FA17 MP10 - Sparse Matrix

Created by Chen, Yuting Wu, last modified by Kim, Dae Hee on Nov 09, 2017

Due Thursday, November 16th, 10:00 PM

Introduction

A sparse matrix is a matrix where most of the elements are zero. The simplest way to store a matrix is to store the value of every element. With a sparse matrix, storing every element means wasting a lot of space storing zeroes. Therefore, it's more efficient to store only the few non-zero elements. A non-sparse matrix can still be stored in this structure, but at some point it becomes more efficient to store every value in a matrix.

In this MP you will be writing code for a data structure called **a list of tuples** that stores sparse matrices more efficiently than a 2-D array.

The Pieces

- main.c - contains the code to call the functions you implement in sparsemat.c.
- sparsemat.h - contains the function definitions and some descriptions of the functions you have to write.
- **sparsemat.c - contains all the functions you have to write. Only code in this file will be graded.**
- cmp_mat.o - contains functions for printing a sparse matrix to the terminal.
- Makefile - A file that tells the make command how to compile the files into a single program.

Details

Text Files

The data for the matrices are saved and loaded to .txt files in the */matrices/input_mats* folder. The input matrices were generated using the MATLAB script *matrices/generate_input_matrices.m* and it is included if you wish to look through it. The first line of the text files contains the dimensions of the matrix separated by a space. Every line after the first is a single non-zero element. The first number is the row index, the second the column index, and the final the value.

For example, for the matrix 4x3 matrix *A*

$$A = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 4 & 5 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

the text file would be

Matrix A

```
4 3
0 1 2
1 1 4
1 2 5
3 2 1
```

List of Tuples (tuples)

A n-tuple is a sequence of elements of size n. Each non-zero entry in a matrix can be represented by a single 3-tuple, also known as a triplet: (row of the element, col of the element, value). The struct **sp_tuples** contains information about the number of columns and rows in the matrix, and the number of nonzero elements. It also has a pointer to the head of a sorted linked list called **tuples_head**. Each node in the list contains information for a single tuple/non-zero entry (A node should not exist with a value of 0). The list should be sorted in row major order.

For the matrix A above, the linked list should be in the order: (0,1,2), (1,1,4),(1,2,5) , (3,2,1). If the matrix is completely empty, tuples_head should point to NULL.

Functions to be Implemented

```
sp_tuples * load_tuples(char* input_file);
```

load_tuples should open a file with the name 'input_file', read the data from the file, and return a matrix of the list of tuples type. If any coordinates repeat in the input file, the newer coordinates (a lower line closer to the end of the text document) should overwrite the old line. If there is an entry with a value of 0 then the corresponding node should be deleted if it exists. The elements in the input text file may be unordered (unlike the example text file above), **but the list of tuples returned will need to be in order**. You do not have to handle cases where the input_file doesn't match the specified format.

```
double gv_tuples(sp_tuples * mat_t,int row,int col);
```

gv_tuples return the value of the element at the given *row* and *column* within the matrix.

```
void set_tuples(sp_tuples * mat_t, int row, int col, double value);
```

set_tuples sets the element at *row* and *col* to *value*. This function will need to do several things:

- if value is 0, it will need to find the node at *row* and *col* if it exists, and delete it from the list. Be sure to free the nodes from memory or they will be lost forever.
- For any other value, the function will need to find the correct location for the node within the sorted linked list. If the entry already exists, the function should replace the old value. If the entry doesn't exist, a node should be created and inserted into the linked list.
- **You also need to update the 'nz' variable of the struct.**

hint: It may be useful to write your own helper functions to organize your code for this function.

```
void save_tuples(char * file_name, sp_tuples * mat_t);
```

save_tuples writes the data in a sparse matrix structure to a text file in the format specified above. Because of the way the linked lists are ordered, writing the entries of the matrix as you traverse the list will give an output in row major order. Your text file output must be in this order even though *load_tuples* should able to handle reading un-ordered text files.

```
sp_tuples * add_tuples(sp_tuples * matA, sp_tuples * matB);
```

