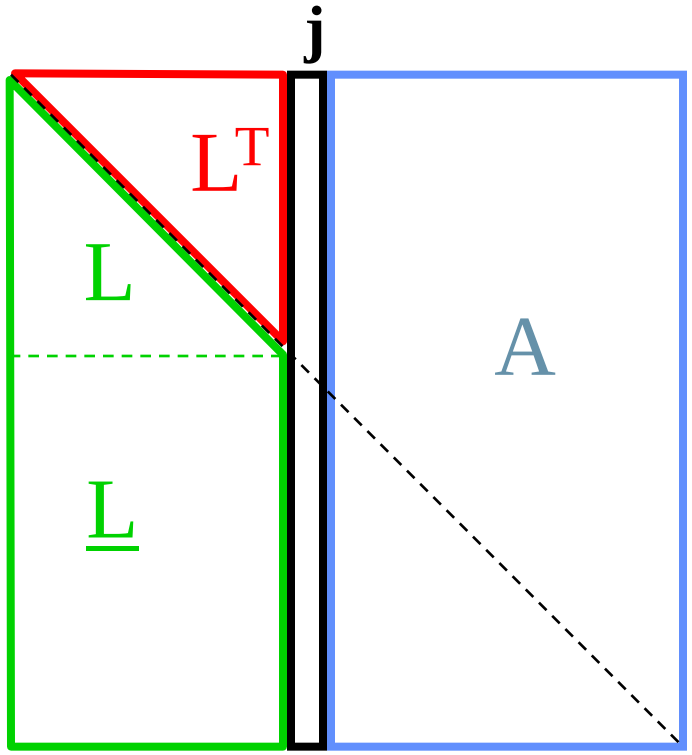
Column Cholesky Factorization

```
for j = 1 : n
   L(j:n, j) = A(j:n, j);
   for k = 1 : j-1
      % cmod(j,k)
      L(j:n, j) = L(j:n, j) – L(j, k) * L(j:n, k);
   end;
   % cdiv(j)
   L(j, j) = sqrt(L(j, j));
   L(j+1:n, j) = L(j+1:n, j) / L(j, j);
end;
```

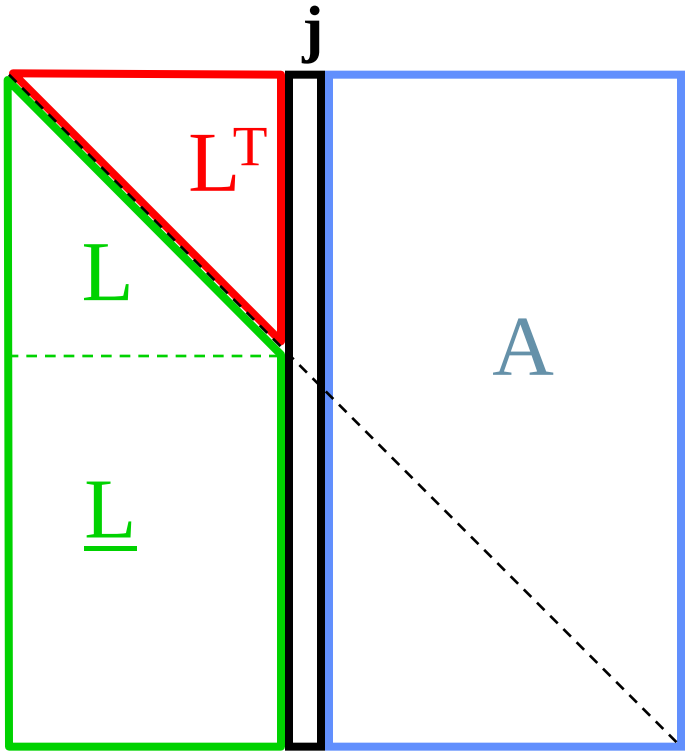


- Column j of A becomes column j of L

Sparse Column Cholesky Factorization

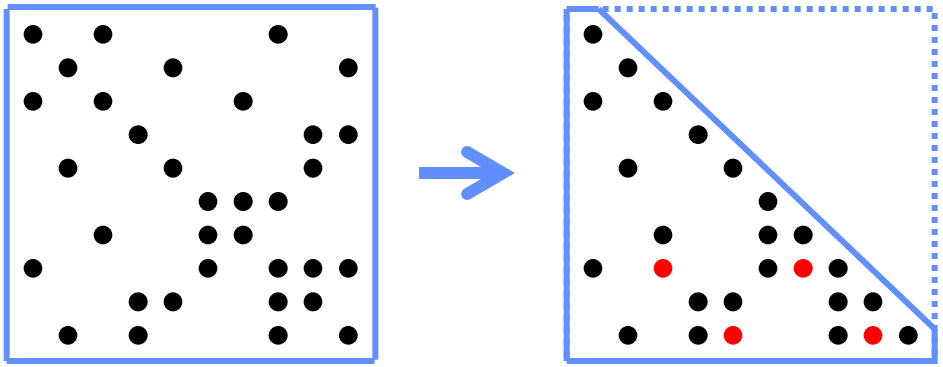
```
for j = 1 : n
   L(j:n, j) = A(j:n, j);
   for k < j with L(j, k) nonzero
      % sparse cmod(j,k)
      L(j:n, j) = L(j:n, j) – L(j, k) * L(j:n, k);
   end;

   % sparse cdiv(j)
   L(j, j) = sqrt(L(j, j));
   L(j+1:n, j) = L(j+1:n, j) / L(j, j);
end;
```

