UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

UNIDAD ACADEMICA DE LOS CICLOS PROFESIONAL
Y DE POSGRADO
DEL COLEGIO DE CIENCIAS Y HUMANIDADES
INSTITUTO DE INVESTIGACIONES EN MATEMATICAS
APLICADAS Y SISTEMAS

ALGORITMOS DE COMPRESION SIN PERDIDA DE INFORMACION PROGRAMADOS BAJO EL PARADIGMA ORIENTADO A OBJETOS

T E S I S

QUE PARA OBTENER EL GRADO DE :

MAESTRO EN CIENCIAS DE LA COMPUTACION

P R E S E N T A :

CARLOS ORDOÑEZ MONDRAGON

DIRECTOR: M. EN C. REFUGIO VALLEJO GUTIERREZ

MEXICO, D. F. DICIEMBRE 1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Agradecimientos:

- A mi madre Guadalupe Mondragón B., y a mi familia por su cariño y apoyo incondicional.

- A mi asesor, el M. en C. Refugio Vallejo Gutiérrez, por haber dirigido esta tesis. Al Ing. Mario Rodríguez Manzanera por sus valiosas observaciones. Al Dr. Luis Morales y al M. en C. Javier García García por haber aceptado ser mis sinodales en el examen de grado. Al Act. Miguel Zamudio por haber hecho una revisión del manuscrito inicial.

- En especial a la Dra. Hanna Oktaba por su invaluable apoyo al programa de la Maestría en Ciencias de la Computación, por aceptar ser mi sinodal en el examen de grado y por animarme a seguir estudiando para obtener un Doctorado.

- A todos mis maestros y amigos de la Maestría.

Carlos Ordóñez Mondragón
México D.F., diciembre de 1995.

Contenido

Capítulo 1

Introducción

1.1 Rama de estudio

La codificación de datos es una rama de la Teoría de la Información [Ing71], que a su vez forma parte del área de Sistemas Digitales de la Ciencia de la Computación. De hecho la compresión que es el tema particular a tratar, puede ser vista como una manera especial de codificar, en la que se trata de lograr de que el espacio ocupado por la información codificada sea mínimo. Es importante hacer notar que los términos compresión, y codificación de datos son usados indistintamente. En todos los métodos de compresión estudiados es posible recobrar los datos originales a partir de su representación codificada, por lo que se les denomina métodos de compresión reversibles [Reg81]. Es menester notar que actualmente compactación es un término usado para describir compresión de datos con pérdida de información, y por lo tanto no reversible, la cual está fuera del alcance de este trabajo.

A lo largo de la presente tesis se estudiarán los algoritmos más representativos para comprimir información, así como adaptaciones necesarias para poder programar tales algoritmos sobre una computadora determinada.

1.2 Aplicaciones

Los usos más comunes de la compresión de datos caen dentro de dos categorías [Lel88]:

- El almacenamiento de datos
- La transmisión de datos

Cuando se emplea compresión para guardar datos siempre se hace referencia al almacenamiento (storage) o memoria secundaria. El almacenamiento secundario de una computadora abarca aquellos dispositivos, generalmente de naturaleza magnética, en los que los datos perduran aunque se interrumpa el suministro de energía eléctrica, en los que el tiempo de acceso es relativamente grande, y cuyos datos almacenados requieren ser transferidos a la memoria principal para poder ser procesados [Dei80]. Como dispositivos de memoria secundaria están: unidades de disco, ya sean fijas o removibles, y cartuchos de cinta. A medida que crece la popularidad de los ambientes gráficos de computación y que los programas se tornan cada vez más poderosos y complejos y por lo tanto requieren más recursos, en especial espacio, se extiende cada vez más el uso de la compresión. De hecho algunos sistemas operativos o sistemas de bases de datos incorporan compresión de manera automática y transparente desde el punto de vista del usuario.

El segundo uso citado con anterioridad cae dentro del campo de las comunicaciones. Al hablar de transmisión o comunicación de datos se hace referencia a enviar y recibir datos entre dos computadoras a través de algún medio físico. Los medios más comunes son cable y ondas. Las redes por su dimensión pueden ser locales (LAN), metropolitanas (MAN), o de cobertura amplia (WAN). Dentro de los cables de uso más extendido está el cable telefónico (par trenzado), cable coaxial y fibra óptica. Como alternativa de enlace se tienen las comunicaciones por ondas, entre las que destacan las microondas y las ondas de radio. Actualmente un ejemplo de red disponible por la que se pueden enlazar computadoras a nivel mundial es Internet que existe desde los 60s que emplea comunicación vía satélite, la red telefónica mundial y la fibra óptica. Tanembaum, en [Tan81], explica con detalle las clasificaciones de redes y los tipos de cable. La fibra óptica es la que permite mayores capacidades y velocidades de transmisión, pero es muy alto su costo. Todos los demás medios de transmisión tienen en general capacidades de volumen menores y anchos de banda también más reducidos. El ancho de banda de un canal de comunicación está definido como la diferencia entre la frecuencia más alta y la frecuencia más baja que se pueden transmitir. La compresión resulta de particular interés para transmitir datos entre computadoras distantes. Aún con la fibra óptica el uso de la compresión es necesario por la popularidad que está tomando el ambiente multimedia en los que el volumen de datos, tales como imágenes y voz, es impresionante. En [Dud95] se describen con detalle los requerimientos de almacenamiento necesarios para transmitir imágenes de video en tiempo real. Por ello, el uso de la compresión permite reducir costos de almacenamiento, transmisión y

equipo de cómputo, aumentando la capacidad de transmisión de un canal de comunicación [Reg81].

1.3 Temas relacionados

La compresión de datos guarda estrecha relación con los siguientes aspectos de la Teoría de la Información:

- la integridad y
- el encriptamiento de datos

El primer aspecto es importante porque hay ciertos métodos de compresión que son muy sensibles a la propagación de errores. Ello se refiere a que si por alguna razón se altera o se pierde parte de los datos codificados, en algunos casos todos los datos siguientes pueden decodificarse incorrectamente o simplemente ya no pueden decodificarse. Por lo anterior es deseable contar con métodos que permitan detectar y/o corregir errores. En la presente tesis se estudia la susceptibilidad al error de una manera sucinta. En general se hace la suposición de que se está trabajando sobre un canal de comunicación libre de ruido (noiseless).

Por otra parte, un efecto colateral de la compresión de datos es el que éstos una vez codificados se vuelven ininteligibles, pues se elimina o al menos se reduce la existencia de patrones, ya que los datos adquieren una nueva representación en la cual la redundancia es baja. Se sabe que esto es aprovechado en ocasiones por intrusos para obtener información cuando interceptan mensajes dentro de una red o husmean información ajena guardada en disco. De lo anterior se desprende que otro uso que se le puede dar a la compresión de datos es mejorar la seguridad que ofrece un sistema de cómputo. El tema de la criptografía se encuentra ampliamente estudiado en [Lem79].

1.4 Tratamiento

A lo largo de este escrito se tratará el tema de la compresión desde el punto de vista de las comunicaciones [Lel88]. De hecho, la mayoría de los trabajos de investigación en la actualidad se dan en este sentido. Los aspectos referentes al hardware [Cap85] se tocarán poco, y más bien el desarrollo se centrará sobre aspectos de programación.

El análisis, diseño y programación orientados a objetos es un conjunto de metodologías de desarrollo de software, que sigue creciendo día con día pues permite lograr dos cosas que no se han podido obtener en varias décadas de desarrollo de software: facilidad de reuso y mantenimiento de programas. Meyer [1988] da un panorama excelente del estado del arte en esta área. Es importante hacer notar que no se pretende ver la clásica programación estructurada y modular como algo obsoleto, sino más bien abordar el problema de desarrollar un programa complejo con una tecnología nueva. De hecho, muchos criterios básicos de programación estructurada fueron respetados para la programación que se efectuó en este trabajo de tesis.

1.5 Objetivos

Las principales metas de esta tesis son:

- Estudiar los algoritmos y métodos de compresión reversible de propósito general más representativos.
- Crear una biblioteca de clases para compresión de datos.
- Comparar experimentalmente ventajas y desventajas de cada método.

En este trabajo se pretende programar algoritmos de uso general. Existen muchos algoritmos que emplean el conocimiento del tipo de información que se va a comprimir, y de estos algoritmos destacan los que se usan dentro del procesamiento digital de imágenes y dentro de las bases de datos. Estos métodos escapan del alcance del presente trabajo porque no se les puede dar otro uso o al efectuar el proceso de compresión se pierde parte de la información original. Los métodos de codificación sobre los que se hará énfasis son:

- Códigos de Huffman
- Códigos Shannon-Fano
- Compresión aritmética
- Códigos Lempel-Ziv (LZ)

No obstante, por su importancia dentro de este tema, se hablará brevemente de la codificación Run-Length y de la compresión diferencial, que son comunes dentro del procesamiento de imágenes.

Así mismo se pretende crear una biblioteca de clases que permita hacer un uso fácil de los algoritmos ya programados, y que por otro lado ayude a darles mantenimiento para hacerlos más eficientes, flexibles, mejores, o libres de posibles errores. Para lo anterior se usará el lenguaje C++ que hoy por hoy es el lenguaje de programación orientada a objetos más popular, aunque en varios aspectos de carácter teórico no sea el mejor. Con lo anterior los programas desarrollados podrán ser utilizados en varias plataformas (o puertos) simplemente recompilándolos, pues existen compiladores de este lenguaje para la mayoría de computadoras en la actualidad. De hecho, actualmente el lenguaje que permite portar programas a distintas computadoras es el lenguaje C, y siendo C++ una extensión del anterior va por la misma senda.

1.6 Ramas de apoyo

Siendo el objetivo de la tesis estudiar los métodos de compresión más representativos y mostrar una aplicación del análisis y diseño orientado a objetos, se usarán conceptos de las siguientes ramas de la Ciencia de la Computación y de las Matemáticas:

- *Análisis de algoritmos.* Rama que reviste particular importancia para estudiar la complejidad que presentan los algoritmos tanto en tiempo como en espacio. Este análisis se describirá de manera breve después de exponer cada algoritmo.

- *Estadística y probabilidad.* El enfoque más extendido para el estudio de la compresión aplica conceptos de estadística y probabilidad. El conocer la probabilidad de aparición de un símbolo, el valor esperado de la longitud de un código y el histograma de frecuencias de mensajes, son algunos de los puntos que son clarificados por estas dos ramas de las matemáticas.

- *Estructuras de datos.* Para efectuar una programación adecuada es necesario emplear conceptos de estructuras de datos. Entre las estructuras que se usarán destacan los diccionarios, las tablas de dispersión (hash), los árboles binarios, y las listas.

- *Análisis numérico.* La generación de códigos aritméticos depende en gran medida de la precisión que tenga el procesador en el que se vayan a efectuar los cálculos. En particular la construcción de estos códigos emplea muchas operaciones en punto flotante, pues el resultado de la compresión no es una sucesión de códigos sino un número real en $[0, 1)$.

1.7 Contenido de la tesis

El capítulo 2 da una introducción a la Teoría de la Información y presenta un modelo abstracto sobre el que se expondrán los algoritmos y se hará el diseño orientado a objetos. Cada uno de los conceptos explicados en este capítulo se usarán a lo largo de todo el texto.

En el capítulo 3 se estudian los algoritmos de compresión más representativos que tienen aplicación general. Estos algoritmos se describen de manera secuencial empleando pseudocódigo. Cada algoritmo se presentará de manera abstracta sin suponer limitaciones inherentes a una computadora en particular. Se hace un pequeño análisis de complejidad temporal por cada uno, y se describen las estructuras de datos necesarias para su instrumentación.

El análisis y diseño orientado a objetos aplicado al problema de la compresión es el tema del capítulo 4. Primero que nada se define la terminología comúnmente usada. Se describen las etapas del análisis y diseño que son necesarios para obtener un modelo de clases, que será la base para el desarrollo del programa en el lenguaje C++. Por último se describen brevemente cada una de las clases del diseño así como las relaciones entre ellas.

En el capítulo 5 se darán detalles sobre la programación. Aquí se plantearán las dificultades que tienen que ser resueltas cuando se programa un algoritmo en una computadora. Entre estos aspectos están el manejo de archivos en disco, las limitaciones del almacenamiento primario, aspectos de eficiencia y consideraciones de portabilidad entre otros.

Dentro del capítulo 6 se muestran resultados experimentales obtenidos al comprimir datos de diversas índoles, tales como imágenes, programas fuente, textos en español, y documentos en inglés entre otros. Se muestran comparaciones en términos del grado de compresión y eficiencia en tiempo. Se anotan ventajas y desventajas sobre cada uno de los métodos de compresión.

Para terminar, en el último capítulo se dan las conclusiones sobre los aspectos estudiados, posibles mejoras al programa, así como los tópicos que todavía son objeto de futura investigación.

Capítulo 2

Teoría de la Información

En este capítulo se explicará cada uno de los términos que serán usados a lo largo del trabajo. Todos los conceptos que se van a definir pertenecen a la Teoría de la Información, a la que se puede ubicar dentro del área de Sistemas Digitales.

La Teoría de la Información puede ser definida como la rama de la Ciencia de la Computación que estudia la manera óptima de codificar datos, para que la transmisión y el almacenamiento de éstos se haga de manera eficiente, y se detecte y/o se minimice la existencia de errores [Ing71]. Por lo tanto es válido decir que la compresión pretende codificar datos de modo que ocupen el espacio mínimo posible.

2.1 Conceptos fundamentales

2.1.1 Definiciones

- *Alfabeto.* Es un conjunto de símbolos. Al que generalmente se le denomina Σ.
- *Bit.* Abreviatura de binary digit. Es un dígito que sólo puede valer 0 ó 1. Los códigos que van a ser estudiados en esta tesis están construidos sobre el alfabeto { 0, 1 }.
- *Palabra.* Uno o más símbolos del mismo alfabeto, no necesariamente distintos, concatenados uno con otro. Es decir, una cadena $\in \Sigma^+$.
- *Mensajes.* Son secuencias de palabras construidas con símbolos pertenecientes a un alfabeto.

- *Tabla de códigos.* Es una función (mapeo) que asocia mensajes sobre un alfabeto con mensajes sobre otro alfabeto. Al primer alfabeto se le llama alfabeto fuente y se le denota con la letra α. Al segundo alfabeto se le llama alfabeto de código y queda identificado con la letra β.

- *Código.* Cada palabra sobre β asociada a un mensaje fuente en particular.

- *Cadena.* Una secuencia de mensajes.

- *Codificación.* El proceso de obtener una cadena de códigos a partir de una cadena fuente.

- *Decodificación.* Es el proceso inverso de la codificación. Es decir, dada una sucesión de códigos obtener la cadena fuente original formada con símbolos del alfabeto de entrada.

- *Codificador.* El algoritmo encargado de codificar datos.

- *Decodificador.* Algoritmo que efectua el proceso inverso al anterior.

Si el lector desea conocer con mayor detalle el tema de la codificación consulte [Ham80]. El alfabeto α de entrada serán todos los símbolos del código ASCII para este trabajo. El alfabeto β sobre el que se van a construir códigos sólo tiene bits. Es decir un código c será una cadena que cumple $c \in (0+1)^+$. Esto tiene una relación directa con la manera en que una computadora representa datos. Lo anterior constituye una razón convincente para utilizar el lenguaje C, que proporciona una gama amplia de operadores de bit, que son característicos de un lenguaje ensamblador.

Es importante mencionar que para programar tres de los algoritmos propuestos más adelante, tales como Shannon-Fano, Huffman y codificación aritmética, una palabra está constituida por un solo símbolo por lo que se usa el término símbolo en vez de palabra en tales casos. Para los métodos que se van a programar la información comprimida será guardada en disco para hacer comparaciones con respecto al grado de compresión. De hecho el mensaje fuente a comprimir será una cadena no nula de símbolos representada por un archivo de entrada de tamaño desconocido y la cadena codificada será guardada en otro archivo de salida.

2.1.2 Clasificación de tablas de códigos

Las tablas de código se distinguen en base a las longitudes, tanto del mensaje fuente como del mensaje destino o código. A los mensajes que son de una longitud fija se les denomina *bloques*, y a los que son de una longitud variable se les llama *variables*.

De lo anterior resultan las siguientes categorías de códigos:

- *bloque-bloque*
- *bloque-variable*
- *variable-bloque*
- *variable-variable*

Los métodos que van a estudiarse utilizan principalmente tablas de código bloque-variable, pues α es generalmente el código ASCII y se construyen códigos de longitud variable sobre $\beta = \{0,1\}$. Los códigos LZ son la excepción, pues pertenecen al esquema variable-bloque.

