POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Łukasz Dziedzic
Nr albumu: 160759
Studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka
Specjalność: Sieci komputerowe

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Implementacja i analiza jakości wybranych algorytmów kompresji danych

Tytuł pracy w języku angielskim: Implementation and analysis of quality of selected data compression algorithms

Potwierdzenie przyjęcia pracy	
Opiekun pracy *podpis*	Kierownik Katedry/Zakładu (pozostawić właściwe) *podpis*
dr hab. inż. Robert Janczewski	

Data oddania pracy do dziekanatu:

POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

OŚWIADCZENIE dotyczące pracy dyplomowej zatytułowanej: Implementacja i analiza jakości wybranych algorytmów kompresji danych

Imię i nazwisko studenta: Łukasz Dziedzic
Data i miejsce urodzenia: 17.09.1996, Gdynia
Nr albumu: 160759

Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki
Kierunek: informatyka

Poziom kształcenia: drugi
Forma studiów: stacjonarne

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. 2018 poz. 1191 z późn. zm.) i konsekwencji dyscyplinarnych określonych w ustawie z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. 2018 poz. 1668 z późn. zm.),[1] a także odpowiedzialności cywilnoprawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia ....
podpis studenta

[1] Ustawa z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce:

Art. 312. ust. 3. W przypadku podejrzenia popełnienia przez studenta czynu, o którym mowa w art. 287 ust. 2 pkt 1–5, rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.

Art. 312. ust. 4. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 5, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o podejrzeniu popełnienia przestępstwa.

STRESZCZENIE

Celem projektu magisterskiego było przeanalizowanie oraz zaimplementowanie wybranych istniejących algorytmów kompresji danych. W pracy w szczególności skupiono się na algorytmach kompresji bezstratnej, jednak zostały w niej także opisane podstawowe aspekty kompresji stratnej. Projekt składa się z dwóch części - programu oraz pracy.

Praca dyplomowa magisterska jest podzielona na kilka rozdziałów. Na samym początku poruszona jest tematyka definiująca czym jest kompresja oraz na co się dzieli. W następnej części dokonano pomiarów jakości wybranych algorytmów kompresji bezstratnej oraz wpływu różnych rozszerzeń formatu plików na końcowy efekt kompresji. Następnie skupiono się na szczegółowej analizie wybranych metod i algorytmów kompresji. Ostatnią część stanowi opis implementacji algorytmu Deflate.

Drugą częścią pracy jest program, w którym dokonano implementacji jednego z najpopularniejszych algorytmów kompresji - Deflate. Program został napisany w języku Java, a do jego zrobienia wykorzystano narzędzie kontroli wersji git.

ABSTRACT

The aim of the master's project was to analyze and implement selected data compression algorithms. The master's thesis focuses in particular on lossless compression algorithms but also describes the basic aspects of lossy compression. The project consists of two parts - the program and the thesis.

The master's thesis is divided into several chapters. At the beginning, there is a topic that defines what compression is and how it is divided. The next part focuses on measuring the quality of selected lossless compression algorithms and the impact of various file format extensions on the final compression effect. Then, the next chapter focuses on a detailed analysis of selected compression methods and algorithms. The last part describes the implementation of the Deflate algorithm.

The program that implements one of the most popular compression algorithms - Deflate is the second part of the project. The code was written in Java and the git version control tool was used to make it.

SPIS TREŚCI

WYKAZ WAŻNIEJSZYCH OZNACZEŃ I SKRÓTÓW

RLE – Run-length encoding - Kodowanie długości serii

DCT – Discrete Cosine Transform - Dyskretna transformacja kosinusowa

$MDCT$ – Modified discrete cosine transform - Zmodyfikowana dyskretna transformacja kosinusowa

FFT – Fast Fourier transform - Szybka transformacja Fouriera

$MP3$ – Moving Picture Experts Group Audio Layer 3 - Format zapisu plików dźwiękowych wykorzystujących kompresję stratną. Nazwa MP3 wywodzi się od skrótu nazwy grupy roboczej ISO/IEC zajmującej się rozwojem kodowania plików audio i wideo.

$LZ77$ – Lempel-Ziv 77 - Algorytm słownikowej kompresji bezstratnej opracowany przez Abrahama Lempela i Jacoba Ziv w 1977.

$LZMA$ – Lempel-Ziv-Markov Chain Algorithm - Algorytm kompresji bezstratnej bazujący na metodzie LZ77 i wykorzystujący proces Markowa.

PPM – Prediction by Partial Matching - Przewidywanie przez częściowe dopasowanie

1. WSTĘP

1.1. Co będzie w pracy?

Praca magisterska będzie dotyczyć analizy i implementacji algorytmów kompresji bezstratnej. Zostaną omówione w niej zarówno podstawowe aspekty kompresji, analiza konkretnych algorytmów, jak i opis implementacji jednego z algorytmów.

Na samym początku zostaną omówione podstawowe aspekty kompresji. Pierwszy podrozdział części teoretycznej będzie odpowiadać na pytania czym jest kompresja i dlaczego się ją stosuje. W dalszej części zostanie dokonany podział algorytmów kompresji na dwa główne typy - kompresję stratną i bezstratną, a także zostaną pokazane przykłady dotyczące odpowiedniego doboru typu do różnych sytuacji. Poza podziałem na kompresję stratną i bezstratną omówione zostaną również dwie metody redukcji rozmiaru danych wykorzystywanych podczas wykonywania kompresji. Pierwsza to metoda słownikowa, druga to metoda statystyczna.

Kolejny podrozdział będzie się skupiał na analizie jakości wybranych algorytmów. Porównane zostaną algorytmy Deflate, BZIP2, LZMA i PPMd pod względem stopnia kompresji i czasu działania programu podczas przetwarzania plików. Każdy z algorytmów zostanie przetestowany dla plików tekstowych, binarnych, surowych plików graficznych i liczb pseudolosowych. Po przeprowadzeniu testów zostaną omówione wyniki analizy.

Następny rozdział będzie się skupiał na podstawowych technikach kompresji. Postanie przeanalizowana metoda statystyczna, metoda słownikowa oraz transformaty, które przekształcają dane w sposób ułatwiający późniejszą kompresję. Do analizy zostaną wykorzystane kodowania Run-Length Encoding, Huffmana, metoda LZ77 i transformata Burrowsa-Wheelera.

W dalszej części zostaną szczegółowo opisane dwa algorytmy - Deflate oraz LZMA. W dwóch podrozdziałach zostanie przedstawione jak krok po kroku zaimplementować każdy z tych algorytmów.

Ostatni rozdział będzie zawierał informacje na temat własnej implementacji algorytmu Deflate. Na początku będzie ogólny opis programu, jego struktury, a w dalszej części zostaną również opisane szczegółowo kluczowe klasy oraz metody. Na samym końcu będą wyniki jakości działania programu.

1.2. Cel pracy

Praca magisterska zakłada trzy główne cele.

- Pierwszy to szczegółowe zapoznanie się z tematyką kompresji algorytmów. W tym celu zostanie przedstawiony ogólny zarys kompresji, kilka algorytmów zostanie omówionych ogólnie oraz zostaną także szczegółowo opisane dwa z nich.
- Drugim celem jest zbadanie jakości poszczególnych algorytmów. Do przeprowadzenia eksperymentów zostaną wykonane testy kompresji plików binarnych, tekstowych, surowych plików graficznych i liczb pseudolosowych dla różnych algorytmów kompresji bezstratnej. Do testów zostaną wykorzystane algorytmy, które wchodzą w skład formatów .7z oraz .zip.
- Trzecim celem będzie napisanie programu umożliwiającą kompresję danych. Poza samym

napisaniem programu istotne będzie zbliżenie się jakością programu do rozwiązań ogólnodostępnych.

2. CZĘŚĆ TEORETYCZNA

2.1. Kompresja - co to jest?

Celem projektu magisterskiego jest implementacja i analiza jakości wybranych algorytmów kompresji danych. Aby zająć się problematyką, na samym początku powinno się zdefiniować czym jest kompresja i dlaczego się ją stosuje.

Z dnia na dzień można zauważyć wzrost mocy obliczeniowej komputerów, telefonów, cyfrowych aparatów fotograficznych, czy innych urządzeń elektronicznych. Powoduje to nieustanne zwiększanie generowania, przetwarzania, transmitowania oraz przechowywania danych. To z kolei sprawia, że potrzebujemy zastosować m.in. mechanizmy redukujące wielkość danych zwane kompresją. Przeglądając strony internetowe stosujemy kompresję, oglądając filmy czy słuchając muzyki korzystamy z kompresji, rozmawiając przez telefon kompresujemy dane, zarządzając plikami na dysku komputera również wykorzystujemy algorytmy kompresji. Jej zastosowań może być nieskończenie wiele.

Czym więc jest kompresja danych? Jest to proces polegający na zakodowaniu informacji w taki sposób, aby zachować oryginalną treść wiadomości przy jednoczesnym zmniejszeniu jej objętości. Osiąga się to poprzez zapisanie pliku tekstowego, utworu muzycznego, filmu, obrazu, czy innego rodzaju danych za pomocą mniejszej liczby bitów. Wyróżniamy dwa typy kompresji - bezstratną oraz stratną. W pierwszym typie dokonuje się redukcji bitów na podstawie analizy statystycznej występowania danych znaków lub ich sekwencji. W tym przypadku można odzyskać wszystkie dane po wykonaniu dekompresji na nich. W drugim typie kompresji redukuje się bity poprzez usunięcie niepotrzebnych lub mało istotnych fragmentów informacji. Po zastosowaniu tego typu kompresji nie jest możliwe późniejsze przywrócenie wszystkich walorów danych. Przykładowo jeżeli porównamy utwór muzyczny zapisany w sposób nieskompresowany oraz znacząco skompresowany za pomocą algorytmów kompresji stratnej, to prawdopodobnie nie uda się zauważyć różnicy słuchając go na głośnikach i wzmacniaczu niskiej jakości, jednak jest duże prawdopodobieństwo, że uda się różnice usłyszeć słuchając go za pomocą sprzętu audiofilskiego.

Wiadomo już czym jest kompresja. Tylko dlaczego powinno się ją stosować? Dobrym przykładem jest wykorzystanie kompresji w internecie. Patrząc z punktu widzenia właściciela strony internetowej istotne jest, aby dane użytkownika zajmowały mało miejsca na serwerze oraz aby w każdym momencie przepustowość łącza pozwalała na transfer danych użytkowników. Często ceny hostingów w środowiskach chmurowych zależą właśnie od tych parametrów. Użytkownikom z kolei zależy na szybkim przesłaniu informacji oraz przesłaniu możliwie małej ilości danych. Koszt przechowywania skompresowanych danych jest mniejszy, a ich czas transmisji jest krótszy. Niezależnie od tego jak szybką sieć ma dany użytkownik oraz jak duży ma pakiet danych do wykorzystania, zawsze będzie mógł zaoszczędzić trochę czasu na pobieraniu. Firma DoubleClick, należąca do Google, przeprowadziła badania odnośnie korelacji użytkowników odwiedzających strony i czasem ich wczytywania. Okazało się, że jeżeli wczytywanie strony mobilnej trwa dłużej niż 3 sekundy, to z prawdopodobieństwem 53% użytkownik zrezygnuje z jej wyświetlenia. Właśnie dlatego stale kompresujemy na stronach filmy, obrazy, czy tekst np. pliki JavaScript.

2.2. Podział algorytmów kompresji

Algorytmy kompresji możemy podzielić na dwa typy - algorytmy kompresji stratnej oraz bezstratnej. Załóżmy, że algorytm dostaje na wejściu informację A. Kompresuje go do postaci B. Następnie algorytm dekompresji przekształca zbiór do C. W przypadku kompresji bezstratnej A jest równe C. W przypadku stratnej A jest różne od C.

2.2.1. Kompresja bezstratna

Kompresja bezstratna charakteryzuje się tym, że po wykonaniu kompresji danych, a następnie ich dekompresji, można odtworzyć początkową postać danych, czyli wszystkie bity danych będą identyczne. Jest stosowana, gdy niezbędne jest odtworzenie informacji w niezmienionej formie. Przykładowo jest stosowana dla plików binarnych. Gdyby zmieniono poszczególne bity programu, jest bardzo prawdopodobne, że nie udałoby się go uruchomić lub zawierałby błędy. Podobnie jest z tekstem. Przykładowo jeżeli dokonuje się kompresji strony internetowej zawierającej między innymi numer konta bankowego, to zmiana nawet jednego bitu w ciągu cyfr mogłaby znacząco zaszkodzić właścicielowi danej strony.

Algorytmy kompresji bezstratnej wykorzystują fakt, że w danych znajduje się znaczna ilość redundantnych informacji. Przykładowo można założyć, że w tekście wykorzystywane są głównie litery alfabetu. Jeżeli przyjmie się uproszczony wariant, że autor wiadomości posługuje się tylko alfabetem łacińskim i korzysta się głównie z małych liter, to można łatwo zmniejszyć objętość do około 25% - 30% oryginalnej wiadomości stosując jedynie lepiej dobrane kodowanie.

Nie wszystkie dane można skompresować za pomocą algorytmów kompresji bezstratnej. W każdym przypadku można wytworzyć skompresowany plik, jednak jednak jego rozmiar może być taki sam jak przed kompresją. Nie jest możliwe efektywne skompresowanie strumienia liczb losowych. Jednocześnie dokonanie kompresji takich liczb czy już skompresowanych danych przez inny algorytm jest trudne.

Algorytmy kompresji bezstratnej wykorzystują głównie dwie metody redukcji rozmiaru danych. Pierwsza to słownikowa, druga to statystyczna.

Metoda słownikowa polega na znajdowaniu powtarzających się ciągów znaków w danym tekście i zastępowanie ich pozycją ze słownika. Dzięki temu długie sekwencje danych zastępowane są znacząco krótszymi kodami adresów. Można wyróżnić dwa rodzaje słowników. Pierwszy to statyczny, drugi to dynamiczny. W pierwszym przypadku słownik będzie zawierał przez cały czas stałą liczbę słów. Mogą to być wcześniej przygotowane najczęściej występujące słowa lub skróty z danego języka. W tej metodzie jedynie słowa występujące w słowniku mogą być skompresowane. W przypadku braku danego słowa, element z tekstu nie będzie skompresowany. W drugiej metodzie słownik początkowo nie zawiera żadnej zawartości. Kolejne słowa są uzupełniane na bieżąco wraz z kolejnymi wczytanymi bitami tekstu. Maksymalna liczba słów jest zdefiniowana przez programistę implementującego dany algorytm. Główną przewagą kodowania ze statycznym słownikiem jest większa szybkość działania algorytmu względem kodowania ze słownikiem dynamicznym. Główną wadą jest mniejsza kompresja względem drugiego rozwiązania.

Metoda statystyczna polega na zapisaniu symboli za pomocą mniejszej liczby bitów.

Liczba bitów obliczana jest na podstawie częstotliwości występowania danych znaków. Im symbol częściej występuje, tym będzie zawierał mniej bitów w zakodowanej postaci. Przykładowo w statystycznym tekście napisanym w języku polskim litera “a” będzie zdecydowanie częściej występować niż litera “f”, stąd też jej kod w wielu przypadkach będzie krótszy. Powyższe stwierdzenie zostało oparte na frekwencji występowania symboli ze słownika języka polskiego PWN [29].

Najbardziej popularne algorytmy kompresji bezstratnej zostaną szczegółowo opisane w dalszej części pracy magisterskiej.

2.2.2. Kompresja stratna

Istnieją sytuacje, w których nie jest bardzo istotne, aby dało się odtworzyć każdy bit skompresowanych danych. Przykładowo w przypadku zdjęć, filmów, czy utworów muzycznych istotne jest, aby człowiek nie był wstanie dostrzec różnicy, jednak zupełnie nieistotne jest czy finalna postać pliku jest identyczna jak początkowa. W takich przypadkach można zastosować kompresję stratną, cechującą się dużo wyższym poziomem kompresji.

Aby odpowiednio skompresować plik za pomocą kompresji stratnej, tak aby człowiek nie był wstanie dostrzec różnicy lub aby wykrywał niewielką różnice, stosuje się modele modele psychoakustyczne oraz psychowizualne. Modele dają informacje jakie dane są bardziej istotne od innych dla percepcji człowieka. Przykładowo wykorzystują informację o tym, że ludzkie oko bardziej rozpoznaje poziom natężenia światła od barwy danego obiektu.

Najprostszymi przykładami kompresji stratnej wydają się być np. zapisanie informacji z co drugiego piksela obrazka lub odcięcie najmniej znaczących bitów odpowiadających za barwę obrazu/dźwięku. Jednak taka kompresja nie da zadowalających efektów i człowiek z łatwością będzie mógł dostrzec różnice pomiędzy oryginalnym a skompresowanym materiałem.

Warto opisać zasadę działania najbardziej popularnych algorytmów do kompresji obrazów czy plików muzycznych.

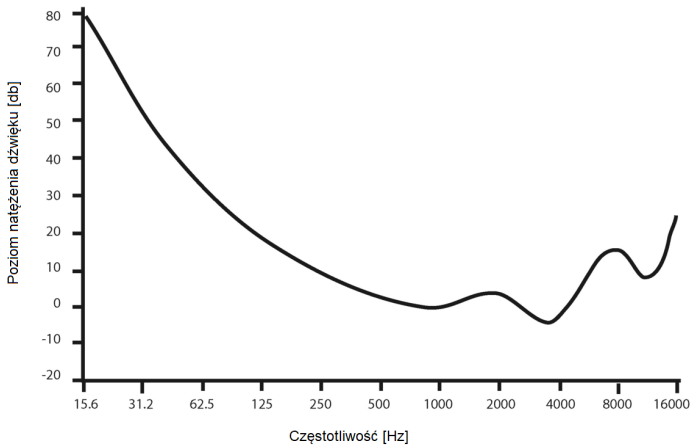
2.2.3. Kompresja stratna - pliki graficzne

Do najczęściej używanych algorytmów kompresji grafiki rastrowej należy algorytm JPEG. Przebiega w sześciu kolejnych krokach. Najpierw zamienia standardową przestrzeń barw RGB na jasność oraz 2 kanały barw. Wyodrębnienie jasności wynika z charakterystyki ludzkiego rozpoznawania obrazów. Dużo łatwiej jest człowiekowi wykryć różnice poziomu jasności niż różnic barw. W następnym kroku odrzucana jest część pikseli odpowiadających za barwy na obrazie. Redukcji można dokonać w trzech trybach - brak zamiany pikseli, zamiana 2 pikseli barwy w 1 oraz zamiana 4 pikseli w 1. W kolejnym kroku następuje podział kanałów na bloki 8×8 lub 16×16 pikseli, a następnie dokonuje się transformaty kosinusowej DCT na blokach. Kanał w tym przypadku oznacza wartość danego koloru. Przykładowo w przypadku, gdy kompresowany obrazek jest zapisany w 24-bitowym formacie zapisu barw RGB, kanał każdego koloru będzie miał wartość w przedziale od 0 do 255. Transformatę można odwrócić. Wynikiem tego będzie zmiana wartości pikseli na na średnią wartość oraz częstości ich zmian, zapisane za pomocą liczb zmiennoprzecinkowych. Następny krok odpowiada za stopień kompresji. W zależności od podanych parametrów algorytmów mniejsza lub

większa ilość danych zostanie odrzucona. Dokonuje się tam zamiany liczb zmiennoprzecinkowych na całkowite. Na koniec koduje się dane za pomocą algorytmu Huffmana opisanego w dalszej części pracy magisterskiej. Dużą zaletą algorytmu jest możliwość doboru parametrów na bieżąco podczas kompresji. Wynikiem tego może być optymalny stosunek jakości obrazu do stopnia kompresji.

2.2.4. Kompresja stratna - pliki dźwiękowe

Warto także opisać jeden z najbardziej popularnych algorytmów kompresji plików dźwiękowych. Kompresja plików audio polegająca na odcięciu kolejnych bitów lub zawężeniu przedziału słyszalnego dla ucha człowieka (około 16 Hz - 22 kHz) dałaby niezadowalające rezultaty. Z tego powodu do kompresji dźwięku stosuje się modele psychoakustyczne, które wykorzystują właściwości ludzkiego słuchu i za pomocą których można usunąć dźwięki niesłyszalne lub słyszalne w niewielkim stopniu przez człowieka. Algorytm MP3 (MPEG-1/MPEG-2 Audio Layer 3) działa w oparciu o model psychoakustyczny. Na poniższym rysunku przedstawione jest postrzeganie dźwięku przez człowieka. Dla zaznaczonych wartości na wykresie, człowiek będzie słyszał dźwięk na takim samym poziomie głośności.



Rys. 2.1. Postrzeganie dźwięku przez człowieka

W pierwszym kroku algorytm dzieli pasmo pliku wejściowego na mniejsze kawałki i stosuje się na nich filtr dyskretnej transformaty kosinusowej (MDCT). Następnie wyjście przekształca się za pomocą szybkiej transformaty Fouriera (FFT), dopasowuje je do modelu psychoakustycznego i ponownie stosuje się na nich filtr dyskretnej transformaty kosinusowej (MDCT). W kolejnym kroku kwantyfikuje i koduje się dane, aby na końcu utworzyć strumień bitów audio, składających się z ramek z 4 parametrami - nagłówka, kodu ułatwiającego wykrywania błędów transmisji, danych audio i danych pomocniczych.

2.3. Wyniki pomiaru czasu i stopnia kompresji wybranych algorytmów i programów kompresujących

Obecnie dostępnych jest wiele programów umożliwiających kompresję danych zarówno darmowych, jak i komercyjnych. Programy różnią się między sobą możliwościami, prędkością działania oraz możliwym stopniem kompresji. W tym rozdziale znajdą się wyniki działania jednego z

najbardziej popularnych programów - 7z. Zostanie przetestowana kompresja dla różnych rodzajów plików, algorytmów kompresji oraz ich parametrów.

2.3.1. Przygotowanie danych

Aby można było ocenić jakość poszczególnych algorytmów, niezbędne jest przygotowanie różnych typów danych. Do przeprowadzenia eksperymentów wykorzystano cztery rodzaje plików. Każdy rodzaj plików będzie zajmował objętość w przybliżeniu równą 1 GB.

Pierwszy z nich to tekst. Kompresja danych z generatora losowych liter nie dałaby w tym przypadku istotnych wyników, ze względu na niewielkie odzwierciedlenie w rzeczywistości. W związku z tym w eksperymencie będą użyte losowe strony z Wikipedii. Do pobrania zbioru zdań i słów wykorzystano poniższy skrypt napisany w języku Python. Generator opiera się na bibliotece „wikipedia"[28].

Listing 2.1. Skrypt do pobierania tekstu z losowych stron z Wikipedii

```
import wikipedia

EXPECTED_TEXT_SIZE = 1000000

def get_page_summary(page_name: str)  > str:
    try:
        return wikipedia.page(page_name).summary
    except wikipedia.exceptions.DisambiguationError as e:
        summary = ''
        for page_name in e.options:
            summary += get_page_summary(page_name)
        return summary
    except wikipedia.exceptions.PageError:
        return ''

def get_random_pages_summary(pages: int = 0)  > str:
    text = ''
    page_names = [wikipedia.random(1) for _ in range(pages)]
    for page_name in page_names:
        for page_summary in get_page_summary(page_name):
            text += page_summary
    return text

def save_output_file(text: str)  > None:
    with open('random_wiki_pages.txt', 'a+') as file:
        text = str(text.encode('UTF 8 '))
        file.write(text)
```

```
text = ''
while len(text) < EXPECTED_TEXT_SIZE:
    text += get_random_pages_summary(5)
    save_output_file(text)
```

Drugim rodzajem pliku będą liczby pseudolosowe. Teoretycznie powinny być one trudne w kompresji. Do ich wygenerowania dopisano do wyżej przedstawionego skryptu metodę generującą liczby pseudolosowe przedstawioną poniżej.

Listing 2.2. Skrypt do generowania liczb pseudolosowych

```
def generate_random_numbers(numbers: int, max_number: int)  > str:
    number_list = []
    for i in range(0, numbers):
        number_list.append(random.randint(1, max_number))
    return '\n'.join(number_list)
```

Trzecim rodzajem danych będą zdjęcia w surowym formacie RAW. Jedno zdjęcie zajmuje około 25 MB przestrzeni na dysku.

Ostatnim typem danych będą pliki binarne. W tym celu zostaną pobrane obrazy systemu operacyjnego Ubuntu.

2.3.2. Parametry środowiska testowego

Poniższych kompresji dokonano na komputerze posiadający procesor czterordzeniowy Intel Core i7-8665U 1.9GHz wyposażony w 16 GB pamięci RAM. Do testów użyto programu 7z uruchomionego na systemie operacyjnym Windows 10 Enterprise. Dla każdego algorytmu i formatu pliku przeprowadzono dwa testy - jeden ze standardową kompresją oraz jeden z największą możliwą w programie. W obydwu przypadkach zastosowano domyślne parametry proponowane przez program 7z, które zostały opisane poniżej.

