Review on Storage and Ordering for Large Sparse Matrix

(Final Project, Math221)

Wei He

(SID: 15763774)

Introduction

Many problems in science and engineering require the solution of a set of sparse matrix equations of the form

$$[A]\{x\} = \{b\} \tag{1}$$

where $[A]$ is a known $N \times N$ matrix which is large and sparse, $\{x\}$ is a vector of unknowns of length *N*, and $\{b\}$ is a known vector of length *N*. Especially for those equations in the fields of structural engineering and mechanical engineering, some special properties are often found to the matrix $[A]$. For example, they are generally symmetric and well diagonal blocked. It is well known that the storage of the coefficient matrix is crucial in the procedure of solving this system of equations. A reasonable scheme of storage not only saves memory and computing time, but also makes some manipulations, e.g. the assembly of the matrix, easier. In addition, the way the unknowns are ordered in the global vectors is equally important since matrix-vector operations constitute the bulk of the work. The order of unknowns also affects the shape of coefficient matrix and furthermore its storage structure. As a result, storage and ordering for the coefficient matrix are two important topics for many years and a large number of schemes have been developed. In this paper, several typical storage structures and ordering algorithms related to large sparse matrix are reviewed.

Part I: Storage for large sparse matrix

Each data structure for a sparse matrix consists of storage for values of matrix elements and storage for pointer, bounds, indices, etc. The total amount of storage needed should be as low as possible without preventing an efficient use of the data structure. Data structures for sparse matrices are designed to decrease the amount of storage for values of matrix elements considerably at the expense of a small increase in administration. Just as Laura C. Dutto, *et al.* said [1], ordinarily it is chosen to limit the number of zeros in order to minimize the memory required and to avoid unnecessary operations with zero values during subsequent numerical calculations. On the other hand, the data

structure must also be chosen so that the software can take advantage of hardware features such as vector registers or parallel processing capabilities. Finally, the software developer needs to be on the lookout for any regularities or patterns present in the problem to solve.

For those sparse matrices especially related to structural engineering and mechanical engineering, they generally have the following properties:

1. Often matrices are symmetric or have a symmetric sparsity pattern.
2. Several blocks of a matrix may be equal.

In this paper, several typical sparse matrix storage structures are briefly introduced. First, for those symmetric matrices, three storage schemes are introduced and they are 2-D band storage structure with equal bandwidth, 1-D band storage structure with varying bandwidth and element by element storage, respectively. These data structures were developed very early and are widely applied in engineering. Then, three storage structures for arbitrary sparsity patterns are presented and they are 3-tuple storage, CSR storage and BSR storage, respectively.

I-1 Data structures for symmetric matrix

2-D band structure [2, 3]

Consider a sparse matrix $K_{n\times n}\left(K = K^T\right)$ and assume its maximum half bandwidth is D, then all non-zero entries in the upper triangular matrix fall into this band. A doubly subscripted array is introduced to store the data in the band. Apparently, the bound of this array is $n\times D$. The diagonal data in the original matrix constitute the first column of the array, and the new row indices and column indices of data are related to original ones by:

$$\begin{aligned} i^* &= i \\ j^* &= j - i + 1 \end{aligned} \qquad 2$$

where the superscript $*$ refer to new indices.

It can be seen that the zero entries outside the maximum bandwidth are all removed and as a result, memory is greatly saved. However, those zero entries inside the band are still stored. Therefore, this strategy is applicable for the case when the coefficient matrix is well banded.

1-D band structure [3]

For this structure, the data inside the varying bandwidth are stored in a 1-D array in the prescribed sequence. Unlike the 2-D band structure before, those zero elements outside the varying bandwidth (not maximum bandwidth) will not enter into the array and the storage is further reduced accordingly. In order to keep the shape of original sparse matrix, another array is needed to store some additional information such as the address of diagonal elements or the number of elements in each column.

Apparently, 1-D band structure needs less memory than 2-D structure. However, it is more difficult to seek and locate elements. More complicated program has to be carefully worked out and running code is time-consuming.

