

Dense and Sparse Matrices

Computational Linear Algebra for Large Scale Problems

Stefano Berrone, Francesco Della Santa and Ulderico Fugacci

stefano.berrone@polito.it

October 11, 2022

Dense and Sparse Matrices

In a large variety of applicative domains, there is the need for working with matrices of “**large**” size.

Due to that, **storing** and **manipulating** such matrices can represent a serious **bottleneck** for computations.

Luckily, in several applications, the matrices we have to deal with have a **high number of zeros entries** (e.g., matrices describing complex systems in which entities have few pairwise interactions).

Can we exploit this feature in order to reduce storage costs and to speed up computations?

Dense and Sparse Matrices

“Definition”

Let $A \in \mathbb{R}^{m \times n}$ be matrix and let Nz be the number of its non-zero entries.

Depending on the working framework, A is called **sparse** if one of these conditions is true:

- most of its entries are zero (i.e., $Nz \ll \frac{mn}{2}$);
- A has $O(\min\{m, n\})$ non-zero entries^a;
- A has $O(\min\{m, n\}^{1+\gamma})$ non-zero entries for some $\gamma < 1$.

^a $f(x) \in O(g(x))$ if $\exists x_0, c > 0$ such that $|f(x)| \leq c|g(x)|$ for any $x > x_0$.

More reasonably:

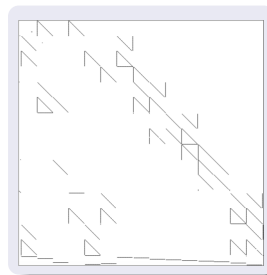
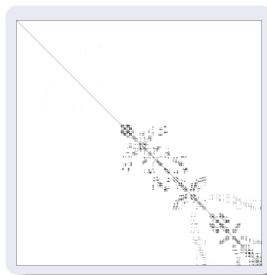
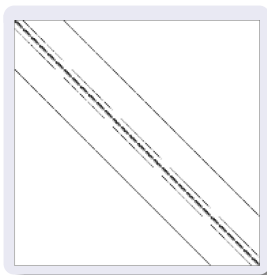
A matrix is **sparse** if we can exploit the fact that a part of its entries is equal to zero.

Dense and Sparse Matrices

Conversely, we can define a non-sparse matrix $A \in \mathbb{R}^{m \times n}$ as **dense**.

Let Nz be the number of the non-zero entries of a matrix A , we call:

- the **sparsity** of A the value $\frac{mn - Nz}{mn}$;
- the **density** of A the value $\frac{Nz}{mn}$.



Roughly, we can classify sparse matrices as **structured** or **unstructured** if their non-zero entries form or not regular patterns.

Dense and Sparse Matrices

Example of a (small) sparse matrix:

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

Dense and Sparse Matrices

Outline:

- 1 Storage Schemes;
- 2 Operations;
- 3 Reorderings;

Storage Schemes

Given a matrix $A \in \mathbb{R}^{m \times n}$, **storing** A means encoding and being able to retrieve each entry $a_{i,j}$ of A .

Usually, storing an $m \times n$ matrix requires an amount of memory proportional to mn and it is computationally very expensive if $mn \gg 1$.

Then, **for a sparse matrix** A , it is convenient to store only the non-zero entries because allows to **drastically reduce the memory usage (PRO)**.

CONs: storing only non-zero entries requires the use of **more complex structures**, making the **access to the matrix entries a bit less immediate**.

Sparse Matrices in Programming Languages:

- **Python:** <https://docs.scipy.org/doc/scipy/reference/sparse.html>;
- **Matlab:** <https://it.mathworks.com/help/matlab/sparse-matrices.html>.

Storage Schemes

Storage schemes for sparse matrices can be classified in:

- ④ Schemes allowing for **efficient modifications** (mainly adopted in matrix **construction**)
 - **Dictionary of Keys (DOK)**;
 - **List of Lists (LIL)**;
 - **Coordinate Format (COO)**.
- ② Schemes allowing for **efficient entry access** and **fast matrix operations**
 - **Compressed Sparse Row/Columns Format (CSR/CSC)**;
 - **Modified Sparse Row/Column Format (MSR/MSD)**;
- ③ Schemes designed for **specific classes of sparse matrices**
 - **Diagonal Format (DIAG)**,
 - **Ellpack-Itpack Format**.

Storage Schemes - Dictionary of Keys (Type 1)

Dictionary of Keys (DOK): The Nz non-zero entries of a sparse matrix $A \in \mathbb{R}^{m \times n}$ are stored as a **dictionary** that maps the key (i, j) into the value $a_{i,j}$, for each non-zero entry $a_{i,j}$ of A :

$$(i, j) \mapsto a_{i,j}.$$

Example:

$$A = \begin{bmatrix} \mathbf{3} & 0 & \mathbf{-2} & 0 \\ 0 & 0 & 0 & 0 \\ \mathbf{-1} & 0 & \mathbf{2} & 0 \\ 0 & \mathbf{3} & 0 & \mathbf{1} \end{bmatrix} \rightsquigarrow \begin{array}{ll} (1, 1) & \mapsto 3 \\ (1, 3) & \mapsto -2 \\ (3, 1) & \mapsto -1 \\ (3, 3) & \mapsto 2 \\ (4, 2) & \mapsto 3 \\ (4, 4) & \mapsto 1 \end{array}$$

Required memory: $O(Nz)$.

Storage Schemes - Dictionary of Keys

Exercise: Express the following matrix in the DOK storage scheme.

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

Exercise: Retrieve the matrix encoded by the following DOK storage scheme.

$$\begin{array}{lll} (1, 3) & \mapsto & -1 \\ (2, 1) & \mapsto & 7 \\ (2, 2) & \mapsto & -4 \\ (3, 2) & \mapsto & 2 \end{array}$$

Storage Schemes - List of Lists (Type 1)

List of Lists (LIL): The sparse matrix $A \in \mathbb{R}^{m \times n}$ is encoded by a list

$$[L_1, \dots, L_m],$$

where L_i , for each row $i = 1, \dots, m$ of A , is the list

$$L_i = [(j_1, a_{i,j_1}), \dots, (j_k, a_{i,j_k})],$$

consisting in the k pairs representing the column index and the value of the non-zero entries in row i .

