Implementando o Algoritmo de Compressão de Lempel-Ziv-Welch

Pedro Garcia Freitas
11/0068408

28 de maio de 2011

Resumo

Neste trabalho, apresentamos e analisamos a performance da implementação própria do algoritmo de Lempel-Ziv-Welch (LZW).

1 Introdução

O algoritmo de Lempel-Ziv-Welch (LZW) é um algoritmo de compressão de uso geral baseado em dicionários, criado por Terry Welch, sendo uma melhoria no algoritmo de Lempel-Ziv (LZ78). Algoritmos derivados deste último, caracterizam-se por serem simples de implementar e com eficiência razoável na taxa de compressão e no tempo de codificação e decodificação.

O trabalho de Welch[2] introduz uma melhoria na compressão baseada em dicionários que previne problemas associados na adaptação do dicionário gerado dinamicamente em tempo de codificação. Segundo o autor desse artigo, diversos problemas desse tipo ocorrem nos algoritmos derivados dos trabalhos de Lempel-Ziv. Dentre tais problemas, o mais evidente é a memória necessária, pois os dicionários precisam crescer conforme novos padrões sejam encontrados na fonte[3]. Isto é, quanto mais dados são lidos da fonte, mais sequências são armazenadas no dicionário, exigindo cada vez mais memória. Como consequência imediata deste comportamento, temos que o aumento no dicionário exige um aumento na quantidade de endereços disponíveis, o que aumenta a complexidade de geração das chaves do dicionário, pois cada vez mais bits são necessários para elas. Para minimizar este efeito, o tamanho do dicionário é fixo. A transmissão (ou armazenamento) feita pelo LZW consiste apenas do dicionário e da sequência de índices (chaves).

2 O Algoritmo de Lempel-Ziv-Welch

Quando apresentado em 1984, Welch utilizou como exemplo a codificação de símbolos de 8 bits em uma sequência codificada de tamanho de até 12 bits. Como temos apenas 8 bits por símbolo, teremos um total de $2^8 = 256$ símbolos distintos. Com a sequência de 12 bits, é possível gerar $2^{12} = 4096$ símbolos distintos. Desta forma, a fonte não-codificada possui símbolos de 0 à 255, onde cada um fica na respectiva posição 0 à 255, enquanto que a fonte codificada possui esses símbolos acrescidos de sequências compostas por estes símbolos, que ficam armazenadas nas posições 256 à 4095. Em cada estágio de compressão, os símbolos de entradas são empilhados à uma sequencia codificada anteriormente, já presente no dicionário. Em seguida, checamos se aquela sequencia é presente na fonte original. Se for, ela é mantida no dicionário. Caso contrário, ela é removida.

2.1 Codificação

Os passos podem ser resumidos, conforme no algoritmo 1.

Algorithm 1 Codificação LZW

Require: Mensagem M
Ensure: Uma lista de combinações dos símbolos de M mapeadas em um dicionário D e a lista dos índices que permitem a decodificação a partir deste dicionário.
Inserir no dicionário D todos os 256 símbolos possíveis que podem ser encontrados em M.
while Não chegar ao final de M **do**
Encontrar em M a maior *string* W formada pela combinação dos símbolos iniciais.
Salvar no arquivo codificado o índice equivalente à sequência W no dicionário.
Adicionar W seguido pelo próximo símbolo lido na entrada do dicionário.
end while

Portanto, o dicionário será inicializado sempre da mesma forma: contendo o valor 0 na posição 0, o valor 1 na posição 1, até conter todos os símbolos possíveis para 8 bits, na posição 255. Isso significa que até a posição 255 teremos sempre caracteres únicos. De posse desses, o algoritmo simplesmente realiza uma varredura e procura uma composição desses caracteres na fonte. Então, o primeiro passo do algoritmo consistirá em gerar sequências de 2 caracteres, que são então armazenadas. O terceiro passo será uma combinação dessas sequências entre elas e os caracteres originais. Dessa repetição, é possível notar que esse comportamento pode se repetir indefinidamente até o dicionário conter todas as combinações que formam a própria mensagem original. Contudo, isso é indesejável, uma vez que estamos expandindo a informação ao armazenar o dicionário, que contém a própria mensagem original adicionada das combinações menores. Para evitar isso, definimos um tamanho fixo, em bits, para as chaves desse dicionário. Com isso, definimos também o crescimento do dicionário para um comprimento máximo.

