Matrizes Esparsas como Aplicação de Listas Encadeadas: Um Projeto de Estrutura de Dados

Atílio G. Luiz[1], Calebe Mesquita da Silva[2], Francisco Emilson S. Souza Filho[2]

[1]Professor Associado Campus Quixadá (CE) – Universidade Federal do ceará (UFC)
Av.José de Freitas Queiroz, 5003 – Cedro – Quixadá – Ceará 63902-580 – Brazil

[2]Alunos do Campus Quixadá (CE) – Universidade Federal do ceará (UFC)
Av.José de Freitas Queiroz, 5003 – Cedro – Quixadá – Ceará 63902-580 – Brazil

gomes.atilio@ufc.br;

calebcamesquita@alu.ufc.br, francisco.filho24@alu.ufc.br

Abstract. *This work presents the implementation of a sparse matrix using circular plain linked lists for the Data Structure course at Universidade Federal do Ceará. It draws parallels to optimization solutions discussed in the literature and details the software development process.*

Resumo. *O presente trabalho apresenta a implementação de uma matriz esparsa usando listas circulares simplesmente encadeadas para o curso de Estrutura de Dados da Universidade Federal do Ceará. Além disso, ele traça paralelos com soluções de otimização discutidas na literatura e detalha o processo de desenvolvimento do software final.*

1. Introdução

Matrizes esparsas são definidas como aquelas em que a maioria de seus elementos é igual a zero, indicando que muitos valores não estão efetivamente presentes ou são desnecessários. A implementação desse tipo de matriz é especialmente útil em contextos onde as dimensões das matrizes são extremamente grandes, o que dificulta o processamento eficiente dos dados pelos processadores atuais, ou quando não todos os elementos da matriz serão utilizados. É relevante ressaltar que matrizes, em especial as multidimensionais, tornam-se inviáveis em software convencional ao atingirem grandes dimensões, pois demandam a alocação de espaço de memória para armazenar todos os elementos de uma matriz tradicional. Por exemplo, uma matriz de dimensões 30.000×30.000 requereria aproximadamente $9 \cdot 10^8$ espaços de memória, o que é excessivo, mesmo para computadores modernos.

Além disso, matrizes esparsas apresentam aplicações vantajosas em softwares destinados a cálculos científicos e engenharia. Outras aplicações diretas incluem a resolução de sistemas lineares na forma $Ax = b$ com múltiplas variáveis e a representação de grafos [Pissanetzki 1984], contribuindo para a resolução de problemas de álgebra linear. Uma forma clara de visualizar essa aplicação é em um programa de planilhas, onde muitos dados podem ser representados como uma matriz esparsa, em que uma célula não preenchida indica a ausência de um valor na matriz. Nesse contexto, é importante diferenciar entre matriz lógica e matriz física. A matriz física refere-se à representação da matriz como um todo; por exemplo, uma matriz A de dimensões 5×5 pode ser visualizada como uma estrutura completa.

Por outro lado, ao nos referirmos a matrizes lógicas, estamos tratando da interpretação da matriz na memória, sem a necessidade de armazenar os valores de todas as células que contêm zero, conforme mencionado anteriormente.

Embora este trabalho não trate diretamente de um software de planilhas, é possível estabelecer um paralelo com esse tipo de aplicação.

Existem diversas maneiras de implementar a ideia mencionada. Os autores [Pissanetzki 1984] e [Schildt 1997] apresentam algumas alternativas, como listas encadeadas, árvores binárias, *arrays* ou matrizes de ponteiros. Este artigo discutirá a abordagem que utiliza listas encadeadas circulares simplesmente encadeadas, que será detalhada posteriormente.

1.1. Justificativa

A justificativa para a escolha desta abordagem baseia-se na eficiência no uso da memória do computador. No entanto, ao acessar seus elementos, não é possível fazê-lo em tempo constante, como ocorre em matrizes convencionais, uma vez que é necessário percorrer os elementos para alcançar a célula ou nó desejado.

1.2. Objetivo

O objetivo deste trabalho é unificar o conhecimento adquirido em sala de aula com a pesquisa, visando elaborar um projeto que reforce o aprendizado dos alunos de forma prática, assegurando uma melhor fixação do conteúdo.

2. Revisão Bibliográfica

2.1. Listas Encadeadas

2.1.1. Estrutura de uma Lista Encadeada

Uma lista encadeada é uma estrutura de dados empregada na programação para representar uma coleção de elementos. Nesta estrutura, os dados são armazenados de maneira dispersa na memória, o que requer a interconexão de cada elemento. Cada dado, denominado *nó*, deve conter o endereço de memória do próximo item da lista [Bhargava 2017], assegurando que os dados permaneçam interligados.

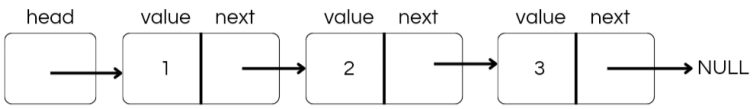

Figura 1. Representação de uma lista simplesmente encadeada

Em algumas abordagens, é possível optar por utilizar um recurso adicional: um nó sentinela no início da lista. Este nó consome alguns bytes adicionais de memória, mas sua utilização facilita as operações de inserção e remoção de elementos, uma vez que elimina a necessidade de verificações adicionais para casos especiais. Ademais, ele permanece sempre na posição inicial da lista encadeada, assegurando que a estrutura mantenha sua eficiência.

