Relatório do projeto SparseMatrix

Ana Beatriz Martins Santiago[1]

[1]Aluna da Universidade Federal do Ceará (UFC) - Campus Quixadá
Quixadá – CE – Brazil

anabsantiago0@gmail.com

Abstract. *This report describes the SparseMatrix data structure, which is based on the concept of sparse matrices. It discusses the very definition of sparse matrices, in order to better explain their application in the implemented data structure. It also discusses the process of its construction, delving deeper into the different decisions made, as well as the difficulties encountered, in addition to also reporting which tests were carried out and their respective details. Furthermore, this report presents the worst-case complexity analysis of three specific functions, which would be: insert, get and sum.*

Resumo. *O presente relatório descreve a estrutura de dados SparseMatrix, que baseia-se no conceito de matrizes esparsas. Descorre-se a respeito da própria definição de matrizes esparsas, afim de melhor explicar sua aplicação na estrutura de dados implementada. Além disso, aborda-se sobre o processo de sua construção, aprofundando-se nas diferentes decisões tomadas, bem como as dificuldades encontradas, além de também relatar quais testes foram executados e seus respectivos detalhes. Ademais, este relatório apresenta a análise de complexidade de pior caso de três funções específicas, que seriam: insert, get e sum.*

1. Introdução

As matrizes esparsas são por definição matrizes nas quais a maioria de suas posições são preenchidas por zero. Por essa razão, matrizes esparsas possuêm potencial em economia de memória, uma vez que só será necessário armazenar os valores diferentes de zero na matriz. Por essa razão, elas são amplamente utilizadas em cenários e contextos mais especializados, como por exemplo, na área de aprendizagem de máquinas e de estatísticas. Dado isso, foi implementado o Tipo Abstrato de Dado (TAD) SparseMatrix, do qual, seguindo mais adiante, há uma descrição detalhada de suas funções e processo de criação.

2. Sparse Matrix

Esta seção aborda detalhes sobre a implementação de SparseMatrix, detalhando cada uma de suas funções, e detalha logo em seguida as principais dificuldades acometidas durante o desenvolvimento do projeto. O TAD foi implementado utilizando-se dos conceitos de Programação Orientada a Objetos (POO), do qual foi criada a classe SparseMatrix, que utiliza (importa) o struct Node e possui os seguintes atributos privados: um ponteiro para Node nomeado m_head, um inteiro para a quantidade de linhas entitulado m_linhas e um inteiro para a quantidade de colunas (m_colunas).

Sobre o struct Node: cada Node é uma célula (nó) da matriz, possuindo um valor double, um ponteiro para o Node à direita na matriz, um ponteiro para o Node abaixo na

matriz e dois inteiros para indicar a linha e a coluna em que o Node se encontra na matriz. Dito isso, ressalta-se que SparseMatrix utiliza-se de uma lista simplesmente encadeada circular, com nós sentinelas, para suas linhas e uma lista simplesmente encadeada circular, com nós sentinelas, para suas colunas, sendo m_head um ponteiro para o primeiro sentinela da matriz (0, 0).

Logo, por armazenar apenas valores diferentes de zero, cada ponteiro para Node à direita na matriz busca o próximo Node com valor diferente de zero na mesma linha, caso não haja tal nó, ele apontará para o sentinela de sua linha, de tal forma que o último nó da linha sempre apontará para o sentinela de sua linha. O mesmo serve para as colunas: cada ponteiro para Node abaixo busca o próximo nó cujo valor é diferente de zero em sua coluna, e caso não o houver, ele apontará para o sentinela de sua coluna, portanto, o último nó da coluna sempre apontará para o sentinela da coluna. A seguir, há uma representação visual dessa estrutura de nós sentinelas, sendo uma matriz de 2x2 vazia, onde os nós sentinelas ocupam toda a linha 0 e toda a coluna 0:

