ROZDZIAŁ 5. KODOWANIE SŁOWNIKOWE

Efektywność prezentowanych dotąd metod, opartych o tzw. modelowanie statystyczne, jest znacząca, przy czym jest ona tym większa, im precyzyjniej uda się określić wartości prawdopodobieństw wystąpienia poszczególnych symboli w konkretnym miejscu strumienia danych i im bardziej te prawdopodobieństwa są zróżnicowane. Koncepcja kodowania słownikowego uzupełnia ten schemat wstępnym etapem dekompozycji w celu większej dekorelacji danych i uproszczenia ich reprezentacji np. przed fazą entropijnego kodowania. Ponieważ pomysł kodera słownikowego nie zawiera praktycznie fazy binarnego kodowania - może być on uzupełniony kodowaniem Huffmana czy arytmetycznym.

Słownikowe algorytmy kodowania polegają na kolejnym czytaniu sekwencji danych wejściowych i przeglądaniu słownika w poszukiwaniu dokładnie takiej samej sekwencji danych. W przypadku zakończenia przeszukiwań sukcesem, indeks właściwej pozycji słownika staje się reprezentacją wyjściową, przy czym im dłuższa jest sekwencja i krótszy indeks, tym efektywniejszą jest kompresja.

W kodowaniu słownikowym można stosować statyczny model słownika budowany na wstępnym etapie analizy danych. Jest on skuteczny, gdyż zna z wyprzedzeniem pojawiające się ciągi danych. Jego praktyczną przydatność ograniczają jednak problemy związane z koniecznością dopisania do zbioru danych skompresowanych słownika użytego przy kodowaniu. Statyczne słowniki są stosowane w praktyce, ale jedynie do określonych przypadków kodowania w miarę stacjonarnych zbiorów danych o charakterystyce znanej *a priori*. Algorytmy przeznaczone do ogólnych zastosowań są w zdecydowanej większości adaptacyjne.

W kodach symboli, pojedynczy symbol jest zastępowany ciągiem bitów o zmiennej długości w zależności od estymacji ilości przenoszonej informacji. W przypadku kodowania słownikowana koncepcja jest odwrotna. Ciąg symboli o zmiennej długości jest zastępowany jedną sekwencją kodową - indeksem w tablicy słownika. Zbiór indeksów słownika tworzy strumień nowej reprezentacji zbioru oryginalnego, w którym poziom korelacji i zależności danych może być niekiedy znaczący. Wynika to głównie z czysto deterministycznego mechanizmu tworzenia słownika na podstawie zredukowanej ze względów praktycznych (wersja adaptacyjna) informacji, dostępnej w wariancie przyczynowym. Ograniczenia te zostały wyjaśnione w treści rozdziału.

5.1. Schemat metody

Ogólny algorytm kodowania słownikowego w wersji statycznej jest bardzo prosty. Pierwszy etap odpowiadający fazie modelowania polega na budowaniu słownika, którego poszczególne pozycje - tzw. frazy - wypełniają pojedyncze symbole alfabetu lub też sekwencje kolejnych symboli w oparciu o wstępną analizę zbioru danych kompresowanych. Następnie słownik ten należy zapisać lub przesłać do dekodera. Zasadnicza faza kodowania strumienia danych polega na znajdowaniu fraz w słowniku, które odpowiadają kolejnym sekwencjom danych wejściowych i zapamiętywaniu (lub przesyłaniu) indeksów tych fraz.

Częściej stosowana adaptacyjna postać ogólnego algorytmu słownikowego jest następująca:

Algorytm 5.1A. Metody słownikowe - adaptacyjne kodowanie

1. Podział strumienia wejściowego na fragmenty, które są testowane ze względu na obecność w słowniku.
2. Testowanie na okoliczność obecności w słowniku kolejnych fragmentów. Poszukiwany jest najdłuższy fragment z kodowanej właśnie części strumienia, który można odnaleźć w słowniku.
3. Kodowanie indeksów słownika lub przepisywanie danych bezpośrednio z wejścia na wyjście w przypadku braku odpowiedniej frazy w słowniku.
4. Modyfikacja słownika w sposób zależny od efektu przeszukiwań słownika. Dodanie nowej frazy na podstawie kodowanego łańcucha symboli.

Algorytm 5.1B. Metody słownikowe - adaptacyjne dekodowanie

1. Dekodowanie danych strumienia wejściowego jako indeksu słownika lub danych bezpośrednich. W przypadku indeksu odczytanie odpowiedniej frazy ze słownika.
2. Zapisanie (transmisja) dekodowanego strumienia danych.
3. Modyfikacja słownika w sposób zależny od odtwarzanych sekwencji danych, identycznie jak w koderze (Algorytm 5.1A, punkt 4).

Kluczowym zagadnieniem przy konstrukcji kodera słownikowego jest struktura słownika. Wpływa ona zasadniczo na skuteczność kodowania, jego czasochłonność, a więc potencjalny zakres zastosowań. Poniżej przedstawiono dwa najbardziej typowe sposoby budowania słownika: jako okno przesuwne (ograniczone strumieniem) oraz oddzielna nieograniczona struktura o dowolnej długości fraz (nie ograniczona strumieniem).

5.2. Algorytm ze słownikiem przesuwnym (LZ 77)

Pierwszym z opublikowanych przez Ziva i Lempela słownikowych metod kodowania jest algorytm, znany obecnie jako LZ 77 [1]. Słownikiem w tej metodzie jest zbiór danych poprzedzających bezpośrednio w strumieniu wejściowym kodowany symbol lub sekwencję symboli.

Główną strukturą danych jest przesuwne okno nałożone na strumień danych kodowanych. Okno to jest podzielone na dwie części:

a) pierwsza wyraźnie większa to właściwy słownik zawierający ostatnio analizowany, zakodowany już ciąg danych, bezpośrednio poprzedzający przeznaczoną do zakodowania sekwencję danych,

b) druga to bufor "patrz wprzód" (ang. look-ahead buffer), w którym znajdują się dane przeznaczone do zakodowania.

Metoda kompresji z przesuwnym oknem scharakteryzowana jest przez algorytm przedstawiony poniżej.

Algorytm 5.2. Metoda LZ 77 - kodowanie

1. Ustawienie struktury okna na początku czytanego strumienia danych wejściowych.
2. Poszukiwanie najdłuższego łańcucha danych w buforze "patrz wprzód" (zaczynającego się od danej znajdującej się na pierwszej pozycji w buforze), który ma swój dokładny odpowiednik w słowniku.

