Available online at www.sciencedirect.com
ScienceDirect
Applied Mathematics and Computation 189 (2007) 314–325
APPLIED MATHEMATICS AND COMPUTATION
www.elsevier.com/locate/amc

Robs algorithm

Robins George

Department of Computer Science, Christ College, Bangalore 550 029, Karnataka, India

Abstract

Sparse matrix is a matrix having a relatively large proportion (proportion – a ratio is a comparison of two numbers. We generally separate the two numbers in the ratio with a colon (:)) of zero elements. To store the elements of the matrix in computer memory, linear array concept of storing is used. When a sparse matrix is stored in full-matrix storage mode, all its elements, including its zero elements, are stored in an array, which is a wastage of memory. In order to avoid the memory and processing overhead many alternate forms are used. Each one has separate time and space complexities and performances. In this paper we are suggesting one way of representing the sparse matrix which has both time and space complexity O($2n$) only, while all other methods work with complexity more than O($3n$) where n is the total number of non-zero elements in the matrix .The implementation of this algorithm in applications may improve the performance especially in the area of adjacency matrix, tree representation, *3D* representation to an object, network communication, electronics, mathematical calculations, picture storage/file storage, file compression, bioinformatics, and the computer performance.

The proposed algorithm has a large scope not only in computing but also in different branches of science, electronics and graphics.

Keywords: Compressed row storage; Compressed column storage; Block compressed row storage; Compressed diagonal storage; Jagged diagonal storage; Effective matrix representation; Linked list method

1. Introduction

A sparse matrix is a matrix having a relatively small number of non-zero elements. In the mathematical subfield of numerical analysis a *sparse matrix* is a matrix populated primarily with zeros.

Definition. Given a sparse $N \times M$ matrix A the *row bandwidth*[1] for the nth row is defined as

$$b_n(\mathbf{A}) := \min_{1 \leqslant m \leqslant M} \{m | a_{n,m} \neq 0\}.$$

E-mail address: robins_143@yahoo.co.in

[1] Row bandwidth: the row bandwidth of a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ is defined as $w(\boldsymbol{A}) = \max_{1 \leqslant i \leqslant m}(l_i(\boldsymbol{A}) - f_i(\boldsymbol{A}) + 1)$, where $f_i(\boldsymbol{A}) = \min\{j \mid \boldsymbol{A}_{ij} \neq 0\}$ and $l_i(\boldsymbol{A}) = \max\{j \mid \boldsymbol{A}_{ij} \neq 0\}$ are column indices of the first and last non-zero elements in the ith row of $\boldsymbol{A}$.

0096-3003/$ - see front matter
doi:10.1016/j.amc.2006.11.096

The *bandwidth* for the matrix is defined as

$$B(\mathbf{A}) := \max_{1 \leqslant n \leqslant N} b_n(\mathbf{A}).$$

Sparsity is a concept, useful in combinatorics and application areas such as network theory, of a low density of significant data or connections. This concept is amenable to quantitative reasoning. It is also noticeable in everyday life. Huge sparse matrices often appear in science or engineering when solving problems for linear models.

When storing and manipulating sparse matrices on the computer, it is often necessary to modify the standard algorithms and take advantage of the sparse structure of the matrix. Sparse data is by its nature easily compressed, which can yield enormous savings in memory usage. More importantly, manipulating huge sparse matrices with the standard algorithms may be impossible due to their sheer size. The definition of huge depends on the hardware and the computer programs available to manipulate the matrix.

Considering the following as an example of a sparse matrix A:

$$\begin{matrix}
* & & & & & * \\
|11 & 0 & 13 & 0 & 0 & 0| \\
|21 & 22 & 0 & 24 & 0 & 0| \\
|0 & 32 & 33 & 0 & 35 & 0| \\
|0 & 0 & 43 & 44 & 0 & 46| \\
|51 & 0 & 0 & 54 & 55 & 0| \\
|61 & 62 & 0 & 0 & 65 & 66|. \\
* & & & & &
\end{matrix}$$

There are 6 rows and 6 columns. Totally 36 elements and 18 elements are non-zero and remaining 18 elements are zeros. The sparse matrix requires us to consider an alternative representation which stores only non-zero elements explicitly. Each element of the matrix is characterized by its row position and column position say (i,j).

