ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1995

Seria: INFORMATYKA z. 29 Nr kol. 1306

Artur MIGAS

KOMPRESJA DANYCH. CHARAKTERYSTYKA METODY LZW

Streszczenie. Artykuł prezentuje zasadę działania algorytmu LZW jako wybranej metody bezstratnej kompresji danych. Zawiera opis samej metody jak i szczegółów jej implementacji, to jest rozmiaru i organizacji danych w pamięci oraz wariantów i parametrów, od których wyraźnie zależy działanie metody i uzyskiwana efektywność kompresji.

DATA COMPRESSION. THE LZW METHOD

Summary. The article explains the main idea of the LZW algorithm as well as the principle of its operation. The algorithm is an example of a lossless compression method. The article also includes details useful for implementing the algorithm: tips about the size and organization of the memory storage, variations and parameters of the method that have a strong influence upon its operation and compression efficiency.

COMPRESSION DES DONNÉES. LA MÉTHODE LZW

Resumé. L'article présent le principe d'opération de l'algorithme LZW comme un example de la méthode de compression sans-perte des données. Il contient déscription de la méthode et aussi les détails d'implémentation: taille et l'organisation des données dans une memoire et autant que les variants et les parametres dont dépend fortement l'efficacité de la méthode.

1. Wstęp

Informatyka - ta popularna, która tak niedawno pojawiła się na biurkach osób prywatnych pod postacią komputerów osobistych - to dziedzina tak młoda i tak niezwykle szybko się rozwijająca, jak chyba żadna inna. Wszelkie zmiany przyjmują się niemal natychmiast, rewolucja goni rewolucję. Widomym znakiem tego rozwoju jest na przykład produkowanie wciąż nowszych modeli komputerów osobistych, które natychmiast wypierają starsze, "słabsze" modele, oraz zwiększanie się możliwości wciąż potężniejszych i bardziej rozbudowanych, nowych wersji programów, co pociąga za sobą oczywiście większe wymagania sprzętowe. Wszelkie liczby, takie jak prędkości transmisji i pojemności dysków, szybkości procesorów, pojemności pamięci operacyjnych po prostu zwiększają się.

Rosną nie tylko programy. We współczesnym świecie, który mknie do przodu niemal tak szybko, jak sama informatyka, otaczani jesteśmy coraz większą ilością informacji i coraz bardziej jesteśmy od niej uzależnieni. Informacja ta przechowywana jest w plikach o coraz większej objętości. Dyski, o których myśleliśmy jeszcze wczoraj, że są gigantyczne, dziś zaczynają nam nie wystarczać. Istniejąca w USA baza tekstów prawniczych LEXIS ma obecnie 40 GB, a tygodniowo przybywa około 3 MB.

Informacja nie jest przechowywana dla samej informacji, lecz po to, aby można z niej było korzystać. W celu udostępnienia informacji użytkownikowi, nierzadko odległemu, należy mu ją przesłać. Ilość danych przesyłanych po różnorakich łączach również rośnie gwałtownie.

Jednak nie odbywa się to bez żadnych kosztów. Olbrzymie dyski twarde, mieszczące olbrzymie bazy danych, kosztują dużo. W popularnym systemie sieciowym *Novell Netware*, jak i w wielu systemach wielodostępnych, zaimplementowano mechanizmy, służące do rozliczania użytkowników nie tylko z czasu użytkowania serwera i korzystania z wielu jego usług, ale również za samo przechowywanie danych na dysku sieciowym. Informacja stała się towarem, a udostępnianie jej - usługą. W USA zysk z eksportu takich usług informatycznych wyniósł w 1985 roku 7 miliardów dolarów. Za czas zajętości łącza przy przesyłaniu coraz większych ilości danych też trzeba płacić.

W związku z tą sytuacją narodził się pomysł, aby istniejące dane zapisać na jak najmniejszej objętości nośnika, czyli dokonać ***kompresji*** tych danych. Zmniejszy to koszty przechowywania, jak również czas, więc i koszt ewentualnego przesyłania. Należy naturalnie pamiętać, że z drugiej strony dane takie przed użyciem wymagać będą dekompresji, co zajmie nieco czasu procesora; opłacalność całego procesu zależy od proporcji zysków

na zajętości nośnika i czasie przesyłania do strat czasu na dekompresję, ale generalnie w większości przypadków stosowanie kompresji jest jak najbardziej korzystne - procesory rozpakowują 10 KB blok danych w 0.1 s, a przesyłać ten blok w sieci rozległej o prędkości transmisji, powiedzmy, 64 kb/s trzeba przez całą sekundę.

Można powiedzieć, że tak jak w biznesie czy gospodarce obowiązującym hasłem - synonimem rozwoju i doskonalenia jest *zwiększać*, *rozszerzać*, tak w dziedzinie przechowywania i przesyłania informacji celem każdego jest *skrócić*, *zmniejszyć*.

Pojęcie kompresji nie jest wcale nowe. Pierwszym zastosowaniem kompresji było wprowadzenie znanego wszystkim znaku ASCII o kodzie 9, czyli tabulacji, zastępującej pewną liczbę - najczęściej osiem - spacji. Jednak dopiero ostatnio, w świetle zjawisk opisanych powyżej, kompresja nabiera naprawdę dużego znaczenia. Szczególnie dużym zainteresowaniem i, jeżeli można tak powiedzieć, zapotrzebowaniem cieszy się zagadnienie kompresji obrazów.

Generalnie mamy dwa rodzaje kompresji: *stratną* i *bezstratną*. W kompresji **bezstratnej** [ang. *lossless*] podczas dekompresji otrzymujemy plik identyczny z wejściowym. Takimi metodami pakuje się programy, dane liczbowe. Druga klasa - **stratne** - [ang. *lossy*] nie pozwala na odzyskanie zbioru identycznego ze źródłowym i jest stosowana tam, gdzie pozbycie się nadmiaru informacji czy też części informacji wraz z zakłóceniami jest pożądane, czyli w kompresji obrazu i dźwięku.

Niniejszy artykuł prezentuje metodę LZW, jej zasadę działania i pewne kwestie związane z implementacją.

2. Metoda LZW

Nazwa metody pochodzi od inicjałów jej twórców: A. Lempela, J. Ziva i T. Welsha. Jest ona obecnie powszechnie uznawana w literaturze za najlepszą uniwersalną metodę kompresji danych. Jest również używana we wszystkich aktualnie używanych, komercyjnych programach pakujących[1]. Stosowane są najczęściej, efektywniejsze od pojedynczych implementacji, jej złożenia z innymi metodami - szczegóły nie są oczywiście publikowane, a nawet poszczególne rozwiązania bywają prawnie chronione.

Po raz pierwszy metoda LZW została wspomniana w maju 1977 roku w artykule [2] Lempela i Ziva: *"Uniwersalna metoda sekwencyjnej kompresji danych"*. Był to dopiero

[1] Chodzi o LHA Haruyasu Yoshizakiego, ARJ Roberta K. Junga i PKZIP Phila Katza

pewien szkic, ogólny pomysł bez żadnych szczegółów. Pomysł ten był następnie rozwijany w dalszych ich publikacjach. Konkretną implementację zaproponował w 1984 roku Terry A. Welsh.

