Politechnika Krakowska
im. Tadeusza Kościuszki

Wydział Inżynierii Elektrycznej i Komputerowej
Informatyka
Grafika Komputerowa i Multimedialna

Dokumentacja projektu

Temat: Kompresja obrazków BMP do własnego 12 bitowego formatu i dekompresja z powrotem do BMP, z wykorzystaniem algorytmów: Upakowanie bitowe, Huffman, LZ77.

Prowadzący: mgr Kamil Nowakowski

Autorzy:

Marek Więcek (Algorytm kompresji upakowanie bitowe)
Marcin Małecki (Algorytm kompresji Huffmana)
Damian Nowakowski (Algorytm kompresji LZ77)
Konrad Obal (Interfejs I/O, Skala szarości, Optymalizacje)

Opis programu

Program napisany jest w języku C++14 i skompilowany w kompilatorze wbudowanym w IDE Visual Studio 2016 Community w wersji Release/x64 na platformę Windows. (Kod źródłowy jest przystosowany do kompilacji na systemach unixowych).
Komunikacja z programem odbywa się poprzez konsolę, wszelkie informacje odnośnie komend dostępnych dla programu, są dostępne po użyciu komendy „-h" lub „--help" na pliku wykonywalnym.
Do operacji na plikach BMP została użyta biblioteka SDL2 (https://www.libsdl.org/download-2.0.php), w związku z czym plik „SDL2.dll" dołączony do programu, jest niezbędny do jego uruchomienia, lub na systemach unixowych istnieje potrzeba niezależnej instalacji tej biblioteki.

Możliwości:

- Kompresja plików dowolnych plików BMP ze składowymi RGB do własnego formatu.
  (Jeśli jest obecna składowa przeźroczystości A jest ona pomijana)
- Dekompresja plików własnego formatu do plików BMP
  (ze względu na specyfikę biblioteki SDL2, obrazek jest zapisywany jako 24-bitowy, chociaż skompresowany plik ma głębie tylko 12-bitową)
- Transformacja obrazków do skali szarości używając standardu **ITU-R BT.709** (https://en.wikipedia.org/wiki/Rec._709)
- Podgląd plików własnego formatu oraz BMP
  (za pomocą komend w konsoli, lub metodą „Drag and Drop")

Struktura własnego formatu:

Rozpoznawana nazwa rozszerzenia pliku to „rgb12".
Wielkość podstawowego nagłówka to 63 bajty.

Podstawowy nagłówek:

- Liczba określająca wielkość poniższego C-stringu (8 bajtów)
- „Magiczny" C-string „.rgb12" (6 bajtów)
  Stosowany jest po to, aby uniknąć sytuacji ładowania pliku, gdy ktoś sobie zmieni nieodpowiedzialnie rozszerzenie na „.rgb12".
- Szerokość obrazka (8 bajtów)
- Wysokość obrazka (8 bajtów)
- Zastosowany algorytm kompresji (1 bajt)

Reszta pliku:

- (opcjonalnie) Własny nagłówek algorytmu
- Binarne dane obrazka specyficzne dla algorytmu

Obsługa błędów

Polega tylko na wyświetleniu błędu na standardowy strumień błędów oraz pominięciu wykonywanego działania. Po zdefiniowaniu makra **_DEBUG** w preprocesorze są wyświetlane dodatkowe informacje o działaniu programu na standardowe wyjście.

Upakowanie bitowe

Upakowanie bitów - 8 bitów na 4 bity

Cel: Stratna kompresja danych

Działanie algorytmu:

Kompresja:

1) Pobranie składowej koloru z pixela.

2) Obcięcie jej wartości do 4 najbardziej znaczących bitów.
3) Umieszczenie jej w odpowiedniej części 8bitowego kontenera
   - ✓ Jeżeli kontener jest pusty, dane wstawiane są do początkowej jego części.
   - ✓ Jeżeli kontener zawiera już dane, następne dane są wstawiane do drugiej jego połówki.
4) Sprawdzany jest stan kontenera, jeżeli jest zapełniony, kontener zapisywany jest do pliku wynikowego.
5) Wybierana jest kolejna składowa, lub kolejny pixel i następuje przejście do kroku 2.

