UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LUÍSA: UM MÉTODO DE COMPRESSÃO BASEADO EM PPM

TRABALHO DE GRADUAÇÃO

Vinícius Fülber Garcia

Santa Maria, RS, Brasil

2016

LUÍSA: UM MÉTODO DE COMPRESSÃO BASEADO EM PPM

Vinícius Fülber Garcia

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Sérgio Luís Sardi Mergen

417
Santa Maria, RS, Brasil

2016

Fülber Garcia, Vinícius

LUÍSA: Um Método de Compressão Baseado em PPM / por Vinícius Fülber Garcia. – 2016.
137 f.: il.; 30 cm.

Orientador: Sérgio Luís Sardi Mergen
Monografia (Graduação) - Universidade Federal de Santa Maria, Centro de Tecnologia, Curso de Ciência da Computação, RS, 2016.

1. TG. 2. Compressão. 3. PPM. 4. Probabilidade. I. Mergen, Sérgio Luís Sardi. II. Título.

E-mail: vfulber@inf.ufsm.com

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

LUÍSA: UM MÉTODO DE COMPRESSÃO BASEADO EM PPM

elaborado por
Vinícius Fülber Garcia

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Sérgio Luís Sardi Mergen, Dr.
(Presidente/Orientador)

Carlos Raniery Paula dos Santos, Dr. (UFSM)

Patrícia Pitthan Barcelos, Drª. (UFSM)

Santa Maria, 13 de Dezembro de 2016.

Aos meus pais que sempre me incentivaram e me proporcionaram todo o suporte necessário.

AGRADECIMENTOS

Nenhum estágio da minha vida foi concluído sozinho. Em todos os momentos contei com o apoio de muitas pessoas e sou grato a todas elas. Tenho certeza que alguns parágrafos não poderão contemplar todos aqueles que de alguma forma contribuíram seja com este trabalho, seja com minha caminhada durante a graduação como um todo. Para esses tenham a certeza que, por mais que não expresso diretamente nesse espaço, todos moram no meu coração estão presentes em meu pensamento.

Devo, antes de qualquer coisa, agradecer profundamente ao meu pai, Celso Oscar de Souza Garcia, e minha mãe, Nadir Fülber Garcia, que não só durante este trabalho, mas durante minha vida toda me apoiaram, me incentivaram e me animaram nos momentos difíceis. Vocês, hoje e sempre, são meus heróis e minhas referências.

Agradeço a minha grande companheira Luísa Perin Lucca por todo amor e carinho nesse período. Sem dúvidas termos decidido percorrer o caminho da computação juntos me tornou uma pessoa muito feliz, mais forte e mais motivada para encarar os desafios que surgiam pela frente.

Gostaria de lembrar todos os meus amigos, em especial ao Guilherme de Freitas Gaiardo e Matheus Garay Trindade pelos incontáveis trabalhos realizados juntos. Também ao Maurício Dal Pozzo Schneider pelos mais de quinze anos de amizade, amizade esta que apesar da distância continua forte como se nos encontrássemos todos os dias.

Agradeço ao grupo PET-CC que teve papel fundamental na minha formação acadêmica. A todos os integrantes do PET-CC que tive o prazer de conhecer e dividir vivências, em especial ao grupo de 2015/2016 que formou uma verdadeira família. Ao professor Giovani Rubert Librelotto que regeu o grupo para obtenção de grandes sucessos nos projetos realizados.

A todos os professores que proveram os ensinamentos necessário tanto para a realização deste trabalho quanto para a vida que seguirá o mesmo. Tenho a honra e a felicidade de dizer que tive grandes mestres durante toda minha graduação. Agradeço em especial ao professor Carlos Raniery Paula dos Santos pela sublime orientação em pesquisas dentro de uma área que ansiava conhecer.

Por fim, faço um agradecimento especial ao professor Sergio Luis Sardi Mergen que me acolheu em seu projeto de pesquisa e por dois anos me orientou com maestria e disposição. Agradeço pela possibilidade de trabalho, pela ajuda em todos os momentos, pelas produções e pelas longas discussões de ideias que sempre se mostravam muito produtivas. Agradeço pela orientação neste trabalho de graduação que me permitiu um enorme crescimento, que carrega no título uma homenagem tanto a minha companheira quanto a sua filha que compartilham o nome Luísa. Obrigado por tudo professor, te desejo muito sucesso sempre.

“A melhor maneira de prever o futuro é inventá-lo.”
— Alan Curtis Kay

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

LUÍSA: UM MÉTODO DE COMPRESSÃO BASEADO EM PPM

AUTOR: VINÍCIUS FÜLBER GARCIA
ORIENTADOR: SÉRGIO LUÍS SARDI MERGEN
Local da Defesa e Data: Santa Maria, 13 de Dezembro de 2016.

Ao longo de décadas foram propostos diversos métodos de compressão de dados, cujo objetivo é transformar símbolos de um arquivo de entrada em códigos binários que ocupem menos espaço. Um método em especial é denominado PPM (*Prediction by Partial Matching*). Este método utiliza informações de contexto para calcular a probabilidade de ocorrência de um símbolo, e usa codificação entrópica para transformar essa probabilidade em código binário. Uma das características do PPM é o acoplamento entre a busca de um símbolo dentro de um contexto e a sua codificação. Esse acoplamento provoca um engessamento que limita a forma com que a compressão é realizada. Este trabalho propõe o LUÍSA, um novo método de compressão baseado no PPM. O método inova ao separar a busca do símbolo e a sua codificação, transformando essas duas etapas em módulos independentes. Essa separação permite que diversas estratégias sejam usadas, em diferentes aspectos da compressão. O texto apresenta algumas dessas estratégias, salientando cenários em que sua aplicação seria relevante. Os experimentos expandem essa análise, demonstrando casos em que o método proposto se destaca em termos de taxa de compressão, na comparação com PPM e GZip.

Palavras-chave: TG. Compressão. PPM. Probabilidade.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

LUÍSA: A COMPRESSION METHOD BASED ON PPM

AUTHOR: VINÍCIUS FÜLBER GARCIA
ADVISOR: SÉRGIO LUÍS SARDI MERGEN
Defense Place and Date: Santa Maria, October 13th, 2016.

Throughout the decades, many compression methods were proposed to transform symbols into binary codes that occupy less space. One method in particular is called PPM (*Prediction by Partial Matching*). This method uses context information to find the probability of each symbol, and uses entropy encoding to transform the probabilities into binary code. One characteristic of PPM is the coupling between the symbol search within a context and the actual coding. This coupling leads to a tight architecture that limits the way the compression is performed. This work proposes LUISA, a novel compression method based on PPM. The method innovates in separating the symbol search and the coding, turning these two stages into independent modules. This separation allows many strategies to be used in different aspects of the compression. This text presents some of these strategies, emphasizing scenarios where their usage is relevant. The experimental evaluation expands this analyzes, showing cases where the proposed method presents better compression ration when compared to PPM and GZip.

Keywords: UW. Compression. PPM. Probability.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE APÊNDICES

LISTA DE ABREVIATURAS E SIGLAS

BMP	*Bitmap*
BWT	*Burrows Wheeler Transform*
CA	Codificação Aritmética
ISO	*International Organization for Standardization*
JSON	*JavaScript Object Notation*
KB	*Kilobyte*
LZ	*Lempel-Ziv*
MB	*Megabyte*
MTF	*Move to Front*
PPM	*Prediction by Partial Matching*
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A compressão de dados é uma área que investiga a construção, otimização e aplicação de algoritmos capazes de reduzir o número de bits necessários para a representação de dados de qualquer origem (LELEWER; HIRSCHBERG, 1987). Dada a crescente demanda de capacidade de armazenamento e transmissão de informações (HILBERT; LóPEZ, 2011), mecanismos de compressão de dados são desenvolvidos a fim de otimizar a utilização dos recursos computacionais destinados a essas tarefas.

Algoritmos de compressão de dados, além de estarem presentes em sistemas de banco de dados e em diversos protocolos de transmissão de informações, estão diretamente ligados ao cotidiano de usuários para tarefas como armazenamento de imagens, vídeos e textos em formatos comumente padronizados por normas da *International Organization for Standardization* (ISO).

Uma das formas de realizar a compressão de dados é através de métodos estatísticos, cujo objetivo é atribuir sequências de poucos bits para os símbolos mais frequentes. Um método estatístico que costuma obter ótimas taxas de compressão para arquivos de texto é chamado de PPM - *Prediction by Partial Matching* (CLEARY; WITTEN, 1984a).

O método PPM codifica um símbolo com base na probabilidade de ocorrência desse símbolo dentro do contexto atual. O contexto é o conjunto de símbolos anteriores, e seu tamanho máximo é parametrizável. Caso o símbolo a codificar nunca tenha sido encontrado antes para um contexto de tamanho n, é codificado um símbolo especial de escape, que sinaliza ao codificador a necessidade de continuar a busca no contexto de menor ordem $n - 1$. Em último caso, uma sequência de escapes é gerada, e a busca desce até a ordem mais baixa possível, onde todos os símbolos do alfabeto podem ser encontrados.

Uma árvore de contextos é utilizada para armazenar os dados estatísticos. Para cada contexto existente, da maior ordem até a menor, a árvore guarda todos os símbolos que já ocorreram, bem como a sua frequência. Essa informação é imprescindível para que se descubra a probabilidade de ocorrência de cada símbolo. Em vez de codificar o símbolo propriamente dito, o PPM codifica a sua probabilidade. Essa codificação é realizada por algum método de compressão entrópico, cuja finalidade é usar menos bits para valores de probabilidade maiores.

No PPM, a codificação entrópica de um símbolo está atrelada à busca desse símbolo na árvore de contextos. A codificação precisa ter acesso aos dados estatísticos que são guardados

na árvore de contextos, de modo que a probabilidade seja descoberta. Essa associação entre busca e codificação provoca um engessamento que dificulta a adoção de estratégias diferenciadas de compressão.

O objetivo principal deste trabalho de graduação é propor um novo método de compressão baseado no PPM, chamado LUÍSA. O método LUÍSA separa a codificação entrópica da busca do símbolo na árvore de contextos, transformando essas duas etapas em módulos independentes. Durante a busca, ao encontrar um símbolo, em vez da codificação de um valor de probabilidade, é gerada uma chave de localização do símbolo. Essa chave é enviada ao módulo de codificação entrópica, que tem por objetivo realizar a compressão propriamente dita.

Essa separação permite que diferentes estratégias sejam adotadas, tanto pelo gerador de chaves quanto pelo codificador entrópico. Por exemplo, dentro do módulo de busca, é possível criar estratégias de geração de chave que prezem pela localidade dos dados. Por outro lado, dentro do módulo de codificação, é possível conceber estratégias que se baseiem em codificação estática.

Este trabalho está estruturado da seguinte forma: o capítulo 2 apresenta o detalhamento e o estado da arte dos principais métodos de compressão existentes na literatura. O capítulo 3 traz informações sobre os trabalhos diretamente relacionados ao LUÍSA, ou seja, trabalhos que propõe adaptações do método PPM para diferentes finalidades. O capítulo 4 apresenta o método LUÍSA. Além de destacar a separação entre a busca do símbolo e a sua codificação, são apresentadas diversas possibilidades de implementação desses dois módulos, salientando cenários em que sua aplicação seria relevante. O capítulo 5 apresenta os experimentos realizados a partir de diferentes construções do método LUÍSA. Os experimentos mostram casos em que o método atinge taxas de compressões maiores do que outras abordagens, inclusive daquelas baseados em PPM. Por fim, o capítulo 6 apresenta as conclusões, onde são ressaltados os pontos fortes e os pontos fracos advindos da separação proposta nesse trabalho.

2 FUNDAMENTOS E ESTADO DA ARTE

2.1 Conceitos Fundamentais

As técnicas de compressão de dados são classificadas de forma a determinar o comportamento dos algoritmos quanto à forma de processamento e o resultado final atingido. As características mais relevantes são adaptabilidade, simetria, manipulação dos dados e perdas.

- Adaptabilidade: propriedade que determina se o algoritmo adapta-se ao arquivo que se objetiva comprimir, ou seja, se as decisões de geração de código mudam durante o processamento de acordo com probabilidades, arranjos ou características específicas encontradas.

- Simetria: essa característica é analisada a partir da complexidade da compressão e descompressão. Um algoritmo simétrico é aquele que apresenta a mesma complexidade para os dois casos.

- Manipulação dos dados: a compressão pode ser realizada a partir de blocos do arquivo original, sendo um a um comprimidos individualmente, ou a partir de um fluxo de dados, comprimindo o arquivo de forma sequencial.

- Perdas: determina se a informação, ao ser descomprimida, terá um resultado idêntico ao original. Na compressão com perdas, pode haver alteração no conteúdo ao comparar a versão original e a versão gerada pela decodificação.

Um conceito importante ligado aos dados a comprimir é a entropia. A entropia determina o grau de previsibilidade de ocorrência de um determinado símbolo em um fluxo de dados. A entropia é um conceito intimamente ligado ao grau de redundância. Quando uma informação é redundante, ela possui alta previsibilidade e baixa entropia. Quanto menor for a entropia dos dados, mais comprimíveis eles são, ou seja, menos bits são necessários para que se possa codificar a informação de forma não ambígua. Assim, outra característica importante dos métodos de compressão é a sua habilidade de comprimir dados de baixa ou alta entropia.

2.2 Breve História dos Métodos de Compressão de Dados

Fazem parte da primeira geração dos métodos de compressão de dados os algoritmos puramente estatísticos, que codificam um símbolo com base na sua frequência. Quanto maior a frequência, menor a quantidade de bits necessários para codificá-lo. Esses mecanismos são também chamados de métodos entrópicos, pois eles exploram a entropia dos dados para realizar a compressão. São exemplos de tais algoritmos a Codificação de Huffman (HUFFMAN, 1952), a Codificação de Shannon-Fano (SHANNON, 1948) e a Codificação Aritmética (WITTEN; NEAL; CLEARY, 1987).

Em um segundo momento surgem os codificadores baseados em dicionários. Esses métodos procuram os símbolos a codificar a partir de um dicionário formado pelos símbolos já processados. Os principais representantes dessa classe são os métodos da família LZ (Lempel-Ziv). Um desses métodos foi usado para definir o padrão DEFLATE (DEUTSCH, 1996) de codificação, utilizado até hoje em compressores comerciais, devido a sua capacidade de gerar boas taxas de compressão e seu baixo tempo de processamento.

Na década de 1990 foi proposto o método de Transformada de Burrows Wheeler (BWT) (BURROWS; WHEELER, 1994), que é particularmente relevante para arquivos textuais. O método é baseado na reorganização dos símbolos do texto de entrada de forma a permitir que um método de codificação entrópica faça a compressão. O trabalho iniciou como uma pesquisa acadêmica, e com o passar dos anos, evoluiu para se tornar uma ferramenta comercial.

Por fim, o método PPM é baseado na probabilidade parcial da ocorrência de um símbolo com base no contexto onde o símbolo apareceu. As variações desse método são mais recentes do que os concorrentes apresentados acima, apesar de as primeiras propostas nessa linha serem razoavelmente antigas (CLEARY; WITTEN, 1984a). Assim como o BWT, os métodos PPM são propícios para compressão de texto. No entanto, seu tempo de processamento é muitas vezes superior. Estudos mais recentes buscam alternativas que consigam reduzir esse tempo de processamento, ou pelo menos melhorar as taxas de compressão.

As próximas seções apresentam métodos que podem ser enquadrados em cada uma das categorias discutidas. Atenção especial é dada aos métodos PPM, que são o foco principal deste trabalho.

Para ilustrar o funcionamento de cada método, será usado como exemplo um texto que deve ser comprimido. Para fins didáticos, considere que o universo de símbolos possíveis é limi-

tado aos caracteres {"C", "é", "u", "_", "V"}, e que o texto a comprimir seja: **"Céu_Céu_Véu"**.

2.3 Codificação Aritmética

O método de Codificação Aritmética (CA) (WITTEN; NEAL; CLEARY, 1987) utiliza um modelo estatístico para representar os dados. Para cada símbolo possível o modelo preserva a sua frequência de ocorrência. Essa frequência, relativa à frequência de todos os símbolos somados, gera um valor de probabilidade. A partir de um símbolo a comprimir, o objetivo da codificação aritmética é codificar essa probabilidade.

Dentro da probabilidade geral, entre 0% e 100%, haverá um intervalo único que é associado a cada símbolo. Esse intervalo terá a largura da probabilidade do símbolo respectivo. A Tabela 2.1 apresenta os intervalos de probabilidade para os símbolos da sequência *Céu_Céu_Véu*. A Figura 2.1 ilustra esses intervalos em um espaço de 0% a 100% de probabilidade.

Símbolo	**Frequência**	**Probabilidade**	**Início do Intervalo (>=)**	**Final do Intervalo (<)**
u	3	0,2727	0,0000	0,2727
é	3	0,2727	0,2727	0,5454
C	2	0,1818	0,5454	0,7272
_	2	0,1818	0,7272	0,9090
V	1	0,0910	0,9090	1,0000

Tabela 2.1 – Probabilidades e intervalos

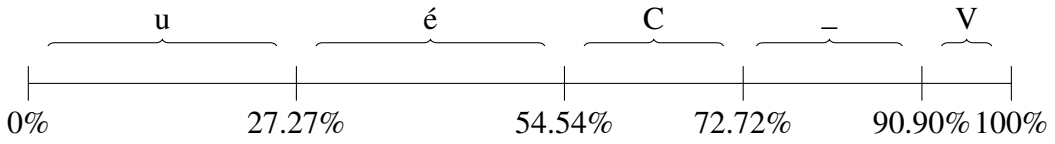

Figura 2.1 – Distribuição de frequências iniciais

A codificação de um símbolo é feita levando em consideração o seu intervalo dentro do intervalo global. O intervalo global é composto por duas extremidades, chamadas de extremidades baixa e alta. Inicialmente a extremidade baixa é 0% e a extremidade alta é 100%.

A cada símbolo processado, as extremidades são atualizadas a partir do intervalo de probabilidades do símbolo. As extremidades seguem sendo atualizadas, conforme os símbolos são processados, levando a intervalos globais cada vez menores. Finalmente, a extremidade baixa (um valor em ponto flutuante) é codificada como um valor binário.

Sendo GB e GA os valores baixo e alto do intervalo global, respectivamente, SB e SA os valores baixo e alto do intervalo do símbolo a codificar, e I o intervalo global $GA - GB$, as

funções de atualização de GA e GB são definidas pelas equações 2.1 e 2.2, respectivamente.

$$FA = FB + I \cdot SA \tag{2.1}$$

$$FB = FB + I \cdot SB \tag{2.2}$$

Um exemplo da codificação aritmética utilizando as faixas de frequência e probabilidades da Figura 2.1 é demonstrado na Tabela 2.2. Em um primeiro momento (linha 1), o intervalo global está dentro das extremidades 0% e 100%. Quando o primeiro símbolo é processado ('C'), o intervalo global fica mais restrito. O novo intervalo corresponde aos valores de probabilidades que equivalem à codificação do 'C'.

Símbolo	**Início do Intervalo**	**Final do Intervalo**
	0,00000000000000000	1,00000000000000000
C	0,54540000000000000	0,72719999999999996
é	0,59497685999999994	0,64455372000000000
u	0,59497685999999994	0,60849646972200000
_	0,60480832018983843	0,60726618523729803
C	0,60614883978672285	0,60659567965235106
é	0,60627069301807968	0,60639254624943650
u	0,60627069301807968	0,60630392239427067
_	0,60629485742044575	0,60630089852103730
V	0,60630034878088346	0,60630089852103730
é	0,60630049869502345	0,60630064860916333
u	0,60630049869502345	0,60630053957660934

Tabela 2.2 – Codificação Aritmética

A última linha abriga o intervalo que corresponde à probabilidade de ocorrência de todos os símbolos da mensagem. É esse valor de ponto flutuante que deve ser transformado em uma cadeia de bits. Sua representação em binário consome 57 bits, enquanto que a mensagem não codificada conta com 80 bits. Nota-se uma redução de 28,75% no tamanho da cadeia original.

A decodificação aritmética consiste em, a partir da sequencia de bits gerada (código), encontrar o intervalo codificado e verificar o símbolo responsável por esse intervalo dentro do intervalo global. Após a decodificação de um símbolo, o código deve ser atualizado, de acordo com a Equação 2.3, para permitir a identificação do próximo símbolo. A variável SB se refere à extremidade baixa do símbolo recém descoberto, e SP refere-se a probabilidade de ocorrência desse símbolo.

$$Código = \frac{Código - SB}{SP} \tag{2.3}$$

Um exemplo da decodificação aritmética utilizando os intervalos de frequência e probabilidades da Figura 2.1 é demonstrado na Tabela 2.3.

Código	Intervalo	Símbolo	SB	SP
0,606300498695023450	0,5454 ∼ 0,7272	C	0,5454	0,1818
0,334986241446773690	0,2727 ∼ 0,5454	é	0,2727	0,2727
0,228405725877424610	0,0000 ∼ 0,2727	u	0,0000	0,2727
0,837571418692426130	0,7272 ∼ 0,9090	_	0,7272	0,1818
0,607103513159659940	0,5454 ∼ 0,7272	C	0,5454	0,1818
0,339403262704400130	0,2727 ∼ 0,5454	é	0,2727	0,2727
0,244603090225156330	0,0000 ∼ 0,2727	u	0,0000	0,2727
0,896967694261666120	0,7272 ∼ 0,9090	_	0,7272	0,1818
0,933815700009164810	0,9090 ∼ 1,0000	V	0,9090	0,0910
0,272700000100711830	0,2727 ∼ 0,5454	é	0,2727	0,2727
0,000000000369313633	0,0000 ∼ 0,2727	u	0,0000	0,2727

Tabela 2.3 – Decodificação Aritmética

A codificação aritmética tem capacidade de aproximar a entropia do arquivo alvo, ou seja, consegue usar um número de bits por símbolo que é próximo ao menor número possível. Além disso, apresenta grande adaptabilidade para a atualização dos símbolos de forma dinâmica, ajustando as frequências e recalculando probabilidades conforme os símbolos são processados. Essas duas características tornaram o método aplicável dentro de outros métodos de compressão, como BWT e PPM.

2.4 Lempel-Ziv

A família dos métodos LZ é um marco na compressão adaptável. Esses métodos utilizam um dicionário de tamanho definido com símbolos já processados. Esse dicionário é uma janela deslizante que possui os últimos símbolos processados. Quando a janela enche, os últimos símbolos são descartados para que os mais recentes sejam inseridos.

Um conjunto de símbolos à frente do dicionário, chamados de contexto (*look ahead buffer*), são os candidatos a codificação. O contexto, assim como a janela deslizante, tem tamanho predeterminado. O objetivo é encontrar, em alguma parte do dicionário, sequências de símbolos equivalentes à sequências de símbolos que se deseja codificar.

A Figura 2.2 ilustra o dicionário e o contexto aplicados sobre a mensagem a codificar, considerando como tamanho máximo de dicionário e contexto oito e quatro, respectivamente. Os quatro primeiros símbolos já foram processados, e passam a fazer parte da janela deslizante. A partir do quinto símbolo inicia o contexto. Nesse caso em especial, os quatro símbolos do

contexto (’Céu_’) podem ser encontrados no dicionário. Assim, esses símbolos são codificados através de um código que localize essa sequência dentro do dicionário.

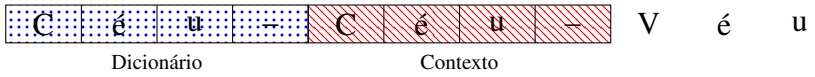

Figura 2.2 – Exemplificação de dicionário e contexto

Várias estratégias de codificação foram propostas. A primeira delas, conhecido como LZ77 (ZIV; LEMPEL, 1977), codifica uma sequência de símbolos através de triplas. Cada tripla possui o deslocamento dentro do dicionário, a quantidade de símbolos após esse deslocamento que serão aproveitados e o primeiro símbolo posterior à sequência codificada. Quanto maior a redundância presente entre dicionário e contexto, maior é o tamanho da sequência representada em cada tripla, resultando em taxas de compressão superiores.

O exemplo apresentado na Tabela 2.4 demonstra a codificação LZ77. O valor zero na primeira parte da tupla indica que o dicionário não é usado. Ou seja, não foi possível usar o dicionário (ou não compensa usá-lo) para codificar nenhuma sequência a partir do contexto. Isso ocorre para as quatro primeiras triplas. Nesses casos, apenas um símbolo é codificado (’C’, ’é’, ’u’ e ’_’). A última parte da tripla guarda esse símbolo, que é codificado sem auxílio do dicionário.

As duas últimas triplas trazem casos em que o dicionário foi usado. Por exemplo, na penúltima tripla foi codificada uma sequência de quatro símbolos, que podem ser localizados dentro do dicionário recuando quatro posições à partir do contexto (’Céu_’). Em seguida é codificado o símbolo presente na última parte da tripla (’V’).

Dicionário	**Contexto**	**Dados**	**Tripla**
#	Céu_	Céu_Véu	(0,0,C)
C	éu_C	éu_Véu	(0,0,é)
Cé	u_Cé	u_Véu	(0,0,u)
Céu	_Céu	_Véu	(0,0,_)
Céu_	Céu_	Véu	(4,4,V)
éu_Céu_V	éu		(4,1,u)

Tabela 2.4 – Codificação LZ77

A Tabela 2.5 mostra como os símbolos são decodificados a partir do processamento das triplas. Os quatro primeiros símbolos são decodificados sem auxílio do dicionário. As últimas duas triplas indicam casos em que se usa o dicionário para codificação.

Tripla	Dicionário	Decodificação
(0,0,C)		C
(0,0,é)	C	Cé
(0,0,u)	Cé	Céu
(0,0,_)	Céu	Céu_
(4,4,V)	Céu_	Céu_Céu_V
(4,1,u)	éu_Céu_V	Céu_Céu_Véu

Tabela 2.5 – Decodificação LZ77

Atualmente, os compressores baseados em métodos LZ são os mais eficientes em relação à velocidade na compressão e descompressão, atingindo taxas de compressão aceitáveis. Devido a esses fatores, foi criado o padrão DEFLATE, que é uma arquitetura de compressão baseada em dicionários deslizantes. Nesse padrão, ainda é aplicada codificação entrópica baseada em Huffman sobre os valores de deslocamento e sobre a quantidade de símbolos a codificar, o que contribui para aumentar a taxa de compressão.

2.5 Transformada de Burrows Wheeler

O método BWT (BURROWS; WHEELER, 1994) é uma técnica de compressão que divide o arquivo em blocos e comprime um bloco de cada vez. O objetivo do método é transformar os dados em um formado de menor entropia, o que permite a aplicação posterior de um método de codificação entrópica. A intuição por trás dessa ideia é que, dentro de um bloco de dados há uma grande probabilidade de um símbolo a codificar ser precedido pelo mesmo grupo de símbolos, especialmente quando o arquivo armazena texto, e isso pode ser usado para gerar saídas intermediárias de baixa entropia.

A técnica consiste em, para cada bloco de dados de tamanho n, construir uma matriz nxn, onde a primeira linha é preenchida com a sequência original de símbolos e as demais são preenchidas realocando o símbolo inicial da linha anterior para o final da sequência. Após a construção da matriz, as linhas são ordenadas em ordem lexicográfica. A informação passada para as próximas etapas é a última coluna junto ao índice que indica a linha onde está localizado o bloco original após a ordenação.

O exemplo apresentado nas Tabelas 2.6 e 2.7 demonstra a organização gerada após a aplicação da transformada em um conjunto de dados usando ordenação determinada pelo código ASCII2.

	0	1	2	3	4	5	6	7	8	9	10
0	C	é	u	_	C	é	u	_	V	é	u
1	é	u	_	C	é	u	_	V	é	u	C
2	u	_	C	é	u	_	V	é	u	C	é
3	_	C	é	u	_	V	é	u	C	é	u
4	C	é	u	_	V	é	u	C	é	u	_
5	é	u	_	V	é	u	C	é	u	_	C
6	u	_	V	é	u	C	é	u	_	C	é
7	_	V	é	u	C	é	u	_	C	é	u
8	V	é	u	C	é	u	_	C	é	u	_
9	é	u	C	é	u	_	C	é	u	_	V
10	u	C	é	u	_	C	é	u	_	V	é

Tabela 2.6 – Distribuição BWT

	0	1	2	3	4	5	6	7	8	9	10
0	C	é	u	_	C	é	u	_	V	é	u
1	C	é	u	_	V	é	u	C	é	u	_
2	V	é	u	C	é	u	_	C	é	u	_
3	_	C	é	u	_	V	é	u	C	é	u
4	_	V	é	u	C	é	u	_	C	é	u
5	u	C	é	u	_	C	é	u	_	V	é
6	u	_	C	é	u	_	V	é	u	C	é
7	u	_	V	é	u	C	é	u	_	C	é
8	é	u	C	é	u	_	C	é	u	_	V
9	é	u	_	C	é	u	_	V	é	u	C
10	é	u	_	V	é	u	C	é	u	_	C

Tabela 2.7 – Ordenação BWT

A sequência gerada é *u__uéééVCC* (símbolos presentes na última coluna) e o índice é 0 (a linha onde se encontra o texto original). O bloco gerado apresenta maior redundância que o bloco original e, dessa forma, é mais propenso a obter melhores taxas de compressão após a execução das etapas seguintes.

A próxima etapa é a aplicação da técnica chamada MTF (Move to Front) (BENTLEY et al., 1986). Essa técnica consiste na construção de um dicionário dinâmico que possui todo o universo de símbolos possíveis. Inicialmente, os símbolos podem ser posicionados em qualquer ordem. A cada ocorrência de um determinado símbolo, o mesmo é movido para a primeira posição do dicionário. Isso provoca um ajuste nas posições dos demais símbolos.

Além de mover o símbolo, a técnica emite o índice correspondente à posição anterior do símbolo. O objetivo do MTF é emitir índices de baixo valor, com predominância do valor zero. Quanto mais os índices emitidos se concentram em valores próximos de zero, menor é a entropia. Isso possibilita que um codificador entrópico seja aplicado a partir da saída do MTF.

Aplicando o MTF sobre a sequência gerada pela BWT em um dicionário inicialmente ordenado pela ordem ASCII2, obtém-se os códigos demonstrados na Tabela 2.8.

Dado	Dicionário	Código
u	{C, V, _, u, é}	3
_	{u, C, V, _, é}	3
_	{_, u, C, V, é}	0
u	{_, u, C, V, é}	1
u	{u, _, C, V, é}	0
é	{u, _, C, V, é}	4
é	{é, u, _, C, V}	0
é	{é, u, _, C, V}	0
V	{é, u, _, C, V}	4
C	{V, é, u, _, C}	4
C	{C, é, u, _, V}	0

Tabela 2.8 – Codificação MTF

A compressão propriamente dita ocorre na terceira etapa do método BWT. Nessa etapa é aplicada alguma codificação entrópica sobre a saída gerada pelo MTF (*33010400440*). Caso seja escolhida a codificação aritmética, o resultado gerado será expresso em 54 bits, o que correspondente a uma redução de 32,5% em relação a cadeia original.

O processo de decodificação, apresentado na Tabela 2.9, consiste na aplicação em ordem inversa das etapas da codificação, revertendo o método entrópico aplicado e, então, removendo a codificação MTF. Este segundo processo exige o mesmo dicionário inicial aplicado na etapa de codificação para que os resultados sejam consistentes.

Código	Dicionário	Dado
3	{C, V, _, u, é}	u
3	{u, C, V, _, é}	_
0	{_, u, C, V, é}	_
1	{_, u, C, V, é}	u
0	{u, _, C, V, é}	u
4	{u, _, C, V, é}	é
0	{é, u, _, C, V}	é
0	{é, u, _, C, V}	é
4	{é, u, _, C, V}	V
4	{V, é, u, _, C}	C
0	{C, é, u, _, V}	C

Tabela 2.9 – Decodificação MTF

Finalmente, a sequência gerada na etapa anterior deve ser ordenada lexicograficamente. Além disso, será necessário um vetor para a aplicação da fórmula de reversão da BWT. Esse vetor relaciona os índices dos símbolos da sequência ordenada e os índices da sequência original, que se deseja obter. Sendo I o índice de um símbolo da sequência original, BO o bloco refe-

rente a sequência codificada original, $BOrd$ o bloco referente a sequência codificada original ordenada, V o vetor de índices relacionando BO e $BOrd$, a fórmula de reversão FR é definida em 2.4.

$$FR = BO[V^i[I]] \tag{2.4}$$

O fragmento $V^i[I]$ indica uma aplicação recursiva de verificações no vetor V, onde i é o índice que apresenta a operação a ser realizada e a quantidade de análises recursivas a serem feitas. A aplicação da reversão na sequência gerada pela Tabela 2.7 é apresentada pelo conjunto de passos presentes na Tabela 2.11 onde se utiliza as estruturas apresentadas na Tabela 2.10.

I	0
BO	{u,_,_,u,u,é,é,é,V,C,C}
BOrd	{C,C,V,_,_,u,u,u,é,é,é}
V	{9,10,8,1,2,0,3,4,5,6,7}

Tabela 2.10 – Dados para reversão

i	**V**	**FR**
0	9	C
1	6	é
2	3	u
3	1	_
4	10	C
5	7	é
6	4	u
7	2	_
8	8	V
9	5	é
10	0	u

Tabela 2.11 – Reversão BWT

O método BWT é eficaz para a compressão de texto, principalmente quando há algum domínio específico evidente, como a presença elevada dos mesmos n-gramas, o que é comum em textos escritos em uma língua específica. A aplicação da transformada sobre esse tipo de arquivo tipicamente resulta em longas sequências do mesmo símbolo. Após a aplicação do MTF, o índice zero é gerado com uma frequência elevada, o que resulta em menores valores de entropia.

2.6 Método PPM

O PPM (CLEARY; WITTEN, 1984a) é um método de fluxo de dados que, dado um símbolo a codificar, descobre a sua probabilidade de ocorrência dentro de um contexto. Esse contexto identifica os símbolos que precedem o símbolo a ser codificado. A probabilidade encontrada é então passada a um método de codificação entrópica. Assim, quanto maior a frequência do símbolo que se busca codificar dentro do contexto, maior será a sua probabilidade,

e menor o número de bits necessários para representá-lo.

O contexto é representado por um modelo de Markov com diferentes ordens. Caso não seja possível obter uma predição para o símbolo a partir da ordem mais alta, uma busca é realizada na ordem diretamente inferior à atual. No pior caso, se o símbolo nunca foi verificado por nenhum fragmento do contexto, uma busca é feita em uma ordem especial onde todos os possíveis símbolos estão contidos. Dessa forma, é garantida a obtenção de uma predição para qualquer símbolo de entrada.

Um problema decorrente do uso do PPM é chamado de Probabilidade Zero. O problema ocorre quando deseja-se codificar um símbolo inédito, ou seja, um símbolo que aparece pela primeira vez no contexto atual. A solução para esse impasse passa pelo uso de um símbolo especial, chamado de escape. Quando um determinado símbolo não for encontrado por uma ordem, é necessário codificar uma informação indicando a passagem para uma ordem inferior. Essa informação é o escape. O símbolo de escape está presente em todas as ordens, tendo um valor de probabilidade associado. Como será visto mais adiante, o PPM admite variações na forma com que a probabilidade do símbolo de escape é calculada.

Para exemplificar, a Figura 2.3 apresenta uma mensagem a codificar. As chaves demarcam um contexto de ordem 2 e a seta demarca o símbolo a codificar. Apesar de estudos empíricos demonstrarem que o tamanho ideal de contexto, ou maior nível do modelo de Markov, está entre 3 e 5 (MOFFAT, 1990), nos exemplos que seguem serão usados contextos de ordem 2, para fins didáticos.

$$\text{C é u \_ C } \underbrace{\text{é u}}_{Contexto} \overset{\downarrow}{\text{\_}} \text{ V é u}$$

Figura 2.3 – Caso de codificação PPM

Uma árvore n-aria é comumente utilizada para o armazenamento e busca dos contextos já conhecidos. A Figura 2.4 mostra o estado da árvore depois do processamento dos sete primeiros símbolos da mensagem. Todos os contextos de ordem 0, 1 e 2 foram mapeados na árvore. O número entre parênteses determina a frequência de aparição do símbolo dentro do contexto. Por exemplo, a ordem 0 mostra que o símbolo 'é' apareceu duas vezes, independente do contexto. A ordem 1 mostra que o símbolo 'é' apareceu duas vezes depois do contexto 'C'.

O PPM admite uma ordem especial, denominada -1. Nessa ordem, todos os símbolos aparecem, mesmo aqueles que ainda não tenham ocorrido. O propósito desse nível é permitir

a codificação de símbolos inéditos em qualquer contexto. Na árvore apresentada, 'V' seria um símbolo inédito. Perceba que ele só existe no nível -1. Caso esse símbolo ocorra, são gerados escapes para cada uma das ordens, descendo até o nível -1. Como esse nível possui todos os símbolos, existe a garantia de que a codificação será bem sucedida.

Inicialmente, todas frequências nesse nível recebem o valor 1, para evitar problemas com probabilidades zeradas. Quando um símbolo novo ocorre, ele passa a existir no nível 0. Assim, nas próximas vezes que esse símbolo ocorrer, não será mais necessário descer até o nível -1. Por essa razão, as frequências desse nível nunca são atualizadas, permanecendo com o valor 1 durante todo o processamento.

