

SPARSE DIRECT METHODS

- Building blocks for sparse direct solvers
- SPD case. Sparse Column Cholesky/
- Elimination Trees - Symbolic factorization

Direct Sparse Matrix Methods

Problem addressed: Linear systems

$$Ax = b$$

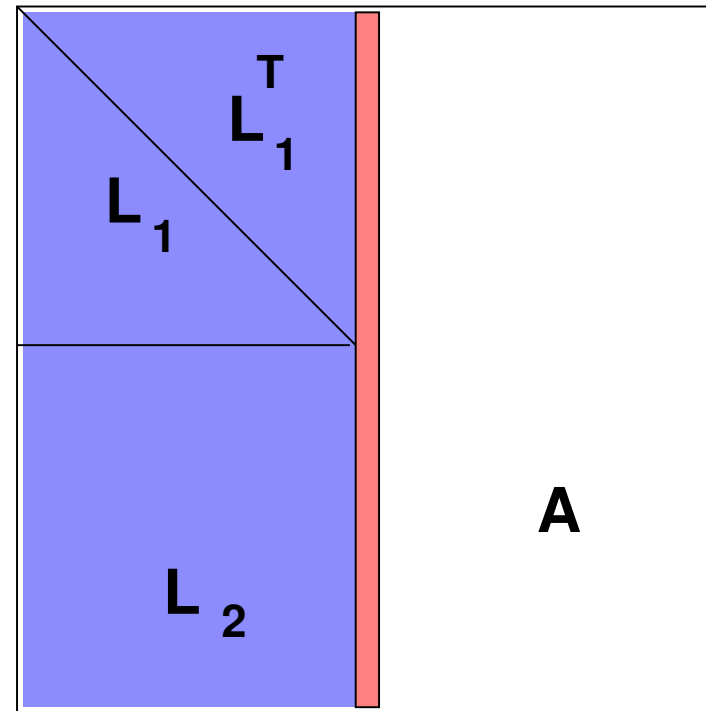
- We will consider mostly Cholesky –
- We will consider some implementation details and tricks used to develop efficient solvers

Basic principles:

- Separate computation of structure from rest [symbolic factorization]
- Do as much work as possible statically
- Take advantage of clique formation (supernodes, mass-elimination).

Sparse Column Cholesky

```
For  $j = 1, \dots, n$  Do:  
   $l(j : n, j) = a(j : n, j)$   
  For  $k = 1, \dots, j - 1$  Do:  
    // cmod(k,j):  
     $l_{j:n,j} := l_{j:n,j} - l_{j,k} * l_{j:n,k}$   
  EndDo  
  // cdiv (j) [Scale]  
   $l_{j,j} = \sqrt{l_{j,j}}$   
   $l_{j+1:n,j} := l_{j+1:n,j} / l_{j,j}$   
EndDo
```



The four essential stages of a solve

1. Reordering: $A \longrightarrow A := PAP^T$

- Preprocessing: uses graph [Min. deg, AMD, Nested Dissection]

2. Symbolic Factorization: Build static data structure.

- Exploits 'elimination tree', uses graph only.
- Also: 'supernodes'

3. Numerical Factorization: Actual factorization $A = LL^T$

- Pattern of L is known. Uses static data structure. Exploits supernodes (blas3)

4. Triangular solves: Solve $Ly = b$ then $L^T x = y$

ELIMINATION TREES

The notion of elimination tree

- Elimination trees are useful in many different ways [theory, symbolic factorization, etc..]
- For a matrix whose graph is a tree, parent of column $j < n$ is defined by

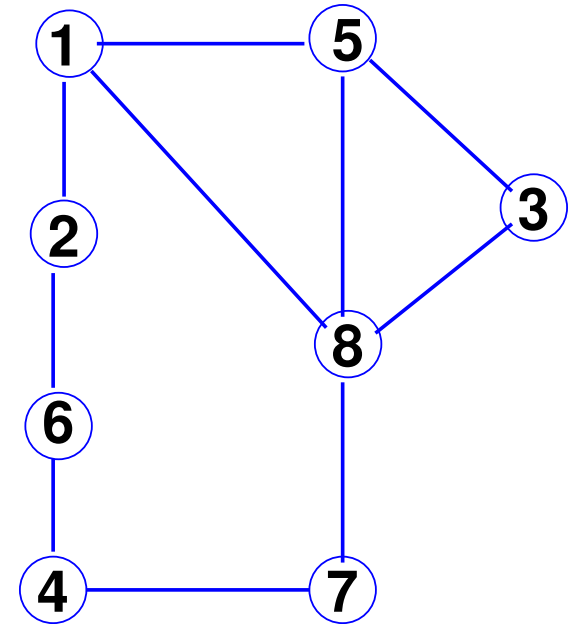
$$\mathit{Parent}(j) = i, \text{ where } a_{ij} \neq 0 \text{ and } i > j$$

