MapMatrix

José Victor Santana Barbosa - 245511

21 de Novembro 2025

Conteúdo

1 Resumo

O seguinte trabalho, apresenta sobre a implementação de uma estrutura de dados chamada MapMatrix, que representa matrizes esparsas utilizando mapas associativos. A estrutura é genérica e pode ser utilizada com diferentes tipos de armazenamento subjacente, desde possuam interface com certas operações básicas. A seguir, serão detalhadas as principais funcionalidades e características da estrutura MapMatrix.

2 Introdução

MapMatrix parte da ideia, de ao invés de representar representar uma matriz como um vetor bidimensional, onde a posição na memoria física do computador tem relação direta com a posição logica da matriz, utilizar uma estrutura de dados chamada mapa associativo (ou dicionario), onde cada elemento da matriz é armazenado como um par chave-valor, onde a chave é a posição do elemento na matriz (representada como uma tupla de índices) e o valor é o valor do elemento na matriz.

2.1 Iteradores

Um recurso que é vastamente utilizado neste projeto são iteradores[1], que servem para percorrer os valores de alguma estrutura de dados. Na interface de um Iterador é obrigatória a existência de apenas uma única operação, que é a operação de *next()*. Essa operação, que caso haja algum valor ainda no iterador, retorna o próximo valor; e, caso não haja mais nenhum valor disponível, retorna vazio.

2.2 Mapas

Um mapa associativo é uma estrutura de dados que armazena pares chave-valor, permitindo a associação de um valor a uma chave única. Para um mapa associativo ser usado em MapMatrix é necessário que ele possua as seguintes operações:

- **new()**: Cria um novo mapa vazio.
- **get(key)**: Retorna o valor associado a chave key, ou um valor padrão caso a chave não exista no mapa.
- **set_or_insert(key, value)**: Insere o par chave-valor (key, value) no mapa. Se a chave ja existir, atualiza o valor associado a chave.
- **remove(key)**: Remove o par chave-valor associado a chave key do mapa.
- **iter() / iter**$_mut()$: $Retornaumiteradorsobreospareschave - valordomapa.$
- **from_iter(iter): Cria um mapa a partir de um iterador que produz pares chave-valor.**
- **clone(): Faz uma copia completa de toda a estrutura.**

3 MapMatrix

A estrutura MapMatrix para ser definida e usada é preciso de dois tipos de mapas auxiliares, um Mapa que armazena os valores da matriz, sendo a chave a posição do elemento e o valor o valor do elemento, e o outro mapa é usando na operação de multiplicação de matrizes, onde é um mapa de índices (colunas ou linhas) para lista de valores. Antes de detalhar as operações da estrutura MapMatrix, primeiro vamos falar sobre um tipo especial de mapa associativo que é usado na implementação, chamado TransposableMap.

[1] https://doc.rust-lang.org/std/iter/trait.Iterator.html

3.1 TransposableMap

Para que execução eficiente de transposição de matrizes, foi criado uma estrutura chamada TransposableMap, que ela basicamente encapsula um mapa associativo com chaves sendo pares de índices e adiciona a funcionalidade de transposição.

Além das funcionalidades básicas de um Mapa Associativo, TransposableMap adiciona a funcionalidade de transpor, que faz inversão de um uma flag interna indicando que a matriz foi transposta.

A partir da flag interna, todas as operações sobre o mapa, é verificado se a flag está ativa, caso sim, inverte os índices das chaves antes de realizar a operação e ao retornar os valores. Caso não, executa as operações normalmente.

A partir dessa estrutura, é possível implementar a operação de transposição de uma matriz em tempo constante, apenas invertendo a flag interna, sem a necessidade de iterar sobre todos os elementos da matriz e trocar suas posições. E sem ser necessário dentro da estrutura MapMatrix, adicionar lógica extra para lidar com a transposição. Já que a o único ponto de entrada e saída de dados da matriz é o mapa encapsulado.

3.2 Operações de MapMatrix

A seguir, são detalhadas as principais operações implementadas na estrutura MapMatrix: Essa sessão as equações são um pouco grandes, mas isso é por causa que é feito de forma genérica, então depende de como o Mapa é implementado, em sessões posteriores as complexidades são descritas de forma mais simples. Vamos usar a seguinte notação

1. k, k_A, k_B: Numero de elementos não nulos na matriz.
2. $|A_C|$, $|B_L|$: Numero de colunas em A, e o numero de linhsa em B.
3. $|A_{c=i}|$: Numero de elementos em uma coluna i na matriz A
4. $|B_{l=i}|$: Numero de elementos em uma linhha i na matriz B
5. $M_{nome}(k)$ Tempo para execução de uma operação no Mapa com o nome especificado
6. $LM_{nome}(k)$ Tempo para execução de uma operação no Mapa de Vetor com o nome especificado
7. $U(k)$ Tempo para execução de uma operação no Mapa com o nome especificado

nota: para complexidade de memoria será considerado o custo de chamada das operações em Map é desprezível.

Será usado a seguinte

3.2.1 Acessar elemento

O acesso de elemento é simples, é basicamente uma leitura do mapa associativo encapsulado, passando a posição do elemento como chave. Caso o elemento não exista na matriz, é retornado 0.

1. **Complexidade de tempo:** $M_{get}(k)$,
2. **Complexidade de espaço:** $\Theta(1)$,

3.2.2 Transpor matriz

A operação de transpor matriz, apenas inverte o tamanho da linha/coluna e troca a flag de transposição, então tem custo constante.

1. **Complexidade de tempo:** $\Theta(1)$,
2. **Complexidade de espaço:** $\Theta(1)$,

3.2.3 Multiplicação por escalar

A multiplicação por escalar é feita copiando o mapa, e então usando um iterador mutável, para cada valor multiplicação pelo escalar. E então retorna esse novo valor

1. **Complexidade de tempo:**
$$M_{clone} + M_{iter}(k)$$
, A complexidade de tempo é o custo de clonar e de iterar mutiplicando cada valor da matriz
2. **Complexidade de espaço:**
$$U(k)$$
, O custo de espaço é o custo da nova matriz criada.

3.2.4 Soma de matrizes

A soma de matrizes primeiro clona A criando a matriz C, então, itera por B e obtêm o valor de A na posição soma e guarda em C.

1. **Complexidade de tempo:**

$$M_{clone}(k_A) + M_{iter}(k_B) + k_B(M_{set}(k_C) + M_{get}(k_A))$$

   A complexidade de tempo é o custo de copiar A, e o custo para cada valor de B, ler de A e escrever em C

2. **Complexidade de espaço:**

$$U(k_C) + U_{set}(k) + U_{set}(k)$$

   , O custo de espaço é o custo da nova matriz criada, mas o custo de espaço para executar a operação de set e get.

3.2.5 Mutiplicação de matrizes

Para a mutiplicação de matriz temos como base a seguinte formula: $C_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$ E como é perceptível, para mutiplicação de matriz, os elementos na numa coluna k de A só será mutiplicado com elementos de uma linha k em B, significa.

Portanto, o que fazemos, é então, é primeiro separar os elementos de A por linha, e o de B por coluna, e então multiplicamos cada linha pela respectiva coluna e somamos em C.

Que o algoritmo possui duas partes principais:

Separação: É criado um mapa das colunas de A e um das linhas de B, e então iteramos em A adicionando nas sua respectiva coluna, e iteramos em B os adicionando nas suas respectivas $linhas$.

Multiplicação: Então, para cada linha de A é feito a leitura da coluna correspondente em B, e então iterado em cada um dos elementos na linha e coluna, e multiplicados e somados a C

1. **Complexidade de tempo:** $|A_C|$ representa a quantidade de colunas significativas em A, $|B_L|$ representa a quantidade de colunas significativas em B

   (a) **Separação:**

$$M_{iter}(k_A) + M_{iter}(k_B) + k_A LM_{add}(|A_C|) + k_B LM_{add}(|B_L|)$$

   A complexidade de tempo é o custo de iterar em cada uma das matrizes, mais o custo de para cada um dos valores adicionar um elemento a mais, onde o tamanho de cada um dos mapa de vetores, é a quantidade de colunas relevantes em A e a quantidade de colunas relevantes em B

   (b) **Multiplicação:** Na multiplicação esse é o custo para iterar em cada coluna de A e ler uma linha em B

$$LM_{iter}(|A_C|) + |A_C|LM_{get}(|B_L|)$$

   Como guardamos cada elemento de cada linha/coluna num vetor, sabemos que tem um custo linear iterar neles, e o custo da nossa multiplicação se dá pela quantidades de elementos na coluna ($A_{c=n}$) e a quantidade de elementos da linha ($B_{l=n}$). Porque iremos fazer cada uma das combinações de elementos na linha e coluna, e para cada combinação iremos ler de ser e atribuir um novo valor, portanto:

$$\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|(M_{get}(k_C) + M_{set}(k_C))$$

   .