Algorytm LZW ma wiele zalet, które uczyniły go najpopularniejszym algorytmem w swojej klasie. Jest dość prosty, ale na pewno nie banalny, jeżeli chodzi o zasadę działania, natomiast nieco trudniejszy do implementacji programowej[2].

Jest *uniwersalny*, to znaczy nie nakłada żadnych wstępnych warunków na dane, które będzie kompresować. Oczywiście jego efektywność, jak każdego kodowania odwracalnego, zależy od redundancji informacji w zbiorze wejściowym, jednak nie chodzi tu bynajmniej o tak banalną i narzucającą się redundancję, jak powtarzanie się wartości seriami lub niewystępowanie niektórych w ogóle.

Jest *jednoprzebiegowy*, czyli przegląda dane tylko raz, oraz jest typu *on-line*. Te dwie cechy (pierwsza wynika z drugiej) są niezbędne, jeżeli chodzi o szanse na sprzętową implementację, dla której algorytm był pierwotnie projektowany. Trudno sobie wyobrazić modem, który w celu przesłania wiadomości (a oczywiste jest, że wiadomość taka jest bardzo długa - dlatego przecież stosujemy kompresję) musi ją całą zgromadzić w swojej pamięci buforowej, następnie przeglądnąć kilka razy, produkując zresztą proporcjonalną do objętości wiadomości ilość informacji pomocniczej i dopiero wtedy wysyła ją zakodowaną na wyjście. Modem taki powinien być w stanie kodować wiadomość na bieżąco, niezależnie od jej całkowitej długości, używając przy tym ograniczonej ilości pamięci, czyli powinien posługiwać się właśnie algorytmem klasy *on-line*.

Jest *dynamiczny* i *adaptacyjny*, to znaczy sposób kompresji zależy od historii, czyli od informacji, jaka wystąpiła w przetworzonej już porcji zbioru wejściowego; algorytm *stara się* dostosować do charakteru danych występujących w danym zbiorze.

Ponadto należy go sklasyfikować jako algorytm **podstawieniowy o stałej[3] długości słowa kodowego**. Podstawieniowy, ponieważ elementy zbioru wejściowego zastępowane są swoimi kodami. Dokładniej należy on do grupy podstawieniowych tekstowych, czyli słownikowych, ponieważ jako elementy zbioru wejściowego rozumiemy podciągi znaków różnej długości, większej niż 1.

"O stałej długości słowa kodowego", ponieważ różnym podciągom wejściowym o różnych długościach przypisuje się na podstawie słownika kody mające tę samą długość.

W następnym punkcie zajmiemy się dokładnie zasadą działania algorytmu LZW.

---

[2] Większość pierwszych publikacji na temat metody, włączając T. A. Welsha, dotyczyła prób implementacji sprzętowej.

[3] choć nie zawsze będzie ona stała; istnieją wariacje używające zmiennej (zwiększającej się) długości kodu.

2.1. Zasada działania

Algorytm LZW, jak każda metoda podstawieniowa, działa w oparciu o słownik. Słownik ogólnie reprezentuje pewne przyporządkowanie. Jest to przyporządkowanie ciągom jedno- i (przede wszystkim) wieloznakowym unikalnych kodów. W każdym momencie kompresji słownik taki zawiera tylko takie ciągi, jakie zostały napotkane dotychczas w toku kompresji danego zbioru; na tym polega adaptowanie się metody do charakteru wejścia. Słownik na początku działania jest pusty, a dokładniej inicjalizuje się go tak, że zawiera 256 jednoznakowych ciągów o kodach 0..255. Następnie, w miarę przetwarzania zbioru wejściowego, słownik ten jest według ściśle określonych zasad uzupełniany, aż do rozmiaru wynikającego z długości słowa kodu wynikowego, na przykład dla kodu o długości 12 bitów mamy $2^{12}=4096$ pozycji w słowniku. Podstawowa wersja algorytmu nie przewiduje usuwania informacji ze słownika[4].

Jak powiedzieliśmy, algorytm LZW jest adaptacyjny, to znaczy podczas działania musi wytwarzać pewną informację pomocniczą, odzwierciedlającą proces adaptacji do konkretnego zbioru wejściowego - jest nią właśnie słownik. Oczywiste wydawać by się mogło, że jest on później niezbędny dla odkodowania zbioru i w związku z tym musi być do tego zbioru dołączony, co zresztą ma miejsce na przykład w metodzie Huffmana i jest jedną z jej poważniejszych wad[5].

W metodzie LZW nie dołączamy słownika do wyniku. Zawsze dużym zaskoczeniem dla zapoznających się z tą metodą jest to, że dekoder jedynie na podstawie czytanego przez siebie zbioru, bez dodatkowych danych, jest w stanie, wykonując takie same kroki jak koder, odbudowywać słownik, korzystając jednocześnie z niego przy dekompresji. Jest to możliwe dzięki ściśle określonym zasadom uzupełniania słownika, które zna i koder, i dekoder.

W pierwszym przybliżeniu pojedynczy krok działania algorytmu polega na:

- pobraniu z wejścia jednego podciągu;
- wysłaniu na wyjście jednego kodu;
- zarejestrowaniu jednej nowej pary ciąg-kod w słowniku (zwanym niżej *uaktualnieniem).*

---

[4] Pewne wariacje metody podważają tę zasadę. O przepełnieniu i reakcji na ten fakt będziemy jeszcze mówić.

[5] szczególnie dokuczliwych dla krótkich wiadomości, gdy tablica kodowania stanowi większą część zbioru wynikowego.

Pewna wspomniana niebanalność koncepcji polega na tym, że **para ciąg-kod tłumaczona** w danym kroku[6] jest **różna od pary** nowo rejestrowanej. Zresztą po pewnym zastanowieniu widać, że nie mogło być inaczej. Postarajmy się to wyjaśnić.

Ogólna zasada jest taka: Koder wykonuje kolejne kroki: pobiera ciąg, wysyła kod i uzupełnia słownik. Później, podczas dekompresji, dekoder wykonuje analogiczne kroki, to znaczy tłumaczy w stronę przeciwną (kod na ciąg) oraz wykonuje identyczne uzupełnienie słownika. Dzięki takiej odpowiedniości koder i dekoder dysponują w każdej chwili słownikiem o takiej samej zawartości. Wyjaśnijmy więc powyższą niebanalność. Gdyby koder w danym kroku wysyłał na wyjście kod, który w tym właśnie kroku jest rejestrowany (wprowadzany do słownika), dekoder w analogicznym momencie, mając jedynie ów kod do dyspozycji, nie miałby możliwości *zgadnąć*, jaki ciąg jest tym kodem oznaczony. Dlatego też koder w danym kroku rejestruje nową parę w słowniku, natomiast na wyjście wysyła kod, który znajdował się już w słowniku **przed** bieżącym krokiem. Wtedy dekoder w odpowiednim miejscu pobiera kod, znajduje jego odpowiednik (ciąg), bo z powyższego założenia na pewno może go znaleźć, a znając ten ciąg **może** wykonać to samo uzupełnienie słownika, co koder, czyli owa *równoległość* jest zachowana[7].

