Politechnika Krakowska
Wydział Inżynierii Elektrycznej i Komputerowej
Katedra Informatyki Technicznej

Instrukcja do ćwiczeń laboratoryjnych
z przedmiotu:

Grafika Komputerowa i Multimedia

Proste algorytmy kompresji bezstratnej

Wstęp teoretyczny

Przez algorytmy proste, będziemy rozumieli takie, które nie dokonują analizy strumienia danych przed kompresją. Dzięki temu algorytmy te są po pierwsze proste, po drugie mają minimalne wymagania pamięciowe (możliwa jest np. bezproblemowa implementacja sprzętowa), ponieważ nie wymagają bufora danych. Oczywistą wadą tych algorytmów jest mniejszy stopień kompresji, w porównaniu do algorytmów analizujących.

Przykładami takich prostych algorytmów są **ByteRun**, zastosowany między innymi w formacie zapisu obrazów IFF ILBM w 1985 roku, oraz **RunLengthEncoding (RLE)** zastosowany w formacie BMP. Oba te proste algorytmy potrafią jedynie skompresować powtarzające się ciągi jednakowych znaków, stąd ich zastosowanie ogranicza się w praktyce do kompresji obrazów o niewielkiej (≤256) ilości kolorów.

Nieco bardzie zaawansowanym algorytmem jest **LZW**, nazwany tak od nazwisk twórców, Lempela Ziva, którzy opracowali podstawy teoretyczne, oraz Welsha, który w 1984 roku opisał implementację, sprzętową zresztą. Algorytm LZW potrafi wykorzystać nie tylko powtarzalność pojedynczych znaków, ale również ich różny zbitek, dzięki budowie tzw. słownika w czasie pracy algorytmu. Dzięki temu szczególnie dobrze LZW spisuje się przy kompresji tekstu, a zwłaszcza tekstów takich jak np. kody źródłowe programów. Co ciekawe słownik kodera nie musi być przesyłany wraz ze skompresowanymi danymi, bowiem dekoder jest w stanie go odtworzyć w czasie dekompresji. To dodatkowo podnosi efektywność metody.

Algorytm LZW znalazł szerokie zastosowanie w praktyce:

- Format zapisu obrazów GIF,
- Formaty zapisu dokumentów PostScript i PDF (kompresja bitmap),
- Programy do kompresji plików (ARC, compress).
- Transmisja modemowa (V.42bis, zmodyfikowany LZW).

Warto zaznaczyć, że przez dłuższy czas algorytm LZW był objęty patentem, co ograniczało jego stosowanie (patent zgłosiła firma Unisys, autor formatu GIF). Obecnie patent ten już nie obowiązuje.

Algorytm ByteRun

Algorytm ten daje kompresję tylko i wyłącznie w obecności powtarzających się symboli. Z założenia algorytm ten operuje na symbolach o rozmiarze bajtu.

W algorytmie tym mamy do czynienia z dwoma sekwencjami:

- sekwencja kopiowania bajtów:

n	b0	b1	b2	b3	b4	b5	...	b(n)

Pierwszy bajt, to liczba n z zakresu <0, 127>, określa ona ilość następujących n+1 bajtów, które mają być przy dekompresji skopiowane bez zmian.

- sekwencja powtarzania bajtów

n	b

n jest z przedziału <-1, -127>, co łatwo poznać po ustawionym najstarszym bicie. Bajt b jest przy dekompresji powtarzany (-n+1) razy, co pozwala na liczbę powtórzeń od 2 do 128. Pozostała, nieprzydzielona wartość n = -128, jest traktowana jako kod pusty (no-op).

Warto zauważyć że dla wielu strumieni danych kompresja ByteRun może dać zysk ujemny, na przykład w przypadku sekwencji ABABABABAB... „skompresowany" strumień będzie o 1 bajt dłuższy niż oryginał.

Algorytm ByteRun stosowany był do kompresji obrazków w formacie IFF ILBM.

