Universidade Federal de Juiz de Fora

Departamento de Ciência da Computação

DCC012 A – Estrutura de Dados II 2019.1

Documentação de Código (Parte 2)

Braulio Silva Mendes Lucas

João Victor Dutra Balboa

Marcus Vinícius Vasconcelos de A. Cunha

Professor: Marcelo Caniato Renhe

Juiz de Fora

Julho de 2019

1. Introdução

Este trabalho tem como objetivo documentar o código referente à segunda parte do trabalho da disciplina de Estruturas de Dados II, da Universidade Federal de Juiz de Fora, no primeiro semestre de 2019.

Nele constam a explicação de cada classe do código, assim como alguns métodos e blocos deste; o resultado dos experimentos realizados; e a divisão de tarefas da equipe de desenvolvimento.

2. Requisitos

Para a execução do código em questão, o usuário necessita de um compilador da linguagem C++ (versão 11). Para maior facilidade, o projeto completo foi enviado juntamente com este relatório, assim podendo ser executado no CodeBlocks IDE.

Ao executá-lo deve-se manter os arquivos necessários dentro da pasta na qual se encontra o projeto com extensão ".cbp" e o arquivo "main.cpp", caso seja executado pelo CodeBlocks ou por lnha de comando. Já no caso de ser executado diretamente pelo arquivo ".exe", os arquivos necessários devem estar na pasta do executável. Estes arquivos consistem nas bases de dados "ratings.csv" e "movies_metadata.csv", e nos arquivos de entrada "entrada.txt" e "entrada2.txt", estes últimos devem ser separados pois o número de linhas definido para leitura se difere nos cenários 1 e 2.

O programa gera uma saída para cada *seed* executada. Dentro dos arquivos de saída, encontram-se as estatísticas para cada valor de N definido no arquivo de entrada. O N corresponde ao número de linhas a serem lidas na base.

3. Descrição das classes

O programa possui as seguintes classes:

- **"ArvoreB", "ArvoreVP" e "ArvoreHuffman":** nas quais são implementadas as Árvore B, Árvore Vermelho-Preto e Árvore de *Huffman*, respectivamente. Além do algoritmo de compressão de *Huffman*, que se encontra na classe "ArvoreHuffman".
- **"NoB", "NoVP" e "NoHuffman":** que representam os nós das árvores em questão.
- **"LZW":** implementação do algoritmo de compressão LZW.
- **"LeituraDaBase":** classe que possui diferentes métodos de leitura da base, sendo estes para gravação em um vetor de *strings* ou de objetos.

Os métodos de leitura da base escolhem randomicamente a linha de acordo com a função que lê a linha em questão, se o valor gerado aleatoriamente é divisível por um primo pré-definido (13 para a base "ratings.csv", por possuir mais linhas e 2 para a "movies_metadata.csv"). Devido ao método escolhido para leitura, os valores se encontram na ordem quando esta é feita e, portanto, são embaralhados por uma função auxiliar.

- **“Rating”:** TAD (Tipo Abstrato de Dados) que representa um objeto contendo a identificação do usuário e a identificação do filme que este usuário classificou. São utilizados como chaves para as Árvores B e Vermelho-Preto.
- **“Cenario1” e “Cenario2”:** controlam a execução de cada cenário.

4. Funcionamento do Programa (Fluxo principal e classes)

Ao ser executado, o programa exibe o nome da equipe no terminal e recebe como parâmetro o número de execuções desejadas para cada conjunto N de linhas. Definido esse valo, são executados os cenários 1 e 2. O número de execuções realizadas para criar este relatório foi 5. Após cada execução, são salvas as estatísticas de desempenho de cada algoritmo.

4.1. Cenário 1

No primeiro cenário, a base é lida num vetor de “Ratings”, uma estrutura que possui o *userID* e o *movieID*. Depois de lida, as chaves (*userID* e *movieID*) são inseridas nas Árvores B de grau 2 e 20 e na Árvore VP, e por fim, é feita a busca por essas chaves em cada Árvore. Portanto, os nós das Árvores possuem o objeto “Rating”.

