Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych

Projekt z KODA

Kodek danych z BWT

Autorzy:
Darek Pryczek
Mariusz Gwiazda
Prowadzący:
dr hab. Artur Przelaskowski

Warszawa 2005

1. Cel projektu

Celem projektu jest implementacja kodeka danych którego mechanizm działania opiera się na transformacie Burrowsa-Wheelera (Burrows-Wheeler Transform).

2. Część teoretyczna

2.1 Kompresja danych

Kompresja danych jest to działanie mające na celu zmianę sposobu zapisu informacji w taki sposób by zmniejszyć redundancję danych (objętość zbioru). przy jednoczesnym zachowaniu przenoszonych informacji. Można powiedzieć że kompresja danych polega na zapisaniu tej samej informacji za pomocą mniejszej liczby bitów.

Metody kompresji danych możemy podzielić na:

- metody bezstratne
- metody stratne.

Kompresja bezstratna polega na takim zmniejszeniu informacji (liczby bitów) aby całą informację dało się z tej postaci odtworzyć do identycznej postaci pierwotnej. Algorytmy kompresji bezstratnej dobrze kompresują zbiory o dużej redundancji(np. pliki tekstowe). natomiast dane o charakterze losowym(np. strumienie liczb losowych) są praktycznie niemożliwe do skompresowania.

W kompresji stratnej nie można odzyskać identycznej postaci informacji pierwotnej. jednak główne właściwości kompresowanego zbioru zostaną odtworzone. Kompresja stratna jest możliwa ze względu na niedoskonałości ludzkich zmysłów (pewne dane mają dla nas większą wartość niż inne). Aby wyodrębnić dane które mają dla nas największe znaczenie stosuje się modele psychowizualne (kompresja obrazów) i psychoakustyczne (kompresja dźwięków). nie istnieją algorytmy kompresji stratnej nadające się do wszystkich zastosowań ponieważ dla różnych typów danych potrzeba zachować różne właściwości.

2.2 Implementacja kodeka

W naszym projekcie wykorzystaliśmy następujące techniki kodowania danych: BWT(Burrows-Wheeler Transform). MTF(Move To Front). sortowanie indeksów(Sadakane Suffix Sort). RLE(Run-Length Encoding). kodowanie arytmetyczne(Arithmetic Coding).

BWT(Burrows-Wheeler Transform): jest jedną z najefektywniejszych metod dekompozycji strumienia danych do postaci pośredniej bardziej podatnej na kompresję za pomocą koderów arytmetycznego, Huffmana itp. Sposób działania transformacji BWT polega na pobraniu bloku danych wejściowych. odpowiednim poprzestawianiu jego elementów przy użyciu algorytmu sortowania. W rezultacie działania tego algorytmu otrzymujemy na wyjściu blok danych zawierający te same elementy co blok wejściowy ale występujące w innej kolejności (występują długie sekwencje tych samych znaków). Transformacja jest procesem odwracalnym. oryginalna kolejność występowania elementów w bloku może zostać w pełni przywrócona. Działanie algorytmu BWT można podzielić na 3 etapy(zaprezentujemy je na przykładzie słowa „banana") :

1) Tworzymy listę sufixów ciągu wejściowego, gdzie n-ty sufix to sekwencja bez n pierwszych symboli. Na koniec ciągu dopisujemy też unikalny symbol terminalny (oznaczamy go tutaj przez "$").

   Sufix 0: "banana$"

Sufix 1: “anana$”
Sufix 2: “nana$”
Sufix 3: “ana$”
Sufix 4: “na$”
Sufix 5: “a$”
Sufix 6: “$”

2) W drugim kroku sortujemy leksykograficznie listę sufixów. Symbol terminalny traktujemy jako alfabetycznie mniejszy od każdego innego symbolu (ale jest to wybór arbitralny).

Sufix 6: “$”
Sufix 5: “a$”
Sufix 3: “ana$”
Sufix 1: “anana$”
Sufix 0: “banana$”
Sufix 4: “na$”
Sufix 2: “nana$”

Posortowana lista sufixów: 5, 3, 1, 0, 4, 2
Sufix “$” pomijamy.

3) Wyjściowa postać danych po transformacji to sekwencja ułożona z pierwszych literek poprzedzających sufixy z posortowanej listy sufixów. W przypadku sufixu 0 (nie ma literki poprzedzającej) zapisujemy ostatnią literkę ciągu. Dodatkowo trzeba zapisać liczbę oznaczającą pozycję w posortowanej liście sufixów, na której znalazł się sufix 0. W naszym przypadku będzie to:

“nnbaaa”, 3

W algorytmie BWT najbardziej czasochłonną fazą jest sortowanie sufixów. Wykorzystujemy do tego celu algorytm Sadakane. Jego złożoność obliczeniowa wynosi O(n*log^2(n)), natomiast złożoność pamięciowa do 8n. Ponieważ nie było to zasadniczym tematem naszej projektu nie będziemy dokładnie opisywać tego algorytmu. Skrótowo jest to modyfikacja algorytmu KMP, którą zaimplementowaliśmy dosyć wiernie na podstawie oryginalnej pracy Sadakane. Naszym autorskim wykładem jest faza wstępnego sortowania po k-pierwszych literkach zrealizowana w sposób wydajny metodą radix-sortową, bez żadnych nadmiarowych przebiegach po sekwencji (praca Sadakane tylko wspominała o tej fazie, ale nie przedstawiała szczegółów implementacyjnych).

MTF(Move To Front): algorytm MTF nie kompresuje danych. ale pomaga ograniczyć redundancję tych danych. Bardzo często jest używany bezpośrednio po transformacji BWT. Transformacja MTF nie zwraca symbolu(bitu) na wyjściu, a zamiast niego zwraca liczbę wskazującą na pozycje symbolu w tablicy symboli. Długość zwróconego kodu jest równa długości ciągu symboli. Zarówno koder jak i dekoder MTF powinny zaczynać kodowanie z taką samą tablicą symboli(wszystkie symbole są na tych samych pozycjach). Dla każdego przetwarzanego symbolu koder zwraca jego pozycję w tablicy i przenosi go na początek tej tablicy. Wszystkie symbole występujące do pozycji na której znajduje się kodowany symbol są przesuwane o jedną pozycję dalej. Dzięki temu częściej występujące symbole w ciągu mają przyporządkowane mniejsze wartości.

Przykład transformacji MTF dla ciągu wejściowego: „abaacabad”. Tablica symboli składa się z elementów {a.b.c.d}. Dla każdego podawanego na wejście znaku z ciągu tablica transformacji przybiera postać:

Symbol	index	Lista symboli
a	0	a b c d
b	1	b a c d
a	1	a b c d
a	0	a b c d
c	2	c a b d
a	1	a c b d
b	2	b a c d
a	1	a b c d
d	3	d a b c

Transformacja MTF jest procesem odwracalnym i bezstratnym.

RLE(Run-Length Encoding): jest to algorytm kodowania długości sekwencji. Główna idea tego kodowania polega na zastąpieniu ciągów o takich samych znakach przez znak oraz liczbę jego powtórzeń w ciągu. Istnieje wiele różnych schematów kodowania RLE które zazwyczaj są przystosowane do typu danych które mają być skompresowane.

Kompresja RLE jest skuteczna tylko w przypadku kompresji danych o dużej redundancji takich jak pliki tekstowe(jeżeli zawierają duże ciągi takich samych znaków np. spacji). obrazów o dużych jednostajnych obszarach (np. obrazy czarno białe). Do zalet kodowania RLE należą: łatwa implementacja. mała złożoność obliczeniowa.

