Sparse Matrices in Numerical Mathematics

Miroslav Tůma

Faculty of Mathematics and Physics

Charles University

mirektuma@karlin.mff.cuni.cz

Praha, November 1, 2022

Outline

Introductory notes

- Created as a material supporting online lectures of NMNV533.
- Assuming basic understanding of algebraic iterative (Krylov space) and direct (dense) solvers (elimination/factorization/solve) (A lot of these is repeated)
- The text deal prevailably with purely algebraic techniques. Such techniques often serve as building blocks for more complex approaches. In particular, some important techniques are mentioned at most. Like:
  - Multigrid/multilevel preconditioners,
  - Domain decomposition,
  - Row projection techniques.
- Only preconditioning of real systems considered here, although extension to complex field is typically straightforward.
- Orientation in variants of Cholesky and LU decompositions is assumed.

Introductory notes: resources / history

- The main resource is:
  Jennifer Scott and Miroslav Tůma: Algorithms for sparse linear systems, Birkhäuser- Springer, 2022, to appear.
- Printed parts of the resource will be provided to students until it will appear (expected open access then).
- Traditional material also the course text in Czech (nowadays outdated, not supported); see the web page of the course.

Introductory notes: resources and history of the course

- A few other resources:
  Davis, T. A. (2006). Direct Methods for Sparse Linear Systems. Fundamentals of Algorithms. SIAM, Philadelphia, PA.
  Davis, T. A., Rajamanickam, S., & Sid-Lakhdar, W.M. (2016). A survey of direct methods for sparse linear systems. Acta Numer., 25, 383-566.
  Duff, I. S., Erisman, A.M., & Reid, J. K. (2017). Direct Methods for Sparse Matrices (Second ed.). Oxford University Press, Oxford.
  George, A. & Liu, J. W. H. (1981). Computer Solution of Large Sparse Positive Definite Systems. Prentice Hall, Englewood Cliffs, NJ.
  Saad, Y. (2003b). Iterative Methods for Sparse Linear Systems (Second ed.). SIAM, Philadelphia, PA.

Motivation

- Most of our activities around solving

$$\mathbf{Ax} = \mathbf{b}$$

- Direct methods
- Iterative methods
- Practical boundaries between them more and more fuzzy.
- Principially different.

Motivation

Direct methods

- Direct methods: Transform A using a finite sequence of elementary transformations: An approach based on factorization (decomposition) and subsequent substitutions.
- The most simple case: $A \to LL^T$ or LDL^T or LU

  In principal = Gaussian elimination. Modern (decompositional) form based a lot on the work of Householder (end of 1950's)
  - Solving systems with triangular matrices like L, U is generally much cheaper and more straightforward that using A.
  - Factorizations are backbone of direct methods.
  - Occasionally other factorizations than LU or LL^T or LDL^T
  - Most of the work is in the (Cholesky, indefinite, LU) decomposition.
  - But: also the computer model (sequential, concurrent processors, multicore, GPU) decides about relative complexity of the two steps.
- The algorithms can be made more efficient/stable using additional techniques before, after or during factorization.
- For example, the solution can be made more accurate by an auxiliary iterative method.

Motivation

Iterative methods

- Compute a sequence of approximations

$$x^{(0)}, x^{(1)}, x^{(2)}, \ldots$$

  that (hopefully) converge to the solution x of the linear system.
- Iterative method are usually accompanied by a problem transformation based on a direct method called preconditioner.
- Usually have to be accompanied by a problem transformation based on a direct method called preconditioner.

Motivation

Iterative methods

- Algebraic preconditioners are tools to convert the problem $Ax = b$ into the one which is easier to solve. They are typically expressed in matrix form as a transformation like:

$$MAx = Mb$$

- M can be then used to apply approximation to A^{-1} to vectors used in the iterative method.
- In practice, it can store approximation to A or A^{-1} (approximate inverse).

Motivation

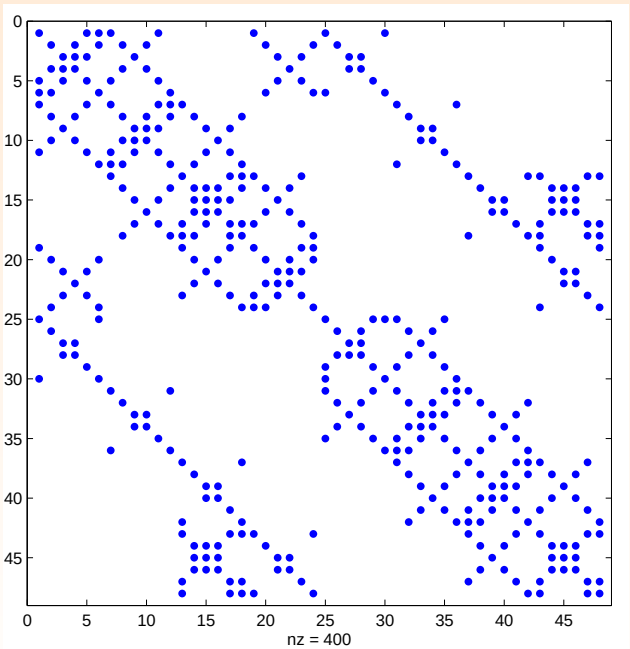
Contrast: direct versus iterative methods

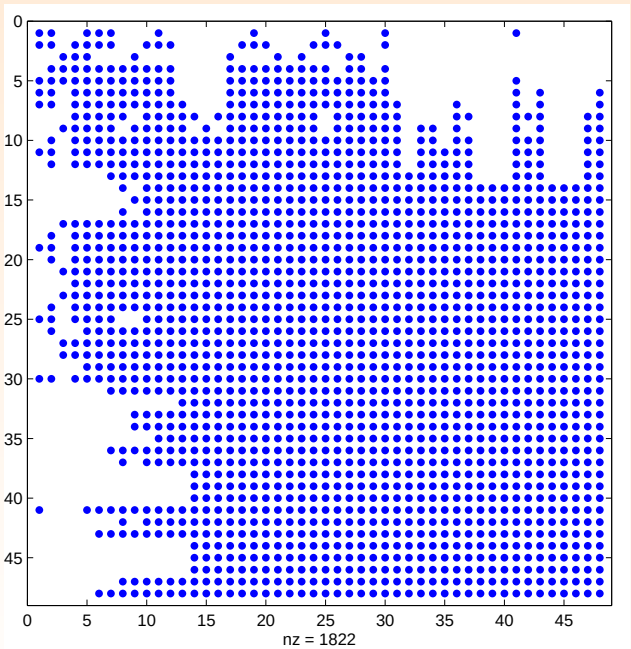
- Direct methods: designed to be robust, designed to solve
  - Properly implemented, they can be used as block-box solvers for computing solutions with predictable accuracy.
  - As we have seen, they can be expensive, requiring large amounts of memory, which increases with the size of A.
- Iterative methods: designed to approximate
  - The number of iterations depends on the initial guess $x^{(0)}$, A and b
  - Use the matrix A only indirectly, through matrix-vector products $\rightarrow$ memory requirements are limited to a (small) number of vectors of length the size of A
  - A does not need to be available explicitly.
  - They can be terminated as soon as the required accuracy in the computed solution is achieved.
  - Typically must be preconditioned. Preconditioner computation is sometimes based on a relaxation of a direct method.

Motivation

Where is the problem with direct methods?

- For example: sparse matrices and resulting factorizations may look like as follows:





Motivation

Where is the problem with direct methods?

- For example: and they can look like as:

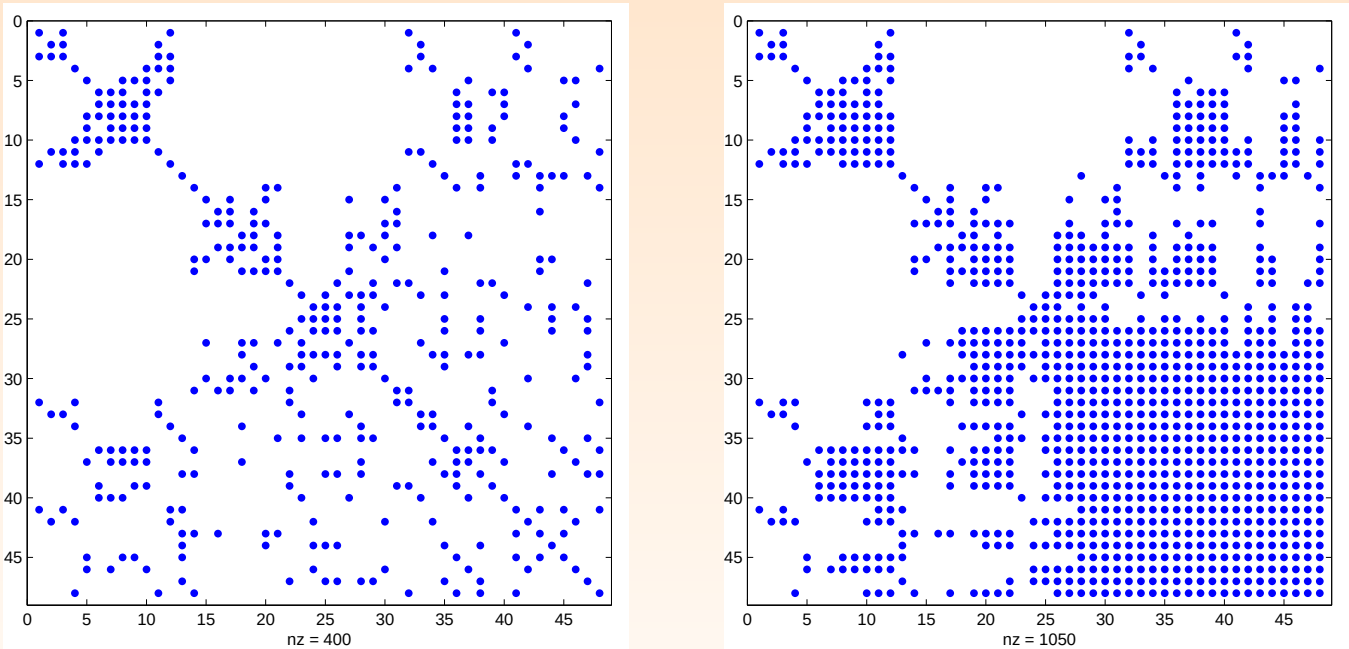


Figure: *The locations of the nonzero entries in a symmetric permutation of the matrix from Figure* **??** *(left) and in* $\bar{L} + \bar{L}^T$ *(right), where* $\bar{L}$ *is the Cholesky factor of the permuted matrix.*

Motivation

Where is the problem with direct methods?

- For example: and they can look like as:

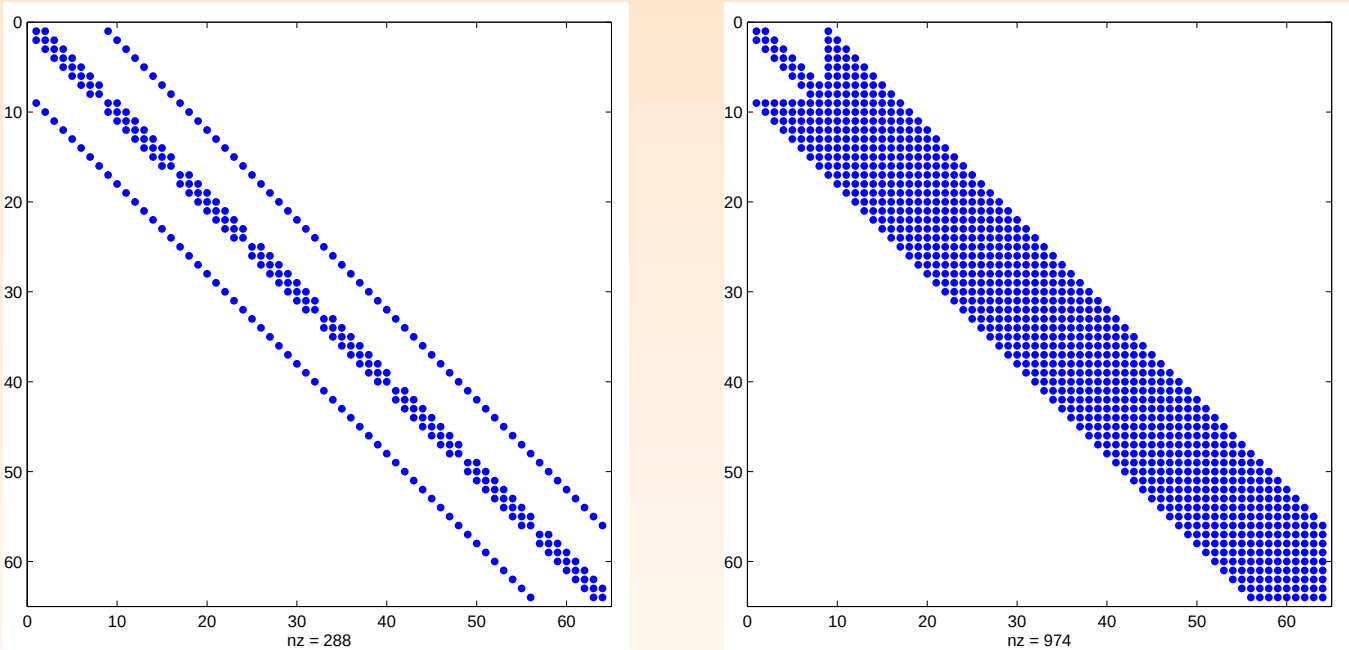


Figure: *The locations of the nonzero entries in a symmetric permutation of the matrix from Figure* **??** *(left) and in* $\bar{L} + \bar{L}^T$ *(right), where* $\bar{L}$ *is the Cholesky factor of the permuted matrix.*

Motivation

Where is the problem with direct methods?

- We need exploit sparsity (mentioned later)
- See the figures above
- We need sparse (complete) factorizations $A = LL^T$, LU (up to the floating-point model)

Where is the problem with iterative methods?

- We must transform (precondition)
- We need sparse (incomplete) factorizations $A = LL^T$, LU (up to the floating-point model) like
  - incomplete decompositions ($A \approx LL^T$, LU etc.)
  - incomplete inverse decompositions ($A^{-1} \approx ZZ^T$, WZ^T etc.)
- Or specific (PDE-based, model-based) approaches.

Outline

Basic Terminology

- Interest in solving linear systems of equations

$$Ax = b, \tag{1}$$

- $A \in \mathbb{R}^{n \times n}, 1 \leq i \leq n$, is nonsingular
- A is sparse
- $b \in \mathbb{R}^n$ (sparse or dense), $x \in \mathbb{R}^n$ is the solutions
- Used throughout:

$$A = (a_{ij}), \quad 1 \leq i, j \leq n.$$

- Matlab-like notation: nonzero (set a priori), $A_{:,j}$, $A_{i,:}$, $A_{i:j,k:l}$, A_{*j}, A_{i*}.

Basic Terminology

- $$\binom{X}{2} = \{Y \subseteq X \mid |Y| = 2\}$$
- Vectors denoted by small letters as v, u, x, matrices by capital letters as $A, B, \ldots$
- $A \in \mathbb{R}^{n \times n}, 1 \leq i \leq n$, is nonsingular, the right-hand side vector $b \in \mathbb{R}^n$ is given and $x \in \mathbb{R}^n$ is the required solution vector. n is the order (or dimension) of A.

Basic Terminology

- A is diagonal if for all $i \neq j$, $a_{ij} = 0$;
- A is lower triangular if for all $i < j$, $a_{ij} = 0$
- A is upper triangular if for all $i > j$, $a_{ij} = 0$.
- A is unit triangular if it is triangular and all the entries on the diagonal are equal to one.
- A is structurally symmetric if for all i and j for which a_{ij} is nonzero the entry a_{ji} is also nonzero.
- A is symmetric if

$$a_{ij} = a_{ji}, \quad \text{for all} \quad i, j.$$

  Otherwise, A is nonsymmetric.
- The symmetry index $s(A)$ of A: the number of nonzeros a_{ij}, $i \neq j$, for which a_{ji} is also nonzero divided by the total number of off-diagonal nonzeros. Small values of $s(A)$: A is far from symmetric.

Basic Terminology: special matrix classes

- A is symmetric positive definite (SPD) if it is symmetric and satisfies
$$v^T A v > 0 \text{ for all nonzero } v \in \mathbb{R}^n.$$
- Otherwise, A is symmetric indefinite.
- Symmetric and (typically) indefinite saddle point matrices have the form
$$A = \begin{pmatrix} G & R^T \\ R & B \end{pmatrix},$$
where $G \in \mathbb{R}^{n_1 \times n_1}$, $B \in \mathbb{R}^{n_2 \times n_2}$, $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 + n_2 = n$, G is a SPD matrix and B is a symmetric positive semidefinite matrix (that is $v^T B v \geq 0$ for all nonzero $v \in \mathbb{R}^{n_2}$). In some applications, $B = 0$.

Basic Terminology: blocks

- Symmetric block structure of A:
- $$A = (A_{ib,\,jb}), \quad A_{ib,\,jb} \in \mathbb{R}^{n_i \times n_j}, \quad 1 \le ib, jb \le nb, \tag{2}$$

  that is,

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ A_{nb,1} & A_{nb,2} & \cdots & A_{nb,nb} \end{pmatrix}.$$

- Assuming the square blocks $A_{jb,\,jb}$ on the diagonal are nonsingular.
- Special cases: A is block diagonal if $A_{ib,\,jb} = 0$ for all $ib \neq jb$, A is block lower triangular if $A_{1:jb-1,\,jb} = 0$, $2 \le jb \le nb$, block upper triangular if $A_{jb+1:nb,\,jb} = 0$, $1 \le jb \le nb - 1$.

Basic Terminology: blocks and reducibility

Definition

Matrix $A \in \mathbf{R}^{n \times n}$ is **reducible**, if there is a permutation matrix P such that

$$P^T A P = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix}, \tag{3}$$

where A_{11} and A_{22} are square nontrivial matrices (of dimension at least 1). If A is not reducible, it is called **irreducible**. Matrices of dimension 1 are always considered to be irreducible.

Remark

Symmetric reducible matrix is block diagonal.

Basic Terminology: sparsity

- A is a sparse matrix if many of its entries are zero.
- Attempts to formalize matrix sparsity more precisely: matrix of order n may be said to be sparse if it has $O(n)$ nonzeros.
- Our choice: A is sparse if it is advantageous to exploit its zero entries. Otherwise, A is dense.
- The sparsity pattern $\mathcal{S}\{A\}$ of A is the set of nonzeros, that is,

$$\mathcal{S}\{A\} = \{(i,j) \mid a_{ij} \neq 0,\ 1 \leq i,j \leq n\}.$$

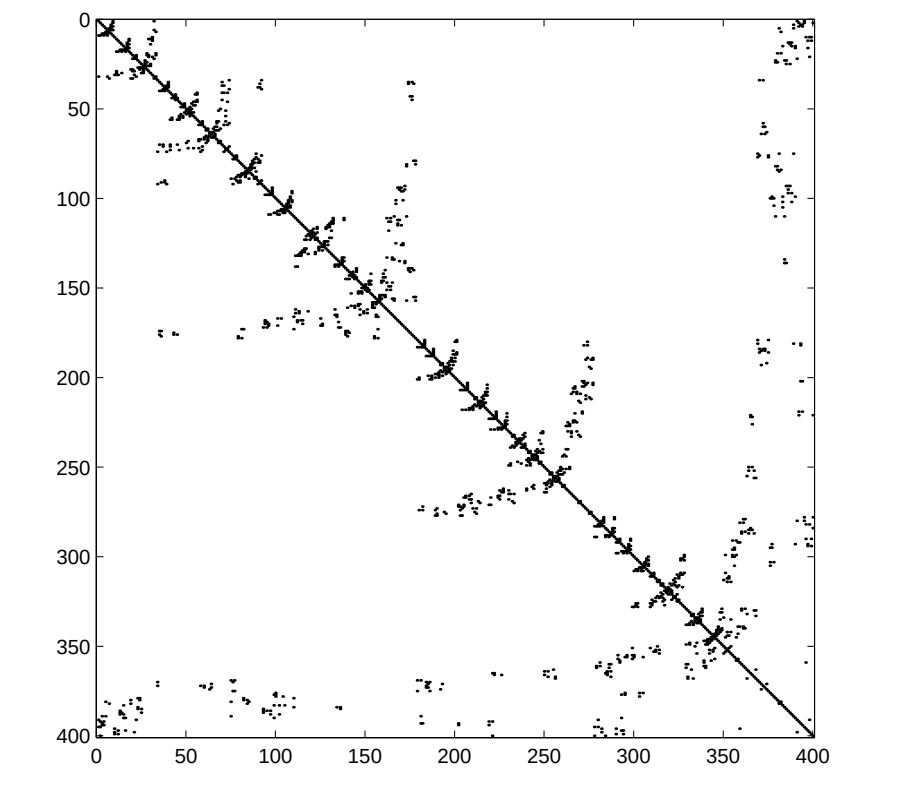
- $\mathcal{S}\{A\}$ is symmetric if for all i and j, $a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$ (the values of the two entries need not be the same). If $\mathcal{S}\{A\}$ is symmetric then A is structurally symmetric.
- The number of nonzeros in A: denoted by $nz(A)$ (or $|\mathcal{S}\{A\}|$). A is structurally (or symbolically) singular if there are no values of the $nz(A)$ entries of A whose row and column indices belong to $\mathcal{S}\{A\}$ for which A is nonsingular.

Basic Terminology: sparsity

Sparsity: taking into account the structure of matrix nonzeros

Definition

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



Basic Terminology: sparsity

Definition

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 $c_{max} << m$.

Definition

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

(pragmatic, application-based 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.

Basic Terminology: sparsity

An example showing importance of 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

Basic Terminology: sparsity

Rough comparison of dense and sparse (dimension, storage, time for decomposition)

Dense matrix

dim	space	dec time (s)
3000	4.5M	5.72
4000	8M	14.1
5000	12.5M	27.5
6000	18M	47.8

Sparse matrix

dim	space	dec time (s)
10000	40k	0.02
90000	0.36M	0.5
1M	4M	16.6
2M	8M	49.8

Basic Terminology: sparsity

- Sparse vectors The sparsity pattern of $v \in \mathbb{R}^n$ is given by

$$\mathcal{S}\{v\} = \{i \,|\, v_i \neq 0\},$$

  and $|\mathcal{S}\{v\}|$ is the length of v.
- Matrix A is factorizable (or strongly regular) if its principal leading minors (the determinants of its principal leading submatrices) are nonzero, that is, if its LU factorization without row/column interchanges does not break down.
- SPD matrices are factorizable.
- For more general A, in exact arithmetic the following standard result holds.

Theorem

If A is nonsingular then the rows of A can be permuted so that the permuted matrix is factorizable. The row permutations do not need to be known in advance. They can be constructed on-the-fly as the factorization proceeds.

Basic Terminology: factorizations

- For symmetric positive definite A, the Cholesky factorization $A = LL^T$, where L is a lower triangular matrix with positive diagonal entries.
  - Rewritten as $A = \widehat{L}D\widehat{L}^T$, where $\widehat{L}$ is a unit lower triangular matrix and D is a diagonal matrix with positive diagonal entries: square root-free Cholesky factorization.
- For nonsymmetric A, the LU factorization $A = LU$, where L is a unit lower triangular matrix and U is an upper triangular matrix. Gaussian elimination is one process to put a matrix into LU form.
  - Rewritten as $A = LD\widehat{U}$, where $\widehat{U}$ is a unit upper triangular matrix and D is a diagonal matrix. This is called the LDU factorization.

Basic Terminology: direct solver phases

- First look: The matrix A is factorized and then, given the right-hand side b, the factors used to compute the solution x.
- Second look:
- Most approaches further split the factorization into a symbolic phase (also called the analyse phase) and a numerical factorization phase that computes the factors.
- The symbolic phase: typically uses only $\mathcal{S}\{A\}$ to compute the nonzero structure of the factors of A without computing the numerical values of the nonzeros.
- The solve phase uses the factors to solve for a single b or for a block of multiple right-hand sides or for a sequence of right-hand sides one-by-one.
- Historically, the symbolic phase was much faster than the factorization phase. But parallelising the factorization $\rightarrow$ timings are much more closer.
- Series of problems in which the numerical values of the entries of A change but $\mathcal{S}\{A\}$ does not: symbolic phase just once.

Basic Terminology: computational environment

- Basic sequential model: the von Neumann architecture:union of a central processing unit (CPU) and the memory, interconnected via input/output (I/O) mechanisms.

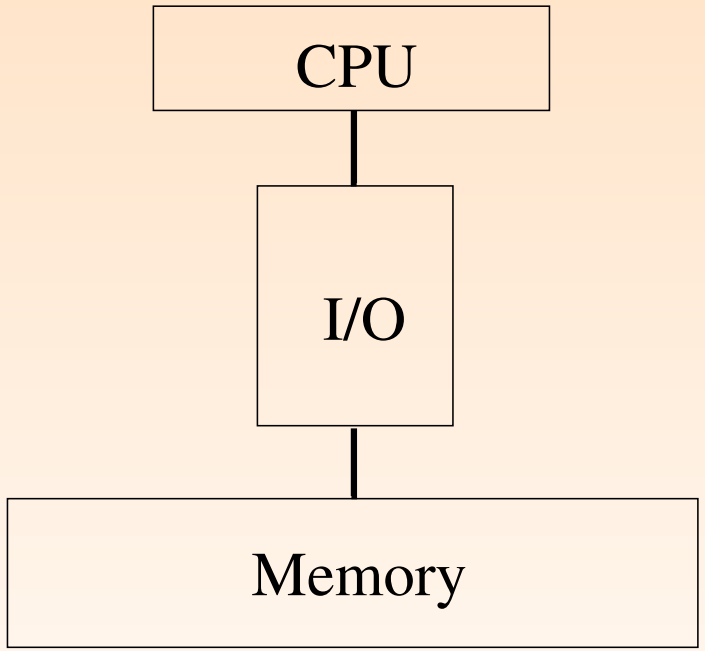


Figure: *A simple uniprocessor von Neumann computer model.*

Basic Terminology: computational environment

- Nowadays: CPU $\rightarrow$ a mixture of powerful processors, co processors, cores, GPUs, and so on.
- Furthermore, performing arithmetic operations on the processing units is much faster than communication-based operations.
- Moreover, improvements in the speed of the processing units outpaces those in the memory-based hardware. Moore's Law is an example of an experimentally derived observation of this kind.

Basic Terminology: computational environment

- Important milestones in processor development have been multiple functional units that compute identical numerical operations in parallel and data pipelining (also called vectorization) that enables the efficient processing of vectors and matrices.
- Vectorization often supported by additional tools like instruction pipelining, registers and by memory architectures with multiple layers, including small but fast memories called caches.
- Superscalar processors that enable the overlapping of identical (or different) arithmetic operations during run-time have been a standard component of computers since the 1990s.
- The ever-increasing heterogeneity of processing units and their hardware environment inside computers: expressing the code via units of scheduling and execution called threads.

Basic Terminology: computational environment

- Computer-based limitations:
  - Compute throughput, that is, the number of arithmetic operations that can be performed per cycle.
  - Memory throughput, that is, the number of operands than can be fetched from memory/cache and/or registers each cycle
  - Latency, which is the time from initiating a compute instruction or memory request before it is completed and the result available for use in the next computation.
- Distinguishing: algorithms compute-bound, memory-bound or latency-bound.
- More ways to hide latency (blocks, prefetch, threads)
- Measuring computational intensity: the ratio of the number of operations to the number of operands read from memory.
- Most chips designed such that dense matrix-matrix multiply, which typically performs n^3 operations on n^2 data (with ratio k for a blocked algorithm with block size k), can run at full compute throughput, whilst matrix-vector multiply performs n^2 operations on n^2 data (ratio 1) and is limited by the memory throughput.

Basic Terminology: computational environment

- The development of basic linear algebra subroutines (BLAS) for performing common linear algebra operations on dense matrices partially motivated by obtaining a high ratio. efficiently.
- Other important motivations behind using the BLAS (standardization, portability).
- Machine-specific optimized BLAS libraries available for a wide variety of computer architectures.

procedure	*comm*	*ops*	ratio
BLAS 1: AXPY: $y = y + \alpha x$	$3n + 1$	$2n$	$2/3$
BLAS 2: GEMV: $y = Ax$	$n^2 + 2n$	$n(2n - 1)$	2
BLAS 3: GEMM: $C = AB$	$3n^2$	$n^2(2n - 1)$	$n/2$

- Consequently, exploiting Level 3 BLAS when designing and implementing matrix algorithms (for both sparse and dense matrices) can improve performance compared to using Level 1 and Level 2 BLAS.

Basic Terminology: finite precision arithmetic

- The IEEE standard (1985) expresses real numbers as $a = \pm d_1.d_2 \ldots d_t \times 2^k$, where k is an integer and $d_i \in \{0, 1\}, 1 \leq i \leq t$, with $d_1 = 1$ unless $d_2 = d_3 = \ldots = d_t = 0$.
- $t = 24$ (single precision), $t = 53$ (double precision), exponent k satisfies $-126 \leq k \leq 127$ (single precision) and $-1022 \leq k \leq 1023$ (double precision).
- Floating-point (FP) operations:

$$fl(a\ op\ b) = (a\ op\ b)(1 + \delta), \qquad |\delta| \leq \epsilon,$$

  (op is a mathematical operation (such as $=, +, -, \times, /, \sqrt{}$) and $(a\ op\ b)$ is the exact result), ϵ is the machine epsilon.
- $2 \times \epsilon$ is the smallest FP number which when added to the FP number 1.0 gives a result different from 1.0.
- ϵ is $2^{-24} \approx 10^{-7}$ (single precision), $\epsilon = 2^{-53} \approx 10^{-16}$ (double precision).
- rounding errors, truncation errors.
- catastrophic errors $\rightarrow$ numerical instability

Basic Terminology: bit compatibility

- Bit compatibility is essential for some users because of regulatory requirements (such as within the nuclear or financial industries) or to build trust in their software from non technical users.
- The critical issue is the way in which N numbers (or, more generally, matrices) are assembled:

$$sum = \sum_{j=1}^{N} S_j,$$

  where the S_j are computed using one or more processors. The assembly is commutative but, because of the potential rounding of the intermediate results, is not associative so that the result sum depends on the order in which the S_j are assembled.
- A straightforward approach to achieving bit compatibility is to enforce a defined order in such operations.
- This may adversely limit the scope for parallelism.

Basic Terminology: complexity

- The computational complexity of a numerical algorithm is typically based on estimating asymptotically the number of integer or floating-point operations or the memory usage.

Definition

A real function $f(k)$ of a nonnegative real k satisfies $f = O(g)$ if there exist positive constants c_u and k_0 such that

$$f(k) \leq c_u g(k) \text{ for all } k \geq k_0. \tag{4}$$

We say that $f = \Theta(g)$ if, additionally, there exists a positive constant c_l such that

$$0 \leq c_l\, g(k) \leq f(k) \leq c_u\, g(k) \text{ for all } k \geq k_0.$$

Basic Terminology: complexity

- While $O(g)$ bounds f asymptotically from above, $\Theta(g)$ represents an asymptotically tight bound.
- As a simple illustration, consider the quadratic function

$$f(k) = \alpha * k^2 + \beta * k - \gamma.$$

  Provided $\alpha \neq 0$, $f(k) = \Theta(k^2)$ and the coefficient of the highest asymptotic term is α. Computational complexity can estimate quantities related to the worst-case behaviour of an algorithm (worst-case complexity), or it can express average behaviour (average-case complexity).
- Unit costs, Sparse matrix algorithms that are $\Theta(n^3)$ are considered to be computationally expensive.

Basic Terminology: complexity

Complexity here and in CS

- Because of the development in computations, MFLOPs may be misleading
- Still terminology $O(.)$ (bounding from above) or $\Theta(.)$ (bounding from both sides) sometimes relevant - consists in replacing the bound (bounds) by $constant \times simpler\ function$ (etalon).
- Simpler functions are, e.g., $n^2, n^3, \log n, \ldots$
- Distinguish **worst case** and **average case** analysis
- Inverse Ackermann function will be introduced in exercises
- In CS: polynomial complexity versus superpolynomial complexity. Our case: even n^3 may be too much.
- Decision problems, polynomial reduction, class $\mathcal{NP}$, etc.

Basic Terminology: complexity and sparsity

- Absolutely crucial for direct methods: complexity for generally dense matrices, sequential case: $O(n^3)$ factorization, $O(n^2)$ substitutions
- Useful for iterative methods as well: repeated multiplications and solve steps. But, expecting rather sparse matrices and (typically) dense vectors.
- Complexity in the sparse case depends on the decomposition model and computer architecture (implementation, completeness/incompleteness)

Outline

Sparse vector in a computer

Example

Consider the sparse row vector $v \in \mathbb{R}^8$

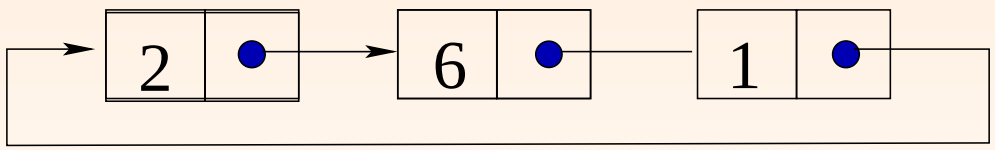
$$v = \begin{pmatrix} 1. & -2. & 0. & -3. & 0. & 5. & 3. & 0. \end{pmatrix}. \tag{5}$$

The real array `valV` that stores the nonzero values and corresponding integer array of their indices `indV` are of length $|\mathcal{S}\{v\}| = 5$ and are as follows:

Subscripts	1	2	3	4	5
`valV`	1.	$-2.$	$-3.$	5.	3.
`indV`	1	2	4	6	7

Sparse vector in a computer

- Alternatively, a linked list can be used.
- linked list - based format: stores matrix rows/columns as items connected by pointers
- linked lists can be cyclic, one-way, two-way
- A figure for demonstration, only values (not their indices) are shown



- rows/columns embedded into a larger array: emulated dynamic behavior

Sparsity

Sparse vector in a computer

- Linked list can be embedded into a large array.

Example

Two possible ways of storing the sparse vector using linked lists.

Subscripts	1	2	3	4	5
Values	1.	$-2.$	$-3.$	5.	3.
Indices	1	2	4	6	7
Links	2	3	4	5	0
Header	1				

Subscripts	1	2	3	4	5
Values	5.	3.	1.	$-2.$	$-3.$
Indices	6	7	1	2	4
Links	2	0	4	5	1
Header	3				

Sparse vectors and matrices in a computer

- Reasons for using linked lists: straightforward adds and removes.

Example

On the left, an entry -4 has been added in position 5. On the right, an entry -2 in position 2 has been removed. $*$ indicates the entry is not accessed. The links that have changed are in bold.

Subscripts	1	2	3	4	5	6
Values	1.	$-2.$	$-3.$	5.	3.	$-4.$
Indices	1	2	4	6	7	5
Links	2	3	4	5	6	0
Header	1					

Subscripts	1	2	3	4	5
Values	1.	$*$	$-3.$	5.	3.
Indices	1	$*$	4	6	7
Links	3	$*$	4	5	0
Header	1				

Sparse vectors and matrices in a computer

Sparse matrix storage

- coordinate (or triplet format: the individual entries of A are held as triplets (i, j, a_{ij}), where i is the row index and j is the column index of the entry $a_{ij} \neq 0$. (dynamic storage format)
- CSR (Compressed Sparse Row) format. The column indices of the entries of A held by rows in an integer array (which we will call `colindA`) of length $nz(A)$, with those in row 1 followed by those in row 2, and so on (with no space between rows). Sorted or unsorted. (static storage format)
- CSC (Compressed Sparse Columns): analogously by columns instead of rows.
- If A is symmetric, only the lower (or upper) triangular part is generally stored.
- Possible to store only $\mathcal{S}\{A\}$.

Sparse vectors and matrices in a computer

Sparse matrix in the coordinate format

- Example matrix $A \in \mathbb{R}^{5\times 5}$

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{pmatrix}
3. & & & -2. & \\
 & 1. & & & 4. \\
-1. & & 3. & & 1. \\
 & & & 1. & \\
 & 7. & & & 6.
\end{pmatrix}
\end{array}. \tag{6}
$$

Example

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowindA`	3	2	3	4	1	1	2	5	3	5
`colindA`	3	2	1	4	4	1	5	5	5	2
`valA`	3.	1.	-1.	1.	-2.	3.	4.	6.	1.	7.

Sparse vectors and matrices in a computer

Sparse matrix stored using linked lists

- Easy adding and deleting entries is possible if t linked lists are used: the matrix held as a collection of columns, each in a linked list. `colA_head` holds header pointers.

Example

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowindA`	3	2	3	4	1	1	2	5	3	5
`valA`	3.	1.	-1.	1.	-2.	3.	4.	6.	1.	7.
`link`	0	10	0	0	4	3	9	0	8	0
`colA_head`	6	2	1	5	7					

If we consider column 4, then $\texttt{colA\_head}(4) = 5$, $\texttt{rowindA}(5) = 1$ and $\texttt{valA}(5) = -2.$, so the first entry in column 4 is $a_{1,4} = -2.$. Next, $\texttt{link}(5) = 4$, $\texttt{rowindA}(4) = 4$ and $\texttt{valA}(4) = 1.$, so the next entry in column 4 is $a_{4,4} = 1.$.

Sparse vectors and matrices in a computer

Sparse matrix in the CSR format

- CSR format represents A as follows. Here the entries within each row are in order of increasing column index.

Example

Subscripts	1	2	3	4	5	6	7	8	9	10
`rowptrA`	1	3	5	8	9	11				
`colindA`	1	4	2	5	1	3	5	4	2	5
`valA`	3.	-2.	1.	4.	-1.	3.	1.	1.	7.	6.

- In our codes we often use: ia: rowptrA, ja: colindA, aa: valindA

Sparse vectors and matrices in a computer

Sparse matrix: static versuis dynamic formats

- 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:
  - – ad-hoc insertions/deletions should be avoided (better algorithms)
  - – much simpler to vectorize
  - – efficient access to rows/columns

Simulating dynamic storage format

- A disadvantage of linked list storage: prohibits the fast access to rows (or columns) of the matrix. And this is needed!
- Simulated dynamism of storage schemes: storage format with some additional elbow space for new non zero entries of A is needed.
- Often the case in approximate factorizations where new non zero entries can be added and/or removed and it is hard to predict the necessary space in advance.
- In this case, the elbow space can embed new non zeros.
- The format is called the DS format.

Sparse matrix: DS formats

- Consider again the sparse matrix $A \in \mathbb{R}^{5\times 5}$ (6). The DS format represents A as follows.

Example

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`rowptrA`	1	5	8	12	14									
`colindA`	1	4			2	5		1	3	5		4		2
`valAR`	3.	-2.			1.	4.		-1.	3.		1.		1.	7.
`rowlength`	2	2	3	1	2									
`colptrA`	1	4	6	9	12									
`rowindA`	1	3		2	5	3			1	4		2	3	5
`valAC`	3.	-1.		1.	7.	3.			-2.	1.		4.	1.	6.
`collength`	2	2	1	2	3									

Sparse matrix: DS formats

- It can happen that the free space between row and/or column segments disappears throughout a computational algorithm. Then the DS format must be reorganized.
- In particular, a row segment can be moved to the end of the arrays `valAR` and `colindA` implying also a corresponding update in `rowptrA`. The space where the row i originally resided is then denoted as free.
- If there is no free space at the end of the arrays `valAR` and `colindA`, a compression of the row segments or full reallocation should be done.
- While the DS format seems to be complicated, it can be extremely useful in some cases. Surprisingly efficient if the amount of changes is limited as it often is in approximate factorizations.

Block formats

- Blocked formats may be used to accelerate multiplication between a sparse matrix and a dense vector.
- The Variable Block Row (VBR) format groups together similar adjacent rows and columns.
- The data structure of the VBR format uses six arrays. Integer arrays `rptr` and `cptr` hold the index of the first row in each block row and the index of the first column in each block column, respectively. In many cases, the block row and column partitionings are conformal and only one of these arrays is needed. The real array `valA` contains the entries of the matrix block-by-block in column-major order. The integer array `indx` holds pointers to the beginning of each block entry within `valA`. The index array `bindx` holds the block column indices of the block entries of the matrix and, finally, the integer array `bptr` holds pointers to the start of each row block in `bindx`.

Sparse matrix: DS formats

Example

Consider the sparse matrix $A \in \mathbb{R}^{8\times 8}$

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\left(\begin{array}{cccccccc}
1. & 2. & & & & 3. & & \\
4. & 5. & & & & 6. & & \\
 & & 7. & 8. & 9. & 10. & & \\
11. & 12. & & & & & 15. & 16. \\
 & 13. & & & & & 17. & \\
14. & & & & & & & 18. \\
 & & 19. & & 20. & & & \\
 & & 21. & 22. & & & &
\end{array}\right)
\end{array}.
$$

Here the row blocks comprise rows 1:2, 3, 4:6 and 7:8. The column blocks comprise columns 1:2, 3:5, 6, 7:8. The VBR format stores A as follows.

Sparse matrix: DS formats

Example

Subscripts	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`rptr`	1	3	4	7	9												
`cptr`	1	3	6	7	9												
`valA`	1.	4.	2.	5.	3.	6.	7.	8.	9.	10.	11.	14.	12.	13.	15.	17.	16.
`indx`	1	5	7	10	11	15	19										
`bindx`	1	3	2	3	1	4	2										
`bptr`	1	3	5	7													

Matmats in CSR/CSC

1) CSR - CSC

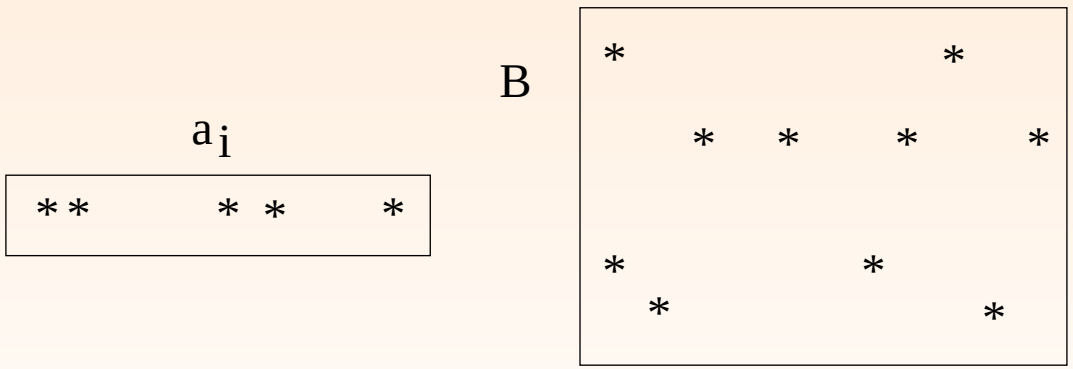
$$C = AB, A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}, B = \begin{pmatrix} b_1, \ldots, b_n \end{pmatrix}, C = (c_{ij}) \tag{7}$$

- Each entry c_{ij} computed as a product of a compressed row of A and compressed column of B
- Not clear whether the result c_{ij} is nonzero
- Consequently: $O(n^3)$ operations, not useful for sparse matrices.

Matmats in CSR/CSC

2) CSR - CSR

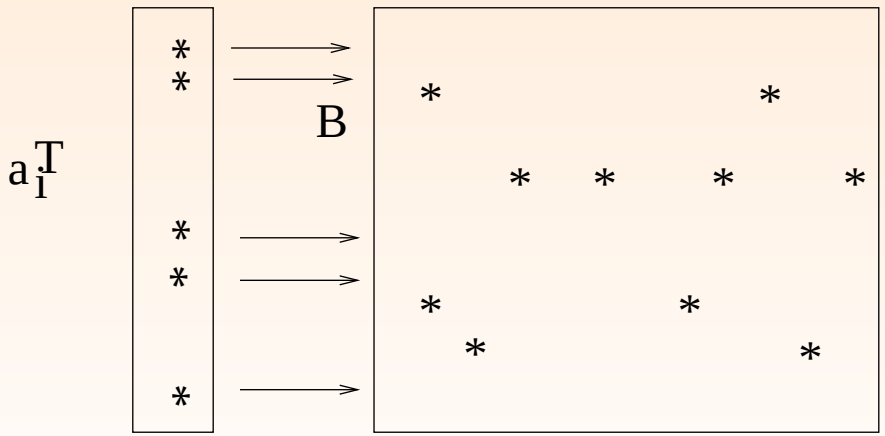
$$C = AB, A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}, B = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, C = (c_{ij}) \tag{8}$$



Matmats in CSR/CSC

2) CSR - CSR

$$C = AB, A = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}, B = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, C = (c_{ij}) \tag{9}$$



Matmats in CSR/CSC

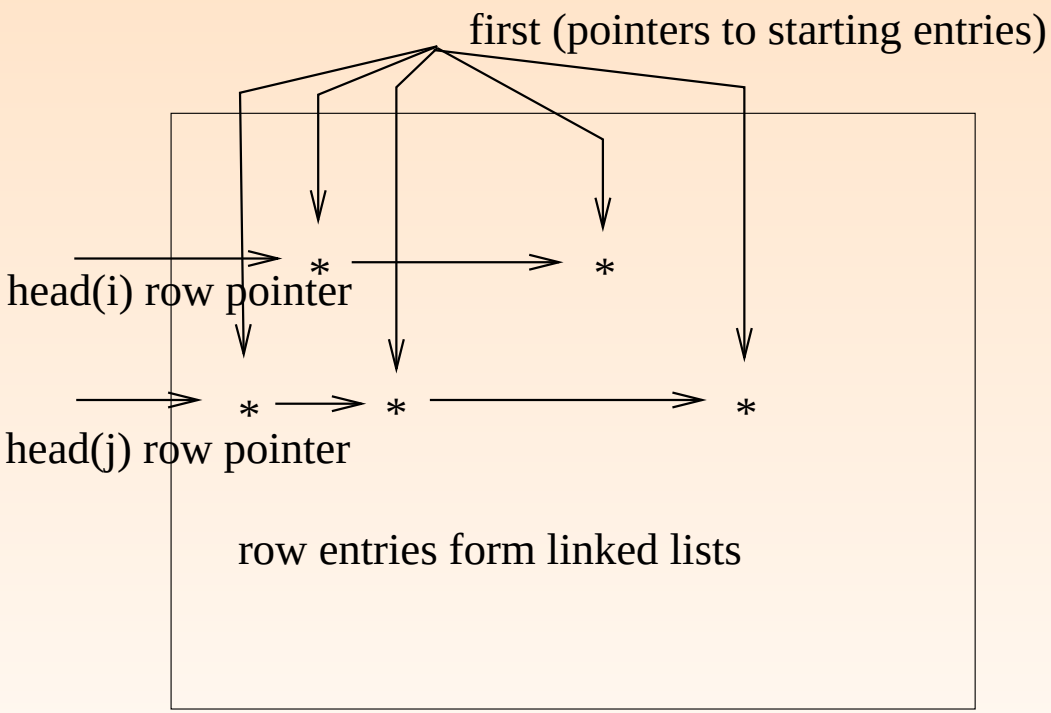
3) CSC - CSR

$$C = AB, A = \left(a_1, \ldots, a_m\right), B = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}, C = (c_{ij}) \tag{10}$$

- How one can store A by CSC and pass it by rows?
- Pointers to first entries in columns: (array $first$)
- First test: nonzero in the first row $\rightarrow$ move one step down, add next nonzero into the list $value(next)$
- Complexity: $O(nonzeros) + O(n)$

Matmats in CSR/CSC

3) CSC - CSR



Based on forming virtual rows in A

Outline

Graphs and sparse matrices

Directed and undirected graphs

- Many sparse matrix algorithms exploit the relationship matrices $\leftrightarrow$ graphs.
- A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a finite set $\mathcal{V}$ of vertices (or nodes), and a set $\mathcal{E}$ of edges defined as pairs of distinct vertices.
- No distinction between the pairs (u, v) and (v, u): the edges are represented by unordered pairs, the graph is undirected.

Definition

Simple undirected graph G is an ordered pair of sets (V, E), where $V = \{v_1, \ldots, v_n\}$ is called the set of vertices of G, $E = \{e_1, \ldots, e_m\}$ is called the set of edges satisfying (no multiple edges, no loops)

$$E \subseteq \binom{V}{2}.$$

- If the pairs are ordered: directed graph (a digraph).

Directed and undirected graphs

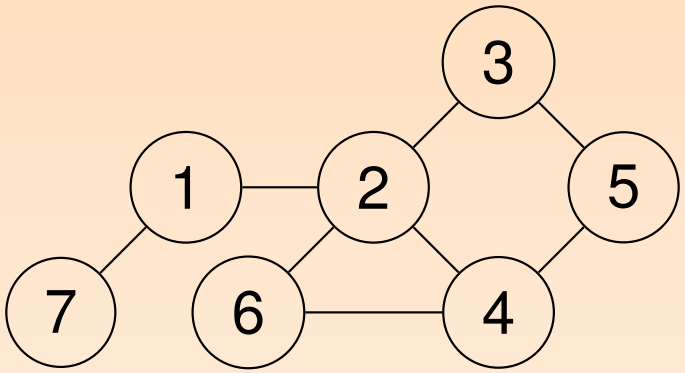


Figure: *An example of an undirected graph.*

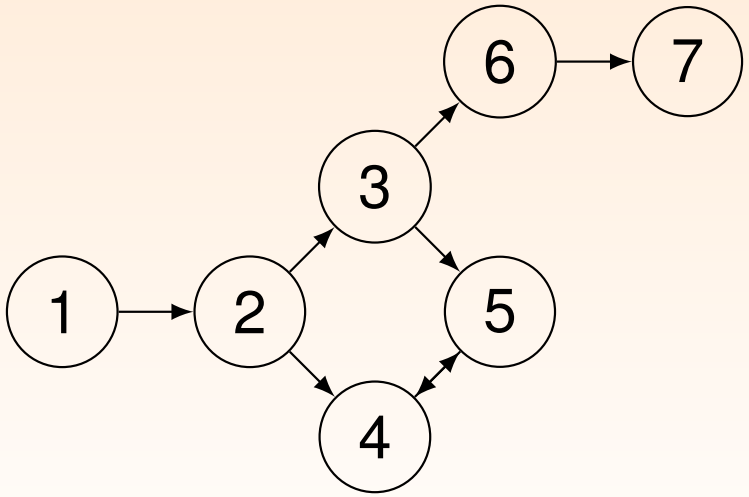
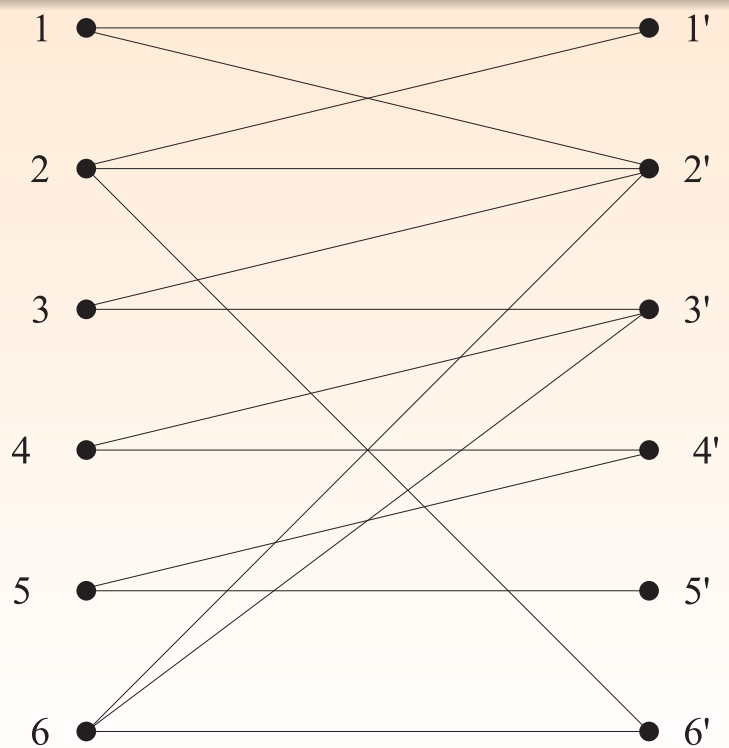


Figure: *An example of a directed graph (digraph). The arrows indicate the direction of an edge. There is an edge* $(4 \to 5)$ *and an edge* $(5 \to 4)$.

Bipartite graphs

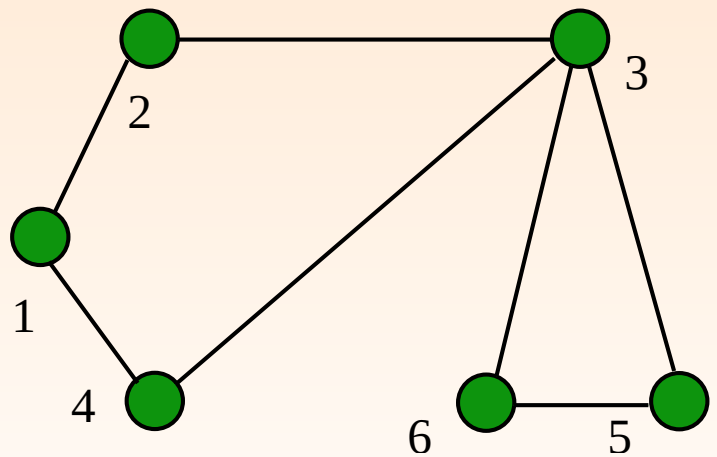
Definition

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.



Graph terminology

- A labelling (or ordering) of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with n vertices is a bijection of $\{1, 2, \ldots, n\}$ onto $\mathcal{V}$. The integer i ($1 \leq i \leq n$) assigned to a vertex in $\mathcal{V}$ is called the label (or simply the number).
- Our standard choice of vertices will be $\mathcal{V} = \{1, \ldots, n\}$ so that the vertices are directly identified by their labels.
- Another example of a labelled undirected graph:



Simple undirected graph $G = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2\}, \{2, 3\}, \{1, 4\}, \{3, 4\}, \{3, 6\}, \{3, 5\}, \{5, 6\}\})$

Graph terminology

- $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ is a subgraph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if and only if $\mathcal{V}_s \subseteq \mathcal{V}$ and $\mathcal{E}_s \subseteq \mathcal{E}$ and $(u_s, v_s) \in \mathcal{E}_s$ implies $u_s, v_s \in \mathcal{V}_s$.
- The subgraph is an induced subgraph if $\mathcal{E}_s$ contains all the edges in $\mathcal{E}$ that have both u and v in $\mathcal{V}_s$.
- Two graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{G}_s = (\mathcal{V}_s, \mathcal{E}_s)$ are isomorphic if there is a bijection $g : \mathcal{V} \to \mathcal{V}_s$ that preserves adjacency, that is $(u, v) \in \mathcal{E}$ if and only if $(g(u), g(v)) \in \mathcal{E}_s$.
- Undirected graph: two vertices u and v in $\mathcal{V}$ are said to be adjacent (or neighbours) if $e = (u, v) \in \mathcal{E}$; the edge e is incident to the vertex u and to the vertex v. u and v are the endpoints of e.
- We also use the notation $(u \longleftrightarrow v)$ for an edge (or $(u \overset{\mathcal{G}}{\longleftrightarrow} v)$ to emphasise the edge belongs to the graph $\mathcal{G}$).

Graph terminology

- The degree $deg_{\mathcal{G}}(u)$ of $u \in \mathcal{V}$ is the number of vertices in $\mathcal{V}$ that are adjacent to u, and the adjacency set $adj_{\mathcal{G}}\{u\}$ is the set of these adjacent vertices (thus $|adj_{\mathcal{G}}\{u\}| = deg_{\mathcal{G}}(u)$).
- If $\mathcal{V}_s$ is a subset of the vertices, then the adjacency set $adj_{\mathcal{G}}\{\mathcal{V}_s\}$ is the set of vertices in $\mathcal{V} \setminus \mathcal{V}_s$ that are adjacent to at least one vertex in $\mathcal{V}_s$.
- A subgraph is a clique when every pair of vertices is adjacent. In the example in Figure 4, $deg_{\mathcal{G}}(2) = 4$ and $adj_{\mathcal{G}}\{2\} = \{1, 3, 4, 6\}$. The induced subgraph with vertices $\mathcal{V}_s = \{2, 4, 6\}$ is a clique.

Graph terminology

- Notation $(u \to v)$ for a directed edge.
- Emphasising the graph to which the edge belongs: $(u \xrightarrow{\mathcal{G}} v)$.
- In a digraph there can be an edge $(u \to v)$ but no edge $(v \to u)$.
- The adjacency set of u can be split into two parts

$$adj^{+}_{\mathcal{G}}\{u\} = \{v \mid (u \to v) \in \mathcal{E}\} \quad \text{and} \quad adj^{-}_{\mathcal{G}}\{u\} = \{v \mid (v \to u) \in \mathcal{E}\}.$$

  In the example given in Figure: $adj^{+}_{\mathcal{G}}\{2\} = \{3, 4\}$, $adj^{-}_{\mathcal{G}}\{2\} = 1$.
- A sequence of k edges in an undirected graph $\mathcal{G}$: a walk.

$$u_0 \longleftrightarrow u_1 \longleftrightarrow \ldots \longleftrightarrow u_{k-1} \longleftrightarrow u_k$$

- If $\mathcal{G}$ is a digraph then the sequence: a directed walk.

$$u_0 \longrightarrow u_1 \longrightarrow \ldots \longrightarrow u_{k-1} \longrightarrow u_k$$

- k is the length of the walk. A walk of zero length: $k = 0$.

Graph terminology

- The vertices u_0 and u_k are connected by the walk and for $k > 0$, u_k is said to be reachable from u_0;
- The set of vertices that are reachable from u_0 is denoted by $\mathcal{R}each(u_0)$. The walk is closed if $u_0 = u_k$; a closed walk is called a cycle.
- Graphs that do not contain cycles are acyclic.
- A (directed) trail is a (directed) walk in which all the edges are distinct and a (directed) path is a (directed) trail in which all the vertices (and therefore also all the edges) are distinct.
- The distance between two vertices is the number of edges in the shortest path connecting them (this is also called the length of the path).

Graph terminology

- If the vertices $\mathcal{V}$ are labelled $1, 2, \ldots, n$ then in the undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a path between a pair of its vertices with labels i and j is denoted by

$$i \overset{\mathcal{G}}{\Longleftrightarrow} j$$

or, if it is clear which graph the path is in, by

$$i \Longleftrightarrow j.$$

If all intermediate vertices on the path are less than $\min\{i, j\}$ then the path is called a fill-path and is denoted by

$$i \underset{min}{\overset{\mathcal{G}}{\Longleftrightarrow}} j \quad \text{or} \quad i \underset{min}{\Longleftrightarrow} j.$$

Graph terminology

- If all intermediate vertices on the path belong to a subset $\mathcal{V}_s$ then the path is denoted by

$$i \overset{\mathcal{G}}{\underset{\mathcal{V}_s}{\Longleftrightarrow}} j \quad \text{or} \quad i \underset{\mathcal{V}_s}{\Longleftrightarrow} j.$$

If $\mathcal{G}$ is a digraph, the double-sided arrow symbols are replaced by one-sided ones $\Longrightarrow$ in the direction of the edges. For example,

$$i \overset{\mathcal{G}}{\Longrightarrow} j, \quad i \Longrightarrow j, \quad i \underset{min}{\Longrightarrow} j \quad \text{and} \quad i \underset{\mathcal{V}_s}{\Longrightarrow} j.$$

Graph terminology

- Digraph with no cycles (directed acyclic graph): DAG.
- In a DAG, if there is a path $u \Longrightarrow v$ then u is called an ancestor of v and v is said to be a descendant of u.
- DAG with two different orderings. Left: vertices 2, 3, 5 and 6 are descendants of 1. Only vertices 5 and 6 are descendants of vertex 4.

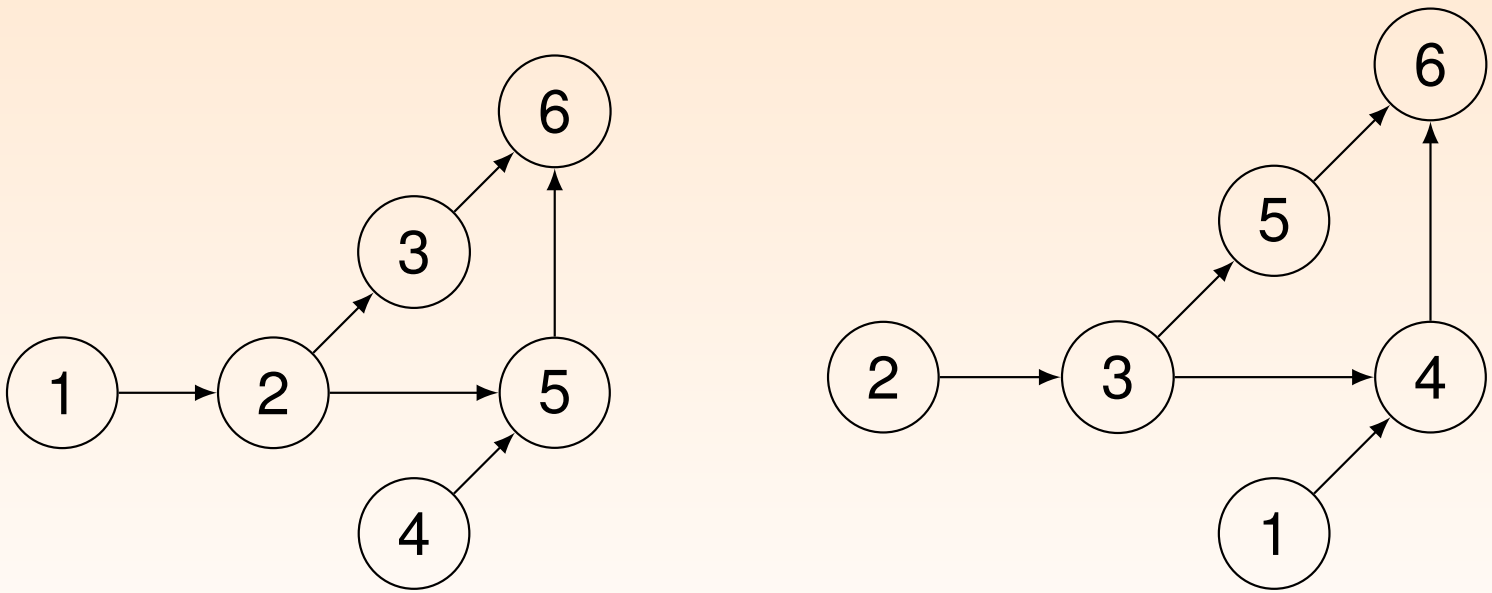


Figure: *An example of a DAG with two different topological orderings.*

Graph terminology: reachability

- Given a graph and a subset $\mathcal{V}_s$ of vertices, if u and v are two distinct vertices that do not belong to $\mathcal{V}_s$, then v is reachable from u through $\mathcal{V}_s$ if u and v are connected by a path that is either of length 1 or is composed entirely of vertices that belong to $\mathcal{V}_s$ (except for the endpoints u and v).
- Given $\mathcal{V}_s$ and $u \notin \mathcal{V}_s$, the reachable set $\mathcal{R}each(u, \mathcal{V}_s)$ of u through $\mathcal{V}_s$ is the set of all vertices that are reachable from u through $\mathcal{V}_s$. Note that if $\mathcal{V}_s$ is empty or u does not belong to $adj_{\mathcal{G}}(\mathcal{V}_s)$ then $\mathcal{R}each(u, \mathcal{V}_s) = adj_{\mathcal{G}}(u)$.
- A simple example is given in Figure.

Graph terminology: reachability

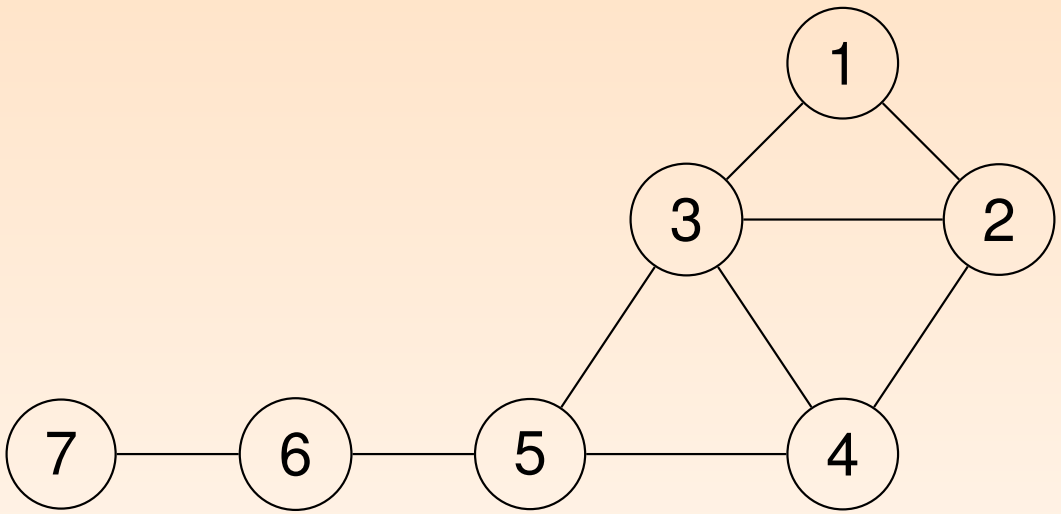


Figure: *An example of an undirected graph to illustrate reachability. If* $\mathcal{V}_s = \{4, 5\}$ *then* $\mathcal{R}each(2, \mathcal{V}_s) = \{1, 3, 6\}$ *and* $\mathcal{R}each(6, \mathcal{V}_s) = \{2, 3, 7\}$.

Graph terminology: connectedness, trees

- An undirected graph is connected if every pair of vertices is connected by a path.
- A connected acyclic graph is called a tree, that is, a tree is an undirected graph in which any two vertices are connected by exactly one path.
- Every tree has at least two vertices of degree 1. Such vertices are called leaf vertices.
- A graph is a forest if it consists of a disjoint union of trees. This is illustrated in Figure. Connectivity is an equivalence relation and consequently, it provides a partition of $\mathcal{V}$ into disjoint equivalence classes.

Graph terminology: connectedness, trees

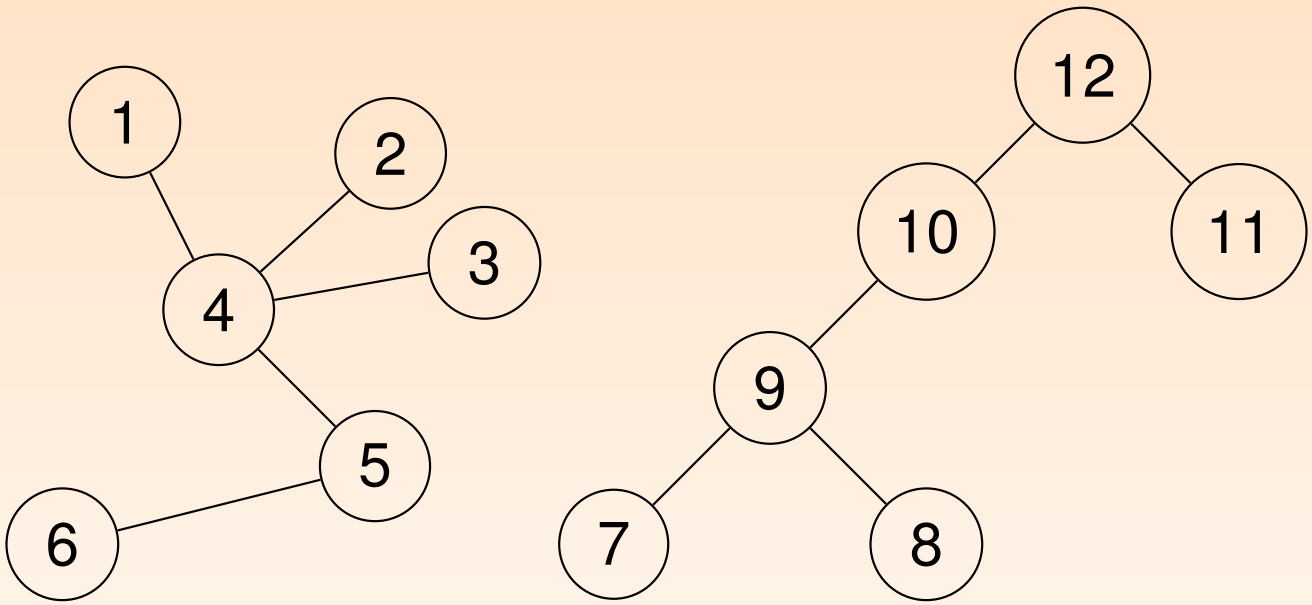


Figure: *An example of an undirected graph with 12 vertices that is a forest (it consists of two disjoint trees). Vertices 1, 2, 3, 6, 7, 8 and 11 are leaf vertices.*

Graph terminology: connected, strongly connected

- If $\mathcal{G}$ is connected then a spanning tree of $\mathcal{G}$ is a subgraph of $\mathcal{G}$ that is a tree containing every vertex of $\mathcal{G}$.
- A directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is strongly connected if for every pair of vertices $u, v \in \mathcal{V}$ there is a path from u to v and a path from v to u.
- Strong connectivity is an equivalence relation on $\mathcal{V}$. It induces a partitioning $\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_s$ such that each $\mathcal{V}_i$ $(1 \leq i \leq s)$ is strongly connected and is maximal with this property: no additional vertices from $\mathcal{G}$ can be included in $\mathcal{V}_i$ without breaking its strong connectivity.
- The $\mathcal{V}_i$ are called strongly connected components (or sometimes just strong components) of $\mathcal{G}$.

Graph terminology: rooted trees

- Any undirected tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ can be converted to a directed rooted tree $\mathcal{T}' = (\mathcal{V}, \mathcal{E}')$ by specifying a root vertex r.
- r can be chosen arbitrarily: any choice gives a directed rooted tree. An edge with endpoints u and v in $\mathcal{E}$ becomes a directed edge $(u \to v)$ in $\mathcal{E}'$ if there is a path from u to r such that the first edge of this path is from u to v.
- A rooted tree is a special case of a DAG.
- v is called the parent of u if the directed edge $(u \to v) \in \mathcal{E}'$; u is said to be a child of v (two or more child vertices are referred to as children). Two vertices in a rooted tree are siblings if they have the same parent. Leaf vertices have no children.
- Given r, this directed path is unique.

Graph terminology: rooted trees

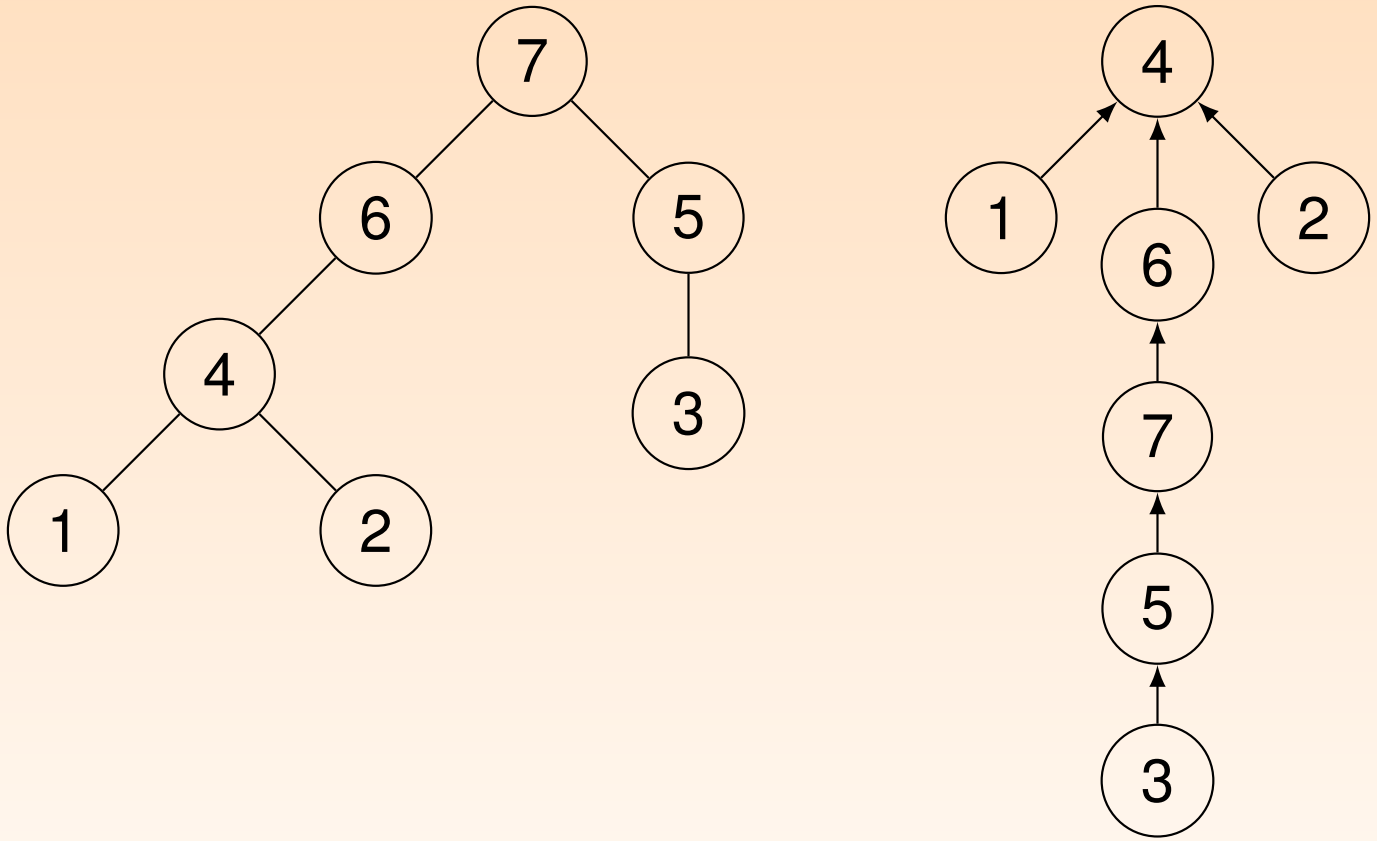
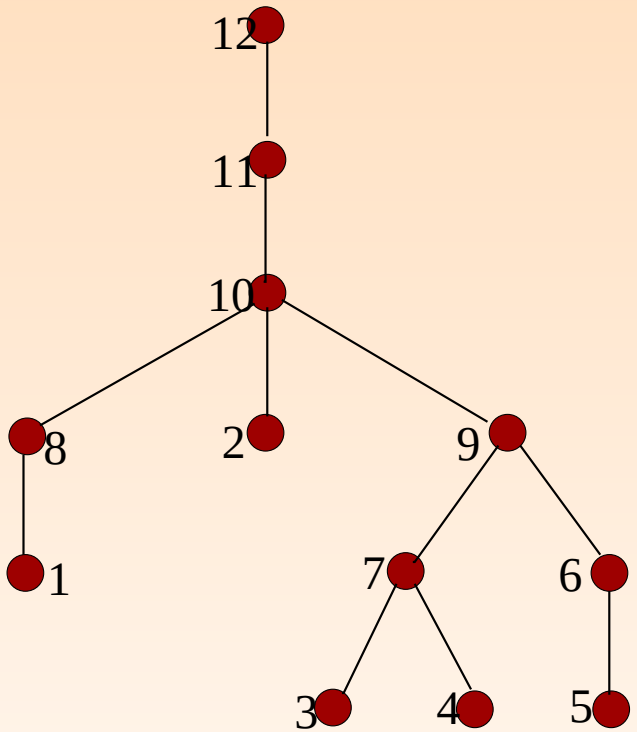


Figure: *An example of a an undirected tree $\mathcal{T}$ (left) and the rooted tree $\mathcal{T}'$ (right) obtained from $\mathcal{T}$ by choosing the root $r = 4$. The arrows indicate the direction of the edges.*

Graphs and their matrices

Another example of a rooted tree



- The root of this tree is 12
- Then, for example, 10 is an ancestor of vertices 2, 8 a 9. These vertices are descendants of 10. Set of ancestors of 10 is $anc(10) = \{10, 11, 12\}$. $parent(i)$ for vertices $1 \ldots 11$ is $8, 10, 7, 7, 6, 9, 9, 10, 10, 11, 12$, null.

