF + RLE	4000	3.66	1.676s
	MTF	4000	3.294	1.772s
grammar.lsp	Bzip2	4000	**2.758**	0.6s
	MTF + RLE	4000	4.54	0.52s
	MTF	4000	4.016	0.53s
cp.html	Bzip2	4000	**2.479**	0.14s
	MTF + RLE	4000	3.365	0.294s
	MTF	4000	3.014	0.293s
lcet10.txt	Bzip2	4000	**2.019**	0.129s
	MTF + RLE	4000	2.655	2.800s
	MTF	4000	2.422	2.842s
plrabn12.txt	Bzip2	4000	**2.416**	0.186s
	MTF + RLE	4000	3.196	2.47s
	MTF	4000	2.824	2.205s
fields.c	Bzip2	4000	**2.18**	0.11s
	MTF + RLE	4000	3.182	0.196s
	MTF	4000	2.831	0.167s
alice29.txt	Bzip2	4000	**2.272**	0.55s
	MTF + RLE	4000	2.974	0.871s
	MTF	4000	2.62	0.786s
Średnio	Bzip2	4000	**2.224**	0.73s
	MTF + RLE	4000	3.092	58.872s
	MTF	4000	2.927	59.107s

Rysunek 7: Wyniki testu dla zbioru canterbry

calgaryS

plik	algorytm	blok [kB]	śr. bitowa	kompr. [s]	dekom. [s]
Średnio	Bzip2	4000	**2.59**	**0.88s**	**0.32s**
	MTF	4000	3.058	2.236s	0.427s
	MTF + RLE	4000	3.458	2.246s	0.454s
	RLE-2 + IFC	4000	3.767	2.210s	3.940s
	RLE-0 + IFC	4000	3.682	2.127s	0.745s
	IFC	4000	3.879	2.193s	0.638s
	DC + RLE	4000	10.848	2.149s	1.423s
	DC	4000	8.014	2.125s	0.823s
	IF + RLE	4000	4.741	2.75s	0.818s
	IF	4000	5.788	2.70s	0.511s

Rysunek 8: Wyniki testu dla zbioru calgaryS

(*a.txt*) oraz znaki w losowej kolejności (*random.txt*). Czasy te były lepsze także od czasów referencyjnego algorytmu Bzip2. Jednakże kompresja pozostałych dwóch plików zajęła kilka minut, a średnie bitowe były gorsze.