2.2 Decodificação

O algoritmo de decodificação lê os valores da entrada compactada (codificada) e salva na saída o valor do dicionário correspondente àquela entrada. Seguindo o processo inverso à codificação, o valor armazenado é colocado é inserido no dicionário, concatenando-se este valor com o primeiro caractere da *string* obtida na próxima leitura. Então, o decodificador obtém o próximo símbolo, que já foi concatenado no passo anterior, e repete o processo até que não haja mais entradas.

3 Detalhes de Implementação

Nossa implementação foi realizada na linguagem Fortran, padrão 2003, tendo como compilador o gfortran na versão 4.7. Por questões de eficiência, algumas funções específicas deste compilador foram utilizadas, o que pode fazer com que o programa não compile em outros compiladores.

O programa é dividido em cinco módulos:

1. **LZW_Shared_Parameter:** Contém informações compartilhadas entre todos os módulos, são elas:

   - `COMPILER_INTEGER_SIZE`: Tamanho (em bits) da palavra utilizada pelo compilador para representar um tipo inteiro. Essa variável é utilizada para manipulação dos bits e na conversão dos tipos caractere (lidos) paraa os tipos inteiros (manipulados).
   - `BITS`: Quantidade de bits para o tamanho das chaves do dicionário. Isto é, o dicionário pode ter até 2^{BITS} elementos. Conforme exemplificado no trabalho de Welch, utilizamos 12 bits, tendo um total de 4096 chaves distintas.
   - `FILEIN`: Descritor do arquivo de entrada (66).
   - `FILEOUT`: Descritor do arquivo de saída (99).
   - `MAX_VALUE`: Tamanho máximo do dicionário.
   - `MAX_CODE`: Localização do maior código. Na prática, ficará na última posição do dicionário, se todo ele for preenchido.
   - `MAX_DICTIONARY_SIZE`: Próximo número primo maior que 2^{12}. O dicionário é um pouco maior para permitir mais combinações possíveis. O objetivo disso é que depois removemos as sequências maiores de menor frequência e enviamos o dicionário apenas com o tamanho máximo de símbolos 2^{12}.
   - `SYMBOL_SIZE`: Tamanho do símbolo original. (8 bits).
   - `MISSING_BITS`: Diferença entre o tamanho (em bits) da variável utilizada para leitura (caractere, 8 bits) para a variável utilizada para operações aritméticas (inteira, 32 bits). Essa constante é mantida para evitar que os *shifts* utilizados

na manipulação dos bits percam os bits ao serem atribuídos novamente para uma variável de menor tamanho no pós-processamento.

2. **LZW_Encoder:** Este é o módulo que contém as funções utilizadas na compressão, sendo que a rotina que fornece tal serviço é chamada `compress`.

3. **LZW_Decoder:** Funções utilizadas para expansão de um arquivo previamente codificado pelo módulo anterior. A interface de utilização das funções deste módulo é feita com uma única rotina `decompress`.

4. **codecIO**: Este módulo possui as funções para leitura e escrita nos arquivos codificados e decodificados. Simula leitura e escrita bit-a-bit.

5. **LZW**: módulo responsável por tratar os parâmetros de entrada, chamar a devida operação e salvar os arquivos de forma devida. É neste módulo que está implementado a aplicação (executável).

4 Resultados da Implementação

Para testar o algoritmo implementado, diversos tipos de arquivos foram utilizados. Para verificar se o arquivo descompactado possui exatamente o mesmo conteúdo do arquivo original, foram utilizados os algoritmos `MD5` e `SHA1` para comparação. A taxa de compactação é apresentada na tabela4.