Graphs and their matrices

- Adjacency graphs provide a link between sparse matrices and graphs. If A is a sparse matrix of order n then an adjacency graph $\mathcal{G}(A)$ (often written simply as $\mathcal{G}$) with n vertices $\mathcal{V}(A) = \{1, \ldots, n\}$ can be associated with it.
- If A is structurally symmetric, then the edge set is

$$\mathcal{E}(A) = \{(i, j) \mid a_{ij} \neq 0,\ i \neq j\}\,.$$

- A digraph can be associated with a nonsymmetric A by setting

$$\mathcal{E}(A) = \{(i \rightarrow j) \mid a_{ij} \neq 0,\ i \neq j\}.$$

  Each diagonal nonzero a_{ii} corresponds to a loop or self edge. Loops are generally omitted from $\mathcal{G}$ since many algorithms that use A assume that the diagonal entries of A are present.
- To capture not only the sparsity pattern of A but also the values of the entries, $\mathcal{G}$ can be transformed into a weighted graph using a mapping $\mathcal{E}(A) \rightarrow \mathbb{R}$ and/or $\mathcal{V}(A) \rightarrow \mathbb{R}$.

Adjacency matrix of an undirected graph

Definition

For a simple undirected graph $G = (V, E)$ with $V = \{1, \ldots, n\}$ the adjacency matrix is the $(0, 1)$ matrix $A_G = (a_{ij})$ $(i, j \in V)$, where a_{ij} is 0 if there is an edge $\{i, j\}$ in E where $i, j \in V$ and zero otherwise.

$$\begin{pmatrix} & 1 & & 1 & & \\ 1 & & 1 & & & \\ & 1 & & 1 & 1 & 1 \\ 1 & & 1 & & & \\ & & 1 & & & 1 \\ & & 1 & & 1 & \end{pmatrix}$$

Adjacency matrix for the undirected graph above.

Incidence matrix of an undirected graph

Definition

For a simple undirected graph $G = (V, E)$ with $V = \{1, \ldots, n\}$ and $E = \{1, \ldots, m\}$ the incidence matrix is the $(0, 1)$ matrix $A_G = (a_{ij})$ $(i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\})$, where $a_{ij} = 1$, if j is the vertex of the edge i and $a_{ij} = 0$ otherwise.

- Incidence matrix is generally rectangular.
- Several slightly differing definitions of adjacency and incidence matrices.
- Various generalizations.

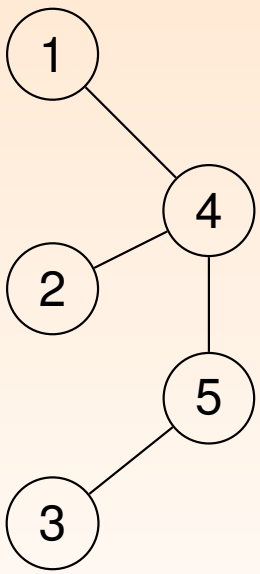
Graphs and their matrices

Transfer between the classes of undirected and directed graphs

- Symmetrization: directed $\to$ undirected
  - Just considering edges from $V \times V$ as from $\binom{V}{2}$
- Orientation: undirected $\to$ directed
  - Not unique. Instead from an edge from $\binom{V}{2}$ we can have one or two edges from $V \times V$.
- In any case, part of terminology is shared between the classes of undirected and directed graphs

Graphs and their matrices

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & & & * & \\ & * & & * & \\ & & * & & * \\ * & * & & * & * \\ & & * & * & * \end{pmatrix} \end{array}$$



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & & & * & \\ & * & & * & \\ & & * & * & \\ * & & & * & * \\ & & * & & * \end{pmatrix} \end{array}$$

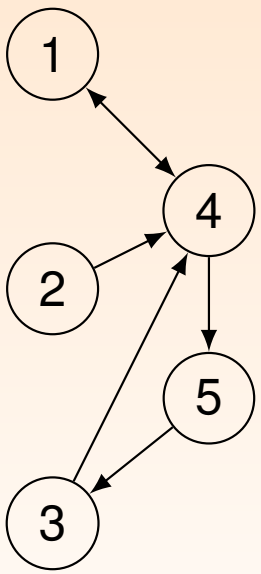


Figure: *An example of a structurally symmetric sparse matrix and its undirected graph (left) and a nonsymmetric sparse matrix and its digraph (right). Arrows indicate the direction of the edges in the digraph.*

Graphs of triangular matrices

- A special case is the directed graph associated with a triangular matrix. If L is a lower triangular matrix and U is an upper triangular matrix then the directed graphs $\mathcal{G}(L)$ and $\mathcal{G}(U)$ have edge sets

$$\mathcal{E}(L) = \{(i \to j) \,|\, l_{ij} \neq 0,\ i > j\}$$
$$\mathcal{E}(U) = \{(i \to j) \,|\, u_{ij} \neq 0,\ i < j\}$$

- It is sometimes convenient to use $\mathcal{G}(L^T)$ in which the direction of the edges is reversed

$$\mathcal{E}(L^T) = \{(j \to i) \,|\, l_{ij} \neq 0,\ i > j\}. \tag{11}$$

It is straightforward to see that $\mathcal{G}(L)$, $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$ are DAGs.

Permutation matrices

- A permutation matrix is a square matrix that has exactly one entry equal to 1 in each row and column, and all remaining entries are zeros (that is, it is a permutation of the identity matrix).
- Premultiplying a matrix by P reorders the rows and postmultiplying by P reorders the columns. P can be represented by an integer-valued permutation vector p, where p_i is the column index of the 1 within the i-th row of P. For example,

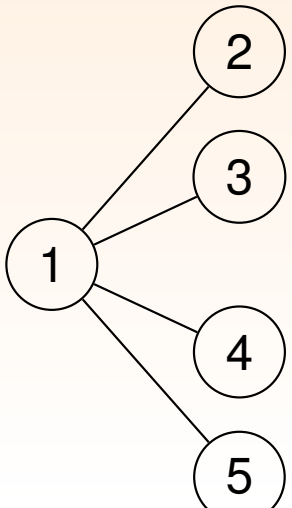
$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad p = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

- The graph of a matrix A is unchanged if a symmetric permutation $A' = PAP^T$ is performed, only the labelling of the vertices changes and thus relabelling $\mathcal{G}(A)$ can be used to permute A.

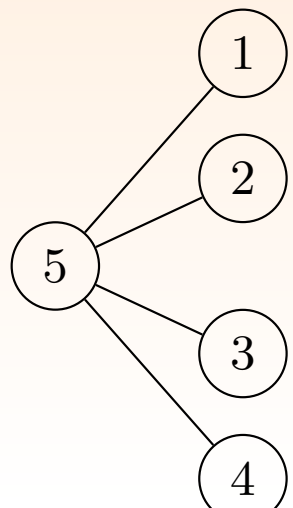
Graphs of arrowhead matrices

Consider the arrowhead matrix A and its graph $\mathcal{G}(A)$. The symmetrically permuted matrix A' and $\mathcal{G}(A')$ are also shown, with P such that the first row and column of A are the last row and column of A'.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix} \end{array}$$



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix} \end{array}$$



Topological orderings of digraphs

- The digraph of a general matrix A is not invariant under nonsymmetric permutations PAQ, with $Q \neq P^T$. A topological ordering of a directed graph is a labelling of its vertices such that for every edge $(i \rightarrow j)$, vertex i precedes vertex j (i.e., $i < j$).
- A topological ordering is possible if and only if the graph has no directed cycles, that is, if it is a DAG. Any DAG has at least one topological ordering.
- Non-unique (see above).

Lists

- A list is an ordered sequence of arbitrary elements

$$(u_0, u_1, \ldots, u_{k-1}, u_k) \tag{12}$$

  u_0 is the head of the list and u_k is its tail. An empty list: $()$.
- A stack is a list in which elements can only be added to or removed from the head. A pointer locates the head of the stack. Let $S = (u_0, u_1, \ldots, u_{k-1}, u_k)$ be a stack. $push(S, v)$ denotes adding v onto the stack by incrementing the pointer by one, giving $(v, u_0, \ldots u_k)$. $pop(S, u_0)$ denotes the stack $(u_1, \ldots u_k)$ that results from decreasing the pointer by one (removing u_0 from the head).
- A queue is a list in which elements can be added to the tail (appended) or removed (popped) from the head. Two pointers locate the head and the tail. Consider the queue $\mathcal{Q} = (u_0, u_1, \ldots, u_{k-1}, u_k)$. The append operation $append(\mathcal{Q}, u_{k+1})$ results in the queue $(u_0, \ldots u_k, u_{k+1})$ and the pop operation $pop(\mathcal{Q}, u_0)$ results in the queue $(u_1, \ldots u_k)$.

Stack and queue once more

Definition

List is called queue, if it enables efficient

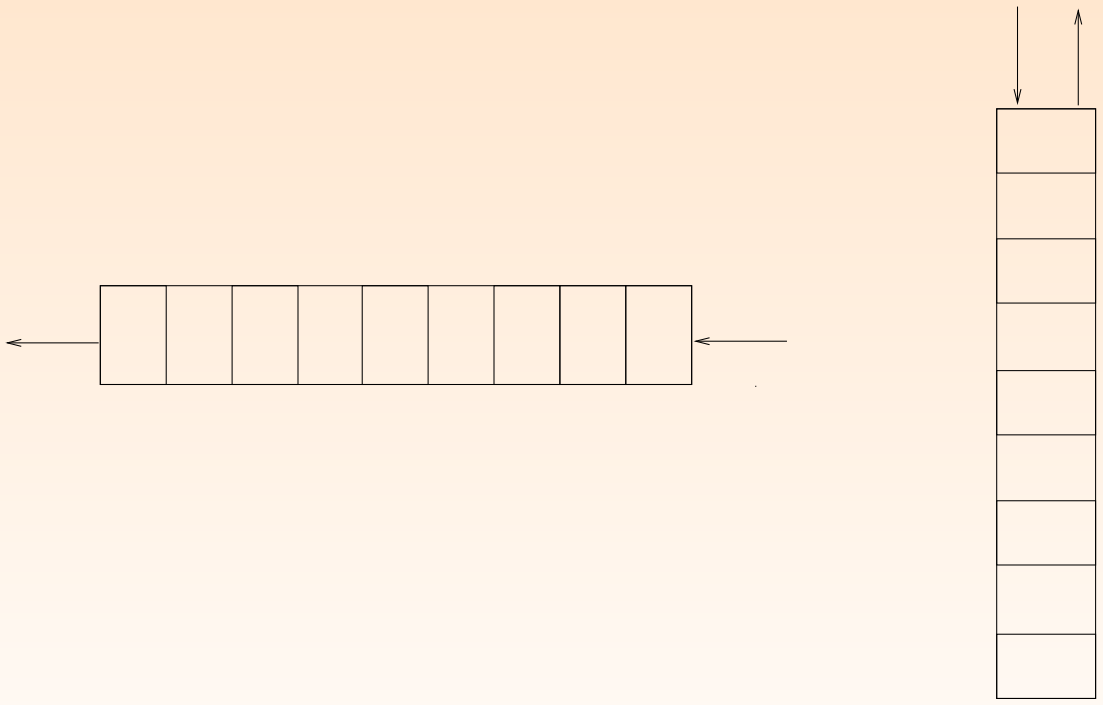
- access head of a list,
- deletion of the list head (pop) and
- adding an element behind the current tail.

List is called stack, if it enables efficient

- access head of a list,
- deletion of the list head (pop) and
- adding and element before the current head (push).

Stack and queue

Queue and stack are schematically depicted below. The arrows represent efficient (easily implementable) operations.



Searching the adjacency graph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$

- The sequence in which the vertices are visited can be used to reorder the graph and hence permute the matrix.
- Given a start vertex, a graph search (graph traversal) performs exploration of the vertices and edges of $\mathcal{G}(A)$
- It generates sets of visited vertices and explored edges.
- $\mathcal{V}_v$: the set of visited vertices, $\mathcal{V}_n$: the set of vertices not yet visited.
- The search: selects an unexplored edge in $\mathcal{E}$ with vertices in $\mathcal{V}_v$. If the other vertex belongs to $\mathcal{V}_n$ then this vertex is moved into $\mathcal{V}_v$ and the edge is flagged as explored.
- The explored edge may be directed or undirected; in an undirected graph, the edge (v, w) formally corresponds to the pair of edges $(w \rightarrow v)$ and $(v \rightarrow w)$.

Graphs and their matrices

- Starting from a chosen start vertex s, a breadth-first search (BFS) explores all the vertices adjacent to s.
- Then all the vertices whose shortest path from s is of length 2, and then length 3, and so on
- A queue is used in its implementation.
- The search terminates when there are no unexplored edges (v, w) with $v \in \mathcal{V}_v$ and $w \in \mathcal{V}_n$ that are reachable from s.

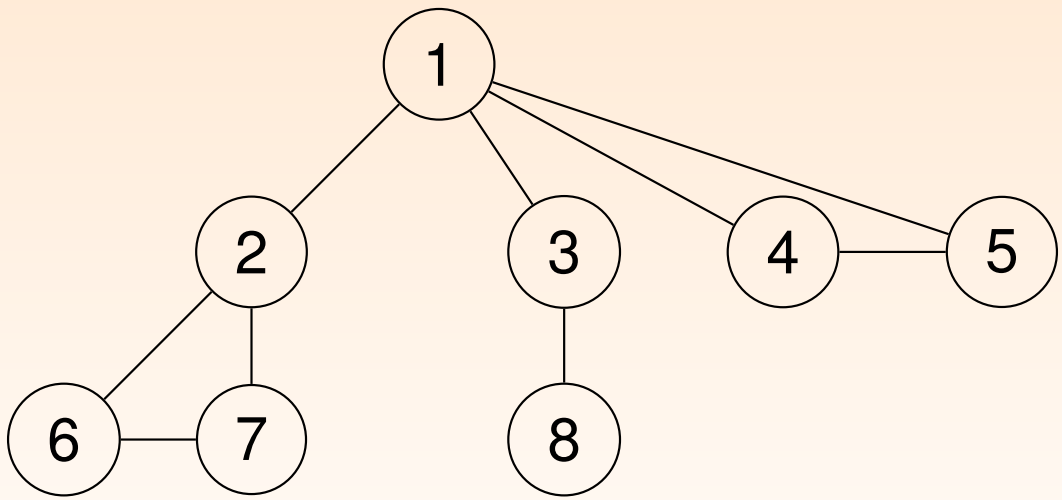


Figure: *An illustration of a BFS of a connected undirected graph, with the labels indicating the order in which the vertices are visited.*

DFS

- A depth-first search (DFS) visits child vertices before visiting sibling vertices
- Starting from a chosen vertex s, the set of vertices that are visited are those vertices u for which a directed path from s to u exists in $\mathcal{G}$.
- Different results depending on s and how ties are broken.
- Like the BFS, all vertices in $\mathcal{R}each(s)$ are visited.
- Traversed edges form a DFS spanning tree. Visiting all the edges of a graph results in a DFS forest that consists of exactly one DFS spanning tree for each connected component.

DFS

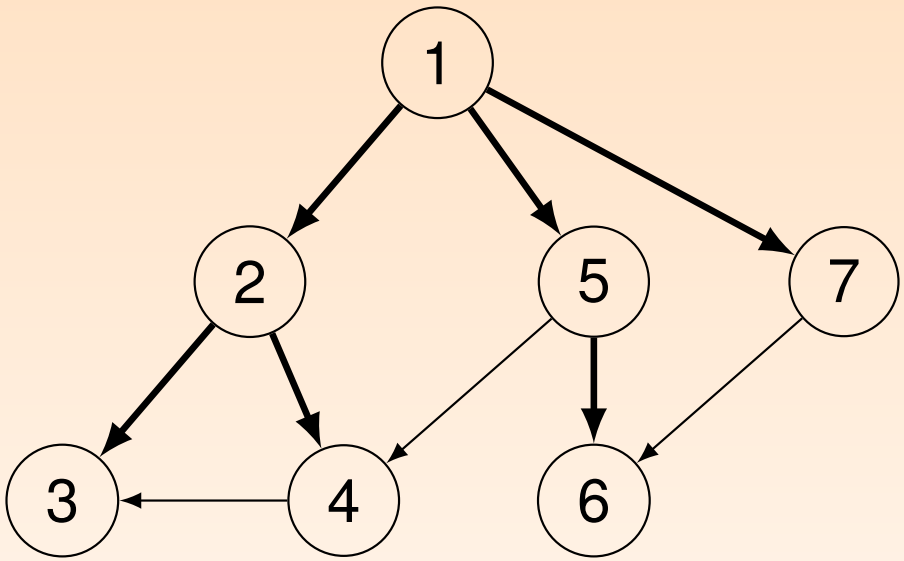


Figure: *An illustration of a DFS of a connected directed graph. The labels indicate the order in which the vertices are visited. The edges of the DFS spanning tree are in bold.*

DFS

- Number of ways to construct the output vertex order for a DFS.
- Given a start vertex s, in a preorder list, the vertices are returned in the order in which they are added into $\mathcal{V}_v$
- In a postorder list, the vertices are in the order in which they are last visited during the DFS algorithm
  For the example in Figure, the vertices are added into $\mathcal{V}_v$ in the order $1, 2, 3, 4, 5, 6, 7$ and this is the preorder list.
- The sequence in which the DFS visits the vertices is $1, 2, 3, 2, 4, 2, 1, 5, 6, 5, 1, 7, 1$. In this sequence, vertex 3 is the first vertex to appear for the last time so the postordering starts with vertex 3. The next vertex to appear for the last time is vertex 4, followed by vertex 2, and so on, resulting in the postorder list $3, 4, 2, 6, 5, 7, 1$.

Algorithm (Find preorder and postorder lists using a DFS)

1: $\mathcal{V}_v = \emptyset$, $preorder = ()$ and $postorder = ()$
2: **for all** $v \in \mathcal{V}$ **do**
3: **if** $v \notin \mathcal{V}_v$ **then**
4: $push(preorder, v)$ ▷ Add v onto the preorder stack
5: $\mathcal{V}_v = \mathcal{V}_v \cup \{v\}$ ▷ Add v to the set of visited vertices
6: dfs_step(v)
7: **end if**
8: **end for**
9: **recursive function** (dfs_step(v))
10: **for all** $(v \to w) \in \mathcal{E}$ **do**
11: **if** $w \notin \mathcal{V}_v$ **then**
12: $push(preorder, w)$ ▷ Add w onto the preorder stack
13: $\mathcal{V}_v = \mathcal{V}_v \cup \{w\}$ ▷ Add w to the set of visited vertices
14: dfs_step(w) ▷ recursive search
15: **end if**
16: **end for**
17: $push(postorder, v)$ ▷ Add v onto the postorder stack
18: **end recursive function**

Outline

Introduction to factorizations

- Entries of factors outside $\mathcal{S}\{A\}$: filled entries. Together the filled entries are called the fill-in.
- Adding the fill-in to $\mathcal{G}(A)$: the filled graph.
- Numerical cancellations in LU factorizations rarely happen. Also difficult to predict.
- non-cancellation assumption: assuming that the result of adding, subtracting or multiplying two nonzeros is nonzero. It follows that if $A = LU$ and $\mathcal{E}_L$ denotes the set of (directed) edges of the digraph $\mathcal{G}(L)$ then for $i > j$

$$a_{ij} \neq 0 \text{ implies } (i \to j) \in \mathcal{E}_L.$$

Observation

The sparsity structures of the LU factors of A satisfy

$$\mathcal{S}\{A\} \subseteq \mathcal{S}\{L + U\}.$$

Introduction to factorizations

- The traditional way of describing Gaussian elimination is based on the systematic annihilation of the entries in the lower triangular part of A column-by-column.
- Assuming A is factorizable, this can be written formally as sequential multiplications by column elimination matrices
- Getting the elimination sequence:

$$A = A^{(1)}, A^{(2)}, \ldots, A^{(n)} \tag{13}$$

of partially eliminated matrices as follows:

$$A^{(1)} \to A^{(2)} = C_1 A^{(1)} \to A^{(3)} = C_2 C_1 A^{(1)} \to \ldots \to A^{(n)} = C_{n-1} \ldots C_1 A^{(1)}$$

- The unit lower triangular matrices C_i $(1 \leq i \leq n-1)$ are the column elimination matrices.

Factorizations

Elementwise, assuming $a_{11} = a_{11}^{(1)} \neq 0$, the first step $C_1 A^{(1)} = A^{(2)}$ is

$$\begin{pmatrix} 1 & & & & \\ -a_{21}^{(1)}/a_{11}^{(1)} & 1 & & & \\ -a_{31}^{(1)}/a_{11}^{(1)} & & 1 & & \\ \vdots & & & 1 & \\ -a_{n1}^{(1)}/a_{11}^{(1)} & & & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & \dots & a_{3n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix},$$

Provided $a_{22}^{(2)} \neq 0$, the second step $C_2 A^{(2)} = A^{(3)}$ is

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & -a_{32}^{(2)}/a_{22}^{(2)} & 1 & & \\ & \vdots & & 1 & \\ & -a_{n2}^{(2)}/a_{22}^{(2)} & & & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ 0 & a_{32}^{(2)} & \dots & a_{3n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{(2)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \dots & \dots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \dots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \dots & a_{nn}^{(3)} \end{pmatrix}.$$

Introduction to factorizations

- The k-th partially eliminated matrix is $A^{(k)}$.
- The active entries in $A^{(k)}$: $a_{ij}^{(k)}$, $1 \leq k \leq i, j \leq n$. The submatrix of $A^{(k)}$ with the active entries: active submatrix.
- $\mathcal{G}(A^{(k)})$ is the k-th elimination graph and is denoted by $\mathcal{G}^k$. If $\mathcal{S}\{A\}$ is nonsymmetric then $\mathcal{G}^k$ is a digraph.
- The inverse of each C_k is the unit lower triangular matrix obtained by changing the sign of all the off-diagonal entries.
- The product of unit lower triangular matrices (beware the order) is a unit lower triangular matrix: provided $a_{kk}^{(k)} \neq 0$ $(1 \leq k < n)$

$$A = A^{(1)} = C_1^{-1} C_2^{-1} \ldots C_{n-1}^{-1} A^{(n)} = LU,$$

- The subdiagonal entries of L are the negative of the subdiagonal entries of the matrix $C_1 + C_2 + \ldots + C_{n-1}$.

Introduction to factorizations: SPD

- If A is a symmetric positive definite (SPD) matrix then setting $U = DL^T$, the LU factorization can be written as

$$A = LDL^T,$$

- This is the square root-free Cholesky factorization.
- Alternatively, it can be expressed as the Cholesky factorization

$$A = (LD^{1/2})(LD^{1/2})^T,$$

where the lower triangular matrix $LD^{1/2}$ has positive diagonal entries.

Introduction to factorizations: generic scheme

LU factorization can be stated in the generic form. l_{ik}: multipliers, $a_{kk}^{(k)}$: pivots. A is factorizable $\Rightarrow a_{kk}^{(k)} \neq 0$ for all k.

Algorithm (Generic LU factorization)

Input: *Factorizable matrix A.*
Output: *LU factorization $A = LU$.*

1: ***for*** ——— ***do***
2: ***for*** ——— ***do***
3: ***for*** ——— ***do***
4: $l_{ik} = a_{ik}^{(k)} / a_{kk}^{(k)}$
5: $a_{ij}^{(k+1)} = a_{ij}^{(k)} - l_{ik} a_{kj}^{(k)}$
6: ***end for***
7: ***end for***
8: ***end for***

Introduction to factorizations: generic scheme

- Three nested loops: six ways of assigning the indices to the loops.
- The performance differs based on sparsity, computer architecture.
- In exact arithmetic: the same L and U.
- To identify the variants, names that derive from the order in which the indices are assigned to the loops can be used.
  - kij and kji: submatrix LU factorizations,
  - jik and jki: column factorizations.
  - The remaining ones: row factorizations (column LU factorization applied to A^T.)

Submatrix LU

Each outermost step of the submatrix LU variants computes one row of U and one column of L. The first step ($k = 1$) can be described using a combination of matrix and vector notation as follows:

$$C_1 A = \begin{pmatrix} 1 & \\ -v/a_{11} & I \end{pmatrix} \begin{pmatrix} a_{11} & u^T \\ v & A_{2:n,2:n} \end{pmatrix} = \begin{pmatrix} a_{11} & u^T \\ & A_{2:n,2:n} - vu^T/a_{11} \end{pmatrix},$$

where

$$v = \left(a_{21}, \ldots, a_{n1}\right)^T, \quad \left(l_{21}, \ldots, l_{n1}\right)^T = v/a_{11}, \quad u^T = \left(a_{12}, \ldots, a_{1n}\right).$$

The $(n-1) \times (n-1)$ active submatrix

$$S = A_{2:n,2:n} - vu^T/a_{11}$$

is the Schur complement of A with respect to a_{11}. If A is factorizable then so too is S and the process can be repeated.

Submatrix LU

- The operations performed at each step k correspond to a sequence of rank-one updates.
- After $k-1$ steps ($1 < k \leq n$), the $(n-k+1) \times (n-k+1)$ Schur complement of A with respect to its $(k-1) \times (k-1)$ principal leading submatrix is the active submatrix of the partially eliminated matrix $A^{(k)}$ given by

$$
\begin{aligned}
S^{(k)} &= \begin{pmatrix} a_{kk} & \dots & a_{kn} \\ \vdots & \ddots & \vdots \\ a_{nk} & \dots & a_{nn} \end{pmatrix} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} u_{jk} & \dots & u_{jn} \end{pmatrix} \\
&= \begin{pmatrix} a_{kk}^{(k)} & \dots & a_{kn}^{(k)} \\ \vdots & \ddots & \vdots \\ a_{nk}^{(k)} & \dots & a_{nn}^{(k)} \end{pmatrix} = A_{k:n,k:n}^{(k)}. \qquad (14)
\end{aligned}
$$

- If A is SPD then the Cholesky and LDLT factorizations are termed right-looking (fan-out) factorizations.

Submatrix LU: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -1 & 4 & -2 & -1 & 3 \\ -1 & 0 & 4 & -1 & 5 \\ 6 & 7 & 8 & 10 & 2 \\ 8 & -1 & -1 & 4 & 7 \end{pmatrix}$$

Submatrix LU: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.25 & 2.75 & -3.75 & 6.5 \\ 1.5 & 8.5 & 15.5 & 26.5 & -7 \\ 2 & 1 & 9 & 26 & -5 \end{pmatrix}$$

Submatrix LU: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.07 & 2.53 & -4 & 6.8 \\ 1.5 & 2.27 & 22.87 & 35 & -17.2 \\ 2 & 0.27 & 9.87 & 27 & -6.2 \end{pmatrix}$$

Submatrix LU: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.07 & 2.53 & -4 & 6.8 \\ 1.5 & 2.27 & 9.03 & 71.1 & -78.6 \\ 2 & 0.27 & 3.89 & 42.6 & -32.7 \end{pmatrix}$$

Submatrix LU: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -3.25 & -3.75 & 4.5 \\ -0.25 & -0.07 & 2.53 & -4 & 6.8 \\ 1.5 & 2.27 & 9.03 & 71.1 & -78.6 \\ 2 & 0.27 & 3.89 & 0.6 & 14.4 \end{pmatrix}$$

Submatrix LU: kij algorithm

Algorithm

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
$u_{nn} = a_{nn}$

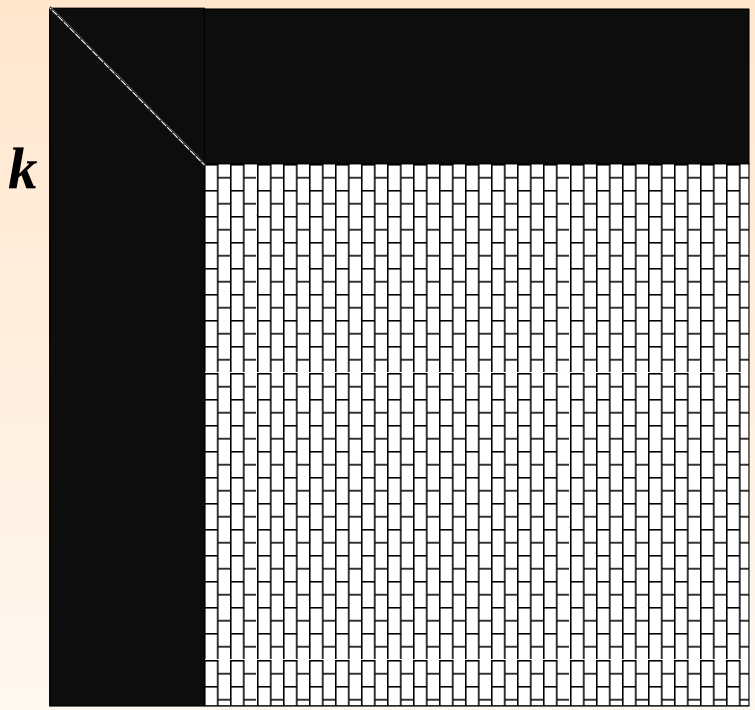
Submatrix LU: kji algorithm

Algorithm

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
$u_{nn} = a_{nn}$

Submatrix LU: depiction



Column LU

- In the column LU factorization, the outermost index is j.
- $l_{11} = 1$, the off-diagonal entries in column 1 of L are obtained by dividing the corresponding entries in column 1 of A by $u_{11} = a_{11}$.
- Assume $j-1$ columns ($1 < j \leq n$) of L and U have been computed. Then

$$\begin{pmatrix} L_{1:j-1,1:j-1} \\ L_{j:n,1:j-1} \end{pmatrix} U_{1:j-1,1:j-1} = \begin{pmatrix} A_{1:j-1,1:j-1} \\ A_{j:n,1:j-1} \end{pmatrix}.$$

Column j of U and then column j of L are computed as

$$U_{1:j-1,j} = L^{-1}_{1:j-1,1:j-1} A_{1:j-1,j}, \quad u_{jj} = a_{jj} - L_{j,1:j-1} U_{1:j-1,j},$$

$$l_{jj} = 1,\ L_{j+1:n,j} = (A_{j+1:n,j} - L_{j+1:n,1:j-1} U_{1:j-1,j})/u_{jj}.$$

- The strictly upper triangular part of $U_{:j}$ is determined from

$$L_{1:j-1,1:j-1} U_{1:j-1,j} = A_{1:j-1,j},$$

- The strictly lower triangular part of column j of L computed as a linear combination of column $A_{j+1:n,j}$ of A and previously computed columns of L.

Column LU: A symmetric

Observation

d_{jj} $(1 \leq j \leq n)$ of the LDLT factorization of the symmetric A is

$$d_{jj} = u_{jj} = a_{jj} - \sum_{k=1}^{j-1} d_{kk} l_{jk}^2.$$

The L factor is the same as is computed by the column LU factorization and

$$d_{jj} L_{j+1:n,j} = A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k}\, d_{kk}\, l_{jk}.$$

The U factor is equal to DL^T. Computing L and D in this way is called the left-looking (fan-in) factorization.

Algorithm (Column LU factorization with interchanging rows (partial pivoting))

Input: *Nonsingular nonsymmetric matrix* A.
Output: *LU factorization* $PA = LU$, *where* P *is a row permutation matrix.*

Interchange rows of A *so that* $|a_{11}| = \max\{|a_{i1}| \,|\, 1 \le i \le n\}$
$l_{11} = 1,\ u_{11} = a_{11},\ L_{2:n,1} = A_{2:n,1}/a_{11}$
for $j = 2 : n$ ***do***
    Solve $L_{1:j-1,1:j-1}U_{1:j-1,j} = A_{1:j-1,j}$
    $z_{1:n-j+1} = A_{j:n,j} - L_{j:n,1:j-1}U_{1:j-1,j}$
    Apply row interchanges to z, A *and* L *so that* $|z_1| = \max\{|z_i| \,|\, 1 \le i \le n - j + 1\}$.
    $l_{jj} = 1, u_{jj} = z_1$ *and* $L_{j+1:n,j} = z_{2:n-j+1}/z_1$
end for

Column algorithm: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -1 & 4 & -2 & -1 & 3 \\ -1 & 0 & 4 & -1 & 5 \\ 6 & 7 & 8 & 10 & 2 \\ 8 & -1 & -1 & 4 & 7 \end{pmatrix}$$

Column algorithm: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 4 & -2 & -1 & 3 \\ -0.25 & 0 & 4 & -1 & 5 \\ 1.5 & 7 & 8 & 10 & 2 \\ 2 & -1 & -1 & 4 & 7 \end{pmatrix}$$

Column algorithm: example

$$A = \begin{pmatrix} 4 & -1 & -5 & -11 & 6 \\ -0.25 & 3.75 & -2 & -1 & 3 \\ -0.25 & -0.07 & 4 & -1 & 5 \\ 1.5 & 2.27 & 8 & 10 & 2 \\ 2 & 0.27 & -1 & 4 & 7 \end{pmatrix}$$

Column algorithm: example

- Two completely different phases of the column construction.
  - applying L_{j-1}^{-1} (as a forward substitution)
  - computing subdiagonal part of L as a linear combination of a column of A with previously computed columns of L
- They are separated in the Cholesky factorization
- Row algorithm is column algorithm for A^T

Column algorithm: jki

Algorithm

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

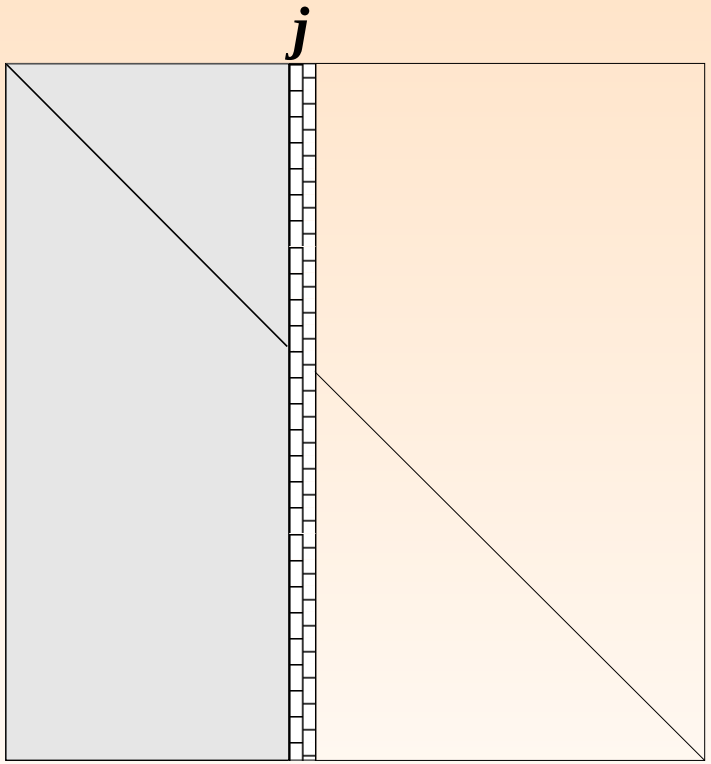
Column algorithm: jik

Algorithm

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

Column algorithm: depiction



Row algorithm: ikj

Algorithm

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

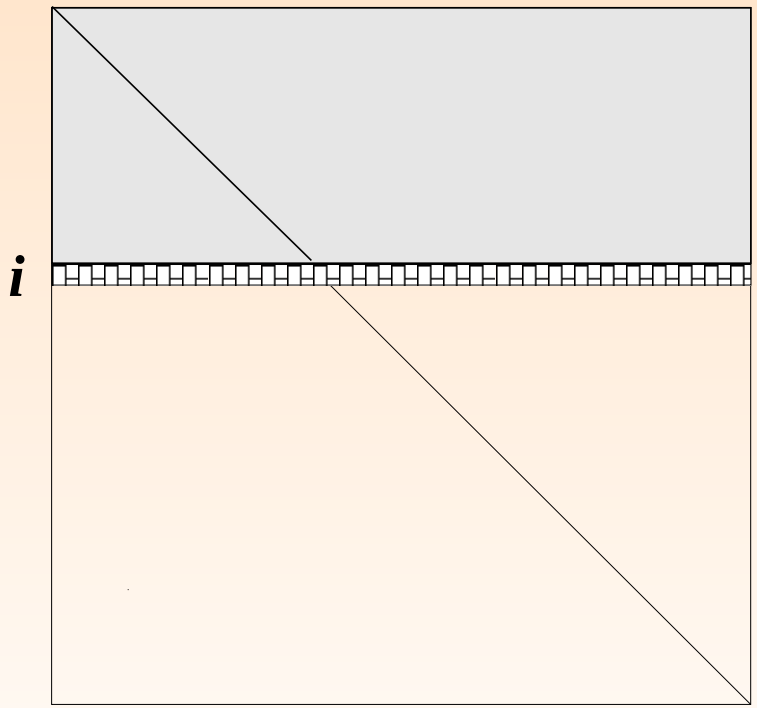
Row algorithm: ijk

Algorithm

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

Row algorithm: depiction



Outside the generic scheme

- An alternative is factorization by bordering.
- Set all diagonal entries of L to 1 and assume the first $k-1$ rows of L and first $k-1$ columns of U ($1 < k \leq n$) have been computed (that is, $L_{1:k-1,1:k-1}$ and $U_{1:k-1,1:k-1}$). At step k, $A_{1:k,1:k}$ satisfies

$$\begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix}.$$

- That is, the lower triangular part of row k of L and the upper triangular part of column k of U are obtained by solving

$$\begin{aligned} L_{k,1:k-1}U_{1:k-1,1:k-1} &= A_{k,1:k-1}, \\ L_{1:k-1,1:k-1}U_{1:k-1,k} &= A_{1:k-1,k}. \end{aligned}$$

The diagonal entry u_{kk} is then given by

$$u_{kk} = a_{kk} - L_{k,1:k-1}U_{1:k-1,k} \quad (\text{with } u_{11} = a_{11}).$$

A not factorizable

- What if A is not factorizable?
- Then there exists a row permutation matrix P such that PA is factorizable.
- Consider the simple 2×2 matrix A and its LU factorization

$$A = \begin{pmatrix} \delta & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \delta^{-1} & 1 \end{pmatrix} \begin{pmatrix} \delta & 1 \\ & 1 - \delta^{-1} \end{pmatrix}.$$

  If $\delta = 0$ this factorization does not exist and if δ is very small then the entries in the factors involving δ^{-1} are very large.
- Interchanging the rows of A we have

$$PA = \begin{pmatrix} 1 & 1 \\ \delta & 1 \end{pmatrix} = \begin{pmatrix} 1 & \\ \delta & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ & 1 - \delta \end{pmatrix},$$

  which is valid for all $\delta \neq 1$.

Lemma

Consider one step of the submatrix factorization of an SPD A. Schur complement of A with respect to (positive) $a_{1,1}$ is positive definite.

Proof.

For $\begin{pmatrix}\alpha & z^T\end{pmatrix}^T$ we have $x^TAx =$

$$\begin{pmatrix}\alpha & z^T\end{pmatrix}\begin{pmatrix}a_{1,1} & a_{1,2:n}\\ a_{2:n,1} & A_{2:n,2:n}\end{pmatrix}\begin{pmatrix}\alpha\\ z\end{pmatrix} = \tag{15}$$

$$\alpha^2 a_{1,1} + \alpha a_{1,2:n}z + \alpha z^T a_{2:n,1} + z^T A_{2:n,2:n}z = \tag{16}$$

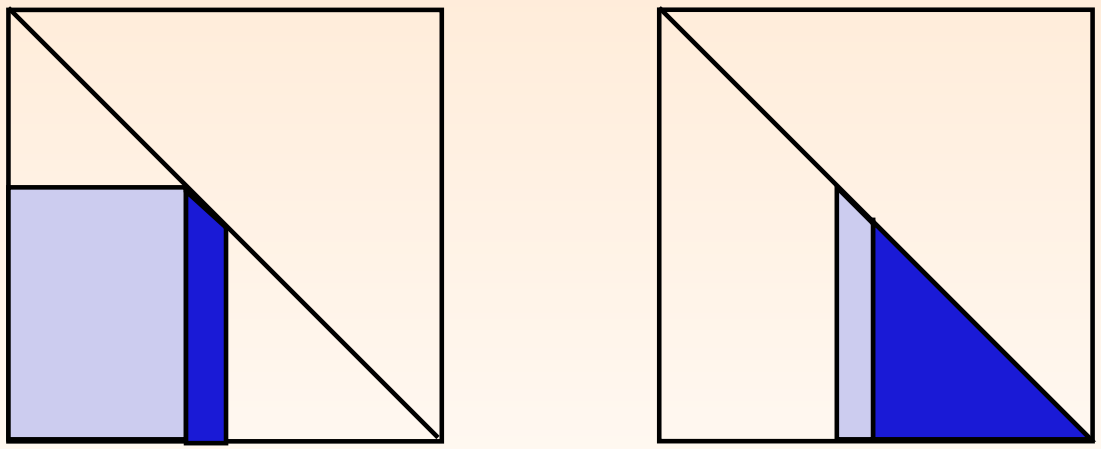
$$(\alpha + a_{1,1}^{-1}a_{1,2:n}z)^T a_{1,1}(\alpha + a_{1,1}^{-1}a_{1,2:n}z) + z^T(A_{2:n,2:n} - a_{2:n,1}a_{1,1}^{-1}a_{1,2:n})z \tag{17}$$

Choosing $z \neq 0$ and setting $\alpha = -a_{1,1}^{-1}a_{1,2:n}z$ we get

$$x^TAx = z^TSz \text{ where } S = A_{2:n,2:n} - a_{2:n,1}a_{1,1}^{-1}a_{1,2:n}.$$

Schemes for Cholesky

- Left-looking schemes (second part of the column LU)
- Right-looking schemes (just the submatrix scheme)
- The row scheme corresponds to the first part of the Cholesky algorithm.



Submatrix (right-looking) Cholesky

$$
\begin{aligned}
A &= \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3:n} \\ a_{2,1} & a_{2,2} & a_{2,3:n} \\ a_{3:n,1} & a_{3:n,2} & A_{3:n,3:n} \end{pmatrix} \\
&= \begin{pmatrix} \sqrt{a_{1,1}} & 0 & \\ \frac{a_{2,1}}{\sqrt{a_{1,1}}} & \sqrt{a^{(1)}_{2,2}} & \\ \frac{a_{3:n,1}}{\sqrt{a_{1,1}}} & \frac{a^{(1)}_{3:n,2}}{\sqrt{a^{(1)}_{2,2}}} & I_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & A^{(2)}_{3:n,3:n} - \frac{a_{3:n,1}a_{1,3:n}}{a_{1,1}} - \frac{a^{(1)}_{3:n,2}a^{(1)}_{2,3:n}}{a^{(1)}_{2,2}} \end{pmatrix} \\
&\quad \begin{pmatrix} \sqrt{a_{1,1}} & \frac{a_{2,1}}{\sqrt{a_{1,1}}} & \frac{a_{1,3:n}}{\sqrt{a_{1,1}}} \\ 0 & \sqrt{a^{(1)}_{2,2}} & \frac{a^{(1)}_{2,3:n}}{\sqrt{a^{(1)}_{2,2}}} \\ & & I_{n-2} \end{pmatrix} \\
&= \begin{pmatrix} l_{1,1} & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 \\ l_{3:n,1} & l_{3:n,2} & I_{n-2} \end{pmatrix} \begin{pmatrix} l_{1,1} & l_{2,1} & l_{1,3:n} \\ 0 & l_{2,2} & l_{2,3:n} \\ 0 & 0 & I_{n-2} \end{pmatrix}
\end{aligned}
$$

Column (left-looking) Cholesky

Algorithm

Column Cholesky factorization: $A \rightarrow$ *square-root factor* $L = (l_{ij})$

1. *for* $j = 1 : n$ *do*
2. *Compute an auxiliary vector* $t_{j:n}$

$$\begin{pmatrix} t_j \\ \vdots \\ t_n \end{pmatrix} = \begin{pmatrix} a_{jj} \\ \vdots \\ a_{nj} \end{pmatrix} - \sum_{\{k|l_{jk}\neq 0\}} l_{jk} \begin{pmatrix} l_{jk} \\ \vdots \\ l_{nk} \end{pmatrix} \tag{18}$$

3. *Get a column of* L *by scaling* $t_{j:n}$

$$\begin{pmatrix} l_{jj} \\ \vdots \\ l_{nj} \end{pmatrix} = \frac{1}{\sqrt{t_j}} \begin{pmatrix} t_j \\ \vdots \\ t_n \end{pmatrix} \tag{19}$$

4. *end* j

Row (up-looking) Cholesky

Algorithm

Row Cholesky factorization: : $A \rightarrow$ *square-root factor* $L = (l_{ij})$. 1. *for* $i = 1 : n$ *do*

2. *Solve the triangular system*

$$L_{1:i-1,1:i-1} \begin{pmatrix} l_{i1} \\ \vdots \\ l_{i,i-1} \end{pmatrix} = \begin{pmatrix} a_{i1} \\ \vdots \\ a_{i,i-1} \end{pmatrix} \tag{20}$$

3. *Compute the diagonal entry* $l_{ii} = \sqrt{\left(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2\right)}$

4. *end* i

Factorizations

Elimination versus decomposition (factorization)

- 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
- Six algorithms of the generic scheme for Cholesky as for LU, but there are also others (bordering, Dongarra-Eisenstat)

Sparse factorizations

Remind the fill-in

- Arrow matrix - original matrices: example showing how bad the fill-in problem can be

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

Sparse factorizations

Remind the fill-in

- Arrow matrix - structure after elimination

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

Sparse factorizations

Remind the fill-in

- Arrow matrix - structure after elimination

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

- Fill-in description and ways to avoid it must capture it dynamically!

Simple fill-in results

- Assume $\mathcal{S}\{A\}$ is symmetric and consider the elimination graph $\mathcal{G}^k$ at step k.
- Its vertices are the $n-k+1$ uneliminated vertices. Its edge set contains the edges in $\mathcal{G}(A)$ connecting these vertices and additional edges corresponding to filled entries produced during the first $k-1$ elimination steps.
- The sequence of graphs $\mathcal{G}^1 \equiv \mathcal{G}(A), \mathcal{G}^2, \ldots$ is generated recursively using Parter's rule:

  To obtain the elimination graph $\mathcal{G}^{k+1}$ from $\mathcal{G}^k$, delete vertex k and add all possible edges between vertices that are adjacent to vertex k in $\mathcal{G}^k$.

Factorizations

- Denoting $\mathcal{G}^k = (\mathcal{V}^k, \mathcal{E}^k)$, $\mathcal{G}^{k+1} = (\mathcal{V}^{k+1}, \mathcal{E}^{k+1})$, the Parter's rule:
  $\mathcal{V}^{k+1} = \mathcal{V}^k \setminus \{k\}$, $\mathcal{E}^{k+1} = \mathcal{E}^k \cup \{(i,j) \,|\, i,j \in adj_{\mathcal{G}^k}\{k\}\} \setminus \{i \,|\, i \in adj_{\mathcal{G}^k}\{k\}\}$.
- If $\mathcal{S}\{A\}$ is symmetric then Parter's rule says that the adjacency set of vertex k becomes a clique when k is eliminated: Gaussian elimination systematically generates cliques.
- As the elimination process progresses, cliques grow or more than one clique joins to form larger cliques: clique amalgamation.
- A clique with m vertices has $m(m-1)/2$ edges. It can be represented by storing a list of its vertices, without any reference to edges.
- If $\mathcal{S}\{A\}$ is nonsymmetric then the elimination graphs are digraphs and Parter's rule generalizes as follows:
- *To obtain the elimination graph $\mathcal{G}^{k+1}$ from $\mathcal{G}^k$, delete vertex k and add all edges $(i \xrightarrow{\mathcal{G}^{k+1}} j)$ such that $(i \xrightarrow{\mathcal{G}^k} k)$ and $(k \xrightarrow{\mathcal{G}^k} j)$.*

Factorizations

- Simple example (symmetric):

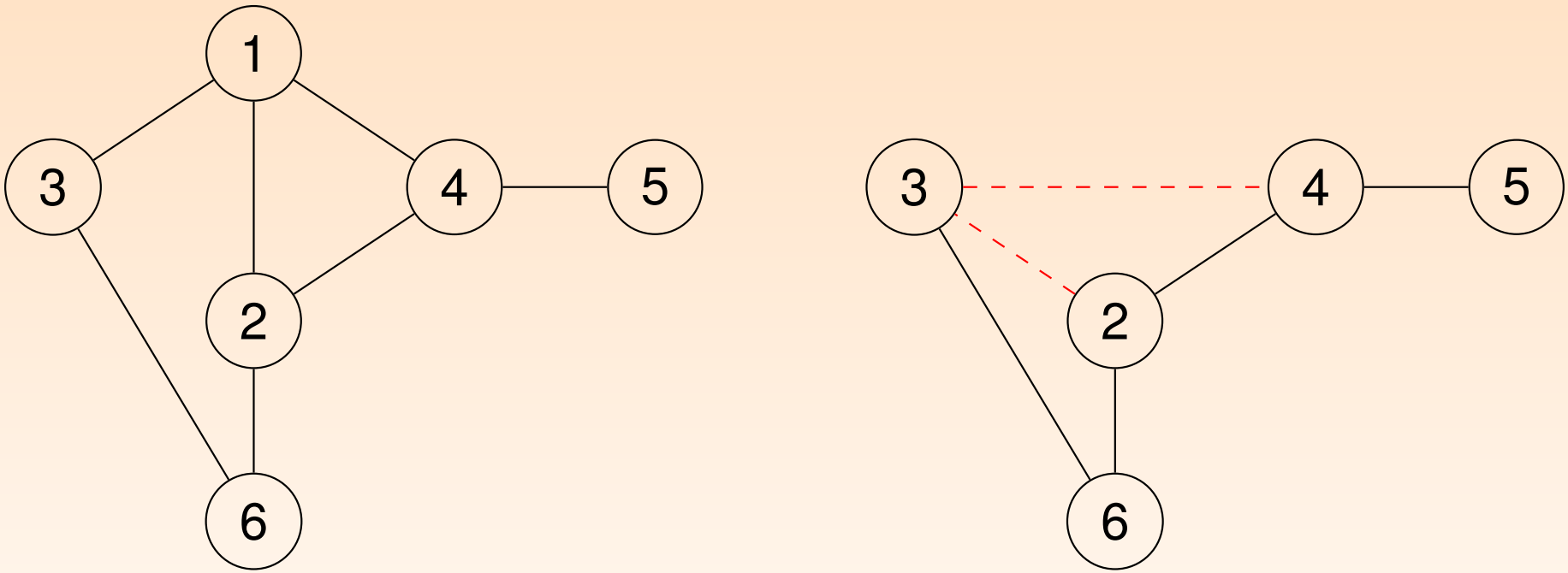


Figure: *Illustration of Parter's rule. The original undirected graph $\mathcal{G} = \mathcal{G}^1$ and the elimination graph $\mathcal{G}^2$ that results from eliminating vertex 1 are shown on the left and right, respectively. The red dashed lines denote fill edges. The vertices $\{2, 3, 4\}$ become a clique.*

Factorizations

- Simple example (nonsymmetric):

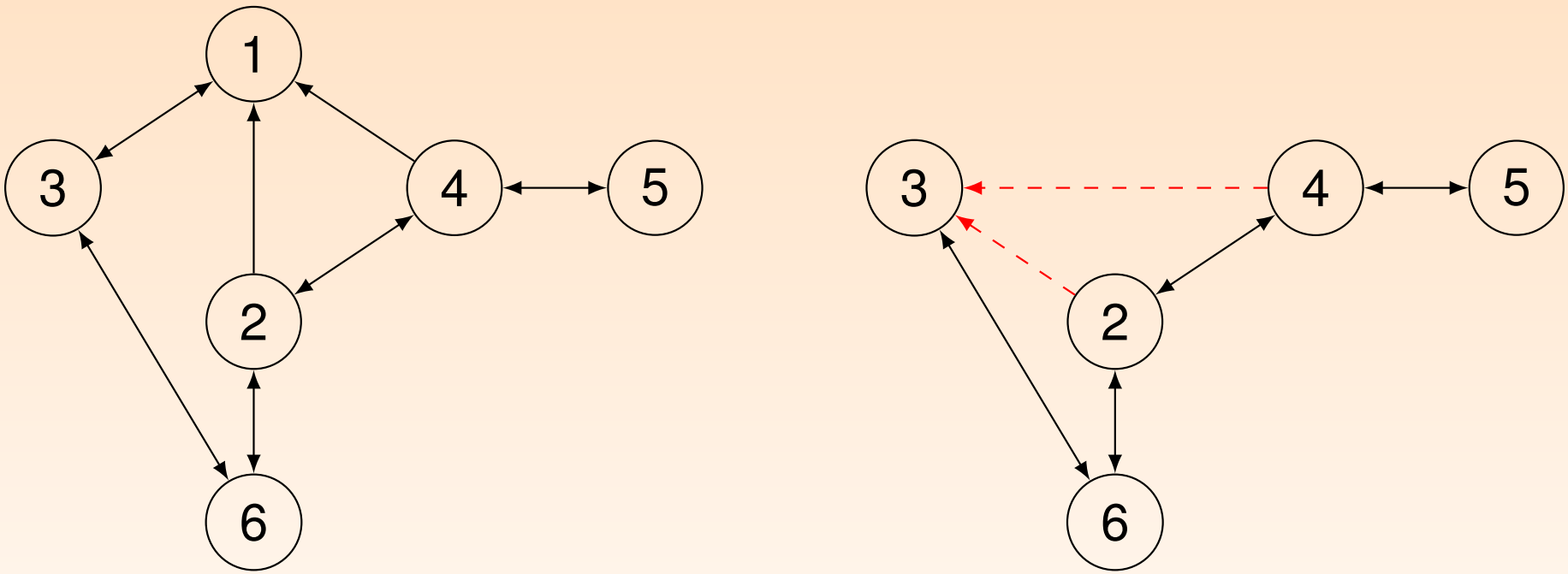


Figure: *Illustration of Parter's rule for a nonsymmetric $\mathcal{S}\{A\}$. The original digraph $\mathcal{G} = \mathcal{G}^1$ and the directed elimination graph $\mathcal{G}^2$ that results from eliminating vertex 1 are shown on the left and right, respectively. The red dashed lines denote fill edges.*

Parter's rule for factors

- The repeated application of Parter's rule specifies all the edges in $\mathcal{G}(L + L^T)$:
  (i, j) *is an edge of* $\mathcal{G}(L + L^T)$ *if and only if* (i, j) *is an edge of* $\mathcal{G}(A)$ *or* (i, k) *and* (k, j) *are edges of* $\mathcal{G}(L + L^T)$ *for some* $k < i, j$.
- This generalizes to a nonsymmetric matrix A and its LU factorization:
  $(i \to j)$ *is an edge of the digraph* $\mathcal{G}(L + U)$ *if and only if* $(i \to j)$ *is an edge of the digraph* $\mathcal{G}(A)$ *or* $(i \to k)$ *and* $(k \to j)$ *are edges of* $\mathcal{G}(L + U)$ *for some* $k < i, j$.

Factorizations

- Parter's rule is a local rule that uses the dependency on nonzeros obtained in previous steps of the factorization. The following result fully characterizes the nonzero entries in the factors using only paths in $\mathcal{G}(A)$.

Theorem

(a) Let $\mathcal{S}\{A\}$ be symmetric and $A = LL^T$. Then

$(L + L^T)_{ij} \neq 0$ if and only if there is a fill-path $i \overset{\mathcal{G}(A)}{\underset{min}{\Longleftrightarrow}} j$.

(b) Let $\mathcal{S}\{A\}$ be nonsymmetric and $A = LU$. Then

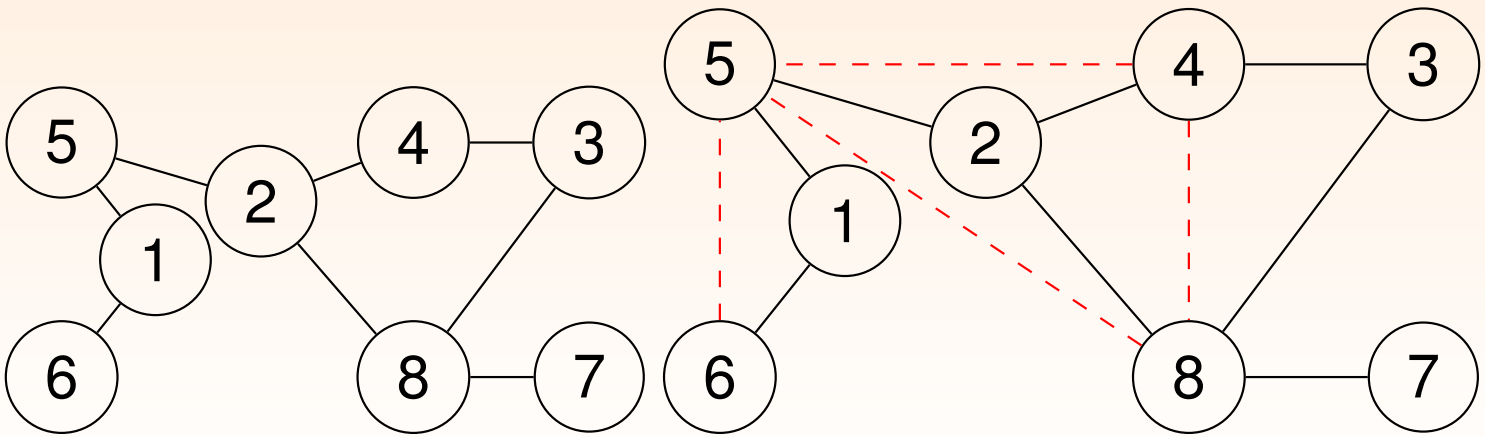
$(L + U)_{ij} \neq 0$ if and only if there is a fill-path $i \overset{\mathcal{G}(A)}{\underset{min}{\Longrightarrow}} j$.

The fill-paths may not be unique.

Factorizations

Symmetric $\mathcal{S}\{A\}$: a filled entry in position $(8,6)$ of L because of a fill-path $8 \overset{\mathcal{G}(A)}{\underset{min}{\Longleftrightarrow}} 6$: $8 \longleftrightarrow 2 \longleftrightarrow 5 \longleftrightarrow 1 \longleftrightarrow 6$.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} * & & & & * & * & & \\ & * & & * & * & & & * \\ & & * & * & & & & * \\ & * & * & * & & & & \\ * & * & & & * & & & \\ * & & & & & * & & \\ & & & & & & * & * \\ & * & * & & & & * & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} * & & & & * & * & & \\ & * & & * & * & & & * \\ & & * & * & & & & * \\ & * & * & * & f & & & f \\ * & * & & f & * & f & & f \\ * & & & & f & * & & f \\ & & & & & & * & * \\ & * & * & f & f & f & * & * \end{pmatrix} \end{array}$$



Corollary

(Edges of $\mathcal{G}^k$ in terms of reachable sets) Assume $\mathcal{S}\{A\}$ is symmetric. Let $\mathcal{V}^k$ be the set of $k-1$ vertices of $\mathcal{G}(A)$ that have already been eliminated and let v be a vertex in the elimination graph $\mathcal{G}^k$. Then the set of vertices adjacent to v in $\mathcal{G}^k$ is the set $\mathcal{R}each(v, \mathcal{V}^k)$ of vertices reachable from v through $\mathcal{V}^k$ in $\mathcal{G}(A)$.

Proof.

The proof is by induction on k. The result holds trivially for $k = 1$ because the $\mathcal{R}each(v, \mathcal{V}^1) = adj_{\mathcal{G}(A)}\{v\}$. Assume the result holds for $\mathcal{G}^1, \ldots, \mathcal{G}^k$ with $k \geq 1$ and let v be a vertex in the graph $\mathcal{G}^{k+1}$ that is obtained after eliminating v_k from $\mathcal{G}^k$. If v is not adjacent to v_k in $\mathcal{G}^k$ then $\mathcal{R}each(v, \mathcal{V}^{k+1}) = \mathcal{R}each(v, \mathcal{V}^k)$. Otherwise, if v is adjacent to v_k in $\mathcal{G}^k$ then $adj_{\mathcal{G}^{k+1}}\{v\} = \mathcal{R}each(v, \mathcal{V}^k) \cup \mathcal{R}each(v_k, \mathcal{V}^k)$. In both cases Parter's rule implies that the new adjacency set is exactly equal to the vertices that are reachable from v through $\mathcal{V}^{k+1}$, that is, $\mathcal{R}each(v, \mathcal{V}^{k+1})$. □ □

Factorizations

Figure depicts a graph $\mathcal{G}(A)$. The adjacency sets of the vertices in $\mathcal{G}^4$ that result from eliminating vertices $\mathcal{V}^4 = \{1, 2, 3\}$ are
$adj_{\mathcal{G}^4}\{4\} = \mathcal{R}each(4, \mathcal{V}^4) = \{5\}$, $adj_{\mathcal{G}^4}\{5\} = \mathcal{R}each(5, \mathcal{V}^4) = \{4, 6, 7\}$,
$adj_{\mathcal{G}^4}\{6\} = \mathcal{R}each(6, \mathcal{V}^4) = \{5, 7\}$, $adj_{\mathcal{G}^4}\{7\} = \mathcal{R}each(7, \mathcal{V}^4) = \{5, 6, 8\}$,
$adj_{\mathcal{G}^4}\{8\} = \mathcal{R}each(8, \mathcal{V}^4) = \{7\}$.

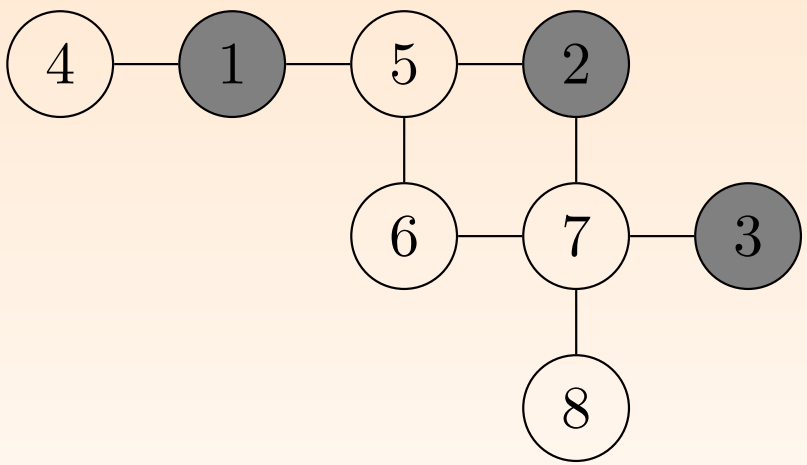


Figure: *An example to illustrate reachable sets in $\mathcal{G}(A)$. The grey vertices 1, 2, and 3 are eliminated in the first three elimination steps ($\mathcal{V}^4 = \{1, 2, 3\}$).*

Factorizations

So far, only implicit results

- Neither the local characterization of filled entries using Parter's rule nor Theorem on paths provide a direct answer as to whether a certain edge belongs to $\mathcal{G}(L + L^T)$ (or $\mathcal{G}(L + U)$); without performing the eliminations
- Results presented so far do not tell us whether a given entry of a factor of A is nonzero.
- Such questions are addressed by deeper theoretical and algorithmic results that are presented later.

From factorization to the solution

- Once an LU factorization has been computed, the solution x of the linear system $Ax = b$ is computed by solving the lower triangular system

$$Ly = b, \tag{21}$$

  followed by the upper triangular system

$$Ux = y. \tag{22}$$

- Triangular solves with a dense right-hand side vectorstraightforward.
- First forward substitution: the component y_1 is determined from the first equation. Substitution into the second equation to obtain y_2, and so on.
- Once y is available, the solution can be obtained by back substitution in which the last equation is used to obtain x_n, which is then substituted into equation $n-1$ to obtain x_{n-1}, and so on.

Algorithm (Forward substitution: simple lower triangular solve $Ly = b$ with b dense)

Input: *Lower triangular matrix L with nonzero diagonal entries and dense right-hand side b.*
Output: *The dense solution vector y.*

Initialise $y = b$
for $j = 1 : n$ ***do***
    $y_j = y_j / l_{jj}$
    for $i = j + 1 : n$ ***do***
        if $l_{ij} \neq 0$ ***then***
            $y_i = y_i - l_{ij} y_j$
        end if
    end for
end for

Sparse right-hand side

- When b is sparse, the solution y is also sparse. In particular, if in Algorithm $y_k = 0$, then the outer loop with $j = k$ can be skipped.
- Furthermore, if $b_1 = b_2 = \ldots = b_k = 0$ and $b_{k+1} \neq 0$, then $y_1 = y_2 = \ldots = y_k = 0$. Scanning y to check for zeros adds $O(n)$ to the complexity.
- But if the set of indices $\mathcal{J} = \{j \mid y_j \neq 0\}$ is known beforehand then we can use the following Algorithm,
- A possible way to determine $\mathcal{J}$ is discussed later.

Algorithm (Forward substitution: lower triangular solve $Ly = b$ with b sparse)

Input: *Lower triangular matrix L with nonzero diagonal entries, sparse vector b and the set $\mathcal{J}$.*
Output: *The sparse solution vector y.*

1: *Initialise* $y = b$
2: ***for*** $j \in \mathcal{J}$ ***do*** ▷ *Take indices from $\mathcal{J}$ in increasing order*
3: $y_j = y_j / l_{jj}$
4: ***for*** $i = j + 1 : n$ ***do***
5: ***if*** $l_{ij} \neq 0$ ***then***
6: $y_i = y_i - l_{ij} y_j$
7: ***end if***
8: ***end for***
9: ***end for***

Outline

Reducibility and blocks

Enhancements due to permuting into a block form

- Permuting to block form is closely connected to matrix reducibility.
- A is said to be reducible if there is a permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{p_1,p_1} & A_{p_1,p_2} \\ 0 & A_{p_2,p_2} \end{pmatrix},$$

  where A_{p_1,p_1} and A_{p_2,p_2} are non trivial square matrices (that is, they are of order at least 1).
- If A is not reducible, it is irreducible. Matrices of order 1 are irreducible.
- If $\mathcal{S}\{A\}$ is symmetric then $A_{p_1,p_2} = 0$ and PAP^T is block diagonal.
- A one-sided permutation can transform an irreducible matrix A into a reducible matrix AQ.

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & \\ 1 & & \end{pmatrix}, \quad Q = \begin{pmatrix} & & 1 \\ & 1 & \\ 1 & & \end{pmatrix}, \quad AQ = \begin{pmatrix} 1 & 1 & 1 \\ & 1 & 1 \\ & & 1 \end{pmatrix}.$$

Hall properties

- A matrix A is said to be a Hall matrix (or to have the Hall property) if every set of k columns has nonzeros in at least k rows ($1 \leq k \leq n$).
- A is a strong Hall matrix (or to have the strong Hall property) if every set of k columns ($1 \leq k < n$) has nonzeros in at least $k+1$ rows.
- If A is square then A has the strong Hall property if and only if the directed graph $\mathcal{G}(A)$ is strongly connected.

Reducibility and blocks

Theorem

Given a nonsingular nonsymmetric matrix A there exists a permutation matrix P such that

$$PAP^T = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,nb} \\ 0 & A_{2,2} & \cdots & A_{2,nb} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{nb,nb} \end{pmatrix}, \tag{23}$$

where the square matrices $A_{ib,ib}$ on the diagonal are irreducible. The set $\{A_{ib,ib} \,|\, 1 \leq ib \leq nb\}$ is uniquely determined (but the blocks may appear on the diagonal in a different order). The order of the rows and columns within each $A_{ib,ib}$ may not be unique.

- This upper block triangular form (BTF) is also known as the Frobenius normal form. It is said to be non trivial if $nb > 1$, and this is the case if A does not have the strong Hall property.

Permutation to BTF

An example of a matrix that can be symmetrically permuted to block triangular form with $nb = 2$ is given in Figure 16.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} * & * & & * & * & \\ & * & & & * & \\ & * & * & & * & * \\ & * & & * & & \\ * & & & & * & \\ & * & * & & & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 6 \\ 3 \\ 5 \\ 4 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{cccccc} 6 & 3 & 5 & 4 & 1 & 2 \end{array} \\ \begin{pmatrix} * & * & & & & * \\ * & * & * & & & * \\ & & * & & * & \\ & & & * & & * \\ & & * & * & * & * \\ & & * & & & * \end{pmatrix} \end{array}$$

Figure: *The sparsity patterns of A (left) and the upper block triangular form PAP^T with two blocks $A_{ib,ib}$, $i = 1, 2$, of order 2 and 4 (right).*

Using BTF

Algorithm (Solve a sparse linear system in upper BTF)

Input: *Upper block triangular matrix (23) and a conformally partitioned right-hand side vector* c.
Output: *The conformally partitioned solution vector* y.

1: **for** $ib = 1 : nb$ **do**
2: Compute $P_{ib} A_{ib,ib} = L_{ib} U_{ib}$
3: **end for**
4: Solve $L_{nb} U_{nb}\, y_{nb} = P_{nb} c_{nb}$ ▷ *Perform forward and back substitutions*
5: **for** $ib = nb - 1 : 1$ **do**
6: **for** $jb = ib + 1 : nb$ **do**
7: $c_{ib} = c_{ib} - A_{ib,jb} y_{jb}$ ▷ *Sparse matrix-vector operation*
8: **end for**
9: Solve $L_{ib} U_{ib}\, y_{ib} = P_{ib} c_{ib}$ ▷ *Perform substitutions*
10: **end for**

Reducibility and blocks

Transversal

- The transversal of a matrix A is the set of its nonzero diagonal elements.
- A has a full or maximum transversal if all its diagonal entries are nonzero. There exist permutation matrices P and Q such that PAQ has a full transversal if and only if A has the Hall property.
- Moreover, if A is nonsingular then it can be nonsymmetrically permuted to have a full transversal. However, the converse is clearly not true (for example, a matrix with all its entries equal to one has a full transversal but it is singular).
- If A has a full transversal then there exists a permutation P_s such that $P_s A P_s^T$ has the BTF form. In the other words, once A has a full transversal, a symmetric permutation is sufficient to obtain the BTF form.

Permutation to BTF

- Finding P_s is identical to finding the strongly connected components (SCCs) of the digraph $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$.
- To find the SCCs, $\mathcal{V}$ is partitioned into non-empty subsets $\mathcal{V}_i$ with each vertex belonging to exactly one subset. Each vertex i in the quotient graph corresponds to a subset $\mathcal{V}_i$ and there is an edge in the quotient graph with endpoints i and j if $\mathcal{E}$ contains at least one edge with one endpoint in $\mathcal{V}_i$ and the other in $\mathcal{V}_j$.
- The condensation (or component graph) of a digraph is a quotient graph in which the SCCs form the subsets of the partition, that is, each SCC is contracted to a single vertex. This reduction provides a simplified view of the connectivity between components.

Permutation to BTF

An example of SCCs. Here five SCCs: $\{p, q, r\}$, $\{s, t, u\}$, $\{v\}$, $\{w\}$, and $\{x\}$.

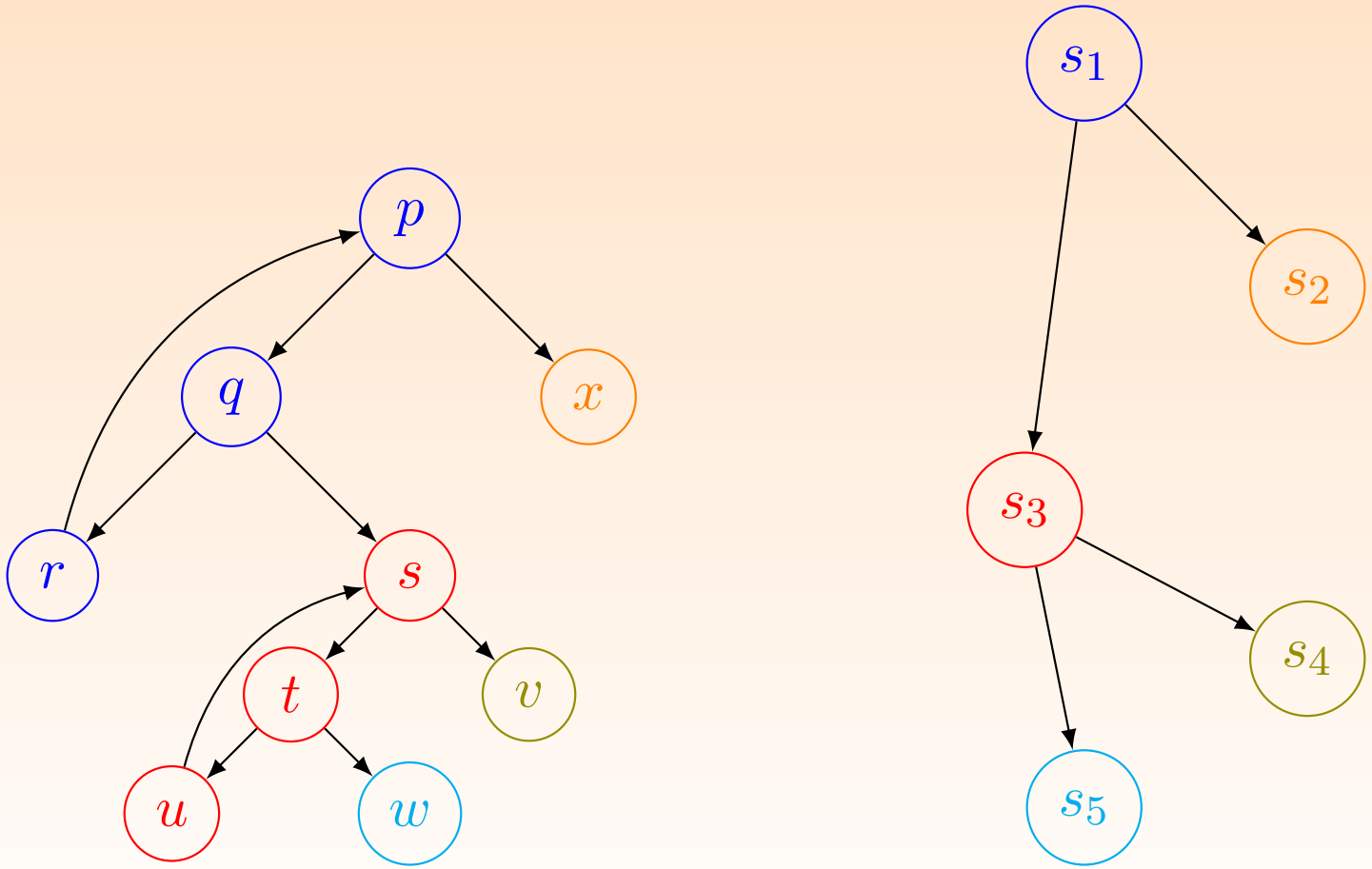


Figure: *An illustration of the strong components of a digraph. On the left, the five SCCs are denoted using different colours and on the right is the condensation DAG $\mathcal{G}_C$ formed by the*

Relationship between SCCs and DAGs

Theorem

The condensation $\mathcal{G}_C$ of a digraph is a DAG (directed acyclic graph).

- Any DAG can be topologically ordered.
- Consequently, $\mathcal{G}_C = (\mathcal{V}_C, \mathcal{E}_C)$ can be topologically ordered and if $\mathcal{V}_i$ and $\mathcal{V}_j$ are contracted to s_i and s_j and $(s_i \longrightarrow s_j) \in \mathcal{E}_C$ then $s_i < s_j$.
- It follows that to permute A to block triangular form it is sufficient to find the SCCs of $\mathcal{G}(A)$.
- That is, topologically ordering the vertices of the condensation $\mathcal{G}_C$ induced by the SCCs is the quotient graph that implies the block triangular form.

Finding SCCs

- Tarjan's algorithm: the key idea here is that vertices of a SCC form a subtree in the DFS spanning tree of the graph.
- The algorithm performs depth-first searches, keeping track of two properties for each vertex v: when v was first encountered (held in $invorder(v)$) and the lowest numbered vertex that is reachable from v (called the low-link value and held in $lowlink(v)$).
- It pushes vertices onto a stack as it goes and outputs a SCC when it finds a vertex for which $invorder(v)$ and $lowlink(v)$ are the same. The value $lowlink(v)$ is computed during the DFS from v, as this finds the vertices that are reachable from v.
- The outermost loop of the algorithm visits each vertex that has not yet been visited, ensuring vertices that are not reachable from the starting vertex are eventually visited.
- The complexity bound is $O(|\mathcal{V}| + |\mathcal{E}|)$.

