

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria Systemów Informatycznych

Kodek obrazu wykorzystujący Asymetryczny System Numeryczny

Artur Tkaczyk

Numer albumu 269356

promotor

dr hab. inż. Grzegorz Pastuszak, prof. PW

WARSZAWA 2020

KODEK OBRAZU WYKORZYSTUJĄCY ASYMETRYCZNY SYSTEM NUMERYCZNY

Streszczenie

Niniejsza praca dyplomowa omawia temat modyfikacji kodera i dekodera entropijnego w implementacji kodeka obrazu opartego na standardzie JPEG 2000. W ramach pracy podmienione zostało kodowanie arytmetyczne wykorzystywane w oryginalnej implementacji na asymetryczny system numeryczny (ANS). Na początku omówiono podstawy teoretyczne dotyczące kompresji oraz kodowania entropijnego. Następną częścią jest przedstawienie podstawowych algorytmów kodowania entropijnego, wraz z wytłumaczeniem ich działania na przykładach. Później przyjdzie czas na omówienie ogólnej koncepcji standardu JPEG 2000. Po części teoretycznej przedstawione zostały wszystkie modyfikacje dokonane na oryginalnej implementacji wraz z fragmentami kodu źródłowego. Na końcu przetestowano poprawność nowego kodera i dekodera oraz porównane było ich działanie z oryginalną wersją.

Słowa kluczowe: JPEG 2000, kodowanie entropijne, kompresja obrazu, ANS, kodowanie arytmetyczne, CABAC, kompresja stratna

IMAGE CODEC USING ASYMMETRIC NUMERAL SYSTEMS

Abstract

This thesis discusses the modification of entropy coder and decoder in the initial implementation of the image codec based on JPEG 2000 standard. In this work, the arithmetic coder used in original implementation was replaced with an asymmetric numeral system (ANS). At the beginning, there is an overview of the basics of the compression and entropy coding. Next part focuses on basic entropy coding algorithms together with the examples. Subsequently, the general concept of JPEG 2000 standard will be explained. After the introduction part, a description of all the changes performed on the original implementation coupled with the fragments of the source code is provided. Ultimately, the correctness of the modified program is tested and comparison of its performance with the original one is shown.

Keywords: JPEG 2000, entropy coding, image compression, ANS, arithmetic coding, CABAC, lossy compression



Politechnika Warszawska
Warsaw University of Technology

załącznik do zarządzenia nr 28/2016r. Rektora PW

Załącznik nr 3 do zarządzenia nr 24/2016
Rektora PW

Warszawa, 03.09.2020

.....
miejscowość i data

Artur Tkaczyk

.....
imię i nazwisko studenta
269356

.....
numer albumu
Informatyka

.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płytcie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

<u>Wstęp.....</u>	9
<u>1.1 Temat pracy dyplomowej.....</u>	9
<u>1.2 Zawartość pracy dyplomowej.....</u>	10
<u>Podstawy teoretyczne kompresji danych.....</u>	11
<u>Metody kodowania entropijnego.....</u>	16
<u>3.1. Kodowanie Huffmana.....</u>	16
<u>3.2. Kodowanie arytmetyczne.....</u>	20
<u>3.3. Asymetryczny system numeryczny (ANS).....</u>	26
<u>Standard kompresji obrazów JPEG 2000.....</u>	32
<u>4.1. Ogólny schemat standardu.....</u>	32
<u>4.2. Dyskretna transformata falkowa.....</u>	34
<u>4.3. Modelowanie kontekstowe.....</u>	35
<u>4.4. Koder entropijny.....</u>	36
<u>Implementacja.....</u>	39
<u>Testowanie.....</u>	56
<u>Podsumowanie.....</u>	61
<u>Bibliografia.....</u>	62
<u>Spis rysunków.....</u>	63
<u>Spis tabel.....</u>	64

Rozdział 1

Wstęp

1.1. Temat pracy dyplomowej

Celem pracy dyplomowej jest zastąpienie kodowania arytmetycznego w implementacji referencyjnej JJ2000 kodera i dekodera zgodnych ze standardem JPEG 2000 na asymetryczne kodowanie numeryczne. ANS jest nowym sposobem kodowania o podobnym stopniu kompresji co kodowanie arytmetyczne, ale za to zapewniające szybsze dekodowanie znaków, zapewniając tym samym alternatywę dla kodowania arytmetycznego (zapewniającego dobry stopień kompresji) i kodowania Huffmana (szybkie kodowanie i dekodowanie). W standardzie JPEG 2000 zastosowane zostało kodowanie CABAC, a więc kontekstowo-adaptacyjne kodowanie arytmetyczne. W tym algorytmie przetwarzanie dokonywane jest na poszczególnych bitach, które posiadają towarzyszące im konteksty. Żeby dostosować nową implementację kodera do pozostałej części implementacji JJ2000, skorzystano więc z kodowania numerycznego w wersji binarnej, zamiast kodowania rANS, które jest w stanie kodować znaki z większego zbioru. W celu usprawnienia działania zarówno kodera, jak i dekodera zastosowano tANS, a więc kodowanie numeryczne, w którym wyniki działań umieszczane są w tablicy, dzięki temu można ominąć cały proces obliczania wartości podczas przetwarzania, zamiast tego pobrane będą wartości z określonych indeksów. Ponadto wykorzystano również zwracane przez modelowanie kontekstowe wartości kontekstów, w celu obliczenia odpowiedniego modelu prawdopodobieństw dla każdej z nich. Modyfikacji dokonano tylko w plikach odpowiedzialnych za kodowanie binarne, a więc poprzedni zbiór funkcji publicznych nie uległ zmianie, ich działanie dostosowano natomiast do nowej implementacji.

Przed dokonaniem odpowiednich zmian, wymagane było pozyskanie odpowiedniej wiedzy niezbędnej do zrozumienia omawianego zagadnienia i wykonania implementacji. W pierwszej kolejności należało poznać standard JPEG 2000, ze szczególnym uwzględnieniem procesu kodowania entropijnego stanowiącego ostatni etap kompresji obrazu. W celu zrozumienia tego procesu trzeba również poznać zastosowaną w standardzie metodę kodowania arytmetycznego oraz jej odmianę w wersji kontekstowo-adaptacyjnej. Standard JPEG 2000 posiada wiele podobieństw do popularnego i szeroko stosowanego standardu JPEG. Tak więc przydatne jest poprzedzenie nauki JPEG 2000, wcześniejszą lekturą standardu JPEG.

Następnym etapem było poznanie metody asymetrycznego kodowania numerycznego, która miała zostać zaimplementowana, a także przejrzenie implementacji referencyjnej, która to miała podlegać modyfikacji. W modyfikowanych klasach należało zachować poprzednie metody publiczne, tak żeby modyfikacje ograniczały się do samych klas kodera i dekodera, tak więc w celu zapewnienia poprawności całego oprogramowania wymagane było poznanie ich znaczenia w programie, co okazało się niezbędne podczas samej implementacji.

Po części teoretycznej omówiona została najważniejsza część pracy inżynierskiej, czyli implementacja kodeka. Po jej skończeniu przeprowadzono serię testów w celu określenia, czy otrzymano oczekiwany rezultat. W pierwszym etapie testów sprawdzana była poprawność działania, poprzez skompresowanie zmodyfikowanym programem pierwotnego obrazu, a następnie zdekompresowanie do pierwotnego formatu i porównanie obu obrazów. Test ten był przeprowadzony na kilku obrazach i różnych akceptowalnych przez koder formatach obrazu. W drugim etapie porównano rozmiar w bajtach skompresowanych plików dla oryginalnego i zmodyfikowanego programu, by określić, czy stopień kompresji uległ znaczącej zmianie. Ostatni test sprawdza wydajność nowej implementacji.

1.2. Zawartość pracy dyplomowej

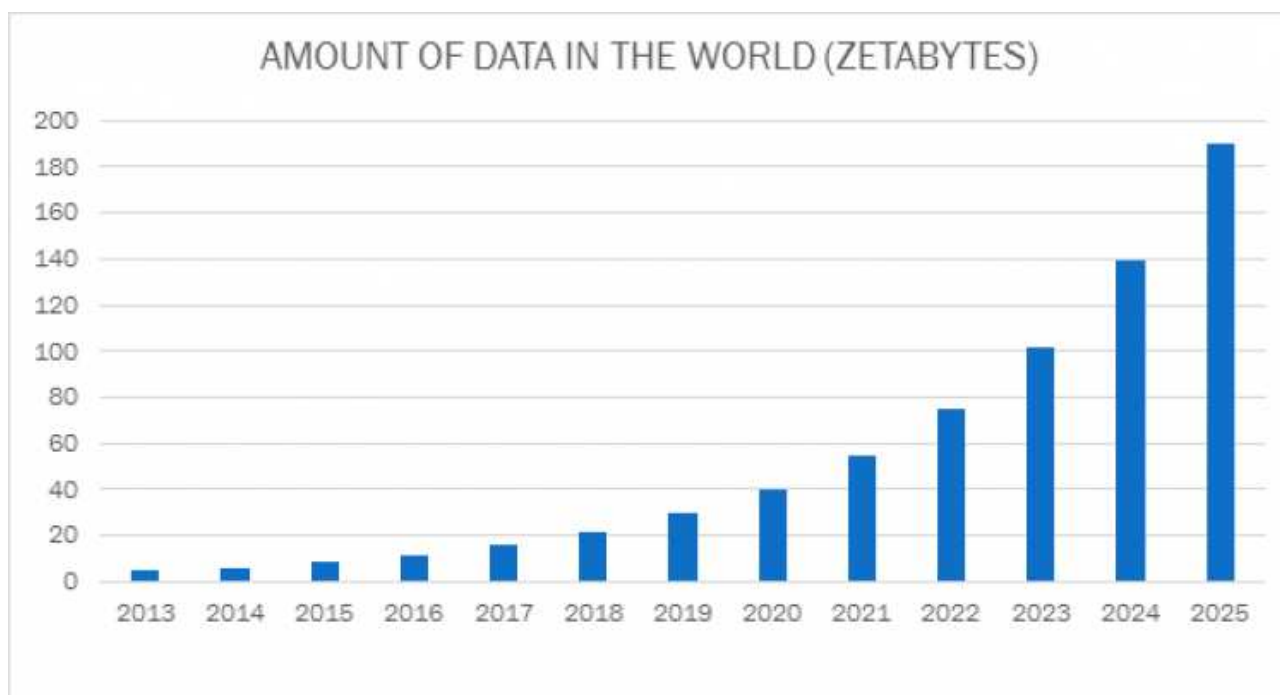
Praca podzielona jest na rozdziały:

- Rozdział 2: Podstawy teoretyczne kompresji danych – Ten rozdział omawia, czym jest kompresja i jakie są jej typy, czym jest entropia i jak się ją oblicza oraz czym jest kodowanie entropijne.
- Rozdział 3: Metody kodowania entropijnego – Ten rozdział przedstawia metody kodowania entropijnego: kodowanie Huffmana, kodowanie arytmetyczne, ANS, CABAC.
- Rozdział 4: Standard kompresji obrazów JPEG 2000 – Ten rozdział ogólnikowo omawia standard JPEG 2000, ponadto nieco więcej uwagi skupia na transformacji falkowej (DWT) oraz koderze i dekodерze entropijnym.
- Rozdział 5: Implementacja – Ten rozdział omawia implementację kodera i dekodera ANS oraz przedstawia dokonane zmiany w postaci fragmentów kodu źródłowego.
- Rozdział 6: Testowanie – Ten rozdział omawia wszystkie przeprowadzone testy, mające na celu ocenienie poprawności, stopnia kompresji i szybkości nowej implementacji oraz przedstawia wyniki wraz z ostatecznymi wnioskami.

Rozdział 2

Podstawy teoretyczne kompresji danych

Szybki rozwój technologiczny pociąga za sobą ciągły wzrost rozmiaru danych przechowywanych na świecie i wszelkie badania wskazują, że proces ten będzie tylko przyspieszał. Ponadto wzrosło zapotrzebowanie na szybkie przesyłanie dużych danych na przykład w celu przeprowadzenia transmisji wideo na żywo. Tak więc by spełnić oczekiwania społeczeństwa, konieczne było ulepszenie metod przechowywania, przesyłania, oraz manipulacji ogromną ilością informacji. W celu usprawnienia tych procesów tworzone są pamięci masowe o większej pojemności oraz łącza o większej przepustowości. W szczególności szybkość rozwoju tych pierwszych jest na tyle duża, że problem braku pamięci przestał być tak znaczący, jak jeszcze kilkanaście lat temu. Oprócz tych usprawnień wymagany jest również rozwój metod kompresji danych.



Rys. 2.1: Szacowana ilość informacji przechowywana na świecie [8]

Jednym z kluczowych typów danych we współczesnych systemach informatycznych są obrazy. W większości przypadków, kiedy słyszymy o obrazach, na myśl przychodzi zdjęcia przedstawiające świat rzeczywisty. Pierwsza fotografia powstała w 1826 roku przedstawia widok z okna w Le Gras, a jej wykonanie wymagało co najmniej ośmiu godzin ciągłej ekspozycji w ciągu dnia. Wykonywane i przechowywane były na płytkach pokrytych światłoczułą substancją. W 1884 roku George Eastman ulepszył metodę fotografii, pozwalając na tworzenie zdjęć na papierze, ułatwiając tym samym pracę fotografom i popularyzując fotografię wśród amatorów. W obecnych czasach zdjęcia przechowywane są w

formie cyfrowej, a ich upowszechnienie jest tak duże, że szacuje się całkowitą liczbę zdjęć cyfrowych wykonanych w samym 2015 roku na bilion [9].

Obrazy są specyficznym typem danych, w którym występuje dwuwymiarowa korelacja między wartościami próbek. Ze względu na obiekty przedstawiane na obrazach można wyróżnić kilka typów obrazów, co pozwala na dostosowanie odpowiedniej metody kompresji do każdego z nich. Te typy to:

- obraz naturalny — Obraz będący cyfrową fotografią świata rzeczywistego. W obrazach tego typu najczęściej stosuje się kompresję stratną, ponieważ małe różnice wartości próbek nie są w tym przypadku widoczne dla ludzkiego oka.
- obrazy graficzne — Są to obrazy tworzone w systemach grafiki komputerowej, edytorach tekstu itp.
- obrazy specjalistyczne — Takie jak obrazy medyczne, satelitarne, wyróżniane są, ponieważ mogą one wymagać specjalnych metod kompresji. W przypadku obrazów medycznych stosuje się kompresję bezstratną w celu uniknięcia błędu w diagnozie pacjenta.
- złożone — Obrazy zawierające fragmenty o różnych charakterystykach, mogą zawierać w sobie grafikę oraz obraz naturalny. Do kompresji takich obrazów warto podzielić go na oddzielne fragmenty, które kompresowane są w różny sposób w zależności od potrzeb.

Kompresja danych to proces przekształcania pierwotnej reprezentacji danych w reprezentację o mniejszej liczbie bitów, odwrotny proces nazywany jest dekompresją. Kompresja może być stratna lub bezstratna. W przypadku kompresji bezstratnej nie tracimy żadnych informacji, to znaczy, że po dekompresji, zrekonstruowana sekwencja danych jest taka sama jak pierwotne dane z dokładnością do pojedynczego bitu. Niestety kompresja bezstratna pozwala na kompresję danych tylko do pewnego stopnia. W przypadku kompresji stratnej tracimy pewną część informacji, ale za to jesteśmy w stanie znacznie bardziej skompresować dane. Celem kompresji stratnej jest uzyskanie danych, których reprezentacja (w postaci obrazu lub pliku audio, wideo) jest wystarczającej jakości, a więc różnice między przetworzonym obrazem a oryginałem nie są wyraźnie widoczne dla ludzkiego oka. Kompresja bezstratna zwykle składa się z dwóch etapów: modelowania (etap mający na celu przekształcenie sekwencji wejściowej do odpowiedniej formy pośredniej ułatwiającej dalsze kodowanie) i kodowania binarnego (na podstawie wyników modelowania tworzy wyjściowy ciąg bitów). Kompresja bezstratna wykorzystuje zwykle fakt, że znaki występujące w ciągu danych mogą występować z różnymi prawdopodobieństwami, dzięki czemu możemy zmniejszyć liczbę zapisanych bitów, jeśli znakom o większym prawdopodobieństwie przypiszemy krótsze ciągi bitów. Kompresja stratna wykorzystuje natomiast fakt, że w niektórych typach danych, takich jak obraz, film, dźwięk, część informacji ma wyraźnie mniejszy wpływ na postrzeganie przez człowieka, przez co można tą część danych pominąć, uznając ją za mniej istotną. W przypadku kompresji obrazu ludzkie oko jest mniej czułe na

wysokie częstotliwości obrazu oraz na niewielkie zmiany barwy piksela, co oznacza, że można porzucić część informacji z tym związanych (poprzez większą kwantyzację), nie zmniejszając jednak w sposób wyraźny jakości obrazu [5].

Wiele z popularnych algorytmów i narzędzi do kompresji danych zostało stworzonych do radzenia sobie z określonym typem danych, na przykład JPEG, MPEG, MP3. Biorą one pod uwagę specjalne cechy danego typu do poprawienia efektywności kompresji. Inne narzędzia, takie jak compress, zip oraz pack mogą być używane do przetwarzania dowolnego pliku z danymi.

