TECNICHE DI PROGRAMMAZIONE — GRAFI

Nel grafo, ci sono nodi che vengono collegati fra archi.
Il grafo è una struttura dati, per casi concreti.

Un grafo può modellare la metro, dove gli archi sono le connessioni fra le fermate e i nodi sono le fermate stesse.
Posso inserire su ogni arco un peso.

La struttura grafo è molto versatile.

Tipi di grafo:
I grafi con loop sono grafici semplici che ammettono la partenza di un arco da se stesso ed arriva a se stesso.

Semplice = NON DIRETTO

Se parlo di una stazione della metro, c'è un senso, mi porta a parlare di ARCHI DIRETTI —> ORIENTATI. Posso andare solo da A a B, se voglio una tratta di ritorno, sono due archi diversi.

Sugli archi posso mettere delle informazioni —> i pesi.

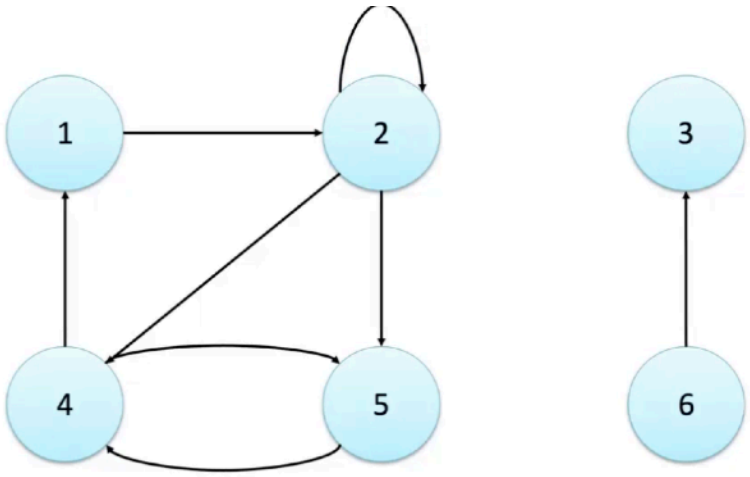
Come metto l'informazione? Uso delle LABEL che inserisco sull'arco.

I nodi sono solitamente delle DATACLASS DAL DATABASE.

SOLITAMENTE:
NODO —> DATACLASS DAL DATABASE.

GRAFI DIRETTI E ORIENTATI
Vengono modellati con un set finito di vertici, un set di oggetti di dataclass, che finiranno in una collection, che avrà bisogno di un __EQ__ E UN __HASH__

E definisco una collezione di tuple.



Grafo ORIENTATO DIRETTO.

Nella struttura dati:

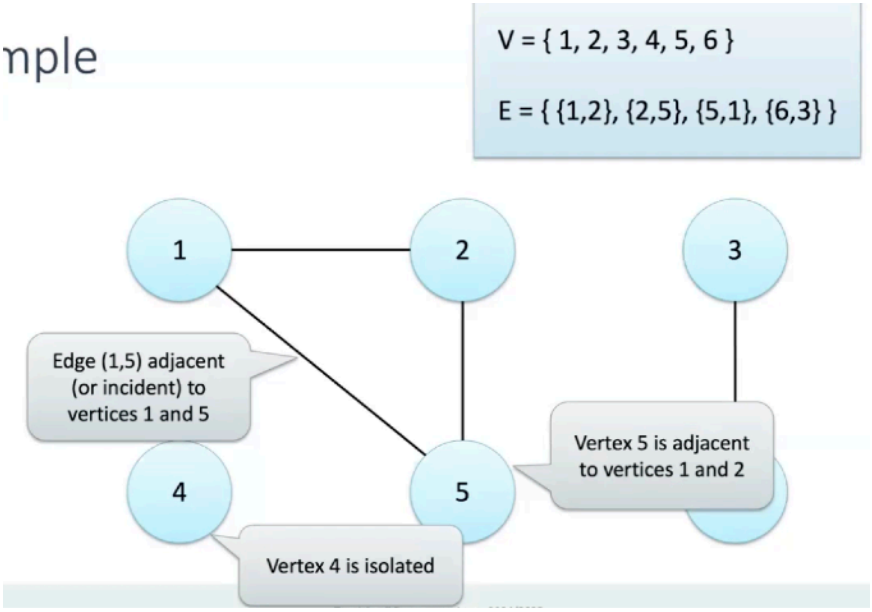
I nodi sono UNA LISTA DI NODI, finita
Gli archi sono una lista o un set di tuple.

Gli archi sono per esempio un set come questo:

E = {(1,2), (2,2), (2,5), (5,4), (4,5), (4,1), (2,4), (6,3)}

NB il primo elemento della tupla è sempre il nodo di partenza e il secondo elemento è sempre il nodo di arrivo.

Grafi indiretti
SEMPRE rappresentati con una tupla nodi, archi
Ma gli archi sono rappresentati da coppie non ordinate, avere B,A o A,B è uguale quindi sono meno archi da considerare



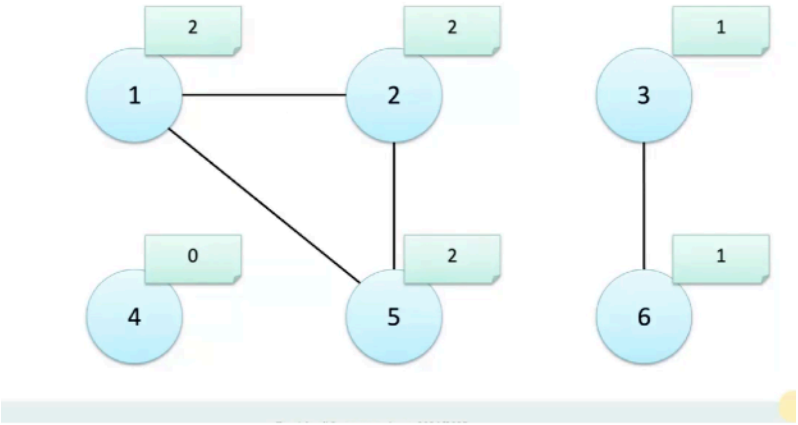
TROVARE LE COMPONENTI CONNESSE???

DEGREE —> ORDINE DEL GRAFO
È associato al singolo vertice, posso dire che il nodo 4 ha degree =4; cioè il numero di archi che partono e arrivano da quel vertice —> VALE SOLO PER GRAFI NON DIRETTI

Nei grafi DIRETTI distinguiamo
In-degree che sono il numero di archi entranti;
Out-degree che è il numero di archi uscenti;
Degree è la somma dei due.

Un vertice con degree=0 è isolato.

ee



In questo grafo, 1-2-5 è UNA COMPONENTE CONNESSA.

Se tolgo l'arco da 1 a 5 ho comunque una componente connessa perchè mi permetterebbe comunque di trovare un percorso che colleghi tutti i nodi.

PERCORSI

I grafi si prestano ad essere ESPLORATI.

Un cammino è UNA SEQUENZA DI NODI.
Unico vincolo: che il grafo abbia gli archi che mi facciano arrivare dal nodo di partenza al nodi di arrivo, la connessione deve esistere.

Albero: grafo connesso con una struttura non diretta, a-ciclica.

Gli alberi sono importanti perché li ottengo da un algoritmo di visita.

Dati due nodi, il cammino fra due nodi è unico —> identifica l'albero.