Cada nó na estrutura deve conter o valor armazenado e, em linguagens que suportam ponteiros, como C++, que é a linguagem adotada neste trabalho, o endereço do próximo nó da sequência. No último nó da lista, deve existir um ponteiro que aponta para um endereço nulo, indicando o término da lista. No código a seguir, apresenta-se uma definição concisa de um nó em C++.

```
struct Node {
 int data;
 Node* next;
};
```

2.1.2. Tipos de Listas Encadeadas

Além da estrutura básica, existem variações que oferecem distintas vantagens, dependendo da aplicação a qual se destinam. Uma dessas variações, que será utilizada neste estudo, é a lista encadeada circular. Nessa modificação, ao invés de o último nó da sequência apontar para um endereço de memória nulo (`nullptr`), o atributo `next` direciona-se para o endereço do primeiro nó da lista, conferindo-lhe a característica de circularidade. Abaixo, a Figura 2 ilustra o comportamento dessa disposição.

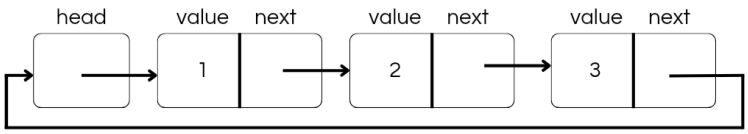

Figura 2. Representação de uma lista encadeada circular

Além desse formato, é possível adicionar um ponteiro adicional na configuração dos nós, com a finalidade de possibilitar o retorno ao nó anterior em tempo constante. Assim, por meio de distintas abordagens para a implementação de uma lista circular, é possível gerar diversas transformações a partir de uma mesma concepção.

2.1.3. Operações básicas

Conforme demonstrado na obra de [Bhargava 2017], nas listas encadeadas, há uma renúncia ao tempo de acesso constante dos *arrays* ($O(1)$) em prol de um acesso que apresenta complexidade linear ($O(n)$), uma vez que é necessário percorrer todos os elementos da lista até se chegar ao elemento desejado. Da mesma forma, a inserção de novos elementos na lista também possui complexidade $O(n)$, dado que é preciso percorrer todos os elementos até o final. Entretanto, existem variações, como as listas circulares duplamente encadeadas, nas quais a inserção pode ocorrer em tempo constante. Em contrapartida, a remoção dos primeiros elementos de uma lista é realizada em $O(1)$, pois o endereço do primeiro elemento é mantido na estrutura.

2.2. Matrizes Esparsas com Lista Encadeada

Neste artigo, trataremos apenas da aplicação de matrizes esparsas com lista encadeada, utilizando-se de nós sentinelas para auxiliar nas operações dentro da matriz.

Nesta configuração, há inicialmente um nó sentinela para cada linha e cada coluna, além de um nó sentinela que conecta ambas as partes, centralizando o início da estrutura. Os nós de cada linha estão interligados de forma circular, conforme explicitado anteriormente, formando uma organização semelhante à representada na Figura 3. Ademais, cada nó deve suportar, além de sua configuração inicial, os valores de linha e coluna da matriz física, mostrado posteriormente.

Assim, ao inserir um elemento de forma semelhante à apresentada na subseção anterior (ver Seção 2.1), é necessário configurar as referências de linha e coluna para manter organizadas as listas encadeadas de linhas e colunas. Na Figura 4, apresenta-se o esquema lógico para a matriz física abaixo.

$$A = \begin{bmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{bmatrix}$$

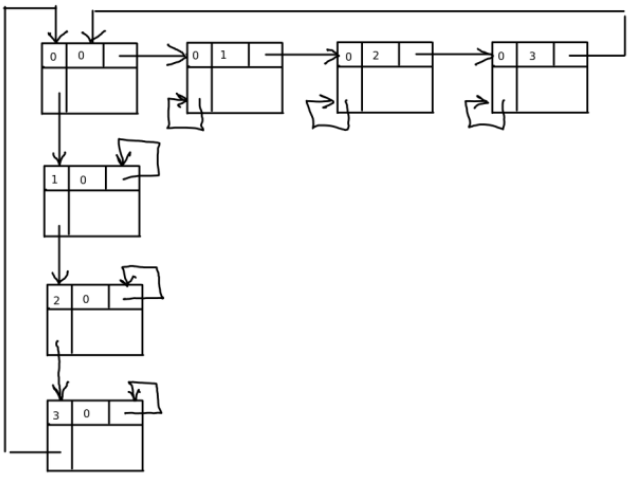

Figura 3. Esquema de uma matriz esparsa vazia

De certa forma, essa disposição pode ser adaptada para diminuir a complexidade das operações, aproximando sua execução de um tempo constante.

2.3. Padrão de Projeto Command

Em um programa computacional, é comum a interação com o usuário por meio de diversas abordagens, como interfaces gráficas ou linhas de comando. Nesse contexto, é necessário que o código seja modular e suficientemente flexível para acomodar alterações futuras. Desse modo, surgem diferentes estratégias para organizar a programação de um software a fim de atender a essas exigências.

No campo da computação, diversos padrões são estabelecidos na literatura [Gamma et al. 1995] e categorizados em várias formas. A presente pesquisa se concentrará no padrão de projeto comportamental denominado *Command*. De maneira geral, esse padrão possibilita a unificação das ações realizadas dentro do sistema em um conjunto exclusivo de componentes.

2.3.1. Estrutura

O primeiro desses componentes é uma classe remetente (também conhecida como invocadora) [Refactoring Guru 2025], responsável por chamar o comando correspondente. Por esse motivo, esse tipo de dado deve conter um conjunto de referências aos comandos necessários para invocar seus métodos. Outro constituinte dessa perspectiva é a definição de uma interface comum a todos os comandos, denominados *comandos concretos*.

