Direct Methods for Sparse Matrices

Miroslav Tůma

Institute of Computer Science

Academy of Sciences of the Czech Republic

and

Technical University in Liberec

This is a lecture prepared for the SEMINAR ON NUMERICAL ANALYSIS: Modelling and Simulation of Challenging Engineering Problems, held in Ostrava, February 7–11, 2005. Its purpose is to serve as a first introduction into the field of direct methods for sparse matrices. Therefore, it covers only the most classical results of a part of the field, typically without citations. The lecture is a first of the three parts which will be presented in the future. The contents of subsequent parts is indicated in the outline.

The presentation is partially supported by the project within the National Program of Research "Information Society" under No. 1ET400300415

Outline

1. Part I.
2. Sparse matrices, their graphs, data structures
3. Direct methods
4. Fill-in in SPD matrices
5. Preprocessing for SPD matrices
6. Algorithmic improvements for SPD decompositions
7. Sparse direct methods for SPD matrices
8. Sparse direct methods for nonsymmetric matrices

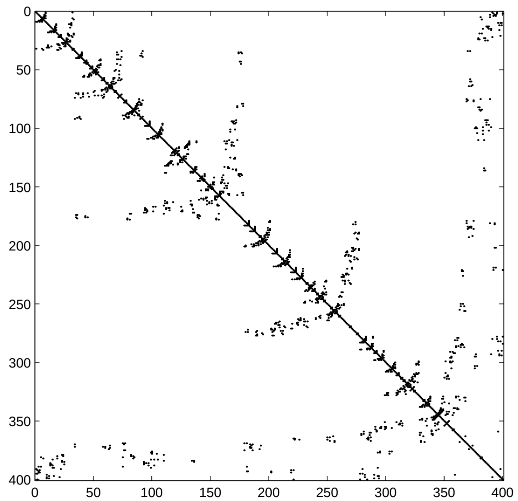
Outline (continued)

1. Part II. (not covered here)
2. Fill-in in LU decomposition
3. Reorderings for LU decomposition
4. LU decompositions based on partial pivoting
5. LU decompositions based on full/relaxed pivoting
6. Part III. (not covered here)
7. Parallel sparse direct methods
8. Parallel SPD sparse direct methods
9. Parallel nonsymetric sparse direct methods
10. Sparse direct methods: sequential and parallel codes

1. Sparse matrices, their graphs, data structures

1.a) Concept of sparse matrices: introduction

Definition 1 *Matrix* $A \in \mathbb{R}^{m \times n}$ *is said to be sparse if it has* $O(\min\{m, m\})$ *entries.*



1.a) Concept of sparse matrices: other definitions

Definition 2 *Matrix* $A \in \mathbb{R}^{m \times n}$ *is said to be sparse if it has row counts bounded by* $r_{max} << n$ *or column counts bounded by* $c_{max} << n$.

Definition 3 *Matrix* $A \in \mathbb{R}^{m \times n}$ *is said to be sparse if its number of nonzero entries is* $O(n^{1+\gamma})$ *for some* $\gamma < 1$.

Definition 4 *(**pragmatic definition: J.H. Wilkinson**) Matrix* $A \in \mathbb{R}^{m \times n}$ *is said to be sparse if we can exploit the fact that a part of its entries is equal to zero.*

1.a) Concept of sparse matrices: an example showing importance of the small exponent γ for $n = 10^4$

γ	$n^{1+\gamma}$
0.1	25119
0.2	63096
0.3	158489
0.4	398107
0.5	1000000

1.b) Matrices and their graphs: introduction

Matrices, or their structures (i.e., **positions of nonzero entries**) can be conveniently expressed by graphs

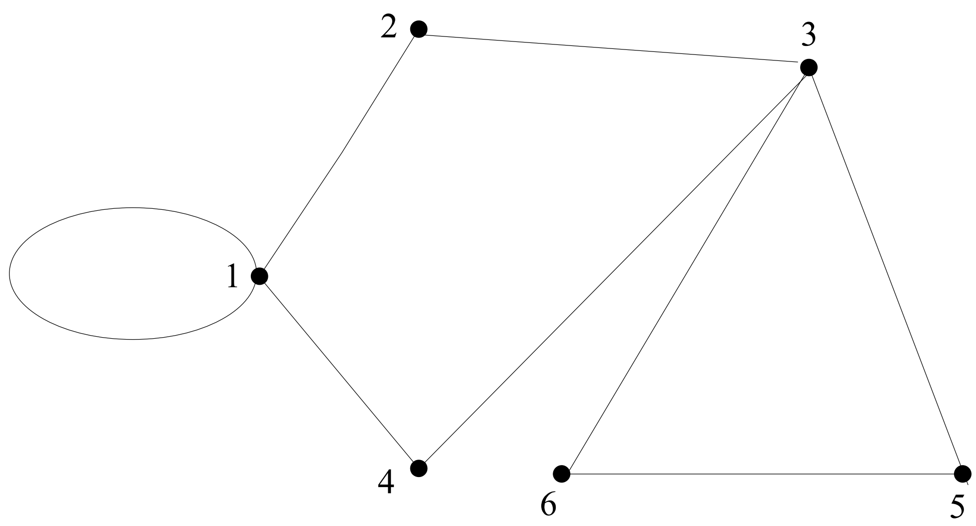
$$\Downarrow$$

Different graph models for different purposes

- undirected graph
- directed graph
- bipartite graph

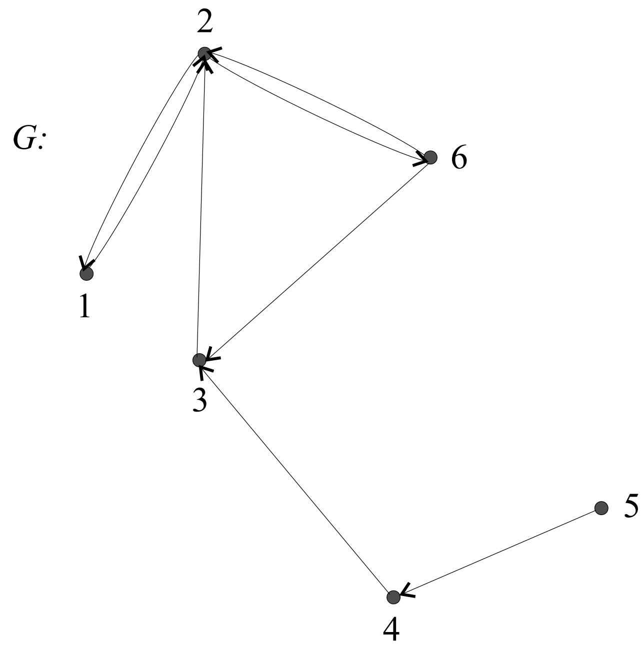
1.b) Matrices and their graphs: undirected graphs

Definition 5 *A simple undirected graph is an ordered pair of sets* (V, E) *such that* $E = \{\{i, j\} | i \in V, j \in V\}$. V *is called the* **vertex (node)** *set and* E *is called the* **edge** *set.*



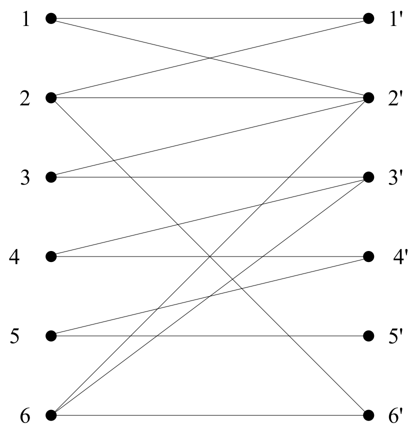
1.b) Matrices and their graphs: directed graphs

Definition 6 *A simple directed graph is an ordered pair of sets* (V, E) *such that* $E = \{(i,j) | i \in V, j \in V\}$. V *is called the* **vertex (node)** *set and* E *is called the* **edge (arc)** *set.*



1.b) Matrices and their graphs: bipartite graphs

Definition 7 *A simple bipartite graph is an ordered pair of sets* (R, C, E) *such that* $E = \{\{i, j\} | i \in R, j \in C\}$. R *is called the* **row vertex** *set,* C *is called the* **column vertex** *set and* E *is called the* **edge** *set.*



1.b) Matrices and their graphs: relation **matrix** $\rightarrow$ **graph**

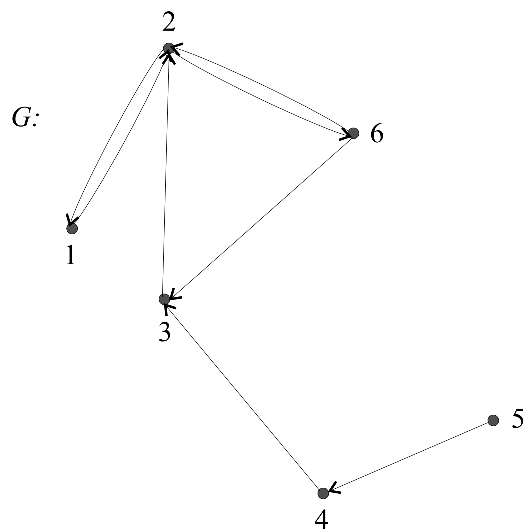
Definition 8

$\{x, y\} \in E$ *or* $(x, y) \in E \Leftrightarrow$ *vertices* x *and* y *are* **adjacent**

$Adj(x) = \{y | y$ and x are adjacent $\}$

Structure of a nonsymmetric matrix and its graph

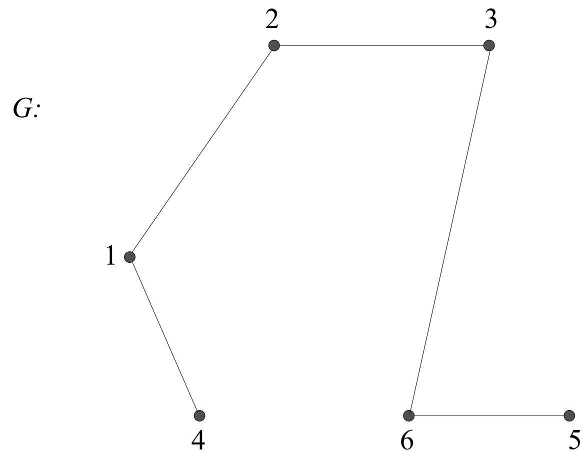
$$\begin{pmatrix} * & * & & & & \\ * & * & & & & * \\ & * & * & & & \\ & & * & * & & \\ & & & * & * & \\ & * & * & & & * \end{pmatrix}$$



1.b) Matrices and their graphs: relation **matrix** $\rightarrow$ **graph**

Structure of a symmetric matrix and its graph

$$\begin{pmatrix} * & * & & * & & \\ * & * & * & & & \\ & * & * & & & * \\ * & & & * & & \\ & & & & * & * \\ & & * & & * & * \end{pmatrix}$$