Dekompresja:

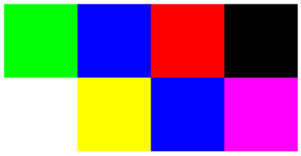
1) Pobierany jest 8bitowy kontener z bufora
2) Uruchamiana jest pętla (przechodząca po wszystkich pixelach) a w niej pętla przechodząca po 3 składowych(czerwona, zielona, niebieska)

   a) Następuje pobranie 8bitowego kontenera z buffora
   b) Sprawdzanie stanu bufora
   c) Jeżeli z kontenera nie było pobranych żadnych danych, pobierane są z niego pierwsze 4 bity
   d) Jeżeli pobrano już jakieś dane z kontenera, pobierane są pozostałe(ostatnie) 4 bity
   e) Wstawianie danych wybranej (w pętli) składowej pixela
   f) Jeżeli pobrano wszystkie dane z kontenera, następuje pobranie kolejnego kontenera

Algorytm działa na identycznej zasadzie dla zapisu obrazków w skali szarości, z tym, że zapisywana jest tylko jedna składowa (ponieważ wszystkie są takie same), dzięki czemu każdy pixel ma tylko 4 bity (nie 12), jeśli ilość bitów jest nieparzysta, to ostatnie 4 bity mają wartość 0 i nie są wykorzystywane przy odczycie. Wielkość skompresowanego pliku jest 3 razy mniejsza niż przy zapisie w „kolorze”.

Zalety: Mała złożoność obliczeniowa, mała ilość zajmowanej pamięci
Wady: Strata jakości

Kompresja



R:0000 0000 -> 0000

Kontener = 0000 0000
G: 1111 1111 -> 1111

Kontener = 0000 1111
Kontener zapisywany jest do pliku i jest zerowany
B: 0000 0000 -> 0000
Kontener = 0000 0000

R:0000 0000 -> 0000

Kontener = 0000 0000 Kontener zapisywany jest do pliku i jest zerowany
G: 0000 0000 -> 0000

Kontener = 0000 0000
B: 1111 1111 -> 1111
Kontener = 0000 1111

Następne pixele i ich składowe będą przetworzone analogicznie.

Dekompresja

Pobierany jest kontener z buffora, z wartością binarną 0000 1111
Wyciągana jest jego pierwsza część czyli 0000 i wstawiane jest to do pierwszej składowej pierwszego pixela (R: 0000 G: 0000 B:0000).
Wyciągana jest druga część z kontenera czyli 1111 i wstawiane jest to do kolejnej składwej pixela(R; 0000 G: 1111 B:0000).
Wyczerpano już kontener więc pobierany jest następny z wartością 0000 0000.
Wyciągana jest jego pierwsza część czyli 0000 i wstawiana jest do niebieskiej składowej pixela(R:0000 G: 1111 B:0000).

Wybierana jest druga część kontenera(0000) i wstawiany jest on do pixela(R: 0000 G: 0000 B:0000).
Kontener został wyczerpany, więc pobierany jest następny z wartością 0000 1111
Pierwsza cześć kontenera jest wstawiana do zielonej składowej pixela(R:0000 G: 0000 B:0000)
Pozostała cześć wstawiana jest do niebieskiej składowej pixela(R:0000 G:0000 B:1111)

Dalej algorytm działa analogicznie

Algorytm Huffmana

Algorytm kodowania Huffmana jest algorytmem bezstratnej kompresji danych. Działanie algorytmu Huffmana polega na określeniu prawdopodobieństw wystąpień $P = \{p_1, ..., p_n\}$ symboli $S = \{x_1, ..., x_n\}$ danego alfabetu.

Kodowanie Huffmana:

1) Określamy prawdopodobieństwo, bądź częstość wystąpienia dla każdego symbolu ze zbioru S.
2) Tworzymy listę drzew binarnych, przechowujących w węzłach symbol oraz prawdopodobieństwo (częstość) wystąpienia danego symbolu. Początkowo drzewa w liście będą stanowiły same korzenie.
3) Dopóki na liście jest więcej niż jedno drzewo, powtarzamy:
    a) Usuwamy z listy dwa drzewa z najmniejszymi prawdopodobieństwami (częstościami) wystąpień[1]
    b) Tworzymy nowe drzewo, przechowujące w korzeniu sumę prawdopodobieństw (częstości) wystąpień symboli drzew usuniętych w punkcie 3a. Usunięte w punkcie 3a drzewa stają się lewym i prawym poddrzewem nowo utworzonego drzewa. Dodajemy nowe drzewo do listy.

