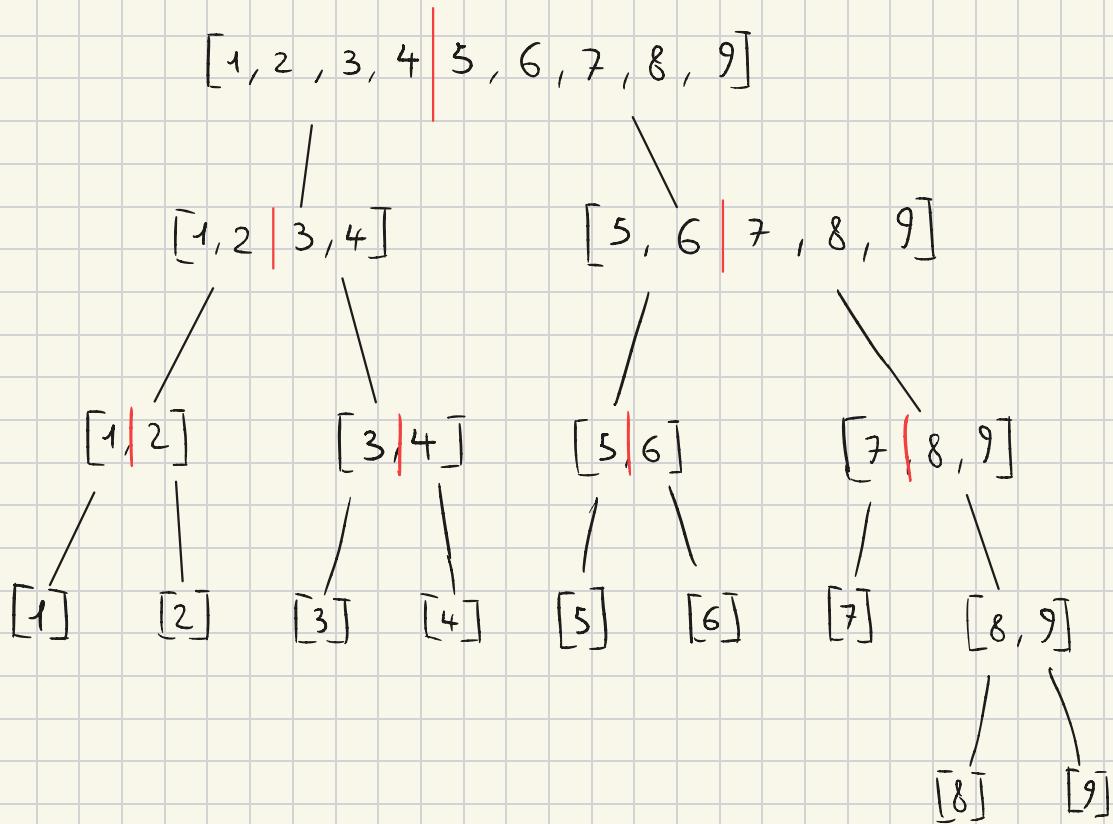




## Dichotomic Search ~ DS()



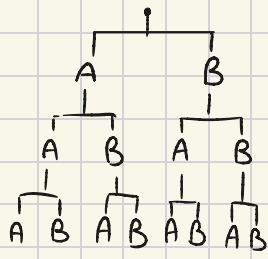
Sequenza di Fibonacci

... 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

- { 0:0, 1:1, ... }

• ottimizzazione costi :

1° g.



2° g.

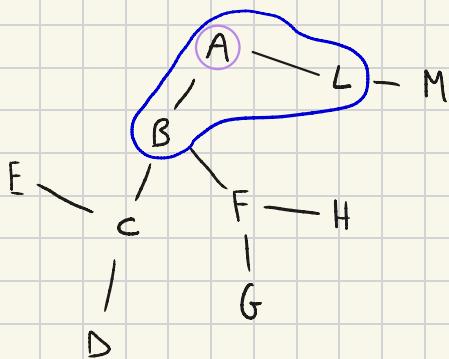
3° g.

fino al 7° g...

if giorno == 7:

else:

IF



1) "A"

2) [B, C]