W naszym przypadku posłużyliśmy się schematem RLE-0 z uwagi na to że procentowo liczba zer po transformacji MTF może osiągnąć nawet 90% (średnio około 60%). Działanie tej metody polega na tym że sekwencje znaków „0” zamieniamy na ich liczbę wystąpień kodowaną za pomocą krótszej sekwencji złożonej z dwóch specjalnych znaków $\{0_a.0_b\}$. W tym celu należy alfabet danych po transformacie MTF rozszerzyć do postaci $A_{rle0} = (A_{mtf} \setminus \{0\}) \cup \{0_a.0_b\}$. Poniższa tabela pokazuje przykładowe kodowanie RLE-0 dla od 1 do 9 wystąpień „0”.

Liczba wystąpień	Kod RLE-0
1	0_a
2	0_b
3	0_a0_a
4	0_a0_b
5	0_b0_a
6	0_b0_b
7	$0_a0_a0_a$
8	$0_a0_a0_b$
9	$0_a0_b0_a$

Rozważaliśmy użycie też innego kodowania symboli po transformacie RLE0, np.:

- użyć jednego dodatkowego znaku specjalnego oznajmiającego wystąpienie sekwencji 0 i po nim zapisywać na stałej, bądź zmiennej (przedziały) liczbie bitów ilość następujących po sobie 0.
- Dodać do alfabetu pewną liczbę (dobraną eksperymentalnie, bądź na podstawie analizy danych) dodatkowych symboli oznaczających kolejne ilości następujących po sobie 0 (np. symbol 256 mógłby oznaczać dwa 0, 257 3 zera, itd.).

- Modyfikacja powyższego polegająca na podzieleniu długości sekwencji 0 na przedziały o różnych długościach i zapisywać na wyjście numer przedziału (za pomocą dodanych do alfabetu symboli specjalnych) oraz po nim numer elementu z danego przedziału.

Zdecydowaliśmy się jednak na wersję z symbolami 0a i 0b prezentowaną powyżej ze względu na jej prostotę oraz na to, iż tą metodą posłużył się i po raz pierwszy opisał dr S. Deorowicz, którego praca była dla nas inspiracją.

Pod względem szczegółów implementacyjnych wyglądało to następująco:
- alfabet wyjściowy transformaty RLE0 to alfabet wejściowy z dodatkowym jednym znakiem (czyli 256 + 1 = 257 znaków)
- symbol 0a kodujemy za jako 0
- symbol 0b kodujemy jako 1
- każdy z 256 symboli alfabetu wejściowego kodujemy jako symbol o kodzie zwiększonym o 1 (czyli np. 97 (literka ‘a’) jest kodowana jako znak 98)

Złożoność algorytmu jest liniowa. Z powodów wydajnościowych nasz program nie generował dodatkowej tablicy reprezentującej dane po transformacie RLE0, tylko połączyliśmy w jedno przejście transformatę MTF, RLE0 oraz kodowanie arytmetyczne.

Kodowanie odległości (distance coder, DC): jest to alternatywny rodzaj transformacji stosowanej do wyniku transformaty BWT, używany zamiast MTF czy pary MTF + RLE0. Zasadnicza jej idea polega na tym, by zastąpić sekwencję symboli sekwencją odległości, gdzie na każdy symbol zapisana jest odległość liczona w pozycjach do następnego wystąpienia tego symbolu. Dodatkowo na sekwencji wyjściowej trzeba dopisać tabelę z pierwszą pozycją każdego symbolu, co może pogarszać wyniki kompresji dla krótkich plików. Trzeba też wprowadzić specjalną odległość oznaczającą, iż dany symbol nie występuje już więcej w sekwencji (w naszej implementacji jest to 0). Algorytm ten można łatwo rozszerzyć o kilka znacząca polepszających stopień kompresji ulepszeń. Po pierwsze, można przestać zapisywać do pliku wyjściowego odległości, jeżeli do końca zostały już same odległości 0 (oznaczające brak kolejnych wystąpień danych symboli). Pozwala to przynajmniej w pewnym stopniu zrekompensować narzut związany z tabelą pierwszych wystąpień, choć to zależy od danych wejściowych i w pesymistycznych przypadkach kończącą sekwencja 0 może mieć długość 1. Po drugie, można zapisywać odległości uwzględniając tylko pozycje jeszcze nie zajęte. Ponieważ podczas kodowania danego symbolu znamy kolejne wystąpienia wszystkich pozostałych symboli, możemy ich pozycji nie brać pod uwagę i zapisywać odległość liczoną tylko ilością pustych pozycji. To znacząco przesuwa dolne partie (odpowiadające mniejszym odległościom) histogramu odległości w kierunku mniejszych liczb i daje zauważalną poprawę efektywności. Trzecią optymalizacją jest rozszerzenie kodera DC o algorytm podobny do RLE, który daje się tu wprowadzić w sposób bardzo naturalny. Wystarczy zauważyć, że w przypadku, gdy symbol w sekwencji wejściowej jest powtórzeniem poprzedniego symbolu nie musimy w ogóle kodować odległości. Dekoder będzie w stanie to rozpoznać po tym, iż nie będzie miał do danej pozycji przypisanego żadnego symbolu. Cała sekwencja takich samych symboli zostanie więc zastąpiona prze jedną tylko odległość. Może to być zarówno odległość od pierwszego symbolu sekwencji jak i od ostatniego, ale eksperymenty pokazały, iż używanie tej drugiej opcji jest zdecydowanie korzystniejsze (ta odległość będzie mniejsza).

Koder DC ze względu na konieczność znajomości pozycji kolejnego wystąpienia symbolu musi operować na blokach danych, co w naturalny sposób łączy go z BWT. Złożoność obliczeniowa naszego kodera jest liniowo zależna od długości bloku i także

liniowo od rozmiarów alfabetu (choć da się to zaimplementować w złożoności logarytmicznej względem rozmiaru alfabetu).
Poniższy przykład ilustruje działanie kodera DC na krótkiej sekwencji symboli z alfabetu {a, b, c}:

Pozycja w sekwencji wejściowej				**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
wejście				a	a	b	a	c	b	b	b	a	b	a	a	c
Odległości liczone ilością pozycji	1	3	5	1	2	3	5	8	1	1	2	2	0	1	0	0
Ucięte końcowe 0 oraz odległości liczone ilością pustych pozycji	1	2	3	1	1	1	3	6	1	1	1	1	0	1		
Zastosowanie RLE	1	2	3	2	-	1	3	6	1	-	-	1	0	1		

Ostatecznie zakodowane odległości to: 1 2 3 2 1 3 6 1 1 0 1
Do tego musimy jeszcze znać rozmiar wejściowej sekwencji i to już wystarcza do w pełni jednoznacznego zdekodowania.

Pozostaje jeszcze problem efektywnego kodowania odległości w pliku wyjściowym. W tym celu odległości dzielimy na przedziały o rosnącym rozmiarze i do pliku wynikowego zapisujemy pary {numer przedziału, numer odległości z danego przedziału}. Pierwszą wielkość z tej pary kodujemy adaptywnym koderem arytmetycznym, druga natomiast jest zapisywana do pliku bez kompresji (ale w celu uproszczenia kodu używamy do tego celu kodera arytmetycznego). Sensownym zestawem przedziałów okazał się być:

przedział	**Liczba bitów koniecznych do zapisani bez kompresji**
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8-9	1
10-11	1
12-13	1
14-15	1
16-19	2
20-23	2
24-27	2
28-31	2
32-39	3
40-47	3
48-55	3

przedział	Liczba bitów koniecznych do zapisani bez kompresji
56-63	3
...	...

Liczba przedziałów jest ustalana na podstawie rozmiaru pliku, gdyż on ogranicza maksymalna odległość.