É importante salientar que a classe remetente não é responsável por criar um comando; geralmente, ele é pré-criado por meio de um construtor no cliente. No contexto da Programação Orientada a Objetos (POO), um cliente pode ser uma classe externa que chama o remetente. Essa estrutura também é responsável por enviar todos os dados necessários para o funcionamento correto do comando a ser executado. A Figura 5 ilustra a estrutura básica do padrão de projeto estabelecido.

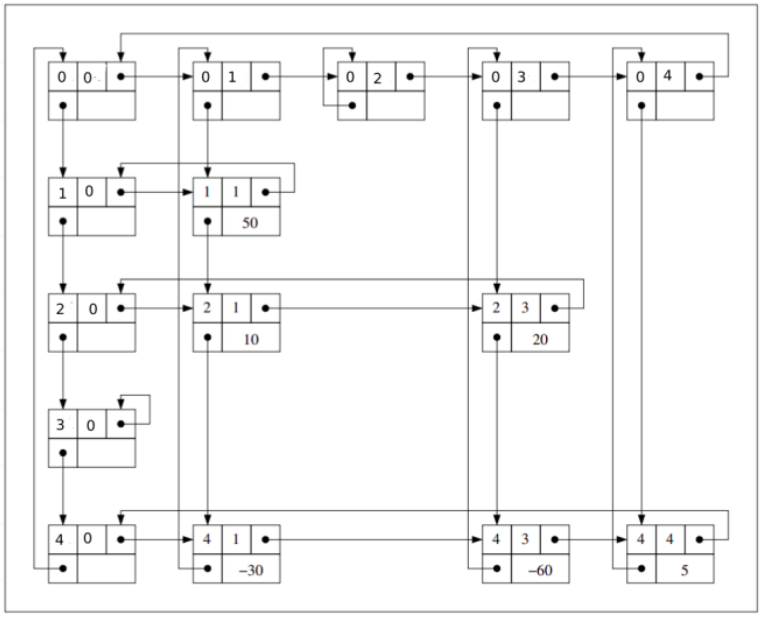

Figura 4. Exemplo de matriz esparsa

2.3.2. Aplicações

Uma vez compreendido o funcionamento desse padrão, é possível descrever suas possíveis aplicações. O padrão Command pode ser aplicado em sistemas que implementam mecânicas de desfazer e refazer ações (*Undo/Redo*), além de facilitar a criação de filas de operações. No contexto deste projeto, tal abordagem contribui para a melhoria da legibilidade e da evolução do código ao longo do tempo.

3. Decisões de Implementação

Com base no conhecimento adquirido durante a revisão bibliográfica (ver Seção 2), a equipe de desenvolvimento decidiu sobre as implementações do projeto. Optou-se por implementar uma classe *Iterator*, com o principal objetivo de simplificar o processo de navegação entre os elementos da matriz e facilitar a comparação entre dois endereços de memória. Além disso, foi adotado o padrão de design *Command*, devido à sua característica de auxiliar na execução do projeto a longo prazo. Por fim, o formato COO serviu como base para o raciocínio lógico por trás do produto matricial.

3.1. Iterator

A classe *Iterator* atua como um intermediário entre a implementação dos métodos da matriz e seus elementos, proporcionando um conjunto de funcionalidades que encapsulam trechos de código repetitivos. Essa abordagem culmina em um código mais organizado e de fácil manutenção. Sua estrutura consiste unicamente no armazenamento do endereço de memória correspondente, permitindo a navegação entre distintos locais dentro da matriz. Dessa forma, o tempo de desenvolvimento da solução apresentada é reduzido.

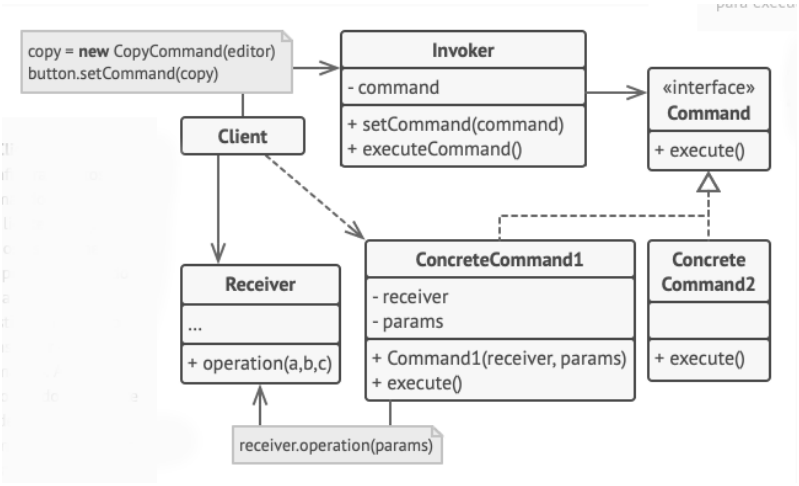

Figura 5. Ilustração da organização do Command

3.2. Command

Como já explicitado na Seção 2.3, esse padrão de projeto oferece uma série de vantagens em termos de organização. Dentre elas, a que consideramos mais significativa é a capacidade de conferir maior flexibilidade ao código. Por meio dessa abordagem, é possível inserir uma camada extra de abstração, mitigando problemas de desenvolvimento, como as inconsistências que podem surgir da falta de atualização de todas as partes interdependentes de uma funcionalidade durante suas modificações. Dessa forma, uma vez estabelecida a base desse padrão, o código é atualizado automaticamente, eliminando o risco de inconsistências.

3.3. Makefile

Segundo [Lambert 2024], Makefiles são utilizados para determinar quais partes de um programa grande precisam ser recompiladas. Na maioria dos casos, essa ferramenta é utilizadas na compilação de projetos escritos em C ou C++. Outras linguagens de programação também possuem suas próprias ferramentas que funcionam de maneira semelhante ao Make.