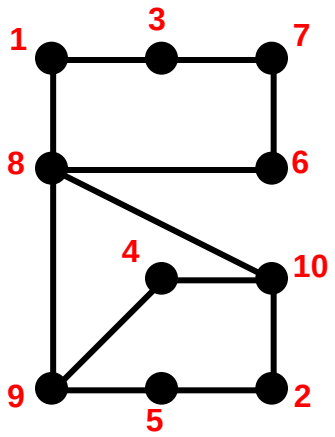


- Column j of A becomes column j of L

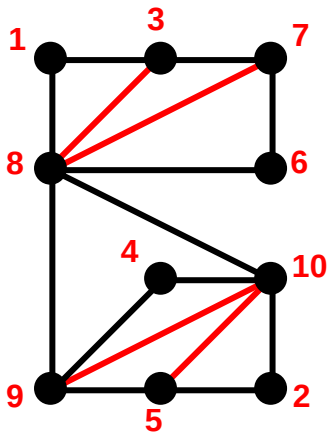
Graphs and Sparse Matrices: Cholesky factorization



Fill: new nonzeros in factor



G(A)



$G^+(A)$
[chordal]

Symmetric Gaussian elimination:

for j = 1 to n

add edges between j's higher-numbered neighbors

Cholesky Graph Game

Given an undirected graph G = G(A),

Repeat:

Choose a vertex v and mark it;

Add edges between unmarked neighbors of v;

Until every vertex is marked

Goal: End up with as few edges as possible.

Output: A labeling of the vertices with numbers 1 to n, corresponding to a symmetric permutation of matrix A.

Path lemma

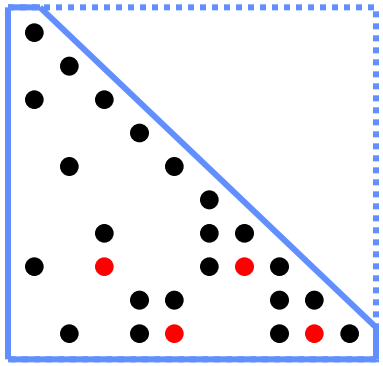
[Davis Thm 4.1]

Let $G = G(A)$ be the graph of a symmetric, positive definite matrix, with vertices $1, 2, \ldots, n$, and let $G^+ = G^+(A)$ be the filled graph.

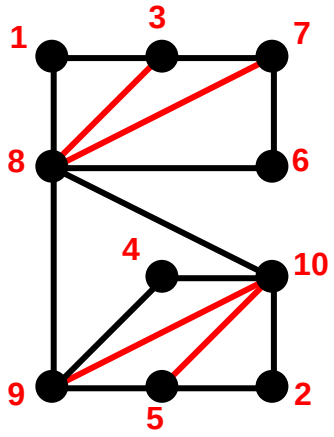
Then (v, w) is an edge of G^+ if and only if G contains a path from v to w of the form $(v, x_1, x_2, \ldots, x_k, w)$ with $x_i < \min(v, w)$ for each i.

(This includes the possibility $k = 0$, in which case (v, w) is an edge of G and therefore of G^+.)

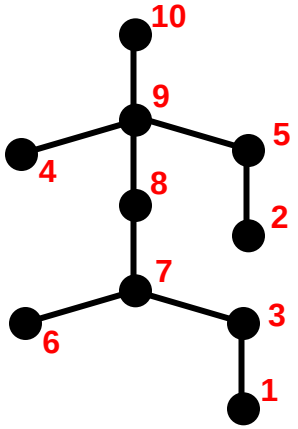
Elimination Tree



Cholesky factor



$G^+(A)$



$T(A)$

$$T(A): \quad parent(j) = \min \{ i > j : (i, j) \text{ in } G^+(A) \}$$

parent(col j) = first nonzero row below diagonal in L

- T describes dependencies among columns of factor
- Can compute $G^+(A)$ easily from T
- Can compute T from G(A) in almost linear time

Complexity measures for sparse Cholesky

- Space:
  - Measured by fill, which is nnz($G^+(A)$)
  - Number of off-diagonal nonzeros in Cholesky factor
    (need to store about n + nnz($G^+(A)$) real numbers).
  - Sum over vertices of $G^+(A)$ of (# of higher neighbors).
- Time:
  - Measured by number of flops (multiplications, say)
  - Sum over vertices of $G^+(A)$ of (# of higher neighbors)^2
- Front size:
  - Related to the amount of "fast memory" required
  - Max over vertices of $G^+(A)$ of (# of higher neighbors).