1.c) Data structures for sparse matrices: sparse vectors

$$v = \begin{pmatrix} 3.1 & 0 & 2.8 & 0 & 0 & 5 & 4.3 \end{pmatrix}^T$$

AV	3.1	2.8	5	4.3

JV	1	3	6	7

1.c) Data structures for sparse matrices: static data structures

- **static**: difficult/costly entry insertion, deletion
- **CSR** (Compressed Sparse by Rows) format: stores matrix rows as sparse vectors one after another
- **CSC**: analogical format by columns
- **connection tables**: 2D array with n rows and m colums where m denotes maximum count of a row of the stored matrix
- A lot of other general / specialized formats

1.c) Data structures for sparse matrices: CSR format example

$$\begin{pmatrix} 3.1 & & 2.8 & \\ & & 5 & 4.3 \\ 2 & 6 & & 1 \\ 1 & & & \end{pmatrix}^T$$

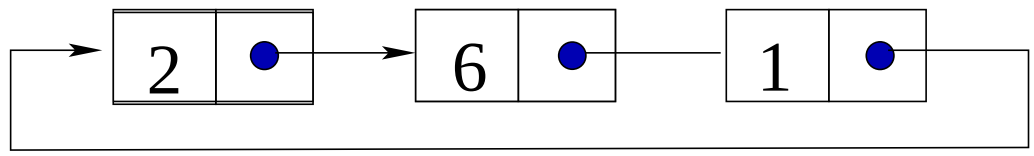
A	3.1	2.8	5	4.3	2	6	1	1

JA	1	3	3	4	1	2	4	1

IA	1	3	5	8	9

1.c) Data structures for sparse matrices: dynamic data structures

- **dynamic**: easy entry insertion, deletion
- **linked list - based** format: stores matrix rows/columns as items connected by pointers
- linked lists can be cyclic, one-way, two-way
- **rows/columns** embedded into a larger array: emulated dynamic behavior



1.c) Data structures for sparse matrices: static versus **dynamic** data structures

- dynamic data structures:
- – more flexible but this flexibility might **not be needed**
- – difficult to **vectorize**
- – difficult to keep **spatial locality**
- – used preferably for storing vectors
- static data structures:
- – we need to avoid ad-hoc insertions/deletions
- – much simpler to vectorize
- – efficient access to rows/columns

2. Direct methods

2.a) Dense direct methods: introduction

- methods based on solving $Ax = b$ by a matrix decomposition – variant of Gaussian elimination; typical decompositions:
- – $A = LL^T, A = LDL^T$ (Cholesky decomposition, LDL^T decomposition for SPD matrices)
- – $A = LU$ (LU decomposition for general nonsymmetric matrices)
- – $A = LBL^T$ (symmetric indefinite / diagonal pivoting decomposition for A symmetric indefinite)

three steps of a (basic!) direct method:

1) $A \rightarrow LU$, 2) y from $Ly = b$, 3) x from $Ux = y$

2.a) Dense direct methods: elimination versus decomposition

- Householder (end of 1950's, beginning of 1960's): expressing Gaussian elimination as a decomposition
- Various reformulations of the same decomposition: different properties in
- – **sparse implementations**
- – **vector processing**
- – **parallel implementations**
- We will show six basic algorithms but there are others (bordering, Dongarra-Eisenstat)

Algorithm 1 **ikj** *lu decomposition (delayed row dense algorithm)*

$l = I_n$

$u = O_n$

$u_{11:n} = a_{1,1:n}$

for i=2:n

  for k=1:i-1

    $l_{ik} = a_{ik}/a_{kk}$

    for j=k+1:n

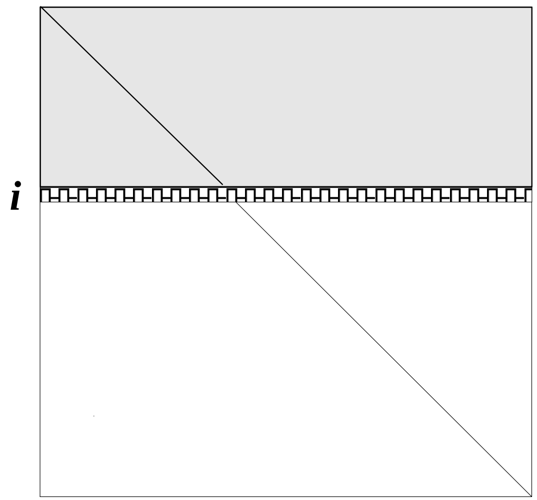
      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$

    end

  end

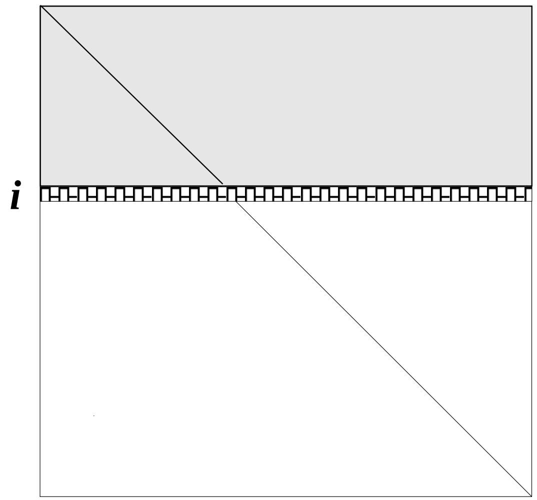
  $u_{ii:n} = a_{ii:n}$

end



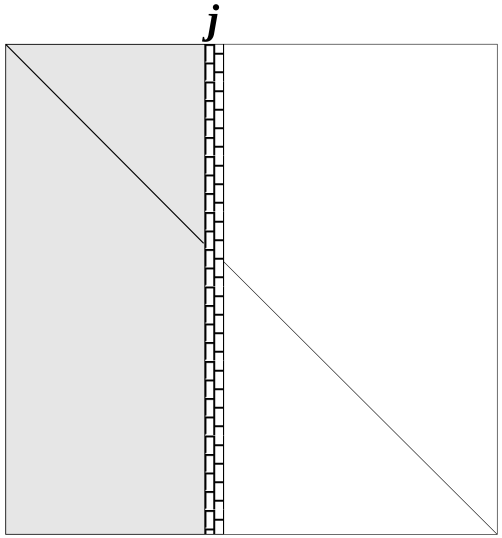
Algorithm 2 ijk *lu decomposition (dot product - based row dense algorithm)*

$l = I_n$, $u = O_n$, $u_{11:n} = a_{11:n}$
for i=2:n
  for j=2:i
    $l_{ij-1} = a_{ij-1}/a_{j-1j-1}$
    for k=1:j-1
      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
    end
  end
  for j=i+1:n
    for k=1:i-1
      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
    end
  end
  $u_{i,i:n} = a_{i,i:n}$
end



Algorithm 3 jki *lu decomposition (delayed column dense algorithm)*

$l = I_n$, $u = O_n$, $u_{11} = a_{11}$
for j=2:n
  for s=j:n
    $l_{sj-1} = a_{sj-1}/a_{j-1j-1}$
  end
  for k=1:j-1
    for i=k+1:n
      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$
    end
  end
  $u_{1:jj} = a_{1:jj}$
end



Algorithm 4 jik *lu decomposition (dot product - based column dense algorithm)*

$l = I_n$, $u_{11} = a_{11}$

for j=2:n

  for s=j:n

    $l_{sj-1} = a_{sj-1}/a_{j-1j-1}$

  end

  for i=2:j

    for k=1:i-1

      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$

    end

  end

  for i=j+1:n

    for k=1:j-1

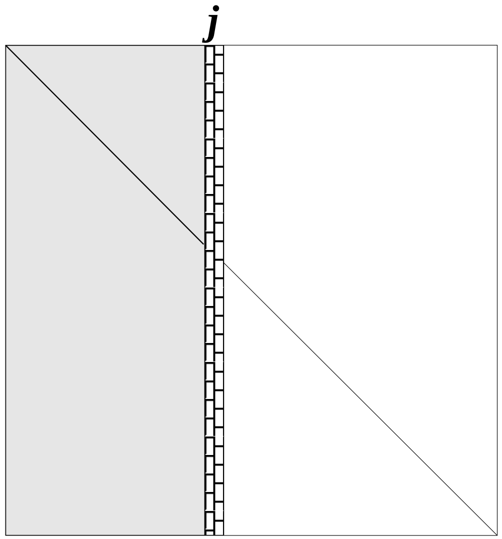
      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$

