UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO
ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES ACATLAN

7
24

FALLA DE ORIGEN

"Análisis Comparativo de algunas Técnicas de Compresión de Datos dentro de los Sistemas Computacionales"

Trabajo de Tesis Profesional para Obtener el Título de la Licenciatura en Matemáticas Aplicadas y Computación

José Luis Cerda Morales
No. de Cta. 9058643-7
Generación 90-94
Asesor: Ing. Emiliano Llano Díaz
Santa Cruz Acatlán, Julio, 1995

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Para papá y mamá

Quienes sembraron en mi la sed de superación...
... y me enseñaron a luchar empecinadamente por conseguirla.
Mi corazón y admiración por siempre.

Para Miriam...

... Porque contigo
concluyó un andar solitario;
ha iniciado un ascenso de dos.
Te amo.

Para mis hermanos Soraya, Norma y Alex ...

... grandiosos guerreros que abren caminos;
siempre dispuesto a luchar a su lado.

Para Laury, Toyis, Luis y Amalia...

... mimás grande cariño.

AGRADECIMIENTOS

A mis padres
A Miriam Parra
A mi gran amigo David Martinez (El chefo)
A Felipe Rodriguez
A Jesús y Raquel Salinas
A mi asesor Emiliano Llano
A Noel Melgar Selvas
A mis maestros

Gracias por las cuerdas y los arneses;
ahora puedo escalar otra montaña.

CONTENIDO

INTRODUCCION **1**

El Problema 1
Hipótesis y Objetivo 2
Estructura 3

CAPITULO I

LA COMPRESION DE DATOS, SU LEXICO E HISTORIA. **5**

Los Dos Reinos 5
Tipos de Redundancia de Datos 6
Medida del Desempeño de la Compresión 7
Efectividad de la Compresión de Datos 8
Compresión de Datos = Modelado + Codificación 8
La Era del Amanecer 9
Codificado 10
Una Mejoría 10
Modelado 11
Modelado Estadístico 11
Esquemas de Diccionario 13
Ziv y Lempel 13
LZ77 13
LZ78 14
Compresión Irreversible 14
Programas para Conocer 15
Equipos 16

CAPITULO II

LOS PROCESOS Y ALGUNAS TECNICAS DE COMPRESION DE DATOS. **18**

Abreviaciones 18
Supresión Nula y Codif. de Longitud Corrida 18

Substitución de Patrones 19
Compresión Diferencial 20
Shannon-Fano 20
El algoritmo 21
El Algoritmo Huffman 22
Codificación Adaptativa 24
Actualizando el Árbol Huffman 25
Codificación Aritmética 27
Cuestiones Practicas 29
Compresión Basada en Diccionario 31
ARC: El padre de la Compresión por Diccionario 32
Compresión por Ventana Deslizante 33
Compresión LZSS 35
Voraz vs. el Mejor Posible 35
Compresión LZ78 36
Compresión LZW 38

CAPITULO III

EVALUACION DE TECNICAS Y RESULTADOS. 41

Resultados de Compresión 41
Modelado para la Compresión de Texto 42
Una Evaluación Empírica de Métodos de Codificación
para Alfabetos Multisímbolos 45
Plataforma de Prueba Experimental 45
Codificación Semi-Estática 46
Códigos Fijos 46
Codificación Huffman 47
Codificación Aritmética Exacta 47
Codificación Aritmética Aproximada 48
Descomposición Binaria 49
Resultados 49
Codificación Adaptativa 52
Estructuras de Datos para Codificación
Adaptativa Huffman 52
Estructuras de datos para Codificación
Aritmética Adaptativa 52
Codificación de Árbol Extendido 52
Probabilidad Escape 53
Resultados 53

Estadísticas para Programas de Compresión 55
Compresión Al Vuelo 59
Selección del Algoritmo de Compresión 61

CAPITULO IV

CONSIDERACIONES Y PROPUESTAS. 62

Ventajas y Desventajas de la Compresión de Datos 62
Métodos Formales y Técnicas de Compresión/Descompresión 65
¿Está Muerta la Codificación Huffman? 65
Compresión por Diccionario para Verificadores de Ortografía .. 66
Modelado por Modelo Multiplicativo: Una Aplicación
a la Codificación Aritmética 67

CONCLUSIONES 69

REFERENCIAS Y FUENTES DOCUMENTALES 73

GLOSARIO 74

Introducción

Considero que la compresión de datos, además de apasionante y distinguible por algunas otras características, es un tema actual, complejo, altamente dinámico y de inconmensurables aplicaciones en diversos campos de la realidad actual y del futuro inmediato. Me decidí a abordar este tema de tesis profesional para obtener el Título de Licenciado en la carrera de Matemáticas Aplicadas y Computación, debido a que desde que inicié mis estudios me persiguió una inquietud nacida al observar un documental televisivo en el que se refirió la compresión de señales efectuada durante la transmisión. Debido a la gran amplitud del tema, en el desarrollo de esta tesis se enfoca la compresión de textos y no la de señales, llevando la intención de ofrecer a estudiosos interesados en la materia, un texto abreviado de consulta que auxilie al desarrollar e implantar sistemas de computo o de comunicación vía módem.

Actualmente se está dando un crecimiento indiscriminado en los volúmenes de los paquetes de aplicación y de la información que procesan. Esto provoca el que, para utilizar las versiones de vanguardia, se tenga que invertir en aumentar la memoria RAM de los equipos, ó en su defecto, definitivamente adquirir equipo mas sofisticado, y esto, no siempre es económicamente posible. Se origina así, un círculo vicioso en el que los usuarios buscan actualizar sus equipos casi inmediatamente después de obtener paquetería nueva. Luego, cuando tienen equipo nuevo, quieren programas con más características. Para tratar de sobrellevar esta situación que se prevé durará hasta que surja otra revolución tecnológica en los sistemas de cómputo, es que se esta adoptando la utilización de las técnicas de compresión de datos en el almacenaje y transmisión de la información. Estas técnicas han tenido su desarrollo principalmente en los Estados Unidos y en algunos países de Europa se esta empezando a aplicar con gran éxito, pero en México, realmente se ha hecho poco por su difusión, identificación y comparación para su adecuada adaptación a los protocolos de almacenaje y transmisión de información. Desde mi punto de vista, éste es una tarea indicada para un Licenciado en Matemáticas Aplicadas y Computación.

El Problema.

Como ya se mencionó antes, la necesidad de utilizar las técnicas de compresión de datos ha ido en aumento al igual que su desarrollo, pero probablemente por su desconocimiento dentro del medio, por oposición al cambio o inclusive por comodidad, su aplicación se determina mas bien por el uso de paquetes comerciales de compresión que no necesariamente son inadecuados, pero si faltos de

orientación y de fundamentación hacia propósitos específicos. Es de esperarse un aumento en las ventajas de las técnicas cuando se orienten hacia una aplicación bien determinada, sin embargo, nos encontramos que por lo menos en México hay muy poca información disponible respecto a las técnicas, y no se diga en lo que respecta a su comparación o identificación de las mismas para trabajar en un ambiente determinado. Es por esto que puede enunciarse el problema científico siguiente:

SE HA AVANZADO POCO EN LO QUE RESPECTA A LA COMPARACION E IDENTIFICACION DE LAS TECNICAS DE COMPRESION DE DATOS PARA SU ADECUADA APLICACION DENTRO DE LOS SISTEMAS COMPUTACIONALES Y DE COMUNICACION, A PESAR DE, QUE SE HAN DESARROLLADO ALGORITMOS DE COMPRESION DE DATOS QUE COMBINAN DISTINTAS TECNICAS CON LO QUE LOGRAN ALCANZAR UNA ALTA EFICIENCIA.

Este problema es acentuado por algunas costumbres o factores de la compresión de datos que determina el que las técnicas no sean comúnmente usadas y aceptadas en el medio computacional como son:

* *Limites de Compresión Incomprendidos.* Los diseñadores de bases de datos típicamente subestiman, por causa de carencia de conocimiento, el monto de compresión que es posible para una base de datos dada. Asimismo, las consecuencias y ventajas de la compresión de datos no son completamente apreciadas.

* *Complejidad del Sistema.* La compresión de datos aumenta una capa de complejidad al diseño, implementación, y operación de un sistema de base de datos. Los diseñadores son renuentes a aceptar la complejidad adicional sin obtener a cambio claros y substanciales beneficios.

* *Mística Matemática.* Mucha de la literatura disponible "habla alrededor" de técnicas de compresión de datos individuales, sumergiéndolas en una mística matemática. Los diseñadores usualmente evitan esas áreas en las cuales ellos se sienten incómodos.

Este problema debe de ser enfrentado, y un modesto pero entusiasta intento de contribución a tal propósito lo constituye éste trabajo de tesis.

La Hipótesis y el Objetivo.

Para tratar de solucionar el problema mencionado anteriormente, se plantea la siguiente hipótesis:

SI SE CONFRONTAN LAS CARACTERISTICAS CUALITATIVAS Y CUANTITATIVAS, ASI COMO EL AMBITO DE APLICACION DE LAS TECNICAS DE COMPRESION DE

DATOS EN UN ANALISIS COMPARATIVO DE ESTAS TECNICAS, ENTONCES SE IDENTIFICARAN SUS APLICACIONES MAS ADECUADAS DENTRO DE LOS SISTEMAS COMPUTACIONALES Y DE COMUNICACION.

Desafortunadamente la realidad práctica no nos permite prescindir del análisis comparativo de las características cualitativas (como son el tipo de modelado, de codificación y de descodificación) ni de las cuantitativas (como el radio, la cantidad de memoria utilizada y la velocidad de compresión) como una herramienta en la toma de decisiones al diseñar sistemas de bases de datos. Sería ridículo pretender que la comparación involucre a todas las técnicas existentes (nada más con considerar la enorme variedad de técnicas y las limitaciones de información descritas anteriormente se traduciría en un trabajo inútil) por lo que sólo se considerarán algunas fundamentales, con la intención de que sirvan como base para un posterior análisis de las técnicas de interés.

La hipótesis anterior nos plantea establecer el siguiente objetivo:

ELABORAR UN TEXTO DE INTRODUCCION A LA COMPRESION DE DATOS COMPUTARIZADA QUE PRESENTE EL ANALISIS DE ALGUNAS DE SUS TECNICAS BAJO UN ENFOQUE COMPARATIVO-DESCRIPTIVO Y QUE CONSTITUYA UNA AYUDA PARA LA ADECUADA IDENTIFICACION, ELECCION, O IMPLEMENTACION DE LAS MISMAS, DENTRO DE LOS SISTEMAS COMPUTACIONALES Y DE TRANSMISION.

La Estructura.

Este trabajo consta de cinco capítulos cuya organización va generalmente paralela a la progresión histórica de la compresión de datos comenzando con la "era del amanecer" alrededor de 1950 subiendo hasta el presente.

El Capitulo I es un capitulo de referencia que intenta establecer el léxico fundamental de la compresión de datos. En él se discute el nacimiento de la teoría de la información, y se introducen una serie de conceptos, términos, y teorías usadas a lo largo del trabajo. Adicionalmente el capítulo I discute la diferencia entre modelado y codificación.

En el Capitulo II se presentan los algoritmos de algunas de las principales técnicas de compresión de datos, organizadas de manera paralela a su progresión histórica desde la década de los 50's hasta el trabajo presente, estableciendo sus cualidades principales.

A lo largo del Capitulo III se discuten las características cuantitativas de los algoritmos, manejándose análisis empíricos para su comparación y evaluación. Se proporcionan los métodos de evaluación y se plantean algunas recomendaciones y consideraciones para su implementación. Adicionalmente se proporciona un algoritmo para la elección de las técnicas de compresión.

Ya en el Capitulo IV se explican algunas propuestas para el adecuado manejo y desarrollo de las técnicas de compresión de datos de una manera formal, así como otras aplicaciones o mejoras a algunas de las técnicas y desarrollos viables a futuro.

Finalmente, se formulan las principales conclusiones derivadas de los temas abordados en los distintos capítulos.

CAPITULO I

La Compresión de Datos, su Léxico e Historia

A pesar del hecho de que el costo de la memoria de las computadoras ha decrecido dramáticamente en los últimos años, el almacenamiento de datos permanece, y probablemente siempre continuará, como un importante factor de costo para muchas aplicaciones de bases de datos a gran escala. Comprimir los datos dentro de un sistema de base de datos es atractivo por dos razones: la reducción del almacenamiento de datos y la mejoría en su desempeño. La reducción del almacenamiento es un beneficio obvio y directo, mientras que la mejoría en el desempeño resulta del poder manejar pequeñas cantidades de datos, para cualquier operación particular en la base de datos.

Como cualquier otra disciplina científica o de ingeniería, la compresión de datos tiene un vocabulario que en un principio puede parecer abrumadoramente extraño para una persona no-iniciada al tema. Términos como compresión Lempel Ziv, codificación aritmética y modelado estadístico quedan en el aire en un imprudente abandono.

Mientras la lista de términos es tan larga como para hacer un glosario, el llegar a dominarlos no es tan desanimante como pareciera en un principio. Con un poco de estudio y algunas notas cualquier programador se podrá ubicar dentro del mundo de argumentos sobre las técnicas de compresión de datos.

Los Dos Reinos

Las técnicas de compresión de datos se pueden dividir en dos grandes familias: Reversible (lossless) y la Irreversible (lossy). La compresión de datos irreversible concede una cierta pérdida de exactitud a cambio de un gran incremento en la compresión, es decir, reduce el tamaño de la representación física de los datos, pero preserva solamente un subconjunto (que es considerado relevante) de la información original. Su principal desventaja es esa, después de la descompresión, la representación de datos original nunca puede ser reconstruida. La compresión irreversible prueba ser efectiva cuando se aplica a gráficas, imágenes y voz digitalizada. La mayoría de éstas técnicas irreversibles se pueden ajustar a distintos niveles de calidad ganando gran exactitud a cambio de una efectividad en la compresión menor. Hasta hace poco, la compresión irreversible había sido implementada primeramente usando circuitos dedicados. En los últimos años , poderosos programas de compresión irreversible han aparecido en las UPC de

escritorio, pero aún así el campo permanece dominado por las implementaciones de electrónica.

La compresión reversible consiste en aquellas técnicas que garantizan generar un duplicado exacto del conjunto de datos de entrada después del ciclo de compresión/expansión; es decir, toda la información es considerada relevante y la descompresión recobrará la representación de datos original. Este tipo de compresión se puede dividir en dos técnicas separadas: semánticamente independientes y semánticamente dependientes. Las semánticamente independientes se pueden usar en cualquier tipo de datos, con varios grados de efectividad y no usa ninguna información respecto al contenido de los datos. Las técnicas semánticamente dependientes dependen y están basadas, en el contexto y la semántica de los datos. La compresión reversible se usa al almacenar los registros de una base de datos, hojas de cálculo, o archivos de procesadores de palabras. En estas aplicaciones, la pérdida de un solo bit podría ser catastrófica. Las técnicas de compresión de datos discutidas aquí serán de éste tipo.

Tipos de Redundancia de Datos

La redundancia de datos ocurre cuando las cadenas de patrones de datos son predecibles y por lo tanto, carga una pequeña o nula "información nueva". Encontrar y explotar, está redundancia en las bases de datos es la base de cualquier técnica de compresión de datos. Hay cuatro tipos básicos de redundancia en las bases de datos, presentándose en algunas bases, una mezcla de estos tipos.

- *Distribución de Caracteres.*

Dentro de una cadena de datos típico, algunos caracteres son usados más frecuentemente que otros. Por ejemplo, dentro del juego de caracteres de ocho bits ASCII 256, cerca de tres cuartos tal vez nunca serán usados en un archivo específico. Consecuentemente, un promedio de solo dos bits, en vez de los ocho bits del paquete de datos, podrían ser usados, obteniéndose alrededor de un 75% de espacio recuperado. En el lenguaje Inglés, los caracteres usualmente ocurren en una distribución bien documentada, con la letra e y el "espacio" siendo las más populares. Al determinar la distribución de los valores de datos, se pueden desarrollar códigos de distribución específica que alojen los caracteres más frecuentemente usados para que sean codificados en un código pequeño y los caracteres usados menos frecuentemente, para que sean codificados en un largo y más elaborado código, todo con un requerimiento mínimo de almacenamiento.

- *Repetición de Caracteres.*

Dentro de una cadena de datos, pueden ocurrir repeticiones de un solo carácter. Usualmente la cadena puede ser compactada en dos campos, codificando el símbolo del carácter repetido dentro de un campo y el número de veces que el carácter se

repite en el otro. Dichas cadenas son infrecuentes en archivos de texto, siendo mayormente espacios en blanco. Pero en las bases de datos científicas y numéricas, prevalece mucho más esta repetición de caracteres, compuesta principalmente de cadenas de espacios en blanco o ceros en campos de datos sin usar, o en campos numéricos de alto orden. En los datos gráficos, estas cadenas se componen mayormente de largas series de espacios homogéneos.

- *Patrones de Alto Uso.*

Es similar a la distribución de caracteres pero con secuencias de caracteres. Los patrones reaparecerán relativamente con una alta frecuencia, por lo que pueden ser representados con pocos bits. Por ejemplo, en los textos, un punto seguido de dos espacios es más común que sus otras combinaciones posibles y por lo tanto pueden ser codificados usando unos cuantos bits. Los campos numéricos sólo contienen secuencias de dígitos, sin letras o símbolos especiales intermezclados. Estos pueden ser codificados en menos de cuatro bits por dígito, en vez de los cinco u ocho bits necesarios para el texto.

- *Redundancia Posicional.*

Ciertos caracteres o patrones, pueden aparecer consistentemente en un lugar en cada bloque de datos. Esto ocurre, por ejemplo, al rastrear datos o cualquier tipo de estructura de datos preformada donde una línea o representación gráfica ocurre en la misma posición cada vez. Aquí se puede desarrollar un código de compresión usando sólo el carácter redundante o símbolo y su(s) localización(es).

Medida del Desempeño de la Compresión

El más popular método pera medir el desempeño de una técnica de compresión es el radio de compresión CR:

$$CR = \frac{\text{tamaño de los datos sin compresión}}{\text{tamaño de los datos con compresión}}$$

donde la unidad del dato depende de la aplicación. Podría ser un carácter individual, una secuencia de caracteres, o un archivo entero. Dado el radio de compresión, la reducción esperada del tamaño de los datos esta dada por:

$$REDUC = (1 - 1 / CR) * 100\%$$

Efectividad de la Compresión de Datos

Como cualquier técnica de compresión, la efectividad de comprimir los datos no sólo depende del sistema, sino que depende de las características de los datos dentro del sistema. Las siguientes características de datos ofrecen un excelente ambiente para la aplicación de las técnicas de compresión de datos.

- *Esparcimiento.*

En las bases de datos esparcidas, los campos tienden a contener largos agrupamientos de ceros, espacios en blanco, o indicadores de datos perdidos. El esparcimiento es la más importante propiedad al determinar el porcentaje global de la compresión que podría ser ejecutada en una base de datos.

- *Distribución de Valores.*

En las bases de datos numéricas con un amplio rango de valores, los números tienden a tener una larga distribución, usualmente sesgada hacia la terminación más baja. Así muchos valores pueden ser almacenados usando, en promedio, más pocos bits que la máxima longitud de estos en la distribución.

- *Atributos de Acoplamiento.*

En bases de datos con muchos valores clave, la repetición de los valores clave en las tablas de datos usualmente toman la forma de un producto cruz e imponen una sobrecarga en el almacenaje, de naturaleza multiplicativa.

- *Frecuencia.*

En largas bases de datos textuales, caracteres de atributos alfanuméricos no ocurren con la misma frecuencia. En esto se le puede sacar ventaja a los códigos de redundancia.

Compresión de Datos = Modelado + Codificación

En general, la compresión de datos consiste en tomar un flujo de símbolos y transformarlos en códigos. Si la compresión es efectiva el flujo resultante de códigos será más pequeño que los símbolos originales. La decisión de sacar un cierto código de un cierto símbolo o conjunto de símbolos se basa en un modelo. El modelo es una simple colección de datos y reglas usadas para procesar los símbolos de entrada y determinar cual(es) código(s) sacar. Un programa usa el modelo para definir con precisión las probabilidades para cada símbolo y el codificador pueda producir un código apropiado basado en esas probabilidades.

Se considera al modelo y al programa de codificación como procesos diferentes debido a las innumerables formas de modelar los datos, pudiendo todas ellas usar el

mismo proceso de codificación para producir su salida. Un simple programa usando la codificación Huffman, por ejemplo, usaría un modelo que diera la probabilidad específica de ocurrencia de cada símbolo en cualquier parte del flujo de entrada. Un programa más sofisticado podría calcular la probabilidad basado en los últimos 10 símbolos del flujo de entrada. Aunque ambos programas usan la codificación Huffman para producir sus salidas, sus radios de compresión probablemente serían radicalmente diferentes.

La Era del Amanecer

La compresión de datos es tal vez la expresión fundamental de la Teoría de la Información. La Teoría de la Información es una rama de las matemáticas que tiene su génesis en los años 40s con el trabajo de Claude Shannon de los laboratorios Bell. El se cuestionó a cerca de la información, e incluyo distintas maneras de almacenar y comunicar mensajes.

La compresión de datos entra en el campo de la Teoría de la Información ya que involucra a la redundancia. La información redundante en un mensaje requiere de bits extra para codificar, y si se quiere estar excento de esa información extra, se tendrá que reducir el tamaño del mensaje.

La Teoría de la Información usa el término de entropía como una medida de cuanta información es codificada en un mensaje. Entre más alta sea la entropía de un mensaje, contendrá una mayor información. La entropía de un símbolo está definida como el logaritmo negativo de su probabilidad. Para determinar el contenido en bits de la información de un mensaje se expresa la entropía usando el logaritmo en base dos:

$$\text{Número de bits} = -\text{Log}_2\,(\text{probabilidad})$$

La entropía de un mensaje entero es simplemente la suma de las entropías individuales de todos los símbolos.

La entropía encaja con la compresión de datos en la determinación de cuantos bits de información están presentes en un mensaje. Si la probabilidad de que aparezca el carácter "e" en éste manuscrito es de 1/16, por ejemplo, el contenido de información en el carácter es de cuatro bits. Entonces la cadena "eeeee" tiene un contenido total de 20 bits. Si utilizamos caracteres standard de 8-bits ASCII para codificar éste mensaje, estaremos usando 40 bits. La diferencia entre los 20 bits de entropía y los 40 bits usados para codificar el mensaje es donde el potencial de la compresión de datos aumenta.

Un importante hecho acerca de la entropía es que a diferencia de la medida termodinámica de entropía, se pueden usar números no absolutos para el contenido de la información de un mensaje dado. El problema es que cuando se calcula la entropía se usa un número que da la probabilidad de un símbolo dado. La representación de probabilidad usada es la probabilidad para un modelo dado, no un número absoluto. Si se cambia el modelo, la probabilidad cambiará con él.

Se pueden ver claramente como cambian las probabilidades cuando se usan distintos ordenes con un modelo estadístico. Un modelo estadístico rastrea la probabilidad de un símbolo basado en los que aparecieron previamente en el flujo de entrada. El orden del modelo determina cuantos símbolos previos son tomados por el contador. Un modelo de orden 0, por ejemplo, no tomará símbolos previos. Uno de orden 1 verificará el carácter previo y así sucesivamente.

Para comprimir los datos bien, se necesita seleccionar modelos que predicen los símbolos con altas probabilidades. Un símbolo que tenga una alta probabilidad tiene un bajo contenido de información y se necesitarán menos bits para codificarlo. Una vez que el modelo está produciendo altas probabilidades, el siguiente paso consiste en codificar los símbolos usando un número apropiado de bits.

Codificado

Una vez que la Teoría de la Información hubo avanzado hasta donde el número de bits de información en un símbolo pudo ser determinado, el siguiente paso fue desarrollar nuevos métodos para codificar la información. Para comprimir datos, se necesita codificar los símbolos con el número exacto de bits de información que contiene el símbolo. Sí el carácter "e" sólo nos da cuatro bits de información, entonces deberá ser codificado con exactamente cuatro bits.

