Universidade Estadual de Campinas
Instituto de Computação da Unicamp

Projeto Final de MC458

Trabalho que desenvolve duas implementações de matrizes esparsas: a primeira combina uma tabela hash com listas ligadas, enquanto a segunda utiliza uma árvore AVL.

Lucas V. Lima (247025)
Kauan C. da Silva (240030)
Bruno M. Saback (281746)

1 DESCRIÇÃO DA ESTRUTURA DE DADOS (HASH)

A matriz esparsa é representada por uma combinação de duas estruturas de indexação que permitem otimizar tanto o acesso direto à posição (i, j) quanto operações que percorrem todos os elementos não nulos, como soma e multiplicação.

1.1 Nós de Armazenamento

Cada elemento não nulo é armazenado em um nó da seguinte forma:

```
struct No_hash {
int i, j;                   // coordenadas
    int valor;              // valor armazenado
    No_hash *prox_hash;     // proximo no bucket da hash
    No_hash *prox_todos;    // proximo na lista global
};
```

O ponteiro *prox_hash* resolve colisões na tabela hash através de encadeamento separado, enquanto o ponteiro *prox_todos* conecta todos os elementos não nulos da matriz em uma única lista global.

1.2 Estrutura da Matriz

A matriz esparsa contém:

- **tabela_hash**: vetor de tamanho variável, redimensionado automaticamente por *rehash* sempre que o fator de carga ultrapassa o limite (0.75). Cada posição do vetor é um bucket que armazena uma lista encadeada de elementos com o mesmo valor de hash.
- **lista_todos**: lista encadeada contendo todos os elementos não nulos da matriz, utilizada em percursos gerais como soma e multiplicação.
- **usados**: número de elementos atualmente armazenados.

1.3 Estrutura para Transposta

Para permitir que a operação de transposição A^T que tenha custo $O(1)$, utiliza-se a seguinte estrutura:

```
struct Matrizes {
    p_matriz_esparsa normal;
    p_matriz_esparsa transposta;
};
```

A matriz transposta é atualizada em paralelo durante as inserções e demais operações (eg. $A \times B = C, C^T = B^T \times A^T$), assim, basta retornar o ponteiro correspondente a transposta. Isso garante a complexidade constante.

Observação: a função de hashing utilizada foi retirada de uma implementação pública amplamente empregada para dispersão eficiente de pares de inteiros (a função MurmurHash3 para a chave composta (i, j)).

2 ANÁLISE TEÓRICA DE COMPLEXIDADE

Denotemos:

- k: número de elementos não nulos,
- n: número de linhas,
- m: número de colunas,
- T: tamanho atual da tabela hash.

O tamanho da tabela hash **não é constante**: ele cresce proporcionalmente ao número de elementos, mantendo o fator de carga (α) abaixo de um limite fixo, $L = 0.75$.

$$\alpha = \frac{k}{T} \leq L \quad \Longrightarrow \quad T \geq \frac{k}{L}$$

Dessa forma, $T = \Theta(k)$, e a carga permanece $\alpha = O(1)$. Isso garante bom desempenho médio, pois o tamanho esperado das listas de colisão é constante.

2.1 Uso de Memória

A estrutura de dados utiliza k nós para armazenar os elementos não nulos e um vetor de ponteiros de tamanho T para a tabela hash.

O consumo total de memória é dado por:

$$\text{Memória} = O(\underbrace{k}_{\text{Nós}} + \underbrace{T}_{\text{Tabela Hash}})$$

Como $T = \Theta(k)$, o custo de memória é:

$$O(k)$$

O consumo é estritamente proporcional à quantidade de elementos não nulos, o que é ideal para matrizes esparsas.

2.2 Acesso e Inserção $\mathbf{A}[i, j]$

O processo de acesso ou inserção envolve três passos principais:

1. Cálculo do valor de hash da chave (i, j): $O(1)$.
2. Acesso ao bucket na tabela hash: $O(1)$.

3. Travessia da lista encadeada do bucket para buscar a chave: $\Theta(\ell)$, onde ℓ é o comprimento da lista de colisões para aquela chave.

Complexidade Esperada (Média)

O desempenho da tabela hash por encadeamento é determinado pelo fator de carga α. Se a função de hashing distribui as chaves uniformemente, o comprimento esperado da lista de colisões é $E[\ell] = \alpha$. Como o *rehash* garante que $\alpha = k/T \leq L = 0.75$, o comprimento esperado é constante:

$$\text{Acesso e Inserção Esperados} = O(1) + O(E[\ell]) = O(1).$$

Uma inserção pode ocasionalmente (em média com $O(1)$ custo amortizado) acionar um *rehash*, que custa $O(k)$ para reconstruir a tabela. No entanto, o custo amortizado da inserção, considerando o rehash, também é $O(1)$.