2. **Complexidade de espaço:**

$$U(k_C) + U(k_A) + U(k_B)$$

   , O custo de espaço é o custo da nova matriz criada, mais o custo de manter o mapa auxiliar de colunas e linhas.

4 Implementações

4.1 HashMapMatrix

Essa é a primeira estrutura de dados solicitada, que é a execução de uma MapMatrix usando um HashMap como Mapa.

Essa estrutura usa uma Tabela Hash, especificamente a Tabela Hash da biblioteca padrão do Rust[2], que é um porte da tabela SwissTable[3] desenvolvida pelo Google.

[2]https://doc.rust-lang.org/std/collections/struct.HashMap.html

[3]https://abseil.io/blog/20180927-swisstables

Para a maior parte das operações que usamos do HashMap a complexidade está definida na propria documentação [4].

Na documentação explicita que as operações de *get*, *insert* e *remove*, possuem complexidade esperada $O(1)$. Que é o caso onde não ocorrem muitas colisões de hash, que é o caso que vamos estar considerando aqui.

Em relação a iterar em todos valores de uma hash table, é especificado na documentação[5], que na iteração é visitado também os buckets vazios, então é $O(capacidade)$ e não $O(tamanho)$ de elementos no hashset, porém como é uma implementação de hashtable, e a capacidade está crescendo junto com a adição de elementos, e não definida manualmente, é razoavel considerar que a implementação usa mecanismos de quando ultrapassar o fator de carga, a capacidade aumentar por um fator de crescimento, e quando é removido uma certa quantidade de elementos é também feito reconstrução da tabela liberando memoria, portanto vou considerar que é razoável especificar que $capacidade \in \Theta(tamanho)$. (na parte de metricas será visto na pratica que é isso que acontece)

Em relação a operação de clone, o possui basicamente a mesma questão da iteração, que basicamente copia toda a tabela e isso é relativo a capacidade da tabela.

Portanto temos que

- **new()**: $\Theta(1) \sim$
- **get(key)**: $\Theta(1) \sim$
- **set_or_insert(key, value)**: $\Theta(1)* \sim$
- **remove(key)**: $\Theta(1)* \sim$
- **iterar todos elementos**: $\Theta(k)$
- **from_iter(iter)**: $\Theta(k) \sim$
- **clone()**: $\Theta(k)$

$\sim$ significa complexidade esperada, * significa complexidade amortizada.

Portando, agora sobre a complexidade de cada uma das operações

E para operação de adição a um vetor no caso de mapa de vetores, como é apenas um get, então também é constante

4.1.1 Acessar elemento

1. **Complexidade de tempo esperado:** $\Theta(1)$,
2. **Complexidade de espaço:** $\Theta(1)$,

4.1.2 Transpor matriz

1. **Complexidade de tempo esperado:** $\Theta(1)$,
2. **Complexidade de espaço:** $\Theta(1)$,

4.1.3 Multiplicação por escalar

1. **Complexidade de tempo esperado:**
$$\Theta(k) + \Theta(k)$$
$$\Theta(k)$$
2. **Complexidade de espaço:** $\Theta(k)$,

4.1.4 Soma de matrizes

1. **Complexidade de tempo esperado:**
$$M_{clone}(k_A) + M_{iter}(k_B) + k_B(M_{set}(k_C) + M_{get}(k_A))$$
$$\Theta(k_A) + \Theta(k_B) + k_B(\Theta(1) + \Theta(1))$$
$$\Theta(k_A + k_B)$$
2. **Complexidade de espaço:**
$$\Theta(k_C)$$

,

[4]https://doc.rust-lang.org/std/collections/#cost-of-collection-operations

[5]https://doc.rust-lang.org/std/collections/struct.HashMap.html#performance-5

4.1.5 Mutiplicação de matrizes

1. **Complexidade de tempo esperado:** $|A_C|$ representa a quantidade de colunas significativas em A, $|B_L|$ representa a quantidade de colunas significativas em B

   (a) **Separação:**

   $$M_{iter}(k_A) + M_{iter}(k_B) + k_A LM_{add}(|A_C|) + k_B LM_{add}(|B_L|)$$
   $$\Theta(k_A) + \Theta(k_B) + k_A\Theta(1) + k_B\Theta(1)$$
   $$\Theta(k_A + k_B)$$

   (b) **Multiplicação:** Na multiplicação esse é o custo para iterar em cada coluna de A e ler uma linha em B

   $$LM_{iter}(|A_C|) + |A_C|LM_{get}(|B_L|)$$
   $$\Theta(|A_C|) + |A_C|\Theta(1)$$
   $$\Theta(|A_C|)$$

   Custo para mutiplicar cada linha por cada coluna

   $$\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|(M_{get}(k_C) + M_{set}(k_C))$$

   $$\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|(\Theta(1) + \Theta(1))$$

   $$\Theta(1)\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|$$

   Para calcularmos a esperança, vamos usar as seguintes duas variaveis indicadores:

   $$X_{a,i} = \begin{cases} 1 & \text{, se coluna de } a \text{ é } i \\ 0 & \text{, se não} \end{cases}$$

   $$Y_{b,i} = \begin{cases} 1 & \text{, se linha de } b \text{ é } i \\ 0 & \text{, se não} \end{cases}$$

   No qual, como está uniformemente distribuído, a chance de uma variável estar em uma linha/coluna de tamanho n é

   $$E[X_{a,i}] = E[Y_{b,i}] = \frac{1}{n}$$

   Então definimos

   $$|A_{c=i}| = \sum_{a \in A} X_{a,i}$$
   $$|B_{l=i}| = \sum_{b \in B} Y_{b,i}$$

   Logo

   $$E[|A_{c=n}|] = \sum_{a \in A} E[X_{a,i}] = \frac{1}{n}\sum_{a \in A} 1 = \frac{k_A}{n}$$
   $$E[|B_{l=n}|] = \sum_{b \in B} E[Y_{b,i}] = \frac{1}{n}\sum_{b \in B} 1 = \frac{k_B}{n}$$

   Então reescrevemos a complexiade esperada dessa parte da seguinte forma:

   $$\mathrm{E}\left[\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|\right] = \sum_{i=1}^{n} \mathrm{E}\left[|A_{c=i}|\right]\mathrm{E}\left[|B_{l=i}|\right]$$

   Logo

   $$\sum_{i=1}^{n} \mathrm{E}\left[|A_{c=i}|\right]\mathrm{E}\left[|B_{l=i}|\right] = \sum_{i=1}^{n} \frac{k_A}{n}\frac{k_B}{n} = n\frac{k_A}{n}\frac{k_B}{n} = \frac{k_A k_B}{n}$$

   Logo então a complexidade esperada de mutilicar as linhas pelas colunas é:

   $$\Theta(|A_C| + \frac{k_A k_B}{n})$$

Juntando então cada parte temos que a complexidade total da mutiplicação de matriz é

$$\Theta\left(k_A + k_B + |A_C| + \frac{k_A k_B}{n}\right)$$

Podemos considerar definir a media de elementos por linha em B,

$$d_B = \frac{k_B}{n}$$

E também ignorar $|A_C|$ porque $k_A > |A_C|$ Então temos:

$$\Theta\left(k_B + k_A d_B\right)$$

E como $d_B < k_B$

$$k_B + k_A d_B \in O(k_A k_B)$$

2. **Complexidade de espaço:**

$$U(k_C) + U(k_A) + U(k_B)$$

, O custo de espaço é o custo da nova matriz criada, mais o custo de manter o mapa auxiliar de colunas e linhas.

4.2 TreeMapMatrix

A segunda estrutura de dados, com requisitos garantidos, foi implementada usando o MapMatrix com o Mapa associativo usado para guardar os dados sendo uma BTree.

Como o HashMap, foi implementado usando a implementação padrão de BTree da biblioteca padrão no Rust[6]. E possui complexidade também documentada para cada operação [7] exceto a de iterar sobre todos valores.

Mas para a operação de iteração na pagina sobre BTree[8], é descrito que a operação de produzir o próprio valor no iterador tem custo $\Theta(\log n)$ no pior caso, mas amortizado possui tempo constante por item $\Theta(1)$, portanto, para ler todos valores, temos tempo linear.

Para a operação de clone é possivel observar o codigo fonte[9] e ver que é feito clone recursivamente da arvore, portanto trivialmente é linear em relação ao tamanho da arvore.