Ambas as Árvores possuem os métodos de inserção, busca e impressão. O atributo em comum é o nó raiz, porém, eles não são a mesma classe, já que se diferem de uma árvore para outra.

4.1.1. Árvore B

A árvore em questão, possui o grau (d), sendo que cada nó possui no mínimo d e no máximo 2d chaves. Os nós possuem um vetor de chaves (“Rating”), um vetor de filhos (“NoB”), uma variável que controla o número de chaves, outra que guarda se ele é folha (booleana) e uma terceira para o grau.

As novas chaves são inseridas sempre nas folhas, são feitas comparações as chaves dos nós partindo da raiz, até chegar no nó folha no qual se deve inserir. Estas comparações são feitas primeiramente pelo *userID* e caso necessite, pelo *movieID*. Ou seja, se a chave a ser inserida possui um *userID* semelhante a alguma chave da árvore, as comparações continuam pelo *movieID*. No caso de o nó já estar cheio, é feita a cisão (*split*).

A busca é semelhante à da Árvore Binária de Busca, porém as chaves são um conjunto de *userID* e *movieID*. Portanto busca-se o id do usuário semelhante e depois o id do filme, garantindo assim que a chave seja esperada. Pois se a busca fosse feita em outra ordem, ocorreriam erros: usuários diferentes qualificam vários filmes, que podem ser iguais, assim, a busca poderia encontrar um id do filme correspondente ao buscado, mas não encontrar o id do usuário, mesmo se este existisse na árvore.

4.1.2. Árvore Vermelho-Preto

Já a Árvore VP possui apenas o ponteiro para o nó raiz e os nós possuem a chave(“Rating”), a cor (variável booleana) e os ponteiros para o nó pai, o filho à esquerda e à direita.

As chaves são inseridas inicialmente pelo algoritmo de inserção da Árvore Binária de Busca, depois disso é feito o ajuste necessário através de rotações e métodos de “recolorir”. A busca é a mesma da *BST* (*Binary Search Tree*), que assim como na inserção, possui apenas adaptações para as chaves sendo um conjunto.

4.2. Cenário 2

No segundo cenário a base é lida numa *string* a ser comprimida pelos algoritmos de *Huffman* e *LZW*. Esta *string* possui o número N de sinopses definidas.

Os atributos e métodos em comum são: código e chave como atributos; métodos para codificar, decodificar, gerar dicionário e salvar código de compressão em arquivo. Porém, todos estes métodos se diferem em sua implementação.

4.2.1. Árvore de *Huffman*

Na Árvore de *Huffman*, além dos métodos citados acima ainda existem os de calcular a frequência de cada caractere e de gerar a tabela de caracteres e seus códigos.

Seu funcionamento consiste em calcular a frequência de cada caractere existente na sinopse (chave do tipo *string* passada como parâmetro) e montar a árvore a partir dos nós de menor frequência, fazendo com que estes tenham um código maior.

4.2.2. LZW

Já no *LZW*, existe o método que constrói o dicionário através da tabela *ASCII*. Seu funcionamento consiste em adicionar novos códigos no dicionário, representando uma combinação da *string* de leitura atual e o próximo caractere. Portanto, combinações semelhantes possuem o mesmo código.

