# MA5233 Computational Mathematics

## Lecture 8: Sparse LU Factorisation

Simon Etter


Semester I, AY 2020/2021

# Sparse LU Factorisation

**Introduction**

We have seen in Lecture 7 that solving partial differential equations essentially amounts to solving very large and sparse linear systems.

So far, we have solved these systems using Julia's backslash function `\` without questioning much how backslash works. Let us now change that.

Looking up the documentation of backslash (`?\`), we find the following statement.

> "For non-triangular square matrices, an LU factorization is used."

This means that `A\b` with square `A` uses the following algorithm.

**Algorithm: Solving $Ax = b$ via LU factorisation**

1. Factorise $A = LU$.
2. Solve $Ly = b$, $Ux = y$.

# Sparse LU Factorisation

**Outlook**

You should have heard about the LU factorisation and its relation to linear systems in your undergraduate studies (or even earlier).

The following slides therefore recapitulate the main ideas of the LU factorisation, but they do not provide a detailed introduction to the topic.

The purpose of this recap is to prepare you for the discussion of sparse LU factorisation, which will be the main topic of this lecture.

# Sparse LU Factorisation

**Thm: LU factorisation**

For every matrix $A \in \mathbb{R}^{n \times n}$, there exist

- a permutation matrix $P \in \mathbb{R}^{n \times n}$,
- a lower-triangular matrix $L \in \mathbb{R}^{n \times n}$ with unit diagonal, and
- an upper-triangular matrix $U \in \mathbb{R}^{n \times n}$

such that

$$PA = LU.$$

The matrices $L$ and $U$ are unique for fixed $P$.

*Proof.* Undergraduate material.

**Discussion**

I will discuss the definition and meaning of the permutation factor $P$ on slide 7. For now, let us ignore this factor and instead address the following questions.

- How do we compute $A = LU$?
- How do we solve $Ly = b$ and $Ux = y$?

# Sparse LU Factorisation

**Computing the LU factorisation**

Main ideas:

- Use the top left entry to eliminate all entries below it.
  This top left entry is called pivot.
- Use the $L$-factor to keep track of the elimination factors.

**Example**

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 1 & -2 \\ -8 & 2 & 3 \\ 12 & 7 & -5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 1 & -2 \\ 0 & 4 & -1 \\ 0 & 4 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 4 & 1 & -2 \\ 0 & 4 & -1 \\ 0 & 4 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 1 & 1 \end{pmatrix} \begin{pmatrix} 4 & 1 & -2 \\ 0 & 4 & -1 \\ 0 & 0 & 2 \end{pmatrix} = LU$$

# Sparse LU Factorisation

## Solving triangular systems

The purpose of the LU factorisation is to reduce the problem of solving arbitrary linear systems $Ax = b$ into that of solving two triangular linear system $Ly = b$ and $Ux = b$.

This is useful because triangular systems are easy to solve.

**Example**

$$\begin{pmatrix} 4 & 1 & -2 \\ 0 & 2 & -1 \\ 0 & 0 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 6 \end{pmatrix}$$

$$\begin{aligned} \text{Third eq.:} \quad & 3x_3 = 6 && \implies && x_3 = 2 \\ \text{Second eq.:} \quad & 2x_2 - x_3 = 4 && \implies && x_2 = 3 \\ \text{First eq.:} \quad & 4x_1 + x_2 - 2x_3 = 3 && \implies && x_1 = 1 \end{aligned}$$

## Def: Back substitution

The above algorithm is known as *back substitution*.

Some authors distinguish between back substitution for solving upper-triangular linear systems and forward substitution for solving lower-triangular linear systems. I call both of these algorithms back substitution.

# Sparse LU Factorisation

## Zero pivot elements

Using the top left entry to eliminate all entries below it does not work if the top left entry is zero. In such cases, we need to swap the top row with another row to obtain a nonzero pivot element.

Example:

$$\begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix} \quad \longrightarrow \quad \begin{pmatrix} 1 & 2 \\ 0 & 3 \end{pmatrix}.$$

In the language of linear algebra, this operation corresponds to replacing

$$Ax = b \qquad \text{with} \qquad (PA)\,x = Pb$$

where $P$ is a permutation matrix defined as follows.

## Def: Permutations and permutation matrices

A *permutation* is a bijective map $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ .

The *permutation matrix* associated with a permutation $\pi$ is a matrix $P \in \mathbb{R}^{n \times n}$ such that $(Pb)[i] = b[\pi(i)]$ for all $b \in \mathbb{R}^n$.

Note that $(PA)[i, j] = A[\pi(i), j]$; hence $A \mapsto PA$ indeed corresponds to swapping rows.

# Sparse LU Factorisation

**Discussion**

The previous slide explains why the LU factorisation theorem on slide 4 includes a permutation matrix factor $P$: this factor corresponds to swapping rows which is needed to avoid division by zero.

Moreover, swapping rows is also important to ensure stability of the LU factorisation with respect to rounding errors, see the example on the next slide.

# Sparse LU Factorisation

**Example**

Assume $\varepsilon \leq \mathtt{eps}()/2$ and consider the LU factorisation

$$A = \begin{pmatrix} \varepsilon & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} \varepsilon & 0 \\ \frac{1}{\varepsilon} & 1 \end{pmatrix} \begin{pmatrix} \varepsilon & 1 \\ 0 & 1 - \frac{1}{\varepsilon} \end{pmatrix} = LU.$$

The rounded factors $\tilde{L} = \mathtt{T}(L)$, $\tilde{U} = \mathtt{T}(U)$ then represent the matrix

$$\tilde{A} = \begin{pmatrix} \varepsilon & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} \varepsilon & 0 \\ \frac{1}{\varepsilon} & 1 \end{pmatrix} \begin{pmatrix} \varepsilon & 1 \\ 0 & -\frac{1}{\varepsilon} \end{pmatrix} = \tilde{L}\tilde{U},$$

which contains an $O(1)$ error compared to $A$.

If we used $\tilde{L}$, $\tilde{U}$ to solve $Ax = b$, we would hence end up solving $\tilde{A}\tilde{x} = b$.

This is the wrong linear system, hence we would obtain solutions $\tilde{x}$ which are not at all close to the solutions of $Ax = b$!

# Sparse LU Factorisation

**Example (continued)**

The $O(1)$ rounding errors observed above can be avoided if we swap the two rows of $A$: the LU factorisation then becomes

$$A = \begin{pmatrix} 1 & 1 \\ \varepsilon & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \varepsilon & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1-\varepsilon \end{pmatrix} = LU,$$

and the rounded factors $\tilde{L} = \mathtt{T}(L)$, $\tilde{U} = \mathtt{T}(U)$ then represent the matrix

$$\tilde{A} = \begin{pmatrix} 1 & 1 \\ \varepsilon & 1+\varepsilon \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \varepsilon & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \tilde{L}\tilde{U},$$

which is exact up to an $O(\varepsilon)$ error.

# Sparse LU Factorisation

**Discussion**

Virtually all software implementations of the LU factorisation attempt to avoid the blow-up of rounding errors illustrated above by choosing the row permutation matrix $P$ according to the following rule.

**Def: Column pivoting** (also known as *partial pivoting*)

Choose $P$ such that each pivot element is the element of largest absolute value in its respective column.

**Example**

$$\begin{pmatrix} 4 & 1 & -2 \\ -8 & 2 & 3 \\ 12 & 7 & -5 \end{pmatrix} \longrightarrow \text{choose } 12 \text{ as pivot element.}$$

**Discussion**

Column pivoting is partially justified by the observation on the next slide.

# Sparse LU Factorisation

**"Thm:" Stability of column-pivoted LU factorisation**

Column-pivoted LU factorisation is *usually* a stable algorithm for evaluating $A \mapsto A^{-1}b$.

**Discussion**

Recall the following definitions and results from Lecture 3.

- A mathematical function $f(x)$ is said to be *well-conditioned* if $f(\tilde{x}) \approx f(x)$ for all $\tilde{x} \approx x$.
- A numerical algorithm $\tilde{f}(x)$ for evaluating $f(x)$ is said to be *stable* if $\tilde{f}(x) \approx f(\tilde{x})$ for some $\tilde{x}$.
- If $f(x)$ is well-conditioned and $\tilde{f}(x)$ is stable, then $\tilde{f}(x) \approx f(x)$.

Stability of column-pivoted LU factorisation would hence be a very powerful result, but unfortunately it is not quite true: there are well-conditioned linear systems $Ax = b$ such that the solution $\tilde{x}$ computed via column-pivoted LU factorisation in floating-point arithmetic is not at all close to the exact solution, see `wilkinson()`.

The good news is that no such linear system has ever been observed in practice; hence the "usually" in the above statement.

# Sparse LU Factorisation

**Wilkinson's bus**

The fact that column-pivoted LU factorisation may fail to be stable puts computational mathematicians in a somewhat uncomfortable position: LU factorisation is the fastest known algorithm for solving linear systems and works perfectly well for all practical purposes, yet using it without additional mathematical analysis and/or extra mechanisms for detecting numerical instability in principle involves a small risk that the computed result might be completely inaccurate.

The approach taken by most computational scientists is to not worry about this small risk and simply pretend that column-pivoted LU factorisation is stable. After all, thousands if not millions of scientists have been solving large-scale linear system using the LU factorisation for decades, so even if there were any practical issues with this algorithm, it would be very unlikely that you are the first one to discover them.

This approach has been summarised by James Wilkinson, one of the pioneers of numerical analysis, as follows:

> "Anyone that unlucky has already been run over by a bus."

# Sparse LU Factorisation

**Discussion**

The above discussion may be summarised as follows.

- LU factorisaction requires column pivoting to ensure stability.
- Column-pivoted LU factorisation is stable for all practical purposes, but we do not have a proof for that.

These statements are correct for general linear systems $Ax = b$, but of course they do not rule out that there may be linear systems for which no pivoting is necessary and for which we can prove stability of the LU factorisation.

A famous class of matrices for which unpivoted LU factorisation is provably stable are the symmetric positive definite matrices.

Symmetric of course means $A^T = A$, and positive definite is defined as follows.

**Def: Positive definite matrix**

$A \in \mathbb{R}^{n \times n}$ is called *positive definite* if $v^T Av > 0$ for all $v \in \mathbb{R}^n \setminus \{0\}$.

One can show that a matrix is positive definite if and only if all its eigenvalues are positive. Hence $-\Delta_n^{(d)}$ is positive definite.

