ESERCIZIO 1

Analizzare su CARTA l'algoritmo ricorsivo per trovare la sequenza di balckout che massimizzi il numero di clienti coinvolti. Utilizzare lo schema in *Figura 1* e le successive domande per impostare l'algoritmo ricorsivo.

```
// Struttura di un algoritmo ricorsivo generico

void recursive (... , level) {

  // E -- sequenza di istruzioni che vengono eseguite sempre
  // Da usare solo in casi rari (es. Ruzzle)
  doAlways();

  // A
  if (condizione di terminazione) {
    doSomething;
    return;
  }

  // Potrebbe essere anche un while ()
  for () {

    // B
    generaNuovaSoluzioneParziale;


    if (filtro) { // C
      recursive (..., level + 1);
    }

    // D
    backtracking;
  }

}
```

Figura 1: Struttura base algoritmo ricorsivo

Rispondere alle seguenti domande:

- Cosa rappresenta il "livello" nel mio algoritmo ricorsivo?
- Com'è fatta una soluzione parziale?
- Come faccio a riconoscere se una soluzione parziale è anche completa?
- Data una soluzione parziale, come faccio a sapere se è valida o se non è valida?
  (nb. magari non posso)
- Data una soluzione completa, come faccio a sapere se è valida o se non è valida?
- Qual è la regola per generare tutte le soluzioni del livello+1 a partire da una soluzione parziale del livello corrente?
- Qual è la struttura dati per memorizzare una soluzione (parziale o completa)?
- Qual è la struttura dati per memorizzare lo stato della ricerca (della ricorsione)?
- Sulla base dello schema presentato in *Fig. 1*, completare i blocchi (alcuni potrebbero essere non necessari)
  - **A** – Condizione di terminazione
  - **B** – Generazione di una nuova soluzione
  - **C** – Filtro sulla chiamata ricorsiva
  - **D** – Backtracking
  - **E** – Sequenza di istruzioni da eseguire sempre

SIMULAZIONE AD EVENTI SUI MIGRANTI

```
package it.polito.tdp.borders.model;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

import org.jgrapht.Graph;
import org.jgrapht.Graphs;
import org.jgrapht.graph.DefaultEdge;

public class Simulatore {

    //la logica deve stare qui
    //Stato del sistema ad ogni passo
    private Graph<Country, DefaultEdge> grafo;

    //tipi di evento- coda prioritaria
    //1 solo evento
    private PriorityQueue<Evento> queue;

    //parametri della simulazione
    private int N_MIGRANTI=1000;
    private Country partenza;

    //valori in output
    //numero di paesi

    private int T;
    // mappa dei paesi e delle persone in essi stanziati
    private Map<Country, Integer> stanziali;

    public void init(Country partenza, Graph<Country, DefaultEdge>
grafo) {
        //ricevo parametri
        this.partenza=partenza;
        this.grafo=grafo;

        //impostazione dello stato inizale
        this.T= 1;
        stanziali= new HashMap<Country, Integer>();
        //popolo la mappa ad uno stato iniziale
        for(Country c: this.grafo.vertexSet()) {
            stanziali.put(c, 0);
```

```
        }
        queue= new PriorityQueue<Evento>();

        //inserisco il  primo evento
        this.queue.add(new Evento(T, N_MIGRANTI, partenza));
    }
    public void run() {
        //estraggo un evento per volta dalla coda e lo esguo finché la
coda non si svuota
        Evento e;
        while((e= queue.poll())!=null) {
            this.T=e.getT();
            //prendo le persone e le sposto negli stati confinanti
            int nPersone= e.getN();
            Country stato= e.getStato();
            List<Country> confinanti= Graphs.neighborListOf(this.grafo,
stato);
            //l'arrotondamento per difetto  viene fatto direttamente se
ho un int
            int migranti= (nPersone/2)/confinanti.size();
            if(migranti>0) {
                //le persone si possono muovere
                for(Country confinante: confinanti) {
                    queue.add(new Evento( e.getT()+1, migranti,
confinante));
                }

            }
            int stanziali= nPersone-migranti*confinanti.size();
            this.stanziali.put(stato, this.stanziali.get(stato)
+stanziali);

        }
    }
    public int getLastT() {
        // TODO Auto-generated method stub
        return T;
    }
    public Map<Country, Integer> getStanziali(){
        return stanziali;
    }

}
```

EVENTO

```
package it.polito.tdp.borders.model;

public class Evento implements Comparable<Evento>{
    //istante di tempo
    private int t;
    //numero di persone che sono arrivate e si spostano
    private int n;
    //il paese in cui arrivano e da cui si sposteranno
    private Country stato;
    public Evento(int t, int n, Country stato) {
        super();
        this.t = t;
        this.n = n;
        this.stato = stato;
    }
    public int getT() {
        return t;
    }
    public int getN() {
        return n;
    }
    public Country getStato() {
        return stato;
    }
    @Override
    public int compareTo(Evento o) {
        // TODO Auto-generated method stub
        return this.t-o.t;
    }



}
```

PERCORSO NON MINIMO (SE CON PESI) CON VISITE

```
Map<Airport, Airport> visita;


List<Airport> percorso= new ArrayList<Airport>();
        Airport partenza = idMap.get(a1);
        Airport destinazione= idMap.get(a2);
        BreadthFirstIterator<Airport, DefaultWeightedEdge> it= new
BreadthFirstIterator<Airport, DefaultWeightedEdge>(this.grafo,
partenza);
        visita.put(partenza, null);
        it.addTraversalListener(new TraversalListener<Airport,
DefaultWeightedEdge>(){

            @Override
            public void
connectedComponentFinished(ConnectedComponentTraversalEvent arg0) {
                // TODO Auto-generated method stub

            }

            @Override
            public void
connectedComponentStarted(ConnectedComponentTraversalEvent arg0) {
                // TODO Auto-generated method stub

            }

            @Override
            public void
edgeTraversed(EdgeTraversalEvent<DefaultWeightedEdge> ev) {
                // OGNI VOLTA CHE ATTRAVERSO UN GRAFO VOGLIO SALVARMI
LE VISITE IN UNA MAPPA
                Airport sorgente= grafo.getEdgeSource(ev.getEdge());
                Airport destinazione=
grafo.getEdgeTarget(ev.getEdge());
                if(!visita.containsKey(destinazione) &&
visita.containsKey(sorgente)) {
                    visita.put(destinazione, sorgente);
                }else if(visita.containsKey(destinazione) && !
visita.containsKey(sorgente)) {
                    //poiché non è orientato
                    visita.put(sorgente, destinazione);
                }
```

```
                }

                @Override
                public void vertexFinished(VertexTraversalEvent<Airport>
    arg0) {
                    // TODO Auto-generated method stub

                }

                @Override
                public void vertexTraversed(VertexTraversalEvent<Airport>
    arg0) {
                    // TODO Auto-generated method stub

                }

            });
            //non salvo niente perché il salvataggio lo fa il traversal
            while(it.hasNext())
                it.next();
            if(!visita.containsKey(partenza) || !
    visita.containsKey(destinazione)) {
                return null;
            }

            //risalgo all'indietro la mappa perché questa va da figlio a
    padre
            Airport step= destinazione;
            while(!step.equals(partenza)) {
                percorso.add(step);
                step= visita.get(step);
            }
            percorso.add(step);
            //per invertire la lista
            Collections.reverse(percorso);
            return percorso;
```

```
SELECT DISTINCT(STATE)

FROM airports

ORDER BY STATE
```

```
SELECT ID

FROM airports
```

```
WHERE STATE='AK'

SELECT STATE, ID

FROM airports

ORDER BY STATE

SELECT a1.STATE as PARTENZA, a2.STATEas ARRIVO, COUNT(Distinct(TAILNUMBER))
as AEREI

FROM flights f, airports a1, airports a2

WHERE a1.ID=f.ORIGIN_AIRPORT_ID && a2.ID=f.DESTINATION_AIRPORT_ID

GROUP BY a1.STATE, a2.STATE

SELECT m.Season , COUNT()

FROM matches m

WHERE (m.HomeTeam='Juventus' && m.FTR='H') (m.AwayTeam='Juventus' &&
m.FTR='A')

GROUP BY m.Season

SELECT m.Season , COUNT()

FROM matches m

WHERE (m.AwayTeam='Juventus' m.HomeTeam='Juventus') && m.FTR='D'

GROUP BY m.Season
```

