DATA STRUCTURES AND APPLICATIONS

Module 1_4

Introduction

Dr. Pushpalatha K

Associate Professor

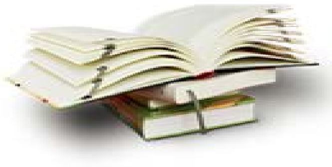
Sahyadri College of Engineering & Management

SPARSE MATRICES

- A matrix which contains many zero entries or very few non-zero entries is called as **Sparse matrix**
- A sparse matrix can be represented in 1-Dimension, 2- Dimension and 3-Dimensional array.
- When a sparse matrix is represented as a two-dimensional array more space is wasted.

$$
\begin{array}{cc}
 & \begin{array}{cccccc} \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col 5} \end{array} \\
\begin{array}{c} \text{row0} \\ \text{row1} \\ \text{row2} \\ \text{row3} \\ \text{row4} \\ \text{row5} \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

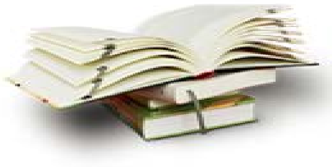
This matrix contains only 8 of 36 elements are nonzero and that is sparse



- **ADT Sparse_Matrix is**
  - **objects**: a set of triples, <row, column, value>, where row and column are integers and form a unique combination, and value comes from the set item.
  - **functions**:
  - for all a, b $\in$ Sparse_Matrix, x $\in$ item, i, j, max_col, max_row $\in$ index
    - **Sparse_Marix Create(max_row, max_col)** ::= return a Sparse_matrix that can hold up to max_items = max _row X max_col and whose maximum row size is max_row and whose maximum column size is max_col.
    - **Sparse_Matrix Transpose(a)** ::= return the matrix produced by interchanging the row and column value of every triple.

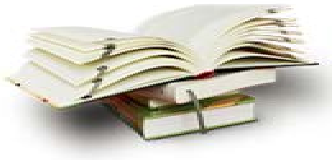
- **Sparse_Matrix Add(a, b)** ::= if the dimensions of a and b are the same return the matrix produced by adding corresponding items, namely those with identical row and column values. else return error

- **Sparse_Matrix Multiply(a, b)** ::= if number of columns in a equals number of rows in b return the matrix d produced by multiplying a by b according to the formula: $d [i] [j] = \sum(a[i][k] \bullet b[k][j])$ where d (i, j) is the (i,j)th element else return error.



Sparse Matrix Representation

- An element within a matrix can characterize by using the triple <row,col,value> This means that, an array of triples is used to represent a sparse matrix.
- Organize the triples so that the row indices are in ascending order.
- The operations should terminate, so we must know the number of rows and columns, and the number of nonzero elements in the matrix.



- **Implementation of the Create operation as below:**

SparseMatrix Create(maxRow, maxCol) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

- **The first row of the sparse matrix stores the following information**
  - **Location of[0,0] stores the row size of the original matrix**
  - **Location of [0,1] stores the column size of the original matrix**
  - **Location of [0,2] stores the number non zero entries of original matrix.**

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

Sparse Representation of

$$
\begin{array}{c|cccccc|}
 & col0 & col1 & col2 & col3 & col4 & col5 \\
\hline
row0 & 15 & 0 & 0 & 22 & 0 & -15 \\
row1 & 0 & 11 & 3 & 0 & 0 & 0 \\
row2 & 0 & 0 & 0 & -6 & 0 & 0 \\
row3 & 0 & 0 & 0 & 0 & 0 & 0 \\
row4 & 91 & 0 & 0 & 0 & 0 & 0 \\
row5 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
$$

Matrix A

Matrix A

	Row	Col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Transpose of Matrix A

	Row	Col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

No. of rows

No. of columns

No. of values

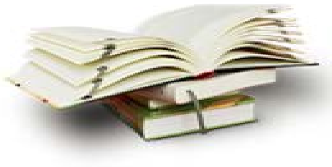
Transposing a Matrix

- To transpose a matrix, interchange the rows and columns.
  - Each element a[i][j] in the original matrix becomes element a[j][i] in the transpose matrix

```
for each row i
        take element <i, j, value> and store it
        as element <j, i, value> of the transpose
```

- Drawbacks:
  - If the original matrix is processed by the row indices it is difficult to know exactly where to place element <j, i, value> in the transpose matrix until all the preceding elements are processed
- This can be avoided by using the column indices to determine the placement of elements in the transpose matrix.

```
for all elements in column j
        place element <i, j, value> in
        element <j, i, value>
```



Assign A[i][j] to B[j][i]

place element <i, j, value>
in element <j, i, value>

For all columns i

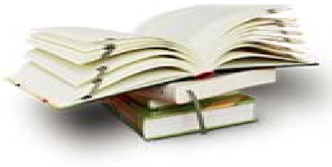
For all elements in column j

Scan the array "columns" times.
The array has "elements" elements.