    end

  end

  $u_{1:jj} = a_{1:jj}$

end



Algorithm 5 kij *lu decomposition (row oriented submatrix dense algorithm)*

$l = I_n$

$u = O_n$

for k=1:n-1

  for i=k+1:n

    $l_{ik} = a_{ik}/a_{kk}$

    for j=k+1:n

      $a_{ij} = a_{ij} - l_{ik} * a_{kj}$

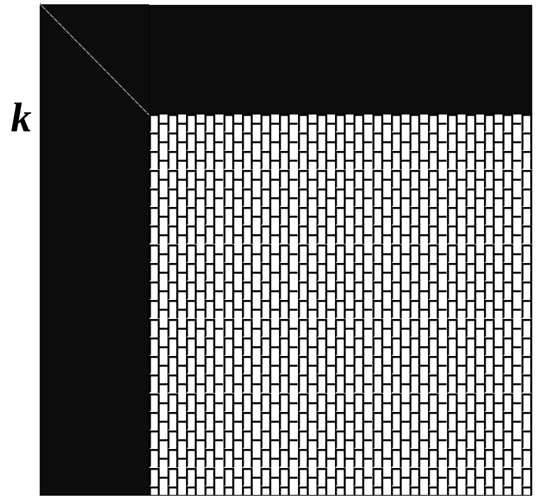
    end

  end

  $u_{kk:n} = a_{kk:n}$

end

$u_{nn} = a_{nn}$



Algorithm 6 kji *lu decomposition (column oriented submatrix dense algorithm)*

$l = I_n$, $u = O_n$

for k=1:n-1

for s=k+1:n

$l_{sk} = a_{s,k}/a_{k,k}$

end

for j=k+1:n

for i=k+1:n

$a_{ij} = a_{ij} - l_{ik} * a_{kj}$

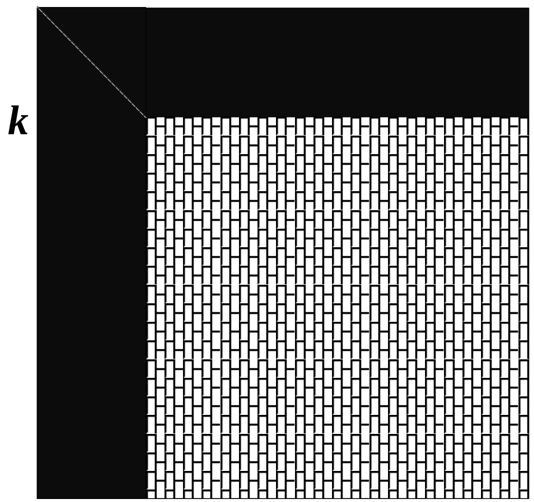
end

end

$u_{kk:n} = a_{kk:n}$

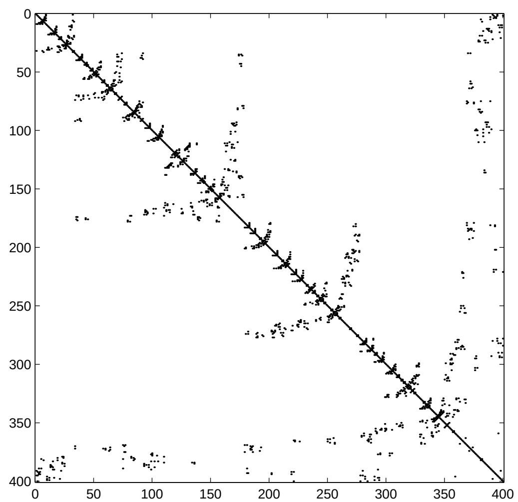
end

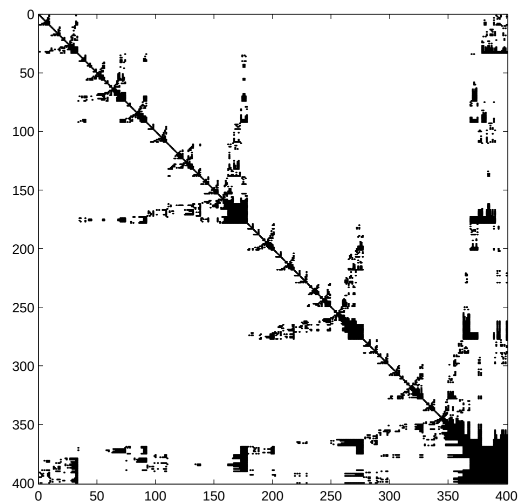
$u_{nn} = a_{nn}$



2.b) Sparse direct methods: existence of fill-in

- Not all the algorithms equally desirable when A is **sparse**
- The problem: **sparsity structure** of $L+U$ ($L+L^T$, $L+B+L^T$) do not need to be the same as the sparsity structure of A: new nonzeros (**fill-in**) may arise





2.b) Sparse direct methods: fill-in

- Arrow matrix

$$\begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix} \qquad \begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix}$$

- How to **describe** the fill-in
- How to **avoid** it

2.b) Sparse direct methods: fill-in description

Definition 9 *Sequence of elimination matrices:* $A^{(0)} \equiv A$, $A^{(1)}$, $A^{(2)}$, ..., $A^{(n)}$: **computed** *entries from factors replace original (zero and nonzero) entries of* A.

- Local description of fill-in using the matrix structure (entries of elimination matrices denoted with superscripts in parentheses)
- Note that we use the **non-cancellation assumption**

Lemma 1 (**fill-in lemma**) *Let* $i > j, k < n$. *Then*
$a_{ij}^{(k)} \neq 0 \Leftrightarrow a_{ij}^{(k-1)} \neq 0$ *or* $(a_{ik}^{(k-1)} \neq 0 \wedge a_{kj}^{(k-1)} \neq 0)$

2.b) Sparse direct methods: fill-in illustration for a symmetric matrix

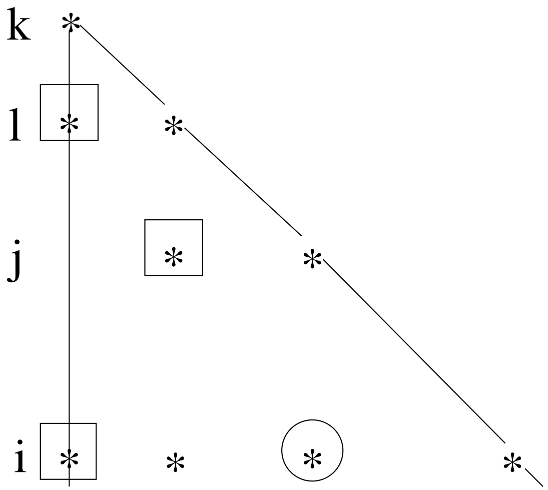
$$
\begin{array}{cccc}
 & k & j & i \\
k & * & * & * \\
j & * & * & \bigcirc \\
i & * & \bigcirc & *
\end{array}
$$

2.b) Sparse direct methods: fill-in path theorem

- Simple global description via the graph model (follows from repeated use of fill-in lemma)

Theorem 1 *(**fill-in path theorem**) Let* $i > j, k < n$. *Then* $a_{ij}^{(k)} \neq 0 \Leftrightarrow \exists$ *a path* $x_i, x_{p_1}, \ldots, x_{p_t}, x_j$ *in* $G(A)$ *such that* $(\forall l \in \hat{t})(p_l < k)$.

2.b) Sparse direct methods: path theorem illustration for a symmetric matrix



path: i–k, k–l, l–j

3. Fill-in in SPD matrices

3.a) Fill-in description: why do we restrict to the SPD case?

- SPD case enables to separate **structural** properties of matrices from their **numerical** properties.
- SPD case is simpler and more transparent
- solving sparse SPD systems is very important
- it was historically the first case with a nontrivial insight into the mechanism of the (Cholesky) decomposition (but not the first studied case)
- SPD case enables in many aspects smooth transfer to the general nonsymmetric case

3.b) Elimination tree: introduction

- Transparent global description: based on the concept of **elimination tree** (for symmetric matrices) or **elimination directed acyclic graph** (nonsymmetric matrices)

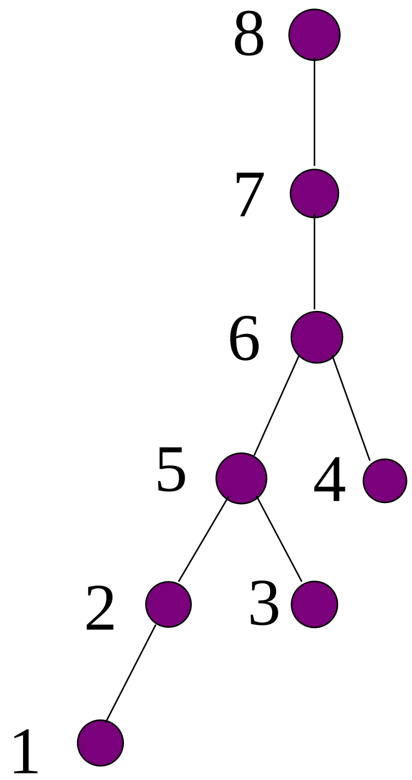
Definition 10 *Elimination tree* $T = (V, E)$ *of a symmetric matrix is a rooted tree with* $V = \{x_1, \ldots, x_n\}$, $E = \{(x_i, x_j) | x_j = \min\{k | (k > i) \wedge (l_{ik} \neq 0)\}\}$

- Note that it is defined for the structure of L
- Root of the elimination tree: vertex n
- If need we denote vertices only by their indices
- Edges in T connect vertices (i, j) such that $i < j$

3.b) Elimination tree: illustration

$$\begin{pmatrix} * & * & & & * & & & * \\ * & * & & & & * & & * \\ & & * & & * & & & * \\ & & & * & & * & * & * \\ * & & * & & * & & & * \\ & * & & * & & * & & * \\ & & & * & & & * & * \\ * & * & * & * & * & * & * & * \end{pmatrix} \qquad \begin{pmatrix} * & * & & & * & & & * \\ * & * & & & f & * & & * \\ & & * & & * & & & * \\ & & & * & & * & * & * \\ * & f & * & & * & f & & * \\ & * & & * & f & * & f & * \\ & & & * & & f & * & * \\ * & * & * & * & * & * & * & * \end{pmatrix}$$

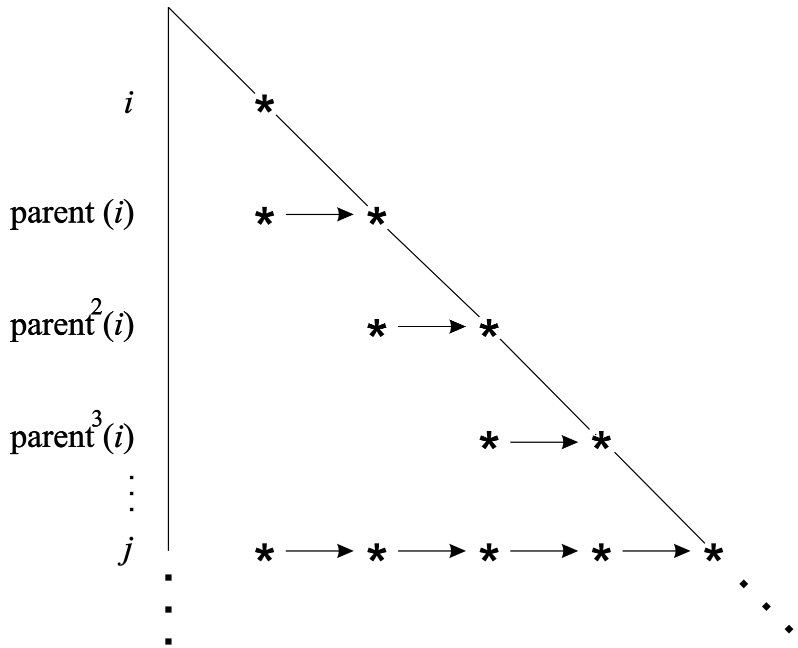
3.b) Elimination tree: illustration (II.)



terminology: **parent, child, ancestor, descendant**

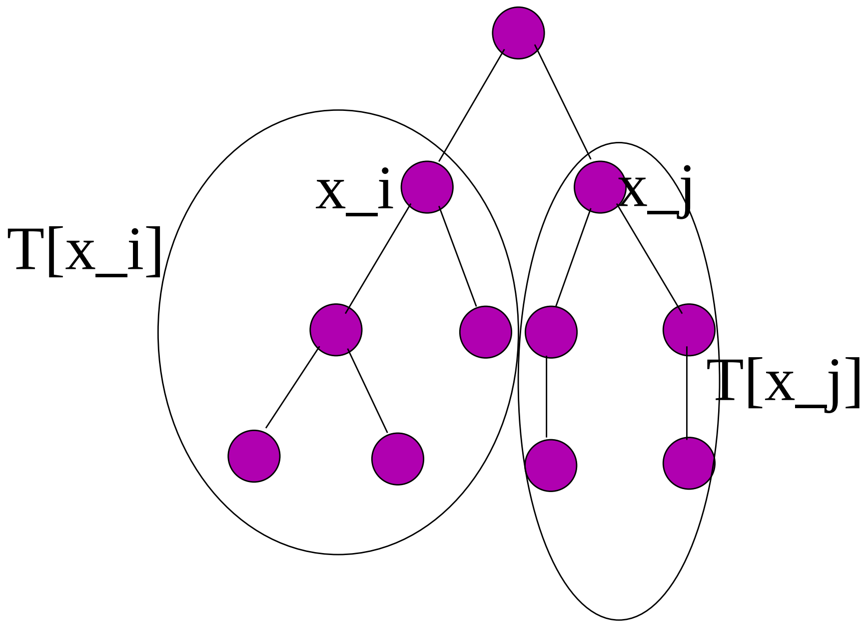
3.b) Elimination tree: necessary condition for an entry of L to be (structurally!) nonzero