- For a general matrix matrix, consider $A = LL^T$, and $G^F =$ 'filled' graph = graph of $L + L^T$. Then

$$\mathit{Parent}(j) = \min(i) \text{ s.t. } a_{ij} \neq 0 \text{ and } i > j$$

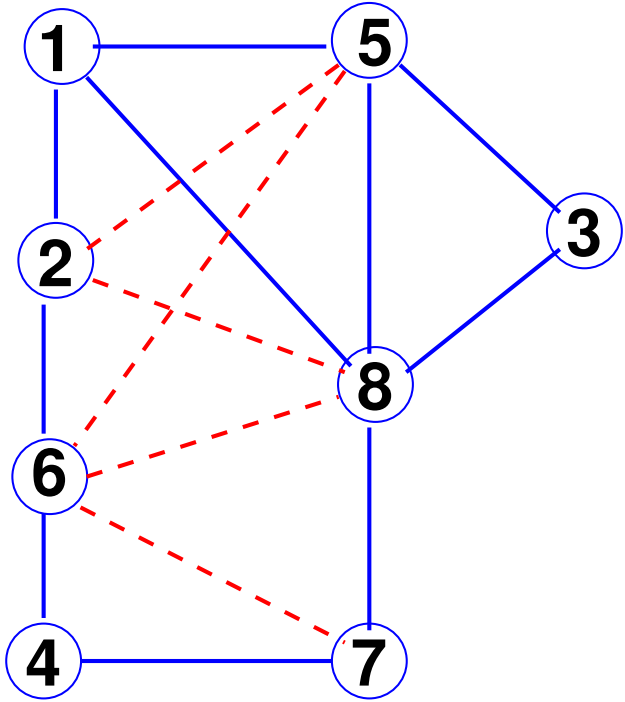
- Defines a tree rooted at column n (Elimintion tree).

Example: Original matrix and Graph

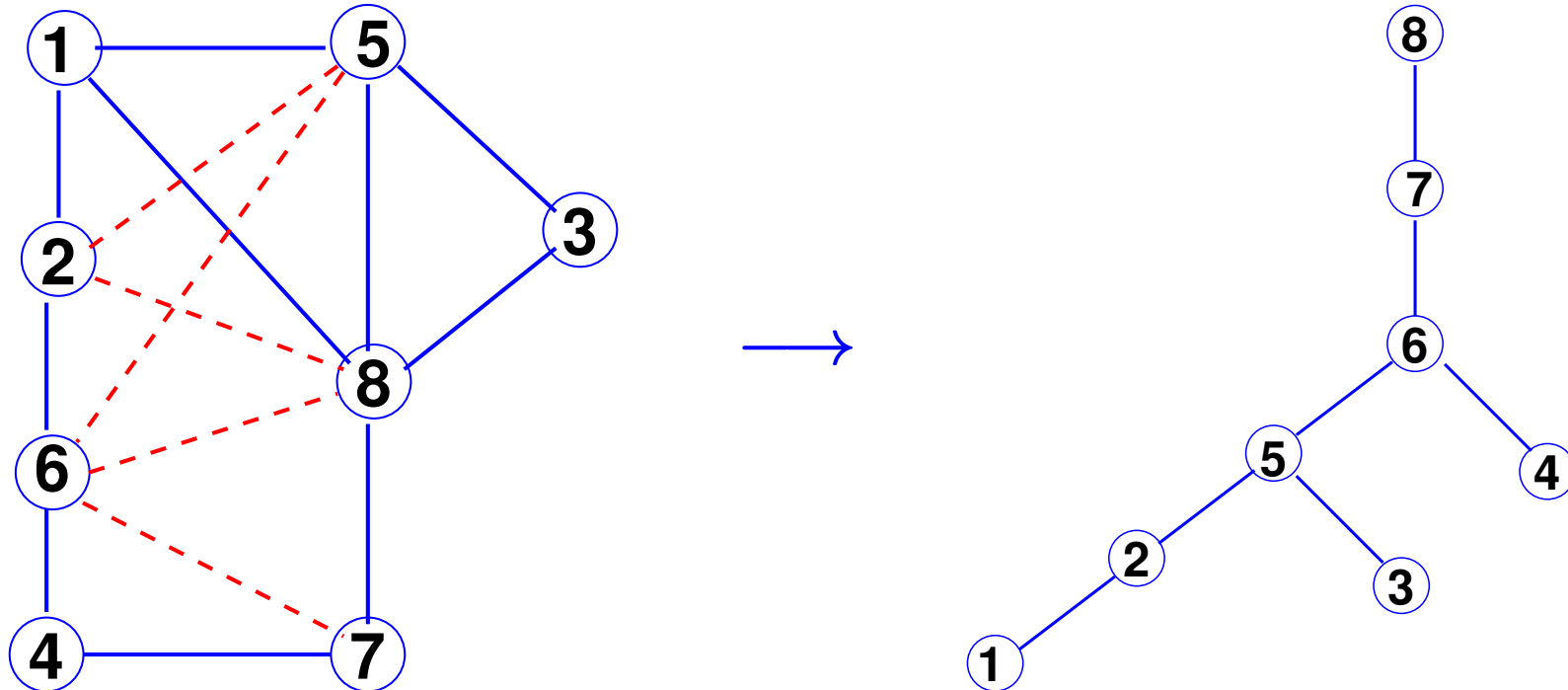
$$\begin{bmatrix} 1 & \star & & & \star & & & \star \\ \star & 2 & & & & & \star & \\ & & 3 & & \star & & & \star \\ & & & 4 & & \star & \star & \\ \star & & \star & & 5 & & & \star \\ & \star & & \star & & 6 & & \\ & & & \star & & & 7 & \star \\ \star & & \star & & \star & & \star & 8 \end{bmatrix}$$


