Análise algoritmo de Huffman

Danilo Farias de Carvalho[1], **Maria Teresa Silva Santos**[1], **Sergio Henrique Silva**[1]

[1]Programa de Pós-Graduação em Computação Aplicada (PPGCAP)
Centro de Ciências Tecnológicas (CCT)
Universidade do Estado de Santa Catarina (UDESC)
Joinville - SC - Brasil

{danilo.carvalho, maria.santos2805, sergio.hs}@edu.udesc.br

Abstract. *The Huffman's Algorithm has as main objective the data compression, for to do this is used the possibilities of occurrence of each symbol, to then determine the word in binary code, varying its size, so that the symbol that occurs the most has the smallest size. This paper is presenting a brief introduction to the history of the algorithm, its complexity, tests carried out that prove the complexity and the actual implementation of the algorithm.*

Resumo. *O Algoritmo de Huffman tem como principal objetivo a compressão de dados, usando as possibilidades de ocorrências de cada símbolo para que seja determinada a palavra em código binário, cujo tamanho varia de forma que o símbolo que mais ocorre tenha o menor tamanho. Será apresentada uma breve introdução à história do algoritmo, sua complexidade, testes realizados que comprovam sua complexidade e a implementação do algoritmo de fato.*

1. Introdução

Dentre os problemas computacionais conhecidos, a compressão de dados é um problema significativo que, ao decorrer do tempo, teve diversas soluções propostas. Uma das soluções para os problemas de compressão é a utilização de algoritmos gulosos (*greedy algorithms*). Em especial, no caso da compressão de dados, o algoritmo guloso de Huffman é um dos mais amplamente usados.

Este método é utilizado para compressão de diversos tipos de mídias diferentes, também como componente de *codecs* mais complexos (Salomon, 2007) – apesar de ter sido inicialmente desenvolvido para compressão de texto (Pu, 2006).

David Huffman, que desenvolveu o método como projeto da disciplina de "Teoria da Informação"em 1950 (Blelloch, 2001) enquanto era aluno de doutorado no MIT (Sayood, 2006) (Wikimedia Foundation, Inc., 2016) e publicou as suas conclusões em 1952 no artigo "*A Method for the Construction of Minimum-Redundancy Codes*" na revista *Proceedings of the I.R.E.* (Huffman, 1952).

A codificação de Huffman é baseada na "Teoria de Informação"de Shannon (Shannon, 1948), em que um elemento de informação (texto, imagem, áudio, etc.) é representado por símbolos, dos quais alguns ocorrem mais vezes que outros. Caso esses símbolos mais comuns sejam representados utilizando-se códigos menores (usando menos *bits*), obteremos uma codificação binária com menos *bits*, como resultado de uma diminuição do comprimento médio (em *bits*) de cada código (Huffman, 1952) (Pu, 2006).

O algoritmo de Huffman é, portanto, baseado na categoria de Codificação de Entropia – a informação é encarada como uma sequência de símbolos genéricos, menosprezando a semântica dos mesmos –, sendo um modo de codificação sem perdas, em que o código resultante é totalmente reversível; ou seja, sua decodificação resulta num fluxo de dados exatamente igual ao fluxo de dados de origem (Ribeiro, Apontamentos das Aulas, 2016).

Nas próximas seções, será analisada a complexidade, será tratada a implementação do algoritmo e serão relatados os testes realizados, expondo a tendência de crescimento do tempo em relação às entradas.

2. Análise de complexidade

Baseando-se em Cormen et. al. (2009), implementou-se o algoritmo de Huffman, e, para análise de complexidade, construiu-se o pseudocódigo representado no Código 1.

```
1 n <- |C| // C1 * 1
2 L <- C // C2 * O(n)
3 for i <- 1 t o n-1 // C3 * O(log n)
4     do alocar um novo nó z // C4 * O(log n)
5     nodeEsquerdo[z] <- x <- POP-MIN-LISTA(L) // C5 * O(log n)
6     nodeDireito[z] <- y <- POP-MIN-LISTA(L) // C6 * O(log n)
7     f[z] <- f[x] + f[y] // C7 * O(log n)
8     INSERE-LISTA (L, z ) // C8 * O(log n)
9 return EXTRACT-MIN (L) // C9 * 1
```

Código 1: Pseudocódigo do algoritmo de Huffman.

Na linha 1 e na linha 2 do Código 1, inicializam-se as variáveis `n`, que representa o número de caracteres em `C`, e `L`, que representa a lista de prioridades contendo os caracteres `C`. É possível observar, da linha 3 à linha 8, o laço que extrai repetidamente os nós de frequência mais baixa para, então, substituí-los da lista. `x`, `y` e `z` representam essas frequências, sendo `z` a soma das frequências `x` e `y`. Após `n - 1` iterações, encontra-se a raiz da árvore de Huffman, que é retornada na linha 9.