Po zakończeniu powyższej procedury powstanie tzw. **drzewo Huffmana,** w którym w korzeniu przechowywane jest prawdopodobieństwo równe 1, a symbole do zakodowania wraz z ich prawdopodobieństwami (częstościami) wystąpień zapisane są w liściach.

Powstałe drzewo Huffmana posłuży do wyznaczenia kodów Huffmana, w następujący sposób:

1) Każdej lewej krawędzi każdego węzła przypisujemy wartość **0**,
a każdej prawej krawędzi wartość **1**
2) Przechodzimy drzewo od gorzenia do każdego liścia,
do kodu z punktu 1. dopisujemy **0**, gdy przechodzimy do lewego poddrzewa,
natomiast **1** gdy przechodzimy do prawego poddrzewa.

Powstałe kody Huffmana stanowią kod dla każdego symbolu przechowywanego w liściu.

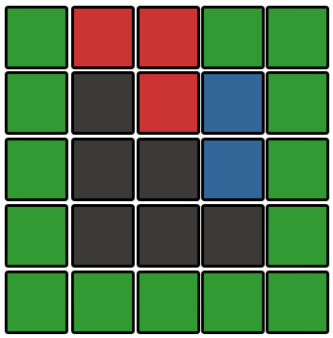
Przykład

Obrazek wielkość: 5*5 = 25

Zbiór symboli: wartość koloru każdego piksela

Prawdopodobieństwo: ilość wystąpień każdego koloru w całym obrazie

Niech: **G** - zielony, **K** - czarny, **R** - czerwony, **B** - niebieski

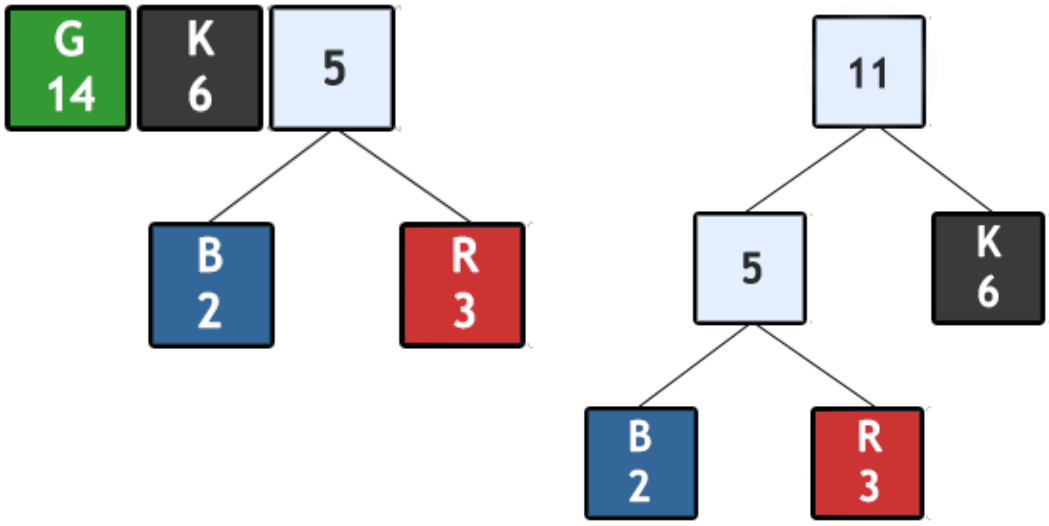


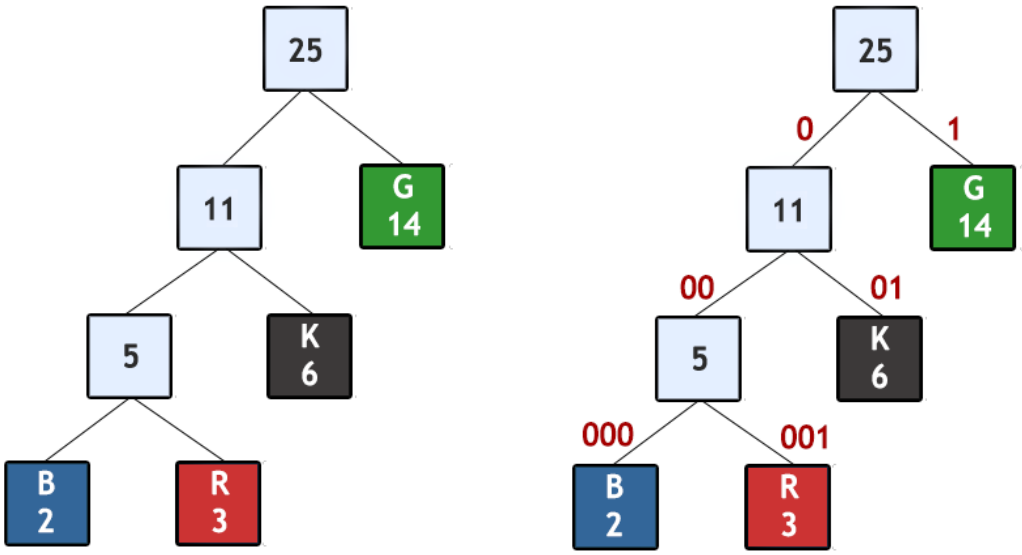
1) Zliczamy ilość wystąpień każdego koloru

G - 14, K - 6, R - 3, B - 2

2) Tworzymy listę drzew.

3) Budujemy drzewo i wyznaczamy kody





Jak wynika z przykładu symbole o największej częstości występowania otrzymują kody najkrótsze, natomiast te które rzadko występują otrzymują kody dłuższe.

Dekodowanie pliku graficznego wygląda podobnie do kodowania.

Z wczytanego nagłówka obrazka zawierającego pary kolor - ilość wystąpień jesteśmy w stanie odbudować drzewo użyte do kodowania.

Po odbudowaniu drzewa algorytm dekodowania przebiega następująco

1. Z pliku wczytywana jest bajt zawierający ciągu wartości 0 i 1.
2. Sprawdzamy czy w drzewie istnieje symbol o kodzie Huffmana równym pierwszemu bitowi wczytanego bajtu w kroku 1.
   Jeżeli symbol o kodzie nie znaleziony -> krok 3.
   Jeżeli symbol o kodzie znaleziony -> krok 4.
3. Pobieramy kolejny bit (wartość 0/1) z wczytanego w kroku 1 bajtu, dołączamy do istniejącego ciągu wartości szukanego kodu i ponownie przeszukujemy drzewo (krok 2).
