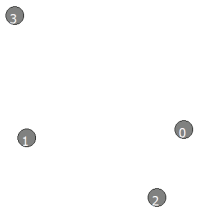
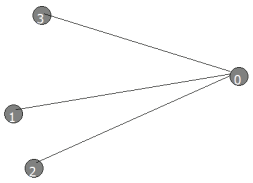
Università degli Studi di Padova
Dipartimento di Matematica
a.a. 2017-2018

RELAZIONE DEL PROGETTO DI PROGRAMMAZIONE AD OGGETTI

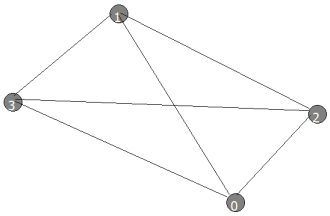
{0,1,2,3}{}



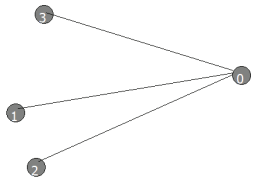
(0,1,2,3}{(0,3)(0,2)(0,1)}



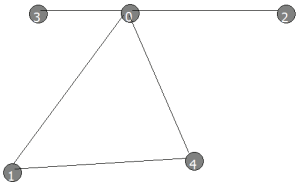
(0,1,2,3}{(0,1)(0,2)(0,3)(1,0)(1,2)(1,3)(2,0)(2,1)(2,3)(3,0)(3,1)(3,2)}



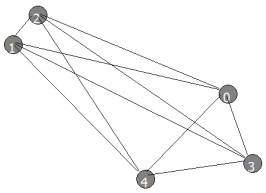
(0,1,2,3}{(0,3)(0,2)(0,1)}



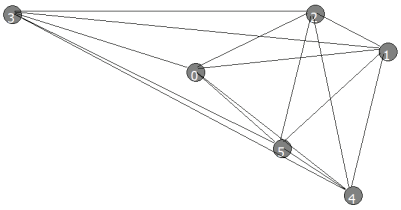
(1,2,3,4,0}{(1,4)(1,0)(2,0)(3,0)(4,1)(4,0)(0,1)(0,2)(0,3)(0,4)}



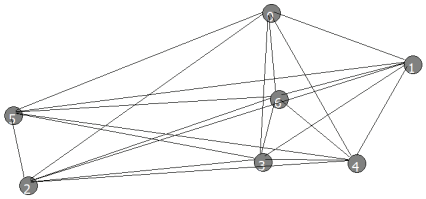
(0,1,2,3,4}{(0,1)(0,2)(0,3)(0,4)(1,0)(1,2)(1,3)(1,4)(2,0)(2,1)(2,3)(2,4)(3,0)(3,1)(3,2)(3,4)(4,0)
(4,1)(4,2)(4,3)}



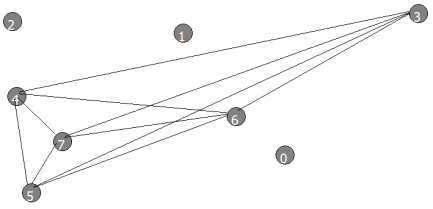
(1,2,3,4,0,5}{(1,4)(1,0)(1,2)(1,3)(1,5)(2,0)(2,1)(2,3)(2,4)(2,5)(3,0)(3,1)(3,2)(3,4)(3,5)(4,1)(4,0)
(4,2)(4,3)(4,5)(0,1)(0,2)(0,3)(0,4)(0,5)(5,1)(5,2)(5,3)(5,4)(5,0)}



(1,2,3,4,0,5,6}{(1,4)(1,0)(1,2)(1,3)(1,5)(1,6)(2,0)(2,1)(2,3)(2,4)(2,5)(2,6)(3,0)(3,1)(3,2)(3,4)(3,5)
3,6)(4,1)(4,0)(4,2)(4,3)(4,5)(4,6)(0,1)(0,2)(0,3)(0,4)(0,5)(0,6)(5,1)(5,2)(5,3)(5,4)(5,0)(5,6)(6,1)
(6,2)(6,3)(6,4)(6,0)(6,5)}



(0,1,2,3,4,5,6,7}{(3,4)(3,5)(3,6)(3,7)(4,3)(4,5)(4,6)(4,7)(5,3)(5,4)(5,6)(5,7)(6,3)(6,4)(6,5)(6,7)
(7,3)(7,4)(7,5)(7,6)}



GRAFOKALK

LA CALCOLATRICE PER GRAFI

GIOVANNI SORICE

MATRICOLA 1144588

Sommario

Introduzione

Il mio progetto Kalk si prefigge lo scopo di creare una “calcolatrice” per grafi.