3) appendo

• nella documentazione ci sono funzioni nx.dijkstra

# RICERCA DEI PERCORSI MINIMI CON NX

1)  $\text{nx.Shortest-path}(G, \text{source} = "A", \text{target} = "D", \text{Weight} = \text{"Weight"})$

restituisce IL percorso  $A \rightarrow D$  meno dispendioso in termini di Weight;  
(listo)

2)  $\text{nx.all_shortest_paths}( //, //, //, //)$

come ① ma restituisce una lista di percorsi,  
(listo) che hanno tutti lo stesso peso minimo

3)  $F_1 = \text{nx.all_pairs.all_shortest_paths}(G, \text{Weight} = \text{"Weight"}) \Rightarrow F_1["A"] \sim$  restituisce un dizionario

come ② ma lo esegue per tutte le coppie di nodi

in cui le chiavi sono i pti di arrivo (a partire da "A") e i valori sono le liste di percorsi minimi

(listo)

es:  $F_1["A"]["B"] \sim \text{nx.all_shortest_paths}(G, "A", "B", \text{Weight} = \dots) \sim$  lista di percorsi minimi  
(listo)

4)  $F_2 = \text{nx.single_source_all_shortest_paths}(G, \text{source} = "A")$

restituisce un dizionario in cui le chiavi sono i nodi di arrivo (a partire da "A") e i valori sono le liste di percorsi minimi  
(listo)

es:  $F_2["C"] \sim$  lista di percorsi minimi  $A \rightarrow C$

5)  $F_3 = \text{nx.shortest_path_length}(G, \text{source} = "A", \text{target} = "D", \text{Weight} = \dots)$

restituisce la lunghezza (somma dei pesi, se possibile) minima tra i percorsi  $A \rightarrow b$

es  $F_3 \sim 4 \sim (1+2+1)$

6)  $F_4 = \text{nx.average_shortest_path_length}(G, \text{Weight} = \dots)$

restituisce la "lunghezza media dei percorsi minimi"; (essa media contribuisce al percorso minimo di ogni coppia di nodi)

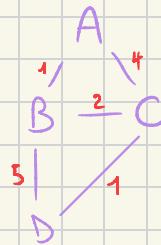
es: la media quale è il tempo minimo tra due stazioni qualsiasi?

7) `nx.has_path(G, source="A", target="B")`

Restituisce un valore booleano: TRUE se esiste almeno un percorso A->B, altrimenti False

NB: utile per evitare eccezioni

• Grafico in studio:



ALGORITMI / metodi

(sopra)  
alcune funzioni permettono di specificare  
l'algoritmo, nell'argomento

① Grafo non pesato: `nx.shortest_path(G, source, target)`

② Grafo pesato (pesi non negativi): `nx.shortest_path(G, source, target, weight, ...)`  
... method = "Dijkstra"

③ Grafo pesato (Pesi Pos. e Neg.): `nx.shortest_path(II, II, II, II, method="bellman-ford")`

④ Quando eseguiamo: `nx.floyd_Warshall(G, weight)`  
tanti percorsi  
(i minimi & coperti)

(restituisce un diz. di dizionari)

Le chiavi sono i pti di partenza

Le chiavi sono i pti di arrivo

⑤ come il ④ ma: `nx.Johnson(G, weight)`  
vale per grafici molto grandi

## Teoria sui Grafi

Grafo semplice ~ ogni nodo è collegato ad un altro mediante un solo arco

**Multigrafo** — ogni nodo può essere collegato ad un altro con più archi.

Pseudografo ~ multigrafo con loops

NB: Loop:  $\stackrel{\text{def}}{=}$  un nodo e' collegato a se stesso mediante un arco indipendente ~



- INOLTRE IL grafo presenta almeno una tra queste caratteristiche:

**orientato**  $\begin{cases} \text{unidirezionale} \sim A \rightarrow B \\ \text{bidirezionale} \sim A \leftrightarrow B \end{cases}$  [DiGraph()]

No N orientato  $\approx A \cdot -\beta$

**Network** := Rete, grafo orientato e pesato

eventualmente, se richiesto, un grafo può essere **Pesato**, cioè:

ogni nodo/arco può possedere degli attributi che salvano dei dati specifici;

