

## TECNICHE DI PROGRAMMAZIONE

### GIT

REPOSITORY: insieme di tutte le versioni del mio codice, salvate localmente o globalmente.

WORKING COPY: una versione (tipicamente l'ultima versione del depository) su cui il programmatore lavora. Alla fine si cerca di far convergere la working copy alla repository.

COMMIT: operazione di modifica del depository.

UPDATE: opposto di commit, aggiorna il repository aggiungendo la nuova working copy

Nello scenario centralizzato il repository è uno ed è presente nel server, a cui accendono i client accedendo alla stessa working copy, ma questo mi espone a dei problemi di vulnerabilità. Nello scenario distribuito ogni client ha una copia del depository, che deve essere allineata grazie alle istruzioni

PUSH: do la mia versione aggiornata del repository

PULL: prendo la versione più aggiornata del repository

Sono simili a commit e update ma anziché riguardare working copy riguardano il repository

Git è un software per il controllo di versione distribuito utilizzabile da interfaccia a riga di comando

### MODULI

Non è mai una buona idea eseguire codice dentro i moduli, generalmente i moduli contengono solo definizioni (ad es di class).

Importare moduli:

**import math**: creo un namespace apposito per tutte le definizioni presenti dentro la libreria math. Per accedere alle funzioni del nuovo namespace uso la notazione col punto

*(è anche possibile usare un alias per il modulo, con import math as m)*

**From math import cos**: aggiungo cos alle funzioni presenti nel namespace corrente

**From math import \***: PERICOLOSO!!! Può andare a sovrascrivere i metodi locali, se hanno lo stesso nome

dir() usato in console mi dice tutte le funzioni che ci sono dentro la libreria

quando definisco un modulo posso capire dov'è stato runnato, perché python quando eseguo un modulo crea in automatico una variabile `__name__`, che vale "main" se eseguita dentro il modulo stesso (modalità standalone), o assume altro valore in base a dove viene eseguita.

`if __name__ == "__main__":` molto utile per printare il codice quando voglio.

Tutto quello che uso per testare deve star dentro una funzione a parte.

**LIBRERIE:** quando ti condividono un progetto, condividono nel package un file “requirements.txt” che contiene tutte le librerie e le versioni che servono per la corretta esecuzione del codice.

Flet: interfacce grafiche

Mysql

Netwotkx: per i grafi

**Dataclass** definisce i metodi minimi per vivere di un oggetto, è una keyword che fa tutto in automatico. Ci metto solo la definizione delle variabili per l'editor, affinché ci sia errore nel momento in cui sto passando un tipo di dato sbagliato alla classe

Per ogni metodo che implemento se ci aggiungo dentro una descrizione tra “” aggiungo una cosa che mi compare quando passo il mouse sopra

**ORDINARE DELLE STRUTTURE CONTENENTI OGGETTI:** prima copio la struttura poi ordino la nuova.

1- **COPIA DI LISTE** con copy() copio i riferimenti all'oggetto, quindi andando a operare su la lista copiata con copy() modifco l'oggetto originario stesso -> devo ciclare sulla lista di oggetti e riempirla con new Oggetto(o.attributo1, o.attributo2...) -> DELEGO CREANDO NEL METODO copy(), dove creo una nuova istanza di quell'oggetto, contenente gli attributi uguali a quelli dell'istanza da copiare.

2- **SORT DI OGGETTI:** in java usavamo compareTo(), mentre

In python si usa **\_\_eq\_\_**, viene chiamato in automatico quando uso ==, in. Serve per fare l'=

Uso **\_\_lt\_\_** mi serve per capire se un metodo è più piccolo di un altro, restituisce True se a < b. **\_\_lt\_\_** non ha senso se non è definito anche un **\_\_eq\_\_**. E' ciò che usa il metodo sort().

**\_\_eq\_\_** ed **\_\_lt\_\_** sono già creati per i tipi predefiniti (int, char ...)

Se devo fare due sort (alfabetico esame e numerico di voto) per un oggetto, ho un problema perché lt è solo **un** metodo.

**Strutture ordinabili:** stringhe, liste, tuple

**Non ordinabili:** dizionari (supportano **\_\_eq\_\_** ma non **\_\_lt\_\_**), set (supporta **\_\_eq\_\_** ma **\_\_lt\_\_** assume il significato di “è sottoinsieme”)

**Le dataclass non supportano **\_\_lt\_\_** e altri metodi di ordinamento, ma posso far creare gli ordinamenti di default se implemento/passo @dataclass(order=True).**

Con @dataclass(order=True) non mi viene implementato **\_\_eq\_\_** di base

## Metodo sort()

Operator è un metodo della classe standard e mi fa accedere a dei campi degli oggetti che possono essere presenti nella lista, e mi fa ordinare la lista sulla base di quelli

A sort gli dico sorta in base a key (che gli passo come funzione) per capire cosa sortare.

Se come key ci metto una funzione che ritorna una stringa, ordina la lista in base al confronto tra quelle stringhe, idem con gli interi ecc.

Questa storia non è funzionale -> uso **key = itemgetter** per LISTE E DIZIONARI, fa l'accesso con [ ]

Esiste anche per oggetti! Uso **attrgetter**, fa l'eccesso con il . (quindi \_\_eq\_\_)

SE l'ordinamento prevede il prendere in considerazione più campi (come punteggio + lode) uso la lambda function. La lambda function è una funzione che non ha un nome e mi serve solo lì

```
nuovo.voti.sort(key=lambda v: (v.punteggio, v.lode))
```

Lambda è una keyword che mi dice che quella funzione è anonima. L'ingresso della funzione è ciò che scrivo immediatamente dopo lambda, ciò che mi dà in output va dopo i: , in questo caso l'output è (v.materia, v.lode)

#ordino la tupla materia: lode, essa mi ordina prima tutti i primi campi, poi tutti i secondi e così via

!! True > False

## ----- STRUTTURE DATI -----



Some types are extremely versatile (list, dict)

Some types are "improvements" of basic types

Some types are very specialized (e.g. for parallel computation)

Operation	str	list	tuple	set	dict
Create	"abc", 'abc'	[a, b, c]	(a, b, c)	{a, b, c}	{a, b, c}; {} dict()
Create empty	""	[] list()	() tuple()	set()	{} dict()
Access i-th item	s[i]	l[1]	u[i]		d[k]
					d.get(k, default)
Modify i-th item		l[1]=x			d[k]=x
Add one item (modify value)		l.append(x)			d[k]=x
Add one item at position (modify value)		l.insert(i, x)			
Add one item (return new value)	s+'x'	l+[x]	u+(x,)		
Join two containers (modify value)		l+ll	u+ul	t.update(t1)	
Join two containers (return new value)	s+sl	x in l	x in u	t.union(t1)	t t1
Does it contain a value?	x in s			x in s	k in d (search keys)
					x in d.values() (search values)
Where is a value? (returns index)	s.find(x)	l.index(x)	u.index(x)		
	s.index(x)				
Delete an item, by index		l.pop(i)	l.pop()		d.pop(k)
Delete an item, by value		l.remove(x)		t.remove(x)	
				t.discard(x)	
Sort (modify value)		l.sort()			
Sort (return new list)	sorted(s)	sorted(l)	sorted(u)	sorted(t)	sorted(d) (keys)
					sorted(d.items())

**DIZIONARI** --- la chiave dev'essere hashable! Ossia possiede una funzione hash, ossia una funzione che preso in input un oggetto ne ritorni una rappresentazione univoca (tramite numero), come l'indirizzo di memoria in cui sono salvati. Per voto ad esempio la funzione hash può tenere in considerazione del campo voto+materia, perché sono identificati da questo. Tutti gli elementi che hanno questo tipo di identificazione possono essere chiave di un dict

<code>d[key] = value</code>	Set a new value for a key
<code>d[key]</code>	Retrieve value from the key. May raise <code>KeyError</code>
<code>d.clear()</code>	Clears a dictionary.
<code>d.get(key, default)</code>	Returns the value for a key if it exists in the dictionary. Otherwise, returns a default value
<code>d.items()</code>	Returns a list of key-value pairs in a dictionary.
<code>d.keys()</code>	Returns a list of keys in a dictionary.
<code>d.values()</code>	Returns a list of values in a dictionary.
<code>d.pop(key, default)</code>	Removes a key from a dictionary, if it is present, and returns its value. Otherwise, returns a default value
<code>d.popitem()</code>	Removes the last key-value pair from a dictionary.
<code>d.update(obj)</code>	Merges a dictionary with another dictionary

MappingProxyType, restituisce un dizionario read-only, che non può essere modificato perché gli ho tolto tutti i metodi per modificarlo.

**RECORDS**: collezioni di tipi di dati eterogenei, sono simili alle classi (senza dei metodi). Possono essere implementati tramite tuple, ma più propriamente sono delle classi o delle dataclass.

Strutture particolari di record

`collections.namedtuple`, `typing.namedtuple` (non più molto usati da quanto py ha implementato le classi)

**SETS**: contenitore mutabile (ossia è mappabile tramite una funzione di hash) e modifica i campi che servono nella funzione di hash. La versione immutabile di un set è il `frozenset` (tipicamente lo si crea a partire da un set), può essere una struttura annidata, quindi può essere chiave di un dict, dato che il frozenset è hashable.

`collections.Counter` è un set che conta il numero di volte in cui un elemento è presente nel set

```
cnt = collections.Counter([1, 2, 3, 3, 4, 5, 1, 8, 3, 5, 2,
2, 3, 8])
Counter({3: 4, 2: 3, 1: 2, 5: 2, 8: 2, 4: 1})
```

Può essere fatta a partire da una stringa, da un dizionario, ecc. UTILE PER FARE STATISTICHE, tipo trovare l'oggetto più comune.

## What can I do with a Counter?

<code>c.most_common(n)</code>	# the 'n' (default: all) most common items
<code>c.total()</code>	# total of all counts
<code>list(c)</code>	# list unique elements
<code>set(c)</code>	# convert to a set
<code>dict(c)</code>	# convert to a regular dictionary
<code>c.items()</code>	# convert to a list of (elem, cnt) pairs
<code>c.elements()</code>	# return a list [elem, ...] with repetitions
<code>Counter(dict(list_of_pairs))</code>	# convert from a list of (elem, cnt) pairs
<code>c.most_common()[:-n-1:-1]</code>	# n least common elements
<code>+c</code>	# remove zero and negative counts
<code>c.clear()</code>	# reset all counts

**STACK/PILE:** come nelle operazioni matematiche, si parte dalla parentesi più interna (che è l'ultima operazione istanziata) e poi di va via via verso la più ‘anziana’. Filosofia LIFO.

**Implementazione:** Uso LISTE, aggiungo append(), tolgo con pop(ultimo indice)

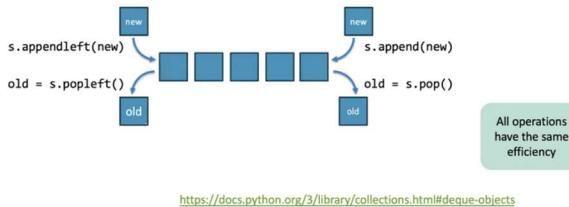
**CODE:** lista in cui vengono aggiunti degli elementi da dx verso sx, e per ‘smaltire gli elementi’ li prendo da sx. Filosofia FIFO.

**Implementazione:** Uso LISTE, Appendo all’ultimo dall’ultimo, pop(0) per togliere -> **PROBLEMI DI EFFICIENZA!**

S = `collections.deque()` ho una coda bidirezionale che mi fa aggiungere e togliere da dx e sx con la stessa efficienza. Lo si può usare anche con le pile

deque: double-ended queue

```
s = collections.deque()
```



## Other deque methods

<code>d = deque()</code>	New empty deque
<code>d = deque(iterable)</code>	Deque from list
<code>d = deque(maxlen=N)</code>	Hosts max N elements, discards older ones if more are added
<code>d.extend(iterable)</code>	Adds list of elements at end
<code>d.extendleft(iterable)</code>	Adds list of elements at beginning
<code>d.rotate(n)</code>	Rotate elements by n steps
<code>d[i]</code>	Access element (slower than lists)
<code>d.index(x), d.insert(i, x), d.remove(x), d.reverse()</code>	Same as lists

!! se l'applicazione mi chiede di accedere dentro la struttura per vari motivi fallo con le liste, non con il deque(), mentre se ti chiede di aggiungere/togliere elementi da dx o sx si usa il deque

**CODE PRIORITARIE:** estraggo sempre l’elemento con priorità massima. Gli elementi vengono aggiunti in qualunque ordine, ma esce sempre quello con priorità max (ossia quello, generalmente, che ha un valore minore). La priorità è stabilita da un sorting order, che può essere stabilito in due modi per gli oggetti:

- si crea la tupla (priorità, oggetto) e creo `__lt__` della tupla, date due tuple il confronto parte dal primo elemento, in questo caso la priorità, e poi a parità di priorità vado a confrontare l'oggetto, in realtà confronto il riferimento all'oggetto, ma tanto se hanno la stessa priorità a me programmatore non cambia niente chi passa prima
- ci si basa sul metodo `__lt__` dell'oggetto

### implementazione code prioritarie:

#### Priority queues in Python

Items `x` must be comparable  
(implement `__lt__`)

##### `heapq` – uses plain lists

- `h = []`
- `h = heapify(iterable)`
- `len(h)`
- `len(h)==0`
- `heapq.heappush(h, x)`
- `x = heapq.heappop(h)`

##### `queue.PriorityQueue`

- `q = queue.PriorityQueue()`
- `q.qsize()`
- `q.empty()`
- `q.full()`
- `q.put(x)`
- `x = q.get_nowait()`

#### 1- `Heapq`:

Heapify, a partire da un iterable, crea una struttura gerarchica ad albero la cui radice è il valore più piccolo, man mano che trovo dei valori più grandi li vado a mettere alla fine. Heappush: metto l'elemento nella posizione corretta di quella struttura gerarchica, heappop toglie la cima di quest'albero. Instanzio delle funzioni.

#### 2- `Queue.PriorityQueue`:

Put/get\_nowait() aggiungo/tolgo elementi. E' getnowait anziché get e basta perché è pensata perché get() mi si pianta se la coda prioritaria è vuota.

## INTERFACCE GRAFICHE – 17/03/2025

utili per essere user-friendly, visualizzare meglio i dati.

Come si fanno? Tramite le librerie:

- 3- Tk: non bellissima un po' obsoleta
- 4- Flutter: da google
- 5- Flet: libreria semplice di py, che si basa su flet, permette di creare applicazioni web e desktop. Semplice da utilizzare e con interfacce capire.

L'applicazione sta tutta dentro una funzione, la funzione main, che prende un solo argomento. Questa carica tutti gli elementi di

```
● ● ●  
1 import flet as ft  
2  
3 def main(page: ft.Page):  
4     # add/update controls on Page  
5     pass  
6  
7 ft.app(target=main)
```

Con view, definiamo un 'interfaccia per la pagina

```
● ● ●  
1 import flet as ft  
2  
3 def main(page: ft.Page):  
4     # add/update controls on Page  
5     pass  
6  
7 ft.app(target=main)
```

```
● ● ●  
1 import flet as ft  
2  
3 def main(page: ft.Page):  
4     # add/update controls on Page  
5     pass  
6  
7 ft.app(target=main, view=ft.AppView.WEB_BROWSER)
```

così

creo una pagina per web browser

OGNI VOLTA CHE MODIFichi UN ELEMENTO DEVI FARE PAGE.UPDATE() !!!!!!!