4. Jeżeli w drzewie został znaleziony symbol o kodzie szukanym to go pobieramy i zerujemy ciągu naszego klucza przeszukiwań drzewa.
5. Jeżeli cały plik wczytany to kończymy algorytm.

Działanie algorytmu Huffmana w projekcie

Implementacja algorytmu Huffmana w projekcie przebiega według powyższej procedury. Przy czym dla obrazka o jednolitym kolorze dodawany jest do drzewa pusty węzeł - kod koloru otrzymuje wówczas wartość 1. Zapis do pliku polega na zastąpieniu danego koloru odpowiednim dla niego kodem. Kody wpisywane do pliku nie są rozszerzanego do bajtów - wpisywane są 'bitami'. Podejście takie wydłuża czas kodowanie oraz znacznie bardziej czas dekodowania, ale pozwala uzyskać wiele wyższy stopień kompresji - dlatego zdecydowano się na realizację takiego podejścia.

Kodowanie wartości poszczególnych pikseli bardzo dobrze sprawdza się dla plików graficznych zawierających wiele pikseli tego samego koloru, jednakże dla obrazów mających zróżnicowane kolory algorytm Huffmana może dać słabszy stopień kompresji - w zależności od poziomu zróżnicowania kolorów.

W celu zdekodowania obrazka wymagane jest odtworzenie drzewa - identycznego do użytego podczas kodowania - co jest realizowane poprzez zapisanie w nagłówku obrazka kolorów oraz odpowiadającym im liczby wystąpień (co niestety powoduje rozrost wynikowego rozmiaru pliku graficznego).

Algorytm LZ77

Opis LZ77 i wyjaśnienie przyjętej koncepcji

Algorytm LZ77 (Lempel-Ziv 77) do kompresji danych wykorzystuje powtarzające sie sekwencje bajtów, które są zapisywane w krótszej postaci, informującej o długości powtarzającego się ciągu oraz gdzie wcześniej wystąpił ten ciąg. LZ77 jest algorytmem kompresji dynamicznej, co oznacza że plik jest analizowany i kompresowany podczas napływania kolejnych bajtów informacji, w przypadku niniejszego projektu tymi bajtami są kolejne podpiksele określające kolory obrazka. Algorytm LZ77 znajduje zastosowanie w takich formatach jak ZIP, gzip, ARJ, RAR, PKZIP, czy PNG.