2.2. Koder

Pojedynczy krok kodera wygląda następująco: Z wejścia pobierane są kolejne znaki i dołączane do bieżącego ciągu. Ciąg ten jest wyszukiwany w słowniku. Jeżeli już się tam znajduje, dołączamy do niego kolejne znaki z wejścia aż do momentu, gdy tak utworzony ciąg nie zostanie znaleziony w słowniku. Ostatni znak, który utworzył taki nie napotkany jeszcze ciąg, nazywamy *ogonem* [ang. *terminating character*], natomiast całą resztę, od pierwszego do przedostatniego znaku - *głową*[8] [ang. *prefix string*]. Pamiętamy, że *głowa* znajdowała się już w słowniku. Wtedy na wyjście wysyłamy **znany** kod *głowy*, natomiast w słowniku rejestrujemy nową parę: cały bieżący ciąg i jego kod. Kolejno spotykanym ciągom przypisujemy kolejne kody (to znaczy wartości 256, 257...). Następnie *ogon*, który jeszcze nie uczestniczył w generacji kodów wyjściowych, staje się początkiem kolejnego bieżącego ciągu i do niego staramy się dołączać kolejne znaki, jak wyżej.

---

[6] Dla jasności: ciąg, który jest pobierany z wejścia, i kod, który w wyniku tego jest wysyłany na wyjście.

[7] Omówiona reguła jest warunkiem koniecznym dekodowalności; niżej omówimy pewien problem związany z ową równoległością.

[8] Nazewnictwo to, niezbędne przy opisywaniu algorytmu, wzorowałem na języku programowania *LISP*; będzie obowiązywało w dalszej części tekstu.

W zrozumieniu zasady może pomóc następujący szkic w pseudokodzie[9]:

```
inicjuj_slownik();
glowa = czytaj_znak();
while not koniec_zbioru_wej
   ogon = czytaj_znak();
   wyszukaj ( glowa+ogon );
   if znaleziono then
      glowa = glowa+ogon; // dołącz ogon
   else                   // do głowy
      pisz_kod ( kod (glowa) );
      rejestruj ( glowa+ogon );
      glowa = ogon;
endwhile
pisz_kod ( kod (glowa) );
```

gdzie:

`czytaj_znak` zwraca kolejny znak ze zbioru wejściowego

`pisz_kod` wysyła na wyjście podany kod

`rejestruj` (*ciąg*) rejestruje *ciąg* nadając mu kolejny kod

`kod` (*ciąg*) zwraca kod *ciągu* ze słownika

`wyszukaj` (*ciąg*) sprawdza obecność *ciągu* w słowniku

2.3. Dekoder

Zanim przedstawimy szczegółowo algorytm dekodera, musimy zasygnalizować kolejną komplikację (czy też niebanalny szczegół), jaka ma miejsce w logicznej konstrukcji dekodera. W jednym z wcześniejszych akapitów udowadnialiśmy, że para (ciąg,kod) tłumaczona w danym kroku jest różna od pary w tym kroku rejestrowanej. Dokładniej rejestracja odbywała się z pewnym wyprzedzeniem w stosunku do tłumaczenia. Taka logiczna *desynchronizacja* była warunkiem koniecznym dekodowalności. W naszej metodzie występuje jeszcze jedna tego rodzaju asymetria - między koderem a dekoderem.

Intuicyjnie wiadomo, że kroki wykonywane przez koder i dekoder są analogiczne, to znaczy, jeżeli koder w pewnym momencie pobiera pewien ciąg i wysyła kod, to dekoder w odpowiednim momencie wykona czynność przeciwną - pobierze ów kod i wyprodukuje ciąg. Natomiast jeżeli chodzi o uaktualnianie słownika, to dekoder jest **opóźniony** o jeden krok w stosunku do kodera. W danym kroku dekoder rejestruje parę, którą koder zarejestrował w kroku poprzednim. Jest to pewna komplikacja logiczna, lecz podobnie jak

[9] Jest to schemat uproszczony i zawiera wiele nieścisłości, ponieważ jego zadaniem jest jedynie przybliżyć zasadę działania.

poprzednio - nie ma innej możliwości. Zauważmy, że koder rejestruje nową parę przeczytawszy już z wejścia następny znak (*ogon*), który nie jest objęty bieżącym kodem (kod odpowiada tylko *głowie*)[10]. Natomiast dekoder po napotkaniu kodu nie ma możliwości *zgadnięcia*, jaki będzie pierwszy znak następnego zdekodowanego ciągu, aby móc wykonać identyczne uaktualnienie jak koder.

Zajmijmy się teraz konkretnie pojedynczym krokiem dekodera. Z wejścia pobierany jest jeden kod, który na podstawie słownika jest tłumaczony na odpowiedni ciąg. Ciąg ten jest wysyłany na wyjście. Poza wyprodukowaniem tego ciągu dekoder wprowadza do słownika ciąg złożony ze wszystkich znaków ciągu wyprowadzonego na wyjście w poprzednim kroku, rozszerzony o pierwszy znak aktualnego ciągu. W tym celu ciąg z poprzedniego kroku musi być przechowywany w zmiennej pomocniczej.

Oto ten schemat w pseudokodzie:

```
while not koniec_zbioru_zakod
   kod = czytaj_kod();
   ciag = dekoduj ( kod );
   pisz_ciag ( ciag );
   stary_ogon = ciag[0]; // pierwszy znak ciągu
   rejestruj ( stara_glowa+stary_ogon );
   stara_glowa = ciag;
endwhile
```

tutaj z kolei:

`czytaj_kod`	zwraca kolejny kod z wejścia
`pisz_ciag`	wysyła na wyjście podany ciąg
`rejestruj` (*ciąg*)	rejestruje *ciąg* nadając mu odpowiedni kod
`dekoduj` (*kod*)	zwraca ciąg oznaczony *kodem*
`stara_glowa`	i
`stary_ogon`	przyjmują takie same wartości jak *głowa* i *ogon* w programie kodera, tylko o jeden krok później.

Aby wyjaśnić pewne problemy, jakie mogą się pojawić podczas implementowania dekodera, prześledźmy następujący przykład. Należy zdekodować ciąg kodów **1,3,2,1,2,5,6,10,1,12**, który został otrzymany w wyniku zakodowania zbioru, składającego się ze znaków 'a','b' i 'c'.

[10] Pamiętamy, że nowy ciąg składa się i z *głowy*, i z *ogona*.

Oznaczenia:
kod wartość przeczytana w pierwszej instrukcji pętli;
stara_glowa wartość zmiennej sprzed modyfikacji w 6. instrukcji pętli;
pozostałe to wartości parametrów aktualnych wywołań procedur.