# Sparse LU Factorisation

**Thm: LU factorisation for symmetric positive definite matrices**

Assume $A \in \mathbb{R}^{n \times n}$ is symmetric positive definite and denote by $\tilde{x}$ the solution to the linear system $Ax = b$ computed via unpivoted LU factorisation. Then, there exists a $\tilde{A} \in \mathbb{R}^{n \times n}$ such that

$$\tilde{A}\tilde{x} = b \qquad \text{and} \qquad \frac{|\tilde{A} - A|}{|A|} = O(\text{eps}()) \text{ uniformly in } n.$$

Absolute values $|A|$, divisions $A/B$ and comparisons $A = c$ are all meant elementwise.

*Proof.* Omitted. See Higham (2002), Accuracy and Stability of Numerical Algorithms, Thm. 9.14 if you are interested.

# Sparse LU Factorisation

**Thm: Runtime of solving linear systems via LU factorisation**

Let $A \in \mathbb{R}^{n \times n}$ by a general dense matrix. Then:

- Computing $A = LU$ requires $O(n^3)$ floating-point operations.
- Solving $Ly = b$ and $Ux = y$ requires $O(n^2)$ floating-point operations.

*Proof.* Undergraduate material (and also a good exercise).

**Observations**

- Computing the LU factorisation is much more expensive than solving the triangular systems; hence if we need to solve multiple linear systems $Ax_k = b_k$ with the same matrix $A$, we should make sure we compute $LU = A$ only once.
- $O(n^3)$ scaling is the worst type of scaling to occur reasonably frequently in computational linear algebra. In practice, it usually means that there is a critical matrix size $n$ up to which the operation runs reasonably fast but beyond which the operation becomes unbearably slow, see `time_lse();`

  This phenomenon is sometimes called the *cubic scaling wall*.

# Sparse LU Factorisation

**LU factorisation for solving the 2d Poisson equation**

In the case of the two-dimensional Poisson equation $-\Delta^{(2)}_{n_{\mathrm{FD}}}\, u_{n_{\mathrm{FD}}} = f$, the matrix size $n_{\mathrm{LSE}}$ scales quadratically in the discretisation parameter $n_{\mathrm{FD}}$.

If we were to use the LU factorisation as described above to solve the finite-difference-discretised Poisson equation $-\Delta^{(2)}_{n_{\mathrm{FD}}}\, u_{n_{\mathrm{FD}}} = f$, the overall runtime would thus be

$$O(n_{\mathrm{LSE}}^3) = O\big((n_{\mathrm{FD}}^2)^3\big) = O\big(n_{\mathrm{FD}}^6\big),$$

which is completely unacceptable.

For example, if we were to solve the Poisson equation with $n_{\mathrm{FD}} = 300$ on a machine where each operation takes about 1 nanosecond, we would expect the LU factorisation to take

$$O(300^6)\,\mathrm{FLOP} \times O(10^{-9})\,\tfrac{\mathrm{seconds}}{\mathrm{FLOP}} \quad \approx \quad O(10)\,\mathrm{days},$$

but in Lecture 7 we have seen that backslash solves $-\Delta^{(2)}_{n_{\mathrm{FD}}}\, u_{n_{\mathrm{FD}}} = f$ with $n_{\mathrm{FD}} = 300$ in a fraction of a second.

# Sparse LU Factorisation

**LU factorisation for solving the 2d Poisson equation (continued)**

The reason why backslash is so much faster than suggested by the undergraduate-level theory is of course because backslash exploits the sparsity of $\Delta_n^{(2)}$.

However, doing so is not as straightforward as one might think because the sparsity patterns of $A$ and $L + U$ do not agree in general.

**Outlook**

The first half of this lecture will present a theory which allows us to predict the sparsity pattern of $L$ and $U$ given the sparsity pattern of $A$.

The second half will demonstrate the application of this theory to the discrete Poisson equation $-\Delta_n^{(d)}\, u_n = f$.

I will begin this journey on the next slide by discussing a simple example which illustrates some of the complications arising in sparse LU factorisations.

# Sparse LU Factorisation

**Example**

$$\begin{pmatrix} 1 & & 1 & \\ & 1 & 1 & \\ & & 1 & \\ 1 & 1 & & 1 \end{pmatrix} = \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ 1 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} 1 & & 1 & \\ & 1 & 1 & \\ & & 1 & \\ & & & 1 \end{pmatrix},$$

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

Observations:

- $A[i,j] = 0$ does not imply $L[i,j] = 0$ or $U[i,j] = 0$.
- Some entries of $L, U$ are zero regardless of the values we assign to the nonzero entries of $A$. Others may be zero or nonzero depending on the values in $A$.

# Sparse LU Factorisation

**Terminology: Sparsity pattern**

The set of all indices $(i, j) \in \{1, \ldots, n\}^2$ such that $A[i, j] \neq 0$ is called the *sparsity pattern* or *structure* of a matrix $A \in \mathbb{R}^{n \times n}$.

Sparsity patterns are conveniently described by drawing a matrix where each zero entry is left empty and each nonzero entry is marked with a bullet ($\bullet$), see middle matrix below.

In all examples considered in this lecture, the diagonal will always be nonzero. I use this fact to write the numbers 1 to $n$ on the diagonal rather than bullets, see right matrix. Doing so makes the sparsity patterns easier to read and talk about.

**Example**

$$\begin{pmatrix} 3 & 0 & 2 \\ 5 & 1 & 0 \\ 0 & 0 & 4 \end{pmatrix} = \left(\begin{array}{c|c|c} \bullet & & \bullet \\ \hline \bullet & \bullet & \\ \hline & & \bullet \end{array}\right) = \left(\begin{array}{c|c|c} 1 & & \bullet \\ \hline \bullet & 2 & \\ \hline & & 3 \end{array}\right).$$

# Sparse LU Factorisation

**Terminology: Structural nonzero**

Let $A \in \mathbb{R}^{n \times n}$ be a matrix and let $B \in \mathbb{R}^{n \times n}$ be some matrix derived from $A$ (e.g. $B = A^2$, $B = A^{-1}$, or $B$ is one of the LU factors).

An entry $B[i, j]$ of $B$ is called *structurally nonzero* if $B[i, j]$ could be nonzero given the sparsity pattern of $A$.

**Example**

Consider

$$A = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}, \qquad A^2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

- $A^2[2,1] = 0 \times 1 + -1 \times 0$ is structurally zero because the only way to make this number nonzero is to assign a nonzero value to $A[2,1] = 0$ and doing so would change the sparsity pattern of $A$.
- $A^2[1,2] = 1 \times 1 + 1 \times -1$ is structurally nonzero because we could make this number nonzero e.g. by changing $A[1,1] = 1$ to $A[1,1] = 2$ and doing so would not change the sparsity pattern of $A$.

# Sparse LU Factorisation

**Terminology: Derived sparsity pattern**

Let $A \in \mathbb{R}^{n \times n}$ be a matrix and let $B \in \mathbb{R}^{n \times n}$ be some matrix derived from $A$ (e.g. $B = A^2$, $B = A^{-1}$, or $B$ is one of the LU factors).

The sparsity pattern / structure of $B$ then refers to the set of indices $(i, j) \in \{1, \ldots, n\}^2$ such that $B[i, j]$ is *structurally nonzero*.

From now on, all statements of the form $B[i, j] \neq 0$ are meant in the structural sense.

**Terminology: Derived sparsity pattern**

An index $(i, j) \in \text{structure}(B) \setminus \text{structure}(A)$ is called a *fill-in entry* of $B$.

**Example**

Consider

$$A = \begin{pmatrix} 1 & & \\ \bullet & 2 & \\ & \bullet & 3 \end{pmatrix}, \qquad A^2 = \begin{pmatrix} 1 & & \\ \bullet & 2 & \\ \bullet & \bullet & 3 \end{pmatrix}.$$

$B[3, 1]$ is a fill-in entry because $A[3, 1] = 0$ but $B[3, 1] \neq 0$.

# Sparse LU Factorisation

**Remark: Ambiguity of structure for derived matrices**

The above definition of structure is ambiguous because any derived matrix is also a matrix and hence we have two definitions which in general lead to different results.

I resolve this ambiguity by imposing that the derived definition is to be preferred whenever it is applicable.

**Remark**

The common theme in the terminology introduced above is that we only distinguish between whether an entry is zero or nonzero, or whether an entry must be zero or could be nonzero for derived matrices.

Reasons for doing so include:

- Reasoning about structure is easier than reasoning about values.
- structure($B$) provides a worst-case estimate for how much memory will be needed to store $B$
- Cancellation (i.e. a structurally nonzero entry being zero) is rare.

# Sparse LU Factorisation

**Outlook**

Being able to predict the amount of fill-in is crucial in many applications, and it turns out that the easiest way to do so is to relate sparse matrices with graphs and then deduce the locations of fill-in entries based on whether and how some vertices are connected.

The next slide introduces the aforementioned matrix graphs, and the slides after that present some first results which illustrate how fill-in entries are related to graph properties.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \qquad \longleftrightarrow \qquad G(A) = \textcircled{1} \quad \textcircled{2} \quad \textcircled{3} \quad \textcircled{4}$$

$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

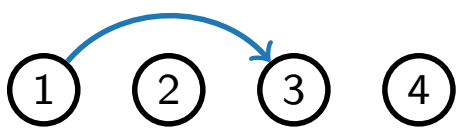
$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \quad$$




$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

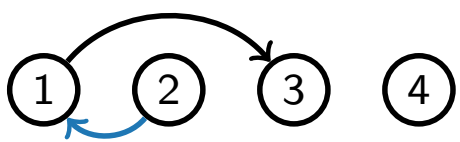
$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \quad$$




$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

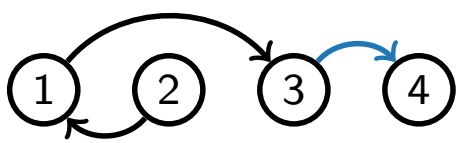
$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \quad$$




$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

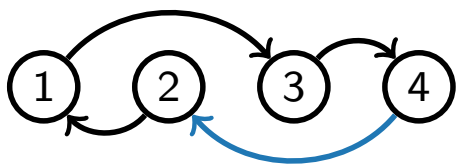
$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) =$$




$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

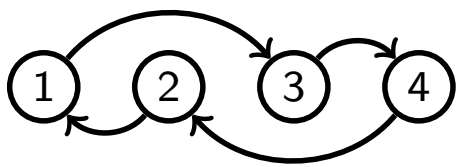
$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \qquad \longleftrightarrow \qquad G(A) =$$




$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Def: Graph of a sparse matrix** $A \in \mathbb{R}^{n \times n}$

Graph $G(A) = \big(V(A), E(A)\big)$ given by

$$V(A) = \{1, \ldots, n\}, \qquad E(A) = \big\{ j \to i \mid A[i,j] \neq 0 \big\}.$$

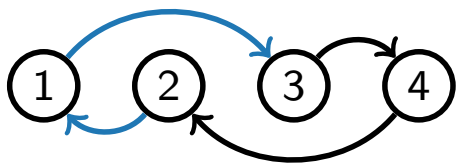
$V(A)$ denotes the set of vertices, $E(A)$ denotes the set of edges.
Note the transpose in $E(A)$: entry $A[i,j]$ corresponds to the edge $j \to i$.

