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 –

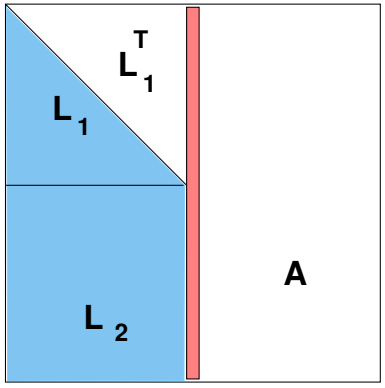
➤ We will consider some implementation details and tricks used to develop efficient solvers

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 $j = 1, \ldots, n$ Do:
  $l(j:n, j) = a(j:n, j)$
  For $k = 1, \ldots, j-1$ Do:
    // cmod(k,j):
    $l_{j:n,j} := l_{j:n,j} - l_{j,k} * l_{j:n,k}$
  EndDo
  // cdiv (j) [Scale]
  $l_{j,j} = \sqrt{l_{j,j}}$
  $l_{j+1:n,j} := l_{j+1:n,j}/l_{jj}$
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 known. Use static data structure. Exploit supernodes

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

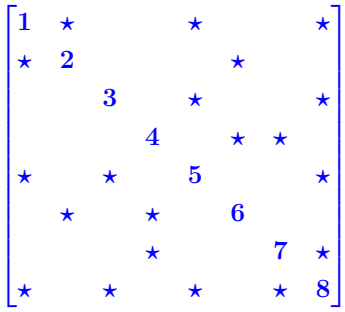
$$Parent(j) = i, \text{ where } a_{ij} \neq 0 \text{ and } i>j$$

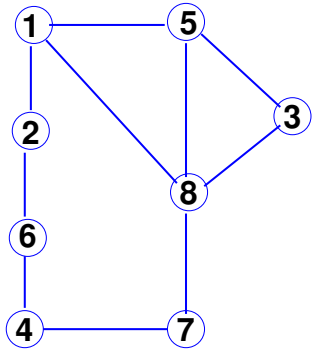
- For a general matrix matrix, consider $A = LL^T$, and G^F = 'filled' graph = graph of $L + L^T$. Then

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

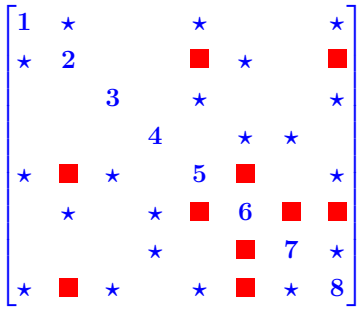
- Defines a tree rooted at column n (Elimintion tree).

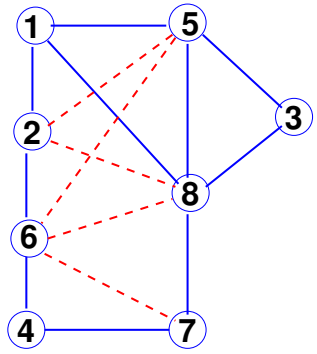
Example: Original matrix and Graph



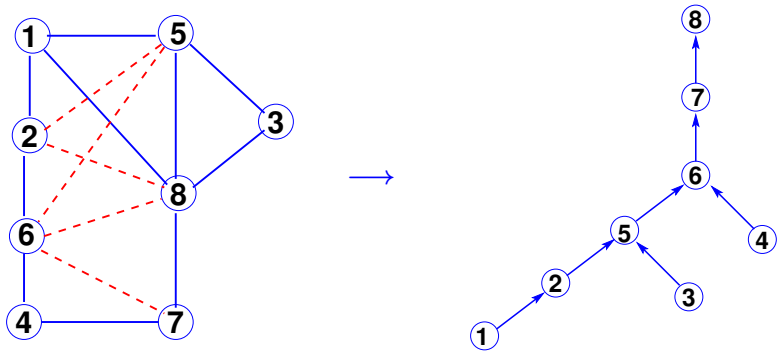


Filled matrix+graph





Corresponding Elimination Tree

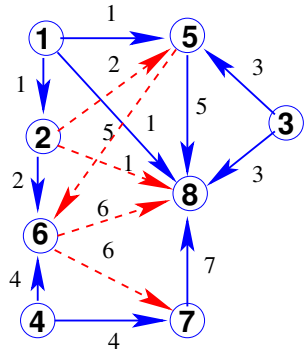


- Parent(i) = 'first nonzero entry in L(i+1:n,i)'
- Parent(i) = min $\{j > i \mid j \in Adj_{G^F}(i)\}$

Where does the elimination tree come from?

- Answer in the form of an excercise.

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 $1 \rightarrow 2 \rightarrow 5$.



