CS4115 Week06 Lab Exercise

Lab Objective: The objective of this week's lab is to consider a matrix compression program. Matrices are used in a huge variety of computer applications and a key issue is the space they consume and the running time of algorithms on them. We will see how we can reduce both. Here's the lab summary:

❶ A quick introduction to the relationship between matrices and graphs

❷ Examine a well-established format for representing matrices sparsely

❸ Consider our own format that is inspired by the above but, at the expense of slightly higher file sizes, is simpler to write as C++ code

❹ Write a program, `matz`, that reads a matrix in the format we have been using and writes it out in a format that ignores all `0`-entries

❺ Submit `week06`'s lab for marking

In Detail

Please be patient and read the entire lab sheet through to the end before beginning with the work. There are a lot of new ideas in this for you to absorb so please don't rush in before you understand the material and what is being asked of you.

❶ One of the most common methods of representing and storing information nowadays is the *graph*. A graph, in the computer science sense, is a collection of *nodes* that contain the information and *edges* that represent links or relationships between pairs of nodes. One common way to represent a graph internally is by means of an $n \times n$ adjacency matrix of type `bool` (boolean). A `true` in entry (i, j) means that node i is connected (adjacent) to node j; the two nodes are "related".

We could use this way to represent street maps, or *street networks*, as they are often called: we represent the junction of two streets with a node and if another street junction is directly reachable then we mark the corresponding entry of the matrix with `true` and `false` otherwise. So each junction has a unique node number associated with it and the non-zero entries of the matrix for that row are the junctions (nodes) that are *immediately* reachable from it. We can go one step better and instead of storing `true` or `false` values in the matrix we can store the distance between them. Note that an edge of this network corresponds to a *piece* of a street

The problem with representing the *street network* with an array / matrix is that, in reality, most junctions are directly connected to very few other junctions. That is, the network is very sparse. Just think for a minute about the case of Limerick city. Can you think of a junction that is adjacent (directly connected) to more than 5 other junctions? Asking the same thing differently, can you find a junction where 5 streets meet? The overwhelming majority of the entries in the matrix will be 0, meaning that the junctions are not directly connected.

Whatever about trying to represent Limerick city and its n street junctions[1] using an $n \times n$ matrix (that uses $\mathcal{O}(n^2)$ space) it would be out of the question on a larger scale. Consider the street networks on this page. The files down the left hand column represent the street networks of regions for the world as taken from the OpenStreetMap project. The two right hand columns labelled n and m indicate, respectively, the number of nodes (street junctions) and street segments in the file.

The street network files are compressed using the `bz2` compression format. This should not cause a problem on the linux lab machines. After you have read Section ❷ below pick one of the countries and spend a few minutes looking at the type of information in its `.graph` and `.graph.xyz` files.

Street Networks

Two things to note about the data on this page:

1. the geo co-ordinates (longitude, latitude) of each node is stored in a *separate* file with the suffix `.xyz` and the information in the `.graph` file simply tells what nodes are connected to what other nodes. So to tell how far apart two nodes are you would have to do a distance calculation between the $x - y - z$ information for the two nodes as given in the `.graph.xyz` files;

2. Notice how sparse all of the networks are. Consider puny little Luxembourg. With just $n = 114,599$ nodes, using a matrix would require $114,599 \times 114,599 = 248,028,913$ entries, yet only $m = 119,666$ of them would be non-zero.

   Put another way, on average every node has just $119666/114599 = 1.044$ neighbours; if we were using an adjacency matrix every row would have on average 1.044 non-zero entries.

It happens in lots of situations where huge matrices are used to represent relationships that the "interesting" data in the matrix is a very small fraction of the entire thing.

❷ The data in the `.graph` file follows the MeTiS format. MeTiS is a very successful program

[1] I'm not sure of the exact value of n for Limerick but I cannot imagine it being more than a couple of thousand.

for analysing the huge graphs that arise in DNA sequencing or data mining or a variety of other applications. This is described in detail in §4.1.1 on p. 9 of the MeTiS user manual. The street network data that you have been looking at follows the format of Fig 2(a) on p. 11. However, the important case for you to look at is Fig 2(b) which is what we will base our format on.

❸ Our goal is to write a program that, given a matrix with a large number of zeros, only the non-zero values should be outputted but in a way that the original matrix can be reconstructed. You could think of this as a conversion program that converts a matrix file with a lot of 0s into a more efficient format.