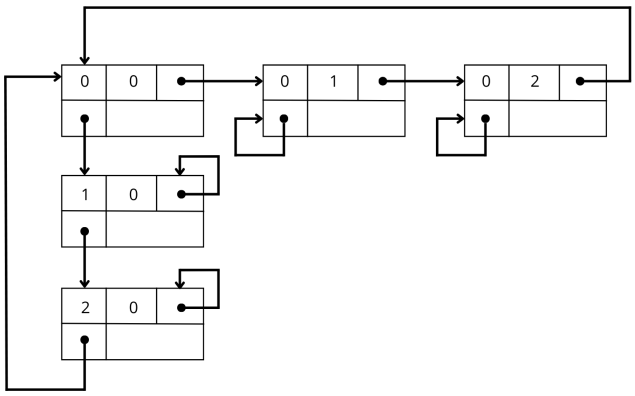

Figura 1. Matriz esparsa vazia

2.1. Funções de SparseMatrix

Segue uma descrição das funções implementadas na classe SparseMatrix, destacando o motivo da adição de algumas funções que não foram planejadas inicialmente:

Construtor default: Construtor público, cria uma matriz esparsa vazia, com apenas um nó sentinela (m_head) apontando para si mesmo tanto pela direita como por baixo. Esse construtor foi adicionado em vista de facilitar manipulações no vector de SparseMatrix da main do projeto.

Construtor da classe: Construtor público, recebe duas referências constantes inteiras, m e n, e cria uma matriz esparsa com m linhas e n colunas. Após atribuir os valores m e n à m_linhas e m_colunas, respectivamente, chama a função read para criar a estrutura de nós sentinelas. Caso m ou n não seja maior que zero, uma exceção é lançada.

Construtor de cópia: Construtor público, recebe uma SparseMatriz como argumento e cria uma cópia dela. Esse construtor foi adicionado em vista da criação do vector de SparseMatrix no arquivo main do projeto e sua manipulação.

Destrutor: Libera a memória alocada dinamicamente (ponteiros para Node).

Função read: Não recebendo argumentos como parâmetros, essa função privada de retorno é void é responsável por criar a estrutura de nós sentinelas detalhada na figura 1. Foi criada para facilitar a criação da matriz esparsa vazia e é utilizada no construtor da classe (como dito anteriormente), no construtor de cópia e na sobrecarga do operador de atribuição.

Função getRef: Recebe duas referências constantes inteiras como argumento, i e j, e retorna uma referência double para o valor na linha i e coluna j da matriz. O modificador de acesso da função é private, pois a mesma só é utilizada para atualizar um valor na matriz, a partir da função set. Ademais, a função verifica se os valores passados por parâmetro são válidos, caso contrário, uma exceção é lançada. Essa função foi adicionada para facilitar a alteração de um valor de um nó pela função set.

Função set: Recebe uma referência constante double value e duas referências constantes inteiras m e n e atualiza o valor da célula i, j da matriz com value. Além disso, chama a função getRef para obter acesso a uma referência do valor da célula em questão. Essa função é privada pois é acessada a partir da função insert, e foi adicionada considerando a situação em que o usuário insere um valor diferente de zero em uma célula que já possui um valor diferente de zero.

Função empty: Pública e constante, retorna true se e somente se a matriz estiver vazia (nenhum valor diferente de 0 inserido), e false caso contrário. A função foi adicionada em prol de facilitar tal verificação quanto ao estado da matriz (vazia ou não), e é utilizada pela função clear.

Função clear: Pública e de retorno void, a função foi adicionada para, caso a matriz não esteja vazia (!empty()), apagar os elementos da mesma (mantendo o número de linhas e de colunas, ela apenas "zera" todos os elementos da matriz). A função executa nada caso contrário.

Função get: Recebe como parâmetro duas referências constantes inteiras i e j, e retorna uma referência constante double para o valor na linha i, coluna j da matriz. Essa função é pública e lança uma exceção caso os valores passados por parâmetros sejam inválidos (menores que zero ou maiores que o número de linhas/colunas da matriz).

Função getLinhas: Função constante pública, retorna um inteiro para a quantidade de linhas da matriz (m_linhas). Foi adicionada para facilitar a manipulação de matrizes em funções externas, por exemplo nas funções sum e multiply, que pertencem à main do projeto.

Função getCol: Função constante pública, retorna um inteiro para a quantidade de colunas da matriz (m_colunas). Assim como getLinhas, foi criada cosniderando facilitar a manipulação das matrizes em outras funções, principalmente na main (funções sum e multiply).