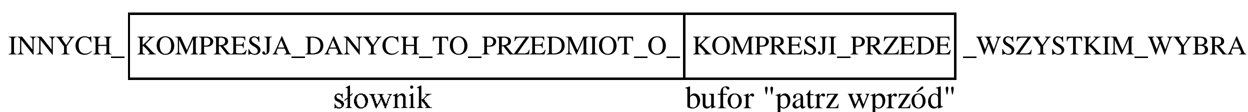
3. Tworzenie sekwencji kodowej tego łańcucha składającej się ze wskaźnika jego położenia w słowniku, długości oraz pierwszego symbolu w buforze występującego bezpośrednio po kodowanej frazie.
4. Przesunięcie okna wzdłuż wejściowego strumieniu danych o długość zakodowanej frazy.
5. Powtarzanie pp. 2, 3 i 4 aż do zakodowania ostatniego symbolu ze zbioru danych wejściowych.

Proces dekodowania w metodzie LZ77 jest, podobnie jak w innych metodach słownikowych, znacznie prostszy od kodowania. Polega na sukcesywnym odtwarzaniu danych poprzez: czytanie kolejnych sekwencji kodowych z wejścia, sięganie do odpowiedniego miejsca w słowniku, przepisywanie na wyjście określonej frazy słownika zakończonej symbolem odczytanym przez dekoder na końcu każdej sekwencji kodowej oraz przesunięcie okna o długość zrekonstruowanego fragmentu strumienia danych. Dekodowanie jest znacznie mniej czasochłonne, gdyż nie trzeba wykonywać szeregu porównań przy poszukiwaniu najdłuższej możliwej frazy w słowniku.

W przykładzie 5.1 pokazany został proces kodowania krótkiego, tekstowego zbioru danych.

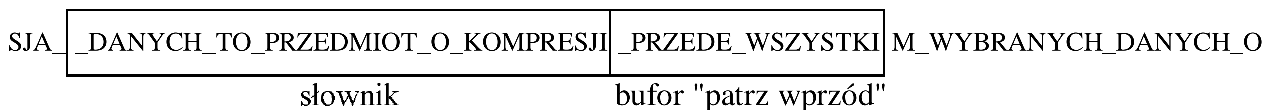
PRZYKŁAD 5.1. Proces kodowania strumienia danych metodą LZ77.

Wykorzystując metodę LZ 77 zakoduj fragment strumienia danych, po którym przesuwana jest struktura okna jak na rys. 5.1 wiedząc, że rozmiar bufora "pat rz wprzód" to 16 znaków, a rozmiar słownika - 32 znaki.



Rys. 5.1. Kompresja strumienia danych tekstowych metodą LZ 77; prezentacja strumienia danych przeznaczonych do kompresji w przykładzie 5.1, na który nasunięto strukturę okna: bufora i słownika wykorzystywaną w tej metodzie.

Najdłuższą frazą z bufora, którą można odnaleźć w słowniku jest KOMPRESJ (8 znaków), czyli przyjmując że pozycje w słowniku numerowane są od 1, pierwsza sekwencja kodowa wygląda następująco: **1, 8, 'I'**. Należy teraz przesunąć okno o 9 pozycji - rys. 5.2.



Rys. 5.2. Kompresja strumienia danych tekstowych metodą LZ 77 - przesunięcie okna o 9 pozycji wzdłuż strumienia danych z przykładu 5.1.

Drugą sekwencją kodową, dla najdłuższej teraz frazy _PRZED, jest **11, 6, 'E'**. Zakodowanie następnego znaku '_', po przesunięciu okna o siedem pozycji, będzie już niestety dużo mniej efektywne. Pomimo tego, iż symbol ten występuje w słowniku aż 5-ciokrotnie, w żadnym z tych przypadków następną literą nie jest 'W'. Stąd trzecia sekwencja kodowa jest następująca: **1, 1, 'W'**.

Skuteczność metody LZ 77 osłabia konieczność dopisania w słowie kodowym symbolu występującego w buforze bezpośrednio po kodowanej frazie. Związane jest to z problemem kodowania symbolu, który nie występuje w słowniku. Według przyjętego

schematu w takiej sytuacji sekwencja kodowa jest następująca: **0, 0, 'nieznany symbol'**. Następuje przy tym przesunięcie okna o jedną pozycję i proces kodowania jest dalej kontynuowany. Gdyby w skład sekwencji kodowej wchodziły jedynie wskaźnik położenia i długość łańcucha, wówczas w sytuacji symbolu nieznanego w słowniku proces kodowania zostałby przerwany, gdyż niemożliwym byłoby dalsze przesunięcie okna w kierunku nowych danych odczytanych ze strumienia (według algorytmu następuje bowiem przesunięcie o 0 pozycji).

Skuteczność kompresji danych metodami słownikowymi zależy zasadniczo od tego, jak długie frazy są wyszukiwane w słowniku. Wpływ na to w przypadku LZ77 ma zarówno wielkości słownika oraz bufora "patrz wprzód", jak też właściwości kompresowanego zbioru danych. Do podstawowych wad algorytmu LZ77 należy zaliczyć fakt, iż o ile jest on skuteczny w przypadku "bliskiego" powtórzenia się frazy, to w przypadku sekwencji danych powtarzających się z dużym odstępem, jego efektywność maleje. Jest to wynikiem ograniczonej wielkości słownika. Podobnie, kodowanie dłuższych fraz, nawet jeżeli takowe pojawiają się w strumieniu danych, możliwe jest jedynie w przypadku odpowiedniego rozmiaru bufora. Proste zwiększenie rozmiarów słownika i bufora nie daje spodziewanej poprawy efektywności. Pociąga to bowiem za sobą obok znacznego wzrostu czasu przeszukiwań i wymagań sprzętowych (większe ilości dostępnej pamięci operacyjnej), także spadek efektywności kodowania fraz krótkich. Następuje bowiem wydłużenie reprezentacji elementów składowych kodu pojedynczej frazy (indeksu położenia i długości łańcucha w słowniku), które muszą opisać większy rozmiar słownika oraz bufora.

