University of Thessaly
Faculty of Engineering
Department of Electrical & Computer Engineering

Study of Direct Methods for Solving Sparse Linear Systems

Diploma Thesis

ATHANASIOS POLYCHRONOU

Supervisor
Panagiota Tsompanopoulou
Associate Professor

Volos, July 2020

University of Thessaly
Faculty of Engineering
Department of Electrical & Computer Engineering

Study of Direct Methods for Solving Sparse Linear Systems

Diploma Thesis

ATHANASIOS POLYCHRONOU

Volos, July 2020

University of Thessaly
Faculty of Engineering
Department of Electrical & Computer Engineering

Athanasios Polychronou,
July 2020

Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Μελέτη Άμεσων Μεθόδων για την Επίλυση Αραιών Γραμμικών Συστημάτων

Διπλωματική Εργασία

ΑΘΑΝΑΣΙΟΣ ΠΟΛΥΧΡΟΝΟΥ

Επιβλέπουσα
Παναγιώτα Τσομπανοπούλου
Αναπληρώτρια Καθηγήτρια

Βόλος, Ιούλιος 2020

Abstract

Computing large linear systems in a reasonable amount of time and space is still an ongoing challenge for computational scientist. In general, it is very common practice that the systems that need to be computed are sparse. Exploiting, sparsity to achieve better execution times and less storage space is one of the key factors in cutting-edge computer algorithms. This thesis, focuses on the theory behind those algorithms, the data structured and the algorithms itself which are currently used by sparse system libraries. More specific the Direct Methods Cholesky, QR, LU for factorizing sparse linear matrices are examined and methods to reduce fill-in as well. Afterwards, experimental cases are performed in Matlab to emphasize the benefits of sparsity exploitation.

Keywords

direct methods, sparse matrices, fill-in, graph theory of sparse matrices, Cholesky Factorization, QR Factorization, LU Factorization, fill-in, multifrontal methods, supernodal methods, Matlab

Περίληψη

Ο υπολογισμός μεγάλων γραμμικών συστημάτων σε ένα λογικό χωρικό και χρονικό πλαίσιο είναι μια διαρκής πρόκληση για τους επιστήμονες της υπολογιστικής μηχανικής. Γενικά, είναι πολύ συχνό φαινόμενο τα συστήματα που χρειάζεται να υπολογιστούν να είναι αραιά. Ένας από τους βασικούς συντελεστές στους κορυφαίας κατασκεύης υπολογιστικούς αλγορίθμους για επίτευξη καλύερων χρόνων εκτέλεσης, καθώς και λιγότερου αποθηκευτικού χώρου, είναι η εκμετάλλευση του αραιής δομής των συστημάτων. Αυτή η διπλωματική εστιάζει στην θεωρία πίσω από αυτούς τους αλγορίθμους, στις δομές δεδομένων καθώς και στους ίδιους τους αλγορίθμους που περιλαμβάνονται στις βιβλιοθήκες λογισμικού επίλυσης αραιών συστημάτων. Πιο συγκεκριμένα, αναλύονται οι άμεσοι μέθοδοι Cholesky, QR, LU όπου επιλύουν ένα σύστημα παραγοντοποιόντας τον αραιό πίνακα. Επίσης, εξετάζονται μέθοδοι ελαχιστοποίησης fill-in στοιχείων που προκύπτουν κατά την παραγοντοποίηση. Έπειτα, εκτελούνται μερικές πειραματικές περιπτώσεις σε Matlab οι οποίες δίνουν έμφαση στα πλεονεκτήματα των παραπάνω αλγορίθμων.

Λέξεις Κλειδιά

άμεσοι μέθοδοι, αραιοί πίνακες, θεωρία γραφημάτων για αραιούς πίνακες, Cholesky Παραγοντοποίηση, QR Παραγοντοποίηση, LU Παραγοντοποίηση

Acknowledgements

I would like to thank associate professor Panagiota Tsompanopoulou for her advice and assistance throughout the whole paper.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

A **Sparse Matrix** is defined as a matrix which most of the elements are zero and the non zero elements of a matrix may be concentrated or distributed less systematically resulting in a regular or irregular shape.Thus,we shall refer to a matrix as sparse if taking advantage of the percentage of its non-zero elements and shape results in solving a problem more economically.

Sparse matrices are usually encountered in a wide variety of problems in engineering, machine learning as well as scientific areas of study. Problems from the real world that are formulated with the use of partial differential equations are commonly solved with the use of finite element method. Typical problems solved in such a way are structural analysis,heat transfer, fluid flow to name just a few. Also, sparse matrices often arise for example due to encoding or in processing data that contains counts or even in language processing in the field of machine learning.

Thus, the construction of algorithms or the modification of existing ones in a way that sparsity is exploited to solve these linear systems more efficiently is a thriving field of study. Algorithms which solve linear systems are divided into two subgroups. The first group consists of *Direct Methods* and the second one of *Iterative Methods*. A *Direct Method* is one which gives an exact solution to the system assuming no round-off error occur and an *Iterative Method* is one which uses successive approximate solutions obtaining a more accurate solution in each step.

1.1 Main Objective

This thesis focuses on studying in depth the **Data Structures** as well as the the **Algorithms** used for obtaining the **Direct Solution** of a sparse system. More importantly, ways to reduce the storage cost, as well as the execution cost of typical factorization methods, in regards to the sparsity of a matrix, are inspected.

The reduction of storage cost is of high importance because computer systems have physical memory limitations, due to their hardware, and many linear systems tend to be arbitrarily big, exceeding the memory capacity of the system. However, if a considerable amount of entries in the element are zeros, they can be discarded and thus, the remaining entries can be stored without causing memory overflows.

Lowering the actual time of an algorithm's execution time is a desired fact for every algorithm.

Exploiting the sparsity can lead to huge difference in execution time of many common algorithms operating on sparse systems. The theory behind those modified algorithms is analyzed in depth, and some pseudo-code examples are given.

The important fact is that the underlying theory discussed is the basis of all the cutting edge matrix solvers, used currently by numerical analysis and numerical computing software, such as MATLAB. Thus, after the theory, some test-cases are executed in MATLAB, to show the benefits mentioned above.

1.2 Thesis Structure

In Chapter 2, some basic linear algebra and graph theory is discussed.

In Chapter 3, the data structures for sparse matrices and basic matrix operations are shown.

In Chapter 4, the solution of a sparse triangular system is implemented, which is very significant because subsequent algorithms rely a lot on it.

From Chapter 5 the modified matrix factorizations are discussed, how matrices are decomposed into a product of more convenient matrices. Starting from the Cholesky factorization, then moving on to the QR factorization in Chapter 6 and finally the LU factorization in Chapter 7. These three are the predominant chapters of the thesis.

In Chapter 8 some methods to reduce fill-in entries are shown. Fill-in entries are those who arise due to the factorization of a matrix and are not in the original matrix.

Finally, in Chapter 9 the experiments in MATLAB are done.

Chapter 2

Theoretical Background

2.1 Introduction

In this section some basic linear algebra and graph theory will be described as well as the notations used in this thesis. More information can be found in [64][32].

2.2 Linear Algebra

A real m-by-n *matrix* is denoted as : $A \in \mathbb{R}^{m\ x\ n}$. An entry in the A *matrix* in *row i* and *column j* corresponds to a_{ij}. The notation a_{ij},a_i or a will be used for row or column *vectors* or for scalars depending on the context.

The symbols for *lower upper* and *triangular* are $L\ U$ respectively. In L for every l_{ij} with $i > j$ holds $l_{ij} = 0$. In U for every u_{ij} with $i < j$ hold $u_{ij} = 0$. The *diagonal* of a matrix $A \in \mathbb{R}^{m\ x\ n}$ is the set of entries $a_{kk}|\{k = 1, ..., n\}$.
The *identity matrix* is denoted as $I \in \mathbb{R}^{n\ x\ n}$ whose *diagonal* is 1 and all the other entries are 0.

The *transpose* of a vector,otherwise a row vector, is a^T and the *transpose* of a matrix is and $A^T \equiv a_{ji}$ for every a_{ij}. A_{i*} and A(i,:) is the whole i row of a matrix. Likewise, A_{*j} and A(:,j) is the whole j column.

Matrix Addition is defined as $A = B + C$ where $a_{ij} = b_{ij} + c_{ij}$. Matrix-Scalar multiplication is defined as $A = cB$ where $a_{ij} = c * b_{ij}$.Matrix-Matrix multiplication is defined as $A = BC$ where if $B \in R^{m\ x\ p}$ and $C \in R^{p\ x\ n}$ then $A \in R^{m\ x\ n}$ and $a_{ij} = \sum_{k=1}^{p} b_{ik} * c_{kj}$. Notice that the number of *columns* of B and *rows* of C must agree.

For two vectors $x, y \in \mathbb{R}^n$ the *inner product* is defined as $x^T y = \sum_{k=1}^{n} x_k * y_k$.
The inverse of a matrix is denoted A^{-1} where $A^{-1}A = AA^{-1} = I$. If a matrix Q holds that $Q^T * Q = I$ then Q is called *orthonormal*. If Q is a square orthogonal matrix then also $QQ^T = I$ holds and $Q^T = Q^{-1}$, this matrix is an *orthogonal* matrix.

P denotes a *permutation* matrix meaning an *identity* matrix I with its *rows* or *columns permuted*.

The symbol $|A|$ denotes the number of *non-zero* elements in matrix A. The set containing the *non-zero* pattern of a matrix A is denoted with $\mathcal{A}$.

2.3 Graph Theory

Any entry in a graph is called a *node* and any line connecting two separate nodes is called an *edge*. Denoting $V = \{1, 2, ..., n\}$ as a set of nodes and $E = \{(i,j)|i,j \in V\}$ as a set of edges. In a *directed* graph the *edge* connecting two nodes has a specific direction from node A to node B for instance. In an *undirected* graph any edge is a path from node A to B as well as from B to A.
If a path from $i \rightsquigarrow j$ exists, it means that there are *edges* connecting the nodes $(i, k, ..., l, j)$. A *cycle* in a graph is a path $j \rightsquigarrow j$, starting and ending at the same node.

A graph with no cycles is called *acyclic*, a *directed acyclic graph* is called DAG. A forest is an *undirected graph* in which any two vertices are connected by at most one path,or equivalently a DAG. A tree is an *undirected graph* n which any two vertices are connected by exactly *one path*, or equivalently a *connected acyclic undirected graph*. Thus, a *forest* is a *disjoint union* of *trees*.

Usually the *tree data structures* used are *rooted trees* where *root* refers to the very first node of the data structure. *Parent* of a node V is the first node connected to V on the path to the *root*. The parent of every node is unique except the root which has no parent. Equivalently, the *child* of a node V is a node whose *parent* is V. A node with no children is called a *leaf.*

A *descendant node* of a node is any node in the path from that node to the leaf node, including the leaf node. An *ancestor node* of a node is any node in the path from that node to the root node, including the root node as well. Thus, the first *ancestor* of a node is its *parent*.

A *clique* is a subset of vertices of a graph G where every two vertices of the clique are adjacent to each other.
The set of all nodes in a graph G reached by a node V is denoted as $Reach_G(V)$.

Chapter 3

Data Structures and Basic Algorithms

3.1 Introduction

In this chapter the basics of which data structures are used to store sparse matrices efficiently and how the basic matrix algorithms are modified according to this specific type of structure.

3.2 Data Structures

Considering a random sparse matrix $A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 5 & 8 & 0 \\ 3 & 0 & 11 & 0 \\ 0 & 4 & 3 & 17 \end{bmatrix}$

This matrix could be stored in a *triplet* form data structure which for a matrix $A \in \mathbb{R}^{m\ x\ n}$ every non-zero row and column index is stored as well as every value. *Triplet* form is easily read and created and many files containing sparse matrices use this method of storage. For this particular example:
row= $\begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 4 & 4 & 4 \end{bmatrix}$, col= $\begin{bmatrix} 1 & 2 & 3 & 1 & 3 & 2 & 3 & 4 \end{bmatrix}$,
value= $\begin{bmatrix} 1 & 5 & 8 & 3 & 11 & 4 & 3 & 17 \end{bmatrix}$.

The size of each array is equal to the number of *non-zero* entries.In the example above is equal to 8. This storage method is highly inefficient for use in most sparse matrix algorithms It preferable to use another storing method that suits those algorithms more. Thus, let us introduce the *Compressed Sparse Column Format* or *CSC Format*[7].

In the *CSC Format* three matrices are used, one Cp of size $n+1$ for storing the column pointers and in the final position the $nz+1$ value, one *i_indices* of size *nz* for storing the row indices of the non-zero entries and finally one *val* of size *nz* storing the values of the non-zero entries.

The logic behind this format is the following: in the Cp matrix an entry Cp[i] translate to the the place in the rowIdx matrix where the i_{th} column is stored.So, row indices of column *i* are placed from rowIdx[Cp[i]] to rowIdx[Cp[i+1]-1]. Equivalently, the numerical values are placed from val(Cp(i)) to val(Cp(i+1)-1). For the example above, the corresponding matrices are:

Cp= $\begin{bmatrix} 1 & 3 & 5 & 8 & 9 \end{bmatrix}$, rowIdx= $\begin{bmatrix} 1 & 3 & 2 & 4 & 2 & 3 & 4 & 4 \end{bmatrix}$,.
val= $\begin{bmatrix} 1 & 3 & 5 & 4 & 8 & 11 & 3 & 17 \end{bmatrix}$.

The values of the third column of A, A_{*3} can be accessed and printed by:

```
for p=Cp(3):Cp(4)-1
    print(val(p))
end
```

The output is: p=5 val[5]=8, p=6 val[6]=11, p=7 val[7]=3
We observe that for storing the matrix above two arrays of size $nz = 8$ are required and one of size $n+1 = 5$. Already, from the storage method it easy to note that a lot less space is used for Cp=[] opposed to col=[] for matrices with a large number of non-zeros.
Transforming a matrix from *triplet* to a *CSC* format is easily done with the following algorithm[23].

```
function C=TripToSparse(A)
    row=A.row
    value=A.value
    col=A.col
    for k=1:nz
        count(col(k))++
    end
    Cp=cumsum(count)
    for k=1:nz
        p=col(k)++
        rowIdx(p)=row(k)
        val(p)=value(k)
    end
end
```

Where cumsum function computes the cumulative sum of a matrix. For instance for T=[2,4,5], the cumsum(T) equals [2,6,11].
The above function is simpler model of the function S=sparse(A) in Matlab stated just for observing the logic behind *CSC* format and understanding the differences between the former and the *triplet* format.
From now on let's use for a matrix A the notation Ai for *rowIdx Ap* for *column pointers Cp* and Cx for *val*.

3.3 Matrix-Vector Multiplication

Suppose we want to computer $z = Ax + y$. We have A stored by column so if we consider A split into column vectors the above equation is $z = [A_{*1}\ A_{*2}\ \cdots\ A_{*n}] * [x_1 x_2\ \cdots\ x_n]^T + y$.
So to compute it the following algorithm is used:

```
function z=gaxpy(A,x,y)
    for k=1:n
        for p=Ap(k):Ap(k+1)−1
            y(Ai(p))=Ax(p)∗x(k)
        end
    end
end
```

We see that the two loops are taking time proportional to the non-zero entries in A, so $O(n+f)$, where f is the number of float point operations[23].

3.4 Matrix Multiplication

Since a matrix is stored by column the multiplication method used is *Column at a Time*. For calculating $C = AB$ column-wise, $C_{*j} = AB_{*j}$ for $j = 1 \cdots n$ is computed. Considering the split used in the previous section $C_{*j} = AB_{*j}$ becomes $C_{*j} = [A_{*1}\ A_{*2}\ \cdots\ A_{*n}] * [b_{1j}\ b_{2j} \cdots\ b_{nj}]^T$. The non-zero pattern of C_{*j} is the set union of A_{*j} for all i which b_{ij} is non-zero. Thus, $\mathcal{C}_j = \bigcup_{i \in \mathcal{B}_j} \mathcal{A}_i$. For computing $\mathcal{C}$ numerical cancellation is ignored. So the resulting matrix has a new non-zero pattern which must be computed simultaneously with multiplication.

```
function C=mat_multiply(A,B)
    nz=0
    for j=1:n
        Cp[j]=nz
        for k=Bp(j):Bp(j+1)−1
            b=Bx(k)
            for p=A(k):A(k+1)−1
                i=Ai(p)
                if(i not in nz pattern of col_j)
                    Ci[nz++]=i
                x(i)+=b∗Ax(p)
            end
            for p=Cp(j):nz
                Cx(p)=x(Ci(p))
            end
        end
    end
    Cp[n]=nz
end
```

The time taken is proportional to n, $|B|$ as well as f which notes the count of floating point operations. Therefore, the algorithm is $O(f + n + |B|)$[23].

3.5 Matrix Addition

Matrix addition is a similar to Matrix Multiplication if the following transformation from $C = \alpha A + \beta B$ to $C = [A\ B][\alpha I\ \beta I]^T$ is noted.

```
function C=mat_add(A,B)
    nz=0
    for j=1:n
         Cp(j)=nz
         for Ap(j):Ap(j+1)-1
              i=Ai(p)
              if(i not in nz pattern of col_j)
                   Ci[nz++]=i
              x(i)+=alpha*Ax(p)
         end

           for Bp(j):Bp(j+1)-1
              i=Bi(p)
              if(i not in nz pattern of col_j)
                   Ci[nz++]=i
              x(i)+=beta*Bx(p)
         end
         for p=Cp(j):nz
              Cx(p)=x(Ci(p))
         end
    end
end
```

3.6 Solving a Triangular System Lx=b

Let's consider a system $Lx = b$ where L is *square* and *lower triangular*. Taking into account the storing format used, the matrix should be accessed by column to solve the system efficiently. Let us consider the following decomposition:

$$\begin{bmatrix} \ell_{11} & 0 \\ \ell_{21} & L_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

The equations deriving are $\ell_{11}x_1 = b_1$ and $\ell_{21}x_1 + L_{22}x_2 = b_2$, where ℓ_{11}, x_1 and b_1 are **scalars** and ℓ_{21}, x_2 and b_2 are **vectors** of size $n-1$. L_{22} is a $(n-1)x(n-1)$ matrix. Unwinding the recursion leads to an algorithm that accesses the matrix L *column-by-column*. Hence, solving for x leads to $x_1 = b_1/\ell_{11}$ and $L_{22}x_2 = b_2 - \ell_{21}x_1$ and observing that b is used once so it is more convenient to replace it with x, the following algorithm is introduced:

```
function x=lsolve(A,b)
```

```
    x=b
    for j=1:n
        x(j) =x(j)/Lx(Lp(j))
        for p=Lp(j)+1:Lp(j+1)-1
            x(Li(p))=x(Li(p))-Lx(p)*x(j)
        end
    end
end
```

Let's see a more detailed example for a 3*x*3 matrix. $\begin{bmatrix} \ell_{11} & & \\ \ell_{21} & \ell_{22} & \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$

Considering the previous notation: ℓ_{11}, x_1, b_1 are the same, $\boldsymbol{\ell}_{21} = [\ell_{21}\ \ell_{31}]$, $\mathbf{x}_2 = [x_2\ x_3]$, $\mathbf{b}_2 = [b_2\ b_3]$ and $\mathbf{L}_{22} = \begin{bmatrix} \ell_{22} & 0 \\ \ell_{32} & \ell_{33} \end{bmatrix}$.

Following the algorithm: $x_1 = b_1/\ell_{11},\ b_2 - \ell_{21}x_1 = \begin{bmatrix} b_2 \\ b_3 \end{bmatrix} - \begin{bmatrix} \ell_{21} \\ \ell_{31} \end{bmatrix} x_1$, that's the first iteration followed by recursion unwinding so now: $\boldsymbol{\ell}_{11} = \ell_{22},\ \boldsymbol{\ell}_{21} = [\ell_{32}], \mathbf{x}_2 = [x_3], \mathbf{b}_2 = [b_2]$ and $\mathbf{L}_{22} = [\ell_{33}]$.

Solving, results in: $x_2 = b_2/\ell_{22},\ b_2 - \ell_{21}x_1 = b_3 - \ell_{32}x_2$ and then the final step is solving $x_3 = b_3/\ell_{33}$.

It easy to note in the above example that the columns of of L are accessed one time each from left to right which is the desired way, this is called a *forward solve*.

3.7 Solving a Triangular System Ux=b

Considering the system $Ux = b$ where U is an *upper triangular* matrix and x, b are vectors. A similar decomposition is used as the one for the L matrix. $\begin{bmatrix} U_{11} & u_{21} \\ 0 & u_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$

The equations derived are: $u_{22}x_2 = b_2$ and $U_{11}x_1 + u_{21}x_2 = b_1$, where u_{22}, x_2, b_2 are scalars and u_{21}, x_1, b_1 are vectors of length $n - 1$, U_{11} is a $(n-1)x(n-1)$ matrix. Solving for x results in $x_2 = b_2/u_{22}$ and $U_{11}x_1 = b_1 - u_{21}x_2$ and the algorithm deriving is:

```
function x=usolve(A,b)
    x=b
    for j=n:1
        x(j)=x(j)/Lx(Li(j+1)-1)
        for p=Up(j):Up(j)-2
            x(Ui(p))=x(Ui(p))-Ux(p)*x(j)


        end
    end
end
```

In the algorithm above, the columns of U are accessed one at a time from right to left, that is called a *backward solve*.

3.8 Solving a Triangular System L^{T}x=b

The matrix L^T is an upper triangular matrix because L is a lower triangular. It's easy to observe that because L is stored by *column*, now L^T is stored by *row*. So, it's optimal way to access this matrix is done by row. Let's consider once again the decomposition used in section 3.6 but with matrix L transposed.

$$\begin{bmatrix} \ell_{11} & \ell_{21}^T \\ 0 & L_{22}^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

The equation the above system produces are: $L_{22}^T x_2 = b_2$ and $\ell_{11}x_1 + \ell_{21}^T x_2 = b_1$, note that $\ell_{21}^T x_2$ is an **inner product** that results in a scalar. Solving for x produces: $L_{22}^T x_2 = b_2$ and $x_1 = (b_1 - \ell_{21}^T x_2)/\ell_{11}$. Hence, the algorithm is:

```
function x=ltsolve(A,b)
    x=b
    for j=n:1
        for p=Lp(j)+1:Lp(j+1)-1
            x(j)=x(j)-Lx(p)*x(Li(p))
        end
        x(j)=x(j)/Lx(Lp(j))
    end
end
```

The algorithm iterates from n to 1 because as it is a backward solve, also accessing a row at at time resembles the common way of solving a problem of this kind.

3.9 Solving a Triangular System U^{T}x=b

Since U is an *upper triangular* stored by *column*, U^T is a *lower triangular* and as above it is optimal to access it by *row*. Considering the decomposition used in section and transposing the matrix results in:

$$\begin{bmatrix} U_{11}^T & 0 \\ u_{21}^T & u_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Solving for x derives the following equations: $U_{11}^T x_1 = b_1$ and $x_2 = (b_2 - u_{21}^T x_1)/u_{22}$ where again $u_{21}^T x_1$ is an **inner product**. The algorithm for this system is:

```
function x=utsolve(A,b)
    x=b
    for j=1:n
        for p=Lp(j)+1:Lp(j+1)-2
            x(j)=x(j)-Ux(p)*x(Ui(p))
    end
    x(j)=x(j)/Ux(Up(j+1)-1)
end
```

The algorithm iterates from 1 to n as it is a forward solve. Once again this algorithm resembles the common way of solving this kind of systems, because many readers are familiar with solving a system by rows and not columns.

Chapter 4

Sparse Lower Triangular Solve

4.1 Introduction

The algorithms used in the previous chapter, only took into consideration the **sparsity** of the *left-hand* side matrix, but it is very common in practice to have a **sparse** *right-hand* side as well. It is of high importance to come up with an algorithm that exploits these features to their fullest because the **lower triangular solve** is a basis for plenty factorization algorithms.

4.2 Sparse Right-Hand Side

For a sparse b and assuming that the **diagonal** of the L matrix is unity, the pseudo-code for solving the system is:

```
x=b
for j=1:n
    if $x_j \neq 0$
    for each i>j for which $\ell_{ij} \neq 0$
            $x_i = x_i - \ell_{ij} * x_j$
```

An algorithm based in the above pseudo-code would take $O(n + f + |b|)$. Floating operations dominate $|b|$ so $O(n + f)$. But this is not very efficient because if there is a single non zero element in b, f is $O(1)$ but the whole loop would be executed again so the total work would be $O(n)$ which is clearly unacceptable.

Suppose all the $x_j \neq 0$ where known before hand and were sorted in ascending order, then the j loop could be avoided using the following pseudo-code:

```
x=b
for each $j \in \mathcal{X}$
    for each i>j for which $l_{ij} \neq 0$
            $x_i = x_i - l_{ij} * x_j$
```

The complexity of this algorithm is $O(|b| + f)$, which is basically $O(f)$[42]. It's a great time improvement but a way to find $\mathcal{X}$ must be implemented.

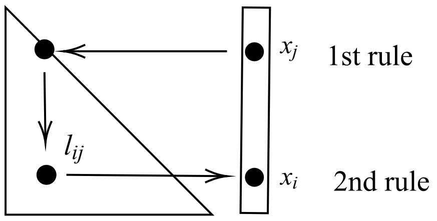

Figure 4.1: Rules Example

4.3 Finding the non-zero set $\mathcal{X}$

4.3.1 Determining $\mathcal{X}$

The entries of **x** become non-zero only in two cases, ignoring numerical cancellation:

1. $b_i \neq 0 \implies x_i \neq 0$
2. $x_j \neq 0 \land \exists(l_{ij} \neq 0) \implies x_i \neq 0$

The first rule is pretty straight forward, if there is a non-zero entry in b_i then the corresponding x_i is also non-zero. The second rule implies that in an line equation for instance:
$l_{i1}x_1 + \cdots + l_{ij}x_j + \cdots + l_{ii}x_i = 0$. If at least one $x_j \neq 0$ then $x_i = -\frac{l_{ij}}{l_i i}x_j$, so $x_i \neq 0$.

These rules that define the set $\mathcal{X}$ can be expressed as a graph traversal problem. Suppose there is an acyclic directed graph $G_L = (V, E)$ where $V = \{1 \cdots n\}$ and $E = \{(j,i) \mid l_{ij} \neq 0\}$. For each $x_j \neq 0$ the corresponding node j is marked. So from the first rule all nodes $\{i \mid i \in \mathcal{B}\}$ are marked. The second rule means that all nodes which can be reached from a marked node shall also be marked. Hence, $\mathcal{X}$ is the set union of all nodes that can be reached from the nodes in $\mathcal{B}$. In graph terminology this is symbolized by $\mathcal{X} = Reach_L(\mathcal{B})$[32].

4.3.2 Computing $\mathcal{X}$

Considering the graph G_L, doing a depth-first search of G_L starting at nodes $\mathcal{B}$, can compute the set $\mathcal{X}$. Depth-First Search (*DFS*) algorithm starts at the *root* node and explores as far as possible along each branch before backtracking. Time used by *DFS* is analogous to the number of edges traversed, plus the number of the initial starting nodes, the nodes belonging in $\mathcal{B}$. Each edge translates into two floating point operations since an edge reflects the second rule hence, the total time taken is $O(|b| + f)$. Because for each $\{(i,j) | j < i\}$ in $x_i = x_i - l_{ij}x_j$, x_j must be first computed, the *DFS* algorithm must be done in a **topological order**, an order that preserves precedence. In topological order, for every edge $i \rightarrow j$, node i comes before node j in the ordering. So the pseudo-codes for computing $\mathcal{X}$ are shown below[23]:

```
function $\mathcal{X}$=reach(L,$\mathcal{B}$)
    for each i for which $b_i \neq 0$
        if node i is unmarked
```

```
          dfs(i)

function dfs(j)
    mark(j)
    for each i for which $l_{ij} \neq 0$
        if node(i) is unmarked
            dfs(i)
    push j onto stack for $\mathcal{X}$
```

Note that **dfs(i)** is a recursive function, but beware because recursive function can create stack overflow for very large inputs, so an iterative approach that mimics the recursion is suggested though recursion is used here since for easier comprehension.

4.4 Solving a Sparse System

Now, a way has been found for calculating $\mathcal{X}$ so the pseudo-code for solving the lower triangular system from section 4.2 can be rephrased to the following algorithm:

```
function x=lsparse_solve(L,B,k)
    X=cs_reach(L,B,k)
    for p=Bp(k):Bp(k)-1 % b is stored in a CSC format
        x(Bi(p))=Bx(p)  % so it is scattered
    end
    for s=1:length(X)
        j=X(s)
        x(j)=x(j)/Lx(Lp(J));
        for p=Gp(j)+1:Gp(j+1)-1
            x(Li(p))=x(Li(p))-Lx(p)*x(j)
        end
    end
end
```

In a similar manner, the algorithm for solving an upper triangular can be implied, just be changing the forward solve to a backward one. The algorithm takes an optimal $O(|b| + f)$ time to execute. An example follows showing all the theory discussed above. Consider the following system Lx=b:

$$\begin{bmatrix} \ell_{11} & & & & & & & \\ & \ell_{22} & & & & & & \\ & \ell_{23} & \ell_{33} & & & & & \\ \ell_{41} & & & \ell_{44} & & & & \\ & & & \ell_{45} & \ell_{55} & & & \\ \ell_{61} & & & & & \ell_{66} & & \\ & & \ell_{73} & & & \ell_{76} & \ell_{77} & \\ \ell_{81} & & & & & & \ell_{87} & \ell_{88} \end{bmatrix} x = \begin{bmatrix} 0 \\ 0 \\ b_3 \\ b_4 \\ 0 \\ b_6 \\ 0 \\ 0 \end{bmatrix}$$

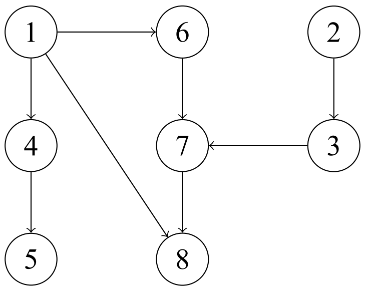

Figure 4.2: Corresponding Graph of G_L

.

So, the set $\mathcal{B}$ is $\{3, 4, 6\}$. Calculating the **Reach** for each node in $\mathcal{B}$ results in: $Reach(3) = \{3, 7, 8\}$ $Reach(4) = \{4, 5\}$, and finally $Reach(6) = \{6, 7, 8\}$, but nodes $\{7, 8\}$ are already marked from $Reach(3)$. The final output is stored in topological order, thus $\mathcal{X} = \{6, 4, 5, 3, 7, 8\}$. Note that, due to topological ordering node 6 comes before node 7,8 although they are derived from the first **Reach**. Observing the matrix it is easy to see that x_6 and x_3 are needed for computing x_7, which is why the node precedence must be maintained.

Chapter 5

Cholesky Decomposition

5.1 Introduction

A matrix *decomposition* or *factorization* is a way of *reducing* a matrix into its constituent parts. That way, complex matrix operations can be performed on decomposed matrix rather than the original matrix itself which leads to easier computations. Additionally, if a plenty of systems include a specific matrix it is convenient to decompose it once, which is the computationally intense part, and then reuse it for as many times as needed. In this Thesis three decomposition methods will be examined, starting with **Cholesky Decomposition** in this chapter. The order in which the methods are presented is in a way that is favors the gradually development of their theory.

5.2 Method Overview

For a Cholesky Decomposition to be applied to a real matrix A two criteria must hold:

1. The matrix must be symmetric, $A^T = A$.
2. The matrix must be positive definite, $(\mathbf{x^T A x}) > 0$ for every non-zero vector $\mathbf{x}$.

Then the decomposition is the product $\mathbf{LL^T} = \mathbf{A}$ where L is a lower triangular matrix with positive diagonal entries. Note that for a n-by-n A matrix only its lower triangular component needs to be stored after the decomposition which saves a big amount of space. Also, $Ax = b \implies LL^Tx = b$ which is a lower and followed by an upper triangular solve, which is easier than solving an arbitrary square system.

If a a 2-by-2 decomposition is used in a similar manner like the chapters above $LL^T = A$ results in:

$$\begin{bmatrix} L_{11} & \\ \ell_{12}^T & \ell_{22} \end{bmatrix} \begin{bmatrix} L_{11} & \ell_{12} \\ & \ell_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} \\ a_{12}^T & a_{22} \end{bmatrix},$$

ℓ_{12} and a_{12} are vectors of size n, a_{22} and ℓ_{22} are scalars and L_{11} and A_{11} are matrices of size $(n-1)x(n-1)$. The deriving equations from the above system are: $L_{11}L_{11}^T = A_{11}, L_{11}\ell_{12} = a_{12}$

and $\ell_{12}^T\ell_{12} + \ell_{22}^2 = a_{22} \implies \ell_{22} = \sqrt{a_{22} - \ell_{12}^{\mathbf{T}}\ell_{12}}$, where $a_{22} > \ell_{12}^T\ell_{12}$ due to A being positive-definite. The algorithm yielding a solution to the system is:

```
function L=up_cholesky(A)
    n=size(A)
    L=zeros(n)
    for k=1:n
        L(k,1:k-1)=(L(1:k-1,1:k-1)\A(1:k-1,k))';
        L(k,k)=sqrt(A(k,k)-L(k,1:k-1)*L(k,1:k-1)')
    end
end
```

This algorithm is called up because it is *up-looking*, meaning it looks at $(k-1)$ rows before constructing kth row of L. It can be seen from the above equations and the algorithm that a triangular solve is used. If A is a sparse matrix that leads to a sparse triangular solve and the theory developed in Chapter 4 can be used but for this algorithm the theory can be taken a step further.

