

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.

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 `setElement`, 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 `getElement`

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 `setElement`

O custo esperado da operação `setElement` é parecido ao de `getElement`. 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, dobrarmos 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 dobrarmos 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 `matrixMultiplication`

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 `matrixAddition`

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 `matrixScalarMultiplication`

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

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
<code>avl_status_string</code>	$O(1)$
<code>create_matrix_avl</code>	$O(1)$
<code>transpose_avl</code>	$O(1)$
<code>_allocation_fail</code>	$O(1)$
<code>_validate_matrix</code>	$O(1)$
<code>_validate_indices</code>	$O(1)$
<code>_validate_same_dimensions</code>	$O(1)$
<code>_height_i</code>	$O(1)$
<code>_height_o</code>	$O(1)$
<code>_balance_factor_i</code>	$O(1)$
<code>_balance_factor_o</code>	$O(1)$
<code>_max</code>	$O(1)$
<code>_left_rotate_i</code>	$O(1)$
<code>_right_rotate_i</code>	$O(1)$
<code>_left_rotate_o</code>	$O(1)$
<code>_right_rotate_o</code>	$O(1)$

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 contém árvores de nós "internos", que contêm 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 para 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 $T(k) \in O(k)$.