Em projetos de grande porte, a compilação de todos os arquivos pode levar muito tempo, considerando que existem muitos arquivos a serem lidos e interpretados pelo compilador. Portanto, o uso do Make é justificado neste trabalho, pois ele auxilia no desenvolvimento do software proposto.

3.4. Unordered Map

De acordo com o site [Programiz 2025], a *Standard Library* (STL) `unordered_map` é um contêiner associativo não ordenado que oferece a funcionalidade de uma estrutura de dados de mapa ou dicionário. O `unordered_map` é implementado como uma tabela *hash*, enquanto um mapa tradicional utiliza uma árvore binária, ambas estruturas de dados amplamente conhecidas entre desenvolvedores.

A decisão de se usarar uma tabela *hash* no projeto surgiu da necessidade de localizar rapidamente o comando solicitado pelo usuário, de modo a não comprometer a execução do projeto nem a experiência do usuário.

Em relação à implementação, a tabela *hash* armazena pares de chave-valor: uma *string* que representa o nome do comando e uma estrutura chamada `CommandInfo`. Esta estrutura contém um ponteiro para o comando especificado e uma função que gera uma classe, que denominaremos de Contexto, responsável por reter os dados necessários para a execução do comando, conforme detalhado abaixo:

```
struct CommandInfo {
  Command *command;
  std::function<ContextCommand *()> contextFactory;
};
```

Assim, conseguimos integrar os dados fornecidos pelo usuário ao comando de maneira orgânica, considerando que um contexto pode ser reutilizado em mais de um comando.

4. Divisão do Trabalho

No presente projeto, foi adotado o modelo de desenvolvimento ágil Scrum como base para sua execução. O Scrum [Sabbagh 2013] é um *framework* ágil, simples e leve, utilizado na gestão do desenvolvimento de produtos complexos em ambientes desafiadores. Assim, adaptou-se o modelo de desenvolvimento à realidade dos integrantes da equipe, dividindo o projeto em partes menores na plataforma Trello, um aplicativo de gerenciamento de projetos com interface intuitiva. Na Figura 6 é mostrado uma parte da disposição das tarefas em categorias, denominado quadro.

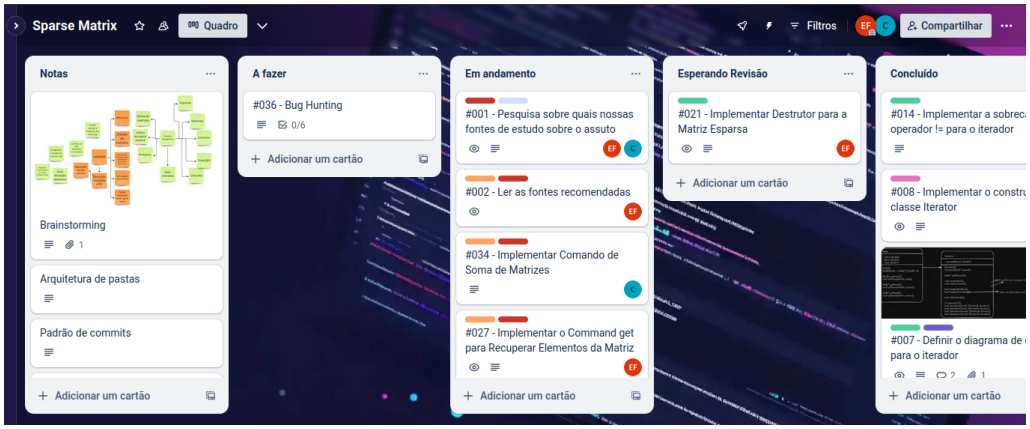

Figura 6. Quadro do Trello utilizado para o gerenciamento de tarefas do projeto

O trabalho, que envolveu tanto a programação do sistema quanto a redação deste artigo, foi subdividido em tarefas menores, como a elaboração da seção de revisão bibliográfica.

Para facilitar a colaboração entre os membros da equipe, foi criado um repositório na plataforma GitHub, mostrado na Figura 7, um serviço baseado em nuvem que oferece um sistema de controle de versão chamado Git. Cada tarefa a ser desenvolvida foi realizada em um ramo do projeto principal, denominado *branch*. Ao final de cada tarefa, um membro da equipe revisava o código produzido, com o objetivo de identificar e minimizar possíveis erros futuros. Após a aprovação, as alterações eram integradas ao ramo principal, garantindo que o desenvolvimento permanecesse organizado e resiliente a falhas.

Isso possibilitou, em caso de problemas ou ajustes necessários, reverter facilmente para um estado anterior, sem comprometer o progresso geral, graças ao controle de versões implementado por meio dos *commits*.

