Sparse Data Structures

The linear systems arising from the discretization of our model problems are extremely sparse. In fact, no matter the size of the grid, the matrix associated with the 2D model problem will have at most five non-zero entries per row/column. Since we're working with sparse matrices, an understanding of how to store and use such matrices on a computer is crucial.

Consider the matrix $A = \begin{bmatrix} 4. & 0. & 0. & 2. \\ 0. & 1. & 0. & 0. \\ 0. & 0. & 5. & 7. \\ 6. & 3. & 0. & 8. \end{bmatrix}$

There are many ways to store a sparse matrix, all relying on some system to indicate the locations and values of the nonzero elements. Suppose that, in general, an $m \times n$ matrix A contains NZ nonzeros.

Coordinate (COO) Format

Maybe the most obvious storage format is to store tuples of the form

(*value*, *row index*, *column index*)

For the matrix above one possible storage scheme could be as follows

$$\begin{aligned} \texttt{val} &: \begin{bmatrix} 8 & 7 & 5 & 3 & 4 & 2 & 1 & 6 \end{bmatrix} \\ \texttt{row\_ind} &: \begin{bmatrix} 4 & 3 & 3 & 4 & 1 & 1 & 2 & 4 \end{bmatrix} \\ \texttt{col\_ind} &: \begin{bmatrix} 4 & 4 & 3 & 2 & 1 & 4 & 2 & 1 \end{bmatrix} \end{aligned}$$

Here, `val` is an NZ-length array of floats that gives the values of the entries of A, and `row_ind` and `col_ind` are NZ-length arrays of integers that give the associated row and column indices, respectively.

Of course, it would probably be convenient if things were ordered in a standard way. One thing we could do is sort the entries by row first, and then by column. So we have

$$\begin{aligned} \texttt{val} &: \begin{bmatrix} 4 & 2 & 1 & 5 & 7 & 6 & 3 & 8 \end{bmatrix} \\ \texttt{row\_ind} &: \begin{bmatrix} 1 & 1 & 2 & 3 & 3 & 4 & 4 & 4 \end{bmatrix} \\ \texttt{col\_ind} &: \begin{bmatrix} 1 & 4 & 2 & 3 & 4 & 1 & 2 & 4 \end{bmatrix} \end{aligned}$$

Chris Ketelsen
APPM 6640
Chapter 1
January 18, 2015

So the COO format requires $3NZ$ pieces of storage. We can do a lot better than that.

Compressed Sparse Row (CSR) Format

From the ordered form of the COO format, it's easy to see that some of the entries in the `row_ind` array are redundant. Once we know we're in row 2, for example, we don't need the other row indices to tell us that we're still in row 2. We just need to know when we've hit row 3. The CSR format leverages this fact to save some storage space. We have

$$\begin{aligned} \texttt{val} &: \begin{bmatrix} 4 & 2 & 1 & 5 & 7 & 6 & 3 & 8 \end{bmatrix} \\ \texttt{row\_ptr} &: \begin{bmatrix} 1 & 3 & 4 & 6 & 9 \end{bmatrix} \\ \texttt{col\_ind} &: \begin{bmatrix} 1 & 4 & 2 & 3 & 4 & 1 & 2 & 4 \end{bmatrix} \end{aligned}$$

Notice that the `val` and `col_ind` arrays have stayed the same, but the `row_ind` array has been replaced by the shorter `row_ptr` array. As the name suggests, the entries in the `row_ptr` array *point* into the other arrays at the location of the start of the next row. For instance, the second entry in `row_ptr` is 3, which tells us that the second row starts with the 3rd elements in the `val` and `col_ind` arrays.

Notice that the `row_ptr` array has length $m + 1$, or one more than the number of rows of A. The last entry in the `row_ptr` array is always one greater than the total number of nonzero entries in the matrix. This is convenient because, as you can see, the difference between consecutive entries in the `row_ptr` array tells us how many nonzeros there are in a give row.

For example, the difference between the second and third elements of `row_ptr` is 1, which tells us that there is one nonzero element in the second row of A. Similarly, the difference in the last two elements of row_ptr is 3 which tells us that there are three nonzero elements in the last row of A.

The CSR format is possibly the most common in computer science for storing nonstructured sparse matrices.

Compressed Sparse Column (CSC) Format

Of course, we could also store A in a completely analgagous column-oriented fashion, where nonzero entries in a column are stored contiguously is memory. In CSC format the given matrix is represented as

$$\begin{aligned} \texttt{val} &: \begin{bmatrix} 4 & 6 & 1 & 3 & 5 & 2 & 7 & 8 \end{bmatrix} \\ \texttt{row\_ind} &: \begin{bmatrix} 1 & 4 & 2 & 4 & 3 & 1 & 3 & 4 \end{bmatrix} \\ \texttt{col\_ptr} &: \begin{bmatrix} 1 & 3 & 5 & 6 & 9 \end{bmatrix} \end{aligned}$$

The CSC format is popular in many scripting languages. In particular, the default format for sparse matrices in both Julia and Matlab is CSC.

