MC458 - Projeto

Juca Magalhães Meniconi
RA: 281803

Lucas Beserra Fernandes
RA: 281815

Vinícius Augusto Silva Brasileiro
RA: 247391

Nesse projeto, exploraremos duas estruturas de dados que representam matrizes esparsas. Iremos analisar operações e características de cada estrutura, a fim de verificar a eficiência dessas estruturas. Com isso, discutiremos as vantagens e desvantagens das implementações quando comparadas entre si e quando comparadas com representações de matrizes não esparsas. O código está disponível em `https://github.com/viniasbr/MC458-projeto`

1 Descrição das Estruturas

1.1 Estrutura 1 - Hash Table

A Estrutura 1 é uma representação de matrizes esparsas baseada em Hash. Armazenaremos todos os elementos não nulos da matriz em um vetor. O local em que cada elemento é armazenado é baseado em um Hash no número da linha e da coluna. Para lidar com possíveis colisões, cada posição do vetor é uma lista ligada. O tamanho do vetor é atualizado dinamicamente para manter o Load Factor entre 0.25 e 0.75, o que é feito junto a uma chamada de `resize`.

1.2 Estrutura 2 - AVL

A Estrutura 2 é uma representação de matrizes esparsas baseada em árvores, mais especificamente em árvores AVL. Teremos duas estruturas que fazem parte da matriz: uma árvore interna e uma externa. Cada árvore obedecerá à propriedade de árvore de busca de acordo com o número da linha ou número da coluna. A árvore externa terá como dado o endereço de uma matriz interna, que representa todos os elementos não nulos da matriz que estão na linha/coluna indicada pela árvore externa. A árvore interna terá o número da coluna/linha de cada elemento, assim como o valor do elemento da matriz, que sabemos que está na posição indicada pelos índices das matrizes externa e interna.

2 Análise das Complexidades

2.1 Estrutura 1 - Hash Table

Para a Estrutura 1, analisaremos a complexidade esperada. O custo de alocação de memória é tido como constante.

2.1.1 Complexidade de `hash`

A operação de Hash inclui apenas um número constante de operações básicas. Assim, essa função $T(k) \in \Theta(1)$. Porém, analisaremos também a esperança de elementos por bucket, que será usada para a análise da complexidade do caso médio de outras operações. Como nesse hash a tabela de espalhamento é um vetor, eventualmente será chamada de vetor de espalhamento ou simplesmente de vetor.

Vamos presumir que cada par linha coluna p que fazemos o hash tem igual probabilidade de cair em cada bucket em um vetor de tamanho m. A probabilidade de p cair no i-ésimo bucket é $\frac{1}{m}$. Se temos n elementos nesse vetor, então a esperança do número de elementos em qualquer bucket é $\frac{n}{m}$. Note que n e m tem uma relação limitada pelo load factor α do hash. Como o aumento ou diminuição do load factor só pode ser feito pela função `set_element_hash`, e essa função ajusta o tamanho do vetor de espalhamento sempre que o número de elementos sai do intervalo aceitável [0.25, 0.75], então

$$0.25 < \frac{n}{m} < 0.75$$

Portanto, a esperança do número de elementos por bucket $T(k) \in O(1)$.

2.1.2 Complexidade de `createHashMatrix`

Nessa operação, alocamos memória de acordo com o Struct `HashMatrix`, o que tem custo assumido como constante. Então, atribuímos valores iniciais para os 5 campos da Struct, cada atribuição em tempo constante. Então retornamos a matriz. $T(k) \in O(1)$.

2.1.3 Complexidade de `resize`

Após algumas operações de custo constante, percorremos o vetor de espalhamento, e para cada elemento do vetor, percorremos o bucket associado a ele. Como sabemos que, em média, teremos um número constante de elementos por bucket, a classe assintótica é igual ao de só percorrer o vetor. Para cada elemento do vetor, faremos uma quantidade constante de operações. Além disso, sabemos, pelo número constante de elementos por bucket e pelo limite de Load Factor entre constantes, que o número de elementos no vetor de espalhamento é igual ao número de elementos na matriz multiplicado por uma constante. Portanto, o custo esperado de `resize` é $T(k) \in O(k)$.

2.1.4 Complexidade de `get_element_hash`

Nessa função, fazemos um número constante de operações, então executamos a função `hash`, de custo constante. Então, percorremos o bucket da posição indicada pelo `hash`. Como já foi discutido, o número esperado de elementos por bucket é constante, portanto, o custo esperado de toda operação é constante, e $T(k) \in O(1)$.