5. Análise de resultados

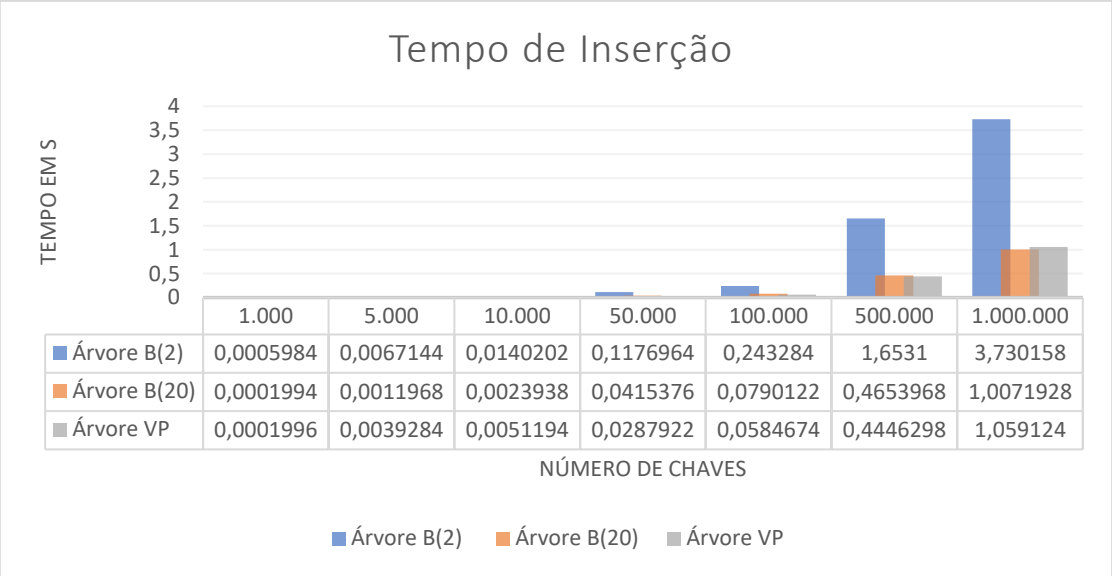
5.1. Cenário 1

Neste cenário, a Árvore VP se destaca, tendo desempenho superior em todos os aspectos (tempo de inserção, comparações de chaves na inserção, tempo de busca e comparação de chaves na busca).

Na inserção, sua superioridade se explica pelo funcionamento desta ser semelhante ao da *BST*, realizando apenas correções para manter suas propriedades (rotações e “recolorimentos”). Nestas correções, não são comparadas mais as chaves, mas sim a cor de cada nó, portanto, estas comparações não entram na estatística.

Seu tempo de inserção é muito semelhante ao da Árvore B de grau 20 (Figura 1), pois nesta segunda, os vetores de chaves dos nós são grandes, fazendo com que sejam feitas comparações dentro do mesmo vetor, sem muitos saltos entre os nós. Isso explica o fato dela reduzir em poucos milésimos ou centésimos, no tempo de inserção em relação a

Árvore VP. Já em relação a B de grau 2, o tempo de inserção é sempre superior devido ao fato de o nó possuir um pequeno vetor para as chaves, sendo necessário fazer muito mais cisões.



	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Árvore B(2)	0,0005984	0,0067144	0,0140202	0,1176964	0,243284	1,6531	3,730158
Árvore B(20)	0,0001994	0,0011968	0,0023938	0,0415376	0,0790122	0,4653968	1,0071928
Árvore VP	0,0001996	0,0039284	0,0051194	0,0287922	0,0584674	0,4446298	1,059124

Figura 1: Estatísticas de tempo de inserção.

Em relação às comparações na inserção, a Figura 2 mostra que a Árvore B de grau 20 apresenta os maiores números, exatamente pelo fato de ter muitas chaves em um nó. Porém, seu tempo de inserção ainda é baixo, pois a maioria destas comparações são feitas dentro do vetor.

