

**UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO**

**LABORATÓRIO ESTRUTURA DE DADOS I**

ATIVIDADE ACADÊMICA

Relatório

Gabriel Filippo Chagas de Andrade

Julho de 2017

**Sumário**

Introdução.................................................................................................. 3

Acerca do Algoritmo................................................................................... 4

List.h......................................................................................... 4

Sparse.h................................................................................... 9

Acerca dos Testes..................................................................................... 20

Considerações Finais................................................................................ 21

**Introdução**

◦ Motivação inicial

Este trabalho viabiliza observar a implementação de operações com matrizes esparsas afim de mostrar seu desempenho quando não gastamos espaço para armazenar elementos repetidos, no caso de matrizes esparsas números 0.

*Exemplo de matriz esparsa: Uma matriz 800 x 900, possui 720.000 elementos e apenas 10 elementos não nulos, ou seja, essa matriz possui 719.990 elementos iguais a zero.*

Ao processar todos estes elementos iguais a zero haveria um consumo enorme de memória, e também um aumento muito grande do tempo de operações básicas como soma e multiplicação.

Será apresentado uma forma de manipular essas matrizes melhorando o seu desempenho em todos esses aspectos.

◦ Acerca dos Testes

Para cálculos de tempo e memória consumida, serão utilizadas duas matrizes 900x900 com 10 elementos não nulos cada, tanto para multiplicação quanto para soma.

Obtenção dos dados através da execução *command time -v ./prog* onde “*prog”*  seria o programa

**Acerca do Algoritmo**

◦ List.h

▪ Estrutura:

***struct******list***

*{*

***long******long******int*** *value;*

***long******long******unsigned*** *row, column;*

***struct******list****\* next;*

*};*

***typedef******struct******list******List****;*

Esta é uma estrutura similar à de lista encadeada, porém adaptada para o problema de Matrizes Esparsas onde se tem um valor, sua posição dada por linha e coluna, e uma referência para o próximo elemento da lista.

▪ Funções:

**List**\* **create\_list**()

{

**return** NULL;

}

Esta função cria inicialmente uma Lista Encadeada adaptada para Matrizes Esparsas inicialmente vazia, como em Lista Encadeada normal.

**void** **free\_list**(**List**\* list)

{

**List**\* temp = list;

**while** (temp != NULL)

{

**List**\* p = temp->next;

**free**(temp);

temp = p;

}

**return**;

}

Esta função é responsável por desalocar toda a memória utilizada para criação de uma Lista Encadeada adaptada para Matrizes Esparsas, onde inicialmente temos uma variável do tipo **List**\* que percorre a lista guardando a referência para o próximo termo em p que também é do tipo **List**\*, desaloca-se temp, e temp passa agora a referenciar o mesmo que p. Assim se repete sucessivamente até liberar totalmente a lista que é quando a variável que percorre a lista se torna igual a NULL.

**List**\* **insert\_ordened**(**long** **long** **int** val, **long** **long** **unsigned** row, **long** **long** **unsigned** column, **List**\* list)

{

**List**\* element;

element = (**List**\*) **malloc** (**sizeof** (**List**));

element->value = val;

element->row = row;

element->column = column;

**List**\* temp = list;

**List**\* prev = NULL;

/\*Procura pelo primeiro elemento que vem após o de inserção,

(linha maior, ou mesma linha e coluna maior) para inserí-lo antes\*/

**while** (temp != NULL && (temp->row < row || (temp->row == row && temp->column < column)))

{

prev = temp;

temp = temp->next;

}

//O elemento a ser inserido é anterior a todos

**if** (prev == NULL)

{

element->next = list;

list = element;

}

//O elemento não deve ser inserido no início

**else**

{

element->next = prev->next;

prev->next = element;

}

**return** list;

}

Esta função é responsável por inserir um elemento na lista de forma ordenada onde os elementos são dispostos na ordem de leitura de uma matriz, onde todos os elementos com linha inferior a um dado elemento estão antes deste, se por ventura algum elemento estiver na mesma linha, ele estará antes se o seu número de colunas for inferior. Lembrando que se não há elementos entre um elemento da lista e seu sucessor e há espaços na matriz entre eles, estes elementos são nulos.