Algorithm (Tarjan's algorithm to find the SCCs)

1: $\mathcal{V}_v = \emptyset, S = (), index = 0,$ ▷ *Each vertex is initially unvisited*
2: **for** *each* $v \in \mathcal{V}$ **do**
3: **if** $v \notin \mathcal{V}_v$ **then**
4: *scomp_step*(v)
5: **end if**
6: **end for**
7: **recursive function** (*scomp_step*(v))
8: $\mathcal{V}_v = \mathcal{V}_v \cup \{v\}$ ▷ *Add* v *to the set of visited vertices*
9: $index = index + 1$ ▷ *Set the index for* v *to smallest unused index*
10: $invorder(v) = index, lowlink(v) = index$
11: $push(S, v)$ ▷ *Put* v *on the stack*
12: *Set* $v = head(S)$ ▷ v *is the current head of* S.
13: **for** *each* $(v \to w) \in \mathcal{E}$ **do** ▷ *Look in the adjacency list of* v
14: **if** $w \notin \mathcal{V}_v$ **then** ▷ w *not yet been visited; recurse on it*
15: *scomp_step*(w)
16: $lowlink(v) = min(lowlink(v), lowlink(w))$
17: **else if** $w \in S$ **then** ▷ w *is in the stack and hence in current SCC*
18: $lowlink(v) = min(lowlink(v), invorder(w))$
19: **end if**
20: **end for**
21: **if** $lowlink(v) = invorder(v)$ **then**
22: *pop all vertices down to* v *from* S *to obtain a new SCC*
23: **end if**
24: **end recursive function**

Reducibility and blocks

Finding blocks: indistinguishability

- Finding sets of columns of A frequently have identical sparsity patterns. For example, for A from a finite element discretisation.

Definition

We say that two vertices u and v of an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ are indistinguishable if they have the same neighbours, that is, $adj_{\mathcal{G}}\{u\} \cup \{u\} = adj_{\mathcal{G}}\{v\} \cup \{v\}$.
A set of mutually indistinguishable vertices is called an indistinguishable vertex set. If $\mathcal{U} \subseteq \mathcal{V}$ is an indistinguishable vertex set then $\mathcal{U}$ is maximal if $\mathcal{U} \cup \{w\}$ is not indistinguishable for any $w \in \mathcal{V} \setminus \mathcal{U}$.

- Indistinguishability is an equivalence relation on $\mathcal{V}$
- Maximal indistinguishable vertex sets represent its classes $\rightarrow$ a partitioning of $\mathcal{V}$ into $nsup \geq 1$ non-empty disjoint subsets

$$\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2 \cup \ldots \cup \mathcal{V}_{nsup}. \tag{24}$$

- An indistinguishable vertex set can be represented by a single vertex, called a supervariable.

Finding blocks: indistinguishability

- $\mathcal{S}\{A\}$ is symmetric, $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$. Let $\mathcal{V}$ be partitioned into indistinguishable vertex sets and reorder the vertices such that those belonging to each subset $\mathcal{V}_1, \ldots, \mathcal{V}_{nsup}$ are numbered consecutively, with those in $\mathcal{V}_i$ preceding those in $\mathcal{V}_{i+1}$ $(1 \leq i < nsup)$.
- If P is the permutation matrix corresponding to this ordering then the permuted matrix PAP^T has a block structure in which the blocks are dense.

Finding supervariables

- Initially place all the vertices in a single vertex set (that is, into a single supervariable).
- Split into two supervariables by taking the first vertex $j = 1$ and moving those vertices that are in the adjacency set of j into a new vertex set (a new supervariable).
- Each vertex j is considered in turn and each vertex set $\mathcal{V}_{sv}$ that contains a vertex in $adj_{\mathcal{G}}\{j\} \cup j$ is split into two by moving the vertices in $adj_{\mathcal{G}}\{j\} \cup j$ that belong to $\mathcal{V}_{sv}$ into a new vertex set.
- As a result of the splitting and moving of vertices a vertex set can become empty, in which case it is discarded.
- Once the supervariables have been determined, the permuted matrix PAP^T can be condensed to a matrix of order equal to $nsup$.

Finding supervariables

Algorithm (Find the supervariables of an undirected graph)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix.*
Output: *Partitioning of $\mathcal{V}$ into indistinguishable vertex sets.*

1: $\mathcal{V}_1 = \{1, 2, \ldots, n\}$
2: ***for*** $j = 1 : n$ ***do***
3: ***for*** $i \in adj_{\mathcal{G}}\{j\} \cup j$ ***do***
4: *Find sv such that $i \in \mathcal{V}_{sv}$*
5: ***if*** *this is the first occurrence of sv for the current index j* ***then***
6: *Establish a new set $\mathcal{V}_{nsv}$ and move i from $\mathcal{V}_{sv}$ to $\mathcal{V}_{nsv}$*
7: ***else***
8: *Move i from $\mathcal{V}_{sv}$ to $\mathcal{V}_{nsv}$*
9: ***end if***
10: *Discard $\mathcal{V}_{sv}$ if it is empty*
11: ***end for***
12: ***end for***

Finding supervariables: illustration

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & * & & & * \\ * & * & & & * \\ & & * & * & * \\ & & * & * & * \\ * & * & * & * & * \end{pmatrix} \end{array}.$$

- Initially, $1, 2, 3, 4, 5$ are put into a single vertex set $\mathcal{V}_1$.
- Vertices $i = 1, 2$ and 5 belong to $adj_{\mathcal{G}}\{1\} \cup \{1\}$; they are moved from $\mathcal{V}_1$ into a new vertex set.
- There is no further splitting of the vertex sets for $j = 2$.
- $adj_{\mathcal{G}}\{3\} \cup \{3\} = \{3, 4, 5\}$. $i = 3$ and 4 are moved from $\mathcal{V}_1$ into a new vertex set. $\mathcal{V}_1$ is now empty and can be discarded. Vertex $i = 5$ is moved from the vertex set that holds vertices 1 and 2 into a new vertex set. For $j = 4$ and 5 no additional splitting is performed.
- Thus, three supervariables are found, namely $\{1, 2\}$, $\{3, 4\}$ and $\{5\}$.

Reducibility and blocks

Approach by Ashcraft

1. Number graph nodes/vertices (use numbers as their labels)
2. Compute vertex checksums:

$$chksum = \sum_{\{u,v\} \in E} w$$

3. Sort vertices by their checksums: in O(|E|+|V|log(|V|)) time
4. Different checksum means different block
5. First tie-breaking rule: if $chksum(u) = chksum(v)$: compare $|adj(u)|$ and $|adj(v)|$
6. Second tie-breaking rule: compare adjacency sets of u and v (the most time consuming)

Reducibility and blocks

Approximate indistinguishability: Saad

- Using symbolic dot products between the rows of the matrix.
- Here we assume that $\mathcal{S}\{A\}$ is symmetric but modifications exist.
- Rewrite A as row vectors

$$A = \left(a_1^T, \ldots, a_n^T\right)^T,$$

  and consider $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$.
- A partition $\mathcal{V} = \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_{nb}$ is constructed using row products $a_i^T a_k$ between different rows of A that express the level of orthogonality between the rows;
- if $a_i^T a_k$ is large then i and k are assigned to different vertex sets.

Reducibility and blocks

Approximate indistinguishability

- Algorithm treats all entries of A as unity (symbolically)
- The symbolic row products can be considered as a generalization of the angles between rows expressed by their cosines, hence
- On output, if $adjmap(i_1) = adjmap(i_2)$ then vertices i_1 and i_2 belong to the same vertex set.
- Symmetry of $\mathcal{S}\{A\}$ simplifies the computation of the symbolic row products because for row i only $k > i$ is considered, that is, only the symbolic row products that correspond to one triangle part of $A^T A$ are checked.
- The procedure outlined in Algorithm 6.4 is controlled by a threshold parameter $\tau \in (0, 1]$.

Reducibility and blocks

Algorithm

Find approximately indistinguishable vertex sets

1: $nb = 0$, $adjmap(1:n) = 0$, $cosine(1:n) = 0$
2: **for** $i = 1 : n$ **do**
3: **if** $adjmap(i) = 0$ **then**
4: $nb = nb + 1$ ▷ *Start a new set*
5: $adjmap(i) = ib$
6: **for** $(i, j) \in \mathcal{E}$ **do** ▷ *Corresponds to an entry in* $A_{i,1:n}$
7: **for** $(k, j) \in \mathcal{E}$ *with* $k > i$ **do** ▷ *Both rows* i *and* k *have an entry in column* j
8: **if** $adjmap(k) = 0$ **then** ▷ k *has not been yet added to some partitioning set*
9: $cosine(k) = cosine(k) + 1$ ▷ *Increase partial dot product*
10: **end if**
11: **end for**
12: **for** k *with* $cosine(k) \neq 0$ **do**
13: **if** $cosine(k)^2 \geq \tau^2 * nz_i * nz_k$ **then** ▷ *Test similarity of row patterns*
14: $adjmap(k) = nb$
15: **end if**
16: $cosine(k) = 0$
17: **end for**
18: **end for**
19: **end if**

Reducibility and blocks

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
* & & * & & * & \\
 & * & & & * & * \\
* & & * & & * & \\
 & & & * & & \\
* & * & * & & * & * \\
 & * & & & * & *
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 3 \\ 2 \\ 6 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 3 & 2 & 6 & 4 & 5 \end{array} \\
\begin{pmatrix}
* & * & & & & * \\
* & * & & & & * \\
 & & * & * & & * \\
 & & * & * & & * \\
 & & & & * & \\
* & * & * & * & & *
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 3 \\ 5 \\ 2 \\ 6 \\ 4 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 3 & 5 & 2 & 6 & 4 \end{array} \\
\begin{pmatrix}
* & * & * & & & \\
* & * & * & & & \\
* & * & * & * & * & \\
 & & * & * & * & \\
 & & * & * & * & \\
 & & & & & *
\end{pmatrix}
\end{array}
$$

Figure: *A example to illustrate Algorithm. The original matrix is given (left) together with the permuted matrix with indistinguishable vertex sets* $\mathcal{V} = \{1, 3\} \cup \{2, 6\} \cup \{4\} \cup \{5\}$ *obtained using* $\tau = 1$ *(centre) and the permuted matrix with approximately indistinguishable vertex sets* $\mathcal{V} = \{1, 3, 5\} \cup \{2, 6\} \cup \{4\}$ *obtained using* $\tau = 0.5$ *(right). The threshold* $\tau = 0.5$ *results in putting row* 5 *into the same set as row* 1*, making the vertex sets only approximately indistinguishable. The permuted matrix has an approximate block form.*

Outline

Symbolic Cholesky

- Symbolic Cholesky: only $\mathcal{S}\{A\}$ used to determine the nonzero structure of the Cholesky factor L without computing the numerical values of the nonzeros.
- Implicitly assumed that all the diagonal entries of A are included in $\mathcal{S}\{A\}$ (even if they are zero - this cannot happen if A is SPD).

- A fundamental difference between dense and sparse Cholesky factorizations is that, in the latter, each column of L depends on only a subset of the previous columns.
- Operations to update a computed column should be also sparse.
- We will see that the symbolic Cholesky can be clearly described using a tree structure.

Symbolic Cholesky

Column replication

- Let us start from the patterns of subsequent Schur complements.
- $$S^{(k)} = A_{k:n,k:n} - \sum_{j=1}^{k-1} \begin{pmatrix} l_{kj} \\ \vdots \\ l_{nj} \end{pmatrix} \begin{pmatrix} l_{kj} & \dots & l_{nj} \end{pmatrix}. \tag{25}$$
- Consider column j of L ($1 \leq j \leq k-1$) and let $l_{ij} \neq 0$ for some $i > j$.

Observation

For any $i > j \geq 1$ *such that* $l_{ij} \neq 0$

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\}. \tag{26}$$

This is called the column replication principle because the pattern of column j *of* L *(rows* i *to* n*) is replicated in the pattern of column* i *of* L*.*

Symbolic Cholesky

Column replication

- Denote the row index of the first subdiagonal nonzero entry in column j of L by $parent(j)$, that is,

$$parent(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}. \tag{27}$$

- If there is no such entry, set $parent(j) = 0$. The row index $parent(parent(j))$ is denoted by $parent^2(j)$, and so on.
- Applying column replication recursively implies the sparsity pattern of column j of L is replicated in that of column $parent(j)$, which in turn is replicated in the pattern of column $parent^2(j)$, and so on.

Column replication

- Example: Consider $j = 1$. Because the first subdiagonal entry in column 1 is in row 3, $parent(1) = 3$, $parent(3) = parent^2(1) = 5$.

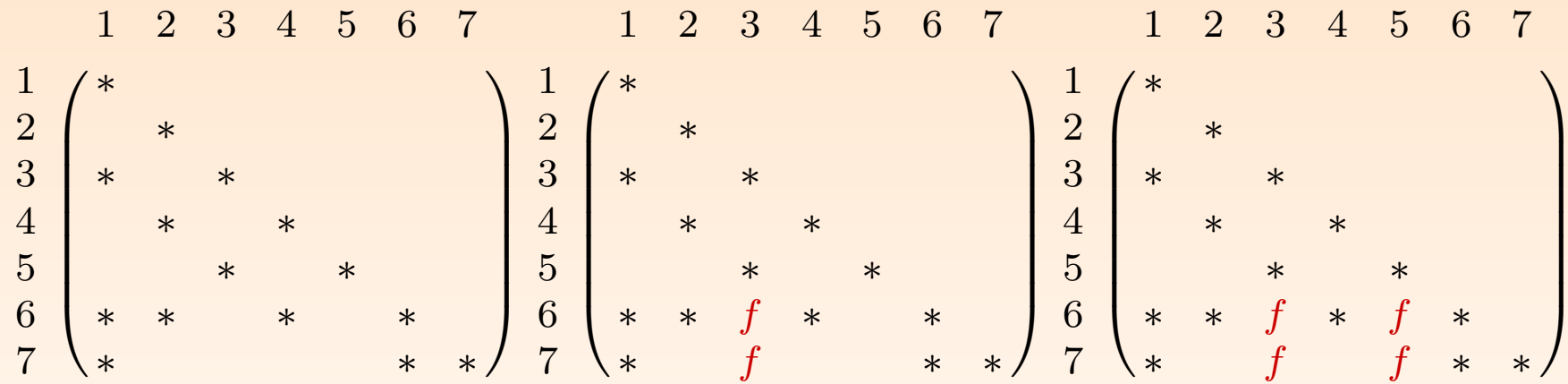
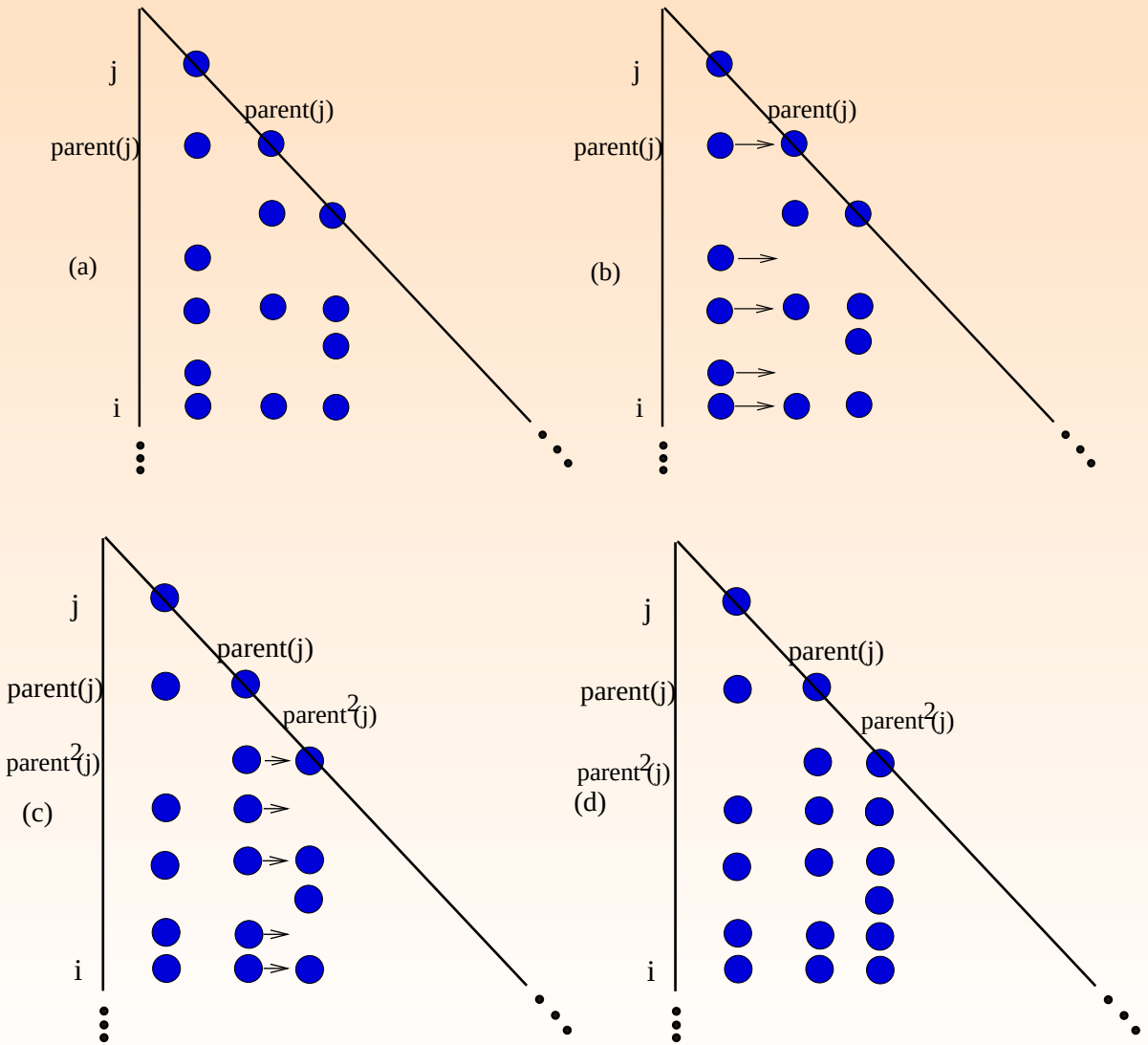


Figure: *An illustration of column replication. On the left are the entries in L before step 1 of a Cholesky factorization (that is, the entries in the lower triangular part of A); in the centre we show the replication of the nonzeros from column 1 in the pattern of column $parent(1) = 3$ (red entries f); on the right, we show the subsequent replication in column $parent^2(1) = 5$.*

Symbolic Cholesky

Fill-in as replication of column structures



Symbolic Cholesky

Existence of nonzeros in columns

The following result shows that, provided A is irreducible, the mapping $parent(j)$ has nonzero values given by (27) for all $j < n$.

Theorem

If A is SPD and irreducible then in each column j ($1 \leq j < n$) of its Cholesky factor L there exists an entry $l_{ij} \neq 0$ with $i > j$.

Proof.

From Parter's rule, each step of the Cholesky factorization corresponds to adding new edges into the graph of the corresponding Schur complement. If A is irreducible then the graphs corresponding to the Schur complements are connected. Consequently, for any vertex j ($1 \leq j < n$) in any of these graphs there is at least one vertex i with $i > j$ to which j is connected. This corresponds to the nonzero entry in column j of L.

□ □

Symbolic Cholesky

Row replication

With the convention $parent^1(j) = parent(j)$, the next theorem shows that if entry l_{ij} of L is nonzero then $parent^t(j) = i$ for some $t \geq 1$ and there is row replication in the sequence $j,\ parent^1(j),\ parent^2(j),\ \ldots, parent^t(j)$.

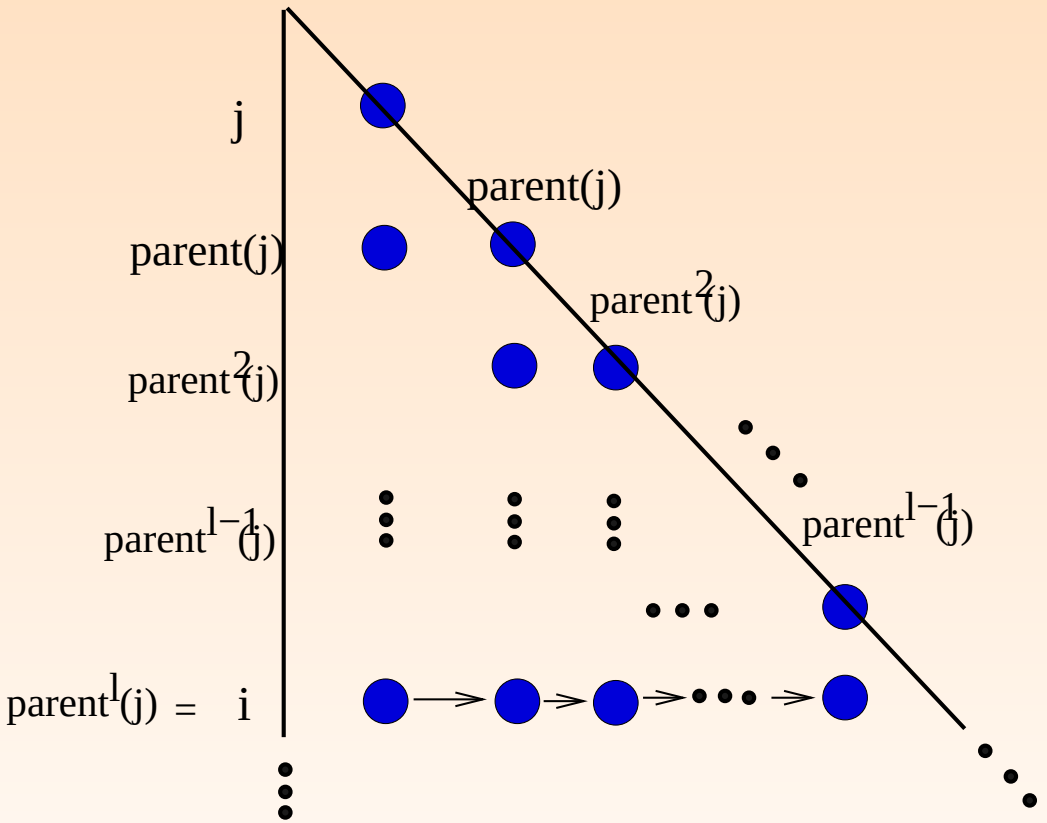
Theorem

Let A be SPD and let L be its Cholesky factor. If $l_{ij} \neq 0$ for some $j < i \leq n$ then there exists $t \geq 1$ such that $parent^t(j) = i$ and $l_{ik} \neq 0$ for $k = j, parent^1(j), parent^2(j),\ \ldots, parent^t(j)$.

Proof.

If $i = parent^1(j)$, the result is immediate. Otherwise, there exists an index k, $j < k < i$ of a subdiagonal entry in column j of L such that $k = parent^1(j)$. Column replication implies $l_{ik} \neq 0$. Applying an inductive argument to l_{ik}, the result follows after a finite number of steps. □ □

Row replication: example



- Replication of column structures $\rightarrow$ replication of nonzeros in a row.

Symbolic Cholesky

Necessary and sufficient condition (NSC) for a fill-in entry

- If there is sequence of nonzeros in a row of L, it is natural to ask where the sequence begins:
- If there is no $k \geq 1$ such that $a_{ik} \neq 0$, no replication of nonzeros can start in row i.

Theorem

Let A be SPD and let L be its Cholesky factor. If $a_{ij} = 0$ for some $1 \leq j < i \leq n$ then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ and $t \geq 1$ such that $a_{ik} \neq 0$ and $parent^t(k) = j$.

Elimination tree

- The discussion of column replication is significantly simplified using elimination trees.
- The elimination tree (or etree) $\mathcal{T}(A)$ (or simply $\mathcal{T}$) of a SPD matrix has vertices $1, 2, \ldots, n$ and an edge between each pair $(j, parent(j))$, where $parent(j)$ has been defined above;
- j is a root vertex of the tree if $parent(j) = 0$.
- The edges of $\mathcal{T}$ are considered to be directed from a child to its parent, that is,

$$\mathcal{E}(\mathcal{T}) = \{(j \longrightarrow i) \mid i = parent(j)\}.$$

  If $\mathcal{T}$ has a single component then the root vertex is n.
- Despite the terminology, the elimination tree need not be connected and in general is a forest. For simplicity, in our discussions, we assume $\mathcal{T}$ has a single component and we say that $\mathcal{T}$ is described by the vector $parent$.

Elimination tree

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\left(\begin{array}{cccccccc}
* & & & & * & * & & \\
 & * & & * & * & & & * \\
 & & * & * & & & & * \\
 & * & * & * & f & & & f \\
* & * & & f & * & f & & f \\
* & & & & f & * & & f \\
 & & & & & & * & * \\
 & * & * & f & f & f & * & *
\end{array}\right)
\end{array}
$$

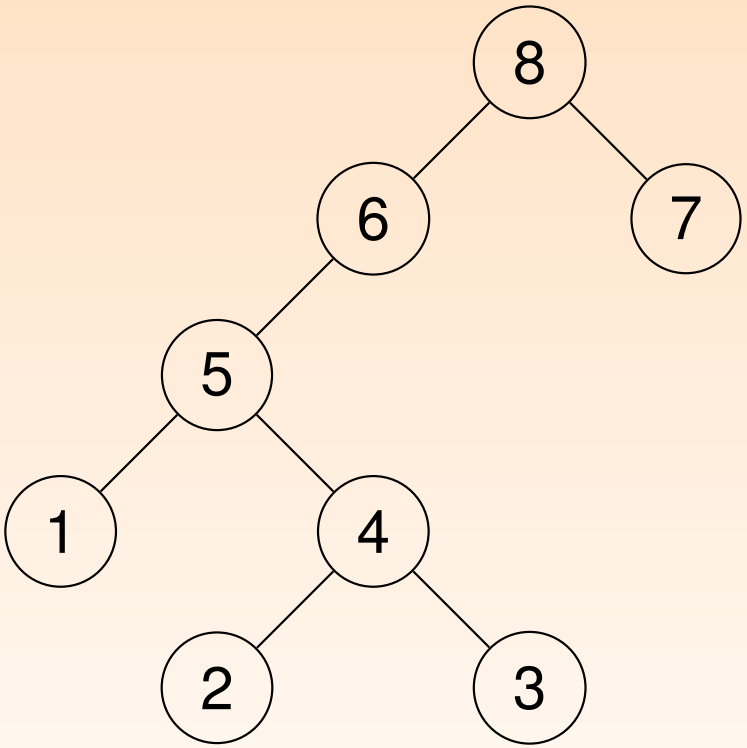


Figure: *An illustration of a sparse matrix A with a symmetric sparsity pattern and its elimination tree $\mathcal{T}(A)$. The root vertex is 8. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.*

Elimination tree: terminology

- Concepts such as child, leaf, ancestor and descendant vertices for directed rooted trees can be applied to $\mathcal{T}$.
- Additionally, $anc_{\mathcal{T}}\{j\}$ and $desc_{\mathcal{T}}\{j\}$ are defined to be the sets of ancestors and descendants of vertex j in $\mathcal{T}$.
- $\mathcal{T}(j)$: the subtree of $\mathcal{T}$ induced by j and $desc_{\mathcal{T}}\{j)$; j is the root vertex of $\mathcal{T}(j)$.
- The size $|\mathcal{T}(j)|$ is the number of vertices in the subtree.
- A pruned subtree of $\mathcal{T}(j)$ is the connected subgraph induced by j and a subset of $desc_{\mathcal{T}}\{j)$. That is, for any vertex i in a pruned subtree of $\mathcal{T}(j)$, all the ancestors of i belong to $\mathcal{T}(j)$.
- A pruned subtree of $\mathcal{T}$ shares the mapping $parent$ with $\mathcal{T}$.

Elimination tree: simple properties

The following observation is straightforward.

Observation

If $i \in anc_{\mathcal{T}}\{j\}$ for some $j \neq i$ then $i > j$.

- The connection between the mapping $parent$ and the sets of ancestors and descendants is emphasized by the next observation.

Observation

If i and j are vertices of the elimination tree $\mathcal{T}$ with $j < i \leq n$ then $i \in anc_{\mathcal{T}}\{j\}$ if and only if $j \in desc_{\mathcal{T}}\{i\}$ if and only if $parent^t(j) = i$ for some $t \geq 1$.

Elimination tree: simple properties

- Replications can be expressed using rooted trees.
- For example, instead of stating that there exists $t \geq 1$ such that $parent^t(j) = i$, we can write that $i \in anc_{\mathcal{T}}\{j\} \setminus \{j\}$.
- Rewriting the necessary and sufficient condition in Theorem above as the following corollary we get a clear characterization of the sparsity patterns of the rows of L.

Corollary

Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. If i and j are vertices of $\mathcal{T}$ with $j < i \leq n$ and $a_{ij} = 0$ then $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $j \in anc_{\mathcal{T}}(k)$ and $a_{ik} \neq 0$.

Elimination tree: row subtrees

- The subtree of $\mathcal{T}$ with vertices that correspond to the nonzeros of row i of L is called the i-th row subtree and is denoted by $\mathcal{T}_r(i)$.
- Formally, row subtree is a pruned subtree of $\mathcal{T}$ induced by the union of the vertex set

$$\{i\} \cup \{k \,|\, a_{ik} \neq 0 \text{ and } k < i\}$$

  with all vertices on the directed paths in $\mathcal{T}$ from k to i, that is, with all their ancestors from $\mathcal{T}_r(i)$.
- The root vertex is i and the leaf vertices are a subset of the column indices in the i-th row of the lower triangular part of A.
- The row subtrees are connected subgraphs of $\mathcal{T}$, even if $\mathcal{T}$ is not connected. If $\mathcal{T}$ can be found without determining the pattern of L, then $\mathcal{T}_r(i)$ can be used to derive the sparsity pattern of row i of L, without having to store each entry explicitly.

Elimination tree: row subtrees

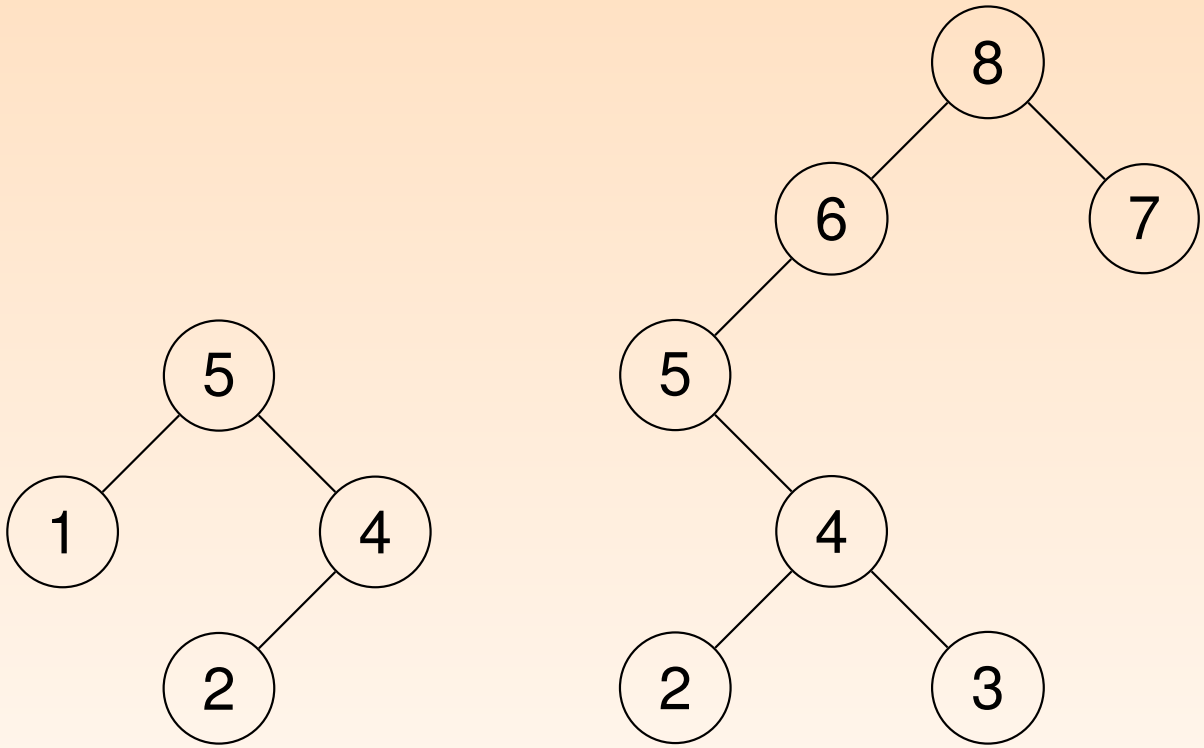
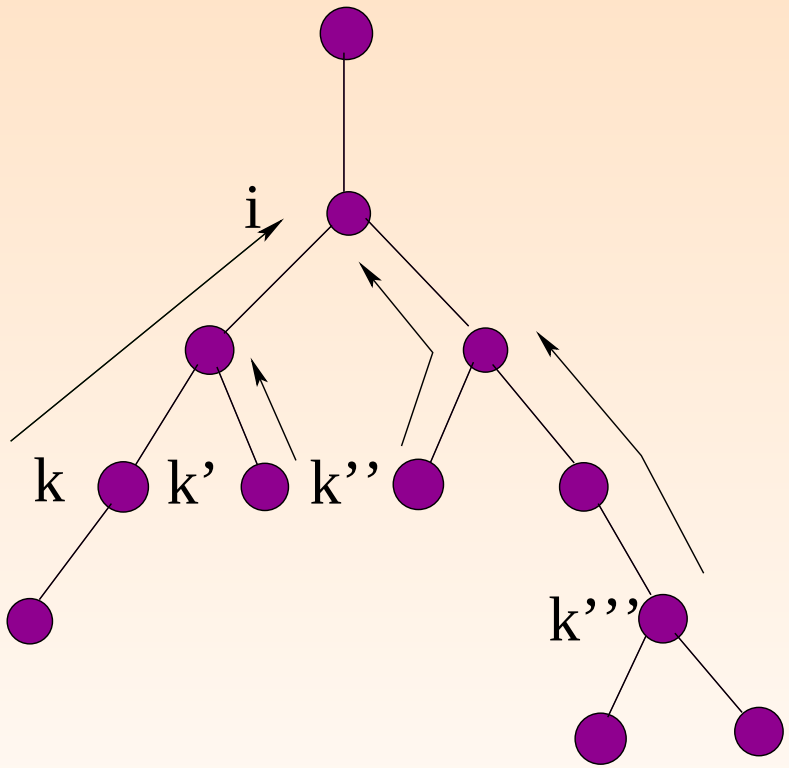


Figure: *The row subtree $\mathcal{T}_r(5)$ of the elimination tree $\mathcal{T}$ from above (left). Vertex 3 has been pruned because $a_{35} = 0$. The row subtree $\mathcal{T}_r(8)$ (right) differs from $\mathcal{T} = \mathcal{T}(A)$ because vertex 1 has been pruned ($a_{18} = 0$).*

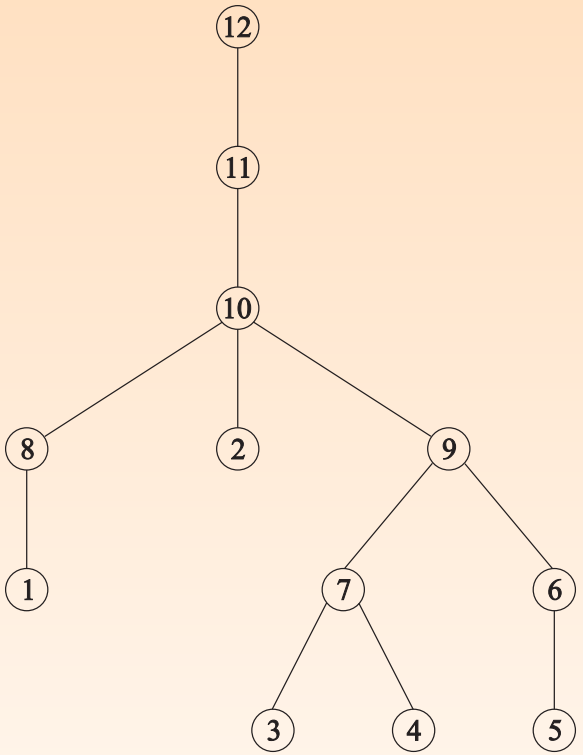
Elimination tree: row subtrees: another example



Symbolic Cholesky

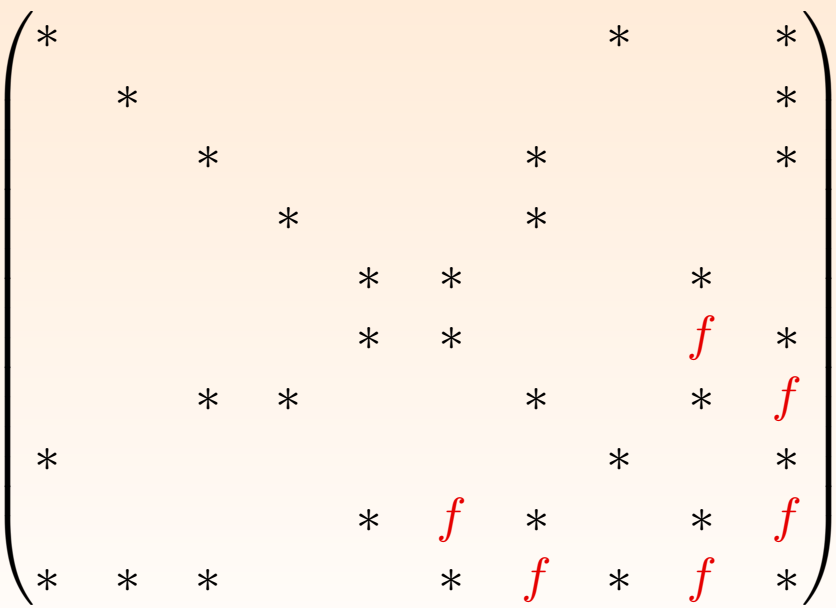
$$\begin{pmatrix}
* & & & & & & & * & & * \\
 & * & & & & & & & & * \\
 & & * & & & & * & & & * \\
 & & & * & & & * & & & \\
 & & & & * & * & & & * & \\
 & & & & * & * & & & f & * \\
 & & * & * & & & * & & * & f \\
* & & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * &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

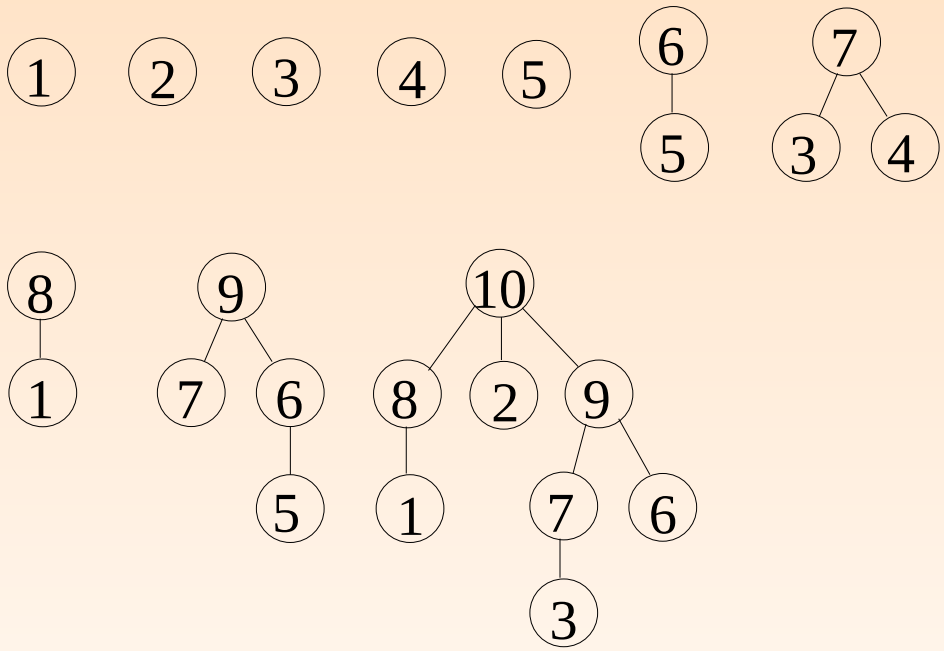
Row subtrees



Symbolic Cholesky

Row subtrees





Symbolic Cholesky

Theorem

If i and j are vertices in the elimination tree $\mathcal{T}$ with $j < i \leq n$ then $i \in anc_{\mathcal{T}}\{j\}$ if and only if there exists a path

$$j \overset{\mathcal{G}(A)}{\underset{\{1,\ldots,i\}}{\Longleftrightarrow}} i. \tag{28}$$

Proof.

Assume $i \in anc_{\mathcal{T}}\{j\}$. Then there is a path $j \overset{\mathcal{T}}{\Longrightarrow} i$ of length $l \geq 1$. Each edge of this path belongs to $\mathcal{G}(L)$ and corresponds either to an edge in $\mathcal{G}(A)$ or to a fill-path in $\mathcal{G}(A)$. Connecting these paths together gives (28).

Conversely, if the path (28) exists, induction on its length can be used to prove the result. If the path is of length 1 then the result holds because i and j are connected in $\mathcal{G}(A)$ by an edge. Consequently, i is an ancestor of j. Now assume that the result is true for all paths of length less than l ($l > 1$), and consider a path of length l. Let m be the largest vertex on this path. If $m < j$ then (28) is a fill-path connecting i and j and, therefore, $i \in anc_{\mathcal{T}}\{j\}$. Otherwise, for $m \geq j$ the assumption implies $i \in anc_{\mathcal{T}}\{m\}$ and $m \in anc_{\mathcal{T}}\{j\}$, that is, $i \in anc_{\mathcal{T}}\{j\}$. □ □

Symbolic Cholesky

Locality in characterization of ancestors

- Given a vertex j in $\mathcal{T}$, the following corollary indicates how to find $parent(j)$ (if it exists).
- If the set of ancestors of j is non empty then the lowest numbered one is its parent.

Corollary

Vertex i is the parent of vertex j in $\mathcal{T}$ if and only if i is the lowest numbered vertex satisfying $j < i \leq n$ for which there is a path (28).

- The existence of (28) is equivalent to requiring i and j belong to the same component of the graph $\mathcal{G}(A_{1:i,1:i})$ corresponding to the $i \times i$ principal leading submatrix $A_{1:i,1:i}$ of A.

Locality in characterization of ancestors

- Figure depicts $\mathcal{G}(A)$ for the matrix A given in Figure above.
- Consider vertex 4. Its set of ancestors for which the paths mentioned above exist comprises vertices 5, 6 and 8. Vertex 7 is not an ancestor of 4 because there is no path from 7 to 4 in the graph $\mathcal{G}(A_{1:7,1:7})$. Among the ancestors of 4, vertex 5 fulfils the condition from Corollary above and is thus the parent of 4.

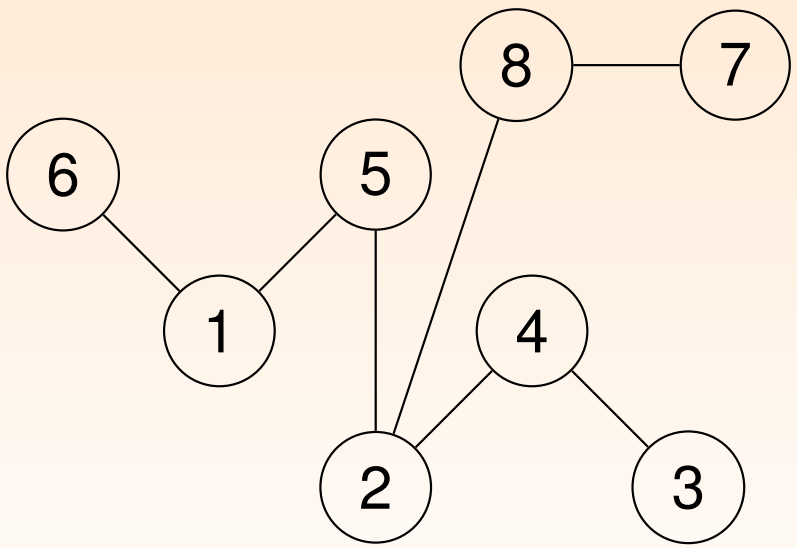


Figure: *The graph $\mathcal{G}(A)$ of the matrix from Figure 20 illustrating Theorem and Corollary above.*

Symbolic Cholesky

Constructing elimination tree

- Locality: $\mathcal{T} = \mathcal{T}(A)$ can be constructed by stepwise extensions of the elimination trees of the principal leading submatrices of A.
- Assume we have $\mathcal{T}(A_{1:i-1,1:i-1})$ and we want $\mathcal{T}(A_{1:i,1:i})$.
- Initialise $\mathcal{T}(A_{1:i,1:i}) = \mathcal{T}(A_{1:i-1,1:i-1})$. If there are no entries in row i of A to the left of the diagonal then there is nothing to do, only an isolated vertex i is added. Otherwise, i is the root of the row subtree $\mathcal{T}_r(i)$ and an ancestor of some vertex in $\mathcal{T}$.
- For each such vertex, say j, its ancestors k with $k < i$ are vertices of $\mathcal{T}(A_{1:i-1,1:i-1})$. Consider the directed path in $\mathcal{T}(A_{1:i,1:i})$ $t \geq 1$ such that $parent^t(j) = i$, we can write that $i \in anc_{\mathcal{T}}\{j\} \setminus \{j\}$. are connected subgraphs of $\mathcal{T}$.
- If $parent^t(j) \neq i$ then the new root of the subtree of $\mathcal{T}(A_{1:i,1:i})$ that contains j is added by setting $i = parent^{t+1}(j)$. Otherwise, i has already been added.

Constructing elimination tree

Algorithm (Construction of an elimination tree)

Input: *A with a symmetric sparsity pattern and its undirected graph $\mathcal{G}$.*
Output: *Elimination tree $\mathcal{T}$ described by the vector $parent$.*

1: **for** $i = 1 : n$ **do**
2: $parent(i) = 0$
3: **for** $j \in adj_{\mathcal{G}}\{i\}$ and $j < i$ **do** ▷ *For row i of the lower triangular part*
4: $jroot = j$
5: **while** $parent(jroot) \neq 0$ and $parent(jroot) \neq i$ **do**
6: $jroot = parent(jroot)$
7: **end while**
8: **if** $parent(jroot) = 0$ **then**
9: $parent(jroot) = i$ ▷ *Make i the parent of $jroot$*
10: **end if**
11: **end for**
12: **end for**

Sparse Cholesky factorization - components

Constructing elimination tree

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

Sparse Cholesky factorization - components

Constructing elimination tree

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

Sparse Cholesky factorization - components

Constructing elimination tree

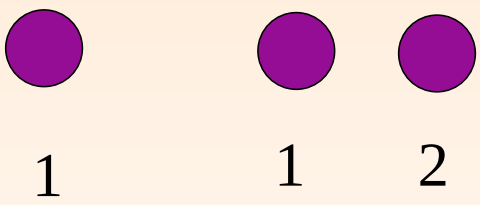
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \quad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$

1

Sparse Cholesky factorization - components

Constructing elimination tree

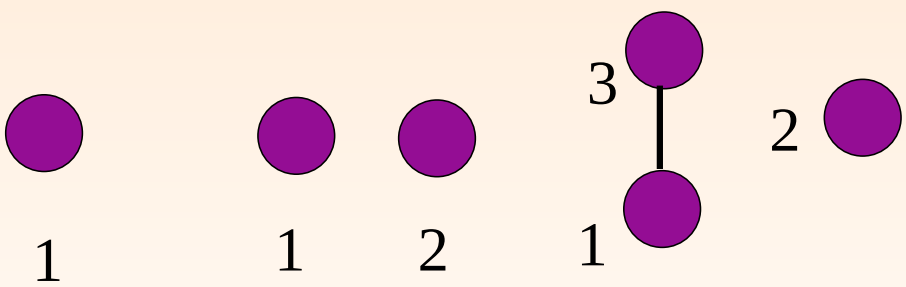
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \qquad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$



Sparse Cholesky factorization - components

Constructing elimination tree

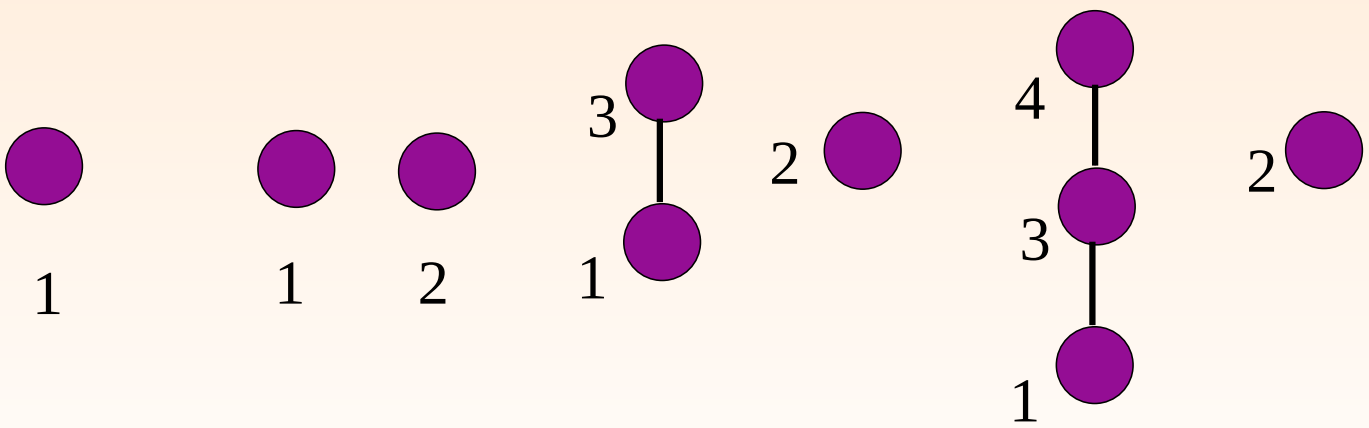
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \quad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$



Sparse Cholesky factorization - components

Constructing elimination tree

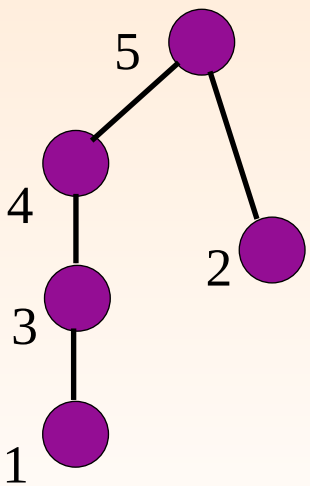
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \qquad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$



Sparse Cholesky factorization - components

Constructing elimination tree

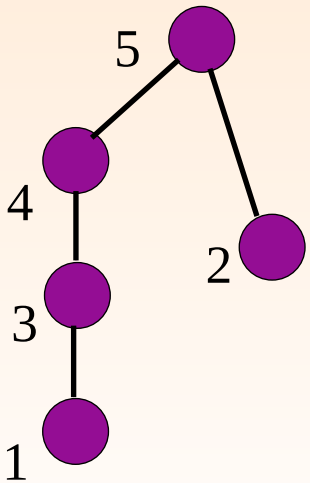
$$\begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & & * \\ * & * & * & & * & \\ & * & & * & & * \end{pmatrix} \qquad \begin{pmatrix} * & & * & & * & \\ & * & & & * & * \\ * & & * & * & * & \\ & & * & * & f & * \\ * & * & * & f & * & f \\ & * & & * & f & * \end{pmatrix}$$

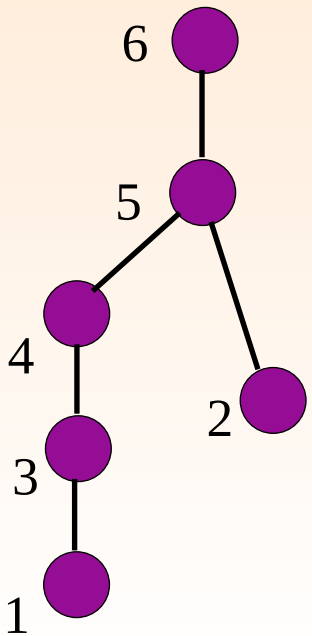


Sparse Cholesky factorization - components

Constructing elimination tree

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





Constructing elimination tree: comments

- The most expensive part of Algorithm to find the elimination tree is the while loop that searches for subtree roots. This search is based on tracing the directed path from j to its root $parent^t(j)$.
- Because this path is unique for a given j, shortcuts can be incorporated; this is called path compression.
- Having found a directed path from j to k, subsequent searches can be made more efficient by introducing a vector $ancestor$ and setting $ancestor(j) = k$.
- The modified algorithm is outlined below. It maintains two structures side-by-side using the current values of $parent$ and $ancestor$. The tree described by $ancestor$ is termed the virtual tree.

Symbolic Cholesky

Algorithm (Construction of an elimination tree using path compression)

Input: *A with a symmetric sparsity pattern and its undirected graph $\mathcal{G}$.*
Output: *Elimination tree $\mathcal{T}$ described by the vector $parent$.*

1: **for** $i = 1 : n$ **do**
2: $parent(i) = 0, ancestor(i) = 0$
3: **for** $j \in adj_{\mathcal{G}}\{i\}$ and $j < i$ **do**
4: $jroot = j$
5: **while** $ancestor(jroot) = 0$ and $ancestor(jroot) = i$ **do**
6: $l = ancestor(jroot)$
7: $ancestor(jroot) = i$ ▷ *To accelerate future searches*
8: $jroot = l$
9: **end while**
10: **if** $ancestor(jroot) = 0$ **then**
11: $ancestor(jroot) = i$ and $parent(jroot) = i$
12: **end if**
13: **end for**
14: **end for**

Constructing elimination tree with compression

- Figure below shows a matrix for which path compression makes building $\mathcal{T}$ significantly more efficient.

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

Figure: *A sparse matrix for which computing the elimination tree using Algorithm 7.2 is muc more efficient than using Algorithm 7.1.*

- For this example, $\mathcal{T}$ is determined by the mapping $parent(6) = 0$; $parent(i) = i + 1$ for $i = 1, \ldots, 5$.

Symbolic Cholesky

Constructing elimination tree with compression

- The complexity of the first Algorithm is $O(n^2)$.
- For this example the complexity of the Algorithm with compression is $O(n)$.
- Formally, the complexity of the new Algorithm is $O(nz(A)\log_2(n))$, where $nz(A)$ is the number of nonzeros of A but the logarithmic factor is rarely reached.
- Additional modifications can reduce the theoretical complexity to $O(nz(A)\,g(nz(A),n))$, where $g(nz(A),n)$ is a very slowly increasing function called the functional inverse of Ackermann's function.

Constructing elimination tree with compression

Independence of subtrees

- The following simple theorem states that there is no edge in $\mathcal{G}(L + L^T)$ between vertices belonging to subtrees of $\mathcal{T}$ with different vertex sets.
- Importance for parallel computations.

Theorem

Consider the elimination tree $\mathcal{T}$ and the Cholesky factor L of A. Let $\mathcal{T}(i)$ and $\mathcal{T}(j)$ be two vertex-disjoint subtrees of $\mathcal{T}$. Then for all $s \in \mathcal{T}(i)$ and $t \in \mathcal{T}(j)$ the entry l_{st} of L is zero.

Symbolic Cholesky

Sparsity pattern of L

- The explicit structure of L is not always required, only the numbers of nonzeros in each row and column of L are needed.
- For example, to allocate the storage
- Let $row_L\{i\}$ denote the sparsity pattern of the off-diagonal part of row i of L, that is,

$$row_L\{i\} = \mathcal{S}\{L_{i,1:i-1}\} = \{j \mid j < i,\ l_{ij} \neq 0\}, \quad 1 \leq i \leq n.$$

- The number of entries in L is

$$nz(L) = \sum_{i=1}^{n} |row_L\{i\}| + n.$$

- $row_L\{i\}$ is given by the vertices of the row subtree $\mathcal{T}_r(i)$.
- The complexity of the algorithm is $O(nz(L))$.

Symbolic Cholesky

Algorithm (Computation of the row sparsity patterns of the Cholesky factor L)

Input: *A with a symmetric sparsity pattern, its undirected graph $\mathcal{G}$ and elimination tree $\mathcal{T}$ described by the vector $parent$.*
Output: *Row sparsity patterns $row_L\{i\}$ of the Cholesky factor L of A ($1 \leq i \leq n$).*

1: ***for*** $i = 1 : n$ ***do*** ▷ *Loop over the rows of A*
2: $row_L\{i\} = \emptyset$ ▷ *Initialisation*
3: $mark(i) = i$
4: ***for*** $k \in adj_{\mathcal{G}}\{i\}$ *and* $k < i$ ***do*** ▷ *Loop over below diagonal entries in column i of A*
5: $j = k$
6: ***while*** $mark(j) \neq i$ ***do*** ▷ *Column j not yet encountered in row i*
7: $mark(j) = i$ ▷ *Flag j as encountered in row i*
8: $row_L\{i\} = row_L\{i\} \cup \{j\}$ ▷ *Add j to the sparsity pattern of row i*
9: $j = parent(j)$ ▷ *Move up the elimination tree*
10: ***end while***
11: ***end for***
12: ***end for***

Cholesky factorization skeleton

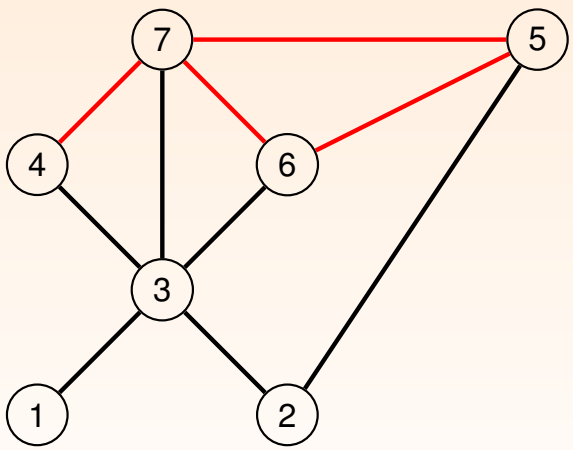
- Efficiency can be improved by employing the skeleton graph $\mathcal{G}(A^-)$ that is obtained from $\mathcal{G}(A)$ by removing every edge (i, j) for which $j < i$ and j is not a leaf vertex of $\mathcal{T}_r(i)$.
- $\mathcal{G}(A^-)$ is the smallest subgraph of $\mathcal{G}(A)$ with the same filled graph as $\mathcal{G}(A)$. The corresponding matrix is the skeleton matrix.
- The complexity of constructing the elimination tree using the skeleton matrix and its graph $\mathcal{G}(A^-)$ is $O(nz(A^-)\, g(nz(A^-), n))$, where $nz(A^-)$ is the number of entries in the skeleton matrix. Because $nz(A^-)$ is often significantly smaller than $nz(A)$, an implementation that processes $\mathcal{G}(A^-)$ rather than $\mathcal{G}(A)$ can be substantially faster.

Symbolic Cholesky

Cholesky factorization skeleton

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} * & & * & & & & \\ & * & * & & * & & \\ * & * & * & * & & * & * \\ & & * & * & & & * \\ & * & & & * & * & * \\ & & * & & * & * & * \\ & & * & * & * & * & * \end{pmatrix} \end{array}$$

$$A^- = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} * & & * & & & & \\ & * & * & & * & & \\ * & * & * & * & & * & * \\ & & * & * & & & \\ & * & & & * & & \\ & & * & & & * & \\ & & * & & & & * \end{pmatrix} \end{array}$$



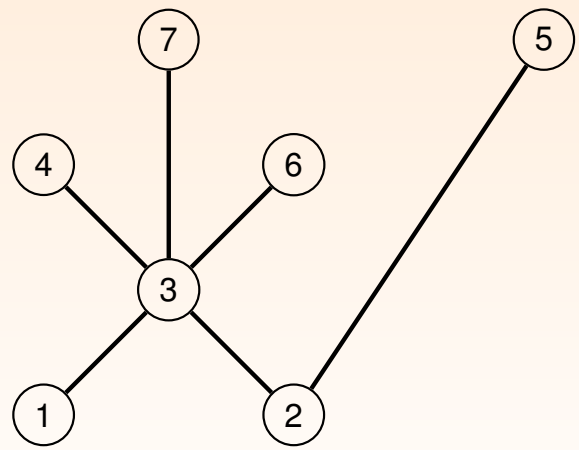
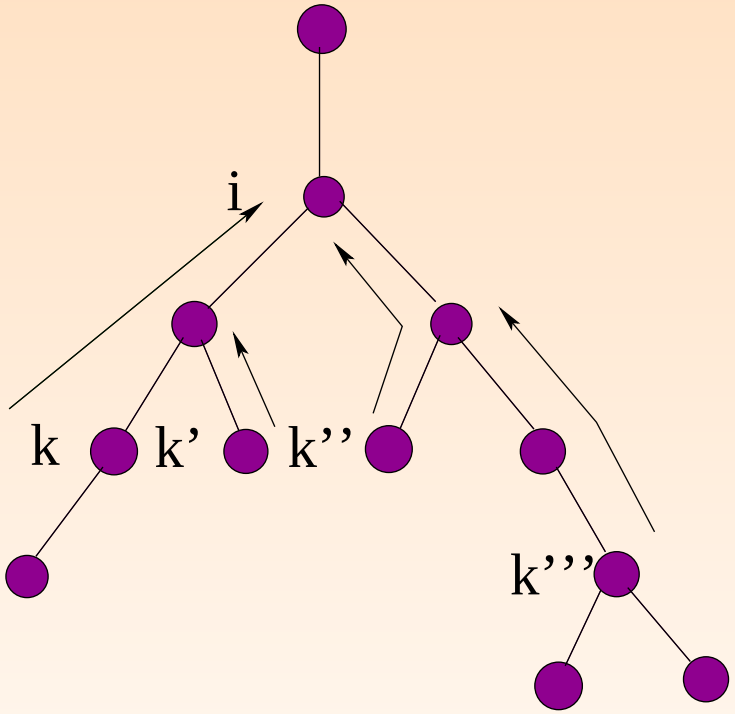


Figure: *An illustration of the sparsity pattern of A and its graph $\mathcal{G}(A)$ (left) and the sparsity pattern of the corresponding skeleton matrix A^- and graph $\mathcal{G}(A^-)$ (right). The entries in A and*

Symbolic Cholesky

Row counts: more spohisticated approach

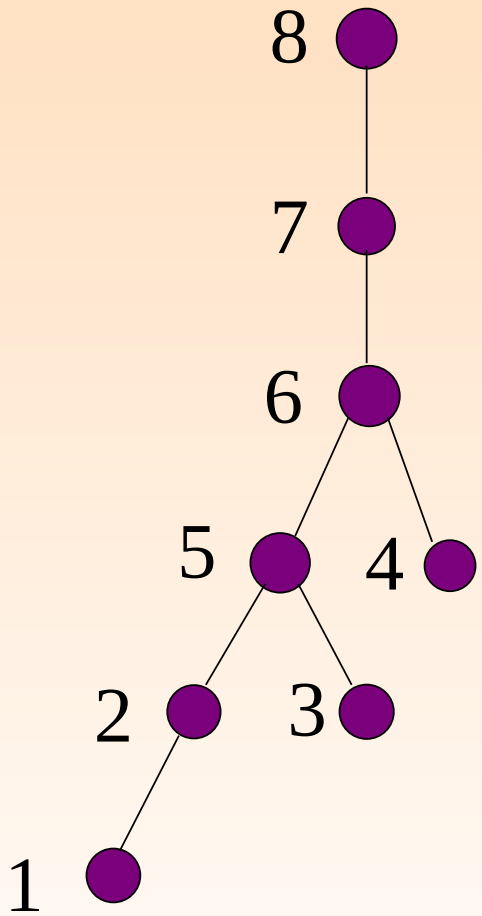
Row counts: more sophisticated algorithm: the idea



- Needed: fast algorithm to determine junctions of branches in the elimination tree,
- and fast algorithm to find leaves of the elimination tree.
- Can be done by traversing the postordered elimination tree.
- The complexity can be then nearly linear in m.

Sparse Cholesky factorization - components

It would be nice to know column structures of L as well



row structure

column structure

row subtrees

?

Column structures

- Analogously to the row sparsity patterns, let $col_L\{j\}$ denote the sparsity pattern of the off-diagonal part of column j of L, that is,

$$col_L\{j\} = \mathcal{S}(L_{j+1:n,j}) = \{i \mid i > j,\ l_{ij} \neq 0\}, \quad 1 \leq j \leq n.$$

- The column replication principle can be written as

$$col_L\{j\} \subseteq col_L\{parent(j)\} \cup parent(j).$$

Column structures

- Theorem describes $col_L\{j\}$ using the vertices of the subtree $\mathcal{T}(j)$.

Theorem

The column sparsity pattern $col_L\{j\}$ of the Cholesky factor L of the matrix A is equal to the adjacency set of vertices of the subtree $\mathcal{T}(j)$ in $\mathcal{G}(A)$, that is,

$$col_L\{j\} = adj_{\mathcal{G}(A)}\{\mathcal{T}(j)\}. \tag{29}$$

Proof.

If $i \in col_L\{j\}$ then $j \in row_L\{i\}$ and Theorem on necessary and sufficient condition for the fill-in implies $j \in anc_{\mathcal{T}}\{k\}$ for some k such that $a_{ik} \neq 0$. That is, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$. Conversely, $i \in adj_{\mathcal{G}}\{\mathcal{T}(j)\}$ implies that in row i the entry in column j of L is nonzero. Thus, $j \in row_L\{i\}$ and hence $i \in col_L\{j\}$. □ □

Column structures

- Algorithm can be used to compute the column counts and the column sparsity patterns because when j is added to $row_L\{i\}$ at line 8, i can be added to $col_L\{j\}$. This does not generally obtain the column sparsity patterns sequentially.

To derive an approach that does compute them sequentially consider

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \bigcup_{\{k \,|\, k \in \mathcal{T}(j)\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}.$$

Using the column replication, this can be significantly simplified

$$col_L\{j\} = \left(adj_{\mathcal{G}(A)}\{j\} \bigcup_{\{k \,|\, j = parent(k)\}} col_L\{k\} \right) \setminus \{1, \ldots, j\}. \quad (30)$$

Symbolic Cholesky

Column structures

- The following algorithm constructs the sparsity pattern of each column j of L as the union of the sparsity pattern of column j of A ($adj_{\mathcal{G}(A)}\{j\}$) and the patterns of the children of j in $\mathcal{T}(A)$.
- Here $child\{j\}$ denotes the set of children j.
- Because any child k of j satisfies $k < j$, the j-th outer step has the information needed to compute the sparsity pattern described by (30). Observe that $\mathcal{T}(A)$ does not need to be input.

Symbolic Cholesky

Algorithm (Determining sparsity patterns of columns of L)

Input: *A with symmetric sparsity pattern and its undirected graph $\mathcal{G}(A)$.*
Output: *Column sparsity patterns $col_L\{j\}$ of the Cholesky factor L of A $(1 \leq j \leq n)$.*

1: **for** $j = 1 : n$ **do**
2: $\quad child\{j\} = \emptyset$ ▷ *Initialisation*
3: **end for**
4: **for** $j = 1 : n$ **do** ▷ *Loop over the columns of L*
5: $\quad col_L\{j\} = adj_{\mathcal{G}(A)}\{j\} \setminus \{1, \ldots, j-1\}$
6: $\quad$ **for** $k \in child\{j\}$ **do** ▷ *Unifying child structures in (30)*
7: $\quad\quad col_L\{j\} = col_L\{j\} \cup col_L\{k\} \setminus \{j\}$
8: $\quad$ **end for**
9: $\quad$ **if** $col_L\{j\} \neq \emptyset$ **then**
10: $\quad\quad l = \min\{i \mid i \in col_L\{j\}\}$
11: $\quad\quad child\{l\} = child\{l\} \cup \{j\}$ ▷ *Parent of j detected using Corollary* **??**
12: $\quad$ **end if**
13: **end for**

Topological orderings

- The outer loop in Algorithm to find column sparsity patterns does not have to be performed in the strict order $j = 1, \ldots, n$.
- What is necessary is that for each step j, the column sparsity pattern for each child of j has already been computed.
- An ordering of the vertices in a tree (and, more generally, in a DAG) is a topological ordering if, for all i and j, $j \in desc_{\mathcal{T}}\{i\}$ implies $j < i$
- Observation above confirms that the ordering of vertices in the elimination tree $\mathcal{T}$ is a topological ordering.
- A new topological ordering of $\mathcal{T}$ defines a relabelling of its vertices corresponding to a symmetric permutation of A.

Topological orderings

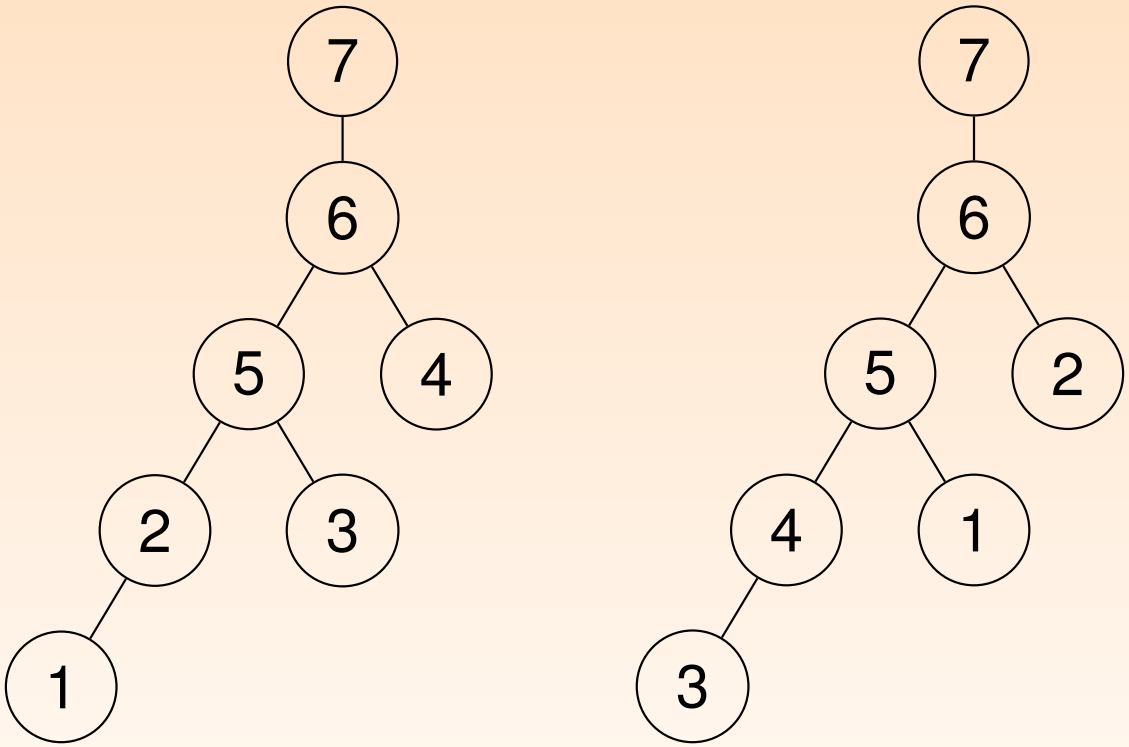


Figure: *Two topological orderings of an elimination tree.*

Symbolic Cholesky

Topological orderings

The sparsity patterns of the Cholesky factors of A and PAP^T can be different but the following result shows that the amount of fill-in is the same.

Theorem

Let $\mathcal{S}\{A\}$ be symmetric. If P is the permutation matrix corresponding to a topological ordering of the elimination tree $\mathcal{T}$ of A then the filled graphs of A and PAP^T are isomorphic.

Topological orderings

- There are many topological orderings of $\mathcal{T}$.
- One class is obtained using the depth-first search.
- This algorithm searches all the components of $\mathcal{T}$ starting at their root vertices.
- In this case, once vertex i has been visited, all the vertices of the subtree $\mathcal{T}(i)$ are visited immediately after i and i is labelled as the last vertex of $\mathcal{T}(i)$.

Postordering

- A topological ordering of $\mathcal{T}$ is a postordering if the vertex set of any subtree $\mathcal{T}(i)$ ($i = 1, \ldots, n$) is a contiguous sublist of $1, \ldots, n$.
- Unless additional rules on how vertices are selected are imposed, a postordering is generally not unique.
- One possible postordering is defined by the depth-first search, and this is apparently not unique.

Postordering

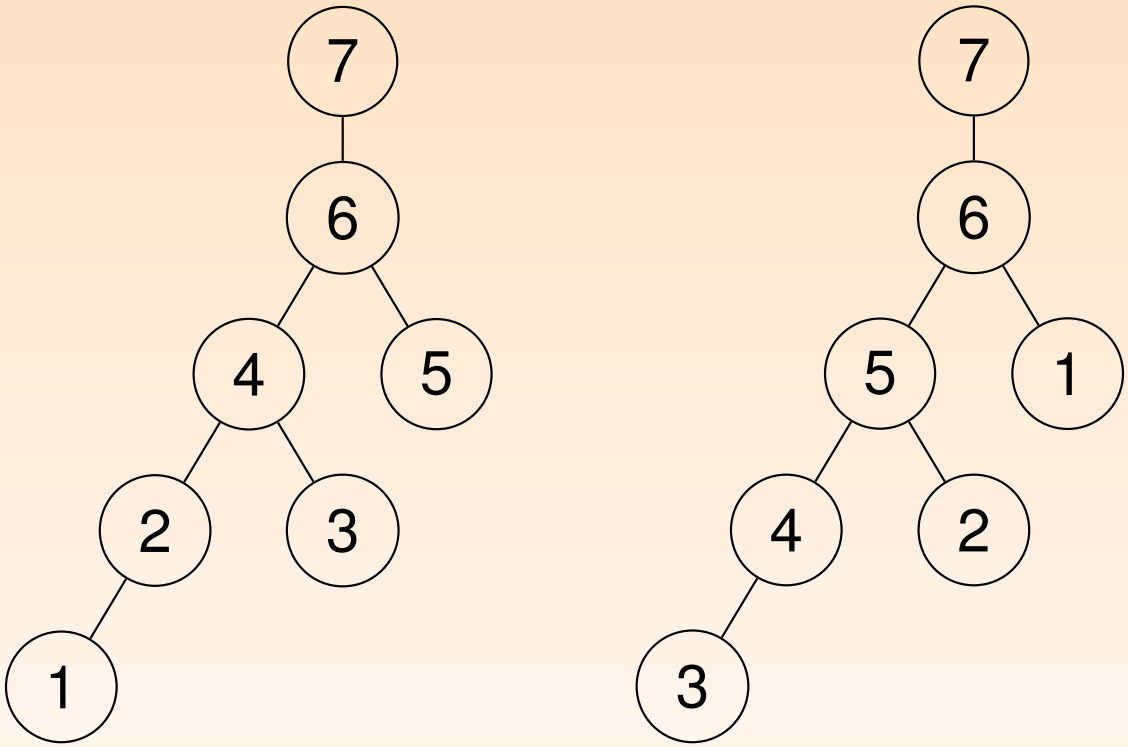


Figure: *An example to illustrate the non uniqueness of postorderings of an elimination tree.*

Symbolic Cholesky

Leaf vertices of row subtrees

- Leaf vertices of row subtrees play a key role in graph algorithms related to sparse Cholesky factorizations.
- They can be used to find the skeleton matrix described above.
- See the next theorem

Symbolic Cholesky

Theorem

Let the elimination tree $\mathcal{T}$ of A be postordered. Let the column indices of the nonzeros in the strictly lower triangular part of row i of A be $c_1, \ldots, c_s$ with $s \geq 1$ and $0 < c_1 < \ldots < c_s < i$. Then c_t is a leaf vertex of the row subtree $\mathcal{T}_r(i)$ if and only if

$$t = 1 \text{ or } (1 < t \leq s \text{ and } c_{t-1} \notin \mathcal{T}(c_t)).$$

Proof.

c_1 is always a leaf vertex of $\mathcal{T}_r(i)$. If this is not the case then there exists a directed path from some vertex $k, k \neq c_1$ to i via c_1 such that $k \in \mathcal{T}_r(i)$ and $a_{ik} \neq 0$. Postordering of $\mathcal{T}$ implies $k < c_1$. This is a contradiction because c_1 is the index of the first nonzero in row i.

Consider now $t > 1$. Assume that $c_{t-1} \in \mathcal{T}(c_t)$ and that c_t is a leaf vertex of $\mathcal{T}_r(i)$. Row replication implies any $k \in anc_{\mathcal{T}}\{c_{t-1}\}$ such that $c_{t-1} \leq k < i$ satisfies $l_{ik} \neq 0$. Because $\mathcal{T}$ is postordered, $c_{t-1} \leq k \leq c_t$ and there is at least one $k < c_t$ satisfying this inequality. It follows that $k = c_{t-1}$. Because k belongs to $\mathcal{T}_r(i)$, c_t cannot be a leaf vertex of $\mathcal{T}_r(i)$, which is a contradiction.

