Sparse Matrix Computation

Esmond G. Ng
(EGNg@lbl.gov)
Lawrence Berkeley National Laboratory

The First International Summer School on Numerical Linear Algebra
August 2006

Outline

- Sparse matrices
  - What they are
  - Where they come from
  - Simple operations and representations
- Sparse Gaussian elimination
  - Sparsity
  - Modeling and analysis
  - Ordering
  - Numerical computation

Goals

- Not intend to be complete.
- Cover the basics.
- Show that it is multi-facet.
  - Theoretical
  - Algorithmic
  - Computational
- Show that it is multi-disciplinary.
  - Numerical linear algebra
  - Combinatorial algorithms
  - Computer science
  - Computer architecture

What are sparse matrices

- Most popular definition:
  - A matrix is sparse if most of the elements are zero.
    - A subjective definition …
- A better definition:
  - A matrix is sparse if there is substantial saving in storage, operations, or execution time when the zero elements are exploited.

Quiz

- Let u and v be two sparse vectors of length n.

- Suppose u has s nonzeros and v has t nonzeros.
- Suppose s, $t \ll n$, and $s < t$.

- How many operations are required to compute $u^T v$?
  - Counting additions, multiplications, and comparisons.

Where do sparse matrices come from

computational fluid dynamics, finite-element methods, statistics, time/frequency domain circuit simulation, dynamic and static modeling of chemical processes, cryptography, magneto-hydrodynamics, electrical power systems, differential equations, quantum mechanics, structural mechanics (buildings, ships, aircraft, human body parts…), heat transfer, MRI reconstructions, vibroacoustics, linear and non-linear optimization, financial portfolios, semiconductor process simulation, economic modeling, oil reservoir modeling, subsurface flow, astrophysics, crack propagation, Google page rank, 3D computer vision, image processing, tomography, cell phone tower placement, multibody simulation, model reduction, nano-technology, acoustic radiation, density functional theory, quadratic assignment, elastic properties of crystals, natural language processing, DNA electrophoresis, information retrieval/data mining, nuclear structure calculations, statistical calculations, economic modeling, …

What do we do with sparse matrices

- … Sparse matrices are at the heart of many scientific and engineering simulations.
  - Solution of systems of linear equations (square, under/over-determined)

    $A\,x = b$
  - Eigenvalue analysis

    $(A - \lambda I)\, x = 0$

    $F(\lambda, x) = 0$
  - Singular value decomposition

    $A = U \Sigma V^T$
  - Many others

Large sparse matrices

- These sparse matrix problems tend to be very large … because of
  - need for high-fidelity modeling, and
  - availability of high performance computing resources.

- Some characteristics …
  - # of unknowns can reach hundreds of millions.
  - Some can be highly structured.
  - Some are badly scaled.
  - Some can be very ill-conditioned.
  - Some even require extra precision arithmetic.

What do we want to do

- Efficient and robust solutions of these sparse matrix problems are important.
  - Time to solution; parallel computing may be needed.
  - Enabling domain scientists to focus on their scientific applications rather than linear algebra solvers.
  - Ensuring the success of scientific simulations.

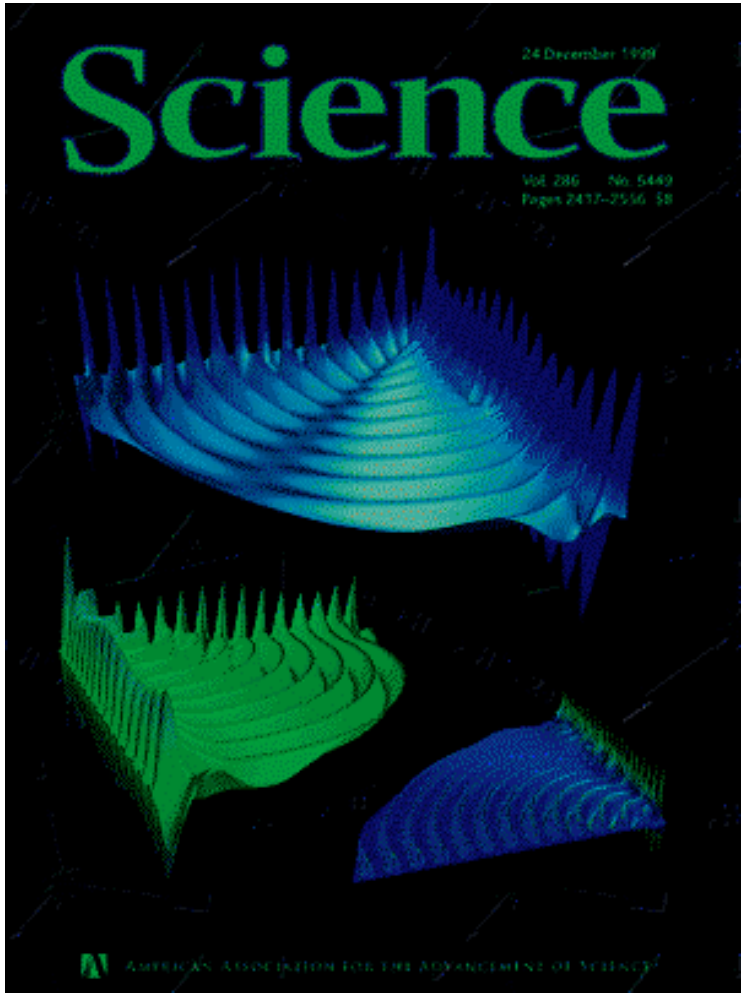
- Old and new are needed:
  - Need changes and improvements to existing algorithms/codes.
  - Investigate new algorithms/codes.

Examples of large-scale scientific simulations

- We will look at a few examples, which illustrate the role of sparse matrices.
  - Atomic Physics.
  - Fusion.
  - Cosmology
  - Accelerator design.
  - Structural biology.

Atomic physics

- Rescigno, Baertschy, Isaacs, McCurdy, Science, Dec 24, 1999.
- First solution to quantum scattering of 3 charged particles.
- The main computational kernel is the solution of sparse complex nonsymmetric linear systems.
  - PDE's are solved using a high-order finite difference scheme.
  - A low order finite difference approximation is used as a preconditioner.

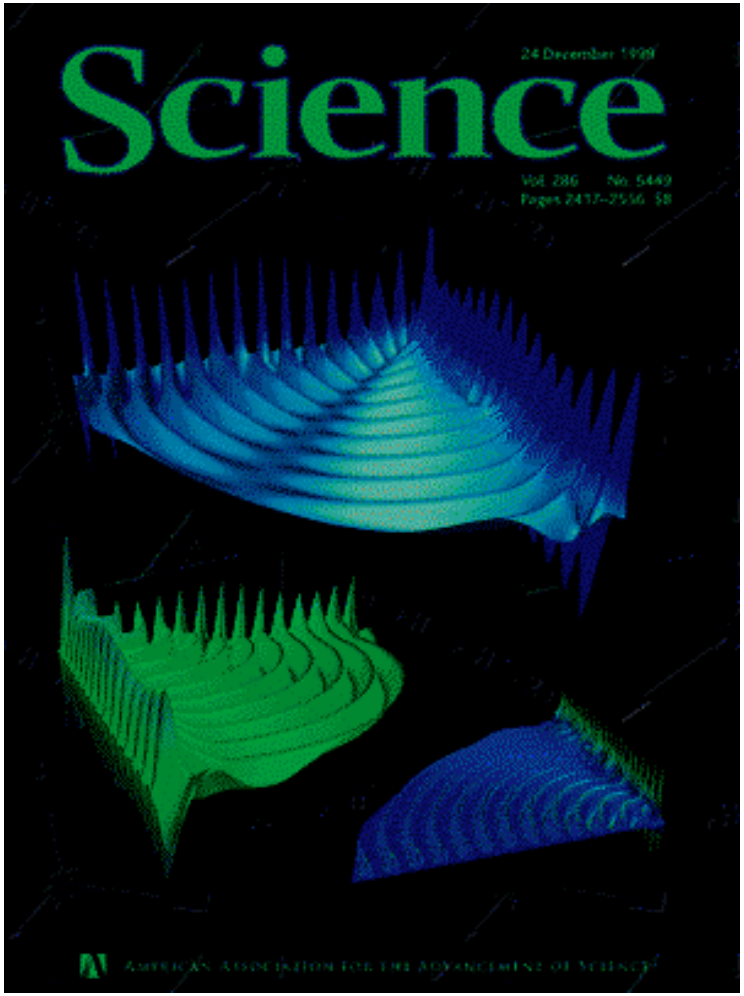


Atomic physics

- Resulting kernel is:

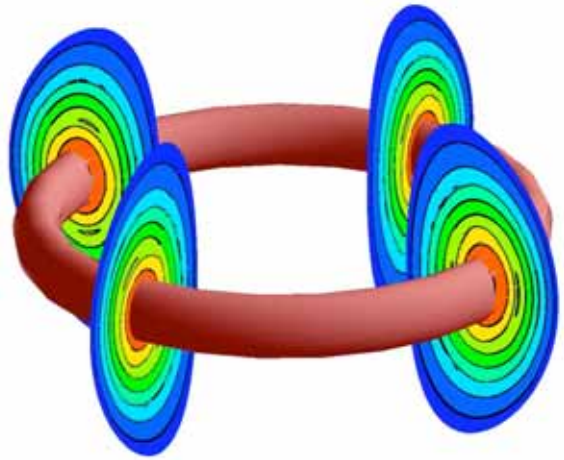
$$M^{-1}Ax = M^{-1}b$$

- M, obtained from a low-order finite difference scheme, is factored using sparse Gaussian elimination.
- The preconditioned linear system is solved using one of the nosymmetric iterative solvers, such as biconjugate gradient or quasi minimal residual.
- Largest linear system solved has 1.79 million unknowns.

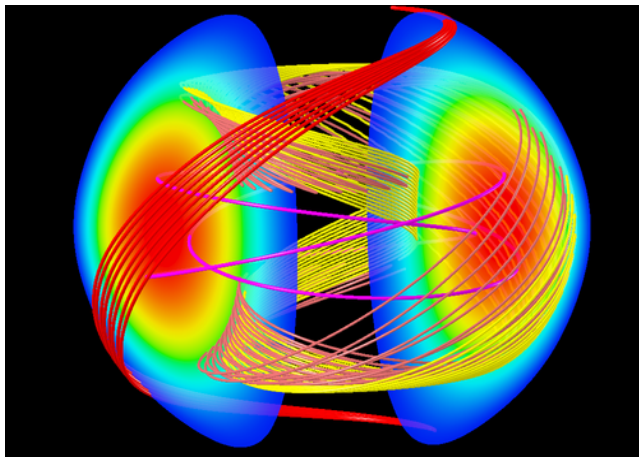


Large-scale fusion simulations

- NIMROD is a parallel simulation code for fluid-based modeling of nonlinear macroscopic electromagnetic dynamics in fusion plasmas.



- The kernel involves the solution of very ill-conditioned sparse linear systems.
- Iterative methods with preconditioning exhibit poor convergence.



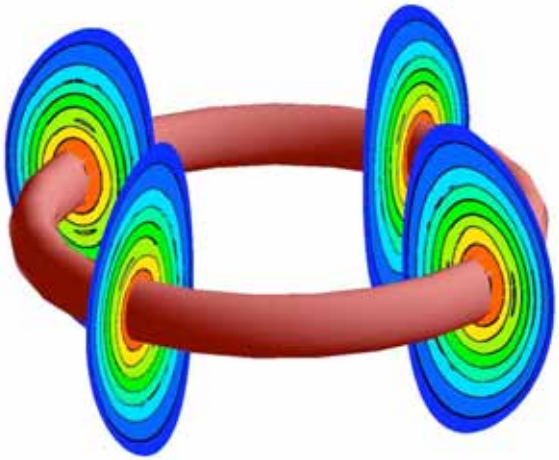
- Explore sparse Gaussian elimination as an alternative.

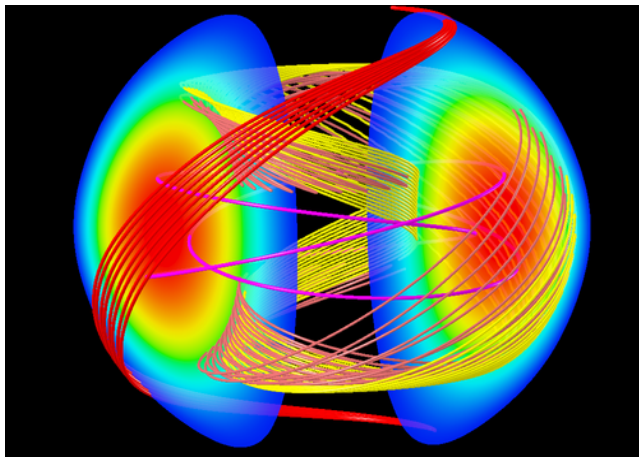
Large-scale fusion simulations

- Gaussian elimination has resulted in >100x improvements.

- The linear systems are large and sparse, with millions of unknowns.
  - Parallel algorithms are required.

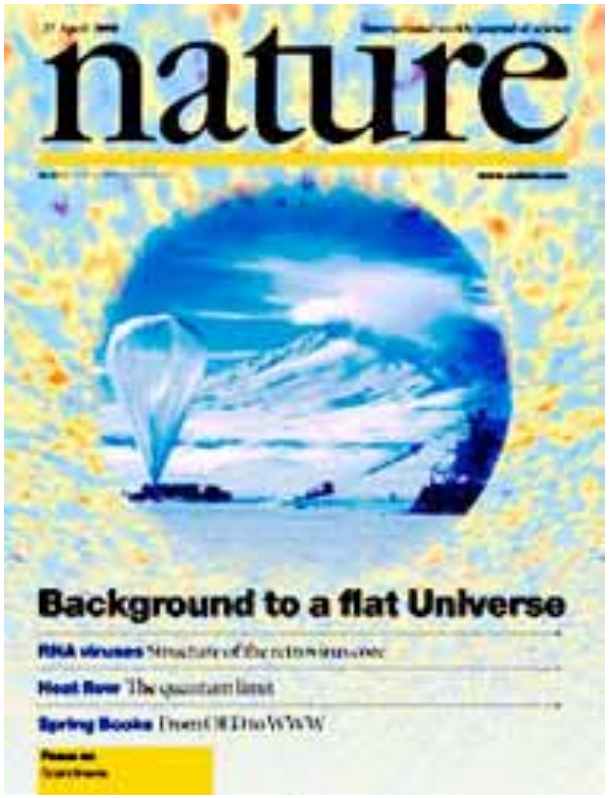
- Physics-based preconditioners can be constructed in some instances.
  - The preconditioned linear systems are solved using conjugate gradient iterations.
  - Gaussian elimination is needed to handle the preconditioners.

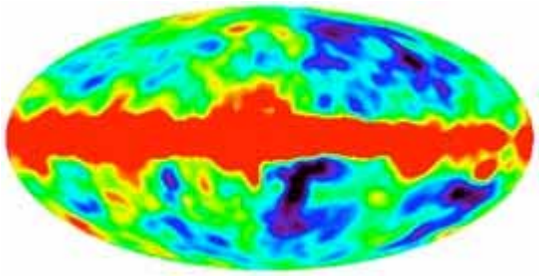


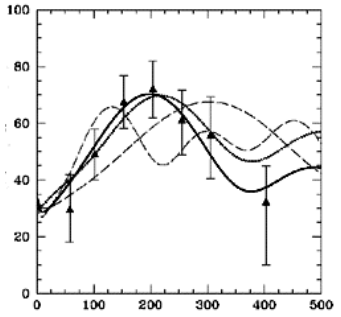


A flat universe

- International efforts in understanding the origin and geometry of the universe.
- Analysis of data from the 1997 North American test flight of BOOMERanG shows a pronounced peak in the CMB "power spectrum" at an angular scale of about one degree -- strong evidence that the universe is flat, and suggest the existence of a cosmological constant.

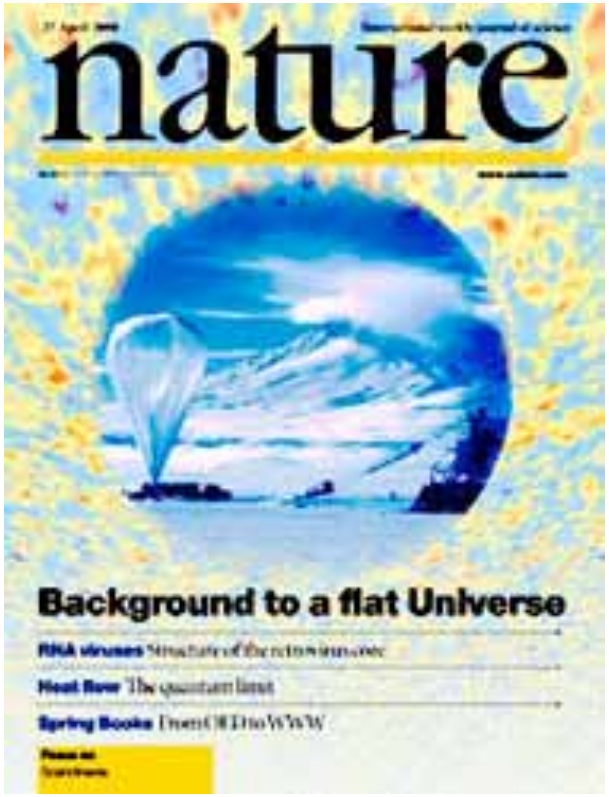






A flat universe

- Analysis is based on the maximum likelihood approach.
- Various statistics have to be computed, including entries of the covariance matrix.
  - Need dense matrix inversion.
- Very compute-intensive …
  - BOOMERanG: 26K pixels, 10.8 Gb, 10^{15} flops
    - Dimensions of matrices ≈ 45,000 to 50,000.
  - PLANCK: 10M pixels, 1.6 Pb, 10^{23} flops
    - New algorithms are needed.
    - In particular, approximations using sparse matrices or structured matrices are required.



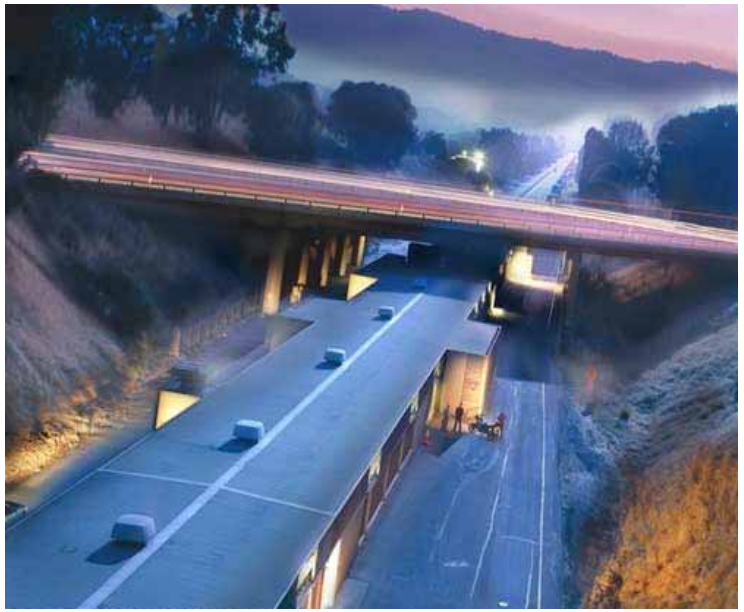
Sparse Matrix Computation

Esmond G. Ng
(EGNg@lbl.gov)
Lawrence Berkeley National Laboratory

The First International Summer School on Numerical Linear Algebra
August 2006

Accelerator science and technology

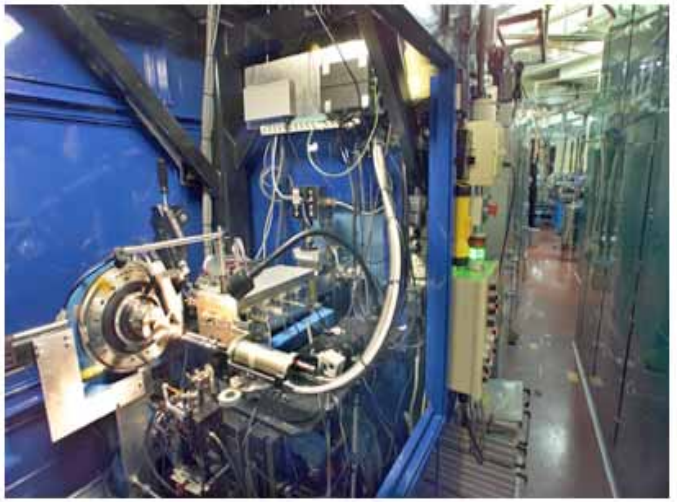
- Accelerators are important.
  - Research in particle physics.
  - Fundamental to understanding of structure of matter.



- Accelerators are expensive.
  - High cost in construction, operations, and maintenance.
  - Represent major federal investment.

Accelerator science and technology

- Accelerator modeling and simulations are indispensable.
  - Understanding the science of accelerators for safe operations.
  - Improving performance and reliability of existing accelerators.
  - Designing next generation of accelerators accurately and optimally.



ALS Beamline 8.3.1

Accelerator Science and Technology

- From SLAC Web Site (April 2003) …

SLAC Experiment Identifies New Subatomic Particle

Physicist Antimo Palano representing the BABAR experiment presented the evidence for the identification of a new subatomic particle named Ds (2317) to a packed auditorium on Monday, April 28 at SLAC. Initial studies indicate that the particle is an unusual configuration of a "charm" quark and a "strange" anti-quark.

Modeling accelerator structures

- The design of accelerator structures requires the solution of Maxwell's equations, which govern how the electric and magnetic fields interact.

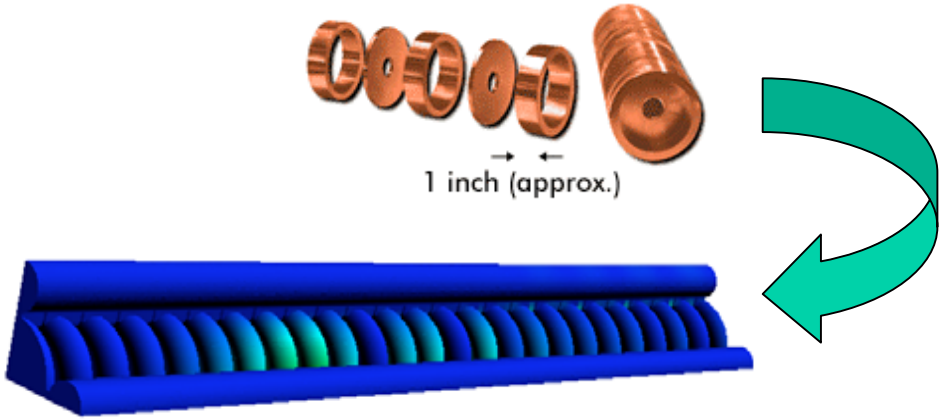
$$\nabla \times E = -\frac{\partial B}{\partial t} \quad ; \quad \nabla \times H = -\frac{\partial D}{\partial t}$$

$$\nabla \cdot D = 0 \quad ; \quad \nabla \cdot B = 0$$

$$D = \varepsilon E \quad ; \quad B = \mu H$$

- Finite element discretization in frequency domain leads to a large sparse generalized eigenvalue problem.

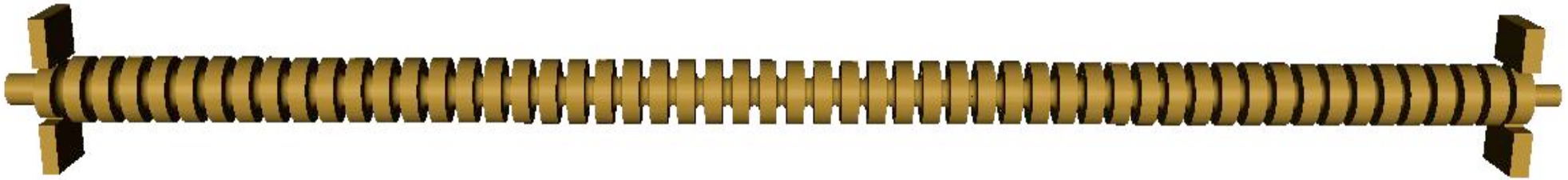
$$Kx = \lambda Mx \; ; K \geq 0, M > 0$$



Modeling accelerator structures

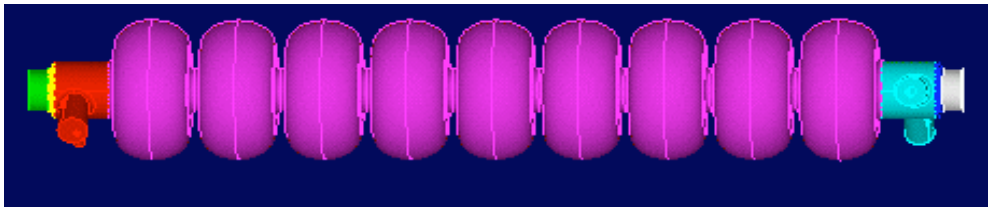
- Design of accelerator structures.
  - Modeling of a single accelerator cell suffices.
    - Relatively small eigenvalue problem.
  - There is an optimization problem here …
    - But need fast and reliable eigensolvers at every iteration + other tools.

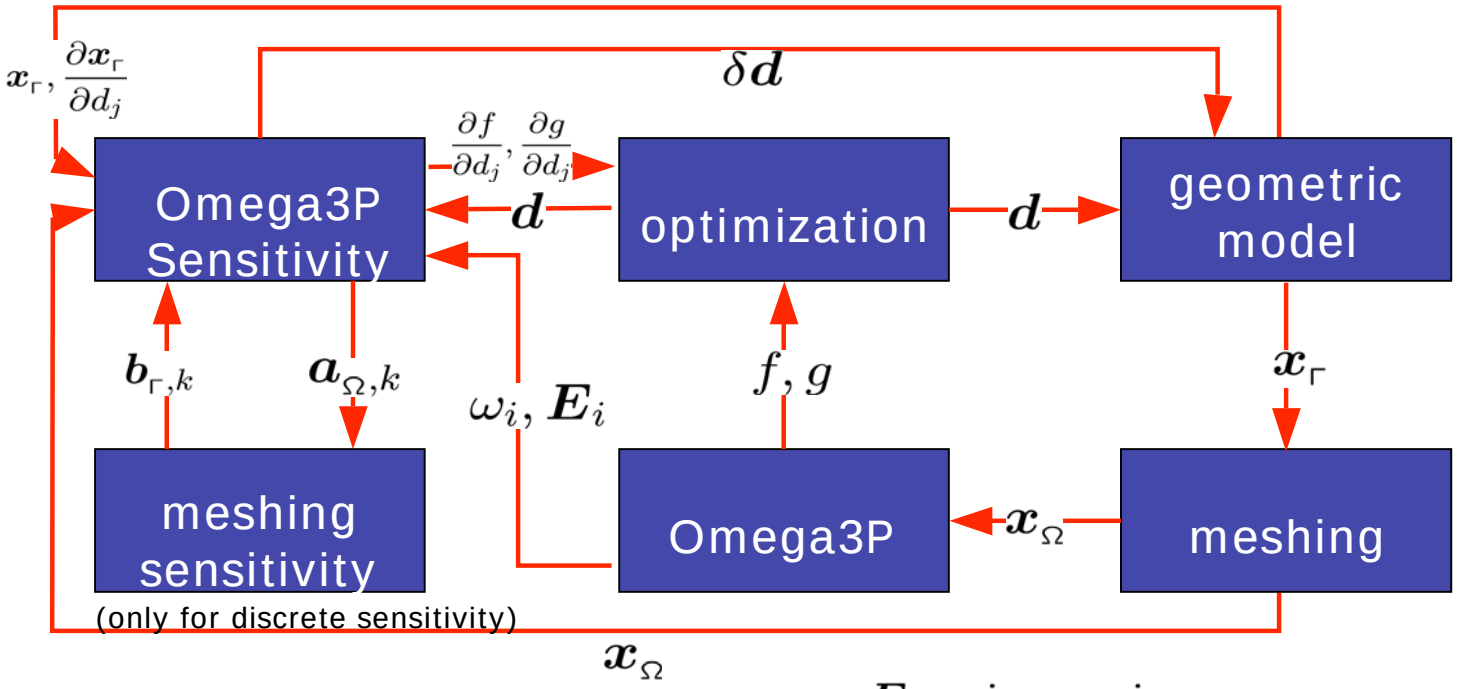
- Understanding the wake field in the structure requires the modeling of the full structure.
  - Need to compute a large number of frequency modes.



Shape optimization of accelerator structures

LBNL, CMU, Columbia, LLNL, SLAC, SNL Collaboration





d : design vector
x_Γ : surface grid
x_Ω : volume grid
f : objective
g : constraint

ω_i, E_i : eigenpairs
δd : perturbed design
$a_{\Omega,k}$: volume grid function
$b_{\Gamma,k} \leftarrow A_{\Omega\Gamma}^T A_{\Omega\Omega}^{-T} a_{\Omega,k}$
$A_{\Omega\Omega}, A_{\Omega\Gamma}$: mesh Hessians

Shape optimization of accelerator structures

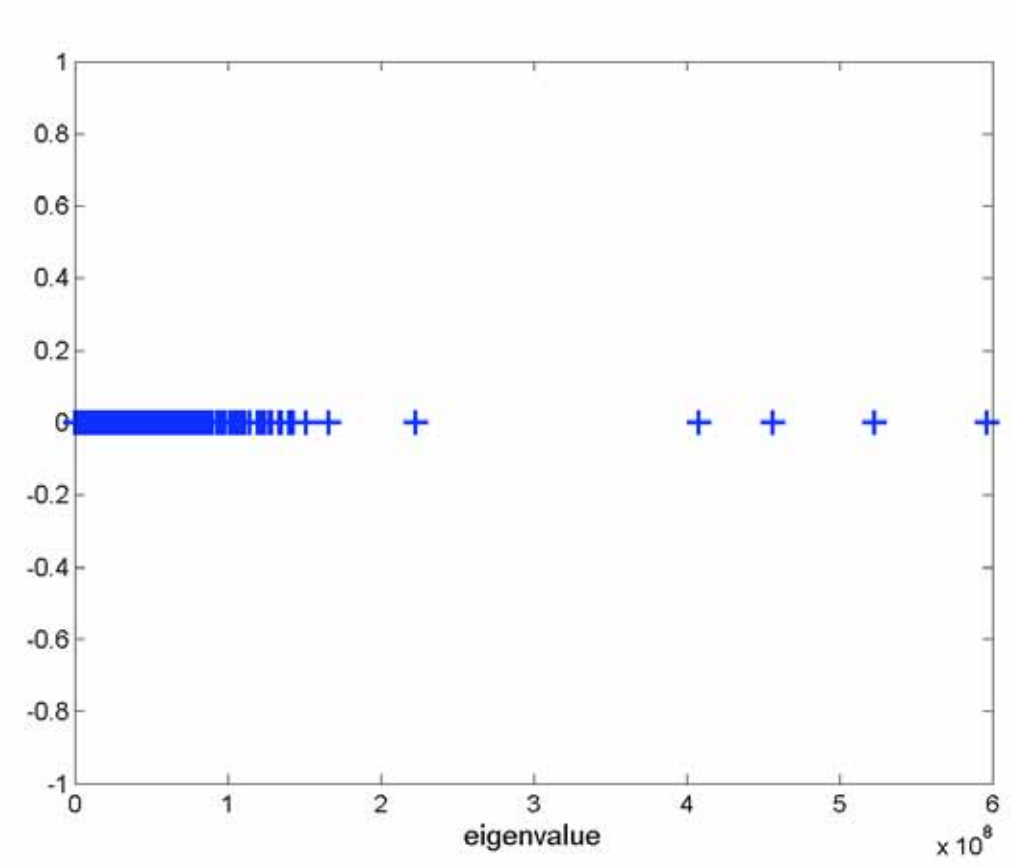
- Two computational kernels:
  - Large sparse eigenvalue calculations.
  - Sensitivity analysis of eigenpairs.
    - Need to compute the adjoint variables.
    - Solution of structured indefinite linear systems.

$$\begin{bmatrix} K - \lambda M & Mv \\ v^T M^T & 0 \end{bmatrix}$$

    - $K - \lambda M$ is practically singular.
      - Need efficient and robust algorithms to solve the adjoint linear systems.

Challenges in eigenvalue calculations

- 3-D structures, high resolution simulations ⇒ extremely large matrices.
  - Need very accurate interior eigenvalues that have relatively small magnitudes.
    - Tolerate only 0.01% error.
  - These eigenvalues are tightly clustered.
  - When losses in structures are considered, the problems will become complex symmetric.



Large-scale eigenvalue calculations

- Shift-invert Lanczos algorithm.
  - Ideal for computing interior and clustered eigenvalues.

$$K x = \lambda M x \quad \rightarrow \quad M\boxed{(K - \sigma M)^{-1}} M x = \mu M x$$

  - Need accurate solution of sparse linear systems …
  - Need special care due to extreme scale of problems.
  - Parallel implementations are needed to speed the solution process.
  - Issues to be resolved …
    - Sparsity concerns
    - Memory constraints
    - Serial bottlenecks
    - Accuracy

Alternative eigenvalue solvers …

- Size of eigenvalue problems is expected to increase substantially in near future …
  - Can be as large as 100 million degrees of freedom.

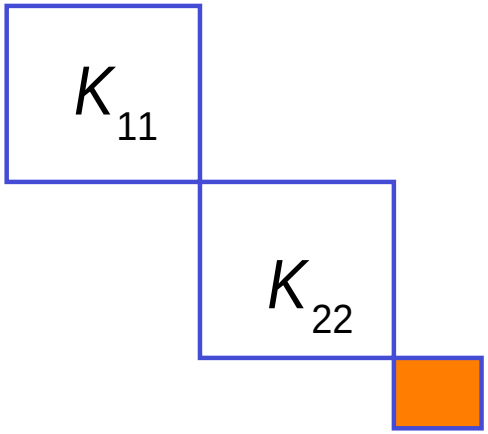
- The solution of linear systems $(K-\sigma M)\,x = b$ will become the bottleneck.
  - Memory …
  - In parallel implementations, communication requirements.
- There are needs to look for alternative solutions.

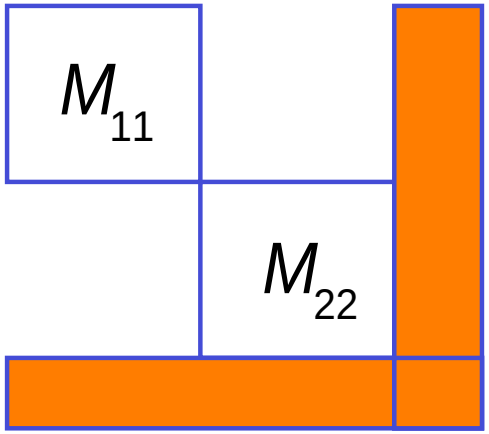
Alternative eigenvalue solvers …

- AMLS (Automatic Multi-Level Substructuring).
  - [Yang, Gao, Bai, Li, Lee, Husbands, Ng (2005)].
  - An eigenvalue solver proposed by Bennighof for frequency response analysis.
  - Analogous to "domain-decomposition" techniques for linear systems.
  - Issues:
    - Analysis of the approximation properties.
    - Memory saving optimizations.
    - Developed mode selection strategies.
    - Null space deflation.
  - Combine expertise in sparse eigenvalue calculations and sparse Gaussian elimination.

Algebraic sub-structuring

- Partition and congruence transform:



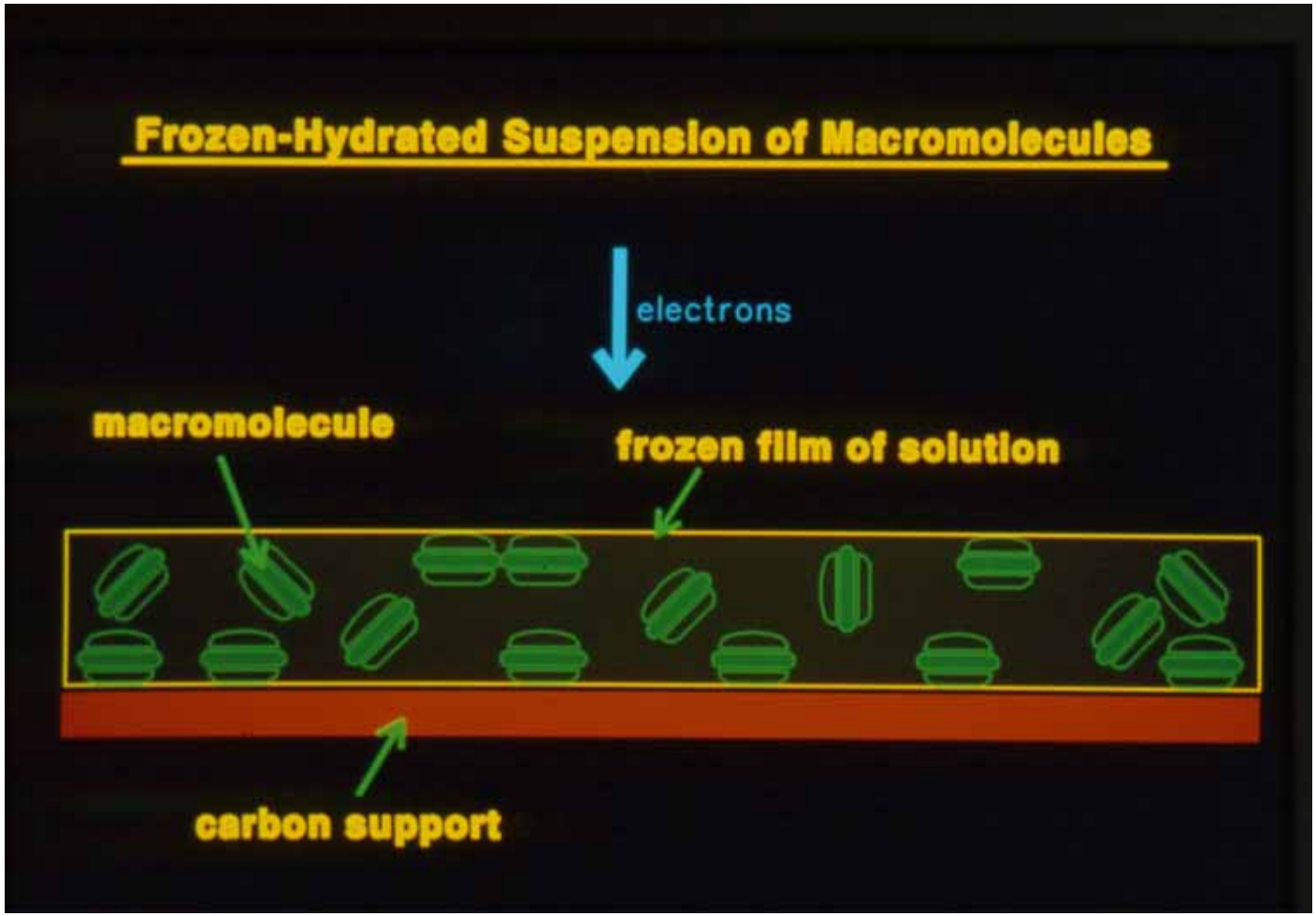


$$\hat{K} = L^{-1}KL^{-T} \qquad \hat{M} = L^{-1}ML^{-T}$$

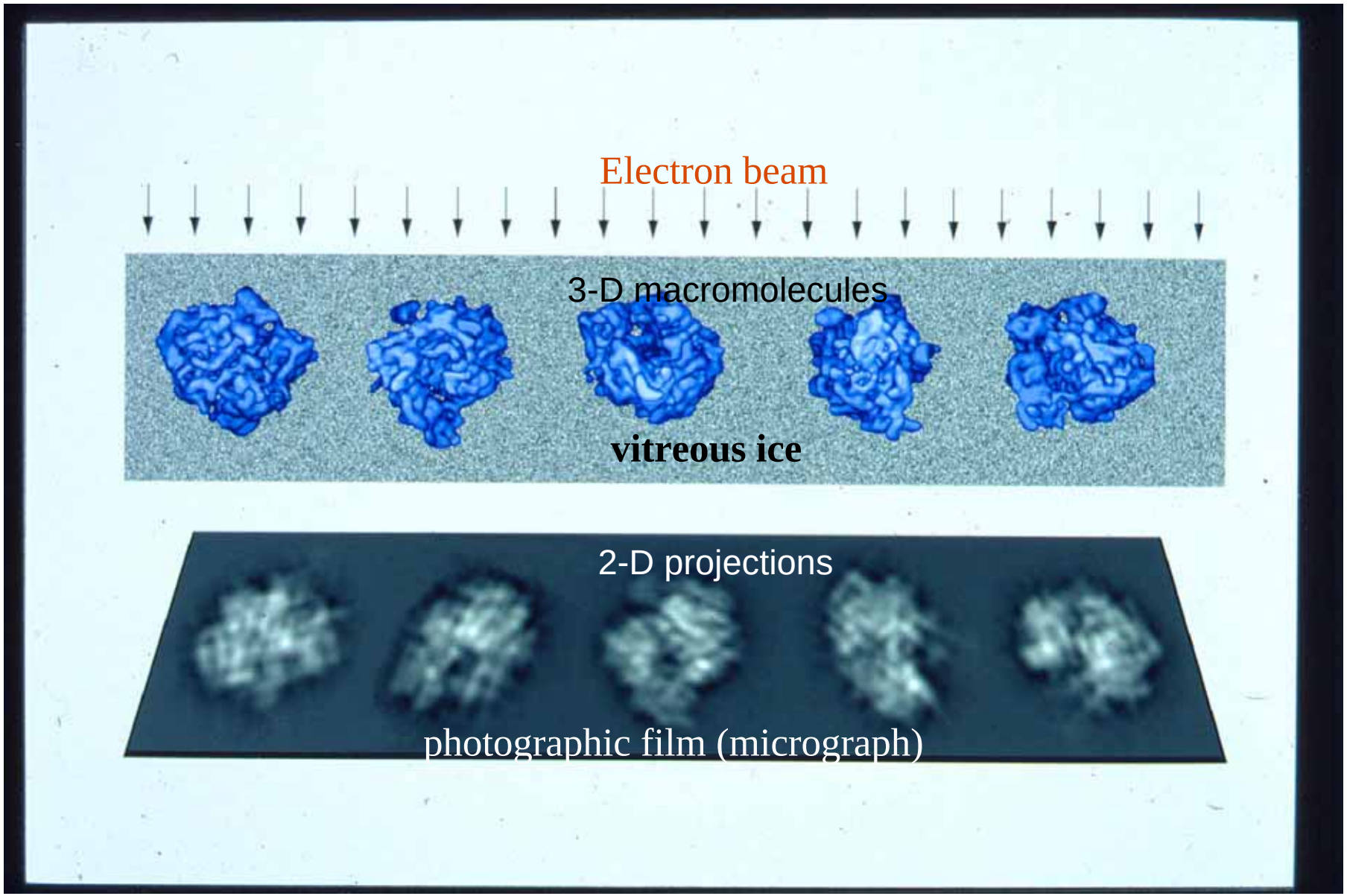
- Subspace assembly:

$$S = \begin{pmatrix} S_1 & & \\ & S_2 & \\ & & I \end{pmatrix}$$

Structural biology - electron cryo microscopy

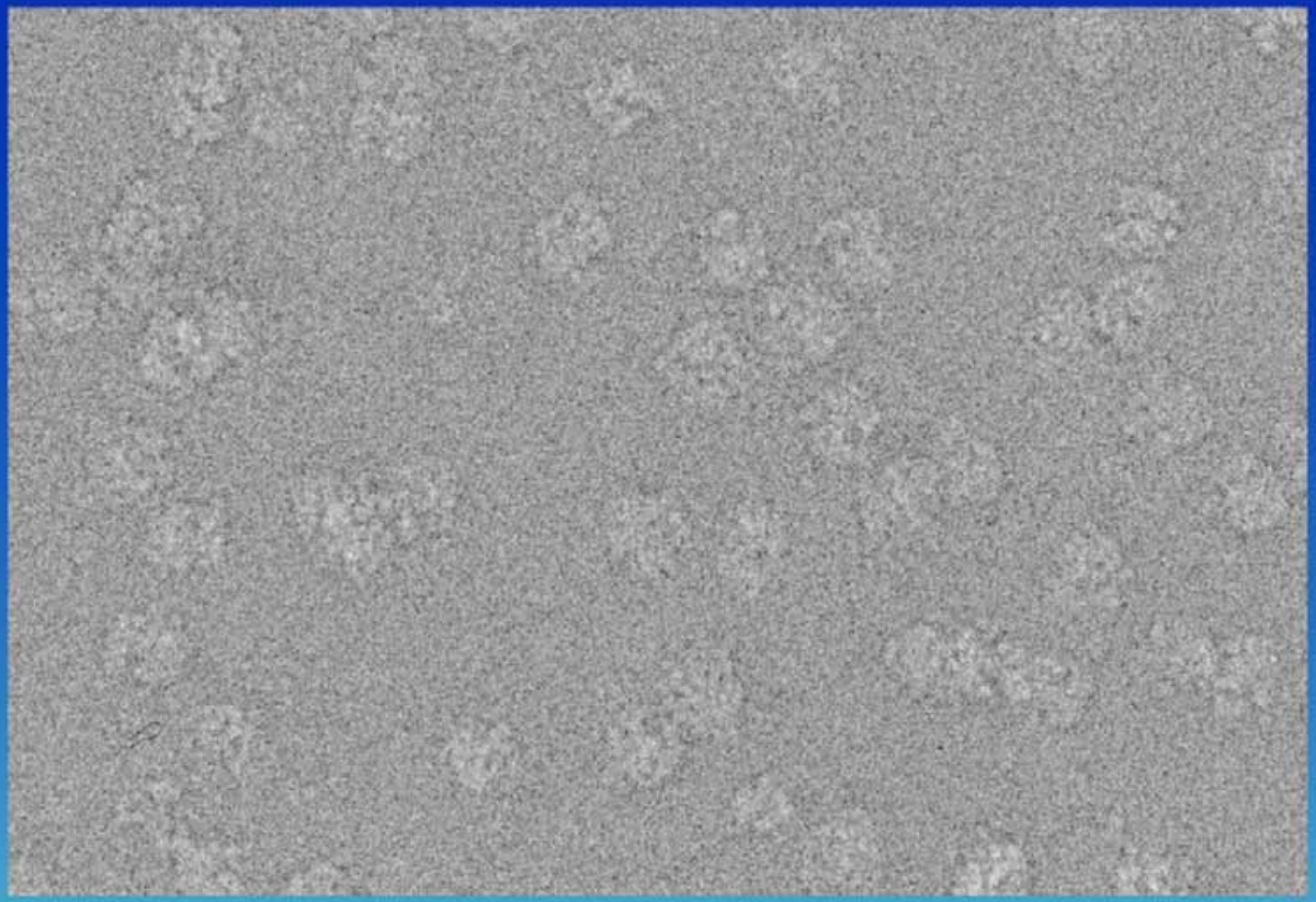


Structural biology - electron cryo microscopy

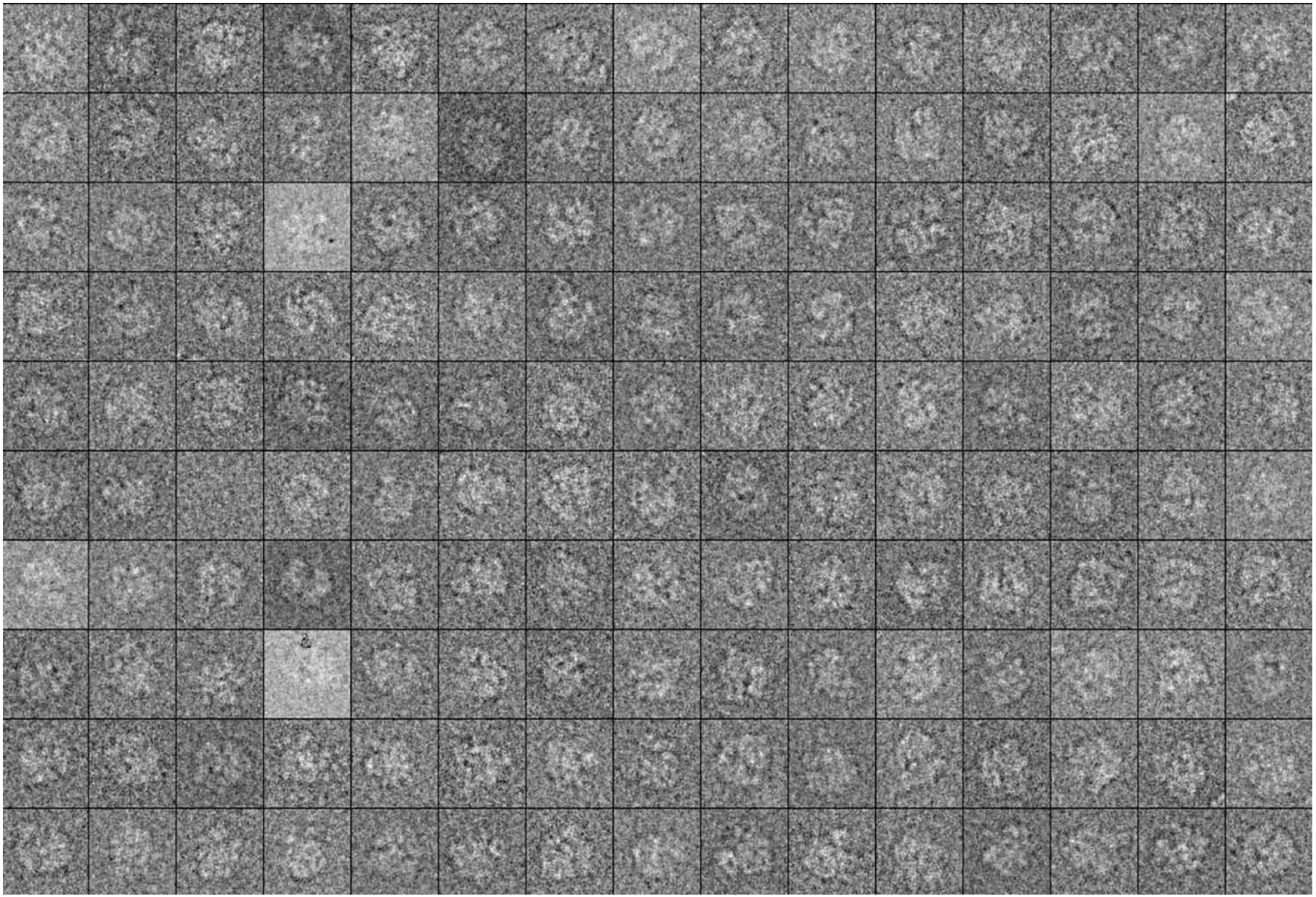


Structural biology - electron cryo microscopy

Cryo-electron micrograph
showing 70S ribosomes from *E. coli*

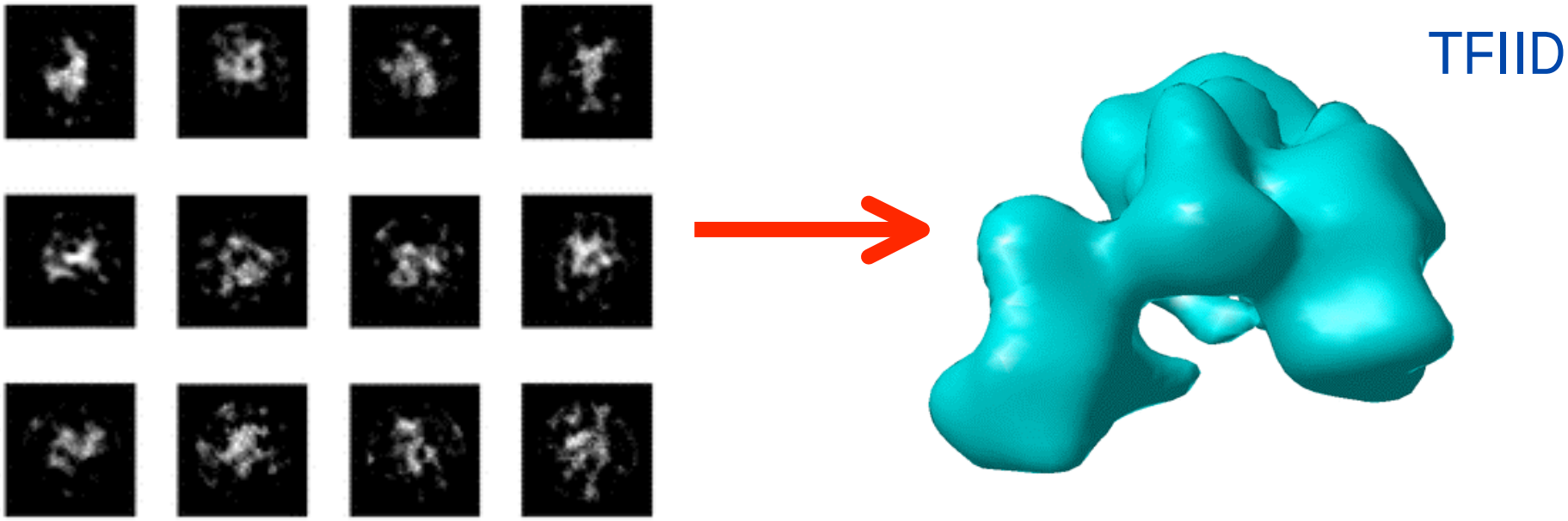


“Particles” selected from a micrograph



Reconstruction of structure of macromolecule

- Determine the 3-D structure (i.e., density map) of macromolecules from 2-D projections.

