Progetto AlgaT
ALGORITMI GREEDY

Realizzato da:

Gavanelli Sofia

Lena Erika

Ritrovato Martina

Indice

1. L'introduzione

Il progetto AlgaT consiste nella creazione di un'applicazione tutorial su un argomento del corso di Algoritmi e Strutture Dati che nel caso specifico è la progettazione Greedy. L'applicazione è suddivisa in varie sezioni: una prima parte di lezione sull'argomento, una seconda parte di domande di autoapprendimento per l'utente, affinché possa comprendere il concetto appieno, e una terza parte in cui è presente un minigioco.

L'utente ha completo potere decisionale all'interno dell'applicazione: questa, infatti, consiste in un ambiente interattivo nel quale è possibile decidere quando e come andare avanti o indietro oppure interagire con i controlli messi a disposizione. Infine, all'utente verrà richiesto di mettere alla prova le conoscenze apprese.

2. L'applicazione

2.1 L'interfaccia

All'avvio dell'applicazione si apre un menù, dal quale è possibile accedere alle lezioni o al minigame, mentre la sezione esercizi, è accessibile solo dalla corrispettiva lezione e presuppone che l'utente risponda correttamente alle domande per poter procedere.

2.2 L'implementazione

La nostra applicazione si ispira all'organizzazione MVC (Model-View-Controller, figura 1), tipica delle applicazioni JavaFX. All'interno del view sono contenuti i file fxml; ognuno dei quali costituisce una scena ed è gestito dal relativo controller che ne implementa tutte le funzioni, mentre all'interno del model sono contenute le classi che si occupano di gestire i vari controller.

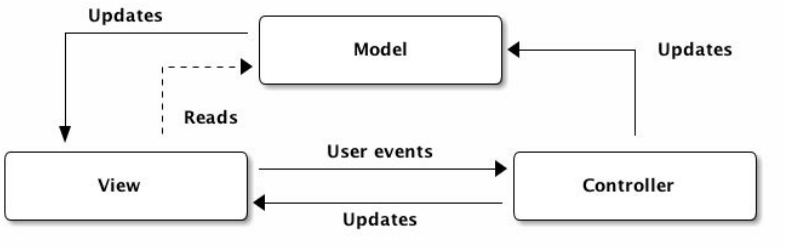


figura 1. l'organizzazione MVC

Le classi che gestiscono l'applicazione sono:

I. progettoController
   → classe che gestisce il progetto nella sua interezza
II. sceneController
   → classe che gestisce tutti i file di una sezione
III. Loader delle lezioni
   → ci sono cinque classi di caricamento, una per ogni suddivisione dell'applicazione (Menù, Introduzione al Greedy, Introduzione agli algoritmi, Kruskal e Prim)