Do oceny efektywności kompresji stosuje się szereg miar, w tym stopień kompresji CR (ang. compression ratio) oraz średnia bitowa BR (ang. bit rate). Stopień kompresji obliczany jest, dzieląc wielkość danych oryginalnych przez ich wielkość po skompresowaniu. Średnia bitowa jest to średnia liczba bitów przypadająca na jeden element źródłowej sekwencji danych.

Informacją nazywamy wszystko to, co można zużytkować do bardziej sprawnego wyboru działań prowadzących do realizacji określonego celu. Entropia jest miarą niepewności związaną z określoną zmienną losową, w teorii informacji jest też miarą informacji. Entropię zmiennej losowej obliczamy poprzez wzór:

$$H(X) = - \sum_{i=1}^n p_i \log_2 p_i$$

Gdzie:

X – zmienna losowa

p_i – prawdopodobieństwo wystąpienia i – tego znaku zmiennej losowej X

Dzięki temu wzorowi jesteśmy w stanie obliczyć średnią ilość informacji w bitach potrzebną do opisanego pojedynczego znaku w ciągu znaków o danym rozkładzie prawdopodobieństw.

Żeby obliczyć ilość informacji związanej z pojedynczym znakiem, można skorzystać ze wzoru:

$$H = \log_2 N = \log_2 \left(\frac{1}{p_N} \right)$$

Na przykład dla rzutu monetą entropia wynosi 1 bit.


Ze wzorów wynika, że bardziej prawdopodobne znaki mają mniejszą entropię, a więc można zapisać je za pomocą mniejszej liczby bitów. Idealna sytuacja miałaby miejsce, gdy każdemu znakowi odpowiada taka sama liczba bitów co jego entropia, co może być trudne, biorąc pod uwagę, że entropia może przyjmować wartości ułamkowe.

Kodowanie entropijne ma na celu usunięcie nadmiarowości z ciągu danych wejściowych, osiąga się to poprzez przypisanie znakowi, lub ciągowi znaków ciąg bitów, w taki sposób,

żeby wielkość danych była jak najbliższa entropii. Nie można jednak osiągnąć wyniku mniejszego od entropii. Kodowanie entropijne wykorzystywane jest zwykle jako ostatni etap kompresji stratnej.

Kompresja obrazów w celu eliminacji nadmiarowości przestrzennej może zastosować wiele zróżnicowanych metod przekształcenia oryginalnych danych do postaci, która jest bardziej podatna na proces kodowania binarnego. Im większa dysproporcja w częstości występowania znaków w pliku, tym większa nadmiarowość informacji. A więc metody przekształcenia obrazu wykonywane przed samą kompresją mają na celu zwrócić równoważną danemu obrazowi reprezentację, której próbki skupione są wokół określonych symboli (na przykład symboli bliskich 0). Ich przykładami są:

- Metody predykcyjne, w których estymowana wartość piksela obliczana jest na podstawie sąsiadujących z nią pikseli, które zwykle są najbardziej z nią skorelowane. W tym przypadku kodowana jest tylko różnica między obliczoną wartością a wartością faktyczną próbki.
- Transformacja falkowa, która rozkłada obraz na szereg obrazów zawierających tylko częstotliwości w ograniczonych przedziałach.
- Podział pikseli na szereg map bitowych, tworzonych poprzez pobranie z każdej próbki bitów znajdujących się na tych samych pozycjach. Najbardziej znaczące bity próbek mają duży stopień skorelowania.

Obraz	Jakość	Rozmiar obrazu (w bajtach)
	100%	258 519


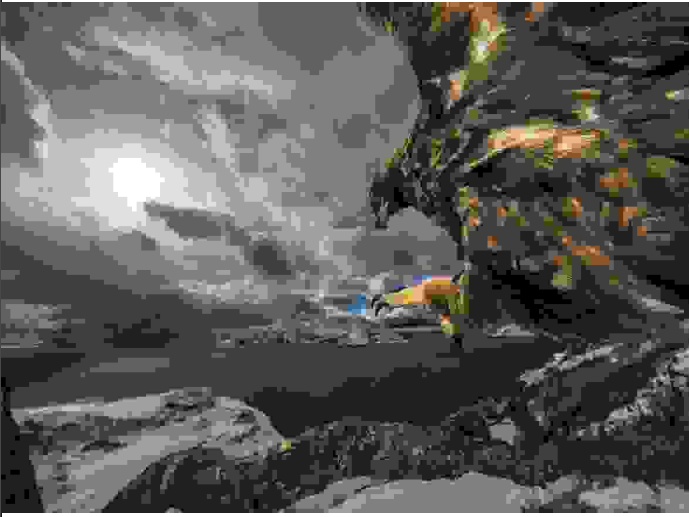
	50%	111 100
	1%	17 690

Tabela 2.1: Porównanie jakości i wielkości obrazu po kompresji przy użyciu standardu JPEG

Rozdział 3

Metody kodowania entropijnego

3.1. Kodowanie Huffmana

Podstawową metodą kodowania binarnego jest przypisanie skończonej grupie znaków z alfabetu wejściowego odpowiednich słów kodowych o zmiennej długości. Zadanie to nie jest jednak takie proste, biorąc pod uwagę, że kody te mają zapewnić jednoznaczne dekodowanie, a ponieważ długość słów kodowych przypisanych do każdego ze znaków jest różna, więc nie można w prosty sposób określić położenia poszczególnych symboli w strumieniu bitowym. Na przykład dla $\{a = 0, b = 1, c = 01\}$ ciąg bitów 011 można zdekodować jako: abb lub cb, dlatego ten kod nie nadaje się do zastosowania przy kodowaniu danych źródłowych. Dlatego wyróżnia się tak zwane kody przedrostkowe, czyli kody, dla których żadna zakodowana wartość znaku nie jest przedrostkiem innej wartości z tego kodu. Kody te mają wyraźną przewagę nad pozostałymi, po pierwsze wyraźnie ułatwiają spełnienie wymagań dotyczących jednoznacznego dekodowania, ale przede wszystkim ułatwiają znacząco implementację dekodera tego kodu. Jest to spowodowane tym, że dekodер może działać „od lewej do prawej”, bez konieczności wnikania w wartość następujących danych po to, by poprawnie zwrócić zdekodowany symbol.

Kodowanie Huffmana jest metodą kodowania entropijnego stworzoną w 1952 roku przez Davida Huffmana i opublikowaną w pracy pod tytułem „A Method for the Construction of Minimum-Redundancy Codes”. Polega ona na znajdowaniu optymalnych kodów przedrostkowych poprzez stworzenie na podstawie grupy znaków i ich wag, drzewa Huffmana przypisującego im optymalne kody. Kodowanie Huffmana cechuje przede wszystkim szybkość działania oraz prostota implementacji. Posiada jednak również sporą wadę, albowiem stopień kompresji zbliża się do entropii tylko dla prawdopodobieństw występowania znaków zbliżonych do potęg dwójki. Spowodowane to jest tym, że zgodnie ze wzorem tylko dla takich prawdopodobieństw, entropia osiąga wartość całkowitą, a z oczywistych względów słowa kodowe przypisane w tym algorytmie nie mogą mieć długości o wartości ułamkowej [2].

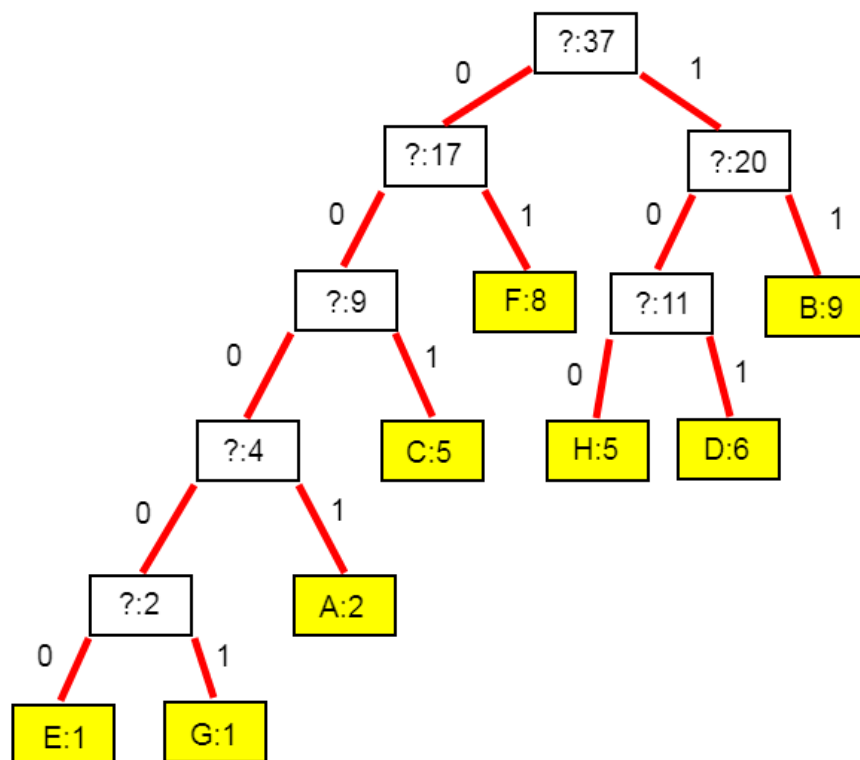
Algorytm

Algorytm na wejście przyjmuje grupę znaków (na przykład litery dla plików tekstowych), a na wyjście zwraca tablicę przypisującą każdemu z nich odpowiednie słowo kodowe. Pierwszym krokiem do tego jest utworzenie dla nich drzewa.

Algorytm tworzenia drzewa Huffmana:

1. Oblicz prawdopodobieństwa występowania poszczególnych znaków.
2. Utwórz listę drzew binarnych, które w węzłach przechowują parę: znak i prawdopodobieństwo.

3. Dodaj do listy drzewa składające się z pojedynczego węzła zawierającego znak i obliczone wcześniej jego prawdopodobieństwo.
4. Pobierz z listy dwa elementy o korzeniach z najmniejszymi prawdopodobieństwami.
5. Dodaj do listy drzewo będące połączeniem obu drzew, w korzeniu nowego drzewa należy wpisać sumę prawdopodobieństw obu drzew.
6. Jeżeli liczba drzew na liście jest różna od jeden, wróć do punktu 4.



Rys. 3.1: Przykładowe drzewo Huffmana

Po stworzeniu drzewa można wyznaczyć odpowiednie słowa kodowe dla każdego znaku, przechodząc po całym drzewie. Zgodnie z umową (nie jest to jednak wymagane do odpowiedniego działania algorytmu, wartości bitów mogą być zamienione) każdemu przejściu do lewego poddrzewa z danego węzła przypisujemy bit 0, natomiast przejście w prawo otrzymuje bit 1. Za każdym razem jak przechodzimy w lewo, dopisujemy do tworzonego kodu 0, natomiast przy przejściu w prawo dopisujemy 1. Tak więc przechodząc po całym drzewie, dopisujemy do aktualnego słowa kodowego znak przypisany do danej krawędzi, a w przypadku natrafienia na liść wpisujemy napotkany tam znak i otrzymane słowo kodowe do tablicy wyjściowej. Stworzenie kodu przy pomocy drzewa automatycznie zapewnia też to, że jest on kodem przedrostkowym, ponadto łatwo zauważyć, że o długości

słowa kodowego decyduje głębokość, na której znajduje się liść w drzewie [6].

Widzimy w tym algorytmie, że na podstawie takiego samego rozkładu prawdopodobieństw można otrzymać zupełnie różne drzewa, a tym samym różne kody. Po pierwsze nie ma żadnej z góry ustalonej reguły określającej, któremu z dwóch wybranych elementów, które stworzą lewe i prawe poddrzewo nowego drzewa binarnego, przypiszemy określone wartości bitu. Ponadto, algorytm nie określa które elementy listy należy wybrać w przypadku gdy ich prawdopodobieństwa są sobie równe. Oznacza to, że dla identycznego przypadku można otrzymać drzewa o różnych głębokościach.

Dekodowanie najlepiej przeprowadzić przy użyciu drzewa, zamiast tablicy ze słowami kodowymi. Proces ten zaczynamy, ustawiając wskaźnik na korzeń drzewa. Następnie kolejno pobieramy kolejne bity z wejścia, w przypadku pobrania bitu 0, przenosimy wskaźnik w lewo od aktualnego węzła. Dla bitu 1 natomiast przenosimy wskaźnik w prawo. Kontynuujemy takie działanie aż do dojścia do liścia drzewa, w którym przechowywany jest zdekodowany znak. Wypisujemy ten znak na wyjście dekodera i przenosimy wskaźnik z powrotem do korzenia drzewa.

Przy dekodowaniu napotkamy jednak na znaczący problem, albowiem zarówno koder, jak i odpowiadający mu dekodery, muszą operować na tych samych drzewach w celu poprawnej dekompresji. A więc to znaczy, że dekodery musi w jakiś sposób zrekonstruować drzewo Huffmana wykorzystane przy kodowaniu. W najprostszym przypadku gdy prawdopodobieństwa dla znaków są podobne dla wszystkich kodowanych znaków (na przykład dla dużych plików tekstowych w tym samym języku częstości występowania znaków powinny być podobne) można ustawić na sztywno jedno drzewo używane dla każdego przypadku. Wiąże się to oczywiście z pewną stratą pod kątem stopnia kompresji danych. W przypadku, gdy nie korzystamy z predefiniowanych drzew, należy przesłać je w jakiś sposób „a priori”. Jednym z prostszych sposobów, aby to zrobić, jest wypisanie słów kodowych na początku pliku. Przykładowo, tablica może być zapisana zgodnie z porządkiem symboli w formie: długość słowa kodowego, słowo kodowe. Innym, nieco lepszym sposobem jest zapisanie na początku pliku samego drzewa Huffmana. W tym przypadku przechodzimy przez drzewo i w przypadku napotkania liścia zwracamy bit 1 oraz wartość znaku znajdującego się w danym liściu, natomiast dla pozostałych węzłów wypisujemy bit 0.

Przedstawiony powyżej algorytm opisuje statyczną, bazową wersję kodowania Huffmana. Powstało wiele modyfikacji stosowanych w zależności od potrzeb. W dynamicznej wersji tej metody, drzewo modyfikowane jest wraz z kolejnymi zakodowanymi znakami ciągu wejściowego, a liście drzewa przechowują liczbę wystąpień, zamiast prawdopodobieństw. Modyfikacja ta ma dwie główne zalety, przede wszystkim nie trzeba obliczać statystyk poszczególnych znaków, co może wymagać przetworzenia całego pliku wejściowego, zamiast tego koder adaptuje się na bieżąco. Drugą zaletą jest to, że rozwiązuje to całkowicie problem przesyłania użytego drzewa Huffmana do dekodera. Znamy bowiem początkową wersję drzewa Huffmana, biorąc pod uwagę to, że wszystkie procesy kompresji zaczynają się z wyzerowanymi statystykami, a tym samym ze zrównoważonym, domyślnym drzewem.

Następnie wszelkie modyfikacje kodu przeprowadzone w koderze mogą zostać zrekonstruowane w identyczny sposób po stronie dekodera, obliczając statystyki zdekodowanych znaków. Wiąże się to jednak ze spadkiem w szybkości działania, co było jedną z głównych zalet kodowania Huffmana.

W celu usprawnienia działania kodowania i przybliżenia długości zakodowanego ciągu do entropii można wprowadzić jeszcze jedną modyfikację, poprzez obliczanie kodów nie dla pojedynczych znaków, ale dla całych grup znaków. Stosowanie tej metody jest jednak ograniczone, ze względu na to, że wielkość pamięci potrzebnej do przechowania tablicy z kodami rośnie wykładniczo wraz z liczbą znaków należących do pojedynczej grupy.