W pierwszym przypadku ze standardową kompresją dla algorytmu Deflate zastosowano rozmiar słownika równy 32 KB i długość słowa 32, dla Deflate64 zastosowano identyczną długość słowa oraz dwukrotnie zwiększono rozmiar słownika, w przypadku algorytmu BZip2 rozmiar słownika wynosił 900 KB, dla algorytmu LZMA rozmiar słownika wynosił 16 MB i długość słowa 32, a dla PPMd rozmiar słownika był równy 16 MB i długość słowa równa 8.

W drugim przypadku z zwiększoną kompresją dla algorytmu Deflate zastosowano identyczny rozmiar słownika równy 32 KB oraz zwiększono długość słowa do 128, dla Deflate 64 rozmiar słownika wyniósł 64 Kb i długość słowa równa 128, dla BZip2 rozmiar słownika wynosi 900 Kb, dla LZMA rozmiar słownika jest równy 64 MB i długość słowa równa 64, a dla PPMd rozmiar słownika wynosi 128 MB i długość słowa 12.

2.3.3. Algorytmy

Do przeprowadzenia testu będą użyte algorytmy Deflate, Deflate64, BZip2, LZMA oraz PPMd. Poniżej znajduje się krótki opis działania każdego z nich.

Algorytm Deflate można podzielić na trzy kroki. W pierwszym z nich dane wejściowe są dzielone na serie bloków, z których każdy może być niezależnie kompresowany. Następnie wykorzystywana jest modyfikacja metody słownikowej LZ77 służącej do znalezienia powtarzających się ciągów bitów. W ostatnim kroku, aby zwiększyć skuteczność kompresji, zapisuje się wyjście za pomocą kodowania Huffmana. Kody są dobierane na podstawie częstości występowania danych znaków. Główna różnica pomiędzy algorytmem Deflate, a Deflate64 polega na wielkości rozmiaru słownika. Dla Deflate64 została zwiększona z 32 KB na 64 KB. Zwiększa się również rozmiar zmiennych zdefiniowanych wewnątrz algorytmu, dzięki którym możliwa jest skuteczniejsza kompresja.

Algorytm BZip2 posiada więcej kroków od Deflate. W pierwszej kolejności uruchomione jest kodowanie RLE szczegółowo opisane w dalszej części pracy magisterskiej. Następnie dane są przekształcane za pomocą transformaty Burrowsa-Wheelera. Transformata przekształca wejście w taki sposób, aby obok siebie znajdowały się podobne dane. Transformata również została szczegółowo opisana w odrębnym rozdziale w dalszej części pracy. W kolejnym kroku wykonywana jest transformata Move To Front. Polega ona na zamianie symboli na ich indeksy występowań. Dzięki niej może się zmniejszyć entropia. Ponownie dane są modyfikowane za pomocą algorytmu RLE w celu zmniejszenia długości ciągów z zerami. W dalszym kroku dane są kodowane za pomocą kodów Huffmana. Tablice kodów Huffmana są stale ulepszane, aby uzyskać możliwie jak najlepsze wyniki. Na końcu konstruuje się drzewo użytych kodów Huffmana i zapisuje na wyjście.

Algorytm LZMA dzieli się na 3 podstawowe kroki. Pierwszy jest opcjonalny. Polega na zapisaniu każdego bajtu jako różnicy bieżącego oraz poprzedniego bajtu. Deltę bajtów wykonuje się, żeby następny krok mógł działać skuteczniej. Drugi krok bazuje na zoptymalizowanej wersji metody słownikowej LZ77, która umożliwia znalezienie powtarzających się symboli. Na wyjście zapisywana jest sekwencja trójek składających się z przesunięcia w buforze słownikowym, długości dopasowanego ciągu oraz następnego znaku do zakodowania. Algorytm LZ77 został dokładnie opisany w rozdziale „Szczegółowy opis algorytmu LZ77". W ostatnim kroku kodowanie każdego bajtu jest zależne od rozkładu prawdopodobieństwa występowania danego symbolu.

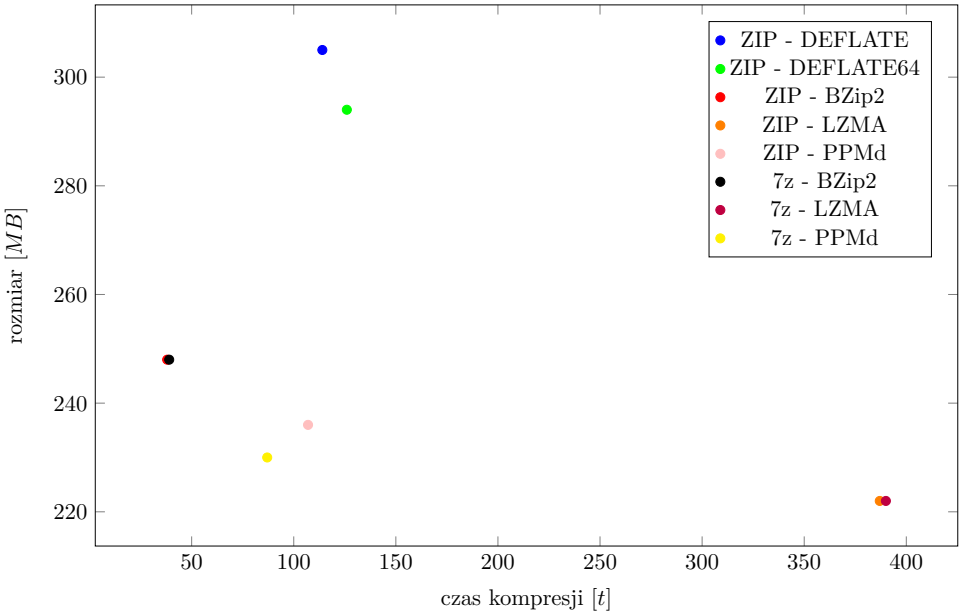
Algorytm PPMd opiera się na modelowaniu statystycznym danych i ich predykcji. W pierwszym kroku dane są strumieniowo czytane oraz na ich podstawie jest tworzony ranking występowania ciągów symboli/bajtów. Najlepszy rezultat kompresji byłby, gdyby użyto całych danych, jednak w przypadku dużego rozmiaru danych byłoby to bardzo czasochłonne i wymagałoby to dużej ilości pamięci RAM. Z tego powodu robi się statystyki dla mniejszych zakresów. Celem jest znalezienie jak najdłuższego dopasowania. W sytuacji gdy w tabeli ze statystykami nie ma danego symbolu, dany symbol nie zostanie zakodowany lub zostanie dodany do tabeli.

2.3.4. Porównanie jakości kompresji dla plików tekstowych

W pierwszej kolejności została sprawdzona kompresja dla plików tekstowych. Zbiór danych zawiera 10 plików tekstowych o łącznym rozmiarze równym 1 GB. Najpierw została spraw-

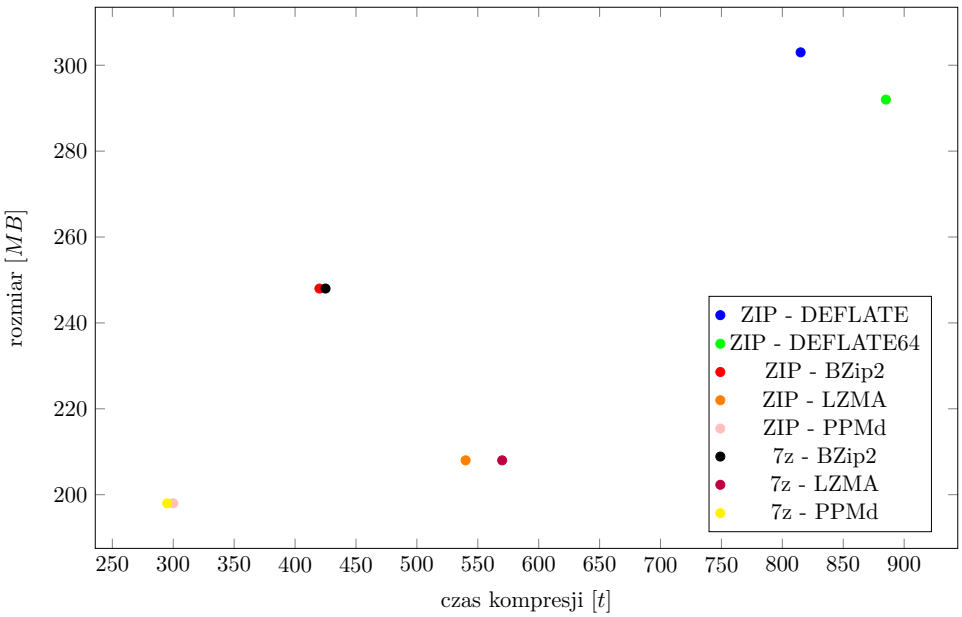
dzona kompresja za pomocą algorytmów wykorzystujących format ZIP, następnie algorytmów wykorzystujących format 7z. Na poniższych wykresach zastosowano legendę z podziałem na dwa wyrazy dla każdego punktu. Po lewej znajduje się rozszerzenie (format) pliku zapisanego na dysk (zip lub 7z), a po prawej algorytm użyty do kompresji danego pliku.

Rys. 2.2. Porównanie jakości kompresji dla plików tekstowych - zwykła kompresja



Na wykresie widać, że samo rozszerzenie pliku nie ma niewielkie znaczenie. Dla algorytmów BZip2 i LZMA wyniki praktycznie się pokrywają, a dla PPMd różnice nie są duże. Dużo ważniejsze od niego jest wykorzystany algorytm do kompresji. Dla tekstu największy stopień kompresji ma algorytm LZMA, a najgorzej poradził sobie algorytm Deflate. Był to spodziewany wynik, gdyż w tekście jest wiele powtarzających się ciągów znaków, które łatwo można zastąpić pozycjami ze słownika.

Rys. 2.3. Porównanie jakości kompresji dla plików tekstowych - najwyższa kompresja



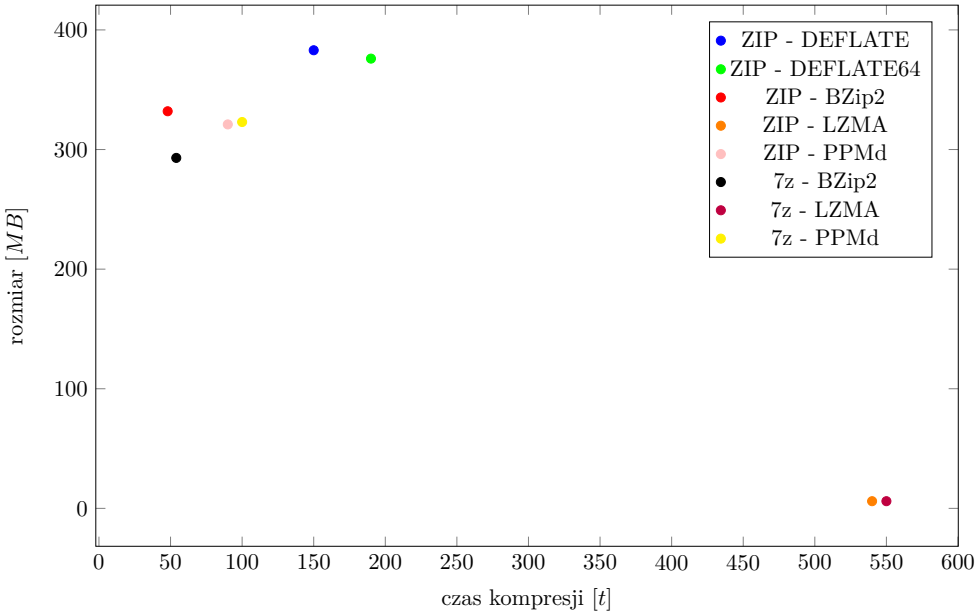
Przeprowadzono także eksperyment, w którym dążono do maksymalnego stopnia kompresji za pomocą algorytmu LZMA. W tym celu dobierano różne rozmiary słownika i długości słowa. Najlepszą kompresje uzyskano dla rozmiaru słownika równą 256 MB i długości słowa 96 znaków. Skompresowany plik miał rozmiar 197 MB, czyli rozmiar stanowił $19,7\%$ pliku początkowego, a czas kompresji wyniósł 11 minut.

Zastosowanie największej kompresji miało pozytywny rezultat dla algorytmów LZMA oraz PPMd. Udało się uzyskać dla PPMd skompresować plik do $20,2\%$, a dla LZMA do $21,1\%$. Jednak jest to wciąż gorszy wynik od manualnych prób opisanych w poprzednim akapicie, gdzie udało się uzyskać kompresję $19,7\%$. Pomimo tego że udało się uzyskać minimalnie lepszą kompresję w stosunku do wyników z poprzedniego wykresu, czas działania wydłużył się kilkukrotnie.

2.3.5. Porównanie jakości kompresji dla plików z liczbami pseudolosowymi

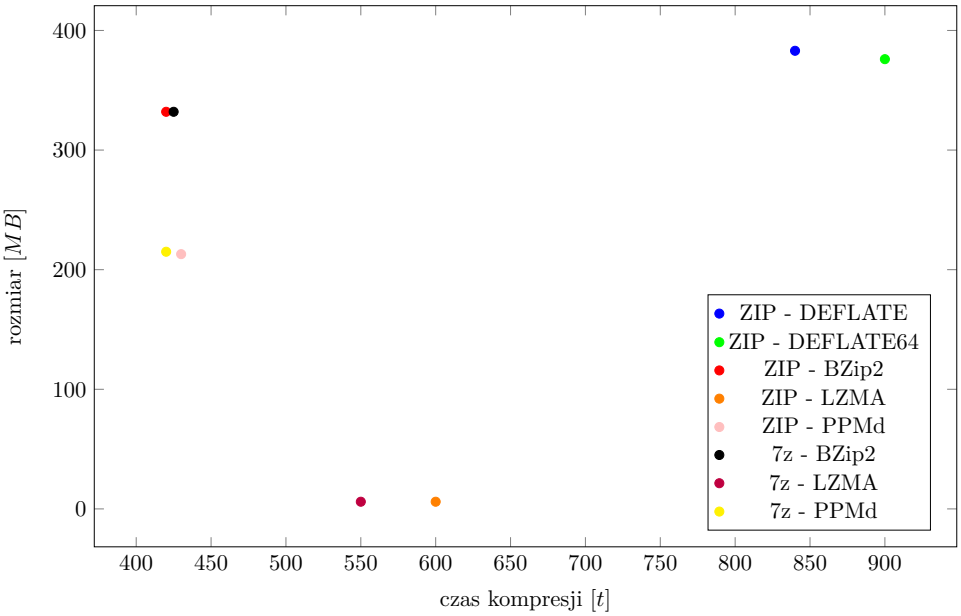
Do przetestowania kompresji liczb pseudolosowych zastosowano plik o rozmiarze 976 MB. Podobnie jak dla plików tekstowych, tutaj również zostanie wykonana kompresja najpierw dla plików ZIP a następnie 7z i będą użyte te same algorytmy.

Rys. 2.4. Porównanie jakości kompresji dla plików z liczbami pseudolosowymi - zwykła kompresja



Bardzo ciekawy jest wynik dla algorytmu LZMA. Rozmiar pliku skompresowanego wyniósł 1% rozmiaru oryginalnego pliku. Tak duży stopień kompresji prawdopodobnie uzyskano dzięki temu, że pomimo tego, że liczby były pseudolosowe, miały bardzo dużo podzbiorów cyfr wspólnych. Takie podobieństwo było dlatego, że do ich wygenerowania wykorzystano funkcję randint z modułu random języka Python. Taki generator tak naprawdę nie generuje liczb losowych, gdyż opiera się na okresowości. Te podzbiory udało się zastąpić krótszymi indeksami ze słownika algorytmu i w ten sposób uzyskano tak dużą kompresję. Pozostałe algorytmy działały zgodnie z założeniem, gdzie uzyskano kompresję dużo gorszą niż w przypadku kompresji artykułów z Wikipedii. Najgorzej poradził sobie algorytm Deflate, gdzie rozmiar skompresowanego pliku wynosił 39% oryginalnego rozmiaru.

Rys. 2.5. Porównanie jakości kompresji dla plików z liczbami pseudolosowymi - najwyższa kompresja

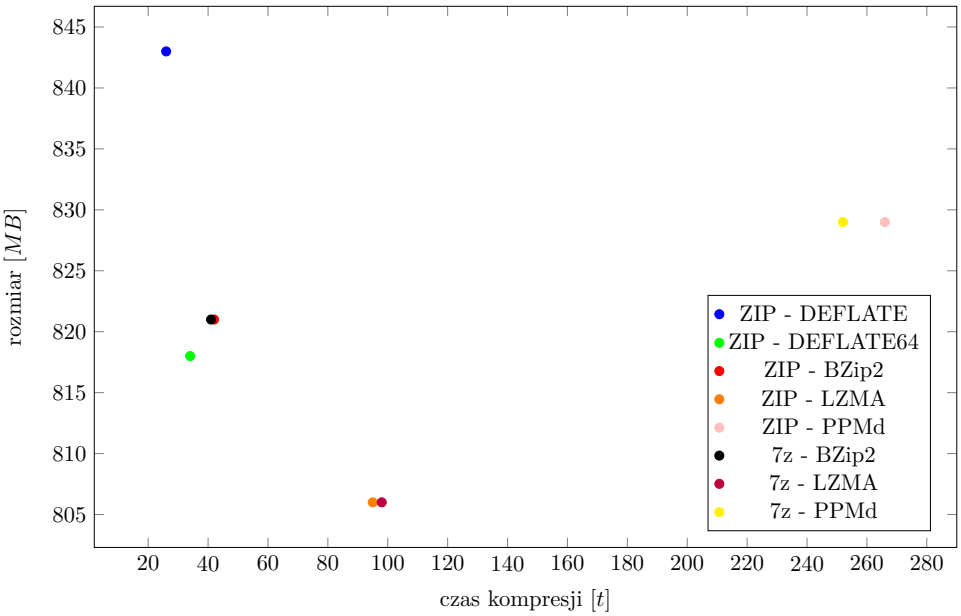


W tym przypadku widać, że ustawienie zwiększające stopień kompresji w programie 7z nie przyniosło dla większości algorytmów poprawy. Jedynie dla algorytmu PPMd uzyskano lepszy stopień kompresji, za to dla każdego pomiaru uzyskano gorsze czasy działania.

2.3.6. Porównanie jakości kompresji dla plików binarnych

Do przetestowania kompresji plików binarnych wykorzystano obraz Ubuntu Server 19.10. Plik ma rozmiar 843 MB. Podobnie jak w poprzednich testach, porównano format zip i 7z oraz algorytmy Deflate, BZip2, LZMA i PPMd.

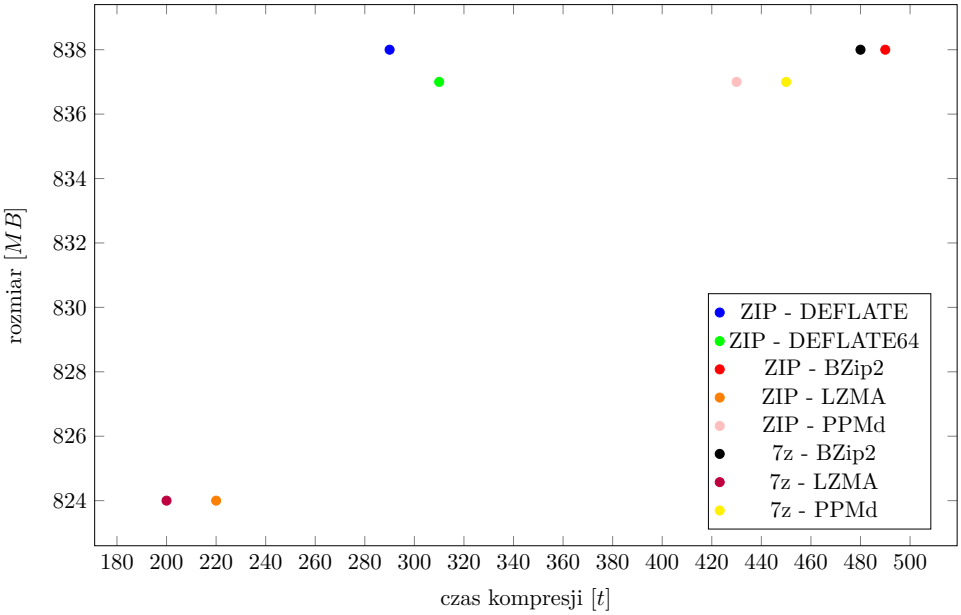
Rys. 2.6. Porównanie jakości kompresji dla plików binarnych - zwykła kompresja



Widać, że zgodnie z hipotezami żaden algorytm nie poradził sobie dobrze z kompresją

danych. Najlepiej skompresowany plik udało się zmniejszyć do poziomu około 95, 6% oryginalnego rozmiaru. W przypadku algorytmu Deflate nie było żadnego zysku, a z kolei w przypadku algorytmu PPMd czasy znacząco odstawały od reszty algorytmów, przy braku zwiększenia kompresji.

Rys. 2.7. Porównanie jakości kompresji dla plików binarnych - najwyższa kompresja

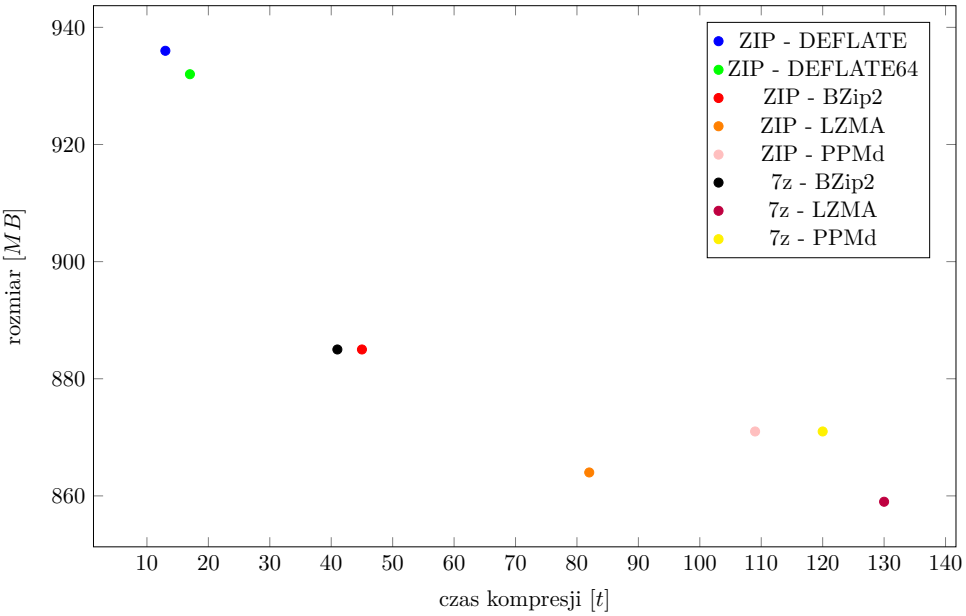


Zwiększenie stopnia kompresji w programie 7z nie przyniosło poprawy jakości kompresji. Jedynie kompresja dla algorytmu Deflate cechowała się większym stopniem kompresji pliku. W przypadku pozostałych nie tylko czas działania był dużo gorszy, ale także stopień kompresji był słabszy w porównaniu do wyników z poprzedniego wykresu.