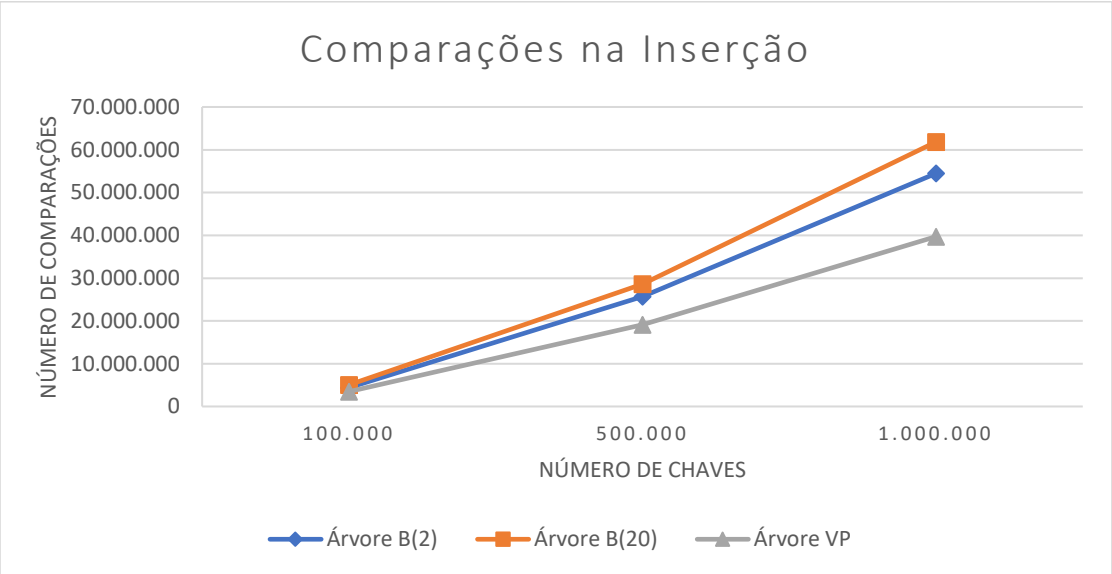
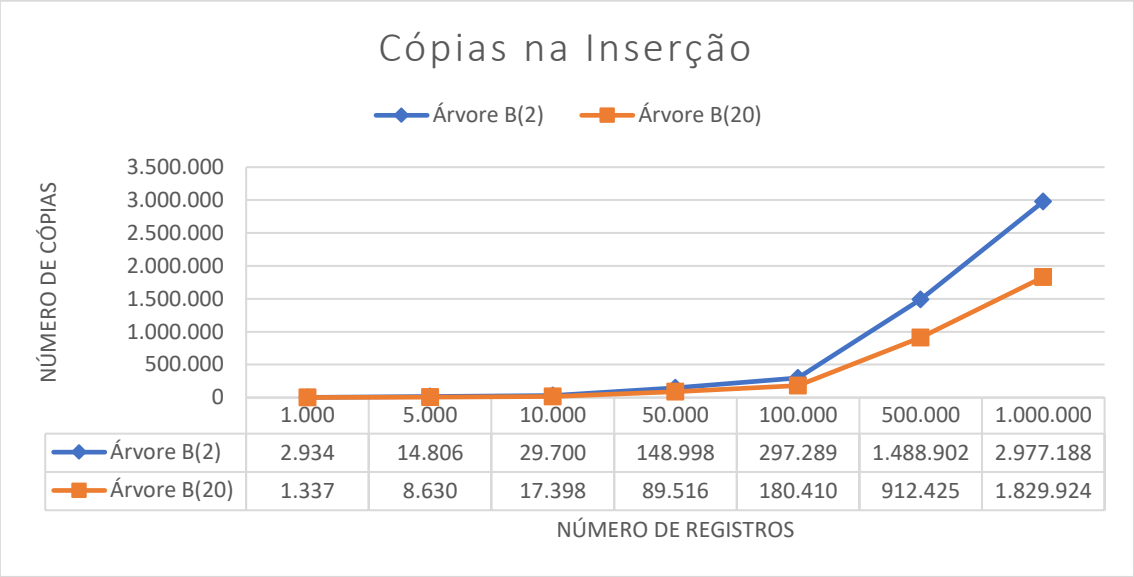


Figura 2: Número de comparações de chaves na inserção.

O que também ajuda a explicar o aumento no tempo de inserção da Árvore B(2) é o número de cisões, que pode ser observado através do número de cópias de registro (Figura 3). Esta árvore realiza muitas cisões, portanto, faz muita cópia de registro. Já a B(20) também faz um número elevado de cópias devido ao tamanho do vetor, por ser

grande, mesmo fazendo menos cisões, quando ela é feita são copiadas muitas chaves de um nó para o outro.

