

Sprawozdanie - implementacja algorytmu LZW

Wojciech Decker

21 stycznia 2017

Spis treści

1	Cel projektu	1
2	Algorytm	1
2.1	Kodowanie	2
2.2	Dekodowanie	2
3	Implementacja	3
4	Prezentacja rezultatów	3
5	Wnioski	4
6	Załączniki	4
A	Algorytm kompresji	4
B	Algorytm dekompresji	6

1 Cel projektu

Celem projektu była implementacja i demonstracja zasady działania kompresji LZW

2 Algorytm

LZW - Lempel-Ziv-Welch został zaprezentowany w 1984r. przez panów Abraham'a Lempel, Jacob'a Ziv i Terry'ego Welch. Jest słownikowym algorytmem kompresji danych, względnie prostym w implementacji. Pozwala na uzyskanie zadowalających wyników przy niskim koszcie obliczeń. Na co dzień jest z powodzeniem wykorzystywany w systemach UNIX, oraz formacie GIF.

2.1 Kodowanie

Podczas kodowania wczytywane porcje danych, w mojej implementacji o długości 8 bitów są tłumaczone na wartości kodów słownika. Inicjalizacja polega na stworzeniu słownika, zawierającego wszystkie podstawowe słowa i przypisane im kody od 1 w górę. Następnie wczytywane kolejno słowa są tłumaczone na kody i sklejane. Jeżeli sklejone kody nie znajdują się w słowniku, są do niego dodawane.

Dokładny przebieg algorytmu wygląda następująco:

1. Zainicjalizuj słownik słowami o długości 1, przypisując im wartości kodowe.
2. Znajdź najdłuższe słowo W znajdujące się na wejściu, które znajduje się w słowniku.
3. Wypisz na wyjście kod przypisany do słowa W
4. Usuń słowo W z wejścia
5. Dodaj słowo W z dodaną kolejną wartością znajdującą się na wejściu przypisując jej kolejną wartość kodu
6. Idź do pkt. 2

W rezultacie otrzymujemy ciąg wartości kodowych, które zostają wypisane na wyjściu. Duży wpływ na skuteczność kompresji ma sposób zapisu kodów. Kody powinny być zapisywane na tylu bitach, ile wymaga najwyższa wartość kodu w słowniku. Pozwala to sukcesywnie zwiększać liczbę bitów wymaganych do zapisu kodu, zmniejszając rozmiar pliku wynikowego.

2.2 Dekodowanie

Proces dekodowania - dekompresji pliku do postaci oryginalnej przebiega analogicznie do kodowania. Tworzony jest identyczny słownik, z tą różnicą, że to wartościom kodów przypisywane są jednoelementowe wartości słów. Wczytywane są kolejne kody, tłumaczone na ciągi słów, nieznane wcześniej słowa są dodawane do słownika z kolejnymi wartościami kodów.

Dokładny przebieg algorytmu dekodującego:

1. Zainicjalizuj słownik słowami o długości 1, przypisując im wartości kodowe.
2. Wczytaj nowy kod pk i przetłumacz na słowo W
3. Wypisz na wyjście słowo W
4. Wczytaj kod k z wejścia
5. Przypisz do pc słowo skojarzone z kodem pk

6. Jeśli kod k jest zdefiniowany w słowniku, dodaj do słownika ciąg pc z dopisanym pierwszym znakiem słowa k . Wypisz na wyjście słowo przypisane do k .
7. W przeciwnym wypadku dodaj do słownika ciąg pc z dopisanym pierwszym znakiem słowa pc i wypisz je na wyjście
8. Do pk przypisz wartość k
9. Idź do pkt. 4

Efektem działania algorytmu dekodującego będzie plik identyczny z oryginalnym.

3 Implementacja

Implementacja algorytmu LZW została wykonana w języku Sala działającym na platformie Java.

Najważniejsze klasy:

LZW	implementacja wykonania algorytmów na wejściu i wyjściu
MutableCompression	algorytm kompresji
MutableDecompression	algorytm dekompresji
DummyIntWriter	zapisuje kodu o długości 32 bitów w 2, 3 lub 4 segmentach po 8 bitów
DummyIntReader	odczytuje kody o zapisane w segmentach przez <i>DIW</i> i rzutuje na kod 32 bitowy

4 Prezentacja rezultatów

Kompresję przeprowadzono na plikach tekstowych i graficznych. Implementacja generuje kody 32 bitowe, zapisując je w segmentach 8-bitowych, pomijając zapis zbędnych segmentów zerowych. Algorytm okazał się najwydajniejszy dla dużych plików tekstowych. Dla krótkiego tekstu *algorithm.txt* zawierającego fragment opisu algorytmu z anglojęzycznej wikipedii współczynnik kompresji wyniósł 82%. Tekst dłuższy - *Pan Tadeusz* - Adama Mickiewicza po kompresji miał 53% rozmiaru oryginalnego. Kompresja Biblii, będącej najdłuższym tekstem dała współczynnik kompresji 43%.

Implementacja została przetestowana pod kątem kompresji obrazów graficznych różnych rozmiarów. Kompresja grafiki dawała współczynnik kompresji na stałym poziomie ok. 154 %.

