12TH INTERNATIONAL DEPENDENCY AND STRUCTURE MODELLING CONFERENCE, DSM'10
22 – 23 JULY 2010, CAMBRIDGE, UK

EFFICIENTLY COMPUTING WITH DESIGN STRUCTURE MATRICES

Shahadat Hossain
Department of Mathematics and Computer Science, University of Lethbridge, Canada

Keywords: sparse data structure, block triangular form, strongly connected component, depth-first search

1 INTRODUCTION

The *design structure matrix or dependency structure matrix (DSM)* provides an effective tool for the representation and analysis of complex system models. In a majority of application areas these matrices are almost invariably sparse in that a large proportion of the entries are identically zero. Algorithms for sparse matrices arising in large-scale problems must exploit the sparsity structure for computational efficiency. Combinatorial structures play an important role in the design of sparse matrix algorithms (Gilbert et. al., 2008). In this paper we borrow novel ideas from combinatorial scientific computing literature and illustrate how they can be applied in *DSM* computations to build new and efficient software tools for analysing the *DSM*. Some of the existing publicly available *DSM* research software use *MATLAB* – an integrated development environment for technical computing. McGill (2005) and Thebeau (2001) discuss *MATLAB* implementation of *DSM* partitioning and clustering where matrices are stored as dense matrices. It is to be noted that *MATLAB,* however, does implement sparse matrix operations using a column-oriented sparse storage of the matrix. Our objective here is to use a general-purpose high-level programming language (e.g., C++) for the implementation of computationally intensive *DSM* algorithms for large-scale problems, thus ensuring maximum portability and extensibility.

Many problems in numerical and combinatorial computing can be modelled by sparse matrices as well as graphs as both of these tools depict dependency relationship among the system elements (Hossain and Steihaug, 2006, 2010; Hossain, 2009). The increasing complexity of product architectures and the large-scale nature of the design processes for new products pose considerable challenges in the planning and realization of their development (Sharman and Yassine, 2004). Some of these challenges are addressed by choosing a suitable representation (or abstraction) of the problem so that novel algorithmic techniques that are successfully applied in solving similar problems in other scientific areas can be used on the problems under discussion. Braha and Bar-Yam (2004, 2007) study statistical properties of directed graphs or networks representing large-scale, complex distributed product development. The duality between sparse matrices and graphs can be exploited in the design and analysis of algorithmic tasks for the underlying computational problems – e.g. in the analysis of complex networks. In this note, the main focus is to identify and explore algorithmic techniques and data structures for efficient representation and computations as required in major *DSM* applications (Braha and Bar-Yam, 2004, 2007; Browning, 2001; Jebala and Eppinger, 1991; Kusiak and Wang, 1993). We propose *Compressed Sparse Row (CSR)* as a unifying data layout scheme for both the *DSM* and its underlying graph, and present precise computational complexity estimates for the *DSM* partitioning algorithms. Specifically, we view the *DSM* computation as two related but separate problems: one that is concerned with the visualization of the pattern of dependencies among the process elements and one that is concerned with the computational tasks of analyzing these relationships. This viewpoint is necessary as for large-scale problems (MacCormack et. al., 2006; Braha and Bar-Yam, 2004, 2007) even storing the *DSM* as a dense matrix may not be feasible.

The two main categories of DSM identified in Browning (2001) are the static DSM where the interacting system elements represent organizational entities or product components and the task-based DSM where the system elements are constrained by precedence relationship. In either case the main computational task can be viewed as a rearrangement of rows and columns of the matrix that optimizes certain objective function. Let $A \in \Re^{n\times n}$ be a *DSM* where the entry in row *i* and column *j*, written *a(i,j)* or a_{ij}, denotes the strength of the interaction between the system elements *i* and *j* with $a(i,j) = 0$ implying no interaction. Without loss of generality, we assume only nonnegative interactions. Let $e_i = (0\ 0\ \dots\ 0\ 1\ 0\ \dots\ 0)^T$ be a vector in $\{0,1\}^n$ where the *i*th entry of the vector, $e_i(i)$, is *1* and all other entries are *0*. Then $P \in \{0, 1\}^{n\times n}$ is called a permutation matrix if its *j*th column $P(:,j) = e_k$ for some $k \in \{1,2, \dots , n\}$ and $P(:,j) \neq P(:,l), j \neq l$.