Przykład działania

Oto linia czterokolorowego obrazka, którą poddamy kompresji ByteRun:

Nadajmy numery kolorom z palety:

0 = niebieski **1 = czerwony** **2 = zielony** **3 = szary**

Pierwsze powtórzenie 3 niebieskich pikseli to oczywiście sekwencja [-2, 0], następnie 4 czerwone to [-3, 1]. Następnie mamy sytuację szczególną, mianowicie dwa piksele tego samego koloru (niebieskie) będą miały przed sobą (zielony piksel) i za sobą (szary, czerwony, szary) sekwencje kopiowania. W takim przypadku bardziej opłaca się „po całości" dać jedną sekwencję kopiowania. Oto dowód: klasycznie mielibyśmy: [0, 2] (kopiowanie zielonego piksela), [-1, 0] (dwa niebieskie) i [2, 3, 1, 3] (kopiowanie 3 pikseli), łącznie 8 bajtów. Zapisanie tego jako jednej sekwencji kopiowania daje nam [5, 2, 0, 0, 3, 1, 3] – o jeden bajt mniej. Dalej prosto, dwa zielone [-1, 2], trzy niebieskie [-2, 0], cztery szare [-3, 3], później jest 7 pikseli do skopiowania [6, 1, 2, 1, 2, 3, 1, 2], dwa niebieskie [-1, 0], trzy czerwone [-2, 1] i dwa szare [-1, 3].

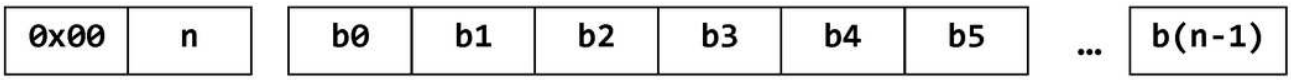
Ostatecznie z 36 pikseli otrzymaliśmy ciąg **[-2, 0, -3, 1, 5, 2, 0, 0, 3, 1, 3, -1, 2, -2, 0, -3, 3, 6, 1, 2, 1, 2, 3, 1, 2, -1, 0, -2, 1, -1, 3]** składający się z 31 bajtów – kompresja zmniejszyła rozmiar danych do 86,1 % oryginału.

Dekodowanie jest bardzo proste, badamy znak liczby, jeżeli dodatni, to kopiujemy na wyjście następne n+1 bajtów, jeżeli ujemny, to powtarzamy następny bajt -n+1 razy (chyba, że n=-128, wtedy pomijamy ten bajt).

Algorytm RLE (Run Length Encoding)

Idea tego algorytmu jest identyczna jak ByteRun – eliminacja redundancji informacyjnej w postać powtarzających się bajtów. Jest jednak kilka różnic, przede wszystkim założeniem jest przetwarzanie danych po 2 bajty, oprócz tego RLE posiada kilka sekwencji sterujących umożliwiających przeskakiwanie większych bloków danych.

Sekwencja kopiowania danych wygląda następująco:



Gdzie n przyjmuje wartości od 3 do 255. Warto zauważyć, że jeden bajt lub dwa bajty muszą być zapisane jako jedna lub dwie sekwencje powtarzania.

Sekwencja powtarzania wygląda następująco:

Gdzie n jest dowolną liczbą od 1 do 255, a b powtarzanym bajtem.

Pozostały nam jeszcze sekwencje specjalne. Sekwencje te są specyficzne dla formatu BMP i używane są przy kompresji maski przezroczystości:

- `[0x00, 0x00]` oznacza koniec linii obrazu,
- `[0x00, 0x01]` to koniec całego obrazu,
- `[0x00, 0x02, x, y]` to przesunięcie się w obrazie o wektor [x, y]. Przeskakiwane dane są traktowane jako przezroczyste.

RLE jest generalnie nieco mniej efektywny niż ByteRun, choć dzięki przetwarzaniu po 2 bajty, może być nieco szybszy. Na współczesnych procesorach jednak, różnica nie jest specjalnie widoczna.

Przykład działania

Kompresją RLE potraktujemy tę samą linię, co poprzednio.

Zaczynamy od sekwencji powtarzania: [3, 0], [4, 1]. Następnie mamy drobny dylemat, czy dwa niebieskie zapisać jako powtórzenie, czy nie...

Przy powtórzeniu będziemy mieli [1, 2], [2, 0], i później albo 6 bajtów na powtórzeniach, albo też 6 na sekwencji kopiowania ([0, 3, 3, 1, 3, 0], końcowe, zbędne zdawałoby się, zero jest skutkiem założenia przetwarzania danych 2-bajtowymi porcjami, każda sekwencja musi mieć parzystą liczbę bajtów). Tak czy tak – 10 bajtów.

