Sparse Matrix: Design choices and Documentation

Coronica Giovanni [IN2000215], Thomas Rossi Mel [IN2000202]

Problem Introduction and Requirements

In this project, we were tasked with implementing a sparse matrix data structure in C++ that efficiently stores and manipulates matrices with a significant number of zero elements. A sparse matrix is a matrix in which most of the elements are zero. To meet the project requirements, we had to implement two popular storage schemes for sparse matrices: the Coordinate (COO) format and the Compressed Sparse Row (CSR) format. The key requirements for this project are as follows:

1. Create a C++ code that efficiently stores sparse matrices in either COO or CSR format.
2. Implement a base class `SparseMatrix` with common functionalities and operations that can be derived for both storage schemes.
3. Derive classes `COOMatrix` and `CSRMatrix` from the base class to represent matrices in COO and CSR formats, respectively.
4. Implement the following operations:
    - Get the number of rows and columns.
    - Get the number of non-zero elements.
    - Read and write an entry of the matrix.
    - Compute the matrix-vector product.
    - Print the matrix to the standard output.
5. Implement utility functions to convert a matrix from COO format to CSR and vice versa.
6. Ensure const-correctness in the interface.

Approach and Design Choices

File Organization

We organized our code into separate files for better maintainability:

- `SparseMatrix.h` contains the base class `SparseMatrix`.
- `COOMatrix.h` and `CSRMatrix.h` define the derived classes `COOMatrix` and `CSRMatrix`.

- Implementation for COO and CSR classes is provided in `COOMatrix.tpp` and `CSRMatrix.tpp` files.
- `main.cpp` contains test cases to validate the correctness of the implementation.

Proxy pattern

One of the primary challenges encountered when implementing read and write operations for the COO and CSR classes in C++ was the limitation of having two distinct implementations for getter and setter operations using the same operator().
This dual behavior is essential as we want to create a new element in the sparse matrix when the setter is called on a new position, while for the getter, we aim to return a zero element without the need to allocate any space in memory.

To address this issue, we adopted a solution involving the use of a Proxy class. In this context, a Proxy class serves as a wrapper for individual elements of the matrix, enabling us to overload the operator= and operator T() methods and have two distinct behaviors for the setter and getter.
Within the SparseMatrix class, we leveraged the operator() overloading and configured it to return instances of the Proxy class.

Method placement

In our design, we made a conscious choice regarding where to implement methods. We placed the method declarations in the class itself (in the header file) when the method definitions are simple and do not require extensive logic or external dependencies (see for example getNonZeros).
For methods with more complex logic or when they are specific to a derived class, we implemented them in separate files (e.g., COOMatrix.tpp or CSRMatrix.tpp). This separation of implementation allows for better code organization, maintainability, and readability.

Conversion Between SparseMatrix Formats

To implement the conversion between different sparse matrix formats, we introduced a method called `copyFrom` in the base abstract class `SparseMatrix`. This method takes a generic `SparseMatrix` object as a parameter and copies its elements element by element into the current `SparseMatrix` object. By introducing this method in the base class, we created a common interface for converting between different derived classes of `SparseMatrix`.

To make this approach work, we had to make the `copyFrom` method protected because it is meant to be called internally when constructing an object of the derived class. We couldn't call pure virtual methods from the base class constructor, so this approach allows us to generalize the conversion process while keeping it within the boundaries of good design principles.

For example, in a derived class like `COOMatrix`, we have a constructor that calls the `copyFrom` method of the base class, seamlessly converting a CSR Matrix to a Coordinate Matrix.

Additional note: Choice of const and reference for variable of type T

In our design, we decided not to use `const &` for input parameters of methods that involve the template type `T`. This decision is based on the understanding that `T` can represent simple primitive types like `int`, `double`, and others. These types are typically small and do not benefit significantly from using a const reference. Therefore, for our design, passing `T` by value is sufficient and makes the code more straightforward.

How to Use the Implementation

1. Create Sparse Matrices

Create instances of the desired matrix format, specifying the number of rows and columns:

```
COOMatrix<double> cooMatrix(3, 3);
CSRMatrix<int> csrMatrix(4, 4);
```

2. Read and Write Matrix Entries

You can use the overloaded `()` operator to read and write matrix entries. For example:

```
cooMatrix(1, 2) = 5.7; // Set the element at row 1, column 2 to 5.7
double value = cooMatrix(1, 2); // Read the element at row 1, column 2
```

3. Matrix-Vector Multiplication

You can perform matrix-vector multiplication using the `operator*`:

```
std::vector<double> vec = {1.0, 2.0, 3.0};
std::vector<double> result = cooMatrix * vec;
```

4. Printing the Matrix

To print the matrix to the standard output, you can use the `operator<<`:

```
std::cout << cooMatrix;
```

5. Additional Methods

A method `equals` has been implemented, to easily check if two matrices have the same elements, without needing to know their implementation.

How to Compile

To compile your code using the provided compilation flags, use a command like:

```
g++ -std=c++17 -Wall -Wpedantic Main.cpp -o sparse_matrix
```

Division of Work

In our team of two, we divided the work as follows:

1. **Implementation of SparseMatrix Base Class:** Both team members collaborated to design and implement the `SparseMatrix` base class, including the common methods and the Proxy Element.
2. **Implementation of COO Format:** One team member focused on implementing the `COOMatrix` class, including methods for COO-specific functionality, while the other team member reviewed and tested the implementation.
3. **Implementation of CSR Format:** The team members switched roles, with one implementing the `CSRMatrix` class and the other reviewing and testing the CSR-specific functionality.
4. **Test Cases:** Both team members collaborated to write test cases in the `main.cpp` file to validate the correctness of the entire implementation, including matrix-vector multiplication and printing.