GRAFO PESATO

Un grafo pesato è un grafo con delle entità o delle dataclass sui nodi e un valore sull'arco, che può indicare diverse cose.

LIBRERIA DEL GRAFO

networkX

È un package che serve per esplorare grafi.
Fornisce una struttura dati per esplorare diverse categorie di grafi, con diversi tipi di oggetto sul Nodo, se è hashable —> con __EQ__ e __HASH__

I nodi sono gli oggetti dataclass che definiamo.

NB POSSIAMO VEDERE IL GRAFO COME UN DIZIONARIO IN CUI UN NODO è LA CHIAVE E COME VALORI HA I NODI ADIACENTI.

NEL GRAFO POSSO INSERIRE QUALSIASI TIPO DI DATO

Per AGGIUNGERE UN NODO —> add_node
Per AGGIUNGERE UN ARCO —> add_edge

Per aggiungere una lista di nodi —> add_nodes_from
Per aggiungere una lista di archi —> add_edges_from

Gli archi possono avere un PESO —> va specificato nel momento di AGGIUNTA DELL'ARCO AL GRAFO

Un grafo è quindi un dizionario di dizionari di dizionari …

Le chiavi sono I NODI e per ogni nodo HO UN DIZIONARIO.

```
grafo.add_edge( u_of_edge: "Due",  v_of_edge: "Uno", weight=0.9)
print(grafo.edges())
```

Restituisce:

```
[('Due', 'Uno')]
```

Con queste istruzioni:

```
print(grafo["Due"])
print(grafo["Due"]["Uno"])
```

Restituisce:

```
{'Uno': {'weight': 0.9}}
{'weight': 0.9}
```

Quindi il grafo è un dizionario di dizionari, dove il primo accesso per [ ] restituisce il nodo vicino il dizionario con il peso associato al suo valore.

Con l'accesso per due volte [ ] [ ] io vado a prendere il dizionario con il peso.

Con il grafo diretto ho sia i PREDECESSORI DI UN NODO che i SUCCESSORI DEL NODO.

Grafo.nodes e **grafo.edges** mi prende LA **LISTA DI NODI E DI ARCHI DEL GRAFO**.

RICORDATI

Quindi:
NetworkX è la libreria da importare per il grafo;
Nx.Graph() crea un grafo non orientato;
nx.DiGraph() crea un grafo orientato;

con **Add_node** —> aggiungo un nodo al grafo;
Posso usare **add_nodes_from(lista_nodi)** e aggiunge una lista di nodi al grafo.

Con **add_edge** —> aggiungo un arco **TRA UN NODO E UN ALTRO NODO** —> Posso **specificare** il suo **PESO("WEIGHT")**.
QUANDO **CREO UN ARCO TRA DUE NODI SI Può DEFINIRE IL PESO DELL'ARCO**.

il grafo è un dizionario di dizionari, dove il **primo accesso per [ ]** restituisce **il nodo vicino il nodo richiesto** con **il peso associato al valore dell'arco che collega i due nodi.**
Con **l'accesso per due volte [ ] [ ] io vado a prendere il dizionario con il peso dell'arco.**

```
print(grafo["Due"])
print(grafo["Due"]["Uno"])
```

```
{'Uno': {'weight': 0.9}}
{'weight': 0.9}
```

Grafo.nodes e **grafo.edges** mi prende LA **LISTA DI NODI E DI ARCHI DEL GRAFO**.

FLUSSO DI ESECUZIONE:

DAO —> PRENDO GLI ELEMENTI DAL DATABASE
MODEL —> CHIAMO I METODI DEL DAO E GESTISCO IL GRAFO
CONTROLLER —> RICHIAMO I METODI DEL MODEL, GESTISCO I BOTTONI DELLA VIEW E RIEMPIO LA VIEW (DROPDOWN E TEXTFIELD)

DAL CONTROLLER PRENDO LA VIEW..

Step da seguire:

Prendo i nodi dal database —> DAO
Li passo al MODELLO, salvo questi nodi
Creo il grafo, aggiungo i nodi al grafo
Creo gli archi, ed aggiungo gli archi al grafo
Passo il grafo al controller che lo aggiunge alla View tramite il bottone.

DIZIONARIO IN CUI ASSOCIO AD OGNI ID_FERMATA UN OGGETTO DI TIPO FERMATA.

```
    self._fermate = DAO.getAllFermate()
    self._grafo = nx.DiGraph()

    self._idMapFermate = {}
    for f in self._fermate:
        self._idMapFermate[f.id_Fermata] = f #associa l'id della fermata all'oggetto fermata
```

Metodo che permette di aggiungere un arco ciccando sulle fermate, con un metodo che restituisce un booleano “hasConnessione”:

```
def addEdges1(self):  new *
    # aggiungo gli archi con doppio ciclo sui nodi e testando se per ogni coppia esiste una connesisone

    for u in self._fermate:
        for v in self._fermate:
            if DAO.hasConnessione(u, v) and u != v:
                self._grafo.add_edge(u, v)
                print(f"aggiunto arco tra: {u} e {v} ")
```

ALTRO MODO: prendo i vicini di un nodo passato come parametro e ciclo sui vicini.

```
def addEdges2(self):  new *
    # ciclo SOLO UNA VOLTA, e faccio una query per trovare tutti i vicini
    for u in self._fermate:
        for v in DAO.getVicini(u):
            self._grafo.add_edge(u, v)
```

NB!!! CASO IN CUI LA QUERY è TROPPO LENTA POSSO CICLARE SU PYCHARM

```
def addEdges3(self):  1 usage  new *
    # faccio una query unica che prende tutti gli archi e poi ciclo qui
    allEdges = DAO.getAllEdges()
    for e in allEdges:
        u = self._idMapFermate[e.id_stazP]
        v = self._idMapFermate[e.id_stazA]
        self._grafo.add_edge(u, v)
```

HO PRESO TUTTI GLI ARCHI DAL DAO E CICLO SU PYCHARM: un arco era una connessione tra un nodo e l'altro, quindi sono andato a prendere dalla mappa le due fermate con gli id che ha quella connessione.

NOTA BENISSIMO

```
    self._fermate = DAO.getAllFermate()
    self._grafo = nx.DiGraph()

    self._idMapFermate = {}
    for f in self._fermate:
        self._idMapFermate[f.id_Fermata] = f #associa l'id della fermata all'oggetto fermata
```

```
def addEdges3(self):  1 usage  new *
    # faccio una query unica che prende tutti gli archi e poi ciclo qui
    allEdges = DAO.getAllEdges()
    for e in allEdges:
        u = self._idMapFermate[e.id_stazP]
        v = self._idMapFermate[e.id_stazA]
        self._grafo.add_edge(u, v)
```

ALGORITMO DI VISITE

Visita in ampiezza: BREADTH FIRST —> VEDO **PER OGNI NODI TUTTI I SUO VICINI** —> vede tutti i nodi a distanza 1, poi a distanza 2, ecc. —> **I NODI VENGONO VISTI A LIVELLI.**

DEPTH FIRST —> visita in profondità—> PARTO DA UN NODO, SCELGO UN SUO VICINO, VISITO QUEL NODO VICINO.
Poi da quel nodo SCELGO UN VICINO A CASO, E COSI VIA.