Conversely, assume for $t > 1$ that $c_{t-1} \notin \mathcal{T}(c_t)$ and c_t is not a leaf vertex of $\mathcal{T}_r(i)$. From the second part of the assumption and the fact that $c_t \in \mathcal{T}_r(i)$ it follows that there is at least one leaf

Leaf vertices of row subtrees

Corollary

Under the assumptions of the previous Theorem, c_t is a leaf vertex of $\mathcal{T}_r(i)$ if and only if

$$t = 1 \text{ or } (1 < t \leq s \text{ and } c_{t-1} < c_t - |\mathcal{T}(c_t)| + 1).$$

- Subtree sizes can be easily computed bottom up.
- Correctness of next Algorithm is guaranteed because $parent$ defines a topological ordering of $\mathcal{T}$.

Symbolic Cholesky

Subtree sizes - needed to get the leaf vertices of row subtrees

Algorithm (Find the sizes of subtrees $\mathcal{T}(i)$ of $\mathcal{T}$)

Input: *Elimination tree $\mathcal{T}$ described by the vector $parent$.*
Output: *Subtree sizes $|\mathcal{T}(i)|$ ($1 \leq i \leq n$).*

1: $|\mathcal{T}(1:n)| = 1$
2: ***for*** $i = 1 : n-1$ ***do***
3: $\quad k = parent(i)$
4: $\quad |\mathcal{T}(k)| = |\mathcal{T}(k)| + |\mathcal{T}(i)|$
5: ***end for***

Symbolic Cholesky

Leaf vertices of row subtrees by columns

- Theorem relaxes the condition that the entries in the rows of A are sorted by increasing column indices. This allows the leaf vertices of the row subtrees to be determined by columns.

Theorem

Consider the elimination tree $\mathcal{T}$ of A. Vertex j is a leaf vertex of some row subtree of $\mathcal{T}$ if and only if there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \leq n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$.

Proof.

Assume that for some $i \in anc_{\mathcal{T}}\{j\}$ vertex j is a leaf vertex of $\mathcal{T}_r(i)$. That is, $i \in adj_{\mathcal{G}(A)}\{j\}$, $i > j$. Suppose there exists k in $\mathcal{T}(j) \setminus \{j\}$ such that $i \in adj_{\mathcal{G}(A)}\{k\}$. Then all the ancestors of $k, k \leq i$, in particular j, belong to $\mathcal{T}_r(i)$ and j cannot be a leaf vertex of $\mathcal{T}_r(i)$. This is a contradiction.

Conversely, assume that j is not a leaf vertex of any row subtree of $\mathcal{T}$ and that there exists $i \in adj_{\mathcal{G}(A)}\{j\}$, $j < i \leq n$, such that $i \notin adj_{\mathcal{G}(A)}\{k\}$ for all $k \in \mathcal{T}(j) \setminus \{j\}$. Because j is not a leaf vertex of any such $\mathcal{T}_r(i)$, Theorem on necessary and sufficient fill-in conditions implies that there exists $k \in \mathcal{T}(j) \setminus \{j\}$ such that $a_{ik} \neq 0$, which gives a contradiction and completes the

Leaf vertices of row subtrees

- To find leaf vertices of row subtrees of $\mathcal{T}$, Algorithm uses a marking scheme based on Theorem above and exploits the postordering of $\mathcal{T}$.
- The auxiliary vector $prev\_nonz$ stores the column indices of the most recently encountered entries in the rows of the strictly lower triangular part of A.

Leaf vertices of row subtrees

Algorithm (Find leaf vertices of row subtrees of $\mathcal{T}$)

Input: *A with a symmetric sparsity pattern and a corresponding postordered elimination tree $\mathcal{T}$.*
Output: *Logical vector $isleaf$ with entries set to true for leaf vertices of row subtrees.*

1: $isleaf(1:n) = false, \quad prev\_nonz(1:n) = 0$
2: *Compute $|\mathcal{T}(1:n)|$* ▷ *Use Algorithm 7.5*
3: ***for*** $j = 1:n$ ***do*** ▷ *Loop over the columns of A*
4: ***for*** i *such that* $i > j$ *and* $a_{ij} \neq 0$ ***do*** ▷ *Row index in strictly lower triangular part of A*
5: $k = prev\_nonz(i)$ ▷ *Column index of most recently seen entry in row i*
6: ***if*** $k < j - |\mathcal{T}(j)| + 1$ ***then***
7: $isleaf(j) = true$ ▷ *j is a leaf vertex by Corollary* **??**
8: ***end if***
9: $prev\_nonz(i) = j$ ▷ *Flag j as the most recently seen entry in row i*
10: ***end for***
11: ***end for***

Symbolic Cholesky

Blocks

- Blocks are absolutely crucial to compute efficiently on contemporary computers: we need as much data as possible for a unit of data transfer inside memory hierarchy.
- In BLAS terminology:

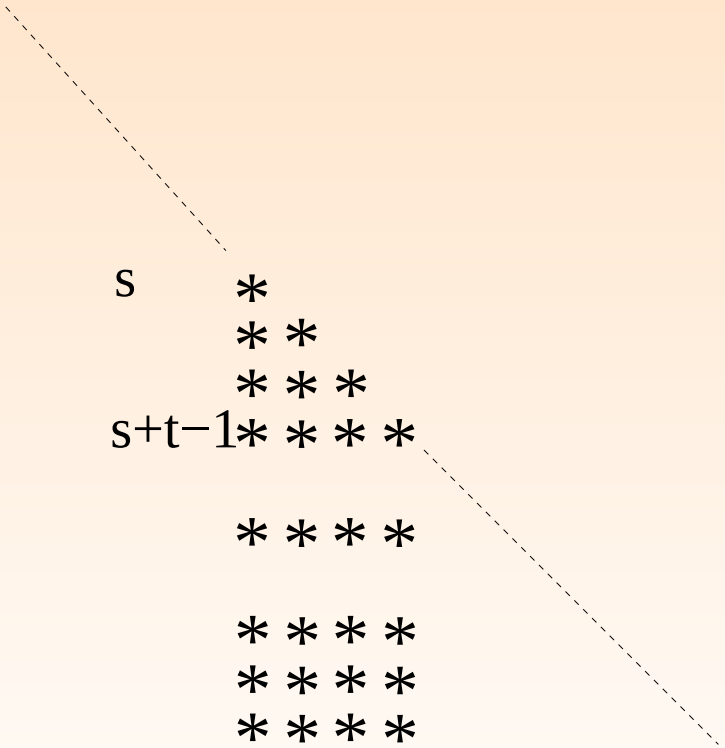
$$z = x + \alpha y \longrightarrow Z = X + \alpha Y \ (\ vector\ opes\ )$$

in general: saxpy $\longrightarrow$ dgemm

- But we have sparse matrices. It is not so straightforward to split their nonzeros into blocks.
- In fact, we need to reorder them in order to get blocks.
  - Application-based blocks in discretized systems.
  - Graph-based strategies which can be very fast.
  - But we need to optimize the block structure of L: supernodes.
  - Help: again our good friend, the elimination tree.

Sparse Cholesky factorization - components

Supernodes



Supernodes and the assembly tree

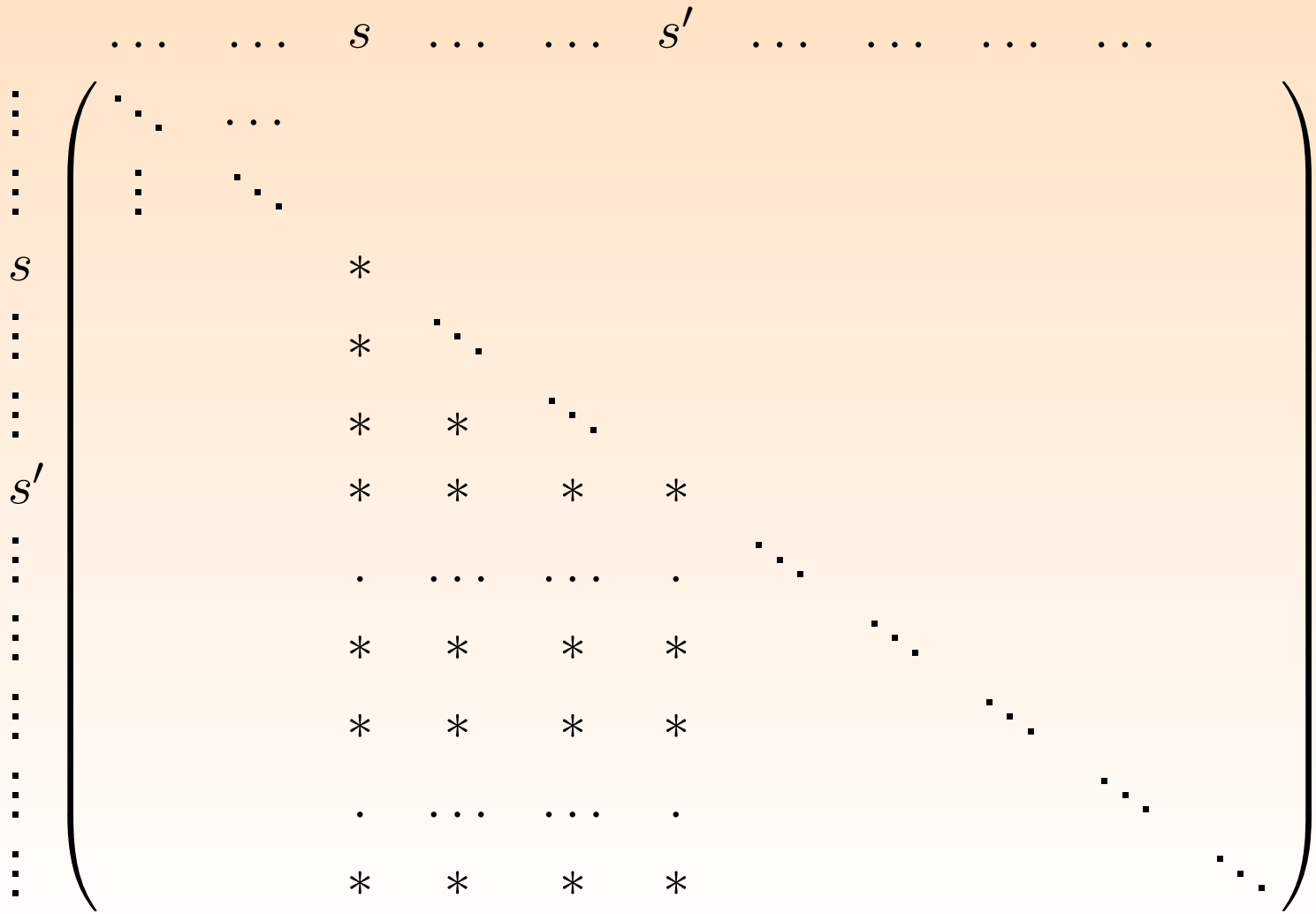
- Because of column replication, the columns of L generally become denser as the Cholesky factorization proceeds.
- To exploit this, we require the concept of supernodes. The idea: group together columns with the same sparsity structure, so that they can be treated as a dense matrix.
- Let $1 \leq s, t \leq n$ with $s + t - 1 \leq n$. A set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a supernode of L if

$$col_L\{s\} \cup \{s\} = col_L\{s+t-1\} \cup \{s, \ldots, s+t-1\}, \tag{31}$$

  and S cannot be extended for $s > 1$ by adding $s-1$ or for $s+t-1 < n$ by adding $s+t$.
- Because S cannot be extended it is a maximal subset of column indices. In graph terminology, a supernode is a maximal clique of contiguous vertices of $\mathcal{G}(L + L^T)$.
- A supernode may contain a single vertex.

Symbolic Cholesky

Supernodes



Supernodes

$$L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} * & & & & & & & \\ * & * & & & & & & \\ & & * & & & & & \\ & & * & * & & & & \\ * & * & * & * & * & & & \\ & & & & * & * & & \\ * & * & & & * & * & * & \\ & & * & * & * & * & * & * \end{pmatrix} \end{array}$$

Figure: *An example to illustrate supernodes in L. The first supernode comprises columns 1 and 2, the second columns 3 and 4, and the third columns 5 to 8.*

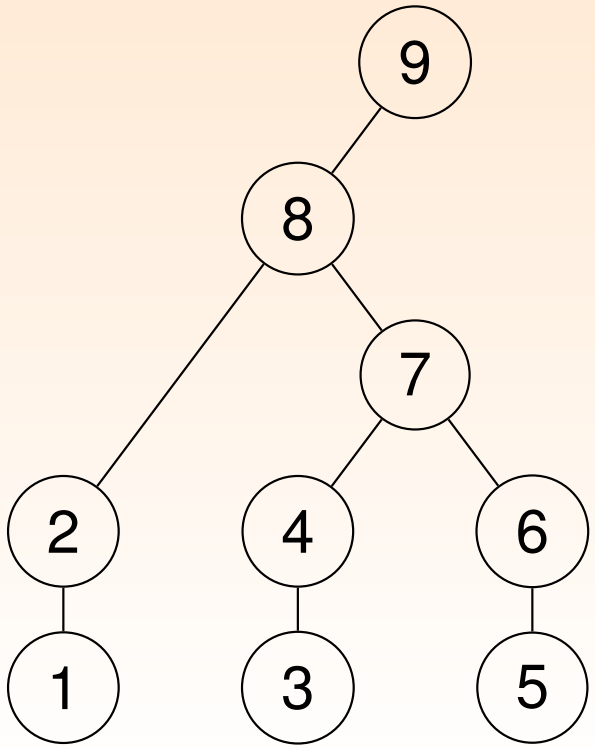
Supernodes

- The supernodal elimination or assembly tree is defined to be the reduction of the elimination tree that contains only supernodes.
- Each vertex of the elimination tree is associated with one elimination and a single integer (the index of its parent) is needed.
- Associated with each vertex of the assembly tree is an index list of the row indices of the nonzeros in the columns of the supernode. These implicitly define the sparsity pattern of L.

Symbolic Cholesky

- Demonstrating the difference between the elimination and assembly trees is given in Figure 28. Here the elimination tree is postordered and there are 5 supernodes: $\{1, 2\}$, 3, 4, 5, $\{6, 7, 8, 9\}$. For supernode 1 that comprises columns 1 and 2, the row index list is $\{1, 2, 8, 9\}$.

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\
\left(\begin{array}{ccccccccc}
* & * & & & & & & * & * \\
* & * & & & & & & * & * \\
 & & * & * & & & & * & \\
 & & * & * & & & * & * & \\
 & & & & * & * & * & & \\
 & & & & * & * & f & * & * \\
 & & & * & * & f & * & f & * \\
* & * & * & * & & * & f & * & f \\
* & * & & & & * & * & f & *
\end{array}\right)
\end{array}
$$



$4; \{$

Convenient characterization of supernodes

Theorem

The set of columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a supernode of L if and only if it is a maximal set of contiguous columns such that $s+i-1$ is a child of $s+i$ for $i = 1, \ldots, t-1$ and

$$|\, col_L\{s\}\,| = |\, col_L\{s+t-1\}\,| + t - 1. \tag{32}$$

Proof.

Let S be a supernode. For $i, j \in S$ with $i > j$ we have $i \in col_L\{j\}$. This implies that in the postordered elimination tree the vertex $i = j+1$ is the parent of j for $j = s, \ldots, s+t-2$. Moreover, from Observation 26, for any $i, j \in S$ with $i > j$,
$i \in col_L\{j\}$ implies $col_L\{j\} \setminus \{1, \ldots, i\} \subseteq col_L\{i\}$. Therefore,

$$|\, col_L\{s+i\}\,| \geq |\, col_L\{s+i-1\}\,| - 1, \quad i = 1, \ldots, t-1, \tag{33}$$

with equality if and only if

$$col_L\{s+i\} = col_L\{s+i-1\} \setminus \{s+i\},$$

Supernodes

- Enhance the efficiency of sparse factorizations and sparse triangular solves because they enable floating-point operations to be performed on dense submatrices rather than on individual nonzeros, thus improving memory hierarchy utilization and allowing the use of highly efficient dense linear algebra kernels (such as Level 3 BLAS kernels).
- Columns within a supernode are numbered consecutively but they can be numbered within the supernode in any order without changing the number of nonzeros in L (assuming the corresponding rows are permuted symmetrically). On some architectures, particularly those using GPUs, this freedom can be exploited to improve the factorization efficiency.
- Supernode amalgamation to achieve better efficiency.

Fundamental supernodes

- In practice, fundamental supernodes are easier to work with in the numerical factorization.
- Let $1 \leq s, t \leq n$ with $s + t - 1 \leq n$. A maximal set of contiguously numbered columns of L with indices $S = \{s, s+1, \ldots, s+t-1\}$ is a fundamental supernode if for any successive pair $i-1$ and i in the list, $i-1$ is the only child of i in $\mathcal{T}$ and $col_L\{i\} \cup \{i\} = col_L\{i-1\}$. s is termed the starting vertex.
- The difference between the sets of supernodes and fundamental supernodes is normally not large, with the latter having (slightly) more blocks in the resulting partitioning of L.
- Note that fundamental supernodes are independent of the choice of the postordering of $\mathcal{T}$.

Symbolic Cholesky

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
* & * & & & * & * \\
* & * & & f & f & \\
 & & * & * & * & * \\
 & & * & * & f & f \\
* & f & * & f & * & * \\
* & f & * & f & * & *
\end{pmatrix}
\end{array}
$$

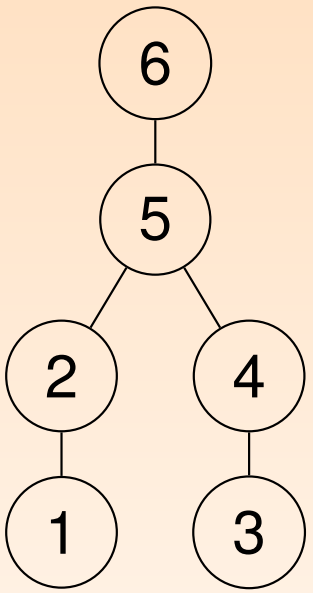


Figure: A matrix A and its postordered elimination tree $\mathcal{T}$ for which the set of supernodes $\{1, 2\}$ and $\{3, 4, 5, 6\}$ and the set of fundamental supernodes $\{1, 2\}, \{3, 4\}$ and $\{5, 6\}$ are different. The filled entries in $\mathcal{S}\{L + L^T\}$ are denoted by f.

Fundamental supernodes and leaves of row subtrees

Theorem

Assume $\mathcal{T}$ is postordered. Vertex s is the starting vertex of a fundamental supernode if and only if it has at least two child vertices in $\mathcal{T}$ or it is a leaf vertex of a row subtree of $\mathcal{T}$.

Proof.

If s has at least two child vertices then, from the definition of a fundamental supernode, it must be the starting vertex of a fundamental supernode. Assume that, for some $i > s$, s is a leaf vertex of $\mathcal{T}_r(i)$. If s is also a leaf vertex of $\mathcal{T}$ then s is a starting vertex of a supernode. The remaining case is s having only one child. Because $\mathcal{T}$ is postordered, this child must be $s-1$. Theorem of necessary and sufficient condititions for the fill-in implies $a_{is} \neq 0$ and $a_{i,s-1} = 0$, that is, $i \in col_L\{s\}$ and $i \notin col_L\{s-1\}$. It follows that

$$\mathcal{S}\{L_{s-1:n,s-1}\} \subsetneq \mathcal{S}\{L_{s:n,s}\} \cup \{s-1\},$$

and vertices s and $s-1$ cannot belong to the same supernode. Hence, s is the starting vertex of a new fundamental supernode.

Conversely, assume that s is the starting vertex of a fundamental supernode S. If s has no child vertices or at least two child vertices, the result follows. If s has exactly one child vertex.

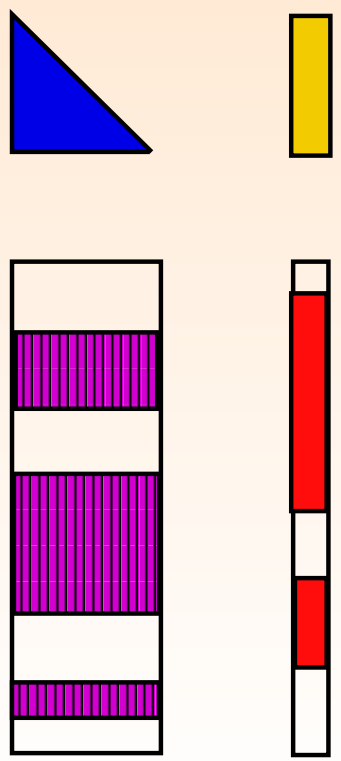
Complexity of symbolic operations

- Because fundamental supernodes are characterized by their starting vertices, they can be found by modifying Algorithm to incorporate marking leaf vertices of the row subtrees and vertices with at least two child vertices.
- Once the elimination tree has been computed, the complexity is $O(n + nz(A))$.
- The computation can be made even more efficient by using the skeleton graph $\mathcal{G}(A^{-})$.

Symbolic Cholesky

Supernodes and efficient computation

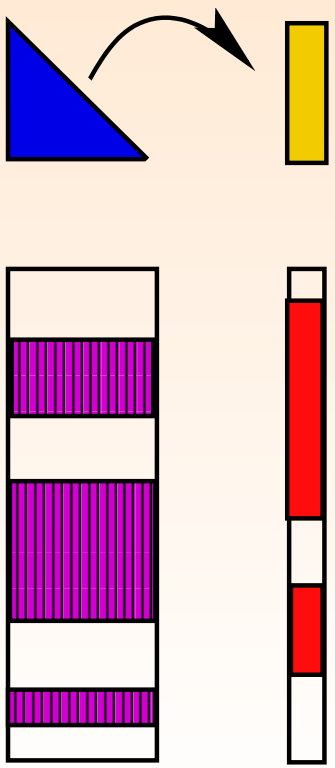
- the loop over rows has no indirect addressing: (dense BLAS1)



Symbolic Cholesky

Supernodes and efficient computation

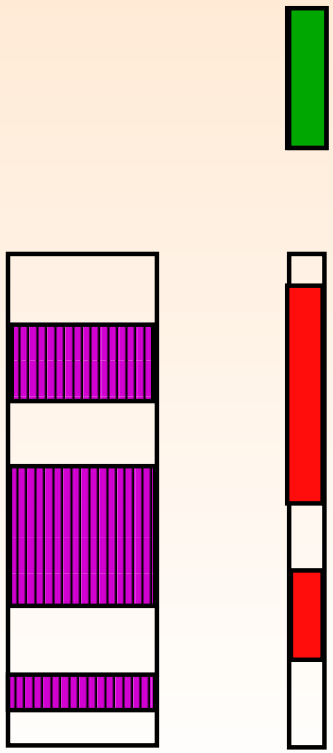
- the loop over rows has no indirect addressing: (dense BLAS1)



Symbolic Cholesky

Supernodes and efficient computation

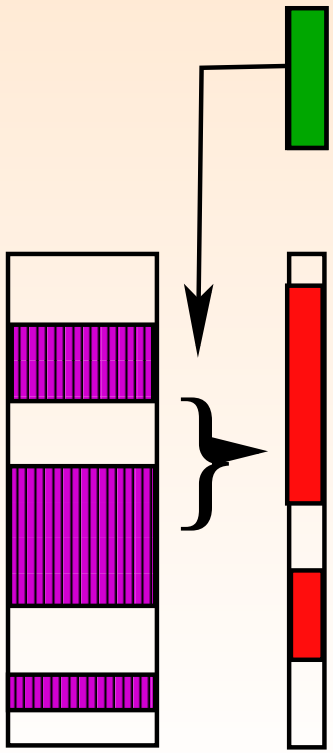
- the loop over rows has no indirect addressing: (dense BLAS1)
- the loop over columns of the updating supernode can be unrolled to save memory references (dense BLAS2)



Symbolic Cholesky

Supernodes and efficient computation

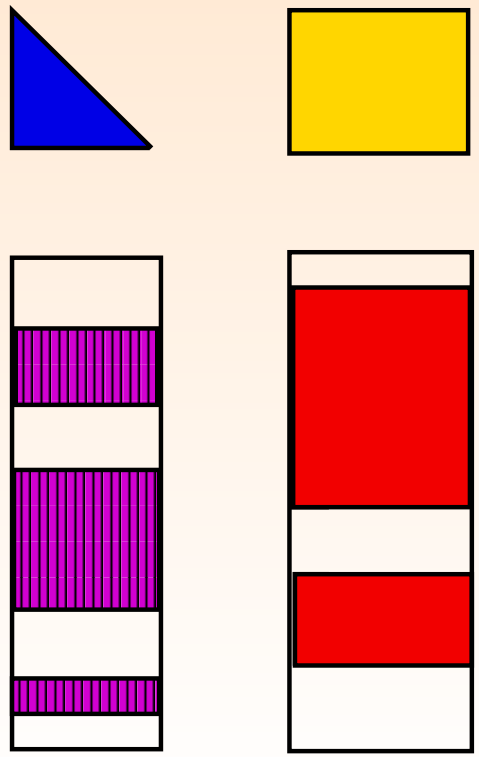
- the loop over rows has no indirect addressing: (dense BLAS1)
- the loop over columns of the updating supernode can be unrolled to save memory references (dense BLAS2)



Symbolic Cholesky

Supernodes and efficient computation

- the loop over rows has no indirect addressing: (dense BLAS1)
- the loop over columns of the updating supernode can be unrolled to save memory references (dense BLAS2)
- parts of the updating supernode can be used for blocks of updated supernode (dense BLAS3)



Outline

Factorization Cholesky

Numerical factorization: blocks and panels needed

- Positive definite (SPD) matrix is factorizable (strongly regular) and (in exact arithmetic) its Cholesky factorization $A = LL^T$ exists.
- Because efficient implementations of sparse Cholesky factorizations rely heavily on exploiting dense blocks, we first consider algorithms for the Cholesky factorization of dense matrices that can be applied to such blocks.
- The first one is an in-place algorithm because L can overwrite the lower triangular part of A (thus reducing memory requirements if A is no longer required).

In-place dense factorization

Algorithm (**In-place dense left-looking Cholesky factorization**)

Input: *Dense SPD matrix* A.
Output: *Factor* L *such that* $A = LL^T$.

1: ***for*** $j = 1 : n$ ***do***
2: $\quad L_{j:n,j} = A_{j:n,j}$ ▷ *Only the lower triangular part of* A *is required*
3: $\quad$ ***for*** $k = 1 : j-1$ ***do***
4: $\quad\quad L_{j:n,j} = L_{j:n,j} - L_{j:n,k}\, l_{jk}$ ▷ *Update column* j *using previous columns*
5: $\quad$ ***end for***
6: $\quad l_{jj} = (l_{jj})^{1/2}$ ▷ *Overwrite the diagonal entry with its square root*
7: $\quad L_{j+1:n,j} = L_{j+1:n,j} / l_{jj}$ ▷ *Scale off-diagonal entries in column* j
8: ***end for***

Reorganized using block panels (left-looking)

Algorithm (**In-place dense left-looking panel Cholesky factorization**)

Input: *Dense SPD matrix* A *with* nb *panels.*
Output: *Factor* L *such that* $A = LL^T$.

1: ***for*** $jb = 1 : nb$ ***do***
2: $\quad L_{jb:nb,jb} = A_{jb:nb,jb}$
3: $\quad$ ***for*** $kb = 1 : jb - 1$ ***do***
4: $\quad\quad L_{jb:nb,jb} = L_{jb:nb,jb} - L_{jb:nb,kb}\, L_{jb,kb}^T$ ▷ *Update block column* jb
5: $\quad$ ***end for***
6: $\quad$ *Compute in-place factorization of* $L_{jb,jb}$ ▷ *Overwrite* $L_{jb,jb}$ *with its Cholesky factor*
7: $\quad L_{jb+1:nb,jb} = L_{jb+1:nb,j}\, L_{jb,jb}^{-T}$ ▷ *Dense triangular solve*
8: ***end for***

Reorganized using panels (right-looking)

Algorithm (**In-place dense right-looking panel Cholesky factorization**)

Input: *Dense SPD matrix A in the form with nb panels.*
Output: *Factor L such that $A = LL^T$.*

1: ***for*** $jb = 1 : nb$ ***do***
2: $\quad L_{jb:nb,jb} = A_{jb:nb,jb}$
3: ***end for***
4: ***for*** $jb = 1 : nb$ ***do***
5: $\quad$ *Compute in-place factorization of* $L_{jb,jb}$ ▷ *Overwrite* $L_{jb,jb}$ *with its Cholesky factor*
6: $\quad L_{jb+1:nb,jb} = L_{jb+1:nb,j}\, L_{jb,jb}^{-T}$ ▷ *Dense triangular solve*
7: $\quad$ ***for*** $kb = jb + 1 : nb$ ***do***
8: $\quad\quad L_{kb:nb,kb} = L_{kb:nb,kb} - L_{kb:nb,jb}\, L_{kb,jb}^{T}$
9: $\quad$ ***end for***
10: ***end for***

Large panels split to blocks, right-looking

Algorithm (**In-place dense right-looking block Cholesky factorization)**

Input: *Dense SPD matrix A in the form (2) with $nb \times nb$ blocks.*
Output: *Factor L such that $A = LL^T$.*

1: **for** $jb = 1 : nb$ **do**
2: $\quad L_{jb:nb,jb} = A_{jb:nb,jb}$
3: **end for**
4: **for** $jb = 1 : nb$ **do**
5: $\quad$ Compute in-place factorization of $L_{jb,jb}$ ▷ Task factorize(jb)
6: $\quad$ **for** $ib = jb + 1 : nb$ **do**
7: $\quad\quad L_{ib,jb} = L_{ib,jb}\, L_{jb,jb}^{-T}$ ▷ Task solve(ib, jb)
8: $\quad\quad$ **for** $kb = jb + 1 : ib$ **do**
9: $\quad\quad\quad L_{ib,kb} = L_{ib,kb} - L_{ib,jb}\, L_{kb,jb}^{T}$ ▷ Task update(ib, jb, kb)
10: $\quad\quad$ **end for**
11: $\quad$ **end for**
12: **end for**

Factorization Cholesky

Tasks for efficient parallelization (still dense case)

- The panel and block descriptions of the factorization enable efficient parallelization. The three main block operations, which are called tasks, are factor(jb), solve(ib, jb) and update(ib, jb, kb).
- There are the following dependencies between the tasks.

  factorize(jb) depends on update(jb, kb, jb) for all $kb = 1, \ldots, jb - 1$.

  solve(ib, jb) depends on update(ib, kb, jb) for all $kb = 1, \ldots, jb - 1$, and factorize(jb).

  update(ib, jb, kb) depends on solve(ib, kb), solve(jb, kb).
- A dependency graph (DAG) can be used to schedule the tasks.

Factorization Cholesky

Sparse Cholesky factorizations

- Several classes of algorithms that implement sparse Cholesky factorizations.
- Their major differences relate to how they schedule the computations.
- First discussed: straightforward extension of the dense Cholesky factorization.

Factorization Cholesky

Sparse Cholesky factorizations

The entries of L satisfy the relationship

$$L_{j+1:n,j} = \left(A_{j+1:n,j} - \sum_{k=1}^{j-1} L_{j+1:n,k} l_{jk} \right) / l_{jj} \quad \text{with} \quad l_{jj} = \left(a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 \right)^{1}$$

Theorem

The numerical values of the entries in column $j > k$ of L depend on the numerical values in column k of L if and only if $l_{jk} \neq 0$.

Sparse Cholesky factorizations

- Straightforward sparse: $\mathcal{S}\{L\}$ already determined, static storage formats based, for example, on compressed columns and/or rows.

Algorithm (**Simplified sparse left-looking Cholesky**)

Input: *SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.*
Output: *Factor L such that $A = LL^T$.*

1: $l_{ij} = a_{ij}$ *for all* $(i,j) \in \mathcal{S}\{L\}$ ▷ *Filled entries in L are initialised to zero*
2: ***for*** $j = 1 : n$ ***do***
3: ***for*** $k \in \{k < j \,|\, l_{jk} \neq 0\}$ ***do***
4: ***for*** $i \in \{i \geq j \,|\, l_{ik} \neq 0\}$ ***do***
5: $l_{ij} = l_{ij} - l_{ik} l_{jk}$
6: ***end for***
7: ***end for***
8: $l_{jj} = (l_{jj})^{1/2}$
9: ***for*** $i \in \{i > j \,|\, l_{ij} \neq 0\}$ ***do***
10: $l_{ij} = l_{ij} / l_{jj}$
11: ***end for***
12: ***end for***

Sparse Cholesky factorizations

Algorithm (**Simplified sparse right-looking Cholesky**)

Input: *SPD matrix A and sparsity pattern $\mathcal{S}\{L\}$.*
Output: *Factor L such that $A = LL^T$.*

1: *For all $(i,j) \in \mathcal{S}\{L\}$ set $l_{ij} = a_{ij}$* ▷ *Filled entries in L are initialised to zero*
2: ***for*** $j = 1 : n$ ***do***
3: $\quad l_{jj} = (l_{jj})^{1/2}$
4: $\quad$ ***for*** $i \in \{i > j \,|\, l_{ij} \neq 0\}$ ***do***
5: $\quad\quad l_{ij} = l_{ij} / l_{jj}$
6: $\quad$ ***end for***
7: $\quad$ ***for*** $k \in \{k > j \,|\, l_{kj} \neq 0\}$ ***do***
8: $\quad\quad$ ***for*** $i \in \{i \geq k \,|\, l_{ij} \neq 0\}$ ***do***
9: $\quad\quad\quad l_{ik} = l_{ik} - l_{ij} l_{kj}$
10: $\quad\quad$ ***end for***
11: $\quad$ ***end for***
12: ***end for***

Sparse Cholesky factorizations

- The theoretical background based on the elimination tree $\mathcal{T}$ enables the dependencies to be searched for efficiently.
- $\mathcal{T}$ allows:
  - row (or column) counts of L to be computed
  - storage allocated
  - supernodes
- In practice, it can be beneficial to split large supernodes into smaller panels to better conform to computer caches.

Block column (left-looking) algorithm: notes on implementation

- The following theorem shows that we need to go through rows of columns of L computed so far within a block algorithm.

Theorem

Let $j > k$. Numerical values of entries in L_{*j} depend on the values of entries in L_{*k} iff $l_{jk} \neq 0$.

- Note that to get the sparsity patterns of columns we need less.

Block left-looking algorithm: notes on implementation

- Construction columns (block columns) one by one.
- Going through the rows can be simulated by linked lists as we saw in CSC-CSR matvec.
- Plan to mention this again when discussing approximate factorizations.

Sparse Cholesky factorizations

- An alternative for sparse matrices is to compute L one row at a time. This is sometimes called an up-looking factorization.
- Asymptotically optimal. Difficult to incorporate high level BLAS.
- The following relation holds for the i-th row of L

$$L_{i,1:i-1}^T = L_{1:i-1,1:i-1}^{-1} A_{1:i-1,i} \quad \text{with} \quad l_{ii}^2 = a_{ii} - L_{i,1:i-1} L_{i,1:i-1}^T.$$

The application of $L_{1:i-1,1:i-1}^{-1}$ can be implemented by solving the triangular system

$$L_{1:i-1,1:i-1} y = A_{1:i-1,i},$$

and setting $L_{i,1:i-1}^T = y$.

Triangular systems and sparsity pattern of L

Theorem

Consider a sparse lower triangular matrix L and the DAG $\mathcal{G}(L^T)$ with vertex set $\{1, 2, \ldots, n\}$ and edge set $\{(j \longrightarrow i) \mid l_{ij} \neq 0\}$. The sparsity pattern $\mathcal{S}\{y\}$ of the solution y of the system $Ly = b$ is the set of all vertices reachable in $\mathcal{G}(L^T)$ from $\mathcal{S}\{b\}$.

Proof.

From factorization Algorithm and assuming the non-cancellation assumption, we see that (a) if $b_i \neq 0$ then $y_i \neq 0$ and (b) if for some $j < i$, $y_j \neq 0$ and $l_{ij} \neq 0$ then $y_i \neq 0$. These two conditions can be expressed as a graph transversal problem in $\mathcal{G}(L^T)$. (a) adds all vertices in $\mathcal{S}\{b\}$ to the set of visited vertices and (b) states that if vertex j has been visited then all its neighbours in $\mathcal{G}(L^T)$ are added to the set of visited vertices. Thus $\mathcal{S}\{y\} = \mathcal{R}each(\mathcal{S}\{b\}) \cup \mathcal{S}\{b\}$. □

Triangular systems and sparsity pattern of L

Figure illustrates the sparsity patterns of a lower triangular matrix L and vector b together with $\mathcal{G}(L^T)$. The vertices that are reachable from $\mathcal{S}\{b\} = \{2, 4\}$ are 5 and 6 and thus $\mathcal{S}\{y\} = \{2, 4, 5, 6\}$.

$$L = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} * & & & & & \\ * & * & & & & \\ & & * & & & \\ * & & * & * & & \\ & & & * & * & \\ & * & * & & * & * \end{pmatrix} \end{array} \qquad b = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{c} 1 \end{array} \\ \begin{pmatrix} * \\ \\ * \\ \\ \\ \end{pmatrix} \end{array}$$

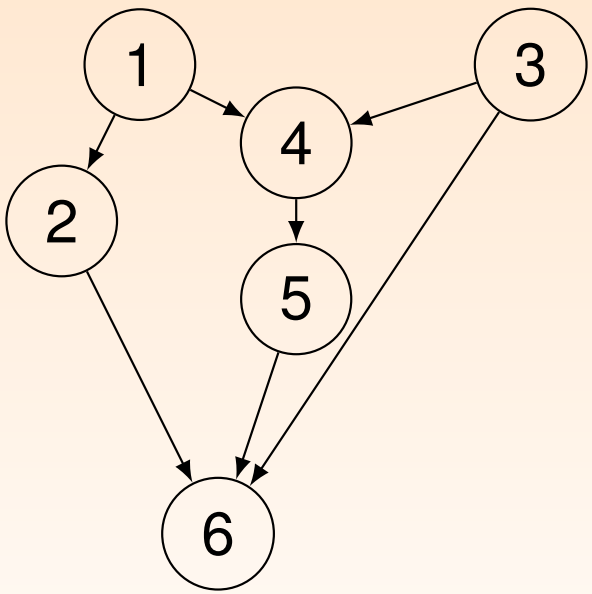


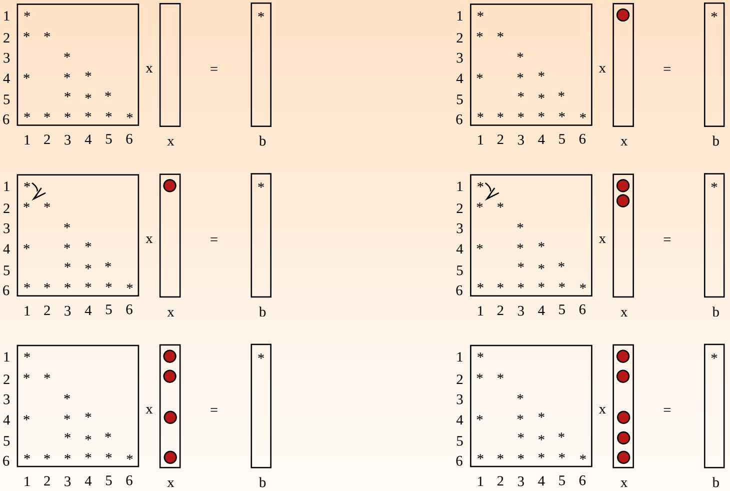
Figure: *An example to illustrate L, b and $\mathcal{G}(L^T)$.*

Triangular systems and sparsity pattern of L

$$\begin{pmatrix} * & & & & \\ & * & & & \\ * & & * & & \\ & & & * & \\ & & * & & * \end{pmatrix} \begin{pmatrix} * \\ * \\ \\ \\ * \end{pmatrix} = \begin{pmatrix} * \\ \\ \\ \\ \end{pmatrix} \tag{34}$$

- The only nonzero of the right-hand side implies the three nonzeros in the solution

Triangular systems and sparsity pattern of L



Up-looking Cholesky - comments

- Sparse row Cholesky factorization is based on the repeated solution of triangular linear systems.
- Theorem above can be used to determine the sparsity pattern of row i at Step 3, that is, by finding all the vertices that are reachable in $\mathcal{G}(L^T_{1:j-1,1:j-1})$ from the set $\{i \,|\, a_{ij} \neq 0,\, i < j\}$.
- A depth-first search of $\mathcal{G}(L^T_{1:j-1,1:j-1})$ determines the vertices in the row sparsity patterns in topological order, and performing the numerical solves in that order correctly preserves the numerical dependencies.
- Another option is to find the row subtrees using $\mathcal{T}(A)$.

Factorization Cholesky

Algorithm (**Sparse up-looking Cholesky factorization**)

Input: *SPD matrix* A.
Output: *Factor* L *such that* $A = LL^T$.

1: $l_{11} = (a_{11})^{1/2}$
2: ***for*** $i = 2 : n$ ***do***
3: *Find* $\mathcal{S}\{L_{i,1:i-1}\}$ ▷ *Sparsity pattern of* $L_{i,1:i-1}$
4: $L^T_{i,1:i-1} = L^{-1}_{1:i-1,1:i-1} A_{1:i-1,i}$ ▷ *Sparse triangular solve*
5: $l_{ii} = a_{ii} - L_{i,1:i-1} L^T_{i,1:i-1}$
6: $l_{ii} = (l_{ii})^{1/2}$
7: ***end for***

Supernodal algorithms

- Sophisticated supernodal algorithms: efficiency in parallel computational environments.
- Assume supernodal structure. Arithmetic of dense trapezoidal matrices. Blocks termed a nodal matrix.

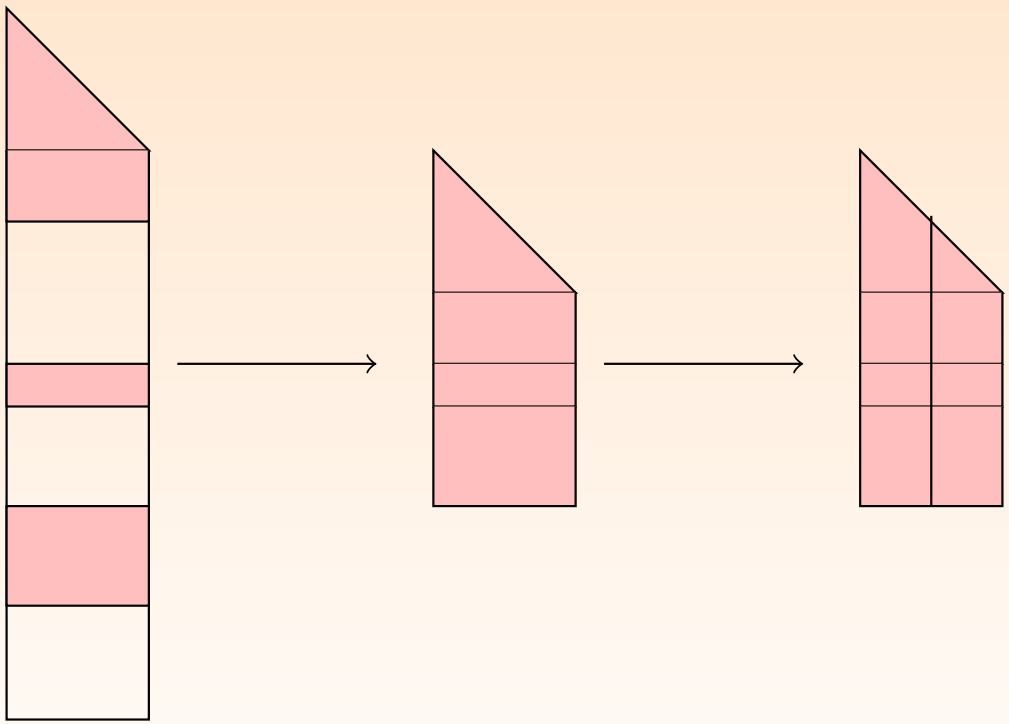


Figure: An illustration of a supernode (left), the corresponding nodal matrix (centre), and the nodal matrix with two panels (right).

Factorization Cholesky

Comments on (right-looking) supernodal processing

- Once a supernode is ready to be factorized, a dense Cholesky factorization of the block on the diagonal of the nodal matrix is performed
- Then a triangular solve is performed with the computed factor and the rectangular part of the nodal matrix.
- Iterate over ancestors of the supernode in the assembly tree.
- For each parent, the rows of the current supernode for each parent columns are identified. The outer product of those rows and the subdiagonal part of the supernode (update operations).
- The resulting matrix can be held in a temporary buffer. The rows and columns of this buffer are matched against indices of the ancestors and are added to them in a sparse scatter operation. For efficiency, the updates may use panels so that the temporary buffer remains in cache.

Factorization Cholesky

DAG-based approach and the sparse case

Each nodal matrix is subdivided into blocks.

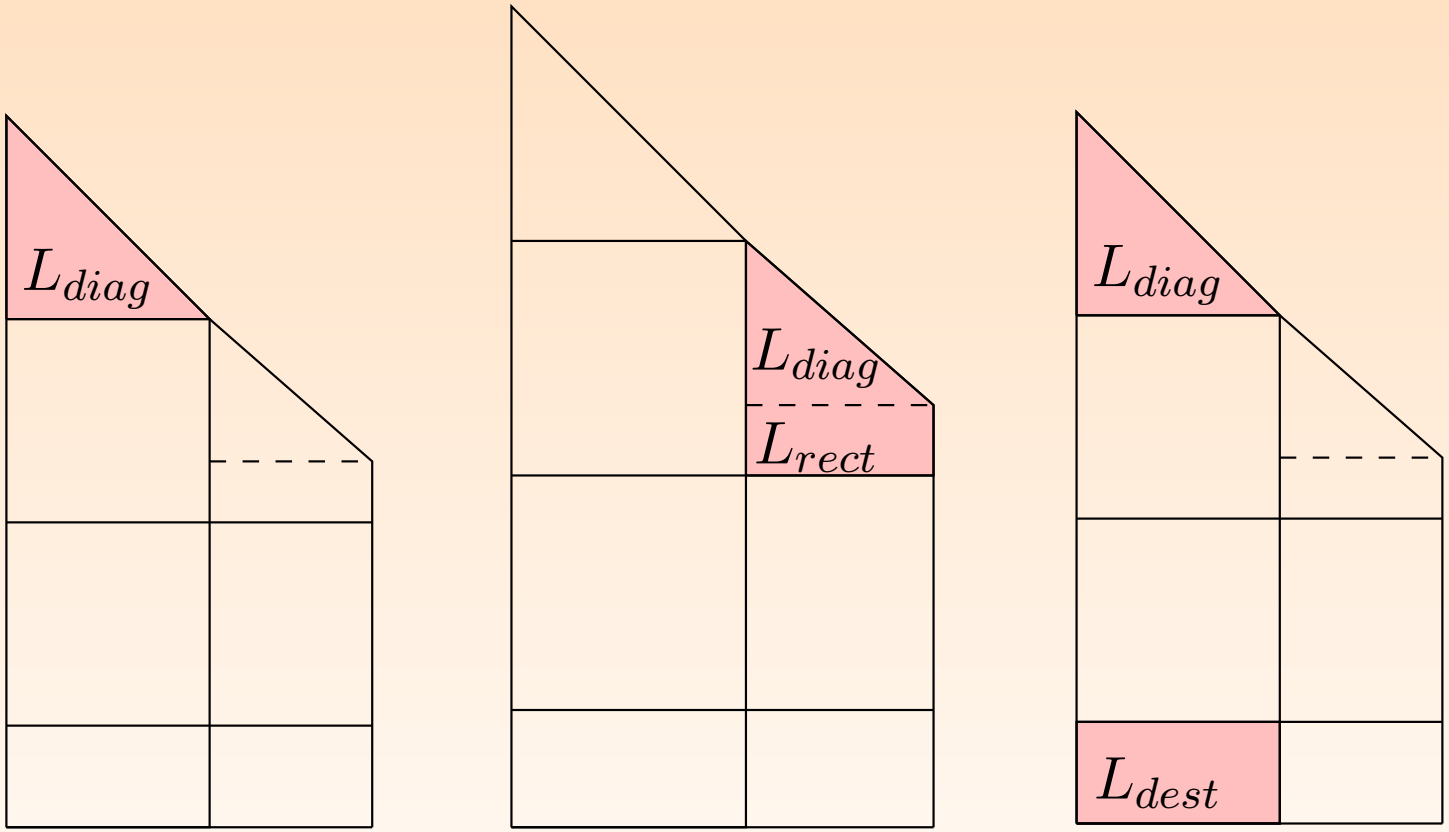


Figure: An illustration of a blocked nodal matrix with two block columns. The first block on the diagonal is triangular and the second one is trapezoidal. The task factorize_block is illustrated on the left and in the centre; the task solve_block is illustrated on the right.

Factorization Cholesky

DAG-based approach and the sparse case

- Different splitting into tasks.
- The key difference: distinguish: update_internal (the same nodal matrix), update_between (different nodal matrices).

factorize_block(L_{diag}) Computes the dense Cholesky factor L_{diag} of the block on the diagonal (leftmost plot). If the block is trapezoidal, the factorization is followed by a triangular solve of its rectangular part $L_{rect} = L_{rect}L_{diag}^{-T}$ (centre).

solve_block(L_{dest}) Performs a triangular solve of an off-diagonal block L_{dest} of the form $L_{dest} = L_{dest}L_{diag}^{-T}$ (rightmost).

update_internal (L_{dest}, L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_{dest}, L_r and L_c belong to the same nodal matrix.

update_between (L_{dest}, L_r, L_c) Performs the update $L_{dest} = L_{dest} - L_r L_c^T$, where L_r and L_c belong to the same nodal matrix and L_{dest} belongs to a different nodal matrix.

Factorization Cholesky

DAG-based approach and the sparse case

- Again, the tasks are partially ordered and a task DAG is used to capture the dependencies.
- For example, the updating of a block of a nodal matrix from a block column of L that is associated with a descendant of the supernode has to wait until all the relevant rows of the block column are available.
- At each stage of the factorization, tasks will be executing (in parallel) while others are held (in a stack or pool of tasks) ready for execution.

Factorization Cholesky

Alternative way: the multifrontal method

Theorem

Let A be SPD and let $\mathcal{T}$ be its elimination tree. The numerical values of entries in column k of the Cholesky factor L of A only affect the numerical values of entries in column i of L for $i \in anc_{\mathcal{T}}\{k\}$ ($1 \leq k \leq n-1$).

Proof.

Setting $S^{(1)} = A$, for $k \geq 2$ the $(n-k+1) \times (n-k+1)$ Schur complement $S^{(k)}$ can be expressed as

$$S^{(k)} = S^{(k-1)}_{k:n,k:n} - \begin{pmatrix} l_{k,k-1} \\ \vdots \\ l_{n,k-1} \end{pmatrix} \begin{pmatrix} l_{k,k-1} & \cdots & l_{n,k-1} \end{pmatrix} = S^{(k-1)}_{k:n,k:n} - L_{k:n,k-1} L^T_{k:n,k-1}. \quad (35)$$

Theorem on replication implies that all nonzero entries l_{ik} in column k of L explicitly used in the update are such that $i \in anc_{\mathcal{T}}\{k\}$. Considering the Cholesky factorization as a sequence of Schur complement updates, only columns i with $i \in anc_{\mathcal{T}}\{k\}$ can be influenced numerically by

Alternative way: the multifrontal method

- The computation of subsequent Schur complements by adding individual updates is straightforward; the multifrontal method employs further modifications and enhancements of this basic concept.
- Because the vertices of $\mathcal{T}$ are topologically ordered, the order in which the updates are applied progresses up the tree from the leaf vertices to the root vertex.
- This allows the computation of $S^{(k)}$ to be rewritten as

$$S^{(k)} = A_{k:n,k:n} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} L_{k:n,j} L_{k:n,j}^T,$$

  emphasizing the role of $\mathcal{T}$.
- In place of Schur complements, the multifrontal method uses frontal matrices connected to subtrees of $\mathcal{T}$.

Alternative way: the multifrontal method

- Assume $k, k_1, \ldots, k_r$ are the row indices of the nonzeros in column k of L.
- The frontal matrix F_k of the k-th subtree $\mathcal{T}(k)$ of $\mathcal{T}$ is the dense $(r+1) \times (r+1)$ matrix defined by

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \ldots & a_{kk_r} \\ a_{k_1k} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_rk} & 0 & \ldots & 0 \end{pmatrix} - \sum_{j \in \mathcal{T}(k) \setminus \{k\}} \begin{pmatrix} l_{kj} \\ l_{k_1j} \\ \vdots \\ l_{k_rj} \end{pmatrix} \begin{pmatrix} l_{kj} & l_{k_1j} & \ldots & l_{k_rj} \end{pmatrix} \tag{36}$$

Factorization Cholesky

- One step of the Cholesky factorization of F_k can be written as

$$F_k = \begin{pmatrix} l_{kk} & 0 & \dots & 0 \\ l_{k_1 k} & & & \\ \vdots & & I & \\ l_{k_r k} & & & \end{pmatrix} \begin{pmatrix} 1 & & \\ & V_k & \\ & & \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1 k} & \dots & l_{k_r k} \\ 0 & & & \\ \vdots & & I & \\ 0 & & & \end{pmatrix} \tag{37}$$

$$= \begin{pmatrix} l_{kk} \\ l_{k_1 k} \\ \vdots \\ l_{k_r k} \end{pmatrix} \begin{pmatrix} l_{kk} & l_{k_1 k} & \dots & l_{k_r k} \end{pmatrix} + \begin{pmatrix} 0 & & \\ & V_k & \\ & & \end{pmatrix}, \tag{38}$$

- V_k is termed a generated element (it is also sometimes called an update matrix or a contribution block).
- The name "generated element" is because the multifrontal method has its origins in the simpler frontal method, which uses a single frontal matrix.

Factorization Cholesky

- Equating the last r rows and columns yields

$$V_k = -\sum_{j \in \mathcal{T}(k)} \begin{pmatrix} l_{k_1 j} \\ \vdots \\ l_{k_r j} \end{pmatrix} \begin{pmatrix} l_{k_1 j} & \dots & l_{k_r j} \end{pmatrix}. \tag{39}$$

Assume that c_j $(j = 1, \dots, s)$ are the children of k in $\mathcal{T}$. The set $\mathcal{T}(k) \setminus \{k\}$ is the union of disjoint sets of vertices in the subtrees $\mathcal{T}(c_j)$. Each of these subtrees is represented in the overall update by the generated element. Thus, F_k can be written in an recursive form as follows

$$F_k = \begin{pmatrix} a_{kk} & a_{kk_1} & \dots & a_{kk_r} \\ a_{k_1 k} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{k_r k} & 0 & \dots & 0 \end{pmatrix} \leftrightarrows V_{c_1} \leftrightarrows \dots \leftrightarrows V_{c_s}. \tag{40}$$

Factorization Cholesky

- Here, the operation $\leftrightarrow\!\!\!\!\updownarrow$ denotes the addition of matrices that have row and column indices belonging to subsets of the same set of indices (in this case, $k, k_1, \ldots, k_r$); entries that have the same row and column indices are summed. This is referred to as the extend-add operator.
- Adding a row and column of A and the generated elements into a frontal matrix is called the assembly.
- A variable is fully summed if it is not involved in any rows and columns of A that have still to be assembled or in a generated element.
- Once a variable is fully summed, it can be eliminated.
- A key feature of the multifrontal method is that the frontal matrices and the generated elements are compressed and stored without zero rows and columns as small dense matrices.
- Integer arrays used to maintain a mapping of the local indices to the global indices of A and its factors.
- Symmetry allows only the lower triangular part of these matrices to be held.

Factorization Cholesky

Algorithm outlines the basic multifrontal method.

Algorithm (**Basic multifrontal Cholesky factorization**)

Input: *SPD matrix A and its elimination tree.*
Output: *Factor L such that $A = LL^T$.*

1: ***for*** $k = 1 : n$ ***do***
2: *Assemble the frontal matrix F_k using (40)* ▷ *Only the lower triangle is needed*
3: *Perform a partial Cholesky factorization of F_k using (37) to obtain column k of L and the generated element V_k*
4: ***end for***

We have the following observation.

Observation

Each generated element V_k is used only once to contribute to a frontal matrix $F_{parent(k)}$. Furthermore, the index list for the frontal matrix F_k is the set of row indices of the nonzeros in column k of the Cholesky factor L.

Factorization Cholesky

- In practical implementations, efficiency is improved by using the assembly tree because it allows more than one elimination to be performed at once.
- This is outlined here. kb is used to index the frontal matrix on the kb-th step ($1 \leq kb \leq nsup$).

Algorithm (**Multifrontal Cholesky factorization using the assembly tree)**

Input: *SPD matrix A and its assembly tree.*
Output: *Factor L such that $A = LL^T$.*

1: $nelim = 0$ ▷ *$nelim$ is the number of eliminations performed*
2: ***for*** $kb = 1 : nsup$ ***do*** ▷ *$nsup$ is the number of supernodes*
3: *Assemble the frontal matrix F_{kb}; let l be the number of fully summed variables in F_{kb}.*
4: *Perform a block partial Cholesky factorization of F_{kb} to obtain columns $nelim + 1$ to $nelim + l$ of L and the generated element V_{kb}*
5: $nelim = nelim + l$
6: ***end for***

Factorization Cholesky

As an example, consider the matrix and its assembly tree given above. The $nsup = 5$ supernodes are $\{1, 2\}, 3, 4, 5, \{6, 7, 8, 9\}$ and so variables 1 and 2 can be eliminated together on the first step. Assembling rows/columns 1 and 2 of the original matrix, the frontal matrix F_1 and generated element V_1 have the structure

$$F_1 = \begin{array}{c} \\ 1 \\ 2 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * & * \\ * & * & * & * \\ * & * & & \\ * & * & & \end{pmatrix} \end{array}, \quad V_1 = \begin{array}{c} \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cc} 8 & 9 \end{array} \\ \begin{pmatrix} f & f \\ f & f \end{pmatrix} \end{array},$$

where f denotes fill-in entries (only the lower triangular entries are stored in practice). Similarly,

$$F_2 = \begin{array}{c} \\ 3 \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 4 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \end{pmatrix} \end{array}, \quad V_2 = \begin{array}{c} \\ 4 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 8 \end{array} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{array}.$$

Factorization Cholesky

The frontal matrix F_3 and generated element V_3 are given by

$$F_3 = \begin{array}{c} \\ 4 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array} \Leftrightarrow V_2, \quad V_3 = \begin{array}{c} \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array}.$$

Then

$$F_4 = \begin{array}{c} \\ 5 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 5 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & \\ * & & * \end{pmatrix} \end{array}, \quad V_4 = \begin{array}{c} \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array},$$

and, finally, with $kb = 5$ we have

$$F_5 = \begin{array}{c} \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 6 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & & * & * \\ & * & & * \\ * & & * & \\ * & * & & * \end{pmatrix} \end{array} \Leftrightarrow V_4 \Leftrightarrow V_3 \Leftrightarrow V_1.$$

Factorization Cholesky

- An important implementation detail is how and where to store the generated elements.
- The partial factorization of F_{kb} at supernode kb can be performed once the partial factorizations at all the vertices belonging to the subtree of the assembly tree with root vertex kb are complete.
- If the vertices of the assembly tree are ordered using a depth-first search, the generated elements required at each stage are the most recently computed ones amongst those that have not yet been assembled.
- This makes it convenient to use a stack.

Factorization Cholesky

- The memory demands of the multifrontal method can be very large. Auxiliary storage can be used.
- The ordering of the children of a vertex in the assembly tree can significantly affect the required stack size.
- Tree and node parallelism.

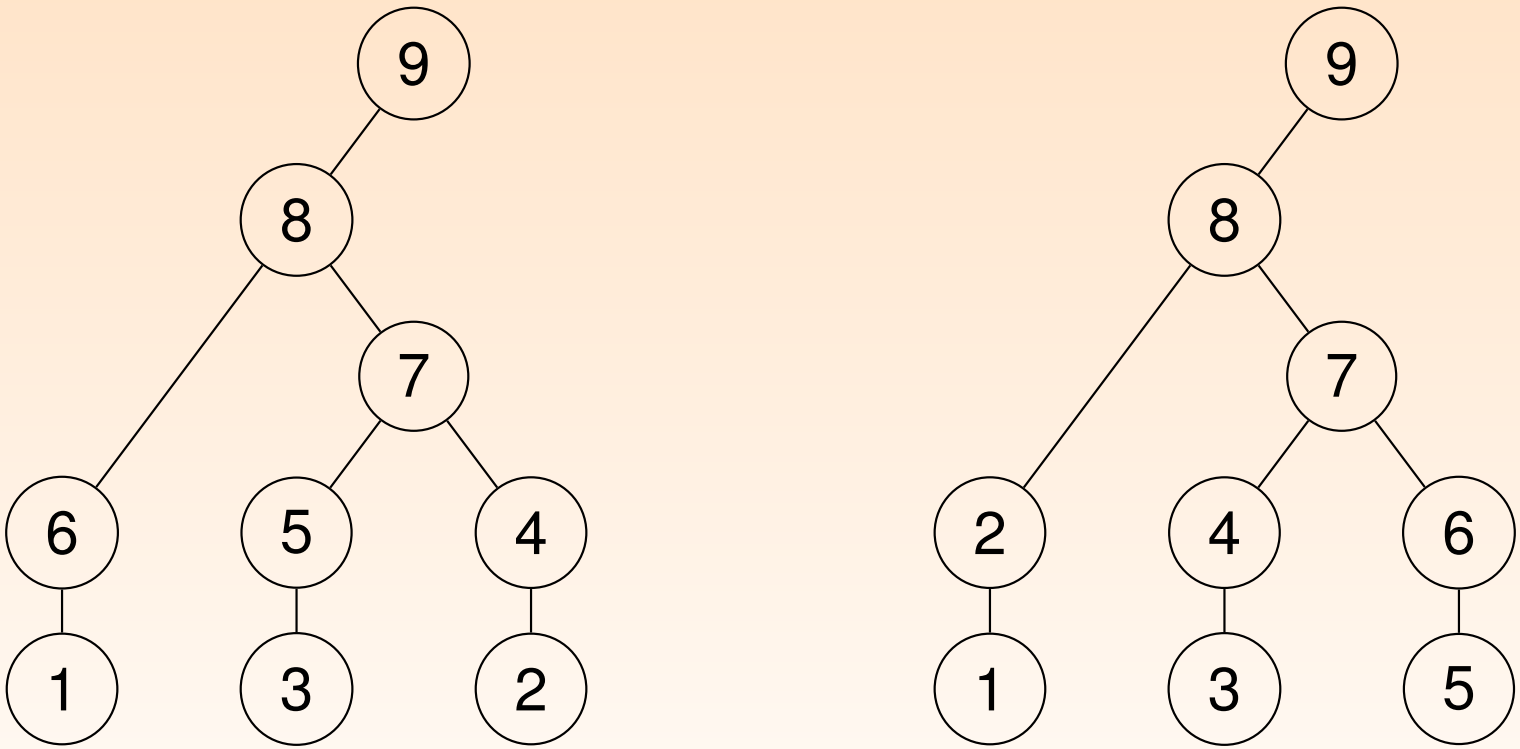
Factorization Cholesky

Multifrontal method: (example 1)

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

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

Multifrontal method: (example 1)



Multifrontal method: (example 1)

$$F_1 = \begin{array}{c} \\ 1 \\ 6 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 6 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * & * \\ * & & & \\ * & & & \\ * & & & \end{pmatrix} \end{array}, \quad V_1 = \begin{array}{c} \\ 6 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccc} 6 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * \\ * & * & f \\ * & f & * \end{pmatrix} \end{array}.$$

Here V_1 is dense and f denotes fill-in entries. Similarly, we have

$$F_2 = \begin{array}{c} \\ 2 \\ 4 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 4 & 7 \end{array} \\ \begin{pmatrix} * & * & * \\ * & & \\ * & & \end{pmatrix} \end{array}, \quad V_2 = \begin{array}{c} \\ 4 \\ 7 \end{array} \begin{array}{c} \begin{array}{cc} 4 & 7 \end{array} \\ \begin{pmatrix} * & f \\ f & * \end{pmatrix} \end{array}, \quad F_3 = \begin{array}{c} \\ 3 \\ 5 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccc} 3 & 5 & 8 \end{array} \\ \begin{pmatrix} * & * & * \\ * & & \\ * & & \end{pmatrix} \end{array}, V_3 = \begin{array}{c} \\ 5 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 5 & 8 \end{array} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{array}.$$

The sparsity pattern of the frontal matrix F_4 is then

$$F_4 = \begin{array}{c} \\ 4 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccc} 4 & 8 & 9 \end{array} \\ \begin{pmatrix} * & * & * \\ * & & \\ * & & \end{pmatrix} \end{array} \leftrightarrows V_2 = \begin{array}{c} \\ 4 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{cccc} 4 & 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & f & * & * \\ f & * & & \\ * & & & \\ * & & & \end{pmatrix} \end{array}, \quad V_4 = \begin{array}{c} \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccc} 7 & 8 & 9 \end{array} \\ \begin{pmatrix} * & f & * \\ f & * & f \\ * & f & * \end{pmatrix} \end{array},$$

and so on

Multifrontal method

Theorem

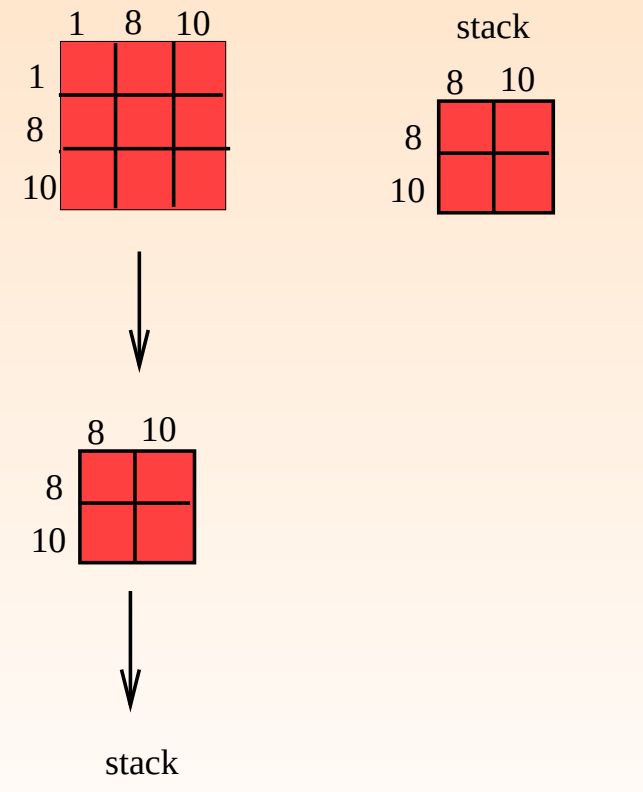
Let $\mathcal{T}$ be postordered. Assume each computed generated element (V_x) is pushed onto a stack. Then when constructing the (frontal matrix) F_j, the required generated elements are on the top of the stack. They can be directly popped from the stack and assembled into F_j.

Proof sketch: 1) Vertices of each subtree of the postordered $\mathcal{T}$ form an interval. 2) Denote c_l, $l = 1, \ldots, s$ children of j in $\mathcal{T}$. 3) Each c_l is the root of a subtree $\mathcal{T}(c_l)$. 4) Once the frontal matrix F_{c_l} for a leaf of $\mathcal{T}(c_l)$ is constructed, all its children have been processed and the generated V_{c_l} is pushed onto the stack. 5) That is, all subtrees $\mathcal{T}_{c_l}$, $l = 1, \ldots, s$ are fully assembled into the generated elements before F_j can be constructed. 6) If F_j is ready to be assembled (step j), the s generated V_{c_l}, $l = 1, \ldots, s$ are on the top of the stack.

Factorization Cholesky

Multifrontal method: (example 2)

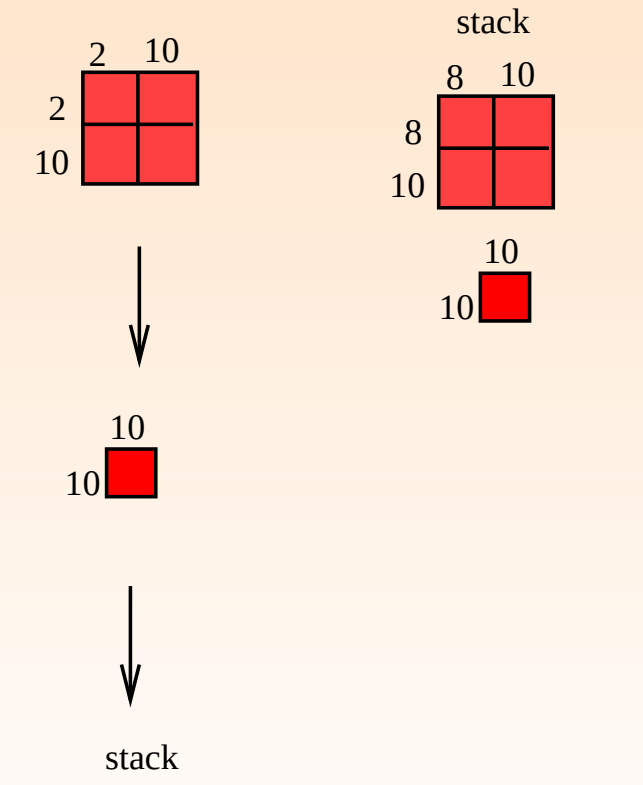
$$\begin{pmatrix}
* & & & & & & * & & * \\
 & * & & & & & & & * \\
 & & * & & & * & & & * \\
 & & & * & & * & & & \\
 & & & & * & * & & * & \\
 & & & & * & * & & f & * \\
 & & * & * & & & * & * & f \\
* & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Factorization Cholesky

Multifrontal method: (example 2)

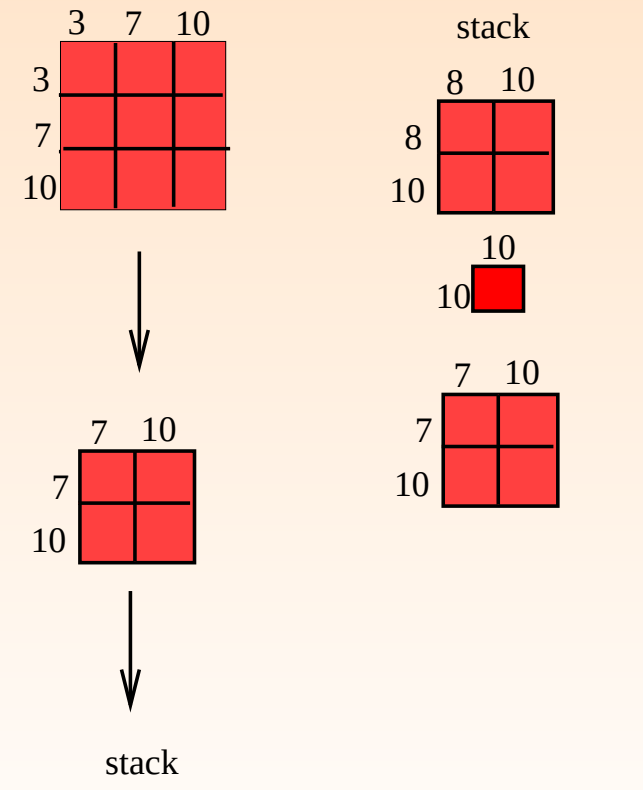
$$\begin{pmatrix}
* & & & & & & * & & * \\
 & * & & & & & & & * \\
 & & * & & & * & & & * \\
 & & & * & & * & & & \\
 & & & & * & * & & * & \\
 & & & & * & * & & f & * \\
 & & * & * & & & * & * & f \\
* & & & & & & * & & * \\
 & & & & * & f & * & * & f \\
* & * & * & & & * & f & * & f & * \\
\end{pmatrix}$$



Factorization Cholesky

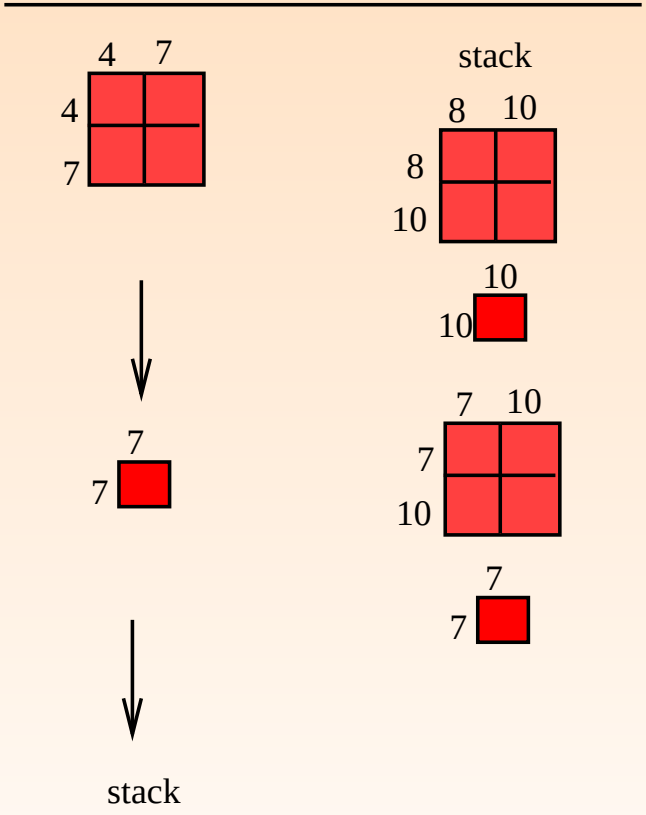
Multifrontal method: (example 2)

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



Direct methods: Multifrontal method

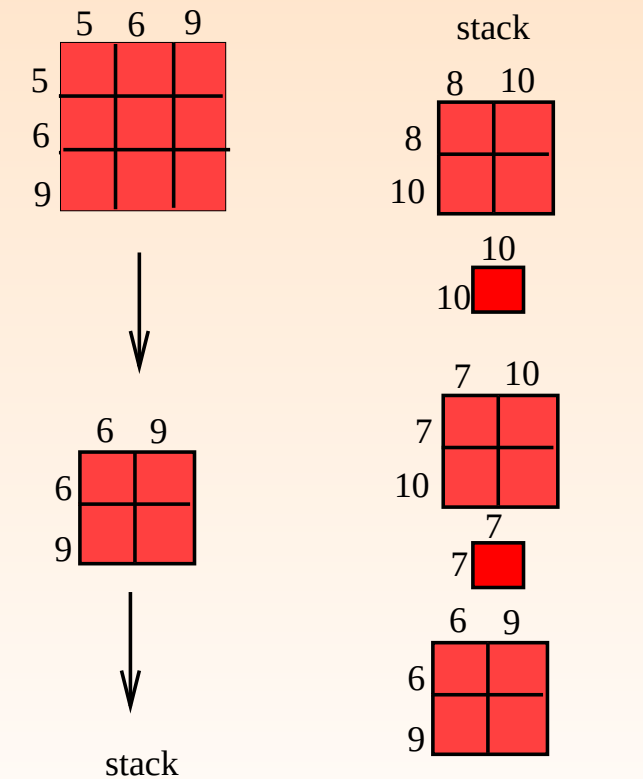
$$\begin{pmatrix}
* & & & & & & * & & * \\
 & * & & & & & & & * \\
 & & * & & & & * & & * \\
 & & & * & & & * & & \\
 & & & & * & * & & * & \\
 & & & & * & * & & f & * \\
 & & * & * & & & * & * & f \\
* & & & & & & * & & * \\
 & & & & * & f & * & * & f \\
* & * & * & & & * & f & * & f & * \\
\end{pmatrix}$$



Factorization Cholesky

Multifrontal method: (example 2)

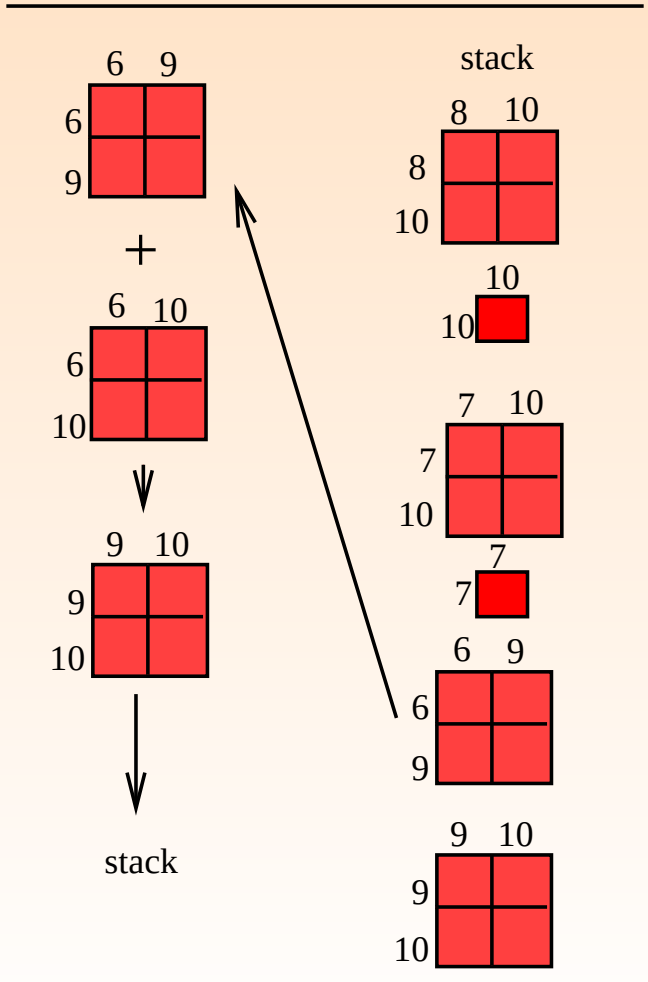
$$\begin{pmatrix}
* & & & & & & * & & * \\
 & * & & & & & & & * \\
 & & * & & & * & & & * \\
 & & & * & & * & & & \\
 & & & & * & * & & * & \\
 & & & & * & * & & f & * \\
 & & * & * & & & * & * & f \\
* & & & & & & * & & * \\
 & & & & * & f & * & & * & f \\
* & * & * & & & * & f & * & f & *
\end{pmatrix}$$



Factorization Cholesky

Multifrontal method: (example 2)

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



Multifrontal method: summary

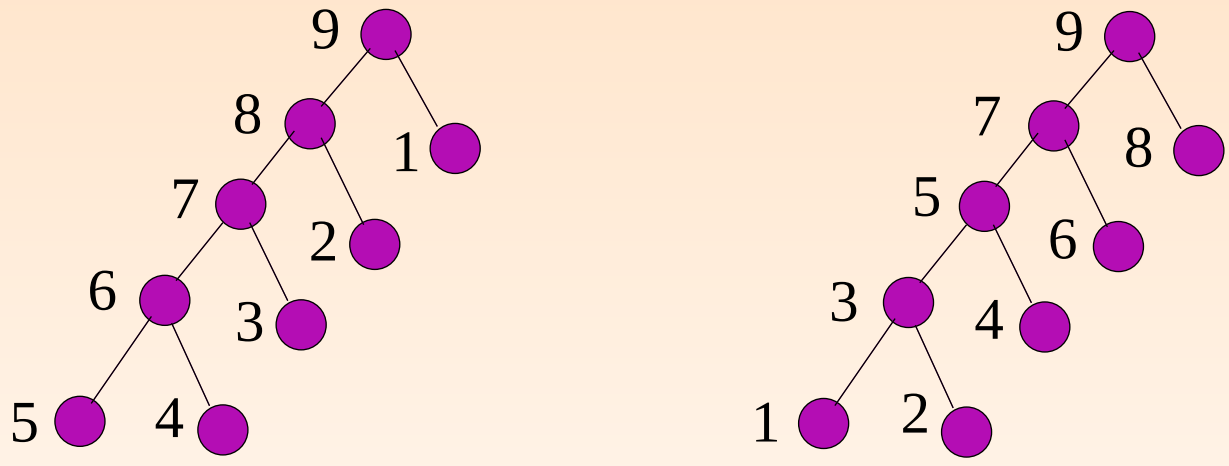
- Right-looking (submatrix) method
- Does not form the Schur complement directly. Instead, the updates are moved to a stack as dense matrices and used when needed.
- The processing order is based on the elimination tree
- We will see that in order to have the needed updates at the stack top, postordering is needed.
- Specific postorderings used to minimize the needed amount of memory.
- Now example, properties repeated once more later.