Filled matrix+graph

1	★			★			★
★	2			■	★		■
		3		★			★
			4		★	★	
★	■	★		5	■		★
	★		★	■	6	■	■
			★		■	7	★
★	■	★		★	■	★	8



Corresponding Elimination Tree

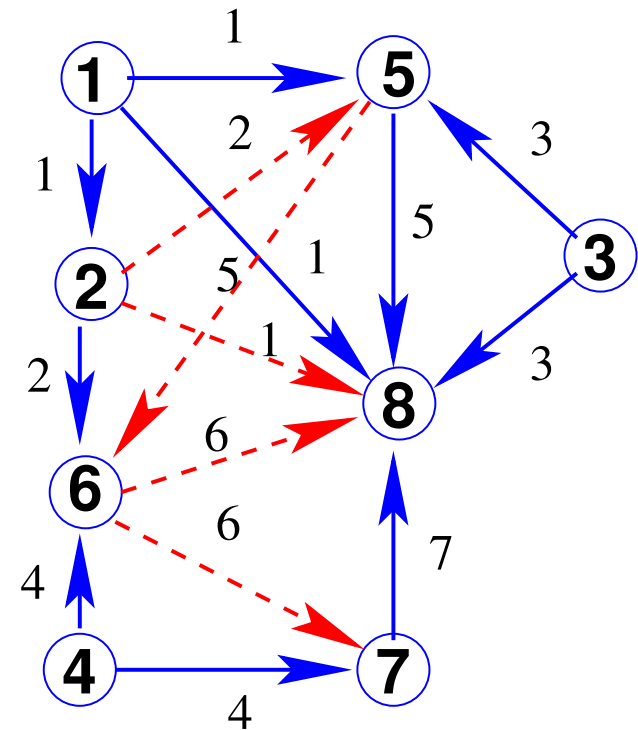


- $\text{Parent}(i) = \text{'first nonzero entry in } L(i+1:n,i)\text{'}$
- $\text{Parent}(i) = \min \{j > i \mid j \in \text{Adj}_{G^F}(i)\}$

Where does the elimination tree come from?

- Answer in the form of an exercise.

Consider the elimination steps for the previous example. A directed edge means a row (column) modification. It shows the task dependencies. There are unnecessary dependencies. For example: $1 \rightarrow 5$ can be removed because it is subsumed by the path $1 \rightarrow 2 \rightarrow 5$.



