Relatório: Implementação de Matriz Esparsa em C++

Nauan Aires e Victor Cavalcante

12 de junho de 2023

1 Introdução

Neste relatório, descrevemos a implementação de uma Matriz Esparsa em C++. Uma matriz esparsa é uma matriz na qual a maioria dos elementos é zero. Para economizar espaço e melhorar a eficiência em operações com matrizes esparsas, utilizamos uma estrutura de dados especializada para representar apenas os elementos não nulos da matriz. Qualquer informação adicional ou explicações mais diretas e pontuais sobre cada função e método utilizado pode ser consultado através do Github do projeto linkado na seção de Referências (Clique aqui para ir a seção)

2 Estrutura de Dados

Para implementar a matriz esparsa, utilizamos uma lista de listas (ou vetor de vetores) para armazenar os elementos não nulos. Cada elemento não nulo é representado por um objeto que contém a posição do elemento na matriz (linha e coluna) e o valor do elemento. A estrutura do projeto foi definida com os seguintes arquivos:
"Node.h"
"SparseMatrix.h"
"main.cpp"
E o nó é definido pela seguinte estrutura:

```
struct Node {
    Node *direita;
    Node *abaixo;
    int linha;
    int coluna;
```

```
    double valor;

    Node(Node *dir, Node *ab, int l, int c, double v) {
        this->direita = dir;
        this->abaixo = ab;
        this->linha = l;
        this->coluna = c;
        this->valor = v;
    }
};
```

Nessa implementação, cada linha da matriz esparsa é representada por um vetor de elementos não nulos. A matriz esparsa em si é um vetor de linhas.

3 Operações

Implementamos as principais operações, como a adição de uma matriz por arquivo, a soma de matriz + matriz, multiplicação de matrizes esparsas, criação de um arquivo através de uma matriz inserida e a leitura de um arquivo contendo uma matriz dentro da aplicação.

Segue abaixo a listagem e demonstração de cada caso de operação.

3.1 Caso 1 - Adicionar Matriz

Para adicionar uma nova matriz (convertida em matriz esparsa pelo código) é necessário que o Usuário digite primeiramente o número de linhas e colunas utilizadas por essa matriz, dado por esta implementação:

```
switch(op){
    case 1: {
    cout << endl << "Criacao da matriz..." << endl;
    int m, n, value;
    cout << "Insira o numero de linhas e colunas: ";
    cin >> m >> n;

    SparseMatrix* matriz = new SparseMatrix(m, n);

    cout << endl;
    matriz->print();
```

```
cout << endl;

int c {0};
while(c == 0){
    cout << "Insira a posicao e valor na matriz: ";
    cin >> m >> n >> value;
    matriz->insert(m, n, value);
    cout << endl;
    matriz->print();
    cout << endl;
    cout << "Finalizar?" << endl;
    cout << "1 - Sim" << endl << "0 - Nao" << endl;
    cin >> c;
    cout << endl;
}

cout << "Matriz adicionada com sucesso!" << endl <<
    endl;
```

Após a adicionar a nova matriz, é dada a opção do usuário salvar a matriz criada em um arquivo, segue abaixo a implementação:

```
cout << "Matriz adicionada com sucesso!" << endl <<
    endl;
cout << "Salvar em um arquivo?" << endl;
cout << "1 - Sim" << endl << "0 - Nao" << endl <<
   endl;
int key;
cin >> key;
if(key == 1){
    cout << "Digite o nome do arquivo para salvar a
       matriz: ";
    string filename;
    cin >> filename;

    ofstream file(filename);
    if(!file.is_open()){
        throw std::runtime_error("Nao foi possivel
           abrir o arquivo: " + filename);
    }

    file << matriz->getMaxM() << " " << matriz->
```

```
            getMaxN() << std::endl;

        for(int i = 0; i < matriz->getMaxM(); i++) {
            for(int j = 0; j < matriz->getMaxN(); j++)
                {
                double value = matriz->get(i, j);
                if(value != 0.0){
                    file << i << " " << j << " " <<
                        value << std::endl;
                }
            }
        }

        file.close();

        cout << "Matriz salva com sucesso!" << endl;
    }
        break;
```