Permutations for sparsity

"I observed that most of the coefficients in our matrices were zero; i.e., the nonzeros were 'sparse' in the matrix, and that typically the triangular matrices associated with the forward and back solution provided by Gaussian elimination would remain sparse if pivot elements were chosen with care"

- Harry Markowitz, describing the 1950s work on portfolio theory that won the 1990 Nobel Prize for Economics

Cholesky Graph Game

Given an undirected graph G = G(A),

Repeat:

Choose a vertex v and mark it;

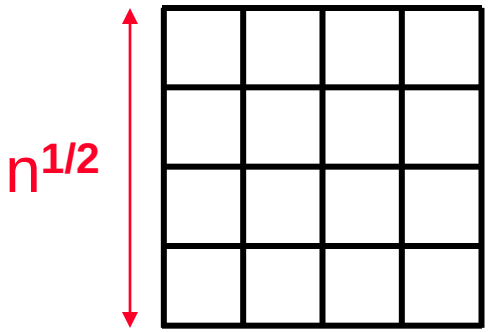
Add edges between unmarked neighbors of v;

Until every vertex is marked

Goal: End up with as few edges as possible.

Output: A labeling of the vertices with numbers 1 to n, corresponding to a symmetric permutation of matrix A.

The (2-dimensional) model problem



- Graph is a regular square grid with $n = k^2$ vertices.
- Corresponds to matrix for regular 2D finite difference mesh.
- Gives good intuition for behavior of sparse matrix algorithms on many 2-dimensional physical problems.
- There's also a 3-dimensional model problem.

Permutations of the 2-D model problem

- Theorem 1: With the natural permutation, the n-vertex model problem has exactly $O(n^{3/2})$ fill.

- Theorem 2: With a *nested dissection* permutation, the n-vertex model problem has exactly $O(n \log n)$ fill.

- Theorem 3: With any permutation, the n-vertex model problem has at least $O(n \log n)$ fill.

Nested dissection ordering

- A separator in a graph G is a set S of vertices whose removal leaves at least two connected components.

- A nested dissection ordering for an n-vertex graph G numbers its vertices from 1 to n as follows:
  - Find a separator S, whose removal leaves connected components T_1, T_2, …, T_k
  - Number the vertices of S from n-|S|+1 to n.
  - Recursively, number the vertices of each component: T_1 from 1 to $|T_1|$, T_2 from $|T_1|+1$ to $|T_1|+|T_2|$, etc.
  - If a component is small enough, number it arbitrarily.

- It all boils down to finding good separators!

Separators in practice

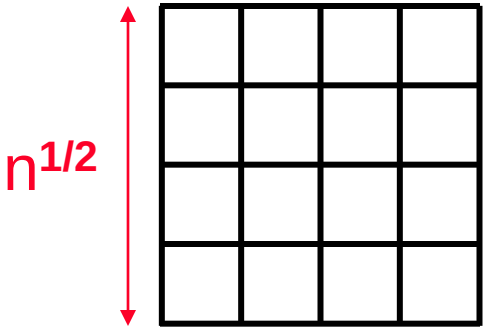
- Graph partitioning heuristics have been an active research area for many years, often motivated by partitioning for parallel computation.
- Some techniques:
  - Spectral partitioning (uses eigenvectors of Laplacian matrix of graph)
  - Geometric partitioning (for meshes with specified vertex coordinates)
  - Iterative-swapping (Kernighan-Lin, Fiduccia-Matheysses)
  - Breadth-first search (fast but dated)
- Many popular modern codes (e.g. Metis, Chaco) use multilevel iterative swapping

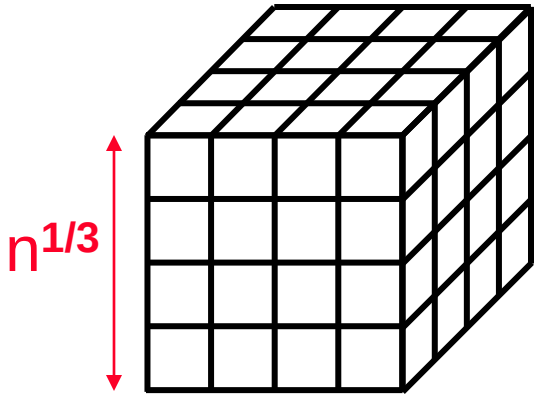
Heuristic fill-reducing matrix permutations

- Nested dissection:
  - Find a separator, number it *last*, proceed recursively
  - Theory: approx optimal separators => approx optimal fill and flop count
  - Practice: often wins for very large problems
- Minimum degree:
  - Eliminate row/col with fewest nzs, add fill, repeat
  - Hard to implement efficiently – current champion is "Approximate Minimum Degree" [Amestoy, Davis, Duff]
  - Theory: can be suboptimal even on 2D model problem
  - Practice: often wins for medium-sized problems
- Banded orderings (Reverse Cuthill-McKee, Sloan, . . .):
  - Try to keep all nonzeros close to the diagonal
  - Theory, practice: often wins for "long, thin" problems
- The best modern general-purpose orderings are ND/MD hybrids.

