Relazione Progetto Programmazione C++

Sparse Matrix

Davide Cozzi
829827
d.cozzi@campus.unimib.it

La Struttura Dati

Ragionando sulla consegna del progetto ho concluso che il modo migliore per implementare una **sparse matrix** fosse quello di utilizzare una variante della *linked list*, dove gli elementi vengono caricati secondo l'ordine di due indici (che rappresentano la posizione di una cella in una matrice di implementazione classica). L'uso di questa variante della *linked list* permette di salvare in memoria solamente i valori indicati dall'utente (ognuno nella posizione desiderata).
In questa implementazione ogni node della **sparse matrix** contiene:

- un puntatore, privato, al node successivo. La scelta di definire questo attributo `private` è causata dal fatto che l'utente non deve essere conscio della struttura dati in uso
- una `struct element` contenente:
  - il valore che deve essere salvato
  - i due indici mediante i quali, logicamente, accedere al valore. I due indici sono definiti `const` in quanto non modificabili dopo la creazione della cella

Per questa *sottoclasse* `node` viene implementato un *costruttore* che genera un node a partire dal valore che si vuole aggiungere e dalla posizione in cui lo si vorrebbe aggiungere, senza specificare (e impostandolo di default a `nullptr`) il `node` successivo, e, ovviamente, viene implementato il *distruttore*, che si limita a settare a `nullptr` il nodo successivo (questo perché la

distruzione dell'intera struttura dati viene implementata nel *distruttore* di `sparse_matrix`).
Parliamo ora della struttura dati completa, chiamata `sparse_matrix`. Si hanno una serie di attributi privati necessari per il corretto funzionamento della stessa:

- il valore di default scelto obbligatoriamente dall'utente al momento della creazione di una matrice sparsa
- indicazioni riguardanti il numero di righe e colonne
- il nodo `head` che punta all'inizio della struttura dati
- il valore di elementi effettivamente contenuti nella struttura dati

Si hanno quindi diversi costruttori utili:

- si ha innanzitutto il costruttore base con solo l'indicazione di default, senza quindi indicazone delle dimensioni della matrice (che diventa, a livello teorico, una matrice quadrata di massima dimensione intera al quadrato)
- il costruttore di una matrice sparsa di dimensione teorica definita dall'utente
- costruttore copia a partire da un'altra `sparse_matrix` definita sullo stesso tipo
- costruttore copia a partire da un'altra `sparse_matrix` definita su un altro tipo (con controllo del cast delegato, secondo specifiche, al compilatore)

Si hanno poi, ovviamente, le implementazioni dei metodi *getter*.
Per quanto riguarda il distruttore si ha che esso chiama un metodo pubblico `clear()` che si appoggia ad un metodo privato `recursive_clear(head)`, che, partendo appunto dal primo elemento, setta a `nullptr` i nodi raggiungibili da `head`, settando, infine, anche `head` a `nullptr`. Il metodo `clear()` è pubblico in quanto permette all'utente di eliminare una `sparse_matrix` da lui creata, evitando quindi di incorrere in *memory leaks*.

Funzioni Principali

Ovviamente il primo discorso da affrontare è l'aggiunta di un determinato elemento in una certa posizione della nostra matrice sparsa, che viene gestito da un metodo `add(valore, posizione_x, posizione_y)`. Si hanno diverse possibili situazioni per l'inserimento di un nuovo valore:

- il caso "banale" è quello in cui ancora si ha una matrice sparsa senza alcun elemento, in tal caso si genera un nuovo `node` costruito sui dati in input e tale `node` diventa la nuova `head` della matrice sparsa
- si ha poi il caso in cui la `head` attuale è posizionata in una cella logicamente successiva a quella dell'elemento che si vuole inserire. Questa situazione si può verificare in 2 casi:
    1. l'`head` attuale si trova sulla stessa riga teorica dell'elemento che si vuole aggiungere ma ad una colonna successiva
    2. l'`head` attuale si trova ad una riga successiva
- se non si rientra nelle due casistiche precedenti si itera su tutta la matrice sparsa, fino all'avvenuta di una delle due possibili condizioni di arresto:
    1. si è arrivato all'ultimo elemento della matrice sparsa (che quindi punta a `nullptr` come nodo successivo). In tal caso, se gli indici del nodo che si vuole inserire coincidono con quelli dell'ultimo elemento se ne sovrascrive il valore, ricordandosi di cancellare il nodo che si voleva inserire per evitare *memory leaks*, altrimenti lo si inserisce come ultimo elemento (il vecchio nodo "tail" ora punterà a tale elemento che a sua volta punterà a `nullptr`)
    2. si sta valutando l'unico caso possibile restante, ovvero l'elemento che vogliamo inserire viene aggiunto in mezzo alla matrice sparsa, nella giusta posizione ordinata sui due indici. In tal caso, se gli indici del nodo che si vuole inserire coincidono con quelli dell'ultimo elemento se ne sovrascrive il valore, ricordandosi di cancellare il nodo che si voleva inserire per evitare *memory leaks*, altrimenti inserisco l'elemento che si vuole aggiungere

in mezzo alla struttura dati, nel punto corretto, sistemando i puntatori ai nodi successivi dell'elemento che si vuole inserire e del suo precedente

lo stesso metodo aggiorna anche il numero massimo di righe e colonne della matrice ipotetica, cercando il massimo indice di riga e di colonna.
L'eventuale inserimento di valori fuori indice viene gestito tramite un'eccezione `IndexOutOfBoundsException()`.
Sempre da specifica si ha poi l'implementazione del supporto agli iteratori di tipo forward in e scrittura. Questo viene implementato con le due classi `iterator` e `const_iterator` che vengono costruti a partire da `node`. Si ha quindi la definizione dei metodi `begin()`, che restituisce un `iterator` o un `const_iterator` costruiti a partire da `head`, e `end()`, che restituisce un `iterator` o un `const_iterator` costruiti a partire da `nullptr`, per segnalare la fine della matrice sparsa.
Un iteratore restituisce in lettura valore e posizione, in particolare `iterator` permette l'accesso in scrittura (per la modifica) del valore di una cella ma non degli indici della stessa.
Per l'accesso in lettura dei valori della matrice si è anche implementato l'overload di `operator()`, che permette l'accesso ad un determinato valore di una cella mediante i due indici. Dato che da specifica deve avvenire la restituzione del valore di default, nel caso in cui si chieda il valore di una cella non indicizzata nella nostra matrice sparsa, si procede iterando nella matrice sparsa come nel caso della `add` fino ad ottenere una condizione d'arresto e, se si incontra il `node` con indici uguali a quelli richiesti in uscita dal ciclo se ne restituisce il valore, altrimenti si restituisce il valore di default.
Infine viene implementata una funzione gloable `evaluate(sparse_matrix, predicato)` che conta il numero di elementi nella matrice che soddifano il predicato (considerando anche i valori di default). Si itera quindi sulla matrice sparsa contando tale numero. Infine si verifica se il valore di default lo soddisfa e, in caso positivo, si conta il numero di celle con valore di default (ovvero numero totale di celle meno il numero elementi effettivamente allocati, numero che è salvato nell'attributo `size`) e lo si somma al conteggio sopra effettuato, restituendo il risultato.

Test

La struttura dati viene testata con il tipo primitivo `float` e ne viene testato il *cast* a `int`. Viene infine testata la matrice su un tipo custom `point`, che viene

rappresentato da due coordinate. Nei vari test si verifica anche la corretta stampa mediante iteratori