A simple example is presented below. The original matrix is stored in 2-D array and 1-D array, respectively. It can be seen that 1-D storage needs less memory, but requires more complicated administrations.

$$\begin{bmatrix} K_{11} & K_{12} & 0 & K_{14} & 0 & 0 & 0 & 0 \\ & K_{22} & K_{23} & 0 & 0 & 0 & 0 & 0 \\ & & K_{33} & K_{34} & 0 & K_{36} & 0 & 0 \\ & & & K_{44} & K_{45} & K_{46} & 0 & 0 \\ & & & & K_{55} & K_{56} & 0 & K_{58} \\ & symmetric & & & & K_{66} & K_{67} & 0 \\ & & & & & & K_{77} & K_{78} \\ & & & & & & & K_{88} \end{bmatrix} \Rightarrow \begin{bmatrix} K_{11} & K_{12} & 0 & K_{14} \\ K_{22} & K_{23} & 0 & 0 \\ K_{33} & K_{34} & 0 & K_{36} \\ K_{44} & K_{45} & K_{46} & 0 \\ K_{55} & K_{56} & 0 & K_{58} \\ K_{66} & K_{67} & 0 & \\ K_{77} & K_{78} & & \\ K_{88} & & & \end{bmatrix}$$

$$(original\ matrix) \qquad\qquad (2-D\ storage)$$

$$\Downarrow$$

$$\begin{bmatrix} A(1) & A(3) & 0 & A(9) & 0 & 0 & 0 & 0 \\ & A(2) & A(5) & A(8) & 0 & 0 & 0 & 0 \\ & & A(4) & A(7) & 0 & A(15) & 0 & 0 \\ & & & A(6) & A(11) & A(14) & 0 & 0 \\ & & & & A(10) & A(13) & 0 & A(21) \\ & symmetric & & & & A(12) & A(17) & A(20) \\ & & & & & & A(16) & A(19) \\ & & & & & & & A(18) \end{bmatrix} \Rightarrow A(21) \quad (1-D\ storage)$$

Supplementary array recording the address of diagonal element:

$$M:[1 \quad 2 \quad 4 \quad 6 \quad 10 \quad 12 \quad 16 \quad 18 \quad 22]$$

Here the last element denotes the length of 1-D array plus one.

Element by element storage [4]

When solving partial differential equations with finite element method in structural engineering or mechanical engineering, large sparse stiffness matrices are often stored element by element. It is a good storage scheme for programmer when the field is homogeneous and the elements are equally sized. For this case, the global stiffness matrix is diagonal blocked and well banded. For example, for a 1-D problem, if the field is homogeneous and discretized uniformly using 3 elements, the stiffness matrix is generally as follows:

$$\begin{bmatrix} K_{11} & K_{12} & & \\ K_{21} & K_{22} & K_{23} & \\ & K_{32} & K_{33} & K_{34} \\ & & K_{43} & K_{44} \end{bmatrix}_{4\times4} \leftarrow \begin{bmatrix} K_{1,11} & K_{1,12} & & \\ K_{1,21} & K_{1,22}+K_{2,11} & K_{2,12} & \\ & K_{2,21} & K_{2,22}+K_{3,11} & K_{3,12} \\ & & K_{3,21} & K_{3,22} \end{bmatrix}$$

The matrix can be stored element by element with a 3-D array:

$$K(1,:,:) = \begin{bmatrix} K_{1,11} & K_{1,12} \\ K_{1,21} & K_{1,22} \end{bmatrix}$$

$$K(2,:,:) = \begin{bmatrix} K_{2,11} & K_{2,12} \\ K_{2,21} & K_{2,22} \end{bmatrix}$$

$$K(3,:,:) = \begin{bmatrix} K_{3,11} & K_{3,12} \\ K_{3,21} & K_{3,22} \end{bmatrix}$$

It can be seen that the first subscript basically indicates the address of block in diagonal and it actually corresponds to the element number. With this storage scheme, it would be very easy to assemble the global stiffness matrix. Also, the stiffness matrix for each element is probably the same, so the work is greatly saved when writing code.