Analogously, LIL can be defined in terms of columns instead of rows.

Example

$$A = \begin{bmatrix} 3 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{aligned} L_1 &= [(1, 3), (3, -2)] \\ L_2 &= [] \\ L_3 &= [(1, -1), (3, 2)] \\ L_4 &= [(2, 3), (4, 1)] \end{aligned}$$

Required memory: $O(Nz)$.

Storage Schemes - List of Lists

Exercise: Express the following matrix in the LIL storage scheme.

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

Exercise: Retrieve the matrix encoded by the following LIL storage scheme.

$$L_1 = [(2, -2), (3, 1)]$$

$$L_2 = [(4, 1)]$$

$$L_3 = [(1, 2), (3, 2)]$$

Storage Schemes - Coordinate Format (Type 1)

Coordinate Format (COO): The sparse matrix $A \in \mathbb{R}^{m \times n}$ is encoded by three arrays of length Nz :

- AA contains the **values** $a_{i,j}$ of the non-zero entries of A ;
- JR contains the **row indexes** of the non-zero entries of A (i.e. $AA[k] = a_{i,j} \Rightarrow JR[k] = i$);
- JC contains the **column indexes** of the non-zero entries of A (i.e. $AA[k] = a_{i,j} \Rightarrow JC[k] = j$).

Example:

$$A = \begin{bmatrix} 3 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{array}{l} AA = [3 \quad -2 \quad -1 \quad 2 \quad 3 \quad 1] \\ JR = [1 \quad 1 \quad 3 \quad 3 \quad 4 \quad 4] \\ JC = [1 \quad 3 \quad 1 \quad 3 \quad 2 \quad 4] \end{array}$$

Required memory: $O(Nz)$.

Storage Schemes - Coordinate Format

Exercise: Express the following matrix in the COO storage scheme.

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

Exercise: Retrieve the matrix encoded by the following COO storage scheme.

$$AA = \begin{bmatrix} 7 & 2 & 4 & 5 & 3 \end{bmatrix}$$

$$JR = \begin{bmatrix} 1 & 2 & 2 & 3 & 3 \end{bmatrix}$$

$$JC = \begin{bmatrix} 1 & 2 & 3 & 1 & 3 \end{bmatrix}$$

Storage Schemes - Compressed Sparse Row Format (Type 2)

Compressed Sparse Row Format (CSR): The sparse matrix $A \in \mathbb{R}^{m \times n}$ is encoded by three arrays:

- AA contains the real values $a_{i,j}$ of the non-zero entries of A **sorted** w.r.t. the lexicographic order of their row-column indexes;
- JA contains the column indexes of the non-zero entries of A ;
- IA is an array of length $m + 1$, recursively defined as:
 - $IA[1] = 1$,
 - $IA[i + 1] = IA[i] + \text{the number of the non-zero entries in row } i \text{ of } A$, for $i = 1, \dots, m$,
i.e. $IA[2] = IA[1] + 2$, $IA[3] = IA[2] + 0$, $IA[4] = IA[3] + 2$, $IA[5] = IA[4] + 2$.

Example

$$A = \begin{bmatrix} 3 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{array}{l} AA = \begin{bmatrix} 3 & -2 & -1 & 2 & 3 & 1 \end{bmatrix} \\ JA = \begin{bmatrix} 1 & 3 & 1 & 3 & 2 & 4 \end{bmatrix} \\ IA = \begin{bmatrix} 1 & 3 & 3 & 5 & 7 \end{bmatrix} \end{array}$$

Storage Schemes - Compressed Sparse Row Format

Some Easy Observations:

- CSR format is also known as **Compressed Row Storage (CRS)** and **Yale Format**;
- CSR format can be analogously defined in terms of columns (**CSC**);
- If $IA[i] \neq IA[i + 1]$, then $IA[i]$ represents the index of array AA starting from which the non-zero entries of row i of A are stored;
- Since $\sum_{i=1}^m (IA[i + 1] - IA[i]) = Nz$, then

$$IA[m + 1] = Nz + 1;$$

- The three arrays are of overall length $2Nz + m + 1$. So, CSR format saves on memory if

$$Nz < \frac{m(n - 1) - 1}{2}.$$

Required memory: $O(Nz + m)$.

Storage Schemes - Compressed Sparse Row Format

Exercise: Express the following matrix in the CSR storage scheme.

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

Exercise: Retrieve the matrix encoded by the following CSR storage scheme.

$$AA = \begin{bmatrix} 2 & 1 & 4 & 3 \end{bmatrix}$$

$$JA = \begin{bmatrix} 2 & 3 & 1 & 2 \end{bmatrix}$$

$$IA = \begin{bmatrix} 1 & 1 & 3 & 4 & 5 \end{bmatrix}$$

Storage Schemes - Modified Sparse Row Format (Type 2)

Remark

In applications, **diagonal elements** of matrices usually are **non-zero** and **accessed more often** than the rest of the entries.

Modified Sparse Row Format (MSR): Let A be a **square** sparse matrix in $\mathbb{R}^{n \times n}$ having **all the diagonal entries non-zero**. A is encoded by two arrays AA and JA of length $Nz + 1$ such that:

- AA contains in the first n positions, the values $a_{i,i}$ of the diagonal sorted w.r.t. the lexicographic order; then, in position $n + 1$ nothing and starting from position $n + 2$ it stores the values $a_{i,j}$ of the non-zero and non-diagonal entries (sorted w.r.t. the lexicographic order, i.e. all the nondiagonal element of each row);
- JA contains in the first n positions an index $JA[i]$ s.t. the elements $AA[JA[i]], \dots, AA[Nz + 1]$ are the non-zero non-diagonal entries of A from row i to row m ; $JA[i + 1] - JA[i]$ is the number of nondiagonal elements of row i . Then, starting from position $n + 2$, $JA[i]$ is the column index of the values $a_{i,j} = AA[i]$.
In the $JA[n + 1]$ it is stored the value $Nz + 2$.

