Análise comparativa de métodos de compactação de dados sem perda

Gabriel C. da Silva
Paulo E. D. Pinto
UERJ, Universidade Estadual do Rio de Janeiro, RJ, Brasil.

Abstract

A compactação de dados é um conjunto de métodos dos quais alguns são bastante antigos, como é o caso do método de Huffman. Nas décadas de 80 e 90 novos métodos foram desenvolvidos, alavancados pelo grande "boom" da Internet. Os métodos seguiram duas grandes linhas: compactação sem perda e compactação com perda.

Neste trabalho são enfocados os principais métodos de compactação sem perda. São descritos cinco desses métodos, é discutida uma implementação para eles e é feita a comparação entre os mesmos, considerando os fatores de taxa de compactação, tempo de execução e memória utilizada.

No final do artigo há uma breve discussão das situações favoráveis de uso de cada um deles.

1 Introdução

A compactação de dados é uma área de interesse permanente na computação e, embora já tenha tido bons resultados desde a década de 1950, continua a ter uma grande dinâmica, dado o surgimento de novos novos enfoques a problemas antigos e a novos problemas surgidos, especialmente no contexto da Internet, que lida com grandes volumes de dados e onde a compactação é um imperativo econômico [2, 5].

Os dois principais motivos econômicos para a compactação de dados são a busca de economia na transmissão de dados e no armazenamento. Na transmissão, é claro que dados menores são transmitidos mais rapidamente. No armazenamento, dados menores gastam menos memória de disco e também podem ser recuperados de memória secundária mais rapidamente.

Há inúmeros métodos de compactação, divididos em dois grandes grupos: compactação sem perda e com perda. A compactação sem perda, como o nome indica, é uma forma de compactar tal que, na descompactação, tenha-se a reprodução exata do original compactado. Ela é utilizada especialmente para textos, programas e alguns tipos de imagem.

A compactação com perda é aquela onde, na descompactação, tem-se alguma perda em relação ao original. É utilizada, em geral, nas aplicações multi-mídia. Esse enfoque é adequado quando não há muito problema em se perder alguns detalhes mínimos de uma imagem ou de uma gravação musical. É evidente que isso é feito para se baratear o armazenamento ou a transmissão ou ainda a velocidade de reprodução de algum elemento multi-mídia.

Os métodos de cada linha diferem bastante entre sí. Neste trabalho serão apenas considerados alguns métodos de compactação sem perda, e todos os exemplos mostrados se referem a textos.

A organização do artigo é a seguinte: No seção 2 descrevemos e ilustramos o funcionamento de oito dos principais métodos de compactação sem perda. Na terceira seção descrevemos um experimento de comparação de cinco desses métodos. Na última seção apresentamos um resumo das comparações feitas.

2 Métodos de Compactação sem perda

O objetivo desta seção é apresentar, de maneira suscinta, os métodos de compressão de dados *sem perda*, ressaltando as idéias básicas que os norteiam e exemplificando-os. Isto possibilitará uma primeira visão comparativa entre os métodos.

Há inúmeros livros onde mais detalhes de todos os métodos podem ser encontrados. Este resumo está fortemente baseado em três deles [3, 4, 5].

Compactação de dados é o processo de converter um conjunto de dados codificados em outro que tenha menor tamanho, o que só é possível devido a características gerais de redundância nos dados. Talvez o processo mais intuitivo e primário seja aquele que identifica repetições de conjuntos de codificações e substitui estas repetições por um

número (as repetições), seguido do trecho que se repete. Este método é denominado **Run Length Encoding** e será deixado de fora do resumo feito.

Uma possível classificação dos principais métodos de compatação sem perda é a seguinte:

a) Métodos estatísticos: *Shannon-Fano, Huffman*
b) Métodos Aritméticos: *Codificação Aritmética*
c) Métodos de Dicionário: *LZ77, LZ78, LZW*
d) Métodos de Contexto: *PPM, BWT*

Será dada, a seguir, uma breve explanação de cada um desses métodos.

2.1 Métodos estatísticos

Neste conjunto de métodos, a compressão é feita símbolo a símbolo ou palavra a palavra, quando se trata de texto, e a idéia é a de se trabalhar com codificações de tamanho variável, de tal forma que o tamanho da codificação de um símbolo varie de forma inversa à sua frequência. Desta forma, símbolos muito frequentes têm codificações pequenas e símbolos pouco frequentes, codificações grandes. É preciso, portanto, ter-se uma estatística sobre a ocorrência de símbolos para, a partir dela, gerar as codificações.

2.1.1 Método de Shanon-Fano

O *método de Shannon-Fano* será descrito apenas por razões históricas, visto ter sido um dos primeiros métodos de compactação. Ele cria codificações de forma recursiva, a partir da ordenação não crescente das frequências.

Particiona-se, sucessivamente, as frequências em dois subconjuntos cuja soma de frequências seja aproximadamente igual. Em cada nível da recursão acrescenta-se um bit diferente a cada conjunto particionado. Ao primeiro subconjunto atribui-se bit 1 e, ao segundo, 0. Suponhamos que tenhamos o seguinte conjunto de símbolos, com suas frequências respectivas:

$\{(a, 40), (b, 16), (c, 15), (d, 15), (e, 14)\}$.

A Tabela 1 ilustra o método aplicado a esse conjunto:

Símbolo	Frequência	Codificação
a	40	11
b	16	10
c	15	01
d	15	001
e	14	000

Tabela 1. Exemplo do método Shannon-Fano

O conjunto inicial foi particionado em dois: {(a, 40), (b, 16)} e {(c, 15), (d, 15), (e, 14)}. Todas as codificações relativas ao primeiro grupo começarão com o bit 1; os do segundo, com o bit 0. Em seguida, o primeiro conjunto foi novamente particionado em {(a, 40)} e {(b, 16)}, terminando as subdivisões. O símbolo a passa a ser codificado como 11 e b, como 10. O segundo subconjunto inicial é então particionado em {(c, 15)} e {(d, 15), (e, 14)}. c recebe código 01. O subconjunto {(d, 15), (e, 14)} é finalmente particionado, recebendo d o código 001 e e, o código 000.

2.1.2 Método de Huffman

O *método de Huffman* cria codificações de forma gulosa, a partir da ordenação não decrescente das frequências, construindo uma árvore estritamente binária (Árvore de Huffman), base para a codificação e decodificação.

Essa árvore é criada de forma 'bottom-up'. Inicialmente, criam-se folhas, uma para cada símbolo e executam-se passos sucessivos onde, em cada passo, são agregadas duas subárvores que tenham peso total mínimo (soma das frequências de suas folhas). As codificações são definidas a partir da árvore criada, considerando os caminhos da raiz para cada folha (atribuindo bit 1 quando se usa aresta da direita e 0, quando se usa aresta da esquerda).

Prova-se que este é o método ótimo de codificação desta classe (aquele que permite criar códigos com o menor tamanho médio ponderado nas codificações). Para exemplificar, usaremos o mesmo conjunto de dados usado no exemplo do método anterior:

$\{(a, 40), (b, 16), (c, 15), (d, 15), (e, 14)\}$.

A Figura 1 ilustra a árvore de Huffman criada.

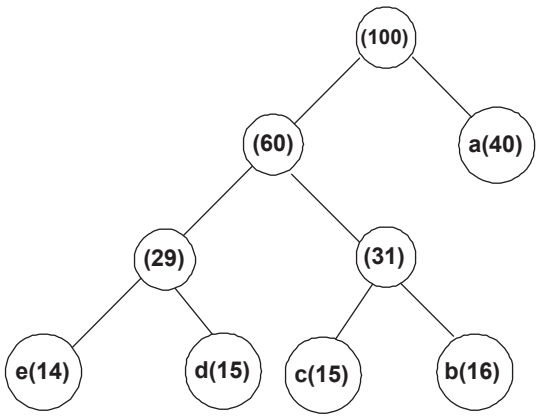

Figura 1. Árvore de Huffman do exemplo

Essa árvore é criada da seguinte maneira: inicialmente, criam-se as folhas, correspondentes aos símbolos a, b, c, d, e. As folhas de menor frequência, $e(14)$ e $d(15)$,

são unidas, obtendo-se um nó intermediário n_1, cuja frequência é 29, correspondendo à soma das frequências daquelas folhas.

Nessa altura, as subárvores existentes correspondem a $\{(a, 40), (n_1, 29), (b, 16), (c, 15)\}$. Agora, as folhas b e c são unidas, pois são as de menor frequência, obtendo-se o nó interno n_2, com frequência $16 + 15 = 31$.