5.3 Elimination Tree

Consider again the equation $L_{11}\ell_{12} = a_{12}$ of a sparse matrix. After the sparse triangular solve the vector ℓ_{12}^T becomes the 2nd or generally the kth row of L. So $\mathcal{X} = Reach_L(\mathcal{B})$ becomes any given row k, $\mathcal{L}_k = Reach_{L_{k-1}}(\mathcal{A}_k)$ where $\mathcal{L}_k$ is the non-zero pattern of line k, L_{k-1} is the already calculated part, in the above decomposition L_{11}, as well as $\mathcal{A}_k$ is the non-zero pattern of the upper triangular part of the kth column of **A**. Bearing in mind this pattern the subsequent relation is established.

Consider any $i < j < k$ where $a_{ik} \neq 0$ and $\ell_{ji} \neq 0$ corresponding to the Figure 5.1. Traversing the graph corresponding to L_{k-1} would start from node i since $i \in \mathcal{A}_k$, hence $i \in \mathcal{L}_k$. Then, the traversal would visit node j because $\ell_{ji} \neq 0$ so from the 2nd rule in 4.3.1 $x_j \neq 0$ thus $j \in \mathcal{L}_k$. Thereafter, the computed vector **x** will become the kth row of L so $\ell_{ki} \neq 0$ and $\ell_{kj} \neq 0$. So, two non-zeros in column i imply that there is a non-zero in column j(ℓ_{kj}), $\ell_{ki} \neq 0 \wedge \ell_{ji} \neq 0 \implies \ell_{kj} \neq 0$.

So the following two rules derive for a Cholesky Decomposition $LL^T = A$ and neglecting numerical cancellation :

1. $a_{ij} \neq 0 \implies \ell_{ij} \neq 0$

2. $i < j < k \wedge \ell_{ji} \neq 0 \wedge \ell_{ki} \neq 0 \implies \ell_{kj} \neq 0$

Note that from the second rule, in the L matrix appear some non-zero entries which are not in the original A matrix, these are called **fill-in** entries.

In graph notion the above translates in an edge (i,j) and an edge (i,k) imply and edge (j,k). If there is a path from j to k the path from i to k is redundant to compute $Reach(i)$ because i would be reached via j. This of course does not affect the $Reach(t)$ of any other nodes t<i. Thus, only the least numbered node j with $j > i$ is needed to compute $Reach(i)$. The previous sentence leads to an

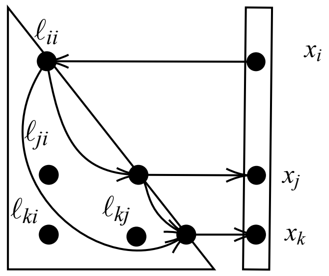

Figure 5.1: Elimination Tree Relations

remarkable observation that any node would have have at most one outgoing edge and since two vertices are connected by exactly one path, this results in a **tree**.

The tree produced is called an **Elimination Tree**. Node j is the parent of node i in the tree, where j is the first off-diagonal entry in the i^{th} column. If a column has no off-diagonal entries it has no parent thus it is a tree by itself. So, an *elimination tree* is actually an **elimination forest** but the former term is used. For a Cholesky Matrix L its elimination tree is denoted as $\mathcal{T}$ as well as for any sub-matrix $L_{1 \ldots k\ 1 \ldots k}$ of L is denoted as $\mathcal{T}_k$. Computing $\mathcal{L}_k$ only takes $O(\mathcal{L}_k)$ time using the **Elimination Tree** which of course is a better time complexity than using the method from the previous chapter. Having established the **Elimination Tree** the theorems and methods to compute it efficiently follow[54].

5.4 Computing the Elimination Tree

Before proceeding to the computational part the subsequent theorems, which will enable us to compute the tree efficiently, must be established. Also, bear in mind that numerical cancellation is once again neglected[52][58][61].

1. Considering a Cholesky Decomposition $LL^T = A$, if $\ell_{ki} \neq 0 \land k > i \implies i$ is a *descendant* of k in $\mathcal{T}$ and the path $i \rightsquigarrow k$ exists in $\mathcal{T}$.
2. The non-zero pattern $\mathcal{L}_k = Reach_{L_{k-1}}(\mathcal{A}_k) = Reach_{\mathcal{T}_{k-1}}(\mathcal{A}_k)$
3. Node j is a leaf of $\mathcal{T}^k \iff a_{jk} \neq 0 \land a_{ik} = 0$ for every *descendant* i of j in $\mathcal{T}$.
4. Considering $LL^T = A$, if $a_{ki} \neq 0$ and $k > i \implies i$ is a *descendant* of k in $\mathcal{T}$ and the path $i \rightsquigarrow k$ exists in $\mathcal{T}$.

Let us examine closer the previous rules. The 1st rule is easier understood by looking at Figure 5.2. Let j be the *parent* of i that is the first off-diagonal entry in column i with $\ell_{ji} \neq 0$. The parent must exist because $\ell_{ki} \neq 0$ holds. If $k = j$ then $k = parent(i)$ and $i \rightsquigarrow k$ is indeed a direct path in $\mathcal{T}$. For $(k > j > i) \land (\ell_{ki} \neq 0) \land (\ell_{ji} \neq 0)$ and recalling rule 2 from previous chapter implies $\ell_{kj} \neq 0$ so the path $j \rightsquigarrow k$ exists and combined with (i,j) edge results in a path $i \rightsquigarrow k$ in $\mathcal{T}$. So removing removing redundant edges to obtain $\mathcal{T}$ has no effect in the *Reach* of any node.

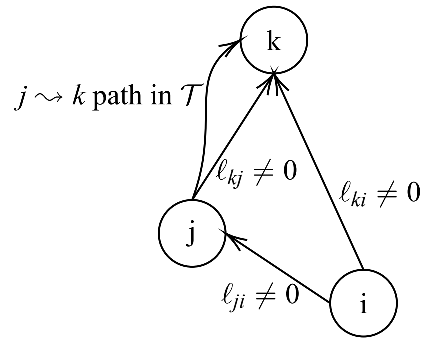

Figure 5.2: 1st Rule

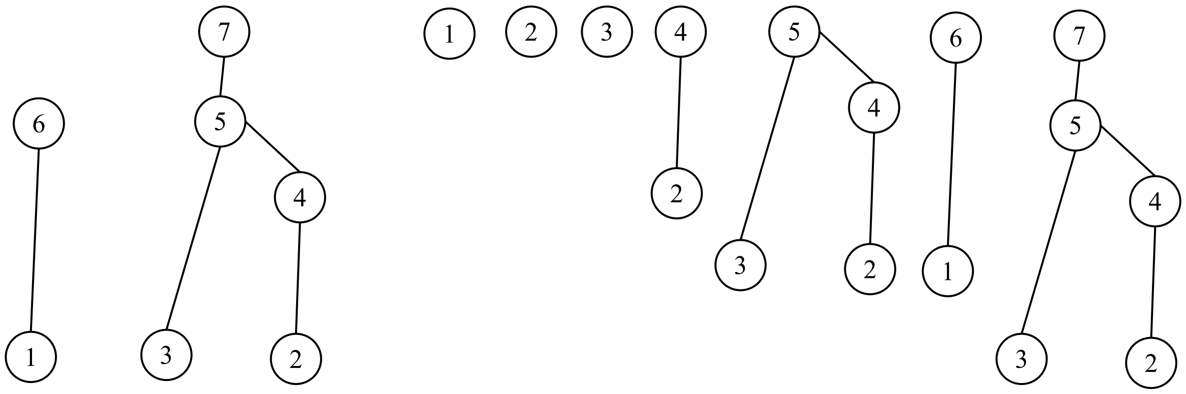

Figure 5.3: Elimination Tree of A and Row Subtrees

From the above statements the 2nd as well as 3rd rules derive easily and show that any sub-tree $\mathcal{T}^k$ is characterized by its leaves. They are also leading to the 4th rule which is quite similar to the first only this time the relation can be established from the original matrix.

Let us view an example in order to comprehend better the concept of the **Elimination Tree**. Consider the following matrix and its corresponding **Cholesky** Decomposition:

$$A = \begin{bmatrix} a_{11} & & & & & a_{16} & \\ & a_{22} & & a_{24} & a_{25} & & a_{27} \\ & & a_{33} & & a_{35} & & a_{37} \\ & a_{42} & & a_{44} & a_{45} & & a_{47} \\ & a_{52} & a_{53} & a_{54} & a_{55} & & a_{57} \\ a_{61} & & & & & a_{66} & \\ & a_{72} & a_{73} & a_{74} & a_{75} & & a_{77} \end{bmatrix} L = \begin{bmatrix} \ell_{11} & & & & & & \\ & \ell_{22} & & & & & \\ & & \ell_{33} & & & & \\ & \ell_{42} & & \ell_{44} & & & \\ & \ell_{52} & \ell_{53} & \ell_{54} & \ell_{55} & & \\ \ell_{61} & & & & & \ell_{66} & \\ & \ell_{72} & \ell_{73} & \ell_{74} & \ell_{75} & & \ell_{77} \end{bmatrix}$$

The **Elimination Tree** is in Figure 5.3, where we observe that its actually a forest and that no fill-in entries exist, thus no element came from rule 2 of previous chapter. Also, each row sub-tree $\mathcal{T}^k$ for $k = 1 \cdots 7$ in ascending order.

Another example is given below. Once again consider the matrix A and its Choleksy Factor L. The vector *parent* denotes is the **Elimination Tree** parent-child relations, where *parent*(i) is the parent of node i in the tree, for example *parent*$(2) = 3$ and *parent*$(10) = 0$ because 10 is the **root** of $\mathcal{T}$:

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & & & a_{16} & & & & a_{110} \\
a_{21} & a_{22} & & & & a_{26} & & & & a_{210} \\
a_{31} & & a_{33} & & & & & & & \\
& & & a_{44} & a_{45} & & a_{47} & & a_{49} & a_{410} \\
& & & a_{54} & a_{55} & & a_{57} & & & \\
a_{61} & a_{62} & & & & a_{66} & & & a_{69} & a_{610} \\
& & & a_{74} & a_{75} & & a_{77} & & a_{79} & \\
& & & & & & & a_{88} & a_{89} & a_{810} \\
& & & a_{94} & & a_{96} & a_{97} & a_{98} & a_{99} & \\
a_{101} & a_{102} & & a_{104} & & a_{106} & & a_{108} & & a_{1010}
\end{bmatrix},$$

$$L = \begin{bmatrix}
\ell_{11} & & & & & & & & & \\
\ell_{21} & \ell_{22} & & & & & & & & \\
\ell_{31} & \ell_{32} & \ell_{33} & & & & & & & \\
& & & \ell_{44} & & & & & & \\
& & & \ell_{54} & \ell_{55} & & & & & \\
\ell_{61} & \ell_{62} & \ell_{63} & & & \ell_{66} & & & & \\
& & & \ell_{74} & \ell_{75} & & \ell_{77} & & & \\
& & & & & & & \ell_{88} & & \\
& & & \ell_{94} & \ell_{95} & \ell_{96} & \ell_{97} & \ell_{98} & \ell_{99} & \\
\ell_{101} & \ell_{102} & \ell_{103} & \ell_{104} & \ell_{105} & \ell_{106} & \ell_{107} & \ell_{108} & \ell_{109} & \ell_{1010}
\end{bmatrix},$$

$$parent = \begin{bmatrix} 2 & 3 & 6 & 5 & 7 & 9 & 9 & 9 & 10 & 0 \end{bmatrix}.$$

In this example, the entries in L matrix written in bold denote the **fill-in** entries that are produced from the Cholesky Decomposition. For example notice the entry ℓ_{32}, because entries ℓ_{21} and ℓ_{31} exist then by rule 2 ℓ_{32} is a non-zero entry as well. Same thing goes for instance for entry ℓ_{95}, due to $\ell_{54} \neq 0 \wedge \ell_{94} \neq 0 \implies \ell_{95} \neq 0$, as well as for all other **fill-in** entries.

From rules 1 and 4 an algorithm that computes the Elimination Tree in nearly $O(|A|)$ time can be constructed. Assume that $\mathcal{T}_{k-1}$ is computed, which is a subset of $\mathcal{T}_k$, then to compute $\mathcal{T}_k$ the children of node k must be found, which are roots in $\mathcal{T}_{k-1}$. Since there holds $a_{ki} \neq 0$ for $i < k$ the path $i \rightsquigarrow k$ exist in $\mathcal{T}$ and it can be traversed up until reaching a root node in $\mathcal{T}_{k-1}$, and since the path leading to node k exist the root node reached is a child of node k.

The traversal to meet the upper time complexity needs a small modification. Let us introduce the concept of **ancestors**. The ancestor of i in the partially constructed tree $\mathcal{T}_{k-1}$ is ideally the *root* of the tree that contains i. Then traversing the path from i to the *root* would take constant time $O(1)$. Thus, the method used leads to a complexity that is $O(|A| \log n)$ but in practice the bound is hardly reached and the complexity is practically almost $O(|A|)$[23]. So the algorithm that computes the Elimination Tree is the one below:

```
function parent=etree(A)
    n=size(A)
    parent=zeros(n,1)
```

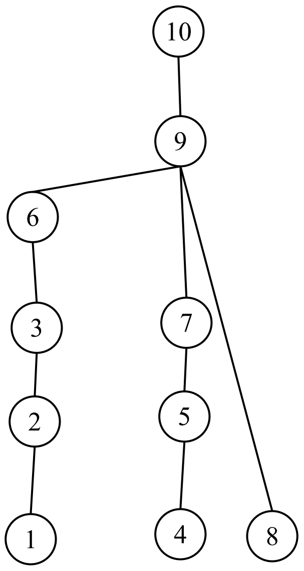

Figure 5.4: Elimination Tree of A (second example)

```
    ancestor=zeros(n,1)
    for k=1:n
        for p=Ap(k):Ap(k+1)-1
            i=Ai(p)
            while(i~=0 && i < k)
                inext=ancestor(i)
                ancestor(i)=k
                if inext==-1
                    parent(i)=k
                end
                i=inext
            end
        end
    end
end
```

5.5 Solving Sparse Lx=b using the Elimination Tree

Recall from Chapter 4.3.2 that for solving a Sparse $Lx = b$ system we need to compute the **Reach** of the lower triangular matrix in respect to the right hand side vector that is $\mathcal{X} = Reach_L(\mathcal{B})$. But in this specific case, the matrix L emerges from a **Choleksy Decomposition** and so it has a more particular pattern than the general case.

To compute the $\mathcal{L}_k = Reach_{k-1}(\mathcal{A}_k)$ where $k-1$ denotes the Graph of L_{k-1} and $\mathcal{A}_k$ denotes the *non-zero pattern* of column k of A which is the next candidate column for the next step in Cholesky Decomposition. The $Reach_L(i)$ is computed simply and fast by traversing the **Elimination Tree** from *node i* to the *root*, $i \rightsquigarrow r$. This needs less time than the general case Reach due to the graph being a **Tree** and not a **DAG**. Thus, to compute the Reach of $\mathcal{L}_k$ the k th *row sub-tree* needs to be traversed for the input $\mathcal{A}_k$, that is every non-zero entry in the kth column of A or the the kth row equivalently. The algorithm computing the **Elimination Tree Reach** is the following[23]:

```
function Lk=ereach(A,parent,k)
    n=size(A,1)
    Lk=zeros(n,1) % output array
    w=zeros(n,1) % work space array for marking values
    s=zeros(n,1) % stack array
    top=n
    mark(w,k) % mark node k as visited
    for p=Ap(k):Ap(k+1)-1
        i=Ai(p)
        if i>k
            continue
        end
        len=1
        while i=parent(i)
            if mark(w,i) % check if node i has already been visited
                break
            end
            s(len)=i
            len=len+1
            mark(w,i)
        end
        while len>0
            Lk(top)=s(len)
            top=top-1
            len=len-1
        end
    end
    Lk=Lk(Lk~=0) % remove if there are any zeros
    for p=1:length(Lk)-1
        mark(w,Lk(i)) % unmark node Lk(i)
    end
    mark(w,k)
end
```

The code takes as an input the matrix A, the step k and the parent array which is the result of the etree function. In the beginning, the variables are set up and the node k is marked as visited. Then, the path $i \rightsquigarrow r$ is traversed and every node in the path is marked as visited. Every node

encountered is placed in the *s* array and then is copied to Lk from the end to the front so as to preserve the *topological order*. Afterwards, the iteration continues with the next *i* node.
The function **mark** is taking as an input a work space array w and a node *i* and if the node is unmarked it marks it and vice versa. As an output it returns true if the node is already marked otherwise false. The implementation code for mark function is omitted.
The total time taken for **ereach** function, $\mathcal{L}_k = Reach$ is $O(\mathcal{L}_k)$ which is arguably faster than the **reach** function from Chapter 4.3.2.

5.6 Postordering the Elimination Tree

If a matrix A is permuted according to the postordering matrix P so as $C = PAP^T$ and $LL^T = C$, then L has the same non-zero elements as before but it would be in better structure resulting often to a faster decomposition. Additionally, the postorder of the Elimination Tree is essential for computing the non-zero entries of each column in L, which is of high importance as will be shown in the future chapters.

The filled graphs of A and $\mathrm{PAP^T}$ are isomorphic if P is a postordering of the elimination tree of A. Likewise, the elimination trees of A and $\mathrm{PAP^T}$ are isomorphic[54].

An isomorphism is a mapping between two structures of the same type that can be reversed by an inverse mapping. Two mathematical structures are isomorphic if an isomorphism exists between them[24].

In a postordering traversal the algorithm processes all nodes of a tree by recursively processing all sub-trees, then finally processing the root. So for the *child-parent* relation to be preserved the elimination tree, every child must have a smaller node number than its parent. For instance if a node *j* has *m* descendants the latter must be numbered from $j-m$ to $j-1$. For a postordering permutation array *post*, post(k)=j means that original node j is now numbered as k in the postordered tree. The relative ordering of the children of a node *j* is also preserved after a postordering. So if c_1<c_2<...<c_n are the n children of j then post(c_1)<post(c_2)<...<post(c_n) holds.

The recursive logic behind computing recursively the postordering using a depth-first search is shown below:

```
function P=postorder(T)
    k=0
    for j=1:length(T)
        dfstree(j)
    end
end


function dfstree(j)
    for i=1:n
        if i is a child of j
            dfstree(i)
            end
        post(k)=j
        k=k+1
        end
end
```

Once again, this recursive algorithm can cause stack overflow for very large inputs as mentioned in Chapter 4.3.2. So an iterative approach that mimics the recursive functions above is suggested.

An example of postordering is following:

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & & & a_{16} & & & & a_{110} \\
a_{21} & a_{22} & & & & a_{26} & & & & a_{210} \\
a_{31} & & a_{33} & & & & & & & \\
& & & a_{44} & a_{45} & & a_{47} & & a_{49} & a_{410} \\
& & & a_{54} & a_{55} & & a_{57} & & & \\
a_{61} & a_{62} & & & & a_{66} & & & a_{69} & a_{610} \\
& & & a_{74} & a_{75} & & a_{77} & & a_{79} & \\
& & & & & & & a_{88} & a_{89} & a_{810} \\
& & & a_{94} & & a_{96} & a_{97} & a_{98} & a_{99} & \\
a_{101} & a_{102} & & a_{104} & & a_{106} & & a_{108} & & a_{1010}
\end{bmatrix},$$

$$L_A = \begin{bmatrix}
\ell_{11} & & & & & & & & & \\
\ell_{21} & \ell_{22} & & & & & & & & \\
\ell_{31} & \ell_{32} & \ell_{33} & & & & & & & \\
& & & \ell_{44} & & & & & & \\
& & & \ell_{54} & \ell_{55} & & & & & \\
\ell_{61} & \ell_{62} & \ell_{63} & & & \ell_{66} & & & & \\
& & & \ell_{74} & \ell_{75} & & \ell_{77} & & & \\
& & & & & & & \ell_{88} & & \\
& & & \ell_{94} & \ell_{95} & \ell_{96} & \ell_{97} & \ell_{98} & \ell_{99} & \\
\ell_{101} & \ell_{102} & \ell_{103} & \ell_{104} & \ell_{105} & \ell_{106} & \ell_{107} & \ell_{108} & \ell_{109} & \ell_{1010}
\end{bmatrix},$$

$$C = \begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} & & & & & & c_{110} \\
c_{21} & c_{22} & & c_{24} & & & & & & c_{210} \\
c_{31} & & c_{33} & & & & & & & \\
c_{41} & c_{42} & & c_{44} & & & & & c_{49} & c_{410} \\
& & & & c_{55} & c_{56} & c_{57} & & c_{59} & c_{510} \\
& & & & c_{65} & c_{66} & c_{67} & & & \\
& & & & c_{75} & c_{76} & c_{77} & & c_{79} & \\
& & & & & & & c_{88} & c_{89} & c_{810} \\
& & & c_{94} & c_{95} & & c_{97} & c_{98} & c_{99} & \\
c_{101} & c_{102} & & c_{104} & c_{105} & & & c_{108} & & c_{1010}
\end{bmatrix},$$

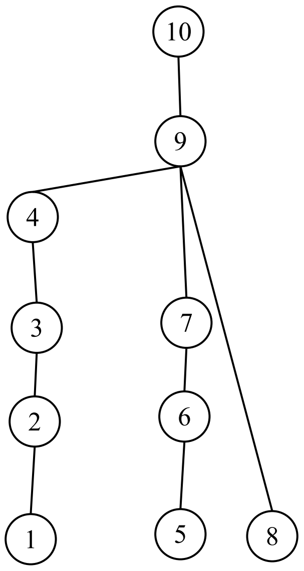

Figure 5.5: Post ordered Elimination Tree

$$L_C = \begin{bmatrix} \ell_{11} \\ \ell_{21} & \ell_{22} \\ \ell_{31} & \ell_{32} & \ell_{33} \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \\ & & & & \ell_{55} \\ & & & & \ell_{65} & \ell_{66} \\ & & & & \ell_{75} & \ell_{76} & \ell_{77} \\ & & & & & & & \ell_{88} \\ & & & \ell_{94} & \ell_{95} & \ell_{96} & \ell_{97} & \ell_{98} & \ell_{99} \\ \ell_{101} & \ell_{102} & \ell_{103} & \ell_{104} & \ell_{105} & \ell_{106} & \ell_{107} & \ell_{108} & \ell_{109} & \ell_{1010} \end{bmatrix},$$

$$post = \begin{bmatrix} 1 & 2 & 3 & 6 & 4 & 5 & 7 & 8 & 9 & 10 \end{bmatrix}.$$

The matrix A is the same as in the second example in Chapter 5.4.C is A(post,post) where post is shown below. So the corresponding elimination tree is shown in 5.4 and the postordered elimination tree is shown in Figure 5.5. The matrices L_C and L_A have the same number of non-zero values so obviously the have the same number of fill-in entries. But, it is easy to note that the elements in matrix C as well as in matrix L$_C$ are distributed in a better pattern.

5.7 Row Counts

The Row Count algorithm computes the number of non-zero entries in each row of the matrix L. The **Row Count** algorithm is a precursor to the **Column Count**. So many of the features of the latter algorithm are used in this one too. Since row counts is a simpler algorithm it is presented first although the in practice only the column counts algorithm is used[23].

To calculate the non-zeros of a row i a simple tree traversal method would suffice. For each $a_{ij} \neq 0$ start traversing the tree from node j up to node i or until finding a node which has been already visited. This is the same as traversing all the nodes in the subtree $\mathcal{T}^i$. The construction of an algorithm in the sense presented above would result in a time complexity of $O(|L|)$. Hopefully, an algorithm with better time complexity of nearly $O(|A|)$ time can be implemented, if some new concepts are introduced first. These concepts are the subsequent:

1. First Descendant
2. Skeleton Matrix
3. Least Common Ancestor
4. Path Decomposition

The key notion behind the algorithm is to decompose each row subtree in a set of paths which are disjoint. Each path will start at a leaf node, where $a_{ij} \neq 0$ in this subtree $\mathcal{T}^i$, and will end in least common ancestor of the current and the previous leaf nodes. Thus the length of each path is found by computing the difference between the level of the starting and ending node, where level denotes the distance of a node to the root of the tree. Now, let us take an better inspection at the rules mentioned before.

The **First Descendant** of a node j is the node with the smallest postorder value in the set of all the descendants of j. Considering the function introduced below the time needed to compute the first descendant is $O(n)$.

```
function [first,level]=first_descendant(n,post,parent,first,level)
    first=zeros(n,1)
    for k=1:n
        i=post(k)
        len=0
        r=i
        while((r!=0) && (first(r)==0)
            first(r)=k
            r=parent(r)
            len=len+1
        end
        if r==0
            len=len-1
        else
            len=len+level(r)
        end
        s=i
        while(s~=r)
            level(s)=len
            len=len-1
            s=parent(s)
        end

    end
end
```

The function starts at node k=1 in the postordered tree and goes all the way up to the root. All the nodes in the way gain 1 as their first descendant. For a node k>1 the algorithm terminates at a node *r* whose first descendant has already been found and so has it's level. The first in the path is once again the starting node *k*. Once the path has been determined, it is traversed again so as to set the level of the nodes along the way. An example is given in Figure 5.6

The next step is to divide the tree into disjoint paths. To do so, the leaf nodes must be found. The **Skeleton Matrix** is a structure created for that cause. The entries that are leaves in each row subtree form the **Skeleton Matrix** Â. Thus, all the entries in the matrix Â are a subset of the entries in the matrix A. The non-zero pattern of the Cholesky factorization for both Â and A matrices is the same.

Below a prototype of the skeleton function is presented, although the actual code is implemented in the row count function. Suppose the matrix is postordered and so the leaves of each row subtree can be determined by using the first descendant of each node in the following manner:

```
function skeleton(first ,n)
    maxfirst=zeros(n,1)
    for j=1:n
        for each i>j for which $a_{ij} \neq 0$
            if first(j)>maxfirst(i)
                j is a leaf in $\mathcal{T}^i$
                maxfirst(i)= first(j)
        end
    end
end
```

The algorithm starts considering a node j and all the row subtrees that contain this node. First(j) is the first descendant in the eliminatrion tree as seen before and maxfirst(i) is the biggest first(j) met so far in the $\mathcal{T}^i$. If first(j) $\leq$ maxfirst(i) then node *j* must have a descendant $d < j$ in $\mathcal{T}^i$, for which first(d)=maxfirst(i)$\geq$first(j). Thus, *j* is not a leaf of the subtree $\mathcal{T}^i$. On the other hand, if first(j)>maxfirst(i) then node j has no descendants in $\mathcal{T}^i$ and so *j* is a leaf. These rules are listed together below considering a postordered tree:

1. If f_j $\leq$j is the first descendant of *j* then all the descendants of *j* are f_j,$f_j + 1, \cdots ,j - 1$.

2. for two nodes $t < j$ then either $f_t \leq t < f_j \leq j$ or $f_j \leq f_t \leq t < j$. In the first case $t \notin descendants(j)$. In the second case $t \in descendants(j)$. As an example for the first case consider $t = 2$,$f_t = 1$ and $j = 6$,$f_j = 5$ and for the second case $t = 7$,$f_t = 5$ and $j = 9$,$f_j = 1$ in Figure 5.6.

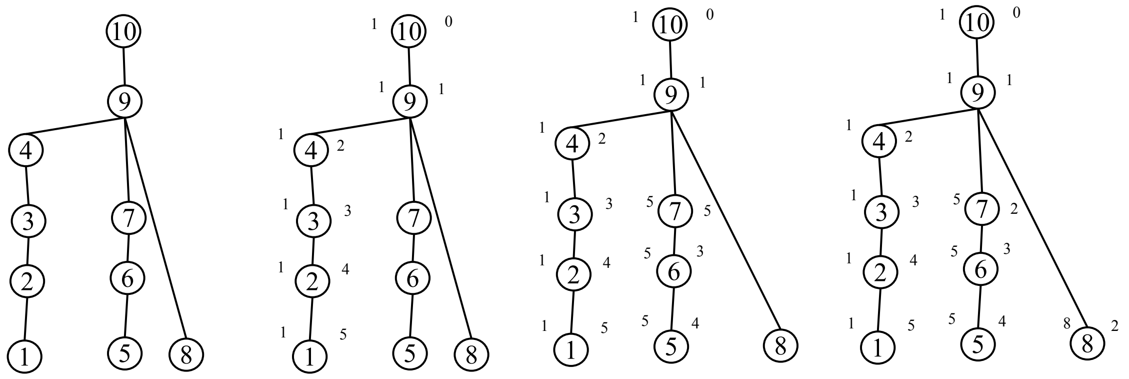
Figure 5.6: First Descendant Function Example (the number on the left is the f_j and on the right the level of each node)

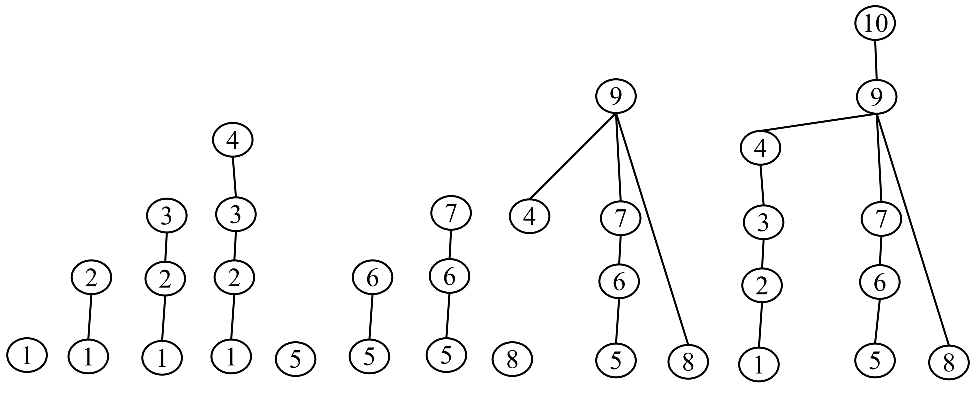
Figure 5.7: Row Subtrees of the postordered elimination tree

3. for a node j and a set S for which holds $\{\forall s \in S, s < j\}$. If node $t \in S$ has the largest first descendant f_t then j has a descendant in S if and only if $f_t \geq f_j$, $\{f_t \geq f_j \iff s \in descendants(j)\}$. Consider as $s = 8, f_s = 8$ and $j = 10, f_j = 1$ in Figure 5.6.

Consider the row subtrees presented in Figure 5.7 as an input to the skeleton function. The algorithm output is show below. The postordered matrix is shown again for convenience.

$$A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} & & & & & & a_{110} \\
a_{21} & a_{22} & & a_{24} & & & & & & a_{210} \\
a_{31} & & a_{33} & & & & & & & \\
a_{41} & a_{42} & & a_{44} & & & & & a_{49} & a_{410} \\
& & & & a_{55} & a_{56} & a_{57} & & a_{59} & a_{510} \\
& & & & a_{65} & a_{66} & a_{67} & & & \\
& & & & a_{75} & a_{76} & a_{77} & & a_{79} & \\
& & & & & & & a_{88} & a_{89} & a_{810} \\
& & & a_{94} & a_{95} & & a_{97} & a_{98} & a_{99} & \\
a_{101} & a_{102} & & a_{104} & a_{105} & & & a_{108} & & a_{1010}
\end{bmatrix}$$

Computation of Â using skeleton function:

```
j=1: first(1)=1, i=2 max(2)=0→1 leaf, i=3 max(3)=0→1 leaf,
     i=4 max(4)=0→1 leaf, i=10 max(10)=0→1 leaf

j=2: first(2)=1, i=4 max(4)=1 not leaf, i=10 max(10)=1 not leaf
j=3: first(3)=1
j=4: first(4)=1, i=9 max(9)=0→1 leaf, i=10 max(10)=1 not leaf
j=5: first(5)=5, i=6 max(6)=0→5 leaf, i=7 max(7)=0→5 leaf,
     i=9 max(9)=1→5 leaf, i=10 max(10)=10→5 leaf

j=6: first(6)=5, i=7 max(7)=5 not leaf,
j=7: first(7)=5, i=9 max(9)=5 not leaf
j=8: first(8)=8, i=9 max(9)=5→8 leaf, i=10 max(10)=5→8 leaf
j=9: first(9)=1, i=10 max(10)=8 not leaf
j=10: first(10)=1
```

$$\hat{A}= \begin{bmatrix} \hat{a}_{11} & \hat{a}_{12} & \hat{a}_{13} & \hat{a}_{14} & & & & & & \hat{a}_{110} \\ \hat{a}_{21} & \hat{a}_{22} & & & & & & & & \\ \hat{a}_{31} & & \hat{a}_{33} & & & & & & & \\ \hat{a}_{41} & & & \hat{a}_{44} & & & & & \hat{a}_{49} & \\ & & & & \hat{a}_{55} & \hat{a}_{56} & \hat{a}_{57} & & \hat{a}_{59} & \hat{a}_{510} \\ & & & & \hat{a}_{65} & \hat{a}_{66} & & & & \\ & & & & \hat{a}_{75} & & \hat{a}_{77} & & & \\ & & & & & & & \hat{a}_{88} & \hat{a}_{89} & \hat{a}_{810} \\ & & & \hat{a}_{94} & \hat{a}_{95} & & & \hat{a}_{98} & \hat{a}_{99} & \\ \hat{a}_{101} & & & & \hat{a}_{105} & & & \hat{a}_{108} & & \hat{a}_{1010} \end{bmatrix}$$

Recall the elements of the list introduced in the beginning of the chapter. The first two concepts are by now known. The **least common ancestor** of two nodes *i*,*j* is an ancestor of both *i* and *j* which has the smallest number and is denoted as q=lca(i,j). For instance, in the subtree $\mathcal{T}^9$ 9=lca(4,5) in Figure 5.7.