Al codificar caracteres usando EBCDIC o ASCII, claramente no vamos a estar muy cerca de un método óptimo. Puesto que cada carácter se codifica usando el mismo número de bits, se introducen muchos errores en ambas direcciones, con la mayoría de los códigos en un mensaje siendo muy largos y algunos siendo muy cortos. Solucionar éste problema de codificación de una manera razonable fue uno de los principales problemas atacado por los practicantes de la Teoría de la Información. Dos propuestas que trabajaron bien fueron el codificador Shannon-Fano y el codificador Huffman como dos diferentes formas de generar códigos de longitud variable cuando se da una tabla de probabilidades para una serie de símbolos dados.

El codificador Huffman, nombrado así por su inventor D.A. Huffman, ejecuta el mínimo monto de redundancia posible en un grupo fijo de códigos de longitud variable. Esto no significa que el codificador Huffman sea un método óptimo de codificación; significa que provee la mejor aproximación para codificar símbolos cuando están usando códigos de anchura fija.

El problema con el codificador Huffman o el Shannon-Fano es que utilizan un número entero de bits en cada código. Si la entropía de un carácter dado es de 2.5 bits, el codificador Huffman de ese carácter será de 2 o 3, mas no de 2.5; y es por esta causa que no puede ser considerado como un método de codificación óptima pero sí como la mejor aproximación que usa códigos fijos con un número entero de bits.

Una Mejoría. Si bien el codificador Huffman es ineficiente debido al uso de un número entero de bits por código, es relativamente fácil de implementar y muy eficiente para codificar y descodificar. Cuando Huffman publicó sus documentos sobre codificación en 1952, instantáneamente se convirtió en el documento más

citado de la Teoría de la Información y probablemente continúe siéndolo. El trabajo original de Huffman tuvo numerosas variaciones menores, y dominó el mundo de los codificadores hasta la década de los ochentas.

Como el costo de los ciclos de CPU fueron bajando, nuevas posibilidades para más eficientes técnicas de codificación emergieron. Una en particular, la codificación aritmética, es un viable sucesor del codificador Huffman.

La codificación aritmética es algo más complicada tanto en el concepto como en su implementación que los códigos standard de ancho variable. Esta no produce un código individual para cada símbolo. En véz de ello, produce un código para un mensaje entero. Cada símbolo que se agrega al mensaje incrementándolo modifica el código que sale. Esta es una mejoría ya que el efecto neto de cada símbolo sobre el código de salida puede ser un número fraccional de bits en vez de un número entero. Entonces si la entropía del carácter "e" es 2.5 bits, es posible sumar exactamente 2.5 bits en el código de salida, por lo que resulta en un codificador más adecuado.

La codificación aritmética requiere más poder del CPU de lo que se disponía hasta hace poco. Ahora inclusive sufrirá de una desventaja significativa en velocidad al compararse con los métodos antiguos de codificación. Pero la ganancia de cambiar a este método es suficientemente significativa para asegurar que la codificación aritmética será la alternativa cuando el costo de almacenamiento o de envío de información sea lo suficientemente alto.

Modelado

Si usamos una metáfora automotiva para la compresión de datos, el codificado sería las llantas, pero el modelado sería el motor. Sin contar la eficiencia del codificador, si no se tiene un modelo que provea de buenas probabilidades, no se comprimirán los datos.

La compresión de datos reversible generalmente es implementada usando dos diferentes tipos de modelado: estadístico ó basado en diccionario. El modelado estadístico lee y codifica un sólo símbolo a la vez usando la probabilidad de que ese carácter aparezca. El modelado basado en diccionario usa un sólo código para reemplazar las cadenas de símbolos. En el modelado basado en diccionario, el problema de codificado es reducido de manera significativa.

Modelado Estadístico. La forma más simple de modelado estadístico usa una tabla estadística de probabilidades. En los primeros días de la teoría de la información, el costo del CPU para analizar los datos y construir un árbol Huffman era considerablemente significativo, por lo que frecuentemente no se realizaban. En vez de eso, los bloques representativos de datos eran analizados una vez, dando una tabla de cuentas de carácter-frecuencia. Los árboles Huffman de codificación/descodificación fueron entonces construidos y almacenados. Los programas de compresión tenían acceso a éste modelo estadístico y comprimían los datos usándolo.

Pero usar un modelo estadístico universal tiene sus limitaciones. Si un flujo de entrada no encaja bien con las estadísticas previamente acumuladas, el radio de compresión será degradado posiblemente hasta el punto donde el flujo de salida llega a ser más largo que el flujo de entrada. Obviamente la mejora siguiente es la de construir una tabla estadística para cada flujo de entrada único.

Construir una tabla estadística Huffman para cada archivo a ser comprimido tiene sus ventajas. La tabla se adapta únicamente a ese archivo en particular, así que daría una mejor compresión que una tabla universal. Pero hay un obstáculo adicional desde que la tabla (o las estadísticas usadas para construir la tabla) tiene que pasar al descodificador a la cabeza del flujo de código comprimido.

Para una tabla de compresión de orden 0, las estadísticas actuales usadas para crear la tabla pueden tomar por lo menos 256 bytes que no es un monto muy grande de sobrecarga. Pero tratando de lograr una mejor compresión, el uso continuo de tablas de alto 'orden hará que las estadísticas que necesitan ser pasadas al descodificador crezcan a un rango alarmante. El sólo moverse a un modelo de orden 1 puede disparar las tablas estadísticas de 256 a 65536 bytes. Aunque los radios de compresión serán indudablemente mejores cuando se muevan al orden 1, la sobrecarga al pasar la tabla estadística probablemente eliminará cualquier ganancia.

Por ésta razón, la búsqueda de compresión en los últimos 10 años se ha concentrado en modelos adaptativos. Cuando se usa un modelo adaptativo, el dato no tiene que ser revisado de una vez para generar las estadísticas. En vez de ello, las estadísticas son continuamente modificadas tan pronto los nuevos caracteres son leídos y codificados.

El punto importante al hacer que éste sistema trabaje es que el proceso de actualización del modelo debe de trabajar exactamente de la misma forma, tanto para el programa de compresión como para el de descompresión. Después de que cada carácter (o grupo de caracteres) son leídos, son codificados o descodificados. Sólo después de haberse completado la codificación o descodificación, puede ser actualizado el modelo para tomar en cuenta los símbolos o grupo de símbolos más recientes.

Un problema con los modelos adaptativos es que ellos comienzan sin conocer esencialmente nada acerca de los datos. Así cuando el programa comienza, no hace un buen trabajo de compresión. La mayoría de los algoritmos adaptativos tienden a ajustarse rápidamente al flujo de datos y comenzarán turnándose en radios de compresión respetables después de sólo unos cuantos cientos de bytes. Asimismo, no le toma mucho a la curva de compresión de datos achatarse, así que leer más datos no mejora el radio de compresión.

Una ventaja que los modelos adaptativos tienen sobre los modelos estadísticos es la habilidad de adaptarse a las condiciones locales. Cuando se comprimen archivos ejecutables, por ejemplo, el carácter de entrada del dato puede cambiar drásticamente como el programa de archivo cambia de un programa en código binario a un dato binario. Un programa adaptativo bien escrito le dará más peso al dato más reciente que al dato viejo, así modificará sus estadísticas para adaptarse mejor al dato que cambió.

Esquemas de Diccionario. Los modelos estadísticos generalmente codifican un sólo símbolo a la vez, leyéndolo primero, calculando una probabilidad y entonces, generando un sólo código. Un esquema de compresión basado en diccionario usa un concepto diferente. Se lee en la entrada de datos y se buscan grupos de símbolos que aparezcan en un diccionario. Si encuentra una cadena que concuerde, un puntero o índice dentro del diccionario puede ser la salida en lugar del código para el símbolo. A más larga concordancia mejor radio de compresión.

Este método de codificación cambia el foco de la compresión por diccionario. Simples métodos de codificación son usados generalmente, y el foco del programa está en el modelado. En la compresión LZW, por ejemplo, códigos simples de ancho uniforme son usados para todas las substituciones.

Un diccionario estático se usa como la lista de referencias de un documento académico. A lo largo del texto de papel, el autor puede simplemente substituir un número que apunta a la lista de referencias en vez de escribir el título completo del trabajo referenciado. El diccionario es estático porque es construido y transmitido con el texto del trabajo (el lector no tiene que construirlo sobre el vuelo).

El problema con un diccionario estático es idéntico al problema que el usuario de un modelo estático estadístico encara: El diccionario necesita ser transmitido junto con el texto, resultando en una carga extra que se suma al texto comprimido. Un esquema de diccionario adaptativo ayuda a evitar este problema.

Mentalmente, nosotros usamos un tipo de diccionario adaptativo cuando realizamos reeplazamientos con siglas en la literatura técnica. La forma standard para usar este diccionario adaptativo es enunciar la frase, y luego poner su abreviación substituta entre paréntesis.

Ziv y Lempel

Hasta 1980, la mayoría de los esquemas generales de compresión usaron el modelado estadístico. Pero en 1977 y 1978, Jacob Ziv y Abraham Lempel describieron un par de métodos de compresión usando un diccionario adaptativo. Estos dos algoritmos originaron una inundación de nuevas técnicas que usaron métodos basados en diccionarios para obtener nuevos radios de compresión impresionantes.

LZ77. El primer algoritmo de compresión descrito por Ziv y Lempel es comúnmente referido como el LZ77. Es relativamente simple. El diccionario consta de todas las cadenas dentro de una ventana del flujo de entrada leído previamente. Un programa de compresión de archivos, por ejemplo, puede usar una ventana de 4K bytes como un diccionario. Mientras nuevos grupos de símbolos son leídos dentro, el algoritmo busca concordancias con las cadenas encontradas en los previos 4K bytes de datos ya leídos. Cualquier concordancia es codificada como indicadores enviados hacia el flujo de salida.

LZ77 y sus variantes forma algoritmos de compresión atractivos. El mantenimiento del modelo es simple; codificar la salida es simple; y programas que trabajan muy rápidamente pueden ser escritos usando LZ77. Programas populares

como el PKZIP y LHarc usan variantes del algoritmo LZ77, y ello tiene probada popularidad.

LZ78. El LZ78 toma una aproximación diferente a la construcción y mantenimiento del diccionario. En vez de tener una ventana de tamaño limitado en el texto precedente, LZ78 construye su diccionario fuera de todos los símbolos previamente vistos en el texto de entrada. Pero en vez de tener carta blanca al acceso a todos las cadenas de símbolos en el texto precedente, un diccionario de cadenas es construido un sólo carácter a la vez. La primera vez que la cadena "Mark" es visto, por ejemplo, la cadena "Ma" es sumada al diccionario. La siguiente vez, "Mar" es sumada. Si "Mark" es visto de nuevo, ésta es sumada al diccionario.

Este procedimiento incremental trabaja bastante bien y aísla frecuentemente cadenas usadas y las suma a la tabla. Al contrario de los métodos LZ77, las cadenas en LZ78 pueden ser extremadamente largas, lo que le permite radios de compresión altos. LZ78 fue el primero de dos algoritmos Ziv-Lempel de éxito popular, a la par de la adaptación LZW de Terry Welch, los cuales forma el coro del programa UNIX COMPRESS.

Compresión Irreversible

Hasta hace poco, la compresión irreversible ha sido realizada primordialmente para circuitos de propósito especial. El advenimiento de los chips Procesadores de Señal Digital (PSD) comenzaron el traslado de la compresión irreversible fuera del tablero de circuitos hacia el escritorio. Los precios de las UPC han bajado al punto de ser práctico el realizar compresión irreversible en PCs para propósito general.

La compresión irreversible es fundamentalmente diferente a la reversible respecto a: que acepta una leve pérdida de datos para facilitar la compresión. La compresión irreversible es generalmente hecha en datos analógicos almacenados digitalmente, con gráficas y archivos de sonido como sus aplicaciones primordiales.

Este tipo de compresión hace frecuentemente dos fases. Una primera fase sobre el dato, desempeña una función de procesamiento de señal de alto nivel. Esto frecuentemente consiste en transformar los datos en un dominio de frecuencia, usando algoritmos FFT. Una vez que los datos han sido transformados, son suavizados al redondear quitando los puntos más altos y más bajos. La pérdida de señal ocurre aquí. Finalmente, los puntos frecuentes son comprimidos usando técnicas reversibles convencionales.

La función de suavizado que opera en el dominio de frecuencia de datos generalmente tiene un "factor de calidad" interconstruido que determina que tanto suavizado ocurre. Mientras más friccionado es el dato, más grande pérdida de señal y una mayor compresión ocurrirá.

En el pequeño mundo de los sistemas, una tremenda cantidad de trabajo se está haciendo para la compresión de imágenes gráficas, tanto fijas como en movimiento. La Organización de Standards Internacional (OSI) y el Comité Consultivo para el Telégrafo y el Teléfono Internacional (CCTTI) se asociaron para formar dos comités: El Grupo Unido de Expertos Fotográficos (GUEF) y el Grupo de Expertos de Imágenes

en Movimiento (GEIM). El standard GUEF usa el algoritmo de la Transformada Discreta del Coseno (TDC) para convertir a imágenes gráficas el dominio de frecuencia. El algoritmo TDC ha sido usado para transformaciones gráficas por muchos años, por lo que la implementaciones eficientes son fácilmente disponibles. El GUEF especifica un factor de calidad de 0a 100, lo que permite que el compresor determine que factor seleccionar.

Usando el algoritmo GUEF en imágenes pueden resultar dramáticos radios de compresión. Con poca ó sin degradación, los radios de compresión de 5-10 por ciento son rutina. Aceptando una degradación menor se ejecutan radios de por lo menos 1-2 por ciento.

La implementación en programas de los algoritmos GUEF y GEIM permanece en disputa por ejecutar su funcionamiento en tiempo real. La mayoría del desarrollo de programas multimedia que usa este tipo de compresión sigue dependiendo del uso de un coprocesador para lograr que la compresión se lleve acabo en un tiempo razonable. Estamos a unos cuantos años de tener capacidades de compresión con sólo programas.

Programas para Conocer

Programas de compresión de propósito general han estado disponibles durante los últimos diez años. No se encontraban si no hasta alrededor de 1980 en que las máquinas con el poder de hacer el análisis necesario para la compresión efectiva comienzan a ser comunes.

En el mundo de Unix, uno de los primeros programas de compresión de propósito general fue COMPACT. COMPACT es una implementación relativamente íntegra de un programa de compresión de orden 0 que usa el codificador Huffman adaptativo. COMPACT produjo una compresión lo suficientemente buena para hacerlo útil, pero era lento. COMPACT fue además propietario del producto, así que no estuvo disponible para todos los usuarios de Unix.

COMPRESS, un programa mejorado, estuvo disponible para los usuarios de Unix unos años después. Es una íntegra implementación del esquema de compresión basado en diccionario LZW. COMPRESS daba una significativamente mejor compresión que COMPACT, y ejecutaba más rápido. Aún mejor, el código origen de COMPRESS estuvo disponible como un programa de dominio público, y probó ser absolutamente portátil. COMPRESS continua siendo de uso amplio entre los usuarios de Unix, aunque su uso continuo es legítimamente cuestionado por la patente de LZW mantenida por Unisys.

En los recientes 1980s, los usuarios de los sistemas CP/M y MS-DOS fueron expuestos en un principio a la compresión de datos a través del programa SQ. SQ realiza compresión de orden 0 usando un árbol estático Huffman en el archivo. SQ da una compresión comparable a la del programa COMPACT, y fue más ampliamente usado por los pioneros de las telecomunicaciones de oficina.

Como en el mundo de Unix, el codificador Huffman le dió pronto camino a la compresión LZW con el advenimiento de ARC. ARC es un programa de propósito general que realiza en un archivo tanto su compresión como su archivamiento, dos

características que seguido van de la mano. (Los usuarios de Unix típicamente archivan primero usando TAR, y luego comprimen el archivo completo.) ARC pudo originalmente comprimir los archivos usando codificación de longitud corrida, codificador estático Huffman de orden 0, ó la compresión LZW. El código original LZW para el ARC aparece como una derivación del código Unix COMPRESS.

Oportuna a la rápida distribución posible gracias al uso de shareware y a las telecomunicaciones, ARC rápidamente pasó a ser standard y comenzaron a cundir imitadores. El mundo de la computación de escritorio contiene ahora docenas de utilerías de compresión de datos, algunas comerciales, algunas de freeware y otras de shareware.

Hoy ARC es sometido a muchas revisiones mayores, y permanece como un programa comercial muy popular. Mientras no haya un corte claro en los standards de compresión, en el MS-DOS el título es sustentado probablemente por el programa Shareware PKZIP. PKZIP es un programa relativamente no muy caro que ofrece radios de compresión superiores y velocidad de compresión.

Dos fuertes competidores de la supremacía del PKZIP surgieron recientemente. El primero LHarc, viene del Japón, y presenta grandes ventajas sobre otros programas de compresión de archivos. Primero, el origen del LHarc está disponible libremente y ha sido portado por numerosos sistemas operativos y plataformas de equipos. Segundo, el autor de LHarc, Haruyasu Yoshizaki (Yoshi), ha concedido explícitamente el derecho de uso de su programa para cualquier propósito, personal o comercial.

El segundo competidor es ARJ de Robert Jung. ARJ está libre para uso no comercial y ha logrado realizar radios de compresión ligeramente mejores que los que puede ofrecer LHarc. ARJ además tiene un código origen portátil ANSI C para hacer extracciones de archivos ARJ.

Equipos

Sólo hasta recientemente se ha hecho mucho trabajo con el soporte de equipos para la compresión de datos. En cierta forma esto es debido a la opinión de muchos diseñadores de bases de datos de que la compresión de datos es de uso limitado, y a los sofisticados algoritmos usados en muchas técnicas de compresión de datos.

Hay varios posibles formatos que el soporte de equipos puede tomar. Una es incluir la compresión de datos, la encriptación, y algunas búsquedas clave idóneas dentro de una avanzada controladora de disco. Otra técnica usa un sistema de microprocesador asistido para descargar el proceso de compresión de datos desde una terminal de computadora a una estructura de minicomputadoras cada una ejecutando una función de administración de datos. Cuando la computadora anfitriona necesita datos, manda el requerimiento a éste sistema, el cual analiza el requerimiento y manda subrequerimientos a las minicomputadoras apropiadas. Las minicomputadoras hacen la lectura del disco actual, la descompresión y compresión de los datos, y mandan los datos requeridos por la computadora.

Varias investigaciones están usando tecnología VLSI para diseñar equipo de propósito especial para la compresión y descompresión de datos en tiempo real. Arquitecturas basadas en memoria y networks neurales se conjuntan con las técnicas standards de algoritmos para silicon. La evaluación del desempeño de estas implementaciones no está completa y necesita de búsqueda adicional.

CAPITULO II

Los Procesos y algunas Técnicas de Compresión de Datos

Una amplia de técnicas están disponibles hoy en día para la compresión de datos. Algunas de las técnicas más populares se discutirán a continuación.

Abreviaciones

Las abreviaciones tienden a encajar en una categoría de codificación de datos, mas que en una categoría de compresión de datos, a partir de que la conversión tiene lugar ó antes de que la base de datos es cargada al disco desde la memoria, ó durante la reorganización de los datos. La reducción de datos obtenible usando abreviaciones puede ser dramática, con reducciones típicas en el orden del 50-90%. Sin embargo, los procesos de codificación y descodificación requieren de una tabla de almacenamiento "de búsqueda" extremadamente grande si el número de posibles ítems de datos es considerablemente grande. Algunas veces, la descodificación misma es hecha manualmente, tablas en mano.

Aunque los radios de compresión pueden ser muy buenos con las abreviaciones, se debe tener gran cuidado cuando diferentes categorías de códigos de abreviación son situados en una sóla tabla (por ejemplo CANADA (CA) y CALIFORNIA (CA) tienen ambos la misma abreviación de dos caracteres). Asimismo, las abreviaciones sólo pueden ser usadas cuando los datos pueden ser predeterminados, o son estáticos, para que las abreviaciones puedan ser únicas. En el lado bueno, a diferencia de la mayoría de las técnicas de compresión, las abreviaciones están limitadas sólo por la imaginación del diseñador, así que hay un número sinfín de códigos que pueden desarrollarse.

Supresión Nula y Codificación de Longitud Corrida (Run-Length)

Supresión nula es un término genérico usado para las técnicas que suprimen ya sea ceros ó espacios en blanco. La técnica toma ventaja de los datos en los que los ceros o los blancos son predominantes. Generalmente, la supresión nula es simple y puede ser fácilmente implementada como una rutina genérica para su uso con muchos diferentes tipos de archivos de datos. Por otro lado, la supresión nula usualmente no realiza un radio alto de compresión de datos como algunas otras

técnicas (reducciones del 20-45%). Dos métodos básicos son usados para comprimir y almacenar los datos:

•Representar secuencias de ceros y blancos por un carácter especial, seguido por un número que indique la longitud de la secuencia. Debido a esto, las secuencias de uno o dos caracteres no son eficientemente representadas por ésta forma y son normalmente dejados sin cambios.

Dato Original: DUNNbbbbbbCbAbb450000b55
Dato Comprimido: DUNN#6CbAbb45@4b55

•Usar un mapa de bits anexado al principio de cada registro de datos, y finalizando en una palabra límite. Cuando es fijada un unidad de datos, ya sea una palabra o byte, las unidades que contengan valores de datos todos nulos o ceros pueden ser denotados por un bit-cero, y las unidades que contengan valores no nulos ó no ceros pueden ser denotados por un bit-uno. Por lo tanto, cada bit en el mapa de bits representa la unidad de dato de tamaño fijo escogida para la aplicación. Estos bits y sus posiciones corresponden exactamente a las posiciones de los caracteres y palabras dentro del registro mismo. Además, al menos que un indicador del tipo de campo y longitud sea anexado al mapa de bits, la compresión se limitará ya sea a nulos ó ceros, y no a ambos, dentro del registro de datos. El siguiente ejemplo asume unidades de dato de media palabra:

Dato Original: DUNNbbbbCb45bbbb
Dato Comprimido: Mapa de Bits: 11001100 Texto: DUNNCb45

Una técnica similar es la idea de la codificación de longitud corrida (run-length). La codificación de longitud corrida es muy efectiva en el procesamiento de imágenes, y bases de datos donde hay largas secuencias de ceros repetidos ó valores perdidos. Simplemente, la codificación de longitud corrida reemplaza las secuencias de valores idénticos por un contador de campo, seguido por un identificador para los valores repetidos. El contador de campo deberá ser abanderado para que pueda ser reconocido de otros valores de datos. Y como anteriormente, la secuencia seleccionada debe contener suficientes valores repetidos para garantizar su reemplazamiento por los campos del contador y del carácter.

Substitución de Patrones

En esta técnica, un algoritmo examina la base de datos entera, o el texto del archivo, buscando patrones comunes de dos o más caracteres que ocurran frecuentemente, y substituye un patrón no usado por aquel común más extenso. Al mismo tiempo un diccionario de substitución (tabla "de búsqueda") es creado y/o actualizado. La eficiencia puede ser un serio problema en ésta técnica puesto que muchas comparaciones pueden hacerse entre el diccionario y los datos para identificar un patrón. Además, los patrones pueden ser de varias longitudes, y pueden

ser subconjuntos de otros patrones. La substitución de patrones generalmente adquiere más grandes radios de compresión que la supresión nula puesto que los patrones, sumados a las cadenas de ceros y blancos, contribuyen a la compresión. Aquí las compresiones son del orden del 35-50%.

Dato Original: CED3690000BB52X0 CED3700000BB86X0
Dato Comprimido: #369?/52X0 #370?/86X0
Patrones: #=CED ?=0000 /=BB

Compresión Diferencial

La codificación de compresión diferencial envuelve el reemplazamiento de una secuencia de caracteres con un valor de código que define su relación, ya sea con una secuencia previa de caracteres similares, o con una secuencia específica. Debido a esto la compresión de éste tipo solo puede ser usada en aplicaciones cuando los datos son de tamaño uniforme, y tienden a variar de una forma relativamente lenta. La compresión de éste tipo de datos puede ser tan alta como del 98%. Un ejemplo de ésto pueden ser las siguientes series de nombres o enteros:

Dato Original: Johnson, Jonah, Jones, Jorgenson
1500, 1520, 1600, 1550, 1570, 1610
Dato Comprimido: (0)Johnson, (2)nah, (3)es, (2)rgenson
1500, 20, 80, -50, 20, 40

Shannon-Fano

El primer método bien conocido para codificar efectivamente los símbolos se conoce ahora como el codificador Shannon-Fano. Claude Shannon de Bell Labs y R. M. Fano de M.I.T. desarrollaron este método casi simultáneamente. Este depende del simple conocimiento de la probabilidad de cada símbolo de aparecer en un mensaje. Dadas las probabilidades, se puede construir una tabla de códigos que tenga algunas propiedades importantes:

- Códigos distintos tienen diferente número de bits.
- Códigos para símbolos con probabilidades bajas tienen más bits, y códigos para símbolos con probabilidades altas tienen menos bits.
- Si bien los códigos son de longitudes de bit diferentes, ellos pueden ser descodificados de forma única.