Sarebbe utile disaccoppiare le funzionalità del programma: si utilizza il pattern MVC

M : model, è il solo responsabile di leggere dati e fare operazioni con quelli. Gestisce veramente la logica del programma. Tipicamente ritorna dei numeri che verranno visti dalla view. Si occupa solo di dati

V : view , classe che crea e visualizza una finestra/interfaccia e fa gli opportuni controlli. Si occupa solo di grafica. Contiene al suo interno una serie di istanze di controlli

VIEW E MODEL NON SI PARLANO MAI DIRETTAMENTE! Le loro interazioni avvengono sempre tramite il control

C : control, contiene le istruzioni che modificano l'interfaccia grafica a partire da operazioni svolte nel modello. L'unico che conosce la logica del programma, conosce il modello e ne richiama opportunamente le funzioni, parla con entrambe le classi. E' una serie di handleQualcosa.

QUANDO NEL CONTROLLER HAI DEI METODI INERENTI AL BOTTORE RICORDATI DI PASSARE COME PARAMETRO L'EVENTO e !!!

## DATABASE IN PYTHON – 24/03/2025

Connessione: istruzione con cui vogliamo connetterci al database.

### Connection

```
import mysql.connector  
  
cnx = mysql.connector.connect(user='scott',  
                               password='password',  
                               host='127.0.0.1',  
                               database='employees')  
  
cnx.close()
```

is a **MySQLConnection** object.

The **MySQLConnection** class is used to open and manage a connection to a MySQL server. It's also used to send commands and SQL statements and read the results.

The arguments of the **connect()** identify the database we want to connect, and the credentials of the user

There are many more arguments...

<https://dev.mysql.com/doc/connector-python/en/connector-python-connectargs.html>

### **RICORDA SEMPRE DI DISCONNETTERTI AL DATABASE UNA VOLTA CHE HAI FINITO DI LAVORARCI SU, ALTRIMENTI IL COMPUTER VA A SCHIFIO!!!!**

L'istruzione connect va messa in un blocco try perché non sempre ha esito positivo, e questo avviene tipicamente in tre situazioni:

1. Credenziali sbagliate
2. Server spento
3. Bgdfgm

Scrivere connect con una lista di parametri mi fa sbandierare a tutti nel codice i miei dati privati, per cui si crea e richiama nel codice configFile in questo modo:

- It is possible to use a separate config file.

```
cnx = mysql.connector.connect(option_files='/etc/mysql/connectors.cnf')
```

Example config file

```
[client]  
user=John  
password=Wick  
host=127.0.0.1  
database=tests  
raise_on_warnings=True
```

Cursore: oggetto che usiamo per selezionare i dati, funziona come un puntatore. E' un iterable che punta ad un indirizzo di memoria e lo scorre man mano. Si cicla tramite esso sul file.

```
import mysql.connector  
  
cnx = mysql.connector.connect(database='world')  
cursor = cnx.cursor()  
cursor_dict = cnx.cursor(dictionary=True)  
cursor_tuple = cnx.cursor(named_tuple=True)  
cursor_prepared = cnx.cursor(prepared=True)
```

**MySQLCursorDict** cursor returns rows as dictionaries  
**MySQLCursorNamedTuple** cursor returns rows as named tuples  
**MySQLCursorPrepared** cursor is used for executing prepared statements

Creo una connessione, poi un cursore a partire da quella connessione.

Cursore chiamato senza argomenti -> ritorna una dati in una tupla

Se voglio mettere i dati in una struttura diversa, ci passo come parametro [XStruttura] = True

Query: stringa da eseguire che ci permette di recuperare dati da un db

```
query = """Select id, name from user"""  
cursor.execute(query)
```

```
import mysql.connector  
  
cnx = mysql.connector.connect(user='John',  
                               password='Wick',  
                               host='127.0.0.1',  
                               database='tests')  
  
cursor = cnx.cursor()  
  
add_test = """INSERT INTO tests  
(id, name, value)  
VALUES (%s, %s, %s)"""  
  
cursor.execute(add_test, (1, "John Doe", 11.3))  
  
cnx.commit()  
cursor.close()  
cnx.close()
```

# Insert/Update/Delete data

- Using the cursor, we can execute **INSERT**, **UPDATE** and **DELETE** statements

```
import mysql.connector

cnx = mysql.connector.connect(user='John',
                               password='Wick',
                               host='127.0.0.1',
                               database='tests')

cursor = cnx.cursor()

update_test = """UPDATE tests
                  SET valutazione = %s
                  WHERE id = %s"""

cursor.execute(update_test, (99.9, 3))

cnx.commit()
cursor.close()
cnx.close()
```

```
import mysql.connector

cnx = mysql.connector.connect(user='John',
                               password='Wick',
                               host='127.0.0.1',
                               database='tests')

cursor = cnx.cursor()

delete_test = """DELETE FROM tests
                  WHERE id = %s"""

cursor.execute(delete_test, (5,))

cnx.commit()
cursor.close()
cnx.close()
```

32

chiede sempre in input una tupla, ci metto la , !!!!!!!

Fetchone -> ritorna una riga da cursor

Fetchmany(N)

Fetchall()

IL CURSOR VA SOLO ANANTI NON ANCHE INDIETRO per cui fetchall, richiamato dopo fetchone, mi da tutte le righe RIMANENTI della tabella al di fuori della prima

## Example

- 0. cursor.execute(query)
- 1. cursor.fetchone()
- 2. cursor.fetchall()

Cursor before the first row			
100	S N Rao	5500.50	1 <sup>st</sup> Record
101	Jyostna	6500.50	2 <sup>nd</sup> Record
102	Jyothi	7550.50	3 <sup>rd</sup> Record

Cursor at the first row

Cursor at the first row			
100	S N Rao	5500.50	1 <sup>st</sup> Record
101	Jyostna	6500.50	2 <sup>nd</sup> Record
102	Jyothi	7550.50	3 <sup>rd</sup> Record

Cursor after the last row			
100	S N Rao	5500.50	1 <sup>st</sup> Record
101	Jyostna	6500.50	2 <sup>nd</sup> Record
102	Jyothi	7550.50	3 <sup>rd</sup> Record

1. Creo connessione
2. Creo cursore a partire dalla connessione
3. Creo query da eseguire
4. Opero sui dati

La conversione di tipi da SQL a python avviene in automatico, anche se volendo la potrei inibire

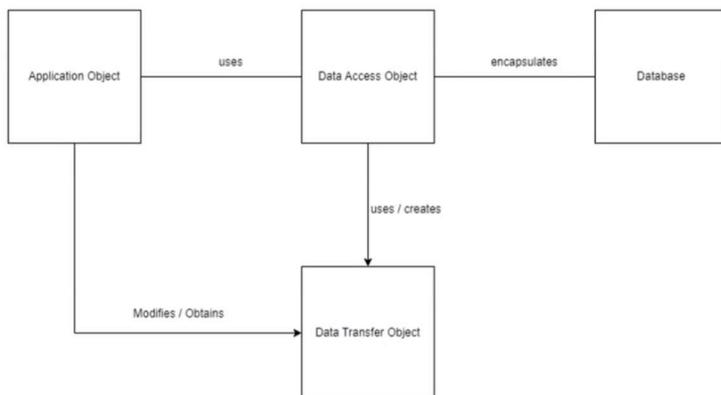
**Pattern DAO** (Data Access Oprgfjknd) : definisce le regole da seguire quando ci interfacciamo ad un database. Lavora sull'astrazione tra database e applicazione, deve avere tutti i metodi che aggiungono, modificano, leggono dati, senza che il db sappia come questo viene fatto, per rendere la mia applicazione versatile con tutte le tipologie di db.

## Data Access Object (DAO) – 1/2

- «**Client**» classes:
  - Application code that needs to access the database
  - Ignorant of database details (connection, queries, schema, ...)
- «**DAO**» classes:
  - Encapsulate all database access code ([mysql-connector-python](#))
  - The only ones that will ever contact the database
  - Ignorant of the goal of the Client

Il dao serve per interrogare il db, senza sapere a quale scopo, il client non sa di starsi interfacciando con un db, non contiene assolutamente query, per lui potrebbe tranquillamente star prendenddo le info da una lista

### DAO Diagram



DTO è un acronimo inglese che sta per "Data Transfer Object". In informatica, è un oggetto che definisce come inviare dati in rete.

Il dao è specifico per il DB, IL DAO NON HA STATO, quindi non possiede variabili!

per ogni tabella ho una dataclass. E offre diversi metodi per fare ricerche nel db

Se ho più DB devo fare piu dao.

Ogni metodo del dao crea una connessione al db. Siccome non ha senso avere un codice ripetuto creo una classe DBConnect da richiamare all'inizio di ogni metodo del dao, e che mi inizializza un attributo che

chiamo `self.dbConnect`. Inoltre, ogni classe del dao ha inizializzato un cursore e una chiusura alla connessione.

Se una query mi modifica il db, devo fare il commmit() per salvare le modifiche! E inoltre, non posso salvare sul db delle info già presenti, quindi conviene creare un metodo hasVoto() per effettuare un controllo nel metodo addVoto(), e qualora questo mi ritorna False aggiungere il voto nel db.

```
class LibrettoDAO:
    # def __init__(self):
    #     self.dbConnect = DBConnect()

    2 usages
    def getAllVoti(self):
        # cnx = mysql.connector.connect(
        #     user = "root",
        #     password = "rootroot",
        #     host = "127.0.0.1",
        #     database = "libretto")
        cnx = DBConnect.getConnection()

        cursor = cnx.cursor(dictionary=True)
        query = """select * from voti"""
        cursor.execute(query)
```

Prima di DBConnect di devo mettere  
`@classmethod` in modo che la classe sia unica  
E non sia instanziata più volte. Quindi,  
anziche fare un `self.dbConnect = DBConnet()`  
uso il metodo direttamente dalla classe, come  
`cnx = DBConnect.getConnection()`

```
def addVoto(self, voto: Voto):
    cnx = mysql.connector.connect(
        user = "root",
        password = "rootroot",
        host = "127.0.0.1",
        database = "libretto")

    cursor = cnx.cursor()

    query = ("insert into "
             "voti (materia, punteggio, data, lode) "
             "values (%s, %s, %s, %s)")

    cursor.execute(query, params= (voto.materia, voto.punteggio, voto.data, voto.lode))
    cnx.commit()
```

```
def hasVoto(self, voto: Voto):
    cnx = mysql.connector.connect(
        user = "root",
        password = "rootroot",
        host = "127.0.0.1",
        database = "libretto")
    cursor = cnx.cursor()
    query = """
        select *
        from voti v
        where v.materia = "Pozioni"
        and v.punteggio = 30
```

**ORM (Object Relational Mapping)** è un pattern che ci dice come trasferire dati tra database e il nostro programma, andando a mappare il database in oggetti. Per ogni tabella devo avere una dataclass DTO associata che mi legga le info. Si usa come nome quello della classe, ma al singolare.

Creo una dataclass per ogni tabella, poi mappo le relazioni all'interno delle classi (mappa solo quelle che ti servono, non tutte!). La classe deve avere **un attributo per ogni classe che mi serve**, (sennò all'esame perdi tempo). Più aggiungere getter (@property) e un setter (@attr.setter).

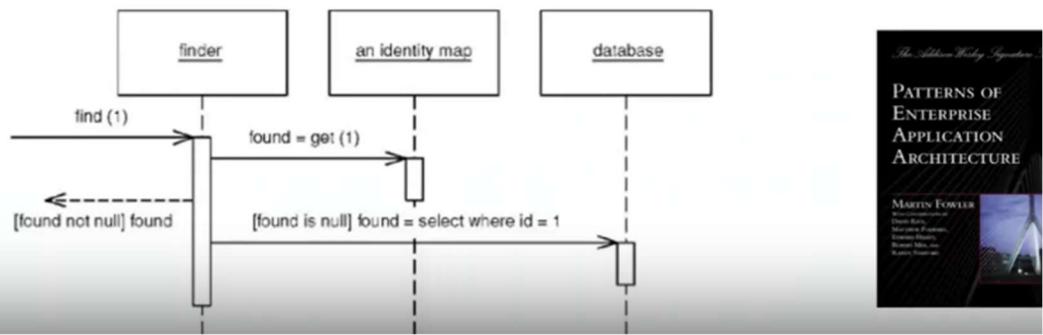
Devo anche definire un `__eq__` e un `__hash__` che confrontano o richiamano hash in base alla chiave primaria.

Chiavi esterne: per mappare la relazioni.

- Relazioni 1:1 metto in entrambe le tab la chiave primaria dell'altra.
- Relazioni 1:N metto da entrambe le parti una classe che mi istanzi l'altra, sia nella parte 1: sia nella parte N:
- Relazioni N:N creo da entrambe le parti due set di oggetti che mappano elementi della tabella collegata. In realtà nel set posso metterci sia solo la chiave, sia l'intero oggetto. Se ci metto tutto l'oggetto nell'interrogare il db devo usare join varie.  
Noi creiamo un oggetto vuoto, che poi inizializzo solo quando mi serve

# Identity Map pattern

- Ensures that each object gets loaded only once, by keeping every loaded object in a map
- Looks up objects using the map when referring to them.



Hai già salvato nella mappa il fatto? Si -> la cerco

No -> creo una identity map

POOLING

Trovare risalire all'oggetto da un dropdown

```
for cod in self._model.getAllCorsi():
    self._view.ddCodins.options.append(ft.dropdown.Option(key=cod.codins,
                                                          data=cod,
                                                          on_click=self._choiceDDCodins))

#nel dd vedo il dd, mi appare la rappresentazione a stringa di cod, ma quando lo clicco mi prende l'oggetto
#(data), e quando lo clicco per salvare il fatto lo faccio con choiceDDCodins

def _choiceDDCodins(self,e):
    new *
    self._ddCodinsValue = e.control.data
    print(self._ddCodinsValue) #qui ho l'oggetto che ho selezionato
```

```
•SELECT c.codins, c.crediti, c.nome, c.pd, count(*)
FROM corso c, iscrizione i
where c.codins = i.codins
and c.pd = 1
group by c.codins, c.crediti, c.nome, c.pd
```

questa query serve per contare gli iscritti a un corso (conto le righe diverse presenti nelle 'sottotabelle' che ho creato con group by). Tutto quello che c'è nel select deve andare nel group by. NELLA GROUP

BY NON CI VA CIO' CHE DEVI CONTARE, PERCHE COUNT(\*) CONTA LE RIGHE DELLE SOTTOTABELLE BLABLA.  
LASCIA ALEMNO UN GRADO DI LIBERTA QUANDO FAI GROUP BY PER CONTARE!!!!

```
def handlePrintCorsiPD(self, e):
    self._view.lvTxtOut.controls.clear()
    pd = self._view.ddPD.value
    if pd is None:
        # self._view.lvTxtOut.controls.append(
        #     ft.Text("Attenzione, selezionare un periodo didattico!", color="red"))
        self._view.create_alert("Attenzione, selezionare un periodo didattico!")
        self._view.update_page()
        return
    corsiPD = self._model.getAllCorsi(pd)

    self._view.controls.append(ft.Text(f"Corsi de |"))

1 usage (1 dynamic)
def handlePrintIscrittiCorsiPD(self, e):
```

07/04/2025 RICORSIONE

**Una funzione ricorsiva è una funzione che richiama se stessa nella definizione.**

La sua utilità sta nel suddividere un problema in sottoproblemi, che a loro volta verranno suddivisi, al fine di arrivare a dei problemi così semplici la cui soluzione è ovvia.

