Matrizes Esparsas

Jose Nilson Bernardo Nunes Filho[1], **João Pedro Batista**[1]

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

nilsonde@alu.ufc.br, joaosvc@alu.ufc.br

Abstract. *This academic work aims to demonstrate the creation of a sparse matrix, implementing its methods and functions, in addition to verifying the efficiency of its use. A presentation of all the functions developed is made, accompanied by a more in-depth analysis of some of them, such as complexity calculation. At the end, it is reported how the work was divided between the participants and personal opinions about each one's experience are presented.*

Resumo. *Este trabalho acadêmico tem como objetivo demonstrar a criação de uma matriz esparsa, implementando seus métodos e funções, além de verificar a eficiência de sua utilização. É feita uma apresentação de todas as funções desenvolvidas, acompanhada de uma análise mais aprofundada de algumas delas, como o cálculo de complexidade. Ao final, é relatado como o trabalho foi dividido entre os participantes e apresentadas opiniões pessoais sobre a experiência de cada um.*

1. Introdução

Matrizes esparsas desempenham um papel fundamental em diversas áreas da computação, como álgebra linear, otimização, aprendizado de máquina e computação científica. Muitas vezes, essas matrizes surgem em cenários onde apenas uma pequena fração dos elementos possui valores diferentes de zero, como em redes sociais, sistemas de recomendação e simulações físicas.

A representação tradicional de matrizes densas pode ser extremamente ineficiente em termos de armazenamento e processamento quando aplicada a esses casos. Por isso, o estudo e a implementação de estruturas de dados específicas para matrizes esparsas são essenciais para otimizar recursos e melhorar o desempenho computacional.

Este trabalho busca explorar as estruturas e algoritmos para manipulação de matrizes esparsas.

2. Implementação

Para criar a matriz esparsa, foi utilizado o TAD (Tipo Abstrato de Dados). Assim, foram criados os seguintes arquivos:

- `node.hpp`: contém a definição da `struct Node`, que representa cada nó da matriz;
- `matrix.hpp`: contém a declaração da classe `SparseMatrix`, responsável pela implementação da matriz esparsa;

- `matrix.cpp`: contém a implementação dos métodos da classe `SparseMatrix`;
- `main.cpp`: implementa uma interface interativa para a manipulação das matrizes.

A seguir, apresenta-se uma explicação detalhada da implementação:

2.1. Arquivo `node.hpp`

```
struct Node {
    Node *right;
    Node *down;
    unsigned row;
    unsigned column;
    double value;

    Node(Node *right = nullptr, Node *down = nullptr, int row = 0, int
        column = 0, double value = 0) {
        this->right = right;
        this->down = down;
        this->row = row;
        this->column = column;
        this->value = value;
    }
};
```

Cada nó da matriz esparsa possui um ponteiro para o próximo nó à direita e um ponteiro para o nó abaixo, além das informações relativas à linha e à coluna que indicam sua posição na matriz. Optou-se por definir parâmetros opcionais no construtor, o que evita conflitos, já que os ponteiros são inicializados com `nullptr` e o valor com 0.

2.2. Arquivo `matrix.cpp`

Na classe `SparseMatrix`, definida no arquivo `matrix.hpp`, encontram-se os seguintes atributos:

- `rows`: indica o número de linhas da matriz esparsa;
- `columns`: indica o número de colunas da matriz esparsa;
- `root`: é um ponteiro para o nó sentinela da matriz, localizado na posição (0, 0).

Todos os nós sentinelas das linhas possuem o valor 0 no campo que indica a linha, e os nós sentinelas das colunas possuem o valor 0 no campo que indica a coluna. Dessa forma, o nó `root`, que tem ambos os valores iguais a 0, funciona como a raiz da estrutura da matriz esparsa.