Lemma 2 *If* $l_{ji} \neq 0$ *then* x_j *is an ancestor of* x_i *in the elimination tree*



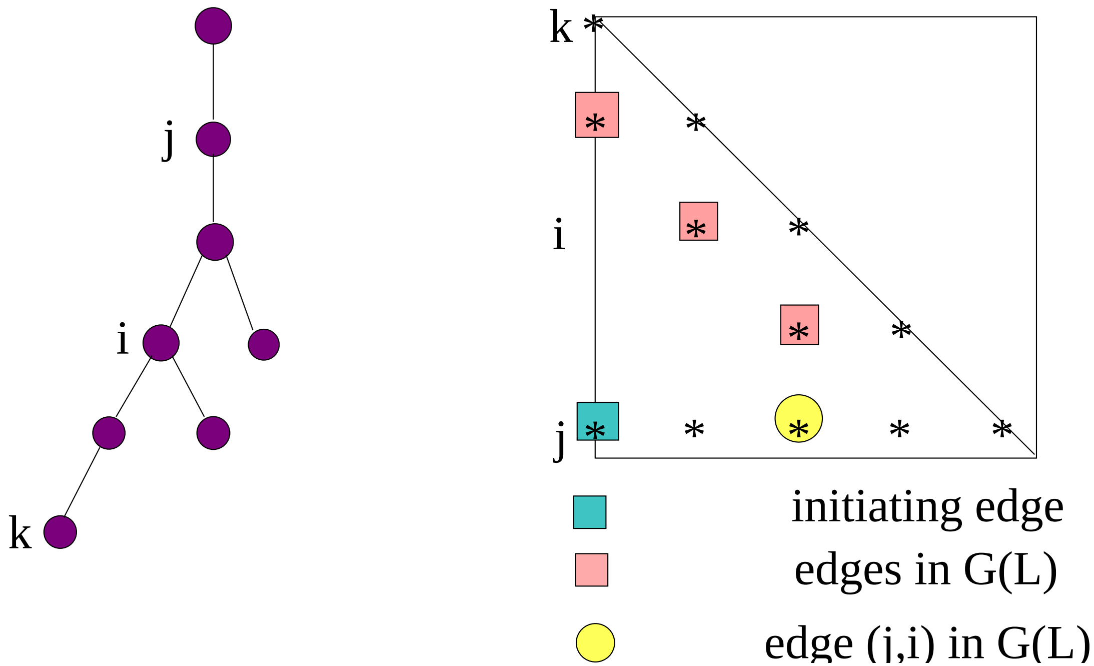
3.b) Elimination tree: natural source of parallelism

Lemma 3 *Let $T[x_i]$ and $T[x_j]$ be disjoint subtrees of the elimination tree T. Then $l_{rs} = 0$ for all $x_r \in T[x_i]$ and $x_s \in T[x_j]$.*

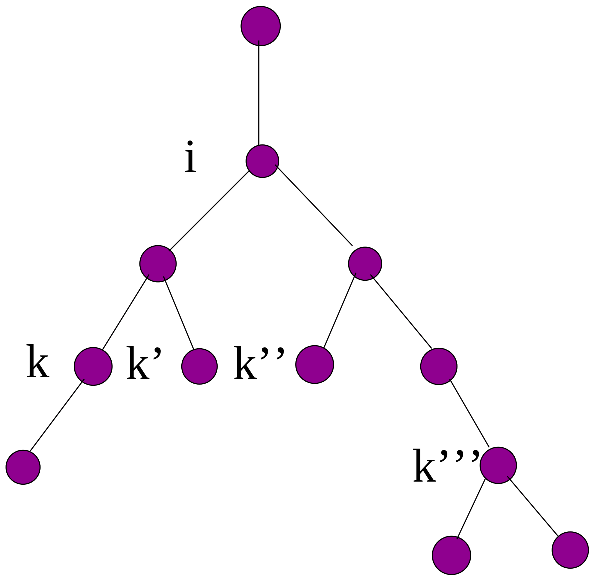


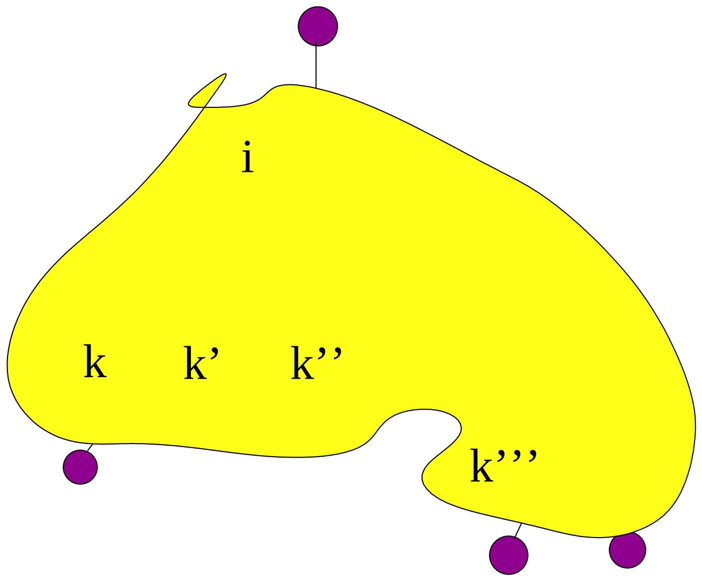
3.b) Elimination tree: full characterization of entries of L

Lemma 4 *For $j > i$ we have $l_{ji} \neq 0$ if and only if x_i is an ancestor of some x_k in the elimination tree for which $a_{jk} \neq 0$.*



3.b) Elimination tree: row structure of L is given by a row subtree of the elimination tree





3.c) Computation of row (and column) counts: algorithm

initialize all $colcounts$ to 1

for $i = 1$ to n do

  $rowcount(i) = 1$

  $mark(x_i) = i$

  for k such that $k < i \wedge a_{ik} \neq 0$ do

    $j = k$

    while $mark(x_j) \neq i$ do

      $rowcount(i) = rowcount(i) + 1$

      $colcount(j) = colcount(j) + 1$

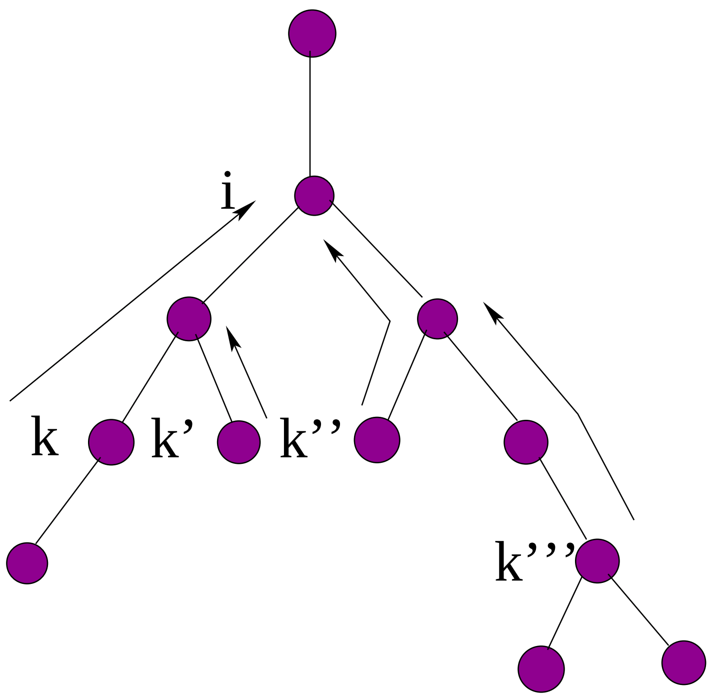
      $mark(x_j) = i$

      $j = parent(j)$

    end while

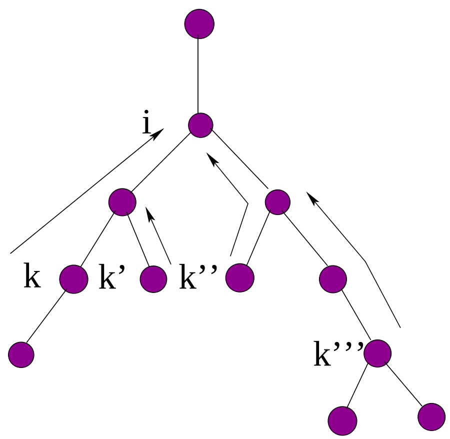
  end k

end i



3.c) Computation of row (and column) counts: illustration

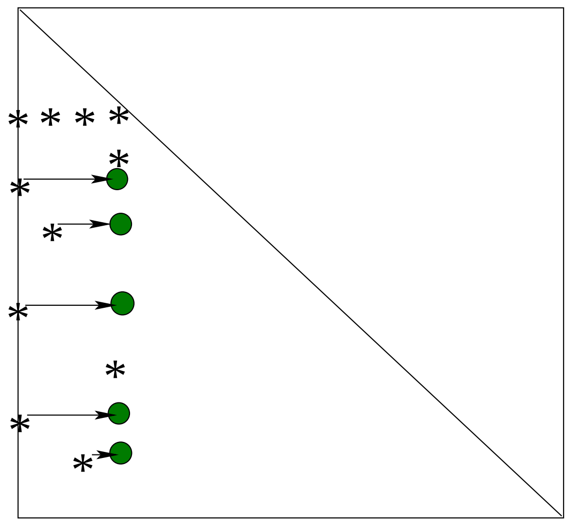
- computational complexity of evaluation row and column counts: $O(|L|)$
- there exist algorithms with the complexity $O(|A|, \alpha(|A|, n))$ based on decomposition of the row subtrees on independent subtrees



3.d) Column structure of L: introduction

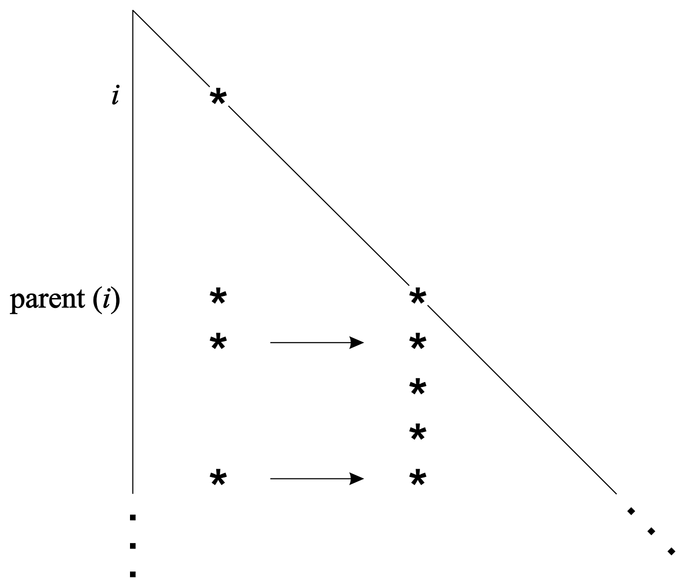
Lemma 5 *Column j is updated in the decomposition by columns i such that $l_{i,j} \neq 0$.*

Lemma 6
$Struct(L_{*j}) = Struct(A_{*j}) \;\cup\; \bigcup_{i,l_{ij}\neq 0} Struct(L_{*i}) \setminus \{1,\ldots,j-1\}$.