2. Matrix in storage

A sparse matrix can be stored in a full-matrix storage mode or a packed storage mode. When a sparse matrix is stored in a *full-matrix storage mode*, all its elements including its zero elements are stored in an array, which is a wastage of memory, so alternate forms are used. There is packed storage modes used for storing sparse matrix.

The efficiency of most of the iterative methods is determined primarily by the performance of the matrix–vector[2] product and therefore on the storage scheme used for the matrix. Often, the storage scheme used arises naturally from the specific application problem.

In this section we are reviewing some of the more popular sparse matrix formats that have been used in numerical software packages such as ITPACK, NSPCG and SPARSPAK[3] some of which have more recently been adopted as part of a new software standard by the BLAS[4] Technical Forum. We demonstrate how the matrix–vector product is formulated using two of the sparse matrix formats. If the coefficient matrix is sparse, large scale eigenvalue[5] problems can be most efficiently solved if the zero elements are neither manipulated nor stored. Sparse storage schemes allocate contiguous storage in memory for the non-zero elements of the matrix,

[2] Matrix–vector – multiplying a matrix and a vector is a special case of matrix multiplication. Circuit equations and linear system dynamics contain the products of a matrix and a vector.

[3] ITPACK, NSPCG and SPARSPAK used for solving large sparse linear systems by iterative methods.

[4] Basic linear algebra subprograms (BLAS) are routines that provide standard building blocks for performing basic vector and matrix operations.

[5] Eigenvalues – eigenvalues are a special set of scalars associated with a linear system of equations (i.e., a matrix equation) that are sometimes also known as characteristic roots, characteristic values.

and perhaps a limited number of zeros. This, of course, requires a scheme for knowing where the elements fit into the full matrix.

There are many methods for storing the data. Here we will discuss compressed row and column storage, block compressed row storage, diagonal storage, jagged diagonal storage, and skyline storage, effective matrix, linked list, etc. The different methods already available for the representation of the sparse matrix are as follows.

2.1. Compressed row storage

The compressed row and column storage formats are the most general: they make absolutely no assumptions about the sparsity structure of the matrix, and they do not store any unnecessary elements.

On the other hand, they are not very efficient, needing an indirect addressing step for every single scalar operation in a matrix–vector product or preconditioner solve. The compressed row storage (CRS) format puts the subsequent non-zeros of the matrix rows in contiguous memory locations. Assuming we have a non-symmetric[6] sparse matrix $\boldsymbol{A}$, we create three vectors: one for floating point numbers (`val`) and the other two for integers (`col_ind`, `row_ptr`). The `val` vector stores the values of the non-zero elements of the matrix $\boldsymbol{A}$ as they are traversed in a row-wise fashion. The `col_ind` vector stores the column indexes of the elements in the `val` vector. That is, if $\mathtt{val(k)} = \mathtt{a}_{i,j}$, then $\mathtt{col\_ind(k)} = \mathtt{j}$. The `row_ptr` vector stores the locations in the `val` vector that starts a row; that is, if $\mathtt{val(k)} = \mathtt{a}_{i,j}$, then $\mathtt{row\_ptr(i)} \leqslant \mathtt{k} < \mathtt{row\_ptr(i+1)}$. By convention, we define $\mathtt{row\_ptr(n+1)} = \mathtt{nnz} + \mathbf{1}$, where the ***nnz*** is the number of non-zeros in matrix $\boldsymbol{A}$. The storage savings for this approach is significant. Instead of storing $\boldsymbol{n^2}$ elements, we need only $\mathbf{2}\mathtt{nnz} + \mathtt{n} + \mathbf{1}$ storage locations. As an example, consider the non-symmetric matrix $\boldsymbol{A}$ defined by

$$\boldsymbol{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

The CRS format for this matrix is then specified by the arrays {`val`, `col_ind`, `row_ptr`} given below:

`val`	10	-2	3	9	3	7	8	7	3 ···9	13	4	2	-1		
`col_ind`	1	5	1	2	6	2	3	4	1 ···5	6	2	5	6		

`row_ptr`	1	3	6	9	13	17	20

If matrix $\boldsymbol{A}$ is symmetric, we only need to store the upper (or lower) triangular portion of the matrix. The tradeoff is a more complicated algorithm with a somewhat different pattern of data access.