2.3.7. Porównanie jakości kompresji dla surowych plików graficznych

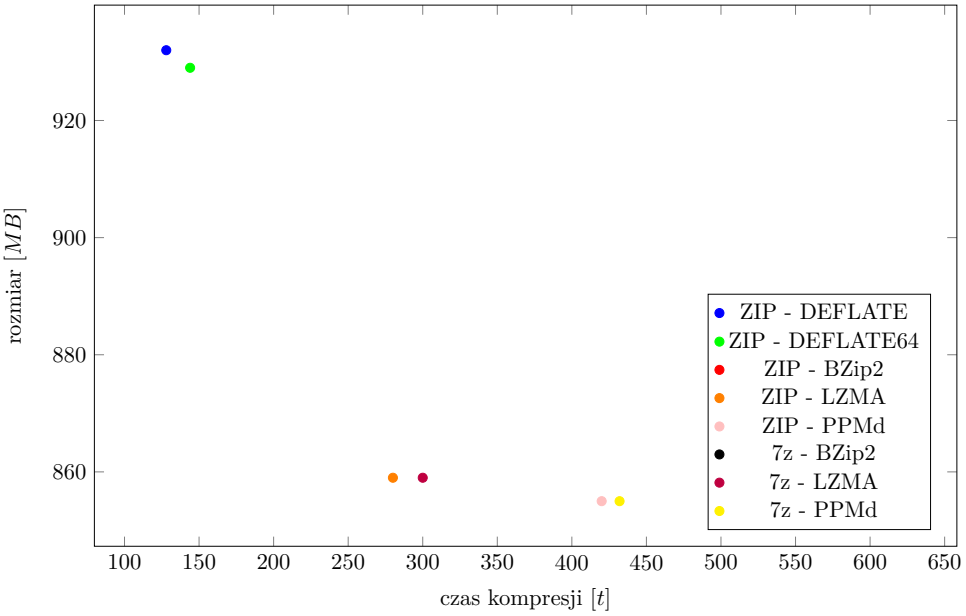
Surowe pliki graficzne stosowane są powszechnie przez profesjonalnych fotografów. Większość lustrzanek cyfrowych może jednocześnie zapisywać zdjęcia jednocześnie w dwóch formatach - bez kompresji oraz skompresowanych za pomocą algorytmów kompresji stratnej (np. JPEG). Te same zdjęcia w formacie JPEG często zajmują około 30% wielkości zdjęć w formacie surowym. Wartość jednak zależy od stopnia kompresji plików skompresowanych. Na poniższych wykresach zostaną porównane te same algorytmy kompresji bezstratnej i formaty plików jak dla poprzednich rodzajów plików. Folder ze zdjęciami posiadał rozmiar 993 MB oraz zawierał 42 elementy.

Rys. 2.8. Porównanie jakości kompresji dla surowych plików graficznych - zwykła kompresja



Zastosowanie algorytmów kompresji bezstratnej na plikach surowych RAW nie dało dużego zmniejszenia objętości plików. W najlepszym przypadku uzyskano rozmiar stanowiący około 86% oryginalnych plików, a w najgorszym około 94%. Właśnie dlatego dużo częściej w fotografii stosuje się algorytmy kompresji stratnej (np. JPEG), które wykorzystują charakterystykę ludzkiego oka i na podstawie tego uzyskują dużą kompresję, przy niewidocznych różnicach dla ludzkiego oka.

Rys. 2.9. Porównanie jakości kompresji dla surowych plików graficznych - najwyższa kompresja



Dla zastosowanych algorytmów zależność wyników pomiędzy zwykłą kompresją, a najwyższą jest podobna do poprzednich rodzajów plików - kilkuprocentowy wzrost stopnia kompresji przy kilkukrotnym wydłużeniu czasu działania.

2.3.8. Podsumowanie porównania kompresji plików

Wykonano porównanie dla czterech różnych rodzajów plików przez pięć różnych algorytmów oraz dla dwóch formatów plików. Na poniższych wykresach przedstawiono zbiorczo średni stopień kompresji oraz średnią szybkość kompresji dla testowanych algorytmów. Stopień kompresji definiuje się za pomocą wzoru:

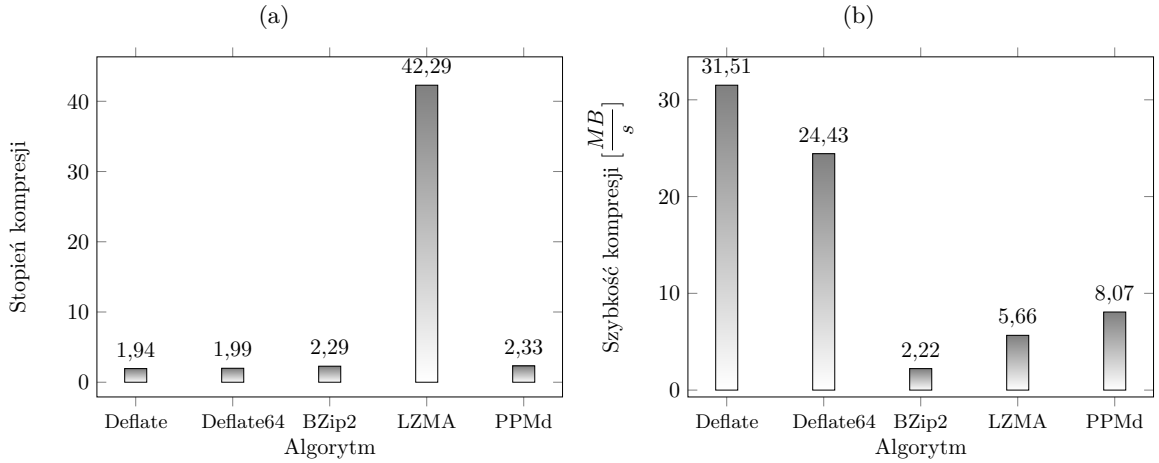
$$\text{StopieńKompresji} = \frac{\text{RozmiarNieskompresowanegoPliku}}{\text{RozmiarSkompresowanegoPliku}}$$

Szybkość kompresji definiuje się jako:

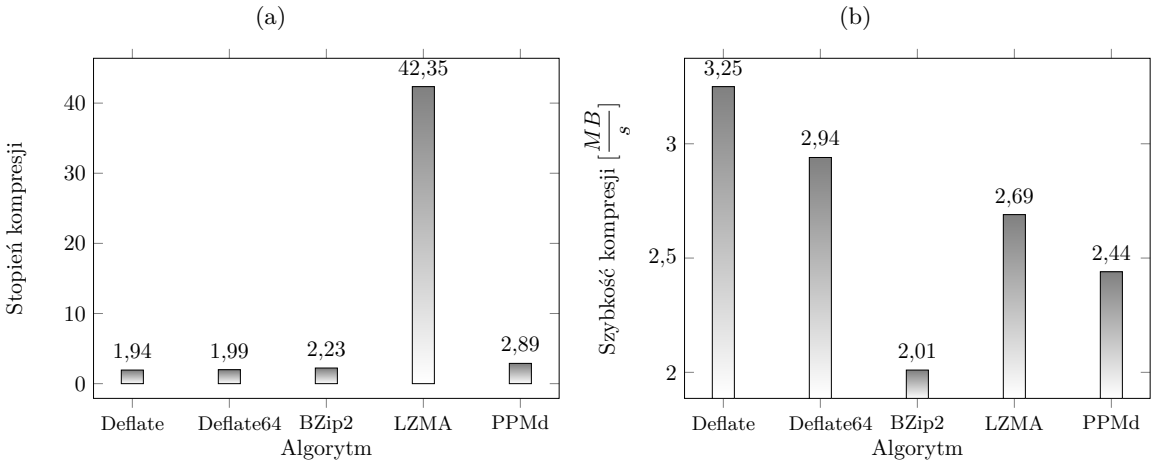
$$\text{SzybkośćKompresji}[\frac{MB}{s}] = \frac{\text{RozmiarNieskompresowanegoPliku}[MB]}{\text{CzasKompresji}[s]}$$

Poniżej przedstawiono wykresy dla dwóch rodzajów kompresji - standardowej oraz najwyższej. Omówione jednak zostaną tylko ze standardową kompresją, gdyż w przypadku podwyższania jakości kompresji w programie 7ZIP, w rzeczywistości stopień kompresji plików wzrastał jedynie o kilka procent, a czas zwiększał się nawet kilkukrotnie.

Rys. 2.10. Porównanie stopnia i szybkości kompresji algorytmów dla wszystkich rodzajów plików - zwykła kompresja



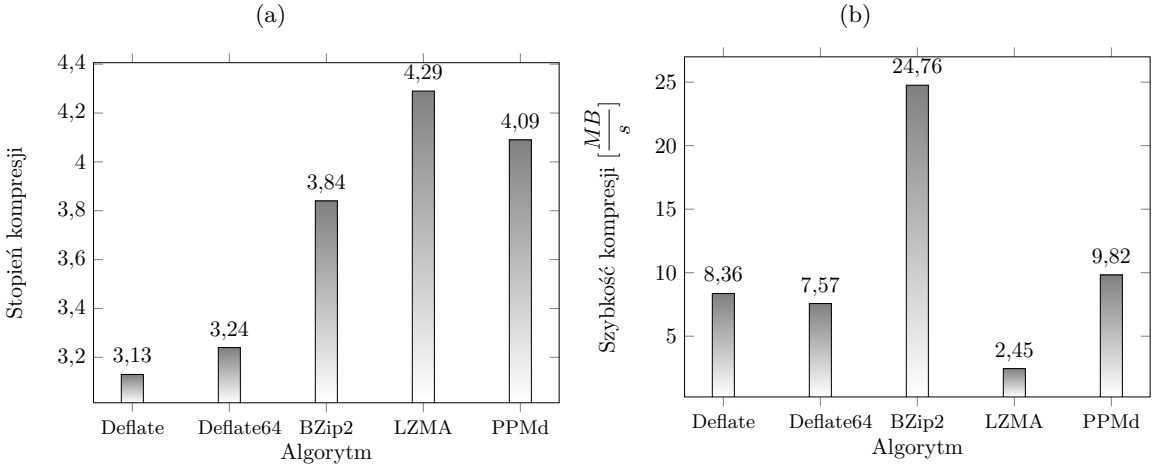
Rys. 2.11. Porównanie stopnia i szybkości kompresji algorytmów dla wszystkich rodzajów plików - najwyższa kompresja



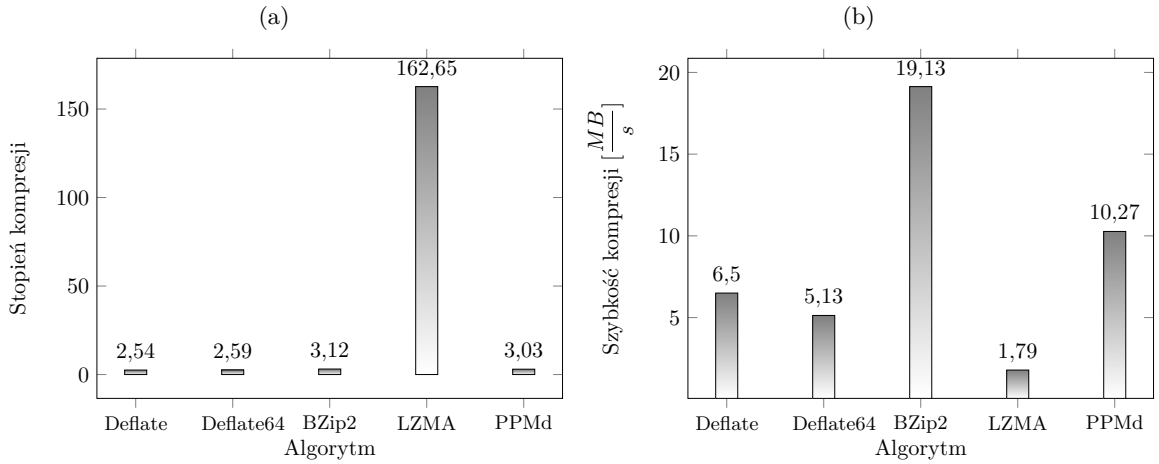
Dla przetestowanych rodzajów plików zdecydowanie najlepszą jakość kompresji miał algorytm LZMA, a największą szybkość kompresji algorytm BZip2. Warto jednak pamiętać, że rzadko kiedy będzie się kompresować pliki w takich samych proporcjach jak w testowanym przypadku. Dlatego właśnie warto również przedstawić wykresy dla każdego algorytmu z osobna.

Na poniższych wykresach widać, że dla każdego rodzaju danych najwyższą jakość miał algorytm LZMA, a często niewiele gorzą miał algorytm PPMd. Z kolei najszybszymi były algorytm BZip2 oraz Deflate. Wniosek z tego jest taki, że gdy nie ma znaczenia czas wykonywania algorytmu, powinien być zastosowany LZMA lub PPMd, gdy jednak ważniejszy jest czas działania od stopnia kompresji, wtedy powinno się zastosować BZip2 lub Deflate.

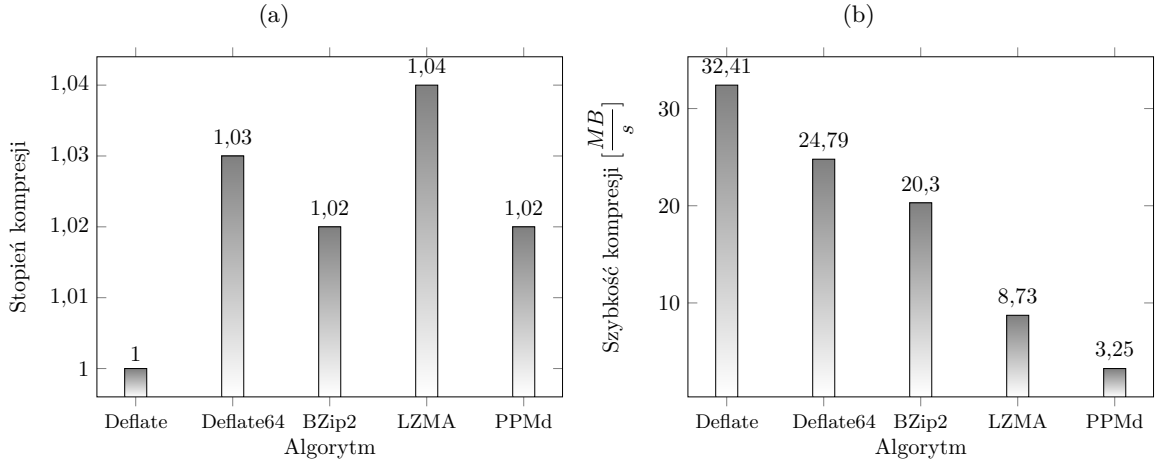
Rys. 2.12. Porównanie stopnia i szybkości kompresji algorytmów dla plików tekstowych - zwykła kompresja



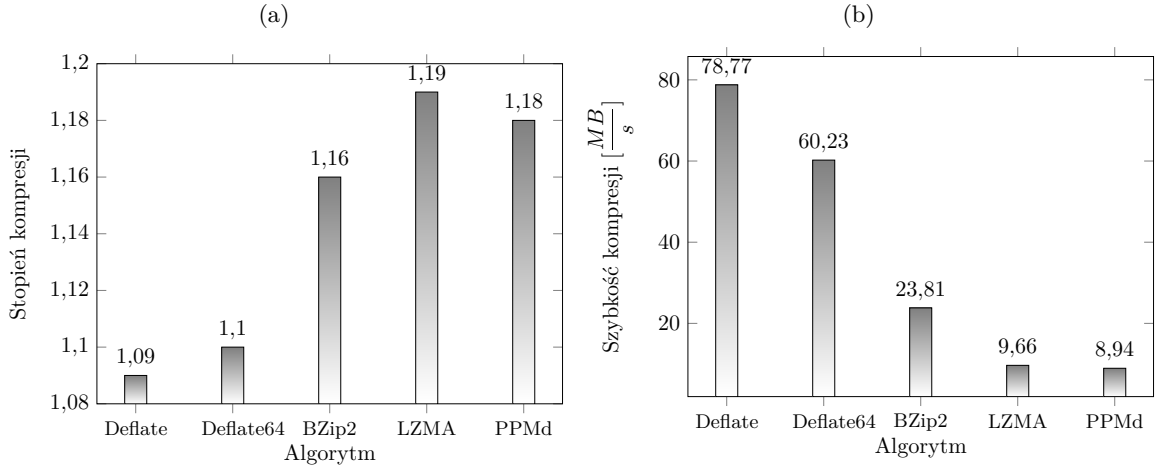
Rys. 2.13. Porównanie stopnia i szybkości kompresji algorytmów dla liczb pseudolosowych - zwykła kompresja



Rys. 2.14. Porównanie stopnia i szybkości kompresji algorytmów dla plików binarnych - zwykła kompresja



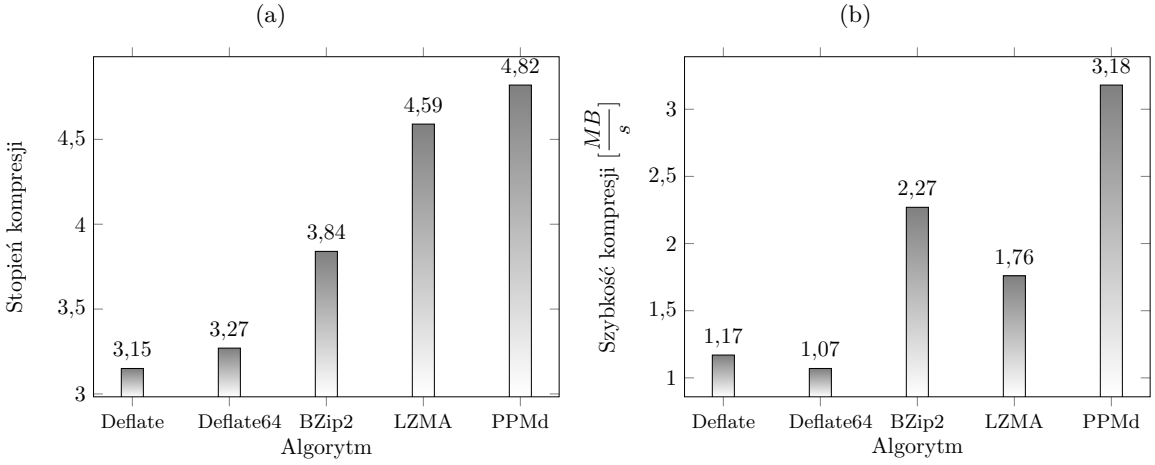
Rys. 2.15. Porównanie stopnia i szybkości kompresji algorytmów dla surowych plików graficznych - zwykła kompresja



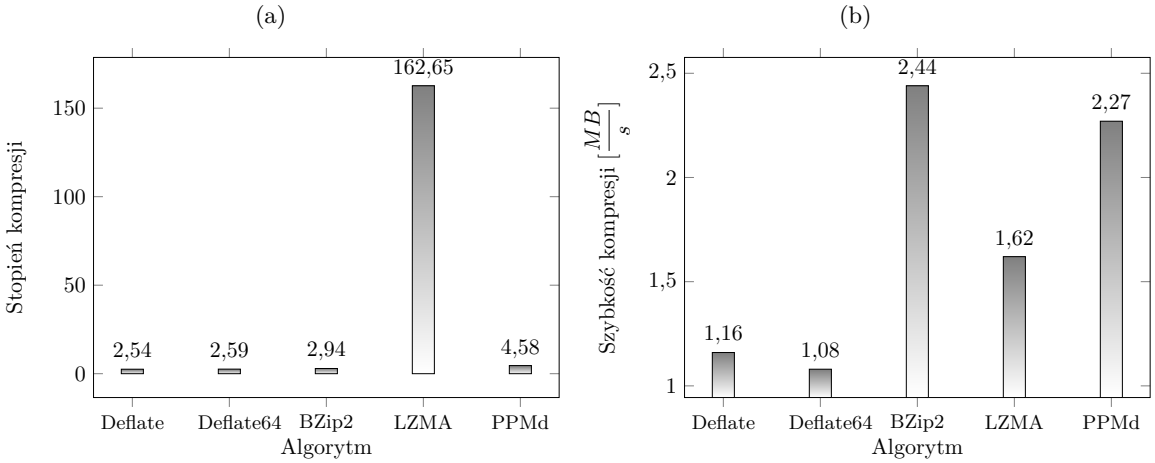
Poniżej znajdują się szczegółowe wykresy stopnia kompresji oraz jej szybkości dla najwyższej kompresji. Łatwo zauważyć, że w rzeczywistości stopień kompresji nie zawsze się zwiększał,

a w niektórych przypadkach nawet się lekko pogorszył.

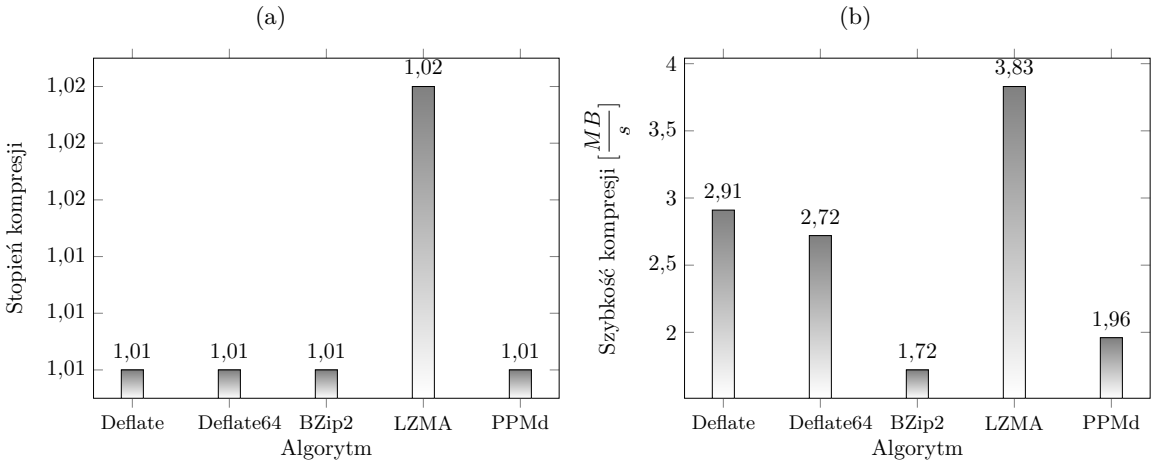
Rys. 2.16. Porównanie stopnia i szybkości kompresji algorytmów dla plików tekstowych - najwyższa kompresja



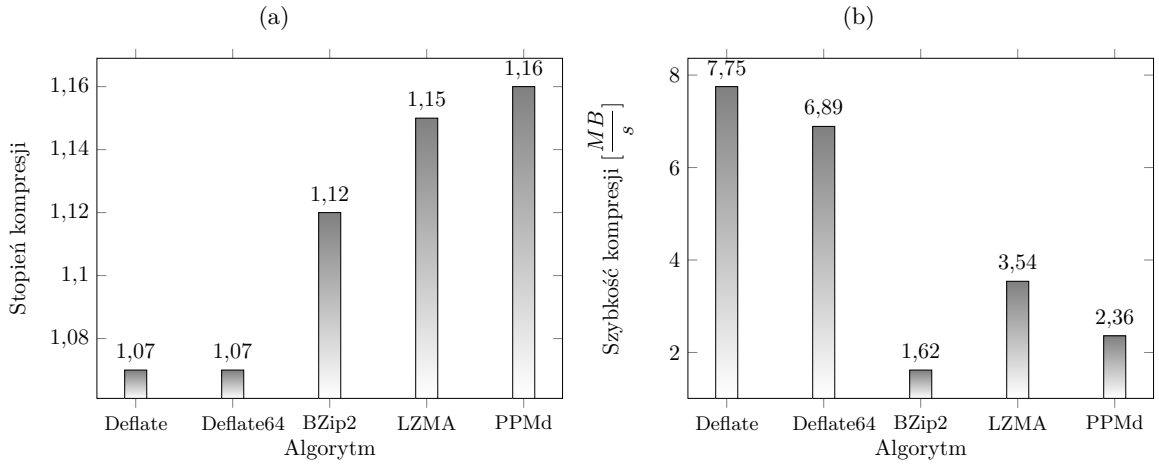
Rys. 2.17. Porównanie stopnia i szybkości kompresji algorytmów dla liczb pseudolosowych - najwyższa kompresja



Rys. 2.18. Porównanie stopnia i szybkości kompresji algorytmów dla plików binarnych - najwyższa kompresja



Rys. 2.19. Porównanie stopnia i szybkości kompresji algorytmów dla surowych plików graficznych - najwyższa kompresja



2.4. Podstawowe techniki kompresji

Istnieje wiele różnych technik kompresji danych. Różnią się między sobą stopniem kompresji oraz skomplikowaniem rozwiązania. Najprostsze wykorzystują kody stałej długości do zakodowania powtarzających się takich samych symboli, inne używają do tego kodów zmiennej długości, jeszcze inne potrafią znajdować powtarzające się ciągi wielu różnych bitów i zastępować je krótszymi sekwencjami. W tym rozdziale zostaną zaprezentowane przykładowe algorytmy wyżej wymienionych technik. Ich zrozumienie znacząco może pomóc w analizie bardziej skomplikowanych algorytmów z następnych rozdziałów. Dla zbioru prostych metod zostanie zaprezentowany algorytm Run-length Encoding, dla metod statystycznych kodowanie Huffmana, a dla metod słownikowych zostanie pokazana ogólna zasada działania podstawowego algorytmu, a szczegółowy opis algorytmów słownikowych będzie w następnych rozdziałach. Zostanie pokazany także przykład transformaty użytecznej między innymi w algorytmie BZip2 - transformaty Burrowsa-Wheelera. Zanim rozpocznie się opisywanie poszczególnych metod, warto wyjaśnić czym charakteryzują się metody statystyczne oraz słownikowe.

