Politecnico di Torino

Computational Linear Algebra for Large Scale Problems

Homework 3 - Sparse Matrix analysis: from Graphs to optimised Storage Patterns

Aurona Gashi s322791
Elisabetta Roviera s328422

21 January 2025

Contents

1 Introduction

In general terms, a sparse matrix is one that has a minimal number of non-zero entries. More precisely, let $A \in \mathbb{R}^{m \times n}$ 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 has $O(min\{m, n\}^{1+\gamma})$ non-zero entries for some $\gamma \leq 1$.

Therefore, a matrix is considered sparse if its structure allows for specialised techniques that leverage the numerous zero entries and their positions to improve computational efficiency. These techniques are based on the principle of omitting the storage of zero elements.

The present document provides an overview of sparse matrices, including their properties, representations, and the data structures employed for their storage.

The concept of leveraging zero entries in matrices was initially pioneered by engineers, particularly electrical engineers during the 1960s. These engineers employed sparsity to address linear systems characterised by irregular structures. The initial sparse matrix technology focused on the development of efficient direct solution methods, economical in both storage and computation, and capable of handling larger problems than dense solvers.

The classification of sparse matrices is typically divided into two broad categories: structured and unstructured. As demonstrated in Figure 1, a comparison is drawn between a structured sparse matrix (on the left) and an unstructured one (on the right) to highlight the distinction between the two. Structured matrices are characterised by non-zero entries that form regular patterns, often along diagonals or in regularly patterned dense submatrices (blocks). In contrast, unstructured matrices exhibit irregularly positioned entries. A simple example of a structured matrix is one with few diagonals, which is common in finite difference methods on rectangular grids. Conversely, unstructured matrices frequently emerge in finite element/volume methods employed on complex geometries. While this distinction is historically less important for direct solvers, it significantly impacts iterative methods, especially matrix-vector products, where performance on high-performance computers varies based on matrix structure. Diagonal storage is ideal for structured matrices on vector computers, while unstructured matrices may require less efficient indirect addressing.

The ensuing sections will provide an overview of graph representations and storage schemes.

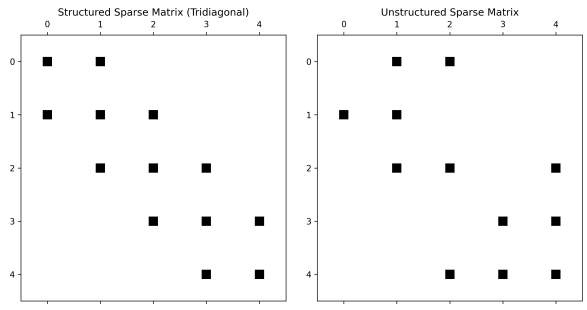

Figure 1: Comparison of a structured and an unstructured sparse matrix

2 Graph Representations

This chapter explores the application of graph theory in the representation of sparse matrices, investigating the efficacy of graph structures as a means to comprehend and manipulate these matrices.

Graph theory is a fundamental tool for representing the structure of sparse matrices and thus plays a key role in techniques related to these matrices. For example, it is used in the identification of parallelism in sparse Gaussian elimination or in preconditioning techniques. As demonstrated in Figure 2, the connection between sparse matrices and their corresponding graph representations can be illustrated by means of both "spy plots" (visualisations of the sparsity pattern) and the associated graphs for a structured and an unstructured sparse matrix. This figure provides a visual comparison of how different matrix structures translate into distinct graph topologies.

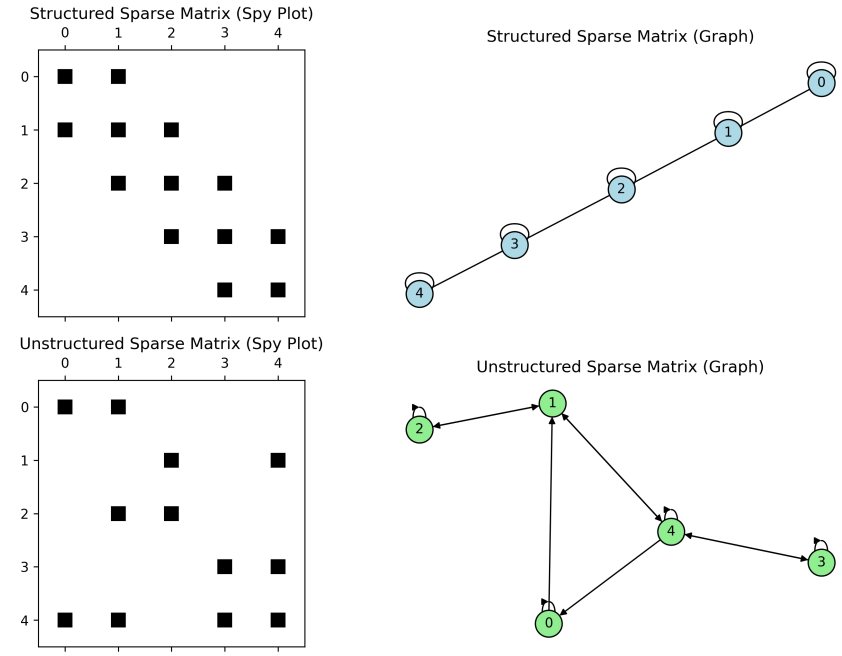

Figure 2: Representation of structured and unstructured sparse matrices with respective spy plots (left) and graphs (right)

2.1 Graphs and Adjacency Graphs

A graph is defined by a set of vertices V and a set of arcs E, where the arcs are pairs of vertices (v_i, v_j). A graph $G = (V, E)$ represents a binary relationship between objects in V. For example, if V represents the world's major cities, an arc between two cities indicates a direct flight between them. If the relationship is symmetric (i.e. if there is a flight from A to B, there is also a flight from B to A), the graph is undirected; otherwise it is directed.

