GA-024: Primeiro Trabalho Prático
Matrizes Esparsas

Antônio Tadeu A. Gomes
Laboratório Nacional de Computação Científica (LNCC/MCT)
atagomes@gmail.com
http://martin.lncc.br
Sala 2C-01

March 31, 2008

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

1 Descrição

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 usuais sobre estas matrizes (somar, multiplicar, inverter, pivotar) também podem ser feitas em tempo muito menor se não armazenarmos as posições que contêm zeros. Uma maneira eficiente de representar estruturas com tamanho variável e/ou desconhecido é através de alocação encadeada, utilizando listas. O objetivo deste trabalho é usar esta representação para armazenar e manipular as matrizes esparsas. Cada coluna da matriz será representada por uma lista linear circular com uma **célula cabeça**. Da mesma maneira, cada linha da matriz também será representada por uma lista linear circular com uma célula cabeça. Cada célula da estrutura, além das células-cabeça, representará os termos diferentes de zero da matriz e devem ter o seguinte tipo:

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

O campo `below` deve ser usado para apontar o próximo elemento diferente de zero na mesma coluna. O campo `right` deve ser 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, deverá haver uma célula com o campo `info` contendo

$A(i, j)$, o campo `line` contendo i e o campo `column` contendo j. Esta célula deverá pertencer à lista circular da linha i e também deverá pertencer à lista circular da coluna j. Ou seja, cada célula pertencerá a duas listas ao mesmo tempo. Para diferenciar as células cabeça, coloque -1 nos campos `line` e `column` destas células. **Importante:** É obrigatório o uso de alocação dinâmica de memória para implementar as listas de adjacência que representam as matrizes!

Considere a matriz esparsa A ilustrada na Figura 1.

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

Figura 1: Matriz esparsa.

A representação dessa matriz pode ser vista na Figura 2.

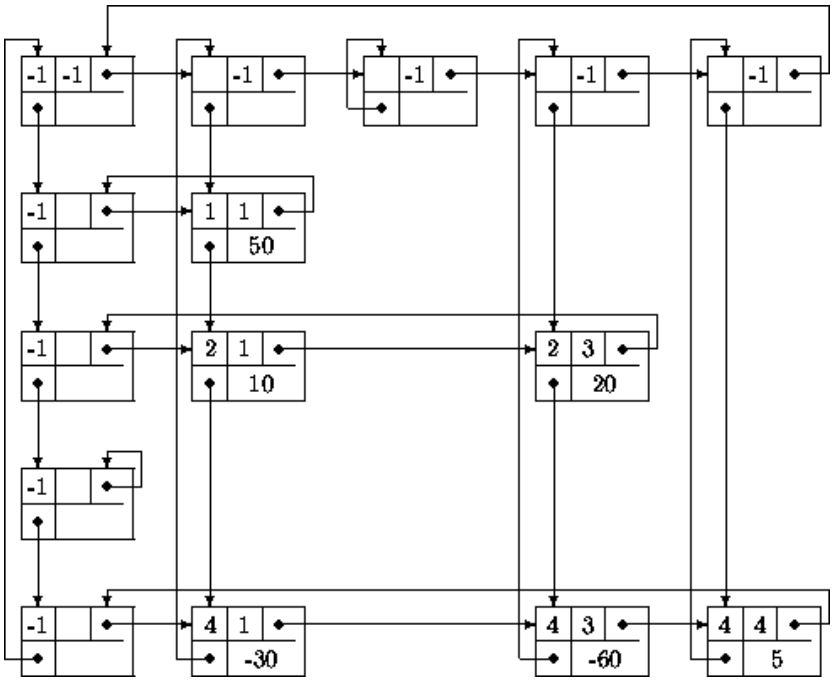

Figura 2: Representação de matriz esparsa como listas encadeadas.

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 necessárias para representar a matriz toda, desde que r seja suficientemente pequeno.

Dada a representação acima, o trabalho consiste em desenvolver cinco operações em C, conforme especificação abaixo:

`Matrix* matrix_create( void )`: lê de `stdin` os elementos diferentes de zero de uma matriz e monta a estrutura especificada acima. 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 conforme ilustrado na Figura 3:

```
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
```

Figura 3: Formato de entrada.

`void matrix_destroy( Matrix* m )`: devolve todas as células da matriz m para a área de memória disponível.

`void matrix_print( Matrix* m )`: imprime a matriz m para `stdout` no formato da Figura 3.

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

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

Dica: considere na sua resolução a implementação de duas operações auxiliares, para leitura e escrita de células individuais da matriz esparsa:

`float matrix_getelem( Matrix* m, int x, int y )` e

`void matrix_setelem( Matrix* m, int x, int y, float elem )`

2 Resolução e Testes

A resolução do trabalho pode ser feita **individualmente** ou **em dupla**. Para o trabalho ser considerado como completo, deverão ser apresentadas:

1. Listagem do programa em C.
2. Listagem dos testes executados.
3. Demonstração da resolução junto ao professor.

A resolução deve ser testada com o seguinte programa:

```
#include "matrix.h"

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

  Matrix *A = matrix_create(); matrix_print( A );
  Matrix *B = matrix_create(); matrix_print( B );
  Matrix *C = matrix_add( A, B ); matrix_print( C );
  matrix_destroy( C );
  C = matrix_multiply( A, B ); matrix_print( C );
  matrix_destroy( C );
  matrix_destroy( A );
  matrix_destroy( B );

  return 0;
}
```

usando as matrizes da Figura 4 como exemplo.

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix} \begin{pmatrix} 50 & 30 & 0 & 0 \\ 10 & 0 & -20 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{pmatrix}$$

Figura 4: Matrizes a serem usadas na resolução.