Multifrontal method: assumptions and properties

- We do need to have the entries from the stack readily available.
- $\rightarrow$ elimination tree should be postordered
- Arithmetic of dense matrices
- Connection with the frontal method (later) is relatively week.
- One of the most important methods for the sparse direct factorization.

Factorization Cholesky

Multifrontal method: postorderings memory issues



- First case: Maximum stack size may be $1 \times 1+2 \times 2+3 \times 3+4 \times 4$
- Second case: Maximum stack size may be 4×4
- Conclusion: Even postorderings can be very different with respect to algorithmic/architectural needs

Outline

Graphs and LU factorization

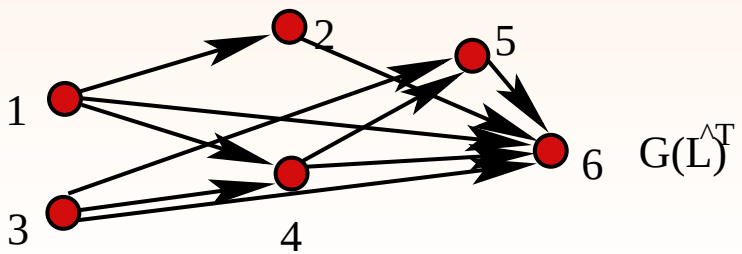
- Structural changes during a sparse Cholesky factorization described by graphs.
- In particular, using elimination tree.
- For LU: more possibilities to capture structure by graphs
- But, often factorizability has to be assumed: much harder combine pivoting and structural predictions.

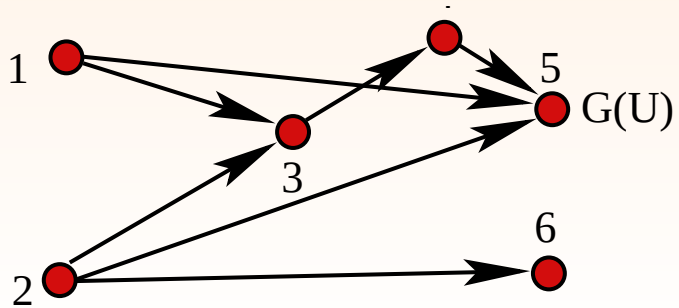
Sparse LU factorization of generally nonsymmetric matrices

LU factorization and DAGs

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} * & & * & & * & \\ * & * & & & * & * \\ & & * & * & & \\ * & & * & * & & \\ & & * & & * & \\ * & * & & & & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} * & & * & & * & \\ * & * & f & & * & * \\ & & * & * & & \\ * & & * & * & f & \\ & & * & f & * & \\ * & * & f & f & f & * \end{pmatrix} \end{array} \tag{41}$$

- **directed acyclic graphs** capture structure of the factors. We use $G(L^T)$ (L by columns) and $G(U)$ (U by rows).





LU factorization and DAGs

- The first graph model uses the elimination DAGs associated with L and U: see previous slide.
- Observation, generalizes predictions to the nonsymmetric case.

Observation

If $i > j$ and $u_{ji} \neq 0$ then the **column replication principle** *states*

$$\mathcal{S}\{L_{i:n,j}\} \subseteq \mathcal{S}\{L_{i:n,i}\},$$

that is, the pattern of column j of L (rows i to n) is replicated in the pattern of column i of L.

- *Analogously, if $i > j$ and $l_{ij} \neq 0$ then the* **row replication principle** *states*

$$\mathcal{S}\{U_{j,i:n}\} \subseteq \mathcal{S}\{U_{i,i:n}\},$$

that is, the pattern of row j of U (columns i to n) is replicated in the pattern of row i of U.

Sparse LU

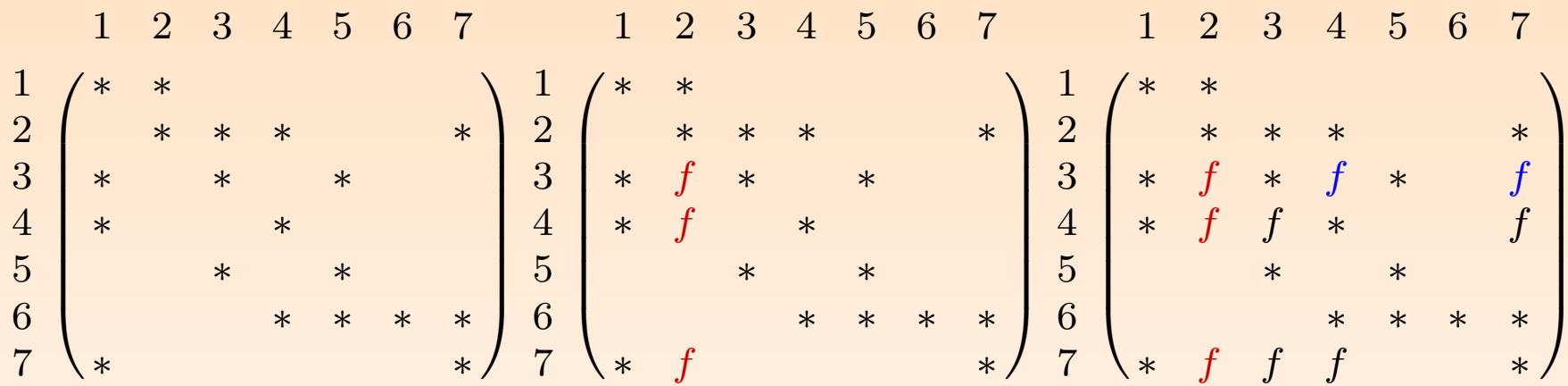


Figure: *An illustration of the column and row replication principles of sparse LU factorizations. Left: the matrix A. Centre: showing column replication. Right: also row replication. Filled entries not involved in the demonstration and resulting from subsequent steps of the factorization are denoted in black.*

Basic sparse LU factorization

- Assumed that A is factorizable so that pivoting is not needed (we will remind this only sometimes).

Algorithm (**Basic sparse LU factorization**)

Input: *Nonsymmetric and factorizable matrix* $A = L_A + D_A + U_A$.
Output: *LU factorization* $A = LU$.

1: $L = I + L_A$ ▷ *Strictly lower triangular part of* A
2: $U = D_A + U_A$ ▷ *Diagonal plus strictly upper triangular part of* A
3: ***for*** $k = 1 : n - 1$ ***do***
4: ***for*** $i \in \{i > k \,|\, l_{ik} \neq 0\}$ ***do***
5: $l_{ik} = l_{ik}/u_{kk}$
6: $U_{i,i:n} = U_{i,i:n} - U_{k,i:n} l_{ik}$ ▷ *Update row* i *of* U
7: ***end for***
8: ***for*** $j \in \{j > k \,|\, u_{kj} \neq 0\}$ ***do***
9: $L_{j+1:n,j} = L_{j+1:n,j} - L_{j+1:n,k} u_{kj}$ ▷ *Update column* j *of* L
10: ***end for***
11: ***end for***

Recursive replications

The following theorem formulates the recursive column replication and the replication of nonzeros along rows of L using directed paths in $\mathcal{G}(U)$.

Theorem

Assume that for some $k < j$ there is a directed path $k \overset{\mathcal{G}(U)}{\Longrightarrow} j$. Then

$$\mathcal{S}\{L_{j:n,k}\} \subseteq \mathcal{S}\{L_{j:n,j}\}. \tag{42}$$

Moreover, if $l_{ik} \neq 0$ for some $i > j$ then $l_{is} \neq 0$ for all vertices s on this path.

- An analogous result holds for the rows of U and directed paths in $\mathcal{G}(L^T)$.

Sparse LU

Generalizing the necessary and sufficient condition for fill-in

Theorem

If $a_{ij} = 0$ and $i > j$ then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $a_{ik} \neq 0$ and there is a directed path $k \overset{\mathcal{G}(U)}{\Longrightarrow} j$.

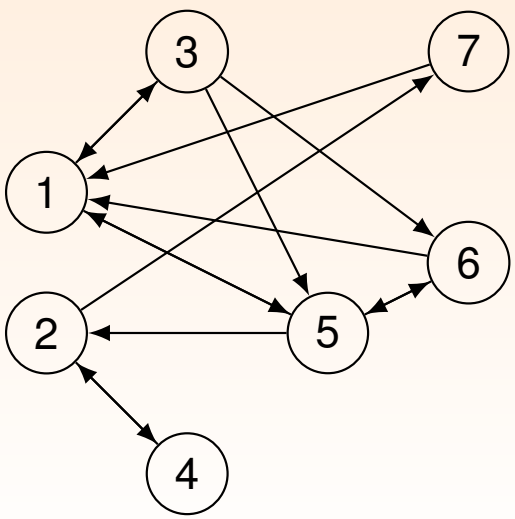
Theorem

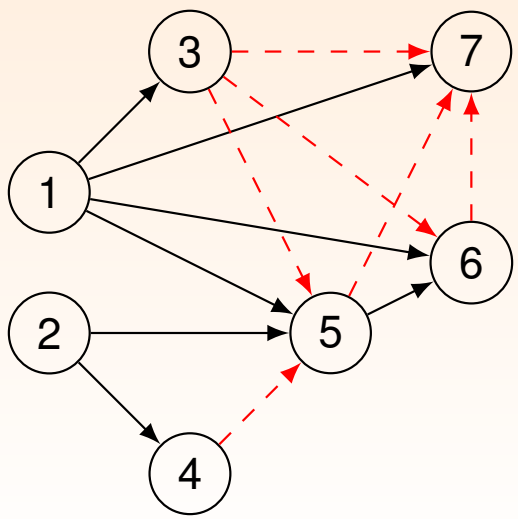
If $a_{ij} = 0$ and $i < j$ then there is a filled entry $u_{ij} \neq 0$ if and only if there exists $k < i$ such that $a_{kj} \neq 0$ and there is a directed path $k \overset{\mathcal{G}(L^T)}{\Longrightarrow} i$.

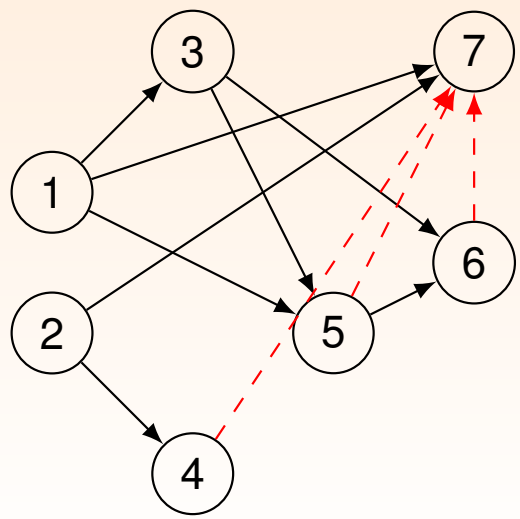
Sparse LU

Consider the path $1 \to 3 \to 5 \to 6$ in $\mathcal{G}(U)$. Its existence implies the fill-in in L, first in column 3, then in columns 5 and 6. Similarly, the path $2 \to 4 \to 5 \to 6$ in $\mathcal{G}(L^T) \Rightarrow$ fill-in at $(4,7)$, $(5,7)$ and $(6,7)$ in U.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} * & & * & & * & & \\ & * & & * & & & * \\ * & & * & & * & * & \\ & * & & * & & & \\ * & * & & & * & * & \\ * & & & & * & * & \\ * & & & & & & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} * & & * & & * & & \\ & * & & * & & & * \\ * & & * & & * & * & \\ & * & & * & & & f \\ * & * & f & f & * & * & f \\ * & & f & & * & * & f \\ * & & f & & f & f & * \end{pmatrix} \end{array}$$





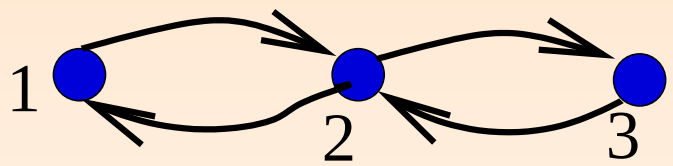


Transitive reduction

- To employ $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$ in efficient algorithms, they need to be simplified. Transitive reductions are sparser and preserve reachability within the graphs.
- A subgraph $\mathcal{G}^0 = (\mathcal{V}, \mathcal{E}^0)$ is a **transitive reduction** of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ if the following conditions hold:
  - $(T1)$ there is a path from vertex i to vertex j in $\mathcal{G}$ if and only if there is a path from i to j in $\mathcal{G}^0$ (reachability condition), and
  - $(T2)$ there is no subgraph with vertex set $\mathcal{V}$ that satisfies $(T1)$ and has fewer edges (minimality condition).
- A transitive reduction is unique for a DAG, as shown in the following theorem and illustrated below.

Transitive reduction

- Transitive reduction is the loop interconnecting vertices $1, 2$ and 3.

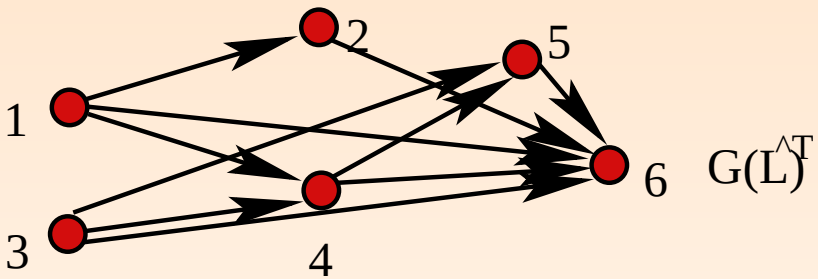


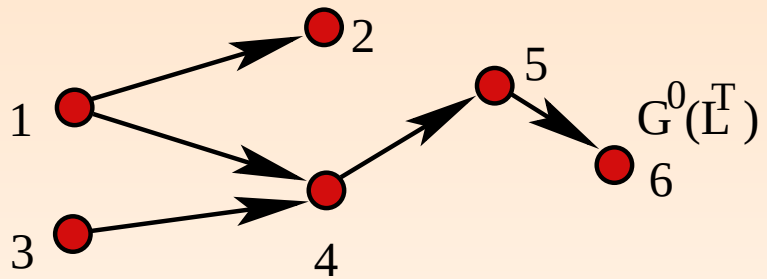
Theorem

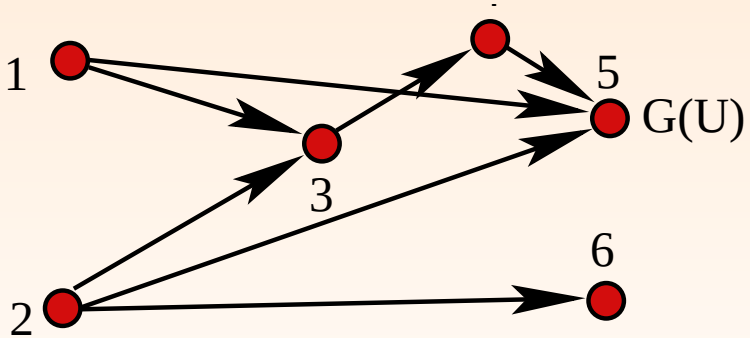
Transitive reduction of a **directed acyclic graph** *is unique and is a subgraph of* G*.*

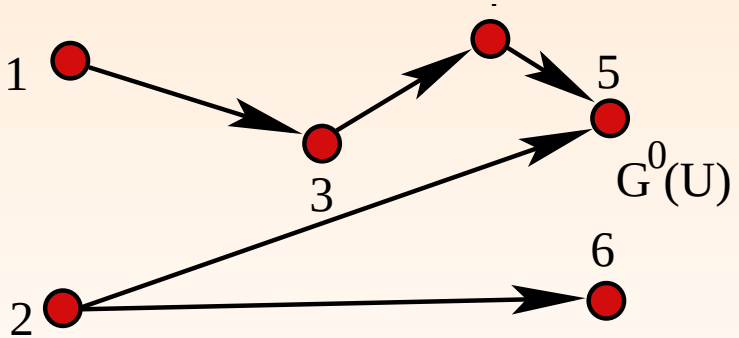
Sparse LU factorization of generally nonsymmetric matrices

Transitive reduction









Sparse LU

Theorem

Let $\mathcal{G}$ be a DAG. The transitive reduction $\mathcal{G}^0$ of $\mathcal{G}$ is unique and is the subgraph that has an edge for every path in $\mathcal{G}$ and has no proper subgraph with this property.

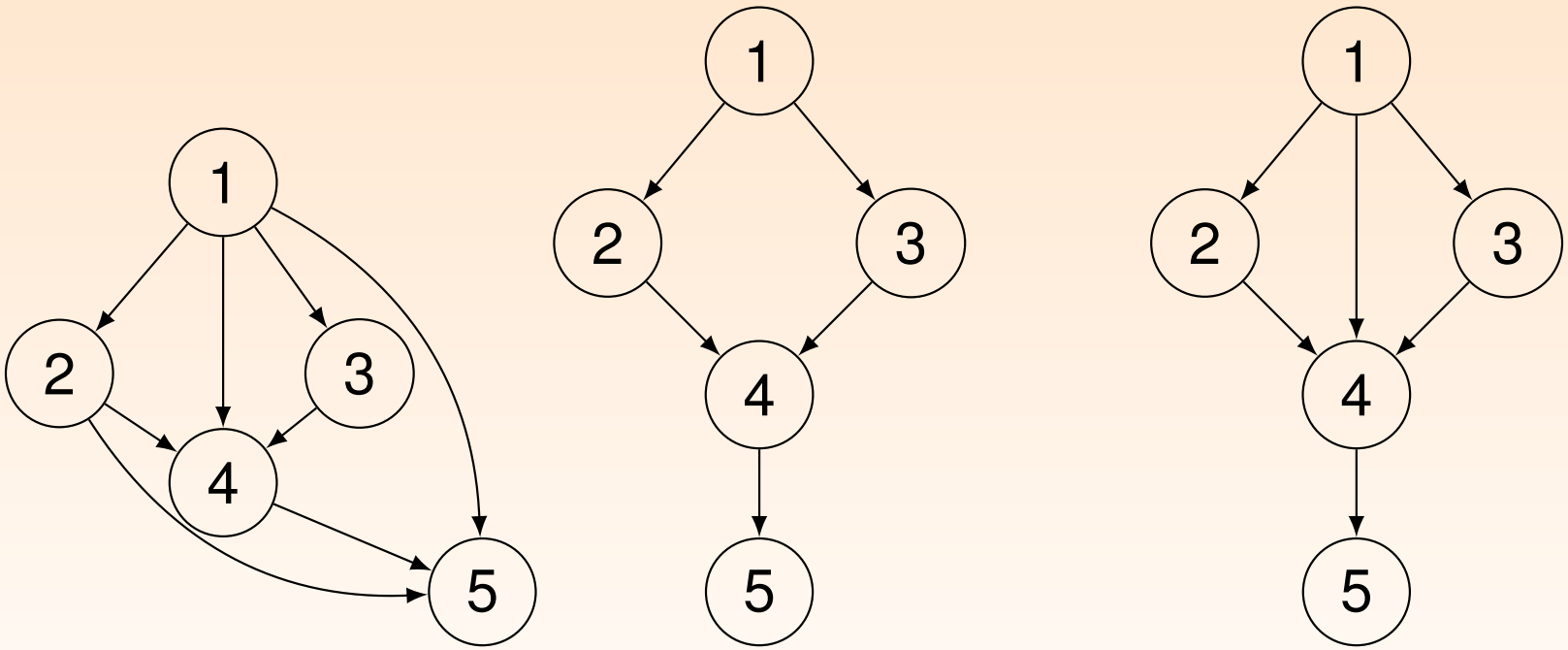


Figure: *Example to show the transitive reduction of a DAG. $\mathcal{G}$ is on the left, its transitive reduction $\mathcal{G}^0$ is in the centre, and one possible $\mathcal{G}'$ that is equireachable with $\mathcal{G}$ is on the right.*

Transitive reduction

If $\mathcal{S}\{A\}$ is symmetric, the role of the transitive reduction is played by the elimination tree.

Theorem

If A is symmetrically structured then the transitive reduction of the DAG $\mathcal{G}(L^T)$ (= $\mathcal{G}(U)$) is the elimination tree $\mathcal{T}(A)$.

Transitive reduction

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
* & & * & & * & * \\
& * & & * & * & \\
* & & * & & * & f \\
& * & & * & f & * \\
* & * & * & f & * & f \\
* & & f & * & f & *
\end{pmatrix}
\end{array}
$$

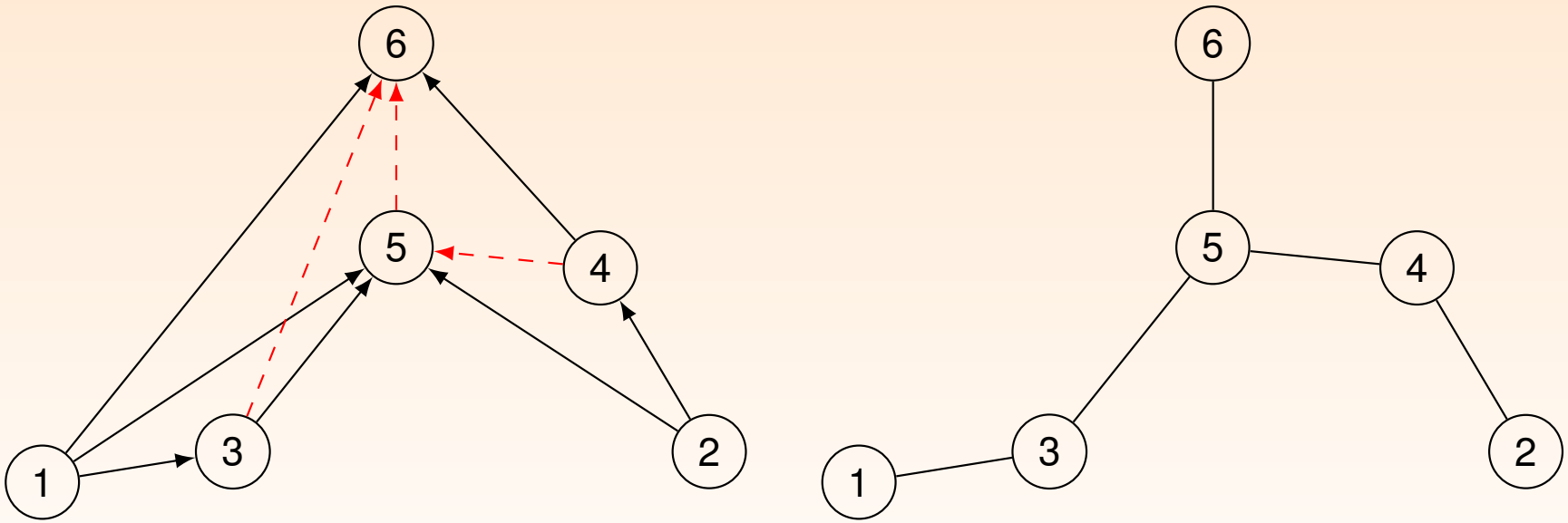


Figure: *The sparsity patterns of $L + U$ of a symmetrically structured A, the DAG $\mathcal{G}(L^T)$ (left) and the elimination tree $\mathcal{T}(A)$ (right). Straightforward to see that $\mathcal{T}(A)$ is the transitive reduction of $\mathcal{G}(L^T)$.*

Sparse LU

Transitive reduction may be expensive to obtain

- Obtaining the exact transitive reduction of a DAG can be expensive. Instead, approximate reductions that drop the minimality condition may be computed.
- A directed graph $\mathcal{G}'$ with the same vertex set as $\mathcal{G}$ that satisfies condition $(T1)$ is said to be **equireachable** with $\mathcal{G}$.
- This is something in between the DAG and transitive reduction. Of course, due to the reachability, a lot of theoretical results are satisfied.

Theorem

Assume $\mathcal{G}'$ is equireachable with $\mathcal{G}(U)$ and for some $k < j$ there is a directed path $k \overset{\mathcal{G}'}{\Longrightarrow} j$. Then the replication theorem can use the reduced DAGs. Moreover, if $l_{ik} \neq 0$ for some $i > j$ then $l_{is} \neq 0$ for all vertices s on the directed path.

Equireachability and efficiency

- Equireachability enables sparse triangular linear systems to be solved more efficiently being sparser.
- The necessary and sufficient conditions for the fill-in from above:

Theorem

If $a_{ij} = 0$ and $i > j$ then there is a filled entry $l_{ij} \neq 0$ if and only if there exists $k < j$ such that $a_{ik} \neq 0$ and a directed path $k \overset{\mathcal{G}'(U)}{\Longrightarrow} j$, where $\mathcal{G}'(U)$ is equireachable with $\mathcal{G}(U)$.

Theorem

If $a_{ij} = 0$ and $i < j$ then there is a filled entry $u_{ij} \neq 0$ if and only if there exists $k < i$ such that $a_{kj} \neq 0$ and a directed path $k \overset{\mathcal{G}'(L^T)}{\Longrightarrow} i$, where $\mathcal{G}'(L^T)$ is equireachable with $\mathcal{G}(L^T)$.

Equireachability: example

Figure 36 depicts $\mathcal{G}(U)$ and $\mathcal{G}'(U)$ for the matrix in Figure above.

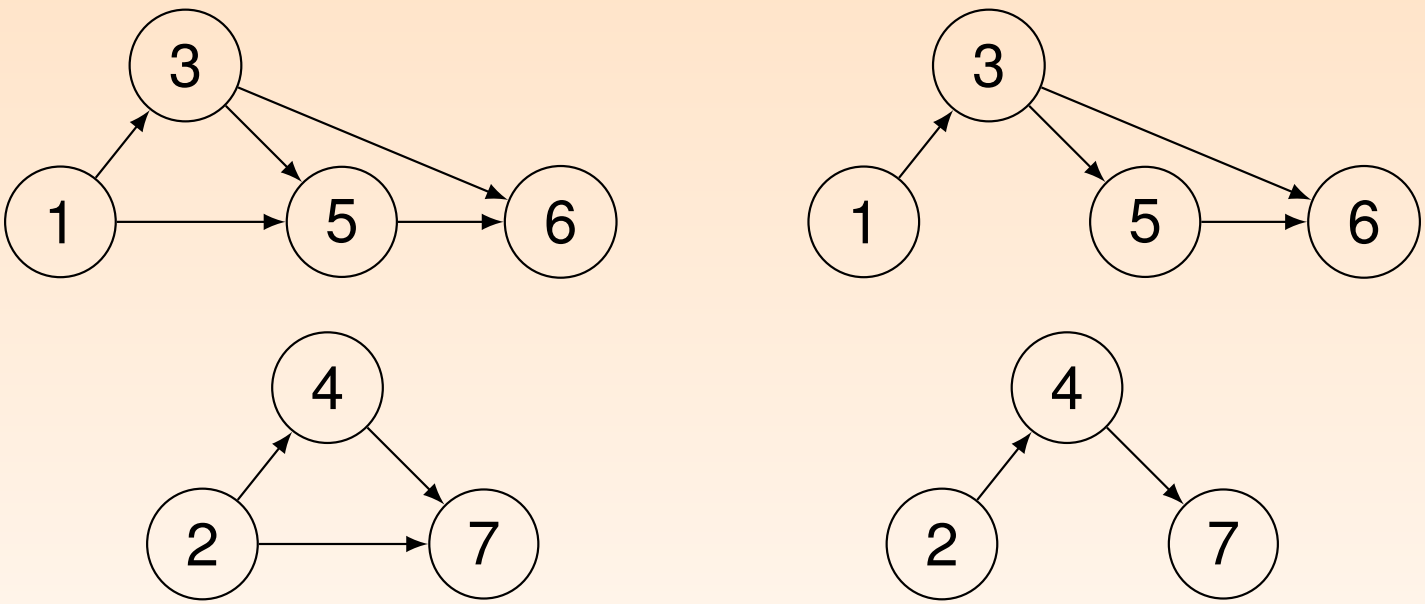


Figure: *The DAG $\mathcal{G}(U)$ for the matrix from Figure* **??** *(left) and $\mathcal{G}'(U)$ which is equireachable with $\mathcal{G}(U)$ (right).*

Column sparsity patterns (for L)

Standard description of the sparsity patterns of the columns of L can be obtained from the Schur complement as follows:

$$\mathcal{S}\{L_{j:n,j}\} = \mathcal{S}\{A_{j:n,j}\} \bigcup_{k<j, u_{kj}\neq 0} \mathcal{S}\{L_{j:n,k}\}, \quad 1 \leq j \leq n.$$

Theorem on the next slide implies that not all the terms in this union are needed to obtain $\mathcal{S}\{L_{j:n,j}\}$.

Column sparsity patterns (for L)

Theorem

If $\mathcal{G}'(U)$ is equireachable with $\mathcal{G}(U)$ then

$$\mathcal{S}\{L_{j:n,j}\} = \mathcal{S}\{A_{j:n,j}\} \bigcup_{(k \to j) \in \mathcal{E}(\mathcal{G}'(U))} \mathcal{S}\{L_{j:n,k}\}, \quad 1 \leq j \leq n. \tag{43}$$

Proof.

Consider an edge $(k \to j)$ in $\mathcal{G}(U)$ but not in $\mathcal{G}'(U)$. Repeatedly applying replication results along the directed path $k \xrightarrow{\mathcal{G}'(U)} j$, we see that $L_{j:n,k}$ is contained in the right-hand side of the predicted structure and therefore $\mathcal{S}\{L_{j:n,j}\}$ is contained in the right-hand side of this structure as well. Because the right-hand side of the formula is trivially contained in the left-hand side, the result follows. □ □

Row sparsity patterns (for U)

An analogous result holds for the rows of U.

Theorem

If $\mathcal{G}'(L)$ is equireachable with $\mathcal{G}(L)$ then

$$\mathcal{S}\{U_{i,i:n}\} = \mathcal{S}\{A_{i,i:n}\} \bigcup_{(k \to i) \in \mathcal{E}(\mathcal{G}'(L^T))} \mathcal{S}\{U_{k,i:n}\}, \quad 1 \leq i \leq n.$$

As an example, consider the matrix above. Because $(3 \to 5)$ is the only edge of $\mathcal{G}'(U)$ in the union on the right-hand side of (43), $\mathcal{S}\{L_{5:7,5}\}$ is given by

$$\mathcal{S}\{L_{5:7,5}\} = \mathcal{S}\{A_{5:7,5}\} \cup \mathcal{S}\{L_{5:7,3}\}.$$

We can see this from the graph $\mathcal{G}'(U)$ in Figure that demonstrates equireachability (top right).

Factorization by bordering

- Factorization by bordering can be used to obtain $\mathcal{S}\{L\}$ by rows and $\mathcal{S}\{U\}$ by columns.
- Assume the sparsity patterns of the first $k-1$ rows of L and the first $k-1$ columns of U ($1 < k \leq n$) have been computed.
- At step k, the matrix $A_{1:k,1:k}$ is

$$\begin{pmatrix} A_{1:k-1,1:k-1} & A_{1:k-1,k} \\ A_{k,1:k-1} & a_{kk} \end{pmatrix} = \begin{pmatrix} L_{1:k-1,1:k-1} & 0 \\ L_{k,1:k-1} & 1 \end{pmatrix} \begin{pmatrix} U_{1:k-1,1:k-1} & U_{1:k-1,k} \\ 0 & u_{kk} \end{pmatrix} \tag{44}$$

- Equating terms for the $(2,1)$ block, row k of L satisfies

$$L_{k,1:k-1} U_{1:k-1,1:k-1} = A_{k,1:k-1},$$

or, equivalently, if y denotes the off-diagonal part of the column k of L^T then it is the solution of the lower triangular system

$$U^T_{1:k-1,1:k-1} y = A^T_{k,1:k-1}.$$

Factorization by bordering

- The sparsity pattern of y is the set of all vertices reachable in the DAG $\mathcal{G}(U_{1:k-1,1:k-1})$ (or in a graph that is equireachable with it) from the nonzeros in $A_{k,1:k-1}$.
- Similarly, equating terms in (44) for the $(1,2)$ block, column k of U satisfies

$$L_{1:k-1,1:k-1}U_{1:k-1,k} = A_{1:k-1,k}.$$

- Again, its sparsity pattern can be determined by searching the DAG $\mathcal{G}(L^T_{1:k-1,1:k-1})$.
- The diagonal entry u_{kk} is then computed as $a_{kk} - L_{k,1:k-1}U_{1:k-1,k}$.
- Determining the sparsity patterns of L and U and computing their numerical values is coupled: computation of the factors needs be mutually interleaved because computing part of one requires information from a part of the other.

Pruning of the elimination DAGs

- The matrix in the centre is the same as the one on the left except that the entries in positions $(4, 6)$ and $(6, 4)$ have been removed (that is, pruned).
- Both matrices have the same sets of reachable vertices in $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$. This suggests how to find $\mathcal{G}'(L^T)$ and $\mathcal{G}'(U)$ that are equireachable with $\mathcal{G}(L^T)$ and $\mathcal{G}(U)$, respectively.

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
* & & * & & & * \\
* & * & & & & \\
& & * & & & * \\
& * & & * & * & * \\
& & & * & * & * \\
* & & & * & * & *
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
* & & * & & & * \\
* & * & & & & \\
& & * & & & * \\
& * & & * & * & \\
& & & * & * & * \\
* & & & & * & *
\end{pmatrix}
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\
\begin{pmatrix}
* & & * & & & \\
* & * & & & & \\
& & * & & & * \\
& * & & * & * & \\
& & & * & * & * \\
& & & & * & *
\end{pmatrix}
\end{array}
$$

Figure: *An example of symmetric pruning. On the left is $\mathcal{S}\{L + U\}$. In the centre is the reduced sparsity pattern obtained by symmetric pruning. On the right is the reduced sparsity pattern that results from symmetric path pruning.*

Pruning of the elimination DAGs

Theorem

If for some $j < s$ both $l_{sj} \neq 0$ and $u_{js} \neq 0$, then there are no edges $(j \to k)$ with $k > s$ in the transitive reductions of $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$.

Proof.

Let $(j \to k)$ be an edge of $\mathcal{G}(U)$, that is, $u_{jk} \neq 0$. Because $l_{sj} \neq 0$ and $u_{jk} \neq 0$ implies that $u_{sk} \neq 0$, there is a path $j \to s \to k$ in $\mathcal{G}(U)$ and the edge $(j \to k)$ does not belong to the transitive reduction of $\mathcal{G}(U)$. The result for $\mathcal{G}(L^T)$ can be seen analogously. □

The theorem implies that if for some $s > 1$ there are edges

$$j \xrightarrow{\mathcal{G}(L^T)} s \quad \text{and} \quad j \xrightarrow{\mathcal{G}(U)} s,$$

then all edges $(j \to k)$ in $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$ with $k > s$ can be pruned. The resulting DAGs $\mathcal{G}'(U)$ and $\mathcal{G}'(L^T)$ have fewer edges and are equireachable with $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$, respectively. The removal of redundant edges based on the Theorem is called **symmetric pruning**.

Other ways of pruning of the elimination DAGs

- If for some $s > 1$ there are paths

$$j \xrightarrow{\mathcal{G}(L^T)} s \quad \text{and} \quad j \xrightarrow{\mathcal{G}(U)} s,$$

  then for all $k > s$ **symmetric path pruning** removes the edges $(j \to k)$ from $\mathcal{G}(U)$ and $\mathcal{G}(L^T)$.
- Consider again previous Figure. In the centre: the sparsity pattern after symmetric pruning. On the right: the reduced sparsity pattern that results from symmetric path pruning. The edge $(1 \to 6)$ is not required in $\mathcal{G}'(L^T)$ or $\mathcal{G}'(U)$ because there are paths

$$1 \xrightarrow{\mathcal{G}(L^T)} 2 \xrightarrow{\mathcal{G}(L^T)} 4 \xrightarrow{\mathcal{G}(L^T)} 5 \xrightarrow{\mathcal{G}(L^T)} 6 \quad \text{and} \quad 1 \xrightarrow{\mathcal{G}(U)} 3 \xrightarrow{\mathcal{G}(U)} 6.$$

Another graph model: nonsymmetric elimination tree

- The elimination DAGs $\mathcal{G}(L)$ and $\mathcal{G}(U)$ can be combined into a single structure called the **nonsymmetric elimination tree** in which edges are replaced by paths.
- This can be advantageous because it is more compact.
- If $\mathcal{S}\{A\}$ is symmetric then its elimination tree is defined in terms of the mapping

$$parent(j) = \min\{i \mid i > j \text{ and } l_{ij} \neq 0\}.$$

  The condition $l_{ij} \neq 0$ is equivalent to $i \xrightarrow{\mathcal{G}(L)} j \xrightarrow{\mathcal{G}(L^T)} i$.
- In the nonsymmetric case, the definition can be generalized using directed paths

$$parent(j) = \min\{i \mid i > j \text{ and } i \xRightarrow{\mathcal{G}(L)} j \xRightarrow{\mathcal{G}(U)} i\}. \qquad (45)$$

Example of the next slide

- Vertices 6, 8 and 10 are the only ones with cycles of the form

$$i \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} i.$$

- Namely,

$$6 \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} 5 \xrightarrow{\mathcal{G}(U)} 6, \quad 8 \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} 8$$

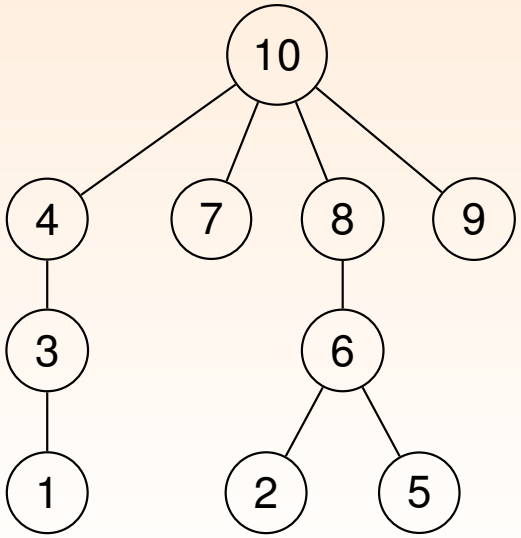
  and

$$10 \xrightarrow{\mathcal{G}(L)} 6 \xrightarrow{\mathcal{G}(L)} 2 \xrightarrow{\mathcal{G}(U)} 10.$$

- In this example, $parent(2) = 6$.

Sparse LU

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}\begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \begin{pmatrix} * & & * & & & & & & & \\ * & * & & & * & & & * & & * \\ * & & * & * & & & & & & \\ & & * & * & & & & & * & * \\ & & & * & * & * & & * & & \\ & * & & & & * & & & & \\ & & & * & & & * & & & \\ * & * & & & & & & * & & \\ & & & & & & & & * & * \\ & & & & * & & * & & & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}\begin{array}{c} \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\ \begin{pmatrix} * & & * & & & & & & & \\ * & * & f & & * & & & * & & * \\ * & & * & * & & & & & & \\ & & * & * & & & & & * & * \\ & & & * & * & * & & * & f & f \\ & * & f & f & f & * & & f & f & f \\ & & & * & & & * & & f & f \\ * & * & f & f & f & f & & * & f & f \\ & & & & & & & & * & * \\ & & & & * & f & * & f & f & * \end{pmatrix} \end{array}$$



Locality: way to find the nonsymmetric elimination tree

- Theorem can be regarded as a generalization of corollary on the standard symmetric elimination tree.

Theorem

Let A be a nonsymmetric matrix. $i = parent(j)$ if and only if $i > j$ and i is the smallest vertex that belongs to the same strong component of $\mathcal{G}(A_{1:i,1:i})$ as vertex j.

- The result is employed in Algorithm below to find the elimination tree.
- The complexity of finding the strong components of a digraph with m edges and n vertices is $O(n+m)$ time. Hence, the complexity of Algorithm is $O(nz(A)\, n)$. More sophisticated approaches with complexity $O(nz(A) \log n)$ exist.

Locality: way to find the nonsymmetric elimination tree

Algorithm (**Basic computation of the elimination tree for nonsymmetric A**)

Input: *Digraph $\mathcal{G}(A)$.*
Output: *The elimination tree given by the mapping $parent$.*

1: $parent(1:n) = 0$
2: ***for*** $i = 1:n$ ***do***
3: *Find the vertex set $\mathcal{V}_C$ of the strong component of $\mathcal{G}(A_{1:i,1:i})$ that contains i*
4: ***for*** $j \in \mathcal{V}_C \setminus \{i\}$ ***do***
5: ***if*** $parent(j) = 0$ ***then***
6: $parent(j) = i$
7: ***end if***
8: ***end for***
9: $parent(i) = 0$
10: ***end for***

Illustration using the matrix from above

- The main loop sets the first nonzero value in the array $parent$ when $i = 3$ because this is the first i for which the set $\mathcal{V}_C \setminus \{i\}$ is non empty; it is equal to $\{1\}$ and thus $parent(1) = i = 3$.
- For $i = 4$, the vertex set $\{1, 3, 4\}$ forms a strong component of $\mathcal{G}(A_{1:4,1:4})$ and so $parent(3) = 4$.
- For $i = 5$, the single vertex $\{5\}$ is a strong component of $\mathcal{G}(A_{1:5,1:5})$ and, therefore, 5 is not a parent of any other vertex (it is a leaf vertex).
- $\mathcal{G}(A_{1:6,1:6})$ has two strong components with vertex sets $\{1, 3, 4\}$ and $\{2, 5, 6\}$. $i = 6$ belongs to the second of these and thus the algorithm sets $parent(j) = i = 6$ for $j = 2$ and 5.

Another graph model: column elimination tree

- An attractive idea for constructing $\mathcal{S}\{L+U\}$ and subsequently computing the LU factorization is based on using the **column elimination tree** $\mathcal{T}(A^TA)$.

Theorem

Assume all the diagonal entries of A are nonzero and let $\widehat{L}\widehat{L}^T$ be the Cholesky factorization of A^TA. Then for any row permutation matrix P such that $PA = LU$

$$\mathcal{S}\{L+U\} \subseteq \mathcal{S}\{\widehat{L}+\widehat{L}^T\}.$$

Sparse LU

- Holds for any row permutation matrix P applied to A. This allows partial pivoting.
- The following result states that $\mathcal{T}(A^T A)$ represents the potential dependencies among the columns in an LU factorization.
- For strong Hall matrices no tighter prediction is possible from the sparsity structure of A.

Theorem

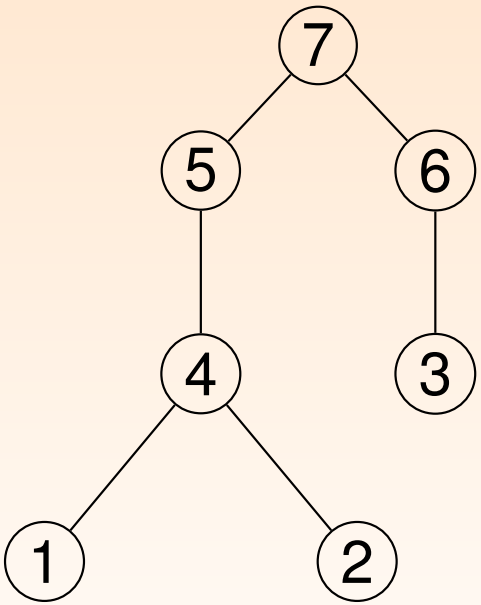
If $PA = LU$ is any factorization of A with partial pivoting then the following hold.

1. *If vertex i is an ancestor of vertex j in $\mathcal{T}(A^T A)$ then $i \geq j$.*
2. *If $l_{ij} \neq 0$ then vertex i is an ancestor of vertex j in $\mathcal{T}(A^T A)$.*
3. *If $u_{ij} \neq 0$ then vertex j is an ancestor of vertex i in $\mathcal{T}(A^T A)$.*
4. *Suppose in addition that A is a strong Hall matrix. If $l = parent(k)$ in $\mathcal{T}(A^T A)$, then there are values of the nonzero entries of A for which $u_{kl} \neq 0$.*

Sparse LU

Figures below illustrate the differences in the sparsity patterns of A and $A^T A$ and of their factors; the corresponding elimination trees are also given.

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\left(\begin{array}{ccccccc}
* & & & * & & & \\
* & * & f & * & & & \\
 & & * & & & * & \\
* & * & & * & f & & * \\
 & * & * & f & * & * & * \\
 & & * & & & * & \\
* & & & * & * & f & *
\end{array}\right)
\end{array}
$$



Sparse LU

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}
\begin{array}{c}
\begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{pmatrix}
* & * & & * & * & & * \\
* & * & * & * & * & * & * \\
 & * & * & f & * & * & * \\
* & * & f & * & * & f & * \\
* & * & * & * & * & * & * \\
 & * & * & f & * & * & * \\
* & * & * & * & * & * & *
\end{pmatrix}
\end{array}
$$

Figure: *Both figures: the sparsity patterns of A and $L+U$ (top) and of A^TA and $\widehat{L}+\widehat{L}^T$, where $A^TA=\widehat{L}\widehat{L}^T$ (bottom). Filled entries are denoted by f. The corresponding elimination trees are also given.*

The column elimination tree

- A potential problem with the column elimination tree:
- $\mathcal{S}\{A^T A\}$ can have significantly more entries than $\mathcal{S}\{L+U\}$. An extreme example is when A has one or more dense rows because $A^T A$ is then fully dense.

Supernodes and LU

- Supernodes group together columns of the factors with the same nonzero structure, allowing them to be treated as a dense submatrix for storage and computation.
- For nonsymmetric matrices, supernodes are harder to characterize.
- The need to incorporate pivoting means it may not be possible to predict the sparsity structures of the factors before the numerical factorization and they must be identified on the fly.
- More ways to define supernodes.
- Cholesky solver: fundamental supernodes are made contiguous by symmetrically permuting the matrix according to a postordering of its elimination tree; this does not change the sparsity of the Cholesky factor.
- For nonsymmetric A, before the numerical factorization, $\mathcal{T}(A^T A)$ can be constructed and the columns of A then permuted according to its postordering to bring together supernodes.

Sparse LU

Theorem

Let A have column elimination tree $\mathcal{T}(A^TA)$. Let p be a permutation vector such that if p_i is an ancestor of p_j in $\mathcal{T}(A^TA)$ then $i \geq j$. Let P be the permutation matrix corresponding to p and let $\widehat{A} = PAP^T$. Then $\mathcal{T}(\widehat{A}^T\widehat{A})$ is isomorphic to $\mathcal{T}(A^TA)$; in particular, relabelling each vertex i of $\mathcal{T}(\widehat{A}^T\widehat{A})$ as p_i yields $\mathcal{T}(A^TA)$. If, in addition, $\widehat{A} = \widehat{L}\widehat{U}$ is an LU factorization without pivoting then $P^T\widehat{L}P$ and $P^T\widehat{U}P$ are lower triangular and upper triangular matrices, respectively, so that $A = (P^T\widehat{L}P)(P^T\widehat{U}P)$ is also an LU factorization.

- In practice, for many matrices the average size of a supernode is only 2 or 3 columns and many comprise a single column.
- Larger artificial supernodes may be created by merging vertex j with its parent vertex i in $\mathcal{T}(A^TA)$ if the subtree rooted at i has fewer than some chosen number of vertices.

Multifrontal LU

- The multifrontal method can be generalized to nonsymmetric A by modifying the definitions of the frontal matrices and generated elements to conform to an LU factorization.
- Natural generalizations to rectangular frontal and generated element matrices do not simultaneously satisfy the statements from above. Rewritten for the LU factorization:
  - (a) Each generated element V_j is used only once to contribute to a frontal matrix.
  - (b) The row and column index lists for the rectangular frontal matrix F_j correspond to the nonzeros in column $L_{j:n,j}$ and nonzeros in row $U_{j,j:n}$, respectively.
- An approach that satisfies (a) can be based on the sparsity pattern of $\mathcal{S}\{A + A^T\}$ and storing some explicit zeros if $\mathcal{S}\{A\}$ is not symmetric. It is then analogous to the symmetric multifrontal method.
- Performs well if $\mathcal{S}\{A\}$ is close to symmetric.

Multifrontal LU

- An approach that satisfies (b) and not necessarily (a) splits the generated elements into smaller ones that are embedded into further rectangular frontal matrices. We illustrate this using the example from above.

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{array}
\begin{array}{c}
\begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \end{array} \\
\begin{pmatrix}
* & & * & & & & & & & \\
* & * & f & & * & & & * & & * \\
* & & * & * & & & & & & \\
 & & * & * & & & & & * & * \\
 & & & * & * & * & & * & f & f \\
 & * & f & f & f & * & & f & f & f \\
 & & & * & & & * & & f & f \\
* & * & f & f & f & f & & * & f & f \\
 & & & & & & & & * & * \\
 & & & & * & f & * & f & f & *
\end{pmatrix}
\end{array},
$$

where $*$ are entries in A and filled entries in $L+U$ are denoted by f.

Multifrontal LU

$$F_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 1 & 3 \end{array} \\ \begin{pmatrix} * & * \\ * & \\ * & \\ * & \end{pmatrix} \end{array}, \quad V_1 = \begin{array}{c} \\ 2 \\ 3 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 3 \end{array} \\ \begin{pmatrix} f \\ * \\ f \end{pmatrix} \end{array}.$$

To construct F_2 that satisfies (b) we can only use part of V_1. Because $a_{13} \neq 0$, the sparsity pattern of column 1 is replicated in that of column 3 of the factors. The entry in position $(2, 3)$ belongs to F_2. The row replication implies that the remaining entries contribute to F_3 and so we split V_1 into two as follows

$$V_1^2 = \begin{array}{c} \\ 2 \end{array} \begin{array}{c} \begin{array}{c} 3 \end{array} \\ \begin{pmatrix} f \end{pmatrix} \end{array}, \quad V_1^3 = \begin{array}{c} \\ 3 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 3 \end{array} \\ \begin{pmatrix} * \\ f \end{pmatrix} \end{array}, \quad V_1 = V_1^2 \leftrightarrow V_1^3,$$

where $\leftrightarrow$ is the extend-add operator and V_1^2 and V_1^3 contribute to F_2 and F_3, respectively.

Multifrontal LU

Then F_2 and the corresponding generated element V_2 are

$$F_2 = \begin{array}{c} \\ 2 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 5 & 8 & 10 \end{array} \\ \begin{pmatrix} * & * & * & * \\ * & & & \\ * & & & \end{pmatrix} \end{array} \Leftrightarrow V_1^2 = \begin{array}{c} \\ 2 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccccc} 2 & 3 & 5 & 8 & 10 \end{array} \\ \begin{pmatrix} * & f & * & * & * \\ * & & & & \\ * & & & & \end{pmatrix} \end{array}, \quad V_2 = \begin{array}{c} \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccc} 3 & 5 & 8 & 10 \end{array} \\ \begin{pmatrix} f & f & f & f \\ f & f & * & f \end{pmatrix} \end{array}.$$

Consider the following splitting of V_2

$$V_2 = \begin{array}{c} \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 3 \end{array} \\ \begin{pmatrix} f \\ f \end{pmatrix} \end{array} \Leftrightarrow \begin{array}{c} \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 5 \end{array} \\ \begin{pmatrix} f \\ f \end{pmatrix} \end{array} \Leftrightarrow \begin{array}{c} \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 8 & 10 \end{array} \\ \begin{pmatrix} f & f \\ * & f \end{pmatrix} \end{array} \equiv V_2^3 \Leftrightarrow V_2^5 \Leftrightarrow V_2^6.$$

The next frontal matrix is

$$F_3 = \begin{array}{c} \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cc} 3 & 4 \end{array} \\ \begin{pmatrix} * & * \\ * & * \end{pmatrix} \end{array} \Leftrightarrow V_1^3 \Leftrightarrow V_2^3 = \begin{array}{c} \\ 3 \\ 4 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{cc} 3 & 4 \end{array} \\ \begin{pmatrix} * & * \\ * & * \\ f & \\ f & \end{pmatrix} \end{array}, \quad V_3 = \begin{array}{c} \\ 4 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 4 \end{array} \\ \begin{pmatrix} * \\ f \\ f \end{pmatrix} \end{array}.$$

The subsequent steps can be described in a similar way.

Multifrontal LU

- Theorem expresses the nested relationship between the nonsymmetric multifrontal method and the nonsymmetric elimination tree.

Theorem

Assume A is a general nonsymmetric matrix and $t = parent(k)$ in $\mathcal{T}(A)$. Then

$$\mathcal{S}\{L_{t:n,k}\} \subseteq \mathcal{S}\{L_{t:n,t}\} \quad and \quad \mathcal{S}\{U_{k,t:n}\} \subseteq \mathcal{S}\{U_{t,t:n}\}.$$

Proof.

Because t is the parent of k, by definition $t \xrightarrow{\mathcal{G}(L)} k \xrightarrow{\mathcal{G}(U)} t$. If $u_{ij} \neq 0$ then a multiple of column i is added to column j during the LU factorization. Thus, by a simple induction argument, for each j on the path $k \xrightarrow{\mathcal{G}(U)} t$, we must have $\mathcal{S}\{L_{j:n,k}\} \subseteq \mathcal{S}\{L_{j:n,j}\}$. In particular, this holds for column t. The second part follows by a similar argument using the path $t \xrightarrow{\mathcal{G}(L)} k$. □

Multifrontal LU

- The theorem shows that the parent relationship in the nonsymmetric elimination tree guarantees that both row and column replications can be applied at the same time.
- Hence all entries of the submatrices of the generated element V_k with indices greater than or equal to $parent(k)$ can be added to $V_{parent(k)}$ using the operation $\dotplus$.
- To illustrate this, consider again the 10×10 example above for which $parent(1) = 3$. Theorem guarantees that V_1 can be embedded into F_3 because $\mathcal{S}\{L_{3:n,1}\} \subseteq \mathcal{S}\{L_{3:n,3}\}$ and $\mathcal{S}\{U_{1,3:n}\} \subseteq \mathcal{S}\{U_{3,3:n}\}$.

Preprocessing for LU

- Consider the case when A does not have a full transversal (that is, it has one or more zeros on the diagonal).
- For numerical stability and to reduce the number of permutations required during the factorization, it can be useful to permute A before the factorization begins to put large nonzero entries on the diagonal.
- Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an edge subset $\mathcal{M} \subseteq \mathcal{E}$ is called a **matching** (or assignment) if no two edges in $\mathcal{M}$ are incident to the same vertex.
- The **cardinality** of a matching is the number of edges in it. A **maximum cardinality matching** (or **maximum matching**) is a matching of maximum cardinality. A matching is **perfect** if all the vertices are matched.

Sparse LU

- A **bipartite graph** is an undirected graph whose vertices can be partitioned into two disjoint sets such that no two vertices within the same set are adjacent, that is, each set is an **independent set**.
- Let the $n \times n$ matrix A have entries $\{a_{ij'}\}$. Associated with A is a bipartite graph defined as a triple $\mathcal{G}_b = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$, where the row vertex set $\mathcal{V}_{row} = \{i \,|a_{ij'} \neq 0\}$ and the column vertex set $\mathcal{V}_{col} = \{j' \,|a_{ij'} \neq 0\}$ correspond to the rows and columns of A and there is an (undirected) edge $(i, j') \in \mathcal{E}$ if and only if $a_{ij'} \neq 0$.
- In Figure below we use prime to distinguish between the independent set of row vertices and the independent set of column vertices, that is, i denotes a row vertex and i' denotes a column vertex.

Sparse LU

$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 1' & 2' & 3' & 4' & 5' & 6' \end{array} \\
\begin{pmatrix}
* & & * & & & \\
* & & * & & & * \\
 & & * & * & & \\
 & * & & * & & \\
 & * & & & * & \\
 & & * & * & & *
\end{pmatrix}
\end{array}
$$

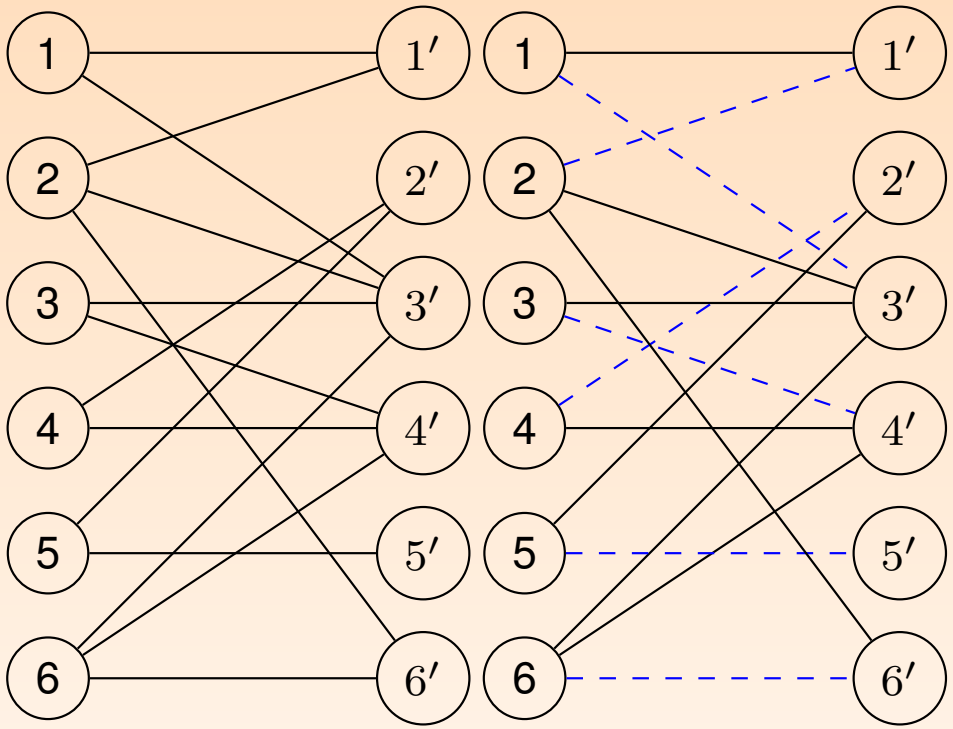


Figure: *A sparse matrix its bipartite graph $\mathcal{G}_b$ (left), perfect matching.*

Sparse LU

- If A is structurally nonsingular, a matching $\mathcal{M}$ in $\mathcal{G}_b$ is perfect if it has cardinality n.
- A perfect matching defines an $n \times n$ permutation matrix Q with entries q_{ij} given by

$$q_{ij} = \begin{cases} 1, & \text{if } (j, i') \in \mathcal{M}, \\ 0, & \text{otherwise.} \end{cases}$$

- Both QA and AQ have the matching entries on the (zero-free) diagonal.

Sparse LU

- Q and the column permuted matrix AQ for the example in Figure above.

$$Q = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} & 1 & & & & \\ & & & 1 & & \\ 1 & & & & & \\ & & 1 & & & \\ & & & & 1 & \\ & & & & & 1 \end{pmatrix} \end{array} \qquad AQ = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 3' & 1' & 4' & 2' & 5' & 6' \end{array} \\ \begin{pmatrix} * & * & & & & \\ * & * & & & & * \\ * & & * & & & \\ & & * & * & & \\ & & & * & * & \\ * & & * & & & * \end{pmatrix} \end{array}$$

Figure: *The permutation matrix Q and the column permuted matrix AQ corresponding to the matrix above. The matched entries are on the diagonal of AQ.*

Augmenting paths

- If a perfect matching exists, it can be found using augmenting paths.
- A path $\mathcal{P}$ in a graph is an ordered set of edges in which successive edges are incident to the same vertex. $\mathcal{P}$ is called an $\mathcal{M}$**-alternating path** if the edges of $\mathcal{P}$ are alternately in $\mathcal{M}$ and not in $\mathcal{M}$.
- An $\mathcal{M}$-alternating path is an $\mathcal{M}$**-augmenting path** in $\mathcal{G}_b$ if it connects an unmatched column vertex with an unmatched row vertex. Note that the length of an $\mathcal{M}$-augmenting path is an odd integer.

Sparse LU

- Let $\mathcal{M}$ and $\mathcal{P}$ be subsets of $\mathcal{E}$ and define the symmetric difference

$$\mathcal{M} \oplus \mathcal{P} := (\mathcal{M} \setminus \mathcal{P}) \cup (\mathcal{P} \setminus \mathcal{M}),$$

- This is set of edges that belongs to either $\mathcal{M}$ or $\mathcal{P}$ but not to both.
- If $\mathcal{M}$ is a matching and $\mathcal{P}$ is an $\mathcal{M}$-augmenting path, then $\mathcal{M} \oplus \mathcal{P}$ is a matching with cardinality $|\mathcal{M}|+1$.
- Growing the matching in this way is called augmenting along $\mathcal{P}$.

Theorem

A matching $\mathcal{M}$ in an undirected graph is a maximum matching if and only if there is no $\mathcal{M}$-augmenting path

Algorithm (**Maximum matching algorithm**)

Input: *An undirected graph.*
Output: *Output maximum matching.*

Find an initial matching $\mathcal{M}$ ▷ *For example,* $\mathcal{M} = \emptyset$
while *there exists a* $\mathcal{M}$*-augmenting path* $\mathcal{P}$ ***do***
    $\mathcal{M} = \mathcal{M} \oplus \mathcal{P}$ ▷ *Augment the matching along* $\mathcal{P}$
end while

Augmenting paths: demonstration

- On the left is a bipartite graph with a matching with cardinality 5. An augmenting path $2 \Longrightarrow 3' \Longrightarrow 3 \Longrightarrow 4' \Longrightarrow 4 \Longrightarrow 2'$ shown.
- Augmenting the matching along this path, the cardinality of the matching increases to 6 and $\mathcal{M} \oplus \mathcal{P}$ is a perfect matching.

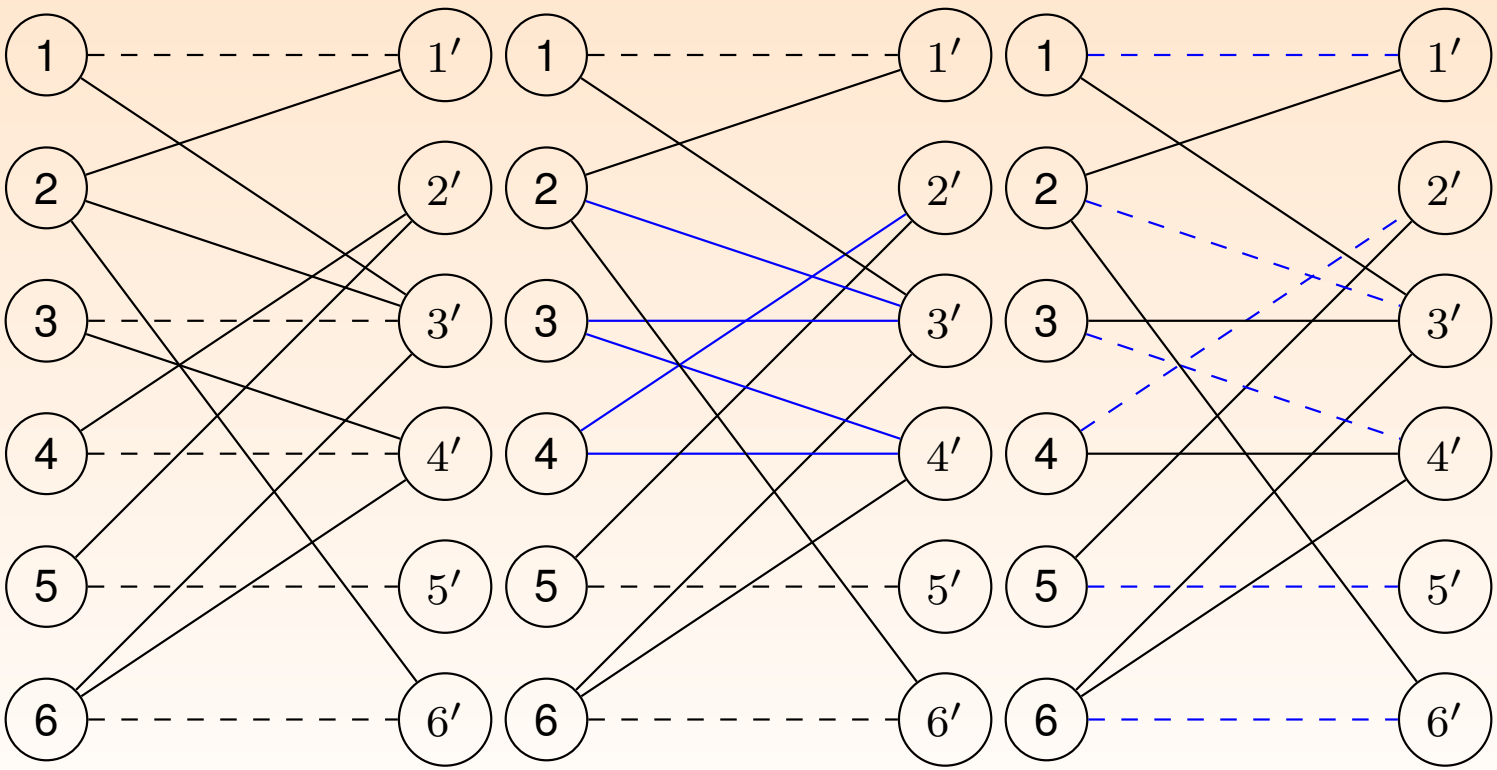


Figure: *Search for a perfect matching using augmenting paths.*

Weighted matchings

- While the maximum matching algorithm finds a permutation of A such that the permuted matrix has nonzero diagonal entries, there are more sophisticated variations that aim to ensure the absolute values of the diagonal entries of the permuted matrix (or their product) are in some sense large.
- The problem: given an $n \times n$ matrix A, find a matching of the rows to the columns such that the product of the matched entries is maximized.
- That is, find a permutation vector q that maximizes

$$\prod_{i=1}^{n} |a_{iq_i}|. \tag{46}$$

- Define a matrix C corresponding to A with entries $c_{ij'} \geq 0$ as follows:

$$c_{ij'} = \begin{cases} \log(\max_i |a_{ij'}|) - \log |a_{ij'}|, & \text{if } a_{ij'} \neq 0 \\ \infty, & \text{otherwise.} \end{cases} \tag{47}$$

Sparse LU

- It is straightforward to see that finding a q that solves the problem is equivalent to finding a q that minimizes

$$\sum_{i=1}^{n} |c_{iq_i}|, \tag{48}$$

- This is equivalent to finding a minimum weight perfect matching in an edge weighted bipartite graph.
- This is a well-studied problem and is known as the bipartite weighted matching or linear sum assignment problem.

Weighted matchings: formally

- If $\mathcal{G}_b = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ is the bipartite graph associated with A then let $\mathcal{G}_b(C) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ be the corresponding weighted bipartite graph in which each edge $(i, j') \in \mathcal{E}$ has a weight $c_{ij'} \geq 0$.
- The weight (or cost) of a matching $\mathcal{M}$ in $\mathcal{G}_b(C)$, denoted by $csum(\mathcal{M})$, is the sum of its edge weights; i.e.,

$$csum(\mathcal{M}) = \sum_{(i,j') \in \mathcal{M}} c_{ij'}.$$

- A perfect matching $\mathcal{M}$ in $\mathcal{G}_b(C)$ is said to be a **minimum weight perfect matching** if it has smallest possible weight i.e., $csum(\mathcal{M}) \leq csum(\widehat{\mathcal{M}})$ for all possible perfect matchings $\widehat{\mathcal{M}}$.

Weighted matchings: shortest augmenting paths

- The key concept for finding a minimum weight perfect matching is that of a **shortest augmenting path**.
- An $\mathcal{M}$-augmenting path $\mathcal{P}$ starting at an unmatched column vertex is called **shortest** if

$$csum(\mathcal{M} \oplus \mathcal{P}) \leq csum(\mathcal{M} \oplus \widehat{\mathcal{P}})$$

for all other possible $\mathcal{M}$-augmenting paths $\widehat{\mathcal{P}}$ starting at the same column vertex.

Weighted matchings: primal-dual algorithm

- A matching $\mathcal{M}_e$ is **extreme** if and only if there exist u_i and $v_{j'}$ (which are termed **dual variables**) satisfying

$$\begin{cases} c_{ij'} = u_i + v_{j'}, & \text{if } (i, j') \in \mathcal{M}_e, \\ c_{ij'} \geq u_i + v_{j'}, & \text{otherwise.} \end{cases} \tag{49}$$

- This is employed by the `MC64` algorithm.
- The dual variables can be used for scaling.
- The algorithm starts with a feasible solution and corresponding extreme matching and then proceeds to iteratively increase its cardinality by one by constructing a sequence of shortest augmenting paths until a perfect extreme matching is found.

Sparse LU

- Can be more efficient if a large initial matching is used.
- For example, Step 3 can be replaced by setting $u_i = \min\{c_{ij'} | \, j' \in \mathcal{S}\{A_{i,1:n}\}\}$ for $i \in \mathcal{V}_{row}$ and $v_{j'} = \min\{c_{ij'} - u_i | \, i \in \mathcal{S}\{A_{1:n,j'}\}\}$ for $j' \in \mathcal{V}_{col}$. In Step 4, an initial extreme matching can be determined from the edges for which $c_{ij'} - u_i - v_{j'} = 0$.

Algorithm (**Outline of the** `MC64` **algorithm)**

Input: *Matrix* A.
Output: *A matching* $\mathcal{M}$ *and dual variables* u_i, $v_{j'}$.

Define the weights $c_{ij'}$ *using (47)*
Construct the weighted bipartite graph $\mathcal{G}_b(C) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$
Set $u_i = 0$ *for* $i \in \mathcal{V}_{row}$ *and* $v_{j'} = \min\{c_{ij'} : (i, j') \in \mathcal{E}\}$ *for* $j' \in \mathcal{V}_{col}$ ▷ *Initial solution*
Set $\mathcal{M} = \{(i, j') | \, u_i + v_{j'}\}$ ▷ *Initial extreme matching*
while $\mathcal{M}$ *is not perfect* ***do***
    Find the shortest augmenting path $\mathcal{P}$ *with respect to* $\mathcal{M}$
    Augment the matching $\mathcal{M} = \mathcal{M} \oplus \mathcal{P}$
    Update u_i, $v_{j'}$ *so that (49) is satisfied for new* $\mathcal{M}$ ▷ *make* $\mathcal{M}$ *extreme*
end while

Sparse LU

- Number of potential problems with the `MC64` algorithm.
  - The runtime is hard to predict and depends on the initial ordering of A.
  - It is a serial algorithm and as such it can represent a significant fraction of the total factorization time of a direct solver.
- Because the complexity of Step 6 of Algorithm is $O((n + nz(A)) \log\ n)$ and the complexity of Step 7 is $O(n)$ and of Step 8 is $O(n + nz(A)$, `MC64` has a worst-case complexity of $O(n(n + nz(A)) \log\ n)$.
- In practice, this bound is not achieved and the algorithm is widely used.

Extension to non-square matrices

- The **Dulmage-Mendelsohn decomposition** is based on matchings and it generalizes the block triangular form.
- It comprises row and column permutations P and Q such that

$$PAQ = \begin{array}{c} \\ \mathcal{R}_1 \\ \mathcal{R}_2 \\ \mathcal{R}_3 \end{array} \begin{array}{c} \begin{array}{ccc} \mathcal{C}_1 & \mathcal{C}_2 & \mathcal{C}_3 \end{array} \\ \begin{pmatrix} A_1 & A_4 & A_6 \\ 0 & A_2 & A_5 \\ 0 & 0 & A_3 \end{pmatrix} \end{array}. \tag{50}$$

  Here A_1 is an $m_1 \times n_1$ underdetermined matrix ($m_1 < n_1$ or $m_1 = n_1 = 0$), A_2 is an $m_2 \times m_2$ square matrix and A_3 is an $m_3 \times n_3$ overdetermined matrix ($m_3 > n_3$ or $m_3 = n_3 = 0$).
- A_1^T and A_3 are strong Hall matrices but A_2 need not be a strong Hall matrix.

DM decomposition: an example

An example decomposition for a 10×10 matrix is given in Figure 43. Here $\mathcal{R} = \{1, 2, \ldots, 9\}$ and $\mathcal{C} = \{2, 3, \ldots, 10\}$.

$$PAQ = \left(\begin{array}{cccc|cccc|cc}
* & * & * & * & * & & & & & \\
 & & * & * & & * & * & & * & \\
 & * & & * & & & & & & \\
\hline
 & & & & * & & & & & * \\
 & & & & * & * & & * & & \\
 & & & & & & * & * & & \\
 & & & & & & * & * & * & \\
\hline
 & & & & & & & & * & * \\
 & & & & & & & & & * \\
 & & & & & & & & * &
\end{array}\right).$$

Figure: An example of a coarse Dulmage-Mendelsohn decomposition. The blue entries belong to the maximum matching. $m_1 = 3$, $m_2 = 4$, $m_3 = 3$, $n_1 = 4$, $n_2 = 4$, $n_3 = 2$. Column 1 and row 10 are unmatched.