Algorytm wykorzystuje dwa bufory, zaimplementowane jako mapy. Jednym z nich jest bufor słownikowy, w programie określony jako *s_buff*, drugim bufor wejściowy(w kompresji)/wyjściowy(w dekompresji), określony w programie jako *la_buff*. Oba bufory mają z góry określone rozmiary. Bufor wejściowy przechowuje na bieżąco odczytywane wartości kolorów, które mają zostać zakodowane, natomiast bufor słownikowy przechowuje wartości, które zostały ostatnio zakodowane. Rozmiar bufora wejściowego/wyjściowego może zostać zmieniony, pod warunkiem że na wartość nie mniejszą od 10, gdyż jest to wielkość minimalna tego bufora, wymagana w niniejszym programie do prawidłowego działania, co jest zdeterminowane przez maksymalną długość sekwencji podpikseli, którą program koduje. Rozmiar bufora słownikowego wynosi 17 i nie może przyjmować wartości większej, gdyż przyjęto w projekcie koncepcję, by zapisywać kodowany ciąg maksymalnie jako 1 bajt (sposób kodowania, opisano pod nagłówkiem *Kompresja*), a większy rozmiar bufora słownikowego spowodowałby fakt, że musielibyśmy przeznaczyć więcej bitów niż zaplanowano na określenie pozycji kodowanej sekwencji. Praktycznie bufor słownikowy może przyjmować wartości mniejsze niż 17, ale im mniejsza jest jego wartość tym gorszy jest efekt kompresji, poprzestając na wartości 2, 1 (czy nawet 0, czyli braku słownika w praktyce)- stąd płynie oczywisty wniosek, iż nie ma sensu zmniejszać jego rozmiaru, innymi słowy wartość 17 zapewnia najlepszą kompresję w niniejszym projekcie.

Rozmiar bufora słownikowego wynosi 17, a nie 16 (która odpowiada maksymalnej długości kodowanej sekwencji), gdyż przy kompresji w tym buforze szukane są powtórzenia ciągów kolorów z bufora wejściowego, a minimalna długość sekwencji wynosi 2, tak więc nigdy nie potrzebujemy określać przy kodowaniu sekwencji pozycji ostatnio zapisanego koloru, gdyż nie kodujemy jednej, powtórzonej wartości. Dużą zaletą algorytmu LZ77 jest fakt, że nie trzeba przesyłać słownika z komunikatem, zawartość słownika na bieżąco jest przywracana przez dekoder.

W projekcie zdecydowano się na kodowanie podpiskeli, a nie całych pikseli z tego powodu, że jeden podpiskel może przyjmować 16 różnych wartości, a cały piksel przyjmowałby 4096 różnych wartości, a zatem zależnie od obrazka, byłoby znacznie mniej prawdopodobne znalezienie takiej samej sekwencji w buforze słownikowym jak w buforze wejściowym, a zarazem dużo bardziej prawdopodobne byłoby znajdowanie o wiele krótszych sekwencji, co dawało gorszy efekt kompresji.

Sposób podejścia do tworzenia kodu dla sekwencji (opisany w pod nagłówkiem *Kompresja*) sprawia, że rozmiar docelowego pliku binarnego nie może być większy od rozmiaru kodowanego obrazka, nie licząc nagłówka dodawanego do pliku, a więc w tym przypadku zostaje wyeliminowany problem, który jest cechą charakterystyczną tego algorytmu, a mianowicie potencjalne zwiększenie rozmiaru pliku. Taki zabieg jest oczywiście możliwy przez fakt, iż w projekcie obcinamy 4 najmniej znaczące bity.

Kompresja

Na początku algorytmu bufor słownikowy jest wypełniany pierwszą wartością koloru jaka została odczytana z obrazka. Dzięki temu już w przypadku następnych podkolorów (od drugiego), może nastąpić kompresja (w przypadku, gdy wystąpią przynajmniej 2, takie same wartości kolorów jak w buforze słownikowym). W przypadku kompresji bufor wejściowy jest uzupełniany po każdym, wykonanym kodowaniu sekwencji, lub kodowaniu pojedynczego znaku (do momentu, aż wszystkie dane zostaną odczytane z obrazka).

