2015

Maestría en Ciencia de Datos

Minería y Análisis de Datos

Compresión de Datos

Instituto Tecnológico Autónomo de México

Profesor:

Dr. Ángel Fernando Kuri Morales

Equipo 5:

Augusto Hernández |112835

Gabriela Flores Bracamontes | 160124

Mónica Ballesteros | 124960

Rodrigo Sarmiento | 116388

Tania Patiño | 137175

México, DF 11 de noviembre de 2015

Tabla de contenido

Figuras

Tablas

1. Introducción

En el presente documento tiene por objeto explicar en qué consiste la compresión de datos, así como proporcionar ejemplos de aplicaciones en diversas áreas. La compresión de datos puede clasificarse de forma general en dos tipos: con pérdida y sin pérdida de información.

Existe una gran variedad de algoritmos para generar códigos de compresión sin pérdida, de los cuales se cubrirá el Código de Huffman (estadístico) y el de Lempel-Ziv-Welch **(**de diccionario**.** Estos permiten que los mensajes permanezcan intactos. Por otra parte, la compresión con pérdida se puede aplicar principalmente a contenido Multimedia (Video, Imágenes, Sonido), o de comunicación (como Fax y Modem), en donde el mensaje puede preservarse sin necesidad de incluir toda la información inicial. Para efectos de este documento se hace énfasis en la compresión de datos sin pérdida.

Comenzaremos por describir la Teoría de la Comunicación, para tener el contexto necesario y desarrollar subsecuentemente los conceptos de la Compresión de Datos. En la siguiente sección se describen los dos algoritmos mencionados en el párrafo anterior (Huffman y Lempel-Ziv-Welch) junto con ejemplos.

1.1. Teoría de la Comunicación

Es una disciplina que se fundamenta en el enfoque matemático del estudio de la recopilación, manipulación de la información [5] y cuantificación de los procesos que se realizan sobre la información.

Información: conjunto de datos procesados que forman un mensaje y modifica el conocimiento de un sistema o individuo, debe distinguirse del significado de la información, pues ésta última resulta una cuantificación subjetiva y personal de cada individuo.

Los componentes de la comunicación consisten de:

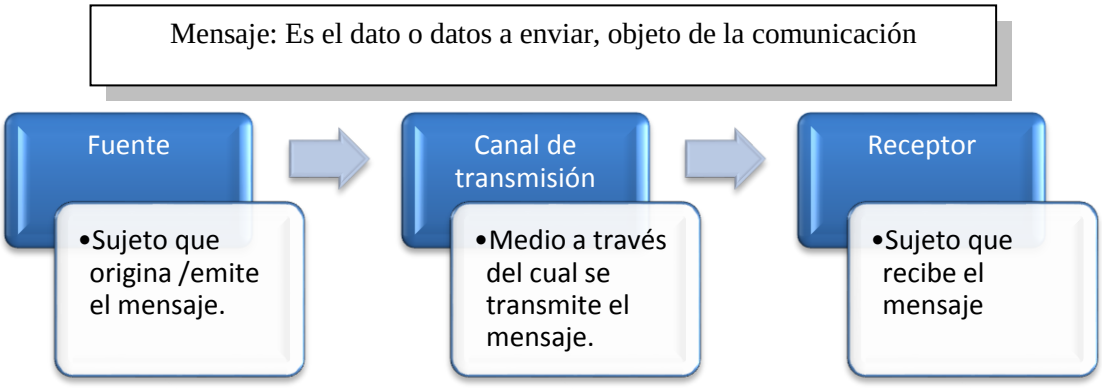

Figura 1.-Componentes de la comunicación

La información tiene dos teorías matemáticas, la teoría estadística de la información (Claude Shannon,1948) y la teoría algorítmica de la información (Kolmogorov, 1965).

Ambas proporcionan una medida de la información contenida en el mensaje (independiente de la interpretación del individuo) [5]. A continuación se presentan algunos elementos introducidos por el trabajo de Shannon "A mathematical theory of communication", en el cual se destaca que mostró que la información se puede cuantificar con precisión absoluta en cualquiera de sus formas.

Algunos conceptos fundamentales para entender la teoría de la información, propuestos en dicho trabajo, son los siguientes:

- **Capacidad del canal y el teorema de codificación de canales ruidosos**: Dentro de estos conceptos se introduce el de que los canales de comunicación tienen un límite de velocidad, que se puede medir en dígitos binarios por segundo. Se probó entonces matemáticamente que es imposible obtener una comunicación libre de errores por encima del límite sin tener pérdida de información, independientemente de cuánto se logre comprimir la información. En contraparte, la ventaja de esto es que es posible transmitir información con error cero, codificando la información de manera que se llegue al límite sin errores, independientemente de la cantidad de ruido en la señal.

- **Representación Digital**: Shannon enunció formalmente que el contenido de un mensaje es independiente de su transmisión. Esto significa que no es importante la forma en que se represente, ya que para el canal todo puede ser un conjunto de ceros y unos.

- **Eficiencia de la representación**: En este mismo trabajo discute la eficiencia de las representaciones de los datos, que hoy en día es homólogo a la compresión. El objetivo básico es codificar de forma que se retire la redundancia en la información para hacer el mensaje más pequeño.

Formalmente, la conceptualización que propone Shannon acerca de una fuente de información S se basa en que es un ente abstracto que produce símbolos de un alfabeto finito, lo que se conoce como **Teoría de estadística de la información.**

El modelo propuesto por Shanon una fuente es una cadena de Markov ergódica. Las cadenas de Markov ergódicas son aquellas en las que la probabilidad de pasar de un estado a otro son constantes. [5].

Cantidad de información: es una función creciente e inversa a la probabilidad emisión del símbolo, es decir, un símbolo con menor probabilidad de emisión proporciona mayor cantidad de información.

Para una fuente de información *S* la cantidad de información *Si* se define como:

$$I(s_i) = -\log_2(p_i)$$

Donde: p_i es la probabilidad de que el símbolo s_i sea emitido por *S* y log_2 (logaritmo base dos).

Entropía: es la medida media de información dada por la esperanza matemática de la información de cada símbolo.

$$H(S) = -\sum_{i=1}^{n} p_i \log_2 (p_i)$$

Considerando como base la teoría estadística de la información, donde se asume que la fuente es ergódica, se basan los esquemas de compresión sin pérdida que se explicarán más adelante, en la sección 2.1.

Por otra parte, Kolmogorov definió información como la longitud medida en bits, del programa más corto capaz de reproducir el mensaje, conocido como la **Teoría algorítmica de la información**.

Los esquemas de algoritmos de compresión con pérdida se fundamentan en la teoría algorítmica de la información.

1.2. Compresión de datos

La generación masiva de información ha hecho surgir la necesidad de un aumento en la capacidad de almacenamiento de datos y en la velocidad de procesamiento en las computadoras. Estos acontecimientos enfatizan la necesidad de compresión de datos.