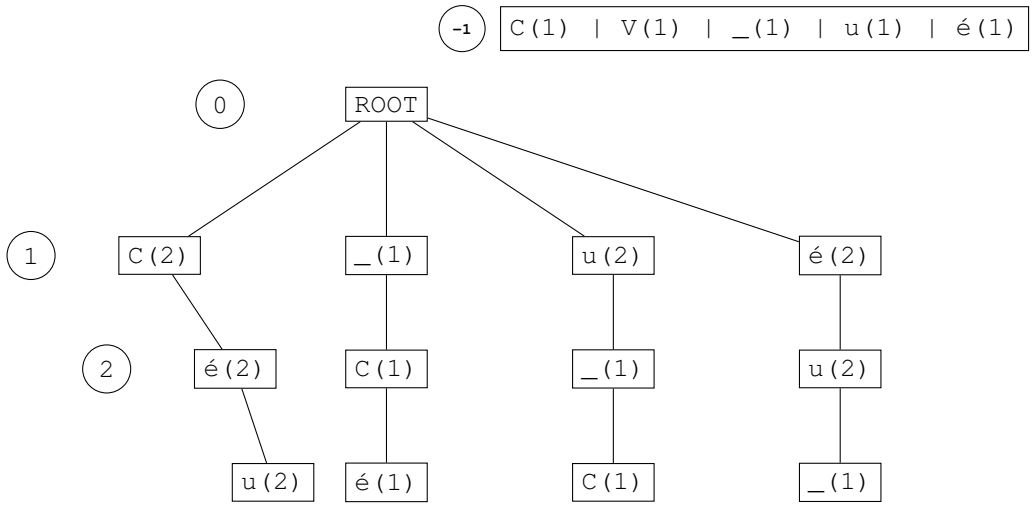

Figura 2.4 – Árvore de contextos PPM parcial

Após os sete primeiros símbolos, o próximo símbolo a codificar é o '_'. Para codificar esse símbolo, duas ações ocorrem: a codificação entrópica e a atualização da árvore. Essas duas ações são descritas a seguir:

Codificação entrópica. A codificação entrópica visa transformar a frequência do símbolo dentro do contexto em uma sequência de bits. O método mais usado para esse fim é a codificação aritmética, que é baseado na probabilidade do símbolo dentro do contexto analisado.

No caso em questão, a partir do contexto 'éu', apenas um símbolo ocorre, que é justamente o que se pretende codificar('_'). No entanto, existe um símbolo oculto que não aparece na árvore, relativo ao escape. Para que se descubra a probabilidade do '_', é preciso saber a frequência do escape. Como mencionado anteriormente, a frequência do escape depende da variante do PPM utilizada. As principais estratégias adotadas para o cálculo dessa frequência

são descritos no Capítulo 3.

Atualização da árvore: Após a codificação do símbolo, sua frequência deve ser incrementada em uma unidade em todas as ordens de contexto onde esse símbolo aparece. Para as ordens em que o símbolo não aparece, novos ramos são criados na árvore, onde o símbolo adicionado é inicializado com a frequência 1. A Figura 2.5 mostra o estado da árvore após o processamento do símbolo '_'. A frequência de três nodos foi atualizada. Esses nodos correspondem ao símbolo '_', nas três ordens mapeadas. Nesse caso, nenhum novo nodo precisou ser criado.

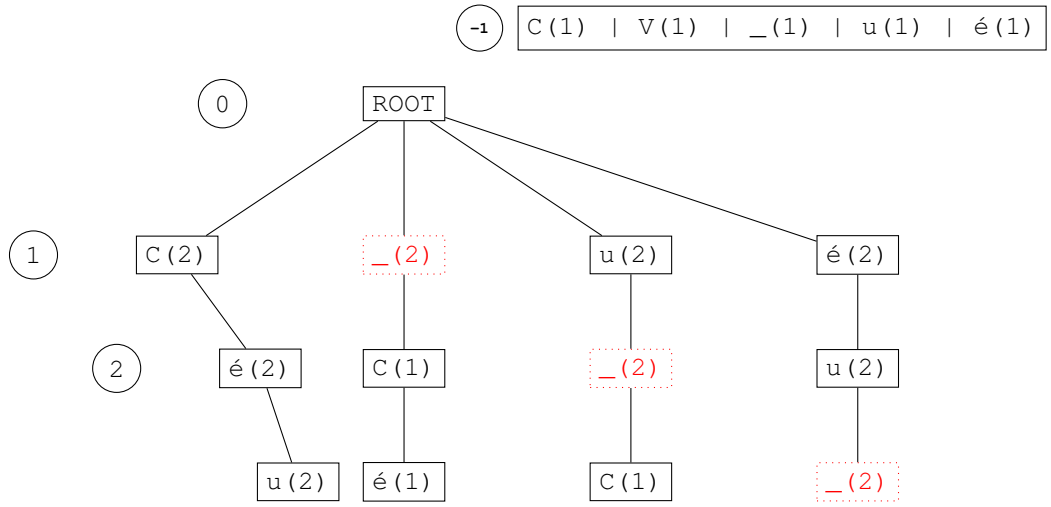

Figura 2.5 – Árvore de contextos PPM parcial atualizada

A árvore de contexto também é usada durante a decodificação. Quando um símbolo é decodificado, ele é usado na atualização da árvore, seja para a criação de novos nodos ou para o incremento da frequência de nodos já existentes. Para que a decodificação ocorra, é necessário que o decodificador aritmético tenha acesso às probabilidades de todos os símbolos possíveis. Esses valores de probabilidade podem ser recuperados a partir da própria árvore.

O método PPM é adequado para a compressão de textos em linguagem natural. Ele se destaca pela capacidade de prever, por exemplo, pequenas palavras que são usuais na escrita, além de prefixos e sufixos comuns, cujo tamanho seja menor do que o número de ordens usadas. Isso faz com que as taxas de compressão para arquivos texto tendam a ser maiores do que as taxas de outros métodos. No entanto, o tempo de processamento é maior. Isso deve-se principalmente à necessidade de realizar buscas na árvore de contexto. Nos casos em que um símbolo só exista em ordem mais baixa, o algoritmo precisa percorrer todos os nós de cada uma das ordens superiores.

3 TRABALHOS RELACIONADOS

Diversas adaptações propostas para o método PPM sugerem diferentes abordagens para a manutenção do símbolo de escape e atualização dos símbolos do dicionário. Versões que exploram esses quesitos buscam reduzir o tempo de execução e aprimorar a taxa de compressão, como por exemplo, apresentando novas estratégias para lidar com o problema da probabilidade zero (ROBERTS, 1982).

As próximas seções apresentam as principais variações propostas para o PPM. O mesmo exemplo usado no capítulo anterior é empregado afim de demonstrar como as diferentes estratégias se relacionam.

3.1 PPM-A, PPM-B e PPM-C

As primeiras três versões implementadas do método PPM mantêm como estrutura básica uma árvore de contextos n-aria contemplando todas as ordens, da mais alta à mais baixa (incluindo a -1). O tamanho máximo de contexto deve ser definido previamente.

3.1.1 PPM-A

A primeira implementação do algoritmo PPM, denominada PPM-A (CLEARY; WITTEN, 1984a), determina uma frequência padrão para o símbolo de escape. Nessa implementação, todo o escape em qualquer ponto do contexto tem frequência um ($FE = 1$), independentemente de quantas vezes o escape foi utilizado.

O cálculo da probabilidade de um símbolo qualquer PS é dado pela Equação 3.1. A variável FT refere-se frequência total, somadas as frequências de todos os símbolos ocorridos no contexto atual, excluindo o escape. Para o cálculo da probabilidade do escape, troca-se o dividendo pelo valor um.

$$PS = \frac{FS}{1 + FT} \tag{3.1}$$

Uma sumarização da codificação dos símbolos presentes na mensagem a ser comprimida é apresentada na Tabela 3.1. Para cada símbolo a codificar, estão associados a ordem em que o símbolo foi encontrado, a probabilidade de ocorrência desse símbolo (após processar todos os escapes necessários), e o número de escapes necessários. A descrição da árvore com-

pleta produzida após a codificação de toda a mensagem é apresentada no apêndice (Figura A.1).

Símbolo	**Ordem de correspondência**	**Probabilidade de correspondência**	**Escapes codificados**
C	-1	16,67%	0
é	-1	16,67%	1
u	-1	16,67%	1
_	-1	16,67%	1
C	0	20%	0
é	1	50%	0
u	2	50%	0
_	2	50%	0
V	-1	16,67%	3
é	0	20%	0
u	1	66,66%	0

Tabela 3.1 – Sumarização do processo de codificação PPM-A

O método A marginaliza a frequência do símbolo de escape. Ou seja, arquivos com uma grande variedade de contextos não são cenários apropriados para seu uso devido as múltiplas gerações do símbolo de escape. Como o escape sempre terá uma probabilidade mínima (uma vez que sua frequência será sempre um), a codificação dos escapes leva a menores taxas de compressão.

3.1.2 PPM-B

Ao contrário do método A, o método classificado como PPM-B (CLEARY; WITTEN, 1984a) incrementa a frequência do escape. Além disso, o método considera a possível existência de anomalias nos dados a serem codificados e busca, dessa forma, minimizar a influência dessas anomalias(CLEARY; WITTEN, 1984b). Para isso, um símbolo só passa a ser considerado dentro de um contexto no momento em que apresentar uma frequência mínima de dois. A frequência do escape passa a ser equivalente à soma do número de símbolos não ignorados. Ou seja, toda vez que um símbolo passa a ter sua frequência superior a um, a frequência do escape é incrementada.

O cálculo da probabilidade de um símbolo qualquer (PS) é definido na Equação 3.2. A probabilidade de um símbolo é dada pela frequência desse símbolo FS dividida pela soma entre a frequência total FT e a frequência do escape FE.

$$PS = \frac{FS}{FT + FE} \tag{3.2}$$

Uma sumarização da codificação dos símbolos presentes na mensagem a ser codificada é apresentada na Tabela 3.2. Percebe-se que, nesse exemplo, um número menor de escapes foi gerado. Isso ocorre porque existem muitos contextos que possuem apenas símbolos a serem ignorados. Nesses casos, nenhum escape é produzido e o processamento desce para a ordem imediatamente inferior.

Símbolo	**Ordem de correspondência**	**Probabilidade de correspondência**	**Escapes codificados**
C	-1	16,67%	0
é	-1	16,67%	0
u	-1	16,67%	0
_	-1	16,67%	0
C	-1	16,67%	0
é	-1	16,67%	1
u	-1	16,67%	1
_	-1	16,67%	1
V	-1	16,67%	1
é	0	12,5%	0
u	1	50%	0

Tabela 3.2 – Sumarização do processo de codificação PPM-B

Para arquivos maiores, as taxas de compressão dos métodos A e B costumam ser semelhantes. Isso ocorre porque os contextos tendem a se consolidar, e todos os símbolos passam a ter frequências maiores do que um. Além disso, a frequência do escape, mesmo incrementada, passa a representar um percentual muito baixo em comparação à frequência total de símbolos.

3.1.3 PPM-C

O método chamado PPM-C (MOFFAT, 1990) objetiva obter melhores taxas de compressão através de modificações nos mecanismos de atualização da árvore de contextos e do símbolo de escape.

De certa forma, esse método mescla características das versões anteriores. Assim como o PPM-A, basta que um símbolo ocorra uma vez para que ele seja considerado válido. Assim como o PPM-B, o símbolo do escape é incrementado toda vez que um escape tiver que ser codificado.

A principal diferença em relação aos outros métodos é referente à atualização das frequências. Quando um símbolo é codificado, em alguma ordem qualquer, apenas o nó respectivo nessa ordem tem a frequência incrementada. A frequência dos símbolos correspondentes existentes nas ordens inferiores não sofrem modificações. Essa alteração evita a supervalorização de contextos que não foram de fato preditos.

Na árvore gerada após a codificação (Figura A.2, no apêndice) é possível ver que as frequências dos nós mudam na comparação com a árvore gerada pelos métodos A e B (Figura A.1). As linhas pontilhadas destacam os nós cuja frequência mudou.

Uma sumarização da codificação dos símbolos presentes na mensagem a ser codificada é apresentada na Tabela 3.3. A ordem onde o símbolo foi encontrado e o número de escapes codificados é equivalente ao método A, uma vez que essas duas versões não ignoram nenhum símbolo já encontrado. No entanto, a probabilidade de cada símbolo muda, inclusive em relação ao método B, já que as frequências individuais são incrementadas de forma diferente.

Símbolo	**Ordem de correspondência**	**Probabilidade de correspondência**	**Escapes codificados**
C	-1	16,67%	0
é	-1	16,67%	1
u	-1	16,67%	1
_	-1	16,67%	1
C	0	12,5%	0
é	1	50%	0
u	2	50%	0
_	2	50%	0
V	-1	16,67%	3
é	0	9,09%	0
u	1	50%	0

Tabela 3.3 – Sumarização do processo de codificação PPM-C

Testes empíricos mostram que o PPM-C atinge melhores taxas de compressão do que PPM-A e PPM-B. Dessa forma, o método se tornou o estado da arte dentro da classe de compressores probabilísticos. A próxima seção apresenta algumas propostas que tiveram origem a partir de extensões do PPM-C.

3.2 Demais Métodos

Diversas variações do PPM foram propostas. Essas modificações tipicamente apresentam mudanças no tratamento da frequência dos símbolos normais e de escape, na manipulação

das ordens e na busca por mecanismos para diminuir o tempo de execução.

Uma técnica interessante que pode ser aplicada sobre qualquer variação do PPM é chamada de *Update Exclusion* (SHKARIN, 2001). Essa técnica atualiza as probabilidades de um nível quando um símbolo não for encontrado no nível superior. A atualização leva em consideração que os símbolos que existem no nível superior com certeza não serão codificados no nível abaixo. Se fossem, não teria sido necessário descer uma ordem de contexto. Assim, quando ocorre uma descida, todos os símbolos que existem no nível superior são descartados, o que leva a um aumento na probabilidade dos demais símbolos. A aplicação dessa técnica leva a um aumento na taxa de compressão. Por outro lado, o tempo de execução aumenta, uma vez que é necessário localizar todos os símbolos que deverão ser descartados.

A possibilidade da não definição prévia da maior ordem, permitindo que esse número seja ajustado durante a codificação, deu origem ao método intitulado PPM* (I.H. WITTEN; CLEARY, 1995). Este método propõe o uso de políticas de análise das sequências de entrada para determinar, dessa forma, a necessidade de expansão ou retração da quantidade de ordens utilizada.

A variação chamada PPM-D (HOWARD, 1993) faz pequenas modificações no método C, permitindo a utilização de frequências fragmentadas. Essa modificação aplica-se à atualização tanto de símbolos do dicionário quanto do símbolo de escape. Uma segunda abordagem da modificação D, o PPM-D+ (TEAHAN, 1995) utiliza múltiplos estimadores de probabilidade em busca de melhores taxas de compressão.

Alguns métodos são criados para arquivos não textuais, como o PPM-AM (ZHANG; ADJEROH, 2008), usado na compressão de imagens do tipo BMP (Bitmap). O PPM-AM busca otimizar a estrutura de contextos para dados dessa natureza, levando em consideração que contextos equivalentes são pouco comuns, enquanto contextos semelhantes são mais frequentes. O PPM-AM não utiliza escapes. Em vez disso, usa um processo de verificação de limiares que define em qual ordem os símbolos são codificados. Para a aceleração da compressão, o método recorre a uma árvore de contextos fixa, construída a partir de um conjunto de imagens consideradas representativas de todos o universo de imagens existentes.

Escalas de recência (*recency* em inglês) podem ser adicionadas ao método PPM (TEAHAN, 1995). Essas escalas tem por objetivo valorizar símbolos que apresenta alta localidade, ou seja, símbolos que estão sendo utilizados com maior frequência em determinado espaço do arquivo de entrada. A técnica é capaz de aumentar a frequência de um símbolo de acordo com

o quão recente foi a última utilização do mesmo. A dificuldade é encontrar um fator de escala adequado que maximize as probabilidades dos símbolos mais prováveis.

Outros autores exploram maneiras de otimizar o tempo de execução dos algoritmos sem afetar a sua taxa de compressão. O PPM-Z (BLOOM, 1996) utiliza estruturas de dados otimizadas para busca. Além disso, algoritmos de entropia estáticos são usados para obter uma velocidade de compressão superior. No entanto, o método utiliza muita memória para gerenciar as estruturas de dados e as tabelas de frequência estáticas. A memória cresce de forma exponencial conforme aumenta o número máximo de ordens utilizado. Dessa forma, a aplicabilidade desse método fica reduzida a casos em que poucas ordens de contexto são suficientes para se obter boas taxas de compressão.

4 PROPOSTA

Este capítulo apresenta um novo método de compressão baseado no PPM, chamado LUÍSA. Para compreender o método, é necessário primeiro analisar como o fluxo de dados no PPM é convertido para um fluxo de saída comprimido. Esse fluxo é descrito na Figura 4.1.

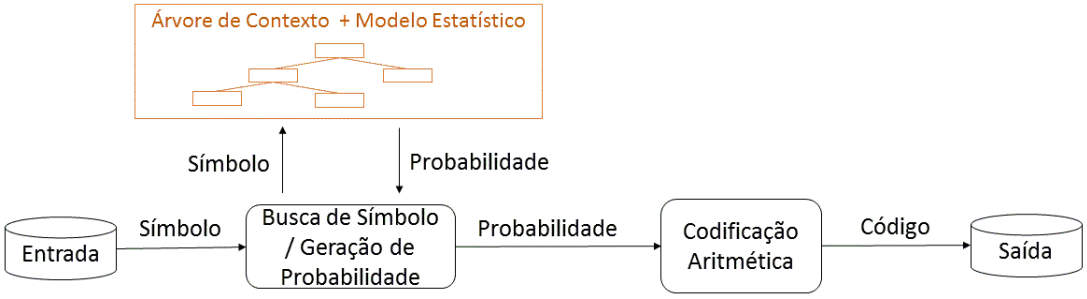

Figura 4.1 – Fluxo de dados no PPM

Para cada símbolo do fluxo de entrada são realizadas buscas na árvore de contexto. A partir do símbolo encontrado, usa-se o modelo estatístico armazenado junto à árvore para calcular a probabilidade de ocorrência desse símbolo na ordem atualmente analisada. A probabilidade é representada por um valor em ponto flutuante compreendido entre 0 e 1. Também devem ser emitidas as probabilidades de escapes para indicar ordens onde o símbolo não foi encontrado.

A etapa final fica a cargo do codificador de entropia, cuja finalidade é realizar a compressão propriamente dita. Como a identificação do símbolo é feita através de uma probabilidade, é natural que seja usada a codificação aritmética nesse passo, já que ela originalmente trabalha com probabilidades.

Apesar de a busca e codificação terem sido representadas como módulos separados, o acoplamento entre os dois é grande. Como a árvore de contextos é usada para calcular a probabilidade de um símbolo, a codificação aritmética ocorre conforme a árvore de contextos é visitada.

O método LUÍSA é uma modificação do PPM que separa o cálculo da probabilidade da árvore de contexto. A Figura 4.2 apresenta o fluxo de dados através do modelo proposto.

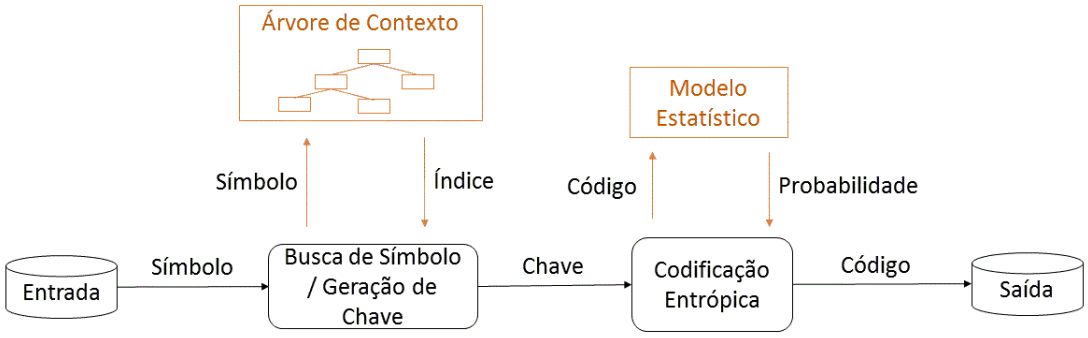

Figura 4.2 – Fluxo de dados no LUÍSA

A diferença mais marcante em relação ao PPM é a desvinculação do módulo de codificação e do módulo de busca. A busca se encarrega de localizar o símbolo na árvore de contexto e emitir uma chave que correspondam a esse símbolo. A partir dessa chave, o codificador entrópico realiza a codificação através de um modelo estatístico que possui a probabilidade dessa chave. Essa arquitetura pressupõe que a chave gerada pelo módulo de busca identifique unicamente o símbolo a codificar. Existem estratégias que podem ser seguidas para assegurar que essa chave não seja ambígua. Tais estratégias serão apresentadas no decorrer do texto.

Uma característica interessante da arquitetura é a possibilidade de usar outros codificadores entrópicos que não sejam a codificação aritmética. Além disso, a frequência de um nodo na árvore deixa de ter um papel imprescindível na codificação, o que permite que diferentes estratégias de redução de entropia sejam usadas. Cada uma dessas possibilidades será investigada nas próximas seções.

4.1 Estratégias para a Geração da Chave

No método LUÍSA, a responsabilidade do módulo de busca é gerar uma chave que localize o símbolo a codificar na árvore de contextos. Para obter maiores taxas de compressão, é importante que os símbolos pertencentes a um contexto sejam ordenados de acordo com a sua estimada probabilidade de ocorrência. Ou seja, quanto maior for a probabilidade de ocorrência de um símbolo, menor deverá ser o seu índice. Por exemplo, o símbolo na posição zero será considerado o mais provável de todos.

Se as estimativas de probabilidade forem adequadas, um número elevado de chaves de baixo valor são enviados ao codificador de entropia. De certa forma, esse fluxo se assemelha ao fluxo usado no método BWT, onde o módulo *Move to Front* é responsável por gerar códigos

numéricos de baixo valor a serem enviados a algum codificador de entropia.

Para assegurar uma compressão sem perdas, é necessário que o módulo de busca emita chaves que identifiquem de forma não ambígua os símbolo do alfabeto que se deseja codificar. Como parte do problema, é necessário sinalizar as situações em que um símbolo não aparece no contexto atual. No PPM, essa questão é tratada como o problema da Probabilidade Zero, e sua resolução envolve o uso de códigos de escape. No LUÍSA, a questão pode ser resolvida através de uma abstração que neste trabalho chamou-se de Chave Impossível. Em suma, a chave impossível é um índice que não representa nenhum dos símbolos encontrados no contexto atual.

Para compreender o problema, considere o exemplo da Figura 4.3. A Figura ilustra os símbolos existentes em duas ordens de contexto n e $n-1$. O índice acima dos símbolos marca a posição absoluta de cada símbolo naquela ordem de contexto. Essa posição representa a expectativa de quais símbolos são mais prováveis.

Os contextos de maior e menor ordem possuem respectivamente 5 e 10 símbolos. Obviamente, o contexto de menor ordem terá pelo menos os símbolos existentes no contexto de maior ordem. Essa característica leva à formação de um índice complementar, que marca a posição relativa do símbolo, descontando todos os símbolos que aparecem em ordens superiores. Na Figura 4.3, o índice abaixo dos símbolos de ordem $n-1$ marca essa posição relativa. Para a composição desse índice, todos os símbolos que já ocorreram na ordem n são ignorados.

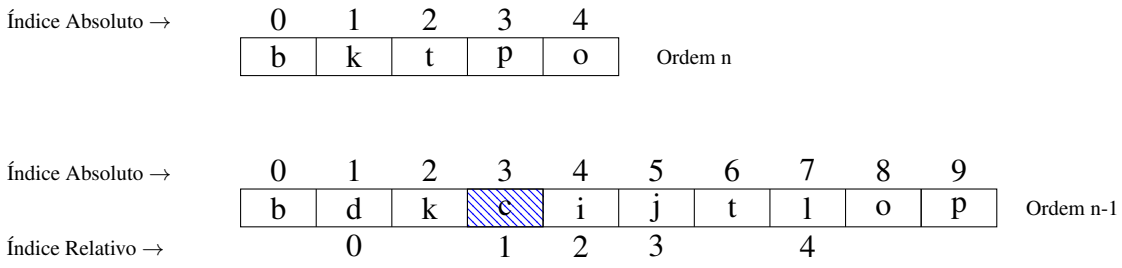

Figura 4.3 – Busca pela correspondência de posições entre ordens de contexto

Considere que o problema seja codificar o símbolo 'c', que só existe no contexto de ordem $n-1$ (marcado em azul). Diversas estratégias podem ser usadas para a geração de código não ambíguo através de chaves impossíveis. O texto a seguir descreve algumas delas.

4.1.1 Uso de Escapes

Caso um símbolo tenha sido encontrado no contexto de mais alta ordem, seu índice é emitido. Caso o símbolo tenha sido encontrado em uma ordem inferior, um código de escape

deve ser emitido. O escape marca a ordem de contexto em que o símbolo a codificar existe. Assim, esse código deve ter um valor de índice impossível dentro do contexto imediatamente superior ao contexto onde o símbolo existe. Uma possibilidade é usar o número de símbolos que existem nesse contexto.

Após o escape, deve ser emitido um índice de localização do símbolo propriamente dito. Duas estratégias podem ser usadas: sem exclusão e com exclusão. A estratégia sem exclusão usa o índice absoluto do símbolo, enquanto a estratégia com exclusão usa o índice relativo. A estratégia com exclusão funciona de forma semelhante ao *Update Exclusion* do PPM, que descarta símbolos de ordens superiores durante o cálculo da probabilidade.

As Tabelas 4.1 e 4.2 exibem as chaves que seriam gerada para cada um dos símbolos presentes na Figura 4.3. Para os símbolos que existem na ordem mais alta, nenhum escape é necessário, e um índice no intervalo de [0..4] marca o símbolo. Para os demais símbolos, o escape de valor cinco é necessário, seguido pelo índice do símbolo. O valor cinco indica que se trata de um símbolo existente na ordem $n-1$, uma vez que o maior índice possível na ordem n é quatro.

Símbolo	**Chave**
b	0
k	1
t	2
p	3
o	4
d	5,1
c	5,3
i	5,4
j	5,5
l	5,7

Tabela 4.1 – Escapes sem Exclusão

Símbolo	**Chave**
b	0
k	1
t	2
p	3
o	4
d	5,0
c	5,1
i	5,2
j	5,3
l	5,4

Tabela 4.2 – Escapes com Exclusão

Como a estratégia com exclusão desconsidera todos os símbolos que existem na ordem n, valores menores de índice são necessários para localizar um símbolo qualquer. Por exemplo, para codificar o 'c', a estratégia sem exclusão usa o valor 3 como índice, enquanto a estratégia com exclusão usa o valor 1.

A estratégia com exclusão trabalha com intervalos menores para os índices. Assim, ela tende a gerar chaves com menor entropia, o que leva a maiores taxas de compressão. No entanto, o tempo de processamento aumenta, uma vez que existe um custo extra referente ao cálculo do índice relativo.

O inconveniente dos escapes é o uso de códigos adicionais para sinalizar a necessidade

de procurar em outras ordens de contexto. Essa adição leva a codificações que ocupam mais espaço, o que torna essa estratégia pouco atrativa do ponto de vista da compressão, em ambos os casos. A próxima seção apresenta uma técnica que dispensa o uso de escapes explícitos para localizar um símbolo qualquer.

4.1.2 Uso de Acúmulo

Ao contrário da técnica anterior, o uso do acúmulo dispensa o uso de escapes. Nessa técnica, o símbolo a codificar é representado pela soma entre um índice de localização do símbolo e o número de símbolos existentes na ordem imediatamente superior.

Ao utilizar essa técnica, o código gerado necessariamente tem valor maior que a quantidade de símbolos presentes na ordem imediatamente superior e menor ou igual que a quantidade de símbolos presentes na ordem onde o símbolo de fato existe. Sendo assim, é impossível que haja uma correspondência em qualquer ordem que não seja aquela onde o símbolo será encontrado pela primeira vez.

Mais uma vez, pode-se utilizar uma estratégia com ou sem exclusão. Na estratégia sem exclusão, o índice é a posição absoluta do símbolo, enquanto a estratégia sem exclusão usa o índice relativizado.

As Tabelas 4.3 e 4.4 exibem as chaves que seriam geradas para cada um dos símbolos presentes na Figura 4.3. O intervalo de códigos [0..4] possui o índice de símbolos que existem no contexto de ordem mais alta. Valores maiores do que 4 marcam símbolos impossíveis nessa ordem, o que significa que a busca deve prosseguir no nível inferior.

Símbolo	**Chave**
b	0
k	1
t	2
p	3
o	4
d	6
c	8
i	9
j	10
l	12

Tabela 4.3 – Acúmulo sem Exclusão

Símbolo	**Chave**
b	0
k	1
t	2
p	3
o	4
d	5
c	6
i	7
j	8
l	9

Tabela 4.4 – Acúmulo com Exclusão

Para exemplificar, considere a codificação do símbolo 'c'. Nesse caso, o número de símbolos na ordem n (5) deve ser somado ao índice de localização do símbolo. Quando se usa acúmulo sem exclusão, a chave gerada é igual a 8 (5+3). Quando se usa acúmulo com exclusão,

a chave gerada tem valor 6 (5+1).

O acúmulo com exclusão tem como objetivo determinar a menor chave possível que identifique o símbolo a codificar no contexto de mais alta ordem em que ele aparece. Das quatro possibilidades aqui investigadas, essa é a estratégia que leva a menores valores de entropia, devido ao encurtamento do intervalo de índices possíveis e à ausência de escapes. A desvantagem tem relação ao custo associado à exclusão, ou seja, o custo em encontrar quantos símbolos devem ser desconsiderados durante o acúmulo, para a montagem do índice relativo.

4.2 Desacoplamento entre Identificação do Símbolo e Codificação Entrópica

No método LUÍSA, o módulo de codificação entrópica recebe chaves para codificar. As chaves podem assumir um valor de um intervalo que inicia em zero e termina em um valor máximo necessário para identificar algum símbolo qualquer. Esse valor máximo depende da estratégia usada para a geração da chave. Por exemplo, se for usado o acúmulo com exclusão, o maior valor de chave necessário é de 255.

O propósito do módulo de geração de chave é gerar informações com baixa entropia, com predominância de valores baixos. Esse modulo não tem relação alguma ao codificador entrópico que irá processar as chaves geradas. Ou seja, o método preza pelo total desacoplamento entre a identificação do símbolo e a sua codificação. Isso confere uma maior versatilidade, ao permitir que se troque o módulo de codificação sem um custo considerável de manutenção do código fonte.

Além disso, como o módulo de codificação trabalha com chaves, e não valores de probabilidade, não existe dependência com a codificação aritmética. Assim, se for conveniente, outros tipos de codificação podem ser usadas, como Huffman ou Shannon-Fano.

Outra possibilidade oriunda do desacoplamento é a de *bufferizar* as chaves antes de enviá-las ao módulo de codificação. Essa *bufferização* pode ser utilizada com o objetivo de paralelizar a execução das atividades de cada módulo, através do paradigma produtor-consumidor.

A estratégia de produtor consumidor é ilustrada na Figura 4.4. As linhas pontilhadas indicam o fluxo de codificação e as linhas tracejadas apresentam o fluxo de decodificação.

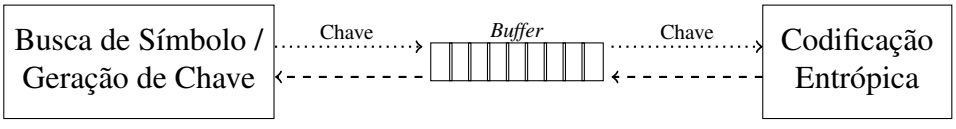

Figura 4.4 – Uso do paradigma 'produtor-consumidor' para paralelizar os módulos do LUÍSA

No processo de codificação, as chaves geradas são produzidas e inseridas no *buffer*. Por sua vez, a codificação de entropia consome as chaves realizando as operações necessárias para a geração da cadeia de bits. O processo inverso ocorre na decodificação, onde o produtor passa a ser o algoritmo de decodificação entrópica e o consumidor passa a ser o módulo de identificação do símbolo a decodificar.

4.3 Codificação Aritmética

Como visto na seção anterior, o método LUÍSA não restringe o tipo de codificador entrópico que deve ser usado para codificar as chaves geradas. Além de usar codificadores não probabilísticos, como Huffman, existe a possibilidade de usar variações da própria codificação aritmética que é usada no método PPM. O objetivo dessa seção é apresentar variações de codificação aritmética que podem ser úteis no contexto do LUÍSA.

Como apresentado no Capítulo 2, o propósito da codificação aritmética é transformar uma chave em um valor de probabilidade com base na sua frequência. Essa probabilidade por sua vez é representada como uma sequência de bits. Para exemplificar, a Figura 4.5 apresenta uma tabela de frequências para um universo de chaves reduzido ({0,1,2,3,4}). Se a chave buscada for zero, a probabilidade retornada seria 33,33%.

Chave →	0	1	2	3	4	Total
Frequências →	4	3	2	2	1	12

Figura 4.5 – Tabela de Frequências

No tocante à codificação aritmética, pode-se empregar diferentes estratégias para o cálculo da probabilidade associada a uma chave. A Figura 4.6 destaca duas possibilidades: usando uma tabela e usando múltiplas tabelas. Com o uso de uma única tabela, todos valores de chaves que chegam ao codificador são mapeados para a mesma tabela. Ou seja, a tabela guarda a frequência geral das chaves, independente de contexto.

Já com a adoção de múltiplas tabelas de frequência, quando chega uma chave, é necessário determinar qual tabela deve ser usada para a sua codificação. Essa decisão deve ser tomada com base em alguma informação de contexto recente, como por exemplo, a natureza dos últimos símbolos processados. Cabe ressaltar que, para que se tenha acesso à essa informação, seria necessário aumentar o acoplamento entre a identificação do símbolo e a sua codificação.

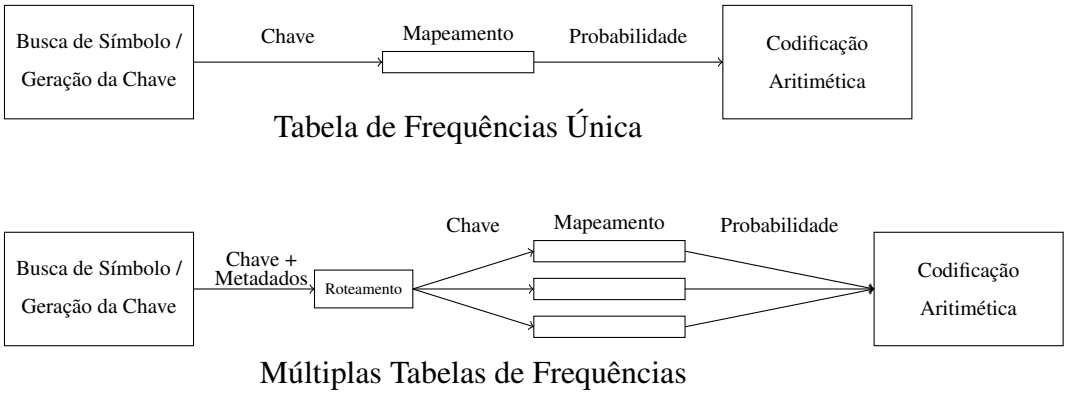

Figura 4.6 – Estratégias para tabelas de frequência

O método LUÍSA também possibilita que seja usada a codificação estática em vez da codificação dinâmica. Em outras palavras, é possível usar alguma tabela de frequência previamente montada em vez de ajustar as frequências conforme as chaves são processadas. Nessa variação, é necessário que todas as chaves geradas sejam armazenadas em um buffer. Esse *buffer* seria então consultado para a criação da tabela de frequências. Após a criação da tabela, caberia ao módulo de codificação processar todas as chaves presentes no *buffer*.

A tabela de frequências precisa ser anexada ao arquivo comprimido, para que o algoritmo de descompressão consiga encontrar as chaves com base nos valores de probabilidade que vão sendo decodificados. É essa exigência que torna o uso de tabelas estáticas inviável no PPM, uma vez que seriam necessárias tabelas para cada um dos contextos encontrados. O espaço ocupado por todas as tabelas acabaria levando a arquivos maiores do que o arquivo original.

Outra possibilidade é adotar alguma estratégia que mescle codificação estática e dinâmica. Nesse caso, o módulo de codificação inicialmente utiliza codificação dinâmica, atualizando a tabela conforme as chaves aparecem. Em determinado momento, o codificador pode decidir por interromper a alimentação da tabela, passando a usar codificação estática.

De modo geral, a vantagem da codificação estática é o menor tempo necessário para a busca do intervalo de probabilidades, usado tanto na compressão quanto na descompressão. Por outro lado, a taxa de compressão pode ser inferior, principalmente na parte inicial do arquivo, quando ainda não existe uma predominância de chaves de baixo valor.

4.4 Métodos de Atualização de Contextos

No método PPM, quando um símbolo aparece dentro de um contexto, deve-se incrementar a sua frequência. Os símbolos que mais se repetem dentro de um contexto terão frequências

maiores, levando a maiores valores de probabilidade.

Como no método LUÍSA a codificação está desacoplada da árvore de contextos, a atualização do contexto tem um propósito diferente. O objetivo é promover os símbolos mais prováveis, de modo que eles tenham índices de menor valor, ou seja, sua posição em relação aos símbolos menos prováveis será menor. Essa promoção pode se basear na frequência do símbolo, mas também pode ser feita com base em outros fatores. Essa seção é dedicada à apresentação de diferentes estratégia para conduzir essa promoção.

4.4.1 LUÍSA-F

A primeira estratégia, chamada de LUÍSA FREQUENCY (ou LUÍSA-F), usa a frequência do símbolo para determinar a sua posição. Quanto mais frequente o símbolo, menor será o seu índice.