Las primeras dos propiedades van de la mano. Desarrollando códigos que varían en longitud deacuerdo a la probabilidad del símbolo ellos son codificados haciendo la compresión posible. Y acomodando los códigos como un árbol binario se soluciona el problema de descodificar estos códigos de longitud variable. Descodificar un código que ingresa, consiste en empezar de la raíz, luego virar a la

izquierda o a la derecha de cada nodo después de leer el bit que ingresa del flujo de datos. Eventualmente se lee una rama del árbol, y el símbolo apropiado es descodificado. La estructura del árbol muestra como los códigos son definidos de manera única aun cuando ellos tienen diferentes números de bits.

El algoritmo. Un árbol Shannon-Fano se construye deacuerdo a un algoritmo específico diseñado para definir una tabla de código efectiva. El algoritmo es simple:

1.- Para una lista de símbolos dados, elabórese una lista correspondiente de probabilidades ó conteos de frecuencia para determinar la frecuencia relativa de ocurrencia de cada símbolo.

2.- Ordene la listas de símbolos de acuerdo a la frecuencia, con las ocurrencias de símbolos más frecuentes en el tope, y las menos comunes en el fondo.

3.- Divida la lista en dos partes, con la frecuencia total de la mitad superior tan cerca como sea posible del total de la mitad inferior.

4.- Se le asigna a la mitad superior de la lista el dígito binario 0 y a la mitad inferior se le asigna el dígito 1. Esto significa que los códigos para los símbolos de la primera mitad comenzarán todos con 0 y los códigos en la segunda mitad comenzarán todos con 1.

5.- Aplique los pasos 3 y 4 recursivamente a cada una de las dos partes, subdividiendo grupos y sumando bits a los códigos hasta que cada símbolo haya tenido su rama correspondiente de código en el árbol.

Considérese el siguiente ejemplo:

Símbolo	Frecuencia	Código
A	15	0 0
		Segunda división
B	7	0 1
		Primera división
C	6	1 0
		Tercera división
D	6	1 1 0
		Cuarta división
E	5	1 1 1

Esos símbolos con la probabilidad de ocurrencia alta tienen pocos bits en sus códigos la que indica que se está en el camino correcto. La fórmula para el contenido de información para un símbolo dado es el negativo del logaritmo en base dos de la probabilidad del símbolo. Para éste mensaje teórico, el contenido de información de

cada símbolo, junto con el número total de bits para ese símbolo en el mensaje, fueron encontrados en la tabla siguiente:

Símbolo	Frecuencia	Cont. Info.	Bits Info.	SF	Bits SF
A	15	1.38	20.68	2	30
B	7	2.48	17.35	2	14
C	6	2.70	16.20	2	12
D	6	2.70	16.20	3	18
E	5	2.96	14.82	3	15

La información de éste mensaje suma alrededor de 85.25 bits. Si codificamos los caracteres usando caracteres ASCII de 8-bits, usaríamos 39 x 8 bits, ó 312 bits. Obviamente hay espacio que puede mejorarse. Con el sistema de codificación Shannon-Fano, toma solamente 89 bits codificar 85.25 bits de información. Claramente se ha recorrido un largo camino en la búsqueda de métodos eficientes de codificación.

El Algoritmo Huffman

La codificación Huffman comparte muchas de las características de la codificación Shannon-Fano. Crea códigos de longitud variable de un número entero de bits. Símbolos con altas probabilidades obtienen códigos cortos. Los códigos Huffman tienen el atributo de prefijo único, el cual significa que ellos pueden ser descodificados correctamente a pesar de ser de longitud variable. La descodificación de un flujo de códigos Huffman se hace generalmente siguiendo un descodificador de árbol binario.

La construcción del árbol de descodificación Huffman se hace usando un algoritmo completamente diferente de el del método de Shannon-Fano. El árbol Shannon-Fano se construye de arriba hacia abajo, comenzando por asignar los bits más significativos a cada código y trabajando bajo el árbol hasta terminar. Los códigos Huffman son construidos de abajo hacia arriba, comenzando con las hojas del árbol y trabajando progresivamente cada vez más cerca de la raíz.

El procedimiento de construcción del árbol es simple y elegante. Los símbolos individuales son tendidos como una cadena de nodos de hoja que van a ser conectados por un árbol binario. Cada nodo tiene un peso, el cual es simplemente la frecuencia o la probabilidad de ocurrencia de los símbolos. El árbol entonces es construido con los siguientes pasos:

1. Se ubican los dos nodos libres con los más bajos pesos.
2. Se crea un nodo padre para estos dos nodos. Se le asigna un peso igual a la suma de los dos pesos hijos.
3. El nodo padre se suma a la lista de nodos libres, y los dos nodos hijos son removidos de la lista.

4. Un nodo hijo se designa como la trayectoria tomada del nodo padre cuando se descodifica un bit 0. El otro es arbitrariamente cambiado al bit 1.
5. Se repiten los pasos previos hasta que queda un solo nodo libre. Este nodo libre es designado como la raíz de árbol.

Este algoritmo puede aplicarse a los símbolos usados en el ejemplo previo. Los cinco símbolos en el mensaje son tendidos, junto con sus frecuencias como se muestra:

15	7	6	6	5
A	B	C	D	E

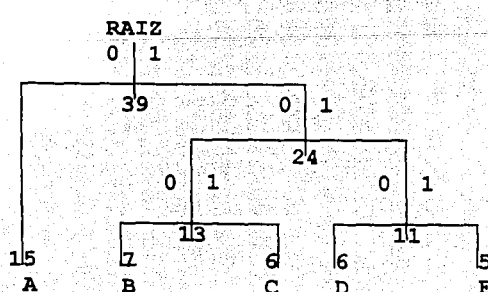
El primer paso a lo largo del árbol identifica los dos nodos libres con los más bajos pesos: D y E. (El empate entre C y D se rompió de manera arbitraria. Mientras que la forma en que se rompen los empates afecta el valor final de los códigos, no afectará el radio de compresión obtenido.) Estos dos nodos conforman un nodo padre, al cual se le asigna un peso de 11. Los nodos D y E se remueven de la lista libre.

Una vez completado este paso, sabemos que bits menos significativos van a estar en los códigos para D y E. D es asignado al ramal 0 del nodo padre, y E es asignado al ramal 1. Estos dos bits serán los BMS de los códigos resultantes.

En el paso siguiente a lo largo de la lista de nodos libres, se toman los nodos B y C como los de peso más bajo. Estos son juntados a un nuevo nodo padre al cual se le asigna un peso de 13 y B y C son removidos de la lista de nodos libres.

En el paso siguiente, los dos nodos con los más bajos pesos son los nodos padres de los pares B/C y D/E. Estos son enlazados juntos con un nuevo nodo padre, al cual se le asigna un peso de 24, y los hijos son removidos de la lista libre. En este punto, se tienen asignados dos bit a cada código Huffman para B, C, D, E, y se tiene que asignar todavía un solo bit al código de A.

Finalmente, en el último paso, solo quedan dos nodos libres. El padre con un peso de 24 se enlaza con el nodo A para crear un padre nuevo con un peso de 39. después de remover los dos nodos hijos de la lista libre, queda un solo nodo padre, lo que significa que el árbol se ha completado quedando así:



Para determinar el código de un símbolo dado, se tiene que caminar del nodo hoja hacia la raíz del árbol Huffman, acumulando nuevos bits a medida que se pasa por cada nodo padre. Desafortunadamente, los bits se obtiene en orden inverso al que se requiere, por lo que se tienen que ir metiendo a una pila y después retirados para generar el código. Esta estrategia da al mensaje la estructura de código siguiente:

A: 0
B: 100
C: 101
D: 110
E: 111

Como se puede ver, los códigos tienen la propiedad de prefijo único. Ya que ningún código es prefijo de otro código, los códigos Huffman pueden ser ciertamente descodificados según arriben a un flujo. Al símbolo con la más alta probabilidad, A, se le asignaron pocos bits, y al símbolo con la probabilidad más pobre, E, se le asignaron muchos.

Hay que notar, sin embargo, que los códigos Huffman difieren en longitud de los códigos Shannon-Fano como se muestra en la tabla siguiente:

Símbolo	Frecuencia	Shannon-Fano		Huffman	
		Tamaño	Bits	Tamaño	Bits
A	15	2	30	1	15
B	7	2	14	3	21
C	6	2	12	3	18
D	6	3	18	3	18
E	5	3	15	3	15

Para éste mensaje con un contenido de información de 85.25 bits, el codificador Shannon-Fano requirió de 89 bits, pero el codificador Huffman requirió de solo 87. En general los dos codificadores tienen un desempeño semejante, pero el codificador Huffman siempre estará por lo menos con una eficiencia igual que la del codificador Shannon-Fano, por lo que pasa a ser el método de codificación predominante en su tipo.

Codificación Adaptativa. Hasta ahora solo se ha usado modelos de orden cero que son esencialmente libres de contexto. Es decir que la probabilidad de un carácter dado es calculada sin tomar en cuenta los caracteres precedentes en el mensaje.

Un inconveniente menor de los programas de codificación Huffman es el requerimiento de que ellos transmitan una copia de la tabla de probabilidades con los datos comprimidos. El programa de expansión no tendría forma de descodificar correctamente los datos sin la tabla de probabilidades. La tabla requiere de la adición de 250 o más bytes extra a la tabla de salida, y consecuentemente esto no consigue

mucha diferencia en el radio de compresión. Incluso archivos pequeños no serán afectados grandemente, ya que la tabla de probabilidad también sería pequeña para estos archivos.

El problema con este "inconveniente menor" es que como se intenta mejorar la habilidad de compresión del programa, la sanción viene a ser más y mas significativa. Si movemos el modelado de orden 0 al orden 1, por ejemplo, se tendrá que transmitir ahora 257 tablas de probabilidades en lugar de solo una. Así al usar una técnica que nos permite predecir caracteres más precisamente, incurriremos en una pena en términos de sobrecarga. Al menos que los archivos a comprimir sean muy largos, esta penalización que se suma cancela cualquier mejora alcanzada al incrementar el orden.

Afortunadamente, la codificación adaptativa nos permite usar modelos de orden alto sin tener que pagar ninguna pena por la adición de estadísticas. Esto lo logra ajustando el árbol Huffman al vuelo (on the fly), basado en los datos previamente vistos y sin tener ningún conocimiento de las estadísticas futuras.

El compresor y el decompresor comienzan con modelos idénticos para codificar y descodificar. Así cuando el compresor genera su primer símbolo codificado, el decompresor será capaz de interpretarlo y se procede a actualizar el modelo. La actualización del modelo toma en cuenta el carácter que acaba de ser visto y actualiza la frecuencia y codifica los datos usados para codificar ese carácter. En un árbol Huffman, esto significa incrementar el conteo de un símbolo en particular y entonces actualizar el árbol de codificación Huffman.

Actualizando el Árbol Huffman. Existe una forma de tomar un árbol de codificación Huffman y modificar su conteo para un carácter nuevo. Todo lo que se requiere es una aproximación ligeramente diferente al construir el árbol en primer lugar. Esta aproximación introduce un concepto conocido como la propiedad de hermandad. Cada nodo (excepto la raíz) tiene un hermano, el otro nodo que comparte el mismo padre. El árbol exhibe la propiedad de hermandad si los nodos se pueden listar en orden de pesos crecientes y si cada nodo aparece adyacente a su hermano en la lista. Un árbol binario es un árbol Huffman si y solo si obedece la propiedad de hermandad.

La propiedad de hermandad es importante en la codificación adaptativa Huffman porque ayuda a mostrar lo que se le necesita hacer a un árbol Huffman cuando es tiempo de actualizar sus conteos. Manteniendo la propiedad de hermandad durante la actualización se asegura el tener un árbol Huffman antes y después de ajustar los conteos.

Actualizar el árbol consta de dos tipos de operaciones básicas. La primera, incrementar el conteo, es fácil de seguir conceptualmente. Para incrementar el conteo del símbolo "c", se comienza en el nodo hoja del símbolo incrementando su conteo. Entonces se sube al nodo padre. Ya que el peso del nodo padre es la suma de los pesos de sus hijos, el incrementar en uno su peso lo ajustará a su valor correcto. Este proceso continuará al subir por todo el camino hasta llegar al nodo raíz.

La segunda operación requerida en el proceso de actualización proviene cuando el incremento del nodo causa la violación de la propiedad de hermandad. Esto ocurre cuando el nodo al ser incrementado tiene el mismo peso que el siguiente nodo más alto en la lista. Si el incremento se hiciera normalmente no se tendría más un árbol Huffman.

Cuando se tiene un incremento que viola la propiedad de hermandad, se necesita mover el nodo afectado a un punto más alto en la lista. Esto significa que el nodo es separado de su posición presente en el árbol e intercambiado con otro nodo mas alto de la lista. Si el nodo recientemente incrementado ahora tiene un peso de W+1 el nodo siguiente tendrá un peso de W. Pueden haber más nodos después del siguiente que también tengan un valor de W. El procedimiento de intercambio se mueve hacia arriba de la lista hasta encontrar el último nodo con un peso de W el cual será intercambiado con el nodo de peso W+1. La nueva lista de nodos entonces tendrá una cadena de uno o más nodos de peso W, seguido por el nodo recientemente incrementado de peso W+1. Después de que el intercambio se ha completado, el proceso de actualización puede continuar. El siguiente nodo a ser incrementado será el nuevo padre del nodo incrementado.

Aunque la codificación Huffman es relativamente fácil de implementar, el proceso de descompresión es bastante complejo, y es el mayor problema con la técnica. En primer lugar, la longitud de cada código a ser leído, para descompresión, no es conocida hasta que los primeros bits son interpretados. Segundo, el método básico para interpretar cada código, es interpretar cada bit en secuencia, y entonces escoger una adecuada traslación de la tabla según el bit sea cero o uno, en otras palabras, una operación de árbol binario. Si un solo bit es mal interpretado, la cadena completa no será descodificada correctamente. En general, la descompresión de códigos Huffman implica un costo y desventaja en su ejecución siempre que el volúmen de los datos es alto.

Un segundo problema menor con la codificación Huffman es que debe de conocerse la frecuencia de distribución del conjunto de símbolos de entrada. Para archivos de texto, en lengua inglesa usando símbolos de un solo carácter, la distribución es bien conocida y relativamente estable. Pero, la distribución de archivos especializados puede ser muy variada. La solución común en este caso es el analizar cada bloque individualmente hasta determinar la distribución de caracteres única de ese bloque. En éste escenario, son necesarios dos pasos referentes a los datos; uno de contar caracteres, computar los códigos y construir la tabla, y uno de codificar. Debido a que la tabla se construye para bloques específicos deberá ser cargada con cada bloque de datos comprimido. Esta propuesta es buena si no se requieren altos rangos de transferencia a través del compresor, y si los bloques de datos que son comprimidos son muy largos comparados con la tabla de traslación. La técnica de codificación Huffman solo vale la pena con una sesgada distribución de caracteres. Generalmente, las compresiones han sido consistentemente reportadas en el rango del 40-55%, con algunos ligeramente más altos. Mucho trabajo se ha hecho en el área de la codificación Huffman y hay muchas variaciones, mayormente a través de algoritmos basados en técnicas enumerativas,

combinatorias y usando árboles desplegados o pilas en vez de árboles binarios para la descompresión.

Codificación Aritmética

Una técnica relativamente nueva, la codificación aritmética, representa un mensaje por un intervalo o rango de números reales entre 0.0 y 1.0. A medida que el mensaje sea más largo, el intervalo necesario para representarlo será más pequeño, y el número de bits necesarios para especificar el intervalo crece. Símbolos sucesivos del mensaje reducen el tamaño del intervalo de acuerdo a las probabilidades del símbolo generadas por la técnica. Los símbolos más probables reducen el rango menos que los improbables, y además suma algunos bits al mensaje.

La salida de un proceso de codificación aritmética es un número menor que 1 y más grande que 0. Este número puede ser descodificado para crear el flujo exacto de símbolos que intervinieron en su construcción. Para construir el número de salida, los símbolos son asignados a un conjunto de probabilidades. El mensaje "BILL GATES", por ejemplo, tendría una distribución de probabilidad como esta:

CARACTER	PROBABILIDAD	RANGO
ESPACIO	1/10	$0.0 \geq r > 0.1$
A	1/10	$0.1 \geq r > 0.2$
B	1/10	$0.2 \geq r > 0.3$
E	1/10	$0.3 \geq r > 0.4$
G	1/10	$0.4 \geq r > 0.5$
I	1/10	$0.5 \geq r > 0.6$
L	2/10	$0.6 \geq r > 0.8$
S	1/10	$0.8 \geq r > 0.9$
T	1/10	$0.9 \geq r > 1.0$

Una vez conocidas las probabilidades de los caracteres, los símbolos necesitan ser asignados a un rango nominalmente de 0 a 1. No importa que carácter es asignado a que segmento del rango siempre y cuando sea hecho de la misma forma tanto por el codificador como por el descodificador.

La porción mas significativa de un mensaje aritméticamente codificado pertenece al primer símbolo (B en nuestro mensaje). Para descodificar el mensaje adecuadamente, el mensaje codificado final tiene que ser un número más grande o igual que .2 y menor que .3.

Durante el resto del proceso de codificación, cada nuevo símbolo restringirá mas ampliamente el rango posible del número de salida. El siguiente carácter a ser codificado, la letra I, pertenece al rango .5 a .6 dentro del nuevo subrango de .2 a.3. Así el nuevo número codificado caerá en algún lugar dentro del 50vo a 60vo percentil

del rango corriente establecido. Aplicando esta lógica se restringirá nuestro número a .25 a .26. Para efectuar esto se tiene el siguiente algoritmo:

$$RANGO = VA - VB$$
$$VA = VB + (RANGO * VAsímbolo)$$
$$VB = VB + (RANGO * VBsímbolo)$$

Este procedimiento es repetido para cada carácter en el mensaje, hasta que el mensaje termina, al codificar un carácter especial de fin de mensaje (EOM) el cual se omite en el ejemplo. Así el valor bajo final, .2572167752, codificará únicamente el mensaje "BILL GATES" usando el presente esquema de codificación.

CARACTER NUEVO	Valor Bajo	Valor Alto
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
ESPACIO	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

Dado este esquema, es relativamente fácil el ver como opera el proceso, de descodificación. Buscando el primer símbolo en el mensaje viendo a cual símbolo pertenece el espacio en el que cae nuestro mensaje codificado. Ya que .2572167752 cae entre .2 y .3 el primer carácter debe ser una B. De aquí la técnica computa un rango nuevo como sigue:

$$NuevoVA = \frac{VA - VBsímbolo}{VAsímbolo - VBsímbolo}$$

$$NuevoVB = \frac{VB - VBsímbolo}{...VAsímbolo - VBsímbolo}$$

El descodificador determina que rango de símbolo abarca estos nuevos valores, y de ésta manera el siguiente símbolo es descodificado, continuando el proceso hasta terminar el mensaje.

Número Codificado	Símbolo de Salida	Bajo	Alto	Rango
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1

0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	ESPACIO	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.2	0.3	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

Cuestiones Prácticas. Codificar y descodificar un flujo de símbolos usando codificación aritmética no es muy complicado. Pero a primera vista parece completamente impráctico. La mayoría de las computadoras soportan números de punto flotante de alrededor de 80 bits. La codificación aritmética esta mejor consumada usando una matemática entera standard de 16 y 32 bit. La matemática de punto flotante ni ayuda ni tampoco se requiere. Lo que sí se requiere es un esquema de transmisión incremental en el cual variables de estado enteras de tamaño fijo reciben bits nuevos en el final bajo y los sacan al final alto, formando un número que puede ser tan largo como el número de bits en la computadora.

Hemos visto que el algoritmo trabaja manteniendo el curso del número alto y el bajo que limitan el rango del posible número de salida. Cuando el algoritmo comienza, el número bajo se inicia a cero y el alto a uno. La primera simplificación hecha para trabajar con matemática entera es cambiar el 1 a .999 ..., o .111 ... en binario. Los matemáticos concuerdan que .111 ... binario es exactamente el mismo que 1 binario.

Para almacenar estos números en registros enteros, primero se justifican de manera que los puntos decimales estén en la parte izquierda de la palabra. Después cargue tantos valores iniciales altos y bajos como quepan dentro del tamaño de palabra con el que se trabaja.

Para encontrar los nuevos rangos se aplica el algoritmo anteriormente descrito. al principio se calcula el rango como 100000 y no como 99999. Dada la naturaleza del algoritmo los valores alto y bajo continuaran creciendo juntos sin que lleguen a concordar por completo. Así una vez que concuerden en el dígito más significativo, ese dígito nunca cambiará. Entonces se puede sacar ese dígito como el primero de nuestro número codificado. Este proceso continúa desarrollándose como se muestra a continuación:

	ALTO	BAJO	RANGO	SALIDA ACUMULADA
Estado inicial	99999	00000	100000	
B Codificada (.2 .3)	29999	20000		
Sale **2**	99999	00000	10000	.2
I Codificada (.5 .6)	59999	50000		.2
Sale **5**	99999	00000	100000	.25
L Codificada (.6 .8)	79999	60000	20000	.25
L Codificada (.6 .8)	75999	72000		.25
Sale **7**	59999	20000	40000	.257
ESPACIO Codif. (0 .1)	23999	20000		.257
Sale **2**	39999	00000	40000	.2572
G Codificada (.4 .5)	19999	16000		.2572
Sale **1**	99999	60000	40000	.25721
A Codificada (.1 .2)	67999	64000		.25721
Sale **6**	79999	40000	40000	.257216
T Codificada (.9 1.)	79999	76000		.257216
Sale **7**	99999	60000	40000	.2572167
E Codificada (.3 .4)	75999	72000		.2572167
Sale **7**	59999	20000	40000	.25721677
S Codificada (.8 .9)	55999	52000		.25721677
Sale **5**	59999	20000		.257216775
Sale **2**				2572167752
Sale **0**				25721677520

Después de contabilizar todas las letras, se sacan dos dígitos extra del valor alto ó del bajo para finalizar la palabra que sale.

Es posible un estancamiento permanente de los valores en el caso de que la palabra codificada tenga alguna cadena de 0s o 9s dentro (lo que pudiera ocasionar que el rango entre el valor alto y el bajo sea tan pequeño que cualquier iteración hacia otro símbolo deje sus mismos valores alto y bajo). Si los dígitos más significativos alto y bajo no concuerdan, pero son números adyacentes, debe de aplicarse una segunda prueba consistente en verificar si el segundo dígito significativo alto es un 0 y si el segundo dígito bajo es un 9. Sí es así, significa que comienza a haber estancamiento por lo que deberá borrarse el segundo dígito de ambos valores y se recorren el resto de los dígitos a la izquierda para llenar el espacio. Los dígitos más significativos se quedan en su lugar y se pone un contador de estancamiento para recordar que se eliminó un dígito y no se esta completamente seguro si fue un 0 o un 9. Cuando finalmente los dígitos más significativos converjan a un solo valor, saque ese valor. Entonces saque los dígitos estancadores previamente descartados. Los dígitos estancadores serán 9s o 0s dependiendo de que ya sea que si el alto y bajo convergieron en el valor alto o bajo.

En la descompresión se pueden usar al igual que en la compresión, enteros de 16 y 32 bit para los cálculos. En vez de usar solo dos números, alto y bajo, el

descodificador debe usar tres números. Los primeros dos, alto y bajo, corresponden exactamente a los valores altos y bajos usados por el codificador. El tercer número, el código, contiene los bits que son leídos del flujo de entrada. El valor del código siempre cae entre los valores alto y bajo. Según ellos se vayan acercando a él tendrán lugar nuevas operaciones de cambio para alejarlos del código.

Los valores alto y bajo en la descompresión serán actualizados después de cada símbolo con exactamente los mismos valores obtenidos en la codificación y utilizando las mismas pruebas de comparación y de estancamiento.