Outline

Backward stability

- Practical computations are invariably based on finite precision arithmetic.
- Computational algorithm

$$z = g(d)$$

  for computing z as a function g of given data d.
- The algorithm is said to be backward stable if the computed solution $\hat{z}$ is the exact solution of $\hat{z} = g(d + \Delta d)$, where the perturbation Δd is "small" for all possible inputs d. What is meant by small depends on the context.
- If d is based on physical measurements (necessarily inaccurate), Δd is small if it is of the same or smaller absolute value as the inaccuracies in determining d.

Forward and backward errors

- The minimum absolute value $|\Delta d|$ among such perturbations is called the (absolute) backward error (or, if divided by $|d|$, the relative backward error).
- The absolute and relative errors of $\hat{z}$ are called forward errors, to distinguish them from the backward error.
- Backward stability is a property of the computational algorithm and to compute solutions with a small backward error we need to consider stable algorithms.

Stability and ill-conditioning

Ill-conditioning

- A related concept is ill-conditioning.
- We say that the problem $z = g(d)$ is ill-conditioned if small perturbations in the data d can lead to large changes in $\hat{z}$.
- The condition number measures how sensitive the output of a function is to its input. Ill-conditioning, which is measured in terms of the condition number, is a property of the problem.

Stability and ill-conditioning

Ill-conditioning

- Provided the backward error, forward error, and the condition number are defined in a consistent manner, the following approximate inequality holds:

$$\text{forward error} \lessapprox \text{condition number} \times \text{backward error}.$$

- This says that the computed solution to an ill-conditioned problem can have a large forward error because even if the computed solution has a small backward error, this error can be amplified by a large condition number.
- By preprocessing the problem it may be possible to improve its conditioning.

Stability and ill-conditioning

Backward error result for LU

ϵ denotes the machine precision.

Theorem

Let the computed LU factorization of a matrix A be $A + \Delta A = \widehat{L}\,\widehat{U}$. The perturbation ΔA that results from using finite precision arithmetic satisfies

$$||\Delta A||_\infty \leq n\, O(\epsilon)\, ||\widehat{L}||_\infty ||\widehat{U}||_\infty + O(\epsilon^2). \tag{51}$$

Moreover, the computed solution $\hat{x}$ of the linear system $Ax = b$ satisfies $(A + \Delta' A)\hat{x} = b$ with

$$||\Delta' A||_\infty \leq n\, O(\epsilon)\, ||\widehat{L}||_\infty ||\widehat{U}||_\infty + O(\epsilon^2). \tag{52}$$

Stability and ill-conditioning

Pivoting to improve stability

- At step k of GE, the computed diagonal entry $a_{kk}^{(k)}$ is termed the pivot ($1 \leq k < n$).
- Gaussian elimination breaks down if a zero pivot is encountered.
- Provided A is nonsingular, row interchanges can be incorporated to prevent this happening
- The systematic use of row permutations is called partial pivoting.
- If $|a_{kk}^{(k)}|$ is very small (compared to other entries in the active submatrix) then it can cause difficulties in finite precision arithmetic because the absolute value of the corresponding computed multiplier $l_{ik} = a_{ik}^{(k)}/a_{kk}^{(k)}$ can then be very large.
- Partial pivoting can be used to ensure $|l_{ik}| \leq 1$, that is, the rows of A that have not yet been pivoted on can be permuted so that the new pivot satisfies

$$\max_{i>k} |a_{ik}^{(k)}| \leq |a_{kk}^{(k)}|.$$

Stability and ill-conditioning

Pivoting to improve stability

- If P_k is the row permutation at stage k and $P = P_{n-1}P_{n-2}\dots P_1$ then the computed factors of PA satisfy

$$||\widehat{L}||_\infty \leq n \quad \text{and} \quad ||\widehat{U}||_\infty \leq n\,\rho_{growth}||A||_\infty,$$

where the growth factor ρ_{growth} is defined to be

$$\rho_{growth} = \max_{i,j,k}\left(\,|a_{ij}^{(k)}|\,/\,|a_{ij}|\,\right). \tag{53}$$

- The bounds (51) and (52) can be rewritten as

$$||\Delta A||_\infty \leq n^3\,\rho_{growth}\,O(\epsilon)\,||A||_\infty, \quad ||\Delta' A||_\infty \leq n^3\,\rho_{growth}\,O(\epsilon)\,||A||_\infty.$$

- In practice, these bounds are pessimistic and the actual errors are typically much smaller.
- Because backward stability of an LU factorization is influenced both by the initial ordering of A and the pivoting strategy, it is said to be conditionally backward stable.

Stability and ill-conditioning

Stability of Cholesky

- Pivoting not needed.
- Cholesky factorization of A is unconditionally backward stable. It enables the stable computation of the solution of the corresponding linear system.

Theorem

Let the computed Cholesky factorization of a SPD matrix A be $A + \Delta A = \widehat{L}\widehat{L}^T$. The perturbation ΔA that results from using finite precision arithmetic satisfies

$$||\Delta A||_\infty \leq n^2\, O(\epsilon)\, ||A||_\infty.$$

Moreover, the computed solution $\hat{x}$ of the linear system $Ax = b$ satisfies $(A + \Delta' A)\hat{x} = b$ with

$$||\Delta' A||_\infty \leq n^2\, O(\epsilon)\, ||A||_\infty.$$

Stability and ill-conditioning

Using the inverse instead of factorization

- No such stability results: The computed inverse is typically not the exact inverse of a nearby matrix $A + \Delta A$ for any small perturbation ΔA.
- Impractical to compute and store A^{-1}, regardless of how sparse A is: see below.

Using the inverse instead of factorization

Theorem

If A is irreducible then the sparsity pattern $\mathcal{S}\{A^{-1}\}$ of its inverse is fully dense.

Proof.

Without loss of generality, assume A is factorizable. For if not, there is a permutation matrix P such that the LU factorization of the row permuted matrix PA is factorizable. In this case, consider PA instead of A because for any permutation matrix P the inverse $(PA)^{-1}$ is fully dense if and only if A is fully dense. Consider the matrix K of order $2n$

$$K = \begin{pmatrix} A & I_n \\ I_n & 0 \end{pmatrix}.$$

After applying n elimination steps to $K = K^{(1)}$, the order n active submatrix of $K^{(n+1)}$ is $-A^{-1}$. Consider entry $(A^{-1})_{ij}$ $(1 \leq i, j \leq n)$. Because A is irreducible and the off-diagonal $(1, 2)$ and $(2, 1)$ blocks of K are equal to the identity matrix, there is a directed path $i \Longrightarrow j$ in $\mathcal{G}(K)$ such that the indices of all the intermediate vertices on the path are less than or equal to n. Theorem on fill paths and the non-cancellation assumption imply $(A^{-1})_{ij} \neq 0$. It follows that

Stability and ill-conditioning

Using the inverse instead of factorization

- Theorem above implies that entries of A^{-1} correspond to paths in $\mathcal{G}(A)$ even when A is not irreducible.

Corollary

If A is factorizable then $(A^{-1})_{ij} \neq 0$ ($1 \leq i, j \leq n$) if and only if there exists a path $i \overset{\mathcal{G}(A)}{\Longrightarrow} j$.

Pivoting strategies for dense matrices

- Convention here: all the quantities (such as $a_{ij}^{(k)}$) are the computed quantities.

Partial pivoting

- Partial pivoting interchanges rows at each step of the factorization to select the entry of largest absolute value in its column as the next pivot.
- If partial pivoting is used, it is straightforward to show that the growth factor satisfies

$$\rho_{growth} \leq 2^{n-1}.$$

- Can be achieved in nontrivial cases. But, generally pessimistic, particularly when n is very large.

Complete pivoting

- A much smaller bound can be obtained if complete (or full) pivoting is used: complete pivoting chooses the pivot to be the largest entry (in absolute value) in the active submatrix,
- That is, at stage k the pivot $a_{kk}^{(k)}$ is chosen so that

$$\max_{i \geq k, j \geq k} |a_{ij}^{(k)}| \leq |a_{kk}^{(k)}|.$$

- Then

$$\rho_{growth} \leq n^{1/2}(2.\, 3^{1/2}.\, 4^{1/3} \ldots n^{1/(n-1)})^{1/2}. \tag{54}$$

- Can be expensive.
- Relaxations in practice.

Rook pivoting

- Rook pivoting: more restrictive than. partial pivoting but cheaper than complete pivoting
- The pivot is chosen to be the largest entry in its row *and* its column, that is,

$$\max_{i>k} (|a_{ik}^{(k)}|, |a_{ki}^{(k)}|) \leq |a_{kk}^{(k)}|.$$

- In practice, the cost is usually a small multiple of the cost of partial pivoting and significantly less than that of complete pivoting.
- The growth factor for rook pivoting satisfies

$$\rho_{growth} \leq 1.5\, n^{(3/4)\log n}.$$

Stability and ill-conditioning

2×2 pivoting

- When the matrix A is symmetric but indefinite, it may not be possible to select pivots from the diagonal (for example, if all the diagonal entries of A are zero).
- If rows of A are permuted (so that off-diagonal entries are selected as pivots) then symmetry is destroyed, which means an LU factorization must be performed and this essentially doubles the cost of the factorization in terms of both storage and operation counts.
- Symmetry can be preserved by extending the notion of a pivot to 2×2 blocks.

Stability and ill-conditioning

Symmetric indefinite

- Consider
$$A = \begin{pmatrix} \delta & 1 \\ 1 & 0 \end{pmatrix}.$$
- If $\delta = 0$, an LDLT factorization in which D is a diagonal matrix does not exist.
- If $\delta \ll 1$ then an LDLT factorization with D diagonal is not stable because $\rho_{growth} = 1/\delta$.
- If the LDLT factorization is generalized to allow D to be a block diagonal matrix with 1×1 and 2×2 blocks on the diagonal then a factorization is obtained that preserves symmetry and is nearly as stable as an LU factorization.

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = LDL^T.$$

Here D has one 1×1 block and one 2×2 block.

Stability and ill-conditioning

Symmetric indefinite

- Rook pivoting can be extended to include 2×2 pivots. At each stage, an iterative procedure searches for an entry that is simultaneously the largest in absolute value in row i and column j of the active submatrix $A^{(k)}$.
- This entry is used to build a symmetric 2×2 pivot; the search terminates prematurely if a suitable 1×1 pivot is found, that is, a pivot that satisfies a threshold test.
- The standard choice for the threshold comes from requiring the same potential maximal growth in the absolute values of the entries of the partially eliminated matrix that results from either two consecutive 1×1 pivots or one 2×2 pivot.
- It can be shown that the appropriate choice is $(1 + \sqrt{17})/8$.
- $$\rho_{growth} < 3n\sqrt{2\ 3^{1/2}4^{1/3} \ldots n^{1/(n-1)}},$$

Pivoting strategies for sparse matrices

- Preserving sparsity needed.
- Threshold partial pivoting: the pivot is chosen to satisfy

$$\max_{i>k} |a_{ik}^{(k)}| \leq \gamma^{-1} |a_{kk}^{(k)}|, \tag{55}$$

where $\gamma \in (0, 1]$ is a chosen threshold parameter.

- It is straightforward to see that

$$\max_i |a_{ij}^{(k)}| \leq (1 + \gamma^{-1}) \max_i |a_{ij}^{(k-1)}|,$$

and

$$\max_i |a_{ij}^{(k)}| \leq (1 + \gamma^{-1})^{nz_j} \max_i |a_{ij}|,$$

where nz_j is the number of off-diagonal entries in the j-th column of the U factor. Furthermore,

$$\rho_{growth} \leq (1 + \gamma^{-1})^{nz_{cmax}},$$

where $nz_{cmax} = \max_j nz_j \leq n - 1$.

Pivoting strategies for sparse matrices

- A threshold can also be incorporated into rook pivoting. The pivot must then be at least γ times the absolute value of any other entry in its row and column of the active submatrix.
- In the symmetric case, if pivots are selected from the diagonal (to preserve symmetry), threshold partial pivoting is the same as threshold rook pivoting.
- Threshold rook pivoting has the potential to limit growth more successfully than threshold partial pivoting.

Stability and ill-conditioning

Threshold 2×2 pivoting and symmetry

- If A is a symmetric matrix then standard fill-reducing ordering algorithms (explained later) and the symbolic factorization phase employ only the sparsity pattern of A.
- In general, if A is indefinite, during the numerical factorization it is necessary to modify the chosen elimination order to maintain stability.
- If symmetry is to be preserved, 1×1 and 2×2 pivots are needed, resulting in an LDLT factorization in which D is a block diagonal matrix with 1×1 and 2×2 blocks.
- Limiting the size of the entries of L so that

$$|l_{ij}| \leq \gamma^{-1} \tag{56}$$

for all i, j, together with a backward stable scheme for solving 2×2 linear systems, suffices to show backward stability for the entire solution process.

Threshold 2×2 pivoting and symmetry

- In the sparse symmetric indefinite case, the stability test for a 1×1 pivot in column t of the active submatrix at stage k is the following standard threshold test

$$\max_{i \neq t,\, i \geq k} |a_{it}^{(k)}| \leq \gamma^{-1} |a_{tt}^{(k)}|, \tag{57}$$

where γ is the threshold parameter. For a 2×2 pivot in rows and columns s and t the corresponding test is

$$\left| \begin{pmatrix} a_{ss}^{(k)} & a_{st}^{(k)} \\ a_{st}^{(k)} & a_{tt}^{(k)} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i \neq s,t; i \geq k} |a_{is}^{(k)}| \\ \max_{i \neq s,t; i \geq k} |a_{it}^{(k)}| \end{pmatrix} \leq \gamma^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \tag{58}$$

where the absolute value of the matrix is interpreted element-wise.

Stability and ill-conditioning

Not only bounding pivots

- In addition to bounding the size of the entries in L, the ability to stably apply the inverse of D to a vector is required.
- This is trivially the case for 1×1 pivots, but for 2×2 pivots it is necessary to check that the determinant $|a_{ss}^{(k)} a_{tt}^{(k)} - a_{st}^{(k)} a_{st}^{(k)}|$ is sufficiently large.
- A major difficulty when stability tests are incorporated into sparse factorizations is that a pivot satisfying the stability criteria may not exist.

Stability and supernodes

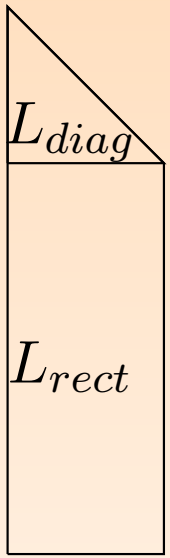


Figure: An illustration of a simple nodal matrix. Pivot candidates are restricted to the square block L_{diag} on the diagonal.

- Pivots can only be chosen from the block L_{diag} on the diagonal (the block is square and symmetric)
- Entries in the off-diagonal block L_{rect} are involved in the stability tests.
- Possibly delaying columns.

Stability and ill-conditioning

Stability and supernodes

- All the off-diagonal entries in a block column must be fully updated before the block on the diagonal is factorized.
- The factorize_block task and all the solve_block tasks for a block column from the SPD case are combined into a single factorize_column task.
- Fewer but larger tasks – reducing the scope for parallelism.
- Delayed pivots arises in the multifrontal method: Frontal matrix F of order n_F of the form

$$F = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & F_{22} \end{pmatrix}, \tag{59}$$

  where F_{11} is a $p \times p$ matrix corresponding to the fully summed variables.
- Pivots can only be selected from F_{11} but the numerical values of the entries in F_{21} must be taken into account when testing for stability.

Sparse indefinite factorization

Algorithm (Simple partial sparse indefinite factorization)

Input: *Symmetric indefinite matrix F of order n_F of the form (59) with F_{11} of order p; threshold $\gamma \in (0, 0.5]$.*
Output: *Updated F; partial factors L_F and D_F and permutation P_F.*

1: $q = 0, t = 0$ ▷ *q holds the sum of the sizes (1 or 2) of the pivots chosen so far*
2: ***while*** $q < p$ ***do***
3: *find_pivot* (piv_size) ▷ *See Algorithm 10.2*
4: *if* $(piv\_size = 0)$ *exit while loop* ▷ *Failed to find a pivot*
5: $q = q + piv\_size$
6: *Update columns $q + 1$ to p of F* ▷ *Right-looking*
7: ***end while***
8: *Apply updates to columns $p + 1$ to n_F of F* ▷ *Left-looking*

Algorithm (Find a pivot in F using threshold partial pivoting)

Input: *F, L_F, D_F, P_F, p, q, t, γ are accessed from the environment of the call.*
Output: *Pivot of size piv_size; columns $q+1 : q+piv\_size$ of L_F, D_F, updated P_F and t.*

---

subroutine find_pivot (piv_size)
$piv\_size = 0$
for $test = 1 : p-q$ ***do***
  $t = t+1$; *if* $(t > p)$ *set* $t = q+1$ ▷ *Column t is searched for a pivot*
  if *(there is s such that $q+1 \le s \le t-1$ and $\begin{pmatrix} f_{ss} & f_{st} \\ f_{st} & f_{tt} \end{pmatrix}$ passes 2×2 pivot test)* ***then***
    $piv\_size = 2$
    Symmetrically permute rows and columns $q+1$ and s of F ▷ *Update P_F*
    Symmetrically permute rows and columns $q+2$ and t of F ▷ *Update P_F*
    Compute columns $q+1$ and $q+2$ of D_F and L_F
    return
  else if *(f_{tt} passes 1×1 pivot test)* ***then***
    $piv\_size = 1$
    Symmetrically permute rows and columns $q+1$ and t of F ▷ *Update P_F*
    Compute column $q+1$ of D_F and L_F
    return
  end if
end for

Indefinite factorization: classical schemes

Algorithm

One step of full indefinite pivoting by Bunch and Parlett (1971)

Set $\alpha = (1+\sqrt{17})/8 \approx 0.64$
Find a_{kk}*: diagonal entry of maximum size*
Find a_{ij}*: off-diagonal entry of maximum size* $(i < j)$
if $|a_{kk}| \geq \alpha |a_{ij}|$ *then*
    use a_{kk} *as* 1×1 *pivot (ready for* $a_{kk} = 0$*)*
else
    use $\begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix}$ *as* 2×2 *pivot*
end if

- Full pivoting: choosing entries of largest magnitudes: can be expensive.
- But a growth factor bound can be derived (only a slightly worse than for LU)

Indefinite factorization: classical schemes

Algorithm

$\alpha = (1+\sqrt{17})/8 \approx 0.64, i = 1$ *(possible an initial preprocessing)*
Find $j \neq i$ *such that* $a_{ji} = \max\{|a_{ki}|, k \neq i\} =: \lambda$
if $|a_{ii}| \geq \alpha\lambda$ *then*
    use a_{ii} *as* 1×1 *pivot*
else
    $\sigma = \max\{|a_{kj}|, k \neq j\}$
    if $|a_{ii}|\sigma \geq \alpha\lambda^2$ *then*
        use a_{ii} *as* 1×1 *pivot*
    else if $|a_{jj}| \geq \alpha\sigma$ *then*
        use a_{jj} *as a* 1×1 *pivot*
    else
        use $\begin{pmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{pmatrix}$ *as* 2×2 *pivot*
    end if
end if

Indefinite factorization: classical schemes

- This scheme shows why the previous algorithm (Bunch-Kaufmann) is useful to factorize sparse matrices
- The price for less searches expressed theoretically by large growth factor bound

$$\begin{pmatrix} d & . & . & \lambda & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ \lambda & . & . & c & . & \sigma & . \\ . & . & . & . & . & . & . \\ . & . & . & \sigma & . & . & . \\ . & . & . & . & . & . & . \end{pmatrix}$$

Stability and ill-conditioning

Indefinite factorization: classical schemes

- What if only f is used to decide such that $|f| \geq \tau|\lambda|$?
- Again weaker bounds
- This can be practical if the processed matrix is not available in a current step (this can happen in multifrontal approaches). Implies rules like

Algorithm

if $|d| \geq \alpha|\lambda|$ *use* d *as* 1×1 *pivot*
if $|d\gamma| \geq \alpha|\lambda|^2$*: use* d *as* 1×1 *pivot*
if $|e| \geq \alpha|\gamma|$*: use* e *as* 1×1 *pivot*
else
use $\begin{pmatrix} d & f \\ f & e \end{pmatrix}$ *as* 2×2 *pivot*

Symmetric indefinite factorization

$$\begin{pmatrix} d & . & f & \lambda & . & . & . \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ f & . & e & . & . & \gamma & . \\ \lambda & . & . & . & . & . & . \\ . & . & \gamma & . & . & . & . \\ . & . & . & . & . & . & . \end{pmatrix}$$

Stability and ill-conditioning

Relaxed and static pivoting

- If pivots are delayed during the numerical factorization then the data structures that were set up during the symbolic phase must be modified. This significantly complicates the development of general and symmetric indefinite sparse direct solvers.

  If no candidate pivot satisfies the threshold test but the pivot that is nearest to satisfying it would satisfy it with a threshold $\gamma_1 < \gamma$, then provided γ_1 is at least some chosen minimum value, relaxed pivoting accepts this pivot and reduces γ to γ_1.

  With relaxed pivoting, delayed pivots can still occur and it may not be possible to use static data structures.

- A standard technique is to employ regularization (modification of A). This can avoid the need for an LDLT factorization in favour of a stable Cholesky factorization.

Special indefinite matrices that avoid pivoting

- Symmetric saddle point matrices: indefinite matrices of the form

$$A = \begin{pmatrix} G & R^T \\ R & -B \end{pmatrix}, \tag{60}$$

where $G \in \mathbb{R}^{n_1 \times n_1}$ is a SPD matrix, $B \in \mathbb{R}^{n_2 \times n_2}$ is a positive semidefinite matrix (including $B = 0$), and $R \in \mathbb{R}^{n_2 \times n_1}$ with $n_1 + n_2 = n$.

- Define the permutation matrix P to be

$$P = [e_1,\ e_{n_1+1},\ e_2,\ e_{n_1+2},\ \ldots\ e_{n_1},\ e_n,\ e_{n_2+1},\ \ldots,\ e_{n_1}]^T.$$

Then the permuted matrix PAP^T has a block form in which each entry $A_{i,j}$ is a 2×2 or 2×1 or 1×2 or 1×1 block. In particular,

$$A_{i,i} = \begin{cases} \begin{pmatrix} g_{ii} & r_{ii} \\ r_{ii} & -b_{ii} \end{pmatrix}, & 1 \le i \le n_2 \\ b_{ii}, & n_2 + 1 \le i \le n_1. \end{cases}$$

Stability and ill-conditioning

- The following theorem shows that a 2×2 pivot updated by the Schur complement of a 1×1 pivot is nonsingular and vice versa.

Theorem

Let A be the symmetric saddle point matrix. Assume $R = (R_1 \; R_2)$ is of full rank with $R_1 \in \mathbb{R}^{n_2 \times n_2}$ nonsingular. Let $G \in \mathbb{R}^{n_1 \times n_1}$ be SPD and partitioned conformally and let $B \in \mathbb{R}^{n_2 \times n_2}$ be positive semidefinite matrix. If A is permuted to the form

$$\left(\begin{array}{cc|c} G_{11} & R_1^T & G_{12} \\ R_1 & -B & R_2 \\ \hline G_{12}^T & R_2^T & G_{22} \end{array}\right)$$

then the Schur complement of the symmetric indefinite matrix $\begin{pmatrix} G_{11} & R_1^T \\ R_1 & -B \end{pmatrix}$ and the Schur complement of the SPD matrix G_{22} are nonsingular.

Stability and ill-conditioning

- Provided R is of full rank and R_1 is nonsingular then the LDLT factorization of PAP^T exists, with 2×2 pivots and 1×1 pivots chosen from the diagonal blocks of PAP^T in any order.
- Assume all the 2×2 pivots are selected ahead of the 1×1 pivots. Then if $B = 0$ and $|r_{ii}| \geq \max_{i \leq j \leq n_1} |r_{ij}|$ $(1 \leq i \leq n_2)$ then the growth factor is bounded by 2^{2n_2}.
- A potential difficulty is that permutation matrices P_r and P_c are needed such that $P_r R P_c = [R_1 \ R_2]$ with R_1 nonsingular. If P_r and P_c can be constructed so that

$$P_r R P_c = \begin{pmatrix} R_{11} & R_{12} \\ & R_{22} \end{pmatrix}, \tag{61}$$

where R_{11} is upper triangular with nonzero diagonal entries then the permuted R is said to have a trapezoidal form.

Special indefinite matrices that avoid pivoting

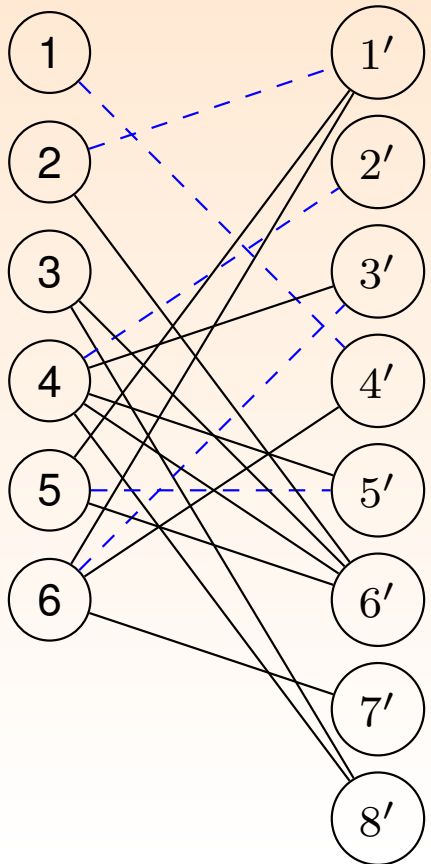
- A simple case where R can be permuted this form is if it satisfies the following one-degree principle. Let R be of full rank and let $\mathcal{G}_b(R) = (\mathcal{V}_{row}, \mathcal{V}_{col}, \mathcal{E})$ be the bipartite graph of R (Section **??**). R can be permuted to trapezoidal form if, for $k = 1, 2, \ldots, n_1 - 1$, the bipartite graph of $R^{(k)}$ has at least one vertex $j'_k \in \mathcal{V}_{col}$ of degree one, where $R^{(1)} = R$ and $R^{(k+1)}$ is obtained by removing from $R^{(k)}$ the column vertex j'_k and its matched row index i_k together with all edges involving j'_k or i_k.

Stability and ill-conditioning

- To illustrate the above mentioned case: consider the 6×8 matrix R and its associated bipartite graph $\mathcal{G}_b(R)$.
- The first column vertex with degree one is $2'$; it is matched with the row vertex 4. Deleting $2'$ and 4 removes edges $\{(4, 2'), (4, 3'), (4, 5'), (4, 6'), (4, 8')\}$. Column vertex $3'$ now has degree one; it is matched with row vertex 6.
- Repeating the process gives a perfect matching $\mathcal{M} = \{(4, 1'), (6, 3'), (1, 4'), (5, 5'), (2, 6'), (3, 8')\}$ together with row and column matched vertex sets $\{4, 6, 1, 5, 2, 3\}$ and $\{2', 3', 4', 5', 1', 6'\}$, respectively, and permutation matrices P_r and P_c of order 6 and 8 can be defined to obtain the trapezoidal form in Figure 45.

Stability and ill-conditioning

$$R = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1' & 2' & 3' & 4' & 5' & 6' & 7' & 8' \end{array} \\ \begin{pmatrix} & & & * & & & & \\ * & & & & & * & & \\ & & & & & * & & * \\ & * & * & & * & * & & * \\ * & & & & * & * & & \\ * & & * & * & & & * & \end{pmatrix} \end{array} \qquad P_r R P_c = \begin{array}{c} \\ 4 \\ 6 \\ 1 \\ 5 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccccc} 2' & 3' & 4' & 5' & 1' & 6' & 7' & 8' \end{array} \\ \begin{pmatrix} * & * & & * & & * & & * \\ & * & * & & * & & * & \\ & & * & & & & & \\ & & & * & * & * & & \\ & & & & * & * & & \\ & & & & & * & & * \end{pmatrix} \end{array}$$



Stability and ill-conditioning

- If after $k \geq 1$ steps all columns of the reduced matrix $R^{(k)}$ have degree greater than 1, the permuted matrix has the form above, where R_{11} is $k \times k$ upper triangular, R_{12} is $k \times (n_1 - k)$ and the $(n_2 - k) \times (n_1 - k)$ block R_{22} has columns of degree greater than one.
- $n_1 - k$ steps of Gaussian elimination (with partial pivoting) can be applied to R_{22} to complete the transformation of R to trapezoidal form.

Solving ill-conditioned problems

- Ill-conditioning is connected to the input data: a problem is ill-conditioned if small changes in the data can lead to large changes in the solution. Assume for the general linear system $Ax = b$ that A and b are perturbed by ΔA and Δb, respectively, and the corresponding perturbation of the solution x is Δx, so that the perturbed problem

$$(A + \Delta A)(x + \Delta x) = b + \Delta b \tag{62}$$

has been solved. The perturbations in A and b may include both data uncertainty and algorithmic errors. Rearranging (62), we obtain

$$A\Delta x = \Delta b - \Delta A - \Delta A \Delta x.$$

Stability and ill-conditioning

- Premultiplying by A^{-1} and considering *any* norm $\|.\|$ and the corresponding subordinate matrix norm yields

$$\|\Delta x\| \leq \|A^{-1}\| \left(\|\Delta b\| + \|\Delta A\| \, \|x\| + \|\Delta A\| \, \|\Delta x\|\right).$$

  It follows that

$$(1 - \|A^{-1}\| \, \|\Delta A\|)\|\Delta x\| \leq \|A^{-1}\| \left(\|\Delta b\| + \|\Delta A\| \, \|x\|\right)$$

  and, provided $\|A^{-1}\| \, \|\Delta A\| < 1$, this gives the following bound on the absolute error

$$\|\Delta x\| \leq \frac{\|A^{-1}\|}{\|A^{-1}\| \, \|\Delta A\|} \left(\|\Delta b\| + \|\Delta A\| \, \|x\|\right).$$

- Dividing by $\|x\|$ and using $\|b\| \leq \|A\| \, \|x\|$, yields the relative error bound

$$\|\Delta x\|/\|x\| \leq \frac{\kappa(A)}{1 - \kappa(A)\|\Delta A\|/\|A\|} \left(\|\Delta A\|/\|A\| + \|\Delta b\|/\|b\|\right). \quad (63)$$

Stability and ill-conditioning

Solving ill-conditioned problems

- Here
$$\kappa(A) = \|A\| \, \|A^{-1}\| \tag{64}$$
is the condition number of the matrix A.
- The inequality shows that the condition number is a relative error magnification factor. If we have a stable algorithm then a neighbouring problem has been solved, that is,
$$\|\Delta A\|/\|A\| + \|\Delta b\|/\|b\|$$
is small. This ensures an accurate solution if $\kappa(A)$ is small. A large condition number means that A is close to being singular ($\kappa(A)$ tends to infinity as A tends to singularity).
- Observe that the condition number is very dependent on the scaling of A. Furthermore, $\kappa(A)$ takes no account of the right-hand side vector b or the fact that small entries of A (including zeros) may be known within much smaller tolerances than larger entries.
- If the matrix norm is that induced by the Euclidean norm (that is,

Iterative refinement

- Iterative refinement can be used to overcome matrix ill-conditioning and improve the accuracy of the computed solution. It may also be used after relaxed or static pivoting.

Algorithm (Iterative refinement of the computed solution of $Ax = b$)

Input: *The vector b and matrix A.*
Output: *A sequence of approximate solutions $x^{(0)}, x^{(1)}, \ldots$.*

Solve $Ax^{(0)} = b$ ▷ $x^{(0)}$ *is the initial computed solution*
for $k = 0, 1, \ldots$ ***do***
    Compute $r^{(k)} = b - Ax^{(k)}$ ▷ *Residual on iteration* k
    Solve $A\,\delta x^{(k)} = r^{(k)}$ ▷ *Solve correction equation*
    $x^{(k+1)} = x^{(k)} + \delta x^{(k)}$
end for

Iterative refinement

- Iterative refinement terminates when either the norm of the residual vector $r^{(k)}$ is sufficiently close to zero that the corresponding correction $\delta x^{(k)}$ is very small or the chosen maximum number of iterations is reached.
- A possible approach is to switch to using the computed factors as a preconditioner for a Krylov subspace solver.

Stability and ill-conditioning

Iterative refinement

- In fixed precision refinement, all computations use the same precision. In mixed precision iterative refinement, the most expensive parts of the computation (the LU factorization of A and solving the correction equation) are performed in single precision and the residual computation in double precision.
- Holding the factors in single precision substantially reduces the memory required and the amount of data movement. The use of half precision (16 bit) arithmetic is also a possibility, assuming it is considerably faster than single precision, with a proportional saving in energy consumption.

Theorem

One step of single precision iterative refinement enough for obtaining componentwise relative backward error to the order of $O(\epsilon)$ under weaker assumptions.
Strong bound for the error norm using double precision iterative refinement.

Scaling to reduce ill-conditioning

- An important way to decrease the condition number is by scaling A before the numerical factorization begins.
- Consider two nonsingular $n \times n$ diagonal matrices S_r and S_c. Diagonal scaling of the system $Ax = b$ transforms it to

$$S_r\, A\, S_c\, y = S_r\, b, \qquad y = S_c^{-1}\, x. \tag{66}$$

Theorem

Let the matrix A be SPD and let D_A be the diagonal matrix with entries a_{ii} $(1 \le i \le n)$. Then for all diagonal matrices D with positive entries

$$\kappa(D_A^{-1/2}\, A\, D_A^{-1/2}) \le nz_{rmax}\, \kappa(D^{-1/2}\, A\, D^{-1/2}),$$

where nz_{rmax} is the maximum number of entries in a row of A.

Equilibration scaling

- Finding an appropriate scaling is an open question, but a number of heuristics have been proposed.
- An obvious choice is to seek to balance entries of the scaled matrix $S_r A S_c$ to have approximately equal absolute values. This is called (approximate) equilibration scaling.

Stability and ill-conditioning

Algorithm 10.7 presents an iterative procedure for computing such a scaling.

Algorithm (Equilibration scaling in the infinity norm)

Input: *The matrix A and convergence tolerance $\delta > 0$.*
Output: *Diagonal scaling matrices S_r and S_c.*

1: $B^{(1)} = A$, $D^{(1)} = I$, $E^{(1)} = I$
2: ***for*** $k = 1, 2, \ldots$ ***do***
3: Compute $\|B^{(k)}_{i,1:n}\|_\infty$ and $\|B^{(k)}_{1:n,i}\|_\infty$, $1 \le i \le n$ ▷ i-th row and column of $B^{(k)}$
4: if $max_i \left\{|1 - \|B^{(k)}_{i,1:n}\|_\infty|\right\} \le \delta$ and $max_i \left\{|1 - \|B^{(k)}_{1:n,i}\|_\infty|\right\} \le \delta$ exit for loop
5: $R = diag\left(\sqrt{\|B^{(k)}_{i,1:n}\|_\infty}\right)$ and $C = diag\left(\sqrt{\|B^{(k)}_{1:n,i}\|_\infty}\right)$
6: $B^{(k+1)} = R^{-1} B^{(k)} C^{-1}$, $D^{(k+1)} = D^{(k)} R^{-1}$, $E^{(k+1)} = E^{(k)} C^{-1}$
7: ***end for***
8: $S_r = D^{(k+1)}$ and $S_c = E^{(k+1)}$

Stability and ill-conditioning

- Our equilibration scaling preserves symmetry. In the nonsymmetric case, Algorithm yields the same results when applied to A and A^T in the sense that the scaled matrix obtained for A^T is the transpose of that for A.
- The infinity norm may be replaced by the 1-norm, resulting in a matrix whose row and column sums are exactly one (this is sometimes called a doubly stochastic matrix).
- It can be advantageous to combine the use of the infinity and one norms.

Matching-based scalings

- Recall the problem of finding a permutation vector q that maximizes the product

$$\prod_{i=1}^{n} |a_{iq_i}|.$$

- The dual variables u_i and v_j computed by the `MC64` algorithm that seeks to compute q can be used:
- Define the diagonal scaling matrices S_r and S_c to have entries

$$(S_r)_{ii} = \exp(u_i),\ (S_c)_{jj} = \exp(v_j - \log(\max_i |a_{ij}|)),\ 1 \le i, j \le n.$$

The entries of the scaled matrix $S_r A S_c$ satisfy

$$|(S_r A S_c)_{ij}| \begin{cases} = 1, & \text{if } (i,j) \in \mathcal{M}, \\ \le 1, & \text{otherwise,} \end{cases}$$

where $\mathcal{M}$ is the maximum weighted matching computed by the `MC64` algorithm.

Stability and ill-conditioning

Combining matching-based scalings and orderings

- The matching-based ordering and scaling can be used independently but they can also be combined.
- After scaling, if the matched entries are non symmetrically permuted onto the diagonal then because they are large, they provide good pivot candidates for an LU factorization.
- This approach is commonly used alongside static pivoting to obtain a factorization of a perturbed matrix, followed by iterative refinement to recover the solution to the original system.

Stability and ill-conditioning

- In the symmetric indefinite case, symmetry needs to maintained and so the objective is to symmetrically permute a large off-diagonal entry a_{ij} onto the subdiagonal to give a 2×2 block $\begin{pmatrix} a_{ii} & a_{ij} \\ a_{ij} & a_{jj} \end{pmatrix}$ that is potentially a good 2×2 candidate pivot.
- Assume that a matching $\mathcal{M}$ has been computed using the `MC64` algorithm and let q be the corresponding permutation vector.
- Any diagonal entries that are in the matching are immediately considered as potential 1×1 pivots and are held in a set $\mathcal{M}_1$.
- A set $\mathcal{M}_2$ of potential 2×2 pivots is then built by expressing q in terms of its component cycles. A cycle of length 1 corresponds to a_{ii} in the matching. A cycle of length 2 corresponds to i and j, where a_{ij} and a_{ji} are both in the matching. k potential 2×2 pivots can be extracted from even cycles of length $2k$ or from odd cycles of length $2k + 1$.
- In practice, most cycles in q are of length 1 or 2.

Stability and ill-conditioning

- Example for based on matching from `MC64`: $\mathcal{M} = \{(1,2),(2,5),(3,1),(4,4),(5,3)\}$, which is nonsymmetric. q has one cycle of length 4 ($1 \to 2 \to 5 \to 3 \to 1$) and one of length 1, giving $\mathcal{M}_1 = \{(4,4)\}$ and $\mathcal{M}_2 = \{(1,2),(2,1),(3,5),(5,3)\}$.

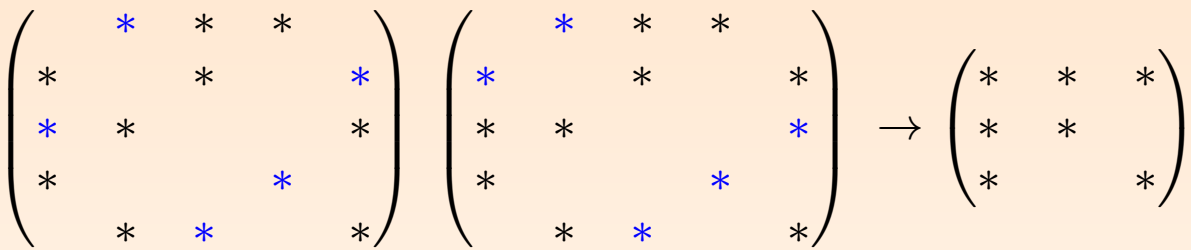


Figure: An illustration of a symmetric matching for a symmetric indefinite matrix. On the left is the matching $\mathcal{M}$ returned by the `MC64` algorithm and in the centre is a symmetric matching $\mathcal{M}_s$ obtained from $\mathcal{M}$. Entries in the matching are in blue. The pairs $(i,j) = (1,2)$ and $(3,5)$ are possible 2×2 pivot candidates. On the right is the compressed matrix that results from combining rows and columns 1 and 2 and rows and columns 3 and 5.

Outline

Complexity of factorizations

- Starting by showing complexity of LU and Cholesky
- The complexity of the most critical steps in the factorization is highly dependent on the amount of fill-in, as can be seen from the following observation.

Observation

The operations to perform the sparse LU factorization $A = LU$ *and the sparse Cholesky factorization* $A = LL^T$ *are* $O(\sum_{j=1}^{n} |col_L\{j\}||row_U\{j\}|)$ *and* $O(\sum_{j=1}^{n} |col_L\{j\}|^2)$ *respectively, where* $|row_U\{j\}|$ *and* $|col_L\{j\}|$ *are the number of off-diagonal entries in row* j *of* U *and column* j *of* L*, respectively.*

Reorderings

Minimizing the fill-in: reorderings

- Key problem: minimizing the fill-in
- Our tools: permutations.
- The problem of finding a permutation to minimize fill-in is NP complete. Thus heuristics are used to determine orderings that limit the amount of fill-in; we refer to these as fill-reducing orderings.
- Frequently, this is done using the sparsity pattern $\mathcal{S}\{A\}$ alone,
- If the matrix is not SPD, additional permutations of A may be needed to make the matrix factorizable.

Minimizing the fill-in: reorderings

Two main classes of reorderings that work with $\mathcal{S}\{A\}$ are commonly used.

Local orderings attempt to limit fill-in by repeated local decisions based on $\mathcal{G}(A)$ (or a relevant quotient graph).

Global orderings consider the whole sparsity pattern of A and seek to find a permutation using a divide-and-conquer approach. Such methods are normally used in conjunction with a local fill-reducing ordering, as the latter generally works well for problems that are not really large.

Minimizing the fill-in: reorderings

- Assumed that A is irreducible. If not,
  - If $\mathcal{S}\{A\}$ is symmetric, the algorithms are applied to each component of $\mathcal{G}(A)$ independently and n is then the number of vertices in the component.
  - If $\mathcal{S}\{A\}$ is nonsymmetric, we assume that A is in block triangular form and the algorithms are used on the graph of each block on the diagonal.
- We also assume that A has no rows or columns that are (almost) dense. If so, such rows and/or columns should be treated independently.

Minimizing the fill-in: reorderings

- Historically, ordering the matrix A before using a direct solver to factorize it was generally cheap compared to the numerical factorization cost.
- It is not the case nowadays due to the development of the computational tools.
- In the symmetric case, the diagonal entries of A are required to be present in $\mathcal{S}\{A\}$ (thus zeros on the diagonal are included in the sparsity structure). The aim is to limit fill-in in the L factor of an LL^T (or LDL^T) factorization of A.
- Two greedy heuristics are the minimum degree (MD) criterion and the local minimum fill (MF) criterion.

Minimum fill-in (MF) criterion

- One way to reduce fill-in is to use a local minimum fill-in (MF) criterion:
  - Select as the next variable in the ordering one that will introduce the least fill-in in the factor at that step.
- This is sometimes called the minimum deficiency approach.
- MF can produce good orderings, its cost is often considered to be prohibitive.
- An approximate variant (AMF).

Basic minimum degree (MD) algorithm

- The best-known and most widely-used greedy heuristic for limiting fill-in.
- It seeks to find a permutation such that at each step of the factorization the number of entries in the corresponding column of L is approximately minimized.
- Less expensive to compute compared to that used by the minimum fill-in criterion.
- The MD algorithm can be implemented using $\mathcal{G}(A)$ and it can predict the required factor storage without generating the structure of L.
- At step k, the number of off-diagonal nonzeros in a row or column of the active submatrix is the current degree of the corresponding vertex in the elimination graph $\mathcal{G}^k$. The algorithm selects a vertex of minimum current degree in $\mathcal{G}^k$ and labels it v_k, i.e., next for elimination.

Minimum degree algorithm

Algorithm (Basic minimum degree (MD) algorithm)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix.*
Output: *A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.*

1: *Set $\mathcal{G}^1 = \mathcal{G}$ and compute the degree $deg_{\mathcal{G}^1}(u)$ of all $u \in \mathcal{V}(\mathcal{G}^1)$*
2: ***for*** $k = 1 : n-1$ ***do***
3: *Compute $mdeg = \min\{deg_{\mathcal{G}^k}(u) \,|\, u \in \mathcal{V}(\mathcal{G}^k)\}$* ▷ *$mdeg$ is the current minimum degree*
4: *Choose $v_k \in \mathcal{V}(\mathcal{G}^k)$ such that $deg_{\mathcal{G}^k}(v_k) = mdeg$*
5: $p(k) = v_k$ ▷ *v_k is the next vertex in the elimination order*
6: *Construct $\mathcal{G}^{k+1}$ and update the current degrees of its vertices*
7: ***end for***
8: $p(n) = v_n$ *where v_n is the only vertex in $\mathcal{G}^n$*

Minimum degree algorithm

- The set of vertices adjacent to v_k in $\mathcal{G}(A)$ is $\mathcal{R}each(v_k, \mathcal{V}_k)$, where $\mathcal{V}_k$ is the set of $k-1$ vertices that have already been eliminated.
- If $u \in \mathcal{R}each(v_k, \mathcal{V}_k)$, $u \neq v_k$, then its updated current degree is $|\mathcal{R}each(u, \mathcal{V}_{k+1})|$, where $\mathcal{V}_{k+1} = \mathcal{V}_k \cup v_k$.
- A tie-breaking strategy is needed when there is more than one vertex of current minimum degree.
- It is possible to construct artificial matrices showing that some systematic tie-breaking choices can lead to a large amount of fill-in but such behaviour is not typical.

Minimum degree algorithm

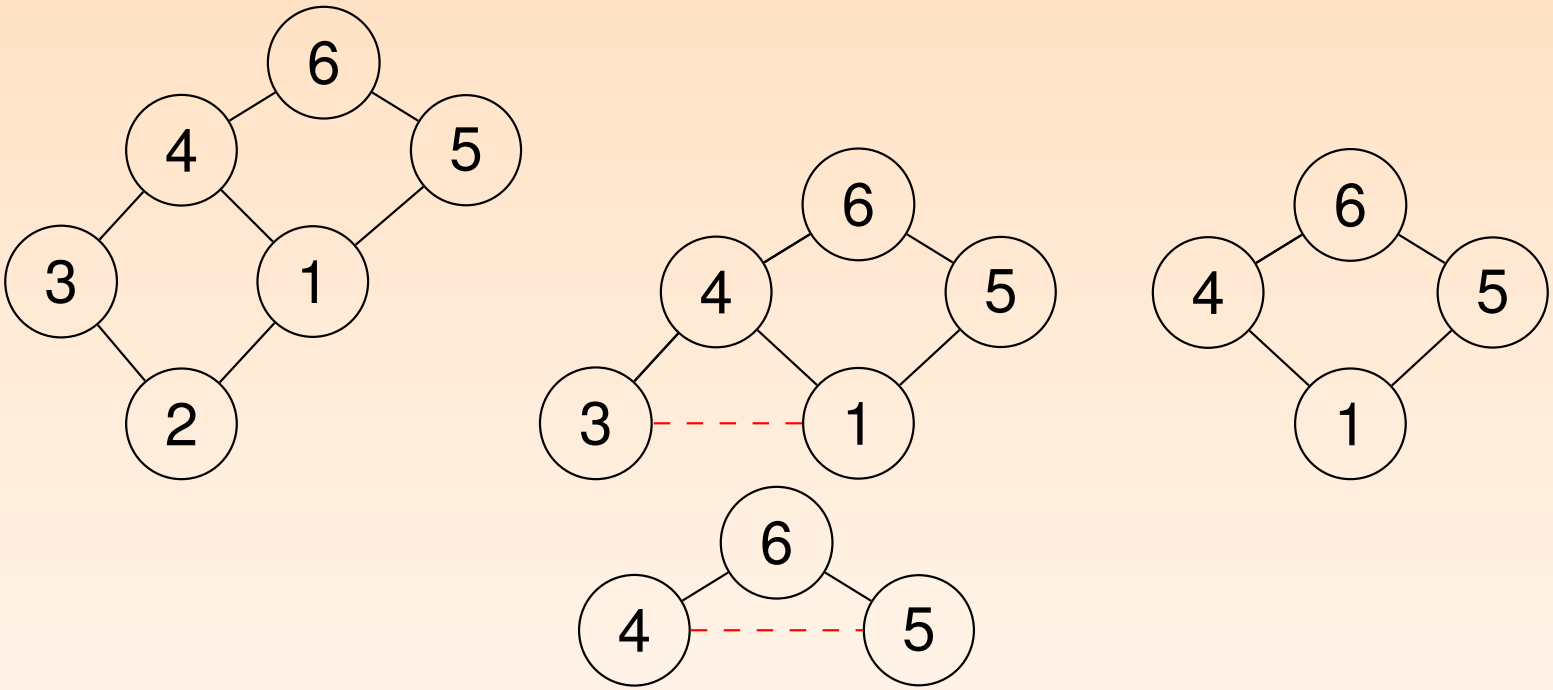


Figure: *An illustration of three steps of the MD algorithm. The original graph $\mathcal{G}$ and the elimination graphs $\mathcal{G}^2$, $\mathcal{G}^3$ and $\mathcal{G}^4$ that result from eliminating vertex 2, then vertex 3 and then vertex 1. The red dashed lines denote fill edges.*

Minimum degree algorithm

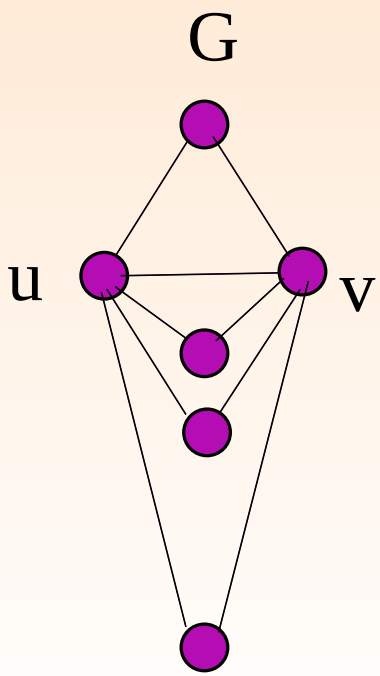
- The construction of each elimination graph $\mathcal{G}^{k+1}$ is central to the implementation.
- Because eliminating a vertex potentially creates fill-in, an efficient representation is needed.
- Moreover, recalculating the current degrees is time consuming.
- Using supervariables is a must.
- $\mathcal{G}_v$ denotes the elimination graph obtained from $\mathcal{G}$ when vertex $v \in \mathcal{V}(\mathcal{G})$ is eliminated.

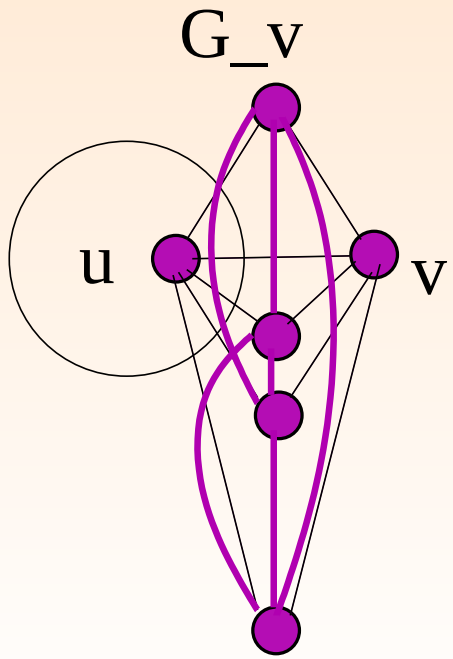
Indistinguishability (reminder)

Definition

Two different vertices **u** and **v** of G are called **indistinguishable** if

$$Adj_G(u) \cup \{u\} = Adj_G(v) \cup \{v\}. \tag{67}$$





Indistinguishability

Theorem

Let u and w be indistinguishable vertices in $\mathcal{G}$. If $v \in \mathcal{V}(\mathcal{G})$ with $v \neq u, w$, then u and w are indistinguishable in $\mathcal{G}_v$.

Proof.

Two cases must be considered. First, let $u \notin adj_{\mathcal{G}}\{v\}$. Then $w \notin adj_{\mathcal{G}}\{v\}$ and if v is eliminated, the adjacency sets of u and w are unchanged and they remain indistinguishable in the resulting elimination graph $\mathcal{G}_v$. Second, let $u, w \in adj_{\mathcal{G}}\{v\}$. When v is eliminated, because u and w are indistinguishable in $\mathcal{G}$, their adjacency sets in $\mathcal{G}_v$ will be modified in the same way, by adding the entries of $adj_{\mathcal{G}}\{v\}$ that are not already in $adj_{\mathcal{G}}\{u\}$ and $adj_{\mathcal{G}}\{w\}$. Consequently, u and w are indistinguishable in $\mathcal{G}_v$. □ □

Indistinguishability

Figure demonstrates the two cases in the proof of Theorem above. Here, u and w are indistinguishable vertices in $\mathcal{G}$. Setting $v \equiv v'$ illustrates $u \notin adj_{\mathcal{G}}\{v\}$. If v' is eliminated then the adjacency sets of u and w are clearly unchanged. Setting $v \equiv v''$ illustrates $u, w \in adj_{\mathcal{G}}\{v\}$. In this case, if v'' is eliminated then vertices s and t are added to both $adj_{\mathcal{G}}\{u\}$ and $adj_{\mathcal{G}}\{w\}$.

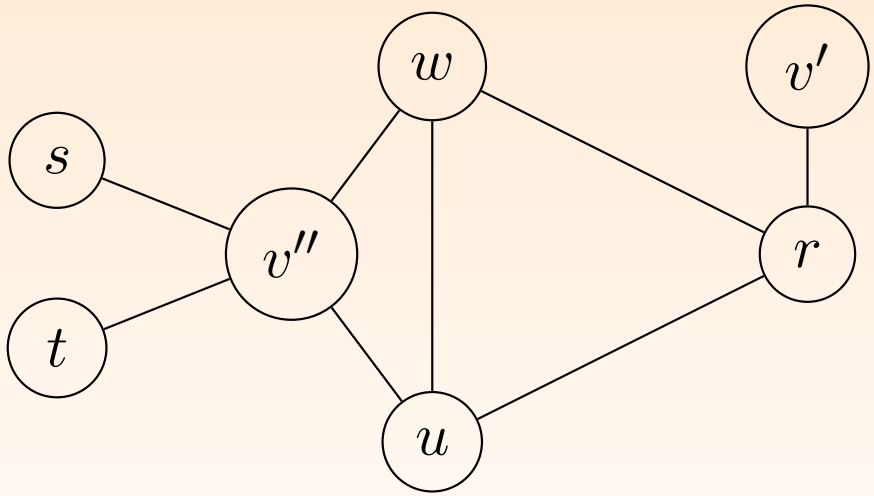


Figure: *An example to illustrate the Theorem.* u *and* w *are indistinguishable vertices in* $\mathcal{G}$*;* $adj_{\mathcal{G}}\{u\} = \{r, w, v''\}$ *and* $adj_{\mathcal{G}}\{w\} = \{r, u, v''\}$.

Indistinguishability

Theorem

Let u and w be indistinguishable vertices in $\mathcal{G}$. If w is of minimum degree in $\mathcal{G}$ then u is of minimum degree in $\mathcal{G}_w$.

Proof.

Let $deg_{\mathcal{G}}(w) = mdeg$. Then $deg_{\mathcal{G}}(u) = mdeg$. Indistinguishable vertices are always neighbours. Eliminating w gives $deg_{\mathcal{G}_w}(u) = mdeg - 1$ because w is removed from the adjacency set of u and there is no neighbour of u in $\mathcal{G}_w$ that was not its neighbour in $\mathcal{G}$. If $x \neq w$ and $x \in adj_{\mathcal{G}}\{u\}$ then the number of neighbours of x in $\mathcal{G}_w$ is at least $mdeg - 1$. Otherwise, if $x \notin adj_{\mathcal{G}}\{u\}$ then its adjacency set in $\mathcal{G}_w$ is the same as in $\mathcal{G}$ and is of the size at least $mdeg$. The result follows. □

Indistinguishability

Theorem above is illustrated in Figure.

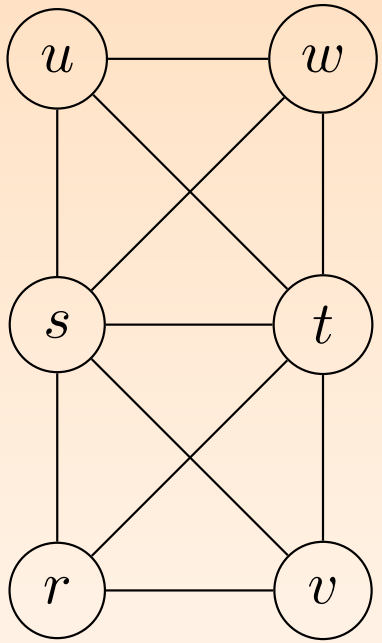


Figure: *An illustration of Theorem. Vertices u and w are of minimum degree (with degree $mdeg = 3$) and are indistinguishable in $\mathcal{G}$. After elimination of w, the current degree of u is $mdeg - 1$ and the current degree of each of the other vertices is at most $mdeg - 1$. Therefore, u is of current minimum degree in $\mathcal{G}_w$. Note that vertices r and v are also of minimum degree and indistinguishable in $\mathcal{G}$; they are not neighbours of w and their degrees do not change when w is eliminated.*

Indistinguishability

- The results can be extended to more than two indistinguishable vertices, which allows indistinguishable vertices to be selected one after another in the MD ordering.
- This is referred to as mass elimination.
- Treating indistinguishable vertices as a single supervariable cuts the number of vertices and edges in the elimination graphs, which reduces the work needed for degree updates.
- The external degree of a vertex is the number of vertices adjacent to it that are not indistinguishable from it. Using this leads to algorithmic efficiency.

Degree outmatching

- A concept that is closely related to that of indistinguishable vertices is degree outmatching.
- This avoids computing the degrees of vertices that are known not to be of current minimum degree.
- Vertex w is said to be outmatched by vertex u if

$$adj_{\mathcal{G}}\{u\} \cup \{u\} \subseteq adj_{\mathcal{G}}\{w\} \cup \{w\}.$$

- It follows that $deg_{\mathcal{G}}(u) \leq deg_{\mathcal{G}}(w)$.

Degree outmatching

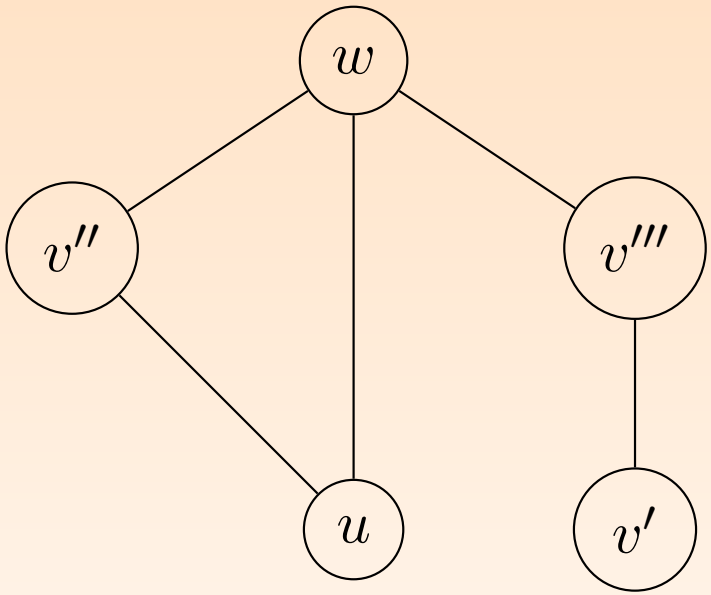


Figure: *An example $\mathcal{G}$ in which vertex w is outmatched by vertex u. v' is not a neighbour of u or w; vertex v'' is a neighbour of both u and w; v''' is a neighbour of w but not of u.*

Degree outmatching

- Importantly, degree outmatching is preserved when vertex $v \in \mathcal{G}$ of minimum degree is eliminated, as stated in the following result.

Theorem

In the graph $\mathcal{G}$ let vertex w be outmatched by vertex u and vertex v ($v \neq u, w$) be of minimum degree. Then w is outmatched in $\mathcal{G}_v$ by u.

Proof.

Three cases must be considered. First, if $u \notin adj_{\mathcal{G}}\{v\}$ and $w \notin adj_{\mathcal{G}}\{v\}$ then the adjacency sets of u and w in $\mathcal{G}_v$ are the same as in $\mathcal{G}$. Second, if v is a neighbour of both u and w in $\mathcal{G}$ then any neighbours of v that were not neighbours of u and w are added to their adjacency sets in $\mathcal{G}_v$. Third, if $u \notin adj_{\mathcal{G}}\{v\}$ and $w \in adj_{\mathcal{G}}\{v\}$ then the adjacency set of u in $\mathcal{G}_v$ is the same as in $\mathcal{G}$ but any neighbours of v that were not neighbours of w are added to the adjacency set of w in $\mathcal{G}_v$. In all three cases, w is still outmatched by u in $\mathcal{G}_v$. □ □

Degree outmatching

- The three possible cases for v in the proof of Theorem are illustrated in Figure above by setting $v \equiv v'$, v'' and v''', respectively.
- If w is outmatched by u then it is not necessary to consider w as a candidate for elimination and
- all updates to the data structures related to w can be postponed until u has been eliminated.

Cliques and quotient graphs

- From Parter's rule, if vertex v is selected at step k then the elimination matrix that corresponds to $\mathcal{G}^{k+1}$ contains a dense submatrix of size equal to the number of off-diagonal entries in row and column v in the matrix that corresponds to $\mathcal{G}^k$.
- For large matrices, creating and explicitly storing the edges in the sequence of elimination graphs is impractical and a more compact and efficient representation is needed.
- Each elimination graph can be interpreted as a collection of cliques, including the original graph $\mathcal{G}$, which can be regarded as having $|\mathcal{E}|$ cliques, each consisting of two vertices (or, equivalently, an edge).

Cliques and quotient graphs

- Let $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_q\}$ be the set of cliques for the current graph and let v be a vertex of current minimum degree that is selected for elimination. Let $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ be the subset of cliques to which v belongs. Two steps are then required.
  1. Remove the cliques $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ from $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_q\}$.
  2. Add the new clique $\mathcal{V}_v = \{\mathcal{V}_{s_1} \cup \ldots \cup \mathcal{V}_{s_t}\} \setminus \{v\}$ into the set of cliques.
- Hence

$$deg_{\mathcal{G}}(v) = |\mathcal{V}_v| < \sum_{i=1}^{t} |\mathcal{V}_{s_i}|,$$

  and because $\{\mathcal{V}_{s_1}, \mathcal{V}_{s_2}, \ldots, \mathcal{V}_{s_t}\}$ can now be discarded, the storage required for the representation of the sequence of elimination graphs never exceeds that needed for $\mathcal{G}(A)$.
- The storage to compute an MD ordering is therefore known beforehand in spite of the rather dynamic nature of the elimination process.

Cliques and quotient graphs

- The index of the eliminated vertex can be used as the index of the new clique. This is called an element or enode (the terminology comes from finite-element methods), to distinguish it from an uneliminated vertex, which is termed an snode.
- A sequence of special quotient graphs $\mathcal{G}^{[1]} = \mathcal{G}(A), \mathcal{G}^{[2]}, \ldots, \mathcal{G}^{[n]}$ with the two types of vertices is generated in place of the elimination graphs.
- Each $\mathcal{G}^{[k]}$ has n vertices that satisfy

$$\mathcal{V}(\mathcal{G}) = \mathcal{V}_{snodes} \cup \mathcal{V}_{enodes}, \qquad \mathcal{V}_{snodes} \cap \mathcal{V}_{enodes} = \emptyset,$$

  where $\mathcal{V}_{snodes}$ and $\mathcal{V}_{enodes}$ are the sets of snodes and enodes, respectively.
- When v is eliminated, any enodes adjacent to it are no longer required to represent the sparsity pattern of the corresponding active submatrix and so they can be removed. This is called element absorption.

Cliques and quotient graphs

- Working with the special quotient graphs can be demonstrated by considering the computation of the vertex degrees.
- To compute the degree of an uneliminated vertex, the set of neighbouring snodes is counted.
- Then, if a neighbour of one of these snodes is an enode, its neighbours are also counted (avoiding double counting).
- More formally, if $v \in \mathcal{V}_{snodes}$ then the adjacency set of v is the union of its neighbours in $\mathcal{V}_{snodes}$ and the vertices reachable from v via its neighbours in $\mathcal{V}_{enodes}$.
- In this way, vertex degrees are computed by considering fill-paths
- Amalgamation improves this strategy: mass elimination.

Cliques and quotient graphs: mass elimination model

Definition

Mass elimination graph Γ of the graph $G = (V, E)$ is a ordered triple $(\mathcal{S}, \mathcal{E}, E)$, where $\mathcal{S} \cup \mathcal{E} = V, \mathcal{S} \cap \mathcal{E} = \emptyset$ and $E \subseteq \binom{\mathcal{S}}{2} \cup \binom{\mathcal{E}(\Gamma)}{2}$ are its edges.

- Edge set $\mathcal{E}$ captures eliminated vertices.
- Edge set $\mathcal{S}$ captures non-eliminated vertices.
- Neighbors of non-eliminated vertices are found as reachability sets.
- Search through the reachability sets can be pruned: $\rightarrow$ approximate minimum degree (AMD) algorithm.

Cliques and quotient graphs

- After elimination of 1, a new edge is added, getting a clique. The elimination of 2 creates no additional fill and $\mathcal{G}^3$ represents the sparsity structure of the corresponding active submatrix $A^{(3)}$.
- Then, 1 is an enode, the fill edge is represented implicitly. After the second step, the enodes 1 and 2 can be amalgamated as well as snodes 3 and 4 being indistinguishable.

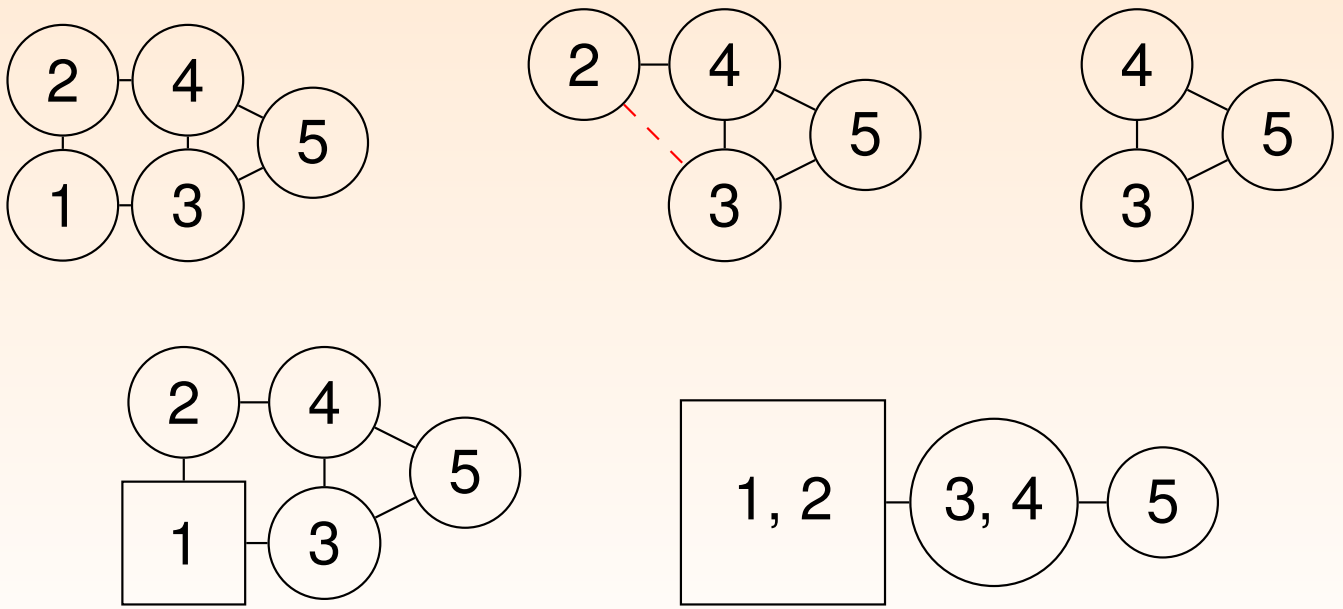


Figure: *The top line shows $\mathcal{G} = \mathcal{G}^1$, $\mathcal{G}^2$ and $\mathcal{G}^3$. The bottom line shows the quotient graphs $\mathcal{G}^{[2]}$ and $\mathcal{G}^{[3]}$ after the first and second elimination steps.*

Multiple minimum degree (MMD) algorithm

- The multiple minimum degree (MMD) algorithm aims to improve efficiency by processing several independent vertices that are each of minimum current degree together in the same step, before the degree updates are performed.
- At each outer loop, $t \geq 1$ denotes the number of vertices of minimum current degree that are mutually non-adjacent and so can be put into the elimination order one after another.
- An example follows.

Reorderings

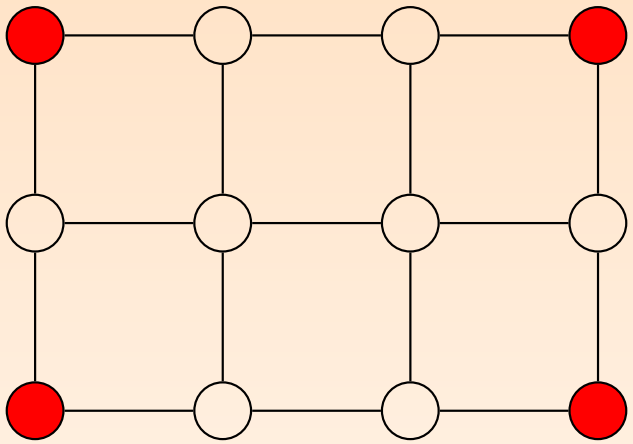


Figure: *The red (corner) vertices of $\mathcal{G}$ are each of degree 2 and are ordered consecutively during the first step of Algorithm 11.2.*

Multiple minimum degree (MMD) algorithm

Algorithm (Basic multiple minimum degree (MMD) algorithm)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix.*
Output: *A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.*

1: *Set $k = 1$, $\mathcal{G}^1 = \mathcal{G}$ and compute the degree $deg_{\mathcal{G}^1}(u)$ of all $u \in \mathcal{V}(\mathcal{G}^1)$*
2: ***while*** $k \leq n$ ***do***
3: *Compute $mdeg = \min\{deg_{\mathcal{G}^k}(u) \,|\, u \in \mathcal{V}(\mathcal{G}^k)\}$*
4: *Find all mutually non-adjacent $v_j \in \mathcal{V}(\mathcal{G}^k)$, $j = 1, \ldots, t$ with $deg_{\mathcal{G}^k}(v_j) = mdeg$*
5: ***for*** $j = 1 : t$ ***do***
6: $p(k) = v_j$ ▷ *Vertex v_j is the next vertex in the elimination order*
7: $k = k + 1$
8: ***end for***
9: ***if*** $k < n$ ***then***
10: *Construct $\mathcal{G}^{k+1}$ and update the current degrees of its vertices*
11: ***end if***
12: ***end while***

MD and MMD: complexity

- The complexity of the MD and MMD algorithms is $O(nz(A)n^2)$ but because for MMD the outer loop of the algorithm update is performed fewer times, it can be significantly faster then MD.
- MMD orderings can also lead to less fill-in, possibly a consequence of introducing some kind of regularity into the ordering sequence.

Approximate minimum degree (AMD) algorithm

- The idea behind the widely-used approximate minimum degree (AMD) algorithm is to inexpensively compute an upper bound on a vertex degree in place of the degree, and to use this bound as an approximation to the external degree.
- The quality of the orderings obtained using the AMD algorithm are competitive with those computed using the MD algorithm and can surpass them.
- The complexity of AMD is $O(nz(A)n)$ and, in practice, its runtime is typically significantly less than that of the MD and MMD approaches.

Minimizing the bandwidth and profile

- An alternative way of reducing the fill-in locally is to add another criterion to the relabelling of the vertices, such as restricting the nonzeros of the permuted matrix to specific positions.
- The most popular approach is to force them to lie close to the main diagonal.
- All fill-in then takes place between the first entry of a row and the diagonal or between the first entry of a column and the diagonal.
- This allows straightforward implementations of Gaussian elimination that employ static data structures.
- Here we again consider symmetric $\mathcal{S}\{A\}$; generalizations are possible.

Reorderings

The band and envelope of a matrix

Denote:

$$\eta_i(A) = \min\{j \mid 1 \le i \le j, \text{ with } a_{ij} \neq 0\}, \quad 1 \le i \le n, \tag{68}$$

that is, $\eta_i(A)$ is the column index of the first entry in the i-th row of A.

Define

$$\beta_i(A) = i - \eta_i(A), \quad 1 \le i \le n.$$

The semibandwidth of A is

$$\max\{\beta_i(A) | \ 1 \le i \le n\},$$

and the bandwidth is

$$\beta(A) = 2 * \max\{\beta_i(A) \mid 1 \le i \le n\} + 1.$$

The band of A is the following set of index pairs in A

$$band(A) = \{(i,j) \mid 0 < i - j \le \beta(A)\}.$$

The band and envelope of a matrix

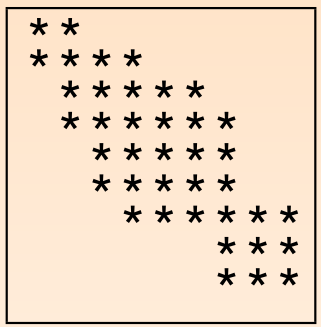
- The envelope is the set of index pairs that lie between the first entry in each row and the diagonal:

$$env(A) = \{(i, j) \mid 0 < i - j \leq \beta_i(A)\}.$$

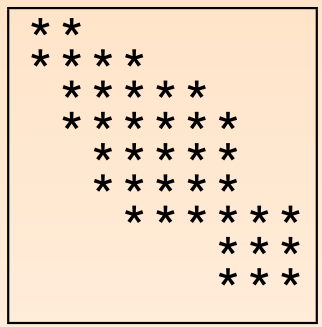
- Note that the band and envelope of a sparse symmetrically structured matrix A include only entries of the strict lower triangular part of A.
- The envelope is easily visualized: picture the lower triangular part of A, and remove the diagonal and the leading zero entries in each row. The remaining entries (whether nonzero or zero) comprise the envelope of A.
- The profile of A is defined to be the number of entries in the envelope (the envelope size) plus n.

Reorderings

The band and envelope of a matrix: shape pushers

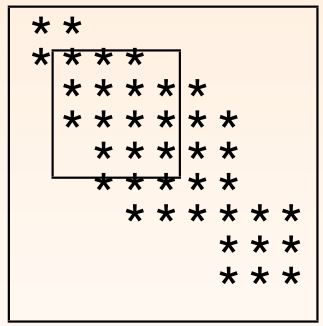


Band



Profile

Moving window



Frontal method - dynamic band

The band and envelope of a matrix

An illustrative example: Here $\eta_1(A) = 1$, $\beta_1(A) = 0$, $\eta_2(A) = 1$, $\beta_2(A) = 1$, $\eta_3(A) = 2$, $\beta_3(A) = 1$, and so on.

Figure: *Illustration of the band and envelope of a matrix A whose sparsity pattern is on the left. In the centre, the positions of $band(A)$ are circled and on the right, the positions of $env(A)$ are circled. The bandwidth is 5 and the envelope size and the profile are 7 and 14, respectively.*

The band and envelope of a matrix

- Static structures!

Theorem

If L is the Cholesky factor of A then

$$env(A) = env(L).$$

Proof.

The proof uses mathematical induction on the principal leading submatrices of A of order k. The result is clearly true for $k = 1$ and $k = 2$. Assume it holds for $2 \leq k < n$ and consider the block factorization

$$\begin{pmatrix} A_{1:k,1:k} & u_{1:k} \\ u_{1:k}^T & \alpha \end{pmatrix} = \begin{pmatrix} L_{1:k,1:k} & 0 \\ v_{1:k}^T & \beta \end{pmatrix} \begin{pmatrix} L_{1:k,1:k}^T & v_{1:k} \\ 0 & \beta \end{pmatrix},$$

where α and β are scalars. Equating the left and right sides, $L_{1:k,1:k} v_{1:k} = u_{1:k}$. Because $u_j = 0$ for $j < \eta_{k+1}(A)$ and $u_{\eta_{k+1}} \neq 0$, it follows that $v_j = 0$ for $j < \eta_{k+1}(A)$ and $v_{\eta_{k+1}} \neq 0$. This proves the induction step. □

A straightforward corollary is that $band(A) = band(L)$.