Storage Schemes - Modified Sparse Row Format

Example

$$A = \begin{bmatrix} 3 & 0 & -2 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 1 \end{bmatrix} \rightsquigarrow \begin{array}{l} AA = [3 \quad 1 \quad 2 \quad 1 \quad * \quad -2 \quad -1 \quad 3] \\ JA = [6 \quad 7 \quad 7 \quad 8 \quad 9 \quad 3 \quad 1 \quad 2] \end{array}$$

Modified Sparse Row format can be analogously defined in terms of columns instead of rows.

Required memory: $O(Nz)$.

Storage Schemes - Modified Sparse Row Format

Exercise: Express the following matrix in the MSR storage scheme.

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

Exercise: Retrieve the matrix encoded by the following MSR storage scheme.

$$\begin{aligned} AA &= \begin{bmatrix} 3 & 2 & 4 & 1 & * & 1 & 1 & 3 \end{bmatrix} \\ JA &= \begin{bmatrix} 6 & 6 & 7 & 8 & 9 & 3 & 1 & 2 \end{bmatrix} \end{aligned}$$

Storage Schemes - Diagonal Format (Type 3)

Diagonally structured matrices are sparse matrices in which the non-zero entries are located along a small number Nd of diagonals.

Diagonal Format (DIAG): A diagonally structured matrix $A \in \mathbb{R}^{n \times n}$ (square) is encoded by:

- an 1-dimensional array $IOFF$ of length Nd storing the “diagonal indexes” the non-zero diagonals;
- a 2-dimensional array $DIAG$ of size $n \times Nd$ such that,

$$DIAG[i, j] = a_{i, i+IOFF[j]};$$

i.e., the j -th column of $DIAG$ is an array of n elements where the last/first k elements are empty, if $IOFF[j] = \pm k$, and the other are the elements of the $(\pm k)$ -th diagonal of A (entries sorted in lexicographic order).

Example

$$A = \begin{bmatrix} \textcolor{red}{3} & 0 & \textcolor{red}{-2} & 0 \\ \textcolor{blue}{2} & \textcolor{red}{1} & 0 & \textcolor{red}{1} \\ 0 & \textcolor{blue}{4} & \textcolor{red}{2} & 0 \\ 0 & 0 & \textcolor{blue}{1} & \textcolor{red}{1} \end{bmatrix} \rightsquigarrow IOFF = [-1 \quad 0 \quad 2], \quad DIAG = \begin{bmatrix} * & \textcolor{red}{3} & \textcolor{red}{-2} \\ \textcolor{blue}{2} & \textcolor{red}{1} & \textcolor{red}{1} \\ \textcolor{blue}{4} & \textcolor{red}{2} & * \\ \textcolor{blue}{1} & \textcolor{red}{1} & * \end{bmatrix}$$

Storage Schemes - Diagonal Format

Some Easy Observations:

- The order in which the diagonals are stored in the columns of *DIAG* can be arbitrary. If several operations are performed with the main diagonal, storing it in the first column may be slightly advantageous.
- All the diagonals except the main diagonal have fewer than n elements, so there are positions in *DIAG* that will not be used.

Required memory: $O(nNd)$.

Storage Schemes - Diagonal Format

Exercise: Express the following matrix in the DIAG storage scheme.

$$A = \begin{bmatrix} \mathbf{1} & 0 & \mathbf{2} & 0 \\ 0 & \mathbf{3} & 0 & \mathbf{4} \\ 0 & 0 & \mathbf{-1} & 0 \\ \mathbf{7} & 0 & 0 & \mathbf{5} \end{bmatrix}$$

Exercise: Retrieve the matrix encoded by the following DIAG storage scheme.

$$IOFF = \begin{bmatrix} -3 & 0 & 1 \end{bmatrix} \quad DIAG = \begin{bmatrix} * & 5 & 8 \\ * & 3 & 5 \\ * & 4 & 2 \\ 8 & 1 & 4 \\ 1 & 2 & * \end{bmatrix}$$

Storage Schemes - Ellpack-Itpack Format (Type 3)

Ellpack-Itpack Format: A diagonally structured matrix $A \in \mathbb{R}^{n \times n}$ (square) is encoded by two 2-dimensional arrays of size $n \times Nd$:

- *COEF* which contains in each of its rows the values of the non-zero entries of the corresponding row of A ;
- *JCOEF*, whose element $JCOEF[i, j]$ represents the column index of the entry represented by $COEF[i, j]$.

Example

$$A = \begin{bmatrix} 3 & 0 & -2 & 0 \\ 2 & 1 & 0 & 1 \\ 0 & 4 & 2 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \rightsquigarrow COEF = \begin{bmatrix} 3 & -2 & * \\ 2 & 1 & 1 \\ 4 & 2 & * \\ 1 & 1 & * \end{bmatrix}, \quad JCOEF = \begin{bmatrix} 1 & 3 & * \\ 1 & 2 & 4 \\ 2 & 3 & * \\ 3 & 4 & * \end{bmatrix}$$

Storage Schemes - Ellpack-Itpack Format

Some Easy Observations:

- Ellpack-Itpack format is based on the fact that in a diagonally structured matrix there are at most Nd non-zero entries per row;
- Ellpack-Itpack format (more specifically, *JCOEF*) can be analogously defined in terms of row indexes instead of column indexes;
- There are positions in *COEF* and *JCOEF* that will remain empty.

Required memory: $O(nNd)$.

Storage Schemes - Ellpack-Itpack Format

Exercise: Express the following matrix in the Ellpack-Itpack storage scheme.