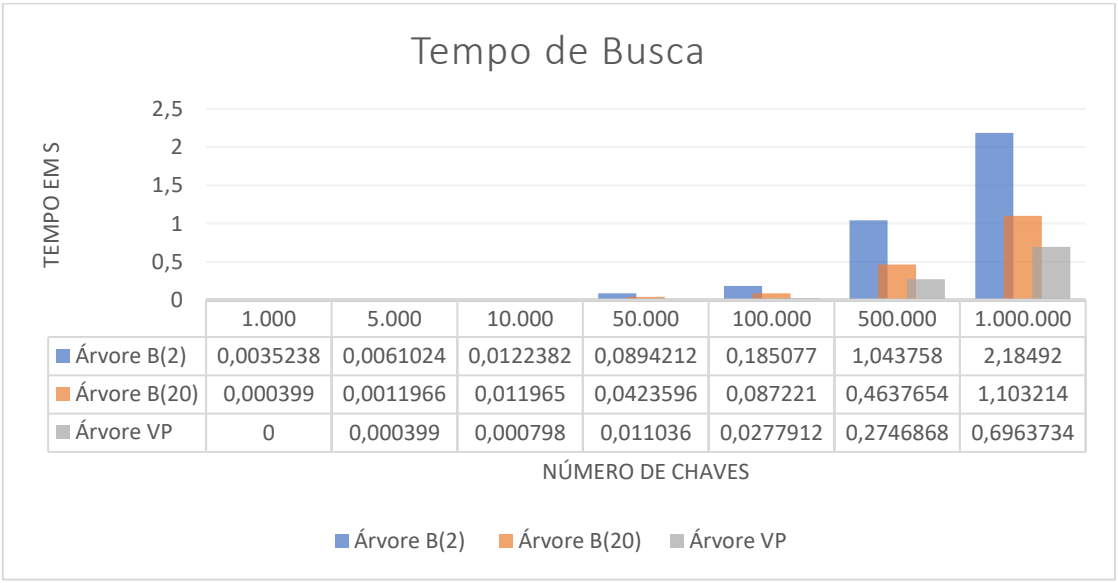


	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Árvore B(2)	2.934	14.806	29.700	148.998	297.289	1.488.902	2.977.188
Árvore B(20)	1.337	8.630	17.398	89.516	180.410	912.425	1.829.924

Figura 3: Número de cópias de registo na inserção.

OBS: A Árvore VP não está presente na figura acima por não realizar cópia de chaves, ela faz apenas a troca de "funções" (e.g. o nó que era pai passa a ser filho).

Já na busca (Figura 4), a Árvore VP tem uma superioridade mais visível no quesito tempo, comparado ao de inserção. Sua propriedade que torna isto possível é exatamente a busca binária. A B(20) não fica muito atrás, mas seu tempo um pouco superior se explica pelo tamanho dos vetores. Porém a B(2), possui tempo maior por conta do grande número de nós, gerando muitos saltos, mesmo tendo menos comparações de chaves que a B(20), como mostra a Figura 5.



	1.000	5.000	10.000	50.000	100.000	500.000	1.000.000
Árvore B(2)	0,0035238	0,0061024	0,0122382	0,0894212	0,185077	1,043758	2,18492
Árvore B(20)	0,000399	0,0011966	0,011965	0,0423596	0,087221	0,4637654	1,103214
Árvore VP	0	0,000399	0,000798	0,011036	0,0277912	0,2746868	0,6963734

Figura 4: Tempo na busca de chaves.