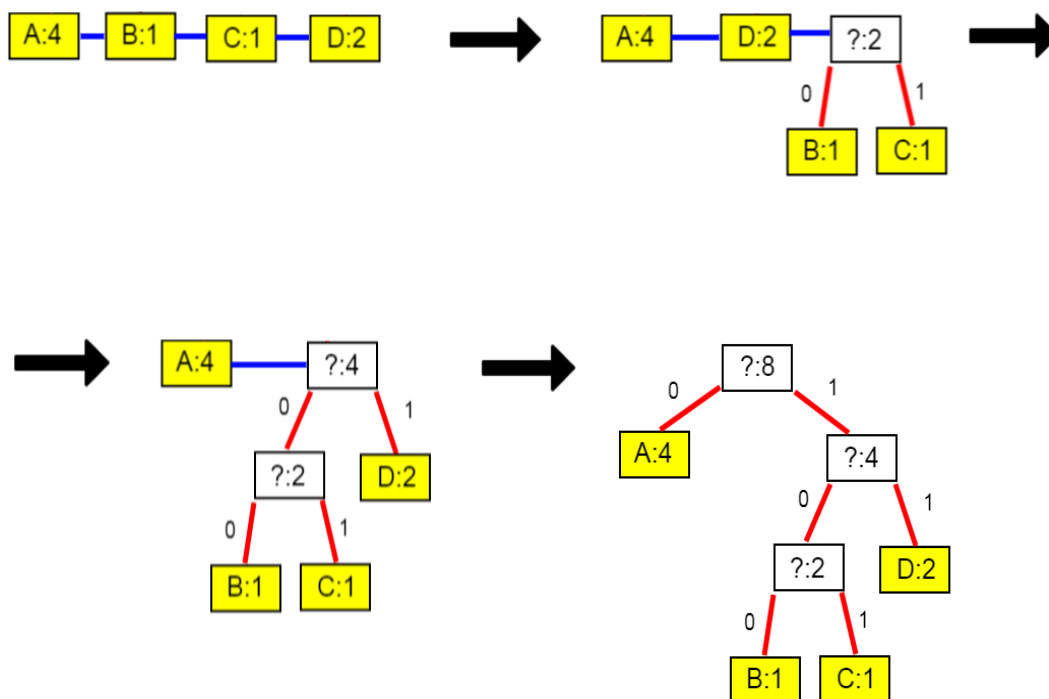
Przykład

Żeby zademonstrować działanie kodowania Huffmana, zakodujmy w nim ciąg: ABAACDAD. Na początku należy obliczyć statystyki występowania wszystkich znaków w kodowanym ciągu.

symbol	wystąpienia
A	4
B	1
C	1
D	2

Tabela 3.1: Wystąpienia znaków dla kodowania Huffmana

Następnie możemy przystąpić do procesu tworzenia drzewa Huffmana, jego etapy pokazane są na rysunku poniżej.



Rys. 3.2: Proces tworzenia drzewa Huffmana dla przykładowego ciągu znaków

symbol	kod
A	0
B	100
C	101
D	11

Tabela 3.2: Kody Huffmana dla każdego znaku

Po skończeniu wyznaczania kodów dla wszystkich kodowanych znaków można przystąpić do kodowania. A więc wejściowy ciąg znaków zamieniamy na ciąg bitów 01000010111011.

W tym przypadku prawdopodobieństwa wystąpienia znaków są potęgami dwójki, a więc wielkość danych jest równy entropii. W przypadku klasycznego kodowania, w którym wszystkie znaki mają słowa kodowe o tej samej długości, każdy znak tutaj zostałby zakodowany za pomocą dwóch bitów, a więc całość miałaby długość 16 bitów. Przy użyciu kodowania Huffmana osiągnięto 14 bitów.

3.2. Kodowanie arytmetyczne

Pomimo niezaprzeczalnych, licznych zalet kodowania Huffmana, nie pozwala on jednak na satysfakcjonujący w wielu przypadkach stopień kompresji. Z celu ulepszenia całego procesu kodowania entropijnego, trzeba było znaleźć sposób na zakodowanie pojedynczego znaku za pomocą ułamków bitów, zamiast ograniczać się tylko liczb całkowitych. Kodowanie arytmetyczne jest metodą kompresji bezstratnej opracowanej około 1960 roku przez Petera Eliasa. Zapewnia on kompresję danych zbliżoną do entropii. Dzieje się to jednak kosztem szybkości działania implementacji, która jest niższa od kodowania Huffmana. Algorytm reprezentuje zakodowany ciąg za pomocą liczby zmiennoprzecinkowej zawierającej się w określonym przedziale [3].

Kolejną zaletą kodowania arytmetycznego jest odseparowanie procesu modelowania od kodowania. W przypadku kompresji obrazu pozwala nam to na zastosowanie różnych sposobów modelowania w zależności od specyficznych lokalnych właściwości obrazu.

Algorytm

Koder arytmetyczny przechowuje aktualny stan za pomocą przedziału, a więc w postaci dwóch wartości oznaczających dolną granicę przedziału oraz jego długość. Stan ten ciągle zmienia się w zależności od nowych znaków wysłanych do zakodowania, ale jego przedział zawsze zawiera się wewnątrz przedziału $[0, 1)$. Wartość dowolnej liczby zawierającej się wewnątrz tego stanu, wraz z informacją o rozkładzie prawdopodobieństwa występowania poszczególnych znaków jest w pełni wystarczająca do przeprowadzenia procesu odwrotnego, a więc poprawnego zdekodowania ciągu symboli. Dekoder podobnie jak koder, również przechowuje stan przedziału, który odpowiednio modyfikuje w trakcie działania całego procesu.

Algorytm kodowania arytmetycznego

1. Ustaw wartość przedziału na wartość początkową wynoszącą $[0, 1)$ oraz oblicz prawdopodobieństwa wszystkich znaków.
2. Podziel przedział na podprzedziały przypisane do każdego ze znaków wejściowych o długościach będących iloczynem długości aktualnego przedziału oraz prawdopodobieństwa tego znaku.
3. Pobierz kolejny znak z wejścia kodera, jeśli nie ma już znaków do zakodowania, przejdź do punktu 5.
4. Ustaw przedział przypisany do pobranego znaku jako aktualny przedział kodera i wróć do punktu 3.
5. Wybierz dowolną wartość zawierającą się w końcowym przedziale i zwróć ją na wyjście.

Zaletą kodowania dla tego algorytmu jest to, że znaki są zwracane z dekodera dokładnie w takiej samej kolejności, co były one kodowane, co sprawia między innymi to, że nie ma potrzeby buforowania danych na wejściu.

Algorytm dekodowania arytmetycznego

1. Ustaw wartość przedziału na wartość początkową wynoszącą $[0, 1)$ oraz pobierz prawdopodobieństwa wszystkich znaków.
2. Podziel przedział na podprzedziały przypisane do każdego ze znaków o długościach będących iloczynem długości aktualnego przedziału oraz prawdopodobieństwa tego znaku.
3. Sprawdź, do którego z utworzonych właśnie przedziałów należy zakodowana wartość i zwróć znak przypisany do wybranego przedziału.
4. Ustaw wybrany przedział jako aktualny stan dekodera.
5. Jeśli nie zdekodowano jeszcze wszystkich znaków przejdź do punktu 2.

Od razu widać pierwszy problem w tej metodzie, albowiem systemy informatyczne nie radzą sobie dobrze z działaniami na liczbach rzeczywistych, a w tym przypadku mamy do czynienia z liczbą o bardzo wysokiej precyzji, a więc bez niezbędnych modyfikacji algorytmu nie da się go poprawnie zaimplementować. Dlatego w praktyce stosuje się renormalizację, gdy tylko długość przedziału spadnie poniżej określonej wartości, zwykle równej 0.5. Ponadto zarówno dolna granica przedziału, jak i jego długość są zwykle reprezentowane w postaci liczb całkowitych zapisywanych za pomocą N bitów, gdzie liczba reprezentująca jego długość zawiera się w granicach od 2^{N-1} do 2^N .

A więc długość przedziału wynosi:

$$a_n = 0.000...0001aaa...aaa$$

gdzie liczba zer znajdujących się za przecinkiem jest równa liczbie przeprowadzonych kroków renormalizacji. N następnych bitów jest uważana jako aktualna długość stanu. Druga wartość wchodząca w skład stanu c_n oznaczająca dolną granicę przedziału wynosi w takim razie:

$$c_n = 0.xxx...xxxccc...ccc$$

gdzie pierwsze bity x zostały przesłane na wyjście kodera podczas kroków renormalizacji, a następne N bitów jest aktualną dolną granicą przedziału stanu kodera.

Przykład

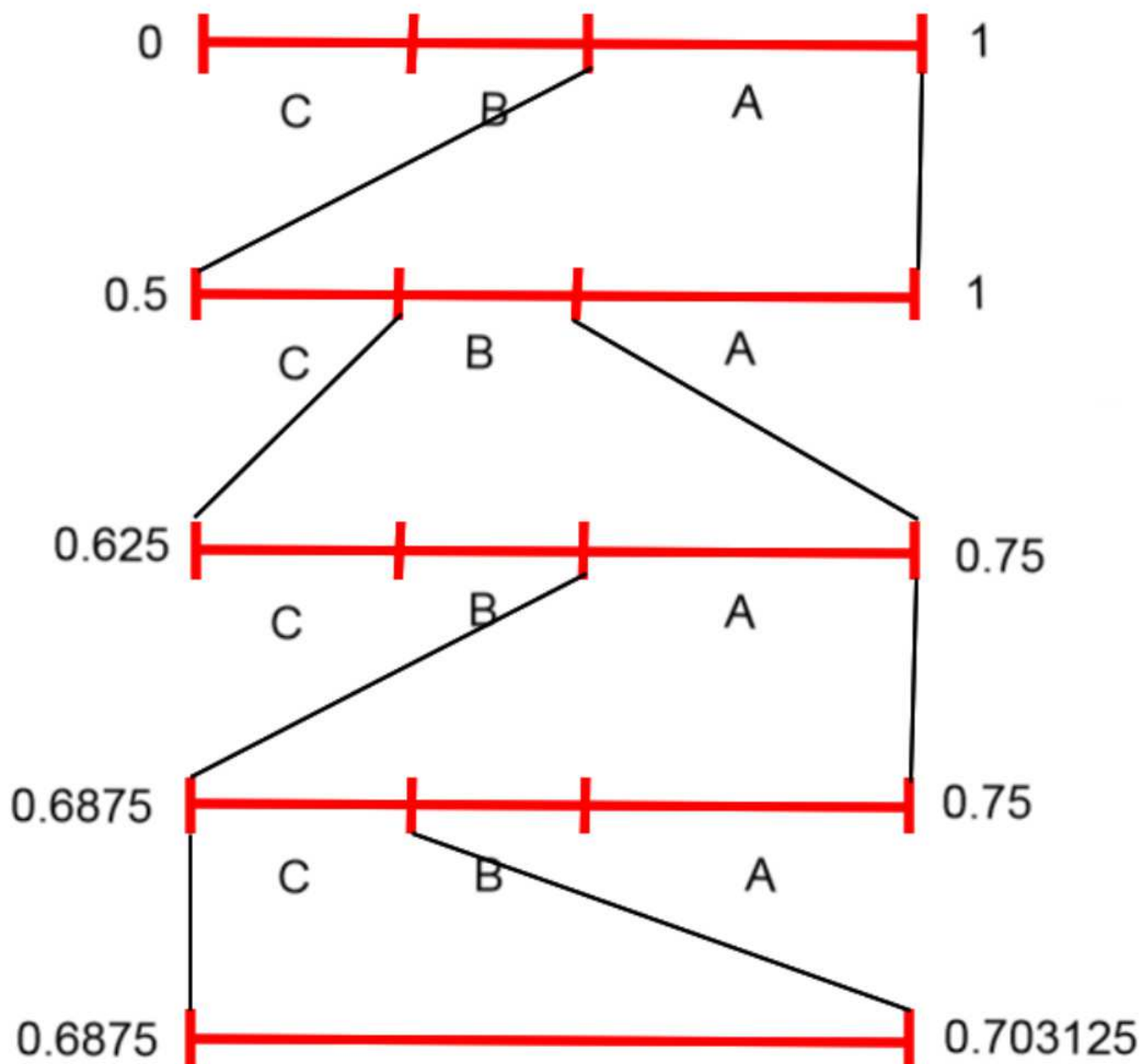
W poniższym przykładzie pokazano zastosowanie algorytmu kodowania arytmetycznego dla sekwencji znaków ABAC. W pierwszej kolejności należy obliczyć prawdopodobieństwa występowania znaków.

Znak	Prawdopodobieństwo
A	0.5
B	0.25

C	0.25
---	------

Tabela 3.3: Prawdopodobieństwa znaków dla przykładowego ciągu

Następnie przystępujemy do kodowania zgodnie z algorytmem.



Rys. 3.3: Przykład działania kodowania arytmetycznego

Ostateczny przedział, który otrzymaliśmy po zakodowaniu wszystkich znaków, wynosi $[0.6875, 0.703125]$. Teraz wystarczy wybrać dowolną liczbę rzeczywistą w tym przedziale i zwrócić ją na wyjście. Wybrana wartość może być dowolna, więc wybierzemy wartość o najkrótszej reprezentacji bitowej, tym samym zmniejszając liczbą danych do przesłania.

Wybrana wartość to: 0.6875, która to w systemie binarnym wynosi: 0.1011. Z uwagi na to, że wszystkie wartości otrzymane w tym kodowaniu są w przedziale od 0 do 1, przechowywanie zera znajdującego się przed przecinkiem jest pozbawione sensu, więc wysyłamy na wyjście ciąg 1011.

W przypadku klasycznego kodowania, do zapisania ciągu wejściowego potrzeba byłoby użyć 8 bitów, przy kodowaniu arytmetycznym wystarczą 4 bity. Ponadto kodowanie arytmetyczne najlepiej się sprawdza przy działaniu na większych ilościach danych, więc przykład ten nie pokazuje dobrze zalet kodowania arytmetycznego.

W specjalnej wersji koderu arytmetycznego, w której zbiór występujących znaków wejściowych zawiera jedynie dwie wartości $\{0, 1\}$, mówimy o binarnym koderze arytmetycznym. W tej wersji koder przyjmuje na wejście wartość aktualnego bitu oraz prawdopodobieństwo wystąpienia bitu 0. Wartość prawdopodobieństwa, z jaką kodowany jest bit, nie musi być wartością stałą, lecz może być na bieżąco estymowana dla każdego kolejnego bitu wejściowego.

Jednym z największych problemów kodowania arytmetycznego jest jego wolne działanie. Wynika to między innymi z tego, że w każdym kroku kodowania symbolu przeprowadzana jest operacja mnożenia w celu obliczenia nowych przedziałów. Operacja mnożenia jest kosztowna czasowo dla sprzętowych implementacji algorytmu. Nawet w przypadku procesorów ze specjalnie dedykowaną jednostką do wykonania szybkiego mnożenia liczb całkowitych, operacja dodawania jest zwykle szybsza w działaniu. Dlatego w celu ominięcia tej czasochłonnej operacji stosuje się przybliżenie dla długości przedziałów. Zauważono, że długość bieżącego przedziału w przypadku zastosowania renormalizacji zawiera się w granicach od 0.5 do 1. Jeżeli przyjmiemy więc jedną z potencjalnych wartości, na przykład 0.75 jako stałą długość przedziału wykorzystywaną do obliczeń granic nowych przedziałów, możemy pominąć operację mnożenia. Można zamienić operację:

$$T = A * p$$

gdzie A oznacza długość przedziału, a p wartość prawdopodobieństwa dla danego znaku, na:

$$T = p'$$

gdzie p' jest wartością po pomnożeniu p przez 0.75.

W tym przypadku nie operujemy już na prawdopodobieństwach, tylko na estymowanych długościach przedziału. Dzięki temu nowe operacje obliczania nowych przedziałów sprowadzają się do operacji dodawania i odejmowania.

Dla $x_n = 0$:

$$C_{n+1} = C_n$$

$$A_{n+1} = p'$$

Dla $x_n = 1$:

$$C_{n+1} = C_n + p'$$

$$A_{n+1} = A_n - p'$$

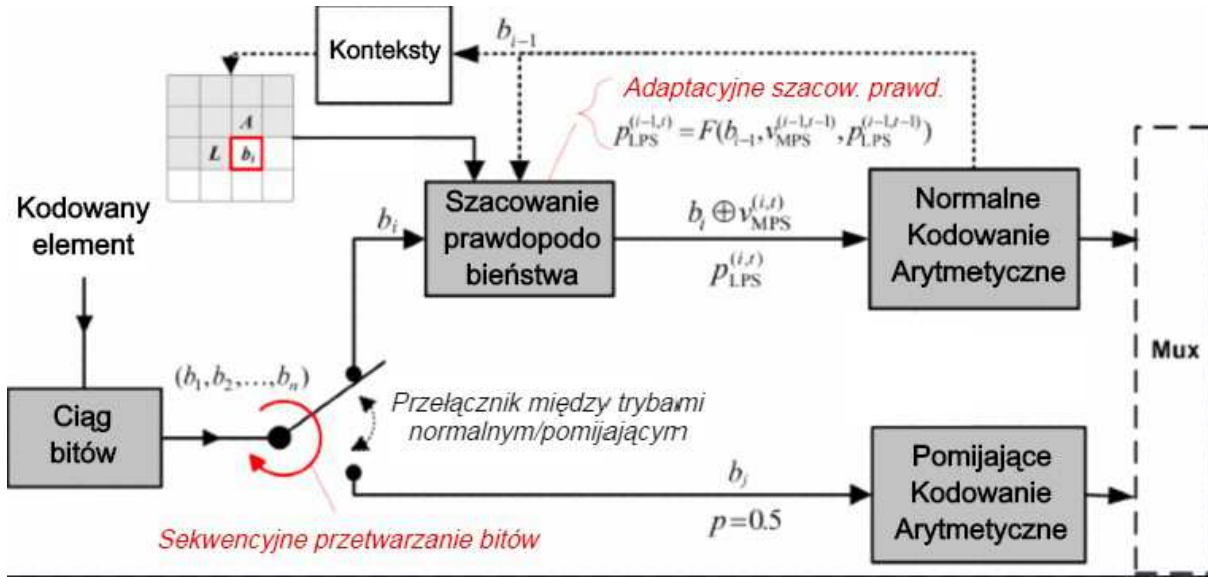
Zastosowanie jednak tego przybliżenia wymusza na nas jednak dodanie jednak kolejnych zmian w algorytmie kodowania arytmetycznego. Obliczone długości przedziałów dla określonych prawdopodobieństw są wartościami oszacowanymi dla ustalonej stałej wartości poprzedniego przedziału. A więc dla przypadków, gdy długość przedziału jest znacząco różna od ustalonej, oszacowane wartości będą wyraźnie różne od stanu faktycznego. W szczególnym przypadku może dojść do sytuacji, gdy długość przedziału przypisanego do najmniej prawdopodobnego symbolu (LPS) jest większa od długości przedziału dla najbardziej prawdopodobnego symbolu (MPS). Najprostszą metodą działania w takiej sytuacji jest zamiana obu symboli. A więc dla znaku wejściowego 0 oraz konieczności wystąpienia zamiany, działamy tak, jakbyśmy otrzymali na wejście bit 1.