In the context of sparse matrices, the adjacency graph of a sparse matrix $G = (V, E)$ has n vertices representing the n unknowns. The arcs represent the binary relations defined by the following condition: there is an arc from i to j if $a_{ij} \neq 0$, indicating that equation i involves unknown j. The graph is undirected if the matrix is symmetric ($a_{ij} \neq 0$ if and only if $a_{ji} \neq 0$). In that case, each arc points in both directions and can

The adjacency matrix is a square matrix (i.e. with the same number of rows and columns) in which both rows and columns represent nodes in the graph. The element in row i and column j of the matrix indicates whether an arc exists between node i and node j

be represented without orientation.

Graphical models are useful, for example, to extract parallelism in Gaussian elimination by identifying independent unknowns (i.e. unknowns which do not depend on each other according to the binary relation). The corresponding rows can be used as pivots simultaneously. In a diagonal matrix, all unknowns are independent; in a dense matrix, each unknown depends on all others. Sparse matrices lie between these two extremes. The graph of A^2 represents the pairs (i, j) for which there is a path of length two from i to j in the graph of A; similarly, the graph of A^k represents paths of length k.

3 Permutations and Reorderings

In the preceding chapters, the concept of sparse matrices and their representation by adjacency graphs was introduced. In order to optimise performance, especially in the context of parallel computing, it is imperative to reorder the elements of a matrix. This operation, formally described as a permutation of rows and/or columns, makes up the central focus of this chapter. The ensuing analysis will explore reordering techniques and their close relationship to adjacency graphs, demonstrating how a permutation can be interpreted and visualised in the corresponding graph. This is a fundamental aspect for the parallel implementation of direct and iterative solution techniques. Formally, given a matrix $A \in \mathbb{R}^{n\times n}$ and a permutation $\pi \in S_n$, the reordered matrix $\pi(A)$ is defined as

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

where $E^{(\pi)} \in GL_n(\mathbb{R})$ is the matrix obtained from the identity matrix I_n reodering its rows according to π. Where S_n is the symmetrical group of degree n (the set of all possible permutations of n objects), and $GL_n(\mathbb{R})$ represents the General Linear Group of degree n on the real numbers (the set of all square matrices with real elements that are invertible).

Let A be a matrix and $\pi = \{i_1, \ldots, i_n\}$ a permutation of the set $\{1, \ldots, n\}$. Then the matrices

$$A_{\pi,*} = \{a_{\pi(i),j}\}_{i=1,\ldots,n;j=1,\ldots,m}$$
$$A_{*,\pi} = \{a_{i,\pi(j)}\}_{i=1,\ldots,n;j=1,\ldots,m}$$

are called row π-permutation and column π-permutation of A, respectively.

It is established that each possible permutation of the set $\{1, 2, \ldots, n\}$ is the result of at most n exchanges, defined as elementary permutations in which only two elements are exchanged. An exchange matrix is defined as the identity matrix with two rows exchanged; such matrices are denoted by X_{ij}, where i and j are the indices of the exchanged rows. It is sufficient to premultiply a matrix A by X_{ij} in order to exchange rows i and j. An arbitrary permutation $\pi = \{i_1, \ldots, i_n\}$ is the product of a sequence of n consecutive exchanges $\sigma(i_k, j_k)$, $k = \{1, \ldots, n\}$. Therefore, the rows of a matrix can be permuted by performing successive exchanges. A similar process can be used to exchange columns i and j of a matrix, with the matrix being postmultiplied by X_{ij}.

Let π be a permutation resulting from the product of the interchanges $\sigma(i_k, j_k)$, $k = \{1, \ldots, n\}$. Then

$$A_{\pi,*} = P_\pi A, \quad A_{*,\pi} = AQ_\pi$$

where

$$P_\pi = X_{i_n,j_n} \ldots X_{i_1,j_1}$$

$$Q_\pi = X_{i_1,j_1} \dots X_{i_n,j_n}$$

The products of exchange matrices are known as permutation matrices, which are defined as identity matrices with permuted rows (or columns). Furthermore, it is demonstrated that the square of an exchange matrix X_{ij} is the identity matrix ($X_{i,j}^2 = I$), thus establishing that the inverse of an exchange matrix is itself. In the event of two permutation matrices, P_π, Q_π, being derived from the same permutation, it can be deduced that

$$P_\pi Q_\pi = X_{i_n,j_n} \dots X_{i_1,j_1} \times X_{i_1,j_1} \dots X_{i_n,j_n} = I$$

Thus P_π and Q_π are non-singular and inverse of each other. It is important to note that permuting rows and columns with the same permutation is equivalent to a similarity transformation. Since the exchange matrices are assumed to be symmetric and the products defining P_π and Q_π occur in reverse order, it can be deduced that Q_π is the transpose of P_π

$$Q_\pi = P_\pi^T = P_\pi^{-1}$$

thereby ensuring the permutation matrices are unitary. The P_π and P_π^T matrices can be expressed in terms of the identity matrix with permuted rows/columns

$$P_\pi = I_{\pi,*}, \quad P_\pi^T = I_{*,\pi}$$

It is then possible to verify directly that

$$A_{\pi,*} = I_{\pi,*}A = P_\pi A, \quad A_{*,\pi} = AI_{*,\pi} = AP_\pi^T$$

The permutation of the rows changes the order of the equations, while that of the columns renames the unknowns. It is important to note that one-sided (row-only or column-only) permutations are less common than "bilateral" (symmetric) permutations, where the same permutation is applied to rows and columns, corresponding to renaming the unknowns and reordering the equations in the same way.