Compresión de datos: es el proceso que consiste en reducir el tamaño de la información eliminando la redundancia de los datos originales y aumentando la entropía de los datos [8]. En otras palabras, el objetivo central es representar de la manera más eficaz un conjunto de datos.

A continuación se detalla la clasificación principal de la compresión de datos mencionados en la introducción (compresión con pérdida y sin pérdida).

Compresión sin pérdida: Reconstruye una copia fiel del mensaje original [7]. Es utilizado cuando se requiere la conservación de absolutamente toda la información y es comúnmente utilizada en texto.
Dentro de este esquema de compresión se distinguen dos tipos básicos, los estadísticos y los de diccionario [7].

Ejemplos:

1. Codificación de Longitud de Cadenas - RLE (iconos, logotipos)
2. Algoritmo aritmético
3. Compresión de Huffman
4. Compresión Lempel-Ziv LZ77 & LZ78 - LZW
5. PPM (Prediction by Partial Matching)
6. Compresión de datos por metasímbolo–MLDC [6]

Compresión con pérdida: Reconstruye una versión lo más parecida posible o aproximada al mensaje original [7], por lo que implica pérdida de información. Un ejemplo de aplicaciones en las que resulta eficaz es en información Multimedia. Por su naturaleza, estas representaciones digitalizadas de fenómenos analógicos no son perfectos, así que la idea de que el mensaje, antes y después de la compresión, no coincida exactamente es más aceptable, y en muchas ocasiones no es necesaria la reconstrucción fiel de la

información, ya que mucha de la información con niveles específicos resulta imperceptible a los sentidos humanos.

La mayoría de las técnicas de compresión con pérdida se pueden ajustar para diferentes niveles de calidad, obteniendo una mayor precisión a cambio de una compresión menos eficaz.

Ejemplos:

1. Transformada de coseno discreta
2. Codificación por transformación (Utilizada en señales de audio o imágenes fotográficas)
3. Compresión fractal (Imágenes digitales)

Además de esta clasificación general, la compresión puede también delimitarse con base en algunas de sus características específicas.

- Por su simetría: En la **compresión simétrica**, se utiliza el mismo método para comprimir y para descomprimir los datos. Por lo tanto, cada operación requiere la misma cantidad de trabajo. En general, se utiliza este tipo de compresión en la transmisión de datos. En la **compresión asimétrica** se requiere más trabajo para una de las dos operaciones. Es frecuente buscar algoritmos para los cuales la compresión es más lenta que la descompresión.
- Por su adaptabilidad: Un método de compresión **no adaptable** es rígido y no modifica su operación o sus parámetros en respuesta de los datos particulares que se van a comprimir. Un método **adaptable** examina los datos y modifica su operación o parámetros de acuerdo a ellos. Un método **semi-adaptable** o de dos pasadas, lee los datos que van a comprimirse para determinar sus parámetros de operación internos (primera fase) y a continuación realiza la compresión (segunda fase) de acuerdo a los parámetros fijados en la primera fase [8].

2. Compresión sin pérdida

En el ámbito de la investigación en el área de compresión de datos ha habido un arduo trabajo respecto a la codificación Huffman, éste es un método para la compresión de datos sin pérdida. Huffman es un algoritmo que sirve como base para programas populares que se ejecutan en diversas plataformas. Algunos de ellos, utilizan sólo el método de Huffman, mientras que en otros, forma parte de un proceso de compresión de varios pasos.

El método de Huffman es similar al método de Shannon–Fano (*Origen*). En general, produce mejores códigos; al igual que el método de Shannon–Fano, obtiene el mejor código cuando las probabilidades de los símbolos son potencias negativas de 2.

La principal diferencia entre estos dos métodos es, que Shannon–Fano (*Origen*) construye sus códigos de arriba abajo (desde los bits de más a la izquierda a los situados más a la derecha), en cambio Huffman lo hace mediante un árbol de código, desde lo más profundo del mismo, hasta arriba (genera los códigos de derecha a izquierda).

En la sección siguiente se describe este algoritmo de Huffman, cómo es que se generan los códigos de Huffman y la construcción de los árboles binarios. Además de explicar un poco acerca de los algoritmos análogos a Huffman.

2.1. Métodos de compresión basados en estadística

Usan las propiedades estadísticas de los datos que van a comprimirse para asignar códigos de longitud variable a los símbolos individuales en los datos.

La compresión estadística obliga a conocer de antemano la distribución de probabilidades de la fuente. Si esto no es posible hay que leer primero toda la fuente (por ejemplo un texto) y calcular luego la estadística de dicha fuente. Por lo tanto la compresión se realiza en dos pasadas.

Existen varios métodos estadísticos propuestos, la principal diferencia entre ellos es la forma en la que cada uno obtiene las probabilidades de los símbolos, la calidad de la compresión que se logra depende de que tan bueno sea ese modelo, el modelo se basa en el contexto en el que ocurren los símbolos. Dadas las probabilidades de los símbolos, la codificación se encarga de convertir dichas probabilidades en una cadena de bits para obtener los datos en forma comprimida. Los métodos estadísticos más comúnmente usados son Codificación Huffman, Codificación Aritmética y PPM.

2.1.1. Código de Huffman

La codificación de Huffman es una técnica para la *compresión de datos*, ampliamente aplicada y muy eficiente. Debe su nombre a David Albert Huffman, que estudió en el MIT, Massachusetts Institute of Technology, y fue en 1952 qué publicó "*A Method for the Construction of Minimum-Redundancy Codes",* en donde dio a conocer el "*algoritmo de Huffman*" que generá el "*código de Huffman*"[1].

El **Algoritmo de Huffman** se define inicialmente con un "bosque de árboles", en donde el peso de un árbol es igual a la suma de las frecuencias de sus hojas. A continuación se seleccionan los dos árboles de menor peso: T_1 y T_2, y se forma un nuevo árbol cuyos sub-árboles son T_1 y T_2 respectivamente. Este proceso se repite C-1 veces. En donde C corresponde al número inicial de árboles con un único nodo (un árbol por cada carácter). Al finalizar el proceso queda un solo árbol, que es justamente el árbol óptimo de *Huffman*.

Algunas aplicaciones del código de Huffman:

Esta técnica de compresión es un **método estadístico** que permite asignar un código binario a diversos símbolos a comprimir, tales como píxeles o caracteres en un texto. Se muestra a continuación un simple ejemplo de Árbol de Huffman para una secuencia de letras dadas y en una tabla se muestran sus correspondientes códigos de Huffman [2].