GRAFO DEFINITIVO

1. Prendere tutti i possibili vertici e aggiungili in una identity map con chiave della mappa come chiave primaria
   - per quanto riguarda la *idMap* questa è un attributo privato del model che viene inizializzato nel costruttore del model con una new e poi passata come parametro al metodo del dao che ci serve per riempirla e creare una lista delle stesse cose ad essa parallela + +per evitare nullificavano pointer exception++
2. crea una classe per modellare i vertici che sia comoda per poi inserirli nelle relazioni estratte dal database
3. nel DAO devi creare la lista dei vertici estraendo già quelli giusti dal database, estrarre le relazioni e inserirli in una lista con un altro metodo
4. Crea il grafo nel model, secondo il tipo di grafo

Metodo per ottenere le componenti connesse .

```
public int componentiConnesse(){
ConnectivityInspector ci= new ConnectivityInspector(grafo);
return ci.connectedSets().size();
    }
```

Visita di un grafo non pesato non per cammini minimi ma per ottenere le connessioni con un paese

```
public List<Country> trovaVicini(Country paese){
    DepthFirstIterator<Country, DefaultEdge> gfi=new
DepthFirstIterator<Country, DefaultEdge>(grafo, paese);
    List<Country> vicini= new ArrayList<>();
    while(gfi.hasNext()) {
            vicini.add(gfi.next());
        }
    return  vicini;
    }
```

Vista dei nodi immediatamente adiacenti un nodo selezionato dopo aver creato il grafo

```
    public List<Author> trovaCoautori(int id) {
            Author a= this.aIdMap.get(id);
            return Graphs.neighborListOf(grafo, a);
        }
```

SIMULAZIONE GESTIONE MIGRAZIONI

La gestione di simulazioni deve essere approcciata per gradi.

1. Creo una classe simulazione con tutta la logica del mio programma
   - devo conoscere lo stato del sistema ad ogni passo
   - conoscere i tipi di evento e la coda prioritaria che contiene i miei eventi
   - definire i parametri della valutazione
   - scrivere i valori che vorrò in output
2. L'evento è una classe separata sempre e se ho più tipi di eventi devo mettervi all'interno una classe enum

```
public class Evento implements Comparable<Evento>{

    public enum TipoEvento{
        CLIENTE_ARRIVA,
        AUTO_RESTITUITA
    }
    //il tempo qui è continuo approssimato al minuto
    private LocalTime tempo;
    private TipoEvento tipo;
}
```

il comparabile va sempre implementato per le code prioritarie

3. Devo poi mettere nel simulatore due metodi: init e run ++che esegue gli eventi dalla coda++
   - nel metodo init devo prendere i parametri esterni selezionati dall'utente, poi impostare lo stato iniziale popolando anche i risultati, a zero per esempio e aggiungere il primo evento alla coda passandogli ciò che ho definito sopra

- nel metodo run estraggo un evento per volta dalla coda e lo eseguo finché la coda non si svuota

può capitare in simulazioni complesse che la coda vada modificata in corsa

Devi scrivere nel model il metodo per interfacciarti con il simulatore

```
SELECT DISTINCT(districtid) id
FROM events
ORDER BY id
```

LABORATORIO 7

```
    void cercaLista(Livello, Listadeiblackout){
        numerooretotale;
        if(totOre<Oremax){
        }
    }
    for(ogni blackout nella lista per il nerc){
        se(numerooretotale<numerooredato && ){

    }
    }
```

Condizioni:

le ore totali devono essere minori della durate

SIMULAZIONE DISCRETA AD EVENTI

Serve uno strumento che mi permetta di simulare degli scenari e il loro impatto su una serie di fattori quantitativi da valutare .

> il modello può essere statico o dinamico se il tempo è una variabile significativa
>
> può lavorare nel tempo continuo o discreto

Noi trattiamo eventi discreti in cui solo in alcuni istanti di tempo accade qualcosa.Il modello è statico.

SIMULAZIONI AD EVENTI DICRETI

Ad un certo istante di tempo succede qualcosa e la tipologia è nota a priori. Il modello mantiene una lista di eventi più o meno correlati fra di loro. Il simulatore deve simulare le conseguenze di ciò che vede a seconda degli eventi precedenti. Gli eventi sono sempre in ordine di tempo.

Ogni volta estraggo gli eventi in ordine di tempo per elaborarli e per questo si usano le code.

Code prioritarie ordinate per tempo da cui estraggo un evento lo simulo leggendo e aggiornano la situazione del mondo e genero nuovi eventi per la lista. Devo poter modificare il simulatore per una serie di parametri di scenario per poterli tarare - esplorare e determinare scenari alternativi, per poi ottenere misure chiave.

La coda degli eventi è una lista. Un evento è un oggetto che contiene almeno 3 campi: il tempo, il tipo di evento ++entra sale etc++ in numero finito

e poi altri dati come ad esempio dove si è entrati o anche chi per poterlo seguire nei suoi percorsi.

Bisogna pre caricare la coda degli eventi iniziali. Il sistema va poi avanti e la simulazione termina o a coda vuota o in un intervallo di tempo massimo impostato.

La classe simulatore deve gestire la coda degli eventi e quindi devo poi avere una classe evento e poi deve gestire lo stato del mondo: può essere 4/5 valori o una mappa o una classe a parte, è un'istantanea. Il modello crea un simulatore, imposta i parametri come configurazione iniziale, poi il simulatore prevarica la coda con certi eventi iniziali non ancora simulati poi faccio il run() del simulatore e si ferma. Poi mi dai valori massimi, minimi e medi, ma anche mappe o array. Internamente si basa su un forte loop dove ad ogni iterazione del loop elabora un evento. Lo stato del mondo cambia l'effetto dell'evento. Il mondo contiene il mio output di interesse.

La funzione interna al simulatore:

- chiedo alla coda prioritaria qual è il prossimo evento
- ha uno switch a seconda del tipo di evento e dei parametri/strategie di simulazione
- interroga lo stato del sistema

Poi:

- potrebbe aggiornare lo stato del mondo
- potrebbe generare ed inserire nuovi eventi futuri
- potrebbe aggiornare le misure di interesse

```
while(estraggo nuovi elementi dalla coda){
    switch(tipo di evento){
leggo e aggiorno lo stato del mondo
```

```
    }
    }
```

Ci sono dei casi in cui eventi accadono ad istanti di tempo noti a priori e so quando ci sarà l'informazione nuova e allora gli eventi corrispondono al passare del tempo, sono più semplici perché la distribuzione degli eventi è uniforme. Di solito sono eventi asincroni e allora in questo caso devo cercare con la coda l'istante più vicino. Se il modello che gestisco non è deterministico allora posso semplificare il sistema e ci metto una variabile casuale.

Simulo eventi iniziali casuali e aggiungo casualità nel processare l'evento.

