GA-024: Primeiro Trabalho Prático
Matrizes Esparsas

Antônio Tadeu A. Gomes
Laboratório Nacional de Computação Científica (LNCC/MCTI)
atagomes@posgrad.lncc.br
https://classroom.google.com/c/NDYzODk1NjA2NTU2
Sala 2C-01

Prazo: sexta-feira da semana de prova

Trabalho baseado em:
http://homepages.dcc.ufmg.br/~meira/aeds2/tp1/

1 Introdução

Matrizes esparsas são matrizes nas quais a maioria das posições são preenchidas por zeros. Para estas matrizes, podemos economizar um espaço significativo de memória se apenas os termos diferentes de zero forem armazenados. As operações matemáticas usuais sobre estas matrizes (somar, multiplicar, inverter, pivotar) também podem ser feitas em tempo muito menor se não armazenarmos as posições que contêm zeros.

O objetivo deste trabalho é implementar e testar uma forma particular de representação para armazenar e manipular as matrizes esparsas: **listas encadeadas**. Elas se contrapõem a outros formatos também usuais para representar matrizes esparsas, tipicamente baseados em conjuntos de vetores de índices e valores (como CRS, CCS etc). Esses formatos fornecem uma maneira eficiente de implementar operações matemáticas usuais sobre matrizes, mas não são tão eficientes para representar estruturas com tamanho variável e/ou desconhecido. É nesse último tipo de cenário que a representação por listas encadeadas apresenta vantagens.

1.1 Representação por listas encadeadas

Na representação por listas encadeadas, cada coluna da matriz é representada por uma lista linear circular com uma **célula cabeça**. Da mesma maneira, cada linha da matriz também é representada por uma lista linear circular com uma célula cabeça. Cada célula da estrutura, além das células-cabeça, representa os termos diferentes de zero da matriz. Um exemplo de tipo para essas células é ilustrado abaixo:

```
struct matrix {
```

```
  struct matrix* right;
  struct matrix* below;
  int line;
  int column;
  float info;
}
```

O campo `below` é usado para apontar o próximo elemento diferente de zero na mesma coluna. O campo `right` é usado para apontar o próximo elemento diferente de zero na mesma linha. Dada uma matriz A, para um valor $A(i, j)$ diferente de zero, há uma célula com o campo `info` contendo $A(i, j)$, o campo `line` contendo i e o campo `column` contendo j. Esta célula pertence à lista circular da linha i e também à lista circular da coluna j. Ou seja, cada célula pertence a duas listas ao mesmo tempo. Para diferenciar as células cabeça, valores inválidos (por exemplo, -1) podem ser usados nos campos `line` e `column` destas células.

Como exemplo, considere a matriz esparsa:

$$A = \begin{vmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{vmatrix}.$$

Uma possível representação dessa matriz usando listas encadeadas pode ser vista na Figura 1.

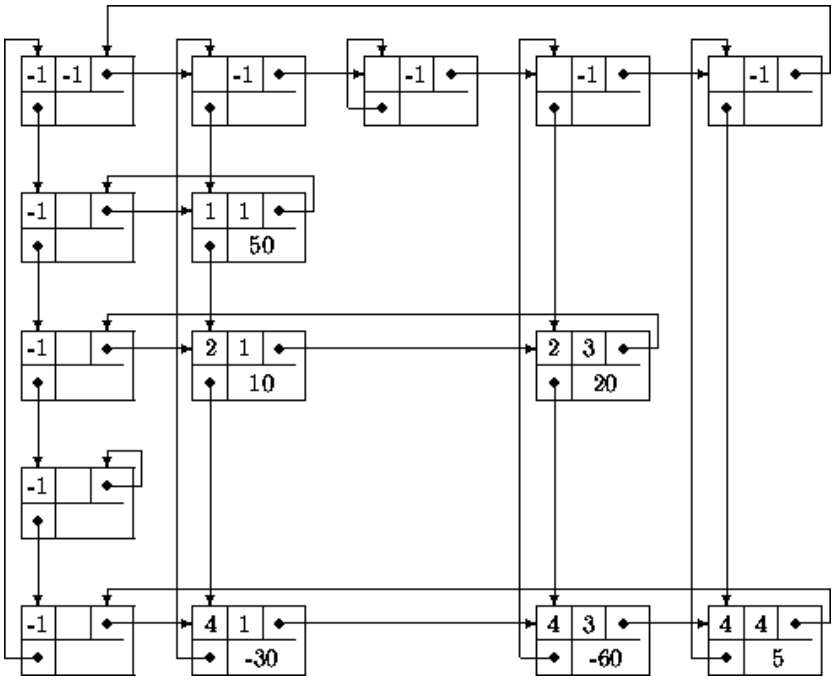

Figura 1: Representação de matriz esparsa como listas encadeadas.

Com esta representação, uma matrix esparsa $m \times n$ com r elementos diferentes de zero gastará $(m+n+r)$ células. É bem verdade que cada célula ocupa

vários bytes na memória; no entanto, o total de memória usado será menor do que as $m \times n$ posições de memória necessárias para representar a matriz toda, desde que r seja suficientemente pequeno.

