Format PNG - prezentacja na KD

Mateusz Dudziński

2020-01-18 Thu

Outline

Informacje ogólne

Bezstratny format kompresji obrazu opracowany pod kątem przesyłania grafiki w internecie.

Następca formamtu GIF.

Wspiera wiele różnych formatów pikseli - RGB, RGBA, Grayscale, Grayscale + Alpha, 48-bit color.

Stosunkowo prosty (jak na format kompresji obrazu).

Nie jest i nigdy nie był opatentowany.

Krótka historia

- Opracowany w 1995
- X 1996: Wersja 1.0 - tego czasu format ten jest rekomendowany przez W3C.
- I 1997: Opublikowany (bez zmian) w dokumencie RFC 2083.
- XII 1998: Wersja 1.1 - niewielkie zmiany i trzy nowe bloki (iCCP, sPLT, oraz sRGB).
- VIII 1999: Wersja 1.2 - dodatkowo jeszcze jeden blok (iTXt).
- XI 2003: PNG zostaje standardem ISO (ISO/IEC 15948:2004) - standard praktycznie nie różni się od wersji 1.2.

Mało (istotnych) zmian na przestrzeni lat - to dobry znak.

Główne założenia

Prostota i przenośność

Jedyny algorytm kompresji zastosowany w tym formacie to popularny do dzisiaj DEFLATE, będący też głównym algorytmem kompresowania danych w programie gzip (zip), który posiada bardzo wiele wspierających bibliotek. Przez co dekodowanie obrazów PNG nie jest trudne.

. . . nawet kosztem kompresji danych

DEFLATE jest uniwersalnym algorytmem kompresji, radzącym sobie, w założeniu, "z każdego rodzaju danymi". Czy to znaczy, że PNG nie w prawie żaden sposób zoptymalizowane pod kątem kompresji obrazów?

Główne założenia

Niewiele lepszy niż zwykły obraz skompresowany zip'em

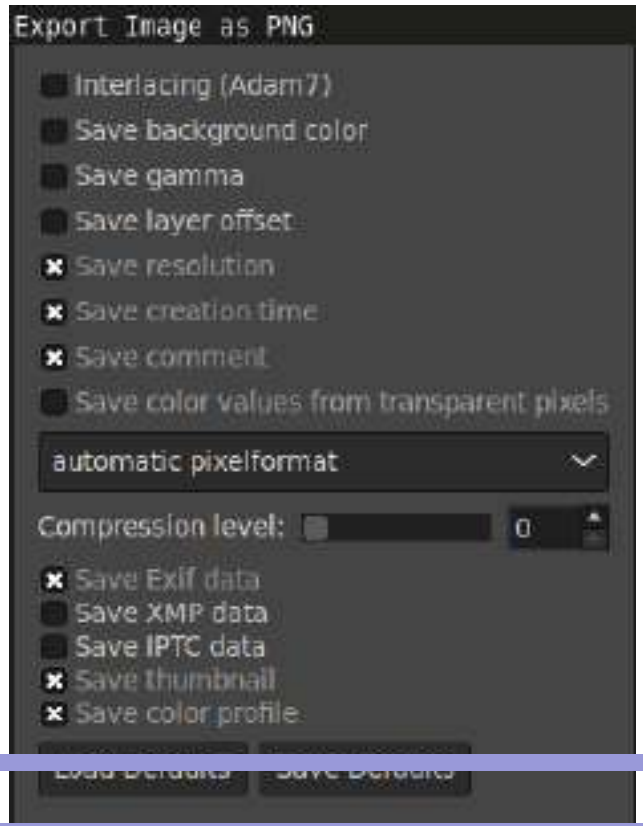
Przy "standardowych" obrazkach, zwykle PNG daje nieznacznie lepszy wynik niż po prostu stworzenie archiwum .zip z tym jedym obrazem jako bitmapą. Odpowiedzialne jest za to użycie w PNG transformat na obrazie zanim wykoda się na nim DEFLATE'a.

Ale ma też inne zalety

- Wiele różnych formatów kolorów (w tym indeksowany z palety)
- Możliwość dodania wielu matadatych (tekst, czas modyfikacji obrazu, fizyczne wymiary)
- Możliwość dodawania własnych metadanych prez "rozszerzanie" formatu, które jest tak zaprojektowane, żeby standardowe dekodery ciągle mogły rozczytać obraz.

Stopień kompresji

Skuteczność kompresji zależy przede wszystkim od tego, ile czasu damy koderowi na kompresowanie pliku. Większość programów pozwala użytkownikowi wybrać poziom kompresji pliku - im wyższy tym więcej czasu zajmie kompresowanie (poziomy 0-9).



Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1080
- Różnica: 8.31Mb / 2.98Mb (2.79)

Skuteczność dla różnych obrazów.

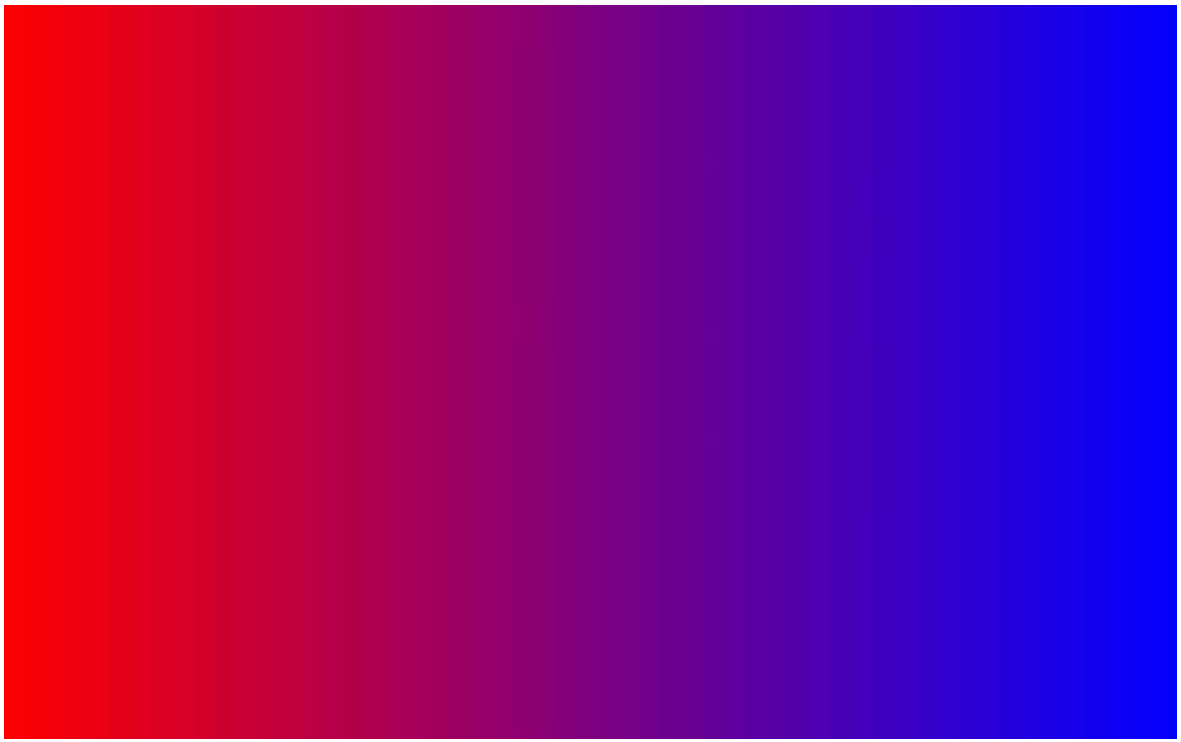
- Rozmiar: 1920 x 1080
- Różnica: 6.94Mb / 2.66Mb (2.61)

Skuteczność dla różnych obrazów.

- Rozmiar: 2000 x 1240
- Różnica: 7.47Mb / 2.43Mb (3.07)

Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1200
- Różnica: 6.92Mb / 1.09Mb (6.35)



- Bardzo długi czas kompresowania obrazu.

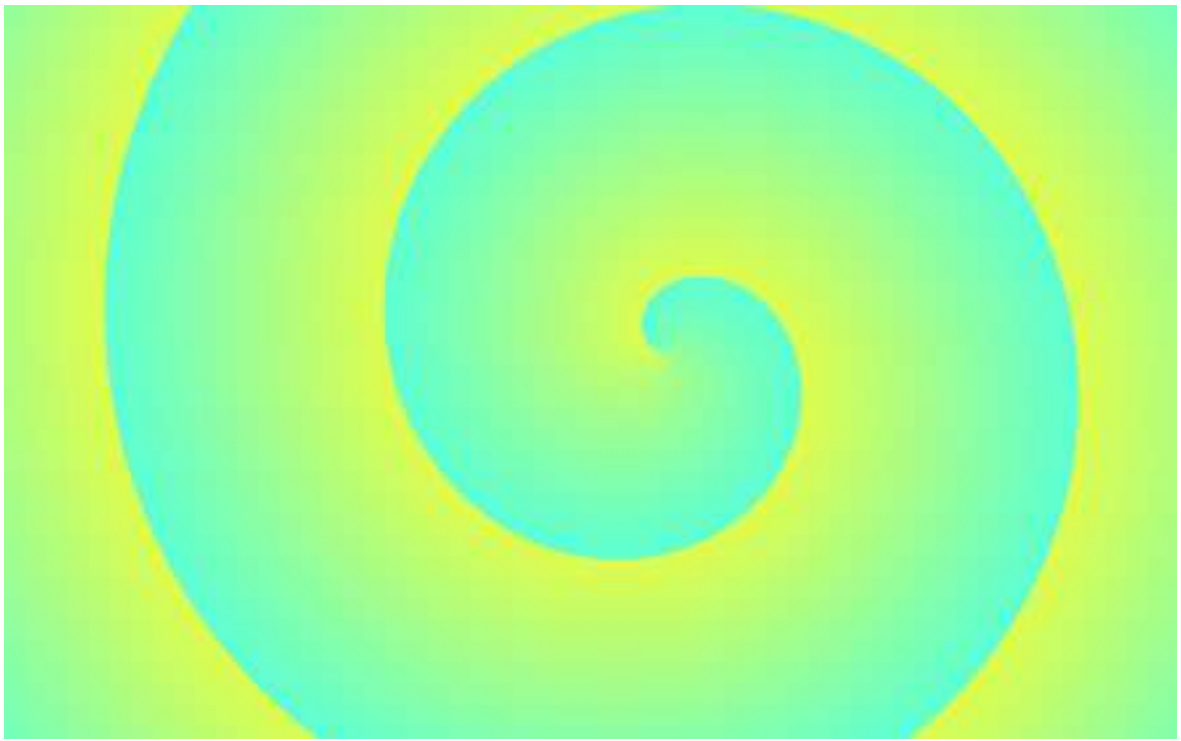
Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1200
- Różnica: 6.92Mb / 1.34Mb (5.16)

- Również bardzo długi czas kompresowania obrazu.

Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1200
- Różnica: 6.92Mb / 1.38Mb (5.01)



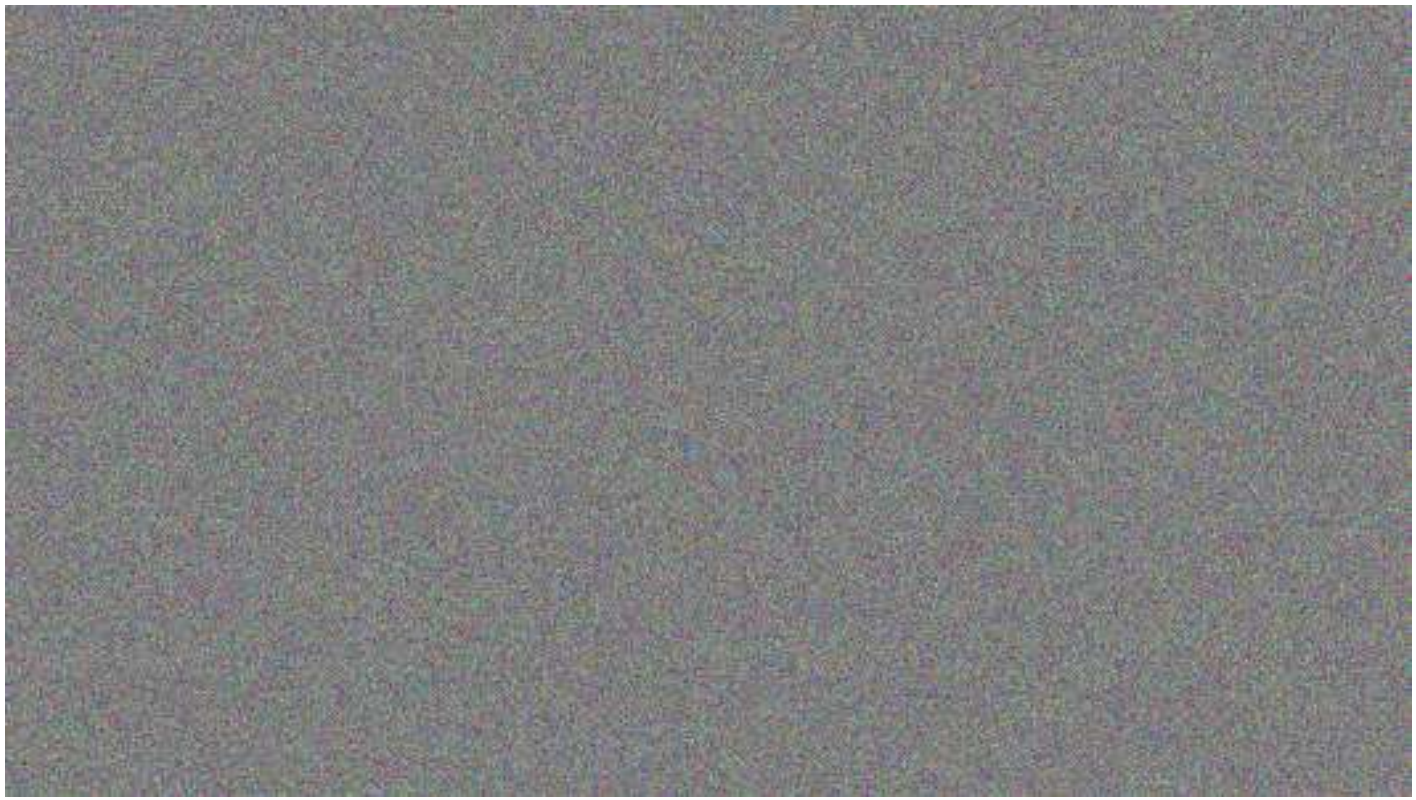
- Również bardzo długi czas kompresowania obrazu.

Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1080
- Różnica: 6.23Mb / 6.6Kb (943.94)

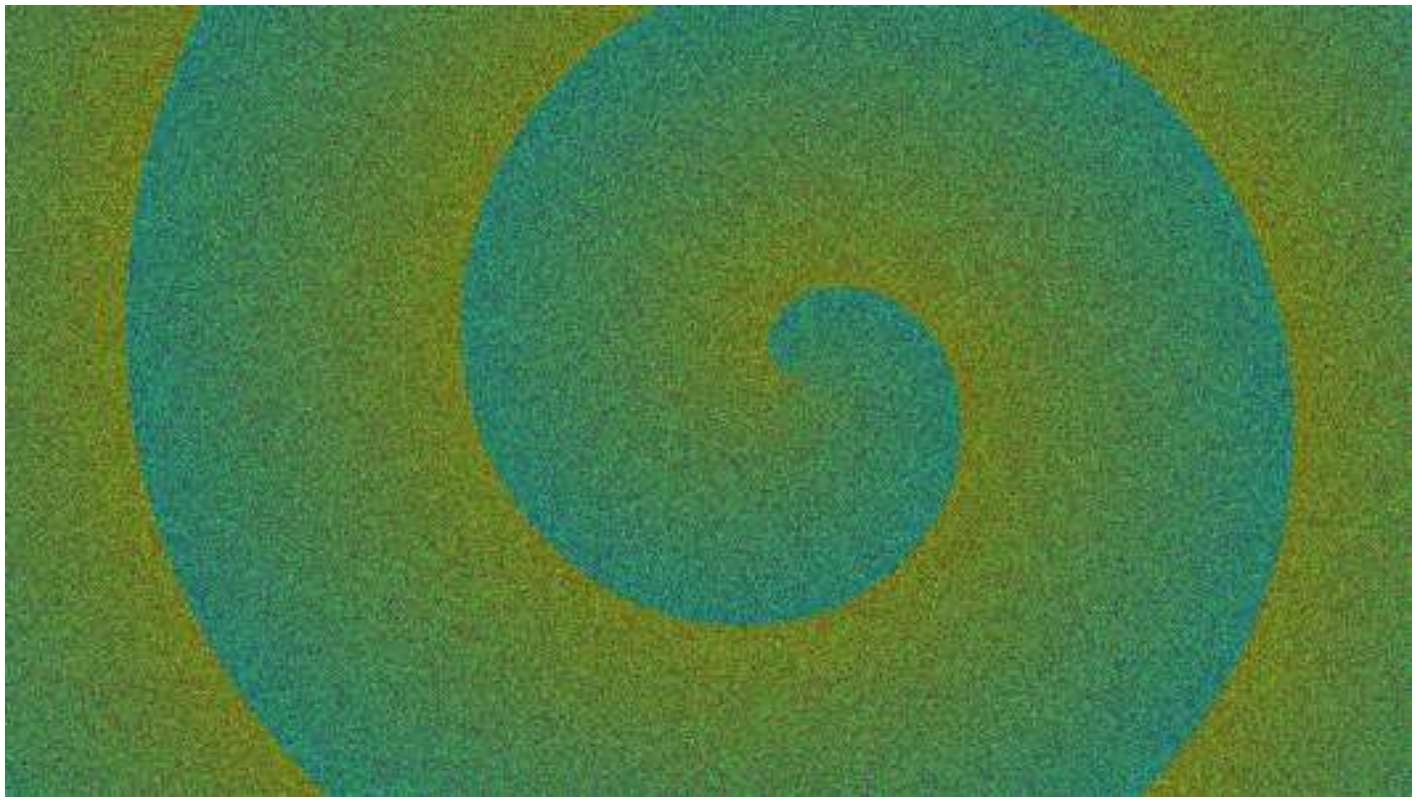
Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1080
- Różnica: 8.31Mb / 7.14Mb (1.16)



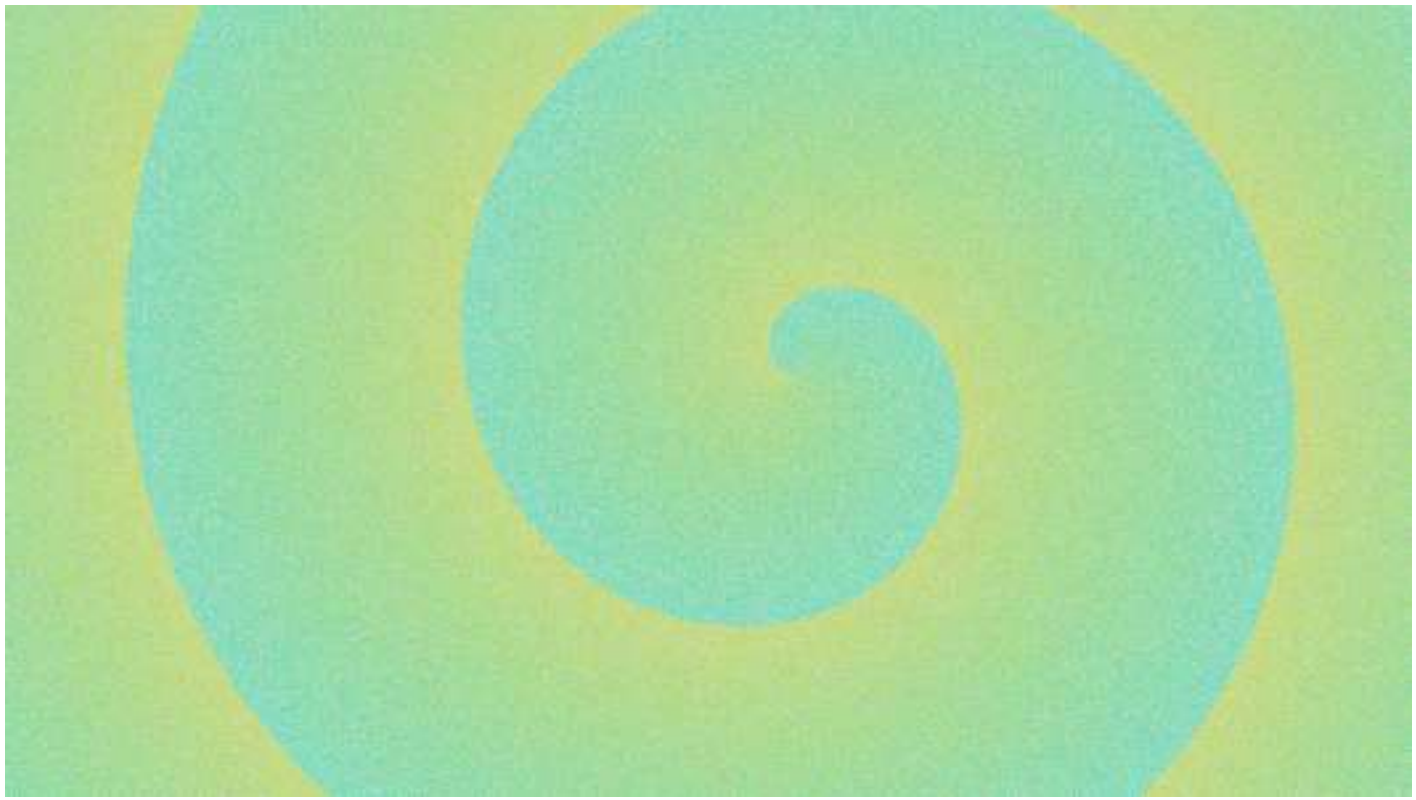
Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1080
- Różnica: 8.31Mb / 7.01Mb (1.18)



