Piotr Skorek

Kompresja danych

Projekt kodeka RLE

Założenia:

W założeniu projektu było wykonanie kodera i dekodera RLE (run-lenght encoding). Kompresja RLE polega na znajdowaniu dużej liczby takich samych bajtów występujących po sobie i grupowaniu ich**.** Każda taka grupa bajtów zapisywana jest za pomocą dwóch bajtów. Pierwszy bajt określa ile razy dana wartość powtórzyła się w tej grupie, natomiast drugi bajt określa wartość. Jeżeli mamy ciąg jakiś bajtów np.:5 5 5 5 5 5 5 8 8 8 9 2 4 6 6 6 6 6 6 4 3 , to po zakodowaniu go za pomocą RLE otrzymamy 7 5 3 8 1 9 1 2 1 4 6 6 1 4 1 3. Czyli z wejściowego ciągu 21 bajtów na wyjściu otrzymaliśmy 16 bajtów. Jeżeli ciąg wejściowy składałby się z bajtów które nie występują po sobie po kolei to kompresji by w ogóle nie było (liczba bajtów na wyjściu byłaby większa niz. na wejściu) . Nieco lepsze efekty można uzyskać kodując liczbę powtórzeń bajtów za pomocą liczby ze znakiem od -128 do 127 a nie jak poprzednio od 0 do 255. Wówczas za pomocą liczb ujemnych zapisujemy liczbę niepowtarzających się bajtów. Jeśli weźmiemy ten sam ciąg wejściowy czyli: 5 5 5 5 5 5 5 8 8 8 9 2 4 6 6 6 6 6 6 4 3 to na wyjściu, po zakodowaniu przy pomocy bajtów ze znakiem otrzymamy 7 5 3 8 -3 9 2 4 6 6 -2 4 3 czyli 13 bajtów. Liczba bajtów na wyjściu zmniejszyła się bowiem zamiast 1 9 1 2 1 4 mogliśmy to zapisać -3 9 2 4 podobnie ciąg 1 4 1 3 zastąpiliśmy 3 bajtami -2 4 3. Mimo tych modyfikacji jest to bardzo prosty sposób kompresji i niezbyt efektywny, choć wykorzystuje się go do kompresji grafiki rysowanej ręcznie, bowiem obrazy w takiej grafice charakteryzują się duża liczba sąsiadujących ze sobą pixeli o takiej samej wartości. Pliki formatu PCX są zakodowane właśnie ta metoda.

Znacznie ciekawszym rozwiązaniem jest koder działający podobnie do RLE, ale zamiast, szukać i kodować ciągi bajtów o takiej samej wartości, to lepiej kodować ciągi powtarzających się takich samych bitów. Czyli kodować ciągi występujących po sobie 1 lub 0 Ponadto liczbę powtórzeń tych bitów zapisywać za pomocą kodu Golomba, co sprawia ze nie musimy z góry przewidywać jaka będzie maksymalna liczba powtórzeń i rezerwować bitów tak, aby zapisać liczbę powtórzeń. Przykładowo, jeśli mamy taki ciąg: 1 1 1 1 1 1 1 0 0 0 1 1 to po kompresji będzie on wyglądał następująco: 1 1 1 0 1 1 0 1 1 0 0 ,czyli zapisaliśmy 7 następnie 3 a następnie 2 w kodzie Golomba zakładając rząd równy 2. Jak widać kompresja nie jest duża, bo z 12 bitów otrzymaliśmy 11 a dodatkowo musimy zakodować na początku, od jakiego bitu zaczynamy 1 lub 0 a wiec w

sumie na wyjściu otrzymamy tez 12 bitów. Jednak to, jaka będzie kompresja będzie zależało od tego jakiś mamy ciąg wejściowy i jak sobie dobierzemy rząd kodu Golom ba. Wydaje mi się ze nie należy się spodziewać oszałamiających wynikó w, bowiem kodowanie to mimo ze binarne to opiera się na zasadzie działania RLE. Napisałem koder, który właśnie realizuje takie założenia. Dla ułatwienia rząd kodu Golomba jest wielokrotnością dwójki i jest to parametr wywołania programu. Dzieki temu będzie można zbadać jak zmiana rzędu kodu Golom ba wpływa na stopień kompresji plików.