2 SPARSE DATA STRUCTURES AND GRAPHS

The CSR scheme is one of the common data structures for representing matrices whose sparsity patterns have no known regular structure. This storage scheme can be implemented using three arrays: `value` to store the nonzero entries row-by-row, `colind` that stores the column indices of the nonzero entries in `value`, and `rowptr` that contains the (array) index of the first nonzero element of each row of the sparse matrix in `colind` and `value` arrays. In the *CSR* storage scheme the sparse matrix is compressed by moving the nonzero entries in each row to the left as shown in Figure 1.

$$\begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & a_{13} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{26} & 0 & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{35} & 0 & 0 & 0 \\ a_{42} & a_{44} & 0 & 0 & 0 & 0 \\ a_{51} & a_{53} & a_{55} & 0 & 0 & 0 \\ a_{62} & a_{64} & a_{66} & 0 & 0 & 0 \end{pmatrix}$$

Figure 1: A sparse matrix (left) and its row-compressed form (right).

The data structures to store the example matrix under the *CSR* scheme are shown in Figure 2.

value

a_{11}	a_{13}	a_{14}	a_{22}	a_{26}	a_{31}	a_{33}	a_{35}	a_{42}	a_{44}	a_{51}	a_{53}	a_{55}	a_{62}	a_{64}	a_{66}

colind

1	3	4	2	6	1	3	5	2	4	1	3	5	2	4	6

rowptr

1	4	6	9	11	14	17

Figure 2: CSR data structure.

Array `value` stores the nonzero entries in each row contiguously; array `colind` stores the column indices of the nonzero entries of `value;` array `rowptr` indexes into `colind` array and stores the index location of the first nonzero entry in each row of the matrix. The nonzero entries in row *i* can then be accessed as `value(rowptr(i))` ⋯ `value(rowptr(i+1)-1).` Note that the *CSR* scheme stores only the nonzero entries of the matrix. Let *nnz(A)* denote the number of nonzero entries in matrix *A*. The storage requirement in the *CSR* scheme for $A \in \Re^{n\times n}$ is therefore $2 \times nnz(A) + n + 1$ units of computer memory with row-wise access to the nonzero entries of the matrix. A companion data structure, the *Compressed Sparse Column (CSC),* can be defined to provide column-wise access to the matrix nonzero entries using arrays `rowind` that stores the row indices of the nonzero entries in `value` and `colptr` that indexes into `rowind` and stores the index location of the first nonzero entry in each column of the matrix. The access to the nonzero entries in column j is provided as `value(colptr(j))` $\cdots$ `value(colptr(j+1)-1)`. With both column- and row-wise accesses the storage requirement for a sparse matrix using the compressed row/column scheme is therefore $3 \times nnz(A) + 2 \times n + 2$ units of computer memory, which is consistent with the design principle for sparse linear algebra (Gilbert et. al., 2008). Associated with matrix A is a directed graph $G = (V, E)$ where V is the set of n vertices and there is a directed edge from vertex v_i to vertex v_j, denoted $(v_i, v_j) \in E$ whenever $a_{ij} \neq 0,\ i \neq j$. Then $nnz(A) = |E|$. The first *DSM* computational problem, known in the literature as *partitioning,* is to sequence the tasks in a task-based *DSM* so as to minimize the feedback marks. As a sparse matrix problem the partitioning problem can be stated as,

find a permutation matrix P such that P^TAP is block lower triangular.

$$P^TAP = \begin{pmatrix} A_{11} & 0 & 0 & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & 0 \\ A_{i1} & \dots & \dots & A_{ii} & 0 & 0 \\ \vdots & \vdots & \vdots & \dots & \ddots & 0 \\ A_{k1} & A_{k2} & \dots & \dots & \dots & A_{kk} \end{pmatrix}$$

Figure 3. A sparse matrix in block lower triangular form (bltf).