3.2 Caso 2 - Somar Matrizes

A soma das matrizes é representada pela soma de dois arquivos que precisam estar presentes dentro da pasta do projeto, sendo assim possível a soma esses dois arquivos através da seguinte implementação:

```
//Método para ler os arquivos contendo matrizes
SparseMatrix *readSparseMatrix(const std::string &
    nomeArquivo){
    std::ifstream arquivo(nomeArquivo);
    if(!arquivo.is_open()){
        throw std::runtime_error("Nao foi possivel
            abrir o arquivo: " + nomeArquivo);
    }

    int m, n;
    arquivo >> m >> n; // Lê as dimensões da matriz

    SparseMatrix *matriz = new SparseMatrix(m, n);

    while(true){
        double value;
```

```
            arquivo >> m >> n >> value;
            if(arquivo.eof()) break;
            matriz->insert(m, n, value);
        }

        arquivo.close();

        return matriz;
    }

    //Método para realizar a soma
    SparseMatrix *sum(SparseMatrix *A, SparseMatrix *B){
        // Verifica se as matrizes têm o mesmo tamanho
        if(A->getMaxM() != B->getMaxM() || A->getMaxN() !=
           B->getMaxN()){
            throw std::invalid_argument("As matrizes devem
               ter o mesmo tamanho para serem somadas.");
        }

        int m = A->getMaxM();
        int n = A->getMaxN();

        // Cria uma nova matriz C com o mesmo tamanho de A
           e B
        SparseMatrix *C = new SparseMatrix(m, n);
        // Percorre todas as posições da matriz
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                // Obtém os valores de A e B nas posições (
                   i, j)
                double valueA = A->get(i, j);
                double valueB = B->get(i, j);

                // Calcula a soma dos valores e insere na
                   matriz C
                double sum = valueA + valueB;
                C->insert(i, j, sum);
            }
        }
```

```
    return C; // Retorna a matriz C resultante da soma
        de A e B
}

case 2: {
    cout << endl << "Digite o nome do arquivo da
        primeira matriz: ";
    string filenameA;
    cin >> filenameA;

    SparseMatrix *A = readSparseMatrix(filenameA);

    cout << "Digite o nome do arquivo da segunda matriz
        : ";
    string filenameB;
    cin >> filenameB;

    SparseMatrix *B = readSparseMatrix(filenameB);

    SparseMatrix *C = sum(A, B);
    cout << "Matriz multiplicada com sucesso!" << endl;
    cout << endl;
    C->print();
    cout << endl;
    cout << "Salvar em um arquivo?" << endl;
    cout << "1 - Sim" << endl << "0 - Nao" << endl;
    int key;
    cin >> key;
    if(key == 1){
        cout << "Digite o nome do arquivo para salvar a
            matriz: ";
        string filename;
        cin >> filename;

        ofstream file(filename);
        if(!file.is_open()){
            throw std::runtime_error("Nao foi possível
                abrir o arquivo: " + filename);
        }
```

```
            file << C->getMaxM() << " " << C->getMaxN() <<
                std::endl;

            for (int i = 0; i < C->getMaxM(); i++) {
                for (int j = 0; j < C->getMaxN(); j++) {
                    double value = C->get(i, j);
                    if(value != 0.0){
                        file << i << " " << j << " " <<
                            value << std::endl;
                    }
                }
            }

            file.close();

            cout << "Matriz salva com sucesso!" << endl;
            cout << endl;
        }


        // Liberar memória
        delete A;
        delete B;
        delete C;

        break;
}
```