Ficamos com as subárvores: $\{(a, 40), (n_2, 31), (n_1, 29)\}$. A seguir, as subárvores de raízes n_1 e n_2 são unidas, obtendo o nó n_3 com frequência $29 + 31 = 60$.

No último passo, a subárvore n_3 e a folha a são unidas, obtendo a árvore mostrada.

A partir dessa árvore, obtem-se as codificações para os diversos símbolos, conforme mostra a Tabela 2.

Símbolo	Frequência	Codificação
a	40	1
b	16	011
c	15	010
d	15	001
e	14	000

Tabela 2. Exemplo do método de Huffman

Pode-se ver que o código de Huffman gerado é melhor que o de Shanon-Fano pois o primeiro obtém um tamanho médio de codificação de 2.2 ($\sum p_i t_i$) onde p_i é a probabilidade do símbolo i e t_i o tamanho de sua codificação. No caso, o cálculo fornece: $0,4 * 1 + 0,16 * 3 + 0,15 * 3 + 0,15 * 3 + 0,14 * 3 = 2,2$. Já para o método de Shannon-Fano, esse tamanho médio é 2,29. O método de Shanon-Fano nunca gera um código melhor que o de Huffman, razão pela qual perdeu a importância.

A decodificação utiliza a mesma árvore gerada para a codificação. A sequência $aadcb$ seria codificada como 11001010011, trocando-se, na sequência, cada símbolo pela codificação dada pela árvore.

Para decodificar, percorre-se sucessivamente a árvore de Huffman, começando pela raiz e seguindo os bits codificados. Sempre que se chegar a uma folha, o símbolo correspondente à mesma é escrito na saída e o processo recomeça novamente pela raiz.

Um grande número de extensões e variações do algoritmo clássico de Huffman tem sido porposto ao longo dos anos.

Por exemplo, Faller [6], Gallager [7], Knuth [8] e Milidiú, Laber e Pessoa [9] trataram de métodos adaptativos para a construção de árvores de Huffman. Essas foram contribuições importantes, pois trata-se de estender um código já existente, considerando pequenas modificações de probabilidades.

O estudo de árvores de Huffman com altura mínima foi feito por Schwartz [10], enquanto que a construção de árvores tipo Huffman com restrição de tamanho das codificações foi feito por Turpin e Moffat [11], Larmore e Hirschberg [12] e Milidiú e Laber [13, 14]. Neste caso, a consideração é quanto ao aspecto prático da construção das árvores.

Por outro lado, Hamming sugeriu, em 1980 [15], a integração entre as tarefas de compressão de dados de Huffman e proteção contra erros, de Hamming. Pinto, Protti e Szwarcfiter [16, 17, 18] desenvolveram essa idéia, apresentando as árvores pares.

2.2 Métodos aritméticos

O máximo teórico de compressão a nível de símbolo, para uma entrada, segundo [3], é dado pela sua *entropia* [1]. No caso de Huffman só se atinge esse valor quando as probabilidades são potências negativas de 2.

Os *Métodos Aritméticos* podem ser vantajosos em situações onde Huffman não seja ótimo. Eles codificam toda a entrada de dados em um único número real no intervalo [0,1), garantido-se que o código é de prefixo e a decodificação sem ambiguidades.

2.2.1 Codificação Aritmética

A *Codificação Aritmética* também baseia-se na estatística de ocorrência de símbolos na entrada e normalmente funciona em dois passos.

O primeiro passo gera a estatística e, o segundo, a codificação. Na recepção, é necessário ter-se a estatística para a decodificação. O intervalo [0, 1) é particionado entre os símbolos, de acordo com a estatística feita. Suponhamos que tenhamos o seguinte conjunto de símbolos, com suas frequências respectivas:

$\{(S, 2), (E, 4), (T, 4)\}$.

A Tabela 3, mostra as faixas de probabilidade que seriam atribuídas a cada um dos símbolos. Chamemos F_{1i} e F_{2i} o início e o fim da faixa de probabilidades atribuída ao símbolo s_i.

Caracter	Frequência	Faixa (F_1, F_2)
S	2	[0 , 0,2)
E	4	[0,2 , 0,6)
T	4	[0,6 , 1)

Tabela 3. Estatística e Faixas na Codificação Aritmética

[1] defi nida por $-\sum_{i=1}^{n} \log p_i$, onde p_i ´e a probabilidade de ocorrˆencia do s´ı mbolo de ´ı ndice i nessa entrada.

O algoritmo define, a cada passo j, correspondente ao exame do símbolo j da entrada, um intervalo, $I_{1j},\ I_{2j}$. O resultado da compactação será o início do último intervalo.

O algoritmo começa com o intervalo [0, 1] e, sucessivamente, vai refinando esse intervalo, de acordo com a seguinte recorrência:

$I_{1j} = I_{1j-1} + (I_{2j-1} - I_{1j-1}) * F_{1i}$,
$I_{2j} = I_{1j-1} + (I_{2j-1} - I_{1j-1}) * F_{2i}$, onde s_i é o símbolo j da entrada.

Vejamos como o método codificaria a entrada 'TESTE', obtendo como resultado o número 0,71808. A Tabela 4 resume os intervalos obtidos a cada passo.

Caracter	F_1	F_2
T	0,6	1,0
E	0,68	0,96
S	0,68	0,736
T	0,7136	0,736
E	0,71808	0,72704

Tabela 4. Intervalos obtidos numa codificação Aritmética

O intervalo inicial é $[I_{10} = 0\ ,\ I_{20} = 1]$. Com a primeira letra, T, temos $I_{11} = 0 + (1 - 0) * 0,6 = 0,6$ e $I_{21} = 0 + (1 - 0) * 1 = 1$. Portanto o intervalo passa para $[0,6\ ,\ 1]$.

Com a próxima letra, E, os novos parâmetros são $I_{12} = 0,6 + (1 - 0,6) * 0,2 = 0,68$ e $I_{22} = 0,6 + (1 - 0,6) * 0,6 = 0,96$. O intervalo passa a ser $[0,68\ ,\ 0,96]$.

Após considerar S, obtemos $I_{13} = 0,68 + (0,96 - 0,68) * 0 = 0,68$ e $I_{23} = 0,68 + (0,96 - 0,68) * 0,2 = 0,736$. O novo intervalo é $[0,68\ ,\ 0,736]$.

A partir de T, ficamos com $I_{14} = 0,68 + (0,736 - 0,68) * 0,6 = 0,7136$ e $I_{24} = 0,68 + (0,736 - 0,68) * 1 = 0,736$. Como intervalo, $[0,7136\ ,\ 0,736]$.

Finalmente, com E, temos $I_{15} = 0,7136 + (0,736 - 0,7136) * 0,2 = 0,71808$ e $I_{25} = 0,7136 + (0,736 - 0,7136) * 0,6 = 0,72704$.

O intervalo final é $[0,71808\ ,\ 0,72704]$ e o resultado da compactação é $0,71808$. Na verdade, qualquer número desse intervalo poderia representar a mensagem codificada. Isso traz grande liberdade na implementação do método. Poder-se-ia adotar, por exemplo, o número $0,71875$ que, como número binário fracionário é representado como o inteiro 10111000, estando o 0 inicial implícito. Ou seja, pode-se trabalhar com inteiros binários. No caso, o inteiro correspondente seria 84.

Na decodificação, faz-se o processo inverso. A cada passo, trabalha-se com um número no intervalo [0, 1), e o símbolo correspondente é aquele cuja faixa de probabilidade contém o número atual. Então, o número atual é transformado, para expurgar o intervalo correspondente ao símbolo decodificado.

Na decodificação do número n usa-se a recorrência inversa à da codificação. Também mantém-se um intervalo $[I_{1i},\ I_{2i}]$ para cada passo, e a recorrência é a seguinte:

$I_{10} = 0; \quad I_{20} = 1;$

s_i = caracter i da decodificação, aquele cuja faixa de probabilidade contém o número $(n - I_{1i-1})/(I_{2i-1} - I_{1i-1})$;

$I_{1i} = I_{1i-1} + (I_{2i-1} - I_{1i-1}) * F_1(s_i);$

$I_{2i} = I_{1i-1} + (I_{2i-1} - I_{1i-1}) * F_2(s_i);$

A decodificação seria a seguinte, supondo que o número que represente a compactação seja $0,71875$:

$s_1 = T$, primeiro caracter da decodificação, correspondente a $(0,71875 - 0)/(1 - 0) = 0,71875$. $I_{11} = 0 + (1 - 0) * 0,6 = 0,6$. $I_{21} = 0 + (1 - 0) * 1 = 1$.