Complexidade de Pior Caso

No pior caso, se todos os k elementos não nulos colidirem no mesmo bucket (situação altamente improvável com uma boa função de hash), a travessia da lista encadeada degenera para busca linear:

$$\text{Acesso e Inserção Pior Caso} = O(1) + O(k) = O(k).$$

2.3 Transposta A^T

A transposta é armazenada em paralelo em uma segunda estrutura (*matrizes->transposta*).

- **Criação da Transposta**: Ao criar o objeto *Matrizes*, o custo é apenas $O(1)$ para alocar e inicializar a segunda estrutura (*matrizes->transposta*).
- **Manutenção**: Cada operação de *inserir_atualizar_matrizes* insere/atualiza $A[i, j]$ e $A^T[j, i]$ (com coordenadas trocadas). O custo é dobrado, mas $O(1) + O(1) = O(1)$ em média.
- **Acesso à Transposta**: Retorna um ponteiro para a estrutura já existente.

$$\text{Transposta} = O(1)$$

2.4 Soma de Matrizes $C = A + B$

A soma é implementada percorrendo todos os elementos não nulos de A e de B, e inserindo/atualizando-os na matriz resultado C.

1. Percorrer A: Os k_A elementos são percorridos pela *lista_todos*. Custo $O(k_A)$.
2. Inserir A em C: Cada elemento de A é inserido em C. Como a inserção em C custa $O(1)$ em média, o custo total é $O(k_A) \times O(1) = O(k_A)$.
3. Percorrer B: Os k_B elementos são percorridos pela *lista_todos*. Custo $O(k_B)$.
4. Somar B em C: Para cada elemento $B[i, j]$, acessa-se $C[i, j]$ (custo $O(1)$ em média), e o resultado é inserido/atualizado em C (custo $O(1)$ em média).

Complexidade Esperada (Média)

O custo total é dominado pelos percursos e pelas operações de inserção/acesso em C:

$$\text{Soma Esperada} = O(k_A) + O(k_B) = O(k_A + k_B).$$

Complexidade de Pior Caso

Se as operações internas de acesso/inserção na matriz C degenerarem para o pior caso $O(k_C)$ (onde k_C é o número de elementos não nulos em C, $k_C \leq k_A + k_B$):

- Inserir A em C: $k_A \times O(k_C) = O(k_A k_C)$.
- Acessar/Atualizar C com elementos de B: $k_B \times O(k_C) = O(k_B k_C)$.

$$\text{Soma Pior Caso} = O(k_A k_C + k_B k_C).$$

2.5 Multiplicação por Escalar αA

A operação percorre a *lista_todos* uma única vez e multiplica o *valor* de cada nó. Esta operação não envolve buscas ou inserções na tabela hash.

$$\text{Multiplicação por Escalar} = O(k).$$

O custo é o mesmo para o caso esperado e o pior caso.

2.6 Multiplicação de Matrizes $C = A \times B$

A implementação percorre todos os k_A elementos de A e, para cada um, percorre todos os k_B elementos de B.

O algoritmo executa $k_A \times k_B$ iterações no loop mais interno. Dentro deste loop, a condição $(b \to i == k)$ filtra os elementos $B[k, j]$ para o cálculo do termo $C[i, j] = \sum_k A[i, k]B[k, j]$.

Para cada iteração (onde (a, b) são nós de A e B):

$$\text{Operação Interna} = \underbrace{O(1)}_{\text{Acesso (chave } i,j)} + \underbrace{O(1)}_{\text{Inserção/Atualização}}$$

O custo total é o número de iterações multiplicado pelo custo da operação interna.

Complexidade Esperada (Média)

Com acesso e inserção esperados em $O(1)$:

$$\text{Multiplicação Esperada} = \sum_{a \in A} \sum_{b \in B} O(1) = O(k_A k_B).$$

Complexidade de Pior Caso

Se as operações de acesso/inserção na matriz C degenerarem para o pior caso $O(k_C)$:

$$\text{Multiplicação Pior Caso} = \sum_{a \in A} \sum_{b \in B} O(k_C) = O(k_A k_B \, k_C).$$

2.7 Resumo das Complexidades

Tabela 1 – Complexidades da estrutura implementada.

