HOMEWORK #2

SUBMISSION DUE DATE: 27/5/2020 23:00
SUBMISSION IN SINGLES OR PAIRS

INTRODUCTION

A **Sparse Matrix** is a matrix in which most of the elements are zero, and often appears in scientific or engineering applications. It can be very beneficial to take advantage of the sparse structure of a matrix when storing and manipulating sparse matrices.

For example, performing 1000 iterations of power iteration on a sparse matrix of size 5000 on nova takes *0.2-0.3 seconds*, compared to *1 minute* for the non-sparse implementation.

In this assignment, you are asked to implement a C module with *two* implementations for sparse matrices (linked lists and arrays), and use it to reimplement your eigenvector program from the previous assignment with these implementations. The program receives an input matrix and an implementation choice and outputs an estimate of its eigenvector with the highest eigenvalue, by storing and manipulating the input matrix as a sparse matrix using the chosen implementation.

The purpose of this assignment is to take a step towards the implementation of the project, and practice advanced topics in C:

1. Implement a module of the final project
2. Practice code modularity and design (and function pointers)
3. Create your first makefile
4. Check your code according to the guidelines given in class

A new **tester** is provided to you, and your submission will be graded accordingly by running the tester with your outputs on various inputs. The tester determines whether the output is correct, and you can use it to test your code. It is detailed later in this document.

SPARSE MATRIX

The sparse matrix module is a generic module for sparse matrices. It contains two implementations, for linked lists and arrays. This section will detail the module and describe the two implementations.

The header file "`spmat.h`" of the sparse matrix module is provided to you. You are required to implement its code in "**`spmat.c`**", and use the module in your main program.

The sparse matrix module uses an inheritance-like interface, such that it can be used without relying on the specific implementation, and future implementations can be easily added and used without modifying existing code that uses the module (except a single initialization call which determines the implementation used). Since there is no inheritance in C, we achieve this using function pointers.

The header file defines the type `spmat`, a struct for a sparse matrix independent of the implementation. This struct is similar to an abstract class, where the function pointers are like abstract (virtual) methods, populated on initialization (the "`allocate`" functions) to reference the implementation-specific functions.

The two "`allocate`" functions allocate and return a new `spmat` struct, with its fields populated according to the relevant implementation. The "`private`" field should be used to reference a struct with data specific to the current implementation (i.e., implementation-specific sparse matrix structure). Note that the "`allocate`" functions only allocate memory and initialize the struct fields, but do not initialize the values of the matrix itself (this is done via the "`add_row`" function).

The `spmat` struct defines the following function pointers:

- **`add_row`** – used to initialize the sparse matrix with values. You may assume this is called exactly *n* times in order, after "`allocate`" and before any other call.
- **`free`** – frees all resources used by the sparse matrix, including any implementation-specific resources as well as the `spmat` struct itself.
- **`mult`** – multiplies the sparse matrix by the given vector *v*, storing the result into the `result` vector (which you may assume is pre-allocated to size n).

Beyond following the header file, the implementation details are up to you. Define types, implement the functions and helper functions, etc. Justify your choices with comments - your code will also be graded for design.

Note that this module should be independent, i.e., it depends on no external code (beyond standard libraries), and no external code depends on its implementation. Any change in the implementation should compile and run successfully, so long as it still satisfies the interface defined by the header file.

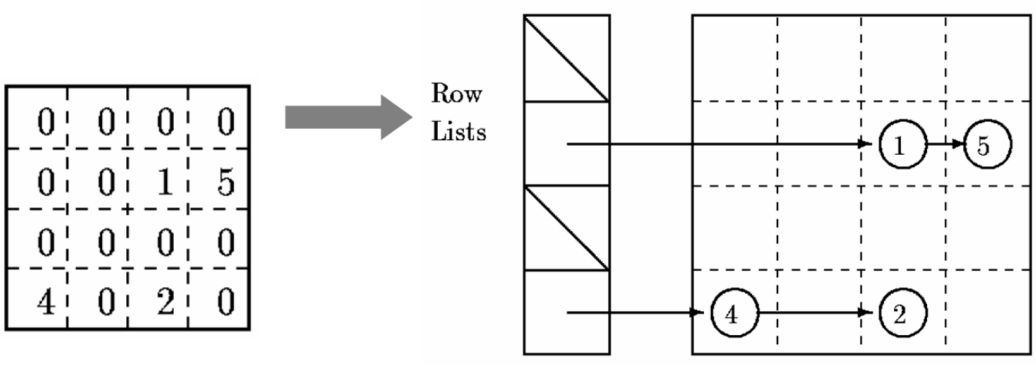
Following are descriptions for the two implementations of a sparse matrix in this assignment, using linked-lists and arrays.

