Sparse Matrices

A **sparse** matrix is one in which most of the elements are zero. The opposite of one of these is a **dense** matrix, where most of the elements are nonzero. We can actually measure this, using sparsity and density:

Sparsity: Number of zero elements / number of elements
Density: Number of nonzero elements / number of elements

And we can also see that since these are both between 0 and 1
sparsity = 1 - density
density = 1 - sparsity

These happen all the time in CS and math, sometimes in scientific or engineering applications, but also in graph or network theory. For my uses, it seems most useful in graph theory for adjacency matrices. While adjacency matrices only have 1s and 0s, we can have applications in science and engineering which have nonzero elements other than 1, so we will talk about both.

Often times, these sparse matrices are huge. When it comes to adjacency matrices, this is especially the case, since they grow in size exponentially w.r..t. the number of nodes in the graph, e.g.

For n nodes, an adjacency matrix of the graph will be of shape (n^2 , n^2). This would subsequently have $n^2 * n^2 = n^4$ entries.

This can also be thought of as:

For an image of shape (n, m), an adjacency matrix of the image / graph will be of shape ($(n * m)^2$, $(n * m)^2$), which would subsequently have n^4m^4 entries.

So we see that adjacency matrices scale horribly in terms of space complexity. Because of this, it's often beneficial (if not necessary) to take advantage of the sparse structure of sparse matrices in order to store them more efficiently. There are quite a few for the general case that help a lot to reduce storage cost, while still allowing efficient compression and decompression.

Compression Algorithms

Special Cases

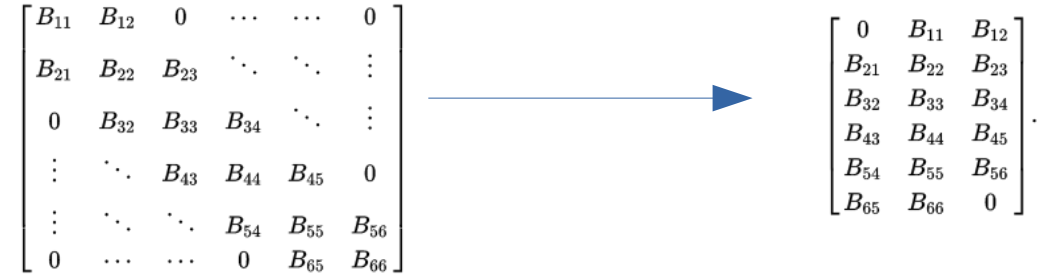
Diagonal: If a matrix only has nonzero elements on it's diagonal, it's pretty obvious that you can just compress it via putting the diagonal elements in a vector, going from a *(n, n) matrix -> (n,) vector*
This would likely work just fine with matrix / vector ops

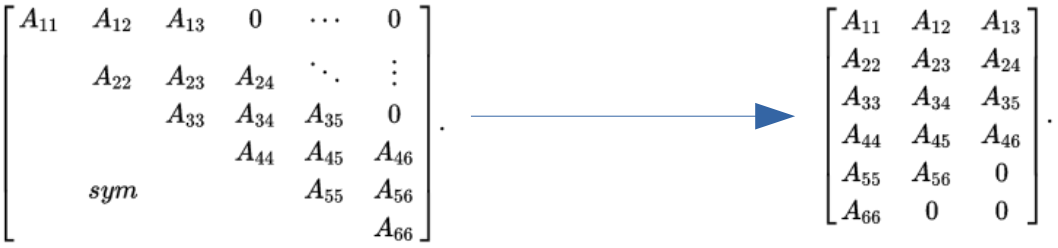
Symmetric: These happen as adjacency matrices of undirected graphs, and can be stored efficiently as an adjacency list.
This doesn't seem like it would be very efficient with matrix / vector ops, so it may be better to use a more general sparse matrix compression algorithm that is built-in to a library instead.

Banded: Not sure when these come up, but they are interesting. These are matrices where the non-zero entries are in a diagonal band, with zero or more entries in diagonals along the side of this main diagonal.

The applications of these, and examples of these, can be found on the relatively minimal wikipedia page: https://en.wikipedia.org/wiki/Band_matrix

But it's pretty obvious how to compress these, like these examples:





General Cases

Some of these offer efficient modification, some offer efficient access and matrix operations. The algorithms tend to be good at one but not the other, but this isn't too bad considering in graph theory you usually won't be modifying your graph once you're doing this - or at least you won't be modifying your graph that often in the case of a network structure.