$s_2 = E$, segundo caracter da decodificação, correspondente a $(0,71875 - 0,6)/(1 - 0,6) = 0,296875$. $I_{12} = 0,6 + (1 - 0,6) * 0,2 = 0,68$. $I_{22} = 0,6 + (1 - 0,6) * 0,6 = 0,96$.

$s_3 = S$, terceiro caracter da decodificação, correspondente a $(0,71875 - 0,68)/(0,96 - 0,68) = 0,138393$. $I_{13} = 0,68 + (0,96 - 0,68) * 0 = 0,68$. $I_{23} = 0,68 + (0,96 - 0,68) * 0,2 = 0,736$.

$s_4 = T$, quarto caracter da decodificação, correspondente a $(0,71875 - 0,68)/(0,736 - 0,68) = 0,691964$. $I_{14} = 0,68 + (0,736 - 0,68) * 0,6 = 0,7136$. $I_{24} = 0,68 + (0,736 - 0,68) * 1 = 0,736$.

Finalmente, $s_5 = E$, quinto caracter da decodificação, correspondente a $(0,71875 - 0,7136)/(0,736 - 0,7136) = 0,229911$. $I_{15} = 0,7136 + (0,736 - 0,7136) * 0,2 = 0,71808$. $I_{25} = 0,7136 + (0,736 - 0,7136) * 0,6 = 0,72704$.

Vê-se que a decodificação foi correta. Os intervalos obtidos foram os mesmos da sequência de codificação.

A literatura ressalta que a codificação aritmética, embora mais complexa, tem vantagens sobre o método de Huffman em algumas situações. Uma delas é quando o tamanho do alfabeto é pequeno e as probabilidades muito desbalanceadas.

A Codificação Aritmética é o método recomendado pelo padrão JBIG para a compressão de imagens, proposto por várias entidades internacionais de padronização, dentre as quais a ISO, a IEC e a CCITT.

2.3 Métodos de Dicionário

Os *Métodos de Dicionário* são métodos de compactação de dados que usam códigos de tamanho fixo, residindo a

possibilidade de compressão na existência de cadeias repetidas na entrada, cuja compactação pode ser feita com uma única codificação. Estes métodos utilizam um dicionário, quase sempre adaptativo, como base da codificação e da decodificação. Como o dicionário tem que ser limitado, de alguma forma, então parte da entrada, aquela que não tiver correspondência no dicionário, pode não ser compactada. Essa linha de métodos de compactação tomou impulso a partir dos métodos LZ77 e LZ78, propostos por Abraham Lempel e Jacob Ziv, em 1977 e 1978, respectivamente, e fizeram surgir um grande número de variantes, a mais importante delas sendo o método LZW, descrito adiante.

2.3.1 LZ77

O método *LZ77*, criado por Ziv e Lempel [19], funciona com uma espécie de dicionário adaptativo, devendo ser imaginado como duas janelas deslizantes colocadas sobre a entrada, onde a primeira contém caracteres já examinados (Buffer de Busca) e a segunda, caracteres a serem examinados (Buffer de Espera).

Os tamanhos dessas janelas dependem da memória disponível, mas têm que ser os mesmos na compactação e na descompactação.

A compactação é feita transformando cadeias do Buffer de Espera em triplas $< o, t, c >$, onde o é a posição da maior cadeia do Buffer de Busca que coincide com o início da cadeia do Buffer de Espera; t é o tamanho dessa cadeia máxima e c só é necessário quando não foi encontrado nenhum casamento entre o Buffer de Busca e o início do Buffer de Espera.

O exemplo a seguir ilustra o método. Suponhamos que se queira compactar a palavra 'TESTES' e que os buffers tenham tamanho 3.

Inicialmente, o Buffer de Busca está vazio. Então as 3 primeiras letras serão codificadas como: $< 0, 0, T >$, $< 0, 0, E >$, e $< 0, 0, S >$. Neste momento, a situação dos buffers é dada pela Tabela 5.

TES	TES

Tabela 5. Codificação de TESTES, por LZ77, após o exame de 3 letras.

Então, com uma última tripla, $< 1, 3, - >$, encerra-se a compactação.

2.3.2 LZ78

Há alguns problemas no método LZ77, especialmente o fato de só se poder fazer referências a dados anteriores que ainda estejam no Buffer de Busca. Estes problemas foram resolvidos, também por Ziv e Lempel, na nova proposta que ficou conhecida como LZ78 [20].

Neste novo método, passa-se a construir, explicitamente, um dicionário adaptativo de prefixos, à medida que o texto vai sendo examinado e compactado. Como consequência, a compactação passa a ser feita em termos de duplas $< i, c >$, onde i indica o índice do dicionário que coincide com o prefixo em análise e c tem o mesmo significado que no método anterior. Cada vez que não há "casamento" entre a entrada e algum elemento do dicionário, é anexado um novo elemento ao dicionário, formado pela concatenação do prefixo encontrado com a primeira letra que não casou, além de ser dada a saída nesse novo elemento.

O método é ilustrado a seguir. Por exemplo, a compactação de 'TESTES-ESTETICOS', levaria á sequência de geração de duplas e inserção no dicionário, mostrada na tabela 6.

2.3.3 LZW

Uma importante variação dos métodos anteriormente descritos foi o *LZW*, proposto em 1984 por Terry Welch [21]. Uma das modificações básicas foi a de se inicializar o dicionário com todos os símbolos que podem ocorrer, o que elimina a necessidade da geração do segundo elemento da dupla, bastando apenas gerar o índice do dicionário, na compactação.

Aquí também, é mantido um prefixo de contexto formado pelos sucessivos símbolos lidos. A cada falta de casamento desse prefixo com o dicioário, o prefixo é anexado ao mesmo. E um novo prefixo contexto se inicia, formado pelo último símbolo lido. Essa maneira de tratar o prefixo de contexto também é diferente em relação aos métodos anteriores.

Dupla	Índice	Entrada no Dicion´ario
$< 0, T >$	1	T
$< 0, E >$	2	E
$< 0, S >$	3	S
$< 1, E >$	4	TE
$< 3, - >$	5	S-
$< 2, S >$	6	ES
$< 4, T >$	7	TET
$< 0, I >$	8	I
$< 0, C >$	9	C
$< 0, O >$	10	O
$< 3, >$	-	-

Tabela 6. Compactação de 'TESTES-ESTETICOS', por LZ78.

O exemplo a seguir ilustra o método, para a compactação da mesma entrada usada no LZ78: 'TESTES-ESTETICOS'. A tabela 7 mostra, à esquerda, a situação inicial do dicionário

Índice	Entrada no Dicion´ario	Índice	Entrada no Dicion´ario
1	C	8	TE
2	E	9	ES
3	I	10	ST
4	O	11	TES
5	S	12	S-
6	T	13	-E
7	-	14	EST
		15	TET
		16	TI
		17	IC
		18	CO
		19	OS

Tabela 7. Dicionário inicial e sua ampliação , no LZW.

e, à direita, os acréscimos ocorridos durante o processo de compactação.

O prefixo de contexto inicial é nulo. Então, quando se lê o caracter **T**, esse prefixo passa para **T**. Quando se lê o próximo caracter, **E**, o prefixo de contexto passa para **TE**. Como **TE** não está no dicionário, é acrescentado como um novo elemento, ao mesmo tempo que se dá a saída do prefixo de contexto anterior, **T**, correspondente a 6. Então o novo prefixo de contexto passa para **E**.

De forma análoga, dá-se saída para **E(2)** e **S(5)**, ao mesmo tempo que são adicionados os prefixos **ES(9)** e **ST(10)** ao dicionário. Neste momento, o prefixo de contexto será **T**. Quando se considera o próximo caracter, **E**, a entrada **TE** já está no dicionário. Então verifica-se o próximo caracter, **S**. O prefixo **TES** não está ainda no dicionário. Então esse prefixo é acrescentado e é dada a saída em **TE(8)**, passando o prefixo de contexto para **S**. E assim, sucessivamente. A sequência de saída de índices seria:

6 2 5 8 5 7 9 6 8 3 1 7 5

Nota-se que este processo tende a gerar mais elementos no dicionário que o anterior.

