Matriz Esparsa

Kalmax S. Sousa[1], **Christian E. Barbosa**[1]

[1]Universidade Federal do Ceará (UFC)
Quixadá – CE – Brasil

Abstract. *This document consists of describing the implementation of a Sparse Floating-Point Matrix, formed by simply circular linked lists. The work was developed in C++ and uses Object Oriented concepts.*

Resumo. *Esse documento consiste em descrever a implementação de uma Matriz Esparsa de pontos flutuantes, formada por listas circulares simplesmente encadeadas. O trabalho foi desenvolvido em C++ e utiliza conceitos de Orientação a Objeto.*

1. Apresentação

Matrizes esparsas são matrizes que possuem uma grande quantidade de elementos nulos. Devido a esse comportamento, a representação de matrizes esparsas em memória é diferente da representação de matrizes densas. Alguns exemplos da utilização dessa estrutura são os mapas de bits em imagens, armazenamento de dados em planilhas eletrônicas, método das malhas para resolução de circuitos elétricos, matrizes de adjacência de grafos, dentre outros. Nesse trabalho, foi implementado um TAD para representar matrizes esparsas, utilizando listas circulares simplemente encadeadas. O trabalho contém 6 arquivos, sendo eles: SparseMatrix.h, Node.h, SparseMatrix.cpp, main.cpp e arquivos de testes (A.txt, B.txt).

1.1. Lista encadeada circular

Dentre diversas formas de representar esse tipo de matriz, a utilizada foi a lista circular simplemente encadeada. Essa lista é formada por um conjunto de objetos do mesmo tipo espalhados na memória e ligados através do armazenamento do endereço do próximo objeto. Nesse tipo especial de lista encadeada, o último objeto armazena o endereço do primeiro, criando a circularidade. Apesar de utilizar pesquisa linear, essa estrutura de dados foi escolhida por ser mais eficiente para a representação de matrizes esparsas, visto que a quantidade de elementos não nulos é muito menor que a quantidade de elementos nulos, tendo como principal vantagem a diminuição do uso de memória.

1.2. Motivação

Para construir essa estrutura, é necessário criar um tipo de objeto (Node.h) que armazena o número da coluna e o da linha em que está localizado, dois ponteiros, um que guarda o endereço do próximo objeto horizontalmente e outro o endereço do próximo verticalmente, e um campo do tipo double para armazenar o valor. Esses objetos são partes das listas circulares que se cruzam, portanto, um objeto está presente em duas listas. E para criar isso, foram utilizados ”nós cabeça” para marcar o início de cada uma das listas e além disso, os objetos que represetam os elementos nulos não devem ser criados. O TAD foi implementado em C++ e possui as seguintes operações: construtor, destrutor, inserção

de elementos, acesso a elementos e impressão de matrizes. Além de ser manipulado na main para somar e multiplicar matrizes. Ademais, o projeto também conta com o TAD Node.h, que representa cada objeto dessa matriz..

2. Classe SparseMatrix

A ultilização da programação orientada a objetos é uma forma de organizar o código de forma que seja mais fácil de entender e de manter. Ultilizamos esse paragdima para a criação da classe SparseMatrix, no processo julgamos ser melhor separar todos os métodos e atributos dessa classe em 2 arquivos diferentes, um para a declaração da classe e outro para a implementação dos métodos.

2.1. SparseMatrix.h

No arquivo SparseMatrix.h contém a declaração da classe SparseMatrix e de alguns atributos e métodos da classe.

2.2. SparseMatrix.cpp

No arquivo SparceMatrix.cpp contém a implementação dos métodos da classe SparseMatrix, sendo eles: construtor, destrutor, inserir elemento na matrix, retornar elemento da matrix e imprimir a matrix.

2.2.1. Construtor

O contrutor da classe SparseMatrix recebe o número de linhas e colunas da matriz e inicializa os atributos da classe com esses valores. Além disso, ele cria um nó cabeça para cada linha e coluna da matriz, e os conecta entre si, inicializando as listas. Sua complexidade é de $O(n)$, visto que só percorre as colunas e depois as linhas.

2.2.2. Destrutor

O destrutor da classe SparseMatrix percorre a matriz e deleta todos os nós da matriz, começando deletando os nós das listas verticais e em seguida os nós cabeça restantes. Tendo complexidade $O(n^2)$.

2.2.3. .insert()

O metodo insert recebe uma linha, uma coluna e um valor, e insere esse valor na matriz na posição (linha, coluna). Caso já exista um valor na posição (linha, coluna), ele é substituído pelo novo valor. Caso o valor seja zero, o nó que contém esse valor é removido da matriz. Portanto a complexidade desse método é O(n), visto que o pior caso é quando o valor é zero e o nó que contém esse valor está no final da lista.

2.2.4. .get()

O metodo get recebe como parâmetro a linha e coluna do elemento a ser retornado. O método percorre a matriz até encontrar a linha e coluna desejada, e então retorna o valor do

elemento. Caso o elemento não exista, o método retorna 0. Devido isso, sua complexidade no pior caso é $O(n^2)$.

2.2.5. .print()

O método print imprime a matriz na tela. Ele deve percorrer toda a matriz, campo a campo, e imprime o valor de cada campo ou zero, caso uma posição tenha o valor nulo. Portanto, a complexidade dessa função é $O(n * m)$, que no pior caso, quando m (linhas) e n (colunas) forem iguais, $O(n^2)$.