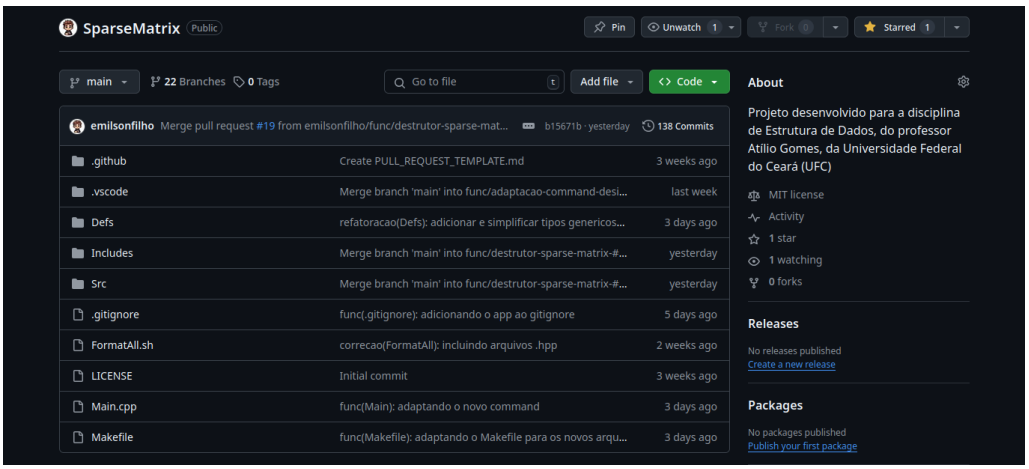

Figura 7. Visão do repositório do projeto no GitHub

5. Dificuldades Encontradas

As principais dificuldades encontradas no desenvolvimento do projeto de matrizes esparsas estão relacionadas com a implementação eficiente das operações de soma e multiplicação, organização do projeto e também ao aprendizado do padrão de projeto Command, que foi adotado como padrão do projeto. A soma de matrizes esparsas exige um tratamento cuidadoso para garantir que somente os elementos não nulos sejam manipulados, buscando ao máximo evitando acessos desnecessários e melhorando o desempenho do nosso projeto. Além disso a estruturação do projeto precisou de uma separação clara entre os módulos responsáveis pela manipulação da matriz, operações matemáticas e entrada e saída de dados, commads, contextos e as exeptions assim garantindo modularidade e reutilização do código. Também a distinção rigorosa entre os arquivos de cabeçalho (.h) e os arquivos de implementação (.cpp) para melhor padronizar todo o projeto. O uso do padrão de projeto Command representou um desafio, exigindo um estudo para sua correta implementação permitindo então a encapsulação de todo codigo, facilitando a manutenção e expansão do sistema para novas operações, comandos e exceções.

6. Testes e Validação

O tratamento de erros no projeto segue um fluxo estruturado que se inicia com a validação dos dados e a verificação da sua integridade antes da continuidade do processo, seja na soma, multiplicação, leitura de matrizes, índices ou falta de logica na entrada do usuário. Somente se os dados estiverem corretos, a execução prossegue normalmente; caso contrário, a exceção (`Exceptions`) correspondente ao erro é lançada para evitar falhas inesperadas e esse mecanismo inclui a geração de `Messages` de erro que informam o usuário sobre a natureza do problema, seja um input incorreto ou a falta de lógica nos parâmetros enviados, permitindo a correção adequada antes de uma nova tentativa e garantindo que erros do usuário não comprometam o funcionamento do programa.

Cada função pode chamar uma função de validação que se certifica de que os dados estão corretos. Caso não estejam, lança a exceção correspondente ao erro com uma mensagem especificada pela função de validação, melhorando a flexibilidade do código. A Figura 8 ilustra esse fluxo.

A implementação desse fluxo exigiu atenção na modularização do código, garantindo que a validação, a manipulação de exceções e a exibição de mensagens fossem separadas de maneira clara e reutilizável, possibilitando a adição de novas exceções e mensagens conforme necessário. O presente modelo de tratamento de erros contribui significativamente para a robustez do sistema assim prevenindo comportamentos inesperados e melhorando a confiabilidade do software.

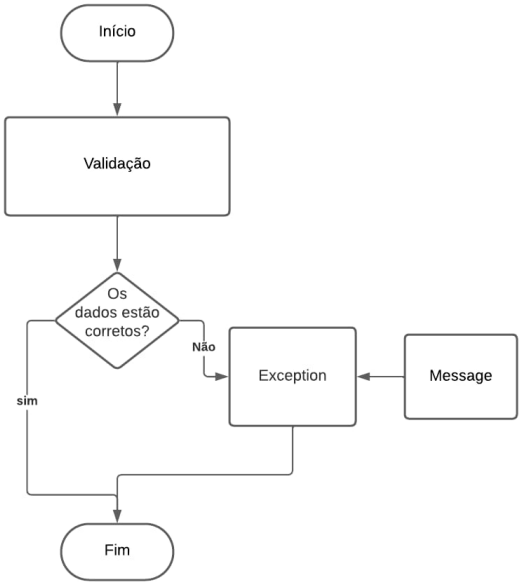

Figura 8. Fluxograma de Tratamento de erros

Ao utilizar `make`, o programa inicia. Digite `help` sempre que precisar de ajuda com algum comando. Para iniciar vamos ler alguma matriz, as disponíveis estão com os nomes m1.txt, m2.txt, m3.txt, m4.txt, m5.txt. Utilize o comando `read` para ler uma matriz, ele recebe o nome da matriz. Não se faz necesário a inserção de nenhum outro elemento além do nome do comando. O programa pedirá o nome do arquivo (não coloque a extensão txt). Ao inserir sua matriz, o programa lhe informará sua posição. Para saber quais matrizes estão no sistema, use `show`. Ao somar ou multiplicar matrizes esparsas, informe a posição da primeira e segunda matriz, que terá como saída a posição do resultado. Ao usar `get`, informe a posição da matriz e a linha e coluna do elemento desejado. Ao utilizar `print`, informe a posição da matriz que será impressa na tela e ela aparecerá na tela. Comandos:

`help` Exibe uma lista de comandos disponíveis.
`show` Mostra todas as matrizes no sistema.
`multiply` Multiplica duas matrizes.
`sum` Soma duas matrizes.
`read` Lê uma matriz determinada por um arquivo.

`get` Exibe um determinado elemento de uma matriz.
`print` Exibe a matriz na tela.
`exit` Fecha a aplicação.

