Assembling Adjacency Relations for the Finite Element Method

P. Zajac[*]

July 9, 2013

Abstract

This work demonstrates the concept of describing adjacency relations as set-valued mappings, which has already been used in graph algorithm implementations for a long time. The primary goal is to provide a way to express matrix sparsity patterns arising from finite element discretisations on unstructured meshes in a directly implementable manner. For this purpose, a formal definition of these mappings and their transposition and composition operations is presented along with investigations on sparsity. Furthermore, two important variants of sparse matrix patterns required by the finite element and the discontinuous Galerkin methods are derived from the theory of set-valued adjacency relation mappings and expressed as compositions thereof.

A generic interface is presented in the second part of this work, along with suggested implementations utilising the `C++` standard library components of the required operations which can be easily integrated in existing finite element software packages. Worst-case runtime estimates are presented for these implementations.

Keywords: adjacency relations, finite elements, generic programming, unstructured sparse matrices

1 Introduction

The usual approach to describe an adjacency relation is by using a (directed) graph $G = (V, E)$ with a finite set of nodes V and a set of edges $E \subseteq V \times V$. This approach has two major advantages: First, it is very generic in the sense that any type of adjacency relation on V can be expressed this way and, second, the use of a basic and common mathematical structure such as a graph offers a wide variety of theorems and algorithms – in this case from the field of graph theory.

One of the most famous graph algorithms used in the field of numerics is the *Cuthill-McKee* algorithm (see [5]), which computes a permutation of the graph's node set with the goal of minimising the bandwidth of a sparse matrix represented by that graph. Although the authors formally define their renumbering scheme on the data structure of a graph, i.e. on sets of nodes and edges, an efficient implementation of the algorithm requires a different data structure, as one needs to be able of iterating over all nodes adjacent to a specific node via some edge efficiently. This, among other reasons, has led to the fact that one rarely describes adjacency relations by sets of edges – as in theory – in implementations of graph algorithms, but instead prefers a data structure which allows efficient access to all neighbours of a particular node. One commonly used data structure that fulfills these requirements is the *row-pointer* and *column-index* array pair used by the *Compressed Sparse Row* storage format for unstructured sparse matrices (see [2]).

Another aspect, which suggests the usage of such a data structure, is that one often has to consider adjacency relations between two disjunct node sets X and Y. A few examples, which appear in the context of finite element software, are adjacency relations between:

- vertices and elements of a mesh, specifying the corner vertices of an element.

[*]Institut für Angewandte Mathematik und Numerik, Technische Universität Dortmund, Vogelpothsweg 87, D-44227 Dortmund, Germany

- elements of two meshes in a refinement hierarchy, specifying which elements in a refined mesh arise from the subdivision of an element in a coarse mesh.
- finite element basis functions and the elements of a mesh, specifying which elements intersect with the support of a basis function.
- boundary components and vertices of a mesh, specifying which vertices belong to a particular boundary component.

In this work, we first want to formalise the data structure that has been used in graph algorithm implementations for a long time and investigate how the concept of *sparse* relations can be defined for this approach. Furthermore, we show how this approach can be utilised to formally express finite element matrix stencils and, finally, we show how the theoretical concept can be realised in a generic `C++` implementation, including estimates for the worst-case runtime.

2 Mathematical Background

We begin our formal section with a definition that allows us to express adjacency relations between finite sets as a set-valued mappings, which will form the basic data structure used throughout this work.

Definition 1: Adjacency Relation Mapping
Let X, Y denote two finite sets, then a mapping $\mathcal{A} : X \to \wp(Y)$ is called an adjacency relation mapping between X and Y, where $\wp(Y)$ denotes the power set of Y. Furthermore, we denote the cardinality of $\mathcal{A}$ by

$$|\mathcal{A}| := \sum_{x\in X} |\mathcal{A}(x)|.$$

We now want to see that for any pair of finite sets X and Y we can express any graph $G = (X, Y, E)$ characterised by an edge set $E \subseteq X \times Y$ as an adjacency relation mapping and vice versa by the correlation $y \in \mathcal{A}(x) \iff (x, y) \in E$. One can say that the adjacency relation mapping is not a fundamentally new mathematical structure, but rather a different way of interpreting a graph. Furthermore, with this equivalence it is easy to see that the cardinality of $\mathcal{A}$ is equal to the cardinality of the edge set E of the graph G that is equivalent to $\mathcal{A}$.

Many definitions and operations on graphs can be translated into versions for adjacency relation mappings, but we will restrict us to two operations, namely the transposition and the composition, as they will suffice for our cause.

Definition 2: Transposition
Let $\mathcal{A} : X \to \wp(Y)$, then the transpose $\mathcal{A}^\top : Y \to \wp(X)$ is defined as

$$\mathcal{A}^\top(y) := \left\{\ x \in X \ \middle|\ y \in \mathcal{A}(x)\ \right\}. \tag{1}$$

Definition 3: Composition
Let $\mathcal{A} : X \to \wp(Y)$ and $\mathcal{B} : Y \to \wp(Z)$, then the composition $\mathcal{B} \circ \mathcal{A} =: \mathcal{C} : X \to \wp(Z)$ is defined as

$$\mathcal{C}(x) := \left\{\ z \in Z \ \middle|\ \exists y \in \mathcal{A}(x) : z \in \mathcal{B}(y)\ \right\} = \bigcup_{y\in\mathcal{A}(x)} \mathcal{B}(y). \tag{2}$$

Finally, we want to propose a lemma that describes the interplay of the transposition and composition operations.