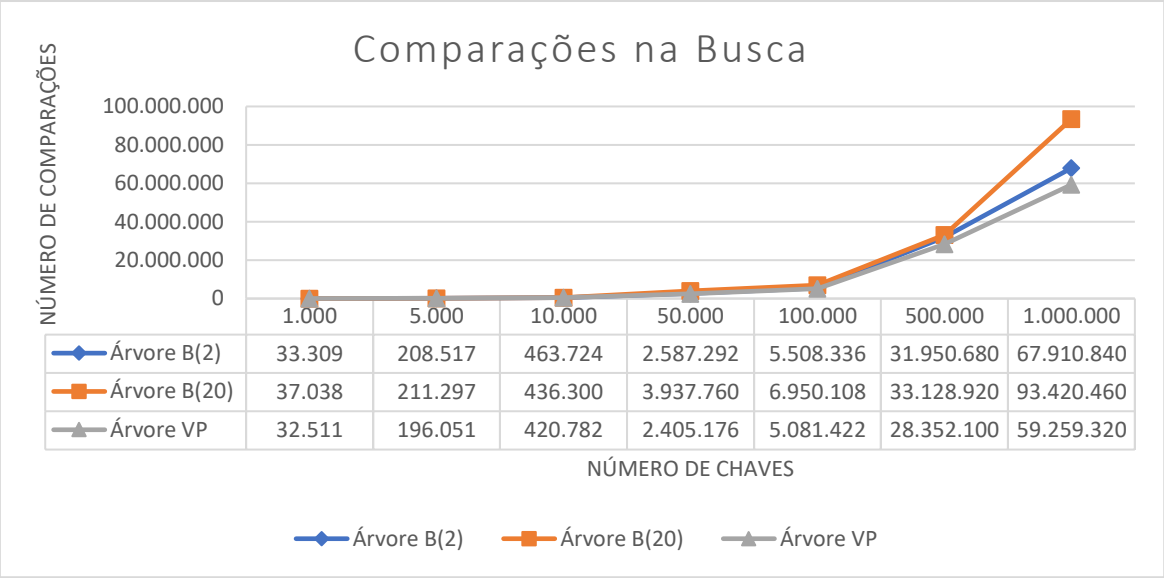


Figura 5: Número de comparações de chaves na busca.

Apesar, da B(20) realizar mais comparações, seu tempo ainda é menor do que ao da B(2) devido aos fatos explicados acima.

5.2. Cenário 2

No cenário 2, foram analisadas estatísticas de desempenho dos algoritmos de compressão de acordo com 3 métricas:

- Tempo para compressão;
- Tamanho do arquivo gerado; e
- Taxa de compressão.

Em relação ao tempo de compressão, o algoritmo de *Huffman* supera o *LZW*, por realizar o percurso em árvore, enquanto o *LZW* busca o padrão no dicionário toda iteração. A Figura 6 mostra a comparação dos tempos.