[6] Symmetric: a *symmetric matrix* is a matrix that is its own transpose. Thus A is symmetric if $A^{\mathrm{T}} = Aa_{ij} = a_{ji}$ for all indices i and j.

2.2. Compressed column storage

Analogous to CRS, there is compressed column storage (CCS), which is also called the *Harwell–Boeing*[7] sparse matrix format. The CCS format is identical to the CRS format except that the columns of $\boldsymbol{A}$ are stored (traversed) instead of the rows. In other words, the CCS format is the CRS format for $\boldsymbol{A}^{\mathrm{T}}$.

The CCS format is specified by the **3** arrays {`val`, `row_ind`, `col_ptr`}, where `row_ind` stores the row indices of each non-zero, and `col_ptr` stores the index of the elements in `val` which start a column of $\boldsymbol{A}$. The CCS format for matrix $\boldsymbol{A}$ is given by

$$\boldsymbol{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

val	10	3	3	9	7	8	4	8	8 ···9	2	3	13	-1		
row_ind	1	2	4	2	3	5	6	3	4 ···5	6	2	5	6		

col_ptr	1	4	8	10	13	17	20

where ***nnz*** is the number of non-zeros in matrix $\boldsymbol{A}$. The storage savings for this approach is significant. Instead of storing $\mathbf{n^2}$ elements, we need only $\mathbf{2}\mathtt{nnz} + \mathtt{n} + \mathbf{1}$ storage locations.

2.3. Block compressed row storage

If the sparse matrix $\boldsymbol{A}$ is composed of square dense blocks of non-zeros in some regular pattern, we can modify the CRS (or CCS) format to exploit such block patterns. Block matrices typically arise from the discretization[8] of partial differential equations in which there are several *degrees of freedom*[9] associated with a grid point.

We then partition the matrix into small blocks with a size equal to the number of degrees of freedom and treat each block as a dense matrix, even though it may have some zeros.

If $\boldsymbol{n}_b$ is the dimension of each block and ***nnzb*** is the number of non-zero blocks in the $\boldsymbol{n} \times \boldsymbol{n}$ matrix $\boldsymbol{A}$, then the total storage needed is $\mathtt{nnz} = \mathtt{nnz} \times \mathtt{n}_b^2$. The block dimension $\boldsymbol{n}_d$ of $\boldsymbol{A}$ is then defined by $\mathtt{n}_d = \mathtt{n}/\mathtt{n}_b$.

Similar to the CRS format, we require three arrays for the BCRS format: a rectangular array for floating point numbers ($\mathtt{val}(\mathbf{1} : \mathtt{nnzb}, \mathbf{1} : \mathtt{n}_b, \mathbf{1} : \mathtt{n}_b)$) which stores the non-zero blocks in (block) row-wise fashion, an

[7] Harwell–Boeing is matrix collection is a set of standard test matrices arising from problems in linear systems, least squares, and eigenvalue calculations from a wide variety of scientific and engineering disciplines.

[8] Discretization In mathematics, discretization concerns the process of transferring continuous models and equations into discrete counterparts. This process is usually carried out as a first step toward making them suitable for numerical evaluation and implementation on digital computers. In order to be processed on a digital computer another process named quantization is essential.

[9] The degrees of freedom of a matrix are calculated based on the pivot elements of the matrix.

integer array (col_ind(1 : *nnzb*)) which stores the actual column indices in the original matrix $\boldsymbol{A}$ of the $(1,\mathbf{1})$ elements of the non-zero blocks, and a pointer array (row_blk($\mathbf{1}$: $n_d + \mathbf{1}$)) whose entries point to the beginning of each block row in val(:,:,:) and col_ind(:). The savings in storage locations and reduction in time spent doing indirect addressing for block compressed row storage (BCRS) over CRS can be significant for matrices with a large $\boldsymbol{n_b}$.

2.4. Compressed diagonal storage

If matrix $\boldsymbol{A}$ is banded with a bandwidth that is fairly constant from row to row, then it is worthwhile to take advantage of this structure in the storage scheme by storing subdiagonals[10] of the matrix in consecutive locations.

Not only can we eliminate the vector identifying the column and row, but we can pack the non-zero elements in such a way as to make the matrix–vector product more efficient. This storage scheme is particularly useful if the matrix arises from a finite element or finite difference discretization on a tensor product grid.