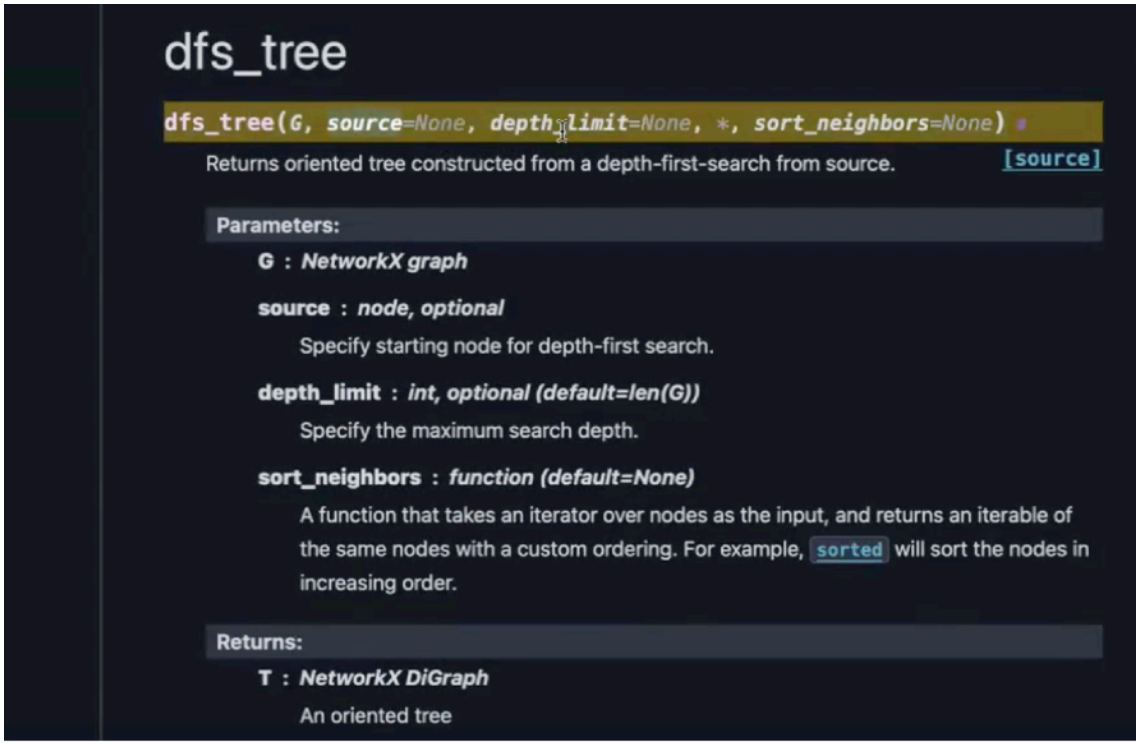
VIENE FUORI UN ALBERO —> ALBERO DI VISITA
Restituisce un nuovo grafo che è un albero, un grafo diretto i cui nodi sono tutti i nodi raggiungibili dal non sorgente e gli archi sono gli archi che permettono di raggiungere il nodo distale.
I nodi sono quindi tutti i nodi raggiungibili e gli archi sono gli archi percorsi nell'algoritmo di percorso quando si è creato l'albero.
Un algoritmo di visita va a mettere tutti i nodi nel grafo albero tutti i nodi della COMPONENTE CONNESSA.
L'albero che ne risulta è l'albero a cui appartengono tutti i nodi della componente connessa.

ALTRA PROPRIETA - BFS
Essendo che vado a esplorare il grafo a livelli, io prendo sempre il percorso minimo, quindi il cammino fra source e final è il minimo.

È utile per trovare le componenti connesse e i cammini minimi.

La libreria networkX ammette metodi per **DFS e BTS —> dfs_edges, dfs_tree, dfs_predecessors, dfs_successors.**

Uguale per BFS



dfs_tree

`dfs_tree(G, source=None, depth_limit=None, *, sort_neighbors=None)`

Returns oriented tree constructed from a depth-first-search from source. [source]

Parameters:

G : *NetworkX graph*

source : *node, optional*

Specify starting node for depth-first search.

depth_limit : *int, optional (default=len(G))*

Specify the maximum search depth.

sort_neighbors : *function (default=None)*

A function that takes an iterator over nodes as the input, and returns an iterable of the same nodes with a custom ordering. For example, `sorted` will sort the nodes in increasing order.

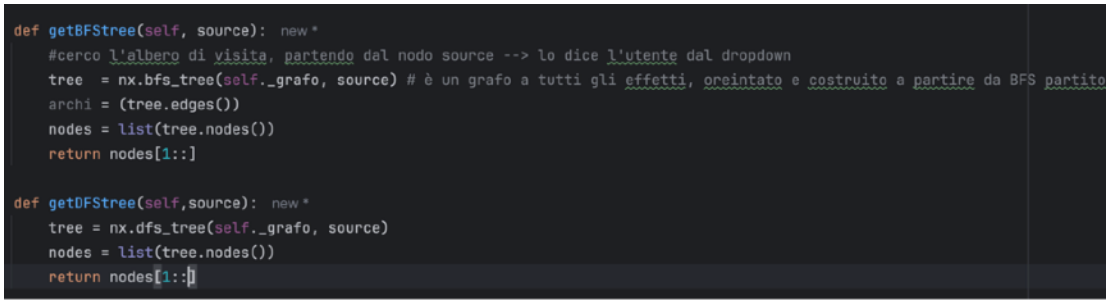
Returns:

T : *NetworkX DiGraph*

An oriented tree

NOTA BENE —> BFS ESPLORA IL GRAFO PER LIVELLI, QUINDI PRIMA QUELLI A DISTANZA 1, POI DISTANZA 2, ECC
DFS ESPLORA IN PROFONDITA —> PRIMA IL SUO PRIMO ALLA SUA SINISTRA, POI ANALIZZA UN VICINO DI QUEL NODO, E COSI VIA

Posso anche usare i metodi: archi = nx.bfs_edges(self.grafo, source) —> restituisce una TUPLA:
Quindi dovrò ciclare
```
res = []
for u,v in archi:
        res.append(v)
```



```
def getBFStree(self, source):  new *
    #cerco l'albero di visita, partendo dal nodo source --> lo dice l'utente dal dropdown
    tree  = nx.bfs_tree(self._grafo, source) # è un grafo a tutti gli effetti, oreintato e costruito a partire da BFS partito
    archi = (tree.edges())
    nodes = list(tree.nodes())
    return nodes[1::]

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

COME FACCIO A COLLEGARLO?
Vado nel controller e devo riempire il metodo CALCOLARAGGIUNGIBILI —> che gestisce il pulsante della View.
Il pulsante deve essere abilitato solo DOPO aver creato il grafo, lo creo quindi disabilitato, lo andrò a rendere attivo solo dopo aver creato il drago.