A seguir, detalhamos os métodos principais da classe:

- `SparseMatrix(size_t m, size_t n);`
  O construtor da matriz esparsa recebe as dimensões (número de linhas e colunas) e verifica se os valores fornecidos são válidos (isto é, não nulos e não superiores ao limite máximo de elementos, definido no código como 30000×30000). Em seguida, cria os nós sentinelas necessários para construir a matriz. No código-fonte, o nó raiz é criado e, utilizando um ponteiro auxiliar, são gerados os demais nós sentinelas. A criação dos nós sentinelas para as linhas e para as colunas é realizada de forma separada, mas seguindo a mesma lógica: para cada nova linha ou coluna, é criado um novo nó, o ponteiro auxiliar é atualizado para apontar para esse nó, e o processo se repete até que todas as linhas e colunas estejam configuradas. A seguir, apresenta-se um pseudocódigo ilustrativo dessa implementação.

```
CONSTRUCTOR SparseMatrix(rows, columns)

root ← NEW Node(row = 0, column = 0)

currentRowSentinel ← root
FOR i FROM 1 TO rows DO
    newRowSentinel ← NEW Node(row = i, column = 0)
    newRowSentinel.right ← newRowSentinel
    currentRowSentinel.down ← newRowSentinel
    currentRowSentinel ← newRowSentinel
END FOR
```

Foram adicionados a parte onde os sentinelas apontam para ele mesmo, a fim de concluir o que foi citado anteriomente sobre a construção das matrizes esparsas.

- `void insert(unsigned i, unsigned j, double value);`
  Na inserção de valores, primeiramente verifica-se se os índices informados são válidos. Em seguida, são utilizados ponteiros auxiliares para acessar a linha e a coluna desejada, de modo a localizar o nó correspondente ou o nó imediatamente anterior à posição onde o novo valor deverá ser inserido. Caso o valor a ser inserido seja `0.00`, a função é encerrada imediatamente, pois valores nulos não são armazenados em matrizes esparsas. Em uma seção subsequente, será apresentada uma análise mais detalhada da complexidade dessa função.
- `double get(unsigned i, unsigned j);`
  Esse método retorna o valor armazenado na posição (i, j) da matriz esparsa. Se o nó correspondente não existir, retorna 0. Inicialmente, verifica-se se os índices são válidos; em seguida, utiliza-se um ponteiro auxiliar para percorrer a linha desejada. Esse ponteiro avança pelas colunas até encontrar um nó com coluna igual a 0 (indicativo do comportamento cíclico da estrutura) ou até localizar a coluna procurada. Se o nó for encontrado, seu valor é retornado; caso contrário, retorna-se 0.
- `double** toMatrix();`
  Esse método converte a matriz esparsa em uma matriz convencional (ou seja, uma matriz de ponteiros para `double`). Para isso, utiliza-se uma lógica similar à aplicada na função `print()`, mas, em vez de exibir os valores, os insere na matriz convencional. A matriz de `double` é alocada dinamicamente e, em seguida, os valores são atribuídos de acordo com seus respectivos índices.
- `string print();`
  Para exibir a matriz esparsa, optou-se por criar um fluxo de saída (stream) em vez de imprimir os valores diretamente, visando uma melhor organização. Na função `print()`, a matriz é percorrida apenas uma vez, evitando o uso repetitivo da função `get()`, que demandaria um tempo de processamento maior, uma vez que seria necessário percorrer parte da estrutura para cada índice exibido. Durante a iteração por cada linha, é criado um fluxo de saída utilizando dois ponteiros auxiliares: um que permanece fixo para representar a linha atual e outro que percorre as colunas dessa linha. Essa separação facilita o manuseio do código.
  O cálculo dos zeros entre os nós das colunas é realizado com base nos índices: por exemplo, se há um nó na coluna 1 e outro na coluna 3, o número de zeros entre eles é dado por $(3 - 1) - 1$. Assim, a fórmula para calcular os zeros entre as colunas é:

$$zeros = (próximacoluna - colunaatual) - 1.$$

Como esse cálculo não é realizado para o último nó da coluna, ao término do laço que percorre as colunas, o último nó é utilizado para calcular a quantidade de zeros restantes por meio da fórmula:

$$zeros = (númerototaldecolunas) - (índicedaúltimacoluna).$$

Dessa forma, para o nó que corresponde à última coluna, são adicionados zeros ao fluxo de saída.

2.3. Arquivo `main.cpp`

Na `main` está presente as funções e os comandos para manusear as matrizes esparsas. A seguir, apresentamos uma breve descrição das funções:

- `sum(A, B)`: Soma duas matrizes esparas A e B, e retorna uma matriz esparsa C;
- `multiplty(A, B)`: Multiplica a matriz esparsa A pela matriz esparsa B, e retorna uma matriz esparsa C;
- `createM(ifstream arc)`: Lê um arquivo '.txt' que contém as dimenções de uma matriz esparsa e suas inserções e retorna uma matriz esparsa C;
- `createC(n, m)`: Lê uma matriz n x m dada pelo usuário e retorna uma matriz esparsa C.