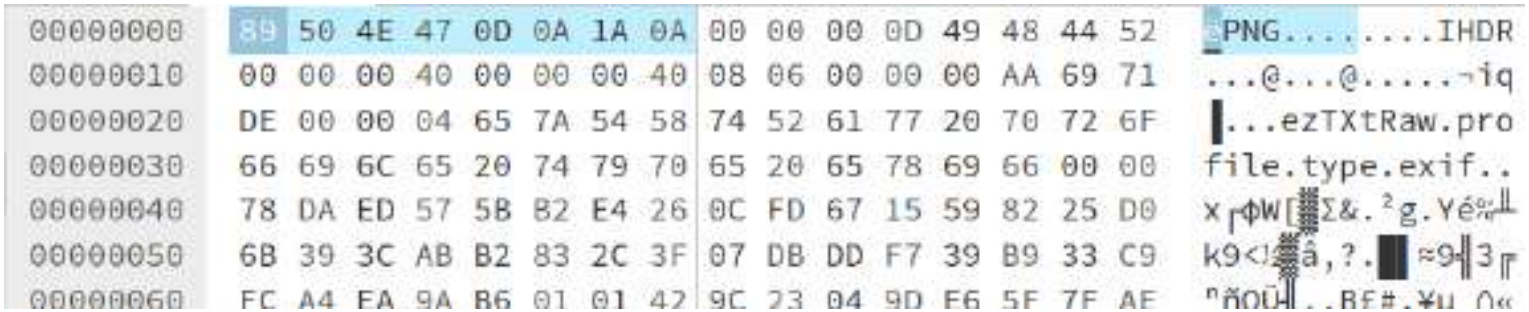
Skuteczność dla różnych obrazów.

- Rozmiar: 1920 x 1080
- Różnica: 8.31Mb / 6.56Mb (1.27)



Stuktura pliku - nagłówek

Pierwsze 8 bajtów są zawsze takie same i służą do rozpoznania czy format pliku to PNG.



Stuktura pliku - bloki

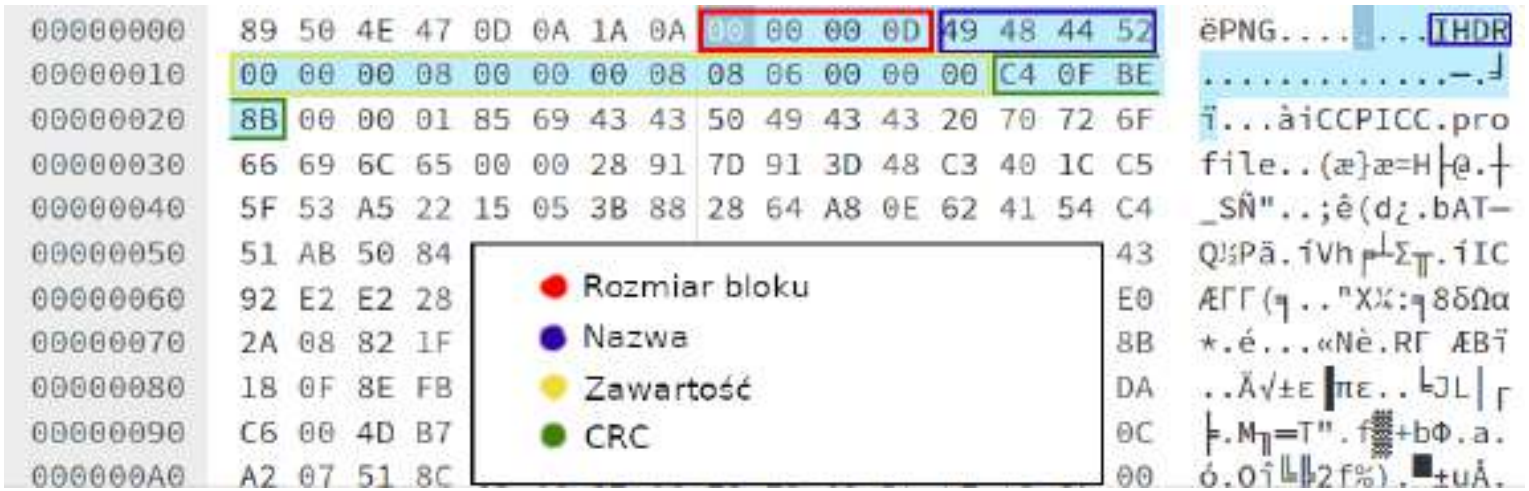
Reszta dokumentu składa się z bloków

[długość]	[rodzaj bloku]	[zawartość]	[suma CRC]
4 bajty	4 bajty	[długość] bajtów	4 bajty

- Zawartość może mieć długość 0.
- Suma CRC jest liczona z rodzaju bloku i zawartości, bez długości.

```
tEXt
||||- "Safe to copy"
|||-- (Standard wymaga wielkiej litery)
||-- "Public (Standard conforming)"
|--- "Not critical"
```

Stuktura pliku - bloki - przykład



Stuktura pliku - bloki - IHDR

Musi wystąpić jako pierwszy blok, zaraz po nagłówku. Zawiera ogólne informacje o obrazie.

- Szerokość: 4B
- Wysokość: 4B
- Bity na pixel (depth): 1B
- Colortype: 1B (Grayscale, Grayscale + A, RGB, RGBA, Indeksowany)
- Metoda kompresji: 1B (Standard: tylko 0)
- Metoda filtrowania: 1B (Standard: tylko 0)
- Typ przeplotu (Interlace): 1B (Standard: tylko 0 (brak) i 1 (Adam7))

Stuktura pliku - bloki - PLTE

Dla obrazów typu indexed color, wartości 'pixeli' to indeksy do tablicy zawierającej dokładne informacje o kolorze.

- Występuje przed zawartością obrazu po IHDR.
- MUSI wystąpić jeżeli poprzedni blok definiował *Colortype* jako *Indeksowany*.
- Może też wystąpić dla RGB i RGBA.
- Zawiera informacje o odwzorowaniu zawartości obrazu na rzeczywiste kolory (służy jako paleta kolorów).
- Rzadko spotykana (blok sPLT).
- Nie trzeba wysyłać całej: zwykle zmieniane są tylko małe wartości. W pliku wysyła się tablicę wartości od 0 do 3N gdzie 3N jest zdeterminowane przed długość bloku.

Stuktura pliku - bloki - IEND

- Musi wystąpić jako ostatni blok.
- Oznacza koniec pliku PNG.

Stuktura pliku - bloki - IDAT

- Najważniejszy blok. Zawiera (potencjalnie skompresowaną) zawartość obrazu.
- Szczegóły za chwilę.

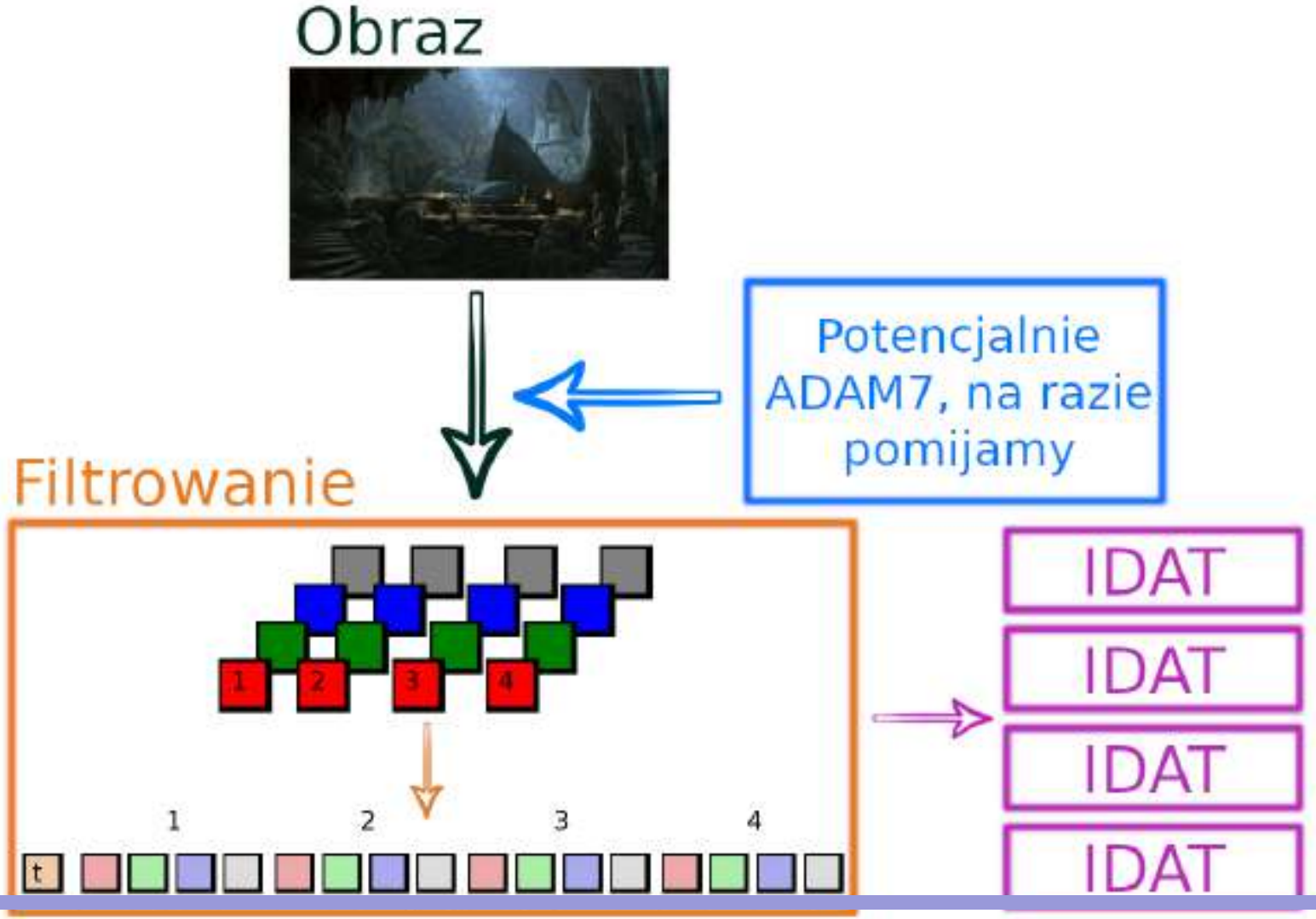
Stuktura pliku - pozostałe (niekrytyczne) bloki

- bKGD - Kolor tła.
- cHRM - Definicja barw względem przestrzeni CIEXYZ (CIE1931).
- gAMA - Gamma obrazu.
- hIST - Opisuje częstotliwość użycia poszczególnych kolorów z palety. Przydatny kiedy dekodujący nie ma aż tyle kolorów ile jest opisane w palecie.
- tEXt - dane tekstowe w postaci "klucz - wartość". Klucz do 70 bajtów, wartość dowolna. Kilka predefiniowanych kluczy *"Author"*, *"Comment"*, itd.
- zTXt - jak wyżej, ale wartość jest poddawana kompresji, dokładnie tym samym algorytmem co zawartość obrazu.