From a programming point of view the file format shown on Fig 2(b) on p. 11 of the MeTiS manual has two problems. Imagine you had to write the conversion program that compressed a given matrix. Firstly, since the format provides the number of rows and non-zero entries on the first line (and another item to discuss later) it requires that you print out as the very first line of the output the number of nodes of the network (rows of the matrix) and no. of edges of the network (non-zero entries of the matrix). **But we will know the number of non-zero entries of the matrix only when we have read it in its entirety.**[2] So we will "cheat" by printing this piece of information as the **last** line of the file.

The second problem comes when you want to actually use the compressed file that the `matz` program will have created. That is, suppose you took a file that contained a matrix and you compressed it using `matz`, the program you will write. I copied a matrix from Week03's lab into this week's lab directory and named it `m1.w03` and to compress it I ran

```
        matz 6 6 < m1.w03 > m1.matz
```

Suppose you now want to read the contents of `m1.matz`. After the initial line that tells you how many nodes (rows), every subsequent line contains the non-zero entries, as in the style of Fig 2(b) you saw previously. But **how many non-zero entries are there?** To solve this would require more C++ knowledge than is worth it at this stage so we will compromise: the first entry of every line will be the *number* of entries to follow on this line.

So how does the file format described in Fig. 2(b) of the MeTiS manual compare to our proposed file format?

2.4	0	5.6	0
0	11.5	1.3	5.4
5.6	1.3	0	0
0	5.4	0	0

For the 4 × 4 matrix shown above the two panels below contrast the two compression formats. The format you should implement is `formatz`.

[2] By reading the entire file *twice* we could get over this: the first time just counts the non-zero entries, printing that out and only on the second time reading the file do we do the real work of writing out the non-zero entries. But even this approach would not work if we were reading from standard input.

```
4 8 1
1 2.4 3 5.6
2 11.5 3 1.3 4 5.4
1 5.6 2 1.3
2 5.4
```

Fig. 2(b)

```
4
2 1 2.4 3 5.6
3 2 11.5 3 1.3 4 5.4
2 1 5.6 2 1.3
1 2 5.4
8
```

formatz

The alert reader will have noticed that the first line of the “Fig. 2(b) format” has a trailing ’1’ after the number of rows and non-zero entries. This tells MeTiS that the edges are non-unitary – if every edge / entry was unitary then there would be no need to specify them at all and this is what is being said in the previous figure, Fig. 2(a). **Our default will be non-unitary or *weighted* edges.** Although the “compression” gains don’t appear to be great in this example they will be when we’re talking about matrices of the size experienced by Google, e-Bay, etc.

❹ With all that as background we can now give the main task. Write a program called `matz` that implements the `formatz` style of matrix compression. The program you write should take the dimensions of the matrix as two command-line arguments just like `matmult`. In the case of a square matrix a typical run might be

```
matz 6 6 < m1.w03 > m1.matz
```

A **sample executable** is available at `~cs4115/labs/week06/matz` that computes the correct output. You can verify your program’s correctness by comparing your results to what that program gives. There are also some input and out example files in that directory as well. Have a mooch around with

```
ls -l ~cs4115/labs/week06
```

The first thing your program should do is parrot back to the output `n`, the number of rows in the matrix. (If this matrix represented whether or not two web pages referred to each other then `n` would be the number of pages in your database.) It should then start processing each of the `n` rows of input, each of which contain `c` numbers. When it has read the `c` numbers on a row it should, as per *formatz*, write out a single line comprising

- the number of non-zero entries encountered, `nz`
- `nz` pairs of numbers where the first of each pair is the column number of a non-zero value and the second of each pair is the value itself

There should be `n` such lines.

The last line of output should be a single integer that is the total count of non-zero numbers encountered. If the matrix represented distances between connected nodes of a network then this number would be the number of edges in the graph.

Next week’s lab will build on this program so it is imperative that you complete this task this week.

Good luck.

❺ The command to submit your program is

```
        handin -m cs4115 -p w06
```

Labs that are to be handed in are open for handin at 09.00 on Thursday of the week it is *assigned* and, in order to get full marks, are due by 14.00 on Thursday of the *following* week; a lateness penalty applies to submissions made until 18.00, Monday after that. This gives you one week to work on each lab that is to be assessed and handin without penalty. The lab sheet will be available for reading prior to the first lab so that you can read it beforehand and come to the lab with questions.

Subject to last-minute changes, the planned schedule of lab assignments due for handing in are:

Lab. Week	Assessed	DueDate
Week02	✗	
Week03	✓	Thu, Week04
Week04	✗	
Week05	✗	
Week06	✓	Thu, Week07
Week07	✓	Thu, Week08
Week08	✗	
Week09	✓	Thu, Week10
Week10	✗	
Week11	✓	Thu, Week12