**Def: Path in** $G = (V, E)$

Ordered sequence $k_0, \ldots, k_p \in V$ such that $k_{q-1} \to k_q \in E$ for all $q$.
The number of edges $p$ is called the length of the path.

**Example** (Numbers and • indicate nonzeros in $A$.)

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \qquad \longleftrightarrow \qquad G(A) =$$




$2 \to 1 \to 3$ is a path of length 2.

Note that I do not draw the diagonal edges $i \to i$ to keep the graph readable.

# Sparse LU Factorisation

**Path theorem for matrix powers**

$$A^p[i,j] \neq 0 \quad \Longleftrightarrow \quad \exists \text{ path } j \to i \text{ of length } p.$$

*Proof.*

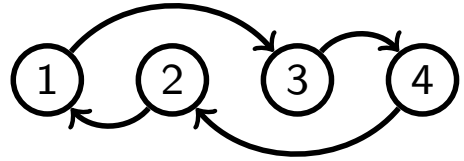
$$A^p[i,j] = \sum_{k_{p-1}} \cdots \sum_{k_1} A[i,k_{p-1}] \ \ldots \ A[k_a,k_{a-1}] \ \ldots \ A[k_1,j].$$

Each term is nonzero iff $j \to k_1 \to \ldots \to k_{p-1} \to i$ is a path in $G(A)$.

# Sparse LU Factorisation

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$A^2 = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \bullet \\ \hline & 2 & \bullet & \bullet \\ \hline \bullet & \bullet & 3 & \\ \hline \bullet & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \quad$$




We observe:

- $A^2[4,1] \neq 0$ because there is a path $1 \to 4$ of length 2, namely $1 \to 3 \to 4$.
- $A^2[2,1] = 0$ because the only path $1 \to 2$, namely $1 \to 3 \to 4 \to 2$, is of length 3.

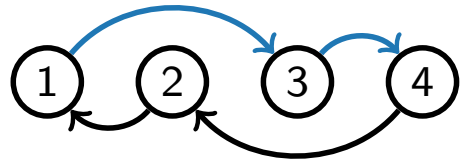
All other entries of $A^2$ can be verified analogously.

Note that nonzeros of $A$ are also nonzeros of $A^2$ since we can extend paths of length 1 to paths of length 2 by adding diagonal edges $i \to i$.

# Sparse LU Factorisation

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$A^2 = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \bullet \\ \hline & 2 & \bullet & \bullet \\ \hline \bullet & \bullet & 3 & \\ \hline \bullet & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \quad$$




We observe:

- $A^2[4,1] \neq 0$ because there is a path $1 \to 4$ of length 2, namely $1 \to 3 \to 4$.
- $A^2[2,1] = 0$ because the only path $1 \to 2$, namely $1 \to 3 \to 4 \to 2$, is of length 3.

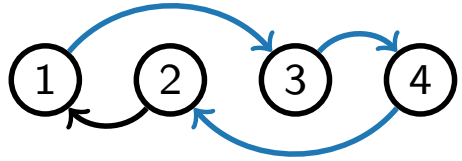
All other entries of $A^2$ can be verified analogously.

Note that nonzeros of $A$ are also nonzeros of $A^2$ since we can extend paths of length 1 to paths of length 2 by adding diagonal edges $i \to i$.

# Sparse LU Factorisation

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$A^2 = \begin{pmatrix} 1 & \bullet & & \bullet \\ & 2 & \bullet & \bullet \\ \bullet & \bullet & 3 & \\ \bullet & & \bullet & 4 \end{pmatrix} \quad \longleftrightarrow \quad G(A) =$$




We observe:

- $A^2[4,1] \neq 0$ because there is a path $1 \to 4$ of length 2, namely $1 \to 3 \to 4$.
- $A^2[2,1] = 0$ because the only path $1 \to 2$, namely $1 \to 3 \to 4 \to 2$, is of length 3.

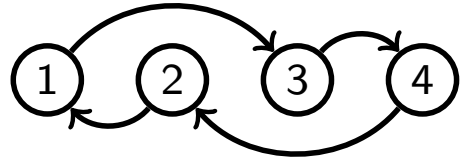
All other entries of $A^2$ can be verified analogously.

Note that nonzeros of $A$ are also nonzeros of $A^2$ since we can extend paths of length 1 to paths of length 2 by adding diagonal edges $i \to i$.

# Sparse LU Factorisation

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$A^2 = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \bullet \\ \hline & 2 & \bullet & \bullet \\ \hline \bullet & \bullet & 3 & \\ \hline \bullet & & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \quad$$




We observe:

- $A^2[4,1] \neq 0$ because there is a path $1 \to 4$ of length 2, namely $1 \to 3 \to 4$.
- $A^2[2,1] = 0$ because the only path $1 \to 2$, namely $1 \to 3 \to 4 \to 2$, is of length 3.

All other entries of $A^2$ can be verified analogously.

Note that nonzeros of $A$ are also nonzeros of $A^2$ since we can extend paths of length 1 to paths of length 2 by adding diagonal edges $i \to i$.

# Sparse LU Factorisation

**Path theorem for inverse**

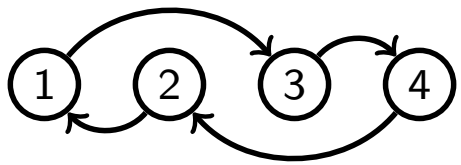
$$A^{-1}[i,j] \neq 0 \quad \Longleftrightarrow \quad \exists \text{ path } j \to i.$$

*Proof (not examinable).*

- $A^{-1} = p(A)$ for polynomial $p(x)$ interpolating $\frac{1}{x}$ on eigenvalues of $A$.
- Hence $A^{-1}[i,j] = \left(\sum_{p=0}^{n-1} c_p\, A^p\right)[i,j]$ is nonzero if there is a path $j \to i$ of arbitrary length.

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$A^{-1} = \left(\begin{array}{c|c|c|c} 1 & \bullet & \bullet & \bullet \\ \hline \bullet & 2 & \bullet & \bullet \\ \hline \bullet & \bullet & 3 & \bullet \\ \hline \bullet & \bullet & \bullet & 4 \end{array}\right) \qquad \longrightarrow \qquad G(A) = \quad$$




$A^{-1}[2,1] \neq 0$ because there is a path $1 \to 2$, namely $1 \to 3 \to 4 \to 2$.
All other entries of $A^{-1}$ can be verified analogously.

# Sparse LU Factorisation

**Path theorem for inverse**

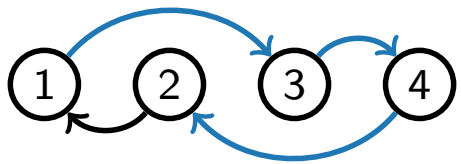
$$A^{-1}[i,j] \neq 0 \quad \Longleftrightarrow \quad \exists \text{ path } j \to i.$$

*Proof (not examinable).*

- $A^{-1} = p(A)$ for polynomial $p(x)$ interpolating $\frac{1}{x}$ on eigenvalues of $A$.
- Hence $A^{-1}[i,j] = \left(\sum_{p=0}^{n-1} c_p A^p\right)[i,j]$ is nonzero if there is a path $j \to i$ of arbitrary length.

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$A^{-1} = \left(\begin{array}{c|c|c|c} 1 & \bullet & \bullet & \bullet \\ \hline \bullet & 2 & \bullet & \bullet \\ \hline \bullet & \bullet & 3 & \bullet \\ \hline \bullet & \bullet & \bullet & 4 \end{array}\right) \quad \longrightarrow \quad G(A) = \quad$$



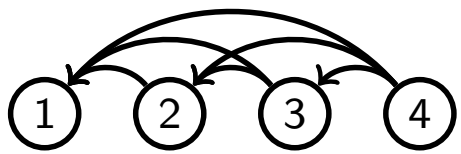

$A^{-1}[2,1] \neq 0$ because there is a path $1 \to 2$, namely $1 \to 3 \to 4 \to 2$.
All other entries of $A^{-1}$ can be verified analogously.

# Sparse LU Factorisation

**Corollaries of path theorem for inverses**

- If $G(A)$ is connected (there exists a path between any pair of vertices), then $A^{-1}$ is dense.
- If $G(A)$ is disconnected, i.e. $A = \begin{pmatrix} A_{11} & 0 \\ 0 & A_{22} \end{pmatrix}$, then $A^{-1} = \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & A_{22}^{-1} \end{pmatrix}$.
- Inverse of upper/lower triangular matrix is upper/lower triangular.

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & \bullet & \bullet \\ \hline & 2 & \bullet & \bullet \\ \hline & & 3 & \bullet \\ \hline & & & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) =$$



# Sparse LU Factorisation

**Discussion**

The path theorems for matrix powers and inverses are sometimes useful for theoretical purposes.

For example, the path theorem for inverses tells us that $(\Delta_n^{(d)})^{-1}$ is dense, which shows that every entry of $u_n$ depends on every entry of $f$. Physically, we can interpret this as saying that if you turn the heat on somewhere, then the temperatures changes everywhere (but maybe only by a very small amount).

The path theorems for matrix powers and inverses are not used very often in applications, however. This is due to the following reason.

- Whenever matrix powers and inverses arise, we almost always immediately multiply them by some vector $b \in \mathbb{R}^n$.
- It is more efficient to compute $p(A)\,b$ or $A^{-1}b$ directly rather than evaluate the matrix functions first.
- The sparsity of $p(A)$ or $A^{-1}$ then has no impact on performance because we never actually store $p(A)$ or $A^{-1}$.

I will next present a similar path theorem for predicting the fill-in in the LU factorisation, and this result is important in applications because the need to evaluate LU factorisations arises all the time.

# Sparse LU Factorisation

**Discussion (continued)**

Note that when we are talking about the LU factorisation, we are strictly speaking talking about two matrices $L$ and $U$ and correspondingly about two sparsity patterns.