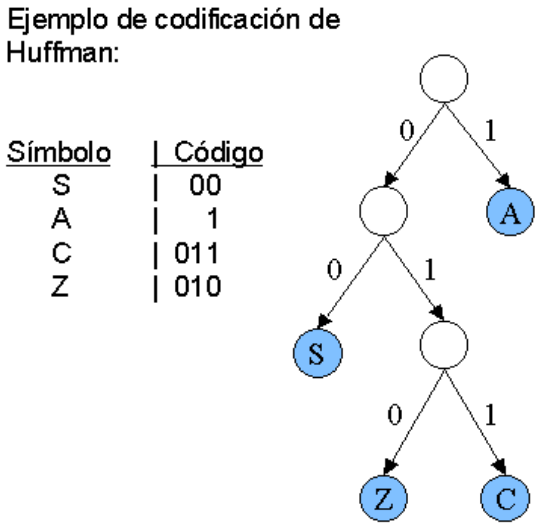

Figura 2.Códigos de Huffman y Árbol de Huffman

2.1.1.1. Ejemplo

Este es un ejemplo de una de las aplicaciones de la compresión de Huffman que se refiere a la sustitución de cadenas de caracteres por códigos que ocupan menos espacio (códigos binarios) (0-1).

Los métodos de compresión buscan en general las cadenas de datos que se repiten más veces en una secuencia: cuanto mayor sean estas cadenas y más veces se repitan mayor será el grado de compresión, pero generalmente no ocurre. Está comprobado que las cadenas grandes casi no se repiten dentro de una secuencia de datos. Usualmente se encuentran cadenas pequeñas que se repiten.

Esta codificación de Huffman, es fácil de implementar en hardware. En la compresión de los símbolos de los datos, el codificador de Huffman crea códigos más cortos para símbolos que se repiten frecuentemente y códigos más largos para símbolos que ocurren ocasionalmente.

Analizando un fragmento de la canción **"***I need somebody***"** que interpreta Brian Adams.®

I've been lookin' for someone
Between the fire and the flame
We're all lookin' for something
To ease the pain

Now who can you turn to
When it's all black and white
And the winners are losers
You see it every night

I need somebody
Somebody like you
Everybody needs somebody
I need somebody
Hey what about you
Everybody needs somebody, oh

When you're out on the front line
And you're watchin' them fall
It doesn't take long to realize
It ain't worth fightin' for, oh

I thought I saw the Madonna
When you walked in the room
Well your eyes were like diamonds

And they cut right through, oh they cut right through

I need somebody
Somebody like you
Everybody needs somebody, oh yeah
I need somebody
Hey what about you
We all need somebody, hey

Another night, another lesson learned
It's the distance keeps us sane
But when the silence leads to sorrow
We do it all again, all again yeah

I need somebody
Somebody like you
Everybody needs somebody, oh yeah
I need somebody
Oh what about you
Everybody needs somebody

I need somebody
Somebody like you
Everybody needs somebody
Need somebody
I need somebody
Yeah what about you baby
We all need somebody
I need somebody
Everybody needs somebody
I need somebody (hey)
Everybody needs somebody

Para generar el código de Huffman, se crea una tabla que asigna un valor de frecuencia a cada frase. En la siguiente tabla, se observan las asignaciones de valor de frecuencia a cada frase. Se ignoran las mayúsculas.

Un fragmento:

I need somebody
Somebody like you
Everybody needs somebody
I need somebody
Hey what about you
We all need somebody

Frase	Símbolo	Frecuencia
I	i	2
need	n	2
somebody	s	5
like	l	1
everybody	e	1

Tabla 1.- Frecuencia de repetición de palabras en la canción "I need somebody®"

El proceso de formar el árbol con nodos a partir del código de *Huffman* sigue los siguientes pasos:

1. Inicialmente se designan los símbolos como los nodos formados por la unión de una hoja y la rama de un árbol.
2. Empezando por los dos nodos de menor peso, agregar el par de menor valor en un nuevo nodo.
3. Repetir el proceso para una nueva serie hasta que la serie de símbolos éste representado por un solo nodo.
4. Mostrar el resultado.

***Nota**:* Un código de Huffman puede ser generado para cada símbolo mediante la asignación de un dígito binario para cada rama. Vamos a asignar el dígito binario 1 para cada rama del lado izquierdo y el dígito binario cero para cada rama del lado derecho.

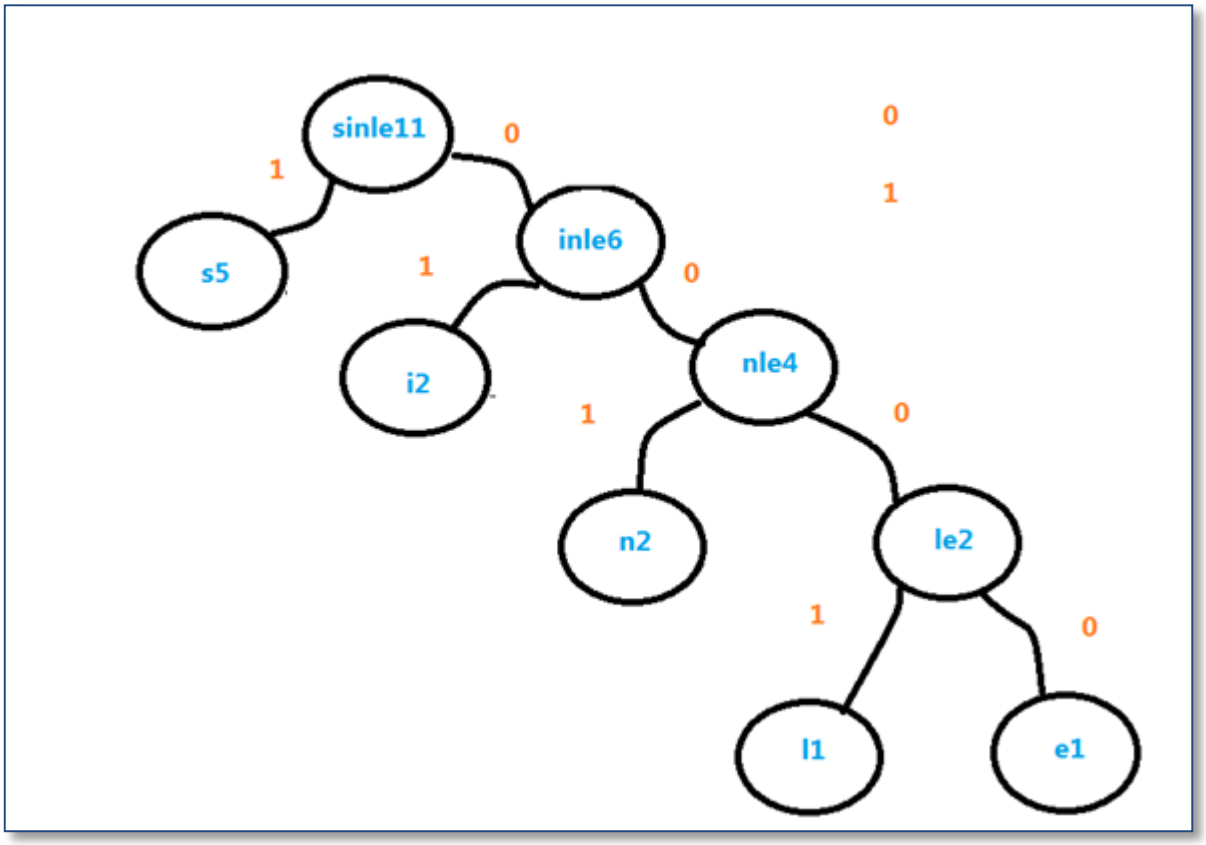

Figura 3.- Figura de árbol de Huffman para la canción I need somebody®

Frase	Símbolo	Frecuencia	Longitud código	Código
I	I	2	2	01
need	N	2	3	001
somebody	S	5	1	1
like	L	1	4	0001
everybody	E	1	4	0000

