Relazione Progetto C++
Grafo

Federico Luzzi
matricola: 816753
email: f.luzzi1@campus.unimib.it

Università degli Studi Milano Bicocca
A.A. 2018/19

Indice

Struttura Dati

Per implementare i nodi e la matrice di adiacenza del grafo utilizzando strutture dati non dinamiche (come richiesto) ho scelto di utilizzare:

- array di tipo templato per i nodi
- matrice di tipo bool per la matrice di adiacenza

In questo modo per l'aggiunta e rimozione di un nodo andrò a elimiare le vecchie strutture inserendo i dati in strutture nuove di dimensioni corrette (vedi spiegazione aggiunta e rimozione nodi).

Classe

Variabili private

Ho definito quindi:

- $*\_vertici$: puntatore a array di tipo template
- $**\_archi$: puntatore a matrice di tipo bool
- n_vert : intero non negativo che registra il numero di nodi del grafo

L'idea è quella di associare a ciascun nodo l'indice della posizione in cui si trova nell'array (attraverso la funzione get_index() lo ricavo). In questo modo la matrice di adiacenza avrà associato all riga e alla colonna i-esima il nodo in posizione i-esima nell'array di nodi.

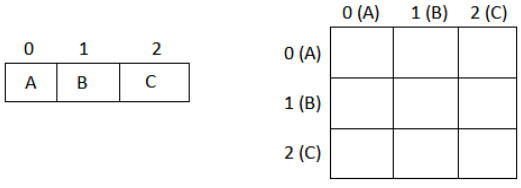

Figure 1: esempio nodi - matrice

Funzioni Public

Costruttore di default Istanzia un grafo vuoto con i valori null alle variabili membro.

Copy constructor Instanzia un grafo copiando i dati da un altro dato in input. Richiama la funzione *i*nsertNodo per inserire uno a uno i nodi nel nuovo grafo e copia la matrice di adiacenza. Se qualcosa non va a buon fine lancia una *b*ad_alloc exception che fa tornare il grafo in uno stato coerente e rilancia l'errore.

Operatore di assegnamento [=] Assegna agli attributi del grafo (this) i valori di quelli del grafo in input. Sfrutta la funzione *std::swap* della libreria standard per scambiare i valori. Utilizza un grafo di appoggio per evitare perdita di dati.

Esistenza di nodo Valuta se esiste un nodo che sia uguale al valore *value* passato in input. Utilizza il const_iterator per controllare l'array di nodi.

Esistenza di arco Controlla se i 2 valori di nodo in input abbiano un arco che li collega. Controlla innanzitutto se esistono e nel caso tramite la funzione get_index ritrova i loro indici associati nella matrice di adiacenza così da valutare l'esistenza o meno dell'arco.

Distruttore Il distruttore richiama la funzione *clean()* che si occupa di eliminare le strutture istanziate.

empty() Questa funzione ha lo scopo di stabilire se il grafo è vuoto.

Inserimento

Inserire nodo Innanzitutto verifica che il nodo che si vuole inserire non sia già presente altrimenti lancia una *noteDuplicateException*. Dopodichè copia i contenuti dell'array (lungo n) di nodi e della matrice di adiacenza (n x n) in strutture temporanee di dimensione $n + 1$. In questo modo si avrà l'array nuovo con l'ultima cella vuota, in questa viene inserito il nuovo valore di nodo dato in input. La matrice di adiacenza nuova avrà una riga e una

colonna in più di valori settati a false pronta per registrare eventuali archi in entrata/uscita dal nodo nuovo. Se tutto va a buon fine sostituisce le vecchie strutture con quelle temporanee appena create. Richiama la funzione *clear()* per ripulire la memoria.

Inserire arco Dati in input due nodi valuta se esistono altrimenti lancia una *nodeNotFoundException*, e se il nodo esiste già in quel caso lancia una *edgeException*. Se queste situazioni non si verificano setta a true la cella della matrice di adiacenza che corrisponde agli indici dei due nodi in input.