3.d) Column structure of L: an auxiliary result

Lemma 7 $Struct(L_{*j})\ \{j\} \subseteq Struct(L_{*parent(j)})$



3.d) Column structure of L: final formula

Consequently:

$$Struct(L_{*j}) = Struct(A_{*j}) \;\cup \bigcup_{i,j=parent(i)} Struct(L_{*i}) \setminus \{1,\dots,j-1\}.$$

This fact directly implies an algorithm to compute structures of columns

3.d) Column structure of L: algorithm

```
for j = 1 to n do
   list_{x_j} = ∅
end j
for j = 1 to n do
   col(j) = adj(x_j) \ {x_1, ..., x_{j-1}}
   for x_k ∈ list_{x_j} do
     col(j) = col(j) ∪ col(k) \ {x_j}
     end x_k
     if col(j) ≠ 0 then
       p = min{i | x_i ∈ col(j)}
       list_{x_p} = list_{x_p} ∪ {x_j}
     end if
   end j
end i
```

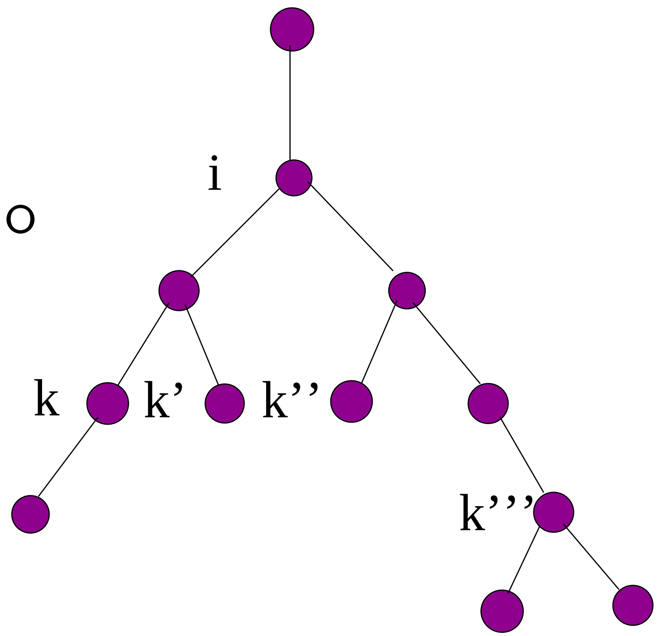
3.d) Column structure of L: symbolic factorization

- array $list$ stores children of a node
- the fact that **parent** of a node has a higher label than the node induce the correctness of the algorithm
- the algorithm for finding structures of columns also called **symbolic factorization**
- the derived descriptions used for:
  - to allocate space for L
  - to store and manage L in **static data structures**
- needed elimination tree

3.e) Elimination tree construction: algorithm

(complexity: $O(|A|, \alpha(|A|, n)))$)

for $i = 1$ to n do
  $parent(i) = 0$
  for k such that $x_k \in adj(x_i) \wedge k < i$ do
    $j = k$
    while $(parent(j) \neq 0 \wedge parent(j) \neq i)$ do
      $r = parent(j)$
    end while
    if $parent(j) = 0$ then $parent(j) = i$
  end k
end i



4. Preprocessing for SPD matrices

4.a) Preprocessing: the problem of reordering to minimize fill-in

- Arrow matrix (again)

$$\begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix} \qquad \begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix}$$

$$\Downarrow$$

Find efficient reorderings to minimize fill-in

4.b) Solving the reordered system: overview

Factorize

$$P^T A P = L L^T,$$

Compute y from

$$L y = P^T b,$$

Compute x from

$$L^T P^T x = y.$$

4.c) Static reorderings: local and global reorderings

Static reorderings

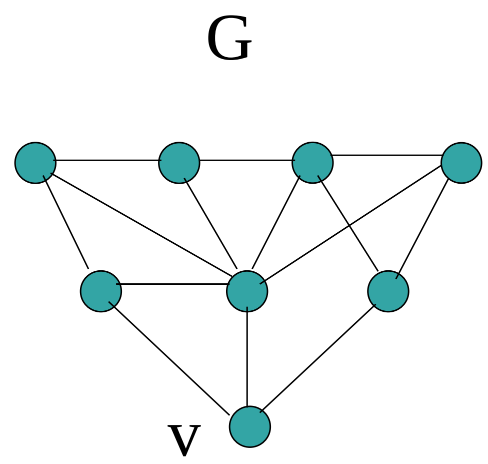
- **static** differs them from dynamic reordering strategies (pivoting)
- two basic types
  - local reorderings: based on local greedy criterion
  - global reorderings: taking into account the whole graph / matrix

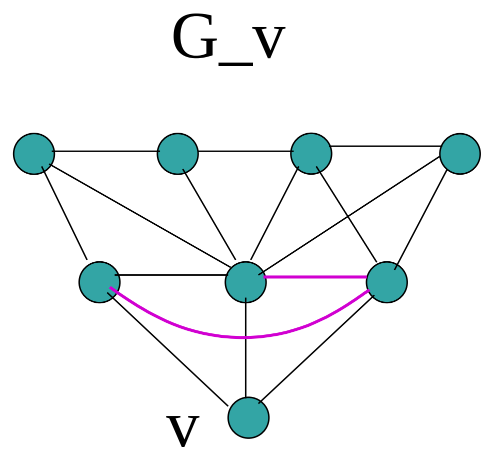
4.d) Local reorderings: minimum degree (MD): the basic algorithm

$G = G(A)$
for $i = 1$ to n do
  find v such that $deg_G(v) = min_{v \in V} deg_G(v)$
  $G = G_v$
end i
The order of found vertices induces their new renumbering

- $deg(v) = |Adj(v)|$; graph G as a superscript determines the current graph

4.d) Local reorderings: minimum degree (MD): an example





4.d) Local reorderings: minimum degree (MD): indistinguishability

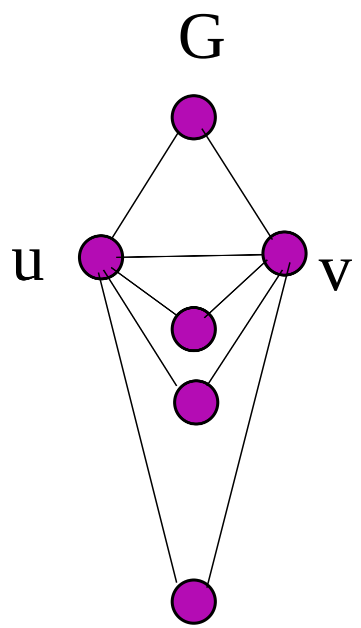
Definition 11 *u a v are called* **indistinguishable** *if*

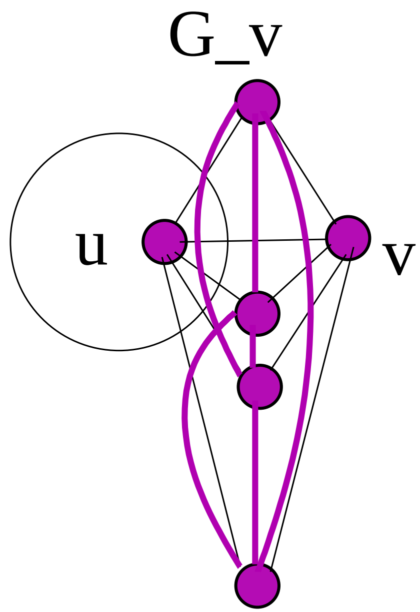
$$Adj_G(u) \cup \{u\} = Adj_G(v) \cup \{v\}. \tag{1}$$

Lemma 8 *If u and v are indistinguishable in G and $y \in V$, $y \neq u, v$. Then u and v are indistinguishable also in G_y.*

Corollary 1 *Let u and v be indistinguishable in G, $y \equiv u$ has minimum degree in G. Then v has minimum degree in G_y.*

4.d) Local reorderings: minimum degree (MD): indistinguishability (example)





4.d) Local reorderings: minimum degree (MD): dominance

Definition 12 *Vertex v is called* **dominated** *by u if*

$$Adj_G(u)\{u\} \subseteq Adj_G(v) \cup \{v\}. \tag{2}$$

Lemma 9 *If v is dominated by u in G, and $y \neq u, v$ has minimum degree in G. Then v is dominated by u also in G_y.*

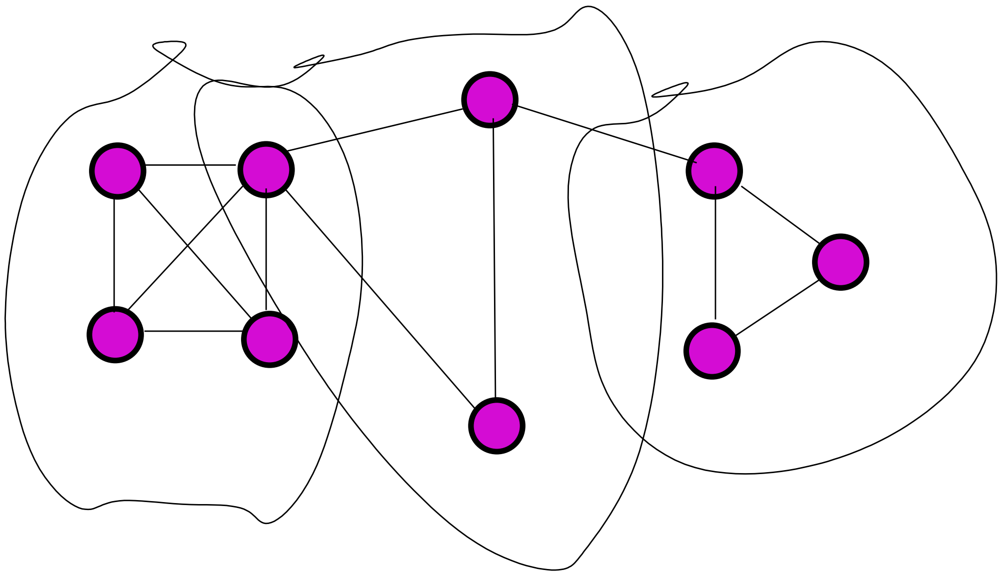
Corollary: Graph degrees do not need to be recomputed for dominated vertices (using following relations)

$$v \notin Adj_G(y) \Rightarrow Adj_{G_y}(v) = Adj_G(v) \tag{3}$$

$$v \in Adj_G(y) \Rightarrow Adj_{G_y}(v) = (Adj_G(y) \cup Adj_G(v)) - \{y\} \tag{4}$$

4.d) Local reorderings: minimum degree (MD): implementation

- graph is represented in a **clique representation** $\{K_1, \ldots, K_q\}$
- – **clique**: a complete subgraph



4.d) Local reorderings: minimum degree (MD): implementation (II.)