Tabela 1

Przykład kodowania

kod	stara_glowa	ciąg wysłany	rejestrowana para	
			ciąg	kod
1	???[12]	a		
3	a	c	ac	4
2	c	b	cb	5
1	b	a	ba	6
2	a	b	ab	7
5	b	cb	bc	8
6	cb	ba	cbb	9
10[11]	ba	bab	bab	10
1	bab	a	baba	11
12	a	aa	aa	12
EOF	aa			

Należy zwrócić uwagę, że pary ciąg-kod są rejestrowane o krok później niż w koderze, który wykonywał identyczne modyfikacje słownika podczas kodowania. Z tego opóźnienia wynika pewien problem, który należy rozwiązać. Otóż w kroku ósmym koder pobiera kod=10, usiłuje znaleźć go w słowniku, ale to się nie udaje, ponieważ jak widzimy w tabelce, ostatnim zarejestrowanym wtedy kodem jest 9. Jest to tak zwany *przypadek wyprzedzenia*[13]. Jaka jest tego przyczyna?

Gdybyśmy podejrzeli pracę kodera w analogicznym momencie, zauważylibyśmy, że koder właśnie zarejestrował parę ('bab',10), a w następnym kroku wysłał na wyjście "świeżo" przydzielony kod 10. Dekoder, jako że jest o jeden krok opóźniony, powinien

---

[11] Ten moment będzie omówiony niżej.

[12] W pierwszym kroku stara_glowa nie ma wartości i nie następuje rejestracja w słowniku, ponieważ dekoder jest opóźniony o jeden krok.

[13] W literaturze nie spotkałem żadnego terminu jednoznacznie tę sytuację określającego, wprowadzam więc własny.

w bieżącym kroku ów kod 10 jednocześnie przetłumaczyć i zarejestrować. Niestety, w danym kroku tłumaczenie poprzedza rejestrowanie, ponieważ do rejestracji musimy znać początkowy znak aktualnie przetłumaczonego ciągu. Można by pomyśleć, że rozwiązaniem byłoby tu kolejne opóźnienie czynności rejestracyjnych w dekoderze względem kodera. Jednak po pewnym zastanowieniu widzimy, że byłoby wtedy jeszcze gorzej: *przypadek wyprzedzenia* zdarzałby się również wtedy, gdyby koder użył kodu wygenerowanego nawet dwa kroki wcześniej.

Jest na to inne rozwiązanie. Można ustalić, że *przypadek wyprzedzenia* zdarza się tylko w jednym konkretnym przypadku. Oznaczmy przez α dowolny znak, a X - dowolny niepusty ciąg. Nasz przypadek będzie miał miejsce wówczas, gdy w zbiorze wejściowym wystąpi ciąg postaci '$\alpha X\alpha X\alpha$', a poprzednio wystąpił już 'αX'. Wtedy podczas tłumaczenia 'αX'[14] rejestruje się ciąg '$\alpha X\alpha$', a w kolejnym kroku jego kod jest już używany na wyjściu. Dzięki temu spostrzeżeniu możemy powiedzieć, że znaleźliśmy brakującą informację - wiemy, że nieznany dotąd pierwszy znak aktualnego ciągu jest taki sam, jak pierwszy znak poprzedniego ciągu. W związku z tym **dekoder, gdy napotka nieznany sobie jeszcze kod, przyjmuje, że oznaczony nim ciąg powstaje przez dołączenie do ciągu poprzednio wysłanego - powtórzonego jego pierwszego znaku i taki właśnie ciąg wysyła na wyjście i rejestruje w słowniku.**

Wracając do konkretnego przykładu, ciągiem poprzednim był 'ba', wobec tego kodowi 10 przypisujemy ciąg 'ba'+*pierwszy*('ba') = 'bab'.

Przedstawiłem zasadę działania algorytmu LZW na pewnym poziomie abstrakcji. Kolejny punkt wyjaśni kilka szczegółów implementacyjnych.

3. Implementacja LZW

W poprzednim punkcie omówiona została podstawowa wersja algorytmu LZW za pomocą pojęć ogólnych. Opis ten nie precyzuje jednak wszystkiego. Implementacja algorytmu nie jest banalna ani prosta. Algorytm ten ma wiele odmian, wariantów i parametrów, od których działanie metody i uzyskiwane wyniki wyraźnie zależą. Mogą one być podstawą do badania i strojenia tego algorytmu. Są to na przykład:

- reprezentacja pamięciowa słownika;
- długość słowa kodowego, która na pewno wpływa na efektywność kompresji i pojemność słownika;

[14] Tego z ciągu '$\underline{\alpha X}\alpha X\alpha$'

- zachowanie się algorytmu w momencie przepełnienia słownika.

Kwestie te będą poruszone w kolejnych podpunktach.

3.1. Reprezentacja słownika

Słownik, jako podstawowa, duża struktura danych, z jakiej korzysta algorytm, jest przez cały czas intensywnie używany i przetrzymywany w całości w pamięci. Dlatego należy zastanowić się zarówno nad jego reprezentacją pamięciową w celu zminimalizowania wymagań pamięciowych, jak i nad organizacją dostępu do danych w nim przechowywanych w celu minimalizacji złożoności czasowej.

W słowniku przechowywane są pary (ciąg,kod). O ile pole **kod** jest stałej długości (zwykle 9..15 bitów), tak pole **ciąg** z definicji może mieć długość dowolną - zależy ona tylko od właściwości statystycznych zbioru wejściowego, przy czym jest to zależność raczej niewyznaczalna teoretycznie. W związku z tą zmienną długością należałoby **dynamicznie** przydzielać obszary na te ciągi, co jest kłopotliwe i spowalnia do nich dostęp zarówno przez samą czynność przydzielania, jak i później, w związku z koniecznością odwołań z adresowaniem pośrednim. Taka naturalna i natychmiast narzucająca się reprezentacja pamięciowa (nazwijmy są słownikiem *kompletnym*) jest więc, jak każde pierwsze przybliżenie, nieoptymalna.

Istnieje inne rozwiązanie, wykorzystujące pewną własność słownika jako całości. Można spostrzec, że w naszej metodzie w **momencie, gdy dowolny ciąg jest rejestrowany w słowniku, jego *głowa* już się w tym słowniku znajduje.**
Wynika to ze sposobu, w jaki słownikiem posługuje się metoda LZW. Twierdzenie to można odnieść rekurencyjnie do głowy ciągu. W konkretnym przypadku, jeżeli rejestrujemy ciąg 'baba', to w słowniku znajduje się już 'bab', a także 'ba' i 'b'[15].

Wobec powyższego, jeżeli mamy przechować ciąg, którego głowa jest znanym ciągiem, to wystarczy podać kod jego głowy i jego ostatni znak, czyli *ogon*. Słownik będzie miał więc następujące pola: kod_głowy, ogon i kod; nazwiemy tę reprezentację słownikiem *przyrostowym*.

Porównajmy wykorzystanie pamięci w implementacji nie wykorzystującej wspomnianej własności (przechowywanie pełnych ciągów, dynamiczna alokacja) oraz dla słownika przyrostowego.

---

[15] Wszystkie jednoznakowe ciągi znajdują się w słowniku od początku.