Complexity of direct methods

Time and space to solve any problem on any well-shaped finite element mesh





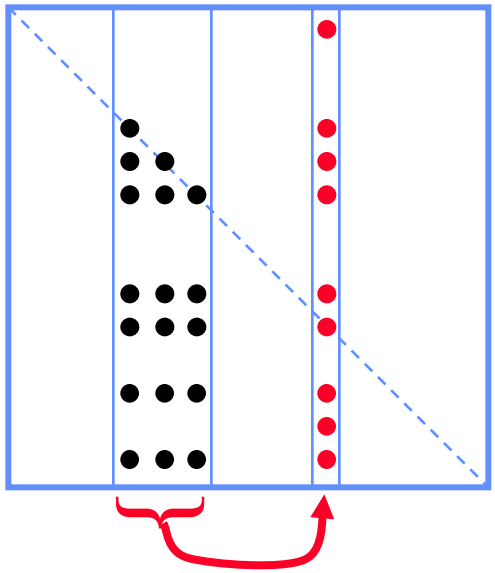
	2D	3D
Space (fill):	$O(n \log n)$	$O(n^{4/3})$
Time (flops):	$O(n^{3/2})$	$O(n^{2})$

Sparse Cholesky factorization to solve $Ax = b$

1. Preorder: replace A by PAP^T and b by Pb
   - Independent of numerics
2. Symbolic Factorization: build static data structure
   - Elimination tree
   - Nonzero counts
   - Supernodes
   - Nonzero structure of L
3. Numeric Factorization: $A = LL^T$
   - Static data structure
   - Supernodes use BLAS3 to reduce memory traffic
4. Triangular Solves: solve $Ly = b$, then $L^Tx = y$

Symmetric supernodes for Cholesky

- Supernode = group of adjacent columns of L with same nonzero structure
- Related to clique structure of filled graph $G^+(A)$

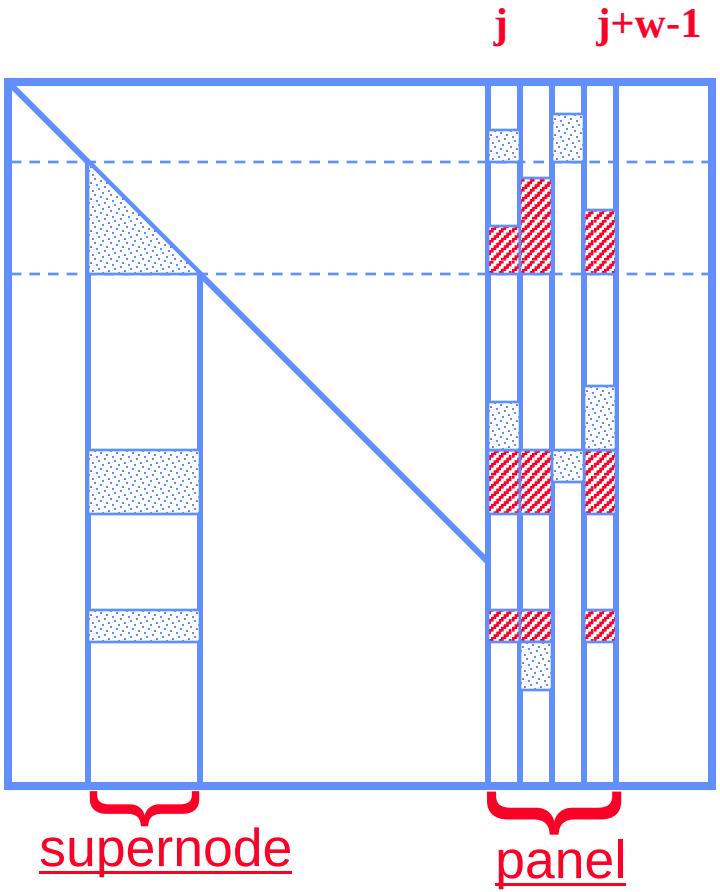


- Supernode-column update: k sparse vector ops become
  1 dense triangular solve
  + 1 dense matrix * vector
  + 1 sparse vector add
- Sparse BLAS 1 => Dense BLAS 2
- Only need row numbers for first column in each supernode
- For model problem, integer storage for L is O(n) not O(n log n)

Supernode-Panel Updates

for each panel do

- Symbolic factorization:
  which supernodes update the panel;
- Supernode-panel update:
  for each updating supernode do
    for each panel column do
      supernode-column update;
- Factorization within panel:
  use supernode-column algorithm