2.1.3 Características necesarias en los códigos

Sean x, y palabras construidas con el alfabeto de entrada y $c \in (0+1)^+$, un código de bits. f es una función que asocia un código a cada palabra fuente. A los códigos generados por los algoritmos se les van a exigir las siguientes características:

- Que sean distintos. Esto se refiere a que el mapeo sea inyectivo. O equivalentemente que un código esté asociado a una sola palabra de la cadena fuente. Es decir, si $x \neq y \Rightarrow f(x) \neq f(y)$.
- Propiedad de prefijo. Lo cual significa que ningún código sea prefijo propio de otro. Ningún código debe ser subcadena inicial de otro. Si $c = c_1c_2,\ldots c_n$ no existe un código $d = c_1c_2\ldots c_j$ con $1 \leq j < n$. Una tabla de código, cuyos códigos cumplen con la propiedad anterior se dice que es decodificable de manera única.
- Decodificación instantánea. Con esta propiedad se puede decodificar sin necesidad de ver caracteres por adelantado (look ahead). Si en un momento dado ya se encontró un código correspondiente a una palabra codificada se puede hacer la decodificación de manera inmediata, sin tener que analizar los siguientes símbolos.

- Código prefijo minimal. Es aquel que tiene la propiedad de que cada prefijo propio de un código concatenado por la derecha con cada símbolo $\in \beta$ es otro código o el prefijo propio de algún código. Esta propiedad evita generar códigos prefijos innecesariamente largos.

Si el lector desea profundizar en el conocimiento de funciones sobre códigos, funciones inyectivas, suprayectivas y tener una idea mejor de operaciones con cadenas de bits consulte [Gri85] y [Ham80].

2.2 Clasificación de métodos

2.2.1 Criterios generales

Existen varios criterios para clasificar los métodos de compresión. El primero es el considerar las longitudes de los códigos generados. Otro criterio es el caracterizarlos en base al cambio del mapeo de mensajes. Otros autores se fijan en si están basados en propiedades estadísticas o si utilizan un esquema de diccionario. Por último, es posible clasificarlos por su aplicación semántica; es decir, si son de propósito general, o sirven únicamente para codificar determinado tipo de información. Dentro de este último criterio existen inclusive subclasificaciones que consideran características contextuales de la cadena fuente. Bell, Witten y Cleary explican ampliamente estos criterios de clasificación según distintos modelos en [Bel89].

2.2.2 De uso general y de uso especial

Muchos de los algoritmos dinámicos y de los métodos aplicables a un tipo de datos en particular escapan del alcance del presente trabajo. Dentro de los algoritmos que son de propósito especial están los que se aplican dentro del procesamiento digital de imágenes [Gon77], y el procesamiento de señales de voz. Muchos de estos algoritmos comprimen perdiendo o descartando información, ya que las imágenes o la voz pueden ser entendibles por el ser humano aún con cierta pérdida de información o bajo la existencia de ruido. Pero dado que se pretende construir un programa de aplicación general, estos métodos quedan también descartados. Un ejemplo de esto es ver y entender una imagen con una resolución menor a la que fue utilizada para su digitalización.

Existen otros algoritmos, que aunque son de compresión reversible, utilizan un conocimiento del tipo de datos a comprimir y si se aplican a otro

tipo de datos se comportan mal, lo que limita su aplicación. Es importante mencionar que en base a un conocimiento del tipo de datos es posible lograr mejores grados de compresión. Ejemplo de algoritmos de compresión sin pérdida de información de uso particular son la compresión diferencial y la codificación Run-Length, que serán vistos brevemente en el capítulo siguiente.

2.2.3 Esquemas estáticos, dinámicos e híbridos

Considerando los estados por los que pasa el mapeo de mensajes a códigos a lo largo del proceso de codificación-decodificación los métodos pueden clasificarse como:

- *Estáticos*, si el mapeo no cambia nunca durante el proceso de codificación.
- *Dinámicos*, si el mapeo de códigos cambia constantemente.
- *Híbridos*, si el mapeo no es completamente dinámico o estático.

En los esquemas estáticos, el codificador y el decodificador se ponen de acuerdo de antemano cómo va a ser la tabla de códigos que van a emplear. El ya clásico método de Huffman [Huf52] es un ejemplo de esquema estático.

En contraste con los esquemas dinámicos, en los que se cambia el mapeo de manera continua. En este caso el codificador tiene que avisar al decodificador de cada cambio que haga en la tabla, o en su defecto tiene que transmitirla toda. Un esquema dinámico conocido es el propuesto por Welch en [Wel84], que mejoró algunas deficiencias del de Lempel y Ziv [Ziv77].

Dentro de los esquemas híbridos destacan aquellos en los que existen varios mapeos predefinidos dentro de un libro de códigos (codebook), y cada uno se identifica con un número. La idea es que si el codificador decide cambiar el mapeo en un momento determinado sólo le avisa al decodificador cuál es el que decide usar, y no lo transmite todo.

Tres de los métodos que se van a estudiar son estáticos pues tanto las frecuencias como los códigos se generan antes de iniciar el proceso de compresión. La excepción es el método LZW, en el cual el contenido de la tabla de códigos cambia continuamente y no son necesarias las frecuencias de los mensajes. Es decir, el algoritmo LZW es un método dinámico.

Ningún método programado supone de antemano una distribución probabilística de aparición de símbolos o palabras; por lo tanto ninguno de ellos puede ser considerado como híbrido dentro de esta tesis.

2.2.4 Esquemas según longitudes

Como ya se había mencionado, los tipos de métodos que van a estudiarse generan códigos tanto de longitud variable como de longitud fija, y se les clasifica usando el mismo criterio. En general se va a considerar α como el código ASCII y a β el conjunto { 0, 1 }.

2.3 Modelo y métricas de compresión

A continuación se va a fijar un marco conceptual sobre el que se discutirán los métodos de compresión. En [Bel89] se investiga a fondo el tema del modelado para la compresión de datos. Existen dos aspectos fundamentales sobre los que se tiene que prestar atención: el primero es el grado de compresión y el segundo la velocidad. En general si se quiere obtener como resultado datos comprimidos con el menor tamaño posible, el método en general va a ser lento. Por el contrario, si se tiene un algoritmo rápido la compresión obtenida puede no ser óptima. Así mismo es importante hacer notar que por el tiempo de comunicación dentro de un ambiente distribuido de cómputo es preferible la rapidez que obtener el mejor grado de compresión posible; en cambio si se desea hacer un respaldo en cinta, lo mejor casi siempre será reducir al mínimo posible el espacio ocupado por los archivos que se respalden. Lo anterior no quiere decir que el algoritmo sea excesivamente lento para que logre su objetivo, sino que es posible tolerar un pequeño retardo con tal de obtener mejores resultados.

2.3.1 El modelo

Primero que nada se hace la suposición de que tanto los símbolos del alfabeto fuente (α), como los del alfabeto de codificación (β) tienen asociado un costo uniforme. Esto quiere decir que no se asocia ningún costo o penalización a la aparición de determinado símbolo dentro de la cadena fuente. El suponer costo no uniforme constituye un problema complicado estudiado por Karp [1961], y Perl [1975]. Se supondrá también que la probabilidad de aparición de cada símbolo de α es independiente probabilísticamente de la que tienen los demás símbolos. Tal distribución probabilística se supondrá que puede ser conocida siempre, y de hecho para los métodos estáticos que se describirán en el siguiente capítulo, se tendrá que construir un histograma de frecuencias antes de empezar el proceso de codificación. Como alternativa se puede suponer un histograma predeterminado según características de los datos que

se vayan a comprimir; lo anterior con el efecto de ahorrar el tiempo dedicado a calcular las frecuencias de aparición de los mensajes, que es proporcional al tamaño de la entrada.

El proceso de compresión lo va a efectuar el codificador que mandará los códigos a través de un canal de comunicación sin ruido (noiseless). Esta es una suposición importante ya que la detección/corrección de errores es un tema complicado estudiado en [Cap85]. Entre las técnicas usadas para detectar y/o corregir errores están la generación de bits de paridad [Man83], los códigos de Hamming y los polinomios CRC [Ham80]. El flujo de los mensajes fuentes será discreto; es decir, pasará un mensaje a la vez. La aparición de cada mensaje será aleatoria.

2.3.2 Métricas

Para decidir qué método es mejor es necesario poder hacer comparaciones o mediciones de la compresión. Las siguientes medidas de compresión son las más conocidas que se han propuesto a la fecha:

- Porcentaje de compresión
- Razón de compresión
- Entropía y redundancia
- Longitud promedio de mensaje

Para efectos experimentales como medida de comparación entre los distintos métodos se usará la razón o porcentaje de compresión entre la cadena fuente y la cadena destino. Existen muchas métricas; Bell, Witten y Cleary, en [Bel89], proponen por ejemplo usar bits/símbolo.

Porcentaje de compresión

En este tipo de medida la idea es obtener una proporción entre la longitud de la cadena fuente f y la longitud de la cadena codificada c. La medida más común es expresar el grado de compresión como un porcentaje P. En este caso se divide la longitud total de la cadena codificada $l(c)$ entre la longitud total de la cadena fuente de mensajes $l(f)$, que convirtiendo a porcentaje es

$$P = 100 * l(c)/l(f).$$

Equivalentemente se divide la longitud promedio de los códigos entre la longitud promedio de los mensajes fuente. Otra idea similar es ver qué porcentaje representa la cadena fuente respecto de la cadena codificada; $100 * l(f)/l(c)$.

A los datos codificados generalmente se les agrega el tamaño de la tabla de conversión de símbolos (mensajes fuente) a códigos, por lo que esto degradará ligeramente el porcentaje de compresión. En [Bel89] se menciona el porcentaje de reducción P' como una alternativa, que respecto a los porcentajes que se presentarán en el capítulo de resultados experimentales, será la diferencia entre 100 y el porcentaje anotado, i.e.

$$P' = 100 - P.$$

Sin embargo, para efectos experimentales se usará P para comparar los distintos métodos de compresión expuestos en el siguiente capítulo.

Razón de compresión

Para conocer la razón de compresión la idea es obtener una proporción entre la longitud de la cadena fuente $l(f)$ y la longitud de la cadena codificada $l(c)$. Generalmente se calcula como

$$r = l(f)/l(c),$$

y se expresa como una razón $r : 1$, donde $r \in \mathbf{R}$. Por ejemplo si $l(f) = 2l(c)$, entonces la razón de compresión será $2 : 1$.

Entropía y redundancia

El concepto de *redundancia* es de particular importancia dentro de la Teoría de la Información. Para entender qué es la redundancia [Ing71] primero hay que definir algunos conceptos auxiliares. $log_2(x)$ es el logaritmo base 2 del argumento x. Dentro de la computación el sistema binario de numeración es de uso cotidiano; en particular se emplea base 2 para las fórmulas siguientes debido al uso de bits.

La probabilidad de aparición de un símbolo x en nuestra cadena de entrada se denota con:

$$p(x).$$

La *información* que aporta un mensaje x se define como:

$$-log_2(p(x)).$$

Es importante notar que si $p(x) = 0$ la fórmula anterior está indefinida. A esto se le conoce como el problema de la *frecuencia cero* [Bel89]. Por ello, los símbolos que tienen frecuencia nula se ignoran en los métodos de Shannon-Fano, Huffman, y códigos aritméticos. Es decir, no se les asigna código alguno.

La *longitud promedio* de un código para una cierta distribución de probabilidad sobre un conjunto de n mensajes se calcula así:

$$L = \sum_{i=1}^{N} p(x_i)l_i,$$

en donde l_i es la longitud del mensaje x_i y $p(x_i)$ su probabilidad, como ya se había dicho.

La *entropía* H [Ing71], [Sha49] de la fuente queda definida como:

$$H = \sum_{i=1}^{N} -p(x_i)log_2(p(x_i)).$$

Lo que mide esta cantidad H es qué tanta información aporta la cadena que se va a codificar. Es importante notar que esta cantidad determina una cota inferior para el número de bits que son necesarios para codificar una serie de mensajes fuente. En [Bel89] se interpreta como una medida del orden o desorden entre los símbolos.

Se define la *redundancia* R como:

$$R = L - H.$$

De lo anterior se desprende que la redundancia es simplemente una diferencia entre la longitud promedio de un código y la cantidad promedio de información [Lel88]. Si una tabla de códigos tiene la propiedad de que la longitud promedio de sus códigos es mínima, entonces es un código de redundancia mínima. Por lo anterior, si se reduce la longitud promedio de los códigos, también disminuye la redundancia.

Todos los métodos de compresión tratan de disminuir la redundancia, que se manifiesta a través de palabras, patrones o símbolos repetidos. Los métodos que generan códigos prefijos minimales (Shannon-Fano y Huffman) aprovechan las frecuencias de los símbolos, mientras que LZ busca patrones repetidos.

Optimalidad asintótica

Una tabla de códigos se dice que es de optimalidad asintótica si la razón o cociente entre la longitud promedio de los códigos y la entropía tiende a 1 cuando la entropía se hace infinita. Por esta propiedad se garantiza que la longitud promedio de un código puede acercarse al mínimo teórico que determina la entropía. En particular, la compresión aritmética construye los códigos tratando que su longitud promedio sea precisamente la entropía como se verá más adelante.

Capítulo 3

Algoritmos clásicos de compresión

Para describir los algoritmos de compresión se utilizará pseudocódigo. Este pseudocódigo servirá para describir los pasos necesarios para codificar o decodificar una cadena bajo un esquema dado. Se prefirió utilizar una notación abstracta en vez de un lenguaje para tener generalidad y no hacer referencia a tipos de datos, instrucciones o limitaciones pertenecientes a un lenguaje de programación en particular. Así mismo en general no se supone limitación alguna sobre la hipotética computadora en la que se vaya a programar cada algoritmo. Este pseudocódigo es parecido al lenguaje de programación Pascal.

3.1 Complejidad en tiempo

A continuación se definirá notación fundamental con la que se acotará el tiempo de ejecución de los algoritmos para codificar y decodificar, así como de sus rutinas auxiliares. El análisis de algoritmos se encarga de estimar el tiempo y el espacio que requiere la ejecución de un algoritmo [Aho83].

Dentro del presente trabajo se hará énfasis sobre la complejidad temporal de cada algoritmo. La complejidad en espacio se considerará sólo en algunos casos. Gran parte de la investigación que se hace hoy en día está enfocada en encontrar algoritmos más eficientes respecto al tiempo que tardan.

3.1.1 Notación

Cuando se dice que $T(n)$ es $O(f(n))$ se sabe que f(n) es una cota superior para la velocidad de crecimiento de $T(n)$. Matemáticamente se dice que $T(n)$ es de orden a lo más $f(n)$ si existen constantes $C, N > 0$ tales que $|T(n)| \leq Cf(n)$ para toda $n > N$, donde n es el tamaño de la entrada.

3.1.2 Cálculo del tiempo

Sean $T_1(n)$ y $T_2(n)$ los tiempos de ejecución de dos fragmentos de un algoritmo P_1 y P_2, con órdenes de complejidad temporal $O(f(n))$ y $O(g(n))$ respectivamente. Las dos reglas fundamentales para estimar el tiempo de ejecución de un algoritmo son las siguientes:

- **Regla de la suma.** El tiempo de ejecución de $P_1; P_2$, es decir de P_1 seguido de P_2, $T_1(n) + T_2(n)$ es $O(max(f(n), g(n)))$. Esta regla se aplica en las secuencias de pasos para un algoritmo.
- **Regla del producto.** $T_1(n)T_2(n)$ es $O(f(n)g(n))$. Esta regla se usa fundamentalmente para las instrucciones de repetición (ciclos).

3.2 Dependientes semánticamente

3.2.1 Codificación Run-length

Este método de compresión, descrito en [Gon77], se aplica principalmente en imágenes y en bases de datos. En general es útil para secuencias largas de repeticiones del mismo dato. Resulta muy ineficiente en grado de compresión si el número de repeticiones no es muy grande. Tiene como ventaja que es muy fácil de programar y produce excelentes resultados con imágenes binarias por ejemplo. Aunque es de aplicación restringida resulta interesante pues es un primer acercamiento natural para tratar de comprimir datos.

Este algoritmo produce una secuencia de parejas:

$$(m_1, l_1), (m_2, l_2), ...(m_i, l_i),$$

donde m_i es el mensaje fuente, y l_i (length) es el número de sus repeticiones contiguas.

Como se puede notar no cambia la representación de cada mensaje, sino que simplemente le asocia el número de veces que se repite consecutivamente. Para efectos de representación, generalmente se usa aritmética binaria sin

signo para indicar el conteo y los siguientes bits son el código de la letra (mensaje) fuente original. Se trata de guardar el mensaje y el conteo en una representación adecuada para la computadora que se esté usando. Por ejemplo para codificar mensajes que sólo tengan letras mayúsculas y dígitos bastarían 6 bits para guardar el código y 2 bits para guardar el conteo, con lo que se aprovecharía la representación más común para una computadora de un byte.