Função insert: Pública, recebe uma referência constante double value e duas referências constantes inteiras m e n. Se value for igual a zero e a célula na linha m, coluna n, não possuir valor, a chamada da função não terá efeito algum. Se value for igual a zero e houver um valor na célula em questão, a função apagará o nó nessa posição. Entretanto para value diferente de zero, se houver um valor no nó m, n, a função chama set(value, m, n), para atualizar o valor do nó, caso contrário a função insere value na célula m, n. A

função retorna void, porém se m ou n forem posições inválidas na matriz, uma exceção será lançada.

Função print: Função constante void pública que imprime os elementos da matriz no terminal.

Sobrecarga do operador de atribuição: Pública, retorna uma referência para uma SparseMatrix. Recebe como parâmetro uma referência constante para uma SparseMatrix "a", libera os nós alocados dinamicamente da matriz que receberá a cópia (incluindo os sentinelas) e copia os dados de "a".

2.2. Funções da main

Logo a seguir, descreve-se funções implementadas na main do projeto, detalhando razões para a adição de uma função que não foi planejada inicialmente:

Função readSparseMatrix: Retorno void, a função recebe uma referência para uma matriz esparsa vazia, m, e uma string file. Se file for o nome de um arquivo .txt válido, a função tentará criar uma matriz lendo os dados de file. Caso o nome do arquivo seja inválido, uma exceção será lançada

Função sum: Recebe duas matrizes esparsas como parâmetro e retorna uma matriz esparsa que é a soma das duas. Caso as dimensões das matrizes sejam diferentes, uma exceção será lançada pois não será possível realizar a soma das matrizes.

Função multiply: Recebe duas matrizes esparsas como parâmetro e retorna uma matriz esparsa que é a multiplicação das duas. Caso não seja possível realizar a multiplicação, uma exceção será lançada alegando que as dimensões das matrizes não são ideais.

Função matrizValida: Recebe uma referência para um vector de SparseMatrix vet, e uma referência constante inteira k, e retorna true se e somente se k for um indíce válido de vet, e retorna false caso constrário. Essa função foi adicionada em vista de facilitar saber se um índice passado pelo usuário é válido no vector de SparseMatrix utilizado na main.

2.3. Principais dificuldades

É destacável que diversas dificuldades foram encontradas na criação de SparseMatrix, portanto, descorre-se aqui a respeito das principais dificuldades acometidas. Primeiramente, ressalta-se que começar o projeto em si foi dificultoso, pois escolher uma das ideias e dar seguimento com a mesma até o fim não foi só difícil, como impossível: A primeira ideia foi desenvolver outra classe, chamada ForwardList, para que SparseMatrix manipulasse ForwardList's (linhas e colunas). Entretanto, houveram dificuldades na implementação dessa ideia, portanto o projeto foi reiniciado com apenas a classe SparseMatrix. Destaca-se que muito provavelmente o projeto deveria ter sido começado dessa forma considerando a vontade do solicitador do projeto.

Outro ponto a ser destacado é a respeito da função insert, que apresentou erros até as fases finais do projeto. Seja por considerar os diversos casos de inserção (valor igual a zero, valor igual a zero em uma célula já preenchida, valor diferente de zero em uma célula vazia, valor diferente de zero em uma célula já preenchida, inserir em uma linha

que já possui nó antes ou depois da coluna solicitada, inserir em uma coluna que já possui nó antes ou depois da linha solicitada...) como pelas diferentes lógicas aplicadas a cada vez que a função não inseria corretamente, onde era necessário parar as atividades atuais e pendentes apenas para entender onde o erro estava dentre cada caso, e muitas vezes apagar boa parte dessa função.

Ademais, algumas dificuldades puderam ser vistam em criar o construtor de cópia e a sobrecarga do operador de atribuição, bem como no destrutor, que deveria liberar toda a memória alocada dinamicamente. Além desses, houveram alguns problemas na elaboração do presente relatório, principalmente no que diz respeito à utilização do Latex. Por fim houveram algumas dúvidas sobre se a maneira como as funções e a classe foram escritas agradará o solicitante.

3. Testes executados

Durante o desenvolvimento do projeto, vários testes foram executados para identificar, analisar e corrigir erros, além de melhorar possíveis pendências. Os testes foram realizados conforme as funcionalidades foram sendo implementadas, visando testar caso a caso de cada função implementada. A seguir lista-se os principais testes realizados para cada funcionalidade implementada e os comandos disponíveis para o usuário manipular as matrizes do vector do arquivo main:

3.1. Testes de criação

Lista de testes de criação:

Criar matriz esparsa vazia, com m linhas e n colunas.