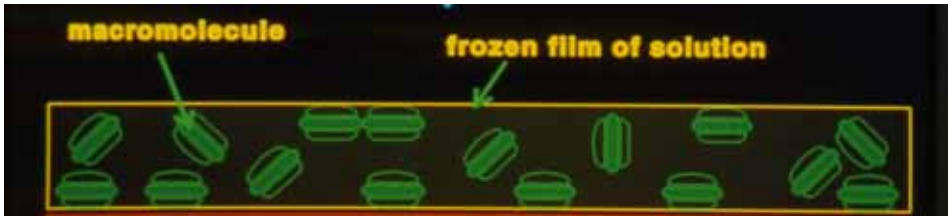


Major steps in cryo EM reconstruction

- Specimen preparation
  - Embed many homogeneous molecules in a thin layer of vitreous ice. (Also called "Single Particle Reconstruction".)
- Produce 2-D images on micrograph.
  - Use low-dose electron microscope.
- Particle selection.
  - The selected particle may not be centered in the box.
- Construct 3-D density map from 2-D projections.

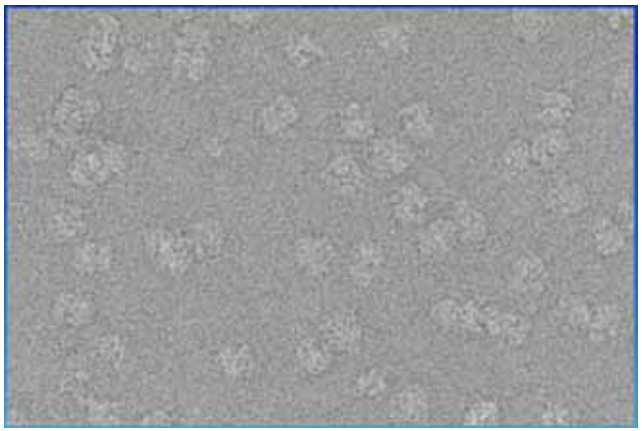
Difficulty of cryo EM

- Orientations of the particles are not known.



  - No control on orientations, which can be random and uneven.
  - They must be determined as part of the solution.
  - Sampling requirement – need to cover as many orientations as possible ⇒ need large number of images.

- Low-dose electron beam ⇒ noisy data
  - A large sample tends to improve the signal-to-noise ratio.
  - Microscope defects and defocus lead to be taken into account.



Mathematical formulation of reconstruction

$$\min_{f,\Phi,\Theta,\Psi,s^h,s^v} \left\| P(\Phi,\Theta,\Psi,s^h,s^v) f - b \right\|_2$$

$f(x,y,z)$	3-D density map to be reconstructed
P	Projection operator
$b^T = [b_1^T, b_2^T, \cdots, b_m^T]$	2-D images from micrographs
$\Phi = [\varphi_1, \varphi_2, \cdots, \varphi_m]$ $\Theta = [\theta_1, \theta_2, \cdots, \theta_m]$ $\Psi = [\psi_1, \psi_2, \cdots, \psi_m]$	Unknown Euler angles $(\varphi_i, \theta_i, \psi_i)$ specifies orientation of i^{th} projection
$s^h = [s_1^h, s_2^h, \cdots, s_m^h]$ $s^v = [s_1^v, s_2^v, \cdots, s_m^v]$	Unknown horizontal and vertical shifts required to center the 2-D images

Difficulty in solving the reconstruction problem

$$\min_{f,\Psi,\Theta,\Phi,s_h,s_v} \left\| P(\Psi,\Theta,\Phi,s^h,s^v) f - b \right\|_2$$

$f(x,y,z)$	3-D density map to be reconstructed
P	Projection operator
$b^T = [b_1^T, b_2^T, \cdots, b_m^T]$	2-D projections from micrographs
$\Psi = [\psi_1, \psi_2, \cdots, \psi_m]$ $\Theta = [\vartheta_1, \vartheta_2, \cdots, \vartheta_m]$ $\Phi = [\phi_1, \phi_2, \cdots, \phi_m]$	Unknown Euler angles $(\psi_i, \vartheta_i, \phi_i)$ specifies orientation of i^{th} projection
$s^h = [s_1^h, s_2^h, \cdots, s_m^h]$ $s^v = [s_1^v, s_2^v, \cdots, s_m^v]$	Unknown horizontal and vertical shifts required to center the 2-D projections

- Nonlinear, and most likely nonconvex.
- Noisy data (e.g., contamination of ice).
- May impose constraints (e.g., horizontal and vertical shifts are usually limited to a few pixels).

Computational Challenges in the Reconstruction

$$\min_{f,\Psi,\Theta,\Phi,s_h,s_v} \left\| P(\Psi,\Theta,\Phi,s^h,s^v) f - b \right\|_2$$

- Large volume of data.
  - Suppose there are m images and each image has n^2 pixels.
    - b is mn^2 ; f is n^3 ; P is mn^2 by n^3.
  - Number of unknowns: $5m+n^3$; amount of data: $n^3 + mn^2$.

- P : mn^2 by n^3 ; f : n^3 ; b : mn^2

$n = 64$; $m = 25{,}000$ $mn^2 = 204{,}800{,}000$; $n^3 = 262{,}144$	$n = 128$; $m = 50{,}000$ $mn^2 = 1{,}638{,}400{,}000$; $n^3 = 2{,}097{,}152$

- To reach atomic resolutions (3Å), millions of images are needed (R. Henderson).

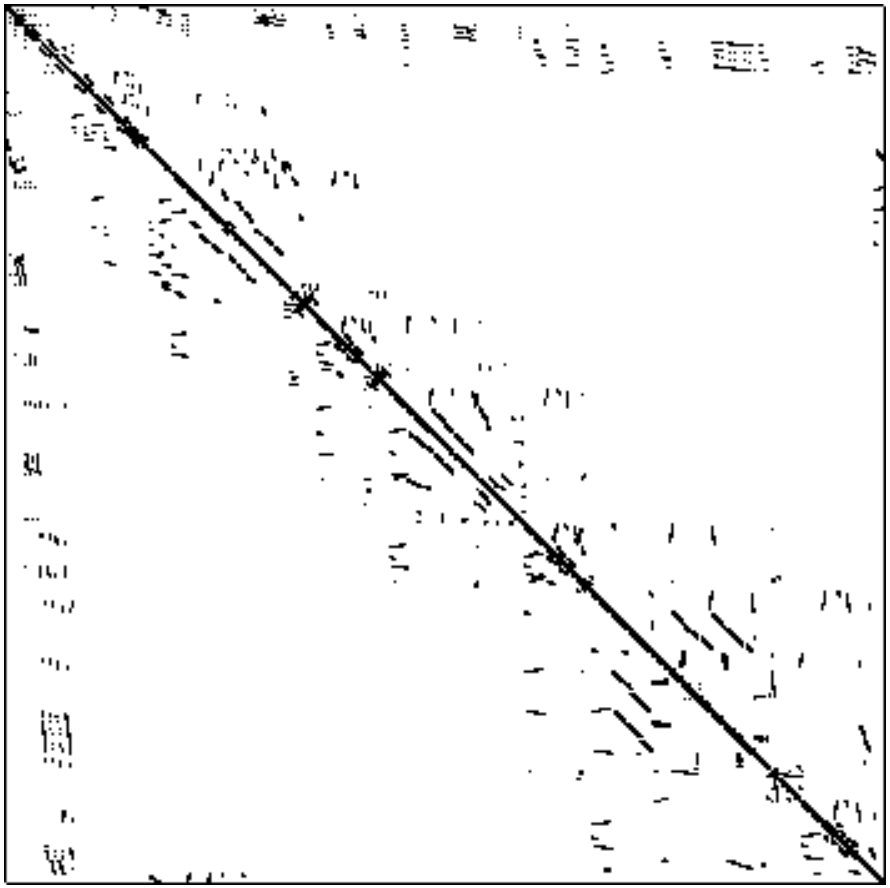
Overall challenges …

- Sparse matrices and structured matrices play a very important role in many large-scale scientific and engineering simulations.
  - Robust and efficient algorithms are important.
  - Accurate solutions are often required.
  - Many unresolved issues remain.
- Scale of the problems leads to many new challenges and open problems.
  - Parallel implementations are absolutely necessary; e.g., representation and accuracy.
  - New computer architectures; e.g., multicore.

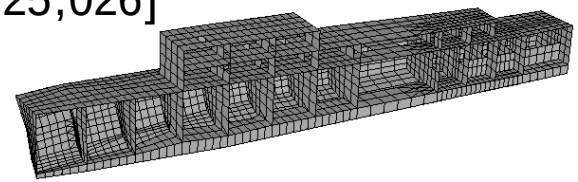
Sparse matrices …

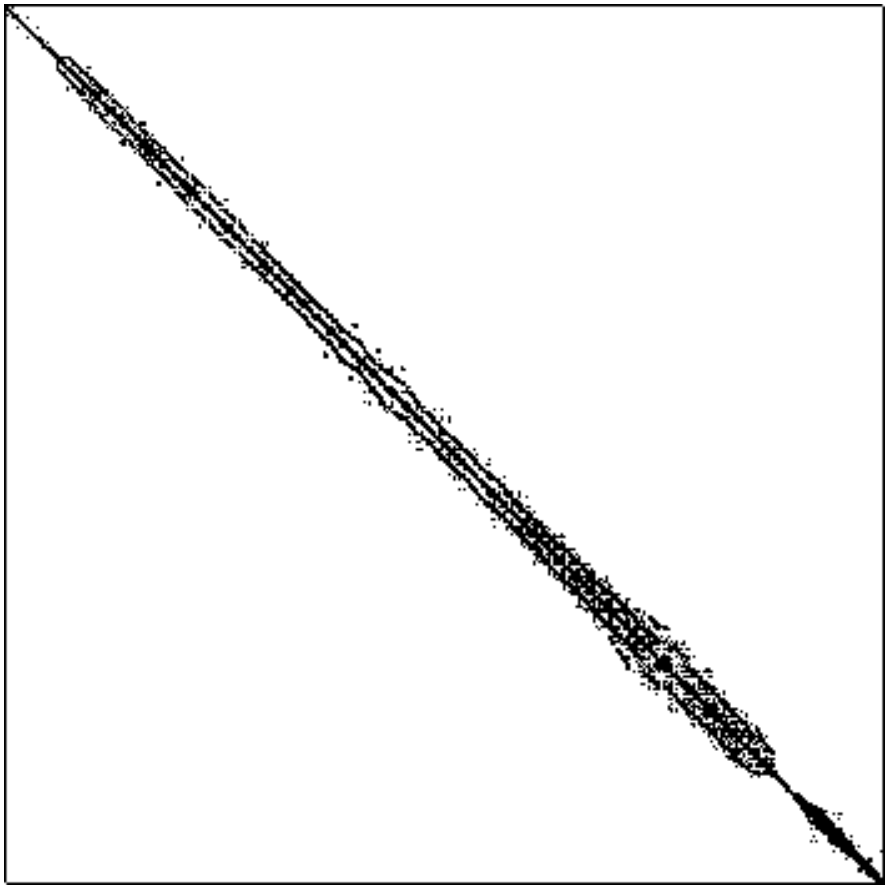
- What do they look like …
  - Will look at the pictures of some of them …

What do sparse matrices look like



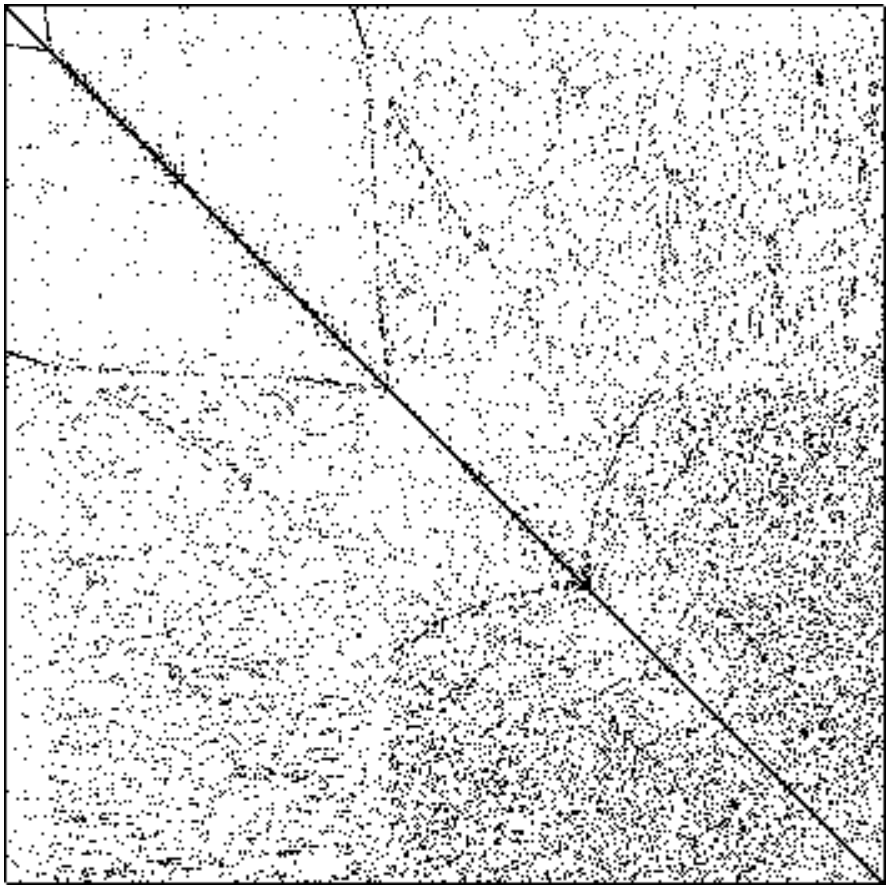
Modeling of a destroyer
[2,680; 25,026]



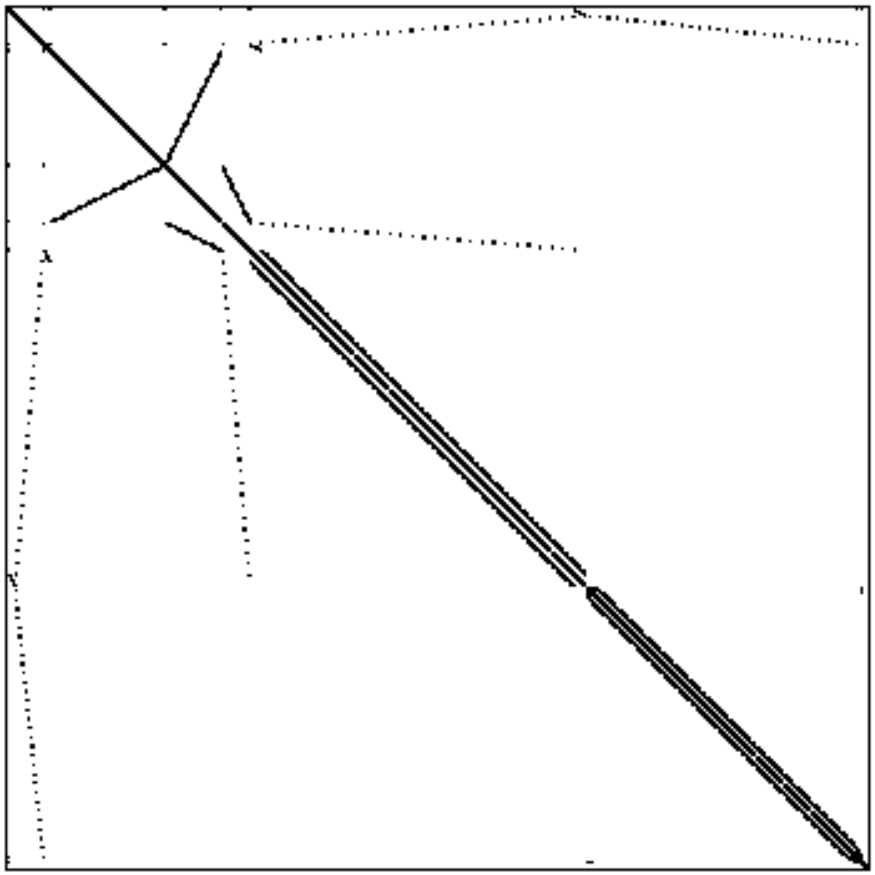


Structural engineering matrix - elemental connectivity for the stiffness matrix of a 1960's design for a supersonic transport (Boeing 2707)
[3,345; 13,047]

What do sparse matrices look like

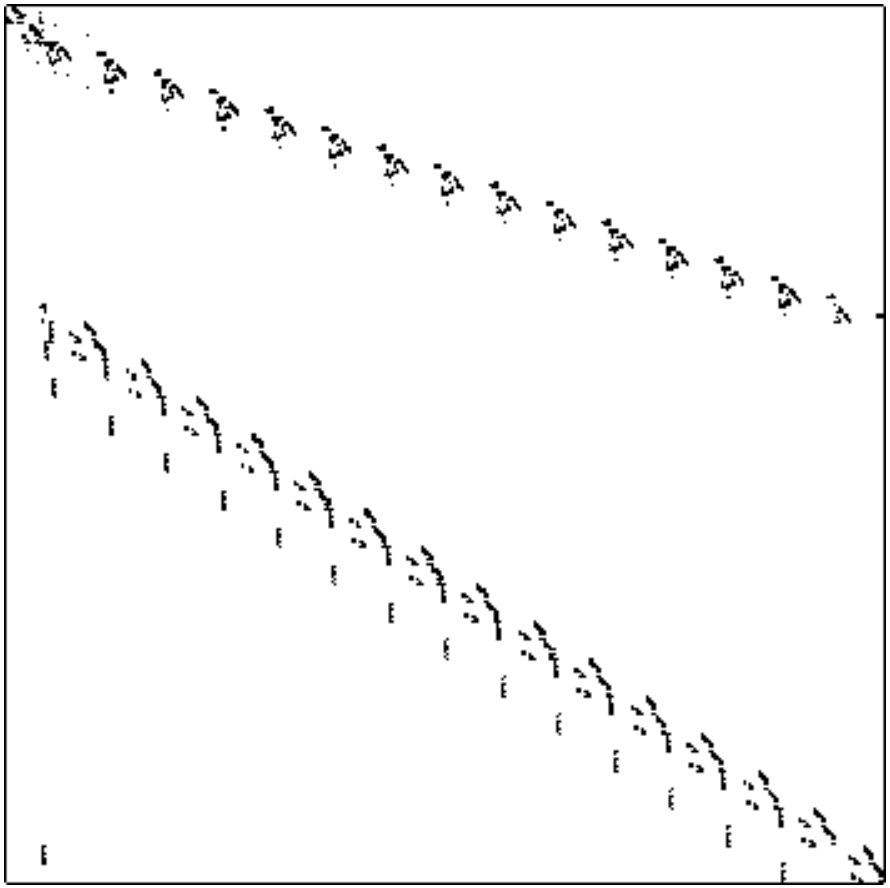


Western US power network - 5300 bus
[5,300; 21,842]

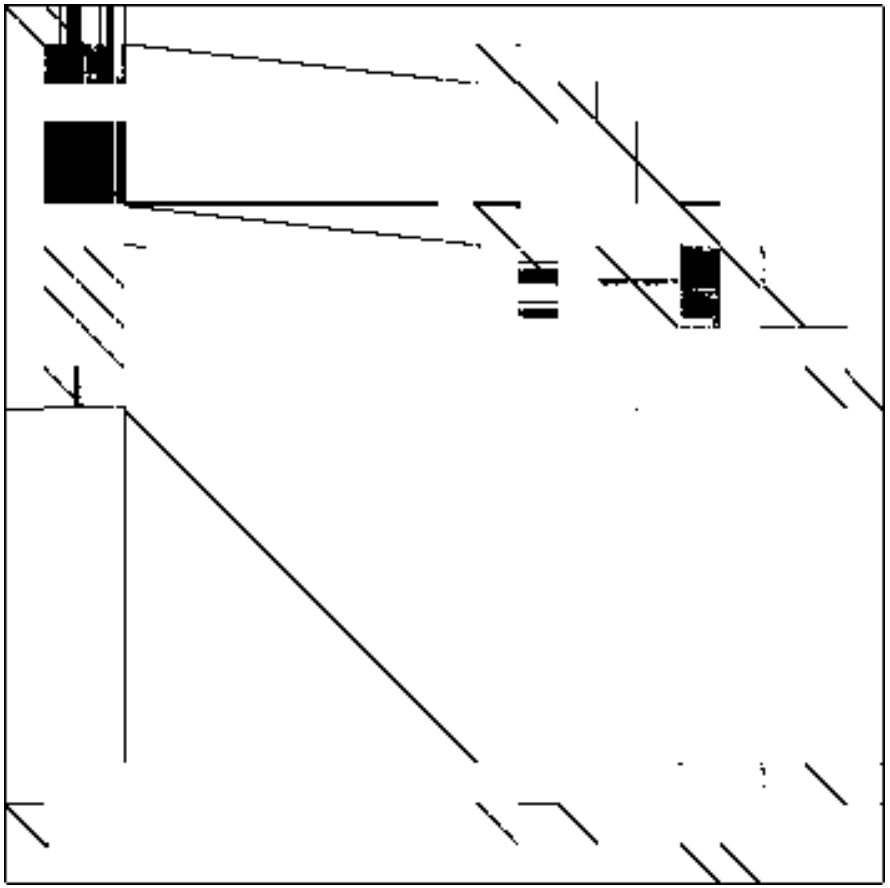


Finite element analysis of a cylindrical shell, graded mesh with 1,666 triangles
[5,357; 106,526]

What do sparse matrices look like

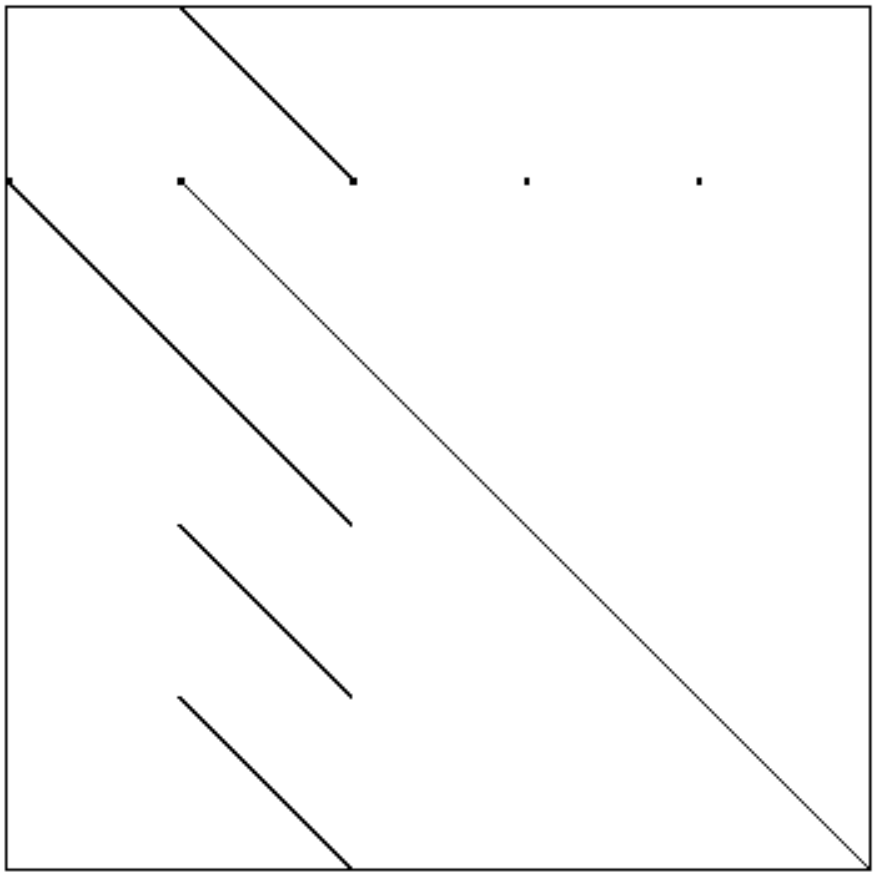


Chemical engineering plant model - 15 stage column section, all rigorous
[2,021; 7,353]

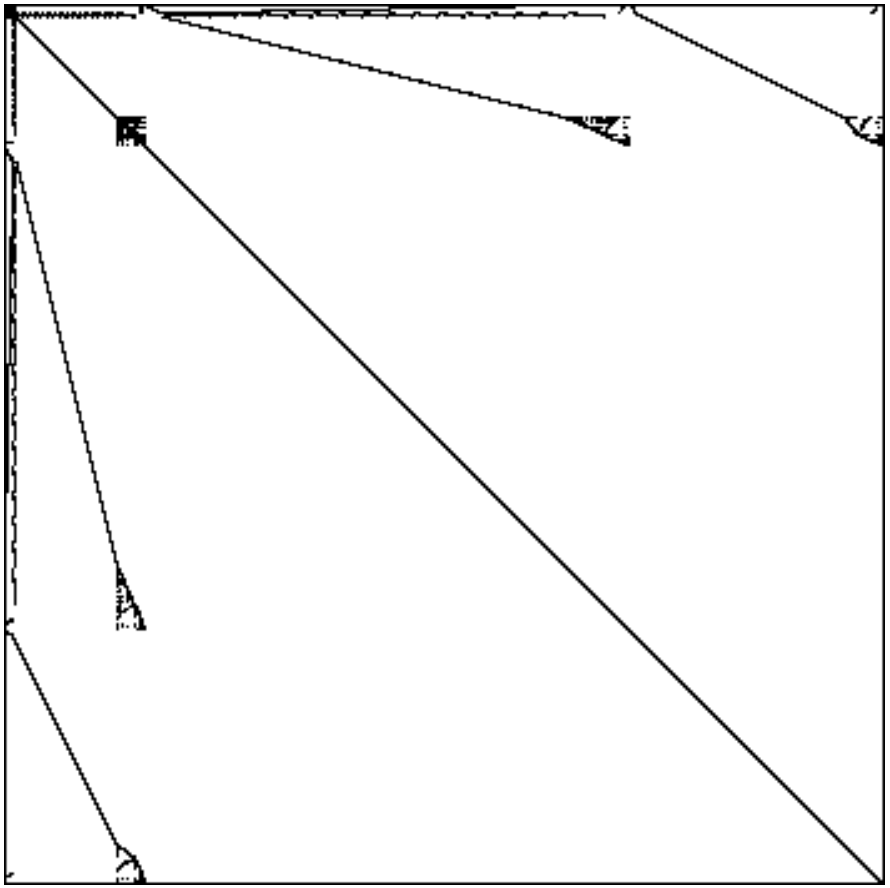


Australian economic model, 1968-1969 data
[2,529; 90,158]

What do sparse matrices look like

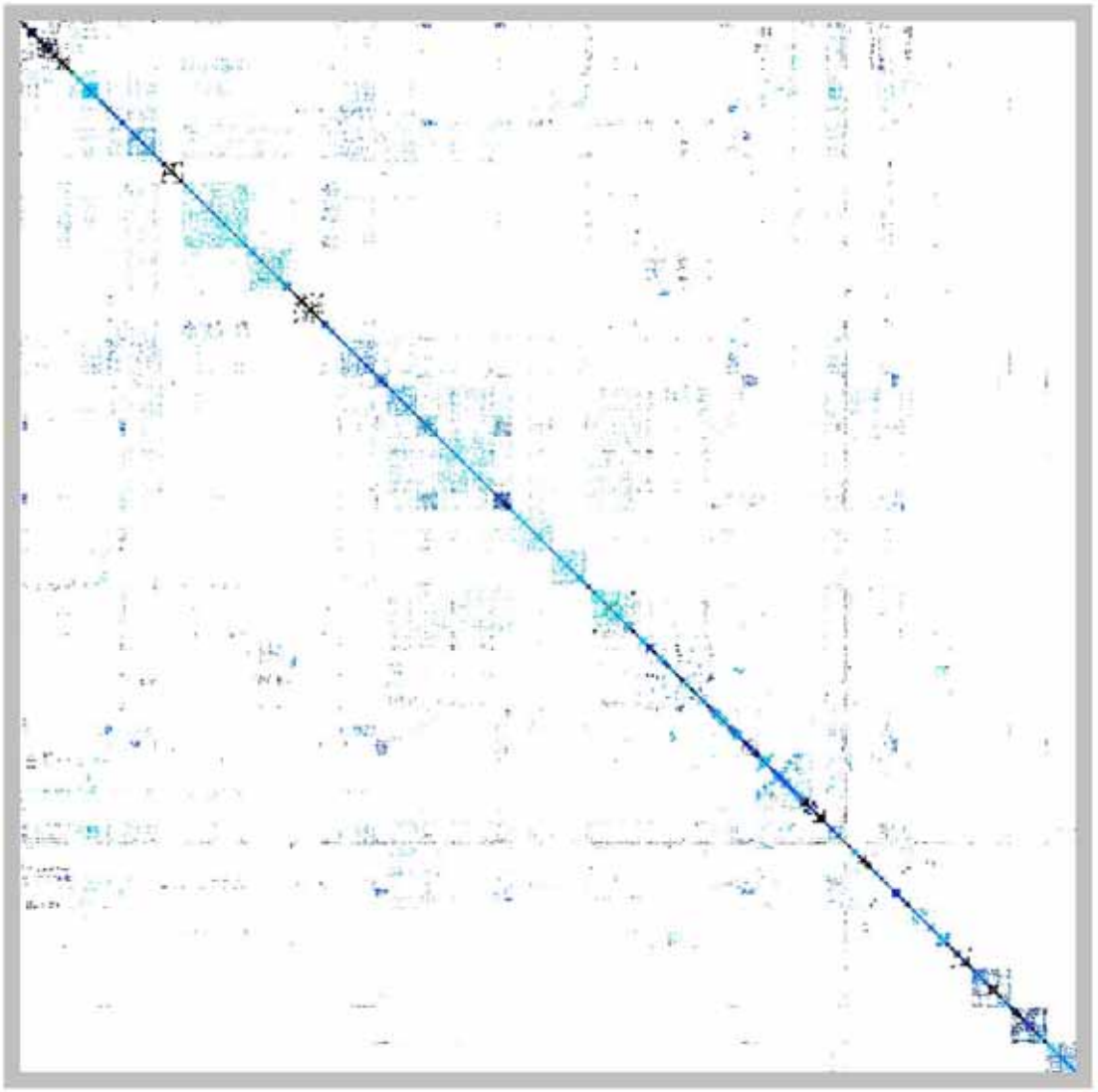


Stability analysis of a model
of an airplane in flight
[4,000; 8,784]



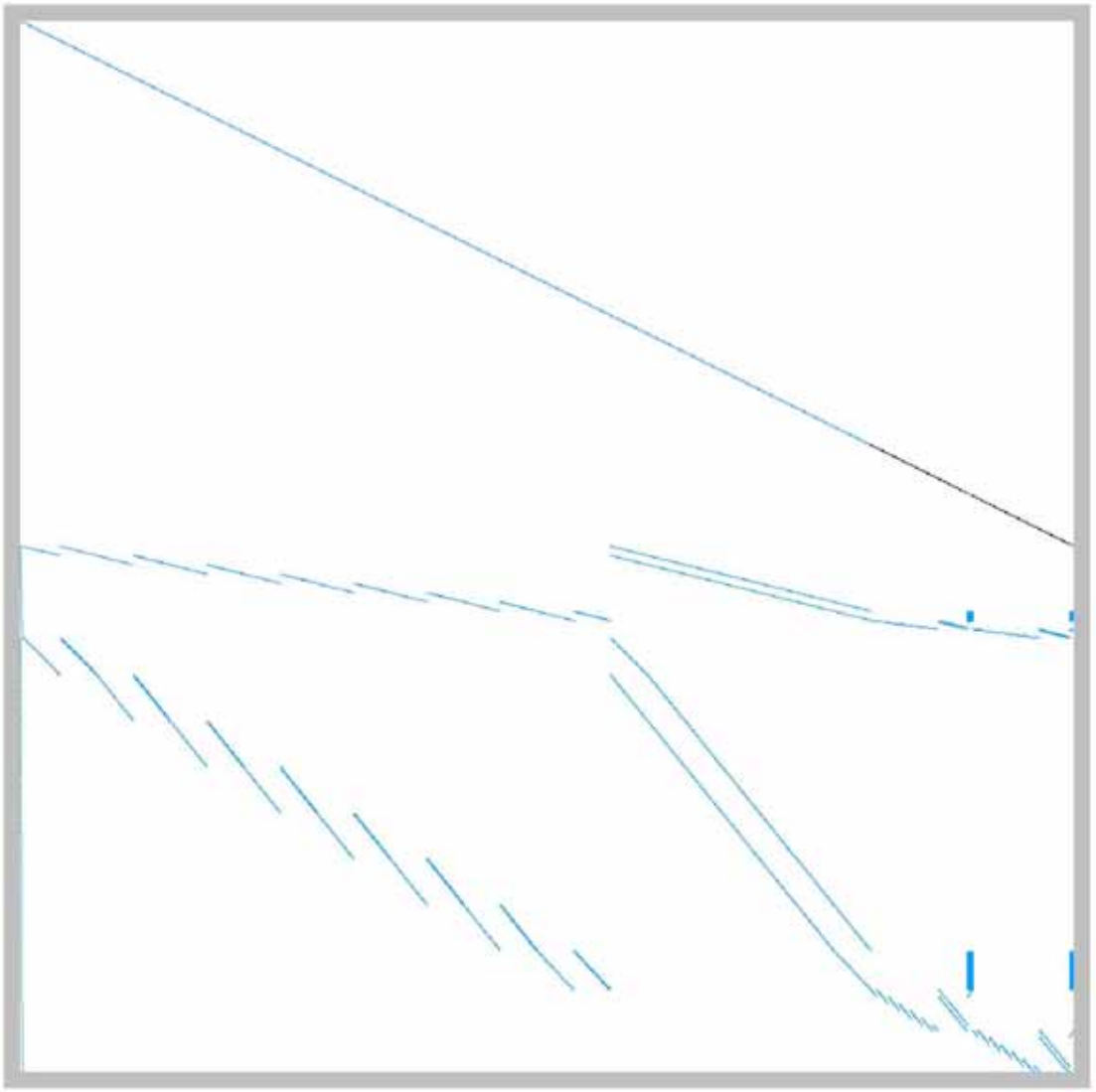
Computer component design - memory circuit
[17,758; 126,150]

What do sparse matrices look like



Linear static analysis of a car body [141,347; 3,740,507]

What do sparse matrices look like



Currents and voltages of a network of resistors
[1,447,360; 5,514,242]

Summary …

- Definition of sparse matrices.

- Sparse matrices and applications.

How do we store sparse vectors and matrices

- Sparse vectors
  - Only store nonzero elements, together with their subscripts.

$$x^T = \begin{bmatrix} 0 & 10 & 0 & 20 & 30 & 0 & 0 & 70 & 0 & 50 \end{bmatrix}$$

  - Store the values of the nonzero elements in a floating-point array, say, val.
  - For each nonzero element, also store the corresponding subscripts in an integer array, say, indx.

$$\text{val}[] = (10, 20, 30, 70, 50)$$

$$\text{indx}[] = (2, 4, 5, 8, 10)$$

  - It is not necessary to sort the contents of val and indx according to the subscripts, but we often do anyway.

How do we store sparse vectors and matrices

- Sparse matrices
  - Stored by rows or columns.
  - Basic idea:
    - Treat each row or column as a sparse vector.
    - Concatenate all floating-point (or integer) arrays into a single array.
    - Need extra information to mark the "boundaries" of each row or column.

$$\begin{bmatrix} 13 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 9 \\ 7 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 11 \\ 3 & 0 & 20 & 0 & 0 \end{bmatrix}$$

How do we store sparse vectors and matrices

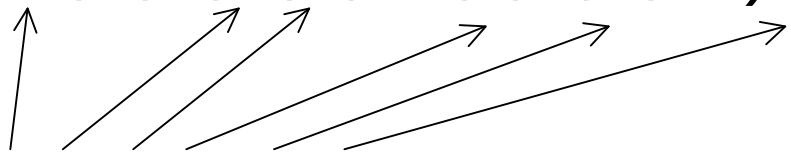
- A column storage scheme (also known as compressed column storage or CSC):

$$\begin{bmatrix} 13 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 9 \\ 7 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 11 \\ 3 & 0 & 20 & 0 & 0 \end{bmatrix}$$

indx[] = (1,3,5 ; 3 ; 1,5 ; 2,3 ; 2,4)

values[] = (13,7,3 ; 2 ; 5,20 ; 1,3 ; 9,11)

ptr[] = (1,4,5,7,9,11)



- ptr has n+1 elements.
  - ptr[n+1] is used to mark the end of the list of pointers.

How do we store sparse vectors and matrices

- Accessing the nonzero elements:

$$\begin{bmatrix} 13 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 9 \\ 7 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 11 \\ 3 & 0 & 20 & 0 & 0 \end{bmatrix}$$

values[] = (13,7,3;2;5,20;1,3;9,11)

indx[] = (1,3,5;3;1,5;2,3;2,4)

ptr[] = (1,4,5,7,9,11)

  - The nonzero elements in column *k* are stored in values[*s*], for *s* = ptr[*k*], ptr[*k*]+1, …, ptr[*k*+1]–1.
  - The corresponding row subscripts are stored in indx[*s*], for *s* = ptr[*k*], ptr[*k*]+1, …, ptr[*k*+1]–1.
  - The number of nonzero elements in column *k* is given by ptr[*k*+1]–ptr[*k*].

How do we store sparse vectors and matrices

- A row storage scheme (also known as compressed row storage or CRS):

$$\begin{bmatrix} 13 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 & 9 \\ 7 & 2 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 11 \\ 3 & 0 & 20 & 0 & 0 \end{bmatrix}$$

values[] = (13,5;1,9;7,2,3;11;3,20)

indx[] = (1,3;4,5;1,2,4;5;1,3)

ptr[] = (1,3,5,8,9,11)

Example: Dot product of dense vectors

- Given:
  - Vectors x and y, both of length n.
- Compute dot product $\alpha = x^T y$.

  $\alpha = 0$

  for $i = 1, 2, \ldots, n$

    $\alpha = \alpha + x_i y_i$

  - Require $2n$ operations (n multiplications and n additions).

- What if one of the vectors is sparse?

Example: Dot product of dense & sparse vectors

- Given:
  - A dense n-vector x.
  - A sparse n-vector y, with $\eta(y)$ nonzero elements.
- Compute dot product $\alpha = x^T y$.
  - A naïve approach is to ignore the zero elements of y and treat y as a dense vector.
    - Require $2n$ operations and $2n$ words.
  - A conceptually more efficient algorithm:

$\alpha = 0$

for $i = 1, 2, \ldots, n$

  if ($y_i \neq 0$) $\alpha = \alpha + x_i y_i$

Example: Dot product of dense & sparse vectors

- Compute dot product $\alpha = x^T y$.

  $\alpha = 0$

  for $i = 1, 2, \ldots, n$

    if ($y_i \neq 0$) then $\alpha = \alpha + x_i y_i$

  - Appear to need $2\eta(y)$ operations **and** n comparisons.
  - False …
    - Because we don't store the entire vector y.
    - Store only the nonzero elements of y.

Example: Dot product of dense & sparse vectors

- Compute dot product $\alpha = x^T y$.
  - Suppose that the nonzero elements of y are stored in an array yval, and their corresponding subscripts in an array yindx.
  - So, yindx and yval are arrays of length $\eta(y)$.

    $\alpha = 0$

    for $k = 1, 2, \ldots, \eta(y)$

      $\alpha = \alpha + x_{\text{yindx}[k]}\, \text{yval}[k]$

  - Require $2\eta(y)$ operations, involving only nonzero operands.
    - No comparisons needed.
  - Require $n+\eta(y)$ floating-point words and $\eta(y)$ integer words.

- What if both x and y are sparse?

Example: Dot product of sparse vectors

- Given:
  - A sparse n-vector x, with $\eta(x)$ nonzero elements
  - A sparse n-vector y, with $\eta(y)$ nonzero elements.
- Compute dot product $\alpha = x^T y$.

$$\alpha = \sum_{i=1}^{n} x_i y_i$$

  - Observations:
    - The only nonzero terms are those for which both x_i and y_i are nonzero.
    - Generally the nonzero elements in x and y do not appear in the same positions; i.e., $x_i \neq 0$ does not imply $y_i \neq 0$, and vice versa.
  - Seem to suggest that subscript matching is inevitable.

Example: Dot product of sparse vectors

- Compute dot product $\alpha = x^T y$.
  - Assume that only the nonzero elements of *x* and *y* are stored in xval and yval, respectively, with their corresponding subscripts in xindx and yindx.

    $\alpha = 0$

    for $s = 1, 2, \ldots, \eta(x)$

      for $t = 1, 2, \ldots, \eta(y)$

        if (xindx[s] = yindx[t]) then $\alpha = \alpha$ + xval[s] yval[t]

  - Number of operations $\leq 2 \min(\eta(x), \eta(y))$.
  - Appear to require $\eta(x)\eta(y)$ comparisons.
  - Require $\eta(x)+\eta(y)$ floating-point words and $\eta(x)+\eta(y)$ integer words.

Example: Dot product of sparse vectors

- A more efficient way to compute dot product $\alpha = x^T y$.
  - Assume that a temporary array temp of length n is available.
  - Assume that $\eta(x) \geq \eta(y)$.

for i = 1, 2, …, n
  temp[i] = 0
for s = 1, 2, …, $\eta(x)$
  temp[xindx[s]] = xval[s]
α = 0
for t = 1, 2, …, $\eta(y)$
  $\alpha = \alpha$ + yval[t] temp[yindx[t]]

Example: Dot product of sparse vectors

- A more efficient way to compute dot product $\alpha = x^T y$.

```
for i = 1, 2, …, n
   temp[i] = 0
for s = 1, 2, …, η(x)
   temp[xindx[s]] = xval[s]
α = 0
for t = 1, 2, …, η(y)
   α = α + yval[t] temp[yindx[t]]
```

  - Number of operations = 2 $\eta(y)$ = 2 min($\eta(x), \eta(y)$).
  - No comparisons needed.
  - Require $\eta(x)+\eta(y)+n$ floating-point words and $\eta(x)+\eta(y)$ integer words.

Exercises

- How to compute the product of the following pairs of objects?
  - Dense matrix and sparse vector.
  - Sparse matrix and dense vector.
  - Sparse matrix and sparse vector.

  - Note: The algorithms depend on how the matrices are stored.

- A more challenging exercises … How to compute the product of two sparse matrices?

Summary …

- Storage schemes for sparse vectors and sparse matrices.

- Simple operations on sparse vectors.
  - Implementations.
  - Data structures.

Solution of sparse triangular linear systems

- Let T be an n-by-n sparse lower triangular matrix.
- Let b be an n-vector.
- Find x so that $Tx = b$.
- The algorithm depends on how the nonzero elements of T are stored.

Solution of sparse triangular linear systems

- Expressing the system in a row-wise fashion …

$$\begin{bmatrix} t_{11} & & & \\ t_{21} & t_{22} & & \\ \vdots & \vdots & \ddots & \\ t_{n1} & t_{n2} & \cdots & t_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$t_{ii}x_i = b_i - \sum_{k=1}^{i-1} t_{ik}x_k, \quad i = 1,2,\ldots,n$$

inner product of a sparse vector and a dense vector

Solution of sparse triangular linear systems

- Assume that the nonzero elements of T are stored by rows using the arrays (rowptr, colindx, values).
  - Assume that the nonzero elements within each row are stored in increasing order of the column subscripts.

- Row-wise algorithm:

for $i = 1, 2, \ldots, n$
  rowbeg = rowptr[i]
  rowend = rowptr[i+1] – 1
  for s = rowbeg, rowbeg+1, …, rowend–1
    $b_i = b_i$ – values[s] $x_{\text{colindx}[s]}$
  $x_i = b_i$ / values[rowend]

Solution of sparse triangular linear systems

- Expressing the system in a column-wise fashion …

$$\begin{bmatrix} t_{11} & & & \\ t_{21} & t_{22} & & \\ \vdots & \vdots & \ddots & \\ t_{n1} & t_{n2} & \cdots & t_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- If we use T_{*i} to denote the i^{th} column of T, then

$$\sum_{i=1}^{n} x_i T_{*i} = b$$

Solution of sparse triangular linear systems

- Assume that the nonzero elements of T are stored by columns using the arrays (colptr, rowindx, values).
  - Assume that the nonzero elements within each column are stored in increasing order of the row subscripts.

- Column-wise algorithm:

for j = 1, 2, …, n
  colbeg = colptr[j]
  colend = colptr[j+1] – 1
  $x_j = b_j$ / values[colbeg]
  for s = colbeg+1, colbeg+2, …, colend
    $b_{\text{rowindx}[s]} = b_{\text{rowindx}[s]} - \text{values}[s]\, x_j$

Solution of sparse triangular linear systems

- It should be easy to show that the number of operations required to solve a sparse triangular linear system is proportional to the number of nonzero elements in T.
  - We sometimes use $|T|$ to denote the number of nonzero elements in T.
  - We will say that the number of operations is $O(|T|)$.
    - $O(f(n))$ describes the asymptotic behavior.
    - $g(n) = O(f(n))$ if there are constants $c > 0$ and $n_0 > 0$, fixed for g and independent of n, such that $0 \leq g(n) \leq c\, f(n)$ for all $n \geq n_0$.

Solution of sparse triangular linear systems

- Similar algorithms when T is upper triangular.

- Exercise: How to compute the solution of a sparse triangular linear system, for which the right-hand side is also sparse?

Solution of "general" sparse linear systems

- Let A be an n-by-n sparse matrix.
- Let b be an n-vector.

- Find x so that $Ax = b$.

- A word on "sparsity" …
  - The sparsity of a vector/matrix often refers to the positions of the nonzero elements in the vector/matrix.
  - Also refer to as the structure or sparsity structure.

Solution of "general" sparse linear systems

- Let A be an n-by-n sparse matrix.
- Let b be an n-vector.

- Find x so that $Ax = b$.