A Figura 4.7 exemplifica a atualização das posições dos símbolos em um contexto após a ocorrência do símbolo 'k'. Como pode-se ver, o símbolo é promovido quando a sua frequência for maior que a dos seus antecessores.

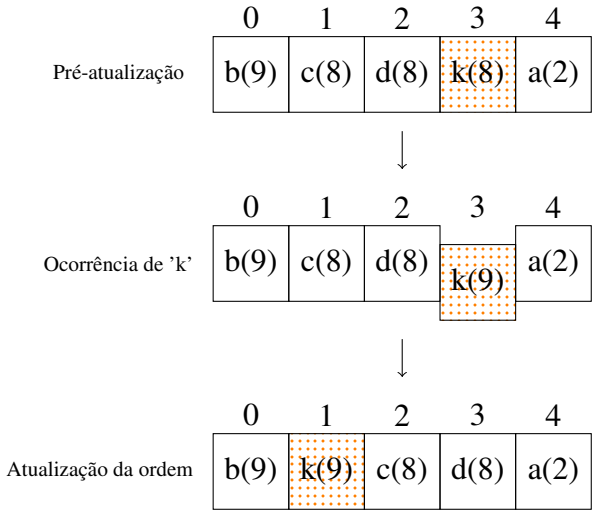

Figura 4.7 – Atualização LUÍSA-F

Esse tipo de estratégia segue o mesmo preceito que o PPM, de que símbolos cuja frequência total seja maior são mais prováveis de ocorrerem novamente. Assim como o PPM, a técnica sofre quando símbolos de elevada frequência deixam de aparecer subitamente, pois eles continuarão sendo predominantes enquanto sua frequência for maior.

4.4.2 LUÍSA-S

Mudanças súbitas nos padrões de ocorrência dos símbolos demandam estratégias que se adaptem mais rapidamente. A técnica LUÍSA SWAP (ou LUÍSA-S) foi criada com esse propósito. Essa técnica não utiliza nenhuma informação acumulativa de frequência para atualizar as posições dos símbolos. Em vez disso, quando um símbolo ocorre, ele é promovido em uma posição, independente de quantas vezes ele tenha ocorrido.

A Figura 4.8 exemplifica a atualização das posições dos símbolos em um contexto após a ocorrência do símbolo 'k'. Como pode-se ver, o símbolo trocou de lugar com o símbolo que estava à sua frente.

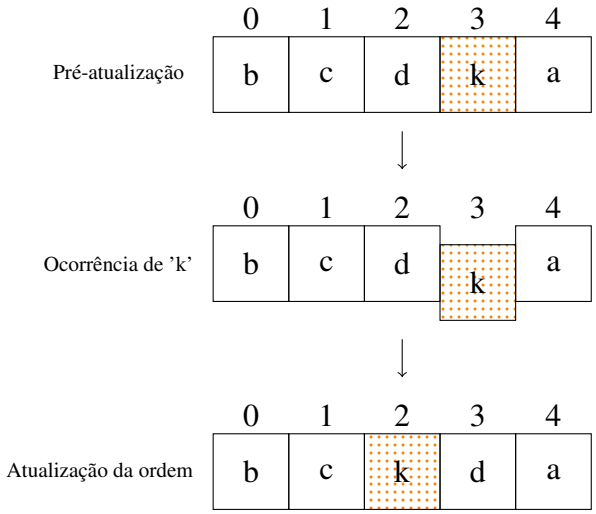

Figura 4.8 – Atualização LUÍSA-S

Além de uma maior adaptabilidade, essa técnica dispensa o uso da frequência em cada nó da árvore de contexto. Isso leva a uma redução do espaço em memória dedicado às estruturas de dados usadas durante a compressão. A desvantagem está relacionada à compressão de arquivos onde símbolos frequentes continuam ocorrendo com frequência elevada. Tais símbolos podem perder posições durante o ajuste do contexto, prejudicando a taxa de compressão.

4.4.3 LUÍSA-FS

A técnica chamada de LUÍSA FREQUENCY SWAP (ou LUÍSA-FS) mescla características de LUÍSA-F e LUÍSA-S. O objetivo é garantir que ao menos uma troca seja realizada, ao mesmo tempo em que símbolos com elevada frequência sejam privilegiados. Quando um símbolo ocorre, ele passa à frente de todos os símbolos antecessores que possuem frequência

menor à sua. Caso nenhum dos antecessores satisfaça essa condição, o símbolo é trocado de lugar com aquele que está imediatamente à sua frente.

Utilizando a técnica de troca, a mobilidade dos símbolos dentro da ordem de contexto aumenta. Dessa forma,um símbolo cuja aparição inicial tenha tardado a ocorrer, mas que siga ocorrendo com frequência elevada, será rapidamente promovido a um índice de baixo valor.

As Figuras 4.9 e 4.10 exemplificam a atualização das posições dos símbolos em um contexto após a ocorrência do símbolo 'k'. Na primeira, a troca ocorre com base na frequência do símbolo, que avança 2 posições. Na segunda, o mecanismo de troca teve que ser acionado, para garantir ao menos uma movimentação.

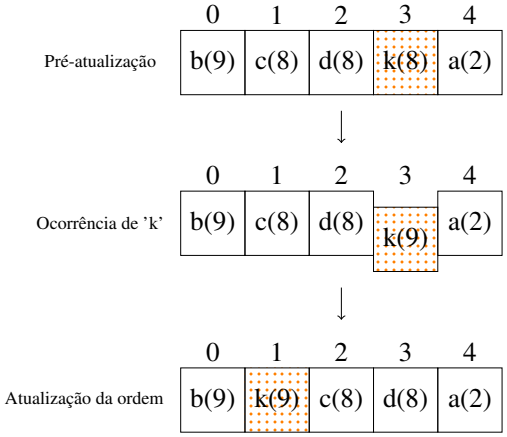

Figura 4.9 – Atualização LUÍSA-FS I

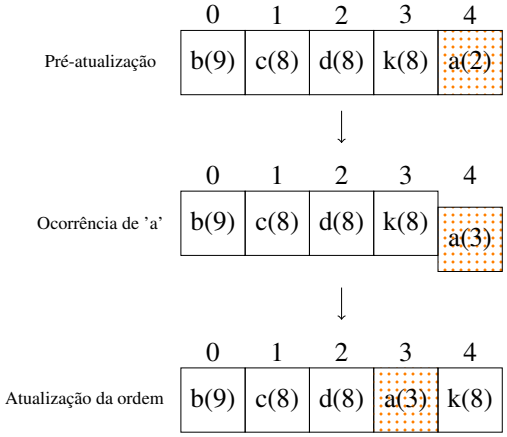

Figura 4.10 – Atualização LUÍSA-FS II

Utilizando essa estratégia de atualização, o algoritmo é capaz de garantir a ascensão mínima de um símbolo, ao mesmo tempo em que privilegia os símbolos muito frequentes. No entanto, assim como nas estratégias anteriores, símbolos novos (ou pouco frequentes) demoram muita para serem promovidos a uma posição de destaque, pois eles são movidos uma posição de cada vez.

4.4.4 LUÍSA-MTF

Apesar de representar um avanço ao conseguir tornar a codificação adaptável sem desprezar a importância que um símbolo conquistou, símbolos novos ou pouco frequentes demoram a serem promovidos. Isso ocorre pelo uso da estratégia de troca, que move esses símbolos uma posição de cada vez.

A técnica LUÍSA MOVE TO FRONT (ou LUÍSA-MTF) foi proposta para contornar essa limitação. Em vez de mover um símbolo por vez, o símbolo é sempre movido para a

primeira posição (índice zero).

A Figura 4.11 exemplifica a atualização das posições dos símbolos em um contexto após a ocorrência do símbolo 'k'. Como pode-se ver, o símbolo passa a ocupar a primeira posição, e todos os demais símbolos que antecediam 'k' retrocedem uma posição.

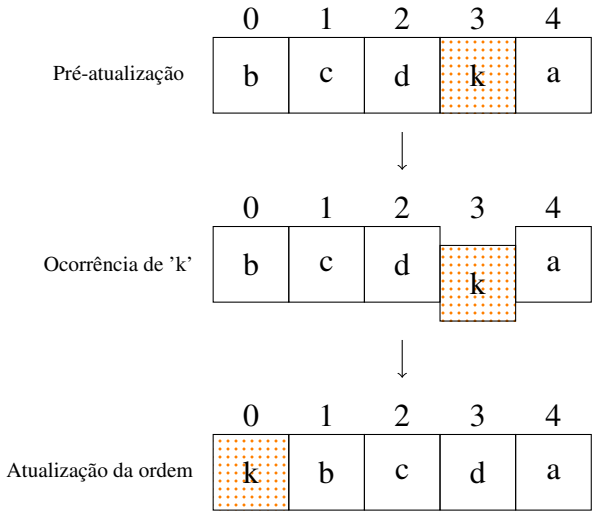

Figura 4.11 – Atualização LUÍSA-MTF

A técnica *Move To Front* já é utiliza dentro do método de compressão BWT. Nesse caso, o propósito é transformar uma entrada composta por muitos símbolos repetidos em índices zero ou índices de baixo valor. Dentro do LUÍSA, o propósito do MTF é semelhante. No entanto, enquanto no BWT é mantida uma lista de índices global, no LUÍSA é mantida uma lista de índices para cada contexto.

A estratégia MTF é interessante para a compressão de arquivos ordenados, ou seja, arquivos compostos por valores que aparecem em ordem ascendente. Exemplos em que os dados podem vir ordenados são listagem de nomes de pessoas e listagem de palavras de dicionários. Nesse tipo de arquivo, quando um símbolo aparece, as chances são grandes que ele volte a aparecer quando voltar a ocorrer o mesmo contexto.

A adaptabilidade oferecida pelos métodos LUÍSA-S, LUÍSA-FS e LUÍSA-MTF é possível pelo fato da codificação ser desvinculada (ao menos parcialmente) da frequência de ocorrência do símbolo. Já no PPM, a frequência é indispensável para a codificação, o que dificulta a adoção de estratégias de promoção.

Conforme discutido no capítulo de trabalhos relacionados, existem variações do PPM que levam em conta a localidade dos dados, para que símbolos novos recebam valores de frequência altos mais rapidamente. Esses valores de frequência são gerados artificialmente,

através de multiplicação da frequência por um fator de escala. O problema com essa abordagem é a dificuldade em conceber um fator de escala que encontre um equilíbrio entre a valorização de símbolos novos sem depreciar de forma muito acentuada os demais.

4.5 Árvore de contexto

A árvore de contexto usada no LUÍSA possui muita semelhança à árvore usada no PPM. A principal diferença é que, enquanto no PPM a árvore tem um papel duplo (busca e emissão da probabilidade), no LUÍSA a árvore é usada somente para a realização de buscas.

Como a responsabilidade da árvore é reduzida, é possível também que menos informações sejam armazenadas em cada nodo, o que leva a uma redução no espaço em memória ocupado pela árvore. Por exemplo, a frequência total dos símbolos em um contexto é uma informação importante no contexto do PPM. No LUÍSA, essa informação não possui relevância e pode ser excluída. Além disso, caso a estratégia usada para o posicionamento dos símbolos não dependa da frequência para nada, é possível remover a frequência de cada símbolo do contexto, levando a uma economia de memória ainda maior.

Outra diferença sutil entre as estruturas usadas pelo PPM e pelo LUÍSA diz repeito ao nível -1, usado pelo PPM. Nesse nível, todos os possíveis símbolos são iniciados com frequência mínima e fixa durante toda a execução do algoritmo. Já no nível zero constam todos os símbolos que já apareceram pelo menos uma vez, independente do contexto. Essa separação visa garantir uma melhor compressão para símbolos inéditos, pois ele terá uma probabilidade igual a todos os demais símbolos. Caso esse símbolo dividisse o espaço com símbolos não inéditos de alta frequência, a sua probabilidade seria mínima, o que levaria a uma menor taxa de compressão. Como no LUÍSA a compressão independe da frequência, é possível mesclar os níveis -1 e 0 em uma única ordem 0. Essa ordem é inicializada no começo do processo de compressão com todos os possíveis símbolos do alfabeto.

5 EXPERIMENTOS

Esse capítulo apresenta experimentos que foram realizados usando o método LUÍSA. Os experimentos demonstram como esse método se compara a outros compressores em termos de taxa de compressão. A taxa de compressão é medida em bpc (*bits per code*), que é a quantidade média de bits necessária para representar um símbolo de 8 bits. Quanto menor o bpc, maior é a taxa de compressão.

O Capítulo 4 apresentou diversas possibilidades de implementação do método. De todas as variações apresentadas, algumas foram implementadas. Como os experimentos focam na taxa de compressão, foram concentrados esforços na implementação das variações que gerem menores bpcs, mesmo que fosse a custo de um maior tempo de processamento. As configurações avaliadas estão descritas abaixo:

- **Geração das chaves:** as chaves foram geradas usando acúmulo com exclusão.
- **Codificação entrópica:** a compressão foi realizada usando codificação aritmética dinâmica, com uma tabela global de frequência.
- **Fluxo dos dados:** as chaves geradas são enviadas ao codificador na forma de *streaming*, sem uso de *buffers* intermediários.
- **Posicionamento dos símbolos:** as quatro estratégias apresentadas neste trabalho foram implementadas. São elas: LUÍSA-F, LUÍSA-S, LUÍSA-FS e LUÍSA-MTF.

Percebe-se que a única variação ocorreu na definição da estratégia para posicionamento dos símbolos. Atenção especial foi dada à esse aspecto porque o LUÍSA se mostrou particularmente competitivo para comprimir dados em que a localidade é acentuada. Algumas estratégias de posicionamento são úteis para dados dessa natureza, conforme será apresentado neste capítulo.

Para fins de comparação, também foi realizada a medição de bpc para outros dois compressores:

- **GZip:** implementa o padrão DEFLATE, que é baseado em um método da família LZ.
- **PPM:** método de compressão que serviu de inspiração para o LUÍSA.

O GZip é uma ferramenta de compressão comercial bastante utilizada, pois alia boas taxas de compressão e velocidade de processamento, tanto na compressão quanto na descompressão. Quanto ao PPM, não existe ferramenta comercial disponível, e as implementações acadêmicas são de difícil configuração. Assim, esse compressor foi desenvolvido como parte deste trabalho. O método PPM-C foi empregado, uma vez que ele é a base para trabalhos nessa área. Todos os algoritmos avaliados foram escritos em C e nenhuma *flag* de otimização foi utilizada para compilação/execução dos códigos fonte.

5.1 Compressão de Arquivos com Baixa Entropia Global

Arquivos com baixa entropia global são aqueles em que um símbolo aparece de forma frequente, dentro de um contexto, em qualquer trecho do arquivo. Ou seja, se um símbolo é frequente em um contexto, é provável que ele continue sendo frequente em todo o arquivo. Exemplos são arquivos de texto compostos por parágrafos, como livros e textos informativos, além de arquivos com estrutura JSON (JavaScript Object Notation) e XML (eXtensible Markup Language).

Para avaliar os métodos de compressão para arquivos dessa natureza, foi utilizado o Corpus *Calgary* (BELL; WITTEN; CLEARY, 1989). O corpus é um *benchmark* de referência para testes de compressão, composto por 14 arquivos de origens variadas. O formato de cada arquivo é detalhado a seguir:

- **bib:** arquivo de texto ASCII contento 725 referências bibliográficas em formato *UNIX refer*. O arquivo ocupa aproximadamente 109KB.
- **book1:** arquivo de texto ASCII contento o livro intitulado "*Far from the Madding Crowd*"de Thomas Hardy. O arquivo ocupa aproximadamente 751KB.
- **book2:** arquivo de texto ASCII contendo o livro intitulado "*Principles of Computer Speech*"de Ian Witten em formato *UNIX troff*. O arquivo ocupa aproximadamente 597KB.
- **geo:** arquivo contendo números em ponto flutuante de 32 bits indicando informações sísmicas. O arquivo ocupa aproximadamente 100KB.
- **news:** arquivo de texto ASCII contendo uma miscelânea de notícias. O arquivo ocupa aproximadamente 369KB.

- **obj1:** executável gerado a partir da compilação do arquivo "progp"para arquitetura VAX. O arquivo ocupa aproximadamente 21KB.
- **obj2:** executável compilado para plataforma Macintosh. O arquivo ocupa aproximadamente 242KB.
- **paper1:** arquivo de texto ASCII contento o artigo intitulado "*Arithmetic Coding for Data Compression*"de Ian Witten em formato *UNIX troff*. O arquivo ocupa aproximadamente 52KB.
- **paper2:** arquivo de texto ASCII contento o artigo intitulado "*Computer (in)security*"de Ian Witten em formato *UNIX troff*. O arquivo ocupa aproximadamente 81KB.
- **pic:** imagem em formato Bitmap com dimensões de 1728 x 2376. O arquivo ocupa aproximadamente 502KB.
- **progc:** código fonte em linguagem C. O arquivo ocupa aproximadamente 39KB.
- **progl:** código fonte em linguagem Lisp. O arquivo ocupa aproximadamente 70KB.
- **progp:** código fonte em linguagem Pascal. O arquivo ocupa aproximadamente 49KB.
- **trans:** transcrição de comandos digitados em um terminal VAX. O arquivo ocupa aproximadamente 92KB.

O teste inicial visa determinar qual das variações do método LUÍSA apresenta melhor desempenho para aquivos com baixa entropia global. Para isso, cada arquivo do conjunto de testes foi submetido à compressão pelos métodos LUÍSA-F, LUÍSA-S, LUÍSA-FS e LUÍSA-MTF. O número máximo de contextos também variou de 2 e 10.

A Figura 5.1 apresenta o resultado da compressão do arquivo *paper1*. Os resultados mostram que o método LUÍSA-F aliado a 6 níveis de contexto leva a um menor bpc. Essa foi a configuração que apresentou a maior taxa de compressão média, considerando todos os arquivos do corpus. Sendo assim, os demais experimentos são baseados nessa variação.

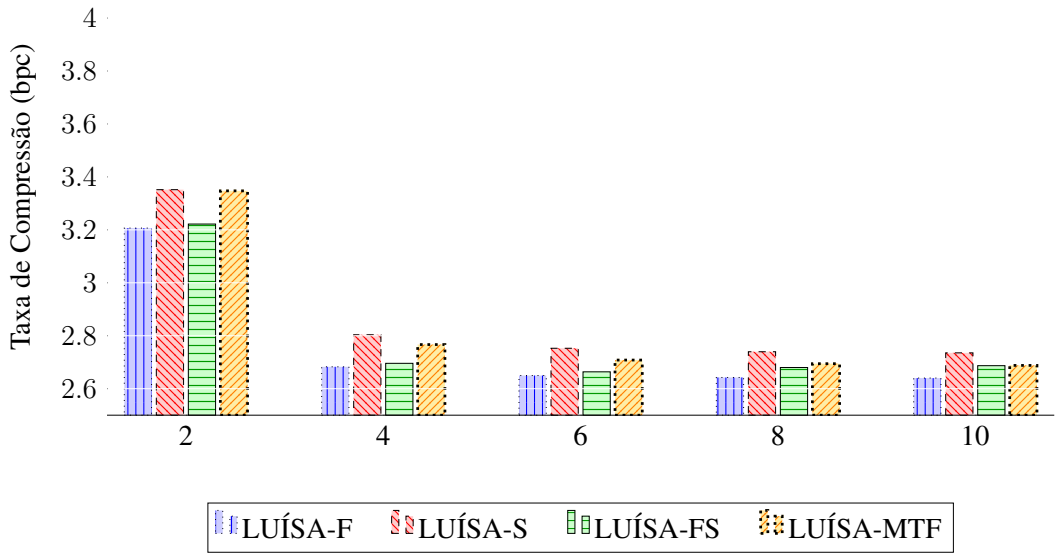

Figura 5.1 – Análise LUÍSA (*paper1*)

As Figuras 5.2 e 5.3 apresentam as taxas de compressão para todos os arquivos do Corpus *Calgary*. Além do LUÍSA-F com 6 ordens de contexto, foram incluídos os bpcs para o GZip e para o PPM-C com 4 ordens de contexto. A escolha do número de contextos para o PPM-C também foi feita de forma empírica, ou seja, escolheu-se a configuração que levou a melhores resultados para a maioria dos arquivos.

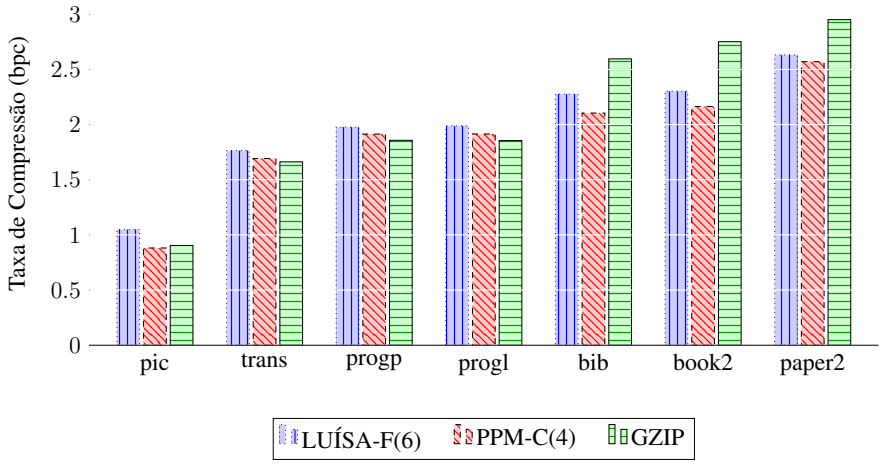

Figura 5.2 – Calgary Corpus (Parte I)

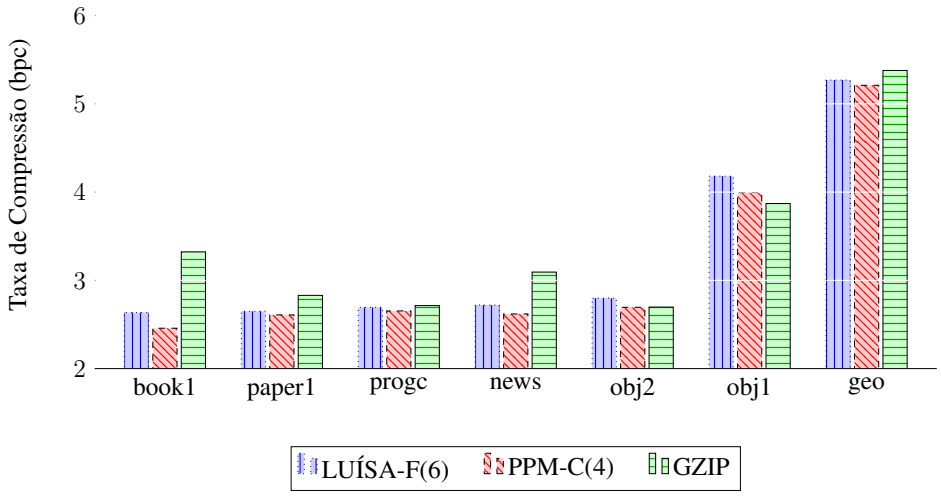

Figura 5.3 – Calgary Corpus (Parte II)

O PPM-C se destaca como o método que atinge melhores taxas de compressão. Dos 14 arquivos considerados, o PPM foi o melhor em 10. Já o GZip foi o melhor nas 4 entradas restantes. Além disso, o PPM foi melhor do que o LUÍSA em todos os cenários testados. Apesar de o LUÍSA não ser superior, ele se mantém muito próximo ao método que obtém a melhor taxa de compressão em todas as situações. Na média, o método usa 0,096 bpc a mais por símbolo, em comparação ao melhor método. Já o GZip usa 0,234 bpc a mais por símbolo.

As Figuras 5.4, 5.5 apresentam o comportamento dos métodos PPM-C e LUÍSA-F quando ocorre variação na quantidade de ordens de contexto. Os resultados mostram que a taxa de compressão com o método PPM é muito sensível ao número de contextos usados. A partir da quantidade de ordens que leva à melhor taxa de compressão (4), o incremento ou decremento de ordens degrada rapidamente a compressão. Já a taxa de compressão com o LUÍSA apresenta um comportamento mais regular. A taxa de compressão aumenta de forma mais acentuada até certo nível, a partir do qual ela se mantém no mesmo patamar.

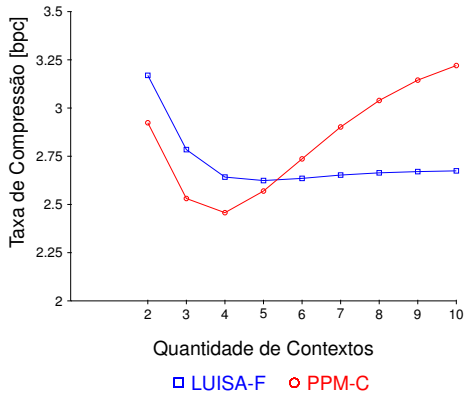

Figura 5.4 – Progressão de Contexto (book1)

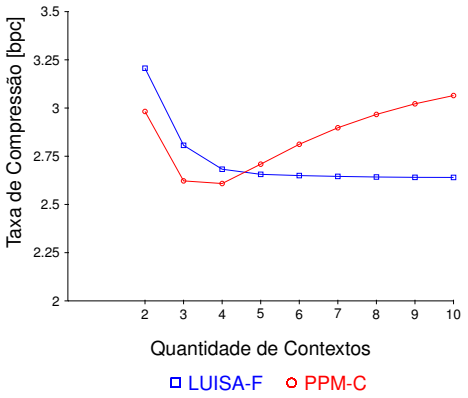

Figura 5.5 – Progressão de Contexto (paper1)

Esse comportamento de estabilidade observado no LUÍSA sugere que seja possível trabalhar com contextos de maior ordem sem o risco de sofrer perdas nas taxas de compressão. Já com o PPM, é necessário escolher um tamanho de ordem de forma mais criteriosa. Por exemplo, conforme indicado nas Figuras 5.4 e 5.5, se qualquer outro tamanho máximo acima de 5 ordens de contexto fosse usado no PPM, o resultado seria pior do que o resultado gerado com o LUÍSA usando 10 ordens de contexto.

A quantidade ideal de ordens de contexto para o método PPM, do ponto de vista da obtenção de melhores taxas de compressão, é variável de acordo com as características dos dados de entrada. De maneira geral, resultados de testes empíricos realizados em diversos conjuntos de dados com baixa entropia global (CLEARY; WITTEN, 1984a), (MOFFAT, 1990), (GARCIA; MERGEN, 2016b) demonstram que a melhor escolha varia entre 3 e 5 ordens de contexto.

5.2 Compressão de Arquivos com Baixa Entropia Local

Arquivos com baixa entropia local são aqueles em que a frequência de um símbolo dentro de um contexto é elevada somente em determinados trechos do arquivo. Ou seja, são arquivos que apresentam localidade acentuada. Se um símbolo ocorreu dentro de um contexto, é esperado que ele ocorra novamente, pelo menos em algumas das próximas ocorrências do mesmo contexto. Já em ocorrências mais distantes, é esperado que o símbolo não volte a surgir. Exemplos onde aparece a localidade acentuada são arquivos ordenados, como dicionários, em que a informação é dividida em entradas ordenadas. Nesse tipo de arquivo, os símbolos tendem a se repetir entre uma entrada e a seguinte. Também é possível citar como exemplo de arquivos com localidade marcante algumas imagens do tipo BMP.

Ao contrário dos arquivos que apresentam baixa entropia globalizada, onde o símbolo aparece de forma frequente em qualquer trecho, arquivos com localidade acentuada prejudicam o desempenho do método PPM, que usa a frequência do símbolo dentro do contexto para o cálculo da probabilidade. Já o LUÍSA pode usar estratégias que não dependam exclusivamente da frequência. Essa seção demonstra como essa diferença no uso da frequência impacta na taxa de compressão.

Um teste inicial foi realizado usando quatro conjuntos de dados ordenados de diferentes formatos. A natureza de cada conjunto é detalhada abaixo:

- Lista de Números Inteiros (LNI): lista de números inteiros contendo 500.000 registros. Os números foram gerados de forma aleatória e estão compreendidos no intervalo entre 0 e 10.000.000. O arquivo ocupa aproximadamente 4,23MB.

- Lista de Nomes de Pessoas (LNP): lista de nomes próprios contendo 5.494 registros. Base de dados disponível online (MARKETING, 2009). O arquivo ocupa aproximadamente 42,7KB.

- Lista de Sobrenomes de Pessoas (LSP): lista de sobrenomes contendo 88.799 registros. Base de dados disponível online (MARKETING, 2009). O arquivo ocupa aproximadamente 765KB.

- Lista de Palavras em Português (LPP): lista de palavras da língua portuguesa obtida através de um dicionário. O dicionário foi criado a partir de arquivos .aff e .dic, utilizados para correção ortográfica. A lista conta com 994.115 registros e ocupa aproximadamente 13,2MB.

O primeiro teste consiste em encontrar a melhor variação do método LUÍSA entre as versões F, S, FS e MTF. A Figura 5.6 apresenta a taxa de compressão obtida para o arquivo contendo a lista ordenada lexicograficamente de palavras da língua portuguesa. Como os demais resultados apresentam taxas similares, os resultados foram omitidos.

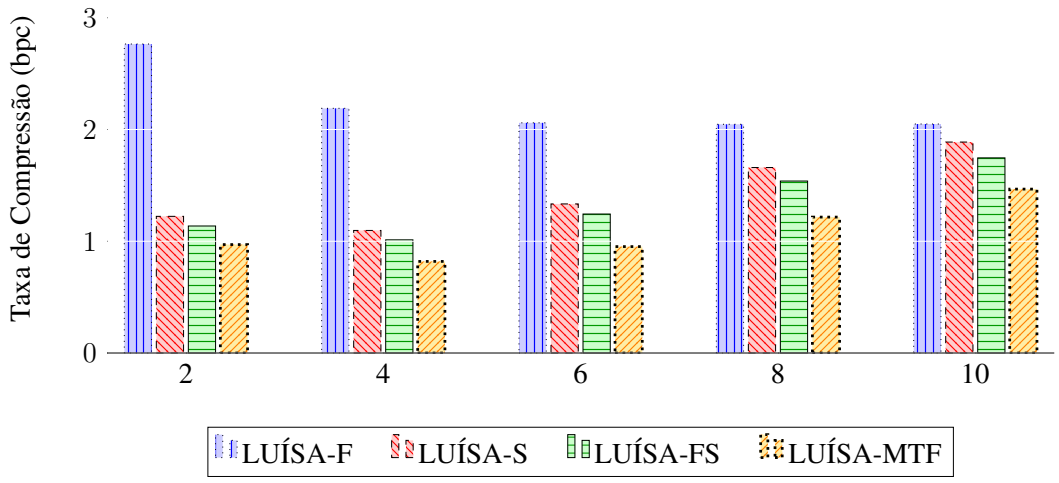

Figura 5.6 – Análise LUÍSA (*LPP*)

A estratégia LUÍSA-MTF com 4 ordens de contexto é a que apresenta os melhores resultados. Nessa estratégia, um símbolo recém codificado é remanejado para a primeira posição

dentro do contexto. Esse deslocamento abrupto permite que a localidade dos dados comece a ser explorada rapidamente. Já a versão LUÍSA-F apresenta os piores resultados, uma vez que ela utiliza unicamente a informação de frequência para remanejar os símbolos. Símbolos com alta frequência que deixam de aparecer comprometem o desempenho quando se usa essa estratégia.

A Figura 5.7 apresenta uma comparação entre o LUÍSA MTF(4), o GZip e o PPM-C(4). A escolha do número de contextos para o PPM-C também foi feita de forma empírica, ou seja, escolheu-se a configuração que levou a melhores resultados para a maioria dos arquivos.

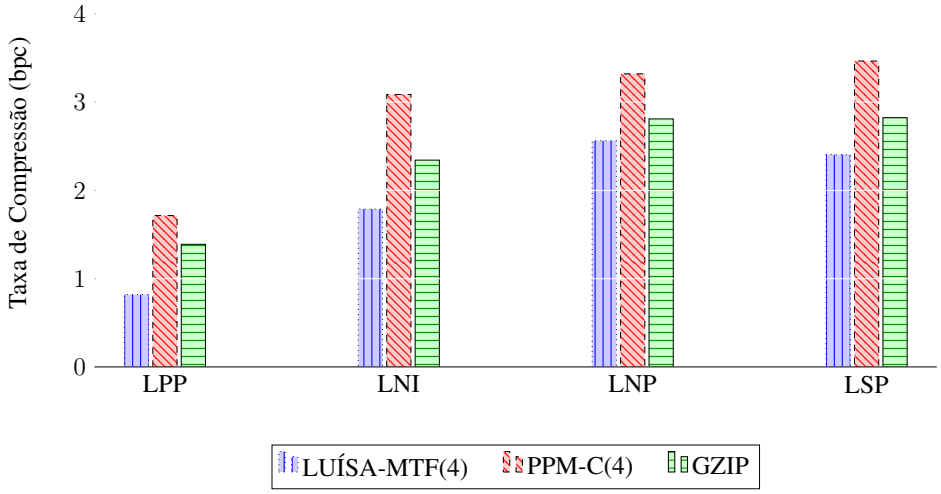

Figura 5.7 – Conjunto Ordenado

A Figura mostra que o LUÍSA MTF apresenta a maior taxa de compressão para todos os conjuntos de dados, seguido pelo GZip e pelo PPM. Como pode-se ver pelo desempenho do PPM, o uso exclusivo da frequência é prejudicial para dados com localidade acentuada.

Outro aspecto relevante é que todos os compressores obtém os melhores resultados quando processam palavras de língua portuguesa. Em parte, isso deve-se ao fato de esse ser o maior arquivo, o que o leva a ter entropia reduzida. Nesse conjunto em especial, o LUÍSA obteve uma taxa de compressão significante, pelo menos duas vezes superior aos concorrentes.

Um experimento complementar foi realizado usando dicionários de diferentes idiomas. Os dicionários foram criados a partir de diferentes arquivos .dic e .aff, usados para correção ortográfica. A tabela 5.1 apresenta características gerais de cada um dos arquivos.

Nome	Registros	Tamanho (KB)
Alemão (AL)	853.064	12.309
Catalão (CT)	1.644.523	22.261
Espanhol (ES)	739.714	8.675
Francês (FR)	676.921	8.715
Inglês (IN)	136.446	1.484
Polonês (PL)	3.623.541	49.935
Português (PT)	994.115	13.555
Tcheco (TC)	4.651.797	58.948

Tabela 5.1 – Dicionários usados nos testes de compressão

A Figura 5.8 apresenta o comparativo das taxas de compressão entre os métodos testados para os diferentes dicionários. Os resultados gerados confirmam que o LUÍSA-MTF tem grande capacidade adaptativa, obtendo altas taxas de compressão quando a localidade dos dados é um fator importante para um arquivo de entrada.

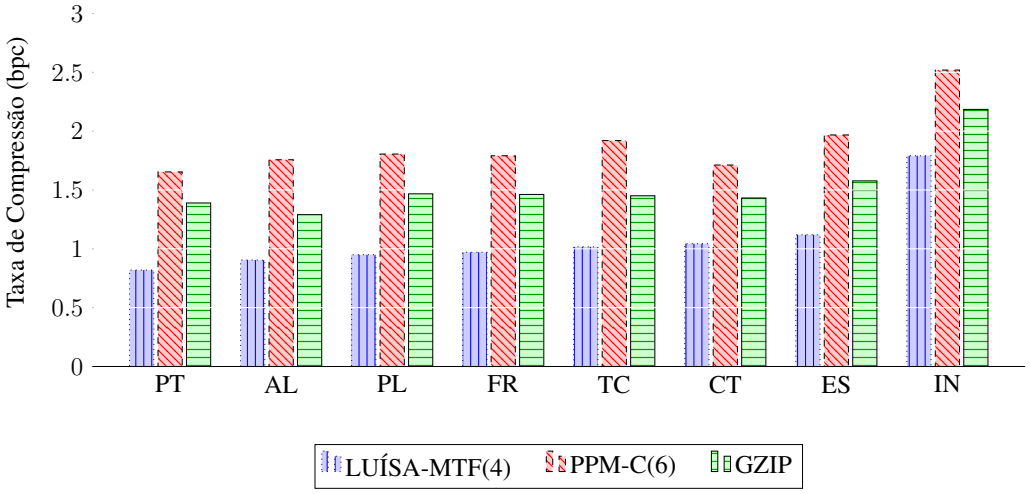

Figura 5.8 – Conjunto de Dicionários

5.3 Compressão de Arquivos Genéricos

As duas seções anteriores mostraram que, dentre as variações do LUÍSA, a versão F obtém melhores resultados quando a baixa entropia é global, enquanto a versão MTF obtém melhores resultados quando a baixa entropia é local. No entanto, pode ser que não se conheça de antemão a natureza dos dados. Nesses casos, é preciso que seja escolhida uma estratégia genérica, capaz de obter boas taxas de compressão para tipos variados de arquivo.

Das quatro estratégias avaliadas, a versão FS é a que satisfaz esse compromisso. Suas taxas de compressão são sempre próximas da versão vencedora, independente da natureza dos

dados. Como essa variação utiliza frequência, símbolos com alta frequência global são privilegiados. Além disso, como símbolos novos ascendem rapidamente, símbolos com alta frequência local também são favorecidos.

As Figuras 5.9 e 5.10 apresentam uma comparação entre a versão F, com melhor taxa média de compressão para o Corpus *Calgary*, e a versão FS, que apresenta a segunda melhor média (comparações estendidas presentes nas Figuras A.3, A.4, A.5 e A.6). Foram utilizados o mesmo método entrópico (codificação aritmética) e números de ordens para as duas versões. Os gráficos constatam que a versão FS apresenta taxas de compressão muito similares a versão F para dados de baixa entropia global.