I-2 Data structures for arbitrary sparsity patterns

3-tuple format [5, 6]

A general data structure for this case was developed in 1970s, and was briefly introduced by Veldhorst in [6]. In this structure, a matrix is viewed as a set of elements, each one uniquely determined by two integers: the row index and the column index. Each non-zero element is stored as a 3-tuple (its value, row index and column index) and two pointers, one referring to the next non-zero element in the same row and the other to the next non-zero element in the same column. It should be noted that this data structure can be extended and adapted to make some important operations on sparse matrices easier. For example, if diagonal access is needed, the data structure can be easily extended by adding to each element a pointer to the next non-zero in the same diagonal. The main disadvantage of this scheme is that the size of administration overhead is very large. For example, it would require more time if we want to insert a newly created non-zero elements in this restricted data structure.

Compressed Sparse Row (CSR) format [1, 7, 8]

If the matrix is sparse and not regularly structured, another two of the most common storage schemes in use today are the Compressed Sparse Row (CSR) scheme and the Block Sparse Row (BSR) format. Here the formats are presented, following the description in [1, 7, 8]. The corresponding data structure in the CSR format consists of three arrays:

1. A real array **A** containing the real values a_{ij}, stored row by row. The length of **A** is

NNZ, which is the number of non-zero coefficients of the matrix. In a finite element context, the coefficient a_{ij} is (logically) nonzero if and only if the equation i is connected with equation j. In this case, the coefficient is considered nonzero even if at a given step of computation it is indeed (by chance) zero.

2. An integer array **JA** containing the column indices of the elements a_{ij} as stored in the array **A**. The length of **JA** is also NNZ.

3. An integer array **IA** containing the pointers to the beginning of each row in the arrays **A** and **JA**. Thus the content of **IA**(i) is the position in arrays **A** and **JA** where the i th row starts. The length of **IA** is N+1 with **IA**(N+1) containing the number **IA**(1)+NNZ, i.e., the address in **A** and **JA** of the beginning of a fictitious row N+1.

In addition, when incomplete LU factorizations of the system matrix are used for preconditioning, it is crucial to be able to sweep across the rows in the lower and the upper triangles of the matrix. While the lower part of the matrix is easily localized, a search is required to locate the leading diagonal coefficient of the row in the upper triangular part in order to sweep the coefficients in the reminder of the row. To avoid repeated searches, a supplementary integer array **IDIAG,** of length N, points to the position of each diagonal coefficient inside **JA** and **A**.

Still take the 4×4 matrix above as example. Though it is symmetric, the idea is applicable for non-symmetric matrices. For this simple matrix, the according arrays are:

$$A=[K_{11}\quad K_{12}\quad K_{21}\quad K_{22}\quad K_{23}\quad K_{32}\quad K_{33}\quad K_{34}\quad K_{43}\quad K_{44}]$$

$$JA=[1\quad 2\quad 1\quad 2\quad 3\quad 2\quad 3\quad 4\quad 3\quad 4]$$

$$IA=[1\quad 3\quad 6\quad 9\quad 11]$$

$$IDIAG=[1\quad 4\quad 7\quad 10]$$

where $NNZ=10$

The Block Sparse Row (BSR) format [1, 7, 8]

The best way to describe block matrices is by viewing them as sparse matrices whose nonzero entries are L-by-L dense blocks. Typically, for block matrices arising from the discretization of partial differential equations, L is a small number, less than ten, equal to the number of degrees of freedom per grid point, e.g., velocity, pressure, viscosity, etc. The BSR format is a simple generalization of the CSR format, using the same data structures **A**, **IA**, **JA**, and **IDIAG** but where in this case the column pointers **JA** point to L-by-L dense blocks. If there are zero elements within each block they are treated as nonzero elements with the value zero.

The block dimension of the matrix A is $NR\approx N/L$, where the letter R stands for “reduced”. The length of **JA** is NNZR, set to the number of nonzero blocks in A and roughly equal to NNZ/L^2. **JA** holds the actual column positions in the original matrix

of the first element of the nonzero blocks. Finally, the pointer array **IA** of NR+1 coefficients, points to the beginning of each block-row in **A** and **JA**. As in the previous case, an additional vector **IDIAG** of length NR is used to point to the position of diagonal blocks in **A** and **JA**.