7. Análise de Complexidade

Nesta seção, realizaremos uma análise das complexidades de pior caso das funções responsáveis pela inserção e recuperação de elementos em uma matriz, bem como da função de soma entre duas matrizes. Para fins de referência, denominaremos essas funções como `insert`, `get` e `sum`, respectivamente. Além disso, consideraremos a estrutura de dados empregada no projeto e seu impacto na eficiência de cada operação. Em cada subseção, empregaremos o código respectivo a cada um a fim de melhorar a referência na análise sobre as linhas do código, fornecendo uma prova mais precisa.

7.1. Análise de Complexidade com Múltiplas Variáveis

Antes de iniciarmos as demonstrações, faz-se necessário explicitar como se determina a complexidade de um algoritmo com múltiplas variáveis. Cormen [Cormen et al. 2022], em seu livro, apresenta uma definição de notação assintótica, a qual será utilizada neste trabalho e descrita logo abaixo.

$$f(n,m) \in O(g(n,m)) \tag{1}$$
$$\Leftrightarrow \exists c, n_0, m_0 > 0 \text{ t.q. } 0 \leqslant f(n,m) \leqslant c \cdot g(n,m), \forall n \geqslant n_0 \text{ ou } m \geqslant m_0 \tag{2}$$

7.2. Insert

A função de inserção de um elemento na matriz esparsa adota o seguinte procedimento: inicialmente, são obtidos, por meio de iteradores, os endereços de memória dos nós sentinelas correspondentes à linha e à coluna desejadas. Em seguida, percorre-se os elementos não nulos da matriz até localizar a posição apropriada. Após isso, aloca-se memória para um novo nó, que é inserido na matriz. A seguir, apresenta-se o código elaborado para uma análise matemática mais detalhada.

```
1  void SparseMatrix::InsertMatriz(int row, int col, float value) {
2      ValidationUtils::verifyValidIndexes(row, col, numRows, numCols);
3
4      if (value == 0) return;
5
6      Node *rowSentinela = head->getDown();
7      while (rowSentinela->getRow() != row)
8          rowSentinela = rowSentinela->getDown();
9
10     Node *colSentinela = head->getNext();
11     while (colSentinela->getCol() != col)
12         colSentinela = colSentinela->getNext();
13
14     Node *prevRow = rowSentinela;
15     Node *currentRow = rowSentinela->getNext();
16     while (currentRow != rowSentinela && currentRow->getCol() < col) {
17         prevRow = currentRow;
18         currentRow = currentRow->getNext();
```

```
19      }
20
21      Node *prevCol = colSentinela;
22      Node *currentCol = colSentinela->getDown();
23      while (currentCol != colSentinela && currentCol->getRow() < row) {
24          prevCol = currentCol;
25          currentCol = currentCol->getDown();
26      }
27
28      if (currentRow != rowSentinela && currentRow->getCol() == col) {
29          currentRow->setValue(value);
30          return;
31      }
32
33      Node *novo = new Node(row, col, value, nullptr, nullptr);
34
35      prevRow->setNext(novo);
36      novo->setNext(currentRow);
37
38      prevCol->setDown(novo);
39      novo->setDown(currentCol);
40  }
```

Considere as linhas 2, 4 e 6 como um único tempo de execução, denotado por c_1. O laço presente na linha 7 é executado, no pior caso, n vezes, onde n representa o número de linhas na matriz. De maneira análoga, a linha 11 será executada m vezes. Para a linha 10, consideramos o tempo de execução como c_2, enquanto as linhas 14 e 15 são designadas como c_3, e as linhas 21 e 22 como c_4. O laço contido nas linhas 16 e 23 se repetirá, no pior caso, m e n vezes, respectivamente, uma vez que assumimos como pior cenário uma lista altamente densa, caracterizada por um elevado número de elementos não nulos. Assim, ao tratar as linhas 28 a 39 como um tempo constante c_5, a função $f(n, m)$ que representa o tempo de execução deste algoritmo é:

$$f(n+m) = c_1 + n + c_2 + m + c_3 + m + c_4 + n + c_5$$

De acordo com a definição apresentada por Cormen no início desta seção (ver Seção 7.1), verifica-se que essa função apresenta uma complexidade de pior caso de $O(n+m)$.

7.2.1. Demonstração

Sejam $a := c_1 + c_2 + c_3 + c_4 + c_5$ e $b := 2 + a$ constantes, temos que:

$$c_1 + n + c_2 + m + c_3 + m + c_4 + n + c_5 = n + n + m + m + c_1 + c_2 + c_3 + c_4 + c_5 \tag{3}$$

$$= 2n + 2m + a \tag{4}$$

$$= 2(n + m) + a \tag{5}$$

$$\leqslant 2(n + m) + a(n + m) \tag{6}$$

$$= (2 + a)(n + m) \tag{7}$$

$$= b(n + m), \forall n \geqslant 1, m \geqslant 1 \tag{8}$$

Portante, seja $c = b, n_0 = 1, m_0 = 1, \exists c, n_0, m_0$ tais que $f(n, m) \leqslant c \cdot (n + m), \forall n \geqslant n_0, m \geqslant m_0$, como queríamos demonstrar.