3.2.2 Compresión diferencial

Este tipo de compresión se utiliza exclusivamente para reducir el tamaño ocupado por imágenes de video, imágenes usadas en animación o imágenes muy parecidas dentro de un canal de comunicación. La idea básica es transmitir una imagen completa solamente cada vez que haya un cambio significativo en la información. Después de efectuada tal transmisión se envían únicamente los cambios que hubo entre la imagen anterior y la actual. Es decir, se transmiten sólo las diferencias (de ahí su nombre) que hay entre una imagen y la siguiente.

Este método de compresión utiliza el hecho de que los contenidos de imagenes tienden a ser muy parecidos para intervalos de tiempo reducidos. Las imágenes de video, por ejemplo, tienden a tener una elevada redundancia local entre una imagen y la siguiente. Esto es intuitivamente natural porque al estar filmando alguna escena en la que un objeto se mueve sólo cambia la porción en la que se registra el movimiento, y todo lo demás permanece igual. Este método de compresión se ha utilizado mucho en la transmisión de imágenes mandadas vía satélite (*Voyager 2*). Como el campo de aplicación de este método es reducido (procesamiento de imágenes) no se entrará en mayor detalle sobre él.

Estos y otros métodos aplicados a reducir el espacio que ocupan las imágenes pueden ser consultados con detalle en [Wil71] y [Cap85].

3.3 Estáticos

3.3.1 Códigos de Huffman

Este es uno de los algoritmos clásicos de compresión, y aunque a la fecha ha sufrido muchas modificaciones para hacerlo más rápido o o lograr que alcance un mayor grado de compresión, ha servido como base para crear otros algoritmos. Su origen se remonta a principios de los 50's cuando fue

inventado por Huffman [Huf52]. Hoy en día su aplicación se ha reducido, pues los de uso más común son los códigos LZ y la codificación aritmética.

La idea del algoritmo es construir códigos prefijos minimales utilizando como herramienta a los árboles binarios. A la letra (símbolo) de mayor frecuencia se le asociará el código más corto, y a la menos probable el código más largo.

Construcción del árbol binario

Primero que nada se determina la probabilidad de aparición de cada símbolo i de la cadena fuente, entendida como un peso, a la que se denota como p_i. Se construyen árboles binarios de un solo nodo asociado a cada letra i con su probabilidad correspondiente p_i. Acto seguido se construye una lista, o bosque con tales árboles de manera tal que sus pesos p_i queden ordenados de menor a mayor.

En cada iteración se construye un árbol binario cuyos hijos son los dos árboles con los pesos más pequeños de la lista y cuya raíz tiene asociada la suma $p_i + p_j$ de los dos pesos correspondientes a sus hijos i y j. Los dos árboles con pesos p_i y p_j respectivamente se borran de la lista y se inserta de manera ordenada el nuevo árbol según la suma $p_i + p_j$. El proceso se detiene cuando la lista se quede con un solo elemento, con peso igual a 1, en cuyo caso queda un solo árbol binario. Si en algún momento el algoritmo tiene dos o más maneras de escoger los dos pesos más pequeños, puede elegir la pareja que sea.

Asignación de códigos

Una vez construido el árbol se asignan los códigos de la siguiente manera: a cada rama izquierda del árbol se le etiqueta con un 0 (cero), y cada rama derecha se etiqueta con un 1 (uno). El código asociado a cada símbolo fuente será el que determine el recorrido desde la raíz hasta cada hoja. Es importante notar que por la construcción del árbol cada nodo interior, y en particular la raíz, tiene asociada una suma de pesos, y ningún símbolo (letra). Por otro lado cada hoja tiene que tener asociada una letra y su peso correspondiente. El asociar a cada rama izquierda del árbol un cero y a la derecha un uno es una convención, pues se puede hacer al revés, y el algoritmo también producirá códigos prefijos minimales.

3.3.2 Códigos Shannon-Fano

Este algoritmo, aunque menos famoso que los códigos de Huffman, produce buenos resultados, y de hecho es asintóticamente óptimo. Es algunos años anterior a los códigos de Huffman y debe su origen a Claude Shannon quien lo describe en [Sha49]. Lo atractivo de este algoritmo es que aplica la técnica de diseño de algoritmos divide y vencerás [Aho83], de una manera natural. Es importante notar que por el tipo de algoritmo utilizado para construir estos códigos el proceso es fácilmente paralelizable.

Asignación de códigos

Primero se debe obtener la lista de probabilidades de aparición de cada mensaje fuente. Esta lista se clasifica en orden decreciente. Se inicializa el código de cada mensaje fuente en vacío. Ahora se itera haciendo lo siguiente: se divide esta lista en dos sublistas de manera que las sumas de probabilidades de cada una sean lo más parecidas posible. A cada mensaje en el primer grupo se le concatena un 0 por la derecha, y cada mensaje del segundo se le concatena un 1 de la misma manera. El proceso se lleva a cabo hasta que cada sublista tenga únicamente un elemento.

Haciendo una analogía con los códigos de Huffman, lo que se hace es construir un árbol binario desde la raíz, mientras que con el algoritmo de Huffman el árbol se construye desde las hojas. Es decir, el método Shannon-Fano construye el árbol binario de arriba hacia abajo (top-down) de una manera recursiva, y el método de Huffman lo efectúa de abajo hacia arriba (bottom-up) terminando cuando tiene una lista unitaria de pesos.

Propiedades de los códigos

Cuando resulta posible dividir cada lista en dos sublistas de igual probabilidad la longitud de cada código es $-log_2(p_i)$, en el peor de los casos la longitud anterior se incrementará en uno. No siempre es posible asegurar que estos códigos producen códigos de longitud óptima, pues existen distribuciones probabilísticas para las que los códigos de Huffman producen mejores resultados. Lo que *sí* es posible garantizar es que los códigos Shannon-Fano son códigos prefijos minimales, por la manera en la que son construidos.

3.3.3 Algoritmos para códigos prefijos minimales

Como ya se mencionó tanto el método de Huffman como el método Shannon-Fano producen códigos prefijos minimales, aunque no necesariamente los mismos. Por lo anterior los procesos de codificación y decodificación son los mismos para ambos métodos y son relativamente simples.

En los párrafos anteriores se describió como generar códigos dada una distribución probabilística para los símbolos de una cadena fuente. Ahora bien, una vez generado el mapeo de símbolos a códigos el proceso de codificación y decodificación serán los siguientes. Hay que notar que el mapeo se construye como una función biyectiva entre símbolos y códigos binarios. Dicho de otro modo a cada símbolo de la cadena fuente le corresponde un solo código. Y cada código construido es el código correspondiente de únicamente un símbolo. Si lo anterior no estuviera garantizado el proceso de decodificación sería imposible pues podría presentarse una situación de ambigüedad.

Sean $x_1, x_2, \ldots x_n$ la cadena fuente, y *cadena* la cadena codificada en bits respectivamente. $+$ es el operador de concatenación para cadenas. El identificador *codigo* es la tabla de mapeo de símbolos a códigos construida con alguno de los dos métodos anteriores. La salida del algoritmo será *cadena*, una sucesión de bits, es decir $cadena \in (0+1)^+$.

Codificación

1. $cadena := \varepsilon$
2. **Para** $i := 1$ **hasta** n **hacer:**
   $cadena := cadena + codigo[x_i]$

Para efectos de decodificación sea $b_1, b_2, \ldots b_{bits}$ la cadena codificada en bits por medio del método de Huffman o Shannon-Fano, donde *bits* es el número total de bits. i es el índice del bit analizado en cada iteración. x_j indica el símbolo decodificado. *simbolo* es una función que retorna el símbolo correspondiente a una *cadena* de bits; en caso de que no exista tal cadena de bits el valor resultante será una cadena nula. La salida del algoritmo es $x_1, x_2, \ldots x_{j-1}$, una sucesión de símbolos.

Decodificación

1. *cadena* := b_1, $j := 1$

2. Para $i := 2$ hasta *bits* hacer:
   Buscar *cadena* en la tabla de códigos
   Si *cadena* se encontró entonces:
   - x_j := *simbolo*(*cadena*)
   - $j := j + 1$
   - *cadena* := b_i
   En caso contrario:
   - *cadena* := *cadena* + b_i

Complejidad en tiempo

Aunque los algoritmos requieren dar dos pasadas sobre la cadena fuente, el mapeo de n mensajes se genera en un tiempo lineal, $O(n)$. Una primera pasada es necesaria para determinar la distribución probabilística con la que se construye el árbol binario, y la segunda para efectuar la codificación. Si se quiere ahorrar tiempo para codificar pueden existir mapeos predefinidos, que ya suponen cierta distribución probabilística para los mensajes de la cadena fuente. Claro que ello puede producir una codificación no óptima.

El algoritmo de codificación tiene un solo ciclo que se efectua tanta veces como símbolos tenga la cadena fuente, en un tiempo $O(n)$. El algoritmo correspondiente para descomprimir contiene un ciclo que se efectua tantas veces como *bits* haya en la cadena codificada. Es importante notar que aunque este último proceso es lineal respecto al número de bits se incurre en cierto gasto de tiempo (overhead) en hacer la búsqueda de cada código. Se decidió emplear una búsqueda binaria dado que los símbolos están ordenados por peso, por lo que el tiempo es $O(nlog_2(N))$, donde n es el tamaño de la cadena de entrada en bits, y N el número de símbolos que tiene asociado una tabla de códigos. Ello se explicará con mayor detalle cuando se hable de la programación.

Por último es importante notar que se está despreciando el tiempo que se lleva generar los códigos prefijos tanto en la codificación como en la decodificación, pues se hace sólo una vez al principio de la codificación y no se hace en la decodificación, y además el número de códigos generados valdrá como máximo el número de símbolos distintos N.

Susceptibilidad al error

Los códigos prefijos minimales tienen la propiedad de que un error no se propaga por mucho tiempo. O sea, en caso de que haya un error de fase o de amplitud el codificador y el decodificador se resincronizan casi inmediatamente; aunque ello no puede garantizarse siempre.

3.3.4 Compresión aritmética

A la compresión aritmética se le conoce comúnmente también como codificación aritmética. Elias es el autor original del método, descrito en [Eli75], posteriormente depurado por Rissanen [Ris76], [Ris83].

La compresión aritmética trabaja con una idea muy distinta a la de los métodos anteriores. La cadena fuente se representa como un subintervalo del intervalo $[0, 1)$. La idea es ir achicando el intervalo cada vez que se analice un símbolo, y entre más pequeño sea un intervalo más bits serán necesarios para representarlo. Al igual que con los códigos Shannon-Fano y los códigos de Huffman se debe tener determinada una distribución probabilística de los símbolos de la cadena de entrada. Un mensaje de alta probabilidad achica menos el intervalo que otro que tenga baja probabilidad. De lo anterior resulta que un mensaje de alta probabilidad usa pocos bits, y uno de baja probabilidad usa muchos bits. El algoritmo comienza con una lista de mensajes de una cadena fuente, en la que cada mensaje tiene asociada una probabilidad. Luego se recorre la cadena fuente de principio a fin haciendo una partición sobre el intervalo con base en el criterio anterior.

Achicamiento del intervalo

Como primer paso tiene que determinarse la probabilidad de aparición de cada símbolo. Se efectua un proceso de achicamiento del intervalo usando las siguientes ecuaciones:

1. izqnuevo = izqanterior + mensajeizq * tamañoanterior

2. tamañonuevo = tamañoanterior * longmensaje

La primera ecuación calcula el nuevo punto izquierdo a partir del intervalo anterior y el símbolo actual. El punto izquierdo del rango (probabilidad acumulada) del mensaje actual indica qué tanto hay que quitarle por la izquierda al intervalo actual.

Codificación

El tamaño del subintervalo final s determina el número de bits que se necesitan para representar un número dentro de él. Dado que cualquier probabilidad no nula y estrictamente menor a 1 tiene un logaritmo negativo, la cantidad de bits necesaria para representar a un subintervalo s será:

$$-log_2(ancho),$$

donde *ancho* es el tamaño del subintervalo s.

Por la construcción del algoritmo resulta que:

$$ancho = \prod_{i=1}^{n} p(x_i),$$

donde n es el número total de mensajes contenidos en la cadena fuente.

Ahora bien aplicando logaritmo base 2 en ambos lados de la ecuación, y utilizando el hecho de que el logaritmo de un producto es la suma de los logaritmos de los factores se cumple la siguiente identidad:

$$-log_2(s) = -\sum_{i=1}^{n} log_2(p(x_i)) = -\sum_{i=1}^{N} p(a_i) log_2(p(a_i));$$

en esta fórmula x_i representa el mensaje fuente i, y a_i el mensaje fuente único número i. n es el número total de mensajes contenidos en la cadena fuente, mientras que N es el número de mensajes distintos. En la implantación de este algoritmo $N = 256$, el número de símbolos del código ASCII. Como se puede ver el número de bits generados por la codificación aritmética es igual a la entropía H definida anteriormente.

Decodificación

Para poder obtener la cadena original el decodificador debe conocer la tabla de probabilidades y rangos de los símbolos de entrada. Como entrada se le da un número contenido en el intervalo que determinó el codificador.

La decodificación la hará el decodificador simulando el comportamiento del codificador. Para lo anterior se fija en qué intervalo se encuentra el número de entrada. Con ello determina el primer símbolo decodificado. A continuación determina subintervalos a partir del intervalo encontrado en la iteración anterior dividiéndolo según la tabla de probabilidades. Al fijarse a qué subintervalo pertenece el número de entrada determina el segundo mensaje decodificado. El proceso anterior se repite hasta que se llegue al final de la cadena decodificada.

Desventajas del método

Existen tres problemas fundamentales [Lel88]:

- La precisión aritmética de la computadora
- Detectar el fin de la cadena decodificada
- La susceptibilidad al error

Resulta evidente que es necesario tener una computadora con precisión aritmética infinita para poder codificar cadenas de longitud ilimitada, lo cual es imposible en la práctica. Cualquier computadora incorpora facilidades para manipular cantidades en punto flotante. El formato de números en punto flotante, para una computadora determinada, está estandarizado y utiliza registros de longitud fija. La compresión aritmética puede programarse sin mayores problemas, siempre y cuando se contemplen las posibilidades de desbordamiento y de anulamiento (overflow y underflow). Con desbordamiento se hace referencia al hecho de que una cantidad no quepa de acuerdo al formato existente. Por anulamiento se entiende que una cantidad muy pequeña se guarda como cero por la imposibilidad de poder ser representada en un registro de longitud fija. Como puede verse ambos casos son problemas de rango análogos.

Para solucionar el problema de saber cuándo acaba la cadena existen dos alternativas. La primera es utilizar un símbolo especial no perteneciente a α para marcar el fin de la cadena. Lo cual permite hacer la decodificación en un paso y hacer variaciones para obtener algoritmos adaptativos. La segunda es enviar o guardar la longitud de la cadena codificada, es decir, guardar la cuenta de códigos que la componen. En ambas soluciones es necesario agregar información adicional a la cadena codificada de cualquier manera.

En cuanto a la tercera desventaja, un bit invertido o perdido provoca que toda la cadena sea decodificada incorrectamente desde ese momento hasta su fin; el codificador y el decodificador no vuelven a sincronizarse.

Algoritmo mejorado

Teniendo en cuenta las deficiencias expuestas anteriormente, a continuación se presentan los algoritmos de codificación y decodificación propuestos por Rubin [Rub79]. Las características más relevantes de los algoritmos son las siguientes:

- Se utiliza sólo aritmética *entera*, guardando la fracción como un cociente entre enteros. Para ello se guarda la parte significativa de los enteros que se van actualizando. En general las operaciones hechas con números enteros son más rápidas que las hechas en punto flotante aún cuando éstas se hagan en un coprocesador aritmético. Lo anterior se debe a que los enteros guardan una relación estrecha con los registros que tiene el procesador de la máquina con la que se esté trabajando. Otro aspecto importante es que la complejidad en tiempo del algoritmo logra mantenerse lineal, y no cuadrática que sería la que tendría el algoritmo si se usan técnicas de precisión múltiple.

- En vez de guardar la longitud de la cadena a codificar, para saber cuando termina, se agrega un símbolo especial denominado Fin De Cadena (FDC). Es importante hacer notar que este símbolo no pertenece al conjunto de símbolos a codificar; es decir, el algoritmo se asegura de que sea distinto por su construcci'on. De esta manera cuando se encuentre el símbolo FDC durante la descompresión automáticamente se detiene el proceso de decodificación.