Now, **Path Decomposition** can be presented. Path Decomposition is a method which decomposes any row subtree into a set of disjoint paths. This can easily be be implemented by using the leaves and the theorems discussed previously. Consider two consecutive leaves $j_{prev} < j$. Each path starts at *j* and ends at q=lca(j_{prev},*j*). An example of the 3 distinct paths produced by this method for row subtree $\mathcal{T}^9$ which has 3 leaves is shown at Figure 5.8.

At this point computing the **Row Counts** is quite easy. That is because the non-zero entries of a particular row *i* can be computed by summing the length of all the distinct paths in $\mathcal{T}^i$. The length of each path can be found by calculating the level difference between the ending and the starting node.

The least common ancestor is efficiently computed by taking into account the following observation. Considering a postordered tree the least common ancestor of node *i* and *j* where $i < j$, can be found by traversing the path $i \rightsquigarrow root$. The first node q found along the way for which holds $q \geq j$ is the

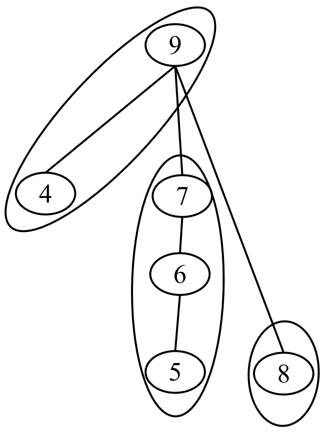

Figure 5.8: Path Decomposition of $\mathcal{T}^9$

least common ancestor.

The leaf function presented below determines if a node *j* is a leaf of $\mathcal{T}^i$ and if so it computes the least common ancestor between this and the previous leaf in this subtree. Again the ancestor concept is used as in **etree** function in Chapter 5.4. In the beginning each element is each own ancestor and the root of a set is always it's own ancestor[39].

```
function [q, maxfirst, prevleaf, ancestor, jleaf] =leaf(i, j, first,
maxfirst, prevleaf, ancestor, jleaf)
    jleaf=0
    if (i<=j || first(j)<=maxfirst(i))
        q=0 %not a leaf
        return
    end

    maxfirst(i)=first(j) % leaf→ update maxfirst
    jprev=prevleaf(i) % load prevleaf and upate the new
    prevleaf(i)=j
    if (jprev==-1) % first leaf
        q=i %i is the root of i subtree
        jleaf=1
        return
    end
    jleaf=2 % not first leaf
    q=jprev
    while (q~=ancestor(q)) % find the root of tree
    q=ancestor(q)
    end
    s=jprev
    while(s~=q)
```

```
            sparent=ancestor(s) % path compression with ancestor method
            ancestor(s)=q
            s=sparent
        end
end
```

The total **Row Count** algorithm then is[41]:

```
function rcount=rowcount(A, parent, post)
    n=A.n
    Ap=A.p
    Ai=A.i
    [first, level]= first_descendant(n, post, parent, first, level)
    for i=1:n
        rcount(i)=1 % for the diagonal
        prevleaf(i)=0
        maxfirst(i)=0
        ancestor(i)=i % every node is its own ancestor
    end
    for k=1:n
        j=post(k) % use postordering through permutation vector
        for p=Ap(k):Ap(k+1)-1
            i=Ai(p)
            [q, maxfirst, prevleaf, ancestor, jleaf] =leaf(i, j, first,
            maxfirst, prevleaf, ancestor, jleaf)
            if jleaf
                rcount(i)=rcount(i)+(level(j)-level(q))
            end
        end
        if (parent(j)~=--1)
            ancestor(j)=parent(j)
        end
    end
end
```

An example of the row count algorithm is presented below using as an input the postordered elimination tree which was used at the Skeleton matrix in 5.7. The max and first values are the same as well as the level.

```
j=1: i=2 prevleaf(2)=0→1 q=2 rcount(2)=1+(5−4)=2,
     i=3 prevleaf(3)=0→1 q=3 rcount(3)=1+(5−3)=3→,
     i=4 prevleaf(4)=0→1 q=4 rcount(4)=1+(5−2)=4,
     i=10 prevleaf(10)=0→1 q=10 rcount(10)=1+(5−0)=6,
     ancestor(1)=2
```

```
j=2: i=4 not leaf, i=10 not leaf, ancestor(1)=3
j=3: i=3 not leaf, ancestor(3)=4
j=4: i=9 prevleaf(9)=0→4 q=9 rcount(9)=1+(2−1)=2,
     i=10 not leaf, ancestor(4)=9
j=5: i=6 prevleaf(6)=0→5 q=6 rcount(6)=1+(4−3)=2,
     i=7 prevleaf(7)=0→5 rcount(7)=1+(4−2)=3,
     i=9 prevleaf(9)=4→5 q=9 rcount(9)=2+(4−1)=5,
     i=10 prevleaf(10)=1→5 q=9 rcount(10)=6+(4−1)=9, ancestor(5)=6
j=6: i=7 not leaf, ancestor(6)=7
j=7: i=9 not leaf, ancestor(7)=9
j=8: i=9 prevleaf(9)=5→8 q=9 rcount(9)=5+(2−1)=6,
     i=10 prevleaf(10)=1→8 q=9 rcount(10)=9+(2−1)=10
     ancestor(8)=9
j=9: i=10 not leaf ancestor(9)=10

rcounts=[1 2 3 4 1 2 3 1 6 10]
```

Additionally if a node k has ancestor(k)=j and then ancestor(j)=i when this path is traversed again the ancestor array will be updated and ancestor(k)=i, this is the path compression which is omitted in the example above.

5.8 Column Counts

Column counts can be computed in the same time as row counts, nearly $O(|A|)$. Let $\mathcal{A}_j$ be the non-zero pattern of j column of A matrix, $\hat{\mathcal{A}}_j$ be the non-zero pattern of j column of Â matrix, $\mathcal{L}_j$ be the non-zero pattern of column j of L matrix and finally c_j be the count of non-zero elements in $\mathcal{L}_j$, so $c_j = |\mathcal{L}_j|$.

If $\mathcal{L}_j$ denotes the non-zero pattern of the j^{th} column of L and $\mathcal{A}_j$ denotes the nonzero pattern of the strictly lower triangular part of the j^{th} column of A, then

$$\mathcal{L}_j = \mathcal{A}_j \cup j \cup (\bigcup_{s=child(j)} \mathcal{L}_s \setminus \{s\})[35].$$

In order to comprehend the previous relation consider the Figure 5.9. Suppose d is a descendant of j then the path $d \rightsquigarrow j$ exists in the elimination tree $\mathcal{T}$. For a node s that is a child of j the path $d \rightsquigarrow s$ exists also in $\mathcal{T}$. Consider a node $i > j$ if $\ell_{id} \neq 0 \implies d \rightsquigarrow s \rightarrow j \in \mathcal{T}^i \implies \ell_{is} \neq 0 \;\&\; \ell_{ij} \neq 0$. Thus the row index $i \in \mathcal{L}_s, \mathcal{L}_j$. So to construct $\mathcal{L}_j$ only $\mathcal{L}_s$, with s being a child of j, is needed of all the descendants of j. Finally, if $i \in \mathcal{L}_j \implies j \in \mathcal{T}^i$ so j is either a leaf or not. If j is a leaf then $i \in \hat{\mathcal{A}}_j \subseteq \mathcal{A}_j$. If not, then j has a child $s \in \mathcal{T}^i$ and so $i \in \mathcal{L}_s$.

So the nonzero pattern of the j^{th} column of L is a subset of the path $j \rightsquigarrow r$ from j to the root of the elimination tree $\mathcal{T}$.

Using only the upper relation the time needed for computing the column counts is $O(|L|)$. If the *least common ancestor* concept is taken into account then the time drops to $O(|A|)$.

If a node j is a leaf of the elimination tree then j will be a leaf for every entry i in column j that

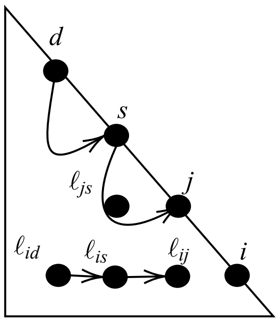

Figure 5.9: The $\mathcal{L}_j$ is the union of its children

$a_{ij} \neq 0$. So every such entry will be in the *skeleton matrix* $\hat{\mathcal{A}}$. The count of non-zero entries in $\mathcal{L}_j$ will be the number of $\hat{\mathcal{A}}_j = \mathcal{A}_j$ plus 1 for the diagonal entry ℓ_{jj}. So, $c_j = |\hat{\mathcal{A}}_j| + 1$, for j leaf of $\mathcal{T}$. On the other hand, if a node j is not a leaf of $\mathcal{T}$ then from 5.8 $\mathcal{L}_j = \mathcal{A}_j \cup j \cup (\bigcup_{s=child(j)} \mathcal{L}_s \setminus \{s\})$. But if an entry is in $\hat{\mathcal{A}}$ then it is not in the pattern of the children of j so $c_j = |\hat{\mathcal{A}}_j| + |\bigcup_{s=child(j)} \mathcal{L}_s \setminus \{s\}|$. If the number of children of j is computed in variable e_j then $c_j = |\hat{\mathcal{A}}_j| + |\bigcup_{s=child(j)} \mathcal{L}_s| - e_j$. The union of the children of j may have so overlapping nodes. If the overlap count was computed an stored in o_j then $c_j = |\hat{\mathcal{A}}_j| - o_j - e_j + \sum_{s=child(j)} c_s$.

For instance, to calculate the $\mathcal{L}_9$ of the tree in Figure 5.6 the children the relationship takes the following form: $\mathcal{L}_9 = \hat{\mathcal{A}}_9 \cup \mathcal{L}_4 \setminus \{4\} \cup \mathcal{L}_7 \setminus \{7\} \cup \mathcal{L}_8 \setminus \{8\} = \{0\} \cup \{9,10\} \cup \{9,10\} \cup \{9,10\} = \{9,10\}$. Also, $c_9 = \hat{\mathcal{A}}_9 - o_9 - e_9 + \sum_{s=child(9)} c_s = 0-4-3+9 = 2$. Node 9 has three children so $e_9 = 3$, nodes $\{9,10\}$ appear both in the three children so overlap $o_j = 4$ and finally $\sum_{s=child(9)} c_s = c_4 + c_7 + c_8 = 3+3+3 = 9$.

The method to find the Skeleton matrix is show in the previous section. Only the method to Figure out the overlaps is then needed, to be able to compute c_j from the relation shown above. The overlaps are closely related to the row subtrees of the elimination tree. If a node j is in the row subtree $\mathcal{T}^i$ that means $i \in \mathcal{L}_j$. So, there are three possible cases:

1. If $j \notin \mathcal{T}^i \iff i \notin \mathcal{L}_j$, so no contribution to o_j
2. If j is a leaf of $\mathcal{T}^i$ then $\hat{a}_{ij} \neq 0$ and since the skeleton matrix and the children set are disjoint, no contribution to o_j here either.
3. If $j \in \mathcal{T}^i$ and not a leaf then j has at least one child and their number is denoted by d_{ij}. Since row i belongs to every non-zero pattern of the children the overlap $o_j = d_{ij} - 1$.

The 3rd rule derives from the fact that if j has d_{ij} number of children in $\mathcal{T}^i$ then it will be the *least common ancestor* of $d_{ij} - 1$ pairs of leaves. Thereby, each time node j becomes a *lca* the overlap count o_j can be incremented by 1.

Suppose $\Delta_j = |\hat{\mathcal{A}}_j| - e_j - o_j$ for a non-leaf node. If j is a leaf then $\Delta_j = |\hat{\mathcal{A}}_j| + 1$. Then,

$$c_j = |\hat{\mathcal{A}}_j| - o_j - e_j + \sum_{s=child(j)} c_s \implies c_j = \Delta_j + \sum_{s=child(j)} c_s.$$

Δ_j is initialized as 1 if j is a leaf and as 0 if not. For every $\hat{a}_{ij} \neq 0$ Δ_j is incremented by 1 and decremented by 1 for every child of j as well as when j becomes the *lca* of two pair of leaves. The algorithm for computing the **Column Count** is given below[41][39].

```
function ccount=colcounts(A,parent ,post)
    n=A.n
    Ai=A.i
    Ap=A.p
    w=zeros(1,n)
    for k=1:n
        j=post(k)
        if (first(j)==0) % j is a leaf
            delta(j)=1
        else
            delta(j)=0 % j not a leaf
        end
        while ((j~=0) && (first(j)==0)) % construct the first of each node
            first(j)=k
            j=parent(j)
        end
        ancestor(k)=k % every node is each own ancestor at first
    end

    for k=1:n
        j=post(k)
        for p=Ap(j):Ap(j+1)-1
            i=Ai(p)
            [q, maxfirst, prevleaf, ancestor, jleaf]=leaf(i, j, first,
            maxfirst, prevleaf, ancestor, jleaf)
            if (jleaf>=1) delta(j)=delta(j)+1 % $a_{i,j} \in \hat{\mathcal{A}}$
            if (jleaf==2) delta(q)=delta(q)-1 % j is subsequent leaf
                                               % so delta(lca)--
        end
        if (parent(j)~=-0)
            ancestor(j)=parent(j) % every set belongs to its father
        end
    end
    ccount=delta
    for j=1:n
```

```
          if (parent(j)~=0)
               ccount(parent(j))=ccount(parent(j))+ccount(j)
               % the colcount of a node j is the sum of its children count
          end
     end
end
```

5.9 Symbolic Analysis

The **Symbolic Analysis** of a matrix is the computing the information that do not depend on each numerical values but mainly on the structure of the matrix, the non-zero pattern that is. This can be helpful because permutations that result in less fill-in entries can be found also because it is common practice for matrices of the same study area to have the same non-zero structure.

Everything computed so far in this chapter is part of the symbolic analysis of a sparse matrix. So, all of this information needs to be stored in a concise data structure for instance **A_symb**.

```
A_symb.parent=etree(A) % find the elimination tree
A_symb.post=post(A_symb.parent) % find the post order permutation
A_symb.cp=colcounts(A,A_symb.parent ,A_symb.post)
A_symb.cp=cumsum(c) % store the column pointers found using colcounts
A_symb.lnz=sum(A_symb.cp) % the number of nz entries in L
```

The entries in the struct A_symb can be extended by storing the permutation found suitable for this kind of matrix as well as the permuted matrix. For now, the elimination tree of A, the postordering of the tree, the column counts of L and the total number of non-zeros in L are computed.

5.10 Up-Looking Cholesky

At this moment, the final step is to compute the numerical values of the factorization using the *up-looking* method, it is the 2-by-2 decomposition introduced in chapter 5.2.

```
function L=up_chol(A,A_symb)
     n=A.n
     cp=A_symb.p
     Lp=c=cp
     Ap=A.p
     Ai=A.i
     parent=A_symb.parent
     for k=1:n
          % Find the non zero pattern of $A_{:k}$
          Lk=ereach(A,parent ,k) % find ereach
          x(k)=0
```

```
        for p=Ap(k):Ap(k+1)−1
            if Ai(i)<=k
                x(Ai(p)=Ax(p)
            end
        end
        d=x(k)
        x(k)=0
        % Triangular solve for $L_{k,:}$
        for Lk_c=1:length(Lk)
            i=Lk(Lk_c) % pattern of Lk
            lki=x(i)/Lx(Lp(i)) % L(k,i)=x(i)/L(i,i)
            x(i)=0
            for p=Lp(i)+1:c(i)−1
                x(Li(p)=x(Li(p))−Lx(p)∗lki
            end
            d=d−lki∗lki
            p=c(i)
            c(i)=c(i)+1
            Li(p)=k
            Lx(p)=lki
        end
        % Check if positive definite and find $L_{kk}$
        if (d<=0) %not positive definite
            L=0
            return
        end
        p=c(k)
        c(k)=c(k)+1
        Li(p)=k
        Lx(p)=sqrt(d)
    end
    Lp(n)=cp(n)
    L.p=Lp
    L.i=Li
    L.x=Lx
    L.n=Ln
end
```

The time taken to compute the Cholesky Decomposition using up_chol function is $O(f)$, the floating point operations performed, where $f = \sum |(L_{:k})|^2\dot{}$Factorization to LDL^T, with column counts complexity $O(|L|)$ is shown here[18].

5.11 Left-Looking Cholesky

The left-looking Cholesky is used more often than the up-looking one as it forms the basis of the **Supernodal** Method. For the left-looking method consider the following decomposition:

$$\begin{bmatrix} L_{11} & & \\ \ell_{12}^T & \ell_{22} & \\ L_{31} & \ell_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11} & \ell_{12} & L_{31}^T \\ & \ell_{22} & \ell_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{31}^T \\ a_{12}^T & a_{22} & a_{32}^T \\ A_{31} & a_{32} & A_{33} \end{bmatrix}.$$

For this decomposition the kth row and column are the middle ones. If the $k-1$ columns of L are known then $\ell_{12}^T\ell_{12} + \ell_{22}^2 = a_{22} \implies \ell_{22} = \sqrt{a_{22} - \ell_{12}^T\ell_{12}}$. Also, $L_{31}\ell_{12} + \ell_{32}\ell_{22} = a_{32} \implies$ $\ell_{32} = \dfrac{a_{32} - L_{31}\ell_{12}}{\ell_{22}}$.

A function prototype is given below for this decomposition:

```
function L=chol_left(A)
    n=size(A,1)
    L=zeros(n)
    for k=1:n
        L(k,k)=sqrt(A(k,k)-L(k,1:k-1)*L(k,1:k-1)')
        L(k+1:n,k)=(A(k+1:n,k)-L(k+1:n,1:k-1)*L(k,1:k-1)')/L(k,k)
    end
end
```

An example of how this algorithm proceeds is shown:

1st iter:$L_{11} = [], \ell_{12}^t = [], L_{31} = []$

$$\begin{bmatrix} \ell_{11} & & & \\ \ell_{12} & \ell_{22} & & \\ \ell_{31} & \ell_{32} & \ell_{33} & \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix} * \begin{bmatrix} \ell_{11} & \ell_{12} & \ell_{31} & \ell_{41} \\ & \ell_{22} & \ell_{32} & \ell_{42} \\ & & \ell_{33} & \ell_{43} \\ & & & \ell_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{31} & a_{32} & a_{33} & a_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \implies \ell_{11} = \sqrt{a_{11}}$$

$\ell_{12} = a_{12}/\ell_{11},\ \ell_{31} = a_{31}/\ell_{11},\ \ell_{41} = a_{41}/\ell_{11}$

2nd iter: $L_{11} = \ell_{11},\ \ell_{12}^T = \ell_{12},\ L_{31} = [\ell_{31}\ell_{41}]$

$$\begin{bmatrix} \ell_{11} & & & \\ \ell_{12} & \ell_{22} & & \\ \ell_{31} & \ell_{32} & \ell_{33} & \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix} * \begin{bmatrix} \ell_{11} & \ell_{12} & \ell_{31} & \ell_{41} \\ & \ell_{22} & \ell_{32} & \ell_{42} \\ & & \ell_{33} & \ell_{43} \\ & & & \ell_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{31} & a_{32} & a_{33} & a_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \implies \ell_{22} = \sqrt{a_{22} - \ell_{12} * \ell_{12}}$$

$\ell_{32} = \dfrac{a_{32} - \ell_{12}\ell_{31}}{\ell_{22}},\ \ell_{42} = \dfrac{a_{42} - \ell_{12}\ell_{41}}{\ell_{22}}$

3rd iter: $L_{11} = \begin{bmatrix} \ell_{11} & \\ \ell_{12} & \ell_{22} \end{bmatrix}, \ell_{12}^T = [\ell_{31}\ell_{32}],\ L_{31} = [\ell_{41}\ell_{42}]$

$$\begin{bmatrix} \ell_{11} & & & \\ \ell_{12} & \ell_{22} & & \\ \ell_{31} & \ell_{32} & \ell_{33} & \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix} * \begin{bmatrix} \ell_{11} & \ell_{12} & \ell_{31} & \ell_{41} \\ & \ell_{22} & \ell_{32} & \ell_{42} \\ & & \ell_{33} & \ell_{43} \\ & & & \ell_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{31} & a_{32} & a_{33} & a_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \implies \ell_{33} = \sqrt{a_{33} - \ell_{12}^T\ell_{12}} =$$

$\sqrt{a_{33} - \ell_{31}^2 - \ell_{32}^2},\ \ \ell_{43} = \dfrac{a_{43} - L_{31}\ell_{12}}{\ell_{33}} = \dfrac{a_{43} - \ell_{41}\ell_{31} - \ell_{42}\ell_{32}}{\ell_{33}}$

4th iter: $L_{11} = \begin{bmatrix} \ell_{11} & & \\ \ell_{12} & \ell_{22} & \\ \ell_{31} & \ell_{32} & \ell_{33} \end{bmatrix}, \ell_{12}^T = [\ell_{41}\ell_{42}\ell_{43}], L_{31} = []$

$$\begin{bmatrix} \ell_{11} & & & \\ \ell_{12} & \ell_{22} & & \\ \ell_{31} & \ell_{32} & \ell_{33} & \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix} * \begin{bmatrix} \ell_{11} & \ell_{12} & \ell_{31} & \ell_{41} \\ & \ell_{22} & \ell_{32} & \ell_{42} \\ & & \ell_{33} & \ell_{43} \\ & & & \ell_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{31} & a_{32} & a_{33} & a_{43} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \implies \ell_{44} = \sqrt{a_{44} - \ell_{12}^T \ell_{12}} = \sqrt{a_{44} - a_{41}^2 - a_{42}^2 - a_{43}^2}$$

$$L = \begin{bmatrix} \ell_{11} & & & \\ \ell_{12} & \ell_{22} & & \\ \ell_{31} & \ell_{32} & \ell_{33} & \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix}$$

In each iteration with bold are marked the already known elements. Also, in each iteration the same variables are used $L_{11}, \ell_{12}^T, L_{31}$, which correspond to the decomposition shown in the beginning of the chapter, for convenience.

5.12 Supernodal Cholesky

It is easy to recall that because of the properties of Cholesky factorization many columns end up having the same pattern as in matrix L below if the diagonal entry block.

$$L = \left[\begin{array}{c|ccc|c} L_{11} & & & & \\ & k_1 & & & \\ \hline & \bullet & L_{22} & & \\ L_{12}^T & \bullet & \bullet & & \\ & \bullet & \bullet & \bullet & k_2 \\ \hline & & L_{32} & & \\ & \bullet & \bullet & \bullet & \\ & \bullet & \bullet & \bullet & \\ L_{31} & & & & L_{33} \\ & \bullet & \bullet & \bullet & \end{array}\right]$$

The columns from k_1 through k_2 can be grouped together. The matrix L_{22} denotes the diagonal block and the matrix L_{32} the supernode consisting of the non-zero elements below the diagonal block.

For a set of nodes $k, k+1, .., k-1$ to form a supernode $\mathcal{S}$ the following must hold: for $i = 1, 2, q-1$, where $q = k_2 - k_1$ the node $k+i-1$ is the only child of node $k+i$ in the elimination tree, $children(k+i) = k+i-1$. These nodes can be grouped together into a single **supernode**[13][57].

Consider the computation of L_{*j} for some $j \geq k_2$. Suppose column A_{*j} has to be modified by L_{*i}, where $i \in \mathcal{S}$. It follows from the notion of supernodes that column A_{*j} will be modified by every $i \in \mathcal{S}$. Thus, a column $j \geq k_2$ is either modified by no column in $\mathcal{S}$ or by every column in it. So, a supernode can be treated as a single unit in the computations. Since, they have the same sparsity structure **dense** vector operations can be used and then applied to the target vector using a

single sparse vector operation that employs indirect addressing.

Consider in the decomposition mentioned in the previous Section 5.11. It can be modified to solve for the blocks introduced above and so:

$$\begin{bmatrix} L_{11} & & \\ L_{12}^T & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11} & L_{12} & L_{31}^T \\ & L_{22}^T & L_{32}^T \\ & & L_{33}^T \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{31}^T \\ A_{12}^T & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{bmatrix}.$$

For this decomposition the k^th row and column are the middle ones. If the $k-1$ columns of L are known then:

$L_{12}^T L_{12} + L_{22} L_{22}^T = A_{22} \implies L_{22} L_{22}^T = A_{22} - L_{12}^T L_{12}$.

$L_{31} L_{12} + L_{32} L_{22}^T = A_{32} \implies L_{32} L_{22}^T = A_{32} - L_{31} L_{12} \implies L_{22} L_{32}^T = A_{32}^T - L_{12}^T L_{31}^T$.

The first equation results into a **dense** Cholesky factorization which needs no exploitation of sparsity at all. The second equation is a triangular solve. L_{32}^T transpose is a **dense** matrix, in this particular example a 3x3 dense matrix. In general it is an orthogonal dense matrix of dimension $(|nz|x(k_2 - k_1))$ where $|nz|$ is the number of non-zeros in the supernode.

The algorithm for computing the supernodal cholesky follows:

```
function L=chol_super(A,s)
    n=size(A)
    L=zeros(n)
    ss=cumsum([1 s])
    for j=1:length(s)
        k1=ss(j)
        k2=ss(j+1)
        k=k1:(k2-1)
        L(k,k)=chol(A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)')'
        L(k2:n,k)=(A(k2:n,k)-L(k2:n,1:k1-1)*L(k,k1:k1-1)')/L(k,k)'
    end
end
```

In the above code s is an integer vector of where $s(j) > 0 \; \forall j$ and $sum(s) = n$. The j^th supernode consists of s(j) columns of L which can be stored as a dense matrix.

So all the key operations of a supernodal Cholesky are:

1. A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)'). A(k,k) is dense. L(k,1:k1-1) are the rows in a subset of the descendants of j^th supernode. The update from each descendant can be done with a single dense matrix multiplication.

2. L(k,k)=chol(A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)')'. A dense Cholesky factorization

3. A sparse matrix product (A(k2:n,k)-L(k2:n,1:k1-1)*L(k,k1:k1-1)'), where the L terms come from the descendants of the j^th supernode.

4. A dense triangular solve L(k2:n,k)=(A(k2:n,k)-L(k2:n,1:k1-1)*L(k,k1:k1-1)')/L(k,k)'.

Supernodal method is among the ones which can exploit **dense matrix kernels**. So, it can achieve a percentage of computers theoretical peak performance on modern computers.[13]

5.13 Multifrontal Cholesky

The Multifrontal Cholesky is based on the Right-Looking Cholesky Decomposition which is presented below:

$$\begin{bmatrix} \ell_{11} & \\ \ell_{21} & L_{22} \end{bmatrix} \begin{bmatrix} \ell_{11} & \ell_{21}^T \\ & L_{22}^T \end{bmatrix} = \begin{bmatrix} a_{11} & a_{21}^T \\ a_{21} & A_{22} \end{bmatrix},$$

where ℓ_{11}, a_{11} are scalars ℓ_{21}, a_{21} are vectors and L_{22}, A_{22} are submatrices. Expanding the relations:

$$\ell_{11}^2 = a_{11} \implies \ell_{11} = \sqrt{a_{11}} \qquad \ell_{21}\ell_{11} = a_{21} \implies \ell_{21} = \frac{a_{21}}{\ell_{11}}$$
$$L_{22}L_{22}^T + \ell_{21}\ell_{21}^T = A_{22} \implies L_{22}L_{22}^T = A_{22} - \ell_{21}\ell_{21}^T.$$

The **Multifrontal** method organizes the operations that take place during the factorization of sparse matrices in such a way that the entire factorization is performed through partial factorizations of a sequence of dense and small submatrices [2]. This method was first developed for solving problems arising from finite-element analysis and then was generalized for solving sparse matrix systems[30]. This method is based on the *Elimination Tree* examined before and two new concepts **frontal** and **update** matrices. Consider a matrix A with its Cholesky Factor L and the nonzero pattern of $\mathcal{L}_{*j} = \{i_0, i_1, \dots, i_r\}$, where $i_0 = j$, the j^{th} column has r off-diagonal entries and $\mathcal{T}[k]$ symbolized the sub-elimination tree with root k.

The **subtree update** matrix is given by $\overline{U} = -\sum_{k\in\mathcal{T}[j]-\{j\}} \begin{bmatrix} \ell_{j,k} \\ \ell_{i_1,k} \\ \vdots \\ \ell_{i_r,k} \end{bmatrix} \begin{bmatrix} \ell_{j,k} & \ell_{i_1,k} & \dots & \ell_{i_r,k} \end{bmatrix}.$

These are the outer-products updates of all the descendants of j. Thus, the **frontal** matrix F_j for A is defined as:

$$F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} \dots a_{j,i_r} \\ a_{i_1,j} & \\ \vdots & \\ a_{i_r,j} & \end{bmatrix} + \overline{U}$$

The nonzeros of the j$^{\text{th}}$ column of L are r+1 and so is the order of the matrices F and $\overline{U}$. The +1 because the diagonal is counted. This relation resembles the last update of the right-looking algorithm, only in this scenario the node j is updated only by the contribution of its descendants.

The subtree update matrix $\overline{U}$ can be split into two sums. The first containing all the descendants k for whom $\ell_{jk} \neq 0, k < j$ and the other all the descendants of j for whom $\ell_{jk} = 0$.

Thus, $\overline{U} = -\sum_{\substack{k\in\mathcal{T}[j]-j \\ \ell_{jk}\neq 0 \\ k<j}} \begin{bmatrix} \ell_{j,k} \\ \ell_{i_1,k} \\ \vdots \\ \ell_{i_r,k} \end{bmatrix} \begin{bmatrix} \ell_{j,k} & \ell_{i_1,k} & \dots & \ell_{i_r,k} \end{bmatrix} - \sum_{\substack{k\in\mathcal{T}[j]-j \\ \ell_{jk}=0}} \begin{bmatrix} 0 \\ \ell_{i_1,k} \\ \vdots \\ \ell_{i_r,k} \end{bmatrix} \begin{bmatrix} 0 & \ell_{i_1,k} & \dots & \ell_{i_r,k} \end{bmatrix}.$

The first component $-\sum_{\substack{k\in\mathcal{T}[j]-j \\ \ell_{jk}\neq 0 \\ k<j}} \ell_{jk} \begin{bmatrix} \ell_{j,k} \\ \ell_{i_1,k} \\ \vdots \\ \ell_{i_r,k} \end{bmatrix}$ is named as **complete update column** to column j and

contains all the nonzero updates to j[6]. Thus, F_j is obtained by the nonzero pattern $\mathcal{A}_j$ and the complete update column to j. Hence, when F_j is computed the first row/column is already completely updated.

Performing an elimination of the first column of the frontal matrix results in the nonzero pattern of L_{*j} and gives the **update** matrix from column j as well. That is like the $\ell_{21}\ell_{21}^T$ term which is propagated to the rest (n-1)-by-(n-1) matrix but only the part that j contributes to the update of the rest of the matrix. Consider F_j expressed as follows:

$F_j = \begin{bmatrix} \ell_{j,j} \\ \ell_{i_1,j} \\ \vdots \\ \ell_{i_r,j} \end{bmatrix} \begin{bmatrix} \ell_{j,j} & \ell_{i_1,j} & \dots & \ell_{i_r,j} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & U_j \end{bmatrix}$, where the first term is the completed j row and column, and U$_j$ is the update matrix from j. If from the term $\overline{U}$ the first row and column which are removed, the submatrix left is the same as U_j. Therefore, the following relation arises:

$$- \sum_{\substack{k\in\mathcal{T}[j]-j \\ \ell_j k=0}} \begin{bmatrix} \ell_{i_1,k} \\ \vdots \\ \ell_{i_r,k} \end{bmatrix} \begin{bmatrix} \ell_{i_1,k} & \dots & \ell_{i_r,k} \end{bmatrix} = \begin{bmatrix} \ell_{i_1,j} \\ \vdots \\ \ell_{i_r,j} \end{bmatrix} \begin{bmatrix} \ell_{i_1,j} & \dots & \ell_{i_r,j} \end{bmatrix} + U_j.[55].$$

Note that because F_j has as first row and column the pattern L_{*j} is a full matrix of dimension $|L_j|$-by-$|L_j|$. Then, the submatrix after the elimination U_j must be also full. Thus it can exploit the dense matrix BLAS and obtain full performance[62]. The U_j block or contribution block to its ascendants frontal matrices is the Schur compliment of its pivot row and column[12]

Another helpful notion introduce to help with the formation of frontal matrices is the **extend-add operator**:$\oplus$. This can help adding two matrices of different size, using extended algebra[9].

Consider, a r-by-r R submatrix and a s-by-s S submatrix with $r, n \leq n$. Each row and column of these matrices correspond to a row and column in the original n-by-n A matrix. Assume that the subscripts of R in A are denoted by $i_1 \leq \dots \leq i_r$ and of S by $j_1 \leq \dots \leq j_n$ as well. Consider now the union of the two sets and $k_1 \leq \dots \leq k_t$ be the corresponding subscripts. Then, the two matrices R,S can be extended by adding zero rows and column in order to match the desired $\{k_1, \dots, k_t\}$ indexing thus making the addition between them possible. Therefore, $R \oplus S$ is the resulting t-by-t T matrix formed by the extended R and S matrices. This process is referred to as the generalized matrix **superposition**[55][63].

An example follows of this operations. Consider $R = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, $S = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$, where $\{3,4\}$ and $\{3,6\}$ are the subscripts of R and S respectively. $R \oplus S = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} e & 0 & f \\ 0 & 0 & 0 \\ g & 0 & h \end{bmatrix} =$

$\begin{bmatrix} a+e & b & f \\ c & d & 0 \\ g & 0 & h \end{bmatrix} = T$, where T is 3-by-3 matrix with subscripts the union $R \cup S = \{3,4,6\}$.