A função recebe como parâmetro o valor a ser inserido e sua posição na matriz. Cria-se inicialmente um elemento do tipo **List**\* que recebe o valor a ser inserido e sua posição. Logo após cria-se dois elementos auxiliares do tipo **List**\* que percorrerão a lista procurando por um elemento que venha imediatamente após o elemento de inserção (De acordo com a forma de ordenação) e o elemento imediatamente anterior a este para fazer o encadeamento, e este só para de percorrer se chegar ao fim da lista (quando temp for igual a NULL), no fim retorna-se a lista modificada.

Há 2 casos de inserção onde:

- A lista está vazia ou o elemento deve ser encadeado antes de todos (quando prev = NULL).

- O elemento deve ser encadeado em qualquer das outras posições (quando prev != NULL), neste caso específico, o próximo elemento do elemento anterior deve ser o novo elemento, e o próximo deste elemento deve ser o próximo do elemento anterior.

**List**\* **remove\_from\_list**(**long** **long** **int** val, **List**\* list)

{

**if** (**is\_empty\_list**(list))

**return** list;

**List**\* temp = list;

**List**\* prev = NULL;

//percorre a lista até chegar ao elemento

**while** (temp != NULL && val != temp->value)

{

prev = temp;

temp = temp->next;

}

//O elemento é o primeiro

**if** (prev == NULL)

{

list = temp->next;

**free** (temp);

}

**else** **if** (temp != NULL)

{

prev->next = temp->next;

**free**(temp);

}

**return** list;

}

Esta função funciona de forma similar à de inserção, percorrendo a lista da mesma forma guardando o elemento anterior ao requerido da lista buscando algum elemento cujo o valor é igual ao passado como parâmetro, no fim retorna-se a lista modificada.

Há 4 casos de remoção:

- A lista está vazia (função a ser visto posteriormente), nesse caso retorna-se a própria lista.

- Percorreu-se a lista e o elemento anterior é igual a NULL, nesse caso o elemento a ser removido é o primeiro da lista, onde modifica-se a lista para referenciar o próximo elemento do primeiro da lista, em seguida desaloca-se o primeiro elemento.

- Percorreu-se a lista e o elemento a ser removido está em qualquer outra posição pois temp é diferente de NULL. Nesse caso refaz-se o encadeamento, fazendo com que o próximo elemento do anterior aponte para o próximo elemento do qual será removido, e em seguida remove-se o elemento.

- Percorreu-se toda a lista e temp é igual a NULL, o que significa que o elemento não foi encontrado, retornando a própria lista.

**int** **is\_empty\_list**(**List**\* list)

{

**return** (list == NULL);

}

Esta função retorna 1 caso a lista esteja vazia (Equivale-se a NULL), e 0 caso contrário.

**void** **print\_list** (**List**\* list)

{

**if** (**is\_empty\_list**(list))

{

**printf**("Lista vazia\n");

**return**;

}

**List**\* temp;

**printf**("Imprimindo Valores...\n\n");

**for** (temp = list; temp != NULL; temp = temp->next)

{

**printf**("\t%lld[%lld][%lld]\n", temp->value, temp->row, temp->column);

}

**return**;

}

Esta função é responsável por imprimir todos os valores da lista, utiliza uma variável do tipo **List**\* para percorrer a lista e imprimindo o seu valor e sua posição.

**int** **search\_list**(**long** **long** **int** val, **List**\* list)

{

**List**\* temp;

**for** (temp = list; temp != NULL; temp = temp->next)

{

**if** (val == temp->value)

{

**return** 1;

}

}

**return** 0;

}

Esta função é responsável por retornar 1 se existe um tal valor na matriz, e 0 caso contrário. Percorre-se a lista de forma já vista até achar ou não o elemento.