Operação	**Complexidade Esperada**	**Pior Caso**
Memória total	$O(k)$	$O(k)$
Acessar $A[i, j]$	$O(1)$	$O(k)$
Inserir/Atualizar $A[i, j]$	$O(1)$ (amortizado)	$O(k)$
Transposta A^T (acesso/retorno)	$O(1)$	$O(1)$
Soma $A + B$	$O(k_A + k_B)$	$O(k_A k_C + k_B k_C)$
Multiplicação por escalar	$O(k)$	$O(k)$
Multiplicação $A \times B$	$O(k_A k_B)$	$O(k_A k_B \, k_C)$

3 DESCRIÇÃO DA ESTRUTURA DE DADOS (ÁRVORE AVL ANINHADA)

A segunda estrutura escolhida para a representação da matriz esparsa baseia-se em uma abordagem hierárquica utilizando árvores binárias de busca balanceadas (AVL). A matriz é representada como uma "árvore de árvores", onde a estrutura externa indexa as linhas e as estruturas internas indexam as colunas. Essa abordagem elimina o uso de vetores, melhorando a complexidade assintótica das operações em Matriz Esparsa que acessam todos os elementos

não nulos, i.e, Multiplicação de Matrizes, Multiplicação de Escalar, etc.

3.1 Nós de Armazenamento

A estrutura é composta por dois tipos de nós. O nó de linha (*AVL_Linha*) serve como índice para as linhas existentes, enquanto o nó de coluna (*AVL_Coluna*) armazena efetivamente o valor e a coordenada da coluna.

```
struct AVL_Coluna {
    int j;              // indice da coluna
    int alt;            // altura para balanceamento
    int valor;          // valor armazenado
    p_avl_coluna esq;   // filho a esquerda
    p_avl_coluna dir;   // filho a direita
};

struct AVL_Linha {
    int i;              // indice da linha
    int alt;            // altura para balanceamento
    p_avl_coluna col;   // raiz da arvore de colunas desta linha
    p_avl_linha esq;    // filho a esquerda
    p_avl_linha dir;    // filho a direita
};
```

Esta separação permite que a busca por um elemento $A[i, j]$ seja realizada em duas etapas: primeiro busca-se a linha i na árvore externa e, em seguida, busca-se a coluna j na árvore interna correspondente.

3.2 Estrutura da Matriz

A matriz esparsa é encapsulada em um contêiner principal que gerencia tanto a matriz original quanto a sua transposta:

- **matriz**: ponteiro para a raiz da árvore AVL de linhas (*p_avl_linha*). Representa a matriz A acessível por $linha \rightarrow coluna$.
- **transposta**: ponteiro para a raiz de uma estrutura idêntica, mas onde os índices i e j são invertidos durante a inserção.

3.3 Estrutura para Transposta

A fim de cumprir o requisito de obtenção da transposta em $O(1)$, a estrutura mantém a transposta atualizada em tempo real:

```
struct Matriz_Arvore {
    p_avl_linha matriz;
    p_avl_linha transposta;
};
```

A função *transposta(matriz)* cria um novo contêiner onde os ponteiros *matriz* e *transposta* são invertidos, permitindo acesso imediato sem necessidade de cópia profunda ou reprocessamento dos dados.

4 ANÁLISE TEÓRICA DE COMPLEXIDADE

Denotemos:

- k: número total de elementos não nulos na matriz.
- N_L: número de linhas não vazias ($N_L \leq \min(n, k)$).
- k_i: número de elementos não nulos na linha i.

Diferentemente da tabela hash, a árvore AVL garante que a altura da árvore seja sempre limitada por $O(\log N)$, onde N é o número de nós na árvore. Não há redimensionamento de vetores (*rehash*), o que oferece tempos de execução mais previsíveis (pior caso garantido).

4.1 Uso de Memória

A estrutura aloca memória dinamicamente para cada elemento não nulo (um nó *AVL_Coluna*) e para cada linha não vazia (um nó *AVL_Linha*).

O consumo total de memória é:

$$\text{Memória} = O(\underbrace{k}_{\text{Nós Coluna}} + \underbrace{N_L}_{\text{Nós Linha}})$$

Como o número de linhas não vazias nunca excede o número de elementos ($N_L \leq k$), o consumo assintótico é:

$$O(k)$$

A estrutura é extremamente eficiente em espaço, alocando apenas o estritamente necessário.

4.2 Acesso e Inserção $\mathbf{A}[i, j]$

O acesso a um elemento envolve duas descidas em árvores AVL:

1. Busca na árvore de linhas para encontrar i: Custo $O(\log N_L)$.
2. Busca na árvore de colunas para encontrar j: Custo $O(\log k_i)$.