Thus, the previous ecome using the extended addition operator:

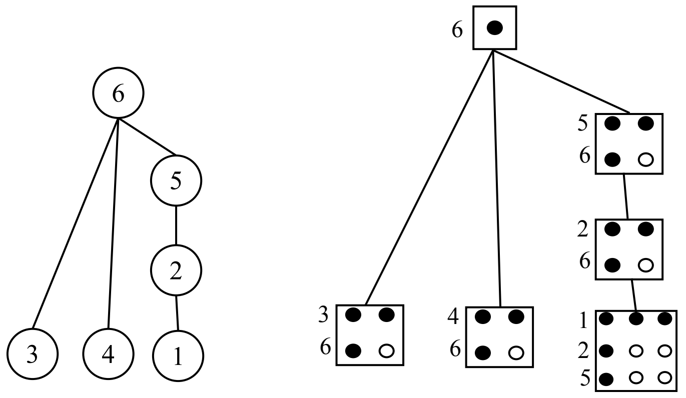

Figure 5.10: Multifrontal Elimination Tree and Frontal Matrices

$$F_j = \begin{bmatrix} a_{j,j} & a_{j,i_1} \dots a_{j,i_r} \\ a_{i_1,j} & \\ \vdots & \\ a_{i_r,j} & \end{bmatrix} \oplus U_{c_1} \oplus \dots \oplus U_{c_s}, \text{ where } c_1, \dots, c_n \text{ are the children of node j.}$$

An example of the multifrontal method is shown below. Let $A = \begin{bmatrix} a_{11} & a_{12} & & & a_{15} & \\ a_{21} & a_{22} & & & a_{25} & \\ & & a_{33} & & & a_{36} \\ & & & a_{44} & & a_{46} \\ a_{51} & a_{52} & & & a_{55} & a_{56} \\ & & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$

and its Cholesky factor $L = \begin{bmatrix} \ell_{11} & & & & & \\ \ell_{21} & \ell_{22} & & & & \\ & & \ell_{33} & & & \\ & & & \ell_{44} & & \\ \ell_{51} & \ell_{52} & & & \ell_{55} & \\ & & \ell_{63} & \ell_{64} & \ell_{65} & \ell_{66} \end{bmatrix}$. The corresponding elimination tree and the frontal matrices relation is shown in 5.10.

$\overline{U}_3 = 0, \overline{U}_4 = 0, \overline{U}_1 = 0$ since they are leaves of the tree and have no children.

$F_3 = \begin{bmatrix} a_{33} & a_{36} \\ a_{63} & \end{bmatrix}, U_3 = -\begin{bmatrix} \ell_{63}^2 \end{bmatrix}$

$F_4 = \begin{bmatrix} a_{44} & a_{46} \\ a_{64} & \end{bmatrix}, U_4 = -\begin{bmatrix} \ell_{64}^2 \end{bmatrix}$

$F_1 = \begin{bmatrix} a_{11} & a_{12} & a_{15} \\ a_{21} & & \\ a_{51} & & \end{bmatrix}, U_1 = -\begin{bmatrix} \ell_{21}^2 & \ell_{21}\ell_{51} \\ \ell_{21}\ell_{51} & \ell_{51}^2 \end{bmatrix}$

$F_2 = \begin{bmatrix} a_{22} & a_{25} \\ a_{25} & \end{bmatrix} \oplus U_1 = \begin{bmatrix} a_{22} & a_{25} \\ a_{25} & \end{bmatrix} - \begin{bmatrix} \ell_{21}^2 & \ell_{21}\ell_{51} \\ \ell_{21}\ell_{51} & \ell_{51}^2 \end{bmatrix} = \begin{bmatrix} a_{22} - \ell_{21}^2 & a_{25} - \ell_{21}\ell_{51} \\ a_{25} - \ell_{21}\ell_{51} & -\ell_{51}^2 \end{bmatrix},$

$U_2 = -\begin{bmatrix} \ell_{51}^2 + \ell_{52}^2 \end{bmatrix}$

$$
\begin{aligned}
F_5 &= \begin{bmatrix} a_{55} & a_{56} \\ a_{65} & \end{bmatrix} \oplus U_2 = \begin{bmatrix} a_{55} & a_{56} \\ a_{56} & \end{bmatrix} - \begin{bmatrix} \ell_{51}^2 + \ell_{52}^2 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} a_{55} - \ell_{51}^2 - \ell_{52}^2 & a_{56} \\ a_{56} & 0 \end{bmatrix} \\
U_5 &= -\begin{bmatrix} \ell_{65}^2 \end{bmatrix} \\
F_6 &= \begin{bmatrix} a_{66} \end{bmatrix} \oplus U_5 \oplus U_4 \oplus U_3 = \begin{bmatrix} a_{66} \end{bmatrix} - \begin{bmatrix} \ell_{65}^2 \end{bmatrix} - \begin{bmatrix} \ell_{64}^2 \end{bmatrix} - \begin{bmatrix} \ell_{63}^2 \end{bmatrix} = \begin{bmatrix} a_{66} - \ell_{65}^2 - \ell_{64}^2 - l_{63}^2 \end{bmatrix}
\end{aligned}
$$

Finally, it is instructive to observe that except from the extended-add operation, the rest of the computations are carried out in full matrix form, which has advantages of course over sparse computations. It is only natural, that this method can yet be improved.

In each frontal matrix there is only one variable fully assembled and may result in small fronts to exploit memory hierarchy available in new hardware as well as all the BLAS subroutines to their fullest[2]. Thus, it is common practice to form supernodes of nodes with same nonzero structure and creating an **amalgamated elimination tree**[5].

Chapter 6

QR Decomposition

6.1 Introduction

QR Decomposition or **QR Factorization** is a method of decomposing a matrix A into an **orthogonal** Q and a matrix R in **upper triangular shape** or **right triangular**, so that **A=QR**.

In contrast to *Cholesky Method* mentioned in Chapter 5, matrix A does not need to be **symmetric, positive-definite**, it does not need to be **square** as well.

6.2 Method Overview

This method results in two components, matrices Q and R. R is in an upper triangular shape meaning R=$\begin{bmatrix} U \\ 0 \end{bmatrix}$, where U is upper triangular, which has the benefits of solving a system with just back substitution. Q is an orthogonal matrix, meaning its columns are **orthonormal** or **orthogonal unit vectors**. These kind of matrices have an interesting set of properties[10].

1. Q^{T}Q=QQ^T=I
2. Q^T=Q^{-1}
3. Let a vector x and $||x||_2 = \sqrt{x^Tx}$ its second norm. For a Q matrix $||Qx||_2 = \sqrt{(Qx)^TQx} = \sqrt{x^TQ^TQx} = \sqrt{x^TIx} = \sqrt{x^Tx} = ||x||_2$.

The third property is very useful because it *preserves* the **second norm**. That means that the multiplication with a an orthogonal matrix just *rotates* a vector in space *without affecting* its *length*.

Lets consider an **overdetermined system**, a system with more equations than unknowns, Ax=b, $A \in \mathbb{R}^{m\ x\ n}$, $x \in \mathbb{R}^{n\ x\ 1}$, $b \in \mathbb{R}^{m\ x\ 1}$. Very often these systems are **inconsistent**, meaning they have no exact solution.

To find an approximate solution to this problem the method of **Ordinary Least Squares(LST)** is usually used. Consider the **residual vector** r=Ax-b, minimizing r will result in an approximate solution for the inconsistent system. The **LST** method minimizes the sum of the squares or the root of the sum of the squares as well, which is the second norm.Thus, $||r||_2 = ||Ax - b||_2$.

Consider a QR Decomposition of A, QR=A where $A \in \mathbb{R}^{m\ x\ n}$, $Q \in \mathbb{R}^{m\ x\ m}$, $R \in \mathbb{R}^{m\ x\ n}$, $m > n$. $QR = A \iff Q^{-1}QR = Q^{-1}A \iff R = Q^{-1}A \iff R = Q^TA$
Substituting this in the **LST** residual results in: $||r||_2 = ||Q^Tr||_2 = ||Q^TAx - Q^Tb||_2 = ||Rx - Q^Tb||_2 = ||Rx - b'||_2$.
If R is split as R=$\begin{pmatrix} R_1 \\ 0 \end{pmatrix}$, $R_1 \in \mathbb{R}^{n\ x\ n}$, then $\begin{pmatrix} R_1 \\ 0 \end{pmatrix} x = \begin{pmatrix} b'_1 \\ b'_2 \end{pmatrix} = \begin{pmatrix} R_1x - b'_1 \\ -b_2 \end{pmatrix}$ and thus the residual is $||Rx - b'||_2^2 = ||R_1x - b'_1||_2^2 + ||b'_2||_2^2$.

In the equation above the only variable which is free is **x**, the others are fixed. Thus, an to minimize the expression an **x** should be chosen which sets $||R_1x - b'_1||_2 = 0 \iff R_1x = b'_1$. This is an upper triangular solve which results in the minimum residual possible.

There are various ways to compute the QR Decomposition such as **Householder Reflections, Gram-Schmidt process, Givens rotations**, each one has its own advantages and disadvantages. In this project the **Householder Reflections** method is going to be used.

6.3 Householder Reflections

Let H be an orthogonal matrix whose first row is $x/||x||_2$, so $H = \begin{bmatrix} x^T/||x||_2 \\ H_1 \end{bmatrix}$.

Then $Hx_1 = \begin{bmatrix} \frac{||x||_2^2}{||x||_2} \\ 0 \end{bmatrix} = \begin{bmatrix} ||x||_2 \\ 0 \end{bmatrix}$.

Let a matrix $A = \begin{pmatrix} x & A_2 \end{pmatrix}$. $HA = \begin{pmatrix} Hx & HA_2 \end{pmatrix} = \begin{bmatrix} \rho_{11} & r_{12}^T \\ 0 & A_{22} \end{bmatrix}$

If this process is recursively applied to A_{22} the result then is an upper triangular matrix R. Thus, $HA = H_mH_{m-1}\dots H_1A = \begin{pmatrix} R \\ 0 \end{pmatrix}$. The essential feature of this algorithm is that $Hx = ce_1$.

A **Householder Transformation** or **Reflection** is a matrix of the form:
$H = I - \beta uu^T$, where $\beta||u||_2^2 = 2$. $u,\ \beta$ are chosen so as the multiple of Hx_1 is one described above. In space notion, u is the vector perpendicular to the mirror that transforms the the vector from n dimensions to one.
It follows that the second Householder reflection should be chosen so as to have the same effect on the first column of XA_{22} while not affecting the column already modified by the previous reflection. A Householder operation takes about $\mathcal{O}(4n)$ because $Hx = (I - \beta uu^T) = x - u(\beta(u^Tx))$[48].

The non-zero pattern of the vector **v** is symbolized by the notion $\mathcal{V}$ and $\mathcal{X}$ is the non-zero pattern of the **x** vector on which the reflection is applied, evidently $\mathcal{V} = \mathcal{X}$. The hypothesis that the matrix A has a zero free diagonal (non-singular matrix) is made for convenience of calculations.

Each Householder Reflection is multiplied by the whole matrix. So lets consider another vector **y** with non-zero pattern $\mathcal{Y}$. $Hy = (I - \beta uu^T)y = y - \beta(u(u^Ty))$. The dot product u^Ty will be non-zero only if **y** has at least one non-zero value in the same place as **u**, which has the same pattern as **x**. So, if the two columns $\mathbf{x}, \mathbf{y}$ have non-zero values in the same row of the matrix. If the two columns are disjoint then the transformation has zero effect on the column **y** and $\mathcal{Y}' = \mathcal{Y}$. On the other hand, if they intersect $\mathcal{Y}' = \mathcal{Y} \cup (\mathcal{Y} \cap \mathcal{X}) = \mathcal{Y} \cup \mathcal{X}$[23].

For instance consider the following matrix: $A = \begin{bmatrix} x_1 & 0 & 0 & 0 \\ 0 & y_2 & 0 & 0 \\ x_3 & 0 & z_3 & 0 \\ 0 & y_4 & 0 & w_4 \end{bmatrix}$. If the H which reflects **x** is applied the non-zero pattern of **y**, **w** will remain the same because $u^T y = u^T w = 0$. On the contrary, $u^T z = u_3 * z_3 = c$ and so, $Hz = z - \beta * c * u = z + c' * u$ with non-zero pattern $\mathcal{Z} = \mathcal{Z} \cup \mathcal{X}$.

So, $HA = \begin{bmatrix} x_1' & 0 & z_1' & 0 \\ 0 & y_2 & 0 & 0 \\ 0 & 0 & z_3' & 0 \\ 0 & y_4 & 0 & w_4 \end{bmatrix}$.

In general, if a matrix is defined as follows:

$$A = \left[\begin{array}{c|c|c} \mathbf{x} & share & no\ share \\ \hline x_1 & & 0 \\ \vdots & y & \vdots \\ x_k & & 0 \\ \hline 0 & & z \\ \vdots & y_l & \end{array}\right]$$

, where **x** is the vector the transformation will annihilate and it is formatted with its non-zero elements from top until x_k and the rest are zero, the *share column* contains the columns which share a non-zero pattern with **x** , the *no share column* contains the columns which do not share a non-zero pattern with **x**. The sets *share, no share* is assumed that they exist.

After the Householder is applied the bottom half of the split will remain unchanged as well as the no share column. On the other hand, the columns contained in the second set because they have at least one match with **x**, they will all inherit its pattern, $\mathcal{Y}' = \mathcal{Y} \cup \mathcal{X}$. So

$$HA = \left[\begin{array}{c|c|c} \mathbf{x} & share & no\ share \\ \hline x_1 & y_1' & 0 \\ \vdots & \vdots & \vdots \\ x_k & y_k' & 0 \\ \hline 0 & & z \\ \vdots & y_l & \end{array}\right].$$

So there is a very interesting pattern emerging from the Householder Reflections, which will be investigated in forward chapters because it is of significant importance for the Symbolic Analysis of QR Factorization.

6.4 Left/Right-Looking Dense QR Decomposition

Consider a matrix $A \in \mathbb{R}^{m\ x\ n}, m > n$, the Decomposition can still be applied if $m = n$ or $m < n$. Let $A^{(1)} = A$ and $A^{(k+1)} = H^{(k)} A^{(k)},\ k = 1 \ldots n$. $H^{(1)}$ is constructed from the first column of A and has length m and is $H^{(1)} = I - \beta_1 u_1 u_1^T$. $H^{(k)}$ is constructed from the kth column of the modified A' matrix and $x^{(k)} = A^{(k-1)}_{k \cdots m, k}$ which has length m-k+1 and is $H^{(k)} = I - \beta_k u_k u_k^T$.

The QR Factorization of A is $R = H_n H_{n-1} \ldots H_1 A = H_1 \iff H_1 H_2 \ldots H_n R = A$, so $Q = H_1 H_2 \ldots H_n$ and $QR = A$. That holds because the Householder matrix is obviously symmetric,

$H^T = (I - \beta uu^T)^T = I^T - (\beta uu^T)^T = I - \beta uu^T = H$. Bear in mind that Q does not need to be explicitly computed[44].

Recall from the previous section that each next reflection should leave the previous unchanged. For instance, after the first Householder suppose that the entry a_{12} exists. Then, an arbitrary chosen Householder Reflection for the second column will change the first column again, due to the properties of the reflection described. So, to leave the first column unmodified the second Reflection should be in the form: $\begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix}$, which if right-multiplied by $= \begin{bmatrix} \rho_{11} & r_{12}^T \\ 0 & A_{22} \end{bmatrix}$ result in:
$\begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix} * \begin{bmatrix} \rho_{11} & r_{12}^T \\ 0 & A_{22} \end{bmatrix} = \begin{bmatrix} \rho_{11} & r_{12}^T \\ 0 & H_2 A_{22} \end{bmatrix}$.

The block matrix is orthogonal: $\begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix} * \begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix}^T = \begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix} * \begin{bmatrix} I & 0 \\ 0 & H_2^T \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & H_2 H_2^T \end{bmatrix} =$
$\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} = I$.

Of course, it is symmetric as well: $\begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix}^T = \begin{bmatrix} I & 0 \\ 0 & H_2^T \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & H_2 \end{bmatrix}$.

The right-looking Algorithm **qr_right** applies the Householder Reflections to the matrix A as soon as it is constructed.

```
function [V,beta,R]=qr_right(A)
    [m n]=size(A);
    V=zeros(m,n);
    Beta=zeros(1,n);
    for k=1:n
        [v,beta,s]=gallery('house',A(k:m,k),2);
        V(k:m,k)=v;
        Beta(k)=beta;
        A(k:m,k:n)=A(k:m,k:n)-v*(beta*(v'*A(k:m,k:n)));
    end
    R=A;
end
```

In the code above, V is a data structure where every column is a Householder Transformation. Beta is the collection of all scalar β values as well. Gallery function computes the Householder transformation of the k^{th} column, from diagonal and below, since the elements above the diagonal should remain intact. The final line is the expansion of $HA = (I - \beta u * u^T)A = A - \beta u(u^T A)$. Note that in the latter form the computations are $O(n^2)$ while in the former is $O(n^3)$. The total work of the Algorithm is $O(n^3)$.

The left-looking algorithm **qr_left** postpones the application of Householder Reflections to the whole matrix and only applies them to the current column.
Consider a matrix $A = \left(\mathbf{a}_1 \,\middle|\, \mathbf{a}_2 \,\middle|\, \dots \,\middle|\, \mathbf{a_n} \right)$ and H_1 the Reflection annihilating the first column. In order to compute the second Householder Reflection only $H_2\mathbf{a}_2$ is needed, the rest computations

$H_1 \left(\mathbf{a}_3 \middle| \cdots \middle| \mathbf{a_n} \right)$ can be postponed until it is time for their own Reflection to be computed. Thus, to compute H_3, $H_1 H_2 \mathbf{a}_3$ is needed and so forth.

```
function [V,beta ,R]=qr_left(A)
    [m n]=size(A);
    V=zeros(m,n);
    Beta=zeros(1,n);
    R=zeros(m,n);
    for k=1:n
        x=A(:,k);
        for i=1:k−1
            v=V(i:m,i);
            beta=Beta(i);
            x(i:m)=x(i:m)−v∗(beta∗(v'∗x(i:m)));
        end
        [v,beta ,s]=gallery('house',x(k:m),2);
        V(k:m,k)=v;
        Beta(k)=beta;
        R(1:(k−1),k)=x(1:(k−1));
        R(k,k)=s
    end
end
```

In the code above, V and Beta serve the same purpose as before. For each column k, the innermost loop extracts all the previous k-1 Reflections(v and beta) and applies them to column k. Then the k^{th} transformation is computed and stored without making any further computations to matrix A. Then the k^{th} column of R is computed, note that the output argument s from gallery is the $||.||_2$ of that particular column.

6.5 Sparse QR Decomposition

Sparse QR Factorization algorithm is based on **qr_left** algorithm which is analyzed in this chapter. For the matrix A which will be factorized the following assumptions are made:

1. Every element in the diagonal of A is nonzero.
2. A has the strong Hall Property
3. Numerical cancellation is ignored.

A matrix $A \in R^{m \, x \, n}, m \geq n$ has the **strong Hall** property if for every subset of k columns, $0 < k < m$, the corresponding submatrix has nonzero elements in at least k+1 rows. (Thus, when m > n, every subset of $k \leq n$ columns has the required property, and when $m = n$, every subset of $k < n$ columns has the property.) For matrix with that property, structural cancellation will not occur[11]. For instance, consider the two following matrices, where * denotes nonzero elements: $\begin{bmatrix} * & * & * & * & * \\ & * & & & \\ & & * & & \\ & & & * & \\ & & & & * \end{bmatrix} \begin{bmatrix} * & * & * & * \\ * & & & \\ & * & & \\ & & * & \\ & & & * \end{bmatrix}$ The first matrix does not have the strong Hall property since for $k = 1$ the first column has nonzero elements in first row. On the contrary, in the second matrix for $k = 1$, each column has two nonzeros, for $k = 2$, four nonzeros and so forth, so the matrix has the strong Hall property.

6.5.1 Row Set-Union Property

Let $\mathcal{R}_{i*}$ and $\mathcal{R}_{*j}$ denote the nonzero pattern of the row i and column j respectively, $\mathcal{V}_k$ is the nonzero pattern of u_k and $A^{(k)}$ the matrix A after the $k-1$ Householder Reflection.

Consider $HA = A - u(\beta(u^T A))$. Then $(HA)_{i*}$, is equal to row i of A, if $i \notin \mathcal{V}$. If, $i \in \mathcal{V}$, the nonzero pattern of $(HA)_{i*}$ is $\bigcup_{i \in \mathcal{V}} \mathcal{A}_{i*}$.
That means the nonzero pattern of any modified row $i \in \mathcal{V}$ is replaced with the set union of all rows modified by the Householder Reflection H[34].

The product $u(\beta(u^T A))_{ij}$ is nonzero only if $i \in \mathcal{V}$ and $j \in \bigcup_{i \in \mathcal{V}} \mathcal{A}_{i*}$.

Consider $A = \left[\begin{array}{c|c|c|c|c} * & & * & & \\ & * & * & & \\ * & & & & * \\ & * & & * & \\ * & & & & \end{array}\right]$, then for the Householder Reflection of first column:

$$\mathcal{V} = \{1, 3, 5\}.\ u^T A = \begin{bmatrix} * & 0 & * & 0 & * \end{bmatrix} \left[\begin{array}{c|c|c|c|c} * & & * & & \\ & * & * & & \\ * & & & & * \\ & * & & * & \\ * & & & & \end{array}\right] = \begin{bmatrix} * & 0 & * & 0 & * \end{bmatrix}.$$

$$\text{Then, } u * (u^T A) = \begin{bmatrix} * \\ 0 \\ * \\ 0 \\ * \end{bmatrix} \begin{bmatrix} * & 0 & * & 0 & * \end{bmatrix} = \begin{bmatrix} * & 0 & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * \end{bmatrix}.$$

For $i = \{2, 4\} \notin \mathcal{V}$ the whole line is 0. For $i = \{1, 3, 5\} \in \mathcal{V}$ and $j = \{2, 4\} \notin (\mathcal{A}_{1*} \cup \mathcal{A}_{3*} \cup \mathcal{A}_{5*})$ the elements are 0. For $i = \{1, 3, 5\}$ and $j = \{1, 3, 5\} \in (\mathcal{A}_{1*} \cup \mathcal{A}_{3*} \cup \mathcal{A}_{5*})$ the elements are nonzero.

Then the product $u(\beta(u^T A))_{ij}$ is subtracted by A to get HA. Thus,

$HA = \left[\begin{array}{ccccc} * & & * & & \\ \hline & * & * & & \\ \hline * & & & & * \\ \hline & * & & * & \\ \hline * & & & & \end{array}\right] - \begin{bmatrix} * & 0 & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * \\ 0 & 0 & 0 & 0 & 0 \\ * & 0 & * & 0 & * \end{bmatrix} = \left[\begin{array}{ccccc} * & & * & & * \\ \hline & * & * & & \\ \hline \circ & & * & & * \\ \hline & * & & * & \\ \hline \circ & & * & & * \end{array}\right]$. The nonzero pattern of rows $i \in \mathcal{V} = \{1,3,5\}$ are replaced by $\bigcup_{i \in \mathcal{V}} \mathcal{A}_{i*} = (\mathcal{A}_{1*} \cup \mathcal{A}_{3*} \cup \mathcal{A}_{5*}) = (\{1,3\} \cup \{1,5\} \cup \{1\}) = \{1,3,5\}$, while the nonzero pattern of the other rows remain intact.

Consider the resulting matrix R of QR=A is of the following form:

$$R = \begin{bmatrix} * & & & & & & & \\ & * & & 0 & * & & * & \\ & & * & & & & & \\ & & & * & & & & \\ & & & & * & & & \\ & & & & & * & & \\ & & & & & & * & \\ & & & & & & & * \end{bmatrix}. A = \begin{bmatrix} * & & & & & & & \\ & * & & & & & & \\ & & * & & & & & \\ & & & * & & & & \\ & * & & 0 & * & & & \\ & & & & & * & & \\ & * & & 0 & & & * & \\ & & & & & & & * \end{bmatrix}.$$

The element $R_{2,4} = 0$ means that the Householder for the second column of A did not involve the fourth column because then the second line would have become the set union of both and the element (2,4) would have been made nonzero by (4,4). This means that for every nonzero element in the second column of A the corresponding in the fourth column of A are zero.
On the contrary, the element $R_{2,5} \neq 0$ and $R_{2,7} \neq 0$ as well. This means that the elements $A_{5,2} \neq 0$ and $A_{7,2} \neq 0$ and in the HA matrix these lines will become the set-union of each other.
So the matrix R is shows which Householder transformation should be applied to each column. Thus, the nonzero pattern of R_{*k} should be computed before applying the actual transformations, in a similar manner as in Cholesky Factorization seen in Chapter 5.

6.5.2 Computation of $\mathcal{R}_{*k}$

Let $QR = A \iff (QR)^T QR = (QR)^T A \iff R^T Q^T QR = A^T A \iff R^T R = A^T A$. R^T is a lower triangular matrix and R is an upper triangular so, if $L = R^T$ then $L^T = R$ and thus $LL^T = A^T A$, which is the Cholesky Factorization of $A^T A$[45].

The following theorems are useful for exploiting the QR-Cholesky relation[37]:

1. If $A_{kk}^{(k-1)}$ is structurally nonzero for all $1 \leq k \leq n$, then $\mathcal{R}_{i*} = \mathcal{A}_{i*}^{(k)}$.

2. $\mathcal{R}_{*k} = Reach_{T_k}(\mathcal{C}_k)$, where $\mathcal{C}_k$ is the upper triangular part of column k of $A^T A$.

Let $C = A^T A = \sum_{i=1}^{m} A_{i*}^T A_{i*}$. For instance $C = \begin{bmatrix} * & & & \\ & * & * & * \\ & & * & \\ & & & * \end{bmatrix} \begin{bmatrix} * & & & \\ & * & & \\ & * & * & \\ & * & & * \end{bmatrix} =$

$$\begin{bmatrix} * \\ \\ \\ \\ \end{bmatrix}\begin{bmatrix} * & & & \end{bmatrix} + \begin{bmatrix} \\ * \\ \\ \\ \end{bmatrix}\begin{bmatrix} & * & & \end{bmatrix} + \begin{bmatrix} \\ * \\ * \\ \\ \end{bmatrix}\begin{bmatrix} & * & * & \end{bmatrix} + \begin{bmatrix} \\ * \\ \\ * \end{bmatrix}\begin{bmatrix} & * & & * \end{bmatrix} =$$

$$\begin{bmatrix} * & & & \\ & & & \\ & & & \\ & & & \end{bmatrix} + \begin{bmatrix} & & & \\ & * & & \\ & & & \\ & & & \end{bmatrix} + \begin{bmatrix} & & & \\ & * & * & \\ & * & * & \\ & & & \end{bmatrix} + \begin{bmatrix} & & & \\ & * & & * \\ & & & \\ & * & & * \end{bmatrix} = \begin{bmatrix} * & & & \\ & * & * & * \\ & * & * & \\ & * & & * \end{bmatrix}.$$

If $a_{ik} \neq 0$, every entry $c_{k1,k2} \neq 0$ for any pair of column indices $k1, k2 \in \mathcal{A}_{i*}$. Consider $\mathcal{A}_{3*} = \{2,3\}$, the third submatrix shows that $c^{[3]}_{2,2}, c^{[3]}_{2,3}, c^{[3]}_{3,2}, c^{[3]}_{3,3} \neq 0$ (All the combinations).
Recall from Chapter 5.8 that $k1 < k2$ means the path $k1 \rightsquigarrow k2$ exists in T^k, thus only k_1 is needed to compute the $Reach_{T^k}(\mathcal{C}_k)$. So, for every entry $a_{ik} \neq 0$ and because all the combinations of $\mathcal{A}_{i*}$ will exist on the C matrix, only the $j \in \mathcal{A}_{i*}$ with minimum index value is needed to compute the $Reach_{T^k}(\mathcal{C}_k)$, or in other words the leftmost entry of the row i. Thus,

$$\mathcal{R}_{*k} = Reach_{T_k}(\mathcal{C}_k) = \mathcal{R}_{*k} = Reach_{T_k}(\{min\mathcal{A}_{i*} | i \in \mathcal{A}_{*k}\})[33].$$

6.5.3 Computation of $\mathcal{V}_k$

Consider the figure 6.1. $k2 = parent(k1)$ and $k = parent(k2)$, because $R = L^T$ the first off diagonal entry in the upper triangular is the parent. If the element $(i, k1)$ is nonzero then by the reflection properties $(i, k2)$ will be as well. Subsequently, (i, k) will become nonzero due to $(i, k2)$. So to find, the nonzero structure of k column after all the Householders have been applied it is sufficient to consider only the pattern of the **children**, all the other descendants are **redundant**.

Now, consider the figure 6.2 where $k = parent(k1, k2)$. The set of nonzeros of $k1, k2$ must be disjoint. Bear in mind the transformation for column $k1$ and assuming the element $(i2, k1)$ is nonzero, this will cause the lines $i, i2$ to become the set-union of each other that is why $(i2, k)$ must be nonzero. On the other hand, if the element $(i2, k2)$ was nonzero then the element $(i, k2)$ would have become nonzero, which means that $k2 = parent(k1)$ which is a contradiction with our first assumption. Thus, the sets of the children of a node are **disjoint**.

So the computation of $\mathcal{V}_k$ is given by: $\mathcal{V}_k = \left(\bigcup_{k=parent(i)} V_i \setminus \{i\} \right) \cup \{i | k = min\mathcal{A}_{i*}\}$.
The term $\{i | k = min\mathcal{A}_{i*}\}$ accounts for all the elements in column k that are the leftmost nonzero in a line and being so, the previous Reflections did not affect those lines. For instance in this matrix the second column has two leftmost entries in lines {2,4} accounted by the previous term.

Because all the sets are disjoint the count of nonzeros of V_k is:

$$|\mathcal{V}_k| = \left(\sum_{k=parent(i)} |\mathcal{V}_i| - 1 \right) + |\{i | k = min\mathcal{A}_{i*}\}|$$

All the theoretical background needed has been analyzed, so now its time to have a look at the algorithmic procedure[23].

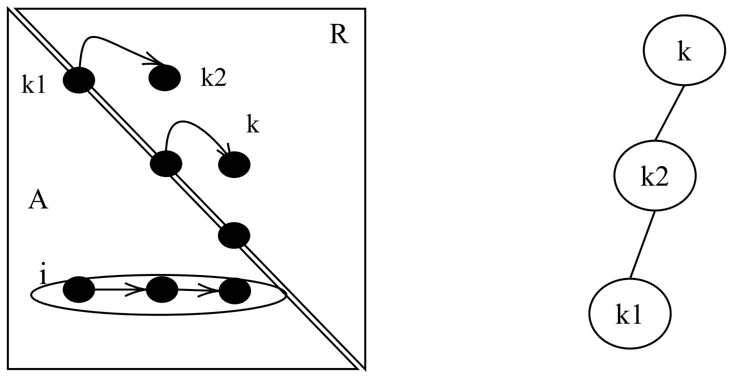

Figure 6.1: Parent-Descendant Relation

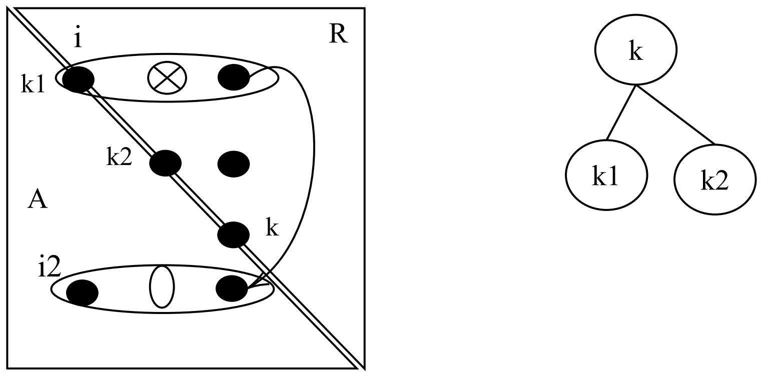

Figure 6.2: Disjoint Children

6.5.4 Sparse QR Algorithm

An overview of the algorithm is the following:

```
function [V,$\beta$,R]=left_qr_sparse(A)
    $\mathcal{T}$=etree($A^TA$)
    |$\mathcal{R}$|=colcounts($A^TA$)
    $|\mathcal{V}_k| = \left(\sum_{k=parent(i)} |\mathcal{V}_i| - 1\right) + |\{i|k = min\mathcal{A}_{i*}\}|, \quad k = 1 \ldots n$

    for k=1:n
        $\mathcal{R}_{*k} = Reach_{T_k}(\{min\mathcal{A}_{i*}|i \in \mathcal{A}_{*k}\})$
        x=$\mathcal{A}_{*k}$
        $\mathcal{V}_k = \mathcal{A}_{*k}$

        for each $i \in \mathcal{R}_{*k}$
            $x = x - u_i(\beta_i(u_i^T x))$
            if parent(i)=k
                $\mathcal{V}_k = \mathcal{V}_k \cup \mathcal{V}_i \notin \{i\}$
            end
        end
        $R_{1...k-1,k} = x_{1...k-1}$
        $[u_k, \beta_k, r_{kk}]$=householder ($x_{k...m}$)
    end
end
```

Now, let us take a deeper insight into this algorithm, analyzing the basic its mechanics and concepts. First of all, a function must be created which computes $|\mathcal{V}_k| = \left(\sum_{k=parent(i)} |\mathcal{V}_i| - 1\right) + |\{i|k = min\mathcal{A}_{i*}\}|$.