- Two popular classes of methods.
  - Iterative methods
  - Direct methods

Iterative solution of sparse linear systems

- Based on the construction of a sequence of approximations to the solution.
  - $\{ x_0 , x_1 , x_2 , \ldots \}$
  - x_0 is an initial guess.
- Many algorithms available for generating the approximations:
  - Basic methods:
    - Jacobi, Gauss-Seidel, successive overrelaxation
  - Projection methods:
    - Steepest descent, minimal residual
  - Krylov subspace methods:
    - Arnoldi's, generalized minimal residual, conjugate gradient, conjugate residual, biconjugate gradient, quasi-minimal residual

Iterative solution of sparse linear systems

- Positives:
  - Relative easy to implement.
  - Minimal storage requirement.

- Negatives:
  - Convergence is not guaranteed.
  - Convegence rate may be slow.
  - Both depends on the spectral radius of the “iteration matrix”.

Iterative solution of sparse linear systems

- Convergence rate:
  - Find nonsingular matrices P and Q.
  - Consider the equivalent linear system $(PAQ)(Q^{-1}x) = (Pb)$.
  - The goal is to reduce the spectral radius of PAQ.
  - P and Q are called the left and right "preconditioners", respectively.

- Preconditioning is a research area of its own.
  - Many classes of methods are available.
    - Polynomial preconditioning.
    - Incomplete factorization.
    - Approximate inverses.
    - Support graphs.

Direct solution of sparse linear systems

- Sparse versions of Gaussian elimination for dense matrices.
  - Transform the given linear system into triangular linear systems that are much easier to solve.
  - Gaussian elimination can be described as a factorization of the given matrix into a product of a lower triangular matrix and an upper triangular matrix:

    $A \rightarrow L\ U$

    L is lower triangular and U is upper triangular.

- If $A = LU$, then the given linear system can be written as
  $A\ x = LU\ x = b$.
- Substituting $y = U\ x$, we have $L\ y = b$.

Direct solution of sparse linear systems

- So, the original linear system has been transformed into two triangular linear systems:

  $L\ y = b$

  $U\ x = y$

- The solution x is therefore obtained after the solution of two triangular linear systems.

- Caveat: Ignore pivoting for stability.

Direct solution of sparse linear systems

- Positives:
  - Finite termination after a finite number of operations.
  - Gaussian elimination is known to be backward stable.
    - Assuming that pivoting for stability is not needed.

- Negatives:
  - Sparsity issues.
  - Algorithms tend to be complicated.
  - Implementations tend to be hard.

Other sparse matrix problems …

- Although we are looking at square linear systems, there are others, such as overdetermined and underdetermined linear systems.

- For non-square problems, other approaches may be more appropriate.
  - e.g., orthogonal decomposition, singular value decomposition.

Important assumption …

- No-cancellation rule.
  - Let x and y be two numbers.
  - We assume that the sum $x+y$ or the difference $x–y$ is always nonzero, regardless of the values of x and y.

- Why? How realistic is such an assumption?

What are the sparsity issues

- Consider a small example.

$$A = \begin{bmatrix} 5 & -1 & -1 & -1 & -1 \\ 1 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

What are the sparsity issues

- Applying Gaussian elimination to A produces the triangular factorization $A = L_A U_A$.

$$A = \begin{bmatrix} 5 & -1 & -1 & -1 & -1 \\ 1 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$L_A = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 \\ 0.2000 & 1.0000 & 0 & 0 & 0 \\ 0.2000 & 0.0476 & 1.0000 & 0 & 0 \\ 0.2000 & 0.0476 & 0.0597 & 1.0000 & 0 \\ 0.2000 & 0.0476 & 0.0597 & 0.0822 & 1.0000 \end{bmatrix}$$

$$U_A = \begin{bmatrix} 5.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000 \\ 0 & 4.2000 & 0.2000 & 0.2000 & 0.2000 \\ 0 & 0 & 3.1905 & 0.1905 & 0.1905 \\ 0 & 0 & 0 & 2.1791 & 0.1791 \\ 0 & 0 & 0 & 0 & 1.1644 \end{bmatrix}$$

What are the sparsity issues

- Performing Gaussian elimination on a sparse matrix can destroy some of the zero elements.

- Now consider another small example.

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & 4 & 1 \\ -1 & -1 & -1 & -1 & 5 \end{bmatrix}$$

What are the sparsity issues

- Applying Gaussian elimination to B produces the triangular factorization $B = L_B U_B$.

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & 4 & 1 \\ -1 & -1 & -1 & -1 & 5 \end{bmatrix}$$

$$L_B = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 0 & 1.0000 & 0 \\ -1.0000 & -0.5000 & -0.3333 & -0.2500 & 1.0000 \end{bmatrix}$$

$$U_B = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 1.0000 \\ 0 & 2.0000 & 0 & 0 & 1.0000 \\ 0 & 0 & 3.0000 & 0 & 1.0000 \\ 0 & 0 & 0 & 4.0000 & 1.0000 \\ 0 & 0 & 0 & 0 & 7.0833 \end{bmatrix}$$

What are the sparsity issues

- Let's look at *A* and *B* again.

$$A = \begin{bmatrix} 5 & -1 & -1 & -1 & -1 \\ 1 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & 4 & 1 \\ -1 & -1 & -1 & -1 & 5 \end{bmatrix}$$

- What is the difference between *A* and *B*?
  - *B* can be obtained from *A* by reversing the order of the rows and columns!

$$B = PAP^{\mathsf{T}} \qquad P = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

What are the sparsity issues

- The order in which Gaussian elimination is applied can influence the number of nonzero elements in the factors.

$$A = \begin{bmatrix} 5 & -1 & -1 & -1 & -1 \\ 1 & 4 & 0 & 0 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 & 1 \\ 0 & 0 & 0 & 4 & 1 \\ -1 & -1 & -1 & -1 & 5 \end{bmatrix}$$

$$L_A = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 \\ 0.2000 & 1.0000 & 0 & 0 & 0 \\ 0.2000 & 0.0476 & 1.0000 & 0 & 0 \\ 0.2000 & 0.0476 & 0.0597 & 1.0000 & 0 \\ 0.2000 & 0.0476 & 0.0597 & 0.0822 & 1.0000 \end{bmatrix}$$

$$L_B = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 0 \\ 0 & 1.0000 & 0 & 0 & 0 \\ 0 & 0 & 1.0000 & 0 & 0 \\ 0 & 0 & 0 & 1.0000 & 0 \\ -1.0000 & -0.5000 & -0.3333 & -0.2500 & 1.0000 \end{bmatrix}$$

$$U_A = \begin{bmatrix} 5.0000 & -1.0000 & -1.0000 & -1.0000 & -1.0000 \\ 0 & 4.2000 & 0.2000 & 0.2000 & 0.2000 \\ 0 & 0 & 3.1905 & 0.1905 & 0.1905 \\ 0 & 0 & 0 & 2.1791 & 0.1791 \\ 0 & 0 & 0 & 0 & 1.1644 \end{bmatrix}$$

$$U_B = \begin{bmatrix} 1.0000 & 0 & 0 & 0 & 1.0000 \\ 0 & 2.0000 & 0 & 0 & 1.0000 \\ 0 & 0 & 3.0000 & 0 & 1.0000 \\ 0 & 0 & 0 & 4.0000 & 1.0000 \\ 0 & 0 & 0 & 0 & 7.0833 \end{bmatrix}$$

Sparse matrix factorizations

- Observations:
  - Performing Gaussian elimination on a sparse matrix can destroy some of the zero elements.
  - The order in which Gaussian elimination is applied can influence the number of nonzero elements in the factors.

Sparse matrix factorizations

- Sparse Gaussian elimination is all about managing "sparsity":
  - Finding ways to reduce the number of zero elements that get turned into nonzero during Gaussian elimination.
    - Combinatorial algorithms.
  - Discovering the sparsity and constructing data structures to store as few zero elements as possible.
    - Computer science.
  - Operating on as few zero elements as possible.
    - Numerical algorithms.
  - Analyzing and understanding the complexity.
    - Computer science.
  - Efficient implementation.
    - Computer architecture.

Direct solution of sparse linear systems

- Consider the solution of the linear system $Ax = b$.
- Once a triangular factorization of A has been computed using Gaussian elimination, the solution to $Ax = b$ can be obtained by solving two sparse triangular systems, which we have looked at already.

- So, we will focus on the triangular factorization of a sparse matrix.
  - We will first look at the case when A is symmetric and positive definite.
  - Then we will consider the case when A is a general sparse matrix.

Aside …

- Let's first look at dense Gaussian elimination more closely.

```
for _____
    for _____
        for _____
            a_ij ← a_ij – a_ik a_kj
```

- Different placements of *i*, *j*, and *k* give different algorithms for performing Gaussian elimination.
  - Total of 6 variants.

Aside … Dense Gaussian elimination

Right-looking formulations

kij version :

for $k = 1, 2, \cdots, n$

  for $i = k+1, k+2, \cdots, n$

    $a_{ik} \leftarrow a_{ik} \,/\, a_{kk}$

  for $i = k+1, k+2, \cdots, n$

    for $j = k+1, k+2, \cdots, n$

      $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$

kji version :

for $k = 1, 2, \cdots, n$

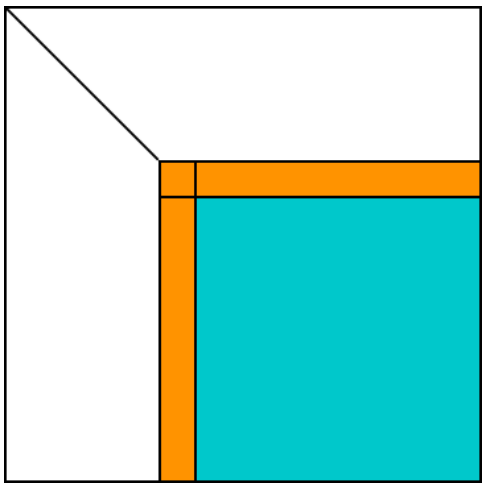
  for $i = k+1, k+2, \cdots, n$

    $a_{ik} \leftarrow a_{ik} \,/\, a_{kk}$

  for $j = k+1, k+2, \cdots, n$

    for $i = k+1, k+2, \cdots, n$

      $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$



Aside … Dense Gaussian elimination

Left-looking formulations

jki version :

for $j = 1, 2, \cdots, n$

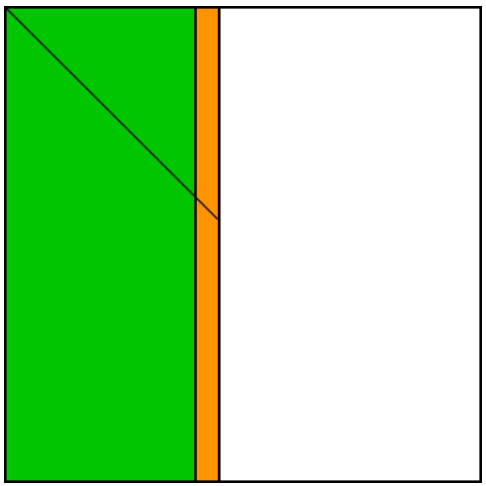
  for $k = 1, 2, \cdots, j-1$

    for $i = k+1, k+2, \cdots, n$

      $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$

  for $i = j+1, j+2, \cdots, n$

    $a_{ij} = a_{ij} \; / \; a_{jj}$



jik version :

for $j = 1, 2, \cdots, n$

  for $i = 1, 2, \cdots, j$

    for $k = 1, 2, \cdots, i-1$

      $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$

  for $i = j+1, j+2, \cdots, n$

    for $k = 1, j-1$

      $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$

    $a_{ij} = a_{ij} \; / \; a_{jj}$

Aside … Dense Gaussian elimination

Top-down (or up-looking) formulation

ikj version :

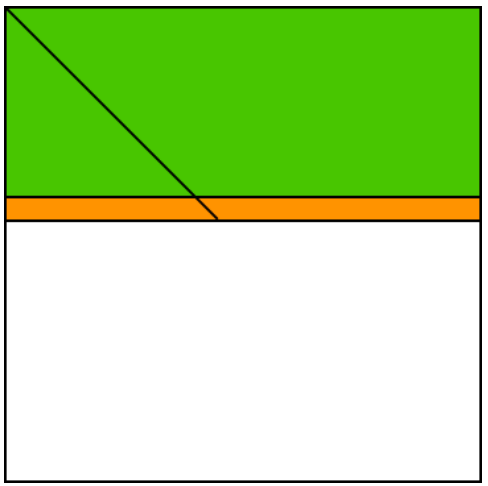
for $i = 1,2,\cdots,n$

  for $k = 1,2,\cdots,i-1$

    $a_{ik} = a_{ik} \,/\, a_{kk}$

    for $j = k+1,k+2,\cdots,n$

      $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$



ijk version :

for $i = 2,3,\cdots,n$

  for $j = 2,3,\cdots,i$

    $a_{i,j-1} = a_{i,j-1} \,/\, a_{j-1,j-1}$

    for $k = 1,2,\cdots,j-1$

      $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$

  for $j = i+1,i+2,\cdots,n$

    for $k = 1,2,\cdots,i-1$

      $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$

Aside … Dense Gaussian elimination

- All 6 variants compute the same factorization (in theory).
- But they access the elements of the matrix differently.
  - Some are more efficient than the others.

- We have the same 6 variants for the sparse case.
  - Some are more natural than the others.
  - Some are easier to implement than the others.
  - Some are more efficient than the others.
  - Some can exploit sparsity easier than the others.

Sparse symmetric positive definite matrices

- A : n by n sparse symmetric positive definite (SPD) matrix.

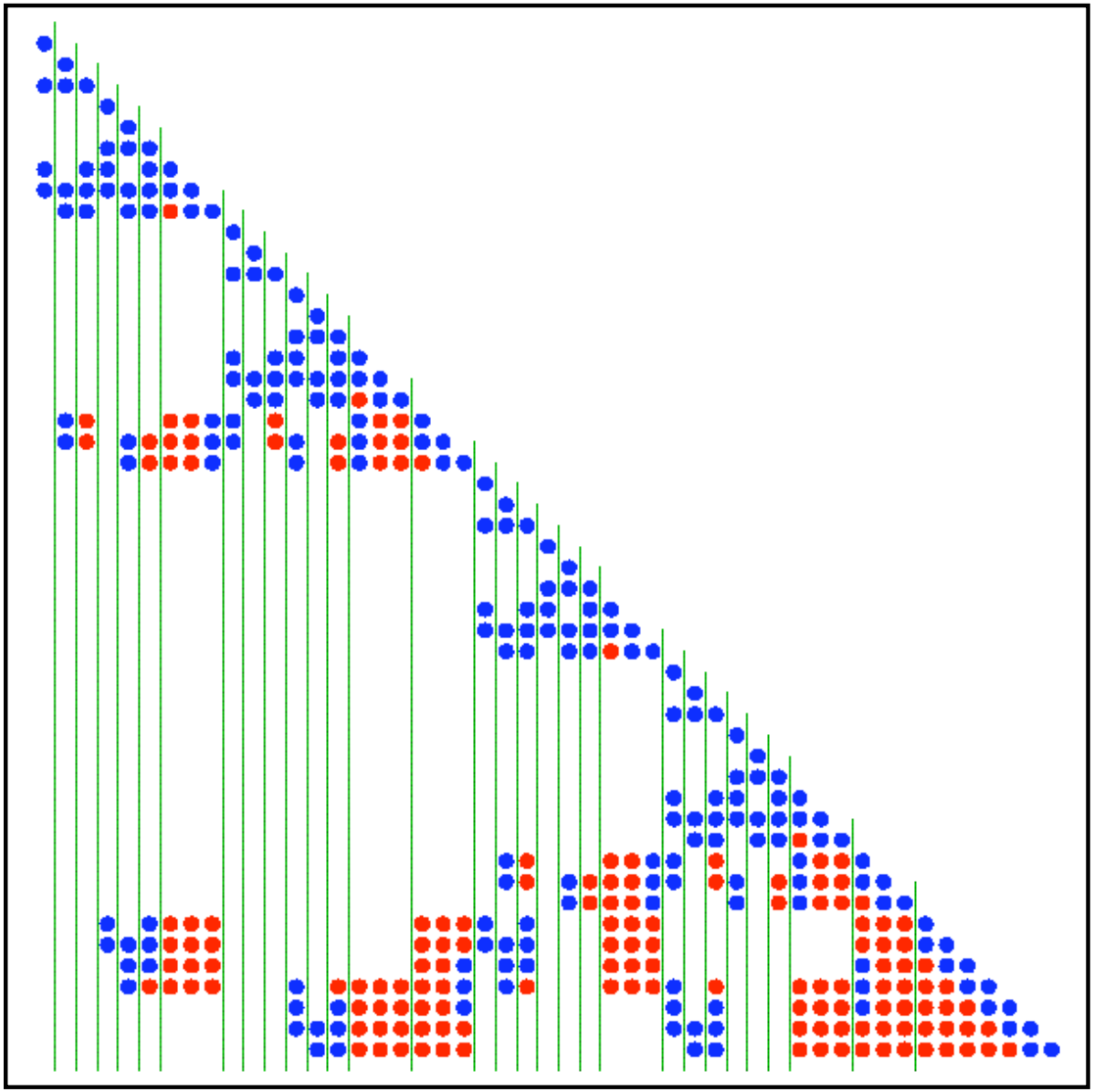
- Cholesky factorization of A:

  $$A = LL^T$$

- L : lower triangular with positive diagonal elements.

- Can compare sparsity of $L+L^T$ with that of A:
  - As we have seen from earlier examples, some zero elements in A will become nonzero in $L+L^T$.
  - That is, in general $|L+L^T| \geq |A|$.
  - Nonzero elements in $L+L^T$ that are zero in A are referred to as fill elements.

Fill in sparse Cholesky factorization



An example based on a 9-point stencil on a regular mesh.

Only the lower triangular part of A/L is shown.

Blue dots are nonzero elements in A.

Red dots are zero elements in A that turn into nonzero during Cholesky factorization. They are fill elements.

Computing sparse Cholesky factorization

- Organize/manage computation to
  - Reduce the amount of fill.
  - Discover fill.
  - Exploit sparsity in numerical factorization.
  - Achieve high performance.

Summary …

- Sparse triangular solutions.

- Introduction to solution of sparse linear systems using Gaussian elimination.
  - Variants of Gaussian elimination.
  - Issue of fill.

Understanding sparsity of Cholesky factors

- Fact: Cholesky factorization of a symmetric positive definite matrix is numerical stable without pivoting.

- It implies that one can study the process of Cholesky factorization, and in particular the structure of Cholesky factor, without considering the actual numerical values of the nonzero elements.

Understanding sparsity of Cholesky factors

- This means that it is possible to determine the structure of the Cholesky factor without actually computing it.
  - For example, using the positions of nonzero elements in a given matrix *A*, one can simulate the Cholesky factorization process and determine where the nonzero elements will be in the Cholesky factor.
  - This is a naïve approach; it requires as many operations as the numerical factorization.

Understanding sparsity of Cholesky factors

- Computing the sparsity structure of the Cholesky factorization is called “symbolic factorization”.

- Why is symbolic factorization a good idea?
  - Permit an efficient data structure to be set up to store the nonzero elements of *L* before computing them.
  - Reduce the amount of data structure manipulation during numerical factorization.
    - Most of the operations during numerical factorization are then floating-point operations.
  - Help design of efficient numerical factorization.

Analyzing sparsity of Cholesky factor

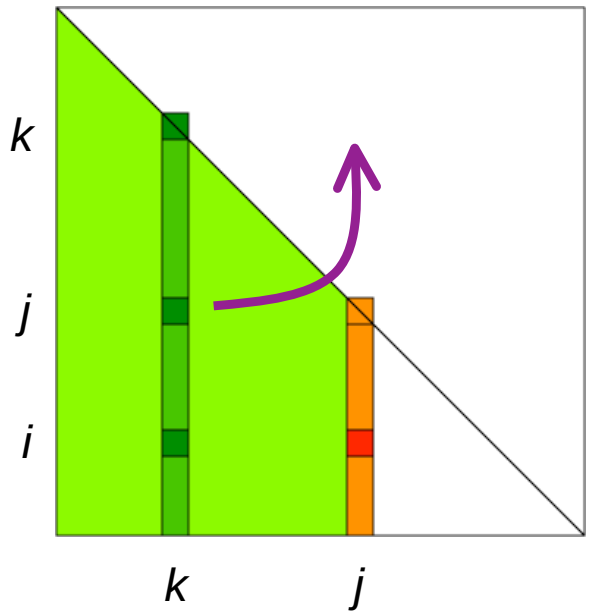
- Lemma [Parter ('61)]

  Let $i > j$. Then $L_{ij} \neq 0$ if and only if at least one of the following conditions holds:

  1) $A_{ij} \neq 0$
  2) For some $k < j$, $L_{ik} \neq 0$ and $L_{jk} \neq 0$.

- Proof is straightforward.
  - Consider the left-looking formulation of Gaussian elimination.

    $$A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$$



Modeling sparse Cholesky factorization

- Since sparse Cholesky factorization is stable without pivoting, we look for a simple way to view the elimination process without any consideration of the numerical values.
- In particular, we are interested in manipulating and analyzing the sparsity structure of the matrix.

- We use a graph-theoretic approach.

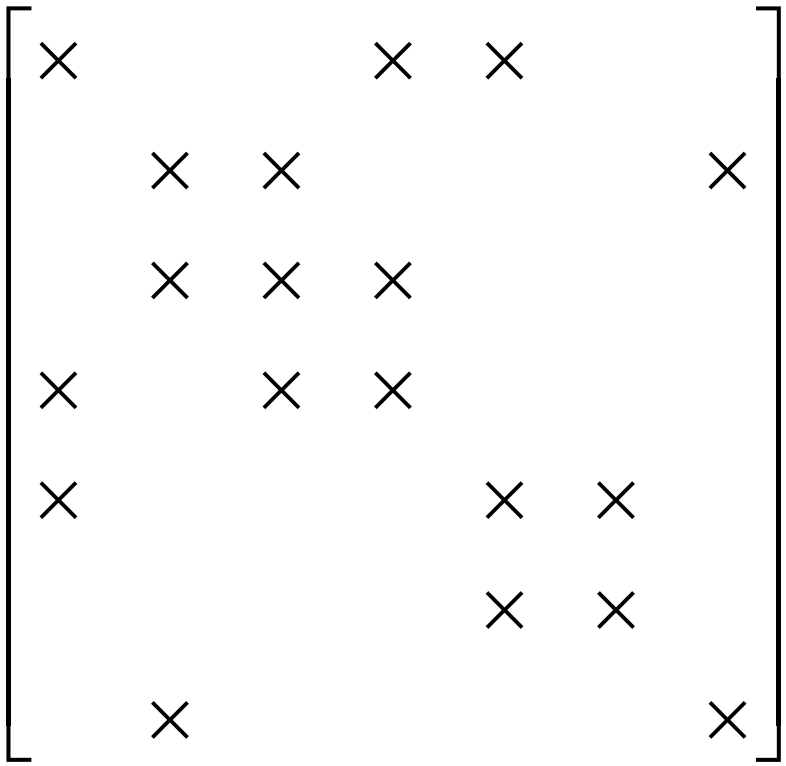
- Let A be an n by n sparse SPD matrix.
- Let L denote the Cholesky of A.
  - That is, $A = LL^T$.

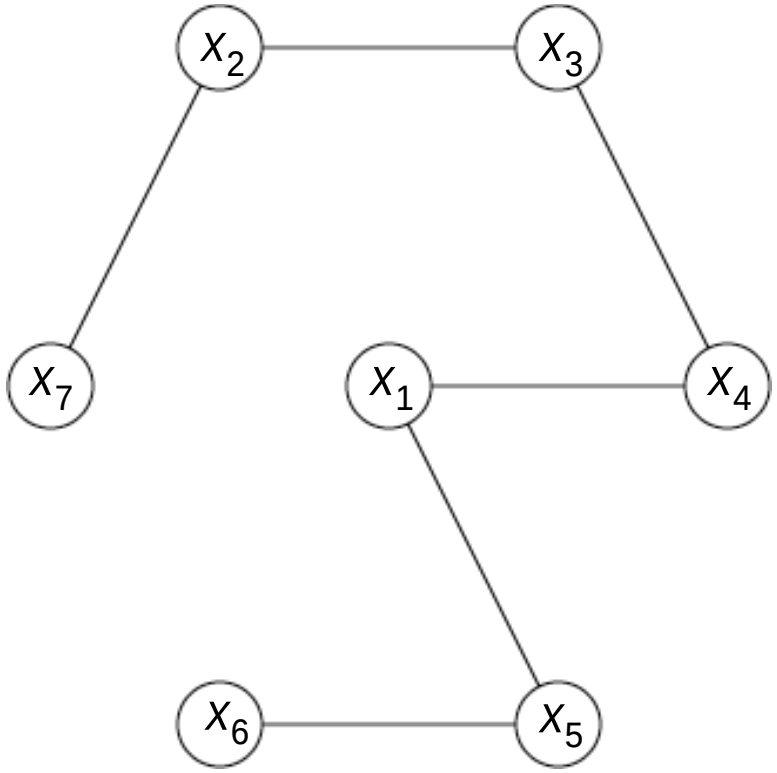
Modeling sparse Cholesky factorization

- The graph-theoretic approach is concerned with the connection between the rows and columns in the matrix A.

- We use an undirected graph $G = (X,E)$ to describe the sparsity structure of A.
  - $X = \{ x_1, x_2, \ldots, x_n \}$ is a set of n vertices.
  - Vertex x_i is associated with row and column i of A.
  - E be a set of edges; each edge joins a pair of distinct vertices.
  - There is an edge $\{x_i,x_j\} \in E$ if and only if A_{ij} is nonzero.
  - We do not allow $\{x_i,x_i\}$; that is, we do not describe the diagonal.

Example illustrating the graph model



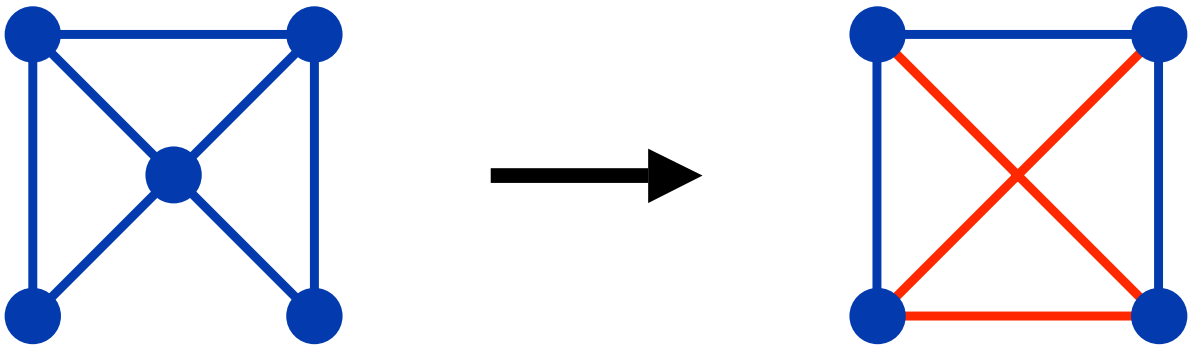


- The graph shows the structure of the matrix (or the relationship between the rows and columns without the numeric information).

A game on graphs

- The graph representation can be used to study the factorization process and analyze how fill arises.

- Rules of one step of the game [Rose ('72)]:
  - Pick a vertex *v* in the graph *G*.
  - Remove v and the edges that are incident on *v*.
  - Add edges to *G* to make the vertices adjacent to *v* into a clique (a complete subgraph).



A game on graphs

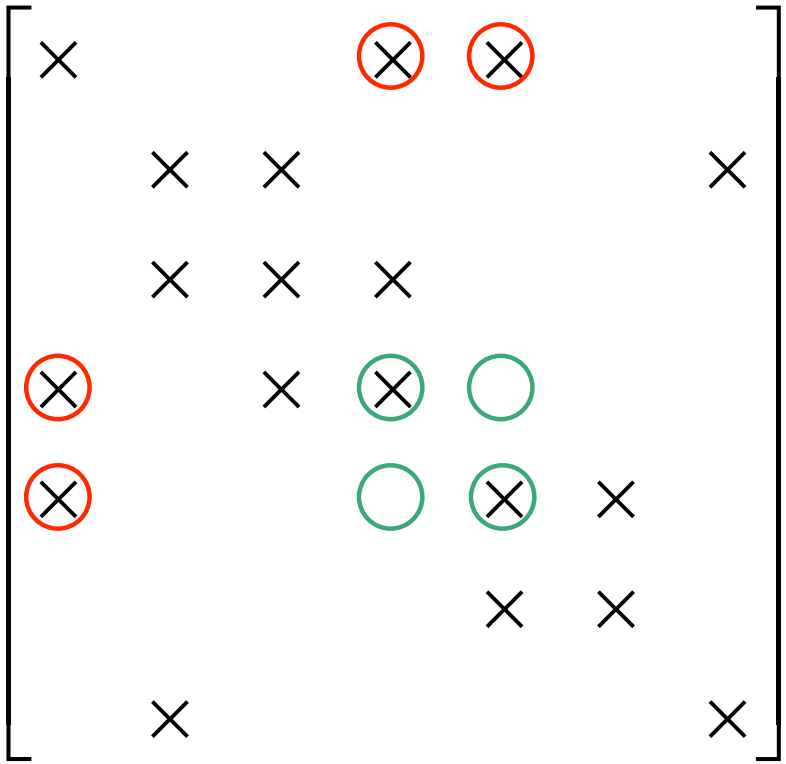
- Rules of one step of the game:
  - Pick a vertex *v* in the graph *G*.
  - Remove v and the edges that are incident on *v*.
  - Add edges to *G* to make the vertices adjacent to *v* into a clique (a complete subgraph).

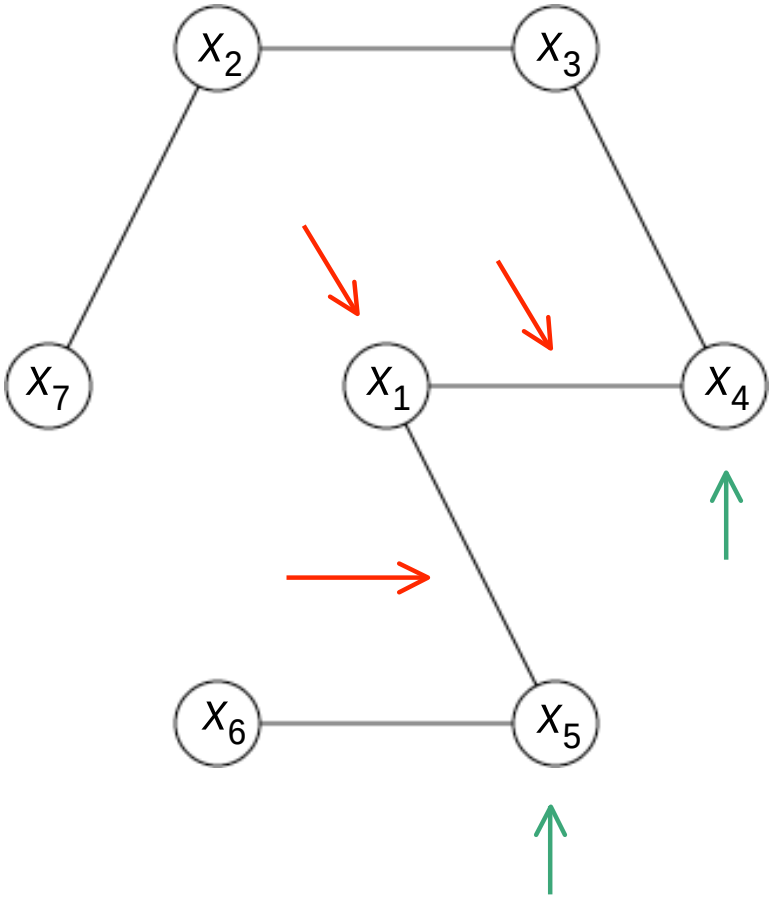
- These rules correspond to the operations involved in one step of Cholesky factorization.

- Consider the previous example.

Example illustrating the graph model

- Start of Cholesky factorization …



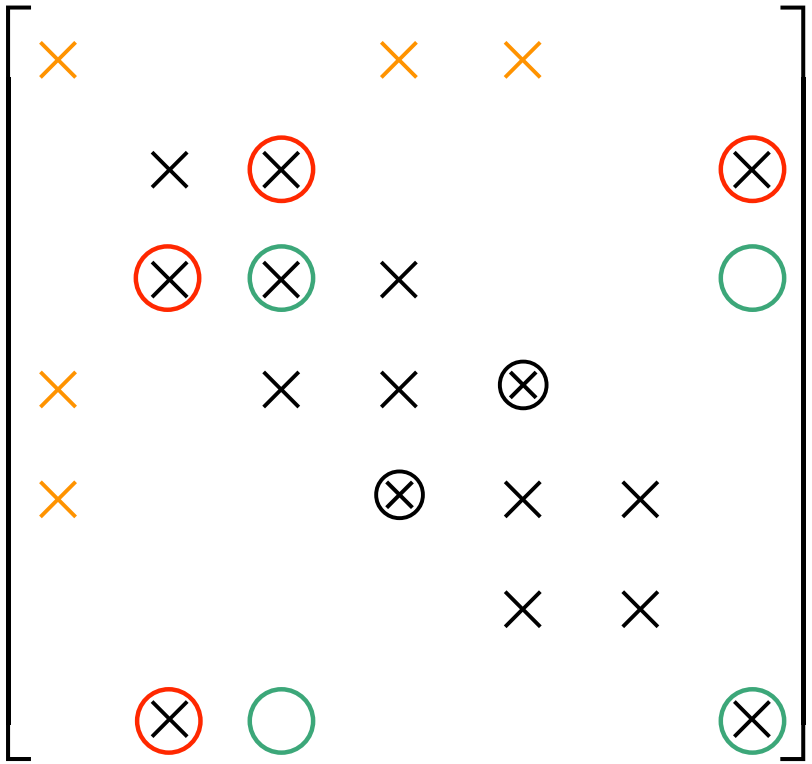


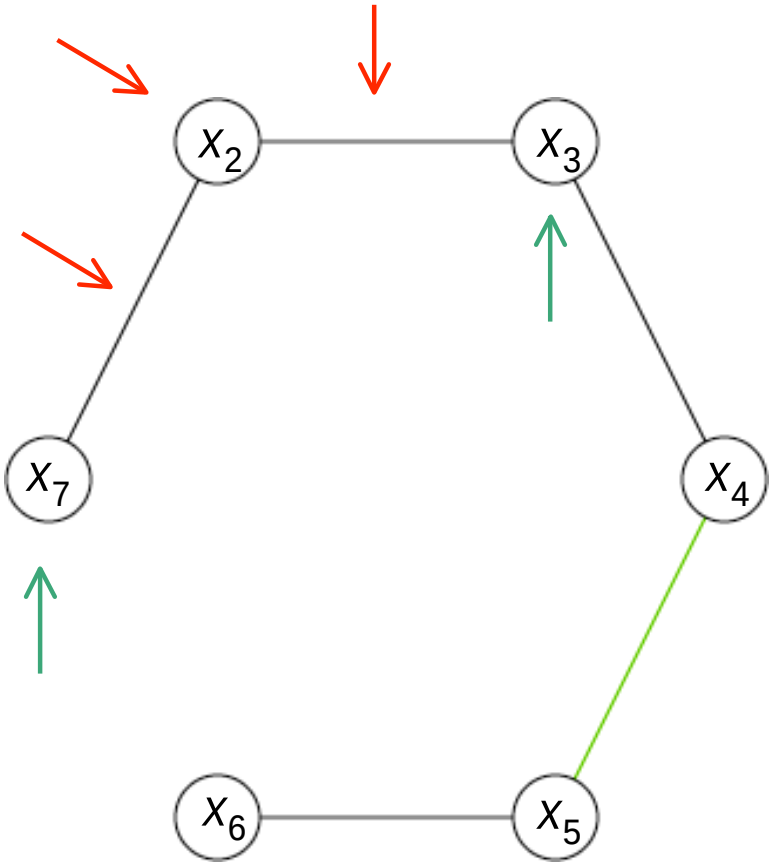
Example illustrating the graph model

- Facts:
  - Suppose column 1 of A has m off-diagonal nonzero elements
  - Then the elimination of row 1 and column 1 results in a rank-1 update, which contains an m by m dense submatrix.
  - The m off-diagonal nonzero elements correspond to the m vertices that are adjacent to x_1 in G.
  - According to the rules, after x_1 (and the incidence edges onto x_1) are removed from G, edges are added to the graph so that the m adjacent vertices are pairwise connected.
    - The m adjacent vertices form a clique.

Example illustrating the graph model

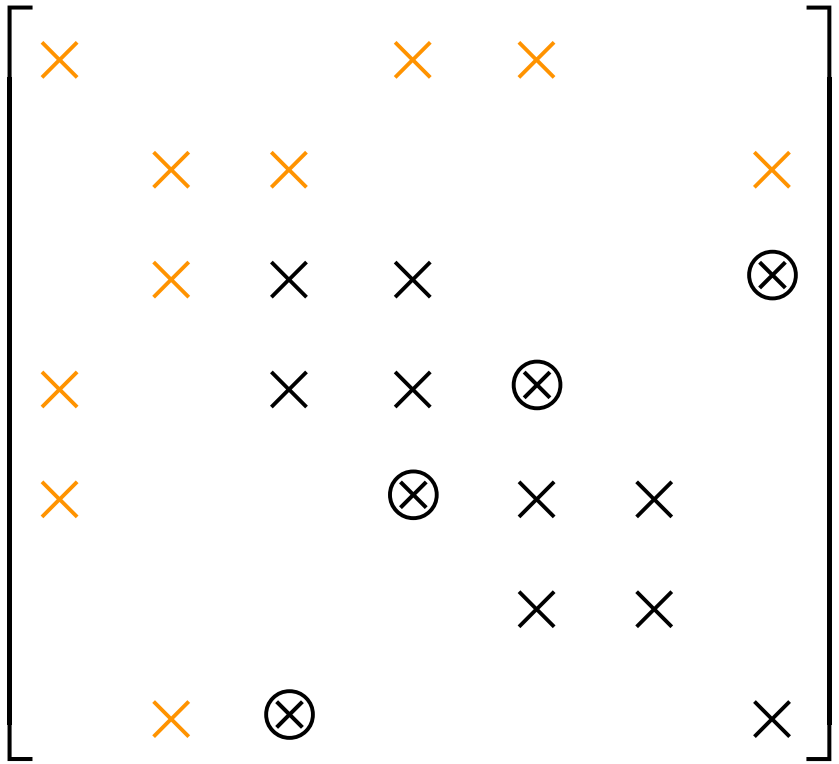
- After step 1 of Cholesky factorization …

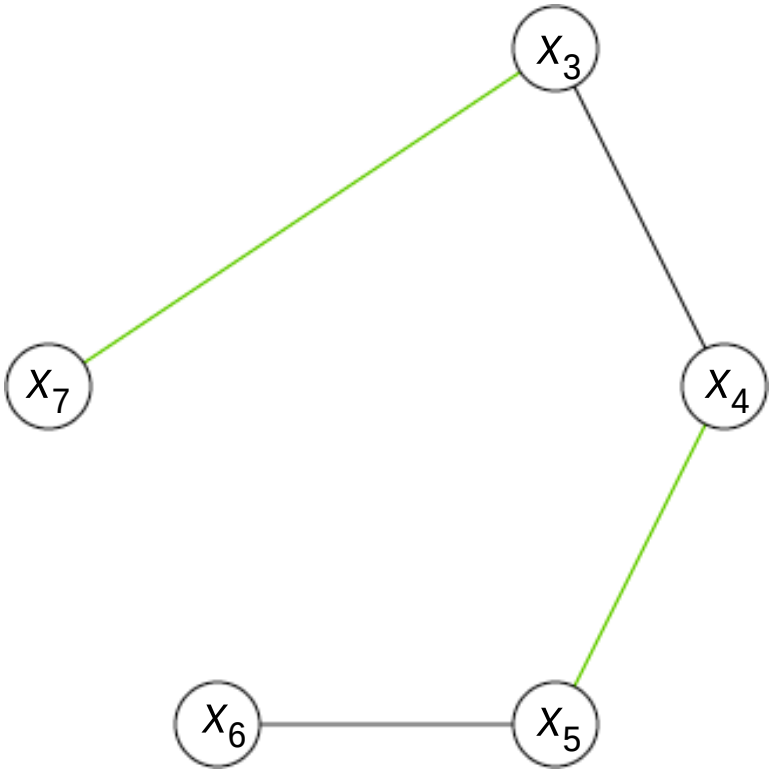




Example illustrating the graph model

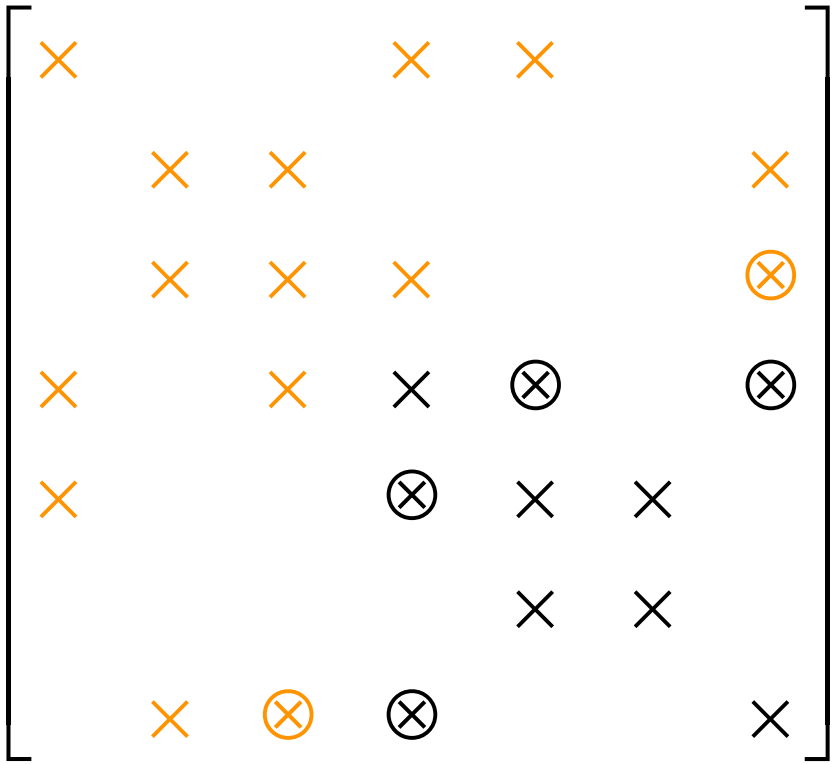
- After step 2 of Cholesky factorization …

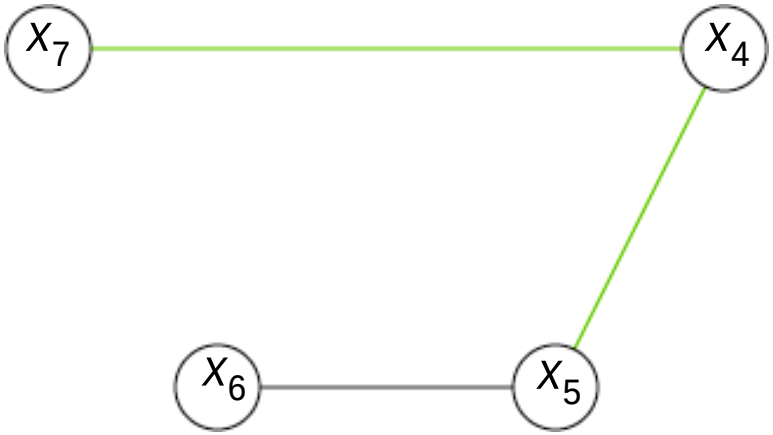




Example illustrating the graph model

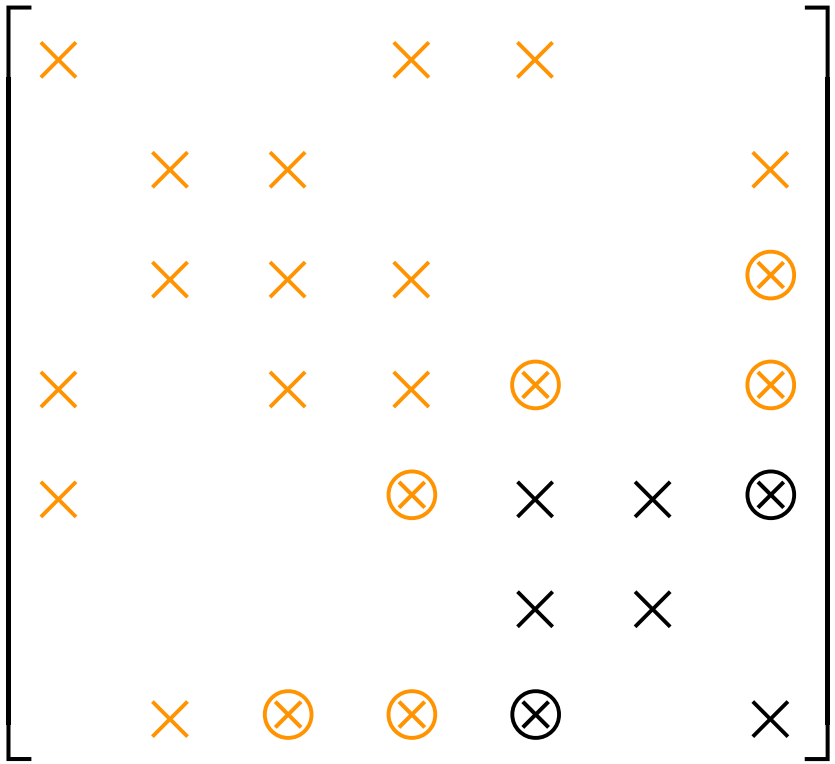
- After step 3 of Cholesky factorization …

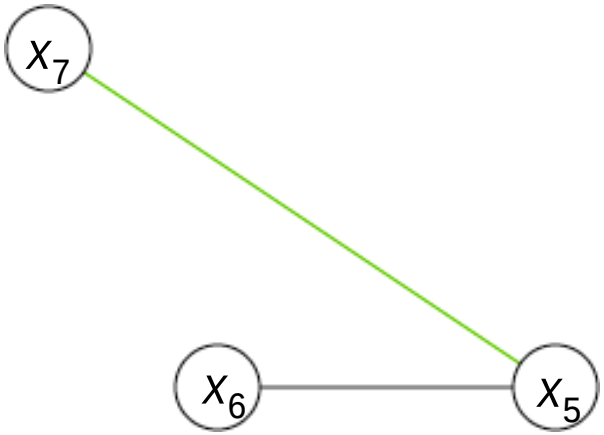




Example illustrating the graph model

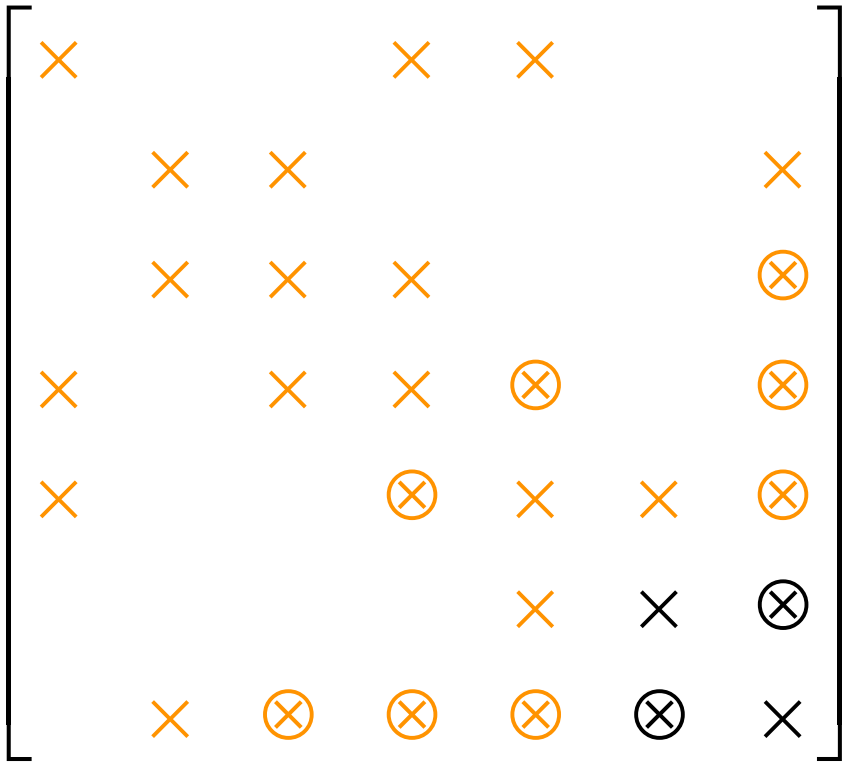
- After step 4 of Cholesky factorization …