Com o panorama do pseudocódigo descrito no Código 1, torna-se possível analisar o tempo de execução do algoritmo de Huffman. O algoritmo cria uma árvore binária completa cujos nós são menores ou iguais aos nós filhos, caracterizando uma *min-heap*, que está representada na Figura 1, onde $f_1 \leq f_2$, $f_2 \leq f_3$ e assim por diante até $f_{n-1} \leq f_n$.

Para que a complexidade do algoritmo de Huffman seja entendida, também usa-se como base a Figura 1. Tendo um conjunto com n caracteres, inicializa-se a lista na linha 2 em tempo O (n) usando a construção da árvore mínima. O laço `for` que abrange as linhas 3 a 8 é executado $n - 1$ vezes. Como cada operação de *heap* utiliza o tempo O $(\log n)$, o laço `for` tem como tempo de execução O $(n \log n)$.

Observando-se as ordens de crescimento de notação assintótica (*big-O*) pode-se concluir que, o algoritmo de Huffman tendo uma complexidade O $(n \log n)$, pode ser considerado um algoritmo de crescimento logarítmico linear. O mesmo tem ordem de crescimento superior a O $(\log n)$ e O (n), porém mais eficiente que n^2, 2^n e $n!$.

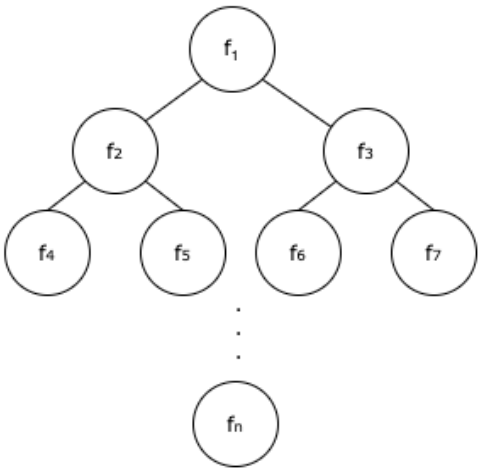

Figura 1. Exemplo de árvore *min-heap* gerada pelo algoritmo de Huffman.

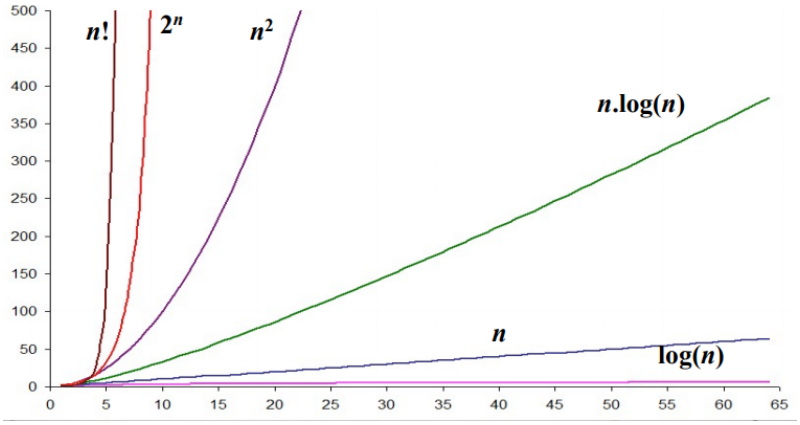

Figura 2. Ordens de Crescimento.

3. Implementação do TDA

A implementação do TDA foi elaborada pelos autores, na linguagem C. É possível analisar o código por completo na plataforma de hospedagem de código-fonte e arquivos com controle de versão *GitHub*. O endereço para acesso é: https://github.com/SergioHs/huffman

4. Relatório de Testes

4.1. Ambiente de Testes

O ambiente de testes Windows, foi composto por:

- Notebook Lenovo;
- Processador AMD-Ryzen 5 PRO – Modelo 3500G ;
- 16 GB de memória RAM DDR4;
- HD SSD no modelo m.2;
- Windows 10 Enterprise;

Ao final dos testes, também foi realizado um novo experimento em uma máquina virtual VMWare Player, em ambiente Linux Ubuntu 20.04 LTS, com 4 GB de RAM, sendo este executado no mesmo notebook do ambiente Windows. Os resultados também poderão ser observados na figura 3.