Kodowanie odbywa się w następujący sposób, w którym rozważane są dwa przypadki. Pierwszym przypadkiem jest sytuacja, w której w buforze słownikowym nie znajdujemy sekwencji takich samych znaków, znajdujących sie na początku bufora wejściowego, tzn. jedynie wartość pierwszego koloru w buforze wejściowym znajduje się w buforze słownikowym, albo w ogóle ta wartość tam nie występuje, ale już dla pierwszego i drugiego elementu bufora wejściowego nie znajdujemy odpowiedniej sekwencji wartości kolorów w buforze słownikowym. W przypadku, gdy tylko jeden podpiksel ma swojego odpowiednika w buforze słownikowym, nie kodujemy go jako sekwencji gdyż nam to nic nie da (w obu przypadkach musimy przekazać ten jeden podpiksel na wyjście jako 1 bajt). Zapisujemy go jako bajt, którego najstarszy bit ma wartość 0, co pozwala nam identyfikować ten przypadek przy późniejszym dekodowaniu.

Drugim przypadkiem jest sytuacja, gdy pierwsze wartości początkowe kolorów w buforze wejściowym mają swój odpowiednik w postaci takiej samej sekwencji w buforze słownikowym, przy czym kodujemy sekwencję o minimalnej długości 2, a maksymalnej długości 9. Minimalna długość sekwencji jest uzasadniona powyżej, natomiast maksymalna długość sekwencji jest związana ze sposobem zapisu zakodowanej sekwencji, przyjętym w niniejszym projekcie, a mianowicie zakodowane dane są zapisywane jako 1 bajt. Najstarszy bit w przypadku kodowania sekwencji jest równy 1, kolejne 3 bity określają nam długość powtarzającej się sekwencji w buforze słownikowym, a ostatnie 4 pozycję początku sekwencji w buforze słownikowym. 3 bity określające długość sekwencji, pozwalają na uzyskanie jej maksymalnej długości 9,

gdyż nie kodujemy danych dla 0 powtarzających się wartości kolorów z bufora wejściowego w buforze słownikowym, ani dla 1 wartości koloru, co oznacza że możemy wykorzystać ten fakt i przez wartość tych 3 bitów równej 0 rozumiemy, że kodujemy 2 podpiskele, a przez wartość 3 bitów równą 1 rozumiemy, że kodujemy 3 podpiskele i tak dalej, aż do wartości 3 bitów równej 7, co oznacza, iż kodujemy 9 podpiskeli- uzyskuje sie to przez odjęcie od znalezionej sekwencji powtarzających się podpikseli wartości 2. Ostatnie cztery bity określające pozycję początkową sekwencji, wskazuję na pozycję w buforze słownikowym, przyjmując wartości od 0 do 15, przy czym najmniejsza wartość wskazuje na pierwszą wartość w buforze słownikowym (najbardziej odległą w danym momencie kodowania, nie licząc inicjalizacji), a kolejne na następne wartości (czyli kolejne po prawej).

Dekompresja

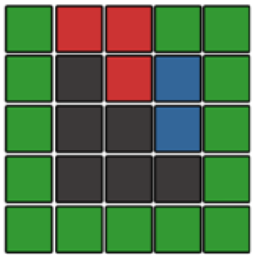
Dekompresja jest łatwiejsza w implementacji niż kompresja. Na początku dekompresji bufor słownikowy jest wypełniany pierwszą, odkodowaną z pliku binarnego wartością koloru. W każdych kolejnych krokach (łącznie z pierwszym wypełnieniem) mamy bufor, który jest wypełniany dokładnie tymi samymi wartościami jak w przypadku kompresji.

Odczytujemy z pliku binarnego kod, który podczas wykonaniu dekodowania, jest zapisywany do bufora wejściowego (który stanowi tak jak dla kompresji bufor wyjściowy *la_buff*) albo jeden kolor, albo ich sekwencja, a w momencie, gdy uzbierają się trzy są ustawiane w odpowiednim miejscu obrazka. Do czynienia z jednym kolorem mamy, gdy najstarszy bit odczytanego bajta jest równy 0, natomiast z sekwencją mamy do czynienia gdy najstarszy bit jest równy 1. Ilość elementów jest zapisana na trzech kolejnych bitach, które odczytujemy i dodajemy do nich wartość 2 (dlaczego zostało wyjaśnione pod nagłówkiem *Kompresja*). Ostatnie cztery bity informują nas o pozycji początkowej odczytanej sekwencji w buforze słownikowym (wyjaśnione pod nagłówkiem *Kompresja*).