Criar matriz esparsa com 0 linhas e 0 colunas (construtor default ou lançamento de exceção).

Criar matriz esparsa a partir de outra matriz esparsa (Construtor de cópia e operador de atribuição).

Criar matriz esparsa a partir da leitura de um arquivo.

Criar matriz esparsa a partir da soma de duas matrizes.

Criar matriz esparsa a partir da multiplicação de duas matrizes.

3.2. Testes de inserção

Lista de testes de inserção:

Inserir valor zero em célula vazia.

Inserir valor zero em célula já preenchida.

Inserir valor diferente de zero em célula já preenchida.

Inserir valor diferente de zero em célula não preenchida cuja linha e coluna são vazias.

Inserir valor diferente de zero em célula não preenchida onde apenas a linha é vazia.

Inserir valor diferente de zero em célula não preenchida onde apenas a coluna é vazia.

Inserir valor diferente de zero em célula não preenchida onde a linha e a coluna possuem nós alocados (antes e/ou depois da célula solicitada).

3.3. Demais testes

Lista dos demais testes:

Acessar valor em célula válida da matriz.

Acessar valor em célula inválida da matriz.

Acessar quantidade de linhas da matriz.

Acessar quantidade de colunas da matriz.

Imprimir matriz na tela.

Somar duas matrizes.

Multiplicar duas matrizes.

Verificar se a matriz está vazia.

Limpar a matriz.

Destruir matriz (desalocando os nós).

Verificar se um índice é válido no vector de matrizes da main.

3.4. Comandos disponíveis para o usuário

Lista-se a seguir os comandos disponíveis para o usuário manipular as matrizes do vector presente no arquivo main, detalhando cada um.

Comando “create m n”: Cria uma matriz esparsa com m linhas e n colunas.

Comando “createWith sum i_matrix j_matrix”: Cria uma matriz esparsa a partir da soma de duas matrizes do vector.

Comando “createWith multiply i_matrix j_matrix”: Cria uma matriz esparsa a partir da multiplicação de duas matrizes do vector.

Comando “createWith file nameFile”: Cria uma matriz esparsa a partir da matriz lida no arquivo .txt “nameFile”. Mais a frente será descorrido detalhes sobre esse comando.

Comando “createWith i_matrix”: Cria uma cópia da matriz i do vector.

Comando “print”: Imprime todas as matrizes do vector no terminal.

Comando “print i_matrix’: Imprime a matriz i do vector no terminal.

Comando “insert i_matrix value m n”: Insere value na matriz i do vector, na linha m, coluna n.

Comando “get i_matrix m n”: Imprime o valor na célula da linha m, coluna n da matriz i do vector no terminal.

Comando “sum i_matrix j_matrix”: Imprime a soma de duas matrizes do vector no terminal.

Comando “multiply i_matrix j_matrix”: Imprime a multiplicação de duas matrizes do vector no terminal.

Comando “clear i_matrix”: Limpa, zerando todas as posições, da matriz i do vector.

Comando “clearAll”: Limpa, zerando todas as posições, de todas as matrizes do vector.

Comando “delete i_matrix”: Deleta a matriz i do vector.

Comando “deleteAll”: Deleta todas as matrizes do vector.

Comando “exit”: Encerra o programa.

Comando “help”: Imprime todos os comandos disponíveis no terminal.

Uma observação sobre o comando “createWith file nameFile”: há uma pasta no diretório do projeto nomeada “matrizes”, onde se encontram os seguintes arquivos .txt: m1, m2, m3, m4, m5, mA, mB. Para usar o comando, o usuário deverá digitar o nome do arquivo sem aspas, sem extensão e sem mencionar a pasta. O usuário pode criar o próprio arquivo .txt para uma matriz e adicioná-lo na pasta “matrizes” para ser lido, nesse caso, o arquivo precisará seguir o seguinte padrão para que a matriz seja criada com sucesso: A primeira linha apresentando o número de linhas e o número de colunas da matriz, respectivamente e separados por espaço. As linhas seguintes apresentarão, nessa ordem e separados por espaços, um indíce para linha, um indíce para coluna e um valor double a ser inserido na matriz na linha e coluna correspondente.