3.1 Relations with the Adjacency Graph

From the perspective of graph theory, a symmetric permutation can be considered as equivalent to the renaming of the vertices of a graph, without altering the arcs. As demonstrated in Figure 3, the concept is illustrated by displaying an original graph and its adjacency matrix, in comparison with the resulting graph after a permutation and the corresponding permuted adjacency matrix. It is evident that the structure of the graph (the connections between the nodes) remains unaltered, while the labels of the nodes are subject to change and, consequently, the order of the rows and columns in the adjacency matrix is also subject to alteration. In the context of the adjacency graph of the original matrix A, if (i, j) is an arc and A' is the permuted matrix, then $a'_{ij} = a_{\pi(i),\pi(j)}$. Therefore, (i, j) is an arc in the graph of A' if and only if $(\pi(i), \pi(j))$ is an arc in the graph of A. In practice, it is as if each node with the 'old' label $\pi(i)$ is simply relabelled with the 'new' label i. Thus, the graph of the permuted matrix remains unchanged, only the labelling of the vertices is altered.

Conversely, non-symmetric permutations do not preserve the graph and have the capacity to transform an undirected graph into a directed one.

Symmetric permutations modify the order in which nodes are considered in an algorithm (e.g. Gaussian elimination), with significant consequences for the performance of

the algorithm. Finally, it is important to note that bilateral non-symmetric permutations can occur, but are more common in directed methods.

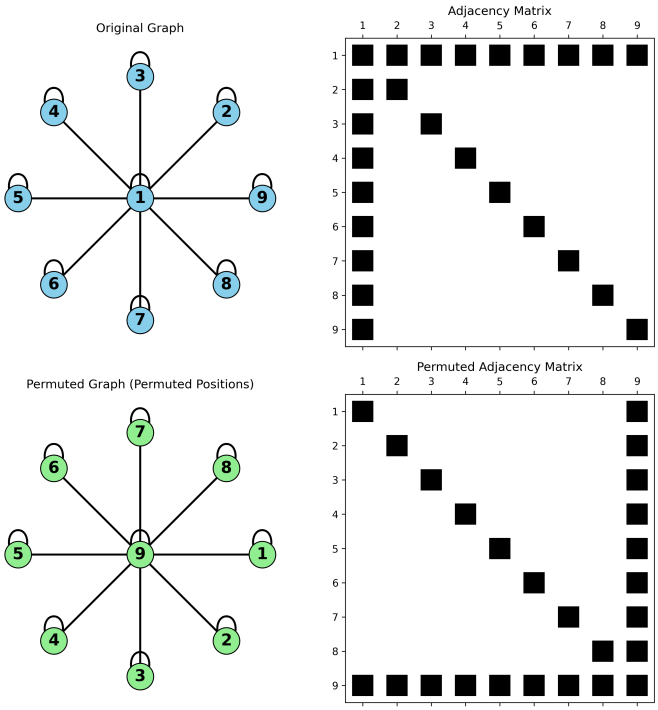

Figure 3: The effect that a permutation has on the nodes of a graph and its adjacency matrix

3.2 Common Reorderings

The type of reordering, or permutation, employed in applications is contingent on whether a direct or iterative method is considered. The following are examples of such reordering that are most useful for iterative methods.

3.2.1 Level-set orderings

The 'level-set' (or 'layer-based') sorting methods are based on exploring the graph by visiting its nodes in successive levels, starting with one or more initial nodes. This approach builds sets of nodes ('level sets') in an iterative manner, where each set contains the unvisited neighbours of the nodes of the previous level, allowing a 'layered' exploration of the graph. The final ordering of the nodes is determined by the order in which the nodes within each level are visited and the order in which the levels themselves are considered, thus yielding multiple variants of this method. In essence, level-set ordering provides a systematic approach to traversing a graph by organising the nodes according to their 'distance' from the initial point, with this organisation influencing the effectiveness of certain iterative algorithms.

Breadth-First Search (BFS) The term 'breadth-first' refers to the manner in which the algorithm explores the nodes of the graph 'in layers', visiting all the neighbours of a node before progressing to the neighbours of its neighbours, and so on. To illustrate this, one may imagine throwing a stone into a pond; the resulting waves expand circularly,

reaching first the neighbouring points and then those further away. The BFS functions in a similar manner.

Algorithm 1 BFS(G,v)

Initialize $S \leftarrow \{v\}$, $seen \leftarrow 1$, $\pi(1) \leftarrow v$; Mark v;
while $S \neq \emptyset$ **do**
  $Snew \leftarrow \emptyset$;
  for $v \in S$ **do**
    for unmarked $w \in adj(v)$ **do**
      Add w to $Snew$;
      Mark w;
      $seen \leftarrow seen + 1$;
      $\pi(seen) \leftarrow w$;
    end for
  end for
  $S \leftarrow Snew$;
end while=0

The Algorithm 1 commences from an initial node v, which is designated as the first level (or level 0), and maintains a register of the nodes that have been previously visited. A set S is utilised for the storage of the nodes belonging to the current level. The primary loop (While seen $< n$) is initiated and continues until all n nodes of the graph have been visited. At each iteration of the main loop, a new empty set S_{new} is created, which will contain the nodes of the subsequent level.

All nodes v in the set S (i.e., the nodes of the current level) are iterated over. For each node v, its unmarked (i.e. not yet visited) neighbours w are checked.

If a neighbour w is unmarked, it is added to the set S_{new} (to be visited at the next level), marked as visited, and its 'predecessor' is stored in the vector π. Specifically, $\pi(seen)$ is set to w, where seen is a counter that is incremented each time a new node is marked. This π vector will eventually contain the information required to reconstruct the path from the source to every other node.

Subsequent to the examination of all neighbours of each node in the current level, the set S is updated with S_{new}, thus progressing to the next level. In summary, the algorithm explores the graph level by level, storing the order of visits and constructing a spanning tree through the vector π. This tree represents the minimum paths (in terms of number of arcs) from the starting node to all other reachable nodes.