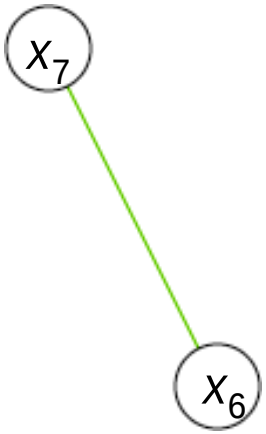




Example illustrating the graph model

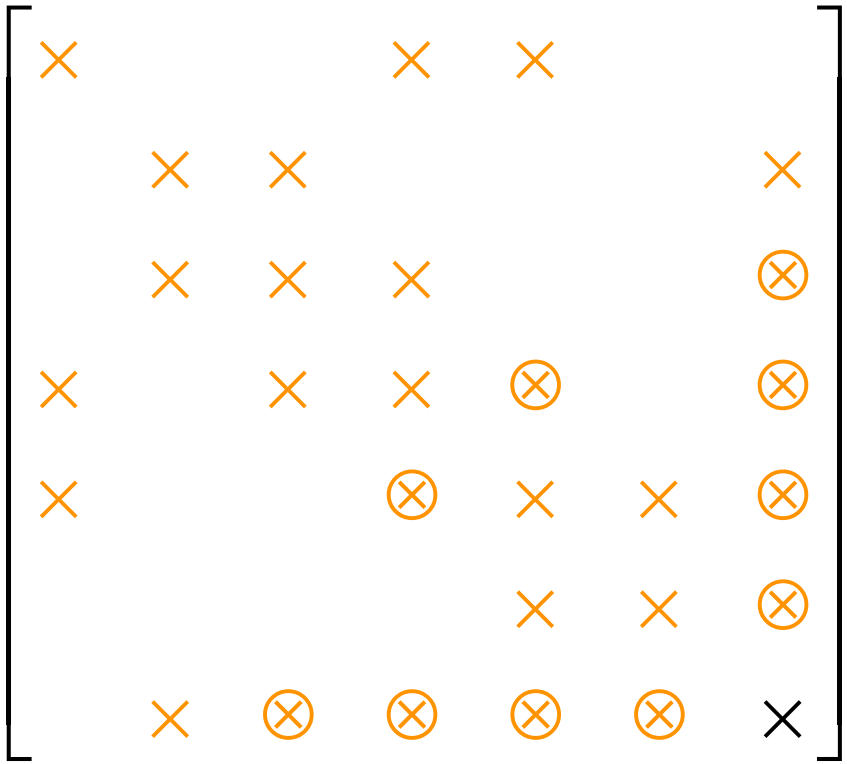
- After step 5 of Cholesky factorization …





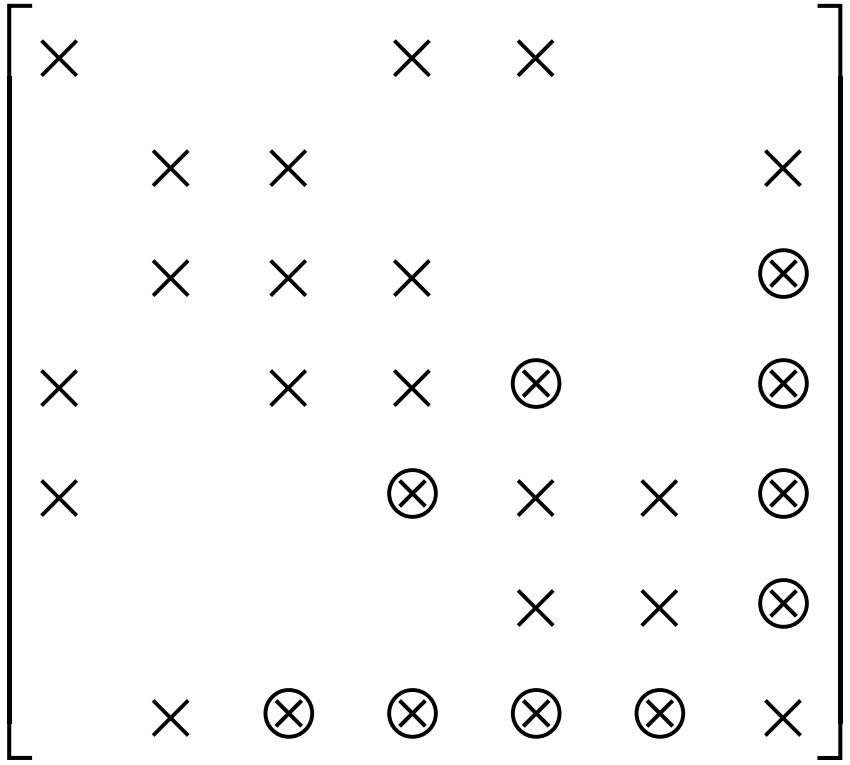
Example illustrating the graph model

- After step 6 of Cholesky factorization …



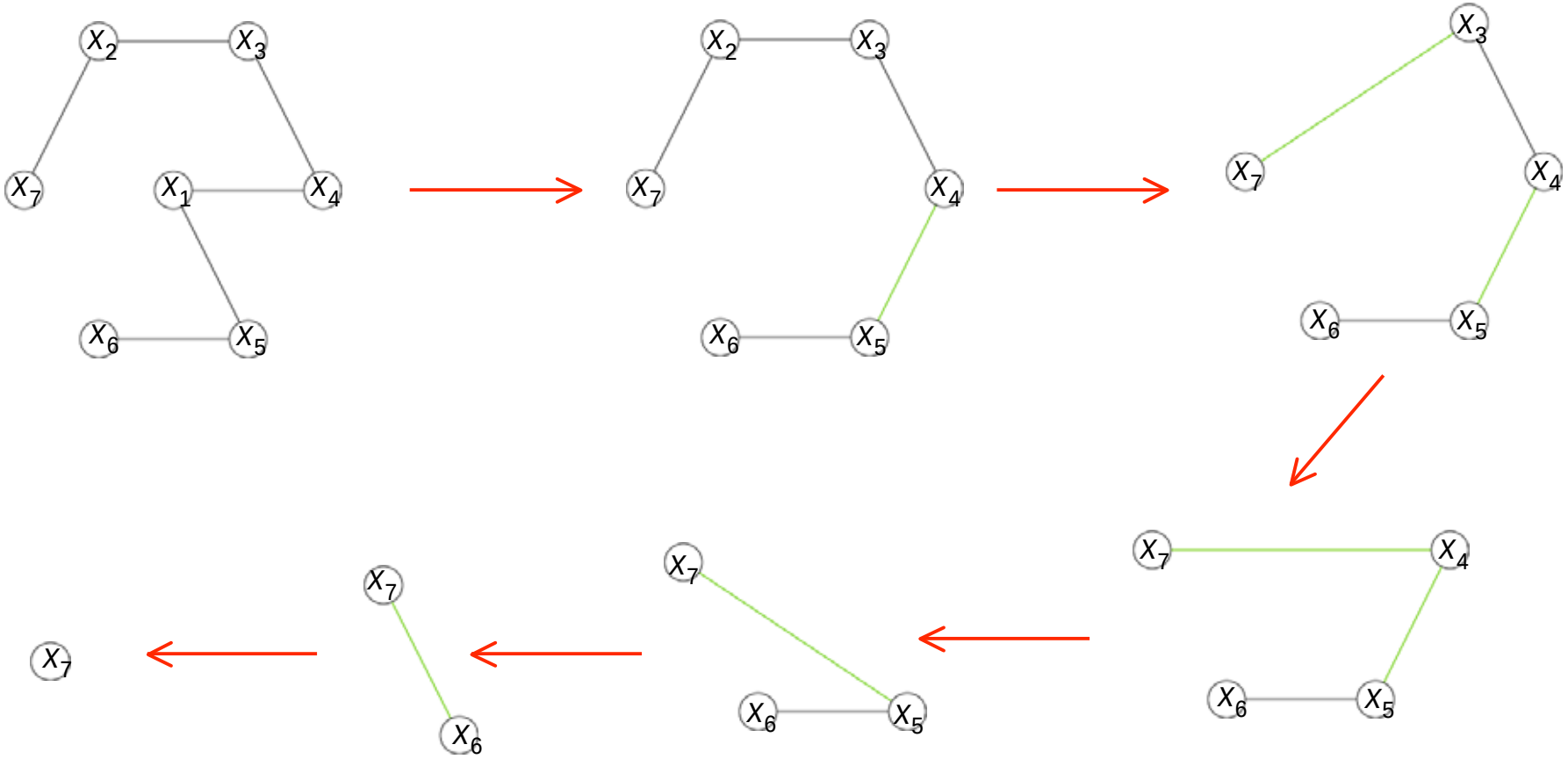
Example illustrating the graph model

- In matrix terminology, we have a sequence of elimination steps.
- $L+L^T$ is sometimes referred to as the filled matrix.
- The sparsity structure of $L+L^T$ is given by:



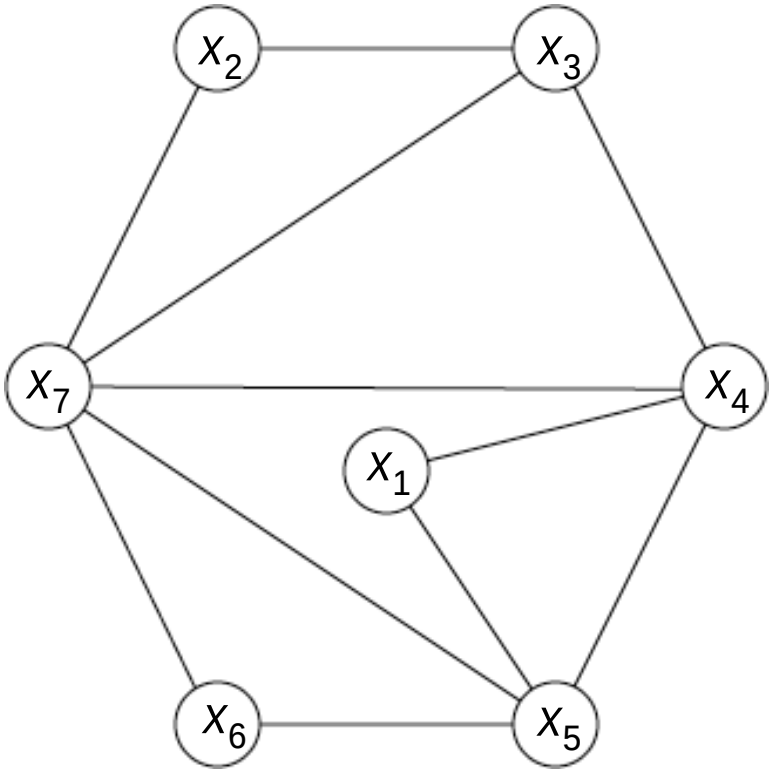
Example illustrating the graph model

- In graph-theoretic terminology, we have a sequence of elimination graphs, each represents the sparsity structure of the Schur complement.



Filled matrix and filled graph

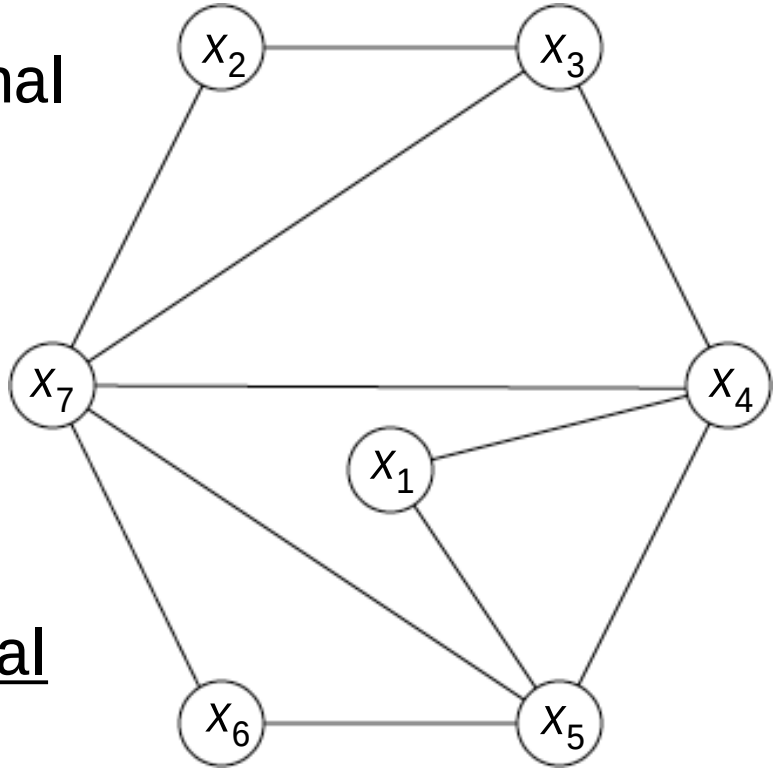
- If we "merge" all the elimination graphs together, we obtain the filled graph, which gives the structure of the filled matrix.



$$\begin{bmatrix} \times & & & \times & \times & & \\ & \times & \times & & & & \times \\ & \times & \times & \times & & & \otimes \\ \times & & \times & \times & \otimes & & \otimes \\ \times & & & \otimes & \times & \times & \otimes \\ & & & & \times & \times & \otimes \\ & \times & \otimes & \otimes & \otimes & \otimes & \times \end{bmatrix}$$

Filled graph

- The filled graph is a very special graph.
- A lot is known about its properties.
  - The filled graph depends on the original graph and the order in which the vertices are eliminated.
  - The filled graph is a chordal graph (also known as a triangulated graph).
  - There is a way (which may not be unique) to eliminate the vertices in the filled graph so that no additional edges are added; e.g., the sequence that was used to generate the filled graph.
  - [Rose ('72); Golumbic ('80): "Algorithmic Graph Theory and Perfect Graphs"].



Perfect elimination

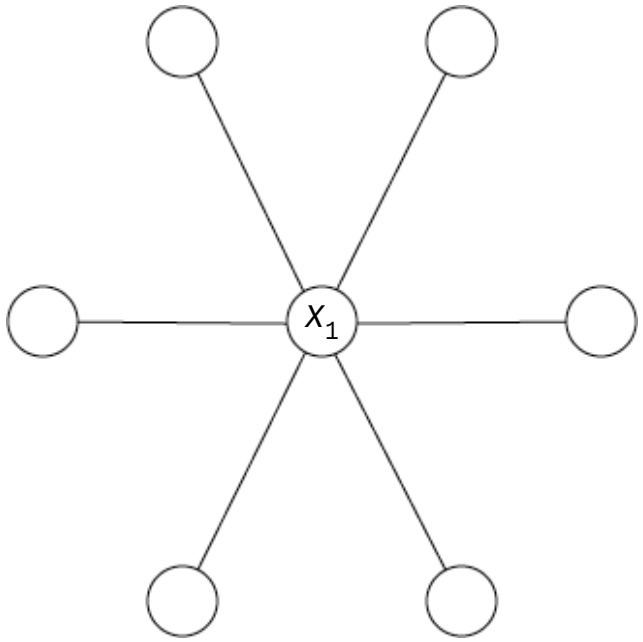
- If no new edges are added to the elimination graphs when the vertices are eliminated, then the order in which the vertices are eliminated is called a perfect elimination sequence (or perfect elimination order).

- It is generally very hard to determine if a given graph has a perfect elimination sequences.

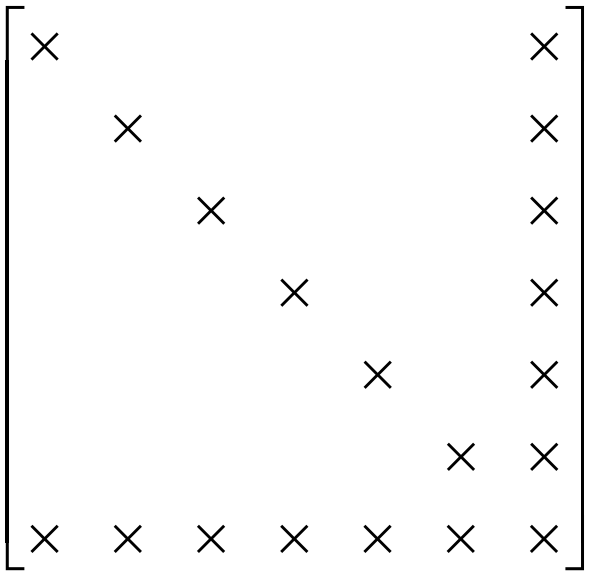
Example

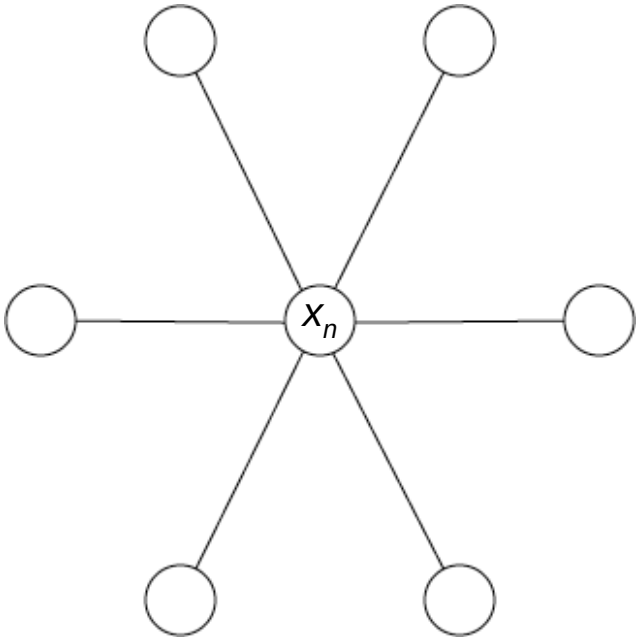
$$\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & & & & & \\ \times & & \times & & & & \\ \times & & & \times & & & \\ \times & & & & \times & & \\ \times & & & & & \times & \\ \times & & & & & & \times \end{bmatrix}$$

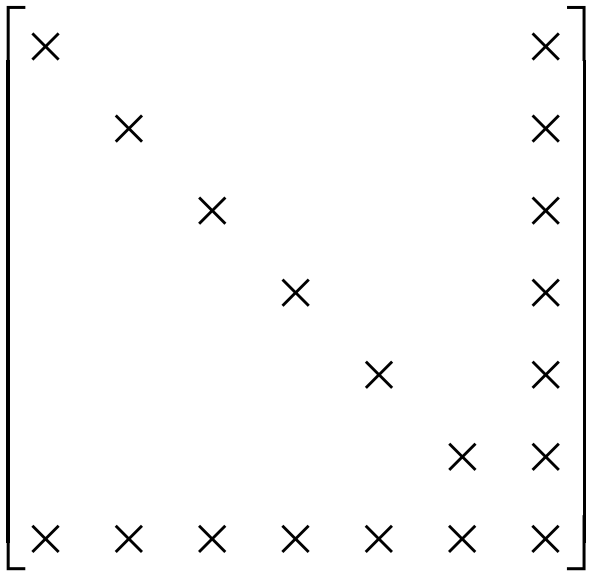
$$\begin{bmatrix} \times & \times & \times & \times & \times & \times & \times \\ \times & \times & \otimes & \otimes & \otimes & \otimes & \otimes \\ \times & \otimes & \times & \otimes & \otimes & \otimes & \otimes \\ \times & \otimes & \otimes & \times & \otimes & \otimes & \otimes \\ \times & \otimes & \otimes & \otimes & \times & \otimes & \otimes \\ \times & \otimes & \otimes & \otimes & \otimes & \times & \otimes \\ \times & \otimes & \otimes & \otimes & \otimes & \otimes & \times \end{bmatrix}$$



Example







Parter's lemma

- Lemma [Parter, '61]

  Let $i > j$. Then $L_{ij} \neq 0$ if and only if at least one of the following conditions holds:

  1) $A_{ij} \neq 0$
  2) For some $k < j$, $L_{ik} \neq 0$ and $L_{jk} \neq 0$.

- Same lemma, but in graph-theoretic terms …

  Let $G = (X,E)$ be the graph of a SPD matrix A. Denote the filled graph of A by $G^+ = (X^+,E^+)$. Then $\{x_i,x_j\} \in E^+$ if and only if at least one of the following conditions holds:

  1) $\{x_i,x_j\} \in E$
  2) $\{x_i,x_k\} \in E^+$ and $\{x_k, x_j\} \in E^+$ for $k < \min\{i,j\}$.

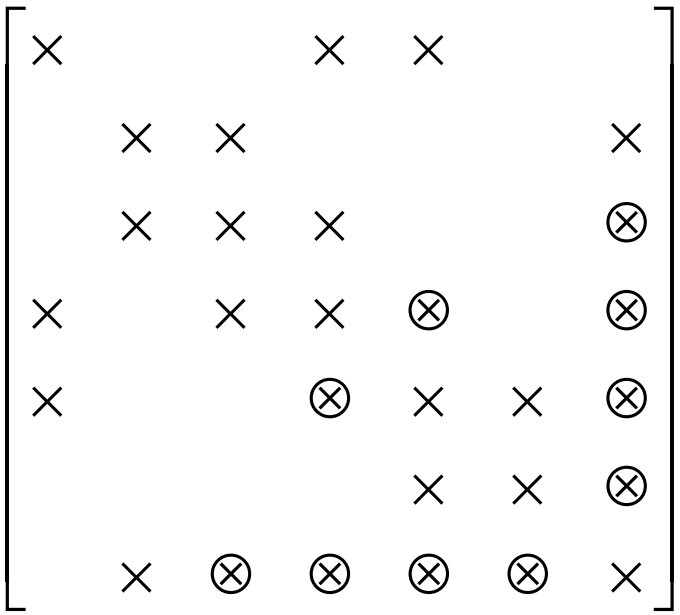
Modeling elimination

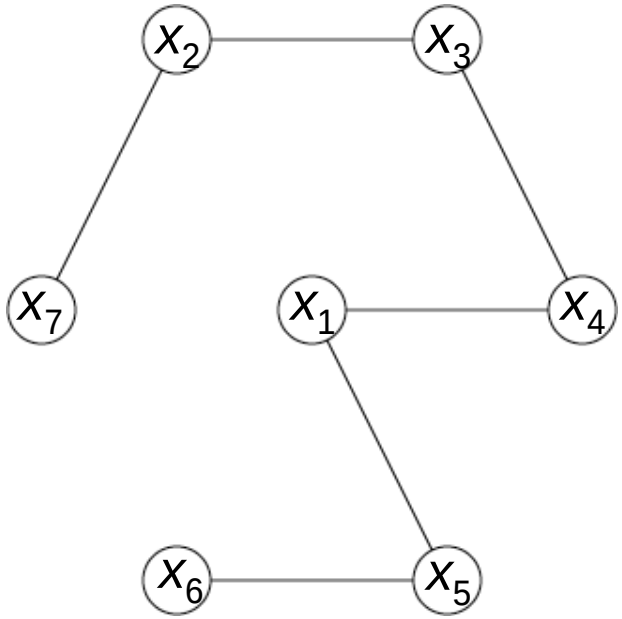
- The elimination graphs show the dynamic changes due to the deletion and addition of edges.
- It may not be practical or efficient to use from an implementation point of view.
  - Why?
- Are there alternative ways to study fill that are more amendable to efficient implementations?

Fill path theorem

- Fill Path Theorem:

  Let $G = (X,E)$ be the graph of a SPD matrix A. Denote the corresponding filled graph by $G^+ = (X^+,E^+)$. Then $\{x_i,x_j\} \in E^+$ if and only if there is a path $(x_i,x_{p_1},x_{p_2},\ldots,x_{p_s},x_j)$ in G such that $p_k < \min\{i,j\}$, for $1 \le k \le s$.





Fill path theorem

- Fill Path Theorem:

  Let $G = (X,E)$ be the graph of a SPD matrix A. Denote the corresponding filled graph by $G^+ = (X^+,E^+)$. Then $\{x_i,x_j\} \in E^+$ if and only if there is a path $(x_i,x_{p_1},x_{p_2},\ldots,x_{p_s},x_j)$ in G such that $p_k < \min\{i,j\}$, for $1 \le k \le s$.

- Proof is by induction.
  - "$\Leftarrow$"

    Assume that there is a path $(x_i,x_{p_1},x_{p_2},\ldots,x_{p_s},x_j)$ in G such that $p_k < \min\{i,j\}$, for $1 \le k \le s$.

    If $s = 0$, then the path is (x_i,x_j). By Parter's lemma, $\{x_i,x_j\} \in E^+$.

    If $s = 1$, then the path is (x_i,x_{p_1},x_j). By the time the elimination reaches x_{p_1}, the fill edge $\{x_i,x_j\}$ will be created in G^+.

Fill path theorem

- Proof (continued) …

  Now consider the path $(x_i, x_{p_1}, x_{p_2}, \ldots, x_{p_s}, x_j)$.

  Pick $p_h = \max\{ p_1, p_2, \ldots, p_s \}$.

  The path is now broken up into two:

  $(x_i, x_{p_1}, x_{p_2}, \ldots, x_{p_h})$

  $(x_{p_h}, x_{p_{h+1}}, \ldots, x_{p_s}, x_j)$

  By induction, the two (shorter) paths correspond, respectively, to two fill edge in G^+: $\{x_i, x_{p_h}\}$, $\{x_{p_h}, x_j\}$, with $p_h < \min\{i, j\}$. By Parter's lemma, this gives the fill edge $\{x_i, x_j\}$ in G^+.

  - "$\Rightarrow$"

    Left as exercise.

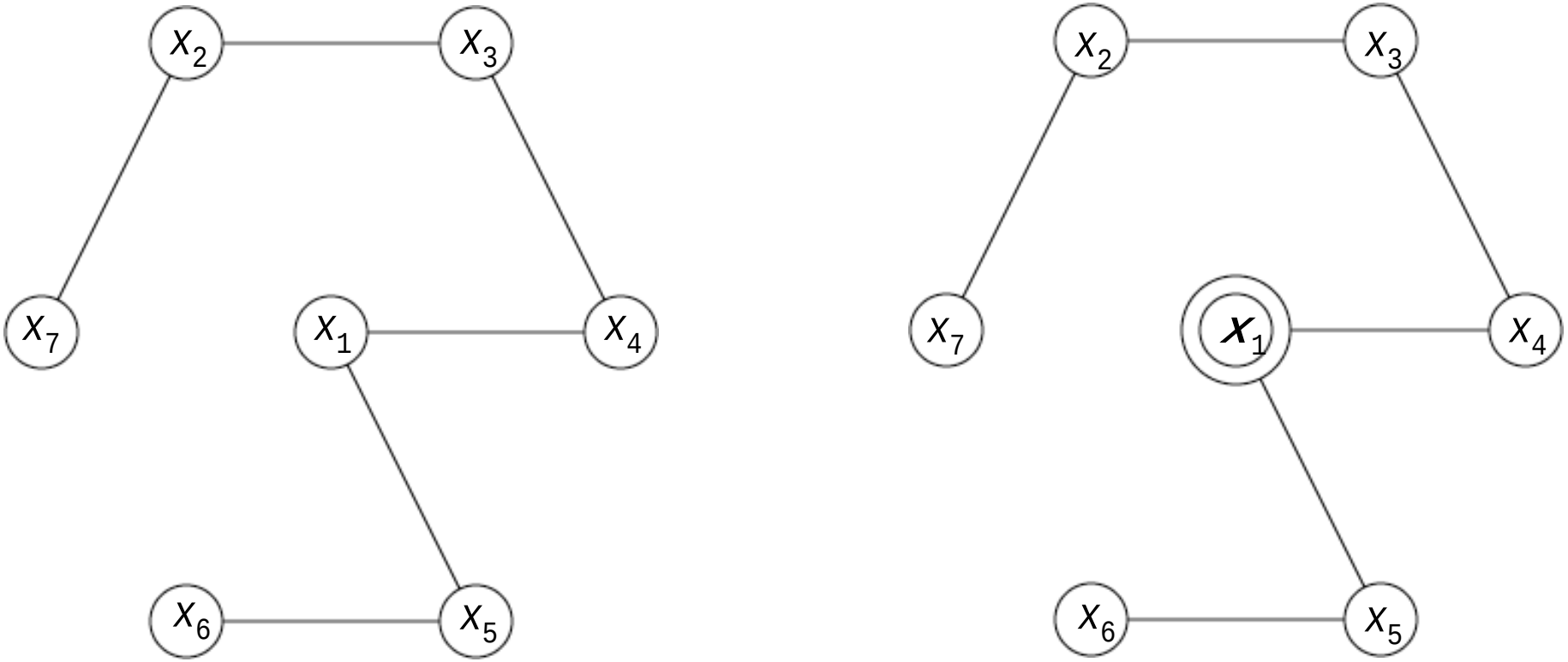
Fill path theorem

- The path $(x_i, x_{p_1}, x_{p_2}, \ldots, x_{p_s}, x_j)$ is often referred to as a fill path.
  - x_i is said to be reachable from x_j through vertices with smaller labelling.
- The importance of the fill path theorem is that it allows the fill edges to be identified from the original graph.
  - There is no need to generate the elimination graphs explicitly.
  - Consequently, there is no need to worry about removing and adding edges.
    - Good from the storage point of view.
  - BUT … paths have to be followed (i.e., traversed) to discover the fill edges.
    - Bad in terms of execution time.

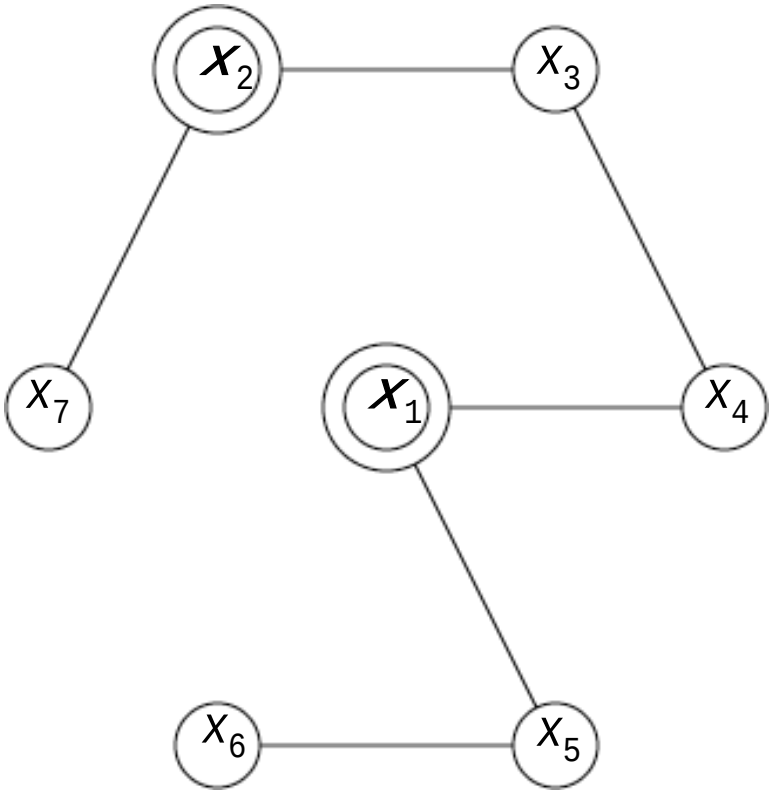
Fill paths

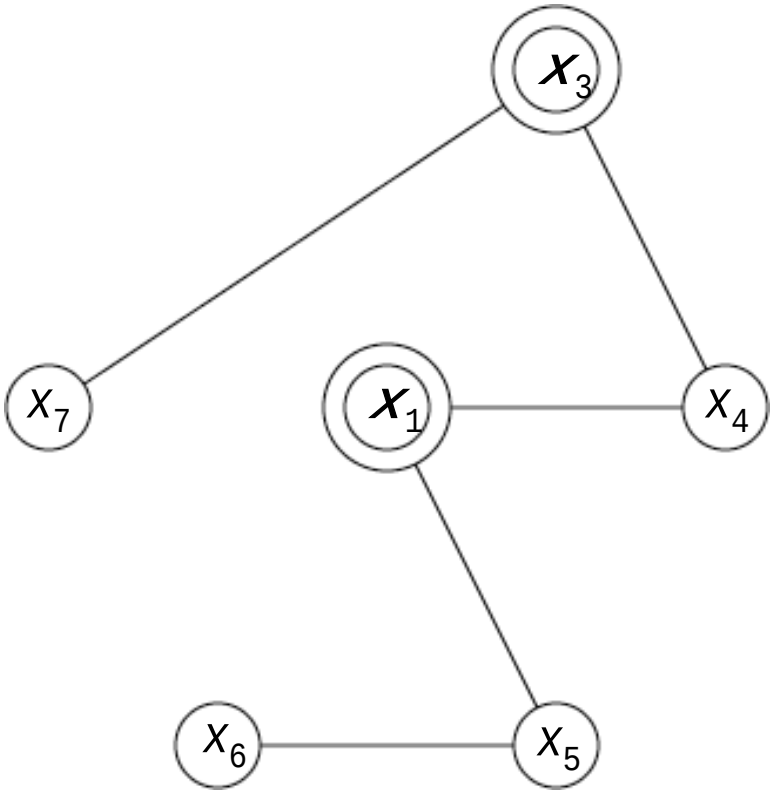
- Remedy:
  - A fill path is a path $(x_i, x_{p_1}, x_{p_2}, \ldots, x_{p_s}, x_j)$ in G such that, for $1 \le k \le s$, $p_k < \min\{i, j\}$.
  - All we need is the existence of such a path. We actually do not need to know the actual "intermediate" vertices along the path.
  - Let's coalesce the "intermediate" vertices into a "supervertex", say, x_{p_s}. So, the fill path is now represented by (x_i, x_{p_s}, x_j).
  - The graph is now "compressed" and becomes smaller (in general).

Compressing fill paths

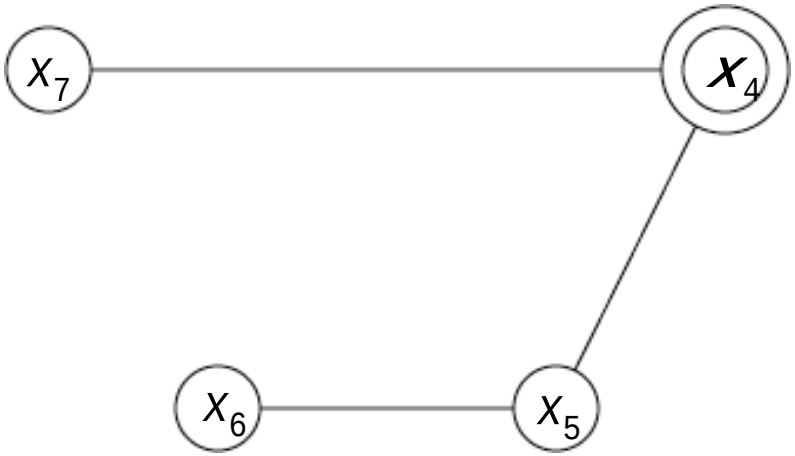


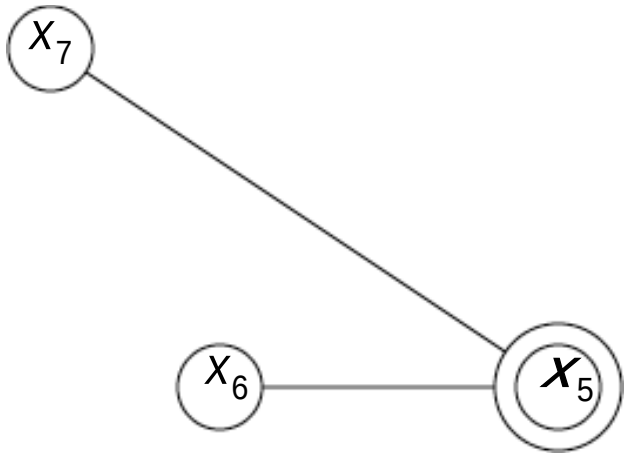
Compressing fill paths

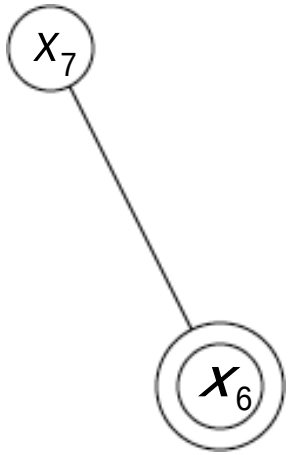




Compressing fill paths







Notion of quotient graphs

- Remedy:
  - The graph is now “compressed” and becomes smaller (in general).
  - Each “compressed” graph is referred to as a quotient elimination graph [George & Liu (‘79)].
  - The quotient elimination graphs carries the same information as the elimination graphs.

Quotient elimination graphs

- Elimination graphs versus quotient elimination graphs:
  - The most important difference is that the length of each fill path in a quotient elimination graph is never more than 2.
  - Another important observation is that the maximum number of edges one will see in a quotient elimination graph is never more than the number of edges in the original graph.
    - Each quotient elimination graph can be store in the space provided for the original graph [George & Liu ('79)].
    - The use of quotient graphs avoids the unpredictability of the number of edges in an elimination graph.
  - It also avoids the need to traverse long fill paths in the fill path theorem.
- Exercise: Graph model for Gaussian elimination of sparse nonsymmetric matrices?

Summary …

- Analysis of sparsity structure in Cholesky factorization.

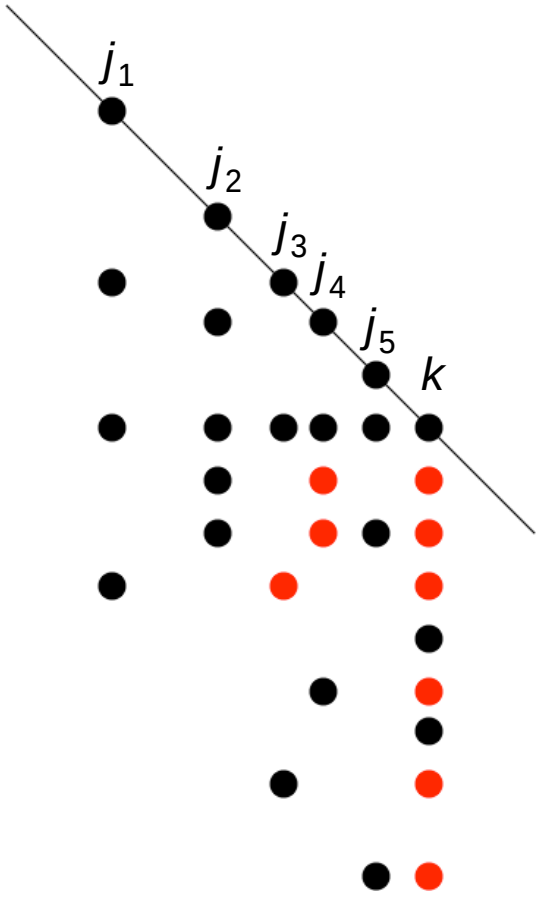
- Graph models.
  - Elimination graphs.
  - Reachability.
  - Quotient elimination graphs.

Symbolic factorization

- Given a SPD matrix *A*. Denote its Cholesky factor by *L*.
- The structure of *L* depends solely on the structure of *A*.
  - Because the factorization is numerical stable without pivoting.
- In theory, the structure of *L* can be computed from the structure of *A*.
  - Simulating numerical factorization is not efficient, since the number of operations required is the same as that in numerical factorization.

Discovering sparsity

- Consider the right-looking formulation of Cholesky factorization.
  - Column j_1 of L modifies columns j_3 and k of A.
  - Column j_2 of L modifies columns j_4 and k of A.
  - Column j_3 of L modifies column k of A.
  - Column j_4 of L modifies column k of A.
  - Column j_5 of L modifies column k of A.

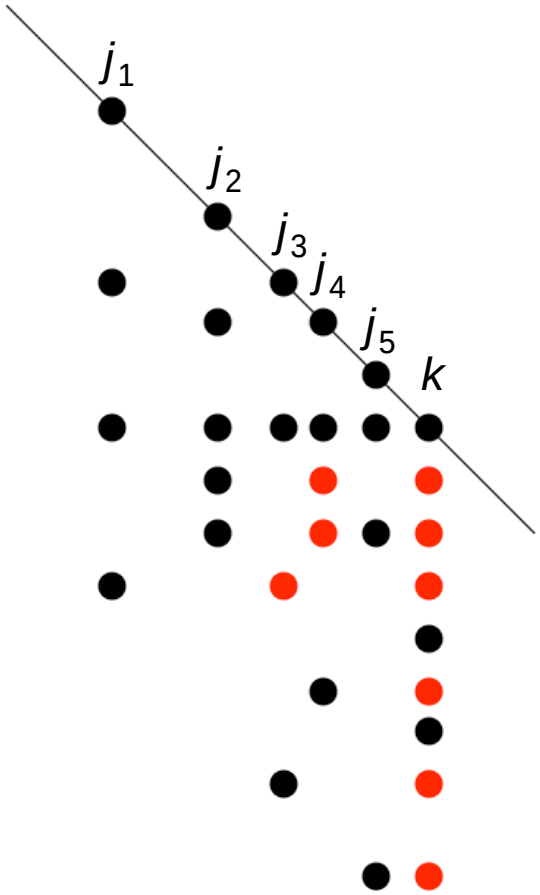


- So, to determine the structure of column k of L, it looks like the structure of columns j_1, j_2, j_3, j_4, and j_5 of L are needed.

Discovering sparsity

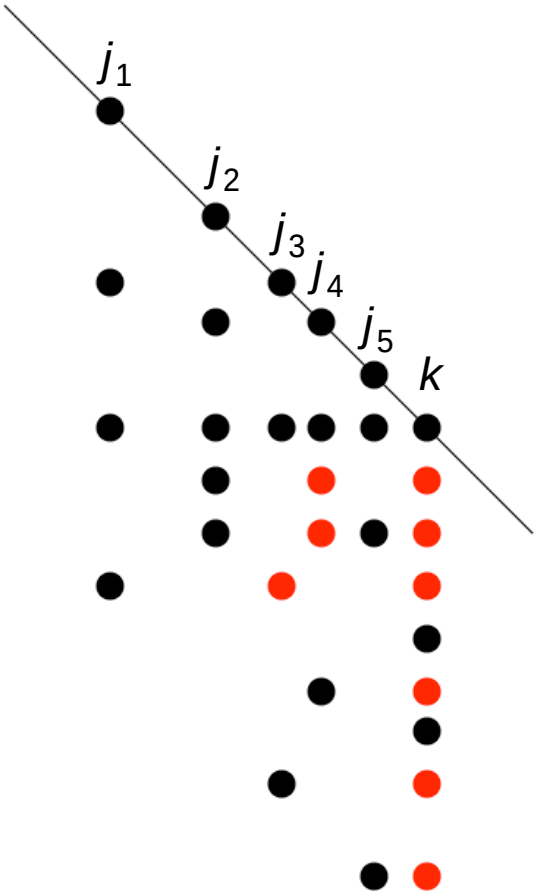
- Observation: When column j_1 / j_2 of L modifies column j_3 / j_4 of A, column j_3 / j_4 of A inherits the sparsity structure of column j_1 / j_2 of L.

- When we compute the structure of column k of L, it is redundant to consider columns j_1 and j_2 of L, since any nonzero positions in those two columns will also appear in columns j_3 and j_4 of L, respectively.



Discovering sparsity

- That is, for the purpose of determining the sparsity structure of column k of L, it is sufficient to consider the sparsity structure of columns j_3, j_4, and j_5 of L, in addition to column k of A.
- Note that the first off-diagonal nonzero elements in columns j_3, j_4, and j_5 of L are in row k.
  - This is not by accident!



Discovering sparsity

- If M is a matrix, then let Struct(M) denote the sparsity structure of M.
  - That is, Struct(M) = $\{(i,j) : M_{ij} \neq 0\}$.
- Also, if L is the Cholesky factor of an n by n SPD matrix A, then for each column of L, define First(j) as followed.
  - If column j of L has more than one nonzero element, then First(j) is the row subscript of the first off-diagonal nonzero element in that column; i.e., First(j) = min $\{ i > j : L_{ij} \neq 0 \}$.
  - Otherwise, First(j) = n+1.

Structural result

- Theorem [Sherman ('75)]:

  Let A be a SPD matrix and L be its Cholesky factor. The structure of column k of L is given by

$$\text{Struct}\left(L_{*k}\right) = \text{Struct}\left(A_{*k}\right) \cup \left\{ \bigcup_{\substack{j \text{ such that} \\ \text{First}(j)=k}} \text{Struct}\left(L_{*j}\right) \right\} - \left\{1, 2, \cdots, k-1\right\}$$

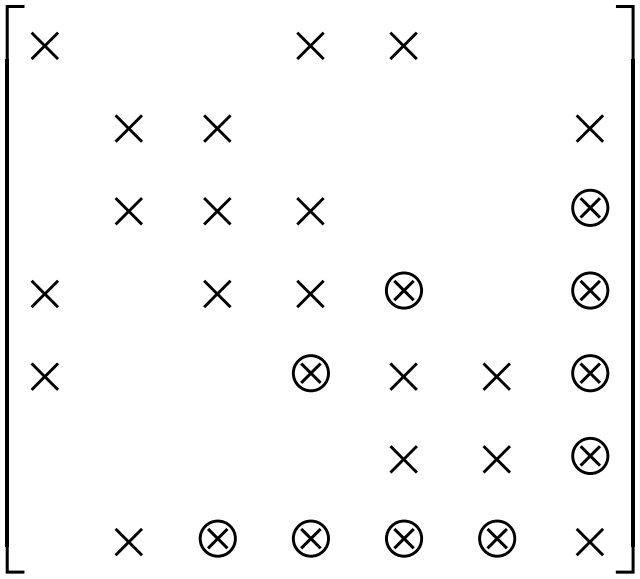
Symbolic factorization

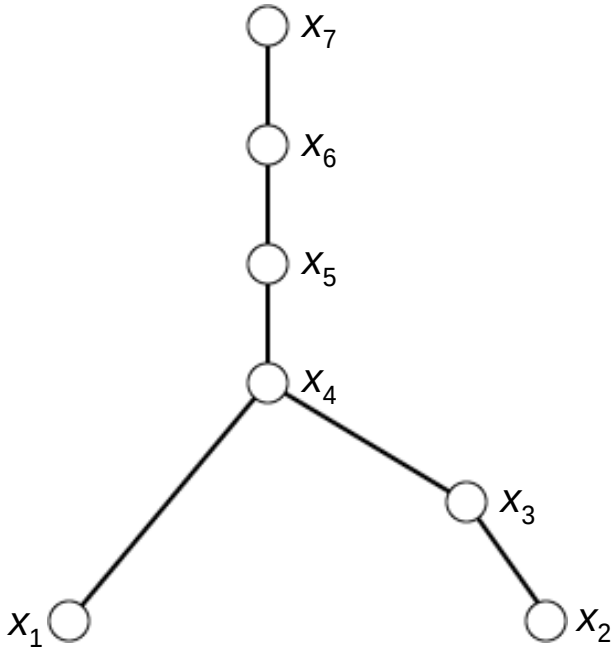
$$\text{Struct}\left(L_{*k}\right) = \text{Struct}\left(A_{*k}\right) \cup \left\{ \bigcup_{\substack{j \text{ such that} \\ \text{First(j)=k}}} \text{Struct}\left(L_{*j}\right) \right\} - \left\{1, 2, \cdots, k-1\right\}$$

- This theorem provides a constructive and efficient way to compute the structure of L_{*k} without computing the entries.
  - The number of operations required is much less than the number of operations in numerical factorization.
- The process of determining the structure of L is called symbolic factorization.
- Knowing the structure of L in advance allows a compact and efficient data structure to be set up for storing the nonzero entries prior to numerical factorization.