Figure 3 shows a sparse matrix symmetrically permuted to block lower triangular form in which the diagonal blocks are square submatrices. Without loss of generality we assume that the diagonal entries of the *DSM* are nonzero (see e.g. MacCormack, 2006). Clearly, if the diagonal blocks are scalar quantities (1×1 matrix), then A can be permuted into a lower triangular form i.e., P^TAP is lower triangular and there are no feedback marks - the ideal arrangement of the tasks in a project. It is therefore of interest to know whether the *bltf* of a *DSM* is unique and if the diagonal blocks themselves can be permuted to a *bltf*. Answers to the above questions as well as other *DSM*-related questions are most conveniently provided using graph theory.

A *directed path* from vertex v_i *to* vertex v_j, written, $v_i \mapsto v_j$ is a sequence of vertices ($v_i \equiv v_{i_1}, v_{i_2}, \cdots, v_{i_l} \equiv v_j$) such that $(v_{i_k}, v_{i_{k+1}}) \in E,\ k = 1 \cdots l - 1$ and that the vertices in the path are all distinct. A directed path $v_i \mapsto v_j$ is called a *directed cycle* whenever $v_i = v_j$. A graph $H = (V', E')$ is a subgraph of graph $G = (V, E)$ if $V' \subseteq V$, $E' \subseteq E$ and $(u, v) \in E'$ implies $u, v \in V'$. A subgraph H is said to be *strongly connected* if for each pair of vertices $u, v \in V'$, $u \mapsto v$, $v \mapsto u$. A subgraph H is said to be a *strongly connected component (scc)* if H is strongly connected and no other strongly connected subgraph of graph G properly contains H. Since each vertex can belong to exactly one strongly connected component, strongly connected components of a graph partitions the vertices of the graph and the partition is unique. With the concept of matrix reducibility, the *DSM* partitioning problem can now be viewed as a graph problem. Matrix A is said to be *reducible* if there is a permutation P for which the *bltf*, as in Figure 3, has at least two diagonal blocks. A matrix is *irreducible* if it is not reducible. The following result (see Brualdi and Herbert, 1991) characterizes the *DSM* partitioning as a graph problem.

Theorem 1. Matrix $A \in \Re^{n \times n}$ with nonzero diagonal entries is irreducible if and only if $G(A)$ is strongly connected.

From Theorem 1 we can deduce that each diagonal block A_{ii} represents a strongly connected component of $G(A)$ and therefore cannot itself be permuted into a block triangular form. Several algorithms exist for finding the *bltf* for sparse matrices with varying asymptotic computational complexity. One method, due originally to Harary (see Duff and Reid, 1978), involves computing the power of the *0-1* matrix (a matrix where the nonzero entries are replaced with a *1*) associated with matrix *A* requiring $O(n^3)$ computational effort (Gebala and Eppinger, 1991). A $O(n^2)$ algorithm, due to Sargent and Westerberg (1964), finds *scc*s utilizing the fact that the vertices in a cycle must belong to the same strongly connected component. The first asymptotically optimal algorithm requiring only $O(|V| + |E|)$ computational effort for partitioning a directed graph into its strongly connected components is due to Tarjan (1972). Duff and Reid (Duff and Reid, 1978) discuss an implementation of Tarjan's algorithm. Central to this elegant algorithm is to use a vertex stack to identify and incrementally build strongly connected components during a *depth-first search (dfs)* of the graph. As each vertex is visited (via an edge) for the first time by the *dfs* algorithm it is pushed onto the stack. Using an auxiliary array of size $n = |V|$ the so called *root* of each strongly connected component is computed. Whenever a root vertex is discovered during the return from a recursive call to the *dfs,* all the vertices in the same connected component (including the root vertex) are found in the stack. These vertices, together with the root vertex, are popped from the stack and are assigned a component number. In our example matrix, the vertices labelled *1* and 4 are the root vertices corresponding to the two strongly connected components as depicted in Figure 4. The order in which the depth-first traversal visits the vertices defines the permutation matrix which can be efficiently represented by a permutation vector. For our example matrix *A* the graph *G(A)* consists of two strongly connected components: the first consisting of vertices $\{v_6, v_2, v_4\}$ with v_4 as the root and the second consisting of vertices $\{v_5, v_3, v_1\}$ with v_1 as the root. The corresponding permutation matrix *P* defined by $P(:,1) = e_6$, $P(:,2) = e_2$, $P(:,3) = e_4$, $P(:,4) = e_5$, $P(:5) = e_3$, $P(:,6) = e_1$ can be represented by the vector (*6 2 4 5 3 1*).