```
def getBFSNodesFromEdges(self, source):  1 usage
    archi = nx.bfs_edges(self._grafo, source)
    res = []
    for u,v in archi:
        res.append(v)
    return res
```

Prendo gli archi dal grafo

partendo da un source con BFS

```
def handleCercaRaggiungibili(self,e):  1 usage (1 dynamic)  Giuseppe Averta *
    # metodo che posso chiamare solo se il grafo è stato creato -->  modifico il view per dire che il pulsante calcola comincia
    # da disabilitato, e lo riabilito quando creo il crafo

    if (self._fermataPartenza is None):
        self._view.lst_result.controls.clear()      # VERIFICA CHE L'UTENTE ABBIA SELEZIONATO UNA FERMATA
        self._view.lst_result.controls.append(ft.Text( value: "attenzione, non hai selezionato la fermata di partenza", color="red"
        self._view.update_page()
        return
    # source è il nodo di partenza dal dropdown
    nodes = self._model.getBFSNodesFromEdges(self._fermataPartenza) # nodi visitabili dalla fermata di partenza
    self._view.lst_result.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(n)
    self._view.update_page()
```

```
def read_DD_Partenza(self,e):  1 usage  Giuseppe Averta
    print("read_DD_Partenza called ")
    if e.control.data is None:
        self._fermataPartenza = None
    else:
        self._fermataPartenza = e.control.data

def read_DD_Arrivo(self,e):  1 usage  Giuseppe Averta
    print("read_DD_Arrivo called ")
    if e.control.data is None:
        self._fermataArrivo = None
    else:
        self._fermataArrivo = e.control.data
```

RICORDATI BENISSIMO

Nx.Graph() crea un grafo non orientato;
nx.DiGraph() crea un grafo orientato;

con **Add_node** —> aggiungo un nodo al grafo;
Posso usare **add_nodes_from(lista_nodi)** e aggiunge una lista di nodi al grafo.

Con **add_edge** —> aggiungo un arco **TRA UN NODO E UN ALTRO NODO** —> Posso **specificare** il suo **PESO("WEIGHT")**.
QUANDO **CREO UN ARCO TRA DUE NODI SI Può DEFINIRE IL PESO DELL'ARCO**.

il grafo è un dizionario di dizionari, dove il **primo accesso per [ ]** restituisce **il nodo vicino il nodo richiesto** con **il peso associato al valore dell'arco che collega i due nodi.**
Con **l'accesso per due volte [ ] [ ] io vado a prendere il dizionario con il peso dell'arco.**

Step da seguire:

Prendo i nodi dal database —> DAO
Li passo al MODELLO, salvo questi nodi
Creo il grafo, aggiungo i nodi al grafo
Creo gli archi, ed aggiungo gli archi al grafo
Chiamo la funzione di buildGraph nel controller, che gestisce il bottone della View tramite il metodo handleCreaGrafo.

PER LEGGERE IL DROPDOWN DELLA VIEW (NEL CONTROLLER) readDD FONDAMENTALE

```
def read_DD_Partenza(self,e):  1 usage  Giuseppe Averta
    print("read_DD_Partenza called ")
    if e.control.data is None:
        self._fermataPartenza = None
    else:
        self._fermataPartenza = e.control.data

def read_DD_Arrivo(self,e):  1 usage  Giuseppe Averta
    print("read_DD_Arrivo called ")
    if e.control.data is None:
        self._fermataArrivo = None
    else:
        self._fermataArrivo = e.control.data
```

Mi salvo in una variabile "self._fermataPartenza" il valore del dropdown.
FARE SOLO IN CASO IN CUI NEL DROPDOWN CI SIANO OGGETTI.
Metodo chiamato dal "fillDD".

PER SAPERE A COSA è ASSOCIATO UN CERTO ID AL SUO OGGETTO CHE LO IDENTIFICA —> mappa nel costruttore e posso ciclare su TUTTI I NODI dal DAO e salvo nodo di partenza e nodi di arrivo, e lo aggiungo al grafo. (NEL MODEL)

```
self._fermate = DAO.getAllFermate()
self._grafo = nx.DiGraph()

self._idMapFermate = {}
for f in self._fermate:
    self._idMapFermate[f.id_Fermata] = f #associa l'id della fermata all'oggetto fermata
```

self._*mappa*[nodo.id] = *nodo;* Mi serve quando devo prendere l'oggetto a cui fa riferimento un certo id.

AGGIUNGO GLI ARCHI AL GRAFO —> FONDAMENTALE. ADDEDGE
SE I NODI SONO POCHI POSSO FARE UNA QUERY NEL DAO CHE PRENDE TUTTI I NODI E DOPPIO CICLO PER AGGIUNGERLI.

```
def addEdges3(self):  1 usage  new *
    # faccio una query unica che prende tutti gli archi e poi ciclo qui
    allEdges = DAO.getAllEdges()
    for e in allEdges:
        u = self._idMapFermate[e.id_stazP]
        v = self._idMapFermate[e.id_stazA]
        self._grafo.add_edge(u, v)
```

HO PRESO TUTTI GLI ARCHI DAL DAO E CICLO SU PYCHARM: un arco era una connessione tra un nodo e l'altro, quindi sono andato a prendere dalla mappa le due fermate con gli id che ha quella connessione. FARE QUANDO HO POCHI NODI NEL GRAFO

ALGORITMI DI VISITA
BFS—> esploro in ampiezza, quindi prima tutti i liv.0, poi i liv.1, ecc.
DFS —> prima un vicino del source, poi un vicino di quel vicino, un vicino del vicino, ecc.

La libreria networkX ammette metodi per **DFS e BTS —> dfs_edges, dfs_tree, dfs_predecessors, dfs_successors.**
NOTA BENE —> BFS ESPLORA IL GRAFO PER LIVELLI, QUINDI PRIMA QUELLI A DISTANZA 1, POI DISTANZA 2, ECC
DFS ESPLORA IN PROFONDITÀ —> PRIMA IL SUO PRIMO ALLA SUA SINISTRA, POI ANALIZZA UN VICINO DI QUEL NODO, E COSI VIA

Posso anche usare i metodi: archi = nx.bfs_edges(self.grafo, source) —> restituisce una TUPLA:
Quindi dovrò ciclare
```
res = []
for u,v in archi:
    res.append(v)
```

```
def getBFSNodesFromEdges(self, source):  1 usage (
    archi = nx.bfs_edges(self._grafo, source)
    res = []
    for u,v in archi:
        res.append(v)
    return res
```

prendo i nodi dell'albero di BFS dal seguente metodo, con source come nodo di origine che sarà quello del dd.
NB NON è "ARCHI" MA "NODI".

Questo è un metodo del controller che gestisce il pulsante CERCA_RAGGIUNGIBILI.
Il metodo cerca i nodi raggiungibili da una stazione di partenza selezionato dall'utente dal dropdown.

METODO per AGGIUNGERE AL GRAFO GLI ARCHI PESATI DA ME —> FARE SE TANTI NODI