Substantial memory savings over CSR result from the reduction of the matrix pointers used in indirect addressing. For example, for L=4, the storage of JA is reduced by a factor of 16 and the storage of IA reduced by a factor of 4. The storage of A is roughly the same. Savings in execution time are also observed since shorter lists are being scanned when searching for columns in a row. Since a binary search algorithm is used, a reduction in size of JA by 16 for L=4 can give at best an 8-fold $(L\log_2 L)$ speed-up for the searches since the number of column pointers in a row is L times less, times the L rows in the block.

Part II: Ordering schemes

It is well known that if we avoid operating on and storing zeros, the way we number or order the unknowns of a sparse system of equations can drastically affect the amount of computation and storage required for their solution [28]. Accordingly, many bandwidth and profile reduction algorithms have been proposed [9-20]. The second part of this paper introduces and compares several typical ordering algorithms. Classical ordering strategies include bandwidth- and profile-reducing orderings, such as reverse Cuthill-McKee [12, 21, 22, 24], Sloan's ordering [22], and Gibbs-Poole-Stockmeyer ordering [23]; Variants of the minimum degree ordering [25, 26, 27]; and nested dissection [28, 29]. In this paper, reverse Cuthill-McKee, Gibbs-Poole-Stockmeyer ordering, and Sloan's ordering will be introduced.

Some definitions about graph:

As discussed by Cuthill and McKee [12, 22], the derivation of an efficient ordering for a sparse matrix is related to the labeling of an undirected graph. Some elementary concepts from graph theory are useful in the development of heuristic labeling strategies and it is appropriate to state some basic definitions. Here the description in [22, 23] is followed.

A graph G is defined to be pair (N(G), E(G)) where N(G) is non-empty finite set of members call nodes, and E(G) is a finite set of unordered pairs, comprised of distinct members of N(G), called edges. A graph satisfying the above definition is said to be undirected because E(G) is comprised of unordered pairs. The occurrence of loops (i.e. edges which join nodes to themselves) and multiple edges (i.e. pairs of nodes which are connected by more than one edge) is excluded.

The degree of a node i in G is defined as the number of edges incident to i. Two nodes i and j in G are said to be adjacent if there is an edge joining them.

A path in G is defined by a sequence of edges such that consecutive edges share a common node. Two nodes are said to be connected if there is a path joining them. A graph G is connected if each pair of distinct nodes is connected.
The distance between nodes i and j in G is denoted d(i,j), and is defined as the number of edges on the shortest path connecting them. The diameter of G is defined as the maximum distance between any pair of nodes, i.e

$$D(G) = \max\{d(i,j) : i, j \in N(G)\} \tag{3}$$

Nodes which are at opposite ends of the diameter of G are known as peripheral nodes.

A pseudo-diameter, $\delta(G)$, is defined by any pair of nodes i and j for which d(i,j) is close to D(G). A pseudo-diameter may be slightly less than, or equal to, the true diameter and is found by some approximate algorithm. Nodes which define a pseudo-diameter are known as pseudo-peripheral nodes.
An important concept in the development of graph labeling algorithms is the rooted level structure. A rooted level structure is defined as the partitioning of N(G) into levels $l_1(r), l_2(r), \ldots, l_h(r)$ such that :

1. $l_1(r) = \{r\}$ where r is the root node of the level structure.

2. For $i > 1, l_i(r)$ is the set of all nodes, not yet assigned a level,

which are adjacent to nodes in $l_{i-1}(r)$.

The level structure rooted at node r may be expressed as the set $L(r) = \{l_1(r), l_2(r), \ldots, l_h(r)\}$, where h is the total number of levels and is known as the depth. The width of level i is defined by $|l_i(r)|$ (i.e. the number of nodes on level i) and the width of level structure is given as