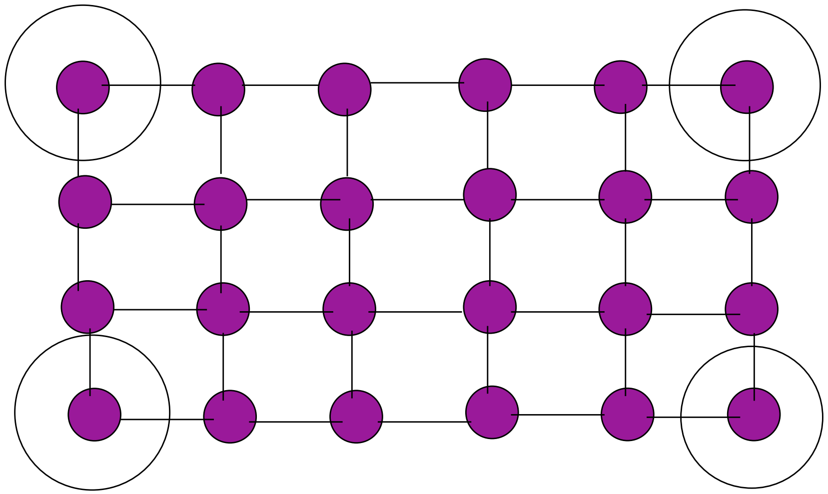
- cliques are **being created** during the factorization
- they answer the main related question: how to **store** elimination graphs with their new edges

Lemma 10

$$|K| < \sum_{i=1}^{t} |K_{s_i}|$$

for merging t cliques K_{s_i} into K.

4.d) Local reorderings: minimum degree (MD): multiple elimination



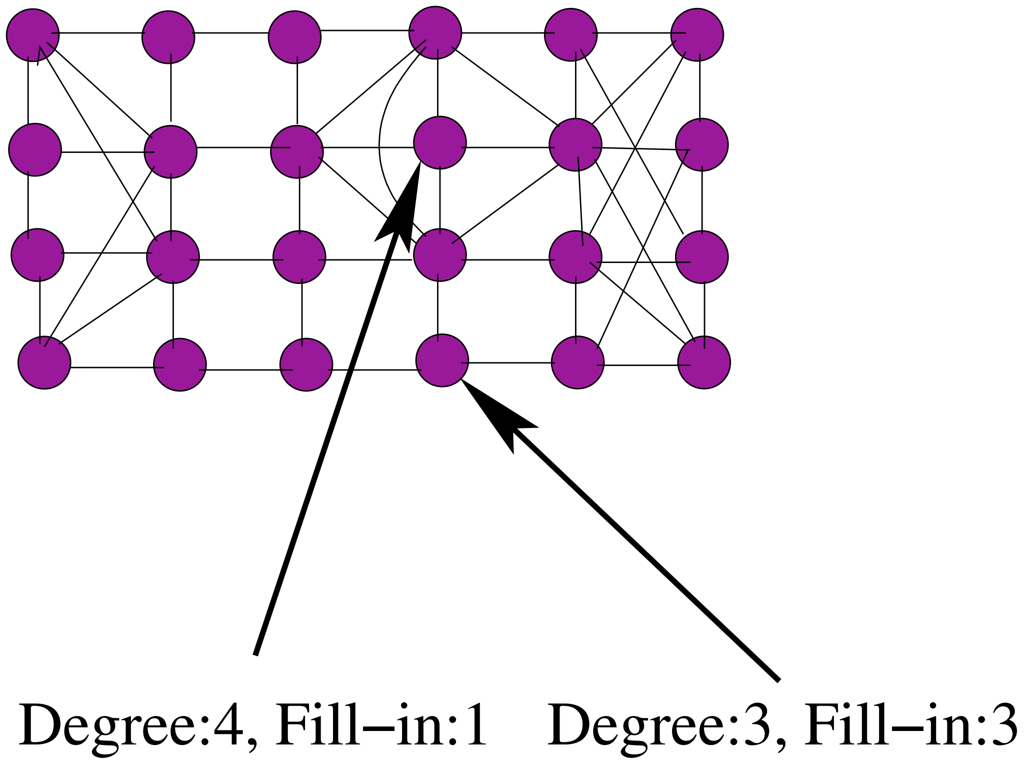
4.d) Local reorderings: minimum degree (MD): MMD algorithm

$G = G(A)$
while $G \neq \emptyset$
  find all v_j, $j = 1, \ldots, s$ such that
  $deg_G(v_j) = min_{v \in V(G)} deg_G(v)$ and $adj(v_j) \cap adj(v_k)$ pro $j \neq k$
  for $j = 1$ to s do
    $G = G_{v_j}$
  end for
end while
The order of found vertices induces their new renumbering

4.d) Local reorderings: other family members

- MD, MMD with the improvements (cliques, indistinguishability, dominance, improved clique arithmetic like clique absorbtions)
- more drastic changes: approximate minimum degree algorithm
- approximate minimum fill algorithms
- in general: local fill-in minimization procedures typically suffer from lack of **tie-breaking** strategies – multiple elimination can be considered as such strategy