- Se detecta tanto la situación de desbordamiento (overflow) como la situación de anulamiento (underflow) cuando los pasos del algoritmo conducen a tales situaciones. Ello evita que se decodifique incorrectamente un símbolo por otro debido a un error de precisión.

Definiciones

Cadena a codificar: $x_1, x_2, \ldots x_n$, símbolos de entrada: $1, 2, \ldots N$, pertenecientes a algún alfabeto Σ, probabilidades de cada símbolo: $P(i) > 0$. Probabilidades acumuladas: $Q(1) = 0, Q(i+1) = Q(i) + P(i)$ con $1 \leq i \leq N$. salida: una cadena s de bits, i.e. $s \in (0+1)^+$.

Teniendo en cuenta que se usa aritmética entera sobre alguna máquina digital se definen los siguientes números. Sea b el número de bits que tiene el entero que vayan a emplearse, en base a B y considerando que no se usa el signo, el máximo entero representable será $B = 2^b - 1$. Consultar [Man83] para entender cómo se obtiene B. $P(i) = p(i)/D$, y $Q(i) = q(i)/D$ donde $p(i)$ y $q(i)$ son enteros representables en b bits. D se define como $\Sigma_{i=1}^{N} p(i)$. Por último sean f y m dos enteros representables en b bits. f representa el extremo izquierdo del intervalo, tal como se explicó en la subsección anterior. Por otro lado m representa el ancho del intervalo, que originalmente se

inicializa en 1, pero como se utiliza aritmética entera se inicializará con B, como se verá más adelante.

Dos funciones que emplean los dos algoritmos son las dos siguientes: $H(x)$ es una función que devuelve el bit más significativo (highest) del entero x, y $L(x)$ se define como $L(x) = x - 2^{b-1}H(x)$, que entrega los $b-1$ bits menos significativos (lowest) de x.

Modificaciones. En base a las definiciones anteriores se presentan los algoritmos correspondientes para comprimir y descomprimir. Se hicieron dos modificaciones importantes a los algoritmos presentados en [Rub79]. Las dos modificaciones son:

- Dado que la multiplicación en una computadora no es asociativa por los problemas de desbordamiento o anulamiento se cambió el orden de las operaciones por medio de las cuales se obtiene f y m.
- Considerando que cualquier procesador permite manipular enteros sin signo se modificaron los algoritmos para que se aproveche totalmente el rango disponible del procesador. Es importante notar que el algoritmo propuesto en el artículo de referencia supone la existencia de $b-1$ dígitos de precisión debido a que las computadoras utilizan el 1^{er} bit de un entero para guardar el signo.

Algoritmo de codificación

Entrada: $x_1, x_2, \ldots x_n$, n símbolos pertenecientes al alfabeto de entrada.
Salida: $b_1, b_2 \ldots b_j$, una sucesión de bits, es decir una cadena en $(0+1)^+$.

1. $f := 0, m := B, j := 0$
2. Para $i := 1$ hasta n hacer:
   $f := f + m/D * q(x_i)$
   $m := m/D * p(x_i)$
   Ajustar a f, m
   Si $m < m'$ entonces: $m := L(B - f)$, Ajustar a f, m
3. Para $i := 1$ hasta b hacer:
   Si $H(f) \neq H(f + m - 1)$ entonces: $j := j + 1, b_j := H(f) + 1$,Parar
   $j := j + 1, b_j := H(f)$
   $f := 2 * L(f)$
   $m := 2 * L(m) + 1$

La rutina para ajustar a f, m es muy parecida al paso 3:

- Mientras $H(f) = H(f + m)$ hacer:
  $j := j + 1, b_j := H(f)$
  $f := 2 * L(f)$
  $m := 2 * L(m) + 1$

Algoritmo para decodificar

Entrada: $s_1, s_2, \ldots s_n$, una sucesión de bits. Salida: $x_1, x_2, \ldots x_i$, una sucesión de símbolos decodificados.

1. $f := 0, m := B, k := b, s := s_1 s_2 \ldots s_k, i := 0$

2. Repetir los pasos siguientes:
   Encontrar la máxima j tal que $f + m/D * q(j) \leq s$
   Si (j=FDC) entonces: Parar
   $i := i + 1, x_i := j$, (j fue el símbolo decodificado.)
   $f := f + m/D * q(j)$
   $m := m/D * p(j)$
   Ajustar f, m, s
   Si $m < m'$ entonces: $m := L(B - f)$, ajustar f, m, s

La rutina para ajustar f, m, s es la siguiente:

- Mientras $H(f) = H(f + m)$ hacer:
  $k := k + 1$
  $s := 2 * L(s) + s_k$
  $f := 2 * L(f)$
  $m := 2 * L(m) + 1$

Complejidad en tiempo

El tiempo requerido para la codificación es proporcional al número n de caracteres a codificar. A esto hay que multiplicarlo por el número promedio de veces que se checa la precisión que es aproximadamente $log_2(N)$, donde N es

el número de símbolos lo que resulta en un tiempo $O(nlog_2(N))$. El decodificador emula el proceso del codificador por lo que el tiempo de ejecución es similar. Es importante mencionar que la búsqueda empleada es binaria, por lo que es válido decir que el tiempo será también $O(nlog_2(N))$. Si se emplea una búsqueda secuencial entonces el tiempo será $O(nN)$, que para una N grande (256 para el código ASCII), dará por resultado una descompresión ineficiente. En [Rub79] es posible encontrar un análisis más profundo del tiempo de ejecución.

3.4 Dinámicos

Entre los métodos dinámicos más importantes están los siguientes:

- LZW, que emplea como estructura de datos un diccionario.
- FGK, que es una variación de los códigos de Huffman.
- V, que presenta mejoras sobre FGK.
- BSTW, que utiliza listas auto-organizadas.

De los algoritmos anteriores el más popular es el primero, y es el que se explicará con detalle a continuación. Los demás no presentan una mejora considerable sobre los códigos de Huffman o Shannon-Fano y son de más interés teórico que práctico.

3.4.1 Método LZW

Este método fue por primera vez propuesto por Ziv y Lempel [Ziv77], y por ello a los códigos generados por este método se les conoce como LZ. Se le han hecho más de 10 variaciones, de las cuales destaca la propuesta por Welch [Wel84]. Como esta última es la más conocida al método se le conoce como LZW, por las iniciales de los tres autores.

El método LZW es el más popular comercialmente, y de hecho los programas de compresión más populares en ambiente UNIX o MS-DOS lo usan como base. Este método a diferencia de los anteriores, determina de manera dinámica el mapeo de los mensajes fuentes a palabras código.

El algoritmo especifica dos cosas:

1. Cómo obtener subcadenas de símbolos a partir de la cadena de símbolos, que tengan una longitud acotada.

2. Un esquema de codificación que asigna un código único de longitud fija a cada subcadena obtenida en el paso anterior.

Las cadenas se seleccionan de manera que tengan una probabilidad de ocurrencia parecida. De lo anterior resulta que los símbolos que tienen mayor probabilidad de ocurrencia quedan agrupados en cadenas más largas y los menos frecuentes quedan inmersos en cadenas más cortas [Lel88].

Con esta estrategia se aprovechan tres cosas:

- La probabilidad de ocurrencia de cada símbolo
- Repeticiones de una cadena
- Detectar patrones

Este método funciona incrementalmente aprendiendo de la cadena fuente o adaptándose a ella a medida que la va leyendo; de ahí que a este algoritmo se le considere como adaptativo (adaptive). Hay que observar también que como en este método se aprovechan las repeticiones, el primer algoritmo que se explicó en este capítulo (Run-Length) queda incluido como un caso particular de éste [Lel88].

A diferencia de los métodos anteriores con LZW

- No se utiliza la probabilidad de aparición de cada símbolo
- Se aprovechan patrones repetidos
- Requiere más espacio en memoria

Codificación

El algoritmo va analizando el mensaje fuente separándolo en partes de longitud creciente. Al tratar de codificar el algoritmo intenta encontrar la cadena prefija de longitud mayor que coincida con alguna entrada existente en el diccionario. Una vez que encuentra tal entrada le agrega el siguiente símbolo de la cadena fuente, y esta nueva cadena se agrega como una nueva entrada de la tabla. Esta entrada queda codificada como una pareja que tiene como primer elemento el número de entrada de la cadena encontrada y como segundo elemento el siguiente caracter de la entrada.

Este algoritmo de codificación cae dentro de la categoría de algoritmos glotones (greedy), pues busca simplemente el patrón concordante más largo. El método requiere que los códigos sean de una longitud fija, y por lo tanto

el número de entradas de la tabla también es fijo. Por lo anterior resulta evidente que estos códigos tienden a ser muy ineficientes para una cadena fuente muy pequeña. Pero tienden a ser asintóticamente óptimos para cadenas de longitud cada vez mayor. Es decir este método llega a producir códigos óptimos a medida que la longitud de la cadena tiende a infinito.

Otro aspecto importante es que si la longitud de los códigos de la tabla no son suficientemente largos puede suceder alguna de las siguientes cosas:

- La tabla puede llenarse antes de haber terminado de codificar la cadena fuente.
- Volverse ineficiente, o alternar entre eficiente e ineficiente si el algoritmo no puede adaptarse continuamente.

Decodificación

Para volver a obtener la cadena original simplemente se recorre la tabla generada por la codificación desde la primera entrada viendo a qué cadena prefija y a qué símbolo corresponde cada código y de esa manera es posible decodificar cada entrada subsecuente. Dado que el acceso a cada entrada se hace de manera secuencial, la decodificación se hace en un tiempo lineal.

Complejidad

El algoritmo de codificación anteriormente expuesto tiene una complejidad en tiempo de $O(n^2)$. Donde n es por supuesto la longitud de la cadena de entrada.

Dado que podría pensarse que el algoritmo original es ineficiente existen dos variaciones:

1. Existe una variación que emplea árboles sufijos y que tiene complejidad lineal ($O(n)$). El problema es que los requerimientos de memoria son altos y el algoritmo se comporta mal con entradas de longitud pequeña.
2. La otra idea consiste en construir una tabla de 2^B entradas, donde B es el número de bits de cada entrada de la tabla. Para acceder a cada elemento de la tabla se usan técnicas de dispersión (hashing). La decodificación se hace de manera recursiva desde el último caracter hasta acabar en la primera entrada, por lo que es necesario auxiliarse de una pila para invertir el orden de los caracteres. Los algoritmos que se expondrán más adelante usarán esta variante.

Desventajas

Como ya se mencionó el método no produce resultados buenos para una cadena de entrada de longitud pequeña. Otro problema es que tiene que determinarse por adelantado la longitud de cada código, lo que fija el tamaño de la tabla consecuentemente. El método general tiene una complejidad cuadrática en tiempo, aunque puede mejorarse modificando el acceso para obtener tiempos lineales; el problema es que aumentan los requerimientos de memoria, o el algoritmo se vuelve todavía más ineficiente para cadenas de longitud reducida. Por último, dado que los códigos LZ son de tipo variable-bloque resultan muy sensibles a la propagación de errores, sobre todo de fase.

Algoritmos para códigos LZ mejorados por Welch

A continuación se listan los algoritmos de compresión y descompresión en pseudocódigo. Estos algoritmos fueron tomados de [Wel84].

Definiciones. c es un símbolo del alfabeto de entrada, es decir $c \in \Sigma$. s es una cadena formada con el alfabeto de entrada, o sea $s \in \Sigma^*$. *tabla* guarda los códigos construidos por el algoritmo para cada cadena s generada. *tabla* se inicializa con cada símbolo de Σ. $+$ es el operador de concatenación de cadenas. n es la longitud de la cadena de entrada.

Algoritmo de compresión

Entrada: $x_1, x_2, \ldots x_n$. Salida: $c_1, c_2, \ldots c_j$, una sucesión de códigos. La función *codigo*(s) devuelve el código correspondiente a la cadena s.

1. $s := x_1, j := 0$

2. Para $i := 2$ hasta n hacer:
   $c := x_i$
   Si $s + c \in$ *tabla* entonces:
   - $s := s + c$
   En caso contrario:
   - $j := j + 1, c_j :=$ *codigo*(s)
   - agregar $s + c$ a *tabla*
   - $s := c$

3. $j := j + 1, c_j := codigo(s)$

Algoritmo de descompresión

Se respetan las definiciones anteriores adicionando a $c_{anterior}$ que representa al código anterior, necesario dentro del proceso de decodificación. Se supone que la cadena codificada contiene al menos un código. $decodifica(c_i)$ devuelve la cadena de símbolos correspondiente al código c_i. c es un símbolo del alfabeto de entrada.

Entrada: una sucesión de códigos $c_1, c_2, \ldots c_n$, n códigos. Salida: una sucesión de símbolos sobre el alfabeto de entrada: $simbolos = x_1, x_2, \ldots x_j$.

1. $c_{anterior} := c_1$, $simbolos := decodifica(c_0)$
2. Para $i := 2$ hasta n hacer:
   Si $c_i \notin tabla$ entonces:
   - $s := decodifica(c_{anterior})$
   - $s := s + c$

   En caso contrario:
   - $s := decodifica(c_i)$

   $simbolos := simbolos + s$, (Emitir s)
   $c := s[1]$
   Agregar $decodifica(c_{anterior}) + c$ a $tabla$
   $c_{anterior} := c_i$

Capítulo 4

Biblioteca de clases

Actualmente la metodología más aceptada para hacer análisis y diseño orientado a objetos es el Método de Booch [Boo91]. Aunque existen otras metodologías populares, tales como la metodología OMT (Object Modeling Technique) de Rumbaugh, en el presente trabajo se hará énfasis sobre la primera.

El objetivo es aplicar una metodología para crear clases que faciliten el mantenimiento y la reusabilidad de los métodos de compresión ya programados. Es importante notar que la programación orientada a objetos persigue otras metas, pero en este capítulo se hará énfasis sobre los dos primeros aspectos mencionados, que de hecho son dos de los problemas más difíciles de solucionar que se han presentado en la ya clásica programación estructurada, la cual se ha venido aplicando en los grandes desarrollos de programas durante las últimas décadas.

4.1 Introducción

4.1.1 Breve historia de la POO

La programación orientada a objetos aunque ha crecido en popularidad recientemente ya tiene mucho tiempo de existir. El precursor de los lenguajes orientados a objetos fue SIMULA, un lenguaje creado en Noruega para desarrollar programas de simulación [Kat94]. Este lenguaje fue creado en 1967 como una extensión al lenguaje Algol, que era muy popular entre la comunidad académica por aquella época.

La programación orientada a objetos (POO) representa un paso similar al que se dió en los 70's con la programación estructurada. La POO

pretende resolver ciertos problemas que no podían ser resueltos con la programación estructurada. Ciertamente toda la teoría existente creada a partir de la programación modular dio una base para que puedan aprovecharse las características más sobresalientes de la POO.

No hay que perder de vista que la POO es otra forma de hacer programación imperativa o procedural. Es decir, se escriben programas en los que se define paso a paso lo que se quiere lograr. Esta característica es fundamental para no confundir un lenguaje orientado a objetos (LOO) con un lenguaje declarativo en el cual se especifica lo que se quiere lograr, y no cómo hacerlo, por ejemplo Prolog.

4.1.2 Modelo de datos

La POO modela la realidad viéndola como una colección de objetos que interactúan entre sí. Esta interacción entre objetos se da mediante una comunicación a través del paso de mensajes. Los objetos se clasifican en categorías llamadas clases según sus características y comportamiento. En la siguiente sección se explicarán con detalle estos conceptos.

Por el hecho de que este modelo ve al mundo real como objetos que se comunican entre sí enviándose mensajes surge una analogía fuerte con los sistemas distribuidos. La concurrencia es una característica esencial de este tipo de sistemas, que es ortogonal a los conceptos que van a verse, es decir, no produce contradicciones; por lo que puede incorporarse naturalmente a un diseño cuando sea necesario.

4.1.3 Objetivos

Dentro del paradigma orientado a objetos se persiguen tres objetivos fundamentales:

- **Abstracción.** Poder concentrarse en el problema, más que en detalles. Acercar el uso de la computadora al problema, y no al revés.
- **Robustez.** Que un programa pueda funcionar bien en la mayoría de los casos, y pueda recuperarse de situaciones inesperadas. Ello se logra a través de ciertas restricciones que tratan de mantener la integridad de los objetos.
- **Reusabilidad.** Facilitar el uso de código que ya se había programado y que requiere sólo pequeños cambios. Poder especializar (extender) definiciones de clases existentes de manera flexible.

Es necesario notar que estos objetivos son buscados desde hace tiempo y existen mecanismos para lograrlos en lenguajes que no caen dentro de este paradigma. Lo notable es que en la POO se alcanzan con cierta facilidad cuando se respetan algunos lineamientos básicos de diseño que se describirán posteriormente.