There are more specifics on how to handle modification of graphs later in the wikipedia page: https://en.wikipedia.org/wiki/Sparse_matrix

Group 1 - Efficient Modification:

- DOK - Dictionary of Keys
- LIL - List of Lists
- COO - Coordinate List

Group 2 - Efficient access and matrix ops:

- CSR - Compressed Sparse Row
- CSC - Compressed Sparse Column

Dictonary of Keys - dictionary that maps (row, column) pairs to value of elements, only done on elements that are nonzero. While efficient for randomly generating sparse matrices (since you can just

randomly generate new keys and values), it is inefficient for going through the values in lexicographical (i.e. alphabetical or numerical) order, as imagined by the dictionary structure.

List of Lists - Stores one list per row, with each entry being the column index and value, e.g.:

[(1, 37), (4, 15)], [(2, 1)], ... [(2, 5)] for

[0, 37, 0, 0, 15]
[0, 0, 1, 0, 0]
[.]
[.]
[.]
[0, 0, 5, 0, 0]

This is efficient and inefficient in similar ways to DOK.

Coordinate List - Goes one step further from LIL and stores a list of just (row, column, value) tuples. Sorting by row index and then column index would also serve to improve access times, and this is another good format good for random construction.

Compressed Sparse Row - beginning of Group 2 Algorithms

This one is a pain in the ass. Pretty sure this is the most complicated one. It's similar to COO but compresses the row indices, which is why it's named that way.

It makes up for it's more complex structure than the group 2 formats by allowing for fast row access and matrix - vector multiplications (meaning you can multiply the matrix by a vector).

This is much easier explained by example:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

A = [5, 8, 3, 6]
IA = [0, 0, 2, 3, 4]
JA = [0, 1, 2, 1]

This always gives three vectors as the result, and the arbitrary names given to each of these tend to differ. However, it makes sense to think about it like this:

> *A = name of matrix, gives the values*
> *IA -> i is used for row indices, and I here is for the rows*
> *JA -> j is used for col indices, and J here is for the columns*

So I like the names here. Let's break down each of these vectors:

> A = the values in the matrix, obtained by going left-right top-bottom through it.
> IA = Starts off as just [0], and for each row *i*,
> *IA[i] = IA[i-1] + Number of nonzero elements in this row*

So we can see how for this matrix, we ended up with [0, 0, 2, 3, 4]:
[0], because we start with [0],
[0, 0], because first row has nothing,
[0, 0, 2], second row has 2 entries
[0, 0, 2, 3], third row has 1 entry
[0, 0, 2, 3, 4], fourth row has 1 entry.

A convenient side effect of this is that the last element in IA is always the number of nonzero elements in the matrix.

JA = the column indices of each element in A, as we encounter them. So 5: 0, 8: 1, 3: 2, 6: 1

While this looks kinda large, it ends up saving tons of space the larger it gets. However, if you're super picky, you can find when it saves on memory when
NNZ (number of non-zero elements) < (m(n-1)-1)/2

We can also compute the length of each row using IA[i+1] - IA[i] for a row i

We can use this to regenerate the array like so:
A gives values
IA splits values into rows: [0], [5, 8], [3], [6]
JA puts values in appropriate locations: [0, 0, 0, 0], [5, 8, 0, 0], [0, 0, 3, 0], [0, 6, 0, 0]

Which is nice, because it means we can do this one row at a time in our sparse matrix, and if we are doing a matrix-vector multiplication it means we can take that row and multiply it by the appropriate vector or scalar, depending on if we are multiplying a row or column vector. We can then store the result as either a scalar or a new row in a new sparse matrix (or just a normal matrix), and then go on to the next row and repeat.

Obviously this is automated in libraries already, but now you understand how it works. This is what makes it efficient for matrix-vector multiplications, as opposed to the group 1 algorithms which would require iterating through a ton in order to generate the row.

Compressed Sparse Column (CSC / CCS) - This is a similar idea to CSR (hence the name), and is an alternative to COO. Instead of looping left-right top-bottom as in CSR, we loop top-bottom left-right. So, our new example would now be:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

val = [5, 8, 3, 6]
row_ind = [1, 1, 2, 3]
col_ptr = [0, 1, 3, 4]

I did mention the notation often differed, and this is the common notation used for these three matrices in CSC.

We see that we get col_ptr the same way we got IA, except we sum over the number of non zero elements in the columns instead of rows. We can also see that row_ind is the same as JA, except for row indices instead of column indices. So it's not much different.

The CSC format is equivalent to the CSR format, for A^T

There isn't much gain in using one or the other, they both have about the same efficiency on arithmetic and matrix-vector ops for the same reasons that I described in the CSR section. MATLAB seems to use this by default, but scipy offers support for every method i've described here.

So yea, scipy supports all of these: https://docs.scipy.org/doc/scipy-0.19.0/reference/sparse.html

Good luck, have fun

-DE