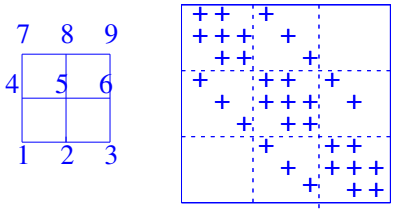
To do: Remove all the redundant dependencies.. What is the result?

Facts about elimination trees

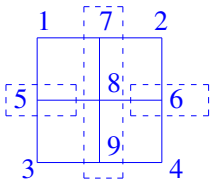
- Elimination Tree defines dependencies between columns.
- The root of a subtree cannot be used as pivot before any of its descendents is processed.
- Elimination tree depends on ordering;
- Can be used to define 'parallel' tasks.
- For parallelism: flat and wide trees → good; thin and tall (e.g. of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

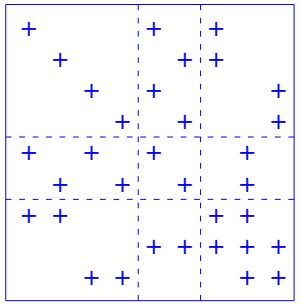
Elim. tree depends on ordering (Not just the graph)

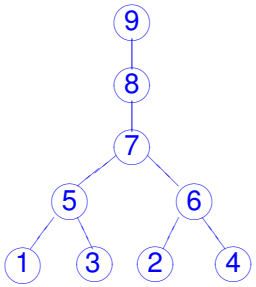
Example: 3×3 grid for 5-point stencil [natural ordering]



➤ Same example with nested dissection ordering





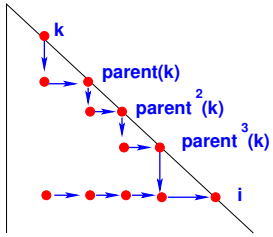


Properties

➤ 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

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



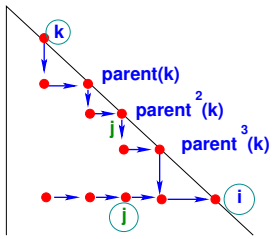
In particular: if $a_{ik} \neq 0, k < i$ then $i \rightsquigarrow k$

➤ 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.

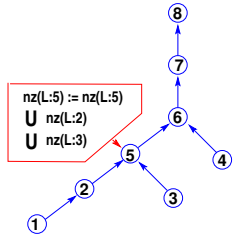
Theorem: For $i > j$, $l_{ij} \neq 0$ iff j is an ancestor of some $k \in Adj_A(i)$ in the elimination tree.



In other words:

$l_{ij} \neq 0, i > j$ iff	$\exists k \in Adj_A(i) s.t.$ $j \rightsquigarrow k$

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.

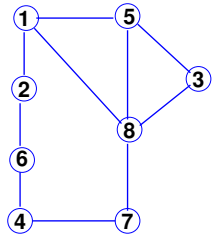
Notation :

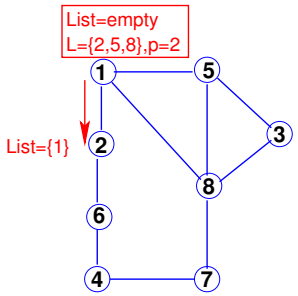
- $nz(X)$ is the pattern of X (matrix or column, or row). A set of pairs (i, j)
- $tril(X)$ = Lower triangular part of pattern [matlab notation] $\{(i, j) \in X \mid i > j\}$
- Idea: dynamically create the list of nodes needed to update $L_{:,j}$.

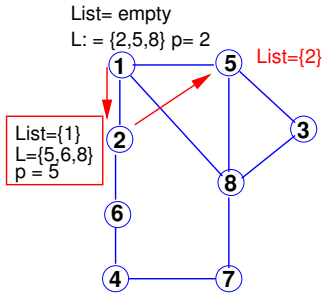
ALGORITHM : 1 . *Symbolic factorization*

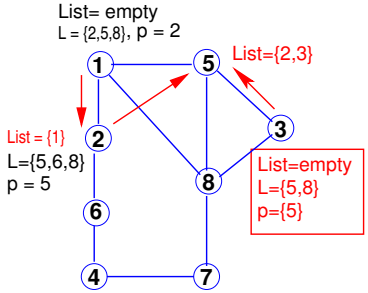
1. *Set:* $nz(L) = tril(nz(A))$,
2. *Set:* $list(j) = \emptyset, j = 1, \cdots, n$
3. *For* $j = 1 : n$
4. *for* $k \in list(j)$ *do*
5. $nz(L_{:,j}) := nz(L_{:,j}) \cup nz(L_{:,k})$
6. *end*
7. $p = \min\{i > j \mid L_{i,j} \neq 0\}$
8. $list(p) := list(p) \cup \{j\}$
9. *End*