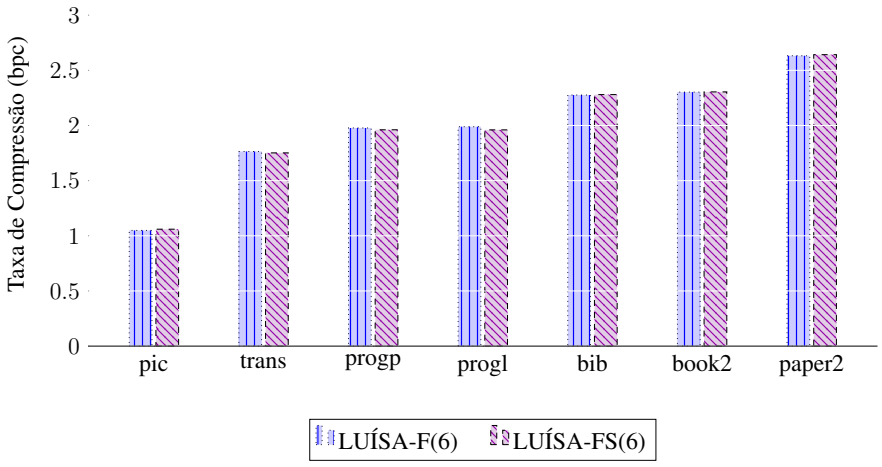

Figura 5.9 – Comparação F e FS (Calgary Corpus - Parte I)

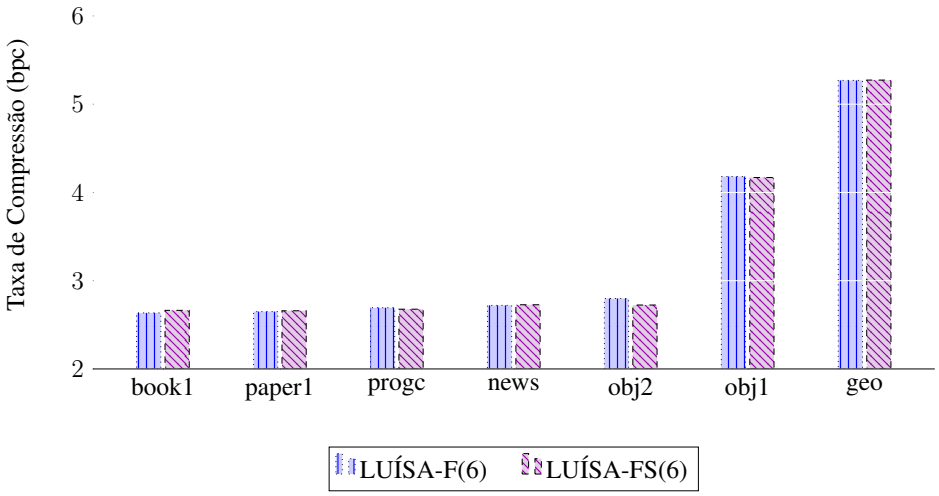

Figura 5.10 – Comparação F e FS (Calgary Corpus - Parte II)

A Figura 5.11 apresenta uma comparação entre a versão MTF, com melhor taxa média de compressão para os arquivos de dicionário, e a versão FS, que apresenta a segunda melhor

média (comparações estendidas presentes nas Figuras A.7 e A.8). Foram utilizados o mesmo método entrópico (codificação aritmética) e números de ordens para as duas versões. Nesse caso, a taxa de compressão apresenta queda mais significativa entre as versões. Mesmo assim, a versão FS apresenta resultados significativamente melhores do que o GZip e PPM-C, para a maioria dos dicionários [1].

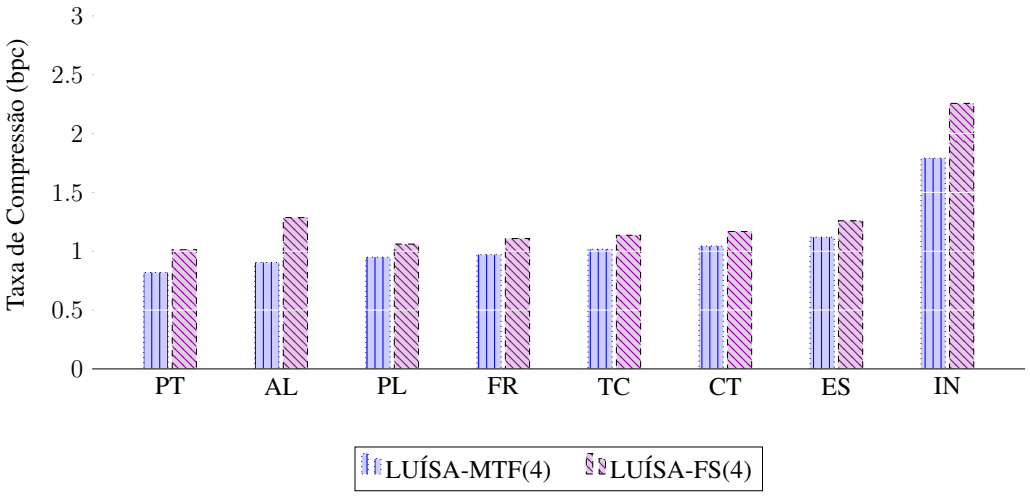

Figura 5.11 – Comparação F e FS (Conjunto de Dicionários)

5.4 Mudança de Compressor Entrópico Final

Conforme apresentado no Capítulo 4, o desacoplamento entre busca e codificação permite que essas duas etapas sejam modularizadas. Como consequência, é possível que o módulo de codificação seja substituído por outro de forma simples.

Nas seções 5.1, 5.2 e 5.3, a codificação aritmética foi utilizada como módulo de codificação. Esse tipo de codificação busca aproximar a entropia dos símbolos e, de forma geral, é uma boa escolha quando não há conhecimento de características específicas dos dados de entrada. Entretanto, quando as chaves que chegam ao codificador possuem determinados padrões, como longas sequências de mesmas chaves agrupadas, outros compressores entrópicos podem obter melhores resultados. Assim, a flexibilidade na escolha da estratégia de codificação é uma característica importante do método LUÍSA.

Essa seção apresenta resultados obtidos quando se usa como módulo de codificação o método PPM e o GZip. Ou seja, os próprios algoritmos usados na comparação com o LUÍSA são aproveitados como módulos de codificação entrópica.

[1] A exceção é o dicionário de língua inglês, em que o GZip é melhor

As Figuras 5.12 e 5.13 apresentam os arquivos do Corpus *Calgary* comprimidos pelo método LUÍSA-FR utilizando 4 ordens de contextos, associado a 3 codificações entrópicas diferentes: codificação aritmética, PPM-C utilizando 4 ordens de contexto e GZip.

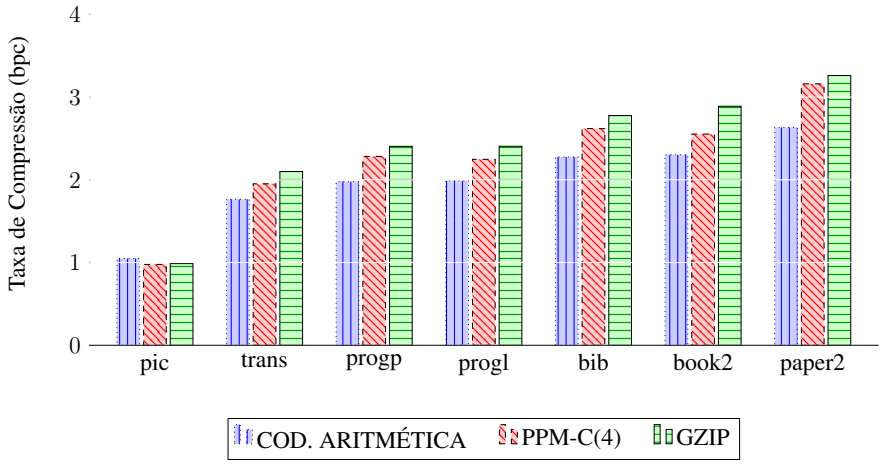

Figura 5.12 – Variação da Codificação Entrópica (Calgary Corpus - Parte I)

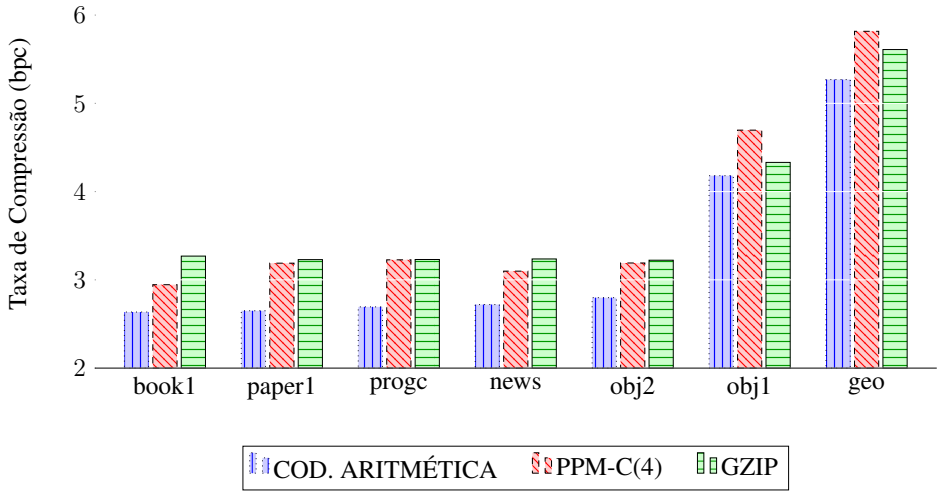

Figura 5.13 – Variação da Codificação Entrópica (Calgary Corpus - Parte II)

De forma geral, os arquivos presentes no Corpus *Calgary* apresentam média entropia com baixa presença de localidade de dados. Essas características levam à produção de chaves com frequência elevada. Porém, essas chaves iguais não estão dispostas em longas cadeias contínuas. Dessa forma, a codificação aritmética se beneficia da capacidade de aproximação de entropia, já que utiliza uma tabela de frequência, e obtém as melhores taxas de compressão entre os algoritmos testados.

A Figura 5.14 analisa o impacto da troca de codificação entrópica no método LUÍSA para os arquivos de dicionário. As chaves foram geradas com o método LUÍSA-MTF associado

a 4 ordens de contexto e os compressores entrópicos utilizados foram codificação aritmética, PPM-C com 6 ordens de contexto e GZip.

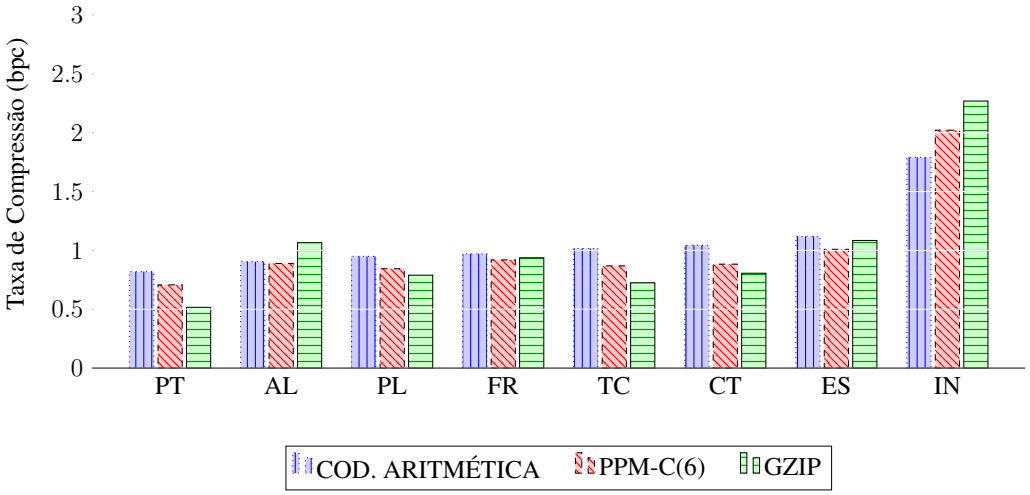

Figura 5.14 – Variação da Codificação Entrópica (Conjunto de Dicionários)

Os arquivos de dicionário apresentam redundância e localidade de dados marcantes. Diferentemente do Corpus *Calgary*, as chaves geradas pelo módulo de busca do LUÍSA formam longas cadeias de chaves iguais. Essa característica faz com que os métodos PPM e GZip sejam por vezes melhores do que a codificação aritmética. No caso específico do dicionário da língua portuguesa (PT), a taxa de compressão obtida quando se usa GZip como codificador é consideravelmente maior.

6 CONCLUSÕES

Este trabalho de conclusão propôs o LUÍSA, um método de compressão baseado no PPM. O método inova ao desacoplar a identificação do símbolo a comprimir e a sua codificação propriamente dita. Essa separação permite que diversas estratégias diferentes possam ser usadas em diferentes etapas da compressão. O texto apresentou algumas das possibilidades, destacando cenários em que elas são oportunas.

Os experimentos realizados tinham como objetivo mostrar o comportamento do LUÍSA quanto à sua capacidade de obter boas taxas de compressão. Para isso, foram implementadas as estratégias que visam maximizar essa medida.

Os resultados indicam que existem variações do LUÍSA promissoras para compressão de arquivos com baixa entropia local, como por exemplo, dicionários. A comparação com o PPM e GZip mostra que o LUÍSA-MTF consegue no mínimo um ganho de 18% em relação aos concorrentes. Para a compressão do dicionário da língua portuguesa, o ganho chegou a 41%.

Os resultados gerados com arquivos de baixa entropia global também são promissores. As taxas de compressão do LUÍSA-F perdem para o PPM. No entanto, o PPM é muito sensível à quantidade de ordens de contexto usada, enquanto o LUÍSA é menos suscetível ao valor usado. Como não é possível antecipar o valor ideal de ordens de contexto, o valor escolhido pode não levar aos melhores resultados no PPM. Por outro lado, o LUÍSA permite o uso de valores altos sem que isso leve a um prejuízo considerável na compressão.

Outra estratégia que merece destaque é o LUÍSA-FS, que valoriza tanto símbolos com frequência local elevada quanto símbolos com frequência global elevada. Os experimentos mostram que essa estratégia obtém um desempenho razoável na compressão de arquivos genéricos, em que não é possível antecipar a presença de símbolos com localidade acentuada.

Este trabalho concentrou esforços na implementação das estratégias mais vantajosas do ponto de vista da compressão. Os tempos de compressão e descompressão não foram abordados neste trabalho, porém existem maneiras de desenvolvimento de técnicas rápidas para o método LUÍSA. Assim, uma ideia interessante a ser investigada é avaliar o que pode ser feito para reduzir o tempo de processamento. Algumas possibilidades incluem:

- **Usar vetores na árvore de contexto:** vetores permitem que se descubra de forma direta se um símbolo existe em determinado contexto, sem que seja necessário percorrer listas encadeadas de nodos. Por outro lado, para evitar um consumo excessivo de memória, o

tamanho máximo do contexto teria que ser reduzido, o que pode causar redução na taxa de compressão.

- **Usar acúmulo sem exclusão:** essa estratégia evita o custo em ter que descobrir quais símbolos são comuns entre dois níveis de contexto vizinhos, uma vez que deixa de ser necessário o índice relativizado. Por outro lado, ela gera um alargamento na tabela de frequências, levando a chaves com maior entropia.

- **Reposicionar os símbolos usando apenas trocas:** o método LUÍSA-S tem um custo constante na atualização dos símbolos, pois apenas uma troca é necessária, sem que seja necessário comparar valores de frequência. Por outro lado, essa estratégia leva a piores taxas de compressão, como indicam os experimentos.

- **Usar codificação estática:** com a codificação aritmética estática, elimina-se o custo em ajustar a tabela de frequência conforme as chaves são geradas. A tabela é gerada de uma só vez, depois que todas as chaves são conhecidas, e usada sem ajustes durante a codificação. Por outro lado, a codificação estática geralmente leva a piores taxas de compressão. Além disso, é necessário armazenar essa tabela de frequências no arquivo comprimido, de modo que seja possível conduzir a descompressão.

- **Usar Codificação de Hufmann estática:** o codificador de Hufmann estático é considerado mais eficiente em termos de velocidade do que o codificador aritmético estático. Por outro lado, a codificação aritmética normalmente usa menos bits para representar cada chave.

- **Usar paralelismo:** através da bufferização das chaves, é possível executar em paralelo a busca e a codificação, através do paradigma de processamento paralelo produtor-consumidor.

Como trabalhos futuros, pretende-se implementar as estratégias mencionadas acima e avaliar o seu uso, quando testadas de forma isolada e de forma combinada. Em termos práticos, as estratégias propostas levam a diferentes taxas de compressão e velocidade de processamento. Quanto mais rápida for a compressão, menor tende a ser a taxa de compressão. Assim, é necessária uma análise de custo-benefício para que se encontre alguma solução que consiga aliar taxas de compressão e tempos de execução aceitáveis.

De modo geral, as contribuições deixadas por este trabalho são:

- Proposta de um novo método de compressão e discussão das diferentes estratégias que podem ser usadas.
- Implementações funcionais de diversas variações do método, que serão disponibilizados para uso público.
- Implementações funcionais de variações do método PPM, que serão disponibilizados para uso público.
- Análises comparativas que demonstram o comportamento do LUÍSA, PPM e GZip, referente às taxas de compressão obtidas.

Além disso, destacam-se as publicações em periódicos e anais de conferência que são frutos diretos e indiretos da pesquisa que permeou este trabalho de graduação. São eles:

- A análise do impacto da utilização de tabelas de dispersão no uso de memória e tempo de execução do método PPM (GARCIA; MERGEN, 2016a).
- A investigação do comportamento do método PPM para a compressão de dados orientados a coluna (GARCIA; MERGEN, 2016b). Este trabalho recebeu a distinção de melhor artigo da trilha de pesquisa do evento em que foi publicado.
- Um estudo sobre a influência de dados esparsos na taxa de compressão de bancos orientados a coluna (GARCIA; MERGEN, 2016c).

REFERÊNCIAS

BELL, T.; WITTEN, I. H.; CLEARY, J. G. Modeling for Text Compression. **ACM Comput. Surv.**, New York, NY, USA, v.21, n.4, p.557–591, Dec. 1989.

BENTLEY, J. L. et al. A Locally Adaptive Data Compression Scheme. **Commun. ACM**, New York, NY, USA, v.29, n.4, p.320–330, Apr. 1986.

BLOOM, C. New techniques in context modeling and arithmetic encoding. In: DATA COMPRESSION CONFERENCE, 1996. DCC '96. PROCEEDINGS. **Anais...** [S.l.: s.n.], 1996. p.426–.

BURROWS, M.; WHEELER, D. J. **A block-sorting lossless data compression algorithm**. [S.l.: s.n.], 1994.

CLEARY, J. G.; WITTEN, I. H. Data Compression Using Adaptive Coding and Partial String Matching. **IEEE Transactions on Communications**, [S.l.], v.32, n.4, p.396–402, 1984.

CLEARY, J.; WITTEN, I. A Comparison of Enumerative and Adaptive Codes. **IEEE Trans. Inf. Theor.**, Piscataway, NJ, USA, v.30, n.2, p.306–315, Sept. 1984.

DEUTSCH, P. **DEFLATE Compressed Data Format Specification Version 1.3**. United States: [s.n.], 1996.

GARCIA, V. F.; MERGEN, S. L. S. Usando Tabelas de Dispersão com PPM. In: XVI ESCOLA REGIONAL DE ALTO DESEMPENHO, 2016., São Leopoldo, RS, Brasil. **Anais...** [S.l.: s.n.], 2016. p.335–338.

GARCIA, V. F.; MERGEN, S. L. S. Compressão de Arquivos Orientados a Coluna com PPM. In: XII ESCOLA REGIONAL DE BANCO DE DADOS, 2016., Londrina, PR, Brasil. **Anais...** [S.l.: s.n.], 2016. p.40–49.

GARCIA, V. F.; MERGEN, S. L. S. Compression of Very Sparse Column Oriented Data. **Communications and Innovations Gazette**, Santa Maria, RS, BR, v.1, n.2, p.61–73, Oct. 2016.

HILBERT, M.; LóPEZ, P. The World's Technological Capacity to Store, Communicate, and Compute Information. **Science**, [S.l.], v.332, n.6025, p.60–65, April 2011.

HOWARD, P. G. **The Design and Analysis of Efficient Lossless Data Compression Systems**. 1993. Tese (Doutorado em Ciência da Computação) — , Providence, RI, USA. UMI Order No. GAX94-06956.

HUFFMAN, D. A. A Method for the Construction of Minimum-Redundancy Codes. **Proceedings of the Institute of Radio Engineers**, [S.l.], v.40, n.9, p.1098–1101, September 1952.

I.H. WITTEN, W. T.; CLEARY, J. Unbounded length contexts for PPM. **Data Compression Conference**, Los Alamitos, CA, USA, p.52, 1995.

LELEWER, D. A.; HIRSCHBERG, D. S. Data Compression. **ACM Comput. Surv.**, New York, NY, USA, v.19, n.3, p.261–296, Sept. 1987.

MARKETING, Q. A. **Free First-name and Last-name Databases (CSV and SQL)**. [Online; Acessado em 05/10/2016].

MOFFAT, A. Implementing the PPM data compression scheme. **IEEE Transactions on Communications**, [S.l.], v.38, n.11, p.1917–1921, Nov 1990.

ROBERTS, M. **Local-order-estimating Markovian analysis for noiseless source coding and authorship identification**. [S.l.: s.n.], 1982.

SHANNON, C. A Mathematical Theory of Communication. **Bell System Technical Journal**, [S.l.], v.27, p.379–423, 623–656, July, October 1948.

SHKARIN, D. A. Improving the Efficiency of the PPM Algorithm. **Problems of Information Transmission**, [S.l.], v.37, n.3, p.226–235, 2001.

TEAHAN, W. J. Probability estimation for PPM. In: IN PROCEEDINGS NZCSRSC'95. AVAILABLE FROM HTTP://WWW.CS.WAIKATO.AC.NZ/WJT. **Anais...** [S.l.: s.n.], 1995. p.papers/NZCSRSC.ps.gz.

WITTEN, I. H.; NEAL, R. M.; CLEARY, J. G. Arithmetic Coding for Data Compression. **Commun. ACM**, New York, NY, USA, v.30, n.6, p.520–540, June 1987.

ZHANG, Y.; ADJEROH, D. A. Prediction by Partial Approximate Matching for Lossless Image Compression. **IEEE Transactions on Image Processing**, [S.l.], v.17, n.6, p.924–935, June 2008.

ZIV, J.; LEMPEL, A. A universal algorithm for sequential data compression. **IEEE Transactions on Information Theory**, [S.l.], v.23, n.3, p.337–343, 1977.

APÊNDICES

APÊNDICE A – Imagens

A.1 Árvores

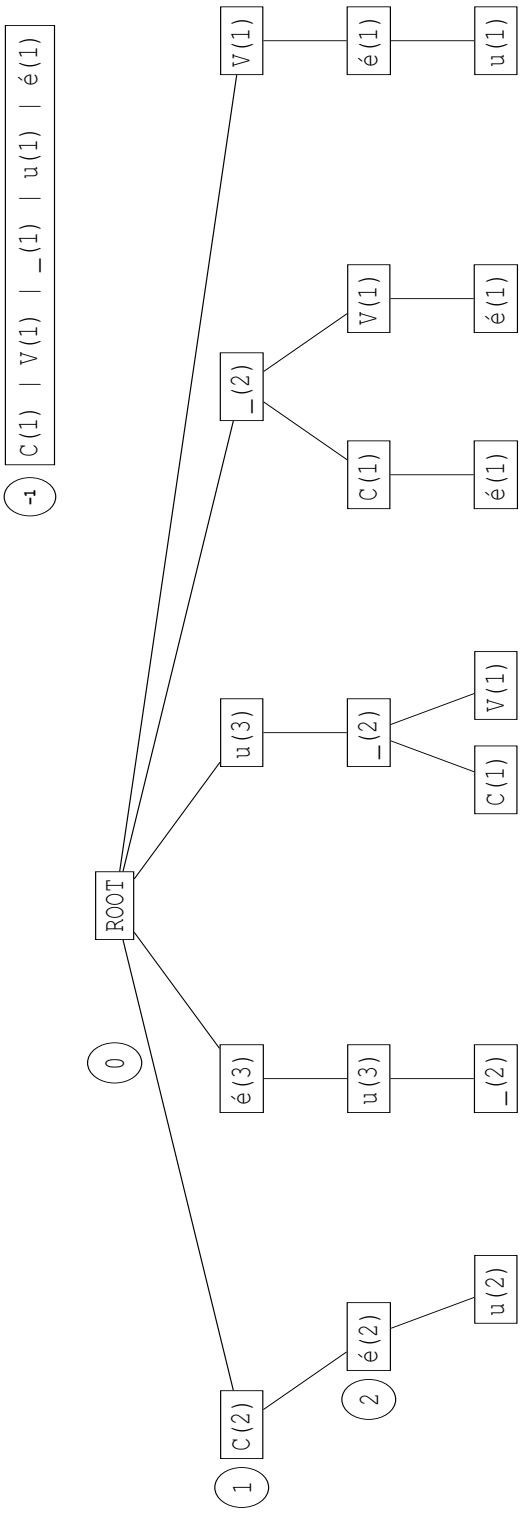

Figura A.1 – Árvore de contextos PPM-A e PPM-B final

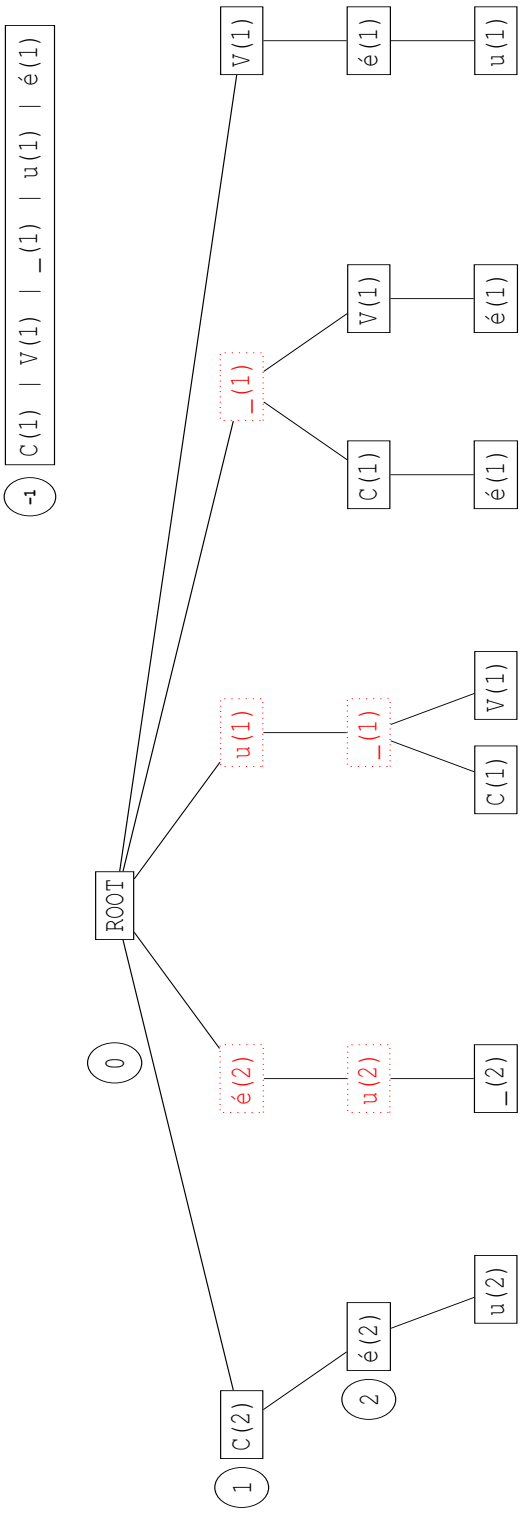

Figura A.2 – Árvore de contextos PPM-C final

A.2 Gráficos

As Figuras A.3, A.4, A.5 e A.6 apresentam a comparação das taxas de compressão para arquivos do Corpus *Calgary*. Os resultados exibem as taxas de compressão para o método

LUÍSA representado pelas versões F (melhor taxa de compressão média) e FS (compressão de arquivos genéricos). Os dados obtidos para o método LUÍSA são comparados com os compressores GZip e PPM-C.

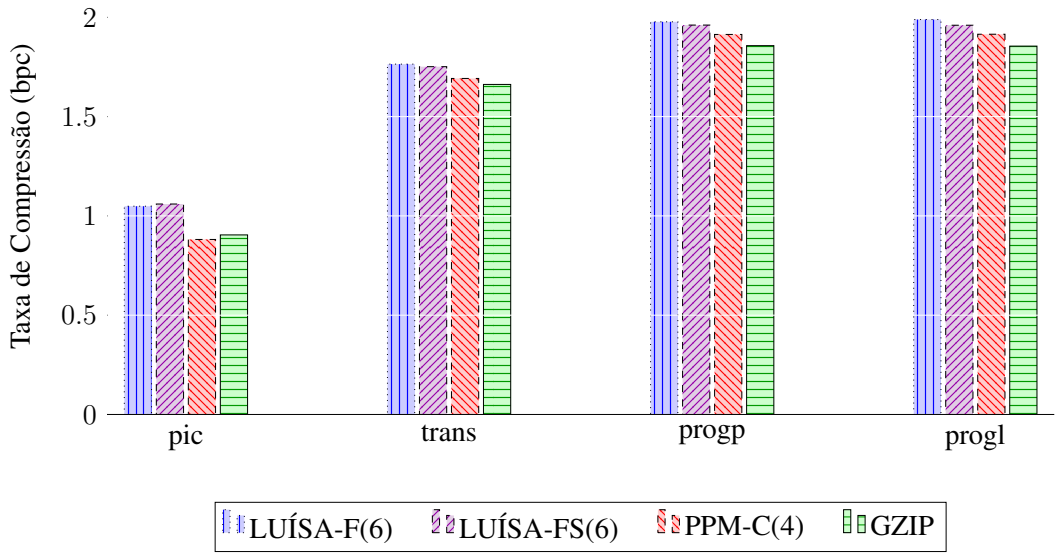

Figura A.3 – Comparações Calgary Corpus (Parte I)

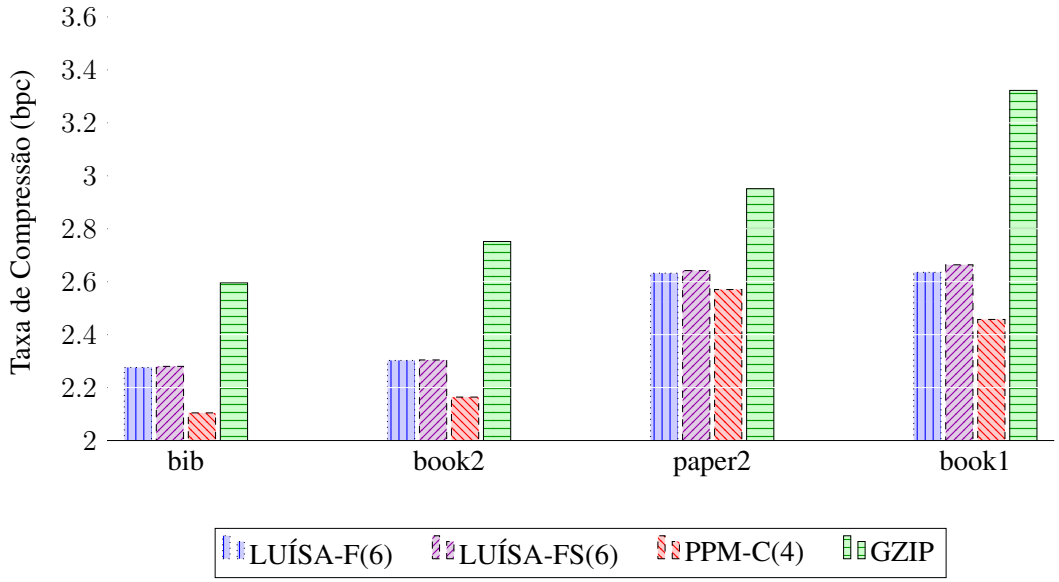

Figura A.4 – Comparações Calgary Corpus (Parte II)

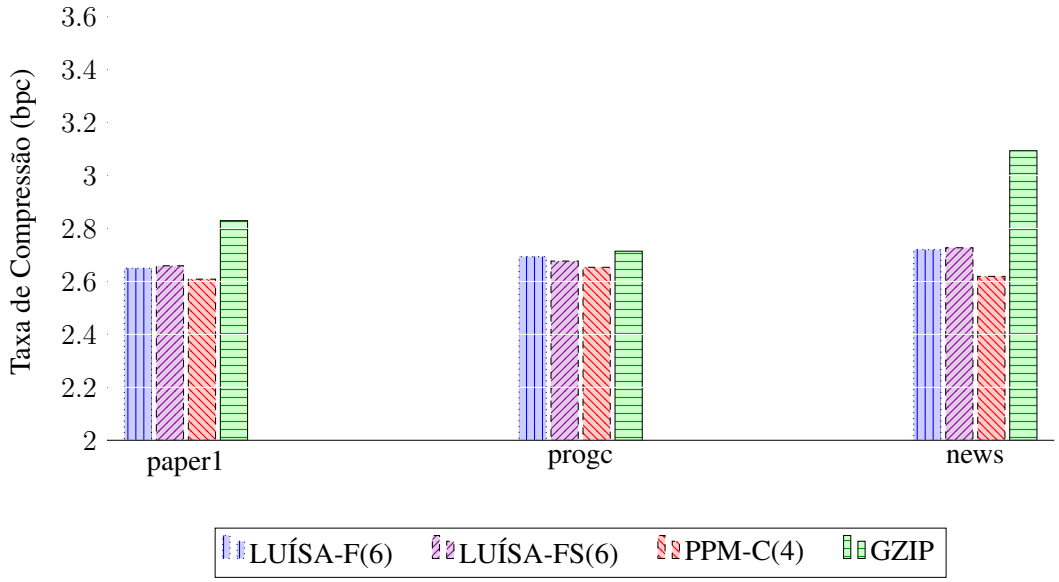

Figura A.5 – Comparações Calgary Corpus (Parte III)

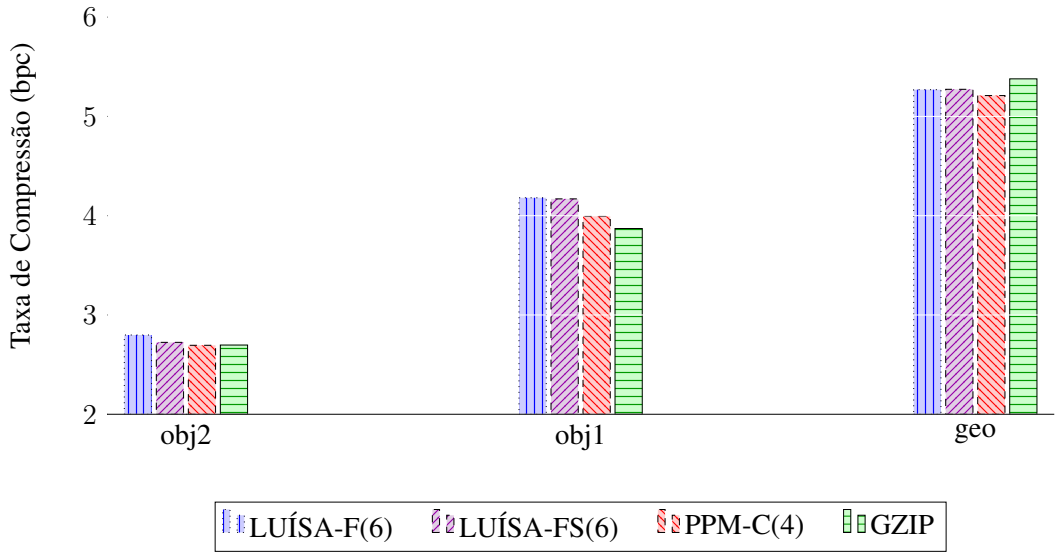

Figura A.6 – Comparações Calgary Corpus (Parte IV)

As Figuras A.7 e A.8 apresentam a comparação das taxas de compressão para arquivos do conjunto de dicionários. Os resultados exibem as taxas de compressão para o método LUÍSA representado pelas versões MTF (melhor taxa de compressão média) e FS (compressão de arquivos genéricos). Os dados obtidos para o método LUÍSA são comparados com os compressores GZip e PPM-C.