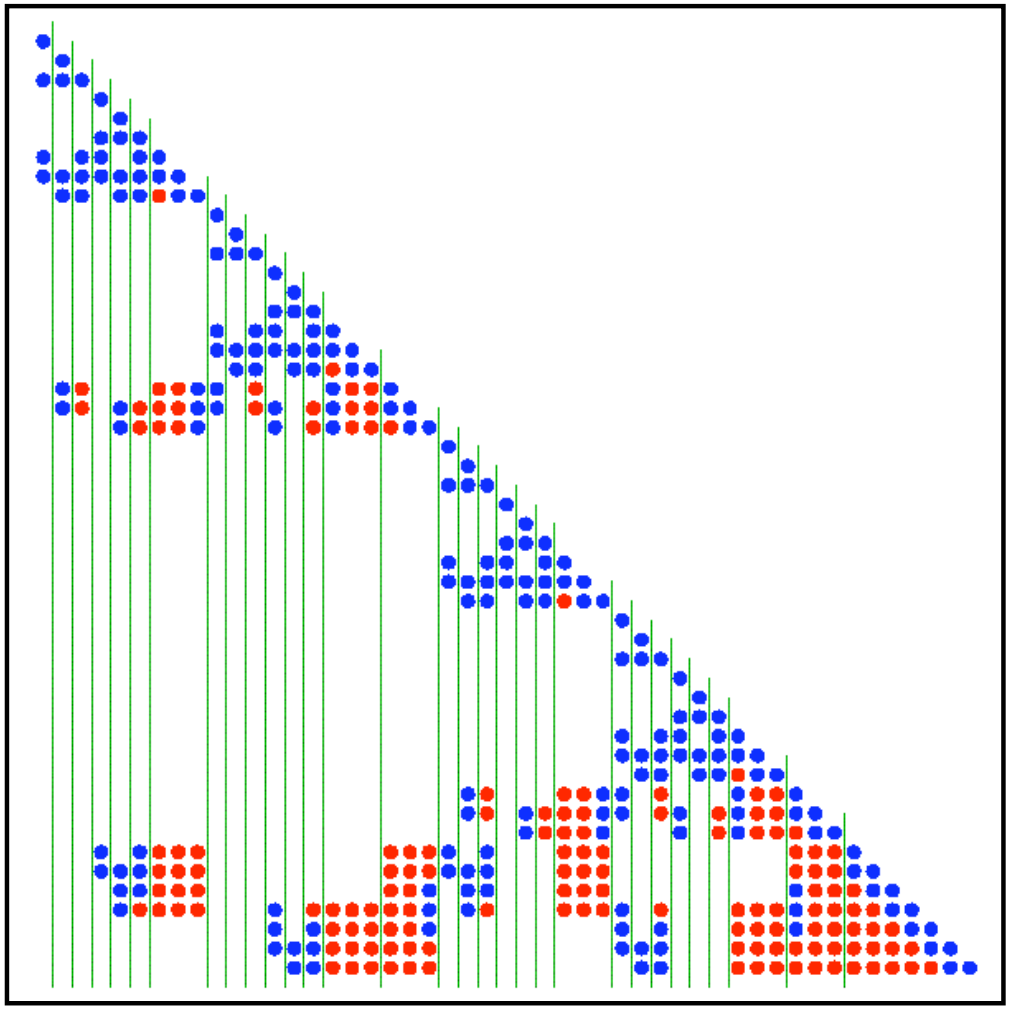
Elimination tree

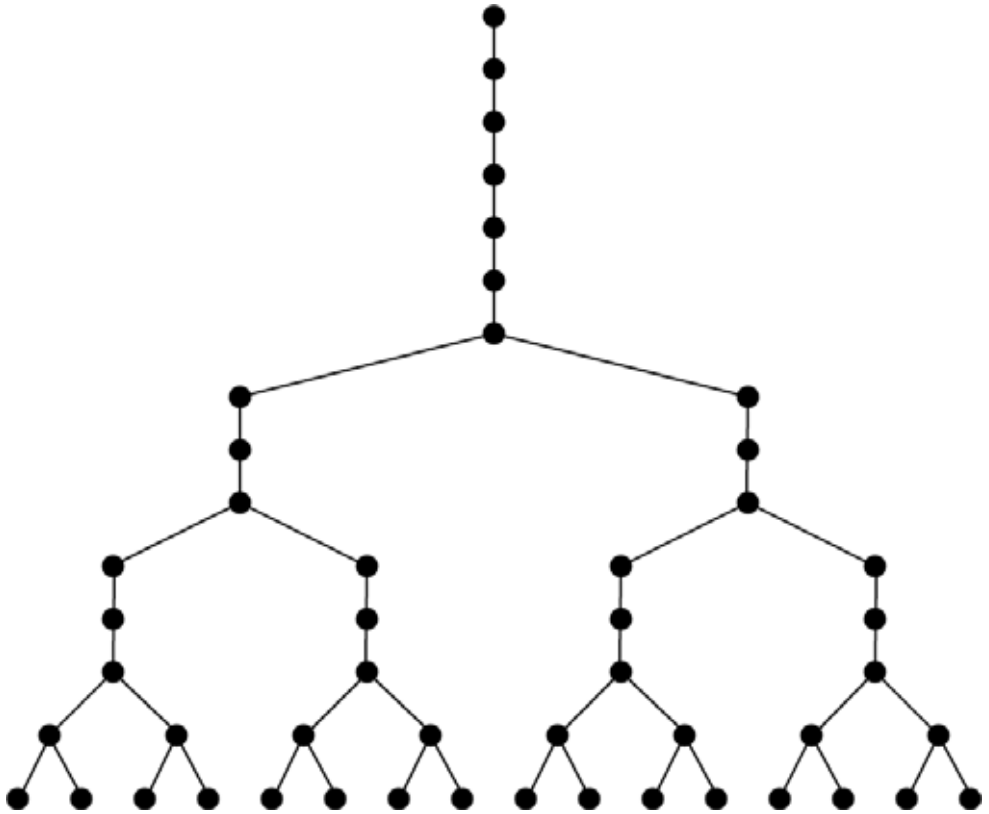
- Let A be an n by n SPD matrix and L be its Cholesky factor.
- The values of First(j), $1 \le j \le n$, can be used to construct a special graph: $T(A) = (X,F)$.
  - $X = \{ x_1, x_2, \ldots, x_n \}$, where x_i corresponds to the ith row/column of the matrix A.
  - Let $i < j$. Then $\{x_i,x_j\} \in F$ if and only if j = First(i) and $j \neq n+1$.





Elimination tree





Elimination tree

- $T(A)$ ...
  - It is an acyclic graph (i.e., a tree).
    - There are n vertices and at most n–1 edges.
      - When will $T(A)$ have less than n–1 edges?
  - $T(A)$ is called the elimination tree of A.
    - An important and powerful tool [Schreiber ('82); Liu ('86)].
  - There is a sense of direction.
    - x_n is often referred to as a root of $T(A)$.
      - How many roots can $T(A)$ have?
    - Recall that $\{x_i, x_{\text{First}(i)}\}$ is an edge in $T(A)$; x_i is called a child of $x_{\text{First}(i)}$ and $x_{\text{First}(i)}$ is called the parent of x_i.

Elimination tree

- $T(A)$, together with Struct(A), can be used to characterize the column structure and row structure of L.

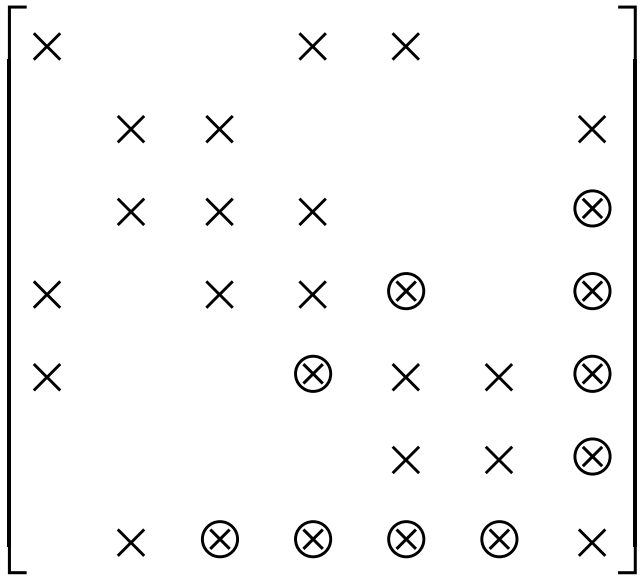
- Ancestors and descendants:
  - Let x_r be a root of $T(A)$. Suppose that there is a path between x_j and x_r in $T(A)$. If x_i ($i > j$) is on the path from x_j to x_r, then x_i is an ancestor of x_j, and x_j is a descendant of x_i.

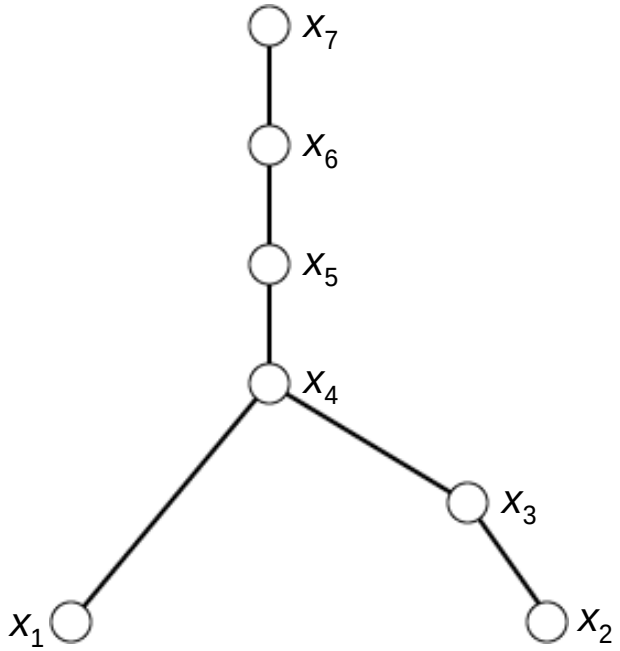
Column structure

- Lemma:

  Suppose that $L_{ij} \neq 0$, with $i > j$. Then x_i is a ancestor of x_j, and x_j is a descendant of x_i in $T(A)$.

- The converse does not hold.

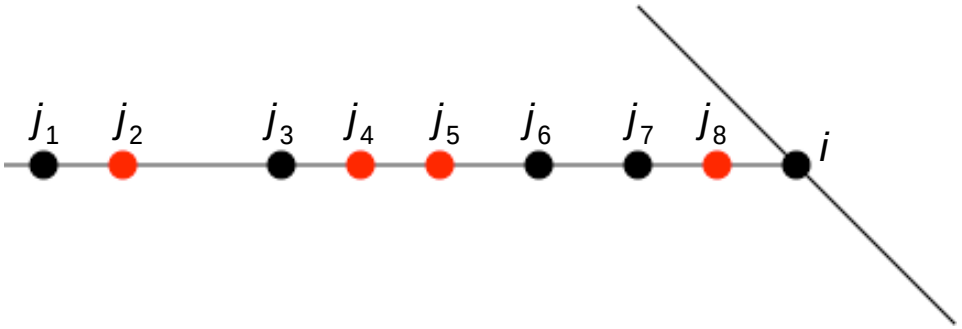


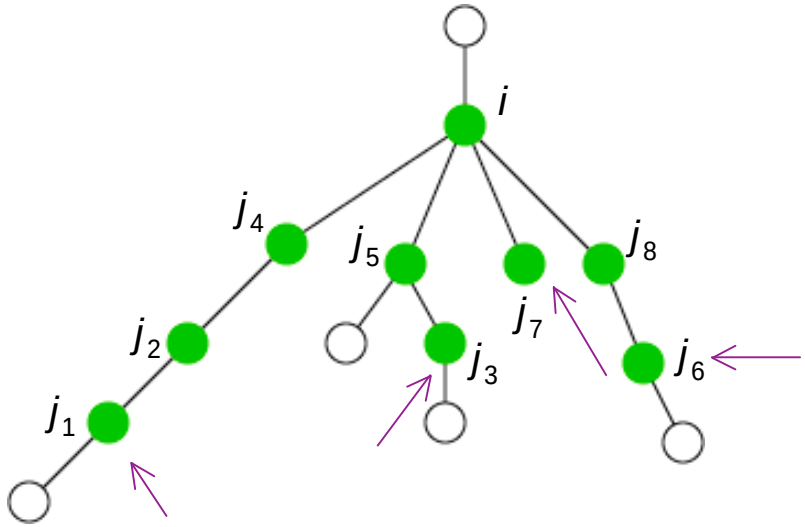


Row structure

- Theorem [Liu ('86)]:

  Let $T(A)$ be the elimination tree of A and L be the Cholesky factor. Let $i > j$. Then $L_{ij} \neq 0$ if and only if x_j is an ancestor of some vertex x_k in $T(A)$ such that $A_{ik} \neq 0$.





Row subtrees

- The nonzero elements in row *i* of the lower triangular part of *A* define a subtree of T(A).
  - Called the i^{th} row subtree [Liu ('86)].

- The "leaves" of the subtree correspond to some of the off-diagonal nonzero elements in row *i* of the lower triangular part of *A*.
- The root of the row subtree correspond to the diagonal element.
- All vertices of the row subtree correspond to nonzero elements in row *i* of *L*.

Elimination tree

- The elimination tree is a powerful tool in sparse Cholesky factorization.
- There is one technical challenge that has not been discussed.
  - What is it?

Computing the elimination tree

- The elimination tree $T(A)$ is defined in terms of the structure of L.
  - How useful is $T(A)$?

- However, $T(A)$ can be computed using only the structure of A.
- The algorithm has complexity "almost linear" in the number of nonzero elements in A.
  - $O(|A|\ \alpha(|A|,n))$, where $\alpha(|A|,n)$ is the inverse of Ackermann's function.
    - For most m and n, $\alpha(m,n) < 4$.

Computing the elimination tree

- The elimination tree of a 1 by 1 matrix is trivial.

- Suppose that the elimination tree $T(B)$ of an $(n-1)$ by $(n-1)$ matrix B is known.
  - $T(B)$ has $n-1$ vertices: x_1 , x_2 , …, x_{n-1}.

- We want to construct the elimination of an n by n matrix A:

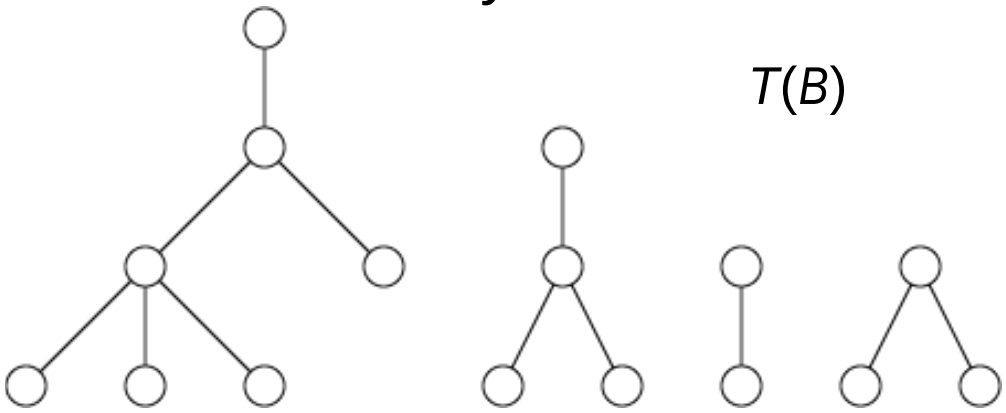
$$A = \begin{bmatrix} B & u \\ u^T & \beta \end{bmatrix}$$

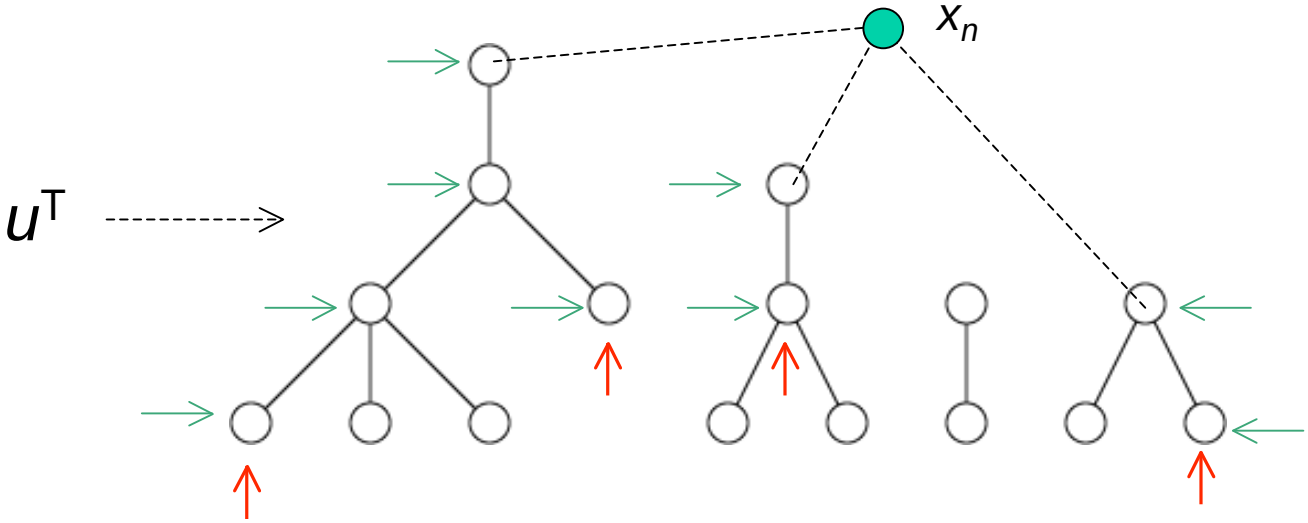
  - How do we add x_n to $T(B)$ to produce $T(A)$?

Computing the elimination tree

- How do we add x_n to $T(B)$ to produce $T(A)$?
  - The definition of the n^{th} row subtree is the key.

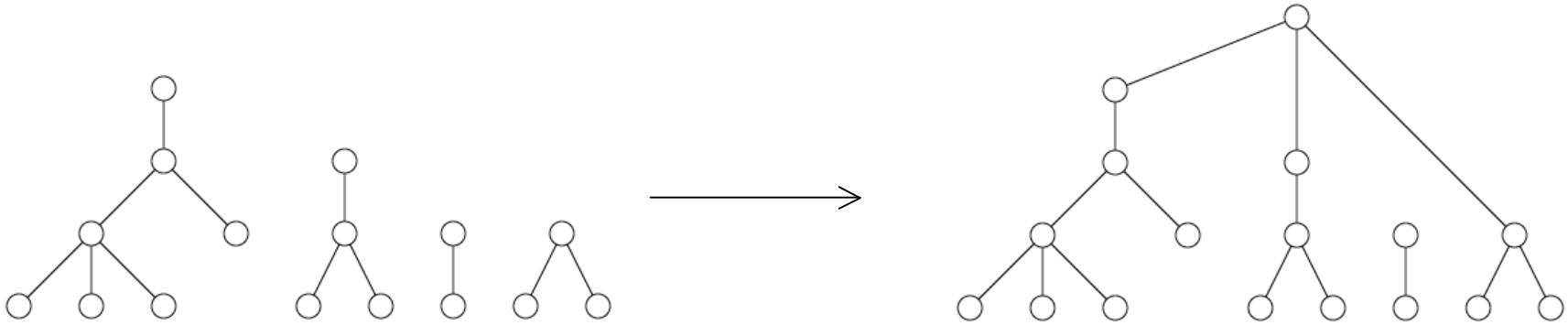
$$A = \begin{bmatrix} B & u \\ u^T & \beta \end{bmatrix}$$





Computing the elimination tree

- To obtain $T(A)$ from $T(B)$, paths in $T(B)$ have to be traversed.
  - Number of edges traversed is at least $O(|L_{n*}|)$.
- Overall complexity is at least $O(L)$.
  - Not desirable.



- To speed up the algorithm, for each vertex, keep track of the root the subtree to which the vertex belongs.

Computing the elimination tree

- To speed up the algorithm, for each vertex, keep track of the root the current subtree to which the vertex belongs.
  - This is Tarjan's disjoint set union operations [Tarjan ('75)].
    - Can be implemented in $O(|A| \alpha(|A|,n))$ time.
  - This requires a temporary array of size n (in addition to the array for storing the elimination tree - i.e., "First" values).

- The complexity is realizable.

Summary …

- Elimination tree.
  - What it is.
  - Why it is important.
  - How it is computed.

- Symbolic factorization.
  - What it means.
  - Why it is important.
  - How the structure of column i of L depends on every column j of L, where x_j is a child of x_i in the elimination tree.

Symbolic factorization

- Output from symbolic factorization.
  - The sparsity structure of L.
- As we discussed previously, the nonzero elements of L can be using a compressed column storage scheme:
  - values: values of nonzero elements of L, arranged column by column.
  - rowindx: corresponding row subscripts.
  - colptr: pointers to beginning of compressed columns.
- This requires $O(n+|L|)$ integer locations and $O(|L|)$ floating-point locations.
  - Can do better!

Postordering elimination tree

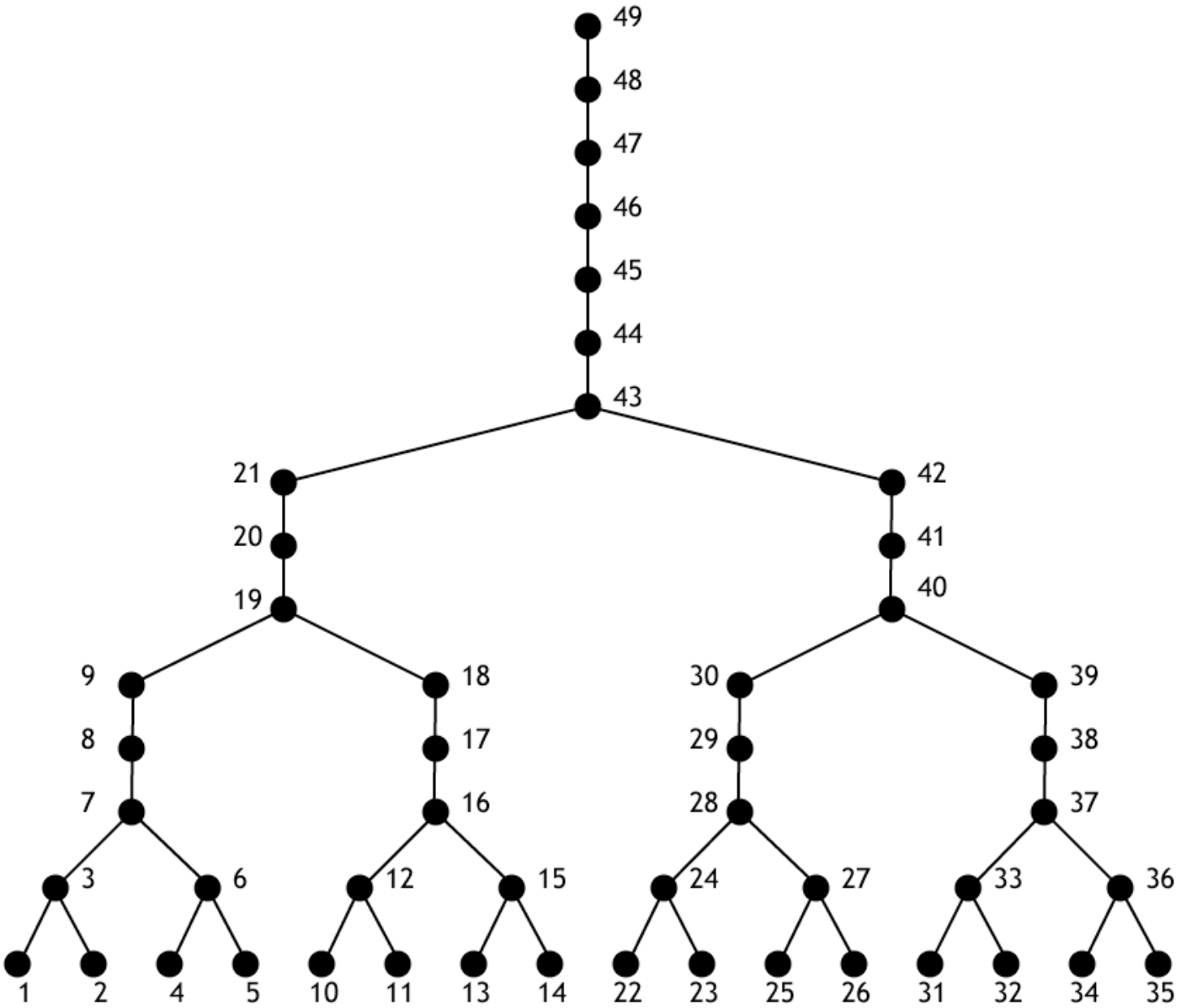
- Given a SPD matrix A and its elimination tree $T(A)$.
- The labeling of vertices in $T(A)$ depends on A (i.e., the sparsity structure of A).
- A postordering of $T(A)$ is a change of the labeling of the vertices so that vertices in each subtree are labeled consecutively before the root of the subtree is labeled.
- A postordering of $T(A)$ corresponds to a symmetric permutation P of the rows and columns of A.

Postordering elimination tree

- A postordering of $T(A)$ corresponds to a symmetric permutation P of the rows and columns of A.
- Theorem [Liu ('86)]:

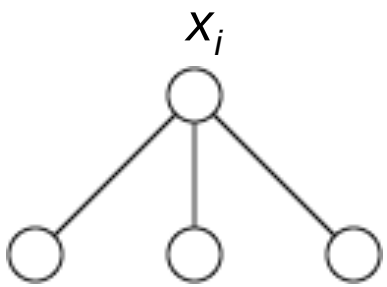
  The Cholesky factors of A and PAP^T have identical number of nonzero elements.
- That is, a postordering of $T(A)$ gives an isomorphic ordering.
- Many advantages for having a postordering of $T(A)$.
- Exercise: How to compute a postordering?

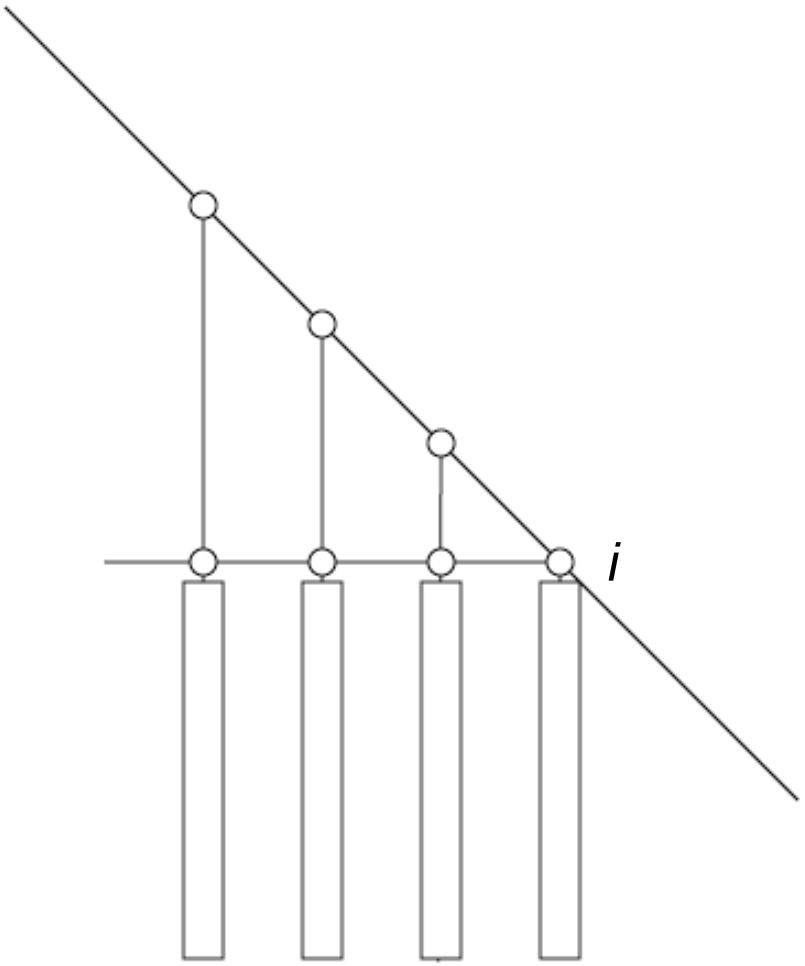
Postordered elimination tree



Simple observations on the elimination tree …

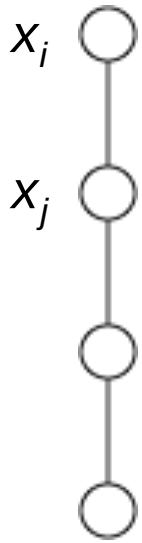
- If x_i has more than one child …

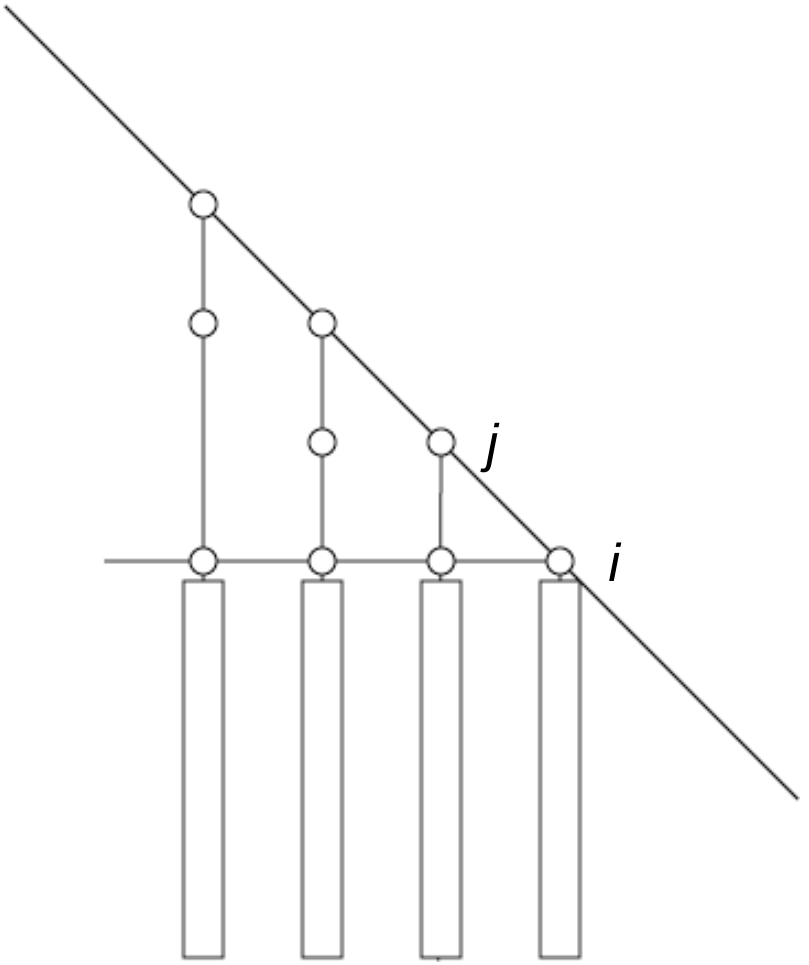




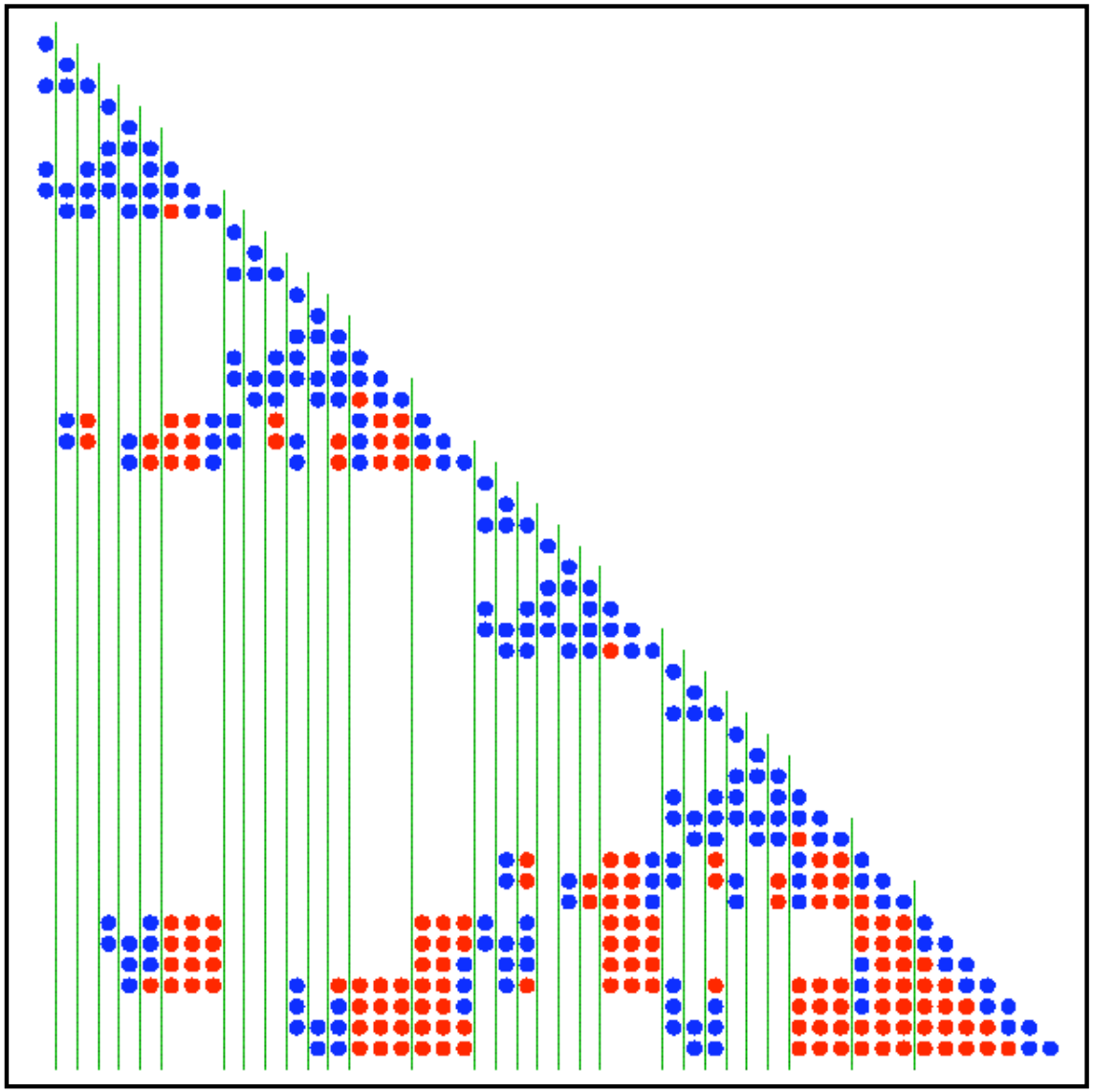
Simple observations on the elimination tree …

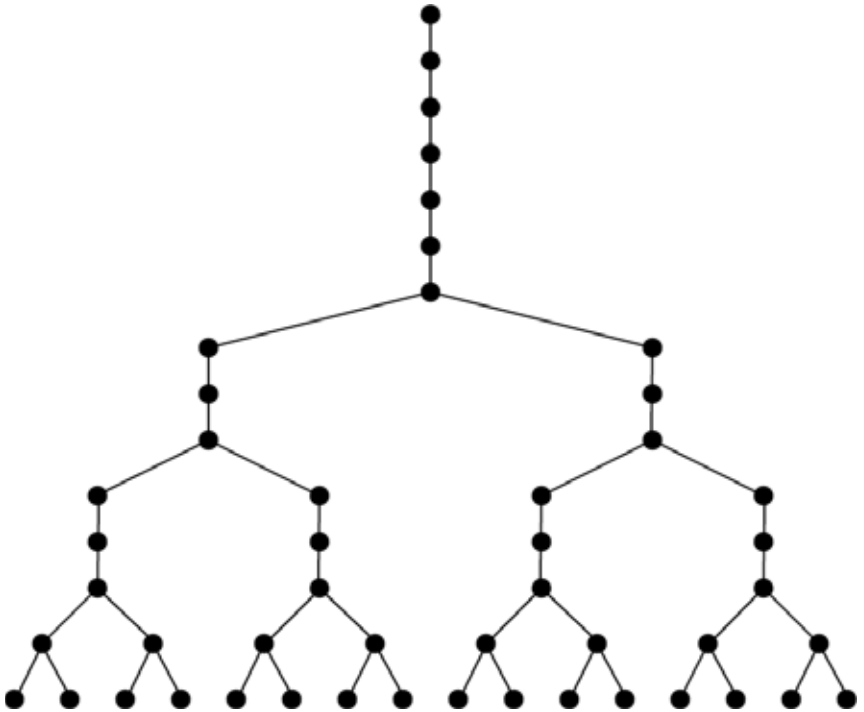
- If x_i has exactly one child …





Elimination tree





Columns associated with some "chains" have identical structure, but columns associated with other "chains" do not have identical structure.

Effect of fill

- Consider the right-looking formulation of Cholesky factorization.
- Once a column is eliminated, it is used to update columns to its right, causing fill to occur.

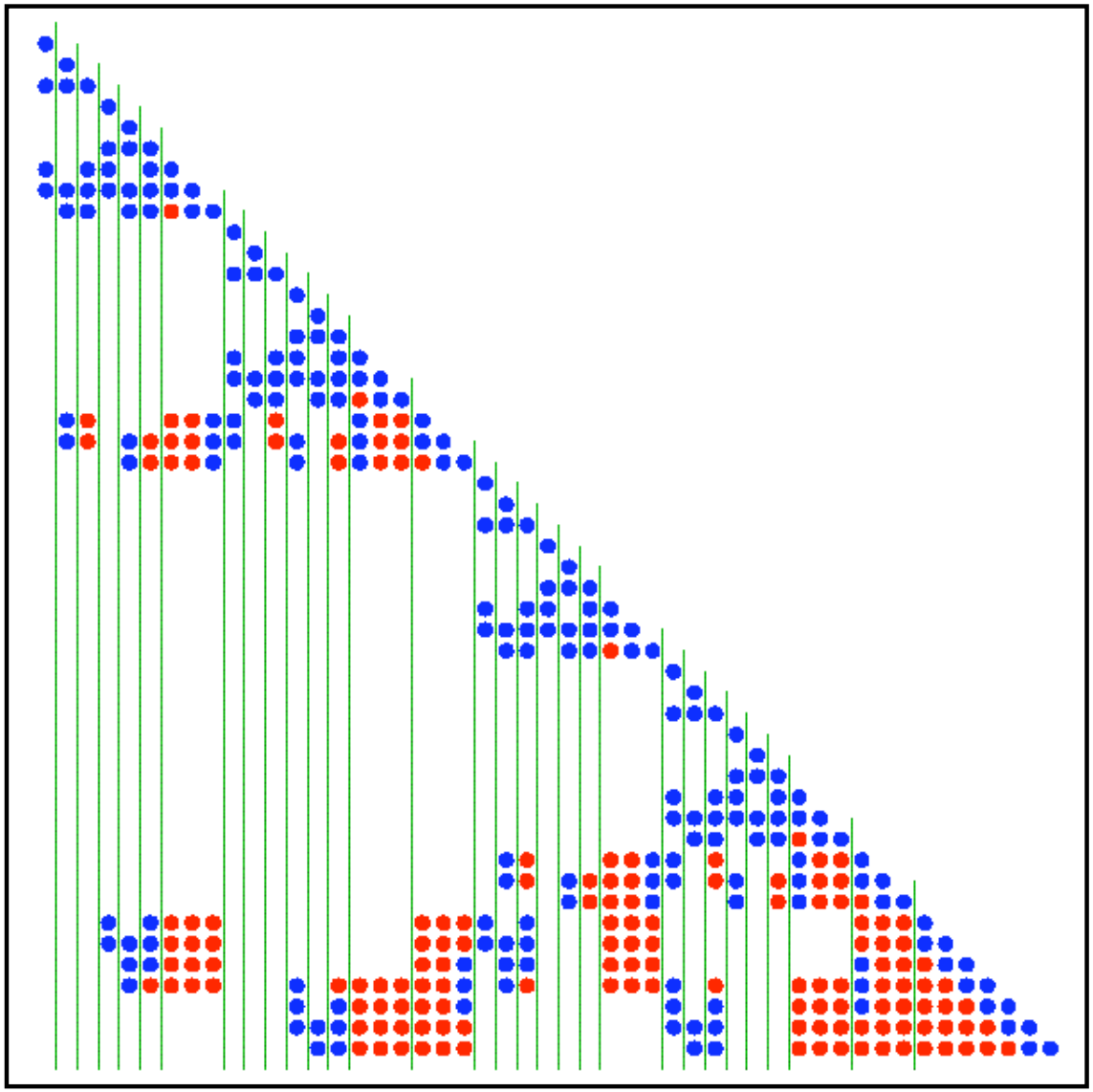
- As the elimination proceeds, the effect of fill "propagates" from left to right.
- Consequently, the structure of L tends to be "richer" towards the last column.

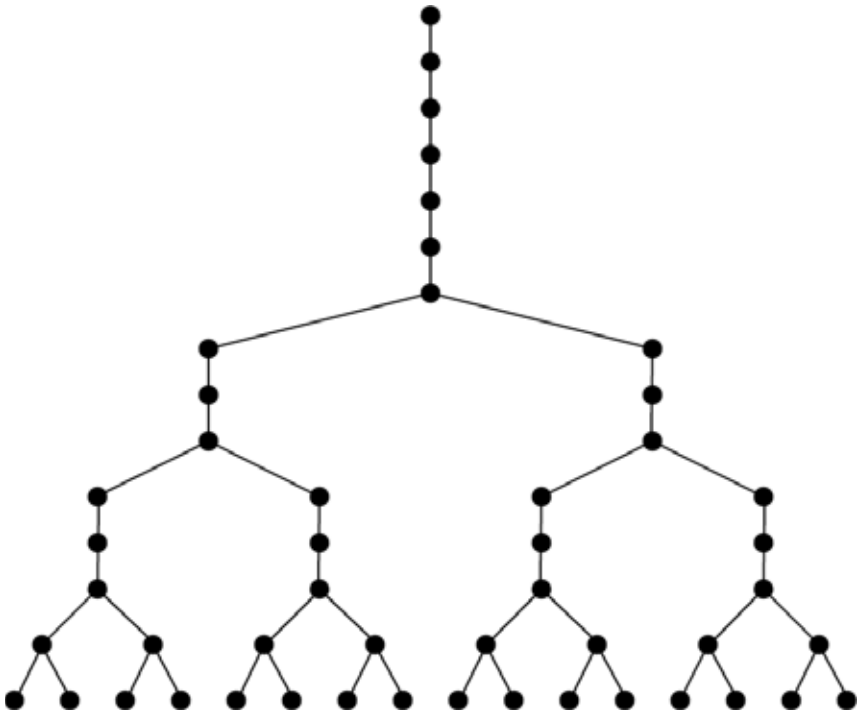
Dense blocks in sparse Cholesky factor

- It is often the case that consecutive columns in L share essentially identical sparsity pattern.
- Such a group of columns is referred to as a supernode.
  - A supernode in L is a group of consecutive columns $\{j, j+1, \ldots, j+t-1\}$ such that
    - columns j to $j+t-1$ have a dense diagonal block, and
    - columns j to $j+t-1$ have identical sparsity structure below row $j+t-1$.

- A postordering of the elimination tree will give the longest possible sets of consecutively labeled columns.

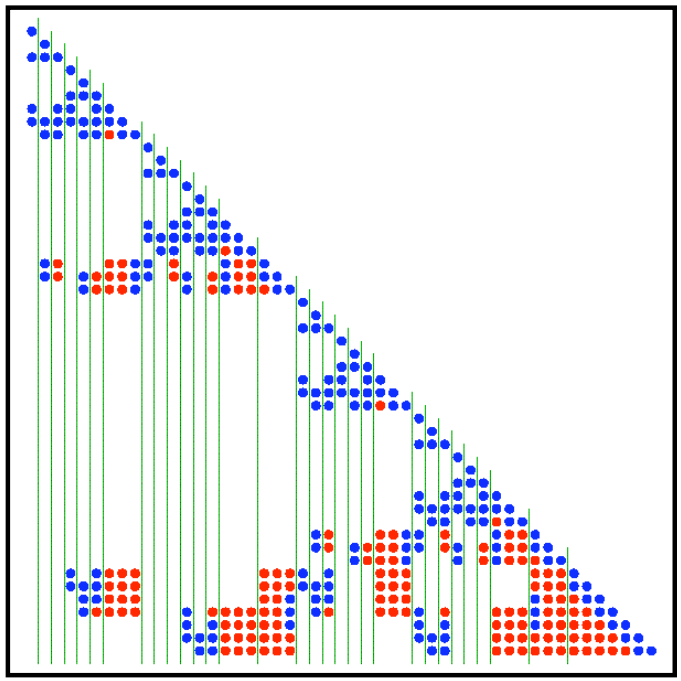
Dense blocks in sparse Cholesky factor

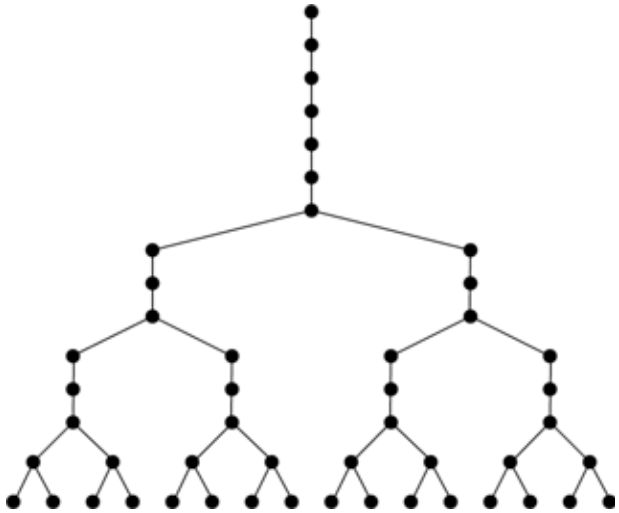




Supernodes in sparse Cholesky factor

- Fundamental supernodes [Liu, Ng, Peyton ('93)]:
  - A fundamental supernode is a set of consecutive columns $\{j, j+1, \ldots, j+t-1\}$ such that x_{j+s} is the only child of x_{j+s+1} in the elimination tree, for $0 \le s \le t-2$.
- That is, the columns in a fundamental supernode must form a simple chain in the elimination tree.



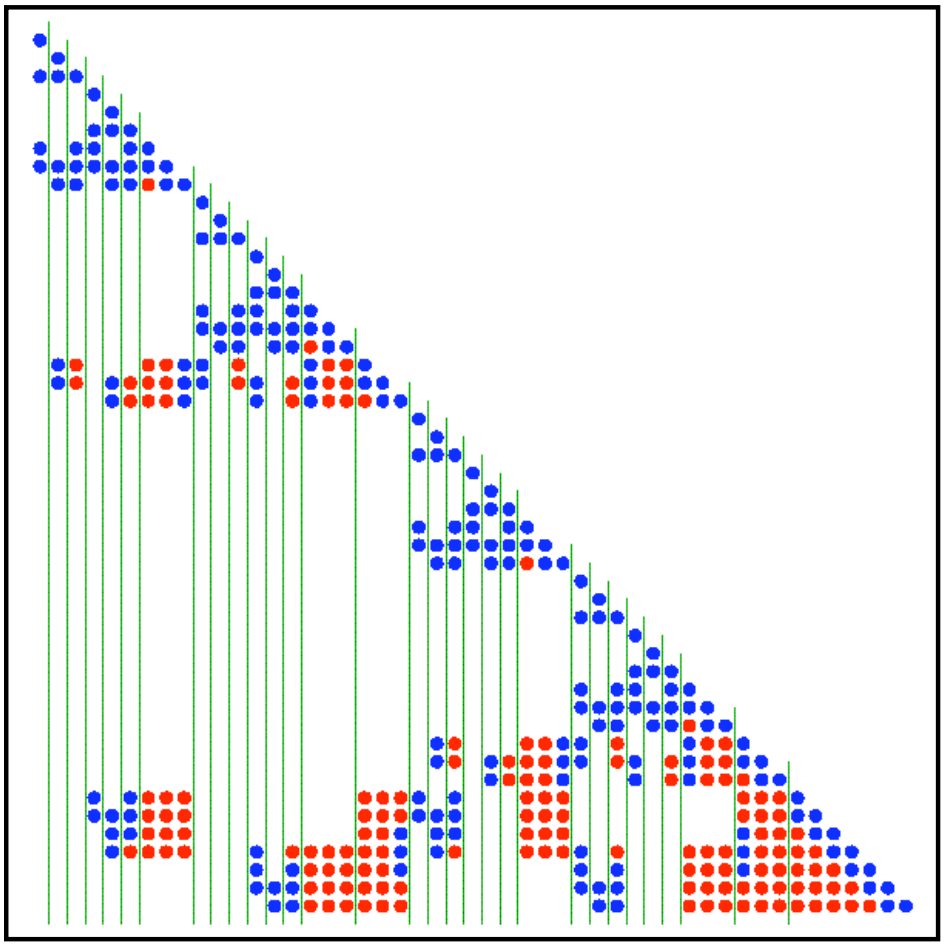


Fundamental supernodes

- Theorem [Liu, Ng, Peyton ('93)]:
  - Column j is the first column in a fundamental supernode if and only if x_j has two or more children in the elimination tree, or x_j is a leaf vertex of some row subtree of the elimination tree.

Supernodes in sparse Cholesky factor

- Supernodes provide a natural way to partition the columns of L.
  - Fundamental supernodes are for convenience.
- An important application of supernodes
  - Representation of the sparsity structure of L.



Data structure for sparse Cholesky factor

- Exploiting supernodes in the representation of the sparsity structure of L.
  - Need only one set of row indices for all columns in a supernode.
    - That is, row indices of the nonzero elements in the first column of each supernode.
- A typical data structure …
  - values: values of nonzero elements of L, arranged column by column.
  - colptr: pointers to beginning of compressed columns.
  - lindx: row indices of nonzero elements in the first column of each supernode.
  - xlindx: pointers to beginning of row indices for each supernode (or column).

Data structure for sparse Cholesky factor

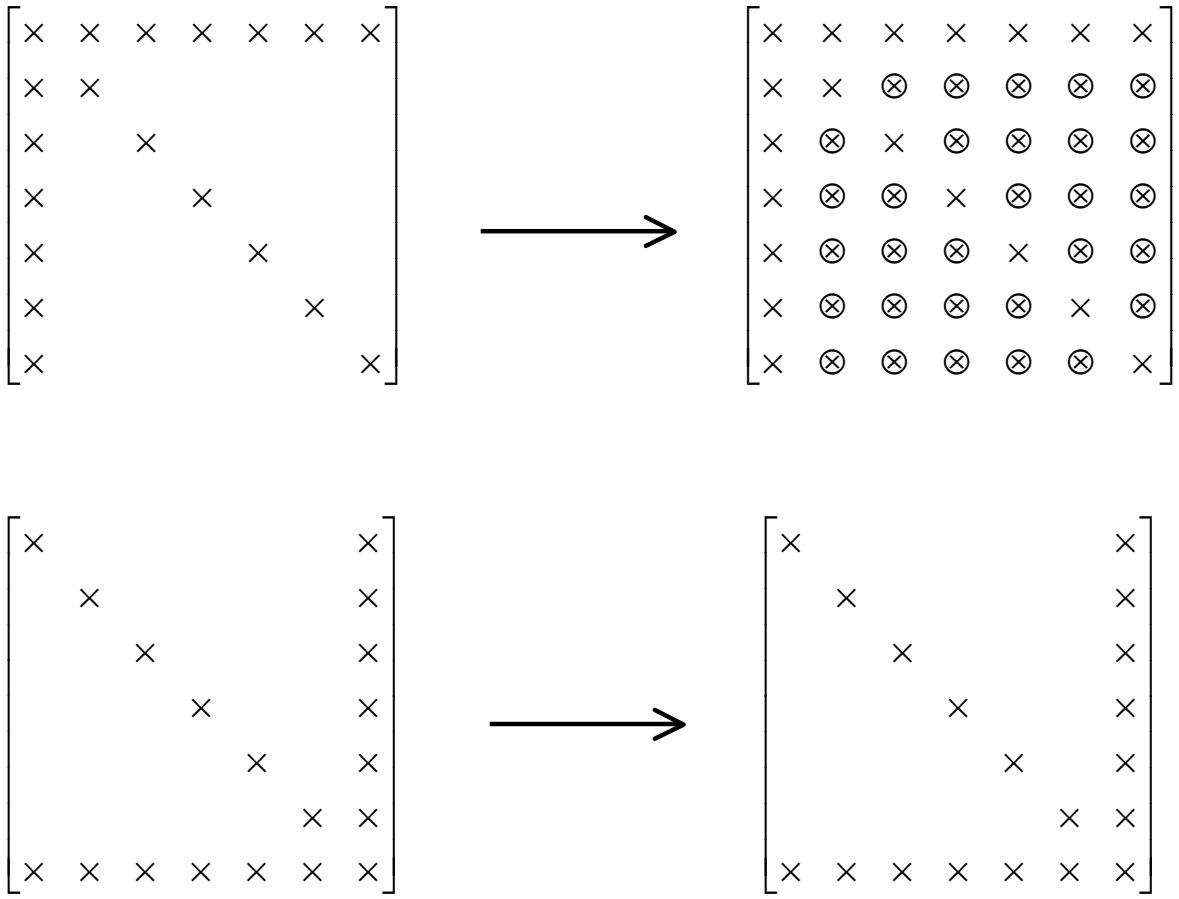
- L - Cholesky factor.
- Γ - a rectangular matrix containing just the first column of each supernode.
  - values: $|L|$
  - colptr: n
  - lindx: $|\Gamma|$
  - xlindx: number of columns in Γ (or n).
- Important observation: $|\Gamma| << |L|$.