CAMMINI MINIMI NEI GRAFI

Dovrò avere una tabella dei predecessori e una tabella delle distanze/pesi per avere cammino minimo da un vertice radice.

Il modo migliore per arrivare a x sia passare da u non dipende dalla destinazione finale del cammino minimo e quindi qualunque percorso minimo che passi da x passa da questo sempre allo stesso modo. Quindi il minimo per arrivare ad un certo punto partendo dalla sorgente è univoco. Mi interessa ottenere il grafo dei cammini minimi. ++ no arco negativo++

Non conviene la permutazione.

Dijkstra

Ho un problema single-source cioè ho un nodo radice e voglio calcolare da lì tutti i cammini minimi nel grafo.

1. ho un vettore delle distanze creato facendo un ciclo: per ogni vertice - l'origine cerco la distanza, la distanza dell'origine pre impostata è 0
2. il set dei vertici visitati è vuoto inizialmente, inoltre ho una coda che inizialmente contiene tutti i vertici possibili di destinazione e che tiene conto delle distanze note dalla sorgente come se fossero infinite

   l'algoritmo cerca di non considerare lo stesso vertice più di una volta ma solo quando è definitivo e non visita in ordine, ma per certezza

3. a ogni iterazione prova i vertici adiacenti al nodo con la distanza minore (partendo sempre dalla sorgente che ha distanza 0<infinito) e presi i vicini ne salva le distanze migliorative (prima iterazione le prende tutte perché migliori di infinito), poi considera quello con la distanza minore, prende quello e aggiorna il cammino, prendendo i suoi vicini e salvando le distanze migliorative e riprendendo dal minimo etc.

tutti i ragionamenti hanno senso solo se oltre abbiamo i pesi altrimenti bastano le visite in ampiezza

Questo algoritmo si implementa così:

```
public List<Fermata> trovaCamminomMinimo(Fermata partenza, Fermata
arrivo){
        DijkstraShortestPath<Fermata, DefaultWeightedEdge> sp= new
DijkstraShortestPath<>(this.grafo);
        GraphPath<Fermata, DefaultWeightedEdge> path=
sp.getPath(partenza, arrivo);
        return path.getVertexList();
    }
```

E' utile quando devo rispondere ad una sola richiesta. Altrimenti meglio floyd warshall e poi non gira mai più.

VISITE AI GRAFI

Ci sono molti tipi di visite, i due principali:

- *visita in ampiezza* parto da un vertice di partenza e poi trovo tutti i vertici raggiungibili dal primo attraversando gli archi del grafo; costruisce una serie di livelli ogni dei quali ha i vertici ad una certa distanza dal nodo sorgente. Il non sorgente è al livello 0, poi dopo un livello posso trovare i vertici al livello successivo: trovo i vertici adiacenti alla sorgente, poi trovo i vertici adiacenti a quelli del livello 1;è un while fino a che l'insieme S è vuoto poiché ad ogni iterazione il numero di vertici visitati è almeno 1 ++ovviamente escludendo quelli già visitati++ in realtà vedo solo la componente connessa alla sorgente perché poi si ferma. Grazie a questo algoritmo trovo i livelli di distanza dalla sorgente ma anche il percorso più breve possibile per arrivare a quella radice così in un colpo solo ho il minimo cammino per raggiungere qualsiasi altro vertice raggiungibile. Ho archi così che permettono di espandere l'albero di visita. L'insieme degli archi sfruttati per i cammini minimi non ripetuti è un albero di visite in ampiezza che codifica tutti i percorsi di ampiezza minima per raggiungere tutti i vertici dalla sorgente. La lunghezza del cammino è pari al numero di archi se il grafo *non è pesato* o il peso è 1 per tutti.
- *visita in profondità* parte da una sorgente e va avanti su un vertice adiacente e così via e arriva fino in fondo per adiacenza, poi fa backtracking e riprende cercando vertici non esplorati. A livello di implementazione ricorsiva è molto semplice, il suo caso terminale è che il vertice a cui arrivo è che non abbia più nessun adiacente e che non abbia ancora vertici non visitati. Ottengo un albero connesso e non ciclico della visita in profondità, ma non dei cammini minimi.

visite in jgrapht

Una visita è implementata mediante un *iteratore* cioè una classe con metodo next. Ho molti iteratori in base al tipo di visita che voglio fare. ++ breathfirstiterator, depthfirstiterator, closesfirstiterator++ ha un metodo next ed un hasnext.

> l'elenco di vertici in ampiezza in profondità non è u cammino tra i vertici, non è detto che siano collegate, come ordine che danno alle stazioni

Definisco un insieme di eventi generati dall'iteratore man mano che vado avanti e trovo vertici e archi, e ogni volta posso agganciarvi un gestore di eventi. Es.: se l'algoritmo trova un nuovo vertice attraverso un arco "informami"

es. `edgeTraversed` che mi dice che l'arco è stato attraversato per questa ricerca.

Creo un oggetto listener che mi intercetta questo evento e mi dice che il vertice di destinazione si raggiunge con quell'arco che fa parte dell'albero di visita.

Listener è una classe che implementa l'interfaccia `TraversalListener` e tutti i suoi metodi . Uso solo edgetraversed. Conviene rappresentare *sempre il precedente che è unico e vado dal fondo indietro*. Devo codificare per ogni vertice quale fosse il suo vertice precedente con una semplice mappa che mi mappa il vertice figlio come chiave e il vertice padre come valore .

L'arco può far parte dell'albero di visita se rispetta solo alcune condizioni. Uno dovrebbe essere un vertice noto uno deve essere un vertice sconosciuto.

Implementazione del metodo per la registrazione degli archi

```
public void edgeTraversed(EdgeTraversalEvent<DefaultEdge> ev) {
```

```
            /*
             * back codifica delle relazioni del tipo child->parent
             * per un nuovo vertice child scoperto devo avere:
             * -child è ancora sconosciuto
             * -parent è già stato registrato
             */
            Fermata sourceVertex= grafo.getEdgeSource(ev.getEdge());
            Fermata targetVertex= grafo.getEdgeTarget(ev.getEdge());
            /*
             * se il grafo è orientato target sarà il child e source sarà
    il parent;
             * se non è orientato potrebbe essere il contrario
             * quindi in generale controllo che il figlio non sia presente
    nella mappa
             */
            if(!back.containsKey(targetVertex) &&
    back.containsKey(sourceVertex)) {
                back.put(targetVertex, sourceVertex);
            }else if(!back.containsKey(sourceVertex) &&
    back.containsKey(targetVertex)) {
                back.put(targetVertex, sourceVertex);
            }
        }
```

quando ho visite in profondità posso usare il metodo già implementato getSpanningTreeEdge e getParent.

per creare la classe che implementa traversa posso farlo come classe privata nel mio model così ho accesso a tutte le variabili di istanza. Oppure inter class anonima usa e getta.

Come trovare cammini minimi in grafi pesati o non semplici etc: per cammino minimo si intende il cammino tra due vertici dati che dia il minimo possibile della funzione peso.

Posso avere 2 problemi:

- il cammino minimo single source cioè da un nodo di partenza trova il cammino minimo per un nodo in arrivo o tutti i nodi di arrivo;
- se invece nel mio algoritmo voglio fare più ricerche dei cammini minimi che però partano da vertici diversi mi conviene usare un algoritmo

di tipo all-pairs, dati due vertici qualunque ho già registrato il cammino minimo ed è immediato trovarlo

per il multigrafo deve avere gli archi e non i vertici come percorso perché altrimenti è poco chiaro

I cammini minimi compongono un albero; il modo più comodo per rappresentare un albero è una mappa di puntatori all'indietro. Devo avere la mappa dei padri e anche il peso del vertice.

se ci sono dei pesi negativi il cammino minimo non esiste

di solito comunque usiamo Dijikstra per il single source e floyd-warshall per l'all pair.