Le tipologie di grafi trattate sono tre, con la possibilità di ampliarle ulteriormente, “Grafo orientato”, “Grafo non orientato” e “Grafo non orientato planare”.

Queste tre classi hanno in comune la struttura e alcune funzioni, mentre altre sono specifiche per ogni classe.

Cos’è un grafo?

“In mathematics, and more specifically in graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called an arc or line). Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.

The edges may be directed or undirected.”

(da: https://en.wikipedia.org/wiki/Graph_(discrete_mathematics))

Descrizione delle gerarchie usate

Nel progetto sono presenti due gerarchie: una per la parte logica, che rappresenta il grafo e le sue classi derivate, e una per la parte grafica. Insieme permettono la creazione e la modifica di un grafo.

Gerarchia dei grafi:

La gerarchia dei grafi consiste in una classe astratta grafo, resa astratta dai metodi di ricerca del taglio minimo, della ricerca delle componenti connesse, della verifica di essere in presenza di un albero e dalla funzione per ottenere il complemento del grafo.

Inoltre, in questa classe si trovano tutti i metodi di gestione e di richiesta di informazioni del grafo, come l’aggiunta o l’eliminazione di un nodo o di un arco. Si possono anche verificare alcune condizioni del grafo, come la presenza di un cammino tra due nodi o richiedere specifiche di base come il numero di nodi o di archi.

Dalla classe grafo derivano due classi concrete, gOrientato e gNOrientato.

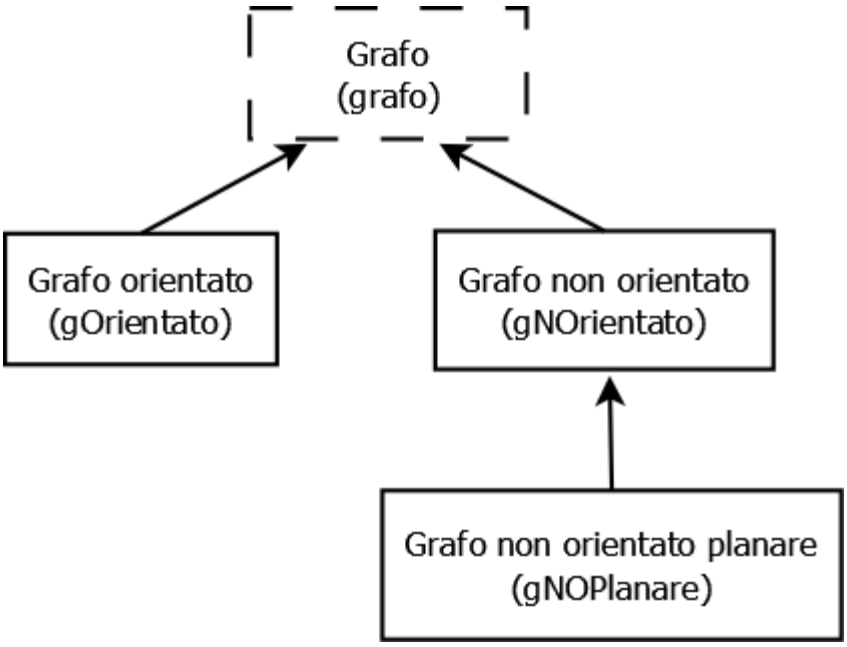
gOrientato tratta i grafi orientati in cui un arco ha un verso e quindi vi è la presenza di un nodo da cui esce un arco e di un nodo in cui entra. Per questo sono presenti i metodi per sapere grado entrante ed uscente di ogni nodo.

Altri metodi caratterizzanti di questa classe sono la ricerca delle componenti fortemente connesse e la verifica della connettività orientata del grafo.

gNOrientato descrive i grafi in cui l’orientamento degli archi non ha rilevanza, dove un arco non ha ne un nodo di arrivo ne uno di partenza. Pertanto, esiste un unico metodo per sapere

il grado di un nodo. I metodi che contraddistinguono questa classe sono la verifica della planarità del grafo e dell'esistenza di sottografi bipartiti o completi.

Dalla classe gNOrientato, deriva la classe concreta gNOPlanare, in cui si vuole mantenere la planarità del grafo e proprio per questo i metodi di gestione vengono ridefiniti in questa classe per mantenere sempre questa specifica.