7.3. Get

O método responsável pela recuperação de um elemento de uma matriz segue um procedimento simples. Primeiro, valida-se se os índices da linha e da coluna são corretos. Após a validação, percorrem-se os nós sentinelas até chegar à linha desejada. Em seguida, busca-se o nó que contém a coluna correspondente. Se não houver um elemento na coluna especificada, retorna-se o valor padrão zero. Abaixo, está o código escrito para melhor compreensão, removendo-se os comentários do código-fonte original.

```
1  double SparseMatrix::getElement(int row, int col) const {
2    ValidationUtils::
3      verifyValidIndexes(row, col, numRows, numCols);
4
5    Iterator it(head);
6
7    while (it.getPointer()->getRow() != row) it.nextInCol();
8
9    while (it.getPointer()->getCol() < col) {
10     if (it.getPointer()->getNext()->getCol() == 0)
11         return 0;
12     it.nextInRow();
13   }
14
15   if (it.getPointer()->getCol() > col) return 0;
16
17   return *it;
18 }
```

A função `verifyValidIndexes` foi implementada com complexidade de tempo constante ($O(1)$) e tem como única responsabilidade a validação dos índices fornecidos.

Considerando que as linhas 2, 3 e 5 são executadas em um tempo constante c_1, é possível analisar a complexidade das linhas subsequentes. O laço presente na linha 7 se repetirá, no pior caso, n vezes, onde n representa o número de linhas da matriz. De maneira análoga, o laço na linha 9 se repetirá m vezes, onde m corresponde ao número de colunas da matriz.

Assumindo que as linhas 10, 11 e 12 são executadas em um tempo constante c_2, o número total de execuções desse trecho de código será de $m \cdot c_2$, uma vez que está inserido no laço de repetição.

Analogamente, definimos c_3 como uma constante que representa o tempo de execução das operações realizadas nas linhas 15 e 17.

Dessa forma, podemos expressar a função que representa o tempo de execução do método `get` da seguinte maneira:

$$f(n, m) = c_1 + n + m \cdot c_2 + c_3$$

De acordo com a definição apresentada por Cormen no início desta seção (ver Seção 7.1), verifica-se que essa função apresenta uma complexidade de pior caso de $O(n + m)$.

7.3.1. Demonstração

Seja $a := c_1 + c_3$ e $b := 1 + c_2 + a$, percebe-se que:

$$\begin{aligned} c_1 + n + m \cdot c_2 + c_3 &= n + m \cdot c_2 + (c_1 + c_3) && (9)\\ &\leqslant (n + m) + (n + m) \cdot c_2 + (n + m)(c_1 + c_3) && (10)\\ &= (n + m)(1 + c_2 + a) && (11)\\ &= b(n + m), \forall n \geqslant 1, m \geqslant 1 && (12) \end{aligned}$$

Portanto, seja $c = b, n_0 = 1, m_0 = 1$, $\exists c, n_0, m_0$ tais que $f(n, m) \leqslant c \cdot (n + m), \forall n \geqslant n_0, m \geqslant m_0$, como queríamos demonstrar.

7.4. Sum

A função de somar duas matrizes esparsas percorre simultaneamente ambas as matrizes, realizando uma série de verificações para assegurar a eficiência do processo. Quando a coluna do nó na primeira matriz é inferior à da segunda, isso indica a ausência de um elemento correspondente na segunda matriz, permitindo que o elemento da primeira matriz seja adicionado diretamente à matriz resultante. Por outro lado, se a coluna for superior, torna-se necessário percorrer os elementos da linha da segunda matriz até que uma condição diferente seja encontrada. Quando as colunas são idênticas, a matriz resultante recebe a soma dos valores correspondentes. Por fim, verifica-se se os nós em comparação são distintos dos nós sentinelas. Abaixo, apresenta-se uma parte do código para promover uma melhor compreensão.

```
1 SparseMatrix *SumMatrix(SparseMatrix *matrixA, SparseMatrix *matrixB) {
2   ValidationUtils::verifyMatricesAreSummable(
3       matrixA->getNumRows(), matrixA->getNumCols(), matrixB->getNumRows
          (),
4       matrixB->getNumCols());
5
6   Node *currentA = matrixA->getHead()->getDown()->getNext();
```

```
7   Node *currentB = matrixB->getHead()->getDown()->getNext();
8   SparseMatrix *result =
9       new SparseMatrix(matrixA->getNumRows(), matrixB->getNumCols());
10
11  while (currentA->getRow() != 0 && currentB->getRow() != 0) {
12    bool i = currentA->getCol() == 0, j = currentB->getCol() == 0;
13    while (!(i && j)) {
14      if (i == true) {
15        result->InsertMatriz(currentB->getRow(), currentB->getCol(),
16                             currentB->getValue());
17        currentB = currentB->getNext();
18      } else if (j == true) {
19        result->InsertMatriz(currentA->getRow(), currentA->getCol(),
20                             currentA->getValue());
21        currentA = currentA->getNext();
22      } else if (currentA->getCol() == currentB->getCol()) {
23
24        result->InsertMatriz(currentA->getRow(), currentA->getCol(),
25                             currentA->getValue() + currentB->getValue
                                 ());
26        currentA = currentA->getNext();
27        currentB = currentB->getNext();
28      } else if (currentA->getCol() > currentB->getCol()) {
29
30        result->InsertMatriz(currentB->getRow(), currentB->getCol(),
31                             currentB->getValue());
32        currentB = currentB->getNext();
33      } else {
34        result->InsertMatriz(currentA->getRow(), currentA->getCol(),
35                             currentA->getValue());
36        currentA = currentA->getNext();
37      }
38
39      if (currentA->getCol() == 0)
40        i = true;
41      if (currentB->getCol() == 0)
42        j = true;
43    }
44
45    currentA = currentA->getDown()->getNext();
46    currentB = currentB->getDown()->getNext();
47  }
48
49  return result;
50 }
```

A função `verifyMatricesAreSummable` foi implementada com complexidade $O(1)$, tendo como única finalidade a validação das matrizes fornecidas.

