Chapter 4

Matrix Operations

Matrix operations are very useful and common in many areas of engineering and sciences and are computationally expensive– and therefore, they are of interest to computer science.

You may have encountered matrices already, but since knowledge of matrices is not listed as a prerequisite of this course, I do not assume that this is the case. The first few sections of this chapter define what a matrix is (at least in the context of S-Q) and some of the basic matrix operations and their properties. Even if you are already familiar with matrices, you might find it useful to skim these sections in order to familiarize yourself with the notation.

4.1 Terms and Definitions

In general, a *matrix* is a 2-dimensional array, together with some operations, such as multiplication and addition, defined for the array.

For the sake of simplicity, this chapter assumes that all the matrices are n-by-n "square" arrays: the number of rows and columns are the same. Generalizing the ideas to work with rectangular arrays is left as an exercise.

Note: the subscripts or indices of the elements of the matrix range from $(0, 0)$ (at the upper left corner of the array) to $(n-1, n-1)$ (at the lower right corner). This is somewhat different than you might see in mathematical texts, where the subscripts start and 1 and end at n. I have made this change in order to make it easier to understand matrices in the context of C's zero-based arrays, but you should be aware that my notation is different than what you might see elsewhere.

For example, the C array written `int A[2][2]` would be written as matrix

$$A = \begin{bmatrix} a_{0,0}, a_{0,1} \\ a_{1,0}, a_{1,1} \end{bmatrix}$$

The matrix element denoted as $a_{i,j}$ corresponds to the C array element `A[i][j]`.

4.2 Matrix Addition and Multiplication

4.2.1 Addition

Addition of two matrices is very simple: the sum of two matrices is the matrix composed of the sums of elements of the two matrices:

$$A + B = \begin{bmatrix} (a_{0,0} + b_{0,0}) & \cdots & (a_{0,n-1} + b_{0,n-1}) \\ \vdots & (a_{i,j} + b_{i,j}) & \vdots \\ (a_{n-1,0} + b_{n-1,0}) & \cdots & (a_{n-1,n-1} + b_{n-1,n-1}) \end{bmatrix}$$

This definition is expressed in C in algorithm 4.1.

Algorithm 4.1 (Matrix Addition) *An algorithm to add two n-by-n matrices A and B, storing the result in n-by-n matrix C.*

```
void    mm_add (int A[N][N], int B[N][N], int C[N][N])
{
        int i, j;

        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        C [i][j] = A [i][j] + B [i][j];
                }
        }
}
```

Algorithm 4.1 is clearly $O(n^2)$.

4.2.2 Multiplication

The algorithm for multiplication is quite a bit different than might be expected– instead of simply multiplying the elements together, each element of the product is the result of several operations, as described in algorithm 4.2.

Algorithm 4.2 (Matrix Multiplication) *An algorithm to multiply two n-by-n matrices A and B, storing the result in matrix C. This algorithm assumes that C is not the same matrix as A or B (if A or B are the same array as C, the wrong result is likely because the contents of C are changed during the calculation).*

```
void    mm_mul (int A[N][N], int B[N][N], int C[N][N])
{
        int i, j, k;
        int sum;

        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        sum = 0;
                        for (k = 0; k < N; k++) {
                                sum += A[i][k] * B[k][j];
                        }
                        C[i][j] = sum;
                }
        }
}
```

Algorithm 4.2 is clearly $O(n^3)$.

4.3 Faster Matrix Multiplication Algorithms

Algorithm 4.2, which defines the operation of matrix multiplication for square matrices, is clearly $O(n^3)$. Since there are many algorithms that use matrix multiplication, often on relatively large matrices, this can have a serious effect on the speed of the programs that use these algorithms. For this reason, many algorithms that use matrix multiplication use various tricks which take advantage of some particular attribute of the specific matrices used by that algorithm. (We'll see an example of one such trick in section 4.4.)