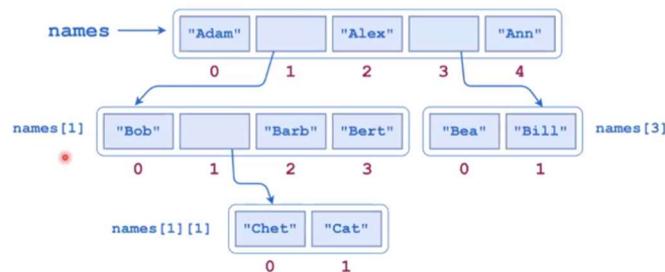
Va sempre implementato un if-else in cui definisco una condizione terminale!!

Si puo anche definire una struttura dati che utilizza la ricorsione, in modo da costruire strutture dati un pezzetto la volta

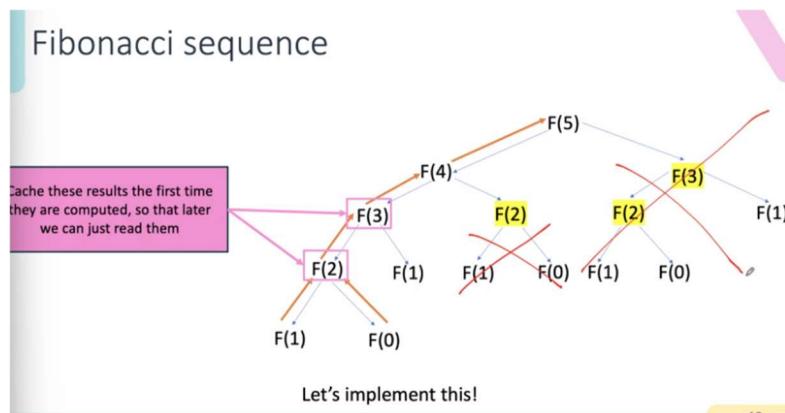
```
def attach_head(elements, input_list):
    if no more elements:
        return input_list
    else:
        return attach_head(elements[:-1],
[elements[-1]] + input_list)
```

Liste annidate: liste i cui elementi sono a loro volta una lista. Se ne volessi contare gli elementi (e sottoelementi) complessivi, è conveniente utilizzare un metodo ricorsivo

```
names = ['Adam', ['Bob', ['Chet', 'Cat'], 'Barb', 'Bert'], 'Alex', ['Bea', 'Bill'], 'Ann']
```



La ricorsione mi permette di avere degli effettivi vantaggi di efficienza, dal momento che, facendo caching dei risultati, posso eliminare interi rami di esplorazione.



Quanto ho tanti sottoproblemi con gli stessi argomenti, può essere utile fare caching usando la tecnica della Memoization. Il caching è fatto automaticamente grazie a delle utilities di python, come `@lru_cache`

`@lru_cache` is a decorator that wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls.

Available since Python 3.2

It uses a dictionary behind the scenes:

- Key: the call to the function, including the supplied arguments
- Value: the function's result
- The function arguments have to be **hashable** for the decorator to work.

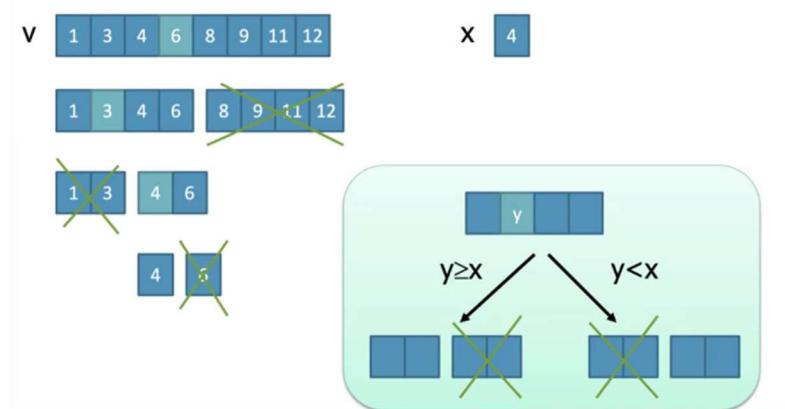
ha senso usarla quando i vari argomenti della funzione devono essere **hashable**, se non lo sono li posso incapsulare in un'altra struttura che sia hashable e utilizzare ugualmente questo decoratore.

**Quicksort:** scelgo arbitrariamente dei pivot, e poi metto alla sua sx nella struttura sortata gli elementi piu piccoli di quello, a dx quelli piu grandi (per quello crescente)

- 1- Scelgo pivot
- 2- Dividere sequenza in sottosequenze

**Dichotomic search:**

### Example: dichotomic search



# Code Outline

```

def recursion(..., level):
    // E - instructions that should be always executed (rarely needed)
    do_always(...)

    // A
    if terminal_condition:
        do_something(...)
        return ...

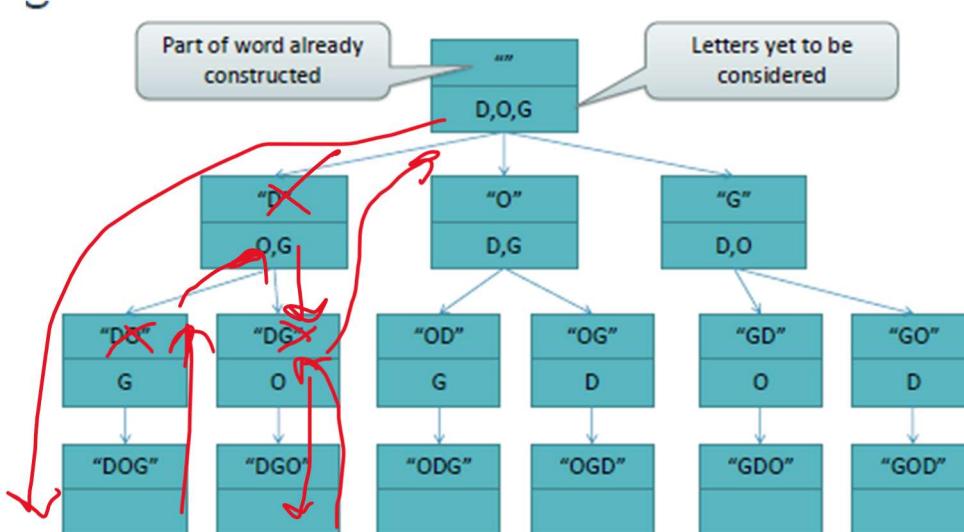
    for ... //a loop, if needed
        //B
        compute_partial()

        if filtro: //C
            recursion(..., level+1)

    //D
    back_tracking

```

In alcuni casi devo togliere alcuni pezzi di soluzione, e questo si dice **backtracking**



## RISOLUZIONE LAB:

```

2 usages new *
def _ricorsione(self, parziale, lista_situazioni):
    #condizione_terminale
    if len(parziale)==15:
        print(parziale)

    #condizione_ricorsiva
    else:
        #cercare_le_città_per_il_giorno_che_mi_serve
        candidates = self.trova_possibili_step(parziale, lista_situazioni)
        # provo ad aggiungere self._ricorsione(parziale, lista_situazioni)
        for candidate in candidates:
            parziale.append(candidate)
            self._ricorsione(parziale, lista_situazioni)
            parziale.pop()

```

```

def calcola_sequenza(self, mese):
    situazioni = MeteoDao.get_situazioni_meta_mese(mese)
    self._ricorsione([], situazioni)

1 usage new *
def trova_possibili_step(self, parziale, lista_situazioni):
    giorno = len(parziale)+1
    candidati = []
    for situazione in lista_situazioni:
        if situazione.data.day == giorno:
            candidati.append(situazione)
    return candidati

```

- 1) Enumera tutte le soluzioni. Cerca di farlo in modo efficiente! Puoi modularizzare il codice con funzioni efficienti, come *trova\_possibili\_step*, che anziché farti ciclare su tutto ti restringono il campo di ricerca per il candidato elemento da aggiungere alla sequenza parziale della ricorsione.
  - 2) metti i vincoli
  - 3)
- )

## -LEZIONE 28/04/2025-----

I **GRAFI** : in un grafo i nodi saranno le dataclass i cui dati li leggo dal database

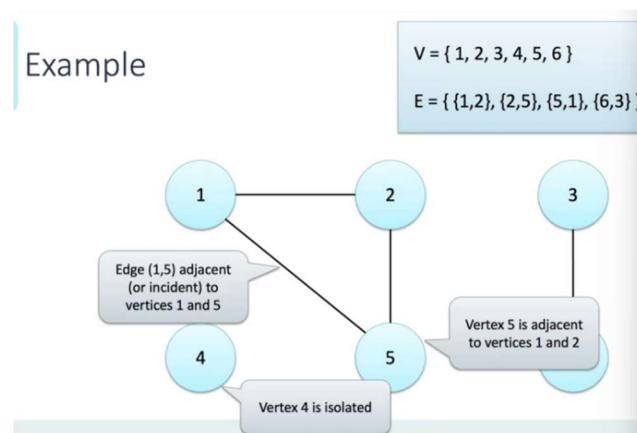
I **VERTICI**: li costruirò su determinate info

I grafi possono essere:

- Graph: un arco tra due nodi
- Multigraph: più archi tra due nodi
- Loops: un graph ha dei nodi che si riconnettono a sé stesso
- Psudograph: multigraph con loop
- DiGraph: gli archi hanno una direzione

I grafi diretti sono rappresentabili in una tupla nodi-archi con la tupla ordinata (nodo partenza-nodo di arrivo)

Nei grafi indiretti la tupla non è ordinata. Può essere anche rappresentata con il nodo di adiacenza



**Grafo connesso**: dati due nodi qualsiasi c'è sempre un percorso che mi faccia arrivare da uno all'altro.

Non è importante che ci sia un arco diretto tra tutti i nodi,

Un grafo non connesso può essere spezzettato in più grafi connessi

Componeti connessi: sottografo connesso di dimensione massima

Nei grafi con archi orientati si parla di fortemente connesso se data ogni coppia di archi c'è un percorso che li congiunge, altrimenti si parla di debolmente connesso.

	Directed	Undirected
No self loops	$n(n - 1)$	$\frac{n(n - 1)}{2}$
With self loops	$n^2$	$\frac{n(n + 1)}{2}$

**Ciclo**: il nodo iniziale e finale sono lo stesso, se un grafo non contiene cicli è detto aciclico

**Foresta:** insieme di alberi (non per forza connessi)

**Albero:** grafo aciclico connesso

In un albero può essere riconosciuto un nodo che è il predecessore di tutti gli altri nodi, non ha genitori

**Densità del grafo:** rapporto del numero di archi che ho e il numero di archi che avrebbe il grafo completo associato

$$d = \frac{|E(G)|}{|E(K_{|V(G)|})|}$$

CODICE PER CREAZIONE GRAFO:

```
● ○ ●  
1 import networkx as nx  
2 g = nx.Graph()  
3 g.add_edge("a","b",weight=1)  
4 g.add_edge("b","c",weight=100)  
5 g.add_edge("a","c",weight=1)  
6 g.add_edge("c","d",weight=1)  
7 print(nx.shortest_path(g,"b","d"))  
8 print(nx.dijkstra_path(g, "b", "d", weight='weight'))
```

Grafo = DiGraph()      è orientato!

Grafo = Graph()      non è orientato

- NetworkX supports many different graph types, like:
  - `nx.Graph()` – undirected
  - `nx.DiGraph()` – directed
  - `nx.MultiGraph()` – supports multiple edges between nodes
  - `nx.MultiDiGraph()` – directed multigraph
- Also provide implementation of notable graphs (like heawood)

```
● ● ●  
1 import networkx as nx  
2 import math  
3 import flet as ft  
4  
5 g = nx.heawood_graph()  
6 print(g.nodes, g.edges)  
7  
8 g.add_node(math.cos) # cosine functionx  
9 g.add_node(ft.Text("Pippo"))  
10 g.add_edge("math.cos", 3)  
11 print(g.nodes, g.edges)
```

Grafo.add\_node()      per aggiungere i nodi

Grafo.add\_edge(nodo partenza, nodo arrivo)

----- per mettere nodi tutti insieme (aggiungendoli da una lista)9

```
nbunch = [1, 2, 3, 4, 5, 6, 7, 8, 9]
grafo2.add_nodes_from(nbunch)
print(grafo2.nodes)
```

Se provo ad aggiunge due volte l'oggetto 1, avendo la stessa hash function mi viene sovrascritto tutto

Idem per gli archi

```
ebunch = [(4, 6), (8, 1), (2, 9)]
grafo2.add_edges_from(ebunch)
```

Sorgente di bachi ma molto utile: se utilizzassi nell'inserire gli archi un arco che punta a un nodo che non esiste me lo crea dinamicamente (potrei dichiarare solo gli archi, senza fare i nodi già)

Non generale con id dei nodi non con nodi stessi!!!!

UN grafo puo essere visto con 3 dic t annidati, le cui chiavi sono sempre i nodi:

- 1) Vedo i vicini, chiavi tutti i nodi connessi a n, i valori sono i parametri di quell'arco
- 2) Trovo gli attributi di quell'arco lì

g[u][v] yields the edge attributes  
n in g tests if node n is in g  
for n in g: iterates through the graph  
for nbr in g[n]: iterates through the neighbors of n  
Data struct for direct graphs is only slightly more complex (two dics, one for successors and one for predecessors)  
You can also use g.nodes() and g.edges() to get corresponding data  
Edges can have arbitrary attributes

```
import networkx as nx

g = nx.Graph()
g.add_edge(1, 2) # default edge data=1
g.add_edge(2, 3, weight=0.9) # specify edge data

elist = [(1, 2, 1), (2, 3, 1), (1, 4, 1), (4, 2, 1),
          ('a', 'b', 5.0), ('b', 'c', 3.0), ('a', 'c', 1.0), ('c', 'd', 7.3)]
g.add_weighted_edges_from(elist)
g.add_edge(2,5,arbitraryAttr = "foo")

print(g[2])
print("-----")
print(g['a'][‘b’])
print("-----")
print('e' in g)
print("-----")
for n in g:
    print (n)
print("-----")
for nbr in g[2]:
    print(nbr)
print("-----")
print(g[2][5][‘arbitraryAttr’])
```

Equivale a for n in grafo.nodes()

## Classic graph operations

- `subgraph(G, nbunch)` - induce subgraph of G on nodes in nbunch
- `union(G1, G2)` - graph union
- `disjoint_union(G1, G2)` - graph union assuming all nodes are different
- `compose(G1, G2)` - combine graphs identifying nodes common to both
- `complement(G)` - graph complement
- `create_empty_copy(G)` - return an empty copy of the same graph class
- `convert_to_undirected(G)` - return an undirected representation of G
- `convert_to_directed(G)` - return a directed representation of G

ESEMPIO GRAFO FATTO A LEZIONE:

- Creo grafo
- Aggiungo oggetti di tipo DATACLASS letti da db come nodi
- Aggiungo archi.