Przykładowo, 4096-cio bajtowy rozmiar słownika (dla danych bajtowych np. w kodzie ASCII) i 32-wu pozycyjny (bajtowy) bufor "patrz wprzód" wymaga zapamiętania bitów dla pojedynczej frazy sekwencji kodowej składającej się z 12 bitów (pozycja w słowniku) plus 5 bitów (długość frazy), plus 8 bitów (pierwszy po frazie symbol), co w sumie daje 25 bitów. W przypadku, kiedy trzeba zakodować np. frazę jednoelementową (trzecia fraza '_' z przykładu 5.1), wówczas zamiast 16 bitów ze strumienia wejściowego (kody ASCII znaków '_' i 'W') otrzymujemy 25 bitów w strumieniu wyjściowym, co jest raczej mało efektywne. Jeszcze gorzej jest w przypadku, gdy kodowany aktualnie symbol nie znajduje się wcale w słowniku - wówczas musimy zapisać 25 bitów zamiast ośmiu.

Znane są różne sposoby poprawy efektywności tej metody. Przykładowo w technice LZSS dokonano dwu zasadniczych zmian próbując usprawnić algorytm LZ77 [2]. Pierwsza dotyczy sposobu zapamiętywania okna - słownika. Tworzone są dodatkowe struktury danych typu drzewo, które ułatwiają przeglądanie słownika i wyszukiwanie najdłuższych fraz (oszczędności czasowe). Druga zmiana związana jest z możliwością użycia dwu sposobów kodowania:

- dla dłuższych fraz: sekwencji kodowej przypisanej danej frazie ze słownika,
- dla fraz krótkich i pojedynczych symboli, które nie występują w słowniku: bezpośredniego przepisania sekwencji symboli na wyjście.

Takie rozwiązanie wymaga jednak dodania dodatkowego bitu poprzedzającego każdą wyjściową sekwencję kodową. Z drugiej strony, można w tym przypadku pominąć w sekwencji kodowej fraz dłuższych następujący po frazie symbol. Ostatecznie zredukowana sekwencja kodowa sprowadza się więc do bitu fraz krótkich/długich oraz indeksu położenia i długości kodowanej frazy o rozmiarach bitowych zależnych od wielkości słownika i bufora, odpowiednio.

Możliwa jest implementacja kodera LZSS z wykorzystaniem binarnego drzewa ułatwiającego przeszukiwanie słownika [3]. Naturalna w przypadku kodera Huffmana czy Shannona-Fano struktura drzewa binarnego musi być tutaj dopasowana do okna przesuwnego

w sposób bardziej złożony. Wykorzystuje się uporządkowanie kolejnych fraz o długości okna przesuwnego według przyjętych reguł (kolejność alfabetyczna, według kodu ASCII, itp.). Każdy rodzic jako dzieci ma węzeł wcześniejszy (w przyjętym porządku) i późniejszy (w tejże kolejności). Poszukując identycznego ciągu znaków do danych z bufora porównujemy je z węzłami drzewa słownika szukając najdłuższego ciągu jednakowych symboli. Taka drzewiasta struktura słownika ma pewne ograniczenia, pozwala jednak zrealizować technikę LZSS przy niewielkich nakładach obliczeniowych (porównywalnych z adaptacyjnym koderem Huffmana, znacznie mniejszych od realizacji kodera arytmetycznego wyższych rzędów).

5.3. Algorytm ze słownikiem nie ograniczonym strumieniem (LZ 78)

Odmienna koncepcja metody słownikowej, wykorzystująca zupełnie inny pomysł budowania słownika występuje w algorytmie LZ 78 [4]. Zamiast okna przesuwanego wzdłuż strumienia danych wejściowych, zastosowano tutaj słownik budowany jako zupełnie osobna, nieprzesuwna struktura. Jest to zbiór fraz utworzonych poprzez analizę sekwencji symboli występujących poprzednio w strumieniu wejściowym, a więc już zakodowanych danych (model przyczynowy). Dlatego też proces wiernej rekonstrukcji słownika może zostać zrealizowany w czasie słownikowej dekompresji.

Algorytm LZ78 można krótko scharakteryzować w trzech zasadniczych punktach:

- koncepcja słownika jako potencjalnie nieograniczonego zestawu fraz już kodowanych, uzupełnionych ewentualnie zbiorem fraz dodatkowych, dobranych na podstawie wiedzy *a priori*;
- kod łańcucha symboli składa się z pozycji identycznej frazy w słowniku oraz pojedynczego symbolu, który występuje bezpośrednio po tym łańcuchu;
- słownik modyfikowany jest na bieżąco; kolejno odnajdywane w trakcie kodowania frazy ze słownika, jako odpowiadające sekwencjom wejściowym, wchodzą w skład nowych fraz uzupełnione o symbol wchodzący w skład kodu łańcucha. Nowe frazy są umieszczane na kolejnych pozycjach w słowniku, który jest w ten sposób dynamicznie rozbudowywany.

Algorytm ten wygląda następująco:

Algorytm 5.3. Metoda LZ 78 - kodowanie

1. Inicjalizacja: na pierwszej pozycji słownika wpisywana jest fraza NULL, która oznacza pustą zawartość słownika w chwili początkowej, a także nieobecność w słowniku szukanej frazy w dalszym procesie kodowania.
2. Przeszukiwanie słownika na okoliczność obecności najdłuższych fraz tworzonych z kolejno czytanych symboli wejściowych.
3. Zapisanie indeksu najdłuższej frazy oraz występującego bezpośrednio po niej w strumieniu wejściowym symbolu jako sekwencji kodowej.
4. Dopisanie do słownika nowej frazy składającej się z odszukanej w punkcie 2 najdłuższej frazy oraz występującego bezpośrednio po niej w strumieniu wejściowym symbolu (z p. 3).

W trakcie dekodowania słownik jest tak samo adaptacyjnie rozbudowywany jak w koderze, na podstawie sekwencji zdekodowanych symboli. Odczytywane kolejno indeksy pozycji w słowniku pozwalają odszukać odpowiednie sekwencje symboli, tworzące

sukcesywnie wraz z symbolami czytanym bezpośrednio po indeksach dekodowany strumień danych.

Poniższy przykład obrazuje działanie algorytmu LZ78 w wersji podstawowej.

PRZYKŁAD 5.2. Proces kodowania strumienia danych metodą LZ78.

Należy zakodować następujący ciąg symboli: ”BOBAS_BOBEK_BOBCIO” metodą LZ 78.