However, for many years it was assumed that general matrix multiplication was an intrinsically $O(n^3)$ operation, so despite the benefits that a faster general multiplication algorithm would have, there was not a lot of research dedicated to finding one. However, in 1969 Strassen discovered an $O(n^{2.81})$ algorithm[1]. This may seem like a small improvement (and in fact, it does not really translate into real savings unless the matrices are large), but it was an astonishing result that spurred a great deal of research in this area. To the best of my knowledge, however, nobody has yet proven what the big-O of the fastest possible general matrix multiplication algorithm is[2]. Perhaps there are some useful and theoretically important discoveries still to be made.

4.4 Operations on Sparse Matrices

As implied earlier, there are tricks that can be used to speed up matrix multiplication if the matrices are known to have particular properties.

One kind of matrix that can be multiplied quickly is a sparse matrix. A *sparse matrix*, like a sparse array, is a matrix where most of the elements are the same value. In most cases, the common value is zero, but this is not necessarily the case. In this section, we'll begin by exploring how sparse matrices that are mostly zero can be multiplied quickly, but at the end we will see how the principles for handling these matrices can also be used to speed up operations on other kinds of sparse matrices.

4.4.1 Multiplying Sparse Matrices

A simple-minded approach to matrix multiplication is to blindly use algorithm 4.2 again, but instead of using the ordinary array operations to access to matrix elements, use the sparse array (or multilist) operations to access them:

```
void     mm_mul_sa (sa_t *A, sa_t *B, sa_t *C)
{
         int i, j, k;
         int sum;
```

[1]Strassen's algorithm is described in section 10.2.4, pages 378-380 of the Weiss book.

[2]Of course, it can't be faster than $O(n^2)$, since an n-by-n matrix has n^2 elements, and each has to be visited once just to fill in the answer– but how close to $O(n^2)$ can we get?

```

        for (i = 0; i < A->num_rows; i++) {
                for (j = 0; j < B->num_cols; j++) {
                        sum = 0;
                        for (k = 0; k < A->num_cols; k++) {
                                sum += saGet (A,i,k) * saGet (B,k,j);
                        }
                        saSet (C,i,j,sum);
                }
        }
}
```

This is a step in the wrong direction, however: since `saGet` and `saSet` are generally $O(n)$ operations themselves, the resulting algorithm is $O(n^4)$ in the worst case.

However, there is a way to speed this up considerably by changing the inner loop (in this code, the `k` loop) to take advantage of two key facts:

- The product of any number and zero is zero. Therefore, if we know that either $A[i][k]$ or $B[k][j]$ is zero, we can skip the multiplication and addition.
- The loop strides through the actual (non-zero) elements in the array in the same order that they appear in the linked lists that are used to represent the array. Therefore, instead of starting at the base of each list (as `saGet` and `saSet` must do), we can keep track of where we are in each list, so that we can find the next element very quickly.

The inner loop of algorithm 4.3 avoids trying each possible k (as algorithm 4.2 did) by moving down both the row and column lists in tandem, searching for elements in the row list that have the same index as elements in the column list. Since the lists are kept in order by index, we know that we can do this in one scan through the lists. Each iteration through the loop moves forward at least one position in one of the lists, so the loop terminates after at most $R + C$ iterations (where R is the number of elements in the row list and C the number of elements in the column list).

4.4.2 Another Note About Software Engineering

Implementing the algorithm this way (i.e. writing this function in a way that depends on so many details of the `sa_t` structure is clearly another abuse of software engineering principles. See section 3.5 for more discussion.

Algorithm 4.3 (Sparse Matrix Multiplication) *An algorithm to multiply two sparse* n*-by-*n *matrices* A *and* B*, storing the result in matrix* C*. This algorithm assumes that* C *is a different matrix than both* A *and* B*, and that* C *is initially empty (all its elements are zero).*

This algorithm is identical to algorithm 4.2, except for the inner block, which is changed as follows:

```
void    mm_mul_sa_fast (sa_t *A, sa_t *B, sa_t *C)
{
        int i, j, sum;
        sa_cell_t *row_p, *col_p;

        for (i = 0; i < A->num_rows; i++) {
                for (j = 0; j < B->num_cols; j++) {
                        row_p = A->rows [i];
                        col_p = B->cols [j];
                        sum = 0;

                        while ((row_p != NULL) && (col_p != NULL)) {
                                if (row_p->index == col_p->index) {
                                        sum += row_p->value * col_p->value;
                                        row_p = row_p->next;
                                        col_p = col_p->next;
                                }
                                else if (row_p->index < col_p->index)
                                        row_p = row_p->next;
                                else
                                        col_p = col_p->next;
                        }
                        saSet (C, i, j, sum);
                }
        }
}
```

4.4.3 Other Sparse Matrices

As promised, we will now extend the techniques we have developed for multiplying sparse matrices where the common value is zero to handle sparse matrices where the common value is some other number.