Jedna długa sekwencja kopiowania [0, 6, 2, 0, 0, 3, 1, 3] ma tylko 8 bajtów. Następnie mamy proste sekwencje powtarzania [2, 2], [3, 0] i [4, 3], po niej zaś sekwencję skopiowania 7 bajtów [0, 7, 1, 2, 1, 2, 3, 1, 2, 0]. I znów powtórzenia [2, 0], [3, 1] i [2, 3].

Ostatecznie otrzymujemy 34-bajtową sekwencję **[3, 0, 4, 1, 0, 6, 2, 0, 0, 3, 1, 3, 2, 2, 3, 0, 4, 3, 0, 7, 1, 2, 1, 2, 3, 1, 2, 0, 2, 0, 3, 1, 2, 3]**. To więcej niż dla ByteRun (winne jest tu przede wszystkim 2-bajtowe wejście do sekwencji kopiowania), skompresowany ciąg ma długość 94,4% oryginału.

Dekodowanie RLE nie nastręcza większych trudności. Typowe obrazki BMP nie zawierają sekwencji specjalnych zaczynających się od 0, a wyłącznie sekwencje powtarzania i kopiowania. Należy jedynie pamiętać o tym, żeby pominąć nadmiarowe zera wyrównujące sekwencje kopiowania do parzystej liczby bajtów.

Kompresja RLE posiada również odmianę RLE4 przeznaczoną do kompresji obrazków 16-kolorowych (w bajcie mieszczą się 2 piksele), kompresji ulegają wtedy nie tylko ciągi jednakowych pikseli, ale również rastry typu ABABABAB, często używane do symulacji większej ilości kolorów.

Algorytm LZW

Przed omówieniem algorytmu kompresji LZW, zdefiniujmy dwa pojęcia, szeroko używane w teorii kompresji bezstratnej.

Alfabet

Alfabet danego kompresora, to zestaw wszystkich znaków, jakie mogą się pojawić na jego wejściu (w danych nieskompresowanych). Przykładowo dla 4-kolorowego obrazka alfabet to `[0, 1, 2, 3]`, dla tekstu zaś alfabet składał się będzie z użytego zestawu znaków. W danym zastosowaniu kompresora alfabet jest najczęściej stały i znany z góry.

Słownik

Słownik kompresora to tablica (lub inna struktura) zawierająca fragmenty wiadomości wejściowej i przypisane im kody na wyjściu kompresora. Słownik z reguły zależy od treści kompresowanej wiadomości i powstaje dynamicznie w czasie kompresji (wyjątkiem są algorytmy RLE i ByteRun, gdzie słownik jest stały i znany z góry). Niektóre algorytmy kompresji wymagają przesłania słownika w skompresowanej wiadomości, inne (jak np. LZW) są w stanie dynamicznie odtworzyć słownik w czasie dekompresji.

Inicjalizacja słownika w algorytmie LZW polega na:

1. Określeniu ilości bitów niezbędnych do zakodowania wszystkich symboli z alfabetu.
2. Wstawieniu całego alfabetu do słownika.

Ponieważ alfabet jest znany zarówno przez kompresor jak i dekompresor, nie zachodzi potrzeba przesyłania części słownika zawierającej alfabet. Pozostała część słownika (zawierająca symbole złożone) jest odtwarzana przez dekoder na bieżąco, więc również nie trzeba jej przesyłać.

