1 Introduction to Sparse Matrices

Sparsity is a concept that appears in numerical linear algebra. Many matrices that appear in practice have few non-zero entries, and such matrices are called *sparse*. For example, we could have a transition matrix where, given a state, only a low ratio of states are available to transition to. A matrix that is not sparse is called *dense*. Note that "few" is not precisely quantified: indeed, sparsity is often not given a precise definition, and what constitutes few elements varies depending on use-case.

Sparse matrices have highly exploitable structure, improving both space and time efficiency. Firstly, we only have to store non-zero matrix entries. We may then have a 100×100 matrix that we can store as a size 100 dictionary (with keys representing the non-zero positions) as opposed to a length 100 list, with each list having length 100. Matrices much larger than 100×100 are common, and hence a significant saving of space is evident. Secondly, sparsity allows quicker matrix calculations, especially of products. Given two sparse matrices and knowledge of where they are non-zero, we can quickly determine which entries in the product are non-zero, and discard zero terms in the eventual sum.

The above discussion is moreso referring to sparse matrices that are finite. In this PDF and **SparseMatrices.py**, we use sparsity to encode and do calculations on infinite matrices. We can see that, more than delivering reductions in runtime, sparsity is necessary to perform product calculations. Take infinite matrices $A = (a_{ij})$ and $B = (b_{ij})$. Suppose we have represented A and B as callable objects which accept arguments (i, j) and returns a_{ij} and b_{ij} respectively. Given that $AB = (c_{ij})$ is defined (for example, these infinite matrices represent bounded operators in $\ell_2(\mathbf{N})$, the space of square integrable sequences), we have $c_{ij} = \sum_{k=0}^{\infty} a_{ik} b_{kj}$. It is not possible to calculate or estimate this sum in finite time without more information about A and B. The concept of sparsity that we discuss will reduce this infinite sum to a finite sum, and hence enable the calculation of c_{ij} given i and j.

We discuss the following definition of sparse due to Colbrook.

Definition 1.1. Let $A = (a_{ij})$ be a computable matrix. That is, we have represented A as a **Callable** in Python which accepts arguments i, j and gives a_{ij}. We call A *sparse* if:

1. every row $(a_{ij})_j$ (i fixed) of A and every column $(a_{ij})_i$ (j fixed) of A has only finitely many non-zero entries
2. we possess a Python function $f : \mathbf{N} \to \mathbf{N}$ satisfying $a_{ij} = 0$ whenever $i > f(j)$ or $j > f(i)$. Note that this condition implies condition (1).

Not only do we know that row i has finitely many non-zero entries, we can also compute an upper bound on where non-zero entries can lie. This gives us a finite window to investigate for non-zero entries. This allows for the computation of the product of sparse matrices, as will be discussed in the following section.

2 matmul

Here we discuss the implementation of matmul found in **SparseMatrices.py**. Let's suppose we have implemented $A = (a_{ij})$ and $B = (b_{ij})$ as **SparseMatrix** objects with associated "f" functions f_A and f_B. Let's write the matrix product AB as (c_{ij}). We can write:

$$c_{ij} = \sum_{k=0}^{\infty} a_{ik} b_{kj}.$$

We write c_{ij} in two different ways. First, we note that if $k > f_A(i)$, we have $a_{ik} = 0$. Hence $a_{ik} b_{kj} = 0$ for $k > f_A(i)$. Consequently, we only have to sum from $k = 0$ to $k = f_A(i)$, since we know all terms beyond this are zero. We do not discount the possibility that the term corresponding to $k = f_A(i)$ can also be zero, but we now have a finite upper bound. We can repeat the same process looking instead at b_{kj} to conclude that $a_{ik} b_{kj} = 0$ for $k > f_B(j)$.

We now have two expressions for c_{ij}:

$$c_{ij} = \sum_{k=0}^{f_A(i)} a_{ik} b_{kj}$$

and:

$$c_{ij} = \sum_{k=0}^{f_B(j)} a_{ik} b_{kj}.$$

Since $|c_{ij}| < \infty$ for each i, j, we can see that the matrix product AB is well-defined. First, we need to show that every row and column of AB has only finitely many non-zero entries. We fix i and look at the expression:

$$c_{ij} = \sum_{k=0}^{f_A(i)} a_{ik} b_{kj}$$

For fixed k, we have $b_{kj} = 0$ if $j > f_B(k)$. If $j > f_B(k)$ for all $0 \leq k \leq f_A(i)$, we have $b_{kj} = 0$ for all $0 \leq k \leq f_A(i)$, and hence $c_{ij} = 0$. Hence, if we set $g_1(i) = \max\{f_B(k) : 0 \leq k \leq f_A(i)\}$, we have $c_{ij} = 0$.

Similarly, fixing j and writing:

$$c_{ij} = \sum_{k=0}^{f_B(j)} a_{ik} b_{kj}.$$

we have $c_{ij} = 0$ for $i > g_2(j) := \max\{f_A(k) : 0 \leq k \leq f_B(j)\}$. Writing $f_{AB}(i) = \max\{g_1(i), g_2(i)\}$, we have $c_{ij} = 0$ whenever $i > f_{AB}(j)$ or $j > f_{AB}(i)$.

We are almost ready to trace our implementation of matmul. We have two expressions for c_{ij}, namely:

$$c_{ij} = \sum_{k=0}^{f_A(i)} a_{ik} b_{kj}$$

and:

$$c_{ij} = \sum_{k=0}^{f_B(j)} a_{ik} b_{kj}.$$

These sums are summing over the same sequence, merely stopping at a different k. In the interest of time efficiency, we should stop at the smaller k. So we use the formula:

$$c_{ij} = \sum_{k=0}^{\min\{f_A(i), f_B(j)\}} a_{ik} b_{kj}.$$

We then have the algorithm:

1. define $g_1(i) = \max\{f_B(k) : 0 \leq k \leq f_A(i)\}$
2. define $g_2(i) = \max\{f_A(k) : 0 \leq k \leq f_B(i)\}$
3. define $g(i) = \max\{g_1(i), g_2(i)\}$
4. define a function $c(i, j) = \sum_{k=0}^{\min\{f_A(i), f_B(j)\}} a_{ik} b_{kj}$
5. create a new **DiagonalMatrix** object using the callable c and $f_{AB} = g$, and return it.