***

ALBERI, CODE E CODE PRIORITARIE

Un albero è un grafo orientato (dalla radice alle foglie), aciclico semplice e connesso perché tutti i nodi hanno un arco che li connette. Se si taglia un solo arco abbiamo una foresta.

Ci permettono di rappresentare i dati in una struttura efficiente, ma non lineare , con natura gerarchica. L'efficienza è data dall'organizzazione gerarchica. Il percorso per raggiungere un nodo è univoco. Una visita è il raggiungimento di un determinato nodo per farci qualcosa. L'attraversamento è un percorso su più nodi avendo una funzione che considera il contenuto di ciascuno dei nodi. I livelli corrispondono alle varie generazioni dei nodi. L'altezza di un albero è il numero totale di livelli.

Un albero particolare è l'albero binario che è un albero in cui ciascun nodo ha 2 figli. E' bilanciato quando i sottoalberi di destra e di sinistra differiscono al più per 1 nodo. Per gli alberi binari c'è una relazione tra il numero dei nodi e l'altezza altezza= log2(NODI) . Viceversa 2^h-1 per i nodi avendo l'altezza h. Questo caratterizza l'efficienza dei nodi. Meglio se mancano nodi sulla destra, mai sulla sinistra.

Nell'attraversamento:

- pre-order si legge il nodo poi il sub-tree di sinistra poi il sub-tree di destra
- in order prima il sub-tree di sinistra poi il nodo e poi il sub-tree di destra
- post-order sinistra-destra-root

Un binary search tree è un albero binario dove a partire dalla route tutto quello che è a sinistra di suo padre contiene elementi più bassi della root mentre ciò che è a destra più alti.

CODE

si basano sulla logica fifo - first in first out. ++talvolta lifo last in first out++. Quando andiamo per ampiezza usiamo una coda fifo.

Priorità: esiste un interfaccia che si chiama queue e ha una classe che si chiama priorityqueue

O uso il comparable o uso il comparator per scegliere la mia regola di priorità. Posso usare le queue posso trovare i cammini minimi in un grafo.

GRAFI E JGRAPHT

Un *grafo* è un insieme di punti di nodi o di vertici che sono dei punti all'interno del grafo e un insieme di archi che può collegare tra loro i vari vertici.

La libreria supporta 16 tipi di grafi e dipendono dal modo in cui connettono i vertici.

In generale:

- *grafo semplice* data una coppia di vertici può esistere oppure no un arco che li collega e i vertici collegati devono essere diversi tra di loro
- *multigrafo* tra una coppia di vertici può esistere più di un arco che li collega
- *pseudografi* in cui sono ammessi dei cappi che partono da un nodo e arrivano in uno stesso nodo ++spesso non servono++
- *non orientati* so la connessione ma non la direzione ,sono una sorta di insieme di coppie non ordiate
- *orientati* l'arco ha un punto di partenza e un punto di arrivo
- *diretti* posso andare in ambo i sensi di un arco
- *pesati/etichettati* posso associare agli archi e ai vertici (come oggetti reali) delle etichette quindi attributi, valori, colori ++es. il grafo pesato.++

Ogni vertice può avere un numero di archi adiacenti diverso, il grado del vertice è il numero di archi che arrivano sul quel vertice e che escono da quel vertice.

un cammino su un grafo è una sequenza di vertici per ciascuno dei quali esiste un arco che li unisce. Un vertice è raggiungibile da un altro se esiste almeno un cammino che posso percorrere per raggiungerlo. Per quanto riguarda i cammini se il punto di partenza è uguale a quello di arrivo allora ho un ciclo. Un grafo è connesso solo se ogni vertice è raggiungibile da ogni altro vertice. un grafo non connesso è composto da più componenti connesse- delle sorta di sottografi.

alberi e foreste un albero è un grafo *connesso* aciclico non orientato non vi posso né aggiungere né togliere archi esistono cammini tra i vertici e sono unici posso scegliere un nodo qualunque e renderla una radice e da essa ho un solo cammino per tutti i vertici. il peso così non è un'etichetta ma anche un costo o cose così.

***

Libreria jGraphT

Graph<V,E> interfaccia principale.

In jgrapht.graph genera grafi di varia tipologia.

In jgrapht.alg ci sono moltissimi algoritmi per generare grafi ++es. minco-st++.

Graph ha un tipo di vertice o un tipo di arco; come vertici ad esempio posso usare corsi o città o qualunque altra cosa. Devo però aver definito bene dashcode e equals. Il tipo di arco è meno interessante esistono infatti due tipi di archi `defaultedge` e `defaultweightededge` (che è pesato).

Un grafo semplice ha un solo arco tra i vari vertici; semplice con loop ha un solo arco tra i vertici ma un arco può convergere nel suo vertice di partenza; multi quando ho bisogno di più archi; pseudo sono multi+self. Default ha i cappi.

Per creare i vertici cerca l'oggetto che ho passato come vertice o un suo equals.

con le parentesi graffe mi indica che non è orientato

con le tonde sono orientati

nell'interfaccia graphs ho una collection di utilities di soli metodi statici.

***

ESERCIZIO METRO PARIGI

1. creo un grafo semplice, orientato, non pesato con le fermate come vertici: metto nel model il grafo con metodi per sfruttarlo
2. Esistono 3 metodi per creare archi a seconda della complessità: la cosa più semplice è sfruttare le connessioni. Posso farlo con un doppio ciclo, con un singolo ciclo ma ci sono problemi nel sistemare l'oggetto in quanto serve una mappa e anche con una singola query per ogni volta e un ciclo ma ci vuole comunque la mappa per selezionare l'oggetto.

***

ESERCIZIO ARTSMIA

- grafo pesato non orientato semplice.
- Faccio un join di una tabella con se stessa per scoprire quando due oggetti sono esattamente nella stessa esposizione

  Query che mi da il conteggio di quante volte due oggetti sono esposti in contemporanea in uno stesso posto

```
SELECT COUNT(*)
FROM `exhibition_objects` eo1, `exhibition_objects` eo2
WHERE eo1.`exhibition_id`= eo2.`exhibition_id` AND eo1.`object_id`=8485
AND eo2.`object_id`=10421
```

Ma è meglio non fare così con il doppio for perché ci vorrebbe troppo tempo

- potremmo pensare di fissare un oggetto e l'altro no

  così che la query mi restituisca tutti i suoi adiacenti con il peso ma l'ultima riga mi visualizza l'oggetto stesso quindi uso un controllo così arrivo ad una soluzione più breve

  * posso infine avere coppie oggetto 1 oggetto 2 per il numero delle volte che sono state visualizzate insieme

Spesso ci serve il riferimento degli oggetti nel grafo. No trucco di solo id per la creazione dell'oggetto. Creo l'oggetto una volta sola ma poi quando mi serve posso sempre ripescarlo lì. La mappa è conveniente ovvero una identity map che è una *hash map*.La chiave è la chiave primaria. Il modo per farla è passare ai metodi del DAO come parametro la mia idMap

CONNECTION POOLING- OBJECT RELATIONAL MAPPING

CONNECTION POOLING

La getConnection costa tantissimo a livello temporale. Ogni volta che chiamiamo quel metodo c'è questo ritardo e può andar bene solo se ho poche query ma pesanti. Per i grafi ciò non va bene. ++non è il caso usare una connessione condivisa tra tutti i metodi++

connection pool è un gruppo di connessioni a disposizione di tutti che vengono aperti all'inizio o alla fine del programma (c'è una libreria di connection pooling) e no l utilizzo am sono pronte all'uso. Faccio la query e la "restituisco".