Lemma 4
Let $\mathcal{A} : X \to \wp(Y)$ and $\mathcal{B} : Y \to \wp(Z)$, then it holds

$$\big(\mathcal{B} \circ \mathcal{A}\big)^\top = \mathcal{A}^\top \circ \mathcal{B}^\top.$$

We omit the proof of this lemma, as it is just an application of the two previous definitions.

2.1 Sparsity

With the previous definitions, we have established a basic formal description of adjacency relations between finite sets. In the following, we want to restrict our further investigations to *sparse* relations and for this purpose, we first need to define this term. The following definition will introduce two variants of sparse relations, that will be used throughout the rest of this work. For that purpose, we will consider families of adjacency relations, as it would be hard to speak of 'a constant independent of n' otherwise.

Definition 5: Weak/Strong Sparsity
For $n \in \mathbb{N}$ let $\mathcal{A}_n : X_n \to \wp(Y_n)$ denote a family of adjacency relations.

1. *We say that $\mathcal{A}_n$ is weakly sparse if there exists a constant $c_{\mathcal{A}} \geq 1$ independent of n, such that for all n*
$$|\mathcal{A}_n| \leq c_{\mathcal{A}} \cdot \big(|X_n| + |Y_n|\big).$$
2. *We say that $\mathcal{A}_n$ is strongly sparse if there exists a constant $c_{\mathcal{A}} \geq 1$ independent of n, such that for all n and all $x \in X_n$ it holds*
$$|\mathcal{A}_n(x)| \leq c_{\mathcal{A}}.$$

Obviously, a strongly sparse family $\mathcal{A}_n$ is also weakly sparse, and one can easily construct counter-examples showing that weakly sparse families are not necessarily strongly sparse.

In the following, we want to investigate how the transposition and composition operations affect these two definitions of sparsity. By observing that for any $\mathcal{A} : X \to \wp(Y)$ it holds that $|\mathcal{A}| = |\mathcal{A}^\top|$ one can immediately conclude that for any weakly sparse family $\mathcal{A}_n : X_n \to \wp(Y_n)$ also the family of the transpositions $\mathcal{A}_n^\top : Y_n \to \wp(X_n)$ is weakly sparse with the same constant c. Unfortunately, one can show that the transposed family of a strongly sparse family is not necessarily strongly sparse: Let $c \in \mathbb{N}$ denote an arbitrary constant, then for $n > c$ we define $X_n := \{1, ..., n\}$, $Y_n := \{1\}$ and set $\mathcal{A}_n(x) := Y_n$ for all $x \in X_n$, then $|\mathcal{A}(x)| = 1$ for all $x \in X$, but $|\mathcal{A}_n^\top(1)| = |X_n| = n > c$. One may argue about the usefulness of a sparsity definition, where a relation $\mathcal{A}$ is strongly sparse, but its transpose may not be. The reason for this 'non-symmetry' is simply that we do not want to restrict the definition more than we need to, as this would rule out practically relevant scenarios which we will consider later.

The effects of composing sparse relations are more complex and contain several cases, which we want to summarise in the following lemma.

Lemma 6: Composition Sparsity
For $n \in \mathbb{N}$ let X_n, Y_n and Z_n denote families of finite sets. Let $\mathcal{A}_n : X_n \to Y_n$ and $\mathcal{B}_n : Y_n \to Z_n$ denote two weakly sparse families and let $\mathcal{C}_n := \mathcal{B}_n \circ \mathcal{A}_n$ denote the family of the compositions of $\mathcal{A}_n$ and $\mathcal{B}_n$, then:

1. *If both $\mathcal{A}_n$ and $\mathcal{B}_n$ are strongly sparse, then $\mathcal{C}_n$ is strongly sparse.*
2. *If $\mathcal{B}_n$ is strongly sparse and at least one of the following conditions holds, then $\mathcal{C}_n$ is weakly sparse.*
   (a) $\exists c_{\mathcal{A}} \geq 1, \forall n : |\mathcal{A}_n| \leq c_{\mathcal{A}} \cdot |X_n|$
   (b) $\exists c_Y \geq 1, \forall n : |Y_n| \leq c_Y \cdot |Z_n|$
3. *If $\mathcal{A}_n^\top$ is strongly sparse and at least one of the following conditions holds, then $\mathcal{C}_n$ is weakly sparse.*
   (a) $\exists c_{\mathcal{B}} \geq 1, \forall n : |\mathcal{B}_n| \leq c_{\mathcal{B}} \cdot |Z_n|$
   (b) $\exists c_Y \geq 1, \forall n : |Y_n| \leq c_Y \cdot |X_n|$

Proof. Throughout this proof, let $n \in \mathbb{N}$ be arbitrary and let $c_\mathcal{A}$ and $c_\mathcal{B}$ denote the sparsity constants of $\mathcal{A}_n$ and $\mathcal{B}_n$, respectively, as given in Definition 5.
• case *1.*: Let $x \in X_n$ be arbitrary, then we have

$$\big|\mathcal{C}_n(x)\big| = \Big|\bigcup_{y\in\mathcal{A}_n(x)} \mathcal{B}_n(y)\Big| \le \sum_{y\in\mathcal{A}_n(x)} \big|\mathcal{B}_n(y)\big| \le \sum_{y\in\mathcal{A}_n(x)} c_\mathcal{B} \le c_\mathcal{A}\cdot c_\mathcal{B}.$$