Metody statystyczne charakteryzują się zmienną długością kodu w zależności od symbolu. Symbole które częściej występują w danym zbiorze będą miały krótsze kody potrzebne do zakodowania symbolu. W tych metodach algorytmy mogą przyjmować na wejściu częstości występowania danych znaków lub samodzielnie je obliczać. Do najbardziej znanych metod statystycznych zalicza się kodowanie Huffmana, kodowanie Shannona-Fano oraz kodowanie arytmetyczne.

Metody słownikowe mają zupełnie odmienną zasadę działania od metod statystycznych. Po pierwsze nie opierają kompresji na kodach zmiennej długości, a po drugie nie korzystają z danych statystycznych podczas wykonywania algorytmu. Nazwa metody słownikowej wywodzi się z tego, że algorytm zapisuje słowa występujące do struktury danych zwanej słownikiem, a następnie w kompresowanych danych zamienia dane słowo jego pozycją ze słownika. Kompresja polega na tym, że zastępuje się długie ciągi znaków krótkimi adresami ze słownika. Do najbardziej popularnych metod zalicza się LZ77, LZ78, LZSS, LZW.

2.4.1. Run-Length Encoding

Metodę Run-Length Encoding można zaliczyć do najprostszych metod kompresji bezstratnej. Opiera się na zastępowaniu ciągu tych samych symboli ich liczbą występowania. Najłatwiej pokazać to na przykładzie. Załóżmy, że mamy do skompresowania poniższy ciąg znaków:

$$AAAAABBBBBBBCDDEEEEE$$

Jednym ze sposobów zapisywania wynikowych danych dla metody RLE jest dodanie liczby występowania symbolu bezpośrednio przed nim. W tym przypadku wynikowy ciąg będzie wyglądał następująco:

$$5A7B1C2D5E$$

W tym przypadku udało się uzyskać dwukrotną kompresję. Metoda RLE pozwala na wiele różnych zapisów. Przykładowo liczby symboli mogą być przed lub za nim. Powyższy przykład da się również zoptymalizować poprzez niezapisywanie liczby występowania symbolu, gdy występuje on jedynie raz. W takim przypadku wynik będzie następujący:

$$5A7BC2D5E$$

W tym przypadku uzyskano ciąg o 1 znak krótszy.

W przypadku tekstu, wykorzystanie metody RLE nie ma większego sensu. Bardzo rzadko występują sekwencje znaków składających się z ponad dwóch identycznych symboli. Dużo bardziej uzasadnione zastosowanie jest podczas kompresji grafiki, obrazków, zdjęć. Łatwo sobie wyobrazić cały rząd pikseli w jednym kolorze. W tym przypadku program będzie przeszukiwał dany plik rząd po rzędzie i zliczał występowanie tych samych pikseli. Metoda jest stosowana w przypadku formatów obrazu, takich jak BMP, PCX, czy TGA.

2.4.2. Kodowanie Huffmana

Kodowanie Huffmana jest popularną oraz jedną z prostszych w implementacji metodą kompresji danych. Sama w sobie nie daje najlepszych rezultatów i z tego względu jest najczęściej wykorzystywana jako etap innych algorytmów zarówno należących do grupy algorytmów kompresji bezstratnej, jak i stratnej. Celem kodowania Huffmana jest zapisanie symboli za pomocą mniejszej liczby bitów.

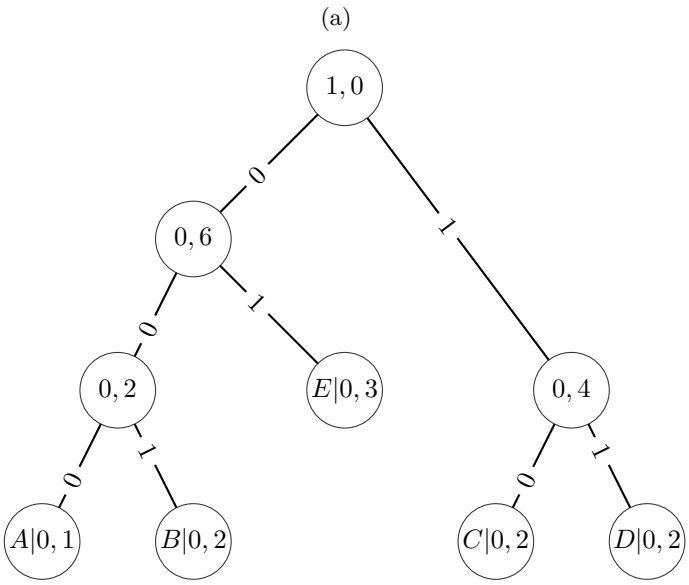
Proces kodowania rozpoczyna się od określenia częstości występowania każdego symbolu dla danego zbioru danych wejściowych. Symbole należy ułożyć na podstawie częstości występowania w kolejności rosnącej. Kolejnym etapem jest zbudowanie drzewa binarnego. Początkowo należy utworzyć listę węzłów niepołączonych ze sobą, gdzie w każdym węźle znajduje się prawdopodobieństwo oraz symbol. Następnie należy znaleźć dwa symbole o najmniejszym prawdopodobieństwie i je połączyć. Na liście elementów powinien znajdować się węzeł zawierający sumę częstości występowania dwóch poprzednich węzłów, zamiast dwóch poprzednich elementów. W następnych

krokach należy powtarzać poprzednie kroki, aż do pozostania na liście tylko jednego elementu. Gdy już pozostanie jeden element, będzie to oznaczało, że budowa drzewa została zakończona. Prawdopodobieństwo w korzeniu powinno mieć wartość równą 1.

Gdy drzewo jest już zbudowane, można określić wartości kodu dla każdego symbolu. W tym celu każdej lewej krawędzi należy przypisać wartość 0, a dla prawej wartość 1. W niektórych algorytmach należy odwrotnie zapisać wartości dla lewych i prawych krawędzi. Aby określić kod danego symbolu, należy przejść najkrótszą drogę od korzenia do węzła i przepisać kolejne przypisane w poprzednim kroku wartości krawędzi.

Najłatwiej pokazać kodowanie na przykładzie. Załóżmy, że chcemy zakodować zbiór $S = \{A, B, C, D, E\}$ o prawdopodobieństwach $0,1; 0,2; 0,2; 0,2; 0,3$. W pierwszym kroku zostaną połączone symbole A i B. Będzie to skutkowało uzyskaniem następujących elementów: $(A + B) = 0,3; C = 0,2; D = 0,2; E = 0,3$. W kolejnym kroku należy połączyć symbole C i D, co da wynik: $(A + B) = 0,3; (C + D) = 0,4; E = 0,3$. Analogicznie jak poprzednio należy wybrać najmniejsze elementy. Wynik połączenia będzie równy: $((A+B)+E) = 0,6; (C+D) = 0,4$. Po kolejnej iteracji drzewo będzie zbudowane: $(((A + B) + E) + (C + D)) = 1$. Poniższy przykład ilustruje końcowy efekt zbudowanego drzewa.

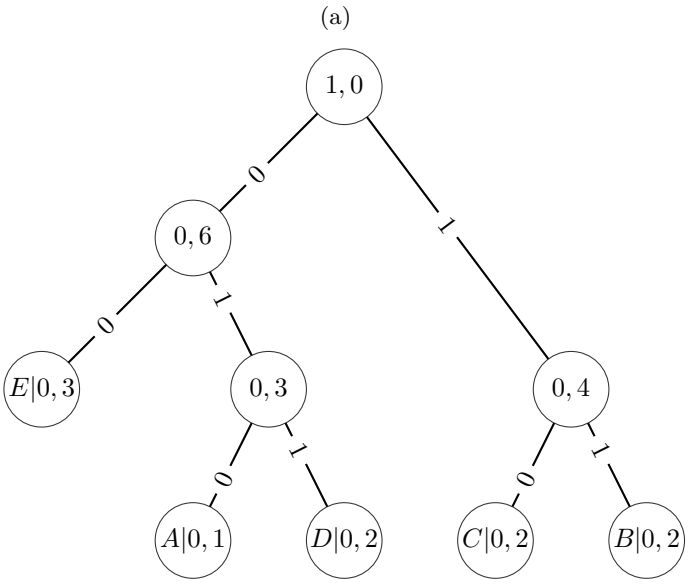
Rys. 2.20. Drzewo Huffmana



Z powyższego grafu można łatwo odczytać kody Huffmana dla zbioru symboli A, B, C, D, E. Będą to odpowiednio $000, 001, 10, 11, 01$.

Kodowanie Huffmana nie definiuje w jaki sposób mają być wybierane węzły, gdy prawdopodobieństwa dwóch symboli są identyczne. Dodatkowo nie jest określone, który z symboli ma być z lewej, a który z prawej strony. Z tego względu algorytmy, które wykorzystują kodowanie Huffmana muszą to same zdefiniować.

Rys. 2.21. Zmodyfikowane drzewo Huffmana



Na podstawie powyższego grafu łatwo można dostrzec, że zmieniły się wartości kodów Huffmana dla zbioru A, B, C, D, E. W nowym przykładzie będą to $010, 11, 10, 011, 00$.

2.4.3. Metoda słownikowa

Celem metody słownikowej jest eliminacja powtarzających się fragmentów bitów lub symboli w danych. Dokonuje się tego poprzez umieszczenie wskaźnika do słownika zamiast jego ciągu znaków. Najłatwiej przedstawić to na poniższym przykładzie.

$$ABCDEABCABCDEFGABCD$$

Pierwszym powtarzającym się podciągiem jest ABC znajdujące się na 5 pozycji ciągu. W poniższym przykładzie $Delta(x, y)$ będzie oznaczać, że należy się cofnąć o x miejsc do tyłu w ciągu, a następnie skopiować y znaków od danego miejsca.

$$ABCDE\Delta(5,3)ABCDEFGABCD$$

Kolejnymi powtarzającymi ciągami są $ABCDE$ oraz $ABCD$. Zapisuje się je w nastepujący sposób.

$$ABCDE\Delta(5,3)\Delta(8,5)FG\Delta(7,4)$$

2.4.4. Transformata Burrowsa-Wheelera

W algorytmach kompresji bezstratnej stosuje się różnego rodzaju transformaty. Mają one na celu przekształcić dane w taki sposób, aby algorytmy kompresji mogły skuteczniej je skompresować. Jedną z nich jest Transformata Burrowsa-Wheelera. Została zaprojektowana Michaela Burrowsa i Davida Wheelera w 1994 roku, a zastosowanie ma między innymi w algorytmie BZip2.

W algorytmie transformaty możemy wyodrębnić trzy główne kroki. W pierwszym generujemy wszystkie rotacje danych, które chcemy przekształcić. Przykładowo dla danych „Praca

magisterska”będziemy mieli poniższy zbiór.

Tabela 2.1. Rotacje dla transformaty Burrowsa-Wheelera

P	r	a	c	a		m	a	g	i	s	t	e	r	s	k	a
r	a	c	a		m	a	g	i	s	t	e	r	s	k	a	P
a	c	a		m	a	g	i	s	t	e	r	s	k	a	P	r
c	a		m	a	g	i	s	t	e	r	s	k	a	P	r	a
a		m	a	g	i	s	t	e	r	s	k	a	P	r	a	c
	m	a	g	i	s	t	e	r	s	k	a	P	r	a	c	a
m	a	g	i	s	t	e	r	s	k	a	P	r	a	c	a	
a	g	i	s	t	e	r	s	k	a	P	r	a	c	a		m
g	i	s	t	e	r	s	k	a	P	r	a	c	a		m	a
i	s	t	e	r	s	k	a	P	r	a	c	a		m	a	g
s	t	e	r	s	k	a	P	r	a	c	a		m	a	g	i
t	e	r	s	k	a	P	r	a	c	a		m	a	g	i	s
e	r	s	k	a	P	r	a	c	a		m	a	g	i	s	t
r	s	k	a	P	r	a	c	a		m	a	g	i	s	t	e
s	k	a	P	r	a	c	a		m	a	g	i	s	t	e	r
k	a	P	r	a	c	a		m	a	g	i	s	t	e	r	s
a	P	r	a	c	a		m	a	g	i	s	t	e	r	s	k

W następnym kroku należy posortować rotacje w kolejności leksykograficznej. Powinno to dać następujący rezultat:

Tabela 2.2. Rotacje w kolejności leksykograficznej dla transformaty Burrowsa-Wheelera

	m	a	g	i	s	t	e	r	s	k	a	P	r	a	c	a
P	r	a	c	a		m	a	g	i	s	t	e	r	s	k	a
a		m	a	g	i	s	t	e	r	s	k	a	P	r	a	c
a	P	r	a	c	a		m	a	g	i	s	t	e	r	s	k
a	c	a		m	a	g	i	s	t	e	r	s	k	a	P	r
a	g	i	s	t	e	r	s	k	a	P	r	a	c	a		m
c	a		m	a	g	i	s	t	e	r	s	k	a	P	r	a
e	r	s	k	a	P	r	a	c	a		m	a	g	i	s	t
g	i	s	t	e	r	s	k	a	P	r	a	c	a		m	a
i	s	t	e	r	s	k	a	P	r	a	c	a		m	a	g
k	a	P	r	a	c	a		m	a	g	i	s	t	e	r	s
m	a	g	i	s	t	e	r	s	k	a	P	r	a	c	a	
r	a	c	a		m	a	g	i	s	t	e	r	s	k	a	P
r	s	k	a	P	r	a	c	a		m	a	g	i	s	t	e
s	k	a	P	r	a	c	a		m	a	g	i	s	t	e	r
s	t	e	r	s	k	a	P	r	a	c	a		m	a	g	i
t	e	r	s	k	a	P	r	a	c	a		m	a	g	i	s

W ostatnim kroku należy zachować ostatnią kolumnę każdej rotacji a także pozycję pod którą znajdują się oryginalne dane. W tym przypadku ciąg znaków z końcowej kolumny jest równy $aackrmatagsPeris$, a indeks jest równy 1 (w skali rozpoczynającej się od 0).

Do testów algorytmu może być przydatny kod napisany w języku Python przedstawiony poniżej.

Listing 2.3. Kod Transformaty Burrowsa Wheelera w języku Python

```
import typing

def burrow_wheeler_transform(text: str) > typing.Tuple[str, int]:
    all_rotations = sorted(get_all_rotations(text))
    index = find_original_sequence(text, all_rotations)
    last_letters_sequence = get_last_letters_sequence(all_rotations)
    return last_letters_sequence, index

def get_all_rotations(text: str) > typing.List[str]:
    text_size = len(text)
    return [text[i:text_size]+text[0:i] for i in range(text_size)]

def find_original_sequence(text: str, rotations: typing.List[str]) > int:
    return rotations.index(text)

def get_last_letters_sequence(rotations: typing.List[str]) > str:
    return ''.join(element[ 1] for element in rotations)

burrow_wheeler_transform('Praca magisterska')
```

Algorytm transformaty odwrotnej jest nieco bardziej skomplikowany. W pierwszej kolejności konieczne jest zauważenie, że z ostatniej kolumny można w łatwy sposób uzyskać pierwszą. Jedynie co trzeba zrobić, to posortować ją w kolejności leksykograficznej. Wynik będzie następujący $\_Paaaacegikmrrsst$. Podczas sortowania należy także zapamiętać pozycję bajtu w ostatniej kolumnie. Dla przekształconego wcześniej tekstu będą następujące wartości: „ ”$\rightarrow 11$, $P \rightarrow 12$, $a \rightarrow 0$, $a \rightarrow 1$, $a \rightarrow 6$, $a \rightarrow 8$, $c \rightarrow 2$, $e \rightarrow 13$, $g \rightarrow 9$, $i \rightarrow 15$, $k \rightarrow 3$, $m \rightarrow 5$, $r \rightarrow 4$, $r \rightarrow 14$, $s \rightarrow 10$, $s \rightarrow 16$, $t \rightarrow 7$. W następnej kolejności należy posortować otrzymane dane w taki sposób, aby na zapamiętanej pozycji bajtu z ostatniej kolumny znajdował się indeks z otrzymanych wyżej wartości. Czyli na zerowej pozycji znajdowała się wartość 11. W nowej tablicy zapisujemy na jedenastej pozycji liczbę 0. Robiąc analogicznie dla pozostałych przypadków powinno się otrzymać $T = [2, 3, 6, 10, 12, 11, 4, 16, 5, 8, 14, 0, 1, 7, 13, 9, 15]$. W kolejnym kroku należy wyznaczyć kolejność znaków. Dla każdego elementu i otrzymujemy następny indeks według wzoru $K = T[K[i]]$, gdzie K to kolejne indeksy ciągu znaków, który chcemy odkodować. $K[0]$ jest równy indeksowi zapisanego podczas kodowania, czyli pozycji pod którą znajdują się oryginalne dane. Dla przedstawionego przypadku wartość listy K będzie równa $[1, 3, 10, 14, 13, 7, 16, 15, 9, 8, 5, 11, 0, 2, 6, 4, 12]$. W ostatnim kroku należy posortować K w kolejności odwrotnej i odczytać znaki z przekształconego wcześniej tekstu według kolejności z K. Finalnie otrzymujemy ciąg początkowy „Praca magisterska”.

Przydatny może być kod w języku Python do analizy zagadnienia.

Listing 2.4. Kod odwrotnej Transformaty Burrowsa Wheelera w języku Python

```
import typing

def burrow_wheeler_restore(index, last_letters_sequence):
    text_size = len(last_letters_sequence)
    first_column = find_first_column(last_letters_sequence)
    characters_order = get_characters_order(first_column, index, text_size)
    return ''.join([last_letters_sequence[i] for i in characters_order])

def get_characters_order(first_column, index, text_size):
    transitions = [ 1] * text_size
    for i, last_column_index in enumerate(element[0] for element in first_column):
        transitions[last_column_index] = i
    characters_order = [index]
    for i in range(0, text_size   1):
        characters_order.append(transitions[characters_order[i]])
    characters_order.reverse()
    return characters_order

def find_first_column(last_letters_sequence: str)  > typing.List[typing.Tuple[int,
    str]]:
    first_row = []
    for i in range(0, len(last_letters_sequence)):
        first_row.append((i, last_letters_sequence[i]))
    return sorted(first_row, key=lambda x: x[1])

burrow_wheeler_restore(1, 'aackrmatags Peris')
```

Algorytm jest szczególnie przydatny, gdy kompresowany ciąg danych zawiera wiele powtarzających się podciągów. W takiej sytuacji na wyjściu będzie wiele miejsc, w których te same znaki występują wielokrotnie obok siebie. Przykładowo bardzo dobrze sprawdzi się podczas przetwarzania danych z dziedziny genetyki, w której genomy zapisane w postaci alfabetu składającego się z 4 znaków A, C, T, G, będą miały wiele powtórzeń. Korzystne ułożenie danych wynika z tego, że rotacje są posortowane w sposób leksykograficzny. Powoduje to częste występowanie tych samych znaków obok siebie. W transformacie bierze się pod uwagę ostatnią kolumnę podczas zapisywania wyjścia. Wynika to z tego, że tylko z ostatniej kolumny da się odzyskać początkowe dane. Pozostałe kolumny nie mają tej cechy.

2.5. Szczegółowy opis algorytmu LZ77

Algorytm LZ77 jest jednym z podstawowych algorytmów kompresji bezstratnej opartym na metodzie słownikowej. Został on zaprezentowany przez Jacoba Ziva i Abrahama Lempela w 1977 roku. Algorytm jest zarówno używany niezależnie w niektórych formatach plików kompresji takich jak PNG, PDF, TIFF, jak i w połączeniu z innymi algorytmami (np. w przypadku algorytmu Deflate).

Główne założenie algorytmu opiera się na zależności, że często w tekście powtarzają się wyrazy lub ciągi liter. Algorytm sprawdza, czy dany ciąg już wystąpił w danym tekście. Jeżeli tak, to kopiuje wskaźnik do poprzedniego wystąpienia zamiast przepisywać dany ciąg znaków. Metoda korzysta z dwóch buforów do kompresji danych. Pierwszy to słownikowy, w którym znajduje się s ostatnio zakodowanych symboli. Znajduje się on z lewej strony. Drugi, znajdujący się z prawej strony, to bufor wejściowy, zawierający w symboli do zakodowania. Zawiera on dane, które mają być zakodowane. Często w przypadku tego algorytmu buforem nazywa się „przesuwające okno". Wynika to z tego, że algorytm przesuwa obszar przeszukiwania z każdym kolejnym zakodowanym znakiem. W algorytmie zapisuje się sekwencje w postaci krotek składających się z trzech elementów $< p, d, z >$. Pierwszy element p oznacza przesunięcie w buforze słownikowym, drugi element d oznacza długość dopasowanego ciągu znaków, a trzeci z zawiera następny znak do zakodowania.

2.5.1. Zasada działania algorytmu

Kolejne kroki algorytmu wyglądają następująco. W pierwszej kolejności należy określić rozmiar buforów słownikowego S oraz wejściowego W. W celu uzyskania najlepszych rezultatów kompresji, rozmiary powinny być całkowitymi potęgami dwójki. Następnym krokiem będzie wpisanie do bufora wejściowego pierwszego symbolu. Bufor słownikowy należy wypełnić kolejnymi znakami kodowanego tekstu. Następną część algorytmu wykonuje się w pętli iterując po kolei po wszystkich elementach buforu wejściowego. Należy dopasować początkowe symbole bufora wejściowego do podciągu bufora słownikowego. Im dopasowana sekwencja będzie dłuższa, tym stopień kompresji będzie większy. Przy dopasowaniu istotne jest to, że część znalezionej sekwencji z bufora słownikowego może być wspólna z buforem wejściowym. Na wyjście należy zapisać trójkę $< p, d, z >$ opisaną wcześniej w poprzednim akapicie. Następnie należy przesunąć pozycje kursora o $z + 1$ pozycji w prawo oraz powrócić do początku pętli.

2.5.2. Przykład działania algorytmu

Najłatwiej pokazać zasadę działania algorytmu na przykładzie. Załóżmy, że chcemy skompresować poniższy ciąg znaków:

ababcabcabcd

Przed rozpoczęciem wykonywania kroków kompresji należy ustalić rozmiary buforów. W przykładzie zostanie porównana opcja dla buforów o rozmiarze 4 elementów i 8 elementów. Na samym początku bufor słownikowy nie zawiera żadnych elementów, stąd nie będzie dopasowania ciągów znaków z buforu wejściowego. Pierwszym znakiem od lewej strony będzie znak "a". Kolejnym krokiem algorytmu będzie zapisanie krotki opisanej w poprzednich akapitach. Pierwszym

elementem jest p (przesunięcie). W tym przypadku nie było przesunięcia w celu znalezienia dopasowanego ciągu, stąd wartość będzie równa 0. Drugim elementem jest d (długość dopasowanego podciągu). Tutaj także wartość będzie równa 0, gdyż nie znaleziono dopasowania. Trzecim elementem jest z (znak znajdujący się po dopasowaniu). Tym znakiem będzie pierwsza litera w ciągu „a". Przesuwamy pozycję kursora o $d + 1$ elementów, czyli w tym przypadku o jedną pozycję. Otrzymamy teraz:

$a[b]abcabcabcd$

Wyjście: $(0, 0, a)$

Aktualnie kursor znajduje się na drugiej pozycji i wskazuje znak „b". Przebieg dopasowania będzie wyglądał podobnie do poprzedniego kroku. W tym przypadku nie ma możliwości znalezienia danego podciągu w buforze słownikowym. Na wyjście zostanie zapisane $(0, 0, b)$ oraz nastąpi przesunięcie o jedną pozycję. Aktualny stan będzie wyglądał następująco.

$ab[a]bcabcabcd$

Wyjście: $(0, 0, a)$, $(0, 0, b)$

Kursor wskazuje znak „a". Tym razem możliwe jest znalezienie dopasowania podciągu „ab". W tym celu należy cofnąć się o 2 pozycje w lewo oraz skopiować 2 znaki. Krotka będzie miała wartość $(2, 2, c)$. Po tym kroku kursor będzie na szóstej pozycji.

$ababc[a]bcabcd$

Wyjście: $(0, 0, a)$, $(0, 0, b)$, $(2, 2, c)$

Następny krok będzie się różnił w zależności od rozmiaru bufora. W przypadku, gdy rozmiar zostanie ustawiony na 4 elementy, będzie można dopasować jedynie podciąg „abc". Wtedy krotka będzie miała postać $(3, 4, b)$.

$ababcabcab[c]d$

Wyjście: $(0, 0, a)$, $(0, 0, b)$, $(2, 2, c)$

Jednak gdy rozmiar zostanie zwiększony do 8 elementów, będzie można dopasować także powtarzający się podciąg z bufora wejściowego. Wtedy będzie można dopasować podciąg składający się z 6 znaków „abcabc". Utworzona krotka będzie miała następującą postać: $(3, 6, d)$. Finalnie algorytm skompresował tekst $ababcabcabcd$ do $(0, 0, a)$, $(0, 0, b)$, $(2, 2, c)$, $(3, 6, d)$

2.6. Szczegółowy opis algorytmu Deflate

Algorytm Deflate jest jednym z podstawowych algorytmów kompresji bezstratnej opartym na darmowej licencji, charakteryzujący się połączeniem wariantu metody słownikowej LZ77 z kodowaniem Huffmana. Został on zaprojektowany przez Phila Katza, a następnie zestandaryzowany w RFC 1951. Jest domyślnym algorytmem plików zip w przypadku wielu popularnych programów, w tym np. wbudowany program do kompresji w systemie Windows 10, czy linuxowy zip.