```
def addEdgesPesati(self):  1 usage  new *
    allEdges = DAO.getAllEdges()
    for e in allEdges:
        u = self._idMapFermate[e.id_stazP]
        v = self._idMapFermate[e.id_stazA]
        if self._grafo.has_edge(u, v):
            self._grafo[u][v]["weight"]+=1
        else:
        self._grafo.add_edge(u, v, weight=1)
```

Accesso posizionale al peso:
Per definire il peso di un arco, creo l'arco se non è già presente e imposto il valore del peso a 1;
Se l'arco è già presente tra u e v, allora incremento l'arco tramite accesso posizionale di 1.

(NEL MODEL)

QUESTO METODO VA BENE SE IL PESO è = NUMERO DI ARCHI TRA U E V

MI SERVE USARE UNA TUPLA NEL CURSORE SOLO QUANDO GLI PASSO UN PARAMETRO.

QUANDO MI VIENE CHIESTO UN DiGraph( ) dovrò creare L'ARCO SIA dal nodo A che dal nodo B.

Nel controller devo chiamare I METODI DEL MODEL —> IN HANDLE_CREA_GRAFO DEVO CHIAMARE BUILDGRAPH DEL MODEL.

NB —> solitamente nelle print mi viene chiesto di stampare "GRAFO CORRETTAMENTE CREATO! IL GRAFO HA … NODI E … ARCHI".
Nel model mi salvo questi numeri con due metodi che mi prendono i numeri: getNumNodi e getNumArchi —> quando gestisco il pulsante crea Grafo nel controller devo poi richiamare i due metodi del model.

Nel controller devo mettere anche la LETTURA DEI DROPDOWN e dei CAMPI DELLA VIEW che sono nell'interfaccia grafica. La lettura del DD devo farla SOLO SE NEL DROPDOWN CI SONO OGGETTI, QUINDI SOLO SE MI INTERESSA PRENDERE L'OGGETTO SELEZIONATO. ALTRIMENTI USARE self._bottone.value.
NEL FILLDD DEVO CHIAMARE IN ON_CLICK IL METODO DI READDD.

NEL MODEL QUANDO DEVO AGGIUNGERE GLI ARCHI HO BISOGNO DI UNA MAPPA PER POTER PRENDERE (PROBABILMENTE) DALL'ID UN OGGETTO. —> QUANDO HO OGGETTI SPECIFICI (ES. FERMATA, ArtObject, ecc)

SE LA QUERY è TROPPO COMPLESSA ED è DIFFICILE PRENDERE SOLO GLI ARCHI CHE MI INTERESSANO (CHE RISPETTANO SOLO CERTE CONDIZIONI), SE IL GRAFO HA POCHI NODI POSSO PRENDERE TUTTI GLI ARCHI E POI CICLARE NEL MODEL PER AGGIUNGERLI.

Solitamente un bottone della View deve essere disabilitato nel controller prima di creare il grafo + QUANDO PRENDO UN'INFORMAZIONE DAL DROPDOWN DEVO VERIFICARE CHE L'UTENTE ABBIA SELEZIONATO QUALCOSA.
TOGLIERE IL DISABLED ALLA FINE DEL METODO CREA GRAFO!!

Questo metodo ha il "disabilitato" per il pulsante della richiesta successiva

```
def handleCreaGrafo(self,e):  1 usage (1 dynamic)  Giuseppe Averta *
    # crea il grafo
    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"grafo contiene {self._model.getNumNodi()} nodi "))
    self._view.lst_result.controls.append(ft.Text(f"grafo contiene {self._model.getNumArchi()} archi "))
    self._view._btnCalcola.disabled = False
    self._view.update_page()
```

NEL CONTROLLER DEVO ANCHE INSERIRE I METODI PER RIEMPIRE I DROPDOWN della View —> metodi fillDD

```
def loadFermate(self, dd: ft.Dropdown()):  2 usages (2 dynamic)  Giuseppe Averta
    fermate = self._model.fermate

    if dd.label == "Stazione di Partenza":
        for f in fermate:
            dd.options.append(ft.dropdown.Option(text=f.nome,
                                                 data=f,
                                                 on_click=self.read_DD_Partenza))
    elif dd.label == "Stazione di Arrivo":
        for f in fermate:
            dd.options.append(ft.dropdown.Option(text=f.nome,
                                                 data=f,
                                                 on_click=self.read_DD_Arrivo))
```

SE VOLESSI USARE UNA QUERY PIU COMPLESSA CHE MI PERMETTE DI GESTIRE PIU COSE NEL DAO, QUINDI AGGIUNGERE GIA ALLA LISTA RESULT GLI OGGETTI arco CON OBJECT1 OBJECT 2 E PESO, POSSO USARE QUESTO MODO:

Dato che dal database non riesco a prendere gli oggetti, ho infatti gli id degli oggetti, devo passare una idMap al metodo per poter appendere alla lista result l'oggetto associato a quell'id —> prendere gli archi creando oggetto Arco e grazie ad idMap associare id a oggetto.

```
@staticmethod  new *
def getAllEdges(idMap):
    conn = DBConnect.get_connection()
    result = []
    cursor= conn.cursor(dictionary=True)
    query = """SELECT eo.object_id as o1, eo2.object_id as 02 , 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[row["o1"]], idMap[row["o2"]], row["peso"]))
    cursor.clo
    conn.close
    if len(result) == 0:
        return None
    return result
```

result: list = []

```
@staticmethod  1 usage  new *
def getPeso(v1,v2):
    conn = DBConnect.get_connection()

    result = []

    cursor = conn.cursor(dictionary=True)
    query = """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 = %s and eo2.object_id = %s
               group by eo.object_id, eo2.object_id """
    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
```

NB QUANDO NEL MODELLO DEVO AGGIUNGERE GLI ARCHI, E GLI ARCHI SONO DEGLI OGGETTI SPECIFICI di tipo Arco, NON POSSO FARE SEMPLICEMENTE "add_Edges_from( )..."
Ma per oggetti specifici (come per esempio l'oggetto "Arco") devo ciclare su tutti gli archi presi dal DAO e aggiungere manualmente ogni arco, in questo modo:

```
def buildGraph(self):  2 usages (1 dynamic)  SteCappioB *
    self._grafo.clear()
    self._grafo.add_nodes_from(self.getAllNodes())
    for arco in self.getAllEdges():
        self._grafo.add_edge(arco.art1, arco.art2, weight=arco.peso)
```

FONDAMENTALE —> quando creo l'oggetto Arco costituito da un oggetto Nodo(partenza), un oggetto Nodo(arrivo) e il suo peso, per poter aggiungere l'arco al grafo, una volta che li ho presi dal DAO devo ciclare su tutti gli archi presi dal DAO, e aggiungerli manualmente, specificando nell'arco che l'attributo Weight è uguale all'attributo peso dell'oggetto Arco.

QUANDO DEVO GESTIRE UN CAMPO TESTUALE INSERITO DALL'UTENTE DEVO GESTIRE LA SITUAZIONE CON UN TRY — CATCH

```
def handleCompConnessa(self,e):  1 usage (1 dynamic)  Giuseppe Averta *
    self.scritto  = self._view._txtIdOggetto.value
    if (self.scritto == ""):
        self._view.txt_result.controls.append(ft.Text("non hai inserito nulla, inserire un id valido"))
        self._view.update_page()
        return
    try:
        id = int(self.scritto)
    except ValueError:
        self._view.txt_result.controls.clear()
        self._view.txt_result.controls.append(ft.Text("il valore inserito non è un numero "))
        self._view.update_page()
        return
    infoConnessa = self._model.getInfoConnessa(id)
```

Quando viene chiesto all'utente di inserire un testo all'interno di un textField nell'interfaccia grafica, devo controllare che:

1) **il testo NON SIA NULL —> stampare in questo caso un alert di errore. Se viene chiesto di inserire un id (che è di tipo intero) devo USARE TRY-CATCH. Try —> a fare intero ciò che ha scritto l'utente e INTERCETTARE ValueError —> se non è un numero —> alert**
2) **Se ha passato il try catch —> posso prendere la componente connessa da un metodo del MODEL —> getInfoConnessa**

PER GESTIRE LA COMPONENTE CONNESSA posso usare dfs (NELLA VIEW)

```python
def getInfoConnessa(self, idInput):  2 usages (1 dynamic)  new *
    # identifica la componente connessa che contiene idInput e ne restituisce la dimensione
    #posso usare BFS

    if not idInput in self.idMap:
        return
    source = self.idMap[idInput]

    #modo 1 conto i successori
    succ = nx.dfs_successors(self._grafo, source)
    res = []
    for s in succ:
        res.extend(s)
    print ("size connessa con modo1:  ", len(res))

    #modo 2: conto i predecessori
    pred = nx.dfs_predecessors(self._grafo, source)
    print ("size connessa con modo2: " ,len(pred.values())+1)

    #modo3: conto i nodi dell'albero di visita
    tree = dfs_tree(self._grafo, source)
    print ("size connessa con modo 3", len(tree.nodes()))
```

NECESSARIO SCRIVERE "If not idInput in self.idMap_ Return Source = self.idMap(idInput)

Il modo 4 è usare "nodes_connected_components" di nx (il MIGLIORE!!!!)