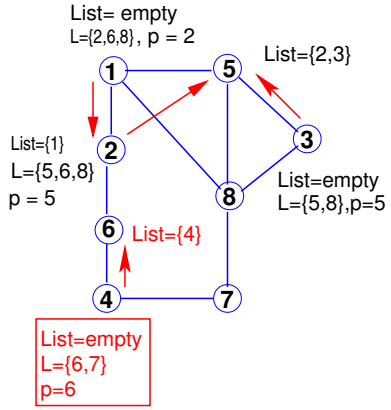
Example: Consider the earlier example:

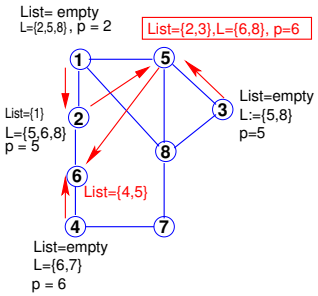


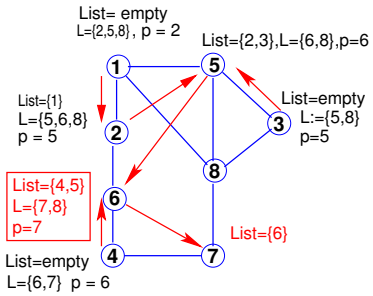












Multifrontal methods

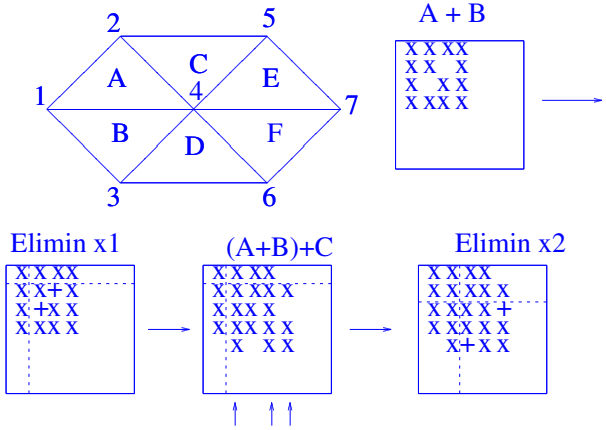
- Start with the frontal method.
- Recall: Finite element matrix: $A = \sum A^{[e]}$

$A^{[e]}$ = element matrix associated with element e.

- An old idea: Execute Gaussian elimination as the elements are being assembled
- Dependency: variabes $\leftrightarrow$ elements, creates an assembly tree.
- Method is called *the frontal method*
- Very popular among finite element users: saves storage

The origin: Frontal method (circa 1970s)

- Assemble $A + B$ then eliminate x_1
- Elimination of x_1 creates an update matrix

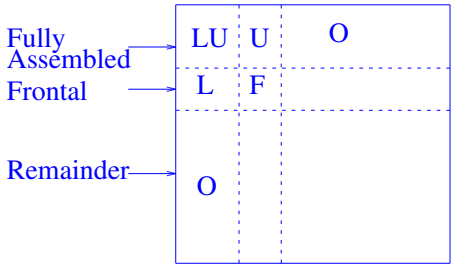


- Matrix has 3 parts:

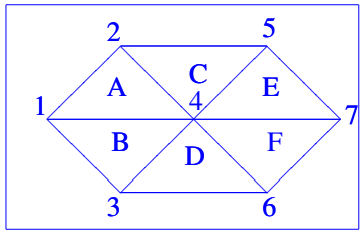
1) Fully assembled (no longer modified)

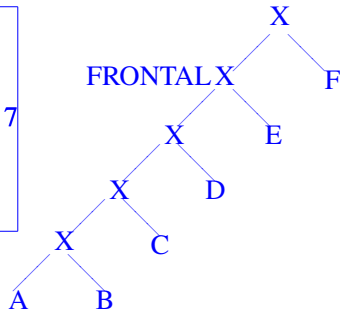
2) Frontal matrix: undergoes assembly + updates

3) Remainder: not accessed yet.



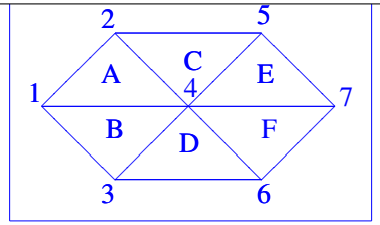
Assembly tree: - analogue to elimination tree

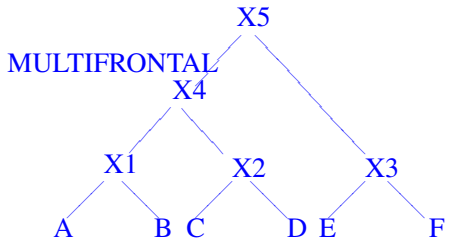




➤ Can proceed from several incoupled elements at the same time → multi-frontal technique [Duff & Reid, 1983]

Assembly tree for Multifrontal Method





Multifrontal methods: extension to general matrices

➤ Elimination tree replaces assembly tree

➤ Proceed in post-order traversal of elimination tree in order not to violate task dependencies.