2.1.5 Complexidade de `set_element_hash`

O custo esperado da operação `set_element_hash` é parecido ao de `get_element_hash`. Fazemos um número constante de operações, com exceção da iteração do bucket, que tem número esperado de elementos constante. Portanto, $T(k) \in \Theta(1)$. Essa operação também serve para remover elementos caso seja atualizado um elemento não nulo com 0, tornando-o nulo. Nesse caso, o custo assintótico não muda. A diferença é que a função

pode chamar `resize` eventualmente, o que faria com que o custo fosse $O(k)$. Porém, vamos verificar o custo amortizado.

Entendemos como *Load Factor* α a razão entre o número n de elementos no vetor e o número m do tamanho do vetor. Ou seja:

$$\alpha = \frac{n}{m}$$

Se o hash está entre o α_min α_max, não fazemos nada. Porém, se o Load Factor ficaria além do intervalo, alteramos o tamanho da tabela com `resize`. Se o α está perto do limite superior, dobramos o tamanho. Se α está perto do limite inferior, então cortamos o tamanho pela metade. Ou seja, estamos sempre dentro dos limites estipulados para o Load Factor. Sabemos que o custo de `resize`, como demostrado abaixo, $T(k) \in O(k)$. Porém, em uma sequência sucessiva de inserções, podemos calcular o custo médio por inserção porque sabemos que não é toda inserção que irá levar a um chamado de `resize`.

Considere uma sequência de n inserções. Cada `resize` custa O(i), sendo i o tamanho atual. O número de inserções que podemos fazer entre cada `resize` é $\Theta(i)$ porque dobramos o tamanho a cada vez. Portanto, se fazemos uma operação O(i) a cada $\Theta(i)$ inserções, então o custo amortizado da inserção é $O(1+\frac{O(i)}{\Theta(i)})$, portanto o custo amortizado da inserção $\in O(1)$.

2.1.6 Complexidade de `matrix_multiplication_hash`

Nessa operação, criamos uma matriz resultado (custo constante). Então percorremos o vetor de espalhamento para a matriz A, e então percorremos os buckets do vetor de A. Para cada elemento no bucket, percorremos o vetor de espalhamento de B, e para cada elemento no vetor de espalhamento de B, percorremos o bucket associado. Apesar de ser um laço quadruplamente aninhado, temos limitantes garantidos para o tamanho do vetor com relação a K, e temos limitantes esperados do número de elementos por bucket. Além de sabermos que, em todos os buckets de uma matriz M, temos exatamente Km elementos. Como já discutido antes, o custo esperado de percorrer o vetor de espalhamento, mesmo se também percorremos os buckets de cada elemento, é linear em K. Como percorremos A, e B de maneira aninhada, o custo esperado é $T(k) \in O(Ka \cdot Kb)$, sendo Ka o número de elementos na matriz esparsa A e Kb o número de elementos na matriz esparsa B.

2.1.7 Complexidade de `matrix_addition_hash`

Nessa operação, criamos uma matriz resultado (custo constante), então percorremos o vetor de espalhamento para as duas matrizes que serão somadas, A e B. Para cada elemento do vetor, percorremos o bucket associado. O custo esperado de percorrer o vetor de espalhamento, mesmo se também percorremos os buckets de cada elemento, é linear em K. Como percorremos A, e depois B, o custo é $T(k) \in O(Ka + Kb)$, sendo Ka o número de elementos na matriz esparsa A e Kb o número de elementos na matriz esparsa B.

2.1.8 Complexidade de `matrix_scalar_multiplication_hash`

Nessa operação, criamos uma matriz resultado (custo constante), então percorremos o vetor de espalhamento, e para cada elemento do vetor, percorremos o bucket associado.

Como já discutido antes, o custo esperado de percorrer o vetor de espalhamento, mesmo se também percorremos os buckets de cada elemento, $T(k) \in O(k)$.

2.1.9 Complexidade de `transpose_hash`

Essa operação tem somente uma linha, que é uma mudança de um valor booleano para seu oposto. Isso tem custo constante. A forma como a transposição é de fato feita é pela mudança entre linhas e colunas nas outras operações. Portanto $T(k) \in \Theta(1)$.