LINKED LISTS

A linked-lists sparse matrix contains a set of linked lists, one for each row of the matrix.

Each cell contains its value and column index, as well as a pointer to the next non-zero element in the same row (or NULL if it is the last non-zero cell in its row). For each row, this linked list skips over cells with a value of zero, thus we avoid storing these cells and can quickly iterate a row, skipping over all zero values.

Here is an illustration of a sparse matrix represented with linked lists:



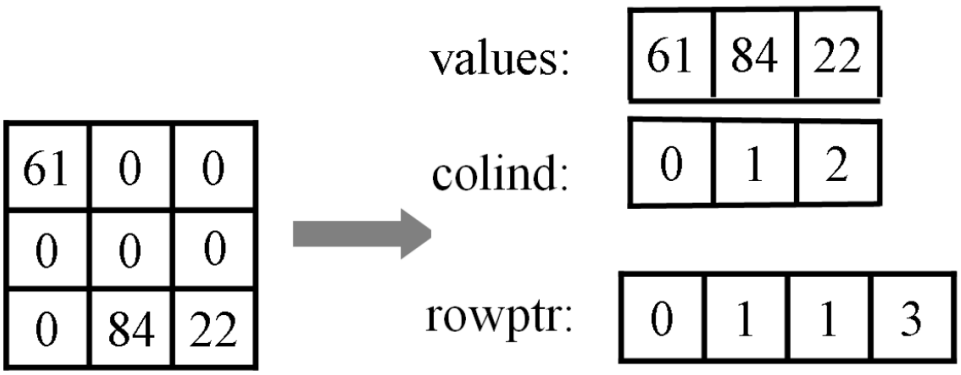
ARRAYS

An arrays sparse matrix uses a compressed array with helper arrays to store only non-zero matrix values.

We store an array of all non-zero values (the *compressed* array) in the original matrix, as well as the *column* index for each such value. In addition, we store an array of row indices, such that for each row we keep the index (in the compressed array) of the first non-zero value from that row onwards.

For convenience, the rows array should be of size n+1, with the last entry containing the number of non-zero elements in the matrix (the size of the compressed array).

Here is an illustration of a sparse matrix represented with arrays:



MAIN PROGRAM

This section will detail the *new* power iteration program. Implement this program in the source file **`main.c`**.

The main program of this assignment will reimplement the power iteration program from the first assignment, performing power iteration with a sparse matrix. For a description of the power iteration algorithm, see the previous assignment.

The program receives **three or four** arguments, use **`assert`** to ensure they are provided:

1. **Input matrix** – file name of a symmetric matrix to perform power iteration on.
2. **Initial vector** – file name of an initial vector b_0 to use for power iteration (optional).
3. **Output file** – file name to output the produced eigenvector into.
4. **Implementation** – a flag determining what sparse matrix implementation to use, either "**`-list`**" or "**`-array`**" (assert or exit otherwise).

All input and output file formats are identical to the previous assignment.

Note: the 2nd argument should be read as a standard matrix (not sparse!) and used instead of a random b_0. Following students' comments of issues with the tester due to the random nature of the algorithm, your results from the tester should now be consistent. If only three arguments are provided, ignore the 2nd argument and instead randomize an initial vector b_0 as in the previous assignment.

As in the previous assignment, the input matrix may be large and we do **not** read it into memory, instead we read it one row at a time. However, after converting it into a sparse matrix we can keep the entire sparse matrix in memory.