Eliminazione

Eliminare nodo Innanzitutto valuta che il grafo non sia vuoto e che il nodo da eliminare esista, altrimenti solleva *emptyException* e *noteNotFoundException*. Crea un nuovo array e matrice di dimensione $n-1$ e $n-1$ x $n-1$ rispettivamente. Copio i vecchi valori nelle nuove strutture facendo attenzione a saltare i valori corrispondenti all'indice del nodo da eliminare. In questo modo gli indici degli altri nodi potranno cambiare ma le loro associazioni rimarranno corrette. Se tutto va a buon fine elimino le vecchie strutture (*clear()*) e mantengo quelle nuove.

Eliminare arco Verifica se i nodi in input esistono e se esiste l'arco in questione, altrimenti lancia una *nodeNotFoundException* e *edgeNotFoundException* rispettivamente. Cerca gli indici corrispondenti ai due nodi e setta a false il valore della cella della matrice di adiacenza corrispondente.

Funzioni get

get_nvert fornisce il numero di nodi dato dal valore n_vert.

get_narchi fornisce il numero di archi del grafo contando i valori true nella matrice di adiacenza.

get_arco Restituisce il valore della matrice di adiacenza nella posizione degli indici dati in input. Se la posizione non è valida lancia una *edgeNotFoundException*. Utilizzato soprattutto per la stampa.

Funzione di stampa (globale) Ho definito una funzione di stampa che sfrutta l'operatore << della libreria standard. Prima stampa il contenuto dell'array dei nodi utilizzando l'iterator e dopodichè la matrice di adiacenza (utilizzando la funzione get_arco() per il valore degli archi). Termina mettendo il *std::endl*, al contrario della prassi usuale che si applica, in quanto secondo me non sarebbe significativo lasciare il cursore di fianco all'utima riga della matrice. Ciò mi ha permesso di visualizzare in modo chiaro il contenuto del grafo durante il debug.

Funzioni private

get_index restituisce la posizione i-esima del nodo passato in input nell'array di nodi. Se questo non esiste solleva una *nodeNotFoundException*.

clean() Questa funzione elimina le strutture puntate dalla classe.

Iteratore

Per l'iteratore ho implementato il **const_iterator** in quanto richiesto dalle specifiche e credo sia il più adatto in quanto non permette la modifica da parte dell'utente dei dati. Il puntatore itera sull'array dei nodi passando tutta la struttura fino alla fine.
Viene dichiarato un puntatore templato che viene inizializzato da un costruttore privato che ha come input il valore dello stesso. Le operazioni che può eseguire sono l'assegnamento, il defereziamento (fondamentale per ricavare il contenuto del nodo). Il puntatore può essere spostato con post-incremento o pre-incremento, sono possibili controlli di uguaglianza e disuguaglianza che vengono sfruttati soprattutto tra l'iteratore di begin e quello di end.

Eccezioni

Per gestire eventuali situazioni non corrette e segnalarlo all'utente ho creato delle eccezioni custom. La classe padre è la **grafoException** che eredità da *std::runtime_error* della libreria standard. Come sottoclassi ho due categorie:

- per i nodi: *emptyException*, *nodeNotFoundException*, *nodeDuplicateException*

- per gli archi: *edgeException, edgeNotFoundException*

Main e Test

Ho definito cinque moduli di test, in base a dove mi trovavo nella scrittura del codice.
Inoltre ho definito una semplice struct **point** con: int $\_x$, double $\_y$ come attributi e i suoi operatori di uguaglianza e di stampa, gli unici necessari per il funzionamento della classe.
Questo mi ha permesso di tenere sotto controllo tutte le varie funzioni man mano che procedevo, controllando che funzionasse anche tutto ciò che avevo scritto in precedenza.

- **testCreate()** crea un grafo e prova a inserire dei nodi testando che eviti di creare duplicati
- **testDelete()** crea un grafo inserendo dei nodi e provando a eliminarne
- **testArchi()** crea un grafo inserendo dei nodi e archi e prova a eliminarne cercando di testare tutte le casistiche possibili
- **testTipi()** crea dei grafi utilizzando tipi diversi come le stringhe della libreria standard
- **testVario()** testa diverse situazioni possibili con grafi di tipi diverso generati anche tramite copy constructor
- **testCustom()** testa un grafo di point (classe custom) per verificarne il funzionamento