Mecánica Computacional, Volumen XVIII - pp. 629-638
Compilado por: E. Dari, C. Padra y R. Saliba
San Carlos de Bariloche, noviembre de 1997

AN EFFICIENT IMPLEMENTATION OF THE FIRST LEAST-CONNECTED NODE STRATEGY

Ignacio Ponzoni
Departamento de Ciencias de la Computación - Universidad Nacional del Sur
Avda. Alem 1253 - 8000 Bahía Blanca - Argentina
Planta Piloto de Ingeniería Química (UNS - CONICET)
12 de Octubre 1842 - 8000 - Bahía Blanca - Argentina

Mabel C. Sánchez and Nélida B. Brignole[†]
Planta Piloto de Ingeniería Química (UNS - CONICET)
12 de Octubre 1842 - 8000 - Bahía Blanca - Argentina

RESUMEN

Se presenta en este trabajo la implementación de un nuevo algoritmo para el particionamiento de matrices a forma triangular inferior en bloques especialmente diseñado para problemas de instrumentación de plantas químicas. El nuevo metodo posee como característica fundamental la capacidad de ser aplicado a matrices estructuralmente singulares. El programa fue implementado en lenguaje C lográndose un código eficiente y altamente portable, en tal sentido se presentan los tiempos de ejecución obtenidos para una batería de ejemplos sobre distintas plataformas de ejecución.

ABSTRACT

The implementation of a new algorithm that partitions matrices to a specific block lower-triangular form was carried out. The method was specially designed for applications to process plant instrumentation. Its main feature is its capacity to deal with structurally singular matrices. The software was implemented in C language, the resulting code being highly portable and efficient. In this respect, the run-times for a series of examples executed under different platforms are presented.

Key Words: *Block Triangular Form, Sparse Matrices, Partitioning Algorithms, Graphs.*

INTRODUCTION

The rearrangement of sparse matrices to yield special forms, such as triangular, tridiagonal and block triangular, is a subject of major concern in many computational applications. In the field of process system instrumentation in particular, a block lower-triangular form (BlTF) with the pattern shown in figure 1 is necessary to classify the unmeasured variables. The diagonal

[†] Author to whom all correspondence should be addressed. E-mail: dybrigno@criba.edu.ar

blocks of the BITF must be square, excepting the last one. A comprehensive discussion on the advantages and drawbacks of the existing partitioning algorithms that could be used to perform this task can be found in Ponzoni *et al* [1]. This study reveals that no procedure is good enough to deal with structurally singular matrices efficiently. Since they frequently arise in this application, the development of better algorithms became a necessity.

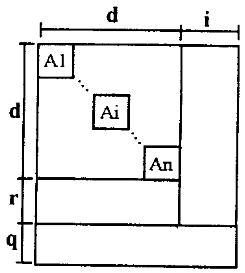

Fig 1. Occurrence matrix **M** in its block lower-traingular form.

Based on Romagnoli and Stephanopoulos' paper [2], which led to PLADAT's package [3], we have proposed a new partitioning strategy, called Global Strategy with First Least-Connected Node (GS-FLCN), that is specially adapted to cope with this problem [4]. The new method makes a incremental decomposition of the occurrence matrix **M**. First, the classical Forward Triangularization method detects removable 1x1 blocks. Then, GS-FLCN uses two procedures for 2x2 and 3x3 removable subsets. We have developed those algorithms by modifying Stadtherr's proposals [5]. Finally, a new algorithm, called First Least-Connected Node (FLCN), has been designed to look for blocks of order 4 or greater. The latter procedure is based on a depth-first search with heuristics through an undirected graph corresponding to $\mathbf{M}^T\mathbf{M}$. This approach overcomes the above-mentioned limitations and is more robust because the strategy succeeds in finding the maximum number of blocks of minimum size.

In this paper, an efficient implementation of the new strategy (GS-FLCN) for the permutation of an $m \times n$ sparse matrix to the lower-triangular block structure shown in figure 1 is described. The technique can be applied to general matrices. The new algorithmic ideas, as well as a comparison with PLADTA's assignment algorithm (DCB) can be found in Ponzoni *et al.* [6].