CABAC

Szczególną wersją kodera arytmetycznego jest kontekstowo-adaptacyjny binarny koder arytmetyczny. Koder tego typu przyjmuje na wejście stowarzyszone ze sobą pary: bit i kontekst. Kontekst jest wartością, która określa, w ramach którego modelu prawdopodobieństwa dany bit ma być kodowany. Oznacza to, że dla każdego poszczególnego kontekstu rozkład prawdopodobieństwa między oba symbolami może być diametralnie inny. Adaptacyjność kodera oznacza natomiast, że rozpoczyna on swoje działanie z domyślnymi, początkowymi wartościami rozkładu prawdopodobieństw dla każdego z poszczególnych kontekstów. Wartości te jednak zmieniają się w trakcie procesu kodowania. Koder przechowuje aktualny stan każdego kontekstu w postaci indeksu do tablicy oraz wartości najbardziej prawdopodobnego symbolu. Tablica, na którą wskazuje stan kontekstu, przechowuje informacje o długości przedziału najmniej prawdopodobnego znaku, oraz informacje wykorzystywane do adaptacji stanu kontekstu. Tak więc, w przypadku kodowania najmniej prawdopodobnego znaku, koder sięga do miejsca w tablicy wskazywanego przez stan kontekstu, skąd pobiera informacje o następnym stanie kontekstu oraz o potencjalnej zamianie wartości znaku uważanego za bardziej prawdopodobny. To znaczy, że prawdopodobieństwo najbardziej prawdopodobnego znaku będzie spadało. Podobnie się dzieje, gdy kodowany jest bardziej prawdopodobny znak, z tą różnicą, że niepotrzebna jest nam informacja o zamianie wartości znaku. W tym przypadku prawdopodobieństwo bardziej prawdopodobnego znaku wzrasta.

Dekoder podobnie jak koder rozpoczyna swoje działanie od zainicjowania stanów kontekstów wartościami początkowymi, a następnie adaptuje ich stan dzięki wartościom zwróconym z dekodera do danej chwili. Dekoder dostaje na wejście zakodowany strumień bitowy, a następnie kolejno dostaje wartość kontekstu, a zwraca zdekodowaną wartość bitu. Dzięki temu, że dekodowanie rozpoczyna działanie od pierwszego zakodowanego znaku, nie jest potrzebne przesyłanie dodatkowych metadanych dotyczących prawdopodobieństw, a

adaptacja kodowania jest prosta w implementacji. Wystarczy, że zarówno koder, jak i dekodery będą używały takich samych wartości początkowych kontekstów.



Rys. 3.4: Schemat działania kodera CABAC [10]

3.3. Asymetryczny system numeryczny

Asymetryczne kodowanie numeryczne (z angielskiego Assymetrical Numeral Systems, w skrócie ANS) jest nowym sposobem kodowania entropijnego opracowanym przez dr. Jarosława Dudę. Łączy on w sobie stopień kompresji porównywalny z kodowaniem arytmetycznym oraz szybkość działania kodowania Huffmana. W przeciwieństwie do kodowania arytmetycznego, który przechowuje aktualny stan za pomocą dwóch wartości, które razem reprezentują przedział, asymetryczne kodowanie numeryczne korzysta jedynie z jednej wartości stanu. Dzięki korzystaniu z ANS można pominąć wykonywanie ciągłych obliczeń podczas samego kodowania. Można to osiągnąć, stosując tANS, a więc wariant ANS, który przechowuje wyniki obliczeń w tablicy [1].

ANS znajduje coraz to szersze zastosowanie, między innymi w kompresorach Zstandard, LZFSE, DivANS, a także w standardzie kompresji obrazów JPEG XL.

Asymetryczne kodowanie numeryczne opiera się na pomysłe przechowywania informacji w pojedynczej liczbie. Nowa wartość stanu kodera po zakodowaniu symbolu s o prawdopodobieństwie p wynosi:

$$x' \approx x/p$$

Natomiast podczas dekodowania nowa wartość stanu dekodera wynosi:

$$x' \approx x * p$$

Operacja dzielenia aktualnego stanu kodera przez wartość prawdopodobieństwa, równoznaczna jest dodawaniu informacji o odpowiedniej entropii:

$$\log(x/p) = \log(x) + \log(1/p)$$

Podobnie jak w przypadku standardowych systemów numerycznych, dodanie nowego bitu informacji do istniejącego strumienia bitowego polega na wstawieniu go na ostatnią pozycję wartości stanu. Sposób ten jednak dla przypadku, gdy prawdopodobieństwo występowania 0 i 1 jest różne, wprowadza nadmiarowość w strumieniu danych. Dlatego ANS uogólnia pojęcie systemów numerycznych, wprowadzając asymetryczność zależną od prawdopodobieństwa. Oznacza to, że wartości będące wynikiem kodowania bardziej prawdopodobnego znaku są gęściej rozłożone.

Jednak pomimo wielu zalet, jest również jedna wada ANS, albowiem dekodowanie znaków w tym przypadku przebiega w odwrotnej kolejności niż były one kodowane, a więc pierwszy zdekodowany symbol jest ostatnim symbolem, który trafił na koder. Powoduje to konieczność buforowania danych oraz utrudnia zastosowanie tego sposobu kodowania w wersji adaptacyjnej.

Z uwagi na to, że wartość stanu kodera, będzie wraz z czasem ciągle rosła, należy zastosować renormalizację wartości. Przyjmujemy, że wartość stanu może znajdować się tylko w ograniczonym przedziale $[2^r, 2^{r+1} - 1]$. Gdy podczas kodowania następny znak spowoduje przekroczenie górnej dopuszczalnej wartości dla stanu, to wysuwa się najmniej znaczące bity stanu kodera, do momentu, aż wartość spadnie poniżej jej górnej granicy. Wysunięte bity wstawia się na wyjście kodera. Przy dekodowaniu natomiast wartość może spaść poniżej dolnej granicy, w przypadku czego należy wsunąć bity z wejścia dekodera na najmniej znaczące pozycje stanu. Wstawiamy bity pobrane z wejścia do momentu, aż przekroczymy dolną granicę dopuszczalnego przedziału. Należy pamiętać, że ostatnie bity wysuwane z kodera ANS będą pierwszymi bitami wstawianymi do dekodera. A więc strumień skompresowanych danych zwróconych z kodera należy na sam koniec odwrócić. W celu obliczenia wartości, do jakiej należy zrenormalizować stan podczas kodowania, można zastosować wzór:

$$x \approx 2 * L * p$$

gdzie L jest równy długości dopuszczalnego przedziału stanów, a p równe jest prawdopodobieństwu danego symbolu.

W praktyce stosuje się jeden z kilku wariantów ANS, przede wszystkim uABS, rANS oraz tANS.

uABS

W przypadku wariantu uABS (uniform Asymmetric Binary System) stosowany jest dwuelementowy alfabet wejściowy. Dzielimy zbiór liczb naturalnych na dwa zbiory, których gęstość jest zależna od rozkładu prawdopodobieństw znaków wejściowych.

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
x_0		0	1		2	3		4	5	6		7	8		9	10		11	12
x_1	0			1			2				3			4			5		

Rys. 3.5: Rozkład liczb naturalnych w koderze uABS dla $P(1) = 0.3$

Można wyznaczyć dokładne wzory do kodowania

Dla kodowania symbolu 0:

$$x' = \left\lceil \frac{x+1}{1-q} \right\rceil - 1$$

Dla kodowania symbolu 1:

$$x' = \left\lfloor \frac{x}{q} \right\rfloor$$

Natomiast wzory wykorzystywane do procesu dekodowania wyglądają następująco:

Dekodowanie symbolu:

$$s = \lceil (x+1)q \rceil - \lceil xq \rceil$$

W zależności od zwróconej wartości s stosuje się jedno z dwóch równań.

Dla $s = 0$:

$$x_0 = x - \lceil xq \rceil$$

Dla $s = 1$:

$$x_1 = \lceil xq \rceil$$

Przykład

W poniższym przykładzie zaprezentowano działanie kodowania uABS na ciągu bitów: 0010101000. Prawdopodobieństwa dla obu bitów wynoszą więc: $P(0) = 0.7$, $P(1) = 0.3$

Przyjmujemy, że stan kodera musi zawierać się w wartościach $[8, 15]$, a początkowa

wartość wynosi 8.

Ponieważ znaki są dekodowane w odwrotnej kolejności, niż były kodowane, więc żeby otrzymać z dekodera znaki w dobrej kolejności, odwracamy ciąg wejściowy. Koder dostaje zatem wejście: 0001010100.

Stan kodera	Kodowany symbol	Stan kodera po renormalizacji	Wysunięte bity	Stan kodera po kodowaniu symbolu
8	0	8	-	12
12	0	6	0	9
9	0	9	-	14
14	1	3	10	10
10	0	10	-	15
15	1	3	11	10
10	0	10	-	15
15	1	3	11	10
10	0	10	-	15
15	0	7	1	11

Tabela 3.4: Działanie uABS dla przykładowego ciągu bitów

Na sam koniec kodowania należy zwrócić pozostałe bity stanowiące końcowy stan kodera, a także odwrócić zwrócony strumień bitów.

Otrzymujemy więc ciąg bitów: 101111111100

Z uwagi na to, że kodowana przykładowa sekwencja była krótka, więc nie udało się zmniejszyć liczby bitów danych. Wejściowy zawiera 10 bitów, a zakodowany 12.

rANS

Kolejną bardzo przydatnym wariantem ANS jest rANS (range Asymmetric Numeral System). To, co różni go od uABS, jest przede wszystkim to, że nie ma ograniczenia co do liczby znaków w alfabecie wejściowym. W pierwszej kolejności obliczamy prawdopodobieństwa każdego z symboli alfabetu i zaokrąglamy ich wartości do ułamków w postaci:

$$p(i) = f(i)/2^n$$

gdzie n jest dowolną wybraną przez nas wartością. Przedział $[0, 2^n - 1]$ dzielimy na krótsze przedziały o długościach $f(i)$. Wzory na kodowanie Wyglądają następująco:

$$x' = \left(\left\lfloor \frac{x}{f(s)} \right\rfloor \ll n \right) + (x \bmod f(s)) + D(s)$$

gdzie funkcja $D(s)$ oblicza granicę dolną przedziału przypisanego do symbolu s .

W celu zdekodowania symbolu należy pobrać ostatnie n bitów aktualnego stanu (oznaczymy tę liczbę jako K) oraz sprawdzić, do którego z przedziałów należy otrzymana wartość. Zwrócona wartość zdekodowanego znaku oznaczymy jako s . Wzór do obliczenia nowego stanu po dekodowaniu:

$$x' = f(s) * (x \gg n) + K - D(s)$$

tANS

Wariant tablicowy asymetrycznego kodowania numerycznego, czyli tANS pozwala na umieszczenie wartości dotyczących następnego stanu kodera, a także parametrów renormalizacji w tablicy. Pozwala to na otrzymanie znacznie szybszego działania, biorąc pod uwagę to, że jedyne operacje wykonywane w trakcie samego kodowania polegają na pobraniu określonych elementów z tablicy.

W przypadku kodowania wyróżniamy trzy wartości, które są umieszczane w tablicy:

- następny stan kodera
- liczba wysuwanych na wyjście bitów
- wartości wysuwanych bitów

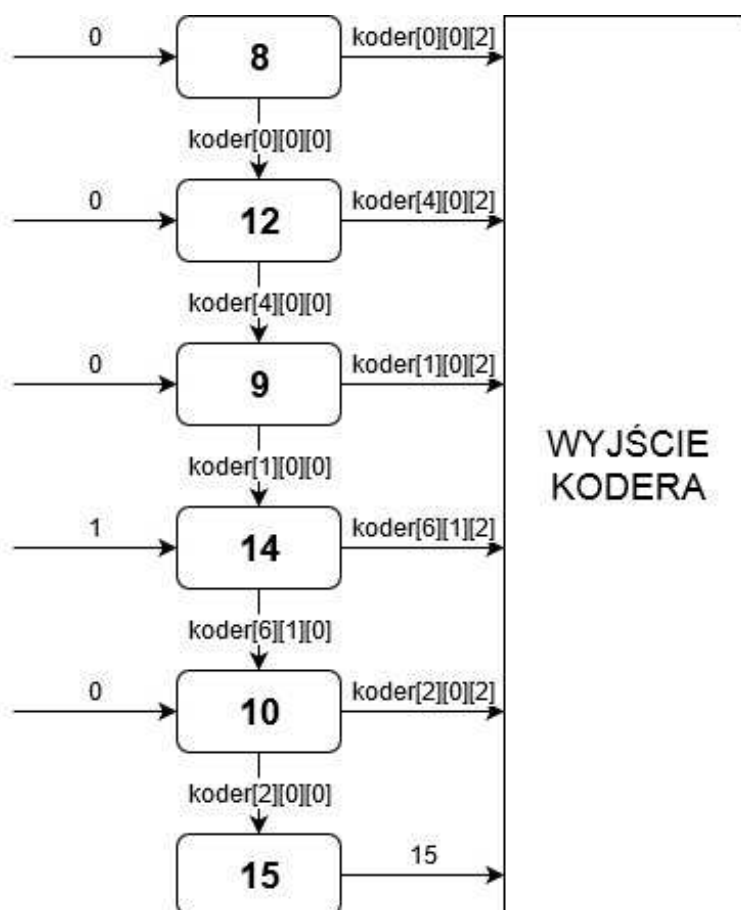
Wartości te zależne są od dwóch wartości: aktualnego stanu kodera i kodowanego znaku.

Natomiast dla procesu dekodowania mamy wartości:

- następny stan dekodera przed renormalizacją
- liczba pobieranych bitów z wejścia
- zdekodowany znak

Tutaj w celu pobrania odpowiednich wartości z tablicy potrzebujemy jedynie stan dekodera.

Na rysunku 3.6 zademonstrowano działanie kodowania tANS dla przykładowego ciągu bitów: 00101 o prawdopodobieństwach $P(0) = 0.7$, $P(1) = 0.3$. Wartości zawierają się w przedziale $[8, 15]$, oraz arbitralnie przyjmujemy stan początkowy jako 8. Następne wartości stanu kodera, które to należy umieścić w tablicy tANS, można odczytać z tabeli z rysunku 3.5. Przejścia z tej tabeli zostały obliczone właśnie dla rozpatrywanego przez nas rozkładu prawdopodobieństw.



Rys. 3.6: Działanie kodera tANS na przykładowym ciągu bitów

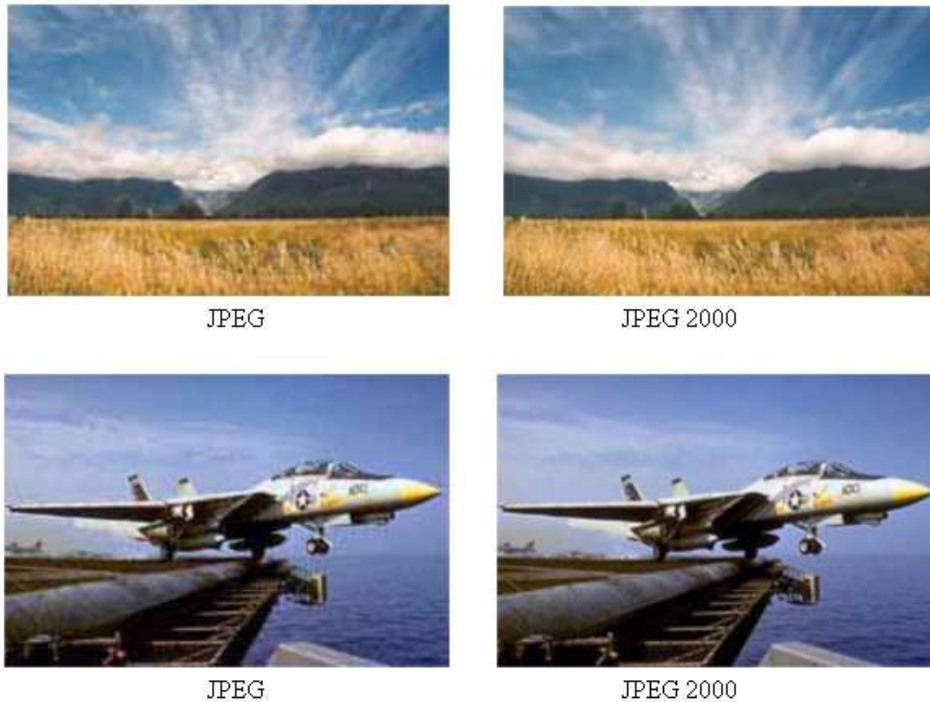
W tym przypadku zmienna koder jest tablicą trójwymiarową, gdzie pierwszy wymiar oznacza stan kodera pomniejszony o 8, drugi wymiar oznacza wartość kodowanego znaku, a w ostatnim przechowywane są trzy wartości, 0 – następny stan, 1 – liczba wysuwanych bitów, 2 – wartość wysuwanych bitów.