Portanto temos que

- **new()**: $\Theta(1)$
- **get(key)**: $\Theta(\log k)$
- **set_or_insert(key, value)**: $\Theta(\log k)$
- **remove(key)**: $\Theta(\log k)$
- **iterar todos elementos**: $\Theta(k)$
- **from_iter(iter)**: $\Theta(k)$
- **clone()**: $\Theta(k)$

Portando, agora sobre a complexidade de cada uma das operações

E para operação de adição a um vetor no caso de mapa de vetores, como é apenas um get, então também é constante

4.2.1 Acessar elemento

1. **Complexidade de tempo:** $\Theta(\log k)$,
2. **Complexidade de espaço:** $\Theta(1)$,

[6] https://doc.rust-lang.org/std/collections/struct.BTreeMap.html
[7] https://doc.rust-lang.org/std/collections/#cost-of-collection-operations
[8]
[9] https://doc.rust-lang.org/stable/src/alloc/collections/btree/map.rs.html224

4.2.2 Transpor matriz

1. **Complexidade de tempo:** $\Theta(\log k)$,
2. **Complexidade de espaço:** $\Theta(1)$,

4.2.3 Multiplicação por escalar

1. **Complexidade de tempo:**

$$\Theta(k) + \Theta(k)$$
$$\Theta(k)$$

2. **Complexidade de espaço:** $\Theta(k)$,

4.2.4 Soma de matrizes

1. **Complexidade de tempo:**

$$M_{clone}(k_A) + M_{iter}(k_B) + k_B(M_{set}(k_C) + M_{get}(k_A))$$
$$\Theta(k_A) + \Theta(k_B) + k_B(\Theta(\log k_C) + \Theta(\log k_A))$$
$$\Theta(k_A + k_B + k_B(\log k_C + \log k_A))$$

Sabendo que $k_B \in O(k_B \log k_C)$ A complexidade final é a seguinte:

$$\Theta(k_A + k_B \log k_C)$$

E como $k_A \in O(k_A \log k_C)$ Então

$$k_A + k_B \log k_C \in O((k_A + k_B)k_C)$$

2. **Complexidade de espaço:**

$$\Theta(k_C)$$

,

4.2.5 Mutiplicação de matrizes

1. **Complexidade de tempo:** $|A_C|$ representa a quantidade de colunas significativas em A, $|B_L|$ representa a quantidade de colunas significativas em B

   (a) **Separação:**

$$M_{iter}(k_A) + M_{iter}(k_B) + k_A LM_{add}(|A_C|) + k_B LM_{add}(|B_L|)$$
$$\Theta(k_A) + \Theta(k_B) + k_A\Theta(\log |A_C|) + k_B\Theta(\log |B_L|)$$
$$\Theta(k_A \log |A_C| + k_B \log |B_L|)$$

   (b) **Multiplicação:** Na multiplicação esse é o custo para iterar em cada coluna de A e ler uma linha em B

$$LM_{iter}(|A_C|) + |A_C|LM_{get}(|B_L|)$$
$$\Theta(|A_C|) + |A_C|\Theta(\log |B_L|)$$
$$\Theta(|A_C| \log |B_L|)$$

   Custo para mutiplicar cada linha por cada coluna

$$\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|(M_{get}(k_C) + M_{set}(k_C))$$

$$\sum_{i=1}^{n} |A_{c=i}||B_{l=i}|(\Theta(\log k_C) + \Theta(\log k_C))$$

$$\Theta(\log k_C) \cdot \sum_{i=1}^{n} |A_{c=i}||B_{l=i}|$$

Onde o nosso pior caso é quanto todos elementos de $|A|$ estão em uma única coluna, e todos elementos de B estão em uma única linha. Onde o somatorio resulta em $k_A k_B$

Podemos provar isso por absurdo, considerando que existe alguma distribuição $|A'_{c=i}|$ de elementos em A e $|B'_{c=i}|$ de elementos em B, que o pior caso. Poranto para essa sequencia ser pior que todos em uma unica linha/coluna, significa então:

$$\sum_{i=1}^{n} |A'_{c=i}||B'_{l=i}| > k_A k_B$$

E como

$$k_A = \sum_{i=1}^{n} |A'_{c=i}|$$

$$k_B = \sum_{i=1}^{n} |B'_{l=i}|$$

Logo isso significa que:

$$\begin{aligned}
\sum_{i=1}^{n} |A'_{c=i}||B'_{l=i}| &> k_A k_B \\
&> \left(\sum_{i=1}^{n} |A'_{c=i}|\right)\left(\sum_{i=1}^{n} |B'_{l=i}|\right) \\
&> \sum_{i=1}^{n}\sum_{i=1}^{n} |A'_{c=i}||B'_{l=j}| \\
&> \sum_{i=1}^{n} |A'_{c=i}||B'_{l=i}| + \sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} |A'_{c=i}||B'_{l=j}|
\end{aligned}$$

Portanto:

$$0 > \sum_{i=1}^{n}\sum_{j=1,j\neq i}^{n} |A'_{c=i}||B'_{l=j}|$$

Que é impossível porque uma soma de tamanhos de conjunto deve ser no mínimo positiva. Portanto é uma contradição e não existe uma distribuição que o resultado seja maior.

Logo chegamos que a complexidade da mutiplicação das colunas pelas linhas no pior caso é quando quando todos elementos de B estão numa linha apenas e todos de A estão numa coluna só logo:

$$\Theta\left(|A_C| \log |B_L| + k_A k_B \log k_C\right)$$

$$\Theta\left(1 \log 1 + k_A k_B \log k_C\right)$$

$$\Theta\left(k_A k_B \log k_C\right)$$

Juntando então cada parte temos que a complexidade total da mutiplicação de matriz é

$$\Theta\left(k_A \log |A_C| + k_B \log |B_L| + k_A k_B \log k_C\right)$$

E como no pior caso tudo estar numa coluna/linha apenas:

$$\Theta\left(k_A \log 1 + k_B \log 1 + k_A k_B \log k_C\right)$$

Chegamos a seguinte complexidade de tempo final:

$$\Theta\left(k_A k_B \log k_C\right)$$

2. **Complexidade de espaço:**

$$U(k_C) + U(k_A) + U(k_B)$$

, O custo de espaço é o custo da nova matriz criada, mais o custo de manter o mapa auxiliar de colunas e linhas.

5 Analise

Agora irei descrever a analise experimental, na qual eu fiz de duas formas.

1. Grafica: Fiz experimentos com matrizes quadradas de lado no intervalo $[10^1, 10^3]$, no qual é feito samples igualmente espaçados. E assim gerei graficos a partir disso.

2. Tabular: Fiz experimentos com matrizes quadradas de lado no intervalo $[10^1, 10^6]$, para cada potencia de 10, é estimado o tempo medio

Estou me referindo por TableMatrix, como a implementação que faz operações com uma representação de linhas e vetores na memoria.

6 Comparação

Para todas as operações que eram para ter uma estimativa de tempo constante, apesar de ser estimativamente constante, há um crescimento mesmo em TableMatrix que faz acesso direto a memoria. Suponho que o problema disso, seja alocamento e desalocamento de memoria nos brenchmarks que deve estar acontecendo. Por causa que as matrizes são criadas e logo após não usadas nunca mais, então logo após de ter sido aplicado a operação o programa já libera a memoria.

Para TableMatrix, só foi possivel fazer samples até 10^3, por causa que além disso, a há estouro de memoria.

6.1 Get

Na leitura de valores, é perceptível que para hashmap é perceptivel que é o que tem a melhor performance que todos os outros, tendo tempo aproximadamente constante em nanosegundos para todos valores.

Provavelmente em TableMatrix há uma perda de desempenho de memoria da memoria com o crescimento da matriz por causa de paginação da memoria.

Tamanho	Ocupação	population	HashMapMatrix	TableMatrix	TreeMatrix
10^1x10^1	1%	1	161.600 ns	314.000 ns	100.000 ns
10^1x10^1	5%	5	152.800 ns	264.850 ns	111.250 ns
10^1x10^1	10%	10	161.550 ns	231.300 ns	146.700 ns
10^1x10^1	20%	20	154.150 ns	222.750 ns	212.850 ns
10^2x10^2	1%	100	207.950 ns	2.588 µs	1.138 µs
10^2x10^2	5%	500	187.700 ns	2.539 µs	2.115 µs
10^2x10^2	10%	1000	142.000 ns	2.740 µs	4.144 µs
10^2x10^2	20%	2000	233.450 ns	2.488 µs	13.849 µs
10^3x10^3	1%	10000	635.300 ns	80.230 µs	65.414 µs
10^3x10^3	5%	50000	858.350 ns	89.192 µs	223.284 µs
10^3x10^3	10%	100000	844.100 ns	91.729 µs	672.044 µs
10^3x10^3	20%	200000	717.250 ns	97.935 µs	1.295 ms
10^4x10^4	0.0001%	100	146.550 ns		545.250 ns
10^4x10^4	0.001%	1000	135.200 ns		4.324 µs
10^4x10^4	0.01%	10000	302.050 ns		44.709 µs
10^5x10^5	1e-05%	1000	214.450 ns		6.978 µs
10^5x10^5	0.0001%	10000	498.400 ns		57.752 µs
10^5x10^5	0.001%	100000	745.450 ns		648.069 µs
10^6x10^6	1e-06%	10000	589.350 ns		43.636 µs
10^6x10^6	1e-05%	100000	1.080 µs		526.994 µs
10^6x10^6	0.0001%	1000000	4.596 ms		5.626 ms

6.2 Set

Possui as mesmas características que Get.