3.3 Caso 3 - Multiplicar Matrizes

Neste caso de multiplicação de matrizes é utilizado a mesma função de leitura citado no caso anterior, porém dessa vez utilizaremos um método de multiplicação citado logo abaixo na implementação, e em seguida a exibição e lógica para o caso 3.

```
//Lógica/Método para a função de multiplicação (
    multiply)
SparseMatrix *multiply(SparseMatrix *A, SparseMatrix *B
    ) {
     if(A->getMaxN() != B->getMaxM()){
```

```
            throw std::runtime_error("As matrizes nao podem
                ser multiplicadas. O numero de colunas de A
                deve ser igual ao numero de linhas de B.");
        }

        int m = A->getMaxM();
        int n = B->getMaxN();
        int p = B->getMaxM();

        SparseMatrix *C = new SparseMatrix(m, p);  // C ter
            á dimensões m x p

        for(int i = 0; i < m; i++){
            for(int j = 0; j < p; j++){
                int sum = 0;
                for(int k = 0; k < n; k++){
                    sum += A->get(i, k) * B->get(k, j);  //
                        Multiplica os elementos
                        correspondentes e soma
                }
                if(sum != 0){
                    C->insert(i, j, sum);  // Insere o
                        elemento não nulo na matriz
                        resultante C
                }
            }
        }

        return C;
    }

    case 3: {
        cout << endl << "Digite o nome do arquivo da
            primeira matriz: ";
        string filenameA;
        cin >> filenameA;

        SparseMatrix *A = readSparseMatrix(filenameA);

        cout << "Digite o nome do arquivo da segunda matriz
            : ";
```

```
string filenameB;
cin >> filenameB;

SparseMatrix *B = readSparseMatrix(filenameB);

SparseMatrix *C = multiply(A, B);
cout << "Matriz multiplicada com sucesso!" << endl;
cout << endl;
C->print();
cout << endl;
cout << "Salvar em um arquivo?" << endl;
cout << "1 - Sim" << endl << "0 - Nao" << endl;
int key;
cin >> key;
if(key == 1){
    cout << "Digite o nome do arquivo para salvar a
       matriz: ";
    string filename;
    cin >> filename;

    ofstream file(filename);
    if(!file.is_open()){
        throw std::runtime_error("Nao foi possivel
           abrir o arquivo: " + filename);
    }

    file << C->getMaxM() << " " << C->getMaxN() <<
       std::endl;

    for(int i = 0; i < C->getMaxM(); i++){
        for(int j = 0; j < C->getMaxN(); j++){
            double value = C->get(i, j);
            if(value != 0){
                file << i << " " << j << " " <<
                   value << std::endl;
            }
        }
    }

    file.close();
```

```
            cout << "Matriz salva com sucesso!" << endl;
            cout << endl;
        }

        // Liberar memória
        delete A;
        delete B;
        delete C;

        break;
    }
```

3.4 Caso 4 - Abrir Arquivo

O caso 4 tem como função exclusiva apenas ler um arquivo através do nome fornecido, utilizará a mesma função Read dos outros casos, porém será filtrado pelo nome, exemplo: 1- se for fornecido o nome M1, irá ler o arquivo M1, que contém uma matriz pré-definida.

```
    case 4: {
        cout << endl << "Digite o nome do arquivo da matriz
            : ";
        string filenameA;
        cin >> filenameA;
        cout << endl;
        SparseMatrix *A = readSparseMatrix(filenameA);
        A->print();

        break;
    }
```

3.5 Caso 5 - Sair

Esse caso tem como função exclusiva sair e finalizar a aplicação, para que o usuário tenha controle visual e também que seja intuitivo para todos os tipos de usuários. Segue abaixo a implementação:

```
    case 5:{
        cout << "Saindo..." << endl;
        return 0;
    }
```

4 Resultados

Testamos nossa implementação utilizando diversas matrizes esparsas de diferentes tamanhos e realizamos operações de Carregar matriz por arquivo, Soma, Multiplicação e Criação de novo texto de matriz. Os resultados foram consistentes e a implementação mostrou-se eficiente para lidar com matrizes esparsas.

Todos os testes podem ser consultados através dos arquivos com a extensão .txt anexados a pasta compactada deste mesmo documento.

Também é possível ver com mais detalhes as implementações mais detalhada sobre cada função através do arquivo Readme.md no GitHub, ou até mesmo anexado a pasta compactada.

5 Como o trabalho foi Dividido e Desenvolvido

O trabalho foi desenvolvido com ajuda do Git e plataforma GitHub, onde Victor Cavalcante tinha domínio sobre a ferramenta e o mesmo também queria manter toda a aplicação em seu repositório GitHub e Nauan como não dominava a ferramenta e tinha um pouco de dificuldade com o conteúdo, o mesmo baixava os arquivos através do repositório GitHub de Victor, modificava e envia novamente para o mesmo fazer atualizar o código dentro da plataforma, afim de prevenir erros e evitar dor de cabeças com o código, pois o mesmo nunca tinha utilizado.
Como Victor tinha mais experiência e facilidade com a parte da programação, fez maioria de suas funções e métodos.
Nauan Fez algumas funções e lógicas mais simples como o Destrutor localizado em SparseMatrix.h, representado por:

```
SparseMatrix::~SparseMatrix(){
    Node *auxLinha = m_head->abaixo;

    while (auxLinha != m_head) {
        Node *auxColuna = auxLinha->direita;
        while (auxColuna != auxLinha) {
            Node *temp = auxColuna;
            auxColuna = auxColuna->direita;
            delete temp;
        }
        Node *temp = auxLinha;
```

```
        auxLinha = auxLinha->abaixo;
        delete temp;
    }

    delete m_head;
}
```