➤ When a node is eliminated an update matrix is created.

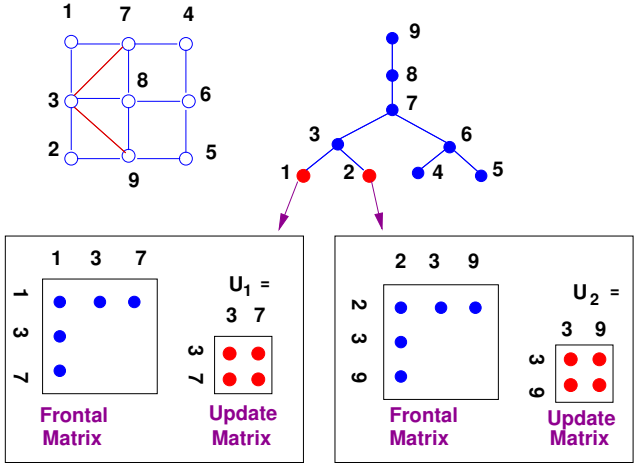
➤ This matrix is passed to the parent which adds it to its frontal matrix.

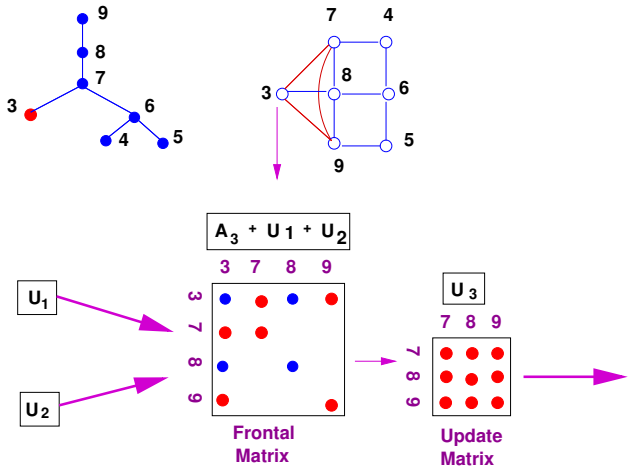
➤ Requires a stack of pending update matrices

➤ Update matrices popped out as they are needed

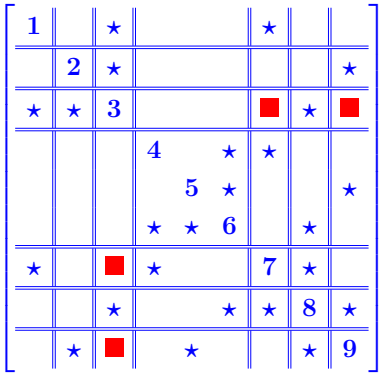
➤ Often implemented with nested dissection-type ordering

➤ More complex than a left-looking algorithm





Eliminating nodes 1 and 2: What happens on matrix



$\leftarrow U_1(3,:)$ $\leftarrow U_2(3,:)$

$\leftarrow U_1(7,:)$

$\leftarrow U_2(9,:)$

Supernodes

Columns inherit patterns of the columns from which they are updated → Many columns with same sparsity pattern. Supernode = a set of contiguous columns in the Cholesky factor L that have the same sparsity pattern.

- The set $\{j, j+1, ..., j+s\}$ is a supernode if

$$NZ(L_{*,k}) = NZ(L_{*,k+1}) \bigcup \{k+1\} \quad j \leq k < j+s$$

where $NZ(L_{*,k})$ is nonzero set of column k of L.

- Other terms used: Mass elimination, indistinguishible nodes, active variables in front, subscript compression,...

- Gain in performance due to savings in Gather-Scatter operations.

A few existing solvers (among many)

Code	Method	Scope	Developer
CHOLMOD	Left-Looking	SPD	T. Davis
MA67	Multifrontal	Symm	HSL
MA48	Right-Looking	UnSymm	HSL
SuperLU	Left-Looking	UnSymm	S. Li (LBL)
Pardiso	Left-Looking	Symm. Patt.	O. Schenk (Lugano)
MA41	Multifrontal	Symm Patt.	HSL
MUMPS	Multifrontal	Symm Patt.	Amestoy (Toulouse)
Pastix	Left+Right-Looking	Symm, symm. patt.	Labri (Bordeaux)
SuperLU_Dist	Right-Looking	UnSymm	S. Li (LBL)