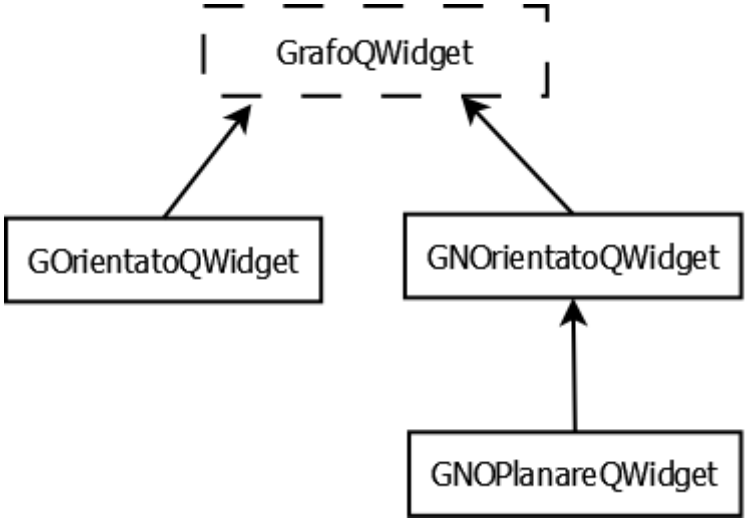


Gerarchia della view:

In parallelo alla gerarchia logica, si sviluppa la gerarchia della view.

La classe astratta GrafoQWidget contiene tutti gli elementi comuni, mentre nelle classi più specifiche si trovano le implementazioni delle caratteristiche proprie di ogni tipologia di grafo trattata.

Questi QWidget sono utilizzati in un QTabWidget, contenuto in un QMainWidget personalizzato che veste anche il compito di controller, dato che è qui che avviene la gestione vera e propria tramite chiamate di metodi al model.



Classi interne

Nel progetto sono definite due classi interne, nodo e listaarchi. Queste due classi vengono utilizzate per la rappresentazione di un nodo e di una lista di archi collegata ad un determinato nodo. Generalmente in essi sono stati ridefiniti alcuni operatori e metodi per facilitare l'interazione.

Descrizione dell'uso del codice polimorfo

In questa gerarchia il poliformismo è essenziale ed utilizzato largamente.

Troviamo infatti polimorfismo nelle funzioni di gestione e di richiesta di informazioni, cioè in void aggiungiN() con e senza parametro intero, void aggiungiA(...) e bool eliminaA(...) entrambi con due parametri di tipo nodo, inoltre anche bool DebolmenteConnesso(), int nArchi().

In aggiunta, le funzioni pure della classe base astratta grafo, vengono implementate nelle sottoclassi rendendo così disponibile un efficace metodo di utilizzo di puntatori e riferimenti polimorfi.

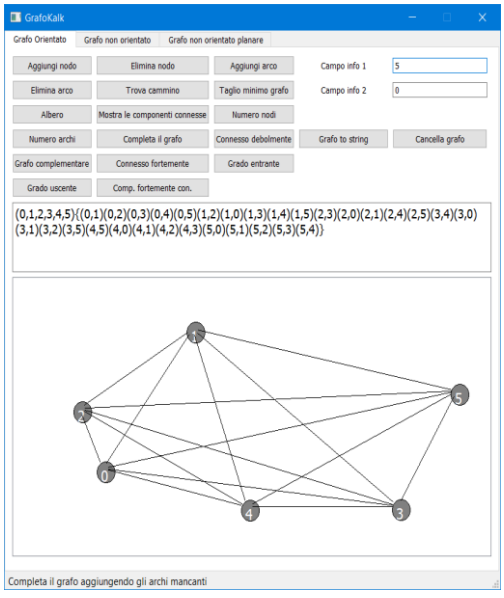
Infine, il distruttore è reso virtuale così da richiamare quello della classe in quel momento usata.

Manuale d'uso

L'utente è in grado di creare un grafo orientato, non orientato e non orientato planare ed eseguire una serie di operazioni comuni e specifiche per ogni tipologia di grafi.

L'interfaccia grafica è divisa in tre tab separate e ad ognuna è associato uno spazio per il grafo. Alcune funzioni richiedono l'input di un numero, compreso tra 0 e 2000, che verrà utilizzato come info dei nodi. Inoltre, la rappresentazione grafica del grafo viene randomizzata ad ogni refresh del grafo. Per i grafi orientati, la segnalazione del verso è fatta nella rappresentazione scritta del grafo.

Per quanto riguarda l'aiuto all'utente durante l'utilizzo, è presente una riga di "help" in fondo all'interfaccia che spiega sinteticamente la funzionalità del bottone.