MODULAR DESCRIPTION OF THE GLOBAL STRATEGY WITH FLCN

The structure of the programme is modular. Figure 2 shows its hierarchical tree. The root (level zero) corresponds to the main procedure called FindSubSets. This routine guides the subset search. Level one contains the roots of all the subtrees that search for subsets of a definite size, which has been indicated between brackets in the figure. The flow control explores each subtree by preordering from the far left. If no subsets are found, the control is transferred to the nearest subtree on the right, whenever it exists. Otherwise, the programme ends. Whenever a subtree locates a subset, the control is moved to the subtree situated to the far left, which is the one that contains the Forward-Triangularization root. In the appendix, we present the main algorithms of the new Global Strategy with FLCN in a Pascal-like style.

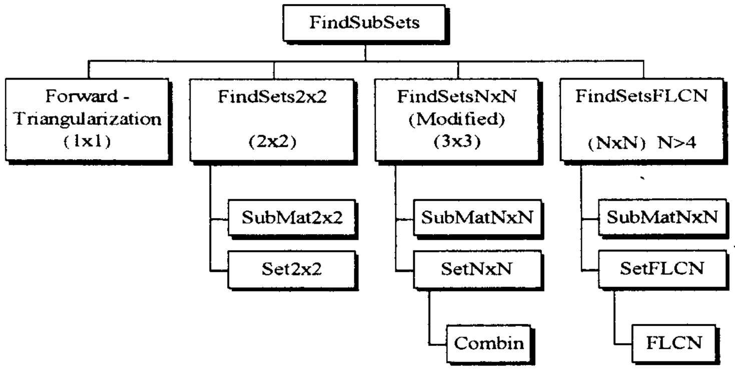

Fig. 2: Hierarchical Structure of the Programme.

IMPLEMENTATION

This section deals with the main features of the implementation. Firstly, the methodology employed for the development of the code is described, and secondly the auxiliary package created to handle sparse matrices is presented. Finally, structural information regarding dynamic memory assignment and portability of the code is provided.

Methodology

At the first stage, the MATLAB software for numeric computation was used. Since it is an interpreter, it is ideal to develop prototypes in the study of experimental algorithms.

Once the methods had been outlined, the code was translated to C language for various reasons. First of all, the language is strong enough to support huge arrays of more than 64 Kb, thus making it possible to handle the enormous data structures that often appear in many application areas. Besides, the implementation is not only highly portable to machines with different sets of instructions and/or operating systems but it is also compatible with existing software. It is also efficient as regards execution times and memory savings.

It is interesting to note that the algorithm requires structural matrix rearrangements. As the available packages for sparse matrices are concerned with mathematical operations, it was necessary to develop auxiliary routines. Several data structures for sparse matrix representation were compared so as to find the most economical ones for our specific purpose. In the cost assessment for the most frequently used operations, time was prioritized over storage. After finishing the supporting code, the whole strategy for subset search was implemented, applying a Top-Down development policy.

Tests have been performed on examples that explore not only the subtrees growing from level one (Figure 2) but also the flow control among them. Some instances were taken from the classical bibliography on the subject and others were generated ad hoc so as to ensure that all parts of the code worked satisfactorily.

Sparse Matrices

A set of subroutines for the storage and handling of sparse matrices was tailor-made for the algorithm.This auxiliary mini-package called SPMATRIX optimizes the run-times for those operations most often required by the main routines. These tasks are listed in Table I, together with their purpose and the corresponding run-time orders of magnitude. The parameter SM identifies a pointer to the sparse matrix structure. The constants p_r and p_c are real because they represent the average amount of non-zeros in a row or column respectively. In most cases, $2 \leq p_r, p_c \leq 4$. The quantities n_r and n_c represent the number of rows and columns contained in the parameters *rows* and *cols* respectively.