Let A be a sparse matrix where most of the elements have value α, and B be a sparse matrix where most of the elements have a value of zero. Our goal is compute AB efficiently.

A can be written as the sum of two matrices: one where all of the elements are α, and the other the difference between this α matrix and A:

$$A = \begin{bmatrix} a_{0,0} - \alpha, & \cdots, & a_{0,n-1} - \alpha \\ \vdots & a_{i,j}, -\alpha & \vdots \\ a_{n-1,0} - \alpha, & \cdots, & a_{n-1,n-1} - \alpha \end{bmatrix} + \begin{bmatrix} \alpha, & \cdots, & \alpha \\ \vdots, & \alpha, & \vdots \\ \alpha, & \cdots, & \alpha \end{bmatrix}$$

We'll denote the matrix that results from subtracting α from every element of A as $A_{a-\alpha}$, and the matrix that consists entirely of αs as A_α.

This is useful, because matrix multiplication is distributive:

$$\begin{aligned} X(Y+Z) &= XY + XZ \\ (X+Y)Z &= XZ + YZ \end{aligned}$$

This property follows from the definition of matrix multiplication; the proof is left as an exercise.

However, matrix multiplication is not commutative: in general $XY \neq YX$. Therefore, it is very important to get the order right when we do multiplication!

Therefore, $AB = (A_{a-\alpha} + A_\alpha)B = A_{a-\alpha}B + A_\alpha B$.

At this point, it seems like we're making more work for ourselves, because we are now doing two matrix multiplications and then adding the results together. However, with a little more work we can simplify this calculation enormously.

Since $A_{a-\alpha}$ and B are both sparse matrices with a common value of zero, the product $A_{a-\alpha}B$ can be computed efficiently using the algorithm described in the previous section.

Since every row in A_α is identical, every row in the product of $A_\alpha B$ must also be identical as well. In fact, each element in the result is αn multiplied by the sum of the elements in the corresponding column of B. Therefore, the product $A_\alpha B$ is

$$A_\alpha B = \begin{bmatrix} \alpha n \sum_{j=0}^{j=n-1} b_{0,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{i,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{n-1,j} \\ \vdots & \vdots & \vdots \\ \alpha n \sum_{j=0}^{j=n-1} b_{0,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{i,j}, \cdots, & \alpha n \sum_{j=0}^{j=n-1} b_{n-1,j} \end{bmatrix}$$

The sum for computing each value of the first row takes $O(n)$, but once the first row is finished, each other row is just a copy of the first. Therefore, this multiplication is $O(n^2)$.

Computing the order of the entire calculation depends upon the "sparseness" of A and B. If they are very sparse, and the sparse array multiplication algorithm from the previous section is modified to take full advantage of this (see the exercises for more information), then the multiplication of $A_{a-\alpha}B$ can be done faster than $O(n^2)$, so that the $O(n^2)$ addition at the end of the calculation dominates the order of calculation. If A and B are not so sparse, then this multiplication can be $O(n^3)$.

4.5 Exercises

1. Another important optimization that can be used to improve algorithm 4.3 is to skip over rows or columns that are entirely zero.

   For example, in the current algorithm, even if a row is completely zero, the inner two loops are run over it anyway, even though no non-zero values can possibly result– $O(n^2)$ work that accomplishes nothing. In the very worst case, multiplying two sparse matrices that are both entirely zero will require $O(n^3)$ operations (or $O(n^2)$, if the solution to the previous problem is used).

   Improve the algorithm so that it handles rows or columns that are entirely zero in a more efficient manner. Discuss any tradeoffs that you make between efficiency and simplicity.

2. Extend the ideas in section 4.4.3 to work when B is a sparse array with common value β (where $\beta \neq 0$).