4 Napotkane trudności

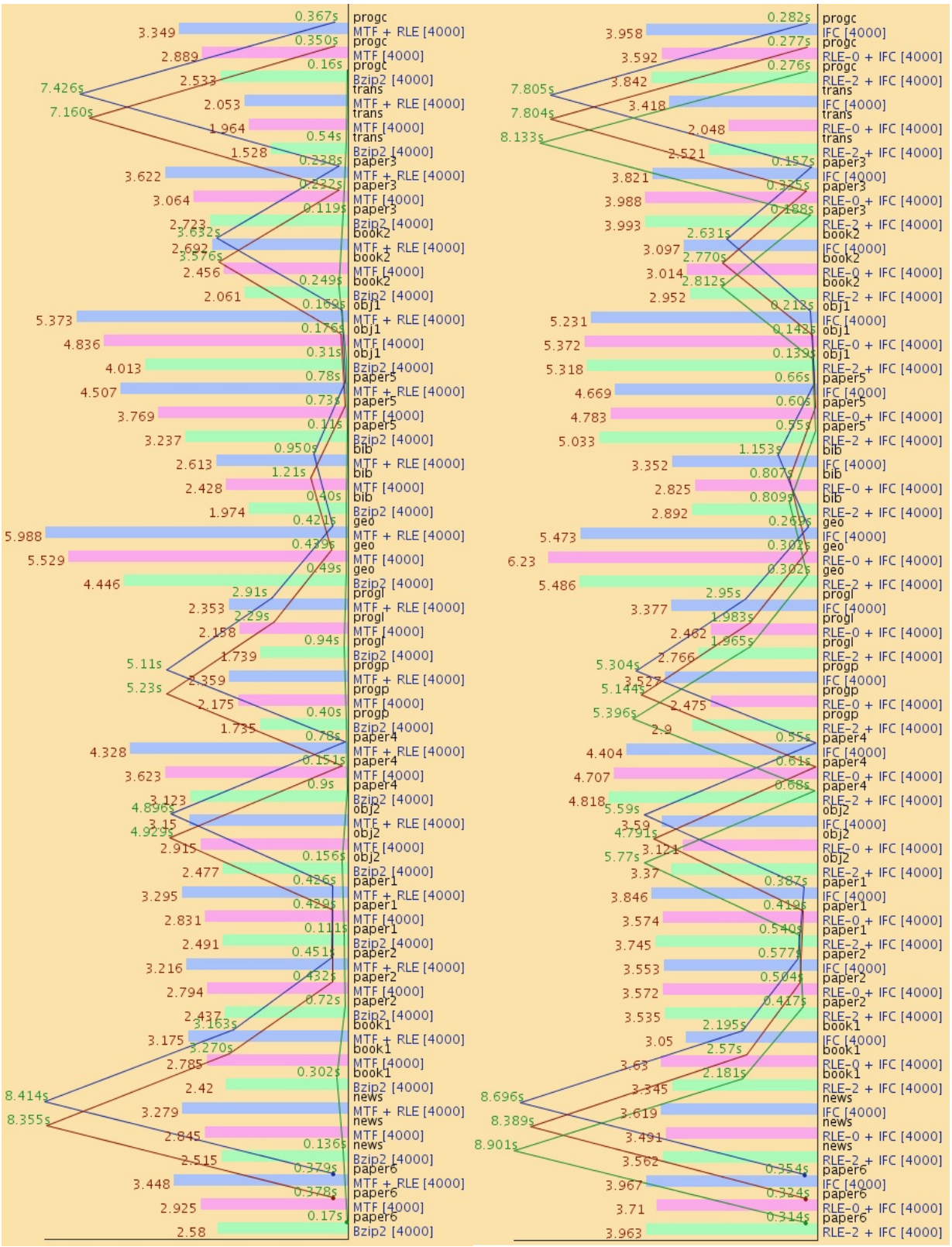
W czasie tworzenia przedstawionej w tym raporcie prezentacji napotkane zostały pewne trudności. Przede wszystkim najpoważniejszą było podejście do takiego implementowania algorytmów, by wynikowe, skompresowane dane zajmowały jak najmniej miejsca. Podstawowym problemem w przypadku takich algorytmów jak Distance Coding, Inversion Frequencies czy Huffman Coding był problem zapisu dodatkowych informacji niezbędnych przy dekompresji. Każdy z wymienionych algorytmów do poprawnego działania potrzebuje dodatkowych danych np. w algorytmie IF niezbędna jest lista zawierająca liczbę wystąpień każdego znaku. Takie dodatkowe dane stanowią szczególny problem w wypadku małych plików, gdyż ich długość, która musi być dodana do pliku wynikowego może nawet kilkukrotnie przekraczać długość skompresowanych danych.