**List**\* **retrieve\_element**(**long** **long** **unsigned** row, **long** **long** **unsigned** column, **List**\* list)

{

**List**\* temp = list;

**while** (temp != NULL)

{

**if** (temp->column == column && temp->row == row)

**return** temp;

/\*Os elementos estão ordenados por linha e coluna,

se achado um elemento maior (que vem depois) na matriz,

então ele é zero pois não está listado\*/

**else** **if** (temp->row > row || (temp->row == row && temp->column > column))

**return** NULL;

temp = temp->next;

}

//Se não achou

**return** NULL;

}

Esta função recebe como parâmetro uma lista, e a posição do elemento, e retorna o ponteiro para o elemento que se encontra nesta posição. Percorre-se a lista como já visto anteriormente e se achado o elemento, o retorna. Se por ventura, o elemento a ser verificado durante o percorrer da lista está situado depois da posição requerida na matriz, retorna-se NULL pois o elemento não está na lista, e assim assume-se que ele é zero (tirando proveito da ordenação).

◦ Sparse.h

▪ Estrutura:

**struct** **matrix**

{

**List**\* values;

**long** **long** **unsigned** height;

**long** **long** **unsigned** width;

};

**typedef** **struct** **matrix** **Matrix**;

Esta é uma estrutura criada para definir uma Matriz Esparsa como um todo, tendo uma referência do tipo **List**\* para guardar todos os elementos não nulos, e duas variáveis para guardar seu tamanho.

▪ Funções:

**Matrix**\* **create\_sparseMatrix**(**long** **long** **unsigned** rows, **long** **long** **unsigned** columns)

{

**Matrix**\* matrix = (**Matrix**\*) **malloc** (**sizeof**(**Matrix**));

matrix->values = **create\_list**();

matrix->height = rows;

matrix->width = columns;

**return** matrix;

}

Esta função cria e retorna uma Matriz Esparsa, alocando o espaço necessário para sua estrutura, guardando nela o seu tamanho (Tamanho de matriz) passado como parâmetro e criando uma lista inicialmente vazia para guardar elementos não nulos (visto anteriormente).

**void** **destroy\_sparseMatrix**(**Matrix**\* matrix)

{

**free\_list**(matrix->values);

**free**(matrix);

**return**;

}

Esta função é responsável por desalocar toda memória utilizada para a Matriz criada, primeiro utiliza-se a função vista anteriormente para desalocar a lista de elementos não nulos contida na matriz, e após desaloca-se a matriz.

**void** **removeFrom\_Position**(**long** **long** **unsigned** row, **long** **long** **unsigned** column, **Matrix**\* matrix)

{

**if** (row > matrix->height || column > matrix->width)

{

**printf**("\t\*\*\*Abortando: \*Função removeFrom\_Position\* Tentando acessar valor fora dos limites da matriz\n\n");

**exit**(1);

}

**List**\* temp = matrix->values;

**List**\* prev = NULL;

//Percorre a lista até chegar na posição

**while** (temp != NULL && row != temp->row && column != temp->column)

{

prev = temp;

temp = temp->next;

}

//Remoção no início da lista

**if** (prev == NULL)

{

matrix->values = temp->next;

**free** (temp);

}

//Remoção no fim da lista

**else** **if** (temp != NULL)

{

prev->next = temp->next;

**free**(temp);

}

**return**;

}

Esta função é responsável por “remover” um elemento de uma determinada posição da matriz passado como parâmetro, mas na verdade é uma função que remove da lista, pois esse elemento passa a ter valor 0 na matriz. A função retorna um erro se por acaso tenta remover um elemento que não é alcançável dentro da matriz, e é muito similar à função de remoção da lista visto anteriormente, com exceção de que ao invés de remover buscando por um valor, a remoção é feita buscando uma posição na matriz.

**void** **checkFor\_nullElements**(**Matrix**\* matrix)

{

**List**\* temp = matrix->values;

**List**\* element;

**while** (temp != NULL)

{

**if** (temp->value == 0)

{

element = temp;

**removeFrom\_Position**(element->row, element->column, matrix);

}

temp = temp->next;

}

**return**;

}

Esta função é responsável por checar se por acaso existe algum elemento na lista de elementos não nulos de uma matriz que equivale-se a zero, pois este não deveria estar lá uma vez que após alguma operação de soma pode restar algum elemento nulo. A função percorre a lista de elementos não nulos da matriz, e se acha um elemento que tenha valor zero, guarda a referência para o próximo elemento da lista e utiliza a função de remover por posição vista anteriormente, e em seguida retoma a referência guardada e continua o processo até ter percorrido toda a lista.