Todas as funções apresentadas retorna um ponteiro para uma matriz esparsa, porque na `main` é criado um `vector` de ponteiros para matrices esparsas. Utilizou-se essa escolha para dizer ao usuário se uma matriz esparsa foi criada ou não. Na explicação dos comandos a seguir criados na `main` é entendido o porquê dessa escolha.

- `--help`: Lista todos os comandos disponíveis;
- `init t`: Inicia `t` matrizes esparsas vazias com 0-indexadas
- `create s m n`: Cria a matriz esparsa no índice `s` com dimensões `m x n`;
- `insert s i j v`: Insere o valor `v` na matriz esparsa do índice `s` na posição `(i, j)`;
- `print s`: Imprime a matriz esparsa no índice `s` e pergunta ao usuário se ele quer salvar essa matriz em um '.txt';
- `get s i j`: Retorna o valor da matriz esparsa `s` na posição `(i, j)`;
- `clear s`: Limpa a matriz esparsa no índice `s` deixando ela como uma matriz esparsa vazia;
- `show`: Mostra todas as matrizes inicializadas, se elas foram criadas e suas dimenções;
- `sum s u`: Soma as matrizes no índice `s` e `u` mostrando o resultado final, perguntando ao usuário se ele quer salvar a matriz resultado, se a resposta for sim, a matriz é salva no final mas matrizes iniciadas;
- `mult s u`: Multiplica a matriz no índice `s` pela matriz no índice `u` mostrando o resultado final, perguntando ao usuário se ele quer salvar a matriz resultado, se a resposta for sim, a matriz é salva no final mas matrizes iniciadas;
- `dimension s`: Retorna as dimenções da matriz esparsa do índice `s`;
- `read s m.txt`: Lê um documento `m.txt` e coloca na matriz esparsa do índice `s`;
- `readc s m n`: Lê uma matriz de tamanho `m` x `n` digitada pelo usuário separada por espaços e coloca na matriz esparsa de índice `s`;

3. Complexidade das funções

A seguir será realizado um cálculo de complexidade do pior caso das funções `get`, `insert` e `sum`.

3.1. Função `get`

```
    Node *auxRow = root->down;
    while (auxRow->row != i)
    {
        auxRow = auxRow->down;
    }
    Node *auxNode = auxRow->right;
    while (auxNode->column != 0 && auxNode->column < j)
    {
        auxNode = auxNode->right;
    }
    return auxNode->column == j ? auxNode->value : 0.0;
```

O pior caso para pegar um valor na matriz esparsas é quando o nó existe e está na última linha e coluna. Como na matriz esparsas os nós são encadeados, para chegar nesse último nó é preciso percorrer todos os nós da linha e depois todos os nós das colunas, assim, no total temos $m + n$ operações, sendo m a quantidade de linha da matriz esparsa e n a quantidade de coluna. Assim, a complexidade da função `get` é $O(m + n)$.

3.2. Função `insert`

```
    Node *auxRow = root->down;
    while (auxRow->row != i)
    {
        auxRow = auxRow->down;
    }
    while (auxRow->right->column != 0 && auxRow->right->column <= j)
    {
        auxRow = auxRow->right;
    }
    if (auxRow->column == j)
    {
        auxRow->value = value;
        return;
    }
    Node *auxColumn = root;
    while (auxColumn->column != j)
    {
        auxColumn = auxColumn->right;
    }
    while (auxColumn->down->row != 0 && auxColumn->down->row <= i)
    {
        auxColumn = auxColumn->down;
    }
```

O pior caso para inserir um valor na matriz esparsa é quando não existe um nó para a posição requerida e, está na última linha e coluna existente. A última linha tem todos os nós das colunas existentes assim como a última coluna tem todos os nós das linhas existentes. Na função inserção, é usado dois nós auxiliares, de linha e coluna, a fim de

encontrar o nó, caso este exista, ou os nós imediatos que antecedem a posição requerida. O nó linha auxiliar percorre a linha até encontrar a linha requerida, ou seja, m linhas. Logo em seguida, ele percorre a coluna até encontrar a coluna que antecedenete, fazendo assim $n-1$ operações sendo n a quantidade de linhas. O mesmo acontece para o nó coluna auxiliar. No total é feito $n-1+m+m-1+n = 2n+2m-2 = 2(n+m-1)$ operações. Portando, a complexidade da função `insert` é $O(m+n)$.