2.2 Estrutura 2 - AVL

Função	Complexidade de Tempo
`avl_status_string`	$O(1)$
`create_matrix_avl`	$O(1)$
`transpose_avl`	$O(1)$
`_allocation_fail`	$O(1)$
`_validate_matrix`	$O(1)$
`_validate_indices`	$O(1)$
`_validate_same_dimensions`	$O(1)$
`_height_i`	$O(1)$
`_height_o`	$O(1)$
`_balance_factor_i`	$O(1)$
`_balance_factor_o`	$O(1)$
`_max`	$O(1)$
`_left_rotate_i`	$O(1)$
`_right_rotate_i`	$O(1)$
`_left_rotate_o`	$O(1)$
`_right_rotate_o`	$O(1)$

Tabela 1: Funções com complexidade de tempo trivial da estrutura AVL.

Lema 1. *Sendo h a altura de uma árvore AVL e n seu número de nós, $h \in O(\log n)$.*

Demonstração. A árvore AVL tem a garantia de que $|h_e - h_d| \leq 1$ para todo nó. Seja $N(h)$ o número mínimo de nós de uma árvore AVL. Temos que, a árvore menos cheia possível que ainda respeita a condição de AVL deve respeitar:

$$N(h) = \begin{cases} 1 & \text{se } h = 1 \\ 2 & \text{se } h = 2 \\ 1 + N(h-1) + N(h-2) & \text{se } h > 2 \end{cases}$$

Vamos provar por indução que $N \in \Omega(2^{\frac{h}{2}})$. Escolhendo $c = \frac{\sqrt{2}}{2}$, caso base:

$$1 \geq c \cdot \sqrt{2} \quad \wedge \quad 2 \geq c \cdot 2$$

Hipótese Indutiva:

$$\forall i (1 \leq i < h) \rightarrow c \cdot 2^{\frac{i}{2}} \leq N(i)$$

Passo Indutivo:

$$
\begin{aligned}
N(h) &= 1 + N(h-1) + N(h-2) \\
&\geq 1 + c \cdot 2^{\frac{h-1}{2}} + c \cdot 2^{\frac{h-2}{2}} \\
&= 1 + c \cdot 2^{\frac{h}{2}} \left(\frac{1}{\sqrt{2}} + \frac{1}{2} \right) \\
&\geq c \cdot 2^{\frac{h}{2}}.
\end{aligned}
$$

Assim, $N(h) \geq c \cdot 2^{\frac{h}{2}}$ e, dado que cada árvore AVL com altura h possui pelo menos $N(h)$ nós, temos $n \geq N(h)$. Logo:

$$
n \in \Omega(2^{\frac{h}{2}}) \implies h \in O(\log n)
$$

□

2.2.1 Complexidade de espaço de `AVLMatrix`

Os campos de tamanho variável de `AVLMatrix` são `main_root` e `transposed_root`. Ambos são árvores de nós "externos", onde cada nó contem árvores de nós "internos", que contem o dado armazenado. O somatório da quantidade de nós internos deve ser k, uma vez que só são armazenados elementos não nulos. No pior caso, cada nó "externo"guarda somente um nó "interno", gerando assim k nós internos e externos para `main_root` e k nós internos e externos para `transposed_root`. Assim, temos um gasto de memória proporcional à constante referente ao tamanho dos nós multiplicada por dois para cada um dos k nós. Sendo assim, a estrutura gasta, trivialmente, $M(k) \in O(k)$ memória para o número de dados não nulos na matriz esparsa k.

2.2.2 Complexidade de `_find_node_i` e `_find_node_o`

A função faz somente um caminho da raiz até a folha, uma vez que, para cada nó, visita somente ou a sub-árvore esquerda ou a sub-árvore direita, sempre. Em cada chamada, é executado esforço computacional constante. Dessa forma, o algoritmo possui complexidade $T(h) \in O(h) \implies T(n) \in O(\log n)$, pelo lema 1, sendo n o número de nós e h a altura.

□

2.2.3 Complexidade de `_insert_i` e `_insert_o`

A função faz somente um caminho da raiz até a folha, uma vez que, para cada nó, visita somente ou a sub-árvore esquerda ou a sub-árvore direita, sempre. Em cada chamada, é executado esforço computacional constante. Dessa forma, o algoritmo possui complexidade $T(h) \in O(h) \implies T(n) \in O(\log n)$, pelo lema 1, sendo n o número de nós e h a altura.

□

2.2.4 Complexidade de `_find_max_i` e `_find_max_o`

A função faz somente um caminho da raiz até a folha, indo sempre pela sub-árvore direita. Assim, esse caminho está limitado pela altura da árvore, e esforço computacional constante

é executado em cada chamada. Segue que o algoritmo possui complexidade $T(h) \in O(h) \implies T(n) \in O(\log n)$, pelo lema 1, sendo n o número de nós e h a altura.