Pod względem implementacyjnym koder i dekoder DC wyglądały następująco:

Kodowanie przeprowadzamy przeglądając całą tablicę wejścia transformaty od końca do początku. Utrzymujemy tablicę pomocniczą o rozmiarach równych ilości symboli w alfabecie. W tej tablicy przechowujemy pozycję następnego (patrząc od początku, z punktu widzenia kolejności przeglądania jest to poprzednie wystąpienie) wystąpienia symbolu, który aktualnie kodujemy. Tablica ta jest inicjalizowana wielkością równą rozmiarowi danych, co oznacza, że takiego symbolu jeszcze nie kodowaliśmy. Jeżeli tablica informuje, że takiego symbolu jeszcze nie był, to kodujemy to odległością 0. Jeżeli taki symbol już wystąpił, to liczymy wstępną odległość (licząc zajęte już pola) odejmując aktualną pozycję w sekwencji od wartości zapisanej w tablicy. Następnie odejmujemy od tego liczbę wszystkich zajętych już pól, które można zidentyfikować przeglądając pomocniczą tablicę i zliczając elementy o pozycji mniejszej niż następna pozycją aktualnego symbolu. W ten sposób otrzymujemy już gotową odległość do następnego symbolu. Po każdym zakodowanym symbolu zapisujemy do niej aktualną pozycję. W ten mechanizm wplatamy jeszcze podobne do RLE wykrywanie powtórzeń, co można łatwo analizując policzone odległości wstępne.
Następnie algorytm kodera zapisuje na podobnych zasadach tablicę odległości do pierwszych wystąpień każdego symbolu z alfabetu, a kolejności od 255 do 0 (ale ponieważ kodujemy od końca, to w sekwencji wyjściowej będą zapisane od 0 do 255).
Na zakończenie zliczamy liczbę zer znajdujących się na końcu powstałej sekwencji odległości i odejmujemy ją od rozmiarów zakodowanej sekwencji.

W finalnej wersji programu wprowadziliśmy optymalizację polegająca na zrezygnowaniu z tablicy pomocniczej na rzecz listy par <symbol, następna pozycja>. Informacyjnie ta lista była równoważna tablicy, ale dzięki temu, że utrzymywaliśmy ją w postaci posortowanej rosnąco względem odległości można było znacząco przyspieszyć algorytm (choć jego złożoność pozostaje taka sama). Znalezienie następnej pozycji teraz jest wolniejsze (zamiast zaglądania w czasie stałym do tablicy jest przeszukiwania listy od początku), ale nie trzeba przeglądać całej tablicy w celu znalezienia symboli występujących pomiędzy kodowanym symbolem a jego następnym wystąpieniem, gdyż informację tą w naturalny sposób uzyskujemy z posortowanej listy. W praktyce wyszukiwanie następnego wystąpienia i zliczanie oraz aktualizacja listy są wykonywane w jednym przebiegu. Ponieważ idea kodowania DC polega na tym, że odległości przeważnie są niewielkie to najczęściej będziemy przeglądali zaledwie kilka pierwszych elementów listy (zamiast całej 256 elementowej tablicy).

Dekoder działa analogicznie, tylko przegląda sekwencje odległości od początku do końca i nie wymaga przez to przechowywania ich w osobnej tablicy (dekodowanie można wykonywać online, tj. symbol po symbolu w jednym przebiegu z dekoderem arytmetycznym).

Kodowanie arytmetyczne (arytmetic coding): idea kodowania arytmetycznego polega na tym aby przyporządkować zbiorowi danych. poddawanemu kompresji. liczbę z zakresu [0.1) (musi być to liczba jednoznacznie dekodowalna). Algorytm poszukiwania tej liczby polega na podzieleniu przedziału początkowego [0.1) na podprzedziały ze względu na alfabet danych oraz prawdopodobieństwo wystąpienia danego symbolu. Ten podział nosi nazwę linii prawdodobieństw. Następnie dla każdego pojawiającego się symbolu na wejściu wybieramy odpowiedni przedział charakterystyczny dla tego symbolu. Następne symbole z kodowanej sekwencji powodują wchodzenie coraz głębiej w ten przedział. Prowadzi to do ostatecznej postaci przedziału kodu, który jest identyfikatorem sekwencji danych. Dowolna liczba z tego przedziału jest szukaną arytmetyczną reprezentacją zbioru danych.

Metoda ta jest bardzo skuteczna ze względu na to że długość otrzymanego kodu jest niewiele większa od entropii danego modelu. Gdyby nie ograniczenia patentowe metoda ta znalazła by szerokie zastosowanie w dzisiejszych koderach.

W naszym programie wypróbowaliśmy dwie wersje kodowania arytmetycznego: statyczną i adaptywną. Metoda statyczna wylicza stałe dla całego bloku prawdopodobieństwa wystąpienia symboli, przyporządkowuje na ich podstawie przedziały a następnie koduje już całą sekwencję nie zmieniając raz ustalonych przedziałów. Wymaga to dodatkowego przejścia po danych, ale później kodowanie jest wykonywane znacznie szybciej. W metodzie statycznej trzeba też przekazać informację o rozkładzie dekoderowi, co powiększa plik wyjściowy i pogarsza współczynnik kompresji (zwłaszcza na małych plikach). Metoda adaptywna natomiast zmienia prawdopodobieństwa (i przez to przedziały) po zakodowaniu każdego kolejnego symbolu. Nasz koder startuje z równomiernym rozkładu prawdopodobieństw (każdy symbol wystąpił tylko raz) i aktualizuje go podczas kodowania. Po zakodowaniu pewnej ilości symboli wszystkie ilości wystąpień symboli są dzielone przez 2, co daję koderowi możliwość adaptacji do zmieniającego się kontekstu. Eksperymenty pokazały, że rozsądną ilością symboli wyzwalającą dzielenie jest 8000. Koder adaptywny jest minimalnie wolniejszy (ze względu na ciągłą konieczność aktualizowania rozkładu), ale daje znacząco lepsze rezultaty.

Przeprowadziliśmy też eksperymenty z koderem binarnym już nie naszego autorstwa. Użyliśmy programu „bac” dostępnego na stronie przedmiotu. Testowaliśmy go wyłącznie w połączeniu z koderem DC. W tym celu specjalna wersja kodera DC generowała plik, w którym kodowała każdą odległość bez użycia kodera arytmetycznego wg schematu:

Odległość (przedział)	**Sekwencja bitowa**
0	0 0
1	0 1
2	1 0
3-10 (8 możliwych)	1 1 0 d2 d1 d0
11-26 (16 możliwych)	1 1 1 0 d3 d2 d1 d0
27-58 (32 możliwe)	1 1 1 1 0 d4 d3 d2 d1 d0
59-122 (64 możliwe)	1 1 1 1 1 0 d5 d4 d3 d2 d1 d0
123-250 (128 możliwych)	1 1 1 1 1 1 0 d6 d5 d4 d3 d2 d1 d0
...	...

Bity dn (d0, d1, ...) kodują jedną z odległość z przedziału.

Eksperymenty pokazały, że na testowanych danych optymalny rząd kodera binarnego to 10.

3. Program

Program został napisany w c++ i skompilowany za pomocą dwóch kompilatorów: gcc 4.9.9.2 oraz Visual Studio 2003. Wersja skompilowana w gcc jest szybsza dlatego też wszystkie testy przeprowadzaliśmy za pomocą tej wersji programu.

Program ma zaimplementowane następujące warianty kompresji:

- BWT + MTF + RLE-0 + adaptative arithmetic
- BWT + MTF + RLE-0 + static arithmetic
- BWT + MTF + adaptative arithmetic
- BWT + MTF + static arithmetic
- BWT + DC + adaptive arithmetic

Program pozwala także na definiowanie wielkości bloku BWT.