**int** **pickValue**(**Matrix**\* matrix, **long** **long** **unsigned** row, **long** **long** **unsigned** column)

{

**if** (row > matrix->height || column > matrix->width)

{

**printf**("\t\*\*\*Abortando: \*Função pickValue\* Tentando acessar valor fora dos limites da matriz\n\n");

**exit**(1);

}

**List**\* temp = matrix->values;

**while** (temp != NULL)

{

**if** (temp->column == column && temp->row == row)

**return** temp->value;

//Os elementos estão ordenados por linha e coluna

//se achado um elemento maior (que vem depois) na matriz, então ele é zero

**else** **if** (temp->row > row || (temp->row == row && temp->column > column))

**return** 0;

temp=temp->next;

}

//se não achou, assume-se que o elemento é zero

**return** 0;

}

Esta função é responsável por retornar o valor de uma posição qualquer da matriz, tendo como parâmetro uma dada posição na matriz.

Retorna erro caso a posição seja inalcançável dentro da matriz. A função tira proveito da ordenação da matriz, percorrendo-a de forma já vista e retornando o valor se achado, caso não tenha achado valor algum ou o elemento verificado no momento está situado após o elemento desejado significa que o elemento buscado é zero.

**void** **insertBy\_Position**(**long** **long** **unsigned** value, **long** **long** **unsigned** row, **long** **long** **unsigned** column, **Matrix**\* matrix)

{

**char** answer;

**if** (row > matrix->height || column > matrix->width)

{

**printf**("\t\*\*\*Abortando: \*Função insertBy\_Position\* Tentando acessar valor fora dos limites da matriz\n\n");

**exit**(1);

}

/\*Caso já exista um elemento nesta posição

é verificado se o usuário deseja sobrescrever ou

ignora a inserção\*/

**if** (**pickValue**(matrix, row, column) != 0)

{

**fflush**(stdout);

**printf**("Já existe um elemento não nulo na posição [%llu][%llu]\n", row, column);

**printf**("Sobrescrever? (s = sim, n = pular inserção)\n\n");

**fflush**(stdout);

**scanf**("\n%c", &answer);

//Se for desejado não sobrescrever, ignora-se a inserção

**if** (answer == 'n' || answer == 'N')

**return**;

/\*Se for desejado sobrescrever, remove-se o elemento existente para adicionar o desejado\*/

**else**

**removeFrom\_Position**(row, column, matrix);

}

//Só insere se o valor a ser inserido é diferente de zero

**if** (value != 0)

matrix->values = **insert\_ordened**(value, row, column, matrix->values);

**return**;

}

Esta função insere um número em uma determinada posição e recebe como parâmetro o valor a ser inserido, e sua posição na matriz. Retorna-se erro se for tentado inserir um elemento fora do alcance da matriz.

Primeiramente a função verifica se já existe algum elemento naquela posição da matriz, utilizando uma função que retorna o valor daquela posição já visto anteriormente, se o número é igual a zero então pode ser inserido normalmente sem problemas (só insere de fato se o elemento a ser inserido difere de zero, caso o contrário ele já estará inserido na matriz de forma subentendida) caso o contrário há uma interação com o usuário para saber se ele deseja sobrescrever aquele elemento da matriz. Se o usuário deseja sobrescrever (digitando s ou S), primeiramente utiliza-se a função de remoção (que torna o elemento igual a zero e remove da lista de elementos não-nulos) já visto anteriormente e procede com a inserção na lista de forma ordenada também já visto anteriormente. Caso contrário a inserção é ignorada pois o usuário optou por não sobrescrever (digitando n ou N).

