COME GESTIRE IL FATTO DI AVERE DELLE CONDIZIONI DI PESI CRESCENTI/DECRESCENTI NEL MOMENTO IN CUI DEVO TROVARE UN PATH NEL GRAFO

```
def getScore(self, listOfNodes):

tot = 0

for i in range(len(listOfNodes) - 1):

tot += self._graph[listOfNodes[i]][listOfNodes[i + 1]]["weight"]

return tot
```

prezzo annuale di ogni prodotto -> table function per il calcolo del prezzo totale

Analizziamo il testo con questa lente:

1. **"Per ciascun prodotto, si calcoli il prezzo di vendita totale nell'anno selezionato A e con il metodo M."**
    - Questo punto è cruciale. Indica la necessità di un *calcolo aggregato* (`SUM`) che deve essere disponibile *per ogni prodotto*. Se questo calcolo fosse richiesto più volte con parametri diversi (anno A e metodo M che possono variare), questo sarebbe un forte indizio.

QUERY PER ARCHI:

quando faccio il join?

- Il Tuo WHERE Richiede Filtri o Condizioni su Campi di Tabelle Diverse (La condizione per l'arco è "pilota 1 ha vinto contro pilota 2 nella *stessa gara*")
- Hai Bisogno di "Ruoli Diversi" per la Stessa Entità? (Il Caso degli Archi)
- **Per collegare alias della stessa tabella padre (es. o1 con o2):** Le condizioni nel WHERE (datediff, store_id, position <, product_number =) sono i tuoi "JOIN". Non sono basati su chiavi primarie/esterne tra le due istanze (che non esistono), ma sulla logica dell'arco.
- **Per collegare tabelle diverse (es. orders e order_items):** Usa le chiavi primarie/esterne (es. o1.order_id = oi1.order_id). Questi sono i JOIN tradizionali.
- **Non dimenticare i filtri!** store_id = %s, year = %s, position is not null

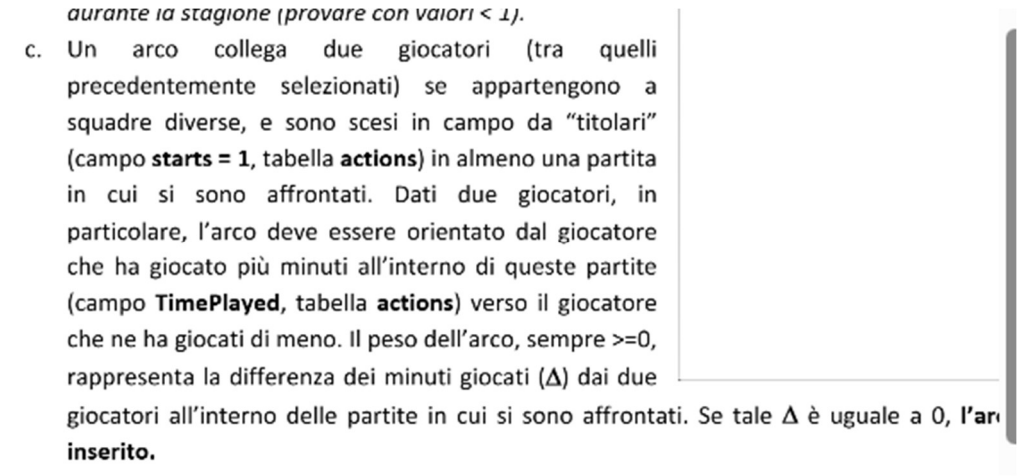
Cosa non dimenticare mai: Assicurati che u e v siano *distinti*

Condizioni Specifiche per gli Archi: Traduci le regole dell'esercizio in SQL.

- **Distinzione tra i Nodi:** u <> v o, se non orientato e vuoi evitare duplicati (es. A-B e B-A), u < v (o u > v). **(Molto importante per grafi non orientati!)**

Cosa non dimenticare mai (WHERE):

1. **Tutti i JOIN:** Se hai N tabelle nel FROM, avrai bisogno di almeno N-1 condizioni di JOIN per collegarle correttamente. Altrimenti, rischi un prodotto cartesiano gigantesco e sbagliato.
2. **Condizione u <> v (o </>):** Quasi sempre necessaria per evitare di creare archi di un nodo verso sé stesso o di duplicare archi in grafi non orientati.



durante la stagione (provare con valori < 1).