Opis programu:

Program został napisany w języku C z wykorzystaniem gotowych bibliotek bitio.c służących do wczytania i zapisania plików bitowo, a także do operacji na tych bitach. Plik cprogram.c służy do kompresji natomiast plik eprogram.c służy do dekompresji.

Programu używa się podając nazwę pliku, czyli cprogram albo eprogram a następnie nazwę pliku wejściowego, nazwę pliku wysjciowego oraz rząd kodu Golomba. Podając rząd podajemy jako parametr potęgę dwójki. Za kazdym razem, kiedy źle wywołamy program wypisywana jest informacja o prawidłowym użyciu programu : Usage "plik_wejsciowy plik_wyjsciowy k (m=2^k - parametr kodu Golomb'a)" .

Po uruchomieniu programu wyświetlana jest informacja o tym, jaki plik został użyty jako wejściowy do jakiego pliku zapisywane są dane, a także informacje o rozmiarze pliku wejściowego i pliku wyjściowego a także o stopniu kompresji. Funkcja *long file_size(char *name)* zwraca nam wartosc pozycji końca pliku za pomocą funkcji *int fseek(FILE*, long, int)* oraz *long ftell (FILE*).* Natomiast funkcja *void print_ratios(char *input,char *output)* oblicza i wyswietla nam współczynnik kompresji. Funkcja *void usage_exit(char *prog_name)* wyświetla nam informacje o tym jak prawidłowo wywoływać program.

Algorytm kompresji realizuje nam funkcja *void CompressFile(BIT_FILE *inf, BIT_FILE *ouf, int argc, char *argv[]).* Pobiera ona najpierw pierwszy bit z pliku wejściowego, a następnie zapisuje go do pliku wyjściowego, po to by wiedzieć od jakich znaków zaczęliśmy kodować : 1 czy 0. Ustawiamy licznik odczytanych bitów na 1. Następnie dopóki nie ma końca pliku zliczam kolejne wystąpienia tych samych bitów zwiększając licznik odczytywanych bitów za każdym razem o 1. Zrezygnowałem z wykrywania końca pliku tak jak jest to robione w funkcji *InputBit(s)* (w pliku bitio.c) bowiem program nie działał. Napotykając koniec pliku wykrzaczal się. Dlatego koniec pliku sprawdzam w prosty sposób za pomocą *feof(inf->file)==0* wewnątrz funkcji kompresji i dekompresji. Dopóki kolejne bity są takie same jak pierwszy wczytany oraz nie napotkamy znaku końca pliku zwiększamy licznik wystąpień. Jeśli wystąpi znak końca pliku lub kolejny wczytany bit jest rożny od poprzednich kończymy zliczanie. I zapisujemy liczbę powtórzeń do pliku wyjściowego w kodzie Golomba. Zrealizowałem to w ten sposób ze przesuwam bitowo w lewo wartość liczby powtórzeń o tyle ile wynosi potęga 2 rzędu kodu Golomb'a. Czyli jeżeli rząd m wynosi 2 to przesuwam o 1 bit w prawo, ponieważ m= 2^1, jeśli rząd by był równy 4 to przesuwam o dwa bity w prawo bo m=4 = 2^2. Inaczej mówiąc realizuje w ten sposób dzielenie liczby powtórzeń przez 2^ k, gdzie k to parametr wczytywany przy wywołaniu programu określający nam rząd kodu Golomb'a m= 2^k. Następnie zapisujemy tyle jedynek ile wynosi liczba po przesunięciu o k bitów a następnie 0(kod unarny). Po czym wypisujemy binarnie resztę z dzielenia liczby powtórzeń przez rząd. Taki algorytm pozwala nam zapisać liczbę w kodzie Golomb'a,ktory składa się z dwóch części: w jednej zapisana jest reszta z dzielenia przez rząd m w zmodyfikowanym kodzie dwójkowym, oraz z drugiej części gdzie jest zapisany wynik dzielenia liczby powtórzeń przez rząd bitowo w kodzie unarnym. Ustawiamy licznik powtórzeń na jeden, bo mamy już wczytany kolejny znak i powtarzamy wszystko aż do końca pliku. Przykładowo, jeśli chcielibyśmy zapisać 1 0 1 w kodzie Golomb'a rzędu 2 (k=1) to powyższa funkcja zrobi nam to w