```python
#modo 4: uso il metodo nodes_connected_component di networkx
conn = nx.node_connected_component(self._grafo, source)
print ("size conessa con modo 4", len(conn))
```

N.B. se mi chiede LE COMPONENTI CONNESSE di TUTTO il grafo (quindi non a partire da un nodo specifico), si fa conn=nx.connected_componentS(self._grafo) e restituisce tutte le componenti connesse del grafo in una lista. Verrà chiesto di returnare la dimensione di tale lista e si farà "len".

Se devo **RIEMPIRE il dropdown** devo anche fare un **metodo di fillDD** nel controller che **cicli** su (solitamente) i **nodi del grafo** e **aggiunga al dd i campi**, in questo modo:
For f in fermata:
dd.options.append(text = f.nome, data = f, on_click = read_DD_fermata)

In questo metodo di append devo specificare QUAL È il testo che si vuole vedere durante l'interfaccia, qual è l'oggetto e qual è il metodo che gestisce il click.

DIJKSTRA IN nx -> ho due metodi che posso chiamare —> nx.dijkstra_path(grafo, nodo_partenza, nodo_arrivo) —> restituisce il percorso minimo da nodo di partenza a nodo di arrivo.

nx.dijkstra_path_length(grafo, nodo_partenza, nodo_arrivo) —> restituisce il valore del percorso, che sarà il valore minimo per arrivare dal nodo di partenza al nodo di arrivo.

SE VOGLIO ORDINARE I VICINI IN BASE AL PESO DELL'ARCO —> Sorted, vicino, vicini - 1!!!

```
def getVicini(self, nodo):  2 usages (1 dynamic)  new *
    vicini = list(self.grafo.neighbors(nodo))
    res = []
    for vicino in vicini:
        peso = self.grafo[nodo][vicino].get('weight',1)
        res.append((vicino, peso))
    viciniSorted = sorted(res, key = lambda x: x[1], reverse=True)
    return viciniSorted
```

NB i vicini sono gli ADIACENTI.

In viciniSorted ho una lista CON TUPLE —> [(nodoVicino, peso) , (nodoVicino2, peso2), …)

```
@staticmethod  1 usage  new *
def getAllEdgesPesati(idMap):
    conn = DBConnect.get_connection()

    result = []

    cursor = conn.cursor(dictionary=True)
    query = """ select f.ORIGIN_AIRPORT_ID as o, f.DESTINATION_AIRPORT_ID as d, count(DISTINCT f.ID) as peso
                from flights f
                group by f.ORIGIN_AIRPORT_ID , f.DESTINATION_AIRPORT_ID
                """
    cursor
    for ro
        if row["o"] in idMap and row["d"] in idMap:
            result.append(Arco(idMap[row["o"]], idMap[row["d"]], row["peso"]))
    cursor.close()
    conn.close()
    return result
```

No data sources are configured to run this SQL and provide advanced code assistance. Disable this inspection via problem

Configure data source More actions...

Se devo prendere come nodi del grafo dei nodi che soddisfano solo una determinata richiesta, come per esempio soddisfare il valore che l'utente ha selezionato dal dropdown, creo l'idMap solo nel buildGraph, passandogli come parametro il valore che deve rispettare, e nel controller quando chiamerò buildGraph del model gli passerò il valore vero.
NB —> QUANDO DEVO PASSARGLI UNA IDMAP DEVO CONTROLLARE CHE IL VALORE ASSOCIATO A QUELL'ID SIA EFFETTIVAMENTE PRESENTE NELL'IDMAP.

```
def buildGraph(self, nMin):  2 usages (1 dynamic)  new *
    for node in self.getAllNodes(nMin):
        self.idMap[node.ID] = node
```

Quando il grafo che devo creare è NON DIRETTO, ma comunque pesato, il peso di un arco deve calcolarsi in questo modo:

```
def buildGraph(self, nMin):  2 usages (1 dynamic)  new *
    for node in self.getAllNodes(nMin):
        self.idMap[node.ID] = node
    self.grafo.add_nodes_from(self.getAllNodes(nMin))

    for edge in self.getAllEdges():
        if self.grafo.has_edge(edge.aer1, edge.aer2):
            self.grafo[edge.aer1][edge.aer2]['weight'] += edge.peso
        else:
            self.grafo.add_edge(edge.aer1, edge.aer2, weight = edge.peso)
```

Dato che è non diretto, ci sarà solo un arco dal nodo A al nodo B, quindi se l'arco c'è già devo fare +=. Altrimenti aggiungo l'arco.

Lettura del dropdown readDD

```
def readDDp(self, e):  1 usage  new *
    if e.control.data is None:
        self._choiceDDP = None
    else:
        self._choiceDDP = e.control.data

def readDDdest(self, e):  1 usage  new *
    if e.control.data is None:
        self.choiceDDdest = None
    else:
        self.choiceDDdest = e.control.data
```

riempimento del dropdown fillDD

```
def fillDD(self,compMin, dd: ft.Dropdown(), dd1: ft.Dropdown()):  1 usage  new *
    for aer in self._model.getAllNodes(compMin):
        dd.options.append(ft.dropdown.Option(text = aer.IATA_CODE, data = aer, on_click=self.readDDp))
        dd1.options.append(ft.dropdown.Option(text = aer.IATA_CODE, data = aer, on_click=self.readDDdest))
```

QUANDO LEGGO IL DD —> salvarmi in una variabile il valore della scelta.

ORDINAMENTO di NODI CON LAMBDA FUNCTION —> ordino, sorted, ordinamento

```
self._view._txt_result.controls.append(ft.Text("di seguito i dettagli sui nodi: "))
sortedNode = sorted(self._model.getAllNodes(annoInput), key = lambda x: x.StateNme)
for n in sortedNode:
    self._view._txt_result.controls.append(ft.Text(f" {n.StateNme}  -- ha {self._model.getNeigh(n)} vicini"))


self._view._txt_result.controls.append(ft.Text(f"il grafo ha {self._model.getGrafo()} componenti connesse "))
self._view.update_page()
```

Se mi chiedono la componente connessa del grafo:—> restituisce una lista di componenti connesse del grafo.

```
def getCompCOnnessa(self, grafo):  AriGBi *
    conn = nx.node_connected_component(grafo)
    return len(conn)
```

Anche: Conn = connected_components —> restituisce le componenti connesse del grafo IN LISTE.

METODO CHE PRENDE SOLO IL PESO DI UN ARCO.
COLLEGATO A QUESTO METODO CI DOVRà ESSERE UN DOPPIO CICLO NEL MODEL PER POTER AGGIUNGERE GLI ARCHI.