Tabla 2.- Frecuencia de repetición de palabras en la canción I need somebody®

En la siguiente fase *Decodificación*, se buscan las cadenas de bits, iniciando desde el nodo superior, y siguiendo hacia la ramificación izquierda o derecha dependiendo del valor tomado por la cadena de bits, se continúa hasta que un nodo de hoja es alcanzado. Así el símbolo decodificado es el símbolo asociado con esa hoja.

De este análisis se tiene que los primeros bits en la cadena de bits de salida se refieren: 01-001-1-0001-0000 …

La eficiencia de este código puede ser calculado comparando el número de bits requeridos para realizar la letra de la canción.

Entonces tenemos que la longitud de este código de Huffman es:

2(2) + 2 (3) + 5 (1) + 1 (4) + 1 (4) = 4 + 6 + 5 + 4 + 4 = 23 bits

Comparando un código de 3 bits, la longitud es de 35(3) = 105 bits; y para un código ideal de 7 símbolos la longitud es de 35 (log27) = 98.3 bits.

El código de Huffman comprime la letra en cerca de 20 por ciento, pero este ejemplo no incluye el costo de transmitir la tabla inicial del código de Huffman al decodificador.

2.1.1.2. Código de Huffman en PHP

Se presentan en estas líneas de código una implementación sencilla del Algoritmo de Huffman en PHP, donde se genera una cadena de Códigos de Huffman.

```
<html>
   <head>
      <meta charset="UTF-8">
      <title></title>
   </head>
   <body>




 <?php
      function encode($symb2freq) {
      $heap = new SplPriorityQueue;
       $heap->setExtractFlags(SplPriorityQueue::EXTR_BOTH);
      foreach ($symb2freq as $sym => $wt)
       $heap->insert(array($sym => ''), -$wt);
   while ($heap->count() > 1) {
      $lo = $heap->extract();
```

```
      $hi = $heap->extract();
      foreach ($lo['data'] as &$x)
          $x = '0'.$x;
      foreach ($hi['data'] as &$x)
          $x = '1'.$x;
      $heap->insert($lo['data'] + $hi['data'],
                  $lo['priority'] + $hi['priority']);
  }
   $result = $heap->extract();
   return $result['data'];
}
$txt = 'this is an example for huffman encoding';
$symb2freq = array_count_values(str_split($txt));
$huff = encode($symb2freq);

echo "Symbol\t\t\tWeight\t\t\tHuffman Code<br>\n";
print "<table border = \"10\" cellpadding = \"17\">";
foreach ($huff as $sym => $code)
{
   print "<tr>";
   printf ("<td>%s</td>",$sym);
   printf ("<td>%s</td>",$symb2freq[$sym]);
   printf ("<td>%s</td>",$code);
   print "</tr>";
}
print "</table>";
$characters = str_split($txt);
echo "Huffman encoded string<br>";
printf ("%s=", $txt);
foreach($characters as $char)
{
   echo $huff[$char];
}
      ?>
  </body>
</html>
```

Tabla 3.- Código Fuente en PHP

2.1.1.3. Códigos análogos de Huffman

Se describen a continuación algunos de los algoritmos análogos al Código de Huffman, hay muchas variantes del este algoritmo de Huffman, este se puede utilizar para encontrar el código de prefijo óptimo.

2.1.1.3.1. Algoritmo de Huffman Adaptativo

El método de Huffman, fue originalmente desarrollado por Faller y Gallager, con mejoras sustanciales de Knuth. Este método asume que el compresor conoce las frecuencias de

ocurrencia de todos los símbolos del alfabeto. En la práctica, las frecuencias raramente o casi nunca se conocen de antemano.

Una aproximación a este problema es hacer que el compresor lea los datos originales dos veces. La primera vez, sólo calcula las frecuencias; la segunda, comprime los datos. Entre los dos pasos, el compresor construye el árbol de Huffman.

Este método se llama semi-adaptativo y normalmente es demasiado lento para ser práctico. El método que se utiliza en la práctica se llama codificación de Huffman adaptativa (o dinámica). Se utiliza como base del programa compact de UNIX.

Este método se utilizó para comprimir/descomprimir datos en el protocolo V.32 para módems de 14 400 baudios.

2.1.1.3.2. Algoritmo de Huffman Canónico

Este es un tipo particular de codificación Huffman que tiene la propiedad de poder ser **descrito de una forma muy compacta**. La palabra *canónico* significa que este código en particular ha sido seleccionado de entre varios de los posibles *Códigos de Huffman* debido a que sus propiedades lo hacen más fácil y rápido de usar.

Un *Código Canónico de Huffman* puede ser almacenado de manera más eficiente, muchos compresores de datos comienzan generando un árbol de Huffman *normal*, y lo convierten a un árbol de Huffman *Canónico* antes de utilizarlo. Los Códigos de Huffman Canónicos son útiles cuando el alfabeto es grande y en aquellos casos en que se requiere una decodificación rápida.

Se observa en la tabla siguiente un ejemplo simple de *Código Huffman* canónico, donde se hacen asignaciones a los cuatro primeros símbolos de los códigos 0, 1, 2 y 3 (de 3 bits), y a los dos últimos símbolos, los códigos 2 y 3 (de 2 bits).

000	**100**	**000**
001	**101**	**001**
100	**000**	**010**
101	**001**	**011**
01	**11**	**10**

11	**01**	**11**

Tabla 4.- Ejemplo simple de código Huffman canónico

2.1.1.3.3. Algoritmo PPM *(Prediction by Partial Matching).*

El método de compresión PPM está en la clasificación de métodos estadísticos. Esta es una técnica sofisticada para la compresión de datos que fue desarrollada por J. Cleary y I. Witten, con algunas adecuaciones hechas por A. Moffat.

Este método consiste en un codificador que mantiene un modelo estadístico del texto. Este codificador introduce el símbolo siguiente S, le asigna una probabilidad P y envía S a un codificador aritmético adaptativo, para codificarlo con la probabilidad.