4.2. Metodologia da Análise dos Testes

Os testes foram realizados com os seguintes requisitos:

- Relatório de execução (tabela) com 5 entradas distintas;
- Tamanhos escolhidos: n = 10, 100, 1.000, 10.000 e 1.000.000;
- Comparação dos resultados de tempos metidos com o comportamento da curva descrita pela equação assintótica.

Para a confecção desta tabela comparativa, realizaram-se 30 execuções para cada entrada e os valores expressos nos resultados são as médias. Isto é importante para retirar qualquer influência de algum processo que esteja sendo executado no computador de testes.

Além disso, também foram gerados e avaliados outros indicadores estatísticos, tais como:

- Valor mínimo;
- Valor máximo;
- Desvio padrão;
- Moda;
- Mediana;
- Cálculo de erro para um intervalo de confiança de 95%.

Quanto mais estreito for o valor do intervalo de confiança, maior é a probabilidade de a porcentagem da população de estudo representar o número real da população de origem, dando maior certeza quanto aos resultados do objeto de estudo. Ou seja, um valor muito baixo de erro para um intervalo de confiança, indicará que a quantidade amostras testadas é o suficiente para a análise que se deseja.

O *Modo Debug*, como é visualizado em algumas IDEs, acrescenta diretivas de compilação para facilitar o *tracer* ou *debug* durante a fase de desenvolvimento, mas é importante que seja retirado quando a aplicação é instalada em ambiente de produção. A função deste modo nesta comparação foi apenas para demonstrar que até mesmo as pequenas configurações em um ambiente de desenvolvimento podem influenciar no tempo

de execução de um algoritmo. Colocar uma versão de um aplicativo em produção neste modo, é um erro que deve ser evitado. Porém, não é incomum que sejam encontrados programadores com este hábito.

Para ajudar a demonstrar o quanto o sistema operacional ou o *hardware* podem influenciar no tempo de execução de um mesmo algoritmo, também foram gerados dados estatísticos em um ambiente Linux. E é justamente esta a importância de se analisar assintoticamente um algoritmo, mitigar diferenças relativas ao ambiente de execução.

Em anexo a este trabalho, será entregue uma planilha detalhada com toda a massa de dados analisada.

4.3. Resultados Obtidos

Análise de Tempo						
Quantidade	(tempo em segundos - médias)			(percentual)		
	Modo Release (windows)	Modo Debug (windows)	Linux	Diferença Release x Debug	Diferença Release x Linux	Diferença Debug x Linux
10	0,003200	0,003167	0,000167	-1,05	-1818,85	-1798,86
100	0,003067	0,003233	0,000157	5,15	-1851,22	-1957,26
1.000	0,003400	0,004467	0,000313	23,88	-985,45	-1325,99
10.000	0,005267	0,005233	0,001607	-0,64	-227,75	-225,67
1.000.000	0,179800	0,203400	0,133063	11,60	-35,12	-52,86

Figura 3. Resultados Gerais Obtidos [Fonte: Os autores (2021)].

Pelo quadro 3, observa-se que:

- O *Modo Release*, na maioria dos casos, teve um desempenho superior ao *Modo Debug*. Para o arquivo de entrada de 1000 elementos, o *Modo Release* foi 23% mais rápido;
- O executável gerado em *Modo Debug*, foi 236% maior do que no *Modo Release* (68 kb e 19kb, respectivamente);
- O desvio padrão, em todos cenários, ficou próximo de zero;
- Para todas as entradas avaliadas, o tempo de execução em Linux foi superior ao tempo em Windows, mesmo este ambiente sendo virtualizado, e executado de dentro do próprio ambiente Windows.