Hay que mencionar que se desea que un programa funcione bien para la mayoría de los casos porque para programas relativamente grandes los criterios de verificación formal no son de uso práctico. Ejemplo de lo anterior son la prueba de que un programa es correcto (correctness) usando aseveraciones (assertions) y la especificación algebraica. Estas dos últimas se mencionan porque son de un gran interés teórico. Se dará un gran paso cuando de la especificación de un problema se genere el programa automáticamente, pero esto seguirá siendo un tema de investigación por mucho tiempo.

4.2 El modelo de objetos

4.2.1 Conceptos fundamentales

Sin entrar en gran detalle a continuación se definirán los conceptos básicos de la POO necesarios para el diseño de clases para hacer compresión de datos. De todos los términos el que resulta más importante es el de *clase*, y de él se derivarán todos los demás [Boo91] y [Mey88].

- **Clase.** Clase es un concepto equivalente al de tipo desde el punto de vista de lenguajes de programación. Una clase definirá la estructura y el comportamiento de un conjunto de objetos. El nombre de *clase* tiene su origen en el hecho de que a los objetos se les *clasifica* según ciertas propiedades.

- **Objeto.** Un objeto es una instancia o ejemplo de una clase en particular. El darle algún valor a la clase, vista como tipo, creará un *objeto* que tiene una identidad y estado determinados.

- **Atributos.** Los atributos son cada uno de los componentes que conforman la estructura de una clase. Todos estos atributos determinarán un estado de un objeto en un momento determinado según los valores que tengan almacenados. Todos los atributos de una clase definen cierta estructura de datos, similar a un registro, por lo que cada atributo constituye un espacio de almacenamiento similar a un campo. Un atributo puede ser de una clase o tipo determinado a su vez.

- **Métodos.** Los métodos son operaciones que se pueden llevar a cabo con objetos. Estas operaciones especificarán un comportamiento para el objeto de una clase determinada. Un método no necesariamente opera con objetos de una sola clase, sino que puede combinar operaciones entre distintos objetos de diversas clases. Sin embargo, un método debe pertenecer a una clase determinada, y por lo tanto debe alterar el estado de objetos de tal clase.

- **Mensajes.** Los mensajes son el mecanismo para hacer operaciones con objetos. Un mensaje enviado a un objeto activa cierto conjunto de métodos. Haciendo analogía con los lenguajes de programación imperativa, un mensaje es algo parecido a una llamada a una subrutina con o sin parámetros.

4.2.2 Relaciones entre clases y objetos

A continuación se explicarán términos que se usarán para diseñar clases. En un buen diseño no existen clases aisladas.

Relaciones entre objetos

Los siguientes conceptos son tomados del área de Inteligencia Artificial de la Ciencia de la Computación. Dado que un objeto puede usar a otro(s) para llevar a cabo determinada tarea, y él mismo puede ser usado a su vez por otro(s), surgen tres relaciones elementales de uso entre objetos que son:

- *Actor*, que es un objeto que utiliza a otros objetos, pero que nunca es empleado por otros.

- *Servidor*, que es utilizado por otros objetos, pero que no utiliza a ningún otro objeto.

- *Agente*, que es un objeto que funge tanto como actor y como servidor.

Otra relación es la de contención, en la que un objeto está contenido en otro, o que un objeto contiene a otros.

Relaciones entre clases

Los tres tipos fundamentales de relación entre clases son la generalización, la agregación y la asociación.

- *Generalización*, que permite agrupar propiedades comunes de objetos.
- *Agregación*: que un objeto de determinada clase sea atributo de otra.
- *Asociación*, en la que existe una conexión semántica entre clases.

Combinando generalización y asociación surge la herencia y la instanciación. Como se explicó anteriormente las clases se usan para clasificar objetos según sus características. El resultado de clasificar a los objetos da una estructura a la que se le conoce como jerarquía de clases.

4.2.3 Acceso y herencia

Interfaz de una clase

La interfaz define cómo una clase puede permitir que otras clases accedan a uno de sus componentes, ya sean atributos o métodos. Para ello se utilizan los especificadores de acceso siguientes:

- Privado: el cual permite el libre acceso al componente sólo a la misma clase. Es decir, ninguna otra clase puede manipular tal componente.
- Público: accesible por cualquier otra clase y la clase misma.
- Protegido: permite el acceso sólo a subclases (herederas) de la clase.

No está demás decir que una clase tiene acceso libre a todos sus propios componentes. Es decir, sin importar qué tipo de acceso tenga un atributo o un método, la misma clase siempre puede manipularlos.

Herencia

La herencia especifica qué características comparte una clase con otra. A la clase que comparte sus propiedades se le llama superclase o clase base. Por otro lado, a la clase que hereda se le llama subclase o clase heredera.

La herencia puede ser:

- Sencilla, si sólo se hereda de una superclase.
- Múltiple, si se hereda de dos o más superclases.

La herencia múltiple introduce ciertos problemas. En el diseño de las clases para hacer compresión no se utiliza herencia múltiple. El uso adecuado de la especificación de acceso combinado con la herencia permiten lograr una especialización a la medida para crear un objeto con las características precisas que sean necesarias.

4.3 Análisis y diseño orientado a objetos

4.3.1 Principio abierto-cerrado

Este principio especifica la filosofía fundamental que debe respetarse cuando se programe bajo el paradigma orientado a objetos.

El principio abierto-cerrado requiere que una clase sea lo suficientemente:

- cerrada, como para que oculte sus características internas. Esto se conoce como ocultamiento de información (information hiding). Con ello se logra una mayor abstracción pues no hay que preocuparse por detalles irrelevantes.
- abierta, como para permitir cambios que permitan adaptar la clase a nuevas necesidades, pero sin alterar el funcionamiento que ya era correcto. La facilidad de mantenimiento es el objetivo en este caso.

4.3.2 Análisis

Durante el análisis inicial es necesario identificar objetos que serán candidatos a clases o atributos: los papeles que representan, que podrán ser instancias de clases y cómo tienen que ser manipulados a través de operaciones bien definidas por métodos. Ver qué eventos se dan a lo largo del tiempo que implicarán una serie de envíos de mensajes a objetos. Es necesario crear un modelo abstracto que se apegue lo más posible al problema real.

Operaciones con clases

La idea es crear categorías según características de los objetos. Estas categorías, que son conjuntos, se irán refinando a través de un proceso iterativo aplicando las siguientes operaciones básicas sobre clases:

- Abstracción: crear una nueva clase que no comparte características con las demás.
- Derivación: crear nuevas clases que especializan el comportamiento de clases existentes.
- Composición: formar una nueva clase uniendo clases distintas
- Factorización: dividir una clase en clases más chicas y especializadas.

4.3.3 Diseño

Entre otras cosas se tienen que respetar los siguientes principios:

- Todos los atributos de una clase deben ser privados o protegidos, nunca públicos.
- Integridad de un objeto. Con el inciso anterior se obliga a otros objetos a accesar a un objeto únicamente a través de sus métodos garantizando con ello mantener la integridad del objeto. Suponiendo que la clase generalmente iba a ser extendida en el futuro los atributos se declararon como protegidos para facilitar la herencia.
- Sólo se puede cambiar el estado de un objeto a través de los métodos (operaciones) definidos.
- Un método debe ser público sólo cuando vaya a ser invocado por un objeto de otra clase. Si sólo existe la posibilidad de que lo use la misma clase debe ser privado para evitar operaciones indeseadas.
- Los métodos más importantes tienen que ser declarados como públicos. Los que se consideran de uso exclusivo de cada clase deben ser privados.
- Una clase constituye una definición estática; es decir, no puede cambiar durante la ejecución de un programa. Lo único que cambia dinámicamente es el estado de los objetos.
- Herencia. Para el modelo de objetos de compresión de datos no se detectó la necesidad de efectuar una herencia múltiple. Todas las herencias son simples y se hacen de manera pública; es decir, cada atributo y cada método retiene el especificador de acceso dado en la clase base.
- Diseñar una jerarquía de clases que sea adaptable a cambios, sobre todo a los de redefinición que son los que más problemas ocasionan.
- Que una clase sea estructuralmente lo más independiente posible de las demás. Esta idea se toma de la Ingeniería de Software. A esta propiedad se le llama cohesión viendo a la clase como un módulo.
- Que la interacción entre clases no sea grande. A esto se le conoce como acoplamiento bajo. Con ello se facilitan las modificaciones manteniendo su alcance a un nivel local.

- Se está suponiendo que el lenguaje que vaya a usarse ya proporciona facilidades de entrada/salida, a través de clases predefinidas, y un conjunto de tipos de datos elementales en base a los cuales se crean tipos y clases nuevos.

4.4 Esquemas de clases para compresión

4.4.1 Introducción

En base a los conceptos y criterios explicados ahora se verá como se diseñaron las clases para poder contar con una biblioteca de clases para comprimir datos. Tal diseño es útil para cualquier LOO, aunque se verá que no siempre es posible por el tipo de operaciones de bajo nivel que son necesarias como se expondrá en el capítulo siguiente.

4.4.2 Clases genéricas

A continuación se describen clases que son estructuras de datos comunes, que son útiles para hacer compresión. Una buena referencia sobre estructuras de datos y algoritmos es [Aho83]. Una clase es genérica cuando puede ser utilizada para contener una colección de datos que pueden ser de un tipo distinto. Este es una de las características que permite hacer reusabilidad de software con facilidad.

Nodo

Clase genérica que define a objetos que pueden guardar referencias a objetos de cualquier tipo. Para efectos del modelo de objetos para comprimir datos basta con saber cuál es el nodo siguiente como se verá en la clase *Lista*; en este caso una referencia al nodo anterior es innecesaria.

Lista

Esta clase permite crear listas simplemente ligadas de objetos de cualquier tipo (clase) usando objetos de la clase elemental *Nodo*. Es decir, *Lista* usa a *Nodo* por agregación. Se usa dentro del algoritmo de Huffman para guardar el bosque (lista de árboles) con el que se generan los códigos prefijos.

Entre sus métodos destaca el método que permite hacer una inserción de un nodo en el lugar adecuado según un orden definido sobre sus elementos.

Arbol Binario

Esta clase define la estructura de un árbol binario. Un árbol binario es una gráfica dirigida acíclica. El árbol se encuentra constituido por nodos, cada uno de los cuales puede tener 0, 1, ó 2 hijos, y un solo padre. La excepción es el nodo denominado raíz que no tiene padre. Esta es una clase genérica que dentro del modelo va a contener objetos de la clase *Código* en cada nodo.

Sus métodos permiten crear nodos en el árbol de abajo hacia arriba, tal como lo requiere el algoritmo de Huffman, en el que se crea un nuevo nodo que tiene como hijos a los dos árboles de peso mínimo y cuyo contenido será la suma de los pesos correspondientes de sus dos hijos. Para los códigos de Shannon-Fano no se utilizan árboles binarios; basta con tener un método que haga particiones sobre la tabla de códigos.

Pila

Esta clase define a la clásica estructura de datos lineal en la que la inserción y el borrado de elementos se hace por un extremo. Al igual que las anteriores es una clase genérica en donde cada uno de sus elementos va a ser un código generado por el algoritmo LZW.

4.4.3 Clases para compresión de datos

A continuación se describen las clases que abarcan el modelo de objetos completo. Tales clases tendrán una fuerte relación de uso y de agregación entre ellas. Para el modelo de de compresión de datos no hubo necesidad de hacer herencia múltiple; y de hecho sólo existe un nivel de herencia entre clases. Los identificadores de clases están enfatizados.

Código

Esta es una de las clases más elementales para poder hacer compresión. Un código como tal va a estar constituido por una cadena de bits de una longitud variable. A tal cadena de bits le va a corresponder a un símbolo, que para efectos de este diseño será uno del código ASCII. Como la mayoría de los algoritmos requieren una distribución probabilística de aparición para poder generar códigos, cada uno de éstos tendrá una frecuencia que será calculada de manera estática.

Dado lo anterior un objeto de esta clase cuenta con una serie de métodos para: incrementar su frecuencia, retornar su código, dar su longitud, poder

ver su estado interno (variables de instancia), o inicializarlo.

Tabla de Códigos

Que tiene como componente un arreglo de códigos. O sea, un código tiene una relación de agregación con una tabla de códigos. Se tiene otro atributo llamado mapeo, que existe para registrar las permutaciones que se hagan sobre una tabla de códigos al querer ordenar los códigos por algún criterio. Ello se verá a detalle en el capítulo sobre programación. Por último existe un atributo de tipo entero, actualizado de manera estática, en el que se guarda la suma total de frecuencias de todos los símbolos de la tabla.

En base a los atributos tabla, mapeo y suma de frecuencias se definen varios métodos para poder ordenar o ver la tabla de códigos. Así mismo se definen operaciones para buscar un código, incrementar la frecuencia de determinado código o calcular la redundancia, entropía, y espacio requerido por un mensaje fuente determinado.

Esta misma clase es capaz de generar sus propios códigos prefijos minimales en base a las frecuencias de los símbolos, por lo que aquí se encuentran definidos los métodos de generación de códigos prefijos minimales de Huffman, y de Shannon-Fano.

Archivo

Para efectos de implantación un mensaje fuente a codificar será considerado como un archivo de símbolos. Se define una clase especial de archivo comprimido que sirve para tal efecto. Tal archivo tiene los siguientes atributos: firma, que es una cadena que indica si el archivo fue comprimido por el programa o no, un nombre de archivo, correspondiente al archivo original, una tabla de frecuencias y el método por el cual se hizo la compresión.

Entre los métodos importantes de esta clase están: el que contabiliza las frecuencias de los símbolos, el que escribe el encabezado de un archivo comprimido, y el que lo lee.

Cronómetro

Usada para medir el tiempo que lleva el proceso de codificación o decodificación con alguno de los algoritmos. Un *Cronómetro* es usado por un objeto de la clase *Sesión*.

Sesión

Esta clase es la que se encarga de poner todos los componentes en uno solo. *Sesión* sirve para crear una interfaz con el usuario. Sus atributos permiten guardar el nombre del archivo de entrada, el de salida, qué proceso se va a ejecutar (comprimir o descomprimir) y el método elegido. Con estos parámetros es posible activar algún método del objeto correspondiente según la elección del usuario.

Codificador Decodificador

Esta clase especifica las características generales que va a tener cualquier compresor/descompresor de datos. Pero no sólo eso, esta clase da la estructura para cualquier codificador/decodificador. De hecho se había visto que comprimir es un caso particular de formas de codificar. Esta es una de las clases más importantes del modelo y de hecho guarda relación de agregación con casi todas las demás clases descritas anteriormente,

Entre sus atributos que son clase están un *Archivo*, que será el archivo a procesar, una *Tabla de Códigos* que le proporcionará los códigos necesarios para codificar o decodificar. Entre sus atributos que no son objetos de alguna clase están los espacios de almacenamiento de entrada/salida (buffers) para hacer la codificación/decodificación, índices y un caracter para procesar .

Esta es una clase abstracta y de hecho sus métodos serán aprovechados por clases herederas de *Codificador Decodificador*. Como métodos importantes están dos que permiten convertir una sucesión de bits a cadena de caracteres y viceversa. Estos métodos van a ser empleados por todas las clases herederas.

A continuación se explicarán las tres clases con las que se programaron los métodos expuestos en el capítulo anterior:

Compresor Prefijo

En esta subclase de *Codificador Decodificador* no es necesario agregar ningún atributo, y de hecho lo único que tiene son métodos para codificar una cadena de símbolos cuando ya se tiene generada una *Tabla de Códigos* prefijos minimales. La generación de códigos no es responsabilidad de esta clase, sino de la *Tabla de Códigos*, ya que cada *Código* está asociado a un símbolo.

Así mismo cuenta con un método para decodificar una cadena de bits que haya sido resultado de concatenar códigos prefijos. El resultado será el archivo original cuyo nombre está almacenado en la clase *Archivo*.

Es importante notar que los métodos son lo suficientemente generales como para manipular cualquier tabla de códigos prefijos; el que éstos sean minimales es una propiedad que se pidió para lograr ocupar menos bits, como se explicó antes.

Compresor Aritmetico

Esta subclase extiende a la superclase *Codificador Decodificador* agregando los atributos necesarios para ir calculando los números f y m de los algoritmos. Es importante notar que tiene dos arreglos para guardar las frecuencias de aparición y las frecuencias acumuladas de cada símbolo. Esto ya se encontraba dentro de la clase *Tabla de Códigos*, pero tuvo que ser redefinido en esta clase para crear el símbolo ficticio Fin De Cadena (FDC).