$$w = \max_{1 \le i \le h}\{|l_i(r)|\} \tag{4}$$

For example, consider the grid of two-dimensional finite elements shown below,

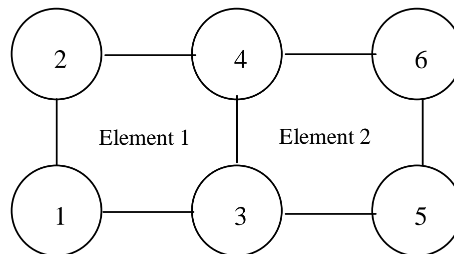

Fig. 1 Grid of four-noded quadrilaterals

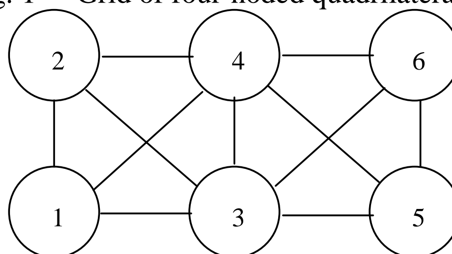

Fig. 2 Graph corresponding to grid of four-noded quadrilaterals

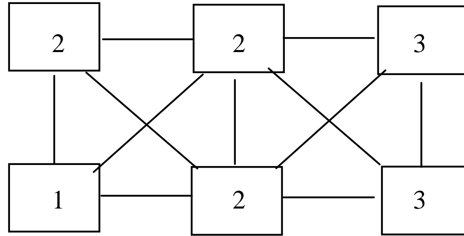

Fig. 3 A rooted level structure

Here, N(G) is the set $\{1, 2, 3, 4, 5, 6\}$, E(G) is the set $\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{3, 6\}, \{4, 5\}, \{4, 6\}, \{5, 6\}$. D(G) is 2. The rooted level structure (rooted at node one) may be expressed as

$$L(1) = \{l_1(1), l_2(1), l_3(1)\} \quad (5)$$

where $l_1(1) = \{1\}, l_2(1) = \{2, 3, 4\}, l_3(1) = \{5, 6\}$. The width and depth for this level structure both equal three.

The reverse Cuthill-McKee algorithm

One of the most widely used bandwidth and profile reduction algorithms is reverse Cuthill-Mckee algorithm. For the reverse Cuthill-McKee algorithm, it is assumed that the graph is connected. If not, the connected components are determined and the algorithms applied to each component separately. The general procedure is as follows [23, 21, 12]:

1.Generate the level structure rooted at each vertex of low degree, and compute its width. Normally, low degree here means less than or equal to $\max\{\min\{(d_{\max} + d_{\min})/2, d_{median} - 1\}, d_{\min}\}$, although this can be controlled somewhat by parameters.
2. For each rooted level structure of minimal width generated in step 1, number the

graph level by level with consecutive positive integers according to the following procedure:

   A. The rooted vertex is assigned the number 1. (If this is not the first component of the original graph the root vertex is assigned the smallest unassigned positive integer.)
   B. For each successive level, beginning with level 2, first number the vertices adjacent to the lowest numbered vertex of the preceding level, in order of increasing degree. Ties are broken arbitrarily. The remaining vertices adjacent to the next lowest numbered vertex of the preceding level are numbered next, again in order of increasing degree. Continue the process until all vertices of the current level are numbered, then begin again on the next level. The procedure terminates when the vertices of all levels have been numbered.

3. For each numbering f produced in step 2.B, compute the corresponding bandwidth $\beta_f(G)$. Select the numbering which produces the smallest bandwidth.
4. The numbering is reversed by setting i to $n-i+1$, for $i=1,2,\ldots,n$.

Step 4 was first suggested by George [21] after he observed that profile could frequently be further reduced by numbering the vertices in decreasing order from n to 1 rather than increasing from 1 to n. It was proved that this modification can never increase the profile, and of course it has no effect on bandwidth.
This algorithm has several shortcomings [23]. The first is that the algorithm is inefficient because of the time consumed performing an exhaustive search to find rooted level structures of minimal width. In the case that all vertices have the same degree, a level structure must be generated from every vertex of the graph. A second problem is that the graph is renumbered, and the corresponding bandwidth recomputed, for every level structure found of minimal width. A third problem is that the bandwidth obtained by a Cuthill-McKee numbering can never be less than the width of rooted level structure used, although the (minimum) bandwidth of a graph can be considerably less than the width of any rooted level structure.

Gibbs-Poole-Stockmeyer ordering

To resolve the three problems, Gibbs, *et al* presented an alternative algorithm. The first two shortcomings are overcome by carefully selecting a starting vertex after generating only a relatively small number of level structures. The graph is renumbered and corresponding bandwidth and profile computed, only once. The third problem is resolved by utilizing a more general type of level structure. Here only main features of this algorithm are reiterated and detailed processes can be found in their paper [23].