3.1 Interfejs programu

Powstały program jest aplikacją konsolową – wywołuje się go z poziomu wiersza poleceń podając odpowiednie parametrami. Lista parametrów znajduje się w tabeli 3.3.1

Parametr	Opis
-c nazwa pliku	kompresuj podany plik (domyślna akcja)
-e nazwa pliku	dekompresuj podane archiwum
-o nazwa pliku	nazwa pliku wyjściowego – jeżeli jest to kompresja to ten parametr określa nazwę archiwum. w przypadku dekompresji określa nazwę pliku wypakowanego
-b rozmiar	rozmiar bloku BWT. wymusza stały rozmiar bloku wyłączając adaptywne algorytmy
-m. --mtf0	użyj MTF0 (domyślnie)
-d, --dc	użyj kodera DC
-r. --rle0	użyj RLE-0 (domyślnie)
--norle0	wyłącz RLE-0
-a	użyj adaptywnego kodera arytmetycznego
-s	użyj statycznego kodera arytmetycznego
-p	wygeneruj wyjście dla programu **bac** (tylko przy opcji -d)
-v. --verbose	wypisywanie dodatkowych informacji na ekranie (także informacji debugujących)
--test nazwa pliku	pakowany plik jest od razu rozpakowywany oraz porównywane są zawartości plików oryginalnego i otrzymanego po rozpakowaniu
--bwt_output nazwa pliku	zapisuje wyjście BWT do pliku
--mtf_output nazwa pliku	zapisuje wyjście MTF do pliku
--rle0_output nazwa pliku	zapisuje wyjście RLE-0 do pliku
--dc_output	zapisuje wyjście DC do pliku
--input_histogram nazwa pliku	Zapisuje histogram danych wejściowych do pliku
--mtf_histogram	Zapisuje histogram danych po MTF do pliku

nazwa pliku	
--rle0_histogram nazwa pliku	Zapisuje histogram danych po RLE-0 do pliku
--dc_histogram	Zapisuje histogram odległości DC do pliku

Tabela 3.3.1 Parametry kodeka

Przykładowe wywołania programu:

Pakowanie:

koda018gcc.exe -c book2 -o book2.koda

koda018gcc book1 -o book1.kodaA10p -b 16384 -d -a -v --bwt_output bwtout.txt

Rozpakowywanie:

koda018gcc.exe -e book2.koda -o book2

koda018gcc -e book1.kodaA10p -o book1.kodaA10u –v

4. Testy

Testy podzieliliśmy na trzy kategorie. najpierw testujemy różne warianty naszego kompresora a następnie najlepszy wariant porównujemy z najbardziej popularnymi uniwersalnymi kompresorami (WinRK 2.1 built 9. WinRAR 3.20. WinZip 9.0 Sp1. bzip2 1.0.2). Trzeci test ma za zadanie pokazać wpływ rozmiaru bloku BWT na skuteczność kompresji.

4.1 Porównanie wariantów kompresji

a) Testy przeprowadzone na „LargeCalgaryCorpus”

Metoda	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
trans (93 695)				
MTF+RLE0+AA	17 955	1.53	0.078	0.031
MTF+RLE0+SA	18 244	1.56	**0.062**	**0.016**
MTF+AA	19 445	1.66	0.109	0.078
MTF+SA	19 628	1.68	**0.062**	0.047
DC + AA	17 718	**1.51**	0.094	**0.016**
progp (49 379)				
MTF+RLE0+AA	10 911	1.77	**0.031**	0.031
MTF+RLE0+SA	11 217	1.82	**0.031**	**0.015**
MTF+AA	11 763	1.91	0.046	0.047
MTF+SA	12 057	1.95	**0.031**	**0.015**
DC + AA	10753	**1.74**	0.046	0.016
progl (71 646)				
MTF+RLE0+AA	15 965	**1.78**	0.062	0.031
MTF+RLE0+SA	16 265	1.82	**0.046**	0.016
MTF+AA	17 285	1.93	0.063	0.047
MTF+SA	17 617	1.97	0.06	**0.05**
DC + AA	15 668	**1.75**	0.062	0.016
progc (39 611)				
MTF+RLE0+AA	12 776	**2.58**	0.031	**0.015**
MTF+RLE0+SA	13 089	2.64	0.016	**0.015**

MTF+AA	13 552	2.74	0.031	0.016
MTF+SA	13 875	2.8	**0.015**	0.016
DC + AA	12 609	**2.55**	0.046	0.016
pic (513 216)				
MTF+RLE0+AA	51 471	0.80	0.438	0.14
MTF+RLE0+SA	53 645	0.84	**0.421**	0.11
MTF+AA	58 579	0.91	0.671	0.391
MTF+SA	72 560	1.13	0.468	0.219
DC + AA	50 242	**0.78**	0.5	**0.109**
paper2 (82 199)				
MTF+RLE0+AA	25 760	2.51	0.078	0.047
MTF+RLE0+SA	26 213	2.55	**0.046**	**0.032**
MTF+AA	27 572	2.68	0.078	0.062
MTF+SA	28 324	2.76	**0.046**	0.047
DC + AA	25 104	**2.44**	0.093	**0.032**
paper1 (53 161)				
MTF+RLE0+AA	16 958	2.55	0.031	0.031
MTF+RLE0+SA	17 285	2.60	**0.015**	0.031
MTF+AA	17 984	2.71	0.046	0.047
MTF+SA	18 404	2.77	**0.015**	0.031
DC + AA	16 506	**2.48**	0.046	**0.016**
obj2 (246 814)				
MTF+RLE0+AA	77 093	**2.50**	0.234	0.125
MTF+RLE0+SA	77 880	2.52	**0.187**	**0.094**
MTF+AA	84 027	2.72	0.296	0.204
MTF+SA	85 529	2.77	0.203	0.125
DC+AA	78 100	2.53	0.312	0.094
obj1 (21 504)				
MTF+RLE0+AA	10 606	**3.95**	**0.015**	0.016
MTF+RLE0+SA	11 002	4.09	**0.015**	0.016
MTF+AA	11 575	4.31	0.031	0.015
MTF+SA	11 954	4.45	**0.015**	**0.001**
DC+AA	10 897	4.05	0.031	0.015
news (377 109)				
MTF+RLE0+AA	120 798	2.56	0.421	0.25
MTF+RLE0+SA	122 292	2.59	**0.343**	**0.172**
MTF+AA	129 846	2.75	0.5	0.328
MTF+SA	132 582	2.81	0.359	0.219
DC+AA	118 921	**2.52**	0.531	**0.172**
geo (102 400)				
MTF+RLE0+AA	61 060	4.77	0.093	0.078
MTF+RLE0+SA	62 570	4.89	**0.078**	**0.062**
MTF+AA	66 150	5.17	0.109	0.094
MTF+SA	69 038	5.39	**0.078**	**0.062**
DC+AA	60 087	**4.69**	0.156	0.078
book2 (610 856)				
MTF+RLE0+AA	161 184	2.11	0.75	0.406
MTF+RLE0+SA	164 852	2.16	**0.609**	**0.281**

MTF+AA	175 676	2.30	0.875	0.546
MTF+SA	183 507	2.40	0.656	0.328
DC+AA	156 189	**2.05**	0.875	**0.281**
book1 (768 771)				
MTF+RLE0+AA	238 695	2.48	1.03	0.609
MTF+RLE0+SA	244 808	2.55	**0.828**	0.422
MTF+AA	253 823	2.64	1.16	0.734
MTF+SA	265 691	2.76	0.875	0.453
DC+AA	229 053	**2.38**	1.23	**0.406**
bib (111 261)				
MTF+RLE0+AA	28 275	2.03	0.109	0.062
MTF+RLE0+SA	28 606	2.06	**0.078**	**0.031**
MTF+AA	31 987	2.30	0.109	0.094
MTF+SA	32 326	2.32	**0.078**	0.047
DC+AA	27 859	**2**	0.109	**0.031**
allcalgarycorpus (3 141 622)				
MTF+RLE0+AA	879 002	2.24	4.19	2.02
MTF+RLE0+SA	914 438	2.33	**3.5**	**1.39**
MTF+AA	949 328	2.42	4.84	2.81
MTF+SA	1 032 707	2.63	3.69	1.77
DC+AA	861 148	**2.19**	4.73	1.45

Wnioski:

Testowana wersja programu to 0.34 udało nam się przyspieszyć program w porównaniu do wersji poprzedniej:

allcalgarycorpus (3 141 622)					
Wersja 0.24	MTF+RLE 0+AA	879 017	**2.24**	5.58	2.14
Wersja 0.33	MTF+RLE 0+AA	879 002	**2.24**	4.19	2.02

Zdecydowanie najlepszą kombinacją (dającą najniższe średnie bitowe) w testach przeprowadzonych na korpusie „Calgary" jest DC + adaptative aritmetic, dlatego też porównamy ten wariant z innymi popularnymi kodekami.