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

Exercise: Retrieve the matrix encoded by the following Ellpack-Itpack storage scheme.

$$COEF = \begin{bmatrix} 5 & 9 & * & * \\ 4 & * & * & * \\ 1 & * & * & * \\ 2 & 3 & * & * \\ 5 & 3 & 2 & * \end{bmatrix} \quad JCOEF = \begin{bmatrix} 1 & 5 & * & * \\ 2 & * & * & * \\ 3 & * & * & * \\ 1 & 4 & * & * \\ 1 & 2 & 5 & * \end{bmatrix}$$

Operations

The **result** of a matrix operation **does not depend on the storage scheme** adopted to encode the involved matrices

BUT

its **time and space complexity** does!

Operations

Storage schemes for sparse matrices **effectively improve complexity** of matrix operations.

Let us see some examples:

- **Addition** of matrices represented in **COO** format;
- **Multiplication** of a matrix represented in **CSR** format by a vector.

Operations and Sparse Matrices in Programming Languages: In Matlab and Python, the operation between sparse matrices can be done using the “normal” operators for matrices (e.g. `*/@` for the matrix product in Matlab/Python).

N.B.: In general, in programming languages, the output of a matrix operation is **dense** if one of the two matrices is dense; otherwise the output is a sparse matrix.

Operations - Addition in COO

Addition of matrices represented in COO format:

Given two matrices $A, A' \in \mathbb{R}^{m \times n}$ encoded, respectively, in the following COO storage schemes:

$$A \rightsquigarrow AA, JR, JC, \quad A' \rightsquigarrow AA', JR', JC',$$

then, their addition $A'' = A + A'$, expressed in the COO format AA'', JR'', JC'' , can be retrieved by running the following algorithm.

Operations - Addition in COO

- ④ **Initialize** indexes: $k, k', k'' \leftarrow 1$;
- ② **while** $(k + k') \leq (Nz + Nz')$ **do**:
 - ① **if** $(JR[k] < JR'[k'])$ or $(JR[k] == JR'[k'] \text{ and } JC[k] < JC'[k'])$ **do**:
 - # (i.e. if $a_{JR[k], JC[k]} \neq 0 = a'_{JR[k], JC[k]}$)
 - ① $AA''[k''] \leftarrow AA[k], JR''[k''] \leftarrow JR[k], JC''[k''] \leftarrow JC[k]$;
 - ② $k'' \leftarrow (k'' + 1), k \leftarrow (k + 1)$;
 - ② **else if** $(JR[k] > JR'[k'])$ or $(JR[k] == JR'[k'] \text{ and } JC[k] > JC'[k'])$ **do**:
 - # (i.e. if $a_{JR'[k'], JC'[k']} = 0 \neq a'_{JR'[k'], JC'[k']}$)
 - ① $AA''[k''] \leftarrow AA'[k'], JR''[k''] \leftarrow JR'[k'], JC''[k''] \leftarrow JC'[k']$;
 - ② $k'' \leftarrow (k'' + 1), k' \leftarrow (k' + 1)$;
 - ③ **else do**:
 - # (i.e. if $(JR[k], JC[k]) = (JR'[k'], JC'[k'])$)
 - # (i.e. if $a_{JR[k], JC[k]} \neq 0 \neq a'_{JR[k], JC[k]}$)
 - ① $AA''[k''] \leftarrow (AA[k] + AA'[k']), JR''[k''] \leftarrow JR'[k'], JC''[k''] \leftarrow JC'[k']$;
 - ② $k'' \leftarrow (k'' + 1), k' \leftarrow (k' + 1), k \leftarrow (k + 1)$;
- ③ **return** AA'', JR'', JC''

Time Complexity: $O(Nz + Nz')$.

Operations - Addition in COO

Exercise: Compute the addition of the following matrices by using their COO format.

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

$$\begin{aligned} AA &= \begin{bmatrix} 3 & -2 & -1 & 2 & 3 & 1 \\ 1 & 1 & 3 & 3 & 4 & 4 \\ 1 & 3 & 1 & 3 & 2 & 4 \end{bmatrix} & AA' &= \begin{bmatrix} 1 & 3 & 4 & 2 & -1 & 5 \\ 1 & 2 & 2 & 3 & 3 & 4 \\ 1 & 2 & 4 & 2 & 3 & 4 \end{bmatrix} \\ JR &= \begin{bmatrix} 1 & 1 & 3 & 3 & 4 & 4 \end{bmatrix} & JR' &= \begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 4 \end{bmatrix} \\ JC &= \begin{bmatrix} 1 & 3 & 1 & 3 & 2 & 4 \end{bmatrix} & JC' &= \begin{bmatrix} 1 & 2 & 4 & 2 & 3 & 4 \end{bmatrix} \end{aligned}$$

Operations - Matrix-by-vector Multiplication in CSR

Multiplication of a matrix represented in CSR format by a vector:

Given a matrix $A \in \mathbb{R}^{m \times n}$ encoded in CSR format

$$A \rightsquigarrow AA, JA, IA,$$

and a vector $\mathbf{v} \in \mathbb{R}^{n \times 1} \simeq \mathbb{R}^n$ encoded as a 1D-array of its coordinates

$$\mathbf{v} \rightsquigarrow [v_1, \dots, v_n]$$

the product $\mathbf{w} = A\mathbf{v} \in \mathbb{R}^m$ can be retrieved by running the following algorithm.

Operations - Matrix-by-vector Multiplication in CSR

- ① Initialize $w \leftarrow 0 \in \mathbb{R}^m$;
- ② for $i = 1, \dots, m$ do:
 - # (remember: $IA[i+1] = IA[i] + \text{non-zero entries of row } A_{i,\cdot}$)
 - # (remember: if $IA[i] \neq IA[i+1]$, then $AA[IA[i]]$ is the first non-zero el. of row $A_{i,\cdot}$)
 - ① $w_i \leftarrow \sum_{h=IA[i]}^{(IA[i+1]-1)} AA[h] v_{JA[h]}$;
- ③ return w

Time Complexity: $O(mNr)$, where Nr is the maximum number of non-zero entries in a row of A .