Arquivo	Tamanho Original	Tamanho Compactado	Taxa de Compressão
article.pdf	3.2Mb	3.5Mb	0.914
lena.eps	328K	302K	1.086
restored.bmp	242K	96	2.520
test.xls	136K	59K	2.305

Referências

[1] Sayood, Khalid. *Introduction to data compression (2nd ed.)*. 1-55860-558-4. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA. 2000.

[2] Welch, T.A., A Technique for High-Performance Data Compression, Computer, vol. 17, no. 6, pp. 8-19, 1984.

[3] SALOMON, David. Data Compression: The Complete Reference. 2 ed. Springer, New York. 2000.

Apêndice A Arquivo: lzw_shared_parameters.f90 – LZW_Shared_

```
!
! lzw_shared_parameters.f90
!
! LZW Common Variables Used by Coder and Decoder
!
! Author: Pedro Garcia Freitas <sawp@sawp.com.br>
! May, 2011
!
! License: Creative Commons http://creativecommons.org/licenses/by-nc-nd/3.0/
!
module LZW_Shared_Parameters
  implicit none
  ! change this if compiler dont use 32 bits for integer
  integer, parameter :: COMPILER_INTEGER_SIZE = 32
  integer, parameter :: BITS = 12
  integer, parameter :: FILEIN = 66
  integer, parameter :: FILEOUT = 99
  integer, parameter :: MAX_VALUE = (2 ** BITS) - 1
  integer, parameter :: MAX_CODE = MAX_VALUE - 1
  integer, parameter :: MAX_DICTIONARY_SIZE = 5021
  integer, parameter :: SYMBOL_SIZE = 8
  integer, parameter :: MISSING_BITS = COMPILER_INTEGER_SIZE - SYMBOL_SIZE
  integer            :: status

  integer, dimension(0:MAX_DICTIONARY_SIZE) :: prefixCodes
  integer, dimension(0:MAX_DICTIONARY_SIZE) :: concatenatedSymbols
end module LZW_Shared_Parameters
```

Apêndice B Arquivo: codecIO.f90 – codecIO

```
!
! codecIO.f90
!
! bit IO routines for coder and encoder.
!
! Author: Pedro Garcia Freitas <sawp@sawp.com.br>
! May, 2011
!
! License: Creative Commons http://creativecommons.org/licenses/by-nc-nd/3.0/
!
```

```
module codecIO
  use LZW_Shared_Parameters
  implicit none

  contains
    subroutine setOutputCode(code)
      integer, intent(in) :: code
      integer             :: shiftedSymbol
      integer             :: buffer
      integer             :: shift
      integer, save       :: outputBitCount = 0
      integer, save       :: outputBitBuffer = 0

      shift = COMPILER_INTEGER_SIZE - BITS - outputBitCount
      shiftedSymbol = ishft(code, shift)
      outputBitBuffer = ior(outputBitBuffer, shiftedSymbol)
      outputBitCount = outputBitCount + BITS

      do
        if (outputBitCount < SYMBOL_SIZE) then
          exit
        endif
        buffer = ishft(outputBitBuffer, -MISSING_BITS)
        call setRawByte(buffer)
        outputBitBuffer = ishft(outputBitBuffer, SYMBOL_SIZE)
        outputBitCount = outputBitCount - SYMBOL_SIZE
      end do
    end subroutine setOutputCode


    subroutine setRawByte(symbol)
      integer   :: symbol

      call fputc(FILEOUT, achar(symbol))
    end subroutine setRawByte


    function getRawByte()
      integer   :: getRawByte
      character :: bufferedByte

      call fgetc(FILEIN, bufferedByte, status)
      getRawByte = iachar(bufferedByte)
```

```
    end function

    function getInputCode()
      integer         :: getInputCode
      integer         :: returnn
      integer         :: shiftedBit
      integer         :: integerInputBuff
      integer, save   :: inputBitCounter = 0
      integer, save   :: inputBitBuffer = 0

      do
        if (inputBitCounter > MISSING_BITS) then
          exit
        endif

        integerInputBuff = getRawByte()
        shiftedBit = ishft(integerInputBuff, MISSING_BITS - inputBitCounter)
        inputBitBuffer = ior(inputBitBuffer, shiftedBit)
        inputBitCounter = inputBitCounter + SYMBOL_SIZE
      end do

      returnn = ishft(inputBitBuffer, BITS - COMPILER_INTEGER_SIZE)
      inputBitBuffer = ishft(inputBitBuffer, BITS)
      inputBitCounter = inputBitCounter - BITS
      getInputCode = returnn
    end function getInputCode
end module codecIO
```