Here is the flow of the program:

1. Read initial vector b_0 into memory, or randomize it if `argc==4` (three arguments)
2. Read the size of the input matrix
    a. If the chosen implementation is arrays, iterate over the input matrix (one row at a time) to count the number of non-zero values
3. Allocate a sparse matrix of the given implementation
4. Read the input matrix (one row at a time)
    a. For each row, add it to the sparse matrix
5. Perform power iteration to obtain the eigenvector
6. Write the produced eigenvector to the output file

Remember: the main program does not depend on the sparse matrix implementation or any of its inner functions. It only includes and uses the provided header file!

ASSUMPTIONS AND REQUIREMENTS

You may make the following assumptions, and must follow the following requirements:

- You may assume the input files are in the correct format (but may not assume that you succeed in opening them or reading/writing).
- You may use **`assert`** to exit on any error in accessing files or allocating memory.
- You may not use functions from *math.h* except **`fabs`** and **`sqrt`**.
- You may assume input files do not lead to math errors, e.g., division by zero.

COMPILE, DEBUG AND MAKEFILE

Your code should compile and run on Nova, following the same instructions in the previous assignment, along with all flags:

```
-ansi -Wall -Wextra -Werror -pedantic-errors
```

The **`-lm`** flag should be added at the end of the `gcc` command to use math functions.

Your code is graded for performance! When your output is correct, the tester outputs the time it took. Your code should exhibit roughly similar performance.

YOUR MAKEFILE

In this assignment you are required to write and submit your own makefile.

The makefile should compile all parts of your program by invoking **"`make`"** or **"`make all`"** (both should work, identically). The **"`make clean`"** command will clean previous compilations and all temporary files.

The makefile should compile each source code in two steps, as described in previous tutorials (see the makefile examples in HW1 and Tutorial 1), and create a single executable.

The executable filename should be **"ex2"**.

TESTER FILE

Attached to this assignment is an executable file **tester**. It can generate random matrix files and check your output files. Using it is an integral part of the assignment.

The tester can operate in three ways:

1. To generate a random sparse symmetric matrix:
   `>> ./tester -input <filename> <n>`
   Where n is the number of rows and columns in the created matrix.

For example: `>> ./tester -input input.arr 1000`

2. To generate a random vector:
   `>> ./tester -vec <filename> <n> <range>`
   Where n is the number of rows in the created vector, and `range` is the range for randomization (use `1` for a floating-point number in `(0,1]`, and `>1` for a random integer in `[1,range]`).
   For example: `>> ./tester -vec b0.arr 1000 1`
3. To check your output:
   `>> ./tester -check <matrix> [b0] <eigen> <implementation>`
   Where `matrix` is the input matrix file, `b0` is an optional initial vector file (if omitted, the initial vector is randomized), `eigen` is your output eigenvector file, and `implementation` is either `-list` or `-array` (to compare performance).
   In case of errors, the tester specifies the first error that it detects and terminates.
   For example: `>> ./tester input.arr b0.arr eigen.arr -list`

SUBMISSION

You may submit this assignment alone or in pairs. Please submit a zip file named **id1_id2_hw2.zip** replacing *id1* and *id1* with the actual **9-digit** ID of both partners. Only a **single** partner should submit the assignment! If you submit alone, name it **id_hw2.zip**.

The zipped file must contain the following files (at the root, **no folders**):

- `spmat.c` – Definitions and implementation of all sparse matrix functions and types.
- `main.c` – The power iteration program.
- `makefile` – your makefile.

Submit **only these files! Do not** add spmat.h, SPBufferset.h, or any other files or folders.

MAC users: create (or check) your zip files under Linux, as otherwise hidden auxiliary files are automatically added and will reduce your grade.

You may leave any output (e.g., debug, time measurements) in your code, but be careful – too many outputs will degrade your performance.

REMARKS

- For questions regarding the assignment, please post in the forum.
- Late submissions are not acceptable unless you have the lecturer approval.
- Borrowing from others' work is unacceptable and bears severe consequences.

GOOD LUCK