Przykład działania mechanizmu

Zakładamy, że rozmiar bufora słownikowego jak i wejściowego wynosi 6.

Kompresujemy dany poniżej obrazek (zupełnie uproszczony model, by pokazać zasadę działania; w projekcie kodujemy podpiksele, więc ilość wszerz kolorów byłaby wielokrotnością liczby 3):



Niech:

- G- zielony
- R- czerwony
- B- niebieski
- K- szary

Sposób odczytywania kolorów:

Zaczynamy od lewego, górnego koloru obrazka; idziemy w prawo, a gdy odczytamy ostatni kolor danej linii, przemieszczamy się do linii poniżej, zaczynając odczytywanie jej od lewej strony do prawej itd.

a) Wypełniamy pierwszym kolorem bufor słownikowy, ładujemy dane do bufora wejściowego (od 2. koloru).

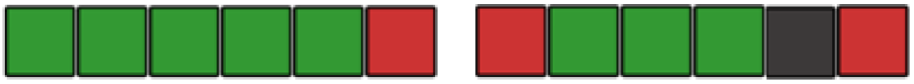
-zapis do pliku binarnego: 0 (jeden kolor) G(wartość koloru)

Bufor słownikowy: Bufor wejściowy:

b) Kolor czerwony nie znajduje się w buforze słownikowym, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 R

Bufor słownikowy: Bufor wejściowy:



c) Kolor czerwony znajduje się w buforze słownikowym, ale nie występuje po nim zielona barwa, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 R

Bufor słownikowy: Bufor wejściowy:

d) Znajdujemy 3 kolory zielone, więc kodujemy sekwencję.

-zapis do pliku binarnego: 1 (sekwencja) 1(długość sekwencji[-2]) 3(pozycja w buforze słownikowym 1[numeracja od lewej, od 0])

Bufor słownikowy: Bufor wejściowy:

e) Kolor szary nie znajduje się w buforze słownikowym, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 K

Bufor słownikowy: Bufor wejściowy:

f) Kolor czerwony znajduje się w buforze słownikowym, ale nie występuje po nim niebieska barwa, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 R

Bufor słownikowy: Bufor wejściowy:

g) Kolor niebieski nie znajduje się w buforze słownikowym, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 B

Bufor słownikowy: Bufor wejściowy:

h) Dwa kolory zielone znajdują się w buforze słownikowym, co więcej kolor szary, więc kodujemy tę sekwencję o długości 3.

-zapis do pliku binarnego: 1 1 1

Bufor słownikowy: Bufor wejściowy:

i) Kolor szary znajduje się w buforze słownikowym, ale nie występuje po nim niebieska barwa, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 K

Bufor słownikowy: Bufor wejściowy:

j) Kolor niebieski znajduje się w buforze słownikowym, następnie po nim znajdujemy dwa kolory zielone oraz dwa kolory szare, mamy sekwencję o długości 5.

-zapis do pliku binarnego: 1 3 1

Bufor słownikowy: Bufor wejściowy:

k) Kolor szary znajduje się w buforze słownikowym, ale nie występuje po nim zielona barwa, więc nie znajdujemy sekwencji.

-zapis do pliku binarnego: 0 K

Bufor słownikowy: Bufor wejściowy:

l) W buforze słownikowym znajdujemy dwa kolory zielone, więc mamy sekwencję o długości 2.

-zapis do pliku binarnego: 1 0 1

Bufor słownikowy: Bufor wejściowy:

m) W buforze słownikowym znajdujemy dwa kolory zielone, więc mamy sekwencję o długości 2.

-zapis do pliku binarnego: 1 0 4

Bufor słownikowy: Bufor wejściowy:

n) W buforze słownikowym znajdujemy dwa kolory zielone (tyle pozostało w buforze wejściowym), więc mamy sekwencję o długości 2.

-zapis do pliku binarnego: 1 0 4

Bufor słownikowy: Bufor wejściowy:

Koniec kompresji, gdyż nie ma już danych w buforze wejściowym, wszystkie kolory zostały przetworzone.