• case *2.*: We have

$$\big|\mathcal{C}_n\big| = \sum_{x\in X_n} \big|\mathcal{C}_n(x)\big| = \sum_{x\in X_n} \Big|\bigcup_{y\in\mathcal{A}_n(x)} \mathcal{B}_n(y)\Big| \le \sum_{x\in X_n}\sum_{y\in\mathcal{A}_n(x)} \big|\mathcal{B}_n(y)\big| \le \sum_{x\in X_n}\sum_{y\in\mathcal{A}_n(x)} c_\mathcal{B} = c_\mathcal{B}\cdot\big|\mathcal{A}_n\big|.$$

If condition *(a)* holds, then we obtain

$$\big|\mathcal{C}_n\big| \le c_\mathcal{B}\cdot\big|\mathcal{A}_n\big| \le c_\mathcal{A}\cdot c_\mathcal{B}\cdot|X_n|.$$

Otherwise, if condition *(b)* holds, we obtain

$$\big|\mathcal{C}_n\big| \le c_\mathcal{B}\cdot\big|\mathcal{A}_n\big| \le c_\mathcal{A}\cdot c_\mathcal{B}\cdot\big(|X_n|+|Y_n|\big) \le c_\mathcal{A}\cdot c_\mathcal{B}\cdot c_Y\cdot\big(|X_n|+|Z_n|\big).$$

• case *3.*: This case follows directly from case *2.*, the fact that $(\mathcal{A}^\top)^\top = \mathcal{A}$ and Lemma 4. ■

As a final note, one can easily construct a counter-example which shows that the composition of two weakly but not strongly sparse families is not necessarily (weakly) sparse: for $n \in \mathbb{N}$ consider $X_n = Y_n = Z_n := \{1, ..., n\}$, let $\mathcal{A}_n(i) := \{1\}$ for all $1 \le i \le n$ and let $\mathcal{B}_n := \mathcal{A}_n^\top$, then the composition $\mathcal{C}_n := \mathcal{B}_n \circ \mathcal{A}_n$ is *dense*, i.e. $|\mathcal{C}_n| = n^2$.

3 Finite Element Matrix Sparsity Patterns

Now we want to apply the framework of the previous section on the adjacency relations appearing in the context of the finite element method, especially with the goal of describing an efficient method for the assembly of matrix sparsity patterns. For the rest of this section, we assume that all meshes are *conforming*, i.e. there shall be no hanging nodes.

Definition 7: Finite Element Dof-Mapping
Let V denote a finite element space defined on a mesh $T = \{T_k\}$, and let $B = \{\varphi_i\}$ denote a basis of V, then the dof-mapping $\mathcal{D} : T \to \wp(B)$ of V is defined as

$$\mathcal{D}(T_k) := \big\{\ \varphi_i \in B\ \big|\ \text{support}(\varphi_i)\cap T_k \neq \emptyset\ \big\}. \tag{3}$$

The dof-mapping is one of the most basic and most important adjacency relations for finite element methods. One major question is now what type of sparsity a dof-mapping relation (and its transpose) fulfills. The answer to that question obviously depends on the construction of the basis functions, but we want to assume that the dof-mapping relation is at least weakly sparse, as we could not expect the resulting matrix to be sparse otherwise. Furthermore, we see that a (family of) dof-mapping relation(s) is strongly sparse if and only if for any element at most a constant number of basis functions do not vanish on that element. Although there exist various approaches to construct finite element basis sets violating this condition (e.g. in the case of p-adaptivity), the vast majority of finite element basis function sets fulfill this condition, especially since one usually imposes upper bounds for those violating scenarios, e.g. by prescribing a maximum allowed degree in the case of p-adaptivity. However, considering the widely used finite elements such as the H^1-conforming P_n and Q_n spaces (see e.g. [3]), but also non-conforming approaches such as *Crouzeix-Raviart* (see [8]), *Rannacher-Turek* (see [9]) and various others (see e.g. [10]) fulfill the strong sparsity condition of the dof-mapping relation.

As for the transposed dof-mapping $\mathcal{D}^\top$, we observe that strong sparsity not only depends on the construction of the basis, but also on the underlying mesh family. Considering the example of the H^1-conforming piecewise linear element P_1, then for the standard Lagrangian basis the family

$\mathcal{D}_n^\top$ will only be sparse if for any mesh in the family the number of elements sharing a common vertex is bounded by a constant. Furthermore, one also may consider finite element spaces, where a single basis function does not vanish on the whole domain (e.g. for use as a Lagrangian multiplier in a problem with pure Neumann boundary conditions, see e.g. [7]) which also leads to violation of the strong sparsity of $\mathcal{D}_n^\top$.