Tamanho	Ocupação	population	HashMapMatrix	TableMatrix	TreeMatrix
10^1x10^1	1%	1	168.000 ns	286.450 ns	663.250 ns
10^1x10^1	5%	5	137.250 ns	238.650 ns	118.700 ns
10^1x10^1	10%	10	150.500 ns	232.150 ns	186.900 ns
10^1x10^1	20%	20	149.100 ns	300.750 ns	383.150 ns
10^2x10^2	1%	100	198.050 ns	3.444 µs	921.800 ns
10^2x10^2	5%	500	145.250 ns	2.602 µs	2.817 µs
10^2x10^2	10%	1000	142.500 ns	3.583 µs	6.031 µs
10^2x10^2	20%	2000	199.050 ns	2.659 µs	11.045 µs
10^3x10^3	1%	10000	756.550 ns	80.424 µs	46.669 µs
10^3x10^3	5%	50000	777.250 ns	102.121 µs	231.141 µs
10^3x10^3	10%	100000	846.000 ns	94.056 µs	525.222 µs
10^3x10^3	20%	200000	753.150 ns	102.774 µs	1.107 ms
10^4x10^4	0.0001%	100	147.700 ns		1.195 µs
10^4x10^4	0.001%	1000	207.500 ns		8.130 µs
10^4x10^4	0.01%	10000	251.600 ns		45.633 µs
10^5x10^5	1e-05%	1000	211.350 ns		5.151 µs
10^5x10^5	0.0001%	10000	380.750 ns		48.725 µs
10^5x10^5	0.001%	100000	775.800 ns		538.387 µs
10^6x10^6	1e-06%	10000	715.550 ns		50.742 µs
10^6x10^6	1e-05%	100000	992.050 ns		541.520 µs
10^6x10^6	0.0001%	1000000	4.347 ms		5.530 ms

6.3 Transpor

Transposição é possível considerar que é afetado pela liberação de memoria por causa que,

1. Em HashSet possui um tempo aproximadamente constante, por causa que o HashSet possui apenas um bloco gigante de memoria, então é preciso liberar apenas bloco.
2. Em TableMatrix é afetado porque são $10^3 + 1$ blocos de memorias de tamanho 10^3 para serem liberados.
3. Em TreeMatrix são diversos blocos de memorias da BTree que precisam ser liberados.

Essa me parece ser a melhor explicação para isso estar acontecendo, tentei implementar os testes sem ser medido a liberação de memoria mas não consegui.

Tamanho	Ocupação	population	HashMapMatrix	TableMatrix	TreeMatrix
10^1x10^1	1%	1	103.250 ns	1.217 µs	70.100 ns
10^1x10^1	5%	5	88.450 ns	1.106 µs	79.050 ns
10^1x10^1	10%	10	80.550 ns	1.073 µs	116.200 ns
10^1x10^1	20%	20	78.700 ns	1.121 µs	178.250 ns
10^2x10^2	1%	100	125.200 ns	37.662 µs	620.900 ns
10^2x10^2	5%	500	148.900 ns	31.996 µs	2.173 µs
10^2x10^2	10%	1000	179.250 ns	31.546 µs	4.720 µs
10^2x10^2	20%	2000	227.700 ns	30.339 µs	10.673 µs
10^3x10^3	1%	10000	521.250 ns	9.856 ms	90.702 µs
10^3x10^3	5%	50000	659.200 ns	10.350 ms	227.072 µs
10^3x10^3	10%	100000	777.150 ns	10.190 ms	566.442 µs
10^3x10^3	20%	200000	514.800 ns	10.782 ms	1.168 ms
10^4x10^4	0.0001%	100	83.500 ns		551.750 ns
10^4x10^4	0.001%	1000	89.250 ns		4.212 µs
10^4x10^4	0.01%	10000	172.600 ns		52.106 µs
10^5x10^5	1e-05%	1000	106.550 ns		4.241 µs
10^5x10^5	0.0001%	10000	176.900 ns		45.456 µs
10^5x10^5	0.001%	100000	538.000 ns		602.392 µs
10^6x10^6	1e-06%	10000	462.400 ns		47.406 µs
10^6x10^6	1e-05%	100000	538.650 ns		560.323 µs
10^6x10^6	0.0001%	1000000	3.309 ms		6.149 ms

6.4 Multiplicação Escalar

Nesse caso, claramente é perceptivel, que para TableMatrix, depende puramente do tamanho da matriz e não de quantos elementos possuem na matriz. Onde no caso $10^3 \times 10^3$ com ocupação menor que 10% é TableMatrix pior que os outros casos, já com 20% é melhor que TreeMatrix.

Tamanho	Ocupação	population	HashMapMatrix	TableMatrix	TreeMatrix
10^1x10^1	1%	1	173.500 ns	1.273 µs	193.350 ns
10^1x10^1	5%	5	167.800 ns	1.111 µs	202.650 ns
10^1x10^1	10%	10	277.800 ns	1.125 µs	257.400 ns
10^1x10^1	20%	20	324.500 ns	1.140 µs	601.300 ns
10^2x10^2	1%	100	875.550 ns	42.736 µs	2.203 µs
10^2x10^2	5%	500	3.936 µs	42.345 µs	11.389 µs
10^2x10^2	10%	1000	7.617 µs	42.840 µs	21.201 µs
10^2x10^2	20%	2000	15.423 µs	39.586 µs	45.983 µs
10^3x10^3	1%	10000	76.033 µs	3.749 ms	213.259 µs
10^3x10^3	5%	50000	469.930 µs	3.019 ms	1.216 ms
10^3x10^3	10%	100000	961.508 µs	3.020 ms	2.533 ms
10^3x10^3	20%	200000	2.280 ms	3.017 ms	5.306 ms
10^4x10^4	0.0001%	100	1.552 µs		2.418 µs
10^4x10^4	0.001%	1000	11.062 µs		24.042 µs
10^4x10^4	0.01%	10000	105.187 µs		217.544 µs
10^5x10^5	1e-05%	1000	7.622 µs		22.811 µs
10^5x10^5	0.0001%	10000	80.119 µs		210.217 µs
10^5x10^5	0.001%	100000	994.864 µs		2.631 ms
10^6x10^6	1e-06%	10000	104.900 µs		210.487 µs
10^6x10^6	1e-05%	100000	1.076 ms		2.838 ms
10^6x10^6	0.0001%	1000000	17.700 ms		26.585 ms

6.5 Adição

Para adição, o TableMatrix se demonstra muito mais eficiente do que as duas outras estruturas, sendo pior apenas nos casos 1%. Apesar da quantidade de elementos TableMatrix ser maior que as outras estruturas e elementos que precisam ser lidos também. Processadores são extremamente otimizados para leitura e escrita sequencial de elementos, que faz com que TableMatrix se torne mais eficiente.

Tamanho	Ocupação	population	HashMapMatrix	TableMatrix	TreeMatrix
10^1x10^1	1%	1	644.550 ns	886.950 ns	445.750 ns
10^1x10^1	5%	5	848.200 ns	755.650 ns	709.400 ns
10^1x10^1	10%	10	1.489 µs	766.650 ns	1.699 µs
10^1x10^1	20%	20	3.319 µs	770.750 ns	3.854 µs
10^2x10^2	1%	100	11.958 µs	39.450 µs	23.220 µs
10^2x10^2	5%	500	80.782 µs	40.411 µs	141.043 µs
10^2x10^2	10%	1000	141.338 µs	42.648 µs	286.892 µs
10^2x10^2	20%	2000	253.380 µs	35.940 µs	511.524 µs
10^3x10^3	1%	10000	1.328 ms	4.213 ms	2.333 ms
10^3x10^3	5%	50000	9.592 ms	3.937 ms	12.612 ms
10^3x10^3	10%	100000	18.137 ms	3.811 ms	26.052 ms
10^3x10^3	20%	200000	44.696 ms	3.782 ms	55.186 ms
10^4x10^4	0.0001%	100	12.680 µs		19.994 µs
10^4x10^4	0.001%	1000	140.318 µs		205.606 µs
10^4x10^4	0.01%	10000	1.348 ms		2.252 ms
10^5x10^5	1e-05%	1000	147.205 µs		208.942 µs
10^5x10^5	0.0001%	10000	1.307 ms		2.536 ms
10^5x10^5	0.001%	100000	16.764 ms		26.623 ms
10^6x10^6	1e-06%	10000	2.129 ms		2.311 ms
10^6x10^6	1e-05%	100000	26.515 ms		30.957 ms
10^6x10^6	0.0001%	1000000	392.212 ms		296.280 ms

6.6 Multiplicação

Mutiplicação aqui aparece como a operação que se tem o maior proveito da esparcidade da matriz, como no caso $10^3 \times 10^3$ e 1%, onde HashMapMatrix é mais de 100x mais performático que TableMatrix.