Typically, to calculate the matrix C = A + B, you would loop through every element of matrix A and B and add them together. With large sparse matrices, this results in a lot of memory accesses and operations just to calculate 0+0. Instead, its more efficient to only do operations on the non-zero elements of A and B, which can be found by traversing the linked lists of both matrices. The algorithm is as follows. A_(i,j) is defined as the value of the element in the ith row and jth column of matrix A. Return NULL if addition between matA and matB is not possible.

```
INPUT: matrix A and B both with size m by n.
INITIATE matrix C with the same size as A and B and all entries = 0
FOR every non-zero entry in A                   (traverse the linked-list in matA)
    i = row of current entry in A; j = column of entry
    C_(i,j) = C_(i,j) + A_(i,j)
FOR every non-zero entry in B                   (do the same for matB)
    i = row of current entry in B; j = column of entry
    C_(i,j) = C_(i,j) + B_(i,j)
```

```
sp_tuples * mult_tuples(sp_tuples * matA, sp_tuples * matB);
```

As with addition, we don't want to spend time and processing power by calculating operations with 0. The algorithm for matrix multiplication is similar to addition but slightly more complex. This algorithm works by traversing matrix A. For each node in A, a nonzero element needs to be multiplied by every entry in B where the row of B matches the column of A. The results of each of these multiplication operations is accumulated into the associated entry in C. Because the structures are all sorted in row major order, you can take advantage of the fact that nodes of the same row will be next to each other when searching matrix B for rows that match. Also note that unlike addition, matrix multiplication is not commutative A*B != B*A. Return NULL if multiplication isn't possible.

```
INPUT: matrix A with size mAxnA, and with matrix B size mBxnB       (make sure that inputs are of valid size: n1 == m2)
INITIATE matrix C with the size mAxnB and all entries = 0
FOR every non-zero entry in A
    iA = row of current entry in A; jA = column of current entry in A
    FOR every non-zero element in B with row iB = jA                (every element in B where the row is equal to the column of the
element in A)
        C_(iA,jB) = C_(iA,jB) + A_(iA,jA) * B_(iB,jB);              (accumulate value into C_(iA,jB))
```

```
void destroy_tuples(sp_tuples * mat_t);
```

destroy_tuples should free all memory associated with the given matrix.

Specifics

- Your program must be written in C in the file sparsemat.c provided for you.
- You must implement the all of the functions listed above correctly.
- Your routine's return values and outputs must be correct.
- Don't change the function definitions for the functions you have to implement.
- Your code must be well commented, including introductory paragraph.

Building and Testing

The commands to build your project and compile your code. Make clean will first remove any compiled code (object files and the mp8 binary) from the MP folder. Make will then compile everything.

```
make clean
make
```

Once compiled, run the program from MP9 directory as:

```
./mp10
```

The above command will print out your matrices to the terminal without *save_tuples* implemented. The golden output is provided to compare in the "matrices/output_mats/gold/" folder. To compare these outputs, save the program's output to a file and compare it to the golden output using the following command from the MP10 folder. diff will have no output if the files match completely.

```
./mp10 > matrices/output_mats/mp10_output.txt
diff matrices/output_mats/mp10_output.txt matrices/output_mats/gold/mp10_output.txt
```

Once *save_tuples* has been implemented, mp10 will also save output files to the folder "matrices/output_mats/". Gold output files are included in "matrices/output_mats/gold/". If your output files match the gold output files exactly, then your output is correct. The filenames of your output files have an extra letter appended to them, corresponding to the tuple datatype. You can look at main.c to see where each of the output matrices come from. For example, the output for the *add_tuples() test* is saved in "a_Ct.txt" and can be compared to the gold solution by running the command.

```
diff matrices/output_mats/a_Ct.txt matrices/output_mats/gold/a_Ct.txt
```

Some Hints and Suggestions

- Remember to initialize all your pointers, even the ones that that don't point anywhere (to NULL). This could be the issue if valgrind complains about conditional moves or jump depending on uninitialized value(s).
- If mp10 segfaults when during a function on a sparse matrix (such as adding two matrices or printing one), it's possible the problem occurred when the matrix was created/modified, but the program doesn't segfault until it attempts to traverse the matrix
- Much of the code for the different functions will be similar. It will help to keep your code organized with comments and subfunctions. For example, one useful helper function for this MP might be a function which deletes a node (and is called by set_tuples when appropriate).

Grading Rubric

Functionality (90%)

- 10% - *load_tuples* functions are implemented correctly
- 10% - *save_tuples* functions are implemented correctly
- 35% - *set_tuples* functions adds/removes/changes nodes correctly.
- 5% - *gv_tuples* return the correct values
- 10% - *add_tuples* return the correct matrix.
- 15% - *mult_tuples* return the correct matrix.
- 5% - *destroy_tuples* free all memory correctly.

Style (5%)

- Code is clear and well-commented, and compilation generates no warnings (note: any warning means 0 points here)

Write-up (5%)

- Introductory paragraph explaining what you did (even if it's just the required work)

Some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.