Chapter 3

Sparse Arrays

This chapter explores sparse arrays. A *sparse array* is an array where nearly all of the elements have the same value (usually zero), and this value is a constant, known ahead of time.

In many fields of mathematics and engineering, sparse arrays (in particular, arrays that consist almost entirely of zeros) are quite common. Therefore, it is worthwhile to develop efficient methods for handling such arrays.

In this chapter, we'll explore one way to efficiently represent sparse arrays, and in chapter 4 we'll see how this representation can pay huge dividends, both in terms of space and time.

Note that the examples given in this chapter are all for two-dimensional arrays. However, it is possible to generalize the ideas presented here for arrays of higher dimension; this is left as an exercise.

3.1 A Troublesome Property of C Arrays

In C (and many other languages), space for arrays is typically allocated ahead of time, during compilation, and enough space is allocated to represent *all* of the elements in the array. For example, consider the following variable definition:

```
double bigArray[1000][1000];
```

Defining `bigArray` allocates enough space to hold 1,000,000 doubles. On the HP workstations used by this course, each `double` requires 8 bytes of memory, so this array requires 8 million bytes of memory. Even if we only use a few elements of this array, defining this variable will soak up all of this space. This is OK if we are going to actually use all of this space– but in many cases, we can do a lot better.

C's allocation method is very wasteful in the case of sparse arrays, where we know what most of the values are going to be ahead of time– it is quite wasteful to allocate vast amounts of memory to hold data that we'll never change and whose value we already know!

3.2 Linked List Representation

As we have already seen, one method for representing a 1-dimensional array is as a linked list. We can extend this representation to deal with 2-dimensional arrays by treating each 2-dimensional array as a 1-dimensional array of rows, where each row is in turn a 1-dimensional array. Each "row" array can be implemented as an ordered linked list (where the order is determined by the column of each element). If an element does not appear in the linked list, then it is assumed to have the default value.

Consider the 4-by-4 array illustrated in figure 3.1. This array consists of 12 zero elements, and 4 non-zero elements. Compared to most sparse arrays, this has a relatively high proportion of non-zero elements, but it will serve to illustrate the principles involved.

Figure 3.1: An ordinary C array.

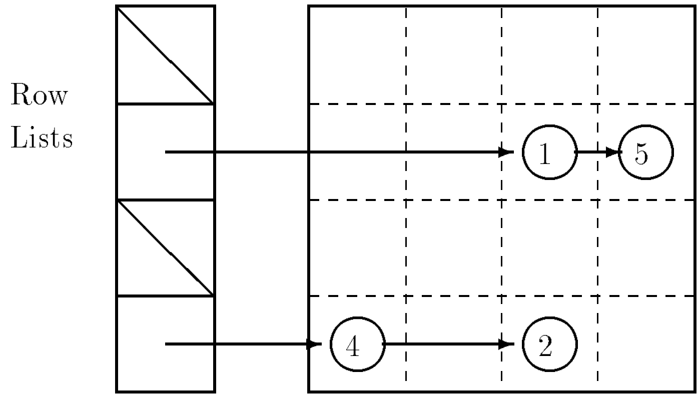
0	0	0	0
0	0	1	5
0	0	0	0
4	0	2	0

An illustration of a sparse array that represents the same array as the array in figure 3.1 is shown in figure 3.2.

The method of accessing elements in a sparse array implemented this way is very simple: find the row list for the row of the element, and search it for the elements in the correct column. If the element is not present, then we simply assume that the value of that element is zero.

Depending on how we access the elements of the array, it may be more appropriate to organize the array by row or by column. For example, if we represent our sparse array

Figure 3.2: A simple sparse array.



with an array of rows (where each row is a linked list), then determining whether or not a particular row is entirely empty is an $O(1)$ operation– either the pointer in the row array is NULL, or it isn't. Determining whether or not a particular column is empty, however, requires checking each row for an element in that column, a potentially $O(n)$ operation.

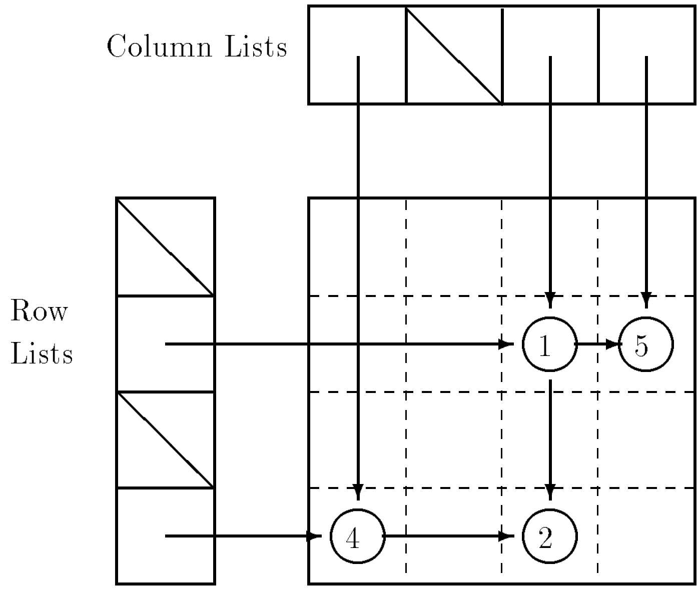
Of course, it is possible to represent the array using both methods: an array of row lists, and an array of column lists. An example of this is illustrated in figure 3.3, which once again represents the same array as in figure 3.2.