Po wynikach testów wyraźnie widać, że stosowanie kodera statycznego pogarsza stopień kompresji. Koder adaptywny i to z pewną możliwością dostosowywania się do aktualnego kontekstu ma przewagę, co przekłada się na niewielki tylko spadek prędkości (będzie o wyraźnie widać w testach na korpusie silesia).

Kolejną obserwacją jest zdecydowana opłacalność stosowania transformacji RLE0. Znacząco podnosi ona stopień kompresji przy kodowaniu MTF, co jest zrozumiałe, gdyż zmniejsza ona rozmiar danych jeszcze przed kodowaniem arytmetycznym, i to przeważnie o 30-40%.

Najlepsze wyniki osiągnął jednak koder DC. Wyższość kodera DC nad kodowaniem MTF + RLE0 bierze się prawdopodobnie stąd, iż:

- koder DC w naturalny sposób stosuje mechanizm podobny do RLE i w przeciwieństwie do niego na każdą sekwencję powtarzających się symboli wypuszcza na wyjście tylko jeden symbol (zamiast serii symboli 0a, 0b). Ponadto jego sposób kodowania

odległości liczonych tylko niezajętymi jeszcze polami daje w rezultacie większe przesunięcie histogramu kodowanych wartości w stronę małych liczb, niż daje MTF. Dalej, transformata MTF każdą sekwencję powtarzających się symboli skraca faktycznie o 1, gdyż pierwszy symbol nie będzie zakodowany za pomocą 0. Te 3 własności razem wzięte przeważają nawet słabsze strony kodera DC, jak kodowanie dużych liczb(alfabet odległości jest znacznie większy od alfabetu symboli) czy nadmiar powodowany koniecznością dodatkowego zakodowania pierwszej pozycji każdego symbolu, w rezultacie koder DC wychodzi z tych testów jako zwycięzca.

Ponieważ pliki testowe są tak niewielkie i czas kompresji zazwyczaj nie przekracza 1 s dlatego też zrezygnowaliśmy z pomiaru czasu w tym teście(wyjątek stanowi plik allcalgarycorpus który jest plikiem powstałym z połączenia wszystkich plików „LargeCalgaryCorpus").

Wszystkie programy miały (jeśli była taka opcja) włączoną opcję najlepszej kompresji.

	rozmiar po kompresji	średnia bitowa
trans (93 695)		
DC+AA	17 718	1.51
WinRAR	14 800	1.26
Bzip2	17 899	1.53
WinRK	13 226	**1.13**
WinZip	19 265	1.64
progp (49 379)		
DC+AA	10753	1.74
WinRAR	9 327	1.51
Bzip2	10 710	1.74
WinRK	8 790	**1.42**
WinZip	11 422	1.85
progl (71 646)		
DC + AA	15 668	1.75
WinRAR	13 215	1.48
Bzip2	15 579	1.74
WinRK	11 616	**1.30**
WinZip	16 580	1.85
progc (39 611)		
DC+AA	12 609	2.55
WinRAR	11 053	2.23
Bzip2	12 554	2.54
WinRK	9 957	**2.01**
WinZip	13 435	2.71
pic (513 216)		
DC + AA	50 242	0.78
WinRAR	46 940	0.73
Bzip2	49 759	0.78
WinRK	29 658	**0.46**
WinZip	57 230	0.89
paper2 (82 199)		
DC + AA	25 104	**2.44**

WinRAR	22 490	2.19
Bzip2	25 041	2.44
WinRK	19 177	**1.87**
WinZip	30 209	2.94
paper1 (53 161)		
DC + AA	16 506	2.48
WinRAR	14 665	2.21
Bzip2	16 558	2.49
WinRK	12 760	**1.92**
WinZip	18 781	2.83
obj2 (246 814)		
MTF+RLE0+AA	77 093	2.50
WinRAR	71 315	2.31
Bzip2	77 441	2.51
WinRK	55 765	**1.81**
WinZip	82 822	2.68
obj1 (21 504)		
MTF+RLE0+AA	10 606	3.95
WinRAR	9 819	3.65
Bzip2	10 787	4.01
WinRK	8 876	**3.30**
WinZip	10 404	3.87
news (377 109)		
DC+AA	118 921	2.52
WinRAR	103 981	2.21
Bzip2	118 600	2.52
WinRK	90 306	1.92
WinZip	145 456	3.09
geo (102 400)		
DC+AA	60 087	4.69
WinRAR	62 687	4.90
Bzip2	56 921	4.45
WinRK	47 233	**3.69**
WinZip	68 791	5.37
book2 (610 856)		
DC+AA	156 189	2.05
WinRAR	140 424	1.84
Bzip2	157 443	2.06
WinRK	124 636	**1.63**
WinZip	209 336	2.74
book1 (768 771)		
DC+AA	229 053	2.38
WinRAR	210730	2.19
Bzip2	232598	2.42
WinRK	187156	**1.95**
WinZip	318156	3.31
bib (111 261)		
DC+AA	27 859	2

WinRAR	24 164	1.74		
Bzip2	27 467	1.97		
WinRK	19 822	**1.43**		
WinZip	35 940	2.58		
allcalgarycorpus (3 141 622)		BR	t_k (s)	t_d (s)
DC+AA	861 148	2.19	4.73	1.45
WinRAR	764 588	1.95	3.5	2.1
Bzip2	862 695	2.20	1.8	**0.5**
WinRK	650 362	**1.66**	71	51
WinZip	1 041 027	2.65	**1.5**	1

Wnioski:

Zdecydowanym zwycięzcą jest WinRK używa on metody PPMD.

Najciekawsze porównanie to porównanie programów naszego oraz programu Bzip2 ze względu na zaimplementowanie w obu programach metody BWT, z tym że Bzip2 używa najpierw transformaty Burrowsa-Wheelera, następnie przekształca wyniki po BWT przez algorytm Move To Front, a w końcu kompresuje dane wyjściowe z MTF za pomocą algorytmu Huffmana. Informacja ta jednak jest niezbyt pewna (wzięta z internetu) ponieważ wydaje nam się że po algorytmie MTF Bzip2 używa RLE-0.

Nasze testy pokazały że lepsze rezultaty daje kombinacja kodowania danych wyjściowych z transformaty BWT za pomocą algorytmu distance coder a następnie użycie adaptative arithmetic.

b) Testy przeprowadzone na korpusie „silesia”.

Korpus ten jest bardziej reprezentatywny niż korpus poprzedni - pliki w tym korpusie bardziej odzwierciedlają dane które są zwykle kompresowane:

- obecnie pracujemy na dużych plikach,
- w innych korpusach używane są fragmenty tekstów tylko w języku angielskim,
- w „silesii” znajdują się pliki projektów programistycznych, zdjęć używanych w medycynie, baza danych, książek w różnych językach itp.

Dlatego też na tym korpusie przeprowadzimy dodatkowo testy DC + binary coder oraz, na niektórych plikach, sam binary coder.