Este método mostrou-se bastante eficiente em termos práticos e passou a ser utilizado em inúmeras situações, substituindo totalmente os métodos LZ77 e LZ78.

Ele é a base para o formato de compressão de imagens 'Grafic Interchange Format (GIF)', de propriedade da Compuserve Information Service e bastante popular.

Outra medida da importância do método LZW é o fato de ele ter se tornado a base para o padrão V.42 da CCITT, para transmissão de dados via Modem.

2.4 Métodos de Contexto

Nesta seção consideramos os *Métodos de Contexto*. Esses métodos têm vários elementos dos métodos já descritos. Tal como os métodos de Dicionário, são métodos de compactação de dados que tiram partido da existência de propriedades (repetição, por exemplo) em cadeias na entrada.

O que diferencia os primeiros desses últimos é que, no caso dos métodos de Contexto, a identificação de repetições é feita de forma mais indireta e a codificação se faz caracter a caracter. O exemplo mais simples deste grupo de métodos, que será apenas ligeiramente exemplificado aquí, é o utilizado no padrão de codificação de imagens JPEG, onde uma sequência de dados de entrada, antes de ser codificada, é transformada pela aplicação de uma operação fixa a elementos consecutivos da mesma. A tabela 8 ilustra o método.

Antes	1	4	6	2	3	6	3	5	8	13	14
Depois	1	1	0	-6	-1	1	-5	0	1	3	-1

Tabela 8. Transformação de dados no padrão JPEG.

No exemplo mostrado, tem-se uma sequência inicial de símbolos. Num primeiro passo, é gerada uma nova sequência a partir da sequência de entrada, repetindo-se o primeiro elemento e, a partir daí, guardando-se a diferença entre o próximo elemento e o anterior somado a 2.

No caso, a maioria dos números ficou pequena e surgiram muitas repetições. A compactação tira partido da repetição dos números 0, 1 e -1, após a transformação. No padrão referido está disponível um certo número de operações, que podem ser testadas, antes da compactação, para determinar qual fornece melhor taxa de compactação.

Os dois métodos descritos a seguir são mais voltados para textos e procuram tirar partido da repetição de certos grupamentos de letras (como ar, gui ...), existentes nas linguagens naturais.

2.4.1 PPM

O método *PPM (Prediction with Partial Matching)*, desenvolvido por Cleary e Witten, em 1984 [22], constrói um dicionário adaptativo dos grupos de repetições (na prática trabalha-se com grupamentos máximos de 10 letras), que passa a ser utilizado para atribuir probabilidades variáveis aos símbolos, sendo a codificação final uma codificação aritmética adaptativa, baseada nessas probabilidades. Ou seja, este método é uma junção dos métodos de Dicionário e Aritmético descritos anteriormente.

Cada tamanho de grupamento é chamado de ordem e trabalha-se no máximo com 10 ordens. A ordem -1 corresponde a uma tabela com todos os caracteres possíveis, todos com a mesma probabilidade.

Durante a compactação, tal como no LZW, é mantido um prefixo de contexto da entrada. Cada novo símbolo é adicionado ao prefixo de contexto e então procura-se na tabela da ordem correspondente ao tamanho do contexto, se aquele prefixo existe. Se existir, é atualizado seu valor, e dada a saída correspondente. Se não existir, o novo prefixo é adicionado à ordem e é codificado um caracter de escape. O novo prefixo de contexto passa a ser o string formado retirando-se o caracter mais à esquerda do mesmo. Um novo teste é feito para este novo prefixo de contexto, agora numa ordem uma unidade menor e adota-se o mesmo procedimento descrito. O processo sempre para quando se chega à ordem 0.

O exato tratamento do caracter de escape pode variar ligeiramente, dando origem a vários métodos derivados, chamados PPMA, PPM*, PPMZ etc.

Como exemplo, vejamos a atribuição de probabilidades na compactação da entrada 'ARARAS', usando-se apenas 2 ordens (níveis) de grupamento, a ordem 0 (que considera apenas o caracter) e a ordem 1 (que considera grupos de dois caracteres).

Após o tratamento das 4 primeiras letras, ARAR, teríamos a situação mostrada na tabela 9.

	Ordem 1	Ordem 0
	$A \to R \quad \frac{2}{3}$	A $\frac{2}{4}$
	$R \to A \quad \frac{1}{3}$	R $\frac{2}{4}$
Total	$\frac{3}{3}$	$\frac{4}{4}$

Tabela 9. Probabilidades após o tratamento dos caracteres ARAR.

Quando se considera o próximo A, ele entra na compactação com probabilidade $\frac{1}{3}$, que é a probabilidade do grupo RA, na ordem 1. A tabela é então atualizada, considerando a nova repetição.

	Ordem 1	Ordem 0
	$A \to R \quad \frac{2}{4}$	A $\frac{3}{5}$
	$R \to A \quad \frac{2}{4}$	R $\frac{2}{5}$
Total	$\frac{4}{4}$	$\frac{5}{5}$

Tabela 10. Probabilidades após o tratamento dos caracteres ARARA.

O próximo a caracter a ser considerado, S, ainda não havia aparecido, não sendo encontrado em nenhuma ordem. Neste caso, o método codifica um caracter de escape, para sinalizar o fato na decodificação, atribui probabilidade $\frac{1}{6}$ a S, codifica esse caracter e atualiza as ordens, conforme mostrado na tabela 11.

2.4.2 BWT

	Ordem 1	Ordem 0
	$A \to R \quad \frac{2}{5}$	A $\frac{3}{6}$
	$R \to A \quad \frac{2}{5}$	R $\frac{2}{6}$
	$A \to S \quad \frac{1}{5}$	S $\frac{1}{6}$
Total	$\frac{5}{5}$	$\frac{6}{6}$

Tabela 11. Probabilidades após o tratamento dos caracteres ARARAS.

O último método aquí descrito, o *BWT (Burrows Wheeler Transform)*, foi desenvolvido por Wheeler, em 1983 e depois melhorado e publicado em 1994 [23]. Também é voltado para textos, e faz uma utilização mais indireta das repetições de grupos de letras, baseando-se em um curioso algoritmo, que faz duas operações sobre a entrada a ser compactada.

O método será melhor descrito com um exemplo. Suponhamos a mesma entrada anterior, 'ARARAS'. A primeira operação do método é criar uma lista de permutações circulares da entrada, conforme mostrado na tabela 12.

1	A	R	A	R	A	S
2	R	A	R	A	S	A
3	A	R	A	S	A	R
4	R	A	S	A	R	A
5	A	S	A	R	A	R
6	S	A	R	A	R	A

Tabela 12. Permutações do método BWT para 'ARARAS'.

A segunda operação sobre a entrada é mostrada na tabela 13 e consiste em ordenar as listas circulares.

Então, o que terá que ser compactado é a última coluna dessa ordenação, no caso: 'SRRAAA'.

Para que se possa restaurar a entrada inicial, tem que ser guardada, também, a posição da entrada na ordenação. No caso, 1.

De posse da coluna e da posição mencionadas, o curioso algoritmo que faz a restauração é bastante simples e de baixa complexidade.

Sua idéia é a seguinte: como S é o último elemento da ordenação dos caracteres A, A, A, R, R, S, então o primeiro elemento da entrada é A (o último elemento de SRRAAA); agora é possível saber que o segundo elemento da entrada é R, (pois tomou-se o terceiro A e R é o terceiro elemento de SRRAAA). Alternando-se o exame dos dois strings: SRRAAA e AAARRS, obtém-se toda a entrada.

A operação mostrada transforma, aparentemente, uma entrada de tamanho n em outra maior, que consiste em um string também de tamanho n, além de um número para informar a posição básica para a decodificação. Isso parece

1	A	R	A	R	A	S
2	A	R	A	S	A	R
3	A	S	A	R	A	R
4	R	A	R	A	S	A
5	R	A	S	A	R	A
6	S	A	R	A	R	A

Tabela 13. Ordenação do método BWT para 'ARARAS'.

contra-senso, à primeira vista. Mas a vantagem do método reside no fato de que o string gerado tende a ter inúmeros caracteres consecutivos repetidos, que correspondem ao primeiro elemento de grupos repetidos de 2 caracteres. Pode-se tirar partido dessas repetições para compactação, utilizando, por exemplo, o método Run Length, mencionado no início desta seção. Porém normalmente o método BWT é usado associado à heurística MTF.

3.4.2.1 A Heur´ıstica MTF ('Move to Front')

A Heurística MTF é muito simples. Consiste em manter todos os símbolos possíveis em uma lista. Cada vez que um símbolo está para ser compactado, utiliza-se sua posição na lista. Simultâneamente, ele é movido para frente na lista.