We say that matrix A = ($a_{i,j}$) is *banded* if there are non-negative constants $\boldsymbol{p}$, $\boldsymbol{q}$, called the left and right *halfbandwidth*[11], such that $a_{i,j} \neq 0$ only if $\boldsymbol{i} - \boldsymbol{p} \leqslant \boldsymbol{j} \leqslant \boldsymbol{i} + \boldsymbol{q}$. In this case, we can allocate for matrix $\boldsymbol{A}$ an array val(1:n,-p:q). The declaration with reversed dimensions (−p:q,n) corresponds to the LINPACK band format , which, unlike compressed diagonal storage (CDS), does not allow for an efficiently vectorizable matrix–vector multiplication if $\boldsymbol{p} + \boldsymbol{q}$. is small.

Usually, band formats involve storing some zeros. The CDS format may even contain some array elements that do not correspond to matrix elements at all. Consider the non-symmetric matrix $\boldsymbol{A}$ defined by

$$\boldsymbol{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

Using the CDS format, we store this matrix $\boldsymbol{A}$ in an array of dimension (6,-1:1) using the mapping

$$\mathtt{val(i,j)} = \mathtt{a_{i,i+j}}.$$

Hence, the rows of the val(:,:) array are

val(:,-1)	0	3	7	8	9	2				
val(:, 0)	10	9	8	7	9	-1				
val(:,+1)	-3	6	7	5	13	0				

Notice the two zeros corresponding to non-existing matrix elements.

[10] Subdiagonals – the elements just below and above the diagonal of a square matrix.
[11] Halfbandwidth – it is half of the bandwidth.

2.5. Jagged diagonal storage

The jagged diagonal storage (JDS) format can be useful for the implementation of iterative methods on parallel and vector processors. Like the CDS format, it gives a vector length of essentially the same size as the matrix. It is more space-efficient than CDS at the cost of a gather/scatter operation. A simplified form of JDS, called ITPACK storage or Purdue storage, can be described as follows. For the following non-symmetric matrix, all elements are shifted left:

$$\begin{bmatrix} 10 & -3 & 0 & 1 & 0 & 0 \\ 0 & 9 & 6 & 0 & -2 & 0 \\ 3 & 0 & 8 & 7 & 0 & 0 \\ 0 & 6 & 0 & 7 & 5 & 4 \\ 0 & 0 & 0 & 0 & 9 & 13 \\ 0 & 0 & 0 & 0 & 5 & -1 \end{bmatrix} \rightarrow \begin{bmatrix} 10 & -3 & 1 & \\ 9 & 6 & -2 & \\ 3 & 8 & 7 & \\ 6 & 7 & 5 & 4 \\ 9 & 13 & & \\ 5 & -1 & & \end{bmatrix}.$$

After which the columns are stored consecutively. All rows are padded with zeros on the right to give them equal length. Corresponding to the array of matrix elements `val(: , :)`, an array of column indices, `col_ind(: , :)`, is also stored:

`val(:, 1)`	10	9	3	6	9	5
`val(:, 2)`	−3	6	8	7	13	−1
`val(:, 4)`	1	−2	7	5	0	0
`val(:, 4)`	0	0	0	4	0	0

`col_ind(:, 1)`	1	2	1	2	5	5
`col_ind(:, 2)`	2	3	3	4	6	6
`col_ind(:, 3)`	4	5	4	5	0	0
`col_ind(:, 4)`	0	0	0	6	0	0

It is clear that the padding[12] zeros in this structure may be a disadvantage, especially if the bandwidth of the matrix varies strongly. Therefore, in the CRS format, we reorder the rows of the matrix decreasingly according to the number of non-zeros per row. The compressed and permuted diagonals are then stored in a linear array. The new data structure is called *jagged diagonals*.

Specifically, we store the (dense) vector of all the first elements in `val`, `col_ind` from each row, together with an integer vector containing the column indices of the corresponding elements. This is followed by the second jagged diagonal consisting of the elements in the second positions from the left. We continue to construct more and more of these jagged diagonals (whose length decreases).

The number of jagged diagonals is equal to the number of non-zeros in the first row, i.e., the largest number of non-zeros in any row of $\boldsymbol{A}$. The data structure to represent the $\boldsymbol{n} \times \boldsymbol{n}$ matrix $\boldsymbol{A}$ therefore consists of a permutation array `(perm(1:n))`, which reorders the rows, a floating point array `(jdiag(:))` containing the jagged diagonals in succession, an integer array `(col_ind(:))` containing the corresponding column indices, and finally a pointer array `(jd_ptr(:))` whose elements point to the beginning of each jagged diagonal.