Stosując metodę kodowania opisaną algorytmem 5.3 uzyskano ostateczną postać kodu wyjściowego jak w tabeli 5.1.

Tabela 5.1. Kodowanie ciągu symboli wejściowych z przykładu 5.1 metodą LZ 78.

Sekwencja wejściowa	Kod wyjściowy		Słownik	
	Indeks	Symbol	Indeks	Fraza
-	-	-	[0]	NULL
'B'	0	'B'	[1]	'B'
'O'	0	'O'	[2]	'O'
'BA'	1	'A'	[3]	'BA'
'S'	0	'S'	[4]	'S'
'_'	0	'_'	[5]	'_'
'BO'	1	'O'	[6]	'BO'
'BE'	1	'E'	[7]	'BE'
'K'	0	'K'	[8]	'K'
'_B'	5	'B'	[9]	'_B'
'OB'	2	'B'	[10]	'OB'
'C'	0	'C'	[11]	'C'
'I'	0	'I'	[12]	'I'
'O'	2	EOF		

Powyższy przykład pokazuje małą skuteczność kodowania tego algorytmu w pierwszej fazie. Słownik wtedy jest pusty i musi zostać dopiero zbudowany kosztem słabej skuteczności kompresji pierwszej partii danych w strumieniu wejściowym. Jest to właściwie cecha wszystkich algorytmów adaptacyjnych, różnią się one jednak szybkością dochodzenia przyjętego modelu do takiego stanu, kiedy cały algorytm zaczyna funkcjonować efektywnie realizując zasadniczą ideę kodowania danej metody. W tym przypadku widać, że słownik budowany jest dosyć wolno i po zakodowaniu 18 symboli wejściowych nie powstała ani jedna fraza trójelementowa w dynamicznie budowanym słowniku. Kod wyjściowy składa się z 13 indeksów i 12 symboli, co przy założeniu postaci słownika opisywanego 8-mio bitowymi indeksami daje nie najlepszy efekt 25 bajtów kodu opisujących 18 bajtów danych.

Nieustanne zwiększanie rozmiaru słownika daje coraz większą efektywność jedynie w przypadku kodowania wejściowego strumienia danych o nieskończonej długości. Dla krótszych zbiorów zwiększanie słownika powyżej pewnej optymalnej wielkości staje się nieefektywne ze względu na większą długość indeksów do słownika. Przykładowo, 16-to

bitowe indeksy pozwalają zapełnić 65536 elementowy słownik, jednak dla zbioru 50 kB tak duża pojemność słownika nie jest potrzebna, a krótszą postać reprezentacji wyjściowej zapewniłyby indeksy 14-to bitowe. W przeciwnym wypadku, kiedy kompresowane są bardzo duże zbiory, założona wielkość słownika prędzej czy później może się przepełnić. Trzeba wówczas zrezygnować z dodawania nowych fraz do słownika i posługiwać się dalej starym słownikiem. Jest to jednak dalece nieefektywne w przypadku, gdy zmienia się charakter danych w strumieniu wejściowym (zmienna statystyka niestacjonarnych źródeł informacji). Można oczywiście zaprojektować dynamiczną strukturę słownika i rozbudowywać go aż do wyczerpania dostępnych zasobów pamięciowych komputera, nie jest to jednak jedyne rozwiązanie. Często korzystniej jest ograniczyć od góry rozmiar słownika, nie tylko ze względów aplikacyjnych. Usunięcie najstarszej części słownika po wyczerpaniu jego możliwości pojemnościowych nie wpływa zasadniczo na długość znajdowanych fraz dla łańcuchów danych niestacjonarnego źródła. Przy zachowaniu tej samej długości bitowej indeksów ograniczonego słownika pozwala to zwiększyć efektywność kompresji zbioru danych.

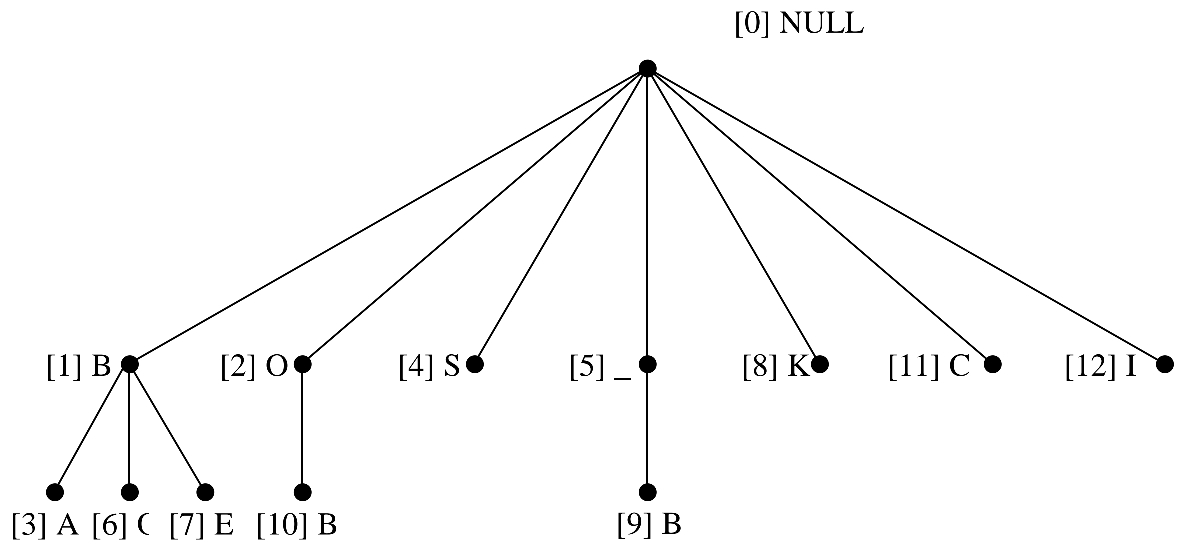
Do kontroli ograniczonej zawartości słownika wykorzystuje się często następujący mechanizm. Estymowany jest na bieżąco stopień kompresji i jeśli jego wartość zaczyna wyraźnie maleć, co jest znakiem nieefektywności aktualnej postaci słownika, wówczas likwiduje się sukcesywnie najstarszą część słownika (niewykorzystywaną od dłuższego czasu) wprowadzając na jej miejsce nowe frazy. Uzyskuje się w ten sposób mechanizm dynamicznej aktualizacji słownika, śledzący zmiany w statystyce kodowanego strumienia i zwiększający w ten sposób jego użyteczność.