However, the statements regarding the sparsity patterns of the two matrices are exactly the same; hence I will avoid repeating the same statement twice by talking about the sparsity pattern of $L + U$ rather than each matrix separately. Doing so does not discard any information because we have

$$(L+U)[i,j] \neq 0 \quad \Longleftrightarrow \quad \begin{cases} L[i,j] \neq 0 & \text{if } i \geq j, \\ U[i,j] \neq 0 & \text{if } i \leq j. \end{cases}$$

We do not have to worry about cancellation between diagonal entries $L[i,i]$, $U[i,i]$ because $\neq 0$ is meant in the structural sense here.

# Sparse LU Factorisation

**Def: Fill path**

Path $j \to k_1 \to \ldots \to k_p \to i$ in $G(A)$ such that $k_1, \ldots, k_p < \min\{i, j\}$.

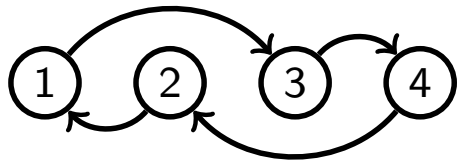
**Fill path theorem**

$$(L+U)[i,j] \neq 0 \quad \Longleftrightarrow \quad \exists \text{ fill path } j \to i.$$

*Proof (not examinable).* See next two slides.

**Example** (Numbers and • indicate nonzeros in $A$. • indicates fill-in.)

$$L+U = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & \bullet & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \qquad \longleftrightarrow \qquad G(A) =$$




We observe:

- $L[3,2] \neq 0$ because there is a fill path $2 \to 3$, namely $2 \to 1 \to 3$ (note that $1 < \min\{1,3\}$).
- $L[4,1] = 0$ because the only path $1 \to 4$, namely $1 \to 3 \to 4$, is not a fill path because $3 \not< \min\{1,4\}$.

# Sparse LU Factorisation

**Def: Fill path**

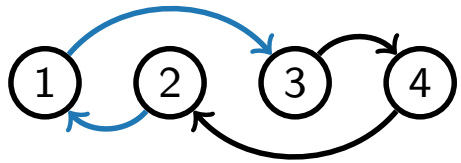
Path $j \to k_1 \to \ldots \to k_p \to i$ in $G(A)$ such that $k_1, \ldots, k_p < \min\{i, j\}$.

**Fill path theorem**

$$(L + U)[i, j] \neq 0 \quad \iff \quad \exists \text{ fill path } j \to i.$$

*Proof (not examinable).* See next two slides.

**Example** (Numbers and • indicate nonzeros in $A$. • indicates fill-in.)

$$L + U = \begin{pmatrix} 1 & \bullet & & \\ & 2 & & \bullet \\ \bullet & \bullet & 3 & \\ & & \bullet & 4 \end{pmatrix} \quad \longleftrightarrow \quad G(A) = \text{  }$$


We observe:

- $L[3, 2] \neq 0$ because there is a fill path $2 \to 3$, namely $2 \to 1 \to 3$ (note that $1 < \min\{1, 3\}$).
- $L[4, 1] = 0$ because the only path $1 \to 4$, namely $1 \to 3 \to 4$, is not a fill path because $3 \not< \min\{1, 4\}$.

# Sparse LU Factorisation

**Def: Fill path**

Path $j \to k_1 \to \ldots \to k_p \to i$ in $G(A)$ such that $k_1, \ldots, k_p < \min\{i, j\}$.

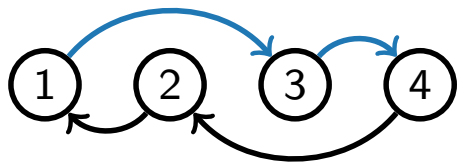
**Fill path theorem**

$$(L+U)[i,j] \neq 0 \quad \iff \quad \exists \text{ fill path } j \to i.$$

*Proof (not examinable).* See next two slides.

**Example** (Numbers and ● indicate nonzeros in $A$. ● indicates fill-in.)

$$L+U = \left(\begin{array}{c|c|c|c} 1 & \bullet & & \\ \hline & 2 & & \bullet \\ \hline \bullet & \bullet & 3 & \\ \hline & & \bullet & 4 \end{array}\right) \qquad \longleftrightarrow \qquad G(A) =$$




We observe:

- $L[3,2] \neq 0$ because there is a fill path $2 \to 3$, namely $2 \to 1 \to 3$ (note that $1 < \min\{1,3\}$).
- $L[4,1] = 0$ because the only path $1 \to 4$, namely $1 \to 3 \to 4$, is not a fill path because $3 \not< \min\{1,4\}$.

# Sparse LU Factorisation

**Lemma for fill path theorem (not examinable)**

Let $i, j \in \{1, \ldots, n\}$ and set $\ell = \{1, \ldots, \min\{i,j\} - 1\}$. Then,

$$U[i,j] = A[i,j] - A[i,\ell]\, A[\ell,\ell]^{-1}\, A[\ell,j] \qquad \text{for } i \le j,$$
$$L[i,j]\, U[j,j] = A[i,j] - A[i,\ell]\, A[\ell,\ell]^{-1}\, A[\ell,j] \qquad \text{for } i \ge j.$$

*Proof.* Consider the block LU factorisation with $\bar{r} = \{\min\{i,j\}, \ldots, n\}$,

$$\begin{pmatrix} A[\ell,\ell] & A[\ell,\bar{r}] \\ A[\bar{r},\ell] & A[\bar{r},\bar{r}] \end{pmatrix} = \begin{pmatrix} I & \\ A[\bar{r},\ell]\, A[\ell,\ell]^{-1} & I \end{pmatrix} \begin{pmatrix} A[\ell,\ell] & A[\ell,\bar{r}] \\ & A[\bar{r},\bar{r}] - A[\bar{r},\ell]\, A[\ell,\ell]^{-1}\, A[\ell,\bar{r}] \end{pmatrix}.$$

Let $\quad L_1 U_1 = A[\ell,\ell], \quad L_2 U_2 = A[\bar{r},\bar{r}] - A[\bar{r},\ell]\, A[\ell,\ell]^{-1}\, A[\ell,\bar{r}].$

The full factorisation is then given by

$$\begin{pmatrix} A[\ell,\ell] & A[\ell,\bar{r}] \\ A[\bar{r},\ell] & A[\bar{r},\bar{r}] \end{pmatrix} = \begin{pmatrix} L_1 & \\ A[\bar{r},\ell]\, A[\ell,\ell]^{-1} L_1 & L_2 \end{pmatrix} \begin{pmatrix} U_1 & L_1^{-1}\, A[\ell,\bar{r}] \\ & U_2 \end{pmatrix}.$$

The claim follows by noting that $L[i,j] = L_2[i,j]$ and $U[i,j] = U_2[i,j]$ have the given form.

# Sparse LU Factorisation

**Fill path theorem (repeated from earlier slide)**

$$(L+U)[i,j] \neq 0 \quad \Longleftrightarrow \quad \exists \text{ fill path } j \to i.$$

*Proof (not examinable).* According to the lemma on the previous slide, we have

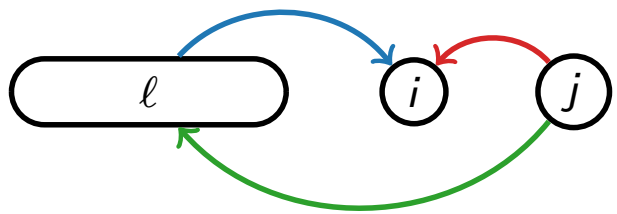
$$U[i,j] = A[i,j] - A[i,\ell]\, A[\ell,\ell]^{-1}\, A[\ell,j] \qquad \text{for } i \leq j,$$
$$L[i,j]\, U[j,j] = A[i,j] - A[i,\ell]\, A[\ell,\ell]^{-1}\, A[\ell,j] \qquad \text{for } i \geq j.$$

1st term makes $U[i,j]$ nonzero if there is a fill path $j \to i$ of length 1.
2nd term makes $U[i,j]$ nonzero if there is a fill path $j \to i$ of length $> 1$.

Same arguments apply for $L[i,j]\, U[j,j]$, and we have

$$L[i,j]\, U[j,j] \neq 0 \iff L[i,j]$$

since $U[j,j]$ is a pivot element and hence must necessarily be $\neq 0$.



# Sparse LU Factorisation

**Discussion**

The fill path theorem is important for two reasons.

- It allows us to predict the number and location of fill-in entries.

  This simplifies memory management and enables us to a-priori decide whether we have the memory capacity and patience to compute the LU factorisation.
- It allows us to transform matrices such that their LU factorisations incur as little fill-in as possible.

The next slide presents an example illustrating the second point.

# Sparse LU Factorisation

**Example**

Consider the two linear systems

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & \\ 1 & & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \Longleftrightarrow \quad \begin{pmatrix} 3 & & 1 \\ & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ x_2 \\ x_1 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}.$$

These two systems have exactly the same solution $(x_1, x_2, x_3)$, but the LU factorisation of the first coefficient matrix incurs fill-in,

$$A_1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & \\ 1 & & 3 \end{pmatrix} = \begin{pmatrix} 1 & & \\ 1 & 1 & \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 \\ & 1 & -1 \\ & & 1 \end{pmatrix},$$

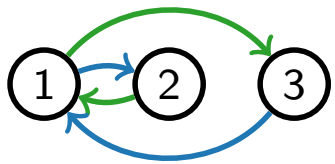
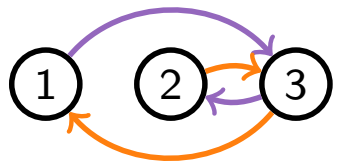
while the LU factorisation of the second coefficient matrix does not,

$$A_2 = \begin{pmatrix} 3 & & 1 \\ & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & & \\ 0 & 1 & \\ \frac{1}{3} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 3 & 0 & 1 \\ & 2 & 1 \\ & & \frac{1}{6} \end{pmatrix}.$$

# Sparse LU Factorisation

**Example (continued)**

We can explain this phenomenon by drawing the graphs

$G(A_1) =$  and $G(A_2) =$ 

and observing the following:

- $G(A_1)$ has two fill paths, namely $3 \to 1 \to 2$ and $2 \to 1 \to 3$.
- $G(A_2)$ has no fill paths: $1 \to 3 \to 2$ and $2 \to 3 \to 1$ are the only paths of length $> 1$, and they are not fill paths because $3 > \min\{1, 2\}$.