następujący sposób : przesunie 1 0 1 o 1 bit w prawo, czyli będziemy mieli 1 0. Następnie zapisze tyle jedynek, jaka wartość jest tej liczby przesuniętej, czyli 1 1 , poczym wstawi 0 a następnie zapisze binarnie wynik z dzielenia modulo liczby 1 0 1(5 dziesiętnie) przez rząd, czyli2. Reszta wynosi 1 wiec na samym końcu zapisuje ta 1. Wynik to 1 1 0 1.

Funkcja *void ExpandFile(BIT_FILE *inf, BIT_FILE *ouf, int argc, char *argv[])* realizuje nam algorytm dekompresji. Jako plik wejściowy podajemy mu plik skompresowany wcześniej za pomocą *cprogram.c* . Pobieramy pierwszy bit z naszego pliku wejściowego, następnie dopóki nie ma końca pliku zliczamy jedynki, dopóki nie napotkamy 0. Liczbę jedynek mnożymy przez rząd i zapisujemy do licznika. Następnie odczytujemy zapisana resztę z dzielenia przez rząd i dodajemy do licznika. Do pliku wyjsciowego zapisujemy określoną ilość bitów (takich jak wczytaliśmy pierwszy bit). Następnie negujemy zmienna,pod którą przechowywany był pierwszy bit. Wiemy bowiem ze następny bit będzie przeciwny. Przykładowo biorąc nasza wcześniej zakodowana 5 (1 0 1) czyli po kompresji mieliśmy 1 1 1 0 1 (pierwsza 1 oznacza nam pierwszy bit, od którego zaczynamy kodować). Dekompresując pobieramy pierwszy bit i go zapisujemy. Następnie liczmy kolejne jedynki aż do wystąpienia 0 i zapisujemy w liczniku, czyli mamy 2. Mnożymy wartość licznika przez rząd i zapisujemy ta wartość, czyli mamy 4 bo rząd mamy równy 2. Następnie do tej wartości dodajemy resztę z dzielenia modulo (5 % 2) do tej wartości i w rezultacie mamy wartość licznika ustawiona na 5. Wypisujemy do pliku wyjściowego 5 jedynek i negujemy zmienna, pod którą mieliśmy zapisany pierwszy bit.

Do sprawozdania dołączam wydruk funkcji *CompressFile* i *ExpandFile.*

Testy i obserwacje:

Przy dekompresji pliku nie musimy pamiętać jaki był rząd kodu Golomb'a ,ponieważ rzad jest zapisywany podczas kodowania.

Obserwacje przeprowadzone na podstawie pliku *.doc (56 KB) pokazują ze przy rzędzie równym 4 otrzymujemy najlepsze rezultaty kompresji.
dla

rzedu : 2	inputbytes: 57344	outputbytes: 488741	bit rate: 16%
rzedu : 4	inputbytes: 57344	outputbytes: 45861	bit rate: 21%
rzedu : 8	inputbytes: 57344	outputbytes: 52101	bit rate: 10%
rzedu : 16	inputbytes: 57344	outputbytes: 61195	bit rate: - 6%
rzedu : 32	inputbytes: 57344	outputbytes: 72350	bit rate: -26%

Jak widać najlepsza kompresje uzyskujemy dla rzędu 4 czyli jeśli chodzi o kod Golomba to reszta z dzielenia modulo jest zapisywana na dwóch bitach.

Przeprowadzając takie same próby na pliku *.doc ale większym (865 KB) otrzymujemy nastepujace wyniki:
dla

rzedu : 2	inputbytes:886272	outputbytes: 790498	bit rate: 11%
rzedu : 4	inputbytes:886272	outputbytes:766589	bit rate: 14%
rzedu : 8	inputbytes:886272	outputbytes:886272	bit rate: 0%
rzedu : 16	inputbytes:886272	outputbytes:1052321	bit rate: -18%
rzedu : 32	inputbytes:886272	outputbytes: 1248916	bit rate: -40%

Widać ze pliki tekstowe (*.doc) należą do plików, w których nie ma zbyt dużej liczby

powtórzeń 0 i 1. Świadczy o tym fakt ze najlepsza kompresja zachodzi dla rzędu 4.