2 Descrição

Dada a forma de representação acima, o trabalho consiste em desenvolver um TAD para criação e manipulação de matrizes.

O TAD deverá se chamar `Matrix` e esse identificador deverá ser usado para referenciar a forma de representação acima.

A interface em C do TAD é especificada abaixo. Toda operação deve retornar 0 em caso de sucesso e $\neq 0$ em caso de erro.

`int matrix_create( Matrix** m )`:
lê de `stdin` os elementos diferentes de zero de uma matriz e monta a estrutura especificada acima **para listas encadeadas**, retornando a matriz criada em `m`. A entrada consiste dos valores de m e n (número de linhas e de colunas da matriz) seguidos de triplas $(i, j, valor)$ para os elementos diferentes de zero da matriz. Por exemplo, para a matriz da Figura 1, a entrada seria:

```
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
0
```

Para facilitar a leitura de `stdin`, deve-se usar "0" como marcador especial de fim de matriz após a última tripla (vide exemplo acima).

`int matrix_destroy( Matrix* m )`:
devolve toda a memória alocada para a matriz `m` para a área de memória disponível (**importante:** é obrigatório neste trabalho o uso de alocação dinâmica de memória para implementar as representações das matrizes!)

`int matrix_print( const Matrix* m )`:
imprime a matriz `m` para `stdout` no mesmo formato usado por `matrix_create()`.

`int matrix_add( const Matrix* m, const Matrix* n, Matrix** r )`:
recebe como parâmetros as matrizes `m` e `n`, retornando em `r` a soma das mesmas (a estrutura da matriz retornada deve ser alocada dinamicamente pela própria operação).

`int matrix_multiply( const Matrix* m, const Matrix* n, Matrix** r )`:
recebe como parâmetros as matrizes `m` e `n`, retornando em `r` a multiplicação das mesmas (a estrutura da matriz retornada deve ser alocada dinamicamente pela própria operação).

`int matrix_transpose( const Matrix* m, Matrix** r )`:
recebe como parâmetro a matriz `m`, retornando em `r` a matriz m^T – a

transposta de `m` (a estrutura da matriz retornada deve ser alocada dinamicamente pela própria operação).

`int matrix_getelem( const Matrix* m, int x, int y, float *elem )`: retorna o valor do elemento (x, y) da matriz `m` em `elem`.

`int matrix_setelem( Matrix* m, int x, int y, float elem )`: atribui ao elemento (x, y) da matriz `m` o valor `elem`.

3 Resolução e Testes

A resolução do trabalho deve ser feita **individualmente**. Para o trabalho ser considerado como completo, deve ser entregue via Google Classroom, **até o prazo informado no início deste documento**:

1. Arquivo-fonte do módulo em C (`matrix.c`).

A resolução deve ser testada, **minimamente**, com o programa de teste a seguir, usando os pares de matrizes A e B abaixo como entradas:

$$A = \begin{vmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{vmatrix}, B = \begin{vmatrix} 50 & 30 & 0 & 0 \\ 10 & 0 & -20 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{vmatrix}; \text{ e}$$

$$A = \begin{vmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 10.5 & 0 & 0 & 0 \\ 0 & 0 & 0.015 & 0 & 0 \\ 0 & 250.5 & 0 & -280 & 33.32 \\ 0 & 0 & 0 & 0 & 12 \end{vmatrix}, B = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 10.5 & 0 & 0 \\ 0 & 0 & 0.3 & 0 \\ 0 & 100 & 0 & 30 \\ 0 & 0 & 0 & 1 \end{vmatrix}.$$

Programa de teste:

```
#include <stdio.h>
#include "matrix.h"

int main( void ) {
  /* Inicializacao da aplicacao ... */

  Matrix *A=NULL;
  Matrix *B=NULL;
  Matrix *C=NULL;

  if( !matrix_create( &A ) )
    matrix_print( A );
  else {
    fprintf( stderr, "Erro na alocacao de A como listas encadeadas.\n" );
    return 1;
  }

  if( !matrix_create( &B ) )
```

```
    matrix_print( B );
  else {
    fprintf( stderr, "Erro na alocacao de B como listas encadeadas.\n" );
    return 1;
  }

  if ( !matrix_add( A, B, &C ) ) {
    matrix_print( C );
  }
  else
    fprintf( stderr, "Erro na soma C=A+B.\n" );
  matrix_destroy( C );

  if ( !matrix_multiply( A, B, &C ) )
    matrix_print( C );
  else
    fprintf( stderr, "Erro na multiplicacao C=A*B.\n" );
  matrix_destroy( C );

  if ( !matrix_transpose( A, &C ) )
    matrix_print( C );
  else
    fprintf( stderr, "Erro na transposicao C=A^T.\n" );
  matrix_destroy( C );

  matrix_destroy( A );
  matrix_destroy( B );

  return 0;
}
```

É importante destacar que os testes acima representam um **conjunto mínimo**, servindo de guia para a resolução; outros testes adicionais podem ser efetuados pelo professor.