Também a função Print, para a visualização da Matriz:

```
void SparseMatrix::print(){
    int m = this->maxm; // Armazena o número de linhas
        da matriz
    int n = this->maxn; // Armazena o número de colunas
        da matriz

    for (int i = 0; i < m; i++){ // Percorre as linhas
        da matriz
        for (int j = 0; j < n; j++){ // Percorre as
            colunas da matriz
            std::cout << get(i, j) << " "; // Obtém o
                valor da posição (i, j) e imprime na
                tela
        }
        std::cout << '\n'; // Quebra de linha após
            imprimir uma linha completa da matriz
    }
}
```

A Função de Soma, representada abaixo:

```
SparseMatrix *sum(SparseMatrix *A, SparseMatrix *B){
    // Verifica se as matrizes têm o mesmo tamanho
    if(A->getMaxM() != B->getMaxM() || A->getMaxN() !=
        B->getMaxN()){
        throw std::invalid_argument("As matrizes devem
            ter o mesmo tamanho para serem somadas.");
    }

    int m = A->getMaxM();
    int n = A->getMaxN();

    // Cria uma nova matriz C com o mesmo tamanho de A
        e B
```

```
    SparseMatrix *C = new SparseMatrix(m, n);
    // Percorre todas as posições da matriz
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            // Obtém os valores de A e B nas posições (
               i, j)
            double valueA = A->get(i, j);
            double valueB = B->get(i, j);

            // Calcula a soma dos valores e insere na
               matriz C
            double sum = valueA + valueB;
            C->insert(i, j, sum);
        }
    }

    return C; // Retorna a matriz C resultante da soma
       de A e B
}
```

E também a função de Ler Arquivos, representada abaixo:

```
SparseMatrix *readSparseMatrix(const std::string &
   nomeArquivo){
    std::ifstream arquivo(nomeArquivo);
    if(!arquivo.is_open()){
        throw std::runtime_error("Nao foi possivel
           abrir o arquivo: " + nomeArquivo);
    }

    int m, n;
    arquivo >> m >> n; // Lê as dimensões da matriz

    SparseMatrix *matriz = new SparseMatrix(m, n);

    while(true){
        double value;
        arquivo >> m >> n >> value;
        if(arquivo.eof()) break;
        matriz->insert(m, n, value);
    }
```

```
    arquivo.close();

    return matriz;
}
```

E Nauan Fez boa parte do relatório em LaTeX para equilibrar os trabalhos, em relação ao Relatório Victor trabalhou em:
• Seção 6 - Dificuldade encontradas
• Seção 7 - Análise de complexidade de pior caso das funções: insert, get e sum
• Seção 8 - Listagem dos testes
• Seção 9 - Referências

6 Dificuldades encontradas

As principais dificuldades encontradas foram com a função Insert, onde quando ia passar para próxima linha sempre quebrava o código e mostrava apenas zeros. A dificuldade foi resolvida através de um amigo, Gustavo Henrique de (ES) que mora juntamente com Victor.

E que por conta desse mesmo problema também foi encontrado problemas no construtor da matriz, que foi resolvido com o mesmo amigo.

7 Análise de complexidade de pior caso das funções: insert, get e sum

Esta aqui a implementação do Insert, onde a complecidade de cada ponto estará comentada:

```
void SparseMatrix::insert(int m, int n, double value){
    // Verifica se as coordenadas estão dentro dos
       limites válidos da matriz
    if (m < 0 || n < 0 || m >= this->maxm || n >= this
       ->maxn){
        throw std::out_of_range("Posição inválida na
           matriz.");
    }

    // Encontra o nó da linha apropriada para inserção
    Node *linhaH = this->m_head->abaixo;
```

```
while (linhaH != m_head && linhaH->linha < m){
    linhaH = linhaH->abaixo;
}

// Encontra o nó da coluna apropriada para inserção
Node *colunaH = m_head->direita;

while (colunaH != m_head && colunaH->coluna < n){
    colunaH = colunaH->direita;
}

// Encontra o nó vizinho à direita na mesma linha
Node *noVizinhoC = linhaH;

while (noVizinhoC->direita != linhaH && noVizinhoC
   ->direita->coluna < n){
    noVizinhoC = noVizinhoC->direita;
}

// Encontra o nó vizinho abaixo na mesma coluna
Node *noVizinhoL = colunaH;

while (noVizinhoL->abaixo != colunaH && noVizinhoL
   ->abaixo->linha < m){
    noVizinhoL = noVizinhoL->abaixo;
}

// Se o elemento já existe na posição, atualiza seu
    valor
if (noVizinhoL->direita->linha == m && noVizinhoC->
   abaixo->coluna == n){
    noVizinhoL->valor = value;
    return;
}

// Cria um novo nó e o insere na posição apropriada
Node *novo = new Node(noVizinhoC->direita,
   noVizinhoL->abaixo, m, n, value);
noVizinhoC->direita = novo;
noVizinhoL->abaixo = novo;
```