Przykład działania

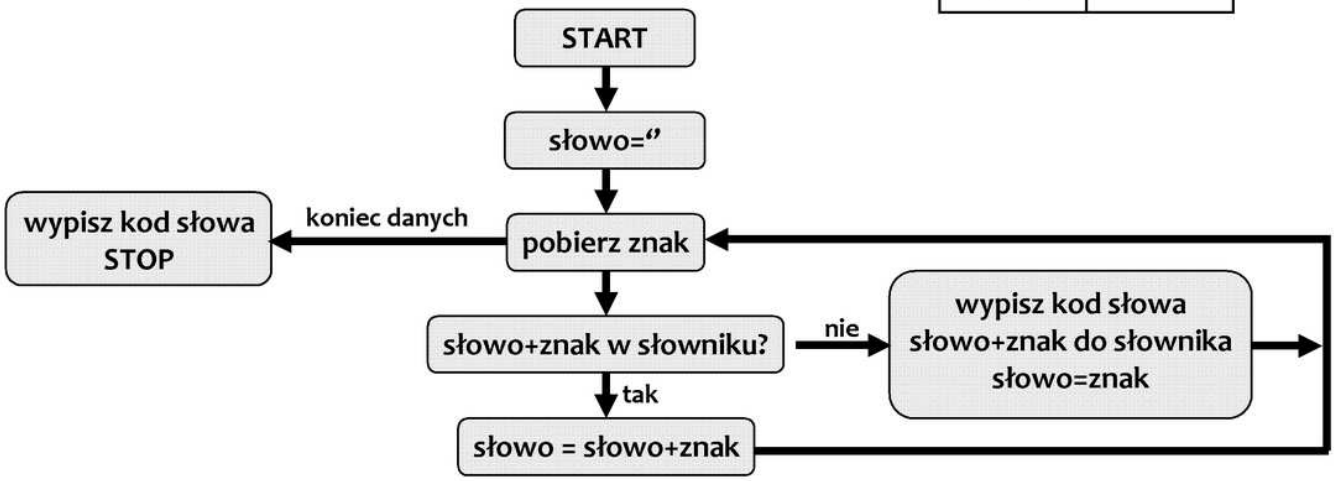
Algorytmem LZW skompresujemy tę samą linię obrazu, która posłużyła nam przy poprzednich algorytmach.

Alfabetem będą tu poszczególne kolory, zakodowane tak jak poprzednio [0, 1, 2, 3]. Do zakodowania alfabetu wystarczą nam 2 bity. Tak więc pierwsze wolne miejsce w słowniku będzie miało kod 4.

Oto słownik na starcie pracy algorytmu:

	0
	1
	2
	3

W czasie pracy algorytm wykonuje sekwencję czynności. Dane są przetwarzane znak po znaku, a więc nie jest wymagany swobodny dostęp do dowolnego elementu kompresowanej wiadomości.



znak	słowo	słownik?	do słownika	na wyjście
		 tak		
		 nie	 4	0
		 tak		
		 nie	 5	4
		 nie	 6	1
		 tak		
		 nie	 7	6
		 nie	 8	1
		 nie	 9	2
		 tak		
		 nie	 10	4
		nie	11	3
		nie	12	1
		nie	13	3
		nie	14	2
		tak		
		nie	15	9
		tak		
		tak		
		nie	16	10
		nie	17	3
		tak		
		nie	18	17
		tak		
		nie	19	8
		tak		

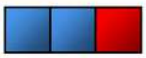
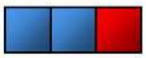
znak	słowo	słownik?	do słownika	na wyjście
		 nie	 20	8
		 tak		
		 nie	 21	11
		 tak		
		 tak		
		 nie	 22	15
		 tak		
		 tak		
		 nie	 23	7
		 tak		
koniec				17

Na wyjście został wyemitowany ciąg **[0, 4, 1, 6, 1, 2, 4, 3, 1, 3, 2, 9, 10, 3, 17, 8, 8, 11, 15, 7, 17]**, liczący sobie 21 bajtów. Ciąg został skompresowany do 58,3% oryginalnej wielkości.

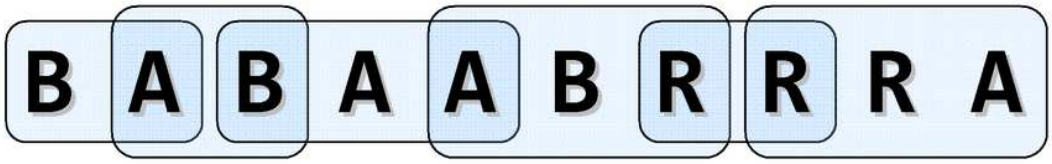
W praktyce ogranicza się wielkość słownika tak, że najdłuższy kod zajmuje 12 bitów, co daje 4096 elementów słownika. Dalsza kompresja przebiega tak samo, ale do słownika nie są już dopisywane kolejne pozycje.

Najczęściej też kody skompresowane zapisywane są na polach bitowych o minimalnej długości, co dodatkowo w istotny sposób polepsza kompresję, szczególnie dla małych alfabetów.