□

2.2.5 Complexidade de `_remove_i` e `_remove_o`

A função faz somente um caminho da raiz até o nó, uma vez que, para cada nó, visita somente ou a sub-árvore esquerda ou a sub-árvore direita, sempre. Em cada chamada não base, é executado esforço computacional constante. Dessa forma, o algoritmo possui complexidade $T(h) = T_r(h)+T_b(h)$. $T_r(h) \in O(h) \implies T_r(n) \in O(\log n)$, e o caso base de remoção de nó com dois filhos faz esforço constante, uma chamada para `_find_max_*`, que faz esforço computacional linear na altura, e uma chamada para a própria função, com a garantia de cair num caso base de esforço computacional constante – uma vez que o nó a ser removido é um nó folha. Dessa forma, ambas chamadas são realizadas em sub-árvores da árvore original, sendo limitadas superiormente pela altura da árvore total. Assim, $T_b(h) \in O(h) \implies T_b(n) \in O(\log n)$, o que leva a $T(h) \in O(h) \implies T(n) \in O(\log n)$.

□

2.2.6 Complexidade de `_free_i_tree` e `_free_o_tree`

Cada nó da árvore é visitado recursivamente, realizando esforço computacional constante em cada chamada no caso "inner". Assim, $T(n) \in O(n)$. No caso "outer", cada nó é visitado recursivamente, e em cada chamada recursiva, cada nó da árvore interna contida no nó externo é visitada pela função "inner". Dessa forma, $T(n) \in O(n)$, com a diferença que n é o número total de nós "inner"em uma árvore "outer", ou seja, o número total de elementos que a estrutura armazena.

□

2.2.7 Complexidade de `_clone_i_tree` e `_clone_o_tree`

Cada nó da árvore é visitado recursivamente, realizando esforço computacional constante em cada chamada no caso "inner". Assim, $T(n) \in O(n)$. No caso "outer", cada nó é visitado recursivamente, e em cada chamada recursiva, cada nó da árvore interna contida no nó externo é visitada pela função "inner". Dessa forma, $T(n) \in O(n)$, com a diferença que n é o número total de nós "inner"em uma árvore "outer", ou seja, o número total de elementos que a estrutura armazena. Essa função cria uma nova árvore que é uma exata cópia da árvore anterior, gastando assim $M(n) \in O(n)$ de memória.

□

2.2.8 Complexidade de `_copy_matrix`

Essa função faz esforço computacional constante e invoca duas funções de esforço computacional linear na quantidade de elementos duas vezes cada. Assim, a função possui complexidade $T(n) \in O(n)$.

2.2.9 Complexidade de `_scalar_multiply_i_tree` e `_scalar_multiply_o_tree`

Cada nó da árvore é visitado recursivamente, realizando esforço computacional constante em cada chamada no caso "inner". Assim, $T(n) \in O(n)$. No caso "outer", cada nó é visitado recursivamente, e em cada chamada recursiva, cada nó da árvore interna contida

no nó externo é visitada pela função "inner". Dessa forma, $T(n) \in O(n)$, com a diferença que n é o número total de nós "inner"em uma árvore "outer", ou seja, o número total de elementos que a estrutura armazena.

□

2.2.10 Complexidade de `_copy_i` e `_copy_o`

Cada nó da árvore é visitado recursivamente, realizando esforço computacional constante em cada chamada no caso "inner". Assim, $T(n) \in O(n)$. No caso "outer", cada nó é visitado recursivamente, e em cada chamada recursiva, cada nó da árvore interna contida no nó externo é visitada pela função "inner". Dessa forma, $T(n) \in O(n)$, com a diferença que n é o número total de nós "inner"em uma árvore "outer", ou seja, o número total de elementos que a estrutura armazena.

□

2.2.11 Complexidade de `_matmul_i_accumulate`

Cada nó da árvore é visitado recursivamente. A árvore interna representa exatamente uma coluna da matriz, e cada nó corresponde a um elemento não nulo. Denominamos l a quantidade de elementos não nulos na dada coluna da matriz, que é a quantidade de elementos na árvore. Para cada chamada, o algoritmo chama dois algoritmos que serão provados como logarítmicos na quantidade de nós da referida árvore, além de realizar esforço computacional constante. Sendo assim, esse algoritmo possui complexidade $T(l, k_C) \in O(l \cdot \log k_C)$.

2.2.12 Complexidade de `get_element_avl`

O algoritmo faz duas chamadas para algoritmos logarítmicos na quantidade de nós da árvore, além de chamadas de tempo constante. Assim, garantimos complexidade $T(k) \in O(1 + \log k)$.