Se describe a continuación el algoritmo de PPM (*Prediction by Partial Matching*):

Algoritmo Compresión mPPM

```
1: Σ ← λ;
2:
3: for all p ∈ T do
4:   if p ∈ Σ then
5:     codifica(PPM_text, p)
6:   else
7:     if σ = 2^16 then
8:       p' ← Σ.lru()
9:       Σ ← Σ − p';
10:    end if
11:
12:    Σ ← Σ ∪ p;
13:
14:    codifica(PPM_text, λ)
15:    for all c ∈ p do
16:      codifica(PPM_voc, c)
17:    end for
18:  end if
19: end for
```

Figura 4.- Algoritmo Compresión PPM

Ahora se describe en este ejemplo un modelo estadístico sencillo que cuenta el número de veces que ha aparecido cada símbolo en el pasado y les asigna a cada uno de ellos una probabilidad basada en eso.

Si se supone que se han introducido y codificado 1217 símbolos hasta el instante actual y que 34 de ellos eran la letra q. Si el símbolo siguiente es una q, se le asigna una probabilidad de 34/1217 y su contador se incrementa en 1.

La próxima vez que aparezca, se le asignará una probabilidad de 35/t, donde t es el número total de símbolos introducidos hasta ese momento (sin incluir la última q) [8].

La desventaja de PPM es que son lentos en la ejecución y requieren mayor memoria para almacenar las estadísticas de los símbolos. El algoritmo PPM obtiene las mejores razones de compresión dentro del grupo de algoritmos de compresión sin pérdida. Su baja velocidad de ejecución y los requerimientos de memoria limitan su uso en la práctica.

2.2. Métodos basados en diccionarios

En las técnicas de compresión basadas en diccionario los símbolos de la fuente o cadenas que se forman con los símbolos de la fuente se representan mediante un índice, es decir un número, que se guarda en un diccionario que se construye a partir de los datos de la fuente. Un diccionario es una lista de símbolos y cadenas de símbolos a los que se les asocia un número índice, puede ser estático o dinámico (adaptativo).

Diccionario estático. En algunas aplicaciones, es suficiente conocer el alfabeto de la fuente y las cadenas relacionadas para armar un diccionario fijo, antes de efectuar la codificación. Este diccionario se usa tanto del lado del transmisor como del receptor. La ventaja de este tipo de diccionarios es la simplicidad. La desventaja es su baja eficiencia de compresión. Además es poco flexible, es decir que una vez diseñado para una aplicación no se puede adaptar con facilidad a otra aplicación.

El ejemplo más simple de “diccionario estático” es un diccionario de inglés utilizado para comprimir texto en español. Imagine un diccionario que contiene, digamos, medio millón de palabras (sin sus definiciones). Una palabra (una cadena de símbolos terminada por un espacio o un signo de puntuación) se lee en la secuencia de entrada y se busca en el diccionario. Si se encuentra una coincidencia, se escribe una entrada de índice para el diccionario en la secuencia de salida.

Un diccionario estático no es una buena opción para un compresor de propósito general. Puede, sin embargo, ser una buena opción para uno que trate un tipo de información en particular.

Diccionario adaptivo. A diferencia del diccionario estático, que está armado por completo de antemano, el diccionario adaptivo no existe antes de iniciar el proceso de codificación y por otra parte una vez que se arma no tiene un tamaño fijo. En realidad puede comenzar con un diccionario vacío o uno pequeño por defecto al comenzar el proceso de codificación y añadirle palabras a medida que aparecen en la secuencia de entrada y eliminar las que han quedado obsoletas, ya que un diccionario extenso retrasa la búsqueda. Dicho método consta de un bucle en el que cada iteración se inicia con la lectura de una cadena de la entrada y la fragmenta (la analiza) en palabras o frases. A continuación, debe buscar en el diccionario cada palabra y, si encuentra una coincidencia, genera un símbolo en la secuencia de la salida. De lo contrario, escribe la palabra sin comprimir y además, la agrega al diccionario. El último paso de cada iteración comprueba si se debe eliminar alguna palabra obsoleta del diccionario. Esto puede sonar complicado, pero tiene dos ventajas:

- Implica operaciones de búsqueda y emparejamiento de cadenas, en vez de cálculos numéricos.

- El decodificador es sencillo (es un método de compresión asimétrico), tiene que leer su secuencia de entrada, determinar si el elemento actual es un símbolo o datos sin comprimir, utilizar los elementos para obtener los datos del diccionario, y ofrecer en la salida los datos finales, sin comprimir. No tiene que analizar la secuencia de entrada de una manera compleja, y no tiene que buscar en el diccionario para encontrar coincidencias.

2.2.1. Lempel-Ziv

Todos los algoritmos de diccionarios adaptivos están basados en los trabajos originales que sobre este tema hicieron Abraham Lempel y Jacobo Ziv en 1977 (LZ77) y 1978 (LZ78).

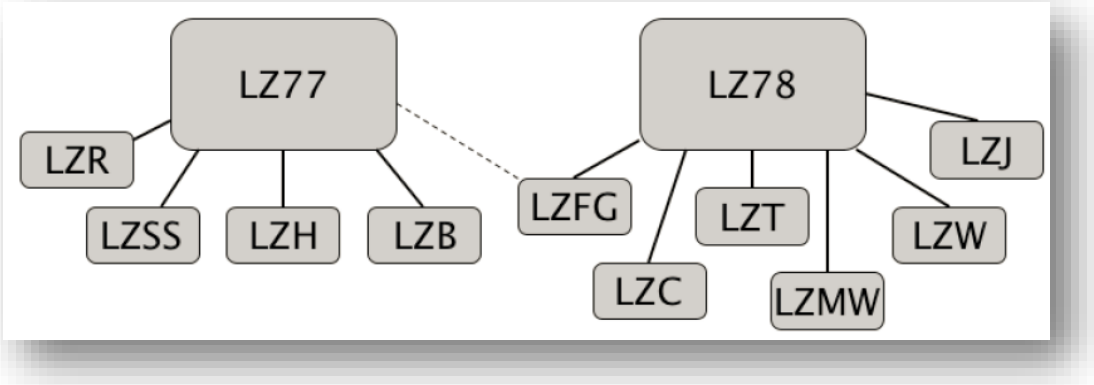

Figura 4.- Familia de Algoritmo Lempel-Ziv

De estos algoritmos se han obtenido numerosos compresores ampliamente utilizados como gzip, pkzip o arj (basados en LZ77) o compress, desarrollado a partir de las propiedades de LZ78. Más recientemente, Igov Pavlov ha planteado una nueva variante del algoritmo LZ77, denominada LZMA, que ha sido integrada en el software de compresión p7zip junto a otras conocidas técnicas como gzip o bzip2

2.2.1.1. Algoritmo LZ77

Representa la raíz de la presente familia de técnicas basadas en diccionario. Su característica principal es el uso de una ventana deslizante de tamaño finito y se divide en dos partes, una ventana de búsqueda (search buffer) y una ventana de texto en avance (lookahead buffer).