Stuktura pliku - pozostałe (niekrytyczne) bloki

- pHYs - Fizyczne wymiary obrazu. Ilość pikseli na jednostkę w osi X i Y.
- tIME - Ostatni czas zmiany zawartości obrazu.
- tRNS - Opis kanału alfa dla całego obrazu - nie dotyczy wersji *RGBA* i *Grayscale + A*.

Kompresja w PNG - schmat



Filtry (przekształcenie)

"Filtry" to nazwa stosowana w specyfikacji. Ale wszystkie filtry są bezstratne (dlatego zamiennie będę używał określenia "przekształcenie")

Metoda przekszałcania zdefiniowana na początku pliku w IHDR.

Póki jest tylko standard definiuje tylko jedną metodę (0).

Modyfikują poszczególne składowe *pixeli*, co czyni je nieco bardziej skomplikowanymi w implementacji, ale znacznie bardziej skutecznymi. Wartość w kolorze czerwonym oznacza różnice kanałów czerwonych itd.

Filtry (przekształcenie)

Każdy wiersz obrazu (*scanline*) posiada osobne przekształcenie.

Jak wybrać optymalny filtr? Tak, żeby algorytm DEFLATE działał najskuteczniej!

- Chcemy jak najwięcej powtarzających się fragmentów (bo LZ77 pozwala "kopiować" takie fragmenty)
- Chcemy używać jak najwięcej tych samych znaków (bo DEFLATE używa kodów Huffmana).
- O DEFLATE za chwilę.
- Prosta, ale bardzo skuteczna i stosowana w praktyce heurystyka: liczymy dla każdego filtra sumę wszystkich wartości w wierszu i wybieramy ten, który daje najmniejszą sumę.

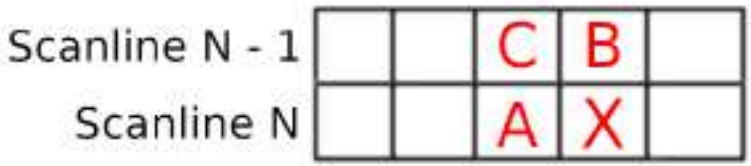
Filtry - Metoda 0 (jedyna zdefiniowana przez standard)

5 różnych typów. Każdy wiersz (*scanline*) może mieć inny.

Typ przekształcenia zajmuje jeden bajt (od 0 do 4) i jest doklejony na początku każdego wiersza.

Zwiększa to dane które kompresujemy, ale nie drastycznie (około 1 promil dla "standardowych" obrazków 1200x1080).

Filtry - Metoda 0 (jedyna zdefiniowana przez standard)



- pixel $= x$
- pixel $= x - a$
- pixel $= x - b$
- pixel $= x - \lfloor \frac{a+b}{2} \rfloor$
- pixel $= x - \mathtt{Paeth}(a, b, c)$

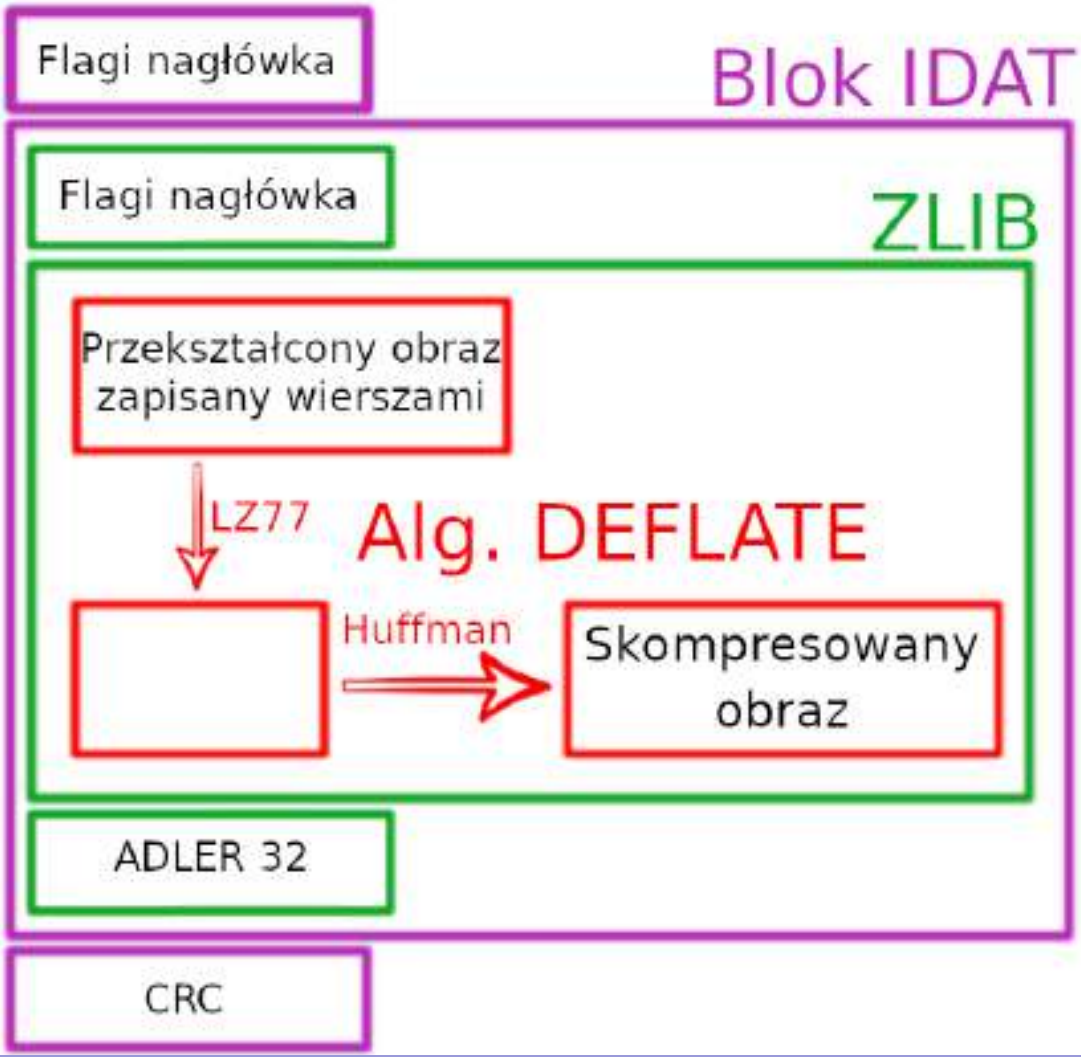
Funckja `Peath` próbuje ustalić w którym z trzech kierunków gradient obrazu jest najmniejszy.

```
function Peath(a, b, c) =
    let p = a + b - c;
    let pa = abs(p - a);
    let pb = abs(p - b);
    let pc = abs(p - c);
    if pa <= pb and pa <= pc
       Pr = a;
    else if pb <= pc
       Pr = b;
    else
       Pr = c;
    return Pr;
```

Kompresja w PNG - schemat

Podobnie jak w filtrach, tylko metoda 0 jest zdefiniowana przez standard - oznacza użycie algorytmu DELFATE i *zlib* jako formatu bloku.

Kompresja w PNG - schemat



ZLIB

Abstrachuje dane generowane przez DEFLATE, dodając nagłówki i sumy kontrolne tworząc format danych o nazwe tej samej co biblioteka.

Bardzo skuteczna biblioteka implmentująca ten format danych, używana powszechnie w przemyśle.

m.in: Linux, GNU Binutils, GDB, libpng, libwww, wszystkie przeglądarki, cURL, Apache HTTPD, OpenSSH, OpenSSL, FFmpeg, rsync, dpkg, RPM, SVN, CVS, git, PostgreSQL, .NET Framework >= 2.0, iPhone, Sony PlayStation, ...

ZLIB - Format danych

Cały blok:

[1 B]	[1 B]	[0/4 B]	[?]	[4 B]
CMF	FLG	DICTID	Zawartość	Suma Adler32

CMF:

[0-3]	[4-7]
CM	CINFO

FLG:

[0-4]	[5]	[6-7]
FCHECK	FDICT	FLEVEL

- CM - (Compression method), zdefiniowany tylko 8 - DEFLATE.
- CINFO - log_2 z rozmiaru okna LZ77, wymanane ≤ 7.
- FCHECK - taka liczba, że: $31 | (\texttt{CMF} * 256 + \texttt{FLG})$
- FDICT - Czy użyty został słownik? (Czy DICTID obecny).
- FLEVEL - Informacja, jak bardzo encoder próbował skompresować plik.

DEFLATE

Dwie metody kompresji, stosowane jedna po drugiej:

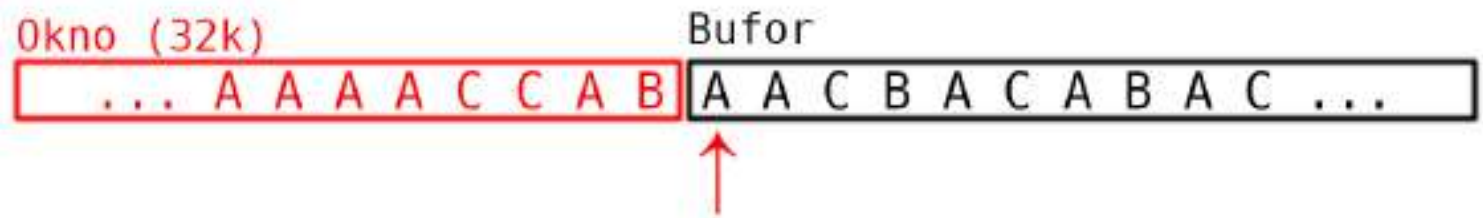
- LZ77 (Abrahama Lempel, Jacob Ziv, 1977)
- Kody Huffmana (David A. Huffman, 1952)
- Strumień może być dowolnie podzielony na bloki. Każdy blok może mieć oddzielnie zdefiniowane drzewo kodów prefiksowych.
- Istnieje możliwość nie kompresowania bloku (koder może uznać, że jakaś część pliku jest niekompresowalna, wydzielić na nią osobny blok i zostawić w niezmienionej formie).