2.2.13 Complexidade de `insert_element_avl`

O algoritmo faz oito chamadas para algoritmos logarítmicos na quantidade de nós da árvore, além de chamadas de tempo constante. Assim, garantimos complexidade $T(k) \in O(1 + \log k)$.

2.2.14 Complexidade de `delete_element_avl`

O algoritmo faz sete chamadas para algoritmos logarítmicos na quantidade de nós da árvore, além de chamadas de tempo constante. Assim, garantimos complexidade $T(k) \in O(1 + \log k)$.

2.2.15 Complexidade de `transpose_avl`

O algoritmo faz apenas operações de tempo constante, garantindo $T(k) \in O(1)$.

2.2.16 Complexidade de `scalar_mul_avl`

O algoritmo faz no máximo uma quantidade constante de chamadas parra algoritmos lineares na quantidade de nós da árvore, além de chamadas de tempo constante. Assim, garantimos complexidade $T(k) \in O(1 + k)$.

2.2.17 Complexidade de `sum_avl`

O algoritmo chama um algoritmo linear na quantidade de nós para a matriz B, fazendo esforço proporcional a k_B. Ele também faz uma chamada a um algoritmo linear para a matriz A, fazendo esforço proporcional a k_A e alocando k_A de memória (o que ainda garante uso de memória $O(k)$). Por último, para cada elemento k_A da matriz esparsa linearizada, são chamados dois algoritmos que fazem esforço logarítmico na matriz C, com k elementos. Assim, temos esforço computacional proporcional a $k_A + k_B + k_A \cdot \log k < k_A + k_B \cdot \log k + k_A \cdot \log k$ o que implica em $T(k_A, k_B, k) \in O(1 + (k_A + k_B) \log k)$.

2.2.18 Complexidade de `matrix_mul_avl`

O algoritmo começa com chamadas a `_free_o_tree`, o que pode ser desconsiderado, já que faz esforço computacional constante se a matriz C estiver vazia por estipulação. A chamada dessa função é uma mera precaução. O algoritmo reserva espaço proporcional a $3k$, o que não viola o requisito de memória $O(k)$. Depois, um algoritmo linear é chamado em A, fazendo esforço computacional proporcional a k_A. Então, para cada elemento de um vetor de tamanho k_A, é feito esforço computacional $\log(r_B)$, sendo r_B a quantidade de linhas não-nulas da matriz, ainda tendo que $r_B < k_N$. Então, para cada elemento k_A é feito no máximo esforço $c_B \log(k_C)$, sendo c_B a quantidade de colunas não-nulas em B. Denotando por $m_B = \max\{c_B, r_B\}$, a complexidade total é proporcional a $k_A + k_A \log r_B + k_A c_B \log k_C < k_A \log m_B \log k_C + k_A \log m_B \log k_C + k_A \log m_B \log k_C = k_A \log k_B \log k_C < 3k_A k_B \log k_C \in O(1 + k_A k_B (1 + \log(1 + k_C)))$. De fato, uma análise assintótica mais justa para o algoritmo é $O(1 + k_A + k_A \cdot \log(1 + m_B) \log(1 + k_C))$

2.2.19 Complexidade de `create_matrix_avl`

O algoritmo faz apenas operações de tempo constante, garantindo $T(k) \in O(1)$.

2.2.20 Complexidade de `free_matrix_avl`

O algoritmo faz duas chamadas para algoritmos lineares na quantidade de nós e faz esforço constante, garantindo $T(k) \in O(k)$.

3 Análise Experimental

3.1 Dados

3.1.1 Resultados de Tamanho

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	39.06 KB	11.44 KB	4.38 KB
100	0.05	501	39.06 KB	39.08 KB	19.77 KB
100	0.1	1001	39.06 KB	70.41 KB	39.49 KB
100	0.2	2001	39.06 KB	132.91 KB	78.93 KB
1000	0.01	10000	3.81 MB	703.16 KB	362.41 KB
1000	0.05	50001	3.81 MB	3.13 MB	2.14 MB
1000	0.1	100001	3.81 MB	6.18 MB	4.29 MB
1000	0.2	200001	3.81 MB	12.28 MB	8.58 MB
10000	1e-08	1	381.47 MB	176.00 B	184.00 B
10000	1e-07	11	381.47 MB	1.58 KB	424.00 B
10000	1e-06	100	381.47 MB	14.09 KB	4.38 KB
100000	1e-09	10	37.25 GB	1.44 KB	400.00 B
100000	1e-08	100	37.25 GB	14.09 KB	4.38 KB
100000	1e-07	1001	37.25 GB	140.45 KB	39.49 KB
1000000	1e-10	101	3.64 TB	14.23 KB	4.40 KB
1000000	1e-09	1000	3.64 TB	140.62 KB	39.47 KB
1000000	1e-08	10000	3.64 TB	1.37 MB	362.41 KB

Tabela 2: Tamanhos (bytes) para Dense, AVL e Hash.