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
  int n,i,j, currentb;
  n = a[0].value;          /* total number of elements */
  b[0].row = a[0].col; /* rows in b = columns in a */
  b[0].col = a[0].row; /* columns in b = rows in a */
  b[0].value = n;
  if (n > 0 )  { /* non zero matrix */
    currentb = 1;
    for (i = 0; i < a[0].col; i++)
    /* transpose by the columns in a */
      for (j = 1; j <= n; j++)
      /* find elements from the current column */
        if (a[j].col == i) {
        /* element is in current column, add it to b */
          b[currentb].row = a[j].col;
          b[currentb].col = a[j].row;
          b[currentb].value = a[j].value;
          currentb++;
        }
  }
}
```

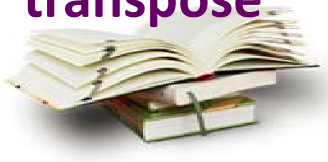
==> O(columns*elements)

```
void transpose (term a[], term b[])
{ /* b is set to the transpose of a */
	int n, i, j, currentb;
	/*total no. of elements */
	n = a[0].value;
	/* rows in b = columns in a */
	b[0].row = a[0].col;
	/* columns in b = rows in a */
	b[0].col = a[0].row;
	b[0].value = n;
if (n > 0){
	currentb = 1;
	for (i = 0; i < a[0].col; i++)
	for (j= 1; j<=n; j++)
	if (a[j].col == i)  {
		b[currentb].row = a[j].col;
		b[currentb].col = a[j].row;
		b[currentb].value = a[j].value;
		currentb++;
		}
	}
}
```



Analysis of Sparse Transpose

- The time complexity of a matrix transpose algorithm is O(columns.elements)
- compared with 2-D array representation i.e. O(columns*elements) vs. O(columns*rows)
  - elements→columns*rows when non-sparse i.e. O(columns^2*rows)
- Solution:
  - Determine the number of elements in each column of the original matrix.
  - Determine the starting positions of each row in the transpose matrix.



- **Compared with 2-D array representation:**
  - O(columns+elements) vs. O(columns*rows)
  - elements ←columns*rows
  - i.e. O(columns*rows)

Buildup row_term & starting_pos

transpose

- **Cost: Additional row_terms and starting_pos arrays are required.**

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
  int row_terms[MAX_COL], starting_pos[MAX_COL];
  int i,j, num_cols = a[0].col, num_terms = a[0].value;
  b[0].row = num_cols;  b[0].col = a[0].row;
  b[0].value = num_terms;
  if (num_terms > 0) { /* nonzero matrix */
    for (i = 0; i < num_cols; i++)            For columns
      row_terms[i] = 0;
    for (i = 1; i <= num_terms; i++)          For elements
      row_terms[a[i].col]++;
    starting_pos[0] = 1;
    for (i = 1; i < num_cols; i++)            For columns
      starting_pos[i] =
                  starting_pos[i-1] + row_terms[i-1];
    for (i = 1; i <= num_terms; i++) {
      j = starting_pos[a[i].col]++;
      b[j].row = a[i].col;   b[j].col = a[i].row;
      b[j].value = a[i].value;
    }
  }
}
                                              For elements
```

Fast Transpose of A Sparse Matrix

- **STEPS**
  - Find the no. of elements in each column of the original matrix
  - Using the no. of elements find starting positions of each row in the transpose matrix.
  - Copy row containing first 0 at location 0

5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	1	1	2	0	1	1
starting_pos =	0	1	2	4	4	5

5	6	6
0	2	4
1	1	8
2	2	2
2	4	2
4	0	9
5	3	5

Fast Transpose of A Sparse Matrix

```
void fast_transpose(term a[ ], term b[ ])
{ /* the transpose of a is placed in b */
      int row_terms[MAX_COL], starting_pos[MAX_COL];
      int i, j, num_cols = a[0].col, num_terms = a[0].value;
      b[0].row = num_cols; b[0].col = a[0].row; b[0].value = num_terms;
      if (num_terms > 0){ /*nonzero matrix*/
            for (i = 0; i < num_cols; i++)
                  row_terms[i] = 0;                  O(num_cols)
            for (i = 1; i<= num_terms; i++)
                  row_term [a[i].col]++;             O(num_terms)
            starting_pos[0] = 1;
            for (i =1; i < num_cols; i++)
                  starting_pos[i]=starting_pos[i-1] +row_terms [i-1];   O(num_cols-1)
```

```
for (i=1; i <= num_terms, i++) {
        j = starting_pos[a[i].col]++;
        b[j].row = a[i].col;
        b[j].col = a[i].row;
        b[j].value = a[i].value;
        }
    }
}
```

O(num_terms)

- **Time Complexity is O(columns + elements)**
  - **It becomes O(columns .rows) if elements --> columns * rows**