5 Podsumowanie

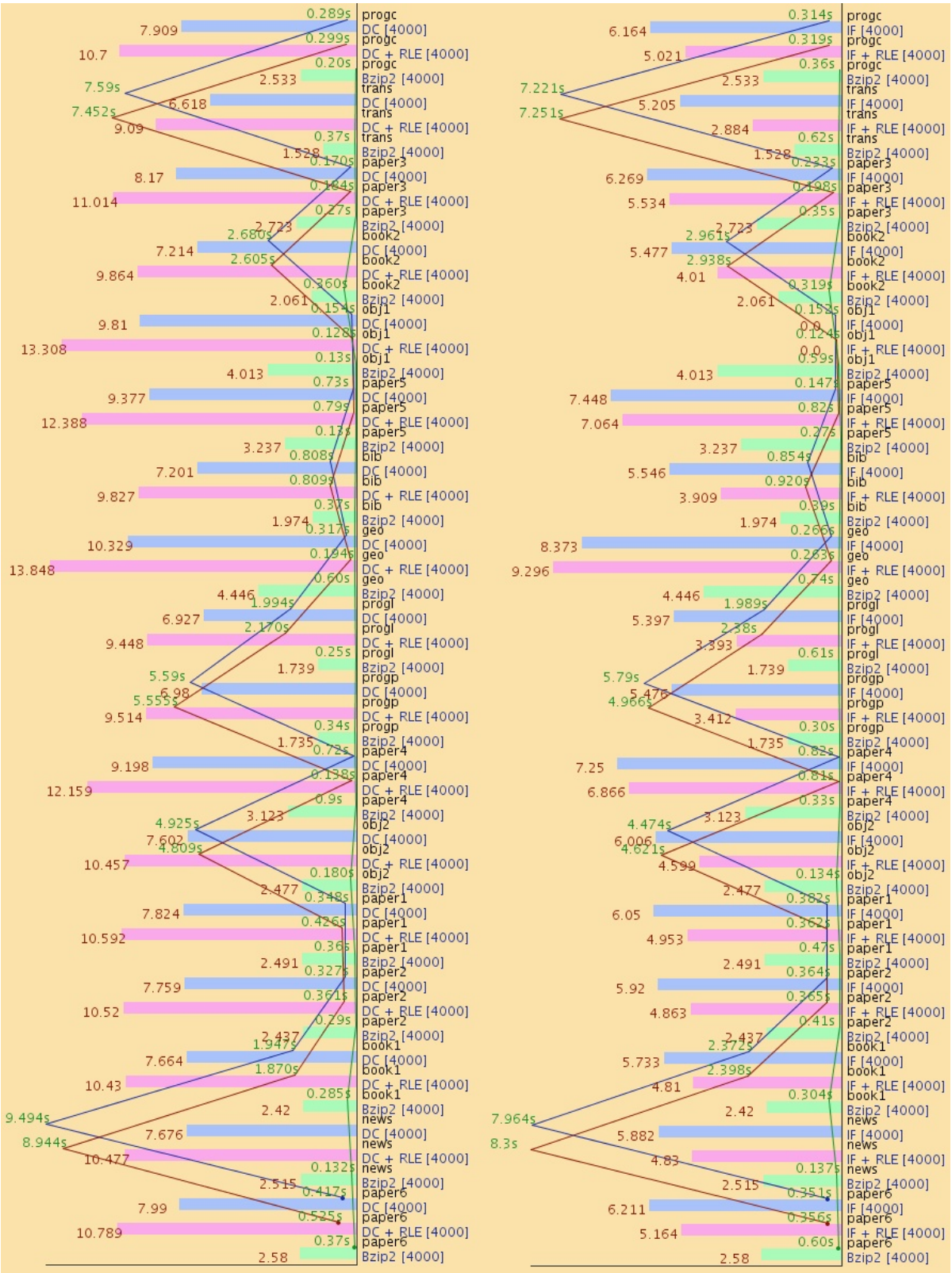
W ramach projektu stworzona została implementacja algorytmów kompresji BWT. Zaimplementowane zostały oprócz samej transformaty różnego rodzaju algorytmy drugiego kroku. Algorytmy te są dość zróżnicowane pod względem czasu działania, uzyskiwanych wyników kompresji, prostoty implementacji.

Stworzona implementacja okazała się być (WYDAJNA?/ NIE?) co pokazują wyniki przeprowadzonych testów.

Co ciekawego udało nam się zrobić? - uzyskać lepszą średnią bitową niż Bzip2 dla jednego pliku - uzyskać lepsze czasy kompresji dla kilku plików - sprawdzić nowe algorytmy fazy drugiej (IFC)



Rysunek 9: Wyniki testu dla zbioru calgaryS, algorytmy MTF i IFC



Rysunek 10: Wyniki testu dla zbioru calgaryS, algorytmy DC i IF

artificl

plik	algorytm	blok [kB]	śr. bitowa	kompr. [s]
alphabet.txt	Bzip2	4000	**0.01**	**0.172s**
	MTF + RLE	4000	0.038	204.406s
	MTF	4000	1.019	204.210s
aaa.txt	Bzip2	4000	**0.0030**	**0.428s**
	MTF + RLE	4000	0.02	699.632s
	MTF	4000	1.005	699.482s
a.txt	Bzip2	4000	**296.0**	0.427s
	MTF + RLE	4000	608.0	0.15s
	MTF	4000	384.0	**0.11s**
random.txt	Bzip2	4000	**6.054**	0.84s
	MTF + RLE	4000	7.918	0.328s
	MTF	4000	6.07	**0.311s**
Średnio	Bzip2	4000	**75.516**	**0.277s**
	MTF + RLE	4000	153.994	226.95s
	MTF	4000	98.023	226.3s

Rysunek 11: Wyniki testu dla zbioru artificl

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.