The BFS algorithm 3.2.1, when implemented in Python, aims to visit all nodes reachable from a given starting node in a graph, effectively finding the shortest path (in terms of the number of edges) between that starting node and every other reachable node.

```
1 def bfs(graph, start_node):
2     if start_node not in graph:  # Check if the starting node exists
          in the graph
3         print(f"Error: Starting node '{start_node}' not found in the
              graph.")  # Print an error message if the start node is
              not found
4         return None  # Return None to indicate an error
5
```

```
    s = deque([start_node])  # Initialize a deque for BFS, starting
        with the start node
    seen = 1  # Initialize a counter to track the order in which
        nodes are visited
    pi = {seen: start_node}  # Initialize a dictionary to map visit
        order to nodes
    marked = {start_node}  # Initialize a set to keep track of
        visited/marked nodes
    bfs_steps = []  # Initialize a list to store the graph at each
        BFS step

    while s:  # While the queue is not empty
        s_new = deque()  # Initialize a new deque for the next level
            of BFS
        current_step_graph = nx.Graph()  # Create a new empty graph
            for the current BFS step
        current_step_graph.add_nodes_from(marked)  # Add the already
            marked nodes to the current step graph

        for node in marked:
            if node in graph:
                for neighbor in graph[node]:
                    if neighbor in marked:
                        current_step_graph.add_edge(node, neighbor) #
                            Add the edge to the graph of the current
                            step
        bfs_steps.append(current_step_graph)  # Append the current
            state of the graph to the list of steps

        for v in list(s):  # Iterate through the nodes in the current
            queue
            if v in graph:
                for w in graph[v]:  # Iterate through the neighbors
                    of the current node v
                    if w not in marked:
                        s_new.append(w)  # Add the neighbor w to the
                            new queue for the next level
                        marked.add(w)  # Mark the neighbor w as
                            visited
                        seen += 1  # Increment the visit order
                            counter
                        pi[seen] = w  # Map the visit order to the
                            neighbor w in the pi dictionary
        s = s_new  # Update the queue for the next level of BFS
marked = set of all the node that have been visited   pi = a dictionary that maps an exploration order to the corresponding node
                                                      encountered during the BFS traversal. It provides a way to retrieve the nodes
                                                      in the order they were visited by the BFS algorithm.
    return pi, marked
```

As illustrated in Figure 4, the BFS algorithm can be utilised in a practical manner. Initiating from the initial node A (depicted in green), the algorithm methodically traverses the graph level by level, visiting all the neighbours of a node prior to progressing to the neighbours of its neighbours. The nodes that are visited at each iteration are highlighted in green. This visualisation facilitates an intuitive comprehension of the algorithm's functionality and its approach to expanding the exploration of the graph.

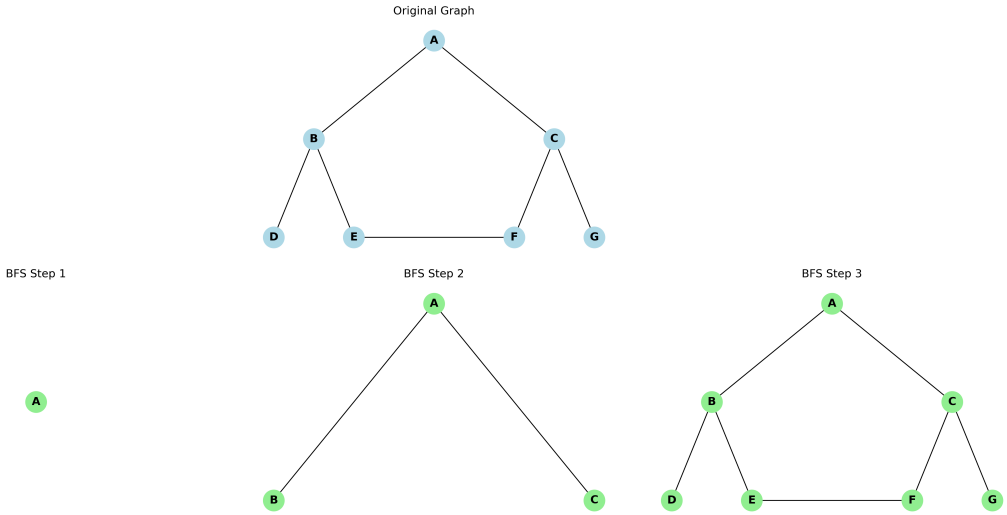

Figure 4: Execution steps of the BFS algorithm

3.2.2 Independent Set Orderings (ISO)

In finite element problems, the matrices that emerge often have a block structure with a diagonal upper left block, which can be represented as:

$$A = \begin{pmatrix} D & E \\ F & C \end{pmatrix}$$

where D is diagonal and C, E and F are sparse matrices. The upper diagonal block corresponds to unknowns from previous refinement levels and is a consequence of the ordering of the equations. Adding new vertices to the refined grid involves assigning new numbers, while keeping the initial vertex numbering unchanged. The old connected vertices are 'cut off' by the new ones, thereby losing their direct relationships through equations. Such sets of vertices that are not directly connected are called independent sets, and these sets are particularly useful in parallel calculations, both for direct and iterative methods.

In the context of an adjacency graph $G = (V, E)$, an independent set S is defined as a subset of V satisfying the following criteria:

$$x \in S \implies \{(x, y) \in E \vee (y, x) \in E\} \implies y \notin S$$

In other words, no element of S is connected to other elements of S by incoming or outgoing arcs. An independent set is said to be maximal if it cannot be extended with elements of its complement to form a larger independent set. It is important to note that a maximal independent set is not necessarily the independent set of maximal cardinality (the search for which is an NP-hard problem). In the following, the term 'independent set' always refers to a maximal independent set.

Greedy Algorithm for ISO The objective of the greedy algorithm for constructing a maximal independent set (ISO) is to identify a set S of high, though not necessarily maximum, cardinality, utilising simple and cost-effective heuristics. The greedy algorithm undertakes an examination of nodes in a predetermined order (typically the natural order $\{1, 2, \ldots, n\}$, but also a permutation). For each examined node:

- If the node has not yet been marked (and thus has not yet been included in S nor have its neighbours been marked), the node is selected as a new member of S.
- The selected node and all its nearest neighbours (i.e. nodes connected to it by incoming or outgoing arcs) are marked, preventing its subsequent inclusion in S.