c. Un arco collega due giocatori (tra quelli precedentemente selezionati) se appartengono a squadre diverse, e sono scesi in campo da "titolari" (campo **starts = 1**, tabella **actions**) in almeno una partita in cui si sono affrontati. Dati due giocatori, in particolare, l'arco deve essere orientato dal giocatore che ha giocato più minuti all'interno di queste partite (campo **TimePlayed**, tabella **actions**) verso il giocatore che ne ha giocati di meno. Il peso dell'arco, sempre >=0, rappresenta la differenza dei minuti giocati (Δ) dai due giocatori all'interno delle partite in cui si sono affrontati. Se tale Δ è uguale a 0, **l'ar inserito.**

```
select a1.PlayerID, a2.PlayerID, (sum(a1.TimePlayed) - sum(a2.TimePlayed)) as w
from actions a1, actions a2, matches m1, matches m2
where m1.MatchID = a1.MatchID and
      m2.MatchID = a2.MatchID and
      a1.TeamID <> a2.TeamID and
      a1.starts = 1 and a2.starts=1 and
      a1.MatchID = a2.MatchID and
      a1.playerID in (select pu.PlayerID
                              from players pu, actions au
                              where pu.PlayerID = au.PlayerID
                              group by pu.PlayerID, pu.Name
                              having avg(au.Goals) > 0.3) and a2.PlayerID in (select
pa.PlayerID

                        from players pa, actions aa

                        where aa.PlayerID = pa.PlayerID

                        group by pa.PlayerID, pa.Name

                        having avg(aa.Goals) > 0.3)
group by a1.PlayerID, a2.PlayerID
having (sum(a1.TimePlayed) - sum(a2.TimePlayed)) > 0
```

b. Alla pressione del bottone "Crea Grafo", creare un grafo **semplice, orientato e non pesato**, in cui:

- I vertici sono tutti e soli i prodotti distinti (colonna *product_number*) le cui vendite sono state condotte con il metodo **M** selezionato nell'anno **A** selezionato.
- Per ciascun prodotto, si calcoli il prezzo di vendita totale nell'anno selezionato **A** e con il metodo **M**. Il ricavo di una singola vendita è dato da *Unit_sale_price* * *Quantity*.
- Due vertici sono collegati tra loro da un arco, se e solo se il secondo prodotto ha ricavato più del primo prodotto di almeno il **S%** in più (esempio: se S=0.15, il ricavo del secondo prodotto, calcolato sull'intero anno, dovrà essere superiore o uguale a 1.15 volte il ricavo del primo prodotto). In questo modo gli archi del grafo rappresenteranno la direzione verso cui si trovano i prodotti più redditizi.
- Visualizzare il numero di vertici ed archi così ottenuti.

```
select gds1.Product_number, gds2.Product_number
from go_daily_sales gds1, go_daily_sales gds2,
(select gds.Product_number, sum(gds.Unit_sale_price*gds.Quantity) as tot
      from go_daily_sales gds
      where year(gds.Date) = 2017 and
            gds.Order_method_code = 1
      group by gds.Product_number) as tot_income1,
      (select gdsa.Product_number, sum(gdsa.Unit_sale_price*gdsa.Quantity) as tota
      from go_daily_sales gdsa
      where year(gdsa.Date) = 2017 and
            gdsa.Order_method_code = 1
      group by gdsa.Product_number) as tot_income2
where gds1.Product_number = tot_income1.Product_number and
      gds2.Product_number = tot_income2.Product_number and
      gds1.Product_number <> gds2.Product_number and
      tot_income2.tota >= 1.6* tot_income1.tot
group by gds1.Product_number, gds2.Product_number
```

c. 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 squad

d. Il peso di ciascun arco del grafo deve corrispondere alla somma dei salari dei giocatori delle nell'anno considerato.

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

```
select t1.Id, t2.ID, s1.tot + s2.tot as weight
from teams t1, teams t2, (select teamID, sum(salary) as tot
                          from salaries a
                          where a.year = 2015
                          group by teamID) s1,
                          (select teamID, sum(salary) as tot
                          from salaries b
                          where b.year = 2015
                          group by teamID) s2
where t1.ID < t2.ID and
      s1.teamID = t1.ID and
```

```
        s2.teamID = t2.ID
group by t1.Id, t2.ID
```

b. Alla pressione del bottone “Crea Grafo”, creare un grafo semplice, **non orientato**, **pesato**, in cui i vertici siano gli **anni** in cui tale squadra ha giocato.

c. Il grafo dovrà essere **completo**, e per ciascuna coppia di anni y1, y2, il **peso** dell’arco che li connette dovrà essere pari alla **differenza in valore assoluto** del **salario cumulativo** delle due squadre (somma dei salari di tutti i **giocatori sotto contratto** con la squadra, rispettivamente nell’anno **y1** e nell’anno **y2**).

