Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Santa Helena
Bacharelado em Ciência da Computação

TRABALHO DE ESTRUTURA DE DADOS

PROF.DR: THIAGO FRANÇA NAVES

TRABALHO 1 - ENCADEAMENTO MATRIZ

Nathan S. Ribeiro Guimarães [1]

SANTA HELENA

08/2024

[1]nathan.2019@alunos.utfpr.edu.br

Uma matriz, matematicamente, é uma tabela bidimensional de números, símbolos ou expressões, organizada em linhas e colunas. Cada elemento de uma matriz é identificado por sua posição dentro da tabela, especificada por um índice de linha e um índice de coluna.

Matrizes são usadas em várias áreas da matemática, como álgebra linear, onde servem para representar transformações lineares, sistemas de equações lineares, e operações em espaços vetoriais. Elas também têm aplicações em física, engenharia e ciência da computação.

Existem diversos tipos de matriz, no entanto as que iremos abordar nesse trabalho e a Matriz Densa e a Matriz Esparsa as quais que foram propostas. Para melhorar nossa concepção vamos abordar os principais aspectos e diferenças de cada uma dessas matrizes.

Matriz Normal (Densa):

- **Armazenamento:** Em uma matriz densa, todos os elementos, incluindo os zeros, são armazenados na memória. Isso significa que, para essa matriz, os elementos são alocados, independentemente de quantos desses elementos são zeros.
- **Uso de Memória:** A memória utilizada é proporcional ao produto do número de linhas e colunas, o que pode ser ineficiente se muitos desses elementos forem zeros.
- **Operações:** As operações sobre matrizes densas são geralmente mais simples de implementar, pois a estrutura é contínua e previsível. No entanto, a execução dessas operações pode ser menos eficiente em termos de tempo e espaço se a matriz contiver muitos zeros.

Matriz Esparsa:

- **Armazenamento:** Em uma matriz esparsa, apenas os elementos não-zero são armazenados, juntamente com suas posições (índices). Isso pode ser feito de várias maneiras, como o formato CSR (*Compressed Sparse Row*) ou CSC (*Compressed Sparse Column*), que usam arrays para armazenar valores, índices de linha ou coluna, e ponteiros para indicar onde cada linha ou coluna começa.

- **Uso de Memória:** A memória utilizada é significativamente menor, especialmente quando a matriz tem uma alta proporção de zeros, pois apenas os elementos não-zero e suas posições são armazenados.
- **Operações:** As operações em matrizes esparsas são mais complexas de implementar devido à necessidade de tratar a estrutura esparsa. No entanto, essas operações podem ser muito mais eficientes em termos de tempo e espaço, especialmente para matrizes muito grandes com poucos elementos não-zero.

A seguir uma imagem que representa as Matrizes:

|1 0 0 0| |0 2 0 0| |0 0 3 0| |0 0 0 4|	| 1 2 3 4 | | 5 6 7 8 | | 9 10 11 12 | |13 14 15 16 |
Matriz Esparsa	Matriz Densa

INTRODUÇÃO

No presente trabalho, exploraremos em detalhes a estrutura e as operações das listas duplamente encadeadas, comparando-as com outras estruturas de dados e apresentando exemplos de sua aplicação com Matriz Esparsas e Matriz Densa.

MATERIAL E MÉTODOS

O desenvolvimento do código foi realizado em C, uma linguagem de programação amplamente utilizada por sua simplicidade e flexibilidade quando se trata de estrutura de dados. Para a implementação e execução do projeto, utilizou-se o Codeblocks, IDE (Ambiente de Desenvolvimento Integrado) de código aberto e gratuito, projetado principalmente para o desenvolvimento em linguagem C/C++. As funcionalidades do código foram implementadas utilizando as bibliotecas padrão da linguagem C, stdio que fornece as como as funções de entrada e saída dos dados e a stdlib que fornece funções das operações do algoritmo como alocação de memória e controle de processos. A interface do projeto foi o próprio terminal as informações foram ajustadas de modo a facilitar a visualização dos dados das matrizes bem como a interpretação do funcionamento.