2.2.1 La struttura nello specifico

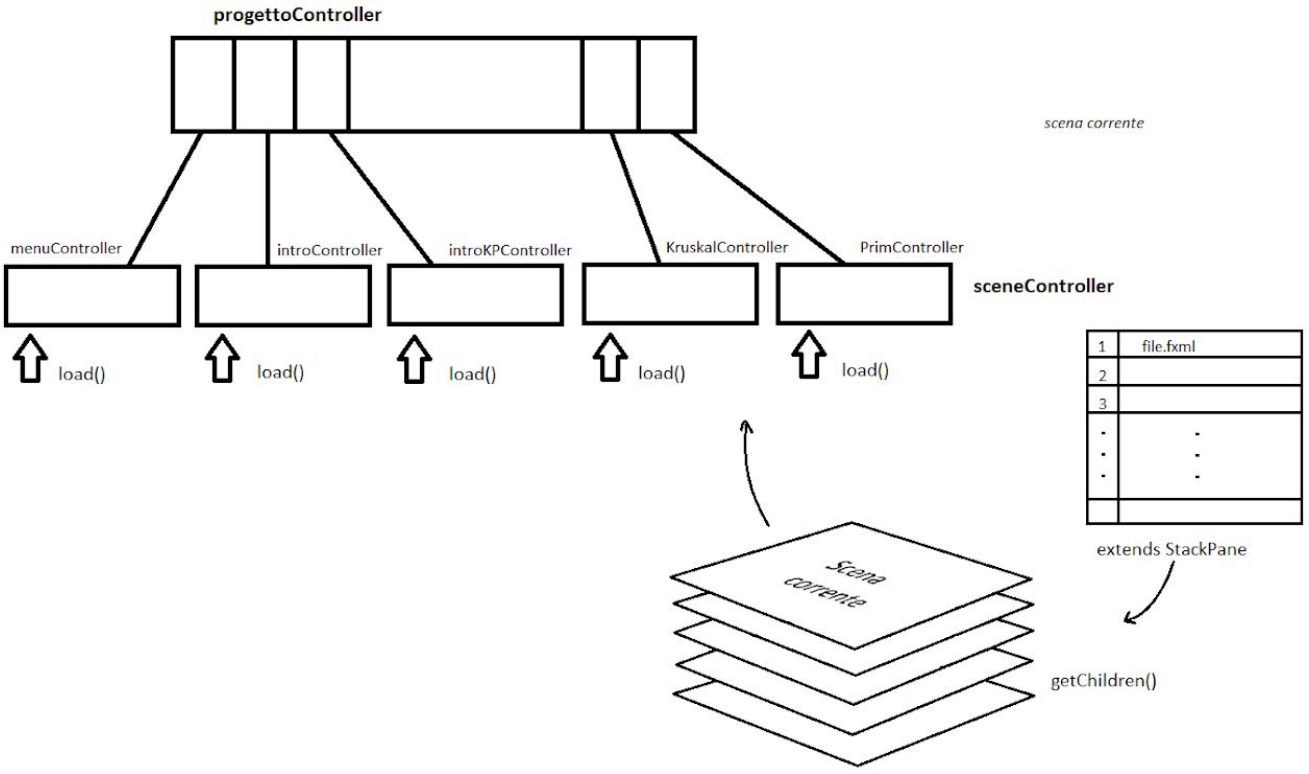
La classe che gestisce l'applicazione nel suo complesso si chiama progettoController e contiene, al suo interno, un array composto dai controller (istanze della classe sceneController) di ogni sezione dell'applicazione (menù, introduzione, introduzione agli algoritmi, algoritmo di Kruskal e Prim). Inoltre, all'interno di questa classe è contenuto il campo *Integer controllerCorrente* ad indicare l'indice a cui corrisponde il sceneController attualmente in esecuzione.

La classe sceneController organizza, invece, la gerarchia dei singoli controller, ovvero delle classi che si occupano di gestire le funzioni e i comportamenti delle diverse scene (file fxml). Ogni controller contiene, infatti, un riferimento al sceneController padre, che tramite la funzione *setSceneParent()* li imposta come figli di un unico StackPane. In questo modo le scene possono essere scambiate modificando tale gerarchia; la funzione *setScene()* (che esegue goBack() e goNext()) imposta, infatti, come primo figlio (cioè lo mette in cima alla pila) il file richiesto dall'utente nel momento in cui clicca su un pulsante dell'interfaccia.

Nel sceneController si trova una HashMap, che gestisce le singole scene (file fxml) associando ad esse un valore intero (chiave), associato all'ordine con il quale vengono visualizzate nella spiegazione. Il sceneController contiene, inoltre, un riferimento alla struttura del progetto (progettoController), che mantiene tutti quanti i sceneController, così da gestirne lo scambio all'interno del progetto.

Il progettoController si occupa, infatti, di caricare in un'unica struttura tutti i sceneController e per fare questo crea tanti sceneController quante sono le sezioni e per ognuno di essi chiama la rispettiva classe loader (es. menuLoad), che si occupa di caricare tutte le scene all'interno del sceneController; questo avviene per mezzo della funzione *load()*, che a sua volta chiama la funzione *loadScene()* per ogni scena da caricare nella tabella Hash.

A questo punto progettoController chiama la funzione *setStructure()* che si occupa di comunicare a tutti i controller a quale struttura Progetto fare riferimento.