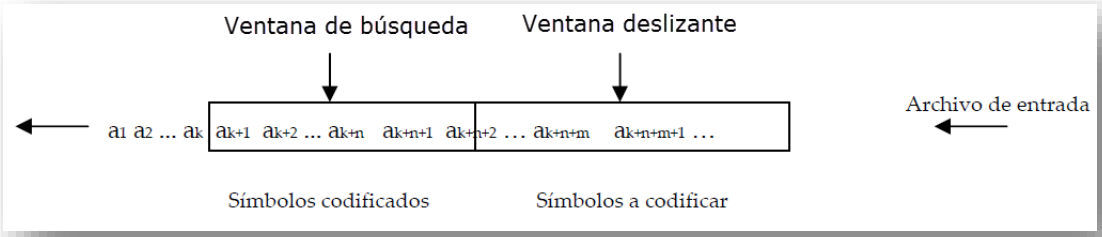

Figura 5.- Diccionario en el método LZ77

Ventana de búsqueda (search buffer). Contiene los caracteres que han sido codificados recientemente y es en sí misma el diccionario.

Ventana en avance (lokahead buffer). Está a continuación de la ventana de búsqueda y contiene los caracteres a ser codificados.

La ventana se desplaza —por eso el nombre de ventana deslizante— a través del texto de entrada, desde el comienzo hasta el final, durante todo el proceso de compresión. La idea es buscar coincidencias entre cadenas de la ventana en avance y la ventana de búsqueda.

A continuación se describe el algoritmo de codificación utilizado:

1.- El codificador busca la subcadena más grande en la ventana de búsqueda (de derecha a izquierda) sea el prefijo de la ventana deslizante.

a.Si tal subcadena existe, la codificación consiste de una tripleta T = (O, L, C), donde O es la distancia en símbolos desde inicio de la subcadena encontrada hasta el final de la ventana de búsqueda, L es la longitud de la subcadena encontrada y C es el símbolo siguiente a la subcadena encontrada en el buffer hacia el frente.

b. Si la cadena no existe, O y L son puestos a cero y C es el primer símbolo del buffer hacia el frente (en la figura anterior, el símbolo ak+n+1).

2.- Ambas ventanas se recorren hacia la izquierda L + 1 posiciones.

```
while (lookAheadBuffer not empty)
 {
get a reference (position, length) to longest match;
if (length > 0)
{
        output (position, length, next symbol);
        shift the window length+1 positions along;
} else {
        output (0, 0, first symbol in the lookahead
        buffer);
        shift the window 1 character along;
        }
}
```

Tabla 5.- Pseudocódigo y algoritmo de codificación del LZ77

A continuación se describe el algoritmo de decodificación:

El decodificador lee cada tripleta T = (O, L, C). Se dan dos casos para L:

a. L es igual a cero:
   i. C se escribe al archivo de salida
   ii. El buffer se recorre a la izquierda una posición y se rellena con C

b. L es diferente de cero:
   i. La subcadena S en el buffer iniciando en la posición O (contando de derecha a izquierda) y de longitud L se copia al archivo de salida.
   ii. C se escribe al archivo de salida
   iii. El buffer se recorre L posiciones a la izquierda, se rellena con la subcadena L.
   iv. El buffer se recorre 1 posición a la izquierda y se rellena con el símbolo C.

2.2.1.2. Algoritmo LZ78

En 1978, Lempel y Ziv presentaron una mejora al algoritmo LZ77 (llamándolo LZ78) eliminando la ventana deslizante. Se tiene un diccionario que contiene las cadenas que han ocurrido previamente. El diccionario está vacío inicialmente y su tamaño esté limitado por la memoria disponible. Para ilustrar la forma en la que el método funciona, considérese un diccionario (arreglo lineal) de N localidades con la capacidad de almacenar una cadena se símbolos en cada una de ellas. . El diccionario se inicializa guardando en la posición cero del diccionario la cadena vacía [11].

Mientras existan símbolos a la entrada

a. S = Null, Pos = 0

b. X = siguiente símbolo de entrada, S = S·X

c. Mientras S exista en el diccionario

i. Pos = Pos(S)

ii. X = siguiente símbolo de entrada, S = S·X

d. Salida: (Pos, X)

e. Guardar S en el diccionario

```
w := NIL;
while (there is input)
{
        K := next symbol from input;
        if (wK exists in the dictionary)
        {
                w := wK;
        } else {
        output (index(w), K);
        add wK to the dictionary;
        w := NIL;
```

```
        }
    }
```

Tabla 6.- Pseudocódigo y algoritmo de codificación LZ78

El proceso es iterativo y termina cuando ya no existen más símbolos a la entrada para codificar. En cada iteración S se inicializa a Null (S = Null indica una cadena vacía que siempre se encuentra en la posición cero del diccionario). El símbolo X del archivo de entrada se lee y se busca la cadena S·X (concatenación de S y X) en el diccionario, si la cadena S·X se encuentra en el diccionario, S es ahora S·X y se lee un nuevo símbolo X.

Nuevamente, se busca S·X en el diccionario y si la cadena se encuentra, se vuelve a leer otro símbolo de entrada y el proceso se repite buscando nuevamente S·X en el diccionario. Si la cadena S·X no se encuentra en el diccionario, se guarda la cadena S·X en una posición disponible en el diccionario y se escribe al archivo de salida la posición de S dentro del diccionario y el símbolo X.

A diferencia del método LZ77, ahora las claves (codewords) se componen solo de dos campos, un campo apuntador o índice que identifica la posición de una cadena dentro del diccionario y un campo que contiene el último símbolo que se ha leído de la entrada. Entre más grande sea el diccionario, más cadenas son almacenadas y se tienen concordancias de cadenas más grandes que mejoran la razón de compresión, pero se requieren apuntadores más grandes y el tiempo de búsqueda se incrementa. La representación del diccionario puede ser una estructura de árbol llamada trie. En este caso, inicialmente, el árbol se compone únicamente de un nodo raíz que representa la cadena vacía. Todas las cadenas que comienzan con la cadena vacía se agregan al árbol como hijos de la raíz. Cada uno de los símbolos que son hijos de la raíz se convierten ahora en la raíz de un subárbol para todas las cadenas que comienzan con ese símbolo.