Por exemplo, usando o conjunto de caracteres ASCII (256 elementos) e codificando a sequência tttWtwttt teríamos como saída os inteiros

116, 0, 0, 88, 1, 119, 1, 0, 0 .

Se o arquivo codificado pelo algoritmo BWT possuir muitos caracteres repetidos, a aplicação do algoritmo MTF pode retornar um arquivo com muitos zeros. Portanto, teríamos uma frequência maior para esses zeros e, usando um código Huffman ou aritmético, obtém-se uma códificação em bits menor para esses zeros, obtendo uma melhor compactação.

Todo o processo de compactação, usando o método BWT, tem a seguinte sequência:

$BWT \Rightarrow MTF \Rightarrow$ método de compactação (geralmente Huffman ou código arimético)

E, para a descompactação:

método de descompactação $\Rightarrow MTF \Rightarrow BWT$.

3 Implementação e Comparação de desempenho

Nesta seção descrevemos as principais questões de implementação dos diversos métodos de compressão sem perda, e faremos uma comparação entre eles.

Para melhor ilustrar e comparar os métodos, foi desenvolvida uma ferramenta para compactação e descompactação de textos.

Foram implementados cinco dos métodos descritos na seção anterior, a saber: Huffman, Codificação Aritmética, LZW, PPM e BWT, que são os principais métodos descritos. Deixamos de lado o Shanon-Fano, LZ77 e LZ78 porque eles foram superados.

O aplicativo desenvolvido tem uma interface simples, mostrada na Figura 2.

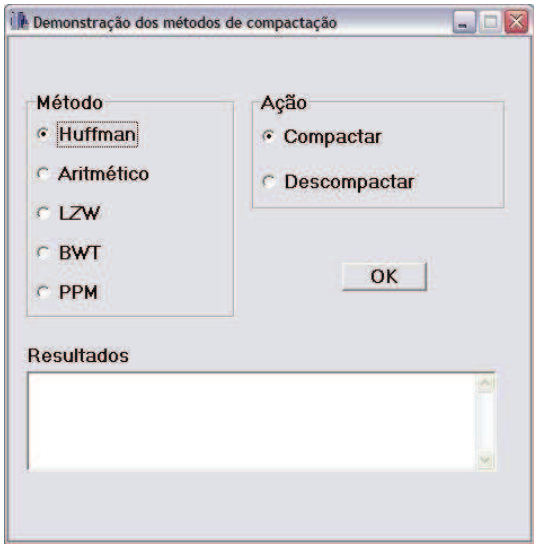

Figura 2. Interface do aplicativo de compactação

É utilizado um arquivo base na demonstração e testes da ferramenta. Este arquivo é o livro *Dom Casmurro* de Machado de Assis, no formato txt. Na ferramenta podemos escolher o método de compactação e a opção de compactar ou descompactar, como pode ser visto. A ferramenta disponibiliza um arquivo compactado e fornece também estatísticas de desempenho do algoritmo correspondente.

3.1 Questões de Implementação

A teoria dos métodos apresentados pode ser de fácil compreensão. Mas há inúmeras questões para implementação real, usando as vantagens e desvantagens de um meio computacional.

Muitas vezes é preciso fazer ligeiras modificações no método apresentado, por limitações de memória, tempo, linguagem etc. Porém sua essência deve ser sempre mantida, é claro.

A implementação dos métodos apresentados nesse projeto foi feita em C/C++. Essas linguagens são as mais uti-

lizadas nesse tipo de programação, por dar suporte a muitos recursos de baixo nível exigidos pela implementação dos métodos e serem universalmente conhecidas. A seguir são mostrados os aspectos mais importantes na implementação de cada um dos métodos.

3.1.1. Huffman

A estatística dos símbolos foi feita utilizando-se uma tabela com 256 símbolos, associando a cada um sua frequência no arquivo. Para a construção da árvore foi utilizada a biblioteca 'priority-queue' do C++, que apresenta algoritmos bastante eficientes para tratamentos de 'heaps', necessários à construção da árvore.

Os bits dos códigos gerados são gravados no arquivo destino em grupos de oito, para formar um byte e poderem ser gravados. Em quase todos os casos, sobram bits ao fim do processo. A esses bits são acrescentados zeros a fim de completar um byte. Esses bits excedentes poderiam gerar símbolos a mais na decodificação. Para evitar isso, é acrescentado ao texto o tamanho do arquivo original, para que o decodificador saiba o momento de parar o processo e evitar gravar símbolos excedentes.

A estatística também é guardada, em forma de pares, contendo símbolos e suas respectivas frequências. Esses pares são antecedidos pelo número de pares a serem guardados. Com os símbolos e suas frequências, o decodificador refaz a árvore gerada na codificação e, lendo bit a bit o arquivo codificado e percorrendo a árvore, gera o arquivo original.

3.1.2. Codificação Aritmética

O código foi implementado usando números inteiros, ao invés de números reais, para os cálculos. Por causa de erros de arredondamento, a implementação não retorna códigos precisos. Porém, tanto o codificador quanto o decodificador sempre fazem arredondamentos da mesma maneira, o que faz com que a decodificação ocorra de forma precisa.

Na próxima página são apresentados os trechos dos códigos referentes às função de compactação e descompactação de um símbolo.

Na compactação, o intervalo corrente é definido pelos inteiros low e $high$. O tamanho do intervalo, portanto, será $R = high - low + 1$. A idéia é manter esse tamanho sempre maior que $R/4$. Quando o intervalo se encontra entre $R/2$ e $3R/4$, isto é, no meio da região de codificação, nenhum bit é gravado, pois o algoritmo não sabe se será 0 ou 1. Ao invés disso, incrementa-se a variável $bits\_seguintes$ e expande-se o meio da região. Se o intervalo se encontra na metade superior da região, o próximo bit será 1, pois o intervalo agora tende a diminuir. Então, o bit 1 é enviado seguido por um número de bits 0, número esse definido pela variável $bits\_seguintes$. Depois desse processo a variável é novamente zerada.

```
void compacta_simb(int simb,int frac[] )
{long range;
     range = (long)(high-low)+1;
     high = low + ( range * frac[simb-1] ) / frac[0] - 1;
     low = low + ( range * frac[simb] ) / frac[0];
     for (;;)
        {if (high < MEIO) { output_bits_seguintes(0); }
         else if (low >= MEIO)
            {output_bits_seguintes(1);
             low -= MEIO;    high -= MEIO;
            }
         else if (low >= PRIM_QRT && high < TER_QRT)
            {bits_seguintes += 1;
             low -= PRIM_QRT;    high -= PRIM_QRT;
            }
         else break;
         low = 2 * low;    high = 2 * high+1;
        }
}

int descompacta_simb( int frac[] )
{long range;    int cum;    int simb;
     range = (long)(high-low)+1;
     cum = (int)((((long)(valor-low)+1)*frac[0]-1)/range);
     for (simb = 1; frac[simb]¿cum; simb++) ;
        {high = low + (range*frac[simb-1])/frac[0]-1;
         low = low + (range*frac[simb])/frac[0];
          for (;;)
            {if (high < MEIO) {}
             else if (low >= MEIO)
                {valor -= MEIO;
                 low -= MEIO;    high -= MEIO;
                }
             else if (low >= PRIM_QRT && high < TER_QRT)
                {valor -= PRIM_QRT;
                 low -= PRIM_QRT;    high -= PRIM_QRT;
                }
             else break;
             low = 2 * low;    high = 2 * high+1;
             valor = 2 * valor + input_bit();
        }
      retornar simb;
}
```

Caso semelhante ocorre quando o intervalo se encontra entre 0 e $R/2$, isto é, na metade inferior da região. Neste caso, o bit 0 é enviado seguido por bits 1, também usando a variável $bits\_seguintes$.

A descompactação segue exatamente o caminho inverso da compactação, garantindo a acurácia do código, mesmo usando precisão inteira.

3.1.3. LZW

Por questões de desempenho, foi definido um tamanho de 12 bits para cada símbolo, ao invés dos oito bits de um byte. Para essa tabela existirão, portanto, 4096 potenciais elementos.

Como, na verdade, os strings de comparação são formados por um código existente adicionado de um caracter, podemos guardá-los como uma única codificação acrescida de um caracter. Isso diminui o tamanho necessário para guardar esses strings, pois não é necessário manter strings de tamanho variável.