The termination of the algorithm occurs upon the examination of all nodes. This approach ensures the construction of a maximal independent set, with the size of the resulting reduced set being $n - |S|$, where n denotes the total number of nodes and $|S|$ represents the cardinality of the found independent set. Consequently, an implicit objective of the algorithm is to maximise the cardinality of S to minimise the size of the reduced system.

The Greedy Algorithm 2 that is described forms the basis for more advanced heuristics that aim to improve the cardinality of the independent set found, e.g. by considering the visiting order of nodes according to their degree.

Algorithm 2 Greedy ISO

Initialize $S \leftarrow \emptyset$;
for $j = 1$ to n **do**
  if `marked`$[j] =$ **false then**
    $S \leftarrow S \cup \{j\}$
    Mark j and all its nearest neighbors;
  end if
end for=0

The algorithm is characterised by its 'greedy' nature, in that at each step it selects the first unmarked vertex it encounters. This is without consideration of whether a different choice might lead to a larger independent set. Consequently, the Greedy ISO algorithm 3.2.2, when implemented in Python, does not guarantee finding the maximum independent set, but rather provides an approximate solution efficiently.

```python
def greedy_iso(adjacency_matrix, pos):
    n = len(adjacency_matrix)  # Number of nodes in the graph
    S = []  # Initialize the independent set (empty at the beginning)
    marked = [False] * n  # List to keep track of visited/marked nodes (initially all False)
    original_graph = nx.from_numpy_array(np.array(adjacency_matrix)) # Create a networkx graph from the adjacency matrix
    subgraphs = [] # list of the subgraphs at each step

    for j in range(n):  # Iterate through all nodes
        if not marked[j]:  # If the current node j is not marked (not yet in the independent set or a neighbor of one)
            S.append(j)  # Add node j to the independent set S
            marked[j] = True  # Mark node j as visited/included in the independent set

            subgraph = original_graph.subgraph(S).copy() # Create a subgraph with the nodes of S and copy it
            subgraphs.append(subgraph) # Add the subgraph to the list of subgraphs
```

```
15
16              for i in range(n):  # Iterate through all other nodes to
                    check for neighbors of j
17                  if adjacency_matrix[j][i] == 1:  # If there's an edge
                        between node j and node i (i is a neighbor of j)
18                      marked[i] = True  # Mark node i as visited/a
                            neighbor of a node in the independent set
19
20      return S, original_graph, subgraphs  # Return the independent set
            , the original graph and the list of subgraphs
```

As illustrated in Figure 5, the Greedy ISO algorithm is executed in the following manner. Initially, Node 0 is selected. This is followed by the addition of Node 3. Subsequently, Node 4 is added. Finally, Node 6 is selected. The blue nodes represent the independent set that is found by the algorithm.

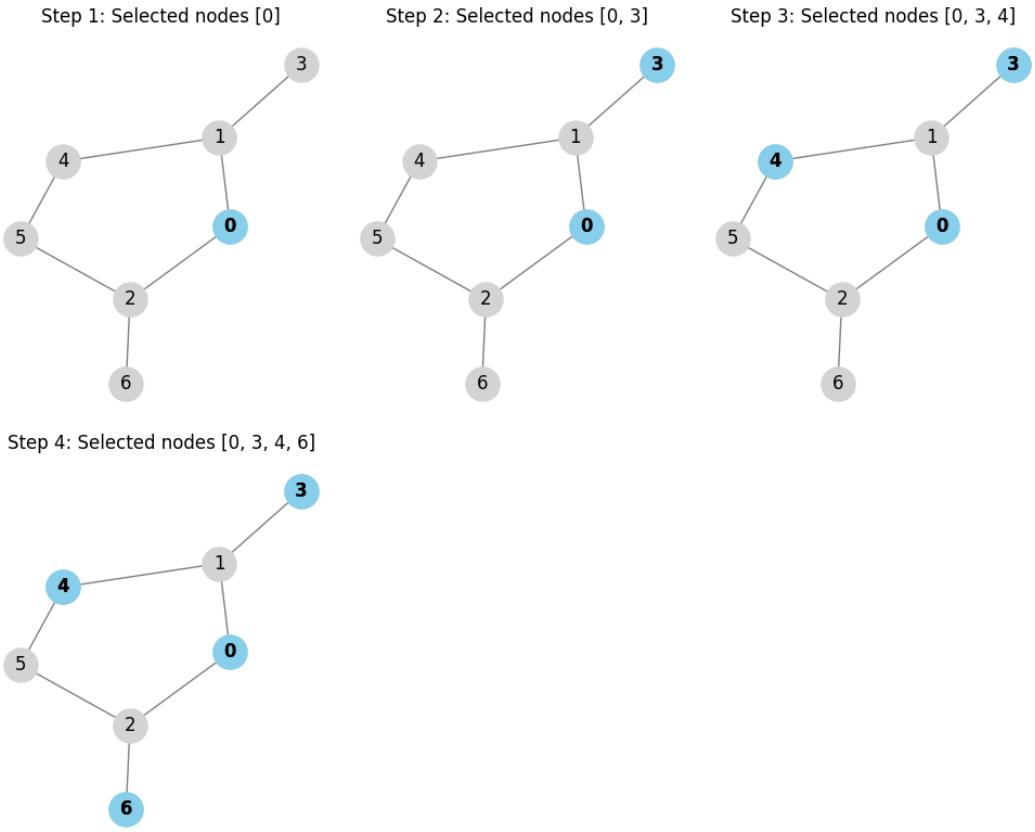

Figure 5: Illustration of the Greedy ISO algorithm for finding an independent set

3.2.3 Multicolor orderings

Graph colouring is a fundamental problem in computer science that involves assigning labels, or "colours," to the vertices (nodes) of a graph such that no two adjacent vertices share the same colour. The primary objective of graph colouring is typically to minimise the number of distinct colours used, achieving an optimal colouring. However, in the context of numerical linear algebra, the emphasis shifts from strict optimality to computational efficiency. Consequently, heuristic approaches that provide adequate, though not necessarily minimal, colorings are often preferred. These "multicolor orderings" are

particularly relevant in parallel computing for tasks such as matrix computations, where coloring can determine independent sets of operations that can be performed concurrently. Generating multicoloring for structured grids, a common occurrence in numerical simulations, can be achieved with relatively simple greedy techniques.

