Coursework Assignment

COMS21103 Data Structures and Algorithms 2011-2
Ian Holyer

This year there will be only one coursework assignment, worth 30% (unlike like last year when there were two). Bogdan's quiz was worth 10% and the exam will be worth 60%. It is an individual assignment based on one set by Dan Page last year.

Matrix

1 Introduction

The goal of this assignment is simple: you are tasked with developing and optimising an implementation of a data structure and algorithms for sparse matrix arithmetic. Such implementations support a huge range of scientific use-cases, and you can view your solution as a limited version of libraries such as Sparse BLAS [1]. The practical nature of this assignment is crucial: it should help you more fully understand and value various topics covered only in theory elsewhere. You should implement your solution in C, using GCC with flags **`-std=c99`** and **`-pedantic`** to make sure you are using the C99 standard and nothing else. Although there may be multiple valid approaches at each stage, your solution should aim to be as efficient as possible wrt. execution time. Within reason, a trade-off favouring time over space is encouraged. Part of your final mark will be derived from how efficient your solution is when compared with others, but eligibility for this portion of the marks depends on it being functionally correct.

I recommend doing this assignment in the lab or your own Linux computer. If you use Windows, be sure you understand the extra complications, use gcc or otherwise to make sure you are meeting the C99 standard, and do a final check of everything in the lab before submitting.

2 Submission and marking

- This assignment is intended to help you learn something; where there is some debate about the right approach, the assignment demands you make an informed decision yourself and back it up with a reasonable argument based on your own background research.
- This assignment description may or may not refer directly to the file marksheet.txt. Either way, you should download this ASCII text file from www.cs.bris.ac.uk/Teaching/Resources/COMS21103/matrix/marksheet.txt then complete and include marksheet.txt it in your submission; this is not optional, and failure to do so may result in a loss of marks.
- Any implementation work will be marked on a platform equivalent to those available in the CS lab (MVB-2.11): take care to check your solution compiles and has been tested using the default operating system and development tool-chain versions available.
- Where appropriate, include instructions that carefully describe how to compile and execute your submission; the ideal solution would include a Makefile (or equivalent) where not already provided.
- To make the marking process easier, your solution should only write error messages to stderr (or equivalent). The only output printed on stdout (or equivalent) should be that specified within the assignment description.
- You should submit your work via the SAFE submission system at wwwa.fen.bris.ac.uk/COMS21103/ including all source code, written solutions and any auxiliary files you think are important (e.g., example inputs and outputs). In the unlikely event that the submission system fails,

contact me (don't rely on email).
- Take note of the strict requirement wrt. functional correctness: the marks outlined in marksheet.txt demand computation of the correct result before the issue of efficiency is even considered. As such, developing a robust testing strategy for your solution is vital.
- The assignment may look fairly closed, but in fact allows a large number of potential directions and choices. Your solution should (concisely) document pertinent facts in marksheet.txt; this does not need to be a lengthy report, but should, for example include an explanation of any optimisation steps applied.

3 Material

If you download and unzip the file www.cs.bris.ac.uk/Teaching/Resources/COMS21103/matrix/question.zip or (question.tar.gz) somewhere secure within your file system, you should find the following:

- Makefile, a GNU make based build system for the source code.
- A set of small test vectors for each stage: those associated with stage 2 are named set1-stage2.input and set1-stage2.output for example.
- util.c and util.h, which combine some standard libraries, and where you can add your own support functions if you want.
- matrix.c and matrix.h, a driver program with incomplete functions for each stage.

You should be able to complete the assignment by altering only the matrix.c and matrix.h files and optionally the util.c and util.h files; either way, only existing files you alter, additional files you create, and marksheet.txt need to be submitted.

4 Description

An unstructured sparse matrix is simply a matrix where a significant number of elements are zero. Using a suitable in-memory data structure that stores only non-zero elements, it is possible to represent such a matrix in a much more compact way than a dense equivalent (where all elements are stored whether zero or not). An (m x n)-element sparse matrix (with m rows and n columns) can be described on-disk by an ASCII text file with the following format:

```
m,n
r0,c0,v0
r1,c1,v1
r2,c2,v2
...
```
$r_{t-1}, c_{t-1}, v_{t-1}$

i.e., a single line containing the dimensions of the matrix, followed by t lines specifying non-zero elements(via a row and column index, and the value); all fields within a given line are separated by a single comma character (i.e., ASCII 44(10) = 2C(16)), and lines are terminated by a UNIX line feed character (i.e., ASCII 10(10) = 0A(16)). (If you are working on windows, you may also want to accept and/or produce files with both a carriage return and line feed character at the end of each line). For example, the file

```
5,5
0,0,-1
0,4,2
1,1,5
1,3,6
2,2,7
3,1,8
3,3,9
4,0,3
4,4,-4
```

represents a (5 x 5)-element matrix M =

```
-1 0 0 0 2
 0 5 0 6 0
 0 0 7 0 0
 0 8 0 9 0
 3 0 0 0 -4
```

where, given M[i, j] denotes the element in the i-th row and j-th column, M[0,0] = -1, M[0,4] = 2 and so on. Note that:

- In the example above, lines describing the non-zero elements are ordered by row and column to make it human-readable; you cannot assume this will always be the case. That is, the correctness of any file (representing a matrix used as input or output) relates to whether it specifies the associated matrix content correctly: the order elements are specified in is irrelevant.
- Each element, i.e., each v_k for $0 <= k < t$ is a signed 32-bit integer; you can ignore the problem of overflow during computation, and assume elements can always be represented by the same data type.
- Two matrices cannot be deemed equal unless both their dimensions and each of their elements are equal; you can assume that the dimensions m and n cannot exceed the range of a 32-bit unsigned integer.
- The row and column indices start at zero, meaning that for any $0 <= k < t$ we know $0 <= r_k < m$ and $0 < c_k < n$.
- You can assume a definition of sparse that says less than 1% of entries in a given row will be non-zero; this is an asymptotic guideline (i.e., it holds for large enough m and n only) however, not an absolute guarantee.

Stage 1

1. Implement a data structure that can represent a sparse matrix. It is essential that the data structure is space efficient since your solution will be tested with matrices whose dimensions are many orders of magnitude larger than those provided.
2. Implement functions that enable an in-memory instance of the sparse matrix data structure to be read from or written to an on-disk file as per the format above.
3. Ensure your program can be executed using a command of the form
   matrix stage1 ${INPUT} ${ROW} ${COL}
   where
   - ${INPUT} names an input file,
   - ${ROW} is an integer row index, and
   - ${COL} is an integer column index.

When executed in this way, the program should read a matrix from the input file then print, one per line and to stdout, the specified element and two comma separated lists of elements in the given row and column. For example, given an input file called stage1.input that matches the example above, one might execute

```
matrix stage1 stage1.input 1 1
```

to produce

```
5
0,5,0,6,0
0,5,0,8,0
```

Stage 2

The transpose of a matrix is simply where the column and row indices are exchanged. That is, the transpose of the (2 x 4)-element matrix

```
1 2 3 4
5 6 7 8
```

is the (4 x 2)-element matrix

```
1 5
2 6
3 7
4 8
```

1. Write a function that takes a sparse matrix as input, and transposes it to form an output.
2. Ensure your program can be executed using a command of the form

   ```
   matrix stage2 ${OUTPUT} ${INPUT}
   ```

   where
   - ${INPUT} names an input file, and
   - ${OUTPUT} names an output file.

   When executed in this way, the program should read a matrix from the input file, transpose it, then write the result to the output file.

Stage 3

1. Write a function that takes two sparse matrices as input, and computes their sum to form an output. If the inputs matrices are not compatible, your program should abort (optionally printing a message to stdout); for this stage, you can assume the input matrices are incompatible if and only if they do not have the same dimensions.
2. Ensure your program can be executed from a BASH shell using a command of the form

   ```
   matrix stage3 ${OUTPUT} ${INPUT_0} ${INPUT_1}
   ```

   where

- ${INPUT_0} and ${INPUT_1} name input files, and
- ${OUTPUT} names an output file.

  When executed in this way, the program should read matrices from the input files, compute their sum, then write the result to the output file.

Stage 4

1. Write a function that takes two sparse matrices as input, and computes their product to form a result. If the inputs matrices are not compatible, your program should abort (optionally printing a message to stderr); the condition for incompatibility is different for this stage than for matrix addition.
2. Ensure your program can be executed using a command of the form

```
matrix stage4 ${OUTPUT} ${INPUT_0} ${INPUT_1}
```

   where
   - ${INPUT_0} and ${INPUT_1} name input files, and
   - ${OUTPUT} names an output file.

When executed in this way, the program should read matrices from the input files, compute their product, then write the result to the output file.

Stage 5

1. Write a function that takes *l* >= 2 sparse matrices as input, and computes their product to form a result. If the inputs matrices are not compatible, your program should abort (optionally printing a message to stderr); the condition for incompatibility is different for this stage than for matrix addition.
2. Ensure your program can be executed using a command of the form

```
matrix stage5 ${OUTPUT} ${INPUT_0} ${INPUT_1} ...
```

   where
   - ${INPUT_0}, ${INPUT_1} and so on name input files, and
   - ${OUTPUT} names an output file.

   When executed in this way, the program should read matrices from the input files, compute their product, then write the result to the output file.

References

[1] I.S. Du, M.A. Heroux and R. Pozo. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. In ACM Transactions on Mathematical Software (TOMS), **28** (2), 239–267, 2002.