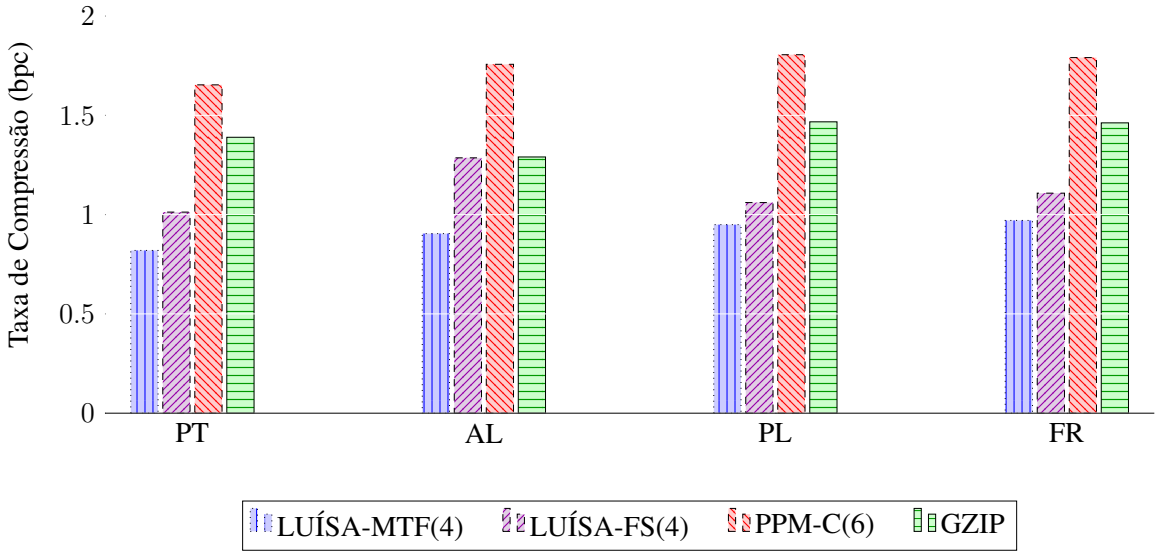

Figura A.7 – Comparações Conjunto de Dicionários (Parte I)

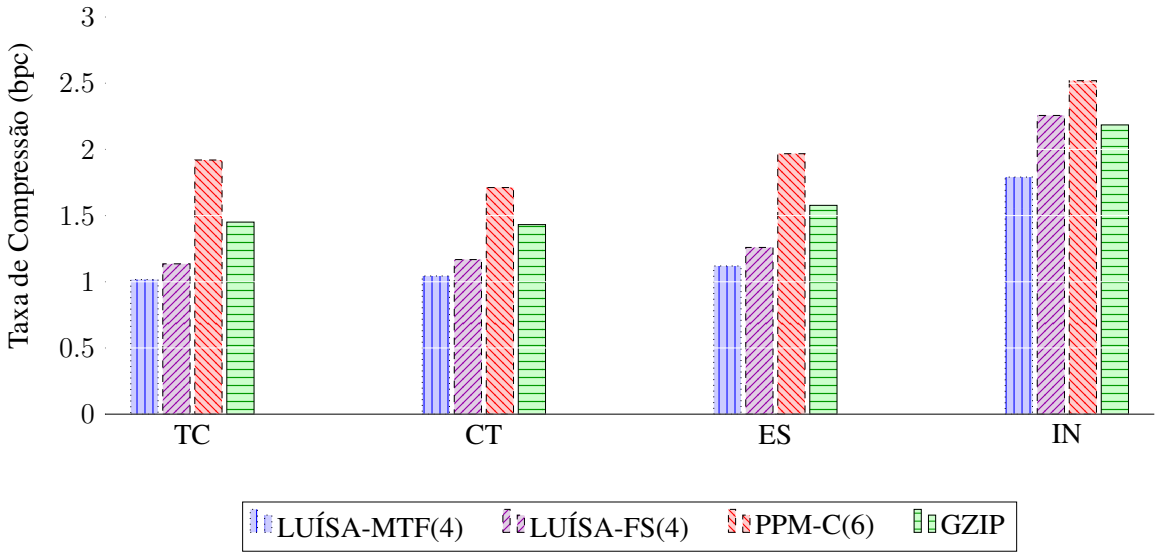

Figura A.8 – Comparações Conjunto de Dicionários (Parte II)

APÊNDICE B – Códigos

B.1 Codificação Aritmética

B.1.1 BitInputStream

```
#if !defined(BITINPUTSTREAM_H)
#define BITINPUTSTREAM_H

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

typedef struct BitInputStream {
   FILE *OriginalDataFile;
   unsigned char ReadBytes[250000];
   int NextReceivedBits;
   int ManyBitsRemaining;
   int ManyBytesRemaining;
   int LastByteBlockSize;
   bool StreamEnd;
} BitInputStream;

//CONSTRUCTOR
BitInputStream* BitInputStreamCreate(char *InputFileName);

//OPERATIONAL
int BitInputStreamNextBit(BitInputStream *MyBitInputStream);
void BitInputStreamCloseStream(BitInputStream *MyBitInputStream);

#endif
```

```
#include "BitInputStream.h"

//#########################################################
BitInputStream* BitInputStreamCreate(char *InputFileName){
   BitInputStream *NewBitInputStream = (BitInputStream*)
      malloc(sizeof(BitInputStream));

   NewBitInputStream->OriginalDataFile = fopen(InputFileName, "rb");
   if (NewBitInputStream->OriginalDataFile == NULL){
      printf("\nNo '%s' file found on 'BitInputStreamCreate'!",
         InputFileName);
      exit(3);
   }

   NewBitInputStream->NextReceivedBits = 0;
   NewBitInputStream->ManyBitsRemaining = 0;
   NewBitInputStream->ManyBytesRemaining = 0;
   NewBitInputStream->StreamEnd = false;

   return NewBitInputStream;
}
```

```
//#########################################################


---------------BEGIN OF MANAGEMENT FUNCTIONS----------------

//#########################################################
int BitInputStreamReadBit(BitInputStream *MyBitInputStream){


   if ((MyBitInputStream->ManyBytesRemaining == 0) &&
       (MyBitInputStream->ManyBitsRemaining == 0) &&
       (!MyBitInputStream->StreamEnd)){
      MyBitInputStream->ManyBytesRemaining =
          fread(MyBitInputStream->ReadBytes, sizeof(char), 250000,
          MyBitInputStream->OriginalDataFile);
      if(MyBitInputStream->ManyBytesRemaining != 0){
         MyBitInputStream->LastByteBlockSize =
            MyBitInputStream->ManyBytesRemaining;
      }
      else{
         MyBitInputStream->ManyBytesRemaining = 1;
         MyBitInputStream->StreamEnd = true;
      }
   }

   if (MyBitInputStream->ManyBitsRemaining == 0){
      MyBitInputStream->NextReceivedBits =
          MyBitInputStream->ReadBytes[MyBitInputStream->LastByteBlockSize -
          MyBitInputStream->ManyBytesRemaining];

      MyBitInputStream->ManyBitsRemaining = 8;
      if (!MyBitInputStream->StreamEnd){
         MyBitInputStream->ManyBytesRemaining--;
      }
   }

   MyBitInputStream->ManyBitsRemaining--;
   return (MyBitInputStream->NextReceivedBits >>
       MyBitInputStream->ManyBitsRemaining) & 1;
}
//#########################################################

//---------------END OF MANAGEMENT FUNCTIONS---------------

---------------BEGIN OF OPERATIONAL FUNCTIONS---------------

//#########################################################
int BitInputStreamNextBit(BitInputStream *MyBitInputStream){
   int ReceivedBit;

   ReceivedBit = BitInputStreamReadBit(MyBitInputStream);

   if ((ReceivedBit != 0) && (ReceivedBit != 1) && (ReceivedBit != EOF)){
      printf("\nInvalid bit value '%d' found on 'BitInputStreamNextBit'!",
          ReceivedBit);
      exit(3);
   }
```

```
   if (ReceivedBit == EOF){
      printf("\nEnd of file value found on 'BitInputStreamNextBit'!");
      exit(3);
   }

   return ReceivedBit;
}
//############################################################

//############################################################
void BitInputStreamCloseStream(BitInputStream *MyBitInputStream){
   fclose(MyBitInputStream->OriginalDataFile);
}
//############################################################

//---------------END OF OPERATIONAL FUNCTIONS---------------
```

B.1.2 BitOutputStream

```
#if !defined(BITOUTPUTSTREAM_H)
#define BITOUTPUTSTREAM_H

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

typedef struct BitOutputStream {
   FILE *FinalDataFile;
   bool EndStream;
   unsigned char ReadyBytes[250000];
   int CurrentByte;
   int ManyBitsInCurrentByte;
   int ManyReadyBytes;
} BitOutputStream;

//CONSTRUCTOR
BitOutputStream* BitOutputStreamCreate(char *OutputFileName);

//OPERATIONAL
void BitOutputStreamWriteBit(BitOutputStream *MyBitOutputStream, int
    ReceivedBit);
void BitOutputStreamCloseStream(BitOutputStream *MyBitOutputStream);

#endif
```

```
#include "BitOutputStream.h"

//############################################################
BitOutputStream* BitOutputStreamCreate(char *OutputFileName){
   BitOutputStream *NewBitOutputStream = (BitOutputStream*)
      malloc(sizeof(BitOutputStream));

   NewBitOutputStream->FinalDataFile = fopen(OutputFileName, "wb+");
   if (NewBitOutputStream->FinalDataFile == NULL){
```

```
      printf("\nCan not open '%s' file on 'BitOutputStreamCreate'!",
         OutputFileName);
      exit(4);
   }

   NewBitOutputStream->CurrentByte = 0;
   NewBitOutputStream->ManyBitsInCurrentByte = 0;
   NewBitOutputStream->ManyReadyBytes = 0;
   NewBitOutputStream->EndStream = false;

   return NewBitOutputStream;
}
//############################################################

//----------------BEGIN OF OPERATIONAL FUNCTIONS---------------

//############################################################
void BitOutputStreamWriteBit(BitOutputStream *MyBitOutputStream, int
    ReceivedBit){

   if ((ReceivedBit != 0) && (ReceivedBit != 1)){
      printf("\nInvalid bit value '%d' found on
         'BitOutputStreamWriteBit'!", ReceivedBit);
      exit(4);
   }

   MyBitOutputStream->CurrentByte = (MyBitOutputStream->CurrentByte << 1)
       | ReceivedBit;
   MyBitOutputStream->ManyBitsInCurrentByte++;

   if (MyBitOutputStream->ManyBitsInCurrentByte == 8){

      MyBitOutputStream->ReadyBytes[MyBitOutputStream->ManyReadyBytes] =
         MyBitOutputStream->CurrentByte;
      MyBitOutputStream->CurrentByte = 0;
      MyBitOutputStream->ManyBitsInCurrentByte = 0;
      MyBitOutputStream->ManyReadyBytes++;

      if ((MyBitOutputStream->ManyReadyBytes == 250000) ||
          (MyBitOutputStream->EndStream)){
         fwrite(MyBitOutputStream->ReadyBytes, sizeof(char),
            MyBitOutputStream->ManyReadyBytes,
            MyBitOutputStream->FinalDataFile);
         MyBitOutputStream->ManyReadyBytes = 0;
      }
   }
}
//############################################################

//############################################################
void BitOutputStreamCloseStream(BitOutputStream *MyBitOutputStream){

   if (MyBitOutputStream->ManyBitsInCurrentByte == 7){
      MyBitOutputStream->EndStream = true;
   }
   BitOutputStreamWriteBit(MyBitOutputStream, 1);
   while (MyBitOutputStream->ManyBitsInCurrentByte != 0){
      if (MyBitOutputStream->ManyBitsInCurrentByte == 7){
```

```
            MyBitOutputStream->EndStream = true;
        }
        BitOutputStreamWriteBit(MyBitOutputStream, 0);
    }
    fclose(MyBitOutputStream->FinalDataFile);
}
//#########################################################

//-----------------END OF OPERATIONAL FUNCTIONS---------------
```

B.1.3 FrequencyTable

```
#if !defined(FREQUENCYTABLE_H)
#define FREQUENCYTABLE_H

#include <stdlib.h>
#include <stdio.h>

typedef struct FrequencyTable{
    int Frequences[257];
    int Total;
} FrequencyTable;

//CONSTRUCTOR
FrequencyTable* FrequencyTableCreate();

//CHECKING
int FrequencyTableGetSymbolLimit();

//OPERATIONAL
void FrequencyTableInsertOn(int Symbol, FrequencyTable *ReceivedTable);
int FrequencyTableGetTotal(FrequencyTable *ReceivedTable);
int FrequencyTableGetFrequency(int Symbol, FrequencyTable *ReceivedTable);
int FrequencyTableGetLowOn(int Symbol, FrequencyTable *ReceivedTable);
int FrequencyTableGetHighOn(int Symbol, FrequencyTable *ReceivedTable);

#endif
```

```
#include "FrequencyTable.h"

//#########################################################
FrequencyTable* FrequencyTableCreate(){
    int i;
    FrequencyTable *NewFrequencyTable;

    NewFrequencyTable = (FrequencyTable*) malloc(sizeof(FrequencyTable));
    for (i = 0; i < 257; i++)
        NewFrequencyTable->Frequences[i] = 1;

    NewFrequencyTable->Total = 257;

    return NewFrequencyTable;
}
//#########################################################
```

```
//---------------BEGIN OF CHECKING FUNCTIONS---------------

//############################################################
int FrequencyTableGetSymbolLimit(){

   return 257;
}
//############################################################

//############################################################
void FrequencyTableCheckSum(long X, long Y){
   long Z;

   Z = X + Y;
   if ((Y > 0 && Z < X) || (Y < 0 && Z > X)){
      printf("Overflow detected on 'FrequencyTableCheckSum'!");
      exit(1);
   }
}
//############################################################

//############################################################
void FrequencyTableCheckSymbolInRange(int Symbol, FrequencyTable
    *ReceivedTable){

   if (ReceivedTable == NULL){
      printf("\nFrequency table is NULL on
         'FrequencyTableCheckSymbolInRange'!");
      exit(1);
   }
   if ((Symbol < 0) || (Symbol >= FrequencyTableGetSymbolLimit())){
      printf("\nIllegal symbol '%d' received in
         'FrequencyTableCheckSymbolInRange' (range: 0 ~ 256)!", Symbol);
      exit(1);
   }
}
//############################################################

//----------------END OF CHECKING FUNCTIONS----------------


//---------------BEGIN OF MANAGEMENT FUNCTIONS---------------

//############################################################
int FrequencyTableAccumulateFrequencyTill(int Symbol, FrequencyTable
    *ReceivedTable){
   int i, Accumulated;

   Accumulated = 0;
   for (i = 0; i < Symbol; i++)
      Accumulated += ReceivedTable->Frequences[i];

   return Accumulated;
}
//############################################################

//---------------END OF MANAGEMENT FUNCTIONS---------------
```

```
//---------------BEGIN OF OPERATIONAL FUNCTIONS---------------

//############################################################
void FrequencyTableInsertOn(int Symbol, FrequencyTable *ReceivedTable){
   FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
   ReceivedTable->Frequences[Symbol]++;
   ReceivedTable->Total++;
}
//############################################################

//############################################################
int FrequencyTableGetTotal(FrequencyTable *ReceivedTable){

   if (ReceivedTable == NULL){
      printf("\nFrequency table is NULL on 'FrequencyTableGetTotal'!");
      exit(1);
   }
   if (ReceivedTable->Total < 0){
      printf("\nIllegal value to frequency table received in
         'FrequencyTableGetTotal'!");
      exit(1);
   }
   return ReceivedTable->Total;
}
//############################################################

//############################################################
int FrequencyTableGetFrequency(int Symbol, FrequencyTable *ReceivedTable){

   FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
   if (ReceivedTable->Frequences[Symbol] < 0){
      printf("\nIllegal value to symbol '%d' frequency received in
         'FrequencyTableGetFrequency'!", Symbol);
      exit(1);
   }
   return ReceivedTable->Frequences[Symbol];
}
//############################################################

//############################################################
int FrequencyTableGetLowOn(int Symbol, FrequencyTable *ReceivedTable){
   int Low, High, Total;

   FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
   Low = FrequencyTableAccumulateFrequencyTill(Symbol, ReceivedTable);
   High = Low + ReceivedTable->Frequences[Symbol];
   Total = ReceivedTable->Total;

   if ((0 > Low) || (Low > High) || (High > Total)){
      printf("\nIllegal value to low or high on table received in
         'FrequencyTableGetLowOn'!");
      exit(1);
   }
   return Low;
}
//############################################################
```

```
//############################################################
int FrequencyTableGetHighOn(int Symbol, FrequencyTable *ReceivedTable){
   int Low, High, Total;

   FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
   Low = FrequencyTableAccumulateFrequencyTill(Symbol, ReceivedTable);
   High = Low + ReceivedTable->Frequences[Symbol];
   Total = ReceivedTable->Total;

    if ((0 > Low) || (Low > High) || (High > Total)){
      printf("\nIllegal value to low or high on table received in
          'FrequencyTableGetHighOn'!");
      exit(1);
   }
   return High;
}
//############################################################

//---------------END OF OPERATIONAL FUNCTIONS---------------
#include "FrequencyTable.h"

//############################################################
FrequencyTable* FrequencyTableCreate(){
   int i;
   FrequencyTable *NewFrequencyTable;

   NewFrequencyTable = (FrequencyTable*) malloc(sizeof(FrequencyTable));
   for (i = 0; i < 257; i++)
      NewFrequencyTable->Frequences[i] = 1;

   NewFrequencyTable->Total = 257;

   return NewFrequencyTable;
}
//############################################################


//---------------BEGIN OF CHECKING FUNCTIONS---------------

//############################################################
int FrequencyTableGetSymbolLimit(){

   return 257;
}
//############################################################

//############################################################
void FrequencyTableCheckSum(long X, long Y){
   long Z;

   Z = X + Y;
   if ((Y > 0 && Z < X) || (Y < 0 && Z > X)){
      printf("Overflow detected on 'FrequencyTableCheckSum'!");
      exit(1);
   }
}
//############################################################
```

```
//#############################################################
void FrequencyTableCheckSymbolInRange(int Symbol, FrequencyTable
    *ReceivedTable){

   if (ReceivedTable == NULL){
      printf("\nFrequency table is NULL on
         'FrequencyTableCheckSymbolInRange'!");
      exit(1);
   }
   if ((Symbol < 0) || (Symbol >= FrequencyTableGetSymbolLimit())){
      printf("\nIllegal symbol '%d' received in
         'FrequencyTableCheckSymbolInRange' (range: 0 ~ 256)!", Symbol);
      exit(1);
   }
}
//#############################################################

//----------------END OF CHECKING FUNCTIONS----------------


//---------------BEGIN OF MANAGEMENT FUNCTIONS---------------

//#############################################################
int FrequencyTableAccumulateFrequencyTill(int Symbol, FrequencyTable
    *ReceivedTable){
   int i, Accumulated;

   Accumulated = 0;
   for (i = 0; i < Symbol; i++)
      Accumulated += ReceivedTable->Frequences[i];

   return Accumulated;
}
//#############################################################

//---------------END OF MANAGEMENT FUNCTIONS---------------


//---------------BEGIN OF OPERATIONAL FUNCTIONS---------------

//#############################################################
void FrequencyTableInsertOn(int Symbol, FrequencyTable *ReceivedTable){
   FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
   ReceivedTable->Frequences[Symbol]++;
   ReceivedTable->Total++;
}
//#############################################################

//#############################################################
int FrequencyTableGetTotal(FrequencyTable *ReceivedTable){

   if (ReceivedTable == NULL){
      printf("\nFrequency table is NULL on 'FrequencyTableGetTotal'!");
      exit(1);
   }
   if (ReceivedTable->Total < 0){
      printf("\nIllegal value to frequency table received in
```

```
            'FrequencyTableGetTotal'!");
         exit(1);
      }
      return ReceivedTable->Total;
}
//############################################################

//############################################################
int FrequencyTableGetFrequency(int Symbol, FrequencyTable *ReceivedTable){

      FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
      if (ReceivedTable->Frequences[Symbol] < 0){
         printf("\nIllegal value to symbol '%d' frequency received in
            'FrequencyTableGetFrequency'!", Symbol);
         exit(1);
      }
      return ReceivedTable->Frequences[Symbol];
}
//############################################################

//############################################################
int FrequencyTableGetLowOn(int Symbol, FrequencyTable *ReceivedTable){
      int Low, High, Total;

      FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
      Low = FrequencyTableAccumulateFrequencyTill(Symbol, ReceivedTable);
      High = Low + ReceivedTable->Frequences[Symbol];
      Total = ReceivedTable->Total;

      if ((0 > Low) || (Low > High) || (High > Total)){
         printf("\nIllegal value to low or high on table received in
            'FrequencyTableGetLowOn'!");
         exit(1);
      }
      return Low;
}
//############################################################

//############################################################
int FrequencyTableGetHighOn(int Symbol, FrequencyTable *ReceivedTable){
      int Low, High, Total;

      FrequencyTableCheckSymbolInRange(Symbol, ReceivedTable);
      Low = FrequencyTableAccumulateFrequencyTill(Symbol, ReceivedTable);
      High = Low + ReceivedTable->Frequences[Symbol];
      Total = ReceivedTable->Total;

       if ((0 > Low) || (Low > High) || (High > Total)){
         printf("\nIllegal value to low or high on table received in
            'FrequencyTableGetHighOn'!");
         exit(1);
      }
      return High;
}
//############################################################

//---------------END OF OPERATIONAL FUNCTIONS---------------
```

B.1.4 Encoder

```
#if !defined(ENCODER_H)
#define ENCODER_H

#include "FrequencyTable.h"
#include "BitOutputStream.h"
#include <limits.h>

typedef struct Encoder{
   long long ManyBitsForLowHigh;
   long long FullOneMask;
   long long MaxBitZeroFullOneMask;
   long long MaxBitOneMask;
   long long RightMaxBitOneMask;
   long long MaxRange;
   long long MinRange;
   long long MaxTotal;

   long long Low;
   long long High;

   int UnderflowShiftBitsSaver;

   BitOutputStream *MyBitOutputStream;
}Encoder;

//CONSTRUCTORS
void EncoderInitiateConstant(Encoder *NewEncoder);
void EncoderResetLowHigh(Encoder *NewEncoder);
Encoder* EncoderCreate(char *OutputFileName);

//MANAGEMENT (FOR ADAPTEDENCODER)
void EncoderUnderflowBitsReview(Encoder *MyEncoder);
void EncoderSetOutput(Encoder *MyEncoder);

//OPERATIONAL
void EncoderUpdateLowHigh(Encoder *MyEncoder, FrequencyTable
    *MyFrequencyTable, int Symbol);
void EncoderFinish(Encoder *MyEncoder);

#endif
```

```
#include "Encoder.h"

//#############################################################
void EncoderInitiateConstant(Encoder *NewEncoder){
   long long MaxLongLongValue = LLONG_MAX;

   NewEncoder->ManyBitsForLowHigh = 32;
   NewEncoder->FullOneMask = ((long long)1 <<
       NewEncoder->ManyBitsForLowHigh)-1;
   NewEncoder->MaxBitZeroFullOneMask = (((long long)1 <<
       (NewEncoder->ManyBitsForLowHigh-1))-1);
   NewEncoder->MaxBitOneMask = ((long long)1 <<
       (NewEncoder->ManyBitsForLowHigh-1));
```

```
   NewEncoder->RightMaxBitOneMask = ((long long)1 <<
       (NewEncoder->ManyBitsForLowHigh-2));
   NewEncoder->MaxRange = ((long long)1 << NewEncoder->ManyBitsForLowHigh);
   NewEncoder->MinRange = ((long long)1 << (NewEncoder->ManyBitsForLowHigh
       - 2))+2;

   if ((MaxLongLongValue/NewEncoder->MaxRange) < NewEncoder->MinRange)
      NewEncoder->MaxTotal = MaxLongLongValue/NewEncoder->MaxRange;
   else
      NewEncoder->MaxTotal = NewEncoder->MinRange;
}
//############################################################

//############################################################
void EncoderResetLowHigh(Encoder *NewEncoder){
   NewEncoder->Low = 0;
   NewEncoder->High = NewEncoder->FullOneMask;
   NewEncoder->UnderflowShiftBitsSaver = 0;
}
//############################################################

//############################################################
Encoder* EncoderCreate(char *OutputFileName){
   Encoder *NewEncoder = (Encoder*) malloc(sizeof(Encoder));

   EncoderInitiateConstant(NewEncoder);
   EncoderResetLowHigh(NewEncoder);
   NewEncoder->MyBitOutputStream = BitOutputStreamCreate(OutputFileName);

   return NewEncoder;
}
//############################################################


//---------------BEGIN OF MANAGEMENT FUNCTIONS---------------

//############################################################
void EncoderUnderflowBitsReview(Encoder *MyEncoder){
   if(MyEncoder->UnderflowShiftBitsSaver == INT_MAX){
      printf("\nUnderflow bits shifter reached maximum value on
          'EncoderUnderflowBitsReview'!");
      exit(2);
   }

   MyEncoder->UnderflowShiftBitsSaver++;
}
//############################################################

//############################################################
void EncoderSetOutput(Encoder *MyEncoder){
   int Index = MyEncoder->UnderflowShiftBitsSaver;
   int BitTaker = (int)(MyEncoder->Low >>
       (MyEncoder->ManyBitsForLowHigh-1));

   BitOutputStreamWriteBit(MyEncoder->MyBitOutputStream, BitTaker);
   for (; Index > 0; Index--)
      BitOutputStreamWriteBit(MyEncoder->MyBitOutputStream, (BitTaker ^
          1));
```

```
   MyEncoder->UnderflowShiftBitsSaver = 0;
}
//#########################################################

//----------------END OF MANAGEMENT FUNCTIONS---------------


//---------------BEGIN OF OPERATIONAL FUNCTIONS--------------

//#########################################################
void EncoderUpdateLowHigh(Encoder *MyEncoder, FrequencyTable
    *MyFrequencyTable, int Symbol){
   long long Range = MyEncoder->High - MyEncoder->Low + 1;
   long long LastLow = MyEncoder->Low;

   if ((MyEncoder->Low >= MyEncoder->High) || ((MyEncoder->Low &
      MyEncoder->FullOneMask) != MyEncoder->Low) || ((MyEncoder->High &
      MyEncoder->FullOneMask) != MyEncoder->High)){
      printf("\nLow (%lld) or High (%lld) value out of normal range
         checked on 'EncoderUpdateLowHigh'!");
      exit(2);
   }
   if ((Range < MyEncoder->MinRange) || (Range > MyEncoder->MaxRange)){
      printf("\nCalculated range out of expected ranges checked on
         'EncoderUpdateLowHigh'!");
      exit(2);
   }

   long long FrequencyTableTotal =
      FrequencyTableGetTotal(MyFrequencyTable);
   long long FrequencyTableLow = FrequencyTableGetLowOn(Symbol,
      MyFrequencyTable);
   long long FrequencyTableHigh = FrequencyTableGetHighOn(Symbol,
      MyFrequencyTable);

   if (FrequencyTableLow == FrequencyTableHigh){
      printf("\nSymbol has zero frequency checked on
         'EncoderUpdateLowHigh'!");
      exit(2);
   }
   if (FrequencyTableTotal > MyEncoder->MaxTotal){
      printf("\nSymbol amount overflow checked on
         'EncoderUpdateLowHigh'!");
      exit(2);
   }

   MyEncoder->Low = LastLow + FrequencyTableLow * Range /
      FrequencyTableTotal;
   MyEncoder->High = LastLow + FrequencyTableHigh * Range /
      FrequencyTableTotal - 1;

   while (((MyEncoder->Low ^ MyEncoder->High) & MyEncoder->MaxBitOneMask)
      == 0) {

      EncoderSetOutput(MyEncoder);

      MyEncoder->Low = (MyEncoder->Low << 1) & MyEncoder->FullOneMask;
```

```
      MyEncoder->High = ((MyEncoder->High << 1) & MyEncoder->FullOneMask)
          | 1;
   }

   while ((MyEncoder->Low & ~MyEncoder->High &
       MyEncoder->RightMaxBitOneMask) != 0) {

      EncoderUnderflowBitsReview(MyEncoder);

      MyEncoder->Low = (MyEncoder->Low << 1) &
          (MyEncoder->MaxBitZeroFullOneMask);
      MyEncoder->High = ((MyEncoder->High << 1) &
          (MyEncoder->MaxBitZeroFullOneMask)) | MyEncoder->MaxBitOneMask |
          1;
   }
}
//############################################################

//############################################################
void EncoderFinish(Encoder *MyEncoder){
   BitOutputStreamCloseStream(MyEncoder->MyBitOutputStream);
   free(MyEncoder->MyBitOutputStream);
}
//############################################################

//----------------END OF OPERATIONAL FUNCTIONS---------------
#include "Encoder.h"

//############################################################
void EncoderInitiateConstant(Encoder *NewEncoder){
   long long MaxLongLongValue = LLONG_MAX;

   NewEncoder->ManyBitsForLowHigh = 32;
   NewEncoder->FullOneMask = ((long long)1 <<
       NewEncoder->ManyBitsForLowHigh)-1;
   NewEncoder->MaxBitZeroFullOneMask = (((long long)1 <<
       (NewEncoder->ManyBitsForLowHigh-1))-1);
   NewEncoder->MaxBitOneMask = ((long long)1 <<
       (NewEncoder->ManyBitsForLowHigh-1));
   NewEncoder->RightMaxBitOneMask = ((long long)1 <<
       (NewEncoder->ManyBitsForLowHigh-2));
   NewEncoder->MaxRange = ((long long)1 << NewEncoder->ManyBitsForLowHigh);
   NewEncoder->MinRange = ((long long)1 << (NewEncoder->ManyBitsForLowHigh
       - 2))+2;

   if ((MaxLongLongValue/NewEncoder->MaxRange) < NewEncoder->MinRange)
      NewEncoder->MaxTotal = MaxLongLongValue/NewEncoder->MaxRange;
   else
      NewEncoder->MaxTotal = NewEncoder->MinRange;
}
//############################################################

//############################################################
void EncoderResetLowHigh(Encoder *NewEncoder){
   NewEncoder->Low = 0;
   NewEncoder->High = NewEncoder->FullOneMask;
   NewEncoder->UnderflowShiftBitsSaver = 0;
}
```

```
//###########################################################

//###########################################################
Encoder* EncoderCreate(char *OutputFileName){
   Encoder *NewEncoder = (Encoder*) malloc(sizeof(Encoder));

   EncoderInitiateConstant(NewEncoder);
   EncoderResetLowHigh(NewEncoder);
   NewEncoder->MyBitOutputStream = BitOutputStreamCreate(OutputFileName);

   return NewEncoder;
}
//###########################################################


//---------------BEGIN OF MANAGEMENT FUNCTIONS---------------

//###########################################################
void EncoderUnderflowBitsReview(Encoder *MyEncoder){
   if(MyEncoder->UnderflowShiftBitsSaver == INT_MAX){
      printf("\nUnderflow bits shifter reached maximum value on
         'EncoderUnderflowBitsReview'!");
      exit(2);
   }

   MyEncoder->UnderflowShiftBitsSaver++;
}
//###########################################################

//###########################################################
void EncoderSetOutput(Encoder *MyEncoder){
   int Index = MyEncoder->UnderflowShiftBitsSaver;
   int BitTaker = (int)(MyEncoder->Low >>
       (MyEncoder->ManyBitsForLowHigh-1));

   BitOutputStreamWriteBit(MyEncoder->MyBitOutputStream, BitTaker);
   for (; Index > 0; Index--)
      BitOutputStreamWriteBit(MyEncoder->MyBitOutputStream, (BitTaker ^
         1));

   MyEncoder->UnderflowShiftBitsSaver = 0;
}
//###########################################################

//----------------END OF MANAGEMENT FUNCTIONS----------------


//---------------BEGIN OF OPERATIONAL FUNCTIONS---------------

//###########################################################
void EncoderUpdateLowHigh(Encoder *MyEncoder, FrequencyTable
    *MyFrequencyTable, int Symbol){
   long long Range = MyEncoder->High - MyEncoder->Low + 1;
   long long LastLow = MyEncoder->Low;

   if ((MyEncoder->Low >= MyEncoder->High) || ((MyEncoder->Low &
      MyEncoder->FullOneMask) != MyEncoder->Low) || ((MyEncoder->High &
      MyEncoder->FullOneMask) != MyEncoder->High)){
```

```
        printf("\nLow (%lld) or High (%lld) value out of normal range
            checked on 'EncoderUpdateLowHigh'!");
        exit(2);
    }
    if ((Range < MyEncoder->MinRange) || (Range > MyEncoder->MaxRange)){
        printf("\nCalculated range out of expected ranges checked on
            'EncoderUpdateLowHigh'!");
        exit(2);
    }

    long long FrequencyTableTotal =
        FrequencyTableGetTotal(MyFrequencyTable);
    long long FrequencyTableLow = FrequencyTableGetLowOn(Symbol,
        MyFrequencyTable);
    long long FrequencyTableHigh = FrequencyTableGetHighOn(Symbol,
        MyFrequencyTable);

    if (FrequencyTableLow == FrequencyTableHigh){
        printf("\nSymbol has zero frequency checked on
            'EncoderUpdateLowHigh'!");
        exit(2);
    }
    if (FrequencyTableTotal > MyEncoder->MaxTotal){
        printf("\nSymbol amount overflow checked on
            'EncoderUpdateLowHigh'!");
        exit(2);
    }

    MyEncoder->Low = LastLow + FrequencyTableLow * Range /
        FrequencyTableTotal;
    MyEncoder->High = LastLow + FrequencyTableHigh * Range /
        FrequencyTableTotal - 1;

    while (((MyEncoder->Low ^ MyEncoder->High) & MyEncoder->MaxBitOneMask)
        == 0) {

        EncoderSetOutput(MyEncoder);

        MyEncoder->Low = (MyEncoder->Low << 1) & MyEncoder->FullOneMask;
        MyEncoder->High = ((MyEncoder->High << 1) & MyEncoder->FullOneMask)
            | 1;
    }

    while ((MyEncoder->Low & ~MyEncoder->High &
        MyEncoder->RightMaxBitOneMask) != 0) {

        EncoderUnderflowBitsReview(MyEncoder);

        MyEncoder->Low = (MyEncoder->Low << 1) &
            (MyEncoder->MaxBitZeroFullOneMask);
        MyEncoder->High = ((MyEncoder->High << 1) &
            (MyEncoder->MaxBitZeroFullOneMask)) | MyEncoder->MaxBitOneMask |
            1;
    }
}
//#########################################################

//#########################################################
```

```
void EncoderFinish(Encoder *MyEncoder){
   BitOutputStreamCloseStream(MyEncoder->MyBitOutputStream);
   free(MyEncoder->MyBitOutputStream);
}
//#########################################################

//----------------END OF OPERATIONAL FUNCTIONS---------------
```