Other Formats

Of course, if you know that your matrix has a specific structure to it, it is often a good idea to tailor your sparse matrix format specifically to its structure. The obvious example is that of a diagonal matrix, where there is really no need to store the indices of the elements. There are similar storage formats for banded matrices. Another example is the case when you know your matrix is symmetric, it is only necessary to store the upper or lower triangular part of the matrix.

Another popular format that is particularly relevant to iterative methods for positive definite systems is called the *modified* compressed sparse row (MSR) format. Often times in iterative methods it is convenient to be able to access the diagonal element in a row before the off-diagonals. The MSR format uses only two arrays – one array of floats and one arrays of integers. For the example matrix A we have

$$\texttt{val} \;:\; \begin{bmatrix} 4 & 1 & 5 & 8 & * & 2 & 7 & 6 & 3 \end{bmatrix}$$

$$\texttt{ind} \;:\; \begin{bmatrix} 5 & 6 & 6 & 7 & 9 & 4 & 4 & 1 & 2 \end{bmatrix}$$

The first n entries of `val` store the diagonal entries of A, the $n+1$ entry is empty (or could possibly use for some other piece of information), and the remaining entries store the off-diagonals (ordered by rows). The first $n+1$ entries in the `ind` array act as pointers to the location in `val` of the first off-diagonal entry in each row. The remaining entries in `ind` are the column indices of the off-diagonal non-zero entries.

Working with Sparse Matrices

It is of course necessary to tailor our algorithms for common matrix computations to the specific sparse data structure that we have chosen. Here we will focus on the simple matrix-vector produce (or mat-vec) because it shares a lot of similarities with parts of the multigrid algorithm. For now we will assume that we've stored the matrix is CSR format.

Before we can think about the mat-vec in terms of sparse matrices, we need to know how it works for general dense matrices. Let A be an $m \times n$ matrix and $\mathbf{x}$ an n-length vector and suppose we want to compute $\mathbf{b} = A\mathbf{x}$. When we perform this operation by hand we typically compute the i^{th} element of $\mathbf{b}$ by taking the inner product of the i^{th} row of A with $\mathbf{x}$:

$$b_i = \sum_{j=1}^{n} a_{ij} x_j \quad \text{for } i = 1, \ldots, m$$

Suppose we represent A by an $m \times n$ two-dimensional array and $\mathbf{b}$ and $\mathbf{x}$ by m- and n-length one dimensional arrays, respectively. A computer code that computes the product $\mathbf{b} = A\mathbf{x}$ might look like:

for $i = 1 : m$ **do**
  $b[i] = 0.0$
  for $j = 1 : n$ **do**
    $b[i] \leftarrow b[i] + A[i,j] * x[j]$

The inner loop performs the inner product between $\mathbf{x}$ and the i^{th} row of A and the outer loop loops over each row of A and entry in $\mathbf{b}$.

Now think about the case when A is sparse. Most of the entries in a given row of A are zero, so including them in the inner loop across each row is a complete waste! So, how do we modify the matvec algorithm to work for a sparse CSR matrix?

First, assume we have the usual CSR arrays, namely `val`, `row_ptr`, and `col_ind`. The outer loop will still loop over the rows, but the inner loop will only access the elements of A in each row that are nonzero, as indicated by the `col_ind` array. We have

for $i = 1 : m$ **do**
  $b[i] = 0.0$
  for $j = row\_ptr[i] : row\_ptr[i+1] - 1$ **do**
    $b[i] \leftarrow b[i] + val[j] * x[col\_ind[j]]$

Deriving the mat-vec algorithm for the CSC matrix is similar, but this time we want the inner loop to take us down the columns of A instead of the rows.. First notice that you can think of a matrix vector product as a linear combination of the columns of A:

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} x_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} x_n = \sum_{j=1}^{n} A_j x_j$$

where A_j is the j^{th} column of A. Based on this organization, we can write down another algorithm for computing the dense matrix-vector product

Set b to zero
for $j = 1, \ldots, n$ **do**
  for $i = 1, \ldots, m$ **do**
    $b[i] \leftarrow b[i] + A[i,j] * x[j]$

Notice that the outer loop picks out a column of the matrix to work on, and the inner loop scans down that column and updates each entry of the $\mathbf{b}$ vector.

The CSC version of the mat-vec is then

```
Set b to zero
for j = 1 : n do
    for i = col_ptr[j] : col_ptr[j+1] − 1 do
        b[row_ind[i]] ← b[row_ind[i]] + val[i] ∗ x[j]
```

So Which Format is Better?

The answer to this question is of course, **it depends**. Algorithms that rely heavily on being able to traverse rows of the matrix would benefit from the use of the CSR format. On the other hand, the CSC format is better for computing mat-vecs on parallel machines.