Compresión Basada en Diccionario

Los algoritmos de compresión basados en diccionario usan una metodología especial para comprimir los datos. Esta familia de algoritmos no codifica los símbolos como cadenas de bits de longitud variable; codifica cadenas de símbolos de longitud variable como objetos (tokens) individuales. Los objetos forman un índice de un diccionario de frases. Si los objetos son más pequeños que las frases que reemplazan, la compresión ocurre. Pero esta definición deja mucho espacio de variación. Considérese, por ejemplo, los métodos para construir y mantener un diccionario.

En algunos casos, es ventajoso el uso de un diccionario predefinido para codificar texto. Si el texto a codificarse es una base de datos conteniendo los registros de todos los vehículos motorizados de Toluca, se puede desarrollar un diccionario de unos cuantos cientos de entradas que se concentre en palabras como "General Motors", "Pérez" y "1977".

Un diccionario como este es llamado diccionario estático. Es construido poco antes de que ocurra la compresión, y no cambia mientras los datos son comprimidos. Una de sus más grandes ventajas es que un diccionario estático puede configurarse para adaptarse al dato que se esta comprimiendo. En el ejemplo de los vehículos motorizados, por ejemplo, la codificación Huffman puede asignar pocos bits a cadenas como "Ford" y algunos más a "Yugo".

Los esquemas de compresión adaptativa no pueden configurar sus diccionarios de antemano, lo cual en un principio pareciera una desventaja mayor. Pero los esquemas de diccionario estáticos tienen que lidiar con el problema de como pasar el diccionario de el codificador a el descodificador que en algunos casos puede ser significativamente perjudicial para la compresión (particularmente en archivos pequeños) y en otros puede permanecer igual por largos periodos de tiempo y estar disponible para ambos el compresor y el descompresor.

En la actualidad los esquemas de compresión basada en diccionario que utilizan diccionarios estáticos son en su mayoría ad hoc, de implementación dependiente y de propósito no general. Los mejor conocidos algoritmos de diccionario son adaptativos. En vez de tener un diccionario completamente definido cuando la compresión comienza, los esquemas adaptativos comienzan ya sea sin diccionario ó con un diccionario base por omisión. A medida que la compresión procede, los algoritmos suman frases nuevas para ser usadas más tarde como objetos codificados.

Los componentes básicos de un algoritmo de compresión de diccionario adaptativo son:

1. Analizar el flujo de texto de entrada dentro de fragmentos probados contra el diccionario.
2. Examinar los fragmentos de entrada contra el diccionario; puede ser o no deseable el reportar sobre concordancias parciales.
3. Sumar frases nuevas al diccionario.
4. Codificar los índices del diccionario y texto ordinario con que son distinguibles.

El correspondiente algoritmo de descompresión tiene un conjunto de requerimientos ligeramente diferente:

1. Descodificar el flujo de entrada dentro de los índices del diccionario o dentro del texto corriente.
2. Sumar nuevas frases al diccionario.
3. Convertir los índices del diccionario en frases.
4. Sacar las frases como texto corriente.

La habilidad para efectuar estas tareas con relativamente bajos costos en los sistemas ha hecho muy popular a los programas basados en diccionario durante los últimos 10 años.

ARC: El Padre de la Compresión por Diccionario para MS-DOS. En 1985, System Enhancements Associates desarrollaron un programa de compresión de propósito general llamado ARC.ARC que paso a ser de hecho el standard para los usuarios de PC en cuestión de meses. Varios factores contribuyeron a que ganara esta posición. Primero, usó ordinariamente una derivación cercana del Compress para comprimir los archivos. Al mismo tiempo, ésto dió una compresión sin igual. Segundo, ARC proporcionó una función de archivamiento o catalogamiento como parte integral del programa. Los usuarios de UNIX estaban acostumbrados a usar el programa "tar" para combinar grupos de archivos dentro de uno solo, pero los usuarios de PC no contaban con una función similar como parte de su sistema operativo. ARC sumó esta capacidad, vital para transferir grupos de archivos por módem o disco flexible (floppy) inclusive.

Con Compress reinando en el mundo UNIX y ARC rigiendo el mundo del MS-DOS, pareciera que el LZ78 sería el método de compresión dominante por años. Imitadores como PKWare's PKARC solo reforzaron la permanencia del LZ78 al proveer mejorías tanto en velocidad como en radios de compresión.

ARC perdió su dominancia del mundo de las computadoras personales ante nuevos contendientes, los más notables PKZIP, por PKWare's; pero también LHarc, por Haruyasu Yoshizaki; y ARJ, por Robert Jung. Estos programas fueron construidos en un algoritmo LZ77 que usa un diccionario basado en una ventana deslizante que

se mueve a través del texto. LZ77 no fue un algoritmo práctico de implementar hasta que se le hicieron refinamientos a mediados de los años 1980s. Ahora LZ77 tiene una posición legítima junto con el LZ78 como co-regidor del mundo de la compresión de propósito general.

Se pueden subdividir la aplicaciones para la compresión basada en diccionario en dos áreas: programas de propósito general y código de máquina específico.

Dentro del mundo del MS-DOS programas como PKZIP, ARC, ARJ y LHarc usan todos algoritmos basados en diccionario para comprimir los archivos de manera de propósito general. La mayoría de estos programas tienen puertos a por lo menos una o dos plataformas, siendo la más popular UNIX.

La compresión basada en diccionario también es usada en algunos programas de oficina de propósito especial. La mayoría de los programas de respaldo, por ejemplo, usa alguna forma de compresión para hacer su operación más rápida y eficiente.

El servicio de información Compuserve desarrolló un esquema de compresión basado en diccionario usado para codificar imágenes gráficas bit-mapeadas. El formato GIF usa una variante del LZW para comprimir secuencias repetidas en imágenes de pantalla. La compresión es claramente necesaria cuando se usan este tipo de imágenes. Las imágenes de computadora ocupan un enorme espacio de almacenamiento. A medida que la resolución del vídeo mejora, el tamaño de las imágenes salvadas crece dramáticamente. Compuserve utiliza un módem para cargar o descargar estas imágenes por lo que la compresión es crucial. La utilización de la compresión de los datos dentro de los protocolos de transmisión de los módem se ratificó al adoptarse un nuevo algoritmo de compresión usado por los manufactores de módem: V.24 bis. Con la adopción de un standard internacional, los constructores de módem pueden ahora implementar la compresión de los datos en sus módem con la seguridad de que sus datos pasarán a módem de otros manufacturadores.

El disco duro es una nueva frontera abordada por la compresión de datos. En el presente, programas de propósito general son frecuentemente usados para comprimir datos almacenados en un disco duro. Pero no es popular el que controladoras o manejadores de disco compriman datos transparentemente a la computadora anfitriona. Varias compañías de programas MS-DOS están trabajando en el manejador de recursos para hacer la compresión transparente al programa de aplicación.

Compresión por Ventana Deslizante

La compresión LZ77 usa el texto visto previamente como un diccionario. Reemplaza frases en el texto de entrada con punteros dentro del diccionario para lograr la compresión. El monto de compresión dependerá del largo de las frases del diccionario, del largo de la ventana dentro del texto previamente visto y de la entropía del texto origen con respecto al modelo del LZ77.

La principal estructura de datos en el LZ77 es una ventana de texto dividida en dos partes. La primera parte consiste en un bloque largo de texto recientemente

descodificado. La segunda, normalmente mucho más pequeña, es una memoria intermedia de preanálisis (MIP) que tiene los caracteres leídos del flujo de entrada pero todavía no codificados.

El tamaño normal de la ventana de texto es de varios miles de caracteres. La memoria intermedia de preanálisis es generalmente mucho más pequeña, tal vez de diez a cien caracteres. El algoritmo trata de corresponder el contenido de la memoria intermedia de preanálisis a una cadena en el diccionario.

El algoritmo LZ77, como originalmente se concibió, editaba secuencias de objetos (tokens). Cada objeto consistía de tres diferentes ítems de datos los cuales definían una frase de longitud variable en la memoria intermedia de preanálisis. Los tres ítems en el objeto son: (1) un equivalente a una frase dentro de la ventana de texto; (2) la longitud de la frase; y (3) el primer símbolo en la memoria intermedia de preanálisis que le sigue a la frase. El siguiente ejemplo muestra una ventana de texto cuando se comprime una parte de código C.

```
for(i=0;i<MAX-1;i++)\r for(j=i+1;j<MAX;j++)\r
```

Cuerpo de la Ventana de Texto MIP

En el ejemplo la memoria intermedia de preanálisis (o buffer look-ahead en el idioma inglés) contiene la frase "<MAX;j++)\r". Buscando a lo largo de la MIP, encontramos que "<MAX" está localizado en la posición 14 en la ventana de texto por lo que existe correspondencia en los primeros cuatro símbolos de la MIP. El primer símbolo no presente en la MIP es el carácter de espacio. Así este objeto se codifica como: 14,4,' '.

El programa de compresión que implementa el algoritmo LZ77 emite el objeto, después desliza la ventana cinco caracteres, que es el ancho de la frase recién codificada. Cinco nuevos símbolo son entonces leídos en la MIP, y el proceso se repite. Si la MIP no llegará a mostrar correspondencia alguna, se podría codificar un sólo carácter a la vez usando una frase de longitud cero: "0,0,';'". Este método no es eficiente pero asegura que el algoritmo puede codificar cualquier mensaje.

El algoritmo de descompresión para el LZ77 es simple, ya que no se tienen que hacer comparaciones. Lee en un objeto, sacando la frase indicada, sacando el siguiente carácter, desliza y repite. Un efecto colateral de este método de compresión es que puede usar frases que no hayan sido codificadas todavía para codificar frases existentes, lo que refleja una rápida adaptación al carácter del flujo de entrada.

Un problema importante con la eficiencia del algoritmo LZ77 ocurre cuando no se encuentran frases concordantes en el diccionario. Cuando éste es el caso, el programa de compresión continua usando los tres mismos componentes del objeto para codificar un solo carácter y esto puede llegar a significar el usar mucho más bits que los requeridos originalmente para representar el carácter.

Compresión LZSS

La compresión LZSS busca sobrepasar algunos de los problemas encontrados en el algoritmo original LZ77. Se hacen dos cambios mayores a la forma en que el algoritmo trabaja. El primero es en la forma en que se mantiene la ventana de texto. Bajo el LZ77, las frases en la ventana de texto son almacenadas como un bloque continuo de texto, sin ninguna otra organización al final de éste. LZSS permanece almacenando el texto en ventanas contiguas pero crea una estructura de datos adicional que mejora la organización de las frases.

Como cada frase sale de la MIP y entra a la porción codificada de la ventana de texto, LZSS suma la frase a una estructura de árbol. Al ordenar las frases dentro de un árbol, el tiempo requerido para encontrar la concordancia más larga en el árbol no será más el proporcional al producto del tamaño de la ventana y a la longitud de frase. En vez de ello, será proporcional al logaritmo en base 2 del tamaño de la ventana multiplicado por la longitud de frase.

El segundo cambio yace en la salida actual de la muestra por el algoritmo de compresión. Mientras que el LZ77 se vio sometido a alternar punteros con simples caracteres sin importar la naturaleza del texto de entrada, LZSS permite a punteros y caracteres ser libremente intermezclados. Cuando inicia por ejemplo, el algoritmo de compresión tal vez no encuentre ninguna frase concordante para los primeros doce o más símbolos de entrada. Bajo el sistema LZ77, el codificador permanecería fingiendo la posición concordante con una longitud de cero para todo símbolo en la salida.

LZSS en cambio, usa un bit como prefijo de todo objeto saliente para indicar ya sea que se trate de un par posición/largo o un símbolo para la salida. Cuando salen varios caracteres consecutivos, este método reduce la sobrecarga de posiblemente varios bytes por carácter hasta un sólo byte por carácter.

Voraz vs. el Mejor Posible. Ambos LZ77 y LZSS son llamados algoritmos "voraces": Ellos no observan dentro del flujo de entrada para analizarlo por la mejor combinación de índices y caracteres. Considérese el como se codificaría la frase "Go To Statement Considered Harmful" si los contenidos del diccionario de frase tuvieran los siguientes fragmentos: "Go T" "o S" "tat" " Stat". Un codificador voraz naturalmente primero codificaría la frase "Go T" de cuatro caracteres de longitud, luego la frase "tat" de tres caracteres de longitud. La salida del codificador hasta este punto se vería así:

Frase/Longitud de "Go T" : 25 bits
Frase/Longitud de "o S" : 25 bits
Frase/Longitud de "tat" : 25 bits

75 bits

Un codificador óptimo codificaría los fragmentos como sigue:

Frase/Longitud de "Go " : 25 bits
Carácter "T" : 9 bits
Carácter "o" : 9 bits
Frase/Longitud de "Stat" : 25 bits

68 bits

Este ejemplo muestra claramente que el codificador voraz no lo hace tan bien como el codificador óptimo. Cuando se usa codificación por diccionario es difícil encontrar ejemplos de codificadores óptimos que sobrepasen a los voraces en mas de un poco porciento. Las diferencias más grandes ocurren cuando solamente hay frases cortas en el diccionario, y hay una posibilidad real de que el codificar símbolos individuales tomará menos espacio que una frase.

El problema con la codificación óptima es simplemente de reembolso. Implementar un codificador óptimo generalmente significa que la velocidad de codificación será drásticamente reducida. En el mundo de la compresión de datos, unos cuantos buenos algoritmos heurísticos son más respetados que un probable algoritmo superior. La heurística voraz es en éste caso definitivamente la opción para la mayoría de los programadores de compresión.

Compresión LZ78

Los algoritmos LZ77 tienen una deficiencia que debió meter en problemas a sus creadores, Jacob Ziv y Abraham Lempel. Estos algoritmos usan solamente una pequeña ventana dentro del texto previamente visto, lo cual significa que continuamente desechaban valiosas entradas del diccionario ya que eran desplazadas fuera del diccionario.

La ventana deslizante hace a los algoritmos LZ77 parciales para con el explotamiento de lo novedoso en el texto. Muchos pero no todos los flujos de datos tienden a "fijarse" más en lo que han visto recientemente que en lo que vieron mucho atrás.

Una segunda deficiencia en la compresión LZ77 es el limitado tamaño de la frase que puede ser correspondida. La correspondencia más larga es aproximadamente del tamaño de la memoria intermedia de preanálisis. Un obvio camino para abordar este problema es simplemente empezar remediando el tamaño de la ventana y de la

memoria intermedia de preanálisis. Mientras se asciende el tamaño de ambos parámetros que parecen dirigir estos problemas, el esquema tiene dos desventajas mayores. Primero, cuando se incrementa el tamaño de la memoria intermedia, se necesitan más bits para codificar el índice de ubicación por lo que asciende el costo en bits de codificar una frase. Un efecto más penoso es que cambiando estos parámetros se incrementará drásticamente el tiempo de la UPC necesario para realizar la compresión. Pero lo realmente penoso está al incrementar el tamaño de la memoria intermedia de preanálisis. Ya que las comparaciones de cadenas entre las frases de la ventana de texto y la memoria intermedia de preanálisis proceden secuencialmente, el tiempo de ejecución se incrementará en proporción directa a la longitud de la memoria intermedia de preanálisis. Estos efectos se combinan para cancelar efectivamente cualquier ganancia obtenida del incrementar alguno de estos parámetros en el algoritmo LZ77.

Para efectivamente hacer de lado estos problemas, Ziv y Lempel desarrollaron una forma diferente de compresión basada en diccionario. Este algoritmo popularmente conocido como LZ78 abandona el concepto de ventana de texto. Bajo el LZ77 el diccionario de frases era definido por una ventana fija de texto previamente visto. Bajo el LZ78, el diccionario es una potencialmente ilimitada lista de frases previamente vistas.

LZ78 el similar al LZ77 en muchas formas. LZ78 saca una serie de objetos con una estructura muy parecida a la del LZ77. Cada objeto del LZ78 consiste de un código que selecciona una frase dada y un carácter que le sigue a la frase. A diferencia del LZ77, la longitud de frase no es pasada hasta que el descodificador la conoce.

A diferencia del LZ77, el LZ78 no tiene una ventana lista llena de texto para usar como un diccionario. Después que la frase es sumada, estará disponible para codificar en cualquier momento en el futuro, no solo para los pocos cientos de caracteres siguientes.

La verdadera dificultad con el LZ78 está en el manejo del diccionario. Las frases son almacenadas en un árbol multiforma por lo que hay un número potencialmente grande de ramas quitadas de cada nodo. Los nodos descendientes son usualmente manejados como una lista de índices no más grandes que el número de nodos descendientes que actualmente existan. Esta técnica hace mejor uso de la memoria disponible, pero es significativamente más lento.

Un efecto lateral negativo del LZ78 que no se encuentra en el LZ77 es que el descodificador debe igualmente mantener este árbol. Con el LZ77, un índice de diccionario fue justo el puntero o índice a una posición previa en el flujo de datos. Pero con el LZ78 el índice es el número de un nodo en el árbol diccionario. El descodificador, por lo tanto, tiene que conservar el árbol en exactamente la misma manera que como el codificador, o podría ocurrir una desastrosa incorrespondencia. El programa UNIX COMPRESS, que usa una variante del LZ78, maneja el problema completo del diccionario al supervisar el radio de compresión del archivo. Si el radio de compresión comienza a deteriorarse, el diccionario es borrado y el programa

comienza de cero. De otra forma, el diccionario existente continua siendo usado, aunque no le sean sumadas nuevas frases.

Compresión LZW

La aproximación original Lempel/Ziv a la compresión de datos fue primeramente publicada en 1977, y los refinamientos de terry Welch al algoritmo fueron publicados en 1984.

LZSS mejoró sobre la compresión LZ77 al eliminar el requerimiento de que cada objeto sacaba una frase y un carácter. LZW hace el mismo mejoramiento sobre el LZ78. De hecho, bajo el LZW, el compresor nunca arroja caracteres individuales, solo frases.

Para lograr esto, el mayor cambio dentro del LZW es precargar el diccionario de frase con símbolos de frase igual al número de símbolos en el alfabeto. Así, no hay símbolos que no puedan ser inmediatamente codificados incluso si todavía no ha aparecido en el flujo de entrada.

El algoritmo es sorpresivamente simple. En la compresión LZW se reemplazan cadenas de caracteres con códigos. No se hace ningún análisis del texto que entra. En vez de ello, solo se suma cada nueva cadena de caracteres que detecta a una tabla de cadenas. La compresión ocurre cuando es emitido un código que reemplaza la cadena de caracteres.

El código generado por el algoritmo LZW puede ser de cualquier longitud, pero debe de contener mas bit en él que un solo carácter. Los primeros 256 códigos (cuando se usan caracteres de 8-bit) son asignados por omisión al juego de caracteres standard. El resto de los códigos son asignados a cadenas según proceda el algoritmo.

Compresión. El algoritmo de compresión LZW en su forma más simple se muestra en el siguiente pseudocódigo:

```
RUTINA LZW_COMPRESS
CADENA=obten carácter de entrada
MIENTRAS entren caracteres EFECTUA
 CARACTER=obten carácter de entrada
 SI CADENA+CARACTER esta en la tabla de cadenas ENTONCES
   CADENA=CADENA + carácter
 SI NO
   genera el código para CADENA
   suma CADENA+CARACTER a la tabla de cadenas
   CADENA=CARACTER
 FIN del SI
FIN del MIENTRAS
genera el código para CADENA
```

Cada vez que un nuevo código es generado, significa que una nueva cadena ha sido sumado a la tabla de cadenas. Examinando el algoritmo nos muestra que el LZW siempre checa si las cadenas ya han sido reconocidas, y si es así, saca los códigos existentes en vez de generar nuevos códigos. Un ejemplo del funcionamiento del algoritmo se muestra a continuación:

Cadena de entrada: /WED/WE/WEE/WEB/WET

Entrada de carácter	Salida de código	Nuevo valor de código y cadena asociada
/W	/	256=/W
E	W	257=WE
D	E	258=ED
/	D	259=D/
WE	256	260=/WE
/	E	261=E/
WEE	260	262=/WEE
/W	261	263=E/W
EB	257	264=WEB
/	B	265=B/
WET	260	266=/WET
<EOF>		T

La cadena de entrada es una corta lista de palabras en inglés separadas por el carácter /. Al iniciar el algoritmo se observa al pasar por primera vez el ciclo, que el sistema checa si la cadena /W se encuentra en la tabla. Cuando no se encuentra, genera un código para /, y la cadena /W se suma a la tabla. Debido a que 256 caracteres ya han sido definidos para los códigos 0-255, la definición de la primera cadena será al código 256. Después de que el sistema lee la tercera letra, E, la segunda cadena de código WE, es sumada a la tabla, y se genera el código para la letra W. Este proceso continua hasta que la segunda palabra, los caracteres / y W, son leídos concordando con la cadena número 256. En este caso el sistema arroja el código 256 y suma una cadena de tres caracteres a la tabla de cadenas. El proceso continua hasta acabar con las cadenas y se hayan generado todos los códigos.

Descompresión. El algoritmo de descompresión toma el flujo de códigos saliente del algoritmo de compresión y lo usa para recrear exactamente el flujo de entrada. Una razón de la eficiencia del algoritmo LZW es que no es necesario que pase la tabla de cadenas al código de descompresión. La tabla puede ser construida exactamente como si ocurriera durante la compresión, usando el flujo de entrada como dato. Esto es posible debido a que el algoritmo de compresión siempre saca los componentes CADENA y CARACTER de un código, antes de que use el código en

el flujo de salida. Esto significa que el dato comprimido no es calcinado al trasladar una larga tabla de cadenas. Este es el algoritmo de descompresión en pseudocódigo:

```
RUTINA LZW_DECOMPRESS
Lee CODIGO_VIEJO
genera CODIGO_VIEJO
MIENTAS entren caracteres EFECTUA
 Lee CODIGO_NUEVO
 CADENA=obten la traslación de CODIGO_NUEVO {línea a modificar}
 genera CADENA
 CARACTER=primer carácter en la CADENA
 suma CODIGO_VIEJO+CARACTER a la tabla de traslación
 CODIGO_VIEJO=CODIGO_NUEVO
FIN del MIENTRAS
```

Al igual que en el algoritmo de compresión, suma una nueva cadena a la tabla de cadenas cada vez que lee un nuevo código. Todo lo que tiene que hacer en adición es trasladar cada código que entra dentro de una cadena y trasladarlo a la salida.

Existe un caso de excepción en la compresión LZW que causa algunos problemas en el lado de la descompresión. Si hay una cadena consistente de un par (CADENA, CARACTER) ya definido en la tabla, y el flujo de entrada observa una secuencia de CADENA, CARACTER, CADENA, CARACTER, CADENA, el algoritmo de compresión sacará un código antes de que el decompresor tenga oportunidad de definirlo. Afortunadamente se puede modificar el algoritmo para sobreponer este problema sustituyendo la línea indicada en el algoritmo de descompresión con los pasos siguientes:

```
SI CODIGO_NUEVO no está en la tabla de traslación ENTONCES
  CADENA=obten la traslación del CODIGO_VIEJO
  CADENA=CADENA+CARACTER
SI NO
  CADENA=obten la traslación del CODIGO_NUEVO
FIN del SI
```

CAPITULO III

Evaluación de Técnicas y Resultados

Resultados de Compresión

Como se mencionó anteriormente, el desempeño de cualquier técnica de compresión es expresada como un radio entre el número de bits que representan el mensaje antes y después de la compresión. Se dan a continuación algunos resultados promedio de la compresión para una variedad de tipos de datos usando la técnica LZW.

- *Texto.* Los resultados son razonablemente buenos para textos simples, con reducciones de alrededor del 45%. Muchos archivos de procesadores de palabras, sin embargo, comprimieron mejor cuando contenían figuras, datos formateados, o datos gráficos. Largos documentos individuales no comprimieron tan bien como grupos de documentos cortos, indicando que la redundancia no se debe precisamente al contenido.

- *Números de Punto Flotante.* Los datos consistentes en números de punto flotante se parecen bastante a los números aleatorios, especialmente la parte fraccional, por lo que se comprimen bastante pobremente en al rededor de un 10%. El exponente se comprime un poco mejor cuando la mayoría de los números tienen la misma magnitud.

- *Datos Científicos Formateados.* Este tipo de datos, compuesto principalmente de enteros, tiende a comprimirse bastante bien al rededor del 50%.

- *Datos del Diario del Sistema.* La información que describe la actividad ocurrida en el sistema es en su mayoría enteros formateados, por lo que comprimió alrededor del 65%. Este tipo de datos tienden estar dentro de un paquete hermético, formato de longitud-fija, por lo que la compresión obtenida se debe a campos nulos y a repeticiones en los valores de dato.