Deciding which representation to use is an important issue. Having both row and column lists makes it easy to traverse the array in both row and column order (and as we'll see in chapter 4, this can be quite useful), but it makes other operations more difficult. For example, if row lists are stored, then exchanging two rows of an array[1] is trivial (simply exchanging the pointers to the first element in each row list is sufficient). Storing both row and column lists makes swapping rows more difficult, however, since exchanging two rows may defeat the ordering of the columns.

As always, knowing how the array will be used is essential to picking the optimal representation.

[1]Row exchange is an essential part of some important mathematical algorithms, such as Gaussian Elimination. We won't study these algorithms in this course, but it is useful to remember that row exchange is something that can be important.

Figure 3.3: A sparse array that uses both row and column lists.



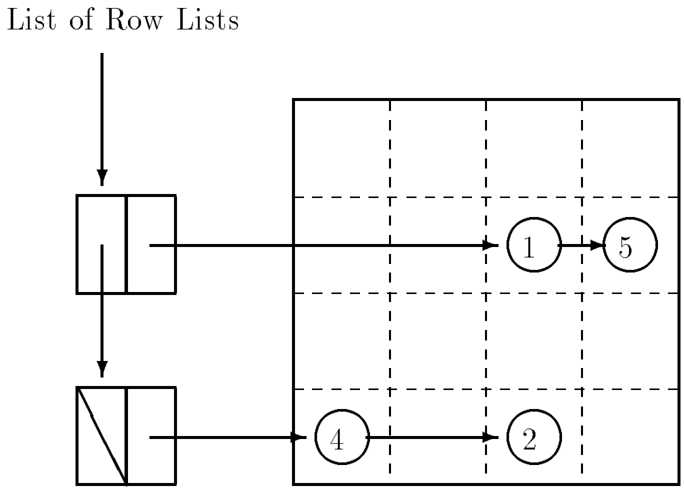
3.2.1 Extremely Sparse Arrays

If most of the rows of the sparse array do not contain any elements, then the array of pointers to row lists is itself a sparse array, and can also be implemented as a linked list. An illustration of this is shown in figure 3.4, which represents the same arrays as figure 3.2, but using a linked list to represent the row array.

From an implementation perspective, adding this is slightly more clumsy than it might appear, because the linked list that represents the row array is a linked list of linked lists, while the linked list that represents each row itself is a linked list of whatever type of element the array actually represents. This means that (at least in a language like C), the linked list routines we use for the rows can't be used for the row array, because of this type mismatch.

If the number of rows that actually contain any elements is small, then this approach can offer large space savings– but if many of the rows are occupied, then as much space can be consumed by this representation as the array. In addition, the access time is considerably slower, because we've replaced an $O(1)$ array lookup with a (potentially) $O(n)$ linked list traversal.

Figure 3.4: A sparse array where the array of rows is replaced by a linked list.



3.3 An Implementation

The code in `sa.c` (with header file `sa.h`) provides a relatively simple and straightforward implementation of sparse arrays. (The entire source for this library is available online, and is not included here.)

Note that the representation that `sa` uses includes both row and column lists; it could be somewhat simpler if only rows or columns were used.

Also, as an extra feature, the default value of the elements in the array is not assumed to be zero. Instead, this value can be specified when the array is created.

```
/*
 * An "sa_cell_t" is a simple singly-linked list
 * structure.  Each cell represents an element of
 * the array, and each list represents a single
 * row or column of the array.
 */

typedef struct _sa_cell_t {
        struct _sa_cell_t *next; /* pointer to next.    */
```

```
        int     index;          /* index in row or col. */
        int     value;          /* value of the elem.   */
} sa_cell_t;

/*
 * An sa_t implements a sparse array.  Note that
 * this implementation includes both row and column
 * lists.
 */

typedef struct  {
        int     num_rows;       /* # of rows in array.  */
        int     num_cols;       /* # of columns.        */
        sa_cell_t **rows;       /* array of row lists.  */
        sa_cell_t **cols;       /* array of col lists.  */
        int     default_val;    /* default elem value.  */
} sa_t;

```

3.3.1 An Optimization: Taking Advantage of Order

Consider the problem of displaying the contents of a sparse array by printing them on the screen. The obvious algorithm is to print the array in the ordinary manner, entirely ignoring the fact that the array is sparse. An implementation of this algorithm is given by function `saPrint`, shown below.

```
void            saPrint (sa_t *sa)
{
        int             i, j;

        for (i = 0; i < sa->num_rows; i++) {
                for (j = 0; j < sa->num_cols; j++) {
                        printf ("%4d ", saGet (sa, i, j));
                }
                printf ("\n");
        }
}
```

Ideally, the cost of printing an n-by-n array is $O(n^2)$, since every element in the array must be printed, and there are n^2 elements. Unfortunately, this method is inefficient because of the cost of the saGet function (which is used to fetch the value of each element in the array). In the worst case, saGet is $O(n)$, so saPrint has a worst-case running time of $O(n^3)$.