Ustawienia kodera binarnego „bac”: rząd 10, wielkość słownika 1 bit;

	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
dickens (10 192 446)				
MTF+RLE0+AA	2 567 334	2.02	17.2	8
MTF+RLE0+SA	2 693 677	2.11	**15**	**5.8**
MTF+AA	2 759 001	2.17	19.3	10.2
MTF+SA	3 042 105	2.39	15.5	6.78
DC+AA	2 469 726	**1.94**	19.6	5.75
DC+BAC	3 008 783	2.36	16.7	3.77
BAC	5 637 446	4.42	177	123
mozilla (51 220 480)				

MTF+RLE0+AA	17 952 279	2.80	74.4	34.2
MTF+RLE0+SA	18512973	2.89	**66.2**	26.1
MTF+AA	19 635 592	3.07	87.1	49.3
MTF+SA	21 282 262	3.32	69.5	32.8
DC+AA	17 740 481	**2.77**	85.4	31.9
DC+BAC	23 606 106	3.69	70.6	**21.2**
BAC	36 085 735	4.85	1020	799
mr (9 970 564)				
MTF+RLE0+AA	2 363 650	1.90	16.8	6.16
MTF+RLE0+SA	2 596 556	2.08	**15.4**	4.86
MTF+AA	2 436 862	1.96	19.9	9.66
MTF+SA	3 466 974	2.78	16.4	6.52
DC+AA	2 357 576	**1.89**	19.1	5.48
DC+BAC	2959106	2.37	16.9	**3.77**
nci (33 553 445)				
MTF+RLE0+AA	1 531 160	0.365	70.5	14
MTF+RLE0+SA	1 575 370	0.374	68.4	12.6
MTF+AA	1 973 360	0.47	85.7	31.3
MTF+SA	2 481 000	0.592	72.9	20
DC+AA	1 448 015	**0.345**	72.8	12.4
DC+BAC	1 776 787	0.424	**67.7**	**10.5**
ooffice (6 152 192)				
MTF+RLE0+AA	2 874 148	3.74	19.3	4.98
MTF+RLE0+SA	2 938 563	3.82	**7.61**	3.61
MTF+AA	3 049 433	3.97	9.97	6.06
MTF+SA	3 166 484	4.12	7.83	4.09
DC+AA	2 853 902	**3.71**	11.3	4.33
DC+BAC	3 730 566	4.85	8.72	**2.69**
BAC	5 004 097	6.5	109	77
osdb (10 085 684)				
MTF+RLE0+AA	2 602 752	**2.06**	16.5	6.45
MTF+RLE0+SA	2 836 487	2.25	**15.2**	5.31
MTF+AA	3 228 843	2.56	19.8	10.2
MTF+SA	3 685 670	2.92	16.2	7.09
DC+AA	2 646 649	2.1	18.5	6.25
DC+BAC	3 549 874	2.82	16.1	**4.49**
reymont (6 627 202)				
MTF+RLE0+AA	1 145 127	1.38	10.6	4.09
MTF+RLE0+SA	1217997	1.47	**9.45**	3.14
MTF+AA	1 315 654	1.59	12.5	6.34
MTF+SA	1 547 740	1.87	10	4.14
DC+AA	1 087 506	**1.31**	11.5	3.14
DC+BAC	1 339 504	1.62	10.3	**2.22**
samba (21 606 400)				
MTF+RLE0+AA	4 427 110	1.64	35.2	11.1
MTF+RLE0+SA	4748179	1.76	32.3	8.53
MTF+AA	4 933 806	1.83	42.2	19
MTF+SA	5 651 868	2.09	34.3	12.1

DC+AA	4 338 186	**1.61**	37.7	9.31
DC+BAC	5 702 334	2.11	**30.9**	**6.45**
BAC	13 703 997	5.07	386	262
sao (7 251 944)				
MTF+RLE0+AA	5 105 017	5.63	13.9	7.75
MTF+RLE0+SA	5 392 285	5.95	**11.7**	6.27
MTF+AA	5 360 379	5.91	13.7	8.48
MTF+SA	5 782 319	6.38	11.8	6.58
DC+AA	5015304	**5.53**	15.8	9.08
DC+BAC	6815467	7.52	12.7	**6.13**
webster (41 458 703)				
MTF+RLE0+AA	7 802 311	1.51	69.5	26.5
MTF+RLE0+SA	8 086 267	1.56	**61.5**	19.9
MTF+AA	8 759 403	1.69	80.7	39.7
MTF+SA	9 958 265	1.92	65	25.9
DC+AA	7508636	**1.45**	78.9	20
DC+BAC	9172309	1.77	65.6	**13.5**
xml (5 345 280)				
MTF+RLE0+AA	434 727	0.65	8.19	2.06
MTF+RLE0+SA	444 227	0.665	7.7	1.72
MTF+AA	548 614	0.82	10.4	4.59
MTF+SA	608 177	0.91	8.36	2.83
DC+AA	422 564	**0.632**	9.3	1.63
DC+BAC	525 752	0.787	**7.66**	**1.3**
BAC	2 964 262	4.43	96	65
x-ray (8 474 240)				
MTF+RLE0+AA	3 973 931	3.75	12.8	7.25
MTF+RLE0+SA	4 495 014	4.24	**11.1**	5.64
MTF+AA	4 027 178	3.80	14.2	8.94
MTF+SA	5 099 643	4.81	11.5	6.45
DC+AA	3 954 976	**3.73**	16.3	7.26
DC+BAC	5 225 474	4.93	12.4	**4.77**
BAC	7 386 823	6.97	153	103
allsilesia (211 938 580)				
MTF+RLE0+AA	53 418 310	2.02	351	135
MTF+RLE0+SA	56 402 315	2.13	**321**	107
MTF+AA	58 817 400	2.22	426	209
MTF+SA	67 045 360	2.53	344	142
DC+AA	52 469 347	**1.98**	382	119
DC+BAC	68245599	2.58	332	**85.9**

Wnioski:

Na przykładzie pliku „allsilesia” dobrze widać przyrost wydajności jaką wniosła wersja programu 0.33 w porównaniu z wersją programu (0.24). Dla porównania przedstawiamy tabelę „allsilesia” z poprzedniej wersji sprawozdania:

allcalgarycorpus (3 141 622)					
Wersja 0.24	MTF+RLE	53 418 297	**2,02**	425	133

	0+AA				
Wersja 0.33	MTF+RLE 0+AA	53 418 310	2.02	351	135

W tym teście także nieznacznie wygrywa kombinacja DC + adaptative aritmetic. Zauważamy dość słabe wyniki kodowania binarnego(BAC), jednak gdy dane przed kodowaniem binarnym przygotujemy za pomocą kodowania odległościowego wyniki się wyraznie poprawiają jednak nie na tyle y pobić zwycięzcę.

W następnym teście porównywamy zwycięzcę(DC+AA) z popularnymi kodekami.