The JDS format for the above matrix $\boldsymbol{A}$ in using the linear arrays {`perm`, `jdiag`, `col_ind`, `jd_ptr`} is given below (jagged diagonals are separated by semicolons):

[12] Padding with zeros (adding more zeros to the matrix), though a standard practice and useful in many applications, does not appear to significantly improve our data in this application.

jdiag	1	3	7	8	10	2;	9	9	8 ···-1;	9	6	7	5;	13		
col_ind	1	1	2	3	1	5;	4	2	3 ···6;	5	3	4	5;	6		

perm	5	2	3	4	1	6

jd_ptr	1	7	13	17

2.6. Skyline storage

The next storage scheme we consider is for skyline matrices, which are also called variable band or profile matrices.[13] It is mostly of importance in direct solution methods, but it can be used for handling the diagonal blocks in block matrix factorization methods. A major advantage of solving linear systems having skyline coefficient matrices is that when pivoting is not necessary, the skyline structure is preserved during Gaussian elimination.[14] If the matrix is symmetric, we only store its lower triangular part. A straightforward approach in storing the elements of a skyline matrix is to place all the rows (in order) into a floating point array `(val( : ))`, and then keep an integer array `(row_ptr( : ))` whose elements point to the beginning of each row. The column indices of the non-zeros stored in `val( : )` are easily derived and are not stored.

For a non-symmetric skyline matrix such as the one illustrated in figure, we store the lower triangular elements in skyline storage (SKS) format and store the upper triangular elements in a column-oriented SKS format (transpose stored in row-wise SKS format). These two separated *substructures* can be linked in a variety of ways. One approach, is to store each row of the lower triangular part and each column of the upper triangular part contiguously into the floating point array `(val( : ))`. An additional pointer is then needed to determine where the diagonal elements, which separate the lower triangular elements from the upper triangular elements, are located.

3. Effective matrix representation

It is a simple way of representing the sparse matrix. In effective matrix representation, the non-zero elements of matrix **A** are stored in a special matrix called effective matrix, where **A**[**0**][**0**] is the total number of rows, **A**[**0**][**1**] is the total number of columns, **A**[**0**][**2**] is the total number of non-zero elements in the matrix. For the each entry in the effective matrix,

A[**i**][**0**] is the row value of the non-zero element, where $i = 1, 1, 2 \ldots n$
A[**i**][**1**] is the column value of the non-zero element $i = 1, 1, 2 \ldots n$
A[**i**][**2**] is the non-zero entry $i = 1, 1, 2 \ldots n$

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

[13] Variable band or profile matrices – related to the diagonal elements.
[14] Gaussian elimination – one method of solving the linear system of equations using substitution.

The corresponding effective matrix is as follows:

```
6   6   17
1   1   10
1   2   -3
2   1    3
------------
------------
6   6   -1
```

Here the space required for the storage of the non-zero elements is $3nz + 3$ where nz is the total number of non-zero elements in the given matrix i.e., the complexity is O($3n$), for each entry we store three parameters about the entry. Using these three entries the original matrix can be regenerated from the effective representation.

4. Linked list method

In the linked list approach, each node having row, column and non-zero value is used to represent a single non-zero entry in the sparse matrix. The structure of the node is as follows

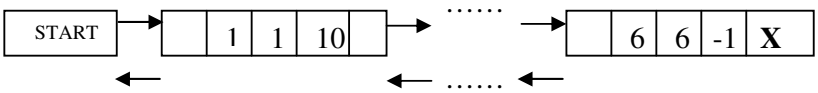
PTR	Row	Col	Val	PTR

The linked list is formed with nodes which contain the information including row value column value, non-zero value, and pointer to other nodes. Each node is formed when a non-zero entry is found in the matrix and added to the list with relevant information.

Consider the example,

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

This can be represented as follows:



In this method the space complexity is O($5n$) as each node requires five unit of information.

5. Proposed structure (Robs method)

We have seen that each element in the sparse matrix is represented using three or more parameters including column and row information and each one having the space complexity greater than or equal to O($3n$) and some of the method have space complexity O(5) too. Sometimes a sparse matrix representation largely depends the distribution of the non-zero elements in the original matrix.