Założeniem algorytmu było to, aby dało się go skompresować i zdekompresować na każdym sprzęcie niezależnie od procesora, liczby dostępnych wątków, czy systemu operacyjnego. W tym celu ustalono kodowanie dla każdej części skompresowanych danych, wprowadzono bloki ze zdefiniowanym maksymalnym rozmiarem.

Domyślnym kodowaniem dla danych będzie „little endian”. Oznacza to, że najpierw będą bity najmniej znaczące. Takie kodowanie jednak nie umożliwia odczytania kodów Huffmana o nieznanej długości. W tym celu wprowadzono dla kodów Huffmana kodowanie „big endian”. W tym kodowaniu bity ułożone są od najbardziej znaczących. Oznacza to, że bity są w kolejności 76543210. Poniżej znajduje się przykład w celu łatwiejszego zobrazowania kodowania little endian. Załóżmy, że chcemy zapisać liczbę 600 w danym kodowaniu.

Tabela 2.3. Kolejność bitów w algorytmie Deflate

0 bajt	1 bajt
0101 1000	0000 0010

Dane zostały podzielone na bloki o różnej długości. Każdy blok zawiera informację o jego rozmiarze, przy czym ten rozmiar nie może przekroczyć 65535 bajtów. Poza rozmiarem każdy blok może mieć różny typ kompresji. Wyszczególnia się następujące 3 typy:

- Brak kompresji - Tryb jest przydatny wtedy, gdy dane są trudne lub niemożliwe do skompresowania. Takim przykładem mogą być dane losowe. Innym przypadkiem, gdy nie chcemy kompresji może być chęć szybkiego scalenia wielu plików w jeden w celu np. szybszego i łatwiejszego późniejszego przesłania tego pliku.
- Kompresja ze statycznymi kodami Huffmana - algorytm zawiera dwie wbudowane tablice, przy pomocy których algorytm powinien dać w miarę dobre rezultaty przy kompresji. W tym przypadku oszczędzamy czas wykonywania programu kosztem stopnia kompresji wynikowego pliku. Pierwsza tablica służy do kodowania znaków, druga tabela odpowiada za kodowanie odległości w kodowaniu słownikowym.
- Kompresja z dynamicznymi kodami Huffmana - w tym trybie dane wejściowe są przetwarzane i na ich podstawie generowana jest tabela odpowiadająca za kodowanie znaków i odległości. W tym podejściu kolejne bloki mogą wykorzystywać informacje o występowaniu znaków, częstotliwości ich powtarzania, powtarzających się ciągów znaków itp. z wcześniej przetworzonych bloków.

2.6.1. Sposób kompresji algorytmu

Każdy blok jest kompresowany oddzielnie. Na początku każdego znajduje się informacja o tym czy to ostatni blok danego pliku oraz o metodzie kompresji. Informacja o ostatnim bloku zawiera jeden bit. 1 oznacza, że to ostatni blok, a 0 że zostały jeszcze inne. Natomiast do opisania sposobu kompresji służą dwa bity. Wyróżniamy następujące bity kompresji:

- 00 - brak kompresji
- 01 - kodowanie ze statycznymi kodami Huffmana
- 10 - kodowanie z dynamicznymi kodami huffmana
- 11 - błąd

2.6.2. Tryb bez kompresji bloku danych

Pierwszy tryb jest w przypadku braku kompresji. Będzie on przy początkowych bitach 000 oraz 100. W tym przypadku pomijamy następne 5 bitów (do pełnego bajta), a następnie odczytujemy czterobitowy parametr *len* typu unsigned int 16-bit informujący o rozmiarze bloku. Kolejnym krokiem jest odczytanie bloku (kolejne *len* bajtów).

2.6.3. Tryb kodowania ze statycznymi kodami Huffmana

Zanim rozpocznie się opisywanie drugiego trybu, warto przedstawić zawartość wbudowanych tabel długości i odległości w algorytm. Poniższe tabele służą do kopiowanie powtarzających się fragmentów bitów w danych do skompresowania. Poniżej znajduje się tabela z kodami długości używanych przez algorytm w wariancie ze statycznymi kodami Huffmana.

Tabela 2.4. Tabela ze statystycznymi kodami długości dla algorytmu Deflate

Kod	Dodatkowe bity	Długości	Kod	Dodatkowe bity	Długości
257	0	3	272	2	31 - 34
258	0	4	273	3	35 - 42
259	0	5	274	3	43 - 50
260	0	6	275	3	51 - 58
261	0	7	276	3	59 - 66
262	0	8	277	4	67 - 82
263	0	9	278	4	83 - 98
264	0	10	279	4	99 - 114
265	1	11, 12	280	4	115 - 130
266	1	13, 14	281	5	131 - 162
267	1	15, 16	282	5	163 - 194
268	1	17, 18	283	5	195 - 226
269	2	19 - 22	284	5	227 - 257
270	2	23 - 26	285	0	258
271	2	27 - 30			

Poza kodami długości, istnieją również kody odległości przedstawione w tabeli 2.3 znajdującej się poniżej.

Tabela 2.5. Tabela ze statystycznymi kodami odległości dla algorytmu Deflate

Kod	Dodatkowe bity Odległość	Kod	Dodatkowe bity	Odległość
0	0 1	15	6	193-256
1	0 2	16	7	257-384
2	0 3	17	7	385-512
3	0 4	18	8	513-768
4	1 5,6	19	8	769-1024
5	1 7,8	20	9	1025-1536
6	2 9-12	21	9	1537-2048
7	2 13-16	22	10	2049-3072
8	3 17-24	23	10	3073-4096
9	3 25-32	24	11	4097-6144
10	4 33-48	25	11	6145-8192
11	4 49-64	26	12	8193-12288
12	5 65-96	27	12	12289-16384
13	5 97-128	28	13	16385-24576
14	6 129-192	29	13	24577-32768

Domyślna długość kodu długości wynosi 5. Może być ona rozszerzona o dodatkowa bity określone w powyższej tabeli. Przykładowo jak chcemy zakodować odległość 4, kod w postaci binarnej będzie zapisany jako 00011, gdy chcemy zakodować odległość 5, dodajemy do ciągu dodatkowy bit i uzyskujemy 001000, a gdy chcemy zakodować wartość 6, to kod Huffmana będzie równy 001001.

Drugi tryb jest w przypadku kodowania ze statycznymi kodami Huffmana. Będzie on przy początkowych bitach 001 oraz 101. Po początkowych bitach od razu znajdują się kody znaków lub kody odległości zdefiniowane w tabelach poniżej. Ostatnim elementem tego trybu jest znak końca bloku. Odpowiedzialny jest za to kod „256”.

W tym trybie schemat działania algorytmu dekompresji wygląda następująco. Należy czytać kolejne bity wejściowe aż do odczytania znaku końca bloku. Gdy kod jest mniejszy od 256 należy skopiować znak do wyjściowego pliku, gdy jest większy należy odczytać długość L z tabeli długości, a następnie odczytać kilka kolejnych bitów i odnaleźć dany ciąg w tabeli odległości. Kolejnym krokiem jest cofnięcie się o dany dystans z tabeli odległości, a następnie skopiować L bajtów od danej pozycji na koniec strumienia wyjściowego. Pętle przerywa odczytanie kodu 256.

W algorytmie Deflate kody nie są zapisane w zwykłej formie. W celu zmniejszenia objętości skompresowanego pliku stosuje się dodatkowo kodowanie Huffmana dla kodów znaków, kodów długości oraz kodów odległości. W tym trybie stosuje się statyczną tabelę wbudowaną w kod programu. Tabela jest przedstawiona poniżej.

Tabela 2.6. Statyczna tabela kodów Huffmana dla algorytmu Deflate

Wartość	Liczba bitów	Kod Huffmana
0–143	8	00110000 – 10111111
144–255	9	110010000 – 111111111
256–279	7	0000000 – 0010111
280–287	8	11000000 – 11000111

W celu znalezienia danej wartości dla danego kodu Huffmana, znajduje się jego przedział, a następnie od danego kodu odejmuje wartość początkową z tego przedziału. Przykładowo dla kodu Huffmana 0000011 będzie wartość 259, a dla wartości 254 będzie pasował kod 111111110.

Najłatwiej pokazać zasadę działania algorytmu na przykładzie. Załóżmy, że mamy do skompresowania ciąg aaabaabccc w kodowaniu ASCII. Pierwszy znak ma kod 97. Jego kod Huffmana wyniesie 01100001. Przepisujemy tą wartość na wyjście. Następnie robimy analogicznie dla 3 kolejnych wartości. W następnym kroku sprawdzamy, czy znaki mogą się powtarzać. Najkrótsza dostępna długość do skopiowania z tabeli długości ciągów wynosi 3. W tym przypadku właśnie ta wartość będzie oczekiwana. Sprawdzamy długość najdłuższego ciągu jaki się powtarza. Będzie to podciąg „aab"o długości 3 znaków. Zatem zapisujemy kod 257 jako 000000, a następnie kod dystansu 3 jako 2 (10 w postaci binarnej). Na końcu zapisujemy 3 znaki „c"z początkowego pliku.

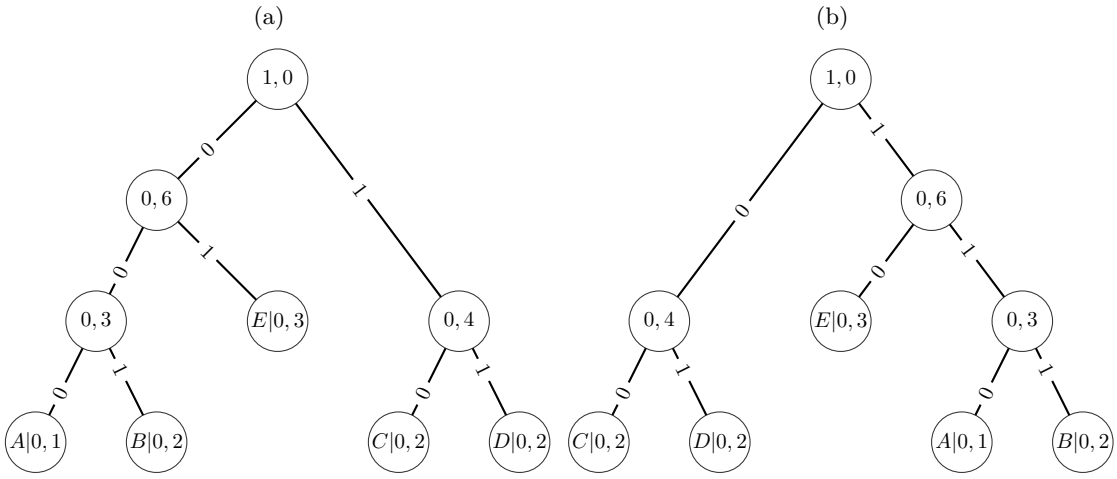
2.6.4. Tryb kodowania z dynamicznymi kodami Huffmana

Trzeci tryb odróżnia się od drugiego tym, że algorytm nie korzysta z wbudowanej tablicy kodów Huffmana. W tym trybie tabela kodów Huffmana jest oddzielnie generowana dla tabeli długości i odległości, przy czym w dokumentacji algorytmu Deflate nie jest określone w jaki sposób ma przebiegać dobór kodów. Każdy blok ma oddzielny ich zestaw. Program może zbierać dane z kolejnych bloków i na podstawie tego poprawiać stopień kompresji dla każdej następnej części pliku. W tym trybie kody Huffmana znajdują się bezpośrednio za bitami nagłówka oraz tuż przed danymi do skompresowania, ułożone w kolejności najpierw kody długości, a następnie kody odległości. Obowiązują dwie zasady tworzenia drzewa tych kodów:

- Kody krótszej długości znajdują się z lewej strony, a dłuższe z prawej.
- Gdy jest kilka symboli o takiej samej długości, mniejsze leksykograficznie symbole znajdują się po lewej stronie.

Powyższe zasady można w lepszy sposób zobrazować modyfikując drzewo Huffmana z rozdziału „Kodowanie Huffmana". Analizowany był tam zbiór $S = \{A, B, C, D, E\}$ o prawdopodobieństwach $\{0,1; 0,2; 0,2; 0,2; 0,3\}$. Na poniższym drzewie a znajduje się omówione wcześniej drzewo. Na rysunku b znajduje się drzewo dopasowane do algorytmu Deflate.

Rys. 2.22. Dwa drzewa Huffmana



Gdy drzewo jest już zbudowane, następnym krokiem będzie wyznaczenie długości dla każdego symbolu. W tym przypadku będzie to $3, 3, 2, 2, 2$. W algorytmie Deflate występuje ograniczenie maksymalnej długości symbolu do 15. Wynika to z tego, że do zapisania wartości wykorzystywane są 4 bity. Nieco bardziej skomplikowane jest odczytanie kodów Huffmana na podstawie długości. Pierwszym krokiem będzie zliczenie występowania każdego kodu. W tym przypadku wartość 2 będzie występowała 3 razy, wartość 3 dwa razy, a pozostałe wartości nie będą występowały. W pierwszym kroku algorytm powinien zwrócić następujący wynik: $F = [0, 0, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$. F oznacza długości liczbę występowania symboli o danej długości. Następnym krokiem będzie określenie bazowej wartości kodu potrzebnej do wyznaczenia kodu Huffmana. W tym celu tworzy się nową tablicę szesnastoznakową N. W tablicy kolejny element będzie oznaczony jako N_i. Pierwszy element będzie miał wartość równą 0, a następne elementy są obliczone według poniższego wzoru.

$$N_i = 2(N_{i-1} + F_{i-1})$$

Powinno to zwrócić następujący wynik:
$N = [0, 0, 0, 6, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]$. Trzecim etapem będzie wyznaczenie samych kodów Huffmana. W tym celu należy utworzyć kolejną tablicę H zawierającą kody Huffmana. Wzór na wyznaczanie kolejnego kodu jest równy $H_i = N_{F_i}; N_{F_i} := N_{F_i} + 1$. Wynik dla przykładu powinien być równy $H = [6, 7, 0, 1, 2]$, co po przeliczeniu na wartości binarne jest równe $H = [110, 111, 00, 01, 10]$. Końcowe wartości zgadzają się z powyższym rysunkiem b.

Kody długości i odległości zawierają w sumie 315 symboli. Jest to już znacząca wartość. Aby uzyskać jeszcze lepszą kompresję, stosuje się kodowanie drzewa kodów Huffmana za pomocą zbioru kodów składających się z 19 symboli w następujący sposób.

- 0...15 - kod danego symbolu. W tym przypadku kopiuje się znak do strumienia wyjściowego.
- 16 - należy powtórzyć poprzedni znak od 3 do 6 razy. W tym celu należy odczytać 2 następne bity i dodać do nich liczbę 3.
- 17 - należy powtórzyć znak 0 od 3 do 10 razy. Aby odszukać informację ile należy zrobić powtórzeń, trzeba odczytać następne 3 bity i dodać do nich liczbę 3.

- 18 - należy powtórzyć znak 0 od 11 do 138 razy. W tym celu należy odczytać kolejne 7 bitów i dodać do nich 11.

Liczba 0 oznacza, że dany symbol nie występuje w danych do kompresji i nie powinien być brany pod uwagę podczas budowy drzewa Huffmana. Przykładowo gdy mamy sekwencję 1, 2, 3, 16 (dodatkowe bity 10), 17 (dodatkowe bity 000), 2, to na wyjściu otrzymamy wartość: 1, 2, 3, 3, 3, 3, 3, 3, 0, 0, 0, 2.

Format bloku w tym trybie wygląda następująco. Na samym początku jest liczba symboli w alfabecie znaków/długości pomniejszona o 257. W dalszej części będzie oznaczona symbolem rozmiarTabeliAlfabetu. Parametr zajmuje 5 bitów. Następnie jest pięciobitowy parametr odpowiadający za liczbę symboli w tabeli odległości pomniejszoną o 1. Parametr będzie oznaczony jako rozmiarTabeliOdległości Kolejnym elementem jest czterobitowa liczba LiczbaElementówWDrzewie odpowiadająca za liczbę elementów w drzewie kodów Huffmana pomniejszona o 4. Kolejne pole zawiera potrojoną wartość liczbaElemntówWDrzewie długości prefiksów kodów używanych do dekodowania alfabetów długości i odległości. Kolejne kody ułożone są w następującej kolejności 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. W przypadku gdy nie wszystkie symbole były użyte do kompresji, niewykorzystane wartości indeksów będą wynosiły zero.

Powyższy opis można zilustrować na przykładzie. Dla zmiennej rozmiarTabeliAlfabetu o wartości 5 liczba elementów w tabeli znaków/długości będzie wynosiła 262 elementy. Dla zmiennej rozmiarTabeliDługości wynoszącej ponownie 5 liczba elementów w tablicy długości będzie wynosić 6. Gdy zmienna
liczbaElemtówWDrzewie będzie miała wartość 5, będzie to oznaczało, że wszystkie kody Huffmana będą zapisane za pomocą 8 bitów lub nie będą występować. Wynika to z tego, że zapisze się tylko pierwsze pięć kodów opisanych w poprzednim akapicie.

Ostatnim dodatkowym elementem w stosunku do trybu drugiego są zapisane kody Huffmana zakodowane za pomocą prefiksów przedstawionych powyżej. Ich sumaryczna liczba wynosi rozmiarTabeliAlfabetu + rozmiarTabeliOdległości. W pierwszej kolejności znajdują się elementy z tabeli długości, a następnie z tabeli odległości. Ostatnim elementem bloku są skompresowane dane w sposób analogiczny jak w trybie drugim.

2.7. Szczegółowy opis algorytmu LZMA

Kolejnym bardzo popularnym algorytmem jest LZMA, zaprojektowany przez Igora Pavlova pomiędzy 1996 r. i 1998 r.. Algorytm jest zmodyfikowaną wersją algorytmu kompresji słownikowej LZ77, umożliwiający uzyskanie wysokiego stopnia kompresji, jednocześnie szybkiej dekompresji i niewielkiego zapotrzebowania na pamięć podczas dekompresji.

Można znaleźć pewne podobieństwo algorytmu LZMA do opisanego w poprzednim podrozdziale algorytmu Deflate. W obu przypadkach wykorzystywana jest metoda słownikowa, a następnie jest kodowanie wyjścia. Główna różnica jest taka, że dla LZMA stosuje się kodowanie zakresu (ang. range encoding), które pozwala na skuteczniejszą kompresję, dzięki wykorzystaniu entropii danych.

Metoda słownikowa opiera się na dwóch buforach - słownikowym znajdującym się po

lewej stronie od pozycji algorytmu w tekście oraz buforze wejściowym znajdującym się po prawej stronie. Celem algorytmu jest znalezienie możliwie najdłuższego ciągu w buforze wejściowym, który istnieje w buforze słownikowym. Wynik działania algorytmu zapisuje się w postaci sekwencji krotek składających się z trzech elementów $< p, d, z >$, które kolejno oznaczają przesunięcie w buforze słownikowym, długość dopasowanego ciągu znaków oraz następny znak do zakodowania. W sytuacji gdy nie uda się znaleźć pasującej sekwencji, na wyjście zostanie zapisany pojedynczy znak. Kodowanie zostało szczegółowo opisane w następnym podrozdziale pt. „Format danych algorytmu LZMA". Więcej informacji znajduje się w rozdziale „Szczegółowy opis algorytmu LZ77".

2.7.1. Format danych algorytmu LZMA

W algorytmie LZMA zapisuje się ciąg bitów zakodowanych za pomocą kodowania zakresu podzielonych na pakiety. Pakiety zawierają albo pojedynczy bajt, albo wskaźnik do wcześniej znalezionego ciągu. W celu optymalizacji procesu wyszukiwania dopasowanych ciągów, stosuje się historię czterech ostatnio używanych dystansów, które zostały przedstawione w poniższej tabeli. Wyróżnia się siedem różnych typów pakietów:

Tabela 2.7. Typy pakietów w algorytmie LZMA

Kod pakietu	Nazwa pakietu	Opis pakietu
0 + kodBajtu	LIT	Pojedynczy bajt zakodowany za pomocą kodowania zakresu
1 + 0 + len + dist	MATCH	Typowa sekwencja LZ77 opisująca długość dopasowanego ciągu oraz przesunięcie w buforze słownikowym
1 + 1 + 0 + 0	SHORTREP	Jedno-bajtowa sekwencja LZ77, w której dystans jest równy ostatnio używanemu dystansowi LZ77
1 + 1 + 0 + 1 + len	LONGREP[0]	Sekwencja LZ77 o *len* długości dopasowanego ciągu, w której dystans jest równy ostatnio używanemu dystansowi LZ77
1 + 1 + 1 + 0 + len	LONGREP[1]	Sekwencja LZ77 o *len* długości dopasowanego ciągu, w której dystans jest równy drugiemu ostatnio używanemu dystansowi LZ77
1 + 1 + 1 + 1 + 0 + len	LONGREP[2]	Sekwencja LZ77 o *len* długości dopasowanego ciągu, w której dystans jest równy trzeciemu ostatnio używanemu dystansowi LZ77
1 + 1 + 1 + 1 + 1 + len	LONGREP[3]	Sekwencja LZ77 o *len* długości dopasowanego ciągu, w której dystans jest równy czwartemu ostatnio używanemu dystansowi LZ77

Do zakodowania pakietu stosuje się różne kodowania długości dopasowania ciągu. Przedstawia to poniższa tabela.

Tabela 2.8. Kodowanie długości dopasowania w algorytmie LZMA

Kod długości	Opis kodu długości
0 + 3 bity	Długość dopasowania wykorzystująca 3 bity. Pozwala zakodować dopasowania o długości od 2 do 9.
1 + 0 + 3 bity	Długość dopasowania wykorzystująca 3 bity. Pozwala zakodować dopasowania o długości od 10 do 17.
1 + 1 + 8 bitów	Długość dopasowania wykorzystująca 8 bitów. Pozwala zakodować dopasowania o długości od 18 do 273.

Do zakodowania pakietu trzeba również zakodować dystans. Odległości są kodowane z użyciem maksymalnie 32 bitów zapisane w formacie big endian, czyli takim w którym najbardziej znaczące bity są na początku. Format używany do zakodowania odległości przedstawiony jest w poniższej tabeli. Pierwsze pole zawiera informację ile dodatkowych bitów potrzeba do zakodowania dystansu, drugie wynika bezpośrednio z pierwszego, w trzecim znajdują się bity prawdopodobieństwa zapisane z dokładnością do $0,5$, a w ostatniej znajdują się bity związane z zawartością danych.