Greedy Multicoloring Algorithm The Greedy Multicolouring algorithm is a rudimentary yet efficacious technique for attaining multicolouring. This algorithm functions by iteratively allocating the least available colour to each node, thereby ensuring that adjacent nodes do not share the same colour. In contrast to the classical graph colouring problem, which aims to minimise the total number of colours used (i.e. to find the chromatic number), the Greedy Multicolouring algorithm prioritises the efficient assignment of valid colours. However, it should be noted that this does not guarantee the use of the smallest possible number of colours.

Algorithm 3 Greedy Multicoloring Algorithm

1: **for** $i = 1, \ldots, n$ **do**
2: Set $Color(i) = 0$
3: **end for**
4: **for** $i = 1, 2, \ldots, n$ **do**
5: Set $Color(i) = \min\{k > 0 \mid k \neq Color(j), \forall j \in Adj(i)\}$
6: **end for**=0

The procedure, described in the Algorithm 3, effectively assigns the lowest permissible color to each node in sequence. "Permissible" denotes a positive integer not already used by any of the node's neighbours. The order in which nodes are processed ($\{1, 2, \ldots n\}$ in the basic algorithm) can be generalized to any permutation $\{i_1, \ldots, i_n\}$ of the node indices. This traversal order significantly influences the resulting coloring.

For bipartite graphs (graphs that can be colored with only two colors), a BFS traversal will yield an optimal two-color (often referred to as Red-Black) ordering. Furthermore, for bipartite graphs, any traversal that visits an unmarked node adjacent to at least one marked node at each step will also produce a valid two-coloring. In the general case of non-bipartite graphs, the number of colors required by this greedy algorithm is guaranteed not to exceed the maximum degree of any node in the graph plus one.

```
def greedy_multicoloring(graph):
  if not graph:
    return None  # Handle empty graph case

  # Convert adjacency list to NetworkX graph object (optional, for
      clarity)
  G = nx.Graph(graph)
  n = len(G.nodes)  # Get the number of nodes in the graph

  # Initialize color assignments for each node
  color = {node: 0 for node in G.nodes}

  # Iterate through each node in the graph
  for i in G.nodes:
    # Find the colors of neighboring nodes (already assigned colors)
    neighbor_colors = set()
```

```
    for j in G.neighbors(i):
      if color[j] != 0:
        neighbor_colors.add(color[j])

    # Find the next available color that doesn't conflict with
      neighbors
    k = 1
    while k in neighbor_colors:
      k += 1
    color[i] = k

  return color
```

The Figure 6 delineates the operation of the Greedy Colouring Algorithm 3.2.3, implemented in Python, on an undirected graph. The subplots demonstrate the evolution of colouring, whereby at each step a node not yet coloured is selected and assigned the minimum colour not yet used by its neighbours. Nodes not yet coloured are shown in light grey.

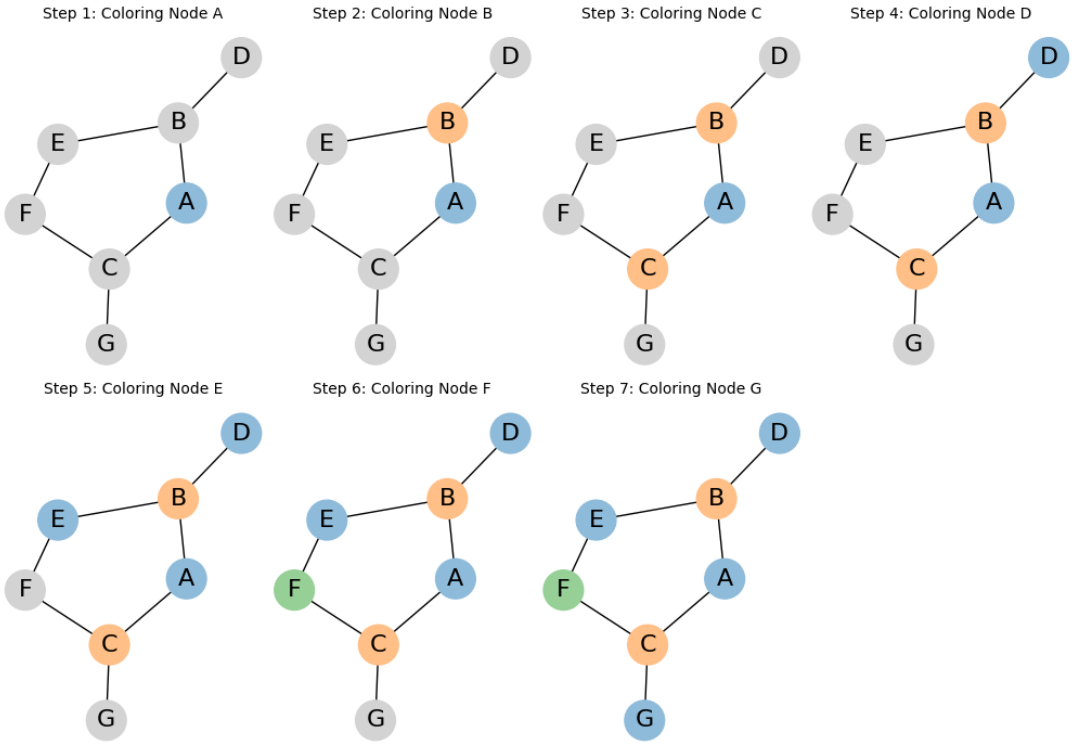

Figure 6: Step-by-step execution of the Greedy Colouring Algorithm

3.3 Irreducibility

In the field of graph theory, a path is defined as a sequence of connected vertices. A graph is said to be connected if there exists a path between any two vertices. A connected component in a graph is defined as a maximal subset of vertices that are all mutually connected.