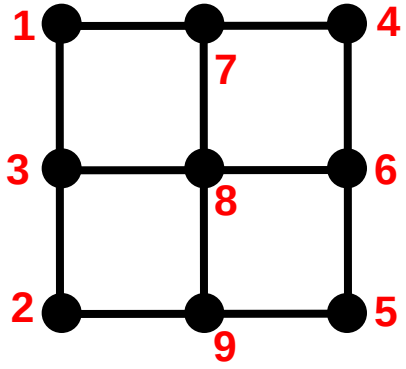
+: "BLAS-2.5" replaces BLAS-1

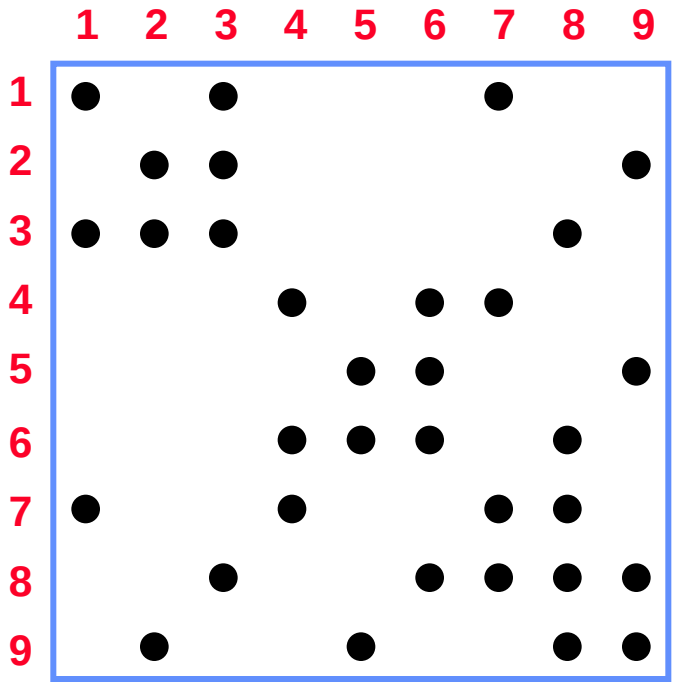
-: Very big supernodes don't fit in cache

=> 2D blocking of supernode-column updates

Symmetric-pattern multifrontal factorization

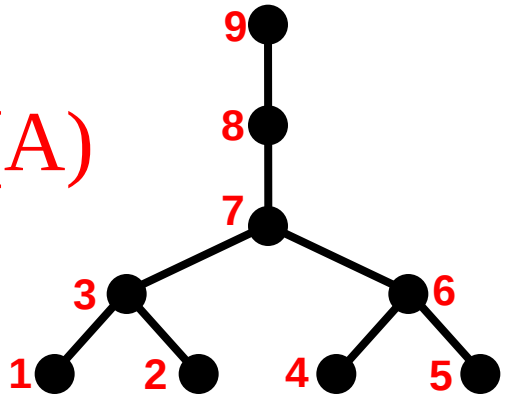
G(A)





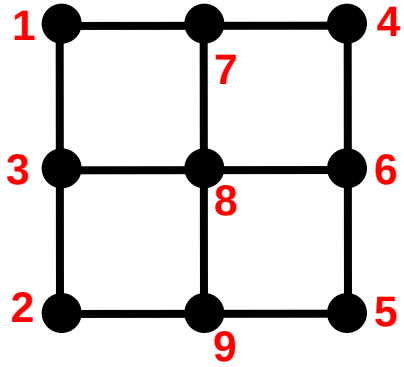
A

T(A)



Symmetric-pattern multifrontal factorization

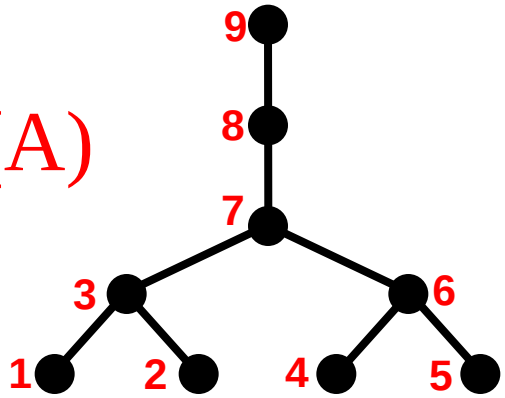
G(A)



For each node of T from leaves to root:

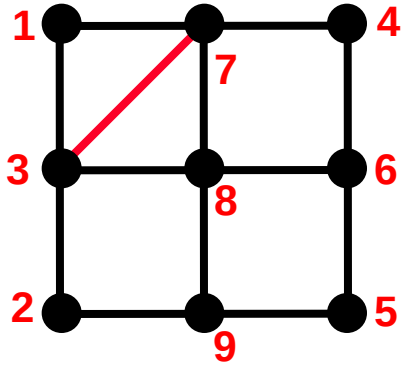
- Sum own row/col of A with children's *Update* matrices into *Frontal* matrix
- Eliminate current variable from *Frontal* matrix, to get *Update* matrix
- Pass *Update* matrix to parent

T(A)



Symmetric-pattern multifrontal factorization

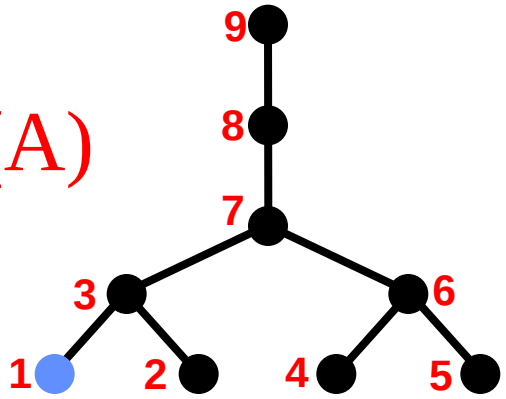
G(A)