Reorderings

The band and envelope of a matrix

- Finding a permutation P to minimize the band or profile of PAP^T is again combinatorially hard and again heuristics are used to efficiently find an acceptable P.
- The popular Cuthill McKee (CM) approach chooses a suitable starting vertex s and labels it 1.
- Then, for $i = 1, 2, \ldots, n-1$, all vertices adjacent to vertex i that are still unlabelled are labelled successively in order of increasing degree, as described in Algorithm below.
- A very important variation is the Reverse Cuthill McKee (RCM) algorithm, which incorporates a final step in which the CM ordering is reversed.

Level-based orderings

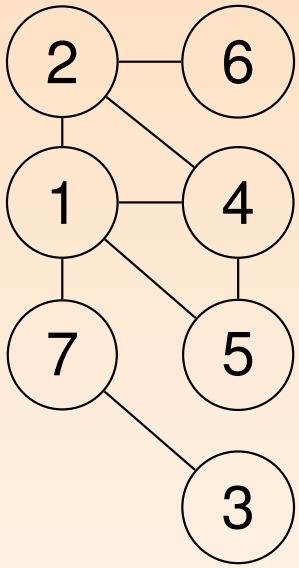
Algorithm (CM and RCM algorithms for band and profile reduction)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix and a starting vertex s.*
Output: *Permutation vectors p_{cm} and p_{rcm} that define new labellings of the vertices of $\mathcal{G}(A)$.*

1: $label(1:n) = false$
2: *Compute* $adj_{\mathcal{G}}\{u\}$ *and* $deg_{\mathcal{G}}(u)$ *for all* $u \in \mathcal{V}(\mathcal{G})$
3: $k = 1,\ v_1 = s,\ p_{cm}(1) = v_1,\ label(v_1) = true$
4: ***for*** $i = 1 : n-1$ ***do***
5: ***for*** $w \in adj_{\mathcal{G}}\{v_i\}$ *with* $label(w) = false$ *in order of increasing degree* ***do***
6: $k = k+1,\ v_k = w,\ p_{cm}(k) = v_k,\ label(v_k) = true$
7: ***end for***
8: ***end for***
9: *For the RCM ordering,* $p_{rcm}(i) = p_{cm}(n-i+1),\ i = 1, 2, \ldots, n.$

Reorderings

- The CM- and RCM-permuted matrices have the same bandwidth but the latter can decrease the envelope.



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{pmatrix} * & * & & * & * & & * \\ * & * & & * & & * & \\ & & * & & & & * \\ * & * & & * & * & & \\ * & & & * & * & & \\ & * & & & & * & \\ * & & * & & & & * \end{pmatrix} \end{array}, \quad \begin{array}{c} \\ 3 \\ 7 \\ 1 \\ 5 \\ 2 \\ 4 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} 3 & 7 & 1 & 5 & 2 & 4 & 6 \end{array} \\ \begin{pmatrix} * & * & & & & & \\ * & * & * & & & & \\ & * & * & * & * & * & \\ & & * & * & & * & \\ & & * & & * & * & * \\ & & * & * & * & * & \\ & & & & * & & * \end{pmatrix} \end{array}, \quad \begin{array}{c} \\ 6 \\ 4 \\ 2 \\ 5 \\ 1 \\ 7 \\ 3 \end{array}\!\!\begin{array}{c} \begin{array}{ccccccc} 6 & 4 & 2 & 5 & 1 & 7 & 3 \end{array} \\ \begin{pmatrix} * & & * & & & & \\ & * & * & * & * & & \\ * & * & * & & * & & \\ & * & & * & * & & \\ & * & * & * & * & * & \\ & & & & * & * & * \\ & & & & & * & * \end{pmatrix} \end{array},$$

Level-based orderings

The importance of the CM and RCM orderings is also expressed by:

Theorem

Let A be symmetrically structured and irreducible. If P corresponds to the CM labelling obtained from Algorithm and L is the Cholesky factor of $P^T AP$ then $env(L)$ is full, that is, all entries of the envelope are nonzero.

- The full envelope of the Cholesky factor of the permuted matrix implies cache efficiency when performing the triangular solves once the factorization is complete.
- A crucial difference between profile reduction ordering algorithms and minimum degree strategies is that the former is based solely on $\mathcal{G}$: the costly construction of quotient graphs is not needed.
- However, unless the profile after reordering is very small, there can be significantly more fill-in in the factor.

Reorderings

Level-based orderings

- Key to the success is the choice of the starting vertex s.
- A good candidate is a vertex for which the maximum distance between it and some other vertex in $\mathcal{G}$ is large.
- Formally, the eccentricity $\epsilon(u)$ of the vertex u in the connected undirected graph $\mathcal{G}$ is defined to be

$$\epsilon(u) = \max_{v \in \mathcal{V}} d(u, v),$$

  where $d(u, v)$ is the distance between the vertices u and v (the length of the shortest path between these vertices).
- The maximum eccentricity taken over all the vertices is the diameter of $\mathcal{G}$ (that is, the maximum distance between any pair of vertices). The endpoints of a diameter (also termed peripheral vertices) provide good starting vertices.
- The complexity of finding a diameter is $O(n^3)$: approximation (pseudo-preferal vertices) are needed.

Level-based orderings

- A heuristic algorithm is used to find pseudo-peripheral vertices. A commonly-used approach is based on level sets. A level structure rooted at a vertex r is defined as the partitioning of $\mathcal{V}$ into disjoint levels $\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_h(r)$ such that

  (i) $\mathcal{L}_1(r) = \{r\}$ and

  (ii) for $1 < i \leq h$, $\mathcal{L}_i(r)$ is the set of all vertices that are adjacent to vertices in $\mathcal{L}_{i-1}(r)$ but are not in $\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_{i-1}(r)$.

- The level structure rooted at r may be expressed as the set $\mathcal{L}(r) = \{\mathcal{L}_1(r), \mathcal{L}_2(r), \ldots, \mathcal{L}_h(r)\}$, where h is the total number of levels and is termed the depth.
- The level sets can be found using a breadth-first search that starts at the root r.

Level-based orderings: GPS

Algorithm (GPS algorithm to find pseudo-peripheral vertices)

1: Construct $\mathcal{L}(r)$ and set $flag = false$
2: **while** $flag = false$ **do**
3: $flag = true$
4: **for** $i = 1 : |\mathcal{L}(r)|$ **do**
5: $w_i \in \mathcal{L}(r)$ ▷ *Select vertex w_i from last level set*
6: **if** $flag = true$ **then**
7: Construct $\mathcal{L}(w_i)$
8: **if** $depth(\mathcal{L}(w_i)) > depth(\mathcal{L}(r))$ **then**
9: $flag = false$ ▷ *Flag that w_i will be used as new initial vertex*
10: **end if**
11: **end if**
12: **end for**
13: **if** $flag = true$ **then**
14: $s = r$ and $t = w_i$ ▷ *s has been chosen; while loop will terminate algorithm*
15: **else**
16: $r = w_i$
17: **end if**
18: **end while**

Level-based orderings

A simple example: starting with $r = 2$, after two passes through the while loop, the GPS algorithm returns $s = 8$ and $t = 1$ as pseudo-peripheral vertices.

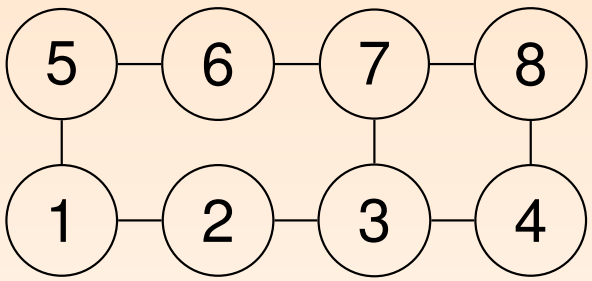


Figure: *An example to illustrate Algorithm 11.4 for finding pseudo-peripheral vertices. With root vertex* $r = 2$*, the first level set structure is* $\mathcal{L}(2) = \{\{2\}, \{1, 3\}, \{4, 5, 7\}, \{6, 8\}\}$*. Setting* $r = 8$ *at Step 16, the second level set structure is* $\mathcal{L}(8) = \{\{8\}, \{4, 7\}, \{3, 6\}, \{2, 5\}, \{1\}\}$ *and the algorithm terminates with* $s = 8$ *and* $t = 1$*.*

Spectral orderings

- The spectral algorithm associates a positive semidefinite Laplacian matrix L_p with the symmetric matrix A as follows:

$$(L_p)_{ij} = \begin{cases} -1 & \text{if } i \neq j \text{ and } a_{ij} \neq 0, \\ deg_{\mathcal{G}}(i) & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

- An eigenvector corresponding to the smallest positive eigenvalue of the Laplacian matrix is called a Fiedler vector. If $\mathcal{G}$ is connected, L_p is irreducible and the second smallest eigenvalue is positive.
- The vertices of $\mathcal{G}$ are ordered by sorting the entries of the Fiedler vector into monotonic order. Applying the permutation symmetrically to A yields the spectral ordering.

Spectral orderings

- The use of the Fiedler vector for reordering A comes from considering the matrix envelope with the size

$$|env(A)| = \sum_{i=1}^{n} \beta_i = \sum_{i=1}^{n} \max_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k).$$

The asymptotic upper bound on the operation count for the factorization based on $env(A)$ is

$$work_{env} = \sum_{i=1}^{n} \beta_i^2 = \sum_{i=1}^{n} \max_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k)^2.$$

Ordering the vertices using the Fiedler vector is closely related to minimizing $weight_{env}$ over all possible vertex reorderings, where

$$weight_{env} = \sum_{i=1}^{n} \sum_{\substack{k<i \\ (k,i)\in\mathcal{G}}} (i-k)^2.$$

Spectral orderings

- Thus, while minimizing the profile and envelope is related to the infinity norm, minimizing $weight_{env}$ is related to the Euclidean norm of the distance between graph vertices.
- Although computing the Fiedler vector can be computationally expensive it does have the advantage of easy vectorization and parallelization and the resulting ordering can give small profiles and low operation counts.

Local fill-reducing orderings for nonsymmetric $\mathcal{S}\{A\}$

- If $\mathcal{S}\{A\}$ is nonsymmetric then an often-used strategy is to apply the minimum degree algorithm (or one of its variants) or a band or profile-reducing ordering to the undirected graph $\mathcal{G}(A + A^T)$.
- This can work well if the symmetry index $s(A)$ is close to 1. But if A is highly nonsymmetric, another approach is required.
- Markowitz pivoting generalizes the MD algorithm by choosing the pivot entry based on vertex degrees computed directly from the nonsymmetric $\mathcal{S}\{A\}$; the result is a nonsymmetric permutation.

Markowitz pivoting

- At step k of the LU factorization, consider the $(n-k+1) \times (n-k+1)$ active submatrix, that is, the Schur complement $S^{(k)}$. Let $nz(row_i)$ and $nz(col_j)$ denote the number of entries in row i and column j of $S^{(k)}$ $(1 \leq i, j \leq n-k+1)$. Markowitz pivoting selects as the k-th pivot the entry of $S^{(k)}$ that minimizes the Markowitz count given by the product

$$(nz(row_i) - 1)(nz(col_j) - 1).$$

- It can be described using a sequence of bipartite graphs of the active submatrices but here we use a matrix-based description that permutes A on the fly.
- Markowitz pivoting is generally incorporated into the numerical factorization phase of an LU solver, rather than being used to derive an initial reordering of A.

Markowitz pivoting

Implementation of the algorithm requires access to the rows and the columns of the matrix.

Algorithm (Markowitz pivoting)

Input: *Matrix A with a nonsymmetric sparsity pattern.*
Output: *$A' = PAQ$, where P and Q are permutation matrices chosen to limit fill in.*

1: *Set $S^{(1)} = A$ and $A' = A$*
2: ***for*** $k = 1 : n - 1$ ***do***
3: *Compute $nz(row_i)$ and $nz(col_j)$ $(1 \leq i, j \leq n - k + 1)$*
4: *Find an entry $s_{ij}^{(k)}$ of $S^{(k)}$ that minimizes $(nz(row_i) - 1)(nz(col_j) - 1)$*
5: *Permute the rows and columns so that $s_{ij}^{(k)}$ is the $(1, 1)$ entry of the permuted $S^{(k)}$*
6: *Compute Schur complement $S^{(k+1)}$ of the permuted $S^{(k)}$ with respect to its $(1, 1)$ entry*
7: ***end for***

Markowitz pivoting

Example: the first pivot is a_{24} with Markowitz count 1; it does not cause fill-in. The second pivot has Markowitz count 2 in $S^{(2)}$; it results in one filled entry.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & * & * & & * \\ & * & & \circledast & \\ * & & * & & * \\ & * & * & * & \\ * & * & & & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 2 \\ 1 \\ 3 \\ 4 \\ 5 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} 4 & 1 & 2 & 3 & 5 \end{array} \\ \begin{pmatrix} * & & * & & \\ & * & * & * & * \\ & * & & * & * \\ * & & \circledast & * & \\ & * & * & & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 2 \\ 4 \\ 1 \\ 3 \\ 5 \end{array}\!\!\begin{array}{c} \begin{array}{ccccc} 4 & 2 & 1 & 3 & 5 \end{array} \\ \begin{pmatrix} * & * & & & \\ * & * & & * & \\ & * & * & * & * \\ & & * & * & * \\ & * & * & f & * \end{pmatrix} \end{array}$$

Figure: *Illustration of Markowitz pivoting. The first and second pivots are circled. The sparsity pattern of $A = S^{(1)}$ is on the left. In the centre is the sparsity pattern after permuting the pivot in position $(2, 4)$ to the $(1, 1)$ position of $S^{(1)}$. There is no fill-in after the first factorization step. On the right is the sparsity pattern after selecting the second pivot that has the original position $(4, 2)$ and permuting it to the $(1, 1)$ position of $S^{(2)}$. The resulting filled entry is denoted by f. Note that the nonsymmetric permutations transform the originally irreducible matrix into a reducible one.*

Reorderings

Local fill-reducing orderings for nonsymmetric $\mathcal{S}\{A\}$

- Markowitz pivoting as described here only considers the sparsity of A and the subsequent Schur complements.
- In practice, small pivots should be avoided.
- Practical implementations: relaxations: restriction to a limited number of rows and columns.
- Dynamic sparse formats like DS needed.

Global nested dissection orderings

- Nested dissection (ND) is the most important and widely-used global ordering strategy for direct methods when $\mathcal{S}\{A\}$ is symmetric; it is particularly effective for ordering very large matrices.
- Identifying a small set of vertices $\mathcal{V}_{\mathcal{S}}$ (known as a vertex separator)
- If removed separates the graph into two disjoint subgraphs described by the vertex subsets $\mathcal{B}$ and $\mathcal{W}$.
- The rows and columns belonging to $\mathcal{B}$ are labelled first, then those belonging to $\mathcal{W}$ and finally those in $\mathcal{V}_{\mathcal{S}}$. The reordered matrix has the form

$$\begin{pmatrix} A_{\mathcal{B},\mathcal{B}} & 0 & A_{\mathcal{B},\mathcal{V}_{\mathcal{S}}} \\ 0 & A_{\mathcal{W},\mathcal{W}} & A_{\mathcal{W},\mathcal{V}_{\mathcal{S}}} \\ A^T_{\mathcal{B},\mathcal{V}_{\mathcal{S}}} & A^T_{\mathcal{W},\mathcal{V}_{\mathcal{S}}} & A_{\mathcal{V}_{\mathcal{S}},\mathcal{V}_{\mathcal{S}}} \end{pmatrix}. \tag{69}$$

Global nested dissection orderings

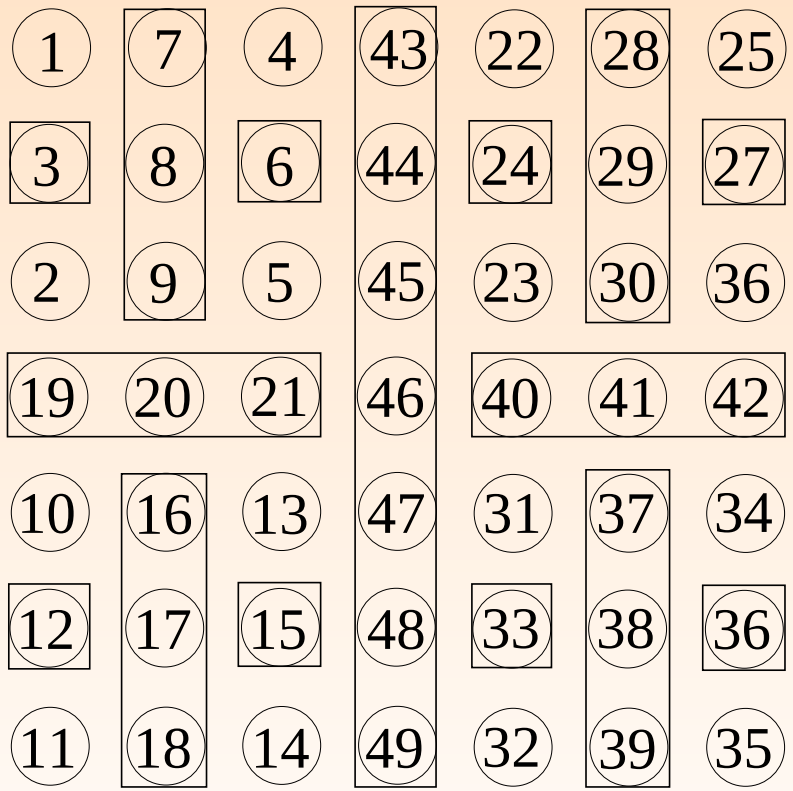
Definition

Vertex separator of an undirected $G = (V, E)$ is subset S of its vertices such that the subgraph induced by $V \setminus S$ has more components than G.

- Induced reordering

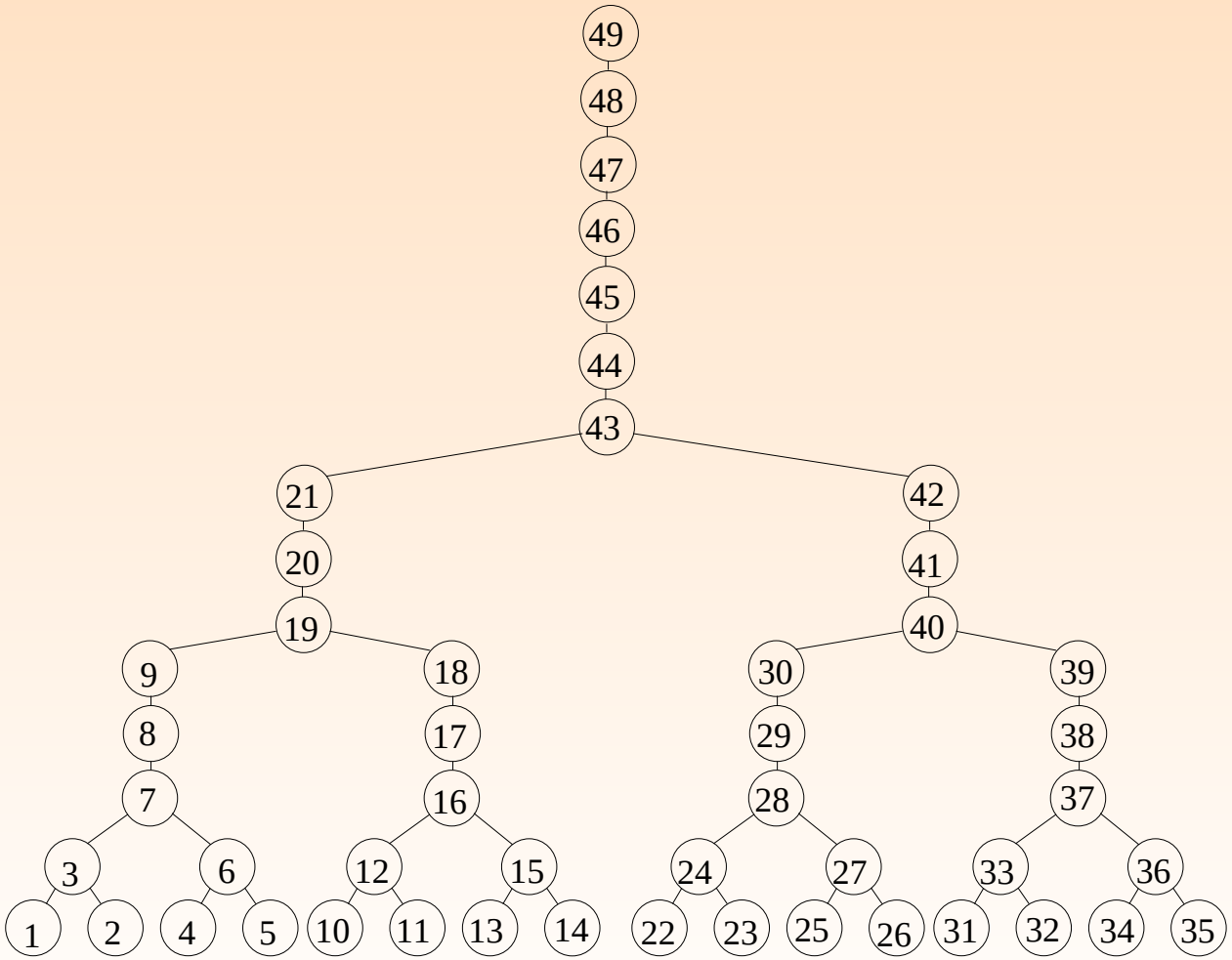
$$A = \begin{pmatrix} A_{11} & 0 & A_{31}^T \\ 0 & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \tag{70}$$

Global nested dissection orderings



Initial reordering

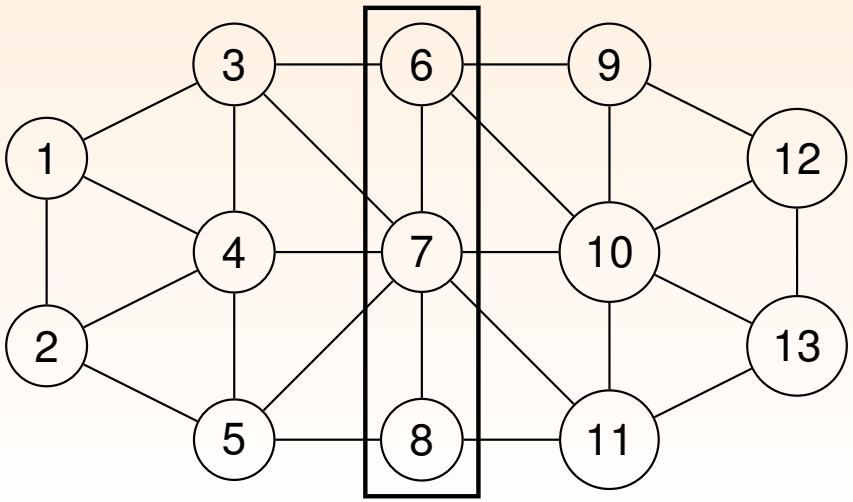
Global nested dissection orderings



Reorderings

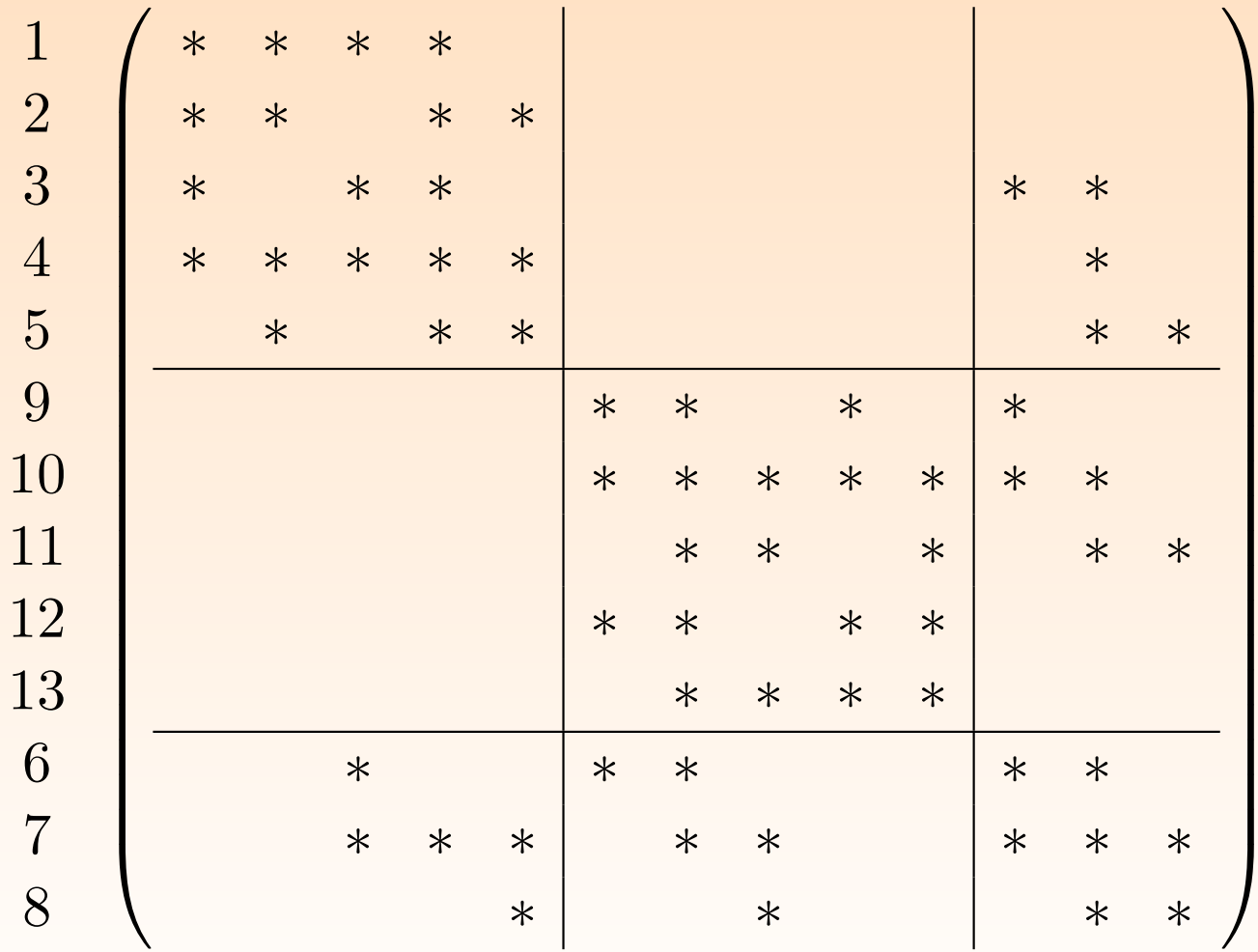
Global nested dissection orderings

$$
\begin{array}{c}
1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13
\end{array}
\begin{pmatrix}
* & * & * & * & & & & & & & & & \\
* & * & & * & * & & & & & & & & \\
* & & * & * & & * & * & & & & & & \\
* & * & * & * & * & & * & * & & & & & \\
& * & & * & * & & & * & & & & & \\
& & * & & & * & * & & * & * & & & \\
& & * & * & & * & * & * & & * & * & & \\
& & & * & * & & * & * & & & * & & \\
& & & & & * & & & * & * & & * & \\
& & & & & * & * & & * & * & * & * & * \\
& & & & & & * & * & & * & * & & * \\
& & & & & & & & * & * & & * & * \\
& & & & & & & & & * & * & * & *
\end{pmatrix}
$$



Reorderings

Global nested dissection orderings



Reorderings

Global nested dissection orderings

- Provided the variables are eliminated in the permuted order, no fill occurs within the zero off-diagonal blocks.
- If $|\mathcal{V}_{\mathcal{S}}|$ is small and $|\mathcal{B}|$ and $|\mathcal{W}|$ are similar, these zero blocks account for approximately half the possible entries in the matrix.
- The reordering can be applied recursively to the submatrices $A_{\mathcal{B},\mathcal{B}}$ and $A_{\mathcal{W},\mathcal{W}}$ until the vertex subsets are of size less than some prescribed threshold.
- Combinations with local reorderings.

Global nested dissection orderings

Algorithm (Nested dissection algorithm)

Input: *Graph $\mathcal{G}$ of a symmetrically structured matrix A and a partitioning algorithm PartitionAlg.*
Output: *A permutation vector p that defines a new labelling of the vertices of $\mathcal{G}$.*

1: ***recursive function*** *(p = nested_dissection(A, PartitionAlg))*
2: ***if*** *dissection has terminated* ***then*** ▷ *Vertex subsets are smaller than some threshold*
3: $p = AMD(\mathcal{V}, \mathcal{E})$ ▷ *Compute an AMD ordering*
4: ***else***
5: *Use PartitionAlg$(\mathcal{V}, \mathcal{E})$ to obtain the vertex partitioning $(\mathcal{B}, \mathcal{W}, \mathcal{V}_S)$*
6: $p_{\mathcal{B}}$ = nested_dissection($A_{\mathcal{B},\mathcal{B}}$, PartitionAlg)
7: $p_{\mathcal{W}}$ = nested_dissection($A_{\mathcal{W},\mathcal{W}}$, PartitionAlg)
8: $p_{\mathcal{V}_S}$ *is an ordering of $\mathcal{V}_S$*
9: *Set* $p = \begin{pmatrix} p_{\mathcal{B}} \\ p_{\mathcal{W}} \\ p_{\mathcal{V}_S} \end{pmatrix}$
10: ***end if***
11: ***end recursive function***

Bordered forms

- The matrix (69) is an example of a doubly bordered block diagonal (DBBD) form. More generally, a matrix is said in DBBD form if it has the block structure

$$A_{DB} = \begin{pmatrix} A_{1,1} & & & & C_1 \\ & A_{2,2} & & & C_2 \\ & & \ldots & & . \\ & & & A_{Nb,Nb} & C_{Nb} \\ R_1 & R_2 & \ldots & R_{Nb} & B \end{pmatrix}, \tag{71}$$

- The blocks can have very different sizes. A nested dissection ordering can be used to permute a symmetrically structured matrix A to a symmetrically structured DBBD form ($\mathcal{S}\{R_i\} = \mathcal{S}\{C_i^T\}$).
- If $\mathcal{S}\{A\}$ is close to symmetric then nested dissection can be applied to $\mathcal{S}\{A + A^T\}$. In finite-element applications, the DBBD form corresponds to partitioning the underlying finite-element domain into non-overlapping subdomains.

Bordered forms

- Coarse-grained parallel approaches aim to factorize the $A_{lb,lb}$ blocks in parallel before solving the interface problem that connects the blocks.
- The block factorization of A_{DB} is

$$A_{DB} = \begin{pmatrix} L_1 & & & & \\ & L_2 & & & \\ & & \ldots & & \\ & & & L_{Nb} & \\ \widehat{R}_1 & \widehat{R}_2 & \ldots & \widehat{R}_{Nb} & L_S \end{pmatrix} \begin{pmatrix} U_1 & & & & \widehat{C}_1 \\ & U_2 & & & \widehat{C}_2 \\ & & \ldots & & . \\ & & & U_{Nb} & \widehat{C}_{Nb} \\ & & & & U_S \end{pmatrix},$$

where

$$\widehat{R}_{lb} = R_{lb}U_{lb}^{-1},\ \widehat{C}_{lb} = L_{lb}^{-1}C_{lb}\ (1 \leq lb \leq Nb),\ L_S U_S = B - \sum_{lb=1}^{Nb} \widehat{R}_{lb}\widehat{C}_{lb}.$$

- Here, for simplicity, no permutations emphasized.

Bordered forms

Algorithm (Coarse-grained parallel LU factorization using DBBD form)

Input: *Matrix A_{DB} in DBBD form (71).*
Output: *Block LU factorization.*

Initialise $S = B$
for $lb = 1 : Nb$ ***do***
  $A_{lb,lb} = L_{lb}U_{lb}$ ▷ *LU factorization of square block on diagonal*
  $\widehat{R}_{lb} = R_{lb}U_{lb}^{-1}$ ▷ *Triangular solve for bottom-border blocks*
  $\widehat{C}_{lb} = L_{lb}^{-1}C_{lb}$ ▷ *Triangular solve for right-border blocks*
end for
$S = S - \sum_{lb=1}^{Nb} \widehat{R}_{lb}\widehat{C}_{lb}$ ▷ *Assemble updates to interface block*
$S = L_S U_S$ ▷ *Factorize updated interface block (Schur complement)*

Bordered forms

- Factorization of each individual $A_{lb,lb}$ and solve steps can be parallelized.
- The assembly of the interface block S and its LU can be partially parallelized.
- S is generally significantly denser than the other blocks.
- If A is not SPD then factorizing the $A_{lb,lb}$ blocks without considering the entries in the border can potentially lead to stability problems. Consider the first step in factorizing $A_{lb,lb}$ and the threshold pivoting test for a sparse LU factorization. The pivot candidate $(A_{lb,lb})_{11}$ must satisfy

$$\max\{\max_{i>1}|(A_{lb,lb})_{i1}|, \max_{k}|(R_{lb})_{k1}|\} \leq \gamma^{-1}|(A_{lb,lb})_{11}|,$$

  where $\gamma \in (0, 1]$ is the threshold parameter.
- Large entries in the row border matrix R_{lb} can prevent pivots being selected within $A_{lb,lb}$.

Singly bordered form

- Singly bordered block diagonal (SBBD) form

$$A_{SB} = \begin{pmatrix} A_{1,1} & & & C_1 \\ & A_{2,2} & & C_2 \\ & & \cdots & . \\ & & A_{Nb,Nb} & C_{Nb} \end{pmatrix},$$

$A_{lb,lb}$ are rectangular $m_{lb} \times n_{lb}$. The linear system becomes

$$\begin{pmatrix} A_{1,1} & & & C_1 \\ & A_{2,2} & & C_2 \\ & & \cdots & . \\ & & A_{Nb,Nb} & C_{Nb} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_{Nb} \\ x_I \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{Nb} \end{pmatrix}, \qquad (72)$$

x_{lb} is of length n_{lb}, x_I is a vector of length n_I of interface variables, and the right-hand side vectors b_{lb} are of length m_{lb}, such that

$$\begin{pmatrix} A_{lb,lb} & C_{lb} \end{pmatrix} \begin{pmatrix} x_{lb} \\ x_I \end{pmatrix} = b_{lb}, \quad 1 \leq lb \leq Nb.$$

Singly bordered form

- A partial factorization of each block matrix is performed, that is,

$$\begin{pmatrix} A_{lb,lb} & C_{lb} \end{pmatrix} = P_{lb} \begin{pmatrix} L_{lb} & \\ \bar{L}_{lb} & I \end{pmatrix} \begin{pmatrix} U_{lb} & \bar{U}_{lb} \\ & S_{lb} \end{pmatrix} Q_{lb}, \tag{73}$$

- Pivots can only be chosen from the columns of $A_{lb,lb}$ because the columns of C_{lb} have entries in at least one other border block C_{jb} ($jb \neq lb$).
- The pivot candidate $(A_{lb,lb})_{11}$ at the first elimination step must satisfy

$$\max_{i>1} |(A_{lb,lb})_{i1}| \leq \gamma^{-1} |(A_{lb,lb})_{11}|,$$

Singly bordered form

Algorithm (Coarse-grained parallel LU factorization and solve using SBBD form)

Input: *Linear system in SBBD form (72).*
Output: *Block LU factorization and computed solution* x*.*

1: $S = 0$ *and* $z_I = 0$
2: ***for*** $lb = 1 : Nb$ ***do***
3: *Perform a partial* LU *factorization (73) of* $(A_{lb,lb},\ C_{lb})$*.*
4: *Solve* $P_{lb} \begin{pmatrix} L_{lb} & \\ \bar{L}_{lb} & I \end{pmatrix} \begin{pmatrix} y_{lb} \\ \bar{y}_{lb} \end{pmatrix} = b_{lb}$
5: $S = S + S_{lb}$ *and* $z_I = z_I + \bar{y}_{lb}$ ▷ *Assemble* S *and* z_I
6: ***end for***
7: $S = P_s L_s U_s Q_s$ ▷ P_s *and* Q_s *are permutation matrices*
8: *Solve* $P_s L_s\, y_I = z_I$ *and then* $U_s Q_s\, x_I = y_I$ ▷ *Forward then back substitution*
9: ***for*** $lb = 1 : Nb$ ***do***
10: *Solve* $U_{lb}\, Q_{lb}\, x_{lb} = y_{lb} - \bar{U}_{lb}\, Q_{lb}\, x_I$
11: ***end for***

Ordering to singly bordered form

- The objective is to permute A to an SBBD form with a narrow column border.
- One way to do this is to choose the number $Nb > 1$ of required blocks and use ND to compute a vertex separator $\mathcal{V}_\mathcal{S}$ of $\mathcal{G}(A + A^T)$ such that removing $\mathcal{V}_\mathcal{S}$ and its incident edges splits $\mathcal{G}(A + A^T)$ into Nb components.
- Then initialise the set $\mathcal{S}_\mathcal{C}$ of border columns to $\mathcal{V}_\mathcal{S}$ and let $\mathcal{V}_{1b}, \mathcal{V}_{2b}, \ldots, \mathcal{V}_{Nb}$ be the subsets of column indices of A that correspond to the Nb components and let $n_{i,kb}$ be the number of column indices in row i that belong to $\mathcal{V}_{kb}$. If $lb = \arg\max_{1 \leq kb \leq Nb} |n_{i,kb}|$ then row i is assigned to partition lb. All column indices in row i that do not belong to $\mathcal{V}_{lb}$ are moved into $\mathcal{S}_\mathcal{C}$.

Ordering to singly bordered form

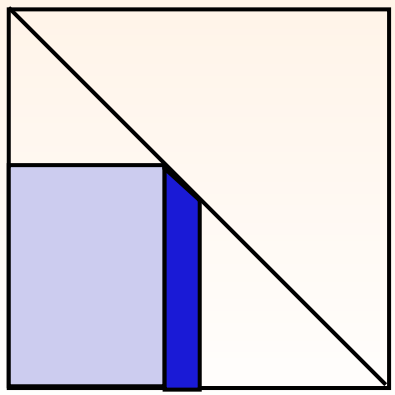
Algorithm (SBBD ordering of a general matrix)

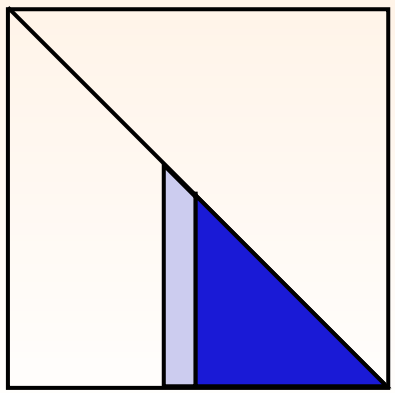
Input: *Matrix A, the number $Nb > 1$ of blocks, vertex separator $\mathcal{V}_S$ of $\mathcal{G}(A + A^T)$.*
Output: *Vector $block$ such that $block(i)$ denotes the partition in the SBBD form to which row i is assigned ($1 \leq i \leq n$) and $\mathcal{S}_C$ is the set of border columns.*

1: *Initialise $\mathcal{S}_C = \mathcal{V}_S$ and $block(1:n) = 0$*
2: *Initialise $\mathcal{V}_{kb}$ to hold the column indices of A that correspond to component kb of $\mathcal{G}(A + A^T)$ after the removal of $\mathcal{V}_S$, $1 \leq kb \leq Nb$*
3: ***for*** *each row i* ***do***
4: *Add up the number $n_{i,kb}$ of column indices belonging to $\mathcal{V}_{kb}$, $1 \leq kb \leq Nb$*
5: *Find $lb = \arg\max_{1 \leq kb \leq Nb} n_{i,kb}$*
6: *$block(i) = lb$*
7: ***for*** *each column index j in row i* ***do***
8: ***if*** *$j \in \mathcal{V}_{kb}$ and $kb \neq lb$* ***then***
9: *Remove j from $\mathcal{V}_{kb}$ and add to $\mathcal{S}_C$*
10: ***end if***
11: ***end for***
12: ***end for***
13: *Assign the rows i for which $block(i) = 0$ equally between the Nb partitions.*
14: *If some column $j \in \mathcal{S}_C$ has nonzero entries only in rows belonging to partition kb then*

Complexity

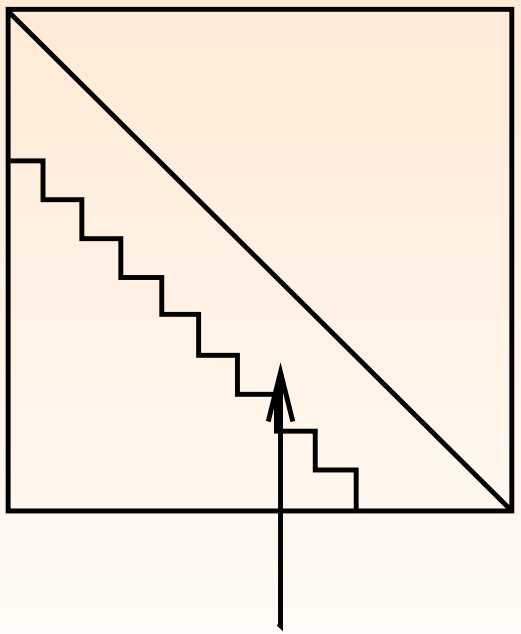
- Overall time dominated by time for the factorization
- General dense matrices
  - Space: $O(n^2)$
  - Time: $O(n^3)$
- General sparse matrices
  - Space: $\eta(L) = n + \sum_{i=1}^{n-1}(\eta(L_{*i}) - 1)$
  - Time in the i-th step: $\eta(L_{*i}) - 1$ divisions, $1/2(\eta(L_{*i}) - 1)\eta(L_{*i})$ multiple-add pairs
  - Time totally: $1/2 \sum_{i=1}^{n-1}(\eta(L_{*i}) - 1)(\eta(L_{*i}) + 2)$





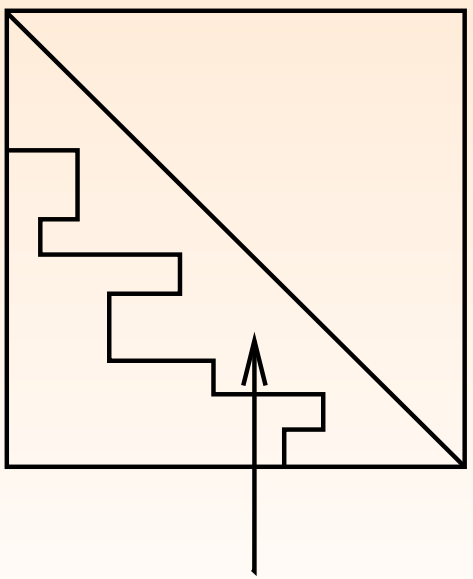
Complexity

- Band schemes ($\beta << n$)
  - Space: $O(\beta n)$
  - Time: $O(\beta^2 n)$



Complexity

- Profile/envelope schemes
  - Space: $\sum_{i=1}^{n} \beta_i$
  - Frontwidth: $\omega_i(A) = |\{k | k > i \wedge a_{kl} \neq 0 \; for \; some \; l \leq i\}|$
  - Time: $1/2 \sum_{i=1}^{n-1} \omega_i(A)(\omega_i(A) + 3)$

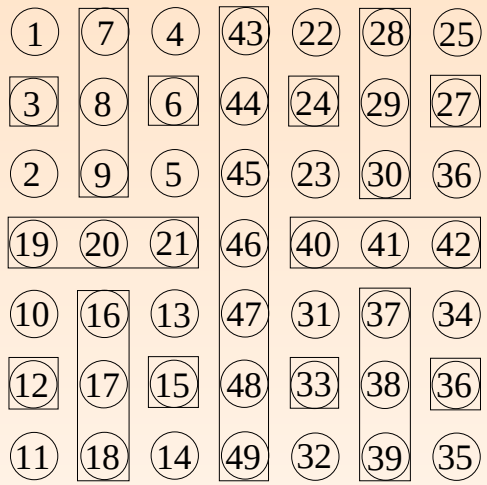


Profile (Envelope)

From direct to iterative methods

Complexity

- General sparse schemes can be analyzed in some cases
  - Nested dissection

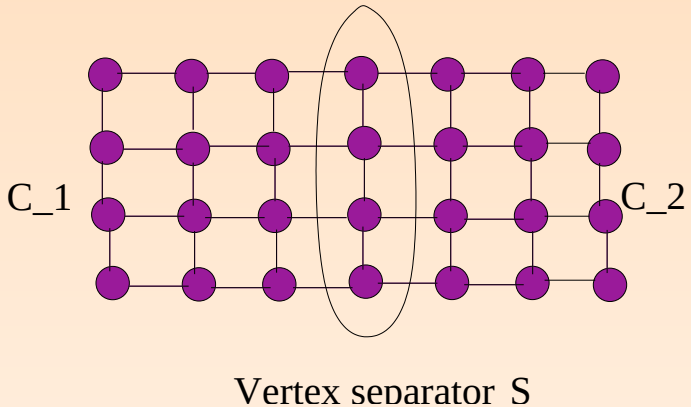


Definition

(α, σ) separation of a graph with n vertices: each its subgraph can be separated by a vertex separator S such that its size is of the order $O(n^{\sigma})$ and the separated subgraphs components have sizes $\leq \alpha n, 1/2 \leq \alpha < 1$.

From direct to iterative methods

Complexity: Generalized nested dissection



- Planar graphs, 2D finite element graphs (bounded degree)
  - $\sigma = 1/2$, $\alpha = 2/3$
  - Space: $O(n \log n)$
  - Time: $O(n^{3/2})$
- 3D Finite element graphs
  - $\sigma = 2/3$
  - Space: $O(n^{4/3})$
  - Time: $O(n^2)$
- Lipton, Rose, Tarjan (1979), Teng (1997).

Outline

Algebraic preconditioning

- Finite precision arithmetic: computed factors are not exact.
- Moreover, the effort to obtain more accurate results can lead to complex coding and unavoidable inefficiencies magnified by modern computer architectures.
- Potential solution: intentionally relaxing the required accuracy of the computed factors.
- Simpler, cheaper, sparser approximate factorization of A (or of A^{-1}): preconditioners.
- Using the preconditioner in combination with an iterative solver.
- an approximate factorization called an incomplete factorization to distinguish it from a complete factorization of a direct method.
- Used with iterative methods for solving $Ax = b$ from their two main classes: stationary (relaxation) iterative methods and Krylov subspace methods.

Stationary iterative methods

- Stationary iterative methods work by splitting A as follows:

$$A = M - N,$$

- The matrix M is chosen to be nonsingular and easy to invert. An initial guess $x^{(0)}$, the iterations are then given by

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b, \quad k = 0, 1, \ldots \tag{74}$$

This can be rewritten as

$$x^{(k+1)} = x^{(k)} + M^{-1}(b - Ax^{(k)}) = x^{(k)} + M^{-1}\, r^{(k)}, \quad k = 0, 1, \ldots \tag{75}$$

where the vector $r^{(k)} = b - Ax^{(k)}$ is the residual on the k-th iteration.

Stationary iterative methods

- By substituting $b = r^{(k)} + Ax^{(k)}$ into $x = A^{-1}\, b$, we obtain

$$x = A^{-1}(r^{(k)} + Ax^{(k)}) = x^{(k)} + A^{-1}\, r^{(k)}.$$

  If M is used to approximate A, we again get the iteration above.

- Further

$$r^{(k+1)} = b - A(x^{(k)} + M^{-1}\, r^{(k)}) = (I - AM^{-1})\, r^{(k)} = \ldots = (I - AM^{-1})^{k} \quad (76)$$

  and if $e^{(k)} = x - x^{(k)}$ is the error vector on iteration k then

$$e^{(k+1)} = M^{-1}N\, e^{(k)} = \ldots = (M^{-1}N)^{k+1}\, e^{(0)} = (I - M^{-1}A)^{k+1}\, e^{(0)}. \quad (77)$$

- The matrix $I - M^{-1}A$ or $I - AM^{-1}$ is called the iteration matrix. In general, monitoring residuals is more practical.

Stationary iterative methods

Theorem

For any initial $x^{(0)}$ and vector b, the stationary iteration converges if and only if the spectral radius of $(I - M^{-1}A)$ is less than unity.

Proof.

The spectral radius of an $n \times n$ matrix C with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_n$ is defined to be

$$\rho(C) = \max\{|\lambda_i| \mid 1 \leq i \leq n\}. \tag{78}$$

Furthermore, the sequence of matrix powers C^k, $k = 0, 1, \ldots$, converges to zero if and only if $\rho(C) < 1$. It follows from (77) that if the spectral radius of $(I - M^{-1}A)$ is less than unity then the iteration (74) converges for any $x^{(0)}$ and b. Conversely, the relation

$$x^{(k+1)} - x^{(k)} = (I - M^{-1}N)(x^{(k)} - x^{(k-1)}) = \ldots = (I - M^{-1}N)^k M^{-1}(b - Ax^{(0)})$$

shows that if the iteration converges for any $x^{(0)}$ and b then $(I - M^{-1}N)^k v$ converges to zero for any v. Consequently, $\rho(I - M^{-1}A)$ must be less than unity, and the result follows. □ □

Algebraic preconditioning

Stationary iterative methods

- Generally impractical to compute the spectral radius and sufficient conditions that guarantee convergence are used.
- A sufficient condition for convergence is $\|I - M^{-1}A\| < 1$.
- Consider splitting (diagonal, strict lower triangular, strict upper triangular)

$$A = D_A + L_A + U_A. \tag{79}$$

- For $\omega > 0$ is a scalar parameter, classical methods include:
  - Richardson method: $M = \omega^{-1} I$,
  - Jacobi and damped Jacobi methods: $M = D_A$ and $M = \omega^{-1} D_A$,
  - Gauss-Seidel and SOR methods: $M = D_A + L_A$ and $M = \omega^{-1} D_A + L_A$.

Algebraic preconditioning

Stationary iterative methods

Theorem

If $A \in \mathbb{R}^{n \times n}$ is strongly diagonally dominant then Jacobi method and Gauss-Seidel method are convergent.

Theorem

If $A \in \mathbb{R}^{n \times n}$ is symmetric with positive diagonal D_A then the Jacobi method is convergent iff A and $2D_A - A$ are positive definite.

Theorem

If $A \in \mathbb{R}^{n \times n}$ is symmetric and positive definite then the Gauss-Seidel method is convergent.

Algebraic preconditioning

Krylov subspace methods

- Non-stationary iterative methods are of the form

$$x^{(k+1)} = x^{(k)} + \omega^{(k)} M^{-1} r^{(k)}, \quad k = 0, 1, \ldots.$$

  where the $\omega^{(k)}$ are scalars.
- In this class, Krylov subspace methods are the most effective.
- Given a vector y, the k-th Krylov subspace $\mathcal{K}^{(k)}(A, y)$ generated by A from the vector y is defined to be

$$\mathcal{K}^{(k)}(A, y) = span(y, Ay, \ldots, A^{k-1} y).$$

- Generate a sequence of approximate solutions $x^{(k)} \in x^{(0)} + \mathcal{K}^{(k)}(A, r^{(0)})$ such that the norm of the corresponding residuals $r^{(k)} \in \mathcal{K}^{(k+1)}(A, r^{(0)})$ converge to zero.
- SPD systems the conjugate gradient method (CG); nonsymmetric systems: GMRES, BiCG, no single method of choice.
- At each iteration only matrix-vector products with A (and possibly with A^T) required.

Algebraic preconditioning

Krylov subspace methods

- Powerful, if (and only if) combined with a preconditioner: the most widely-used class of preconditioned iterative methods.
- Unfortunately, for a given A, b and $x^{(0)}$, it is usually not possible to predict the rate of convergence.
- If A is a SPD matrix then for CG

$$\|x - x^{(k)}\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^k \|x - x^{(0)}\|_A,$$

  where $\kappa(A)$ is the spectral condition number.
- Can be highly pessimistic. Does not show the potential for CG to converge superlinearly or that the rate of convergence depends on the distribution of all the eigenvalues of A.

Algebraic preconditioning

Krylov subspace methods

- For non-SPD matrices, less is known. Like to emphasize favourable properties: the minimal residual method (MINRES) for solving symmetric indefinite systems in exact arithmetic, has the norm of the residual monotonically decreasing.
- No general descriptive convergence theory is available for Krylov subspace methods for nonsymmetric systems (including GMRES).

Algebraic preconditioning

Krylov subspace methods

- Preconditioning corresponds to the application of a matrix (or linear operator) to the original linear system to yield a different linear system that has more favourable properties. Consider the preconditioned linear system

$$M^{-1}Ax = M^{-1}b. \tag{80}$$

Here M^{-1} is applied to A from the left. We say that A is preconditioned from the left and M is a left preconditioner. Analogously, the linear system can be preconditioned from the right

$$AM^{-1}y = b, \qquad x = M^{-1}y. \tag{81}$$

Krylov subspace methods

The following result states that it is not possible to determine a priori which variant is the best.

Theorem

Let δ and Δ be positive numbers. Then for any $n \geq 3$ there exist nonsingular $n \times n$ matrices A and M such that all the entries of $M^{-1}A - I$ have absolute value less than δ and all the entries of $AM^{-1} - I$ have absolute values greater than Δ.

- The choice between left and right preconditioning may be based on properties of the coupling of the preconditioner with the iterative method or on the distribution of the eigenvalues of A.

Algebraic preconditioning

Krylov subspace methods

- The computed quantities readily available during a preconditioned iterative method depend on how the preconditioner is applied and this may influence the choice. These quantities may be used, for example, to decide when to terminate the iterations.
- An obvious advantage of right preconditioning is that in exact arithmetic, the residuals for the right preconditioned system are identical to the true residuals, enabling convergence to be monitored accurately.
- In some cases, the numerical properties of an implementation and/or the computer architecture may also play a part.

Krylov subspace methods

- For M in factorized form $M = M_1 M_2$, two-sided (or split) preconditioning is an option. The iterative method then solves the transformed system

$$M_1^{-1} A M_2^{-1}\, y = M_1^{-1}\, b, \qquad x = M_2^{-1}\, y. \tag{82}$$

- If A and M are SPD matrices then we would like the preconditioned matrix $M_1^{-1} A M_1^{-T}$ to be SPD. However, it is not necessary to use a two-sided transformation with the preconditioned conjugate gradient (PCG) method because it can be formulated using the M-inner product in which the matrix $M^{-1}A$ is self-adjoint.

Krylov subspace methods

Theorem

Let A and M be SPD matrices. Then $M^{-1}A$ is self-adjoint in the M-inner product.

Proof.

Self-adjointness is implied by the following chain of equivalences.

$$\langle M^{-1}Ax, y\rangle_M = \langle Ax, y\rangle = \langle x, Ay\rangle = \langle x, MM^{-1}Ay\rangle = \langle Mx, M^{-1}Ay\rangle = \langle x, M^{-1}Ay\rangle_M.$$

□ □

- Left preconditioned CG with the M-inner product is mathematically equivalent to right preconditioned CG with the M^{-1}-inner product.
- If A is symmetric but not PD, the PCG method can breakcdown (division by a zero quantity).

Algebraic preconditioning

Krylov subspace methods

- An obvious goal: to achieve rapid convergence.
- The preconditioner should aim to reduce the condition number, but this is not necessarily sufficient to give fast convergence.
- For general matrices, despite the lack of theoretical guarantees regarding convergence, many useful preconditioners motivated by bounding the condition number of the preconditioned matrix.
- Choosing a preconditioner is often based on how costly it is to compute and on some indicators that potentially reflect its quality.
- In particular, the accuracy of a preconditioner M can be assessed using the norm of the error matrix

$$\|E\| = \|M - A\|,$$

and its stability can be measured using

$$\|M^{-1}E\| = \|I - M^{-1}A\| \quad \text{or} \quad \|EM^{-1}\| = \|I - AM^{-1}\|.$$

- In some cases, the inverse M^{-1} is computed directly. In this case we have an approximate inverse preconditioner.

Algebraic preconditioning

Simple preconditioners

- The simplest preconditioner consists of the diagonal of the matrix $M = D_A$. This is known as the (point) Jacobi preconditioner.
- Block versions can be derived by partitioning $\mathcal{V} = \{1, 2, \ldots, n\}$ into mutually disjoint subsets $\mathcal{V}_1, \ldots, \mathcal{V}_l$ and then setting

$$m_{ij} = \begin{cases} a_{ij} & \text{if } i \text{ and } j \text{ belong to the same subset } \mathcal{V}_k \text{ for some } k, 1 \\ 0 & \text{otherwise.} \end{cases}$$

- The SSOR preconditioner, like the Jacobi preconditioner, can be derived from A without any work. If A is symmetric then using the notation (79), the SSOR preconditioner is defined to be

$$M = (D_A + L_A) D_A^{-1} (D_A + L_A)^T, \tag{83}$$

or, using a parameter $0 < \omega < 2$, as

$$M = \frac{1}{2-\omega} (\frac{1}{\omega} D_A + L_A)(\frac{1}{\omega} D_A)^{-1} (\frac{1}{\omega} D_A + L_A)^T.$$

- Finding optimal value of ω typically expensive.

The Eisenstat trick

- Generally cheaper to apply M^{-1} and A separately,
- But: consider the matrix splitting (79) and let M be given by

$$M = (D + L_A)\,[D^{-1}(D + U_A)] = M_1\,M_2, \tag{84}$$

  where D is a nonsingular diagonal matrix.
- The SSOR matrix is one example in the symmetric case but more generally $D \neq D_A$.
- Using two-sided preconditioning, (82) becomes

$$A'y = M_1^{-1}AM_2^{-1}\,y = (D+L_A)^{-1}A[D^{-1}(D+U_A)]^{-1}\,y = (D+L_A)^{-1}b. \tag{85}$$

- Setting

$$\bar{L} = D^{-1}L_A, \quad \bar{U} = D^{-1}U_A, \quad \bar{A} = D^{-1}A, \quad \text{and} \quad \bar{b} = (I + \bar{L})^{-1}D^{-1}\,b,$$

  we obtain

$$\begin{aligned} A' &= (D + L_A)^{-1}A[D^{-1}(D + U_A)]^{-1} = [(D + L_A)^{-1}D]D^{-1}A[D^{-1}(D + U_A)]^{-1} \\ &= [D^{-1}(D + L_A)]^{-1}D^{-1}A(I + D^{-1}U_A)^{-1} = (I + \bar{L})^{-1}\bar{A}(I + \bar{U})^{-1}. \end{aligned}$$

The Eisenstat trick

- That is, the system becomes

$$A'y = (I + \bar{L})^{-1}\bar{A}(I + \bar{U})^{-1}\, y = (I + \bar{L})^{-1} D^{-1}\, b = \bar{b}. \qquad (86)$$

  If y solves (86) then the solution x of $(I + \bar{U})\, x = y$ solves $Ax = b$.
- Further

$$\begin{aligned} A' &= (I + \bar{L})^{-1}\,(I + \bar{L} + D^{-1}D_A - 2I + I + \bar{U})(I + \bar{U})^{-1} \\ &= (I + \bar{L})^{-1}\,[(I + \bar{L})(I + \bar{U})^{-1} + (D^{-1}D_A - 2I)(I + \bar{U})^{-1} + I] \\ &= (I + \bar{U})^{-1} + (I + \bar{L})^{-1}\,[(D^{-1}D_A - 2I)(I + \bar{U})^{-1} + I]. \end{aligned}$$

- Thus to compute $z = A'w = (I + \bar{L})^{-1}\bar{A}(I + \bar{U})^{-1}w$ for a given w, it is necessary only to solve two triangular systems

$$(I + \bar{U})\, z_1 = w \quad \text{followed by} \quad (I + \bar{L})\, z_2 = (D^{-1}D_A - 2I)\, z_1 + w,$$

  and then set $z = z_1 + z_2$.
- This trick is not a preconditioner: it is a way of applying the preconditioner of special shape.

Algebraic preconditioning

Some special classes of matrices

- The development of algebraic preconditioners has been closely connected to solving linear systems from the discretization of PDEs.
- Two-dimensional Poisson problem, discretized using a uniform regular grid, finite differences, zero Dirichlet conditions on the boundary, natural ordering.

$$A = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ -1 & 4 & -1 & & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ -1 & & & 4 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}. \tag{87}$$

- If the spatial discretization on the domain is characterized by the mesh parameter h then the size of A is inversely proportional to h.
- $\kappa(A)$ depends asymptotically on h^{-2}.

Algebraic preconditioning

Some special classes of matrices

- Matrices with similar banded sparsity patterns with nonzeros on only a small number of subdiagonals arise from simple finite difference or finite element discretizations of other partial differential equations.
- Particular cases of special classes of matrices help to describe the theoretical background behind the discretized systems.
- Let the off-diagonal entries of the nonsingular matrix A be nonpositive (that is, $a_{ij} \leq 0$ for all $i \neq j$). Then A is a (nonsingular) M-matrix if one of the following holds:
  - $A + D$ is nonsingular for any diagonal matrix D with nonnegative entries;
  - all the entries of A^{-1} are nonnegative;
  - all principal minors of A are positive.
- The matrix above is an example of an M-matrix. A symmetric M-matrix is known as a Stieltjes matrix, and such a matrix is positive definite.

Some special classes of matrices

- The class of nonsingular H-matrices includes matrices coming from simple discretizations of convection-diffusion problems. The comparison matrix $C(A)$ of A is defined to have entries

$$C(A)_{ij} = \begin{cases} -|a_{ij}|, & i \neq j, \\ |a_{ij}|, & i = j. \end{cases}$$

- If $C(A)$ is a nonsingular M-matrix then A is a nonsingular H-matrix.
- A is diagonally dominant by rows if

$$\sum_{j=1,\, j\neq i}^{n} |a_{ij}| \leq |a_{ii}|, \quad 1 \leq i \leq n. \tag{88}$$

- A is strictly diagonally dominant by rows if strict inequality holds for all i. A is (strictly) diagonally dominant by columns if A^T is (strictly) diagonally dominant by rows.

Some special classes of matrices

- A is said to be irreducibly diagonally dominant if it is irreducible and the inequalities are satisfied with strict inequality for at least one row i. If A is strictly diagonally dominant by rows or columns or is irreducibly diagonally dominant then it is nonsingular and factorizable.
- The class of diagonally dominant matrices is closely connected to that of nonsingular H-matrices. For example, the property that there exists a diagonal matrix D with positive entries such that AD is strictly diagonally dominant is equivalent to A being a nonsingular H-matrix.

Some special classes of matrices: once more

- Still theoretical assumptions are rather strong.
- Concept of special matrices

Theorem

Matrix A is called a **regular M-matrix** *if $a_{ij} \leq 0, i \neq j$, is regular and $A^{-1} \geq 0$.*

Theorem

A is a **H-matrix** *if $B = |D_A| - |A - D_A|$ is an M-matrix.*

- Many equivalent definitions

Introduction to incomplete factorizations

- Incomplete factorizations fall into three main classes:
  - Threshold-based methods: locations of permissible fill-in are determined in conjunction with the numerical factorization of A; entries of the computed factors of absolute value greater than a prescribed threshold $\tau > 0$ are dropped.
  - Memory-based methods in which the amount of memory available for the incomplete factorization is prescribed and only the largest entries in each row (or column) are retained.
  - Structure-based methods: an initial symbolic factorization phase determines the location of permissible entries using only $\mathcal{S}\{A\}$. This allows the memory requirements to be determined before an incomplete numerical factorization is performed. The specified set of positions is called the target sparsity pattern.

Algebraic preconditioning

Introduction to incomplete factorizations

- The basic dropping approaches can be combined and employed in conjunction with sparsifying A before the factorization commences.
- Sparsification of A after permuting reveals a block structure.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} * & * & * & & & \\ * & * & f & * & * & * \\ * & f & * & f & f & f \\ & * & f & * & * & * \\ & * & f & * & * & * \\ & * & f & * & * & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 2 \\ 4 \\ 1 \\ 3 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 2 & 4 & 1 & 3 & 5 & 6 \end{array} \\ \begin{pmatrix} * & * & * & & * & * \\ * & * & f & & * & * \\ * & f & * & * & f & f \\ & & * & * & f & f \\ * & * & f & f & * & * \\ * & * & f & f & * & * \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ 2 \\ 4 \\ 1 \\ 3 \\ 5 \\ 6 \end{array}\!\!\begin{array}{c} \begin{array}{cccccc} 2 & 4 & 1 & 3 & 5 & 6 \end{array} \\ \begin{pmatrix} * & * & & & * & * \\ * & * & & & * & * \\ & & * & * & & \\ & & * & * & & \\ * & * & & & * & * \\ * & * & & & * & * \end{pmatrix} \end{array}$$

Figure: Illustration of matrix sparsification. f denotes filled entries in the factors. On the left is the original matrix A with its filled entries; in the centre is the permuted matrix with its filled entries; on the right is the sparsified permuted matrix after dropping the entries of A in positions $(1,3)$ and $(3,1)$ (it has no filled entries).

Incomplete factorization breakdown

- Dropping entries can lead to breakdown of the incomplete factorization, that is, a zero pivot may be encountered during the factorization (or a non positive pivot in the Cholesky case).
- It is only possible to predict when this will happen in special cases, as stated in the following theorem, which is a consequence of the fact that being an M-matrix or an H-matrix is preserved in the sequence of the Schur complements during the factorization.

Theorem

Let A be a nonsingular M-matrix or H-matrix. If the target sparsity pattern of the incomplete factors contains the positions of the diagonal entries then the incomplete factorization of A does not break down.

Incomplete factorization breakdown

- To illustrate the error accumulation in the incomplete factorization of an M-matrix using dropping, consider the following example.
- Let E be the error matrix. E is initialised to zero and at each stage of the factorization the dropped entries are added into it.
- After one step of the complete factorization of A the partially eliminated matrix $A^{(2)}$ is

$$A^{(2)} = \begin{pmatrix} 4 & -1 & & -1 & & & & & \\ & 3.75 & -1 & -0.25 & -1 & & & & \\ & -1 & 4 & & & -1 & & & \\ & -0.25 & & 3.75 & -1 & & -1 & & \\ & -1 & & -1 & 4 & -1 & & -1 & \\ & & -1 & & -1 & 4 & & & -1 \\ & & & -1 & & & 4 & -1 & \\ & & & & -1 & & -1 & 4 & -1 \\ & & & & & -1 & & -1 & 4 \end{pmatrix}.$$

Incomplete factorization breakdown

- Suppose the filled entries -0.25 in positions $(2, 4)$ and $(4, 2)$ are dropped. Then the values of the corresponding diagonal entries in the subsequent elimination matrices are larger than they would have been without any dropping.
- Furthermore, as all the off-diagonal nonzero entries are negative, for any target sparsity pattern the dropped entries are negative.
- The M-matrix property applies to all subsequent Schur complements, which implies that all the entries added into E are negative and so the absolute values of the entries in E grow as the factorization proceeds (the contributions can never cancel each other out).
- Thus, although the factorization does not break down, the growth in the error is potentially a problem for the accuracy of an incomplete factorization of an M-matrix.
- Modifying the diagonal entries of A is a common approach to avoid breakdown in an incomplete factorization.

Algebraic preconditioning

Incomplete factorization breakdown

$$A = \begin{pmatrix} 3 & -2 & & 2 \\ -2 & 3 & -2 & \\ & -2 & 3 & -2 \\ 2 & & -2 & 8 \end{pmatrix}, \; L = \begin{pmatrix} 1 & & & \\ -2/3 & 1 & & \\ & -6/5 & 1 & \\ 2/3 & 4/5 & -2/3 & 1 \end{pmatrix}, \; D = \begin{pmatrix} 3 & & & \\ & 5/3 & & \\ & & 3/5 & \\ & & & 16/3 \end{pmatrix}.$$

$$\widetilde{L} = \begin{pmatrix} 1 & & & \\ -2/3 & 1 & & \\ & -6/5 & 1 & \\ 2/3 & & -10/3 & 1 \end{pmatrix}, \; \widetilde{D} = \begin{pmatrix} 3 & & & \\ & 5/3 & & \\ & & 3/5 & \\ & & & 0 \end{pmatrix}.$$

Figure: An example to illustrate breakdown. The matrix A and its square-root free factors are given together with the incomplete factors $\widetilde{L}$ and $\widetilde{D}$ that result from dropping the entry l_{24} during the factorization. $\tilde{d}_{44} = 0$ means the incomplete factorization has broken down.

- Remedy: perturb the diagonal value causing the breakdown.
- Practice of making simple ad hoc modifications not very positive. If breakdown (or near-breakdown) occurs, it may be too late.

Incomplete factorization breakdown

- An alternative and more effective strategy to avoid breakdown is to modify all the diagonal entries of A a priori and then compute an incomplete factorization of $A + \alpha I$, where the shift $\alpha > 0$ is a scalar parameter.
- It is always possible to find α such that $A + \alpha I$ is nonsingular and diagonally dominant and is thus an H-matrix.
- However, being an H-matrix is not a necessary condition for a matrix to be factorizable and, in practice, much smaller values of α can provide incomplete factorizations for which $\|E\|$ is small.
- A simple trial-and-error procedure for choosing a shift is given below.

Incomplete factorization breakdown

Algorithm (Trial-and-error global shifted incomplete factorization)

Input: *Matrix A, incomplete factorization algorithm, initial shift $\alpha^{(0)}$*
Output: *Shift α such that $A + \alpha I \approx \widetilde{L}\widetilde{U}$.*

1: ***for*** $k = 0, 1, 2, \ldots$ ***do***
2: $A + \alpha^{(k)} I \approx \widetilde{L}\widetilde{U}$ ▷ *Perform incomplete factorization*
3: *If successful,* $\alpha = \alpha^{(k)}$ *and return*
4: $\alpha^{(k+1)} = 2\alpha^{(k)}$
5: ***end for***

Incomplete factorization breakdown

- An alternative approach to avoid small pivots:to follow what is done in sparse direct solvers and incorporate partial or threshold pivoting within the incomplete factorization algorithm: preprocessing by reordering, scaling etc.
- One way to attempt to minimize the norm of the error matrix E is to select the pivot candidate to minimize the sum of the absolute values of the dropped (discarded) entries. However, this minimum discarded fill ordering is typically too expensive to be useful in practice.

Algebraic preconditioning

Factorizations as preconditioner components

- Sometimes (incomplete) factorizations are employed as components in the construction of more complex preconditioners. Here some possible approaches are briefly discussed.

Incomplete factorization breakdown

Polynomial preconditioning

- Polynomial preconditioning selects a polynomial ϕ and applies a Krylov subspace method to solve either

$$\phi(A)Ax = \phi(A)\,b$$

(left preconditioning) or

$$A\,\phi(A)\,y = b, \quad x = \phi(A)\,y$$

(right preconditioning). ϕ should be of small degree and chosen to enhance convergence.
- Consider the characteristic polynomial $\phi_n(\mu) = \det(A - \mu I)$ of A ($\det$ denotes the determinant).
- The Cayley-Hamilton theorem states that A satisfies its own characteristic equation so that

$$\phi_n(A) = \sum_{j=0}^{n} \beta_j\, A^j = 0,$$

Polynomial preconditioning

- Provided A is nonsingular,

$$A^{-1} = (-1)^{n+1} \frac{1}{\det(A)} \sum_{j=1}^{n} \beta_j \, A^{j-1}.$$

A preconditioner can be constructed by taking the first k terms, possibly weighted by some suitable scalar coefficients, that is,

$$M^{-1} = \sum_{j=0}^{k} \gamma_j \, A^k.$$

Algebraic preconditioning

Polynomial preconditioning

- An important question is why such a preconditioner can help in the presence of the optimality properties of Krylov subspace methods.
- For example, at iteration $k+1$ of the CG method, $x^{(k+1)}$ satisfies

$$x^{(k+1)} = x^{(0)} + \phi_k(A)\, r^{(0)},\ k = 0, 1, \ldots,$$

where ϕ_k is a monic polynomial of degree k. This polynomial is optimal in the sense that $x^{(k+1)}$ minimizes

$$\|x - x^{(k+1)}\|_A^2. \tag{89}$$

- A preconditioner that is a polynomial in A cannot speed the convergence because the resulting iteration again forms the new $x^{(k+1)}$ as $x^{(0)}$ plus a polynomial in A times $r^{(0)}$, and thus the same or a higher degree polynomial is needed to achieve the same value of the A-norm of the error.
- Consequently, the number of matrix-vector multiplications cannot decrease.

Polynomial preconditioning

- Nevertheless, polynomial preconditioning can be useful for a number of reasons.
  - The polynomial can improve the eigenvalue distribution of the preconditioned matrix and result in a reduction in the number of iterations required for convergence (even though the overall complexity may increase).
  - It requires very little memory and its implementation can be straightforward.
  - It can decrease the number of synchronization points in iterative methods as represented by inner products. This is potentially important for message-passing parallel architectures.

Algebraic preconditioning

Polynomial preconditioning

- Even if only a small number of terms are used in approximating A^{-1}, a crucial issue is getting $\gamma_0, \ldots, \gamma_k$.
- A straightforward way of doing this: based on the Neumann series of a matrix C given by $\sum_{j=0}^{+\infty} C^j$, which is convergent if and only if $\rho(C) < 1$.
- In this case,

$$(I - C)^{-1} = \sum_{j=0}^{+\infty} C^j. \tag{90}$$

- Now let $\bar{M}$ be a nonsingular matrix and $\omega > 0$ a scalar such that the matrix $C = I - \omega \bar{M}^{-1} A$ satisfies $\rho(C) < 1$.
- Using (90),

$$A^{-1} = \omega(\omega \bar{M}^{-1} A)^{-1} \bar{M}^{-1} = \omega \left(I - C\right)^{-1} \bar{M}^{-1} = \omega \left(\sum_{j=0}^{+\infty} C^j \right) \bar{M}^{-1}.$$

Algebraic preconditioning

Polynomial preconditioning

- Truncating the summation gives as a possible preconditioner

$$M^{-1} = \omega \left(\sum_{j=0}^{k} C^j \right) \bar{M}^{-1}.$$

- Observe that

$$I - M^{-1}A = I - \omega \left(\sum_{j=0}^{k} C^j \right) \bar{M}^{-1}A = I - \left(\sum_{j=0}^{k} C^j \right) (I - C) = C^{k+1},$$

  which shows the positive effect of increasing k. If A and $\bar{M}$ are SPD matrices then M can be used with the CG method preconditioned from the left because $M^{-1}A$ is self-adjoint in the $\bar{M}$-inner product.
- Generalizations of the approach weight the powers of C in M^{-1} using additional scalars. The choice of $\bar{M}$ is crucial for the effectiveness of the approach.

Outline

Incomplete factorizations

World of incomplete factorizations

- Direct factorizations may not be feasible (data structures and pivoting, operation counts, stability)
- Even by direct factorizations improving solution when using less accurate arithmetic (smaller ϵ) may be needed.
- The incomplete factors denoted here by $\widetilde{L}$ and $\widetilde{U}$;
- SPD case: $\widetilde{U} = \widetilde{L}^T$.
- We assume that the sparsity patterns of A and its incomplete factors always include the positions of the diagonal entries.
- Notation (other mentioned later): ILU(0) factorization (or an IC(0) factorization if A is SPD): $\mathcal{S}\{\widetilde{L} + \widetilde{U}\} = \mathcal{S}\{A\}$.

Exactness within the target sparsity pattern

Theorem

Consider the incomplete LU factorization $A + E = \widetilde{L}\widetilde{U}$ with sparsity pattern $\mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. The entries of the error matrix E are zero at positions $(i, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$.

Proof.

The result clearly holds for $j = 1$. Let $(i, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and assume without loss of generality that $i > j > 1$. The (i, j) entry of $\widetilde{L}$ is computed as

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\, \tilde{u}_{kj} \right) / \tilde{u}_{jj}$$

with the sums over k implying $(i, k) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$ and $(k, j) \in \mathcal{S}\{\widetilde{L} + \widetilde{U}\}$. This gives

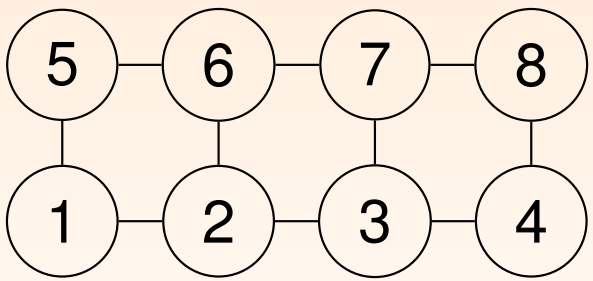
$$a_{ij} = \widetilde{L}_{i,1:j-1}\widetilde{U}_{1:j-1,j} + \tilde{l}_{ij}\tilde{u}_{jj} = \widetilde{L}_{i,1:j}\widetilde{U}_{1:j,j} = L_{i,1:j}U_{1:j,j},$$

and the corresponding entry of E is zero. □

Incomplete factorizations and patterns

- Theorem $\Rightarrow$ extending $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ gives a larger set of entries of A for which $(E)_{ij}=0$.
- In some situations, there are straightforward ways to extend $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$. In simple discretizations of a PDE may be a natural choice is to allow $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ to include fill-in along a few additional diagonals within the band.