N.B.: Each iteration of the main cycle can be computed independently from the other ones.

Operations - Matrix-by-vector Multiplication in CSR

Exercise: Compute the matrix-by-vector multiplication $\mathbf{w} = A\mathbf{v}$ of the following matrices by using CSR format for representing A .

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix} \rightsquigarrow \begin{matrix} AA = [1 & 3 & 4 & 2 & -1 & 5] \\ JA = [1 & 2 & 4 & 2 & 3 & 4] \\ IA = [1 & 2 & 4 & 6 & 7] \end{matrix};$$

$$\mathbf{v} = \begin{bmatrix} -2 \\ 1 \\ -3 \\ 2 \end{bmatrix}$$

Reorderings

Pattern Matters: The **pattern** formed by the non-zero entries of a matrix A can have a **significant impact** on the execution of algorithms.

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

(*"not-so-nice" pattern*)

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

(*"nice" pattern*)

Reorderings - Permutations

Given a matrix $A = (a_{i,j}) \in \mathbb{R}^{n \times n}$ (square), the pattern of the non-zero entries of A can be modified by **reordering** its rows and columns according with a **permutation**¹ $\sigma \in S_n$

$$\sigma = \begin{pmatrix} 1 & 2 & \dots & n \\ \sigma(1) & \sigma(2) & \dots & \sigma(n) \end{pmatrix}.$$

Let $A' = \sigma(A)$ the matrix obtained reordering A with respect to σ , then:

$$A' = (a'_{i,j}) := (a_{\sigma(i), \sigma(j)}).$$

¹see “Linear Algebra: Basic Tools” lectures.

Reorderings - Permutations

Example: Given the following matrix $A \in \mathbb{R}^{9 \times 9}$ and the permutation $\sigma \in S_9$

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 8 & 6 & 4 & 1 & 9 & 7 & 5 & 2 \end{pmatrix},$$

we obtain the following **reordering** for matrix A .

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{matrix} & \begin{pmatrix} \blacksquare & & & \blacksquare & & & & \blacksquare & \\ & \blacksquare & & \blacksquare & \blacksquare & & & \blacksquare & \\ & & \blacksquare & \blacksquare & & & \blacksquare & & \blacksquare \\ \blacksquare & \blacksquare & & \blacksquare & & & & & \\ & & & & \blacksquare & & & & \\ & & & & & \blacksquare & & & \blacksquare \\ & & & & & & \blacksquare & & \blacksquare \\ \blacksquare & & \blacksquare & & & & & \blacksquare & \\ & \blacksquare & & \blacksquare & & \blacksquare & & & \blacksquare \end{pmatrix} \end{matrix} \quad \sigma(A) = \begin{matrix} & \begin{matrix} 3 & 8 & 6 & 4 & 1 & 9 & 7 & 5 & 2 \end{matrix} \\ \begin{matrix} 3 \\ 8 \\ 6 \\ 4 \\ 1 \\ 9 \\ 7 \\ 5 \\ 2 \end{matrix} & \begin{pmatrix} \blacksquare & & & & & \blacksquare & \blacksquare & & \blacksquare \\ & \blacksquare & & & \blacksquare & & & & \blacksquare \\ & & \blacksquare & & & \blacksquare & & & \blacksquare \\ & & & \blacksquare & \blacksquare & \blacksquare & & & \blacksquare \\ \blacksquare & \blacksquare & & & \blacksquare & & & & \\ & & \blacksquare & & & & \blacksquare & \blacksquare & \\ \blacksquare & & & & & & & \blacksquare & \blacksquare \\ & \blacksquare & & & & & & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & & \blacksquare & & & & & \blacksquare \end{pmatrix} \end{matrix}.$$

Reorderings - Permutations

Formally, given a matrix $A \in \mathbb{R}^{n \times n}$ and a permutation $\sigma \in S_n$, the **reordered matrix** $\sigma(A)$ is defined as

$$\sigma(A) = E^{(\sigma)} A (E^{(\sigma)})^{-1},$$

where $E^{(\sigma)} \in GL_n(\mathbb{R})$ is the matrix obtained from the identity matrix I_n reordering its rows according with σ (i.e., the entry of position (i, j) is 1 if $\sigma(i) = j$ and 0 otherwise).

N.B.: Let $\sigma_k \cdots \sigma_1 = (i_k, j_k) \cdots (i_1, j_1)$ be a 2-cycle decomposition of σ , then $E^{(\sigma)} = E^{(i_k, j_k)} \cdots E^{(i_1, j_1)}$ (see the elementary matrices in “Linear Algebra: Basic Tools” lectures).

Remarks:

- By definition, A and $\sigma(A)$ are **similar** matrices.
- Given a matrix A , we want to find a permutation σ such that $\sigma(A)$ has a (more) **“pleasant” pattern**.

Reorderings

Example (continued): By reordering the rows and the columns of A according with a different permutation $\sigma' \in S_9$,

$$\sigma' = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 1 & 4 & 8 & 2 & 3 & 7 & 9 & 6 \end{pmatrix}$$

we obtain the following reordered matrix $\sigma'(A)$:

$$\sigma'(A) = \begin{matrix} & \begin{matrix} 5 & 1 & 4 & 8 & 2 & 3 & 7 & 9 & 6 \end{matrix} \\ \begin{matrix} 5 \\ 1 \\ 4 \\ 8 \\ 2 \\ 3 \\ 7 \\ 9 \\ 6 \end{matrix} & \begin{pmatrix} \blacksquare & & & & & & & & \\ & \blacksquare & \blacksquare & \blacksquare & & & & & \\ & \blacksquare & \blacksquare & & \blacksquare & & & & \\ & \blacksquare & & \blacksquare & \blacksquare & & & & \\ & & \blacksquare & \blacksquare & \blacksquare & \blacksquare & & & \\ & & & & \blacksquare & \blacksquare & \blacksquare & \blacksquare & \\ & & & & & \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ & & & & & & \blacksquare & \blacksquare & \blacksquare \\ & & & & & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}.$$

Reorderings

Common Pipeline: the general strategy adopted to “improve” the pattern of a matrix A consists in:

- 1 Consider the **adjacency graph** $G := G(A)$ of A ;
- 2 Depending on the kind of pattern we aim to reach, **run a specific algorithm on the graph G returning as output a permutation σ** ;
- 3 **Compute $\sigma(A)$.**

N.B.: For the sake of clarity, we will consider in these notes just the case in which the considered matrix A is such that:

- A has a **symmetric pattern**;
- the adjacency graph G of A is **connected**.