2.3 La grafica

2.3.1 application.css

Per gestire lo stile dell'applicazione abbiamo utilizzato un file CSS (Cascading Style Sheets). Questo file è organizzato in selettori (di classe e di identificatore) a cui abbiamo assegnato una serie di regole circa le proprietà che si occupano di gestire background, colori e font.
Una cosa particolare che si può notare nel nostro CSS è il selettore *::hover* che seleziona l'elemento quando vi si passa sopra il puntatore del mouse; questo selettore è stato usato per effettuare una transizione: quando viene attivato i colori dei bottoni vengono invertiti.

2.3.2 Altri aspetti

Per rendere più gradevole l'aspetto grafico dell'applicazione, ci siamo poi occupate di alcuni aspetti tecnici riguardanti la gestione dell'interfaccia e come l'utente possa interagire con essa. Un punto essenziale ci sembrava quello di poter ridimensionare la finestra e di conseguenza l'interfaccia a seconda delle esigenze dell'utente. Pertanto all'avvio il main chiama la funzione start all'interno di progettoController e carica il primo sceneController come figlio dello StackPane root, le dimensioni dello StackPane e quelle della scena sono legate tramite binding così che al ridimensionamento dello stage segua un ridimensionamento proporzionale della scena stessa. La scena viene infatti scalata della stessa misura (in proporzione) della quale viene allargato lo StackPane.

Nel cercare di gestire il ridimensionamento abbiamo riscontrato vari problemi:i le dimensioni dello stage, infatti, non corrispondevano a quelle dello stackPane (root delle scene) e, passando da una lezione all'altra, la finestra diminuiva di qualche pixel ogni volta. Per provare a risolvere questo problema abbiamo calcolato la differenza tra le due finestre, ma questa variava in base alla risoluzione del computer; pertanto, abbiamo optato per una soluzione meno elegante dichiarando una coppia di variabili globali (*double stageWidth, double stageHeight*) che vengono calcolate all'avvio dell'applicazione e variano in relazione al monitor utilizzato, misurando la differenza tra dimensioni dello stage e dimensioni della scena.

Ogni scena è, infine, strutturata su un AnchorPane così che i bottoni principali (back, next, menù..) e i pane siano ancorati tutti alla stessa distanza dai bordi, mentre le informazioni interne sono organizzate in vari Pane.

3. Le lezioni

La lezione tratta il concetto Greedy e tutti gli argomenti ad esso collegati.

Una prima lezione introduce il concetto Greedy e ne esamina le caratteristiche principali (proprietà, vantaggi e svantaggi). Dopo questa introduzione, si passa alla parte implementativa: gli algoritmi; quest'ultimi vengono introdotti da una premessa che fornisce le conoscenze necessarie, nonché una breve dimostrazione di correttezza. Infine, uno alla volta, vengono esaminati i due algoritmi: prima Kruskal e poi Prim.

3.1 L'introduzione al Greedy

Come anticipato, la prima lezione che si incontra è un'introduzione al concetto di Greedy. L'organizzazione prevede una spiegazione molto breve, con esempi applicabili alla quotidianità dell'utente, come quello delle monete riprodotto sotto forma di minigame guidato.

3.1.1 Piccola parentesi: il gioco

Questo minigame è presente in due versioni:

- *guidato*: nella lezione l'utente, dopo aver appreso il concetto base di questa tecnica e visto un esempio proposto, è invitato a fare un tentativo con una cifra diversa; in questa versione la scelta dell'utente è obbligata e l'unico pulsante attivato di volta in volta sarà quello corrispondente alla scelta ingorda; una volta terminato, l'utente potrà proseguire nella lezione.
- *integrale*: accessibile dal menu, l'importo rappresentato di volta in volta è generato casualmente e l'utente sarà libero di fare scelte giuste e sbagliate; nel caso di soluzione sbagliata verrà mostrata quella invece corretta.