Tabela 2.9. Kodowanie odległości w algorytmie LZMA

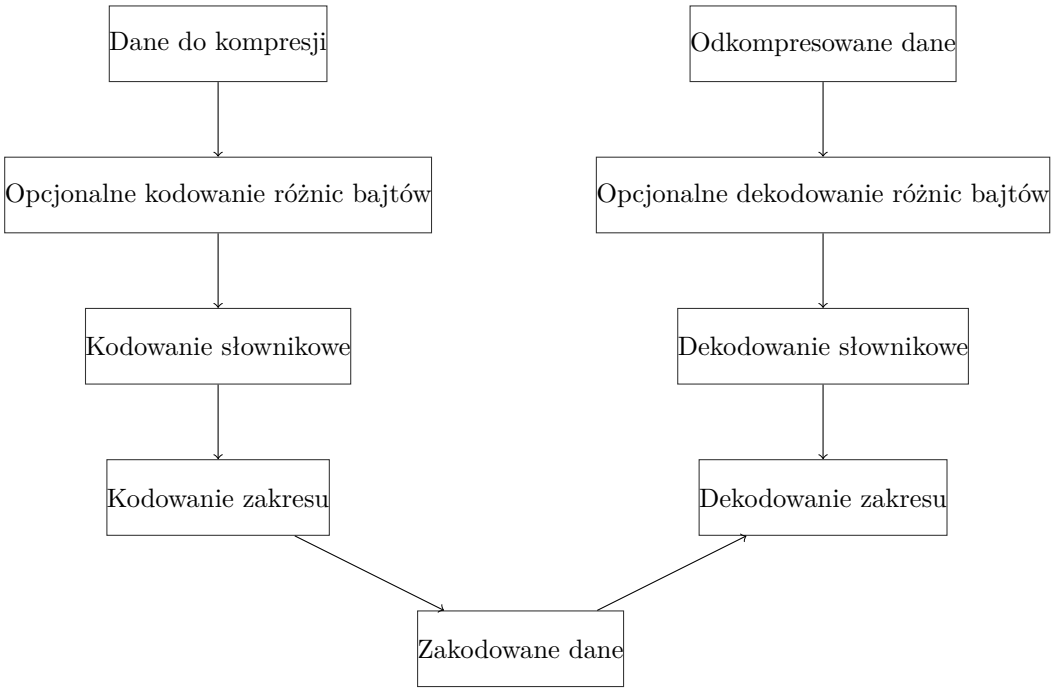
Bity długości (6 bitów)	Najwyższe 2 bity	Bity prawdopodobieństwa	Bity kodowanego zakresu
0	00	0	0
1	01	0	0
2	10	0	0
3	11	0	0
4	10	0	1
5	11	0	1
6	10	0	2
7	11	0	2
8	10	0	3
9	11	0	3
10	10	0	4
11	11	0	4
12	10	0	5
13	11	0	5
14 - 62 (parzyste)	10	((bityDługosci / 2) -5)	4
15 - 63 (nieparzyste)	11	(((bityDługosci - 1) / 2) -5)	4

2.7.2. Ogólna zasada działania algorytmu LZMA

Zarówno algorytm kompresji jak i dekompresji można podzielić na 3 podstawowe kroki. W pierwszym, opcjonalnym kroku dane są modyfikowane w taki sposób, aby kolejne bajty nie zawierały swoich danych, a były równe różnicy w stosunku do poprzedniego bajtu. W drugim wykonywana jest główna część algorytmu słownikowego - znajdowane są powtarzające się ciągi i zapisywane są za pomocą indeksów. W trzecim dane są zapisywane za pomocą kodowania zakresu. Dekompresja przebiega analogicznie, w odwrotnej kolejności. Kroki algorytmu można przedstawić na poniższym schemacie.

Rys. 2.23. Kroki algorytmu LZMA

(a)



Kroki pokazane na schemacie zostaną szczegółowo opisane w kolejnych podrozdziałach.

2.7.3. Kodowanie różnic bajtów

Algorytm LZMA pozwala na kodowanie różnic bajtów (ang. delta coding). Polega ono na zapisaniu kolejnych bajtów danych jako różnica względem poprzedniego. W implementacji LZMA pierwszy bajt zawsze będzie miał swoją wartość, a kolejne będą zawierały różnice. Można to przedstawić na przykładzie. Załóżmy, że na wejściu znajduje się ciąg:
$1, 2, 3, 10, 12, 7, 8, 3$
Po zastosowaniu kodowania różnic bajtów, wynik będzie następujący:
$1, 1, 1, 7, 2, -5, 1, -5$

W powyższym przykładzie zamiast 7 wartości do zakodowania, będą 4. Właśnie dlatego stosuje się kodowanie różnic bajtów. Pozwala ono poprawić efektywność kompresji. Szczególnie przydatne jest to przy zdjęciach, gdzie często występują niewielkie różnice pomiędzy kolejnymi pikselami.

2.7.4. Wykrywanie dopasowań w algorytmie LZMA

Algorytm LZMA może korzystać nawet z wielogigabajtowych słowników, stąd zwykłe wyszukiwanie liniowe byłoby bardzo nieefektywne. Aby rozwiązać ten problem została wykorzystana funkcja skrótu (ang. hash). Znalezienie dopasowania w buforze słownikowym odbywa się za pomocą haszowania dwóch, trzech lub czterech bajtów w zależności od rozmiaru buforu słownikowego. Za każdym razem zostanie użyty aktualny bajt wykorzystywany przez algorytm z buforu wejściowego oraz następne bajty znajdujące się zaraz obok niego po prawej stronie. Wynikiem haszowania jest indeks do tablicy w której znajdują się dopasowania. Sam rozmiar tablicy jest

zdefiniowany za pomocą kolejnych potęg liczby 2, przy założeniu, że rozmiar tablicy najczęściej się równa możliwie blisko połowy rozmiaru słownika. Czyli dla rozmiaru słownika 64 MB (2^{26}) rozmiar tablicy dopasowań będzie równy 2^{25} a funkcja skrótu powinna zwrócić indeks o długości 25 bitów.

W przypadku mniejszej tablicy dopasowań zwiększa się ryzyko kolizji dopasowań o tym samym haszu. Aby to rozwiązać należy zwiększyć rozmiar tablicy. Warto jednak zauważyć, że samo zwiększanie tablicy bez zwiększania liczby bajtów do haszowania nie ma większego sensu, gdyż z każdego bajtu można wygenerować skończoną liczbę haszy. Dla dwóch bajtów będzie to wartość 2^{16}, czyli 64 kB, dla trzech bajtów 2^{24} czyli 16 MB, a dla czterech 2^{32} czyli 4 GB.

Gdy uzyska się indeks, w następnym kroku należy zapisać lub znaleźć dopasowanie w tablicy dopasowań. W algorytmie LZMA istnieją dwie metody umożliwiające to. Pierwsza, szybsza metoda to łańcuch skrótu (ang. hash-claims). Druga, bardziej efektywna, ale za to wolniejsza to metoda drzewa binarnego (ang. binary tree). Obie zostaną szczegółowo opisane w dwóch kolejnych podrozdziałach.

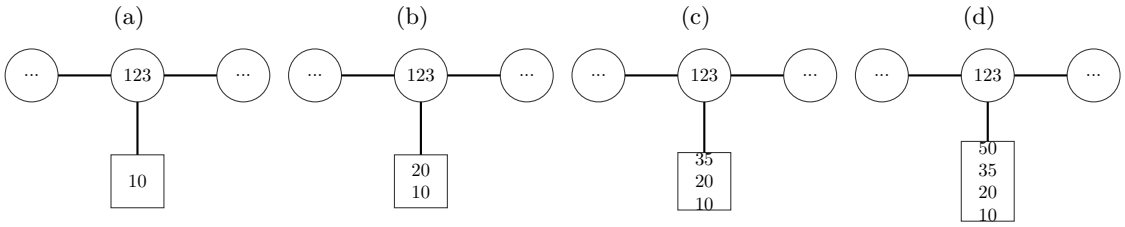
Samo generowanie haszy można przedstawić na przykładzie. Zakładając, że algorytm ma zakodować dane *pracamagisterska* oraz, że do znalezienia dopasowania algorytm wykorzystuje dwa bajty, hasz zostanie wygenerowany dla każdej pary *pr*, *ra*, *ac*, *ca*, *am*, *ma*, *ag*, *gi*, *is*, *st*, *te*, *er*, *rs*, *sk*, *ka*.

2.7.5. Wykrywanie dopasowań w algorytmie LZMA - Łańcuch skrótu

W metodzie łańcucha skrótu wygenerowany hasz wskazuje na listę pozycji buforu słownikowego w tablicy dopasowań. Jako że lista może być bardzo długa, to algorytm umożliwia ograniczenie jej maksymalnego rozmiaru do dowolnej wartości X, przy czym wartością domyślną ograniczenia jest 24. W takiej sytuacji będą tam przechowywane X ostatnich pozycji. W przypadku, gdy się doda nową wartość na danym miejscu, będzie ona porównywana jako pierwsza, gdy ponownie dane miejsce zostanie sprawdzone. Najłatwiej zilustrować to na przykładzie. Wyobraźmy sobie, że mamy ciąg znaków, w którym kilkukrotnie pojawia się podciąg ab, funkcja skrótu wyznaczy wartość 123 dla podciągu. Ciąg znaków wygląda następująco:

Dane: .....abf...abd...abc...abe...
Pozycja: ..10.....20.....35....50....

Rys. 2.24. Przykład łańcucha skrótu w algorytmie LZMA



Dla podanego przykładu, gdy algorytm dotarł do pozycji 10, znalazł znak „a". Następnie wziął znak ten znak oraz następny „b" i wygenerował z tego pozycję 123. W ten sposób ciąg zaczynający się od „abf" został zapisany do tabeli dopasowań. W następnym kroku algorytm

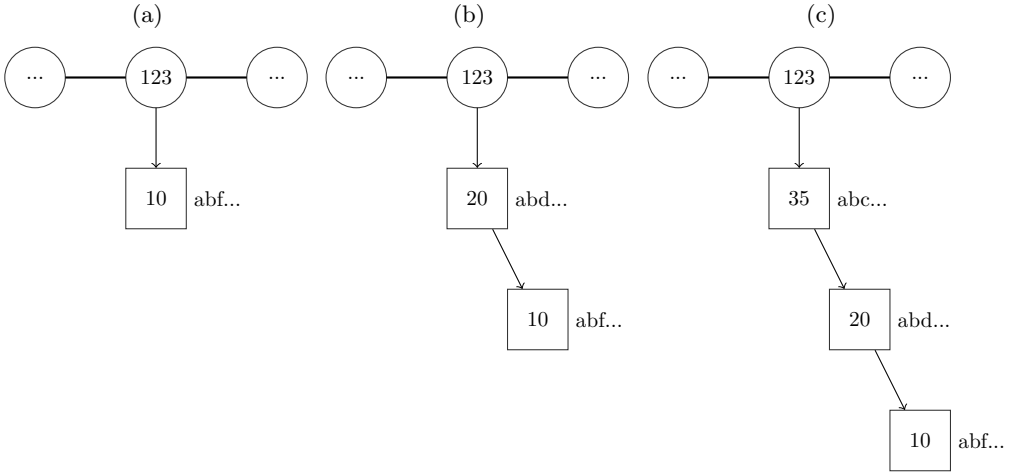
znalazł ciąg zaczynający się od „abd”, który również ma ten sam hasz. Ponownie zapisał jego wartość, przy czym wartość została dodana na początek listy. Dwa następne kroki są analogiczne do poprzednich.

2.7.6. Wykrywanie dopasowań w algorytmie LZMA - Drzewa binarne

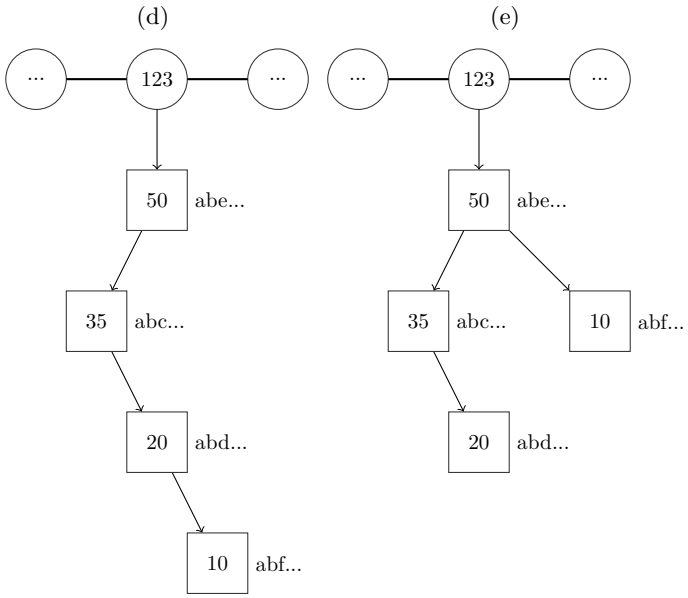
Drugą metodą wykorzystywaną w algorytmie LZMA do znajdowania dopasowań jest zrealizowana za pomocą binarnych drzew poszukiwań (ang. Binary Search Tree, BST). W przeciwieństwie do poprzedniej metody, w tej dane są sortowane leksykograficznie. Wynika to z budowy takich drzew binarnych. Ich głównym założeniem jest to, że liście po lewej stronie danego węzła będą zawierały wyłącznie mniejsze elementy, a liście znajdujące się po prawej będą zawierały wyłącznie elementy nie mniejsze od klucza węzła. Do przykładu budowy takiego drzewa zostaną wykorzystane dane z poprzedniego podrozdziału.

Dane: .....abf...abd...abc...abe...
Pozycja: ..10.....20.....35....50....

Rys. 2.25. Przykład drzewa binarnego w algorytmie LZMA



Rys. 2.26. Przykład drzewa binarnego w algorytmie LZMA



Na powyższym przykładzie algorytm najpierw znalazł ciąg zaczynający się od „abf", który został zapisany pod indeksem 123. W następnym kroku algorytm znalazł ciąg zaczynający się od „abd". Nowa wartość zawsze zostaje zapisana w korzeniu drzewa a następnie porównywana jest dana wartość, z poprzednią wartością korzenia. W tym przypadku poprzednia wartość korzenia była większa, więc została zapisana po prawej stronie. Krok przedstawiony na obrazku c jest analogiczny do przedstawionego na rysunku b. Nieco ciekawsze jest przekształcenie na rysunkach d i e. W kroku d nowa wartość wynosi „abe" i jest leksykograficznie większa od „abc". W tej sytuacji poprzedni korzeń jest zapisywany po lewej stronie. W ten sposób utworzone drzewo nie jest poprawnym drzewem poszukiwań, gdyż wartość w liściu 10 jest leksykograficznie większa od korzenia. W tej sytuacji należy naprawić drzewo i przenieść liść na prawą stronę od korzenia. W punkcie e uzyskano poprawne drzewo poszukiwań.

2.7.7. Kodowanie słownikowe w algorytmie LZMA

Gdy już znane jest w jaki sposób można wyszukiwać dopasowania, można przejść do głównej części algorytmu. W algorytmie nie jest określone w jaki sposób należy znaleźć dopasowanie, ani jaka powinna być jego długość. Aby optymalnie znaleźć dopasowania, program kompresujący musiałby przed rozpoczęciem kompresji mieć wiedzę na temat zawartości kompresowanych danych. Jednak ze względu na efektywność nie stosuje się takiego podejścia, a zamiast tego różne implementacje stosują ograniczenie maksymalnego rozmiaru dopasowania. W dalszym opisaniu kroków algorytmu zmienna będzie się nazywać OdpowiedniaDługość. Gdy znajdzie takie dopasowanie, przestaje dalej szukać. Poniższy schemat przedstawia podstawowe kroki algorytmu LZMA. Warto jednak pamiętać, że różne implementacje mogą mieć nieco zmienione wyszukiwanie powtarzających się sekwencji.

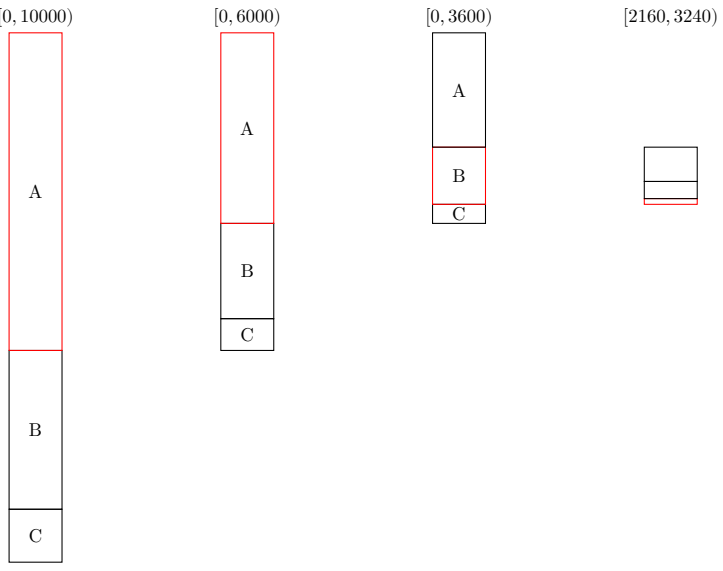
Kodowanie słownikowe w algorytmie LZMA można podzielić na kilka kroków. W pierwszym należy znaleźć powtarzające się sekwencje wykorzystując metody opisane w poprzednich podrozdziałach. Następnie należy sprawdzić czy dopasowanie znajduje się w ostatnio używanych

dopasowaniach. Szczegóły odnośnie zapisu ostatnio używanych dopasowań zostały opisane w podrozdziale „Format danych algorytmu LZMA”. Jeżeli udało się znaleźć w ostatnio używanych dopasowaniach i jednocześnie długość dopasowanego ciągu wynosi co najmniej OdpowiedniaDługość, na wyjście należy zapisać pakiet *REP. Jeżeli udało się znaleźć dopasowanie o długości co najmniej OdpowiedniaDługość, ale nie było go w ostatnio używanych, należy na wyjście zapisać pakiet MATCH. W przeciwnym przypadku, gdy znalezione dopasowanie jest krótsze, należy sprawdzić kolejne najbliższe dopasowania każdej długości w kolejności od najdłuższych do najkrótszych i sprawdzić czy dopasowanie było podzbiorem ostatnio zapisanych dopasowań. Jeżeli tak, to należy zapisać na wyjście pakiet *REP z odpowiednią długością dopasowania. Gdy nowego dopasowania nie ma w ostatnio używanych, należy zapisać na wyjście pakiet MATCH. W sytuacji, gdy nie znaleziono dopasowania lub dopasowanie jest długości jednego bajtu, na wyjście zapisywany jest pakiet LIT.

2.7.8. Kodowanie zakresu - ogólna zasada działania

Ostatnim etapem kodowania algorytmu jest kodowanie zakresu. Założeniem kodowania zakresu jest to, że zamienia się wszystkie symbole w jedną liczbę. Na wejściu znajduje się zbiór elementów oraz prawdopodobieństwo występowania każdego z nich w danym zbiorze. Algorytm kodowania rozpoczyna się od podziału zbioru na N podzbiorów. Rozmiary kolejnych podzbiorów są wprost proporcjonalne do prawdopodobieństw występowania symboli w zbiorze danych. Gdy tabela prawdopodobieństw jest zainicjowana, następnym krokiem ustawianie początku i końca zakresu oraz dzielenie zbiorów na kolejne, mniejsze podzbiory. Najłatwiej to pokazać na przykładzie. Na poniższym przykładzie zostanie zakodowany ciąg $AABC$, a zbiór będzie posiadał następujące prawdopodobieństwo wystąpień symboli: $\{A: 0,6; B: 0,3; C: 0,1\}$. Wartości zostały wybrane arbitralnie, a każdy algorytm na własny sposób na ich dobór. Na przykładzie do zakodowania symboli zostanie wykorzystany system dziesiątkowy. Jako że w ciągu znajdują się 4 symbole, liczba wszystkich możliwych kombinacji wynosi 10^4, stąd początkowy zakres będzie wynosił $[0, 10000)$.

Rys. 2.27. Przykład kodowania zakresu dla ciągu $AABC$ i prawdopodobieństwa $\{A: 0,6; B: 0,3; C: 0,1\}$



Na powyższym rysunku została przedstawiona w sposób graficzny zasada działania kodowania zakresu. W pierwszym kroku został podzielony zakres $[0, 1000)$ na trzy mniejsze zakresy:
$A : [0, 6000)$
$B : [6000, 9000)$
$C : [9000, 10000)$

Został wzięty pierwszy symbol z ciągu A, a wartość zakresu po pierwszym kroku wyniosła $[0, 6000)$. W kolejnym kroku analogicznie podzielono zakres na podzbiory:
$AA : [0, 3600)$
$AB : [3600, 5400)$
$AC : [5400, 6000)$

Ponownie następny symbol w kodowanym ciągu jest równy A. Ponownie podzielono zbiór na podzbiory:
$AAA : [0, 2160)$
$AAB : [2160, 3240)$
$AAC : [3240, 3600)$

Następnym znakiem w danych jest B. Po nowym podziale podzbiory wyglądają następująco:
$AABA : [2160, 2808)$
$AABB : [2808, 3132)$
$AABC : [3132, 3240)$

Ostatnim znakiem w ciągu jest C, stąd poszukiwany zbiór będzie miał zakres $[3132, 3240)$. Do zakodowania można użyć trzycyfrowego prefiksu 314. Ze względu na to, że w tym przykładzie korzystano z systemu dziesiątkowego, jest dziesięć pasujących trzycyfrowych prefiksów. Problem zostanie rozwiązany przy używaniu systemu binarnego. Właśnie takiego używa LZMA do kodowania zakresu.

2.7.9. Kodowanie zakresu w algorytmie LZMA

LZMA wykorzystuje dwie metody do kodowania zakresu. Pierwsza zakłada stałe prawdopodobieństwo, druga używa prawdopodobieństwa zależnego od zawartości danych. Do implementacji wykorzystywane są następujące zmienne: *low* oznaczająca początek zakresu, zainicjowana wartością 0, *range* oznaczająca koniec zakresu, zainicjowana wartością $2^{32} - 1$, *cache* zmienna tymczasowa potrzebna do normalizacji, zainicjowana wartością 0 i *cache_size* określająca rozmiar *cache*, zainicjowana wartością 1. Do zakodowania zakresu wykorzystywana jest dodatkowa metoda normalizacji opisana w kilku następujących krokach. W pierwszym należy porównać czy zmienna *low* jest mniejsza niż $2^{32} - 2^{24}$. Jeżeli jest, to należy przepisać bajt ze zmiennej *cache* i zapisać *cache_size* bajtów z wartością $0xFF$ na wyjście, ustawić wartość zmiennej *cache* na bity $24 - 31$ ze zmiennej *low* i ustawić wartość zmiennej *cache_size* na 0. W przypadku gdy wartość *low* jest

większa niż $2^{32} - 2^{24}$, to należy przepisać bajt ze zmiennej *cache* i dodać do niego jeden oraz wykonać analogicznie pozostałe kroki z poprzedniego przypadku. W następnym kroku należy zwiększyć o 1 wartość *cache_size*, ustawić zmienną *low* na najniższe 24 bity zmiennej przesunięte w lewo o 8 bitów, a następnie dokonać przesunięcia bitowego o 8 bitów w prawo w zmiennej *range*.

Prostszą metodą kodowania zakresu jest metoda ze stałym prawdopodobieństwem. W pierwszym kroku sprawdza się czy wartość zmiennej *range* jest mniejsza od 2^{24}. Jeżeli tak, to należy wykonać normalizację opisaną powyżej. W następnym kroku należy ustanowić nową wartość zmiennej *range* poprzez podzielenie jej przez 2 i zaokrąglenie w dół otrzymanego wyniku. W ostatnim kroku należy sprawdzić czy kodowany bit jest równy 1. Jeżeli tak, to należy zwiększyć dolną wartość zakresu *low* o górną wartość zakresu *range*.

Drugą metodą kodowania zakresu jest metoda z prawdopodobieństwem opierającym się na zawartości danych. Dla tej metody również w pierwszym kroku sprawdza się czy wartość zmiennej *range* jest mniejsza od 2^{24} i jeżeli jest, to należy wykonać normalizację. W kolejnym kroku należy do tymczasowej zmiennej *temp* przypisać wartość zaokrągloną w dół $range/2^{11}$ przemnożoną przez wartość prawdopodobieństwa *prob*. Trzeci krok zależy od tego jaki bit jest aktualnie kodowany. Jeżeli jest to 0, to należy przypisać górnej wartości zakresu *range* wartość zmiennej tymczasowej *temp*, a następnie zwiększyć wartość prawdopodobieństwa *prob* o zaokrągloną w dół wartość $(2^{11} - prob)/2^5$. W przypadku gdy aktualnie kodowany bit jest równy 1, to należy przypisać górnej wartości zakresu *range* wartość poprzednią wartość pomniejszoną o wartość zmiennej tymczasowej *temp*. Następnie należy zwiększyć dolną wartość zakresu *range* o wartość *temp*, a na końcu zmniejszyć wartość prawdopodobieństwa *prob* o zaokrągloną w dół wartość $prob/2^{11}$.

Niezależnie od wyboru metody kodowania na końcu należy pięć razy wykonać normalizację. Opis algorytmu jest oparty na bazie kodu źródłowego programu XZ napisanego przez Lasse Collina, w którym nie został wyjaśniony ostatni krok.

3. CZĘŚĆ PRAKTYCZNA

3.1. Opis implementacji

Poprzedni rozdział zawierał ogólny opis metod kompresji oraz wyniki wydajności i jakości popularnego programu dostępnego na rynku. Ten rozdział będzie dotyczyć mojej implementacji aplikacji umożliwiającej kompresję plików w formacie zip oraz wykorzystującego algorytm Deflate. Szczegółowo aplikacja zostanie opisana w kolejnych podrozdziałach.

3.1.1. Wymagania sprzętowe i systemowe

Program działa prawidłowo zarówno na systemach operacyjnych z rodziny Windows jak i Linux, a także innych systemach wyposażonych w Javę w wersji 8 lub wyższej. Dowolność tych systemów wynika z tego, że kod programu został napisany w języku Java. W tym języku kod jest kompilowany do kodu bajtowego, który jest następnie wykonywany przez maszynę wirtualną. Niezależnie od systemu operacyjnego skompilowany kod bajtowy będzie miał taką samą strukturę, jednak wydajność na poszczególnych systemach może się nieznacznie różnić. Na wydajność spory wpływ będzie miało zastosowanie odpowiedniej maszyny wirtualnej oraz wersji Javy. Przykładowo Java w wersji 11 będzie nawet kilkanaście procent szybsza od Javy w wersji 8[32], a kompilując kod w wersji 15 można uzyskać kolejne kilkanaście procent do wydajności[33].