1. Finding a starting vertex (Finding endpoints of a pseudo-diameter).

In their work, they found that level structures of small width are usually among those of maximal depth. Clearly, increasing the number of levels always decreases the average number of vertices in each level, and tends to reduce the width of the level

structures as well. Ideally, one would like to generate level structures rooted at endpoints of a diameter. Since there is no known efficient procedure that always finds such vertices, they employ an algorithm to find the endpoints of a pseudo-diameter, that is, a pair of vertices that are at nearly maximal distance apart.
Procedure:

A. Pick an arbitrary vertex of minimal degree and call it v.

B. Generate a level structure L_v rooted at vertex v. Let S be the set of vertices which are in the last level of L_v.

C. Generate level structures rooted at vertices $s \in S$ selected in order of increasing degree. If for some $s \in S$ the depth of L_s is greater than the depth of L_v, then set $v \leftarrow s$ and return to step B.

D. Let u be the vertex of S whose associated level structure has smallest width, with ties broken arbitrary. The algorithm terminates with u and v the endpoints of a pseudo-diameter.

2. Minimizing level width.
In the process of finding a pseudo-diameter, level structures L_u and L_v rooted at the endpoints u and v are constructed respectively. It is possible to combine these two level structures into a new level structure whose width is usually less than that of either of the original ones, using the algorithm described by Gibbs, *et al*.

3. Numbering.
The numbering procedure is similar to that of the reverse Cuthill-McKee algorithm in that it assigns consecutive positive integers to the vertices of G level by level. A few modifications were necessary, however, since the level structures obtained by algorithm 2 are of a more general type than the rooted ones used in the reverse Cuthill-McKee algorithm. When the resulting numbering is similar to that obtained by the (forward) Cuthill-McKee algorithm, profile can be further reduced by using the reverse numbering.

Sloan's ordering

In Sloan's paper [22], an algorithm for reducing the profile and wavefront of a sparse matrix was described. The procedure is applicable to any sparse matrix with a symmetric pattern of zeros and may be used to generate efficient labeling for finite element grids. In particular, it may be used to provide efficient nodal numberings for profile solution schemes, as well as efficient element numberings for frontal solution schemes. Application of the algorithm to some test problems indicates that it is more effective than the reverse Cuthill-McKee, Gibbs *et al* schemes. Detailed timing comparisons indicate that the new algorithm is substantially faster, and requires less storage too. In addition, one of major attractions of the proposed scheme is its simplicity.
Once the graph that corresponds to the sparse matrix is established, the labeling scheme is comprised of two distinct steps (following the description in [22]).

1. Selection of pseudo-peripheral nodes

It has been shown by Gibbs, *et al*. that pseudo-peripheral nodes make good starting points for profile and wavefront reduction algorithms. Here is a method for locating a pair of pseudo-peripheral nodes, which are endpoints of a pseudo-diameter:

A. (First guess for starting node) Scan all nodes in G and select a node s with the smallest degree.
B. (Generate rooted level structure) Generate the level structure rooted at node s, i.e. $L(s) = \{l_1(s), l_2(s), \ldots l_h(s)\}$.
C. (Sort the last level) Sort the nodes in $l_h(s)$ in ascending sequence of degree. These nodes are at maximum distance from s.
D. (Shrink the last level) Let m equal $|l_h(s)|$. Shrink the last level by forming a list Q of the first $\lfloor (m+2)/2 \rfloor$ (the largest integer less than or equal to $(m+2)/2$) entries in the sorted list $l_h(s)$.
E. (Initialize) Set $w_{\min} \leftarrow \infty$ and $h_{\max} \leftarrow h$.
F. (Test for termination) For each node $i \in Q$, in order of ascending degree, generate $L(i) = \{l_1(i), l_2(i), \ldots l_h(i)\}$. If $h > h_{\max}$ and $w = \max\{|l_j(i)|\} < w_{\min}$, set $s \leftarrow i$ and go to step 3. Else, if $w < w_{\min}$, set $e \leftarrow i$ and $w_{\min} \leftarrow w$.
G. (Exit) Exit with starting node s and end node e which define a pseudo-diameter.