Rozdział 4

Standard kompresji obrazów JPEG 2000

4.1. Ogólny schemat standardu

Standard kompresji obrazów JPEG 2000 został opracowany w 2000 roku przez *The Joint Photographic Experts Group*.



Rys. 4.1: Porównanie jakości obrazu po kompresji JPEG oraz JPEG 2000 [7]

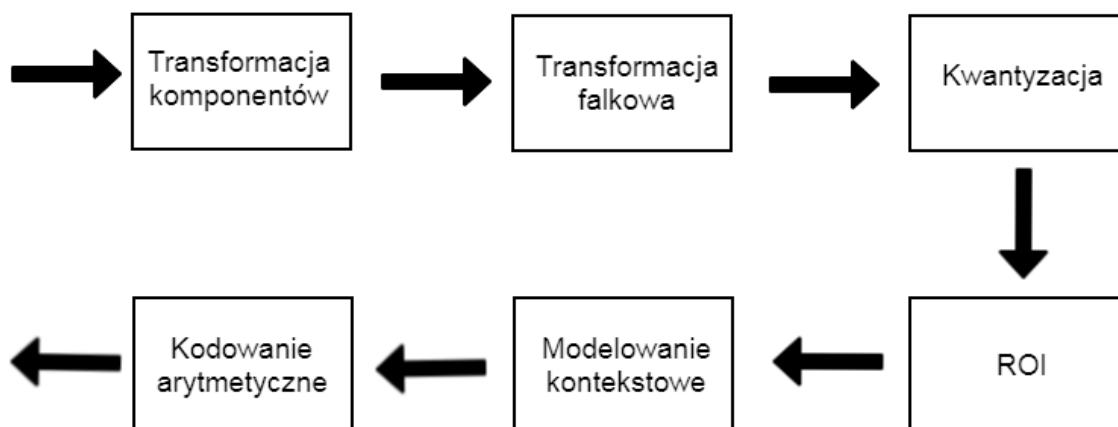
Cechy charakterystyczne

- Lepsza jakość obrazu dla silnie skompresowanych danych. Standard JPEG 2000 ma przewagę nad JPEG dla każdego stopnia kompresji obrazów, ale to właśnie dla silnie skompresowanych obrazów, różnice te są najlepiej widoczne.
- Możliwość kompresji obrazów posiadających różną liczbę bitów na próbkę. JPEG 2000 można stosować jednakowo dla obrazów posiadających 1 bit na próbkę oraz dla obrazów posiadających ponad 16 bitów na jeden kolor próbki.
- Możliwość kompresji progresywnej, zarówno jakości obrazu, jak i jego rozdzielczości. Jest to cecha ważna szczególnie w przypadku transmisji obrazu przez wolne łącze internetowe, kompresja progresywna pozwala wyświetlić obraz gorszej jakości lub rozdzielczości posiadając tylko część danych.
- Możliwość zastosowania kompresji stratnej oraz bezstratnej. Pomimo że JPEG 2000

jest stosowany przeważnie w trybie kompresji stratnej, to pozwala on również wykorzystać kompresję bezstratną.

- Łatwy dostęp do dowolnego regionu obrazu.
- Standard JPEG 2000 pozwala na to, by otrzymać obrazy o różnej rozdzielczości i jakości z tego samego skompresowanego strumienia danych.

Standard



Rys. 4.2: Schemat kodera JPEG 2000

Ogólny schemat działania kodera zgodnego ze standardem JPEG 2000 składa się z następujących kroków:

- Przesunięcie wszystkich próbek obrazu w razie potrzeby. Próbkę obrazu powinny być w granicach $[-N, N]$. Jeśli wymóg ten nie jest spełniony, przeprowadzane jest przesunięcie wartości wszystkich próbek obrazu, tak by wymóg ten został spełniony
- Transformacja modeli przestrzeni barw obrazu. Kompresja JPEG 2000 o wiele lepiej radzi sobie z kompresją danych zapisanych w przestrzeni barw YCbCr. A więc jeśli zastosowano inną przestrzeń (na przykład RGB), powinno się w celu poprawy kompresji przetransformować obraz na YCbCr. Model YcbCr jest o tyle lepszy, że zawiera w oddzielnych składowych informacje o jasności i barwie obrazu. Dzięki temu, że ludzkie oko jest bardziej wyczulone na zmiany w jasności niż barwie, można to wykorzystać, zmieniając stopień kwantyzacji dla poszczególnych komponentów. Krok ten nie jest jednak wymagany.
- Dyskretna transformata falkowa (Discrete Wavelet Transform) - transformacja obrazu dzieląca go na cztery oddzielne obrazy składające się z określonych częstotliwości pierwotnego obrazu. DWT dzieli obraz na pasma LL (niskie częstotliwości w obu kierunkach), HH (wysokie częstotliwości w obu kierunkach), LH, HL (w jednym kierunku wysokie częstotliwości, oraz niskie w drugim). Następnie dokładnie taką

samą transformację można zastosować rekurencyjnie wobec pasma LL tyle razy, ile będzie to wymagane.

- Kwantyzacja jest stosowana tylko w przypadku kompresji stratnej obrazu. Można modyfikować liczbę progów kwantyzacji stosowanych dla poszczególnych pasm obrazu.
- ROI (Region of Interest) - W koderze można zaznaczyć obszary obrazu, dla którego zastosowane zostanie skalowanie skwantowanych próbek.
- Modelowanie kontekstowe – dla każdego bloku skwantowanych próbek zastosowane jest modelowanie kontekstowe, które zwraca ciąg par: kodowany bit i kontekst z nim związany. Dzieli się skwantowane próbki na płaszczyzny bitowe, czyli ciąg bitów znajdujących się na tych samych pozycjach we wszystkich próbkach kodowanego bloku. Dla każdej płaszczyzny bitowej przeprowadzane są trzy przejścia (każdej z wyjątkiem płaszczyzny zawierającej najbardziej znaczące bity). Każdy bit na płaszczyźnie bitowej musi zostać zakodowany na jednym z przejść. Na podstawie przejścia, w ramach którego kodowany jest bit oraz jego sąsiedztwa obliczany jest kontekst kodowanego bitu
- Koder arytmetyczny – kontekstowo – adaptacyjny koder arytmetyczny (CABAC) typu MQ. Przyjmuje pary bit i kontekst, a na wyjście zwraca skompresowany ciąg bitów.

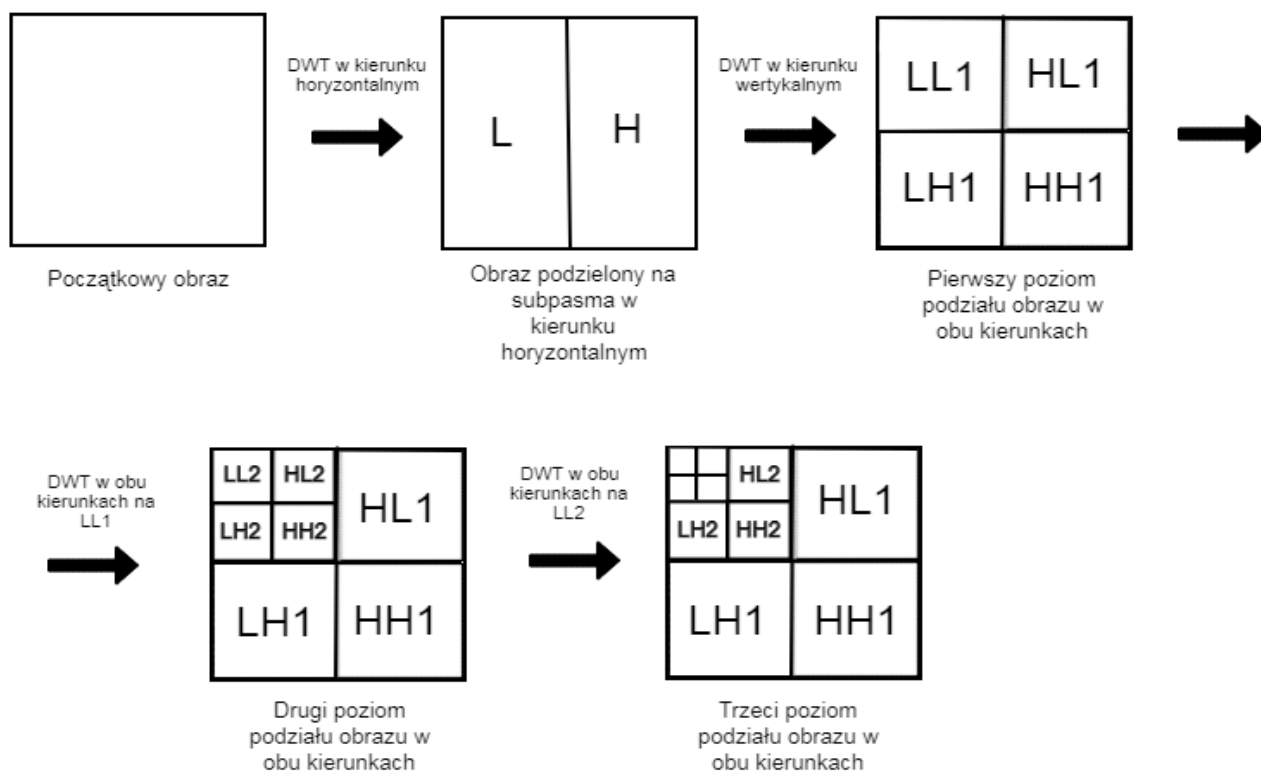
Dekompresja obrazu przebiega bardzo podobnie, tylko w odwrotnej kolejności. Standard JPEG 2000 pozwala na bezstratną kompresję obrazu, dlatego wyróżniamy dwie ścieżki kompresji: stratną i bezstratną. Ścieżka bezstratna wymaga oddzielnych procesów transformacji kolorów, dyskretnej transformaty falkowej oraz nie posiada kroku kwantyzacji.

W skompresowanym ciągu bitowym wartości od 0xFF90 do 0xFFFF oznaczają markery o specjalnym znaczeniu, dlatego koder entropijny musi zadbać, by takie wartości nie występowały w zakodowanym ciągu [4].

4.2. Dyskretna transformata falkowa

W standardzie JPEG 2000 transformatę Fourier'a zamieniono na dyskretną transformatę falkową. Zastosowano dwuwymiarową transformatę (w kierunku horyzontalnym i wertykalnym). Pojedyncze DWT to operacja, która dostając skończony ciąg próbek $x[i]$ zwraca dwa ciągi $a[i]$ oraz $b[i]$. Pierwszy z nich oznacza pasmo o niskiej częstotliwości i jego powstanie może być rozumiane jako filtrowanie dolnoprzepustowe ciągu wejściowego, a następnie porzucenie co drugiej próbki. Drugi z ciągów natomiast jest pasmem o wysokiej częstotliwości i jego powstanie może być rozumiane jako filtrowanie górnoprzepustowe ciągu wejściowego, a następnie porzucenie co drugiej próbki. Łączna długość ciągów wyjściowych jest równa ciągu wejściowemu oraz nie tracimy na transformacie żadnych danych. Jeżeli wykonamy DWT na próbkach obrazu najpierw w jednym, a następnie w drugim kierunku otrzymamy transformatę dwuwymiarową.

Ponieważ większość danych istotnych dla ludzkiej percepcji znajduje się paśmie o niskiej częstotliwości, dlatego możemy rekurencyjnie dokonywać kolejnych podziałów na paśmie LL.



Rys. 4.3: Dyskretna transformata falkowa

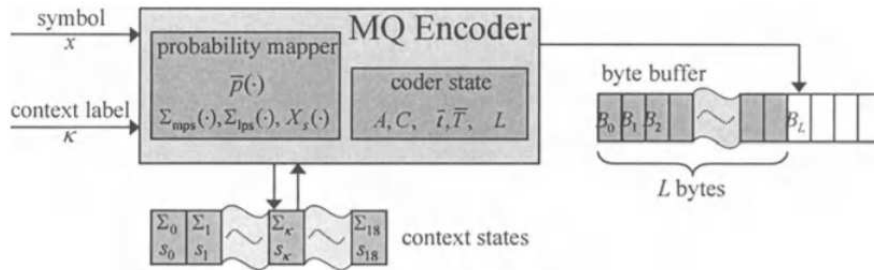
4.3. Modelowanie kontekstowe

Modelowanie kontekstowe ma za zadanie, dostając na wejście bloki skwantowanych współczynników DWT, przekazać do kodera MQ ciąg par: bit i kontekst. Podczas dekodowania, modelowanie zwraca do dekodera MQ aktualną wartość kontekstu.

Współczynniki DWT składają się z wartości współczynnika oraz jego znaku. Modelowanie dzieli blok współczynników na mapy bitowe, a więc bity znajdujące się na tej samej pozycji współczynnika. Następnie przetwarza kolejne mapy bitowe, zaczynając od najbardziej znaczącej, przechodząc w stronę mniej znaczących. Na każdej z tych map wykonuje się trzy przejścia (significance propagation pass, magnitude refinement pass, cleanup pass). Wyjątkiem jest pierwsza kodowana mapa bitowa, dla której przeprowadzane jest tylko ostatnie przejście. Każdy bit dla określonej mapy zostaje przetworzony w jednym z tych trzech przejść. Bit znaku kodowany, gdy tylko współczynnik DWT tego znaku stanie się znaczący (napotka pierwszy bit 1, zaczynając od najbardziej znaczących bitów).

Wartość kontekstu dla kodowania obliczana jest na podstawie: numeru przejścia, w którym znak jest kodowany oraz jego najbliższego otoczenia.

4.4. Koder entropijny



Rys. 4.4: Schemat kodera MQ

Koder MQ składa się z następujących elementów:

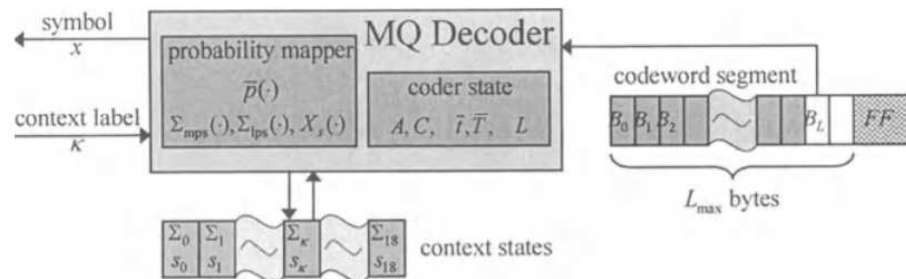
- Stan dla każdego z 19 kontekstów. Stan kontekstu składa się z dwóch wartości: najbardziej prawdopodobny symbol (0 lub 1), indeks do aktualnego modelu prawdopodobieństwa (wartość od 0 do 46)
- Tablica zawierająca 47 elementów oznaczających długość przedziału dla najmniej prawdopodobnego symbolu. Indeks do określonego elementu tej tablicy znajduje się w stanie kontekstu: P
- Tablica zawierająca 47 elementów oznaczających indeks następnego stanu kontekstu, jeśli zakodowano właśnie najmniej prawdopodobny symbol. Wykorzystywana do aktualizacji stanu kontekstu: $nLPS$
- Tablica zawierająca 47 elementów oznaczających indeks następnego stanu kontekstu, jeśli zakodowano właśnie najbardziej prawdopodobny symbol. Wykorzystywana do aktualizacji stanu kontekstu: $nMPS$
- Tablica zawierająca 47 elementów określających czy dojdzie do zmiany najbardziej prawdopodobnego symbolu dla danego kontekstu w przypadku zakodowania najmniej prawdopodobnego symbolu: $switch$
- Granica dolna aktualnego przedziału: C
- Długość aktualnego przedziału: A
- Liczba przesunięć rejestru C , zanim dojdzie do następnej transmisji bajta: t
- Rejestr pośredni zawierający bajt, który będzie zapisany na wyjście: T
- Liczba zapisanych bajtów na wyjście: L

Kodowanie:

- mps – MPS dla danego kontekstu, p – długość przedziału LPS dla danego kontekstu

(pobrana z tablicy), x – kodowany symbol

- $s = \text{mps}, A = A - p$
- Jeśli $A < p$
 - $s = 1 - s$
- Jeśli $x = s$:
 - $C = C + p$
- W przeciwnym razie:
 - $A = p$
- Jeśli $x = \text{mps}$
 - $p = \text{nMPS}(p)$
- W przeciwnym wypadku:
 - $\text{mps} = \text{mps xor switch}(p)$
 - $p = \text{nLPS}(p)$
- Dopóki $A < 2^{15}$
 - $A = 2A, C = 2C, t = t - 1$
 - Jeśli $t = 0$
 - zwróć bajt na wyjście kodera



Rys. 4.5: Schemat dekodera MQ

Dekoder dostaje na wejście zakodowaną sekwencję oraz wartość kontekstu, zwraca natomiast zdekodowany symbol.