4. Análises de Complexidade de Pior Caso

Essa seção aborda as análises de complexidade de pior caso de três funções: insert, get e sum. A fim de melhor explicar a linha de raciocínio e a prova das análises, foram anexadas imagens da tela do código destas funções.

4.1. Função get

```
// Complexidade O(n)
const double& get(const int& i, const int& j) const{
    if(i<=0 || j<=0 || i>m_linhas || j>m_colunas) throw std::invalid_argument("Argumento Invalido"); // c0
    Node *aux1 = m_head; // c1
    Node *aux2; // c2
    for(int m=0;m<i;m++) aux1 = aux1->abaixo; // c3*m iterações
    aux2 = aux1->direita; // c4
    while(aux2->col != j) { // c5*n iterações
        aux2 = aux2->direita; // c6
        if(aux2 == aux1) return aux2->valor; // c7
    }
    return aux2->valor; // c8
```

Figura 2. Função get

Acompanhando linha a linha da função get, chegamos à expressão: $c0+c1+c2+c3*m+c4+c5*n+c6+c7+c8$. Para $a = c0+c1+c2$, $b = c4+c6+c7+c8$, a expressão fica: $a+b+c3*m+c5*n$. Para entradas n, m suficientemente grandes, podemos concluir que $a, b, c3$ e $c5$ tornam-se insignificantes, uma vez que são constantes. Logo, a expressão

final fica: $m + n$. Portanto o pior caso de get ocorre quando os valores i e j passados por parâmetro são válidos (o que contorna o lançamento de exeção), quando a linha i é a última da matriz, e a coluna j também é a última da matriz. Logo, a complexidade de pior caso da função é $O(m + n)$, que é linear. Para fins de simplificação, a complexidade de get será referenciada como $O(n)$ a seguir.

4.2. Função insert

A priori, vale lembrar aqui que value, m e n são valores passados por parâmetros para a função, sendo value um double para ser adicionado na linha m, coluna n da matriz. Com isso, segue a descrição dos casos mais importantes observados:

Caso 1, complexidade $O(n)$: Ocorre quando o value é igual a zero, do qual ocorrerá a verificação se a célula na posição (m, n) já possui um valor diferente de 0. Para essa verificação, a função get é chamada (complexidade $O(n)$). Se get retornar um valor diferente de zero, um for e um while serão executados para se chegar à celula em questão a fim de apagá-la. Logo, esse caso executa no pior caso quatro loops separados, equivalendo a uma complexidade de $O(n)$.

```
if(value == 0) { // Caso 1 --- O(n)
    if(get(m, n) != 0){ // se houver um valor na celula (i, j) --- O(n)
        Node *aux1 = m_head;
        Node *aux2;
        for(int i=0;i<m;i++) aux1 = aux1->abaixo; // n iterações
        while(aux1->direita->col != n) aux1 = aux1->direita; // n iterações
        aux2 = aux1->direita;
        aux1->direita = aux2->direita;
        delete aux2; // deleta a celula
    }
    return;
}
```

Figura 3. Caso 1

Caso 2, complexidade $O(n)$: Ocorre quando value é diferente de zero e há um valor na célula solicitada (m, n) (verificação através de get, $O(n)$). Esse caso então chama a função set para alterar o valor da célula, que chama a função getRef. Essa última função possui complexidade $O(n)$, uma vez que a mesma executa 2 loops separados. Logo esse caso também executa quatro loops separados (dois de getRef e dois de get), equivalendo a uma complexidade linear, ou seja de $O(n)$.

```
if(get(m, n) != 0) { // se houver um valor na calula (i, j) -- Caso 2 O(n)
    set(value, m, n); // atualiza o valor da celula --- O(n)
    return;
}
```

Figura 4. Caso 2

```
// Função que recebe dois inteiros, i e j
// e retorna uma referencia para o valor double
// da linha i, coluna j da matriz
// Complexidade O(n)
double& getRef(const int& i, const int& j){
    if(i<=0 || j<=0 || i>m_linhas || j>m_colunas) throw std::invalid_argument("Argumento Invalido");
    Node *aux1 = m_head;
    Node *aux2;
    for(int m=0;m<i;m++) aux1 = aux1->abaixo; // n iterações
    aux2 = aux1->direita;
    while(aux2->col != j) { // n iterações
        aux2 = aux2->direita;
        if(aux2 == aux1) return aux1->valor;
    }
    return aux2->valor;
}

// Função que recebe uma referência double value e dois int m, n
// e atualiza o valor na celula (m, n) para value
// Complexidade O(n)
void set(const double& value, const int& m, const int& n){
    double& novo = getRef(m, n); // O(n)
    novo = value;
}
```