Praticamente ho un'interfaccia detta data source che implementa il connection pool; crea e gestisce il prestito delle connessioni. Così uso DataSource invece di driver manager. E il suo metodo close lo rimette semplicemente nel pool.

Non è implementato nativamente, bisogna aggiungere una libreria aggiungendo il pooling verso jdbc mysql.

```
HikariDataSource ds= new HikariDataSource();

ds.setJdbcUrl("jdbc:mysql://localhost:3306/nomedatabase");
ds.setUsername("root");
ds.setPassword("password");

...

ds.getConnection();
connection.close(); //sempre altrimenti va fuori

//per chiudere del tutto il connection pool
ds.close();
//dovrei metterlo in un metodo di stop
```

restituisci sempre le connessioni

Riuso sempre lo stesso oggetto.

Creo come variabile statica con metodo getconnection.

inizializzazione Hikari data source.

```
    private static HikariDataSource ds= null;

        public static Connection getConnection() {


            String jdbcURL= "jdbc:mysql://localhost:3306/dizionario";


            try {
                if(ds==null) {
                ds= new HikariDataSource();
                ds.setJdbcUrl(jdbcURL);
                ds.setUsername("root");
                ds.setPassword("provaprova");
                }

                Connection conn= ds.getConxnection();
                //Connection conn= DriverManager.getConnection(jdbcURL);
                return conn;
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            return null;
        }
```

Devo però usare una libreria di logging.

libreria di logging

metodo per fare system.out, con cui decidere come trattare i messaggi del programma mandandoli sulla console o sul database. Con il simple stamperà i messaggi nella console2

***

OBJECT RELATIONAL MAPPING

Dobbiamo avere delle regole di corrispondenza.

1. per ogni entità del database crea un oggetto java (le tabelle delle relazioni molti a molti non servono)
2. il nome delle classi bean devono essere gli stessi delle entità al singolare
3. La classe dovrebbe avere una proprietà privata per ogni colonna nella tabella con gli stessi nomi, con sintassi java con cammello usando tipi di dato simili; id numerici non mi servono (ovvero le chiavi esterne).
4. il costruttore principale deve accettare tutti i campi nel bean TUTTI (no chiavi esterne)
5. Ci deve essere un metodo get-set per tutte le proprietà
6. equal e hashCode devono necessariamente usare le chiavi primarie.

Alcune relazioni devo esplicitarle come proprietà della classe bean, ma solo nel verso in cui mi interessa, perché altrimenti è costoso. Rappresento solo le relazioni che mi servono.

non devo usare la semplice foreign key per creare la mia associazione ma creo un nuovo oggetto nella tabella studente di java.

Devo evitare scansioni di dati in java (loop o loop annidati).

La relazione 1-molti si mappa facendo una collezione. Set per essere fedele ad SQL ma a volte può essere comodo avere una lista per ordinarli con order by.

DATA E TEMPO IN JAVA

Il tempo macchina è un numero intero a partire da una certa data iniziale. Il nostro tempo però è articolato soprattutto in loop ripetitivi che non si incontrano in modo ordinato.

UTC coordinated universal time- è il riferimento rispetto al quale viene misurato il tempo.

CET= central european time.

La data si scrive ANNO-MESE-GIORNOTHH:MM:SS+00:00 non è unico perché l'utente ha bisogno di questo.

Java ha delle classi java.util.date e java.util.calendar ma sono vecchie e deprecate.

java time

Tutti gli oggetti della libreria sono dati immutabili e quindi devo sempre creare oggetti nuovi ma con metodi di factoring. Ho oggetti separati per le mie rappresentazioni.

La classe principale è LocalDate

Metodi di factoring:

- of prende dei valori esistenti e li mette nell'oggetto
- from fa delle conversioni da datetime a date per esempio (magari perdo qualche informazione)
- parse data una stringa in un oggetto data
- get interrogare i campi
- with restituisce una copia di quell'oggetto con uno dei campi modificato ++es incrementi++

Instant segna il tempo macchina in nanosecondi.

Usa tutte le funzioni già fatte dell'aritmetica.

Period and Duration ++le useremo poco++ la durata è indipendente dal momento di inizio cioè da una data effettiva. Importante: `period.between(data, data)`

C'è compatibilità per conversione.

date in sql

Standard mysql:

- DATE YYYY:MM:DD
- DATE TIME data + ora
- TIMESTAMP rappresenta data e ora però da 70 a 38 una sorta di tempo macchina
- TIME che è usato per l'orario o un intervallo di tempo
- YEAR che viene codificato solo su un byte fino a 2069 da evitare

Per quanto possibile fai tutte le operazioni sulle date con le query sql assolutamente

Mai usare stringe o interi.

Se non posso fare l'operazione allora devo estrarre la data direttamente mysql dà 3 classi posso usare

- java.sql.date che *non è .util* solo data non ora
- java.sql.Time
- java.sql.timestamp

Come faccio a ricondurmi a java 8: date posso ricondurlo a toLocalDate valueOf serve per settaria. Nei metodi del dao si fanno queste operazioni con le date

***

appunti miei sulle date

COSE UTILI

- Se duplico un oggetto di una classe mia allora devo implementare un metodo clone del tipo

```
public Quadrato clone() {
        Quadrato q = new Quadrato(this.size) ;
        q.mappa = new ArrayList<Integer>(this.mappa) ;
        return q ;
    }
```

se invece sono dello stesso tipo già di java basta che uso il loro stesso costruttore

- La ricorsone ha sempre una interfaccia pubblica con cui passo i parametri del metodo ricorsivo e faccio la return della struttura dati popolati all'interno della ricorsione

- Set meglio di list in ricorsione per evitare i duplicati

- il punto D dell'esame si deve risolvere con poche linee di codice sfruttando una visita in profondità, classi per il cammino minimo, elenco dei vicini di un certo nodo che si risolve con Graphs con neighbourlistof e poi li ordino

- come estrarre una LocalDateTime da database con formato data+tempo

```
LocalDateTime.of(res.getDate("reported_date").toLocalDate(),
res.getTime("reported_date").toLocalTime())
```

- comparare due numeri double senza perdere cifre decimali nell'approssimazione

```
public int compareTo(Vicino o) {
        // TODO Auto-generated method stub

        return (int)(this.distanza*1000-o.getDistanza()*1000);
    }
```

- se devi usare un comparatore su un certo risultato prendendo

esempio mio script esercizio voti nobel

```
    private list combinazioneMigliore(lista combinazione, int m){
        int mediaMigliore;
        List migliore;
        //con for each
        //salvataggio
        if(m=0){
            int somma=0;
            int sommac=0;
            int media=0;
            for(esame e: combinazione){
                somma+= e.crediti*e.voto;
                sommac+= e.crediti;
            }
            media= somma/sommac;
        if(media> mediaMigliore){
            mediaMigliore= media;
            migliore= new List(combinazione);

        }
        }
        for(esame e: esamitutti){
            int creditiresidui= m-e.crediti;
            if(!combinazione.contains(e) && creditiresidui>=0 ){
                combinazione.add(e);

                combinazioneMigliore(combinazione, creditiResidui);

                combinazione.remove(e)
            }
        }
    }
```

Il professore fa una doppia chiamata ricorsivi una aggiungendo l’esame l’altra non aggiungendo l’esame e poi facendo backtracking

esempio mio script per laboratorio 5

```
passo come parametro 0
public List<String> calcolaAnagrammi(Stringa parola,String parziale,
int L){
        if(L==parola.length){
            anagrammi.add(parziale.clone());
            return;
        }
        for(int i=0; i<parola.length; i++){
//aggiungi un if con un counter particolare
            parziale+=parola.charAt(i);

            calcolaAnagrammi(parola, parziale, L+1);

            parziale= parziale.substring(0, pariziale.length-1);
        }
}
```

Il livello nella ricorsione per gli anagrammi è la lettera scritta. Una soluzione parziale è formata dalle lettere della parola ricombinata per tutte o parte delle lettere; non si può sapere se è valida o no una soluzione, ma so che è completa quando il livello è pari alla lunghezza della parola da anagrammare. Il controllo di validità per l'anagramma si fa in un altro momento. La struttura dati è una lista.