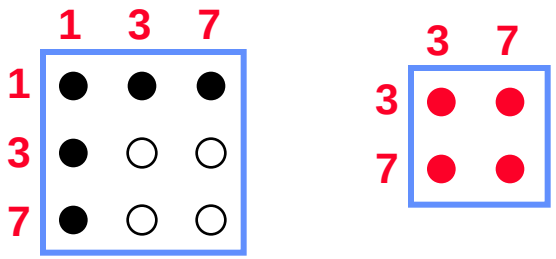


For each node of T from leaves to root:

- Sum own row/col of A with children's *Update* matrices into *Frontal* matrix
- Eliminate current variable from *Frontal* matrix, to get *Update* matrix
- Pass *Update* matrix to parent

T(A)

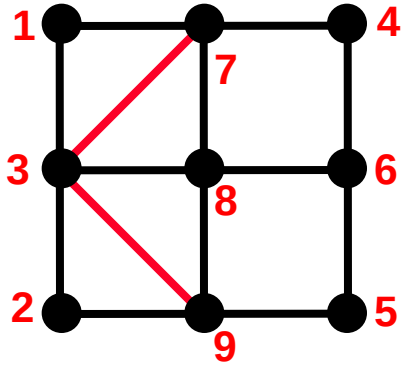




$F_1 = A_1$ => U_1

Symmetric-pattern multifrontal factorization

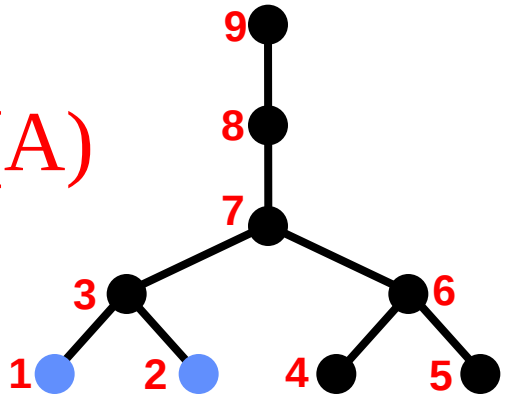
G(A)

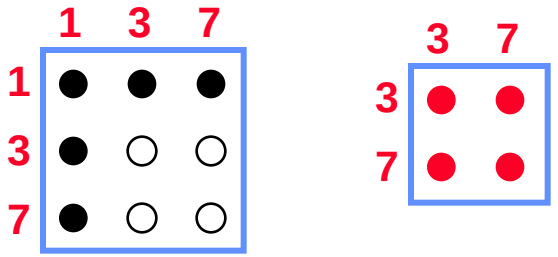


For each node of T from leaves to root:

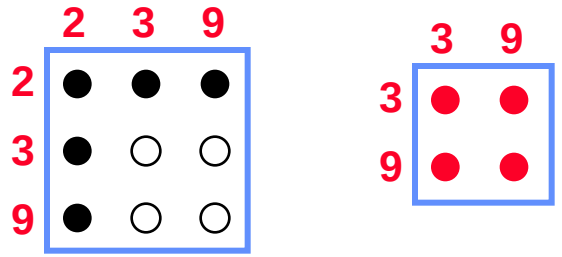
- Sum own row/col of A with children's *Update* matrices into *Frontal* matrix
- Eliminate current variable from *Frontal* matrix, to get *Update* matrix
- Pass *Update* matrix to parent

T(A)