Dodatkowym rozwiązaniem pozwalającym na jeszcze lepsze skonstruowanie słownika jest zmiana jego wielkości, dokonywana dynamicznie w trakcie kodowania kolejnych partii wejściowej sekwencji danych. Zaczynając np. od indeksów 9-cio bitowych słownika o 512 pozycjach, po jego zapełnieniu zwiększamy dwukrotnie jego rozmiar i rozbudowujemy go o nowe frazy, tym razem już z 10 -cio bitowymi indeksami. Proces ten jest kontynuowany aż do rozmiarów słownika korespondującego z wielkością pliku wejściowego i statystyką wystąpień różnych sekwencji symboli lub też górnego ograniczenia rozmiaru słownika. Zasadniczą zaletą takiej struktury słownika są krótkie indeksy dla niewielkich zbiorów danych, wydłużane dla zbiorów odpowiednio dużych. Ponieważ dekoder w takim rozwiązaniu naśladuje wiernie proces budowania słownika z kodera, dokładnie znany jest moment, kiedy przepełnia się aktualnych rozmiarów słownik i zaczynają się większe indeksy w wejściowym strumieniu dekodera.

Oddzielnym, bardzo istotnym problemem jest czas przeszukiwania słownika, który drastycznie rośnie dla dużych struktur słownikowych, ograniczając ich możliwości aplikacyjne. Istotną sprawą jest w tym przypadku sposób budowania słownika. Konstruuje się go najczęściej w postaci drzewa, indeksem zaś jest numer węzła w drzewie słownikowym. Drzewiasta struktura słownika z przykładu 5.2 została przedstawiona na rys. 5.3.

Słownik w postaci drzewa znacznie ułatwia proces przeszukiwań, będący centralną procedurą algorytmów słownikowego kodowania. Taka drzewiasta struktura wymaga jednak dużych zasobów pamięci do jej przechowywania. Statyczne struktury danych słownika z rys. 5.3 winny rezerwować dla każdego węzła, bez względu na poziom, 256 miejsc dla węzłów potomnych. Wykładniczo rośnie więc pojemność takiego drzewa na kolejnych poziomach oddalania się od korzenia, czyli węzła NULL. Koszt uzyskania długich fraz słownika z szybkim dostępem jest w tym przypadku bardzo drogi ‘pamięciowo’. Często stosuje się więc nieco wolniejsze struktury dynamiczne, znacznie bardziej oszczędnie gospodarujące pamięcią operacyjną. Wymagają one pamiętania wszystkich węzłów wewnętrznych takiego drzewa na

kolejnych poziomach wraz ze wskaźnikami na węzły potomne, alokacji pamięci na nowe węzły przy kolejnych nowych frazach dołączanych do słownika oraz poruszania się według wskaźników przy porównywaniu łańcuchów wejściowych z frazami słownika.



Rys. 5.3. Słownik o strukturze drzewa dla danych z przykładu 5.2, według tabeli 5.1, pozwalający zmniejszyć czas przeszukiwań.

W skład sekwencji kodowej algorytmu LZ78, tak jak w algorytmie LZ 77, wchodzi symbol następujący bezpośrednio po kodowanym łańcuchu danych, jako zabezpieczenie na wypadek konieczności tworzenia sekwencji kodowej nieznanych dla słownika symboli. Zostało to wyeliminowane w modyfikacji algorytmu LZ 78 znanej jako LZW [5]. Do usprawnień wprowadzonych w tej technice należy zapisywanie na początkowych pozycjach słownika jednoelementowych fraz, to znaczy wszystkich jednoelementowych symboli używanego alfabetu (np. wartości od 0 do 255 dla danych ośmiobitowych). Umożliwia to konstrukcję sekwencji kodowej, w skład której wchodzi jedynie indeks, czyli zawsze zapisywana jest pozycja ze słownika, nie ma zaś potrzeby dołączania pojedynczego symbolu. Jest to odpowiednik usprawnień z LZSS redukującej skład sekwencji kodowej. Praktyczny sposób tworzenia kodu metodą LZW przedstawia przykład 5.3.

PRZYKŁAD 5.3. Proces kodowania strumienia danych metodą LZW.

Należy zakodować ten sam ciąg symboli jak w przykładzie 5.2: ”BOBAS_BOBEK_BOBCIO”, ale tym razem metodą LZW.

Algorytm LZW (LZ78 plus opisane wyżej zmiany) pozwala efektywniej zakodować ten ciąg symboli. Obrazuje to tabela 5.2.

Uzyskany w tym przypadku kod wyjściowy zawiera 15 indeksów, co pozwala stwierdzić rzeczywistą (choć niewielką, zakładając 9-cio bitowy rozmiar indeksów) kompresję w stopniu $\frac{18 \cdot 8\,\text{bitów}}{15 \cdot 9\,\text{bitów}} = \frac{144\,\text{bitów}}{135\,\text{bitów}} \cong 1.07$, w przeciwieństwie do wyników z przykładu 5.2. Warto przy tym zwrócić uwagę na trzy frazy trójelementowe, które pojawiły się w słowniku. Świadczy to o lepszych potencjalnych możliwościach słownika w dalszym procesie kodowania.

Zastosowana w technice LZW adaptacyjna metoda budowania słownika tworzy nowe frazy z symboli, które pojawiły się już na wejściu, ale niekoniecznie zostały zakodowane w strumieniu wyjściowym. Chodzi tutaj o symbol, który znajduje się w tzw. pamięci, czyli został zapamiętany w koderze i oczekuje na zakodowanie. To przesunięcie o jeden informacji dostępnej w koderze i dekoderze może powodować w pewnych przypadkach trudności z wiernym odtworzeniem oryginalnej sekwencji danych. Może się tak zdarzyć, że trzeba zdekodować pozycję w słowniku, która jeszcze nie została zapełniona. Ponadto, sama procedura dekompresji nie jest tak prosta jak w przypadku algorytmu LZ78. Odczytywanie wejściowych indeksów i przesyłanie na wyjście odpowiednich fraz ze słownika, stale uzupełnianego, jest tutaj wspomagana przez dodatkowe rejestry i mechanizmy zabezpieczenia przed sytuacjami krytycznymi, a także odpowiednią kolejność operacji w słowniku i strumieniu wyjściowym.

