Algoritmo LZSS

Alumno: Kenny Alejandro Código grupo: 8.1

Lempel–Ziv–Storer–Szymanski (LZSS) es un algoritmo de compresión de datos sin pérdida (*lossless*) y que es derivado del LZ77[1]. Este algoritmo fue creado en 1982 por James Storer y Thomas Szymanski,quienes lo describieron en el artículo "Compresión de datos mediante sustitución textual" publicado en Journal of the ACM (1982, pp. 928-951)[2].

¿Cuando se usan los algoritmos de compresión lossless?

Se utilizan cuando la información a comprimir es crítica y no se puede perder información,por ejemplo en los archivos ejecutables, tablas de bases de datos, o cualquier tipo de información que no admita pérdida.

¿Cómo comprime el LZSS?

Al ser un algoritmo deducido del LZ77, creo que es conveniente realizar una breve explicación de la manera en que este comprime.

- LZ77[3][5] Algoritmo que sobre una secuencia de símbolos usa una *Sliding Window* (ventana corrediza).Esta contiene a dos sub-windows: el diccionario o *search-buffer* (que contiene los símbolos ya codificados) y el *look-ahead-buffer o* LAB (contiene los símbolos por codificar).
    - ○ Descripción de compresión: Por cada elemento que esté al inicio del LAB se ha de buscar en el search-buffer la coincidencia de tamaño más larga.Acto seguido se ha de codificar esta coincidencia mediante el uso de un token.Un token,para LZ77, es una tripleta de datos (posición,longitud,símbolo):
        - ■ posición o desplazamiento *o offset*: ubicación del prefijo más largo del LAB encontrado en el diccionario actual; este campo utiliza log2 (|dict.|) bits, donde |dict.| es la longitud del diccionario;
        - ■ longitud o *length*: longitud de la cadena combinada; esto requiere log2 (|LAB|) bits;
        - ■ símbolo: el primer símbolo siguiente a la cadena coincidente, para símbolos ASCII, esto usa 8 bits.
    - ○ Implementación:

        - Diccionario[9]: Circular buffer(similar a circular queue [6])
        - Look-ahead-buffer[8][9]: Circular buffer(similar a u circular queue[6])

- LZSS[3][4][5] Al igual que su antecesor hace uso similar tanto de la Sliding Window como de la búsqueda de similitudes entre el diccionario y el LAB. Existen dos diferencias principales respecto al LZ77:
    - La primera que el token se reduce de 3 (posición, longitud, símbolo) a sólo 2(posición, longitud)
    - La segunda es que su antecesor siempre guarda un par posición/longitud aún si la coincidencia era de un solo byte (en cuyo caso usa más de ocho bits para representar un byte) por lo que se añade el uso de un bit de Flag (bandera) para saber si lo que sigue a este es un literal o un par posición/longitud. Con lo cual el token final sería de (bit(code)):
        - bit = 0 ⇒ code = (símbolo)
        - bit = 1 ⇒ code = (posición, longitud)
    - Implementación:
        - Diccionario: Circular buffer[9](similar a un circular queue),Binary search trees[4][8][9],Suffix trees[7] ,Suffix arrays[3],Linked list[8],Hash table[9]
        - Look-ahead-buffer[4][8][9]:Circular buffer(similar a circular queue [6])

Toma de decisiones:

- Implementación Integrada(Implementación final):
    - Token de (flag,offset, length)
    - Search buffer:
        - Tamaño 4096 bytes[3][8][9](12 bits para expresar el offset)
        - Implementado con Circular buffer[9](similar a una circular queue en acciones de insertar y quitar) tomó esta opción debido a que es menos compleja que otras opciones.
    - Look ahead buffer
        - Tamaño 16 bytes[9] (4 bits para expresar el tamaño)
        - Implementado con circular buffer(similar a una circular queue en acciones de insertar y quitar) debido a que es la manera adecuada y

aconsejada por diversas fuentes como [4], [8] y [9].

- ○ Flags juntados de 8 en 8 matches y guardados en un byte que es escrito siempre antes que los 8 matches, sugerida en [9].
- ○ Lectura de los datos de entrada no como una lista de caracteres sino como una lista de bytes, lo cual me ayuda a no depender de la codificación del texto de entrada.

● Implementación Optimizada(No integrada por causas relativas al tiempo limitado y a la complejidad que esta conlleva):

- ○ Search buffer implementada con binary tree, con el objetivo de mejorar el tiempo de búsqueda de una coincidencia.Tomó esta decisión basándose en las conclusiones de la referencia [8] y en datos adicionales encontrados en [4] y [9].

Breve descripción de investigacion y busqueda sobre LZSS

La búsqueda de información sobre mi algoritmo la he dividido en 3 etapas que siguen un una cronología secuencial y son las siguientes:

1. Búsqueda de bibliografía de mediante libros: Esta fue la etapa inicial y de toma de contacto con la información y conceptos genéricos relacionados con el algoritmo.Los libros consultados fueron [4] ,[10] y [11].
2. Búsqueda de información en la web de forma genérica:Esta es la etapa de reforzamiento sobre los conocimientos aprendidos y para resolver las dudas no cubiertas por los libró de la primera etapa. Las páginas web y artículos leídos fueron las siguientes [1] ,[2],[3].
3. Búsqueda de implementaciones de ejemplo y de mejoras: Esta fue la etapa previa a la implementación del algoritmo y con la que me encontré vicisitudes para poder entender la forma en que los autores implementan los códigos. Los artículos leídos fueron [5] [8][9]

Proceso de implementación:
Las 3 primeras versiones de mi algoritmo las implemente en C y la última en Java.