A significant advantage of this implementation is the achievement of several constant-order operations ($O(1)$). Moreover, the order of the operations DeleteRow and DeleteCol can also be regarded as constant due to p_r and p_c's low values and small ranges.

Table I: Description and Performance of SPMATRIX's Routines.

Function	*Purpose*	*Run-time*
DeleteRow (SM,r)	Deletes SM's r-th row.	$O(p_r)$
DeleteCol (SM,c)	Deletes SM's c-th column.	$O(p_c)$
CountRow (SM,r)	Returns the number of non-zeros in SM's r-th row.	$O(1)$
CountCol (SM,c)	Returns the number of non-zeros in SM's c-th column.	$O(1)$
QDeleteRow (SM,r)	Returns True, if the row has been deleted False, otherwise	$O(1)$
QDeleteCol (SM,c)	Returns True, if the column has been deleted False, otherwise	$O(1)$
NumRowSM (SM)	Returns SM's number of rows.	$O(1)$
NumColSM (SM)	Returns SM's number of columns.	$O(1)$
MakeSubMatrix (SM,SubSM,*rows*,*cols*)	Builds a submatrix containing the rows and columns from SM indicated by the vectors *rows* and *cols*.	$O(p_r.n_r+p_c.n_c)$

In this way, the proposed implementation improves the run times of the assignment algorithms. This representation is the most appropriate for these problems because the performance of the operations that bear the burden of the computational cost has been optimized.

Recursion and Dynamic Memory Assignment

The efficient handling of memory resources becomes critical in recursive algorithms because the programme's execution stack can grow considerably when deep recursivity levels are reached, i.e. for several nested recursive calls. In this case, the dynamic memory assignment is of the utmost importance since it ensures that only the strictly necessary amount of space is being allocated and released at the right moment.

Thanks to the dynamic assignment of all its data structures, this implementation offers two significant advantages. First, the programme uses the minimum amount of memory. If the structures had been allocated statically, they should have been defined for the worst case, usually resulting in costly overdimensioned arrays. In this case, the static handling of memory would have been particularly inefficient due to the recursivity, because it is impossible to foresee the depth that will be reached during the calculations. Another major advantage is the fact that the memory is set free as soon as the data structures are no longer necessary. In this way, there is always a maximum amount of space available.

Portability

The programme was developed in a Pentium PC with a 133 Mhz processor clock rate under an MSDOS 6.2 operating system. It was compiled with MicroSoft C 7.0, using the huge option for memory management. If the user is going to work with small or medium-size problems and he needs faster runnings, he should recompile the software after performing the changes in pointers and memory allocation indicated in the next paragraph.

To ensure the portability of the code, it was recompiled and run both in a Pentium machine under a LINUX operating system and in an ALPHA DEC 3000 MODEL 300/AXP with a 150 Mhz processor clock rate under an OSF/1 operating system. To enable successful execution in a UNIX environment, i.e. for LINUX and OSF/1, the pointers type "huge" must be redefined as "standard" and the functions for memory allocation must be modified as follows:

The function	__huge *	must be changed to	*,
The function	_halloc(n,size)	must be changed to	malloc(n*size),
The function	_hfree(ptr)	must be changed to	free(ptr).

EXAMPLES

The code worked satisfactorily for a variety of test cases. The performance under all the above-mentioned configurations is illustrated in this section. The first problem, which was used by Pissanetzky [7] to test Tarjan's algorithm, corresponds to the matrix given in figure 3.

$$\begin{bmatrix}
1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&0&0\\
1&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&1&1&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&1&1&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&1\\
0&0&0&0&0&0&1&1&0&0&0&1&1&0&0\\
0&0&0&0&0&0&0&1&0&0&0&1&1&1&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&1
\end{bmatrix}$$

Fig. 3. Case Study 1.

Case study 2, in particular, is associated to an industrial plant. The sector consists of two reactors R1 and R2 and two heat exchangers HX1 and HX2. Figure 4 shows a schematic representation of the process.