Oto przykład przebiegu kodowania pewnego zbioru. Będzie to przykład uproszczony dla przejrzystości. Uproszczenie polega na tym, że alfabet wejściowy składa się tylko ze znaków 'a','b' i 'c'. Na początku zainicjowany słownik wygląda tak:

Tabela 2

Początkowa, przykładowa zawartość słownika

ciąg	kod
a	1
b	2
c	3

i zajmuje 3x1 + 3x2 = 9 bajtów. Ogólnie, gdy liczność alfabetu wejściowego wynosi n, początkowa objętość słownika wynosi $3n$. Dla kodu ASCII wyniesie 768 bajtów. Liczba ta jest identyczna dla obu przypadków reprezentacji pamięciowej słownika i nie ma większego znaczenia, ponieważ stanowi niewielki ułamek objętości słownika podczas działania algorytmu, a po wtóre - dane wypełniające słownik na początku są ściśle zdeterminowane i dzięki drobnej przeróbce algorytmu nie muszą być przechowywane w ogóle, więc równie dobrze możemy przyjąć, że objętość początkowa wynosi 0.

Pozycje 4 i wyższe będą przeznaczane na nowo rejestrowane ciągi.

Niech zawartością zbioru wejściowego będzie: **'acbabcbbababaaa'**.

Pomijając krokowe śledzenie działania algorytmu podsumujmy, że na wyjściu otrzymamy ciąg kodów: **1,3,2,1,2,5,6,10,1,12**, a słownik będzie miał zawartość w pierwszym wariancie:

Tabela 3

Przykładowa zawartość słownika

ciąg	kod	ciąg	kod
a	1	ab	7
b	2	bc	8
c	3	cbb	9
ac	4	bab	10
cb	5	baba	11
ba	6	aa	12

a w drugim:

Tabela 4

Przykładowa zawartość słownika

kod_glowy	ogon	kod
-	a	1
-	b	2
-	c	3
1	c	4
3	b	5
2	a	6
1	b	7
2	c	8
5	b	9
6	b	10
10	a	11
1	a	12

Co prawda zajętość pamięci wyniesie tu w pierwszym przypadku 49 bajtów, a w drugim aż 60, więc wynik pozornie nie potwierdzi naszych powyższych rozważań, ale jest to spowodowane jedynie tym, że przetworzono zaledwie 15 bajtów. Nie należy się spodziewać, że prawdziwe cechy tego algorytmu ujawnią się aż tak wcześnie jak po przetworzeniu mniej niż 0.1% przeciętnej ilości danych wejściowych[16]. Po przetworzeniu dalszych 10 bajtów: **'cbbcbbbaba'** i uzyskaniu na wyjściu kodów **9,9,11** w słowniku zarejestrowane zostaną kolejne cztery ciągi i objętości słownika wyniosą odpowiednio 73 i 80 bajtów. Widać tu, że pierwszy wariant już zaczyna "doganiać" drugi (w negatywnym sensie) - różnica się zmniejsza. Oczywiście trudno tu krokowo prześledzić i rozrysować zawartość słownika, gdy zawiera on kilka tysięcy ciągów, możemy jednak zysk pamięciowy oszacować. Nowodopisany ciąg zajmuje teraz tylko jeden wiersz, niezależnie od swojej długości. Jeden wiersz stanowi 5 bajtów (jednobajtowy ogon i po dwa bajty na kod_glowy i kod), podczas gdy ciągi mogą być dużo dłuższe (zresztą programiście zależy na długich ciągach). W pierwszym wariancie każdy ciąg zajmował tyle bajtów, z ilu znaków się składał + 2 na kod, czyli dla ciągów dłuższych niż 3 znaki drugi wariant już bardziej się opłaca. Teoretycznie nie sposób przewidzieć, jakie długości ciągów będą

[16] Jest to oszacowanie bardzo zgrubne - chodzi tylko o rząd wielkości

dominowały, więc należy posłużyć się eksperymentem. Z naszych badań wynika, że po zakodowaniu około 20 KB pliku tekstowego w słowniku zdarzały się ciągi nieco przekraczające 20 znaków, choć najczęściej nie przekraczały dziesięciu. W tej sytuacji wymagania pamięciowe wariantu drugiego są o około połowę mniejsze niż pierwszego. Są to liczby zgrubne, otrzymane dla jednego, konkretnego pliku tekstowego i choć mogą być zupełnie inne dla innych rodzajów plików, jednak proporcja będzie mniej więcej zachowana. Teoretycznie po 20 KB tekstu może zarejestrować się ciąg o długości 200, choć wymagałoby to specjalnie spreparowanego pliku. Z drugiej strony plik tekstowy daje w podobnych warunkach najdłuższe ciągi, w porównaniu z innymi typami plików.

Rozwiązanie drugie ma również inną, dużą zaletę. W słowniku występują już tylko pola o stałej długości, co pozwala bardzo szybko i wygodnie odwoływać się do nich poprzez indeksy lub wykorzystując funkcje mieszające.

Omawiane rozwiązanie ma również wady. Pierwsza, być może niewiele znacząca, polega na tym, że wpisanie nowego ciągu wymaga znajomości kodu jego głowy. Jednak w konkretnym przypadku naszego algorytmu nie jest to prawie wcale kłopotliwe, ponieważ sam algorytm wymaga, aby wprowadzenie ciągu było poprzedzone odnalezieniem jego głowy, a przez to też jej kodu. Wystarczy po prostu zapamiętywać ten kod w zmiennej pomocniczej.

Druga wada jest poważniejsza. Przy pierwotnej organizacji słownika w celu odnalezienia ciągu odpowiadającego danemu kodowi wystarczało jedno odwołanie (być może pośrednie). Teraz ciągi trzeba wyszukiwać "znak po znaku", czyli kosztuje to tyle odwołań, ile jest znaków w ciągu (a dokładnie o jedno mniej). Ponadto uzyskujemy te znaki od ostatniego do pierwszego, więc należy zastosować pewną strukturę podobną do stosu, która pozwoli przed wyprowadzeniem na wyjście odwrócić ich kolejność.

Ogólnie jednak powyższa modyfikacja jest korzystna i często stosowana.

3.2. Długość słowa kodowego

Zajmijmy się zanalizowaniem wpływu długości słowa kodowego na efektywność kompresji i inne parametry, charakteryzujące pracę algorytmu. Według definicji metody długość ta musi wynosić co najmniej 9 bitów. Formalnego górnego ograniczenia nie ma - praktycznie można przyjąć 15 bitów. Zmiana długości słowa kodowego wpływa na efektywność kompresji w sposób nieregularny.

Po pierwsze, oczywiste wydaje się, że większa długość kodu powoduje proporcjonalnie większą objętość zbioru wynikowego, ponieważ każdy kod, czyli element zbioru wyjściowego, zajmuje więcej bitów. Generalnie zawsze staramy się tak kodować, aby

zużyć jak najmniejszą porcję bitów na element i po prostu nie marnować bitów. Znaki 'a','b','c' z przykładów z poprzedniego punktu mogliśmy zakodować aż na ośmiu bitach, jednak minimalną ilością bitów dla rozróżnienia trzech elementów jest 2. Wniosek: **długość kodu powinna być jak najmniejsza.**

Z drugiej strony, im większy rozmiar słownika, tym więcej ciągów można w nim zarejestrować, a im więcej ciągów rejestrujemy, tym większa szansa, że dojdzie do rejestracji i użycia dłuższych ciągów[17], co z kolei oznacza bezpośrednio lepszy stopień kompresji. W tym kontekście zależałoby nam więc na jak **największym rozmiarze słownika.** Oznacza to jednak **zwiększanie** długości kodu, bo przecież istnieje zależność $rozmiar\_słownika = 2^{długość\_kodu}$.