Implementacja kompresji danych wspiera wielowątkowe przetwarzanie danych. Wielowątkowość została zaimplementowana do przetwarzania wielu plików na raz, czy sortowania danych niezbędnych danych do uzyskania kompresji. Aplikacja działa zarówno na procesorach jednowątkowych jak i wielowątkowych, jednak żeby uzyskać lepszą wydajność zaleca się wykonywać program na urządzeniach z wieloma dostępnymi wątkami.

Program ma wymagania co do pamięci RAM użytej do kompresji danych. Zużywa w przybliżeniu od dwukrotnego do czterokrotnego rozmiaru kompresowanego pliku. Nie da się dokładnie oszacować potrzebnego rozmiaru, gdyż wartość zależy od zawartości kompresowanych danych. Przykładowo program, który ma więcej powtarzających się ciągów znaków będzie wymagał więcej pamięci RAM od programu, który nie ma żadnych dwóch takich samych ciągów. Wartość została określona na podstawie wielokrotnych testów z różnymi typami danych.

3.1.2. Dokumentacja aplikacji konsolowej

Do obsługi programu wykorzystuje się interfejs konsolowy. Aby uruchomić program należy w pierwszej kolejności skompilować plik do formatu jar. Jest wiele możliwości dokonania tego, jednak tutaj zostanie przedstawiony jeden ze sposobów. W strukturze programu zostały dodane zależności repozytorium Maven umożliwiające łatwą i szybką kompilację kodu. Aby tego dokonać należy w głównym katalogu programu wykonać poniższe polecenie.

```
mvn package
```

Polecenie wygeneruje w folderze „target” plik jar „filecompresser-1.0.jar” pozwalający na uruchomienie programu. Nazwę wygenerowanego pliku jar definiuje się w pliku z zależnościami „pom.xml” poprzez określenie parametru project.artifactId oraz project.version. Sam pro-

gram przyjmuje kilka różnych argumentów, na podstawie których określa kroki które ma wykonać. Ogólny schemat komendy do uruchomienia programu wygląda następująco.

java -jar filecompresser-1.0.jar tryb_programu ścieżka_do_pliku dodatkowe_parametry

tryb_programu należy zastąpić jedną z dwóch opcji - „compress" oznaczającą kompresję pliku lub folderu lub „decompress" oznaczającą dekompresję archiwum.
Argument ścieżka_do_ pliku należy zastąpić miejscem pliku lub folderu, który chcemy przetwarzać. Ścieżka może być relatywna względem pliku programu lub pełna.

W przypadku dekompresji argument dodatkowe_parametry nie jest wykorzystywany. Przykładowe polecenie dekompresujące plik „test.zip" będzie wyglądało następująco.

java -jar filecompresser-1.0.jar decompress test.zip

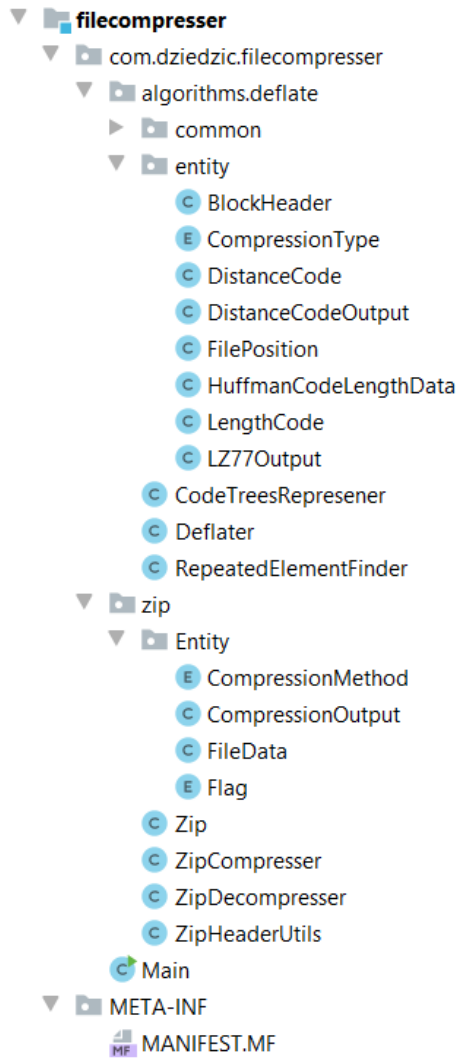
Tryb kompresujący pozwala na zdefiniowanie kilku parametrów pozwalających na dopasowanie jakości i czasu działania programu. Program wspiera następujące dodatkowe_parametry.

- –huffman-codes-mode - Do wyboru są dwie opcje - static i dynamics. Jest to parametr określający tryb wykorzystywanego kodowania Huffmana. Dla statycznego program wykorzysta do kompresji wcześniej zdefiniowane tablice kodów zgodne ze standardem RFC1951. W przypadku dynamicznych program zlicza występowanie każdego symbolu w danym bloku, a następnie stara się dopasować kody Huffmana w sposób optymalny. Domyślnie program będzie dopasowywał automatycznie tryb kodowania kodami Huffmana w zależności od rozmiaru bloków danych używanych do kompresji. W przypadku małych bloków danych kodowanie dynamicznymi kodami może być nieefektywne ze względu na narzut danych związany z zapisaniem definicji używanych kodów. Dla bloków o rozmiarze poniżej 512 bajtów za każdym razem będzie wykonywana kompresja ze statystycznymi kodami. Wynika to z tego, że narzut związany z utworzeniem dynamicznego słownika kodów jest na tyle duży, że byłoby nieopłacalne korzystać z dynamicznych kodów.
- –max-block-size - Parametr określający maksymalny rozmiar bloku danych na jakich działa kompresja algorytmu Deflate. Wartość może być z przedziału 1024 i 32768. W przypadku mniejszych bloków danych zwiększa się szansa na efektywne kodowanie statystyczne (za pomocą kodów Huffmana), a za to w przypadku większych bloków kodowanie słownikowe powinno dać lepsze efekty. Wynika to z tego, że w mniejszych blokach jest większa szansa na to, że będzie mniej różnych symboli, a to przekłada się na to, że kody Huffmana będą mogły być zapisane na mniejszej liczbie bitów. W przypadku kodowania słownikowego, większy bufor oznacza zwiększenie zakresu poszukiwań powtarzających się symboli.
  Przykładowe polecenie kompresujące folder „test" z dynamicznymi kodami Huffmana będzie wyglądało następująco.

  java -jar filecompresser-1.0.jar compress test –huffman-codes-mode dynamics

3.2. Struktura aplikacji

Aplikacja została napisana za pomocą języka Java w wersji 8. Jej poglądową strukturę aplikacji przedstawia poniższy rysunek.



Rys. 3.1. Struktura klas aplikacji

Jak widać na powyższym rysunku, została podzielona na dwa główne pakiety klas - algorytm kompresji Deflate oraz część zarządzającą plikami zip. Pakiet zip jest odpowiedzialny za dwie funkcjonalności - dekompresje i kompresje plików. W przypadku dekompresji odpowiada za odczyt skompresowanych plików, rozpoznanie parametrów kompresji, rozdysponowanie odpowiednich części danych do wątków i na końcu zapisanie przetworzonych plików. W przypadku kompresji zadania są bardzo zbliżone - odczyt plików do kompresji, rozdzielenie danych na wątki, a następnie zapisanie jednego pliku wyjściowego z rozszerzeniem zip. Na powyższym rysunku widać w pakiecie zip cztery klasy oraz folder „Entity". Klasa Zip odpowiada za odczyt argumentów i rozpoznanie trybu działania programu. Na podstawie tego w klasach ZipCompresser lub ZipDecompresser dokonują się metody potrzebne do zapisania lub odczytania plików na dysku. Klasa ZipHeaderUtils zawiera metody niezbędne do prawidłowego odczytania i zapisania informacji w plikach zip na temat parametrów kompresji, jak i poszczególnych plików wewnątrz skompresowanego pliku. W

Folderze Entity znajdują się typy danych używane w pakiecie zip.

Drugi pakiet jest odpowiedzialny za kompresje lub dekompresje danych. W przypadku kompresji, na wejściu przekazywany jest do pakietu ciąg bajtów, w którym następuje jego próba kompresji. Efektem działania metod i klas pakietu jest zwrócenie skompresowanej formy pliku.. Podobnie jest w przypadku dekompresji - pakiet dostaje ciąg skompresowanych bajtów, a zwraca oryginalną wartość pliku.

3.3. Szczegółowy opis wybranych klas

W tym rozdziale zostaną przedstawione kluczowe klasy i metody z punktu widzenia działania programu. W pierwszej kolejności zostaną przedstawione klasy z pakietu zip. Pomimo tego, że format zapisanego pliku nie jest głównym aspektem kompresji, to bez zapisania danych w ściśle określonym formacie nie byłoby możliwości późniejszego odczytania danych.

Format pliku zip ma bezpośredni wpływ na sposób implementacji. Poniższa tabela przedstawia poglądowy format archiwum. Bardzo istotną funkcje pełni nagłówek każdego pliku. Zawiera niezbędne informacje odnośnie metody kompresji, skompresowanego rozmiaru, rozmiaru bez kompresji, sumy kontrolnej pliku, podstawowych danych o pliku jak jego nazwa, czy data modyfikacji.

Tabela 3.1. Format pliku zip

[nagłówek pliku 1]
[dane pliku 1]
[opis danych pliku 1]
.
.
.
[nagłówek pliku n]
[dane pliku n]
[opis danych pliku n]
[nagłówek szyfrowania archiwum]
[dodatkowe dane archiwum]
[nagłówek głównego katalogu pliku 1]
.
.
.
[nagłówek głównego katalogu pliku n]
[koniec głównych katalogów]

W pierwszej kolejności zostanie szczegółowo opisana ścieżka dekompresji plików w tym pakiecie. Jako że nagłówki plików nie są umieszczone jeden po drugim na początku skompresowanego pliku, tylko są w jego różnych częściach, cały plik jest wczytywany na początku do pamięci RAM. Następnie program iteruje po kolejnych nagłówkach i uruchamia przetwarzanie plików w nowych wątkach. Każdy wątek otrzymuje w parametrach jedynie zakres bajtów z aktualnie przetwarzanego pliku. Od uruchomienia przetwarzania plików odpowiedzialny jest poniższy fragment kodu.

Listing 3.1. Odczyt skompresowanych plików z archiwum zip

```
while (isNextFile(content, offset)) {
  if (zipHeaderUtils.checkLocalFileHeaderSignature(Arrays.copyOfRange(content, offset
      , offset + 4))) {
    FileData fileData = zipHeaderUtils.getLocalFileHeader(content, offset);
    threads.add(decompressFile(fileData,
          Arrays.copyOfRange(content,
                offset + fileData.getFileHeaderSize(),
                offset + fileData.getFileHeaderSize() + fileData.getCompressedSize())
                    , path));
    offset += fileData.getFileDataSize();
}}
```

W następnej kolejności program nowy wątek, w którym inicjowana jest klasa odpowiedzialna za rozpoczęcie dekompresji i na końcu zapisaniu przetworzonych plików na dysk. Rozpoczynanie wątków jest zrobione w poniższy sposób.

Listing 3.2. Uruchomienie dekompresji dla pliku

```
private Thread decompressFile(FileData fileData, byte[] content, String path) {
  ZipDecompresser zipDecompresser = new ZipDecompresser(fileData, content, path);
  return zipDecompresser.start();
}
```

W przypadku kompresji pliki przetwarzane są jeden po drugim, a następnie zapisywane do plików tymczasowych, które po zakończeniu działania programu zostaną usunięte z dysku. Tymczasowe pliki są przechowywane na dysku w celu redukcji zapotrzebowania na pamięć RAM. W przypadku kompresji, klasy Zip i ZipCompresser ograniczają się do przekazania kolejnych paczek danych do klas odpowiedzialnych za kompresje, zapisania danych tymczasowych na dysk, utworzenie odpowiednich nagłówków, utworzenia gotowego pliku skompresowanego zip. Przykład w jaki sposób zapisywany jest jeden z nagłówków pliku przedstawia poniższa funkcja.

Listing 3.3. Zapis nagłówka pliku w archiwum zip

```
private byte[] generateCentralDirectoryFileHeader(FileData fileData) {
  byte[] header = new byte[46 + fileData.getFilename().length() + fileData.
      getExtraFieldLength()];
  ZipHeaderUtils zipHeaderUtils = new ZipHeaderUtils();
  int offset = 0;
  offset = zipHeaderUtils.setFileCentralDirectorySignature(header, offset);
  offset = zipHeaderUtils.setVersion(header, offset);
  offset = zipHeaderUtils.setPKZIPVersion(header, offset);
  offset = zipHeaderUtils.setFlags(header, offset);
  offset = zipHeaderUtils.setDeflateCompressionMethod(header, offset);
```

```
  offset = zipHeaderUtils.setModificationDateTime(header, offset, fileData.
      getModificationDateTime());
  offset = zipHeaderUtils.setCRC32Checksum(header, offset, fileData.getCrc32Checksum
      ());
  offset = zipHeaderUtils.setCompressedSize(header, offset, fileData.
      getCompressedSize());
  offset = zipHeaderUtils.setUncompressedSize(header, offset, fileData.
      getUncompressedSize());
  offset = zipHeaderUtils.setFilenameLen(header, offset, fileData.getFileNameLength()
      );
  offset = zipHeaderUtils.setExtraFieldsLen(header, offset, fileData.
      getExtraFieldLength());
  offset = zipHeaderUtils.setExtraCommentsLen(header, offset, fileData.
      getExtraFieldLength());
  offset = zipHeaderUtils.setDiskStart(header, offset, 0);
  offset = zipHeaderUtils.setAdditionalAttributes(header, offset);
  offset = zipHeaderUtils.setFileOffset(header, offset, fileData.getOffset());
  zipHeaderUtils.setFilename(header, offset, fileData.getFilename(), fileData.
      getFileNameLength());
  return header;
}
```

Każda z tych metod zapisuje odpowiednią liczbę bitów z danymi do tablicy wyjściowej. Przykładowa metoda wygląda następująco.

Listing 3.4. Zapis rozmiaru pliku w archiwum zip

```
int setCompressedSize(byte[] output, int offset, int compressedSize) {
  BitReader bitReader = new BitReader();
  bitReader.setBitsLittleEndian(output, offset, 32, compressedSize);
  offset += 32;
  return offset;
}
```

Główna część programu zlokalizowana jest w drugim pakiecie „algorithms". Znajdują się tam dwie rozbudowane klasy - Deflater i CodeTreeRepresenter. Pierwsza z nich jest odpowiedzialna za operacje potrzebne do kompresji i dekompresji. Zasada działania dekompresji polega na odczytywaniu kolejnych bloków danych. W pierwszej kolejności tworzona jest instancja klasy CodeTreeRepresenter, w której odczytywana jest struktura danych potrzebna do dekompresji pozostałych danych. W następnej kolejności odczytywane są kolejne bity danych, aż do odczytania całej zawartości pliku. Odczytane bity następnie są dopasowywane do wygenerowanych wcześniej rekordów z alfabetów i na podstawie tego zapisywane są bajty na wyjście. W celu poprawy wydajności tego procesu odczytywana jest grupa bitów, z których następuje próba dopasowania podzbioru

bitów do wygenerowanych alfabetów.

Listing 3.5. Główna pętla algorytmu dekompresji

```
private void readBlock(byte[] content, BitReader bitReader, CodeTreesRepresener
    codeTreesRepresener, byte[] output, FilePosition filePosition) {
  boolean endOfBlock = false;
  int bitsNumber = codeTreesRepresener.getBiggestHuffmanLength();
  boolean canReuseBits = false;
  int codeInt = 0;

  while (!endOfBlock) {
    if (!canReuseBits)
      codeInt = bitReader.getBits(content, filePosition.getOffset(), bitsNumber);
    else
      codeInt = codeInt >> 1;
    canReuseBits = true;

    HuffmanCodeLengthData huffmanLengthCode = codeTreesRepresener.
        getHuffmanLengthCode(bitsNumber, codeInt);

    if (huffmanLengthCode != null) {
      filePosition.increaseOffset(bitsNumber);
      if (huffmanLengthCode.getIndex() < END_OF_BLOCK) {
        copyByteToOutputStream(output, filePosition, huffmanLengthCode);
      }
      else if (huffmanLengthCode.getIndex() == END_OF_BLOCK)
        endOfBlock = true;
      else {
        copyMultipleBytesToOutputStream(content, bitReader, codeTreesRepresener,
            output, filePosition, huffmanLengthCode);
      }
      bitsNumber =  codeTreesRepresener.getBiggestHuffmanLength();
      canReuseBits = false;
      continue;
  }
  bitsNumber  ;
}
```

W klasie CodeTreeRepresenter odczytuje się i zapisuje się informacje o strukturze danych potrzebnej do przetworzenia skompresowanej części danych. Interesujące są w tej klasie dwie metody. Pierwsza potrzebna do wygenerowania dynamicznych kodów Huffmana na podstawie zna-

nych długości poszczególnych kodów, a druga do znalezienia potrzebnych długości dla kodów.

Metoda generateDynamicsHuffmanLengthCodes na wejściu przyjmuje listę elementów z podanymi długościami kolejnych kodów Huffmana. Pierwszym krokiem potrzebnym do znalezienia wartości kodów jest zliczenie wartości występowania każdej długości i zapisania tego w postaci tablicy. Początek metody, inicjacja danych w niej i zliczenie występowania kodów wygląda następująco.

Listing 3.6. Początek metody generateDynamicsHuffmanLengthCodes

```
void generateDynamicsHuffmanLengthCodes(List<HuffmanCodeLengthData>
    huffmanCodeLengthDataList) {
  int [] codeLengthOccurrences = new int[MAX_HUFFMAN_LENGTH];
  int [] nextCodes = new int[MAX_HUFFMAN_LENGTH];
  int [] huffmanCodes = new int[huffmanCodeLengthDataList.size()];

  for (HuffmanCodeLengthData huffmanCodeLengthData : huffmanCodeLengthDataList) {
    codeLengthOccurrences[huffmanCodeLengthData.bitsNumber]++;
  }
  codeLengthOccurrences[0] = 0;
```

W kolejnym kroku dla każdego rekordu występowania długości obliczany jest unikatowy kod, który będzie w następnym kroku użyty do określenia kodów Huffmana. Kod wygląda następująco.

Listing 3.7. Początek metody generateDynamicsHuffmanLengthCodes

```
for (int i = 1; i < MAX_HUFFMAN_LENGTH; i++) {
  nextCodes[i] = 2 * (nextCodes[i   1] + codeLengthOccurrences[i   1]);
}
```

W ostatnim kroku dopasowywane są odpowiednie kody Huffmana dla danych na początku metody długości. Odpowiada za to poniższy kod.

Listing 3.8. Początek metody generateDynamicsHuffmanLengthCodes

```
for (int i = 0; i < huffmanCodes.length; i++) {
  if (huffmanCodeLengthDataList.get(i).bitsNumber == 0)
    continue;
  huffmanCodes[i] = nextCodes[huffmanCodeLengthDataList.get(i).bitsNumber];
  nextCodes[huffmanCodeLengthDataList.get(i).bitsNumber]++;
}

for (int i = 0; i < huffmanCodes.length; i++) {
  if (huffmanCodeLengthDataList.get(i).bitsNumber == 0)
    continue;
  huffmanCodeLengthDataList.get(i).huffmanCode = huffmanCodes[i];
```

```
  huffmanCodeLengthDataList.get(i).bitsNumber = max(getBitsNumber(huffmanCodes[i]),
      huffmanCodeLengthDataList.get(i).bitsNumber);
}
```

Część kodu odpowiadająca za kompresje ma znacznie więcej metod. Z najistotniejszych jest jedna odpowiadająca za znalezienie powtarzających się ciągów bajtów oraz wiele metod odpowiadających za kompresję statystyczną poszczególnych symboli. W przypadku kompresji statystycznej na końcu przetwarzania każdego bloku danych obliczana jest liczba występowań każdego symbolu, następnie na podstawie tego przypisywane są długości ich kodów, a na końcu generowane są kody Huffmana. Do wygenerowania kodów wykorzystywana jest metoda opisana powyżej „generateDynamicsHuffmanLengthCodes".

Kompresja słownikowa (znalezienie powtarzających się ciągów bajtów) jest wykonywana na początku metody kompresującej. Działa ona na zasadzie wygenerowania tablicy haszującej dla każdych kolejnych trzech symboli początkowych danych. Do wygenerowania tablicy haszującej zastosowano HashMultimap dostarczoną od Google [20]. Kolejne indeksy w tablicy wskazują na pozycje w kompresowanym ciągu. Następnie wraz z iterowaniem po kolejnych powtarzających się symbolach porównywane są symbole i zapisywane jest jedynie początkowa pozycja kopiowania i liczba znaków do przepisania. Kod odpowiedzialny za kompresję słownikową przedstawiony jest poniżej. Dla przejrzystości na poniższym listingu zostały usunięte inicjacje niektórych zmiennych.

Listing 3.9. Fragment metody performLZ77compression

```
private List<LZ77Output> performLZ77compression(byte[] content) {
  ...

  Multimap<Integer, Integer> hashDictionary = HashMultimap.create();

  for (int i = 0; i < content.length; i++) {
    if (i >= 5 && i + 2 < content.length) {
      int key = (int)content[i] * 100000 + (int)content[i + 1] * 1000 + (int)content[
          i + 2];
      int maxMatchedElements = 0;
      int indexOfMatchedSubstring = 0;
      if (hashDictionary.containsKey(key)) {
        RepeatedElementFinder repeatedElementFinder = new RepeatedElementFinder(
        content, hashDictionary, i, key, maxMatchedElements, indexOfMatchedSubstring)
            .invoke();
        maxMatchedElements = repeatedElementFinder.getMaxMatchedElements();
        indexOfMatchedSubstring = repeatedElementFinder.getIndexOfMatchedSubstring();

        LengthCode lengthCode = codeTreesRepresener.findLengthCodeByLength(
            maxMatchedElements);
```

```
        DistanceCode distanceCode = codeTreesRepresener.findDistanceCode(i
            indexOfMatchedSubstring);
        if (lengthCode != null && distanceCode != null) {
          compressedContent.add(new LZ77Output(lengthCode, (...));
          compressedContent.add(new LZ77Output(distanceCode, (...));
        } else { maxMatchedElements = 0;}
    }

    i += maxMatchedElements;
    ...
    if (i >= 2) {
     int key = (int)content[i   2] * 100000 + (int)content[i   1] * 1000 + (int)
         content[i];
     hashDictionary.put(key, i   2);
    }
    compressedContent.add(new LZ77Output(((int) content[i]) & 0x0ff, 0, 0));
  }
  compressedContent.add(new LZ77Output(END_OF_BLOCK, 0, 0));
  return compressedContent;
}
```

W klasie odpowiadającej za szukanie powtarzających się ciągów symboli dokonano ograniczenia do porównywania maksymalnie 1000 ciągów z takim samym początkowym haszem. Pomimo tego, że tablice haszujące mają dostęp z czasem jednostkowym, to przeglądanie wielu elementów zapisanych pod jednym indeksem przetwarzane jest w sposób liniowy. Na podstawie eksperymentów została dobrana wartość 1000 jako dobry kompromis pomiędzy czasem działania programu, a jakością kompresji.

3.4. Proces tworzenia aplikacji - lista zmian w programie

Do napisania programu wykorzystano narzędzie do kontroli wersji git. Proces tworzenia aplikacji został przedstawiony w poniższej tabeli w kolejności od najnowszej zmiany do najstarszej.