Como $N_L \leq k$ e $k_i \leq k$, a complexidade é limitada por $O(\log k)$.

Complexidade Garantida (Pior Caso)

Graças ao balanceamento da AVL, não existem casos degenerados (como listas encadeadas). Portanto:

$$\text{Acesso e Inserção} = O(\log k).$$

Esta é a principal vantagem desta estrutura sobre a hash: o pior caso é estritamente logarítmico, não linear.

4.3 Transposta A^T

Assim como na estrutura anterior, a transposta é mantida em paralelo.

- **Manutenção**: Na função *inserir_matriz*, cada inserção é realizada duas vezes: uma na *matriz* (custo $O(\log k)$) e outra na *transposta* (custo $O(\log k)$). O custo total permanece $O(\log k)$.
- **Acesso**: A função *transposta* apenas aloca um novo contêiner e inverte os ponteiros.

$$\text{Obter Transposta} = O(1)$$

4.4 Soma de Matrizes $C = A + B$

A operação de soma (*soma_matrizes*) realiza os seguintes passos:

1. **Cópia Profunda de A**: A função *copiar* realiza um percurso completo (DFS) na árvore de A para criar C. Como visita cada nó uma vez, o custo é $O(k_A)$.
2. **Fusão com B**: A função *somar_linhas* percorre a árvore B. Para cada elemento de B, realiza uma busca e inserção na estrutura de C.

Para cada um dos k_B elementos de B, realizamos uma operação de inserção na árvore C, que possui (no máximo) $k_A + k_B$ elementos. O custo de cada inserção é $O(\log(k_A + k_B))$.

$$\text{Soma Total} = O(k_A) + O(k_B \cdot \log(k_A + k_B)).$$

Simplificando, o custo é dominado pela inserção dos elementos de B na estrutura:

$$O(k_B \log k_{result})$$

4.5 Multiplicação por Escalar αA

A função *escalar* utiliza recursão (DFS) através das funções *mult_escalar* e *mult_escalar_coluna*. Ela visita cada nó da estrutura (linhas e colunas) exatamente uma vez para atualizar o valor. Não há buscas ou rotações.

$$\text{Multiplicação por Escalar} = O(k).$$

4.6 Multiplicação de Matrizes $C = A \times B$

O algoritmo implementado (*multiplicacao_matrizes*) utiliza a estratégia de iterar sobre os elementos não nulos.

1. Itera sobre todas as linhas de A.
2. Itera sobre todas as colunas de uma linha de A (elemento A_{il}).
3. Para cada A_{il}, busca a linha correspondente l na matriz B (B_{lj}). Esta busca custa $O(\log N_{L,B})$.
4. Itera sobre as colunas de B na linha l.
5. Realiza a soma do produto em $C[i, j]$. Isso requer uma busca/inserção na árvore C, com custo $O(\log k_C)$.

O número de operações internas é proporcional ao número de pares (A_{ik}, B_{kj}) que contribuem para o resultado. Se definirmos N_{ops} como o número de multiplicações elementares necessárias:

$$\text{Multiplicação} = O(N_{ops} \cdot \log k_C)$$

No pior caso, isso pode ser aproximado considerando d_B como o grau médio (número de elementos por linha) de B:

$$\text{Complexidade} = O(k_A \cdot d_B \cdot \log k_C)$$

O fator $\log k_C$ surge devido à necessidade de buscar a posição correta na árvore de resultados para acumular a soma.

4.7 Resumo das Complexidades

A Tabela 2 resume o desempenho da estrutura baseada em AVL. Note a consistência entre o caso esperado e o pior caso, característica fundamental das árvores balanceadas.

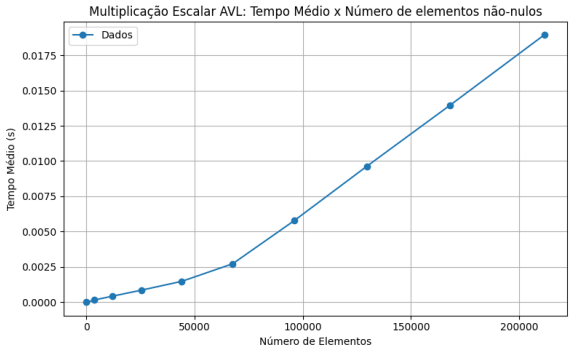
Tabela 2 – Complexidades da estrutura implementada (AVL).