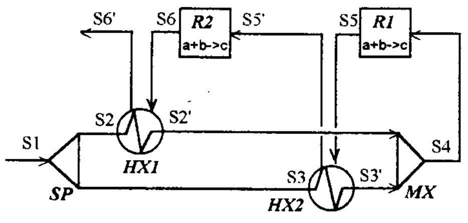

Fig. 4. Process Topology.

Table II : Model Variables

VARIABLES							
1	a S1	9	T S6'	17	U R2		
2	b S1	10	a S5	18	T S2		
3	c S1	11	b S5	19	T S3		
4	T S1	12	c S5	20	Fr S2		
5	T S4	13	U R1	21	Fr S3		
6	T S5	14	a S6	22	T S2'		
7	T S5'	15	b S6	23	T S3'		
8	T S6	16	c S6				

Table II lists the process variables. There are 3 compounds (a, b, c) and 10 streams (1, 2, 2', 3, 3', 4, 5, 5', 6, 6'). The letter T stands for temperature, Fr means flowrates and U represents the heat transfer coefficients.

In this case, CDB's strategy could only identify two 1x1 subsets linking equation 9 with variable T S2 and equation 10 with variable T S3. In contrast, GS-FLCN succeeded in finding all the assignment subsets reported by Joris [8]. The results are shown in Table III.

Table III: Assignment subsets for case study 2.

		M E A S U R E D V A R I A B L E S									U N M E A S U R E D V A R I A B L E S													
Eq.	Balances	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	29	21	22	23
R1	Mass a	x									x			x										
	Mass b		x									x		x										
	Mass c			x									x	x										
	Energy	x	x	x		x	x				x	x	x											
R2	Mass a										x				x			x						
	Mass b											x				x		x						
	Mass c												x				x	x						
	Energy							x	x		x	x	x		x	x	x							
SP	Energy 1-2				x														x					
	Energy 1-3				x															x				
	Mass																				x	x		
HTX1	Energy	x	x	x					x	x					x	x	x		x		x		x	
HTX2	Energy	x	x	x			x	x			x	x	x							x		x		x
MX	Energy	x	x	x		x															x	x	x	x

Case studies 3 and 4 belong to a group of examples specially developed for these tests. The structure of the corresponding matrices is shown in Figures 5 and 6.

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}$$

Fig. 5. Case Study 3.

$$\begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}$$

Fig. 6. Case Study 4.

Table IV sums up all the results. It can be concluded that GS-FLCN yields the best partitions. It is interesting to note that in cases 1, 3 and 4 CDB could not even solve the problem.

Table IV. Characteristics of the assignments obtained for some test problems

		ALGORITHMS					
		Size CS		CDB		GS-FLCN	
Case	Source	row	col	n°sets	size	n°sets	size
1	Pissanetzky	15	15	0	---	1 1 1 1	5 1 7 2
2	Joris	14	14	2	1	2 3	1 4
3	This paper	12	12	0	---	1 1 3 1	6 2 1 3
4	This paper	15	14	0	---	1 1 1 6	3 1 4 1

RUN-TIMES AND CONFIGURATIONS

The dependence of the run-times upon several execution platforms is given in table V for the examples presented in the previous section. In all cases, the ALPHA machine exhibited the best performance as regards execution times. It is evident that MATLAB's prototype is computationally expensive. The reported values reveal the improvements that can be achieved by using different platforms. Configuration 3 is more powerful than configuration 2 mainly because UNIX is a 32-bit operating system whereas MSDOS works with 16 bits. It is interesting to note that configurations working under UNIX do not have the constraints on the maximum size of the arrays that limit the performance under MSDOS. Nevertheless, it should be noted that the results are also influenced by the quality of the code generated by the compilers.