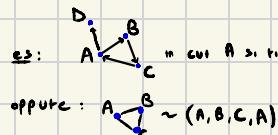
- Un grafo non orientato si dice **Connesso** se  $\forall$  coppia di vertici esiste un path che li unisce; altrimenti, si definisce il caso di un grafo composto da più sottografi detti: **Componenti connesse**; NB: un grafo "connesso" ha una sola componente connessa.
  - un grafo orientato si dice **fortemente connesso** se  $\forall (v_i, v_j)$  esiste almeno un path che li collega;

## Paths (Percorsi)

- un **Path** è rappresentato da una sequenza di vertici o di archi  $\xrightarrow{[v_1, v_2, \dots, v_n]}$  NB Path semplice se  $v_i \neq v_j$
  - La Lunghezza **Length** è il n° di archi d' un percorso ( $\sim n-1$ )

Path sintético ~ existe  $y_i = y_j$  nel setor

## Path architecture ≈ No. failures



• Un Grafo si dice **Completo** se ogni  $(v_i, v_j)$  è adiacente ( $\sim$  collegata direttamente con un arco)

- n° di archi: grafo Completo orientato  $\left[ \begin{array}{l} \text{With self loops} \sim n^2 \\ \text{No self loops} \sim n(n-1) \end{array} \right]$
- n° di archi: grafo Completo non orientato  $\left[ \begin{array}{l} \text{With self loops} \sim \frac{n(n+1)}{2} \\ \text{No self loops} \sim \frac{n(n-1)}{2} \end{array} \right]$

• Densità di un grafo  $\sim \frac{\text{n° di archi}}{\text{n° di archi possibili}}$

$\rightarrow$  ci sono undirezionali e bidirezionali

### Tree and Forest (Grafi aciclici)

Albero  $\sim$  grafo indiretto连通的  $\sim$  es:  (NB: unico comp.连通的)

Forest  $\sim$  grafo indiretto non连通的  $\sim$  es:  (NB: più componenti connesse)

NB: una foresta contiene alberi  $\sim$  trees  $\in$  forest

NB: se il grafo presenta ciclicità, non esiste la precedente distinzione

• Per studiare un albero (tree), è usuale individuare una "radice" (Root)

• La radice induce un ordine dei nodi:

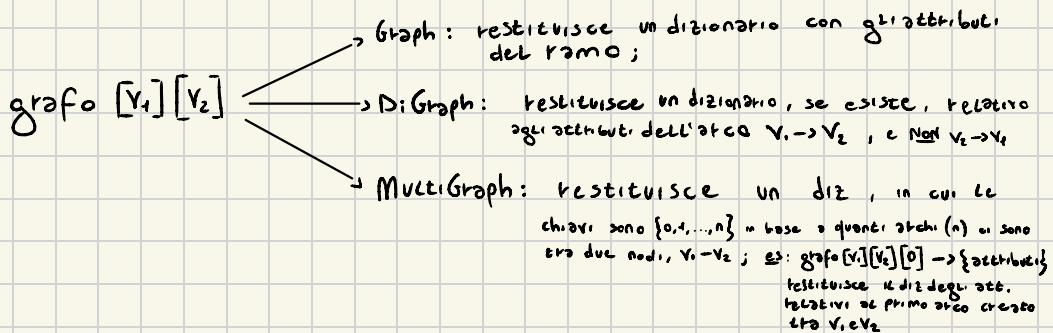
- La radice non ha parenti, è definita "antenata" di tutti i nodi
- I "figli" sono lontani dalla radice
- I "genitori" sono vicini alla radice
- I nodi più lontani:  $\stackrel{\text{def}}{=}$  foglie (leaves)

• Un albero è composto da Rami (branches)  $\sim$  edges adiacenti tra loro

• Un Grafo pesato è uno speciale "grafico etichettato" (labeled)  $\stackrel{\text{(labels)}}{\sim}$  in cui le etichette sono numeri (tendenzialmente positivi); ogni arco ha un'etichetta;