Dekodowanie:

- mps – MPS dla danego kontekstu, p – długość przedziału LPS dla danego kontekstu (pobrana z tablicy)
- $s = \text{mps}, A = A - p$

- Jeśli $A < p$
 - $s = 1 - s$
- Jeśli $C^{active} < p$
 - zwróć $x = 1 - s$
 - $A = p$
- W przeciwnym wypadku:
 - zwróć $x = s$
 - $C^{active} = C^{active} - p$
- Jeśli $x = \text{mps}$
 - $p = \text{nMPS}(p)$
- W przeciwnym wypadku:
 - $\text{mps} = \text{mps} \text{ xor } \text{switch}(p)$
 - $p = \text{nLPS}(p)$
- Dopóki $A < 2^{15}$
 - zrenormalizuj stan dekodera

Rozdział 5

Implementacja

Zmian dokonano na implementacji referencyjnej kodera i dekodera JJ2000 [11]. Procesem kontekstowego kodowania entropijnego zajmuje się klasa MQCoder w pakiecie `ucar.jpeg.jj2000.j2k.entropy.encoder`. Dekodowanie natomiast obsługiwane jest przez klasę MQDecoder w pakiecie `ucar.jpeg.jj2000.j2k.entropy.decoder`. Udało się przeprowadzić odpowiednie modyfikacje, dokonując zmian jedynie w tych dwóch klasach, nie zmieniając ich publicznych metod, ani żadnych pozostałych klas w oryginalnej implementacji.

W poniższej implementacji zastosowano ANS w wariantcie tANS, operującym na alfabecie dwuelementowym. Tak więc w obydwu klasach pierwszym etapem było utworzenie tablic z wynikami, do których koder oraz dekodery mogły sięgać.

Ponieważ wykorzystano ANS w postaci stablicowanej, a więc dopuszczalne stany kodera muszą się zawierać w granicach $[2^n, 2^{n+1} - 1]$. Przedział dopuszczalnych stanów został ustalony na $[1024, 2047]$, co daje łącznie 1024 dopuszczalnych stanów. Zwiększenie tej wartości nie zmniejszy liczby stosowanych renormalizacji, a więc nie przyspieszy kodowania, za to znacznie zwiększy wielkość tablic kodowania, a więc zapotrzebowanie na pamięć. Liczba stanów nie może jednak również zbyt mała, ponieważ nie pozwoliłoby to stosowanie modeli prawdopodobieństw, w których jest duża dysproporcja między częstością występowania poszczególnych bitów, co mogłoby obniżyć stopień kompresji. Testy kompresji wykazały, że taka liczba stanów jest w pełni wystarczająca i nie wpływa znacząco na stopień kompresji.

Podobnie jak w referencyjnej implementacji, bity kodowane są w ramach kontekstów. Każdy kontekst posiada odpowiadający mu aktualny stan, który przechowuje informacje o stosowanym modelu prawdopodobieństwa, oraz mniej prawdopodobnym bicie.

Klasa MQCoder

Pierwszym krokiem było wygenerowanie tablic kodowania dla każdego modelu prawdopodobieństw. Liczba modeli oraz wartości prawdopodobieństw pozostały takie same jak w referencyjnej implementacji.

W referencyjnej implementacji wykorzystującej CABAC wykorzystywano tablice przechowujące długości przedziału dla mniej prawdopodobnego znaku. Na jej podstawie można obliczyć prawdopodobieństwa oraz granice poniżej których wartość stanu musi zostać zrenormalizowana, tak aby po zakodowaniu wartość stanu mieściła się w wymaganych granicach. Na podstawie tych dwóch wartości tworzona jest tablica kodowania. Inicjowanie tych wartości przeprowadzane jest w bloku statycznym.

```

static
{
    probabilities = new double[47];
    limits = new int[47][2];
    getProbabilitiesAndLimits(probabilities, limits);
    coderLookupTable = getCoderLookupTable();
}

```

Metoda `getProbabilitiesAndLimits` oblicza prawdopodobieństwo dla każdego modelu wykorzystując fakt, że pierwszy model odpowiada prawdopodobieństwu 0.5. Ponadto, z uwagi na to, że przedział dozwolonych stanów jest ograniczony, więc dla niskich wartości prawdopodobieństwa, wartości następnych stanów kodera zawsze przekraczały górną granicę, a więc przyjęto, że ich wartości jako ostatnio wykryte wystarczająco wysokie prawdopodobieństwo, tym samym je wyłączając.

```

public static void getProbabilitiesAndLimits(double[] probabilities, int[][] limits) {
    double wholeRange = qe[0] * 2;
    double last_correct_probability = 0.5;
    int[] last_correct_limits = null;
    boolean values_incorrect = false;
    for (int i = 0; i < 47; i++)
    {
        if (!values_incorrect)
        {
            double temp_prob = qe[i] / wholeRange;
            int[] temp_limits = getLimits(temp_prob);
            if (temp_limits[0] < 1)
            {
                probabilities[i] = last_correct_probability;
                limits[i] = last_correct_limits;
                values_incorrect = true;
            }
            else
            {
                probabilities[i] = last_correct_probability = temp_prob;
                limits[i] = last_correct_limits = temp_limits;
            }
        }
        else
        {
            probabilities[i] = last_correct_probability;
            limits[i] = last_correct_limits;
        }
    }
}

```


Wartości długości mniej prawdopodobnego znaku, wykorzystywane do obliczania prawdopodobieństw:

```
final static
int[] qe = {0x5601, 0x3401, 0x1801, 0x0ac1, 0x0521, 0x0221, 0x5601,
            0x5401, 0x4801, 0x3801, 0x3001, 0x2401, 0x1c01, 0x1601,
            0x5601, 0x5401, 0x5101, 0x4801, 0x3801, 0x3401, 0x3001,
            0x2801, 0x2401, 0x2201, 0x1c01, 0x1801, 0x1601, 0x1401,
            0x1201, 0x1101, 0x0ac1, 0x09c1, 0x08a1, 0x0521, 0x0441,
            0x02a1, 0x0221, 0x0141, 0x0111, 0x0085, 0x0049, 0x0025,
            0x0015, 0x0009, 0x0005, 0x0001, 0x5601 };
```

Początkowo tablica kodowania przechowywana była w postaci czterowymiarowej tablicy, jednak okazało się, że miało to duży wpływ na szybkość działania oprogramowania, więc ostatecznie użyta została tablica jednowymiarowa. Ponadto trzy wartości pobierane za każdym razem z tablicy połączone są bitowo w jedną liczbę całkowitą, aby ograniczyć liczbę dostępów do tablicy. Tablica przechowuje wartości: następny stan, liczba bitów zwracanych na wyjście oraz wartość tych bitów. Tablica dla każdego modelu obliczana jest osobno, a następnie łączone są one w jednowymiarową tablicę.

```
public static int[] getCoderLookupTable() {
    int[][][][] lookupTable = new int[47][STATE_RANGE][2][3];

    for (int i = 0; i < 46; i++)
    {
        lookupTable[i] = getCoderLookupTableForProbability(probabilities[i], limits[i]);
    }

    // Last lookup table has the same probabilities as the first one
    lookupTable[46] = lookupTable[0];

    int[] lookupTableFinal = new int[47 * STATE_RANGE * 2];
    int index = 0;
    for (int i = 0; i < 47; i++)
    {
        for (int j = 0; j < STATE_RANGE; j++)
        {
            for (int k = 0; k < 2; k++)
            {
                lookupTableFinal[index++] = (lookupTable[i][j][k][0] & 0xFF) |
                    ((lookupTable[i][j][k][1] & 0xFF) << 8) |
                    ((lookupTable[i][j][k][2] & 0xFFFF) << 16);
            }
        }
    }

    return lookupTableFinal;
}
```

W metodzie `getCoderLookupTableForProbability` obliczane są wartości następnego stanu oraz parametry renormalizacyjne. Obliczone wcześniej limity wykorzystywane są do wyznaczenia parametrów renormalizacyjnych, natomiast prawdopodobieństwo wykorzystywane jest do obliczenia następnego stanu, wykorzystując wzory ANS.

```
private static int[][][] getCoderLookupTableForProbability(double p, int[] limits) {
    int[][][] lookupTable = new int[STATE_RANGE][2][3];
    int state;
    int shiftedBits;
    int nrOfShifted;

    for (int i = 0; i < STATE_RANGE; i++)
    {
        // LPS
        state = i + STATE_RANGE;
        shiftedBits = 0;
        nrOfShifted = 0;
        while(state > limits[0])
        {
            shiftedBits += (state & 1) << nrOfShifted;
            nrOfShifted++;
            state >>= 1;
        }
        state = coderZeroEquation(state, p);
        lookupTable[i][0][0] = shiftedBits;
        lookupTable[i][0][1] = nrOfShifted;
        lookupTable[i][0][2] = state;

        // MPS
        state = i + STATE_RANGE;
        shiftedBits = 0;
        nrOfShifted = 0;
        while(state > limits[1])
        {
            shiftedBits += (state & 1) << nrOfShifted;
            nrOfShifted++;
            state >>= 1;
        }
        state = coderOneEquation(state, p);
        lookupTable[i][1][0] = shiftedBits;
        lookupTable[i][1][1] = nrOfShifted;
        lookupTable[i][1][2] = state;
    }

    return LookupTableCorrection(lookupTable);
}
```

Metody `coderZeroEquation` i `coderOneEquation` wykorzystywane są do obliczenia następnego stanu kodera.

```
private static int coderZeroEquation(int state, double probability) {  
    return (int)(Math.ceil(((double) (state + 1)) / probability)) - 1;  
}
```

```
private static int coderOneEquation(int state, double probability) {  
    return (int) Math.floor(((double) state) / (1.0 - probability));  
}
```

Ponieważ ANS dekoduje znaki w odwrotnej kolejności niż były one kodowane, więc należy odwrócić bity na wejściu kodera, tak aby nie trzeba było tego robić na wyjściu dekodera. Metody wykorzystywane wcześniej do kodowania, obecnie zajmują się buforowaniem znaków i kontekstów wejściowych, a także zbieraniem statystyk do występowania poszczególnych bitów w ramach określonego kontekstu.

Metoda do kodowania pojedynczego znaku:

```
public final void codeSymbol(int bit, int context) {  
  
    // Buffer symbol and context  
    if (pointer >= symbolBuffer.length) {  
        resizeBuffer();  
    }  
    symbolBuffer[pointer] = bit;  
    contextBuffer[pointer] = context;  
    pointer++;  
  
    // Update statistics  
    totalCount[context]++;  
    if (bit == 1)  
        oneCount[context]++;  
}
```

Nie wiadomo z góry ile będzie łącznie symboli wejściowych, oraz wartość ta może się zmieniać w zależności od bloku danych, ponadto stosowanie klasy `List` do przetrzymywania bitów ma bardzo negatywny wpływ na szybkość kodera, dlatego zastosowane zostały zwykłe tablice `symbolBuffer` i `contextBuffer` do przechowywania danych, a w przypadku, gdy bufor zostanie przepełniony, wywoływana jest funkcja `resizeBuffer` służąca do zwiększenia długości bufora.

Oryginalna metoda `codeSymbol` wyglądała następująco:

```

public final void codeSymbol(int bit, int context) {
    int q;
    int li;
    int la;
    int n;
    li = I[context];
    q=qe[li];
    if(bit==mPS[context]) {
        a -= q;
        if(a>=0x8000){
            c += q;
        } else {
            if(a<q) {
                a = q;
            } else {
                c += q;
            }
            I[context]=nMPS[li];
            a <<= 1;
            c <<= 1;
            cT--;
            if(cT==0) {
                byteOut();
            }
        }
    }
    } else {
        la = a;
        la -= q;
        if(la<q) {
            c += q;
        } else {
            la = q;
        }
        if(switchLM[li]!=0) {
            mPS[context] = 1-mPS[context];
        }
        I[context]=nLPS[li];
        n = 0;
        do {
            la <<= 1;
            n++;
        } while (la<0x8000);
        if (cT>n) {
            c <<= n;
            cT -= n;
        }
    }
}

```

```

    } else {
        do {
            c <<= cT;
            n -= cT;
            byteOut();
        } while (cT <= n);
        c <<= n;
        cT -= n;
    }
    a = la;
}
}

```

W oryginalnej metodzie codeSymbol w pierwszej kolejności modyfikowane są wartości rejestrów *a* i *c* które przechowują wartość obecnego przedziału. Potem aktualizuje się model prawdopodobieństw zgodnie z przejściami opisanymi w tablicach *nMPS* i *nLPS*, a także wylicza parametry renormalizacji w przypadku przekroczenia przez rejestry dopuszczalnego zakresu. W przypadku gdy wartość rejestru *cT* spadnie do 0, przenoszony jest pojedynczy bajt na wyjście kodera. Wszystkie te operacje wykonywane są w metodzie służącej do kodowania znaku, co przekłada się spadek wydajności. W zmodyfikowanej metodzie korzystającej ze stabilizowanych wyników wykonywana jest znacznie znacznie mniejsza liczba operacji.

Metoda codeSymbols służy do kodowania całej grupy bitów, stosowanie tej metody zmniejszy liczbę wywołań funkcji. Metoda fastCodeSymbols służy do kodowania ciągu takich samych bitów, które mają przypisane takie same konteksty.

```

public final void codeSymbols(int[] bits, int[] cX, int n) {
    for (int i = 0; i < n; i++)
    {
        // Buffer symbol and context
        if (pointer >= symbolBuffer.length) {
            resizeBuffer();
        }
        symbolBuffer[pointer] = bits[i];
        contextBuffer[pointer] = cX[i];
        pointer++;

        // Update statistics
        totalCount[cX[i]]++;
        if (bits[i] == 1)
            oneCount[cX[i]]++;
    }
}

```

```

public final void fastCodeSymbols(int bit, int ctxt, int n) {
    for (int i = 0; i < n; i++)
    {
        // Buffer symbol and context
        if (pointer >= symbolBuffer.length) {
            resizeBuffer();
        }
        symbolBuffer[pointer] = bit;
        contextBuffer[pointer] = ctxt;
        pointer++;
    }

    // Update statistics
    totalCount[ctxt] += n;
    if (bit == 1)
        oneCount[ctxt] += n;
}

```

Samo kodowanie odbywa się w metodzie `terminate`, która jest zawsze wywoływana na zakończenie kodowania bloku danych. Funkcja `terminate` zwraca liczbę zapisanych do tej pory bajtów. Na początku wyliczane są modele prawdopodobieństw w metodzie `calculateProbabilities`, później wszystkie zbuforowane wcześniej symbole są kodowane w odwrotnej kolejności. Po przetworzeniu już ostatniego znaku w bloku danych, wartości przechowywane w rejestrach zwracane są na koniec strumienia. Ponieważ zakodowane dane muszą być jeszcze odwrócone na sam koniec, więc wykorzystywany jest bufor pośredni, na którym działają metody: `bitsOut` i `flushToOutput`. Następnie dodawane są na samym początku właściwego wyjściowego strumienia informacje o modelach prawdopodobieństwa. Na sam koniec bufor pośredni i odwracany w metodzie `getReversedBuffer`, a w funkcji `addAdditionalBits` wstawiane są dodatkowe zerowe bity i końcowe dane wstawiane są do wyjściowego bufora. Żeby więc otrzymać liczbę zapisanych przez `terminate` bajtów należy dodać do siebie liczbę bajtów wstawionych przez `addAdditionalBits` (cały zakodowany blok) oraz bajty przechowujące informacje o modelach (19 bajtów).

```

public int terminate() {

    calculateProbabilities();

    // Encoding of the all buffered symbols. Encoding is performed in reverse order
    for (int i = pointer - 1; i >= 0; i--)
    {
        int c = contextBuffer[i];
        int s = (symbolBuffer[i] == mPS[c] ? 1 : 0);

        int value = coderLookupTable[I[c] * STATE_RANGE * 2 + (state - STATE_RANGE) * 2 + s];

        int nos = (value >>> 8) & 0xFF;
        int sh = value & 0xFF;
        state = (value >>> 16) & 0xFFFF;

        // Output bits if needed
        if (nos > 0)
        {
            bitsOut(sh, nos);
        }
    }

    // Flushes state and bit buffer to output stream
    flushToOutput();

    // Add states to code stream
    for (int i = 0; i < I.length; i++)
    {
        out.write( (b: I[i] | (mPS[i] << 7));
    }

    // Reverse bit stream and shift it to the beginning of the first byte
    byte[] reversedBuffer = getReversedBuffer();

    // Add additional 0 bit after every 0xFF byte
    return addAdditionalBits(reversedBuffer) + I.length;
}