A matrix is deemed reducible if its corresponding graph is not connected. This can be expressed as a block upper triangular matrix following a symmetric permutation. This

implies that the linear system associated with this matrix can be solved by sequentially solving a series of smaller subsystems represented by the diagonal blocks on the matrix.

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} & \cdots \\ & A_{22} & A_{23} & \cdots \\ & & \ddots & \vdots \\ & & & A_{pp} \end{pmatrix}$$

4 Storage Schemes

This chapter explores storage methodologies for sparse matrices, with a particular focus on three primary formats: Coordinate Format (COO), Compressed Sparse Row (CSR), and Compressed Sparse Column (CSC). Sparse matrices, distinguished by a substantial number of null elements, pose considerable challenges in scientific computing and necessitate efficient management to address complex problems in diverse application domains. To address the limitations imposed by dense representations, these specific storage schemes have been developed, which are implemented in Python and accompanied by practical examples to illustrate their operation.

4.1 Coordinate Format (COO)

In the COO, for each non-zero element, the following is stored.

- Its value: the actual number contained in the cell.
- The row index: the row to which the element belongs.
- The column index: the column to which the element belongs.

Typically, these three pieces of information are stored in three separate arrays.

- An array for values contains the numeric values of non-zero elements.
- An array for row indices contains the row indices corresponding to the values in the previous array.
- An array for column indices contains the column indices corresponding to the values in the previous array.

The memory space required by the COO format is $O(Nz)$, where Nz represents the number of non-zero elements in the matrix. The COO was implemented manually in the 4.1 and then the Python command 'coo_matrix' was used to test its correctness following the implementation of the Python framework, which adopted zero-based indexing.

```
def coordinate_format(matrix):
    # Find the positions of the non-zero elements
    non_zero_positions = matrix.nonzero()

    # Extract the non-zero values from the matrix
    AA = matrix[non_zero_positions]

```

```
    # Get the row and column indices of the non-zero values
    JR = non_zero_positions[0] + 1
    JC = non_zero_positions[1] + 1
```

Let us then show an example. Considering the matrix A, we can obtain the vectors AA, JR, JC using the Python code 4.1.

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

Non-zero values (AA): $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$
Row indices (JR): $[0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4]$
Column indices (JC): $[0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4]$

Despite its simplicity, the COO format is often inefficient due to the redundancy of information. The CSR and CSC formats represent a significant step forward in this field. These schemes exploit the idea of storing only non-zero elements, organising them by rows or columns respectively. This compression saves considerable memory space and improves computational performance, especially for typical operations such as matrix-vector multiplication.

4.2 Compressed Sparse Row (CSR) Format

The CSR Format is a common method of efficiently representing sparse matrices. This format involves storing only the non-zero elements of a matrix, organised row by row. To represent a sparse matrix in CSR format, three arrays are required.

- Values (AA): An array of size equal to the total number of non-zero elements in the array. Each element of the array contains the $a_{i,j}$ value of a non-zero element of the original array, sorted according to the lexicographic order of their row-column indices.
- Column indices (JA): An array of the same size as AA. Each element of JA indicates the column to which the corresponding element in AA belongs.
- Row indices (IA): An array of the same size as the number of rows in the array plus one. The array is defined recursively as follows: $IA[1] = 1$, $IA[i+1] = IA[i]+$ the number of non-zero elements in row i of A, for $i = \{1, \ldots, m\}$.

The element IA(i) indicates the index in AA and JA from which the non-zero elements of the i-th row are found, with the last element of IA indicating the index following the last non-zero element of the matrix. The advantages of this format are two-fold. Firstly, it is efficient for operations on rows: many operations on the rows of a sparse matrix can be performed efficiently using the CSR format. Secondly, it is compact: it stores only the non-zero elements, saving space.

The CSR format is also known as Compressed Row Storage (CRS) and Yale Format. 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

m = number of rows

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

The required memory space is $O(Nz+m)$.

The CSR was implemented manually 4.2 and then the Python command 'csr_matrix' was used to test its correctness. Following the implementation of the Python framework, which adopted zero-based indexing, the array containing the column indices commenced at 0. Consequently, the initial IA element was assigned a value of 0. This resulted in a one-unit discrepancy between the numerical values of the indices and the theoretical description, which utilises a base-one convention.

```
def csr_manual(matrix):
    rows, cols = matrix.shape  # Get the number of rows and columns
       of the input matrix
    nnz = np.count_nonzero(matrix)  # Count the number of non-zero
       elements in the matrix

    AA = np.zeros(nnz)  # Initialize the AA array (values) with zeros
       . Its size is equal to the number of non-zero elements
    JA = np.zeros(nnz, dtype=int)  # Initialize the JA array (column
       indices) with zeros
    IA = np.zeros(rows + 1, dtype=int)  # Initialize the IA array (
       row pointers) with zeros

    index = 0  # Initialize an index to keep track of the current
       position in the AA and JA arrays.
    for i in range(rows):
        IA[i] = index  # Set the i-th element of IA to the current
           index. This marks the beginning of the non-zero elements
           for row i in AA and JA.
        for j in range(cols):
            if matrix[i, j] != 0:
                AA[index] = matrix[i, j]  # If it's non-zero, store
                   its value in AA at the current index
                JA[index] = j  # Store its column index in JA at the
                   current index
                index += 1  # Increment the index to point to the
                   next position in AA and JA
    IA[rows] = index  # After processing all rows, set the last
       element of IA to the final index value. This marks the end of
       the non-zero elements in AA and JA

    return AA, JA, IA
```

Let us then show an example. Considering the matrix A, we can obtain the vectors AA, JA, IA using the Python code 4.2.

