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.

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:

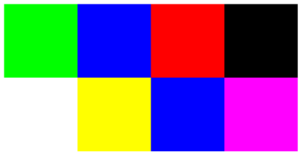
1) Pobierany jest 8bitowy kontener z bufora
2) Uruchamiana jest pętla (przechodząca po wszystkich pixelach) a w niej pętla przechodząca po 3 składowych(czerwona, zielona, niebieska)

   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.

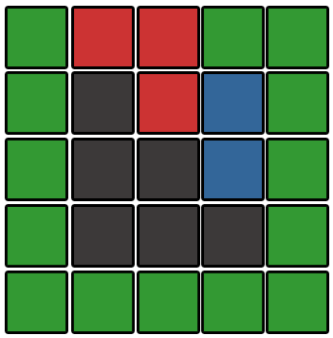
Przykład

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

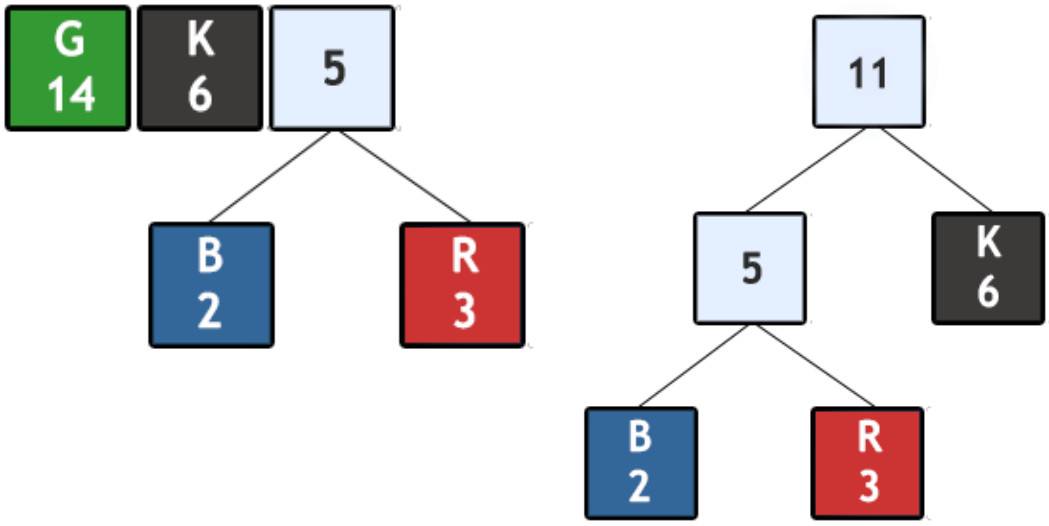


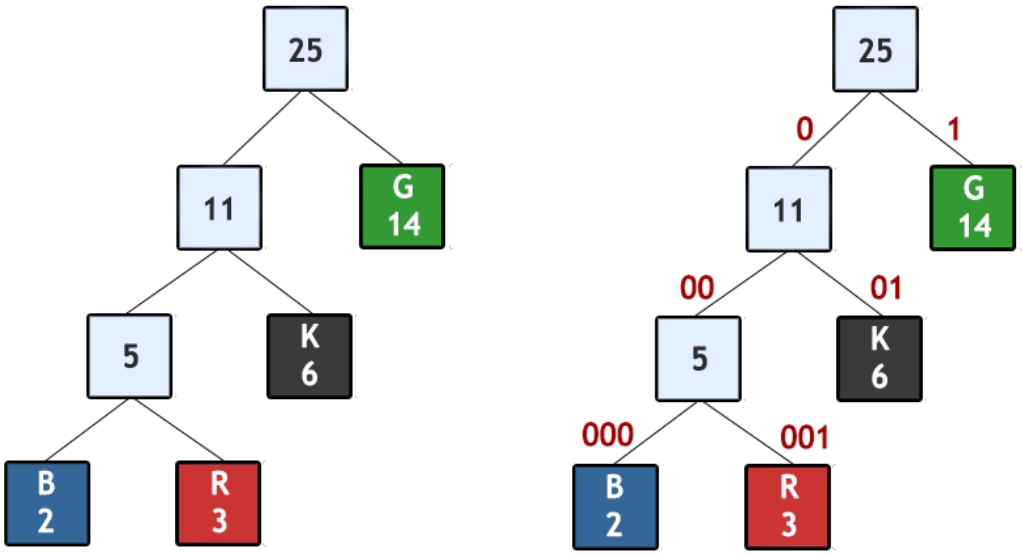
1) Zliczamy ilość wystąpień każdego koloru

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).