- *Programas.* El código fuente tiende a ser comprimido en un factor mejor que dos. En general se puede comprimir mejor que el texto debido a que las palabras son frecuentemente repetidas, y prevalecen los blancos. La programación altamente estructurada reditúa en un código origen con una compresión más grande de alrededor del 70%. Los códigos ejecutables y objeto, por otro lado, consisten en patrones arbitrarios de bit que no comprimen tan bien.

- *Tamaños de Carácter Mezclados.* Algunas veces, los datos son registrados en tamaños de carácter no-estandard. Cuando los símbolos de ocho-bits ASCII son almacenados en bytes de nueve-bits, la compresión observada es ligeramente mayor que la de los símbolos de ocho-bits almacenados en bytes de ocho-bits. Esto se debe, a que la longitud de los datos comprimidos depende del número de símbolos encontrados en vez de su codificación. Los datos de seis-bits comprimieron sorprendentemente bien en una base de nueve-bits debido a los patrones repetidos, tales como blancos, que tienen el mismo patrón en cada ocurrencia. Como una nota aparte, cuando los datos de nueve-bits fueron comprimidos como si fueran caracteres de ocho-bits, los resultados fueron pobres pero aceptables. Básicamente todo esto nos dice , que la compresión puede ser efectiva incluso cuando tratamos con tamaños de caracteres mezclados.

- *Cintas de Recuperación.* Los respaldos de computadoras o cintas de recuperación, conteniendo una sección cruzada de datos usada en el sistema y en los archivos de usuario, se han comprimido en algo menos que el 50% para los datos científicos, y ligeramente algo más que el 50% para los datos del desarrollo del programa.

Modelado para la Compresión de Texto

La codificación aritmética nos permite el comprimir un archivo tan bien como sea posible para un modelo dado del modelo que generó el archivo. Para obtener la máxima compresión de un archivo, necesitamos un buen modelo y una eficiente manera de representar (o aprender) el modelo. Si admitimos dos fases sobre el archivo, podemos identificar un modelo adecuado durante la primera fase, codificarlo, y usarlo para la codificación optima durante la segunda fase. Una alternativa es permitir que el modelo se adapte a las características del archivo en una sola fase, en efecto, aprendiendo el modelo. La propuesta adaptativa tiene ventajas en la práctica: no hay retraso codificando y no hay necesidad de codificar el modelo mientras que el descodificador pueda mantener el mismo modelo que el codificador de una manera sincronizada.

En el siguiente teorema de [S92] se comparan la codificación libre de contexto usando un método de dos fases y un método adaptativo de una fase comprobándose la factibilidad de la aplicación.

TEOREMA. Para todo archivo de entrada, el código adaptativo con peso inicial de 1 da exactamente la misma longitud de código que el código semi-adaptativo decremental en el cual el modelo de entrada es codificado basado en la asunción de que todas las distribuciones de símbolos son igualmente comunes.

En el método de dos fases, los conteos exactos de los símbolos son codificados después de la primera fase; durante la segunda fase cada conteo de símbolo es decrementado siempre que este ocurra, así en cada punto los conteos relativos reflejan la correcta probabilidad del símbolo para el resto del archivo. En el método adaptativo de un paso, todos los símbolos tienen conteos iniciales de 1; se suma 1 al conteo de símbolo siempre que ocurra.

Con lo anterior se demuestra que el uso de un código adaptativo no incurre en sobrecarga extra alguna, pero no elimina el costo de describir el modelo.

Modelos Adaptativos. Los modelos adaptativo más simples no cuentan con contextos para condicionar las probabilidades; la probabilidad de un símbolo es solo su frecuencia relativa en la parte que ya ha sido codificada del archivo. (Se necesita un mecanismo para codificar un símbolo la primera vez, cuando su frecuencia es cero; la forma más fácil es comenzar todos los conteos de símbolo en 1 en vez de 0). El promedio de longitud de código por símbolo de entrada de un archivo codificado usando un modelo adaptativo de orden 0 es bastante cercano a la entropía de orden 0 del archivo. La compresión adaptativa puede ser mejorada tomando ventaja de la localidad de referencia y especialmente usando modelos de orden alto.

Escalado. Un problema al mantener los conteos de símbolos es que los conteos pueden venir arbitrariamente largos, requiriendo incrementar la precisión aritmética en el codificador y más memoria para almacenar los conteos. Al reducir periódicamente todos los conteos de símbolos en el mismo factor, podemos mantener aproximadamente las mismas frecuencias mientras se usa solo un monto fijo de almacenamiento para cada conteo. Este proceso es llamado escalado (scaling). Este nos permite el uso de aritmética de baja precisión posiblemente debilitando la compresión debido a la reducida exactitud del modelo. Por otro lado, introduce una efecto de localidad de referencia, el cual seguido mejora la compresión.

En la mayoría de los archivos de texto encontramos que la mayoría de las ocurrencias de por lo menos algunas palabras están agrupadas en una parte del archivo. Podemos tomar ventaja a esta localidad asignando más peso a las ocurrencias recientes de un símbolo en un modelo adaptativo. En la práctica hay varias formas de hacer esto:

- *Recomenzar periódicamente el modelo.* Esto seguido descarta demasiada información para ser efectivo, pero da buenos resultados con largos modelos dinámicos de Marcov.
- *Usar una ventana deslizante en el texto.* Esto requiere de excesivos recursos computacionales.
- *Dar incrementos de peso exponencialmente a símbolos sucesivos.* Esto es moderadamente difícil de implementar debido a los cambiantes incrementos de peso.

- *Escalado periódico.* Esto es simple de implementar, rápido y efectivo en operación, y ameno al análisis. También tiene la computacionalmente deseable propiedad de mantener los pesos de los símbolos pequeños.

Modelos de Orden Alto. La única manera de obtener mejoras substanciales en la compresión es el uso de modelos más sofisticados. Para archivos de texto, el incremento en la sofisticación invariablemente toma la forma de condicionar las probabilidades de los símbolos en contextos consistentes de uno o más símbolos de texto precedente.

Una dificultad significativa al usar modelos de orden alto es que muchos contextos no ocurren lo suficientemente seguido para dar estimados confiables sobre la probabilidad del símbolo. Cleary y Witten [BCW90] trataron este problema con la técnica llamada Predicción por Igualdad Parcial (PPM). En los métodos PPM se mantienen los modelos de varias longitudes de contexto, u ordenes. En cada punto se usa el modelo de orden alto dentro del cual el símbolo ha ocurrido en el contexto corriente, con un símbolo escape especial indicando la necesidad de bajar a un orden menor.

El método PPMC desarrollado por Moffat [M90] es ampliamente considerado como el mejor método para estimar las probabilidades escape. En PPMC cada peso de símbolo en un contexto es considerado el número de veces que haya ocurrido en el contexto. El evento escape, esto es, la ocurrencia de un símbolo la primera vez en el contexto, también es tratada como un símbolo, con su propio conteo. Cuando una letra ocurre la primera vez su peso es 1; el conteo de escape es incrementado en 1, así el peso total se incrementa en 2. En todas las demás veces el peso total se incrementa en 1.

El método PPMD es similar al PPMC excepto que hace el tratamiento de los nuevos símbolos más consistente al sumar 1/2 en vez de 1 a los contadores de escape y de los nuevos símbolos al presentarse una nueva ocurrencia; por lo que el peso total siempre se incrementa en 1. En la siguiente tabla se muestra que para archivos de texto PPMD comprime consistentemente alrededor de 0.02 bits por carácter mejor que PPMC.

Archivo	PPMC	PPMD
Libro	2.65	2.63
Noticias	2.91	2.90
Documento	2.48	2.46
Programa en C	2.48	2.47

Una Evaluación Empírica de Métodos de Codificación para Alfabetos Multi-Símbolos

Es usual en la compresión de datos el delinear claramente los roles del modelado y de la codificación. Aquí se examinarán los requerimientos de recursos y la eficiencia de compresión de la fase de codificación, concentrándonos en aplicaciones con alfabetos medios y largos. Se implementaron y probaron una variedad de métodos de codificación, incluyendo versiones estáticas y adaptativas de la codificación Huffman, codificación aritmética exacta y aproximada, y codificación tipo árbol.

Los resultados son sorprendentes. Cuando se puede usar codificación semi-estática de dos pasos, la codificación Huffman es cuatro veces más veloz en una SPARCstation 2 que la codificación aritmética, y en ocasiones resulta en compresión superior. Cuando se requiere de un codificador adaptativo, la diferencia en velocidad es pequeña, pero la implementación Gallenger de la codificación dinámica Huffman permanece más veloz que la codificación aritmética en la mayoría de las situaciones. Más aún, la pérdida a través del uso de códigos Huffman es despreciable en circunstancias extremas. Donde es necesaria alta velocidad, la codificación de árbol es considerada de igual importancia, aunque produzca una compresión pobre.

Plataforma de Prueba Experimental.

Para aislar el componente codificación de cualquier consideración de modelado, se trabajo con un flujo $x_1, x_2, \ldots, x_n+1, \ldots$ de "números de símbolo" ordinales. Si r_n símbolos distintos han aparecido en los primeros n símbolos del flujo, entonces el $n+1$vo. símbolo x_{n+1} puede ser cualquier valor en $1 \leq x_{n+1} \leq r_n+1$, con r_n+1 representando un símbolo "nuevo" de que no haya ocurrido previamente.

En un sistema de compresión sofisticado, donde, digamos, las predicciones están condicionadas por un contexto de caracteres precedentes, se le pasaría al codificador un "número de contexto" para crear un índice de un arreglo de codificadores. De este modo, aunque los experimentos parezcan mostrarse para el uso de un modelo de orden cero, los resultados son generales y se aplican a todas las situaciones de codificación.

Se usaron cuatro propuestas para generar los flujos de prueba. El primer archivo de prueba se produjo considerando texto en inglés en el nivel de carácter, y asignando números de símbolo a caracteres en orden de aparición. La segunda prueba de datos tomó el mismo archivo inicial, pero usó palabras como objetos, asignándole números de símbolo ordinales a las palabras. Para el tercer grupo de datos se tomaron como símbolos las no-palabras que aparecían entre las palabras. Finalmente el cuarto grupo de datos estuvo basado nuevamente en caracteres, tomándose como símbolos los caracteres que aparecieron en el contexto "th", es decir, el flujo que podría esperarse en cualquier contexto de un modelo multi-contexto tal como el esquema PPM. Estos método para generar pruebas de datos dieron distintamente diferentes distribuciones de símbolos. Las aproximaciones de los caracteres producen una distribución relativamente uniforme sobre un pequeño alfabeto; la palabra tokenstation produce un alfabeto muy largo, pero nuevamente

con una distribución no sesgada; mientras que los datos no-palabra y los datos contexto-"th" generan distribuciones muy sesgadas (es decir, entropía baja) en un alfabeto largo y pequeño respectivamente. En cada caso se generaron archivos conteniendo 10^6 números de símbolo de un subconjunto de la colección documental TIPSTER. El texto usado consistió de artículos de la AP newswire, incluyendo las instrucciones SGML markup conjuntas. Alrededor de 67 Mb de texto se requirieron para obtener 10^6 contextos "th", y 6 Mb para las palabras y las no-palabras. La tabla siguiente describe estos archivos de prueba más detalladamente. Pmax es la probabilidad de los símbolos más comunes. La columna final muestra un ejemplo.

Archivo	Origen	Símbolos Distintos	Entropía (bits/símbolo)	Pmax	Símbolos en "to be or not to be, that is the question."
C	Caracteres	91	0.49103	0.1496	1,2,3,4,5,3,2,6,3,7,2,1,3,...
P	Palabras	41647	109.438	0.0465	1,2,3,4,1,2,5,6,7,8
N	No palabras	658	21.816	0.7064	1,1,1,1,1,2,1,1,1,3
T	Contexto th	57	18.526	0.6631	1,2

Parámetros de los archivos de texto (10^6 símbolos)

Todos los experimentos se realizaron en una ligeramente cargada 25 MIP Sun SPARCstation 2, con programas escritos en C y compilados usando gcc. Cada codificador leyó un archivo de enteros binarios y escribió un archivo de códigos binarios, y cada programa de descodificación leyó un archivo de códigos binarios y escribió un archivo de enteros binarios. Tanto como fue posible se minimizaron todas las sobrecargas comunes por lo que cualquier diferencia en el funcionamiento sería correctamente identificada.

Codificación Semi-Estática.

La codificación semi-estática dejo de ser favorita en los años recientes debido al anhelo de compresión "modo filtro" de una fase. Sin embargo, hay algunas situaciones en las cuales el descodificador puede hacer dos fases, por ejemplo, cuando el texto va a ser comprimido para su distribución en un CD-ROM.

Códigos Fijos. Un código obvio es el usar una codificación fija de 32 bits, esto es, copiar el flujo de símbolos de entrada a la salida. Este método no comprime, pero provee un límite alto de velocidad. Es difícil imaginar algún programa de compresión que pueda operar mas rápido que un programa de copiado enteramente hecho a la medida.

Otro código simple es el que representa el entero x_{n+1} **en** $\lceil \log_2 (r_n+1) \rceil$ bits, donde rn es el número de los distintos tipos de símbolos en los primeros n símbolos. Técnicamente, éste método es adaptativo, desde que el codificado puede realizarse como una fase y el valor de rn no está fijado, pero se incluyó en los métodos estáticos como una segunda marca del desempeño.

Codificación Huffman. La más ampliamente conocida propuesta de compresión de dos fases es la codificación Huffman. La versión implementada uso el "código Huffman canónico" descrito por Hirschberg y Lelewer [HL90], el cual proporciona una rápida tabla de búsqueda (look up) descodificando con unas cuantas operaciones por bit.

Codificación Aritmética Exacta. En los últimos 10 años la codificación aritmética se ha convertido en una importante técnica. Como apoyo a la discusión anterior, primero se esquematizará el procedimiento seguido por un codificador aritmético típico [N87,WNC87]. Se asume que el símbolo i ha ocurrido C_i veces; que hay r distintos símbolos; y que $n=\Sigma C_i$ es el número total de símbolos. La codificación es caracterizada por dos valores: L, el límite bajo, y R, el rango corriente. Se supone que b bits son usados para representar los enteros L y R, los cuales son iniciados a cero y a 2^{b-1} -1 respectivamente. Entonces para codificar el símbolo i, se realizan los siguientes pasos:

1.- $T \leftarrow R \times (\Sigma C_j)/n$;
2.- $L \leftarrow L + T$;
3.- $R \leftarrow R \times (\Sigma C_j)/n - T$;
4.- MIENTRAS $R < 2^{b-2}$ EFECTUA
SI $L + R \leq 2^{b-1}$ ENTONCES bitplusfollow (0)
SI NO SI $L \geq 2^{b-1}$ ENTONCES bitplusfollow (1); $L \leftarrow L-2^{b-1}$
SI NO bitstofollow $\leftarrow$ bitstofollow + 1; $L \leftarrow L-2^{b-2}$;
$L \leftarrow 2 \times L$; $R \leftarrow 2 \times R$.

Se debe tener cuidado con las terminaciones después de que el último símbolo se haya codificado; una descripción de éste y del uso de bitplusfollow y bitstofollow se puede encontrar en [N87,WNC87].

Claramente, siempre se tendrán $2^{b-2} \leq R < 2^{b-1}$ previo al paso 1. Supongamos que deseamos que n sea tan grande como 2^f para un entero f, y permitir que el conteo sea tan pequeño como uno (C_i=1). Entonces $R>0$ después del tercer paso puede ser garantizado sólo si $R \geq n$ antes del paso uno, esto es, si $b \geq f+2$. Similarmente, si se ha de eliminarse la sobrecarga en las multiplicaciones de los pasos uno y tres, se debe tener w, el ancho total de palabra, igual que $2^w - 1 \geq (2^{b-1} - 1) \times 2^f$. Por tanto se requiere $w \geq b-1+f \geq 2f+1$.

Witten, Neal y Cleary publicaron un código y análisis para un codificador aritmético multi-alfabeto similar al anterior que paso a ser ampliamente conocido como una descripción estándar de la técnica (implementación CACM). Esta diseñada más para codificación adaptativa que para la semi-estática, y es relativamente ineficiente si la compresión de dos fases es posible. Para obtener una implementación semi-estática, se reemplazó la búsqueda lineal en el descodificador por una búsqueda binaria y se removió el índice de arreglos.

La desventaja de ésta implementación "aritmética exacta" es que con enteros de 32 bits (es decir $w=32$), f se restringe a ser a lo más 15, en consecuencia se debe de mantener $n \leq 2^{15}$. Cuando el alfabeto es grande ésto puede producir una inexactitud significativa y la compresión puede deteriorarse; si $r>2^{f}$, el escalamiento no se puede realizar del todo, y el descodificador fallará. El valor de f soportado por cualquier codificador aritmético es una importante consideración para los alfabetos multisimbolos.

Se implementaron otros dos códigos basados en el código CACM. Primero, se usó una mantisa de 52 bits de dato tipo "doble". Segundo se usó el tipo de dato "long long" del compilador gcc, y se implementó un codificador de 64 bit.

El último par de codificadores exactos, denotados como "Shift", hacen uso de una técnica sugerida por Witten y escalaron la frecuencia reemplazando con corrimientos todas las multiplicaciones y divisiones por n, reteniendo las multiplicaciones y divisiones por R. Para la codificación semi-estática se esperaba que estas variantes corrieran más rápido que el codificador CACM sin pérdida en la compresión.

Codificación Aritmética Aproximada. Un método alternativo tanto para permitir grandes valores de f como para evitar la sanción en tiempo de multiplicación y división es el usar la codificación aritmética aproximada, donde, de una manera u otra, las probabilidades son aproximadas y todas las operaciones multiplicativas son reemplazadas por corrimientos. Esta es una bien conocida estrategia para alfabetos binarios (véase [L91]); recientemente varias técnicas han sido desarrolladas para codificación aproximada en alfabetos multisímbolos.

Rissanen y Mohuiddin sugirieron que el valor de R/n sea aproximado por una potencia de dos, y que cualquier probabilidad "dejada de lado" sea sumada al símbolo más probable (SMP). En esta propuesta, la probabilidad efectiva del SMP se incrementará en por lo menos 0.5, y la probabilidad de todos los demás símbolos empeorarán. Por lo tanto la codificación resultante será por lo menos 20% ineficiente, y cuando Pr(SMP) sea largo, dará un desempeño bastante cercano a un codificador exacto. Más aún, debido a este proceso de aproximación, se requiere solamente que b<w, y con w=32, puede tener f=30.

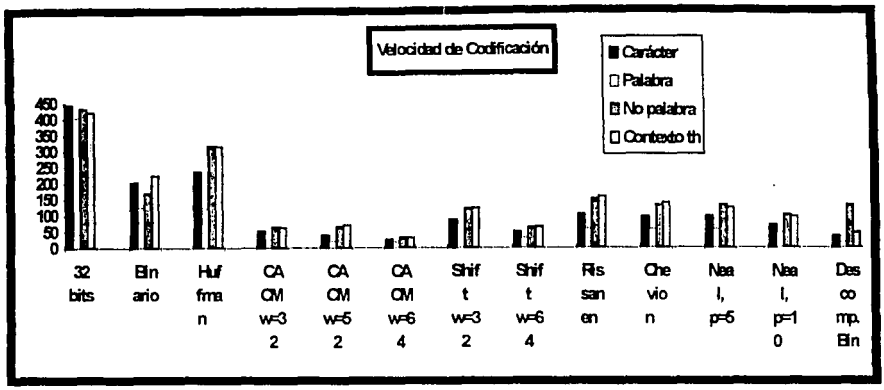
Neal [N87] introdujo un tercer método que también aproximaba el resultado de la división, pero permitía que la precisión del resultado fuera controlada por un parámetro de precisión p, computando efectivamente $[R/n] \times \Sigma C_j$ donde R es almacenado en $w=b$ bits y n es almacenado en $b-p=f$ bits. Cuando p es pequeña, pocos bits de precisión son retenidos en la división, y la imprecisión de la codificación aumenta, pero así también la velocidad. Cuando p es grande, más operaciones corrimiento/suma son requeridas, pero la codificación es más exacta. Por ejemplo, en enteros de 32 Bits, $p=2$ es similar a la esquema de Rissanen y Mohuiddin y se obtiene $f=30$; cuando $p=17$, la codificación esta cercana a lo exacto, correspondiendo al esquema CACM, pero debiendo mantener $n \leq 2^{15}$. Después de la experimentación inicial, se uso $p=5$ y $p=10$ durante las pruebas.

Descomposición Binaria. Alfabetos multi-símbolos pueden ser codificados como una secuencia de decisiones binarias, por ejemplo, codificando cada uno de los bits en el número de símbolo dentro de un contexto condicional de estructura de árbol, de los bits de alta significancia que ya hayan sido codificados. Esto entonces permite el uso de rápidos codificadores aritméticos binarios, como el codificador Q o la codificación Quasi-aritmética, y significa que no hay restricciones en el máximo tamaño del alfabeto ya sea que cada nodo en el árbol pueda ser escalado individualmente. Se implementó esta aproximación usando el codificador de aproximación binaria descrito por Neal con $p=2$, la variante más rápida, esperando establecer si $\log_2 r$ pasos de codificación binaria pudiera ejecutar un paso de codificación multi-alfabeto.

Resultados.

Velocidad. En la tabla se muestran las velocidades de cada uno de los métodos semi-estáticos para codificación y descodificación, y una columna del rendimiento efectivo global calculada como el total del tiempo requerido para codificar y descodificar los cuatro archivos de prueba divididos entre los $8x10^6$ símbolos en que fueron procesados. Los tiempos incluyen el costo de la primera fase de acumulación de frecuencias de archivo, y para el codificador Huffman, el tiempo requerido para construir el código Huffman. Mas adelante esto fue típicamente menor que el 1% del tiempo de codificación, la única excepción fue el archivo de palabras "p", donde el enorme alfabeto requirió 1.95 segundos (de un tiempo de codificación total de 8.23 segundos) para la construcción del código Huffman.

Método	Vel. Codif.				Vel. Decodif.				Global
	c	p	n	t	c	p	n	t	
Código 32-bit	444	405	431	420	424	392	391	375	409
Binario	202	125	170	223	180	114	161	202	164
Huffman	235	122	317	315	228	129	284	413	217
CACM-long, w=32	51	-	63	63	37	-	39	41	-
CACM-doble, w=52	39	21	65	70	30	17	42	48	33
CACM-longlong, w=64	26	19	30	31	20	16	22	22	22
Shift-long, w=32	87	-	120	122	55	-	60	67	-
Shift-longlong,w=64	49	32	62	64	28	21	30	35	35
Rissanen	101	59	150	158	76	43	87	103	82
Chevion	94	57	130	137	83	47	89	105	82
Neal, p=5	96	58	131	123	50	32	55	61	62
Neal, p=10	70	47	100	95	47	32	51	57	55
Descomp. Binaria	36	16	130	47	37	16	32	50	28



Inclusive incluyendo este costo, es impresionante la velocidad del codificador Huffman; fue más rápido que el codificador binario, ya sea que se hayan realizado el mismo número de operaciones por bit, y el codificador binario arroje más bits. El codificador Huffman fue también claramente más veloz que cualquiera de los codificadores aritméticos.

Sobre los codificadores exactos, impresionaron con su velocidad; buscando la explicación de ello, se descubrió que en la SPARCstation usada, la multiplicación entera es más lenta que la multiplicación de punto flotante. Más aún, el código CACM realiza sus operaciones de adición en una base "por bit", y realiza operaciones multiplicativas en una base "por símbolo". Por lo tanto en alfabetos sesgados donde el radio de símbolos de entrada a bits de salida es alto, el codificador aritmético de doble precisión fue más veloz que el codificador entero. No fue posible correr el codificador de 32-bit en el archivo "p", ya que se tenía que $r>2^{\prime}$, y el escalamiento resultó en probabilidades cero.

Los codificadores aproximados fueron, como se esperaba, más rápidos que los exactos, y más rápidos que las variantes "Shift". Ellos fueron un útil compromiso entre la velocidad del codificador Huffman y la precisión de los codificadores aritméticos exactos. El método de descomposición binaria fue muy lento, y además, asumiéndose que un significativamente más rápido codificador binario aritmético pueda ser usado, resulta ser no competitivo. Simplemente hay demasiadas operaciones realizándose.