Para se fazer a busca no array, foi usada uma função hash com deslocamento de bits ("shifting") e aplicando a função ou exclusivo (xor). O prefixo e o caracter lidos são combinados para formarem um endereço no array. O código 256, por exemplo, não estará necessariamente na posição 256 do array, mas, sim, na posição baseada em um índice formado pelo próprio string. Isso reduz drasticamente o número de comparações na busca, podendo ser bem sucedida com apenas uma comparação.

Como o código de um string não é mais sabido pela sua posição no array, e para se usar o esquema de armazenamento de strings, devemos ter três arrays: um para os códigos, outro para os prefixos e outro para guardar o caracter adicionado.

No início, o algoritmo marca cada entrada na tabela de códigos com -1, indicando que aquele endereço não está sendo usado. Ao se fazer a busca, se o prefixo e o caracter são encontrados em suas respectivas tabelas no endereço hash formado pela combinação dos dois, esse endereço é retornado, indicando que esse string foi encontrado. Se o valor encontrado na tabela de códigos naquele endereço for igual a -1, o endereço é retornado indicando que o mesmo pode ser utilizado. Se o elemento no array estiver em uso, novas buscas são feitas até que se encontre o string na tabela ou haja uma entrada disponível.

Para que se garanta que a segunda busca funcione, é preciso que o tamanho da tabela seja um número primo. Isso porque se o endereço buscado e o tamanho da tabela não forem mutuamente primos, uma busca por um endereço disponível pode falhar, mesmo se houver endereços disponíveis. Com isso, o tamanho dos três arrays passa a ser um número primo imediatamente maior que o número de entradas possíveis. No nosso caso, 5021.

Durante a descompactação, estamos procurando por um código em particular. Isso significa que podemos guardar os prefixos e os caracteres adicionados em uma tabela indexada por seus próprios strings. Isso elimina a necessidade de uma função hash e de uma tabela de códigos. Um problema na descompactação é que os strings são decodificados na ordem reversa. Isso é facilmente resolvido usando um esquema de pilha para armazenar o string, gravando-o na ordem reversa.

3.1.4. BWT

Na implementação do BWT, não fazemos as $n - 1$ rotações do string (n = número de elementos do string). Se o tamanho do bloco lido fosse de 1 Kbyte, então teríamos que criar mais 1023 blocos de bytes por bloco, fazendo as rotações e ordenando por seus contextos. Ao invés disso, cada uma das rotações é representada por um índice em um array. Esse array é formado pelos bytes do bloco. Para simular as rotações esse array é tratado como uma fila circular. Para ilustrar essa idéia vejamos um exemplo:

P R O G R A M A N D O

Começando a ler a partir da primeira letra **R**, da esquerda para a direita, encontramos a primeira rotação (ROGRAMANDOP).

Usando o mesmo array, partindo da primeira letra O, encontramos a segunda rotação (OGRAMANDOPR). Fazemos esse processo sucessivamente até chegarmos na letra P inicial, encontrando a última rotação (PROGRAMANDO).

No nosso exemplo, para comparar as duas primeiras rotações no algoritmo de ordenação, basta comparar as duas primeiras letras correspondentes (R e O). Se forem iguais, basta comparar as duas letras imediatamente anteriores (P e R). E assim sucessivamente, até que se encontrem letras diferentes ou que n comparações sejam feitas, significando que as duas rotações são iguais.

Notemos que o elemento imediatamente anterior à letra P da primeira rotação seria a última letra O do array, ou seja, o último elemento do bloco.

Foi criada uma função de comparação baseada nesse esquema, passando-a como parâmetro para a função qsort da linguagem C (baseada no algoritmo quicksort). Porém, a ordenação não pode ser feita no array em sí, senão perderíamos todas as rotações. Para isso, é criado um array de índices, cujos índices são as posições dos elementos no array e cujos elementos, inicialmente, são iguais a seus índices. Esse array é ordenado de acordo com as comparações feitas com os elementos do bloco. No fim desse processo, o elemento de índice i no array de índices nos mostra a posição do elemento na posição i do bloco original no bloco transformado. Então com esse array de índices montamos o bloco transformado. O índice do primeiro elemento do bloco lido no bloco transformado será o primeiro elemento do array de índices.

Por fim, gravamos no arquivo de saída o tamanho do bloco, o bloco transformado e o índice do primeiro elemento. É necessário passar o tamanho do bloco, pois raramente teremos o tamanho do último bloco lido igual ao tamanho fixo de bloco adotado no algoritmo.

Para a decodificação não foi preciso ordenar, de fato, o bloco transformado. Montamos primeiro um array $CONT$ de tamanho 256 (ou seja, cada índice corresponde a um byte possível) mais 1, onde $CONT[i]$ é o número de bytes no bloco menores ou iguais a i. Montamos também outro array $PRED$ de tamanho igual ao número de elementos do bloco, onde $PRED[i]$ é o número de vezes em que o byte na posição i no bloco aparece nas posições anteriores a i. Assim, o bloco original é reconstruído aplicando uma função de transformação T do tipo:

```
Transforma;
      indice = indice inicial;
      Para ( i = 0 ; i < TAMANHO DO BLOCO ; i++ ):
        BLOCO_ORIGINAL[ i ] = BLOCO[ indice ]
        indice = T[ PRED[ i ] + CONT[ BLOCO[ i ] ] ]
      Fim-Para
Fim;
```

No algoritmo, *indice inicial* é o índice obtido no arquivo codificado, que é o índice do primeiro elemento no bloco original no bloco transformado.

Antes e depois da transformação aplicamos o simples algoritmo MTF. Para a compactação foi utilizado o algoritmo de codificação aritmética do ítem anterior.

3.1.5. PPM

Umas das principais características do PPM é a grande quantidade de memória utilizada para representar as tabelas de contexto e sua complexidade.

De fato, por se tratar de um modelo de várias ordens, seu algoritmo é mais complexo que os demais, com o uso de listas encadeadas e tabelas hash. Portanto, dos métodos apresentados aquí, este tende a ser mais lento que os demais, porém com expectativa de obter melhores resultados.

3.2 Comparação dos métodos

Faremos dois tipos de comparações entre os cinco métodos implementados. No primeiro tipo de comparação, os métodos são comparados entre sí e fazemos uma análise dos resultados. Em seguida, comparamos nossos resultados com resultados análogos contidos em [5].

As comparações consistem em contrastar e analisar os seguintes parâmetros de performance:

a) *Taxa de compactação* que será expressa em *bits por byte*, e indica quantos bits, em média, são necessários para representar cada caracter, após a compactação.

b) *Tempo de execução*, que é o tempo que o método gasta para a compactação e/ou descompactação.

c) *Memória*, que é a quantidade computacional de memória que cada método gasta na sua execução.

Nos dois tipos de comparação incluímos também o conhecido software de compactação GZIP, em sua versão 1.2.4. Este software é facilmente disponínel na Internet.

A inclusão do GZIP nos testes funciona como uma ponte para os dois tipos de comparação, já que elas foram feitas em ambientes distintos.

Para a comparação usamos como benchmark o Canterbury Corpus, a principal coleção de arquivos para comparação de métodos de compactação, livremente disponível na página web *www.corpus.canterbury.ac.nz*.

Os arquivos dessa coleção têm como característica o fato de que seus resultados são típicos, nos algoritmos de compactação atuais.

A tabela 14 descreve os arquivos da coleção.

Num.	Arquivo	Descriç ˜ao	Tamanho (em bytes)
1	alice29.txt	livro do L. Carrol	152.089
2	asyoulik.txt	peç a de Shakespeare	125.179
3	cp.html	p´agina HTML	24.603
4	fi elds.c	c´odigo fonte em C	11.150
5	grammar.lsp	c´odigo fonte em LISP	3.721
6	kennedy.xls	planilha do EXCEL	1.029.744
7	lcet10.txt	artigo t´ecnico	426.754
8	plrabn12.txt	poesia de Milton	481.861
9	ptt5	Documento de teste do CCITT(bitmap imagem)	513.216
10	sum	execut´avel do SPARC	38.240
11	xargs.1	manual do GNU	4.227

Tabela 14. Relação de arquivos do Canterbury Corpus.

Os testes foram feitos num computador padrão IBM-PC, com processador AMD Sempron 2400+ de 1,67Ghz, 256 MB de memória e sistema operacional Microsoft Windows XP.

A seguir apresentamos os resultados do primeiro grupo de comparações.