Od tej pory pobieramy ze strumienia już praktycznie tylko pojedyńcze bity.

LZ77

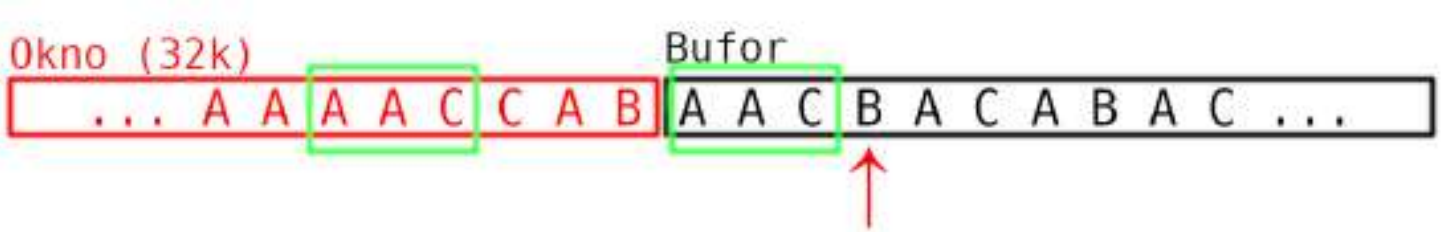
- Metoda szukania powtórzonych już ciągów w strumieniu danych. W alfabecie oprócz literałów (naszym przypadku literały to wartości pixeli $\in [0; 255]$), dodajemy jeszcze parę $< A; B >$, oznaczającą powtórzenie B znaków zaczynając od pozycji A pozycji przed kursorem.
- Okno kompresji na początku jest puste. Enkoder pisze do niego bajty, dokładnie takie, jakie zobaczy dekoder. Referencja do poprzedniego ciągu może wystąpić tylko w obrębie tego okna (zwykle 32k), DEFLATE nie pozwala na więcej.
- Zwykle na początku kompresji okno jest puste. `DICTID` w nagłówku `zlib` pozwala się umówić koderowi i dekoderowi na początkowy stan tego okna.
- Potencjalnie gorsze od RLE dla bardzo specyficznych danych ($\mathcal{O}(log2n)$ vs $\mathcal{O}(1)$)? Jak się zaraz okaże, niekoniecznie!

LZ77 - przykład



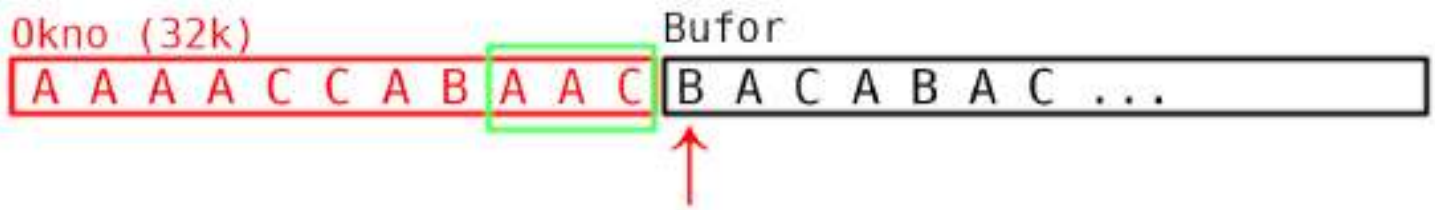
Wynik: . . . A A A A C C A B

LZ77 - przykład



Wynik: ... A A A A C C A B

LZ77 - przykład



Wynik: `...  A A A A C C A B <6;3>`

LZ77 - porównanie z RLE

Najprostsze RLE:

4M2a3t2e1u1s2z $\rightarrow$ MMMMaattteeuszz

5184000\0 $\rightarrow$ 1200 $\times$ 1080 pikseli o wartości 0 (czarny obraz)

Wydaje się, że PNG jest istotnie gorsze w tym przypadku, bo potrzebuje 6.6KB na taki obraz.

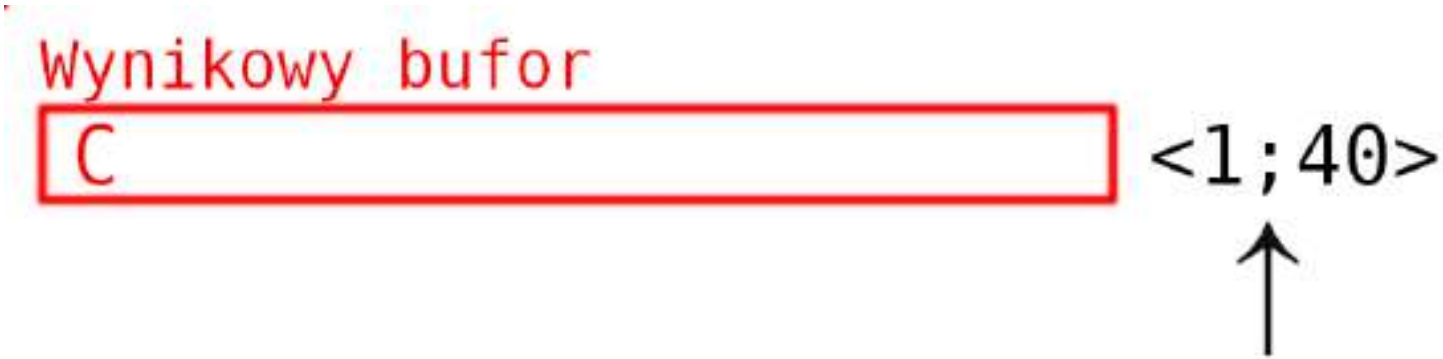
Możemy wyprodukować wykładniczo rosnące ciągi przy użyciu LZ:

`0 <1; 1> <2; 2> <4; 4> <8; 8> ...`

Ale zlib pozwala nam zakodować to lepiej!

LZ77 - porównanie z RLE

Ale zlib pozwala nam zakodować to lepiej!



LZ77 - porównanie z RLE

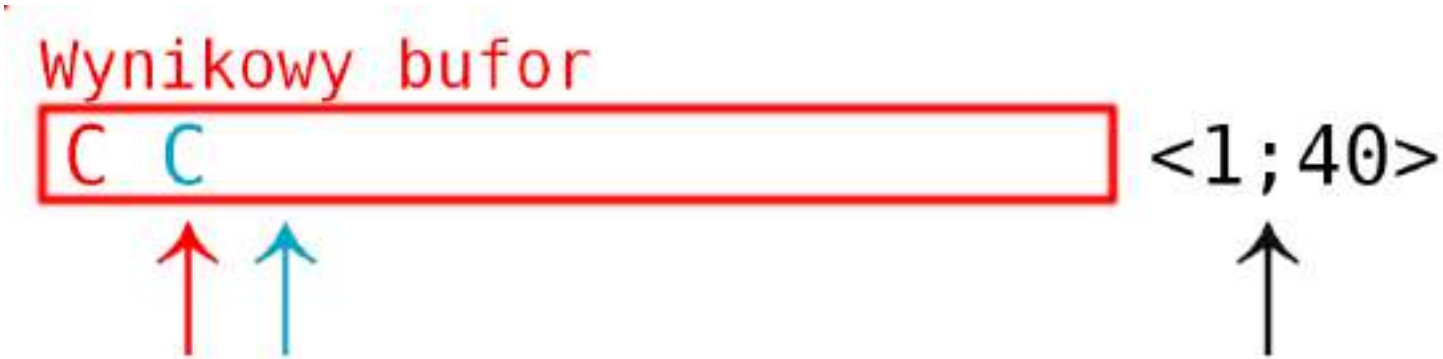
Ale zlib pozwala nam zakodować to lepiej!

LZ77 - porównanie z RLE

Ale zlib pozwala nam zakodować to lepiej!

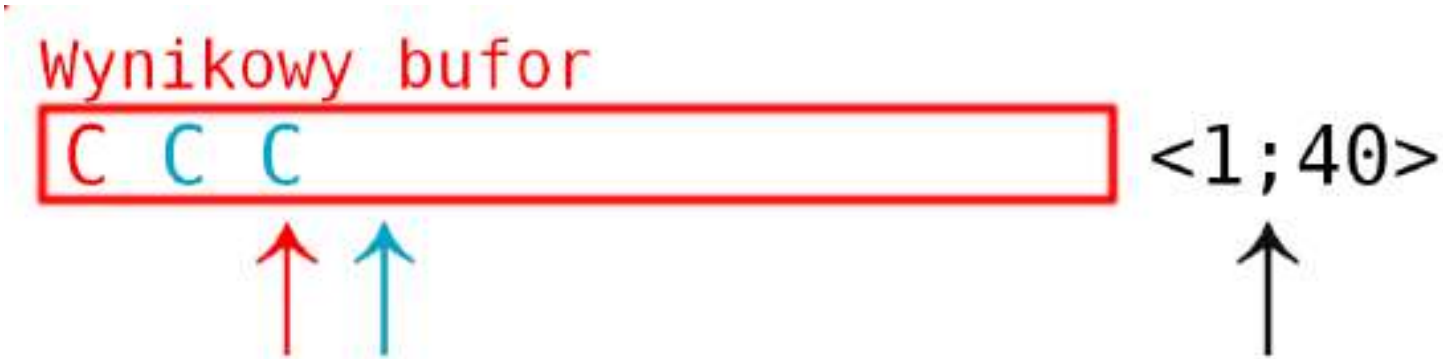
LZ77 - porównanie z RLE

Ale zlib pozwala nam zakodować to lepiej!



LZ77 - porównanie z RLE

Ale zlib pozwala nam zakodować to lepiej!



LZ77 - porównanie z RLE

Czyli możemy zakodować w $\mathcal{O}(1)$ powtórzenie dowolnie wiele razy jednego znaku.

No to dlaczego potrzebujemy 6.6KB na taki prosty obraz?

- Bo deflate przeznacza bardzo niewiele bitów na długość podciągu który kopiujemy - możemy maksymalnie skopiować 258 znaków.
- A to dlatego, że na liczbę mówiącą ile należy się cofnąć, jest przeznaczona stała liczba bitów!

LZ77 - jak zapisywać literały oznaczające skok?

Jaki wogóle mamy alfabet?

- Znak z $[0; 255]$ - bajt o takiej wartości, 256 - koniec.
- Znak z $[257; 285]$ - skopiowanie ciągu który już się pojawił.
- Wartości $\in [257; 285]$ mówią nam, że będziey kopiować i definiują ile bajtów skopiujemy (zaraz zobaczymy jak). Jak będziemy wiedzieć skąd kopiować? Użyjemy do tego drugiego alfabetu.
- Drugi alfabet będzie nam mówił ile musimy się cofnąć.
- W obu przypadkach bo znaku następują dodatkowe bity podające informacje. Np. '277' oznacza kopiowanie od 67 do 82 bitów. A następne 4 bity mówią nam dokałdnie ile musimy dodać do 67.