Przykład działania dekompresji

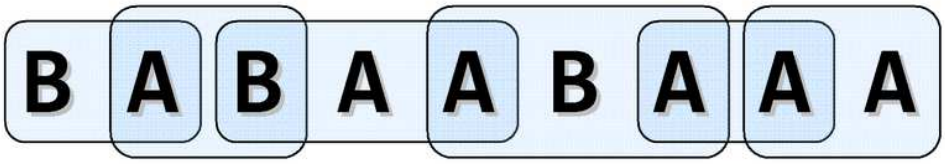
na wejściu	w słowniku?	słowo	do słownika	na wyjście
0	tak			
4	nie		4 	
1	tak		5 	
6	nie		6 	
1	tak		7 	
2	tak		8 	
4	tak		9 	
3	tak		10 	
1	tak		11 	
3	tak		12 	
2	tak		13 	
9	tak		14 	
10	tak		15 	
3	tak		16 	
17	nie		17	
8	tak		18	
8	tak		19	
11	tak		20	
15	tak		21	
7	tak		22	
17	tak		23	
koniec				

Inny przykład działania kompresji



- B jest w słowniku. BA nie znajduje się w słowniku; dodajemy BA do słownika, a na wyjście generujemy B
- AB nie znajduje się w słowniku; dodajemy AB do słownika, a na wyjście generujemy A
- BA jest w słowniku. BAA nie znajduje się w słowniku; dodajemy BAA do słownika, a na wyjście generujemy BA
- AB jest w słowniku. ABR nie znajduje się w słowniku; dodajemy ABR do słownika, na wyjście generujemy AB
- RR nie znajduje się w słowniku; dodajemy RR do słownika, na wyjście generujemy R
- RR jest w słowniku. RRA nie znajduje się w słowniku; dodajemy RRA do słownika, na wyjście generujemy R
- A jest w słowniku; jesteśmy na końcu wiadomości więc na wyjście generujemy A

wyjście	Słownik	
	Kod	Ciąg znaków
66	256	BA
65	257	AB
256	258	BAA
257	259	ABR
82	260	RR
260	261	RRA
65		

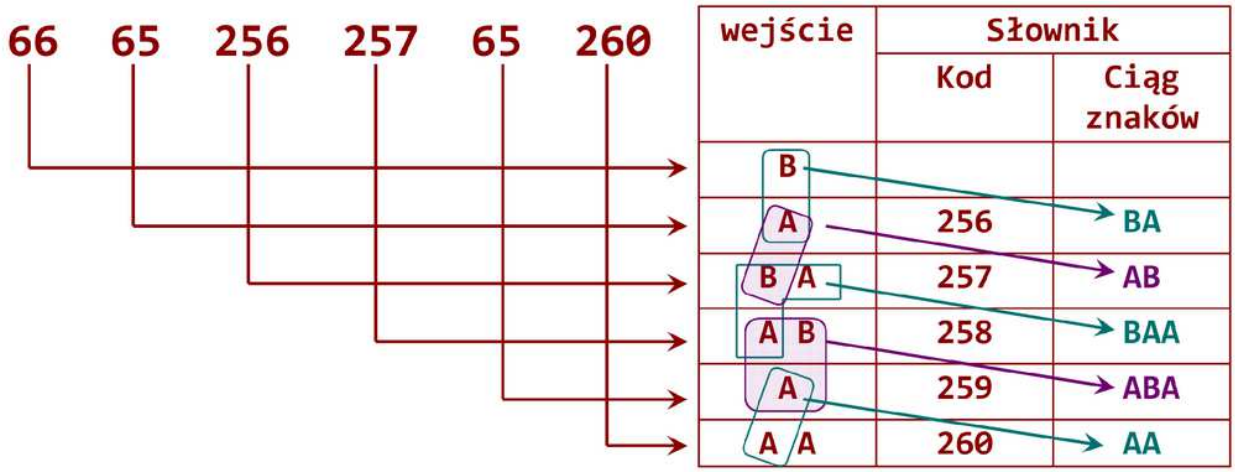


- B jest w słowniku. BA nie znajduje się w słowniku; dodajemy BA do słownika, a na wyjście generujemy B
- AB nie znajduje się w słowniku; dodajemy AB do słownika, a na wyjście generujemy A
- BA jest w słowniku. BAA nie znajduje się w słowniku; dodajemy BAA do słownika, a na wyjście generujemy BA
- AB jest w słowniku. ABA nie znajduje się w słowniku; dodajemy ABA do słownika, na wyjście generujemy AB
- AA nie znajduje się w słowniku; dodajemy AA do słownika, na wyjście generujemy A
- AA jest w słowniku; jesteśmy na końcu wiadomości więc na wyjście generujemy AA