	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
dickens (10 192 446)				
DC+AA	2 469 726	1.94	19.6	5.75
WinRAR	2 395 134	1.88	13.1	9
Bzip2	2 799 528	2.20	7	2.3
WinRK	2 081 533	**1.63**	183	174
WinZip	3 944 251	3.10	**4**	**2**
mozilla (51 220 480)				
DC+AA	17 740 481	2.77	85.4	31.9
WinRAR	15 424 829	2.41	88.4	5.3
Bzip2	17 914 392	2.80	32.7	9.7
WinRK	13 301 823	**2.08**	1445	1225
WinZip	19 150 618	2.99	**11**	**4**
mr (9 970 564)				
DC+AA	2 357 576	1.89	19.1	5.48
WinRAR	3 202 135	2.57	25.8	**0.5**
Bzip2	2 441 280	1.96	4	1.2
WinRK	2 311 706	**1.85**	223	216
WinZip	3 712 919	2.98	**2**	1.5
nci (33 553 445)				
DC+AA	1 448 015	0.345	72.8	12.4
WinRAR	2 090 554	0.50	13.7	6.3
Bzip2	1 812 734	0.43	43.4	6
WinRK	1 366 259	**0.33**	343	310
WinZip	3 286 605	0.78	**3**	**2.5**
ooffice (6 152 192)				
DC+AA	1 448 015	0.345	72.8	12.4
WinRAR	2 302 618	2.99	9	**0.3**
Bzip2	2 862 526	3.72	3.85	1.4
WinRK	1 676 433	**2.18**	157	136
WinZip	3 113 687	4.05	**2**	1
osdb (10 085 684)				
MTF+RLE0+AA	2 602 752	2.06	26.5	7
WinRAR	3 189 081	2.53	26.3	**1**
Bzip2	2 802 792	2.22	7.5	2.3
WinRK	2 327 107	**1.85**	154	151
WinZip	3 790 933	3.01	**2**	**1**

reymont (6 627 202)				
DC+AA	1 087 506	1.31	11.5	3.14
WinRAR	1 071 622	1.29	5.2	3.5
Bzip2	1 246 230	1.50	4.3	1.5
WinRK	962 611	**1.16**	85	75
WinZip	1 928 524	2.33	**1.5**	**1**
samba (21 606 400)				
DC+AA	4 338 186	1.61	37.7	9.31
WinRAR	3 894 150	1.44	17.7	8.5
Bzip2	4 549 790	1.68	13.8	3.5
WinRK	3 547 240	**1.31**	395	367
WinZip	5 504 361	2.04	**4**	**1.8**
sao (7 251 944)				
DC+AA	5 015 304	5.53	15.8	9.08
WinRAR	5 563 384	6.14	11.6	**0.4**
Bzip2	4 940 524	5.45	6.5	2
WinRK	3 899 159	**4.30**	429	445
WinZip	5 370 299	5.92	**2.5**	0.7
webster (41 458 703)				
DC+AA	7 508 636	1.45	78.9	20
WinRAR	7 172 998	1.38	11.6	24.5
Bzip2	8 644 714	1.67	26.2	7
WinRK	6 026 007	**1.16**	733	693
WinZip	12 405 376	2.39	**7.5**	**2.5**
xml (5 345 280)				
DC+AA	422 564	0.632	9.3	1.63
WinRAR	407 763	0.61	2.7	1.3
Bzip2	441 186	0.66	4.2	0.82
WinRK	322 368	**0.48**	47	43
WinZip	716 701	1.07	**1**	**1**
x-ray (8 474 240)				
DC+AA	3 954 976	3.73	16.3	7.26
WinRAR	4 134 254	3.90	3	**0.8**
Bzip2	4 051 112	3.82	4.3	1.8
WinRK	3 926 189	**3.71**	363	397
WinZip	6 046 173	5.71	**2**	1
allsilesia (211 938 580)				
DC+AA	52 469 347	1.98	382	119
WinRAR	50 808 227	1.92	225	62.2
Bzip2	54 573 629	2.06	142	37.1
WinRK *	43 611 153	**1.65**	4 472	4 160
WinZip	68 979 726	2.60	**34**	**16**

Wnioski:

Wnioski są identyczne jak w przypadku porównania „Calgary Corpus”.

c) Testy przeprowadzone na plikach skrajnych:

- plik a.txt składa się z literek „a”.

- plik ab.txt składa się z sekwencji „ab”.
- plik ala.txt składa się z sekwencji „ala ma kota”.
- plik random.txt posiada wygenerowaną pseudolosową zawartość.

	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
a.txt (10 485 760)				
MTF+RLE0+AA	51	**0.000039**	8.05	0.55
MTF+RLE0+SA	551	0.000423	8.14	0.516
MTF+AA	41 215	0.031442	13.7	6.59
MTF+SA	583	0.000445	9.64	3.06
DC+AA	56	0.000040	**8.02**	**0.391**
ab.txt (10 485 760)				
MTF+RLE0+AA	61	0.000047	9.97	0.532
MTF+RLE0+SA	552	0.000426	9.78	0.531
MTF+AA	41 217	0.031443	15.3	6.59
MTF+SA	585	0.000446	11.3	3.06
DC+AA	59	**0.000046**	**9.66**	**0.39**
ala.txt (10 485 760)				
MTF+RLE0+AA	125	0.000095	25	0.547
MTF+RLE0+SA	585	0.000485	24.7	0.531
MTF+AA	41 233	0.000637	30.2	6.61
MTF+SA	834	0.000637	26.3	3.08
DC+AA	98	**0.000083**	**24.6**	**0.406**
random.txt (10 485 760)				
MTF+RLE0+AA	10 140 048	7.736	21.5	13.7
MTF+RLE0+SA	10 131 913	7.730	19.6	**11.9**
MTF+AA	10 139 835	7.736	21.2	13.6
MTF+SA	10 131 731	**7.730**	**19.4**	**11.9**
DC+AA	10 389 052	7.93	25.2	20

Wnioski:

Ten test służył tylko sprawdzeniu czy nasz program jest odporny na pliki o zawartości mogącej go potencjalnie zawiesić, tzn wprowadzić algorytm sortowania w bardzo głęboką rekurencję, co w praktyce mogłoby oznaczać zupełnie nieakceptowalny czas kompresji. Test wypadł pomyślni, nasz koder bardzo ładnie sobie z takimi szczególymi plikami poradził.

Test porównawczy kompresorów na plikach skrajnych:

	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
a.txt (10 485 760)				
MTF+RLE0+AA	51	0.000039	8.05	0.55
WinRAR	5 205	0.003971	1.7	0.21
Bzip2	49	**0.000037**	**0.8**	**0.17**
WinRK	970	0.000740	50	47
WinZip	10 299	0.007858	1	1

ab.txt (10 485 760)				
DC+AA	59	**0.000046**	9.66	**0.39**
WinRAR	5 264	0.004016	1.7	0.8
Bzip2	373	0.000285	20.3	0.6
WinRK	969	0.000739	51	50
WinZip	10 303	0.007861	**1**	1
ala.txt (10 485 760)				
DC+AA	98	**0.000083**	24.6	**0.406**
WinRAR	5 344	0.004077	1.9	0.76
Bzip2	798	0.000609	38	0.7
WinRK	1 165	0.000889	58	57
WinZip	20 476	0.015622	**1**	1
random.txt (10 485 760)				
MTF+SA	10 131 731	**7.730**	19.4	11.9
WinRAR	10 485 834	8.00	20.3	0.13
Bzip2	10 530 234	8.03	12.5	3.6
WinRK	10 680 775	8.15	1177	1098
WinZip	10 487 478	8.00	**3**	**1**

d) Testy przeprowadzone na plikach obrazów.
Jako testowe obrazy przyjeliśmy obrazy „Lena” w trzech wersjach:

- kolorowej (lena_std.tif 512 x 512 x 24 BPP) .
- w odcieniach szarości (lena_grey.tif 512 x 512 x 8 BPP)
- czarno-białej (lena_bw.bmp 512 x 512 x 1 BPP)

	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
lena_std.tiff (786 572)				
MTF+RLE0+AA	590 354	**6.004**	1.03	0.844
MTF+RLE0+SA	595 654	6.058	**0.781**	0.594
MTF+AA	591 050	6.011	1.03	0.844
MTF+SA	596 373	6.066	**0.781**	**0.578**
DC+AA	591 447	6.02	1.56	0.75
lena_grey.tif (264 168)				
MTF+RLE0+AA	177 076	5.363	0.281	0.25
MTF+RLE0+SA	179 175	5.426	**0.187**	0.172
MTF+AA	177 433	5.373	0.312	0.25
MTF+SA	179 570	5.438	**0.187**	**0.156**
DC+AA	176 362	**5.360**	0.468	0.203
lena_bw.bmp(32 830)				
MTF+RLE0+AA	18 410	4.49	0.031	0.031
MTF+RLE0+SA	18 787	4.58	0.031	0.015
MTF+AA	19 337	4.71	0.031	0.031
MTF+SA	19 811	4.83	**0.015**	**0.016**
DC+AA	17 715	**4.32**	0.062	**0.016**