The above algorithm is similar to the procedure given by Gibbs *et al.*, but includes two important modifications [22]. The first modification is the introduction of the shrinking strategy in step D. This step significantly reduces the amount of computation necessary to locate the pseudo-peripheral nodes, but at the same time ensures that their rooted level structures are deep and narrow. It follows naturally from the empirical observation that nodes with high degrees are not often selected as potential starting or end nodes in step F.

The second modification occurs in step F and incorporates the 'short circuiting' strategy suggested by George and Liu. Inserting the condition that w must be less than $w_{\min}$ permits the assembly of wide level structures to be aborted before completion and often leads to considerable savings (especially for large graphs).

The above algorithm usually locates the pseudo-peripheral nodes in two or three iterations, and is considered to be efficient. The pseudo-diameter produced is often a true diameter, but there is no guarantee of this.

2. Node labeling algorithm

To begin the labeling procedure two pseudo-peripheral nodes, which define a pseudo-diameter, are required. These serve as starting and end nodes for the labeling. The algorithm relabels the starting node as node one and then forms a list of nodes that are eligible to receive the next label. This list is comprised of all active and preactive nodes, and is maintained as a priority queue. The node with the highest priority is labeled next. The priority of each node in the queue is related to its current degree and its distance from the end node. Nodes with low current degrees and large distances from the end node assume the highest priority. Once a node is selected for labeling, it is deleted from the queue and renumbered. The queue of eligible nodes is then updated by using the connectivity information for the graph and the process is repeated until all the nodes have been assigned new labels.

The detailed algorithm can be found in Sloan's paper. The basic idea behind the algorithm is that, during each stage of the labeling process, nodes with small current degrees and long distances from the end node are labeled first. Selecting nodes with small current degrees causes the current 'front' of active nodes to grow by a minimum amount during each step, while selecting nodes with large distances from the end node attempts to take the global structure of the graph into account.

This algorithm has been implemented in standard FORTRAN 77 and the performance is tested. It can be concluded that the procedure is a substantial improvement on previous algorithms since it is fast, reliable, requires little storage, and is simple to implement.

Conclusions

This paper only deals with several classical schemes on the storage and ordering for large sparse matrix. There are still many other papers on these two topics (e.g. [30-37]). The researchers in different scientific or engineering fields would have matrices with different properties. It is the key point to choose a proper algorithm for each linear system according to its property and some special requirements, if any. For example, for the storage of sparse matrix, some schemes are good at saving running time, while some others are easy to administrate. One should balance different requirements and choose the best data structure.

References

[1] Laura C. Dutto, *et al*, Effect of the storage format of sparse linear systems on parallel CFD computations, *Comput. Methods Appl. Mech. Engrg.* 188, 441, 2000

[2] A. Jennings, A compact storage scheme for the solution of simultaneous equations, *Computer J.*, 9, 281-285,1966

[3] X. C. Wang, *et al.*, Basic theory and numerical methods of FEM (Chinese version), Tsinghua University, 1997