Recall from Chapter 5.9 the structure **A_symb** used for the symbolic analysis and enter three new fields: leftmost, pinv, unz.

```
A_symb.pinv % store the inverse permutation (QR)
A_symb.parent % store the etree (Cholesky ,QR)
A_symb.post % store the post order permutation
A_symb.cp % store the column pointers found using colcounts(Chol),
           % row counts (QR)
A_symb.leftmost % the leftmost entry (QR)
A_symb.lnz  % the number of nz entries in L (Cholesky) or in V (QR)
A_symb.unz % the number of nz entries in R (QR)
```

Leftmost is an array in which leftmost(i)=$min\mathcal{A}_{i*}$ namely the column where the first nonzero entry in each row lies. Pinv is a row permutation which is needed to make sure the matrix A has nonzero diagonal. Finally, unz is a variable which contains the number of nonzero elements of each column of R matrix.

```
function [unz, pinv, leftmost]=vcount(A,parent)
    [m n]=size(A);
    next=array(m,1); % next entry in col k
    head=array(n,1); % first entry in col k
    tail=array(n,1); % last entry in col k
    nque=array(n,1); % number of entries each col

    for i=1:n
        leftmost(i)=min(find(A(i,:)));
    end

    for i=m:1
        pinv(i)=0;
        k=lefmost(i);
        if (nque(k)==0)
            nque(k)=nque(k)+1;
            tail(k)=i;
        end
        next(i)=head(k);
        head(k)=i;
    end
    lnz=0;

    for k=1:n
        i=head(k); % remove row i from queue
        lnz=lnz+1; % count diagonal entry V(k,k)
        nque(k)=nque(k)-1;
        pinv(i)=k;
        if (nque(k)<=0) % empty below the diagonal
            continue;
        end;
        lnz=lnz+nque(k); % nque(k) is the nnz count below the diagonal
        pa=parent(k);   % transfer all rows to parent
        if (pa~=-1)
            if(nque(pa)==0)
                tail(pa)=head(k);
            end
            next(tail(k))=head(pa);
            head(pa)=next(i);
            nque(pa)=nque(pa)+nque(k);
        end
    end
    for i=1:m % rows that did not hold a diagonal entry
        if (pinv(i)=0)
            k=k+1;
            pinv(i)=k;
        end
    end
end
```

The algorithm in the beginning finds the leftmost element of each row. Then, in the second loop, creates a ”list” of columns k which contains all the indexes i which are true to leftmost(i)=k. Then, in the third loop for each column k, the head row is removed and is the row the Householder does not zero out and so must not be used again, which made sure by pinv (Any row in the list of k would have been suitable to be the remainder of Householder). The assignment lnz=lnz+nque(k) adds to the sum the count of elements below the diagonal V(k+1:m,k) and the following lines of code transfer the remaining nodes of k to its parent.

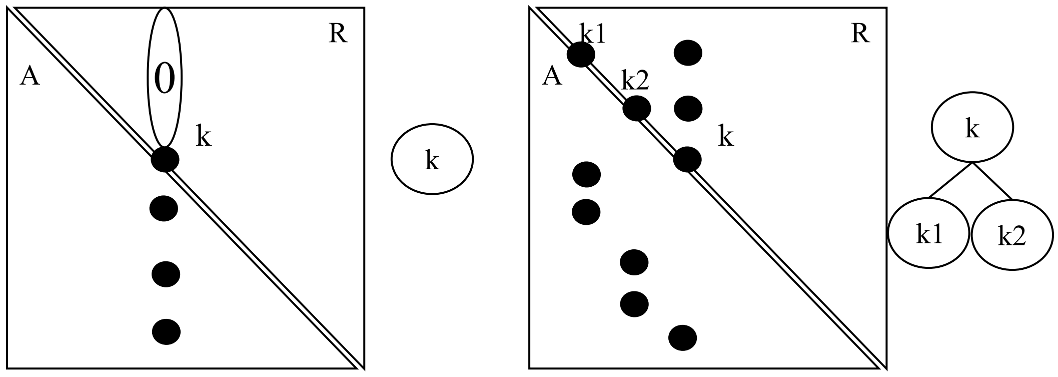

Figure 6.3: Node with no children

That last part is the algorithmic expression of the relation $|\mathcal{V}_k| = \left(\sum_{k=parent(i)} |\mathcal{V}_i| - 1\right) + |\{i|k = min\mathcal{A}_{i*}\}|$. Consider the node k of the left matrix in figure 6.3. Node k has no children, is a leaf, and so no previous Householder Reflections would have acted on it. So all the elements in the column are the leftmost in their rows. Thus, nque(k)=4 will account for the all those elements and that is the number of nonzeros in that $\mathcal{V}_k$.

Now, consider the node k of the right matrix. k has two children k1,k2. The number of nonzeros of $\mathcal{V}_k$ is the summation of the nonzeros of the children plus its own leftmost nonzeros. So if each child passes their nonzero elements to their father after it has been calculated, the latter has a list of all the nonzeros, which will pass on to its own father and so forth. Thus, list concatenation will account for the transfer and in constant time for each one, so in total O(n) because the total children of the graph are n.

To do the symbolic analysis some functions from previous chapter and the **vcount** introduced above should be called.

```
A_symb.parent=etree(A) % find the elimination tree
A_symb.post=post(A_symb.parent)
c=colcounts(A,A_symb.parent ,A_symb.post)
A_symb.cp=cumsum(c) % store the column pointers found using
                    %colcounts(Chol), row counts (QR)
[A_symb.lnz , A_symb.pinv , A_symb.leftmost]=vcount(A,A_symb.parent)
                    % use vcount to find lnz ,pinv and leftmost
A_symb.unz=sum(A_symb.cp)
```

At this moments every symbolic calculation is made, so it is possible to move on to the numerical factorization of QR.

```
function [V,R]=qr(A,A_symb)
    [m n]=size(A);
    Ai=A.i; Ap=A.p;
```

```
q=A_symb.q; parent=A_symb.parent;
pinrv=A_symb.pinv;
leftmost=A_symb.leftmost;
rnz=1; lnz=1;
w=zeros(m);

for k=1:n % compute V, R
    Rp(k)=rnz; % R(:,k) starts here
    Vp(k)=p1=lnz; % V(:,k) starts here
    w(k)=k; % V(k,k) pattern of V
    Vi(vnz)=k;
    vnz=vnz+1;
    top=n;
    for p=Ap(k):Ap(k+1)-1 % find R(:,k) pattern
        i=leftmost(Ai(p));
        len=1;
        while (w(i)~=k) % traverse up to k
            s(len)=i;
            len=len+1;
            w(i)=k;
            i=parent(i);
        end
        while(len>1) % push path on stack
            top=top-1;
            len=len-1;
            s(top)=s(len);
        end
        i=pinvA(i(p));
        x(i)=Ax(p); % x(i)=A(:,col)
        if ( i>k && w(i) < k) % pattern (V:,k)=x(k+1,m)
            Vi(vnz)=i;
            vnz=vnz+1; % add i to pattern of V(:,k)
            w(i)=k;
        end
    end
    for p=top:n % for i in R(:,k)
        i=s(p); % R(i,k) is nonzero
        for j=Vp(i):Vp(i)-1 % apply the previous Householders
            mul=mul+Vx(j)*x(Vi(p));
        end
        mul=mul*beta(i);
```

```
            for j=Vp(i):Vp(i)-1
                x(Vi(p))=x(Vi(p))-Vx(p)*mul;
            end
            Ri[rnz]=i;
            Rx[rnz]=x(i);
            rnz=rnz+1;
            x(i)=0;
        end
        for p=p1:vnz
            Vx(p)=x(Vi(p));
            x(Vi(p))=0;
        end
        Ri(rnz)=k;
        [v,beta]=house(x); % apply the current Householder
    end
    Rp(n)=rnz; % finalize the matrices
    Vp(n)=vnz;
    R.p=Rp; R.i=Ri; R.x=Rx;
    V.p=Vp; V.i=Vi; V.x=Vx;
end
```

This code, although it is the numeric part, does some symbolic work actually. That is because the colcounts returns the count of rows of R, because $R = L^T$ but the new matrices need to be stored by column, so the column pointers are computed on the way.
Inside the first for loop, the **Reach** of each node is computed, but the ascension up the tree, starts from the leftmost node of each nonzero row in column k, the w(i) is used to mark the visited nodes at each iteration. Each column inherits the the nonzeros of the children and the if statement checks for any new nonzeros in matrix A and not in the tree.
In the second loop the previous Householder transformations corresponding to column *k* are applied. Finally, the Householder vector of the k^{th} column is computed and stored.

An example of symbolic QR Decomposition is shown below:

$$A = \begin{bmatrix} * & & & & & \\ * & * & & * & * & \\ & & * & & & * \\ * & & & * & & \\ & & * & & * & \\ & & & * & * & * \\ & & & & & * \\ * & & & & & \\ * & * & & * & & \\ * & & & * & * & \end{bmatrix}, V = \begin{bmatrix} * & & & & & \\ * & * & & & & \\ & & * & & & \\ * & * & & * & & \\ & & * & & * & \\ & & & * & * & * \\ & & & & & * \\ * & * & & * & * & * \\ * & * & & * & * & * \\ * & * & & * & * & * \end{bmatrix}, R = \begin{bmatrix} * & * & & * & * & \\ & * & & * & * & \\ & & * & & * & * \\ & & & * & * & * \\ & & & & * & * \\ & & & & & * \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

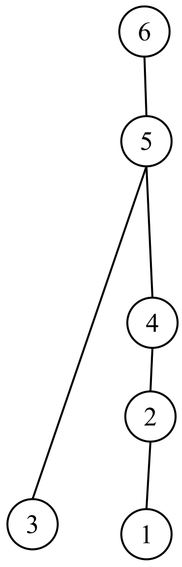

Figure 6.4: Column Elimination Tree

$$parent = \begin{bmatrix} 2 & 4 & 5 & 5 & 6 & 0 \end{bmatrix}$$

Chapter 7

LU Decomposition

7.1 Introduction

LU Decomposition is the matrix form of Gaussian Elimination and it is older than the Decomposition mentioned before. It is one of the most important and versatile matrix algorithms. It factorizes a matrix A to an a product of a lower triangular and an upper triangular matrix so **A=LU**.

7.2 Method Overview

Consider a matrix $A \in \mathbb{R}^{n\ x\ n}$, which is analyzed into two factors $L, R \in \mathbb{R}^{n\ x\ n}$ so that LU=A, where L is **lower triangular** and U is **upper triangular**. Then in matrix notation:

$$\begin{bmatrix} \ell_{11} & & & & \\ \ell_{21} & \ell_{22} & & & \\ \ell_{31} & \ell_{32} & \ell_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \dots & \ell_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \Longrightarrow$$

$$\begin{array}{lllll} \ell_{11}u_{11} = a_{11} & \ell_{11}u_{12} = a_{12} & \ell_{11}u_{13} = a_{13} & \dots & \ell_{11}u_{1n} = a_{1n} \\ \ell_{21}u_{11} = a_{21} & \ell_{21}u_{21} + \ell_{22}u_{22} = a_{22} & \ell_{21}u_{13} + \ell_{22}u_{23} = a_{23} & \dots & \ell_{21}u_{1n} + \ell_{22}u_{2n} = a_{2n} \\ \ell_{31}u_{11} = a_{31} & \ell_{31}u_{12}+\ell_{32}u_{22} = a_{32} & \ell_{31}u_{13}+\ell_{32}u_{23}+\ell_{33}u_{33} = a_{33} & \dots & \ell_{31}u_{1n}+\ell_{32}u_{2n}+\ell_{33}u_{3n} = a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ \ell_{n1}u_{11} = a_{n1} & \ell_{n1}u_{12} + \ell_{n2}u_{22} = a_{n2} & \ell_{n1}u_{13} + \ell_{n2}u_{23} + \ell_{n3}u_{33} = a_{n3} & \dots & \end{array}$$

$$\dots \qquad \ell_{n1}u_{1n3} + \ell_{n2}u_{2n} + \ell_{n3}u_{3n} + \cdots + \ell_{nn}u_{nn} = a_{nn}.$$

This is a system with n^2 equations and $\sum_{i=1}^{n}\sum_{j=i}^{n} 1 = \frac{n(n+1)}{2}$ unknown variables from each matrix, so in total $n(n+1) = n^2 + n$. The variables surpass the equations, that gives us freedom to choose arbitrarily the values of the n variables. Thus, the n-diagonal entries of the lower triangular matrix L are set equal to 1. Substituting to the previous equations:

$$\begin{bmatrix} 1 & & & & \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \dots & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ & u_{22} & u_{23} & \dots & u_{2n} \\ & & u_{33} & \dots & u_{3n} \\ & & & \ddots & \vdots \\ & & & & u_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \implies$$

$$\begin{array}{lllll} u_{11} = a_{11} & u_{12} = a_{12} & u_{13} = a_{13} & \dots & \ell_{11}u_{1n} = a_{1n} \\ \ell_{21}u_{11} = a_{21} & \ell_{21}u_{21} + u_{22} = a_{22} & \ell_{21}u_{13} + u_{23} = a_{23} & \dots & \ell_{21}u_{1n} + u_{2n} = a_{2n} \\ \ell_{31}u_{11} = a_{31} & \ell_{31}u_{12}+\ell_{32}u_{22} = a_{32} & \ell_{31}u_{13}+\ell_{32}u_{23}+u_{33} = a_{33} & \dots & \ell_{31}u_{1n}+\ell_{32}u_{2n}+u_{3n} = a_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ \ell_{n1}u_{11} = a_{n1} & \ell_{n1}u_{12} + \ell_{n2}u_{22} = a_{n2} & \ell_{n1}u_{13} + \ell_{n2}u_{23} + \ell_{n3}u_{33} = a_{n3} & \dots & \\ \dots & \multicolumn{4}{l}{\ell_{n1}u_{1n3} + \ell_{n2}u_{2n} + \ell_{n3}u_{3n} + \cdots + u_{nn} = a_{nn}.} \end{array}$$

Writing this in block matrix form leads to the **right-looking** LU factorization discussed in next section.

7.3 Right-Looking LU Decomposition

$\begin{bmatrix} 1 & \\ \ell_{21} & \ell_{22} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} \\ & U_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & A_{22} \end{bmatrix}$, where u_{11}, a_{11} are scalars, ℓ_{21}, u_{12} are vectors and ℓ_{22}, U_{22} are submatrices.

$$u_{11} = a_{11} \qquad u_{12} = a_{12} \qquad \ell_{21}u_{11} = a_{21}$$
$$\ell_{21}u_{12} + \ell_{22}U_{22} = A_{22} \implies \ell_{22}U_{22} = A_{22} - \ell_{21}u_{12}$$

The algorithm for the right case is shown below for a dense matrix:

```
function [L,U] = lu_right(A)
    n=size(A,1);
    L=eye(n);
    U=zeros(n);
    for k=1:n
        U(k,k:n)=A(k,k:n);
        L(k+1:n,k)=A(k+1:n,k)/U(k,k);
        A(k+1:n,k+1:n)=A(k+1:n,k+1:n)-L(k+1:n,k)*U(k,k+1:n);
    end
end
```

This algorithm is using outer products, the $\ell_{21}u_{12}$ matrix which is subtracted by the A submatrix. That makes it hard to implement it in the sparse case because it adds entries in the middle of the matrix.

7.4 Numerical Issues

7.4.1 Numerical Issues

This is the first algorithm shown so far in which, a lot of attention must be paid to the **numerical** factorization along with the **symbolic** one. LU factorization is closely related to Gaussian Elimination which is unstable in its original form. Notice from 7.2 that to find ℓ_{21} a division is needed with u_{11}, $\ell_{21} = a_{21}/u_{11}$ and that division is needed for every element in the L matrix. To complete the process, we must ensure that no division with zero is made, so **pivoting** is used.[60]

Consider the matrix $A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$. If the process begins, with the A matrix as it is, it will fail, but if the lines are pivoted so that $A = \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$, the LU factorization can continue and is in fact complete. Thus, $LU = PA$, where P is a permutation matrix in this example $P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 2 & 3 \\ 0 & 1 \end{bmatrix}$

Another reason that Gaussian Elimination is unstable is due to the small pivots that may occur. For instance, consider: $A = \begin{bmatrix} 0.000100 & 1 \\ 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$. Solving Ax=b results in:

$x = \begin{bmatrix} 1.00010001\ldots \\ 0.998998\ldots \end{bmatrix} \approx \begin{bmatrix} 1.00 \\ 1.00 \end{bmatrix}$ using decimal approximation of 3 points.

If Gaussian Elimination is used: $\begin{bmatrix} 0.000100 & 1 \\ 0 & 1-10000 \end{bmatrix} x = \begin{bmatrix} 1 \\ 1-10000 \end{bmatrix}$.

The number 1-10000 is not represented exactly due to **floating point arithmetic** but it will be rounded to the nearest floating point number, which is assumed to be -10000. Thus:

$\begin{bmatrix} 0.000100 & 1 \\ 0 & -10000 \end{bmatrix} x = \begin{bmatrix} 1 \\ -10000 \end{bmatrix} \implies x = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. This result is totally inaccurate compared to the one above. [26]

Obviously, this is an error that happens to LU factorization too. Consider $A = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 2 \end{bmatrix}$.

Its LU factorization is : $L = \begin{bmatrix} 1 & \\ 10^{20} & 1 \end{bmatrix}$, $U = \begin{bmatrix} 10^{-20} & 1 \\ 0 & 2-10^{20} \end{bmatrix}$. Once again, assuming that due to rounding the term u_{22} becomes -10^{20} and so: $L = \begin{bmatrix} 1 & \\ 10^{20} & 1 \end{bmatrix}$, $U' = \begin{bmatrix} 10^{-20} & 1 \\ 0 & -10^{20} \end{bmatrix}$. This leads to: $LU' = \begin{bmatrix} 10^{-20} & 1 \\ 1 & 0 \end{bmatrix}$. Evidently, $LU' \neq A$, because although LU was **computationally stable**, it was not **backward stable**.[60].

Once again, pivoting before applying the Gaussian Elimination or the LU factorization would minimize the problem. For instance consider the first example:

$\begin{bmatrix} 1 & 1 \\ 0.000100 & 1 \end{bmatrix} x = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \implies \begin{bmatrix} 1 & 1 \\ & 1-0.000100 \end{bmatrix} x = \begin{bmatrix} 2 \\ 1-2*0.000100 \end{bmatrix}$. Once again due to rounding assumed, this results in: $\begin{bmatrix} 1 & 1 \\ & 1 \end{bmatrix} x = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \implies x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, which is the same as the 3 decimal point approximation.

The some happens in the LU case, $LU = PA = \begin{bmatrix} 1 & 2 \\ 10^{-20} & 1 \end{bmatrix} \implies L = \begin{bmatrix} 1 & \\ 10^{-20} & 1 \end{bmatrix}, U = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$

$LU = \begin{bmatrix} 1 & 2 \\ 10^{-20} & 1 \end{bmatrix} = A.$

The problem described above is of high importance because it amplifies the insignificant data which are added to significant data resulting in **noise** in the solution. The reason it did not happen in the previous algorithms is because QR is just permuting the data while keeping the length constant, orthogonal operations are safe numerically. In the Cholesky case is because it has an excellent numerical stability due to the fact that $||A||_2 = ||LL^T||_2 = ||L^T||_2^2 = ||L||_2^2$ [47]. Only a very **ill-conditioned** matrix will result in an unstable Cholesky Factorization.[25].

7.4.2 Partial Pivoting

One method of overcoming the instability issues in the above, is called **Partial Pivoting**. In fact, it is a generalization of the technique used to overcome the problems tackled previously.

In this method, at each iteration of the Gaussian Elimination or LU Decomposition the largest element in the column which will be zeroed out. This results in a pivoting value with magnitude that does not surpass 1. In iteration k, the pivot is the largest of n-(k-1) sub-diagonal entries in column k. When located it is moved into the pivot position $A_{kk}^{(k-1)}$, where $A^{(k)}$ denotes the matrix A after the kth elimination has been applied, so $A^{(n-1)} = U$. Also, in order to move the row of the largest element a multiplication with a permutation matrix is needed at each step, denoted by P_k. Also ℓ_k denotes the L component at each step. Consider an example below: $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$

$P_1A = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 0 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}$, then using ℓ_1 to eliminate both entries in the first column:

$$\ell_1 P_1 A = \begin{bmatrix} 1 & & \\ -4/7 & 1 & \\ -1/7 & 0 & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 0 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 0 \\ 0 & 3/7 & 6 \\ 0 & 6/7 & 3 \end{bmatrix} = A^{(1)}$$

$$P_2A^{(1)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 7 & 8 & 0 \\ 0 & 3/7 & 6 \\ 0 & 6/7 & 3 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 0 \\ 0 & 6/7 & 3 \\ 0 & 3/7 & 6 \end{bmatrix}$$

$$\ell_2 P_2 A^{(1)} = \begin{bmatrix} 1 & & \\ 0 & 1 & \\ 0 & -1/2 & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 0 \\ 0 & 6/7 & 3 \\ 0 & 3/7 & 6 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 0 \\ 0 & 6/7 & 3 \\ 0 & 0 & 9/2 \end{bmatrix} = A^{(2)} = U$$

$L = \ell_1'^{-1} \ell_2'^{-1} = \begin{bmatrix} 1 & & \\ 4/7 & 1 & \\ 1/7 & 1/2 & 1 \end{bmatrix}$, where $\ell_1' = P_2 \ell_1 P_2^{-1}$ and $\ell_2' = \ell_2$. $P = P_1P_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$

Therefore, $(L'_{n-1}\dots L'_1)(P_{n-1}\dots P_1)A = U \implies (P_{n-1}\dots P_1)A = (L'_{n-1}\dots L'_1)^{-1}U \implies PA = LU$, where $L = (L'_{n-1}\dots L'_1)^{-1}$ and $P = (P_{n-1}\dots P_1)$.

Partial pivoting is the most common way to maintain numerical stability, but permuting rows is affecting the symbolic analysis too. In many cases, they antagonize each other. Consider the following scenario: $A = \begin{bmatrix} * & & & \\ * & * & & \\ * & & * & \\ \star & * & * & * \end{bmatrix}$ and the entry a_{41} is the largest entry on the first column. Then, by permuting the rows 1 and 4 and doing the Elimination results in the following matrix: $A' = \begin{bmatrix} \star & * & * & * \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix}$, neglecting numerical cancellations. Now, the matrix has been transformed from having a *sparse* structure to a *dense* one due to all those fill-in entries.

In the next section a method to predict a bound on fill-in created by LU factorization is going to be exploited

7.5 Upper fill-in bound

Recall from 7.3 the relation $\ell_{22}U_{22} = A_{22} - \ell_{21}u_{12}$ and consider $\ell_{21}u_{12}$ having this structure: $\ell_{21}u_{12} = \begin{bmatrix} * \\ \\ * \\ * \\ * \end{bmatrix} \begin{bmatrix} & * & * & * \end{bmatrix} = \begin{bmatrix} & * & * & * \\ & & & \\ & * & * & * \\ & * & * & * \\ & * & * & * \end{bmatrix}$ which is then subtracted by A_{22}, so this non-zero pattern is added to A_{22} structure. ℓ_{21} has the non-zero structure of the column a_{21} which are going to be zeroed-out by the Elimination. Thus in order to zero-out all the elements, the fill-in entries propagated to the rest of the matrix is the multiplication of the column going to be erased by the pivot row.

If the all the rows having an entry that is going to be deleted are grouped in the upper part of the matrix and all the non-zero elements are grouped to the left side, while in the bottom line are the rows not affected at all the following structure arises:

$$A = \left[\begin{array}{c|cccc|cc} \star & & & & * & 0 & \dots \\ * & & * & * & & \vdots & \dots \\ * & & * & & & 0 & \dots \\ * & & & * & & \vdots & \dots \\ * & * & * & & & 0 & \dots \\ \hline 0 & & & * & & * & \dots \\ \vdots & & & \vdots & & \vdots & \end{array}\right] \implies A' = \left[\begin{array}{c|cccc|cc} \star & & & & * & 0 & \dots \\ * & & * & * & * & \vdots & \dots \\ * & & * & & * & 0 & \dots \\ * & & & * & * & \vdots & \dots \\ * & * & * & & * & 0 & \dots \\ \hline 0 & & & * & & * & \dots \\ \vdots & & & \vdots & & \vdots & \end{array}\right].$$

The matrix A' shows that the non-zero structure of the pivot row is going to be propagated only inside the upper left partition. Recall from 6.3 the same partition for QR. When QR is used, the upper left partition is all filled up with the set-union of the rows so it all becomes nonzero, which is the ultimate case for LU. Thus, LU fill-in is bounded by QR.

This bound holds for partial pivoting too. If the largest entry is on the fourth row then:

$$A = \left[\begin{array}{c|cccc|cc} \ast & & & & \ast & 0 & \dots \\ \ast & & \ast & \ast & & \vdots & \dots \\ \ast & & \ast & & & 0 & \dots \\ \star & & & \ast & & \vdots & \dots \\ \ast & \ast & \ast & & & 0 & \dots \\ \hline 0 & & & \ast & & \ast & \dots \\ \vdots & & & \vdots & & \vdots & \end{array}\right] \Longrightarrow PA = \left[\begin{array}{c|cccc|cc} \star & & & \ast & & \vdots & \dots \\ \ast & & \ast & \ast & & \vdots & \dots \\ \ast & & \ast & & & 0 & \dots \\ \ast & & & & \ast & 0 & \dots \\ \ast & \ast & \ast & & & 0 & \dots \\ \hline 0 & & & \ast & & \ast & \dots \\ \vdots & & & \vdots & & \vdots & \end{array}\right] \Longrightarrow$$

$$A' = \left[\begin{array}{c|cccc|cc} \star & & & \ast & & \vdots & \dots \\ \ast & & \ast & \ast & & \vdots & \dots \\ \ast & & \ast & \ast & & 0 & \dots \\ \ast & & & \ast & \ast & 0 & \dots \\ \ast & \ast & \ast & \ast & & 0 & \dots \\ \hline 0 & & & \ast & & \ast & \dots \\ \vdots & & & \vdots & & \vdots & \end{array}\right].$$

The nonzeros propagated are still in the upper left part of the matrix. When the symbolic factorization is being done, there is no information about the numerical values so we do not know which row is going to be the pivot row but no matter which is the one, is going to be bounded by the QR and this is a bound that can be predicted beforehand.

If the matrix A is **strong Hall** then R is an **upper bound** on the nonzero pattern of U. Meaning that, $r_{ij} \neq 0 \iff u_{ij} \neq 0$[43][40][37]. In Householder Reflections the nonzero pattern of all rows affected by the transformation becomes the set-union of all these rows. In LU with Partial Pivoting these rows are candidate for pivoting but only one is selected as a pivot. The remaining candidate rows are modified by adding a scaled multiple of the pivot row [23]. So QR acts in a way that sets all candidate rows as pivot rows.

If the matrix A is **strong Hall** and $a_{kk}^{(k-1)} \neq 0, k = 1 \dots n$, the Householder matrix V is an upper bound on the nonzero pattern of L acquired with partial pivoting. Meaning, $v_{ij} \neq 0 \iff \ell_{ij} \neq 0$ [43][40].

So a worst-case estimate for the fill-in in LU can be computed and the way is already known from previous chapters. Doing Cholesky Symbolic Analysis of ($A^T A$) gives the exact structure of QR which is an upper bound for LU, without using numerical values at all. Therefore, there are orderings which can be used to reduce the fill-in in Cholesky, leading to less fill-in in QR and then tighten the bound of LU. This is done by using column permutation matrices P_c, the row permutations do not affect QR because $(PA)^T PA = R^T R \implies A^T P^T PA = R^T R \implies A^T A = R^T R$ due to P being orthogonal. In column permutation $(AP_c)^T AP_c = R^T R \implies P_c^T A^T AP_c = R^T R$. Since, a reduced fill-in column ordering has been used then, partial pivoting of LU is free to choose the more suitable row as a pivot.

7.6 Left-Looking LU Decomposition

The *Left-Looking* method computes one column of L and U at a time. At each step all the previous columns of L and A are accessed. Each matrix is decomposed into a 3-by-3 block matrix:

$$\begin{bmatrix} L_{11} & & \\ \ell_{21} & 1 & \\ L_{31} & \ell_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & u_{12} & U_{13} \\ & u_{22} & u_{23} \\ & & U_{33} \end{bmatrix} = \begin{bmatrix} A_{11} & a_{12} & A_{13} \\ a_{21} & a_{22} & a_{23} \\ A_{31} & a_{32} & A_{33} \end{bmatrix},$$

where $\ell_{21}, u_{23}, a_{21}, a_{23}$ are row vectors, $\ell_{32}, u_{12}, a_{12}, a_{32}$ are column vectors, $\ell_{22} = 1, u_{22}$ are scalars and the rest are submatrices. The middle row and column of each matrix is considered the k^th row and column which is computed at each step. Thus, the previous k-1 columns are assumed to be known. Expanding the system results into $3x3 = 9$ equations but only 3 are actually needed:

1. $L_{11}u_{12} = a_{12}$, which is a sparse triangular solve for column-vector u_{12}
2. $\ell_{21}u_{12} + u_{22} = a_{22} \implies u_{22} = a_{22} - \ell_{21}u_{12}$, which is a scalar minus a dot-product to solve for the pivot entry u_{22}
3. $L_{31}u_{12}+\ell_{32}u_{22} = a_{32} \implies \ell_{32} = \dfrac{a_{32} - \ell_{31}u_{12}}{u_{22}}$, which is a sparse matrix-vector multiplication subtracted by a column vector and then divided by a scalar to solve for column vector ℓ_{32}

The equations above can be altered in a way that the solution to all of them is given by just 1 triangular solve. If a lower trapezoidal matrix of the following form is constructed: $\begin{bmatrix} L_{11} & & \\ \ell_{21} & 1 & \\ L_{31} & 0 & I \end{bmatrix}$, and then solving a system with the right-hand side being the k^th column of A results in:

$$\begin{bmatrix} L_{11} & & \\ \ell_{21} & 1 & \\ L_{31} & 0 & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix}$$

Expanding the following equations arise:

1. $L_{11}x_1 = a_{12}$
2. $\ell_{21}x_1 + x_2 = a_{22}$
3. $L_{31}x_1 + x_3 = a_{32}$

If we substitute x_1 with u_{12}, x_2 with u_{22} and x_3 with $\ell_{32}u_{22}$ then:

1. $L_{11}u_{12} = a_{12}$
2. $\ell_{21}u_{12} + u_{22} = a_{22}$
3. $L_{31}u_{12} + \ell_{32}u_{22} = a_{32}$

These are exactly the same equations as before grouped into one triangular solve. After the solution only a division is needed $\ell_{32} = x_3/u_{22}$ and the unknowns are computed.

Once x is found, entries in the rows k through n can be searched for the entry with the largest magnitude so as to be chosen as the pivotal entry [22]. That is because the 2,3 relations of the upper equations are just multiplications, so if the permutations are applied beforehand or afterwards the result is the same. Only the first relation is affected by the order, because it is a triangular solve and the correct order matters to yield an accurate solution. Therefore, the only consistency needed is between the nonzero in the L matrix and the target vector in the triangular solve, it does not really matter if the rows are permuted in the correct position right away as long as the are consistent with each other.

7.6.1 Left-Looking Dense LU Algorithm

The Matlab algorithm of the Left-Looking method is presented below. The algorithm handles a dense matrix and uses partial pivoting.[23]

```
function [L,U,P]=left_looking_lu(A)
    n=size(A,1);
    P=eye(n);
    L=zeros(n);
    U=zeros(n);
    for k=1:n
        % create LHS matrix :: 1:k-1 columns of L, k:n Identity ,
        x_start=[L(:,1:k-1) [zeros(k-1,n-k+1);eye(n-k+1)]];
        x=x_start\(P*A(:,k)); % sparse triangular solve to find x
        U(1:k-1,k)=x(1:k-1); % result of x1 are the (1:k-1,k) entries
                             % of U
        [a,i]=max(abs(x(k:n))); % find the new pivot
        i=i+k-1; % dimensions of submatrix to
                 % dimensions of the whole matrix
        L([i k],:)=L([k i],:); % row permutations of L
        P([i k],:)=P([k i],:); % row permutations of P
        x([i k])=x([k i]); % row permutations of x
        U(k,k)=x(k); % store the pivot entry
        L(k,k)=1; % diagonal entries of L=1
        L(k+1:n,k)=x(k+1:n)/x(k); % ℓ32 = x3/u22
    end
end
```

Note that the algorithm first computes the x solution vector and then chooses the pivot row and does the swapping because it is needed for the next iteration to be used in the triangular solve to find $x_1^{(k+1)}$.

Implementing the row permutation to a sparse method, is costly and not worth the cost. That is because the matrices are stored by column, to swap row indices in each column in each iteration outweighs the cost of the sparse triangular solve. Thus, in total the row swapping can cost asymptotically a lot more than the whole LU factorization[22].

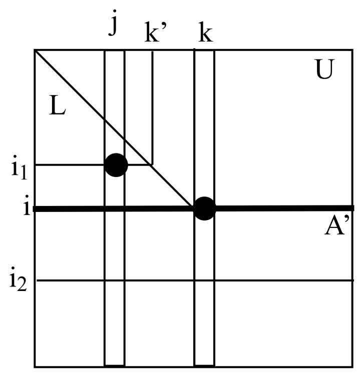

Figure 7.1: New and Old Data Space Relation

7.6.2 Left-Looking Sparse LU Algorithm

To exploit sparsity in the Left-Looking Algorithm, a convenient method to do the partial pivoting must be implemented that does not do any row permutation during the procedure but applies all of them at the end.

To do so the data structure is left unchanged and thus, the old row indices are used, the ones corresponding to the rows in A matrix. The one thing that needs to be tracked is actually which row i becomes the kth pivot row at step k. To keep track of the actual place of a row an inverse permutation vector is used. It is like looking at the L matrix through a lens that transforms the data from the old space to the new. Therefore, after all the row i needs to become the row k in the final L matrix, as well as every other row i'.