wyjście	Słownik	
	Kod	Ciąg znaków
66	256	BA
65	257	AB
256	258	BAA
257	259	ABA
65	260	AA
260		

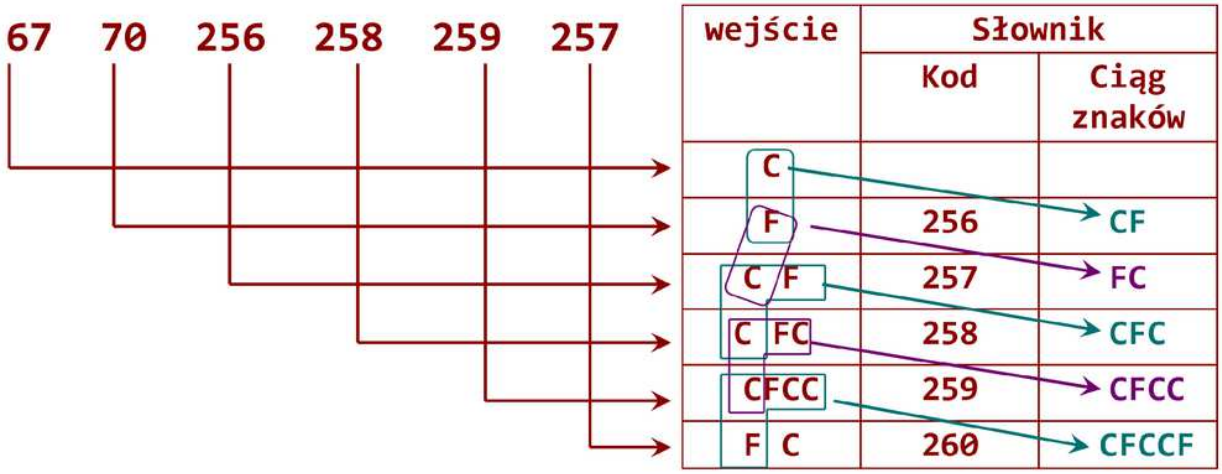
Inny przykład działania dekompresji

- **Dekompresujemy ciąg [66, 65, 256, 257, 65, 260]**



- 66 znajduje się w słowniku, generujemy na wyjściu B
- 65 znajduje się w słowniku, generujemy na wyjście A, dodajemy do słownika BA
- 256 znajduje się w słowniku, generujemy na wyjście BA, dodajemy do słownika AB
- 257 znajduje się w słowniku, generujemy na wyjściu AB, dodajemy do słownika BAA
- 65 znajduje się w słowniku, generujemy na wyjście A, dodajemy do słownika ABA
- 260 NIE ZNAJDUJE SIĘ w słowniku, generujemy na wyjściu poprzednie wyjście + pierwszy znak poprzedniego wyjścia tj. AA, dodajemy do słownika AA

- **Dekompresujemy ciąg [67, 70, 256, 258, 259, 257]**



- 67 znajduje się w słowniku, generujemy na wyjściu C
- 70 znajduje się w słowniku, generujemy na wyjście F, dodajemy do słownika CF
- 256 znajduje się w słowniku, generujemy na wyjście CF, dodajemy do słownika FC
- 258 NIE ZNAJDUJE SIĘ w słowniku, generujemy na wyjściu poprzednie wyjście + pierwszy znak poprzedniego wyjścia tj. CFC, dodajemy do słownika CFC
- 259 NIE ZNAJDUJE SIĘ w słowniku, generujemy na wyjściu poprzednie wyjście + pierwszy znak poprzedniego wyjścia tj. CFCC, dodajemy do słownika CFCC
- 257 znajduje się w słowniku, generujemy na wyjście FC, dodajemy do słownika CFCCF