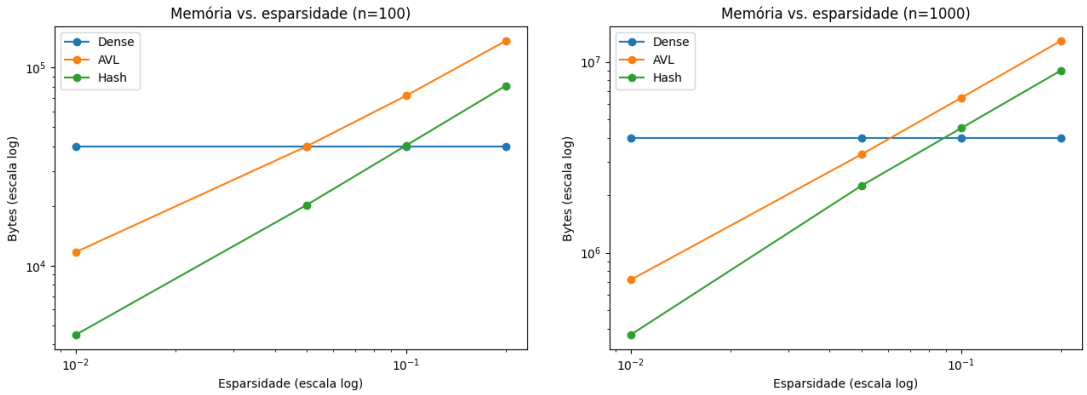

Figura 1: Memória vs. esparsidade (log-log) para $n = 100$ e $n = 1000$.

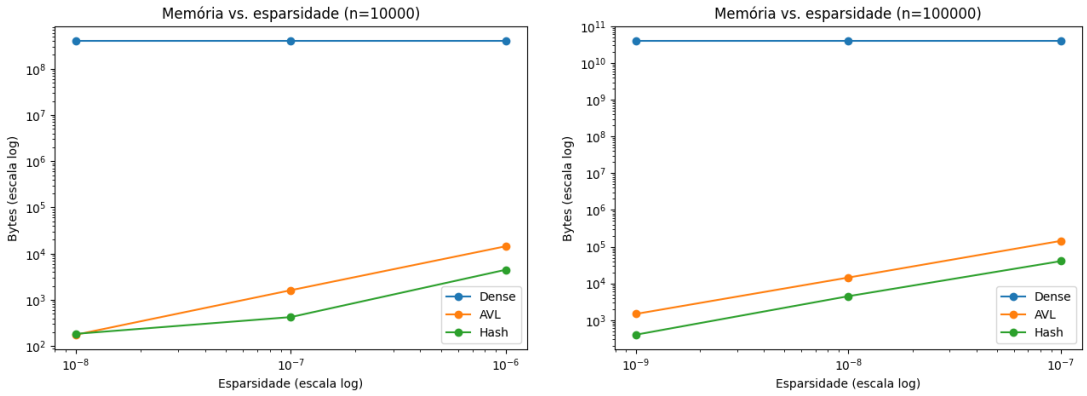

Figura 2: Memória vs. esparsidade (log-log) para $n = 10{,}000$ e $n = 100{,}000$.

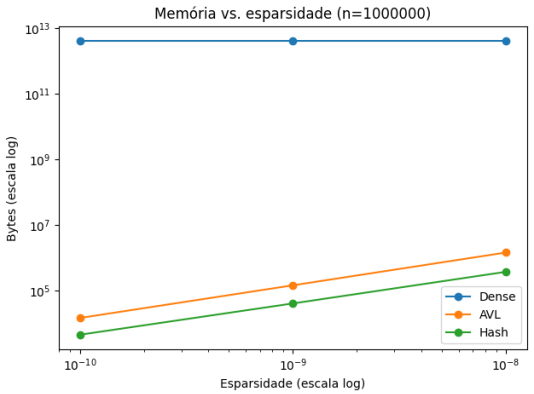

Figura 3: Memória vs. esparsidade (log-log) para $n = 1{,}000{,}000$.