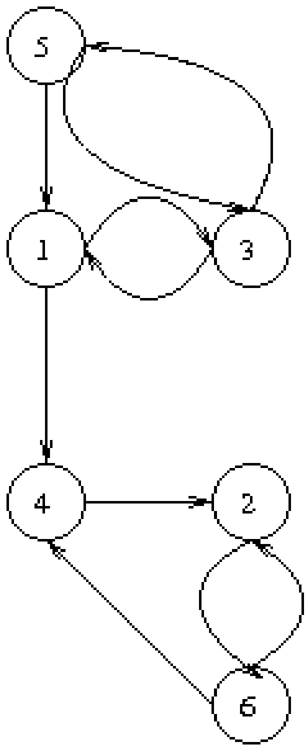


Figure 4. The directed graph associated with the matrix of Figure 1

$$\begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{pmatrix} \quad \begin{pmatrix} a_{66} & a_{62} & a_{64} & 0 & 0 & 0 \\ a_{26} & a_{22} & 0 & 0 & 0 & 0 \\ 0 & a_{42} & a_{44} & 0 & 0 & 0 \\ 0 & 0 & 0 & a_{55} & a_{53} & a_{51} \\ 0 & 0 & 0 & a_{35} & a_{33} & a_{31} \\ 0 & 0 & a_{14} & 0 & a_{13} & a_{11} \end{pmatrix}$$

Figure 5: Block triangular form (on the right) of the example matrix (on the left).

In Figure 5 the original matrix on the left is symmetrically permuted into a block lower triangular form. The top-left diagonal block represents the first strongly connected component and the bottom-right diagonal block represents the second strongly connected component. It is also easy to see that the algorithm sketched above can be used to perform related computation – e.g., *tearing* (Kusiak and Wang, 1993; Gebala and Eppinger, 1991) – without a second pass over the graph vertices. With respect to the implementation of Tarjan's algorithm for *DSM* partitioning we first note that since $G(A)$ is a directed graph, for each vertex v_i we need access to edges of the form (v_i, v_j), $a_{ij} \neq 0$ (also known as outgoing edges) only; therefore, the *CSR* representation of the sparse matrix suffices. Secondly, the main computational step in the algorithm is to access the outgoing edges at each vertex. This is efficiently done by `j = colind(k), k = rowptr(i) ⋯ rowptr(i+1) - 1` to access the vertices adjacent to vertex v_i via the outgoing edges. Therefore, the *CSR* storage scheme provides an efficient implementation of Tarjan's algorithm.

3 CONCLUDING REMARKS

In this paper we have proposed a sparse data structure for efficiently computing with *DSM* matrices and have provided a precise characterization of the computational complexity of the *DSM* partitioning. Although our discussion is centred around partitioning, our proposal extends to other computations such as tearing and clustering (MacCormack et al., 2006). We have also sketched Tarjan's asymptotically optimal partitioning algorithm. For sparse rectangular matrices, a more general partitioning procedure based on bipartite graph matching that utilizes strong Hall property can be found in Pothen and Fan (1990). The research presented in this paper is a preliminary report on the design of a software toolkit for *DSM* partitioning and tearing.

ACKNOWLEDGEMENT

The author wishes to thank the anonymous referees for helpful comments on the presentation of the paper. This research was supported in part by NSERC Discovery Grant.

REFERENCES

Dan Braha and Yaneer Bar-Yam (2004). The Topology of Large-Scale Engineering Problem-Solving Networks. *Physical Review E*, 69(1).

