Relazione progetto C++
Febbraio 2020

Nome: Riccardo
Cognome: Confalonieri

Introduzione

Dopo aver valutato il problema proposto, ho deciso di rappresentare un singolo valore della matrice sparsa come un elemento, il quale contiene una tripletta di dati: le coordinate di riga e colonna nella matrice e un valore di tipo generico T.
Dato che il problema richiedeva una particolare attenzione nell'utilizzo della memoria, ho deciso di rappresentare l'intera matrice come una lista semplice di nodi dove ogni nodo contiene una coppia di dati: un elemento (appena descritto), e il puntatore al nodo successivo.

Tipi di dati

Ho deciso di definire due nuovi tipi di dati tramite typedef per rendere più leggibile il codice e permettere all'utente di utilizzare questi nuovi tipi nel suo codice per evitare possibili errori di conversione e semplificare la scrittura del codice. Questi nuovi tipi sono:

- ***value_type*** è il tipo generico T scelto dall'utente in fase di inizializzazione della matrice sparsa.
- ***sm_size*** è un unsigned int. Utilizzato per le righe e colonne della matrice e per la size totale della stessa.

Gli altri tipi di dati sono: l'elemento che, come descritto nell'introduzione, contiene una tripletta di dati contenuti in una "struttura dati" (struct) chiamata *element* così rappresentata:

- ***Value*** di tipo value_type che rappresenta il valore che l'utente ha assegnato a questa cella della matrice sparsa, se l'utente non assegna esplicitamente un valore alla cella value_type avrà come valore il valore di default.
- ***i*** di tipo sm_size che rappresenta la coordinata della riga della matrice in cui "salvare" l'elemento
- ***j*** di tipo sm_size che rappresenta la coordinata della colonna della matrice in cui "salvare" l'elemento

Le coordinate i e j sono dichiarate costanti, una volta inizializzate, in quanto le specifiche del progetto richiedevano che tali coordinate non fossero modificabili. Questo permette, in un secondo momento, di modificare il valore di un determinato elemento ma non le sue coordinate nella matrice.
Questa struct è dichiarata nel file SparseMatrix.h in una sezione pubblica per permettere all'utente, nel caso in cui volesse, di istanziare degli element nel suo codice e/o di salvarsi gli element ritornarti durante la scansione della matrice tramite Iterator come descritto in seguito.
Tale struct non contiene il costruttore di default per coerenza con la classe SparseMatrix nella quale è contenuta e anche perché non avrebbe senso istanziare un elemento senza specificarne il suo valore.

L'altro tipo di dato, già citato nell'introduzione, è una lista semplice, realizzata anch'essa tramite una struct *node* cosi rappresentata:

- ***field*** di tipo element, che è l'elemento inserito nella matrice
- ***next*** di tipo puntatore node, il puntatore al nodo successivo

Ho scelto di utilizzare una lista semplice con il solo campo next in quanto il progetto richiede di implementare gli iteratori di tipo forward e quindi non è necessario poter scorrere all'indietro i nodi, e anche nel resto del progetto per come l'ho implementato non vi è alcuna necessità di un puntatore al nodo precedente.
Questa struttura mi serve per poter salvare tutti e soli gli elementi effettivamente inseriti nella matrice cosi da salvaguardare spazio e ottimizzare l'utilizzo della memoria come richiesto. Ho deciso di mantenere la lista ordinata con le coordinate i,j del campo field crescenti in modo da semplificare la richiesta di scorrimento incrementale per gli iteratori.