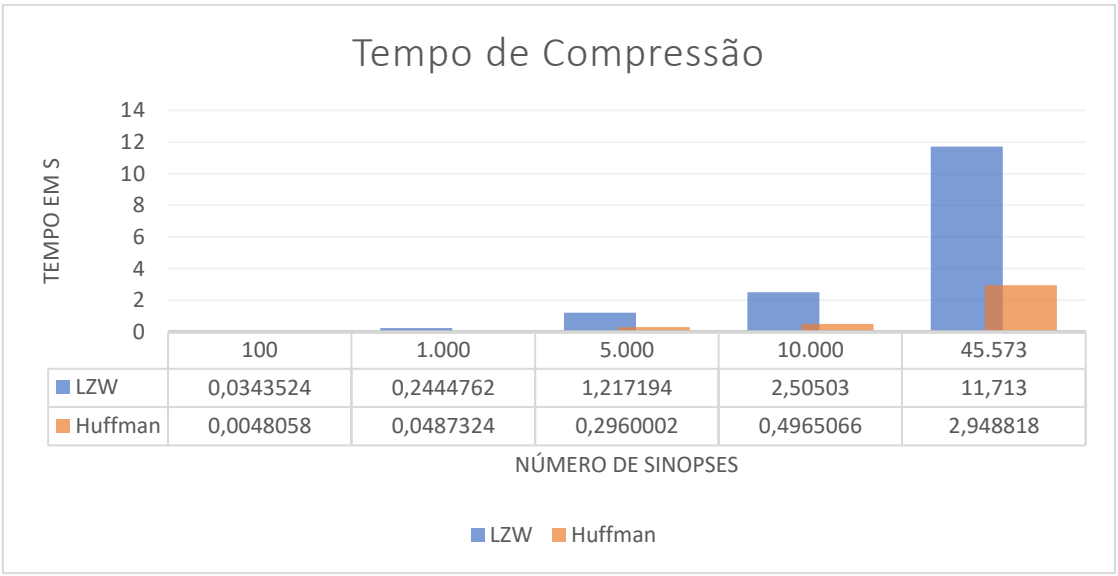


Figura 6: Tempo para compressão dos arquivos.

Comparando o tamanho do arquivo de texto gerado, o algoritmo de *Huffman* perderia até para o arquivo original, pois gera mais caracteres do que este. Porém, como seus códigos são binários (0 e 1), podem ser representados como *bits*, ocupando 8 vezes menos espaço do que um caractere comum (que ocupa 1 *byte*, ou seja, 8 *bits*). Portanto, ele é capaz de comprimir mais o arquivo do que o LZW (Figura 7). Já este, para um número pequeno de sinopses, gera arquivos maiores do que o próprio original, por criar mais códigos inteiros (que ocupam o mesmo tamanho que um caractere). Mas à medida que o número de sinopses cresce, ele começa a comprimir o arquivo, devido ao fato de o arquivo original passar a ter mais combinações de caracteres, que são o foco da compressão do *LZW*.

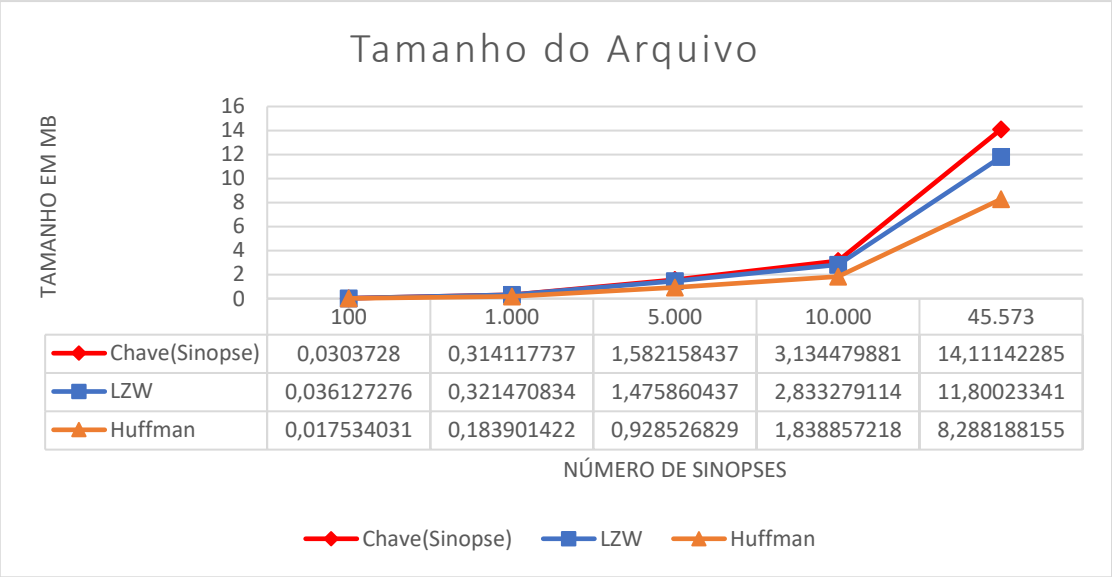


Figura 7: Tamanho dos arquivos comprimidos em relação ao original.