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

Non-zero values (AA): $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$

Column indices (JA): $[0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4]$

Row indices (IA): $[0, 2, 5, 9, 11, 12]$

4.3 Compressed Sparse Column (CSC) Format

The CSC Format is the counterpart of the CSR Format, but organises non-zero elements by columns instead of rows. The data structure is very similar to the CSR, but with a slightly different interpretation of the indices. Similarly to CSR, the CSC Format uses three arrays:

- Values (AA): An array containing the values of non-zero elements.
- Row Indices (IA): An array of the same size as Values. Each element of this vector indicates the row to which the corresponding element in value belongs.
- Column Indices (JA): An array of size equal to the number of columns in the matrix plus one. The array is defined recursively as follows: $JA[1] = 1, JA[j+1] = JA[j]+$ the number of non-zero elements in column j of A, for $j = \{1, ..., n\}$. The last element, $JA[n+1]$, is the number of non-zero entries in the matrix plus 1.

The CSC format is particularly suitable for representing sparse matrices when column-oriented operations are dominant. By organizing non-zero elements by columns, it optimizes access to these elements, making it highly efficient for operations such as matrix-vector multiplication with a column vector. Complementary to the CSR format, which optimizes row-oriented operations, the choice between CSC and CSR depends on the primary access pattern: CSR is generally preferable for row-based operations, while CSC is most efficient for column-based ones.

The CSC was implemented manually 4.3 and then the Python command 'csc_matrix' was used to test its correctness. Similar to the CSR case, we indexed the rows starting from 0 and set the first element of Column Indices to 0.

```
def csc_manual(matrix):
    rows, cols = matrix.shape  # Get the number of rows and columns
        of the input matrix
    num_non_zero_elements = np.count_nonzero(matrix)  # Count the
        number of non-zero elements in the matrix

    # Allocate arrays for the CSC data
    data = np.zeros(num_non_zero_elements)  # Initialize the 'data'
        array (values) with zeros
    row_indices = np.zeros(num_non_zero_elements, dtype=int)  #
        Initialize the 'row_indices' array with zeros
    column_pointers = np.zeros(cols + 1, dtype=int)  # Initialize the
         'column_pointers' array with zeros

    current_index = 0
    for col_index in range(cols):
        column_pointers[col_index] = current_index  # Set the current
             element of 'column_pointers' to the current index. This
            marks the beginning of the non-zero elements for the
            current column
        for row_index in range(rows):
            if matrix[row_index, col_index] != 0:
                data[current_index] = matrix[row_index, col_index]  #
                    If it's non-zero, store its value in 'data' at
                    the current index
                row_indices[current_index] = row_index  # Store its
                    row index in 'row_indices' at the current index
```

```
                current_index += 1  # Increment the index to point to
                    the next position in 'data' and 'row_indices'
    column_pointers[cols] = current_index  # After processing all
        columns, set the last element of 'column_pointers' to the
        final index value. This marks the end of the non-zero elements

    return data, row_indices, column_pointers
```

Let us then show an example. Considering the matrix A, we can obtain the vectors data, row_indices, column_pointers using the Python code 4.3.

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix}$$

Non-zero values (data): $[1, 3, 6, 4, 7, 10, 2, 5, 8, 11, 9, 12]$
Column indices (column_pointers): $[0, 3, 4, 6, 10, 12]$
Row indices (row_indices): $[0, 1, 2, 1, 2, 3, 0, 1, 2, 3, 2, 4]$

5 Conclusions

This study examined the notion of sparse matrices, with a particular focus on their properties, representations, and storage schemes, paying particular attention to reordering techniques based on graph theory. Sparse matrices, defined by a high percentage of null elements, offer substantial advantages in terms of computational and memory efficiency over dense matrices, particularly when dealing with large problems. The study identified two primary classes of sparse matrices: structured matrices, which exhibit predictable patterns of zeros, and unstructured matrices, which feature irregularly arranged zeros. A pivotal aspect pertains to the interconnection with graph theory, which furnishes a formidable instrument for depicting the configuration of sparse matrices and facilitating techniques such as Gaussian elimination.

The influence of graph theory on reordering strategies was examined. Level-set (or layer-based) sorting, such as Breadth-First Search, involves exploring the graph starting with one or more initial nodes, visiting 'layered' nodes and generating a sort based on distance from the starting nodes. This approach is useful for reducing matrix bandwidth and improving convergence of iterative methods. Sorting based on Independent Sets aims to identify groups of non-adjacent nodes in the graph, corresponding to variables that can be processed in parallel. This approach is particularly advantageous for direct and iterative methods on parallel architectures, as it allows independent operations to be performed simultaneously. Finally, Multicolour Sorting assigns 'colours' to the nodes of the graph so that adjacent nodes have different colours. This approach creates partitions that can be processed in parallel, thus optimising performance, especially in distributed computing contexts. The choice of reordering method depends heavily on the structure of the matrix and the solution method employed.

In the context of storage analysis, a range of representations were examined, encompassing the Coordinate Format, the more efficient Compressed Sparse Row and the Compressed Sparse Column formats. The CSR format organises non-zero elements by rows, while the CSC format organises them by columns, offering specific advantages depending on whether the primary operations involve rows or columns. Finally, the analysis

of irreducibility facilitates the determination of the decomposition of sparse matrices into smaller subproblems, thereby enhancing the efficiency of the solution.

In conclusion, an understanding of sparse matrices, their representations, and, above all, reordering techniques based on graph theory, is fundamental for optimising numerical calculations in a wide range of applications. By exploiting these techniques, increasingly complex problems can be addressed, thereby maximising computational efficiency and memory usage.

In order to facilitate a comprehensive reproduction of the results, all figures and source codes have been integrated and made accessible within an interactive notebook [Sparse Matrices.ipynb].

References

[1] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[2] Berrone, S., Della Santa, F., & Fugacci, U. (2024). *Dense and Sparse Matrices*. Presentation slides for Computational Linear Algebra For Large Scale Problems, Politecnico di Torino.