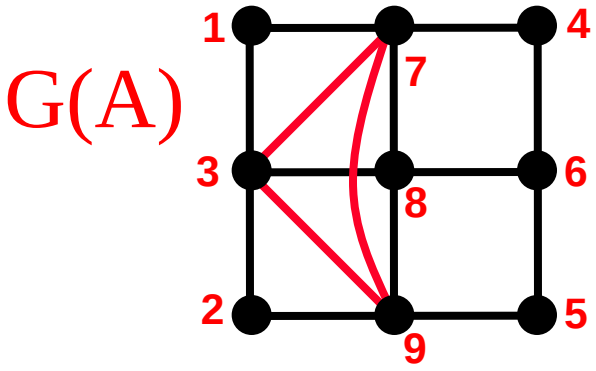


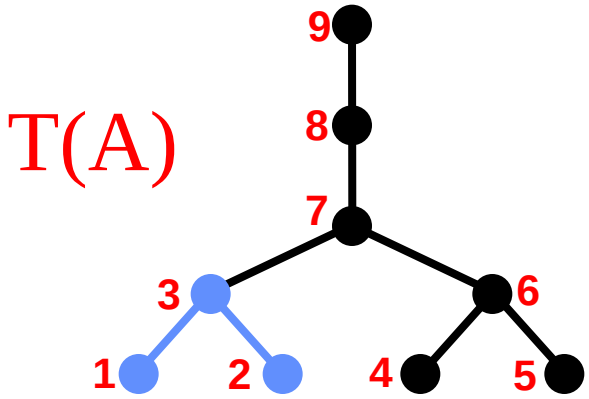
$F_1 = A_1$ $\Rightarrow U_1$

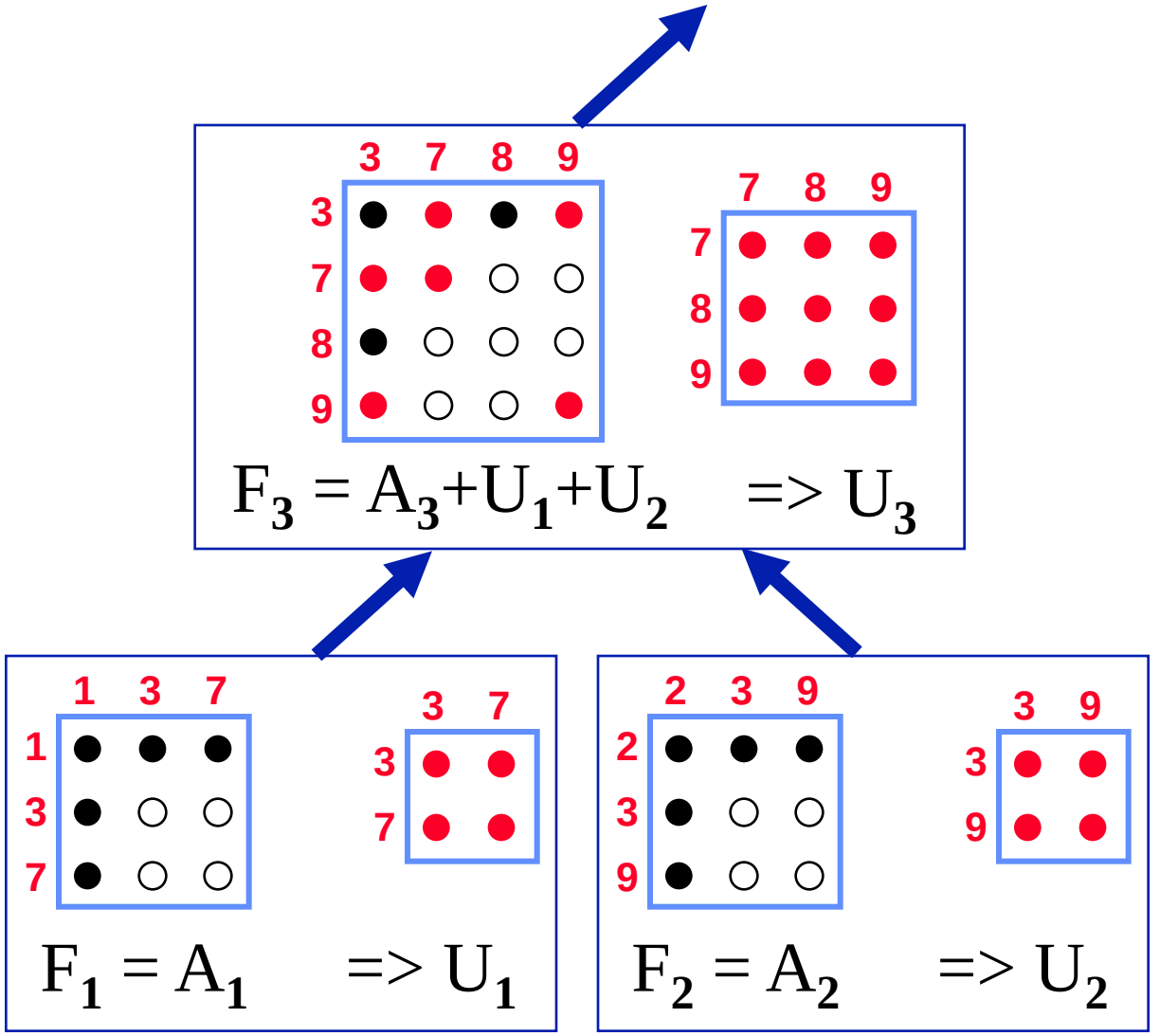


$F_2 = A_2$ $\Rightarrow U_2$

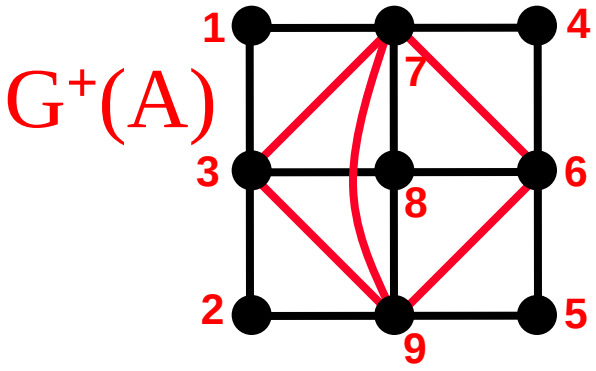
Symmetric-pattern multifrontal factorization