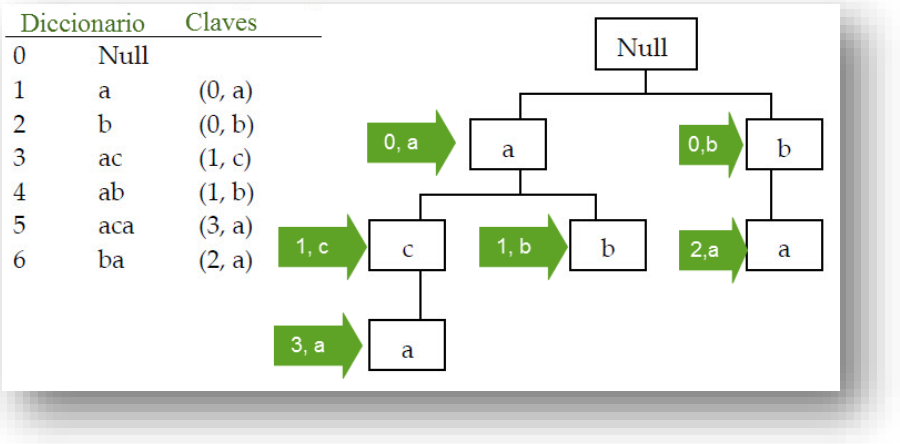

Figura 6.- Diccionario lineal y de árbol LZ78

La representación de árbol tiene dos ventajas principales: la asignación de memoria se realiza conforme nuevos símbolos se agreguen al árbol y la búsqueda de cadenas se realiza más rápido. Cuando el espacio de memoria asignado se agota, las nuevas cadenas ya no se agregan pero el árbol aún sirve para continuar con la codificación. El árbol puede vaciarse por completo y reiniciar ella construcción con los nuevos símbolos de entrada o bien, se pueden eliminar del árbol solo aquellas cadenas que sean menos utilizadas. En este último caso, no existe un buen algoritmo que decida que nodos deben eliminarse y de qué forma debe realizarse tal eliminación. La codificación en LZ78 es más rápida que la realizada en LZ77 pero el decodificador en LZ78 es más complejo que en LZ77. El decodificador debe construir y mantener el diccionario de la misma forma que lo realizó el codificador.

2.2.1.3. Algoritmo LZW

El algoritmo Lemper-Ziv-Welch (LZW) es una variante del algoritmo LZ78 que fue desarrollada por Terry Welch en 1984. Su principal característica es que elimina el segundo campo del índice <i,c>. El algoritmo sólo manda el índice i al diccionario. Este método inicia creando un diccionario con todos los símbolos en su alfabeto. Comúnmente con símbolos de 8 bits, las primeras 256 entradas del diccionario se ocupan antes de que se introduzca cualquier dato. Como el diccionario ya está comenzado, el siguiente carácter siempre se encontrará en el diccionario. La entrada al codificador se acumula en un patrón p mientras p esté contenida en el diccionario. Si la entrada de otra letra “a” resulta en un concatenado p*a, esta no está en el diccionario, entonces el índice de p se transmite al receptor y el patrón p*a se añade al diccionario y empezamos otro patrón con la letra a

2.2.1.4. Ejemplo: Compresión del algoritmo LZW

A continuación utilizamos un ejemplo sencillo para ilustrar el algoritmo

Usando la siguiente secuencia se hace una tabla inicia del alfabeto contenido [9]:

wabbazwabbazwabbazwabbazwoozwoozwoo

Alfabeto Inicial	
Índice	Entrada
1	*z*
2	a
3	b
4	o
5	w

Cuando el algoritmo lee la cadena de texto, primero encuentra la letra w, que ya está en el alfabeto.

Por tanto, concatena la siguiente letra, formando el patrón wa que se agrega al diccionario y ahora el algoritmo lee la letra a, que ya está en el diccionario, por lo que de igual forma se concatena a la siguiente letra y se añade el nuevo patrón ab al diccionario.

Esta acción se repite hasta que se encuentran los patrones únicos de dos letras. Posteriormente, cuando el algoritmo comienza a encontrar repetidos los patrones de 2 letras, les concatena la siguiente letra para crear nuevos patrones de 3 y 4 letras que añade al diccionario hasta que termina de leer la cadena de texto.

El resultado del algoritmo es la secuencia: 5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4 y la siguiente tabla:

Diccionario final			
Índice	Entrada	Índice	Entrada
1	*Z*	14	*azw*
2	*A*	15	*wabb*
3	*B*	16	*baz*
4	*O*	17	*zwa*
5	*W*	18	*abb*
6	*wa*	19	*bazw*
7	*ab*	20	*wo*
8	*bb*	21	*oo*
9	*ba*	22	*az*
10	*az*	23	*zwo*
11	*zw*	24	*ooz*
12	*wab*	25	*zwoo*
13	*bba*		

Descompresión del algoritmo LZW: El decodificador inicia con la tabla del alfabeto inicial y usamos el código que nos arrojó el algoritmo de compresión:

5 2 3 3 2 1 6 8 10 12 9 11 7 16 5 4 4 11 21 23 4

Sabemos que el índice 5 de la tabla es w, por lo que es con lo que iniciamos la descompresión con esa letra. Posteriormente, tenemos el índice 2 que arroja la letra a. Teniendo ambas letras concatenamos para crear una nueva entrada en el diccionario wa que se añade al diccionario.

Al continuar la descompresión y empezar a encontrar concatenaciones de dos letras repetidas comienzan a crearse nuevas al diccionario. Este proceso continua hasta replicarse el diccionario de la compresión que permite interpretar el resultado del algoritmo de compresión.

2.2.2. Código fuente Lempel-Ziv-Welch en python

```
def compress(uncompressed):
    """Compress a string to a list of output symbols."""

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((chr(i), chr(i)) for i in xrange(dict_size))
    # in Python 3: dictionary = {chr(i): chr(i) for i in range(dict_size)}

    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            # Add wc to the dictionary.
            dictionary[wc] = dict_size
            dict_size += 1
            w = c

    # Output the code for w.
    if w:
        result.append(dictionary[w])
    return result


def decompress(compressed):
```

```
    """Decompress a list of output ks to a string."""
    from cStringIO import StringIO

    # Build the dictionary.
    dict_size = 256
    dictionary = dict((chr(i), chr(i)) for i in xrange(dict_size))
    # in Python 3: dictionary = {chr(i): chr(i) for i in range(dict_size)}

    # use StringIO, otherwise this becomes O(N^2)
    # due to string concatenation in a loop
    result = StringIO()
    w = compressed.pop(0)
    result.write(w)
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError('Bad compressed k: %s' % k)
        result.write(entry)

        # Add w+entry[0] to the dictionary.
        dictionary[dict_size] = w + entry[0]
        dict_size += 1

        w = entry
    return result.getvalue()


# How to use:
compressed = compress('TOBEORNOTTOBEORTOBEORNOT')
print (compressed)
decompressed = decompress(compressed)
print (decompressed
```

Output:

```
['T', 'O', 'B', 'E', 'O', 'R', 'N', 'O', 'T', 256, 258, 260, 265, 259, 261, 263]
```

```
TOBEORNOTTOBEORTOBEORNOT
```

2.3 Comparativo de algoritmos de compresión

Kodituwakku et. al. (2010), realizan un estudio en el que comparan el desempeño de varios algoritmos de compresión sin pérdida en archivos de texto. Para ello usan diferentes algoritmos en una serie de textos. Para medir el desempeño miden una serie de para metros para comparar entre algoritmos.

