SparseMatrix utilizando TAD'S em C++

Íkaro Freitas de Almeida

[1]Universidade Federal do Ceará - Campus Quixadá (UFC)
Quixadá – CE – Brazil
Trabalho desenvolvido para a disciplina de Estrutura de Dados
Professor / Orientador : Atílio Gomes Luiz

ikaroalmeida@alu.ufc.br

Abstract. *This meta-article describes, explains and conceptualizes Sparse Matrices, their applications, implementation difficulties in code and in general situations. It also demonstrates a functional and efficient implementation model that can be replicated and reused.*

Resumo. *Este meta-artigo descreve, explica e conceitua sobre Matrizes Esparsas, suas aplicações, dificuldades de implementação em código e em situações no geral. Também demostra um modelo de implementação, funcional e eficiente que pode ser replicado e reutilizado.*

1. Informações Gerais

Para começar, é interessante definir o que é uma matriz esparsa para o andamento desse projeto. É uma matriz na qual a grande maioria de seus elementos possui um valor padrão (por exemplo zero) ou são nulos ou faltantes. Um bom exemplo seria uma matriz que representa um contorno de uma imagem em preto e branco, seria um desperdício gastar m x n posições de memória sendo que apenas uma pequena parcela dos elementos tem valor diferente de zero.

2. Requisitos de desenvolvimento

Para a construção eficiente de uma matriz esparsa, deve-se cumprir alguns pontos importantes, como utilizar uma **Estrutura de Dados** adequada (neste projeto, foram utilizadas listas encadeadas para a implementação, porém não foram utilizadas bibliotecas prontas, a estrutura de dados foi feita manualmente durante todo o processo de criação de código), desenvolver **Operações básicas** utilizando as matrizes criadas, e manter o código **Modular** e **Reutilizável**.

3. Ferramentas utilizadas

Foram utilizadas diversas ferramentas neste projeto, desde o ambiente de desenvolvimento até a plataforma de versionamento. Com o compilador e todas as dependências instaladas, o projeto foi implementado na linguagem **C++**, optada pelo seu desempenho e facilidade de compreensão acerca da estrutura de dados utilizada.

Para a construção do software, o editor de texto escolhido foi o **Visual Studio Code**, amplamente o mais utilizado por desenvolvedores.

Também foram usados diversos livros, artigos e pesquisas para auxiliar no estudo e criação do projeto, todos devidamente apresentados na seção de referências.

4. Estruturas de Dados para Matrizes Esparsas

Prezando por eficiência e código legível, **Listas encadeadas** foram utilizadas na criação das matrizes, alguns pontos foram levados em consideração:

4.1. Alocação de memória

Diferente de outras estruturas de dados, uma lista encadeada não precisa de um tamanho fixo pré-definido. Ela cresce e reduz conforme necessário.

4.2. Inserção e Remoção Eficientes

m uma lista encadeada, inserir ou remover elementos no início ou meio da lista é eficiente (O(1) no início, O(n) na posição arbitrária), pois basta alterar os ponteiros.

4.3. Uso Eficiente de Memória para Elementos Grandes

Se os elementos da estrutura forem muito grandes e a realocação de arrays for custosa, a lista encadeada pode evitar cópias desnecessárias.

5. Implementação em C++

Deve-se apresentar a divisão de arquivos utilizada, em que a classe e seus atributos foram feitas em um arquivo de cabeçalho **SparseMatrix.h**, todos os métodos e funções-membro foram implementadas em um arquivo **SparseMatrix.cpp**. No código, é de grande importância destacar a criação o **nó** (Node), que indica a posição, endereço, e de certa forma um índice para um elemento da matriz

```
// Criando o nó
struct Node {
    Node* right;
    Node* down;
    int line;
    int colun;
    double value;
```