Operação	Complexidade Esperada	Pior Caso (Garantido)
Memória total	$O(k)$	$O(k)$
Acessar $A[i, j]$	$O(\log k)$	$O(\log k)$
Inserir/Atualizar $A[i, j]$	$O(\log k)$	$O(\log k)$
Transposta A^T (acesso)	$O(1)$	$O(1)$
Soma $A + B$	$O(k_B \log k_C + k_A)$	$O(k_B \log k_C + k_A)$
Multiplicação por escalar	$O(k)$	$O(k)$
Multiplicação $A \times B$	$O(k_A \cdot d_B \cdot \log k_C)$	$O(k_A \cdot d_B \cdot \log k_C)$

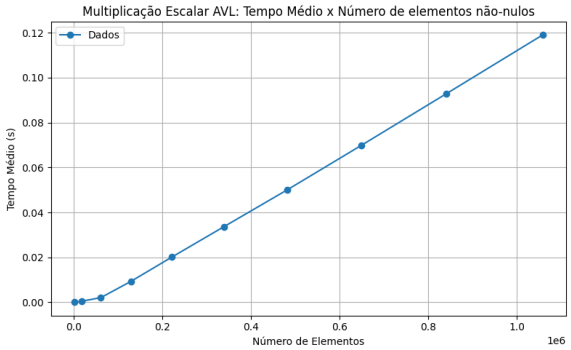
5 RESULTADOS EXPERIMENTAIS VISUALIZADOS

A seguir os gráficos mostram o tempo de execução para diferentes esparcidades em relação ao número de nós não nulos.

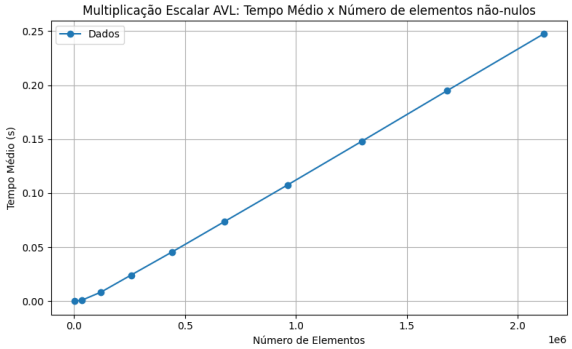
Figura 1 – Resultados para Multiplicação por Escalar (AVL)