Kolejnym plikiem, na którym testowałem mój program była to bitmapa. Takich rezultatów można było się spodziewać, bowiem właśnie na takich plikach najlepiej działa algorytm RLE. Do testu stworzyłem obrazek z czarnymi kwadratami na białym tle.

Dla

rzedu : 2	inputbytes:1760490	outputbytes: 881216	bit rate: 50%
rzedu : 4	inputbytes:1760490	outputbytes: 441594	bit rate: 75%
rzedu : 8	inputbytes:1760490	outputbytes: 222026	bit rate: 88%
rzedu : 16	inputbytes:1760490	outputbytes: 112343	bit rate: 94%
rzedu : 32	inputbytes:1760490	outputbytes: 57723	bit rate: 97%
rzedu : 64	inputbytes:1760490	outputbytes: 30723	bit rate: 99%
rzedu : 128	inputbytes:1760490	outputbytes: 17469	bit rate: 100%
rzedu : 2048	inputbytes:1760490	outputbytes: 6692	bit rate: 100%

Dla rzędu 2048 (parametr 11) uzyskałem najlepszy stopień kompresji. Jak wiadomo mapa bitowa t o długie ciągi zer i jedynek, dlatego uzyskujemy takie dobre wyniki kompresji.
Przetestowałem tez biały obrazek (*.bmp) czyli złożony z samych 1.

Dla

rzedu : 2	inputbytes:1760490	outputbytes: 880571	bit rate: 50%
rzedu : 4	inputbytes:1760490	outputbytes: 440594	bit rate:: 75%
rzedu : 8	inputbytes:1760490	outputbytes: 220717	bit rate: 88%
rzedu : 16	inputbytes:1760490	outputbytes: 110747	bit rate:: 94%
rzedu : 32	inputbytes:1760490	outputbytes: 55924	bit rate: 97%
rzedu : 64	inputbytes:1760490	outputbytes: 28619	bit rate: 99%
rzedu : 128	inputbytes:1760490	outputbytes: 15901	bit rate: 100%

Najlepszy wynik dla tego pliku otrzymuje dla rzędu 8192 (parametr 13).

rzedu : 8192	inputbytes:1760490	outputbytes: 2457	bit rate: 100%

Na podstawie tych przykładów można stwierdzić ze dla plików, w których mamy długie ciągi zer i jedynek najlepszy stopień kompresji uzyskujemy dla dużych rzędów, natomiast dla plików, w których te ciągi nie SA zbyt długie (pliki tekstowe *.doc) najlepsze są rzędy małe.

Potwierdzeniem może być przykład kolejnego pliku. Stworzyłem obrazek, w którym jest dużo linii, dużo szczegółów i kolorów. Przypuszczalnie tak zapisany plik będzie miął krótsze ciągi powtarzających się zer i jedynek a w związku z tym przy mniejszych rzędach będzie osiągnę maximum kompresji. Oto rezultaty przeprowadzonych testów:

dla

rzedu : 2	inputbytes:1760490	outputbytes: 1094919	bit rate: 38%
rzedu : 4	inputbytes:1760490	outputbytes: 775479	bit rate: 56%
rzedu : 8	inputbytes:1760490	outputbytes: 676102	bit rate:: 62%
rzedu : 16	inputbytes:1760490	outputbytes: 673967	bit rate: 62%
rzedu : 32	inputbytes:1760490	outputbytes: 749228	bit rate: 58%
rzedu : 64	inputbytes:1760490	outputbytes: 857047	bit rate: 52%
rzedu : 128	inputbytes:1760490	outputbytes: 971340	bit rate: 45%

Zgodnie z przypuszczeniami dla małych rzędów otrzymujemy najlepsza kompresje (dla rzędu 16),widac tez dobrze na tym przykładzie ze im krótsze są ciągi zer i jedynek tym mniejszy powinien być rząd. Dla rzędu 128 stopień kompresji wyniósł tylko 45% i wraz ze wzrostem rzędu stopień ten malał.