Dan Braha and Yaneer Bar-Yam (2007), The Statistical Mechanics of Complex Product Development: Empirical and Analytical Results. *Management Science*, 53(7):1127-1145, July.

Browning, T.R. (2001). Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions, *IEEE Transactions on Engineering Management*, 48(3), 292-306.

Brualdi R. A. & Herbert J. R. (1991). *Combinatorial Matrix Theory*, Cambridge University Press, Cambridge.

Duff I.S. & Reid J.K. (1978). An Implementation of Tarjan's Algorithm for the Block Triangularization of a Matrix. *ACM Transactions on Mathematical Software*, 4(2):137-147.

Gebala D.A. & Eppinger S.D. (1991). Methods for Analyzing Design Procedures. In L. A. Stauffer, (Ed.), *ASME Design Technical Conferences, 3rd International Conference on Design Theory* *and Methodology*, *DTM '91*, pp. 227-233, Miami, Florida, 22.-25.09.1991.

Gilbert J.R., Shah V.B. & Reinhardt S. (2008). A Unified Framework for Numerical and Combinatorial Computing. *Computing in Science and Engineering*, 10(2):20-25.

Hossain S. & Steihaug T. (2010). Graph Models and their Efficient Implementation for Sparse Jacobian Matrix Determination, to appear in *Proceedings of the Cologne-Twente Workshop on Graphs and Combinatorial Optimization*, Cologne, Germany.

Hossain S. (2009). CsegGraph: A graph colouring instance generator. *International Journal of Computer Mathematics*, 86(10):1956-1967.

Hossain S. & Steihaug T. (2006). Graph coloring in the estimation of sparse derivative matrices: Instance and applications. *Discrete Applied Mathematics*, 156(2):280-288.

Kusiak A. & Wang J. (1993). Efficient Organizing of Design Activities. *International Journal of Production Research*, 31(4):753-769.

MacCormack A., Rusnak J. & Baldwin C.Y. (2006). Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. *Management Science*, 52(7):1015-1030.

McGill E.A. (2005). Optimizing the Closures Development Process Using the Design Structure Matrix, Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA.

Pothen A. & Fan C. (1990). Computing the Block Triangular Form of a Sparse Matrix. *ACM Transactions on Mathematical Software*, 16(4):303-324.

Sargent, R.W.H. and Westerberg, A.W. (1964). "Speed-up" in Chemical Engineering Design. *Trans. Inst. Chem. Engrs.*, 42:190-197.

Tarjan, R.E. (1972). Depth first search and linear graph algorithms. *SIAM J. Comptg.*, 1:140-160.

Thebeau R. E. (2001) Knowledge Management of System Interfaces and Interactions for Product Development Processes, Master's Thesis, Massachusetts Institute of Technology, Cambridge, MA.

Contact: Shahadat Hossain
University of Lethbridge
Department of Mathematics and Computer Science
4401 University Drive
Lethbridge, AB T1K 3M4
CANADA
Phone: (1) 403 329 2476
Fax: (1) 403 317 2882
e-mail: shahadat.hossain@uleth.ca
URL: http://www.cs.uleth.ca/~hossain

Efficiently Computing with Design Structure Matrices

Shahadat Hossain

Department of Mathematics and Computer Science
University of Lethbridge, Alberta
Canada

BY MODELLING DEPENDENCIES
MANAGING COMPLEXITY

Outline

- Motivation and Background
- Efficient Computer Representation for Large-scale DSM
- Problem Formulation
- Strongly-connected Components in Linear Time
- Concluding Remarks

The DSM

The DSM depicts pair-wise dependency between system elements

- Only a small fraction of the possible dependencies are actually present
- The dimension of the matrix is large (10^2 ~ 10^6 or more)
- Important DSM computations have combinatorial components
  - Partitioning: Rearrange the columns and rows to reduce the feed-back marks.
  - Tearing: Remove a subset of marks that optimizes certain objective function

Large-scale Problems (1)

A. MacCormack, J. Rusnak and C. Y. Baldwin, Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code, *Management Science*, 52(7):1015-1030, 2006.