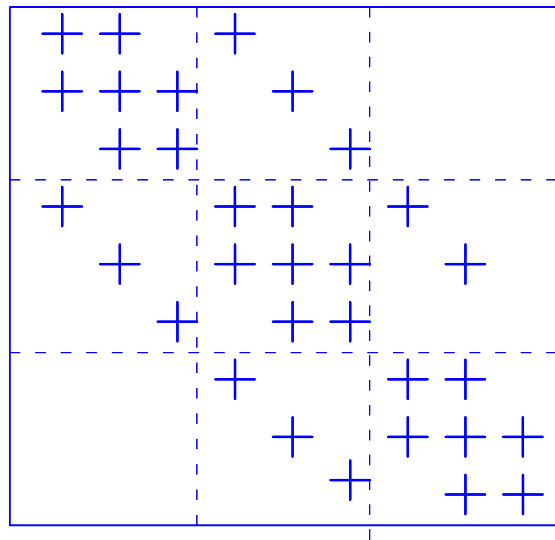
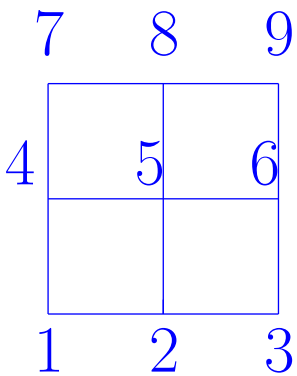
To do: Remove all the redundant dependencies.. What is the result?

Facts about elimination trees

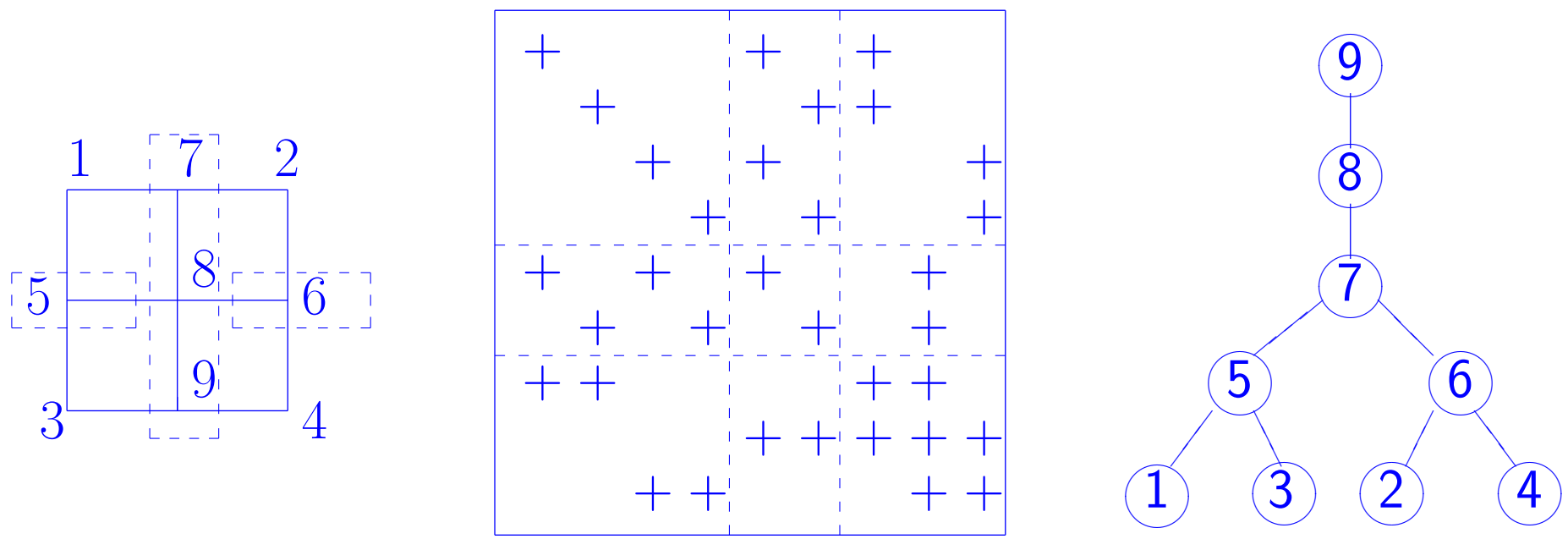
- Elimination Tree defines dependencies between columns.
- The root of a subtree cannot be used as pivot before any of its descendents is processed.
- Elimination tree depends on ordering;
- Can be used to define 'parallel' tasks.
- For parallelism: flat and wide trees → good; thin and tall (e.g. of tridiagonal systems) → Bad.
- For parallel executions, Nested Dissection gives better trees than Minimum Degree ordering.