Nell'esempio in questione parliamo di metropolitane, per cui un arco di aggiunge se esiste un arco tra u e v.

MA questo mi richiede due cicli for -> INEFFICIENTE!!!!!!

→ Bisogna fare una query che, dato che in una metropolitana non ci sono stazioni scollegate in assoluto (non avrebbe senso averne), sfrutto la caratteristica di adiacenza tra nodi di un arco e definisco il metodo **getVicini. IN PIU' CICLO SOLO IL NECESSARIO**, perché con due for eseguo anche query che non ritornano risultati

```
@staticmethod
def getVicini(u: Fermata):
    conn = DBConnect.get_connection()

    result = []

    cursor = conn.cursor(dictionary=True)
    query = """SELECT *
                FROM connessione_c...
                where c.id_stazP = %s"""

    cursor.execute(query, (u.id_fermata,))

    for row in cursor:
        result.append(Connessione(**row))
    cursor.close()
    conn.close()
    return len(result) > 0
```

- ➔ Posso fare ancora meglio definendo getAllEdges in cui mi prendo tutto quanto presente nel database e mi gestisco tutto nel database

```
def addEdges3(self):
    """
    faccio una query unica che prende tutti gli archi, e poi ciclo qui.
    Returns:
    """

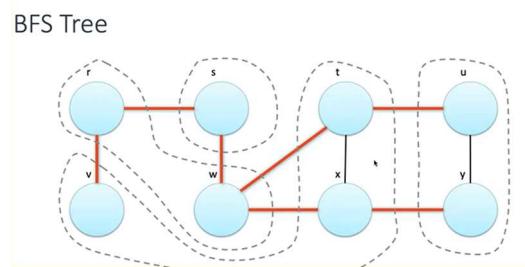
    allEdges = DAO.getAllEdges()
    for edge in allEdges:
        u = self._idMapFermate[edge.id_stazP]
        v = self._idMapFermate[edge.id_stazA]
        self._grafo.add_edge(u,v)
```

## LEZ 30/04/2025-----VISITE SU GRAFO

Un grafo, una volta selezionato un nodo di partenza, puo essere visitato:

- **In ampiezza (BFS):** ossia visitando da un nodo tutti i nodi vicini, e scegliendone uno tra quelli. E' una ricerca a livelli, e ricorsiva nella quale da un nodo, visito tutti i vicini (presenti in un set) finchè non gli ho già esplorati tutti. A questo punto mi sposto di nodo e riprendo da capo. Ottengo una struttura ad albero in cui l'origine è il nodo source, i vertici sono quelli che sono presenti nella componente连通的, mentre gli archi sono un sottinsieme degli archi del grafo. Esploro a livelli, ogni livello contiene i nodi con la stessa distanza dal nodo sorgente.

**BFS utile per ricerca cammino che minimizza num mi archi (non cammino minimo!) e componenti connesse**



- **In profondità(DFS):** ossia andando a scegliere i nodi secondo degli specifici criteri, non per forza da quelli adiacenti. Ad ogni passo visito un nuovo nodo, che sia sdiacente al nodo appena visitato/selezionato. Se non ci sono nodi vicini torno indietro e faccio la selezione. E' una ricorsione

**BFS utile per ricerca delle componenti connesse (è anche tipicamente più veloce di BFS)!**

Nei casi in cui ho grafi orientati non è detto che mi venga data la componente connessa perché il verso degli archi potrebbe non consentirmelo. In questo caso dovrà fare la ricerca delle componenti connesse partendo da più punti, e poi unendo insieme gli alberi che ho ottenuto.

Package traversal: ha i metodi implementati per dfs e bfs

### Traversal

#### Depth First Search

Basic algorithms for depth-first searching the nodes of a graph.

<code>dfs.edges</code> (G[, source, depth_limit, ...])	Iterate over edges in a depth-first-search (DFS).
<code>dfs.tree</code> (G[, source, depth_limit, ...])	Returns oriented tree constructed from a depth-first-search from source.
<code>dfs.predecessors</code> (G[, source, depth_limit, ...])	Returns dictionary of predecessors in depth-first-search from source.
<code>dfs.successors</code> (G[, source, depth_limit, ...])	Returns dictionary of successors in depth-first-search from source.
<code>dfs.preorder_nodes</code> (G[, source, depth_limit, ...])	Generate nodes in a depth-first-search pre-ordering starting at source.
<code>dfs.postorder_nodes</code> (G[, source, ...])	Generate nodes in a depth-first-search post-ordering starting at source.
<code>dfs.labeled_edges</code> (G[, source, depth_limit, ...])	Iterate over edges in a depth-first-search (DFS) labeled by type.

#### Breadth First Search

Basic algorithms for breadth-first searching the nodes of a graph.

<code>bfs.edges</code> (G, source[, reverse, depth_limit, ...])	Iterate over edges in a breadth-first-search starting at source.
<code>bfs.layers</code> (G, sources)	Returns an iterator of all the layers in breadth-first search traversal.
<code>bfs.tree</code> (G, source[, reverse, depth_limit, ...])	Returns an oriented tree constructed from a breadth-first-search starting at source.
<code>bfs.predecessors</code> (G, source[, depth_limit, ...])	Returns an iterator of predecessors in breadth-first-search from source.
<code>bfs.successors</code> (G, source[, depth_limit, ...])	Returns an iterator of successors in breadth-first-search from source.
<code>descendants_at_distance</code> (G, source, distance)	Returns all nodes at a fixed <code>distance</code> from <code>source</code> in G.
<code>generic_bfs.edges</code> (G, source[, neighbors, ...])	Iterate over edges in a breadth-first search.

<https://networkx.org/documentation/stable/reference/algorithms/traversal.html>

Complessità computazione:  $O(V+E)$  in entrambi i casi.

Ottengo una lista di fermate

```
def getBFSNodesFromTree(self, source):
    tree = nx.bfs_tree(self._grafo, source)
    archi = list(tree.edges())
    nodi = list(tree.nodes())
    return nodi[1:]
```

1 usage

```
def getDFSNodesFromTree(self, source):
    tree = nx.dfs_tree(self._grafo, source)
    nodi = list(tree.nodes())
    return nodi[1:]
```

Se ciclo sull'iterable di archi ho una coppia di nodi, se voglio arrivare a tutti i nodi raggiungibili da quel nodo mi basta prendere il secondo nella tupla.

```
def getBFSNodesFromEdges(self, source):
    archi = nx.bfs_edges(self._grafo, source)
    res = []
    for u, v in archi:
        res.append(v)
    return res
```

Per il bottone self.\_brnCalcolaRaggiungibili metti disabled = True, in modo che non mi faccia i valori mentre il grafo ancora non esiste. Setta disabled = False dopo aver creato il grafo, in handleCalcolaGrafo()

```
1 usage (1 dynamic)
def handleCreaGrafo(self,e):
    self._model.buildGraph()
    self._view.lst_result.controls.clear()
    self._view.lst_result.controls.append(ft.Text("Grafo correttamente creato!"))
    self._view.lst_result.controls.append(ft.Text(f"Il grafo contiene {self._model.getGraphSize()} nodi"))
    self._view.lst_result.controls.append(ft.Text(f"Il grafo contiene {self._model.getGraphSize()} archi"))
    self._view._btnCalcola.disabled = False
    self._view.update_page()
```

```
def handleCercaRaggiungibili(self,e):
    if self._fermataPartenza is None:
        self._view.lst_result.controls.clear()
        self._view.lst_result.controls.append(
            ft.Text(value="Attenzione, stazione di partenza non selezionata.", color="red"))
        self._view.update_page()
    else:
        nodes = self._model.getBFSNodesFromEdges(self._fermataPartenza)
        self._view.lst_result.controls.clear() ←
        self._view.lst_result.controls.append(
            ft.Text(f"Di seguito, le stazioni raggiungibili a partire da {self._fermataPartenza}:"))
        for n in nodes:
            self._view.lst_result.controls.append(ft.Text(n))
        self._view.update_page()
```

Nell'esercizio della metropolitana di parigi non mi aggiunge gli archi duplicati perche il grafo è in DiGraph(), ossia prevede un solo arco tra due nodi; se fosse un MultiGraph() potrei avere più archi.

Creo archi pesati: ho due strade – farlo con py

```
1 usage
def addEdgesPesati(self):
    allEdges = DAO.getAllEdges()
    for edge in allEdges:
        u = self._idMapFermate[edge.id_stazP]
        v = self._idMapFermate[edge.id_stazA]

        if self._grafo.has_edge(u, v):
            self._grafo[u][v]["weight"] += 1
        else:
            self._grafo.add_edge(u, v, weight=1)
```

Farlo con sql con una group by, creando un nuovo metodo nel dao

```
•select id_stazP , id_stazA, COUNT(*) as n
from connessione c
group by id_stazP , id_stazA
```

```
1 usage
def addEdgesPesatiV2(self):
    self._grafo.clear_edges()
    allEdgesPesati = DAO.getAllEdgesPesati()

    for e in allEdgesPesati:
        self._grafo.add_edge(
            self._idMapFermate[e[0]],
            self._idMapFermate[e[1]],
            weight=e[2]
        )
```

**LEZIONE 05/05/2025: TEMA D'ESAME (ArtsMia)**

Permettere all'utente di inserire, in una casella di testo, il numero identificativ odi un oggetto verificando che sia corretto. -> devo chiedere in giro se quell'oggetto è stato già visto.

**IL DAO IMPLEMENTA IL CAZZO SINGLETONE, NON HA BISOGNO DI INIT!!! Inoltre, i suoi metodi sono @staticmethod**

All'esame la view è data, il DBConnect anche, anche un (1) metodo DAO. Questo è lo scheletro generico.

```
class DAO():

    @staticmethod
    def getAllNodes():
        conn = DBConnect.get_connection()
        cursor = conn.cursor(dictionary=True)

        result = []
        query = """select * from objects_o"""

        cursor.execute(query)

        for row in cursor:
            result.append(ArtObject(**row))

        cursor.close()
```

Nel result ci appendo oggetti di tipo X che ho creato io in una dataclass. Affichè una dataclass sia civile deve avere come metodi `__eq__`, `__hash__`, `__str__`.

\*\*row fa l'unpack di tutti i dati presenti in quella row, quindi mi aiuta a non elencare tutti i santi.

A QUESTO PUNTO CREA NEL MAIN UN BELA tstDAO PER FARE TESTING.

```
from database.DAO import DAO

listObjects = DAO.getAllNodes()

print(len(listObjects))
```

Idem scrivo il modello, dove creo il graph e poi lo testo con tstModel

```
4 usages
5 class Model:
6     def __init__(self):
7         self._graph = nx.Graph()
8
9     1 usage
10    def buildGraph(self):
11        nodes = DAO.getAllNodes()
12        self._graph.add_nodes_from(nodes)
13
14    1 usage
15    def getNumNodes(self):
16        return len(self._graph.nodes)
17
18
```



```
view.py controller.py model.py tstModel.py DAO.py tstDAO.py artObject.py
1 from model.model import Model
2
3 mymodel = Model()
4
5 mymodel.buildGraph()
6 print("N nodi:", mymodel.getNumNodes(), "; N Edges:", mymodel.getNumEdges())
7
```

CREO ARCHI. Se l'arco asiste il peso sarà doverso da zero

Se i nodi sono decine o centinaia questo metodo è ok, sono query semplici e tutto è ok. MA PER GRAFI  
MOLTO GRANDI LA QUERY DIVENTA TROPPO LENTA

```
def addEdgesV1(self):
    for u in self._nodes:
        for v in self._nodes:
            peso = DAO.getPeso(u,v)
            if (peso != None):
                self._graph.add_edge(u, v, weight = peso)
```

Devo creare getPeso, in cui per capire il peso degli archi posso fare il count, ma con il join sulle chiavi esterne! Problemi: non voglio prendere gli archi di ritorno dato l'arco di andata, per cui aggiungo la riga che sta prima della group by. Uso questa query che mi ritorna gli archi che esistono e il loro peso.

```
SELECT eo.object_id, eo2.object_id, count(*) as peso
FROM exhibition_objects eo, exhibition_objects eo2
WHERE eo.exhibition_id = eo2.exhibition_id
and eo.object_id < eo2.object_id
and eo.object_id = 46 and eo2.object_id = 267
GROUP BY eo.object_id, eo2.object_id
```

In questo caso la query è parametrica, per cui nell'eseguirla devo passare i parametri in questa maniera:

```
cursor.execute(query, (v1.object_id, v2.object_id))

for row in cursor:
    result.append(row["peso"])

cursor.close()
conn.close()

if len(result) == 0:
    return None
return result
```

Nel risultato ci appendo solo il peso. Come ulteriore accortezza se vedo che la lunghezza è 0 ritorno None, e nel model con l'altro controllo non aggiungo l'arco.

PER GRAFI DI GRANDI DIMENSIONI ESEGUP QUESTA QUERY PIU COMPLESSA UNA SOLA VOLTA.

```
•SELECT eo.object_id as o1, eo2.object_id as o2, count(*) as peso
FROM exhibition_objects eo, exhibition_objects eo2
WHERE eo.exhibition_id = eo2.exhibition_id
and eo.object_id < eo2.object_id
GROUP BY eo.object_id, eo2.object_id
order by peso desc
```

Ora di queste 3 cose che mi ritorna devo decidere cosa fare: creo una classe, passare una tupla al model o fjkfnkj.

In questo caso vediamo come passare degli oggetti, per cui inizio creando la dataclass Arco. **NON LA USERO' NEL CODICE PER CUI NON MI SERVE FARE HASH, EQ E STR**. Ci lascio solo la definizione così

```
@dataclass
class Arco:
    o1: ArtObject
    o2: ArtObject
    peso: int
```

Con questo sistema (nuova quey + uso class arco) devo **creare un'idMap** che creo nel model e il dao risultante sarà

```
@staticmethod
def getAllArchi(idMap):
    conn = DBConnect.get_connection()
    cursor = conn.cursor(dictionary=True)

    result = []
    query = """SELECT eo.object_id as o1, eo2.object_id as o2, count(*) as peso
               FROM exhibition_objects eo, exhibition_objects eo2
              WHERE eo.exhibition_id = eo2.exhibition_id
                and eo.object_id < eo2.object_id
             GROUP BY eo.object_id, eo2.object_id
            ORDER BY peso DESC"""

    cursor.execute(query)

    for row in cursor:
        result.append(Arco(idMap[o1], idMap[o2], row["peso"]))

    cursor.close()
    conn.close()
```

```
class Model:
    def __init__(self):
        self._graph = nx.Graph()
        self._nodes = DAO.getAllNodes()
        self._idMap = {}
        for v in self._nodes:
            self._idMap[v.object_id] = v
```

### ❓ Quindi perché serve una `idMap` ?

Perché quando lavori con interfacce, query, ID che ti arrivano da fuori (es. da un file, dall'utente o da una GUI), non hai l'oggetto `Aeroporto`, ma solo il suo `id` (es. `1234`).

👉 E il grafo non ti permette di cercare i nodi per ID a meno che non li hai indicizzati tu.

### 💡 Esempio pratico:

Hai un grafo creato così:

```
python Copia Modifica
self._grafo.add_edge(aeroporti[1215], aeroporti[13110])
```

Ora ti arriva l'ID `1215` da un form o una funzione. Vuoi sapere:

- È un nodo del grafo?
- Quali archi ha?
- Che nome ha l'aeroporto?