Consideremos o tempo de execução das linhas 6 e 7 como uma constante única c_1. O laço na linha 11 será executado n vezes, onde n representa a quantidade de linhas das matrizes. O tempo de operação da linha 12 será considerado como uma constante c_2. O laço interno na linha 13 se repetirá m vezes, sendo m a quantidade de colunas a serem percorridas. Dentro desse laço interno, diversas validações ocorrem, mas todas invocam a mesma função (`InsertMatriz`), responsável por inserir um elemento na

matriz, cuja complexidade foi demonstrada na seção 7.2, sendo $O(n + m)$. As operações dentro dos condicionais são tratadas como uma constante c_3. As linhas 39 a 42 serão abordadas como uma constante c_4. As linhas 45 e 46 também serão consideradas como a constante c_5, e o retorno da função será representado por c_6. Assim, a função que descreve o comportamento de execução dessa rotina pode ser expressa pela definição a seguir:

$$f(n, m) = 1 + c_1 + n\left[c_2 + m\left(c_3 + n + m + c_4\right) + c_5\right] + c_6$$

Seguiremos a mesma base de demonstração das provas anteriores. De acordo com a definição dada na Seção 7.1, percebe-se que essa função tem complexidade $O(nm(n + m))$.

7.4.1. Demonstração

Seja $a := 1 + c_1 + c_2 + c_3 + c_4 + c_5 + c_6$ e $b := a + 1$, percebe-se que:

$$\begin{aligned}
&1 + c_1 + n\left[c_2 + m\left(c_3 + n + m + c_4\right) + c_5\right] + c_6 && (13)\\
&= nc_2 + nm\left(c_3 + n + m + c_4\right) + nc_5 + 1 + c_1 + c_6 && (14)\\
&= n\left(mc_3 + mn + mm + mc_4\right) + nc_2 + nc_5 + 1 + c_1 + c_6 && (15)\\
&= nmc_3 + nmn + nmm + nmc_4 + nc_2 + nc_5 + 1 + c_1 + c_6 && (16)\\
&\leqslant nmc_3 + nmn + nmm + nmc_4 + nmc_2 + nmc_5 + nm + nmc_1 + nmc_6 && (17)\\
&= nm\left(c_3 + c_4 + c_2 + c_5 + 1 + c_1 + c_6\right) + nmn + nmm && (18)\\
&= nma + nmn + nmm && (19)\\
&= nm\left(a\right) + nm\left(n + m\right) && (20)\\
&\leqslant nm\left(n + m\right)\left(a\right) + nm\left(n + m\right) && (21)\\
&= nm\left(n + m\right)\left(a + 1\right) && (22)\\
&= b \cdot nm(n + m), \forall n, m \geqslant 1 && (23)
\end{aligned}$$

Portanto, seja $c = b$, $n_0 = 1$, $m_0 = 1$, $\exists c, n, m$ tais que $f(n, m) \leqslant c \cdot nm(n + m), \forall n \geqslant n_0, m \geqslant m_0$, como queríamos demonstrar.

8. Conclusão

Neste trabalho, exploramos a implementação de uma nova estrutura de dados, utilizando os conhecimentos adquiridos em sala de aula e incorporando novos conceitos por meio da literatura. As decisões tomadas foram orientadas para otimizar o funcionamento do projeto como um todo, priorizando flexibilidade e segurança.

Ao longo do desenvolvimento, compreendemos a importância das estruturas de dados no armazenamento e nas operações dos dados fornecidos, além de percebermos como a otimização de algoritmos em computação pode influenciar essa dinâmica, identificando as vantagens e desvantagens de cada estrutura em seus respectivos contextos. No entanto, algumas limitações surgiram, exigindo uma análise mais aprofundada sobre a

abordagem mais adequada para o problema em questão, incluindo o uso de outras estruturas de dados que ainda não haviam sido discutidas em sala de aula. Também aprendemos sobre o funcionamento de um projeto em colaboração com mais de uma pessoa, ampliando a perspectiva dos membros envolvidos para além da simples ideia de um material a ser construído.

Em suma, este projeto contribuiu para uma compreensão mais aprofundada das técnicas de manipulação de matrizes esparsas e do impacto que a escolha de uma estrutura de dados apropriada pode exercer na eficiência computacional.

Referências

Bhargava, A. Y. (2017). *Entendendo algoritmos: um guia ilustrado para programadores e outros curiosos*. Novatec, 1st edition. p. 46-49.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). *Introduction to Algorithms*. The MIT Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Pattens: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Disponível em: http://www.javier8a.com/itc/bd1/articulo.pdf. Acesso em: 17 jan. 2025. p.233-242.

Lambert, C. (2024). Makefile tutorial by example. Disponível em: https://www.makefiletutorial.com. Acesso em: 19 jan. 2025.

Pissanetzki, S. (1984). *Sparse Matrix Technology*. Academic Press, online edition. Disponível em: https://encurtador.com.br/jb5rn. Acesso em: 17 jan. 2025. p.1-34.

Programiz (©2014-2025). C++ unordered map. Disponível em: https://www.programiz.com/cpp-programming/unordered-map. Acesso em: 22 jan 2025.

Refactoring Guru (©2014-2025). Command. Disponível em: https://refactoring.guru/pt-br/design-patterns/command. Acesso em: 19 jan. 2025.

Sabbagh, R. (2013). *Scrum: gestão ágil para produtos de sucesso*. Casa do Código.

Schildt, H. (1997). *C Completo e total*. Makkron Books, 3th edition. Disponível em: https://www.inf.ufpr.br/lesoliveira/download/c-completo-total.pdf. Acesso em 16 jan. 2025. p. 566-571.