Nese caso também HashMapMatrix e TreeMatrix tem valores muito proximos também, apesar de que HashMapMatrix é constantemente mais performatico.

Tamanho	Ocupação	population	HashMapMatrix	TableMatrix	TreeMatrix
10^1x10^1	1%	1	1.774 µs	1.688 µs	930.250 ns
10^1x10^1	5%	5	2.377 µs	1.760 µs	2.231 µs
10^1x10^1	10%	10	5.196 µs	1.894 µs	4.310 µs
10^1x10^1	20%	20	9.396 µs	1.807 µs	11.884 µs
10^2x10^2	1%	100	43.326 µs	1.481 ms	54.660 µs
10^2x10^2	5%	500	400.137 µs	1.725 ms	1.662 ms
10^2x10^2	10%	1000	1.187 ms	1.594 ms	2.486 ms
10^2x10^2	20%	2000	4.040 ms	1.632 ms	8.941 ms
10^3x10^3	1%	10000	13.442 ms	1.946 s	31.565 ms
10^3x10^3	5%	50000	509.380 ms	1.896 s	882.779 ms
10^3x10^3	10%	100000	2.175 s	1.966 s	3.271 s
10^3x10^3	20%	200000	8.239 s	2.184 s	9.949 s
10^4x10^4	0.0001%	100	35.514 µs		53.087 µs
10^4x10^4	0.001%	1000	420.390 µs		555.832 µs
10^4x10^4	0.01%	10000	4.302 ms		7.091 ms
10^5x10^5	1e-05%	1000	355.582 µs		730.562 µs
10^5x10^5	0.0001%	10000	4.135 ms		5.417 ms
10^5x10^5	0.001%	100000	76.117 ms		91.635 ms
10^6x10^6	1e-06%	10000	4.191 ms		5.961 ms
10^6x10^6	1e-05%	100000	85.859 ms		88.136 ms
10^6x10^6	0.0001%	1000000	1.221 s		1.515 s

7 Analise Assintótica

Foi gerado graficos, onde o eixo X é a quantidade de elementos que cada matriz possui, e o eixo Y é a razão da duração do sample por alguma função assintótica.

Isso é usado por causa de que:

$$\lim_{n\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R} \to f(x) \in \Theta(x)$$

$$\lim_{n\to\infty} \frac{f(x)}{g(x)} = \infty \to f(x) \in \Omega(g(x))$$

$$\lim_{n\to\infty} \frac{f(x)}{g(x)} = 0 \to f(x) \in O(g(x))$$

Portanto, dado isso, podemos definir então por cima, de que caso a razão dos samples estiver tendendo 0 então é O, caso esteja crescente então é Ω, caso esteja nem decrescendo indefinidamente nem crescendo indefinidamente então é Θ

Nos graficos há tres partes:

1. Curva do Supremo: Curva pontos (k, d) que possuem a propriedade $\forall(k', d'), k > k' \to d > d'$, portanto o ponto que é supremo dos valores a partir dele para frente.
2. Curva do Infimo: Curva dos pontos (k, d) que possuem a propriedade $\forall(k', d'), k > k' \to d < d'$, portanto o ponto que é o infimo dos valores a partir dele para frente.
3. Pontos Samples coloridos: Os pontos que foram obtidos do experimento, onde são coloridos com base na ocupação.
4. Linha media ponderada: Linha horizontal da media ponderada da duração, onde o peso de cada ponto é a população.

7.1 TreeMapMatrix

7.1.1 set

Para modificar um valor, tem tempo, que a razão tende a crescer um pouco a mais que $\log k$, porém, extremamente inferior a linear, demonstrando que tem uma aproximação ao estimado que é $\Theta(\log k)$

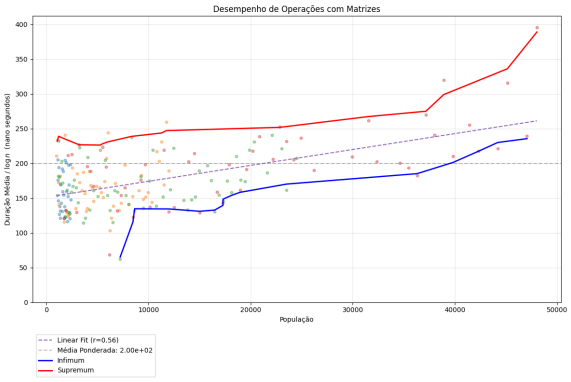

Figura 1: $\log k$

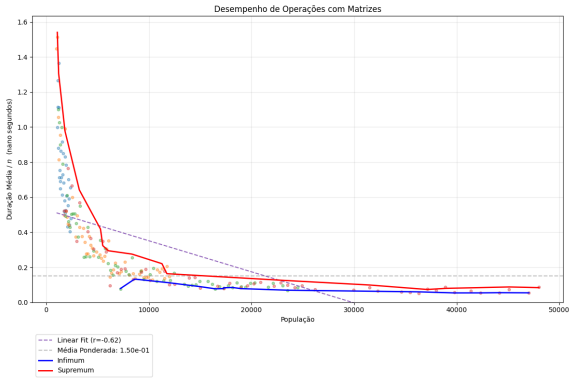

Figura 2: k

7.1.2 get

Para acesso, as mesmas caracteristicas que set.

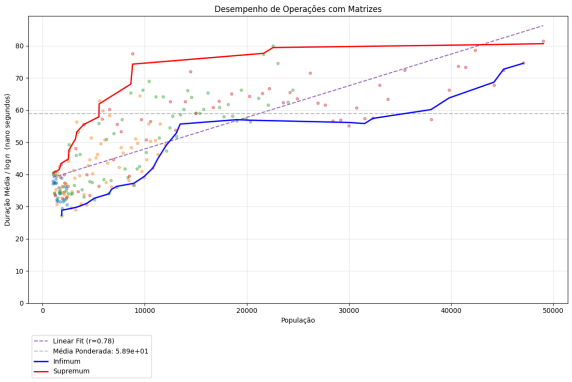

Figura 3: $\log k$

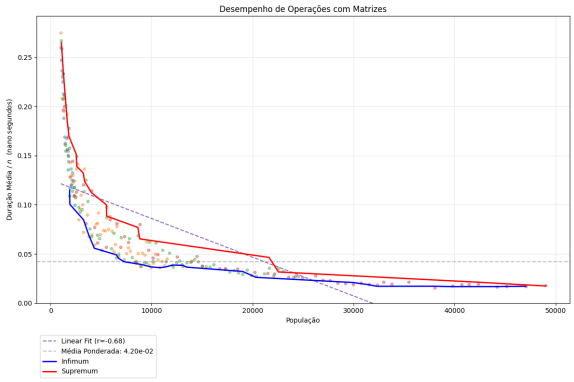

Figura 4: k

7.1.3 Transposição

O comportamento da transposição tende a parecer linear ao crescer, mas demonstra uma taxa de crescimento.

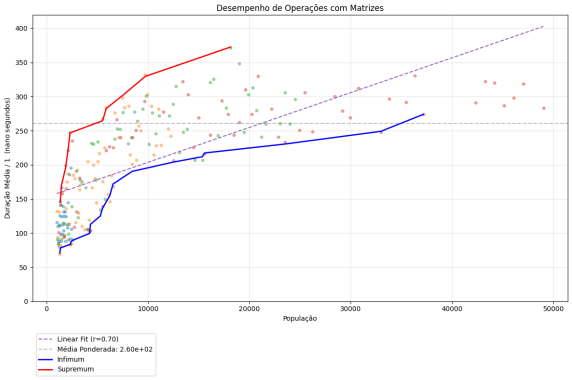

Figura 5: 1

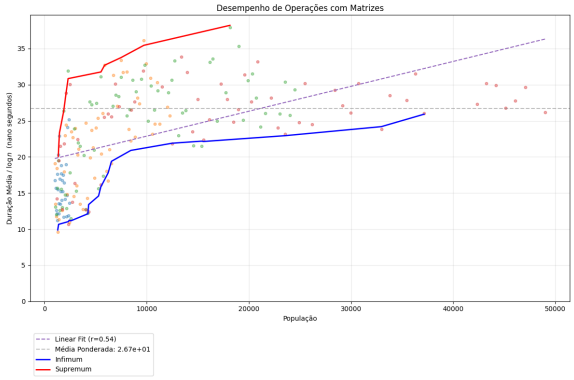

Figura 6: k

7.1.4 Multiplicação Escalar

O grafico se demonstra crescente em relação a $\log k$ e decrescente em relação a k, portanto é uma complexidade pior que logaritmica, mas melhor que linear