4.d) Local reorderings: illustration of minimum fill-in reordering

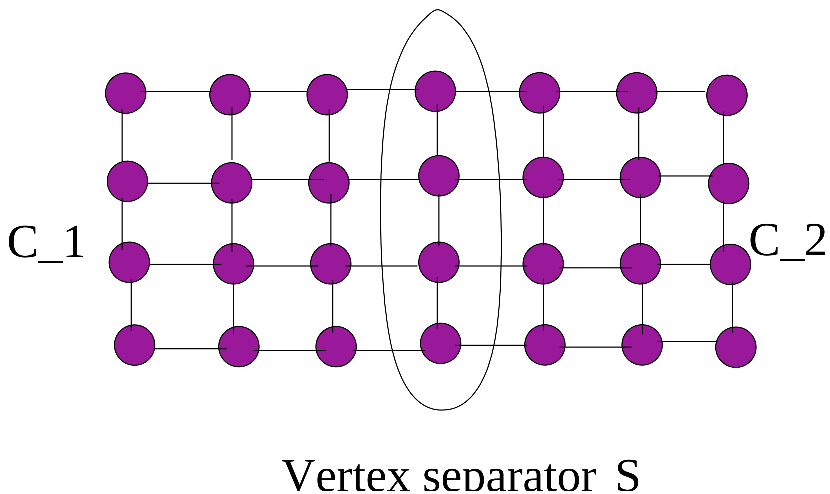


4.e) Global reorderings: nested dissection

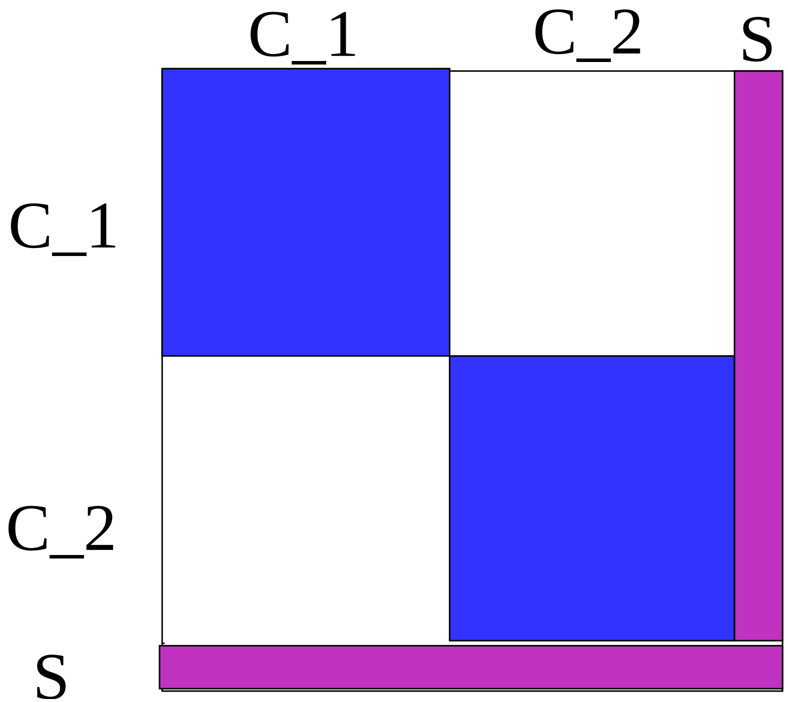
Find separator

Reorder the matrix numbering nodes in the separator last

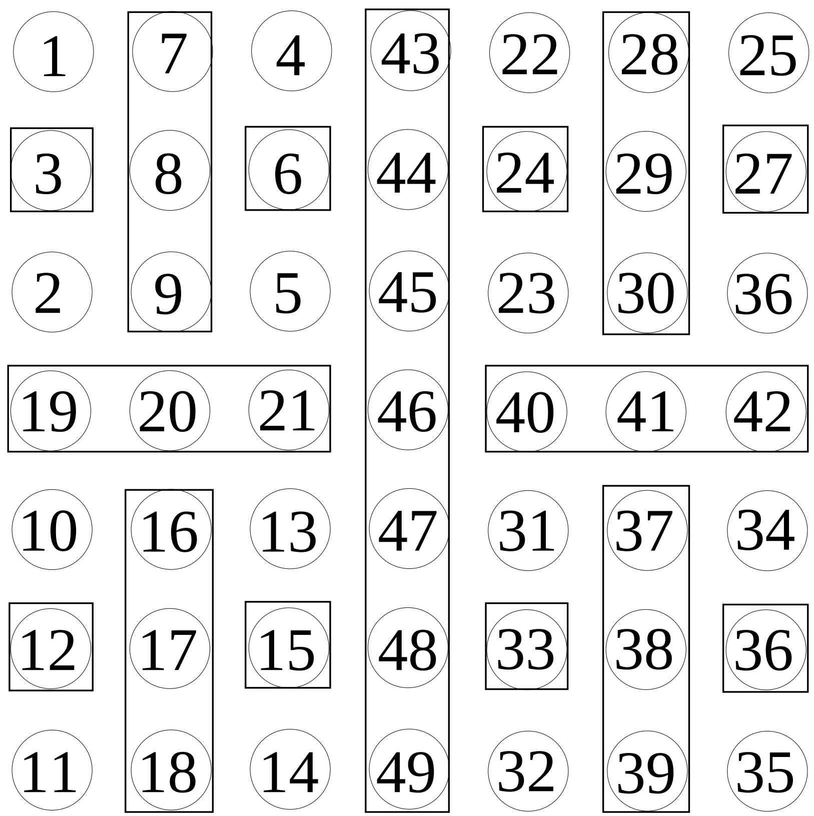
Do it recursively



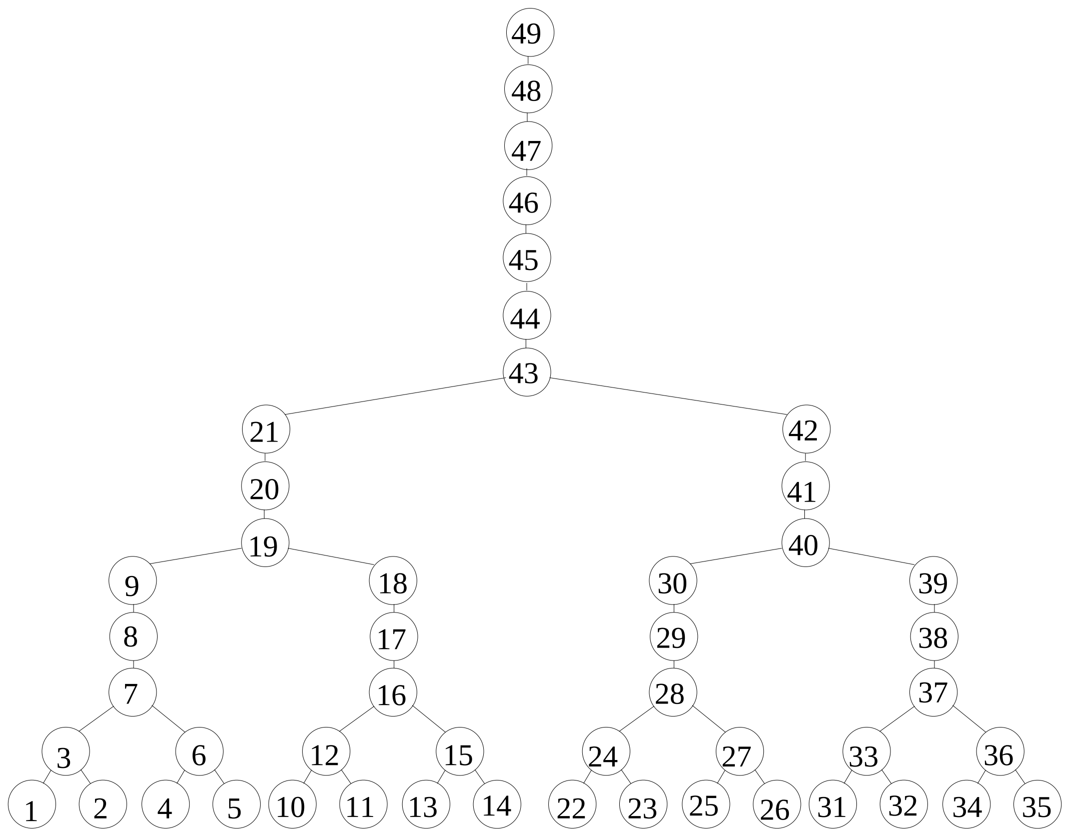
4.e) Global reorderings: nested dissection after one level of recursion



4.d) Global reorderings: nested dissection with more levels



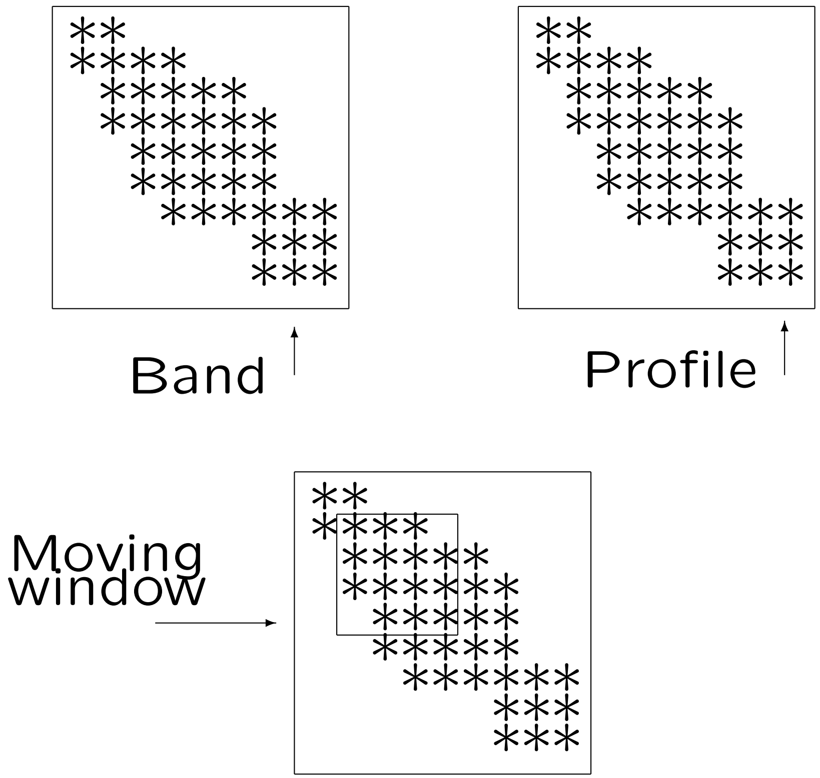
4.e) Global reorderings: nested dissection with more levels: elimination tree



4.f) Static reorderings: a preliminary summary

- the most useful strategy: **combining** local and global reorderings
- modern nested dissections are based on **graph partitioners**: partition a graph such that
  - components have very similar sizes
  - separator is small
  - can be correctly formulated and solved for a general graph
  - theoretical estimates for fill-in and number of operations
- modern local reorderings: used after a few steps of an incomplete nested dissection

4.f) Static reorderings: classical schemes based on pushing nonzeros towards the diagonal



Frontal method - dynamic band

4.f) Static reorderings: classical schemes based on pushing nonzeros towards the diagonal: an important reason for their use

$$Band(L + L^T) = Band(A)$$

$$Profile(L + L^T) = Profile(A)$$

4.f) Static reorderings: classical schemes based on pushing nonzeros towards the diagonal: pros and cons

- +: simple data structure – locality, regularity
- –: structural zeros inside
- +: easy to vectorize
- –: short vectors
- +: easy to use out-of-core
- –: the other schemes are typically more efficient and this is more important

Evaluation: for general case – more or less historical value only; can be important for special matrices, reorderings in iterative methods

4.f) Static reorderings: an example of comparison

Example (Liu): 3D finite element discretization of the part of the automobile chassis $\Longrightarrow$ linear system with a matrix of dimension 44609. Memory size for the frontal (= dynamic band) solver: 52.2 MB; memory size for the general sparse solver: 5.2MB!

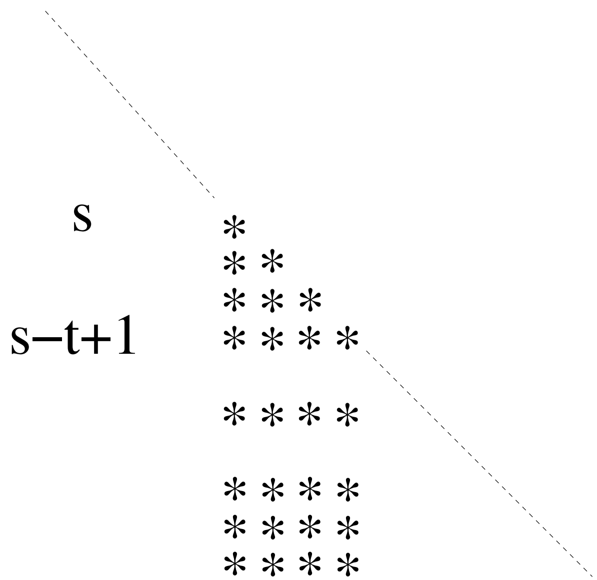
5. Algorithmic improvements for SPD decompositions

5.a) Algorithmic improvements: introduction

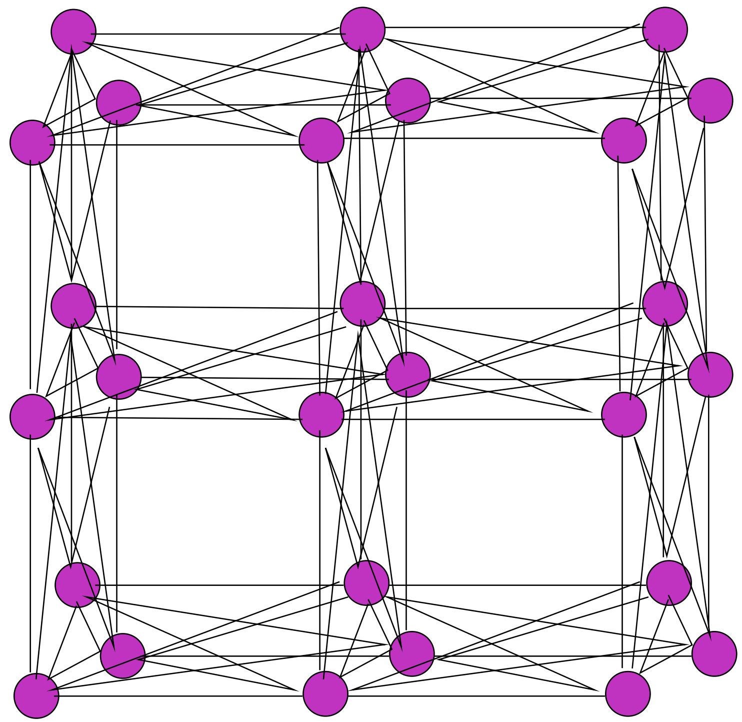
- **blocks and supernodes**: less tolerance to memory latencies and increase of efective memory bandwidth
- **Reorderings based on the elimination tree**

5.b) Supernodes and blocks: supernodes

Definition 13 *Let* $s, t \in M_n$ *such that* $s + t - 1 \leq n$. *Then the columns with indices* $\{s, s+1, \ldots, s+t-1\}$ *form a* supernode *if this the columns satisfy* $Struct(L_{*s}) = Struct(L_{*s+t-1}) \cup \{s, \ldots, s+t-2\}$, *and the sequence is maximal.*