In summary, we assume that a (family of) dof-mapping relation(s) is strongly sparse, however, we do not assume the same for its transpose, as it simply will not be necessary for our purpose. As a first example of matrix sparsity pattern assembly for the finite element method, we consider the *standard matrix stencil*, which describes the adjacency relation of test- and trial-space[1] basis-function pairs under a (discrete) weak partial differential operator $L : V \times W \to \mathbb{R}$ consisting only of integrals over the elements T_k of a mesh T:

$$L(\varphi,\psi) := \sum_{T_k \in T} \int_{T_k} \ell_k\big(\varphi,\psi\big)(x)\ dx \qquad \text{for some} \qquad \ell_k : V|_{T_k} \times W|_{T_k} \to \mathcal{L}^1(T_k).$$

For this kind of operator, we can directly observe that a pair of test and trial basis functions ψ_i and φ_j can only generate a non-zero entry a_{ij} in the matrix representing the discrete differential operator if their supports intersect. For the purpose of implementing the matrix stencil assembly, this condition is usually relaxed in the sense that a pair of test- and trial-functions generates a non-zero entry if there exists an element $T_k \in T$ which intersects with the supports of both the test and the trial basis functions. Combining this relaxed condition with our previous definition of the dof-mapping relation, we can now formulate a lemma which describes the standard matrix stencil.

Lemma 8: Standard Matrix Stencil
Let V and W be two finite element spaces defined on the same mesh T, where V denotes the test space and W denotes the trial space. Furthermore, let $\mathcal{D}_V$ and $\mathcal{D}_W$ denote the dof-mappings for some bases B_V and B_W of V and W, respectively, then the standard matrix stencil $\mathcal{A} : B_V \to \wp(B_W)$ is given by

$$\mathcal{A} = \mathcal{D}_W \circ \mathcal{D}_V^\top.$$

Proof. By definition, for any test basis function $\psi_i \in B_V$ the set of all elements $T_k \in T$ on which ψ_i does not vanish is given by $\mathcal{D}_V^\top(\varphi_j)$. Now, for each $T_k \in T$ the set of all trial basis functions φ_j, which do not vanish on T_k, is given by $\mathcal{D}_W(T_k)$. In consequence, $\mathcal{D}_W \circ \mathcal{D}_V^\top(\psi_i)$ is the set of all $\varphi_j \in B_W$, for which the supports of the test-trial basis function pair ψ_i, φ_j intersect with some common $T_k \in T$. ■

The previous lemma states that the standard matrix stencil is given by the composition of the trial-space dof-mapping applied onto the transpose of the test-space dof-mapping, so in consequence we can apply our results from Lemma 6. As we assume both $\mathcal{D}_W$ and $\mathcal{D}_V$ to be strongly sparse, we can at least apply case 2. or 3. of the lemma, since in our case condition (b) states that the number of elements shall be bounded linearly by the number of test- or trial-space basis functions, which is naturally fulfilled for the vast majority of finite element spaces. Furthermore, if we additionally assume that $\mathcal{D}_V^\top$ is strongly sparse, then by case 1. of Lemma 6 we get that $\mathcal{A}$ is also strongly sparse – which covers most of the practically relevant scenarios.

In addition to integrals over elements, various methods require integrals over the facets of the mesh – two notable examples are the *discontinuous Galerkin method* (see e.g. [4]) or the *edge-oriented jump-stabilisation method* (see [6]). Such methods require an *extended matrix stencil*, as a pair of test- and trial-functions generates non-zero entries if the closures of their supports intersect. In analogy to the standard matrix stencil, one usually also relaxes this relation in the sense that a pair of test- and trial-functions generates a non-zero entry if there exists a facet, such that the supports of the test and the trial basis function intersect with some element adjacent to that facet, respectively. For this purpose, we will first define a relation, which describes element neighbours, as well as a proposition showing how this relation can be assembled by the methods we have already described.

[1]The term *trial space* is a synonym for *ansatz space.*

Lemma 9: Facet Neighbours
Let $T = \{T_k\}$ denote a mesh and let $F = \{F_i\}$ denote the set of all facets of T. Furthermore, let $\mathcal{F} : T \to \wp(F)$ denote the relation

$$\mathcal{F}(T_k) := \left\{ \ F_i \in F \ \middle| \ F_i \text{ is a facet of } T_k \ \right\},$$

then the relation $\mathcal{N} : T \to \wp(T)$ describing adjacencies of elements over facets (including self-adjacencies), i.e.

$$\mathcal{N}(T_k) = \left\{ \ T_l \in T \ \middle| \ T_l \text{ shares a common facet with } T_k \ \right\},$$

is given by

$$\mathcal{N} = \mathcal{F}^\top \circ \mathcal{F}. \tag{4}$$

As long as one assumes that the mesh T is conformal, i.e. there exist no hanging nodes, then $\mathcal{N}$ is strongly sparse by construction, as each $T_k \in T$ has only a fixed number of facets, over which it may be adjacent to another $T_l \in T$. By using the relation of facet neighbours, we can now propose how the extended matrix stencil can be expressed by the means of a composition of relations, again using the relaxed formulation.