**void** **createInput\_sparseMatrix**(**Matrix**\* matrix)

{

**long** **long** **unsigned** elements;

**long** **long** **unsigned** temp, row, column;

**char** answer;

//Verifica com o usuário a quantidade de elementos não nulos desejado

**do**

{

**printf**("Entre com a quantidade de elmentos não entre 0 e %Lu\n\n", matrix->height\*matrix->width);

**fflush**(stdout);

**scanf**("%llu", &elements);

}

**while** (elements < 0 || elements > matrix->height\*matrix->width );

//Verifica com o usuário o valor a ser inserido e sua posição para cada elemento

**for** (**long** **long** **unsigned** i = 0; i < elements; i ++)

{

**fflush**(stdout);

**printf**("Entre com o %lluº elemento e suas posições\n\n", i + 1);

**fflush**(stdout);

**scanf**("%llu %llu %llu", &temp, &row, &column);

/\*Caso já exista um elemento nesta posição

é verificado se o usuário deseja sobrescrever ou

ignora a inserção\*/

**if** (**pickValue**(matrix, row, column) != 0)

{

**fflush**(stdout);

**printf**("Já existe um elemento não nulo na posição [%llu][%llu]\n", row, column);

**printf**("Sobrescrever? (s = sim, n = pular inserção)\n\n");

**fflush**(stdout);

**scanf**("\n%c", &answer);

//Se for desejado não sobrescrever, ignora-se a inserção

**if** (answer == 'n' || answer == 'N')

**continue**;

/\*Se for desejado sobrescrever, remove-se o elemento existente para adicionar o desejado\*/

**else**

**removeFrom\_Position**(row, column, matrix);

}

**if** (temp != 0)

matrix->values = **insert\_ordened**(temp, row, column, matrix->values);

}

**return**;

}

Esta função interage com o usuário para inserir uma quantidade de elementos não nulos determinado pelo usuário. Recebe do usuário um número de elementos que deve estar obrigatoriamente entre 0 e a altura\*largura da matriz (que é o máximo de elementos que esta pode ter). Para cada elemento do número de elementos que o usuário entrou, a função funciona igual a de inserção por posição visto anteriormente, com a exceção apenas de que se o usuário não deseja sobrescrever o número, ele ignora a inserção imediatamente através do **continue** e incrementa o contador de elementos (que vai até a quantidade de elementos que o usuário deseja). Repete-se até completar a quantidade de elementos que o usuário deseja, lembrando que se o usuário desejou inserir um 0 nada é inserido na lista pois ele já está na lista de forma subentendida

**void** **createAleatory\_sparsematrix**(**long** **long** **unsigned** elements, **Matrix**\* matrix)

{

**if** (elements > matrix->height\*matrix->width)

{

**printf**("\t\*\*\*Abortando: Tentando criar uma matriz aleatória com mais elementos do que ela pode ter\n\n");

**exit**(1);

}

**srand**(**time**(NULL));

**long** **long** **unsigned** row, column, val;

**for** (**long** **long** **unsigned** i = 0; i < elements; i ++)

{

//Gera aleatoriamente valores até 100, linhas e colunas

row = **rand**()%matrix->height;

column = **rand**()%matrix->width;

val = **rand**()%100;

**while** (val == 0)

val = **rand**()%100;

//Checa se não existe elemento não nulo

**if** (**pickValue**(matrix, row, column) == 0)

matrix->values = **insert\_ordened**(val, row, column, matrix->values);

//Reinicia o processo de inserção novamente

**else**

i--;

}

**return;**

}

Esta função é uma alternativa de gerar uma Matriz Esparsa, porém de forma aleatória onde é passado por parâmetro a quantidade de elementos não nulos desejado. A função aborta o programa caso deseja-se mais elementos do que a matriz pode comportar.

É gerado aleatoriamente uma linha, uma coluna e um valor através da função **rand**() da biblioteca *time.h* que gera números entre 0 e o desejado, caso seja gerado algum valor 0 repete-se o processo de gerar aleatoriamente até este não ser nulo.

Checa se existe algum elemento não nulo nessa posição através da função que retorna um valor de uma posição já visto anteriormente, caso não exista o elemento é inserido de forma ordenada na lista (já visto anteriormente), caso contrário decrementa-se o contador (que vai até a quantidade desejada de elementos) e reinicia o processo.

**void** **print\_sparseMatrix**(**Matrix**\* matrix)

{

**printf**("Dimemnsão da matriz %Lu x %Lu\n\n", matrix->height, matrix->width);

**List**\* temp = matrix->values;

**int** i = 0;

//Percorre todos os elementos da matriz

**while** (i < matrix->width \* matrix->height)

{

//Se o elemento da vez é não nulo, o imprime

**if** (temp != NULL && i == temp->row \* matrix->width + temp->column)

{

//Se o elemento a ser impresso está no início de uma linha

**if** (temp->column == 0)

**printf**("\n\t%lld ", temp->value);

**else**

**printf**("\t%lld ", temp->value);

temp = temp->next;

}

**else**

{

//Se o elemento a ser impresso está no início de uma linha

**if** (i % matrix->width == 0)

**printf**("\n\t0 ");

**else**

**printf**("\t0 ");

}

i++;

}

**printf**("\n\n");

**return**;

}

Esta função é uma alternativa imprimir uma Matriz Esparsa de forma inteira imprimindo elemento a elemento através de um contador.