```

Oryginalna metoda terminate zajmowała się jedynie tym, by odpowiednio zakończyć kodowanie pojedynczego bloku danych, a więc dodaniem do strumienia wystarczającej liczby bitów z rejestrów tak, by móc później poprawnie zdekodować cały blok. Stosowany jest jeden z kilku algorytmów służący do kończenia kodowania i który to określany jest w parametrze.

```

public int terminate() {
    switch (ttype) {
        case TERM_FULL:
            int tempc = c+a;
            c = c|0xFFFF;
            if(c>=tempc) {
                c = c-0x8000;
            }
            int remainingBits = 27-cT;
            do {
                c <<= cT;
                if(b!=0xFF) {
                    remainingBits -= 8;
                } else {
                    remainingBits -= 7;
                }
                byteOut();
            } while(remainingBits>0);
            b |= (1<<(-remainingBits))-1;
            if (b==0xFF) { // Delay 0xFF bytes
                delFF = true;
            } else {
                if (delFF) {
                    out.write( b: 0xFF);
                    delFF = false;
                    nrOfWrittenBytes++;
                }
                out.write(b);
                nrOfWrittenBytes++;
            }
            break;
        case TERM_PRED_ER:
        case TERM_EASY:
            int k; // number of bits to push out
            k = (11-cT)+1;
            c <<= cT;
            for (; k>0; k-=cT,c<<=cT) {
                byteOut();
            }
            if (k<0 && ttype==TERM_EASY) {
                b |= (1<<(-k))-1;
            }
            byteOut(); // Push contents of byte buffer
            break;
    }
}

```



```

case TERM_NEAR_OPT:
    int cLow;
    int cUp;
    int bLow;
    int bUp;
    cLow = c;
    cUp = c+a;
    bLow = bUp = b;
    cLow <= cT;
    cUp <= cT;
    if ((cLow & (1<<27))!=0) { // Carry bit in cLow
        if (bLow==0xFF) {
            delFF = true; // delay 0xFF
            bLow = cLow>>>20;
            bUp = cUp>>>20;
            cLow &= 0xFFFFFF;
            cUp &= 0xFFFFFF;
            cLow <= 7;
            cUp <= 7;
        } else { // we can propagate carry bit
            bLow++; // propagate
            cLow &= ~(1<<27); // reset carry in cLow
        }
    }
    if ((cUp & (1<<27))!=0) {
        bUp++; // propagate
        cUp &= ~(1<<27); // reset carry
    }
    while(true) {
        if(delFF) { // If delayed 0xFF
            if (bLow<=127 && bUp>127) break;
            out.write( b: 0xFF);
            nrOfWrittenBytes++;
            delFF = false;
        } else { // No delayed 0xFF
            if (bLow<=255 && bUp>255) break;
        }
        if (bLow<255) {
            if (nrOfWrittenBytes>=0) out.write(bLow);
            nrOfWrittenBytes++;
            bUp -= bLow;
            bUp <= 8;
            bUp |= (cUp >>> 19) & 0xFF;
            bLow = (cLow>>> 19) & 0xFF;
        }
    }
}

```

```

        cLow &= 0x7FFFF;
        cUp  &= 0x7FFFF;
        cLow <<= 8;
        cUp  <<= 8;
    } else { // bLow = 0xFF
        delFF = true;
        bUp -= bLow;
        bUp <<= 7;
        bUp |= (cUp >> 20) & 0x7F;
        bLow = (cLow >> 20) & 0x7F;
        cLow &= 0xFFFFF;
        cUp  &= 0xFFFFF;
        cLow <<= 7;
        cUp  <<= 7;
    }
}
break;
default:
    throw new Error("Illegal termination type code");
}
int len;
len = nrOfWrittenBytes;
a = 0x8000;
c = 0;
b = 0;
cT = 12;
delFF = false;
nrOfWrittenBytes = -1;
return len;
}

```

W metodzie `calculateProbabilities` obliczane są dla każdego kontekstu modele prawdopodobieństw, które będą wykorzystane podczas kodowania z danym kontekstem. W tym celu wykorzystane były zebrane wcześniej statystyki. Po czym wszystkie zbuforowane wcześniej symbole kodowane są w odwrotnej kolejności z wykorzystaniem wygenerowanych wcześniej tablic kodowania. W razie potrzeby bity zwracane są na wyjście wywołując `bitsOut`. Po ostatnim zakodowanym bicie stan kodera, a także jego bufor bitowy przesyłane na wyjście. Żeby odpowiadający dekodery mógł poprawnie zdekodować dane, aktualne stany kontekstów dodawane są na początku zakodowanego bloku danych. A na koniec ciąg wyjściowy jest odwracany i dodawane są dodatkowe bity 0, tak aby zakodowanym strumieniu danych nie pojawiały się znaczniki standardu JPEG 2000.

```

private void calculateProbabilities() {
    for (int ctxt = 0; ctxt < I.length; ctxt++)
    {
        // It doesn't update state of the uniform context
        if (I[ctxt] != 46)
        {
            // Calculate LPS probabilities and set MPS for a context
            double p;
            if (totalCount[ctxt] == 0)
            {
                mPS[ctxt] = 0;
                p = 0.5;
            }
            else
            {
                if ((oneCount[ctxt] << 1) > totalCount[ctxt])
                {
                    mPS[ctxt] = 1;
                    p = ((double) (totalCount[ctxt] - oneCount[ctxt])) / ((double) totalCount[ctxt]);
                }
                else
                {
                    mPS[ctxt] = 0;
                    p = ((double) oneCount[ctxt]) / ((double) totalCount[ctxt]);
                }
            }

            // Find the probability model that is the closest to counted probability
            int minState = -1;
            double minDifference = 1.1;
            for (int j = 0; j < 46; j++)
            {
                double diff = Math.abs(probabilities[j] - p);
                if (diff < minDifference)
                {
                    minDifference = diff;
                    minState = j;
                }
            }

            I[ctxt] = minState;
        }
    }
}

```

Metoda bitsOut zajmuje się wysyłaniem bitów do bufora bitowego i tworzeniem tam pełnych bajtów, które następnie zwracane są do pośredniego bufora bajtowego. W oryginalnej implementacji nie było potrzeby tworzenia metody do zbierania pojedynczych bitów, ponieważ w przypadku wartość rejestru cT spadała do 0, cały bajt był pobierany z rejestru c , przechowującego dolną granicę przedziału.

```

private void bitsOut(int bits, int count) {
    int nrOfBitsToWrite = count;
    int bitsToWrite = bits;

    // Go in the loop until all bits has been written
    while(nrOfBitsToWrite > 0)
    {
        // Calculate number of bits that can be written in this iteration
        int writtenInCurrent = Math.min(32 - nrOfBits, nrOfBitsToWrite);

        // Write bits to bit buffer
        bitsBuffer >>= writtenInCurrent;
        bitsBuffer |= bitsToWrite << (32 - writtenInCurrent);

        bitsToWrite >>= writtenInCurrent;
        nrOfBits += writtenInCurrent;
        nrOfBitsToWrite -= writtenInCurrent;

        // If the bit buffer is full flush it to output
        if (nrOfBits == 32)
        {
            for (int i = 0; i < 4; i++)
            {
                outBuffer.write( b: (bitsBuffer >>> (i * 8)) & 0xFF);
            }
            nrOfBits = 0;
            bitsBuffer = 0;
        }
    }
}

```

Znaczniki standardu JPEG 2000 mieszczą się w granicach 0xFF90 – 0xFFFF. W celu zapewnienia, aby wartości z tego przedziału nie znajdowały się w zakodowanym bloku danych, dodawany jest bit 0, po każdym wykrytym bajcie 0xFF. Odpowiedzialna jest za to metoda `addAdditionalBits`.

```

private int addAdditionalBits(byte[] output) {
    int nrOfWrittenBytes = 0;

    int nrOfCarried = 0;
    int carry = 0;
    for (byte b : output) {
        int temp = carry | ((b & 0xFF) >>> nrOfCarried);
        carry = (b << (8 - nrOfCarried)) & 0xFF;
        if (temp == 0xFF) {
            nrOfCarried++;
            carry = (carry >>> 1) & 0x7F;
        }

        nrOfWrittenBytes++;
        out.write(temp);
        if (nrOfCarried == 8) {
            nrOfWrittenBytes++;
            out.write(carry);
            nrOfCarried = 0;
            carry = 0;
        }
    }

    if (nrOfCarried != 0)
    {
        nrOfWrittenBytes++;
        out.write(carry);
    }

    return nrOfWrittenBytes;
}

```

MQDecoder

Podobnie jak w koderze zaczynamy od utworzenia tablic dekodowania dla każdego modelu prawdopodobieństwa. Jednak podczas generowania tablicy dekodowania, wykorzystywana jest stworzona wcześniej tablica do kodowania. Początkowo tablica do dekodowania przechowywana była w trójwymiarowej tablicy, jednak podobnie jak z kodowaniem, pojawiły się problemy dotyczące szybkości tego rozwiązania, więc w tym przypadku również wszystko przetrzymywane jest w tablicy jednowymiarowej, a trzy wartości pobierane podczas każdej iteracji dekodowania łączone są bitowo ze sobą i przechowywane jako liczba całkowitoliczbowa.

Metoda `getDecoderLookupTable` pobiera tablicę kodowanie z klasy `MQCoder` i na jej podstawie tworzy odpowiadającą mu tablicę dekodowania.

```
public static int[] getDecoderLookupTable() {
    int[] decoderLookupTable = new int[47 * STATE_RANGE];
    int[] coderLookupTable = MQCoder.coderLookupTable;
    for (int i = 0; i < 47; i++)
    {
        for (int j = 0; j < STATE_RANGE; j++)
        {
            for (int k = 0; k < 2; k++)
            {
                int coderValue = coderLookupTable[i * STATE_RANGE * 2 + j * 2 + k];
                int nextState = ((coderValue >>> 16) & 0xFFFF) - STATE_RANGE;
                int nrOfShifted = (coderValue >>> 8) & 0xFF;
                int decoderIndex = i * STATE_RANGE + nextState;
                decoderLookupTable[decoderIndex] = (k & 0xFF) |
                    (((j + STATE_RANGE) >> nrOfShifted) & 0xFFFF) << 8 |
                    ((nrOfShifted & 0xFF) << 24);
            }
        }
    }

    return decoderLookupTable;
}
```

Dzięki temu, że odwrócenie ciągu bitowego miało już miejsce w koderze, nie ma już potrzeby buforować danych w dekodерze. Metoda służąca do dekodowania pojedynczego znaku `decodeSymbol` pobiera wyniki operacji kodowania z wygenerowanej wcześniej tablicy.

```
public final int decodeSymbol(int context) {

    // Decode symbol
    int value = decoderLookupTable[I[context] * STATE_RANGE + (state - STATE_RANGE)];
    int tempState = (value >>> 8) & 0xFFFF;
    int shift = (value >>> 24) & 0xFF;
    if (shift > 0)
        tempState = (tempState << shift) | getBits(shift);
    int symbol = (value & 0xFF) == 1 ? mPS[context] : (1 - mPS[context]);
    state = tempState;
    return symbol;
}
```

W oryginalnej metodzie `decodeSymbol` modyfikowane są wartości rejestrów *a* i *c* które przechowują wartość obecnego przedziału, a także aktualizuje się model prawdopodobieństw zgodnie z przejściami opisanymi w tablicach *nMPS* i *nLPS* oraz wylicza parametry renormalizacji w przypadku przekroczenia przez rejestry dopuszczalnego zakresu. W przypadku gdy wartość rejestru *cT* spadnie do 0, przenoszony jest pojedynczy bajt na początek rejestru *c* dekodera. Wszystkie te operacje wykonywane są w metodzie służącej do dekodowania znaku, co przekłada się spadek wydajności. W zmodyfikowanej metodzie korzystającej ze stabilizowanych wyników wykonywana jest znacznie znacznie mniejsza liczba operacji.

```
public final int decodeSymbol(int context){
    int q;
    int la;
    int index;
    int decision;
    index = I[context];
    q = qe[index];
    a -= q;
    if ((c>>>16) < a) {
        if(a >= 0x8000){
            decision = mPS[context];
        }
        else {
            la = a;
            if(la >= q){
                decision = mPS[context];
                I[context] = nMPS[index];
                if(cT==0)
                    byteIn();
                la<<=1;
                c<<=1;
                cT--;
            }
            else{
                decision = 1-mPS[context];
                if(switchLM[index]==1)
                    mPS[context] = 1-mPS[context];
                I[context] = nLPS[index];
                do{
                    if(cT==0)
                        byteIn();
                    la<<=1;
                    c<<=1;
                    cT--;
                }
            }
        }
    }
}
```

```

        }while(la < 0x8000);
    }
    a = la;
}
else {
    la = a;
    c -= (la<<16);
    if(la < q){
        la = q;
        decision = mPS[context];
        I[context] = nMPS[index];
        if(cT==0)
            byteIn();
        la<<=1;
        c<<=1;
        cT--;
    }
    else {
        la = q;
        decision = 1-mPS[context];
        if(switchLM[index] == 1)
            mPS[context] = 1-mPS[context];
        I[context] = nLPS[index];
        do{
            if(cT==0)
                byteIn();
            la<<=1;
            c<<=1;
            cT--;
        } while (la < 0x8000);
    }
    a = la;
}
return decision;
}

```

Metoda decodeSymbols służy do dekodowania większej liczby bitów, które następnie umieszczają w tablicy.


```

public final void decodeSymbols(int[] bits, int[] cX, int n) {
    for (int j = 0; j < n; j++)
    {
        // Decode symbol
        int value = decoderLookupTable[I[cX[j]] * STATE_RANGE + (state - STATE_RANGE)];
        int tempState = (value >>> 8) & 0xFFFF;
        int shift = (value >>> 24) & 0xFF;
        if (shift > 0)
            tempState = (tempState << shift) | getBits(shift);
        int symbol = (value & 0xFF) == 1 ? mPS[cX[j]] : (1 - mPS[cX[j]]);
        state = tempState;

        bits[j] = symbol;
    }
}

```

W referencyjnej implementacji wykorzystywana była również metoda fastDecodeSymbols, służąca do dekodowania całej grupy symboli w trybie przyspieszonym.

```

public final boolean fastDecodeSymbols(int[] bits, int ctxt, int n) {

    for (int j = 0; j < n; j++)
    {
        // Decode symbol
        int value = decoderLookupTable[I[ctxt] * STATE_RANGE + (state - STATE_RANGE)];
        int tempState = (value >>> 8) & 0xFFFF;
        int shift = (value >>> 24) & 0xFF;
        if (shift > 0)
            tempState = (tempState << shift) | getBits(shift);
        int symbol = (value & 0xFF) == 1 ? mPS[ctxt] : (1 - mPS[ctxt]);
        state = tempState;
        bits[j] = symbol;
    }
    return false;
}

```

Podczas dekodowania potrzebne jest pobieranie pojedynczych bitów, a wejście dekodera jest buforem operującym tylko na pełnych bajtach. W związku z tym wymagane było dodanie bufora bitowego, który zapełnia się danymi z wejścia kodera, gdy tylko bufor zostanie opróżniony. Pobranie bitów z bufora obsługuje metoda getBits:

```

private int getBits(int count) {
    int countToRead = count;
    int bits = 0;

    // Go in a loop until all wanted bits has been read
    while(countToRead > 0)
    {
        // Calculate number of bits that can be read in the current iteration
        int c = Math.min(countToRead, nrOfBits);

        int b = bitBuffer >>> (32 - c);
        bitBuffer <<= c;
        nrOfBits -= c;
        countToRead -= c;
        bits = (bits << c) | b;

        // If bit buffer is already empty, fill it
        if (nrOfBits == 0)
        {
            for (int i = 0; i < 4; i++)
            {
                int readByte = in.read();
                if (readByte == -1)
                    break;
                nrOfBits += 8;
                bitBuffer |= readByte << ((3 - i) << 3);
            }

            // Delete extra 0 bits after every 0xFF byte
            dropExtraBits();
        }
    }
    return bits;
}

```

W oryginalnej implementacji nie ma odpowiadającej metody do `getBits`, ponieważ tam zostało to zaimplementowane tak, że najmniej znaczące 16 bitów rejestru `c` (dolna granica przedziału) służyło na wejściowy bufor bitowy. To znaczy, że w przypadku, gdy brakowało danych do przetworzenia pobierany był pojedynczy bajt z wejścia dekodera i wstawiany za właściwą wartością w rejestrze (na bardziej znaczącej pozycji). Oznacza to, że pobieranie pojedynczych bitów otrzymane jest poprzez przesunięcie bitowe rejestru.