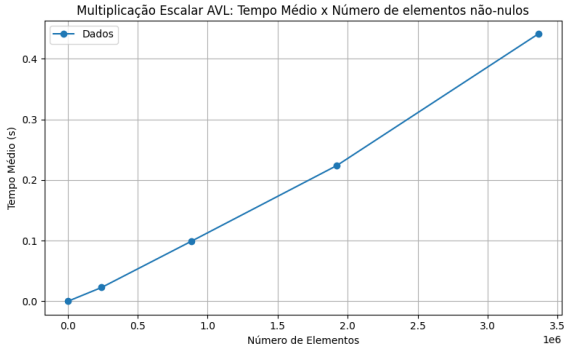
(a) Cenário: Um



(b) Cenário: Cinco

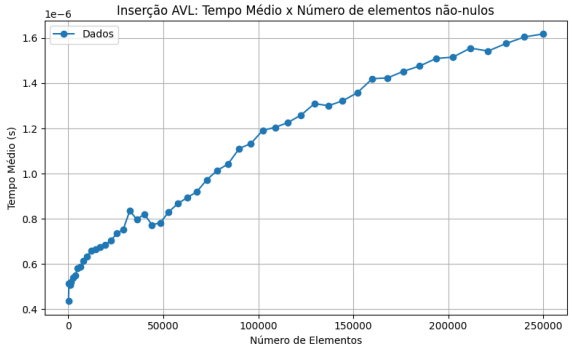


(c) Cenário: Dez

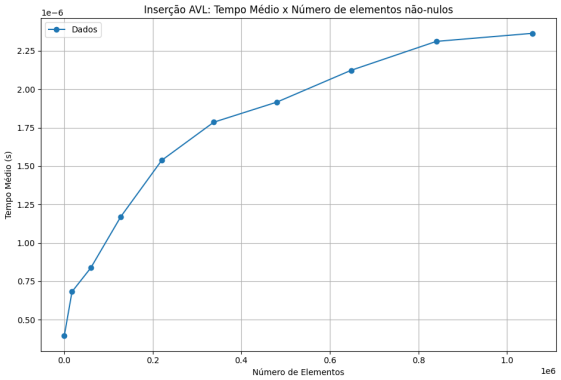


(d) Cenário: Vinte

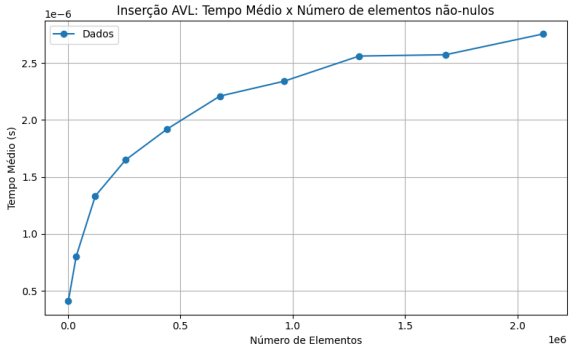
Figura 2 – Resultados para Inserção (AVL)



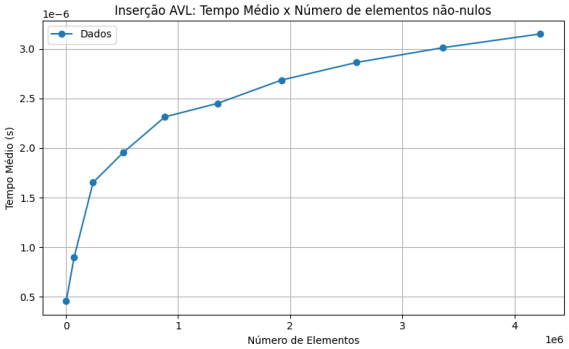
(a) Cenário: Um



(b) Cenário: Cinco

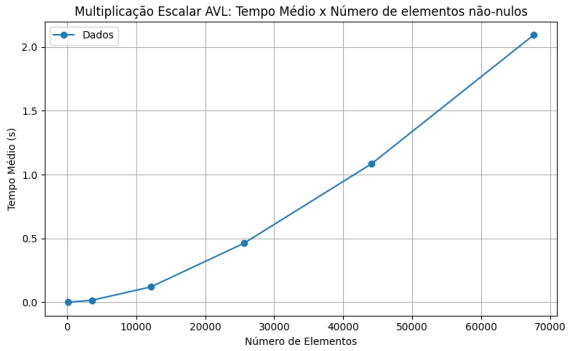


(c) Cenário: Dez

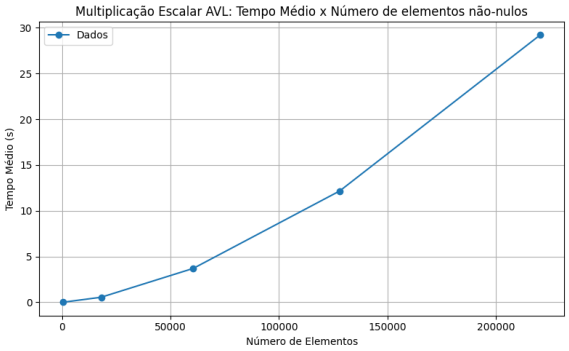


(d) Cenário: Vinte

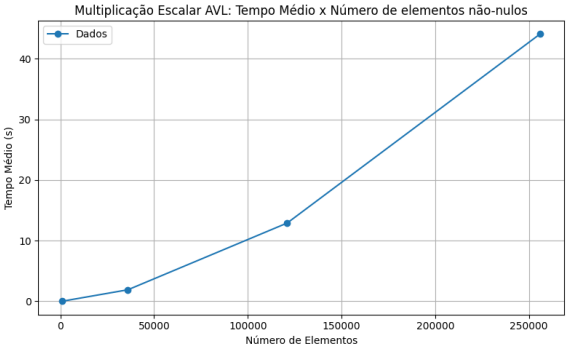
Figura 3 – Resultados para Multiplicação de Matrizes (AVL)