COMPLESSITA' COMPUTAZIONALE

O() è il limite massimo sul tempo dell'esecuzione

omega() è il limite inferiore, andamento minimo

theta()è la funzione di cui è nell'ordine g, cioè f è tra k x g e p x g dove k e g sono 2 costanti.

Metodo di analisi semplice della o grande basata su 5 regole

1. ogni funzione singola del mio programma ++esempio println++ ha complessità costante
2. istruzioni diversi hanno tempi di esecuzione diversi ma sono ignorati questi tempi
3. nelle strutture condizionali if-else si calcola il ramo if e si calcola il ramo else ma si usa il ramo peggiore
4. se ho una sequenza di passi la complessità è data dalla somma dei singoli passi ricordando che l'o grande maggiore assorbe il resto
5. se ho un'istruzione all'interno di un ciclo e questo ciclo viene eseguito n volte allora la complessità sarà n per la complessità della singola operazione

La ritorsione è una delle ragioni maggiori di una complessità di tipo computazionale.

Posso ridurre il fattore di branching e generare meno problemi e poi a generarli di dimensione più ridotta

LAB04 TIPS UTILI

1. usa una stringa per avere il carattere monospace e incolonnare i dati

   ++scrivi stringa++

RICORSIONE

Divido problemi complessi in problemi più semplici di cui faccio una ricerca sistematica delle soluzioni per arrivare alla soluzione del problema globale ++ ovviamente devo avere un metodo operativo per elencare le possibili soluzioni++. Si parla di ritorsione ogni qual volta ho una funzione o una classe che al suo interno richiama sé stessa.

Per creare un algoritmo ricorso mi servono 2 cose:

- una regola di ricorrenza cioè che esprima un problema che esprima lo stesso problema in termini di se stesso però più piccolinoooo
- una regola di terminazione
- serve un metodo poi per mettere insieme le soluzioni

Devo ragionare per livelli: primo divido il problema e poi lo ricombino.

Nel profondo della ricorsione se dobbiamo salvare qualcosa ne dobbiamo per forza salvare una copia con il metodo clone o un costruttore clonante, perché il valore è quello su cui si lavora e cambia continuamente.

N.b. ci sono due tipi di problemi:

- problemi in cui voglio tutte le possibili soluzioni
- problemi per i quali mi basta una soluzione sola e posso interrompere gli algoritmi ricorsivi

Se sono fortunato e il problema lo consente posso fare pluning ovvero potatura: avendo cioè un albero delle possibilità posso eliminare direttamente un ramo di soluzioni.

MODEL VIEW CONTROLLER PASSI FONDAMENTALI

Passi fondamentali per realizzare il programma sfruttando il modello Model View Controller.

1. *creazione del model* che va messo in un secondo package significativo separato dal package principale
2. *modeifiche del main* nel main devonsao essere effettuate varie modifiche

```
FXMLLoader loader= new
FXMLLoader(getClass().getResource("SpellChecker.fxml"));
BorderPane root = (BorderPane)loader.load();
//scene e richiamo css [...]
Scene scene = new Scene(root); //tolgo le dimensioni
SpellCheckerModel model= new SpellCheckerModel();
SpellCheckerController controller= loader.getController();
controller.setModel(model);
```

3. *modifiche della classe controller* deve essere richiamato il modello nel controller per far funzionare tutto

```
private SpellCheckerModel model;

public void setModel(SpellCheckerModel model) {
        this.model = model;
    }
```

ESERCIZIO GESTORE DEI CORSI

Approccio top down, esempio: scrivo già il risultato che mi aspetto nel controller anche se non ho ancora definito le funzioni nel modello.

Considero i dati che si scambiano il database e java: ORM *object relational mapping*, devo riprodurre le strutture delle tabelle che sfrutto con le query nelle mie classi quindi nei miei oggetti. Per quasi tutte le tabelle nel database ci deve essere la classe in java con lo stesso nome, con gli stessi nomi degli attributi tutti privati e con tutti i getter e setter; non tutte le tabelle perché le tabelle relazionali per le relazioni molti a molti si possono non fare. Devo anche generare oltre getters e setters e il costruttore hash code ed equals utilizzando la chiave primaria. Le classi così create sono dette java beans.

Nei model e java beans non metto l'interazione col database, così creo delle classi DAO. Per ogni tabella creo un DAO specifico. Il package è a parte.

> Posso decidere di filtrare con la query o filtrare dopo che ho preso tutti i dati della tabella.

Nelle classi Dao non includo comunque la mia connessione al database, per ometterei dettagli della mia connessione.

ENTRY è una classe già fatta che mi restituisce ad ogni ciclo la coppia chiave-valore

DATABASE E JDBC

Connessione tra database e java in 2 passaggi:

1. caricare un driver
2. interagire col driver

Quindi ogni volta che voglio dei dati dal database devo:

1. definire url di connessione
2. stabilire la connessione
3. creare un oggetto statement
4. eseguire una query o un update
5. processare i risultati
6. chiudere la connessione

Il database è di supporto al modello nel MVC.

Richiamiamo le funzionalità del driver in modo indiretto. JDBC deve sapere: quale driver ci serve, a quale database collegarci. Tutto è nella stringa di connessione ovvero connection url. Nel caso di mysql la stringa di connessione è fatta così:

```
jdbc:mysql://[host:port],[host:port].../ [database][?propertyName1]
[=propertyValue1 ][&propertyName2][=propertyValue2]
```

Dovrebbe venire circa una cosa del tipo:

```
jdbc:mysql://localhost:3306/dizionario?
user=root&password=provaprova&useJDBCCompliantTimezoneShift=true&useLeg
acyDatetimeCode=false&serverTimezone=CET
```

?serverTimezone=Europe/Rome

++ con la & solo se segue ad altre cose++

?useUnicode=true&useLegacyDatetimeCode=false&serverTimezone=Europe/Rome

Posso aprire una connessione al database, chiedendo alla libreria JDBC di analizzare il contenuto della stringa e caricare il driver opportuno per aprire così una connessione, in cui possono passare dati e query.

> Mai istruzioni nel main perché è un metodo statico e quindi non mi permette di salvare variabili istanza; quindi ogni volta per provare una classe creo un metodo normale non statico e il main crea una istanza di se stesso

Creo la stringa di connessione e la inserisco

```
    try {
                Connection conn=DriverManager.getConnection(jdbcURL);
            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
```

così si creerà un oggetto di tipo connessione.

Il driver manager va a cercare in tutte le librerie quali sono le classi che implementano l'interfaccia driver e in tutte le classi cerca la gestione di mysql.

OGNI volta che creo una connessione devo subito chiuderla ++anche se java ha la garbage collection, così è il server a liberarsi++

```
    conn.close();
```

Anche se gli oggetti di connessione non ci sono più.