Tabela 5.2. Kodowanie ciągu symboli wejściowych z przykładu 5.3 metodą LZW.

Sekwencja wejściowa	Kod wyjściowy	Pamięć	Słownik	
			Indeks	Fraza
-	-	-	[0]-[255]	Kolejne symbole - litery, cyfry i inne znaki w kodzie ASCII
'BO'	Ind('B')	'O'	[256]	'BO'
'B'	Ind('O')	'B'	[257]	'OB'
'A'	Ind('B')	'A'	[258]	'BA'
'S'	Ind('A')	'S'	[259]	'AS'
'_'	Ind('S')	'_'	[260]	'S_'
'B'	Ind('_')	'B'	[261]	[_B]
'OB'	256	'B'	[262]	'BOB'
'E'	Ind('B')	'E'	[263]	'BE'
'K'	Ind('E')	'K'	[264]	'EK'
'_'	Ind('K')	'_'	[265]	'K_'
'BO'	261	'O'	[266]	'_BO'
'BC'	257	'C'	[267]	'OBC'
'I'	Ind('C')	'I'	[268]	'CI'
'O'	Ind('I')	'O'	[269]	'IO'
-	Ind('O')	-	-	-

Aby zdekodować strumień wyjściowy kodera z przykładu 5.3 wygodnie jest wykorzystać dwa pomocnicze rejestry, które pozwalają na wygodne odtworzenie dynamicznego słownika z kodera. Pierwszy z nich, POPRZEDNI_INDEKS, zawiera odczytany w poprzednim kroku algorytmu dekodowania indeks wejściowy (opóźnione wejście). Drugi to PIERWSZY_SYMBOL, w który wpisywany jest pierwszy symbol zdekodowanego właśnie i wysyłanego na wyjście łańcucha. Nowe frazy w słowniku tworzone są poprzez połączenie zawartości rejestrów POPRZEDNI_INDEKS i PIERWSZY_SYMBOL. Proces dekodowanie według algorytmu LZW pokazuje tabela 5.3.

W algorytmie dekompresji widać wyraźne przesunięcie pomiędzy fazą czytania indeksów wejściowych i ich dekodowania, a tworzeniem słownika. Są takie momenty, kiedy symbol, np. 'O' po zdekodowaniu indeksu [256], jest już na wyjściu, ale nie ma go jeszcze w słowniku, w odpowiedniej frazie. W tym samym momencie w koderze, przy przesyłaniu na wyjście indeksu [256] już zapełniona jest pozycja [262] w słowniku frazą 'BOB' z symbolem 'O'. Dekoder natomiast ma jedynie pozycję [261] z frazą '_B' bez symbolu 'O'.

Tabela 5.3. Dekodowanie ciągu symboli z przykładu 5.3 metodą LZW. Ciąg ten jest następujący: Ind(B), Ind(O), Ind(B), Ind(A), Ind(S), Ind(_), 256, Ind(B), Ind(E), Ind(K), 261, 257, Ind(C), Ind(I), Ind(O).

Indeksy wejściowe	POPRZEDNI _INDEKS	Wyjściowy łańcuch symboli	PIERWSZY _SYMBOL	Słownik	
				Indeks	Fraza
-		-	-	[0]-[255]	Kolejne symbole alfabetu
Ind('B')	-	'B'	'B'	-	-
Ind('O')	Ind('B')	'O'	'O'	[256]	'BO'
Ind('B')	Ind('O')	'B'	'B'	[257]	'OB'
Ind('A')	Ind('B')	'A'	'A'	[258]	'BA'
Ind('S')	Ind('A')	'S'	'S'	[259]	'AS'
Ind('_')	Ind('S')	'_'	'_'	[260]	'S_'
256	Ind('_')	'BO'	'B'	[261]	[_B]
Ind('B')	256	'B'	'B'	[262]	'BOB'
Ind('E')	Ind('B')	'E'	'E'	[263]	'BE'
Ind('K')	Ind('E')	'K'	'K'	[264]	'EK'
261	Ind('K')	'_B'	'_'	[265]	'K_'
257	261	'OB'	'O'	[266]	'_BO'
Ind('C')	257	'C'	'C'	[267]	'OBC'
Ind('I')	Ind('C')	'I'	'I'	[268]	'CI'
Ind('O')	Ind('I')	'O'	'O'	[269]	'IO'

Krytyczna sytuacja dekodowania indeksu pozycji słownika, która nie została jeszcze wypełniona wynika właśnie z tego przesunięcia. Ma to miejsce w przypadku koincydencji następujących zdarzeń:

- w słowniku znajduje się już na pewnej pozycji pojedynczy symbol (znak) i sekwencja symboli (łańcuch) tworząc jedną frazę;
- w strumieniu wejściowym danych do kompresji pojawia się ciąg danych: znak, łańcuch, znak, łańcuch, znak.

W tym przypadku algorytm kompresji użyje jako kod indeks pozycji słownika, która nie została jeszcze określona w analogicznym momencie procesu dekompresji. Rozwiązanie tego problemu polega na zapisaniu brakującej pozycji słownika w ten sposób, że wraca się do ostatnio analizowanego kodu i w odpowiadającym mu łańcuchu znaków dopisuje się

pierwszy znak tego łańcucha na koniec tworząc nowy łańcuch znaków. Rozpatrzmy to na konkretnym przykładzie:

PRZYKŁAD 5.4. Proces kodowania strumienia danych metodą LZW: sytuacja krytyczna.

Dla przykładowego strumienia danych proces kodowania wygląda jak w tabeli 5.4. Należy prześledzić proces dekodowania tego strumienia.

Tabela 5.4. Kodowanie krytycznej sekwencji danych w metodzie LZW.

Sekwencja wejściowa	Kod wyjściowy	Pamięć	Słownik	
			Indeks	Fraza
....	....	'_'	....	....
'SPORT'	Ind('_SPOR')	'T'	[1000]	'_SPORT'
'_TO'	Ind('T_T')	'O'	[1001]	'T_TO'
...	...	...	...	...
...	...	'_'	...	...
'SPORT_'	1000	'_'	[2000]	'_SPORT_'
'SPORT_T'	2000	'T'	[2001]	'_SPORT_T'

Jeśli teraz prześledzimy proces dekompresji, to w momencie odczytu indeksu [1000] i zdekodowania frazy '_SPORT' tworzona jest pozycja [1999] słownika. Kolejny indeks [2000] pojawia się więc w chwili, kiedy ta pozycja w słowniku jeszcze nie istnieje. Proste odwołanie się w tym przypadku do aktualnej zawartości słownika nie pozwala zrekonstruować ciągu symboli oryginalnego strumienia danych. Pokazuje to tabela 5.5. Można jednak w tym przypadku modyfikując nieco algorytm ustalić zawartość pozycji [2000] słownika i wiernie odtworzyć zakodowaną sekwencję.