Pliki typu jpg nie dają się skompresować tym algorytmem. Jest to zrozumiale bowiem są one już zakodowane i taki sposób kompresji nie zmniejszy na pewno ich rozmiaru. Dla przykładu jeśli weźmiemy zdjęcie to dla:

rzedu : 2	inputbytes:35094	outputbytes:48081	bit rate: -33%
rzedu : 4	inputbytes:35094	outputbytes:57082	bit rate: -58%

Warto porównać działanie tego algorytmu z jakimś innym Kodakiem binarnym. Zrobię to na przykładzie kodeka arytmetycznego. Kodowanie arytmetyczne jest znacznie bardziej zaawansowanym algorytmem a co się z tym wiąże daje o wiele lepsze rezultaty.Swoje działanie opiera na zasadzie rozróżniania sekwencji symboli na podstawie jakiejś liczby z zakresu (0 , 1). Jak wiadomo takich liczb z tego zakresu jest nieskończenie wiele czyli praktycznie każdej sekwencji symboli można przyporządkować unikalna liczbę. Oto wyniki przeprowadzone na tych samych plikach:

plik (*.doc) o rozmiarze 56 KB po kompresji ma rozmiar 23,2 KB

Natomiast

plik (*.doc) o rozmiarze 865 KB po kompresji ma rozmiar 308 KB

na tym przykładzie widać ze kodowanie arytmetyczne jest o lepsze, rozmiary plików po kompresji kodekiem arytmetycznym są mniejsze od plików skompresowanych przy najlepiej dobranym rzędzie.

Jeśli chodzi o pliki *.bmp to rezultaty kompresji za pomocą kodeka arytmetycznego są gorsze w porównaniu do kodowania metoda RLE.

plik(*.bmp – czarne kwadraty na białym tle) o rozmiarze 1,67MB po kompresji ma rozmiar 23,2KB ,a w przypadku kodeka RLE po kompresji rozmiar wynosi 6,6 kB.

plik (*.bmp – biały obrazek) o rozmiarze 1,67MB po kompresji ma rozmiar 1,65 KB

plik (*.bmp – duża ilość kolorów i szczegółów) o rozmiarze 1,67MB po kompresji ma rozmiar 241 KB

Czyli metoda kodowania za pomocą RLE okazuje sie lepsza dla plikow typu bmp od kodowania arytmetycznego,ale tylko w przypadku pliku z czarnymi kwadratami na białym tle.To potwierdza teze że stopień kompresji zależy glównie od charakteru danych wejsciowych.

Pliki typu *.jpg podobnie jak w przypadku kodeka RLE nie dało się skompresować kodekiem arytmetycznym

Moznaby się zastanowić nad poprawa nieco algorytmu RLE. Na przykład można by spróbować zapisać bity w ten sposób ze zapisujemy tylko ciągi 0 a w momencie kiedy następuje zmiana bitu to zaznaczamy to zmianę 1. Na przyklad mając taki ciąg bitów wejściowych: 1 1 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 0 0 to można by było go zapisać w ten sposób 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0. Czyli powtarzające się ciągi zer lub jedynek zapisujemy za pomocą tylu zer ile razy się powtórzy zero lub jedynka po czym zaznaczamy zmianę ciągu za pomocą 1. Być może poprawiłoby to nieco działanie tego algorytmu ale i tak wszystko by zależało od pliku wejściowego bo jeżeli nie byłoby długich ciągów zer i jedynek to nawet takie wstępne uporządkowanie bitów niewiele by poprawiło działanie. Moznaby się zastanowić nad stworzeniem adaptacyjnego kodeka wykorzystującego ta metodę kompresji, czyli dostosowywać się do lok olanej charakterystyki zbioru danych. Na przykład adaptacyjnie zmieniać rząd kodu Golomba lub tez na samym początku statystycznie wyznaczać najlepsza wartość rzędu. Myślę ze mogłoby to nieco poprawić i usprawnić działanie tego algorytmu, ale i tak ten sposób kodowania nie będzie lepszy od adaptacyjnego kodaka arytmetycznego, który pozwala uzyskać duża skuteczność kodowania.