Inicia-se uma variável do tipo **List**\* que contém os elementos não nulos do vetor e inicia-se o contador. Se a variável aponta pra um elemento diferente de NULL da lista e sua linha e coluna correspondem ao i-ésimo elemento da matriz (o estado atual do contador) ele pode ser impresso normalmente sem quebra de linha se ele não é o primeiro elemento da linha, caso contrário o imprime com quebra de linha, e em ambos os casos a variável passa a apontar para o próximo elemento da lista dando continuidade ao processo até o contador chegar na quantidade de elementos da matriz. Se por acaso a variável aponta para NULL e não foram impressos todos os elementos da lista ou sua linha e coluna não correspondem ao i-ésimo elemento da matriz (o estado atual do contador) significa que o elemento nesta posição é zero, e se este i-ésimo elemento está no início de uma linha, imprime-se zero com quebra de linha, caso contrário imprime-se zero sem quebra de linha.

**void** **printCompact\_sparseMatrix**(**Matrix**\* matrix)

{

**printf**("Dimemnsão da matriz %Lu x %Lu\n\n", matrix->height, matrix->width);

**if**(**is\_empty\_list**(matrix->values))

{

**printf**("\tMatriz totalmente nula\n\n");

**return**;

}

**print\_list**(matrix->values);

**printf**("\tO resto dos elementos são nulos\n\n");

**return**;

}

Esta função é uma alternativa imprimir uma Matriz Esparsa de forma compacta e fácil de visualizar imprimindo apenas os elementos não nulo e suas posições. Se a matriz possui lista de elementos não nulos vazia, é impresso que a matriz é nula. Caso contrário imprime-se os elementos não nulos e suas posições através da função de imprimir a lista já visto anteriormente e imprime que o resto dos elementos são nulos.

**Matrix**\* **mult\_sparseMatrix**(**Matrix**\* a, **Matrix**\* b)

{

//Verifica se pode ser efetuada a multiplicação

**if** (a->width != b->height)

{

**printf**("\t\*\*\*Abortando: \*Função mult\_sparseMatrix\* Número de linhas da matriz A é diferente da B\n\n");

**exit**(1);

}

//Cria a matriz resposta

**Matrix**\* destiny = **create\_sparseMatrix**(a->height, a->width);

**List**\* temp\_a = a->values;

**List**\* temp\_b;

/\*A partir do elemento não nulo A, procura-se

os elementos de B que correspondem ao elemento

em A(número da coluna de A = número da linha de B).

Soma-se o elemento na matriz resposta

(posição[linha de A][Coluna de B]) com a multiplicação

dos correspondentes\*/

**while** (temp\_a != NULL)

{

temp\_b = b->values;

**while**(temp\_b != NULL)

{

**if** (temp\_a->column == temp\_b->row)

{

//Se o elemento já é não-nulo, soma-se

**if** (**retrieve\_element**(temp\_a->row, temp\_b->column, destiny->values) != NULL)

{

**retrieve\_element**(temp\_a->row, temp\_b->column, destiny->values)->value += temp\_a->value \* temp\_b->value;

}

//Cria o elemento para guardar a multiplicação

**else**

{

destiny->values = **insert\_ordened**(temp\_a->value \* temp\_b->value, temp\_a->row, temp\_b->column, destiny->values);

}

}

temp\_b = temp\_b->next;

}

temp\_a = temp\_a->next;

}

//Checa se há na lista da matriz resultado algum elemento nulo

**checkFor\_nullElements**(destiny);

**return** destiny;

}

Esta função realiza a multiplicação de duas Matrizes Esparsas, guardando-as em uma terceira que é retornada ao término da função.