Consider Figure 7.1. At the kth step the *i* row of the A matrix is chosen as pivot. So the row *i* of A which will become the *k* row of L is in the old space. On the contrary, the column indices reflect the new space. Thus, in the matrix L the columns are indexed in the proper way while the rows still use the old indices.

Then, for the lower triangular solve, which is the only equation that the ordering really matters, suppose the Reach(j) needs to be computed. J reaches to row i_1 which is the row of the A matrix, thus it needs to be remapped to the new index of L and so pinv(i_1)=k' so Reach(j)=k'. All the rows below row i have not yet been chosen as pivots so they reflect to Identity and thus, they do not have any outgoing edges and will not add extra work to the depth-first search. The only issue that arises is that since these lines have no name in the new work space, the only consistent way to refer to all these nodes is through their old names.

So, generally the part 1:k-1 in of matrix $\begin{bmatrix} L_{11} & & \\ \ell_{21} & 1 & \\ L_{31} & 0 & I \end{bmatrix}$ are well-defined and have been already chosen as pivots and have an index in the new data space. On the other hand, the k+1:n part is the implicit identity which is not chosen yet nor has a name in the new data space. The sparse triangular solve has to cope with this data duality so as the results to be consistent. The algorithm for the sparse left-looking LU is shown below[23].

```
function [L,U,pinv]=left_looking_sparse_lu(A)
```

```
n=A.n;
pinv=zeros(n,1);
Lp=zeros(n,1);
xi=zeros(n,1);
lnz=1;
unz=1;
for k=1:n
    [x, X]=lsparse_solve(L,A,k,pinv); % x=L\A(:,col)
    ipiv=0;
    a=-1;
    for p=1:length(X) % find the pivot
        i=X(p); % non zero value of x
        if (pinv(i)==0) % has not been chosen as pivot yet
            t=abs(x(i)); % absolute value of possible pivot
            if(t>a)
                a=t; % largest pivot candidate until now
                ipiv=i; % pivot 's row index
            end
        else
        % U(pinv(i),k)=x(i), creating the vector u12 = x1
            Ui(unz)=pinv(i);
            Ux(unz)=x(i);
            unz=unz+1;
        end
    end
    pivot=x(ipiv); % the chosen pivot
    Ui(unz)= k; % U(k,k) index
    Ux(unz)=pivot; % U(k,k) assignment
    unz=unz+1;

    % dividing by pivot to find the column of L
    for p=1:length(X)
        i=xi(p);
        if (pinv(i)==0) % x(i) is an entry in L(:,k)
            Li(lnz)=i; % row index of i
            Lx(lnz)=x(i)/pivot; % value of L divided by pivot
        end
        x(i)=0;
    end
end
Lp(n+1)=lnz; % terminate column pointers
```

```
    Up(n+1)=unz;
    for p=1:lnz % point old row indices to the new
        Li(p)=pinv(Li(p));
    end
end
```

Recall lsparse_solve from Section 4.4, some adjustments need to be made so as to use the pinv vector of transformation in its computations. Furthermore the Reach and DFS functions need to be modified from Section 4.3.2. The adjustments are shown below.

```
function [x, X]=lsparse_solve(L,B,k,pinv)
    X=cs_reach(L,B,k,pinv)
    for p=Bp(0):Bp(1)-1 % b is stored in a CSC format,
        x(Bi(p))=Bx(p);  % so it is scattered
    end
    for s=1:length(X)
        j=X(s);
        if(pinv)
            J=pinv(j);
        end
        if (J==0)
            continue;
        end
        x(j)=x(j)/Lx(Lp(J));
        for p=Gp(J)+1:Gp(J+1)-1
            x(Li(p))=x(Li(p))-Lx(p)*x(j);
        end
    end
end
```

```
function $\mathcal{X}$=reach(L,$\mathcal{B}$,pinv)
    for each i for which $b_i \neq 0$
        if node i is unmarked
            dfs(i,pinv)

function dfs(j,pinv)
    mark(j)
    jnew=pinv(j)
    for each i for which $l_{ijnew} \neq 0$
        if node(i) is unmarked
            dfs(i,pinv)
    push j onto stack for $\mathcal{X}$
```

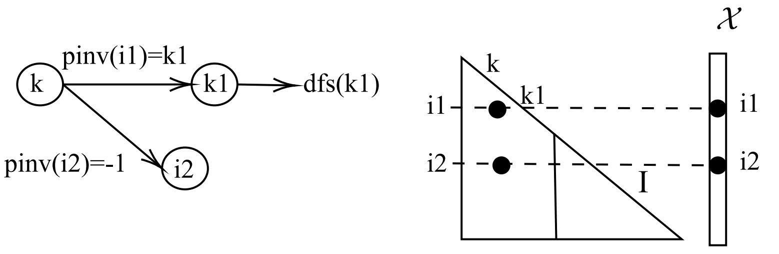

Figure 7.2: Sparse Solve Example Between New and Old Data Space

Thus, to find the reach of a node, which depends on the column pattern we look through the lens to find all the dependencies recursively. But, when it comes to storing the x solution the old indices are used so the data is consistent with the not yet pivotal entries. The marking has to happen at the row index while the new recursion at the column index.

The total left-looking algorithm for the sparse LU factorization is $O(n+|A|+f)$, where n is the dimension of the A, |A| are the nonzeros and f the number of floating-point operations performed. This is essentially *O(f)* except for some cases, when A is diagonal for instance[23].

In the Figure 7.2 above, an instance of the sparse triangular solve is shown. The column k reaches to L(i_1,k) which will be L(k_1,k) in the new data space, and to a row i_2. Since i_1 maps to k_1 in the new data, the dfs(k_1) is made to find nodes which depend on the latter while the i_1 is marked. Row i_2 maps to -1 which means that has not been yet selected as pivot. Then it has yet no dependencies, so dfs is not made to this node, still it has to be in nonzero structure of $\mathcal{X}$. Thus, in $\mathcal{X}$ the old indices are used.

An example of left-looking LU with numerical values follows.

$$A=\begin{bmatrix} 5 & 0 & 8 & 0 & 5 & 0 \\ 6 & 5 & 0 & 0 & 9 & 0 \\ 0 & 4 & 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 11 & 3 & 0 \\ 0 & 9 & 0 & 0 & 5 & 0 \\ 0 & 0 & 2 & 0 & 6 & 5 \end{bmatrix}, L=\begin{bmatrix} 1 & & & & & \\ 0 & 1 & & & & \\ 0.8333 & -0.4630 & 1 & & & \\ 0 & 0.8889 & 0 & 1 & & \\ 0 & 0 & 0.25 & 0 & 1 & \\ 0 & 0.44 & 0.6250 & 0 & -0.3484 & 1 \end{bmatrix}$$

$$U=\begin{bmatrix} 6 & 5 & 0 & 0 & 9 & 0 \\ & 9 & 0 & 0 & 5 & 0 \\ & & 8 & 0 & -0.1852 & 0 \\ & & & 11 & -1.444 & 0 \\ & & & & 6.0463 & 5 \\ & & & & & 1.7420 \end{bmatrix}, P=\begin{bmatrix} 2 & 5 & 1 & 4 & 6 & 3 \end{bmatrix}$$

Notice that all the entries in L are less than 1 magnitude thanks to partial pivoting. P matrix shows the permutations applied to A in order to do the partial pivoting, so L*U=A(P,:). This means that if this row permutation has known before hand(it is computed on the fly) and the matrix PA was factorized, P'=I, L'=L and U'=U. No row permutation would have been made because it is already in the optimal order.

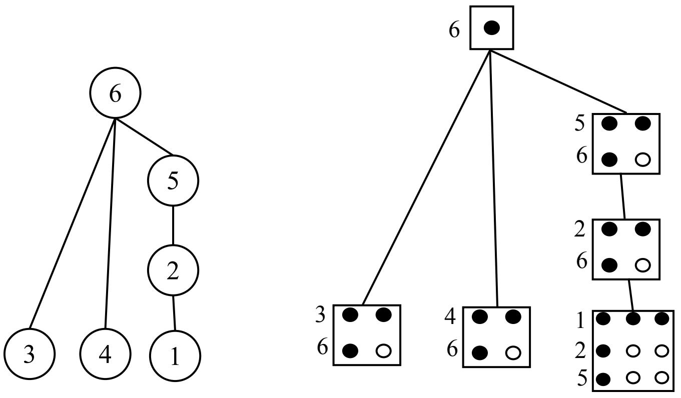

Figure 7.3: Multifrontal Elimination Tree and Frontal Matrices

7.7 Multifrontal Method

The LU factorization can be used in a multifrontal way. The logic behind Multifrontal LU is the same as the Multifrontal Cholesky inspected in Section 5.13. If an unsymmetric matrix with a symmetrical nonzero pattern is factorized with LU that results in square frontal matrices like Cholesky, only this time the first row/column of each frontal matrix gives the U/L pattern respectively[23]. The basis of this method has been presented in [31][30].

Recall again the matrix from Section 5.13, $A = \begin{bmatrix} a_{11} & a_{12} & & & a_{15} & \\ a_{21} & a_{22} & & & a_{25} & \\ & & a_{33} & & & a_{36} \\ & & & a_{44} & & a_{46} \\ a_{51} & a_{52} & & & a_{55} & a_{56} \\ & & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix}$, and corresponding elimination tree and frontal matrices shown in Figure 7.3. Its LU factors are:

$$L = \begin{bmatrix} 1 & & & & & \\ \ell_{21} & 1 & & & & \\ & & 1 & & & \\ & & & 1 & & \\ \ell_{51} & \ell_{52} & & & 1 & \\ & & \ell_{63} & \ell_{64} & \ell_{65} & 1 \end{bmatrix}, U = \begin{bmatrix} u_{11} & u_{12} & & & u_{15} & \\ & u_{22} & & & u_{25} & \\ & & u_{33} & & & u_{36} \\ & & & u_{44} & & u_{46} \\ & & & & u_{55} & u_{65} \\ & & & & & u_{66} \end{bmatrix}.$$

The multifrontal method proceeds as follows:

$\overline{U}_3 = 0, \overline{U}_4 = 0, \overline{U}_1 = 0$ since they are leaves of the tree and have no children.

$$F_3 = \begin{bmatrix} a_{33} & a_{36} \\ a_{63} & \end{bmatrix}, U_3 = -\begin{bmatrix} \ell_{6,3}a_{3,6} \end{bmatrix} = -\begin{bmatrix} \ell_{63}u_{36} \end{bmatrix}$$

$$F_4 = \begin{bmatrix} a_{44} & a_{46} \\ a_{64} & \end{bmatrix}, U_4 = -\begin{bmatrix} \ell_{64}a_{46} \end{bmatrix} = -\begin{bmatrix} \ell_{64}u_{46} \end{bmatrix}$$

$$F_1 = \begin{bmatrix} a_{11} & a_{12} & a_{15} \\ a_{21} & & \\ a_{51} & & \end{bmatrix}, U_1 = -\begin{bmatrix} \ell_{21}a_{12} & \ell_{21}a_{15} \\ \ell_{51}a_{21} & \ell_{51}a_{15} \end{bmatrix}$$

$$F_2 = \begin{bmatrix} a_{22} & a_{25} \\ a_{52} & \end{bmatrix} \oplus U_1 = \begin{bmatrix} a_{22} & a_{25} \\ a_{52} & \end{bmatrix} - \begin{bmatrix} \ell_{21}a_{12} & \ell_{21}a_{15} \\ \ell_{51}a_{21} & \ell_{51}a_{15} \end{bmatrix} = \begin{bmatrix} a_{22} - \ell_{21}a_{12} & a_{25} - \ell_{21}a_{51} \\ a_{52} - \ell_{21}\ell_{51} & -\ell_{51}a_{15} \end{bmatrix}$$
$$= \begin{bmatrix} u_{22} & u_{25} \\ u_{52} & -\ell_{51}a_{15} \end{bmatrix}, U_2 = \begin{bmatrix} -\ell_{51}a_{15} - \ell_{25}u_{25} \end{bmatrix}$$
$$F_5 = \begin{bmatrix} a_{55} & a_{56} \\ a_{65} & \end{bmatrix} \oplus U_2 = \begin{bmatrix} a_{55} & a_{56} \\ a_{65} & \end{bmatrix} - \begin{bmatrix} -\ell_{51}a_{15} - \ell_{25}u_{25} & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} a_{55} - \ell_{51}^2 - \ell_{52}a_{15} - \ell_{25}u_{25} & a_{56} \\ a_{65} & 0 \end{bmatrix}$$
$$= \begin{bmatrix} u_{55} & u_{56} \\ u_{65} & 0 \end{bmatrix}, U_5 = - \begin{bmatrix} \ell_{65}u_{65} \end{bmatrix}$$
$$F_6 = \begin{bmatrix} a_{66} \end{bmatrix} \oplus U_5 \oplus U_4 \oplus U_3 = \begin{bmatrix} a_{66} \end{bmatrix} - \begin{bmatrix} \ell_{65}u_{65} \end{bmatrix} - \begin{bmatrix} \ell_{64}u_{46} \end{bmatrix} - \begin{bmatrix} \ell_{63}u_{36} \end{bmatrix} = \begin{bmatrix} a_{66} - \ell_{65}u_{65} - \ell_{64}u_{46} - \ell_{63}u_{36} \end{bmatrix}$$

The LU multifrontal method has been extended for matrices with unsymmetric pattern[31]. If A is an unsymmetric matrix then $M = A + A^T$ denotes a structurally symmetric matrix. The M matrix can be used for computing the elimination tree. The symmetric pattern can also used to form the frontal square matrices as well and the method is similar to the one mentioned above[4][15].

Another method is to assemble **rectangular** frontal matrices. For a matrix A which is unsymmetric, the column elimination tree can be computed, this is the elimination tree of A^TA which shows the relations. The QR analysis is proved useful for this multifrontal method as in each step k, the size of the frontal matrix F_k is bounded by the size of V_k and R_{k*}, thus F_j is at most V_k-by-R_{k*}[21][23].

An example follows of an unsymmetric matrix A. Its QR factorization is shown so as to see the upper bounds, as well as the column elimination tree and the amalgamated column elimination tree, which results by grouping together parent nodes with same pattern.

$$A = \begin{bmatrix} * & * & & & & * & \\ * & * & & & & * & * \\ & & * & & * & * & \\ * & & & * & * & & \\ & & & * & * & * & * \\ & & & & * & * & \\ & & & & * & * & * \end{bmatrix} \qquad [QR] = \begin{bmatrix} * & * & & * & * & * & * \\ * & * & & * & * & * & * \\ & & * & & * & * & \\ * & * & & * & * & * & * \\ & & & * & * & * & * \\ & & & & * & * & * \\ & & & & * & * & * \end{bmatrix}$$

Recall that QR performs the set-union of the rows affected by Householder and it is in a sense like choosing all candidate pivot rows, as partial pivots thus providing the upper bound. Below LU factors of A are shown, $\circledast$, $\circ$ denote fill-in with no partial pivoting and possible fill-in from partial pivoting respectively.

$$[LU] = \begin{bmatrix} * & * & & \circ & \circ & * & \circ \\ * & * & & \circ & \circ & * & * \\ & & * & & * & * & \\ * & \circledast & & * & * & \circledast & \circ \\ & & & * & * & * & * \\ & & & & * & * & \circledast \\ & & & & * & * & * \end{bmatrix}$$

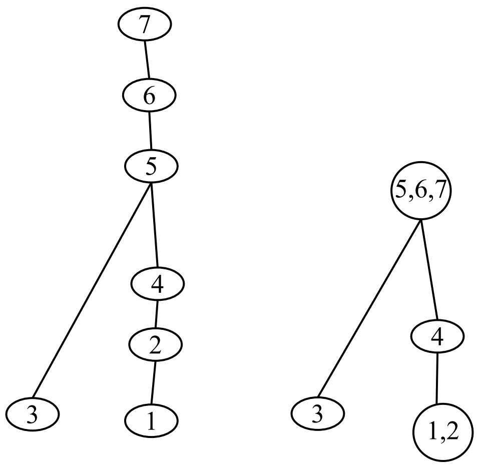

Figure 7.4: Column Elimination Tree and And Amalgamated Column Elimination Tree

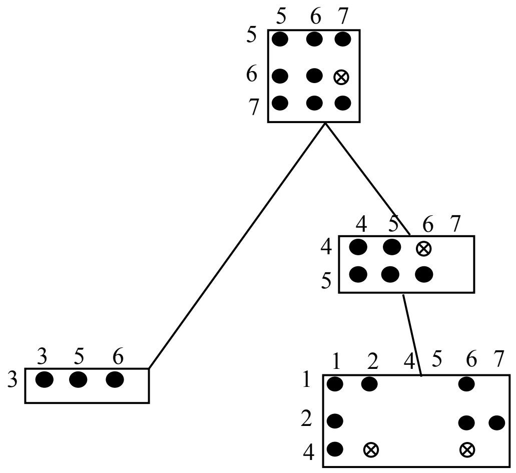

Figure 7.5: Frontal Matrices Relation

In the two Figures above the two elimination trees and the relation of the frontal matrices is shown. In the frontal matrices, the fill-in with no partial pivots that the elimination will produce are shown for convenience.

This was the last factorization method, discussed. In the next chapter, some methods to reduce fill-in entries will be inspected.

Chapter 8

Fill-in Reduction Methods

8.1 Introduction

In the algorithms inspected up until now, it was evident that fill-in entries increase in a slightly big amount the total work that is needed to be done to compute a matrix factorization. It was also shown that the way a matrix was ordered made a huge impact on the propagation of fill-in entries. Thus, it is only natural that some methods have been developed in order to find a more suitable matrix ordering so as to reduce fill-in. However, actually finding the optimal the order that minimizes the fill-in is an **NP-Complete** problem, so **heuristics** are used to find a proper result[66].

8.2 Minimum Degree Ordering

If the permutation chosen reduces the fill-in entries in Cholesky factorization, then, QR factorization will also have reduced fill-in due to $R^T = L$ and thus, the upper-bound of LU will also be reduced. Therefore, a permutation that maintains the symmetry and positive-definiteness of the matrix A must be found. Consider the following system of equations:

$Ax = b, A \in \Re^{n\ x\ n}, A^T = A, x^TAx > 0, x \in \Re^n$. Assume P is a permutation matrix then:

$(PAP^T)(Px) = Pb$, where $P^TP = I$.

Thus, factorizing PAP^T into LL^T will yield a system with less fill-in.

The **Minimum Degree** algorithm is one of the most used and effective algorithms to find such a matrix P. In a general sense, the logic of this algorithm mimics the partial pivoting in symmetric Gaussian Elimination. Viewed in a right-looking Cholesky manner of Section 5.13, at step k the matrix $A^{(k)}$ is to be updated with the outer product of the $L(:,k) * L(:,k)$. So at each step, the row and corresponding column with the less entries is chosen as pivot row. After n steps, the sequence in which the pivots where chosen corresponds to the suitable ordering.

Assume that $A^{(k)}$ is matrix A at the kth step of the elimination and $G^{(k)} = (V, E)$ the corresponding undirected graph. The vertices and edges of $G^{(k)}$ are $V_{G^{(k)}} = \{k, k+1, \ldots, n\}$ and $E_{G^{(k)}} = \{(i,j) | a_{ij}^{(k)} \neq 0\}$. Such a graph is called an **Elimination Graph**. When the update from $L(:,k)$ to $A^{(k)}$ happens, which is like adding a dense submatrix to A and removing k row and column, in the equivalent graph it is like adding a clique in $G^{(k)}$ and removing vertex k and its corresponding edges. The clique is formed by all the neighbours of k which actually shows the fill-in entries that arise. Continuing the elimination, each time picking the vertex with the least degree, until no vertices remain to eliminate is called the *elimination game*[49].

The elimination graph at each step is reduced by a vertex and its edges, but a clique is added instead, which may contain a great amount of edges. This may result in an excessive need of space, far more than the space allocated for the original graph. Assume $m = |E|$ are the number of edges in the original graph, and $m^{+} = |E^{+}|$ are the number of edges in the filled elimination graph. The original space reserved for the graph is $O(n + m)$ but then $O(n^{-} + m^{+})$ is required[46]. This is a bad practice because dynamic memory needs to be allocated which may not be available at that time and which also can slow down the work in comparison to static memory. That is why, a data structure to do this work in bounded $O(n + m)$ space is introduced. 1

A **Quotient Graph** is a graph which creates edges *implicitly* instead of the elimination graph which creates them *explicitly*. To do this, no vertex is removed but instead it is replaced by *element* or *enode* k. The element k has $\mathcal{L}_k$ neighbours and a node i adjacent to element k means it is adjacent to every node in $\mathcal{L}_k$. Thus, every node i not yet eliminated is adjacent to the nodes from matrix A where $a_{ij} \neq 0$ defined as $\mathcal{A}_i$ and to all those nodes which have now become elements defined as $\mathcal{E}_i$[38].

$$\text{adj(i)=}\mathcal{L}_i = \left(\mathcal{A}_i \cup \bigcup_{e \in \mathcal{E}_i} \mathcal{L}_e \setminus \{i\} \right)$$

The degree of a node i degree(i)=$|adj(i)|$.

Some more techniques can be implemented to improve the algorithm. The first is called **Element Absorption** which removes any detected clique redundancy in the elimination or quotient graph. This will speed up the whole process since less overhead from manipulating the set of cliques in involved. Consider the set of cliques of a graph are $K = \{K_1, K_2, \ldots, K_n\}$ and $K_i, K_j \in K$. If $K_i \subseteq K_j$ then K can be represented as $K = \{K_1, K_2, \ldots, K_n\} - K_i$. It is as if the element j is absorbing the element i thus element absorption. Consider a graph with 6 nodes forming the following cliques:
$\{1,5\}\{2,5,6\}\{3,5,6\}\{4,5,6\}$. Assuming that the elimination order of minimum degree is : 1,2,3 then the resulting cliques $\{5\}, \{5,6\}, \{5,6\}$ can be represented by $\{4,5,6\}$. Thus, all cliques are absorbed into one in the end[36].

Another technique to improve the performance of the algorithm is **pruning**. If two entries i,j are in the pivotal element $\mathcal{L}_k$ then j and i can be removed from $\mathcal{A}_i$ and $\mathcal{A}_j$ respectively. That is because j and i may have been adjacent by an element a_{ji}, a_{ij} due to symmetry, but now they adjacency is represented by the element k[23][3].

Assume now, that two nodes have the same adjacency list if in each one itself is included. That means, $adj(i) \cup i = adj(j) \cup j$. These two nodes are *indistinguishable* and obviously have the same degree. If one of the two is selected as the next pivot vertex, then the other one can be chosen without introducing any new fill-in entries. Thus, those two elements can be chosen together as pivotal elements. The logic behind it is the same manner, as performing **supernodes**. Eliminating two or more nodes similar nodes is called **mass elimination**[1]. One of these nodes can be chosen as the representative of the supernode containing all of them. This technique reduces the number of times the adjacency list of each node needs to be evaluated.

Since mass elimination is used, the size of the resulting clique may be different from when only one node was used. The size of the clique as calculated until now by the adjacency of a node may be referred as the *true degree* of that node. Now, for mass elimination the **external degree** of a node is implemented which denotes the number of nodes that are *distinguishable* from it[53].

In practice two are the main minimum degree algorithms which are improved versions of the basic minimum degree algorithm.

1. Multiple Minimum Degree
2. Approximate Minimum Degree

The first algorithm tries to reduce the number of degree updates of the nodes. When a search is made for the minimum degree vertex, some ties may result. That means multiple vertices may have the minimum degree. This algorithm finds the independent of each other minimum degree vertices and eliminates them. Since the nodes are independent the elimination of one will not affect the others. After all the independent nodes have been eliminated, there comes a degree update of all the nodes[53].

The second algorithm does not try to reduce the number of degree updates of the nodes, unlike the first, but tries to reduce the computational cost of the degree-update. This is done by computing an upper bound to the degrees, instead of the actual degree, referred as approximate degree. Let the approximate degree of node i be[23]:

$$\overline{d}_i = |\mathcal{A}_i| + |\mathcal{L}_k\{i\}| + \sum_{e\in\mathcal{E}_i\setminus\{k\}} |\mathcal{L}_e \setminus \mathcal{L}_k|$$

where k is the current pivot element and $\mathcal{A}_i$ is already pruned. In other words, the degree of a vertex i cannot be larger than the sum of the degrees of the adjacent nodes and elements to it.

The second algorithm produces almost the same fill-in to the first algorithm but is typically faster[3]. The algorithm template of each of these algorithms can be found in this paper[46].

An example of the basic minimum degree algorithm is shown below. Consider the following matrix:

$$A = \begin{bmatrix} * & * & & & & * & & \\ * & * & * & & & * & & \\ & * & * & & & & & * \\ & & & * & * & * & & \\ & & & * & * & & & * \\ * & * & & * & & * & * & \\ & & & & & * & * & * \\ & & * & & * & & * & * \end{bmatrix} \quad A^{(1)} = \left[\begin{array}{c|ccccccc} \circ & \circ & & & & \circ & & \\ \hline \circ & * & * & & & * & & \\ & * & * & & & & & * \\ & & & * & * & * & & \\ & & & * & * & & & * \\ \circ & * & & * & & * & * & \\ & & & & & * & * & * \\ & & * & & * & & * & * \end{array}\right]$$

$$A^{(2)} = \left[\begin{array}{cc|cccccc}
\circ & \circ & & & & \circ & & \\
\circ & \circ & \circ & & & \circ & & \\
\hline
 & \circ & \ast & & & \circledast & & \ast \\
 & & & \ast & \ast & \ast & & \\
 & & & \ast & \ast & & & \ast \\
\circ & \circ & \circledast & \ast & & \ast & \ast & \\
 & & & & & \ast & \ast & \ast \\
 & & \ast & & \ast & & \ast & \ast
\end{array}\right]
\quad
A^{(3)} = \left[\begin{array}{ccc|ccccc}
\circ & \circ & & & & \circ & & \\
\circ & \circ & \circ & & & \circ & & \\
 & \circ & \circ & & & \circ & & \circ \\
\hline
 & & & \ast & \ast & \ast & & \\
 & & & \ast & \ast & & & \ast \\
\circ & \circ & \circ & \ast & & \ast & \ast & \circledast \\
 & & & & & \ast & \ast & \ast \\
 & & \circ & & \ast & \circledast & \ast & \ast
\end{array}\right]$$

$$A^{(4)} = \left[\begin{array}{cccc|cccc}
\circ & \circ & & & & \circ & & \\
\circ & \circ & \circ & & & \circ & & \\
 & \circ & \circ & & & \circ & & \circ \\
 & & & \circ & \circ & \circ & & \\
\hline
 & & & \circ & \ast & \circledast & & \ast \\
\circ & \circ & \circ & \circ & \circledast & \ast & \ast & \circledast \\
 & & & & & \ast & \ast & \ast \\
 & & \circ & & \ast & \circledast & \ast & \ast
\end{array}\right]
\quad
A^{(5)} = \left[\begin{array}{ccccc|ccc}
\circ & \circ & & & & \circ & & \\
\circ & \circ & \circ & & & \circ & & \\
 & \circ & \circ & & & \circ & & \circ \\
 & & & \circ & \circ & \circ & & \\
 & & & \circ & \circ & \circ & & \circ \\
\hline
\circ & \circ & \circ & \circ & \circ & \ast & \ast & \circledast \\
 & & & & & \ast & \ast & \ast \\
 & & \circ & & \circ & \circledast & \ast & \ast
\end{array}\right]$$

$$A^{(6)} = \left[\begin{array}{cccccc|cc}
\circ & \circ & & & & \circ & & \\
\circ & \circ & \circ & & & \circ & & \\
 & \circ & \circ & & & \circ & & \circ \\
 & & & \circ & \circ & \circ & & \\
 & & & \circ & \circ & \circ & & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
\hline
 & & & & & \circ & \ast & \ast \\
 & & \circ & & \circ & \circ & \ast & \ast
\end{array}\right]
\quad
A^{(7)} = \left[\begin{array}{ccccccc|c}
\circ & \circ & & & & \circ & & \\
\circ & \circ & \circ & & & \circ & & \\
 & \circ & \circ & & & \circ & & \circ \\
 & & & \circ & \circ & \circ & & \\
 & & & \circ & \circ & \circ & & \circ \\
\circ & \circ & \circ & \circ & \circ & \circ & \circ & \circ \\
 & & & & & \circ & \circ & \circ \\
\hline
 & & \circ & & \circ & \circ & \circ & \ast
\end{array}\right]$$

In this example the natural order is the ideal for the minimum degree algorithm. Each matrix shows one step of the elimination and the corresponding elimination and quotient graphs are shown in the Figure 8.1. In the matrix the circled star denotes a fill-in entry and so a new edge in the elimination graph, white circles are the deleted entries so far.

Note in the quotient graphs that in G^1, due to 2,6 being connected via the element 1, their intervening edge is omitted (pruning). Also, in G^3 and G^5 quotient graphs the element 2 is absorbed by 3 and the elements 3,4 are absorbed by 5 respectively (element absorption). Furthermore, in the G^5 quotient graph, the 6,7 edges are indistinguishable so they form a supernode and are eliminated together (mass elimination).

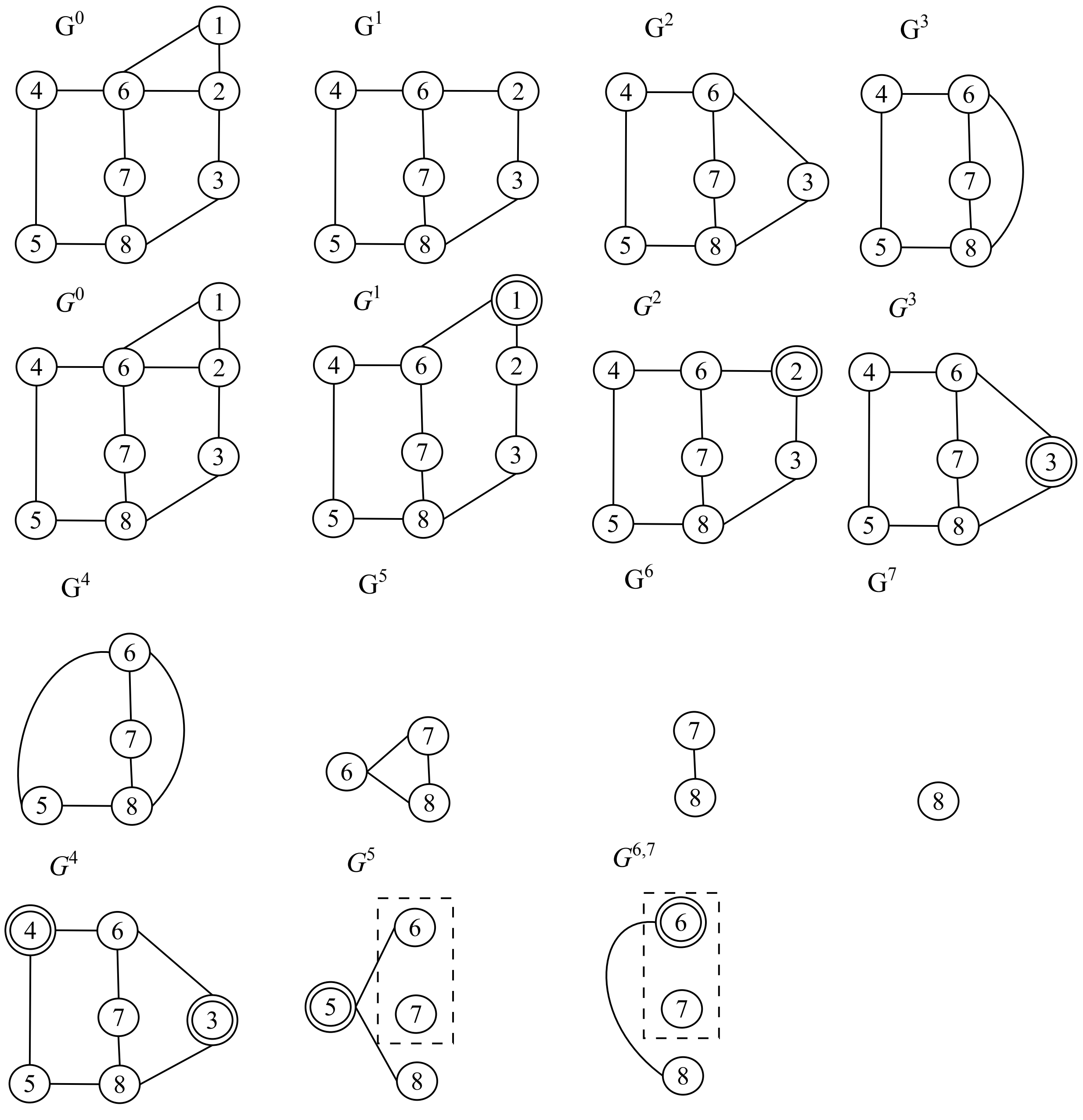

Figure 8.1: Elimination and Quotient Graphs

8.3 Maximum Matching

It is a very common phenomena that a matrix A of order n if permuted properly, can yield a zero-free diagonal matrix. Finding such a permutation is the equivalent of finding a *maximum cardinality matching* in a *bipartite graph*[28].

An undirected graph G=(V,E) in which V is split into r classes, so that every edge $e \in E$ has its ends in different classes is called *r-partite*. Thus, for 2 classes the result is a **bipartite graph**. A bipartite graph contains no cycle of odd length. In bipartite graph a subset of edges $\mathcal{M}$ is called a **matching** if any vertex is incident on at most one edge in $\mathcal{M}$[14]. A **maximum** matching is a matching of maximum cardinality, in case of the matrix A the cardinality is n if it is rank efficient. That means that for any other matching $\mathcal{M}'$, $|\mathcal{M}| \geq |\mathcal{M}'|$. An *alternating* path in $\mathcal{M}$ is a path whose edges alternate between those in $\mathcal{M}$ and those not in $\mathcal{M}$. A $\mathcal{M}$-*augmenting* path $\mathcal{P}$ is a $\mathcal{M}$-alternating path whose starting and ending nodes are unmatched[24]. If G has no $\mathcal{M}$-*augmenting* path then $\mathcal{M}$ is of maximum cardinality[8].