5.b) Supernodes and blocks: blocks

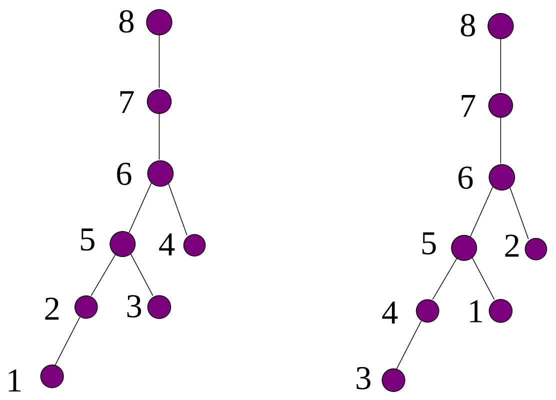


5.b) Supernodes and blocks: some notes

- **enormous influence** on the efficiency
- different definitions of supernodes and blocks
- blocks found in $G(A)$, supernodes are found in $G(L)$
- blocks are induced by the application (degrees of freedom in grid nodes) or efficient algorithms for finding blocks
- efficient algorithms to find supernodes
- complexity: $O(|A|)$

5.c) Reorderings based on the elimination tree: topological reorderings

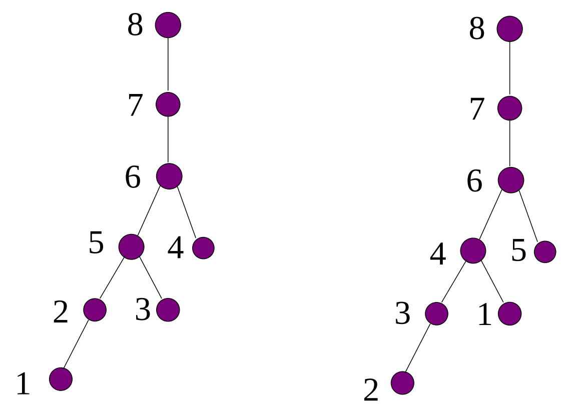
Definition 14 **Topological reorderings** *of the elimination tree are such that each node has smaller index than its parent.*



Tree with two different topological reorderings

5.c) Reorderings based on the elimination tree: postorderings

Definition 15 Postorderings *are topological reorderings where labels in each rooted subtree form an interval.*

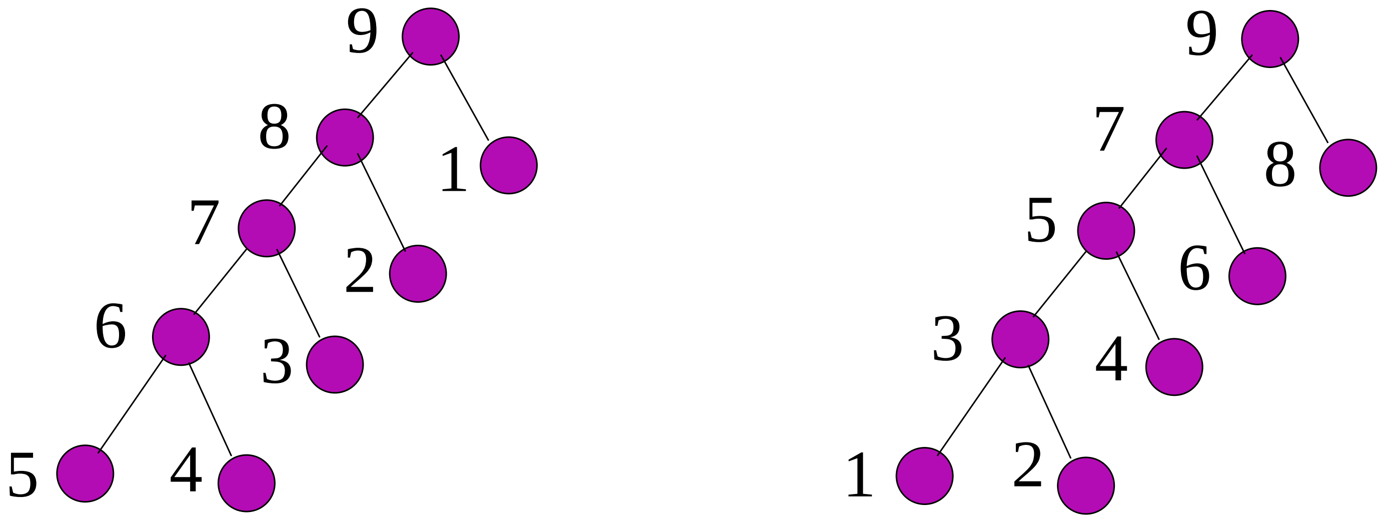


Postordered tree

5.c) Reorderings based on the elimination tree: postorderings

- Postorderings efficiently use **memory hierarchies**
- Postorderings are very useful in **paging environments**
- They are crucial for **multifrontal methods**
- Even some postorderings are better than the other: transparent description for multifrontal method, see the example

5.c) Reorderings based on the elimination tree: postorderings: example



6. Sparse direct methods for SPD matrices

6.a) Algorithmic strategies: introduction

- Some algorithms **strongly modified** with respect to their simple dense counterparts because of special data structures
- different symbolic steps for different algorithms
- different amount of overhead
- of course, algorithms provide the same results in exact arithmetic

6.b) Work and memory

- the same number of operations if no additional operations performed

- $\mu(L)$: number of the arithmetic operations

- $\eta(L) \equiv |L|$, $\eta(L_{*i})$: size of the i-th column of L, etc.

$$
\begin{aligned}
|L| &= \sum_{i=1}^{n} \eta(L_{*i}) = \sum_{i=1}^{n} \eta(L_{i*}) \\
\mu(L) &= \sum_{j=1}^{n} [\eta(L_{*j}) - 1][\eta(L_{*j}) + 2]/2
\end{aligned}
$$

6.c) Methods: rough classification

- **Columnwise** (left-looking) algorithm
- – columns are updated by a linear combination of previous columns
- Standard **submatrix** (right-looking) algorithm
- – non-delayed outer-product updates of the remaining submatrix
- **Multifrontal** algorithm
- – specific efficient delayed outer-product updates
- Each of the algorithms represent a **whole class** of approaches
- Supernodes, blocks and other enhancements extremely important

6.d) Methods: sparse SPD columnwise decomposition

Preprocessing

– prepares the matrix so that the fill-in would be as small as possible

Symbolic factorization

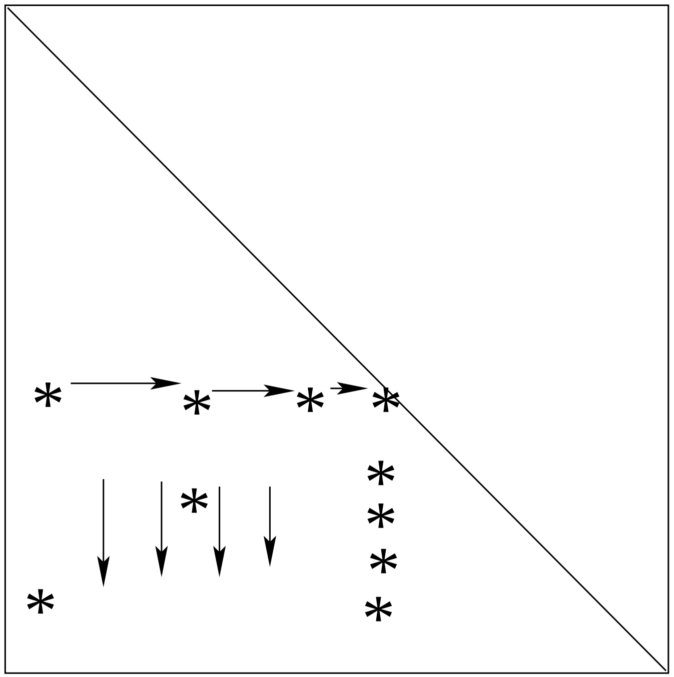
– determines structures of columns of L. Consequently, L can be allocated and used for the actual decomposition

– due to a lot of enhancements the boundary between the first two steps is somewhat blurred

Numeric factorization

– the actual decomposition to obtain numerical values of the factor L

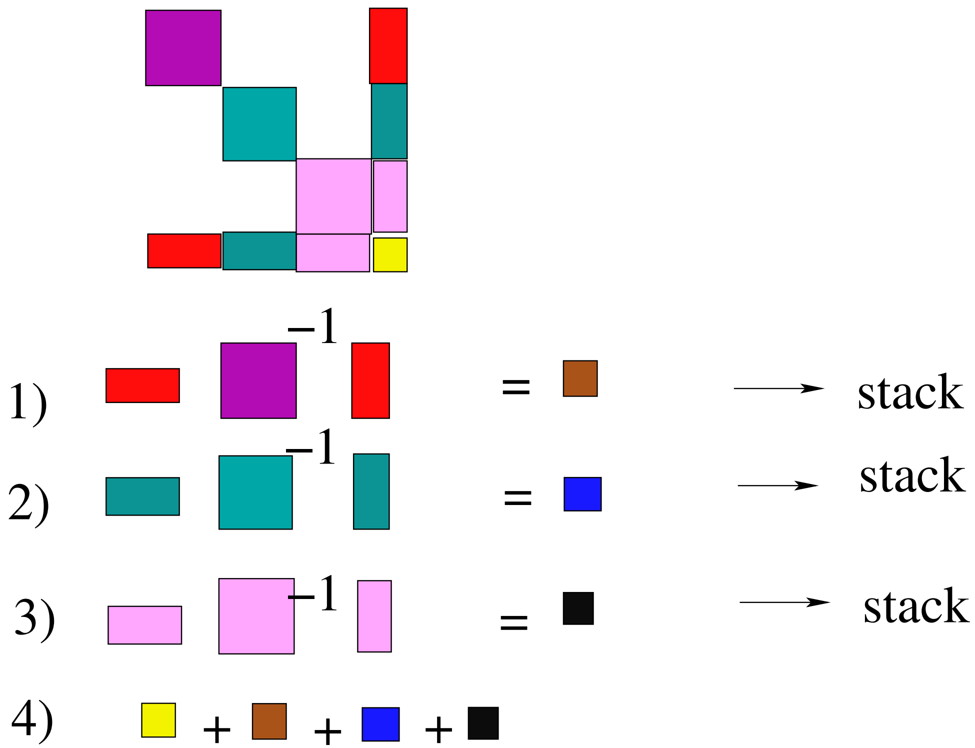
6.d) Methods: sparse SPD columnwise decomposition: numeric decomposition



6.e) Methods: sparse SPD multifrontal decomposition

- Right-looking method with delayed updates
- The updates are pushed into a stack and popped up only when needed
- Postorder guarantees having needed updates on the top of the stack
- some steps of the left-looking SPD modified

6.e) Methods: sparse SPD multifrontal decomposition: illustration



7. Sparse direct methods for nonsymmetric systems

7.a) SPD decompositions versus nonsymmetric decompositions

- LU factorization instead of Cholesky
- **nonsymmetric (row/column)** permutations needed for both **sparsity preservation** and maintaining **numerical stability**
- consequently: **dynamic reorderings (pivoting)** needed
- a different model for fill-in analysis: generalizing the elimination tree