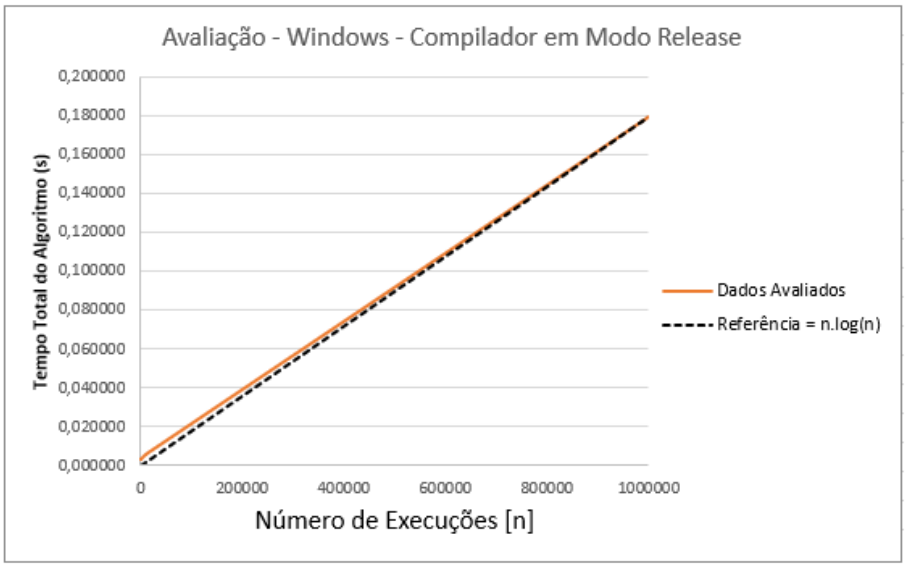

Figura 4. Gráfico de Análise de Tempo - Windows - Modo Release [Fonte: Os autores (2021)].

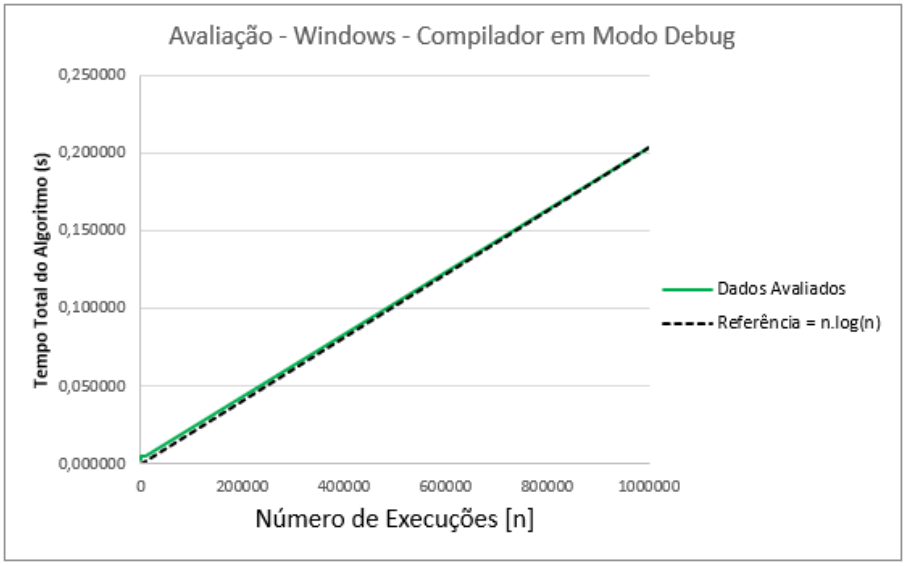

Figura 5. Gráfico de Análise de Tempo - Windows - Modo Debug [Fonte: Os autores (2021)].

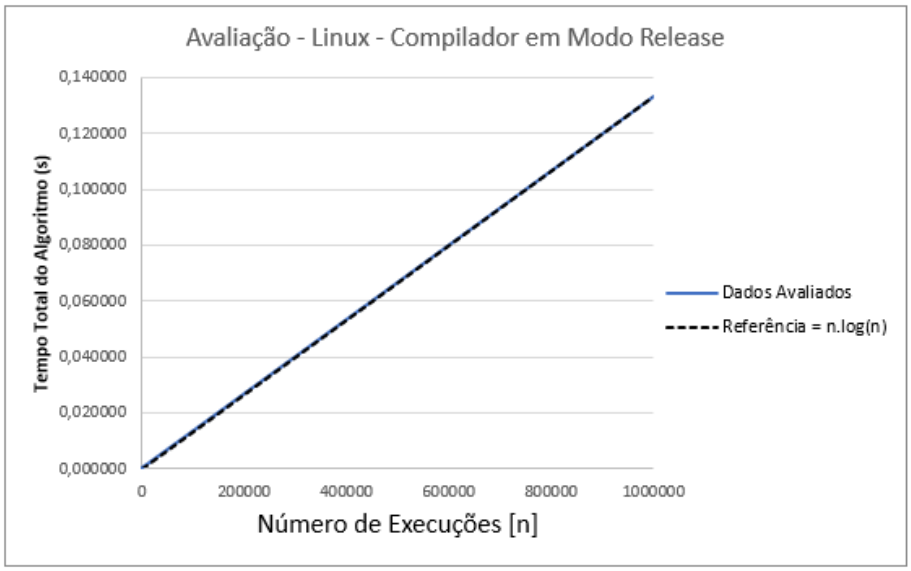

Figura 6. Gráfico de Análise de Tempo - Linux - Modo Release[Fonte: Os autores (2021)].