Lemma 10: Extended Matrix Stencil
Let T, V, W, B_V, B_W, $\mathcal{D}_V$, $\mathcal{D}_W$ as in Lemma 8. Furthermore, let $\mathcal{N} : T \to \wp(T)$ as in (4), then the extended matrix stencil $\widehat{\mathcal{A}} : B_V \to \wp(B_W)$ is given by

$$\widehat{\mathcal{A}} := \mathcal{D}_W \circ \mathcal{N} \circ \mathcal{D}_V^\top.$$

We omit the proof of the lemma, as it is similar to the one of the standard matrix stencil. Furthermore, since $\mathcal{N}$ is both strongly sparse and symmetric, i.e. $\mathcal{N} = \mathcal{N}^\top$, we obtain weakly or even strong sparsity of $\widehat{\mathcal{A}}$ under the same assumptions as for the standard matrix stencil.

By using the concepts of adjacency relations that we have learned so far, one might also construct and formally describe far more complex matrix stencils than the two (most common) variants we have handled in this section. One notable example may be the assembly of a matrix stencil for a combination of test- and trial-spaces defined on two different meshes discretising the same analytical domain – in this case, one only needs to exchange the facet-neighbour relation $\mathcal{N}$ of the previous lemma by an according adjacency relation describing intersecting element pairs of the two different meshes. However, we will not continue investigating this and other more complex scenarios in this work as this would exceed its scope.

4 Implementational Aspects

We now focus on the practical aspects of realising our concept of adjacency relations in the context of a finite element software implementation. A first practical observation is that entities coming into consideration for adjacency relations (e.g. vertices, basis functions or matrix rows) are usually enumerated consecutively in some context-dependent order, so that one may restrict to handling adjacency relations for sets of indices rather than dealing with the objects themselves. Therefore, we want to identify an element $x_i \in X = \{x_0, ..., x_{n-1}\}$ by its unique index `i` $\in \{0, ..., n-1\}$ and analogously $y_j \in Y = \{y_0, ..., y_{m-1}\}$ by `j` $\in \{0, ..., m-1\}$.

In this section we will define an abstract interface for adjacency relations as well as describe data structures and algorithms that realise the transposition and composition operations, as we have defined them formally in the previous section. The following descriptions are expressed by means of `C++` classes and templates, but the interfaces and algorithms may as well be translated into any other programming language capable of object-oriented and generic programming, of course.

4.1 Adjactor Interface

As a first step, we want to define a *duck-type* interface specifying the minimum requirements for a class that shall represent an adjacency relation mapping $\mathcal{A}$ between (the indices of) two finite sets

X and Y, so that it can be used by algorithms assembling adjacency related information, such as matrix stencils or mesh neighbourhoods.

A class implements the *adjactor* interface, if it

1. has two member functions returning the cardinalities of the domain and image sets X and Y, respectively:
   `std::size_t size_domain() const;`
   `std::size_t size_image() const;`
2. contains a public nested class (or typedef) `image_iterator` which has at least
   - a standard constructor:
     `image_iterator();`
   - a copy-constructor:
     `image_iterator(const image_iterator& other);`
   - a pre-increment operator:
     `image_iterator& operator++();`
   - an inequality-comparison operator:
     `bool operator!=(const image_iterator& other) const;`
   - a dereferentiation operator that returns[2] the index `j` of the current $y_j \in Y$:
     `std::size_t operator*()`
3. has a member function returning an `image_iterator` representing the first $y_j \in \mathcal{A}(x_i)$ for any $x_i \in X$:
   `image_iterator begin_image(std::size_t i) const;`
4. has a member function returning an `image_iterator` representing the first position past the last $y_j \in \mathcal{A}(x_i)$ for any $x_i \in X$:
   `image_iterator end_image(std::size_t i) const;`

For the complexity analysis that we will carry out in the following sections, we assume that all functions and operators of a class implementing the above interface have a complexity of $\mathcal{O}(1)$. Furthermore, each $y_j \in \mathcal{A}(x_i)$ shall be stored exactly once – or more precisely: for any index `i` of $x_i \in X$ and any index `j` of $y_j \in \mathcal{A}(x_i)$, there shall $\exists!\ k \in \{1, ..., |\mathcal{A}(x_i)|\}$, such that `j` is returned by the dereferentiation operator of the `image_iterator` given by applying the pre-increment operator $(k-1)$ times onto the return value of `begin_image(i)` – this condition shall be referred to as *injectivity* from now on. However, we do not assume that the indices `j` of $y_j \in \mathcal{A}(x_i)$ are sorted in any order.

This interface definition is designed to be as lean as possible to allow existing implementations of classes managing adjacency relations to be easily extended, so that they may be used for our purposes without the necessity of reimplementing existing code. Moreover, any *input iterator*, as defined by the `C++` standard (see [1, 24.2.3]), fulfills the conditions of the `image_iterator` interface, thus allowing a wide choice of potential class templates for use as internal data containers.

As we have seen in the theoretical background before, we will need to compute transpositions and compositions of objects whose underlying class(es) implement(s) the adjactor interface. The first approach – in analogy to the concept of *container adaptors* from the `C++` standard library – would be writing two class templates, which realise the transposition or composition of an arbitrary (pair of) adjactor object(s). At least for the transposition operation, this will in general not be possible, as it would require iterating over all $x_i \in X$ such that $\mathcal{A}(x_i) \ni y_j$ for any $y_j \in Y$ and some $\mathcal{A} : X \rightarrow \wp(Y)$, which is a functionality that our interface is not designed to offer in an efficient way. The composition of two (or more) adjactor objects, however, could be implemented as a class template in such a manner and, although the implementation of the `image_iterator` class is a bit more complex in this case, this approach offers the possibility of describing compositions of arbitrary length by recursively nesting such composite adjactors.