Metodi presenti

Metodi comuni:

- void AggiungiN(Info nodo facoltativa);

  Aggiunge un nodo al grafo, con l'info indicata se presente.

- bool EliminaN(Info nodo);

  Elimina il nodo e i relativi archi dal grafo.

- virtual void AggiungiA(Info nodo,Info nodo);

  Aggiunge l'arco tra i due nodi del grafo.

- virtual bool EliminaA(Info nodo,Info nodo);

  Elimina l'arco tra i due nodi del grafo.

- bool cammino(Info nodo, Info nodo) const;

  Restituisce true se e solo se esiste un cammino tra i due nodi.

- virtual int TaglioMin()const;

  Restituisce il taglio minimo del grafo, cioè il numero minimo di archi da togliere per rendere il grafo non connesso (sconsigliato usarla sui grafi con un alto numero di archi, ad esempio K7, per l'elevato tempo computazionale necessario).

- virtual bool Albero()const;

  Restituisce true se e solo se il grafo è un albero.

- virtual std::vector<grafo*> ComponentiConnesse()const;

  Restituisce le componenti debolmente connesse del grafo.

- int NNodi()const;

  Restituisce il numero dei nodi.

- virtual int NArchi()const;

  Restituisce il numero degli archi.

- virtual void CompletaGrafo();

  Inserisce nel grafo gli archi mancanti.

- virtual bool DebolmenteConnesso()const;

  Restituisce true se e solo se il grafo è debolmente connesso.

- virtual grafo* ComplementoGrafo()const;

  Restituisce il complemento del grafo, cioè ritorna un grafo con gli stessi nodi ma con gli archi non presenti nel grafo chiamante.

- void cancellaGrafo();

Cancella tutti i nodi e gli archi.

- std::string toString()const;

  Trasforma il grafo in una stringa, suddividendolo in due parti principali. La prima parte definisce i nodi e sono racchiusi tra parentesi rotonde. La seconda parte definisce gli archi, tutti gli archi sono racchiusi tra parentesi graffe, mentre la rappresentazione di un singolo arco è tra parentesi rotonde.

Es. di un grafo orientato (1,2,3) {(1,2) (2,1) (3,2)}

(1,2,3) rappresenta i nodi

{...} rappresenta tutti gli archi

(1,2) rappresenta esattamente l'arco tra 1 e 2, orientato da sinistra verso destra.

Metodi specifici di gOrientato:

- unsigned int GradoEntrante(Info nodo)const;

  Restituisce il grado entrante del nodo con l'info indicata.

- unsigned int GradoUscente(Info nodo)const;

  Restituisce il grado uscente del nodo con l'info indicata.

- virtual bool *FortementeConnesso*()const;

  Restituisce true se e solo se il grafo è fortemente connesso, cioè se per ogni nodo esiste un cammino orientato verso tutti gli altri nodi.

- std::vector<grafo*> ComponentiFortementeConnesse()const;

  Restituisce le componenti debolmente connesse del grafo (ho implementato DFS-based linear-time algorithms rivisitato).

- gOrientato trasposto()const;

  Restituisce il grafo trasposto, cioè ritorna il grafo con gli archi inversamente direzionati.

Metodi specifici di gNOrientato:

- bool planare()const;

  Restituisce true se e solo se il grafo è planare, ciò è verificato tramite il Teorema di Kuratowski e dalla formula di Eulero.

- std::vector<int> bipartito()const;

  Restituisce un vettore con le cardinalità del grafo bipartito se il grafo sottomesso è bipartito.

- unsigned int Grado(const grafo::nodo & n)const;

  Restituisce il grafo di un nodo.

Metodi specifici di gNOPlanare:

gNoplanare non ha metodi aggiuntivi specifici, ma, come anche le altre classi fanno, ridefinisce alcune funzioni delle classi padri per mantenere la planarità del grafo.

Ore utilizzate per lo sviluppo:

- Progettazione: 7h
- Progettazione grafica: 5h
- Sviluppo codice modello: 24h
- Sviluppo codice view: 13h
- Test: 5h
- Debug: 4h

Informazioni sullo sviluppo:

Sistema operativo: windows 10 Home 64-bit

Compilatore: gcc 4.9.2

QT: 5.5.1

Informazioni sulla compilazione:

Per compilare il progetto è sufficiente generare un Makefile appropriato utilizzando il comando qmake, e lanciare make. Il file .pro non deve essere generato dato che è già incluso nei file del progetto.

Per la parte di java, compilare con javac Kalk.java Grafo/*.java e java Kalk