Apêndice C Arquivo: lzw_encoder.f90 – LZW_Encoder

```
!
! lzw_encoder.f90
!
! LZW Coder (Compressor)
!
! Author: Pedro Garcia Freitas <sawp@sawp.com.br>
! May, 2011
!
! License: Creative Commons http://creativecommons.org/licenses/by-nc-nd/3.0/
!
module LZW_Encoder
  use LZW_Shared_Parameters
```

```
use codecIO
implicit none

integer, parameter                          :: HASH_SHIFT = BITS - SYMBOL_SIZE
integer, dimension(0:MAX_DICTIONARY_SIZE) :: symbolValues

contains
  subroutine compress()
    integer   :: symbol
    integer   :: codedSymbol
    integer   :: index
    integer   :: nextSymbol

    nextSymbol = COMPILER_INTEGER_SIZE * SYMBOL_SIZE
    symbolValues(:) = -1

    codedSymbol = getRawByte()
    do
      symbol = getRawByte()
      if (status /= 0) then
        exit
      endif

      index = getPositionOnDictionary(codedSymbol, symbol)
      if (symbolValues(index) /= -1) then
          codedSymbol = symbolValues(index)
      else
        if (nextSymbol <= MAX_CODE) then
          symbolValues(index) = nextSymbol
          nextSymbol = nextSymbol + 1
          prefixCodes(index) = codedSymbol
          concatenatedSymbols(index) = symbol
        endif
        call setOutputCode(codedSymbol)
        codedSymbol = symbol
      endif
    end do
    call setOutputCode(codedSymbol)
    call setOutputCode(MAX_VALUE)
    call setOutputCode(0)
  end subroutine
```

```
    function getPositionOnDictionary(hashPrefix, hashSymbol)
      integer, intent(in) :: hashPrefix
      integer, intent(in) :: hashSymbol
      integer             :: getPositionOnDictionary
      integer             :: index
      integer             :: offset

      index = ishft(hashSymbol, HASH_SHIFT)
      index = ieor(index, hashPrefix)
      if (index == 0) then
        offset = 1
      else
        offset = MAX_DICTIONARY_SIZE - index
      endif

      do
        if (symbolValues(index) == -1) then
          getPositionOnDictionary = index
          exit
        endif

        if (prefixCodes(index) == hashPrefix .and. &
            & concatenatedSymbols(index) == hashSymbol) then
          getPositionOnDictionary = index
          exit
        endif

        index = index - offset
        if (index < 0) then
          index = index + MAX_DICTIONARY_SIZE
        endif
      end do
    end function
end module LZW_Encoder
```