IMPLEMENTAZIONE:

Per quanto concerne la scena e quindi il file FXML, il gioco presenta sul lato sinistro una VBox contenente dei Pane cliccabili con immagini delle monete come background. Accanto a ogni pane vi sono Label counter, e, ogni volta che viene cliccata una moneta, la procedura *clickMoney()* aggiorna l'importo e il rispettivo contatore richiamando le procedure *moneyClicked()* e *moneyCount()*. Al centro della scena si presenta l'importo da rappresentare, che viene via via aggiornato sempre dalla procedura *moneyClicked()*.
All'avvio del gioco viene chiamata la procedura *startGame()* che azzera gli array contenenti i dati relativi alle scelte dell'utente e la soluzione (la procedura viene chiamata anche quando si decide di fare una nuova partita e quindi gli array contengono i dati della partita precedente), genera un nuovo importo casuale compreso tra 1000 e 5000, e calcola la soluzione greedy che viene quindi scritta nell'array correctCounters[]. Il gioco termina quando l'importo diventa minore o uguale a 0: in questo caso viene chiamata la procedura *checkCorrect()* che compara gli array correctCounters[] e labelCounters[]*;* in caso di risposta corretta le Label diventano verdi, altrimenti diventano rosse e vengono affiancate dalla risposta esatta.

3.2 Gli algoritmi

3.2.1 L'algoritmo di Kruskal

Il primo algoritmo esaminato è l'algoritmo di Kruskal; il concetto di minimo albero di copertura è necessario alla comprensione dell'algoritmo e, quindi, ne viene proposto un ripasso all'inizio della lezione. In seguito a questo ripasso, messo anche alla prova da un esercizio svolgibile dall'utente, viene presentata una prima idea dell'algoritmo e del suo svolgimento. Quest'ultimo, inoltre, viene affrontato per punti e l'utente stesso può svolgere alcune delle funzioni che, ad algoritmo implementato, sarebbero svolte dal computer (ad esempio l'ordinamento degli archi).

IMPLEMENTAZIONE:

L'implementazione di questa parte di applicazione è molto lineare ed intuitiva; le scene sono pensate secondo un ordine ben preciso (ripasso di un argomento precedente, introduzione e spiegazione per punti dell'algoritmo) pertanto l'utente è invitato fin da subito a seguire la successione delle scene, senza spostamenti significativi. Tuttavia, ci sono due possibili collegamenti con altre scene, realizzati con l'utilizzo delle funzioni *setScene()* e *loadScene()*, che possono collegare l'utente con argomenti già proposti precedentemente, ma la cui conoscenza è necessaria alla comprensione.
Più avanti, nella terza scena, l'utente è invitato a riordinare gli archi del grafo in ordine crescente: questo è stato possibile attraverso l'implementazione della funzione *toColor()* che viene chiamata ogni volta che un arco viene cliccato. Gli archi sono stati inseriti in un array secondo l'ordine in cui devono essere selezionati, quindi, se l'arco cliccato è quello giusto questo apparirà nella parte inferiore della scena; gli archi, infatti, sono stati posizionati in una HBox e poi resi invisibili: una volta selezionati la funzione *setVisible()* passerà da false a true.
La parte di esercizi, infine, è composta da tre domande a risposta multipla e due domande aperte; poiché la struttura dei file .fxml è la stessa, sono presenti solo due controller: uno che gestisce il primo tipo di esercizio e l'altro per il secondo. Il CKruskal_7811 gestisce le domande a risposta multipla in cui le possibili risposte sono state inserite in una VBox e la risposta giusta ha sempre lo stesso id (RightOpt) così che l'esercizio possa essere gestito sempre dalla stessa funzione, *GetAnswer()*. Il controller CKruskal_domande gestisce, invece, le domande a risposta aperta che sono caricate tramite un file di testo (domandeK.txt) poiché il file fxml è uno solo e all'interno della classe è contenuta la funzione *setDomanda(Integer n)* che determina quale domanda mostrare all'utente; questa viene determinata scorrendo il file di testo finché non si raggiunge la riga in cui è contenuta la domanda richiesta, indicata dall'indice preso in input dalla funzione. Questa funzione è poi chiamata all'interno

della funzione che passa alla scena successiva e che, in questo caso, passa alla scena finale solo se le domande sono state esaurite.

