MC458 - Projeto

Juca Magalhães Meniconi
RA: 281803

Lucas Beserra Fernandes
RA: 281815

Vinícius Augusto Silva Brasileiro
RA: 247391

1 Descrição das Estruturas

1.1 Estrutura 1 - Hash Table

1.2 Estrutura 2 - AVL

2 Análise das Complexidades

2.1 Estrutura 1 - Hash Table

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 $T(k) \in O(\log k)$.

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 $T(k) \in O(\log k)$.

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 $T(k) \in O(\log k)$.

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 $T(k) \in O(k)$.

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 $T(k_A, k_B, k) \in O((k_A + k_B) \log k)$.

2.2.18 Complexidade de `matrix_mul_avl`

NÃO TO CONSEGUINDO FECHAR ESSA PROVA
Dois problemas: não consigo fazer o d_B aparecer pq ele é definido pelo número médio de elementos em linhas não nulas da matriz B, e não consigo fazer o log de k_B ser absorvido na complexidade.

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 $T(k) \in O(k)$.