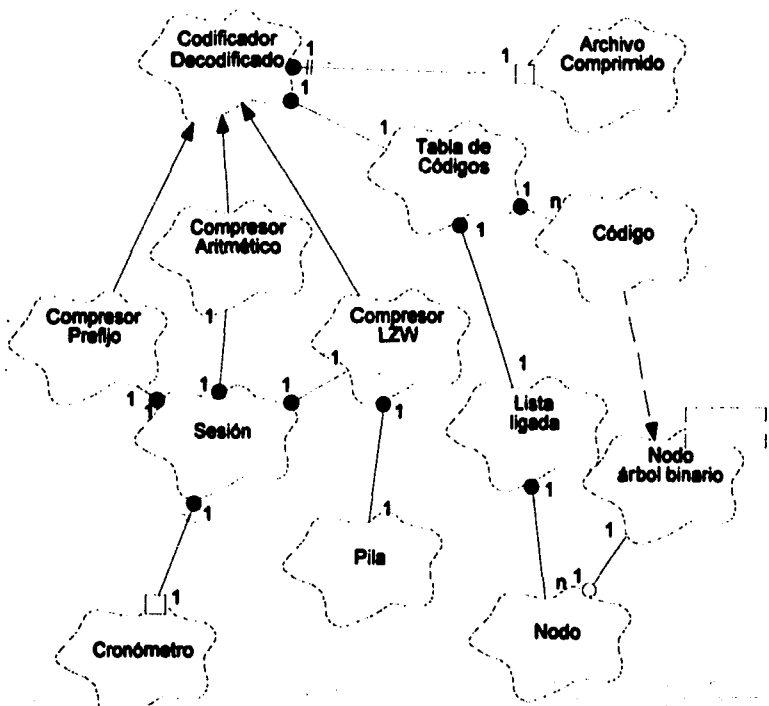
Sus dos métodos más importantes son simplemente los algoritmos descritos en el capítulo anterior. Ambos son de acceso privado, y accesibles por medio de los métodos *comprimir* y *descomprimir* que son públicos. Los demás métodos son métodos auxiliares para ir haciendo la codificación o decodificación, y por lo tanto son privados.

Compresor LZW

Esta clase heredera extiende a *Codificador Decodificador* agregando una tabla de dispersión para ir guardando los códigos, y una *Pila* necesaria para el proceso de decodificación. Esta *Pila* (stack) es necesaria por la manera en la que se guarda una cadena de longitud n con su código asociado: el código de la subcadena inicial de longitud $n-1$ y el caracter concatenado al final de ella.

Esta clase tiene métodos para hallar una cadena en la tabla de dispersión, para emitir o leer un código, y para decodificar un código por la regla del patrón concordante más largo.

A continuación se muestra el modelo de objetos para hacer compresión.



Capítulo 5

Programación usando C++

En el presente capítulo se hablará de la manera en la que se programaron los algoritmos ya descritos respetando el modelo de objetos diseñado en el capítulo anterior. Hasta donde fue posible se trataron de aprovechar características del lenguaje C++. No se pretende dar una exposición sobre el lenguaje C++, pero sí hablar de sus características más relevantes.

5.1 Introducción

C++ es un superconjunto del lenguaje C. Es decir, C como lenguaje es un subconjunto propio del lenguaje C++, por lo que cualquier programa válido en C sigue siendo válido en C++. A C++ se le considera un lenguaje híbrido ya que combina todas las características de C y los conceptos fundamentales de la programación orientada a objetos. C es un lenguaje que tiene mucho tiempo de ser usado en ambientes profesionales de desarrollo desde los 70's. Entre las aplicaciones que se le han dado están la programación de sistemas operativos, compiladores, bases de datos, comunicaciones de datos, entre otras.

La popularidad de C++ crece rápidamente, y no hay duda que se convertirá en el lenguaje de POO más usado en los 90's. Aunque existen rivales como Eiffel [Mey92], Smalltalk, y CLOS (una extensión de Lisp) ninguno ha tenido la aceptación de C++. Esto se debe a que C ya antes era el lenguaje de uso más extendido, y por toda la experiencia y programación existentes lo lógico era conservarlas. Ello quiere decir que C++ no es necesariamente el mejor lenguaje ni el más puro orientado a objetos, pero sí el que tiene más compiladores en más computadoras, lo cual constituyó una de las razones

- Poder pasar funciones como parámetro a otra función. Se gana mucha abstracción pues es posible aplicar programación funcional.
- Una colección amplia de rutinas de biblioteca. Desde rutinas para manejar la memoria y el disco, hasta funciones de biblioteca para hacer cálculos numéricos.
- Apuntadores. Gran parte del poder del lenguaje C, reside en aprovechar los apuntadores y toda una aritmética que existe entre ellos.
- Operadores de bit. Entre estas operaciones están el and, or, not, xor y los corrimientos. Estas operaciones son típicas en un lenguaje ensamblador o de máquina.
- Un preprocesador poderoso que permite lograr una compilación modular, eficiente y condicional inclusive.
- Hacer programas muy eficientes. Aunque ello a veces obscurece el entendimiento a otro programador.
- Dar libertad al programador. Esto es un arma de dos filos pues permite mezclar tipos y hace difícil rastrear errores.
- Portabilidad. Gracias a que C ha sido estandarizado y su uso está tan extendido que existe un compilador virtualmente para cualquier computadora. En este aspecto es de gran ayuda el preprocesador.

5.2.2 Extensiones importantes de C++

Como se dijo C++ extendió al lenguaje C agregando características para programar bajo el paradigma orientado a objetos, pero agregó otras que también ayudan a ganar abstracción, que no son necesariamente características de un lenguaje orientado a objetos; se empieza por las de orientación a objetos.

- El poder definir clases, entendidas como tipos nuevos. Con ello la facilidad de declarar los atributos y métodos (funciones miembro) como privados, protegidos o públicos para facilitar la herencia conservando la encapsulación (ocultamiento de información).
- Permitir la sobrecarga de operadores. Ello se refiere a que un mismo operador se utilice para operaciones de una semántica parecida, pero con datos de distinto tipo.

multiprocesador. A la fecha no existe un estándar para hacer programación en paralelo. Existen bibliotecas y extensiones al lenguaje C y C++, pero están supeditadas al uso de determinada computadora y no están suficientemente probadas.

- Integridad de objetos. Esto se logra a través de la verificación de precondiciones y poscondiciones después de invocar algún método; pero es responsabilidad del programador que esta integridad se respete; no es algo automático. El manejo de excepciones ayuda un poco en este sentido.

- Existen ciertas extensiones que no son estándar, pero esto se estabilizará con el tiempo. De hecho se usaron características que están estandarizadas hasta donde fue posible.

- Abusar de la libertad que se le da al programador. Una asignación es una expresión. Una expresión puede ir en cualquier condición. Por ejemplo: en la condición de un *if* es válido hacer una asignación. Dentro de la inicialización de un *for* es válido hacer una serie de asignaciones. Los apuntadores permiten hacer cosas que en lenguajes como *Pascal* o *Modula* serían imposibles.

5.3 Limitaciones de una computadora

5.3.1 Memoria vs disco

En una computadora todo tiene que ser procesado en el almacenamiento (memoria) primario [Dei80]. Aunque el costo de la memoria ha descendido de cualquier manera sigue siendo un recurso caro y limitado. En el programa que se creó para hacer compresión se manipulan dos archivos, uno de entrada y otro de salida.

Cuando se hace un proceso de compresión el archivo de entrada es alguno que contiene datos y el de salida es un archivo en el que se ha reducido la redundancia con alguno de los métodos explicados anteriormente. Cuando se descomprime el archivo de entrada será alguno codificado por determinado método y la salida será el archivo de datos original.

En general no es posible hacer suposición alguna respecto al tamaño de los archivos que se manejan. Tal tamaño es variable y en muchos casos puede rebasar el espacio disponible en memoria. Por tal razón los archivos tienen que ser procesados por pedazos. Estos pedazos son llevados a la memoria en

el proceso de lectura, o al disco en el proceso de escritura. A estos pedazos se les llama buffers, que es un término común en inglés, y forman parte de un objeto de la clase *Codificador Decodificador*.

Es importante mencionar que la entrada/salida podría hacerse a nivel de byte, pero el costo de hacerlo así sería muy alto pues se sacrificaría velocidad de procesamiento. Cuando se hacen operaciones de lectura o escritura en disco la parte más lenta es el acceso a disco. El proceso de transmisión de datos es el más rápido pues es electrónico y no incluye operaciones mecánicas. Usando buffers se minimiza el número de accesos a disco. Las rutinas de biblioteca de entrada/salida del lenguaje facilitan la lectura de disco cuando un buffer queda vacío o mandar automáticamente a disco (commit) el contenido de este espacio de almacenamiento cuando se llena. Consúltese [Dei80] para conocer en detalle las operaciones en disco.

5.3.2 Precisión y rango

Cuando se utiliza una computadora para hacer cálculos en punto flotante el problema fundamental que tiene que resolverse es el de la precisión. La codificación aritmética planteada en su versión más simple supone una precisión aritmética infinita [Lel88]. Como se estudió esto no es posible en la práctica. Existen alternativas para implantar aritmética de precisión infinita en una computadora; por ejemplo usando listas ligadas de dígitos. Tales listas podrían ocupar tanto espacio como hubiera disponible en la memoria y así aumentaría la precisión de la máquina. Pero aún así tendría que guardarse un número finito de dígitos decimales. En suma la precisión hay que controlarla.

En el algoritmo programado se utiliza únicamente aritmética entera por lo que en la implantación se utilizan números de tipo *long int*. El rango de un entero de este tipo es bastante amplio, $-2^{31} \ldots 2^{31} - 1$, pues su tamaño es de 32 bits en complemento 2; ver [Man83].

Sólo es necesario almacenar la frecuencia de cada símbolo. En teoría debería guardarse su probabilidad de ocurrencia, que es el resultado de dividir su frecuencia entre la suma total de frecuencias, pero ello no se hace por los problemas que conllevan los números en punto flotante. Por lo tanto se define un tipo para el programa que identifica a enteros largos sin signo: *unsigned long int*. Como en este caso no se usa el bit del signo sobre la representación de 32 bits el rango posible es $0 \ldots 2^{32} - 1$. En la práctica es difícil rebasar el límite superior de este rango que es mayor a $4 * 10^9$, unos 4 gigabytes.

5.4 Aprovechando características de C++

5.4.1 Operadores de bit

Esta es una de las características poderosas del lenguaje C++ heredadas de C, que evita el programar en lenguaje ensamblador o en lenguaje de máquina. Las operaciones de bit son operaciones de muy bajo nivel pues son las que llevan a cabo los circuitos combinacionales elementales.

Dentro de los códigos prefijos minimales se convierte de bits a cadena de caracteres (*char**) y viceversa empleando operaciones de máscara de bits (bit mask operations). Por ejemplo para convertir de una cadena de caracteres 1's y 0's a bits se hace lo siguiente. Para prender un bit en una posición (0...7) determinada de un *byte* se usa un corrimiento a la izquierda del número 1 (uno), (al correr un 1 a la izquierda todas los bits menos 1 quedan en 0), y luego se aplica una operación or con el operador | y el *byte*. Similarmente si se desea poner en 0 (apagar) un bit determinado se hace un corrimiento a la izquierda del 1, tal resultado se niega con un *not*, quedando en 0 el bit deseado y en 1 todos los demás; después se aplica un and (&) con el *byte* quedando en 0 precisamente el bit deseado. Para convertir de bits a cadena (char *) el proceso es más sencillo: se aplican corrimientos para ver si determinado bit vale 1. Si vale 1 se concatena por la derecha un '1'; en caso contrario un '0'.

En la codificación aritmética los operadores que más se utilizan son el de corrimiento a la derecha >> por ejemplo para emitir un bit. O el corrimiento a la izquierda << para multiplicar por dos.

Al usar los códigos LZ que son las entradas al diccionario se generan números enteros cuya longitud no necesariamente es un múltiplo de la longitud de un byte. Por lo tanto se usan corrimientos y operaciones *or* para irlos emitiendo, suponiendo una longitud de 8 bits para un byte.

A continuación se listan los operadores de bit que se emplearon en el programa.

Tabla 5.1

Operador	Uso
<<	Corrimiento a la izq.
>>	Corrimiento a la der.
~	not
∧	xor
&	and
|	or

Para saber con detalle cuál es el resultado de aplicar estos operadores a una pareja de bits, o a una cadena de bits consulte [Man83]. Los corrimientos, por ejemplo, llenan con 0's las posiciones vacías.

5.4.2 Apuntadores

Hasta donde es posible cada objeto mantiene referencias a otros objetos evitando con ella la réplica de objetos en memoria. Todo esto se hace usando apuntadores. Por mantener un diseño sencillo no se mantienen apuntadores inversos, sino sólo en un sentido. Con esto quiere darse a entender que si un objeto emplea a un objeto de otra clase sólo existe una referencia entre el objeto actor y el servidor, pero no existe otra entre el que es usado y el cliente. Otro aspecto interesante es que los objetos en general se pasan por referencia por medio de apuntadores para hacer más eficientes las invocaciones de métodos.

La excepción a lo anterior es la clase *Archivo* en la que quedan guardadas las frecuencias de los 256 símbolos del código ASCII a fin de que éstas puedan ser recuperadas en el proceso de decodificación aritmética o de códigos prefijos minimales. Un compresor aritmético necesita calcular frecuencias tanto de aparición como acumuladas para cada símbolo. Además un compresor de tipo aritmético requiere al igual que los códigos LZ crear un símbolo ficticio que marca el fin de la cadena codificada. Por ello se crean dos nuevos arreglos adicionales que son distintos a los de un objeto de tipo Archivo para los procesos respectivos.

Para hacer un uso eficiente de la memoria es indispensable el uso de apuntadores. Para este propósito están las clases *NodoArbolBin* y *NodoLista*, que se utilizan para construir árboles binarios y listas simplemente ligadas, en las que en cada nodo se guardan referencias (apuntadores) a objetos. Ese objeto es de alguna clase por la que se parametriza la clase genérica. De este modo se construye el bosque (lista de árboles) para el método de Huffman

en donde cada nodo es un árbol binario. Y ahora cada árbol binario contiene a su vez apuntadores a códigos en los que está almacenado cada símbolo del código ASCII, su frecuencia y su código en bits.

En el programa frecuentemente se manipulan cadenas, apuntadores a caracteres en C++; y en este aspecto el lenguaje es sumamente eficiente. Por otro lado todos los objetos pasados como parámetros por referencia se accesan utilizando apuntadores en la definición de cada función miembro. Lo anterior se prefirió al uso del operador de referencia: &, pues usando * se tiene que utilizar & en la llamada y es posible detectar cuándo se le van a hacer modificaciones al objeto. Si el paso por referencia se hace por razones de eficiencia entonces sí se usa el operador de referencia &.

5.4.3 Portabilidad

Hoy en día los únicos lenguajes en los que se pueden desarrollar programas que pueden correr casi en cualquier máquina son C y C++. Buena parte de ello se debe a que desde que nació C, el antecesor de C++, se tuvo en mente la programación de sistemas. Desde 1979 han existido comités que han estandarizado el lenguaje ya sea desde el aspecto de construcciones del lenguaje y tipos o por el uso de bibliotecas. Existe virtualmente al menos un compilador de C, aunque no siempre de C++, para cada computadora fabricada en la actualidad. La tendencia es que C++ ocupe el lugar que C tiene en la actualidad.

No hay que perder de vista que sí pueden surgir ciertos problemas de portabilidad en la práctica. En general ello se debe a que se hacen ciertas operaciones de bajo nivel que están muy relacionadas con la arquitectura de la máquina usada.

Ejemplos de problemas potenciales son:

- El uso de apuntadores cuyo tamaño varía por el tamaño del bus de datos de la computadora o por el modo de direccionamiento empleado.

- Emplear datos de tipo entero (int) en los programas que pueden ser de longitud distinta, y que por lo tanto pueden originar problemas de rango. Ello se debe a que en C los enteros generalmente tienen una longitud igual a la de los registros de un procesador.

- El uso de bibliotecas especializadas que no estén estandarizadas. En esta categoría caen bibliotecas para programar en ambientes gráficos,

operaciones con bases de datos o rutinas que manipulan un dispositivo periférico especial.

- Variaciones de un sistema operativo a otro.

El problema de los apuntadores se resuelve definiendo constantes adecuadas para el tamaño de los buffers. Este problema se origina en el sistema operativo MS-DOS en el que la memoria se divide en segmentos de 64k máximo. En otras computadoras como una SUN, el acceso a la memoria se hace dando direcciones de 32 bits por lo que el tamaño posible de los arreglos, entre ellos los buffers, puede ser mucho mayor.