Wnioski:

Najlepsze wyniki osiągamy za pomocą wariantu kompresji DC+AA

	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
lena_std.tif (786 572)				
MTF+RLE0+AA	590 354	6.004	1.03	0.844
WinRAR	481752	4.90	**0.6**	**0.1**
Bzip2	584468	5.94	**0.6**	0.2
WinRK	526700	**5.36**	27	26
WinZip	733485	7.46	2	1
lena_grey.tif (264 168)				
DC+AA	176 362	5.360	0.468	0.203
WinRAR	168 448	5.10	**0.12**	**0.04**
Bzip2	174 447	5.28	0.17	0.08
WinRK	167 144	**5.06**	8	7
WinZip	223 635	6.77	<1	<1
lena_bw.bmp(32 830)				
DC+AA	17 715	4.32	0.062	**0.016**
WinRAR	17 571	4.28	**0.03**	0.03
Bzip2	17 811	4.34	1	1
WinRK	16 551	**4.03**	7	6
WinZip	18 503	4.51	1	1

Ponownie najlepszy okazał się WinRK.

e) Test badający zależność stopnia kompresji od rozmiaru okna BWT. Test zostanie przeprowadzony na zwycięzcy (BWT + DC + AA), wielkość bloku zmieniamy od 1 Kb do całego rozmiaru pliku z krokiem ^2 (1Kb. 2Kb. 4Kb. 16Kb....).

Rozmiar okna BWT [Kb]	rozmiar po kompresji	średnia bitowa	czas kompresji (s)	czas dekompresji (s)
allcalgarycorpus (3 141 622)				
1	1 696 885	4.32	3.83	3.69
2	1 481 334	3.77	3.33	2.41
4	1 316 443	3.35	3.09	1.83
8	1 190 558	3.03	2.97	1.42
16	1 094 942	2.79	2.94	1.19
32	1 019 484	2.6	3.17	1.08
64	961 041	2.45	2.86	1.05
128	914 436	2.33	3.14	1.14
256	891 818	2.27	3.47	1.22
512	871 773	2.22	3.84	1.28
1024	862 773	2.2	4.2	1.36
2048	859 989	2.19	4.42	1.38
3141	861 148	2.19	4.59	1.44
4092	861 148	2.19	4.61	1.42

dickens (10 192 446)				
1	6 299 616	4.94	15.5	12.2
2	5 487 887	4.31	14.3	7.76
4	4 871 178	3.82	12.9	5.92
8	4 393 763	3.45	12.4	4.72
16	4 008 650	3.15	11.7	4.03
32	3 694 972	2.9	11.5	4.3
64	3 434 636	2.7	12.3	3.59
128	3 208 875	2.5	12.6	3.98
256	3 019 270	2.37	13.7	4.39
512	2 864 002	2.25	15.4	4.67
1024	2 728 439	2.14	16.6	4.83
2048	2 621 946	2.06	17.4	4.94
4092	2 556 958	2.01	18.1	5.13
6144	2 533 841	1.99	18.7	5.24
8192	2 517 244	1.98	19.2	5.28
10240	2 469 726	1.94	20.1	5.42

samba (21 606 400)				
1	10 649 876	3.94	27.6	23.9
2	9 034 768	3.35	23.8	15
4	7 847 258	2.91	21.7	11.4
8	6 968 159	2.58	20.2	8.81
16	6 287 917	2.33	19.5	7.47
32	5 757 402	2.13	19.5	6.66
64	5 341 941	1.98	19.8	6.47
128	5 046 303	1.87	22.4	7.08
256	4 841 271	1.79	25.2	7.08
512	4 682 494	1.73	31	7.5
1024	4 618 408	1.71	32.5	7.81
2048	4 547 436	1.68	33.4	8.11
4092	4 427 763	1.64	34.7	8.38
8192	4 356 896	1.61	37.2	8.69

Wnioski:

Test ten pokazuje że wielkość bloku BWT ma wpływ zarówno na stopień kompresji pliku jak i na czas tej kompresji. W przypadku pliku **allcalgarycorpus** skuteczność kompresji do pewnego momentu się poprawia tj. do wielkości bloku równej 2 MB następnie dalsze zwiększanie okna nie ma sensu - prowadzi tylko do pogorszenia się statystyk czasowych kompresji. W przypadku dwóch pozostałych plików stopień kompresji wyraźnie poprawia się aż do momentu, gdy rozmiar bloku jest równy rozmiarowi pliku. Przyczyna powyższego jest taka, iż pliki dickens i samba zawierają dosyć jednorodne pod względem charakterystyki dane, co powoduje, iż zwiększanie rozmiaru bloku daje większe możliwości grupowania podobnych kontenstów, natomiast plik **allcalgarycorpus** jest konkatenacja plików o różnych charakterystykach. Nadal jednak optymalny rozmiar bloku jaki szacujemy dla tego pliku na 1MB jest większy niż średni rozmiar pliku w korpusie calgary (176kb), jak i niz maksymalny rozmiar pliku (768kb).

Prędkość kompresji wraz ze wzrostem rozmiarów boku spada nieznacznie. Ujawnia to delikatnie nieliniową charakterystykę algorytmu sortowania.

Historia modyfikacji programu:

wersja 1-7: praca nad BWT i jego optymalizacją(eliminacja niepotrzebnych tablic i implementacja algorytmu Sadakane)
wersja 8: dodaliśmy implementację transformaty MTF
wersja 9-11: dodaliśmy własny koder arytmetyczny symboli z kontekstem rzędu 0 , oraz niezbędną do jego działania klasy wejścia/wyjścia bitowego
wersja 12: zdecydowaliśmy się na zmianę koncepcji kodera arytmetycznego (z kolejką bitową) na wersję identyczną z omawianą na wykładzie.
wersja 12-15: dodaliśmy zapisywanie i czytanie z pliku (do tej pory koder zapisywał zakodowaną sekwencję do tablicy w pamięci)
wersja 16-17: dodawanie RLE-0
wersja 18: podział pliku wejściowego na bloki oraz kodowanie arytmetyczne statyczne.
wersja 19: optymalizacja sortowania (zoptymalizowane sortowanie dla krótkich sekwencji 2-4 elementy). Próba ustabilizowania sortowania za pomocą median-3
wersja 20-21: usuwanie błędów, zmiany kosmetyczne
wersja 22: poprawa implementacji - przerobienie klasy sortowania na wersje z użyciem szablonów
wersja 23-25: próba wyeliminowania dziwnego zachowania się sortowania
wersja 26-27: dodanie kodera DC
wersja 28-32: optymalizacja algorytmu sortowania: wstępne sortowanie po pierwszej literce zostało zastąpione sortowaniem po k-pierwszych literek(eksperymenty pokazały że optymalnie k=4)
wersja 33: optymalizacja kodera DC
wersja 34: dla sekwencji 10Mb literek „a” sortowanie generuje sekwencję „median-3killer” powoduje to wpadanie w bardzo głęboką rekurencję – powrót do poprzedniej wersji sortowania (za medianę uznajemy środkowy element zbioru)
wersja 35-36: eksperymenty z koderem arytmetycznym rzędu 1 – nieznaczna poprawa średniej bitowej, na tej wersji nie przeprowadziliśmy wszystkich testów
wersja 37: dalsza optymalizacja kodowania DC (zamiana tablicy na listę symboli oraz ich odległości co zostało opisane w punkcie „kodowanie odległościowe” tego sprawozdania)
wersja 38-39: zmiany kosmetyczne, pielęgnacja kodu.