Tabela 3.2. Proces tworzenia programu

[ZIP] Run compression in parallel
[DEFLATE][ZIP] Added additional cmd arguments
[DEFLATE] Slightly improved compression quality
[DEFLATE] Added possibility to compress using both dynamics Huffman codes and LZ77 compression
[DEFLATE] Refactored LZ77 compression method
[DEFLATE] Added LZ77 compression
[ZIP] Created CompressionOutput class

Tabela 3.3. Proces tworzenia programu

[OTHER] Added mvn dependency which allows to generate executable jar file Dziedzic
[ZIP] Reduced RAM memory requirements
[DEFLATE] Fixed bug in detecting last block of data
[DEFLATE] Improved Huffman code matching
[DEFLATE] Compress repeated zero codes in Huffman tree representation
[DEFLATE] Added basic compression with dynamics Huffman codes
[ZIP] Refactored code - extracted addCentralDirectoryHeaders method
[ZIP] Fixed bug in central directory zip header
[ZIP] Added central directory information to zip header
[ZIP] Added end of central directory information to zip header
[ZIP] Added PKZIP version to zip header
[ZIP] Added possibility to save compressed file to disk
[ZIP] Added additional arguments to main. Refactored ZIP module
[DEFLATE] Fixed methods responsible for writing bits to output
[ZIP] Run decompression in parallel
[DEFLATE] Significantly improved decompression performance - optimize getting Bits method and removed printing decompression progress
[DEFLATE] Improved performance - reduced number of iterations in getDistance loop
[DEFLATE] Improved performance - reduced number of getBitsLittleEndian method occurrence
[DEFLATE] Improved performance - use only necessary bytes for getting little endian bits
[DEFLATE] Refactored code - removed warnings
[DEFLATE] Fixed bug in reading block without compression
[DEFLATE] Fixed bug in copying multiple bytes to output
[DEFLATE] Add support for uncompressed block
[DEFLATE] Significantly improved decompression performance - use only necessary content for getting bits
[DEFLATE] Improved decompression performance - reduced number of read bits operations
[DEFLATE] improved decompression performance - copy multiple bytes to output
[DEFLATE] Improved decompression performance
[DEFLATE] Improved decompression efficiency
[ZIP] Add possibility to specify file from command line
Add MANIFEST.MF to jar file
[ZIP] Detect central directory signature in zip file
[ZIP] Added support for empty files in zip
[DEFLATE] Fixed bugs in decompressing method
[DEFLATE] Fixed writing bits methods and added unit tests for it
[DEFLATE] Added possibility to write bits to output stream
[DEFLATE] Created template to compress data
[DEFLATE] Added possibility to decompress files using dynamics Huffman codes
[DEFLATE] Generate dynamics Huffman codes for the literal and distance alphabets
[DEFLATE] Read lengths of the codes for the literal and distance alphabets
[DEFLATE] Read first part of dynamics Huffman codes representation

Tabela 3.4. Proces tworzenia programu

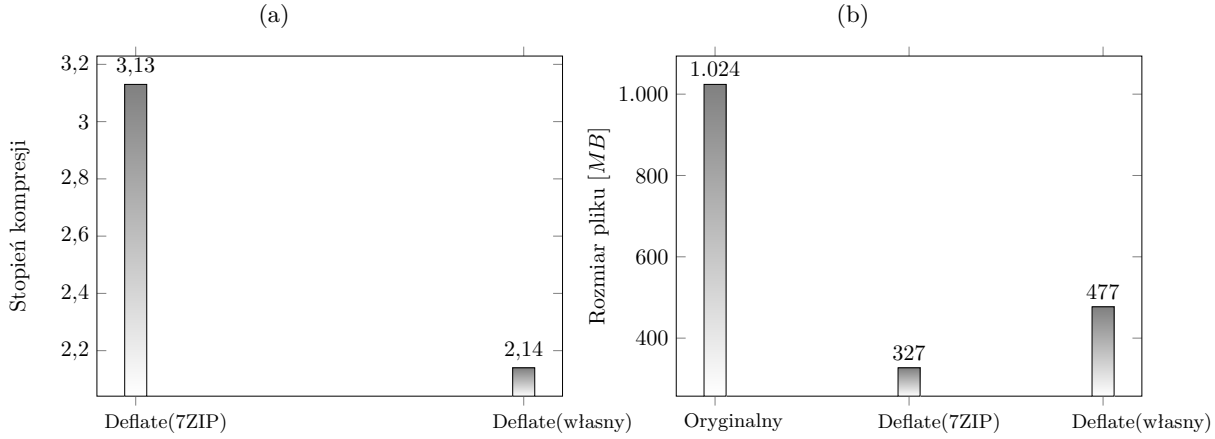
[ZIP] Added additional logging
[ZIP] Create folder with decompressed files
[DEFLATE] Added possibility to read multiple blocks from one file
[DEFLATE] Extracted END_OF_BLOCK variable
[DEFLATE] Added possibility to read content from block compressed fixed Huffman codes
[DEFLATE] Added possibility to read code representation of deflated blocks
[DEFLATE] Added bit reader
[DEFLATE] Added the smallest Huffman length to code trees representation
[DEFLATE] Added generator of code trees representation
[DEFLATE] Created classes for distance length codes
[DEFLATE] Added possibility to read deflate block header
[DEFLATE] Created class for deflate algorithm
[ZIP] Added possibility to get headers form all files in zip
[ZIP] Extracted header methods to separate class
[ZIP] Added filename and file size detection
Added gitignore file
[ZIP] Added checksum detection
[ZIP] Fixed bits endians in Flags enum
[ZIP] Fixed bits endians in Compression Method enum
[ZIP] Added modification date detection
[ZIP] Added compression method detection
Created class for compression algorithm
Created project

3.5. Badanie jakości algorytmów

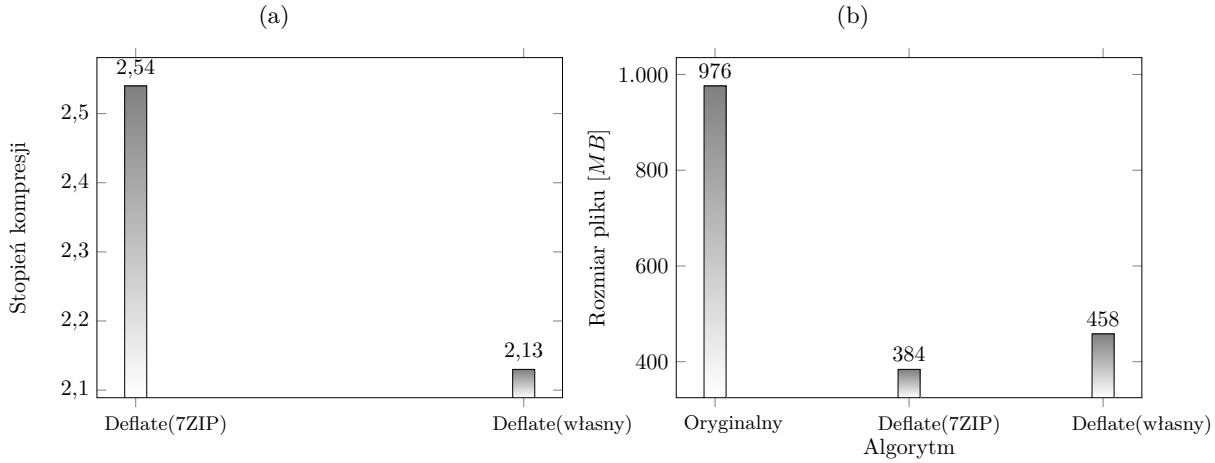
3.5.1. Jakość uzyskanej kompresji

Jednym z celów pracy magisterskiej było zbliżenie jakości napisanego programu do jakości dostępnych programów. Do sprawdzenia wyników kompresji przeprowadzono testy na takich samych danych, jak w rozdziale „Wyniki pomiaru czasu i stopnia kompresji wybranych algorytmów i programów kompresujących”. Poniżej znajduje się porównanie stopnia kompresji algorytmu Deflate z programu 7ZIP oraz własnego.

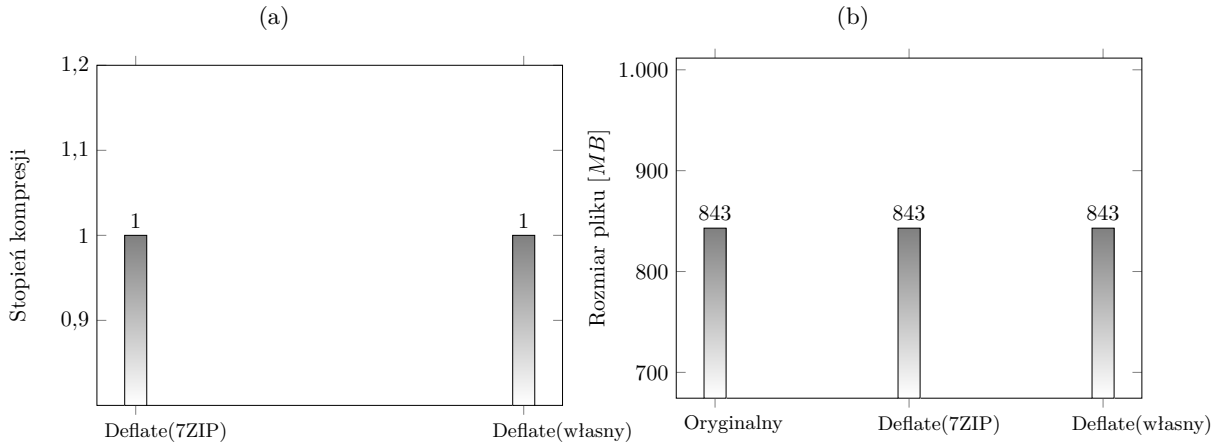
Rys. 3.2. Porównanie stopnia kompresji algorytmu Deflate dla programu 7ZIP i własnego - pliki tekstowe



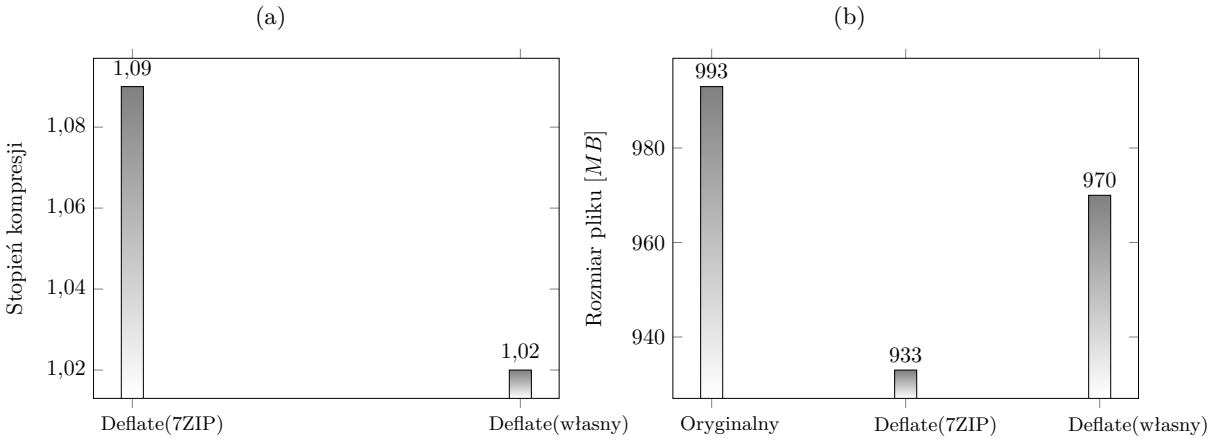
Rys. 3.3. Porównanie stopnia kompresji algorytmu Deflate dla programu 7ZIP i własnego - liczby pseudolosowe



Rys. 3.4. Porównanie stopnia kompresji algorytmu Deflate dla programu 7ZIP i własnego - pliki binarne



Rys. 3.5. Porównanie stopnia kompresji algorytmu Deflate dla programu 7ZIP i własnego - surowe pliki graficzne

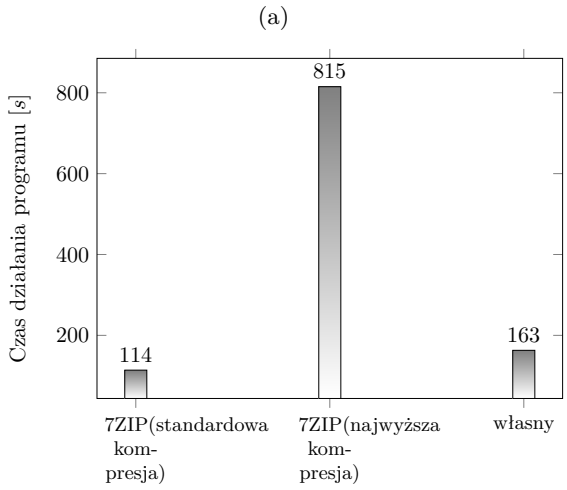


Dla testowanego pliku tekstowego uzyskano stopień kompresji wynoszący około 68% stopnia z programu 7ZIP. Dużo lepszy rezultat uzyskano dla liczb pseudolosowych. Udało się skompresować plik do 47% początkowej wartości, co sprawia, że stopień kompresji wyniósł około 84% stopnia kompresji uzyskanego za pomocą programu 7ZIP. Dla surowych plików graficznych oba programy nie uzyskały zbyt wysokiej kompresji, jednak wartość procentowa stopnia kompresji własnego programu i 7ZIP jest dobra, gdyż udało się uzyskać 93,6% programu 7ZIP. W przypadku obu programów dla plików binarnych nie uzyskano żadnej kompresji.

3.5.2. Czas uzyskanej kompresji

Drugim badanym parametrem był czas uzyskanej kompresji. Na każdym z poniższych wykresów będą się znajdowały 3 pomiary. Pierwszy będzie zawierał pomiar czasu programu 7ZIP dla algorytmu Deflate z ustawioną standardową kompresją, drugi będzie dotyczył tego samego programu i algorytmu, ale z ustawioną najwyższą kompresją, trzeci będzie dotyczył czasu działania własnego programu.

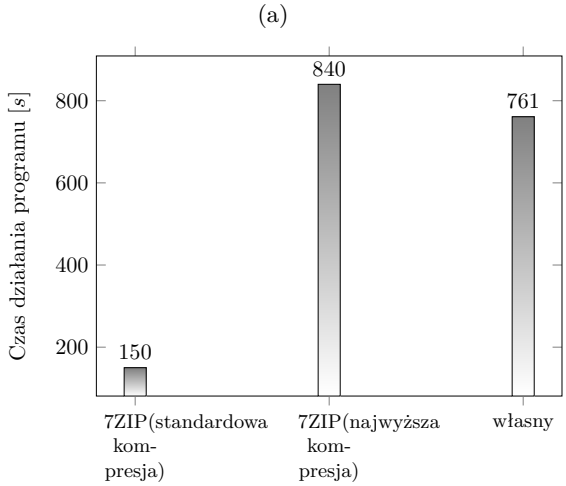
Rys. 3.6. Porównanie szybkości kompresji algorytmu Deflate dla programu 7ZIP i własnego - pliki tekstowe



Dla plików tekstowych udało się uzyskać w przybliżeniu 70% szybkości programu 7ZIP z

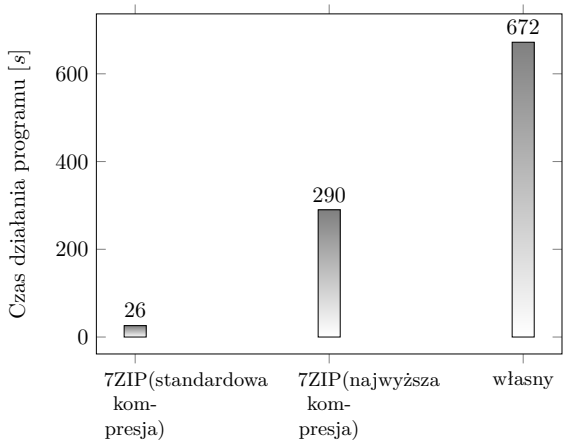
ustawioną standardową kompresją. W tym przypadku w kompresowanym folderze znajdowało się kilka plików tekstowych, które program kompresował za pomocą wielu wątków.

Rys. 3.7. Porównanie szybkości kompresji algorytmu Deflate dla programu 7ZIP i własnego - liczby pseudolosowe



Dla liczb pseudolosowych uzyskano dużo gorszą kompresję w porównaniu do zwykłego tekstu. Wynikało to z tego, że program wykorzystuje wielowątkowość do kompresji poszczególnych plików zamiast bloków danych. Eksperyment został ponowiony dla wielu plików z liczbami pseudolosowymi. W tym celu podzielono plik na szesnaście równych części. W ten sposób udało się uzyskać czas równy 218 sekund, a stanowi to około 69% szybkości programu 7ZIP z ustawioną standardową kompresją.

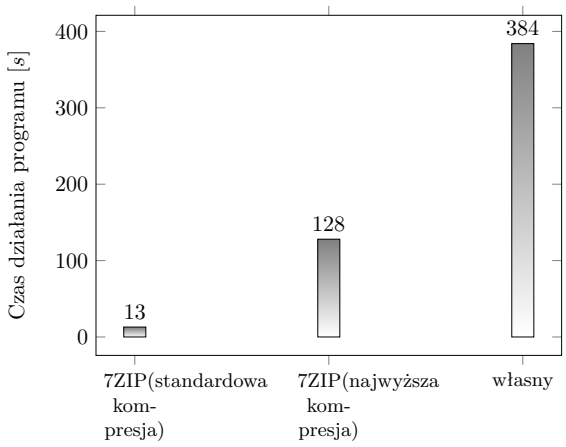
Rys. 3.8. Porównanie szybkości kompresji algorytmu Deflate dla programu 7ZIP i własnego - pliki binarne



Dla plików binarnych uzyskano czas działania bez porównania gorszy. Wynika to z tego, że program 7ZIP dla większości bloków nie próbował kompresować danych. Własna implementacja za każdym razem, niezależnie od rodzaju danych, próbuje tego dokonać.

Rys. 3.9. Porównanie szybkości kompresji algorytmu Deflate dla programu 7ZIP i własnego - surowe pliki graficzne

(a)



W tym wypadku również własny program poradził sobie dużo gorzej od programu 7ZIP. Tutaj także, podobnie jak w poprzednim przykładzie, 7ZIP pozostawił wiele bloków bez kompresji, stąd czas przetwarzania pliku był dużo niższy.

3.5.3. Analiza wyników

Implementowany program ma bardzo zróżnicowane wyniki dla różnych typów danych w stosunku do programu 7ZIP. Tak duże różnice mogą wynikać przede wszystkim z różnic doboru kodów Huffmana. We własnej implementacji minimalna liczba bitów potrzebna do zapisania kodów alfabetu/długości jest zależna od liczby tych elementów. Dokładna minimalna wartość wynosi:

$$\text{minLiczbaBitów} = \lfloor \sqrt{\text{liczbaElemntów}} \rfloor$$

Maksymalna liczba bitów wynosi:

$$\text{minLiczbaBitów} = \lfloor \sqrt{\text{liczbaElemntów}} \rfloor + 1$$

W przypadku pliku z liczbami pseudolosowymi liczba różnych elementów wynosi 13. Znaki zapisane w pliku to 10 cyfr, spacja oraz podwójny znak nowej linii. Dla 13 elementów algorytm zapisze trzy elementy za pomocą trzech bitów oraz dziesięć elementów za pomocą czterech bitów. Przy założeniu, że będzie równomierny rozkład liczb, to za pomocą samych kodów Huffmana program powinien osiągnąć kompresję na poziomie 41% - 42%. Program jednak uzyskał minimalnie gorszą kompresję dla liczb pseudolosowych wynoszącą 47%. Gorszy wynik jest spowodowany tym, że w niektórych blokach algorytm znalazł powtarzające się sekwencje, a tym samym zwiększyła się liczba elementów w tablicy kodów alfabetu/długości. W takiej sytuacji minimalna długość kodów Huffmana w niektórych blokach wyniosła 4 bity. Ręcznie modyfikując kod źródłowy programu można wyłączyć kodowanie słownikowe. Podczas przeprowadzenia takich testów udało się osiągnąć kompresję dla liczb pseudolosowych 42%.

Gorsza kompresja w przypadku tekstu oraz plików graficznych jest spowodowana tym,

że liczba elementów w słowniku jest dużo większa niż w przypadku liczb pseudolosowych. Przykładowo, gdy kompresowany plik ma 128 różnych symboli, kompresja za pomocą kodów Huffmana wyniesie maksymalnie 87,5%.

3.5.4. Możliwe ulepszenia

Znaczącą poprawę jakości kompresji można uzyskać modyfikując metodę doboru dynamicznych kodów Huffmana. Aktualna implementacja dobiera je wykorzystując dwie różne liczby bitów. Zwiększenie liczby doboru bitów do np. 4 różnych liczb bitów sprawiłoby, że program dużo lepiej radziłby sobie w sytuacjach, w których jest dominacja pojedynczych symboli.

Druga modyfikacja dotyczy szybkości działania programu. Aktualnie wątki wykorzystywane są do kompresji poszczególnych plików. W przypadku, gdy jest tylko jeden plik do skompresowania, program będzie wykonywał większość kodu w sposób jednowątkowy. Zmiana logiki w taki sposób, żeby wątki były przypisane do kompresji poszczególnych bloków danych zamiast plików rozwiązałaby problem.

WYKAZ LITERATURY

[1] Apoorv Gupta and Aman Bansal and Vidhi Khanduja. *Modern Lossless Compression Techniques: Review, Comparison and Analysis*. Spraw. tech. https://www.researchgate.net/publication/317599235_Modern_Lossless_Compression_Techniques_Review_Comparison_and_Analysis.

[2] Augusto Horita and Ricardo Bonna and Denis S. Loubach and Ingo Sander. *Lempel-Ziv-Markov Chain Algorithm Modeling using Models of Computation and ForSyDe*. Spraw. tech. https://www.researchgate.net/publication/336795137_Lempel-Ziv-Markov_Chain_Algorithm_Modeling_using_Models_of_Computation_and_ForSyDe.

[3] Deutsch Peter. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. 1996. https://www.ietf.org/rfc/rfc1951.txt.

[4] Deutsch Peter. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950. 1996. https://www.ietf.org/rfc/rfc1950.txt.

[5] E.Jebamalar Leavline and D.Asir Antony Gnana Singh. *Hardware Implementation of LZMA Data Compression Algorithm*. Spraw. tech. https://www.researchgate.net/publication/275038202_Hardware_Implementation_of_LZMA_Data_Compression_Algorithm.

[6] Salomon David. *Data Compression The Complete Reference*. Fourth Edition. wydawca: Springer, grud. 2006. ISBN: 978-1-84-628602-5.

[7] Sayood Khalid. *Introduction to data compression*. Fifth Edition. wydawca: Morgan Kaufmann, paź. 2017. ISBN: 978-0-12-809474-7.

[8] Suman M. Choudhary and Anjali S. Patel and Sonal J. Parmar. *Study of LZ77 and LZ78 Data Compression Techniques*. ISO 9001. http://www.ijesit.com/Volume%204/Issue%203/IJESIT201503_06.pdf.

WYKAZ STRON INTERNETOWYCH

[9] Transformata Burrowsa-Wheelera. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Burrows%E2%80%93Wheeler_transform.

[10] BZip2. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Bzip2.

[11] Deflate. *odczyt z dnia 25.09.2021*. https://en.wikipedia.org/wiki/DEFLATE.

[12] Delta encoding. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Delta_encoding.

[13] Lossless compression. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Lossless_compression.

[14] Lossy compression. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Lossy_compression.

[15] Algorytm MP3. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/MP3.

[16] Model psychoakustyczny. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Psychoacoustics.

[17] Rysunek Postrzeganie dźwięku przez człowieka. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Psychoacoustics#/media/File:Perceived_Human_Hearing.svg.

[18] Range encoding. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Range_encoding.

[19] Run-length encoding. odczyt z dnia 25.09.2021. https://en.wikipedia.org/wiki/Run-length_encoding.

[20] Klasa HashMultimap. Klasa HashMultimap - odczyt z dnia 25.09.2021. https://guava.dev/releases/19.0/api/docs/com/google/common/collect/HashMultimap.html.

[21] Binarne drzewo poszukiwań. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwa%C5%84.

[22] Algorytm JPEG. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/JPEG.

[23] Kodowanie Huffmana. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/Kodowanie_Huffmana.

[24] Kodowanie słownikowe. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/Kodowanie_s%C5%82ownikowe.

[25] Kompresja bezstratna. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/Kompresja_bezstratna.

[26] Kompresja stratna. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/Kompresja_stratna.

[27] Move To Front. odczyt z dnia 25.09.2021. https://pl.wikipedia.org/wiki/Move_To_Front.

[28] Wikipedia data parser. odczyt z dnia 25.09.2021. https://pypi.org/project/wikipedia/.

[29] Frekwencja liter w polskich tekstach. odczyt z dnia 25.09.2021. https://sjp.pwn.pl/poradnia/haslo/frekwencja-liter-w-polskich-tekstach;7072.html.

[30] Why does speed matter?, Bojan Pavic and Chris Anstey and Jeremy Wagner. odczyt z dnia 25.09.2021. https://web.dev/why-speed-matters/.

[31] Transformata Burrowsa-Wheelera. odczyt z dnia 25.09.2021. https://www.geeksforgeeks.org/burrows-wheeler-data-transform-algorithm/.

[32] How Much Faster Is Java 11. odczyt z dnia 25.09.2021. https://www.optaplanner.org/blog/2019/01/17/HowMuchFasterIsJava11.html.

[33] How Much Faster Is Java 15. odczyt z dnia 25.09.2021. https://www.optaplanner.org/blog/2021/01/26/HowMuchFasterIsJava15.html.

WYKAZ RYSUNKÓW

WYKAZ TABEL