3.1.2 Resultados de Tempo

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	100 ns	100 ns	100 ns
100	0.05	501	< 100 ns	200 ns	100 ns
100	0.1	1001	100 ns	200 ns	100 ns
100	0.2	2001	< 100 ns	200 ns	100 ns
1000	0.01	10000	< 100 ns	300 ns	100 ns
1000	0.05	50001	100 ns	300 ns	100 ns
1000	0.1	100001	< 100 ns	600 ns	200 ns
1000	0.2	200001	< 100 ns	700 ns	400 ns
10000	1e-08	1	—	100 ns	100 ns
10000	1e-07	11	—	< 100 ns	< 100 ns
10000	1e-06	100	—	< 100 ns	100 ns
100000	1e-09	10	—	100 ns	< 100 ns
100000	1e-08	100	—	100 ns	100 ns
100000	1e-07	1001	—	200 ns	100 ns
1000000	1e-10	101	—	100 ns	100 ns
1000000	1e-09	1000	—	100 ns	100 ns
1000000	1e-08	10000	—	300 ns	100 ns

Tabela 3: Temporização de `get`

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	< 100 ns	200 ns	100 ns
100	0.05	501	100 ns	200 ns	< 100 ns
100	0.1	1001	< 100 ns	300 ns	< 100 ns
100	0.2	2001	100 ns	300 ns	< 100 ns
1000	0.01	10000	< 100 ns	300 ns	< 100 ns
1000	0.05	50001	< 100 ns	400 ns	< 100 ns
1000	0.1	100001	< 100 ns	600 ns	100 ns
1000	0.2	200001	< 100 ns	500 ns	100 ns
10000	1e-08	1	—	200 ns	< 100 ns
10000	1e-07	11	—	100 ns	100 ns
10000	1e-06	100	—	100 ns	100 ns
100000	1e-09	10	—	100 ns	< 100 ns
100000	1e-08	100	—	200 ns	< 100 ns
100000	1e-07	1001	—	100 ns	< 100 ns
1000000	1e-10	101	—	100 ns	100 ns
1000000	1e-09	1000	—	200 ns	100 ns
1000000	1e-08	10000	—	300 ns	< 100 ns

Tabela 4: Temporização de `set`

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	2.40 µs	100 ns	100 ns
100	0.05	501	2.40 µs	< 100 ns	100 ns
100	0.1	1001	2.40 µs	< 100 ns	< 100 ns
100	0.2	2001	2.40 µs	< 100 ns	< 100 ns
1000	0.01	10000	430.30 µs	< 100 ns	< 100 ns
1000	0.05	50001	366.10 µs	< 100 ns	100 ns
1000	0.1	100001	444.40 µs	< 100 ns	100 ns
1000	0.2	200001	390.40 µs	< 100 ns	100 ns
10000	1e-08	1	—	100 ns	< 100 ns
10000	1e-07	11	—	< 100 ns	< 100 ns
10000	1e-06	100	—	< 100 ns	100 ns
100000	1e-09	10	—	< 100 ns	< 100 ns
100000	1e-08	100	—	< 100 ns	< 100 ns
100000	1e-07	1001	—	< 100 ns	100 ns
1000000	1e-10	101	—	100 ns	100 ns
1000000	1e-09	1000	—	< 100 ns	< 100 ns
1000000	1e-08	10000	—	< 100 ns	< 100 ns

Tabela 5: Temporização de `transpose`

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	2.40 µs	6.60 µs	5.70 µs
100	0.05	501	2.40 µs	19.90 µs	26.60 µs
100	0.1	1001	2.40 µs	33.40 µs	34.40 µs
100	0.2	2001	2.30 µs	62.90 µs	64.40 µs
1000	0.01	10000	236.90 µs	366.70 µs	362.10 µs
1000	0.05	50001	243.80 µs	1.84 ms	2.60 ms
1000	0.1	100001	217.00 µs	3.43 ms	4.99 ms
1000	0.2	200001	221.40 µs	8.18 ms	9.88 ms
10000	1e-08	1	—	700 ns	200 ns
10000	1e-07	11	—	800 ns	300 ns
10000	1e-06	100	—	5.30 µs	4.80 µs
100000	1e-09	10	—	700 ns	300 ns
100000	1e-08	100	—	4.30 µs	5.10 µs
100000	1e-07	1001	—	44.10 µs	30.50 µs
1000000	1e-10	101	—	5.40 µs	4.20 µs
1000000	1e-09	1000	—	44.30 µs	30.70 µs
1000000	1e-08	10000	—	553.50 µs	248.70 µs