Elim. tree depends on ordering (Not just the graph)

Example: 3×3 grid for 5-point stencil [natural ordering]




➤ Same example with nested dissection ordering

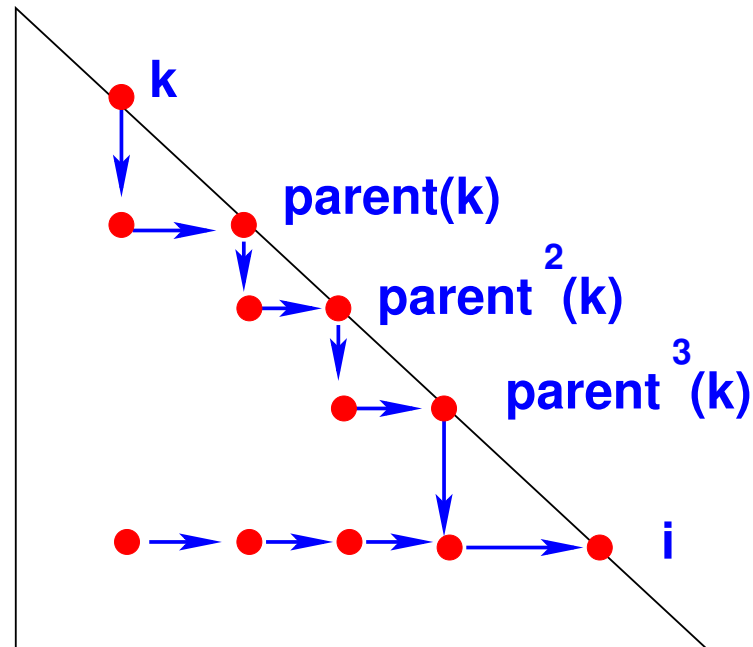


Properties

➤ The elimination tree is a spanning tree of the filled graph [a tree containing all vertices] - obtained by removing edges.

➤ If $l_{ik} \neq 0$ then i is an ancestor of k in the tree

 In the previous example: follow the creation of the fill-in (6,8).



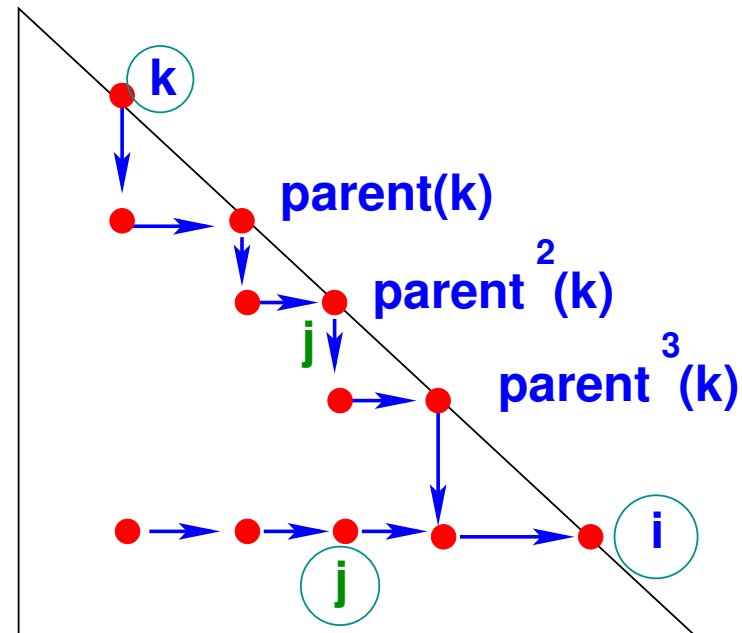
In particular: if $a_{ik} \neq 0, k < i$ then $i \rightsquigarrow k$

➤ **Consequence:** no fill-in between branches of the same subtree

Elimination trees and the pattern of L

- It is easy to determine the sparsity pattern of L because the pattern of a given column is “inherited” by the ancestors in the tree.