B.1.5 Decoder

```
#if !defined(DECODER_H)
#define DECODER_H

#include "BitInputStream.h"
#include "Encoder.h"

typedef struct Decoder {
   long long ManyBitsForLowHigh;
   long long FullOneMask;
   long long MaxBitZeroFullOneMask;
   long long MaxBitOneMask;
   long long RightMaxBitOneMask;
   long long MaxRange;
   long long MinRange;
   long long MaxTotal;

   long long Low;
   long long High;

   long long Code;

   BitInputStream *MyBitInputStream;
} Decoder;

//PRIVATE CONSTRUCTOR (FOR ADAPTEDDECODER)
void DecoderInitiateConstant(Decoder *NewDecoder);
void DecoderResetLowHigh(Decoder *NewDecoder);
void DecoderStartCode(Decoder *NewDecoder);

//CONSTRUCTOR
Decoder* DecoderCreate(char *InputFileName);

//MANAGEMENT (FOR ADAPTEDDECODER)
int GetNextBit(Decoder *MyDecoder);
void DecoderUnderflowBitsReview(Decoder *MyDecoder);
void DecoderGetInput(Decoder *MyDecoder);

//OPERATIONAL
int DecoderGetByte(Decoder *MyDecoder, FrequencyTable *MyFrequencyTable);
void DecoderFinish(Decoder *MyDecoder);

#endif
```

```
#include "Decoder.h"
```

```
//##########################################################
void DecoderInitiateConstant(Decoder *NewDecoder){
   long long MaxLongLongValue = LLONG_MAX;

   NewDecoder->ManyBitsForLowHigh = 32;
   NewDecoder->FullOneMask = ((long long)1 <<
      NewDecoder->ManyBitsForLowHigh)-1;
   NewDecoder->MaxBitZeroFullOneMask = (((long long)1 <<
      (NewDecoder->ManyBitsForLowHigh-1))-1);
   NewDecoder->MaxBitOneMask = ((long long)1 <<
      (NewDecoder->ManyBitsForLowHigh-1));
   NewDecoder->RightMaxBitOneMask = ((long long)1 <<
      (NewDecoder->ManyBitsForLowHigh-2));
   NewDecoder->MaxRange = ((long long)1 << NewDecoder->ManyBitsForLowHigh);
   NewDecoder->MinRange = ((long long)1 << (NewDecoder->ManyBitsForLowHigh
      - 2))+2;

   if ((MaxLongLongValue/NewDecoder->MaxRange) < NewDecoder->MinRange)
      NewDecoder->MaxTotal = MaxLongLongValue/NewDecoder->MaxRange;
   else
      NewDecoder->MaxTotal = NewDecoder->MinRange;
}
//##########################################################

//##########################################################
void DecoderResetLowHigh(Decoder *NewDecoder){
   NewDecoder->Low = 0;
   NewDecoder->High = NewDecoder->FullOneMask;
}
//##########################################################

//##########################################################
void DecoderStartCode(Decoder *NewDecoder){
   int Index;

   NewDecoder->Code = 0;
   for(Index = 0; Index < NewDecoder->ManyBitsForLowHigh; Index++)
      NewDecoder->Code = (NewDecoder->Code << 1) |
         BitInputStreamNextBit(NewDecoder->MyBitInputStream);
}
//##########################################################

//##########################################################
Decoder* DecoderCreate(char *InputFileName){
   Decoder *NewDecoder = (Decoder*) malloc(sizeof(Decoder));

   NewDecoder->MyBitInputStream = BitInputStreamCreate(InputFileName);
   DecoderInitiateConstant(NewDecoder);
   DecoderResetLowHigh(NewDecoder);
   DecoderStartCode(NewDecoder);

   return NewDecoder;
}
//##########################################################

//---------------BEGIN OF MANAGEMENT FUNCTIONS--------------

//##########################################################
```

```
int GetNextBit(Decoder *MyDecoder){
   int NextBit;

   NextBit = BitInputStreamNextBit(MyDecoder->MyBitInputStream);
   if (NextBit != -1)
      return NextBit;
   else
      return 0;
}
//#########################################################

//#########################################################
void DecoderUnderflowBitsReview(Decoder *MyDecoder){
   MyDecoder->Code = (MyDecoder->Code & MyDecoder->MaxBitOneMask) |
       ((MyDecoder->Code << 1) & (MyDecoder->FullOneMask >> 1)) |
       GetNextBit(MyDecoder);
}
//#########################################################

//#########################################################
void DecoderGetInput(Decoder *MyDecoder){
   MyDecoder->Code = ((MyDecoder->Code << 1) & MyDecoder->FullOneMask) |
       GetNextBit(MyDecoder);
}
//#########################################################

//#########################################################
void DecoderUpdateLowHigh(Decoder *MyDecoder, FrequencyTable
    *MyFrequencyTable, int Symbol){
   long long Range = MyDecoder->High - MyDecoder->Low + 1;
   long long LastLow = MyDecoder->Low;

   if ((MyDecoder->Low >= MyDecoder->High) || ((MyDecoder->Low &
       MyDecoder->FullOneMask) != MyDecoder->Low) || ((MyDecoder->High &
       MyDecoder->FullOneMask) != MyDecoder->High)){
      printf("\nLow (%lld) or High (%lld) value out of normal range
          checked on 'DecoderUpdateLowHigh'!");
      exit(2);
   }
   if ((Range < MyDecoder->MinRange) || (Range > MyDecoder->MaxRange)){
      printf("\nCalculated range out of expected ranges checked on
          'DecoderUpdateLowHigh'!");
      exit(2);
   }

   long FrequencyTableTotal = FrequencyTableGetTotal(MyFrequencyTable);
   long FrequencyTableLow = FrequencyTableGetLowOn(Symbol,
       MyFrequencyTable);
   long FrequencyTableHigh = FrequencyTableGetHighOn(Symbol,
       MyFrequencyTable);

   if (FrequencyTableLow == FrequencyTableHigh){
      printf("\nSymbol has zero frequency checked on
          'DecoderUpdateLowHigh'!");
      exit(2);
   }
   if (FrequencyTableTotal > MyDecoder->MaxTotal){
      printf("\nSymbol amount overflow checked on
```

```
            'DecoderUpdateLowHigh'!");
        exit(2);
    }

    MyDecoder->Low = LastLow + FrequencyTableLow * Range /
        FrequencyTableTotal;
    MyDecoder->High = LastLow + FrequencyTableHigh * Range /
        FrequencyTableTotal - 1;

    while (((MyDecoder->Low ^ MyDecoder->High) & MyDecoder->MaxBitOneMask)
        == 0) {

        DecoderGetInput(MyDecoder);

        MyDecoder->Low = (MyDecoder->Low << 1) & MyDecoder->FullOneMask;
        MyDecoder->High = ((MyDecoder->High << 1) & MyDecoder->FullOneMask)
            | 1;
    }

    while ((MyDecoder->Low & ~MyDecoder->High &
        MyDecoder->RightMaxBitOneMask) != 0) {

        DecoderUnderflowBitsReview(MyDecoder);

        MyDecoder->Low = (MyDecoder->Low << 1) &
            (MyDecoder->MaxBitZeroFullOneMask);
        MyDecoder->High = ((MyDecoder->High << 1) &
            (MyDecoder->MaxBitZeroFullOneMask)) | MyDecoder->MaxBitOneMask |
            1;
    }
}
//###########################################################

//----------------END OF MANAGEMENT FUNCTIONS----------------

//---------------BEGIN OF OPERATIONAL FUNCTIONS---------------

//###########################################################
int DecoderGetByte(Decoder *MyDecoder, FrequencyTable *MyFrequencyTable){
    long long TotalSaver = MyFrequencyTable->Total;

    if (TotalSaver > MyDecoder->MaxTotal){
        printf("\nCan not decode because total is out of range on
            'DecodeGetByte'!");
        exit(2);
    }

    long long Range = MyDecoder->High - MyDecoder->Low + 1;
    long long Offset = MyDecoder->Code - MyDecoder->Low;
    long long ValueToCheck = ((Offset + 1) * TotalSaver - 1) / Range;

    if ((ValueToCheck * Range / TotalSaver > Offset) || (ValueToCheck < 0)
        || (ValueToCheck >= TotalSaver)){
        printf("\nCan not decode because value to check is out of range or
            is invalid on 'DecodeGetByte'!");
        exit(2);
    }
```

```
    int CheckGapStart = 0;
    int CheckGapEnd = FrequencyTableGetSymbolLimit();
    int CheckGapMiddle;

    while (CheckGapEnd - CheckGapStart > 1){
        CheckGapMiddle = (CheckGapEnd + CheckGapStart) >> 1;

        if (FrequencyTableGetLowOn(CheckGapMiddle, MyFrequencyTable) >
            ValueToCheck)
            CheckGapEnd = CheckGapMiddle;
        else
            CheckGapStart = CheckGapMiddle;
    }

    if (CheckGapStart == CheckGapEnd){
        printf("\nCan not decode because a byte has not been recognized on
            'DecodeGetByte'!");
        exit(2);
    }

    int DecodedSymbol = CheckGapStart;

    if ((FrequencyTableGetLowOn(DecodedSymbol, MyFrequencyTable) * Range /
        TotalSaver > Offset) || (FrequencyTableGetHighOn(DecodedSymbol,
        MyFrequencyTable) * Range / TotalSaver <= Offset)){
        printf("\nCan not decode because a byte has not been recognized on
            'DecodeGetByte'!");
        exit(2);
    }

    DecoderUpdateLowHigh(MyDecoder, MyFrequencyTable, DecodedSymbol);

    if ((MyDecoder->Code < MyDecoder->Low) || (MyDecoder->Code >
        MyDecoder->High)){
        printf("\nGenerated code is out of range on 'DecodeGetByte'!");
        exit(2);
    }

    return DecodedSymbol;
}
//############################################################

//############################################################
void DecoderFinish(Decoder *MyDecoder){
    BitInputStreamCloseStream(MyDecoder->MyBitInputStream);
    free(MyDecoder->MyBitInputStream);
}
//############################################################

//----------------END OF OPERATIONAL FUNCTIONS---------------
```

B.2 LUÍSA-FS

B.2.1 NodeStructs

```
#ifndef NODESTRUCTS_H
#define NODESTRUCTS_H

#include <stdlib.h>

typedef struct FrequencyNode {
   int NodeSymbol;
   int NodePosition;
   long long NodeFrequency;

   struct ControlNode *NodeControl;
   struct ControlNode *NodeChildren;
   struct SearchNode *NodeSearcher;
   struct FrequencyNode *NodeRightSibling;
   struct FrequencyNode *NodeLeftSibling;
} FrequencyNode;

typedef struct SearchNode{
   struct FrequencyNode *NodeData;
   struct SearchNode *NodeSuffix;
   struct SearchNode *NodeNext;
} SearchNode;

typedef struct ControlNode {
   int NodeTotal;

   struct FrequencyNode *NodeFrequencyChildren;
   struct SearchNode *NodeSearchChildren;
   struct SearchNode **SearchPointerChildren;
} ControlNode;

#endif
```

B.2.2 FrequencyNode

```
#ifndef FREQUENCYNODE_H
#define FREQUENCYNODE_H

#include "NodeStructs.h"

FrequencyNode* FNCreate(int NodeSymbol);
FrequencyNode* FNInsert(FrequencyNode *InsertNode, ControlNode
    *NodeControl);
FrequencyNode* FNUpdate(FrequencyNode *UpdateNode);

#endif
```

```
#define ERROR (1)
#include "FrequencyNode.h"

FrequencyNode* FNCreate(int NodeSymbol){
   FrequencyNode *NewFrequencyNode;

   NewFrequencyNode = (FrequencyNode*) malloc(sizeof(FrequencyNode));
   NewFrequencyNode->NodeSymbol = NodeSymbol;
```

```
   NewFrequencyNode->NodePosition = -1;
   NewFrequencyNode->NodeFrequency = 1;
   NewFrequencyNode->NodeChildren = (ControlNode*)
      malloc(sizeof(ControlNode));
   NewFrequencyNode->NodeChildren->NodeTotal = 0;
   NewFrequencyNode->NodeChildren->NodeFrequencyChildren = NULL;
   NewFrequencyNode->NodeChildren->NodeSearchChildren = NULL;
   NewFrequencyNode->NodeChildren->SearchPointerChildren = NULL;
   NewFrequencyNode->NodeControl = NULL;
   NewFrequencyNode->NodeSearcher = NULL;
   NewFrequencyNode->NodeLeftSibling = NULL;
   NewFrequencyNode->NodeRightSibling = NULL;

   return NewFrequencyNode;
}

FrequencyNode* FNInsert(FrequencyNode *InsertNode, ControlNode
   *NodeControl){
   FrequencyNode *Index, *SecondaryIndex;

   InsertNode->NodeControl = NodeControl;
   if (NodeControl->NodeFrequencyChildren == NULL){
      NodeControl->NodeFrequencyChildren = InsertNode;
      InsertNode->NodePosition = 1;
   }
   else{
      for (Index = NodeControl->NodeFrequencyChildren; ((Index != NULL) &&
          (Index->NodeFrequency != 1));
         Index = Index->NodeRightSibling){
         SecondaryIndex = Index;
      }

      if (Index == NULL){
         SecondaryIndex->NodeRightSibling = InsertNode;
         InsertNode->NodeLeftSibling = SecondaryIndex;
         InsertNode->NodePosition = SecondaryIndex->NodePosition + 1;
      }
      else{
         if (Index == NodeControl->NodeFrequencyChildren){
            NodeControl->NodeFrequencyChildren = InsertNode;
         }
         if (Index->NodeLeftSibling != NULL){
            Index->NodeLeftSibling->NodeRightSibling = InsertNode;
         }
         InsertNode->NodeRightSibling = Index;
         InsertNode->NodeLeftSibling = Index->NodeLeftSibling;
         Index->NodeLeftSibling = InsertNode;
         InsertNode->NodePosition = Index->NodePosition;
         for (; Index != NULL; Index = Index->NodeRightSibling){
            Index->NodePosition++;
         }
      }
   }

   return NodeControl->NodeFrequencyChildren;
}

FrequencyNode* FNUpdate(FrequencyNode *UpdateNode){
```

```
   FrequencyNode *Index;

   UpdateNode->NodeFrequency++;
   if (UpdateNode->NodeLeftSibling == NULL){
      return UpdateNode;
   }

   for(Index = UpdateNode->NodeLeftSibling; ((Index != NULL) &&
       (Index->NodeFrequency <= UpdateNode->NodeFrequency));
      Index = Index->NodeLeftSibling){
      Index->NodePosition++;
      UpdateNode->NodePosition--;
   }
   if (Index == UpdateNode->NodeLeftSibling){
      Index->NodePosition++;
      UpdateNode->NodePosition--;
      Index = Index->NodeLeftSibling;
   }

   if (UpdateNode->NodeRightSibling != NULL){
      UpdateNode->NodeRightSibling->NodeLeftSibling =
         UpdateNode->NodeLeftSibling;
   }
   UpdateNode->NodeLeftSibling->NodeRightSibling =
       UpdateNode->NodeRightSibling;

   if (Index == NULL){
      UpdateNode->NodeControl->NodeFrequencyChildren->NodeLeftSibling =
         UpdateNode;
      UpdateNode->NodeRightSibling =
         UpdateNode->NodeControl->NodeFrequencyChildren;
      UpdateNode->NodeLeftSibling = NULL;
      UpdateNode->NodeControl->NodeFrequencyChildren = UpdateNode;
   }
   else{
      UpdateNode->NodeRightSibling = Index->NodeRightSibling;
      UpdateNode->NodeLeftSibling = Index;
      Index->NodeRightSibling->NodeLeftSibling = UpdateNode;
      Index->NodeRightSibling = UpdateNode;
   }

   return UpdateNode->NodeControl->NodeFrequencyChildren;
}
```

B.2.3 SearchPointer

```
#ifndef SEARCHPOINTER_H
#define SEARCHPOINTER_H

#include "NodeStructs.h"

SearchNode* SPInsert(SearchNode *InsertNode, ControlNode *NodeControl);
SearchNode* SPSearch(int SearchSymbol, ControlNode *NodeControl);
SearchNode* SPCheck(int SearchSymbol, ControlNode *NodeControl);

#endif
```

```
#include "SearchPointer.h"

SearchNode* SPInsert(SearchNode *InsertNode, ControlNode *NodeControl){

   if (NodeControl->SearchPointerChildren == NULL){
      NodeControl->SearchPointerChildren = (SearchNode**) calloc(256,
          sizeof(SearchNode*));
   }
   NodeControl->SearchPointerChildren[InsertNode->NodeData->NodeSymbol] =
       InsertNode;

   return InsertNode;
}

SearchNode* SPSearch(int SearchSymbol, ControlNode *NodeControl){
   int Index;

   if (NodeControl->SearchPointerChildren == NULL){
      return NULL;
   }
   if (NodeControl->SearchPointerChildren[SearchSymbol] != NULL){
      return NodeControl->SearchPointerChildren[SearchSymbol];
   }
   else{
      for (Index = SearchSymbol - 1; Index >= 0; Index--){
         if (NodeControl->SearchPointerChildren[Index] != NULL){
            return NodeControl->SearchPointerChildren[Index];
         }
      }
   }

   return NULL;
}

SearchNode* SPCheck(int SearchSymbol, ControlNode *NodeControl){
   if (NodeControl->SearchPointerChildren == NULL){
      return NULL;
   }
   return NodeControl->SearchPointerChildren[SearchSymbol];
}
```

B.2.4 SearchNode

```
#ifndef SEARCHNODE_H
#define SEARCHNODE_H

#include "FrequencyNode.h"
#include "SearchPointer.h"

SearchNode* SNCreate(FrequencyNode *NodeData);
SearchNode* SNInsert(SearchNode *InsertNode, ControlNode *NodeControl);
SearchNode* SNSearch(int SearchSymbol, SearchNode *InitialNode);

#endif
```

```
#define ERROR (2)
#include "SearchNode.h"

SearchNode* SNCreate(FrequencyNode *NodeData){
   SearchNode *NewSearchNode;

   NewSearchNode = (SearchNode*) malloc(sizeof(SearchNode));
   NewSearchNode->NodeData = NodeData;
   NewSearchNode->NodeSuffix = NULL;
   NewSearchNode->NodeNext = NULL;

   return NewSearchNode;
}

SearchNode* SNInsert(SearchNode *InsertNode, ControlNode *NodeControl){
   SearchNode *Index, *Last;

   InsertNode->NodeData->NodeSearcher = InsertNode;

   if (NodeControl->NodeSearchChildren == NULL){
      NodeControl->NodeSearchChildren = InsertNode;
   }
   else{
      for(Index = NodeControl->NodeSearchChildren, Last = NULL; ((Index !=
          NULL) &&
         (Index->NodeData->NodeSymbol < InsertNode->NodeData->NodeSymbol));
            Index = Index->NodeNext){
         Last = Index;
      }
      if (Last == NULL){
         InsertNode->NodeNext = NodeControl->NodeSearchChildren;
         NodeControl->NodeSearchChildren = InsertNode;
      }
      else{
         InsertNode->NodeNext = Last->NodeNext;
         Last->NodeNext = InsertNode;
      }
   }

   return NodeControl->NodeSearchChildren;
}

SearchNode* SNSearch(int SearchSymbol, SearchNode *InitialNode){
   SearchNode *Index, *Last;

   for (Index = InitialNode, Last = NULL; ((Index != NULL) &&
       (Index->NodeData->NodeSymbol <= SearchSymbol));
        Index = Index->NodeNext){
      Last = Index;
   }

   return Last;
}
```

B.2.5 ControlNode

```
#ifndef CONTROLNODE_H
#define CONTROLNODE_H

#include <stdbool.h>
#include "SearchNode.h"

ControlNode* CNCreate();
ControlNode* CNDictionary(bool SearchPointerActivated);
SearchNode* CNSearch(bool SearchPointerActivated, int SearchSymbol,
    SearchNode *InitialNode, ControlNode *NodeControl);
SearchNode* CNInsert(bool SearchPointerActivated, int InsertSymbol,
    ControlNode *NodeControl);
SearchNode* CNAuthenticate(int SearchSymbol, int LowerSearchPointerLevel,
    int *SymbolLevel, ControlNode **NodeControls);
FrequencyNode* CNUpdate(FrequencyNode *UpdateNode);

#endif
```

```
#define ERROR (3)
#include "ControlNode.h"

ControlNode* CNCreate(){
   ControlNode *NewControlNode;

   NewControlNode = (ControlNode*) malloc(sizeof(ControlNode));
   NewControlNode->NodeTotal = 0;
   NewControlNode->NodeFrequencyChildren = NULL;
   NewControlNode->NodeSearchChildren = NULL;
   NewControlNode->SearchPointerChildren = NULL;

   return NewControlNode;
}

ControlNode* CNDictionary(bool SearchPointerActivated){
   int Index;
   ControlNode *NewDictionayNode;
   FrequencyNode *InsertFrequencyNode;
   SearchNode *InsertSearchNode;

   NewDictionayNode = CNCreate();
   for (Index = 0; Index < 256; Index++){
      InsertFrequencyNode = FNCreate(Index);
      InsertFrequencyNode->NodeFrequency = 0;
      FNInsert(InsertFrequencyNode, NewDictionayNode);
      InsertSearchNode = SNCreate(InsertFrequencyNode);
      SNInsert(InsertSearchNode, NewDictionayNode);
      if (SearchPointerActivated){
         SPInsert(InsertSearchNode, NewDictionayNode);
      }
      NewDictionayNode->NodeTotal++;
   }

   return NewDictionayNode;
}

SearchNode* CNSearch(bool SearchPointerActivated, int SearchSymbol,
    SearchNode *InitialNode, ControlNode *NodeControl){
```

```
   if (SearchPointerActivated){
      return SPCheck(SearchSymbol, NodeControl);
      /*Obs.: we used SPCheck because when the lower hash level found
         every level after will be hash too,
      if exits unconnected hash levels use SPSearch to get a suffix link.*/
   }
   else{
      if (InitialNode != NULL){
         return SNSearch(SearchSymbol, InitialNode);
      }
      else{
         return SNSearch(SearchSymbol, NodeControl->NodeSearchChildren);
      }
   }
}

SearchNode* CNInsert(bool SearchPointerActivated, int InsertSymbol,
    ControlNode *NodeControl){
   FrequencyNode *InsertFrequencyNode;
   SearchNode *InsertSearchNode;

   InsertFrequencyNode = FNCreate(InsertSymbol);
   FNInsert(InsertFrequencyNode, NodeControl);
   InsertSearchNode = SNCreate(InsertFrequencyNode);
   SNInsert(InsertSearchNode, NodeControl);
   if (SearchPointerActivated){
      SPInsert(InsertSearchNode, NodeControl);
   }
   NodeControl->NodeTotal++;

   return InsertSearchNode;
}

SearchNode* CNAuthenticate(int SearchSymbol, int LowerSearchPointerLevel,
    int *SymbolLevel, ControlNode **NodeControls){
   int Index;
   SearchNode *NodeIndex = NULL;

   if (SPCheck(SearchSymbol, NodeControls[LowerSearchPointerLevel]) !=
       NULL){
      return NULL;
   }
   else{
      for (Index = LowerSearchPointerLevel - 1; NodeIndex == NULL;
         NodeIndex = SPCheck(SearchSymbol, NodeControls[Index]), Index--);
      SymbolLevel[0] = Index + 1;
      return NodeIndex;
   }
   /*Obs.: we used this function because when the lower hash level found
      every level after will be hash too, so, if the last
   hash level has the symbol, lower levels maybe has too, else, the first
      hash level with the symbol is the search level, it
   is a pre-search function.*/
}

FrequencyNode* CNUpdate(FrequencyNode *UpdateNode){
   return FNUpdate(UpdateNode);
```

```
}
```

B.2.6 LuisaStructs

```
#ifndef LUISASTRUCTS_H
#define LUISASTRUCTS_H

#include "ControlNode.h"
#include "Encoder.h"
#include "Decoder.h"

typedef struct LuisaTree{
   int ContextsAmount;
   int HashAmount;
   ControlNode **ContextsTree;
} LuisaTree;

typedef struct LuisaCoder {
   FILE *InputFile;
   Encoder *ArithmeticEncoder;
   FrequencyTable *ArithmeticTable;
   LuisaTree *TreeControl;
} LuisaCoder;

typedef struct LuisaDecoder {
   FILE *OutputFile;
   Decoder *ArithmeticDecoder;
   FrequencyTable *ArithmeticTable;
   LuisaTree *TreeControl;
} LuisaDecoder;

#endif
```

B.2.7 LuisaTree

```
#ifndef LUISATREE_H
#define LUISATREE_H

#include "LuisaStructs.h"

LuisaTree* LTCreate(int ContextsAmount, int HashAmount);
SearchNode* LTSearch(int SearchSymbol, int *SearchLevel, LuisaTree
    *TreeControl);
void LTUpdate(int UpdateSymbol, SearchNode *NodeSymbol, LuisaTree
    *TreeControl);

#endif
```

```
#define ERROR (4)
#include "LuisaTree.h"

LuisaTree* LTCreate(int ContextsAmount, int HashAmount){
   LuisaTree *NewLuisaTree;
```

```
   NewLuisaTree = (LuisaTree*) malloc(sizeof(LuisaTree));
   NewLuisaTree->ContextsAmount = ContextsAmount;
   NewLuisaTree->HashAmount = HashAmount;
   NewLuisaTree->ContextsTree = (ControlNode**) calloc((ContextsAmount +
      1), sizeof(ControlNode*));
   if (HashAmount >= 0){
      NewLuisaTree->ContextsTree[0] = CNDictionary(true);
   }
   else{
      NewLuisaTree->ContextsTree[0] = CNDictionary(false);
   }

   return NewLuisaTree;
}

SearchNode* LTSearch(int SearchSymbol, int *SymbolLevel, LuisaTree
   *TreeControl){
   int Index;
   SearchNode *NodeIndex;

   NodeIndex = NULL;
   for (Index = TreeControl->ContextsAmount;
      TreeControl->ContextsTree[Index] == NULL; Index--);
   if (TreeControl->HashAmount >= 0){
      if (Index <= TreeControl->HashAmount){
         NodeIndex = CNAuthenticate(SearchSymbol, Index, SymbolLevel,
            TreeControl->ContextsTree);
      }
      else{
         NodeIndex = CNAuthenticate(SearchSymbol, TreeControl->HashAmount,
            SymbolLevel, TreeControl->ContextsTree);
      }
   }
   if (NodeIndex == NULL){
      for (; Index >= 0; Index--){
         if (Index > TreeControl->HashAmount){
            NodeIndex = CNSearch(false, SearchSymbol, NodeIndex,
               TreeControl->ContextsTree[Index]);
         }
         else{
            NodeIndex = CNSearch(true, SearchSymbol, NodeIndex,
               TreeControl->ContextsTree[Index]);
         }
         if ((NodeIndex != NULL) && (NodeIndex->NodeData->NodeSymbol ==
            SearchSymbol)){
            SymbolLevel[0] = Index;
            return NodeIndex;
         }
         if (NodeIndex != NULL){
            NodeIndex = NodeIndex->NodeSuffix;
         }
      }
   }

   return NodeIndex;
}
```

```
void LTUpdate(int UpdateSymbol, SearchNode *NodeSymbol, LuisaTree
    *TreeControl){
   int Index;
   SearchNode *InsertedNode, *WaitingNode, *IndexNode;

   WaitingNode = NULL;
   for (Index = TreeControl->ContextsAmount;
      TreeControl->ContextsTree[Index] == NULL; Index--);
   for (; Index >= 0; Index--){
      if (NodeSymbol->NodeData->NodeControl ==
         TreeControl->ContextsTree[Index]){
         break;
      }

      if (Index <= TreeControl->HashAmount){
         InsertedNode = CNInsert(true, UpdateSymbol,
            TreeControl->ContextsTree[Index]);
      }
      else{
         InsertedNode = CNInsert(false, UpdateSymbol,
            TreeControl->ContextsTree[Index]);
      }
      if (Index < TreeControl->ContextsAmount){
         TreeControl->ContextsTree[Index + 1] =
            InsertedNode->NodeData->NodeChildren;
      }
      if (WaitingNode != NULL){
         WaitingNode->NodeSuffix = InsertedNode;
      }
      WaitingNode = InsertedNode;
   }
   if (WaitingNode != NULL){
      WaitingNode->NodeSuffix = NodeSymbol;
   }

   for(IndexNode = NodeSymbol; Index >= 0; IndexNode =
      IndexNode->NodeSuffix, Index--){
      CNUpdate(IndexNode->NodeData);
      if (Index < TreeControl->ContextsAmount){
         TreeControl->ContextsTree[Index + 1] =
            IndexNode->NodeData->NodeChildren;
      }
   }
}
```

B.2.8 LuisaCoding

```
#ifndef LUISACODING_H
#define LUISACODING_H

#include "LuisaTree.h"

LuisaCoder* LCCreate(char *InputFileName, char *OutputFileName, int
    ContextsAmount, int HashAmount);
void LCExecute(LuisaCoder *Coder);
```

```
#endif
```

```
#define ERROR (5)
#include "LuisaCoding.h"

int LCGetCode(int CodeSymbol, SearchNode **UpdateMatch, LuisaTree
    *TreeControl){
   int MatchLevel, UpdateFactor;
   SearchNode *MatchSymbol, *Index;

   MatchSymbol = LTSearch(CodeSymbol, &MatchLevel, TreeControl);
   UpdateFactor = 0;
   if ((MatchLevel < TreeControl->ContextsAmount) &&
       (TreeControl->ContextsTree[MatchLevel+1] != NULL)){
      for (Index =
          TreeControl->ContextsTree[MatchLevel+1]->NodeSearchChildren;
          Index != NULL; Index = Index->NodeNext){
          if (Index->NodeSuffix->NodeData->NodePosition >
              MatchSymbol->NodeData->NodePosition){
             UpdateFactor++;
          }
      }
   }

   UpdateMatch[0] = MatchSymbol;

   return MatchSymbol->NodeData->NodePosition + UpdateFactor - 1;
}

void LCProcess(int CodeSymbol, LuisaCoder *MainCoder){
   int SymbolCode;
   SearchNode *UpdateMatch;

   SymbolCode = LCGetCode(CodeSymbol, &UpdateMatch,
      MainCoder->TreeControl);
   EncoderUpdateLowHigh(MainCoder->ArithmeticEncoder,
      MainCoder->ArithmeticTable, SymbolCode);
   FrequencyTableInsertOn(SymbolCode, MainCoder->ArithmeticTable);
   LTUpdate(CodeSymbol, UpdateMatch, MainCoder->TreeControl);
}

LuisaCoder* LCCreate(char *InputFileName, char *OutputFileName, int
    ContextsAmount, int HashAmount){
   LuisaCoder *NewCoder;

   NewCoder = (LuisaCoder*) malloc(sizeof(LuisaCoder));
   NewCoder->TreeControl = LTCreate(ContextsAmount, HashAmount);
   NewCoder->ArithmeticEncoder = EncoderCreate(OutputFileName);
   NewCoder->ArithmeticTable = FrequencyTableCreate();
   NewCoder->InputFile = fopen(InputFileName, "rb");
   if (NewCoder->InputFile == NULL){
      exit(ERROR);
   }
   if (HashAmount > -1){
      EncoderUpdateLowHigh(NewCoder->ArithmeticEncoder,
         NewCoder->ArithmeticTable, ContextsAmount);
      EncoderUpdateLowHigh(NewCoder->ArithmeticEncoder,
```

```
            NewCoder->ArithmeticTable, 2);
         EncoderUpdateLowHigh(NewCoder->ArithmeticEncoder,
            NewCoder->ArithmeticTable, HashAmount);
      }
      else{
         EncoderUpdateLowHigh(NewCoder->ArithmeticEncoder,
            NewCoder->ArithmeticTable, ContextsAmount);
         EncoderUpdateLowHigh(NewCoder->ArithmeticEncoder,
            NewCoder->ArithmeticTable, 1);
      }
      return NewCoder;
}

void LCExecute(LuisaCoder *MainCoder){
      unsigned char ReadBytes[500000];
      int Amount, Index;

      for (Amount = fread(ReadBytes, sizeof(char), 500000,
         MainCoder->InputFile); Amount > 0; Amount = fread(ReadBytes,
         sizeof(char), 500000, MainCoder->InputFile)){
         for (Index = 0; Index < Amount; Index++){
            LCProcess(ReadBytes[Index], MainCoder);
         }
      }
      EncoderUpdateLowHigh(MainCoder->ArithmeticEncoder,
         MainCoder->ArithmeticTable, 256);
      EncoderUpdateLowHigh(MainCoder->ArithmeticEncoder,
         MainCoder->ArithmeticTable, 256);
      EncoderFinish(MainCoder->ArithmeticEncoder);
      fclose(MainCoder->InputFile);
}
```

B.2.9 LuisaDecoding

```
#ifndef LUISADECODING_H
#define LUISADECODING_H

#include <stdbool.h>
#include "LuisaTree.h"

LuisaDecoder* LDCreate(char *InputFileName, char *OutputFileName);
void LDExecute(LuisaDecoder *MainDecoder);

#endif
```

```
#define ERROR (5)
#include "LuisaDecoding.h"

int LDGetData(int DecodeSymbol, SearchNode **UpdateMatch, LuisaTree
    *TreeControl){
   bool CheckedValues[256] = {false};
   int CodeSearch, ContextIndex;
   FrequencyNode *ValuesIndex;

   for (ContextIndex = TreeControl->ContextsAmount;
```

```
      TreeControl->ContextsTree[ContextIndex] == NULL; ContextIndex--);
   for (; ContextIndex >= 0; ContextIndex--){
      if (TreeControl->ContextsTree[ContextIndex]->NodeTotal >
         DecodeSymbol){
         break;
      }
   }

   CodeSearch = 0;
   if ((ContextIndex < TreeControl->ContextsAmount) &&
      (TreeControl->ContextsTree[ContextIndex+1] != NULL)){
      ContextIndex++;
      for (ValuesIndex =
         TreeControl->ContextsTree[ContextIndex]->NodeFrequencyChildren;
         ValuesIndex != NULL;
         ValuesIndex = ValuesIndex->NodeRightSibling){
         CheckedValues[ValuesIndex->NodeSymbol] = true;
         CodeSearch++;
      }
      ContextIndex--;
   }

   for (; ContextIndex >= 0; ContextIndex--){
      for (ValuesIndex =
         TreeControl->ContextsTree[ContextIndex]->NodeFrequencyChildren;
         ValuesIndex != NULL;
         ValuesIndex = ValuesIndex->NodeRightSibling){
         if (!CheckedValues[ValuesIndex->NodeSymbol]){
            CheckedValues[ValuesIndex->NodeSymbol] = true;
            CodeSearch++;
         }

         if (CodeSearch == DecodeSymbol + 1){
            UpdateMatch[0] = ValuesIndex->NodeSearcher;
            return ValuesIndex->NodeSymbol;
         }
      }
   }

   exit(ERROR);
}

int LDProcess(LuisaDecoder *MainDecoder){
   int DecodeSymbol, DecodedSymbol;
   SearchNode *UpdateMatch;

   DecodeSymbol = DecoderGetByte(MainDecoder->ArithmeticDecoder,
      MainDecoder->ArithmeticTable);
   if (DecodeSymbol == 256){
      return 256;
   }
   FrequencyTableInsertOn(DecodeSymbol, MainDecoder->ArithmeticTable);
   DecodedSymbol = LDGetData(DecodeSymbol, &UpdateMatch,
      MainDecoder->TreeControl);
   LTUpdate(DecodedSymbol, UpdateMatch, MainDecoder->TreeControl);

   return DecodedSymbol;
}
```

```
LuisaDecoder* LDCreate(char *InputFileName, char *OutputFileName){
   int ContextsAmount, HashAmount;
   LuisaDecoder *NewDecoder;

   NewDecoder = (LuisaDecoder*) malloc(sizeof(LuisaDecoder));
   NewDecoder->ArithmeticDecoder = DecoderCreate(InputFileName);
   NewDecoder->ArithmeticTable = FrequencyTableCreate();
   NewDecoder->OutputFile = fopen(OutputFileName, "wb+");
   if(NewDecoder->OutputFile == NULL){
      exit(ERROR);
   }
   ContextsAmount = DecoderGetByte(NewDecoder->ArithmeticDecoder,
       NewDecoder->ArithmeticTable);
   if (DecoderGetByte(NewDecoder->ArithmeticDecoder,
       NewDecoder->ArithmeticTable) == 2){
      HashAmount = DecoderGetByte(NewDecoder->ArithmeticDecoder,
          NewDecoder->ArithmeticTable);
   }
   else{
      HashAmount = -1;
   }
   NewDecoder->TreeControl = LTCreate(ContextsAmount, HashAmount);

   return NewDecoder;
}

void LDExecute(LuisaDecoder *MainDecoder){
   unsigned char ReadyBytes[500000];
   int Amount, DecodedSymbol;

   Amount = 0;
   for (DecodedSymbol = LDProcess(MainDecoder); DecodedSymbol != 256;
       DecodedSymbol = LDProcess(MainDecoder)){
      ReadyBytes[Amount] = DecodedSymbol;
      Amount++;

      if (Amount == 500000){
         fwrite(ReadyBytes, sizeof(char), Amount, MainDecoder->OutputFile);
         Amount = 0;
      }
   }
   if (Amount > 0){
      fwrite(ReadyBytes, sizeof(char), Amount, MainDecoder->OutputFile);
   }

   DecoderFinish(MainDecoder->ArithmeticDecoder);
   fclose(MainDecoder->OutputFile);
}
```

B.2.10 LuisaExecute

```
#ifndef LUISAEXECUTE_H
#define LUISAEXECUTE_H

#include <string.h>
```

```
#include "LuisaCoding.h"
#include "LuisaDecoding.h"

void HelpInformations();
void ExecuteDecoding(char *InputFileName, char *OutputFileName);
void ExecuteEncoding(char *InputFileName, char *OutputFileName, int
    ContextsAmount, int HashAmount);

#endif
```

```
#define ERROR (6)
#include "LuisaExecute.h"

void HelpInformations(){
   printf("\n================ HELP INFORMATIONS ================\n");
   printf("ENCODE: *.exe -c Input_File_Name.ext Output_File_Name.ext
       contexts_amount hash_amount\n");
   printf("DECODE: *.exe -d Input_File_Name.ext Output_File_Name.ext\n");
   printf("PS: 1 <= contexts_amount <= 256\n");
   printf("PS2: -1 <= hash_amount <= contexts_amount (-1 = no hash)\n");
   printf("===================================================\n");
}

void ExecuteDecoding(char *InputFileName, char *OutputFileName){
   LuisaDecoder *MainDecoder;

   MainDecoder = LDCreate(InputFileName, OutputFileName);
   LDExecute(MainDecoder);
}

void ExecuteEncoding(char *InputFileName, char *OutputFileName, int
    ContextsAmount, int HashAmount){
   LuisaCoder *MainCoder;

   if ((ContextsAmount > 0) && (ContextsAmount < 257) && (HashAmount <=
       ContextsAmount) && (HashAmount >= -1)){
      MainCoder = LCCreate(InputFileName, OutputFileName, ContextsAmount,
          HashAmount);
      LCExecute(MainCoder);
   }
   else{
      HelpInformations();
   }
}
```

APÊNDICE C – Artigos

A Tabela C.1 apresenta artigos publicados em periódicos e eventos sobre a otimizações e aplicações de métodos de compressão de dados.

Título	Evento/Periódico
Usando Tabelas de Dispersão com PPM	Escola Regional de Alto Desempenho
Compressão de Arquivos Orientados a Colunas com PPM	Escola Regional de Banco de Dados
Compression of Very Sparse Column Oriented Data	Communications and Innovations Gazette

Tabela C.1 – Artigos publicados

Usando Tabelas de Dispersão com PPM

Vinicius F. Garcia[1], **Sergio L. S. Mergen**[1]

[1]Universidade Federal de Santa Maria
Santa Maria – RS – Brasil

vfulber@inf.ufsm.br, mergen@inf.ufsm.br

Resumo. *Ao longo das décadas foram propostos diversos algoritmos para compressão de dados. Um dos que está entre os que obtém as melhores taxas de compressão para textos é chamado PPM (Prediction By Partial Matching). O método PPM usa informações de contexto para calcular a probabilidade de ocorrência de um símbolo. Apesar das boas taxas de compressão, esse método tem um desempenho em tempo razoavelmente inferior aos métodos de compressão usados comercialmente. Em parte essa deficiência está associada ao custo de localizar um símbolo a partir de um contexto. O objetivo desse artigo é investigar como tabelas de dispersão podem ajudar a reduzir esse custo.*

1. Introdução

Hoje em dia os métodos de compressão de arquivos mais usados são baseados em dicionário e pertencem à família de compressores LZ. De modo geral, as diversas variações desse método usam um índice de deslocamento (*offset*) para codificar uma sequência de símbolos, sendo que esse *offset* aponta para uma parte do arquivo de entrada já processada onde essa sequência pode ser encontrada. Esses codificadores levaram à especificação de um padrão para codificação (chamado DEFLATE) e serviram de base para a criação de alguns dos compressores de dados mais usados comercialmente, como gzip e zlib [Harnik et al. 2014].