	Mozilla	Linux
Number of source files	1684	1778
Function/source file	17.7	12.8
Memory to store full matrix (4 bytes/element)	13.5 GB	7 GB

Large-scale Problems (2)

The Statistical Mechanics of Complex Product Development: Empirical and Analytical Results.*Management Science.* Vol. 53 (7). pp. 1127-1145. July 2007.

Problem	Dimension	Number of nonzeros	Density (%)
Vehicle	120	417	2.89
Operating Software	466	1245	3×10^{-2}
Pharma. Facility	582	4123	3.4×10^{-3}
Hospital Facility	889	8178	1.3×10^{-3}

Sparse Matrix Representation and Computation

- Computer representation of a sparse matrix should be proportional to max(n, nnz) computer words
  - Coordinate storage: three arrays `I` (row index), `J` (column index), `v` (value) of size `nnz`, each;
  - Compressed row (column) storage: three arrays `rowptr` of size `n+1`, `colind` of size `nnz`, and `value` of size `nnz`
  - Compressed diagonal storage
  - …
- Running time of a basic sparse matrix operation (e.g., sparse matrix multiplied by a dense vector) should be proportional to the size of the data accessed and the number of nonzero floating point operations.

Compressed Sparse Row (CRS) Representation

$$\begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{pmatrix} \quad \begin{pmatrix} a_{11} & a_{13} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{26} & 0 & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{35} & 0 & 0 & 0 \\ a_{42} & a_{44} & 0 & 0 & 0 & 0 \\ a_{51} & a_{53} & a_{55} & 0 & 0 & 0 \\ a_{62} & a_{64} & a_{66} & 0 & 0 & 0 \end{pmatrix}$$

Figure 1: A sparse matrix (left) and its row-compressed form (right).

value

a_{11}	a_{13}	a_{14}	a_{22}	a_{26}	a_{31}	a_{33}	a_{35}	a_{42}	a_{44}	a_{51}	a_{53}	a_{55}	a_{62}	a_{64}	a_{66}

colind

1	3	4	2	6	1	3	5	2	4	1	3	5	2	4	6

rowptr

1	4	6	9	11	14	17

Figure 2: CSR data structure.

- Storage requirement in `CRS` scheme:

`nnz + nnz + n +1`
`= 2 nnz + n + 1`

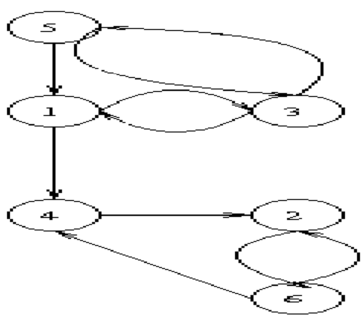
- Column indices of the nonzeros in row `i`:

`colind(l)..colind(h)`

`l = rowptr(i)`
`h = rowptr(i+1)-1`

A DSM and its Graph

$$\begin{pmatrix} a_{11} & 0 & a_{13} & a_{14} & 0 & 0 \\ 0 & a_{22} & 0 & 0 & 0 & a_{26} \\ a_{31} & 0 & a_{33} & 0 & a_{35} & 0 \\ 0 & a_{42} & 0 & a_{44} & 0 & 0 \\ a_{51} & 0 & a_{53} & 0 & a_{55} & 0 \\ 0 & a_{62} & 0 & a_{64} & 0 & a_{66} \end{pmatrix}$$



- For a sparse matrix *A* define a directed graph *G(A) = (V, E)* where
  - There are n vertices: $V = \{v_1, v_2, \ldots, v_n\}$
  - for $a_{ij} \neq 0$, $i \neq j$ there is a directed edge from v_i to v_j, denoted (v_i, v_j) in *E*
- $|V| = n$, $|E| = nnz$

DSM Partitioning

Matrix Partitioning Problem

Given a task-based *DSM A* find a permutation matrix *P* such that the number of feed-back marks in P^TAP is minimized (over all such permutation of the rows and columns of *A)*.