(a) Cenário: Um

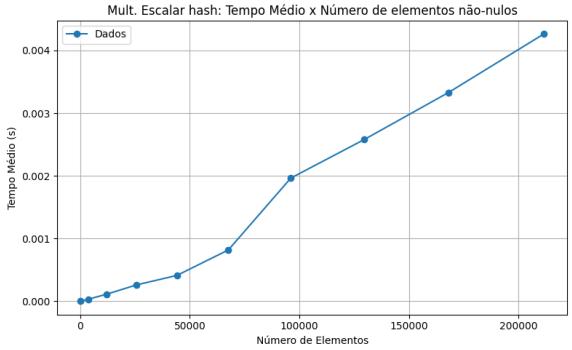


(b) Cenário: Cinco

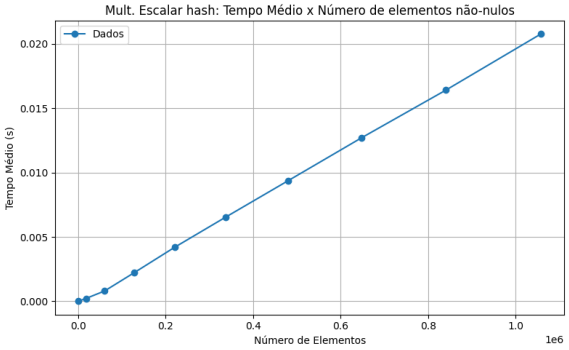


(c) Cenário: Dez

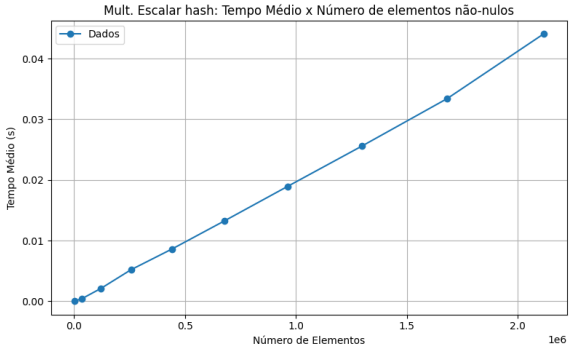
Figura 4 – Multiplicação por Escalar (Hash)



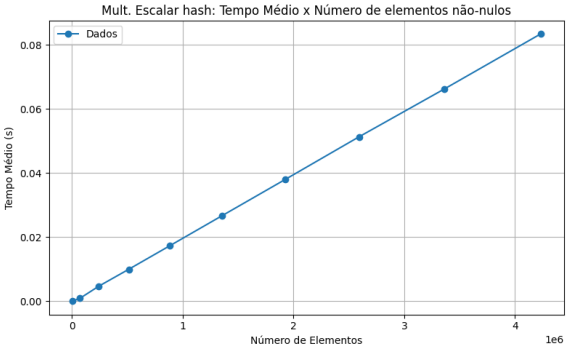
(a) Cenário: Um



(b) Cenário: Cinco

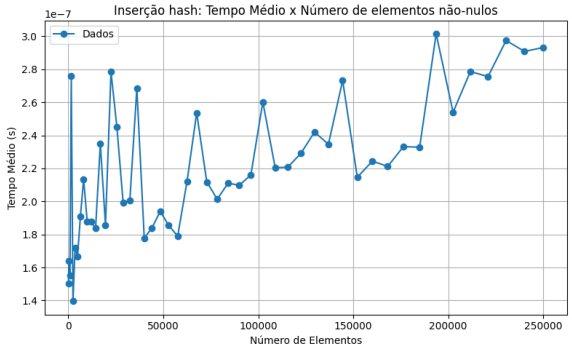


(c) Cenário: Dez

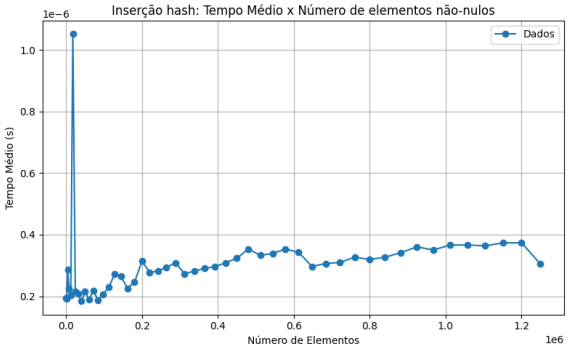


(d) Cenário: Vinte

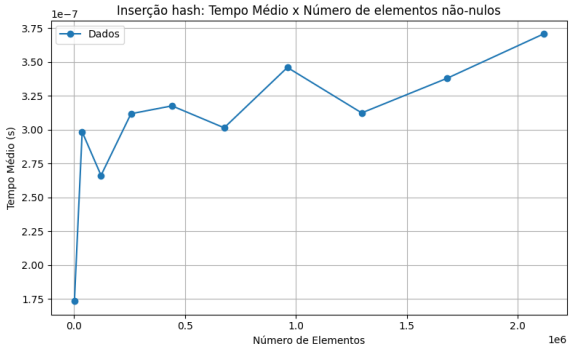
Figura 5 – Inserção de Elementos (Hash)