Requerimientos de Memoria. Todos los codificadores semi-estáticos fueron compactos. La descodificación Huffman y todos los programas de codificación aritmética, requirieron solo 4 bytes por símbolo del alfabeto. La codificación Huffman fue un poco más cara; requirió de 8 bytes por símbolo durante la generación del código al final de la primera fase, y de 5 bytes por símbolo durante la segunda fase. El método de descomposición binaria requirió de dos contadores de frecuencia por

nodo del árbol (uno sería suficiente si no se requiriera el escalado individual), pero fueron almacenados como enteros de 16-bits, para un total de 4 bytes por símbolo.

Compresión. En la tabla siguiente se muestra la compresión obtenida como un porcentaje relativo a la entropía misma (listada en la tabla de parámetros). Por ejemplo, si la entropía misma fue de 9.5 bits/símbolo, y un código generó un promedio de 9.7 bits/símbolo, sería descrito como (9.7-9.5)/9.5=+2.11%. En todos los casos el costo de preparación de las estadísticas fue incluido -para el código Huffman, una lista de longitudes de código codificadas como diferencias de la máxima longitud de código; y para los códigos aritméticos, una lista de frecuencias de símbolo codificadas usando el código Elias γ (véase [BCW90]). Esta sobrecarga fue muy pequeña, y maximizó en el archivo "p", donde contribuyó con +1.68% para los métodos de codificación aritmética, y +1.07 para el codificador Huffman.

Método	Compresión				Global
	c	p	n	t	
Binario	+28.60	+31.39	+283.39	+170.36	+71.29
Huffman	+0.67	+1.36	+6.36	+4.99	+2.08
CACM-long, w=32	+0.04	-	+1.02	+0.08	-
CACM-doble, w=52	+0.03	+1.68	+0.17	+0.04	+0.95
CACM-longlong, w=64	+0.03	+1.68	+0.17	+0.04	+0.95
Rissanen	+5.66	+5.04	+1.89	+1.84	+4.55
Chevion	+0.63	+2.35	+0.57	+0.30	+1.54
Neal, p=5	+0.23	+1.90	+0.23	+0.07	+1.14
Neal, p=10	+0.03	+1.68	+0.17	+0.04	+0.96
Descomp. Binaria	+5.04	+4.18	+2.19	+1.79	+3.95

Nótese la pérdida de eficiencia en la compresión del archivo "n" cuando el CACM estuvo restringido a $f=15$, resultando de la imprecisión introducida por el conteo escalonado. La pérdida en la compresión pretendida al correr el archivo "p" habría sido más severa. Nótese también la relativamente pequeña pérdida en la compresión del código Huffman relativa a la entropía, incluso en las distribuciones sesgadas. Realmente, sobre el archivo "p", una vez incluido el costo de los componentes estadísticos, el codificador Huffman dió mejor compresión que el codificador aritmético.

Los codificadores aproximados también se desempeñan sorprendentemente bien, y en contraste al codificador Huffman, da su mejor compresión relativa cuando la distribución del símbolo no es uniforme. De los tres, el método aproximado de Neal estuvo uniformemente mejor que los esquemas de Rissanen y Mohuiddin, y que Chevion. Parece, para este dato al menos, poca razón para preferir un codificador exacto sobre un codificador aproximado o sobre el Huffman, particularmente cuando el alfabeto es largo.

El codificador aproximado Neal, permite imprecisión en las estadísticas, causada por el conteo escalonado, en el intercambio contra la ineficiencia en la codificación, causada por el uso de precisión aritmética reducida. Por ejemplo, en el archivo "p" el codificador Neal da su mejor desempeño con $p=10$, e incrementando p resultando en más bits de salida en vez que menos.

Codificación Adaptativa

Estructuras de Datos para la Codificación Adaptativa Huffman. Gallanger describió un mecanismo para ajustar dinámicamente un árbol Huffman a corresponder a los conteos de frecuencia cambiantes, y al hacerlo, introdujo la era de la codificación adaptativa de un paso. Desde entonces muchos autores han extendido esta técnica: Comack y Horspool describieron un mecanismo para incrementar el conteo de frecuencia en montos no unitarios; Knuth consideró la cuestión de la eficiencia en alfabetos largos que fueron que fue promovido por Gallanger, describiendo una variante que requirió $O(n+r+H)$ tiempo donde H es el número de bits producido; y Vitter describió una alternativa igualmente eficiente que manejaba símbolos nuevo más eficientemente. Se muestran las evaluaciones de las propuestas de Gallanger, Knuth, y Vitter.

Estructuras de Datos para la Codificación Aritmética Adaptativa. Para la codificación aritmética adaptativa la operación potencialmente cara es el cálculo de ΣC_j. En su implementación ilustrativa Witten al. describió el uso de una lista de movimiento-al-frente, así que los símbolos con grandes valores de C_j aparecen rápidamente en la ordenación. En sus experimentos sobre caracteres de dato (correspondientes al archivo "c") encontraron este método aceptablemente rápido.

Sin embargo, para una búsqueda lineal de alfabeto largo es inapropiado, y, en total, la codificación del flujo podría tomar O(nr) tiempo. Una alternativa obvia es el uso de un árbol balanceado para el registro de las sumas parciales de los valores de C_j, mejorando este límite a $O(nlogr)$. Moffat describió una estructura de árbol implícita que mejora el costo a $O(n+r+H)$, concordando (asintoticamente) el límite que había sido previamente ejecutado por el codificador Huffman.

Jones [J88] también consideró el problema del cálculo de los valores ΣC_j. El sugirió el uso de un árbol extendido, el cual, en un sentido menos amplio, atañe el mismo límite que el árbol implícito. Se codificaron cada una de estas tres alternativas usando como base el codificador aproximado de Neal con $p=10$ y $f=22$.

El método de descomposición binario es también adecuado para la codificación adaptativa, ya que ΣC_j no necesita ser calculado explícitamente. El tiempo de corrida para este método es de $O(nlogr)$.

Codificación de Árbol Extendido. Un segundo uso para los árboles extendidos sugerido por Jones es como un mecanismo para derivar un código prefijo y realizar la codificación por sí mismo. No hay un balance explícito restringiéndose sobre un árbol extendido como lo hay por decirlo así para un árbol Huffman dinámico, y así los códigos generados podrían ser algo pobres. Sin embargo en una larga secuencia de

símbolos, el árbol extendido garantiza generar por lo menos un factor constante más largo que la entropía H del flujo, y corre en un tiempo de $O(n+r+H)$. Se interesó en ver si estos factores constantes eran pequeños o grandes.

Probabilidad Escape. Un importante componente de un codificador adaptativo es el mecanismo usado para estimar la probabilidad de que el siguiente símbolo sea "nuevo", y varios métodos diferentes han sido descritos (véase [HV92,WB91]). La tabla muestra la compresión que resulta cuando éstos son usados con el codificador exacto de 64 bits con los datos de prueba. Los métodos A y B se desempeñaron pobremente y, sobre los otros, XC y D tuvieron una ligera ventaja sobre el método C.

Los diferentes métodos también tienen un precio en términos de requerimiento de recursos. El método B requiere de montos significativos de espacio extra, ya que los símbolos para los cuales c=1 deben mantenerse en un estanque de "escapes" potenciales hasta su segunda aparición. El método XC fue más lento que los otros métodos, ya que cada símbolo no-escape requiere de dos pasos de codificación, uno para transmitir la información "el símbolo no es nuevo", y entonces el segundo para identificar el símbolo. El método D fue marginalmente más lento que los métodos A y C, debido a que los conteos son incrementados al doble por símbolo no-escape más que antes. Sin embargo, esta diferencia fue pequeña, y para el resto de la evaluación se usó el método D con $Pr(esc)=r/2n$ y $Pr(i)=(2C_i-1)/2n$.

Método	Compresión (% entropía)				Global
	c	p	n	t	
A	+0.01	-0.19	0.00	+0.01	-0.10
B	+0.02	-0.34	0.00	+0.02	-0.18
C	+0.02	-4.50	-0.14	+0.02	-2.48
D	+0.01	-4.59	-0.16	+0.01	-2.54
XC	+0.01	-4.60	-0.16	+0.01	-2.55

Cálculo de los Escapes para los métodos adaptativos.

El tratamiento de los símbolos nuevo en los codificadores adaptativos Huffman de Gallenger, Knuth, y Vitter corresponden más cercanamente al método A. Sin embargo, también se implementaron estos para el uso del método D. El codificador extendido corresponde más cercanamente al método C.

Resultados.

Velocidad. La tabla siguiente muestra la velocidad de codificación y descodificación de los codificadores adaptativos. Nótese el bajo desempeño del método de Gallenger en el archivo "P", causado por el alfabeto largo. Si no fuera por este pobre funcionamiento, predecido por Gallenger y remediado por Knuth, éste sería el método más rápido. Desafortunadamente la cura es, en algunos casos, peor que el

problema, y ambos métodos Knuth y Vitter fueron más lentos que el Gallanger en archivos sesgados.

El codificador aritmético usando la lista de movimiento-al-frente también se desempeñó pobremente en el alfabeto largo, y sería preferible ya sea el árbol implícito o el árbol extendido para calcular ΣC_j, incluso para alfabetos pequeños o sesgados.

El codificador extendido corrió muy rápido, y por su velocidad, fue sin igual; el método de descomposición binaria, a pesar del código simple requerido durante la adaptación, no fue competitivo.

Método	Vel. Codif.				Vel. Decodif.				Global
	c	w	n	t	c	w	n	t	
Gallanger	62	1	103	123	61	1	105	124	5
Knuth	40	12	64	70	36	11	60	67	28
Vitter	26	9	41	40	20	8	33	37	18
Neal+lista-maf	44	<1	61	73	31	<1	41	48	2
Neal+árbol implícit	52	27	69	71	36	21	45	46	39
Descomp. binaria	21	9	17	26	17	9	18	27	15
Código extendido	82	39	133	119	87	40	174	175	79

Códigos adaptativos velocidad en '000 símbolos por segundo.

Requerimientos de Memoria. En contraste al método estático, los códigos dinámicos Huffman fueron extravagantes en el espacio de memoria, requiriendo típicamente 50-70 bytes por símbolo. Los codificadores aritméticos fueron más frugales, 16 bits por símbolo para la lista-maf y el árbol implícito; mientras el codificador Neal+árbol extendido y el programa Código-extendido requirieron ambos de como 20 bytes por símbolo debido a los punteros involucrados en la manipulación del árbol extendido. Sin embargo, todos los métodos exceptuando los dos que usan árboles extendidos requieren, en ausencia de un límite alto apriori en el tamaño del alfabeto, que el espacio para los arreglos sea reubicado según crezca el alfabeto.

Compresión. La tabla muestra la compresión que resulto con el uso de métodos adaptativos. Tal vez la característica más interesante es el notar que es posible para un codificador adaptativo desempeñar mejor que la entropía propia, particularmente cuando el alfabeto es largo. Para ilustrar el efecto, considérese, como un caso extremo, la entrada 1,2,3,...n. La entropía misma de este flujo es $n\log_2 n$ bits. Sin embargo, usando los métodos C y D, este flujo sería representado en n-1 bits, uno por cada nuevo símbolo (después del primero). Con $n=10^6$, esto da una figura de compresión de -99.5%. Aún más remarcable es el método XC con el flujo 1,1,2,3,...n-1, el cual requerirá de 41 bits de salida cuando $n=10^6$. La explicación de esta aparente paradoja es simple: la entropía misma es una medida relativa a un modelo, y al

distinguir en los codificadores adaptativos entre las ocurrencias primera y las subsecuentes de cada símbolo, no se esta usando mas un modelo de orden cero.

Comparado a los métodos que mantienen probabilidades explícitas, el código extendido da una compresión relativamente pobre.

Método	Compresión (% entropía) c	w	n	t	Global
Gallanger	+0.66	-4.28	+6.10	+4.96	-1.06
Knuth	+0.66	-4.28	+6.10	+4.96	-1.06
Vitter	+0.66	-4.29	+6.10	+4.96	-1.06
Neal, p=10	+0.01	-4.59	-0.16	+0.01	-2.54
Descomp. binaria	+3.68	-2.68	+1.77	+2.37	-0.15
Código-extendido	+15.92	+10.13	+13.90	+17.36	+12.64

Códigos adaptativos: Compresión usando el método D

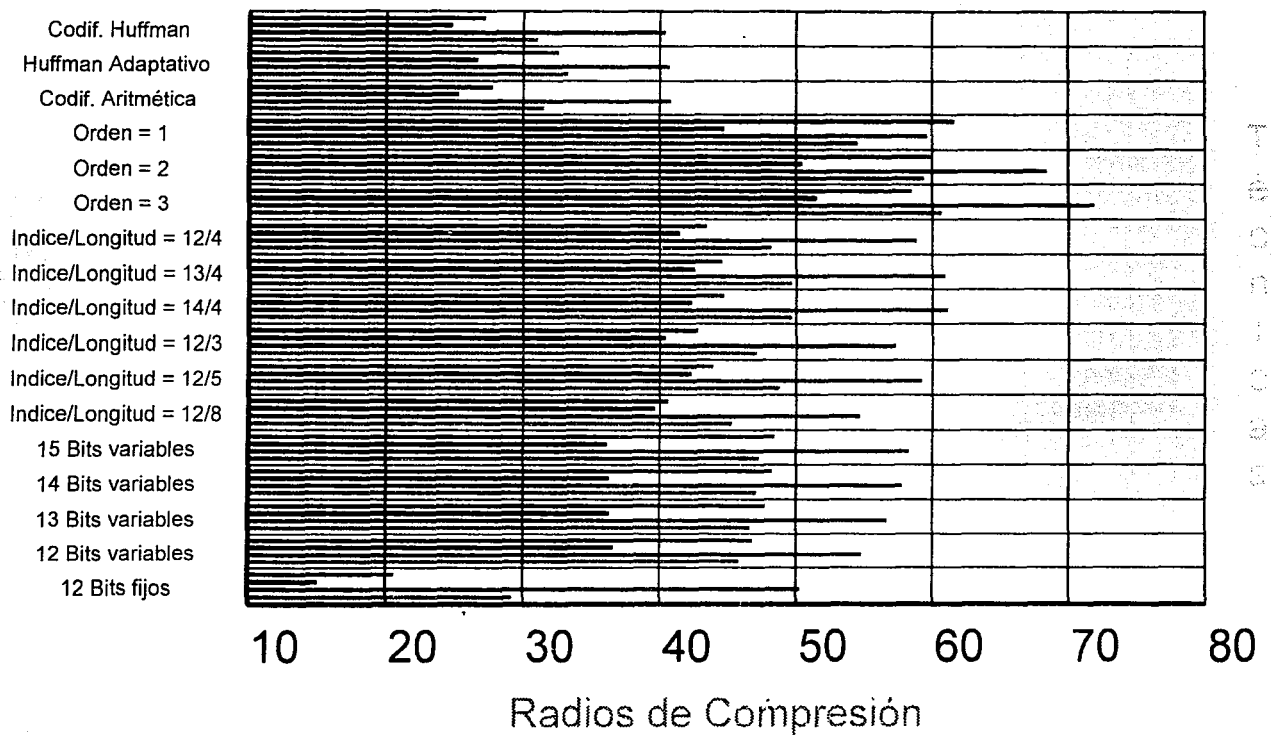
Estadísticas para Programas de Compresión

Se dan a continuación las estadísticas de algunos algoritmos de compresión mencionados anteriormente. Los archivos que se que se comprimieron para generar las estadísticas consistieron en cerca de 6 megabytes de datos divididos en tres categorías igualmente duras. La primera categoría es texto, consistente de manuscritos, programas memos, y otros archivos legibles. La segunda categoría consiste en datos binarios, incluyendo archivos bases de datos, archivos ejecutables, y datos de hojas de cálculo. La tercera categoría consiste en archivos de gráficas almacenadas en formatos de pantalla de descarga. Las velocidades de compresión y expansión dadas aquí deben ser tomadas con mesura. Primero, no se hizo ningún intento por optimizar estos programas. Segundo, se podrá observar alguna variación dependiendo del compilador usado para construir el ejecutable. La mayoría de los ejecutables se construyeron usando Borland C++ 2.0, pero en algunos casos, los requerimientos de memoria expandida condujeron al uso de Zortech C++ 3.0.

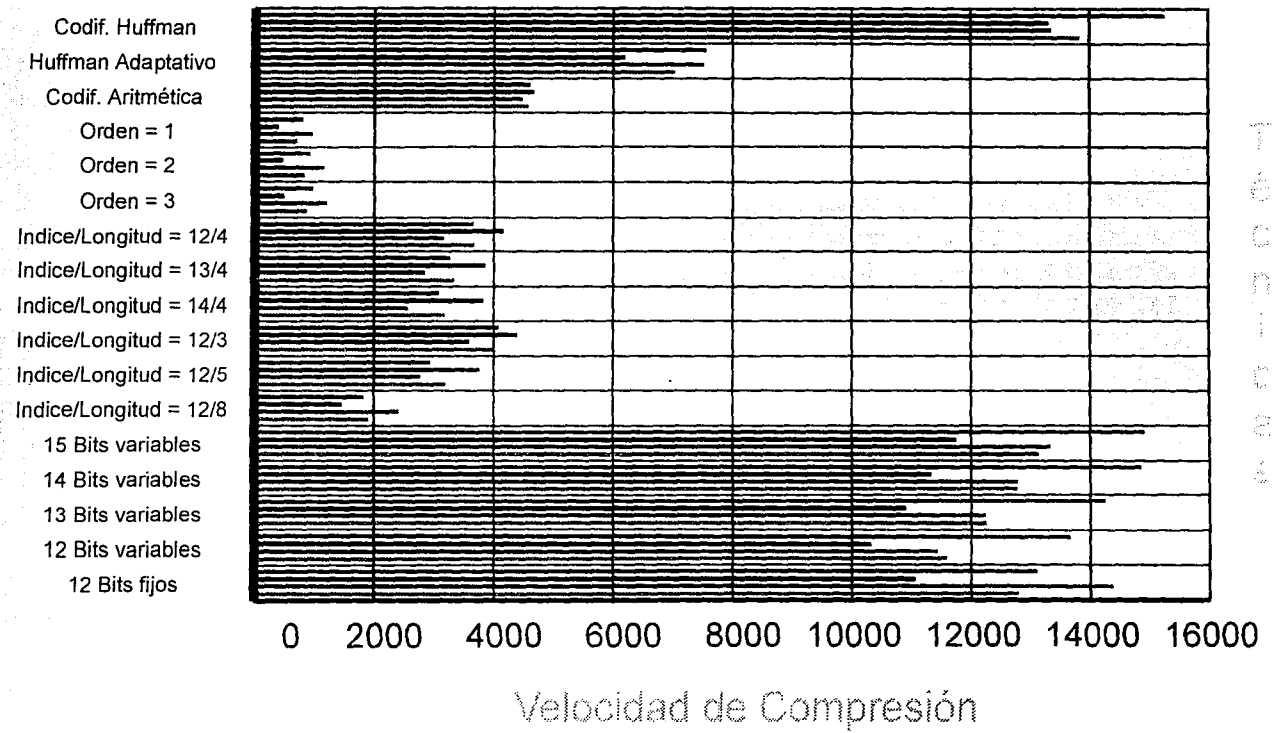
	RADIOS DE COMPRESION Gráficas	Ejecutables	Texto	Global
HUFF.C Codif. Huffman	27.22	24.79	40.38	31.04
AHUFF.C Huffman Adaptativo	32.59	26.69	40.72	33.27

ARITH.C Codif. Aritmética	27.78	25.25	40.81	31.51
ARITHN.C Orden=1	61.66	44.74	59.60	54.49
ARITHN.C (1) Orden=2	59.85	50.47	68.43	59.37
ARITHN.C (1) Orden=3	58.51	51.55	71.86	60.67
LZSS.C Indice/Longitud=12/4	43.45	41.44	58.83	48.22
LZSS.C Indice/Longitud=13/4	44.57	42.56	60.91	49.69
LZSS.C (2) Indice/Longitud=14/4	44.71	42.32	61.10	49.70
LZSS.C Indice/Longitud=12/3	42.83	40.38	57.27	47.10
LZSS.C Indice/Longitud=12/5	43.91	42.31	59.21	48.81
LZSS.C Indice/Longitud=12/8	40.60	39.60	54.67	45.29
LZW15V.C 15 bits variables	48.44	36.15	58.28	47.31
LZW15V.C 14 bits variables	48.23	36.27	57.76	47.11
LZW15V.C 13 bits variables	47.76	36.34	56.71	46.65
LZW15V.C 12 bits variables	46.78	36.61	54.82	45.81
LZW12V.C 12 bits fijos	20.61	15.07	50.32	29.20

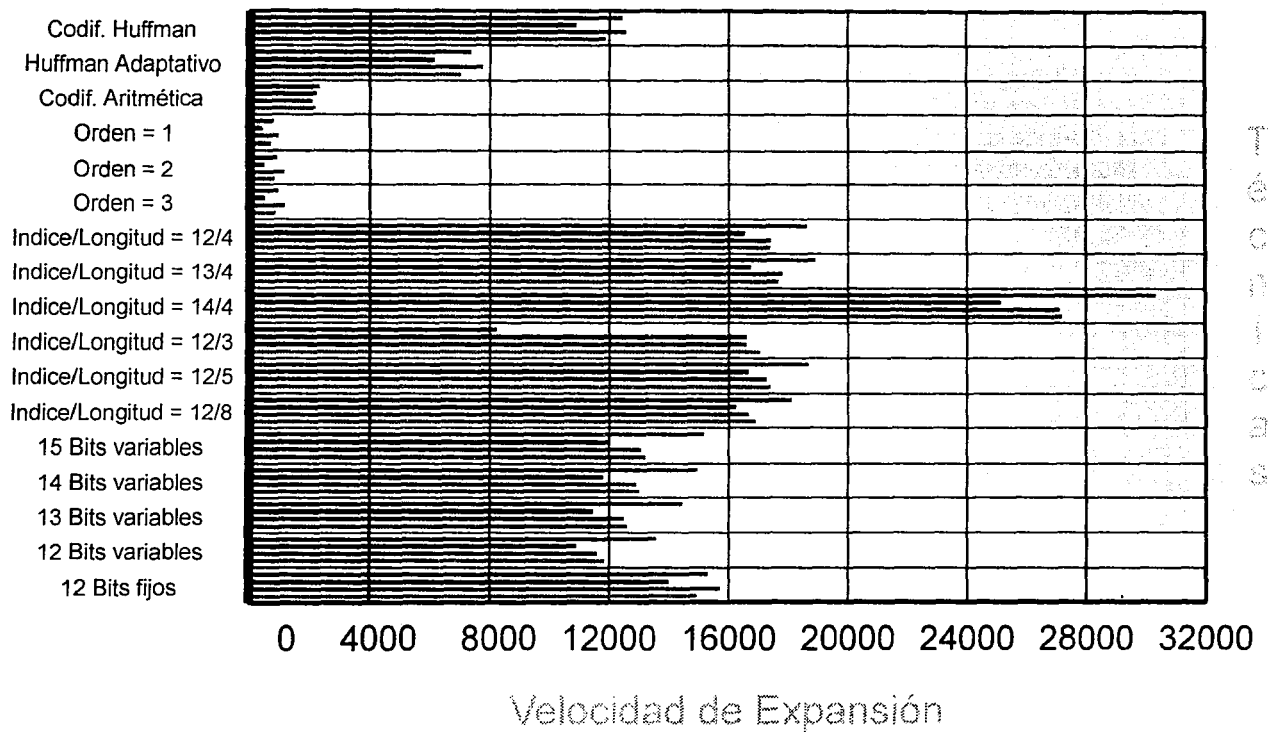
Gráficas para Programas de Compresión



Gráficas para Programas de Compresión



Gráficas para Programas de Compresión



	VELOCIDAD DE COMPRESION			
	Gráficas	Ejecutables	Texto	Global
HUFF.C Codif. Huffman	15273	13306	13353	13835
AHUFF.C Huffman Adaptativo	7553	6200	7523	7028
ARITH.C Codif. Aritmética	4616	4667	4474	4584
ARITHN.C Orden=1	804	395	953	702
ARITHN.C (1) Orden=2	929	462	1158	834
ARITHN.C (1) Orden=3	975	483	1206	871
LZSS.C Indice/Longitud=12/4	3657	4159	3165	3671
LZSS.C Indice/Longitud=13/4	3268	3853	2839	3336
LZSS.C (2) Indice/Longitud=14/4	3082	3832	2563	3180
LZSS.C Indice/Longitud=12/3	4093	4392	3594	4027
LZSS.C Indice/Longitud=12/5	2942	3764	2773	3194
LZSS.C Indice/Longitud=12/8	1831	1460	2411	1899
LZW15V.C 15 bits variables	14913	11744	13332	13140
LZW15V.C 14 bits variables	14850	11332	12783	12769

LZW15V.C 13 bits variables	14256	10904	12241	12257
LZW15V.C 12 bits variables	13669	10320	11432	11591
LZW12V.C 12 bits fijos	13106	11057	14379	12786

VELOCIDAD DE EXPANSION

	Gráficas	Ejecutables	Texto	Global
HUFF.C Codif. Huffman	12428	10892	12560	11891
AHUFF.C Huffman Adaptativo	7366	6141	7755	7041
ARITH.C Codif. Aritmética	2320	2205	2074	2188
ARITHN.C Orden=1	793	405	944	700
ARITHN.C (1) Orden=2	905	472	1136	824
ARITHN.C (1) Orden=3	946	492	1173	855
LZSS.C Indice/Longitud=12/4	18622	16526	17424	17394
LZSS.C Indice/Longitud=13/4	18899	16716	17798	17673
LZSS.C (2) Indice/Longitud=14/4	30334	25120	27120	27196
LZSS.C Indice/Longitud=12/3	8248	16572	16576	17047

LZSS.C Indice/Longitud=12/5	18680	16674	17266	17409
LZSS.C Indice/Longitud=12/8	18113	16230	16673	16879
LZW15V.C 15 bits variables	15167	11986	13077	13206
LZW15V.C 14 bits variables	14942	11804	12911	13018
LZW15V.C 13 bits variables	14445	11462	12495	12609
LZW15V.C 12 bits variables	13576	10893	11608	11845
LZW12V.C 12 bits fijos	15326	13978	15705	14950

NOTAS:
(1) Construido con Zortech 286 DOS Extender, a fin de accesar toda la memoria extendida. Modelos de alto orden pueden usar megabytes de memoria.