Observations:

- A permutation matrix is a (column/row) permuted identity matrix.
- If P^TAP is lower triangular then the permutation *P* results in optimum arrangement of the tasks.
  - Not all DSM can be permuted to lower triangular form.

"Find a Permutation Matrix *P* Such that P^TAP is Block Lower Triangular"

Block Lower Triangular Form (*bltf*)

$$P^TAP = \begin{pmatrix} A_{11} & 0 & 0 & 0 & 0 & 0 \\ A_{21} & A_{22} & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & 0 & 0 & 0 \\ A_{i1} & \cdots & \cdots & A_{ii} & 0 & 0 \\ \vdots & \vdots & \vdots & \cdots & \ddots & 0 \\ A_{k1} & A_{k2} & \cdots & \cdots & \cdots & A_{kk} \end{pmatrix}$$

Figure 3. A sparse matrix in block lower triangular form (bltf).

- The diagonal blocks are square
- $k = n$ implies P^TAP is lower triangular
- $k = 1$ implies P^TAP yields no improvement w.r.t. feedback marks

Matrix *A* is *reducible* if $\exists$P such that P^TAP has at least two blocks i.e., (k $\geq$2)

$K = 1$ implies irreducible

- Can the diagonal blocks themselves be permuted into *bltf* ?
- Is *bltf* decomposition unique?

The Graph Problem

A *directed path* from vertex v_i *to* vertex v_j, written, $v_i \mapsto v_j$ is a sequence of vertices ($v_i \equiv v_{i1}, v_{i2}, \cdots, v_{il} \equiv v_j$) such that $(v_{ik}, v_{ik+1}) \in E, k = 1 \cdots l-1$ and that the vertices in the path are all distinct.

A path is a *cycle* if the start and the end vertices are the same.

A graph $H = (V', E')$ is a *subgraph* of graph $G = (V, E)$ if $V' \subseteq V, E' \subseteq E$ and $(u, v) \in E'$ implies $u, v \in V'$.

A subgraph H is said to be *strongly connected* if for each pair of vertices $u, v \in V', u \mapsto v, v \mapsto u$.

A subgraph H is said to be a *strongly connected component (scc)* if H is strongly connected and no other strongly connected subgraph of graph G properly contains H.

Theorem (Brualdi and Herbert, 1991).

Matrix $A \in \Re^{n \times n}$ with nonzero diagonal entries is irreducible if and only if G(A) is strongly connected.

Algorithms and Complexity

- Diagonal blocks cannot themselves be permuted into *bltf.*
- Strongly connected component *i* corresponds to the diagonal block $A_{ii;}$

Matrix-based Algorithms:

- *$O(n^3)$* algorithm due to Harary (Harary F., J. Math. Phys. 38, 1959) consists of repeatedly multiplying the binary matrix A with itself.

Graph-based Algorithms:

- $O(n^2)$ algorithm due to Sargent and Westerberg (Sargent R.W.H. and Westerberg A.W., Trans. Ins. Chem. Engrs. 42, 1964)

Optimal Algorithms:

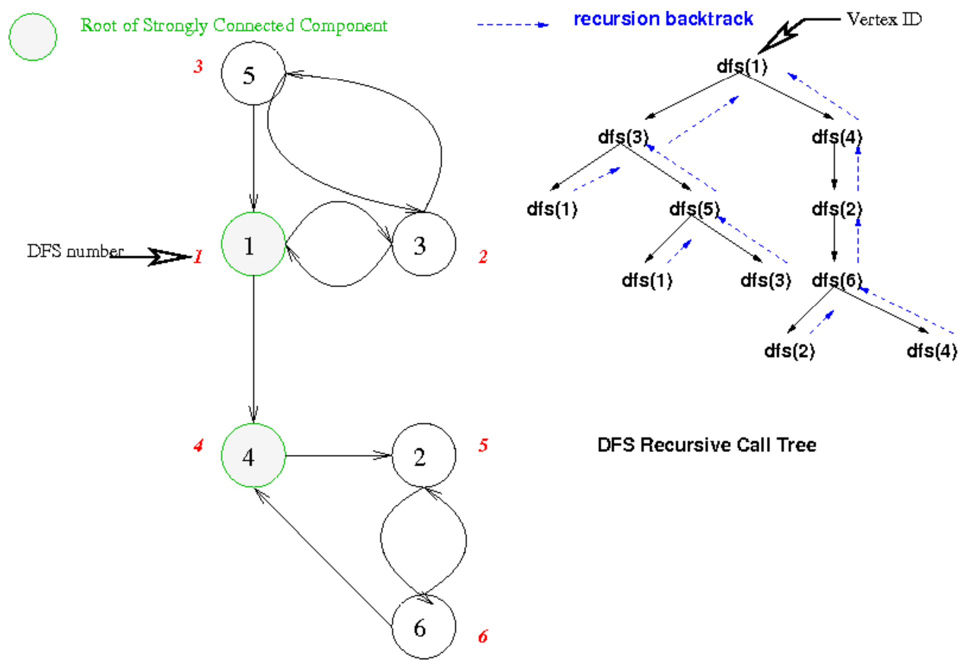
- The first O(n+nnz) algorithm due to Tarjan (R. E. Tarjan, SIAM J. Comput. 1(2), 1972) relies on depth-first search (*dfs*) technique to find strongly connected component.
- The Kosaraju-Sharir (M. Sharir, Computer and Mathematics with Appl., 7(1), 1981) algorithm performs two *dfs* of the graph; the second *dfs* is performed on the modified graph.
- The Cheriyan, Melhorn, Gabow algorithm (J. Cheriyan and K. Melhorn, Algorithmica 15, 1996; H.N. Gabow, Inf. Proc. Let. 74(3-4), 2000) maintains all the *scc*s during *dfs*

Tarjan's Algorithm for finding Strongly Connected Components

```
Alg. SCC_Tarjan
In: Directed graph G = (V,E)
Out: list cmpt[1 .. n]
Method:
  initialize array root of size n;
  initialize array dfsn of size n;
  create an empty stack st of size n;;
  initialize list cmpt;
  compNum ← 1;
  dfsNum ← 1;
  for each vertex v in V do
    if v is not processed then
      dfsn[v] ← dfsNum;
      dfsNum ← dfsNum + 1;
      Dfs(v);
    endif
  endfor
endSCCTarjan
```

```
Alg. Dfs (v)
Method:
  root[v] ← v; push v onto st;
  for each edge (v,w) in E do
    if w is not processed then
      dfsn[w] ← dfsNum;
      dfsNum ← dfsNum + 1;Dfs(w);
    endif
    if w is not assigned to a SCC then
      if dfsn[w] < dfsn[v] then
        root[v] ← w;
      endif
    endif
  endfor
  if root[v] is v then
    pop vertices z from st until vertex v is popped, for
     each such z, cmpt[z] ← compNum;
    compNum ← compNum + 1;
  endif
endDfs
```


Illustration



Notes on Implementation

- For each vertex *v* only the outgoing edges *(v,w)* are needed to access *v*'s neighboring vertices *w*
  - `colind(rowptr(v)) .. colind(rowptr(v+1) - 1)`
- Each edge (nonzero) is accessed only once
- Depth of recursion can at most be *n*
- The root finding stack `st` may never contain more than *n* elements.
- The number of auxiliary arrays can be reduced and can be lumped together into one large array for better *data locality*

Summary and Future Research

Summary

- DSM *partitioning* is equivalent to finding strongly connected components of an associated graph
  - Tarjan's SCC algorithm is asymptotically optimal!
- The CRS scheme provides an efficient implementation for DSM partitioning
  - A graph data structure is not constructed explicitly! same representation suffice for the sparse DSM and its graph.
  - The associated graph being a directed graph allows us to use only row-oriented data access.

Future Research

- Numerical testing for large-scale DSM
- Precise computational complexity of other DSM computations
- New heuristics for computationally intractable DSM computation that exploit "special local structures"
- Software tool development