However, we can take advantage of the ordered property of the lists in order to speed this up quite a bit. Since the order we wish to print the elements in is the same as the order that they appear in the lists, there is no reason to start over at the head of the list (as saGet must do) each time we need to access an element. Instead, we can keep track of the location of the last element that we visited, and begin our search there.

Alternatively, we can take even more advantage of the fact that the row lists are kept in order. We can simply traverse each row list, looking ahead to see where the next non-zero element is and printing out the appropriate number of zeros between. This principle is illustrated in saPrintFast.

```
void            saPrintFast (sa_t *sa)
{
        int             i, j;
        sa_cell_t       *cursor;

        for (i = 0; i < sa->num_rows; i++) {
                cursor = sa->rows [i];
                for (j = 0; j < sa->num_cols; j++) {
                        if ((cursor == NULL) || (cursor->index > j)) {
                                printf ("%4d", sa->default_val);
                        }
                        else {
                                printf ("%4d", cursor->value);
                                cursor = cursor->next;
                        }
                        printf (" ");
                }

                printf ("\n");
        }

        return ;
}
```

3.3.2 Cursors

In `saPrintFast`, `cursor` is used to keep track of where we are in the list. In general, a *cursor* is a way to keep track of an "interesting" location in a data structure, so that this location doesn't need to be recomputed again. In this case, we keep track of where we are in the list so we don't have to traverse to the same spot again.

The notion of a cursor is also mentioned in Weiss, section 3.2.8, in a somewhat different context– but the idea is the same. Instead of searching the `CursorSpace` array for the index of the unused element, the index of an available element (if there are any) is always stored in the first location of the array.

3.4 How Space-Efficient Are Sparse Arrays?

In the implementation shown in `sa.c`, on the HP workstations (which use 4-byte integers and 4-byte pointers) the space requirements of an n by n sparse array containing m non-zero elements are:

- `sizeof (sa_t)` = 20 bytes.
- `2 * sizeof (sa_cell_t *) * n` = $8n$ bytes ($4m$ each for the `rows` and `cols` arrays).
- `2 * sizeof (sa_cell_t) * m` = $24m$ bytes for the m values.
- The `malloc` library may need to allocate some extra space for its own use.

As we can see, this representation is not worthwhile unless the arrays are indeed fairly sparse: each element requires at least 6 times as much memory to represent as it would in an ordinary array. (If only row or column lists are maintained, then the requirements are cut in half, but this may have a profound effect on the running time of algorithms that access the array.)

3.5 A Note About Software Engineering

This course focuses on algorithms, and how they can be implemented. Unfortunately, there isn't time to also focus equally on the software engineering aspects of how these algorithms are implemented in C.

If you have taken a course in software engineering (or have been introduced to these principles in an introductory programming course or from actual work experience), then you probably realized that the implementation of cursors shown above is terrible from the perspective of software engineering. It makes assumptions about the nature of the underlying `sa_t` and `sa_cell_t` structures that should be hidden by a layer of abstraction. A small change to the `sa_cell_t` structure could easily break this code.

In most cases, would be better to implement the idea of cursors as part of the `sa` library itself, so that functions that use this library do not depend on picayune details of the actual structures used by the library. The underlying algorithms will be entirely the same, of course, but the code would be more abstract and more reusable. In this text, however, I will usually sacrifice software engineering principles in favor of clean and concrete algorithms.

3.6 Exercises

1. In the sparse array representation used in this chapter, the linked lists are always kept in order by index. Keeping these lists in order requires a small amount of extra work during insertion, but can also save work during lookup. Discuss the tradeoffs between keeping the lists ordered versus unordered. List any situations you can imagine where one or the other method is troublesome or inefficient.

2. Devise an efficient algorithm that finds the intersection of two ordered linked lists (i.e. the list of values that appear in both lists) and prints it in order. The algorithm must visit each node in both lists *at most* once.

   Hint– consider the use of a cursor.

3. In many contexts, it is appropriate for uninitialized elements of a sparse array to be treated as having an undefined value, instead of having the default value. In such an array, for example, using `saGet` on an element that hadn't been assigned a value with `saSet` would be considered an error. What changes (if any) to the `sa` library would be necessary to implement this behavior?

4. Implement a library, similar to `sa`, that implements three-dimensional sparse arrays.

5. As mentioned in section 3.5, some of the `sa` code could be rewritten in a much better manner, with a cleaner and more abstract interface. Starting with the current `sa` library, define a cleaner interface for sparse arrays, and implement the corresponding library. Be sure to include cursors in your library.