[4] T. I. Zohdi, A supplement to finite element class notes, Fall 2002

[5] D. E. Knuth, The art of computer programming, vol.1, fundamental algorithms, Addison-Wesley, Reading, Mass., 1973
[6] M. Veldhorst, An analysis of sparse matrix storage schemes, *Mathematical center tracts* 150, 1982
[7] M. A. Heroux, A proposal for a BLAS toolkit, SPARKER Working note 2, CERFACS, TR/PA/92/90, Technical Report, 1992
[8] Y. Saad, SPARSKIT: A basic tool kit for sparse matrix computations, Technical Report 90-20, Research Inst. Adv. Comp. Science, NASA Ames Research Center, Moffett Field, CA, USA 1990
[9] G. G. Alway, *et al*, An algorithm for reducing the bandwidth of a matrix of symmetric configuration, *Comput. J.*, 8, 264, 1965
[10] F. A. Akyuz, *et al*, An automatic relabeling scheme for bandwidth minimization of stiffness matrices, *J. Amer. Inst. Aeronaut. Astronaut*, 6, 728, 1968
[11] R. Rosen, Matrix bandwidth minimization, Proc. ACM National Conference, Brandon Systems Press, Princeton, N.J.; 585, 1968
[12] E. Cuthill, *et al*, Reducing the bandwidth of sparse symmetric matrices, Proc. ACM National Conference, Association for Computing Machinery, New York, 157, 1969
[13] R. Levy, Resequencing of the structural stiffness matrix to improve computational efficiency, *Jet Propulsion Laboratory Tech. Rev.*, 1, 61, 1971
[14] I. P. King, An automatic reordering scheme for simultaneous equations derived from network systems, *Internat. J. Numer. Mech. Engrg.*, 2, 523, 1970
[15] I. Arany, *et al*, An improved method for reducing the bandwidth of sparse symmetric matrices, Proc. IFIP Conference, North-Holland, Amsterdam, 1246, 1971
[16] H. R. Grooms, Algorithm for matrix bandwidth reduction, *Amer. Soc. Civil Engrg., J. Struct. Div*. 98, 203, ST1 1972
[17] E. Roberts, Relabeling of finite-element meshes using a random process, TMX-2660, National Aeronautics and Space Administration, Lewis Research Center, Cleveland, Ohio, 1972
[18] R. J. Collins, Bandwidth reduction by automatic renumbering, *Internat J. Number. Mech. Engrg.*, 6, 345, 1973
[19] P. T. R. Wang, Bandwidth minimization, reducibility, decomposition, and triangularization of sparse matrices, Ph.D dissertatin, Ohio State University, Columbus, 1973
[20] K. Y. Cheng, Minimizing the bandwidth of sparse symmetric matrices, *Computing*, 11, 103, 1973
[21] A. George, Computer implementation of the finite element method, STAN-CS-71-208, Computer Science Dept., Stanford Univ. Stanford, CA, 1971
[22] S. W. Sloan, An algorithm for profile and wavefront reduction of sparse matrices, *Int. J. Numer. Methods. Eng.*, 23, 239, 1986
[23] N. E. Gibbs, *et al*, An algorithm for reducing the bandwidth and profile of a sparse matrix, *SIAM J. Numer. Anal.*, 13, No.2, 236, 1976
[24] E. Cuthill, Several strategies for reducing the bandwidth of matrices, *Sparse matrices and their applications*, 157, by D. J. Rose, et al (Plenum, New York, 1972)

[25] P. Amestoy, *et al.*, An approximate minimum degree ordering algorithm, *SIAM J., Matrix Anal. Appl.*, 17, 886, 1996
[26] A. George, *et al.*, The evolution of the minimum degree algorithm, *SIAM Rev*. 32, 1, 1989
[27] J. W. H. Liu, Modification of the minimum degree algorithm by multiple elimination, *ACM Trans. Math. Software*, 11, 141, 1985
[28] A. George, Nested dissection of a regular finite element mesh, *SIAM J. Numer Anal.*, 10, 345, 1973
[29] R. J. Lipton, *et al.*, Generalized nested dissection, *SIAM J. Numer. Anal.*, 16,346, 1979
[30] N. Neuβ, *et al.*, A new sparse-matrix storage method for adaptively solving large systems of reaction-diffusion-transport equations, *Computing*, 68, 19, 2002
[31] M. Benzi, Preconditioning techniques for large linear systems: a survey, *J. Comp. Phys*. 182, 418, 2002
[32] G. Manzini., Note on the ordering of sparse linear systems, *Theoretical computer science*, 156, 301, 1996
[33] M. T. Heath, *et al.*, Parallel algorithms for sparse linear systems, *SIAM Review*, 33 (3), 420, 1991
[34] K. V. Camarda, *et al.*, Matrix ordering strategies for process engineering: graph partitioning algorithms for parallel computation, *Computers and chemical engineering*, 23, 1063, 1999
[35] G. Karypis., *et al.*, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, *J. Parallel and Distributed Computing*, 48, 71, 1998
[36] H.B. Gooi, *et al.*, Efficient ordering algorithms for sparse matrix/vector methods, *Electrical Power & Energy Systems*, 20, No1, 53, 1998
[37] W Y Lin, *et al.*, Minimum communication cost reordering for parallel sparse Cholesky factorization, *Parallel Computing*, 25, 943, 1999