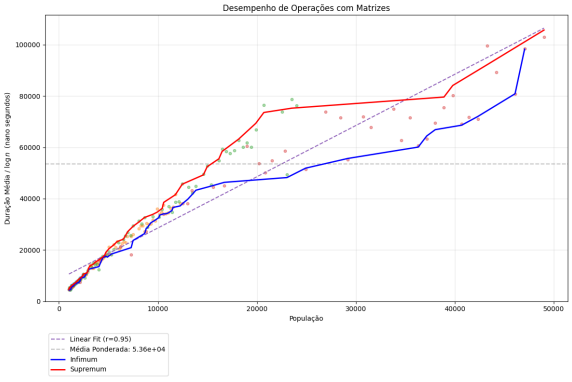

Figura 7: $\log k$

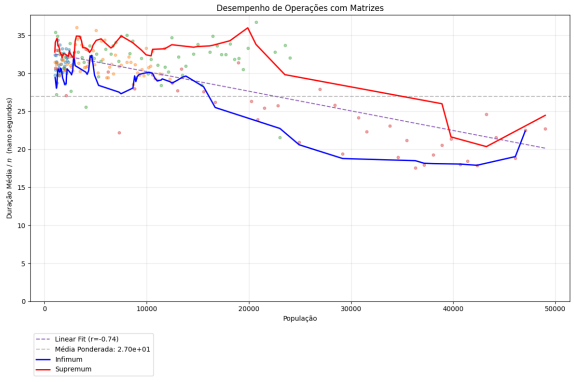

Figura 8: k

7.1.5 Adição

O comportamento para soma, tem uma tendencia extremamente alta para convergir na razão pelo linear, mostrando um comportamento claramente na parte testada $\Theta(k)$, melhor do que o calculado.

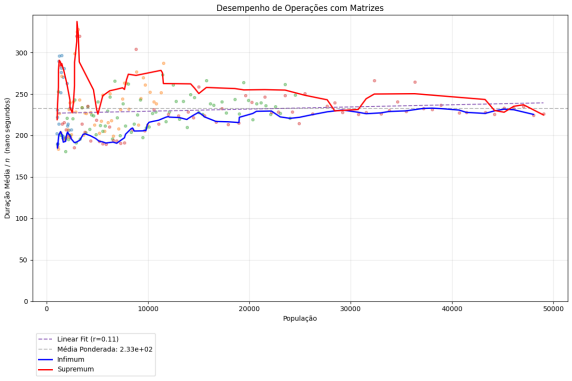

Figura 9: k

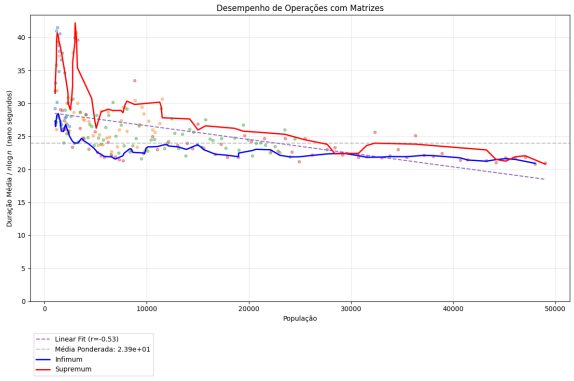

Figura 10: $k \log k$

7.1.6 Multiplicação Matrizes

Para multiplicação de matrizes, o algoritmo demonstra ter um comportamento próximo de $k \log k \sqrt{k}$ porém com uma tendencia de crescimento, porém com uma tendencia decrescente em relação a k^2.

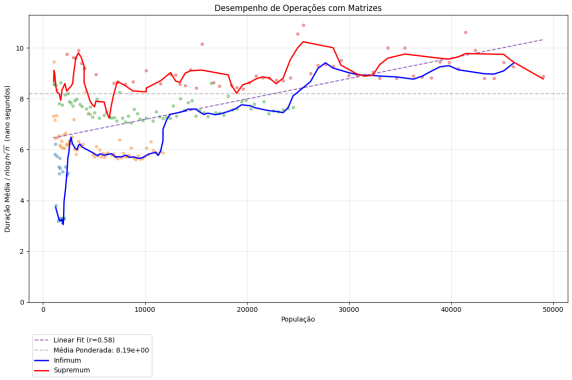

Figura 11: $k \log k \sqrt{k}$

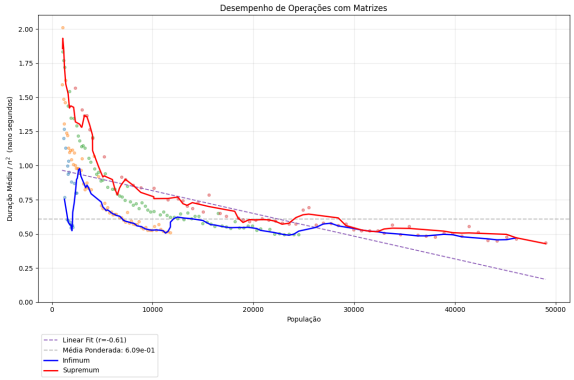

Figura 12: k^2

7.2 HashMapMatrix

7.2.1 set

Para o set, apesar do esperado ter sido estimado como constante, as métricas tenderam para estar se aproximando em relação a $\log k$, e ser muito menor que linear.

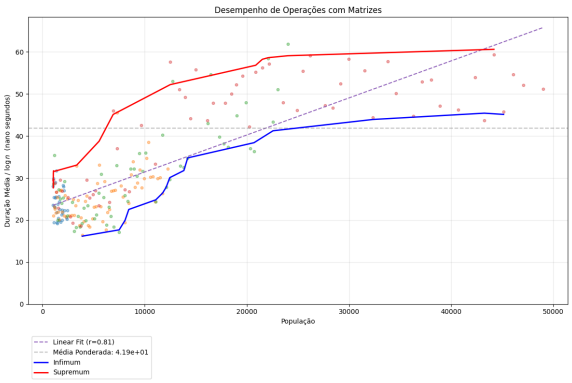

Figura 13: $\log k$

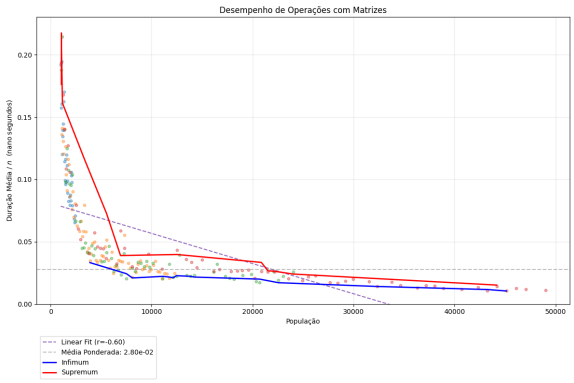

Figura 14: k

7.2.2 get

Para acesso, as mesmo comportamento que set.

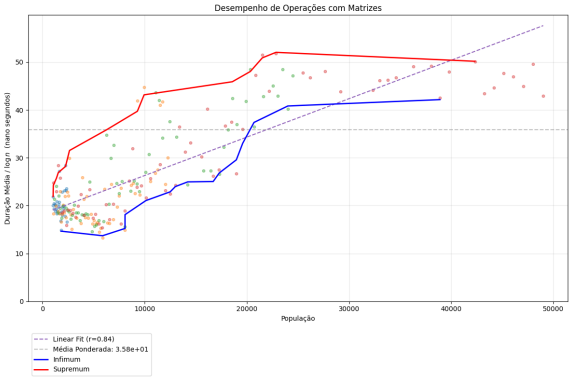

Figura 15: $\log k$

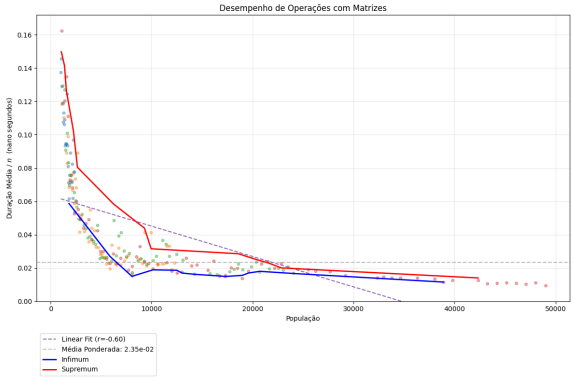

Figura 16: k

7.2.3 Transposição

O comportamento da transposição está próximo do comportamento do get e do set, mostrando que provavelmente deve ser um problema em relação ao experimento, o fato de estar mais próximo de log do que constante.