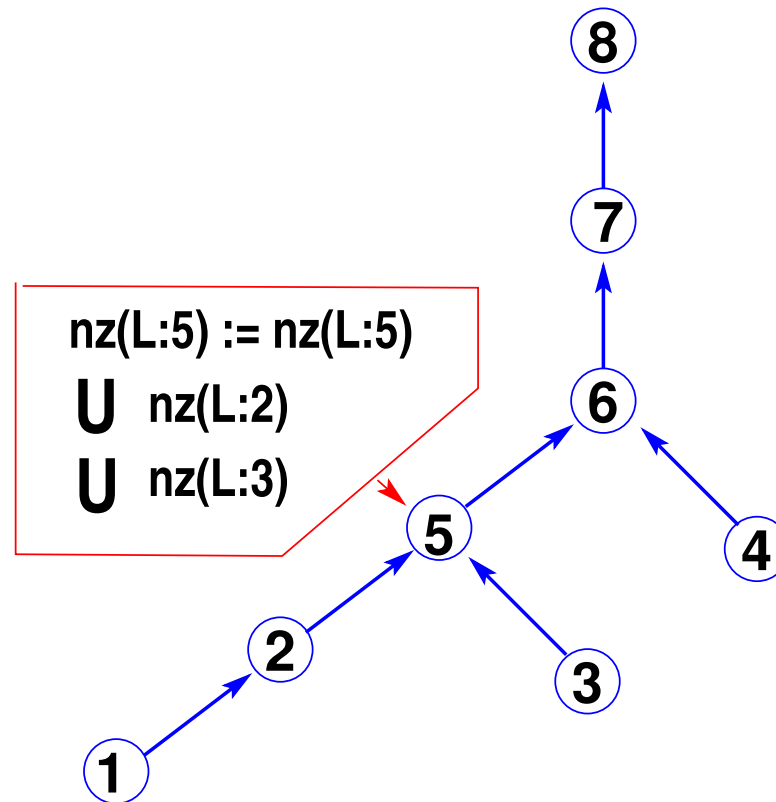
Theorem: For $i > j$, $l_{ij} \neq 0$ iff j is an ancestor of some $k \in Adj_A(i)$ in the elimination tree.



In other words:

$$l_{ij} \neq 0, i > j \text{ iff } \left| \begin{array}{l} \exists k \in Adj_A(i) \text{ s.t.} \\ j \rightsquigarrow k \end{array} \right.$$

In theory: To construct the pattern of L , go up the tree and accumulate the patterns of the columns. Initially L has the same pattern as $TRIL(A)$.



- **However:** Let us assume tree is not available ahead of time
- **Solution:** Parents can be obtained dynamically as the pattern is being built.
- This is the basis of symbolic factorization.

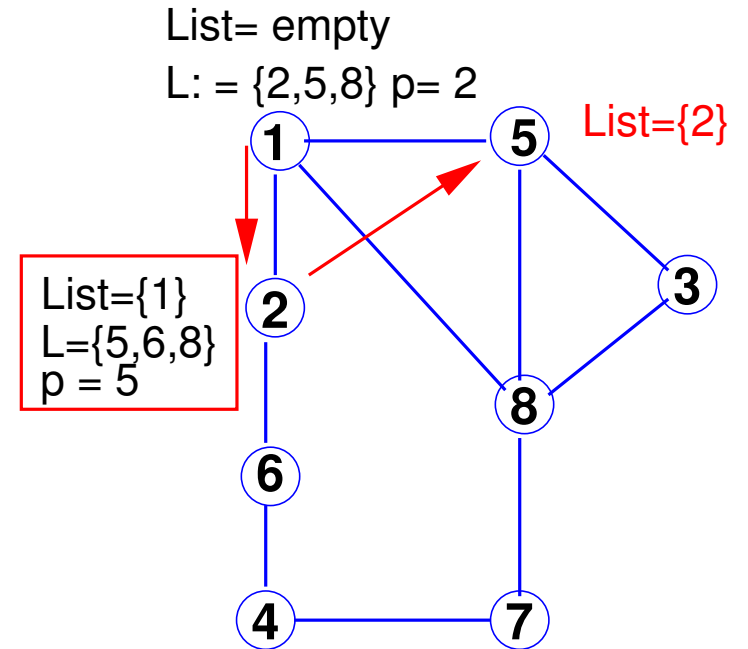
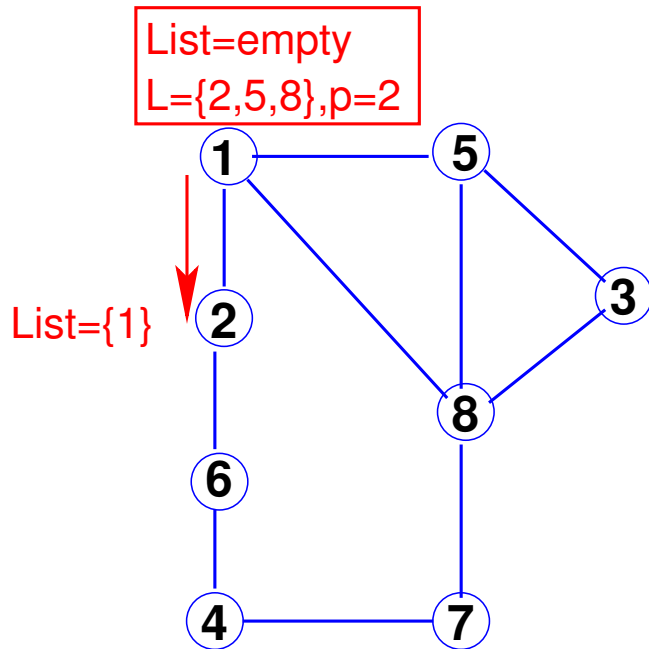
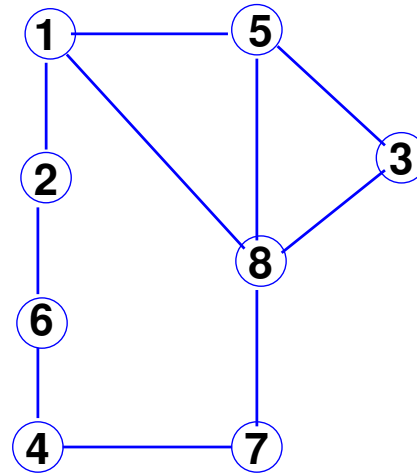
Notation :