# Grafi con NetworkX

- Un Nodo può essere un qualsiasi oggetto hashable (es: int, float, str, tuple,)  
oggetti immutabili  
↳ o particolare
- N.B.: si possono creare classi DTO definendo il metodo `__hash__`, assegnando una chiave primaria della classe (se consigliato, meglio usare un id come nodo)
- Un ramo (edge) è una tupla costituita dai nodi adiacenti ~  $\text{edge} = (\text{nodo1}, \text{nodo2})$ 
  - ogni ramo può conservare dati, salvati in un dizionario, le cui chiavi indicano gli attributi (es: "Weight") del ramo;
- Diversi tipi di grafo con NX:
  - `NX.Graph()` ~ non orientato, semplice
  - `NX.DiGraph()` ~ orientato, semplice
  - `NX.MultiGraph()` ~ multigrafo (non semplice)
  - `NX.MultiDiGraph()` ~ multigrafo orientato (non semplice)
- È seconda del grafo che si utilizza:



## Funzioni utili:

- `nx.subgraph(g, lista.nodi.volti).Copy()` ~ la funzione genera una sorta di Alias, quindi aggiungo `Copy()`; si tratta del grafo g che ammette una certa insieme di vertici e archi;  
N.B.: utilizzabile per creare un grafico relativo. Locale, quindi con archi che rispettano certi requisiti;
- `nx.union(g1, g2)` ~ unisce le componenti connesse in un unico grafo che ha somma 2 componenti connesse

# Creazione di una UI

## 1) file main.py in cui :

- 1.1 si definisce la funzione di avvio, `main(page: ft.Page)`

NB: si rinomina la pagina di Fleet per semplicità

- 1.2 all'interno definisco le variabili:

`my_model = models.Model()` → classe Model dal file model.py

`my_view = UI.View(page)` → // View // // UI, scrive sulla pagina

`my_controller = UI.Controller(my_view, my_model)` → // Controller // // UI, comunica con model e controller

- 1.3 | siccome non si può collegare il controller alla view prima di inizializzarla, allora:

`my_view.set_controller(my_controller)`

- 1.4 | a questo ptò, carico l'interfaccia:

`my_view.load_interface()`

- 1.5 | infine, fuori dalla funz. `main(...)`:

chiama la funzione di Fleet che genera l'app e gli assegno il Main ~ `ft.app(target=main)`

## 2) In base a quanto scritto nel main.py :

### [cartella model]

#### Model.py



definisco la classe Model



definisco \_\_init\_\_(self)  
con gli attributi dei dati  
da salvare



definisco le varie funzioni  
che estraranno i dati  
dal database,  
chiamando le loro volte  
funzioni dal database.dao.Dao

NB: la [cartella database] contiene  
i file: `DB.connect.py`,  
`connector.cnf`, `dao.py`

#### View.py



definisco la classe View



definisco \_\_init\_\_(self, page: ft.Page)

e tra gli attributi:

• \* contenuto pagina  
- self.\_page = page  
- self.\_page.title = "..."  
- self.\_page.horizontal\_alignment = "center"  
- self.\_page.theme\_mode = ft.ThemeMode.light  
• \* definisco il controller  
- self.\_controller = None

\* eventualmente altri attributi da inizializzare (e.g. len...)

↓  
definisco `render` e `update` del controller

↓  
definisco una funzione di successo:

```
show_alert(message)
def: ft.AlertDialog(title + ft.Text(message))
self._page.open(dialog)
self._page.update()
```

infine, bisogna definire la funzione  
`load_interface(self)` che termina  
con update della page

### [cartella UI]

#### Controller.py



definisco la classe Controller



definisco  
\_\_init\_\_(self, view: View, model: Model)  
con attributi:  
- self.\_view = view  
- self.\_model = model



infine definisco TUTTI gli  
Handle che fanno interagire  
Model e View;

- eventuale cartella dto definita nella cartella model:

- metodo tradizionale:  
(esempio spedizione)

```
class Spedizione:
    def __init__(self, id, Hub-origine, Hub-attivo):
        self.id = id
        self.Hub-origine = Hub-origine
        ...
    + definizione dei metodi dunder
```

- dataclass:  
(xe002 \_\_init\_\_)

```
from dataclasses import dataclass
@dataclass
class Spedizione:
    id: int
    Hub-origine: str
    ...
    + definizione dei metodi dunder
```

- come trattare ereditarietà con metodo tradizionale:

Class Spedizione-foglia:

```
def __init__(self, attributi-padre + att. figlia)
    super().__init__(attributi padri)
    self.attributofiglia = attributofiglia
    ...
    fino agli n att. della figlia
```

## Appunti sul DTO

- oggetto che ha diverse relazioni:

es: class Impranto:

```
    id: int
    ...
    *relazioni
    consumi: List = None
```

\* definisco un metodo get che comunica con il DAO

def get\_consumi(self)

self.consumi = DAO.read\_consumi(self.id)

NB: ci possono essere degli attributi di tipo datetime, per cui:

from datetime import datetime

class PostInstagram:

id: int

date-posted: datetime

• la cosa particolare è:

oggetto.Post-1g.date-posted.Year → mette l'anno

L'attributo {day  
month  
Year}

~  
1/2/2009

- definizione: \_\_eq\_\_(self, other) → return isinstance(other, Classe) and self.id == other.id

# APPUNTI SU FLET

- 1) List ~ LV=flet.ListView(...) → LV.controls.append(flet.Text(...))
  - 2) dropdown ~ DD=flet Dropdown(...) → DD.options.append(flet.dropdown.Option(...))
    - ↓  
eventualmente:  
(key=..., text=...)
    - ↓  
e' un STR
  - 3) campo di testo ~ TXT<sub>f</sub>=flet.TextField(...) → si scrive dalla UI
- NB: DD e TXT<sub>f</sub> acquisiscono un **value**, mentre LV contiene dei controlli (g. flet.Text()) che vengono stampati nella UI
- 4) pulsanti elevati ~ pulsante=flet.ElevatedButton(**Text=...**, **on-click=...**, ...) → si cava dalla UI
    - ↓  
titolo del bottone (STR)
    - ↓  
funzione del controller

# Appunti sul DAO

- gestione errore di connessione al DB:

- se c'è un errore, Connessione DB.get\_connection() → None

- quindi intercettiamo l'errore nel DAO con un If,

```
~ conn=Connessione DB.get_connection()  
If not conn:  
    Print(...)  
    Return None  
(NB: return, di default ritorna None)
```

- controllo esecuzione della query:

- potrebbe infatti verificarsi che la query sia scritta sintatticamente, quindi:

- 1) inizializzare le variabili result, cursor e query

- 2) eseguire la query in un Try/Except:

Try:

```
cursor.execute(query)
```

```
for row in cursor:
```

```
...
```

NB: in automatico il metodo execute assegna una lista iterabile al cursor (lista estratta dal DB)

```
Except Exception as e:
```

```
Print(...)
```

```
result = None
```

- 3) gestire la chiusura del cursore e della connessione:

Finally:

```
cursor.close()
```

```
conn.close()
```

- 4) ritornare il risultato che può essere None, e quindi va gestito nel model ed eventualmente bisogna trasferire la responsabilità di stampare un messaggio nella UI al controller;

## GUIDA Pratica

- UTILIZZATE GLI ID COME NODI (NON GLI OGGETTI STESSI)
- MAPPARE GLI OGGETTI CON TUTTI GLI ATTRIBUTI ~ { id: oggetto ... }
- EFFETTUARE I CONTROLLI NEL CONTROLLER
  - DISABILITARE I TASTI DI Elevated Button NELLA VIEW (disabled=True), RIATTIVARCI IN ORDINE NEL CONTROLLER (disabled=False)
- POSSO CREARE DELLE CLASSI CHE HANNO ATTRIBUTI PRE-ASSEGNATI, PER ESEMPIO:

```
@DataClass
class Libro:
    id: int
    titolo: str
    ...
    [noleggio: bool = False] → attributo di istanza => si modifica con i metodi
    [materiale = "Carta"] → pre-assegnato
    ...
    def arriva_noleggio(self):
        self.noleggio = True
    def conclude_noleggio(self):
        self.noleggio = False
```

(independent da)

→ attributo di classe

ESISTONO ANCHE GLI ATTRIBUTI DI CLASSE (COMUNI A TUTTI GLI OGGETTI),  
T.C. SE LI MODIFICHiamo, VALE PER TUTTI GLI OGGETTI DELLA STESSA CLASSE.

↓  
~ Classe.attributo = ...

~ es: Libro.materiale = carta