Em todos os cenários avaliados, o comportamento da curva é o mesmo observado na figura 2, para uma ordem de crescimento em O $(n \log n)$. Além da análise visual, foi adicionado ao gráfico uma curva de tendência, o que confirma a curva de crescimento avaliada.

WINDOWS - COMPILADOR EM MODO RELEASE					
Análise de Tempo					
Carga Avaliada (quantidade de entradas)					
(tempo em segundos)					
Quantidade	**10**	**100**	**1.000**	**10.000**	**1.000.000**
Média	0,00320	0,00307	0,00340	0,00527	0,17980
Mínimo	0,00200	0,00200	0,00200	0,00400	0,16000
Máximo	0,00400	0,00500	0,00400	0,00700	0,25100
Desvio	0,00071	0,00091	0,00067	0,00078	0,01938
Moda	0,00300	0,00300	0,00400	0,00500	0,16900
Mediana	0,00300	0,00300	0,00350	0,00500	0,17350
Erro - Para Int. de Confiança de 95%	0,00026	0,00032	0,00024	0,00028	0,00693

Figura 7. Quadro de Análise de Tempo - Windows - Modo Release [Fonte: Os autores (2021)].

WINDOWS - COMPILADOR EM MODO DEBUG					
Análise de Tempo					
Carga Avaliada (quantidade de entradas)					
(tempo em segundos)					
Quantidade	**10**	**100**	**1.000**	**10.000**	**1.000.000**
Média	0,00317	0,00323	0,00447	0,00523	0,20340
Mínimo	0,00100	0,00200	0,00200	0,00400	0,18900
Máximo	0,00500	0,00500	0,04000	0,00700	0,25700
Desvio	0,00083	0,00104	0,00676	0,00097	0,01884
Moda	0,00300	0,00300	0,00300	0,00500	0,19400
Mediana	0,00300	0,00300	0,00300	0,00500	0,19400
Erro - Para Int. de Confiança de 95%	0,00030	0,00037	0,00242	0,00035	0,00674

Figura 8. Quadro de Análise de Tempo - Windows - Modo Debug [Fonte: Os autores (2021)].

LINUX - COMPILADOR EM MODO RELEASE					
Análise de Tempo					
Carga Avaliada (quantidade de entradas)					
(tempo em segundos)					
Quantidade	**10**	**100**	**1.000**	**10.000**	**1.000.000**
Média	0,00017	0,00016	0,00031	0,00161	0,13306
Mínimo	0,00013	0,00012	0,00024	0,00141	0,10889
Máximo	0,00023	0,00023	0,00056	0,00205	0,17561
Desvio	0,00003	0,00003	0,00007	0,00019	0,01435
Moda	0,00020	0,00013	0,00026	Não há repetições	Não há repetições
Mediana	0,00016	0,00015	0,00029	0,00151	0,13521
Erro - Para Int. de Confiança de 95%	0,00001	0,00001	0,00003	0,00007	0,00514

Figura 9. Quadro de Análise de Tempo - Linux - Modo Release [Fonte: Os autores (2021)].

Para a avaliação do intervalo de confiança, obtiveram-se valores muito próximos de zero, com os valores sendo demonstrados na terceira e quarta casa decimal. Isto indica que estatisticamente a quantidade de amostras utilizadas foi o suficiente, para termos confiança nos resultados obtidos.

Referências

Salomon, D. (2007). Data Compression (4th ed.). London: Springer.

Pu, I. M. (2006). Fundamental Data Compression. Oxford, United Kingdom: Elsevier.

Blelloch, G. E. (2001). Introduction to Data Compression. In G. E. Blelloch, Algorithms in the Real World. Pittsburg, PA, United States of America: Carnegie Mellon University.

Sayood, K. (2006). Introduction to Data Compression. San Francisco: Morgan Kauffman.

Wikimedia Foundation, Inc. (30/03/2016). Huffman Coding. Acessado em 21/06/2021, de Wikipedia, the free encyclopedia: https://en.wikipedia.org/wiki/Huffman$_coding$

Huffman, D. A. (1952). A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the I.R.E., 1098-1101.

Shannon, C. E. (1948). A Mathematical Theory of Communication. Bell System Technical Journal, 27, 379–423 623–656.

Ribeiro, N. (2016). Apontamentos das Aulas. Multimédia II. Porto: Universidade Fernando Pessoa.

Ferreira, Ronan Loschi Rodrigues, and Fabrício Benevenuto. "Algoritmos para compressão de URLs."

Cormen,T. H. Leiserson, C. E. Rivest, R. L. and C. Stein, Introduction to Algorithms, 3rd ed. The MIT Press, 2009, iSBN-13: 978-0-262-53305-8