Summary …

- Postordering of elimination tree.
  - What it is.
- Supernodes.
  - What they are.
  - Impact of postordering.
  - Compact representation of the sparsity structure of L.

Effect of ordering

- Fact: Different arrangements (or orderings) of the rows and columns of a SPD matrix will result in different sparsity structure in the Cholesky factor.



Effect of ordering

- A more precise description:
  - The amount of fill depends on the "structure" of A.
  - Can change the structure by permuting the rows and columns of A.

- The ordering problem: Find "good" permutations to reduce fill in the Cholesky factor of A.

The ordering problem

- In graph-theoretic terminology: Find a labelling of the vertices so that the elimination of the vertices according to the labelling will reduce the number of fill edges in the filled graph of A.
  - A labelling of the vertices corresponds to a symmetric permutation of A.

- The ordering problem:
  - Combinatorial in nature (n! choices).
  - NP-complete [Yannakakis ('81)].
  - Rely heavily on heuristic algorithms.

The fill path theorem

- Fill Path Theorem:

  Let $G = (X,E)$ be the graph of a SPD matrix A. Denote the corresponding filled graph by $G^+ = (X^+,E^+)$. Then $\{x_i,x_j\} \in E^+$ if and only if there is a path $(x_i,x_{p_1},x_{p_2},\ldots,x_{p_s},x_j)$ in G such that $p_k < \min\{i,j\}$, for $1 \le k \le s$.

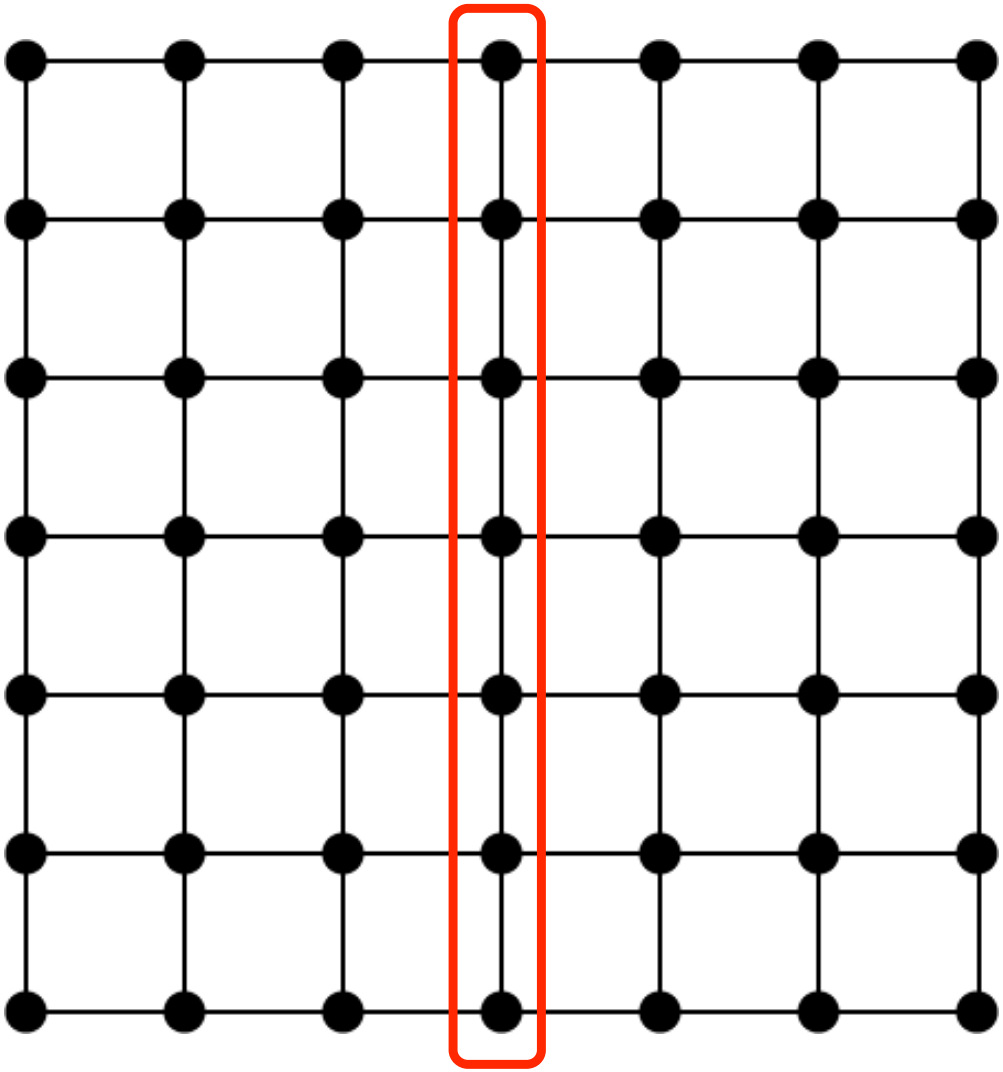
- The "fill path theorem" provides a heuristic way of labeling the vertices in the graph of A to reduce fill:
  - Label the vertices so that all paths joining any two vertices do not satisfy the fill path theorem.

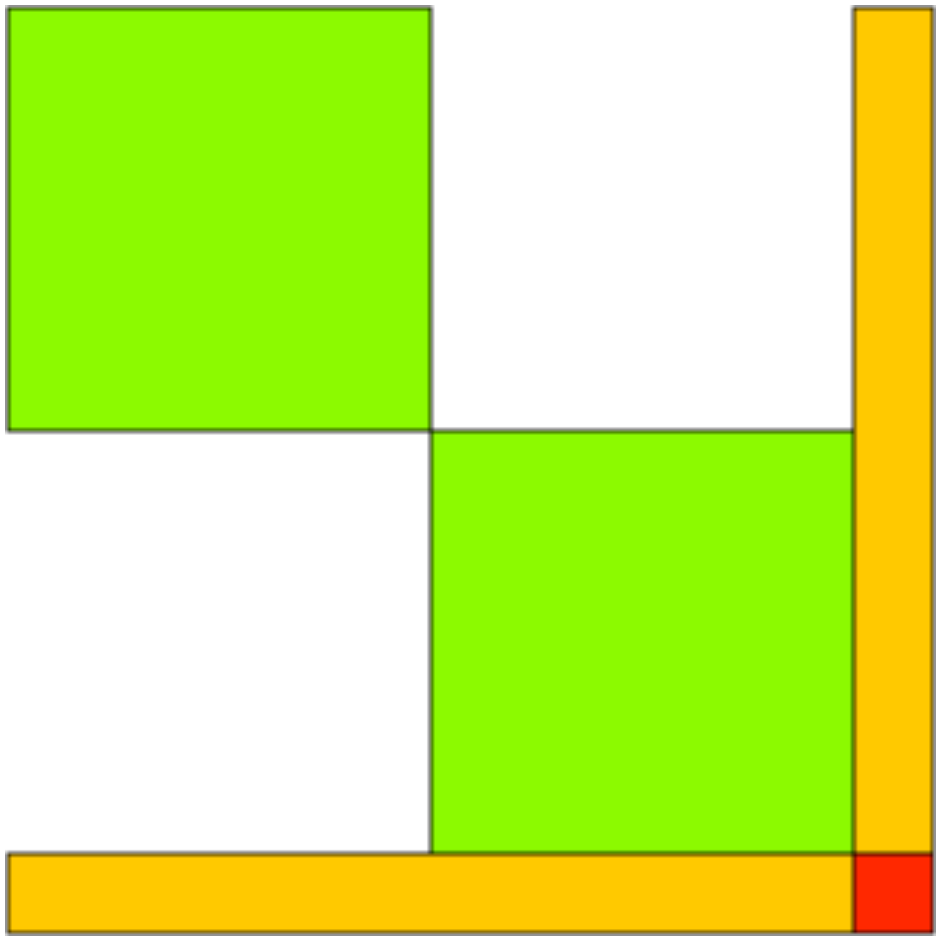
Graph separators and fill

- Given a graph *G*.
- Find a set of vertices *S* such that the removal of *S*, together with all incident edges, partitions *G* into 2 or more pieces (say, 2).
- *S* is called a separator.

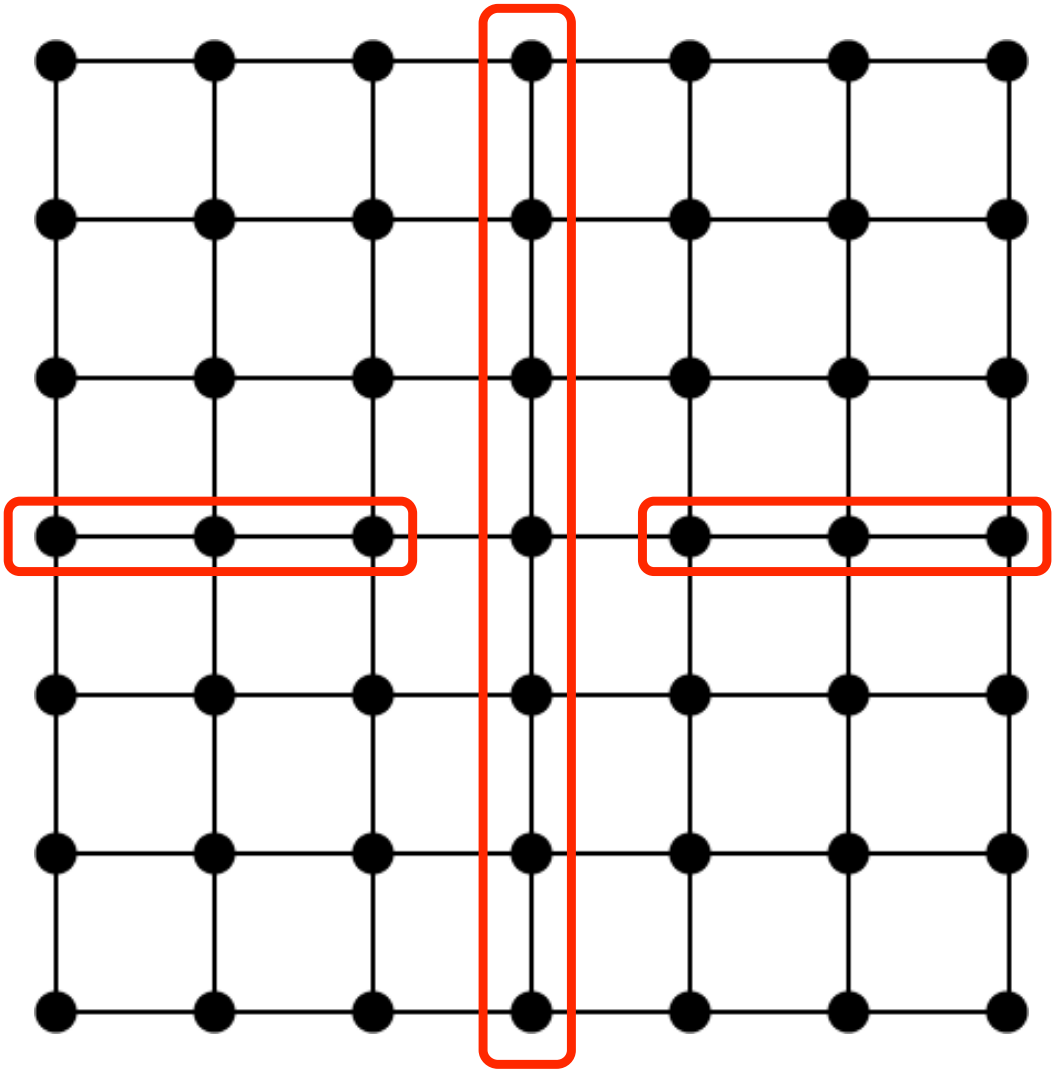
- Label the vertices in the two pieces first, followed by those of *S*.

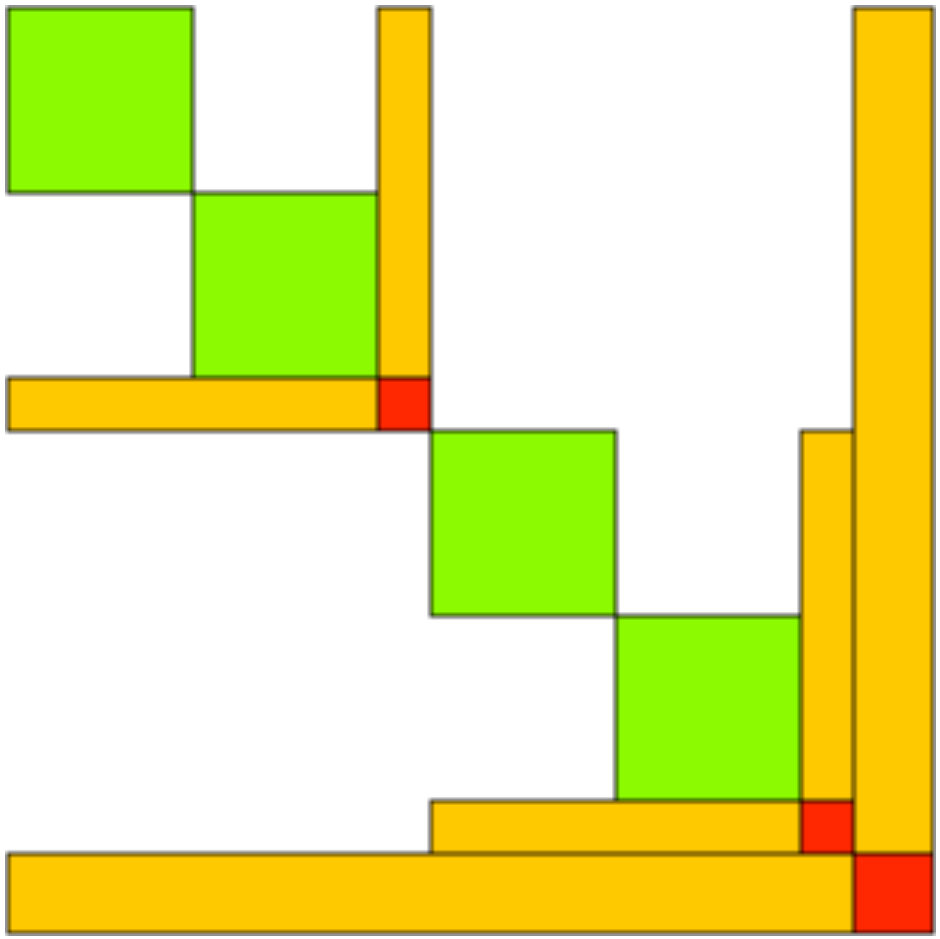
Separators and fill





Separators and fill



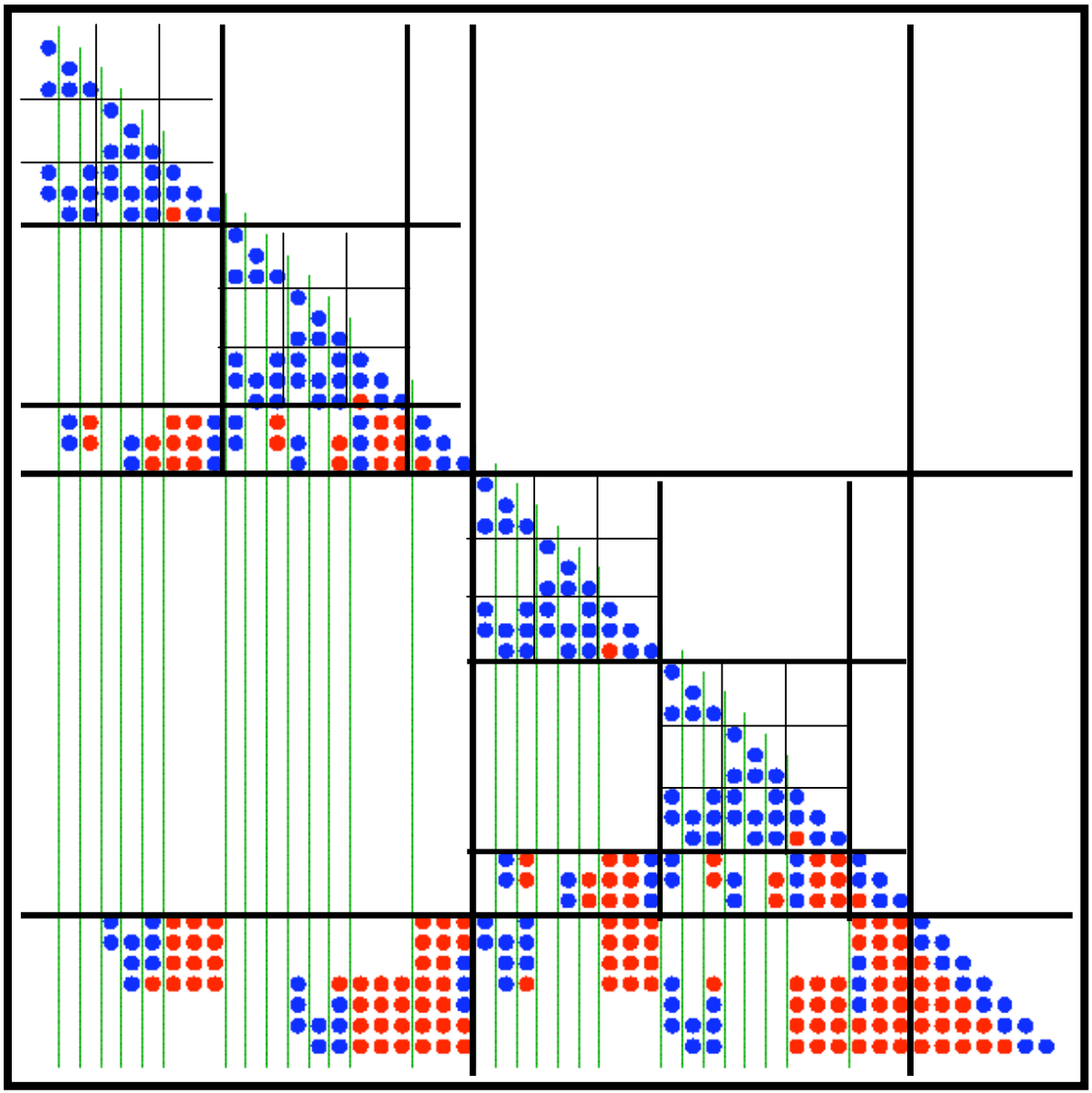


Nested dissection

- If the heuristic is applied recursively to the graph, one obtains the so-called nested dissection ordering.
- [George ('73)].

- A digestion …
  - Remember AMLS?
  - Nested dissection order is ideal for the AMLS algorithm.

The nested dissection ordering



Nested dissection ordering on a 7 by 7 grid, with blue dots representing the original nonzero elements and red dots representing the fill elements.

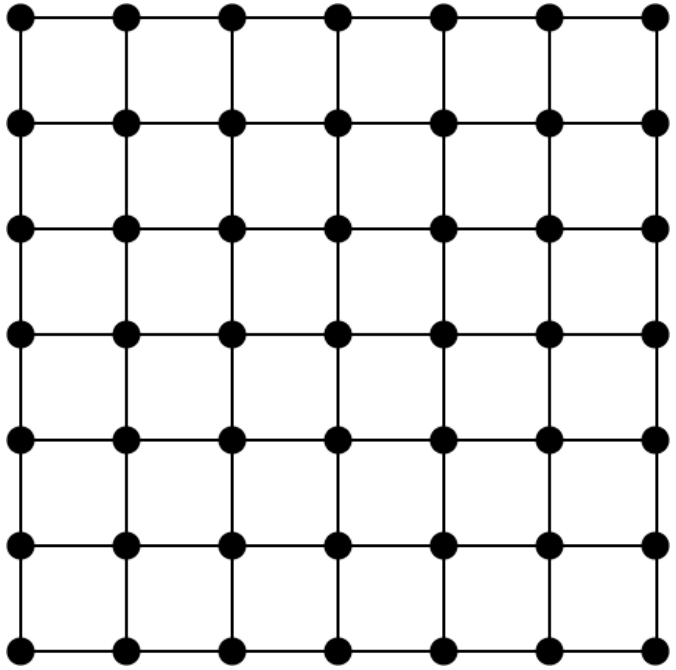
The nested dissection orderings

- Nested dissection is a top-down algorithm.
  - Labeling the last set of columns first.
  - Generation of the separator requires a global view of the graph.

- General graphs:
  - Needs heuristic algorithms to find separators and generate nested dissection orderings [George & Liu ('78)].
  - Quality of orderings depends on choice of separators.
    - New implementations of nested dissection are based on more sophisticated graph partitioning techniques.
      - [Pothen, Simon, Wang ('92); Hendrickson, Leland ('93); Hendrickson, Rothberg ('96); Gupta, Karypis, Kumar ('96,'97); Ashcraft, Liu ('96,'97); …]

Complexity of nested dissection orderings

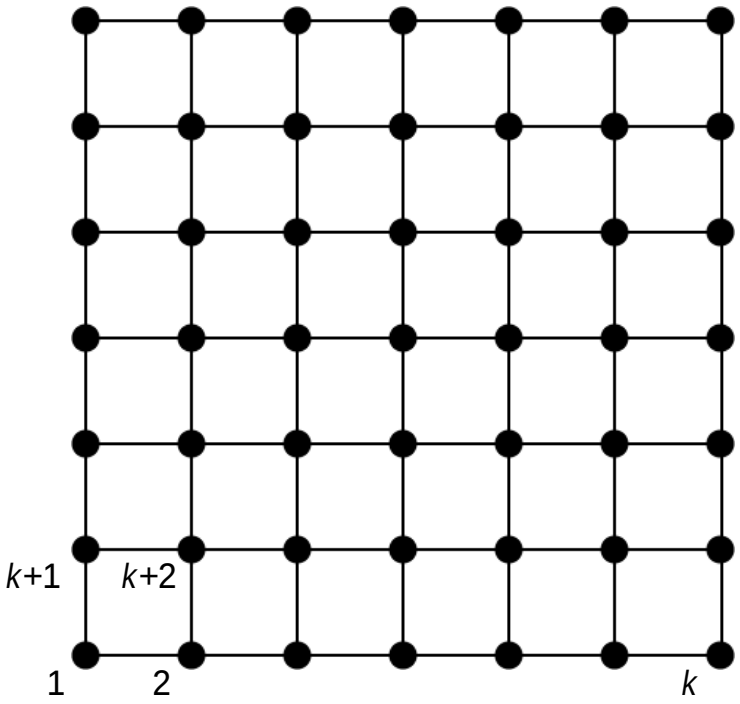
- Consider a nested dissection ordering for a k^2 by k^2 matrix defined on a k by k finite element and finite difference grids.

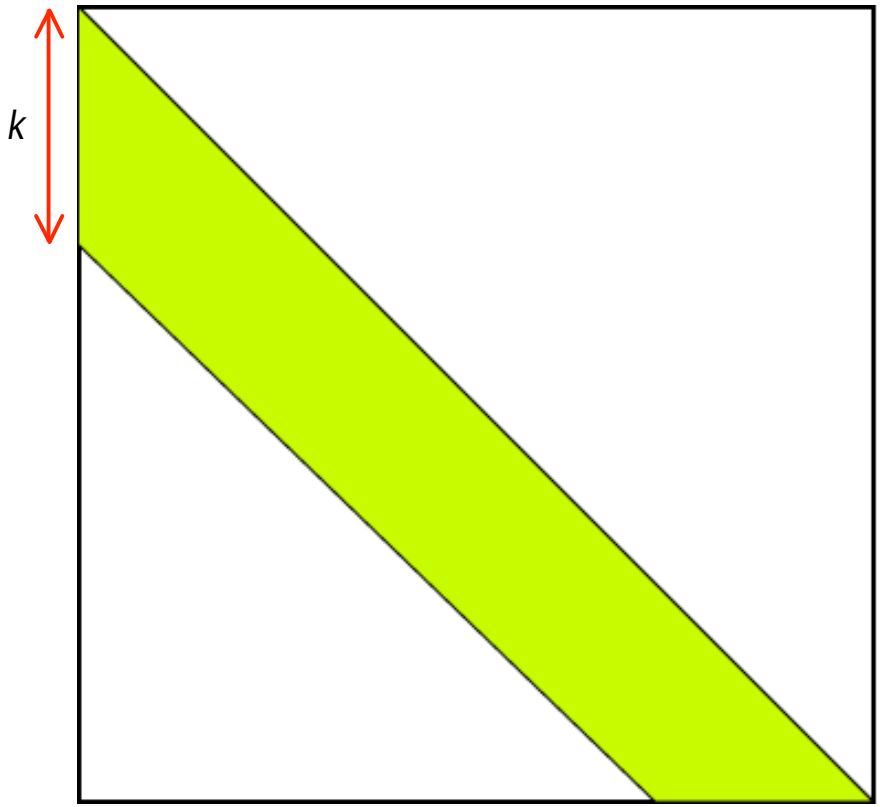


- The number of operations required to apply Gaussian elimination to the permuted matrix is $O(k^3)$.
- The number of nonzero elements in the corresponding Cholesky factor is $O(k^2 \log k)$.
- [George ('73)].

A digression ...

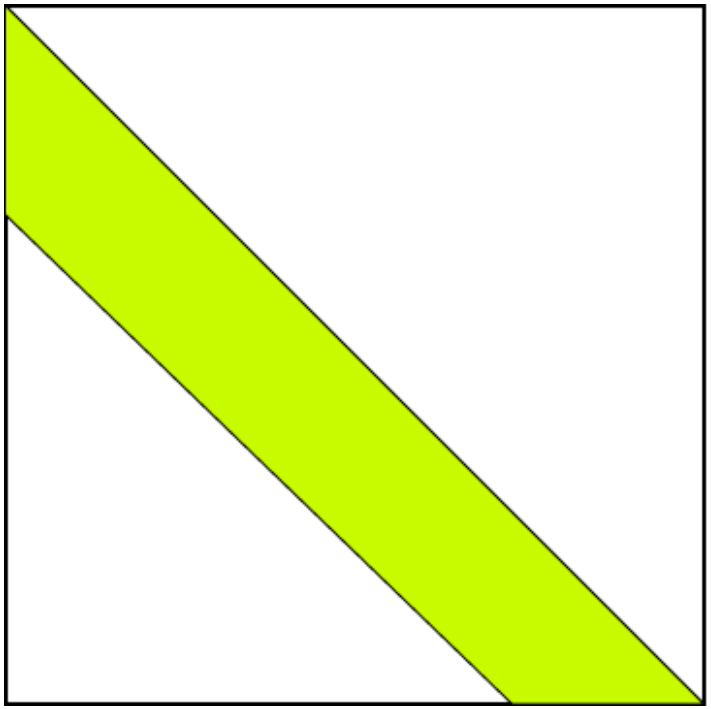
- A convenient way to solve the linear system associated with the *k* by *k* grid is to use a band solver.
  - The grid points are labeled row by row.





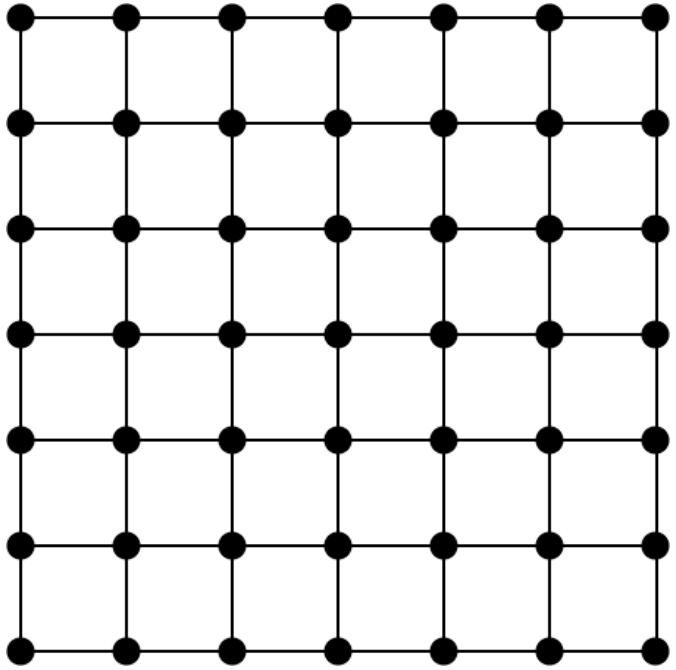
A digression …

- For each column, one step of dense Cholesky factorization has to be applied to a k by k submatrix.
- The number of operations required is $O(k^2)$ per column.
- Over the k^2 columns, the total number of operations required is $O(k^4)$.
- The number of elements that have to be stored is $O(k^3)$.
- So, more sophisticated approach is needed to generate better labelling.



Complexity of nested dissection orderings

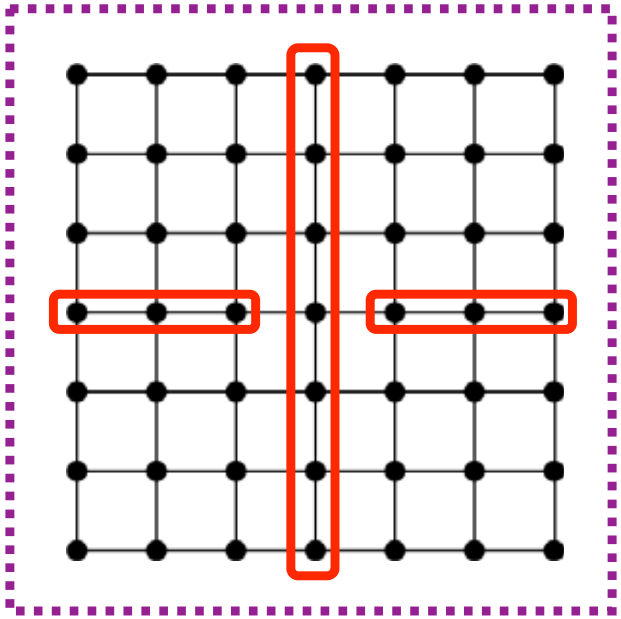
- Consider a nested dissection ordering for a k^2 by k^2 matrix defined on a k by k finite element and finite difference grids.



- The number of operations required to apply Gaussian elimination to the permuted matrix is $O(k^3)$.
- The number of nonzero elements in the corresponding Cholesky factor is $O(k^2 \log k)$.
  - Will prove this …

Complexity of nested dissection orderings

- Proving the complexity of nested dissection on k by k meshes.
  - For convenience, assume that the mesh is surrounded by a separator (to get a simple recurrence equation).
  - Counting only nonzero elements.
    - Similar approach for counting operations.
  - Let Fill(k) denote the number of nonzero elements in the Cholesky factor of the permuted matrix associated with a k by k mesh surrounded by a separator.
  - Let $\theta(k)$ be the number of nonzero associated with a separator of size k.



Complexity of nested dissection orderings

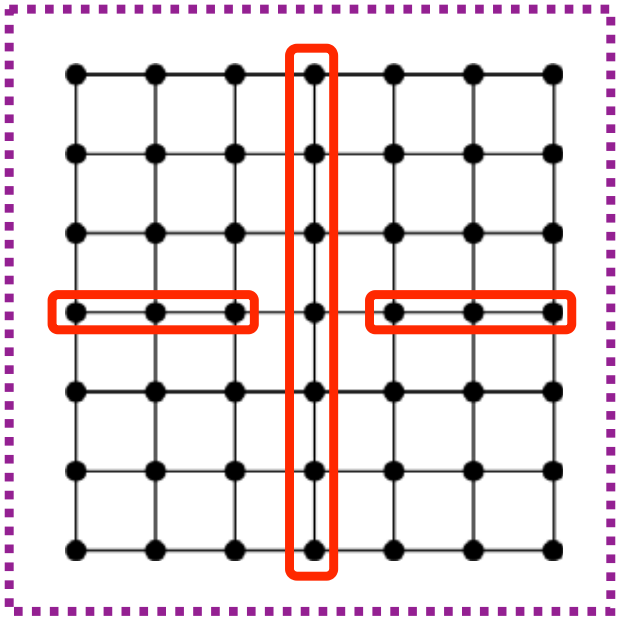
- Calculating fill ...

$$\text{Fill}(k) = 4\ \text{Fill}(k/2) + 2\left[\theta(k/2)\right] + \theta(k)$$

$$\theta(k/2) = \sum_{i=1}^{k/2}\left[2k + 2(k/2) + i\right] = 13k^2/8 + O(k)$$

$$\theta(k) = \sum_{i=1}^{k}\left[4k + i\right] = 9k^2/2 + O(k)$$

$$\text{Fill}(k) = 4\ \text{Fill}(k/2) + 31k^2/4 + O(k)$$



Complexity of nested dissection orderings

- Calculating fill - excluding lower order terms …

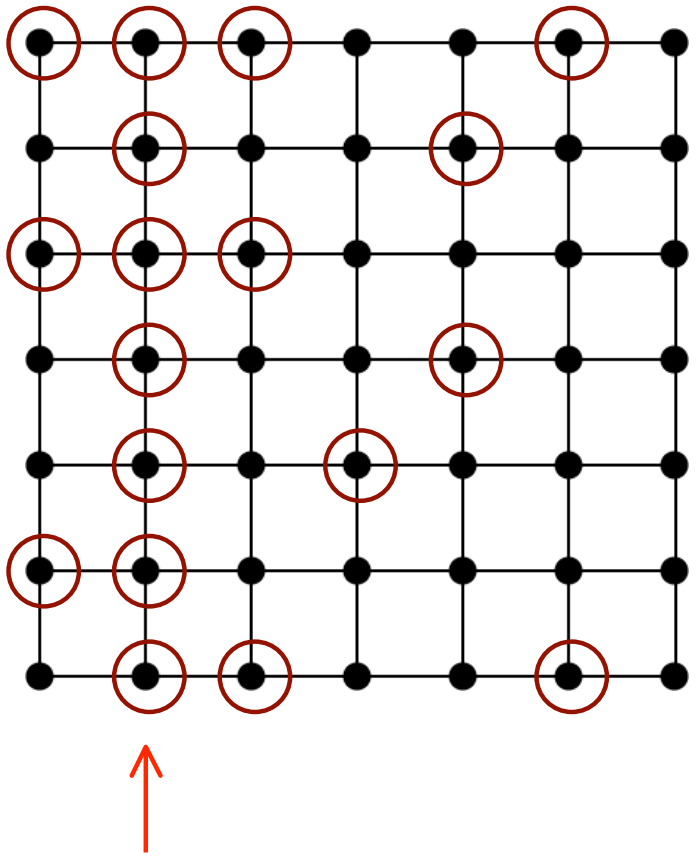
$$\begin{aligned}
\text{Fill}(k) &= 4\ \text{Fill}(k/2) + (31/4)k^2 \\
&= 4\left[\,4\ \text{Fill}(k/2^2) + (31/4)(k/2)^2\,\right] + (31/4)k^2 \\
&= 4^2\ \text{Fill}(k/2^2) + 4(31/4)(k/2)^2 + (31/4)k^2 \\
&= 4^2\left[\,4\ \text{Fill}(k/2^3) + (31/4)(k/2^2)^2\,\right] + 4(31/4)(k/2)^2 + (31/4)k^2 \\
&= 4^3\ \text{Fill}(k/2^3) + 4^2(31/4)(k/2^2)^2 + 4(31/4)(k/2)^2 + (31/4)k^2 \\
&= \sum_{i=0}^{\log_2 k}\left[\,4^i\,(31/4)(k/2^i)^2\,\right] \\
&= \sum_{i=0}^{\log_2 k}(31/4)k^2 \\
&= (31/4)k^2\log_2 k
\end{aligned}$$

Lower bound complexity

- For matrices defined on k by k finite element and finite difference meshes:
  - Number of nonzero elements in the Cholesky factor $\geq O(k^2 \log k)$
  - Number of operations required to compute the Cholesky factor $\geq O(k^3)$
    - Will prove this …
  - Hoffman, Martin, Rose ('73); George ('73).
- These are lower bounds and independent of how the matrices are permuted (or the meshes are labelled).
- Conclusion: Nested dissection orderings on k by k meshes are optimal asymptotically.

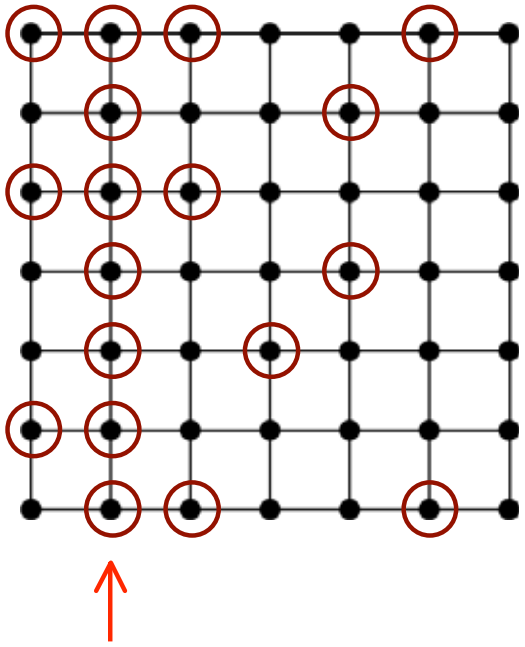
Lower bound complexity

- Proving the lower bound on operations …
  - Consider eliminating the mesh points in some order.
  - Consider the moment when an entire horizontal or vertical mesh line is eliminated.
  - Suppose it is a vertical mesh line.
  - Each horizontal mesh line (perhaps with the exception of one) has at least one mesh point that has not been eliminated.
    - There is a fill path from this mesh point to another uneliminated mesh point on another horizontal mesh line.



Lower bound complexity

- Proving the lower bound on operations …
  - Each horizontal mesh line (perhaps with the exception of one) has at least one mesh point that has not been eliminated.
    - There is a fill path from this mesh point to at least one uneliminated mesh point on each of the remaining horizontal mesh lines.
    - There are at least $O(k)$ such mesh points.
    - This gives a dense submatrix that needs to be factored.
    - The size of the dense submatrix is at k by k.
    - The number of operations required to factor this submatrix is k^3.
  - So, the number of operations required to compute the Cholesky factor is bounded below by $O(k^3)$.



Optimality of nested dissection orderings

- Nested dissection orderings for matrices defined on k by k finite element and finite difference grids are optimal (asymptotically).
  - Fill = $O(k^2 \log k)$
  - Operations = $O(k^3)$
  - [Hoffman, Martin, Rose ('73); George ('73)].

- For a planar graph with n vertices:
  - $\exists$ separators that have $O(n^{1/2})$ vertices [Lipton, Tarjan ('79)].

    $\Rightarrow$ generalized nested dissection orderings that produce $O(n \log n)$ fill and require $O(n^{3/2})$ operations [Lipton, Rose, Tarjan ('79)].

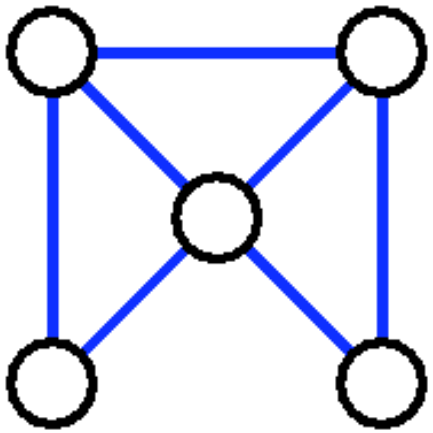
Being greedy ...

- Instead of a top-down approach, one can take a bottom-up view.
  - Start with the graph of the matrix, vertices are eliminated one by one so that a specific metric is minimized.
  - This gives a greedy local heuristic scheme for labeling the vertices.
- It does not guarantee a global minimum, but often works extremely well, particularly for general sparse matrices.

- What metric(s)?

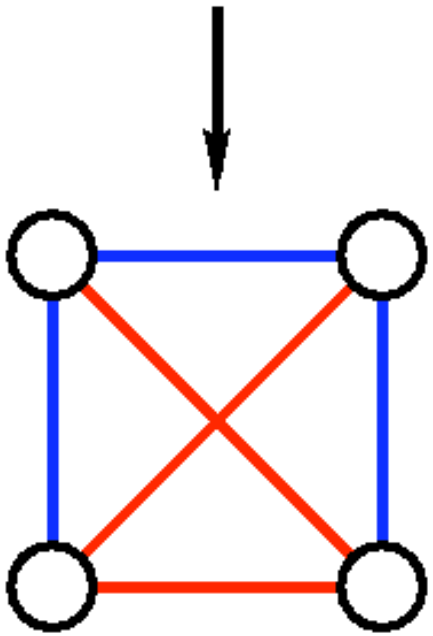
Notion of degree and deficiency

- Degree of vertex *v* is the number of vertices adjacent to *v*.
  - no. of nonzero entries in column/row v of the submatrix remaining to be factored.



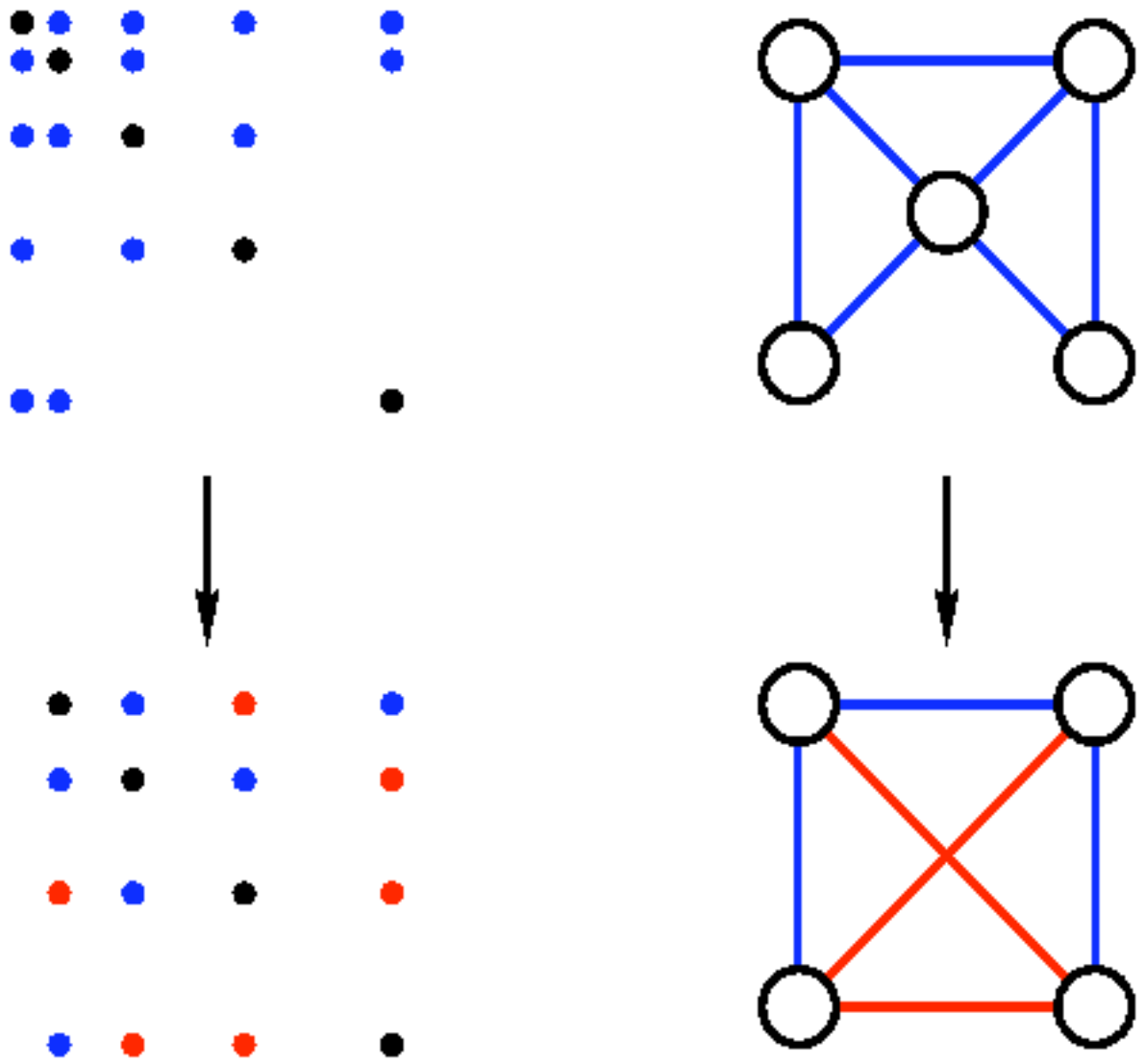
Deg = 4

- The number of edges to be added to the graph when vertex *v* is eliminated is the deficiency of vertex *v*.
  - no. of nonzero entries to be added to lower triangular part of the submatrix remaining to be factored.



Def = 3

Degree and deficiency



The minimum degree algorithm

- Use degree as the metric [Tinney, Walker ('67); Rose ('72)].
  - At each step eliminate the vertex with the minimum degree; break ties arbitrary.
  - This minimizes the number of nonzero entries in the rank-1 update of sparse symmetric Gaussian elimination.

- Facts:
  - Simple heuristic.
  - Very effective in reducing fill.
  - Hard to implement efficiently, but …
  - Little is known about its complexity.

The minimum degree algorithm

- Hard to implement because of dynamic changes in G, but …
- Efficient implementations do exist, requiring $O(|E|)$ space.
  - Based on quotient elimination graphs.
    - [George & Liu (‘80)]
    - [Eisenstat (early ‘80)]
    - [Liu (‘85)]
    - [Amestoy, Davis, Duff (‘94)]

An extreme example

- Little is known about its complexity, but …

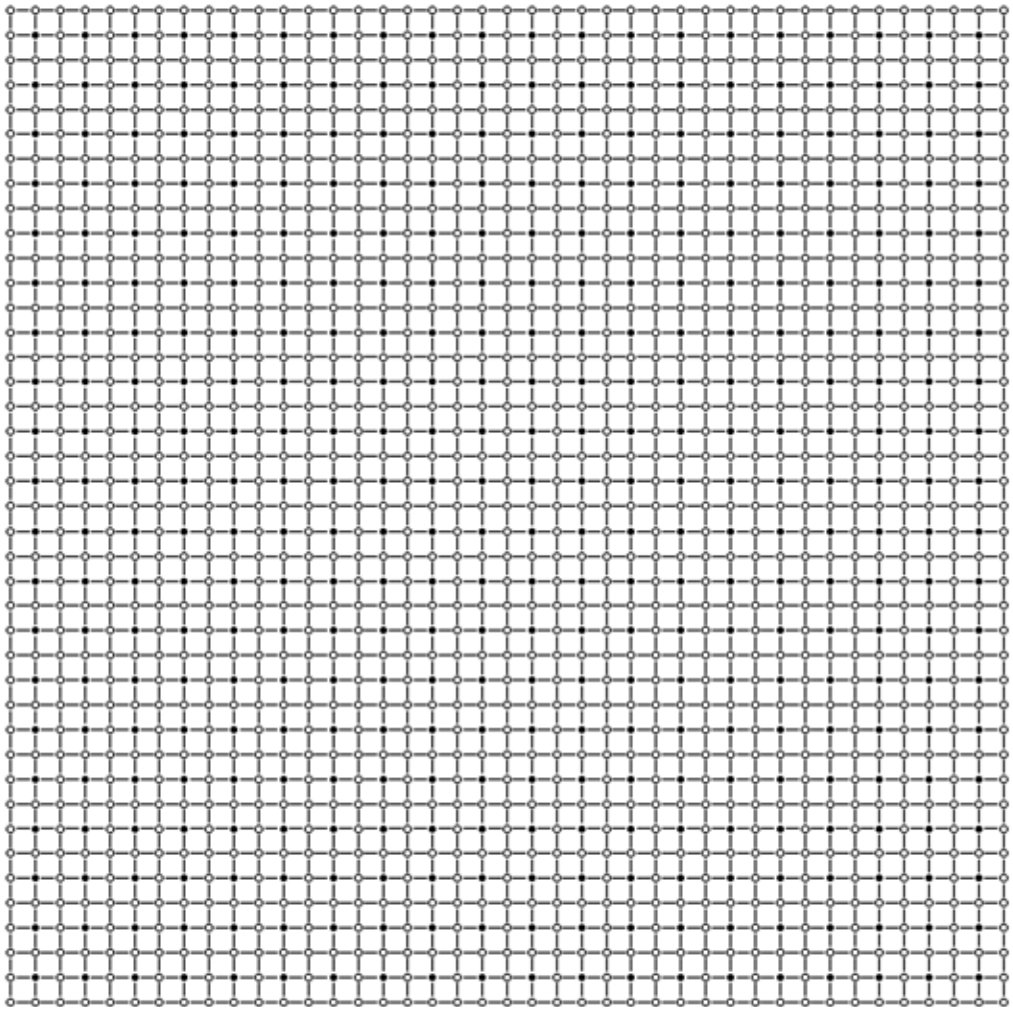
- For k by k torus graphs (lower bound on fill = $O(k^2 \log k)$):
  - Good news:
    ∃ min deg orderings with $O(k^2 \log k)$ fill.
  - Bad news:
    ∃ min deg orderings with $O(k^{2\log_3 4})$ fill [Berman, Schnitger ('90)].