Table V: Run-Times (in miliseconds)

	Config. 1	*Config. 2*	*Config. 3*	*Config. 4*
Machine	Pentium	Pentium	Pentium	ALPHA
Operating System	MSDOS	MSDOS	LINUX - UNIX	OSF/1 - UNIX
Implementation •	MATLAB version 4.0	Microsoft C version 7.0	GNU project C version 2.7	DEC C Compiler
Case Study 1	49930	380	60	50
Case Study 2	2250	50	10	less than 1 μsec
Case Study 3	152470	1150	180	133
Case Study 4	5280	50	10	less than 1 μsec

CONCLUSIONS

An efficient implementation of a new strategy for the rearrangement of sparses matrices to a specific BITF is presented in this paper. In contrast with previous works, the novel procedure, called GS-FLCN, is able to deal with structurally singular matrices satisfactorily. It is also more robust because it succeeds in finding the maximum number of blocks of minimum size. The implementation is highly portable and can handle huge matrices. Besides, memory resources were optimized by using dynamic allcation of data structures.

REFERENCES

[1] **Ponzoni, I., Sánchez, M. C. and Brignole, N. B.,** *Algoritmos Eficientes en Problemas de Asignación,* Mecánica Computacional, Vol. XV, 1995, pages. 415-424.

[2] **Romagnoli, J. A. and Stephanopoulos, G.,** *On the Rectification of Measurement Errors for Complex Chemical Plants,* Chem. Eng. Sci., Vol. 35, 1980, pages. 1067-1081.

[3] **Sánchez, M. C., Bandoni, A. J. and Romagnoli, J. A.,** *PLADAT: A Package for Process Variable Classification and Plant Data Reconciliation,* Comp. Chem. Eng., 1992, pages S499-S506.

[4] **Ponzoni, I., Sánchez, M. C. and Brignole, N. B.,** *A New Partitioning Algorithm for Classification of Variables in Process Plant Monitoring,* accepted for its presentation in the AIChE 1997Annual Meeting, Los Angeles, Estados Unidos. November 16-21, 1997.

[5] **Stadtherr, M. A., Gifford, W. A. and Scriven, L. E.,** *Efficient Solution of Sparse Sets of Design Equations,* Chem. Eng. Sci., Vol. 29, 4 , 1974, pages 1025-1034.

[6] **Ponzoni, I., Sánchez, M. C. and Brignole, N. B.,** *Estudio del Desempeño de Algoritmos de Clasificación de Variables No Medidas en Problemas de Instrumentación,* accepted for its presentation in the ENPROMER'97, Bahía Blanca, Argentina. September 1-4, 1997.

[7] **Pissanetzky, S.,** *Sparse Matrix Technology,* Chapter 5, Academic Press Inc., 1984.

[8] **Joris, P., and Kalitventzeff, B.,** *Process Measurement Analysis and Validation,* Proc. of XVIII Congress on The Use of Computers in Chem. Eng., Italy, 1987, pages 41-46.

APPENDIX: GLOBAL STRATEGY WITH FLCN

Algorithms

Global Strategy

```
1. Construction of the Occurrence Matrix.
2. Forward-Triangularization.
3. Set n=2
3.1 Call Construction of the Occurrence n-
    Submatrix.
3.2 Call Subroutine 2.
    (location of 2x2 removable subsets)
    If any subsets are found
       then go back to step 2.
       else go to step 4.
    end if
4. Set n = 3
4.1 Call Construction of the Occurrence n-
    Submatrix.
4.2 Call Modified Algorithm (parameter n)
    (location of 3x3 removable subsets)
    If any allowed subsets were detected
      then go back to step 2.
      else go step 5.
    end if
5. Set n = 4
   5.1 Call Construction of the Occurrence n-
       Submatrix.
   5.2 Apply step 1 of Subroutine 2.
   5.3 Call FLCN Algorithm - (n, submatrix)
       (location of nxn removable subsets)
       If any allowed subsets were detected
         then go back to step 2.
         else
           If (n > maximum size of subset)
             then Stop and return the classification.
             else set n = n+1 and go to step 5.1.
           end if
       end if
```

---

1. *Construction of the Occurrence Matrix*

```
1-Make up an occurrence matrix for the system
  of equations to be solved by filling position (i,j)
  with a 1 if the j-th variable appears in the i-th
  equation. Otherwise place a 0.
2-Define a rearranged occurrence matrix that will
  contain the removable subsets found on
  completion of the work.
```