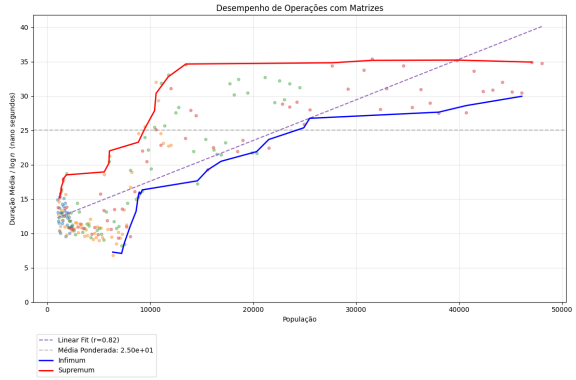

Figura 17: 1

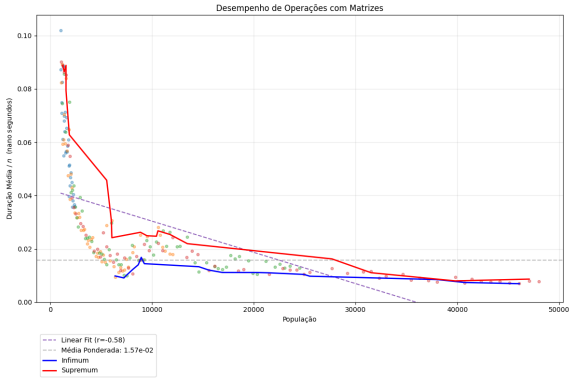

Figura 18: k

7.2.4 Multiplicação Escalar

Na multiplicação de escalar é possivel perceber que tem uma performance suavemente melhor que $k \log k$ e suavemente pior que k. Mas além disso um detalhe importante é perceber o comportamente ocilatorio do valor, que se dá ao quanto da tabela é ocupada, onde quando temos uma tabela com a capacidade proxima do tamanho, temos uma melhor performance, e como a ocupação da tabela vai variando, então a performance varia, mas se mantêm aproximadamente a razão.

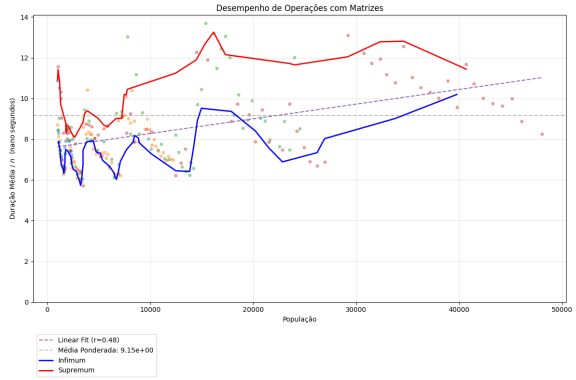

Figura 19: k

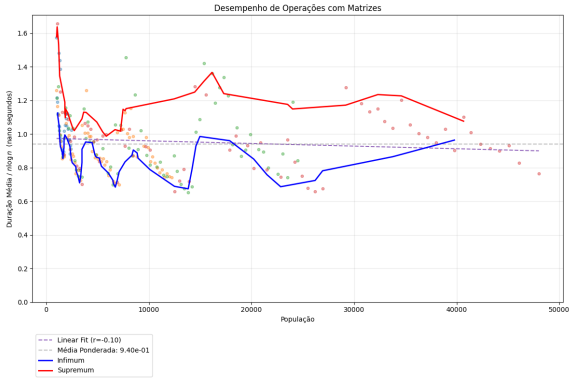

Figura 20: $k \log k$

7.2.5 Adição

Tem um comportamento um pouco pior que linear, e um pouco melhor que log-linear, que provavelmente é causado pelas colisões de hash.

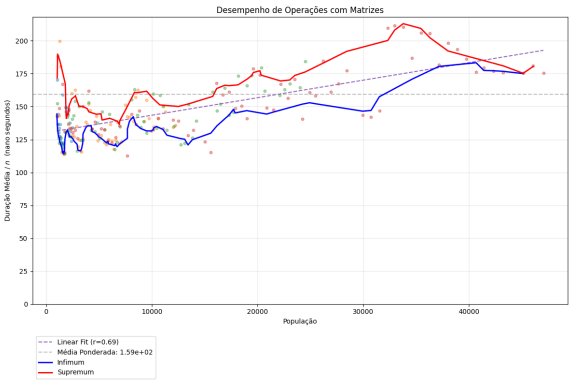

Figura 21: k

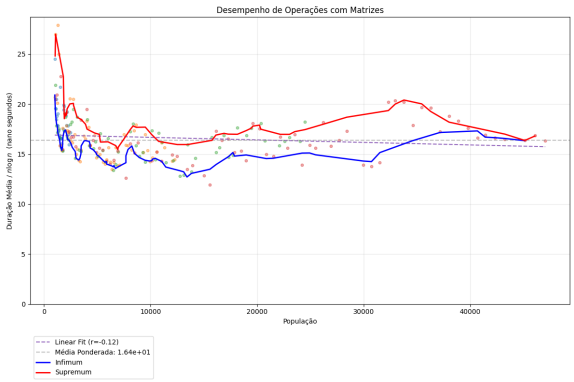

Figura 22: $k \log k$

7.2.6 Multiplicação Matrizes

Para multiplicação de matrizes, o algoritmo demonstra ter um comportamento próximo de $k \log k\sqrt{k}$ porém com uma tendencia de crescimento, porém com uma tendencia decrescente em relação a k^2, então como estimado é melhor que k^2.

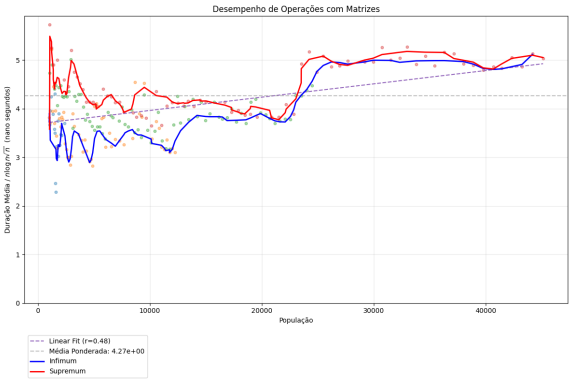

Figura 23: $k \log k\sqrt{k}$

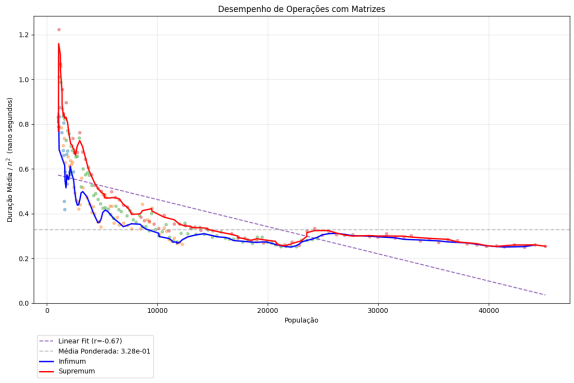

Figura 24: k^2

7.3 TableMatrix

Para TableMatrix, a analise assintótica em relação ao quantidade de elementos não nulo, não faz sentido, porque não existe um limite superior de quão ruim um caso pode ser, onde o caso sempre cresce indefinidamente junto ao tamanho da matriz e não em relação a quantidade de elementos não nulos.

Isso é perceptivel nos graficos de que há uma clara separação na maior parte deles do tempo em relação a cada taxa de ocupação (indicada pelas cores dos pontos).