Medidas de desempeño de los algoritmos de compresión: La principal preocupación al usar estos algoritmos es la eficiencia del espacio. Sin embargo, por el tipo de compresión y algoritmos usados una medida de desempeño general es complicada y debería de haber distintas medidas para evaluar el desempeño de esa variedad de algoritmos de compresión.

Medida	**Interpretación**
La ratio o razón de compresión:	Un valor de 0,6, significa que los datos ocupan, tras la compresión, un 60% que su tamaño original. Valores mayores que 1, implican un flujo de compresión métricas mayor que el de entrada (compresión negativa). $$\frac{Tamaño\ Archivo\ Comprimido}{Tamaño\ Archivo\ Original}$$
Factor de compresión	Los valores superiores a 1 indican compresión, y los valores menores que 1 implican expansión. La expresión 100× (1 − Razón de compresión), es también una medida razonable del rendimiento de la compresión. Un valor de 60 significa que la cadena de salida ocupa el 40 % de su tamaño original (o que la compresión ha producido un ahorro del 60 %). $$\frac{Tamaño\ Archivo\ Original}{Tamaño\ Archivo\ Comprimido}$$
Entropía	Este método se usa si el archivo se basa en información estadística y nos permite ver la cantidad de información del archivo comprimido
Eficiencia del código	El tamaño promedio del código es el número promedio de bits que

	nos permiten representar una palabra. La eficiencia del código se calcula como el ratio entre la entropía del archivo original y el tamaño promedio del código.
Tiempo de compresión y descompresión	Permite ver el tiempo que consume el algoritmo en ambos sentidos
Velocidad de compresión	Puede medirse en ciclos por byte (CPB). Este es el número promedio de ciclos de máquina, que se necesita para comprimir un byte.
Error cuadrado medio (MSE o Mean Square Error)	La máxima señal de la proporción de ruido (PSNR o Peak Signal to Noise Ratio), se utilizan para medir la distorsión causada por la compresión con pérdida, de imágenes y películas.

Resultados

LZW: El algoritmo da buenos ratios de compresión. La desventaja es que el tamaño del diccionario incrementa cuando aumentan las entradas que se añaden por el algoritmo. El algoritmo muestra baja eficiencia, ya que se requiere una cantidad considerable de recursos para procesar el diccionario. El ratio de compresión disminuye a medida que el tamaño del archivo aumenta.

Resultados del Código LZW						
Archivo	Tamaño Archivo	Número Caracteres	Tamaño Archivo Comprimido	Razón de Compresión	Tiempo de Compresión	Tiempo de Descompresión
1	22,094	21,090	13,646	61.76	51,906	7,000
2	44,355	43,487	24,938	56.22	167,781	7,297
3	11,252	10,848	7,798	69.30	15,688	3,422
4	15,370	14,468	7,996	52.02	21,484	3,234
5	78,144	74,220	24,204	30.97	279,641	11,547
6	39,494	37,584	21,980	55.65	66,100	5,428
7	118,223	113,863	58,646	49.61	517,739	18,423
8	71,575	68,537	36,278	50.69	187,640	5,611

Adaptative Huffman: El árbol dinámico del algoritmo se tiene que modificar para cada carácter del archivo, por lo que el tiempo de compresión y descompresión se hacen relativamente altos para el algoritmo. En este ejemplo se observa como uno de los

archivos (4) tienen una gran cantidad de palabras que se repiten lo que hace que el ratio de compresión sea de 55%.

Resultados del código de Huffmann Adaptativo						
Archivo	Tamaño Archivo	Número Caracteres	Tamaño Archivo Comprimido	Razón de Compresión	Tiempo de Compresión	Tiempo de Descompresión
1	22,094	21,090	13,432	60.79	80,141	734,469
2	44,355	43,487	26,913	60.68	223,875	1,473,297
3	11,252	10,848	7,215	64.12	30,922	297,625
4	15,370	14,468	8,584	55.85	41,141	406,266
5	78,144	74,220	44,908	57.47	406,938	2,611,891
6	39,494	37,584	22,863	57.89	81,856	1,554,182
7	118,223	113,863	73,512	62.18	526,070	1,271,041
8	180,395	172,891	103,716	57.49	611,908	1,554,182

Huffman Estático: El tiempo de compresión del estático contra el adaptativo es relativamente menor, ya que el estático usa un código fijo. Al ser un algoritmo estadístico, es conveniente usar también las medidas de entropía y eficiencia de código. La entropía del archivo va de 4.3 a 5.1. Lo que quiere decir que para comprimir 1 byte, el algoritmo necesita sólo de 4-5 bits. Asimimo, la eficiencia del código es mayor a 98% en todos los casos, por lo que puede considerarse como eficiente.

Resultados del Código de Huffmann							
Archivo	Tamaño Archivo	Tamaño Archivo Comprimido	Razón de Compresión	Tiempo Compresión	Tiempo Descompresión	Entropía	Eficiencia del código
1	22,094	13,826	62.58	16,141	16,574	4.79	99.28
2	44,355	27,357	61.68	54,719	20,606	4.81	99.36
3	11,252	7,584	67.40	3,766	6,750	5.03	99.44
4	15,370	8,961	58.30	5,906	9,703	4.35	98.67
5	78,144	45,367	58.06	156,844	224,125	4.54	99.07
6	39,494	23,275	58.93	13,044	12,638	4.57	98.91
7	118,223	74,027	62.62	134,281	99,086	4.94	99.27
8	180,395	104,193	57.76	368,720	288,232	4.60	99.44

Referencias

[1] Gary Stix (September 1991). "Profile: Information Theorist David A. Huffman". Scientific American (Nature Publishing Group) 265 (3): pp.54–58. Retrieved July 13, 2011.
[2] Weiss A. Mark; Estructuras de Datos en Java., Addison-Wesley, 2000. ISBN 9788478290352.
[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). Introduction to Algorithms (first ed.). MIT Press and McGraw-Hill. ISBN 978-0-262-03141-7.
[4] Huffman, D., A method for the construction of a minimal redundancy codes, Proc.IRE, 1098-1101, Sept, 1952.
[5] Kuri, A., Notas: Teoría de la Información. Capítulo 1.
[6] Kuri, A., (2004) "Pattern based lossless data compression"
[7] Kuri, A. and Herrera, O., (2003), "Transformación de un mensaje generado por una Fuente No Ergódica a otro generado por una Fuente Ergódica para Compresión de Datos".
[8] Salomon, D. 2007. Data Compression the Complete References Fourth Edition. Springer-Verlag London.
[9] Sayood, K. 2006. Introduction to Data Compression Third Edition. Morgan Kaufmann Series in Multimedia Information and Systems.
[10] Morales, M 2003. Notas sobre compression de datos.
[11] Ziv, J. and Lempel, A., "Compression of Individual Secuences via Variable-Rate Coding", IEEE Transactions on Information Theory, vol. 24, pp. 530-536, 1978.