Different cases can be easily reconducted to this situation.

Reorderings - Adjacency Graph

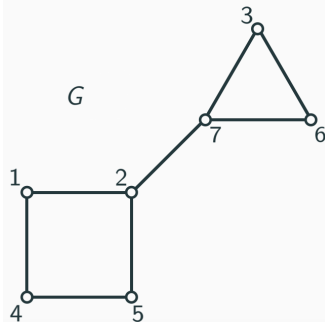
Definition: Let $A = (a_{i,j})$ be a matrix in $\mathbb{R}^{n \times n}$. The **adjacency graph**² of A is the graph $G = (V, E)$ such that:

- the set of **vertices** V is the set $\{v_1, \dots, v_n\}$ where the labels of vertexes are such that $\ell(v_i) = i$;
- for each $v_i, v_j \in V$, (v_i, v_j) is an **edge** in E if and only if $a_{i,j} \neq 0$.

Example:

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & & \blacksquare & & & \\ \blacksquare & \blacksquare & & & \blacksquare & & \blacksquare \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ \blacksquare & & & \blacksquare & \blacksquare & & \\ & \blacksquare & & \blacksquare & \blacksquare & & \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ & \blacksquare & \blacksquare & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

\rightsquigarrow



²Conversely, A is the **adjacency matrix** of the graph G .

Reorderings

Discussed methods

Two common classes of reorderings obtained via the adjacency graph are:

- **Level-set** orderings (returning as output a **banded** matrix);
- **Independent set** orderings (returning as output a matrix having a **diagonal upper-left block**).

Reorderings - Level-Set Orderings

Definition

Given a matrix $A = (a_{i,j}) \in \mathbb{R}^{n \times n}$, the **bandwidth** $Bw(A)$ of A is the smallest non-negative k such that, if $|i - j| > k$, then $a_{i,j} = 0$.

In other words: the maximum absolute value among the indexes of non-null diagonals is k .

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & & & & & \\ \blacksquare & & \blacksquare & & & & \\ & \blacksquare & \blacksquare & & & & \\ & & & \blacksquare & & & \\ & & & & \blacksquare & \blacksquare & \blacksquare \\ & & & & \blacksquare & & \blacksquare \\ & & & & & \blacksquare & \blacksquare \\ & & & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix} \rightsquigarrow Bw(A) = 2$$

Matrix A is called **banded** if it has a “small” bandwidth.

Reorderings - Level-Set Orderings

Given a matrix $A \in \mathbb{R}^{n \times n}$, the following algorithm traverses the adjacency graph of A w.r.t. the so-called **Cuthill-McKee ordering**. This algorithm re-label the vertexes $v \in V$ such that the adjacency matrix of the re-labeled graph has a **bandwidth smaller or equal** to the one of A .

Algorithm's inputs: adjacency graph $G = (V, E)$ and $v \in V$ with lowest degree (i.e. number of edges touching it).

- ④ **Initialize:** $i \leftarrow 1$, $S \leftarrow [v]$, $visited(v) \leftarrow True$, $visited(w) \leftarrow False$ for any $w \in V \setminus \{v\}$;
- ② **while** $i < |V| = n$ **do**:
 - ① $v \leftarrow S[i]$
 - ② $S' \leftarrow []$;
 - ③ **for** w neighbour of v **s.t.** $visited(w) = False$ **do**:
 - ① **append** w to S' and **set** $visited(w) \leftarrow True$;
 - ④ **sort** S' w.r.t. degree-ascending order;
 - ⑤ **append** S' to S ;
 - ⑥ $i \leftarrow (i + 1)$;
- ③ **return** S

Reorderings - Level-Set Orderings

The list of vertexes returned by the **Cuthill-McKee** algorithm let us to define the permutation σ that transforms A into a matrix A' with smaller or equal bandwidth. More specifically:

$$\sigma = \begin{pmatrix} 1 & \cdots & n \\ \ell(S[1]) & \cdots & \ell(S[n]) \end{pmatrix},$$

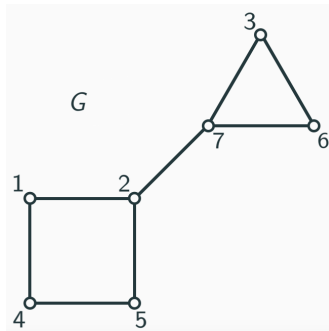
where (remember) ℓ is the **label function** for the graph G .

Reorderings - Level-Set Orderings

Example/Exercise: Consider matrix $A \in \mathbb{R}^{7 \times 7}$ as follows and extract its adjacency graph G .

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & & \blacksquare & & & \\ \blacksquare & \blacksquare & & & \blacksquare & & \blacksquare \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ \blacksquare & & & \blacksquare & \blacksquare & & \\ & \blacksquare & & \blacksquare & \blacksquare & & \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ & \blacksquare & \blacksquare & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

\rightsquigarrow



Run the above algorithm on graph G , with starting vertex $v = 1$, will return the following permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 4 & 2 & 5 & 7 & 3 & 6 \end{pmatrix}.$$

Reorderings - Level-Set Orderings

Example/Exercise (continued): Given the obtained permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 4 & 2 & 5 & 7 & 3 & 6 \end{pmatrix},$$

compute $\sigma(A)$.