Apêndice D Arquivo: lzw_decoder.f90 – LZW_Decoder

```
!
! lzw_decoder.f90
!
! LZW Decoder (Expanssor)
!
```

```
! Author: Pedro Garcia Freitas <sawp@sawp.com.br>
! May, 2011
!
! License: Creative Commons http://creativecommons.org/licenses/by-nc-nd/3.0/
!
module LZW_Decoder
  use LZW_Shared_Parameters
  use codecIO
  implicit none

  type DECODE_BUFFER_STACK
    integer, dimension(0:MAX_DICTIONARY_SIZE) :: decoderStack
    integer                                   :: top
  end type DECODE_BUFFER_STACK

  type(DECODE_BUFFER_STACK) :: stack

  contains
    subroutine decompress()
      integer            :: nextSymbol
      integer            :: newSymbol
      integer            :: oldSymbol
      integer            :: symbol
      integer            :: popedSymbol

      nextSymbol = COMPILER_INTEGER_SIZE * SYMBOL_SIZE
      oldSymbol = getInputCode()
      symbol = oldSymbol

      call setRawByte(oldSymbol)

      do
        newSymbol = getInputCode()

        if (newSymbol == MAX_VALUE) then
          stop
        endif

        if (newSymbol >= nextSymbol) then
          stack%decoderStack(0) = symbol
          call decodeSymbol(stack%decoderStack(1:), oldSymbol)
        else
          call decodeSymbol(stack%decoderStack(:), newSymbol)
```

```
    endif

    symbol = stack%decoderStack(stack%top)

    do while(stack%top >= 0)
      popedSymbol = stack%decoderStack(stack%top)
      call setRawByte(popedSymbol)
      stack%top = stack%top - 1
    end do

    if (nextSymbol <= MAX_CODE) then
      prefixCodes(nextSymbol) = oldSymbol
      concatenatedSymbols(nextSymbol) = symbol
      nextSymbol = nextSymbol + 1
    endif
    oldSymbol = newSymbol
  end do
end subroutine


subroutine decodeSymbol(buffer, code)
  integer, intent(in)  :: code
  integer              :: symbol
  integer              :: j, i
  integer, dimension(:), intent(inout) :: buffer

  j = 0
  symbol = code
  stack%top = 0
  do
    if (symbol < COMPILER_INTEGER_SIZE * SYMBOL_SIZE) then
      exit
    endif

    if (j >= MAX_CODE) then
      print *, "Decoding error"
      stop
    endif

    i = stack%top + 1
    buffer(i) = concatenatedSymbols(symbol)
    symbol = prefixCodes(symbol)
    stack%top = stack%top + 1
```

```
        j = j + 1
      end do
      i = j + 1
      buffer(i) = symbol
    end subroutine decodeSymbol
end module LZW_Decoder
```

Apêndice E Arquivo: lzw.f90 – LZW

```
!
! lzw.f90
!
! LZW Coder and Decoder
!
! Author: Pedro Garcia Freitas <sawp@sawp.com.br>
! May, 2011
!
! License: Creative Commons http://creativecommons.org/licenses/by-nc-nd/3.0/
!

module LZW
  use LZW_Shared_Parameters
  use LZW_Encoder
  use LZW_Decoder
  implicit none

  contains
    subroutine init(input, output, operation, fileName)
      character(len=100), intent(in) :: input
      character(len=100), intent(in) :: output
      character(len=1)   :: operation
      character(len=100) :: fileName

      if (operation /= 'd' .and. operation /= 'e') then
        print *, "Usage: " // trim(fileName) // " <operation> input output"
        print *, "Possible operations: "
        print *, "    e -> encode (compress)"
        print *, "    d -> decode (inflate)"
        stop
      endif

      open(unit=FILEIN, file=input, action="read", status="old", access='stream', for
```

```
      open(unit=FILEOUT, file=output, action="write", status="replace", &
           access='stream', form="formatted")

      if (operation == 'd') then
        print *, "Decoding..."
        call decompress()
      else
        print *, "Coding..."
        call compress()
      endif

      close(unit=FILEIN)
      close(unit=FILEOUT)
    end subroutine init
end module LZW

program main
  use LZW

  character(len=100) :: input
  character(len=100) :: output
  character(len=1)   :: operation
  character(len=100) :: fileName

  call getarg(0, fileName)
  call getarg(1, operation)
  call getarg(2, input)
  call getarg(3, output)
  call init(input, output, operation, fileName)
end program main
```