A função retorna um erro caso as matrizes a serem multiplicadas venham ferir o conceito Amxn \* Bnxi onde A e B são duas matrizes que só podem ser multiplicadas se o número de colunas em A for o mesmo número de linhas em B.

Inicialmente cria-se a matriz resposta que irá comportar a multiplicação de duas matrizes que denominaremos A e B.

Utiliza-se duas variáveis temporárias que referenciam os elementos não nulos de A e de B respectivamente. Percorre-se todos os elementos não nulos de A, e para cada elemento não nulo de A percorre-se todos os elementos não nulos de B procurando seu elemento correspondente. Para cada elemento de A, se sua coluna equivaler a linha de algum elemento B, deve se multiplicar os dois elementos correspondentes (o elemento de A e de B) e somar com o elemento na matriz destino que corresponde a sua posição correta que é linha de A e coluna de B. O próximo passo é apontar a variável dos elementos não nulos de A para o próximo elemento e repetir o processo.

Ao somar a multiplicação dos elementos correspondentes de A e B com o seu lugar correspondente na matriz destino (linha de A e coluna de B), este elemento na matriz destino pode não existir na lista de elementos não nulos, o que significa que ele é zero. Neste caso insere-se ordenado a multiplicação através da função já vista anteriormente. Caso contrário, busca o ponteiro para o elemento na posição da matriz de destino através da função já vista e soma com seu valor atual.

Quando for percorrido todos os elementos em A, já foram multiplicados todos os elementos correspondentes. O funcionamento da função é de certa forma similar ao algoritmo de multiplicação comumente utilizado.

Ao final, utiliza-se a função já visto anteriormente para checar se alguma soma deu resultado zero, para retirar esse elemento da lista de elementos não nulos da matriz destino.

**Matrix**\* **sum\_sparseMatrix**(**Matrix**\* a, **Matrix**\* b)

{

**if** (a->height != b->height || a->width != b->width)

{

**printf**("\t\*\*\*Abortando: \*Função sum\_sparseMatrix\* Soma inválida, matrizes com tamanhos diferentes\n\n");

**exit**(1);

}

//Cria a matriz resposta

**Matrix**\* destiny = **create\_sparseMatrix**(a->height, a->width);

**List**\* temp\_a = a->values;

**List**\* temp\_b = b->values;

//Faz uma comparação elemento-não-nulo a eçemento-não-nulo

**while** (temp\_a != NULL && temp\_b != NULL)

{

//Se existe dois elementos nas mesmas posições

**if** (temp\_a->row == temp\_b->row && temp\_a->column == temp\_b->column)

{

destiny->values = **insert\_ordened**(temp\_a->value + temp\_b->value, temp\_a->row, temp\_a->column, destiny->values);

temp\_a = temp\_a->next;

temp\_b = temp\_b->next;

}

/\*Se A possui elemento antes(posição) de B, então soma-se elemento de A com 0 (pois assume-se que este elemento em B é 0)\*/

**else** **if** (temp\_a->row < temp\_b->row || (temp\_a->row == temp\_b->row && temp\_a->column < temp\_b->column))

{

destiny->values = **insert\_ordened**(temp\_a->value, temp\_a->row, temp\_a->column, destiny->values);

temp\_a = temp\_a->next;

}

/\*Se B possui elemento antes(posição) de A, então soma-se elemento de B com 0 (pois assume-se que este elemento em A é 0)\*/

**else**

{

destiny->values = **insert\_ordened**(temp\_b->value, temp\_b->row, temp\_b->column, destiny->values);

temp\_b = temp\_b->next;

}

}

//Significa que acabaram os elmentos-não-nulos em B

**while**(temp\_a != NULL)

{

destiny->values = **insert\_ordened**(temp\_a->value, temp\_a->row, temp\_a->column, destiny->values);

temp\_a = temp\_a->next;

}

//Significa que acabaram os elementos-não-nulos em A

**while**(temp\_b != NULL)

{

destiny->values = **insert\_ordened**(temp\_b->value, temp\_b->row, temp\_b->column, destiny->values);

temp\_b = temp\_b->next;

}

//Checa se há na lista da matriz resultado algum elemento nulo

**checkFor\_nullElements**(destiny);

**return** destiny;

}

Esta função realiza a soma de duas Matrizes Esparsas, guardando-as em uma terceira que é retornada ao término da função.