3.2.1. Comparações internas

Mostraremos os resultados para cada um dos parâmetros.

3.2.1.1 Taxa de Compactação

Inicialmente vejamos a taxa de compactação, expressa em bits por byte, que significa o número médio de bits usados para representar um byte no arquivo destino. Essa medida é usada aqui por ser uma forma clara e direta de comparação.

A tabela 15 mostra os resultados.

A próxima tabela, (16), mostra o valor médio da taxa de compactação e apresenta o valor também em percentual.

Num.	GZIP	Huffman	Cod. Aritm.	LZW	PPM	BWT
1	2,87	4,61	4,59	5,60	2,40	3,51
2	3,13	4,84	4,84	6,11	2,70	3,69
3	2,60	5,27	5,30	5,88	2,51	3,48
4	2,26	5,04	5,14	6,15	2,28	2,69
5	2,68	4,67	4,94	5,36	2,64	2,95
6	1,56	3,59	3,37	5,75	1,92	1,47
7	2,72	4,70	4,66	6,34	2,11	3,52
8	3,25	4,58	4,56	5,72	2,47	3,78
9	0,89	1,66	1,17	1,09	1,32	1,06
10	2,69	5,37	5,18	7,17	3,01	3,37
11	3,32	4,92	5,18	5,87	3,28	3,50

Tabela 15. Taxas de compactação (bits/byte) dos arquivos do Canterbury Corpus.

M´etodo	bits/byte (m´edio)	%
GZIP	2,07	25,92
Huffman	3,74	46,69
Cod. Arit.	3,55	44,39
LZW	5,01	62,67
PPM	2,02	25,23
BWT	3,50	29,57

Tabela 16. Taxas médias de compactação dos métodos.

O método PPM teve o melhor desempenho dentre todos, sendo o único dos métodos com o resultado geral melhor que o do GZIP.

Huffman, BWT e a Codificação Aritmética tiveram resultados intermediários.

Todos os métodos implementados tiveram seu melhor desempenho com o arquivo bitmap. Com esse arquivo, os algoritmos LZW tiveram seu melhor resultado.

Apesar de não ter tido um desempenho geral muito bom, o BWT comprimiu melhor a planilha, que é o maior arquivo da coleção. Talvez o BWT tenha melhor taxa de compactação com arquivos maiores que os listados aqui. Além disso, o BWT foi melhor que o LZW na compactação do arquivo de imagem.

Fora esses casos particulares, a codificação dos outros arquivos não trouxe grandes diferenças em relação ao resultado geral.

3.2.1.2 Tempo de execução

Outro fator importante é o tempo que o método gasta para a compactação e descompactação. Medir a velocidade de compactação dos métodos não é simples, pois depende da arquitetura do equipamento utilizado e de quão bom é o compilador. O tempo de execução pode ser influenciado, também, pelo tipo de arquivo que está sendo comprimido.

Mostraremos separadamente os tempos para a compactação e descompactação.

A Tabela 17 mostra os tempos de compactação em milisegundos. Tempos menores que 1 milisegundo foram arredondados. Já a tabela 18, mostra os tempos de descompactação.

Num.	GZIP	Huffman	Cod. Aritm.	LZW	PPM	BWT
1	78	3	4	2	11	72
2	9	2	6	1	10	56
3	6	1	2	1	3	19
4	4	1	2	1	1	13
5	3	1	1	1	1	9
6	28	12	33	11	2972	6021
7	18	4	14	3	30	1996
8	25	6	17	4	41	2002
9	11	3	9	3	24	36975
10	5	2	2	2	3	33
11	5	1	1	1	1	9
Total	**192**	**36**	**91**	**30**	**3097**	**47205**

Tabela 17. Tempos de compactação dos arquivos do Canterbury Corpus.

Num.	GZIP	Huffman	Cod. Aritm.	LZW	PPM	BWT
1	22	3	6	5	13	22
2	11	3	4	3	11	14
3	5	1	1	1	3	10
4	8	1	2	1	2	8
5	5	1	1	1	1	9
6	31	18	42	22	2970	55
7	14	9	18	11	34	26
8	22	11	19	12	45	31
9	17	8	11	3	30	24
10	9	2	3	2	3	10
11	8	1	1	1	1	8
Total	**152**	**58**	**108**	**62**	**3113**	**217**

Tabela 18. Tempos de descompactação dos arquivos do Canterbury Corpus.

A tabela 19 mostra a velocidade relativa dos métodos. Esses tempos são relativos ao tempo de compactação do GZIP. Por exemplo, um método com um tempo relativo 2, seria 2 vezes mais lento para compactar um arquivo que o GZIP. Os tempos de descompactação são normalizados tendo como base o mesmo valor. Desta forma, cada tempo de compactação pode ser diretamente comparado com o seu tempo de descompactação.

Da tabela vemos que os métodos Huffman e LZW são mais rápidos, tanto na compactação quanto na descompactação, enquanto que os mais lentos são os métodos PPM e BWT.

De fato, os métodos que possuem uma taxa de compactação mais modesta tendem a ser mais rápidos que os métodos com taxas de compactação melhores. Vimos também que a compactação de alguns arquivos da coleção afetaram sensivelmente o tempo de compactação e descompactação dos métodos PPM e BWT.

M´etodo	Compactaç ˜ao	Descompactaç ˜ao
GZIP	1	0,8
Huffman	0,2	0,3
Cod. Arit.	0,5	0,6
LZW	0,2	60,3
PPM	16,1	16,2
BWT	245,9	1,1

Tabela 19. Velocidades relativas de compactação/descompactação dos métodos.

3.2.1.3 Memória

Outro importante fator na escolha de um método de compactação é a quantidade de memória computacional que ele precisa utilizar. Exemplo desses recursos são as tabelas de frequência usadas pelos métodos estatísticos, os elementos da árvore de Huffman, a tabela utilizada pelo LZW para montar seu dicionário de símbolos, as tabelas de contexto utilizadas pelo PPM, etc.

Analisaremos cada método separadamente.

3.2.1.3.1 Huffman

Na compactação, o array de frequências possui 256 elementos de tipo inteiro, portanto 1024 bytes. Outro array, o de índices para os nós da árvore, possui também 256 elementos de tipo inteiro. Cada nó da árvore é um struct com sete campos: frequência (inteiro), tipo (byte), símbolo (byte), código (inteiro), tamanho do código (inteiro) e dois ponteiros para os nós filhos (inteiro), num total de 22 bytes.

Considerando d o número de símbolos distintos e que árvore possui geralmente $2d$ nós, são usados $44d$ bytes para os nós. Portanto, o processo de compactação consome $2kb + 44d$ bytes de memória. A única estrutura presente na descompactação é a árvore com os mesmos $44d$ bytes.

3.2.1.3.2 Codificação Aritmética

O array $caracter - indice$ possui 256 elementos de tipo inteiro e o array $indice - caracter$ possui 257 do unsigned char. Para a estatística dos símbolos foram utilizados dois arrays. Um com as frequências e o outro com as frequências acumuladas, cada um com 257 elementos de tipo inteiro. Essas estruturas são usadas tanto na compactação quanto na descompactação. No total, o processo consome 3337 bytes (um pouco mais que 3kb).

3.2.1.3.3 LZW

O algoritmo de compactação utiliza três arrays, cada um com 5021 elementos. Um deles é usado para os códigos, de tipo inteiro. Outro é usado para os prefixos, de tipo inteiro, também. O terceiro é usado para os caracteres que são concatenados, do tipo byte. Isso nos dá um total de 45189 bytes. Além desses arrays, a descompactação usa também um quarto array, para empilhar os caracteres do string decodificado, com 4000 bytes.

3.2.1.3.4 PPM

O algoritmo utiliza dois arrays com 65536 elementos do tipo context (struct de 20 bytes, usado para representar o contexto), usados pelas ordens 3 e 4. A ordem 2 utiliza um array com 65536 elementos do tipo $context\_02$ (struct de 12 bytes). A ordem 1 utiliza três arrays totalizando 65536 bytes e a ordem 0 utiliza um array de 256 bytes.

3.2.1.3.5 BWT

O tamanho do bloco é de 4096 bytes (4kb). O array de índices para as rotações possui 4096 elementos inteiros e o array com os últimos bytes de cada rotação, isto é, os bytes a serem gravados na saída, possui 4096 elementos do tipo byte. Ambas as estruturas são usadas na compactação e na descompactação. No total, são usados 20kb no processo de transformação. O algoritmo MTF utiliza um array simples de 256 elementos do tipo unsigned char (256 bytes). Para o processo de compactação, foi utilizado o mesmo algoritmo da Codificação Aritmética apresentado anteriormente.