Ponieważ po zakodowaniu dodawane bity 0 po każdym bajcie 0xFF, dlatego metoda `dropExtraBits` służy do usunięcia tych bitów z danych. Metoda ta usuwa dodatkowe bity tylko wewnątrz bufora bitowego, dlatego wywoływana jest ona po każdym zapełnieniu bufora.

```

private void dropExtraBits() {
    int tempBitBuffer = 0; // bit buffer after deleting extra bits
    int tempBitBufferCount = 0; // Number of bits written to tempBitBuffer

    // Iterates over bytes in the bit buffer
    for (int i = 3; i >= 0; i--)
    {
        int offset = i * 8; // Offset of the byte in the bit buffer

        // Check if previous byte was 0xFF
        if (ffByteDetected)
        {
            // Add all the bits except extra one to the bit buffer
            int tempA = (bitBuffer << 1) & (0xFE << offset);
            tempBitBuffer |= tempA << ((24 - offset) - tempBitBufferCount);
            tempBitBufferCount += 7;
        }
        else
        {
            // Add all the bits to the bit buffer
            int tempA = bitBuffer & (0xFF << offset);
            tempBitBuffer |= tempA << ((24 - offset) - tempBitBufferCount);
            tempBitBufferCount += 8;
        }

        // Check if the current byte == 0xFF
        ffByteDetected = ((bitBuffer >>> offset) & 0xFF) == 0xFF;
    }

    bitBuffer = tempBitBuffer;
    nrOfBits = tempBitBufferCount;
}

```

W implementacji CABAC nie ma metody dropExtraBits, zamiast tego detekcja sytuacji w której należy usunąć dodatkowy bit ze strumienia oraz samo ich usuwanie przebiega w metodzie byteIn służącej do pobierania bajtu z wejścia. Wygląda ona następująco:

```

private void byteIn(){
    if(!markerFound){
        if(b==0xFF){
            b=in.read()&0xFF; // Convert EOFs (-1) to 0xFF

            if(b>0x8F){
                markerFound=true;
                // software-convention decoder: c unchanged
                cT=8;
            }else{
                c += 0xFE00 - (b<<9);
                cT=7;
            }
        } else {
            b=in.read()&0xFF; // Convert EOFs (-1) to 0xFF
            c += 0xFF00 - (b<<8);
            cT=8;
        }
    }
    else {
        // software-convention decoder: c unchanged
        cT=8;
    }
}
}

```

Rozdział 6

Testowanie

Przeprowadzone zostały testy dotyczące poprawności implementacji, stopnia kompresji, a także wydajności.

Poprawność

Kodek akceptuje na wejście trzy formaty obrazów: PPM, PGM i PGX. W celu sprawdzenia poprawności implementacji wybrano trzy przypadkowe obrazy, które następnie zostały przekonwertowane na każdy z tych formatów. Przeprowadzono kolejno proces kompresji, a następnie dekompresji na każdym z nich, po czym porównano zdekompresowany obraz z tym przed kompresją. Testy te potwierdziły poprawność działania.

Kompresja

Następnie w celu porównania stopnia kompresji danych, wybrano 10 obrazów w formacie PPM, które następnie zostały skompresowane kodekiem oryginalnym oraz kodekiem wykorzystującym ANS. Po czym porównano wielkości plików. Zastosowana została kompresja stratna z 5 poziomami dekompozycji falkowej.

Obraz	CABAC (bajty)	CABAC (bpp)	ANS (bajty)	ANS (bpp)	Różnica
jezioro.ppm	31632	5.036	32081	5.108	1.4%
latarnia.ppm	265059	4.855	264984	4.853	-0.03%
tree.ppm	593661	6.801	587352	6.728	-1.07%
wildlife.ppm	272805	2.204	278919	2.254	2.19%
zachod.ppm	299414	2.495	301677	2.514	0.75%
panda.ppm	77795	1.779	78746	1.801	1.21%
lew.ppm	193373	1.653	198307	1.695	2.49%
balwan.ppm	357944	2.623	362909	2.659	1.37%
tygrys.ppm	375716	2.087	373602	2.076	-0.57%
natura.ppm	2381601	7.753	2363245	7.693	-0.78%
Suma:	4849000		4841822		-0.15%

Tabela 6.1: Porównanie wielkości skompresowanych plików

Z analizy wyników z tabeli 6.1 można wywnioskować, że stopień kompresji obu implementacji jest bardzo podobny. Zarówno ANS jak i CABAC w założeniu są algorytmami, które pozwalają na zbliżenie się do wartości entropii kodowanych danych, ponadto zmodyfikowany kodek korzysta z takich samych estymat. Stąd też podobne wyniki dla obu implementacji. Są jednak dwie główne różnice pomiędzy koderami, które miały znaczący wpływ na wyniki. Po pierwsze w implementacji ANS nie korzystano z adaptacji modeli prawdopodobieństwa dla poszczególnych kontekstów, a więc wszystkie bity z danego bloku

danych i kontekstu były kodowane przy użyciu takiego samego kontekstu. Po drugie w zmodyfikowanej implementacji na początku bloku danych dodawana jest informacja o użytych modelach prawdopodobieństw, tak aby dekodery mógł poprawnie zdekodować dane. Pierwsza z tych różnic zmniejszyła rozmiar skompresowanych obrazów. Spowodowane jest to tym, że korzystając z adaptacyjnych algorytmów, początkowe znaki bloku danych kodowane są z użyciem prawdopodobieństwa wyrażnie różnego od faktycznego rozkładu. Zgodnie z tabelą 6.1 różnica między implementacjami wynosi -0.15%. W przypadku zakodowania danych bez dodawania informacji o wykorzystanych modelach przy kodowaniu, różnica ta wzrosła do -2.81%. Dla każdego bloku kodowego, modele zajmują 19 bajtów w strumieniu.

Wydajność

Ostatnim rodzajem przeprowadzonych testów były testy wydajnościowe. Do zautomatyzowania testów stworzony został skrypt w języku Python. Wyniki w tabelach 6.2 oraz 6.3 są średnią ze 100 przebiegów.

Obraz	CABAC	ANS	Różnica
jezioro.ppm	70	79	11,39%
latarnia.ppm	204	224	8,93%
tree.ppm	496	542	8,49%
wildlife.ppm	521	548	4,93%
zachod.ppm	485	501	3,19%
panda.ppm	144	144	0%
lew.ppm	455	516	11,82%
balwan.ppm	590	617	4,38%
tygrys.ppm	807	838	3,7%
natura.ppm	1840	1974	6,79%
Suma:	5612	5983	6,2%

Tabela 6.2: Wyniki szybkości działania kodowania (w milisekundach)

Obraz	CABAC	ANS	Różnica
jezioro.ppm	55	70	21,43%
latarnia.ppm	194	194	0%
tree.ppm	457	427	-7,03%
wildlife.ppm	434	419	-3,58%
zachod.ppm	430	421	-2,14%
panda.ppm	102	114	10,53%
lew.ppm	368	358	-2,79%
balwan.ppm	515	490	-5,1%
tygrys.ppm	652	676	3,55%
natura.ppm	1729	1671	-3,47%
Suma:	4936	4840	-1,98%

Tabela 6.3: Wyniki szybkości działania dekodowania (w milisekundach)

Dla dekodowania zmodyfikowany kodek wykorzystujący ANS działa szybciej. Jest to spowodowane tym, że wszystkie wyniki operacji zostały umieszczone w tablicy. W przypadku kodowania, oryginalna implementacja używająca CABAC okazała się bardziej wydajna. Pomimo tego, że koder ANS również przechowywał wyniki w tablicy, to nie udało się tym razem przyspieszyć działania. Spowodowane jest to koniecznością buforowania znaków wejściowych przed ich przetworzeniem, a także drugiego buforowania na wyjściu kodera.

Złożoność obliczeniowa

W trakcie procesu kodowania wykorzystywane są następujące metody:

- `codeSymbol`, `codeSymbols`, `fastCodeSymbols` – wykorzystywane do wysyłania znaków na wejście kodera.
- `getNumCodedBytes` – może być wywołana wiele razy podczas kodowania bloku, służy do zwracania długości zakodowanego strumienia danych.
- `terminate` – wywoływana jeden raz na sam koniec. Służy do zakończenia kodowania danego bloku.
- `finishLengthCalculation` – wywoływana jest po `terminate`. Służy do obliczenia liczby bajtów potrzebnych do poprawnego odkodowania wszystkich znaków wysłanych na wejście kodera na dany moment.

Złożoności obliczeniowe dla kodowania CABAC:

Metody `codeSymbol`, `codeSymbols` i `fastCodeSymbols` korzystają z innej metody `byteOut` służącej do wysłania pojedynczego bajtu na wyjście. Jej działanie nie zależy od liczby znaków, a więc wykonuje się w czasie stałym: $O(1)$. Metoda `codeSymbol` również ma złożoność $O(1)$. Funkcja `codeSymbols` służy do kodowania całej grupy znaków, czas jej działania zależy od liczby bitów do zakodowania, wykonuje się w czasie liniowym: $O(n)$. Metoda `fastCodeSymbols` wykorzystywana jest do przetwarzania symboli w trybie przyspieszonym. Może być on jednak użyty tylko w szczególnych przypadkach, gdy nie są one spełnione, kodowanie przebiega w identyczny sposób jak w `codeSymbols`. Nawet w przypadku użycia, czas działania jest zależny od n . Dlatego złożoność `fastCodeSymbols` wynosi $O(n)$.

Działanie metody `getNumCodedBytes` zależy w głównej mierze od wybranego algorytmu obliczania długości zakodowanych danych. Jednak ich czas działania jest zawsze taki sam, wykonuje się w czasie stałym. Funkcja `terminate` w implementacji CABAC zajmuje się zwróceniem na wyjście wszystkich danych przechowywanych w aktualnym stanie kodera i niezależnie od stosowanego typu zakończenia danych wykonuje się w czasie stałym. Czas działania funkcji `finishLengthCalculation` jest zależny od liczby wywołań metody `getNumCodedBytes` dla danego bloku danych. Funkcja ta operuje na wartościach zwróconych przez `getNumCodedBytes` oraz przez `terminate`.

A więc złożoność kodera CABAC wynosi:

$$f(n, x) = n * O(1) + x * O(1) + O(1) + O(x+1) = O(n) + O(x)$$

gdzie:

n – liczba kodowanych znaków

x – liczba wywołań `getNumCodedBytes`

Wartość x zależy w dużej mierze od n . W najbardziej pesymistycznym przypadku metoda `getNumCodedBytes` będzie wywoływana po każdym zakodowanym znaku, a więc wyniesie n . Dlatego dla przypadku pesymistycznego:

$$f(n, x) = O(n) + O(x) = O(n) + O(n) = O(n)$$

Złożoności obliczeniowe dla kodowania ANS:

Metody do wysyłania znaków na wejście `codeSymbol`, `codeSymbols` i `fastCodeSymbols` mają kolejno złożoności: $O(1)$, $O(n)$ i $O(n)$. Funkcja `fastCodeSymbols` nie korzysta tutaj z trybu przyspieszonego, a więc działa podobnie jak `codeSymbols`. Metoda `getNumCodedBytes` w nowej implementacji zwraca jedynie wartość 0, a funkcja `finishLengthCalculation` podobnie jak implementacja w CABAC zależy od ilości wywołań `getNumCodedBytes`, a więc ich złożoności wynoszą $O(1)$, $O(x + 1)$.

Szybkość działania metody `terminate` zależy bezpośrednio od liczby znaków wejściowych, a więc ma złożoność $O(n)$.

A więc złożoność kodera CABAC wynosi:

$$f(n, x) = n * O(1) + x * O(1) + O(n) + O(x+1) = O(n) + O(x)$$

$$f(n) = O(n)$$

W trakcie procesu dekodowania wykorzystywane są następujące metody:

- `decodeSymbol`, `decodeSymbols`, `fastDecodeSymbols` – wykorzystywane do zwracania znaków z dekodera.
- `nextSegment` – metoda służąca do ładowania nowego bloku danych do zdekodowania

Złożoności obliczeniowe dla dekodowania CABAC:

Metody `decodeSymbol`, `decodeSymbols` i `fastDecodeSymbols` korzystają z innej metody `byteIn` służącej do wysłania pojedynczego bajtu do bufora dekodera. Jej działanie nie zależy od liczby znaków, a więc wykonuje się w czasie stałym: $O(1)$. Metoda `decodeSymbol` również ma złożoność $O(1)$. Funkcja `decodeSymbols` służy do dekodowania całej grupy znaków, czas jej działania zależy od liczby bitów do zakodowania, wykonuje się w czasie liniowym: $O(n)$. Metoda `fastDecodeSymbols` wykorzystywana jest do przetwarzania symboli w trybie prześpieszonym. Może być on jednak użyty tylko w szczególnych przypadkach, gdy nie są one spełnione, dekodowanie przebiega w identyczny sposób jak w `decodeSymbols`. Nawet w przypadku użycia, czas działania jest zależny od n . Dlatego złożoność

fastDecodeSymbols wynosi $O(n)$.

Metoda nextSegment ustawia nowy bufor, skąd pobierane będą dane, a następnie inicjuje ponownie dekodery. Jej złożoność obliczeniowa jest więc stała.

A więc złożoność dekodera CABAC wynosi:

$$f(n) = n * O(1) + O(1) = O(n)$$

Złożoności obliczeniowe dla dekodowania ANS:

Złożoności dla poszczególnych metod przy implementacji ANS są takie same jak przy dekodzie CABAC.

A więc złożoność dekodera ANS wynosi:

$$f(n) = n * O(1) + O(1) = O(n)$$

Rozdział 7

Podsumowanie

W ramach pracy inżynierskiej zaimplementowano asymetryczne kodowanie numeryczne, które następnie zastąpiło kodowanie CABAC wykorzystywane w referencyjnej implementacji kodeka obrazu opartego na standardzie JPEG 2000. Podmianie uległo zarówno kodowanie, jak i dekodowanie. Głównym celem była poprawność działania, co zostało osiągnięte. Uzyskany stopień kompresji jest zgodny z przewidywaniami. Różnica pod tym względem z referencyjną implementacją okazała się znikoma. Nie udało się również osiągnąć lepszej wydajności. Ocena szybkości działania wykazała niewielką poprawę dla dekodowania, jednak dla kodowania zaobserwowano spadek.

Proces pisania pracy dyplomowej umożliwił pozyskanie istotnej wiedzy w zakresie kompresji danych, w szczególności kompresji obrazów. Ponadto pozwolił na poprawienie umiejętności pracy z gotowym kodem, a podczas zapoznawania się z oryginalną implementacją, można było nauczyć się ogólnie stosowanych praktyk programistycznych.

Bibliografia

- [1] J. Duda, *Asymetric numeral systems : entropy coding combining speed of Huffman coding with compression rate of arithmetic coding* (2013)
- [2] D. Huffman, *A Method for the Construction of Minimum-Redundancy Codes* (1952)
- [3] G. G. Langdon Jr, *An Introduction to Arithmetic Coding* (1984)
- [4] M. Marcellin, D. Taubman, *JPEG2000 Image Compression Fundamentals, Standard and Practise* (2002)
- [5] A. Przelaskowski, *Kompresja danych: podstawy, metody bezstratne, kodery obrazów* (2005)
- [6] J. Zelenski, *Huffman Encoding and Data Compression* (2012)
- [7] JPEG and JPEG2000 comparison, <http://www.verypdf.com/pdfinfoeditor/jpeg-jpeg-2000-comparison.htm>
- [8] Amount of data statistics, <https://www.cm.com/blog/the-more-personalized-the-better-why-you-need-a-customer-data-platform/>
- [9] Photos statistics, <https://businessinsider.com.pl/international/people-will-take-12-trillion-digital-photos-this-year-thanks-to-smartphones/bwd8pxp>
- [10] CABAC diagram, https://www.researchgate.net/figure/CABAC-block-diagram-from-the-encoder-perspective-Binarization-context-modeling_fig1_290180658
- [11] JJ2000 software implementation, <https://github.com/Unidata/jj2000>

Spis rysunków

<u>2.1 Szacowana ilość informacji przechowywana na świecie.....</u>	<u>11</u>
<u>3.1 Przykładowe drzewo Huffmana.....</u>	<u>17</u>
<u>3.2 Proces tworzenia drzewa Huffmana dla przykładowego ciągu znaków.....</u>	<u>19</u>
<u>3.3 Przykład działania kodowania arytmetycznego.....</u>	<u>22</u>
<u>3.4 Schemat działania kodera CABAC.....</u>	<u>25</u>
<u>3.5 Rozkład liczb naturalnych w koderze uABS dla $P(1) = 0.3$.....</u>	<u>26</u>
<u>3.6 Działanie kodera tANS na przykładowym ciągu bitów.....</u>	<u>29</u>
<u>4.1 Porównanie jakości obrazu po kompresji JPEG oraz JPEG 2000.....</u>	<u>31</u>
<u>4.2 Schemat kodera JPEG 2000.....</u>	<u>32</u>
<u>4.3 Dyskretna transformata falkowa.....</u>	<u>34</u>
<u>4.4 Schemat kodera MQ.....</u>	<u>35</u>
<u>4.5 Schemat dekodera MQ.....</u>	<u>36</u>

Spis tabel

<u>2.1 Porównanie jakości i wielkości obrazu po kompresji przy użyciu standardu JPEG.....</u>	<u>15</u>
<u>3.1 Wystąpienia znaków dla kodowania Huffmana.....</u>	<u>19</u>
<u>3.2 Kody Huffmana dla każdego znaku.....</u>	<u>21</u>
<u>3.3 Prawdopodobieństwa znaków dla przykładowego ciągu.....</u>	<u>21</u>
<u>3.4 Działanie uABS dla przykładowego ciągu bitów.....</u>	<u>27</u>
<u>6.1 Porównanie wielkości skompresowanych plików.....</u>	<u>56</u>
<u>6.2 Wyniki szybkości działania kodowania (w milisekundach).....</u>	<u>57</u>
<u>6.3 Wyniki szybkości działania dekodowania (w milisekundach).....</u>	<u>57</u>