# Sparse LU Factorisation

**Fill-in-reducing permutations**

The above example shows that it is sometimes possible to reduce the amount of fill-in generated by the LU factorisation by replacing

$$Ax = b \qquad \text{with} \qquad (PAQ)(Q^{-1}x) = Pb$$

for some suitably chosen permutation matrices $P$ and $Q$.

As indicated, the row permutation $P$ and column permutation $Q$ could in general be different, but both in the above example and throughout this lecture I will only consider $P = Q$. This is because all the matrices $A$ considered in the remainder of this lecture are symmetric, and symmetry is a property that we do not want to give up.

# Sparse LU Factorisation

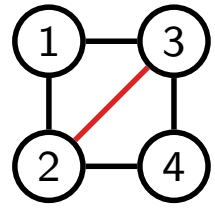
## Discussion

In the above example, we managed to eliminate fill-in altogether, but the below example shows that this is not always possible.

## Example

Consider the matrix

$$A = \left(\begin{array}{c|c|c|c} 1 & \bullet & \bullet & \\ \hline \bullet & 2 & \bullet & \bullet \\ \hline \bullet & \bullet & 3 & \bullet \\ \hline & \bullet & \bullet & 4 \end{array}\right) \quad \longleftrightarrow \quad G(A) = \begin{array}{c} \\ \\ \\ \\ \end{array}$$




For better readability, I replaced each pair of directed edges  with a single undirected edge  in this picture. Red bullets and edges represent fill-in as usual.

The LU factorisation of $PAP$ involves fill-in for any permutation matrix $P$:

For $P = I$, the fill-in occurs when eliminating the first column, and since all vertices in $G(A)$ are the same up to the (arbitrary) numbers assigned to them, the same must be true for any $P$.

# Sparse LU Factorisation

**Choosing $P$**

The above examples raise the question whether there is an efficient algorithm for determining $P$ such that the amount of fill-in is as small as possible.

Unfortunately, the answer is no: one can show that determining the optimal $P$ is an NP-complete problem.

NP-completeness is a concept from theoretical computer science and loosely speaking means that we generally cannot do better than to go through all permutation matrices $P$, compute their fill-in and pick the one matrix $P$ which results in the least amount of fill-in.

There are $n!$ permutation matrices $P \in \mathbb{R}^{n \times n}$; hence going through all of them is not feasible except for very small $n$. Instead, $P$ is usually chosen using a heuristic strategy like the one shown on the next slide.

# Sparse LU Factorisation

**Def: Approximate minimum degree (AMD) order**

Iteratively eliminate the vertex which at this point in the elimination process has the lowest degree (number of neighbours).

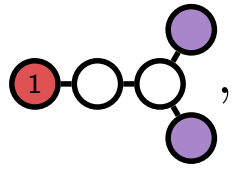
"Eliminate $v$ in step $k$" means to choose the permutation matrix $P$ such that the row and column associated with vertex $v$ ends up in position $k$.

The below examples will further clarify and motivate this definition.

# Sparse LU Factorisation

**Example**

Consider the graph

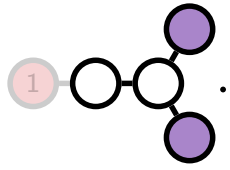
$G =$ 

,

where each undirected edge ○—○ represents a pair of directed edges ○⇄○.

All three vertices marked in red and purple have degree 1 and could therefore be eliminated first according to the minimum degree rule. I arbitrarily chose to eliminate the red vertex first, i.e. to assign it the number 1.

Since the red vertex now has the lowest possible number, the remaining vertices must be numbered higher than 1. The remaining vertices could therefore be connected by fill paths passing through 1, but this is not the case in this specific example. I indicate this by introducing a copy $G_1$ of $G$ where the red vertex has been removed,

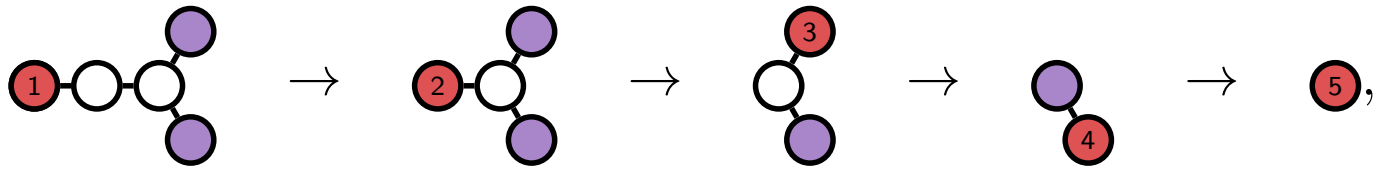
$G_1 =$ 

.

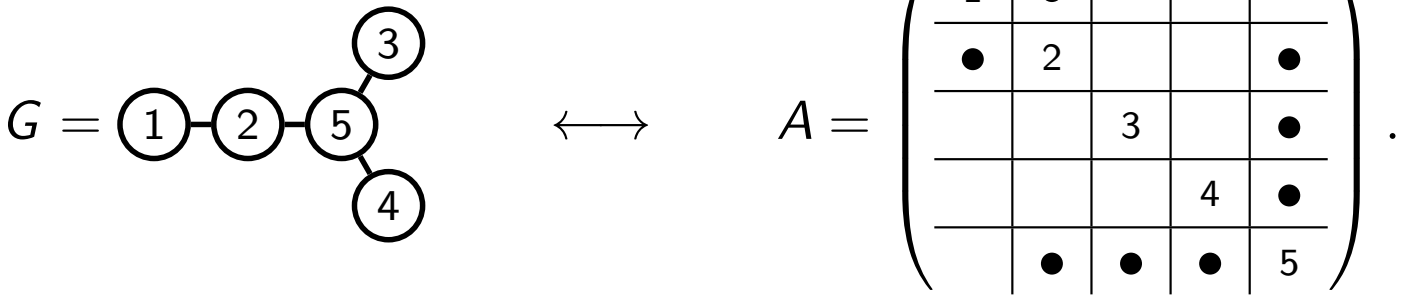
# Sparse LU Factorisation

**Example (continued)**

Repeating this process, I obtain the sequence of graphs




which results in an ordered graph $G$ and a corresponding sparsity pattern $A$ given by

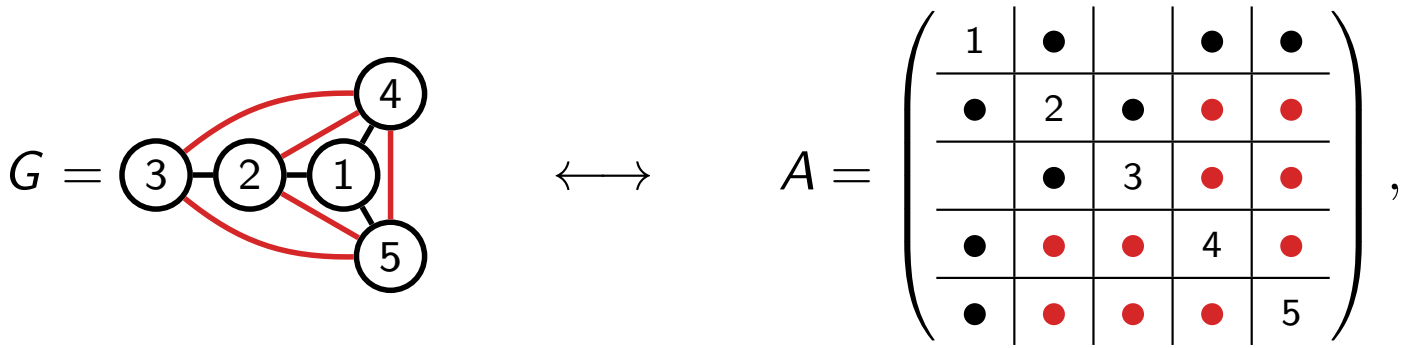



We observe that for this graph, minimum degree ordering results in an LU factorisation which involves no fill-in!

# Sparse LU Factorisation

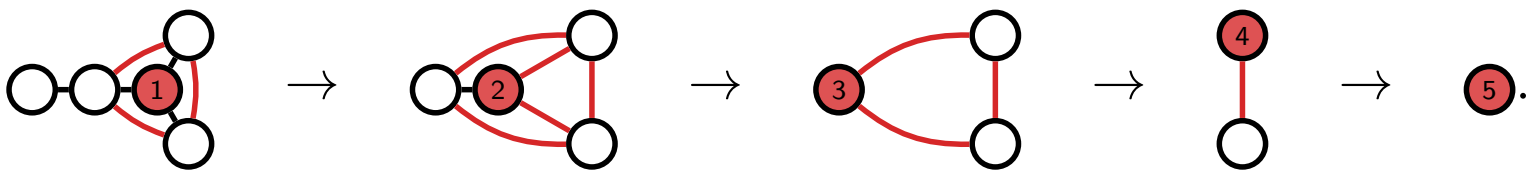
**Example (continued)**

To see that this is a nontrivial achievement, consider the alternative elimination order




which leads to the fill-in indicated in red.

This elimination order yields the sequence of reduced graphs




In this sequence, I added red edges whenever it is clear that these two vertices will be fill-path connected in the final order regardless of the numbers that we assign to the remaining vertices.

I will call these edges *fill path edges* in the following.

# Sparse LU Factorisation

**Discussion**

Fill path edges can be determined using the following result.

**Lemma**

Eliminating a vertex $v$ fill-path-connects all its fill path neighbours.

*Proof.* Consider the three vertices 

, where the red edges indicate that we have fill paths $j \to \ldots \to k$ and $k \to \ldots \to i$, and the red colouring of vertex $k$ indicates that it is about to be eliminated.

Eliminating $k$ means that $k$ will be numbered less than $i$ and $j$; hence $j \to \ldots \to k \to \ldots i$ is a fill path from $j$ to $i$ and $j, i$ will be fill-path-connected after the elimination.

# Sparse LU Factorisation

**Why minimum degree?**

The above lemma implies that eliminating a vertex of degree $d$ introduces to at most $\binom{d}{2} = \frac{d\,(d-1)}{2}$ additional fill-in edges; hence eliminating the vertices of lowest degree first corresponds to greedily minimising the amount of fill-in introduced in each elimination step.

The number of newly introduced fill-in edges is *at most* $\binom{d}{2}$ because some of the neighbours may already be (fill-path-)connected.

This motivates why minimimum degree ordering is a reasonable heuristic for minimising the amount of fill-in. Note however that it is only a heuristic: there is no guarantee that the amount of overall fill-in could not be lower if we had accepted a little bit of extra fill-in in early steps.