Jak DEFLATE koduje skok LZ?

Pierwszy element z pary wyznacza ilość kopiowanych bajtów.
Po nim mogą nastąpić dodatkowe bity.

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

Jak DEFLATE koduje skok LZ?

Na indeks z którego kopiujemy mamy 5 bitów + ew. dodatkowe bity.

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

Jak DEFLATE koduje skok LZ?

Przykład:

1 0 0 0 0 1 1 1 0 1 0 1 0 1 0 0 1 0 1

- Ile kopiujemy? $100001110_2 = 270_{10}$
  $T_1(270) = (2, 23)$ (czytamy 2 bity i dodajemy do 23)
  $10_2 = 2_{10}$
  Czyli kopiujemy $23 + 2 = 25$.
- Skąd kopiujemy? $1010_2 = 10_{10}$
  $T_2(10) = (4, 33)$ (czytamy 4 bity i dodajemy do 33)
  $0101_2 = 5_{10}$
  Czyli kopiujemy z pozycji $33 + 5 = 38$ bajtów wstecz.

Zatem zakodowany skok to $< 25; 38 >$.

Podsumowanie tej części:

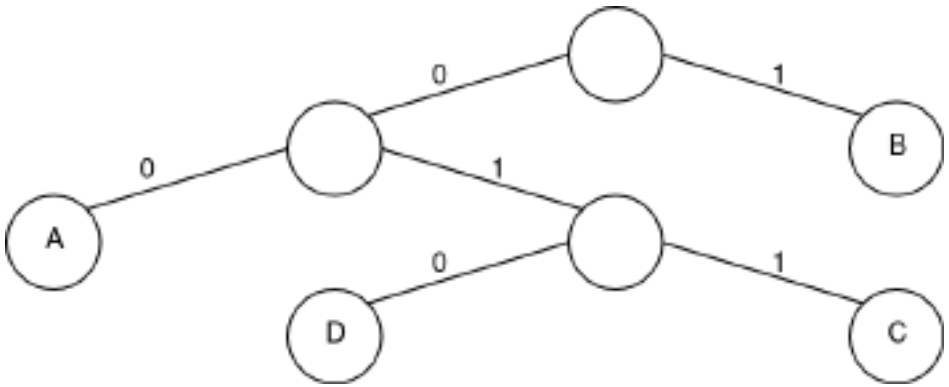
- Nasz ciąg skompresowanych danych składa się teraz z wartości bajtowych $B = [0; 255]$ oraz z (koncepcyjnie) par $< A; \Delta >$ oznaczających kopoiowanie A bajtów z pozycji B bajtów za obecny indeksem.
- Wartości bajtowe z B i wartości z A łączymy w jeden alfabet $\Sigma = [0; 258]$. Na odległości mamy drugi alfabet: $\Delta = [0; 31]$. Po niektórych wartościach związanymi z kopiowaniem LZ mogą nastąpić dodatkowe bity, ale będziemy wiedzieć ile ich jest (literał nam to wyznacza).

Kodowanie Huffmana

- Każde słowo z alfabetu Σ zastępujemy ciągiem bitowym odwrotnie proporcjonalnym do pradopodobieństwa jego wystąpienia.
- Inaczej, chcemy żeby najczęsciej występujące elementy były reprezentowane przez jak najkrótsze ciągi bitów.
- Minimalizujemy: $\sum_{i \in \Sigma}(p_i * |H(i)|)$, gdzie $H(i)$ to ciąg bitów jaki przypisujemy słowu i.

Kodowanie Huffmana

- Kody wyrazów z naszego alfabetu reprezentujemy przez drzewo binarne (drzewo Huffmana).



Takie drzewo daje nam wszystkie informacje niezbędne do dekodowania:

1 0 0 1 1 0 0 0 1 0 0 0 1 0 1 1 → B A B B A D A B C

\+ +++ + + +++ +++++ +++ + +++++

Kodowanie Huffmana

- Jak konstruować takie drzewo?
  - Oczywiście, zachłannie.
  - Używając kolejki priorytetowej można szukać najmniejszego poddrzewa w czasie $\Theta(nlogn)$. Drzewo o n liściach ma $2n-1$ węzły, więc cały algorytm można wykonać w czasie $\Theta(nlogn)$.

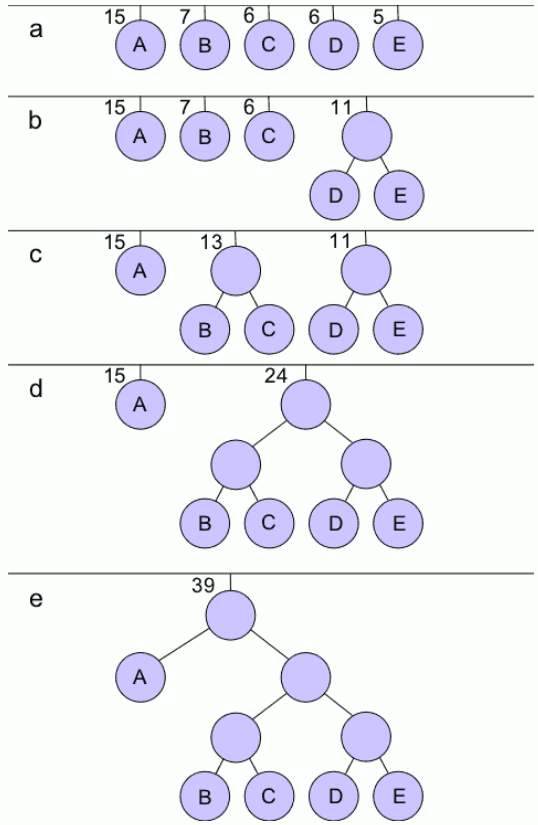
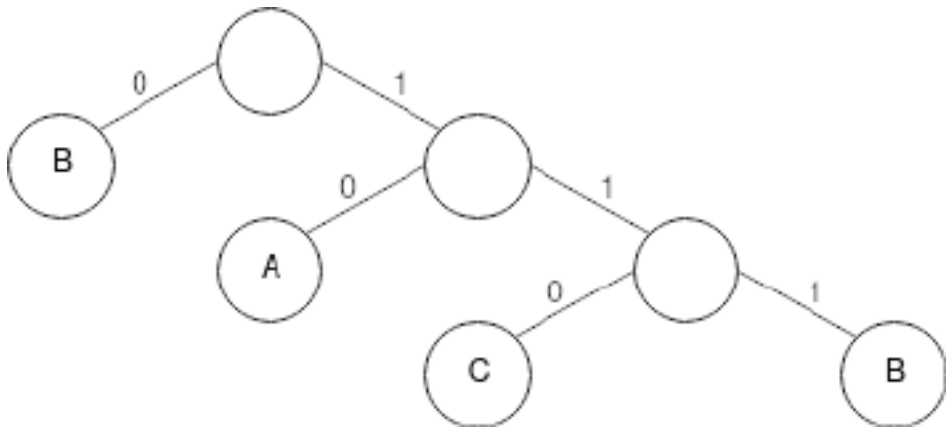


Figure: en.wikipedia.org

Kodowanie Huffmana

- Wysłanie całego drzewa (np. podając korzeń i wszystkie krawędzie), jest nieakceptowalne, bo zawiera za dużo pamięci.
- DEFLATE radzi sobie inaczej:
  - Dla każdej długości N, kody mają kolejne wartości (w sensie leksykograficznym).
  - Krótsze kody poprzedzają leksykograficznie dłuższe.
- Drzewo z poprzedniego przykładu wyglądałoby tak:



Dzięki temu dekoder może wygenerować drzewo Huffmana na podstawie tylko i wyłącznie informacji o długości kodów dla poszczególnych znaków.

- Algorytm wygląda tak:
  1. Niech blcount[N] oznacza liczbę kodów o długości N, N >= 1.

Dzięki temu dekoder może wygenerować drzewo Huffmana na podstawie tylko i wyłącznie informacji o długości kodów dla poszczególnych znaków.

- Algorytm wygląda tak:
  1. Niech blcount[N] oznacza liczbę kodów o długości N, N >= 1.
  2. Znajdź wartość najmniejszego leksykograficznie kodu dla każdej długości:

```
code = 0;
blcount[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + blcount[bits-1]) << 1;
    next[bits] = code;
}
```

Dzięki temu dekoder może wygenerować drzewo Huffmana na podstawie tylko i wyłącznie informacji o długości kodów dla poszczególnych znaków.

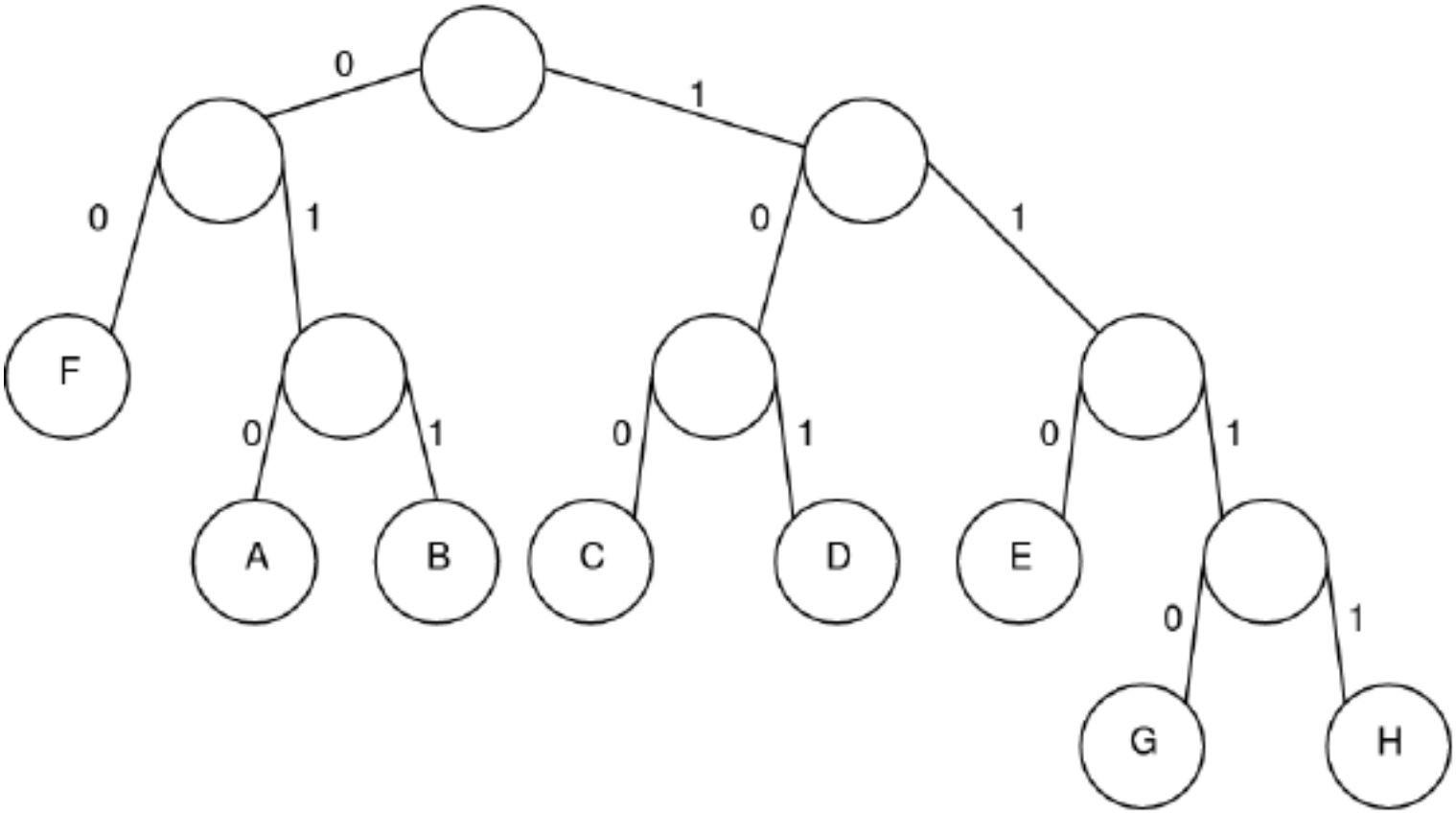
- Algorytm wygląda tak:
  1. Niech blcount[N] oznacza liczbę kodów o długości N, N >= 1.
  2. Znajdź wartość najmniejszego leksykograficznie kodu dla każdej długości.
  3. Kolejnym znakom przypisz leksykograficzne kolejne wartości ciągów:

```
for (n = 0; n <= MAX_CODE; n++) {
    len = input[n]; // długość ciągu dla znaku n;
    if (len <> 0) { huffman[n] = next[len]; next[len]++; }
}
```