[2]The dereferentiation operator might as well return a (const) reference to an object of type `std::size_t` or any other type, as long as the result is implicitly convertible to `std::size_t`.

One drawback is that this way of realising compositions will inevitably violate the injectivity condition, thus leading to potentially significant runtime overhead when iterating over the $y_j \in \mathcal{A}(x_i)$, especially if the (nested) composition consists of more than two adjactors.

Another approach, which we will describe in the following, is implementing a 'reference' adjactor class that conforms to the above interface and which is capable of computing and storing the (possibly temporary) transpositions and compositions of other adjactor objects. In total, such a reference adjactor class must provide at least three operations, preferably realised as templatised constructors:

1. The class has to be capable of converting any object whose underlying class implements the adjactor interface into its own representation. This operation is called *rendering (the conversion of)* an adjactor.
2. The class has to be capable of computing the *transposition* of any object whose underlying class implements the adjactor interface and storing the result in its own representation. This operation is called *rendering the transposition* of an adjactor.
3. The class has to be capable of computing the *composition* of any two objects (with compatible cardinalities, of course) whose (possibly different) underlying classes implement the adjactor interface and storing the result in its own representation. This operation is called *rendering the composition* of two adjactors.

4.2 Dynamic Reference Adjactor

The first approach for the implementation of a reference adjactor class uses an `std::vector<std::set<std::size_t>>` as an internal data container, which – considering our required operations – is the natural choice for a `C++` implementation utilising the language's standard library components. This class is called the *dynamic reference adjactor*, where the word *dynamic* derives from the capability of easily and efficiently adding and/or removing adjacencies (in graph terms: edges), as long as the two sets X and Y remain unchanged.

The dynamic reference adjactor class implements the interface described in section 4.1 as follows:

- The cardinality of the image set Y, returned by the `size_image()` function, is stored separately in an attribute of type `std::size_t`, as it cannot be deduced from the internal data container itself.
- The cardinality of the domain set X, returned by the `size_domain()` function, is given by the length of the outer `std::vector` container.
- The `image_iterator` is a typedef for the `const_iterator` of the inner `std::set` container.
- The `begin_image(i)` and `end_image(i)` functions call the `begin()` and `end()` functions of the inner `std::set` container stored at position `i` of the outer `std::vector` container.

Each of the three rendering operations is easy to implement as a function template with less than five lines of code for the dynamic reference adjactor class, as all rendering operations only require iterating over all adjacencies of the (two) input adjactor(s) and successively inserting the corresponding adjacency into the data container, which is a trivial task thanks to the `insert()` function of the `std::set` class template.

The first drawback of this implementation is its worst-case runtime: assume we have $\mathcal{A} : X \to \wp(Y)$ and assume there exists an $x_i \in X$ such that $|\mathcal{A}(x_i)| = |Y| =: n$, i.e. x_i is adjacent to all $y \in Y$, then for rendering $\mathcal{A}(x_i)$ for this particular x_i, we obtain a worst-case runtime of

$$\sum_{k=1}^{n} \log(k) = \log(n!) = \Theta\big(n \log(n)\big),$$

since the complexity of the `insert()` function of the inner `std::set` container is logarithmic in the size of that set (see [1, Table 102]). However, if we assume that $\mathcal{A}$ is strongly sparse, then we obtain

$$\sum_{x\in X} \log\left(|\mathcal{A}(x)|!\right) \leq |X| \cdot \max_{x\in X} \log\left(|\mathcal{A}(x)|!\right) \leq |X| \cdot \log(c_{\mathcal{A}}!),$$

which leads to a complexity of $\mathcal{O}(|X|)$ for the operation of rendering the conversion. By the same argument, one can also show that rendering the transposition $\mathcal{A}^\top$ has linear complexity if $\mathcal{A}^\top$ is strongly sparse. Furthermore, rendering the composition $\mathcal{B} \circ \mathcal{A}$ also leads to linear complexity, if both $\mathcal{A}$ and $\mathcal{B}$ are strongly sparse, as can be seen in Lemma 6.

The second drawback is that in general one has no control over the internal data management of the inner `std::set` containers. To be more precise, one usually cannot bound the amount of memory that a single container reserves, which – considering the amount of sets required – may lead to *out-of-memory* situations although the actual amount of memory required to store the total adjacency information would still be within the machine's resources.

As a final summary, one may say that the dynamic reference adjactor provides an approach that is easy to implement and which suffices under the assumptions that a) all input adjactors are strongly sparse and b) the memory required to store the resulting dynamic reference adjactor is significantly lower than the total amount of memory available.

4.3 Static Reference Adjactor

The second approach is the *static reference adjactor* class using two `std::vector<std::size_t>` objects to manage adjacency information. The design of this class is closely related to the CSR matrix storage format (see e.g. [2]).

The static reference adjactor class consists of three attributes:

- A value of type `std::size_t` storing the cardinality of the image set Y, returned by the `size_image()` function.
- An object of type `std::vector<std::size_t>`, named `x_ptr`, whose size equals one plus the cardinality of the domain set X which is returned by the `size_domain()` function.
- A second object of type `std::vector<std::size_t>`, named `y_idx`, whose size equals the cardinality of $\mathcal{A}$.