The basic concept of the **augmenting paths** algorithms is given a matching $\mathcal{M}$, search for an $\mathcal{M}$-*augmenting* path. If none is found, then $\mathcal{M}$ is of maximum cardinality and the algorithm stops. Otherwise, if a path is found, then $\mathcal{M}$ is increased by the edges of the path not in $\mathcal{M}$.

A sparse matrix A forms a bipartite graph $G_A = (V_\mathcal{R} \cup V_\mathcal{C}, E)$ where, $V_\mathcal{R}$ and $V_\mathcal{C}$ denote the rows and columns of A respectively. Therefore, for an entry $a_{ij} \neq 0$ with $i \in \mathcal{R}$ and $j \in \mathcal{C}$ there is an edge in E. For $a_{i'j'} = 0$ there is no edge in the graph. If A is square n-by-n and $\mathcal{M}$ a perfect matching, then a permutation matrix Q can be found with $Q_{ji} = 1 \iff (i,j) \in \mathcal{M}$ so that AQ is the zero-free diagonal matrix, the same holds for a permutation matrix P and PA. If A is rectangular m-by-n or has a cardinality $l < n$ two permutation matrices P,Q can be found which permute the matched rows and columns to the first l positions, thus the l-by-l submatrix of PAQ can be have a zero-free diagonal[28].

There are two basic algorithmic methods to find the augmenting paths. One is based on depth-first search while the other is based on breadth-first search. Here a DFS algorithm is going to be shown, which is used in MATLAB, a full presentation of both DFS and BFS algorithms can be found in this CERFACS report[28].

The algorithm starts with an empty matching. If a match is made between a row $i \in \mathcal{R}$ and a column $j \in \mathcal{C}$ then the row i is noted as matched to column j, or else is noted as unmatched. In each iteration a column k not yet matched, not in $\mathcal{C}$, is picked and an *alternating augmenting path* is found so as to extend the matching. The path starts from k and then traverses to any edge to a row $i_1 \in \mathcal{A}_k$. If i_1 is unmatched the path stops, if not, it traverses to $j_1 = match(i_1)$ and then it traverses any edge to a row $i_2 \in \mathcal{A}_{j_1}$ and so forth until an unmatched row i_k is found. Then the matching is updated to include the column k and the row i_k.

A heuristic called cheap, is implemented to reduce the average time complexity of the algorithm. Cheap is actually, a pointer which divides the nonzero set of a column j, $\mathcal{A}_j$, into matched and unmatched rows. Thus, when considering a column j, only the second part needs to be considered. Therefore, the DFS without the heuristic needed time $O(n|A|)$ because the whole graph was searched at every step, but now since each edge is traversed only once, the time complexity reduced $O(|A|)$, while the average time is greatly reduced in practice[23].

An example follows for better comprehension of the algorithm. Consider a matrix

$$A = \begin{bmatrix} * & & & & & \\ * & & & & * & \\ & * & * & * & & \\ & & & * & & \\ & & * & & & * \\ & * & & & * & \end{bmatrix}$$

j=1 $\mathcal{A}_1 = \{1,2\} \implies$ 1→1 match(1)=1
j=2 $\mathcal{A}_2 = \{3,6\} \implies$ 2→3 match(3)=2
j=3 $\mathcal{A}_3 = \{3,5\} \implies$ 3→5 match(5)=3
j=4 $\mathcal{A}_4 = \{3,4\} \implies$ 4→4 match(4)=4
j=5 $\mathcal{A}_5 = \{2,6\} \implies$ 5→2 match(2)=5
j=6 $\mathcal{A}_6 = \{5\} \implies$ 6→5⇒3→3⇒2→6 $\implies$ match(6)=2 ,match(3)=3 , match(5)=6
match=(1 ,5 ,3 ,4 ,6 ,2)

From the match matrix the permutation vector p can be formed, where p=(1,6,3,4,2,5) and then permuting the matrix PA=A(p,:) results in a zero free diagonal matrix $PA = \begin{bmatrix} * & & & & & \\ & * & & & * & \\ & * & * & * & & \\ & & & * & & \\ * & & & & * & \\ & & * & & & * \end{bmatrix}$

In the j=6 case the alternating augmenting path can be shown clearly, as the path alternates between edges where every other edge is matched and then the matching is extended when the unmatched row 6 is found.

8.4 Block Triangular Form

The **Block Triangular Form**(BTF) of a sparse matrix leads to reduced computational time, as well as reduced storage space for many algorithms, such as the LU and QR factorizations. The **Strong Hall** property of a matrix produces strict bounds on the nonzero pattern of its factors. A matrix A may have full rank but may not be Strong Hall, however, it can be permuted into (BTF),

$$PAQ = \begin{bmatrix} A_{11} & \dots & A_{1k} \\ & \ddots & \vdots \\ & & A_{kk} \end{bmatrix}$$

where each diagonal block has the strong Hall property[59].

Consider a system Ax=b where matrix A permuted into an upper block triangular form

$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$, expanding the systems leads to the following equations:

$$A_{33}x_3 = b_3 \implies L_{33}U_{33}x_3 = b_3$$

$$A_{22}x_2 + A_{23}x_3 = b_2 \implies L_{22}U_{22}x_2 = b_2 - A_{23}x_3 \implies L_{22}U_{22}x_2 = b_2'$$

$$A_{11}x_1 + A_{12}x_2 + A_{13}x_3 = b_1 \implies L_{11}U_{11}x_1 = b_1 - A_{12}x_2 - A_{13}x_3 \implies L_{11}U_{11}x_1 = b_1'$$

Only the diagonal blocks need to be factorized to solve for a solution, all the other blocks are used just for matrix-vector multiplication, which is more convenient than factorizing the whole matrix.

Factorizing only the diagonal block, can help reduce fill-in too. Consider the following case where A is partitioned as before, where $A_{11} = \begin{bmatrix} * & * \\ * & * \end{bmatrix}$ and $A_{12} = A_{13} = \begin{bmatrix} * & * \\ 0 & 0 \end{bmatrix}$. If only A_{11} is factorized there will be not fill-in caused due to elimination of the first column. On the contrary, if A has factorized not in block form all the nonzeros from the first line would be propagated to the second line, thus causing 4 fill-in entries.

Given an arbitrary sparse matrix, before permuting the matrix into a block triangular form, it must be permuted to have a zero-free diagonal. So, the maximum matching algorithm mentioned in the previous section is used for that purpose. Then symmetric permutations are used to obtain the BTF[27].

To form the BTF of a matrix is the same as finding the **strongly connected components**(SCC) of a directed graph. The graph G=(V,E) formed by A, has V=$\{1 \dots n\}$ and E=$\{(i,j)|a_{ij} \neq 0\}$. A strongly connected component is a maximal subgraph in which there is a path between any pair of nodes. A strongly connected component cannot be enlarged by adding extra nodes to it because a SCC is defined as maximal, thus it's a contradiction. Each node can belong to one SCC solely, which may be itself.

If all the strongly connected components $\{C_1, C_2, \dots, C_n\}$ are formed so that the first C_1 has no path to any of the remaining nodes $\{C2, \dots, C_n\}$, then C_2 is picked so that it has no path to the rest of the nodes $\{C_3, \dots, C_n\}$ and so forth, that results in a lower triangular block matrix, where the blocks are the strong components. If the nodes are picked so that each component has no affiliation with its previous components, that results in a upper triangular block matrix.

One simple method to compute the SCC of a matrix, is to perform a topological DFS on the matrix A, like the one mentioned in Section 4.3.2 and get the resulting stack $\mathcal{X}$. Then perform again a DFS in the transposed matrix A^T but in the reverse order of their finishing time in $\mathcal{X}$, meaning starting from the last node in $\mathcal{X}$ and advance towards the first. All the nodes reachable from each node in the second DFS form a SCC[14].

Another implementation by Tarjan widely used is shown here[65][29]. The algorithm mentioned here needs $O(V+E)$ time for each DFS and for reversing the graph, while Tarjan algorithm needs just one DFS.

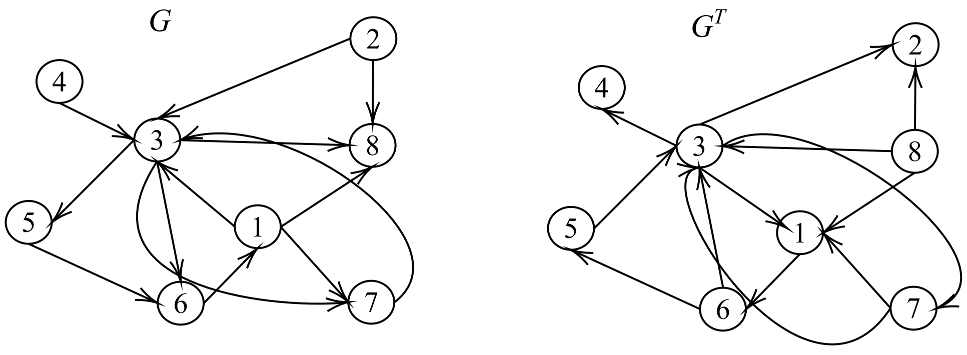

Figure 8.2: Directed graph of A and A^T

An example follows of finding the SCC of a sparse matrix
$$A^T = \begin{bmatrix} * & & * & & & & * & * \\ & * & * & & & & & * \\ & & * & & * & * & * & * \\ & & * & * & & & & \\ & & & & * & * & & \\ * & & & & & * & & \\ & & * & & & & * & \\ & & & & & & & * \end{bmatrix}$$

where its graph and transposed graph are shown in Figure 8.2.
Doing a topological DFS in G, results in $\mathcal{X} = \{4, 2, 1, 3, 8, 7, 5, 6\}$. Then performing a DFS in G^T starting from node the end of $\mathcal{X}$ leads to:

DFS(4)=$\{4\}$
DFS(2)=$\{2\}$
DFS(1)=$\{7, 3, 5, 6, 1\}$
DFS(8)=$\{8\}$

Thus, this matrix has 4 strongly connected components.

If the nodes inside the components are placed in natural order, as well as the blocks themselves, then the resulting vector is the permutation vector which permutes A into BTF. In this example, $p = \{2, 4, 1, 3, 5, 6, 7, 8\}$.

Chapter 9

Experiments

9.1 Introduction

In this chapter some experimental cases are going to be demonstrated which make use of all the theory mentioned so far. The experiments were done in *MATLAB*, using its built-in functions[56]. *SuiteSparse* which is a suite for sparse matrix algorithms, has a *MATLAB* interface, which is used in the experiments as well[20][23].

The largest amount of matrices used in experiments are from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection)[50]. There are some cases, where the matrices used are random generated matrices.

For the experiments' execution a Virtual Machine which features 64GB RAM and 16 Computational Cores is used, which set up in the University of Thessaly.

9.2 Data Structures

In this section, the differences between storing all entries of a spares matrix, as if it was full, and storing it in CCS format will be demonstrated.

Each entry is a **double** precision element, using 8-bytes of memory. For n-by-n square matrices, a dense matrix needs n^2*8 bytes, whereas, CCS needs only (2*nnz+n+1)*8 bytes.

In this experiment some random square matrices were allocated and the results is shown in the table below. The columns from left to right denote, the matrix's dimension, the nonzero elements, memory requirements for sparse storage in megabytes, memory requirements for dense storage in megabytes, the difference between dense and sparse storage in megabytes.

As it can be seen from the table exploiting sparsity when storing a matrix makes a huge impact in regards to storage performance. For an 8GB RAM machine, the maximum dimension for a square matrix is 31622, $\frac{31622^2 * 8}{10^9} = 7.9996(GB)$. Thus a 31623-by-31623 identity matrix cannot be stored in dense format. However, this matrix has only 31623 non zero elements(diagonal), therefore, storing it in CCS format only takes $\frac{2 * 31623 + 31623 + 1}{10^6} = 0.7590(MB)$.
So, the latter format allows the computer to store high dimension data, whereas the former would have failed.

n	nnz	CCS(MB)	TRIPLET(MB)	DIFF(MB)
100	3290	0.053448	0.08	0.026552
200	13134	0.21175	0.32	0.10825
400	52708	0.84654	1.28	0.43346
800	2.11E+09	33.875	5.12	17.325
1600	8.44E+09	13.515	20.48	69.654
3200	3.38E+10	54.027	81.92	27.893
6400	1.35E+11	216.08	327.68	111.6
12800	5.40E+11	864.34	1310.7	446.38

9.3 Cholesky Factorization

In this section test cases for Choleksy Factorization are performed. The matrices this time are downloaded from SuiteSparse Matrix Collection, corresponding to data from real-life systems.

The first comparison is between an optimized dense Cholesky solver with the built-in function of Matlab *chol* and the solver from *cs_sparse*, *cs_chol*, which exploits sparsity, and its based on the algorithm described in Chapter 5. The second algorithm does not make effective use of BLAS[51].

The second comparison is between the dense solve function *chol* of Matlab as well, and the *CHOLMOD* used from SuiteSparse, which is also how Matlab computes Cholesky factorization of sparse matrices. The *CHOLMOD* algorithm makes use of the BLAS as it performs dense computations. The *CHOLMOD* is based on **Supernodal** Cholesky Factorization[19]. The matrices used for this comparison are the same as above.

The third comparison is between *cs_chol* and *CHOLMOD* which both exploit sparsity.

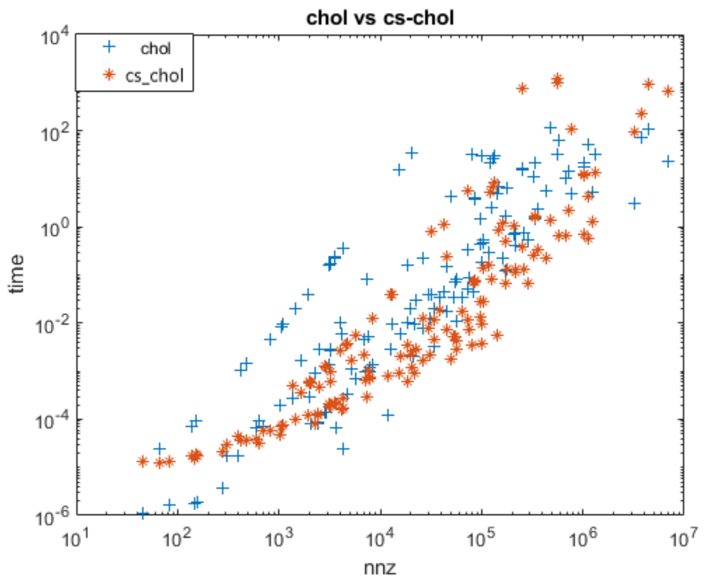

Figure 9.1: Dense Cholesky vs Sparse Choleksy

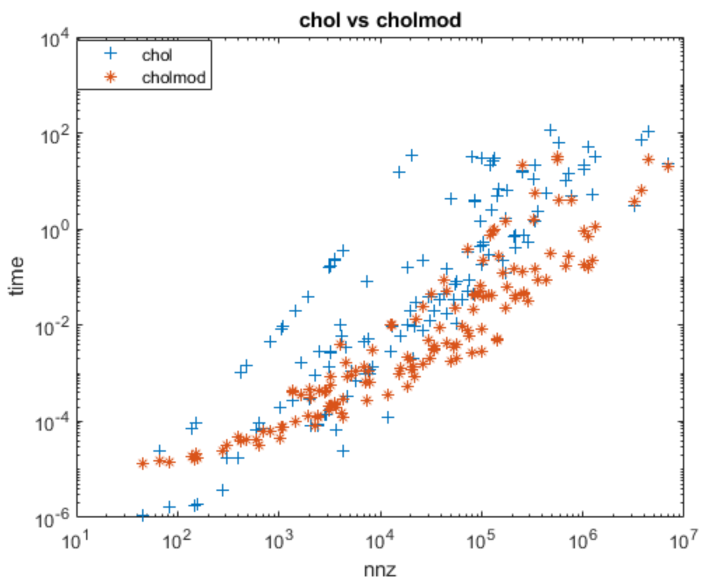

Figure 9.2: Dense Choleksy vs Supernodal Cholesky

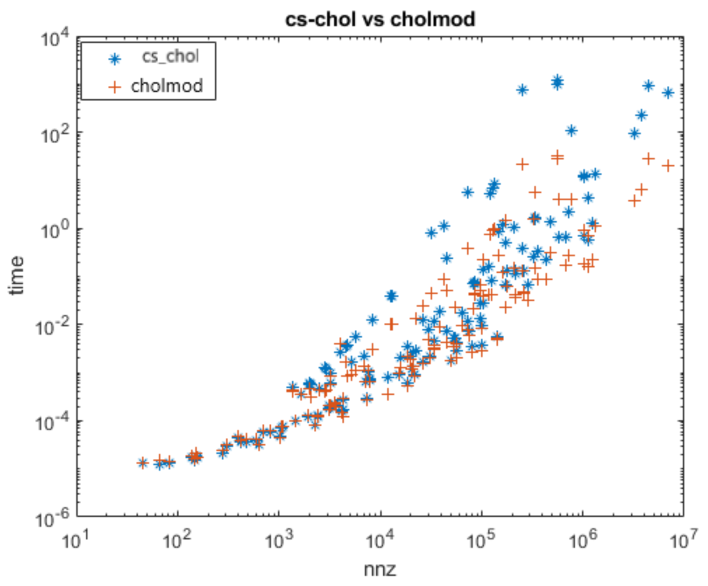

Figure 9.3: Sparse Choleksy vs Supernodal Cholesky

It is apparent that exploiting sparsity has benefits even using the not optimized solver *cs_chol*, as it is evident from the Figure 9.1. Of course, the optimized sparse solver *CHOLMOD* is faster than using the dense solver as seen in figure 9.2. In the final figure a comparison between *cs_chol* and *CHOLMOD* is made. For small matrices the results are almost identical. For large matrices though, the Supernodal algorithm is quite a lot faster. In practice it has been observed that the number of flops=$|f|$ divided by the non zeros in L=$|L|$ is a threshold in choosing between the up-looking method and the supernodal one. In particular if $\frac{|f|}{|L|} = 40$ is the crucial value, if smaller the former method is used and if larger the latter[13].

9.4 QR Factorization

In this section test cases for QR Factorization are executed. The matrices are downloaded from SuiteSparse Matrix Collection. The matrices are mainly rectangular deriving from *linear least squares* problems. Two cases are presented below.

The first case is comparing the optimized dense built-in solver in Matlab *qr* with the optimized sparse solver *SPQR* which uses a **Multifrontal** QR method. This method uses uses LAPACK and multithreaded BLAS to obtain high performance inside each frontal matrix. This method resembles the **Multifrontal** LU method described in Section 7.7, more information about this method can be found in the following paper [16].

The second case is comparing *SPQR* with the *cs_qr*, which is based on the algorithm described in Chapter 6. In this case, only matrices with more rows than columns were used because *cs_qr* is unable to handle other matrices.

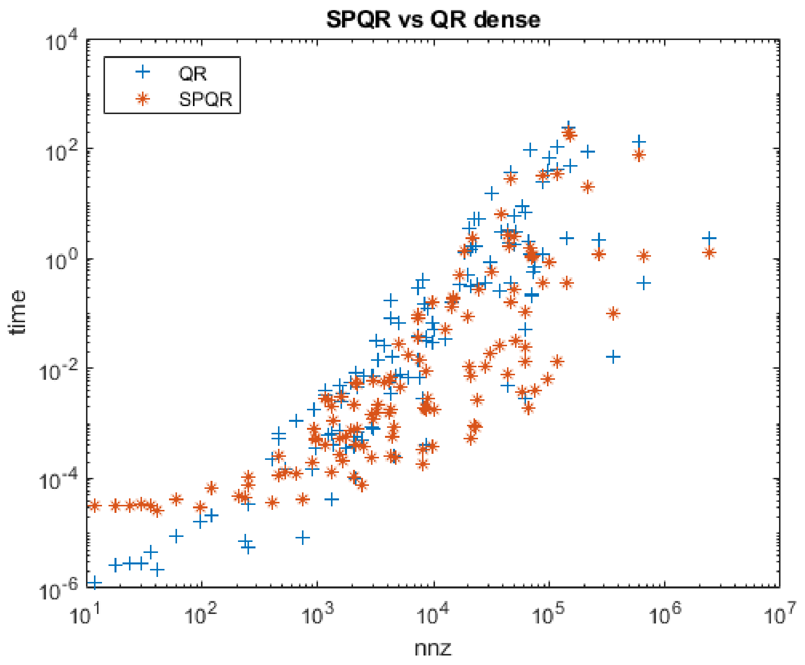

Figure 9.4: Multifrontal QR vs Dense QR

It is obvious that as the matrices grow larger, *SPQR* achieves better results than the dense solver. It important to mention that it uses less memory than the dense of course, which is important as QR tends to create a lot of fill-in, especially when no column ordering is used. If proper ordering was used, the performance from *SPQR* would be even higher. Dense algorithm would perform almost the same, with less fill-in due to all entries being checked.

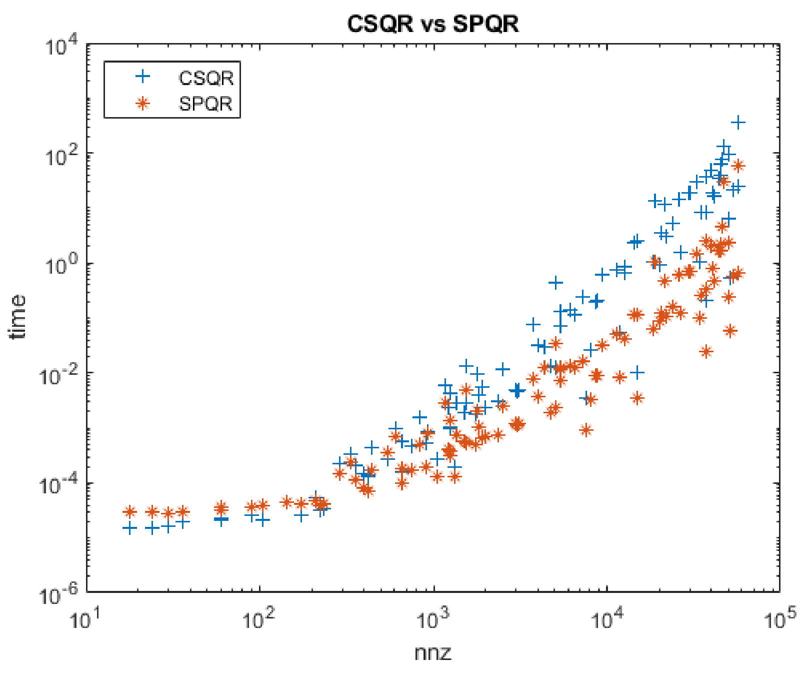

Figure 9.5: Multifrontal QR vs Sparse QR

In the second case, it can be noted that for small matrices a left sparse qr decomposition is better than using the multifrontal method. However, as the number of non zero elements increases the multifrontal method achieves better results.

9.5 LU Factorization

For the LU factorization experimental cases, matrices deriving from structural problems, network problems, 2d/3d problems and reduction models are used. These matrices are from the SuiteSparse Collection once again. Three comparisons between diffrent algorithms are made below.

The first case is comparing dense LU solver built in Matlab vs the sparse one. The sparse one is based on the algorithm discussed in Chapter 7.

The second case is a comparison between the multifrontal LU and the dense LU solver from above. The multifrontal LU solver uses *umfpack* which makes effective use of the BLAS and LAPACK subroutines[17].

The third case is a comparison between the simple sparse solver and the multifrontal one, which both exploit sparsity but the multifrontal is optimized.

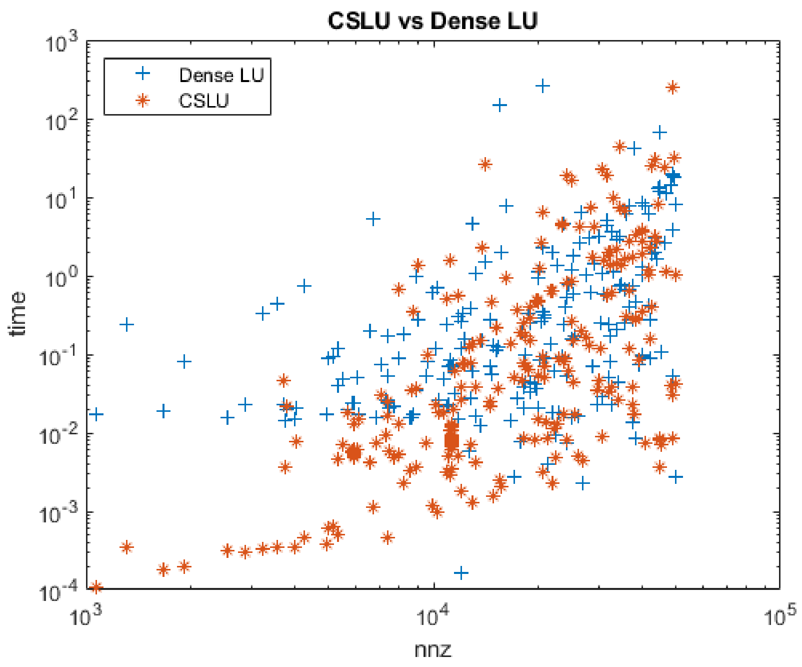

Figure 9.6: Dense LU vs Sparse LU

In the first case, the simple sparse algorithm performs a lot faster as the number of non zeros is low. As the number of nonzero increases the performance of the sparse and the dense is quite similar. In some cases, the dense algorithm performs better than the sparse one.

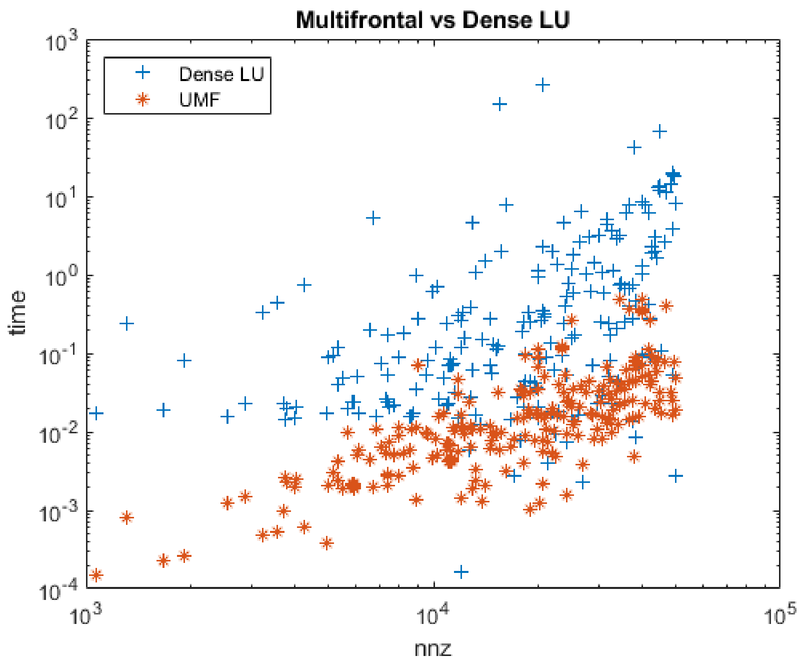

Figure 9.7: Dense LU vs Multifrontal LU

In the second case, the Multifrontal algorithm performs better in almost all scenarios. The time difference between the two algorithms is quite sizeable.

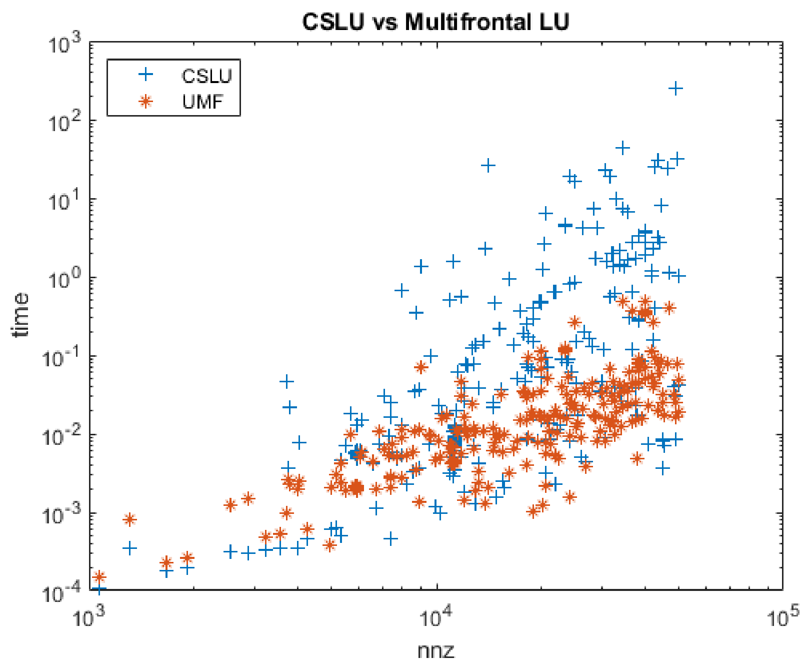

Figure 9.8: Sparse LU vs Multifrontal LU

Between the two sparse algorithms, as the matrices grow bigger the multifrontal is evidently better. For smaller matrices, the simple sparse algorithm reaches yields similar or better time measurements.

9.6 Fill-in Reduction

In this section square positive definite matrices mainly from structural problems are used. These matrices are analyzed symbolically to find the resulting non zeroes of their Cholesky factor. Then, the approximate minimum degree algorithm is used on each matrix and the symbolic analysis is performed once again.

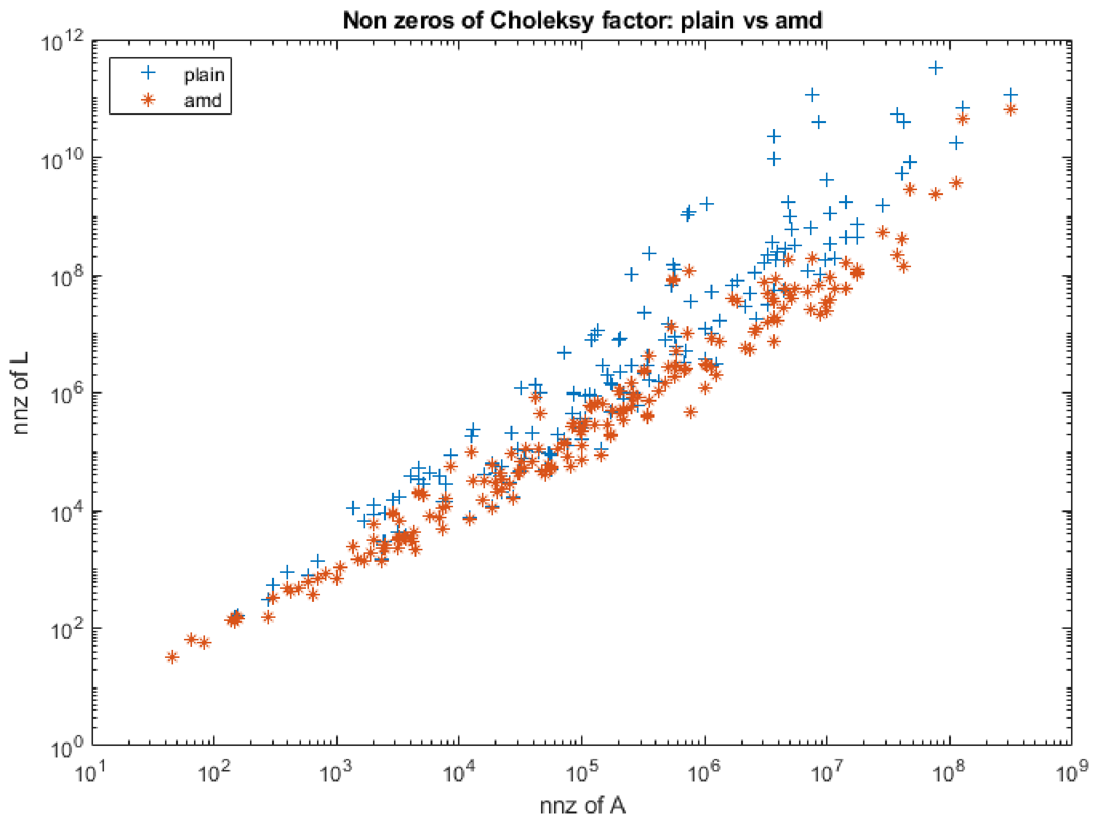

Figure 9.9: Minimum Degree Ordering

The difference between the pre-ordered factorization and the plain factorization is immense. Thus, pre-ordering a matrix will decrease a lot the memory needed to store the factor, as well as it will yield better execution times since there are less float point operations to be performed.

9.7 LU vs Cholesky

In this section a comparison is made between the supernodal Cholesky and the Multifrontal LU for factorizing positive-definite square matrices. The matrices are downloaded from SuiteSparse and are mainly power network and structural problems.

The number of entries in these matrices is considerable larger than those of used previously. Additionally, each matrix is permuted by using the approximate degree algorithm before being factorized.

As it is evident from Figure 9.10 the Cholesky factorization performs significantly better than LU for square and positive-definite matrices.