👉 Senza `idMap`, non puoi fare nulla con quel `1215`. Ma se hai fatto:

```
python Copia Modifica
self._idMap = {a.id: a for a in self._grafo.nodes}
```

Allora puoi fare:

```
python Copia Modifica
if 1215 in self._idMap:
    aeroporto = self._idMap[1215]
    print("Nome:", aeroporto.name)
    print("Vicini:", self._grafo.neighbors(aeroporto))
```

## **DISATTIVA I TASTI DELLA VIEW PRIMA DI FARE BUILD GRAPH!!**

```
def buildGraph(self):
    self._graph.add_nodes_from(self._nodes)
    self.addAllEdges()

def addEdgesV1(self):
    for u in self._nodes:
        for v in self._nodes:
            peso = DAO.getPeso(u, v)
            if (peso != None):
                self._graph.add_edge(u, v, weight=peso)
```

1 usage

```
from database.DAO import DAO
from model.model import Model

listObjects = DAO.getAllNodes()
mymodel = Model()
mymodel.buildGraph()
#
edges = DAO.getAllArchis(mymodel.getIdMap())

print(len(listObjects))
```

TESTO IL DAO:

Per usare il nuovo dao nle modello

```

def addAllEdges(self):
    allEdges = DAO.getAllArchi(self._idMap)
    for e in allEdges:
        self._graph.add_edge(e.o1, e.o2, weight=e.peso)

```

### c) cercare la componente connessa partendo da un nodo

1- verifico che quel nodo esista, altrimenti return none. Per fare questo creo la funzione hasNode che

```

def hasNode(self, idInput):
    # return idInput in self._graph
    return idInput in self._idMap

```

Controlla se idInput è una chiave presente in self.\_idMap, e se non è presente ritorna False.

Nel controller scrivo:

```

if not self._model.hasNode(idInput):
    self._view.txt_result.controls.clear()
    self._view.txt_result.controls.append(
        ft.Text(value="L'id inserito non corrisponde "
                    "ad un node del grafo.", color="red"))
    self._view.update_page()
    return

infoConnessa = self._model.getInfoConnessa(idInput)

```

2- Se c'è faccio una ricerca DFS, ma dfs\_edges mi da gli edges ovviamente, non i nodi della componente connessa, ma li posso ricavare. Per ogni dono del grafo ho tutti i nodi in cui posso andare

\*\*\* modo1\*\*\* conto i succ del nodo source  
 !!! attenta a fare extend wusndo ssgiungi un nodo alla lista di successori, perche non prende doppioni (forse). tutto cio perrhcè i successori di un nodo possono essere diversi.

```

succ = nx.dfs_successors(self._graph, source).values()
res = []
for s in succ:
    res.extend(s)
print("Size connessa con modo 1: ", len(succ.values()))

```

\*\*\*modo2\*\*\* conto i prec del nodo source

```

pred = nx.dfs_predecessors(self._graph, source)
print("Size connessa con modo 2: ", len(pred.values()))

```

\*\*\*modo3\*\*\*conto i nodi del dfs tree

```

dfsTree = nx.dfs_tree(self._graph, source)
print("Size connessa con modo 3: ", len(dfsTree.nodes()))

```

\*\*\*modo4\*\*\* viene tutto gratis facendo quest'istruzione

```

conn = nx.node_connected_component(self._graph, source)
print("Size connessa con modo 3: ", len(conn))

return len(conn)

```

**!!prima di scrivere il txtResult fai**

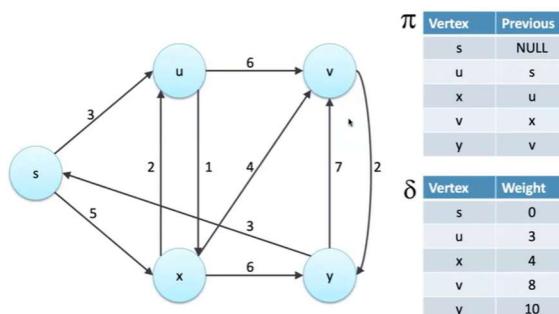
```
Self._view.txt_result.clear()
```

## PATHS IN GRAPHS (shortert path and cycles)

Parliamo di percorso ottimo sulla base dei pesi che sono sugli archi.

Se ci sono cicli molto probabilmente dovrai attaccarti al tram ma prima farti delle domande.

Per salvare i cammini minimi mi serve una struttura che mi memorizzi i nodi e i costi. Ho la struttura dei predecessori e quella dei successori



## Finding shortest paths

- Single-source shortest path (SS-SP)
  - Given  $u$  and  $v$ , find the shortest path between  $u$  and  $v$
  - Given  $u$ , find the shortest path between  $u$  and any other vertex
- All-pairs shortest path (AP-SP)
  - Given a graph, find the shortest path between any pair of vertices

Depending on the problem, you might want:

- The value of the shortest path weight
  - Just a real number
- The actual path having such minimum weight
  - For simple graphs, a sequence of vertices
  - For multigraphs, a sequence of edges

Il predecessore ottimo non è univoco, ma dipende dai nodi sorgente e destinazione che sono stati identificati.

LEMMA SUI CAMMINI OTTIMI:

- Dato un cammino ottimo da  $s$  a  $v$ , il cammino ottimo tra due nodi più vicini di  $s$  e  $v$ , è parte del cammino ottimo  $s \rightarrow v$ 
  - Therefore,  $p_{ij}$  is a shortest path from  $v_i$  to  $v_j$ .



18

Most of the shortest-path algorithms are based on the relaxation technique:

- Vector  $d[u]$  represents  $d(s,u)$
- Keeping track of an updated estimate  $d[u]$  of the shortest path towards each node  $u$
- Relaxing (i.e., updating)  $d[v]$  (and therefore the predecessor  $p[v]$ ) whenever we discover that node  $v$  is more conveniently reached by traversing edge  $(u,v)$

21

## Initial state

Initialize-Single-Source( $G(V,E)$ ,  $s$ )

- for all vertices  $v \in V$ 
  - do
    - $d[v] \leftarrow \infty$
    - $p[v] \leftarrow \text{NIL}$
  - $d[s] \leftarrow 0$
- - We consider an edge  $(u,v)$  with weight  $w$
  - Relax( $u, v, w$ )
    - if  $d[v] > d[u] + w(u,v)$
    - then
      - $d[v] \leftarrow d[u] + w(u,v)$
      - $p[v] \leftarrow u$

Ho trovato un arco non ancora testato, che mi riduce la stima del costo che ho per raggiungere quell'arco.

Testo un arco e ferifoco se mi riduce la mia attuale stima del costo  $v$ .

## Lemma

- Consider an ordered weighted graph  $G=(V, E)$ , with weight function  $w: E \rightarrow \mathbb{R}$ .
- Let  $(u,v)$  be an edge in  $G$ .
- After relaxation of  $(u,v)$  we may write that:
- $d[v] \leq d[u] + w(u,v)$

Se ho aggiornato il cammino ottimo ho il  $<$ , ossia ho migliorato il cammino, altrimenti è  $=$ .

Il sottografo dei cammini ottimi è un albero centrato in  $s$  (nodo da cui inizio la ricerca). Qualunque relaxation io faccia, non ho dei nodi che mi entrino in  $s$ , mi avvicinerò sempre più a destinazione.

Gli algoritmi si dicono in all-sources, ossia che mi danno matrici contenenti l'ottimo dato qualunque nodo di partenza, o one-source, se dato in input un nodo mi contengono tutti i cammini ottimi verso un nodo destinazione.  
Alcuni algoritmi non convergono ma se ne accorgono, altri no.

**FLOWD.WARSHALL:** è un algoritmo all-sources, ossia contiene i cammini da tutti a tutti.

Mi servono due strutture dati:

$\text{Dist}[i][j]$  mi contiene il valore del cammino minimo trovato fino a quel momento. Se non l'ho ancora trovato il valore lì dentro è infinito. La posso calcolare da qualunque nodo del cammino.

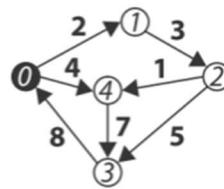
$\text{Pred}[s][j]$  mi fa ricostruire da capo il cammino ottimo che ho cercato. Da quale nodo passo per arrivare in  $j$ ? Pred lo calcolo da s, ossia dal primo nodo del cammino

## Floyd-Warshall: initialization

```
def FLOYD_WARSHALL(V, E, w):
    #Initialise
    for u in V:
        for v in V:
            dist[u][v] = w
            pred[u][v] = None
        dist[u][u] = 0

    #Set distances with existing edges
    for n in neighborhood(u):
        dist[u][n] = w(u,n)
        pred[u][n] = u
```

Initialize  $\text{dist}[][]$  matrix with existing edges



	0	1	2	3	4
0	0	2	$\infty$	$\infty$	4
1	$\infty$	0	3	$\infty$	$\infty$
2	$\infty$	$\infty$	0	5	1
3	8	$\infty$	$\infty$	0	$\infty$
4	$\infty$	$\infty$	$\infty$	7	0

$\text{dist}[u][v]$

Ciclo 2 volte per inizializzare: la prima permette i nodi a infinito e none, la seconda serve per mettere i costi sugli archi.

Poi:

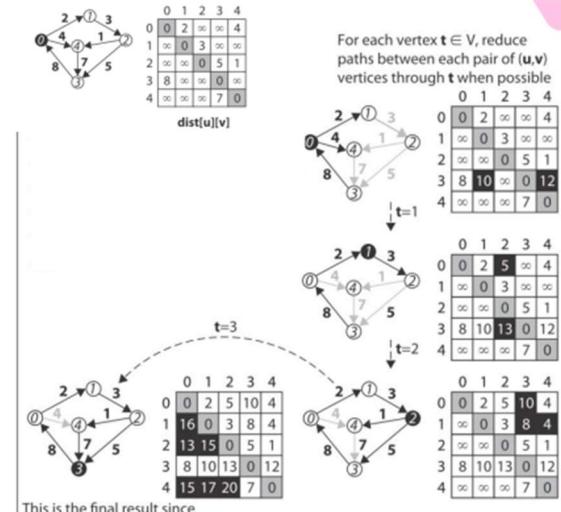
```
def FLOYD_WARSHALL(V, E, w):
    #Initialise
    for u in V:
        for v in V:
            dist[u][v] = w
            pred[u][v] = None
        dist[u][u] = 0

    #Set distances with existing edges
    for n in neighborhood(u):
        dist[u][n] = w(u,n)
        pred[u][n] = u

    #Relax by cycling on nodes (three times)
    for t in V:
        for u in V:
            for v in V:
                #Get a new path between u and v through t
                newDist = dist[u][t] + dist[t][v]

                #Check if the new path is better than previous best
                if (newDist < dist[u][v]):
                    dist[u][v] = newLen
                    pred[u][v] = pred[t][v]

    return dist, pred
```



Provo i nodi da mettere, e se il costo è più piccolo aggiorno il valore. Alla fine del primo loop tengo i costi ottimi a partire da qualsiasi nodo verso qualsiasi altro nodo. Il problema è che stiamo ciclando 3 volte in maniera annidata, per cui ho complessità  $n^3$ .

BELLMAN-FORD-MOORE: per il single-source. Posso fissare almeno il nodo di cui parto osì elimino almeno un ford. Invece di provare tutti i nodi, provo tutti gli archi. Funziona solo per archi positivi.

```

● ● ●
def Bellman_Ford_Moore(V, E, s, w):

    #Initialization
    dist[s] = 0          #set distance to source = 0
    for v in V-{s}:      #set all other dist to infinity and predecessors
        to None
        dist[v] = infinity
        pred[v] = None

    #Relax edges repeatedly
    for i in range(1, len(V)):
        for (u, v) in E:
            if dist[v] > dist[u] + w(u, v): ←
                dist[v] = d[u] + w(u, v) #set new shortest path value
                pred[v] = u #update the predecessor

    for (u, v) in E:
        if dist[v] > dist[u] + w(u, v):
            PANIC!

    return dist, pred

```

Per ogni nodo, ciclo su tutti gli archi e vedo se ho una stima del costo di  $w$  migliore (riga con la freccia).

Assumo che il cammino che mi porta in  $u$  sia già ottimo, per cui non

Faccio tutto un numero di volte pari al numero di nodi, in maniera di essere sicura che l'algoritmo converga.

Alla fine di tutto faccio un double-check e se il double-check fallisce (riga PANIC) vuol dire che le assunzioni fatte sul grafo, ossia l'assenza di cicli, non sono veritieri. Dovrei fare una return senza niente.

DIJKSTRA: è single-source e vale su archi non negativi. È greedy ma garantisce l'ottimalità

```

● ● ●
def Dijkstra(V, E, s, w):

    #Initialise
    dist[s] = 0
    Q = []
    for v in V-{s}:
        dist[v] = infinity
        prev[v] = None
        Q.append(v) # Build a list of unvisited nodes

    #Run relaxation iteration
    while Q is not empty:
        u = q in Q with min dist[q]    # Pick one element of Q
        Q.remove(u)

        for v in neighborhood(u) still in Q: # Cycle on neighbors of k
            newDist = dist[u] + w(u,v)      # Verify if path through u-v is better
            if newDist < dist[v]:
                dist[v] = newDist           # Update the new shortest path
                prev[v] = u

    return dist, pred

```

È l'algoritmo più efficiente che ci sia.

## Dijkstra efficiency

- The simplest implementation is:
- $O(E + V^2)$
- But it can be implemented more efficiently:
- $O(E + V \cdot \log V)$



Floyd-Warshall:  $O(V^3)$   
Bellman-Ford-Moore :  $O(V \cdot E)$

Grafo: directed\_level\_hakimi\_graph serve per dire che è un grafo con lo stesso numero di archi entranti e uscenti.

```

import networkx as nx
import matplotlib.pyplot as plt
import random

G = nx.directed_hakimi_graph([3] * 15,
                             [3] * 15,
                             create_using=None)

for v in G.edges():
    print(G[v[0]][v[1]])
    G[v[0]][v[1]]['weight'] = random.randrange(1,10)
    print(G[v[0]][v[1]])
    print("-----")

print(G.nodes())
print(G.edges())
# nx.draw(G, with_labels=True, font_weight='bold')

pos=nx.spring_layout(G) # pos
nx.nx_agraph.graphviz_layout(G)
nx.draw_networkx(G,pos)
labels = nx.get_edge_attributes(G,'weight')
nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)

print(nx.dijkstra_path(G, 0, 7))
print(nx.dijkstra_path_length(G, 0, 7))

optpath = nx.dijkstra_path(G, 0, 7)
optedges = []
for i in range(0, len(optpath)-1):
    optedges.append([optpath[i], optpath[i+1]])

nx.draw_networkx_edges(G, pos, optedges,
                      edge_color="red")

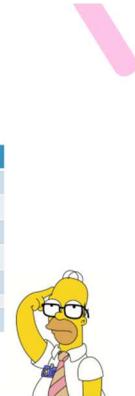
plt.savefig("plot")
plt.show()

```

Dijkstra è il migliore, Floyd-Warshall il peggiore.