(2) Construido con Zortech 386 DOS Extender. Uno de los arreglos en el programa fue más largo que 64K, y esta fue una fácil manera de reconstruir el código sin el uso de punteros "inmensos" basados en MS-DOS.

Los radios se expresan como 100 * (tamaño comprimido / tamaño original)

Compresión Al Vuelo (On-the-Fly)

Los compresores al vuelo son paquetes de programas que incrementan el espacio de almacenamiento, al comprimir y descomprimir automáticamente los datos. Estos compresores usan una variedad de algoritmos de compresión reversibles y sus creadores son herméticos respecto a las técnicas exactas utilizadas. Aún así, se presentan los resultados de las pruebas de banco efectuadas a algunos paquetes de compresión.

En las pruebas que se le hicieron a los paquetes de compresión para PC y para Mac, se buscó que fueran lo más similar posible. Para probar la velocidad se corrieron una mezcla de aplicaciones (procesadores de palabras y bases de datos) y pruebas de bajo nivel. Para medir las puras velocidades, se corrieron operaciones de

E/S de archivos secuenciales y aleatorias, en varios datos representativos de los archivos mas comunes: texto, ejecutables, gráficas bit-mapeadas, bases de datos, y datos precomprimidos. También se midieron los radios de compresión para cada uno de los cinco tipos de datos. En el lado de las PC, donde las aplicaciones de compresión se expandieron sobre una partición entera, se llenó la partición con datos hasta que el sistema se desbordó. La cantidad de datos escritos, comparados con el espacio físico en el disco, proporcionaron el radio de compresión.

La figura muestra el radio de compresión compuesto -el promedio no-pesado de los radios de compresión para cada tipo de archivo-. Se usó una Compaq Deskpro 386/33L como PC de prueba. La Mac de prueba fue una SE/30.

RESULTADOS DE REFERENCIA

Radios de compresión e Indices de rendimiento efectivo para las tres principales categorías de aplicaciones. Para cada tipo de aplicación, el radio de compresión describe la compresión sobre datos representativos. Indices de rendimiento efectivo para Procesadores de Palabra y Bases de Datos son una combinación de resultados de referencias de aplicaciones y pruebas de bajo nivel. Los numeros altos son mejores.

	PROCESADORES DE TEXTO		BASES DE DATOS		GRAFICAS	
	Radio de Compresión	Rendimiento Efectivo	Radio de Compresión	Rendimiento Efectivo	Radio de Compresión	Rendimiento Efectivo
Productos PC						
DoubleDensity	2.00	0.38	2.00	0.26	2.00	0.37
Stacker	2.06	1.05	1.92	0.95	2.10	1.02
SuperStar Pro	1.50	0.62	1.40	0.64	1.30	0.56
XtraDrive	1.70	0.73	1.70	0.62	1.72	0.73
Productos Mac						
AutoDoubler	1.66	0.46	2.06	0.46	1.20	0.42
Stuffit SpaceSaver	2.18	0.60	2.25	0.61	1.30	0.61
SuperDisk	2.28	0.46	2.04	0.46	1.49	0.54
TimesTwo	1.79	0.52	1.70	0.46	1.20	0.40

A continuación se presentan los resultados de las pruebas de otros cinco paquetes de compresión que son:

- PKZIP 1.10, un popular archivador shareware para MS-DOS.
- ARJ 2.10, un archivador para MS-DOS de uso no comercial.
- LHA 2.10, un archivador freeware para MS-DOS.
- PAK 2.51, un archivador shareware de bajo costo.
- COMPRESS, utilidad UNIX para numerosas plataformas.

Radio de Compresión		Vel. de Compresión (bytes/seg)		Vel. de Expansión (bytes/seg.)	
ARJ	41.4%	PKZIP	13987	PKZIP	39745
LHA	41.8%	ARJ	13623	ARJ	34652
PKZIP	43.8%	LHA	12168	LHA	28090
PAK	45.0%	PAK	9452	PAK	20979
COMPRESS	52.6%	COMPRESS	6132	COMPRESS	8516

NOTA: Los radios se expresan como

100 * (tamaño comprimido / tamaño original) %

Selección del Algoritmo de Compresión

En orden de hacer una elección racional de un algoritmo de compresión, es necesario asegurar que la operación de compresión está mejorando todo el sistema y no degradándolo. Una elección pobre del algoritmo puede resultar en un gran incremento de complejidad y costo junto con un incremento menor en el desempeño, sobre el sistema original. Es deseable, por lo tanto, tener un procedimiento lógico para la selección del algoritmo.

El siguiente procedimiento asume que están dados el radio de compresión requerido y la distorsión máxima (en caso de compresión irreversible).

1.- Calcúlese la entropía (H) y el máximo radio de compresión (Rmáx = $\log_2 M / H$, donde M es el número de niveles).

2.- Compárese el radio máximo de compresión con el que se requiere. Si el radio de compresión que se requiere excede al máximo radio de compresión, entonces será conveniente usar una combinación de reducción de entropía (irreversible) y de reducción de redundancia (reversible). En este caso, ir al paso 6.

3.- Si el radio de compresión que se requiere no excede al máximo radio de compresión, entonces seleccione dos o más algoritmos de reducción de redundancia que cumplan con el requerimiento de radio de compresión.

4.- Compárense estos algoritmos candidatos en base a la distorsión total y a la complejidad.

5.- Escoja el algoritmo que cumpla con el radio requerido y con la distorsión requerida con la mínima complejidad. Esto completa la selección.

6.- Escójase un esquema de reducción de entropía que introduzca menos de la máxima distorsión permisible y que provea de un radio de compresión aproximadamente igual al cociente del valor requerido dividido por el máximo valor.

7.- Seleccione dos o mas algoritmos de reducción de redundancia que provean el máximo radio de compresión.

8.- Ir a los pasos 4 y 5.

CAPITULO IV

Consideraciones y Propuestas

Cuando se pretenda incorporar las técnicas de compresión de datos en un sistema, sería conveniente considerar las ventajas y las desventajas que conllevan cada una de éstas técnicas en mayor o menor grado y que afectan en forma determinante el aprovechamiento del sistema.

En éste capitulo se veran algunas de las principales ventajas y desventajas de la compresión de los datos así como cuatro propuestas viables y convenientes para una aplicación más eficiente y adecuada de ciertas técnicas en la compresión de texto: la utilización de metodos formales para la descripción uniforme de las técnicas de compresión, la valoración de la técnica Huffman como un buen método para la codificación con grandes alfabetos o para sistemas que requieran de una gran confiabilidad, la utilización de la técnica de codificación de longitud corrida como un método para la compresión del diccionario de los verificadores de ortografía y, la utilización del modelado multiplicativo en los modelos adaptativos de alfabetos pequeños de bajo consumo de memoria.

Ventajas y Desventajas de la Compresión de Datos

Las técnicas de compresión pueden tener un efecto positivo no solo en el costo del almacenamiento y de la transferencia de datos, sino también, en las áreas de seguridad de bases de datos, procedimientos de respaldo y recuperación, así como en el mejoramiento de las bases de datos. Distingamos algunas de las ventajas de la compresión de datos.

•*Almacenamiento.* Claro que la más obvia ventaja de la compresión de datos es el reducir los requerimientos de almacenamiento de la información, y a la vez, incrementar la capacidad del medio de almacenaje. A pesar de que los avances de la tecnología continuamente reducen los costos e incrementan la capacidad del medio físico de almacenamiento de datos, el interés y la necesidad de la investigación sobre la compresión de datos se acrecienta constantemente. La principal razón para esto es el explosivo crecimiento en el almacenamiento de datos y en las aplicaciones de procesamiento de datos que están fuera del paso de los avances en la tecnología.

•*Transferencia de Datos.* Puesto que la compresión de datos usa un pequeño número de bytes, la transferencia de los datos comprimidos requiere de menos tiempo, resultando en una proporción de transferencia más alta. En los sistemas de E/S, ésta rápida proporción de transferencia del disco a la memoria principal o

viceversa, puede reducir el tiempo de transacción y el tiempo de respuesta del usuario. Además, puesto que esto también reduce la carga en el canal E/S, el CPU puede procesar muchos más requerimientos de E/S y esto incrementa la utilización del canal. Pero más importante es el rol de la compresión al reducir los costos de transmisión de tremendas cantidades de datos en enlaces de comunicación a larga distancia.

•*Seguridad en los Datos.* Esta ventaja es un efecto colateral de la compresión de datos. Para usar o interpretar datos comprimidos, tienen que ser restaurados a alguna forma descomprimida. Para conseguir esto, se debe de disponer de un algoritmo de descodificación junto con una clave de descodificación (una tabla, estructura de árbol, etc.). Incluso con el mismo algoritmo, archivos diferentes pueden usar diferentes claves de descodificación. Estas claves se pueden separar de los archivos comprimidos para restringir el acceso a los datos correspondientes. Pero; debido a que la seguridad de los datos es un derivado, la compresión de datos solo puede mejorar la seguridad de los datos, y no así, por sí misma, proveer el alto nivel de seguridad necesario para algunas aplicaciones.

•*Respaldo y Recuperación.* En la mayoría de los sistemas, el esquema básico de respaldo/recuperación se basa en mantener copias de los archivos de las bases de datos, junto con el rastreo en cinta. Copias de respaldo de las bases de datos se pueden mantener en su forma comprimida, reduciendo el número de cintas necesarias para el almacenamiento de los datos y reduciendo el tiempo de lectura-escritura en estas cintas. Esto también reducirá el tiempo empleado en la recuperación, y diminuirá el lapso de tiempo entre puntos de registro de la cintas.

•*Mejoramiento del Desempeño.* En algunas aplicaciones, la compresión de datos puede conducir a otros tipos de mejoras en el desempeño de los sistemas. Por ejemplo, en algunas estructuras de índice es posible, a través de la compresión, empacar más claves dentro de un bloque índice dado. Cuando se busca en la base de datos por un valor clave dado, la clave primeramente se comprime y entonces comparada con las claves comprimidas en el bloque índice. El efecto neto es que pocos bloques tienen que ser recobrados y así el tiempo de búsqueda se reduce. Argumentos similares se pueden aplicar para los buffers usados para almacenar partes de la base de dato en la memoria principal. Comprimiendo registros, se incrementa la capacidad efectiva del buffer y la probabilidad de encontrar el registro clave.

•*Usuario.* Cuando se implementan apropiadamente, los procesos de compresión y descompresión pueden ser totalmente transparentes al usuario.

Aunque la compresión de datos tenga algunas ventajas muy atrayentes, introduce varios problemas como son:

•*Sobrecarga de Procesos.* La sobrecarga causada por el proceso de compresión al codificar y descodificar los datos es uno de las más serias desventajas de la compresión de datos. En algunas aplicaciones esta sobrecarga puede ser tan grande que desalienta cualquier consideración para la aplicación de las técnicas de compresión. Primeramente, el proceso de expansión de las bases de datos de solo-lectura tienen un mayor impacto sobre el desempeño que en el proceso de compresión. Pero, para la mayoría de las bases de datos, que son volátiles, el proceso de codificación es lento y más complicado que el proceso de descodificación. Nótese sin embargo, que hasta recientemente la mayoría de la compresión de datos se ha implementado con programas. Con el advenimiento de la tecnología VLSI, muchas técnicas se abren a su implementación completa o parcial en dispositivos electrónicos, reduciendo así la sobrecarga de procesos dramáticamente.

•*Ruptura de las Propiedades de Dato.* Debido a la manipulaón de los datos, muchas técnicas pueden, y de hecho lo hacen, romper las propiedades de los datos que pueden ser importantes para algunas aplicaciones. Por ejemplo, al no preservar el orden léxico de los datos comprimidos, las ordenaciones eficientes y los esquemas de búsqueda pueden quedar inaplicables. En algunos casos, los datos deben ser expandidos primero, y luego se hace la búsqueda en los datos originales.

•*Portabilidad.* Como en muchos sistemas, las implementaciones en los programas de las técnicas de compresión de datos son codificadas en lenguajes de ensamblador para la eficiencia de una máquina en particular, y debido a que no hay estándares bien definidos en el área, las utilerías de la compresión de datos no son completamente portátiles o fácilmente modificable.

•*Longitud de Salida.* El tamaño de un registro comprimido es usualmente desconocido y depende de la técnica usada, y del contenido del registro. Por ejemplo, diferentes registros de igual longitud pueden ser comprimidos en distintos tamaños. Esta incertidumbre de la longitud de salida crea problemas en el planeamiento de la distribución del espacio y al actualizar registros dentro de archivos comprimidos. Adicionalmente, algunos métodos de búsqueda y acceso resultan más difíciles de implementar, según puedan variar los campos clave.

•*Confiabilidad.* Al reducir la redundancia en los datos, las técnicas de compresión de datos también reducen la habilidad de recobrarse de errores de datos. Por ejemplo, un solo bit de error en la técnica Huffman, causará que el descodificador malinterprete todos los bits subsecuentes, produciendo una salida incorrecta. Este problema por lo menos en el lado de los equipos, es mitigado por el confiable almacenamiento en disco y la tecnología de comunicaciones.

•*Claves de Descodificación.* Como se vio anteriormente, muchas técnicas de compresión crean un gran número de claves de descodificación y tablas que son necesarias para la correcta expansión de los códigos comprimidos. Estas claves y

tablas van en oposición al propósito principal de la compresión de datos, de reducir los requerimientos de almacenamiento. Pero en la mayoría de los casos, el espacio ganado de los datos principales, excede al espacio que necesita ser sumado por las claves o tablas.

Métodos Formales y las Técnicas de Compresión/Descompresión

Una larga variedad de aplicaciones específicas y medio-específicas de las técnicas de compresión de datos han sido desarrolladas para proveer las características específicas de costo-beneficio requerida por aplicaciones diferentes, usando tecnologías diferentes. Mientras esto ha ocurrido con éxito en la practica, se han creado barreras significativas en la comunicación entre los expertos de la compresión de datos que operan en dominios diferentes.

Idealmente, debería de ser posible abstraer de un dominio específico, y de una tecnología específica mixta, y desarrollar un catalogo de técnicas/algoritmos cuya aplicación, eficiencia, y propiedades pueden ser determinadas por un razonamiento formal. Esto requiere que se deduzca una manera resumida de representar todos los datos, inclusive datos expresados en términos de múltiples medias. En pocas palabras, lo que se requiriere es una forma canónica.

Las Técnicas de Descripción Formal (TDF) son probadamente promisorias en la definición de una forma canónica para sistemas de procesamiento de texto y ayudan en derivar isomorfísmos (transformaciones) entre la forma canónica y otros formatos de datos más eficientemente almacenados/transmitidos, y viceversa.

Esto nos habilita para concentrarnos en desarrollar eficientes algoritmos de compresión para la forma canónica, sabiendo que podemos transformarlos para operar en otros formatos de datos.

¿Está Muerta la Codificación Huffman?

Recientes publicaciones sobre la compresión de datos sugieren que la codificación Huffman ha pasado de moda. Estas publicaciones enfatizan la suboptimalidad de los códigos Huffman. Realmente, la "optimalidad" de los códigos Huffman han sido sobrenfatizado en el pasado y no siempre se ha mencionado que los códigos Huffman son óptimos solo si:

- El conjunto de elementos a codificar es fijado a lo largo del archivo; y
- Cada palabra-código es construida consistiendo en un número entero de bits.

Los códigos aritméticos sobreponen las restricciones anteriores y tienen algunas muy fuertes ventajas:

- Permite códigos muy cercanos al límite de entropía.
- Son fácilmente utilizables con un modelo adaptativo, produciendo codificación eficiente si el alfabeto o sus características cambian sobre el archivo;
- El procedimiento de codificación puede ser natural y simplemente extendido para abarcar incluso un alfabeto infinito.
- Pero estas ventajas son a un costo relativo para los códigos Huffman. Algunos beneficios de los códigos Huffman son:
- Los códigos aritméticos corren significativamente mas lento que los códigos Huffman.
- Para grandes alfabetos , la diferencia entre ambos es pequeña, pero incluso para alfabetos pequeños o altamente segados, la simple redefinición del alfabeto puede seguido, reducir enormemente la ventaja.
- Cuando las probabilidades sean inexactas, como en el caso de la generación de modelos para predecirlas, los códigos Huffman pueden ser mas eficientes.
- Un código Huffman puede ser comunicado muy eficientemente por bien establecidas "Representaciones Canónicas".
- Los códigos Huffman, al contrario de los códigos aritméticos, son fuertes contra el error de línea.

Estas consideraciones sobre la codificación Huffman indican que es apropiado reconsiderar su valor en la compresión.

Un Método de Compresión por Diccionario para Verificadores de Ortografía

En algunos verificadores de ortografía, en orden de adquirir rápida velocidad de indagación, es empleado el direccionamiento calculado para comprimir el diccionario sin tratamiento de colisión. Muchas palabras incorrectas pueden tener los mismos valores de dirección como aquellos de las palabras correctas. Por lo tanto, algunas palabras mal escritas no pueden ser identificadas.

Es factible la utilización de un método de compresión reversible que usa un n-grama y codificación de longitud corrida con un índice no-denso para la compresión eficiente del diccionario y rápida recuperación directa de cualquier vocabulario. Se usa el n-grama como un conductor de caracteres (prefijo) de una palabra. La técnica

modificada de n-grama propuesta, es el construir una matriz n-grama, y entonces comprimirla con codificación de longitud corrida. El resto de la cadena (sufijo) que no es codificado en un mapa de bits de n-grama es asignado después del elemento bit del prefijo correspondiente. Ya que puede haber más de un carácter en el sufijo y se le asigna más que un sufijo a un elemento bit, dos bits extra podrían usarse como un delimitador de dos sufijos o caracteres dentro de un sufijo. El arreglo índice no-denso que contiene el mapa entre la matríz y el diccionario comprimido es generado durante la codificación de longitud corrida para acelerar el proceso de indagación del diccionario.

Comparado con los paquetes comerciales de compresión reversible, el método propuesto proporcionaría mayor velocidad de acceso manteniendo un excelente radio de compresión.

Modelado por Modelo Multiplicativo: Una Aplicación a la Codificación Aritmética

Pueden hacerse simples modificaciones a un modelo adaptativo para la codificación aritmética. Estas modificaciones pueden ser usadas por todo modelo que no requiera de una gran capacidad de memoria. Esto no incrementa el tiempo de compresión significativamente pero requiere de más memoria. sin embargo, otros modelos diseñados para la codificación aritmética, que dan un radio de compresión similar requieren de más memoria y son más complicados.

Cuando cualquier método de compresión toma en consideración solo las frecuencias de las ocurrencias de los símbolos en los datos, entonces la eficiencia de la compresión en promedio no puede ser menor a la entropía de los símbolos. Sin embargo es posible sobrepasar esta restricción cuando las dependencias entre los símbolos son tomadas en cuenta. Desafortunadamente el considerar la conexiones entre los símbolos requiere de una gran memoria. Pero no es necesario el recordar todas las frecuencias. Muchas de ellas son insignificantes. Algunos símbolos de un contexto a otro, aparecen muy raramente o en absoluto. Cuando un modelo mantiene frecuencias de símbolos individuales, entonces el modelo puede ser duplicado: el primer modelo puede contabilizar las frecuencias de los símbolos en un cierto contexto, y el segundo las de otro. El modelo puede ser multiplicado muchas veces, no solo duplicado, y uno de los modelos multiplicados se puede escoger por ejemplo de acuerdo al último símbolo.

Esta idea tiene significativas ventajas:

- El modelo multiplicado permite de una forma simple el mejorar el desempeño de los métodos de compresión.
- Da buenos resultados para datos consistentes de símbolos, que pertenecen a un alfabeto pequeño.
- El modelo puede multiplicarse dinámicamente de acuerdo a la memoria libre.

La única desventaja es que no puede aplicarse a modelos consumidores de memoria.

Conclusiones

1.- En general, la motivación más común para el uso de la compresión de datos es un problema de almacenamiento o un requerimiento de diseño. Pero seguido se selecciona una simple técnica que ofrece solo la compresión de datos básica necesaria para la aplicación, en vez de combinar varios métodos, o seleccionar un método de compresión más complejo el cual ofrecería mayores ventajas que solo la compresión de los datos. Además, la compresión de datos puede no ser, y en algunos casos no lo es, deseable. Los beneficios asociados con la compresión de los datos de una base de datos específica son afectados por muchas variables: el tamaño de la base de datos, la cantidad y tipo de redundancia en la base de datos, el tipo y frecuencia de requerimientos de recuperación y actualización, y la eficiencia, requerimientos de memoria y la complejidad de la técnica de compresión. Todos estos factores deben de ser considerados antes de hacer cualquier decisión sobre cual(es) técnica(s) usar.

2..- La reducción de redundancia es un proceso reversible, y como tal, debe operar sobre una fuente de datos discreta. Las dos fuentes discretas comúnmente usadas para la reducción de redundancia son: la fuente estadísticamente independiente, y la fuente de Markov.

3.- Un término teórico del desempeño muy útil para las técnicas de reducción de redundancia es la entropía, que se define como el promedio de información disponible de una fuente de información.

4.- Para una fuente estadísticamente independiente, los códigos Shannon-Fano y Huffman tienen su máximo radio de compresión cuando la probabilidad de cada nivel es una potencia negativa de 2. Cuando éstas probabilidades no son potencias negativas de 2, el código Huffman proporciona un radio de compresión más alto que el código Shannon-Fano.

5.- Se mostraron los detalles de una implementación de la codificación aritmética, remarcando sus ventajas (flexibilidad y su cercanía al optimo) y su principal desventaja su lentitud. Se revisó un codificador rápido, basado en codificación aritmética de precisión reducida el cual da una mínima pérdida de la eficiencia en la compresión. Se detectó que la pérdida de eficiencia en la compresión comparada con la codificación aritmética exacta es despreciable.

6.- Se introdujo el árbol comprimido, una nueva estructura de datos para una eficiente representación de alfabetos multisímbolos por una serie de opciones binarias. El nuevo esquema de estimación de probabilidad determinístico permite una rápida actualización del modelo almacenado en el árbol comprimido usado solo un byte por cada nodo; el modelo puede proporcionar el codificador de precisión reducida con las probabilidades que necesita. Al escoger alguno de los métodos para computar la probabilidad escape nos habilita el uso del altamente efectivo algoritmo

PPM, y el uso de un modelo de Markov aleatorio mantiene los requerimientos de tiempo y espacio controlables incluso para modelos de alto orden.