An extreme example of minimum degree

- Notation and convention:
  - A vertex in the current elimination graph is denoted by either a circle or a black dot.
  - A vertex to be eliminated next is denoted by a black dot.
  - All vertices on the boundary of a polygon are in the current elimination graph and are pairwise connected.
    - That is, the vertices on the boundary of a polygon form a clique.

An extreme example - initialization

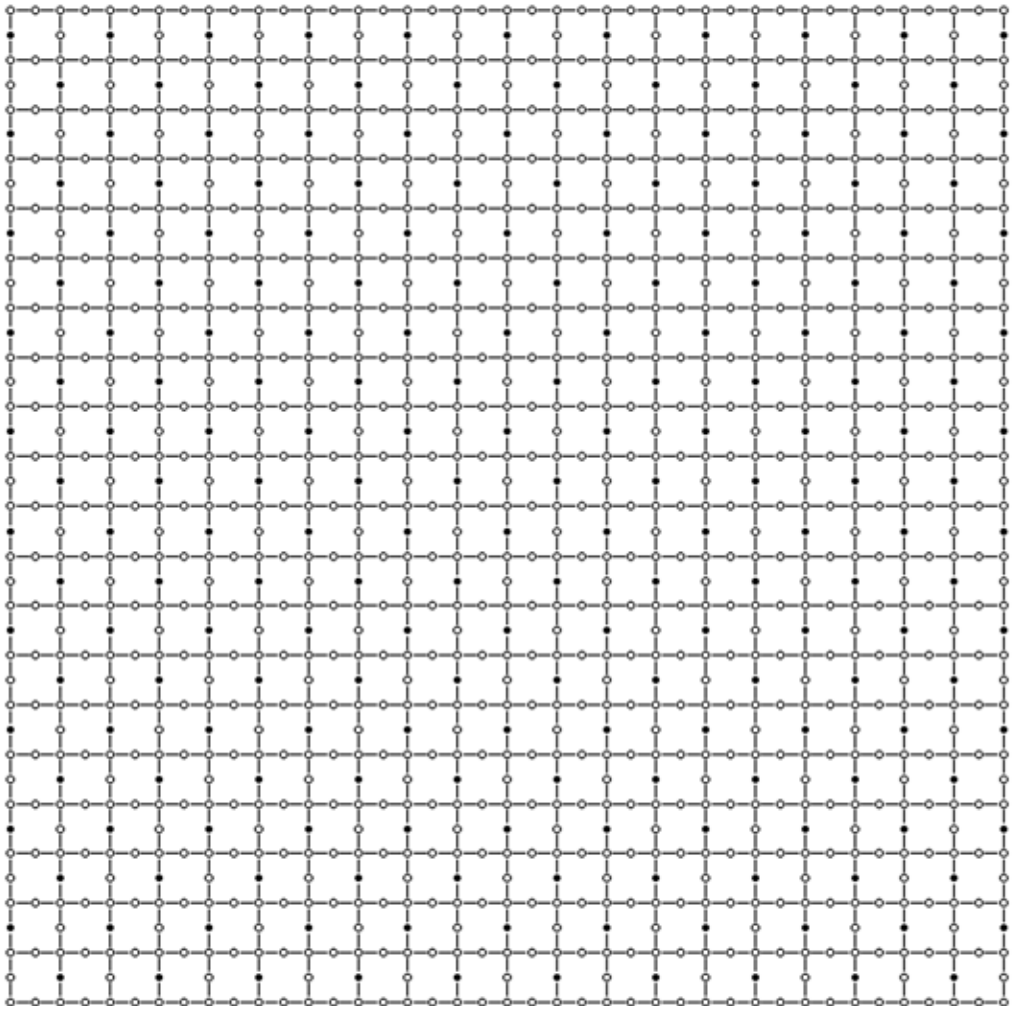
- Each vertex in the initial configuration has degree 8.
  - Remember that the graph is a torus.
- Will eliminate all <u>independent</u> vertices.
  - Number of vertices to be eliminated = $k^2/4$.



An extreme example - initialization

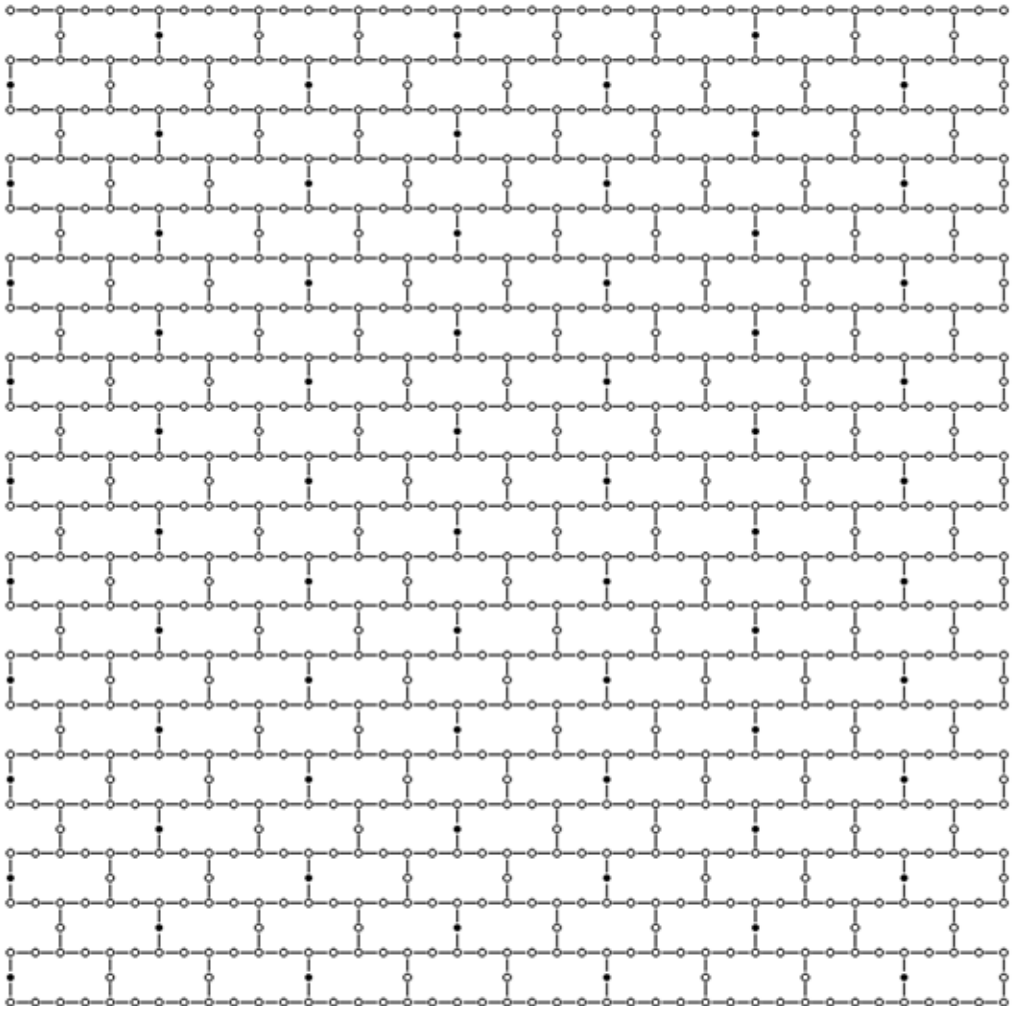
- Two classes of vertices:
  a) between 2 polygons: degree = 12.
  b) between 4 polygons: degree = 20.

- Will eliminate all independent vertices in class (a).
  - Number of vertices to be eliminated = $k^2/16$.



An extreme example - initialization

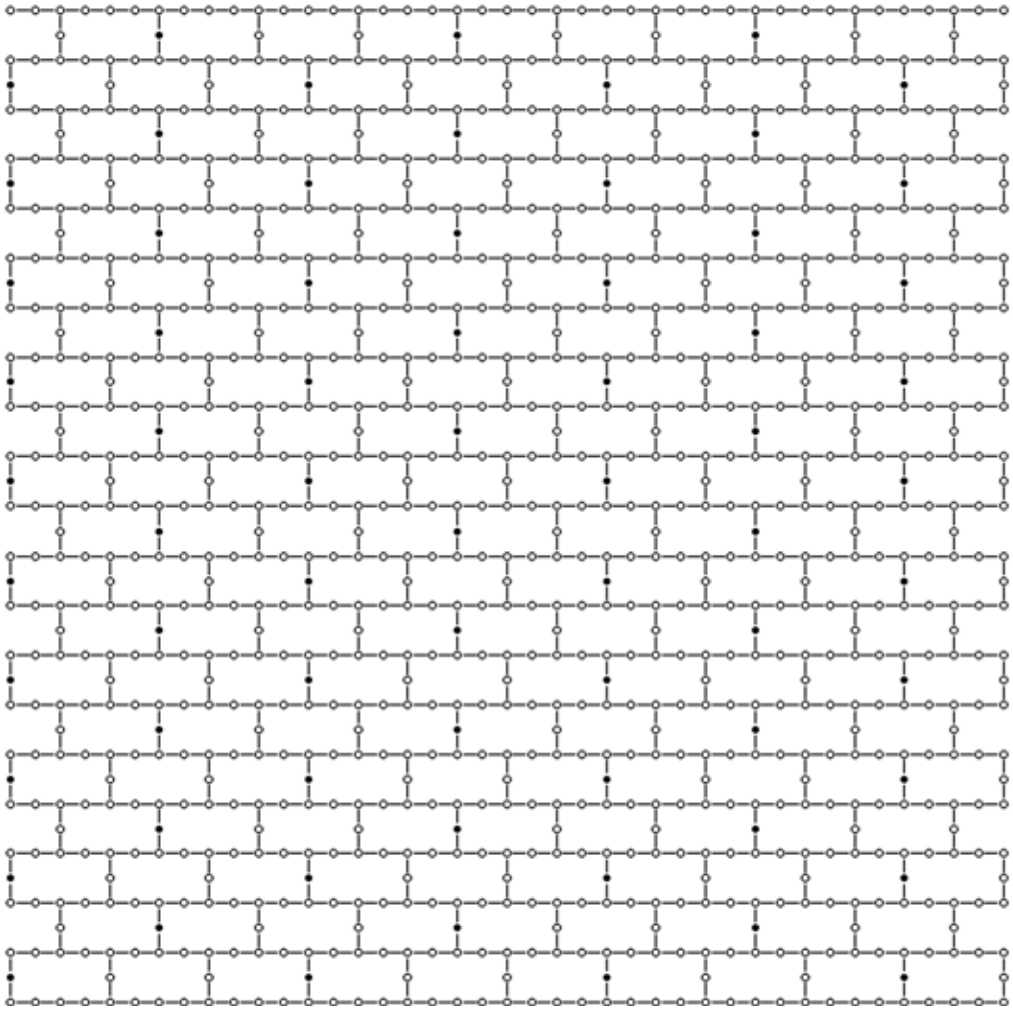
- We will call the resulting graph the brick graph.
  - Let p be the number of bricks.
  - Number of vertices in the brick graph = $5p = \delta k^2$, for some constant δ.



An extreme example - initialization

- Three classes of vertices:
  a) between 2 “horizontal” bricks: degree = 20.
  b) between 2 “vertical” bricks: degree = 20.
  c) between 3 bricks: degree = 27.

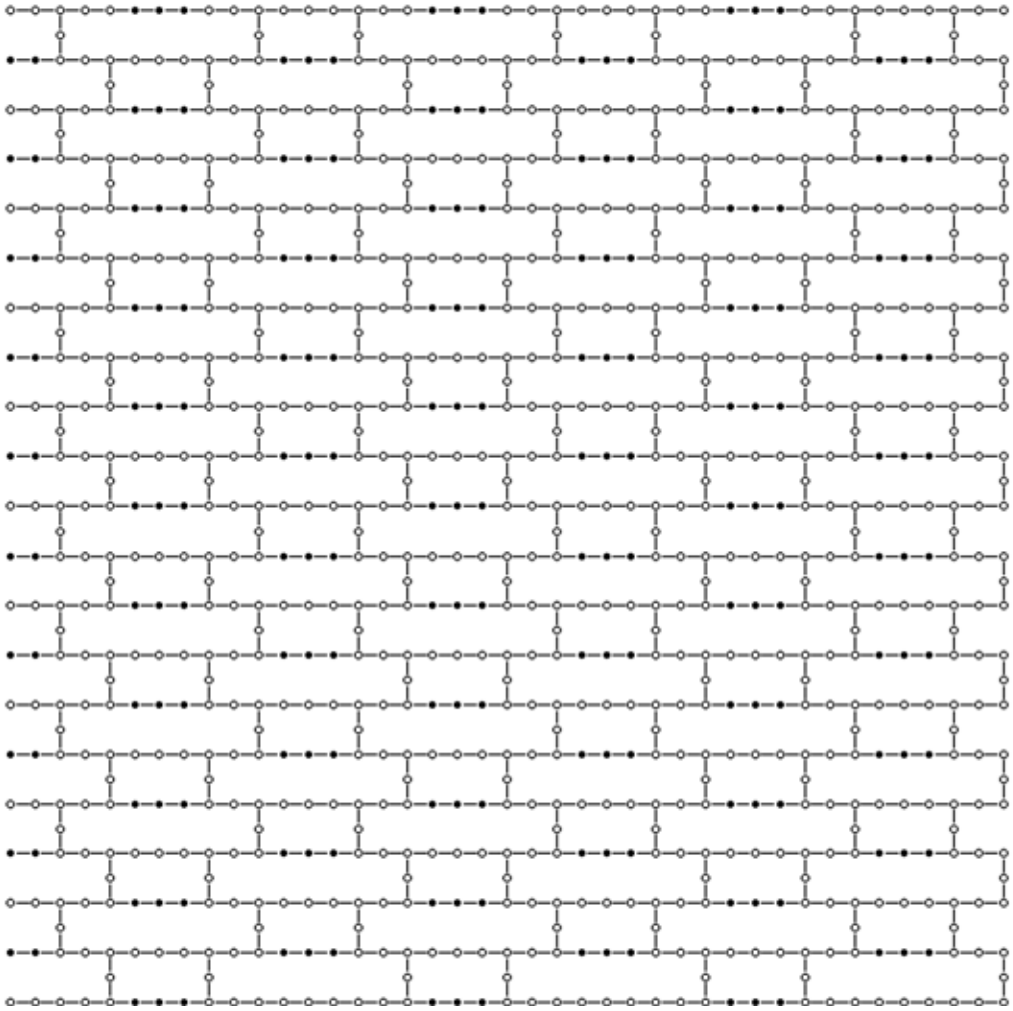
- Will eliminate independent vertices in class (a).
  - Two bricks are merged into a larger brick.



Proceeding with the minimum degree algorithm

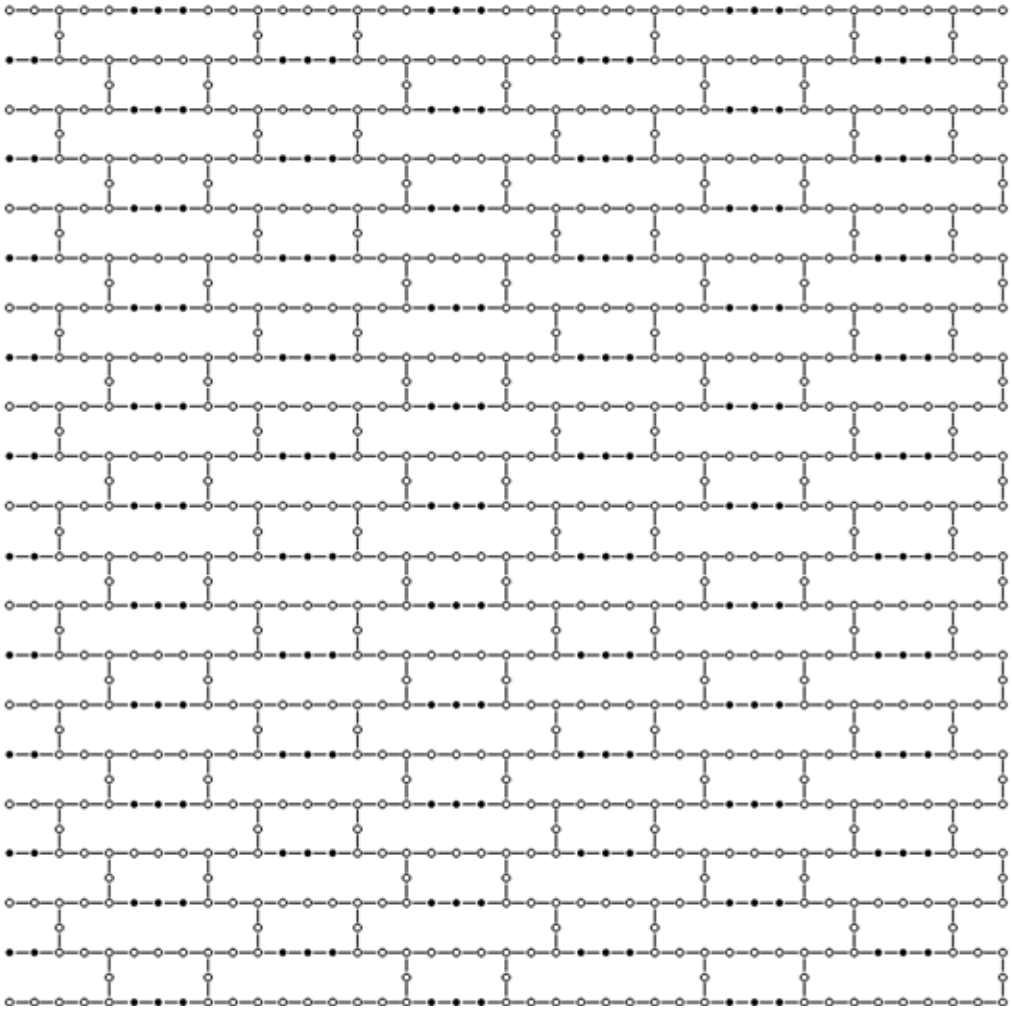
- Four classes of vertices:
  a) between 2 “vertical” bricks (a small brick on top of a large brick): degree = 26.
  b) between 2 “horizontal” bricks: degree = 28.
  c) between 2 large bricks: degree = 36.
  d) between 3 bricks: degree = 41.

- Will eliminate independent vertices in class (a).



Proceeding with the minimum degree algorithm

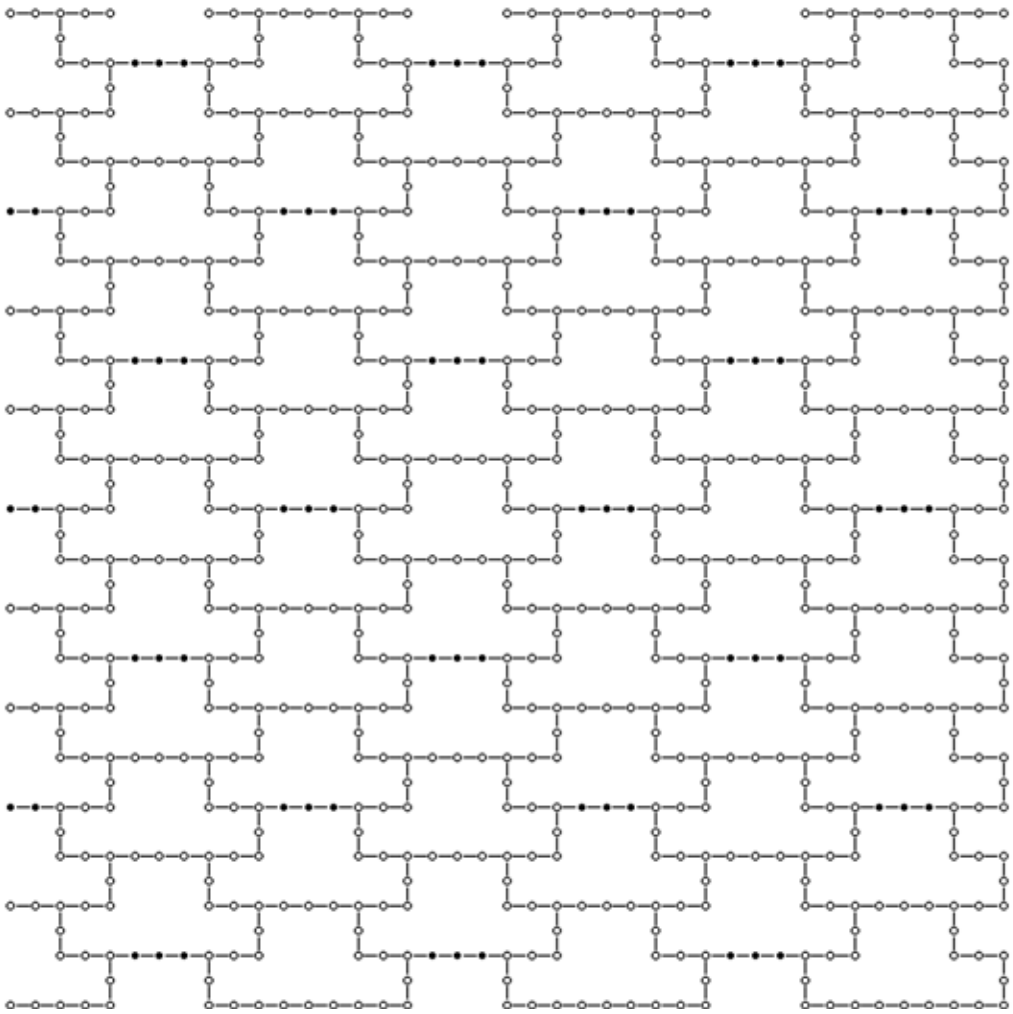
- Eliminate independent vertices in class (a).
  - i.e., those between 2 “vertical” bricks (a small brick on top of a large brick).
  - Note that the set of 3 vertices shared by the small and large bricks are indistinguishable from each other.
  - A small brick and a large brick are merged into a single brick (referred to as a T-brick).



Proceeding with the minimum degree algorithm

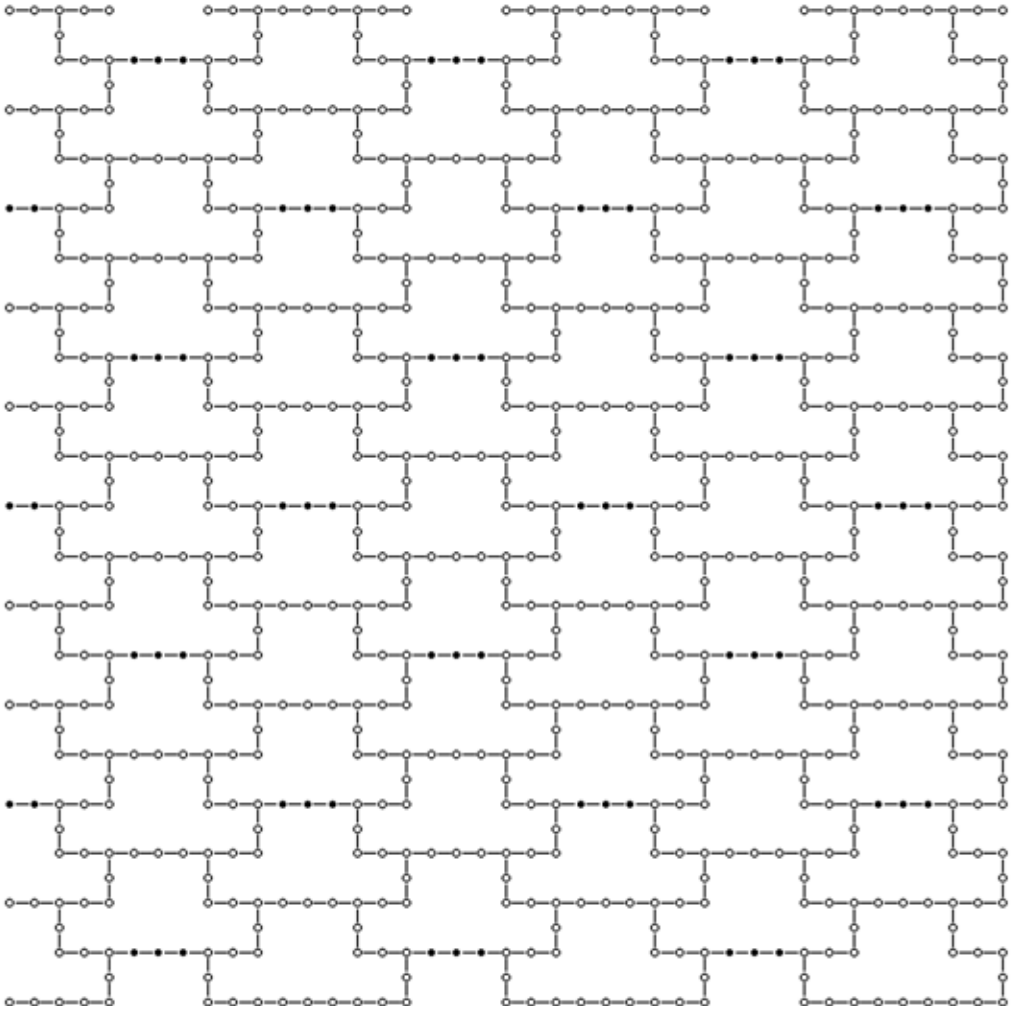
- Two classes of vertices:
  a) between 2 T-bricks: degree = 42.
  b) between 3 T-bricks: degree = 57.

- Will eliminate independent vertices in class (a).



Proceeding with the minimum degree algorithm

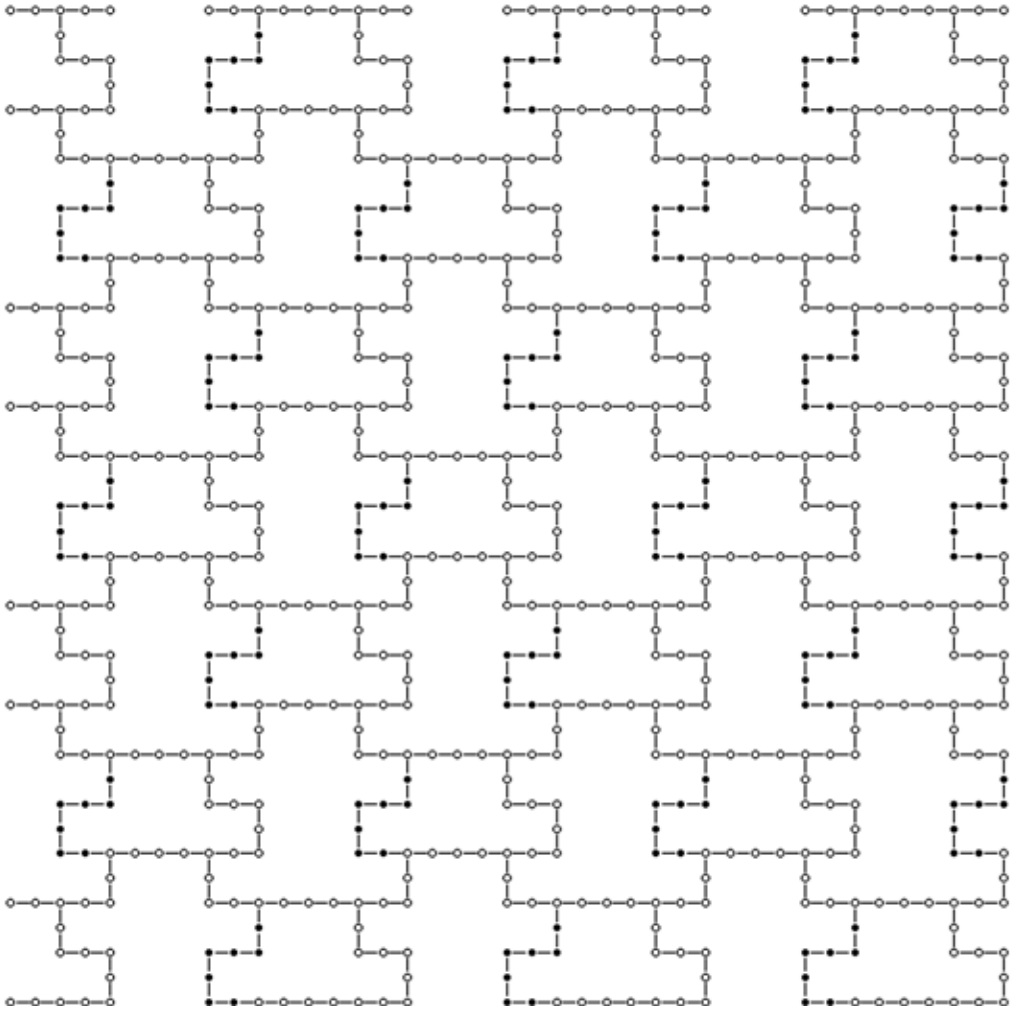
- Eliminate independent vertices in class (a).
  - i.e., those between 2 T-bricks.
  - These independent vertices are selected from those that are shared by 2 vertical T-bricks, with one on top of the other.
  - Two T-bricks are merged to form a new double-T-brick.



Proceeding with the minimum degree algorithm

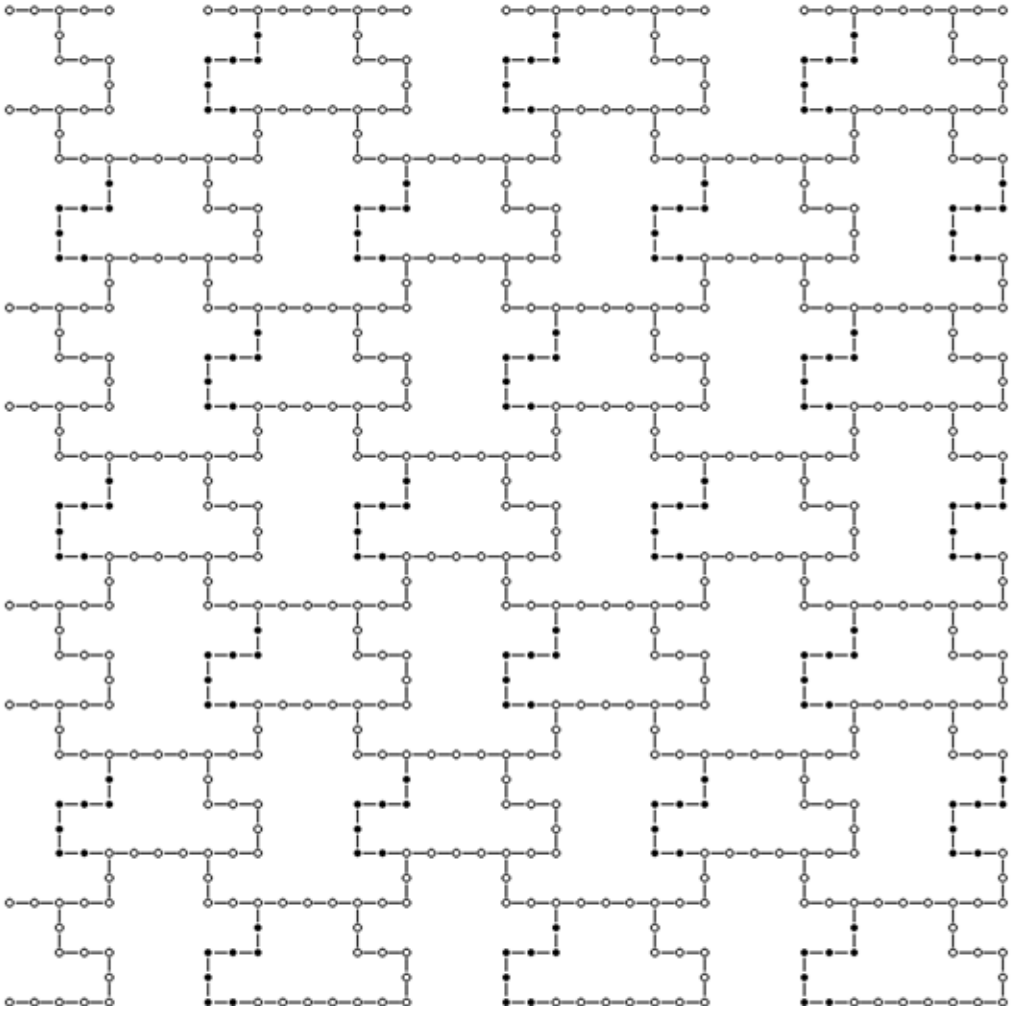
- Four classes of vertices:
  a) between a T-brick and a double-T-brick in the horizontal direction: degree = 54.
  b) between a T-brick and a double-T-brick in the vertical direction: degree = 58.
  c) between 2 double-T-bricks: degree = 74.
  d) between a T-brick and 2 double-T-bricks: degree = 85.

- Will eliminate independent vertices in class (a).



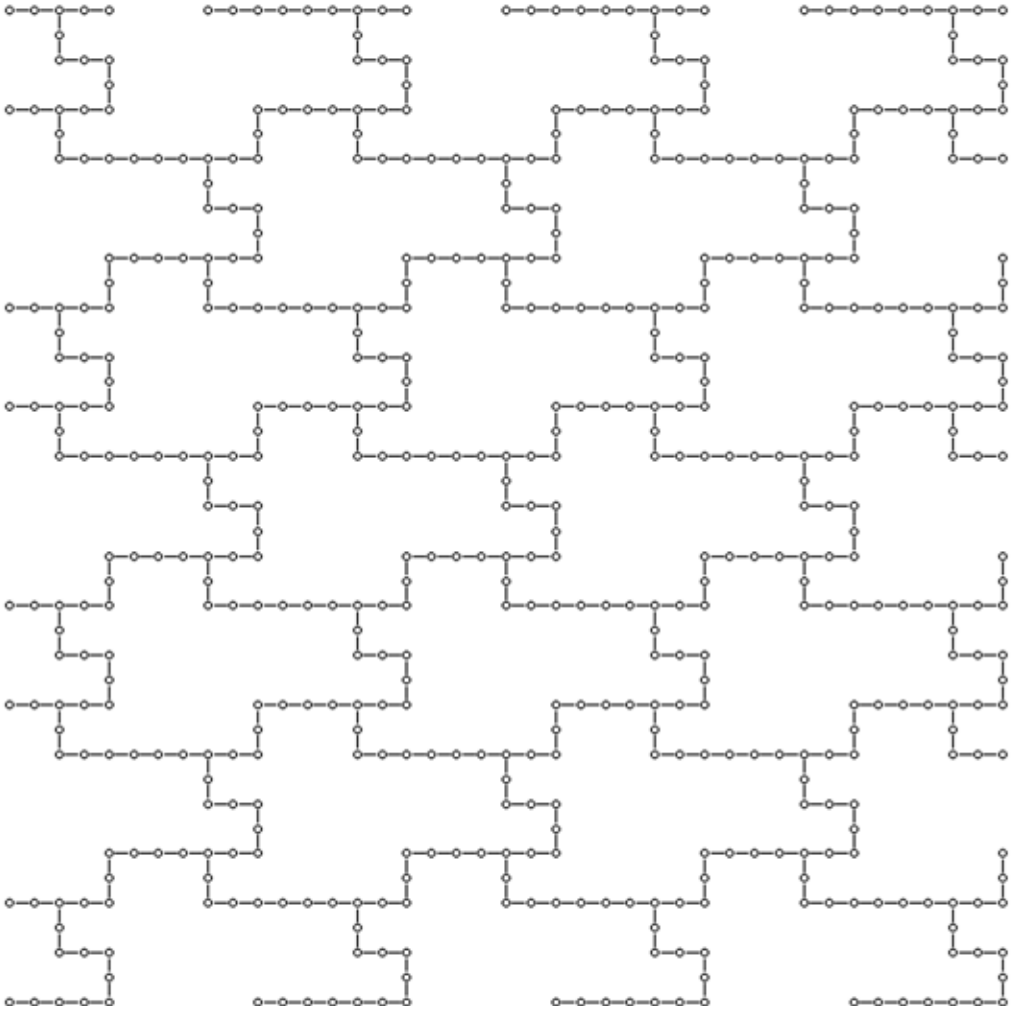
Proceeding with the minimum degree algorithm

- Eliminate independent vertices in class (a).
  - i.e., those shared by a T-brick and a double-T-brick in the horizontal direction: degree = 54.
  - Note that the set of 7 vertices shared by the T-brick and the double-T-brick are indistinguishable from each other.
  - One T-brick and one double-T-brick are merged to form a brick with an odd shape.

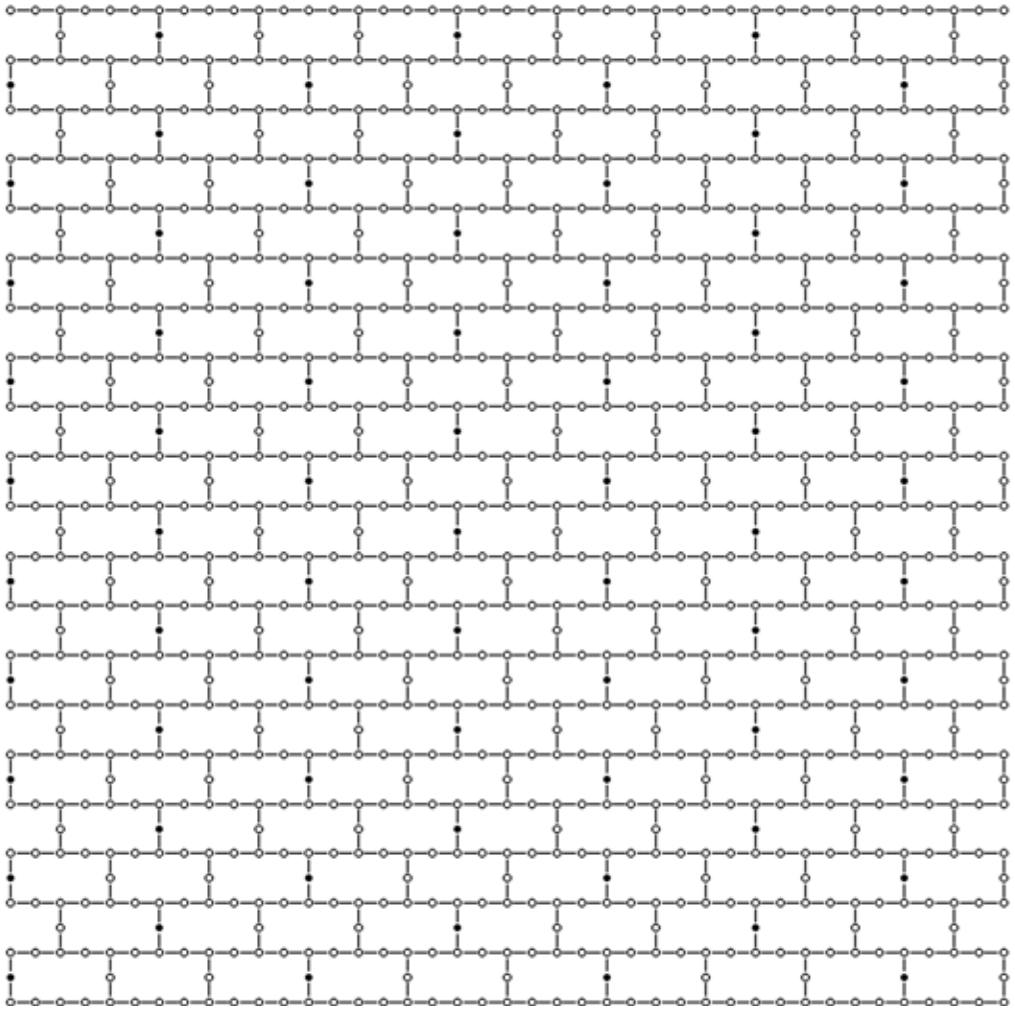


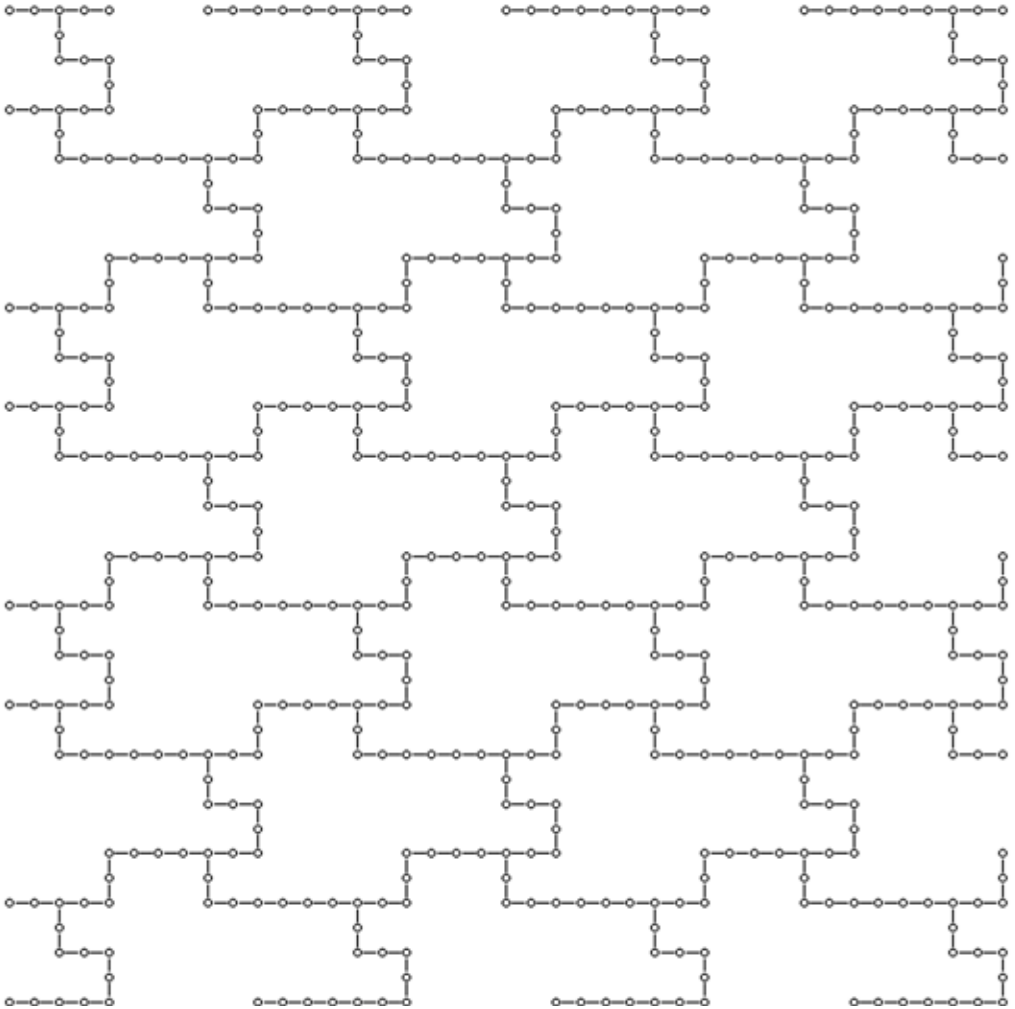
Proceeding with the minimum degree algorithm

- At this point, we obtain a mesh shown on the right.
  - The mesh is isomorphic to the one obtained after the initialization.
  - Vertices that were shared by 2 bricks now become “supernodes”, each of which contains 7 vertices that are indistinguishable from each other.
  - Vertices that were shared by 3 bricks are “supernodes”, each of which contains 1 vertex.



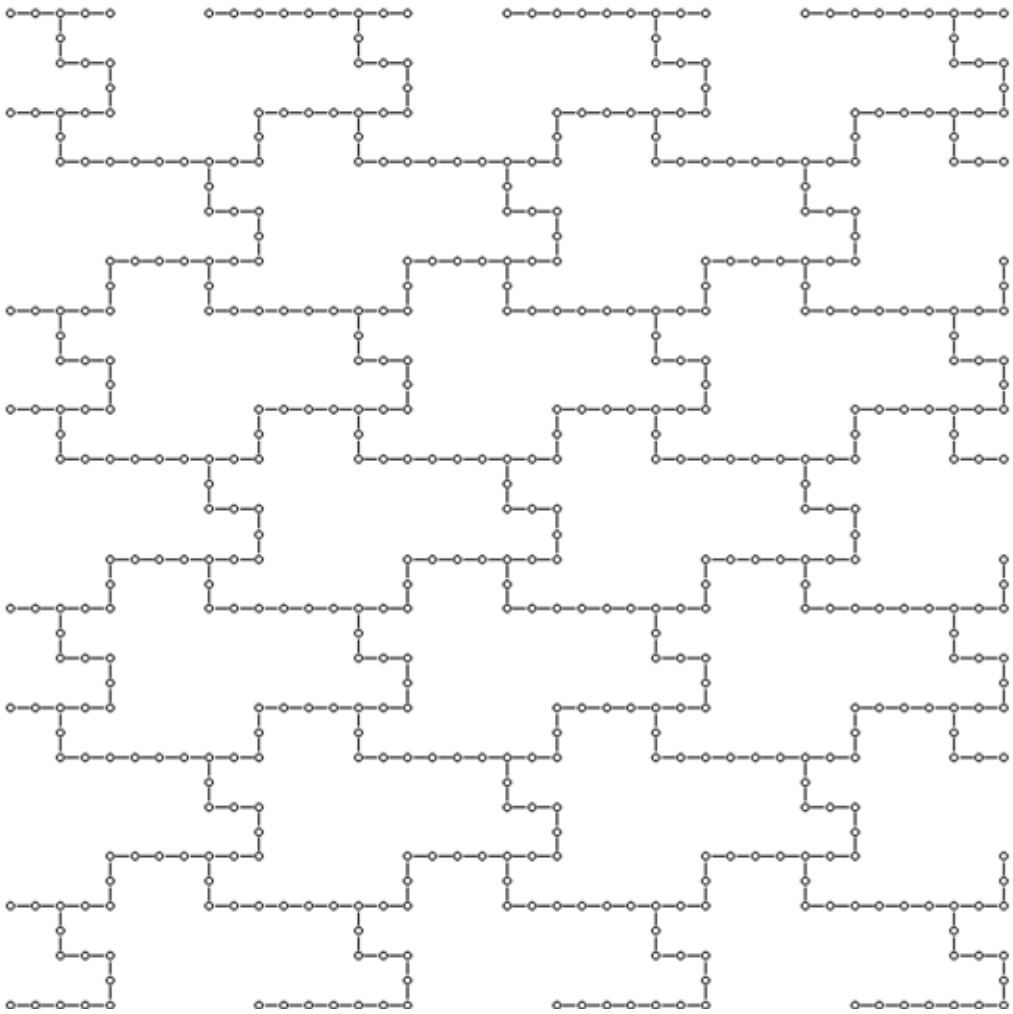
Proceeding with the minimum degree algorithm





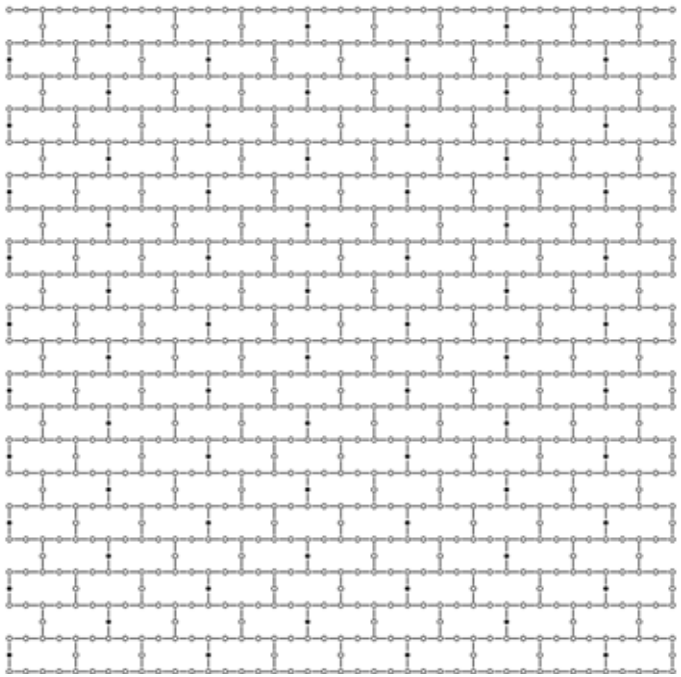
Proceeding with the minimum degree algorithm

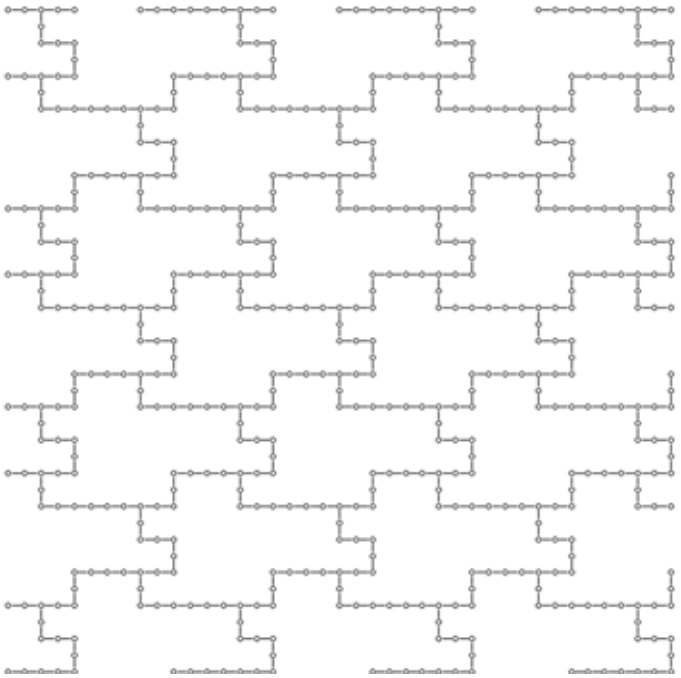
- The same strategies can be applied to the mesh on the right, although the degrees will be different and the number of vertices eliminated in each phase will be different.
- So, the algorithm goes through a number of cycles.
- Each cycle transforms a mesh into an isomorphic mesh.