Le istruzioni sql vengono inviate con le statement quindi per fare una query devo creare uno statement con l'istruzione e dire al mio driver JDBC di mandarla al database ed eseguirla. Basta una connessione sola per più query diverse.

++statement è un'interfaccia++

++chiedo alla connessione di creare la connessione, factoring++

```
Statement statement= connection.createStatement();
```

Devo assolutamente usare java.sql non le altre

Quindi scrivo una stringa che è la nostra query e poi chiedo allo statement di eseguire la mia query con executeQuery, che verrà salvata in un oggetto resultset.

```
String sql= "SELECT * FROM parola " +
                    "WHERE nome LIKE 'Z%'";

ResultSet rs= st.executeQuery(sql);
```

prima "debugga" la query dentro il mio dbms

ResultSet è un riferimento collegato al risultato, ha un meccanismo a finestra/cursore: di partenza non trasferisce nulla nel programma, il cursore è virtuale e si muove sulla tabella dei risultati e posso spostarlo a mio piacimento. L'oggetto implementa due famiglie di metodi:

- quelle per spostare il cursore
- quelle per raccogliere i dati

All'inizio il cursore è posizionato prima della prima riga, arriva fino alla riga dopo l'ultima riga. Il metodo per andare avanti è il metodo .next() che è bollano quindi vero se c'è qualcosa dopo, false se non c'è niente.

Per raccogliere i dati uso metodi getTipodidato("identificativo della colonna di quel dato").

Se non devo fare una query ma devo eseguire INSERT, UPDATE, DELETE allora devo usare il comando executeUpdate(stringasql) che mi ritorna un intero che mi dice quante righe sono state modificate nel database. Per un generi-

co execute per esempio per la creazione c'è execute(sql) che ritorna un booleano.

Stringa parametrica

Se l'utente deve inserirmi un pezzo della query per esempio con una getText. PreparedStatement posso separare l'istruzione, niente interpolazione sotto forma di stringa. Separiamo un template della query sql da scrivere e aggiungiamo le posizioni parametriche. La gestione che si effettua con preparedStatement è :

- definisco la stringa della query
- dove devo mettere delle variabili metto il punto interrogativo '?'

Creo un oggetto sfruttando prepareStatement(passolastringa), passo la stringa e poi in esecuzione passo i parametri. Questo perché con prepara statement hanno dei metodi di set: il numero del punto interrogativo da sostituire e il valore da restituire.

STATEMENT NON SI USA MAI.

Design pattern- DAO

Possiamo creare delle classi di due tipi:

- le classi client che usa i dati- la classe MODEL che però non deve sapere come è strutturato il database
- le classi DAO data access object - quindi un oggetto che è lì per fornire accesso ai dati , quindi tutto il codice di accesso ai database è lì ++accedono senza sapere perché sta interrogando il database++ lo

potrei far viaggiare con il database. ++ è l'unico ad usare DriverManager, Datasource, ResultSet++

Il client e il Dao si interfacciano grazie agli oggetti detti "data transfer object" che non hanno metodi ma proprietà ++ al massimo hanno validazioni++

DAO

Il dao non ha stato quindi non ha variabili di istanza a parte la connection, forse, perché così è l più generica possibile. Quindi la rendo tipizzata. Il dao deve offrire metodi CRUD[1].

E deve anche offrire una classe di ricerca. Costruisco poi nel Model degli oggetti che facciano da specchio alla struttura del database.

l'identità di un oggetto in java e la tabella con la sua chiave devono essere coerenti

Mi serve anche creare una classe che abbia come unico scopo in assoluto creare la connessione con il database, così nei dado

[1] create- read- update- delete(MAI)

SET E MAPPE

SET

Modellano degli insiemi e quindi non possono contenere elementi duplicati. Perciò abbiamo solo add, remove e contains. *Non abbiamo accesso posizionale.* La ricerca è immediata. Se il nostro programma ha bisogno solo di add, remove e contains allora è meglio usare un set per la sua velocità. Quando iteriamo su un set non possiamo supporre un determinato ordinamento. I LinkedHashSet hanno un ordine predittibile, che è quello di inserimento. Queste operazioni sono velocissime perché internamente si basano sulle HashTabes, che permettono di implementare un array associativo (che non usa un indice, ma una "parola" –> chiave). Un set salva delle chiavi. Grazie alle HashFunctions prendo la mia chiave, la trasformo in un numero e uso quel numero come indice di un array, ovvero il luogo in memoria dove salvo le informazioni.

Hash Function

Mappa un set di dimensione arbitraria, su un insieme più piccolo di indici. La funzione H converte le stringhe in un numero e poi quei numeri vengono compressi nell'intervallo da 0 a m-1(dimensione array). Si usa l'operatore % modulo.

L'altro modo è avere un numero tra 0 e 1 ed espanderlo alla dimensione dell'array.

++ deve essere deterministico, si usa in crittografia, ma lì non deve essere invertibile++

Quando sappiamo che i nostri valori sono fissi e non troppo grandi possiamo salvare il nostro valore come chiave. In quel caso la funzione è perfetta. Altrimenti la probabilità di collisione ci sarà sempre.

Metodi per buone funzioni di Hash:

- Open addressing ma è comunque troppo casuale
    * Chaining negli slot dove vanno a finire più elementi vado a salvare questi elementi in una lista

  N.b. per le stringhe c'è già un hashcode ben implementato da utilizzare

METODO HASH ECLIPSE

Per tutti gli oggetti che non sono stringhe il metodo hash è implementato ma tiene conto dell'indirizzo di memoria dell'oggetto e non del contenuto dell'oggetto *problemi*.

Equal ed hashcode *devono* essere sempre ridefiniti insieme

Con il .contains delle liste la funzione di hash comunque non centra perché per ogni elemento si richiede il l'equals.

Se devo rifare un pezzetto dell'equals devo andare di pari passo con la funzione di hashcode.

MAPPE

Le mappe sono un tipo di dato più moderno, implementate o con HashTable o con strutture ad albero. Le mappe non derivano dall'interfaccia collection ma da quella mappa.

N.b. containskey è più veloce

LABORATORIO 03 TIPS UTILI

1. per utilizzare e settare in maniera corretta le label devo utilizzare la libreria

```
import import javafx.scene.control.Label;
```

2. per avere un menu a tendina scomodo da sfruttare devo utilizzare la ComboBox e le istruzioni da utilizzare sono :