Nazwa pliku	rozmiar oryginalny	rozmiar po kompresji	współczynnik kompresji
algorithm.txt	4043	3320	.82117
lena16.png	206871	322398	1.55844
lena24.png	131864	198330	1.50404
lena48.png	61866	91776	1.48346
lena64.png	42263	65394	1.54731
lena8.png	389740	608799	1.56206
lena.png	511937	796173	1.55521
pan-tadeusz.txt	493390	263472	.53400
pismo_swiete.txt	5389056	2340807	.43436
short.txt	7	12	1.71428

Tabela 1: Tabela wyników kompresji

5 Wnioski

Kompresja daje największy zysk dla długich plików tekstowych. Kompresja obrazów przynosi skutek odwrotny do zamierzonego. Przyczyną małej wydajności kompresji może być sposób alokacji bitów do zapisu kodu, operujący na blokach 8-bitowych zamiast na pojedynczych bitach.

6 Załączniki

A Algorytm kompresji

Listing 1: Algorytm kompresji

```
package pl.edu.pw.ee.decker.lzw.internal

import com.typesafe.scalalogging.Logger
import pl.edu.pw.ee.decker.lzw.LZW

import scala.collection.mutable

/**
 * Created by clutroth on 31.12.16.
 */
class MutableCompression(val dictionary: mutable.TreeMap[
    List[Byte], Int]) {
    var lastString: List[Byte] = List()
    // Log.init(dictionary)

    object Log {
        val log = Logger[MutableCompression]
```

```

def str(list: List[Byte]): String =
  new String(new String(list toArray))

def init(dictionary: mutable.TreeMap[List[Byte], Int]) =
  log debug (dictionary toString())

def logDic(msg: String)
  (lastString: List[Byte], byte: Byte,
   dictionary: mutable.TreeMap[List[Byte], Int]) =
  log debug s"$msg ${str(lastString)}(${byte toChar})"
  as code ${dictionary(lastString :+ byte)}"

def newWord(lastString: List[Byte], byte: Byte,
  dictionary: mutable.TreeMap[List[Byte], Int]) = {
  logDic("new word")(lastString, byte, dictionary)
  log debug s"returned $returned"
}

def knownWord(lastString: List[Byte], byte: Byte,
  dictionary: mutable.TreeMap[List[Byte], Int]) =
  logDic("known word")(lastString, byte, dictionary)
}

def put(b: Byte): Option[Int] = {
  val currentString = lastString :+ b
  if (dictionary contains currentString) {
//    Log.knownWord(lastString, b, dictionary)
    lastString = currentString
    None
  } else {
    val lastCode = dictionary(lastString)
    dictionary += currentString -> ((dictionary size) +
      1)
//    Log.newWord(lastString, b, dictionary)(lastCode)
    lastString = List(b)
    Some(lastCode)
  }
}

def buffer: Int =
  if (lastString isEmpty)
    throw new IllegalStateException("Buffer is empty")

```

```

    else
        dictionary(lastString)
}

```

B Algorytm dekompresji

Listing 2: Algorytm kompresji

```

package pl.edu.pw.ee.decker.lzw.internal

import com.typesafe.scalalogging.Logger

import scala.collection.mutable

/**
 * Created by clutroth on 03.01.17.
 */
class MutableDecompression(val dictionary: mutable.
    TreeMap[Int, List[Byte]]) {
    var previousCode: Option[Int] = None

    //Log.init(dictionary)

    object Log {
        val log = Logger[MutableDecompression]

        def str(list: List[Byte]): String =
            new String(list.toArray)

        def init(dictionary: mutable.TreeMap[Int, List[Byte]]): Unit =
            log.debug s"initialized wth dictionary: $dictionary"

        def seqToStr(s: (Int, List[Byte])) =
            Seq(s) map (s => (s._1, str(s._2))) head

        def msg(m: String)(code: Int, word: List[Byte],
            dictionary: mutable.TreeMap[Int, List[Byte]]) =
            log.debug s"$m code ($code->$word) added ${seqToStr(
                ((dictionary toSeq) last))}"

        def newOne(code: Int, word: List[Byte], dictionary:
            mutable.TreeMap[Int, List[Byte]]) =

```

```

    msg("new one")(code, word, dictionary)

def existing(code: Int, word: List[Byte], dictionary:
    mutable.TreeMap[Int, List[Byte]]) =
    msg("existing")(code, word, dictionary)

def firstCode(code: Int, word: List[Byte]) =
    log debug s"first code $code for word $word(${str(
        word)})"

}

def put(code: Int): List[Byte] = {
    if (previousCode isEmpty) {
        previousCode = Some(code)
        val word = dictionary(code)
        //Log.firstCode(code, word)
        word
    } else {
        val lastWord = dictionary(previousCode get)
        previousCode = Some(code)
        if (dictionary contains code) {
            val word = dictionary(code)
            addWord(dictionary, lastWord :+ (word head))
            //      Log.existing(code, word, dictionary)
            word
        } else {
            val word = lastWord :+ (lastWord head)
            addWord(dictionary, word)
            //      Log.newOne(code, word, dictionary)
            word
        }
    }
}

def addWord(dictionary: mutable.Map[Int, List[Byte]],
    word: List[Byte]): Unit = {
    dictionary += (((dictionary size) + 1) -> word)
}
}

```