Widzimy, że tendencje te są sprzeczne, co zresztą ma miejsce prawie w każdym problemie optymalizacji parametrycznej i zapewne optymalną długością okaże się pewna wartość pośrednia. Jedyną drogą jej ustalenia są eksperymenty.

Spróbujmy teoretycznie przewidzieć wyniki porównania wersji z krótszym i dłuższym kodem. Można się spodziewać, że dla krótkich zbiorów, jak również w początkowych fragmentach długich zbiorów, przewagę będzie miała wersja z krótszym kodem, ponieważ algorytm kodujący LZW, niezależnie od długości kodu, na początku wysyła tę samą ilość i te same wartości kodów, z tym że wspomniana wersja zużyje na nie mniej miejsca. Różnica pojawia się dopiero dla dłuższych plików, kiedy w wersji o krótszym kodzie następuje przepełnienie słownika i związane z tym problemy, które będą zresztą niżej omówione.

Natomiast duża efektywność drugiej wersji (o dłuższym kodzie) ujawni się dopiero na dalszym fragmencie długiego zbioru wejściowego, kiedy metoda *zdąży* już zarejestrować długie, dające silną kompresję, ciągi. Mniejszy słownik wersji o krótszym kodzie na tym etapie kompresji dawno uległby przepełnieniu.

Skoro zastosowanie krótszego kodu jest korzystne przy kompresji początkowego fragmentu zbioru, natomiast dłuższego - na dalszym etapie, można zdecydować się na rozwiązanie kombinowane[18], to znaczy wprowadzenie możliwości **zmian długości słowa kodowego** w trakcie kompresji danego zbioru. Będzie to polegało na tym, że na początku będziemy używać minimalnego 9-bitowego kodu wyjściowego, co daje słownik 512-pozycyjny (w tym 256 pozycji na nowe ciągi). Dopiero w momencie zarejestrowa-

---

[17] Bo długi ciąg zarejestruje się w słowniku tylko wtedy, gdy jego krótsi poprzednicy już tam będą.

[18] Będzie ono zresztą znaczną modyfikacją pierwotnej koncepcji Lempela i Ziva.

nia[19] ciągu numer 511 (dokładnie po zarejestrowaniu) można będzie zwiększyć długość kodu o 1. Od tego momentu na wyjście wysyłane będą kodu 10-bitowe, aż do zarejestrowania kodu 1023, który spowoduje wydłużenie kodu do 11. Wydłużenia możemy stosować do osiągnięcia wartości 14 lub 15 bitów.

Zastanówmy się, jak duże mogą być korzyści z zastosowania nowej koncepcji względem wersji o kodzie 9- i 14-bitowym. Jeżeli chodzi o wersję 9-bitową, to nie będzie żadnych różnic do momentu zarejestrowania kodu 511, kiedy to nasza metoda przejdzie dalej, podczas gdy słownik 9-bitowy po prostu przepełni się. Porównując z wariantem o stałej długości kodu równej 14 stwierdzamy, że wszystkie wysyłane na wyjście wartości kodów będą identyczne, przy czym nasza nowa koncepcja zawsze używa jedynie niezbędnej liczby bitów (na początek 9), podczas gdy ta druga zawsze czternastu. Niezależnie od liczby kodów, jakie zostaną wysłane, przed zarejestrowaniem kodu pięćsetjedenastego nasza metoda wygeneruje zbiór wyjściowy stanowiący 9:14=64% zbioru wygenerowanego przez wersję "14". Z drugiej strony, na wartościach bezwzględnych różnica ta nie będzie tak znacząca. W pesymistycznym przypadku przed zarejestrowaniem kodu 511 wyślemy tylko około 256 kodów, co zajmie 256*9=2304 bitów=288 bajtów, przy 256*14=3584 bitach=448 bajtach wersji "14", czyli 160 B różnicy. W przypadku średnim może ich być około dwa razy więcej, więc dla krótkich zbiorów różnica jest nieduża, lecz zauważalna. Między zarejestrowaniem kodu 511 a 1023 wysyłamy minimalnie kolejnych 512 kodów, czyli 512*10=640 B, do różnicy dodajemy kolejne 256 B. Następnie, w miarę wydłużania zmiennego kodu, proporcje procentowe (10:14, 11:14) rosną (czyli pogarszają się), natomiast wydłużają się okresy między kolejnymi zmianami długości kodu. Każdy okres powinien być dwa razy dłuższy od poprzedniego. Wreszcie, gdy długość kodu osiągnie 14, różnice między chwilowymi stopniami kompresji obu wersji znikają, pominąwszy oczywiście ilość kodu dotychczas wytworzonego.

Reasumując, wersja o zmiennej długości kodu łączy zalety krótkiego kodu w początkowym fragmencie z zaletami długiego kodu w dalszych fragmentach i musi mieć niewielką przewagę nad wersją "14", co jest najbardziej widoczne na początku.

3.3. Przepełnienie słownika

Należy wreszcie rozważyć osobny problem, który pojawiał się już w tekście wielokrotnie, mianowicie ustalenie zachowania algorytmu w momencie przepełnienia słownika Dotychczas pomijaliśmy ten problem, ograniczając się do stwierdzenia, że w każdym

[19] *Uwaga:* nie "użycia na wyjściu"

kroku uzupełniamy słownik, natomiast nigdy z niego nic nie usuwamy, ponieważ autorzy koncepcji, Lempel i Ziv, również tego nie rozważali.

W tej sytuacji możliwych jest parę rozwiązań. Po pierwsze, można przyjąć, że od momentu zapełnienia słownika kolejne operacje rejestrowania nowych ciągów będą ignorowane, słownik będzie zamrożony [ang. *freeze*]. Zaletą tego rozwiązania jest prostota, natomiast wadą jest fakt, że w momencie zamrożenia algorytm traci zdolności adaptacyjne. Wada ta nie musi być wcale dokuczliwa, ponieważ i tak najszybsza adaptacja ma miejsce na początku, w obszarze tak zwanej *rozbiegówki*, i jeżeli charakter danych nie zmieni się w dalszej części pliku, to taki zamrożony słownik może być bardzo efektywny. Natomiast, jeżeli charakter danych zmieni się, stopień kompresji pogorszy się wyraźnie z powodu używania nieadekwatnego słownika.

Można powiedzieć, że odwrotną charakterystykę ma rozwiązanie drugie, polegające na wyczyszczeniu, czyli ponownej inicjalizacji [ang. *reset*] słownika. Wtedy, po przepełnieniu słownika, przetwarzanie zaczyna się jakby od nowa, w szczególności ponownie wchodząc w okres *rozbiegówki*, ze wszystkimi tego następstwami: chwilowo spada stopień kompresji, natomiast algorytm ma możliwość zaadaptowania się do nowego rodzaju danych. Okaże się to skuteczniejsze dla plików, w których często zmienia się charakter danych, natomiast dla pliku jednorodnego ciągłe przechodzenie przez *rozbiegówki* będzie niepotrzebną stratą[20] i lepsza okaże się wersja *freeze*.