## Shortest Paths wrap-up

Algorithm	Problem	Efficiency	Limitation
Floyd-Warshall	AP	$O(V^3)$	No negative cycles
Bellman-Ford	SS	$O(V \cdot E)$	No negative cycles
Repeated Bellman-Ford	AP	$O(V^2 \cdot E)$	No negative cycles
Dijkstra	SS	$O(E + V \cdot \log V)$	No negative edges
Repeated Dijkstra	AP	$O(V \cdot E + V^2 \cdot \log V)$	No negative edges
Breadth-First visit	SS	$O(V + E)$	Unweighted graph



Un ciclo di un grafo, a volte chiamato anche circuito, è un sottoinsieme dell'insieme degli archi che forma un cammino tale che il primo nodo del cammino corrisponda all'ultimo. Un ciclo che utilizza ciascun vertice di un grafo una sola volta è chiamato ciclo hamiltoniano. Un cammino hamiltoniano, chiamato anche cammino di Hamilton, è un cammino tra due vertici di un grafo che visita ciascun vertice una sola volta. Un cammino euleriano, chiamato anche catena di Eulero, sentiero di Eulero, cammino di Eulero o versione "euleriana" di una qualsiasi di queste varianti, è un cammino sugli archi del grafo che utilizza ciascun arco del grafo originale una sola volta. Un ciclo euleriano, chiamato anche circuito euleriano, circuito di Eulero, tour euleriano o tour di Eulero, è un percorso che inizia e termina nello stesso vertice del grafo. Un grafo euleriano è un grafo che ammette un ciclo euleriano.

Tutti i nodi: hamilton

Tutti i vertici: euler. **E' possibile se tutti i nodi hanno un grado pari**

## Eulerian cycles: Hierholzer's algorithm (1)

- Let us assume that G is an Eulerian graph.
- Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v.
  - It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w.
  - The tour formed in this way is a closed tour, although it may not cover all the vertices and edges of the initial graph.

## Eulerian cycles: Hierholzer's algorithm (2)

- As long as there exists a vertex v that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from v, following unused edges until returning to v, and join the tour formed in this way to the previous tour.

## Hierholzer's algorithm Pseudocode

Given an Eulerian Graph G, find an Eulerian circuit of G.

1. Identify a circuit in G and call it  $R_1$ . Mark the edges of  $R_1$  as visited. Let  $i=1$
2. If  $R_i$  contains all edges of G, break.
3. If  $R_i$  does not contain all edges of G, then let  $v_i$  be a node of  $R_i$  that is incident with an unmarked edge  $e_i$
4. Build a new circuit  $Q_i$ , starting from node  $v_i$  and using edge  $e_i$ . Mark edges of  $Q_i$  as visited.
5.  $R_{i+1}$  will result as the conjunction in  $v_i$  of  $R_i$  and  $Q_i$
6. Increment  $i$  by 1 and go to step 2

DATO UN GRAFO AGGIUNGI IL MINIMO NUMERO DI ARCHI AFFICHE IL GRAFO SODDISFI LA CONDIZIONE.

## LEZIONE 12/05/2025

RICERCA CAMMINO MINIMO. Step 1: aggiungere pesi agli archi

Posso fare però due cose:

- 1- Modifico di poco la query e uso una idMap per recuperare le informazioni

```
•select id_stazP, id_stazA, id_linea  
from connessione c  
group by id_stazP , id_stazA, id_linea  
  
select * from linea l
```

- 2- prendo tutte le coppie, e se le coppie sono ripetute prendo solo quella con il valore più alto di velocità

```
•SELECT c.id_stazP , c.id_stazA , max(l.velocita)  
FROM connessione c , linea l  
WHERE c.id_linea = l.id_linea  
GROUP by c.id_stazP , c.id_stazA  
order by c.id_stazP asc , c.id_stazA asc
```

Le informazioni che mi verranno date le metto in una tupla (nel DAO).

Nel modello quando faccio buildGraphPesato richiamerò il nuovo metodo che creerà

```
ef buildGraphPesato(self):  
    self._grafo.clear()  
    self._grafo.add_nodes_from(self._fermate)  
    self.addEdgesPesatiTempi()
```

addEdgesPesatiTempi contiene questo:

```
"""Aggiunge archi con peso uguale  
al tempo di percorrenza dell'arco"""  
self._grafo.clear_edges()  
allEdges = DAO.getAllEdgesVel()  
for e in allEdges:  
    u = self._idMapFermate[e[0]]  
    v = self._idMapFermate[e[1]]  
    peso = getTraversalTime(u, v, e[2])
```

geopy.distance va importata, è una libreria che serve a darmi le distanze geodetiche tra due punti.

```
1 usage  
def getTraversalTime(u, v, vel):  
    dist = geopy.distance.distance(*args: (u.coordX, u.coordY),  
                                    (v.coordX, v.coordY)).km # in km  
    time = dist/vel * 60 # in minuti  
    return time
```

Quando aggiungo un nodo con il secondo metodo non ho bisogno di fare l'if è presente, perché c'è già il distinct nella query.

RICERCA CAMMINO MINIMO --- posso farlo con diversi metodi alternativi, ma usa uno di quelli che permette di specificare il nodo target!

```
def getShortestPath(self, u, v):
    return nx.single_source_dijkstra(self._grafo, u, v)
```

Queste funzionalità le devo richiamare nel controller, facendo gli opportuni controlli (tipo che i nodi esistono o che siano stati inseriti)

Se sorgente/destinazione non esistono, il path è vuoto. PRIMA DI SCRIVERE QUALUNQUE COSA FAI self.\_view.controls.clear()!!

```
def handleCerca(self, e):
    if self._fermataPartenza is None or self._fermataArrivo is None:
        self._view.lst_result.controls.clear()
        self._view.lst_result.controls.append(
            ft.Text(value=f"Attention, selezionare "
                         f"fermata di partenza e di arrivo",
                     color="red"))
        self._view.update_page()
        return
    totTime, path = self._model.getShortestPath(self._fermataPartenza,
                                                self._fermataArrivo)

    if path == []:
        self._view.lst_result.controls.clear()
        self._view.lst_result.controls.append(
            ft.Text(value=f"Non ho trovato un cammino fra {self._fermataPartenza} "
                         f"e {self._fermataArrivo}",
                     color="red"))
        return
```

Se sono qui, un cammino ce l'ho

```
self._view.lst_result.controls.clear()
self._view.lst_result.controls.append(
    ft.Text(value=f"Ho trovato un cammino fra {self._fermataPartenza} "
                 f"e {self._fermataArrivo} che impiega {totTime} minuti.",
            color="green"))

for n in path:
    self._view.lst_result.controls.append(ft.Text(n))
```

## FACCIO L'ESERCIZIO ARTSMIA ----- completa l'esercizio per raggiungere i 30L

### PUNTO 2

- Permettere all'utente di inserire, in una seconda casella di testo, un numero intero, detto LUN, compreso tra 2 e la dimensione della componente connessa relativa al vertice selezionato al punto 1c.
- Alla pressione del bottone "Cerca oggetti", il programma dovrà cercare il *cammino di peso massimo*, avente lunghezza<sup>1</sup> pari a LUN, il cui vertice iniziale coincida con il vertice selezionato nel punto 1.c., che comprenda esclusivamente vertici che abbiano tutti la stessa *classification*.<sup>2</sup>
- Al termine della ricerca, il programma dovrà stampare il cammino, indicando gli oggetti incontrati (ordinati per *object\_name*) ed il peso totale del cammino trovato.

1- Aggiungo bottone e relative cose al controller e model.

```
self._ddlLun = ft.Dropdown(label="Lun")
self._btnCerca = ft.ElevatedButton(text="Cerca Oggetti",
                                    on_click=self._controller.handleCerca)
```

Affichè le cose vengano allineate nonostante siano in una riga 2el, nell'altra 3 el, uso coggetti di tipo Container, perché qui poi è possibile gestire la width manualmente

```
self._btnCompConnessa = ft.ElevatedButton(text="Cerca Connessa", on_click=self._controller.handleCerca,
                                           bgcolor="orange",
                                           color="white",
                                           width=200,
                                           disabled=True)

row1 = ft.Row( controls: [
    ft.Container(self._btnAnalizzaOggetti, width=250),
    ft.Container(self._txtIdOggetto, width=250),
    ft.Container(self._btnCompConnessa, width=250)],
               alignment=ft.MainAxisAlignment.CENTER)
self._page.controls.append(row1)

# row 2
self._ddlLun = ft.Dropdown(label="Lun")
self._btnCerca = ft.ElevatedButton(text="Cerca Oggetti",
                                    on_click=self._controller.handleCerca)
row2 = ft.Row( controls: [
    ft.Container(content=None, width=250),
    ft.Container(self._ddlLun, width=250),
    ft.Container(self._btnCerca, width=250)],
               alignment=ft.MainAxisAlignment.CENTER)
self._page.controls.append(self._txt_result)
self._page.update()
```

Ha senso attivare i tastini della view dopo ..... quindi li cerco in handleComponentaConnessa

```
sizeCompConnessa = self._model.getInfoConnessa(idInput)

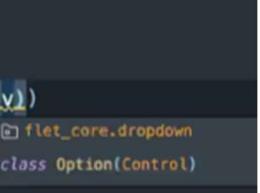
self._view.txt_result.controls.clear()
self._view.txt_result.controls.append(
    ft.Text(f"La componente connessa che contiene "
           f"il nodo {self._model.getObjectFromId(idInput)} ha dimensione pari a {sizeCompConnessa}"))
self._view._ddlLun.disabled = False
self._view._btnCerca.disabled = False
self._view.update_page()
```



Per riempire il dropdown ciò che ho fatto finora è

```
myValues = range(2, sizeCompConnessa)
for v in myValues:
    self._view._ddlLun.options.append(ft.dropdown.Option(v))

self._view.update_page()
```



A tooltip for the 'ft.dropdown.Option' class is displayed, showing its inheritance from 'Control'.

Posso fare tutto in maniera più pythonica così:

Per ogni elemento di myValues applico la lambda function che in questo caso consiste in creare da x un oggetto di tipo dropdown.Option( )

Per ogni intero della lista di partenza assegno un ddOption

```
myValues = list(range(2, sizeCompConnessa))
# for v in myValues:
#     self._view._ddlLun.options.append(ft.dropdown.Option(v))

myValuesDD = list(map(lambda x: ft.dropdown.Option(x), myValues))
self._view._ddlLun.options = myValuesDD

self._view.update_page()
```

Arrivato ad handleCerca sono sicura che l'oggetto esista, devo solo vedere se l'utente ha selezionato qualcosa dal dd.

```
def handleCerca(self, e):
    source = self._model.getObjectFromId(int(self._view._txtIdOggetto.value))

    lun = self._view._ddlLun.value
    if lun is None:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text(value="Attenzione, selezionare un parametro"))
        self._view.update_page()
        return
    lunInt = int(lun)
    path, pesoTot = self._model.getOptPath(source, lunInt)
```

(aggiornamento dopo ricorsione)

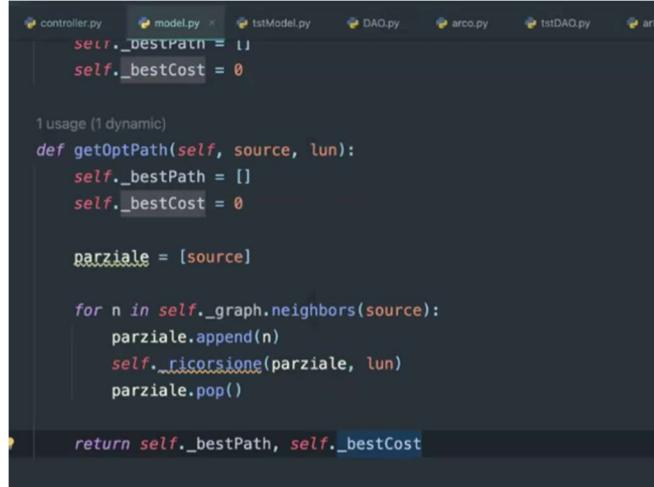
```
handleCerca(self, e):
    source = self._model.getObjectFromId(int(self._view._txtIdOggetto.value))

    lun = self._view._ddlLun.value
    if lun is None:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text(value="Attenzione, selezionare un parametro"))
        self._view.update_page()
        return
    lunInt = int(lun)
    path, pesoTot = self._model.getOptPath(source, lunInt)

    self._view.txt_result.controls.clear()
    self._view.txt_result.controls.append(ft.Text(f"Cammino trovato con peso totale={pesoTot}"))
```

TROVO I VICINI CON LA RICORSIONE:

inizio con una sol pariale che contiene solo in nodo di partenza, poi provo tutte le strade possibili andando a testare i possibili vicini.



```
controller.py model.py tstModel.py DAO.py arco.py tstDAO.py artC
self._bestPath = []
self._bestCost = 0

1 usage (1 dynamic)
def getOptPath(self, source, lun):
    self._bestPath = []
    self._bestCost = 0

    parziale = [source]

    for n in self._graph.neighbors(source):
        parziale.append(n)
        self._ricorsione(parziale, lun)
        parziale.pop()

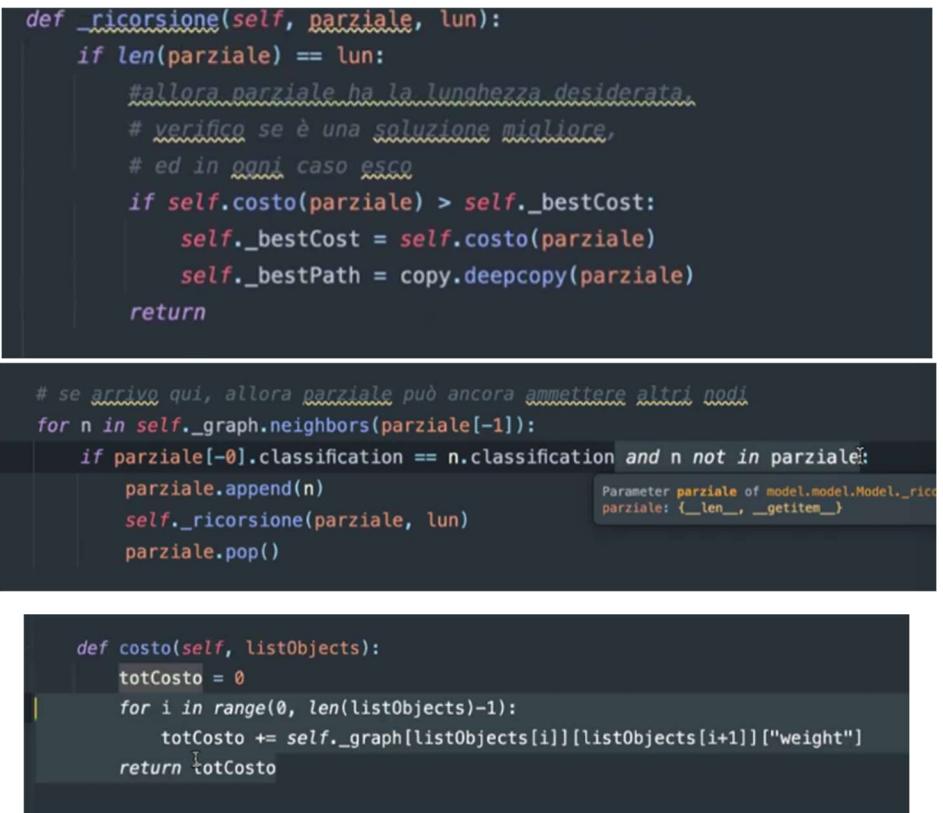
    return self._bestPath, self._bestCost
```

Ora tutto sta in dove faccio la ricorsione. Importante fare backtracking! Non dimenticare.

