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.

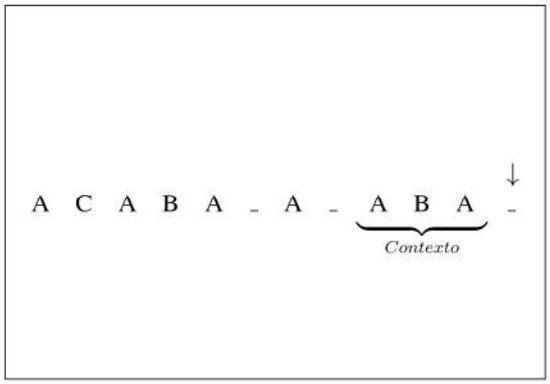

Figura 1. Símbolos a codificar

```
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.

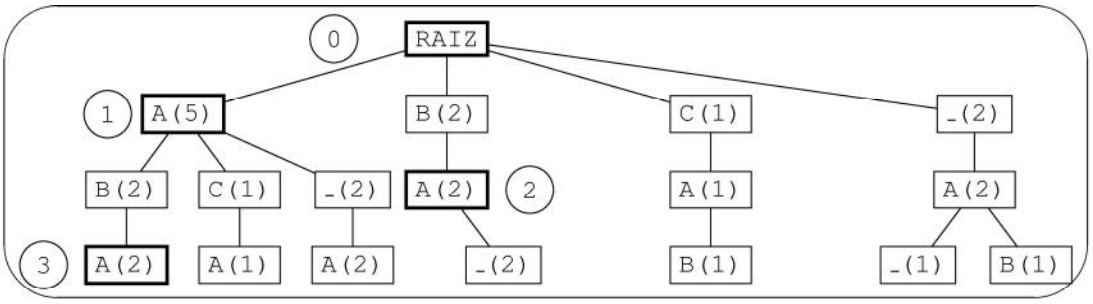

Figura 3. Árvore de Contexto (ordem = 3)

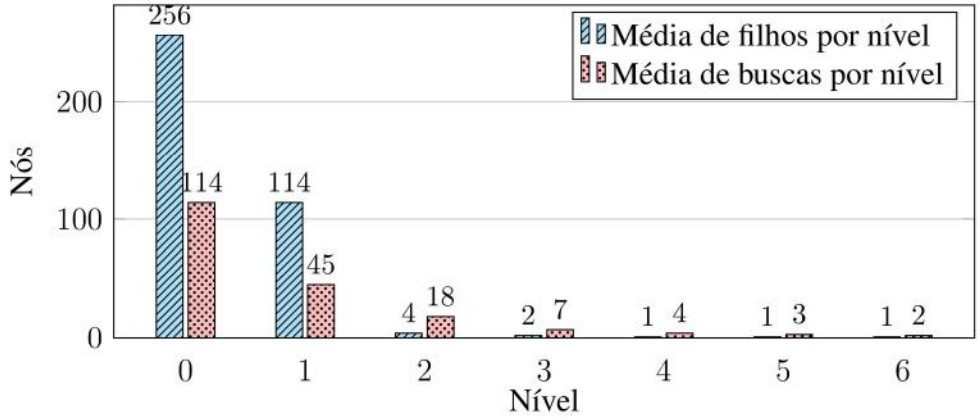

Figura 4. Estatísticas da árvore de contexto do Corpus Calgary

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

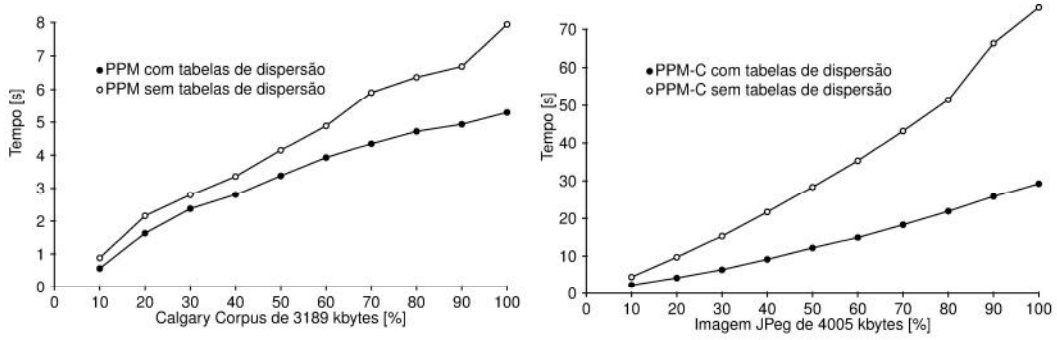

Figura 5. Tempo de Execução

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

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.