Wnioski:

Duża cześć wniosków została już przytoczona wcześniej. Kodowanie wykorzystujące działanie RLE nie należy z pewnością do najbardziej efektywnych metod kompresji. Kodek ten sprawdza się przy kompresji plików *.bmp bowiem w tych plikach mamy do czynienia z długimi ciągami zer i jedynek i tylko podobnie zbudowane pliki warto kompresować za pomocą takiego kodeka. Próby ulepszenia tego kodeka poprzez zastosowanie metod statystycznych na przykład przy ustalaniu rzędu kodu Golomba prawdopodobnie nie poprawiłyby stopnia kompresji dla plików, w których mamy małe ciągi zer i jedynek.

Funkcja służąca do kompresji:

```
void CompressFile(BIT_FILE *inf, BIT_FILE *ouf, int argc, char *argv[])
{
 int i,b,k,m;
 int cnt,x,d;

 k=atoi(argv[0]);       // pobierz argument wywolania programu (liczba k)
 m=pow(2,k);            // m=2^k

 OutputBits(ouf,k,8);  // zapisz do pliku wyjsciowego bajt okreslajacy liczbe k

 b=InputBit(inf);       // pobierz pierwszy bit z pliku wejsciowego
 OutputBit(ouf,b);      // zapisz go do pliku wyjsciowego
 cnt=1;                         // licznik odczytanych bajtów na 1

 while (feof(inf->file)==0)    // dopoki nie ma konca pliku
 {
```

```
  while((i=InputBit(inf))==b)          // licz kolejne wystapienia tych samych bitow
  {
   if (feof(inf->file))
    break;                             // koniec pliku - wyjdz z petli
   cnt++;                              // zliczanie kolejnych wystapien takich samych bitow
  }
  b=i;                                 // bedziemy zliczac nowe bity
  x=cnt>>k;                            // podziel - przesun bitowo w lewo (x=cnt/(2^k))
  for (d=0;d<x;d++)
   OutputBit(ouf,1);                   // wypisz x jedynek (kod Golomba)
  OutputBit(ouf,0);                    // zero konczace kod
  OutputBits(ouf,cnt%m,k);             // wypisz binarnie reszte z dzielenia cnt mod m
  cnt=1;                               // juz zliczono nowy bit  - ustaw licznik na 1
 }
}
```

Funkcja służąca do dekompresji:

```
void ExpandFile(BIT_FILE *inf, BIT_FILE *ouf, int argc, char *argv[])
{
 int b,m,k;
 int cnt,d;

 k=InputBits(inf,8);    // pobierz bajt z pliku wejsciowego okreslajacy liczbe k
 m=pow(2,k);            // m=2^k

 b=InputBit(inf);       // pobierz pierwszy bit z pliku wejsciowego
 cnt=0;                 // zeruj licznik powtorzen bitow

 while (feof(inf->file)==0)    // dopoki nie ma konca pliku
 {
  while(InputBit(inf)==1)      // licz jedynki (kod Golomba)
  {
   if (feof(inf->file))
    return;                    // koniec pliku - wyjdz z funkcji
   cnt++;                      // zliczanie kolejnych wystapien '1' w kodzie Golomba
  }

  cnt*=m;                      // liczba jedynek w kodzie Golomba * m
  cnt+=InputBits(inf,k);       // odczytaj zapisana reszte z dzielenia przez m i dodaj do cnt

  if (feof(inf->file))
   return;                     // jesli osiagnieto koniec pliku - wyjdz (jesli kod skonczyl sie
wczesniej)

  for (d=0;d<cnt;d++)
   OutputBit(ouf,b);           // wypisz okreslona ilosc (cnt) bitow do pliku wyjsciowego

  cnt=0;               // zeruj licznik powtorzen bitow
  b=!b & 1;            // zaneguj bit (nastepny bit bedzie przeciwny)
 }
}
```