Tabela 5.5. Dekodowanie krytycznej sekwencji danych w metodzie LZW.

Indeksy wejściowe	POPRZEDNI _INDEKS	Wyjściowy łańcuch symboli	PIERWSZY_ SYMBOL	Słownik	
				Indeks	Fraza
...	...	...	...	...	...
Ind('_SPOR')	Ind('...')	'_SPOR'	'_'	[999]	'..._'
Ind('T_T')	Ind('_SPOR')	'T_T'	'T'	[1000]	'_SPORT'
...	...	...	...	...	...
...	...	...	...	...	...
1000	...	'_SPORT'	'_'	[1999]	'..._'
2000	1000	?	?	[2000]	?

Ponieważ opisana sytuacja jest jedyną możliwą, kiedy występuje odwołanie do nie zapisanej jeszcze pozycji słownika, zmodyfikowany algorytm dekodowania może wykonać w tym przypadku następujący zabieg zaradczy: powrócić do poprzednio odczytanego łańcucha

'_SPORT' i przepisać pierwszy symbol tego łańcucha na koniec tworząc frazę '_SPORT_'. Jest to brakująca pozycja słownika o indeksie [2000]. Można więc przewidzieć brakujący jeszcze element słownika na podstawie już posiadanych informacji przez dekoder. Taki sam efekt można uzyskać wówczas, jeśli ustalimy następującą kolejność dekodowania w sytuacjach krytycznych:

- zawartość rejestru POPRZEDNI_INDEKS wpisujemy jako początek nowej frazy w słowniku,
- ustalamy wartość rejestru PIERWSZY_SYMBOL (znany jest już początek wyjściowego łańcucha symboli, bo będzie nim tworzona fraza słownika),
- tworzymy pełną postać nowej frazy w słowniku,
- wysyłamy na wyjście zdekodowany ciąg symboli.

5.4 Praktyczne aplikacje metod słownikowych

Techniki kompresji słownikowej są obecnie najbardziej popularną formą bezstratnej kompresji danych. Główną zaletą metod słownikowych jest szybkość i stosunkowo duża efektywność kompresji przy dobrze dopracowanej konstrukcji słownika. W teoretycznych rozważaniach można dowieść, że przy nieskończenie dużym słowniku algorytmy te zbiegają do entropijnej granicy efektywności kodowania [6]. Ponadto podstawowy schemat jest bardzo podatny na wszelkie modyfikacje w wersjach adaptacyjnych. Można go także efektywnie łączyć ze wstępną predykcją oraz entropijnymi technikami kodowania indeksów słownika z dobranymi modelami statycznymi lub dynamicznymi. Powoduje to, że metody bezstratnej kompresji słownikowej od lat są najbardziej popularną techniką kodowania danych w różnego typu archiwizerach.

Jak wspomniano, zwiększenie skuteczności kompresji można uzyskać poprzez dodatkowe kodowanie strumienia indeksów słownika przy pomocy metod entropijnych (szczególnie wtedy, gdy słownik nie jest zbudowany optymalnie). W tym przypadku należy dopasować wejście schematu kodowania do rozmiaru indeksów, szczególnie w algorytmach LZW ze zmiennym rozmiarem słownika, a także poszczególnych elementów sekwencji kodowej (np. indeks i symbol w LZ78, położenie i długość w LZSS). Przy zwiększaniu rozmiarów indeksów słownika czy też charakteru elementów kodowanej sekwencji następuje odpowiednie dostosowanie struktur danych oraz przełączenie modelu opisującego statystyczne własności strumienia danych. Przykładem połączenia techniki LZSS z adaptacyjnym koderem Huffmana jest archiwizer LHarc.

Algorytm LZ77 stał się podstawą wielu bardzo dobrych i powszechnie stosowanych programów kompresujących, jak chociażby Unix-owego gzip oraz PKZIP, a także LHarc czy ARJ [7], a z nowszych RAR. W większości przypadków indeksy kodowane są metodami opartymi o schemat Huffmana. Ponieważ programy te wykorzystują algorytm LZ77, są silnie niesymetryczne, czyli faza kompresji jest dużo dłuższa od dekompresji ze względu na konieczność wielokrotnego przeszukiwania słownika w czasie kompresji. Stosunek czasów kompresji i dekompresji wynosi średnio około 6:1, z wyjątkiem RAR (10:1). Jednocześnie, w większości przypadków RAR pozwala uzyskać nieznacznie wyższe stopnie kompresji od pozostałych programów (mniejszą o koło 2% średnią bitową).

Nowszą techniką kompresji wykorzystującą algorytm LZ77 jest standard PNG (ang. Portable Network Graphics) [8],[9],[10], przeznaczony do bezstratnej kompresji obrazów cyfrowych. Wykorzystane w niej zostały również prosta predykcja i koder Huffmana. Ze

względu na przeznaczenie do kompresji obrazów transmitowanych w sieciach zastosowano tutaj progresywny sposób kodowania obrazu: wpierw kodowana jest niskorozdzielcza wersja obrazu a następnie w kolejnych porcjach kodowane są coraz bardziej szczegółowe informacje obrazowe. Algorytm ten ma zastąpić stosowany dotąd powszechnie w internecie format GIF (ang. Graphics Interchange Format) [11].

Historia rozwoju metody ze słownikiem osobnym była już wspominana: opublikowany w 1978 przez Ziva i Lempela algorytm doczekał się większego zainteresowania dopiero w 1984 roku (po publikacji Terry Welcha). Opisana przez niego technika została następnie zaimplementowana w programie Unix Compress i znana jest obecnie jako kompresja LZW. Powstał także popularny odpowiednik programu unix-owego (oparty na LZW) dla środowiska MS-DOS pod nazwą ARC. Ponadto firma *Compuserve Information Service* w opracowanym przez siebie i powszechnie znanym formacie plików GIF zastosowała słownikową technikę kodowania opartą na LZW. Ponieważ w algorytmie LZW przy kompresji słownik jest budowany adaptacyjnie i proces ten musi być dokładnie powtórzony przy dekompresji - techniki te są w przybliżeniu symetryczne, czyli złożoność obliczeniowa kodera i dekodera są zbliżone.