```
@staticmethod  1 usage   S310883 *
def getPeso(anno, IdProd1, IdProd2):

    conn = DBConnect.get_connection()

    result = []

    cursor = conn.cursor(dictionary=True)
    query = """SELECT count(DISTINCT gds2.`Date`) as N
    FROM go_daily_sales gds, go_daily_sales gds2
    WHERE YEAR(gds.`Date`) = %s and gds.`Date` = gds2.`Date` and gds.Retailer_code = gds2.Retailer_code
     and gds.Product_number = %s and gds2.Product_number = %s  """
    cursor.execute(query, (str(anno),IdProd1,IdProd2 ))

    for row in cursor:
        result.append(row["N"])
    cursor.close()
    conn.close()
    return result
```

```
def addAllEdges(self, color, anno):  1 usage   new *
    for u in self.grafo.nodes():
        for v in self.grafo.nodes():
            if u!=v:
                peso = DAO.getPeso(anno, u.Product_number, v.Product_number)
                if peso[0]>0:
                    self.grafo.add_edge(u,v, weight = peso[0])
```

ORDINARE GLI ARCHI IN BASE AL PESO —> FONDAMENTALE

```
def getArchiSorted(self):  1 usage (1 dynamic)   new *
    archi = self.grafo.edges(data = True)
    lista = sorted(archi, key=lambda x:x[2]['weight'], reverse=True)
    return lista
```

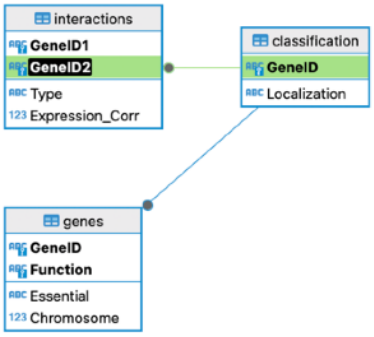
La proprietà .edges del grafo restituisce una lista con tutti gli archi, costituiti con la seguente struttura:
edge ha —> (nodo1, nodo2, {'weight':peso})

Per ciclare sui pesi devo:
Accedere alla posizione due dell'arco—> vado nel dizionario con 'weight' —> accedere per [ ] per andare al valore.

Query quando ho una tabella che contiene come chiave —> due chiavi e la tabella di unione con due id

```
SELECT DISTINCTROW g1.GeneID as g1id, g1.`Function` , g2.GeneID as g2id, g2.`Function` ,
g1.Chromosome as ch1, g2.Chromosome as ch2, c1.Localization, c2.Localization, i.Expression_Corr as peso

FROM genes g1, genes g2, classification c1, classification c2, interactions i
WHERE g1.GeneID = c1.GeneID
and g2.GeneID = c2.GeneID
and c1.Localization = c2.Localization
and g1.GeneID <> g2.GeneID
and g1.Chromosome >= %s
and g1.Chromosome <= %s
and g2.Chromosome >= %s
and g2.Chromosome <= %s
and ((i.GeneID1=g1.GeneID AND i.GeneID2=g2.GeneID) or (i.GeneID1=g2.GeneID AND i.GeneID2=g1.GeneID))
having g1.Chromosome <= g2.Chromosome
```



NB QUANDO DEVO USARE LA IDMAP NEL DAO DEVO AGGIUNGERE UN'STRIUZIONE SPECIFICA ALTRIMENTI NON FUNZIONA

```
@staticmethod  1 usage  new *
def getAllEdgesPesati(anno, idMap):
    conn = DBConnect.get_connection()
    result = []
    cursor = conn.cursor(dictionary=True)
    query = """
                SELECT gds2.Retailer_code as r1 , gds.Retailer_code as r2 , COUNT(DISTINCT gds2.Product_number) as peso
FROM go_daily_sales gds , go_daily_sales gds2
WHERE Year(gds2.`Date`) = %s and Year(gds.`Date`) = %s and
                gds2.Product_number = gds.Product_number and gds.Retailer_code > gds2.Retailer_code
GROUP by gds2.Retailer_code, gds.Retailer_code
            """

    cursor.execute(query, (anno,anno))
    for row in cursor:
        if row["r1"] in idMap and row["r2"] in idMap:
            result.append(Arco(idMap[row["r1"]], idMap[row["r2"]], row["peso"]))
    cursor.close()
    conn.close()
    return result
```

DA SAPERE
Se i nodi sono i retailer, per gli archi io non devo confrontare che i due retailer code sono uguali, perchè i due retailer devono essere diversi
Se il peso dell'arco è il numero di prodotti che i due retailer hanno venduto entrambi, in comune, dovrò fare la join sul product_number

PER OGNI ARCO CAPIRE COSA DEVE ESSERE DIVERSO E COSA DEVE ESSERE UGUALE
—> SE DIVERSO AVRANNO >
SE UGUALI SARà JOIN

Se pochi nodi —> posso fare due metodi, 1 per prendere l'arco e uno per prendere il peso.

Digraph —> FARE TUTTO UGUALE, DEVO SOLO STARE ATTENTO QUANDO AGGIUNGO GLI ARCHI IN BUILDGRAPH, CHE SEGUIRà IL VERSO in ordine di aggiunta.
self.grafo.add_edge(arco.vincitore, arco.perdente, weight=arco.peso) —> crea l'arco dal vincitore al perdente.

Componente connessa:
se di un nodo —> node_connected_components(grafo, nodo) + verificare che il nodo sia nel grafo / idMap.

se del grafo —> connected_components(grafo)

ORDINAMENTO: ORDINARE I VICINI IN BASE AL PESO DELL'ARCO —> Sorted, vicino, vicini. —> nell'ordinamento "x" è l'elemento della collezione che voglio ordinare, accedo con [ ]

```
def getVicini(self, nodo):  2 usages (1 dynamic)  new *
    vicini = list(self.grafo.neighbors(nodo))
    res = []
    for vicino in vicini:
        peso = self.grafo[nodo][vicino].get('weight',1)
        res.append((vicino, peso))
    viciniSorted = sorted(res, key = lambda x: x[1], reverse=True)
    return viciniSorted
```

NB i vicini sono gli ADIACENTI ad un nodo.
In viciniSorted ho una lista CON TUPLE —> [(nodoVicino, peso) , (nodoVicino2, peso2), …)

Lettura del dropdown readDD **riempimento del dropdown fillDD**

```
def readDDp(self, e):  1 usage  new *
    if e.control.data is None:
        self._choiceDDP = None
    else:
        self._choiceDDP = e.control.data

def readDDdest(self, e):  1 usage  new *
    if e.control.data is None:
        self.choiceDDdest = None
    else:
        self.choiceDDdest = e.control.data
```

```
def fillDD(self,compMin, dd: ft.Dropdown(), dd1: ft.Dropdown()):  1 usage  new *
    for aer in self._model.getAllNodes(compMin):
        dd.options.append(ft.dropdown.Option(text = aer.IATA_CODE, data = aer, on_click=self.readDDp))
        dd1.options.append(ft.dropdown.Option(text = aer.IATA_CODE, data = aer, on_click=self.readDDdest))
```

QUANDO LEGGO IL DD —> salvarmi in una variabile il valore della scelta.

ORDINAMENTO di NODI CON LAMBDA FUNCTION —> ordino, sorted, ordinamento

```
self._view._txt_result.controls.append(ft.Text("di seguito i dettagli sui nodi: "))
sortedNode = sorted(self._model.getAllNodes(annoInput), key = lambda x: x.StateNme)
for n in sortedNode:
    self._view._txt_result.controls.append(ft.Text(f" {n.StateNme}  -- ha {self._model.getNeigh(n)} vicini"))

self._view._txt_result.controls.append(ft.Text(f"il grafo ha {self._model.getGrafo()} componenti connesse "))
self._view.update_page()
```

ORDINAMENTO GLI ARCHI IN BASE AL PESO —> FONDAMENTALE