**Why approximate minimum degree (AMD)?**

It turns out that updating the vertex degrees after each elimination step is computationally expensive; most software packages therefore proceed by ordering vertices based on a more easily computed approximation of the vertex degrees.

# Sparse LU Factorisation

**Outlook**

I will next demonstrate how the theoretical tools developed above allow us to understand the performance of LU factorisation applied to the discrete Laplacian matrix $-\Delta_n^{(d)}$.

In one dimension, we can quite easily show the following result.

**Thm: Runtime and memory of LU($\Delta_n^{(1)}$)**

LU factorisation of $\Delta_n^{(1)}$ requires $O(n)$ runtime and memory.

*Proof.* See next slide.

# Sparse LU Factorisation

*Proof of $O(n)$ runtime and memory of $LU(\Delta_n^{(1)})$.*

Recall from Lecture 7 that $\Delta_n^{(1)}$ is of the form

$$\Delta_n^{(1)} = (n+1)^2 \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & 1 & \ddots & \ddots & \\ & & \ddots & \ddots & 1 \\ & & & 1 & -2 \end{pmatrix}.$$

The graph associated with this matrix is

$$G\left(\Delta_n^{(1)}\right) = \quad 1 \quad 2 \quad 3 \quad \cdots \quad n \quad .$$

It is easily seen that any path $j \to i$ with $j < i$ in this graph is of the form

$$j \;\to\; j+1 \;\to\; \ldots \;\to\; i-1 \;\to\; i,$$

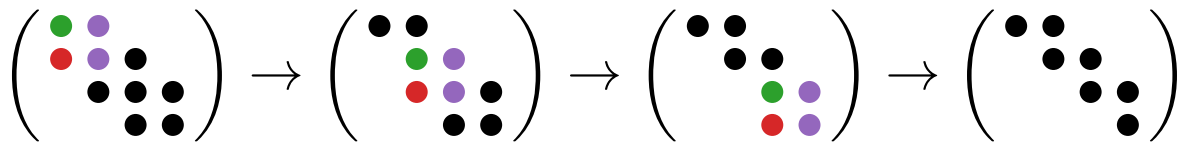
and likewise for $j > i$. Since all intermediate vertices are $> \min\{i, j\}$, none of these paths are fill paths; hence the LU factorisation of $\Delta_n^{(1)}$ incurs no fill-in and requires only $O(n)$ memory.

# Sparse LU Factorisation

*Proof of $O(n)$ runtime and memory of $LU(\Delta_n^{(1)})$ (continued).*

To show the $O(n)$ runtime estimate, we observe:

- There is only a single entry (●) to eliminate in each of the $n$ columns.
- Each elimination requires only $O(1)$ operations because the top row has only two nonzero entries (● and ●).



Evolution of the $U$-factor in the LU factorisation of $\Delta_n^{(1)}$.

Hence the overall number of operations is

$$n \text{ columns} \times 1 \ \frac{\text{eliminations}}{\text{column}} \times O(1) \ \frac{\text{operations}}{\text{elimination}} = O(n) \text{ operations.}$$

# Sparse LU Factorisation

**Discussion: Optimality of LU factorisation of $\Delta_n^{(1)}$**

The $O(n)$ runtime and memory estimates for the LU factorisation of $\Delta_n^{(1)}$ are much better than the $O(n^3)$ runtime and $O(n^2)$ memory estimates that would result if we did not exploit the sparsity of $\Delta_n^{(1)}$.

Furthermore, these estimates are optimal: any algorithm for solving $-\Delta_n^{(1)} u_n = f$ must at the very least read all of $f \in \mathbb{R}^n$ and write into all of $u \in \mathbb{R}^n$, and these operations clearly require at least $O(n)$ runtime and memory.

# Sparse LU Factorisation

**Thm: Runtime and memory of LU($\Delta_n^{(2)}$) without permutations**

LU factorisation of $\Delta_n^{(2)}$ without permuting its rows and columns requires

$$O(n^4) = O(N^2) \text{ runtime}, \qquad \text{and} \qquad O(n^3) = O(N^{3/2}) \text{ memory},$$

where $N = n^2$ denotes the size of $\Delta_n^{(2)} \in \mathbb{R}^{N \times N}$.

*Proof.* As before, the proof consists in drawing the graph of $\Delta_n^{(2)}$ and studying the fill-path-connectedness of pairs of vertices $(i, j)$.

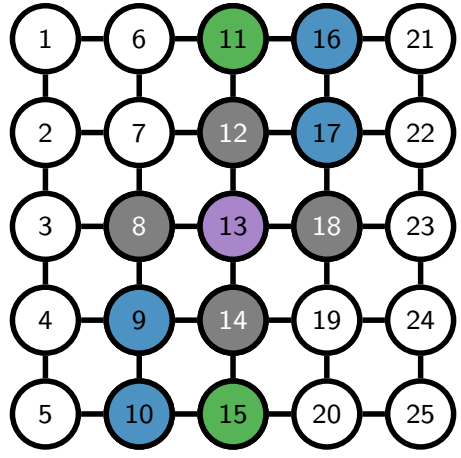
However, considering a general pair of vertices $(i, j)$ in $G(\Delta_n^{(2)})$ for arbitrary $n$ is too abstract to allow for a reasonable discussion.

Instead, I will discuss fill-path-connectedness for just the particular choice $n = 5$ and $j = 13$ and leave it up to you to convince yourself that repeating the presented arguments for different $n$ and $(i, j)$ yields the claimed generalisations.

# Sparse LU Factorisation

*Proof of costs of $LU(\Delta_n^{(2)})$ without permutations (continued).*

The graph of $\Delta_5^{(2)}$ is given by

$$G\left(\Delta_5^{(2)}\right) = $$ 

.

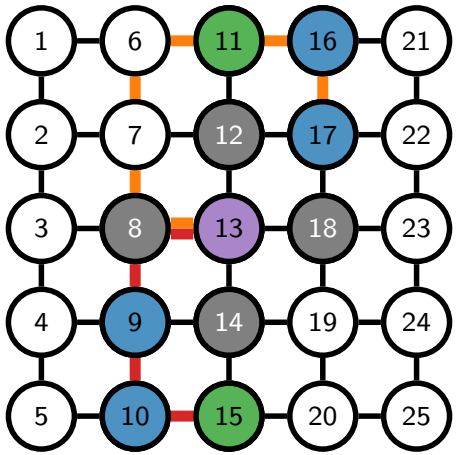
For better readability, I replaced each pair of directed edges ⟲ with a single undirected edge —— in this picture.

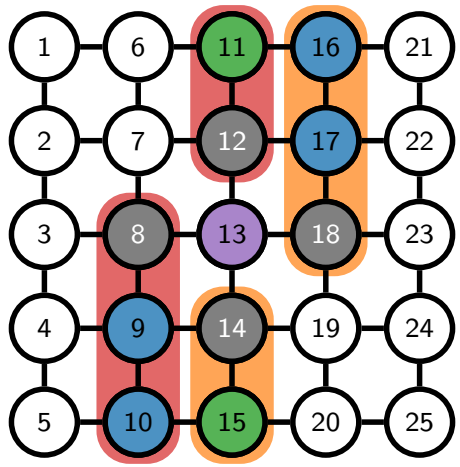
Furthermore, I marked in **black** all neighbours of vertex 13, and I marked in blue and green all vertices which are fill-path-connected to vertex 13 but which are not neighbours of vertex 13.

The next slide will show that this colouring is correct.

# Sparse LU Factorisation

*Proof of costs of $LU(\Delta_n^{(2)})$ without permutations (continued).*




*Proof that coloured vertices are fill-path-connected to 13:* (left graph)

Observe that all subpaths of

$13 \to 8 \to 9 \to 10 \to 15$ and $13 \to 8 \to 7 \to 6 \to 11 \to 16 \to 17$

of the form $13 \to$ (blue or green vertex) are fill paths.

*Proof that white vertices are not fill-path-connected to 13:* (right graph)

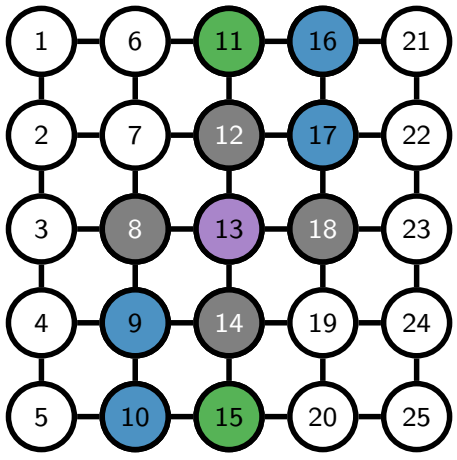
Any path from 13 to a vertex $\leq 7$ has to go through at least one vertex $v \in \{8, \ldots, 12\}$ and is therefore not a fill path since $v > 7$.

Any path from 13 to a vertex $\geq 19$ has to go through at least one vertex $v \in \{14, \ldots, 18\}$ and is therefore not a fill path since $v > 13$.

# Sparse LU Factorisation

*Proof of costs of $LU(\Delta_n^{(2)})$ without permutations (continued).*

$$G\left(\Delta_5^{(2)}\right) =$$




.

We conclude from this picture that the sparsity pattern of row 13 of $L + U$ is given by

$$(L+U)[13,:] = ($$  $$),$$

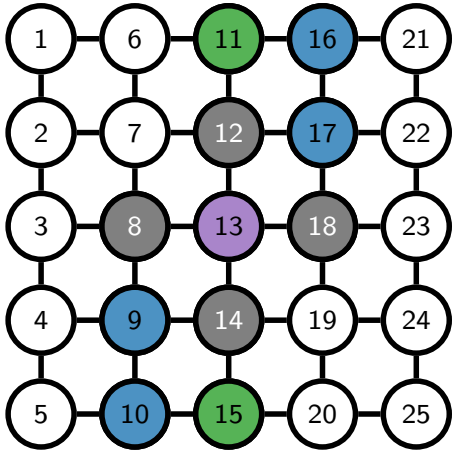


and the sparsity pattern of all of $L + U$ is as shown on the next slide.