2.2.6. .getColumns()

O método getRows retorna o número de linhas da matriz. Decidimos criar esse método para auxiliar outras funções, onde é essa informação é necessária. O método possui complexidade $O(1)$.

2.2.7. .getRows()

O método getRows retorna o número de linhas da matriz. Decidimos criar esse método para auxiliar outras funções, onde é essa informação é necessária. O método possui complexidade $O(1)$.

3. Main.cpp

A função main ler duas matrizes esparsas A e B de arquivos de texto e imprimir as matrizes A, B, C = A + B e D = A * B. A função deve lançar uma exceção caso as matrizes não possam ser somadas ou multiplicadas e essa excessão é tratada. A main ultliza todas as funções criada na classe SparseMatrix para realizar as operações, mais as funções read, sum e multiply que são criadas fora da classe.

3.1. Função readSparseMatrix

A função readSparseMatrix lê uma matriz esparsa de um arquivo de texto e retorna um ponteiro para a matriz lida (SparseMatrix*). A função deve lançar uma exceção caso o arquivo não exista ou não esteja no formato correto. Visto que ela usa a função insert, que retorna um erro caso a posição seja inválida. Sua complexidade é $0(n)$, dependendo do número de linhas do arquivo lido.

3.2. Função sum

A função sum recebe duas matrizes esparsas e retorna uma terceira matriz esparsa que é a soma das duas primeiras. As matrizes só podem ser somadas se tiverem o mesmo número de linhas e colunas, caso isso não aconteça, é dispara uma exceção e vai ser impressa a mensagem: ”As matrizes não podem ser somadas”. A complexidade dessa função é $O(n^2)$, visto que o pior caso é quando a matriz resultante é a soma de duas matrizes esparsas com todos os elementos diferentes de zero.

3.3. Função multiply

A função multiply recebe duas matrizes esparsas A e B e retorna uma nova matriz esparsa C = A * B. A função deve ser implementada em $O(n^2)$, onde n é o número de elementos não nulos da matriz C. A função deve lançar uma exceção caso as matrizes não possam ser multiplicadas.

4. Divisão de trabalho

Inicialmente buscamos juntos entender como representar essa matriz e julgamos melhor separar quais funções cada um iria implementar. Criamos um repositório no GitHub para melhorar a união dos códigos e construímos a estrutura básica dos arquivos, nomeando as funções e variáveis. A divisão das implementações ficou a seguinte, Christian: destrutor, insert, read e sum, e Kalmax: construtor, get, print e multiply. Porém, mesmo com as funções divididas, sempre que algum concluia uma implementação, o outro revisava, testava e até modificava, caso fosse necessário. Já a construção do relatório foi feito em conjunto.

5. Dificuldades encontradas

Não ocorreram muitos empasses ao longo da construção do projeto, porém, podemos citar como dificuldade algumas inconsistências que tivemos com o repositório no início do projeto, além de alguns problemas com loops infinitos dentro das funções e algumas dúvidas básicas que surgiram. No entanto, tudo isso foi simples de resolver e em conjunto conseguimos superar tudo com êxito.

6. Testes executados

6.1.

Input:

A.txt

```
4 4
1 1 50.0
2 1 10.0
2 3 20.0
4 1 -30.0
4 3 -60.0
4 4 -5.0
```

B.txt

```
cuzzc 4 3
1 1 1.4
1 2 1.2
1 3 1.3
3 1 10
3 2 10
3 3 10
```

Output:

```
A:
50 0 0 0
10 0 20 0
0 0 0 0
-30 0 -60 -5
B:
1.4 1.2 1.3
```

0 0 0
10 10 10
0 0 0
As matrizes não podem ser somadas
D:
70 60 65
214 212 213
0 0 0
-642 -636 -639

6.2.

Input:

A.txt

7 8
1 3 1267.23
3 1 50.0
5 6 0.126
6 4 10.0
7 8 657

B.txt

7 8
1 5 19
3 6 73.123
3 8 123
6 8 92347

Output:

A:
0 0 1267.23 0 0 0 0 0
0 0 0 0 0 0 0 0
50 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0.126 0 0
0 0 0 10 0 0 0 0
0 0 0 0 0 0 0 657
B:
0 0 0 0 19 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 73.123 0 123
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 92347
0 0 0 0 0 0 0 0
C:
0 0 1267.23 0 19 0 0 0
0 0 0 0 0 0 0 0
50 0 0 0 0 73.123 0 123
0 0 0 0 0 0 0 0
0 0 0 0 0 0.126 0 0
0 0 0 10 0 0 0 92347

0 0 0 0 0 0 0 657
As matrizes não podem ser multiplicadas

6.3.

Input:

A.txt	B.txt
1 1	1 1
1 1 2	1 1 2

Output:

A:
2
B:
2
C:
4
D:
4

7. Referências

KOFFMAN, Elliot B. e Paul A. T. Wolfgang. **Object, abstraction, data structures and design using C++**. John Wiley Sons, Inc. 2006.

COSTA, Adelmiro Diniz. **Armazenamento, recuperação e tratamento de matrizes esparsas de grande porte.** 1980. 164 f. Tese: Mestre em ciências (Engenharia de Sistemas e Computação) - Universidade Federal do Rio de Janeiro, COPPE, Rio de Janeiro, 1980. Disponível em: https://www.cos.ufrj.br/uploadfile/1364228573.pdf. Acesso em: 14 set. 2022.