}

Verificação das coordenadas: Essa parte possui uma complexidade constante O(1), pois envolve apenas comparações e verificações simples.

Busca do nó da linha apropriada: A complexidade depende do número de linhas presentes antes da linha desejada. No pior caso, em uma matriz esparsa completamente preenchida, a busca percorrerá todas as linhas da matriz até encontrar a posição correta. Portanto, a complexidade dessa parte é O(n), onde n é o número total de linhas da matriz.

Busca do nó da coluna apropriada: De maneira similar à busca da linha, a complexidade depende do número de colunas presentes antes da coluna desejada. No pior caso, em uma matriz esparsa completamente preenchida, a busca percorrerá todas as colunas da matriz até encontrar a posição correta. Assim, a complexidade dessa parte é O(m), onde m é o número total de colunas da matriz.

Busca do nó vizinho à direita na mesma linha: Essa busca percorre os nós da linha até encontrar a posição correta. No pior caso, onde todos os elementos da linha estão preenchidos, a busca percorrerá todos os nós da linha. Portanto, a complexidade é O(n), onde n é o número total de colunas da matriz.

Busca do nó vizinho abaixo na mesma coluna: De maneira similar à busca na linha, essa busca percorre os nós da coluna até encontrar a posição correta. No pior caso, onde todos os elementos da coluna estão preenchidos, a busca percorrerá todos os nós da coluna. Assim, a complexidade é O(m), onde m é o número total de linhas da matriz.

Verificação se o elemento já existe: Essa verificação envolve apenas comparações de valores de nós, portanto, possui uma complexidade constante O(1).

Inserção de um novo nó: Essa parte envolve a criação de um novo nó e a atualização dos ponteiros dos nós vizinhos. Portanto, possui uma complexidade constante O(1).

Segue abaixo a análise da função get():

```
double SparseMatrix::get(int m, int n){
    // Verifica se as coordenadas estão dentro dos
       limites válidos da matriz
    if (m < 0 || n < 0 || m >= this->maxm || n >= this
```

```
        ->maxn){
            throw std::out_of_range("Posicao invalida na
                matriz.");
        }

        // Encontra o nó da linha correta na lista
            encadeada
        Node *linhaH = m_head->abaixo;

        while (linhaH != m_head && linhaH->linha < m){
            linhaH = linhaH->abaixo;
        }

        // Verifica se a linha correta foi encontrada
        if (linhaH != m_head && linhaH->linha == m){
            Node *colunaH = linhaH->direita;
            // Percorre os nós na linha até encontrar a
                coluna desejada
            while (colunaH != linhaH && colunaH->coluna !=
                n){
                colunaH = colunaH->direita;
            }

            // Verifica se a coluna correta foi encontrada
            if (colunaH != linhaH && colunaH->coluna == n){
                return colunaH->valor; // Retorna o valor
                    do nó
            }
        }

        return 0; // Retorna 0 se não foi encontrado nenhum
            valor na posição especificada
}
```

Verificação das coordenadas: Essa parte possui uma complexidade constante O(1), pois envolve apenas comparações e verificações simples.

Busca do nó da linha correta: A complexidade depende do número de linhas presentes antes da linha desejada. No pior caso, em uma matriz esparsa completamente preenchida, a busca percorrerá todas as linhas da matriz até encontrar a posição correta. Portanto, a complexidade dessa parte é O(n),

onde n é o número total de linhas da matriz.

Verificação da existência da linha correta: Essa verificação envolve apenas comparações de valores de nós, portanto, possui uma complexidade constante O(1).