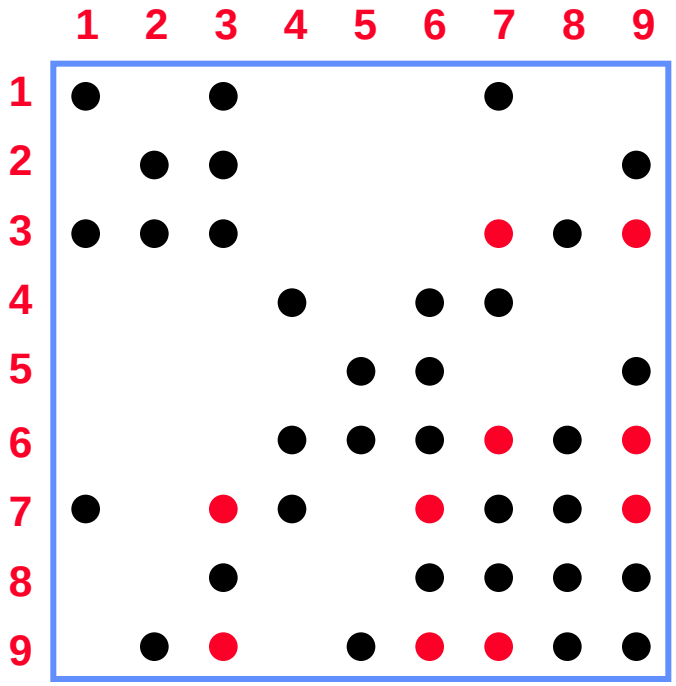




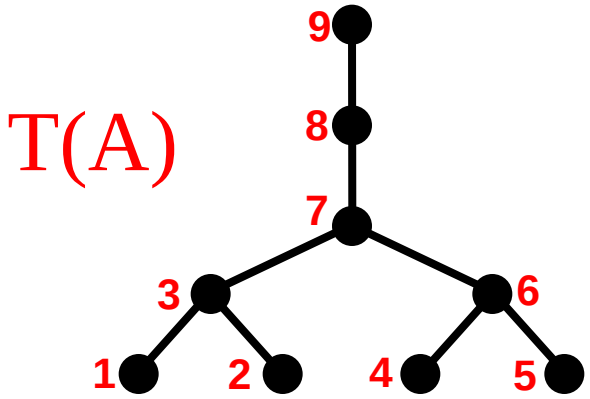


Symmetric-pattern multifrontal factorization

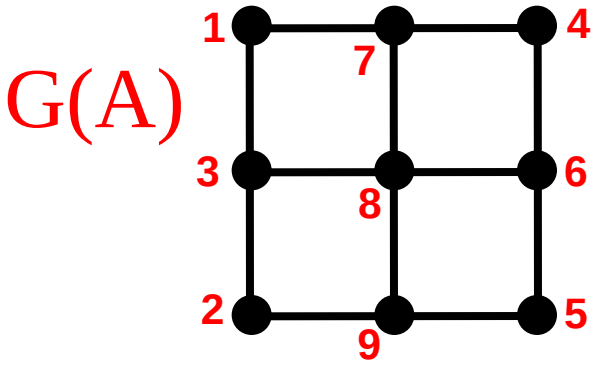


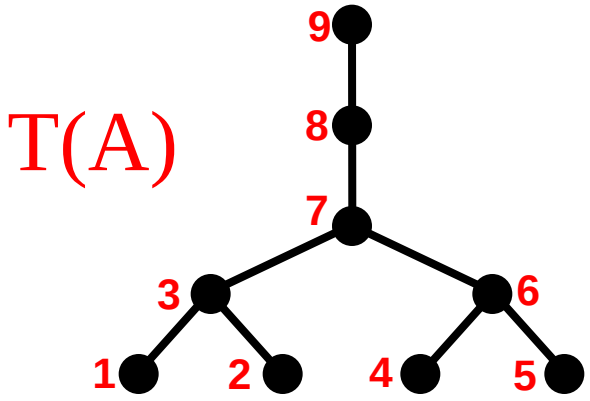


L+U



Symmetric-pattern multifrontal factorization





- Really uses supernodes, not nodes
- All arithmetic happens on dense square matrices.
- Needs extra memory for a stack of pending update matrices
- Potential parallelism:
  1. between independent tree branches
  2. parallel dense ops on frontal matrix

MUMPS: distributed-memory multifrontal

[Amestoy, Duff, L'Excellent, Koster, Tuma]

- Symmetric-pattern multifrontal factorization
- Parallelism both from tree and by sharing dense ops
- Dynamic scheduling of dense op sharing
- Symmetric preordering
- For nonsymmetric matrices:
  - optional weighted matching for heavy diagonal
  - expand nonzero pattern to be symmetric
  - numerical pivoting only within supernodes if possible (doesn't change pattern)
  - failed pivots are passed up the tree in the update matrix