Conforme relatado, é possível perceber que para 100 e 1.000 sinopses, o arquivo gerado pelo *LZW* é maior que o original, mas a partir de 5.000, eles passam a ser menores.

Por fim, uma métrica relacionada ao tamanho dos arquivos é a taxa de compressão, que pode ser calculada dividindo-se o número de *bytes* do arquivo comprimido pelo original. A Figura 8 mostra essa taxa, que representa a porcentagem do arquivo comprimido em relação ao original (e.g. se o arquivo original pesa 1,582158437 *megabytes*, que significa 100%, o arquivo comprimido 0,928526829 representa 58,6% do total).

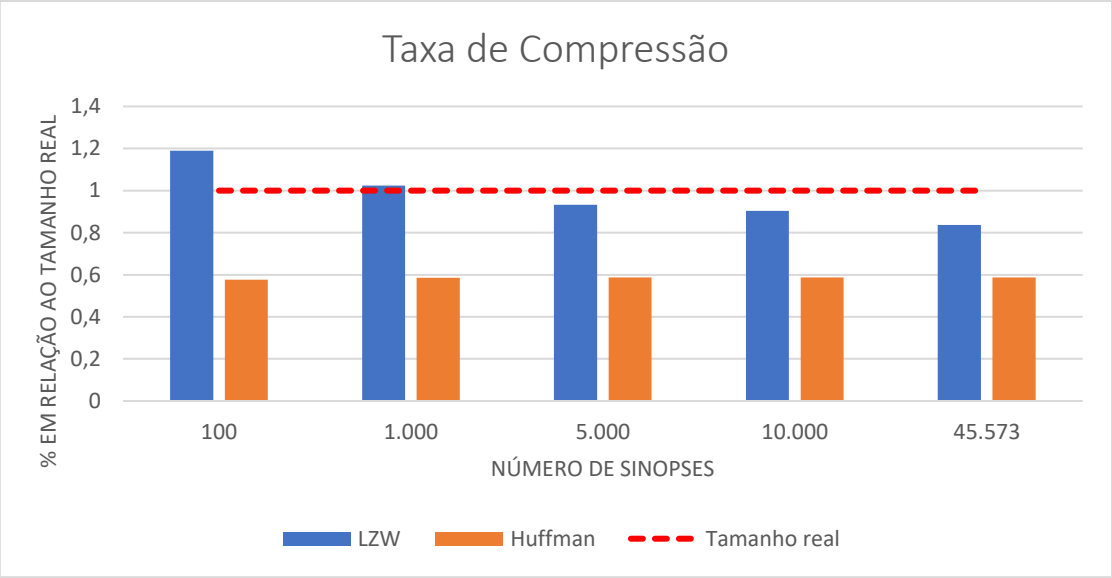


Figura 8: Taxa de compressão dos arquivos gerados pelos diferentes algoritmos.

É possível perceber que nos dois primeiros casos o arquivo comprimido pelo algoritmo *LZW* é maior do que o arquivo original, conforme explicação acima.

6. Divisão de tarefas

A divisão de tarefas desta etapa do trabalho consta a seguir:

- Braulio: implementação da Árvore VP, do controle do cenário 2 e escrita do relatório.
- João Victor: implementação da Árvore B, do controle do cenário 1 e escrita do relatório.
- Marcus: implementação dos algoritmos de compressão (*Huffman* e *LZW*) e escrita do relatório.

7. Conclusão

Pode-se concluir com este trabalho, que a escolha do algoritmo a ser usado na resolução de problemas é muito importante. Pois cada um se comporta melhor com um tipo de padrão, a Árvore B por exemplo, para armazenar uma quantidade grande de dados; a VP quando inserções e remoções são muito frequentes; *Huffman* quando existe um número grande caracteres repetidos; e *LZW* para grande quantidade de dados, com combinações repetidas de caracteres.

É extremamente necessário que se conheça cada tipo de estrutura e os casos em que elas se sobressaem, a fim de conseguir utilizar as vantagens de cada uma de acordo com a aplicação e a necessidade.