A função retorna erro caso a dimensão das matrizes sejam diferentes.

É criada a matriz de destino utilizando a função que cria uma Matriz Esparsa já visto anteriormente e inicializam-se duas variáveis do tipo **List**\* que referenciam os valores não nulos das matrizes denominadas A e B respectivamente.

As duas listas são percorridas simultaneamente enquanto a lista de A e a lista de B não chegam ao fim. Durante isto se por acaso o elemento a ser verificado de A têm mesma linha e coluna do elemento a ser verificado de B, somam-se os dois e os guarda na matriz destino utilizando a função de inserção ordenada já vista anteriormente.

Se por acaso o elemento a ser verificado em A vem antes do elemento a ser verificado em B (tem linha menor, ou linha igual e coluna menor) soma-se ele com 0 pois assume-se que seu elemento correspondente de mesma linha e coluna em B seja 0 e apenas a variável que referencia os elementos não nulos de A passa a referenciar seu próximo dando continuidade.

Se por acaso o contrário acontece, soma-se a o elemento a ser verificado em B com 0, pois certamente ele vem antes do elemento a ser verificado em A e assume-se que seu elemento correspondente em A tenha valor 0, apenas a variável que referencia os elementos não nulos de b passa a referenciar seu próximo elemento dando continuidade.

Ao fim desde processo, se por acaso a lista de A não foi percorrida até o fim, insere cada elemento de A na matriz destino utilizando a função de inserção ordenado já visto anteriormente pois como B já foi percorrido, assume-se que todos os elementos correspondentes de A são iguais a zero.

Se por acaso a lista de B não foi percorrida até o fim, insere cada elemento de B na matriz destino utilizando a função de inserção ordenado já visto anteriormente pois como A já foi percorrido, assume-se que todos os elementos correspondentes de A são iguais a zero.

Encerrando a função, checa se há algum elemento nulo na lista de elementos não nulos através da função já vista anteriormente.

**Acerca do Testes**

◦ Soma

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***-*** | **-** | **-** | **-** | ***Média*** |
| ***Memória em Megabytes*** | **1,42** | **1,46** | **1,42** | **1,44** |
| ***Tempo em Segundos*** | **0** | **0** | **0** | **0** |

A complexidade de pior caso do algoritmo, tomando na como a quantidade de elementos não nulos da primeira matriz e nb a da segunda (não importando o tamanho da matriz) é O(na + nb).

Esta complexidade se dá quando nenhum elemento em A corresponde com B, apenas inserindo cada elemento de cada matriz na matriz destino.

Se o número de elementos não nulos for a matriz inteira na é o número de elementos de A e nb é o número de elementos em B.

◦ Multiplicação

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***-*** | **-** | **-** | **-** | ***Média*** |
| ***Memória em Megabytes*** | **1,39** | **1,40** | **1,38** | **1,39** |
| ***Tempo em Segundos*** | **0** | **0** | **0** | **0** |

A complexidade de pior caso do algoritmo, tomando na como a quantidade de elementos não nulos da primeira matriz, nb a da segunda e nc a da matriz destino (não importando o tamanho da matriz) é O(na \* nb \* nc).

Esta complexidade se dá quando se multiplica linhas de A que tem todos os elementos não nulos por colunas de B que tem todos os elementos não nulos, onde apenas na primeira vez se criaria o elemento na matriz destino e teria de buscar o elemento e este teria de percorrer a lista da matriz destino toda para achar o elemento.

**Considerações Finais**

◦ Conclusão

Utilizar Matrizes Esparsas pode ser uma ótima saída uma vez que são matrizes enormes com poucos elementos não nulos, assim obtendo um ótimo tempo para operações e um baixo consumo de memória em vista de operações de matrizes de forma comum. Porém deve ser analisado caso a caso pois pode não ter relativamente um bom tempo e um baixo consumo de memória em certas ocasiões, como por exemplo a matriz ser totalmente não nula (o que deixaria de ser uma Matriz Esparsa), uma vez que é um algoritmo que faz muitas verificações e percorre listas muitas vezes perdendo o acesso direto a matriz como na forma usual de manipular matrizes.