Implementazione

La classe SparseMatrix contiene quattro attributi:

- ***_head*** puntatore al primo nodo della lista
- ***_D*** di tipo value_type, che rappresenta il valore di default degli elementi della matrice
- ***_size*** di tipo sm_size, che rappresenta il numero di elementi effettivamente inseriti
- ***_nRows*** di tipo sm_size, che rappresenta il numero di righe della matrice
- ***_nCols*** di tipo sm_size, che rappresenta il numero di colonne della matrice

Come espressamente richiesto nella consegna non è possibile istanziare una matrice sparsa senza definirne il valore di default _D, per fare questo ho deciso di non implementare il costruttore di default, anche se metodo fondamentale, in quanto altrimenti sarebbe stato possibile istanziare una matrice senza definire _D.
Ho deciso inoltre di definire la matrice con un numero di righe e colonne fissato, ho optato per questa scelta per garantire più controlli all'utente durante l'inserimento degli elementi nella matrice. Non è infatti possibile inserire un elemento al di fuori dei bounds di riga e colonna fissati. In questo modo in ogni istante si ha sempre un controllo su quale sia la dimensione massima che può assumere la matrice. Il valore di questi bounds è definito, insieme al valore di default, dall'utente in fase di inizializzazione della matrice.
Una volta definiti i valori di _D, _nRows e _nCols essi risultano non modificabili all'utente che potrà accedervi solo in lettura attraverso i rispettivi metodi get messi a disposizione.

Gestione della memoria

Le specifiche del progetto richiedevano un utilizzo di memoria minimale per questo ho deciso di utilizzare una lista di nodi per rappresentare la matrice salvandone al suo interno soltanto gli elementi esplicitamente inseriti nella matrice dall'utente. Tutti gli altri elementi hanno come valore il valore di default di tipo T definito dall'utente in fase di inizializzazione della matrice sparsa. In questo modo sono riuscito a salvaguardare lo spazio, infatti finché l'utente non inserisce alcun elemento la lista risulterà vuota e tutte le celle della matrice avranno valore di default.
Ogni volta che viene inserito un nuovo elemento viene aumentata la _size in questo modo anche l'utente può controllare quanti elementi ha effettivamente inserito.

Per l'inserimento degli elementi nella lista ho utilizzato questa logica:

- ***nuova cella***: in questo caso l'utente sta inserendo un valore in una nuova posizione della matrice. Viene quindi istanziato un nuovo nodo e inserito nella lista nella posizione corretta in base al valore delle coordinate i,j, il valore di _size viene aumentato
- ***cella esistente***: in questo viene scandita tutta la lista fino a trovare l'element con le coordinate inserite dall'utente e ne viene modificato il valore. In questo caso la _size non viene aumentata perché quel nodo esisteva già.

Durante l'inserimento se l'utente decide di inserire esplicitamente il valore di default questo viene trattato come un valore qualsiasi e quindi viene utilizzata la medesima logica di inserimento appena descritta.
Per un utilizzo di memoria ancor più minimale si poteva optare per non salvare l'element se esso aveva valore uguale a quello di default ma questa implementazione aveva delle incongruenze con quanto richiesto nel progetto infatti nelle specifiche è richiesto che gli iteratori scorrano tutti e soli gli elementi esplicitamente inseriti e quindi ho assunto che se l'utente vuole inserire il valore di default questo debba ritenersi come una scelta dell'utente e non uno spreco di memoria.
Inoltre vi sarebbe stata un ulteriore incongruenza se l'utente, attraverso l'utilizzo dell'iteratore, decidesse di modificare i valori assegnandogli il valore di default, in questo caso non potendo effettuare nessun controllo sulla modifica dei valori si sarebbe verificata un'incongruenza in quanto si avrebbero nella lista dei valori di default che in realtà avevamo deciso di non salvare.
Per queste motivazioni ho deciso di ritenere l'inserimento del valore di default nella lista come un inserimento valido anche se richiede di istanziare un nuovo nodo e quindi di occupare memoria.

Metodi implementati

Come già descritto ho deciso di non implementare il costruttore di default, mentre sono stati implementati tutti gli altri metodi fondamentali tra cui il costruttore con parametri che prende in input il valore di default e il valore del numero di righe e di colonne della matrice, in particolare questi ultimi due valori sono di tipo sm_size, cioè unsigned int, quindi nel caso in cui l'utente decida di effettuare una chiamata al costruttore con un valore negativo non otterrà un errore ma, coerentemente con la gestione di C++ degli unsigned int, otterrà una matrice in cui il numero di righe e colonne sarà uguale al valore di (max_unsigned_int – valore passato al costruttore).