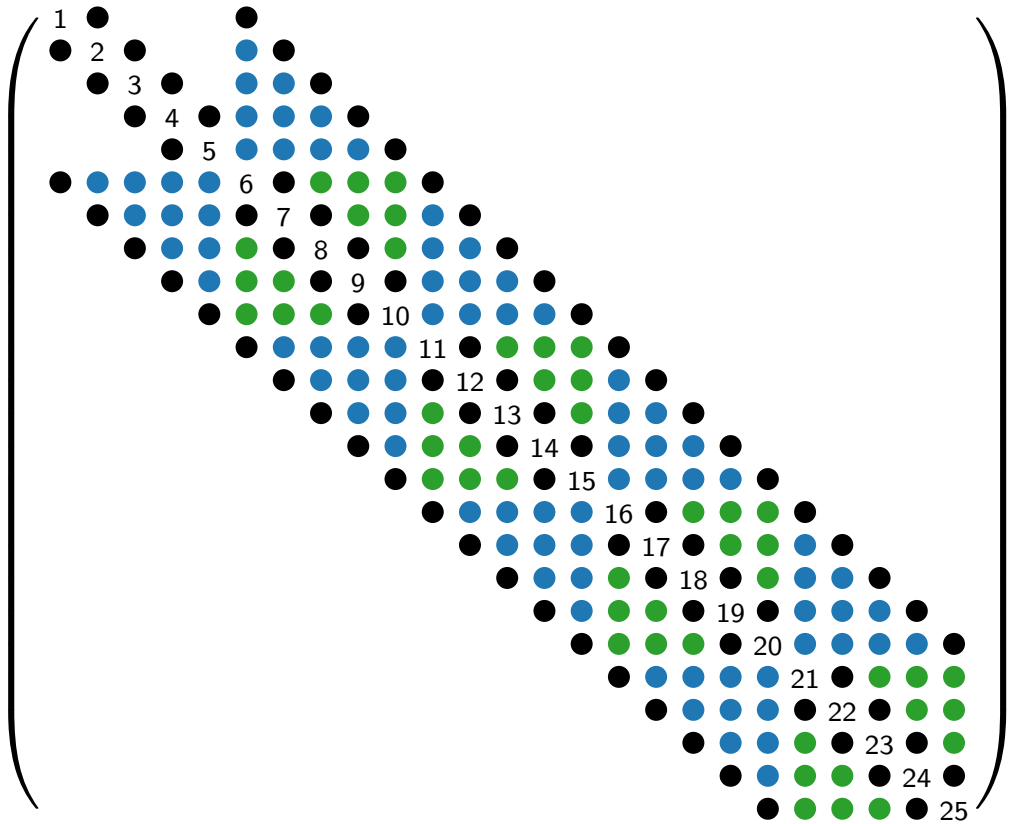
# Sparse LU Factorisation

*Proof of costs of $LU(\Delta_n^{(2)})$ without permutations (continued).*

$G\left(\Delta_5^{(2)}\right) =$




$\longrightarrow \qquad L + U =$




$$(L+U)[i,j] = \begin{cases} \bullet \iff i \text{ and } j \text{ are in the same column of } G(\Delta_5^{(2)}). \\ \bullet \iff i \text{ and } j \text{ are in neighbouring columns of } G(\Delta_5^{(2)}). \end{cases}$$

# Sparse LU Factorisation

*Proof of costs of $LU(\Delta_n^{(2)})$ without permutations (continued).*

Generalising the above to arbitrary $n$, we observe that every column of $L + U$ has at most $2n + 1$ nonzero entries, and all but the first and last $n$ columns have exactly $2n + 1$ nonzero entries.

The overall number of nonzero entries is hence

$$n^2 \text{ columns} \times O(n) \ \tfrac{\text{nonzeros}}{\text{column}} = O(n^3) \text{ nonzeros}$$

as claimed on slide 51.

To verify the $O(n^4)$ runtime estimate, we observe:

- There are at most $n$ entries to eliminate in each of the $n^2$, and in all but the first and last $n$ columns there are exactly $n$ entries to eliminate.
- Each elimination requires $O(n)$ operations because the top row has $O(n)$ nonzero entries.

Hence the overall number of operations is

$$n^2 \text{ columns} \times O(n) \ \tfrac{\text{eliminations}}{\text{column}} \times O(n) \ \tfrac{\text{operations}}{\text{elimination}} = O(n^4) \text{ operations.}$$

# Sparse LU Factorisation

**Discussion**

The $O(n^4) = O(N^2)$ runtime of LU factorisation applied to $\Delta_n^{(2)}$ is of course better than the $O(N^3)$ runtime that would result if we did not exploit the sparsity of $\Delta_n^{(2)}$, but it is still fairly large.

It turns out that we can provably do better using the following vertex order.

---

**Algorithm** Nested dissection order

---

1: Partition the vertices into three sets $V_1, V_2, V_{\text{sep}}$ such that there are no edges between $V_1$ and $V_2$ (subscript sep stands for *separator*).
2: Arrange the vertices in the order $V_1, V_2, V_{\text{sep}}$, where $V_1$ and $V_2$ are ordered recursively according to the nested dissection algorithm and $V_{\text{sep}}$ is ordered arbitrarily.

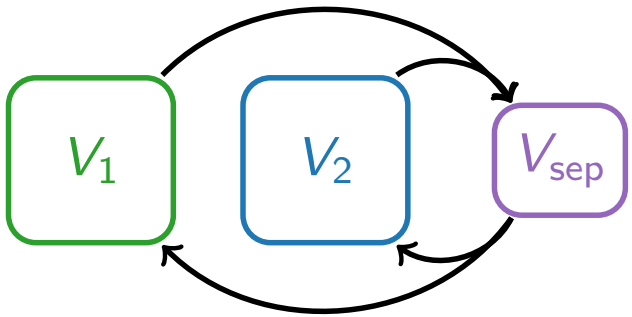
---

# Sparse LU Factorisation

## Discussion

Choosing $P$ according to the nested dissection order leads to a matrix $PAP$ of the form

$$PAP = \left( \quad \right) \quad \longleftrightarrow \quad G(PAP) =$$




This shape guarantees that $L[V_2, V_1] = U[V_1, V_2] = 0$ because any path from $V_1$ to $V_2$ must pass through $V_{\text{sep}}$ and is therefore not a fill-path.

On the other hand, the blocks $L[V_{\text{sep}}, V_1 \cup V_2]$, $U[V_1 \cup V_2, V_{\text{sep}}]$ and in particular $(L + U)[V_{\text{sep}}, V_{\text{sep}}]$ are likely to be fairly dense.

We therefore expect that nested dissection is most effective if

- $|V_{\text{sep}}|$ is as small as possible (this minimises the amount of likely fill-in), and
- $|V_1|$ and $|V_2|$ are of roughly equal size (this maximises the number of entries which are guaranteed not to contain any fill-in).

# Sparse LU Factorisation

**Discussion (continued)**

Nested dissection ordering is rarely used in applications because determining good separators algorithmically is difficult.

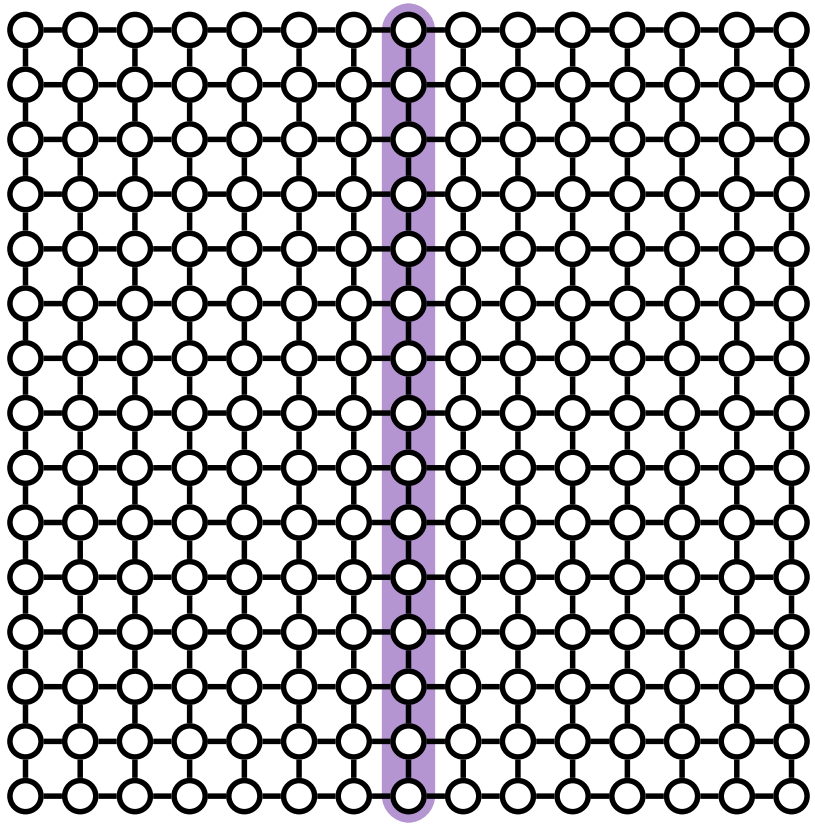
However, $G(\Delta_n^{(2)})$ is simple enough that we can easily determine good separators analytically: at each level of the nested dissection recursion, we simply choose $V_{\text{sep}}$ as the middle slices.

These separator sets are clearly the smallest separators such that the "leftover sets" $V_1$, $V_2$ are of equal size; hence these sets fulfill both of the above criteria.

This process is illustrated on the next slide.

# Sparse LU Factorisation

**Separators for** $G(\Delta_n^{(2)})$

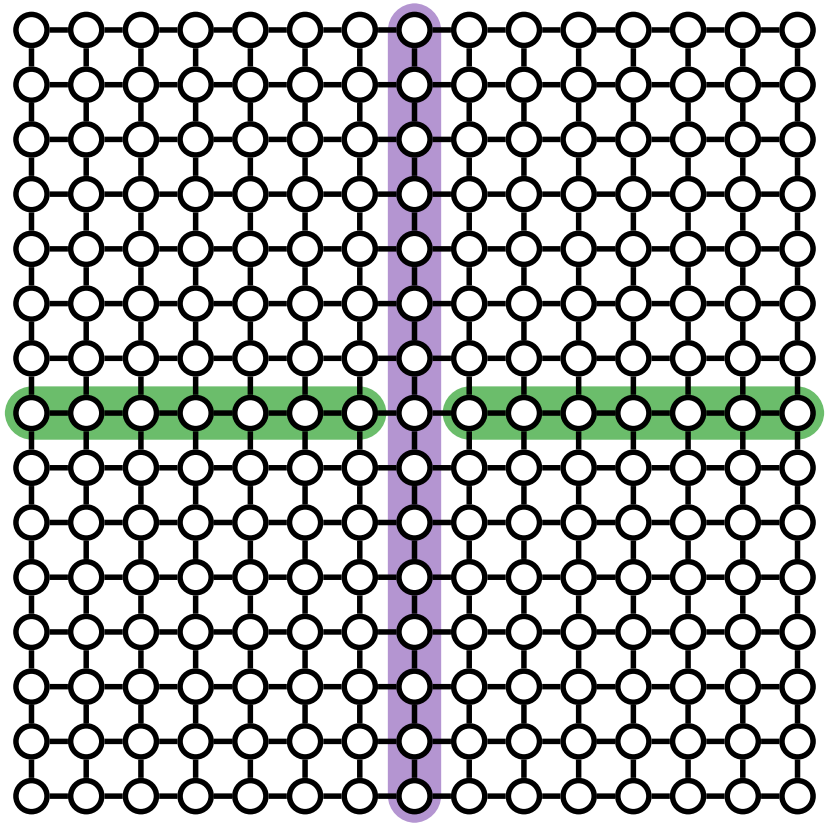


The highlighted vertices indicate the separator sets $V_{\text{sep}}$ at various levels of the nested dissection recursion.