Busca do nó da coluna correta: Após encontrar a linha correta, essa busca percorre os nós na linha até encontrar a coluna desejada. No pior caso, onde todos os elementos da linha estão preenchidos, a busca percorrerá todos os nós da linha. Portanto, a complexidade é O(n), onde n é o número total de colunas da matriz.

Verificação da existência da coluna correta: Essa verificação envolve apenas comparações de valores de nós, portanto, possui uma complexidade constante O(1).

Segue abaixo a análise da função sum(soma):

```
SparseMatrix *sum(SparseMatrix *A, SparseMatrix *B){
    // Verifica se as matrizes têm o mesmo tamanho
    if(A->getMaxM() != B->getMaxM() || A->getMaxN() !=
       B->getMaxN()){
        throw std::invalid_argument("As matrizes devem
            ter o mesmo tamanho para serem somadas.");
    }

    int m = A->getMaxM();
    int n = A->getMaxN();

    // Cria uma nova matriz C com o mesmo tamanho de A
       e B
    SparseMatrix *C = new SparseMatrix(m, n);
    // Percorre todas as posições da matriz
    for(int i = 0; i < m; i++){
        for(int j = 0; j < n; j++){
            // Obtém os valores de A e B nas posições (
               i, j)
            double valueA = A->get(i, j);
            double valueB = B->get(i, j);

            // Calcula a soma dos valores e insere na
```

```
                matriz C
                double sum = valueA + valueB;
                C->insert(i, j, sum);
            }
        }

        return C; // Retorna a matriz C resultante da soma
            de A e B
}
```

Verificação do tamanho das matrizes: Essa parte envolve comparações dos tamanhos das matrizes A e B. Como essas informações são obtidas diretamente através dos métodos getMaxM() e getMaxN(), a complexidade é O(1), ou seja, constante.

Criação da matriz C: A criação da matriz C envolve a alocação de memória para a nova matriz e a inicialização de seus atributos de tamanho. Essa operação possui uma complexidade constante O(1).

Percorrendo todas as posições das matrizes A e B: Essa parte envolve dois loops aninhados que percorrem todas as posições das matrizes A e B. O número de iterações é determinado pelo tamanho das matrizes, m e n. Portanto, a complexidade dessa parte é O(m * n), onde m é o número de linhas e n é o número de colunas.

Obtendo os valores de A e B e calculando a soma: Essa operação envolve a chamada dos métodos get() para obter os valores das matrizes A e B nas posições (i, j), e a soma dos valores obtidos. Essas operações têm uma complexidade constante O(1).

Inserção do valor calculado na matriz C: A inserção do valor calculado na matriz C envolve a chamada do método insert(), que tem uma complexidade constante O(1).

8 Listagem dos testes

Todos os testes realizados foram utilizando os arquivos txt anexados junto ao projeto. Segue abaixo a pré visualização dos dois arquivos:

M1.txt

```
3  3
0  0  1
0  1  2
0  2  3
1  0  4
1  1  5
1  2  6
2  0  7
2  1  8
2  2  9
```

M2.txt

```
3  3
0  0  1
0  1  1
0  2  1
1  0  1
1  1  1
1  2  1
2  0  1
2  1  1
2  2  1
```

soma.txt

```
3  3
0  0  2
0  1  3
0  2  4
1  0  5
1  1  6
1  2  7
2  0  8
2  1  9
2  2  10
```

9 Referências

1. Minicurso de Latex (Link aqui)
2. Vídeo para auxiliar construção e lógica inicial do projeto (Link aqui)
3. Projeto concluído no GitHub e acesso a informações extras (Link aqui)

4. Repositório GitHub que utilizamos para nos orientar (Link aqui)
5. Ferramenta de I.A utilizada para auxiliar a encontrar erros e soluções no código e relatório LaTeX (Link aqui)

10 Conclusão

Neste trabalho, apresentamos a implementação de uma Matriz Esparsa em C++ utilizando uma estrutura de dados especializada. A implementação mostrou-se eficiente para lidar com matrizes esparsas e permite realizar as principais operações de forma adequada. A utilização de uma estrutura de dados especializada para matrizes esparsas é fundamental para economizar espaço e melhorar a eficiência em operações com matrizes esparsas.

Qualquer dúvida ou informação adicional pode ser consultada através do Github do projeto (link listado acima).