- $nz(\mathbf{X})$ is the pattern of \mathbf{X} (matrix or column, or row). A set of pairs (i, j)
- $tril(\mathbf{X}) =$ Lower triangular part of pattern [matlab notation]
 $\{(i, j) \in \mathbf{X} \mid i > j\}$
- Idea: dynamically create the list of nodes needed to update $\mathbf{L}_{:,j}$.

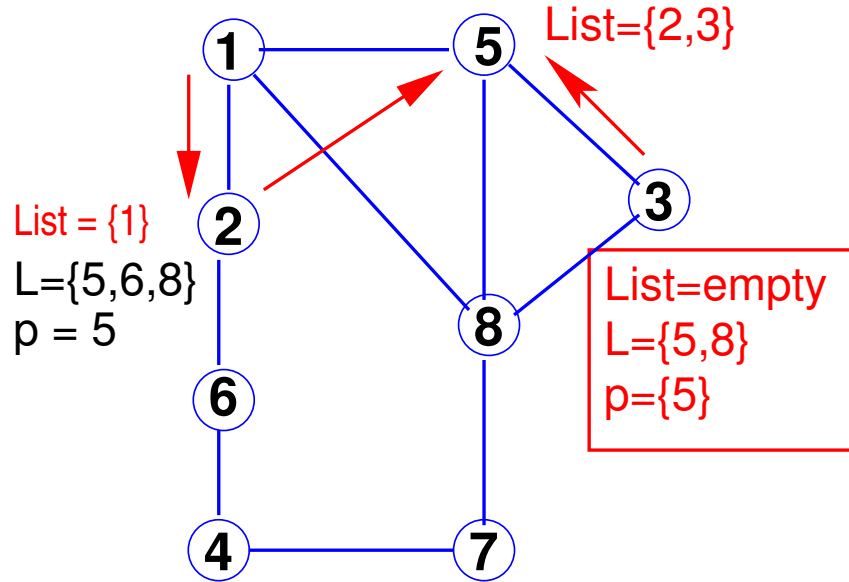
ALGORITHM : 1. *Symbolic factorization*

1. Set: $nz(L) = tril(nz(A))$,
2. Set: $list(j) = \emptyset, j = 1, \dots, n$
3. For $j = 1 : n$
4. for $k \in list(j)$ do
5. $nz(L_{:,j}) := nz(L_{:,j}) \cup nz(L_{:,k})$
6. end
7. $p = \min\{i > j \mid L_{i,j} \neq 0\}$
8. $list(p) := list(p) \cup \{j\}$
9. End

Example: Consider the earlier example:



List= empty
L = {2,5,8}, p = 2



List= empty
L = {2,6,8}, p = 2