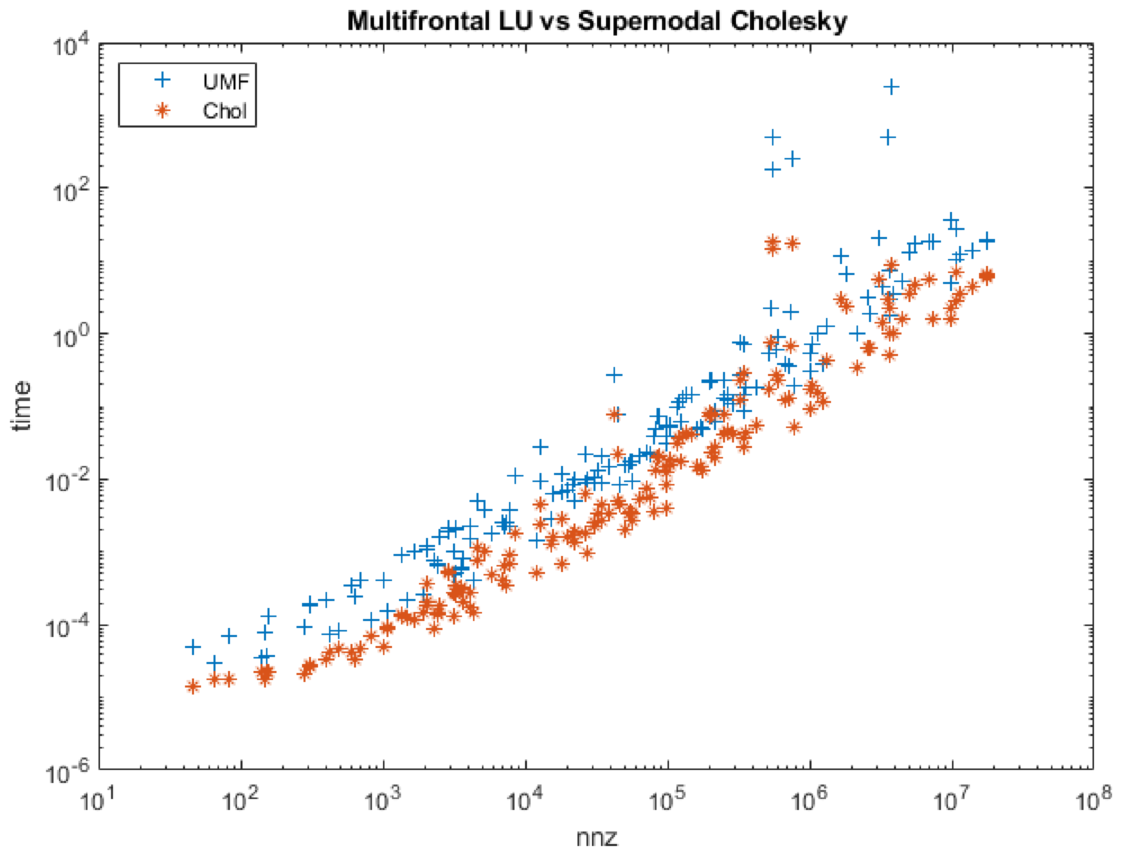

Figure 9.10: Supernodal Choleksy vs Multifrontal LU

Chapter 10

Epilogue

10.1 Conclusions

In this thesis, the benefits of exploiting the sparse structure of a matrix was shown and how certain algorithms are modified to utilize it. The theory behind Cholesky, QR and LU factorizations was demonstrated as well as the corresponding algorithms and the way they are remodeled to take advantage of the matrix structure. The resulting algorithms are the basis of the algorithms currently used by sparse solving softwares and libraries.

In particular, this thesis starts with the demonstration of data structures for sparse matrices and of common matrix algorithms. Then, how to solve a sparse lower triangular system and its relation to finding the reach of a graph. Afterwards, the Cholesky Factorization is shown as well as the symbolic analysis of a matrix, the elimination tree, the column count of the resulting matrix and how to actually, compute the the factors numerically.

Subsequently, the QR Factorization is introduced and its relation with Cholesky Factorization is shown, as well as how to compute it efficiently using the Householder Reflections. QR leads the way to LU factorization, because the former is an upper-bound to the latter. LU factorization, which is the oldest method of all, is demonstrated and numerical and symbolical trade-offs are shown, as well as methods to maintain numerical stability. Afterwards, certain methods are examined on how to construct a zero free diagonal and how to reduce the fill-in entries as well. Fill-in entries are entries that arise during the factorization of a matrix and they do not belong in the original.

In the final chapter, experimental cases are performed. A sample of matrices that arise from real life problems are used. The properties of the matrices vary in size, sparsity and shape. The sparse methods are compared to their corresponding dense ones, showing the benefits in time complexity as well as in space requirement.

10.2 Further Work

Considering the future extensions to this thesis there are a number of topics to examine. Firstly, study in depth the Supernodal and Multifrontal Algorithms and how they are implemented in computer code. Furthermore, study more fill-in reducing algorithms and their code implementation as well. Additionally, the concept of parallelization concerning the algorithms presented can be studied, which are the cases that it is beneficial to do it and how the parallelization methods are linked with dependencies already established.

Besides the Direct Methods, the Iterative Methods are another way of solving a linear system. The latter methods, use an initial guess to find an approximate solution, usually resulting in a faster solution. On top of that, they can be used to solve non-linear systems as well. This is an major topic that can be examined and compared with the methods presented in this thesis.

Bibliography

[1] George Alan and David R McIntyre. On the application of the minimum degree algorithm to finite element systems. In Ilio Galligani and Enrico Magenes, editors, *Mathematical Aspects of Finite Element Methods*, pages 122–149, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.

[2] Patrick Amestoy, Alfredo Buttari, Iain Duff, Abdou Guermouche, Jean-Yves L'Excellent, and Bora Uçar. Multifrontal Method. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1209–1216. Springer US, Boston, MA, 2011.

[3] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An Approximate Minimum Degree Ordering Algorithm. Technical Report 4, †Computer and Information Sciences Department University of Florida, 1996.

[4] Patrick R. Amestoy and Chiara Puglisi. An unsymmetrized multifrontal LU factorization. Technical report, Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA, 7 2000.

[5] Cleve Ashcraft. A vector implementation of the multifrontal method for large sparse, symmetric positive definite linear systems. In *Report ETA-TR-51*. ETA Division, Boeing Computer Services Seattle, WA, 1987.

[6] Cleve Ashcraft, Stanley C. Eisenstat, and Joseph W. H. Liu. A Fan-In Algorithm for Distributed Sparse Numerical Factorization. *SIAM Journal on Scientific and Statistical Computing*, 11(3):593–599, 5 1990.

[7] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994.

[8] Claude Berge. TWO THEOREMS IN GRAPH THEORY. Technical report, Princeton University, 1957.

[9] Jayanta Biswas. Extension of Matrix Algebra and Linear spaces of Linear Transformations. *International Journal of Novel Research in Physics Chemistry & Mathematics*, 3:29–42, 5 2016.

[10] Åke Björck. *Numerical Methods in Matrix Computations*, volume 59. Springer, 2015.

[11] Richard Brualdi and Bryan Shader. Strong Hall Matrices. *Siam Journal on Matrix Analysis and Applications - SIAM J MATRIX ANAL APPLICAT*, 15, 5 1994.

[12] David Carlson. What are Schur complements, anyway? *Linear Algebra and Its Applications*, 74(C):257–275, 2 1986.

[13] Yanqing Chen, Timothy A Davis, William W Hager, Sivasankaran Rajamanickam, and ; W W Hager. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw*, 35, 2008.

[14] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to Algorithms , Second Edition*. MIT Press, 2001.

[15] T A Davis. Unsymmetric-pattern multifrontal methods for parallel sparse LU factorization. Technical report, Computer and Information Science and Engineering Department, University of Florida, 1991.

[16] Tim Davis. Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Transactions on Mathematical Software - TOMS*, 6 2008.

[17] Timothy A Davis. Algorithm 832: UMFPACK V4.3-An Unsymmetric-Pattern Multifrontal Method. Technical Report 2, University of Florida, 2004.

[18] Timothy A Davis. Algorithm 849: A Concise Sparse Cholesky Factorization Package. Technical Report 4, ACM, 2005.

[19] Timothy A Davis. User Guide for CHOLMOD: a sparse Cholesky factorization and modification package. Technical report, University of Florida, 2011.

[20] Timothy A Davis. Algorithm 9xx: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra. Technical report, Texas A&M University, 2018.

[21] Timothy A Davis and Iain S Duff. An Unsymmetric-Patttern Multifrontal Method for Sparse LU Factorization. Technical Report 1, Computer and Information Science and Engineering Department, University of Florida, 1997.

[22] Timothy A Davis, Sivasankaran Rajamanickam, and Wissam M Sid-Lakhdar. A survey of direct methods for sparse linear systems. Technical report, Acta Numerica, Department of Computer Science and Engineering, Texas A&M Univ, 2016.

[23] Timothy A. Davis and Society for Industrial and Applied Mathematics. *Direct methods for sparse linear systems*. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2006.

[24] Reinhard Diestel. *Graph theory. 5th edition.* Springer, 2017.

[25] Felipe Diniz. Condition number and matrices. Technical report, Federal University of Rio de Janeiro Rio de Janeiro, RJ, Brazil, 5 2017.

[26] B Dougalis, D Noutsos, and A Hadjidimos. Lecture Notes Numerical Linear Algebra, 2000.

[27] I S Duff and Aere Harwell. On Algorithms for Obtaining a Maximum Transversal. Technical report, ACM, 1981.

[28] I S Duff, K Kaya, and B Uçar. Design, Implementation, and Analysis of Maximum Transversal Algorithms. Technical report, CERFACS, 2010.

[29] I. S. Duff and J. K. Reid. An Implementation of Tarjan's Algorithm for the Block Triangularization of a Matrix. *ACM Transactions on Mathematical Software (TOMS)*, 4(2):137–147, 6 1978.

[30] I. S. Duff and J. K. Reid. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Transactions on Mathematical Software (TOMS)*, 9(3):302–325, 9 1983.

[31] I. S. Duff and J. K. Reid. The Multifrontal Solution of Unsymmetric Sets of Linear Equations. *SIAM Journal on Scientific and Statistical Computing*, 5(3):633–641, 9 1984.

[32] Alan. George, J. R. (John R.) Gilbert, and Joseph W. H. Liu. *Graph theory and sparse matrix computation*. Springer-Verlag, 1993.

[33] Alan George and Michael T. Heath. Solution of sparse linear least squares problems using givens rotations. *Linear Algebra and Its Applications*, 34(C):69–83, 1980.

[34] Alan George, Joseph Liu, and Esmond Ng. A Data Structure for Sparse QR and LU Factorizations. *SIAM Journal on Scientific and Statistical Computing*, 9(1):100–121, 1988.

[35] Alan. George and Joseph W. H. Liu. *Computer solution of large sparse positive definite systems*. Prentice-Hall, 1981.

[36] Alan George and Joseph W.H. Liu. Evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 3 1989.

[37] Alan George and Esmond Ng. An Implementation of Gaussian Elimination with Partial Pivoting for Sparse Systems. *SIAM Journal on Scientific and Statistical Computing*, 6(2):390–409, 4 1985.

[38] J George and J Liu. A quotient graph model for symmetric factorization. Technical report, Department of Computer Science, University of Waterloo, 5 1979.

[39] J R Gilbert, X S Li, E G Ng, and B W Peyton. COMPUTING ROW AND COLUMN COUNTS FOR SPARSE QR AND LU FACTORIZATION *. Technical Report 4, DARPA, 2001.

[40] John R. Gilbert. Predicting Structure in Sparse Matrix Computations. *SIAM Journal on Matrix Analysis and Applications*, 15(1):62–79, 1 1994.

[41] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. An Efficient Algorithm to Compute Row and Column Counts for Sparse Cholesky Factorization. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1075–1091, 10 1994.

[42] John R. Gilbert and Tim Peierls. Sparse Partial Pivoting in Time Proportional to Arithmetic Operations. *SIAM Journal on Scientific and Statistical Computing*, 9(5):862–874, 9 1988.

[43] J.R. Gilbert and E.G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. Technical report, Oak Ridge National Laboratory (ORNL), Oak Ridge, TN, 10 1992.

[44] G Golub. Numerical Methods for Solving Linear Least Squares Problems*. *Numerische Mathematik*, 7:206–208, 1965.

[45] Gene H Golub and Charles F Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, USA, 1996.

[46] P Heggernes, S C Eisenstat, G Kumfert, and A Pothen. The Computational Complexity of the Minimum Degree Algorithm. Technical report, he Computer Science Research Institution, 2000.

[47] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 1 2002.

[48] Alston Householder. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM (JACM)*, page 10, 1958.

[49] Stephen Ingram. Minimum Degree Reordering Algorithms: A Tutorial. Technical report, University of British Columbia, 2006.

[50] Scott P Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The SuiteSparse Matrix Collection Website Interface Software • Review • Repository • Archive. *Journal of Open Source Software*, 2019.

[51] C Lawson, F Krogh, S Gold, David Kincaid, J Sullivan, E Williams, Richard Hanson, Karen Haskell, Jack Dongarra, and Cleve Moler. Linear-Algebra Programs. *ACM*, 6 1982.

[52] Joseph W. Liu. A Compact Row Storage Scheme for Cholesky Factors Using Elimination Trees. *ACM Transactions on Mathematical Software (TOMS)*, 12(2):127–148, 6 1986.

[53] Joseph W.H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software (TOMS)*, 11(2):141–153, 6 1985.

[54] Joseph W.H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1 1990.

[55] Joseph W.H. Liu. Multifrontal method for sparse matrix solution. Theory and practice. *SIAM Review*, 34(1):82–109, 7 1992.

[56] MATLAB. *MATLAB version 9.4.0.813654 (R2018a)*. The MathWorks Inc., Natick, Massachusetts, 2018.

[57] Esmond Ng and Barry W. Peyton. A Supernodal Cholesky Factorization Algorithm for Shared-Memory Multiprocessors. *SIAM Journal on Scientific Computing*, 14(4):761–769, 7 1993.

[58] S. Parter. The Use of Linear Graphs in Gauss Elimination. *SIAM Review*, 3(2):119–130, 4 1961.

[59] Alex Pothen and Chin-Ju Fan. Computing the Block Triangular Form of a Sparse Matrix. Technical report, Pennsylvania State University, 1990.

[60] Matthew W Reid. Pivoting for LU Factorization. Technical report, University of Puget Sound, 2014.

[61] Robert Schreiber. A New Implementation of Sparse Gaussian Elimination. *ACM Transactions on Mathematical Software (TOMS)*, 8(3):256–276, 9 1982.

[62] Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra, editors. *Computational Science — ICCS 2002*, volume 2329 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[63] B Speelpenning. *The generalized element method*. Report (University of Illinois at Urbana-Champaign. Dept. of Computer Science)no. 946. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, 1978.

[64] Gilbert Strang. *Introduction to linear algebra*. SIAM, fifth edition, 2016.

[65] Robert Tarjan. DEPTH-FIRST SEARCH AND LINEAR GRAPH ALGORITHMS*. Technical Report 2, Department of Computer Science,Cornell University, 1972.

[66] Mihalis Yannakakis. Computing the Minimum Fill-In is NP-Complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 3 1981.

Appendix I

Computer Algorithms

In this appendix the computer algorithms analyzed and used in this thesis are summarized.

I.1 Data Structures and Basic Algorithms

```
function C=TripToSparse(A)
    row=A.row
    value=A.value
    col=A.col
    for k=1:nz
        count(col(k))++
    end
    Cp=cumsum(count)
    for k=1:nz
        p=col(k)++
        rowIdx(p)=row(k)
        val(p)=value(k)
    end
end

function z=gaxpy(A,x,y)
    for k=1:n
        for p=Ap(k):Ap(k+1)-1
            y(Ai(p))=Ax(p)*x(k)
        end
    end
end

function C=mat_multiply(A,B)
    nz=0
    for j=1:n
        Cp[j]=nz
        for k=Bp(j):Bp(j+1)-1
            b=Bx(k)
            for p=A(k):A(k+1)-1
                i=Ai(p)
                if(i not in nz pattern of col_j)
                    Ci[nz++]=i
                x(i)+=b*Ax(p)
            end
            for p=Cp(j):nz
                Cx(p)=x(Ci(p))
            end
        end
    end
    Cp[n]=nz
end

 function C=mat_add(A,B)
    nz=0
    for j=1:n
        Cp(j)=nz
        for Ap(j):Ap(j+1)-1
            i=Ai(p)
            if(i not in nz pattern of col_j)
                Ci[nz++]=i
            x(i)+=alpha*Ax(p)
        end

          for Bp(j):Bp(j+1)-1
            i=Bi(p)
            if(i not in nz pattern of col_j)
                Ci[nz++]=i
            x(i)+=beta*Bx(p)
        end
        for p=Cp(j):nz
            Cx(p)=x(Ci(p))
        end
    end
 end
```

```
function x=lsolve(A,b)
    x=b
    for j=1:n
        x(j) =x(j)/Lx(Lp(j))
        for p=Lp(j)+1:Lp(j+1)-1
            x(Li(p))=x(Li(p))-Lx(p)*x(j)
        end
    end
end

function x=usolve(A,b)
    x=b
    for j=n:1
        x(j)=x(j)/Lx(Li(j+1)-1)
        for p=Up(j):Up(j)-2
            x(Ui(p))=x(Ui(p))-Ux(p)*x(j)

        end
    end
end

function x=ltsolve(A,b)
    x=b
    for j=n:1
        for p=Lp(j)+1:Lp(j+1)-1
            x(j)=x(j)-Lx(p)*x(Li(p))
        end
        x(j)=x(j)/Lx(Lp(j))
    end
end

function x=utsolve(A,b)
    x=b
    for j=1:n
        for p=Lp(j)+1:Lp(j+1)-2
            x(j)=x(j)-Ux(p)*x(Ui(p))
    end
    x(j)=x(j)/Ux(Up(j+1)-1)
end
```

I.2 Sparse Lower Triangular Solve

```
function $\mathcal{X}$=reach(L,$\mathcal{B}$)
    for each i for which $b_i \neq 0$
        if node i is unmarked
            dfs(i)

function dfs(j)
    mark(j)
    for each i for which $l_{ij} \neq 0$
        if node(i) is unmarked
            dfs(i)
    push j onto stack for $\mathcal{X}$
```

```
function x=lsparse_solve(L,B,k)
    X=cs_reach(L,B,k)
    for p=Bp(k):Bp(k)-1 % b is stored in a CSC format
        x(Bi(p))=Bx(p)  % so it is scattered
    end
    for s=1:length(X)
        j=X(s)
        x(j)=x(j)/Lx(Lp(J));
        for p=Gp(j)+1:Gp(j+1)-1
            x(Li(p))=x(Li(p))-Lx(p)*x(j)
        end
    end
end
```

I.3 Cholesky Decomposition

```
function L=up_cholesky(A)
    n=size(A)
    L=zeros(n)
    for k=1:n
        L(k,1:k-1)=(L(1:k-1,1:k-1)\A(1:k-1,k))';
        L(k,k)=sqrt(A(k,k)-L(k,1:k-1)*L(k,1:k-1)')
    end
end
```

```
function Lk=ereach(A,parent,k)
    n=size(A,1)
    Lk=zeros(n,1) % output array
    w=zeros(n,1) % work space array for marking values
    s=zeros(n,1) % stack array
    top=n
    mark(w,k) % mark node k as visited
    for p=Ap(k):Ap(k+1)-1
        i=Ai(p)
        if i>k
            continue
        end
        len=1
        while i=parent(i)
            if mark(w,i) % check if node i has already been visited
               break
            end
            s(len)=i
            len=len+1
            mark(w,i)
        end
        while len>0
            Lk(top)=s(len)
            top=top-1
            len=len-1
        end
    end
    Lk=Lk(Lk~=0) % remove if there are any zeros
    for p=1:length(Lk)-1
        mark(w,Lk(i)) % unmark node Lk(i)
    end
    mark(w,k)
end

function P=postorder(T)
    k=0
    for j=1:length(T)
        dfstree(j)
    end
end


function dfstree(j)
    for i=1:n
        if i is a child of j
```

```
            dfstree(i)
        end
    post(k)=j
    k=k+1
    end
end

function [first,level]=first_descendant(n,post,parent,first,level)
    first=zeros(n,1)
    for k=1:n
        i=post(k)
        len=0
        r=i
        while((r!=0) && (first(r)==0)
            first(r)=k
            r=parent(r)
            len=len+1
        end
        if r==0
            len=len-1
        else
            len=len+level(r)
        end
        s=i
        while(s~=r)
            level(s)=len
            len=len-1
            s=parent(s)
        end

    end
end

function [q, maxfirst, prevleaf, ancestor, jleaf] =leaf(i, j, first,
maxfirst, prevleaf, ancestor, jleaf)
     jleaf=0
    if (i<=j || first(j)<=maxfirst(i))
        q=0 %not a leaf
        return
    end
```

```
        maxfirst(i)=first(j) % leaf → update maxfirst
        jprev=prevleaf(i) % load prevleaf and upate the new
        prevleaf(i)=j
        if (jprev==−1) % first leaf
            q=i %i is the root of i subtree
            jleaf=1
            return
        end
        jleaf=2 % not first leaf
        q=jprev
        while (q~=ancestor(q)) % find the root of tree
        q=ancestor(q)
        end
        s=jprev
        while(s~=q)
            sparent=ancestor(s) % path compression with ancestor method
            ancestor(s)=q
            s=sparent
        end
end

function rcount=rowcount(A, parent, post)
    n=A.n
    Ap=A.p
    Ai=A.i
    [first, level]= first_descendant(n, post, parent, first, level)
    for i=1:n
        rcount(i)=1 % for the diagonal
        prevleaf(i)=0
        maxfirst(i)=0
        ancestor(i)=i % every node is its own ancestor
    end
    for k=1:n
        j=post(k) % use postordering through permutation vector
        for p=Ap(k):Ap(k+1)−1
            i=Ai(p)
            [q, maxfirst, prevleaf, ancestor, jleaf] =leaf(i, j, first,
            maxfirst, prevleaf, ancestor, jleaf)
            if jleaf
                rcount(i)=rcount(i)+(level(j)−level(q))
            end
        end
        if (parent(j)~=--1)
            ancestor(j)=parent(j)
        end
    end
end

function ccount=colcounts(A,parent,post)
    n=A.n
    Ai=A.i
    Ap=A.p
    w=zeros(1,n)
    for k=1:n
        j=post(k)
        if (first(j)==0) % j is a leaf
            delta(j)=1
        else
            delta(j)=0 % j not a leaf
        end
        while ((j~=0) && (first(j)==0)) % construct the first of each node
            first(j)=k
            j=parent(j)
        end
        ancestor(k)=k % every node is each own ancestor at first
    end

    for k=1:n
        j=post(k)
        for p=Ap(j):Ap(j+1)-1
            i=Ai(p)
            [q, maxfirst, prevleaf, ancestor, jleaf]=leaf(i, j, first,
            maxfirst, prevleaf, ancestor, jleaf)
            if (jleaf>=1) delta(j)=delta(j)+1 % $a_{i,j} \in \hat{\mathcal{A}}$
            if (jleaf==2) delta(q)=delta(q)-1 % j is subsequent leaf
                                              % so delta(lca)- -
        end
        if (parent(j)~=-0)
            ancestor(j)=parent(j) % every set belongs to its father
        end
    end
    ccount=delta
    for j=1:n
        if (parent(j)~=0)
            ccount(parent(j))=ccount(parent(j))+ccount(j)
            % the colcount of a node j is the sum of its children count
        end
    end
end

function L=up_chol(A,A_symb)
    n=A.n
    cp=A_symb.p
    Lp=c=cp
    Ap=A.p
    Ai=A.i
    parent=A_symb.parent
    for k=1:n
        % Find the non zero pattern of $A_{:k}$
        Lk=ereach(A,parent,k) % find ereach
        x(k)=0
        for p=Ap(k):Ap(k+1)-1
            if Ai(i)<=k
                x(Ai(p)=Ax(p)
            end
        end
        d=x(k)
        x(k)=0
        % Triangular solve for $L_{k,:}$
        for Lk_c=1:length(Lk)
            i=Lk(Lk_c) % pattern of Lk
            lki=x(i)/Lx(Lp(i)) % L(k,i)=x(i)/L(i,i)
            x(i)=0
            for p=Lp(i)+1:c(i)-1
                x(Li(p)=x(Li(p))-Lx(p)*lki
            end
            d=d-lki*lki
            p=c(i)
            c(i)=c(i)+1
            Li(p)=k
            Lx(p)=lki
        end
        % Check if positive definite and find $L_{kk}$
```

```
          if (d<=0) %not positive definite
               L=0
               return
          end
          p=c(k)
          c(k)=c(k)+1
          Li(p)=k
          Lx(p)=sqrt(d)
     end
     Lp(n)=cp(n)
     L.p=Lp
     L.i=Li
     L.x=Lx
     L.n=Ln
end

function L=chol_super(A,s)
     n=size(A)
     L=zeros(n)
     ss=cumsum([1 s])
     for j=1:length(s)
          k1=ss(j)
          k2=ss(j+1)
          k=k1:(k2-1)
          L(k,k)=chol(A(k,k)-L(k,1:k1-1)*L(k,1:k1-1)')'
          L(k2:n,k)=(A(k2:n,k)-L(k2:n,1:k1-1)*L(k,k1:k1-1)')/L(k,k)'
     end
end
```

I.4 QR Decomposition

```
function [V,beta,R]=qr_right(A)
     [m n]=size(A);
     V=zeros(m,n);
     Beta=zeros(1,n);
     for k=1:n
          [v,beta,s]=gallery('house',A(k:m,k),2);
          V(k:m,k)=v;
          Beta(k)=beta;
          A(k:m,k:n)=A(k:m,k:n)-v*(beta*(v'*A(k:m,k:n)));
     end
```

```
    R=A;
end

function [V,beta,R]=qr_left(A)
    [m n]=size(A);
    V=zeros(m,n);
    Beta=zeros(1,n);
    R=zeros(m,n);
    for k=1:n
        x=A(:,k);
        for i=1:k-1
            v=V(i:m,i);
            beta=Beta(i);
            x(i:m)=x(i:m)-v*(beta*(v'*x(i:m)));
        end
        [v,beta,s]=gallery('house',x(k:m),2);
        V(k:m,k)=v;
        Beta(k)=beta;
        R(1:(k-1),k)=x(1:(k-1));
        R(k,k)=s
    end
end

function [unz, pinv, leftmost]=vcount(A,parent)
    [m n]=size(A);
    next=array(m,1); % next entry in col k
    head=array(n,1); % first entry in col k
    tail=array(n,1); % last entry in col k
    nque=array(n,1); % number of entries each col

    for i=1:n
        leftmost(i)=min(find(A(i,:)));
    end

    for i=m:1
        pinv(i)=0;
        k=lefmost(i);
        if (nque(k)==0)
            nque(k)=nque(k)+1;
            tail(k)=i;
        end
```

```
          next(i)=head(k);
          head(k)=i;
     end
     lnz=0;

     for k=1:n
          i=head(k); % remove row i from queue
          lnz=lnz+1; % count diagonal entry V(k,k)
          nque(k)=nque(k)-1;
          pinv(i)=k;
          if (nque(k)<=0) % empty below the diagonal
               continue;
          end;
          lnz=lnz+nque(k); % nque(k) is the nnz count below the diagonal
          pa=parent(k);   % transfer all rows to parent
          if (pa~=-1)
               if(nque(pa)==0)
                    tail(pa)=head(k);
               end
               next(tail(k))=head(pa);
               head(pa)=next(i);
               nque(pa)=nque(pa)+nque(k);
          end
     end
     for i=1:m % rows that did not hold a diagonal entry
          if (pinv(i)=0)
               k=k+1;
               pinv(i)=k;
          end
     end
end

function [V,R]=qr(A,A_symb)
     [m n]=size(A);
     Ai=A.i; Ap=A.p;
     q=A_symb.q; parent=A_symb.parent;
     pinrv=A_symb.pinv;
     leftmost=A_symb.leftmost;
     rnz=1; lnz=1;
     w=zeros(m);
```

```
for k=1:n % compute V, R
    Rp(k)=rnz; % R(:,k) starts here
    Vp(k)=p1=lnz; % V(:,k) starts here
    w(k)=k; % V(k,k) pattern of V
    Vi(vnz)=k;
    vnz=vnz+1;
    top=n;
    for p=Ap(k):Ap(k+1)-1 % find R(:,k) pattern
        i=leftmost(Ai(p));
        len=1;
        while (w(i)~=k) % traverse up to k
            s(len)=i;
            len=len+1;
            w(i)=k;
            i=parent(i);
        end
        while(len>1) % push path on stack
            top=top-1;
            len=len-1;
            s(top)=s(len);
        end
        i=pinvA(i(p));
        x(i)=Ax(p); % x(i)=A(:,col)
        if ( i>k && w(i) < k) % pattern (V:,k)=x(k+1,m)
            Vi(vnz)=i;
            vnz=vnz+1; % add i to pattern of V(:,k)
            w(i)=k;
        end
    end
    for p=top:n % for i in R(:,k)
        i=s(p); % R(i,k) is nonzero
        for j=Vp(i):Vp(i)-1 % apply the previous Householders
            mul=mul+Vx(j)*x(Vi(p));
        end
        mul=mul*beta(i);
        for j=Vp(i):Vp(i)-1
            x(Vi(p))=x(Vi(p))-Vx(p)*mul;
        end
        Ri[rnz]=i;
        Rx[rnz]=x(i);
        rnz=rnz+1;
```

```
                x(i)=0;
            end
            for p=p1:vnz
                Vx(p)=x(Vi(p));
                x(Vi(p))=0;
            end
            Ri(rnz)=k;
            [v,beta]=house(x); % apply the current Householder
        end
        Rp(n)=rnz; % finalize the matrices
        Vp(n)=vnz;
        R.p=Rp; R.i=Ri; R.x=Rx;
        V.p=Vp; V.i=Vi; V.x=Vx;
end
```

I.5 LU Decomposition

```
 function [L,U] = lu_right(A)
    n=size(A,1);
    L=eye(n);
    U=zeros(n);
    for k=1:n
        U(k,k:n)=A(k,k:n);
        L(k+1:n,k)=A(k+1:n,k)/U(k,k);
        A(k+1:n,k+1:n)=A(k+1:n,k+1:n)-L(k+1:n,k)*U(k,k+1:n);
    end
 end


function [L,U,P]=left_looking_lu(A)
    n=size(A,1);
    P=eye(n);
    L=zeros(n);
    U=zeros(n);
    for k=1:n
        % create LHS matrix :: 1:k-1 columns of L, k:n Identity ,
        x_start=[L(:,1:k-1) [zeros(k-1,n-k+1);eye(n-k+1)]];
        x=x_start\(P*A(:,k)); % sparse triangular solve to find x
        U(1:k-1,k)=x(1:k-1); % result of x1 are the (1:k-1,k) entries
                                  % of U
        [a,i]=max(abs(x(k:n))); % find the new pivot
        i=i+k-1; % dimensions of submatrix to
```

```
                % dimensions of the whole matrix
        L([i k],:)=L([k i],:); % row permutations of L
        P([i k],:)=P([k i],:); % row permutations of P
        x([i k])=x([k i]); % row permutations of x
        U(k,k)=x(k); % store the pivot entry
        L(k,k)=1; % diagonal entries of L=1
        L(k+1:n,k)=x(k+1:n)/x(k); % $\ell_{32} = x_3/u_{22}$
    end
end


function [L,U,pinv]=left_looking_sparse_lu(A)
    n=A.n;
    pinv=zeros(n,1);
    Lp=zeros(n,1);
    xi=zeros(n,1);
    lnz=1;
    unz=1;
    for k=1:n
        [x, X]=lsparse_solve(L,A,k,pinv); % x=L\A(:,col)
        ipiv=0;
        a=-1;
        for p=1:length(X) % find the pivot
            i=X(p); % non zero value of x
            if (pinv(i)==0) % has not been chosen as pivot yet
                t=abs(x(i)); % absolute value of possible pivot
                if(t>a)
                    a=t; % largest pivot candidate until now
                    ipiv=i; % pivot's row index
                end
            else
            % U(pinv(i),k)=x(i), creating the vector $u_{12} = x_1$
                Ui(unz)=pinv(i);
                Ux(unz)=x(i);
                unz=unz+1;
            end
        end
        pivot=x(ipiv); % the chosen pivot
        Ui(unz)= k; % U(k,k) index
        Ux(unz)=pivot; % U(k,k) assignment
        unz=unz+1;
        % dividing by pivot to find the column of L
        for p=1:length(X)
            i=xi(p);
            if (pinv(i)==0) % x(i) is an entry in L(:,k)
                Li(lnz)=i; % row index of i
                Lx(lnz)=x(i)/pivot; % value of L divided by pivot
            end
            x(i)=0;
        end
    end
    Lp(n+1)=lnz; % terminate column pointers
    Up(n+1)=unz;
    for p=1:lnz % point old row indices to the new
        Li(p)=pinv(Li(p));
    end
end

function [x, X]=lsparse_solve(L,B,k,pinv)
    X=cs_reach(L,B,k,pinv)
    for p=Bp(0):Bp(1)-1 % b is stored in a CSC format,
        x(Bi(p))=Bx(p);  % so it is scattered
    end
    for s=1:length(X)
        j=X(s);
        if(pinv)
            J=pinv(j);
        end
        if (J==0)
            continue;
        end
        x(j)=x(j)/Lx(Lp(J));
        for p=Gp(J)+1:Gp(J+1)-1
            x(Li(p))=x(Li(p))-Lx(p)*x(j);
        end
    end
end
```

```
function $\mathcal{X}$=reach(L,$\mathcal{B}$,pinv)
    for each i for which $b_i \neq 0$
        if node i is unmarked
            dfs(i,pinv)
```

```
function dfs(j,pinv)
    mark(j)
    jnew=pinv(j)
    for each i for which $l_{ijnew} \neq 0$
        if node(i) is unmarked
            dfs(i,pinv)
    push j onto stack for $\mathcal{X}$
```