3.2.2 L'algoritmo di Prim

Questa sezione si occupa di fornire una breve introduzione all'algoritmo di Prim, algoritmo atto a determinare gli alberi di copertura minimi per un grafo non orientato e connesso. Oltre a costituire un esempio di progettazione Greedy, esso mostra un altro approccio al medesimo problema affrontato dall'algoritmo di Kruskal e trattato nella precedente sezione, evidenziando come, talvolta, la progettazione Greedy possa risultare particolarmente efficace, al punto di offrire due o più soluzioni allo stesso quesito.

IMPLEMENTAZIONE:

All'inizio della sezione è presente un menù grazie al quale l'utente può navigare tra i vari sotto-paragrafi che costituiscono la spiegazione dell'algoritmo di Prim. Il menu è costituito da un insieme di StackPane che evidenziano l'argomento al quale si vuole andare, passando l'evento al cerchio sottostante che a sua volta si colora quando viene selezionato; il controller sposta poi la visualizzazione della scena corrente a quella richiesta.
Se l'utente decide, invece, di procedere con ordine e quindi di avanzare con le frecce gli viene richiesta la verifica di alcune conoscenze, terminata la quale egli può procedere. Si ha poi una semplice e schematica esposizione del problema e la verifica della correttezza di tale algoritmo, che è similare a quella per l'Algoritmo di Kruskal e pertanto è possibile ricollegarsi alla parte di introduzione a tali algoritmi.
Si procede poi con una semplice esposizione di come tale algoritmo possa essere implementato; dopo di che l'utente può verificare di aver capito grazie alla costruzione guidata di un albero di copertura minimo. Viene, infine, trattata brevemente l'efficienza dell'algoritmo prima di passare alla sezione esercizi.
Gli esercizi si articolano su sei domande, due a risposta chiusa e quattro a risposta aperta, le cui risposte sono caricate da un file.txt posto all'interno del programma, così come vengono caricate le definizioni che vengono fornite all'utente tramite l'apertura di alertBoxes nella prima parte della sezione. Il controllo delle risposte alle domande aperte è gestito dalla funzione *controlla( )*, chiamata quando viene attivato il pulsante inserisci. Tale funzione gestisce le risposte date dall'utente alle quattro domande aperte, che si trovano in fondo alla sezione; di volta in volta viene caricata una diversa domanda dal file domandePrim.txt e sempre da questo vengono controllate le risposte.

4. Conclusioni

La struttura del progetto è stata una delle cose che ci ha impegnato maggiormente; inizialmente avevamo fatto alcune prove con una struttura dati array, che però si è poi rivelata inefficiente. Pertanto, abbiamo deciso di strutturare la gerarchia delle scene con delle HashMap.
Una volta caricati singoli blocchi di scene in più mappe, per muoversi con più semplicità da una scena ad un'altra, avevamo pensato di adottare un'unica grande tabella hash nella quale inglobare tutte quante le altre; cosa che abbiamo fatto utilizzando la seguente funzione:

```
public void addMap(sceneController sc) {
        try {
                HashMap<Integer,Node> mappa = getMap(sc);
                for (Entry<Integer, Node> entry : mappa.entrySet()) {
            this.addScene(entry.getKey(), entry.getValue());
            System.out.println(entry.getKey());
            }
          } catch (Exception e) {
                System.out.println(e.getMessage());
        }
 }
```

In questo modo era possibile muoversi da una sezione ad un'altra utilizzando le stesse funzioni che permettono di muoversi da una scena ad un'altra all'interno di una stessa sezione. Tuttavia, anche questa soluzione si è rivelata particolarmente inefficiente, poiché, qualora si fosse voluto aggiungere delle scene ad una sezione o addirittura un'intera sezione, sarebbe stata necessaria una riorganizzazione parziale/totale degli indici della tabella Hash.
Pertanto, scelta la struttura dati per i singoli sceneController, per l'organizzazione di questi ultimi abbiamo optato per una struttura dati statica (un vettore) all'interno della quale fossero già caricati tutti i sceneController così che non fosse necessario caricarli ogni volta che l'utente si fosse spostato da una sezione ad un'altra.

L'indice dell'array al quale andare è passato manualmente alla funzione *loadController(Integer i, Stage window)* [implementata all'interno di progettoController], cosa fattibile dal momento che il numero di sceneController non è elevato, ma che rappresenta un aspetto indubbiamente migliorabile.