Ideałem byłoby rozwiązanie kombinowane, gdzie następowałoby wyczyszczenie *(reset)* w przypadku wykrycia nowego charakteru danych, a zamrożenie *(freeze)* w przypadku niewykrycia zmian, ale nie jest to realizowalne z powodu choćby niemożności sformalizowania, co oznacza intuicyjne określenie "zmiana charakteru danych".

Obydwa rozwiązania nie wymagają w ogóle danych pomocniczych i prawie wcale czasu, co powoduje, że są stosowane. Oczywiście najprawdopodobniej lepsze rezultaty można by uzyskać stosując bardziej zaawansowane techniki, znane choćby z koncepcji pamięci wirtualnej. Sądzę, że analogia daje się zauważyć. Tu mamy ograniczony słownik, do którego wprowadzamy wciąż nowe wiersze, aż do zapełnienia. Tam również mamy ograniczoną pamięć fizyczną oraz programy domagające się dostępu do wciąż nowych stron, co również powoduje przepełnienie pamięci fizycznej i konieczność wymiany stron. Jako przykładowe można tu podać koncepcje: LRU [ang. *least recently used*] - najdawniej używany i LFU [ang. *least frequently used*] - najrzadziej używany. Jednak uważam, że metody te nie mogą mieć praktycznego znaczenia u nas z następujących powodów.

Po pierwsze, wymagają pewnych struktur danych w pamięci, które przechowywałyby informacje o częstości lub czasie używania poszczególnych elementów (ciągów) słownika.

---

[20] Co zresztą potwierdziło się później w badaniach.

Po drugie, wymagają też czasu i specjalnych procedur obsługi.

Po trzecie, w związku ze specyfiką metody LZW nie mamy pełnej swobody w wyborze elementu, który miałby być usunięty. Nawet gdyby najrzadziej używanym ciągiem okazał się 'ba', nie możemy go usunąć, jeżeli, a jest to bardzo prawdopodobne, w słowniku znajdują się jakieś ciągi będące jego rozszerzeniem, to znaczy 'bab','baba', ponieważ słownik straciłby swoją podstawową własność, mówiącą o obecności głowy każdego ciągu.

3.4. Badania

Wiele z powyższych spostrzeżeń wynika z przeprowadzonych badań. Spośród licznych plików użytych podczas badań jako dane wejściowe do kompresji przedstawię sześć najciekawszych i istotnie różniących się od siebie. Są to:

Tabela 5

Pliki przykładowe

Symbol	długość	zawartość
F1	110510	**util.doc**: tekst ASCII
F2	66174	2 x **format.com**: program (DOS 5.0)
F3	95034	2 x arkusz (Lotus 1-2-3)
F4	117332	**cpp.exe**: program (pakiet BC++ 2.0)
F5	69589	**filedoc.txt** + **format.com**
F6	69589	**format.com** + **filedoc.txt**

F1: *util.doc* jest regularnym tekstem, opisem programów w instalacji BC++ 2.0

F2: dwukrotnie skopiowany program *format.com*, w celu zaobserwowania kompresji dla powtarzających się fragmentów

F3: dwukrotnie skopiowany arkusz kalkulacyjny *a.wk1*, w którym, nawiasem mówiąc, zebrana jest część wyników badań, przeprowadzonych w związku z tą pracą

F4: program Borland C++ preprocessor z niejednorodną zawartością

F5,F6 konkatenacje pliku tekstowego *filelist.doc* (oczywiście z BC++ 2.0) i programu *format.com* z DOSa; przekonamy się, że efekty kompresji zależą od kolejności.

Metoda najintensywniej adaptuje się w początkowym okresie i efektywność kompresji całego zbioru zależy od tego, czy dane, znajdujące się na początku, różnią się charakte-

rem od reszty pliku, czy stanowią jednolitą całość. Stopnie kompresji plików F5 i F6 metodą Huffmana musiałyby być identyczne, ponieważ adaptacja[21] tej metody opiera się na właściwościach statystycznych całego zbioru, a nie jakichkolwiek jego fragmentów.

Generalnie dla wszystkich plików daje się wyraźnie zauważyć okres *rozbiegówki*. Obejmuje on mniej więcej pierwsze 4 KB zbioru wejściowego. Charakteryzuje się bardzo szybkimi zmianami stopnia kompresji i nieregularnym zachowaniem wyników, w porównaniu z dalszym przebiegiem.

Zgodnie z naszymi oczekiwaniami metoda o mniejszej długości kodu (12) szybciej się adaptuje i w pierwszym okresie jest lepsza niż warianty 13 i 14. Natomiast przepełnienie i zamrożenie następuje dość wcześnie, a później stopień kompresji stabilizuje się (jak dla F1) lub wyraźnie spada (dla pozostałych). W literaturze wyraźnie wymienia się wartość 12 jako uniwersalną i zalecaną, podczas gdy ma ona przewagę nad wersją "14" tylko dla krótkich plików (do kilkunastu KB); jedynie dla przykładu F1 graniczną długością było prawie 40KB. Dla dłuższych plików okazuje się zupełnie nieskuteczna - dla pliku F4 powoduje wydłużenie pliku type *exe* o kilkanaście procent. Wariant o długości kodu 11 może być od razu zdyskwalifikowany, ponieważ regularnie daje wyniki wyraźnie gorsze (o 20 i więcej procent) od "12" i jest bardzo niestabilny. Powoduje wydłużenie pliku *exe* aż o 30%. Skracanie długość kodu nie ma więc sensu.

Wydłużenie długość kodu do 13 i 14 powoduje, że metoda wolniej i dłużej adaptuje się, natomiast dla plików długich cały czas utrzymuje się na wyrównanym poziomie kompresji, a nawet z czasem kompresja się polepsza. Przepełnienie następuje tu dużo później: po około 30 KB tekstu F1 i arkusza F3, a 15 KB innych plików dla długości 13, a odpowiednio po 70 i 30 KB dla długości 14.

Średni stopień kompresji dla odmiany ze zmienną długością kodu jest regularnie nieco niższy (lepszy) niż dla długości stałej 14. Jego przewaga nad wszystkimi innymi wariantami jest wyraźna dla plików do 10 KB. Dla plików dłuższych wyniki zbliżają się do wersji "14", zawsze pozostając lepszymi od innych. Jest to zupełnie zgodne z wcześniejszą analizą teoretyczną.