Complexity analysis

- First note that a total of 9 bricks in the top mesh are merged to form a new brick in the bottom mesh.
- At the end of each cycle, the number of bricks in the mesh will be reduced by a factor of 9.
- There will be approximately $\log_9 p$ cycles.





Complexity analysis

- Consider the beginning of cycle i.
- We want to measure the size of a brick in terms of the initial brick graph.
- Each vertex shared by 2 bricks is a "supernode" containing α_i vertices from the initial brick graph.
  - $\alpha_0 = 1$.
- Each vertex shared by 3 bricks is a "supernode" containing exactly 1 vertex from the initial brick.
- Each boundary segment of a brick is constructed from 4 boundary segments of some bricks in cycle $i-1$.
- So, $\alpha_{i+1} = 4\ \alpha_i + 3 = 2 \times 4^{i+1} - 1$.

Complexity analysis

- Changes in cycle i:
  - Number of bricks = $p/9^i$.
  - Number of vertices associated with each brick = $6\alpha_i+6 = 12\times4^i$.
- Total number of edges τ_i in the elimination graph at the beginning of step i:

$$\frac{1}{2}\left(12\times4^i\right)^2\frac{p}{9^i} \le \tau_i \le \left(12\times4^i\right)^2\frac{p}{9^i}$$

Complexity analysis

- Total number of edges τ in the filled graph is bounded below by:

$$\sum_{i=0}^{\log_9 p} \sigma\left(12 \times 4^i\right)^2 \frac{p}{9^i}$$

  where σ is some constant.

- Easy to show that $\tau \geq \theta(p^{\log_3 4})$.

- Since $p = \delta_0 k^2$, for some constant δ_0, $\tau \geq \theta(k^{2 \log_3 4})$.

The minimum deficiency algorithm

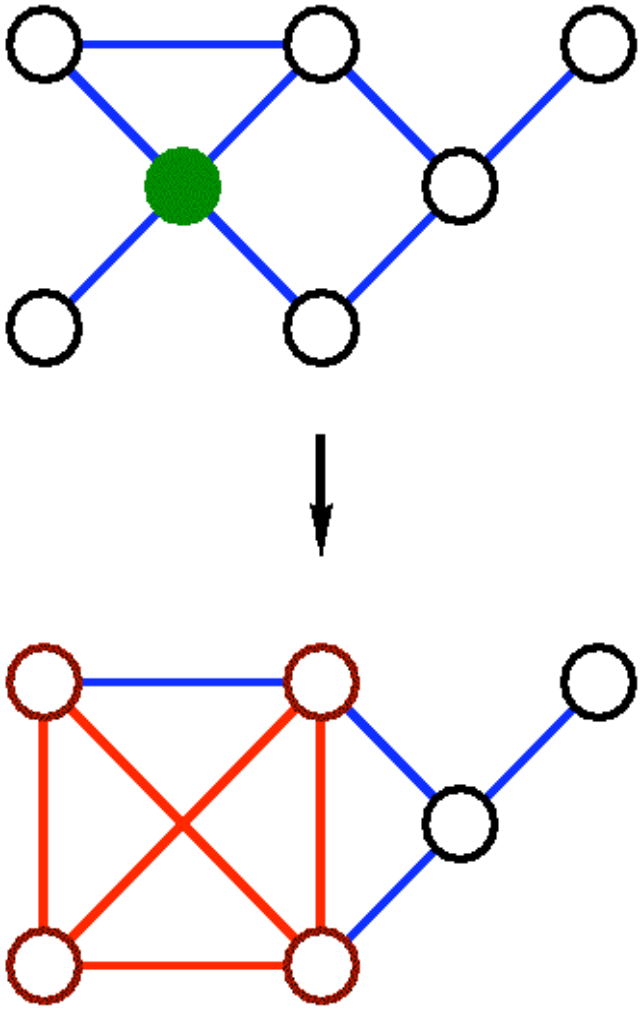
- Use deficiency as the metric [Tinney, Walker ('67)].
  - At each step eliminate the vertex with the minimum deficiency; break ties arbitrary.
  - This minimizes the number of nonzero entries introduced in the rank-1 update of sparse symmetric Gaussian elimination.
  - It is different from the minimum degree algorithm.
    - The deficiency could be zero even though the degree might be nonzero.

The minimum deficiency algorithm

- Extremely expensive to implement, but produce significantly better orderings than the minimum degree algorithm [Rothberg ('96); Ng, Raghavan ('97)].
  - 9% less fill.
  - 21% fewer operations in Gaussian elimination.
  - 250 times more expensive to compute.

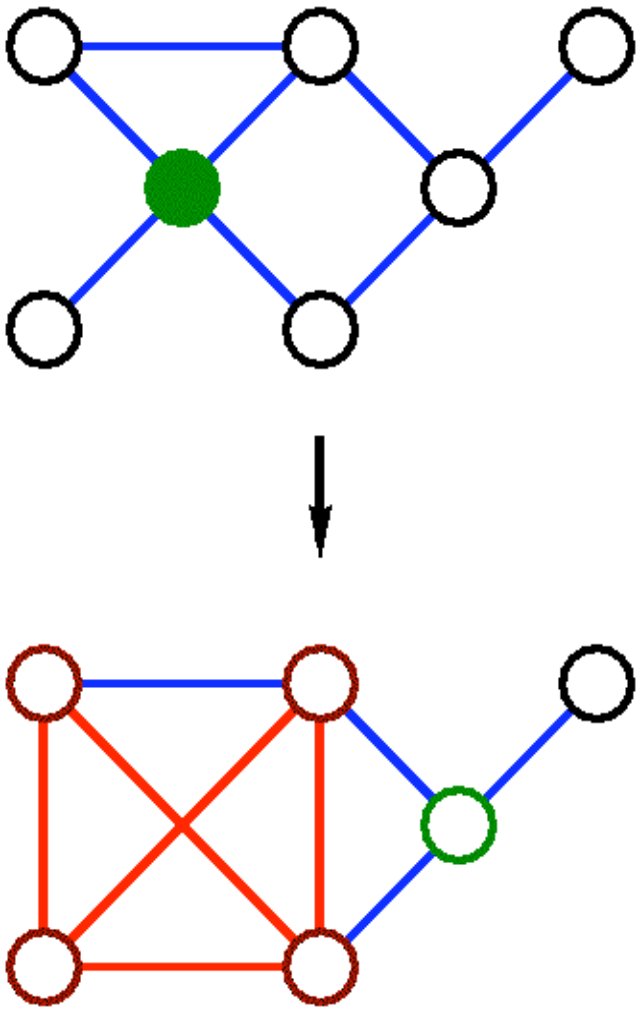
Cost of the minimum deficiency algorithm

- Why is the minimum deficiency algorithm more expensive than the minimum degree algorithm?
- Eliminating vertex v in the minimum degree algorithm:
  - Neighbors of *v* are affected.
  - Degrees of these neighbors may need to be updated.



Cost of the minimum deficiency algorithm

- Eliminating vertex *v* in the minimum deficiency algorithm:
  - Neighbors of *v* are affected.
  - Deficiencies of these neighbors may need to be updated.
  - Neighbors of neighbors of *v* may also be affected.
  - Deficiencies of these neighbors of neighbors may also need to be updated.



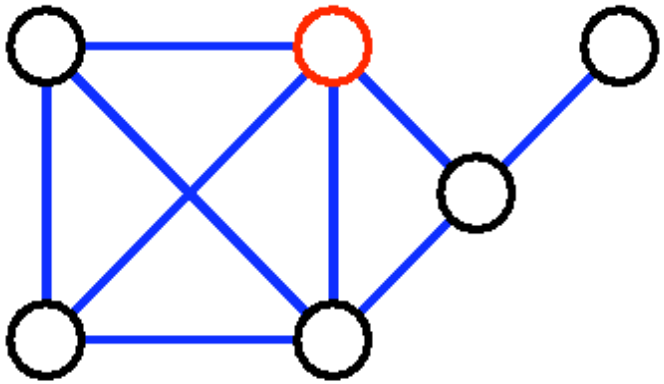
Cost of the minimum deficiency algorithm

- Computing the deficiency of a vertex is a nontrivial task.

- Let S be the set of vertices that are adjacent to v.
- Suppose that d is the degree of vertex v; $d=|S|$.
- Let c be the number of edges currently connecting vertices in S.

- The deficiency of v is $d(d-1)/2 - c$.

- Neighbors of neighbors of v have to be examined to determine c.



Alternatives to minimum deficiency algorithm

- A minimum deficiency ordering is expensive to compute.
  - More vertices need deficiency update at each step.
  - More edges need to be visited in computing deficiency.

- Are there other alternatives (based on greedy heuristics) that are as good as, or better than, minimum degree?

Alternatives to minimum deficiency algorithm

- Two possibilities:
  - Cheap approximations to deficiency?
  - Computing approximate deficiency of fewer vertices?

- Some ideas proposed in [Ng, Raghavan ('97)], [Rothberg, Eisenstat ('97)].

Open problems

- Hybrid orderings:
  - Combine top-down and bottom-up approaches?
- Bottom-up approaches (i.e., local greedy heuristics):
  - Other metrics?
  - Effect of tie breaking?
    - Look-ahead strategies?
  - Complexity of algorithms?
- Minimizing operations in sparse symmetric Gaussian elimination?
- Equivalent to minimizing fill?

Summary …

- The ordering problem.
- Nested dissection.
- Local greedy heuristics.
  - Minimum degree
  - Minimum deficiency
  - Effect of tie-breaking

Numerical sparse Cholesky factorization

- Numerical Cholesky factorization is generally the most time consuming phase in the solution process.

- Assume that symbolic factorization has been performed to determine the sparsity structure of the Cholesky factor.
- Then, conceptually, computing the Cholesky factorization of a sparse matrix is not much different from the dense case.
  - Just need to avoid operation on zero elements.

- Efficient implementations are hard though.

Issues in numerical factorization

- Issues:
  - Compact data structure
    - indirect addressing may be needed to access nonzero elements.
  - Memory hierarchy and data locality
    - cache versus main memory
  - Pipelined arithmetic units and vector hardware.
    - dense versus sparse operations
  - Multiprocessing capability.

Review - dense left-looking Cholesky

- Left-looking dense Cholesky

for $j = 1, 2, \ldots, n$

  /* column modifications - cmod(j,k) */

  for $k = 1, 2, \ldots, j-1$

    for $i = j, j+1, \ldots, n$

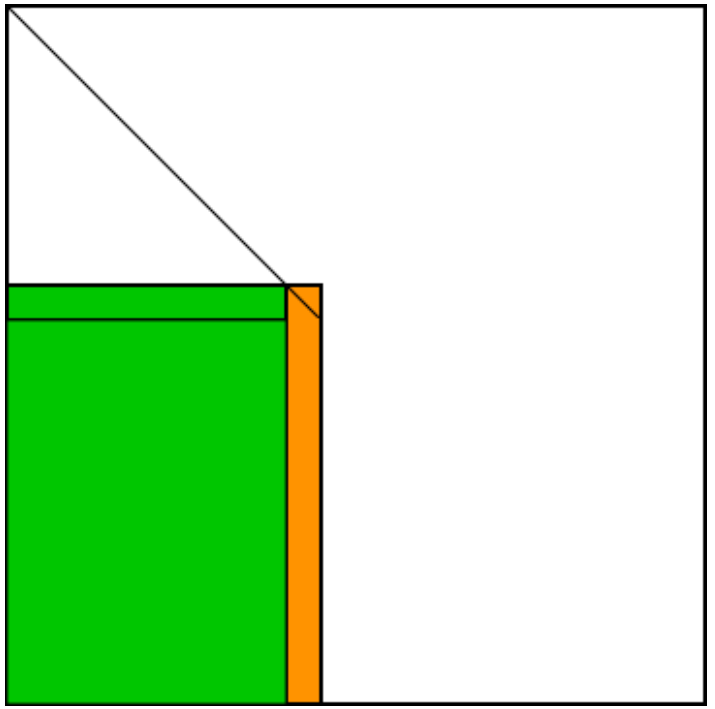
      $A_{ij} \leftarrow A_{ij} - L_{ik} L_{jk}$

  /* column scaling - cdiv(j) */

  $L_{jj} \leftarrow (A_{jj})^{1/2}$

  for $i = j+1, j+2, \ldots, n$

    $L_{ij} \leftarrow A_{ij} / L_{ij}$



Review - dense left-looking Cholesky

- Dense column-column Cholesky factorization
  - column-column algorithm

    for j = 1, 2, …, n

      for k = 1, 2, …, j–1

        cmod(j,k)

      cdiv(j)

- cmod(target,source)
  - target, source := column
- Disadvantages:
  - Little reuse of data in fast memory.

Review - dense left-looking block Cholesky

- Dense panel-panel Cholesky
  - panel-panel algorithm

    for jp = 1, 2, …, n_{panels}

      for kp = 1, 2, …, jp–1

        cmod(jp,kp)

      cdiv(jp)

- cmod(target,source):
  - target, source := block of columns
- Advantages:
  - Good reuse of data in fast memory (BLAS-3, LAPACK)

Left-looking sparse block Cholesky

- Sparse panel-panel Cholesky
  - panel-panel algorithm

    for $jp = 1, 2, \ldots, n_{\text{panels}}$

      for each panel kp such that $L_{jp,kp} \neq 0$

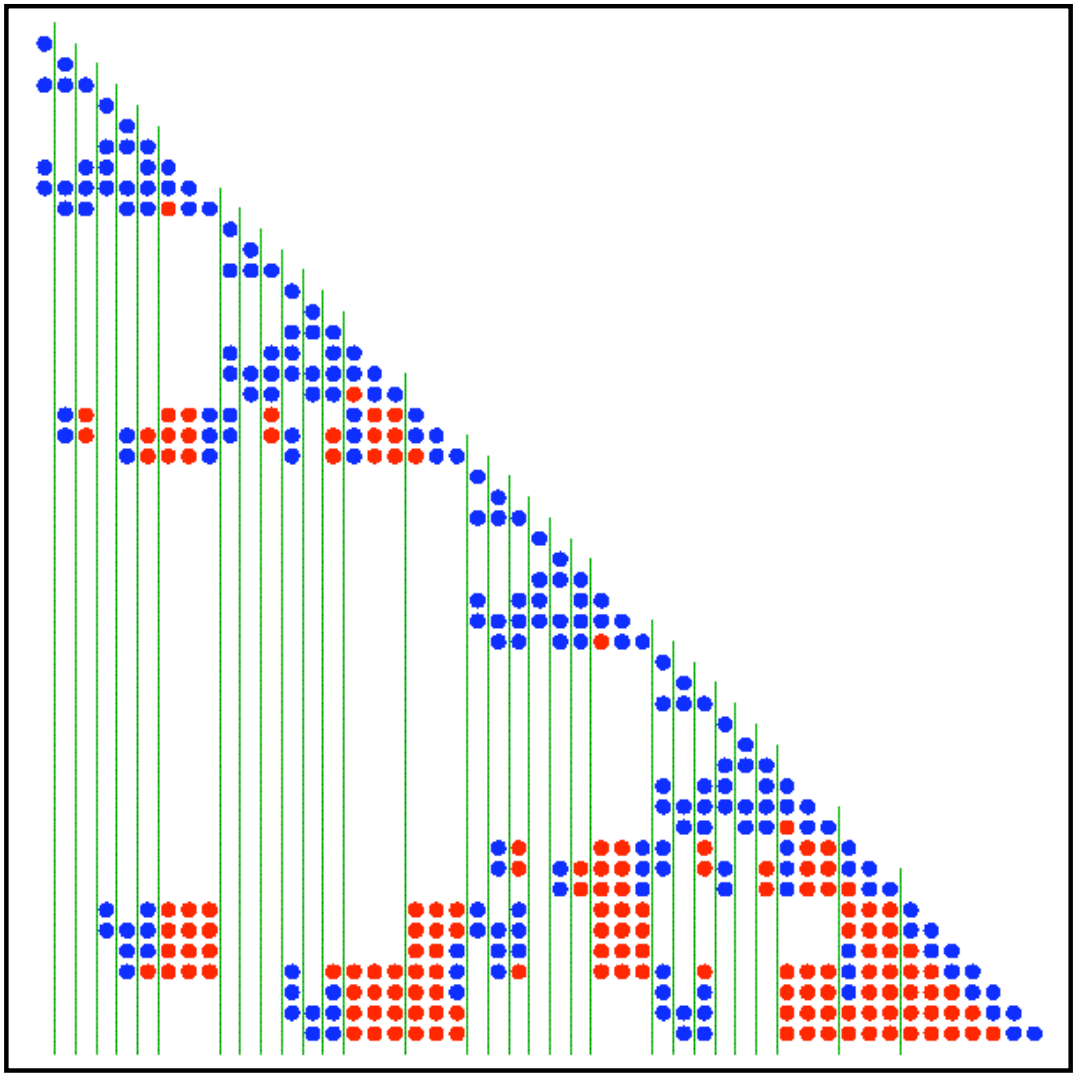
        cmod(jp,kp)

      cdiv(jp)

- Selecting panels in the sparse case - desirable features:
  - allow dense matrix kernels
  - reduce indirect addressing
  - do not allow zeros in the data structure for L

Impact of supernodes on factorization

- Let K be a supernode and consider $j \notin K$.
- Column j is modified by either all columns of K or no columns of K.



Impact of supernodes on factorization

- Supernodes provide a natural way to partition the columns of L.
  - Column j is modified by either all columns of a supernode or no columns of a supernode.

- Columns within each supernode can be treated as a single unit.
  - sparsity structure - one set of row indices for all columns in a supernode
  - updates to column j from the columns in a supernode can be accumulated before applying them to column j.

Left-looking column-column formulation

- Sparse column-column Cholesky
  - column-column algorithm

    for j = 1, 2, …, n

      for each k such that $L_{jk} \neq 0$

        cmod(j,k)

      cdiv(j)

- cmod(target,source):
  - target := column
  - source := column

Left-looking column-column formulation

- Popular approach:
  - Waterloo SPARSPAK, Yale YSMP
- Advantages:
  - easy to implement
  - low work space requirement
- Disadvantages:
  - indirect addressing
  - little reuse of data in fast memory

Left-looking block-column formulation

- Sparse supernode-column Cholesky
  - supernode-column algorithm

    for $j = 1, 2, \ldots, n$

      for each supernode K such that $L_{jK} \neq 0$

        cmod(j,K)

      cmod(j,J), where $j \in J$

      cdiv(j)

- cmod(target,source):
  - target := column
  - source := supernode

Left-looking block-column formulation

- Previous work
  - [Ashcraft et al (1987), Simon et al (1989)]
- Advantages:
  - Enable dense vector operations to reduce indexing overhead
    - When all columns of a supernode update column *k*, the update can be computed as a dense matrix-vector product (level-2 BLAS).
      - Can use loop unrolling to reduce memory traffic.
      - No indirect indexing is needed in computing the update.
  - Low work space requirement
    - Level-2 BLAS : Need a vector to hold a matrix-vector update.
- Disadvantages:
  - little reuse of data in fast memory

Left-looking block-block formulation

- Sparse supernode-supernode Cholesky
  - supernode-supernode algorithm

    for $J = 1, 2, \ldots, n_{\text{supernodes}}$

      for each supernode K such that $L_{JK} \neq 0$

        cmod(J,K)

      cdiv(J)

- cmod(target,source):
  - target := subset of columns within supernode
  - source := supernode
- cdiv(source):
  - source := supernode

Left-looking block-block formulation

- Previous work
  - [Ashcraft et al ('87); Duff & Reid ('83)]
  - [Ng & Peyton (1993); Rothberg & Gupta (1993)]
- Advantages:
  - cmod's can be implemented as dense matrix-matrix multiplications to reduce indexing overhead
    - If a supernode updates several columns of another supernode, the update can be computed as a dense matrix-matrix product (level-3 BLAS).
      - use loop unrolling to reduce memory traffic
      - organize computation to reuse data in fast memory
      - Further reduce indirect indexing and memory traffic.
  - cdiv's can be implemented using dense block Cholesky factorization.

Left-looking block-block formulation

- Disadvantages:
  - increased work space requirement
    - Level-3 BLAS : Need a matrix to hold a matrix-matrix update.
      - Modest amount of storage in most cases.

Left-looking block-block formulation

- Sparse supernode-supernode Cholesky
  - supernode-supernode algorithm

    for $J = 1, 2, \ldots, n_{\text{supernodes}}$

      for each supernode K such that $L_{JK} \neq 0$

        cmod(J,K)

      cdiv(J)

- For large supernode K, subdivide K into blocks, such that each block fits into fast memory.
- Organize computation in terms of blocks within supernodes.

Issue in left-looking formulation

- For simplicity, let's look at sparse column-column Cholesky again:
  - column-column algorithm

    for j = 1, 2, …, n

      for each k such that $L_{jk} \neq 0$

        cmod(j,k)

      cdiv(j)

- Exercise: There is one technical difficulty that has not been resolved.
  - What is it?
  - How to resolve it?

Determining row structure in a column approach

col-col algorithm

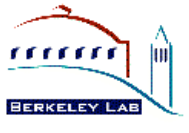
for j = 1, 2, …, n

for each k such that $L_{jk} \neq 0$

cmod(j,k)

cdiv(j)

- The data structure is column-oriented.
  - We store the nonzero elements of L by columns.
  - So is the sparsity structure; i.e., we store the row indices by columns.
    - We know the sparsity structure by columns.
    - We do not know the sparsity structure by rows.
  - But we need the sparsity structure of each row.

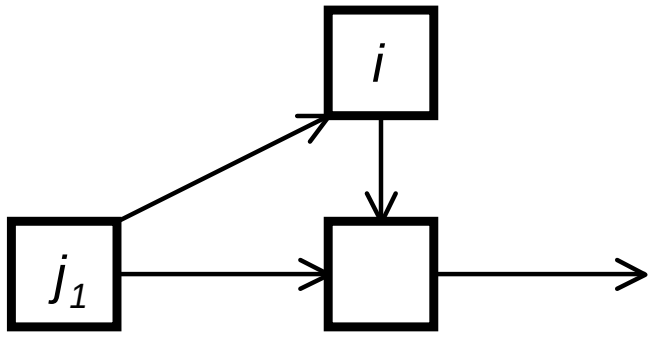


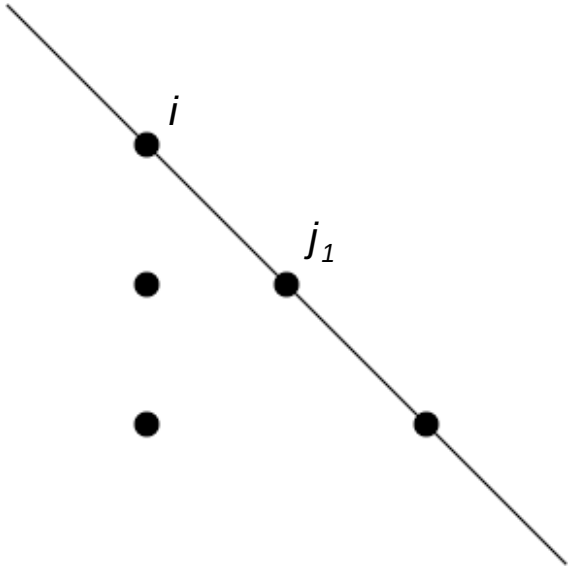
Determining row structure in a column approach

- Solutions:
  - Store the column subscripts by rows as well as the row subscripts by columns.
    - Not a good idea.
  - Use the fact that the columns subscripts of the nonzero elements in a row is a row subtree in the elimination tree.
    - Need to know how to traverse the elimination tree.
      - Implementation can be a bit complicated, but doable.
  - Dynamically create the sparsity structures of the rows during numerical factorization.

Determining row structure in a column approach

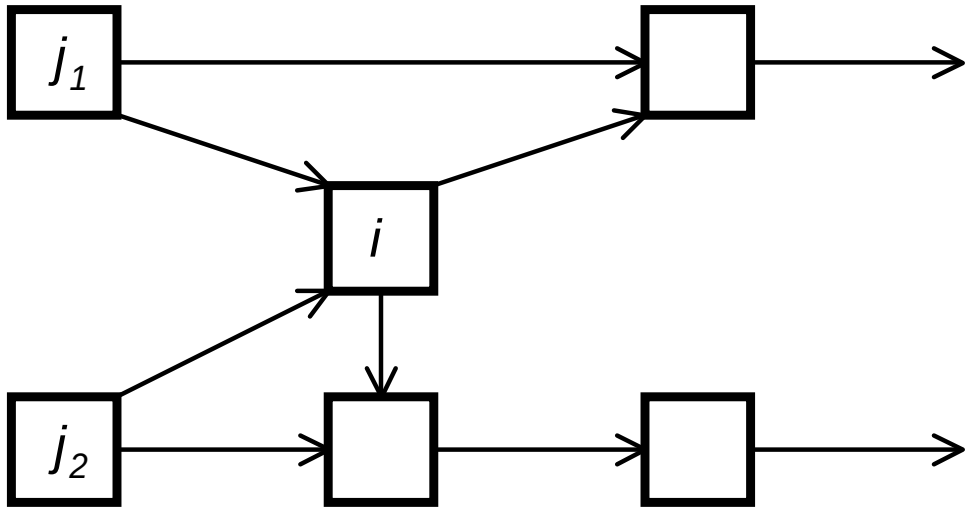
- Creating the row structures …
  - For each row, maintain a link-list (which is initially empty).
  - Suppose we have just computed column *i* of *L*.
    - Suppose that the first off-diagonal nonzero element in column *i* of *L* is in row j_1.
      - So, column j_1 of *A* is the next column to be modified by column *i* of *L*.
    - Insert *i* into a link-list for row j_1.

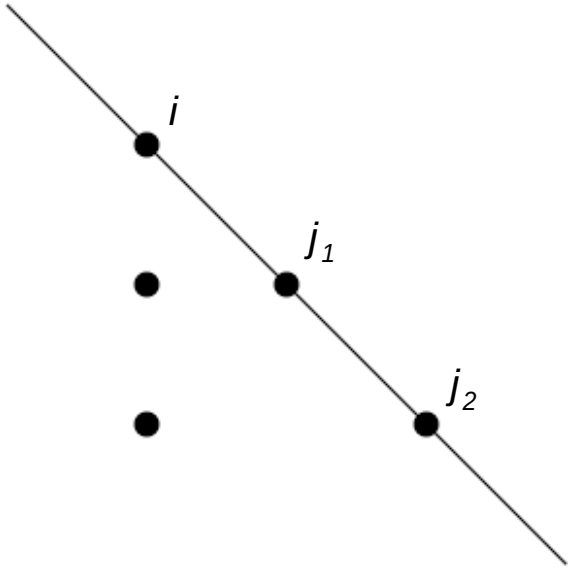




Determining row structure in a column approach

- Creating the row structures …
  - Suppose we are ready to compute column j_1 of L.
    - We first remove a column (say, i) from the link-list of row j_1.
      - Column i of L will modify column j_1 of A.
    - Suppose that the nonzero element right below row j_1 in column i of L is in row j_2.
      - Meaning that column i of L will modify column j_2 of A next.
    - Insert i into the link-list for row j_2.





Right-looking formulation

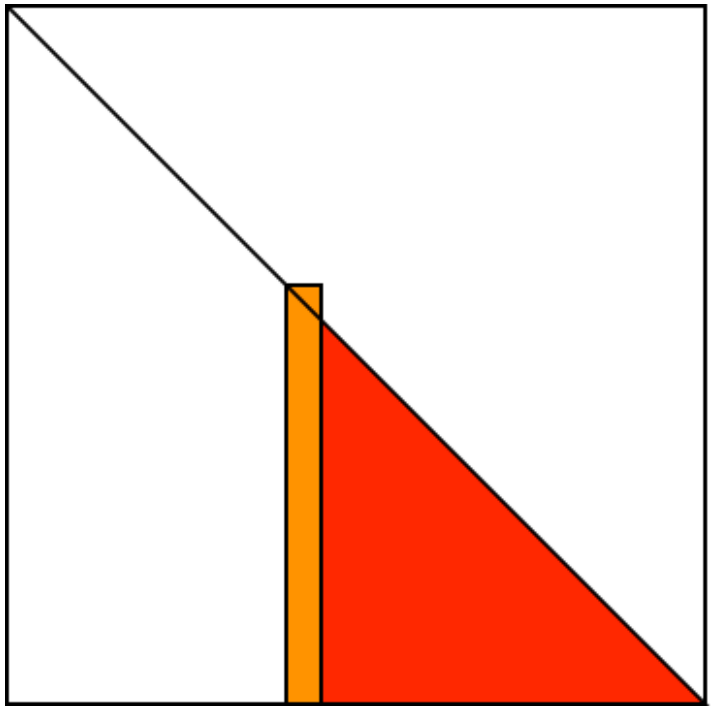
- Right-looking Cholesky
  - dense panel-panel algorithm

    for jp = 1, 2, …, n_{panels}

      cdiv(jp)

      for kp = jp+1, jp+2, …, n_{panels}

        cmod(kp,jp)



- Just like the left-looking formulations, different definitions of panels give different variants of dense right-looking algorithms.

Sparse right-looking Cholesky

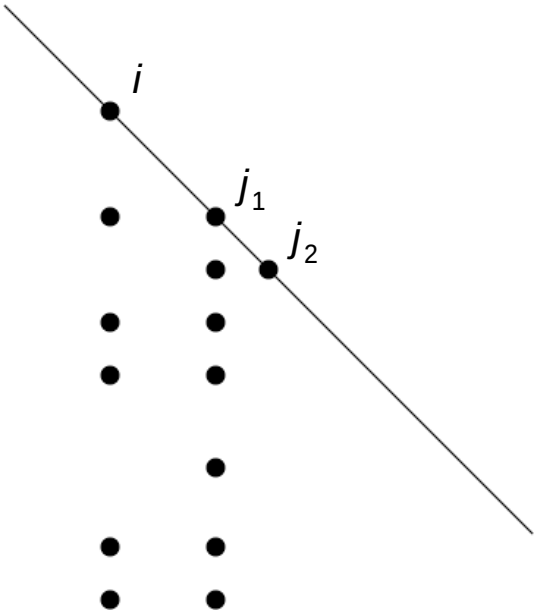
- Sparse versions are similarly defined.
- Straightforward implementation is inefficient in the sparse case ...
  - indirect addressing may require expensive row subscript searching and matching.
- Multifrontal method ([Duff & Reid 83], ...)
  - can be viewed as an efficient implementation of the right-looking algorithm

Multifrontal approach

- Multifrontal method ([Duff & Reid 83], ...)
  - Update to future columns of A is not applied immediately to the active submatrix, but is saved for later use.
  - At a later stage of the factorization, update is then retrieved and applied to the active submatrix.
  - A column modification arrives at its target via a sequence of update matrices.
  - Computation entirely involves dense matrices.
  - Natural incorporation of supernodes ---
    - Uses same kernel routines as left-looking supernode-supernode Cholesky.

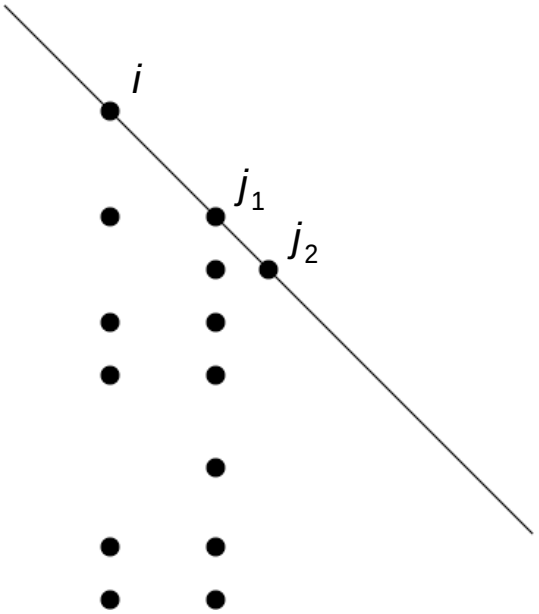
Multifrontal approach

- Suppose we have just computed column i of L.
  - We compute the update matrix for future columns of A.
    - This will be a dense rank-5 update (in this example).
  - Instead of applying the update immediately to columns j_1, j_2, …, suppose we save the update somewhere (to be determined).



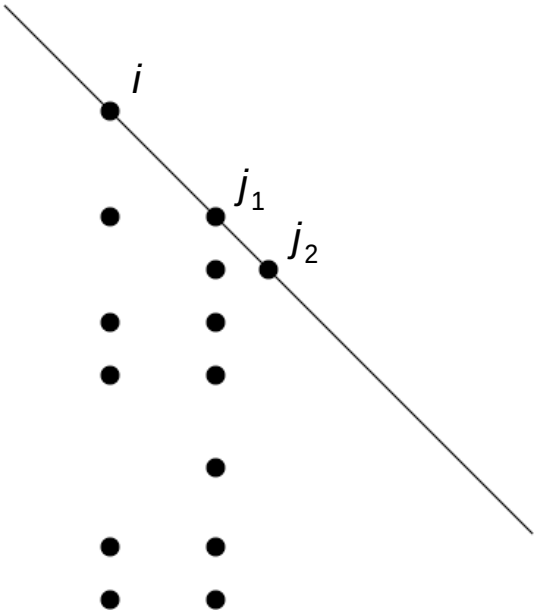
Multifrontal approach

- Suppose we get to column j_1.
  - We first retrieve the rank-5 update due to column i of L from somewhere, and apply the update to column j_1 of A.
  - Apply all other updates to column j_1 of A in a similar fashion.
  - Compute column j_1 of L.
  - Note that all computation can be done in a dense matrix, whose order is the same as the number of nonzero elements in column j_1 of L.
    - The dense matrix is called a frontal matrix.



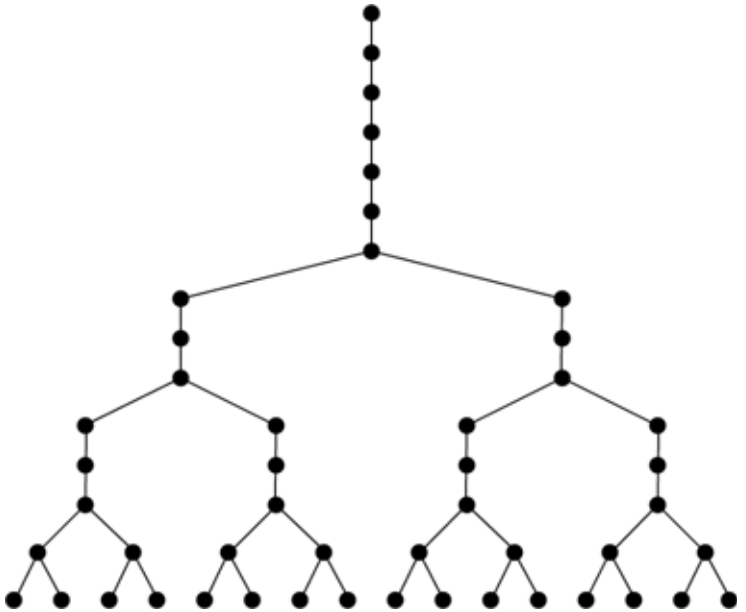
Multifrontal approach

- After column j_1 of L has been computed …
  - Now we repeat the same process; i.e., we compute the update matrix (due to column j_1) for future columns of A and save the update somewhere (for column j_2 of A in this example).
  - Note that the update matrix due to column j_1 of L will have incorporated the update matrix due to column i of L. There is no need to keep the update matrix due to column i of L.



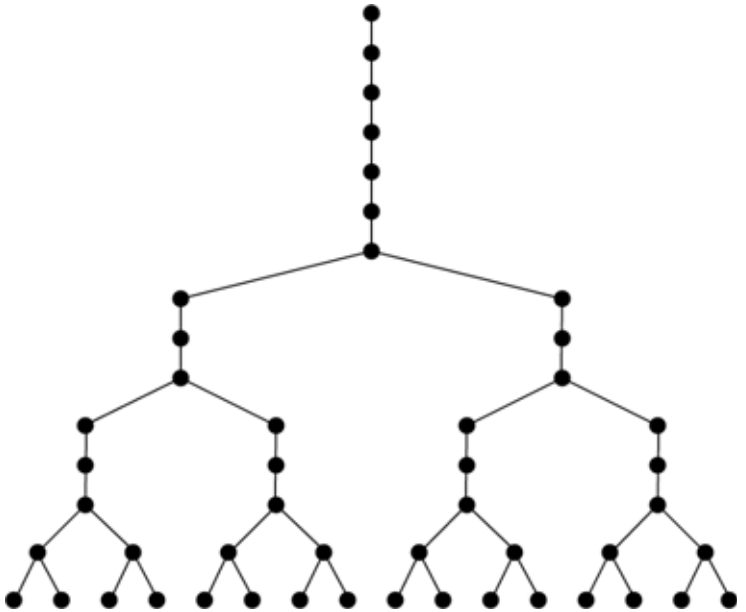
Managing update matrices in multifrontal

- How to manage the update matrices?
  - Very difficult in general …
- Suppose the elimination tree is postordered.
  - The vertices in each subtree are labeled consecutively before the root of the subtree is labeled.
  - By the time we compute column i of L, all the columns associated with the subtree rooted at x_i must have been computed.



Managing update matrices in multifrontal

- When the elimination tree is postordered …
  - The necessary update matrices required by column *i* of *A* come from the children of x_i.
    - This suggests that a "first-in first-out" data structure (i.e., a stack) for storing the update matrices.
  - Using a stack, the update matrices required by column *i* of *A* are always at the top of stack.



Multifrontal approach

- Advantages:
  - Good for vectorization
  - Reduced indirect addressing
  - Best for locality of memory references

- Disadvantages:
  - Complicated and can require substantial storage for stack of update matrices
    - Issue: How to organize the computation to reduce the size of the stack?
  - Data movements

Left-looking versus multifronal

- Left-looking and multifrontal perform exactly the same computation, but in different order.
  - Supernodal versions can be implemented using exactly the same dense matrix kernels.
- Multifrontal needs to save and manage the updates.
  - Can organize the computation so that updates can be managed by a stack.
    - $\Rightarrow$ extra working storage.
    - $\Rightarrow$ extra data movement.
- Multifrontal has better data locality.
- Multifrontal is more appropriate for out-of-core implementation.

General framework

- Let A be a given sparse SPD matrix.
- General framework :
  - Ordering :
    - Find P so PAP^T has a sparse Cholesky factor L_P.
  - Symbolic factorization :
    - Determine the structure of L_P.
    - Set up efficient data structure to store L_P.
  - Numerical factorization :
    - Compute L_P.
  - Triangular solution :
    - Solve $L_p u = Pb$ and $L_P^T v = u$.
    - Set $x = P^T v$.

Sparse SPD solvers

- Different choices of ordering, symbolic factorization, and numerical factorization algorithms result in different solvers.

- Not all ordering/symbolic factorization/numerical factorization algorithms are equal.
- Different algorithms for the same step may have different complexities and may produce significantly different output.

Summary ...

- Numerical factorization of sparse SPD matrices.
- Implementation of left-looking algorithm.
- Implementation of multifrontal algorithm.

Numerical comparisons

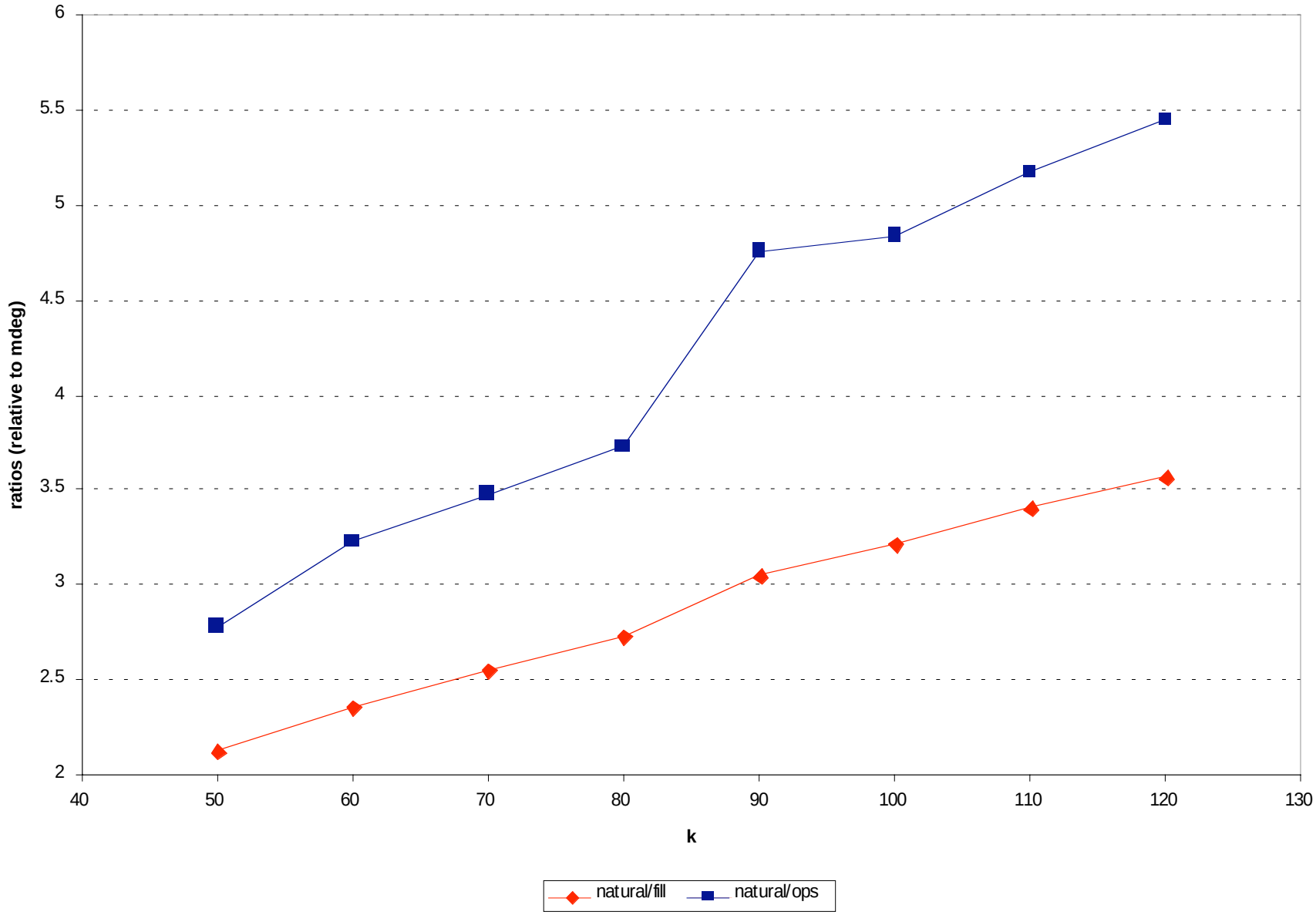
- What to compare?
  - Orderings :
    - profile-reduction orderings
    - minimum degree orderings
  - Numerical factorizations :
    - left-looking column-column
    - left-looking supernode-supernode
    - right-looking multifrontal

Profile versus minimum degree

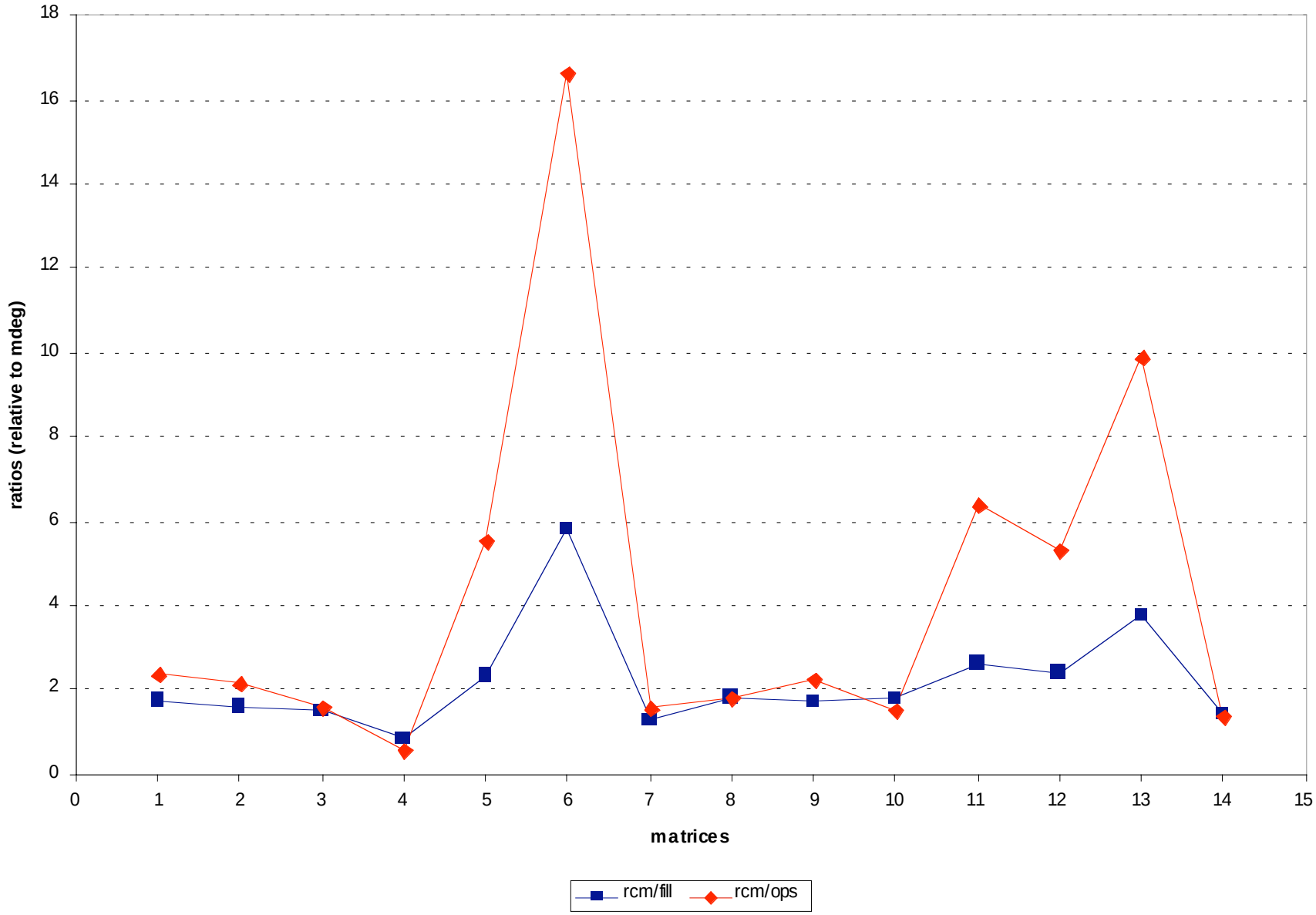
- Consider two sets of matrices:
  - Finite element grids (50×50 to 120×120).
  - A set of structural analysis matrices from the Harwell-Boeing Collection.

matrix	n	|A|/2
BCSSTK13	2,003	42,943
BCSSTK14	1,806	32,630
BCSSTK15	3,948	60,882
BCSSTK16	4,884	147,631
BCSSTK17	10,974	219,812
BCSSTK18	11,948	80,519
BCSSTK19	817	3,835
BCSSTK23	3,134	24,156
BCSSTK24	3,562	81,736
BCSSTK25	15,439	133,840
BCSSTK26	1,922	16,129
BCSSTK28	4,410	111,717
BCSSTK29	13,992	316,740
BCSSTK33	8,738	300,321

Profile versus minimum degree



Profile versus minimum degree



Performance of numerical factorizations

- What machines?
  - IBM RS/6000 model 530.
- Test problems?
  - Harwell-Boeing matrices.
  - Each matrix was ordered using an implementation of minimum degree with multiple eliminations.
  - Factorization times reported in CPU seconds.
- Numerical factorization?
  - left-looking column-column
  - left-looking supernode-supernode
  - right-looking multifrontal

Test matrices for numerical factorizations

problem	n	|A|	$\|L_P\|$	flops	col-col	sup-sup	sup-mf
BCSSTK13	2,003	83,883	271,671	58,550,598	7.33	3.04	3.10
BCSSTK14	1,806	63,454	112,267	9,793,431	1.32	0.61	0.65
BCSSTK15	3,948	117,816	651,222	165,035,094	20.40	8.08	8.32
BCSSTK16	4,884	290,378	741,178	149,100,948	18.61	7.47	7.52
BCSSTK18	11,948	149,090	662,725	140,907,823	17.86	8.07	8.47
BCSSTK23	3,134	45,178	420,311	119,155,247	14.71	6.00	6.26
BCSSTK24	3,562	159,910	278,922	32,429,194	4.28	1.72	1.74
NASA1824	1,824	39,208	73,699	5,160,949	0.74	0.36	0.36
NASA2910	2,910	174,296	204,403	21,068,943	2.81	1.23	1.25
NASA4704	4,704	104,756	281,472	35,003,786	4.56	1.94	1.96

Summary …

- Compare “banded” ordering and minimum degree ordering.
- Demonstrated performance of general purpose sparse SPD solvers.

Outstanding issues

- Relaxing symbolic factorization?
  - Supernodal amalgamation to enhance performance of matrix-matrix multiplication kernels, at the expense of storing and operating on some zeros?
  - 2-D partitions (block-columns and block-rows)?
  - Combining left-looking and multifrontal?

- Effective parallel implementations of ordering, symbolic factorization, and numerical factorization?