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\ \begin{pmatrix} * & * & & & * & & & \\ * & * & * & & & * & & \\ & * & * & * & & & * & \\ & & * & * & & & & * \\ * & & & & * & * & & \\ & * & & & * & * & * & \\ & & * & & & * & * & * \\ & & & * & & & * & * \end{pmatrix} \end{array}$$



Incomplete factorizations

Incomplete factorizations and patterns

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

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

Figure: An 8×8 banded sparse SPD matrix A and its graph $\mathcal{G}(A)$. The first three steps of a Cholesky factorization are shown. Filled entries are denoted by f.

Crout incomplete factorizations

- The Crout variant: computes $\widetilde{U}$ (by rows) and $\widetilde{L}$ (by columns).

Algorithm (Crout incomplete LU factorization)

Input: *Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$.*
Output: *Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.*

1: ***for*** $j = 1 : n$ ***do***
2: $\quad \tilde{l}_{jj} = 1,\ \widetilde{L}_{j+1:n,j} = A_{j+1:n,j}$
3: $\quad \widetilde{U}_{j,j:n} = A_{j,j:n}$
4: $\quad$ ***for*** $k = 1 : j-1$ *such that* $(j,k) \in \mathcal{S}\{\widetilde{L}\}$ ***do***
5: $\quad\quad \widetilde{U}_{j,j:n} = \widetilde{U}_{j,j:n} - \tilde{l}_{jk}\,\widetilde{U}_{k,j:n}$ ▷ *Sparse linear combination*
6: $\quad$ ***end for***
7: $\quad$ *Sparsify* $\widetilde{U}_{j,j+1:n}$ ▷ *Drop entries from row j of $\widetilde{U}$*
8: $\quad$ ***for*** $k = 1 : j-1$ *such that* $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ ***do***
9: $\quad\quad \widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j} - \tilde{u}_{kj}\,\widetilde{L}_{j+1:n,k}$ ▷ *Sparse linear combination*
10: $\quad$ ***end for***
11: $\quad$ *Sparsify* $\widetilde{L}_{j+1:n,j}$ ▷ *Drop entries from column j of $\widetilde{L}$*
12: $\quad \widetilde{L}_{j+1:n,j} = \widetilde{L}_{j+1:n,j}/\tilde{u}_{jj}$
13: ***end for***

Row incomplete factorizations

Algorithm (Row incomplete LU factorization)

Input: *Matrix A and, optionally, a target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$.*
Output: *Incomplete LU factorization $A \approx \widetilde{L}\widetilde{U}$.*

1: ***for*** $i = 1 : n$ ***do***
2: $\tilde{l}_{ii} = 1,\ \widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$
3: $\widetilde{U}_{i,i:n} = A_{i,i:n}$
4: *Sparsify* $\widetilde{L}_{1,1:i-1}$ *and* $\widetilde{U}_{i,i+1:n}$
5: ***for*** $k = 1 : i-1$ *such that* $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ ***do***
6: $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
7: $\widetilde{L}_{i,k+1:i-1} = \widetilde{L}_{i,k+1:i-1} - \tilde{l}_{ik}\,\widetilde{U}_{k,k+1:i-1}$
8: *Sparsify* $\widetilde{L}_{i,k+1:i-1}$
9: $\widetilde{U}_{i,i:n} = \widetilde{U}_{i,i:n} - \tilde{l}_{ik}\,\widetilde{U}_{k,i:n}$
10: *Sparsify* $\widetilde{U}_{i,i+1:n}$
11: ***end for***
12: ***end for***

Incomplete factorizations based on shortest fill-paths

- Entries of the factors that correspond to nonzero entries of A are assigned the level 0 while each potential filled entry in position (i, j) is assigned a level as follows:

$$level(i,j) = \min_{1 \le k < \min\{i,j\}} (level(i,k) + level(k,j) + 1). \tag{91}$$

- Given $\ell \geq 0$, during the factorization a filled entry is permitted at position (i, j) provided $level(i, j) \leq \ell$.
- The resulting level-based incomplete factorization is denoted by ILU(ℓ) (or IC(ℓ)); the basic row variant is given below.

Incomplete factorizations

Algorithm (Level-based incomplete LU factorization)

1: *Initialise* $level$ *to 0 for nonzeros and diagonal entries of* A *and to* $n+1$ *otherwise*
2: **for** $i = 1 : n$ **do** ▷ *Loop over rows*
3: $\tilde{l}_{ii} = 1$, $\widetilde{L}_{i,1:i-1} = A_{i,1:i-1}$ *and* $\widetilde{U}_{i,i:n} = A_{i,i:n}$ ▷ *Initialise row* i *of* $\widetilde{L}$ *and* $\widetilde{U}$
4: **for** $k = 1 : i-1$ *such that* $level(i,k) \leq \ell$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik}/\tilde{u}_{kk}$
6: **for** $j = k+1 : i-1$ **do**
7: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ *and update* $level(i,j)$
8: **end for**
9: **for** $j = i : n$ **do**
10: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ *and update* $level(i,j)$
11: **end for**
12: **end for**
13: **for** $k = 1 : i-1$ **do** ▷ *Drop factor entries in row* i *for which* $level$ *is too high*
14: *if* $level(i,k) > \ell$ *then* $\tilde{l}_{ik} = 0$
15: **end for**
16: **for** $k = i : n$ **do**
17: *if* $level(i,k) > \ell$ *then* $\tilde{u}_{ik} = 0$
18: **end for**
19: **end for**

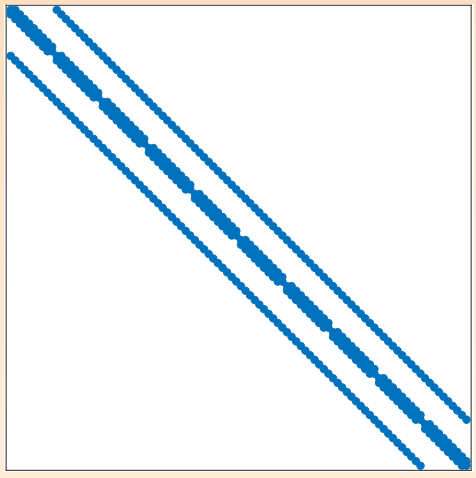
Incomplete factorizations

Incomplete factorizations based on shortest fill-paths

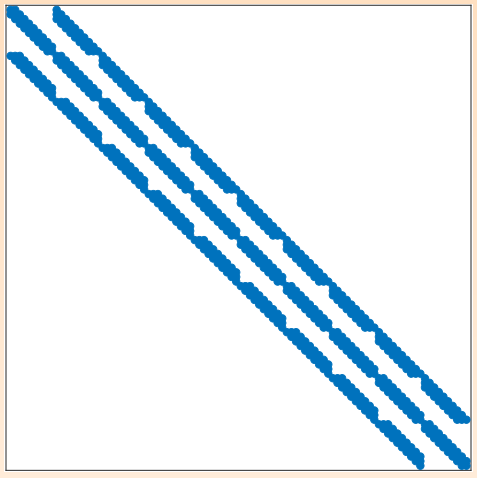
- Figure depicts $\mathcal{S}\{\widetilde{L} + \widetilde{L}^T\}$ for the IC(ℓ) factorization of A from the discretized Laplace equation on a square grid and for a matrix with a more general symmetric sparsity structure.
- The fill-in is typically generated irregularly throughout the factorization: initially few updates are needed but later steps involve many updates, leading to large amounts of dropping.
- The amount of fill-in can grow quickly with increasing ℓ and, as a result, l is typically small.
- Level-based dropping is often combined with threshold-based dropping or with sparsifying A before the factorization commences.

Incomplete factorizations

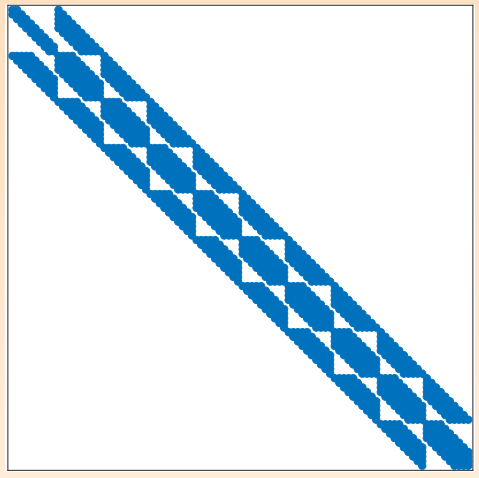
Incomplete factorizations based on shortest fill-paths



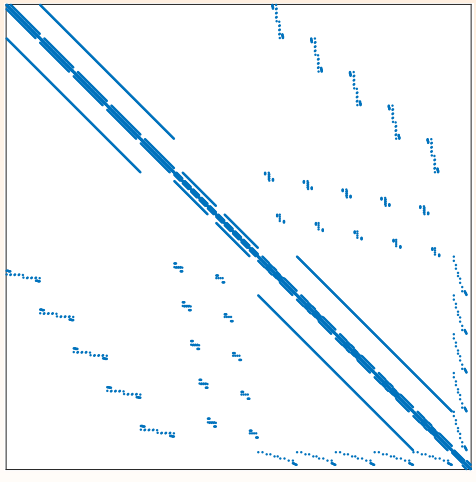
IC(0)



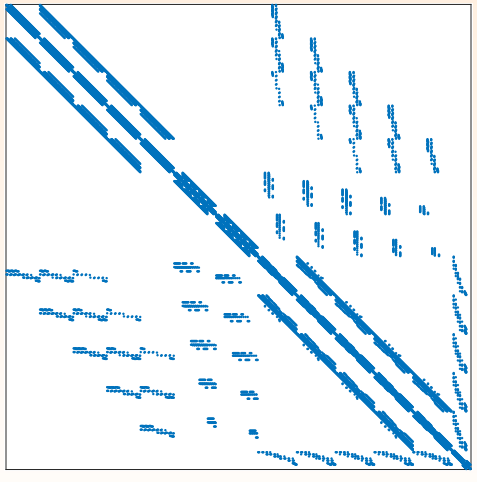
IC(2)



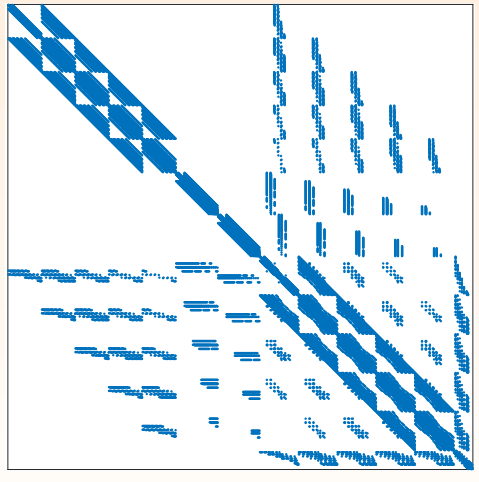
IC(4)



IC(0)



IC(2)



IC(4)

Incomplete factorizations

Incomplete factorizations based on shortest fill-paths

- The level-based strategy comes from observing that in practical examples the absolute values of the entries in the factors in positions for which $level$ is large are often small. This is the case for model problems arising from discretized PDEs.
- Theoretical understanding follows.

Theorem

Consider the ILU(ℓ) factorization of A. $level(i,j) = k$ for some $k \leq \ell$ if and only if there is a shortest fill path $i \Longrightarrow j$ of length $k+1$ in the adjacency graph $\mathcal{G}(A)$.

Incomplete factorizations based on shortest fill-paths

Algorithm (Find the sparsity pattern of row i of the ILU(ℓ) factor $\widetilde{U}$ of A: breadth first search)

1: $\mathcal{S}\{\widetilde{U}_{i,i:n}\} = \{i\}$, $\mathcal{Q} = \{i\}$ ▷ *Queue holds i initially*
2: $length(i) = 0$
3: $visited(i) = i$
4: **while** $\mathcal{Q}$ *is not empty* **do**
5: $pop(\mathcal{Q}, k)$ ▷ *Take k from the queue*
6: **for** $j \in adj_{\mathcal{G}(A)}(k)$ *with* $visited(j) \neq i$ **do**
7: $visited(j) = i$
8: **if** $j < i$ *and* $length(k) < \ell$ **then**
9: $append(\mathcal{Q}, j)$ ▷ *Add j to the queue*
10: $length(j) = length(k) + 1$
11: **else if** $j > i$ **then**
12: $\mathcal{S}\{\widetilde{U}_{i,i:n}\} = \mathcal{S}\{\widetilde{U}_{i,i:n}\} \cup \{j\}$ ▷ *Add j to the sparsity pattern of row i of $\widetilde{U}$*
13: **end if**
14: **end for**
15: **end while**

Incomplete factorizations

Modifications based on maintaining row sums

- Assume that the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ contains $\mathcal{S}\{A\}$.
- Modified incomplete factorizations (MILU or MIC in the SPD case) seek to maintain equality between the row sums of A and $\widetilde{L}\widetilde{U}$, that is, $\widetilde{L}\widetilde{U}e = Ae$ (e is the vector of all ones).

Incomplete factorizations

Algorithm (Modified incomplete factorization (MILU))

1: *Initialise* $\tilde{l}_{ij} = (I + L_A)_{ij}$ *for all* $(i,j) \in \mathcal{S}(\widetilde{L})$ ▷ $\mathcal{S}(L_A) \subseteq \mathcal{S}(\widetilde{L})$
2: *Initialise* $\tilde{u}_{ij} = (D_A + U_A)_{ij}$ *for all* $(i,j) \in \mathcal{S}(\widetilde{U})$ ▷ $\mathcal{S}(U_A) \subseteq \mathcal{S}(\widetilde{U})$
3: **for** $k = 1 : n-1$ **do**
4: **for** $i = k+1 : n$ *such that* $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik} / \tilde{u}_{kk}$ ▷ *Check that* $\tilde{u}_{kk}$ *is nonzero*
6: **for** $j = i : n$ *such that* $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
7: **if** $(i,j) \in \mathcal{S}\{\widetilde{U}\}$ **then**
8: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
9: **else**
10: $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ ▷ *Modify diagonal instead of creating fill-in*
11: **end if**
12: **end for**
13: **for** $j = k+1 : i-1$ *such that* $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
14: **if** $(i,j) \in \mathcal{S}\{\widetilde{L}\}$ **then**
15: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
16: **else**
17: $\tilde{u}_{ii} = \tilde{u}_{ii} - \tilde{l}_{ik}\,\tilde{u}_{kj}$ ▷ *Modify diagonal instead of creating fill-in*
18: **end if**
19: **end for**
20: **end for**

Modifications based on maintaining row sums

- Equality of row sums: If all the filled entries are retained (that is, $\mathcal{S}\{\widetilde{L}+\widetilde{U}\} = \mathcal{S}\{L+U\}$) then the claim holds trivially.
- Otherwise, if an entry in column j of row i of A belongs to the target sparsity pattern then its value is modified in Step 8 if $i \leq j$ or in Step 15 if $i > j$. Otherwise, the i-th diagonal entry of $\widetilde{U}$ is modified (Step 10 or Step 17). In each case, $\tilde{l}_{ik}\,\tilde{u}_{kj}$ is subtracted from entries of the i-th row of the incomplete factors.
- Consider row i of $\widetilde{L}\widetilde{U}$. This product is given by

$$\begin{aligned}
\sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j}^{n} \tilde{u}_{jk} &= \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jj} + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} + \sum_{k=i}^{n} \tilde{u}_{ik} = \\
&= \sum_{j=1}^{i-1} \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\tilde{u}_{kj} \right) + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} + \sum_{k=i}^{n} \left(a_{ik} - \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jk} \right) \\
&= \sum_{j=1}^{n} a_{ij} + \sum_{j=1}^{i-1} \tilde{l}_{ij} \sum_{k=j+1}^{n} \tilde{u}_{jk} - \left(\sum_{j=1}^{i-1} \sum_{k=1}^{j-1} \tilde{l}_{ik}\tilde{u}_{kj} + \sum_{k=i}^{n} \sum_{j=1}^{i-1} \tilde{l}_{ij}\tilde{u}_{jk} \right).
\end{aligned}$$

Modifications based on maintaining row sums

- Rearranging the indices in the double summations, the last three sums cancel out.
- Moreover, the added double summation is the sum of all the modification terms $\tilde{l}_{ik}\,\tilde{u}_{kj}$ in the MILU Algorithm, and the sum of the two subtracted double summations also comprises all the modification terms.
- Consequently, the row sums of A are preserved in the product of the incomplete factors.

Incomplete factorizations

Modifications based on maintaining row sums

- Theorem above provides motivation for maintaining constant row sums in the case of a model PDE problem.
- The result is also valid for Neumann or mixed boundary conditions, and there are extensions to three-dimensional problems and MIC(ℓ) with $\ell > 0$. However, although Theorem holds for MILU factorizations, the approach may not be useful for general A.

Theorem

Let A come from a discretized Poisson problem on a uniform two-dimensional rectangular grid with Dirichlet boundary conditions and discretization parameter h. Then the condition number $\kappa((\widetilde{L}\widetilde{U})^{-1}A)$ for the level-based MIC(0) preconditioner is $O(h^{-1})$.

Incomplete factorizations

Modifications based on maintaining row sums

- RILU/RIC: the update term $\tilde{l}_{ik}\,\tilde{u}_{kj}$ may be multiplied by a parameter θ ($0 < \theta < 1$) before it is subtracted from the diagonal entry $\tilde{u}_{ii}$.
- This is a practical way to extend MILU to linear systems not coming from discretized PDEs. Clearly, using $\theta < 1$ reduces the amount by which the diagonal entries are modified.

Dynamic compensation

- Instead of accepting a filled entry in position (i, j), the idea is to add a (weighted) multiple of its absolute value to the corresponding diagonal entries $\tilde{u}_{ii}$ and $\tilde{u}_{jj}$.
- Provided the number of modifications is small, this can be useful if A is diagonally dominant and scaled so that its diagonal entries are nonnegative.
- The parameter ω controls the amount by which the diagonal entries of $\widetilde{U}$ are modified but if $\omega < 1$ then breakdown can still occur.
- Dynamic compensation can be successful when incorporated into an IC factorization of a SPD matrix A because the resulting local modifications correspond to adding positive semidefinite matrices to A.
- In practice, the behaviour of the resulting preconditioner can be very different from that computed using the MIC approach.

Incomplete factorizations

Algorithm (ILU factorization with dynamic compensation)

1: $\tilde{l}_{ij} = (I + L_A)_{ij}$ *for all* $(i,j) \in \mathcal{S}(\widetilde{L})$
2: $\tilde{u}_{ij} = (D_A + U_A)_{ij}$ *for all* $(i,j) \in \mathcal{S}(\widetilde{U})$
3: **for** $k = 1 : n-1$ **do**
4: **for** $i = k+1 : n$ *such that* $(i,k) \in \mathcal{S}\{\widetilde{L}\}$ **do**
5: $\tilde{l}_{ik} = \tilde{l}_{ik} / \tilde{u}_{kk}$
6: **for** $j = i : n$ *such that* $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
7: **if** $(i,j) \in \mathcal{S}\{\widetilde{U}\}$ **then**
8: $\tilde{u}_{ij} = \tilde{u}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
9: **else**
10: $\rho_{ij} = (\tilde{u}_{ii}/\tilde{u}_{jj})^{1/2}$
11: $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega\rho_{ij}\,|\tilde{l}_{ik}\,\tilde{u}_{kj}|, \quad \tilde{u}_{jj} = \tilde{u}_{jj} + \omega|\tilde{l}_{ik}\,\tilde{u}_{kj}|\,/\rho_{ij}, \quad \tilde{u}_{ij} = 0.$
12: **end if**
13: **end for**
14: **for** $j = k+1 : i-1$ *such that* $(k,j) \in \mathcal{S}\{\widetilde{U}\}$ **do**
15: **if** $(i,j) \in \mathcal{S}\{\widetilde{L}\}$ **then**
16: $\tilde{l}_{ij} = \tilde{l}_{ij} - \tilde{l}_{ik}\,\tilde{u}_{kj}$
17: **else**
18: $\rho_{ij} = (\tilde{u}_{ii}/\tilde{u}_{jj})^{1/2}$
19: $\tilde{u}_{ii} = \tilde{u}_{ii} + \omega\rho_{ij}\,|\tilde{l}_{ik}\,\tilde{u}_{kj}|, \quad \tilde{u}_{jj} = \tilde{u}_{jj} + \omega|\tilde{l}_{ik}\,\tilde{u}_{kj}|\,/\rho_{ij}, \quad \tilde{l}_{ij} = 0.$
20: **end if**

Dynamic compensation: getting closer to special matrices

- A related scheme, called diagonally compensated reduction, modifies A before the factorization begins by adding the values of all of its positive off-diagonal entries to the corresponding diagonal entries and then setting these off-diagonal entries to zero.
- If A is SPD then the resulting matrix is a symmetric M-matrix and the incomplete factorization of an M-matrix is breakdown-free. However, the modified matrix may be too far from A for its incomplete factors to be useful.

Memory-limited incomplete factorizations

- A be SPD, consider the decomposition

$$A = (\widetilde{L} + \widetilde{R})\,(\widetilde{L} + \widetilde{R})^T - E.$$

- The error matrix E is $E = \widetilde{R}\widetilde{R}^T$.
- On step j of the incomplete factorization, the first column of the Schur complement $S^{(j)}$ is split into the sum

$$\widetilde{L}_{j:n,j} + \widetilde{R}_{j:n,j},$$

where $\widetilde{L}_{j:n,j}$ contains the entries that are retained in column j of the final incomplete factorization and $\widetilde{R}_{jj} = 0$ and $\widetilde{R}_{j+1:n,j}$ contains the entries that are discarded.

Memory-limited incomplete factorizations

- If a complete factorization was being computed then the Schur complement would be updated by subtracting

$$(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})\,(\widetilde{L}_{j+1:n,j} + \widetilde{R}_{j+1:n,j})^T.$$

- However, the incomplete factorization discards the term

$$E^{(j)} = \widetilde{R}_{j+1:n,j}\,\widetilde{R}_{j+1:n,j}^T.$$

  Thus, $E^{(j)}$ is implicitly added to A and because $E^{(j)}$ is positive semidefinite, the approach is naturally breakdown-free.
- The obvious choice for $\widetilde{R}_{j+1:n,j}$ is the smallest off-diagonal entries in the column.

Memory-limited incomplete factorizations

- Figure depicts the first step of this approach. In the first row and column, $*$ and δ denote the entries of $\widetilde{L}_{1:n,1}$ and $\widetilde{R}_{1:n,1}$, respectively.
- Standard sparsification scheme: no fill (left)
- Using intermediate memory: right.

$$\begin{pmatrix} * & * & * & \delta & \delta \\ * & f & f & & \\ * & f & f & & \\ \delta & & & & \\ \delta & & & & \end{pmatrix} \quad \begin{pmatrix} * & * & * & \delta & \delta \\ * & f & f & f & f \\ * & f & f & f & f \\ \delta & f & f & & \\ \delta & f & f & & \end{pmatrix}$$

Figure: An illustration of the fill-in in a standard sparsification-based IC factorization (left) and in the approach that uses intermediate memory (right) after one step of the factorization. Entries with small absolute value in row and column 1 are denoted by δ. The filled entries are denoted by f.

Memory-limited incomplete factorizations

- Enables the structure of the complete factorization to be followed more closely than is possible using a standard approach. If the small entries at positions $(1,3)$ and $(3,1)$ are not discarded then there is a filled entry in position $(3,2)$ and this allows the incomplete factorization using intermediate memory to involve the (large) off-diagonal entries in positions $(5,2)$ and $(6,2)$ in the second step of the IC factorization.

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

Incomplete factorizations

Memory-limited incomplete factorizations

- Unfortunately, because the column $\widetilde{R}_{j+1:n,j}$ must be retained to perform the updates on the next step, the total memory requirements are essentially as for a complete factorization.
- Relaxations are needed: e.g., introducing two drop tolerances so that only entries of absolute value at least τ_1 are kept in $\widetilde{L}$ and entries smaller than τ_2 are dropped from $\widetilde{R}$.
- Or, limiting the fill-in.
- But, then, no longer breakdown-free approach.

Incomplete factorizations

Algorithm (Crout memory-limited IC factorization)

1: $w(1:n) = 0$
2: **for** $j = 1 : n$ **do**
3: **for** $i = j : n$ *such that* $a_{ij} \neq 0$ **do**
4: $w_i = a_{ij}$ ▷ w *is a vector of length* n
5: **end for**
6: **for** $k < j$ *such that* $\tilde{l}_{jk} \neq 0$ **do**
7: **for** $i = j : n$ *such that* $\tilde{l}_{ik} \neq 0$ **do**
8: $w_i = w_i - \tilde{l}_{ik}\,\tilde{l}_{jk}$
9: **end for**
10: **for** $i = j : n$ *such that* $\tilde{r}_{ik} \neq 0$ **do**
11: $w_i = w_i - \tilde{r}_{ik}\,\tilde{l}_{jk}$
12: **end for**
13: **end for**
14: **for** $k < j$ *such that* $\tilde{r}_{jk} \neq 0$ **do**
15: **for** $i = j : n$ *such that* $\tilde{l}_{ik} \neq 0$ **do**
16: $w_i = w_i - \tilde{l}_{ik}\,\tilde{r}_{jk}$
17: **end for**
18: **end for**
19: *Copy into* $\widetilde{L}_{j:n,j}$ *the* $lsize + nz(A_{j:n,j})$ *entries of* w *of largest absolute value*
20: *Copy into* $\widetilde{R}$ *the* ... *entries of* ... *that are the next largest in absolute value*

Fixed-point iterations for computing ILU factorizations

- Given the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, the goal is to iteratively generate incomplete factors fulfilling the ILU property

$$(\widetilde{L}\widetilde{U})_{ij} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$$

- Parallel computation using the constraints

$$\sum_{\substack{k=1 \\ (i,k),(k,j)\in\mathcal{S}\{\widetilde{L}+\widetilde{U}\}}}^{\min(i,j)} \tilde{l}_{ik}\,\tilde{u}_{kj} = a_{ij}, \quad (i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\},$$

and the normalization $\tilde{l}_{ii} = 1$.

Fixed-point iterations for computing ILU factorizations

- Using the relations

$$\tilde{l}_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik}\, \tilde{u}_{kj} \right) / \tilde{u}_{jj}, \quad i > j, \tag{92}$$

$$\tilde{u}_{ij} = a_{ij} - \sum_{k=1}^{i-1} \tilde{l}_{ik}\, \tilde{u}_{kj}, \quad i \leq j, \tag{93}$$

the approach can be formulated as a fixed-point iteration method of the form $w^{k+1} = f(w^k)$, $k = 0, 1, \ldots$, where w is a vector containing the unknowns $\tilde{l}_{ij}$ and $\tilde{u}_{ij}$. Each fixed-point iteration is called a sweep.

Fixed-point iterations for computing ILU factorizations

Algorithm (Fixed-point ILU factorization)

Input: *Matrix A, the target sparsity pattern $\mathcal{S}\{\widetilde{L}+\widetilde{U}\}$, and initial incomplete factors $\widetilde{L}$ and $\widetilde{U}$.*
Output: *Updated incomplete factors.*

Set $\tilde{l}_{ij}$ and $\tilde{u}_{ij}$ to initial values
for $sweep = 1, 2, \ldots$ ***do***
    for $(i,j) \in \mathcal{S}\{\widetilde{L}+\widetilde{U}\}$ ***do***
        if $i > j$ ***then***
            Compute $\tilde{l}_{ij}$ using (92)
        else
            Compute $\tilde{u}_{ij}$ using (93)
        end if
    end for
end for

Ordering in incomplete factorizations

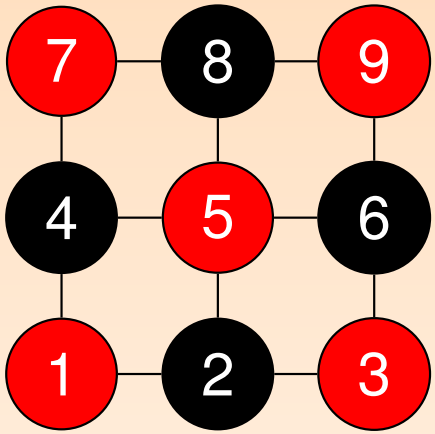
- Can have a positive effect on the robustness and performance of preconditioned Krylov subspace methods.
- The best choice of ordering for an incomplete factorization preconditioner may not be the same as for a complete factorization.
- When the natural (lexicographic) ordering is used, the incomplete triangular factors resulting from a no-fill ILU factorization can be highly ill-conditioned, even if the matrix A is well conditioned.
- Allowing more fill-in in the factors, for example, using ILU(1) instead of ILU(0), may solve the problem but it is not guaranteed.
- Minimum degree orderings: the rows (and columns) of the permuted matrix can have significantly different counts.
- A strategy is to specify that the permitted fill-in is proportional to the row/column counts of the complete factorization.

Ordering in incomplete factorizations

- Global orderings cut local connections within the graph of A.
- When used with incomplete factorizations, can lead to poor quality preconditioners.
- A global ordering that specifically targets incomplete factorizations is a red-black (or checker board) ordering.

Incomplete factorizations

Ordering in incomplete factorizations



$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\
\left(\begin{array}{ccccccccc}
4 & -1 & & -1 & & & & & \\
-1 & 4 & -1 & & -1 & & & & \\
 & -1 & 4 & & & -1 & & & \\
-1 & & & 4 & & & -1 & & \\
 & -1 & & -1 & 4 & -1 & & -1 & \\
 & & -1 & & -1 & 4 & & & -1 \\
 & & & -1 & & & 4 & -1 & \\
 & & & & -1 & & -1 & 4 & -1 \\
 & & & & & -1 & & -1 & 4
\end{array}\right)
\end{array}
\qquad
\begin{array}{c} \\ 1 \\ 3 \\ 5 \\ 7 \\ 9 \\ 2 \\ 4 \\ 6 \\ 8 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} 1 & 3 & 5 & 7 & 9 & 2 & 4 & 6 & 8 \end{array} \\
\left(\begin{array}{ccccccccc}
4 & & & & & -1 & -1 & & \\
 & 4 & & & & -1 & -1 & & \\
 & & 4 & & & -1 & -1 & -1 & -1 \\
 & & & 4 & & & -1 & & -1 \\
 & & & & 4 & & & -1 & -1 \\
-1 & -1 & -1 & & & 4 & & & \\
-1 & & -1 & -1 & & & 4 & & \\
 & -1 & -1 & & -1 & & & 4 & \\
 & & -1 & -1 & -1 & & & & 4
\end{array}\right)
\end{array}
$$

Figure: *A model problem to illustrate a red-black ordering.*

Exploiting block structure

- Blocking methods for complete factorizations can be adapted to incomplete factorizations. The aim is to speed up the computation of the factors and to obtain more effective preconditioners.
- In a block factorization, scalar operations of the form

$$\tilde{l}_{ik} = a_{ik}/\tilde{u}_{kk}$$

are replaced by matrix operations

$$\widetilde{L}_{ib,kb} = A_{ib,kb}\widetilde{U}^{-1}_{kb,kb},$$

and scalar multiplications of entries of the factors are replaced by matrix-matrix products. When dropping entries, instead of considering the absolute values, simple norms of the block entries (such as the one-norm, max-norm or Frobenius norm) are used.

- An incomplete factorization can start with the supernodal structure of the complete factors.

Outline

Sparse approximate inverses

Approximate inverse preconditioners

- Standard solves by substitution steps can present a computational bottleneck. In particular, in parallel computational environment.
- But it is M^{-1}, which represents an approximation of A^{-1}, that is applied by performing forward and back substitution steps
- Therefore, an alternative strategy to standard (incomplete) factorizations is to directly approximate A^{-1} by explicitly computing M^{-1}.

Sparse approximate inverses

Approximate inverse preconditioners

- But, there is a problem: The sparsity pattern of the inverse of an irreducible matrix A is dense, even when A is sparse.
- But, perhaps there is a way ...: although A^{-1} is fully dense, the following result shows this is not the case for the factors of factorized inverses.

Theorem

Assume the matrix A is SPD and let L be its Cholesky factor. Then $\mathcal{S}\{L^{-1}\}$ is the union of all entries (i,j) such that i is an ancestor of j in the elimination tree $\mathcal{T}(A)$.

- A consequence of this result is that L^{-1} need not be fully dense.
- Algorithmically, if A is SPD it may be advantageous to preorder A to limit the number of ancestors of vertices in $\mathcal{T}(A)$.
- For example, by ND applied to $\mathcal{S}\{A\}$ or to $\mathcal{S}\{A + A^T\}$.

Sparse approximate inverses

Basic approaches

- An obvious way: to compute an incomplete LU factorization of A and then perform an approximate inversion of the incomplete factors.
- But, two levels of approximation.

Sparse approximate inverses

Basic approaches

- Another straightforward approach is based on bordering.
- Let $A_j = A_{1:j,1:j}$ and its inverse factorization

$$A_j^{-1} = W_j D_j^{-1} Z_j^T$$

is known (W_j and Z_j are unit upper triangular matrices and D_j is a diagonal matrix).

$$\begin{pmatrix} Z_j^T & 0 \\ z_{j+1}^T & 1 \end{pmatrix} \begin{pmatrix} A_j & A_{1:j,j+1} \\ A_{j+1,1:j} & a_{j+1,j+1} \end{pmatrix} \begin{pmatrix} W_j & w_{j+1} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} D_j & 0 \\ 0 & d_{j+1,j+1} \end{pmatrix},$$

where for $1 \leq j < n$

$$w_{j+1} = -W_j D_j^{-1} Z_j^T A_{1:j,j+1}, z_{j+1} = -Z_j D_j^{-1} W_j^T A_{j+1,1:j}^T,$$

$$d_{j+1,j+1} = a_{j+1,j+1} + z_{j+1}^T A_j w_{j+1} + A_{j+1,1:j} w_{j+1} + z_{j+1}^T A_{1:j,j+1}.$$

- Starting from $j = 1$, this suggests a procedure for computing the inverse factors of A. Sparsity can be preserved by dropping.

Approximate inverse by bordering

Algorithm (Nonsymmetric inverse bordering algorithm)

Input: *Generally nonsymmetric* A.

Output: *A unit upper triangular matrix* $\widetilde{Z}$ *and diagonal matrix* $\widetilde{D}$ *such that* $A^{-1} \approx \widetilde{Z}^T \widetilde{D}^{-1} \widetilde{W}^T$.

Set $\widetilde{Z}_1 = (1)$, $\widetilde{W}_1 = (1)$, $\widetilde{D}_1 = (a_{11})$.
for $j = 2 : n$ ***do***
  Set $\tilde{z}_j = -\widetilde{Z}_{1:j-1,1:j-1} \widetilde{D}^{-1}_{1:j-1,1:j-1} \widetilde{W}^T_{1:j-1,1:j-1} A_{1:j-1,j}$
  Set $\tilde{w}_j = -\widetilde{W}_{1:j-1,1:j-1} \widetilde{D}^{-1}_{1:j-1,1:j-1} \widetilde{Z}^T_{1:j-1,1:j-1} A^T_{j,1:j-1}$
  Set $\tilde{d}_j = A_{jj} + A^T_{1:j-1,j} \tilde{w}_j + A_{j,1:j-1} \tilde{z}_j + \tilde{w}_j^T A_{j-1} \tilde{z}_j$
  Set $\widetilde{Z}_j = \begin{pmatrix} \widetilde{Z}_{1:j-1,1:j-1} & \tilde{z}_j \\ 0 & 1 \end{pmatrix}$
  Set $\widetilde{W}_j = \begin{pmatrix} \widetilde{W}_{1:j-1,1:j-1} & \tilde{w}_j \\ 0 & 1 \end{pmatrix}$
end for
Set $\widetilde{Z} = \widetilde{Z}_n, \widetilde{W} = \widetilde{W}_n, \widetilde{D} = \widetilde{D}_n$

Inverse by bordering: notes

- If A is symmetric, $W = Z$ and the required work is halved.
- Furthermore, if A is SPD then it can be shown that, in exact arithmetic, $d_{jj} > 0$ for all j and the process does not break down.
- The computation of Z and W are tightly coupled restricting the potential to exploit parallelism.
- This implies a potential problem with efficient implementation.

Sparse approximate inverses

Frobenius norm minimization: SPAI

- Denote $K = M^{-1}$.
- Use minimization of

$$\|I - AM^{-1}\|_F^2 = \|I - AK\|_F^2 = \sum_{j=1}^{n} \|e_j - Ak_j\|_2^2, \tag{94}$$

  over all K with pattern $\mathcal{S}$.
- A left approximate inverse can be computed by solving a minimization problem for $\|I - KA\|_F = \|I - A^T K^T\|_F$.
- The problem reduces to least squares problems for the columns of K that can be computed independently and, if required, in parallel.

Sparse approximate inverses

Frobenius norm minimization: SPAI

- These least squares (LS) problems are all of small dimension when $\mathcal{S}$ is chosen to ensure K is sparse.
- Let $\mathcal{J} = \{i \,|\, k_j(i) \neq 0\}$ be the set of indices of the nonzero entries in column k_j. Further, denote $\mathcal{I} = \{m \,|\, A_{m,\mathcal{J}} \neq 0\}$.
- Let $\widehat{e}_j = e_j(\mathcal{I})$ be the vector of length $|\mathcal{I}|$ that is obtained by taking the entries of e_j with row indices belonging to $\mathcal{I}$.
- To solve the LS problem for k_j, construct the $|\mathcal{I}| \times |\mathcal{J}|$ matrix $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and solve

$$\min_{\widehat{k}_j} \|\widehat{e}_j - \widehat{A}\,\widehat{k}_j\|_2^2. \tag{95}$$

- This can be done using QR factorization of $\widehat{A}$. Extending $\widehat{k}_j$ to have length n by zeros gives k_j.

SPAI algorithm

- Construction: starting with a chosen column sparsity pattern $\mathcal{J}$ for k_j, construct $\widehat{A}$, solve (95) for $\widehat{k}_j$, set $k_j(\mathcal{J}) = \widehat{k}_j$ and define the residual vector

$$r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j.$$

- If $\|r_j\|_2 \neq 0$ then k_j is not equal to the j-th column of A^{-1} and a better approximation can be derived by augmenting $\mathcal{J}$.
- Augmentation: let $\mathcal{L} = \{l \,|\, r_j(l) \neq 0\}$ and define

$$\widetilde{\mathcal{J}} = \{i \,|\, A_{\mathcal{L},i} \neq 0\} \setminus \mathcal{J}. \tag{96}$$

- One or more candidate indices that can be added to $\mathcal{J}$ can be chosen. For example, such that most effectively reduce $\|r_j\|_2$.

Sparse approximate inverses

SPAI algorithm

- A possible heuristic is to solve for each $i \in \widetilde{\mathcal{J}}$ the minimization problem
$$\min_{\mu_i} ||r_j - \mu_i A e_i||_2^2.$$
- This has the solution $\mu_i = r_j^T A e_i / \|Ae_i\|_2^2$ with residual $\|r_j\|^2 - (r_j^T A e_i)^2 / \|Ae_i\|_2^2$.
- Indices $i \in \widetilde{\mathcal{J}}$ for which this is small are appended to $\mathcal{J}$.
- The process can be repeated until either the required accuracy is attained or the maximum number of allowed entries in $\mathcal{J}$ is reached.

Sparse approximate inverses

SPAI algorithm

- Solving the unconstrained LS problem after extending $\widehat{A}$ to $A_{\mathcal{I}\cup\mathcal{I}',\mathcal{J}\cup\mathcal{J}'}$ is typically performed by updating the previous problems.
- Assume the QR factorization of $\widehat{A}$ is

$$\widehat{A} = A_{\mathcal{I},\mathcal{J}} = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix},$$

  where Q_1 is $|\mathcal{I}| \times |\mathcal{J}|$.
- The QR factorization of the extended matrix is

$$\begin{aligned} A_{\mathcal{I}\cup\mathcal{I}',\mathcal{J}\cup\mathcal{J}'} &= \begin{pmatrix} \widehat{A} & A_{\mathcal{I},\mathcal{J}'} \\ & A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix} = \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} R & Q_1^T A_{\mathcal{I},\mathcal{J}'} \\ & Q_2^T A_{\mathcal{I},\mathcal{J}'} \\ & A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix} \\ &= \begin{pmatrix} Q & \\ & I \end{pmatrix} \begin{pmatrix} I & \\ & Q' \end{pmatrix} \begin{pmatrix} R & Q_1^T A_{\mathcal{I},\mathcal{J}'} \\ & R' \\ & 0 \end{pmatrix}. \end{aligned}$$

SPAI algorithm

- Q' and R' are from the QR factorization of the $(|\mathcal{I}'| + |\mathcal{I}| - |\mathcal{J}|) \times |\mathcal{J}'|$ matrix

$$\begin{pmatrix} Q_2^T A_{\mathcal{I},\mathcal{J}'} \\ A_{\mathcal{I}',\mathcal{J}'} \end{pmatrix}.$$

- Factorizing this matrix and updating the trailing QR factorization to get the new $\widehat{k}_j$ is much more efficient than computing the QR factorization of the extended matrix from scratch.
- Many variations of the basic approach.

SPAI algorithm

Algorithm (**SPAI preconditioner (right-looking approach)**)

Input: *Nonsymmetric matrix A, a convergence tolerance $\eta > 0$, an initial sparsity pattern $\mathcal{J}_j$ and the maximum number nz_j of permitted entries for column j of K $(1 \le j \le n)$.*
Output: *$K \approx A^{-1}$ with columns k_j $(1 \le j \le n)$.*

1: ***for*** $j = 1 : n$ ***do*** ▷ *The columns may be computed in parallel*
2: Set $\mathcal{J} = \mathcal{J}_j$ and $\mathcal{I} = \{m \,|\, A(m, \mathcal{J}) \neq 0\}$, $\|r_j\|_2 = \infty$
3: Construct $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and solve (95) for $\widehat{k}_j$
4: $r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j$
5: ***while*** $|\mathcal{J}| < nz_j$ and $\|r_j\|_2 > \eta$ ***do***
6: Construct $\widetilde{\mathcal{J}}$ given by (96) ▷ *$\widetilde{\mathcal{J}}$ is the candidate set*
7: Determine new indices $\mathcal{J}' \subset \widetilde{\mathcal{J}}$ to add to $\mathcal{J}$
8: $\mathcal{I}' = \{m \,|\, A_{m,\mathcal{J}'} \neq 0\} \setminus \mathcal{I}$
9: $\mathcal{I} = \mathcal{I} \cup \mathcal{I}'$ and $\mathcal{J} = \mathcal{J} \cup \mathcal{J}'$ ▷ *Augment the sparsity pattern*
10: Construct new $\widehat{A} = A_{\mathcal{I},\mathcal{J}}$ and new $\widehat{k}_j$ ▷ *Update the QR factorization*
11: $r_j = e_j - A_{1:n,\mathcal{J}}\widehat{k}_j$
12: ***end while***
13: $k_j(\mathcal{J}) = \widehat{k}_j$ ▷ *Extend $\widehat{k}_j$ to k_j by setting entries not in $\mathcal{J}$ to zero.*
14: ***end for***

SPAI algorithm

The example: The algorithm starts with $\mathcal{J}_1 = \{1, 2\}$.

$$A = \begin{pmatrix} 10 & -2 & & & \\ -1 & 10 & -2 & & \\ & -1 & 10 & -2 & \\ & & -1 & 10 & -2 \\ & & & -1 & 10 \end{pmatrix}, \widehat{A} = \begin{pmatrix} 10 & -2 \\ -1 & 10 \\ & -1 \end{pmatrix}, \widehat{k}_1 = \begin{pmatrix} 0.1020 \\ 0.0101 \end{pmatrix}, r_1 = \begin{pmatrix} 1.00 \times 10^{-4} \\ 1.00 \times 10^{-3} \\ 1.01 \times 10^{-2} \\ 0 \\ 0 \end{pmatrix}.$$

$$\widehat{A} = \begin{pmatrix} 10 & -2 & \\ -1 & 10 & -2 \\ & -1 & 10 \\ & & -1 \end{pmatrix}, \widehat{k}_1 = \begin{pmatrix} 0.1021 \\ 0.0104 \\ 0.0010 \end{pmatrix}, r_1 = \begin{pmatrix} 1.0 \times 10^{-5} \\ 1.1 \times 10^{-4} \\ 1.1 \times 10^{-3} \\ 1.0 \times 10^{-2} \\ 0 \end{pmatrix}, k_1 = \begin{pmatrix} 0.1021 \\ 0.0104 \\ 0.0010 \\ 0 \\ 0 \end{pmatrix}.$$

Figure: An illustration of computing the first column of a sparse approximate inverse using the SPAI algorithm with $nz_1 = 3$. On the top line is the initial tridiagonal matrix A followed by the matrix $\widehat{A}$ and the vectors $\widehat{k}_1$ and r_1 on the first loop of Algorithm. The bottom line presents the updated matrix $\hat{A}$ that is obtained on the second loop by adding the third row and column of A and the corresponding vectors $\widehat{k}_1$ and r_1 and, finally, k_1.

Sparse approximate inverses

SPAI algorithm

- When A is symmetric, there is no guarantee that the computed K will be symmetric. One possibility is to use $(K + K^T)/2$ to approximate A^{-1}.
- The SPAI preconditioner is not sensitive to reorderings of A. This has the advantage that A can be partitioned and reordered in whatever way is convenient, for instance to better suit the needs of a distributed implementation.
- The disadvantage is that orderings cannot be used to reduce fill-in and/or improve the quality of the approximate inverse.
- For instance, if A^{-1} has no small entries, SPAI will not find a sparse K, and because the inverse of a permutation of A is just a permutation of A^{-1}, no permutation of A will change this.

FSAI preconditioner: SPD case

- The factorized sparse approximate inverse (FSAI) preconditioner for an SPD matrix A is defined as the product

$$M^{-1} = G^T G,$$

where the sparse lower triangular matrix G is an approximation of the inverse of the (complete) Cholesky factor L of A.

- Theoretically, a FSAI preconditioner is computed by choosing a lower triangular sparsity pattern $\mathcal{S}_L$ and minimizing

$$\|I - GL\|_F^2 = tr\left[(I - GL)^T (I - GL)\right] \tag{97}$$

over all G with sparsity pattern $\mathcal{S}_L$.

FSAI preconditioner: SPD case

- Differentiating the formula with respect to the entries of G and setting to zero yields

$$(GLL^T)_{ij} = (GA)_{ij} = (L^T)_{ij} \quad \text{for all} \quad (i,j) \in \mathcal{S}_L. \tag{98}$$

- Because L^T is an upper triangular matrix while $\mathcal{S}_L$ is a lower triangular pattern, the matrix equation (98) can be rewritten as

$$(GA)_{ij} = \begin{cases} 0 & i \neq j, \quad (i,j) \in \mathcal{S}_L \\ l_{ii} & i = j. \end{cases} \tag{99}$$

FSAI preconditioner: SPD case

- G is not available directly because L is unknown. Instead, $\overline{G}$ is computed such that

$$(\overline{G}A)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i,j) \in \mathcal{S}_L, \tag{100}$$

  where $\delta_{i,j}$ is the Kronecker delta function ($\delta_{i,j} = 1$ if $i = j$ and is equal to 0, otherwise).
- The FSAI factor G is then obtained by setting

$$G = D\overline{G},$$

  where D is a diagonal scaling matrix.
- An appropriate choice for D is

$$D = [diag(\overline{G})]^{-1/2}, \tag{101}$$

  so that

$$(GAG^T)_{ii} = 1, \quad 1 \leq i \leq n.$$

FSAI preconditioner: SPD case

Theorem

Assume A is SPD. If the lower triangular sparsity pattern $\mathcal{S}_L$ includes all diagonal positions then G exists and is unique.

Proof.

Set $\mathcal{I}_i = \{j \mid (i,j) \in \mathcal{S}_L\}$ and let $A_{\mathcal{I}_i,\mathcal{I}_i}$ denote the submatrix of order $nz_i = |\mathcal{I}_i|$ of entries a_{kl} such that $k,l \in \mathcal{I}_i$. Let $\bar{g}_i$ and g_i be dense vectors containing the nonzero coefficients in row i of $\overline{G}$ and G, respectively. Using this notation, solving (100) decouples into solving n independent SPD linear systems

$$A_{\mathcal{I}_i,\mathcal{I}_i}\,\bar{g}_i = e_{nz_i}, \quad 1 \le i \le n,$$

where the unit vectors are of length nz_i. Moreover,

$$(\overline{G}A\overline{G}^T)_{ii} = \sum_{j\in\mathcal{I}_i} \delta_{i,j}\overline{G}_{ij} = \overline{G}_{ii} = (A^{-1}_{\mathcal{I}_i,\mathcal{I}_i})_{ii}.$$

This implies that the diagonal entries of D given by (101) are nonzero. Consequently, the computed rows of G exist and provide a unique solution. □ □

FSAI preconditioner: SPD case

Algorithm (**FSAI preconditioner**)

Input: *SPD matrix A and lower triangular sparsity pattern $\mathcal{S}_L$ that includes all diagonal positions.*
Output: *Lower triangular matrix G such that $A^{-1} \approx GG^T$.*

1: **for** $i = 1 : n$ **do**
2: Construct $\mathcal{I}_i = \{j \mid (i,j) \in \mathcal{S}_L\}$, $A_{\mathcal{I}_i,\mathcal{I}_i}$ and set $nz_i = |\mathcal{I}_i|$
3: Solve $A_{\mathcal{I}_i,\mathcal{I}_i}\, \bar{g}_i = e_{nz_i}$
4: Scale $g_i = d_{ii}\bar{g}_i$ with $d_{ii} = (\bar{g}_{i,nz_i})^{-1/2}$ ▷ $\bar{g}_{i,nz_i}$ is the last component of $\bar{g}_i$
5: Extend g_i to the row $G_{i,1:i}$ by setting entries that are not in $\mathcal{I}_i$ to zero
6: **end for**

FSAI preconditioner: SPD case

- Monotonicity property.

Theorem

Let L be the Cholesky factor of the SPD matrix A. Given the lower triangular sparsity pattern $\mathcal{S}_L$ that includes all diagonal positions, let G be the FSAI preconditioner computed using Algorithm above. Then any lower triangular matrix G_1 with its sparsity pattern is contained in $\mathcal{S}_L$ a $(G_1 A G_1^T)_{ii} = 1$ $(1 \leq i \leq n)$ satisfies

$$||I - GL||_F \leq ||I - G_1 L||_F.$$

FSAI preconditioner: SPD case

- The performance is highly dependent on the choice of $\mathcal{S}_L$.

Theorem

Let L be the Cholesky factor of the SPD matrix A. Given the lower triangular sparsity patterns $\mathcal{S}_{L1}$ and $\mathcal{S}_{L2}$ that include all diagonal positions, let the corresponding FSAI preconditioners computed using Algorithm 14.3 be G_1 and G_2, respectively. If $\mathcal{S}_{L1} \subseteq \mathcal{S}_{L2}$ then

$$||I - G_2 L||_F \leq ||I - G_1 L||_F.$$

FSAI preconditioner: general case

- The FSAI algorithm can be extended to a general matrix A. Two input sparsity patterns are required.
- First, lower and upper triangular matrices $\overline{G}_L$ and $\overline{G}_U$ are computed such that

$$(\overline{G}_L A)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i,j) \in \mathcal{S}_L,$$

$$(A\overline{G}_U)_{ij} = \delta_{i,j} \quad \text{for all} \quad (i,j) \in \mathcal{S}_U.$$

- Then D is obtained as the inverse of the diagonal of the matrix $\overline{G}_L A \overline{G}_U$, and the final nonsymmetric FSAI factors are given by $G_L = \overline{G}_L$ and $G_U = \overline{G}_U D$. The computation of the two approximate factors can be performed independently.
- This generalization is well defined if, for example, A is nonsymmetric positive definite. There is also theory that extends existence to special classes of matrices, including M- and H-matrices.

Sparse approximate inverses

Determining a good sparsity pattern

- Input pattern is expected to filter out entries of A^{-1} that contribute little to the quality of the preconditioner.
- For instance, it might be appropriate to ignore entries with a small absolute value, while retaining the largest ones. But, locations of large entries in A^{-1} are generally unknown, and this makes the a priori sparsity choice difficult.
- A a banded SPD matrix: the entries of A^{-1} are bounded in an exponentially decaying manner along each row or column: there exist $0 < \rho < 1$ and a constant c such that for all i, j

$$|(A^{-1})_{ij}| \leq c\rho^{|i-j|}.$$

  The scalars ρ and c depend on the bandwidth and $\kappa(A)$.
- A common choice for a general A is $\mathcal{S}_L + \mathcal{S}_U = \mathcal{S}\{A\}$, .
- An alternative strategy uses the Neumann series expansion of A^{-1}: using the pattern of a small power of A, i.e., $\mathcal{S}\{A^2\}$ or $\mathcal{S}\{A^3\}$.

Sparse approximate inverses

Factorized approximate inverses based on incomplete conjugation

- An alternative way: using incomplete conjugation (A-orthogonalization) in the SPD case and on incomplete A-biconjugation in the general case. For SPD matrices, the approach represents an approximate Gram-Schmidt orthogonalization that uses the A-inner product $\langle ., . \rangle_A$.
- Sparsity pattern not needed in advance.
- When A is a SPD matrix the AINV preconditioner is defined in the form

$$A^{-1} \approx M^{-1} = ZD^{-1}Z^T,$$

  Z is unit upper triangular, D is a diagonal matrix with positive entries.
- Practical implementations need to employ sparse matrix techniques. The left-looking scheme computes the j-th column z_j of Z as a sparse linear combination of the previous columns $z_1, \ldots, z_{j-1}$. The key is determining which multipliers (the α's in Steps 4 and 5 of the two algorithms, respectively) are nonzero and

Factorized approximate inverses based on incomplete conjugation

Algorithm (**AINV preconditioner (left-looking approach)**)

Input: *SPD matrix A and sparsifying rule.*
Output: *$A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.*

1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ ▷ *Initialise Z to hold the columns of the identity matrix*
2: ***for*** $j = 1 : n$ ***do***
3: ***for*** $k = 1 : j - 1$ ***do***
4: $\alpha = A_{k,1:n}\, z_j^{(k-1)} / d_{kk}$
5: $z_j^{(k)} = z_j^{(k-1)} - \alpha z_k^{(k-1)}$
6: *Sparsify* $z_j^{(k)}$ ▷ *Drop entries from* $z_j^{(k)}$
7: ***end for***
8: $d_{jj} = A_{j,1:n}\, z_j^{(j-1)}$
9: ***end for***
10: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$

Factorized approximate inverses based on incomplete conjugation

Algorithm (**AINV preconditioner (right-looking approach)**)

Input: *SPD matrix A and sparsifying rule.*
Output: *$A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.*

1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ ▷ *Initialise Z to hold the columns of the identity matrix*
2: ***for*** $j = 1 : n$ ***do***
3: $\quad d_{jj} = A_{j,1:n}\, z_j^{(j-1)}$
4: $\quad$ ***for*** $k = j + 1 : n$ ***do***
5: $\quad\quad \alpha = A_{j,1:n}\, z_k^{(j-1)} / d_{jj}$
6: $\quad\quad z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
7: $\quad\quad$ *Sparsify $z_k^{(j)}$* ▷ *Drop entries from $z_k^{(j)}$*
8: $\quad$ ***end for***
9: ***end for***
10: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$

AINV preconditioner: general case

- In the general case, the AINV preconditioner is given by an approximate inverse factorization of the form

$$A^{-1} \approx M^{-1} = W D^{-1} Z^T,$$

  where Z and W are unit upper triangular matrix and D is a diagonal matrix.
- Z and W are sparse approximations of the inverses of the L^T and U factors in the LDU factorization of A, respectively.

AINV preconditioner: general case

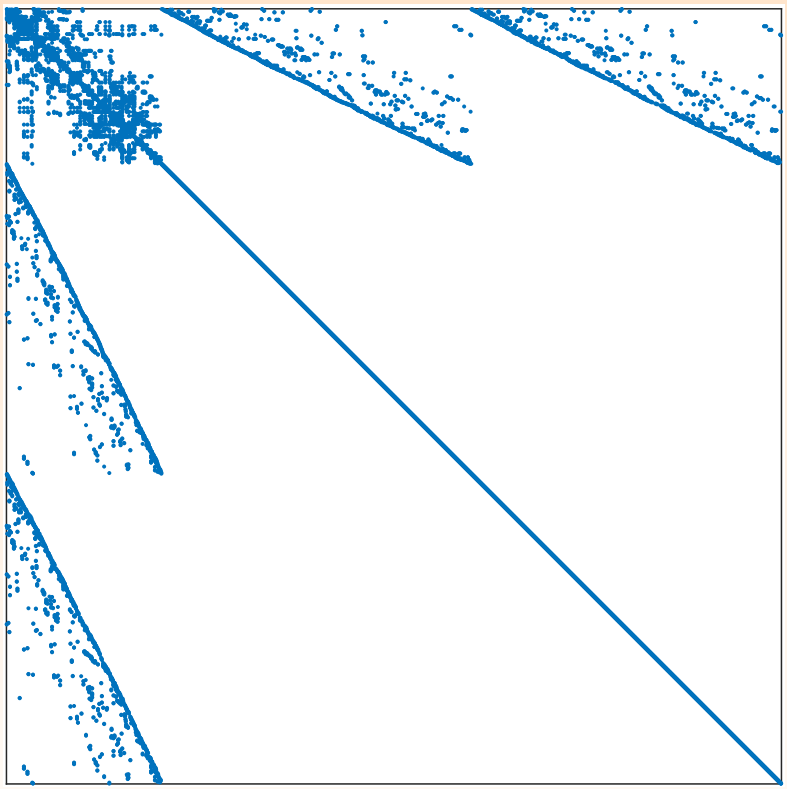
Algorithm (**Nonsymmetric AINV preconditioner (right-looking approach)**)

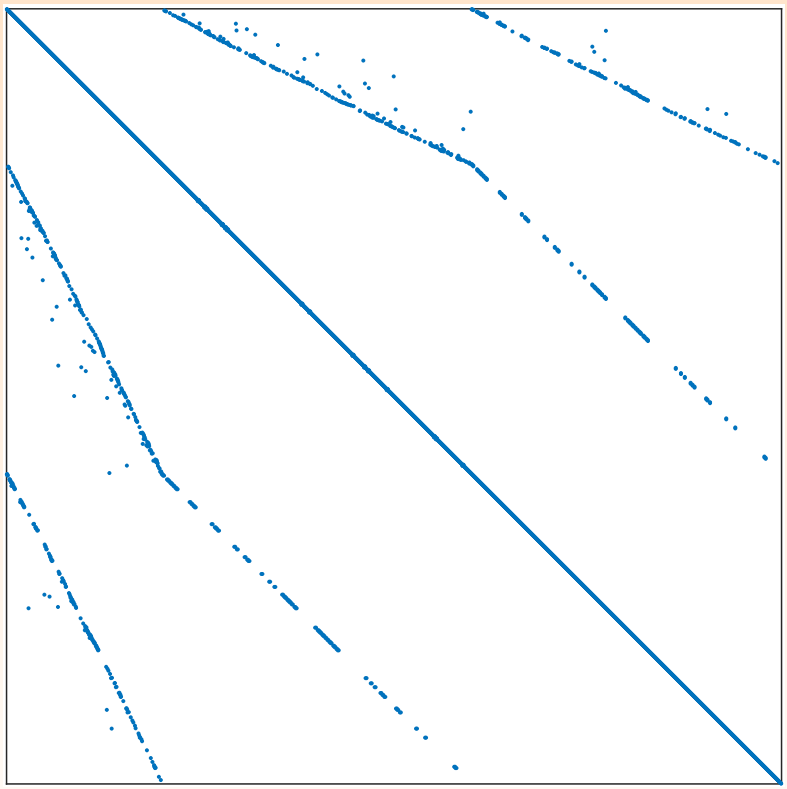
1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$ *and* $[w_1^{(0)}, \ldots, w_n^{(0)}] = [e_1, \ldots, e_n]$
2: ***for*** $j = 1 : n$ ***do***
3: $\quad d_{jj} = (A_{1:n,j})^T z_j^{(j-1)}$ *or* $d_{jj} = A_{j,1:n}\, w_j^{(j-1)}$
4: $\quad$ ***for*** $k = j+1 : n$ ***do***
5: $\quad\quad \alpha = (A_{1:n,j})^T z_k^{(j-1)} / d_{jj}$
6: $\quad\quad z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
7: $\quad\quad$ *Sparsify* $z_k^{(j)}$ ▷ *Drop entries from* $z_k^{(j)}$
8: $\quad\quad \beta = A_{j,1:n}\, w_k^{(j-1)} / d_{jj}$
9: $\quad\quad w_k^{(j)} = w_k^{(j-1)} - \beta w_j^{(j-1)}$
10: $\quad\quad$ *Sparsify* $w_k^{(j)}$ ▷ *Drop entries from* $w_k^{(j)}$
11: $\quad$ ***end for***
12: ***end for***
13: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$ *and* $W = [w_1^{(0)}, \ldots, w_n^{(n-1)}]$

Sparse approximate inverses

AINV preconditioner

- Matrix A, AINV preconditioner

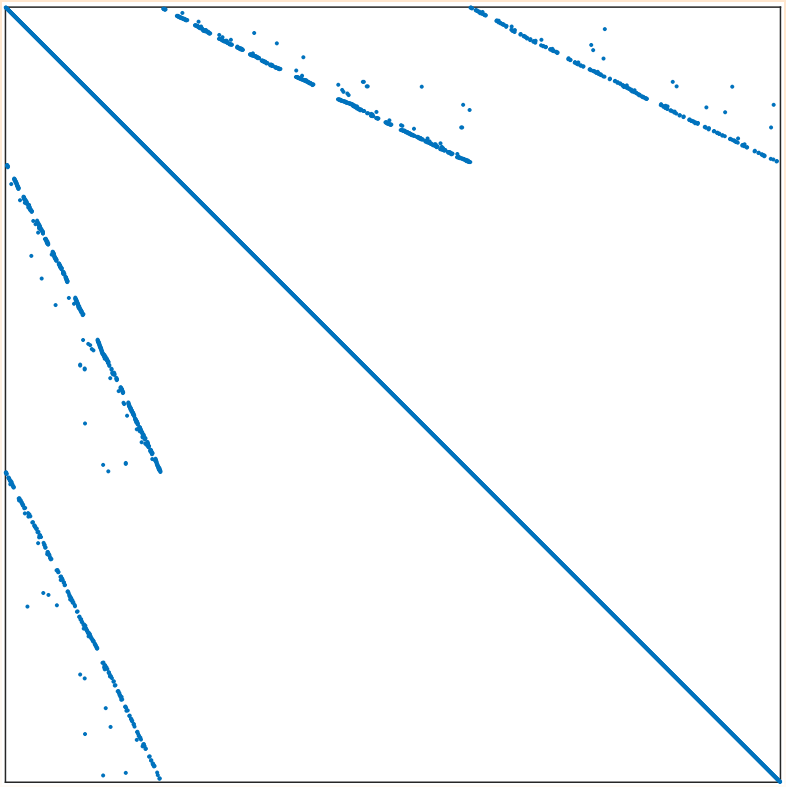


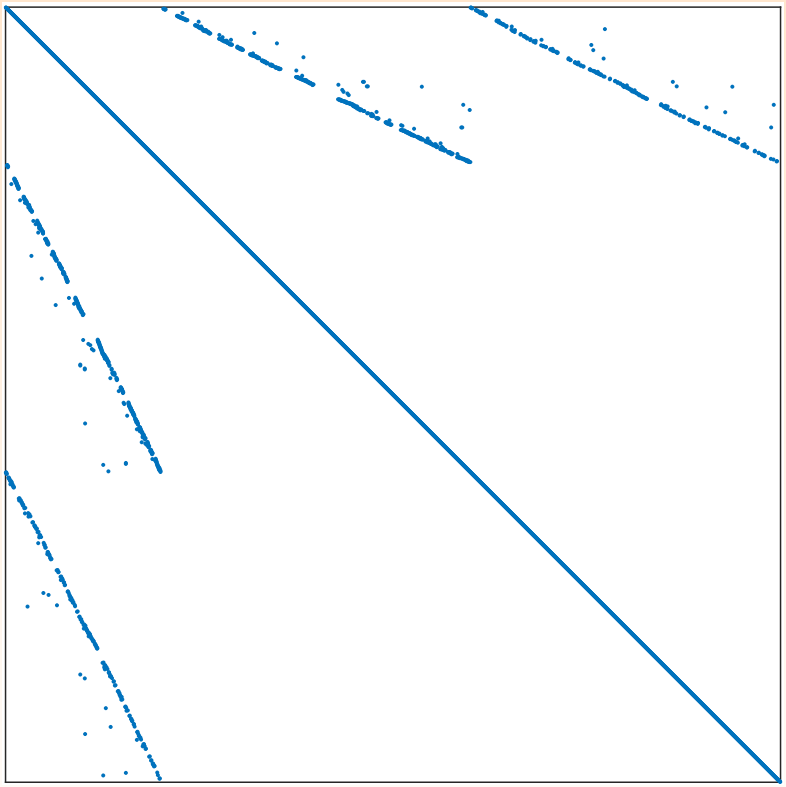


Sparse approximate inverses

AINV preconditioner

- ILUT, inverse ILUT





Sparse approximate inverses

SAINV: stabilization of the AINV method

- The following result is analogous to the SPD case.

Theorem

If A is a nonsingular M- or H-matrix then the AINV factorization of A does not break down.

- For more general matrices breakdown can happen because of the occurrence of zero d_{jj} or, in the SPD case, negative d_{jj}.
- In practice, exact zeros are unlikely but very small d_{jj} can occur (near breakdown), which may lead to uncontrolled growth in the size of entries in the incomplete factors and, because such entries are not dropped when using a threshold parameter, a large amount of fill-in.
- The next theorem indicates how breakdown can be prevented when A is SPD through reformulating the A-orthogonalization.

SAINV: stabilization of the AINV method

Theorem

Consider AINV algorithm with no sparsification (Step 7 is removed). The following holds

$$A_{j,1:n}\, z_k^{(j-1)} \equiv e_j^T A z_k^{(j-1)} = \langle z_j^{(j-1)}, z_k^{(j-1)} \rangle_A, \quad 1 \le j \le k \le n.$$

Proof.

Because $AZ = Z^{-T} D$ and $Z^{-T} D$ is lower triangular, entries 1 to $j-1$ of the vector $A z_k^{(j-1)}$ are equal to zero. Z is unit upper triangular so entries $j+1$ to n of its j-th column $z_j^{(j-1)}$ are also equal to zero. Thus, $z_j^{(j-1)}$ can be written as the sum $z + e_j$, where entries j to n of th vector z are zero. The result follows. □

SAINV: stabilization of the AINV method

Algorithm (**SAINV preconditioner (right-looking approach)**)

Input: *SPD matrix A and sparsifying rule.*
Output: *$A^{-1} \approx ZD^{-1}Z^T$ with Z a unit upper triangular matrix and D a diagonal matrix with positive diagonal entries.*

1: $[z_1^{(0)}, \ldots, z_n^{(0)}] = [e_1, \ldots, e_n]$
2: ***for*** $j = 1 : n$ ***do***
3: $\quad d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A$
4: $\quad$ ***for*** $k = j + 1 : n$ ***do***
5: $\quad\quad \alpha = \langle z_k^{(j-1)}, z_j^{(j-1)} \rangle_A / d_{jj}$
6: $\quad\quad z_k^{(j)} = z_k^{(j-1)} - \alpha z_j^{(j-1)}$
7: $\quad\quad$ *Sparsify* $z_k^{(j)}$ ▷ *Drop entries from* $z_k^{(j)}$
8: $\quad$ ***end for***
9: ***end for***
10: $Z = [z_1^{(0)}, \ldots, z_n^{(n-1)}]$

Sparse approximate inverses

From AINV to Cholesky

- The factors Z and D obtained with no sparsification can be used to compute the square root-free Cholesky factorization of A.
- The L factor of A and the inverse factor Z computed using AINV Algorithm without sparsification satisfy

$$AZ = LD \quad \text{or} \quad L = AZD^{-1}.$$

- Using $d_{jj} = \langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A$, and equating corresponding entries of AZD^{-1} and L gives

$$l_{ij} = \frac{\langle z_j^{(j-1)}, z_i^{(j-1)} \rangle_A}{\langle z_j^{(j-1)}, z_j^{(j-1)} \rangle_A}, \quad 1 \leq j \leq i \leq n.$$

- Thus, the SAINV algorithm generates the L factor of the square root-free Cholesky factorization of A as a by-product of orthogonalization in the inner product $\langle ., . \rangle_A$ at no extra cost and without breakdown.

Sparse approximate inverses

Stabilization-like for general A

- The stabilization strategy can be extended to the nonsymmetric AINV algorithm using the following result.

Theorem

Consider nonsymmetric AINV Algorithm with no sparsification (Steps 7 and 10 removed). The following identities hold:

$$A_{j,1:n}\, z_k^{(j-1)} \equiv e_j^T A z_k^{(j-1)} = \langle w_j^{(j-1)},\, z_k^{(j-1)} \rangle_A,$$

$$(A_{1:n,j})^T w_k^{(j-1)} \equiv e_j^T A^T w_k^{(j-1)} = \langle z_j^{(j-1)},\, w_k^{(j-1)} \rangle_A, \quad 1 \le j \le k \le n.$$

- The nonsymmetric SAINV algorithm obtained using this reformulation can improve the preconditioner quality but it is not guaranteed to be breakdown free.

Approximate inverse by global iterations

Consider one-dimensional Newton-Raphson iterations to find a scalar value p which is the root of a given function f, that is

$$f(p) = 0.$$

The method approaches p by a sequence of approximations $p_0, p_1, \ldots$. Consider a tangent of f at p_k for some integer $k \geq 0$ in the following form

$$y = f'(p_k)p_k + b. \tag{102}$$

The tangent crosses $(p_k, f(p_k))$ and this can be put down as

$$f(p_k) = f'(p_k)p_k + b. \tag{103}$$

This implies

$$b = f(p_k) - f'(p_k)p_k \tag{104}$$

and we get a function of x given by

$$y = f'(x)x + f(p_k) - f'(p_k)p_k. \tag{105}$$

Approximate factorizations, splitting and preconditioning

Assume that the root is achieved at p_{k+1}. Then

$$0 = f'(p_{k+1})p_{k+1} + f(p_k) - f'(p_k)p_k \tag{106}$$

and therefore

$$p_{k+1} = p_k - \frac{f(p_k)}{f'(p_k)}. \tag{107}$$

For f beeing the function of the inverse given by

$$f(x) = 1/x - a$$

we have

$$p_{k+1} = p_k - \frac{1/p_k - a}{-1/p_k^2} = p_k(2 - ap_k). \tag{108}$$

Finding the matrix inverse in case it is well-defined:

$$G_{i+1} = G_i(2I - AG_i),\ i = 1, \ldots$$

for the sequence of non-factorized approximate inverses $G_0, \ldots$. The main problem: G is for irreducible A fully dense.

Approximate factorizations, splitting and preconditioning

Consider the computation of the j-th diagonal entry and assume the exactly computed quantities. The computation from the formula

$$d_j = A_{jj} + A^T_{1:j-1,j} w_j + A_{j,1:j-1} z_j + w_j^T A_{j-1} z_j \tag{109}$$

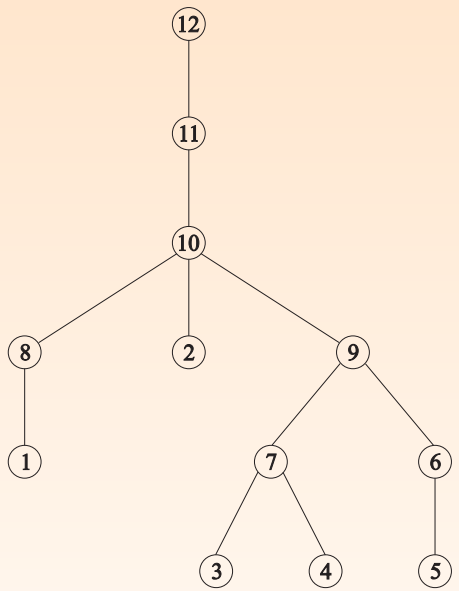
can be easily replaced by the mathematically equivalent formula which we used in the algorithms using biconjugation computation:

$$d_j = A_{j,1:j-1} z_j \text{ or } d_j = A^T_{1:j-1,j} w_j. \tag{110}$$

Notes on parallel approaches

1. Shared memory computers

- 1st level of parallelism: tree structure of the decomposition.
- 2nd level of parallelism: local node parallel enhancements.



- Both may/should be coordinated.
- **Tree parallelism** potential decreases towards its root.
- Potential for the local parallelism (larger dense matrices) increases towards the root.

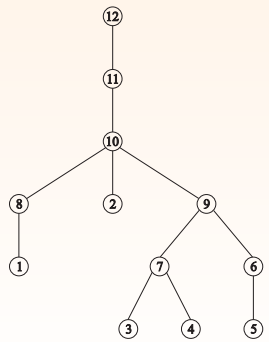
Decomposition and computer architectures: 1st level of parallelism

Two basic possibilities for the tree parallelism

- Dynamic task scheduling on shared memory computers
- Direct static mapping: subtree to subcube

1. Dynamic task scheduling on shared memory computers

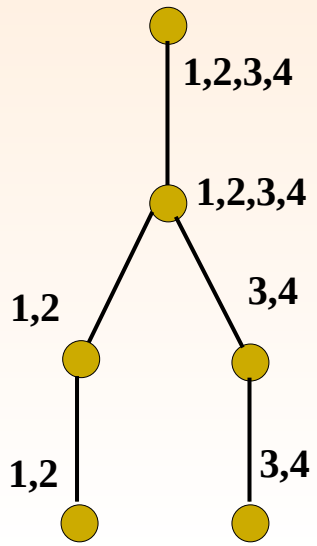
- Dynamic scheduling of the tasks
- Each processor selects a task
- Again, problem of elimination tree reordering
- Not easy to optimize memory, e.g., in the multifrontal method



Decomposition and computer architectures: 1st level of parallelism: II

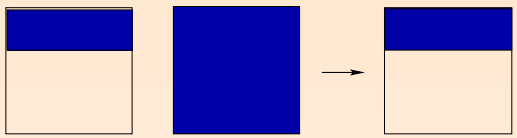
2. Direct static mapping: subtree to subcube

- Recursively map processors to the tree parts from the top
- Various ways of mapping.
- Note: In the SPD (non-pivoting) case the arithmetic work can be computed and considered
- Localized communication
- More difficult to share the work among processors in more complex models

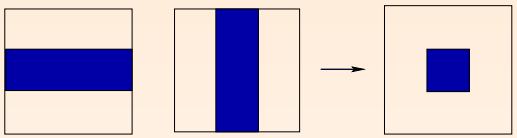


Decomposition and computer architectures: 2nd level of parallelism

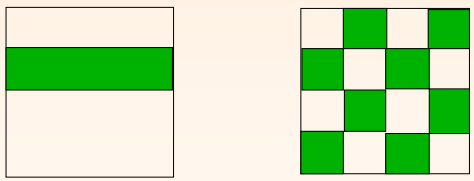
- Block Cholesky/LU factorization
- BLAS / parallel BLAS operations



1D partitioning



2D partitioning



1D and 2D block cyclic distribution

(Only illustrative figures for the talk!)

Decomposition and computer architectures: Distributed memory parallelism

Basic classical parallelization approaches (consider Cholesky)

- Fan-in approach
  - Demand-driven column-based algorithm
  - Required data are aggregated updates asked from previous columns
- bf Fan-out approach
  - Data-driven column-based algorithm
  - Updates are broadcasted once computed and aggregated
  - Historically the first approach; greater interprocessor communication than fan-in
- Multifrontal approach
  - Example: MUMPS