Dzięki temu dekoder może wygenerować drzewo Huffmana na podstawie tylko i wyłącznie informacji o długości kodów dla poszczególnych znaków.

- Załóżmy że mamy alfabet $\Sigma = \{A, B, C, D, E, F, G, H\}$ i że drzewo jest kodowane przez ciąg (3, 3, 3, 3, 3, 2, 4, 4), tzn. A ma kod długości 3, B długości 3, . . . , F długości 2, . . .
  1. `blcount[N] = [0, 0, 1, 5, 2]`
  2. `next[N] = [0, 0, 0, 2, 14]`
  3. `output[i] = [2, 3, 4, 5, 6, 0, 14, 15] = [010, 011, 100, 101, 110, 00, 1110, 1111]`

Zdekodowane drzewo



Jeszcze większa kompresja drzwa Huffmana

- Ciąg takich długości, choć istotnie mniejszy niż wymienianie wszystkich krawędzi, ciągle jest bardzo duży.
- Szczególnie, że na jedną liczbę musielibyśmy przeznaczyć cały bajt (albo nawet dwa bajty).
- Ponieważ mamy dwa alfabety, jeśli jeden z nich będzie bardzo mały (potencjalnie zerowy), to nie chcemy marnować dużo bitów na długości kodów Huffmana.
- Zamiast podawać po kolei długości wszystkich kodów zdefiniujmy alfabet, który pozwoli nam wyrazić to samo, ale krócej.

Alfabet dla długości kodów jest następujący:

- 0-15: Kod długości $[0; 15]$.
- 16: Skopiuj poprzednią długość $(3 + k) \in [3; 6]$ razy. Dwa następne bity wyznaczją k.
- 17: Powtórz kod długości 0, $(3 + k) \in [3; 10]$ razy. 3 następne bity wyznaczją k.
- 18: Powtórz kod długości 0, $(11 + k) \in [11; 138]$ razy. 7 następnych bitów wyznacza k.

Przykład:

1 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 1 0 $\rightarrow$ 8 16 3 16 2

+++++++ +++++++++ +++ +++++++++ +++

$\rightarrow$ 8 8 8 8 8 8 8 8 8 8 8

Ten alfabet możemy skompresować kodami Huffmana!

- Na początku bloku wysyłamy długości kodów dla tego alfabetu $\Sigma_2 = \{0, 1, ..., 18\}$.
- Ale nawet to można ulepszyć. Jest szansa, że nie użyjemy wszystkich liter alfabetu. Więc najpierw wysyłamy 4 bitową informację, dla ilu znaków z Σ_2 podejemy długości kodów. Następnie podajemy te długości.
- W jakiej kolejności? Po kolei? $(0, 1, 2, 3, ...)$
- Nie: `16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15`
- Przez to, jeżeli elementy alfabetu, jakich potrzebujemy to $0, 5, 6, 7, 8, 16, 17, 18$. Możemy wysłać tylko pierwsze 8 liczb zamiast 16.

Definicja drzewa i tak zajumie dużo miejsca.

- Co jeśli mamy na tyle dużo danych, że chcemy je skompresować, ale na tyle mało, że definicja tablicy Huffmana zajmuje duży fragemnt kompresowanych danych?
- Istnieje możeliwość zakodowania za pomocą z góry ustalonego drzewa Huffmana: na początku bloku DEFLATE jest bit oznaczający że takie drzewo jest użyte. Wtedy definicja drzewa nie następuje. DEFLATE definiuje takie "domyślne" drzewo.
- Nie jest to kodowanie optymalne, ale możemy odzystać stracone miejsce oszczędzając na pamięci nie wysyłając drzewa Huffmana, które choć bardzo skompresowane, ciągle zajmuje niemało miejsca.

Ustalona tablica Huffmana:

Znak	Bity	Wartości
0 - 143	8	00110000-10111111
144 - 255	9	110010000-111111111
256 - 279	7	0000000-0010111
280 - 287	8	11000000-11000111

Wartości elementów alfabetu definiującego nawroty $[0; 31]$ są wszystkie reprezentowane przez 5-cio bitowe kody (w kolejności leksykograficznej).

Jeżeli koder uzna, że dane i tak są niekompresowalne, może zostawić dane bez kompresji.

Większość programów zapisujących obrazy w formacie PNG zrobi tak, dla poziomu kompresji 0. Takie obrazy mogą jednak ważyć *więcej* niż zwykła bitmapa, bo choć DEFLATE pozostawia dane bez kompresji, filtry zostały zastosowane (PNG nie pozwala na zapisanie obrazu bez filtrów), więc dane są rozdymane.

Przykład dekodowania (static Huffman):

00110001 00110000 00110001 00110002 00110004 00110002
00110003 00110002 00110004 00110003 00110002 00110004
0000101 0 00005 1 00110000 00110000
1 0 1 2 4 2 3 2 4 3 2 4 265 0 5 1 0 0
1 0 1 2 4 2 3 2 4 3 2 4 (11; 8) 0 0
1 0 1 2 4 2 3 2 4 3 2 4 4 2 3 2 4 3 2 4 4 2 3 0 0

Niech to będzie cały zdekodowany scanline RGBA. 1 oznacza rodzaj filtru. Wtedy żeby odzyskać dodajemy wartość z poprzedniego pixela.

0 1 2 4 2 3 2 4 3 2 4 4 2 3 2 4 3 2 4 4 2 3 0 0
0 1 2 4 2 4 4 8 5 6 8 12 5 6 8 12 8 8 12 16 10 11 12 16

Zalecany algorytm

- Kompresor kończy blok kiedy uznaje, że rozpoczęcie nowego bloku (z innym drzewem Huffmana) jest opłacalne, lub kiedy kończy mu się miejsca w buforze pamięci.
- Zalecane użycie tablicy haszującej z dowiązaniami do szukania powtórzonych ciągów używając funkcji haszującej operujących na 3-bajtowych sekwencjach.
- Załóżmy że dostajemy na wejściu "XYZ" (te znaki nie muszą być różne).
  - Sprawdzamy hasz słowa "XYZ".
  - Jeśli nie mamy takiego w tablicy, to kompresor pisze literał "X" a następnie dodaje następny bajt "W" do aktualnego ciągu otrzymując "YZW".
  - Jeśli wystąpienie istnieje, kompresor sprawdza wszystkie ciągi znajdujące się w komórce haszującej się tak jak "XYZ". Wymierając najdłuższy pasujący ciąg.

Zalecany algorytm

- Przeszukiwanie list odbywa się od tych które wystąpiły ostatnio do tych, które wystąpiły najdawniej (mały dystans skoku przewaga podczas kodowania Huffmana).
- Listy to zwykłe, jednostronne listy. Usuwanie wartości jest leniwe. Podczas przeszukiwania gdy kompresor natrafia na ciąg, który wystąpił dalej niż rozmiar okna, usuwa go wtedy.
- Bardzo długie listy są ograniczone przez jakąś ustaloną stałą długość (one rosną dość szybko).
- "Lazy match": załóżmy że mamy ciąg "XYZWABC". Może się okazać, że zamiast teraz kodować skok bardziej nam się opłaca zakodować literał "X" a następnie "YZWABC [następne bajty które pasują]". Kompresor może wyemitować skok, a następnie zrezygnować z niego kodując "X" a następnie lepszy skok.
- Im bardziej zależy użytkownikowi na kompresji, tym więcej czasu jest poświęcone szukaniu najlepszego ciągu.

Podsumowanie kompresji:

- Mając już gotowy ciąg literaów i par oznaczających kopię LZ liczymy wystąpienia i budujemy otpymalne drzewo Huffmana algorytmem zachłannym, albo decydujemy się użyć domyślnego drzewa Huffmana.
- Następnie zapisujemy długości ciągów dla kolejnych elementów alfabetu.
- Te długości kodujemy alfabetem Σ_2 (tym $[0; 18]$).
- Wysyłamy długości elementów Σ_2, ale tylko tyle ile potrzebujemy.
- Dekoder rozszyfrowuje Σ_2 a następnie drzwo Huffmana.
- Następnie czyta resztę zakodowanej informacji albo dodając bajty albo kopiując.

Adam7

Przeplot pozwala nam wysałać obraz tak, żeby dekodujący zobaczył go wcześniej, w mniejszej rozdzielczości.

PNG używa algorytmu ADAM7, który jest skuteczniejszy niż algorytm używany w GIF'ie

- Więcej iteracji w dwóch wymiarach, a nie tylko w jednym

Adam7

Obraz

1 2

3 3

Przechodzi na:

1 . 2 .

. . . .

3 . 3 .

. . . .