Quando chiamo la ricorsione c'è per prima cosa l'if di terminazione, e se ci entro parziale ha la lunghezza che io voglio, per cui:

- Il costo è minore -> esco
- Il costo è maggiore o maggiore/uguale -> salvo la soluzione ed è la nuova migliore

Ricorda di fare la deepcopy()



```
def _ricorsione(self, parziale, lun):
    if len(parziale) == lun:
        # allora parziale ha la lunghezza desiderata
        # verifico se è una soluzione migliore,
        # ed in ogni caso esco
        if self.costo(parziale) > self._bestCost:
            self._bestCost = self.costo(parziale)
            self._bestPath = copy.deepcopy(parziale)
        return

    # se arrivo qui, allora parziale può ancora ammettere altri nodi
    for n in self._graph.neighbors(parziale[-1]):
        if parziale[-1].classification == n.classification and n not in parziale:
            parziale.append(n)
            self._ricorsione(parziale, lun)
            parziale.pop()

def costo(self, listObjects):
    totCosto = 0
    for i in range(0, len(listObjects)-1):
        totCosto += self._graph[listObjects[i]][listObjects[i+1]]["weight"]
    return totCosto
```

Ogni volta che chiamo ricorsione dentro la ricorsione aggiungo un elemento, quando esco dalla chiamata con il pop lo tolgo l'elemento, forzando a provare tutti i possibili valori (infatti la ricorsione sta dentro un for). Quindi lun iniziale = 1 -> chiamo: rifaccio ricorsione e ottendo un vettore con una len2 -> lo tolgo

????????????? aspe

Lunghezza pari a lun -> condizione di terminazione

Con stessa classification -> aggiungo il vincolo in ricorsione, faccio l'append solo nel caso in cui la roba è verificata. FILTRO I NODI AMMISSIBILI E ESPLORO SOLO I RAMI (facendo l'append) AMMISSIBILI

## TEMA D'ESAME 02/07/2018 – QUERY UN PO'DIFFICILI

# Prova d'esame del 02/07/2018 – Turno C

Si consideri il database "extflightdelays", contenente informazioni su aeroporti, compagnie aeree, orari di partenza e di arrivo dei voli effettuati negli Stati Uniti durante il 2015. Il database (disponibile su Kaggle all'indirizzo: <https://www.kaggle.com/usdot/flight-delays/data>) è strutturato secondo il diagramma ER della pagina seguente.

Si intende costruire un'applicazione JavaFX che permetta di interrogare tale base dati, e calcolare informazioni a proposito dei voli tra i diversi aeroporti. L'applicazione dovrà svolgere le seguenti funzioni:

### PUNTO 1

- Permettere all'utente di inserire un numero di compagnie aeree minimo  $x$ , e di selezionare il bottone "Analizza aeroporti".
- Alla pressione del bottone, creare un grafo che rappresenti gli aeroporti su cui operano più compagnie.

Esame 02-07-2018 TURNO C

# compagnie minimo	<input type="text"/>	Analizza aeroporti
Aeroporto di partenza	<input type="button" value="▼"/>	Aeroporti connessi
Aeroporto di destinazione	<input type="button" value="▼"/>	
Numero tratta massimo	<input type="text"/>	Cerca itinerario

Il grafo deve essere semplice, non orientato e pesato, i vertici devono rappresentare gli aeroporti su cui operano almeno  $x$  compagnie aeree (in arrivo o in partenza), e gli archi devono rappresentare le rotte tra gli aeroporti collegati tra di loro da almeno un volo. Il peso dell'arco deve rappresentare il numero totale di voli tra i due aeroporti (poiché il grafo non è orientato, considerare tutti i voli in entrambe le direzioni: A->B e B->A).

- Permettere all'utente di selezionare, da un menu a tendina, uno degli aeroporti presenti nel grafo (**a1**).
- Alla pressione del bottone "Aeroporti connessi", stampare l'elenco degli aeroporti adiacenti a quello selezionato, in ordine decrescente di numero totale di voli.

Il rischio qui è di aggiungere archi tra nodi che non erano parte del grafo. !!! devi filtrare!

Devo considerare i voli in entrambe le ricorsioni.

CREAZIONE GRAFO:

Per nodi voglio gli aeroporti con almeno X compagnie, che query faccio?

Ho due possibilità:

CIO CHE METTO NEL SELECT DEVE ANDARE NELLA GROUP BY pari pari!!

```
•SELECT a.ID, a.IATA_CODE , f.AIRLINE_ID, count(*)
  FROM airports a, flights f
 WHERE a.ID = f.ORIGIN_AIRPORT_ID or a.ID = f.DESTINATION_AIRPORT_ID
  GROUP BY a.ID, a.IATA_CODE , f.AIRLINE_ID
```

usando però il count di distinct airline\_id

OPPURE faccio delle query annidate

```
•SELECT t.ID, t.IATA_CODE, count(*) as N
  FROM (SELECT a.ID, a.IATA_CODE , f.AIRLINE_ID, count(*)
        FROM airports a, flights f
       WHERE a.ID = f.ORIGIN_AIRPORT_ID or a.ID = f.DESTINATION_AIRPORT_ID
      GROUP BY a.ID, a.IATA_CODE , f.AIRLINE_ID) t
  GROUP BY t.ID, t.IATA_CODE
```

Prima conto il numero di voli eseguiti da una certa compagnia tra due aeroporti, poi .....

Con having ci aggiungo il filtro, in modo da prendere quelli con almeno X voli

```
SELECT t.ID, t.IATA_CODE, count(*) as N
FROM (SELECT a.ID, a.IATA_CODE , f.AIRLINE_ID, count(*)
      FROM airports a, flights f
     WHERE a.ID = f.ORIGIN_AIRPORT_ID or a.ID = f.DESTINATION_AIRPORT_ID
   GROUP BY a.ID, a.IATA_CODE , f.AIRLINE_ID) t
  GROUP BY t.ID, t.IATA_CODE
having N >= 5
```

Nel model

```
class Model:
    def __init__(self):
        self._graph = nx.Graph()
        self._airports = DAO.getAllAirports()
        self._idMapAirports = {}
        for a in self._airports:
            self._idMapAirports[a.ID] = a

    def buildGraph(self, nMin):
        nodes = DAO.getAllNodes(nMin, self._idMapAirports)
        self._graph.add_nodes_from(nodes)
```

Posso creare l'oggetto aeroporto in questa maniera esotica usando l'idMap, cos' che ho come nodo l'oggetto idMap

```
conn = DBConnect.get_connection()

result = []

cursor = conn.cursor(dictionary=True)
query = """SELECT t.ID, t.IATA_CODE, count(*) as N
           FROM (SELECT a.ID, a.IATA_CODE , f.AIRLINE_ID, count(*)
                 FROM airports a, flights f
                WHERE a.ID = f.ORIGIN_AIRPORT_ID or a.ID = f.DESTINATION_AIRPORT_ID
              GROUP BY a.ID, a.IATA_CODE , f.AIRLINE_ID) t
             GROUP BY t.ID, t.IATA_CODE
            having N >= %s
            order by N asc"""

cursor.execute(query, (nMin,))

for row in cursor:
    result.append(idMapAirports[row["ID"]])

cursor.close()
conn.close()
return result
```

## CREAZIONE ARCHI:

posso, al solito, creare una query più semplice e poi gestire tutto in python (tipo problemi di duplicazione) oppure fare una query più complicata e star chill dopo.

MODO 1: query semplice -----

```
@staticmethod
def getAllEdgesV1(idMapAirports):
    conn = DBConnect.get_connection()

    result = []

    cursor = conn.cursor(dictionary=True)
    query = """SELECT f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID, count(*) as n
               from flights f
              group by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
              order by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
            """
    cursor.execute(query)

    for row in cursor:
        # result.append((idMapAirports[row["ORIGIN_AIRPORT_ID"]],
        #                 idMapAirports[row["DESTINATION_AIRPORT_ID"]],
        #                 row["n"]))

        result.append(Arco(idMapAirports[row["ORIGIN_AIRPORT_ID"]],
                           idMapAirports[row["DESTINATION_AIRPORT_ID"]],
                           row["n"]))
```

Nel memorizzare le info sugli archi o faccio così, ossia memorizzare info in una tupla, oppure una dataclass Arco che avrà come attributi AeroportoP, AeroportoD, Peso, da cui posso prendere tutte le info col punto.

Il model dovrà implementare tutto a mano cioè la somma nel num di voli da un lato all'altro, devo verificare che i nodi siano nel grafo e che l'arco non sia già presente!

```
def addAllArchiV1(self):
    allEdges = DAO.getAllEdgesV1(self._idMapAirports)
    for e in allEdges:
        if e.aeroportoP in self._graph and e.aeroportoD in self._graph:
            if self._graph.has_edge(e.aeroportoP, e.aeroportoD):
                self._graph[e.aeroportoP][e.aeroportoD]["weight"] += e.peso
            else:
                self._graph.add_edge(e.aeroportoP, e.aeroportoD, weight = e.peso)
```

MODO 2: query più complicata -----

Prendo due copy della stessa tabella con le righe invertite

```
*select t1.ORIGIN_AIRPORT_ID, t1.DESTINATION_AIRPORT_ID, t1.n+t2.n
  FROM (SELECT f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID, count(*) as n
        from flights f
       group by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
      order by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID ) t1
    left JOIN (SELECT f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID, count(*) as n
        from flights f
       group by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
      order by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID ) t2
      on t1.ORIGIN_AIRPORT_ID = t2.DESTINATION_AIRPORT_ID
     and t1.DESTINATION_AIRPORT_ID = t2.ORIGIN_AIRPORT_ID
 WHERE
```

Left join con origin airport id mi fa avere lo stesso fatto con le righe invertite, perché ho deviso che voglio che mi vengano affiancate le righe con aeroporto di partenza1 = aerop di arrivo2 e aerop part2 = aerop arrivo1. Il where mi aiuta con i doppioni

ORIGIN_AIRPORT_ID	DESTINATION_AIRPORT_ID	n	ORIGIN_AIRPORT_ID	DESTINATION_AIRPORT_ID	n
0	20	10	20	0	11
0	93	10	93	0	10
0	228	10	228	0	8
1	86	34	86	1	31
2	20	6	20	2	10
2	51	8	51	2	5

Così facendo però ci vengono anche dei null perché alle volte non ho il volo di ritorno!! Questa questione la risolvo con il metodo COALESCE() di SQL. Esso prende una lista di oggetti e mi restituisce il primo di questi che non è null

```
*select t1.ORIGIN_AIRPORT_ID, t1.DESTINATION_AIRPORT_ID, COALESCE(t1.n, 0) + COALESCE (t2.n,
  FROM (SELECT f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID, count(*) as n
        from flights f
       group by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
      order by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID ) t1
    left JOIN (SELECT f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID, count(*) as n
        from flights f
       group by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
      order by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID ) t2
      on t1.ORIGIN_AIRPORT_ID = t2.DESTINATION_AIRPORT_ID
     and t1.DESTINATION_AIRPORT_ID = t2.ORIGIN_AIRPORT_ID
    where t1.ORIGIN_AIRPORT_ID < t1.DESTINATION_AIRPORT_ID or t2.ORIGIN_AIRPORT_ID is NULL
```

Se in alcune righe non ho il volo di ritorno senza mettere il secondo pezzo dell'OR non mi funziona perché me le taglia fuori.

Ho gli archi già puliti per cui posso fare

```
def addAllArchiV2(self):
    allEdges = DAO.getAllEdgesV2(self._idMapAirports)
    for e in allEdges:
        if e.aeroportoP in self._graph and e.aeroportoD in self._graph:
            self._graph.add_edge(e.aeroportoP, e.aeroportoD, weight=e.peso)
```

Casi tipici in cui posso riempire il DD dopo aver creato il grafo. **NON RIEMPIRLO NELL' `__init__` !!!!**

Passo a riempire il **bottone analizza**, che conterrà anche il `fillDD`

```
def handleAnalizza(self, e):
    cMinTxt = self._view.txtInCMin.value
    if cMinTxt == "":
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("Inserire un valore numerico."))
        return

    try:
        cMin = int(cMinTxt)
    except ValueError:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("Il valore inserito non è un intero."))
        return

    if cMin <= 0:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("Inserire un intero positivo."))
        return

    self._model.buildGraph(cMin)

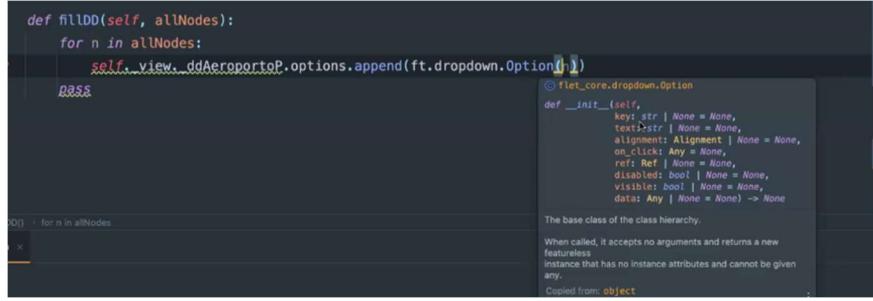
    allNodes = self._model.getAllNodes()
    self.fillDD(allNodes)
    nNodes, nEdges = self._model.getGraphDetails()
    self._view.txt_result.controls.clear()
    self._view.txt_result.controls.append(ft.Text("Grafo correttamente creato:"))
    self._view.txt_result.controls.append(ft.Text(f"Numero di nodi:{nNodes}"))
    self._view.txt_result.controls.append(ft.Text(f"Numero di nodi:{nEdges}"))

analizza()
```

Le info ce le ha il model, dove scrivo

```
def getGraphDetails(self):
    return self._graph.number_of_nodes(), self._graph.number_of_edges()
```

Funzioni per la creazione dei dd



Ricorda: dara è cio che è effettivamente presente del DD, key= cio che viene visualizzato nel DD, ONCLICK serve perché ogni volta che il fra mi clicca qualcosa, mi salvo cio che ha selezionato in una var (Che dichiaro ora inizializzandola a None nell'init). Questa variabile conterrà **L'OGGETTO DI TIPO AEROPORTO CHE IL BRO HA SELEZIONATO DAL dd, DOVE PERO' LUI VEDE SOLO CIO CHE HO INDICATO COME KEY.**

```
def fillDD(self, allNodes):
    for n in allNodes:
        self._view._ddAeroportoP.options.append(
            ft.dropdown.Option(data=n,
                               key=n.IATA_CODE,
                               on_click=self.pickDDPartenza
            ))
```

1 usage

```
def pickDDPartenza(self, e):
    self.choiceDDAeroportoP = e.control.data
```

Posso sortare gli all nodes per IATA CODE crescente, e farlo in getAllNodes

```
1 usage (1 dynamic)
def getAllNodes(self):
    nodes = list(self._graph.nodes)
    nodes.sort(key=lambda x: x.IATA_CODE)
    return nodes
```

## PUNTO D:

Prendo tutti i vicini e li ordino con il peso dell'arco che li collega. Il bro usa una lista di tuple. Il secondo elemento della tupla sono i pesi, e lo sorth in base a quelli

```
def getSortedNeighbors(self, node):
    neighbors = self._graph.neighbors(node) # self._graph[node]
    neighborTuples = []
    for n in neighbors:
        neighborTuples.append((n, self._graph[node][n]["weight"]))

    neighborTuples.sort(key=lambda x: x[1])
    return neighborTuples
```

## Ricerca vicini

```
1 usage (1 dynamic)
def handleConnessioni(self, e):
    if self._choiceDDAeroportoP == None:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("Attenzione, selezionare una voce dal menù."))
        return
    viciniTuple = self._model.getSortedNeighbors(self._choiceDDAeroportoP)
    self._view.txt_result.controls.clear()
    self._view.txt_result.controls.append(ft.Text(f"Di seguito i vicini di {self._choiceDDAeroportoP}"))
    for v in viciniTuple:
        self._view.txt_result.controls.append(ft.Text(f"{v[0]} - peso: {v[1]}"))
    self._view.update_page()
```

## Ricerca percorso – controller

```
def handlePercorso(self, e):
    if self._choiceDDAeroportoP == None:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("Attenzione, selezionare una voce dal menù come partenza."))
        return
    if self._choiceDDAeroportoD == None:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("Attenzione, selezionare una voce dal menù come destinazione."))
        return

    path = self._model.getPath(self._choiceDDAeroportoP, self._choiceDDAeroportoD)
    |

    if len(path) == 0:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text(f"Cammino fra {self._choiceDDAeroportoP} e {self._choiceDDAeroportoD} non trovato! Di seguito i nodi del cammino:"))
    else:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text(f"Cammino fra {self._choiceDDAeroportoP} e {self._choiceDDAeroportoD} trovato! Di seguito i nodi del cammino:"))
        for p in path:
            self._view.txt_result.controls.append(ft.Text(p))
```

Ricerca percorso --- model

```
usage (raydynamic)
def getPath(self, v0, v1):
    path = nx.dijkstra_path(self._graph, v0, v1, weight=None)
    return path
```

Weight = None -> dijkstra assume che tutti gli archi hanno peso 1, per cui mi trova il minimo in termini di numeri di archi.