---

2. *Forward-Triangularization*

```
1-Find a row in the occurrence matrix with only
  one entry. This entry represents a removable
  1x1 subset.
2-If this subset is allowed then remove it by
  deleting the row and column in which it occurs,
  and place it in the first free row and column in
  the reordered matrix.
3-Repeat the steps 1 y 2 until no more entries
  can be added to the new matrix.
```

3.1 *Construction of the Occurrence n-Submatrix*

```
1-Choose all rows in the occurrence matrix that
  contain n nonzeros at the most, together with
  the columns in which those elements appear.
2-The columns with only one nonzero are deleted.
  The removal might lead to rows with only one
  nonzero. Those rows are also eliminated. The
  procedure is repeated until all rows and
  columns contain at least two elements.
```

3.2 *Subroutine 2: location of 2x2 subsets*

```
1-Find a column with only one entry. Delete it
  together with the row in which it appears.
  Delete all columns without entries. Repeat this
  step until no columns with less than two entries
  remain in the submatrix.
2-Find the column of the submatrix with the
  greatest number of entries. Delete it and all the
  rows in which it has entries. Look for a column
  whose number of entries has been reduced by
  two. This column along with the deleted
  column and the two rows containing common
  entries, forms a removable 2x2 subset.
3-If this subset is allowed
    then
      Delete the second column that had been
      found. Place these rows and columns in
      the first two free rows and columns of the
      reordered matrix.
    else
      go to step 2.
  end if.
```

4 - Repeat steps 2, 3 and 4 until the submatrix is empty, which indicates that all the 2x2 removable subsets have been located.

4.2 *Modified Algorithm: location of nxn subsets*

```
1 - Apply step 1 of Subroutine 2.
2 - Find the column with the greatest number of
    entries and delete it. Delete the rows in which
    that column has entries. If there are n-1
    columns whose number of entries has been
    reduced by one or more, and if n rows are
    empty after those columns have been removed,
    delete those n-1 columns temporarily. The n
    deleted columns and the n empty rows form a
    removable nxn subset.
3 - If this subset is allowed
      then
        Place these rows and columns in the first
        n free rows and columns of the reordered
        matrix.
        Eliminate the columns that had been
        temporarily deleted in step 2 .
      else
        Recover the columns that had been
        temporarily deleted in step 2.
        Go back to step 2.
    end if.
4 - Repeat steps 2, 3 and 4 until the submatrix is
    empty. This means that all the nxn removable
    subsets have been located.
```

5.2 *First Least-Connected Node Algorithm*

Input Data:

submatrix	matrix found in the previous step;
n	size of the desired subset;
node	current node;
stamps	logical array that indicates whether a node has been visited.

Output Data:

success	logical variable valued: true: if an *nxn* removable set has been located false: otherwise;
rows	array containing the row numbers of the detected subset;
columns	array containing the column numbers of the detected subset.

```
FLCN(submatrix, n, node, stamps, success)

If n = 0
 then
   If there are n empty rows after having deleted
      the n columns associated with the n nodes in
      the path.
    then an nxn removable subset made up of
        the n empty rows and the n columns
        associated with the n nodes in the way
        has been found.
      If IsAllowable(SubSet)
       then
         Assign the n empty rows to the nxn set.
         success = true
       else
         success = false
      end if
    else
      success = false
   end if
 else
   finish = false
   While (not finish) and (not success) do
    [*]Choose an unstamped node adjacent to node.
     If there are several adjacent nodes, pick out
        the least-connected one. If there are
        several least-connected nodes, select the
        one with the lowest label.
     If there is no such node
       then
         finish = true
         success = false
       else
         node = the node chosen in [*].
         stamps(node) = true
         FLCN(submatrix, n-1, node,
                 stamps, success)
         If success
          then
            Define the column associated with
            node of the nxn set as the n-th column
         end if
     end if
   end while
end if
```