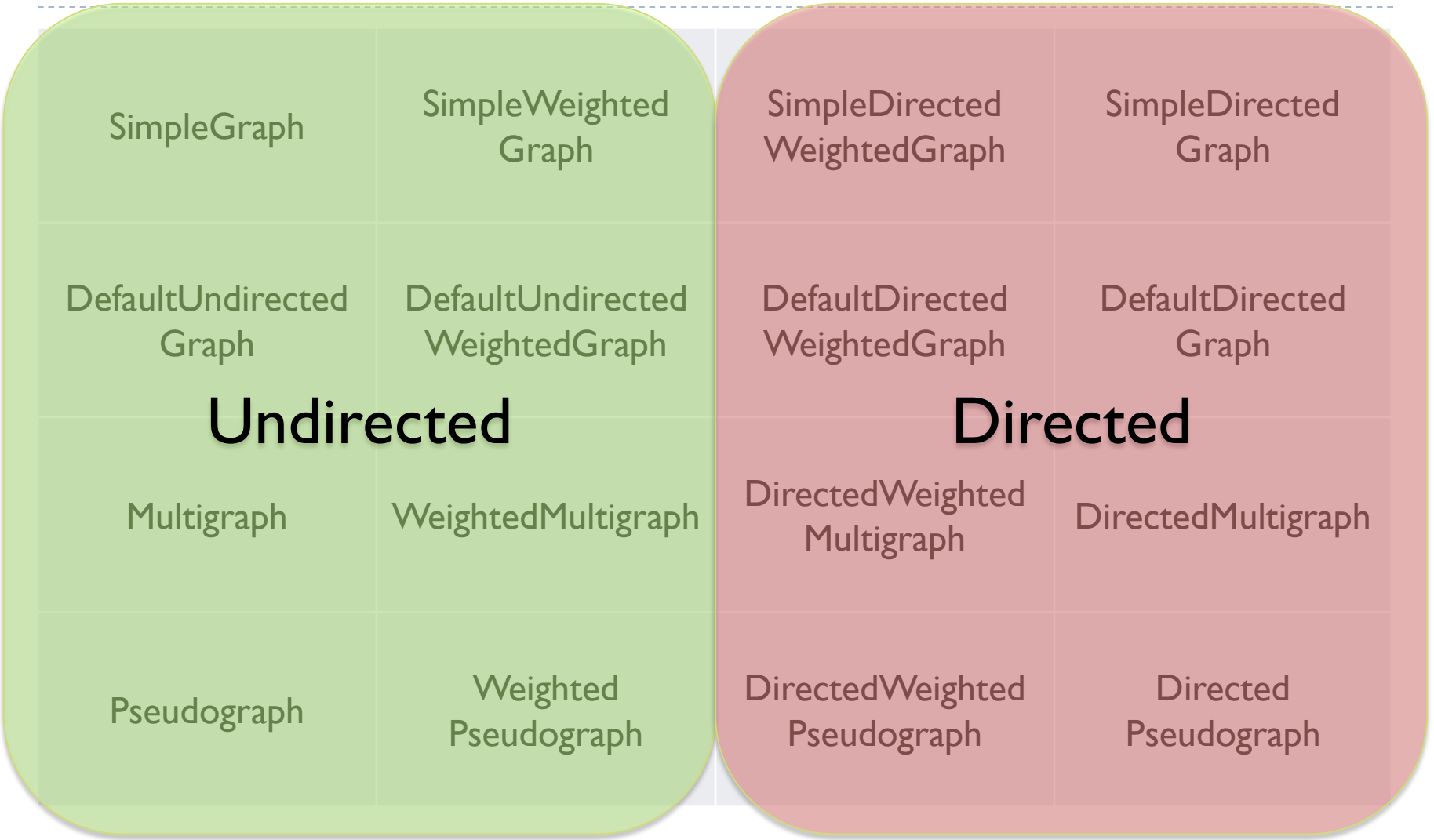
```
import javafx.scene.control.ComboBox;
// questo va messo nel costruttore dove viene inizializzato il model
language.getItems().addAll("italian", "english");

//per verificare la scelta dell'utente
language.getValue();
```

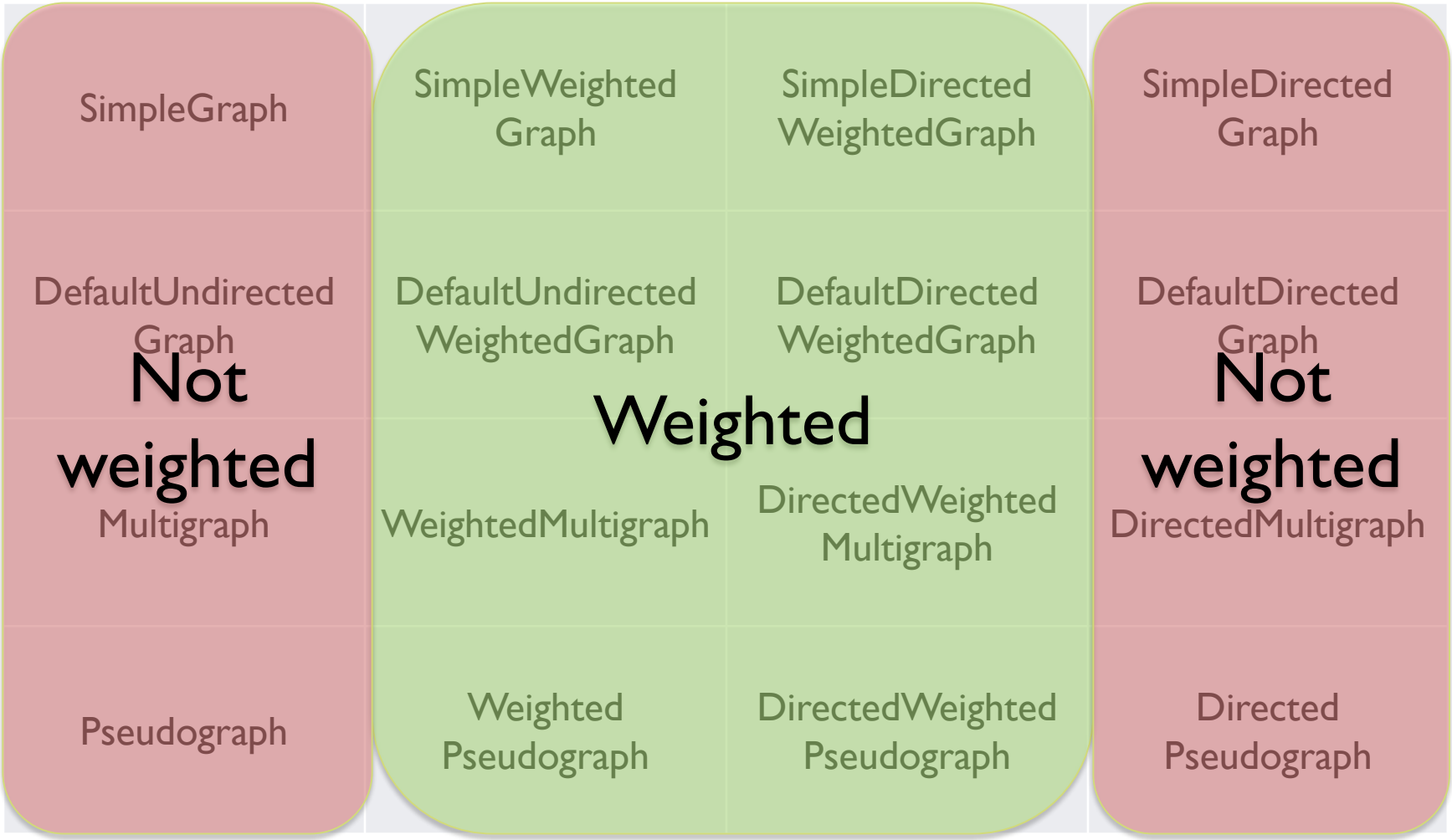
3. validazioni da fare per questo programma:
   - verifica che il dizionario sia stato caricato e correttamente
   - verifica che l'utente abbia selezionato una lingua ovvero sblocca il campo di testo e i bottoni solo dopo che il valore della lingua non è nullo
   - metti inizialmente i bottoni a disable
   - verificare che l'utente abbia scritto qualcosa
4. quando vuoi sfruttare la proprietà disable di qualche campo prima clicca sul bottone disable di scene builder
5. per creare un ciclo sfruttando un tokenizer si fa così:

```
StringTokenizer st = new StringTokenizer(inputText, " ");
while (st.hasMoreTokens()) {
            inputTextList.add(st.nextToken());
        }
```

Graph Implementation Classes



Graph Implementation Classes



Graph Implementation Classes