3.2.2. Outra comparação

Em seu livro, *Managing Gigabytes*, Ian H. Witten, Alistair Moffat, and Timothy C. Bell [5] fazem uma comparação entre vários programas de compactação, usando a Canterbury Corpus como referência.

A maioria dos programas usa as mesmas técnicas aqui apresentadas, o que nos permite fazer uma comparação entre os resultados obtidos. Vale ressaltar, porém, que esses testes foram feitos numa Sun SPARCstation 5, processador 170 MHz, ou seja, em um equipamento com uma arquitetura diferente da que utilizamos.

Os programas usados por eles e suas respectivas técnicas estão descritos na Tabela 20.

M´etodo	Descriç ˜ao
Bzip2	Codifi cador Huffman de bits usando BWT
Char	Codifi cador aritmtico
Compress.	Programa de compactaç ˜ao do Unix, utiliza LZ78 e LZW
Dmc	Codifi cador aritmtico
Gzip-b	Gzip, opç ˜ao best
Gzip-f	Gzip, opç ˜ao fast
Huffword	Codifi cador Huffman de palavra
Lzrw1	Codifi cador LZ77
Pack	Codifi cador Huffman
PPM	Codifi cador aritmtico usando a tcnica PPM

Tabela 20. Métodos usados no experimento descrito no livro [5].

Para os parâmetros de tempos de compactação/descompactação e taxas de compactação, eles obtiveram o resultado mostrado na Tabela 21.

M´etodo	Velocidade Relativa		Taxa de Compactaç ˜ao	
	Compactaç ˜ao	Descompactaç ˜ao	Bits/Byte	%
Bzip2	5,5	2	2,33	27,9
Char	2,9	4	4,49	56,1
Compress.	1,0	0,6	3,31	41,4
Dmc	24,3	24,5	2,40	30,0
Gzip-b	7,0	0,3	2,53	31,6
Gzip-f	1,1	0,4	2,91	36,4
Huffword	2,2	0,9	2,95	36,9
Lzrw1	0,7	0,4	4,18	52,3
Pack	0,6	0,9	4,53	56,5
PPM	5,3	5,9	2,11	26,4

Tabela 21. Resultados do experimento descrito no livro [5].

Os resultados obtidos são semelhantes aos apresentados neste estudo. A exceção é o uso do programa compress para medir o tempo relativo.

Nessa comparação, os métodos LZ77 (gzip e lzrw1) foram os mais rápidos na descompactação, seguidos pelos codificadores LZ78 e Huffman, e os mais lentos foram os codificadores aritméticos.

No nosso teste o codificador aritmético foi mais rápido que os LZ. Isto se deve, possivelmente, pelo maior processamento da nossa arquitetura e consequente maior capacidade de cálculo. Aqui também, os métodos Huffman e LZ foram mais rápidos que os demais. Quanto às taxas de compactação, os resultados são bastante semelhantes. Nesse teste, também, os métodos foram muito bem com o arquivo de imagem. Os autores afirmam que as longas sequências de zeros no arquivo são rapidamente codificadas pelos métodos.

Nos métodos baseados em estatísticas, aos zeros são dados rapidamente uma alta probabilidade, enquanto os métodos baseados em dicionário rapidamente acumulam frases para representar as longas sequências.

4 Conclusões

A escolha entre um ou outro método de compactação pode não ser simples. Essa decisão depende muito do tipo de compactação que se quer e dos recursos disponíveis. Um método pode ser bom para imagens, porém não tão bom para textos e vice-versa.

Um método pode ser mais rápido, porém comprimir menos dados. Ou comprimir mais, ser rápido, mas requererer muita memória. Isso sem falar, é claro, do processo de descompactação, em que se pode ter um cenário bastante diferente que o da compactação.

Por exemplo, o processo de decodificação de um código de Huffman costuma ser bem mais rápido e econômico em termos de memória que o processo de codificação. Porém, no LZW os dois processos tendem a se igualar em velocidade e uso de memória, pois o processo de decodificação é praticamente o caminho reverso da codificação.

Outro fator importante a considerar, é o fato de alguns métodos serem patenteados, o que pode levar à uma opção por não utilizar métodos nessa linha.

Um dado importante, porém, é que, para aplicações estáticas, os códigos de Huffman são melhores, ao passo que, para aplicações adaptativas e online, a Codificação Aritmética funciona melhor.

Todos os métodos adaptativos requerem uma significante quantidade de memória durante a compactação e a descompactação, para guardarem as tabelas específicas para o arquivo que está sendo comprimido.

Em geral, métodos que têm melhor compactação usam tabelas mais detalhadas e mais memória para guardá-las. Métodos como o LZW utilizam dezenas de quilobytes de memória, enquanto métodos de alto desempenho baseados em símbolos utilizam milhares de quilobytes, ou mesmo megabytes.

Referências

[1] D.A. Huffman. *A method for the construction of minimum redundancies code*. In *Proceedings of IRE*, (40):1098-1101, 1951.

[2] R. Baeza e Y. Netto. *Modern Information Retrieval* Addison Wesley, 1999

[3] D. Salomon. *Data Compression The Complete Reference* Springer, 1998

[4] K. Sayood. *Introduction to Data Compression* Morgan Kaufmann Pub, 2000.

[5] I. H. Witten, A. Moffat e T. C. Bell *Managing Gygabytes Compressing and Indexing Documents and Images* Morgan Kaufmann Pub, 1999.

[6] N. Faller. An adaptative Method for Data Compression. *Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, Naval Postgraduate School, Monterrey, Ca., pp. 593-597, 1973.

[7] R. G. Gallager. Variations on a Theme by Huffman. *IEEE Transactions on Information Theory*, 24(1978), pp. 668-674.

[8] D. E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms*, 6(1985), pp. 163-180.

[9] R. L. Milidiú, E. S. Laber and A. A. Pessoa. Improved Analysis of the FGK Algorithm. *Journal of Algorithms*, Vol. 28, pp. 195-211, 1999.

[10] E. S. Schwartz. An Optimum Encoding with Minimal Longest Code and Total Number of Digits. *Information and Control*, 7(1964), pp. 37-44.

[11] A. Turpin and A. Moffat. Practical length-limited coding for large alphabeths. *Computer J.*, Vol. 38, No 5, pp. 339-347, 1995.

[12] L. L. Larmore and D. S. Hirshberg. A fast algorithm for optimal length-limited Huffman codes. *JACM*, Vol. 37 No 3, pp. 464-473, Jul. 1990.

[13] R. L. Milidiú and E. S. Laber. The Warm-up Algorithm: A Lagrangean Construction of Length Restricted Huffman Codes. *Siam Journal on Computing*, Vol. 30 No 5, pp. 1405-1426, 2000.

[14] R. L. Milidiú and E. S. Laber. Improved Bounds on the Ineficiency of Length Restricted Codes. *Algorithmica*, Vol. 31 No 4, pp. 513-529, 2001.

[15] R. W. Hamming.*C*oding And Information Theory. Prentice Hall, 1980.

[16] P. E. D. Pinto, F. Protti and J. L. Szwarcfiter. A Huffman-based Error detecting Code. In: Celso C. Ribeiro and Simone L. Martins (Eds.), *Experimental and Efficient Algorithms - Proc. of the WEA 2004*, *Lecture Notes in Computer Science*, Vol. 3059, pp. 446-457, 2004.

[17] P. E. D. Pinto, F. Protti and J. L. Szwarcfiter. Parity Codees. *Rairo Inf. Theor. Appl.* (39), pp. 263-278, 2005.

[18] P. E. D. Pinto, F. Protti and J. L. Szwarcfiter. *Exact and Experimental Algorithms for a Huffman-based error detecting code.* Submitted, 2005.

[19] J. Ziv e A. Lempel. A Universal Algorithm for Data Compression. *IEEE Transactions on Information Theory*, IT-23(3), pp. 337-343, 1977.

[20] J. Ziv e A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, IT-24(5), pp. 530-536, 1978.

[21] T.A.Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, pp. 8-19, junho 1984.

[22] J.G. Cleary e I.H.Witten. Data Compression using adaptative Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4): pp. 396-402, 1984.

[23] M. Burrows e D.J.Wheeler. A Block Sorting Data Compression Algorithm. *Technical Report SRC 124, Digital Systems Research Center*, 1994.