Tabela 6: Temporização de `scalar_mul`

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	2.50 µs	10.70 µs	6.10 µs
100	0.05	501	2.40 µs	50.10 µs	33.00 µs
100	0.1	1001	2.40 µs	105.40 µs	44.30 µs
100	0.2	2001	2.40 µs	221.90 µs	83.70 µs
1000	0.01	10000	242.90 µs	1.22 ms	434.20 µs
1000	0.05	50001	217.80 µs	7.90 ms	3.10 ms
1000	0.1	100001	214.80 µs	15.79 ms	6.27 ms
1000	0.2	200001	245.80 µs	36.39 ms	12.99 ms
10000	1e-08	1	—	500 ns	300 ns
10000	1e-07	11	—	1.40 µs	600 ns
10000	1e-06	100	—	7.70 µs	5.10 µs
100000	1e-09	10	—	1.00 µs	600 ns
100000	1e-08	100	—	6.50 µs	5.40 µs
100000	1e-07	1001	—	3.71 ms	38.90 µs
1000000	1e-10	101	—	8.20 µs	6.30 µs
1000000	1e-09	1000	—	80.50 µs	38.70 µs
1000000	1e-08	10000	—	1.09 ms	368.20 µs

Tabela 7: Temporização de `sum`

n	sparsity	k	Dense	AVL	Hash
100	0.01	100	310.00 µs	14.30 µs	21.30 µs
100	0.05	501	309.90 µs	267.30 µs	418.40 µs
100	0.1	1001	327.80 µs	1.15 ms	1.67 ms
100	0.2	2001	318.20 µs	4.14 ms	6.20 ms
1000	0.01	10000	438.25 ms	16.82 ms	250.56 ms
1000	0.05	50001	439.00 ms	617.18 ms	20.23 s
1000	0.1	100001	468.95 ms	1.84 s	83.25 s
1000	0.2	200001	437.42 ms	7.44 s	329.74 s
10000	1e-08	1	—	200 ns	100 ns
10000	1e-07	11	—	300 ns	400 ns
10000	1e-06	100	—	1.70 µs	15.60 µs
100000	1e-09	10	—	300 ns	600 ns
100000	1e-08	100	—	1.70 µs	13.10 µs
100000	1e-07	1001	—	30.90 µs	933.60 µs
1000000	1e-10	101	—	2.00 µs	12.50 µs
1000000	1e-09	1000	—	21.60 µs	998.00 µs
1000000	1e-08	10000	—	366.80 µs	278.44 ms

Tabela 8: Temporização de `matmul`

3.2 Interpretação

Através da análise dos dados experimentais e da compreensão da diferença entre um pior caso assintótico garantido e um pior caso assintótico esperado, é possível tirar algumas conclusões:

Primeiro, pelos dados é possível observar que o uso de matrizes densas, em geral, é mais vantajoso para pequenos tamanhos. Se é possível armazenar a matriz por completo, as operações são ordens de grandeza mais rápidas, e pensando em memória, o *overhead* gerado pelas implementações propostas pode ser maior em tamanhos pequenos. Porém, para grandes matrizes, a situação é a inversa – os dados experimentais param em $n = 10^3$ pois o tempo para a execução das multiplicações de matrizes densas era impraticável, e para $n = 10^4$ é virtualmente impossível armazenar matrizes densas em hardware de consumidor. Já para as implementações de AVL e Hash, o baixo grau de esparsidade especificado pelo enunciado permite as operações em tempo baixíssimo, com uso de memória totalmente viável. Mesmo em graus de esparsidade ligeiramente mais altos, a dependência assintótica da memória apenas nos dados efetivamente armazenados torna viável o uso dessas estruturas. Dessa forma, pode-se concluir que o uso mais vantajoso desse tipo de representação de matriz esparsa é para armazenar e operar em cima de dados matrici-

ais de grandes ordens – com a implementação densa, existem aplicações simplesmente impossíveis, que tornam-se viáveis com o uso das estruturas esparsas.

Entre as duas estruturas propostas, existem motivos para usar uma *versus* a outra. Em todas as operações com exceção da `matmul`, a estrutura baseada em Hash se mostrou experimentalmente mais rápida, mas suas garantias são fracas – estatisticamente, ela respeita os limites impostos, mas os limites para o pior caso possível são altíssimos comparados à implementação por AVL. Assim, em aplicações de tempo real com necessidade de garantias fortes, é péssima prática utilizar uma estrutura que tem sua performance totalmente dependente na distribuição dos dados de entrada.

Referências

[1] GILBERG, R. F.; FOROUZAN, B. A. *Data Structures: A Pseudocode Approach with C*. Cengage Learning, 2004.