1. La primera versión estuvo centrada en implementar la base del algoritmo. Para poder interpretar y debugar la salida, la codificación de un token era la siguiente:
    a. Flag escrito usando 1 byte.
    b. Si el Flag era 1 - > tanto el tamaño como el offset eran escritos usando 2 bytes cada uno.
    c. Si el Flag era 0 - > se escribía el carácter(1 byte).
2. La segunda versión estuvo enfocada a aplicar la mejora de juntar los flags de los 8 tokens consecutivos para ser almacenarlos ocupando 1 solo byte.

3. La tercera versión estuvo orientada a representar la pareja(length, offset) de un token usando 2 bytes de los cuales los 4 bits de mayor peso son para el tamaño y los 12 bits restantes son del offset.

4. La cuarta versión estuvo dirigida hacia la traducción del código de C a Java .Además, en esta aplique la mejora de tratar los datos de entrada no como caracteres sino como una lista de bytes.

Descripción de los test realizados

1) Jp1.txt
   Objetivo: Comprobar que el algoritmo soporta la compresión de textos vacíos.
   Resultados: El algoritmo funciona con normalidad.
   Tamaño Inicial:0
   Tamaño final: 21
   Ratio: -1.0 (valor esperado cuando el tamaño de archivo inicial es 0)
2) Jp2.txt
   Objetivo: Comprobar que el algoritmo es capaz de comprimir y descomprimir textos simples escritos en codificación ASCII. Tamaño final
   Resultados: El texto descomprimido es igual al original.
   Tamaño Inicial:3500
   Tamaño final: 2023
   Ratio: 0.58
3) Jp3.txt
   Objetivo: Comprobar que el algoritmo es capaz de comprimir y descomprimir textos con caracteres especiales(Ñ,ó,etc) de codificaciones distintas de ASCII(UTF-8,UTF-16,etc).
   Resultados: El texto descomprimido es igual al original.
   Tamaño Inicial: 446866
   Tamaño final: 228795
   Ratio: 0.51
4) Jp4.txt
   Objetivo: Comprobar que con textos que tienen poca repetición, el ratio de compresión es mayor, es decir , el fichero resultante representa el 58% del tamaño del fichero inicial.
   Resultados:
   Tamaño Inicial: 943
   Tamaño final: 768
   Ratio : 0.81
5) Jp5.txt
   Objetivo: Comprobar que con textos que tienen mucha repetición, el ratio de compresión es menor, es decir, el fichero resultantes representa el 15% del tamaño del fichero inicial.
   Resultados:
   Tamaño Inicial:166718
   Tamaño final: 24476
   Ratio:0.15

Referencias

[1]Wikipedia's information about Lempel–Ziv–Storer–Szymanski
https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymans ki

https://es.wikipedia.org/wiki/LZSS
[2] J. Storer and T. Szymanski. Data compression via textual substitution. Journal of ACM, 29(4):928–951,1982.

Link

[3] On the Use of Suffix Arrays for Memory-Efficient Lempel-Ziv Data Compression
https://www.researchgate.net/publication/24166892_On_the_Use_of_Suffix_Arrays_for_Mem ory-Efficient_Lempel-Ziv_Data_Compression

[4] DAVID SALOMON, *Data Compression:the complete reference.*Edición 4. London:Springer: 2007,ISBN-10: 1-84628-602-6, pp 179-181 1er)

1) Libro fisico:
   https://discovery.upc.edu/iii/encore/record/C__Rb1338375__Sdavid%20salomon__Orighthresult__U__X4;jsessionid=442BDE3691D1E311727E1A2206ECD4A4?lang=cat&suite=def
2) Libro electronico:
   http://read.pudn.com/downloads167/ebook/769449/dataCompress.pdf

[5] GLZSS: LZSS Lossless Data Compression Can Be Faster pg 2
http://staff.ustc.edu.cn/~bhua/publications/GLZSS-2014.pdf

[6] Circular queue *https://www.youtube.com/watch?v=xQdoA_7k4I4*

[7] N. Jesper Larsson, Structures of String Matching and Data Compression pg21-32
http://www.larsson.dogma.net/thesis.pdf

[8] Longest-match String Searching for Ziv– Lempel Compression
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.8587&rep=rep1&type=pdf
[9] Michael Dipperstein , LZSS (LZ77) Discussion and Implementation
http://michael.dipperstein.com/lzss/#linked_list

[10]A Concise introduction to data compression [Recurs electrònic] / David Salomon

*https://cataleg.upc.edu/record=b1440562~S1*cat*

[11] Introduction to data compression / Khalid Sayood

https://cataleg.upc.edu/record=b1440516~S1*cat