Se

# Sparse LU Factorisation

**Separators for** $G(\Delta_n^{(2)})$

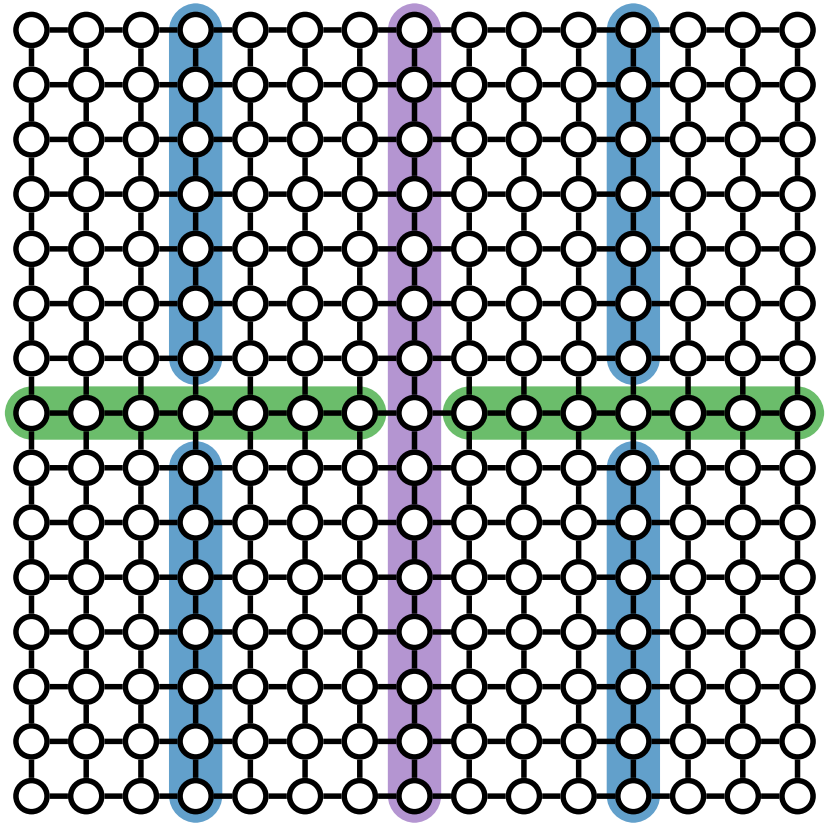


The highlighted vertices indicate the separator sets $V_{\text{sep}}$ at various levels of the nested dissection recursion.

Se

# Sparse LU Factorisation

**Separators for** $G(\Delta_n^{(2)})$

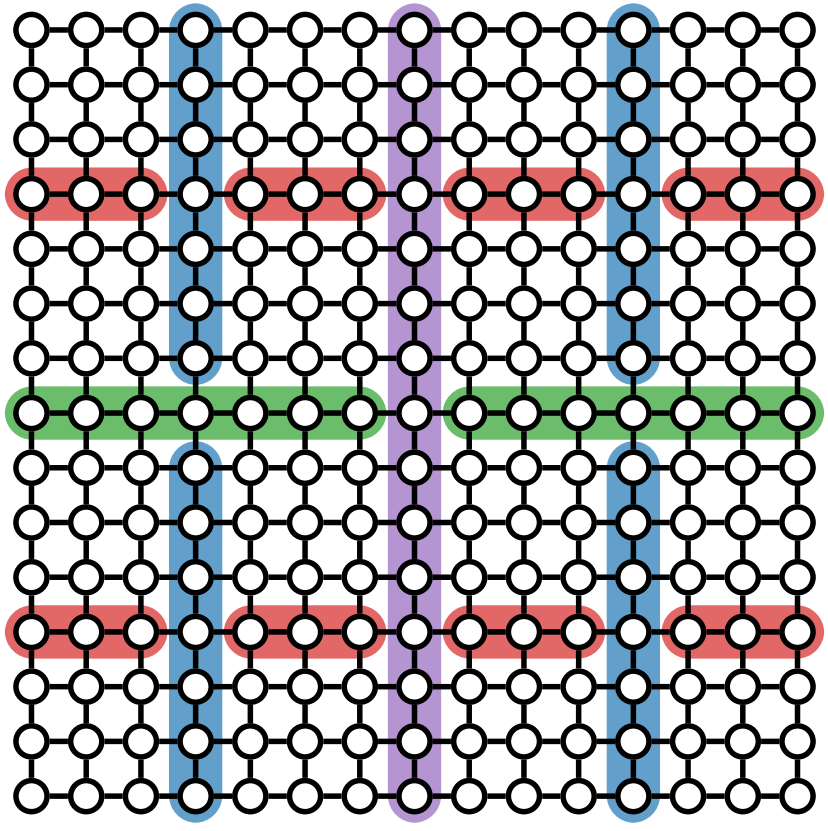


The highlighted vertices indicate the separator sets $V_{\text{sep}}$
at various levels of the nested dissection recursion.

Se

# Sparse LU Factorisation

**Separators for** $G(\Delta_n^{(2)})$



The highlighted vertices indicate the separator sets $V_{\text{sep}}$
at various levels of the nested dissection recursion.

Se

# Sparse LU Factorisation

**Thm: LU factorisation of $\Delta_n^{(d)}$ with nested dissection order**

The runtime and memory requirements of LU factorisation of $\Delta_n^{(d)}$ with nested dissection ordering are as follows ($N = n^d$ denotes the matrix size).

| | Runtime | Memory |
|---|---|---|
| $d = 2$ | $O(N^{3/2})$ | $O(N \log(N))$ |
| $d = 3$ | $O(N^2)$ | $O(N^{4/3})$ |

*Partial proof.* A full proof of this result is beyond the scope of this module . Instead, I will present a simple argument why the runtime for $d = 2$ cannot be lower for the nested dissection order from slide 60.

This argument is based on the observation that for the top-level separator $V_{\text{sep}}^{(1)}$, we have

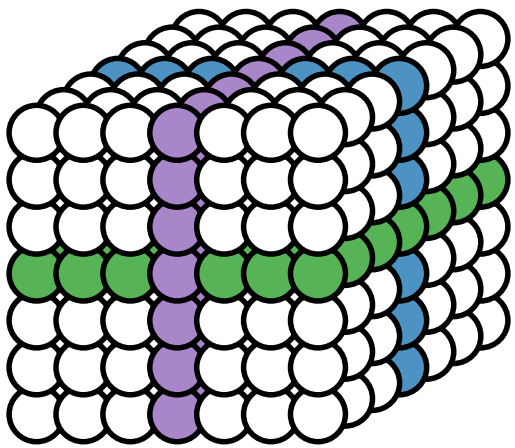
$$|V_{\text{sep}}^{(1)}| = n \qquad \text{and} \qquad (L + U)[V_{\text{sep}}, V_{\text{sep}}] \text{ is dense.}$$

Factorising just this bottom-right corner will therefore require at least $O(n^3) = O(N^{3/2})$ operations; hence the overall runtime cannot be lower.

# Sparse LU Factorisation

*Partial proof (continued).*

The above argument can also easily be generalised to $d = 3$ dimensions: In this case, the separator sets are two-dimensional slices through a three-dimensional cube; hence the top-level separator $V_{\text{sep}}^{(1)}$ is of size $|V_{\text{sep}}^{(1)}| = n^2$, and the number of operations required to factorise this block is $O\big((n^2)^3\big) = O(n^6) = O(N^2)$.



# Sparse LU Factorisation

**Thm: Optimality of nested dissection**

No permutation matrix $P$ leads to runtime and memory requirements of $\mathrm{LU}(P\Delta_n^{(2)}P)$ lower than those reported on slide 61.

*Proof.* Omitted. See Hoffmann, Martin, Rose (1973), Complexity bounds for regular finite difference and finite element grids if you are interested.

# Sparse LU Factorisation

**Conclusion**

LU factorisation is a very convenient algorithm for solving sparse linear systems because all you need to do as a user is to pass the coefficient matrix $A$ and the right-hand side $b$ to the backslash function and SuiteSparse takes it from there.

In addition, we have seen that LU factorisation has optimal complexity when applied to equations which arise from the finite-difference discretisation of one-dimensional partial differential equations.

Unfortunately, this optimality property no longer holds in $d > 1$ dimensions, and in particular the quadratic scaling for $d = 3$ can be fairly limiting in applications.

In the next lecture, we will therefore look at another class of algorithms which sometimes can solve large and sparse linear systems more efficiently.

# Sparse LU Factorisation

**Summary**

- Path and fill path theorems:

$$
\begin{aligned}
A^p[i,j] \neq 0 &\iff \exists \text{ path } j \to i \text{ of length } p \\
A^{-1}[i,j] \neq 0 &\iff \exists \text{ path } j \to i \\
(L+U)[i,j] \neq 0 &\iff \exists \text{ fill path } j \to i
\end{aligned}
$$

- Cost of sparse LU factorisation for partial differential equations:

| | Runtime | Memory |
|---|---|---|
| $d = 1$ | $O(N)$ | $O(N)$ |
| $d = 2$ | $O(N^{3/2})$ | $O(N \log(N))$ |
| $d = 3$ | $O(N^2)$ | $O(N^{4/3})$ |

$N = O(n^d)$ denotes the number of unknowns.

# Sparse LU Factorisation

**Recommended exercise**

- Run `fill_in_example()` and see if you can understand its output using the fill path theorem.
- Have a look at the functions `nested_dissection()`, `runtimes()` and `sparsity_pattern()` provided in the code file for this lecture.