The interplay of the `x_ptr` and `y_idx` vectors coincides with that of the *row-pointer* and *column-index* arrays of the CSR matrix storage format, i.e. for the index `i` of any node $x_i \in X$, the indices `j` of all $y_j \in \mathcal{A}(x_i)$ are given by the entries `y_idx[k]` for `x_ptr[i]` $\leq$ `k` $<$ `x_ptr[i+1]`. Please note that – in contrast to many CSR conventions – we do not assume that the indices `j` of $y_j \in \mathcal{A}(x_i)$ for some x_i are sorted in any way. However, we enforce that the subvector of `y_idx` storing the indices of $y_j \in \mathcal{A}(x_i)$ for some x_i does not contain duplicate values.

In the following, we want to describe how the three render operations can be implemented efficiently.

4.3.1 Conversion Rendering

For the operation of rendering an $\mathcal{A} : X \to \wp(Y)$ represented by an object `A`, whose underlying class implements the adjactor interface, we proceed as follows:

1. Allocate the `x_ptr` array of length `A.size_domain()+1` and set `x_ptr[0] = 0`.
2. Successively loop over all indices `i` of $x_i \in X$ in increasing order, compute $|\mathcal{A}(x_i)|$ by iterating from `A.begin_image(i)` to `A.end_image(i)` and set `x_ptr[i+1] = x_ptr[i] + card`, where `card` is a local variable storing $|\mathcal{A}(x_i)|$.
3. Allocate the `y_idx` array of length `x_ptr.back()`.
4. For each index `i` of $x_i \in X$, loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and store them successively in the sub-vector of `y_idx` beginning at position `x_ptr[i]`.

The worst-case runtime analysis for this render operation is straight forward: step 1. has a complexity of $\mathcal{O}(|X|)$, whereas steps 2., 3. and 4. have a complexity of $\mathcal{O}(|\mathcal{A}|)$, which is also the natural lower bound for the complexity of the total render operation.

4.3.2 Transposition Rendering

For the operation of rendering the transposition $\mathcal{A}^\top$ of $\mathcal{A} : X \to \wp(Y)$ represented by an object `A`, whose underlying class implements the adjactor interface, we proceed as follows:

1. Allocate a temporary `std::vector<std::size_t>` named `temp` of length `A.size_image()` and initialise its entries to zero.

2. For each index `i` of $x_i \in X$, loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and increment `temp[j]` by one.

3. Allocate the `x_ptr` array and compute its entries successively by `x_ptr[i+1] = x_ptr[i] + temp[i]` starting with `x_ptr[0] = 0`.

4. Copy `x_ptr` to `temp` (ignoring the last entry).

5. Allocate the `y_idx` array of length `x_ptr.back()`.

6. For each index `i` of $x_i \in X$, loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and for each `j` set `y_idx[temp[j]] = i` and increment `temp[j]` by one.

Also for this rendering operation, we obtain a total complexity of $\mathcal{O}(|\mathcal{A}|)$, which originates from the complexity of steps 2., 5. and 6., whereas steps 1., 3. and 4. yield a complexity of $\mathcal{O}(|Y|)$.

4.3.3 Composition Rendering

Before we describe the actual algorithm for this rendering operation, we want to discuss an issue that appears with adjactor compositions. For the description of the adjactor interface, we have assumed that any index `j` of $y_j \in \mathcal{A}(x_i)$ shall be stored exactly once within the family of `image_iterator`'s describing $\mathcal{A}(x_i)$. However, for the composition $\mathcal{B} \circ \mathcal{A}$, this will in general not hold, as some $z_k \in Z$ may be adjacent to a single $x_i \in X$ via two (or more) different $y_{j_1}, y_{j_2} \in Y$, i.e. $y_{j_1}, y_{j_2} \in \mathcal{A}(x_i)$ and $z_k \in \mathcal{B}(y_{j_1}) \cap \mathcal{B}(y_{j_2})$. To avoid rendering duplicate indices, this scenario has to be considered in the implementation, which in the following approach is realised using a `mask` vector keeping track of the $z_k \in Z$ that have already been considered as being adjacent to some $x_i \in X$.

For the operation of rendering the composition $\mathcal{B} \circ \mathcal{A}$ of $\mathcal{A} : X \to \wp(Y)$ and $\mathcal{B} : Y \to \wp(Z)$ represented by two objects `A` and `B`, respectively, whose (possibly different) underlying classes implement the adjactor interface, we proceed as follows:

1. Allocate the `x_ptr` array of length `A.size_domain()+1` and set `x_ptr[0] = 0`.

2. Allocate a temporary `std::vector<bool>` named `mask` of length `B.size_image()` and initialise its entries to `false`.

3. Successively loop over all indices `i` of $x_i \in X$ in increasing order and:

   (a) initialise a local variable of type `std::size_t` named `card` to zero.

   (b) loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and loop over all adjacent indices `k` of $z_k \in \mathcal{B}(y_j)$, and:
   If `mask[k]` is `false`, then increment `card` by one and set `mask[k]` to `true`, otherwise skip this index `k` (as it has already been considered).

   (c) loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and loop over all adjacent indices `k` of $z_k \in \mathcal{B}(y_j)$ and set `mask[k]` to `false`.