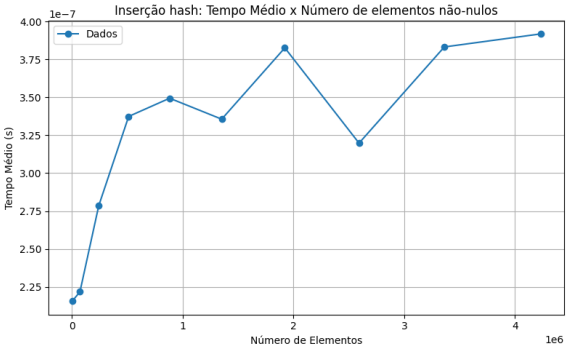
(a) Cenário: Um



(b) Cenário: Cinco



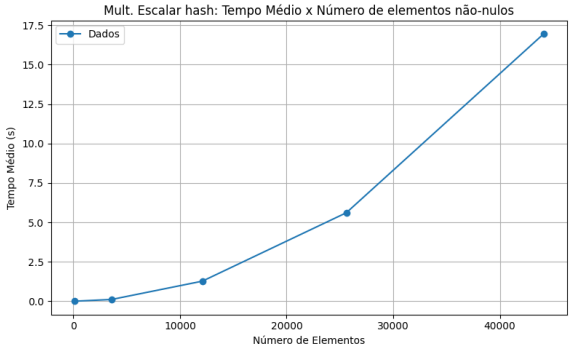
(c) Cenário: Dez



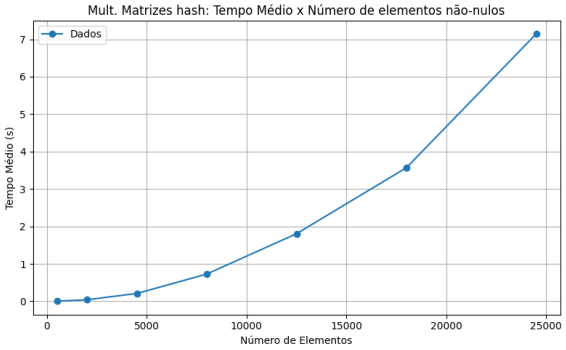
(d) Cenário: Vinte

Vale notar que os gráficos de inserção na estrutura Hash não apresentam uma linha perfeitamente constante, exibindo oscilações visíveis. Esse comportamento deve-se ao tamanho inicial reduzido da tabela, que força a execução frequente de operações de *rehash* (redimensionamento) conforme o fator de carga é atingido. O custo computacional dessas realocações gera picos momentâneos no tempo de execução, dificultando a visualização empírica imediata da complexidade $O(1)$ amortizada.

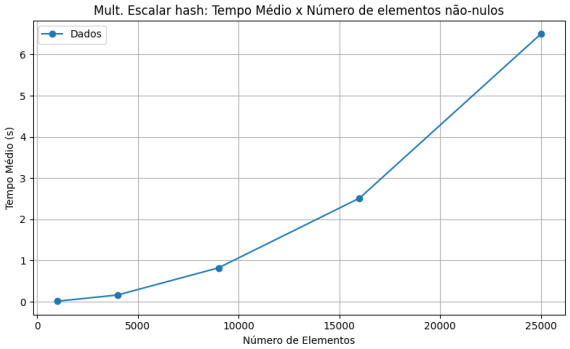
Figura 6 – Multiplicação de Matrizes (Hash)



(a) Cenário: Um



(b) Cenário: Cinco



(c) Cenário: Dez

6 CONCLUSÃO

A análise comparativa entre as implementações baseadas em Tabela Hash e Árvore AVL evidenciou um claro *trade-off* entre desempenho médio e previsibilidade. Enquanto a estrutura de Hash oferece complexidade amortizada $O(1)$, ideal para cenários de acesso aleatório intensivo, os resultados experimentais corroboraram que os custos de redimensionamento (*rehash*) introduzem oscilações de desempenho. Em contrapartida, a abordagem hierárquica com Árvores AVL, embora possua um custo teórico logarítmico $O(\log k)$, demonstrou superioridade na estabilidade das operações e na eficiência de memória, eliminando os riscos de pior caso linear ($O(k)$) associados a colisões. Portanto, conclui-se que a escolha entre as estruturas depende da criticidade da aplicação: a Hash é preferível para maximizar a velocidade média, enquanto a AVL é a escolha robusta para sistemas que exigem tempos de resposta consistentes e garantidos.

Referências

Austin Appleby. Murmurhash3. <https://github.com/aappleby/smhasher>, 2011.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.