In our approach only two parameters are used to identify an entry in the sparse matrix and using this information we can regenerate the original matrix, where this method takes two operations only so the complexity of this approach is O($2n$), which saves the memory and computing time for representing and regenerating the matrix.

6. Method of storing the matrix

Consider the sparse matrix with order m and n containing nz non-zero elements. The above matrix can be represented in the form of a two column matrix. The first entries in this approach are as follows:

$A[0][0]$ contains the value $m + n$,

$A[0][1]$ contains the value $m - n$.

The first column contains the information which is used to represent the index of the non-zero value and the second column contains the actual non-zero value. Each element in the first column is determined by the formula $i^*n + j$, where i is the row value, j is the column value and n is the total number of columns. Consider the example

$$A = \begin{bmatrix} 10 & -3 & 0 & 0 & 0 & 0 \\ 3 & 9 & 6 & 0 & 0 & 0 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 0 & 0 & 8 & 7 & 5 & 0 \\ 0 & 0 & 0 & 9 & 9 & 13 \\ 0 & 0 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

This can be represented as

```
12   0
1   10
2   -3
7    9
-------
-------
36  -1
```

In the above representation the sparse matrix is regenerated by the sequence of three steps as follows, find the value of m and n (the dimension of the original matrix can be obtained from the following steps)

```
Since m+n is 12 and
   m−n is 0 in the example
   2m is 12.
So m=n=6.
```

The position of the non-zero is calculated using the first column information in the sparse matrix representation.

```
r_cval = A[i][0] where i = 1,1,2...n
rVal = r_c Val/n                  //row value
cVal = r_c Val MOD n              // column value
Val = A[i][1] where i = 1,1,2...n        // non-zero entry
```

So we put the information Val in B[r_Val][c_Val]=Val, where B is the regenerated original matrix. Since it takes two computation and two units of storage only, the complexity of the above approach is $O(2n)$ where n is the total number of non-zero elements in the given matrix. The proposed algorithm has a large scope not only in computing but also in different branches of science, electronics and graphics are some of them.

7. Implementation

The above algorithm can be implemented in any one of the programming languages, here the implementation in c.

```
.....
.....

for(i=0;i<nz;i++)
{
   printf("Enter the Row value");
   scanf("%d",&r);
   printf("Enter the Column value");
   scanf("%d",&c);
   printf("Enter the value of element");
   scanf("%d",&val);
   tr[i+1][1]=val;       // non zero value
   tr[i+1][0]=((r)*tc)+c;    // tc is total column in the matrix

   if(max_row< r)
          max_row=r;
   if(max_col< c)
          max_col=c;
   }
printf("\n\t Effective matrix (NEW)...\n");
printf(".....\n \ n");
for(i=0;i< nz + 1;i++)
   {
   for(j=0;j<2;j++)
        printf("%4d",tr[i][j]);
        printf("\n");
   }
printf("\n \ n\t.....Sparse Matrix (NEW).....");
printf("\n.....\n \ n\t");
size= tr[0][1];
for(i=1;i < nz+1;i++)
   {
   r_c=tr[i][0];
        r=r_c/size;
             c=r_c%size;
          val=tr[i][1];
   sp[r][c]=val;
   }

.....
.....
```

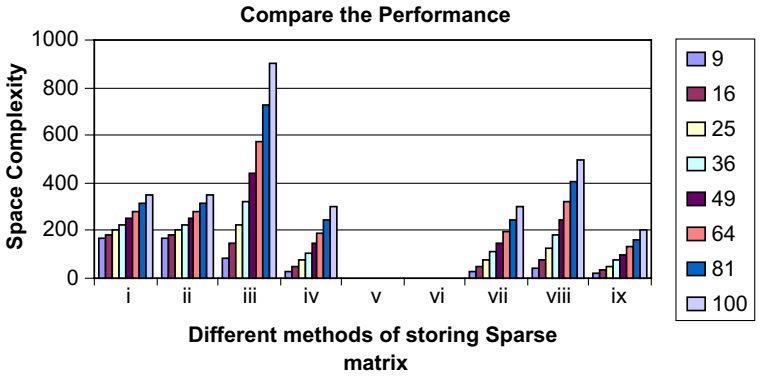
8. Performance

If the number of non-zero elements in the matrix is n, then the space complexity of this algorithm is proportional to $O(2n)$. The two operations can be used to determine the row value and column value of the each

non-zero element in the sparse matrix, and the effective matrix can be easily transformed into the original matrix form with the above two operations, so for each element present in the matrix only two operations are needed. So the space complexity and time complexity improved, which is directly proportional to the number of non-zero elements in the sparse matrix, which is equal to O(2*n*). We have seen that rest of the methods take more than two operations to transform the sparse matrix representation into the original form. This algorithm uses less storage space and computation time and which is O(2*n*), where *n* is the total number of non-zero elements in the matrix.