Para esse trabalho vamos utilizar o conceito de listas duplamente encadeadas para construirmos uma matriz esparsa segundo Weiss (2011), A implementação de listas duplamente encadeadas envolve a criação de nós que armazenam referências tanto para o próximo quanto para o nó anterior, permitindo operações de inserção e remoção em tempo constante. Este método é particularmente útil em aplicações que requerem acesso frequente e eficiente aos elementos da lista, como editores de texto e navegadores de internet.

Os códigos disponibilizados representam uma matriz densa e uma matriz esparsa utilizando o conceito de listas duplamente encadeadas, os códigos das duas Matrizes foram disponibilizados no envio do arquivo, no entanto também podem ser consultados por meio do repositório no github: LINK.

A tabela com a representação das matrizes abordadas está disponível no seguinte link: Matriz Esparsa - Matriz Densa

CÓDIGO MATRIZ - DENSA:

Os códigos permitem, criar, inserir, remover e imprimir elementos na matriz, conforme proposto no enunciado do trabalho, o algoritmo consegue exibir os vizinhos de um elemento específico. A matriz é manipulada por meio de um conjunto de funções que garantem a alocação dinâmica da memória e a correta conexão dos elementos.

vamos exibir a saída após a execução do projeto MatrizDensa:

Mat. Densa	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

```c
#include <stdio.h>
#include "Matriz.h"

int main() {
    //Criando matriz (x, y)
    Matriz *matriz = criarMatriz(3, 3);

    //Inserindo elementos na matriz
    inserirElemento(matriz, 0, 0, 10);
    inserirElemento(matriz, 0, 1, 20);
    inserirElemento(matriz, 0, 2, 30);
    inserirElemento(matriz, 1, 0, 40);
    inserirElemento(matriz, 1, 1, 50);
    inserirElemento(matriz, 1, 2, 60);
    inserirElemento(matriz, 2, 0, 70);
    inserirElemento(matriz, 2, 1, 80);
    inserirElemento(matriz, 2, 2, 90);

    //Imprimindo a matriz
```

```
Matriz 3x3:
10 20 30
40 50 60
70 80 90

Vizinhos do elemento na posicao (1, 1):
Vizinhos do elemento na posicao (1, 1):
Acima: 20
Abaixo: 80
Esquerda: 40
Direita: 60

Removendo o elemento da posicao (1, 1)...
Matriz apos a remocao:
10 20 30
40 0 60
70 80 90

Process returned 0 (0x0)   execution time : 0.154 s
Press any key to continue.
```

Figura 01 - Resultado Matriz Densa

criarMatriz:

- Cria e inicializa uma matriz com o número especificado de linhas e colunas. Cada posição é inicialmente definida como null

```
5   // Função para criar matriz densa
6   Matriz* criarMatriz(int linhas, int colunas) {
7       Matriz *matriz = (Matriz*)malloc(sizeof(Matriz));
8       matriz->linhas = linhas;
9       matriz->colunas = colunas;
10      matriz->elementos = (No***)malloc(linhas * sizeof(No**));
11
12      for (int i = 0; i < linhas; i++) {
13          matriz->elementos[i] = (No**)malloc(colunas * sizeof(No*));
14          for (int j = 0; j < colunas; j++) {
15              matriz->elementos[i][j] = (No*)malloc(sizeof(No));
16              matriz->elementos[i][j]->valor = 0; //definido valores
17              matriz->elementos[i][j]->anterior = NULL;
18              matriz->elementos[i][j]->proximo = NULL;
19          }
20      }
```

inserirElemento:

- Insere um valor na posição especificada da matriz. Se o elemento na posição não existir, aloca memória para ele.

```
25  // Função para inserir um elemento na matriz
26  void inserirElemento(Matriz *matriz, int linha, int coluna, int valor) {
27      if (linha >= 0 && linha < matriz->linhas && coluna >= 0 && coluna < matriz->colunas) {
28          matriz->elementos[linha][coluna]->valor = valor;
29      } else {
30          printf("Posicao invalida.\n");
31      }
32  }
```

removerElemento:

- Remove o elemento na posição especificada da matriz e libera a memória alocada para ele.

```
36  //Função para remover um elemento da matriz
37  void removerElemento(Matriz *matriz, int linha, int coluna) {
38      if (linha >= 0 && linha < matriz->linhas && coluna >= 0 && coluna < matriz->colunas) {
39          matriz->elementos[linha][coluna]->valor = 0; //Definindo como 0 para representar remoção
40      } else {
41          printf("Posicao invalida.\n");
42      }
43  }
```

imprimirMatriz:

- Imprime a matriz no console. Posiciona 0 para elementos que não foram inicializados.

```
47    // Função para imprimir a matriz
48    void imprimirMatriz(Matriz *matriz) {
49        for (int i = 0; i < matriz->linhas; i++) {
50            for (int j = 0; j < matriz->colunas; j++) {
51                printf("%d ", matriz->elementos[i][j]->valor);
52            }
53            printf("\n");
54        }
55    }
```

exibirVizinhos:

- Exibe os valores dos elementos vizinhos (acima, abaixo, esquerda, direita) ao elemento na posição especificada.

```
58  //Função para exibir os vizinhos de um elemento
59  void exibirVizinhos(Matriz *matriz, int linha, int coluna) {
60      if (linha >= 0 && linha < matriz->linhas && coluna >= 0 && coluna < matriz->colunas) {
61          printf("Vizinhos do elemento na posicao (%d, %d):\n", linha, coluna);
62
63          if (linha > 0) //Elemento acima
64              printf("Acima: %d\n", matriz->elementos[linha-1][coluna]->valor);   //teste
65          if (linha < matriz->linhas - 1) //Elemento abaixo
66              printf("Abaixo: %d\n", matriz->elementos[linha+1][coluna]->valor);
67          if (coluna > 0) //Elemento à esquerda
68              printf("Esquerda: %d\n", matriz->elementos[linha][coluna-1]->valor);
69               //printf("Esquerda: %d\n", matriz->elementos[linha][coluna-2]->valor)
70          if (coluna < matriz->colunas - 1) //Elemento à direita
71              printf("Direita: %d\n", matriz->elementos[linha][coluna+1]->valor);
72
73      } else {
74          printf("Posicao invalida.\n");
75      }
76  }
```

destruirMatriz:

- Libera a memória alocada para a matriz e todos os seus elementos.

```
77  //Função para destruir a matriz e liberar memória
78  void destruirMatriz(Matriz *matriz) {
79      for (int i = 0; i < matriz->linhas; i++) {
80          for (int j = 0; j < matriz->colunas; j++) {
81              free(matriz->elementos[i][j]);
82          }
83          free(matriz->elementos[i]);
84      }
85      free(matriz->elementos);
86      free(matriz);
87  }
88
```

CÓDIGO MATRIZ - ESPARSA:

criaMatriz:

- Cria e inicia uma matriz esparsa, retornando um ponteiro para a estrutura MatrizEsparsa.

```
6   MatrizEsparsa* criaMatriz() {
7       MatrizEsparsa* matriz = (MatrizEsparsa*) malloc(sizeof(MatrizEsparsa));
8       matriz->cabeca = NULL;
9       return matriz;
10  }
```

insereElemento:

- Insere um valor em uma posição específica da matriz esparsa. Caso o valor seja zero, a inserção é ignorada.

```
12  // Função para inserir um elemento na matriz
13  void insereElemento(MatrizEsparsa* matriz, int linha, int coluna, int valor){
14      if (valor == 0)
15          return;
16
17      No* novoNo = (No*) malloc(sizeof(No));
18      novoNo->linha = linha;
19      novoNo->coluna = coluna;
20      novoNo->valor = valor;
21      novoNo->proxLinha = NULL;
22      novoNo->proxColuna = NULL;
23
24      if (matriz->cabeca == NULL){
25          matriz->cabeca = novoNo;
26          return;
27      }
28
29      No* atual = matriz->cabeca;
30      No* anterior = NULL;
31
32      while(atual != NULL && (atual->linha < linha || (atual->linha == linha && atual->coluna < coluna))){
33          anterior = atual;
34          atual = atual->proxLinha;
35      }
36
37      if(anterior == NULL){
38          novoNo->proxLinha = matriz->cabeca;
39          matriz->cabeca = novoNo;
40      } else {
41          anterior->proxLinha = novoNo;
42          novoNo->proxLinha = atual;
43      }
44  }
```

recuperaElemento:

- Recupera o valor armazenado em uma posição específica da matriz esparsa. Se o elemento não existir, retorna zero.

```
46  //Função para recuperar o valor de um elemento
47  int recuperaElemento(MatrizEsparsa* matriz, int linha, int coluna){
48      No* atual = matriz->cabeca;
49      while (atual != NULL) {      //percorrendo toda a matriz
50          if (atual->linha == linha && atual->coluna == coluna) {
51              return atual->valor;
52          }
53          atual = atual->proxLinha;
54      }
55      return 0; //Retorna 0 se o elemento não estiver na matriz
56  }
```

removeElemento:

- Remove um elemento dado uma posição na matriz esparsa, liberando a memória.

```
58  //Função para remover um elemento
59  void removeElemento(MatrizEsparsa* matriz, int linha, int coluna){
60      No* atual = matriz->cabeca;
61      No* anterior = NULL;
62
63      while (atual != NULL && (atual->linha != linha || atual->coluna != coluna)){
64          anterior = atual;
65          atual = atual->proxLinha;
66      }
67
68      if (atual == NULL)
69          return;
70
71      if (anterior == NULL){
72          matriz->cabeca = atual->proxLinha;
73      } else{
74          anterior->proxLinha = atual->proxLinha;
75      }
76      free(atual); //libera memoria
77  }
```

imprimeMatriz:

- Exibe uma representação visual da matriz esparsa no terminal, mostrando todos os elementos diferentes de vazio.

```
118 // Funcao para imprimir a matriz esparsa (apenas elementos nao nulos)
119 void imprimeMatriz(MatrizEsparsa* matriz) {
120     No* atual = matriz->cabeca;
121     while (atual != NULL) {
122         printf("Elemento na posicao (%d, %d): %d\n", atual->linha, atual->coluna, atual->valor);
123         atual = atual->proxLinha;
124     }
125 }
```

exibeVizinhos:

- Exibe os valores dos elementos vizinhos (acima, abaixo, à esquerda e à direita) dado uma posição na matriz.

```
79  // Função para exibir os elementos vizinhos (acima, direita, esquerda, abaixo)
80  void exibeVizinhos(MatrizEsparsa* matriz, int linha, int coluna){
81      No* atual = matriz->cabeca;
82      int acima = 0, abaixo = 0, direita = 0, esquerda = 0;
83
84      while (atual != NULL){
85          if (atual->linha == linha - 1 && atual->coluna == coluna){
86              acima = atual->valor;
87          }
88          if (atual->linha == linha + 1 && atual->coluna == coluna){
89              abaixo = atual->valor;
90          }
91          if (atual->linha == linha && atual->coluna == coluna - 1){
92              esquerda = atual->valor;
93          }
94          if (atual->linha == linha && atual->coluna == coluna + 1){
95              direita = atual->valor;
96          }
97          atual = atual->proxLinha;
98      }
99
100     printf("Vizinhos da posicao (%d, %d):\n", linha, coluna);
101     printf("Acima: %d\n", acima);
102     printf("Abaixo: %d\n", abaixo);
103     printf("Esquerda: %d\n", esquerda);
104     printf("Direita: %d\n", direita);
105 }
```

destroiMatriz:

- Libera toda a memória alocada para a matriz esparsa, e remove completamente a matriz alocada.

```
107 //Função para destruir a matriz esparsa e liberar a memória
108 void destroiMatriz(MatrizEsparsa* matriz){
109     No* atual = matriz->cabeca;
110     while (atual != NULL){          //percorre toda a matriz e remove
111         No* prox = atual->proxLinha;
112         free(atual);
113         atual = prox;
114     }
115     free(matriz); //liberando memoria
116 }
```

TESTES EXECUTADOS:

Foi realizada uma série de testes para essa matriz esparsa, no entanto para demonstrar o funcionamento desse código utilizarei exemplos fornecidos pelo professor no trabalho:

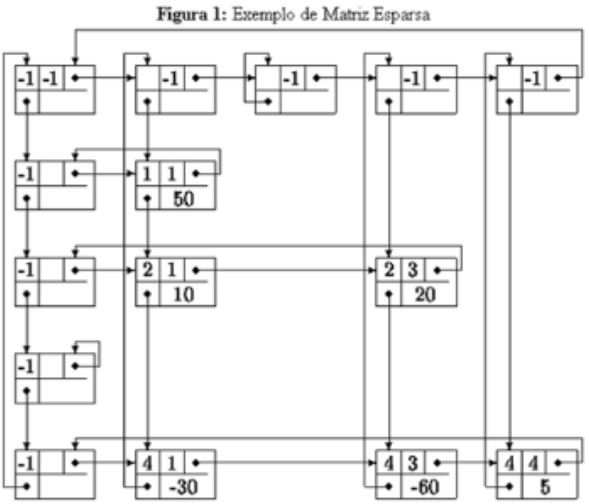

Figura 02: Exemplo Trabalho

Para melhor exibir o comportamento dessa matriz foi criado uma planilha com os respectivos valores:

	0	1	2	3	4
0	0	0	0	0	0
1	0	50	0	0	0
2	0	10	0	20	0
3	0	0	0	0	0
4	0	-30	0	-60	5

Figura 03 : Representação da Matriz Esparsa

O link para essa planilha pode ser acessado em: Matriz Esparsa

Inserimos os valores na nossa matriz diretamente no nosso código por meio da função insereElemento:

```
// dados fornecidos no exemplo do trabalho
insereElemento(matriz, 1, 1, 50);
insereElemento(matriz, 2, 1, 10);
insereElemento(matriz, 2, 3, 20);
insereElemento(matriz, 4, 1, -30);
insereElemento(matriz, 4, 3, -60);
insereElemento(matriz, 4, 4, 5);
```

Figura 04 : Inserindo Valores

Após a inserção solicitamos para exibir os respectivos elementos vizinhos de alguns elementos da seguinte forma:

```
// Exibe os vizinhos do elemento na posição (2, 1)
exibeVizinhos(matriz, 2, 1);

// Exibe os vizinhos do elemento na posição (2, 3)
exibeVizinhos(matriz, 2, 3);

// Exibe os vizinhos do elemento na posição (4, 3)
exibeVizinhos(matriz, 4, 3);
```

Figura 05 : Exibindo Vizinhos

Como resultado foi obtido:

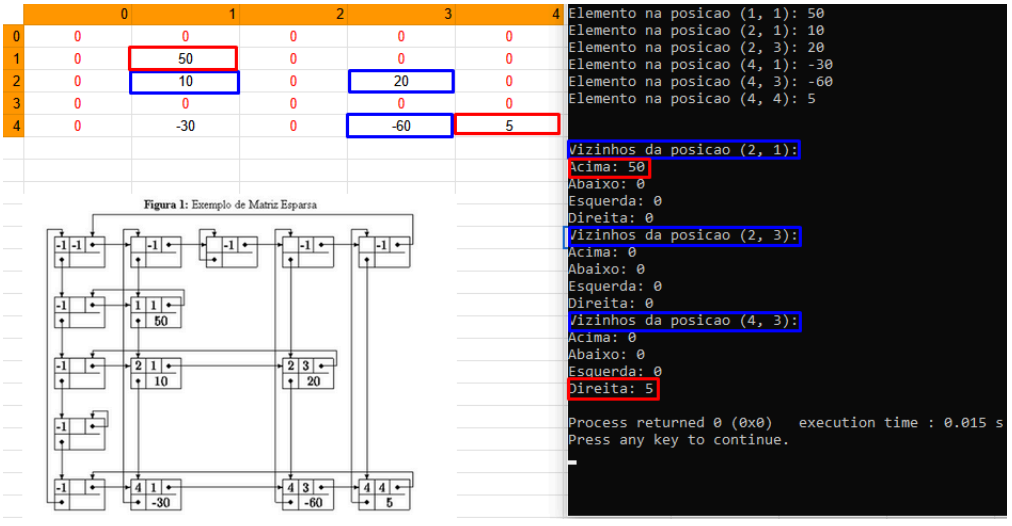

Figura 06 : Resultado da Consulta com Valores de Exemplo

Foram realizados demais testes com dimensões diferentes e o algoritmo obteve um bom desempenho:

```
// inserindo 20 elementos na matriz esparsa
insereElemento(matriz, 0, 0, 5);
insereElemento(matriz, 0, 1, 10);
insereElemento(matriz, 0, 2, 15);
insereElemento(matriz, 0, 3, 20);
insereElemento(matriz, 0, 4, 25);
insereElemento(matriz, 1, 0, 30);
insereElemento(matriz, 1, 1, 35);
insereElemento(matriz, 1, 2, 40);
insereElemento(matriz, 1, 3, 45);
insereElemento(matriz, 1, 4, 50);
insereElemento(matriz, 2, 0, 55);
insereElemento(matriz, 2, 1, 60);
insereElemento(matriz, 2, 2, 65);
insereElemento(matriz, 2, 3, 70);
insereElemento(matriz, 2, 4, 75);
insereElemento(matriz, 3, 0, 80);
insereElemento(matriz, 3, 1, 85);
insereElemento(matriz, 3, 2, 90);
insereElemento(matriz, 3, 3, 95);
insereElemento(matriz, 3, 4, 100);
insereElemento(matriz, 4, 5, 100);
```

```
Elemento na posicao (3, 4): 100
Elemento na posicao (4, 5): 100


Vizinhos da posicao (2, 1):
Acima: 35
Abaixo: 85
Esquerda: 55
Direita: 65
Vizinhos da posicao (2, 3):
Acima: 45
Abaixo: 95
Esquerda: 65
Direita: 75
Vizinhos da posicao (4, 3):
Acima: 95
Abaixo: 0
Esquerda: 0
Direita: 0

Process returned 0 (0x0)   execution time : 0.051 s
Press any key to continue.
```

Figura 07 : Resultado da Consulta com Demais Valores

RESULTADOS E DISCUSSÕES

Os dois algoritmos atingiram seus respectivos resultados dentro de suas limitações. Durante o desenvolvimento do algoritmo houve uma dificuldade inicial para remoção de um elemento da matriz sem que os demais elementos da cadeia não se perdessem. Foi possível visualizar sem muito esforço que o algoritmo de matriz esparsa usando o conceito de listas duplamente encadeadas é um algoritmo com um nível de complexidade maior se comparado ao algoritmo de matriz densa uma vez que demanda mais interações e operações maiores para percorrer a matriz e exibir os vizinhos dado uma determinada posição de um elemento, no entanto ambos são algoritmos úteis. É importante ressaltar que a escolha do algoritmo ideal depende da análise cuidadosa dos requisitos do problema, como o tamanho da matriz, a densidade de elementos não nulos e os recursos computacionais disponíveis.

REFERÊNCIAS

Estrutura de Dados- Aula 7- Listas Duplamente Encadeadas

https://www.youtube.com/watch?v=aNMM38rAEGM&list=PLRYRf6MtfBftvKW45V7qA9m71PN7jzeaE&index=8

Weiss (2011):

Weiss, M. A. (2011). *Data Structures and Algorithm Analysis in C* (3rd ed.). Pearson.