```
def getArchiSorted(self):  1 usage (1 dynamic)  new *
    archi = self.grafo.edges(data = True)
    lista = sorted(archi, key=lambda x:x[2]['weight'], reverse=True)
    return lista
```

Quando scrivo x:x[2][‘weight’] io accedo al dizionario contenente il peso e prendo il valore riferito a ‘weight’.
La proprietà .edges del grafo restituisce una lista con tutti gli archi, costituiti con la seguente struttura:

edge ha —> (nodo1, nodo2, {‘weight’:peso})

Per ciclare sui pesi devo:
Accedere alla posizione due dell’arco—> vado nel dizionario con ‘weight’ —> accedere per [ ] per andare al valore.

Quando nel DAO devo passargli una idMap devo controllare che le righe siano effettivamente presi nell’idMap:
Prima di fare append di row, controllare:

if row[“id1”] in idMap and row[“id2”] in idMap:
…

Io raggruppo secondo un campo o una coppia(di solito negli archi una coppia), tramite GROUP BY.
Group by genera delle sotto-tabelle per ogni campo specificato nella Group by.
Se io specifico nella Group by —> group by driverId —> genera delle sottotabelle PER OGNI DRIVERID contenenti tutti i campi in cui è presente quel driverId nella macrotabella.

Nodi e archi simulazione d'esame

```
SELECT d.*
FROM drivers d , results r  , races r2
WHERE r2.`year` = 1951 and d.driverId = r.driverId and r.raceId = r2.raceId and r.`position` is not null
group by d.driverId
```

```
SELECT re.driverId as vincitore, r2.driverId as perdente, count(*)
from results re, results r2, races ra
where ra.raceId = re.raceId and ra.raceId = r2.raceId and ra.`year` = 1963 and r2.`position` > re.`position`
and r2.`position` is not NULL
and re.`position` is not null
GROUP BY re.driverId, r2.driverId
```

Metodo fillDD associata al change di un dropdown della View —> NB SVUOTARE IL DROPDOWN

```
def fillDDShape(self, e):  1 usage (1 dynamic)  new *
    self.choiceDDyear = self._view.ddyear.value
    if self._view.ddyear.value is None:
        self._view.txt_result1.controls.clear()
        self._view.txt_result1.controls.append(ft.Text("inserisci un anno"))
        self._view.update_page()
        return
    for shape in self._model.getShapeOfYears(self.choiceDDyear):
        self._view.ddshape.options.append(ft.dropdown.Option(str(shape)))
    self._view.update_page()
```

PRIMA DI RIEMPIRLO.

Quando un dropdown dipende dalla scelta di un altro dropdown (o di premere un bottone) devo associare al dd / bottone da cui dipende, il metodo on _ change = fillDD.
Quindi il dropdown che seleziona uno stato (che dipende dal valore dell'anno selezionato) avrà la "e" di evento

Nel controller il metodo fillDDstato avrà nei parametri "e" per evento. —> NB la "e" ce l'ha chi deve cambiare qualcosa.
NB nel controller quando scrivo il metodo fillDDstate devo scrivere questo:

```
def fillDDstate(self, e):  1 usage (1 dynamic)  new *
    self.choiceDDyear = self._view.ddyear.value
    self._view.ddstate.options.clear()
    if (self._view.ddyear.value is None):
        self._view.txt_result1.controls.clear()
        self._view.txt_result1.controls.append(ft.Text("perfavore seleziona un anno
        self._view.update_page()
        return

    for stat in self._model.getAllStateAnno(self.choiceDDyear):
        self._view.ddstate.options.append(ft.dropdown.Option(str(stat)))

    self._view.update_page()
```

Cioè che quello che si deve riempire è proprio il dropdown dello stato

METODO CHE ASSOCIA AD OGNI NODO IL SUO PUNTEGGIO

```
def calcolaPuntiNodo(self, nodo):  2 usages  SteCappioB
    cntVittorie=0
    cntSconfitte=0
    for arco in self.grafo.out_edges(nodo, data=True):
        cntVittorie += arco[2]['weight']
    for arco2 in self.grafo.in_edges(nodo, data=True):
        cntSconfitte+=arco2[2]['weight']
    puntiNodo = cntVittorie-cntSconfitte
    return puntiNodo

def calcolaPuntiTutti(self):  1 usage (1 dynamic)  SteCappioB
    mappaNodoPunti={}
    for nodo in self.grafo.nodes():
        punteggio = self.calcolaPuntiNodo(nodo)
        mappaNodoPunti[nodo]=punteggio

    mappaSorted = dict(sorted(mappaNodoPunti.items(), key=lambda x: x[1], reverse=True))
    return next(iter(mappaSorted)), self.calcolaPuntiNodo(next(iter(mappaSorted)))
```

Quando devo calcolare un certo valore per un certo nodo posso associare il nodo al valore in una mappa.
Per ottenere il valore più alto (o più basso) devo prima ordinare la mappa dove ho associato il nodo al valore.
Per farlo devo prima di tutto dire "dict" perchè è una mappa e voglio mi restituisca appunto un dict, poi "sorted" e fare sorted(mappa.items( ), key = lambda x: x[1]

Con x:x[1] prendo l'elemento della collezione che voglio ordinare e prendo ciò che c'è in [1], ossia il punteggio.
(In [0] c'è il nodo)

Per prendere solo IL PRIMO elemento della mappa uso: **next(iter(mappaSorted))** e il suo punteggio sarà **self.calcolaPuntiNodo(next(iter(mappaSorted)))**

Se mi chiedono le "componenti debolmente connesse" —> metodo "*nx.weakly_connected_components*"

NB —> quando accedo "edge" per notazione [ ] —> RICORDARSI DATA = TRUE
edge [2]["weight"] accedo al valore di peso
self.grafo[nodo1][nodo2]["weight"] associo al valore di peso

NB nella query io inserisco il != se il grafo è orientato, perchè non devo togliere le ripetizioni ma devo tenere l'arco sia uscente che entrante.
Inserisco il > nei grafi non diretti perchè voglio togliere gli archi specchiati, perché sono gli stessi archi che vengono ripetuti in avanti e indietro.

Devo mettere nel count la key word "Distinct" quando il peso si basa su qualcosa esterno alla tabella.

Ordinare vicini per peso

```
def getVicini(self, nodo):  1 usage (1 dynamic)  new *
    vicini = list(self._grafo.neighbors(nodo))
    self.mappaVicini = {}
    for vicino in vicini:
        self.mappaVicini[vicino]=self._grafo[nodo][vicino]['weight']
    mapOrdinata = dict(sorted(self.mappaVicini.items(), key=lambda x:x[1], reverse=True ))

    return mapOrdinata
```

Dict perchè è una mappa.

NB inserire data = True se da problemi.

Ordinare componenti connesse in base alla dimensione di ogni componente connessa.

```
def getCompConnesse(self):  1 usage  new *
    conn = list(nx.weakly_connected_components(self.grafo))
    return len(conn)


def sortComp(self):  1 usage  new *
    lista = list(nx.weakly_connected_components(self.grafo))
    listaS = sorted(lista, key=lambda  x:len(x), reverse=True)
    return listaS[0]
```