9. Discussion

Assume the order of the matrix 150×150, and block size is 4



SI	Non-zero	I	ii	iii	iv	v	vi	vii	viii	ix	Best
1	9	169	169	81	27	Matrix		29	45	20	ix
2	16	183	183	144	48	dependent[a]		50	80	34	ix
3	25	201	201	225	75			77	125	52	ix
4	36	223	223	324	108			110	180	74	ix
5	49	249	249	441	147			149	245	100	ix
6	64	279	279	576	192			194	320	130	ix
7	81	313	313	729	243			245	405	164	ix
8	100	351	351	900	300			302	500	202	ix

(i) Compressed row storage; (ii) compressed column storage; (iii) block compressed row storage; (iv) compressed diagonal storage; (v) jagged diagonal storage; (vi) skyline storage; (vii) effective matrix; (viii) linked list; (ix) new approach (Rob's method).

[a] The sparse matrix formed depends on the distribution of the non-zero elements within the matrix not directly to the number of non-zero elements present.

From the above table we could see that the new method's complexities are better than other methods for the given data. The complexity of the new algorithm is O(2*n*), where the *n* is the total number of non-zero elements in the matrix.

10. Conclusion

The above table shows the comparison of most common methods which are used to represent the sparse matrix with the proposed structure. We could not include the jagged diagonal storage and skyline storage as the performance depends on the distribution of the non-zero elements in the matrix and not the number of non-zero elements in the given matrix.

The results reveal that the proposed structure performs well for all kinds of inputs. When the number of non-zero elements in the matrix is small then there is not much variation in the performance, but when the

number of non-zero elements are large, the difference in the performance is noticeable. In real we have to handle the matrix with order is more than 10.

Using this algorithm the following areas can be benefited:

- Adjacency matrix – a directed graph can be represented using the sparse matrix.
- Tree representation – the array representation of the trees are done with the help of the sparse matrix.
- 3D representation to an object – objects with attributes are represented in a 3D geometry.
- Used in network communication – information can be sent in the form of a sparse matrix.
- Electronics – circuits, input and outputs from the circuits can be represented using the sparse matrix.
- Mathematical calculations such as to finding the inverse of a matrix can be made by using the Gaussian elimination method.
- Picture storage/file storage – these operations reduces the time by 2.
- File compression – a better file compression algorithm can be formed.
- Bioinformatics.
- Combinatorics.
- Network theory.
- Chemistry (Blakley's method of chemical equation Solving chemical equations balancing equations using matrices).

Acknowledgement

The thoughts and viewpoints presented in this paper have seen the eminently priceless contribution from a significant number of individuals in the form of constructive criticism, well wishes and notably, words of wisdom and inspiration. We are grateful to THE ALMIGHTY GOD for His blessings in completing this humble effort. We thank Him for His love and kindness showered on us, devoid of which we would never have been able to accomplish this project.

We also extend our sincere thanks to all the members of the master of computer application faculty and department of computer science for their suggestions, guidance and timely encouragement.

Further reading

Books

1. Data Structures Using C by Aaron M. Tenenbaum.
2. Direct Methods for Sparse Linear Systems Timothy A. Davis.
3. Sparse Matrix Computations by James R. Bunch.

Websites

1. <http://en.wikipedia.org/wiki/Sparse_matrix>.
2. <http://www.nist.gov/dads/HTML/sparsematrix.html>.
3. <http://www.cise.ufl.edu/research/sparse/matrices/>.
4. <http://www.mathworks.com/access/helpdesk/help/techdoc/math/sparse.html>.
5. <http://math.nist.gov/MatrixMarket/index.html>.

Journals

1. ACM.
2. IEEE.
3. IETK.
4. Bytes.