Eksperymenty przeprowadzone z różnymi wersjami koderów słownikowych pozwalają porównać efektywność przedstawionych modyfikacji algorytmów budowania słownika i konstrukcji kodu wyjściowego. Przedstawione w tabeli 5.6 wyniki dotyczą prostych, nie komercyjnych realizacji koderów według tych samych koncepcji programowania (taki sam sposób budowania struktur danych, analogiczne warunkowania w pętlach, itd.). Można więc na ich podstawie wnioskować o skuteczności różnych rozwiązań. I tak efektywność metody LZSS jest wyraźnie większa od kodera LZW o stałej wielkości słownika (pozycja w słowniku określana jest przez 12-to bitowy indeks).

Tabela 5.6. Porównanie efektywności kompresji koderów słownikowych, w tym: LZSS, LZW - jako praktyczne realizacje dwu podstawowych technik słownikowych, LZW_ZS (ze zmiennym słownikiem), LZSS+HUF oraz LZSS+ARI (połączenie metod słownikowych z entropijnymi). Zbiory Z1-Z4 to zbiory tekstowe, a Z5-Z6 to obrazy w skali szarości (bajtowe). Tabela zawiera wartości średnich bitowych skompresowanych zbiorów.

Zbiór	Z1	Z2	Z3	Z4	Z5	Z6	Średnio
Długość(kB)	10	100	1000	4000	256	256	
LZSS	4,66	7.70	5.49	2.44	8.20	7.88	6.06
LZW	4.69	10.29	8.97	2.64	9.18	9.00	7.46
LZW_ZS	4.44	9.55	4.40	1.52	8.22	7.23	5.89
LZSS+HUF	3.43	6.76	4.18	1.18	7.29	6.96	4.97
LZSS+ARI	3.41	6.74	4.16	1.18	7.25	6.93	4.94

Modyfikacja schematu LZW, oznaczona przez LZW-ZS, polegająca na wprowadzeniu zmiennej wielkości słownika pozwala uzyskać jednak znacznie większą efektywność kompresji. Słownik zapełniony w chwili początkowej 256-cioma elementami alfabetu (bajtowa struktura danych) posiada jeszcze drugie tyle wolnych miejsc na dynamiczną rozbudowę. Indeksy w chwili początkowej są więc 9-cio bitowe. Po zapełnieniu tego słownika emitowane są indeksy 10-bitowe, potem 11-bitowe, 12-bitowe, itd. Kolejną poprawę efektywności kompresji uzyskano poprzez kodowanie fraz wyjściowych kodera LZSS

metodami entropijnymi: Huffmana i arytmetycznym. Pozwoliło to zmniejszyć średnią bitową reprezentacji wyjściowej algorytmu LZSS o ponad 1 bit/symbol.

Dodatkowo, przytoczono wyniki porównania skuteczności kompresji dwóch znanych archiwizerów Unix-owych: gzip (oparty na idei LZ77) i compress (oparty na idei LZ78) na szerokiej gamie różnorodnych danych testowych. Wyniki porównań zaprezentowano w tabeli 5.7.

Tabela 5.7. Porównanie efektywności kompresji dwóch archiwizerów: gzip i compress. Wyniki zostały zaczerpnięte z [12]. Zbiory Z1-Z3 to zbiory tekstowe, Z4-Z6 to losowo, niezależnie generowane zbiory zer i jedynek, a Z7-Z8 zawierają dane z eksperymentów biologicznych. Tabela zawiera wartości średnich bitowych skompresowanych zbiorów.

Zbiór	Z1	Z2	Z3	Z4	Z5	Z6	Z7	Z8	Średnio
Długość(kB)	4530	3953	2415	2048	2048	2048	3096	2877	
Gzip	2,37	2,41	2,40	1,25	0,78	0,36	2,52	2,35	1,81
Compress	2,22	2,84	3,27	1,03	0,57	0,25	2,42	2,26	1,86

Na podstawie wyników z tabeli 5.7 trudno stwierdzić, która z obu testowanych technik kompresji jest bardziej skuteczna. Potwierdza to zbliżone możliwości kompresji danych obydwu koncepcji Lempela i Ziva: LZ77 i LZ78, zarówno co do uzyskiwanych stopni kompresji, jak też stopnia złożoności i czasochłonności budowanych na ich podstawie koderów.

Bibliografia:

1. J. Ziv, and A. Lempel, *A Universal Algorithm for Sequencial Data Compression*, IEEE Trans. Information Theory, 23(3):337-343, 1977.
2. J. A. Storer, T. G. Syzmanski, Data Compression via Textual Substitution, Journal of the ACM, 29: 928-951, 1982.
3. T. C. Bell, *Better OPM/L Text Compression,* IEEE Trans. COM-34: 1176-1182, 1986.
4. J. Ziv J., and A. Lempel, *Compression of Individual Sequences via Variable-Rate Coding*, IEEE Trans. Information Theory, 24(5):530-536, 1978.
5. T. Welch, *A Technique for High-Performance Data Compression*, IEEE Computer, 17(6):8-19, 1984.
6. A. D. Wyner, J. Ziv, *The Sliding-Window Lempel-Ziv Algorithm is Asymptotically Optimal*, Proc. IEEE, 82(6):872-877, 1994.
7. ARJ technical informations, 1993, pod adresem: http://www.dogma.net/DataCompression/ArchiveFormats/arj.txt.
8. PNG Specification version 1.0, W3C Recommendation, 1996, pod adresem http://www.w3.org/Graphics/PNG/.
9. PNG home page: http://www.cdrom.com/pub/png/.
10. W .Skarbek - redakcja, *Multimedia - Algorytmy i Standardy Kompresji,* Akademicka Oficyna Wydawnicza PLJ, Warszawa, rozdz.3, 1998.
11. Graphics Interchange Format, version 89a, CompuServe Incorporated, 1990, pod adresem: http://www.w3.org/Graphics/GIF/spec-gif89a.txt.
12. Y.Matias, N.Rajpoot, S.C.Sahinalp, *The Effect of Flexible Parsing for Dynamic Dictionary Based Data Compression*, Proc. Data Compression Conference'99, pp. ,1999.