3.3. Função `sum`

```
1    SparseMatrix *result = new SparseMatrix(A->sizeRow(), B->
         sizeColunms());
2    for (size_t i = 1; i <= A->sizeRow(); i++)
3    {
4        for (size_t j = 1; j <= B->sizeColunms(); j++)
5        {
6            result->insert(i, j, A->get(i, j) + B->get(i, j));
7        }
8    }
9    return result;
```

Para o pior caso da função `sum`, é preciso considerar que todos os nós das matrizes esparsas dadas como entradas não tenham valores vazios, ou seja, que todos os nós existam. Na função de soma, para cada posição da matriz esparsa resultado criada, é feito dois `get` na para somar os dois valores e colocar na matriz esparsa resultante, ou seja, para cada posição é feito dois `get` e um `insert`. Essa operação acontece $m * n$ vezes, sendo m e n a quantidade de linhas e colunas respectivamente. Note que cada `insert` vai sempre criar um novo nó, assim, é possível usar o resultado obtido na função de inserção anterior.

No pior caso, considere que todas as posições em `A` e `B` são não nulas (portanto existem nós nessas posições). Para cada uma das $m \times n$ posições, executamos:

- 2 chamadas a `get` (uma em cada matriz);
- 1 chamada a `insert` para adicionar o resultado na matriz `result`.

Cada `get` e `insert` custam $O(m+n)$ no pior caso, logo cada posição custa $O(m+n)$. Realizando isso para $m \times n$ posições, a complexidade total é $O(m\,n\,(m+n))$.

4. Detalhes sobre a execução do trabalho

Exploraremos nessa sessão como foi dividido o trabalho, quais foram as dificudades que a equipe teve em relação a criação da estrutura de dados Matriz Esparsa, e os testes feitos em matrizes esparsas de grandes dimenções que matrizes normais não é possível fazer.

4.1. Teste matrizes

Na pasta `testes`, há arquivos `.txt` que servem de entrada para criar matrizes esparsas e efetuar operações de soma e multiplicação. Adicionalmente, há um arquivo `.txt` com medições de tempo médio das operações de inserção e visualização (`print()`) em uma matriz `30000 × 30000`, usando diferentes computadores para demonstrar o desempenho. Arquivos `.txt` de resultados de operações entre matrizes esparsas foram criados como resultado das operações. Assim, se existe um `A.txt` e `B.txt` representando uma matrizes esparsas, arquivo de resultado gerado será `AoperacaoB.txt`, como no exemplo da soma, `A+B.txt`.

4.2. Dificuldades

- O discente João Pedro Batista, já com bastante experiência em programação, relatou dificultades na criação dos testes, devido ao grande impasse de manuseio de grandes matrizes. - O discente José Nilson Bernardo, mesmo já tendo experiências anteriores com o LaTeX, relatou que sentiu dificuldades com novas formatações. Outro impasse para o mesmo, foi em relação ao sistema de deleção de espaços revervados na memória, como acontece em ponteiros.

4.3. Divisão de atividades

- **João Pedro**: (Destrutor da `SparseMatrix`; Método `print`; Complemento da main interativa; Cores na saída das operações; Criação das funções `sum` e `mult`; Correções nos tratamentos de erros; Criação dos arquivos `.txt`; Teste da matrizes;)
- **José Nilson**: (Relatório em LaTeX; Método `insert`; Criação inicial da main interativa; Método `get`; Método `toMatrix`; Construtor da `SparseMatrix`; Criação da funções `readM` e `readC`; Tratamento de erros; Teste das matrizes; Cálculo de notação Big O)

5. Conclusão

As matrizes esparsas aqui apresentadas são úteis para armazenar dados em estruturas matriciais de grandes dimensões com poucos valores não nulos. A implementação mostrou-se eficiente, sobretudo em termos de espaço, além de permitir operações como soma e multiplicação de modo relativamente simples, desde que sejam respeitados os limites de memória e tempo.

References

Sanderson, C.; Curtin, R. Practical Sparse Matrices in C++ with Hybrid Storage and Template-Based Expression Optimisation. Math. Comput. Appl. 2019, 24, 70. https://doi.org/10.3390/mca24030070