En cuanto a los enteros, se emplean enteros *unsigned long int* siempre que sea posible. Para otros casos se usa un *short int* cuando existe la seguridad que no va a existir una variable $> 2^{15} - 1 = 32,767$. En ambientes de estaciones de trabajo el tipo de datos *int* es de 32 bits por lo que es equivalente a un *long int*.

La interfaz del programa es sencilla. Todos los parámetros se especifican en la línea de comandos. Estos parámetros son los archivos de entrada y salida respectivos, qué método se desea emplear así como elegir entre un proceso de compresión o descompresión. No se usan bibliotecas de ambientes gráficos pues aparte de que no ayudan en gran cosa a este programa tienen ciertos problemas de portabilidad. Destacan sobre todo las bibliotecas para desarrollo en ambientes gráficos XWindows.

Todas las constantes y tipos principales quedan definidos en un archivo de cabecera llamado *global.h*.

5.5 Bibliotecas estandarizadas

5.5.1 Medida del tiempo

Para medir el tiempo se emplean *time* y *difftime* que son dos funciones de biblioteca estandarizadas. Las convenciones de llamada se encuentran especificadas en el archivo *time.h*. En SUN no se encuentra definida la función *difftime*, por lo que se definió una macro para poder usarla.

Para efectos de medición lo que interesa es tener una idea del tiempo en segundos que tarda cada algoritmo, y no los segundos aproximados a milésimas. Lo anterior sería necesario si se hiciera compresión de datos en tiempo real, o los algoritmos tuvieran tiempos muy parecidos o en general el proceso fuera muy rápido. Lo anterior no sucede pues se hacen continuamente operaciones de lectura/escritura a disco que como se había explicado

son muy lentas, y a pesar de los avances que se han dado en este sentido siguen teniendo un orden de magnitud de milisegundos (mseg).

Es importante mencionar que la mayoría de discos duros disponibles en la actualidad tienen tiempos de acceso que van de los 10 a los 40 milisegundos. Para un archivo de tamaño considerable las operaciones de compresión se tardarán segundos, lo cual se verá con detalle en el capítulo siguiente. Dentro de algunas bibliotecas es posible encontrar funciones que permiten conocer el tiempo transcurrido hasta con milésimas de precisión, pero no son portables entre MS-DOS y UNIX.

5.5.2 Entrada/salida

Por la rapidez y la flexibilidad toda la entrada/salida del programa se hace usando funciones de *stdio.h*. Este archivo de cabecera es de C. C++ tiene su correspondiente *iostream.h* en el que existen clases a las que llama flujos (streams) para hacer lectura/escritura. El problema con estas clases es que hay métodos que no están estandarizados, y son más lentos pues internamente se usan las rutinas clásicas del lenguaje C.

El tipo básico empleado para manipular archivos es FILE que es un registro (*struct*) que permite hacer la manipulación de archivos a bajo nivel que requieren los procesos de compresión.

En el programa se usan las funciones *putc* y *getc*, que son extremadamente eficientes pues están definidas como macros. Estas dos funciones son usadas por la codificación aritmética y el método LZW. La asociación de un buffer para hacer entrada y salida de manera más eficiente se hace con la instrucción *setvbuf*.

En cuanto a los códigos prefijos minimales se emplean las funciones *fread* y *fwrite* que leen o escriben respectivamente un buffer de una longitud proporcionada por el programador. El control explícito de cuándo se tiene que leer o escribir es responsabilidad del programa.

5.6 Eficiencia

5.6.1 Optimización de ciclos

Hasta donde fue posible se trató de verificar el mínimo de condiciones para dejar de ejecutar un ciclo como *while* o *do while*. Es decir se trató de dejar expresiones booleanas mínimas para hacer más rápidos los procesos. En C

es posible salirse de un ciclo con la instrucción *break* dentro de un *if*, pero en general se evitó con el objeto de hacer más claro el programa.

De las estructuras de repetición disponibles en C++ la más eficiente es la instrucción *for*. Siempre que se manipularon arreglos se usó esta instrucción. Las variables índice o contadores se mantuvieron en registros para que el acceso a ellas fuera inmediato para el procesador.

5.6.2 Funciones miembro en línea

En C++ están identificadas por la palabra reservada *inline*. Este tipo de funciones permite que la llamada a un método se haga por nombre (call-by-name). En este tipo de llamada en vez de pasar los parámetros por la pila se hace una macrosustitución de los parámetros por las variables y expresiones de la llamada y se inserta el código correspondiente en vez de hacer la llamada por pila [Str91]. Existe un sacrificio (tradeoff) de velocidad por espacio, pero en general vale la pena debido a que los procesos de codificación y decodificación son lentos para archivos grandes .

Es importante notar que la POO tiende a definir de manera natural métodos cortos, pues todas las operaciones con objetos deben hacerse a través de sus métodos definidos para no quebrantar el encapsulamiento y la integridad del objeto. La mayoría de las funciones de la clase *Código* quedaron definidas de esta manera. Si una función definida en el esquema de una clase no tiene ciclos automáticamente se considera como en línea. Los métodos que hacen los cálculos para obtener el bit superior $H(f)$ y obtener los bits inferiores $L(f)$ de f quedan definidos como funciones miembro en línea dado que son empleadas continuamente. Las operaciones correspondientes al nodo de una lista ligada o un árbol binario también están en línea.

5.6.3 Operaciones con números enteros

Como se dijo anteriormente se emplearon siempre variables de tipo entero. Estas son manipuladas particularmente con rapidez por cualquier computadora dada su estrecha relación con los registros del procesador. En C++ los operadores de incremento y decremento ++ y -- son muy eficientes pues la mayoría de los compiladores los transforma en una instrucción específica (INC o DEC) del procesador cuando se hace la generación de código. La manera alternativa de incrementar una variable (i=i+1) es generalmente más lenta pues implica un análisis sintáctico y semántico más complicado, y varias instrucciones de máquina al momento de generar código objeto.

5.6.4 Búsqueda

Para hacer más rápido el acceso a los códigos a un símbolo se usó búsqueda binaria para los códigos prefijos generados por el método de Shannon-Fano y de Huffman. La llave de búsqueda es necesaria en el proceso de decodificación en el que tiene que consultarse la tabla de códigos por cada bit analizado.

En el caso de la codificación aritmética, en la que se tiene que hacer una consulta constante de la tabla de frecuencias buscando por frecuencia, también se empleó la búsqueda binaria.

Para los códigos Lempel Ziv se usaron técnicas elementales de dispersión (hashing) para tener tiempos de búsqueda promedio constantes, $O(1)$. Lo anterior origina que estos códigos tengan los algoritmos con mejores tiempos de respuesta como se verá en el capítulo siguiente.

5.7 Ingeniería de Software

5.7.1 Control de adiciones y cambios

Dada la magnitud del programa se mantuvieron ciertos controles para llevar un registro de los cambios y hacer más fácil la localización de errores o problemas. Estos son requisitos elementales para el desarrollo de un proyecto de software.

Bitácora

Este es un registro cronológico de las adiciones y cambios que se le hicieron a los distintos módulos que conforman el programa. El registro se separa por fechas, y dentro de cada fecha se describe cada modificación individualmente anotando la clase y/o el método al que se le hizo la modificación.

También se anotan errores detectados durante la fase de prueba y se sugieren posibles mejoras para el programa ya sea para mejorar la rapidez, para hacer más fácil la reutilización de los métodos, o para hacer más entendible el código en C++.

Cabecera de módulos

Cada módulo del sistema lleva una sección de comentarios al principio del archivo de código fuente (source code), en el que están anotados datos generales y fechas. Primero se registra la fecha en la que se creó el módulo y

a continuación una serie de fechas en las que se le hicieron modificaciones al módulo.

En general cada archivo contiene el esquema de la clase y la programación de los métodos correspondientes. Hay algunos casos en los que hay dos o más clases en un archivo cuando éstas se encuentran muy relacionadas. Un ejemplo de ello son las clases *NodoLista* y *Lista* con las que se construyen listas simplemente ligadas.

5.7.2 Identificadores del programa

Tipos y constantes

Todas las declaraciones de tipos y constantes globales se encuentran en el archivo *global.h*. Dentro del archivo se encuentran separadas las constantes de los tipos. La ventaja de declarar constantes es que ello permite probar el programa con distintos parámetros sin alterar el código programado. Los tipos globales permiten tener una referencia uniforme y evitar problemas de compatibilidad.

Entre los tipos importantes se tienen: *ulong* y *uint4* que son los identificadores de tipo para enteros largos sin signo (unsigned long); ulong se utiliza para guardar frecuencias y servir de índice para accesar buffers. *byte* que representa básicamente el valor del código de un símbolo del código ASCII. Frecuencia, la cual sirve para dar una definición consistente de las frecuencias del código ASCII, tipo utilizado por la clase *Archivo*.

A continuación se listan los tipos nuevos importantes.

Tabla 5.2

Tipo	Contenido
byte	entero sin signo de 1 bytes
int2	entero con signo de 2 bytes
uint2	entero sin signo de 2 bytes
int4	entero con signo de 4 bytes
uint4	entero sin signo de 4 bytes
Frecuencia	tabla de frecuencias del código ASCII

Constantes

Entre las constantes existen dos que indican cómo puede abrir archivos el programa ($LECTURA$ = "rb", $ESCRITURA$ = "wb"). Lo importante es que ambas obligan a abrir un archivo para lectura o escritura en modo

binario. Un archivo en C++ puede ser abierto en modo texto o binario. El modo binario es preferido porque incluye a cualquier tipo de archivo y por lo tanto hace al programa de uso general. Otras constantes indican el tamaño máximo de un buffer o de la cadena en la que se almacenan los códigos prefijos minimales. Dado que el código ASCII tiene 256 símbolos existe una constante entera *DIMTC* con tal valor.

Clases, objetos y variables

Las clases se nombran usando un identificador que sea lo más explícito posible. En general se respetan los siguientes criterios: no abreviar, comenzar el nombre de una clase nueva con mayúscula y poner el nombre, que en general es un sustantivo, en singular.

Los objetos comúnmente se nombran usando los artículos: un o una según el género concatenado con el nombre de la clase correspondiente. Nombres de objetos, atributos y variables empiezan con minúscula siempre.

Métodos y funciones

El identificador de un método siempre es algún verbo en infinitivo concatenado con el nombre de algún objeto sobre el que actúe. Tal identificador siempre comienza en minúscula. Por la nomenclatura no se distinguen entre métodos observadores, iteradores, o transformadores. A las funciones miembro (métodos) que no alteran el estado de un objeto se les agrega la palabra reservada const en su declaración. La razón de poner los verbos en infinitivo es que es más fácil recordar el nombre que alguna conjugación, más porque en el español existen muchos verbos irregulares y la conjugación tiene que coincidir con la persona (desde el punto de vista gramatical).

Todos los métodos cortos son declarados en línea para hacerlos más rápidos. En especial los métodos de la clase *Codigo* y la clase *Compresor Aritmético* caen en esta categoría. Por otro lado se trató de no crear métodos muy largos, ya que ello dificulta el proceso de depuración de un programa. En general ningún método tiene una longitud superior a una página. Además como se había citado la POO generalmente tiende a definir funciones cortas.

Respecto a la declaración de parámetros, en general los que son de entrada van primero, y los de salida después. Por razones de eficiencia los objetos grandes se pasan por referencia usando apuntadores. Cuando es posible se asignan valores por defecto a los parámetros.

5.7.3 Escritura del código fuente

Indentación

El código está indentado adecuadamente siempre a tres espacios. La llave ({) que indica que empieza un bloque de instrucciones en general empieza en el mismo renglón de alguna estructura de control (if, while, etc), alineando el indicador de fin de bloque con el principio (}). Para indicar el principio y fin de la declaración de métodos y funciones las llaves siempre ocupan un renglón cada una sola, excepto en el caso que sean declaradas dentro del esquema de una clase.

Comentarios

Una parte muy importante de un programa está constituida por los comentarios. Cuando otro programador o uno mismo necesita modificar el código fuente lo primero que se analiza son los comentarios, si es que estos existen. En un lenguaje como C++ que tiene una notación tan críptica y en el que existe una cultura por la eficiencia, los comentarios resultan indispensables para poder entender qué es lo que hace un programa.

Los comentarios se ponen a la derecha de las declaraciones de variables y atributos. Para las funciones miembro se da una breve explicación de su uso. Existen algunos comentarios que ocultan instrucciones, sobre de todo funciones de salida, como *printf*, *cout*, que se insertaron para efectos de depuración.

Capítulo 6

Resultados experimentales

6.1 Equipo de cómputo usado

6.1.1 Bajo sistema operativo MS-DOS

El programa se corrió en una computadora personal con procesador CISC 486SLC Texas Instruments con un reloj a una velocidad de 40 MHz. Los archivos se procesaron y almacenaron en un disco duro de 170Mb con un tiempo promedio de acceso de 15 milisegundos.

6.1.2 Bajo sistema operativo UNIX

Se usaron además estaciones de trabajo SUN con procesador RISC modelo SPARC para hacer pruebas de portabilidad. Sin embargo, los resultados mostrados no se obtuvieron en este tipo de equipo. Los porcentajes de compresión para un determinado archivo son exactamente los mismos, pues el formato que se adoptó es estándar en ambos puertos. Los tiempos, por otro lado, serían proporcionales a los que se muestran en este capítulo.

6.2 Tipos de datos a comprimir

La tabla siguiente resume los tipos de datos y tamaños correspondientes a los archivos que se utlizarán para efectos experimentales.

Tabla 6.1

Archivo	Contenido	Tamaño
aa	sólo A's	701,345
bc	programa ejecutable	1,410,992
c	lenguaje C++	865,398
imagen	imágenes digitales	589,424
pas	lenguaje Pascal	407,472
tex	texto en español	453,378

Cantidades en bytes

6.3 Grado de compresión

A continuación se anota el porcentaje P al que queda reducido cada archivo después del proceso de compresión. Como se explicó en el segundo capítulo la medida de compresión que va a ser usada es el porcentaje que representa el archivo comprimido respecto del archivo original. Todos los porcentajes están redondeados al entero más cercano.

Tabla 6.2

Archivo	Shannon	Huffman	Aritmética	LZW
aa	13%	13%	1%	1%
bc	79%	78%	78%	78%
c	63%	61%	61%	49%
imagen	79%	78%	77%	151%
pas	63%	62%	61%	40%
tex	61%	60%	60%	45%

Porcentajes redondeados

6.3.1 Comparación general entre métodos

Como se puede apreciar, en general el grado de compresión es muy parecido para el método Shannon-Fano, los códigos de Huffman y la compresión aritmética. Esto resulta natural ya que todos los métodos se basan en la distribución probabilística de aparición de los símbolos. Por la manera en que se construyen los códigos la codificación aritmética es la que obtiene los mejores resultados, pero a costa de mucho tiempo para controlar la precisión como se verá luego. Sin embargo para el archivo que contiene sólo A's el resultado es mucho mejor para la codificación aritmética. La justificación está

en que los códigos prefijos minimales necesitan emplear al menos un bit, por lo que el porcentaje es precisamente el entero más cercano a 12.5% = 1/8, ya que el código ASCII usa 8 bits para representar cada símbolo.

La codificación aritmética puede usar un número fraccionario de bits por símbolo, como se vió en el tercer capítulo. Por lo tanto puede bajar de la cota de 1 bit/símbolo para el archivo que contiene sólo A's, superando con ello por bastante margen a los códigos prefijos minimales en este caso e igualando el porcentaje alcanzado por el algoritmo LZW.

En la última columna se puede apreciar que el algoritmo LZW produce mejores grados de compresión para todos los archivos excepto para el que contiene imágenes. LZW trata de encontrar patrones empezando por los más cortos. De esta manera se obtienen buenos resultados con programas fuente y con texto escrito en español.

6.3.2 Análisis sobre los tipos de datos

El problema con el archivo *imagen*, que contiene imágenes digitalizadas, es que los patrones son demasiado largos y el algoritmo no busca el óptimo para ver repeticiones de ellos sino que se conforma con el primer patrón más largo que encuentre. Además el diccionario que utiliza para la decodificación es finito y es probable que el grado de compresión se degrade cuando se llena y entonces ya no se puedan agregar más entradas de códigos nuevos. Lo anterior sugiere que una posible mejora sería buscar patrones de manera óptima, pero ello constituye un problema NP-completo por la cantidad combinatoria de posibilidades para analizar [Lel88], como se dijo en el capítulo 3. Otra posible mejora sería descartar los códigos usados menos recientemente (Least Recently Used) o los menos frecuentemente usados (Least Frequently Used), pero ello implicaría eliminar la simplicidad del algoritmo pues se tendría que guardar y actualizar una fecha/hora o una frecuencia por cada palabra en el diccionario.