Uma outra abordagem relacionada ao uso de dicionários é chamada de PPM (*Prediction by Partial Matching*). Para cada símbolo a ser codificado, o método PPM verifica a probabilidade de ocorrência desse símbolo dentro do contexto composto pelos últimos símbolos lidos [Sayood 2012]. Métodos PPM se destacam por obterem boas taxas de compressão, especialmente para arquivos texto. Essa taxa é muitas vezes superior aos resultados obtidos pelos compressores da família LZ, mas o tempo de processamento é visivelmente superior.

Parte do processamento do algoritmo de compressão PPM envolve o armazenamento e a posterior procura por símbolos dentro de algum contexto específico. Pela natureza do problema, os contextos costumam ser armazenados em árvores, e a busca é implementada como um percorrimento de listas encadeadas. Como a busca em listas encadeadas pode consumir um tempo considerável de processamento, o objetivo desse artigo é investigar o uso de tabelas de dispersão como forma de abreviar essa busca. De modo geral, a ideia é passar a usar tabelas de dispersão para localizar nós da árvore em algumas situações específicas. As próximas seções trazem mais informações sobre o funcionamento do PPM e sobre a proposta para uso de tabelas de dispersão.

2. PPM

Para compreender o funcionamento do PPM, considere o texto a comprimir indicado na Figura 1. A seta indica o próximo símbolo a ser codificado(_), e as chaves indicam o maior tamanho de contexto a ser analisado. Dado o símbolo a codificar, a codificação é determinada pela probabilidade de ocorrência desse símbolo dado o contexto que o precede.

Figura 1. Símbolos a codificar

```
CODIFICA(simbolos_lidos, simbolo)

1: contexto ← ultimos n simbolos_lidos
2: for ordem de n a 0 do
3:     no ← busca_simbolo(contexto, simbolo)
4:     if no is not NULL then
5:         codifica(no)
6:         return
7:     else
8:         codifica(escape)
9:         encurta(contexto)
10:    end if
11: end for
```

Figura 2. Pseudo-código PPM

A probabilidade depende de quais símbolos já ocorreram no passado quando esse contexto foi encontrado, e quais foram as frequências de ocorrência desses símbolos. Com os dados de frequência a saída pode ser gerada usando algum codificador de entropia, como a codificação aritmética, de modo que símbolos mais frequentes gerem menos bits durante a codificação [Howard and Vitter 2012].

O pseudo-código da Figura 2 mostra como a busca ocorre. Caso o símbolo a codificar tenha sido encontrado precedendo o contexto atual, sua frequência nesse contexto é codificada. Caso contrário, é codificada a frequência de um símbolo especial ($escape$) e a busca recomeça removendo um símbolo do contexto (ex. de ABA para BA). No pior caso o contexto é reduzido à ordem zero (contexto vazio), onde todos os símbolos possíveis existem. A descompressão segue o caminho inverso. Símbolos decodificados alimentam o contexto e servem de indícios para a decodificação do próximo símbolo.

3. Resultados e Discussão

A Figura 3 mostra a árvore de contexto que seria gerada para o texto usado como exemplo. O nível zero ($RAIZ$) é reservado para o contexto vazio. Os níveis abaixo do raiz passam a levar em consideração o contexto. Por exemplo, o nó folha mais à esquerda da árvore indica que o símbolo A apareceu duas vezes após o contexto AB.

Os círculos na árvore indicam os nós que fazem parte do contexto atual. Eles identificam os símbolos presentes em ABA, em cada uma das ordens, de zero a três. A operação que encurta o contexto descrita no pseudo-código pode ser vista como a navegação de um nó de ordem maior para um nó de ordem menor nessa árvore. A busca em um dos nós desse contexto implica em verificar se ele possui como filho o símbolo que se deseja codificar. Para evitar um consumo excessivo de memória, os filhos de um nó são usualmente representados por listas encadeadas. Desse modo, a busca equivale ao percorrimento em uma lista encadeada.

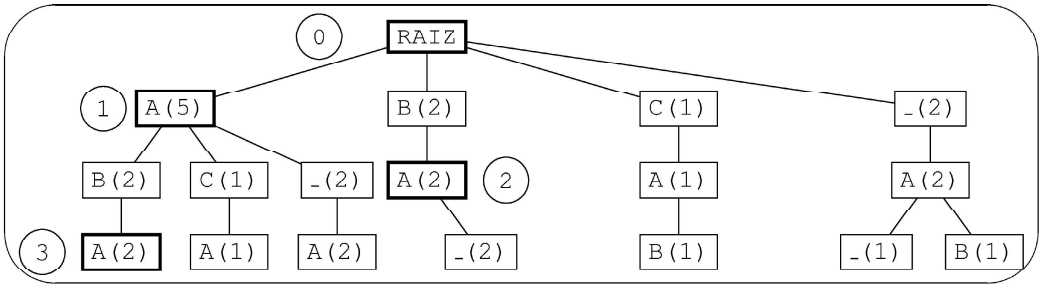

Figura 3. Árvore de Contexto (ordem = 3)

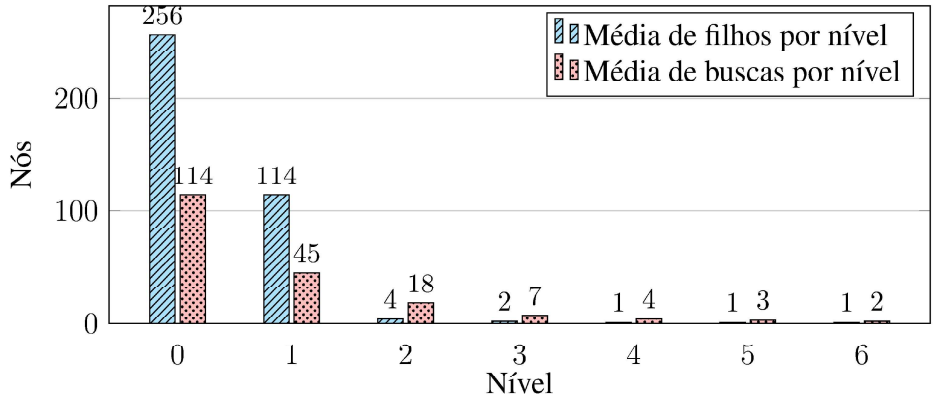

Figura 4. Estatísticas da árvore de contexto do Corpus Calgary

Para fins de ilustração, no exemplo tanto o universo de símbolos possíveis (A,B,C,_) quanto o tamanho máximo do contexto (três) são restritos. Em aplicações reais o universo de símbolos compreende todas as 256 combinações que formam um byte. Além disso, resultados empíricos com algoritmos PPM mostram que a taxa de compressão é maior quando se usa contextos de ordens cinco e seis.

O gráfico da Figura 4 mostra algumas estatísticas que corroboram a ideia sugerida nesse artigo. Os dados foram coletados a partir da árvore de contexto gerada pelo corpus Calgary, um conhecido *benchmark* para compressão de dados. O número médio de filhos nos dois primeiros níveis (que correspondem às ordens de contexto 0 e 1) é consideravelmente superior aos demais níveis. Consequentemente, a busca por um símbolo específico tem um custo maior nos dois primeiros níveis, como demonstrado no gráfico [1].

Esses resultados mostram que boa parte do tempo de processamento do algoritmo é gasto com pesquisa em nós de nível 0 e 1. Desse modo, pode-se reduzir esse tempo usando tabelas de dispersão para acessar nós que pertençam a esses níveis. Duas funções foram empregadas nas tabelas de dispersão. A função para o nível 0, chamada de $F0$, simplesmente mapeia contextos de um byte para o seu equivalente valor decimal. Já a função para o nível 1 ($F1$) mapeia contextos de dois bytes para um número de 0 a 65.536 conforme apresentado na Equação 1:

[1] o fato de haver uma média de nós visitados maior do que a média de filhos pode ocorrer quando alguns contextos específicos são mais acessados do que outros com menos filhos

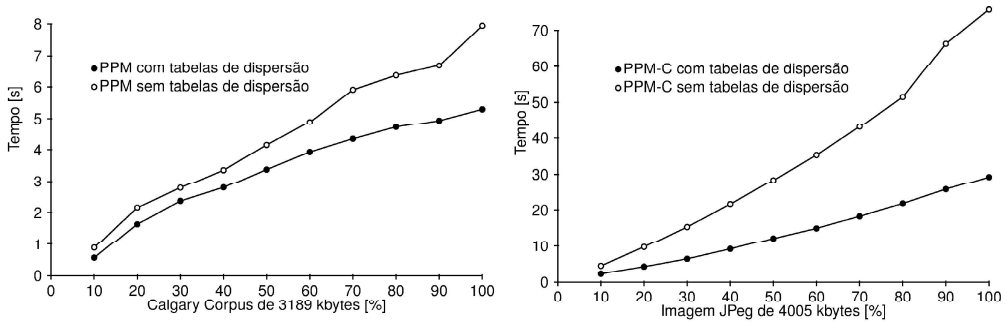

Figura 5. Tempo de Execução

$$F1(b_1, b_2) = 256 \times decimal(b_1) + decimal(b_2) \quad (1)$$

O algoritmo PPM foi desenvolvido em C e usa contextos de tamanho máximo igual a seis. As funções sugeridas e suas respectivas tabelas de dispersão foram incorporadas ao código. As tabelas das funções $F0$ e $F1$ foram traduzidas como vetores de 256 e 65.536 posições respectivamente. Em ambos os casos, cada possibilidade de entrada é mapeada para uma posição única do vetor, o que evita conflitos. Como as posições do vetor guardam ponteiros para nós, e considerando ponteiros de 4 bytes, os dois vetores ocupam 1 kb e 256 kbytes, respectivamente.

A Figura 5 apresenta o tempo total de execução para compressão de dados em um computador com processador Core I7 4500U. O tempo compreende a construção das estruturas de dados, a busca e a codificação aritmética. O gráfico da esquerda mostra que o uso de tabelas de dispersão traz ganhos de desempenho para a compressão de fatias (de 10% a 100%) do corpus Calgary. O gráfico da direita traz os resultados para a compressão de uma imagem em formato jpeg (já comprimido). Nesse caso o ganho é ainda mais evidente. O que diferencia os dois cenários (e explica os tempos de execução) é que padrões são menos comuns no arquivo de imagem, o que faz com que muito mais buscas sejam realizadas nos níveis 0 e 1.

Outros testes foram realizados. Por questão de espaço os resultados foram omitidos. De qualquer forma, percebe-se que mesmo em cenários em que buscas em contextos menores são menos comuns, o uso de tabelas de dispersão é uma estratégia útil para redução do tempo de processamento. Comprovada a eficácia dessa técnica, o próximo passo é investigar o uso de tabelas de dispersão para contextos de maior ordem. Por exemplo, se forem considerados contextos de três símbolos, o número de sequências possíveis é superior a 16 milhões. Nesse caso, o desafio é escolher funções que gerem tabelas com baixa sobrecarga de espaço e que distribua os valores uniformemente.

Referências

Harnik, D., Khaitzin, E., Sotnikov, D., and Taharlev, S. (2014). A fast implementation of deflate. In *Data Compression Conference (DCC), 2014*, pages 223–232. IEEE.

Howard, P. G. and Vitter, J. S. (2012). Arithmetic coding. *Image and Text Compression*, 176:85.

Sayood, K. (2012). *Introduction to data compression*. Newnes.

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

Compressão de Arquivos Orientados a Colunas com PPM

Vinicius F. Garcia[1], **Sergio L. S. Mergen**[1]

[1]Universidade Federal de Santa Maria
Santa Maria – RS – Brasil

vfulber@inf.ufsm.br, mergen@inf.ufsm.br

Abstract. *Column oriented databases belong to a kind of NoSQL database in which the values of the same column are stored contiguously in secondary memory. This physical organization favors compression, mainly because the proximity of data of the same nature decreases the information entropy. With respect to high cardinality columns that store text, several compression methods can be used. One of them, called PPM, is usually good in obtaining high compression rates, but the execution time is poor for conventional files. The purpose of this paper is to analyze whether this compression method is able to explore the nature of column oriented data to obtain more expressive results in comparison with is main competitors.*

Resumo. *Bancos de dados orientados a colunas pertencem a um tipo de banco NoSQL em que os valores de uma mesma coluna são armazenados contiguamente em memória secundária. Essa organização física favorece a compressão, uma vez que a aproximação dos dados de mesma natureza diminui a entropia da informação. Considerando especificamente colunas de alta cardinalidade que armazenam texto, diversos tipos de compressores podem ser usados. Um deles, chamado PPM, costuma obter boas taxas de compressão, mas possui um tempo de processamento considerado alto para arquivos convencionais. O objetivo desse artigo é verificar se esse método consegue explorar a natureza dos dados orientados a coluna de forma a obter resultados mais expressivos em relação aos seus concorrentes.*

1. Introdução

Os bancos de dados NoSQL tem recebido muita atenção recentemente. Ao contrário dos bancos de dados relacionais, essa nova vertente utiliza diferentes formas de organização de arquivos. Utilizando arquiteturas baseadas na computação em nuvem, esse tipo de solução oferece um bom escalonamento para determinados tipos de aplicações que usam padrões de acesso aos dados bem específicos.

Um dos tipos de banco NoSQL que se popularizou é conhecido como orientados a colunas (Han et al., 2011). Diferentemente dos SGBDs convencionais que armazenam registros de tabelas consecutivamente em arquivos, os sistemas orientados a colunas armazenam todos os valores de uma mesma coluna consecutivamente, possivelmente em arquivos separados, conforme ilustrado na Figura 1.

Essa forma de organização é útil em alguns cenários específicos, como por exemplo, para acelerar a execução de consultas analíticas que acessam poucas colunas, uma vez que é possível delimitar os arquivos que o processador de consultas deve varrer. Além

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

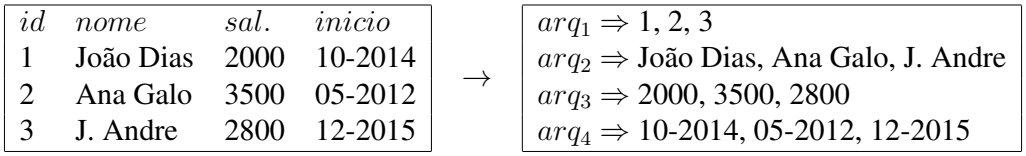

Figura 1. Orientação à registros e à colunas

disso, é possível representar de maneira mais eficiente as colunas que aceitam valores nulos, sem precisar recorrer a mapas de bits indicando campos nulos.

Outro ponto que merece destaque com relação aos bancos orientados a colunas é a sua capacidade de compressão de dados. Nota-se que com essa organização os arquivos passam a ser formados por valores que pertencem ao mesmo domínio e tipo de dados. Isso reduz a entropia da informação, e os algoritmos de compressão podem se beneficiar disso para obter uma taxa de compressão superior.

Caso uma coluna possua baixa cardinalidade (poucos valores distintos), um método de compressão bastante eficaz é a substituição do valor por um código que referencia uma entrada em um dicionário. Já para a compressão de colunas de texto de alta cardinalidade, os métodos que exploram padrões dentro do texto, como LZ (Ziv e Lempel, 1978) e BWT (Burrows e Wheeler, 1994), são mais adequados.

Outro método que encontra grande utilidade na compressão de dados textuais de alta cardinalidade é conhecido como PPM (*Prediction by Partial Matching*) (Moffat, 1990). As taxas de compressão obtidas pelas diversas variações do PPM concorrem com os resultados obtidos pelos demais compressores. O que impede o seu uso mais disseminado é o elevado tempo de processamento.

As estatísticas de desempenho dos compressores PPM são oriundas de testes realizados sobre *benchmarks* conhecidos na área de compressão de dados, como o corpus CALGARY (Council, 2008), formado por arquivos de formatos variados, como textos, imagens e códigos em linguagens de programação. No entanto, bancos de dados orientados a colunas possuem características bem distintas no que diz respeito aos padrões que podem ser encontrados.

Assim sendo, o objetivo desse artigo é investigar o comportamento do PPM na compressão de dados textuais de alta cardinalidade e analisar se seu uso é viável para arquivos orientados a colunas. A avaliação envolve a análise do tempo de execução e a taxa de compressão do PPM, comparando os resultados àqueles que são obtidos pelo LZ e BWT.

O artigo está estruturado da seguinte forma: a seção 2 apresenta resumidamente algumas das principais estratégias de compressão de dados textuais propostas na literatura. Na seção 3 o método de compressão PPM é apresentado, afim de explicar porque esse método parece especialmente adequado para dados orientados a colunas. A seção 4 apresenta experimentos que foram feitos comparando o desempenho de diversos algoritmos de compressão em cenários compostos por arquivos orientados a colunas e arquivos convencionais pertencentes ao corpus *Calgary*. A seção 5 traz as considerações finais.

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

2. Algoritmos de Compressão de dados

A compressão de dados textuais sem perda recebeu muita atenção da comunidade científica em décadas passadas. Uma das primeiras ideias exploradas foram as chamadas técnicas de codificação estatística, como a codificação de Huffman (Huffman et al., 1952) e a codificação aritmética (Witten et al., 1987). Nos dois casos a compressão é obtida representando os caracteres (ou símbolos) mais frequentes do arquivo usando menos bits.

O código de Huffman emprega uma árvore binária que mapeia símbolos como cadeias de bits. Por sua vez a codificação aritmética emprega uma tabela que guarda a frequências de ocorrências dos símbolos já processados. Sabendo essas frequência, pode-se calcular a probabilidade de ocorrência de qualquer símbolo. Essa probabilidade é então codificada como um valor binário de ponto fixo. Os codificadores de Huffman e aritméticos podem ser estáticos, quando usam árvores/tabelas de frequência predeterminadas, ou dinâmicos, quando as árvores/tabelas são construídas à medida que a compressão ocorre.

Mais tarde surgiram técnicas de compressão que passaram a levar em consideração não apenas a frequência dos símbolos, mas o fato de que muitos símbolos costumam aparecer juntos. As técnicas que exploram essa característica são conhecidas como baseadas em dicionário. Os algoritmos dessa categoria mais usados hoje em dia derivam de uma ideia proposta por Ziv e Lempel (1978), e são referenciados pelas iniciais de seus autores (LZ). De modo geral, a sequência de símbolos já processada do arquivo de entrada forma o dicionário. Uma sequência de símbolos a codificar é representada através de um índice de deslocamento e uma largura. O índice indica um ponto no arquivo de entrada onde essa sequência já foi encontrada. A largura determina quantos símbolos a partir desse índice são equivalentes aos símbolos que se deseja comprimir. Essas duas informações são representadas através de um codificador estatístico, sendo o código de Hufmann mais comumente utilizado. Essa ideia levou à especificação de um padrão para codificação chamado DEFLATE (Deutsch, 1996) e serviu de base para a criação dos compressores de dados mais usados comercialmente, como gzip e lzip.

Outra técnica que se mostrou particularmente interessante para a compressão de texto foi proposta por Burrows e Wheeler (1994). Seu nome, BWT, é um acrônimo que remete aos nomes dos autores (*Burrows Wheeler Transform*). O passo inicial do algoritmo gera todas as permutações que se obtém ao rotacionar o texto a comprimir um símbolo de cada vez. Essas permutações são armazenadas em uma matriz onde as linhas são ordenadas. No próximo passo todas as colunas com exceção da última são descartadas. É possível reconstruir o texto original usando apenas essa última coluna e um índice que localiza a linha da matriz onde o texto original estaria armazenado. Esse método parte da constatação de que alguns símbolos são normalmente precedidos por determinados símbolos. Caso isso ocorra, a última coluna da matriz será composta por muitos símbolos repetidos. Isso abre espaço para a aplicação de uma técnica chamada MTF (*Move to Front*) que visa transformar essa saída em outra composta por valores de 0 a 255 com a predominância de valores baixos. O último passo (que é onde a compressão realmente ocorre) envolve codificar essa saída composta por valores numéricos usando algum codificar estatístico como *Huffman* ou codificação aritmética.

Também existem trabalhos voltados à bancos de dados orientados a colunas que exploram colunas que possuem determinadas características (Abadi et al., 2009). Por

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

exemplo, quando é comum que os valores sejam compostos por muitos caracteres em branco consecutivos, pode-se empregar técnicas de supressão de nulos, cujo objetivo é remover um símbolo de elevada ocorrência (como o espaço em branco, por exemplo), deixando em seu lugar a sua localização e quantidade (Westmann et al., 2000). Caso mais símbolos costumem aparecer de forma consecutiva, uma técnica simples e que encontra empregabilidade em diversas aplicações é a RLE (*Run Length Encoding*), onde símbolos consecutivos repetidos são substituídos por um par composto pelo símbolo e pelo número de repetições. Essa técnica pode ser útil em sistemas orientados a colunas que guardam os valores ordenados (Abadi et al., 2006). Já o uso de dicionários e vetores de bits (Wu et al., 2002) são indicados para casos em que a quantidade de valores distintos é baixa (baixa cardinalidade). De modo geral, esses trabalhos tem objetivos ortogonais aos propostos neste artigo, cujo foco é em dados textuais de alta cardinalidade e que não necessariamente estejam ordenados.

3. PPM

O método de compressão PPM codifica um símbolo de cada vez. Para gerar um código é levado em consideração o contexto, que são os símbolos que precedem o símbolo a ser codificado. Para compreender o funcionamento do PPM, considere o texto a comprimir indicado abaixo. A seta indica o próximo símbolo a ser codificado, e as chaves indicam o contexto a ser analisado.

$$\begin{array}{cccccccccccccc} & & & & & & & & & & & & \downarrow & \\ A & C & A & B & A & \_ & A & \_ & A & \underbrace{B} & \underbrace{A} & \_ & & \\ & & & & & & & & & \multicolumn{2}{c}{Contexto} & & & \end{array}$$

Dado o símbolo a ser codificado ('_'), a codificação é determinada pela probabilidade de ocorrência desse símbolo dado o contexto que o precede. Para realizar esse cálculo é necessário analisar quais símbolos já ocorreram no passado quando esse contexto foi encontrado, e quais são as frequências de ocorrência desses símbolos. Quanto maior a frequência, maior é a probabilidade. A probabilidade pode ser computada usando codificação aritmética, de modo que símbolos mais prováveis gerem menos bits durante a codificação.

Pela Tabela 1 é possível observar quais símbolos ocorreram até então (e suas frequências) para cada ordem do contexto atual. Por exemplo, para o contexto de maior ordem ('BA') o histórico mostra que apenas um símbolo ocorreu ('_'), tendo ocorrido uma vez. Para cada contexto um símbolo especial é reservado, chamado de escape (ESC). A frequência desse símbolo depende da implementação do PPM. A implementação clássica considera que a frequência é equivalente ao número de símbolos distintos que já ocorreram naquele contexto (Moffat, 1990). O propósito desse símbolo especial será descrito mais adiante.

No caso em questão, o símbolo '_' tem uma probabilidade de ocorrência equivalente a 50% após o contexto 'BA'. Esse percentual é traduzido em um código binário de ponto fixo através de codificação aritmética. Em seguida a tabela de frequência dos contextos é atualizada com o símbolo recém processado e o codificador avança para processar o próximo símbolo.

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

Tabela 1. Contextos de ordens de zero a dois e seus respectivos símbolos/frequências

Ordem	Contexto	Símbolo (Frequência)
2	B A	ESC(1), _(1)
1	A	ESC(3), B(2), C(1), _(2)
0		ESC(4), A(5), B(2), C(1), _(2)

Caso o símbolo a codificar fosse 'C' em vez de '_' (supondo o texto 'ACABA_A_ABAC'), observa-se que em nenhum momento no passado esse símbolo foi encontrado depois do contexto 'BA'. Nesse caso deve ser codificado o símbolo de escape, com probabilidade de 50%. Esse símbolo sinaliza que o contexto deve ser reduzido (de 'BA' para 'A') e a busca feita novamente. Dessa vez, três símbolo ocorreram após 'A'. Um deles é aquele que se deseja codificar, com probabilidade de 12,5% (uma ocorrência dentre as oito existentes). A existência de probabilidades iguais (ex. a probabilidade de aparecer 'B' ou '_' depois de 'A') é resolvida pela codificação aritmética através da divisão da escala de probabilidades em intervalos.

O pseudo-código do Algoritmo 1 mostra como o contexto diminui de tamanho a medida que a busca avança. No pior dos casos o contexto é reduzido à ordem 0 (zero). Nesse caso leva-se em consideração a frequência total dos símbolos, independente de onde eles apareceram. Todos os símbolo possíveis são contemplados nessa lista, então a busca sempre será bem sucedida nesse nível. A descompressão segue o caminho inverso. Símbolos decodificados alimentam o contexto e servem de indício para a decodificação do próximo símbolo.

Algoritmo 1: CODIFICAÇÃO USANDO PPM

```
   Entrada: simbolos_lidos, simbolo_a_codificar
 1 início
 2     contexto ← ultimos n simbolos_lidos
 3     para cada ordem de n a 0 faça
 4         no ← busca_simbolo(contexto, simbolo_a_codificar)
 5         se no não for nulo então
 6             codifica_simbolo(no)
 7             retorna
 8         fim
 9         senão
10             codifica_escape()
11             encurta_contexto()
12         fim
13     fim
14 fim
```

O algoritmo original e muitas de suas variações utilizam um tamanho máximo de contexto. Experimentos indicam que melhores taxas de compressão são obtidas ao utilizar contextos cujo tamanho máximo (n) está compreendido no intervalo de três a sete (Moffat, 1990). Também existem variações que não limitam o tamanho do contexto (Cleary e Teahan, 1997). No entanto, seu consumo de memória é elevado, apesar da preocupação

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

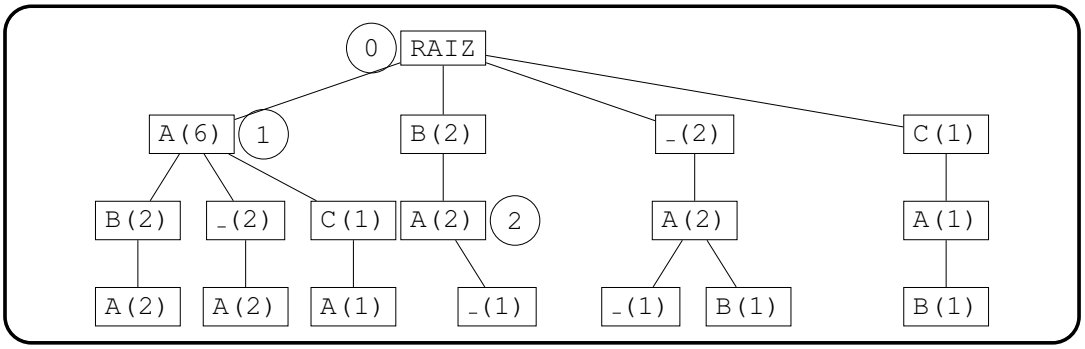

Figura 2. Árvore de contexto (ordem máxima = 2)

no uso de estruturas de dados que minimizem esse custo.

A Figura 2 mostra a árvore de contextos que seria gerada para o texto usado como exemplo, considerando um tamanho máximo de contexto igual a 2 e um universo de símbolos composto por 'A', 'B', 'C' e '_'. O contexto de cada nó é identificado pela concatenação do símbolo do nó atual e dos símbolos dos nós ascendentes. Para cada nó é também armazenada a frequência de ocorrência do contexto correspondente. Dentro de um nó os filhos são ordenados pela frequência. Essa organização visa obter menores tempos na busca de um símbolo (linha 4 do pseudo-código). Caso esse símbolo seja muito frequente, poucos nós deverão ser visitados até que ele seja encontrado.

O nível zero é reservado para o contexto vazio. Todos os símbolos possíveis aparecem como filhos do raiz com uma frequência não nula, para que sua probabilidade de ocorrência seja superior a zero. Isso garante que símbolos que nunca sucederam um contexto específico possam ser codificados quando o contexto for encurtado até ficar vazio.

Os círculos indicam os nós que fazem parte do contexto atual, ou seja, os nós que identificam os símbolos presentes em 'BA', em cada uma das ordens, de zero a dois. A operação que encurta contexto (linha 11 no pseudo-código) pode ser vista como a navegação de um nó de ordem maior para um nó de ordem menor.

A taxa de compressão dos algoritmos PPM depende fortemente da existência de padrões de repetição nos símbolos a processar. Isso é bastante comum em textos, onde as mesmas palavras (ou compostas pelo mesmo radical) costumam aparecer com frequência. Isso leva à geração de árvores em que cada pai possua poucos filhos muito frequentes e muitos filhos pouco frequentes. Nesses casos, as sequências de caracteres que costumam aparecem juntas são codificadas com poucos bits, uma vez que a probabilidade de ocorrência dos caracteres dentro dessas palavras comuns será alta.

Consideração uma organização de arquivos orientado a coluna, em que os valores de cada coluna são armazenados de forma consecutiva, a expectativa é que sejam encontrados ainda mais padrões do que o que se costuma encontrar em arquivos de texto convencionais. A próxima seção investiga como a natureza dos dados encontrados em arquivos orientados a colunas se relaciona com o PPM e com outros compressores.

XII Escola Regional de Informática de Banco de Dados - ISSN 2177-4226 13 a 15 de abril de 2016
Sociedade Brasileira de Computação - SBC Londrina - PR, Brasil

4. Experimentos

Os experimentos desta seção mostram como diversos compressores de texto se comportam ao comprimir dados orientados a colunas. Os compressores testados são o PPM, codificação aritmética, BWT e LZ. Todos foram implementados em C. Os dois primeiros foram criados como parte deste trabalho. Para os dois últimos foram utilizados os compressores BZIP2 e GZIP, respectivamente. Nenhuma *flag* de otimização foi utilizada na compilação/execução dos códigos-fontes.

Os dados das colunas foram gerados a partir do TPC-H, um conhecido *benchmark* usado para avaliar a performance de processamento de transações de bancos de dados (Council, 2008). O modelo de dados do TPC-H é composto por oito tabelas (PART, SUPPLIER, PARTSUPP, CUSTOMER, NATION, LINEITEM, REGION, ORDERS). O gerador de dados foi configurado com um fator de escala igual a 1, o que resultou em um volume de dados de aproximadamente 1 GB.

Após a geração dos dados, as tabelas foram segmentadas de modo a se aproximar da organização física de arquivos empregada em SGBDS orientados a colunas. Para isso, uma tabela com x colunas foi dividida em x arquivos distintos. Em cada arquivo os valores da coluna respectiva foram adicionados consecutivamente, separados um do outro por um símbolo de uso reservado.

Foram selecionadas para compressão apenas colunas que armazenassem tipos de dados textuais e tivessem alta cardinalidade. Diversas colunas que satisfaziam os critérios foram avaliadas. De modo geral, em todas elas o resultado foi semelhante. Desse modo, foi escolhida a coluna COMMENT da tabela CUSTOMER como referência.

A primeira análise é voltada exclusivamente ao método de compressão PPM. A intensão é descobrir o melhor tamanho de contexto para o domínio de dados escolhido de modo a obter melhores taxas de compressão sem que isso acarrete em perdas significativas de desempenho. A relação entre esses dois fatores (compressão x desempenho) se dá basicamente pelo tamanho da árvore gerada. Contextos curtos geram árvores menores. Assim, gasta-se menos tempo na manutenção da árvore. Por outro lado, a probabilidade de ocorrência de um símbolo qualquer tende a ser menor, o que diminui a taxa de compressão.

A Figura 3 apresenta resultados empíricos relacionando o tamanho do contexto aos fatores desempenho (gráfico da esquerda) e taxa de compressão (gráfico da direita). O desempenho é medido como o tempo necessário em milissegundos para realizar a compressão. A taxa de compressão é medida em bits por código(bpc), que indicam quantos bits são necessários para compactar cada byte do arquivo de entrada. Foram testadas diversas versões do PPM, variando o tamanho máximo do contexto de dois (PPM-2) até sete (PPM-7). Os gráficos permitem ver como os resultados variam conforme parcelas maiores do arquivo COMMENT são processadas.

Como pode-se ver, o PPM-2 obteve o pior desempenho nos dois fatores analisados. Apesar de pouco tempo ser gasto na geração da árvore, muito tempo é investido na busca de nós a partir de um pai. Como os nós filhos são ordenados pela frequência, a busca por nós com baixa frequência provoca um acesso a um maior número de filhos até que se encontre o nó correto. Além disso, como o arquivo comprimido é maior, perde-se mais tempo em operações de gravação do arquivo de saída.

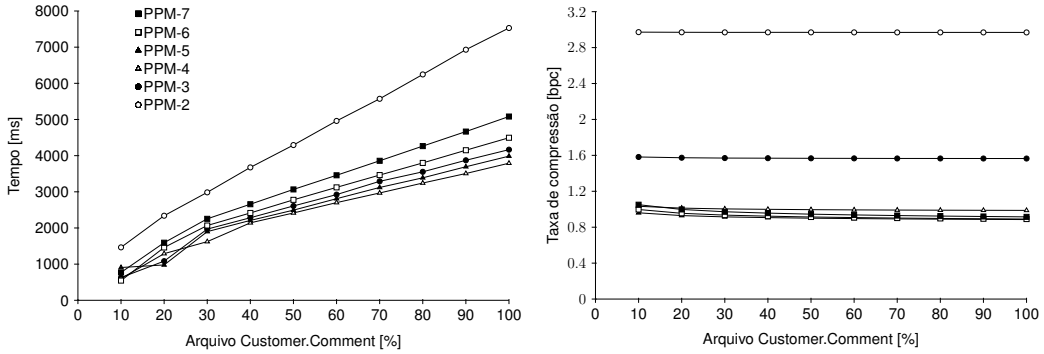

Figura 3. Diferentes versões do PPM variando o tamanho máximo do contexto

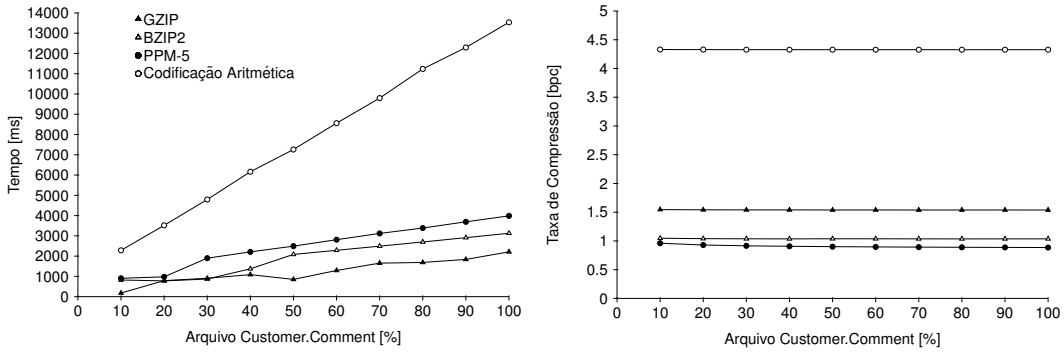

Figura 4. Algoritmos de compressão processando um arquivo orientado a colunas

O PPM-7 também gerou resultados insatisfatórios quanto ao tempo de processamento. Isso ocorre em boa parte porque a manutenção da árvore requer muito trabalho. Por outro lado, o PPM-7 obteve uma boa taxa de compressão. No entanto, a taxa é semelhante aos resultados obtidos por versões do algoritmo que usaram tamanhos máximos de contexto menores. Analisando os dois fatores em conjunto, percebe-se que os PPM-4 e PPM-5 apresentam uma boa relação custo-desempenho, sendo que o PPM-4 é levemente superior no quesito desempenho enquanto o PPM-5 é melhor na taxa de compressão. Como o objetivo é atingir boas taxas de compressão sem perdas significativas de desempenho, a versão PPM-5 foi utilizada no demais experimentos.

Os próximos gráficos(Figura 4) comparam o desempenho e a taxa de compressão de todos os algoritmos avaliados. Novamente a medição foi feita considerando parcelas do arquivo COMMENT. Os resultados mostram que a compressão aritmética pura perde nos dois fatores. Os outros três algoritmos apresentam um custo benefício semelhante, sendo que o GZIP apresenta o menor tempo de execução enquanto o PPM-5 apresenta a maior taxa de compressão. Caso a intenção seja otimizar a ocupação de espaço em disco, a alternativa que implementa o PPM seria preferível.

Para finalizar, os gráficos da Figura 5 incluem na comparação com a coluna COMMENT o corpus CALGARY. Esse corpus possui uma coleção de arquivos de variados formatos e tamanhos, e é bastante utilizado como *benchmark* de compressão de dados (Arnold e Bell, 1997). A tabela CUSTOMER, de onde foi extraída a coluna COMMENT, também foi adi-

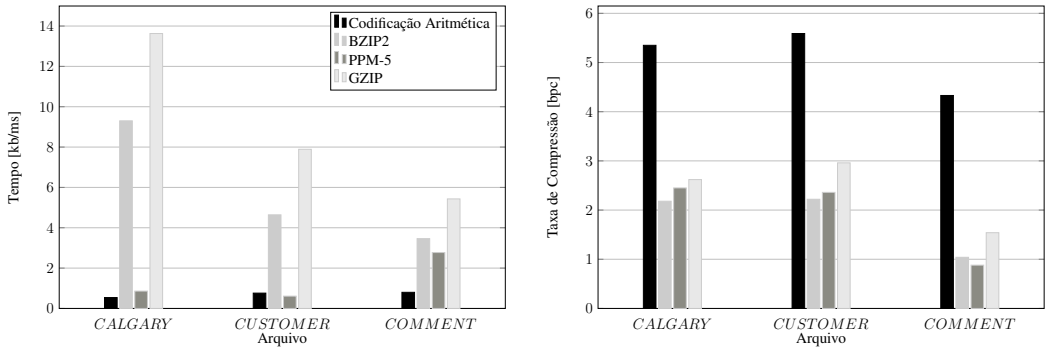

Figura 5. Algoritmos de compressão processando arquivos variados

cionada. Como essa tabela é orientada a registros, cabe fazer uma análise referente ao comportamento dos algoritmos de acordo com a organização física dos dados. Os arquivos CALGARY, CUSTOMER e COMMENT ocupam respectivamente 3.2, 23.9 e 10.8 mbytes.