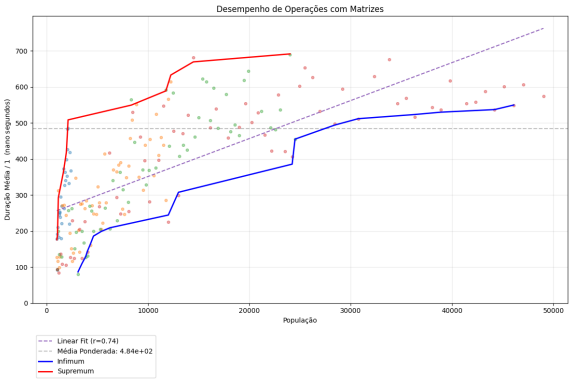

Figura 25: Set

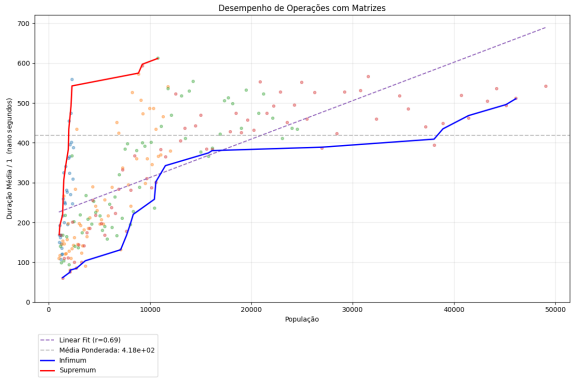

Figura 26: Get

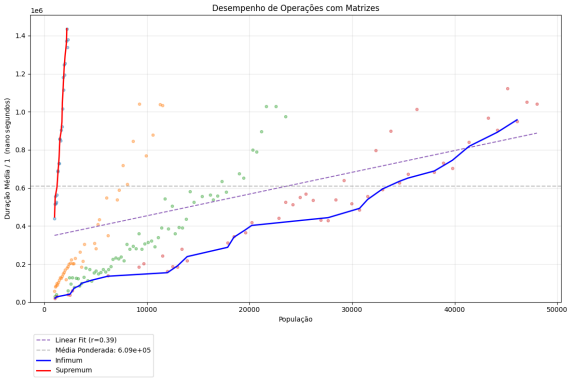

Figura 27: Transpor

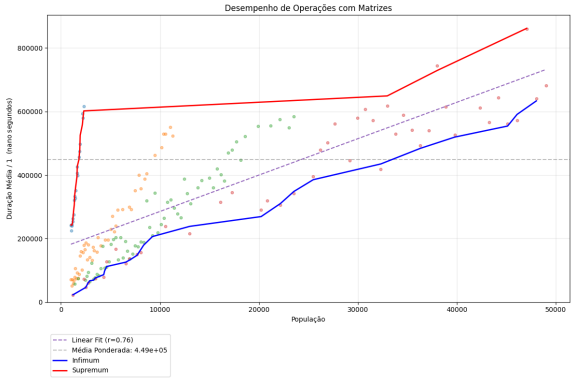

Figura 28: Multiplicação escalar

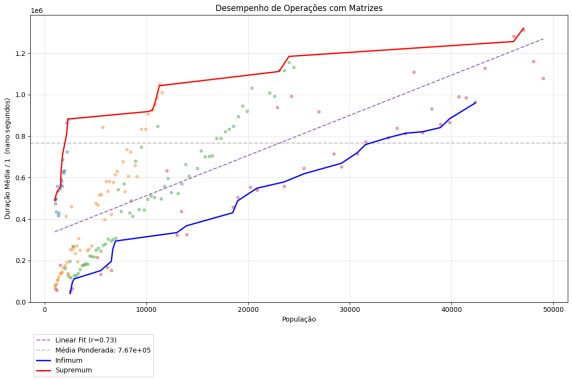

Figura 29: Adição

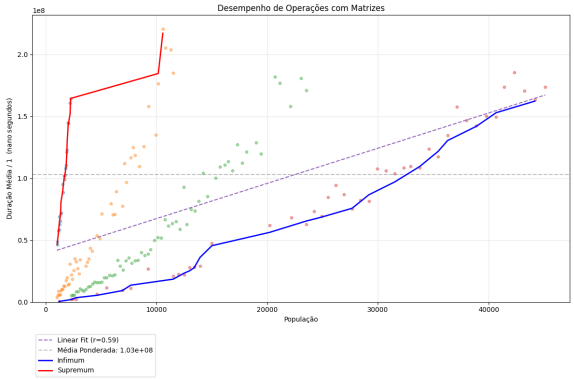

Figura 30: Multiplicação Matriz

Pegando a multiplicação de matriz em destaque é perceptivel, que a linha do supremo segue exatamente uma linha de cores iguais, que é exatamente os pontos do 1% de ocupação, onde é 1% de matrizes muito maiores que a população, então se diverge das outras.

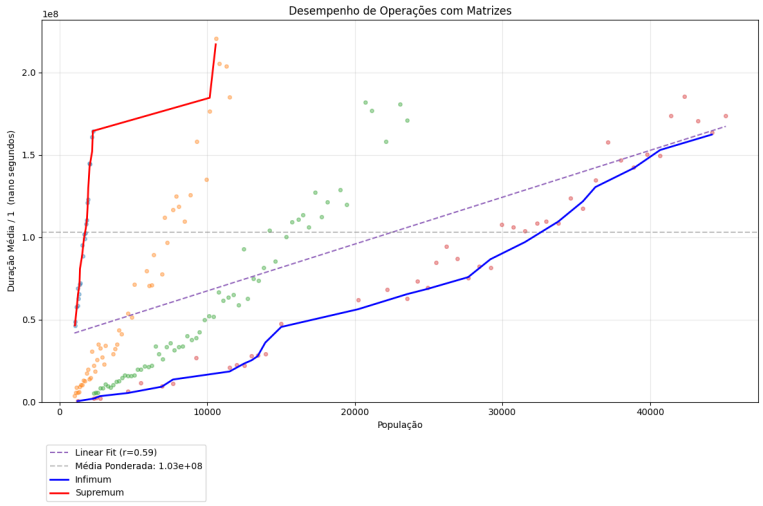

Figura 31: Multiplicação Matriz

Mas caso, plotamos os valores, mas ao invés de colocarmos no eixo x a quantidade de elementos não nulos, colocarmos a quantidade de elementos no total, é perceptível que os dados são muito mais correlacionados.

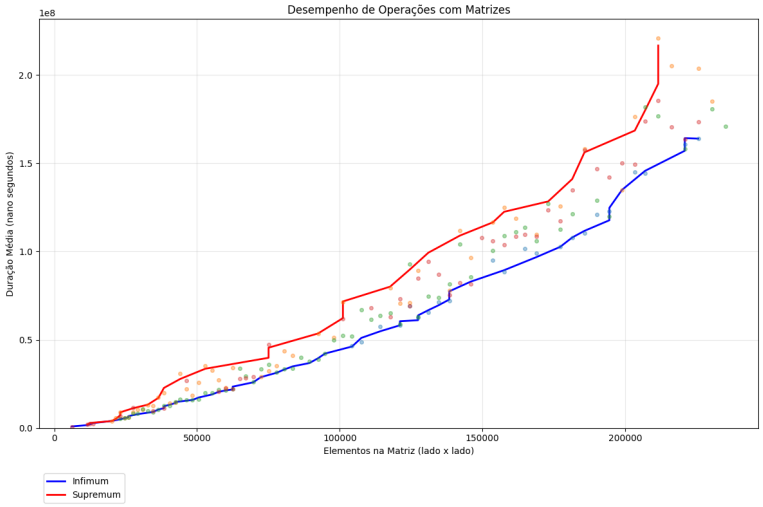

Figura 32: Multiplicação Matriz

8 Conclusão

Dado toda analise teórica e medições feitas é possível tirar certas conclusões:

Como a memoria é usada, liberada e feita é extremamente importante, visto que mesmo para a TableMatriz, acesso aleatório piorou com o crescimento da matriz, por causa de fatores como leitura da memoria, caching de memoria e paginação da memoria pelo sistema operacional (Uma matriz $10^2 \times 10^2$ já é maior que 4kb que é maior que uma pagina de sistema).

Para implementação que não são esparsas é extremamente mais eficiente o uso de matrizes diretas, porque tem um aproveitamento muito melhor da arquitetura do computador.

HashTableMatrix tem uma eficiência maior no geral do que TreeMapMatrix, porém, ao custo de que pode ter custos imprevisíveis certas operações. Sendo um deles o custo de aumentar o tamanho da HashTable, que apesar de ser amortizado pode acabar levando a interrupções temporárias em sistemas, como por exemplo, caso haja uma matriz esparsa que ocupe 1GB de memoria, e seja necessário aumentar o tamanho dela, isso levaria a ter que começar a fazer uma copia de 1GB de memoria. Problema esse que não existe no TreeMapMatrix, por causa que o crescimento e decrescimento é granular, sem a necessidade de liberação e alocação de quantidades gigantes de memoria.

8.1 Nota: Abstração

Um problema que aconteceu durante o desenvolvimento, foi de que foi tentando fazer multiplicação de matrizes de tamanho $10^5 \times 10^5$ com 1% de ocupação porém não foi possível e não foi por causa do tempo de processamento, mas sim por causa por causa de que a matriz resultante da multiplicação tinha uma taxa de ocupação muito maior que fazia com que a matriz resultante e os artefatos intermediários não cabiam na memoria.

Para isso considerei usar resolver esse problema, aproveitando exatamente de como funciona o MapMatrix, sem reescrever a MapMatrix, e implementar uma BTree persistida no disco, no qual não iria afetar de forma tão grande o desempenho por ser no disco, por causa que a implementação de MapMatrix de multiplicação e soma (mais custosos), usa iteradores que faz com que para iterar na matriz, bastasse ler cada bloco de disco para memoria, iterar sobre eles, fazendo as operações sequencialmente. Fazendo assim leitura do disco em grandes blocos e de forma sequencial algo extremamente mais eficiente do que acesso aleatório do disco.

Isso sem precisar modificar o funcionamento do MapMatrix, já que é feito sobre um dos pilares da programação a **abstração**.