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & & \blacksquare & & & \\ \blacksquare & \blacksquare & & & \blacksquare & & \blacksquare \\ \blacksquare & & \blacksquare & & & \blacksquare & \blacksquare \\ \blacksquare & & & \blacksquare & \blacksquare & & \\ & \blacksquare & & \blacksquare & \blacksquare & & \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ & \blacksquare & \blacksquare & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

$$Bw(A) = 5$$

\rightsquigarrow

$$\sigma(A) = \begin{matrix} & \begin{matrix} 1 & 4 & 2 & 5 & 7 & 3 & 6 \end{matrix} \\ \begin{matrix} 1 \\ 4 \\ 2 \\ 5 \\ 7 \\ 3 \\ 6 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & \blacksquare & & & & \\ \blacksquare & \blacksquare & & \blacksquare & & & \\ \blacksquare & & \blacksquare & \blacksquare & \blacksquare & & \\ & \blacksquare & \blacksquare & \blacksquare & & & \\ & & \blacksquare & & \blacksquare & \blacksquare & \blacksquare \\ & & & & \blacksquare & \blacksquare & \blacksquare \\ & & & & \blacksquare & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

$$Bw(\sigma(A)) = 2$$

Reorderings - Level-Set Orderings

Some Observations:

- Choosing a different **initial vertex** v can result in a completely different output.
- As a consequence of the usage of the Cuthill-McKee ordering, an **“arrow” pointing upward** can appear in the top-left corner of the retrieved matrix. A remedy consists in reorder the vertices backward (reverse Cuthill-McKee ordering).

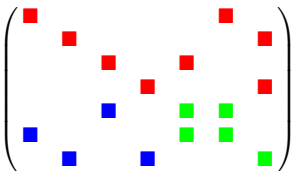
Reorderings - Independent Set Orderings

Another desirable pattern for a matrix A in $\mathbb{R}^{n \times n}$ is characterized by the following block structure for A :

$$A = \begin{bmatrix} D & B \\ C & E \end{bmatrix}$$

with D diagonal matrix and B , C , E sparse matrices.

Example



$$\begin{pmatrix} \text{red} & & & & & & & & & \\ & \text{red} & & & & & & & & \\ & & \text{red} & & & & & & & \\ & & & \text{red} & & & & & & \\ & & & & \text{red} & & & & & \\ & & & & & \text{red} & & & & \\ & \text{blue} & & & & & & & & \\ & & \text{blue} & & & & & & & \\ & & & \text{blue} & & & & & & \\ & & & & \text{blue} & & & & & \\ & & & & & \text{green} & & & & \\ & & & & & & \text{green} & & & \\ & & & & & & & \text{green} & & \\ & & & & & & & & \text{green} & \\ & & & & & & & & & \text{green} \\ & & & & & & & & & & \text{green} \end{pmatrix}$$

Such a characterization of A having a diagonal upper-left block is especially useful in parallel computing, for implementing both direct and iterative methods.

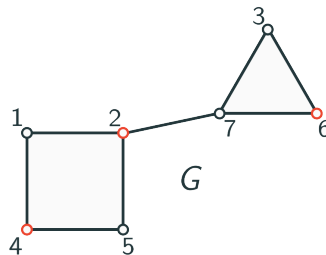
Reorderings - Independent Set Orderings

Given a matrix $A \in \mathbb{R}^{n \times n}$, a matrix similar to A and having a diagonal upper-left block can be obtained with two steps:

- ① retrieving an **independent set** S of vertices of the adjacency graph G ;
- ② **sorting rows and columns** of A moving to the first positions the ones with indices in S .

Definition: Given a graph $G = (V, E)$, an **independent set of vertices** S of G is a subset $S \subseteq V$ such that $(v, w) \notin E$ for all $v, w \in S$, $v \neq w$ (i.e. v and w not adjacent).

Example: $S = \{2, 4, 6\}$ is an independent set of vertices of the graph G depicted on the right.



Reorderings - Independent Set Orderings

Given a graph $G = (V, E)$, an independent set S of vertexes of G can be retrieved by running the following **greedy algorithm**:

- ① **Initialize:** $S = \emptyset$ empty set, $\text{marked}(v) = \text{False}$ for each $v \in V$;
- ② **for** $i = 1, \dots, |V|$ **do**:
 - ① **if** $\text{marked}(v_i) = \text{False}$ **do**:
 - ① $S \leftarrow (S \cup \{v_i\})$;
 - ② $\text{marked}(v_i) \leftarrow \text{True}$ and $\text{marked}(w) \leftarrow \text{True}$ for each w neigh. of v_i ;
- ③ **return** S ;

If G is the adjacency graph of a matrix $A \in \mathbb{R}^{n \times n}$, the output $S = \{v_{i_1}, \dots, v_{i_s}\}$ can be used to define a **permutation** σ such that $\sigma(A)$ will have a **diagonal upper-left block**:

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & s & s+1 & s+2 & \cdots & n \\ i_1 & i_2 & \cdots & i_s & j_1 & j_2 & \cdots & j_{n-s} \end{pmatrix}.$$

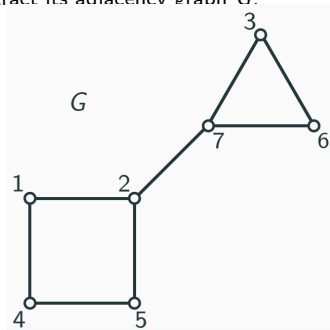
where the set $\{j_1, \dots, j_{n-s}\}$ consists of the labels of vertexes of V not in S .

Reorderings - Independent Set Orderings

Example: Consider matrix $A \in \mathbb{R}^{7 \times 7}$ as follows and extract its adjacency graph G .

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & & \blacksquare & & & \\ \blacksquare & \blacksquare & & & \blacksquare & & \blacksquare \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ \blacksquare & & & \blacksquare & \blacksquare & & \\ & \blacksquare & & \blacksquare & \blacksquare & & \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ & \blacksquare & \blacksquare & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

\rightsquigarrow



A run of the greedy algorithm on graph G returns the following **independent set** of vertexes S of G :

$$S = \{1, 3, 5\}.$$

Reorderings - Independent Set Orderings

Example (continued): The retrieved independent set $S = \{1, 3, 5\}$ induces the following permutation

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 3 & 5 & 2 & 4 & 6 & 7 \end{pmatrix},$$

based on which we can compute $\sigma(A)$.