   (d) set `x_ptr[i+1] = x_ptr[i] + card`.

4. Allocate the `y_idx` array of length `x_ptr.back()`.

5. For each index `i` of $x_i \in X$ and:

   (a) initialise a local variable of type `std::size_t` named `cur` to `x_ptr[i]`

(b) loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and loop over all adjacent indices `k` of $z_k \in \mathcal{B}(y_j)$, and:
If `mask[k]` is `false`, set `y_idx[cur] = k`, increment `cur` by one and set `mask[k]` to `true`, otherwise skip this index `k` (as it has already been considered).

(c) loop over all adjacent indices `j` of $y_j \in \mathcal{A}(x_i)$ and loop over all adjacent indices `k` of $z_k \in \mathcal{B}(y_j)$ and set `mask[k]` to `false`.

For convenience we define $\mathcal{C} := \mathcal{B} \circ \mathcal{A}$, and we see that steps 1. ans 2. yield a complexity of $\mathcal{O}(|X| + |Z|)$. Furthermore, step 4. gives us $\mathcal{O}(|\mathcal{C}|)$, which is again a natural lower bound for the overall complexity, and if we assume that $\mathcal{C}$ is (at least) weakly sparse, we have $|\mathcal{C}| \in \mathcal{O}\big(|X| + |Z|\big)$ by definition.

The analysis of steps 3. and 5. is a bit more complex: for instance, in step 3.(b), the variable `card` is incremented exactly $|\mathcal{C}(x_i)|$ times – thus again leading to a lower bound of $\mathcal{O}(|\mathcal{C}|)$ for step 3. – but in general one cannot provide an upper bound for the number of accesses on the `mask` array without additional assumptions on the sparsity of $\mathcal{A}$ and $\mathcal{B}$. If we assume that $\mathcal{A}$ and $\mathcal{B}$ fulfill one of the three scenarios suggested by Lemma 6, then the proof of the lemma shows that the three nested loops (over the indices `i`, `j`, `k`) cannot perform more than $c_{\mathcal{C}} \cdot \big(|X| + |Z|\big)$ iterations in total, where $c_{\mathcal{C}}$ is the sparsity constant as estimated in the proof – the same argument also holds for the three other nested loops in steps 3. and 5.

Summarising all the partial estimates, we obtain a complexity of $\mathcal{O}(|X| + |Z|)$ for this rendering operation, assuming that $\mathcal{A}$ and $\mathcal{B}$ fulfill the requirements of Lemma 6.

5 Conclusions

We have proposed a formal description of adjacency relation mappings, which suggest an alternative to the usual approach of describing adjacency relations by graphs. These provide, in a directly implementable manner, a means to express complex adjacency relations as compositions and transpositions of more fundamental relations which appear in the context of finite element software packages. We demonstrated this principle for the standard and extended finite element matrix sparsity patterns. Although these are only two examples, they show the advantages of adjacency relation mappings over graphs, as matrix sparsity patterns cannot be expressed as simple operations on the latter.

Finally, we presented a generic interface which can be subsequently included in `C++` finite element software packages with minimal changes to existing code due to its weak assumptions on the underlying data containers. Furthermore, the two reference adjactor classes, utilising `C++` standard library containers, provide efficient implementations of the necessary transposition and composition operations. These allow rendering various complex adjacency relations required in the context of the finite element method, most notably the assembly of matrix sparsity patterns.

Acknowledgements

This work was supported (in part) by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing" (SPPEXA).

References

[1] *Information technology – Programming languages – C++*; 3rd edition, ISO/IEC 14882:2011(E)

[2] Y. Saad: *Iterative Methods for Sparse Linear Systems*; 2nd edition, SIAM, 2003, ISBN 978-0-89871-534-7

[3] P.G. Ciarlet: *The Finite Element Method for Elliptic Problems*; 2nd edition, SIAM, 2002, ISBN: 978-0-89871-514-9

[4] J. S. Hesthaven, T. Warburton: *Nodal Discontinuous Galerkin Methods*; Springer, 2008, ISBN 978-0-387-72067-8

[5] E. Cuthill, J. McKee: *Reducing the Bandwidth of Sparse Symmetric Matrices*; Proceedings 24th National Conference ACM 1969, pp. 157–172

[6] A. Ouazzi, S. Turek: *Unified edge-oriented stabilization of nonconforming FEM for incompressible flow problems: Numerical investigations*; J. Num. Math, Volume 15 (2007), No. 4, pp. 299–322

[7] P. Bochev, R.B. Lehoucq: *On the Finite Element Solution of the Pure Neumann Problem*; SIAM, Vol. 47 (2001), No. 1, pp. 50–66

[8] M. Crouzeix, P.-A. Raviart: *Conforming and Nonconforming Finite Element Methods for solving the stationary Stokes equations I*; RAIRO, Volume 3 (1973), pp. 33–76

[9] R. Rannacher, S. Turek: *Simple Nonconforming Quadrilateral Stokes Element*; Numerical Methods for Partial Differential Equations, Volume 8 (1992), pp. 97–111

[10] J.-P. Hennart, J. Jaffré, J.E. Roberts: *A Constructive Method for Deriving Finite Elements of Nodal Type*; Numerische Mathematik, Volume 53 (1988), pp. 701–738