A następnie:

1 4 2 4

5 5 5 5

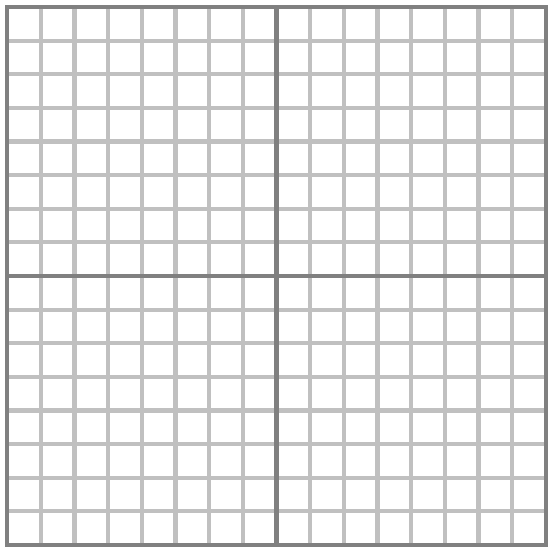
3 4 3 4

5 5 5 5

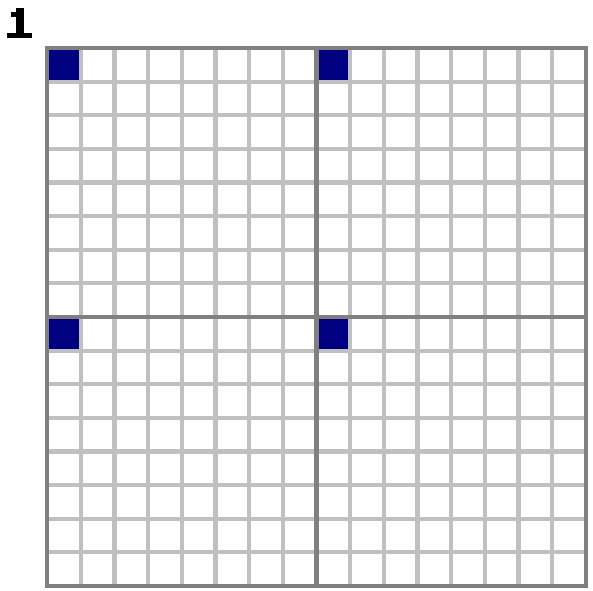
Adam7 lepiej niż w GIF'ie

- W GIF'ie `1 2` zostałby zamieniony `1 . 2 .` a następnie wypełniony `1 3 2 4`.
- Tylko wertykalnie w porównaniu do ADAM7.

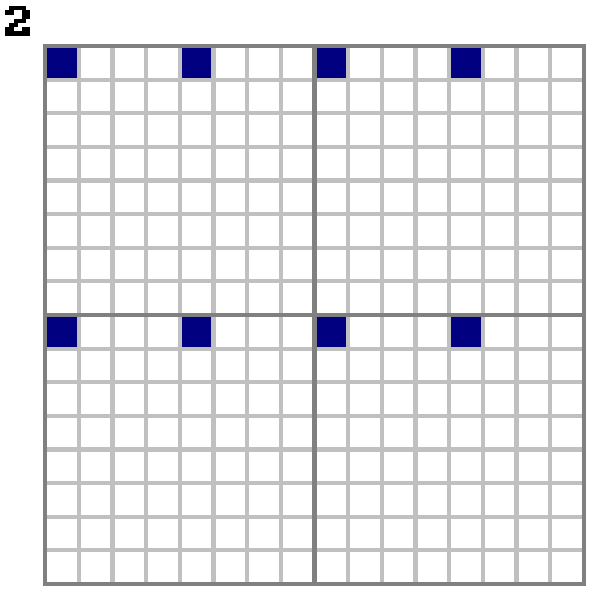
Adam7 (start)



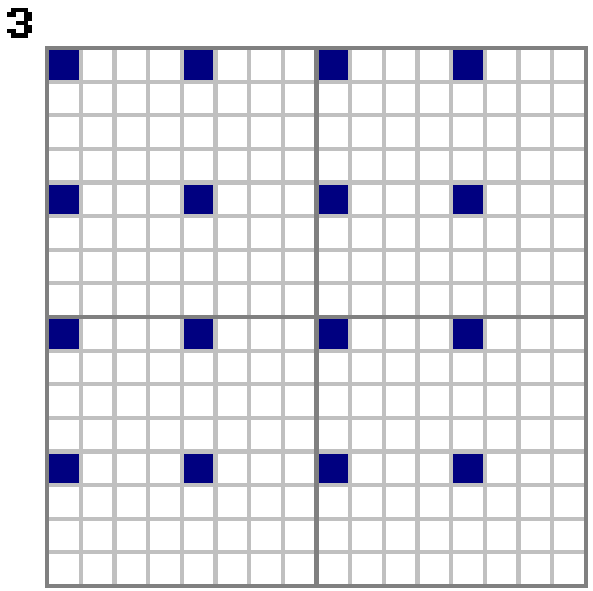
Adam7 (etap 1)



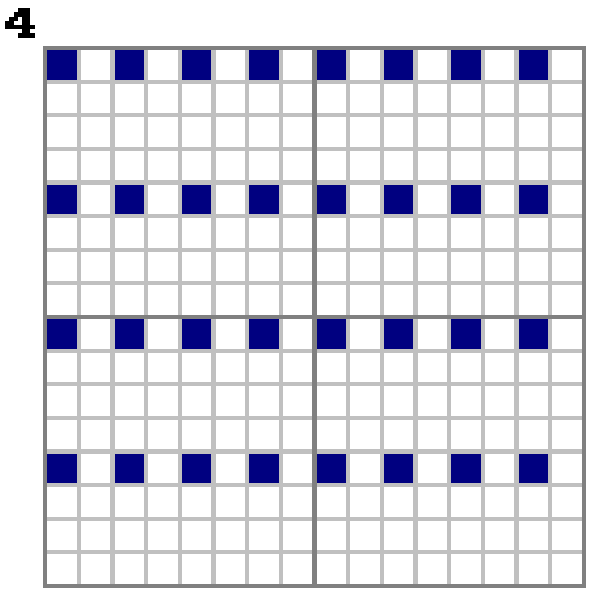
Adam7 (etap 2)



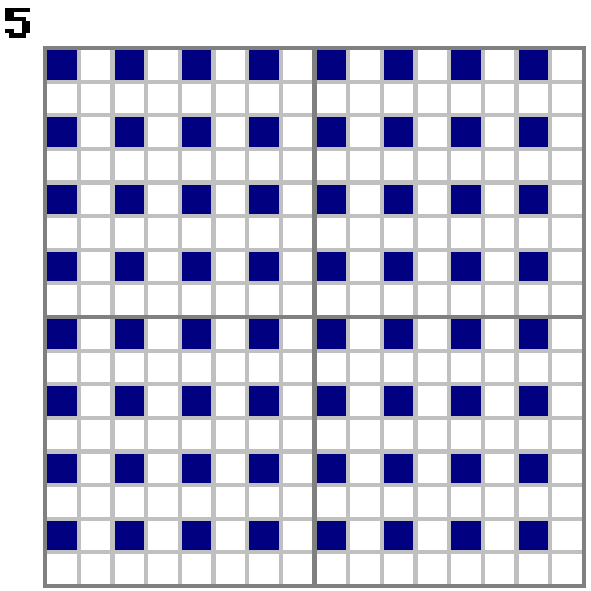
Adam7 (etap 3)



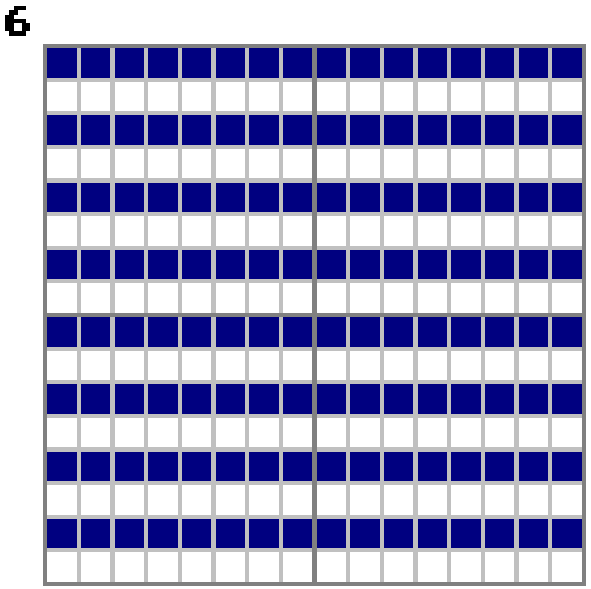
Adam7 (etap 4)



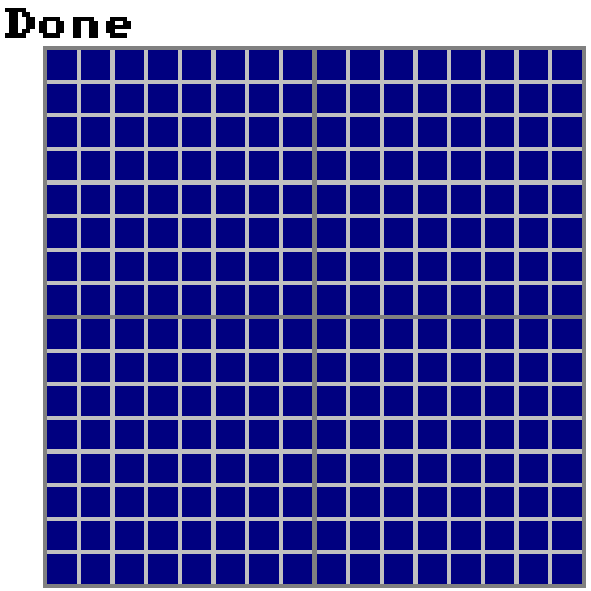
Adam7 (etap 5)



Adam7 (etap 6)



Adam7 (etap 7)



Podsumowanie

- Format PNG to prosty format - jego założenia to nie najbardziej efektywna kompresja, tylko przenośność i prostota.
- Sercem kompresji jest bezstratny algorytm "kompresji ogólnej", nie zaprojektowany pod kodowanie bajtów obrazu.
- Na jego skuteczność wpływają filtry które transforumją obraz do znacznie bardziej przystępnej postaci.
- Znacznie lepsze efekty kompresji można osiągnąć stosując algorytm kompresji stratnej - np. JPG.
- PNG jest jednak całkiem skuteczny na "typowych" obrazach - stopień kompresji pomiędzy 2.5 a 3.5.
- Jest jednak ustandaryzowany, łatwo rozszeżalny, a do dekompresji można użyć bardzo popularnej biblioteki na dogodnej licencji.

Referencje

1. T. Boutell,
   "PNG (Portable Network Graphics) Specification"
2. L. Peter Deutsch,
   "DEFLATE Compressed Data Format Specification (RFC 1951)"
3. L. Peter Deutsch, J-L. Gailly,
   "ZLIB Compressed Data Format Specification"
4. Ziv J., Lempel A.,
   "A Universal Algorithm for Sequential Data Compression"

Dziękuję za uwagę.