Sería ingenuo esperar que la evaluación empírica presentada desentierre claros "ganadores" en las dos categorías de codificación (semi-estáticos y adaptativos). No obstante se pueden sacar algunas conclusiones.

7.- Para la codificación semi-estática en un alfabeto no sesgado, la codificación Huffman aparece preferible. Si el alfabeto es sesgado entonces un codificador aritmético aproximado incrementará la compresión pero en decremento de velocidad. Sobre los codificadores aproximados, el método de Neal parece la mejor opción. Parece haber un pequeño lugar para un codificador de descomposición binaria.

8.- Para la codificación adaptativa, se recomienda el uso de la codificación aritmética, particularmente si hay como prospecto un alfabeto largo o uno sesgado. Además se sugiere el uso del codificador aproximado Neal con una apropiada elección de p; y, al menos que el alfabeto sea pequeño, debe ser acoplado con la implícita estructura de árbol de Moffat. Si se sabe que el alfabeto no es largo o sesgado, y el espacio en memoria es menos importante que la velocidad, el método Gallenger podría ser considerado. El codificador Splay podría ser usado si la pura velocidad fuera más importante que la compresión y el espacio en memoria.

9.- En general podemos considerar a la codificación Huffman como una técnica bastante rápida pero de pobre desempeño, apta para aplicaciones donde la velocidad y la confiabilidad es crucial mas allá del radio de compresión. Cuando los modelos no sean muy representativos de los datos, el método Huffman o el Huffman adaptativo pueden ser una mejor opción.

10.- Con la codificación aritmética sucede el caso contrario. Logra los más altos radios de compresión en relación inversa a su velocidad. Mientras mayor sea el orden del algoritmo tiende a ser mejor su radio de compresión y a reducirse su velocidad de desempeño. Los algoritmos de orden alto son recomendables en los casos donde el espacio sea el factor primordial y la velocidad no sea significativa.

11.- Dentro de los algoritmos LZSS, son recomendables los algoritmos de longitud 4 o 5 que dan un muy buen radio de compresión a una velocidad media. En el caso del LZW se mete muy duro en la pelea con altas velocidades y con buen radio de compresión, destacándose el LZW de 15 bits variables como el mejor en su tipo.

12.- En teoría el LZ78 debería comprimir más y mejor según como el tamaño del diccionario se incremente. Pero esto solo es verdad cuando la longitud del texto de entrada tiende hacia el infinito. En la practica, pequeños archivos comenzarán a sufrir rápidamente conforme el tamaño de código vaya creciendo. También debe considerarse que tanto tiempo de la UPC tomará el manejo del diccionario.

13.- Al evaluar los paquetes comerciales de compresión al vuelo, las tablas nos muestran, en el lado de las PC, que el claro ganador es Stacker. Tiene de todo: el mejor desempeño, buenos radios de compresión de datos aunado a una rápida

operación, una buena interface para Windows y DOS, un juego informativo de programas de utilidad, y un manual bien escrito.

14.- En el lado de las Mac, todos los paquetes estuvieron cercanos tanto en el índice de rendimiento efectivo como en el radio de compresión. En el índice de rendimiento efectivo, el SpaceSaver ganó, aunque en los radios de compresión, no hubo un claro campeón. SpaceSaver y SuperDisk corrieron una cerrada carrera; sin embargo se eligió al SpaceSaver en base a la mezcla de desempeño, precio, y el confort que viene de un prestigiado nombre en la compresión.

15.- ARJ fue el mejor de los cinco paquetes de compresión evaluados según la última tabla del capitulo III, seguido muy de cerca por LHA. Habrá que esperar ver las nuevas versiones de los programas para seguir la comparación.

16.- Debido a su cercania al desempeño optimo de compresión, la codificación aritmética se puede utilizar para mejorar otros métodos y actividades relacionadas con la compresión. Los valores de salida de la codificación Ziv-Lempel no están uniformemente distribuidos por lo que se puede usar la codificación aritmética para una compresión posterior de la salida. Por ejemplo, en la compresión de imagenes reversibles seguido se utiliza la codificación predictiva por lo que es común encontrar que la predicción de los errores sigue una distribución de Laplace; así, si se usan tablas precomputadas por la codificación aritmética se pueden obtener excelentes radios en la compresión de imágenes.

A pesar de que se ha contado con técnicas efectivas de compresión de datos por alrededor de 40 años, todavia permanecen varias areas en las cuales se requiere mayor indagación. Las investigaciones futuras deben de encaminarse a lograr que la compresión de datos sea una mas viable y menos misteriosa herramienta para el diseñador de bases de datos. Algunas de estas areas se describen a continuación:

- *La Transmisión de Datos.* Según dependamos mas y mas de la información generada por computadora, también necesitaremos mas y mas de la tranmisión de los datos generados. Se debe poner mayor atención en diseñar mejores algoritmos de compresión o por lo menos usar y modificar los existentes para reducir el costo de las transmisiones. En particular, estos algoritmos deben de tener una muy baja tolerancia a la propagación de errores y a demoras en el almacenamiento antes o despues de que los datos sean empaquetados para su transmisión.

- *Técnicas para la Electrónica.* Poco trabajo se ha hecho sobre la incorporación de las técnicas de compresión de datos a la electrónica. Generalmente se comienza comparando los costos y el desempeño de las diferentes técnicas conocidas para despues identificar al mejor de los métodos disponible para su implementación en dispositivos electrónicos.

- *Compresión de los Matadatos.* Los metadatos o los archivos de datos sobre datos pueden llegar a ser un poco largos. Aunque en la compresión de los metadatos se pueden usar las técnicas estandar, es muy importante que el acceso a los

metadatos sea rapido y eficiente, ya que los metadatos son fundamentales para el acceso, interpretación y despliegue de la información básica. Se debe enfocar la busqueda en los efectos sobre el desempeño del sistema al comprimir metadatos, así como en el tiempo de procesamiento para expandir los metadatos, el tamaño relativo de los metadatos en comparación con los datos basicos y el número de accesos necesarios para cada acceso de los datos basicos.

Referencias y Fuentes Documentales

[A92] Gersho Allen, **"Vector Quantization and Signal Compression"**, Kluwer Academic, Boston, 1992, 732.

[BCW90] T.C. Bell, J.G. Cleary y I.H. Witten, **"Text Compression"**, Prentice-Hall, 1990.

[C85] V. Capellini, **"Data Compression and Error Control Techniques with Aplications"**, Academic, Londres, 1985, 304.

[CKW91] D. Chevion, E.D. Karnin, y E. Walach, **"High Efficiency, Multiplication Free Aproximation of Arithmetic Coding"**, In Proc. DCC, 1991, 43-52.

[D76] Lee D. Davisson, **"Data Compression"**, Hutchinson & Ross, Stroudsburg Pennsylvania, 1976, 407.

[G78] R.G. Gallenger, **"Variations on a Theme by Huffman"**, IEEE Tr. Inf. Th., IT-24:668-674, 1978.

[GW93] Rick Grehan y Stan Wszola, **"Shrink to Fit"**, Byte, Abril, 1993, 150-162.

[HL90] D. Hirschberg y D. Lelewer, **"Efficient Decoding of Prefix Codes"**, C. ACM, 33:449-459, 1990.

[HV92] P.G. Howard y J.S. Viterr, **"Practical Implementations of Arithmetic Coding"**, Kluwer Academic, Norwell Massachusetts, 1992, 85-512.

[J88] D. Jones, **"Aplication of Splay Trees to Data Compression"**, C. ACM, 31:996-1007, 1988.

[K85] D. Knuth, **"Dynamic Huffman Coding"**, J. Alg., 6:163-180, 1985.

[L91] G.G. Langdon, **"Probabilistic and Q-coder Algorithms for Binary Source Adaptation"**, In Proc. DCC, 1991, 13-21.

[M90] A. Moffat, **"Implementing the PPM Data Compression Scheme"**, IEEE Tr. Comm., 38:1917-1921, 1990.

[M92] Mark Nelson, **"The Data Compression Book"**, M&T Books, EU., 1992, 285.

[N87] R.M. Neal, **"Fast Aritmetic Coding using Low-Precision Division"**, Manuscrito, Calgary, 1987.

[RH93] Mark A. Roth y Scott J. Van Horn, **"Database Compression"**, SIGMOD RECORD, Vol. 22 3:31-39, 1993.

[RM89] J. Rissanen y K.M. Mohuiddin, **"A Multiplication-Free Multialphabet Arithmetic Code"**, IEEE Tr. Comm., 37:93-98, 1989.

[S92] James A. Storer, **"Image and Text Compression"**, Kluwer Academic, EU., 1992, 354.

[SC93] James A. Storer, Martin Cohn, **"Proceedings Data Compression Conference"**, Snowbird, Utah, March-April 1993, 450-490.

[V89] J.S. Vitter, **"Algoritm 673: Dynamic Huffman Codes"**, J. ACM, 1987.

[WB91] I.H. Witten y T.C. Bell, **The zero frecuency problem: Estimating the probabilities of novel events in adaptive text compression"**, IEEE Tr. Inf. Th., 37:1085-1094, 1991.

[WN91] Williams, Ross Neil, **"Adaptive Data Compression"**, Kluwer Academic, Boston, 1991, 365.

[WNC87] I.H. Witten, R. Neal, y J.G. Cleary, **"Arithmetic Coding for Data Compression"**, C. ACM, 30:520-541, 1987.

Glosario

Adaptativa(o), compresión y modelado

Las técnicas de compresión de datos que usan un modelo pueden usar un modelo fijado por el flujo entero que están procesando, o bien, modificar el modelo según el flujo es procesado. Las técnicas que modifican el modelo según se va procesando se dice que usan modelado adaptativo. Un ejemplo de una técnica de compresión adaptativa sería la compresión LZW.

Alfabeto

Un alfabeto es el conjunto de todos los posibles símbolos que pueden aparecer en un mensaje. Por ejemplo, cuando se comprimen archivos de texto ASCII, el alfabeto consistiría de los caracteres del 0x00 hasta 0x7f.

Archivo

Un archivo es un volumen o fichero conteniendo una o más fichas que pudieron o no, haber sido comprimidas. Un archivo es típicamente usado de manera conveniente para almacenar o transportar fichas. Programas como ARC y PKZIP comprimen los ficheros antes de ubicarlos en los archivos.

ARC, programa MS-DOS

ARC es un programa comercial de archivamiento creado por System Enhancement Associates, de Wayne, N.J. ARC fue una de las primeras utilerías de compresión/archivamiento en conseguir una amplia popularidad en el mundo computacional, comenzando a mediados de los años 1980s.

ARJ, programa MS-DOS

ARJ es un programa comercial de archivamiento creado por Robert Jung. ARJ está libre de cargo para uso individual, pero los usuarios comerciales deben pagar una licencia. ARJ también se proporciona con un código fuente ANSI C para la extracción de ficheros de archivos ARJ que pueden ser distribuidos sin restricciones.

CCITT

CCITT es el Comité Consultivo Internacional del Telégrafo y el Teléfono. Esta organización de estándares es responsable de sancionar muchos métodos de compresión y transmisión usados en la actualidad, incluyendo varias técnicas MPC y MADPC, transmisiones de fax, e incluso los estándares GUEF y GEIM.

Códigos, Codificación

Los símbolos a ser almacenados o manipulados por una computadora son convertidos a códigos. Este proceso es referido como codificación. Dos de los más comunes métodos de codificación de texto escrito son el ASCII y el EBCDIC. La compresión de datos puede ocurrir si se usan métodos más eficientes de codificación, como la codificación Huffman.

Codificación Aritmética

Las técnicas tradicionales de codificación como el ASCII o la codificación Huffman codifican los símbolos en un único patrón de bits. La codificación aritmética en cambio, toma un texto entero y lo codifica como un solo número de punto flotante menor que 1 y mayor o igual a 0. La codificación aritmética puede codificar los textos mas eficientemente eliminando los efectos cuantitativos de otras técnicas de compresión.

Codificación por Bloques

La compresión de imágenes se hace frecuentemente, codificando pequeño bloques de imagen independientes unos de otros. Por ejemplo, el algoritmo GUEF usa un tamaño de bloque de 8 por 8 cuando comprime gráficas.

Codificación Huffman

Huffman es un método para codificar símbolos que varía la longitud del símbolo en proporción a su contenido de información. Los símbolos con una baja probabilidad de aparición son codificados con un código usando muchos bits. Los símbolos con una alta probabilidad de aparición son representados con un código usando pocos bits. Los códigos Huffman pueden ser descodificados apropiadamente debido a que obedecen a la propiedad del prefijo, lo que significa que ningún código puede ser un prefijo de otro código.

Codificación Lineal Predictiva

CLP es una técnica de codificación que transmite datos de voz usando un modelo del sistema vocal.

Codificación de Longitud Corrida (Run Lenght Encoding)

La codif. de longitud corrida o CLC, es una técnica simple usada para comprimir series de símbolos idénticos en un flujo de datos. Típicamente el CLC codifica una serie de símbolos como un símbolo y un conteo. Por su simplicidad de implementación es frecuentemente usado en programas de compresión, aunque su desempeño es relativamente pobre.

Codificación Shannon Fano

La codificación Shannon Fano es una técnica de codificación desarrollada en los años 1950s que procuraba minimizar el número de bits usados en un mensaje cuando las probabilidades de los símbolos en el mensaje eran

conocidas. La codificación Shannon Fano ha sido generalmente reemplazada por la codificación Huffman que produce juegos de códigos de probabilidad óptima, resultando en un desempeño marginalmente mejor que los códigos Shannon Fano.

Codificación Escape

Un código Escape es un símbolos especial usado para "escapar" del contexto corriente. Dentro de la compresión de datos, los códigos escape son frecuentemente usados cuando un símbolo, no encontrado en el diccionario corriente de símbolos, necesita ser codificado. El código Escape le indica al descodificador cambiar a un contexto diferente, donde el símbolo pueda ser apropiadamente codificado.

COMPACT, programa UNIX

COMPACT es un programa UNIX que usó la codificación dinámica Huffman para comprimir archivos.

COMPACT, programa UNIX

COMPACT es un programa UNIX que usa una implementación LZW para comprimir archivos. COMPRESS ha difundido su uso en la comunidad UNIX y está disponible para el dominio público.

Compresión Irreversible

La compresión irreversible se refiere a una técnica de compresión donde el flujo comprimido no puede ser expandido a una copia exacta del flujo de entrada. La compresión irreversible puede ser usada en representaciones digitalmente almacenadas de un fenómeno analógico como imágenes gráficas y ejemplos almacenados de audio. La habilidad de sacrificar poca resolución permite a los algoritmos irreversibles comprimir archivos a radios significativamente menores.

Compresión Reversible (Lossles)

La compresión reversible es usada para comprimir un flujo de texto para que pueda ser expandido como una copia idéntica del flujo. Este tipo de compresión es normalmente requerida para archivos de datos.

Diccionario, adaptativo/estático

Los métodos de macrosubstitución usan un diccionario para comprimir los datos. Una cadena de símbolos es codificada como un puntero dentro de un diccionario. Un método adaptativo, como el LZ77, está continuamente modificando su diccionario. Un diccionario estático comprimirá un flujo por entero usando el mismo diccionario.

Entropía, contenido de información

La entropía es la medida del contenido de información dentro de un objeto. El concepto de "entropía absoluta" permanece elusivo. En general, la entropía se calcula con respecto a un modelo dado. La entropía puede expresarse en bits. De ésta forma generalmente se refiere como "contenido de información".

Freeware

Freeware es un término aplicado a los programas que son distribuidos sin cargo y pueden ser usados libremente por cualquiera. Se distingue de los programas de dominio publico por el hecho de que el propietario de los programas retiene el derecho de copia del trabajo. El retener los derechos le permite al propietario moderar o controlar cualquier modificación o redistribución del paquete.

GEIM

GEIM representa al Grupo de Expertos de Imágenes en Movimiento. Como el GUEF, el GEIM se refiere a ambos, al grupo y al estándar que el grupo está desarrollando. GEIM es un comité sancionado por la OIS para trabajar sobre la transmisión digital de calidad de vídeo y sonido. El objetivo del GEIM es el de poder mandar una imagen de alta calidad y sonido estéreo a través de un canal de 1.5 Mbps.

GUEF

GUEF representa al Grupo Unido de Expertos en Fotografía. Se refiere como un grupo "unido" ya que su comité es sancionado por el CCITT y por la OSI, dos prominentes grupos de estándares internacionales. Guef refiere tanto al comité como a su trabajo en progreso -un estándar de compresión que defina un método para la compresión de imágenes fotográficas. Las imágenes comprimidas con el algoritmo GUEF se somete a una compresión "irreversible". El nivel de la compresión puede variar, resultando en una perdida o ganancia en la resolución.

LHarc

LHarc es un programa de compresión freeware autorizado por Haruyasu Yoshizaki. Utiliza la variante LZSS de la compresión LZ77, seguido de una etapa de procesado Dinámico Huffman.

LZW, LZSS, LZ-77, LZ-78

Jacob Ziv y Abraham Lempel publicaron un par de documentos en 1977 y 1978 donde describen dos diferentes técnicas de compresión basadas en diccionario. LZ77 substituía cadenas de una ventana de tamaño fijo en un texto previamente visto. LZ78 reconstruye una frase del diccionario de un texto previamente visto, sin ningún limite sobre que tanto antes pudo haber aparecido la frase. Estos documento impulsaron a otros investigadores a refinar

éstas técnicas, resultando en algoritmos de compresión superiores a los codificadores Huffman basados en estadísticas.

Modelo

Los algoritmos de compresión generalmente mantienen un "modelo", que es un conjunto de estadísticas acumuladas que describen el estado del descodificador. Por ejemplo, en un programa de compresión simple, el modelo puede ser la frecuencia de cada símbolo de un archivo de entrada.

MADPC

Modelo Adaptativo Diferencial del Pulso de Código. La codificación estándar MPC es una técnica común para la codificación de datos de audio. Las conversaciones telefónicas y los discos compactos de audio son ambos un uso convencional MPC. MPC toma muestras de la forma de onda a pasos uniformes y codifica el nivel de la forma de onda. MDPC es la Modulación Diferencial del pulso de Código. MDPC no codifica el nivel, y en cambio codifica la diferencia de la última muestra. MADPC da un paso más, y modifica la codificación de la diferencia dependiendo del estado de la forma de onda. La codificación MPC en los sistemas telefónicos utiliza 64 Kbits por segundo. MADPC puede reducir la tasa a 32 e incluso a 16 Kbits por segundo con relativamente poca reducción de la calidad de la voz.

Modelo Estático

Un modelo estático es aquel que no cambia mientras el flujo de los símbolos es comprimido. Un ejemplo de ésto sería un simple programa de compresión de código Huffman de orden 0. La implementación clásica de este programa cuenta todos los caracteres diferentes en un archivo durante el primer paso. Estos datos son usado para construir un árbol de codificación Huffman, que constituye el modelo estático. Entonces se ejecuta el segundo paso sobre los datos para realizar la compresión.

OIS

OIS es la Organización Internacional de Standards. OIS es uno de los cuerpos (junto con el CCITT) involucrados en los esfuerzos de estandarización GUEF y GEIM.

Orden (de modelo)

El orden de un modelo refiere a qué tantos símbolos previos son tomados en consideración cuando se codifica. Por ejemplo, un modelo de orden 0 ignora todo símbolo previo cuando se determina qué código usar para un símbolo dado. Así incluso si el carácter previo dentro de un archivo fuera "q", la probabilidad de una "u" en un modelo de orden 0 no subiría. Un modelo de orden 1 tomaría en cuenta la "q" e incrementaría grandemente la probabilidad de "u".

Frase

Una frase es una cadena de símbolos de una longitud arbitraria. Cuando se programa en C, el término "cadena" usualmente puede ser substituido por frase.

PKZIP

PKZIP es un programa popular de compresión/archivamiento. PKZIP es distribuido vía canales shareware por PKWare, Glendale, WI. Este programa usa diferentes algoritmos de compresión basados en diccionario para comprimir archivos de entrada. Ha adquirido suficiente popularidad en el mundo del MS-DOS para ser aceptado como un "standard", aunque el código fuente del programa permanece en propiedad.

Radios de Compresión

Los radios de compresión son usados para describir las diferencias entre un archivo y una copia comprimida del mismo. Existen varias diferentes maneras de expresar éste número. Un método común es un radio entre la entrada y la salida, como en "un radio de compresión 4:1". Otro método popular es expresar la diferencia entre los archivos como un porcentaje en el rango de 0% a 100% (o más, si la compresión falló en reducir el tamaño del archivo). Algunas personas invierten esta escala, usando 100% como el "mejor" radio de compresión.

Shannon, Claude

Claude Shannon es conocido como el padre de la teoría de la información por sus trabajos hechos en los año 1940s y 1950s. Shannon definió el concepto de "contenido de información" y la entropía en relación a los datos.

Shareware

Shareware se refiere a un común método de distribución de programas. Shareware se basa en el concepto de "pruébalo antes de comprarlo". El usuario del programa shareware esta autorizado por el creador de probar el programa por un tiempo limitado. Después de que el periodo de evaluación ha terminado, el usuario debe registrar el programa si quiere continuar usándolo. El registrarlo consiste generalmente en un pago por el que se proporciona una mejor documentación, soporte y otras consideraciones.

SQ, programa de compresión MS-DOS

Uno de los primeros programas de compresión fue SQ, y su contraparte USQ. Estos dos programas implementaron un simple algoritmo de compresión Huffman de orden 0. Ellos primero se desarrollaron para el sistema operativo CP/M, y fueron implementados para MS-DOS tan pronto como estuvo listo. SQ no realiza archivamiento, así que fue frecuentemente combinado con LU, un programa que combina archivos dentro de una librería. La aparición de ARC como programa Shareware sumergió al SQ en la oscuridad. ARC no solo ofrecía una compresión superior con su algoritmo LZW, sino que combinaba la

compresión con funciones de librería en un solo programa. Esto le permitió comprimir grupos de archivos y moverlos en una sola operación.

Símbolos

En la terminología de la compresión de datos, un símbolo es una unidad atómica de información. Programas de compresión de propósito general frecuentemente comprimen flujos de bytes, donde el byte es la misma cosa que el símbolo. Sin embargo, un símbolo podría ser fácilmente un número de punto flotante, o una palabra hablada, etc.

TAR, programa UNIX

TAR es un programa estándar UNIX para crear archivos. Toma un grupo de archivos combinándolos en uno sólo. TAR no realiza ninguna compresión de los archivos. Los archivos UNIX son frecuentemente creados usando TAR para empaquetar un grupo de archivos juntos, y luego usar COMPRESS para efectuar la compresión. El archivo resultante se encuentra en formato "TAR.Z".

Teoría de la Información

La teoría de la información es el estudio del almacenaje, procesamiento y la transmisión de la información. Esta rama de la ciencia generalmente se reconoce como creada por Claude Shannon en los laboratorios Bell poco después de la segunda guerra mundial.

Token

Un objeto usado para codificar algo. En la compresión basada en diccionario, un token es un objeto que puede ser usado para descodificar una frase.

Transformada Discreta del Coseno

La TDC es usada en el método de compresión de imagen GUEF. La TDC es similar a la Transformada de Fourier, en ella se transforma un conjunto de datos del dominio de espacio al dominio de frecuencia y de regreso. Una vez que una imagen fotográfica es transformada por el TDC a un conjunto de información de frecuencia, puede ser comprimida efectivamente usando técnicas irreversibles. Para expandir la misma imagen se tiene que convertir la información de frecuencia de nuevo en información de espacio.

Verificado de la Redundancia Cíclica (CRC)

Un CRC es un número generado al aplicar una fórmula a un bloque de datos, generalmente para usarse como verificador de suma. Una buena fórmula CRC generaría un número diferente para tantas condiciones distintas de error como sea posible. La fórmula CRC referida en éste trabajo es la comúnmente usada fórmula específica CCITT de 32 bits.

Welch, Terry

Terry Welch tomó el algoritmo de compresión de datos LZ78 y lo refinó como una "Técnica de Compresión de Datos de Alto Desempeño". Su patente sobre el algoritmo LZW es ahora controlada por Unisys.

Ziv y Lempel

Jacob Ziv y Abraham Lempel son dos israelitas teoristas de la información quienes publicaron la documentación base sobre la compresión basada en diccionario en 1977 y 1978.