O gráfico da esquerda exibe a velocidade de processamento, medida em kbytes processados por milissegundo. Aqui pode-se ver que BZIP2 e GZIP são visivelmente mais rápidos do que o PPM e a codificação aritmética na compressão de arquivos convencionais ou da tabela orientada a registros. No entanto, essa relação de desempenho é menos impactante na compressão do arquivo orientado a colunas. Além disso, a velocidade do GZIP e BZIP2 diminui na compressão do arquivo COMMENT, enquanto a velocidade do PPM aumenta. Isso mostra que a proximidade de valores de um mesmo domínio impulsiona o PPM e causa um efeito contrário em seus principais concorrentes.

Já o gráfico da direita exibe a taxa de compressão. Todos os compressores obtiveram melhores resultados no arquivo orientado a colunas. Isso demostra que a redundância desse tipo de arquivo é bem explorada pelos algoritmos. O que vale a pena destacar aqui é a distinção que existe na compressão do CALGARY e CUSTOMER em comparação à COMMENT. Nos dois primeiros, BZIP2 e GZIP foram superiores ao PPM. Já no arquivo COMMENT essa relação se inverteu. Esse resultado sugere que métodos baseados em contexto como o PPM são mais eficazes na compressão de informações textuais cujo universo de valores aceitos pertence a um domínio de dados mais restrito.

5. Conclusões

Arquivos orientados a colunas são realmente bem explorados por compressores de dados baseados em padrões. Os experimentos realizados mostraram que as taxas de compressão são maiores quando se lida com dados bem comportados cujos valores pertencem a um domínio de dados bem definido, uma vez que a entropia tende a ser menor.

Outro ponto importante levantado nos experimentos foi a descoberta de que o método de compressão PPM se mostra particularmente viável para esse tipo de dados, apresentado taxas de compressão superiores aos métodos concorrentes e um tempo de processamento não muito superior. Aqui cabe ressaltar que também foram realizados experimentos medindo tempo de execução na descompressão. Esses resultados foram omitidos por serem análogos aos resultados obtidos na compressão, posicionando o PPM como método de compressão com desempenho razoavelmente competitivo.

Os resultados obtidos servem de motivação para a investigação de formas de melhorar o desempenho do PPM. Nessa linha de pesquisa, um dos pontos a serem explorados surgiu de uma constatação feita durante os experimentos. Conforme a Figura 3 indica, a taxa de compressão do PPM permanece constante ao longo do processamento do arquivo orientado a colunas. Isso sugere que a árvore de contextos existente após o processamento do trecho inicial do arquivo apresenta uma relação de probabilidades similar à arvore de contextos existente após o processamento de todo o arquivo. Assim, a ideia a investigar é a interrupção na manutenção da árvore de contexto quando alguns critérios forem atingidos, na expectativa de que a árvore existente seja um modelo de predição bom o suficiente para a codificação dos próximos símbolos.

Referências

Abadi, D., Madden, S., e Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM.

Abadi, D. J., Boncz, P. A., e Harizopoulos, S. (2009). Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665.

Arnold, R. e Bell, T. (1997). A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference, 1997. DCC'97. Proceedings*, pages 201–210. IEEE.

Burrows, M. e Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm (relatorio tecnico).

Cleary, J. G. e Teahan, W. J. (1997). Unbounded length contexts for ppm. *The Computer Journal*, 40(2 and 3):67–75.

Council, T. P. P. (2008). Tpc-h benchmark specification. [Online; accessado em 19 de janeiro de 2016].

Deutsch, L. P. (1996). Deflate compressed data format specification version 1.3.

Han, J., Haihong, E., Le, G., e Du, J. (2011). Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE.

Huffman, D. A. et al. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Moffat, A. (1990). Implementing the ppm data compression scheme. *Communications, IEEE Transactions on*, 38(11):1917–1921.

Westmann, T., Kossmann, D., Helmer, S., e Moerkotte, G. (2000). The implementation and performance of compressed databases. *ACM Sigmod Record*, 29(3):55–67.

Witten, I. H., Neal, R. M., e Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Wu, K., Otoo, E. J., e Shoshani, A. (2002). Compressing bitmap indexes for faster search operations. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 99–108. IEEE.

Ziv, J. e Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536.

COMPUTING 2nd Edition Volume I

Compression of Very Sparse Column Oriented Data

Vinícius Fulber Garcia[1], **Sérgio Luís Sardi Mergen**[1]

[1]Universidade Federal de Santa Maria

vfulber@inf.ufsm.br, mergen@inf.ufsm.br

Abstract. *Column oriented databases store columns contiguously on disk. The adjacency of values from the same domain leads to a reduced information entropy. Consequently, compression algorithms are able to achieve better results. Columns whose values have a high cardinality are usually compressed using variations of the LZ method. In this paper, we consider the usage of simpler methods based on run-length and symbols probability in scenarios where datasets are very sparse. Our experiments show that simple methods provide promising results for the compression of very sparse datasets where the delimiter is the predominant symbol.*

1. Introduction

Column oriented databases are not new. However the recent interest in NoSQL databases has lead to an increasing amount of research in this area. Differently from traditional relational databases that store rows contiguously on disk, the column oriented approach stores columns contiguously on disk. This physical organization allows faster access for queries that require a small amount of columns, mainly because less pages need to be load from disk into memory (Matei e Bank, 2010). Another advantage of the columnar approach over the row approach is data compression. In a column oriented database all information stored in a page belong to the same domain and data type. The information homogeneity leads to a reduced information entropy, and this fact can be explored by data compression algorithms (Abadi et al., 2006).

Compression in column oriented databases is achieved in several different ways. If cardinality is low (a small number of unique values exists), lightweight methods are normally used. Examples include dictionary encoding, RLE and bitmap indexing. If cardinality is high, variations of the heavyweight LZ compression method are normally used. Heavyweight methods are more flexible than lightweight ones. They are generally suited when data contains a certain degree of redundancy, since they are able to encode redundant parts effectively, achieving good compression ratio associated with an acceptable response time.

There is a special case of high cardinality columns where the underlying datasets are very sparse. Besides the patterns that are naturally found when the column values are stored in adjacent positions, the sparsity may also contain additional patterns composed of sequences of adjacent undefined values. Multiple occurrences of these sequences lead to redundancy, which opens an opportunity for compression. Methods based on LZ are naturally potential candidates to explore sequences of undefined values. However, it is an interesting perspective to investigate how different (and simpler) compression strategies handle such patterns.

In this paper we focus on the compression of very sparse datasets. Our goal is to verify how an LZ compression method behaves when compressing a significant number of long runs of the delimiter. We also intend to verify whether using simpler methods compensate in those cases, both in terms of compression ratio and compression/decompression speed.

This paper is organized as follows: Section 2 presents a motivational example. Sections 3 reviews compression methods relevant to our work: Section 3.1 and 3.2 focus on the methods called Run Length Encoding (RLE) and Entropy Encoding (EC), and Section 3.3 reviews concepts related to the LZ family of compression methods. For each of the aforementioned methods we outline in general terms how the sparsity is explored to achieve compression. The final part reports the experimental results and presents our concluding remarks.

2. Motivational Example

Figure 1 compares two distinct physical organizations of a fictitious PROJECT table composed by columns YEAR, STATUS and COMMENT. The NSM (N-ary Storage Model) page layout is employed by conventional relational databases that store records contiguously on disk. On the other hand, the DSM (Distributed Storage Model) page layout is employed by databases that store columns contiguously on disk. As we can see, a NSM page keeps whole records, whereas a DSM page keeps values from single columns (Ailamaki et al., 2001). The DSM design subsumes strategies used by modern columnar databases, such as MonetDB and Vertica (Lamb et al., 2012).

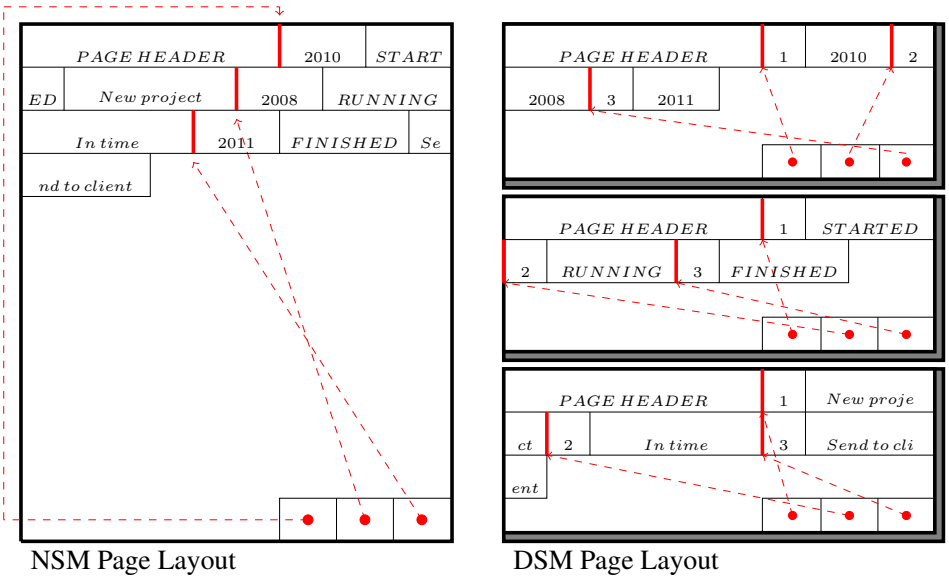

Figure 1. Three different page layouts

In both page layouts, the data grow from one side and auxiliary vectors grow from the other side. The vectors contain offset information, indicating where a record begins (in the case of NSM) or a field begins (in the case of DSM). Presence vectors could also be informed to indicate whether the value of a field is unknown. To simplify, the columns of

our fictitious table do not accept null values. The blank spaces illustrated in the example correspond to empty fields, which are different than unknown.

Several lightweight compression methods are useful when values of the same column are stored contiguously in a page (Zukowski et al., 2006). For instance, fields from the YEAR column could be compressed using frame of reference, a method that codes values as a difference to a reference value (the average year). On the other hand, low cardinality columns such as STATUS could be compressed using dictionary encoding, a method that codes values as indexes that point to a dictionary where all possible values are mapped. Other useful techniques when cardinality is low include packing values into bytes to allow byte aligned dictionary indexing, run length encoding for sorted columns and bitmap encoding that uses different bitmaps to each possible distinct value.

The beauty of some lightweight methods (like Dictionary Encoding and Frame of Reference) is that they allow direct access to random fields, without the need to decompress the previous fields. In some cases it is also possible to operate directly on compressed data. For instance, it is easy to compare if a given compressed field is greater than a specific year by using the reference value as part of the comparison.

However, not all columns support this kind of compression. Take for instance the COMMENT column. Comments are usually variable length sentences written in natural language. The lack of regularity in textual data types inhibits the usage of simple encoding schemes. The same limitation would apply to the YEAR and STATUS columns if the data domain was more relaxed. For instance, if text is allowed as part of the content of YEAR, the presence of any non-numeric character would break the frame of reference method. Similarly, if the possible values for STATUS are not restricted, data can become too skewed for a dictionary encoding to succeed.

The presence of irregular values demands the usage of heavyweight compression methods, where the purpose is to apply a more sophisticated approach (and usually more time consuming) to reduce the number of bits needed to encode a message. Heavyweight methods do not allow direct access to individual fields from a compressed message. However, they may be the only solution if one wants to reduce the space occupied by the values from a column.

It is important to observe that the usage of compression methods that lack the ability to work directly on compressed data implies that offset vectors are useless. Instead, we can tell one field from the other by a reserved symbol used as a delimiter. For instance, assuming the semi-colon (;) is the delimiter, the status fields presented in Figure 1 could be serialized in a DSM based page as 'started;running;finished'.

The presence of delimiters in very sparse datasets form redundancy patterns that can be explored to increase compression. For instance, assume as our running example a new collection of status fields whose possible values are 'a' and 'b'. Also, assume all values are undefined, except two, as indicated below:

; ; ; a ; ; ; b

Observe that adjacent undefined status form runs of the delimiter symbol. This kind of pattern is naturally compressed by heavyweight methods. What we argue in this

paper is that some lightweight methods could be useful for very sparse datasets as well.

Before going any further, we observe the existence of null suppression techniques useful for encoding very sparse datasets composed by a significant amount of null fields. For instance, Abadi et al. (2007) propose innovative page layouts to be used depending on the sparsity of data, as Figure 2 shows. If data is very sparse, null values are not represented at all. Instead, the pages keep a list of the positions where data is not null and another list with the values contained in those positions. This page layout was proposed as part of the C-Store database (the academic version of the Vertica database). The pitfall of the approach is that it was designed to work with fixed length columns in order to enable trivial vectorization. On the other hand, in this paper we consider variable length columns that could not be smoothly accommodated in a vector.

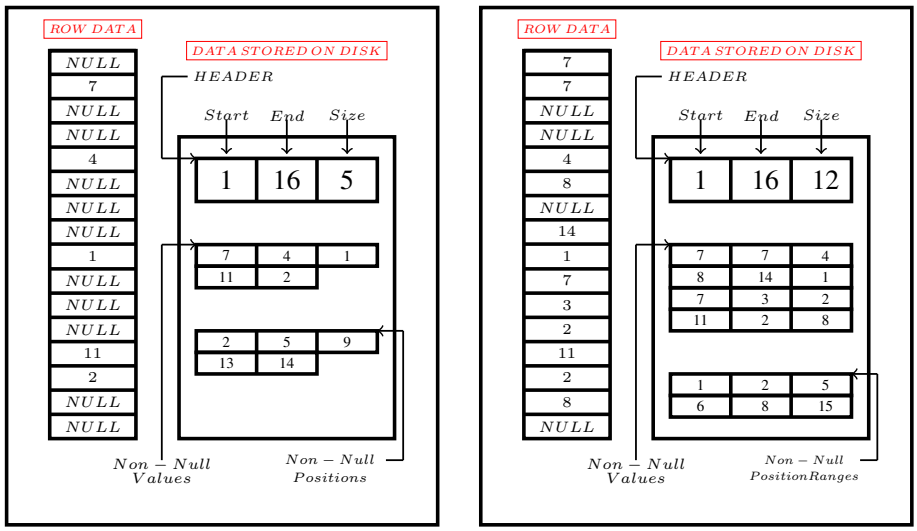

Figure 2. A page layout for storing very sparse fields

To restate, in this paper we care about very sparse datasets from variable length skewed/irregular columns that are not properly handled by null suppression techniques. Next section shows how welterweight and heavyweight methods handle the redundancy patterns formed when the delimiter appears in consecutive positions.

3. Compression Methods

This Section presents state of the art lossless compression methods for texts. The methods first appeared decades ago, and their essence remained the same throughout the years. In what follows we present the original techniques, discussing how sparse datasets can benefit from the core ideas.

3.1. Run Length Encoding

Run Length encoding (RLE) is one of the most simple compression methods. Basically, the purpose is to replace runs with codes. Many variations of this technique exist. One of them is to encode every run, regardless of its length. In this case the run is replaced by a code composed by the symbol and the count of how many times it repeats.

Another approach is to only encode runs whose length is higher than a predefined threshold. An escape symbol is needed to indicate that the coding scheme is about to be used. If no symbol can be reserved to be used as escape, one trick is to enter the coding scheme after the symbol repeats itself a specific number of times.

The nature of data in very sparse column oriented datasets allows a more proper encoding scheme. In most cases no symbols will be part of a run except for the delimiter. Therefore, one strategy is to encode the delimiter as the symbol itself followed by the count of how many other delimiters belong to the run. All other symbols are copied without encoding.

Figure 3 shows how the running example message would be encoded using this scheme. The message with nine bytes is compressed within six bytes, which yields a compression ratio of 33%, or 5,3 bits per code (bpc). Observe that the symbols are encoded as characters and the counting is encoded as decimal numbers. Also, the codes have a fixed 8 bit representation to allow byte aligned processing.

message	;	; ;	a	;	; ; ;	b
RLE	;	2	a	;	3	b
bytes	00111011	00000010	01100010	00111011	00000011	01100011

Figure 3. Encoding the running example message with RLE

Despite the simplicity, this technique is useful in many different compression scenarios, as part of more sophisticated methods, such as JPEG, to perform lossy image compression (Skodras et al., 2001) and BWT, to perform lossless text compression (Burrows e Wheeler, 1994). Using RLE in isolation is worthy only in very specific scenarios, where very long runs prevail. The experiments in Section 4 discusses how sparse the column oriented files should be so RLE outperforms other methods.

3.2. Entropy Encoding

In information theory, the entropy of a message is the minimal amount of bits necessary to represent the symbols of the message. This number of bits is strongly related to how predictable the message is. A low entropy means it is more easy to predict what symbols are more likely to appear, so less bits are needed to encode a message. For instance, files where one symbol is much more common than the others (like the delimiter in very sparse columns oriented files) have a low entropy, since we can predict that symbol will appear in most positions inside that file.

Two of the most popular compression methods based on entropy are the Huffman coding (Huffman et al., 1952) and arithmetic coding (Witten et al., 1987). The purpose of entropy encoding (or statistical encoding) methods is to reduce the number of bits used to encode symbols that occur frequently. The way these methods achieve this is quite different, as we explain next.

Huffman Coding: The Huffman coding assigns to each symbol of the message an unique and unambiguous sequence of bits, reserving the smaller sequences to the most frequent symbols. A binary tree is used to create that mapping. The construction of

the tree is fairly simple. First all symbols are transformed into nodes and each node is assigned with its corresponding frequency. Than, a new node is created to contain as children the two less frequent nodes. The frequency of the newly created node is the sum of the child's frequencies. The process repeats until all nodes (but one) have a parent. The node without a parent becomes the root. Bits are assigned to all edges, using zero and one to left and right edges respectively.

Figure 4 shows the tree generated based on the running example message. Observe that coding the delimiter requires a single bit. After compression, the nine characters of the file are transformed into a 11 bit sequence (00010000011), giving a compression ratio of approximately 85%, or 1,22 bpc.

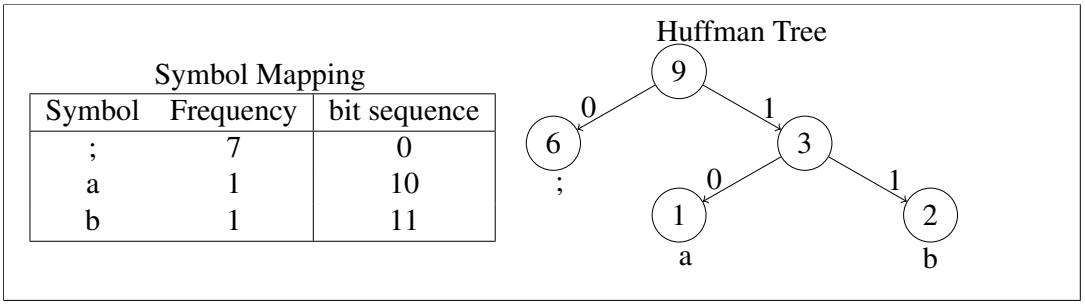

Figure 4. The Huffman tree construction based on the running example message

Arithmetic Coding: Unlike huffman code, in the arithmetic coding method there is no unique bit sequence that determines each symbol. Instead, the bits lead to a value that indicates the probability of occurrence of that exact sequence of symbols being compressed.

The probability ranges from zero to one. For sufficiently long messages, the probability would require a floating point precision higher than computers are able to express. To circumvent this technical problem, a fixed point precision value is used. When the value is about to overflow, the most meaningful bits are flushed out and the value is shifted to the right.

Figure 5 illustrates how the encoded probability is updated as the symbols are processed. Initially the probability of each symbol is divided into a scale ranging from zero to one and the probability range is updated as the symbols are processed. To simplify, the first case shows the probability distribution after the nine symbols of the message were processed. At this point, the probability of finding another delimiter is 77%. If the delimiter is indeed found, the probability is updated as demonstrated in the second case. As it shows, the probability of finding another delimiter after the first nine symbols drops to 60%. The probability keeps being updated as the remaining symbols are processed, and eventually the most significant bits are flushed.

In comparison to Huffman code, the arithmetic coding is better at approximating the entropy of the message. Moreover, it simplifies the process of adaptation, where the probability of each symbol is updated as the symbols are being read. Huffman encoding, on the other hand, needs to apply a preprocessing step over the whole sequence in order to determine the individual frequencies, or use complex tree manipulation operations in

order to adjust the tree topology.

To conclude, the example shows that the entropy encoding resulted in a much better compression ratio if compared to RLE. The reason is mainly related to the size of the message, too small for RLE to be effective. Our experiments in Section 4 present a more proper analysis that verifies how the methods behave in terms of compression ratio and compression/decompression speed when working with very sparse (and long) datasets.

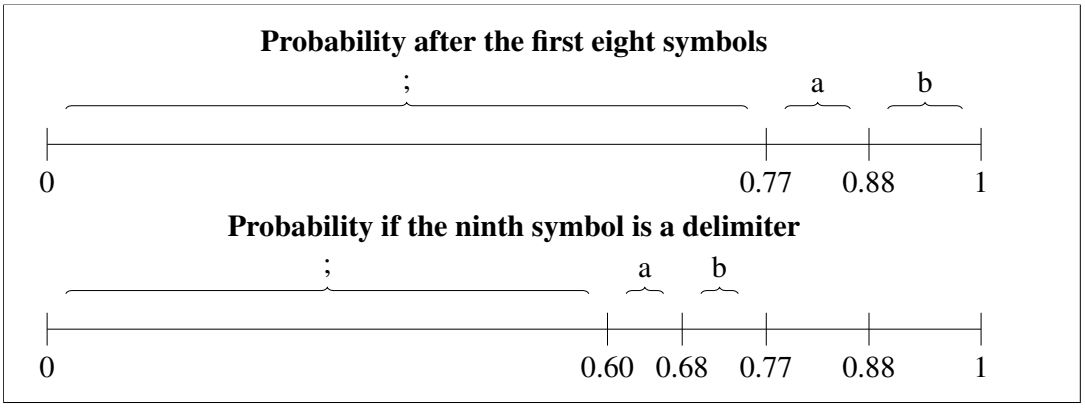

Figure 5. Encoding probabilities based on the running example message

3.3. Lempel-Ziv

Compression methods part of the LZ family are based on an idea proposed by Ziv e Lempel (1978). The method is a kind of dictionary encoding, where the dictionary is a sliding window composed by the most recently processed symbols of the message. The method checks a lookahead buffer if there is a sequence of symbols starting at the current symbol that can be dictionary encoded.

Many works propose variations related to the way the sequences found in the dictionary are encoded. In what follows we discuss the idea whose specification was later standardized: A sequence is encoded as a pair formed by an index value and an length value. The index refers to an offset inside the sliding window where the sequence being encoded can be found. The length refers to the number of characters encoded, starting from the offset. Is the offset is set to zero, it means the dictionary will not be used to encode the next symbol. In this case, the length receives the literal value that correspond to the unmatched symbol. The pairs are further compressed using two separate huffman trees, one for the offset and the other for the length.

Figure 6 shows what codes are generated for the running example message. The circle marks the current symbol. Symbols before the current symbol are part of the dictionary. Non-delimiter symbols are encoded as literals (no dictionary entry is used). The symbols from the first run of delimiters are also encoded as literals. Observe that the second (or the third) delimiters of this run could point to a dictionary entry, since a delimiter is already part of the dictionary. However, small sequences are not encoded using the dictionary as a way to achieve better compression. The reason is that the literal codes are more 'huffman compressible' than the dictionary codes.

Observe that the first three delimiters from the second run are packed as a single code. These run is part of the sliding window, as denoted by the rectangle surrounding the three delimiters. The last delimiter of the run could not be encoded, since there is no run in the dictionary composed by more than three delimiters.

Message									Offset	Length / Literal
;	;	;	a	;	;	;	;	b	0	;
;	;	;	a	;	;	;	;	b	0	;
;	;	;	a	;	;	;	;	b	0	;
;	;	;	a	;	;	;	;	b	0	a
;	;	;	a	;	;	;	;	b	4	3
;	;	;	a	;	;	;	;	b	0	;
;	;	;	a	;	;	;	;	b	0	b

Figure 6. Encoding the running example message with LZ

This compression scheme led to the specification of the DEFLATE standard (Deutsch, 1996), which imposes an agreement that all compliant compressors should follow, such as establishing the maximum size of the sliding window (32k) and the maximum size of the lookahead buffer (258 bytes). The standard forms the foundation of some of the most used compression algorithms used nowadays, like gzip and lzip. Besides, variations of LZ are also used by columnar databases Vertica and Sybase IQ to compress low cardinality columns.

This kind of dictionary based compression is known for associating good compression ratios and response time. Also, decompression is usually much faster than compression, since decoding a sequence of symbols is as simple as reading bytes already processed (from the sliding window). With respect to column oriented data formed by several runs of the delimiter symbol, runs already processed can be used as dictionary entries for coding other runs yet to come. The impact this kind of information have on LZ based compression is detailed in Section 4.

4. Experimental Results

The purpose of this Section is to evaluate how simple compression algorithms behave when dealing with very sparse datasets from columnar databases. Three simple algorithms were developed as part of this work. Their description is given below:

ARIT: An entropy based arithmetic encoding method that encodes a symbol based on its probability of occurrence.

RLE1: A method that encodes a run of delimiters using two bytes. The first byte is the escape code, which is the delimiter by itself. The second byte is a decimal value that represents the length of the run. Runs longer than 256 characters need to be compressed with successive codes.

RLE2: A method that encodes a run of delimiters using three bytes. The first byte is the escape code, which is the delimiter by itself. The other two bytes form a decimal value that represents the length of the run. Runs longer than 65536 characters need to be compressed with successive codes.

To perform a meaningful evaluation, we compared these methods with the commercially available GZIP, representing a method from the LZ family. All algorithms were written in C and no compiler optimization flags were set. Also, we observe that GZIP could be fine tuned to spend more time looking for matches inside the sliding window, trading a better compression ratio for a worst execution time. The default is to let GZIP decide when to stop looking for better matches (longer sequences). It turns out that the default setting is the one that usually brings the best results. Therefore, no fine tuning parameters were set either.

We used as dataset a database with tables generated from TPC-H, a well known benchmark for database performance evaluations (Council, 2008). Each column of the dataset was stored as a separate file and a special symbol was used as the field delimiter. During compression, each file was further divided into 8kb chunks. The chunks are used to emulate the way databases arrange data into pages (tranfer units between disk and memory). This size is a typical choice that databases use to reduce the amount of pages IO for random access and to allow the storage of reasonably large rows inside a page. For the sake of comparison, MySQL uses 16kb as the default page size and Oracle 10g uses pages from 4kb to 8kb. Also, the chunks where individually compressed. The reason is simple: limiting compression to single pages allows random access to a page without having to decompress the whole file.

We report the results achieved when compressing the following three columns from the CUSTOMER table.

- **COMMENT:** a column that stores user comments as natural text with length ranging from 29 to 116 characters.
- **ACCBAL:** a column that stores account balances with a two digits precision. Examples: '121.65' and '9561.95'.
- **MKTSEGMENT (MKT):** a column that stores one of five values of market segments. Examples: 'BUILDING' and 'HOUSEHOLD'. We artificially reduced the size of the file by running a preprocessing step that replaced values with a single character.

These columns are representative samples from three groups. The COMMENT column represent columns whose values are reasonably long. The MKT column represent columns whose values are single characters. The ACCBAL column represent columns whose values lie in-between these two extremes. Naturally, when values are shorter, the proportion of delimiters with respect to any other symbol is higher. Our intention is to verify how the compression methods behave when this proportion changes.

The selected columns were originally dense (all fields had content). Since our purpose is to evaluate sparse datasets, we artificially change the datasets by replacing fields at random positions with empty values[1]. The testbed is composed by several very sparse datasets where the amount of empty values varies from 99.05% to 99.95% of the total number of fields.

Figure 7 shows the COMMENT column results. On the left we present the compression ratio in terms of bits per code (bpc) achieved when varying the sparsity. GZIP is clearly the winner, regardless of the sparsity considered. The two versions of RLE behave similarly, and gradually approach the compression of GZIP. However, they are no match

[1]Null values could also be easily be accounted, if we assume a presence vector is used inside the pages

for GZIP even when more than 99.9% of the values are empty. Besides, we can see that ARIT is better than RLE when the sparsity is reduced. However, it is still much worst than GZIP.

On the right of Figure 7 we present the time needed to compress all chunks and the time needed to decompress them. We report the results when using the highest level of sparsity (99.95%). The first thing to notice is that ARIT is slower than the alternatives in compression and decompression. On the other hand, there is no significant difference in execution time considering GZIP and the run length methods.

Another interesting fact is that decompression is generally slower than compression. This behaviour was observed even with GZIP, where decompression is usually faster. We have also observed that the speed oscillates when working with different levels of sparsity. This lack of linearity indicates that the processing cost of GZIP and RLE is not a driving factor when working with small sized problems. For instance, with a 99.95% sparsity, the compression of GZIP uses 0,09 bpc. Chunks with 8kb becomes compressed pages of approximately 92 bytes. The decompression cost in this case is clearly affected by the fluctuations that occur during IO operations or CPU usage.

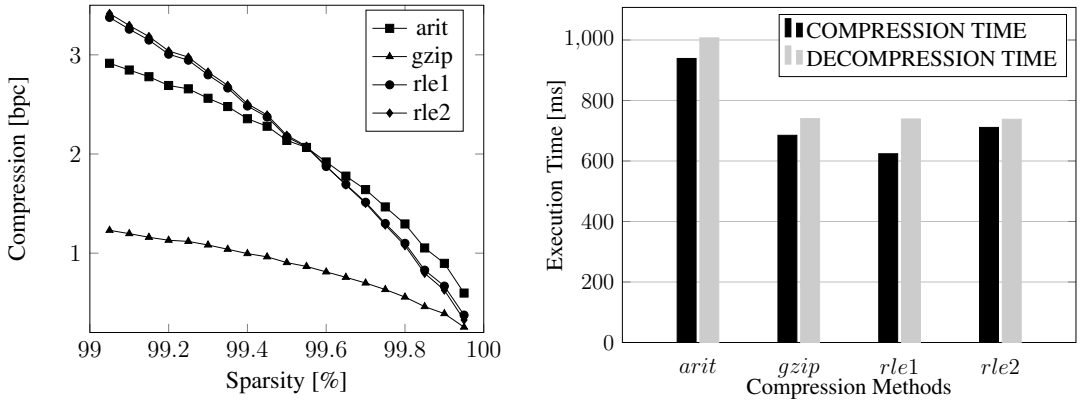

Figure 7. Compression results of CUSTOMER.COMMENT

Figure 8 shows the ACCBAL column results (compression in terms of bits per code on the left and execution time with 99.95% sparsity on the right). The first thing to notice is that the compression of ACCBAL is generally better than COMMENT. The reason is rather obvious: the information entropy is reduced when the frequency of the delimiter increases and the other symbols become less meaningful. In other word, patterns become more frequent than other arbitrary sequences, and compression methods are build to leverage from patterns.

It is also interesting to observe that the RLE versions were able to achieve much better results than before. For once, they are better than ARIT regardless of the sparsity considered. Additionally, RLE2 overpasses GZIP when sparsity reaches a certain level (99.8%). At this point the frequency of the delimiter reaches 98% of all symbol occurrences. For the sake of comparison, when compressing COMMENT, a sparsity of 99.8% corresponds to a delimiter frequency of 87%.

As for the execution time, the arithmetic encoding again shows the worst performance whereas the other methods had similar results. The measurements again suggest

that these algorithms does not differ significantly when working with small sized problems.

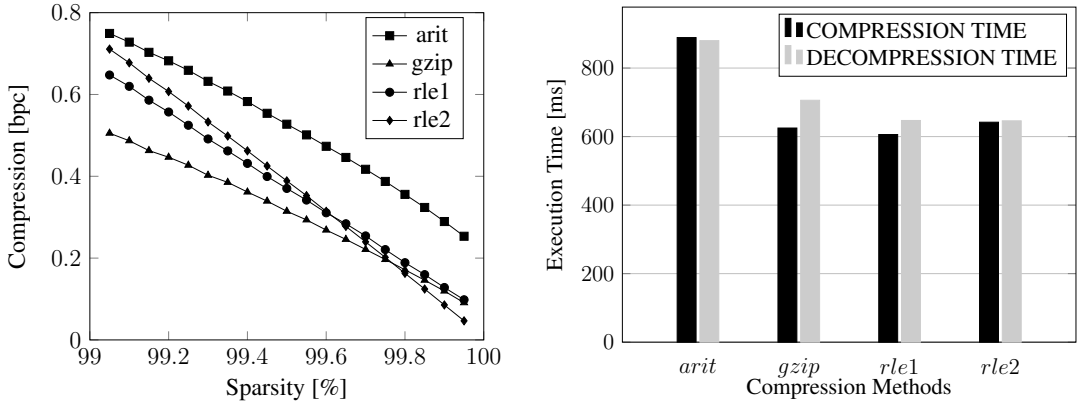

Figure 8. Compression results of CUSTOMER.ACCBAL

Figure 9 shows the market segment results. It is the test case where the delimiter appears with the highest frequency possible. Our evaluation shows how the algorithms behave with this extreme scenario. The first thing to notice is that the compression is generally higher when compared to the other tested scenarios. This happens mostly because the vocabulary is reduced (one delimiter plus five other characters), which leads to the occurrence of more patterns.

Most importantly, we call the attention to the fact that RLE2 becomes better than LZ when sparsity is above 99.3%, whereas in the previous experiment it only happened when sparsity was over 99.8%. With respect to the comparison between RLE1 and RLE2, the turning point (when RLE2 becomes better than RLE1) happens when sparsity reaches 99.65%, regardless of the column. Since these methods only compress the delimiter, the average number of characters per field does not change the turning point.

The results on the right side of Figure 9(running time with 99.95% sparsity) satisfy our previous observations that all but the arithmetic encoding method have a similar running speed when the message is small. This is interesting remark, if we consider that the key strength of GZIP is speed. When sparsity is high and pages are small, other methods become equally competitive.

5. Concluding Remarks

We have shown that, when sparsity is very high, column oriented data that is normally compressed using variations of the LZ method may also benefit from simpler methods based on the run length of delimiters, both in terms of compression ratio and speed.

With respect to execution time, the cost for compression and decompression of the RLE methods evaluated are similar to LZ when the files are segmented as pages. With respect to compression ratio, the results indicate two things. First, only sparsity matters if one wants to decide between the two RLE versions presented. However, if one wants to decide between RLE and GZIP, the average field size also matters. For columns where the fields are long, the sparsity would have to be much higher in order for RLE to overcome LZ.

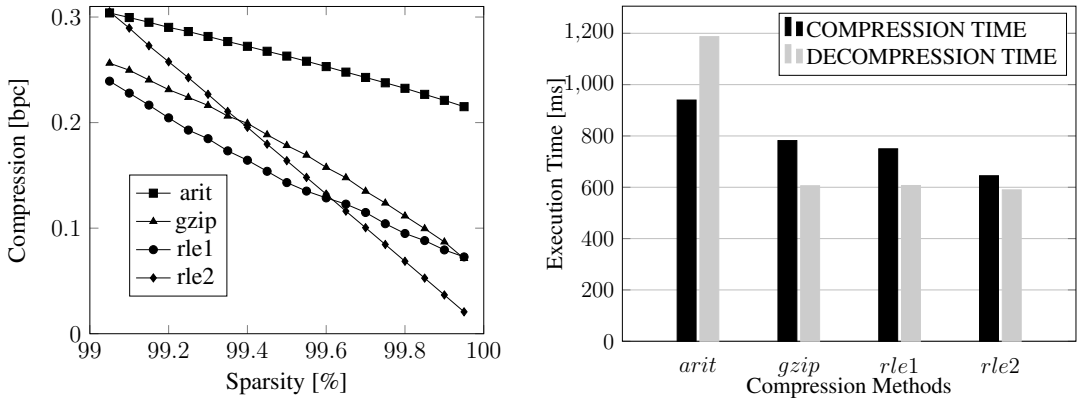

Figure 9. Compression results of CUSTOMER.MKTSEGMENT

One interesting (and useful) property of RLE is that it enables direct computation of counts of undefined values. However, unlike other lightweight methods (applied to low cardinality values), it does not provide direct access to random values. One possible way to overcome this limitation is the usage of page layouts designed to handle very sparse datasets composed by high cardinality values. This idea is similar in spirit with RLE, in the sense that no characters are encoded (other than the delimiter), and we have shown that this strategy is promising in some scenarios. The investigation of this novel page layout is left as future work.

References

Abadi, D., Madden, S., e Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM.

Abadi, D. J. et al. (2007). Column stores for wide and sparse data. In *CIDR*, pages 292–297.

Ailamaki, A., DeWitt, D. J., Hill, M. D., e Skounakis, M. (2001). Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Burrows, M. e Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm (technical report).

Council, T. P. P. (2008). Tpc-h benchmark specification. [Online; accessado em 19 de janeiro de 2016].

Deutsch, L. P. (1996). Deflate compressed data format specification version 1.3.

Huffman, D. A. et al. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., e Bear, C. (2012). The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801.

Matei, G. e Bank, R. C. (2010). Column-oriented databases, an alternative for analytical environment. *Database Systems Journal*, 1(2):3–16.

Skodras, A., Christopoulos, C., e Ebrahimi, T. (2001). The jpeg 2000 still image compression standard. *IEEE Signal processing magazine*, 18(5):36–58.

Witten, I. H., Neal, R. M., e Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Ziv, J. e Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536.

Zukowski, M., Heman, S., Nes, N., e Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA. IEEE Computer Society.