Figura 5. Função getRef e set, respectivamente

Caso 3 complexidade $O(n)$: Ocorre quando: value é diferente de zero, quando a célula na linha m, coluna n não possui valor válido (verificação através de get, $O(n)$), quando a linha m possui todas as posições antes da coluna n com nós alocados, e quando a coluna n possui todas as posições antes da linha m com nós alocados. Dessa forma, esse caso executa seis loops separados, sendo dois da função get, dois para navegar até a linha m, coluna n da matriz, e mais dois para percorrer os nós da linha m que estão antes da coluna n, e os nós da coluna n que estão antes da linha m. A complexidade final desse caso é $O(n)$.

```
if(get(m, n) != 0) { // se houver um valor na celula (i, j) -- Caso 2 O(n)
    set(value, m, n); // atualiza o valor da celula --- O(n)
    return;
}
// Caso 3 --- O(n)
Node *linhaAtual = m_head;
Node *colAtual = m_head;
Node *auxLinha, *auxCol;
for(int i=0;i<m;i++) linhaAtual = linhaAtual->abaixo; // n iterações
for(int i=0;i<n;i++) colAtual = colAtual->direita; // n iterações
auxLinha = linhaAtual;
auxCol = colAtual;
if(linhaAtual->direita != linhaAtual){ // se houver uma celula na linha
    while(auxLinha->direita->col < n && auxLinha->direita != linhaAtual){ // n iterações
        auxLinha = auxLinha->direita; // navega ate a celula que vem antes da nova celula
    }
}
if(colAtual->abaixo != colAtual){ // se houver uma celula na coluna
    while(auxCol->abaixo->linha < m && auxCol->abaixo != colAtual){ // n iterações
        auxCol = auxCol->abaixo; // navega ate a celula que vem antes da nova celula (i, j
    }
}
Node *novo = new Node(value, m, n, auxCol->abaixo, auxLinha->direita);
auxLinha->direita = novo;
auxCol->abaixo = novo;
```

Figura 6. Caso 3

Conclusão: Portanto, o pior caso, a função insert possui complexidade $O(n)$.

4.3. Função sum

Observando as partes essenciais da função sum, percebe-se que a chamada da função insert ocorre dentro de um loop de complexidade $O(m)$ e possui um tempo de execução de $O(n)$. Dessa forma, a complexidade nessa seção do código será $O(m * n)$. Além disso, esse laço de $O(m)$ está aninhado dentro de outro loop com complexidade $O(n)$, o que resulta em uma complexidade total de $O(n * m * n)$, simplificada para $O(m * n^2)$. No caso em que $m = n$, a complexidade final se torna $O(n^3)$.

```
SparseMatrix sum(SparseMatrix& A, SparseMatrix& B){
    if(A.getLinhas() == B.getLinhas() && A.getCol() == B.getCol()){ //O(1)
        SparseMatrix C(A.getLinhas(), B.getCol()); // O(n)
        for(int i=1;i<=A.getLinhas();i++){ // O(n)
            for(int j=1;j<=B.getCol();j++){ // O(m)
                try {
                    C.insert(A.get(i, j)+B.get(i, j), i, j); // O(n)
                } catch (invalid_argument e){
                    cout<<e.what()<<endl;
                }
            }
        }
        return C; // O(1)
    } else {
        throw invalid_argument("As matrizes possuem dimensoes diferentes");
    }
}
```

Figura 7. Função sum

5. Relatório em Latex

Este relatório foi feito em Latex através do Overleaf. Segue o link para visualizar o projeto `https://pt.overleaf.com/read/hjdccybqmtkn#36d798`

Referências

Brownlee, J. (2019). A gentle introduction to sparse matrices for machine learning. `https://machinelearningmastery.com/sparse-matrices-for-machine-learning/`.

Equipe Estatística Fácil. O que é: Sparse matrix (matriz esparsa). `https://estatisticafacil.org/glossario/o-que-e-sparse-matrix-matriz-esparsa/`.