Il costruttore secondario che prende in input una matrice definita su un tipo Q, e delegando al compilatore l'operazione di conversione, istanzia la matrice di tipo T con lo stesso numero di righe, colonne, e gli stessi elementi della matrice Q correttamente convertiti al nuovo tipo T.

Il metodo ***add()*** permette di inserire un nuovo elemento nella matrice, prende in input le coordinate e il valore dell'elemento. Il funzionamento logico di questo metodo è descritto nella sezione "Gestione della memoria". Questo metodo può fallire se le coordinate eccedano le dimensioni massime della matrice, in questo caso viene lanciata l'eccezione "index_out_of_bounds", oppure nel caso in cui non sia possibile allocare memoria per un nuovo nodo da aggiungere alla lista, in questo caso viene rilanciata all'esterno l'eccezione catturata e spetta all'utente gestirla. In entrambi i casi la matrice non viene svuotata e rimane com'era prima della chiamata al metodo add(). La gestione di questo metodo è rimandata ad un metodo add() privato che utilizzo come helper, a cui invece di passare coordinate e valore viene passato un elemento già istanziato con questi valori. Ho deciso di implementare questo metodo di helper per semplificare la chiamata alla add dai costruttori secondari e dal copy constructor, in questo modo infatti da questi metodi è possibile richiamare direttamente la add sugli elementi dei nodi che vengono visitati senza ulteriori processamenti.

Il metodo ***clear()*** è un metodo di appoggio privato che svuota tutta la matrice, setta gli attributi a 0 e il puntatore al primo nodo a nullptr. Viene chiamato quando viene invocato il distruttore della classe oppure dal copy constructor o dal costruttore templato nel caso in cui non sia possibile copiare tutta la matrice passata come parametro, restituendo così all'utente una matrice vuota ma coerente con i valori degli attributi.

Il metodo ***operator()*** che prende in input le coordinate i,j e restituisce l'indirizzo del valore dell'elemento in quella posizione. Nel caso in cui le coordinate eccedano le dimensioni massime della matrice, in questo caso viene lanciata l'eccezione "index_out_of_bounds". Tale metodo deve permette l'accesso in sola lettura, quindi sia il metodo sia il valore tornato sono costanti.
I metodi getDefaultValue(), getNumElement(), getNumCols(), getNumRows() permettono di ottenere il valore degli attributi della classe. Il valore ritornato è costante in quanto questi valori non sono modificabili.

La funzione globale ***evaluate()*** che prende in input una matrice di tipo M e un predicato P e restituisce il numero di valori della matrice, compresi quelli di default, che soddisfano P. Per effettuare il controllo scandisco la matrice utilizzando gli iteratori costanti dato che non sono necessarie modifiche.

Iteratori

La specifica di progetto richiedeva l'implementazione degli iteratori di tipo forward che ritornassero un'istanza della struct element in ordine in base alle coordinate degli elementi. Avendo gestito la matrice come una lista ordinata risulta semplice scorrere quella lista e ritornare lo struct element presente in ogni nodo della lista.
Come richiesto sono stati implementati sia gli iterator che i const_iterator.

Main

Nel file main.cpp vengono testati tutti i metodi pubblici resi disponibili dalla classe SparseMatrix. Vengono testate diverse modalità di utilizzo dei metodi e tramite assert viene verificata la correttezza dei risultati, non è quindi richiesto nessun input da parte dell'utente.
Vengono inoltre testati gli iteratori sia costanti che non, provando nel primo caso a modificare il valore contenuto nella struct element controllando che questa modifica sia effettivamente propagata con successo anche nella matrice su cui si sta iterando.
Ho provveduto anche a testare diverse inizializzazioni della classe passando come valore di default tipi standard come int, float ma anche strutture di appoggio come People e classi come car per verificare il corretto funzionamento anche con tipi custom.
Ho effettuato anche un controllo dei metodi costanti tramite il metodo test_constness a cui passo una sparse matrix di interi (per semplicità nei test) su cui invoco tutti i metodi definiti const nell'interfaccia pubblica della mia classe.