Para el archivo *bc* los resultados con los cuatro algoritmos es casi el mismo. Lo anterior se debe a que un programa en código ejecutable tiene en general todos los símbolos del código ASCII, mientras que en un archivo plano de texto como los primeros de la tabla sólo se hallan un poco más de 90 símbolos con frecuencias no parecidas. Lo anterior resulta de que para un archivo de texto sólo se utilizan los siguientes caracteres de control: para indicar fin de línea (10,13) en MS-DOS, ó 10 en UNIX y el 26 =^Z para el fin de archivo. De ahí en adelante únicamente se emplean símbolos del código ASCII que están entre el 32 y el 127. Del 128 en adelante se clasifican

como ASCII extendido y no se usan. La redundancia en *bc* es menor pues las repeticiones (frecuencias) de los símbolos no están muy alejadas unas de otras. En general se puede decir que los archivos más difíciles para comprimir son los que contienen código objeto.

Como observación nótese que los archivos *aa* y *bc* presentan dos extremos desde el punto de vista teórico. En *aa* hay una redundancia muy alta, pues cada A dentro del archivo proporciona poca información. Por otro lado en *bc* hay una redundancia baja pues ningún método logra bajar su tamaño del 78 %, y este grado de compresión es casi idéntico para todos los métodos.

6.4 Tiempo de ejecución

6.4.1 En compresión

A continuación se muestran los tiempos que se tarda el proceso de compresión para cada archivo medido en segundos. Como el proceso de codificación es el mismo para el método de Shannon-Fano y el método de Huffman, y la longitud promedio de los códigos es muy parecida, los tiempos de compresión son casi iguales. El tiempo mostrado corresponde a ambos códigos prefijos.

Tabla 6.3

Archivo	Prefijos	Aritmética	LZW
aa	28	67	8
bc	102	312	46
c	57	159	18
imagen	41	22	23
pas	27	75	8
tex	30	83	9

Cantidades en segundos

Sin comparación el algoritmo que produce mejores tiempos de respuesta es el LZW, en el cual es determinante la tabla de dispersión que se utiliza para manipular códigos. En desempeño le sigue el algoritmo de codificación para códigos prefijos minimales. En en este caso, el acceso a los códigos generados es inmediato pues se usa un arreglo en el que la indexación de una entrada se hace a través de cada símbolo leído. Hay cierto gasto de tiempo en este caso por hacer la conversión de cadena a bits por lo que se degrada algo el tiempo de ejecución.

El más lento de los tres es la codificación aritmética. En este caso el controlar la precisión de la computadora a nivel de bit representa un costo muy alto para obtener en el mejor de los casos un punto más de porcentaje de compresión, y aún así quedar atrás del algortimo LZW en muchos casos.

No hay que olvidar que tanto la compresión con códigos prefijos minimales y la codificación aritmética requieren dar dos pasadas sobre cada archivo, una primera para conocer la frecuencia de cada símbolo, y una segunda para poder hacer la codificación. Se podrían suponer ciertas distribuciones probabilísticas, pero ello degradaría siempre el grado de compresión. Como se había dicho anteriormente quizá lo mejor sería guardar libros de códigos (codebooks) en donde cada página fuera una tabla de códigos con distribuciones promedio según el tipo de datos.

6.4.2 En descompresión

La tabla siguiente muestra los tiempos que lleva el proceso de descompresión para cada archivo medido en segundos. Como el proceso de decodificación es el mismo para el método de Shannon-Fano y el método de Huffman, y la longitud promedio de los códigos es muy parecida, los tiempos de compresión son casi iguales el tiempo que se muestra es para ambos tipos de códigos prefijos.

Tabla 6.4

Archivo	Prefijos	Aritmética	LZW
aa	67	456	4
bc	676	881	22
c	336	750	10
imagen	282	604	12
pas	158	327	5
tex	171	363	5

Cantidades en segundos

En este caso los tiempos de descompresión para los códigos LZW son mucho menores que los de la codificación aritmética y los códigos prefijos. Nuevamente el uso de técnicas de dispersión produce excelentes tiempos de acceso a los códigos. El costo de buscar códigos en los otros dos algoritmos de decodificación está en la búsqueda del código prefijo, o de la frecuencia acumulada mayor o igual calculada, que indicará cuál es el siguiente símbolo decodificado. Aunque ambas búsquedas son binarias aprovechando el orden

de los códigos o de las frecuencias acumuladas se tiene un costo $O(log_2(m))$, donde $0 \leq m \leq 256$, para cada símbolo decodificado.

Analizando la eficiencia de las rutinas para códigos prefijos minimales, Shannon-Fano y Huffman, se puede decir lo siguiente. Hay que notar que aún cuando conlleva cierto gasto de tiempo (overhead) convertir del tipo cadena a bits y viceversa ello no influye demasiado en el tiempo de respuesta pues C++, y en particular C, son muy eficientes para manipular cadenas por la gran optimización que logran los compiladores aprovechando el uso de apuntadores.

Dentro del proceso de decodificación de códigos prefijos y códigos aritméticos no hay que hacer de nueva cuenta dos pasadas sobre el archivo ya que las frecuencias de los símbolos se encuentran almacenadas en la cabecera (header) del archivo comprimido.

Capítulo 7

Conclusiones y perspectivas

7.1 Conclusiones generales

El uso de la compresión es habitual en sistemas de telecomunicaciones, que hoy en día utilizan en gran medida la red telefónica mundial, los radiotransmisores y los satélites. Algunos sistemas operativos y manejadores de bases de datos la incorporan de manera transparente al usuario final para el almacenamiento de datos. Existen muchos sistemas de respaldo que guardan los datos en un formato comprimido. Sobre todo crece su uso porque ahora la información no abarca únicamente texto, bases de datos o programas, sino también audio y video. El video resulta de particular interés pues es el que utiliza mayor espacio en la práctica. Es importante resaltar que existen pocos estándares, lo que ha retardado la extensión de su uso. Por ejemplo, en ambiente de computadoras personales el formato de los archivos creados por el programa *pkzip*, de Pkware, es un estándar de facto, del mismo modo que lo es el formato empleado por el programa *gzip*, de la FSF, en ambiente UNIX.

El enfoque empleado para el estudio de la compresión se hizo desde el punto de vista de las comunicaciones de datos. La compresión de datos es un problema complejo. Para su estudio es necesario aplicar conocimientos de varias disciplinas de las Matemáticas y la Ciencia de la Computación. Desde un punto de vista teórico plantea aplicar conocimientos de teoría de la información, probabilidad, estadística, análisis de algoritmos, lenguajes formales, estructuras de datos y análisis numérico. Mientras que desde un punto de vista práctico implica aplicar conocimientos de ingeniería de software, arquitectura de computadoras, sistemas operativos y lenguajes de programación.

Los seis métodos expuestos plantean maneras distintas de tratar de eliminar la redundancia de un archivo. El método Run-Length simplemente cuenta las repeticiones contiguas de cada símbolo. Desgraciadamente ello produce resultados buenos únicamente para comprimir imágenes binarias o de pocos colores. La compresión diferencial se fija únicamente en las diferencias entre imágenes consecutivas. El método de Shannon-Fano y el de Huffman construyen códigos prefijos minimales basados en la distribución probabilística de los símbolos de la cadena fuente. Por lo anterior los algoritmos de codificación y decodificación son los mismos para ambos. La codificación aritmética por otro lado, aunque también se basa en las frecuencias de los símbolos, trata de representar a la cadena fuente como un número real en el intervalo $[0, 1)$. Por último, el algoritmo LZW trata de generar códigos LZ que correspondan a los patrones que se repiten más. El enfoque es novedoso pues no se utilizan en absoluto las probabilidades de aparición de los símbolos sino que el algoritmo (adaptivo) va aprendiendo de la cadena fuente a medida que va leyendo. Todos los demás métodos de compresión existentes en la actualidad son una combinación o variación de los métodos estudiados en esta tesis. Existen en particular variaciones de los códigos LZ que junto con los códigos de Huffman permiten obtener mejores grados de compresión pero perdiendo rapidez y simplicidad.

La programación orientada a objetos proporciona una perspectiva diferente para modelar la compresión. El modelo de clases diseñado es simple pero permite mantenimiento futuro para programar nuevos algoritmos o hacer modificaciones. Las clases genéricas son particularmente útiles para evitar reprogramar.

El lenguaje C++ resultó ser una herramienta ideal para programar los algoritmos gracias a que combina conceptos de orientación a objetos, las estructuras de control clásicas de un lenguaje procedural así como recursos para hacer operaciones de bajo nivel que son típicas de un lenguaje de máquina o ensamblador. Los métodos constructores y los valores por omisión para los parámetros evitan preocuparse por detalles de inicialización. Las funciones en línea, los operadores de bit y los apuntadores son de gran ayuda para crear un programa eficiente. Los parámetros por omisión dan gran flexibilidad para no alterar código ya programado, o que el programa sepa qué hacer en situaciones inesperadas. Todo el programa pudo haber sido desarrollado en lenguaje C, pero el desarrollo hubiera sido más lento dada la baja tipificación que tiene. El haber usado cualquier otro lenguaje hubiera hecho al programa no portable. Un lenguaje ensamblador sólo hubiera permitido programar uno de los métodos en un tiempo razonable. En contra de C++

está que no es un lenguaje OO puro y que es de difícil lectura y depuración.

7.2 Mejoras al programa

De acuerdo a los resultados experimentales es fácil darse cuenta porqué el 80 % de los programas comerciales de compresión usan los códigos LZ. El algoritmo LZW es rápido y el grado de compresión que logra en general es excelente. Sin embargo no aprovecha la distribución de probabilidad de los patrones. El combinarlo con otro método, por ejemplo usando el método de Huffman como filtro, produce mejores resultados [Lel88].

Por el momento el programa desarrollado no puede recuperarse de una anomalía, tal como perder una parte de un archivo codificado o detectar/corregir errores. Los códigos prefijos minimales son autocorrectivos, es decir, pasado cierto número de códigos después de un error se vuelve a decodificar correctamente. Con la compresión aritmética el problema es serio pues al haber un error por cambio o pérdida de un bit provoca que todos los siguientes bits produzcan una decodificación incorrecta. Por último, los códigos LZ no producen errores si sólo se cambian unos bits pues se emplean códigos de tipo bloque, es decir cada uno de ellos es de la misma longitud; sin embargo, un desfasamiento provocado por algún bit de más o de menos da como resultado una decodificación incorrecta hasta el final del archivo comprimido. Un mecanismo de verificación por códigos de redundancia cíclica (CRC) podría permitir recuperar parte de la información o corregir cambios de valores de bits. De hecho algunos programas comerciales de compresión incorporan esta característica de manera automática.

Otra posible mejora es que el programa determinara automáticamente qué método conviene usar, pero ello no podría determinarse a priori con los códigos LZ por ejemplo ya que se construyen de manera dinámica.

7.3 Direcciones de futura investigación

Aunque la compresión tiene mucho tiempo de ser estudiada sigue siendo un tema de investigación. Hay cotas que no se conocen cuando se aplica la compresión a determinado tipo de datos. Su aplicación crece sobre todo dentro del ambiente multimedia.

Aunque la compresión con pérdida (lossy) no fue estudiada en esta tesis, promete ser un área de investigación, sobre todo para imágenes y voz. Los

algoritmos que explotan información local permiten lograr grados de compresión mayores, pero son más complejos y lentos.

La susceptibilidad al error de los algoritmos es un tema que no ha sido investigado en profundidad. En particular, el detectar y corregir errores dentro de la compresión aritmética y el algoritmo LZW son aspectos que merecen atención dada la baja tolerancia que tienen estos algoritmos a la presencia de errores. No hay que olvidar que los mecanismos de detección y corrección de errores requieren aumentar la redundancia de la cadena codificada, por lo cual el porcentaje de compresión se verá degradado.

La demanda de aplicaciones en tiempo real demanda que los algoritmos programados sean muy eficientes. El explotar el paralelismo que ofrecen de manera natural los algoritmos de compresión es un tema de investigación en la actualidad. En particular los códigos LZ permiten lograr una paralelización sencilla utilizando multiprocesamiento simétrico [Gon85].

Bibliografía

Aho83: Aho, A., Hopcroft, J., Ullman, J., *Data structures and algorithms*. Addison-Wesley, 1983.

Bel89: Bell, T., Witten, I., Cleary, J., Modeling for Text Compression. *ACM Computing Surveys*, Vol.21 No. 4, 1989.

Boo91: Booch, Grady, *Object Oriented Design with Applications*. The Benjamin/Cummings Pub. Comp., 1991.

Cap85: Cappellini, V., *Data Compression and Error Control Techniques with Applications*. Academic Press, London, 1985.

Cap86: Capocelli, R.M., Giancarlo R., and Taneja, I.J. Bounds on the redundancy of Huffman Codes. *IEEE Transactions on Information Theory*, 32-6, 1986.

Con73: Connell, J.B. A Huffman-Shannon-Fano code. *Proc. IEEE*, 61,7, 1973.

Dei80: Deitel, H.M. *An introduction to Operating Systems*. Addison-Wesley, 1980.

Dud95: Duda, A., Distributed Multimedia Systems. *CNRS-IMAG*, Grenoble, Francia, 1995.

Eli75: Elias, P., Universal codeword sets and representation of the integers. *IEEE Transactions of Information Theory*, 21, 2, 1975.

Gla76: Glassey, C.R. and Karp, R.M. On the optimality of Huffman Trees. *SIAM J. Appl. Math.*, 31, 2, 1976.

Gon77: González, R.C., and Witnz, P., *Digital Image Processing*. Addison-Wesley, Reading, Mass, 1977.

Gon85: González, M.E., and Storer, J.A., Parallel Algorithms for Data Compression, *Journal of the Association for Computing Machinery*, Vol. 32, No. 2, 1985.

Gri85: Grimaldi, R., *Discrete and Combinatorial Mathematics. An Applied Introduction.* Addison-Wesley, 1985.

Ham80: Hamming, R.W., *Coding and Information Theory.* Prentice-Hall, Englewood Cliffs, 1980.

Huf52: Huffman, D.A. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 40,9, 1952.

Ing71: Ingels, F.M., *Information and Coding Theory.* Intext, Scranton, Pennsylvania, 1971.

Kar61: Karp, R.M., Minimum redundancy coding for the discrete noiseless channel. *IRE Transactions on Information Theory*, 7, 1, 1961.

Kat94: Katrib, Miguel, *Programación orientada a objetos en C++.* Infosys, 1994.

Lan83: Langdon, G.G. and Rissanen J.J., A double-adaptive file compression algorithm. *IEEE Trans. Comm.*, 31,11, 1983.

Lel88: Lelewer, D., and Hirchsberg, D. Data Compression. *ACM Computing Surveys*, Vol. 19, No. 3, 1986.

Lem79: Lempel, A., Criptology in Transition. *ACM Computing Surveys.* Vol. 11, No. 4, 1979.

Man83: Mano, M. M., *Computer System Architecture.* Prentice Hall, 1983.

Mey88: Meyer, B., *Object Oriented Software Construction.* Prentice Hall, 1988.

Mey92: Meyer, B., *Eiffel The Language.* Prentice Hall, 1992.

Per75: Perl, Y., Garey M.R., Efficient generation of optimal prefix code: equiprobable words using unequal cost letters. *Journal of ACM*, 22, 2, 1975.

Ral93: Ralston, A., Reilly, E., *Encyclopedia of Computer Science*, Third edition. Van Nostrand Reinhold, 1993.

Reg81: Reghbati, H.K., An overview of data compression techniques. *Computer*, 14, 4, 1981,

Ris76: Rissanen, J. J., Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, Vol. 20, 1976.

Ris83: Rissanen, J. J., A universal data compression system. *IEEE Transactions on Information Theory*, 25, 6, 1983.

Rub79: Rubin, F., Arithmetic stream coding using fixed precision registers. *IEEE Trans. on Information Theory*, 25,6, 1979.

Rut72: Ruth, S.S. and Kreutzer P.J., Data Compression for large Business Files. *Datamation*, 18,9, 1972.

Sha49: Shannon, C.,and Weaver, W. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, 1949.

Str91: Stroustrup Bjarne, *The C++ programming language*. Addison-Wesley, 1991.

Tan81: Tanembaum, A.S., *Computer Networks*. Prentice Hall, 1981.

Wil71: Wilkins, L.C., Wintz, P.A., Bibliography on data compression, picture properties and picture coding. *IEEE Transactions on Information Theory*, 17, 2, 1971.

Wit87: Witten, I.H., Neal, R.M. and Cleary, J.G., Arithmetic coding for data compression. *CACM*, 30, 6, 1987.

Ziv77: Ziv, J. and Lempel, A., A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23,3, 1977.