$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \blacksquare & \blacksquare & & \blacksquare & & & \\ \blacksquare & \blacksquare & & & \blacksquare & & \blacksquare \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ \blacksquare & & & \blacksquare & \blacksquare & & \\ & \blacksquare & & \blacksquare & \blacksquare & & \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ & \blacksquare & \blacksquare & & & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

\rightsquigarrow

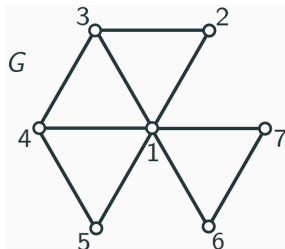
$$\sigma(A) = \begin{matrix} & \begin{matrix} 1 & 3 & 5 & 2 & 4 & 6 & 7 \end{matrix} \\ \begin{matrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \\ 6 \\ 7 \end{matrix} & \begin{pmatrix} \color{red}\blacksquare & & & \blacksquare & \blacksquare & & \\ & \color{red}\blacksquare & & & & \blacksquare & \blacksquare \\ & & \color{red}\blacksquare & \blacksquare & \blacksquare & & \\ \blacksquare & & & \blacksquare & \blacksquare & & \blacksquare \\ \blacksquare & & & & & \blacksquare & \\ & \blacksquare & & & \blacksquare & & \\ & & \blacksquare & & & \blacksquare & \blacksquare \\ & \blacksquare & & & \blacksquare & \blacksquare & \blacksquare \end{pmatrix} \end{matrix}$$

Reorderings - Independent Set Orderings

Remark: in the **for** cycle of the previous greedy algorithm, vertices V of a graph G are traversed in the **natural order** $1, 2, \dots, n$ (of the labels).

An **alternative** strategy for obtaining an independent set S of **larger size**, consists in traversing the vertexes with respect to the order given by the **increasing value of the degrees**.

Exercise: run the two versions of the greedy algorithm on the following graph G .



scipy.sparse - The Sub-Package

For working with sparse matrices in Python, we suggest the usage of the **scipy** package, in particular the sub-package **scipy.sparse**³.
In the next slides, we will report the basic informations also written in the official documentation.

³official online-guide: <https://docs.scipy.org/doc/scipy/reference/sparse.html>.

scipy.sparse - Storage Schemes

The storage schemes in `scipy.sparse` are

① Type 1 storage schemes:

- ① `scipy.sparse.dok_matrix()`: **DOK** based sparse matrix;
- ② `scipy.sparse.lil_matrix()`: Row-based **LIL** sparse matrix;
- ③ `scipy.sparse.coo_matrix()`: sparse matrix in **COO**rdinate format;

② Type 2 storage schemes:

- ① `scipy.sparse.csr_matrix()`: **CSR** matrix;
- ② `scipy.sparse.csc_matrix()`: **CSC** matrix;
- ③ `scipy.sparse.bsr_matrix()`: Block Sparse Row matrix;

③ Type 3 storage schemes:

- ① `scipy.sparse.dia_matrix()`: Sparse matrix with **DIAG**onal storage.

scipy.sparse - Basic Functions

Here we report three **basic functions** of the **scipy.sparse** subpackage. See the official guide to learn more functions.

Underlined arguments are the **optional** ones.

- `scipy.sparse.identity(n, dtype, format)`: creates a sparse identity matrix I_n of data type dtype (default float) and sparse format format (default `dia_matrix`);
- `scipy.sparse.rand(m, n, density, format, dtype, random_state)`: creates a sparse matrix $\mathbb{R}^{m \times n}$ in format format (default `coo_matrix`) with density density (default 0.01).

Equivalently: `scipy.sparse.random()`;

- `scipy.sparse.find(A)`: returns three arrays JR, JC, AA corresponding to the representation of the sparse matrix A in the **COO** format;

scipy.sparse - Suggestions for Matrix Operations I

- **dok_matrix** format:

- PROs:
 - Allows for efficient $O(1)$ access of individual elements
 - Can be efficiently converted to a `coo_matrix` once constructed.

- CONs:
 - Bad for any kind of operation.

- **lil_matrix** format:

- PROs:
 - Supports flexible slicing;
 - Changes to the matrix sparsity structure are efficient.

- CONs:
 - Arithmetic operations `LIL + LIL` are slow (consider `CSR` or `CSC`);
 - Slow column slicing (consider `CSC`);
 - Slow matrix vector products (consider `CSR` or `CSC`);
 - Consider using the `COO` format when constructing large matrices.

- **coo_matrix** format:

- PROs:
 - Facilitates fast conversion among sparse formats;
 - Very fast conversion to and from `CSR/CSC` formats;
 - Fast format for constructing sparse matrices;

- CONs:
 - does not directly support arithmetic operations;
 - does not directly support slicing;

scipy.sparse - Suggestions for Matrix Operations I

- **csr_matrix** format:

- PROs:
 - Efficient arithmetic operations $\text{CSR} + \text{CSR}$, $\text{CSR} * \text{CSR}$, etc.;
 - Efficient row slicing;
 - Fast matrix vector products.
- CONs:
 - Slow column slicing operations (consider CSC);
 - Changes to the sparsity structure are expensive (consider LIL or DOK).

- **csc_matrix** format:

- PROs:
 - Efficient arithmetic operations $\text{CSC} + \text{CSC}$, $\text{CSC} * \text{CSC}$, etc.;
 - Efficient column slicing;
 - Fast matrix vector products (CSR may be faster);
- CONs:
 - Slow row slicing operations (consider CSR);
 - Changes to the sparsity structure are expensive (consider LIL or DOK).

- **dia_matrix** format:

PRO/CONs: depends on the applications.