Figure 1. Criação do nó da lista encadeada

Ademais, é criado a classe **SparseMatrix**, que contém todos os atributos e métodos que a estrutura possui, e também um método chamado **Construtor padrão**, onde é estabelecido a maneira de criação de uma matriz esparsa vazia

```
// Criando a classe referente a matriz
class SparseMatrix {
private:
    int lines;
    int coluns;
    Node** LineHeaders;
    Node** ColunHeaders;
```

Figure 2. Criação da classe de uma Matriz Esparsa

Figure 3. Construtor da classe

```
// Construtor padrão
SparseMatrix::SparseMatrix(int m, int n) : lines(m), coluns(n) {
    if (m <= 0 || n <= 0) {
        throw invalid_argument("Inserir dimensões válidas");
    }
    LineHeaders = new Node*[m + 1];
    ColunHeaders = new Node*[n + 1];

    for (int i = 1; i <= m; ++i) {
        LineHeaders[i] = new Node(i, 0, 0.0);
        ColunHeaders[i]->right = LineHeaders[i];
    }
    for (int j = 1; j <= n; ++j) {
        ColunHeaders[j] = new Node(0, j, 0.0);
        ColunHeaders[j]->down = ColunHeaders[j];
    }
}
```

6. Testes e Validação

Obviamente, foram criados testes para validar o funcionamento da ideia, baseados em resultados obtidos por outros desenvolvedores, e do que se é conhecido sobre eficiência, código limpo e desenvolvimento.

6.1. Teste funcionais

O código foi destrinchado, para encontrar possíveis bugs, trechos repetidos, ilegíveis ou inúteis, fazendo assim a arquitetura modular e eficaz.

6.2. Teste de Inserção e Acesso a Elementos

Inserir elementos em diferentes posições (início, meio e fim) e tentar acessar elementos que não foram inseridos (deve retornar zero ou um valor padrão).

6.3. Teste de Operações Matemáticas

Verificar se os elementos comuns são somados e multiplicados corretamente e comparar o resultado com uma implementação densa ou biblioteca confiável (como Eigen).

6.4. Testes de Performance

Examinar o tempo de execução, complexidade, medir o tempo de inserção, busca e multiplicação, e analisar o pior e melhor caso possível.

Figure 4. Complexidade dos algoritmos

Função	Melhor Caso	Pior Caso
SparseMatrix(int m, int n)	$O(m+n)$	$O(m+n)$
~SparseMatrix()	$O(m+k)$	$O(m+k)$
insert(i, j, v)	$O(1)$	$O(n+m)$
getValue(i, j)	$O(1)$	$O(n)$
print()	$O(mn)$	$O(mn)$
sum(A, B)	$O(k)$	$O(mn)$
multiply(A, B)	$O(k \cdot d)$	$O(mnp)$

7. Aplicações Práticas

Essa estrutura é muito versátil, na engenharia e computação pode simular sistemas físicos, pode criar redes neurais quando usada em inteligência artificial e machine learning, pode representar grafos, processar imagens (JPEG e MPEG usam matrizes esparsas para armazenar dados), além de ter usabilidade na computação distribuída.

8. Conclusão

Ao longo deste artigo, exploramos as principais estruturas de dados utilizadas para representar matrizes esparsas, suas operações matemáticas essenciais e os desafios envolvidos em sua implementação. Além disso, analisamos a complexidade computacional das funções principais, garantindo uma abordagem eficiente para diferentes aplicações.

Com a crescente demanda por processamento de dados em larga escala, a otimização do uso de matrizes esparsas continua sendo um campo de estudo relevante. Futuras melhorias podem incluir paralelização, uso de GPUs e algoritmos mais eficientes, permitindo aplicações ainda mais rápidas e escaláveis. Assim, a compreensão e implementação correta dessas estruturas se tornam cada vez mais essenciais para o desenvolvimento de soluções computacionais avançadas.

9. Referências

Matrizes Esparsas - Universidade Federal do Maranhão (UFMA)
SCHILDT, Herb - C Completo e Total (1997)
MARTIN, Robert Cencil - Clean Code (2008)