Ogólnie najlepsze stopnie kompresji uzyskaliśmy dla pliku tekstowego F1 i arkusza F3. Powszechnie wiadomo, że pliki tekstowe są łatwe do kompresji dla bardzo wielu metod. Dzieje się tak dlatego, że takie pliki są bardzo redundancyjne - pewne ciągi (wyrazy) powtarzają się często, a wiele znaków (nie-alfanumerycznych) nie występuje wcale. Interesujące są zaskakująco dobre wyniki kompresji arkusza kalkulacyjnego F3. Zwykle arkusze, jako tablice liczb rzeczywistych, są podobne do plików losowych i kompresują

[21] "adaptacja" należy rozumieć potocznie; Metoda Huffmana nie jest metodą adaptacyjną w sensie definicji.

się gorzej od kodów programów[22]. Ten arkusz jednak po pierwsze składa się z dwukrotnie powtórzonej porcji danych, co jest dużą redundancją, po drugie zawiera dużo komórek pustych - dwoma najczęstszymi bajtami są 0 (częstość 12.6%) i 255 (2.5%).

Najtrudniej z kolei kompresuje się pliki pseudolosowe. Wynika to zarówno z rozważań teoretycznych, jak i doświadczeń praktycznych. Uzyskiwany dla plików losowych stopień kompresji, niezależnie od wariantu, wahał się między 170%, a 180% (!). Nie uważam tego jednak za porażkę, bowiem wszystkie programy komercyjne, którymi starałem się skompresować takie pliki, również nie były w stanie go skrócić. Po rozpoznaniu charakteru pliku decydowały się one po prostu na skopiowanie pliku w stosunku 1:1, bez kompresji, nazywając tę metodę *storing*.

Interesujące wyniki otrzymałem dla pliku F4. Obserwujemy tutaj, poza słabymi rezultatami kompresji w ogóle i relatywnie dużymi różnicami między efektywnością poszczególnych wariantów, ciekawe zjawisko między 80 a 100 KB wejścia, występujące najwyraźniej dla wersji "14". Następuje tam wyraźne pogorszenie stopnia kompresji. Można to wyjaśnić, przeglądając zawartość tego pliku. Właśnie w tym obszarze wyraźnie zmienia się charakter danych, bowiem znajdują się tam komunikaty tekstowe programu, czyli jakby wstawka pliku tekstowego. W tym kontekście taka wstawka jest niekorzystna, ponieważ dotychczas algorytm nie miał wcześniej okazji zetknąć się z takimi danymi i do nich zaadaptować. Późniejsze gwałtowne polepszenie kompresji ma związek z tym, że powyżej 100 KB w pliku F4 znajdują się tabele - dane, które już występowały w obszarze między 10 a 20 KB.

Wykonałem w tej sytuacji dodatkowy eksperyment: zastąpiłem pierwsze 10 KB zbioru fragmentem z obszaru 90..100 KB tak, aby metoda mogła zaadaptować się do występujących tam tekstów. Dzięki temu uzyskaliśmy charakterystyczny dla plików tekstowych kształt *rozbiegówki*, a po dojściu do 80 KB, zgodnie z oczekiwaniami, zamiast pogorszenia nastąpiła wyraźna poprawa stopnia kompresji, ponieważ metoda tym razem była przygotowana do przetwarzania tekstów. Równie dobrze skompresował się fragment 100..120 KB, czyli tabele, których metoda również *nauczyła* się na początku (okres adaptacji wersji 14 trwał 32 KB, wersji 13 - 15 KB).

Plik F4 był jedynym, dla którego opłacalniejsza była wersja *reset*, ponieważ, jak widzimy, był niejednorodny.

Wykres dla pliku F5 w pierwszej części jest bardzo podobny do wykresu dla pliku F1, ponieważ są to pliki tekstowe. Wyraźne pogorszenie następuje, gdy przechodzimy do drugiej części - obszaru z kodami programu typu *com*. Zamrożenie słownika nastąpiło już po 32 KB. Wykres dla pliku F6 w pierwszej części jest identyczny z wykresem dla pliku

[22] Tak twierdzi na przykład Welsh.

F2, ponieważ są to te same dane. Potem w pliku F2 efektywność nieco wzrasta, natomiast w pliku F6 utrzymuje się na poziomie 0.8 do 0.9, gdy wchodzimy w obszar danych tekstowych. Obszar adaptacji[23] kończy się dopiero na 51 KB.

4. Podsumowanie

Metoda LZW jest jedną z najlepszych i ostatnio najpopularniejszą metodą kompresji bezstratnej. W literaturze jest szeroko uznawana za najlepszą w swej klasie uniwersalnych, jednoprzebiegowych algorytmów kompresji bezstratnej typu on-line. Osiąga lepsze stopnie kompresji od popularnej i będącej często odniesieniem w porównaniach metody Huffmana, ponieważ nie traktuje zbioru wejściowego jak ciągu nie skorelowanych ze sobą wartości przypadkowych, lecz adaptuje się do charakterystyki wejścia, to znaczy rejestruje ciągi znaków powtarzające się na wejściu i efektywnie koduje je (zastępuje) za pomocą dużo krótszych kodów.

Szkic implementacji został podany w tekście artykułu. Zaproponowano reprezentację pamięciową najistotniejszej struktury danych, z jakiej korzysta metoda - słownika - oraz pokazano wpływ takich parametrów, jak długość kodu na zachowanie się metody podczas kompresji.

Jak wiadomo, dla każdej metody kompresji, w związku z jej zasadą działania, istnieją dane, do których jest ona lepiej predysponowana, natomiast gorzej może radzić sobie z innymi danymi, które często sprzyjają użyciu innej metody. Jest to naturalne i niesprzeczne z definicją kompresji uniwersalnej, ponieważ wszystkimi metodami uniwersalnymi można skompresować dowolne dane, z tym że pewne metody osiągną nieco lepsze stopnie kompresji, a inne nieco gorsze. Można powiedzieć, że pewne metody *eksploatują* pewne parametry danych wejściowych (ciągi powtarzających się identycznych znaków, niezrównoważony histogram częstości występowania poszczególnych znaków itp.), a inne - inne. W związku z tym najprawdopodobniej *złożenia* znanych metod będą dawać stopnie kompresji wyższe niż poszczególne metody. Ten temat może stać się tematem dalszych badań.

[23] Dane dotyczą wersji "14" *freeze*.

LITERATURA

[1] Weiss J., Schremp D.: Putting Data on a Diet. IEEE Spectrum, USA, VIII 1993, vol. 30, iss. 8.

[2] Lempel A., Ziv J.: A Universal Algorithm for Sequencial Data Compression. IEEE Transactions. Information Theory, USA, V 1977.

[3] Welsh T.: A Technique for High Performance Data Compression. Computer 6/84.

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 8 marca 1995 r.

Abstract

The article presents the LZW compression method and its implementation. The first chapter describes how the computer science has been developing recently and justifies the necessity to use compression in the fields of storing and transmitting data. The LZW method is presented as an example of a lossless compression method. The main qualities of the method are enumerated and the principle of operation is explained with examples in pseudocode. The third chapter deals with some implementation details such as size and organization of the dictionary, which is built dynamically, accordingly to the incoming data and kept in the memory. The length of the code word may vary from 9 to 15 bits, which has a great influence upon the speed of filling the dictionary, the speed of adapting to the statistical characteristics of data source. The length can also be variable (increasing), which gives slightly better performance than the case of constant length. There are two ways to handle dictionary overflow - flushing or resetting. Others prove to be inapplicable.