ALTERNATIVE—

```
# path = nx.shortest_path(self._graph, v0, v1)
# |
# myDict = dict(nx.bfs_predecessors(self._graph, v0))
# path = [v1]
# while path[0] != v0:
#     path.insert(0, myDict[path[0]])
#
# return path
```

Shortest\_path implementa o dijkstra o bellman ford, di default implementa il primo

Fds\_predecessor mi da un iteratore come ritorno, su cui ciclare per prenderci il predecessore dell'ultimo nodo che ho aggiunto

## LEZIONE 19/05/2025

Continuazione tema d'esame: punto 2:

### PUNTO 2

- a. Permettere all'utente di selezionare da un menu a tendina un aeroporto di destinazione **a2**, e un numero massimo di tratte **t** che è disposto a percorrere.
- b. Alla pressione del bottone "Cerca itinerario", sviluppare un algoritmo ricorsivo agente sul grafo creato al punto 1 per cercare l'itinerario di viaggio tra **a1** e **a2** che massimizzi il numero totale di voli per ciascuna delle tratte del percorso selezionato (in altre parole, il percorso che massimizzi la somma dei pesi degli archi traversati), utilizzando al massimo **t** tratte.
- c. Al termine della ricerca, il programma dovrà stampare l'itinerario, indicando gli aeroporti visitati, e il numero totale di voli disponibili sul percorso.

Nella realizzazione del codice, si lavori a partire dalle classi (Bean e DAO, FXML) e dal database contenuti nel progetto di base. È ovviamente permesso aggiungere o modificare classi e metodi.

Tutti i possibili errori di immissione, validazione dati, accesso al database, ed algoritmici devono essere gestiti, non sono ammesse eccezioni generate dal programma.

- 1- Imposto il metodo per la ricorsione, va nel model. Si chiama getCammino ottimo, che prende in input aerop1, aerop2, num tratte
- 2- Inizializzo sei parametri self che memorizzano il bestPath e la besrObjFunc
- 3- Creo la var ricorsione

```
new
def getCamminoOttimo(self, v0, v1, t):
    self._bestPath = []
    self._bestObjFun = 0

    parziale = [v0]

    self._ricorsione(parziale, v1, t)

    return self._bestPath, self._bestObjFun

1 usage new *
def _ricorsione(self, parziale, v1, t):
```

NELLA RICORSIONE:

- 1- vedo se la parziale è una possibile soluzione, se lo è devo verificare che sia ottima
- 2- esco

per far ciò, siccome ho un aeroporto di arrivo, mi basta vedere se l'ultimo valore presente nella lista parziale è l'ottimo (questa è la condizione di terminazione). Per di più, voglio l'ottimo che utilizzi al massimo t tratte.

objVal mi calcola il calore della funzione obiettivo

se posso aggiungere nodi, prendo uno la volta i vicini dell'ultimo nodo aggiunto e aggiungo un nodo la volta, facendo ripartire la ricorsione ogni pezzettino aggiunto.

**!!! fai il controllo per accertarti che l'ultimo nodo aggiunto non sia già parte del percorso!**

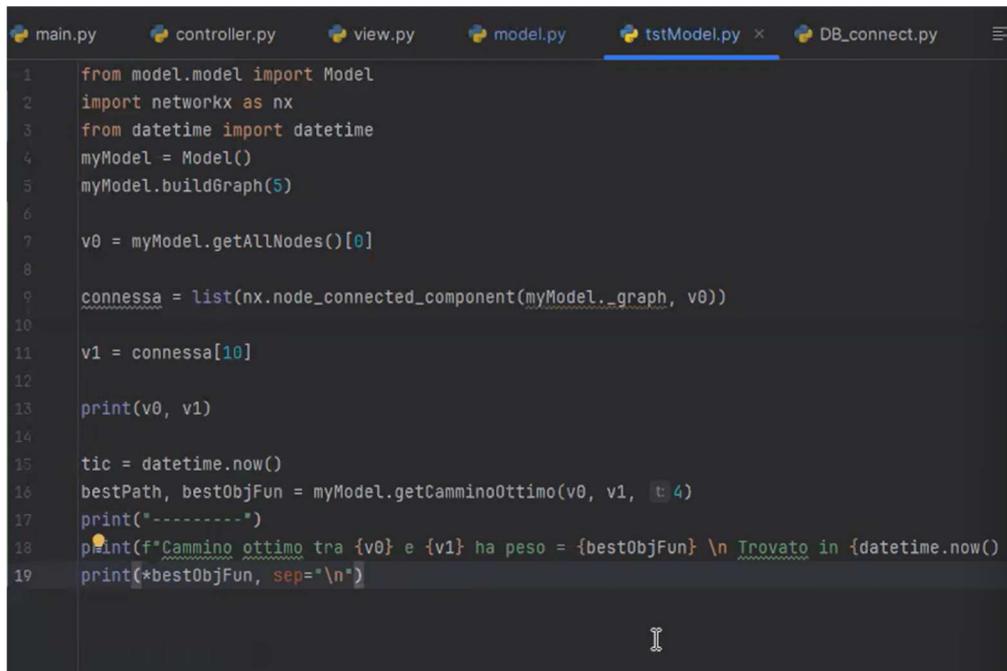
```
2 usages new *
def _ricorsione(self, parziale, v1, t):
    #Verificare se parziale è una possibile soluzione
    #verificare se parziale è meglio del best
    #esco
    if parziale[-1] == v1:
        if self.getObjFun(parziale) > self._bestObjFun:
            self._bestObjFun = self.getObjFun(parziale)
            self._bestPath = copy.deepcopy(parziale)

    if len(parziale) == t+1:
        return

    #Posso ancora aggiungere nodi
    #prendo i vicini e aggiungo un nodo alla volta
    # ricorsione
    for n in self._graph.neighbors(parziale[-1]):
        if n not in parziale:
            parziale.append(n)
            self._ricorsione(parziale, v1, t)
            parziale.pop()
```

Perché faccio il pop? Perché se sono uscito fuori da `self.ricorsione()` vuol dire che quel ramo l'ho esplorato tutto, quindi torno indietro per ripartire

TESTO LA RICORSIONE:



```
1 from model.model import Model
2 import networkx as nx
3 from datetime import datetime
4 myModel = Model()
5 myModel.buildGraph(5)
6
7 v0 = myModel.getAllNodes()[0]
8
9 connessa = list(nx.node_connected_component(myModel._graph, v0))
10
11 v1 = connessa[10]
12
13 print(v0, v1)
14
15 tic = datetime.now()
16 bestPath, bestObjFun = myModel.getCamminoOttimo(v0, v1, t=4)
17 print("-----")
18 print(f"Cammino ottimo tra {v0} e {v1} ha peso = {bestObjFun} \n Trovato in {datetime.now() - tic}")
19 print(*bestPath, sep="\n")
```

Nel controller, al solito, fai il clear prima di fare append dei nuovi risultati

```
usage (1 dynamic) ± Giuseppe Averta *
def handleCerca(self, e):
    v0 = self._choiceDDAeroportoP
    v1 = self._choiceDDAeroportoD
    t = self._view._txtInTratteMax.value
    tint = int(t)

    path, scoretot = self._model.getCamminoOttimo(v0, v1, tint)
    self._view.txt_result.controls.clear()
    self._view.txt_result.controls.append(ft.Text(f"Il percorso ottimo fra {v0} e {v1} è:"))
    for p in path:
        self._view.txt_result.controls.append(ft.Text(p))
    self._view.txt_result.controls.append(ft.Text(f"Score: {scoretot}"))
    self._view.update_page()
```

## TEMA D'ESAME 7/11/2023 – squadre

03FYZ – Tecniche di programmazione

## Esame del 07/11/2023 – appello riservato

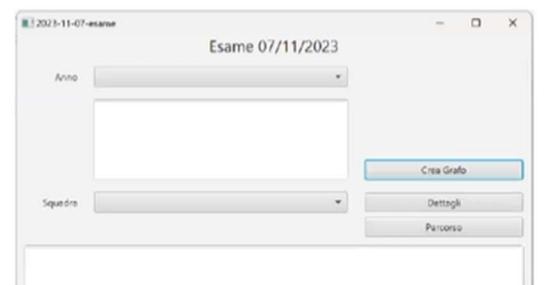
Si consideri il database contenuto nel file `lahmansbaseballdb_small.sql`, presente nella cartella “db” del progetto.

La struttura del database consiste di quattro tabelle principali: `teams`: lista di tutte le squadre iscritte al campionato per ogni anno dal 1871 al 2019; `people`: lista che raccoglie i dettagli dei giocatori professionisti; `appearances`: lista delle partecipazioni dei giocatori al campionato con una (o più) squadre (si noti che non è escluso a priori che un giocatore possa aver cambiato squadra durante l'anno di campionato); `salaries`: lista degli ingaggi del singolo giocatore per uno specifico anno. Il diagramma ER del database è fornito nella cartella “db” ed è riportato nella pagina seguente.

Si intende costruire un'applicazione JavaFX che permetta di che svolga le seguenti funzioni:

### PUNTO 1

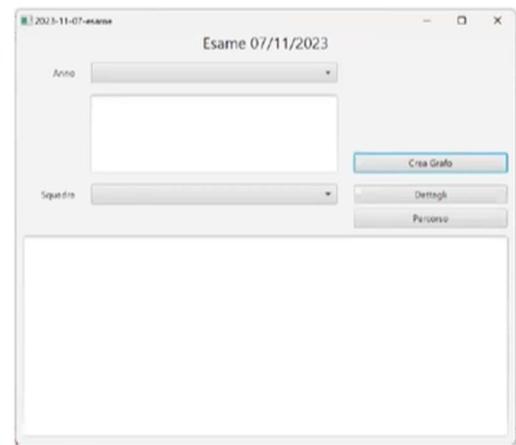
- Permettere all'utente di selezionare da un apposito menu a tendina il valore di un anno di campionato, tra quelli disponibili nel database (`teams.year`), a partire dal 1980 (gli anni precedenti non devono comparire).
- Non appena viene selezionato l'anno (intercettando l'evento `onAction`), si dovrà stampare il numero di squadre (`teams`) che ha giocato in tale anno, e l'elenco delle rispettive sigle nella prima area di testo.



## PUNTO 1

- Permettere all'utente di selezionare da un apposito menu a tendina il valore di un anno di campionato, tra quelli disponibili nel database (`teams.year`), a partire dal 1980 (gli anni precedenti non devono comparire).
- Non appena viene selezionato l'anno (intercettando l'evento `onAction`), si dovrà stampare il numero di squadre (`teams`) che ha giocato in tale anno, e l'elenco delle rispettive sigle, nella prima area di testo (`txtSquadre`). Nello stesso momento occorre aggiornare il contenuto della tendina "Squadre".
- Alla pressione del bottone "Crea Grafo", occorre creare un grafo completo, non ordinato, pesato, in cui i vertici siano le squadre di cui al punto precedente, e gli archi colleghino tutte le coppie distinte di squadre.
- Il peso di ciascun arco del grafo deve corrispondere alla somma dei salari dei giocatori delle due squadre nell'anno considerato.

Nota: potrebbe essere conveniente calcolare e memorizzare la somma dei salari di ciascuna squadra.



Per prima cosa devo riempire il ddAnno. FAI ORDER BY!

```
class DAO():
    usage new *
    @staticmethod
    def getAllYears():
        conn = DBConnect.get_connection()

        results = []

        cursor = conn.cursor(dictionary=True)
        query = "select distinct(year) from teams t where `year` >= 1980 order by `year` desc"
        cursor.execute(query)

        for row in cursor:
            results.append(row["year"])

        cursor.close()
        conn.close()
        return results
```

Neel controller:

```
def fillDDYear(self):
    years = self._model.getYears()
    yearsDD = map(lambda x: ft.dropdown.Option(x), years)
    self._view._ddAnno.options = yearsDD
    self._view.update_page()
    self._view

#yearsDD = []
#for year in years:
#    yearsDD.append(ft.dropdown.Option(year))
```

Questa funzione per ogni elemento di `years` applica la funzione `lambda`, pssia `ft.dropdown.Option()`

On\_chan

Per le selezioni di menu a tendina va on\_change = handleDDYearSelection(self, e)

Gestisco tutto a cascata finche non arrivo al dao. Creo un oggetto di tipo team

```
2 usages(1 dynamic) new *
@staticmethod
def getTeamsOfYear(year):
    conn = DBConnect.get_connection()

    results = []

    cursor = conn.cursor(dictionary=True)
    query = "select * from teams t where t.'year' = %s"
    cursor.execute(query, (year,))

    for row in cursor:
        results.append(Team(**row))

    cursor.close()
    conn.close()
    return results
```

```
new *
@dataclass
class Team:
    ID: int
    year: int
    teamCode: str
    divID: str
    div_ID: int
    teamRank: int
    games: int
    gamesHome: int
    wins: int
    losses: int
    divisionWinnner: str
    leagueWinner: str
    worldSeriesWinner: str
    runs: int
    hits: int
    homeruns: int
    stolenBases: int
    hitsAllowed: int
    homerunsAllowed: int
    name: str
    park: str
```