   NOTA: un giocatore è **sotto contratto** con una squadra in un certo anno, se in quell’anno la suddetta squadra corrisponde un **salario** al giocatore.

Qury che non sono riuscita a fare, per tema **2023-06-30-B.pdf (scaricato)**

IN MODEL:

```
        //Leggere i giocatori per ogni anno
        this.annoToPlayers.clear();
        for (int anno : vertici) {
            this.annoToPlayers.put(anno, this.dao.getPlayersSalaryTeamYear(name, anno));
        }

        //verificare ogni coppia di anni, e creare un arco con il peso corrispondente
        for (int i = 0; i <vertici.size(); i++) {
            for (int j = i+1; j < vertici.size(); j++) {
                List<PeopleSalary> giocatori1 = new ArrayList<PeopleSalary>(this.annoToPlayers.get(vertici.get(i)));
                List<PeopleSalary> giocatori2 = new ArrayList<PeopleSalary>(this.annoToPlayers.get(vertici.get(j)));

                double peso = Math.abs(getCumulativeSalary(giocatori1) - getCumulativeSalary(giocatori2));
                Graphs.addEdgeWithVertices(this.grafo, vertici.get(i), vertici.get(j), peso);
            }
        }
    }

   public double getCumulativeSalary(List<PeopleSalary> giocatori) {
        double salaryCum = 0;
        for (PeopleSalary ps : giocatori)
            salaryCum += ps.getSalary();

        return salaryCum;
   }
```

```java
public List<PeopleSalary> getPlayersSalaryTeamYear(String name, int anno){
        String sql = "SELECT people.*, salaries.salary, teams.name, teams.year " +
                        "FROM people, teams, salaries " +
                        "WHERE teams.year = ? " +
                        "AND teams.name = ? " +
                        "AND people.playerID=salaries.playerID " +
                        "AND salaries.teamID=teams.ID";

        List<PeopleSalary> result = new ArrayList<PeopleSalary>();
```

b. Alla pressione del bottone "Crea Grafo", creare un grafo **semplice, non orientato e pesato**, in cui:

- I vertici sono tutti e soli i prodotti distinti presenti nel database con colore ***c***
- Due vertici sono collegati tra loro da un arco, se e solo se nel corso dell'anno ***a*** specificato esiste almeno un retailer che abbia venduto **entrambi** i prodotti in uno **stesso giorno** (calcolato tenendo conto delle informazioni contenute in *go_daily_sales).* Il peso dell'arco indica il **numero di giorni distinti** dell'anno ***a*** in cui **entrambi** i prodotti sono stati venduti da un medesimo retailer.

  Esempio: supponiamo di avere scelto l'anno 2015 e consideriamo due vertici, P1 e P2. Supponiamo che il retailer R1 venda entrambi P1 e P2 nei giorni 2015-02-25 e 2015-06-11, mentre il retailer R2 li venda entrambi nei giorni 2015-06-11 e 2015-11-20. L'arco P1<->P2 esiste ed ha peso 3.
- Stampare:
  - *i.* Sull'area di testo con id *txtResGrafo* il numero di vertici ed archi
  - *ii.* Sull'area di testo con id *txtArchi*, i tre archi di peso maggiore (seguendo la convenzione **prodotto1, prodotto2, peso**). Tra i vertici di questi tre archi, stampare i prodotti che sono presenti in più di uno dei tre archi.
    Esempio: se i tre archi sono A <-> C, A<->B, C<->D, i prodotti da stampare sono A, C.

PUNTO 2

```sql
select gds1.Product_number, gds2.Product_number, count(distinct gds1.Date) as tot
from go_products gp1,  go_products gp2, go_daily_sales gds1 , go_daily_sales gds2
where gp1.Product_color = 'Red' and
      gp2.Product_color = 'Red' and
      gp1.Product_number = gds1.Product_number and
      gp2.Product_number = gds2.Product_number and
      gds1.Product_number < gds2.Product_number and
      gds1.Retailer_code = gds2.Retailer_code and
      gds1.`Date`  = gds2.`Date` and
      year(gds1.Date) = year(gds2.Date) and
      year(gds1.Date) = 2015
group by gds1.Product_number, gds2.Product_number
order by tot desc
```

QUI ERA INTERESSANTE COME VOLEVA CONFONDERTI SUL PESO DELL'ARCO. Osserva: gds1.Date = gds2.Date !!!

b. Alla pressione del bottone "Crea Grafo", creare un grafo **semplice, non orientato e pesato**, in cui:

- I vertici sono tutti e soli i rivenditori situati nella nazione ***n*** selezionata
- Due vertici sono collegati tra loro da un arco, se e solo se nel corso dell'anno abbiano venduto un numero di prodotti in comune (calcolato tenendo conto della colonna *Product_number* della tabella *go_daily_sales*) ***maggiore o uguale*** a ***M*** nel corso dell'anno ***a***. Il peso é pari al numero di prodotti con lo stesso Product_number venduto dai due rivenditori.
  Esempio: nell'anno *a*, il rivenditore A ha venduto i prodotti con id 111, 222, 333 ed il rivenditore ha venduto i prodotti con id 111, 333, 444; il numero di prodotti in comune che hanno venduto, cioè il peso, è 2. Questo numero non dipende dalla quantità di pezzi venduti.

```
select gds1.Retailer_code as u, gds2.Retailer_code as v, count(distinct
gds1.Product_number)
from go_daily_sales gds1, go_daily_sales gds2, go_retailers gr1, go_retailers gr2
where gr1.Country = 'France' and
      gr2.Country ='France' and
      gds1.Retailer_code < gds2.Retailer_code and
      gds1.Retailer_code = gr1.Retailer_code and
      gds2.Retailer_code = gr2.Retailer_code and
      year(gds1.`Date`) = 2017 and year(gds2.`Date`) = 2017 and
      gds1.Product_number = gds2.Product_number
group by u,v
having count(distinct gds1.Product_number) >= 3
```

TEMA UTILE X SOMMA PESI IN ARCHI BIDIREZIONALI!

b. Alla pressione del bottone, creare un grafo che rappresenti gli aeroporti su cui operano più compagnie.
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**).

```
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):    CRUCIALE

                    self._graph[e.aeroportoP][e.aeroportoD]["weight"] += e.peso

                else:

                    self._graph.add_edge(e.aeroportoP, e.aeroportoD, weight = e.peso)
```

```
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
                    """
```

#COALESCE(expr1, expr2, ..., exprN) restituisce il primo valore NON NULL tra quelli passati.

È una funzione di gestione dei NULL.

Se expr1 è NULL, passa a expr2, e così via fino a trovare un valore.

```
@staticmethod
def getAllNodes(year):
                conn = DBConnect.get_connection()
                result = []
                cursor = conn.cursor(dictionary=True)
                query = &amp;quot;&amp;quot;&amp;quot;SELECT c.customer_id, c.first_name, c.last_name,
                COALESCE(SUM(oi.quantity), 0) AS totale_acquistato
                FROM customers c
                LEFT JOIN orders o ON c.customer_id = o.customer_id
                LEFT JOIN order_items oi ON o.order_id = oi.order_id
                GROUP BY c.customer_id&amp;quot;&amp;quot;&amp;quot;
                cursor.execute(query, (year,))
                for row in cursor:
                result.append(Node(**row))
                cursor.close()
                conn.close()
                return result
```

prendi le squadre costruttori che hanno vinto il campionato

```
@staticmethod

def getMenu11():

        conn = DBConnect.get_connection()

        result = []

        cursor = conn.cursor(dictionary=True)

        query = &amp;quot;&amp;quot;&amp;quot;WITH ultimi_round AS (

        SELECT year, MAX(round) AS max_round

        FROM races

        GROUP BY year

        )

        SELECT DISTINCT

        r.year AS anno,

        cs.constructorId AS constructorId

        FROM races r

        JOIN constructorStandings cs

        ON r.raceId = cs.raceId

        JOIN ultimi_round ur

        ON r.year = ur.year AND r.round = ur.max_round

        WHERE cs.position = 1&amp;quot;&amp;quot;&amp;quot;

        cursor.execute(query)

        for row in cursor:

        result.append(Oggetto(**row))

        cursor.close()

        conn.close()
```

Due piloti sono collegati se hanno condiviso nella stessa stagione la stessa posizione finale in

gara

```
# il peso è il numero di occorrenze

                @staticmethod

                def getAllEdges2(year):

                conn = DBConnect.get_connection()

                result = []

                cursor = conn.cursor(dictionary=True)

                query = &amp;quot;&amp;quot;&amp;quot;select least(r1.driverId, r2.driverId) as driverId1,
                greatest(r1.driverId, r2.driverId) as

                driverId2, count(*) as peso

                from results r1, results r2, races ra1, races ra2

                where r1.driverId != r2.driverId and r1.position = r2.position AND r1.position IS
                NOT

                null and

                ra1.raceId = r1.raceId and ra2.raceId = r2.raceId and ra1.year = ra2.year and

                ra1.year = %s

                group by driverId1, driverId2&amp;quot;&amp;quot;&amp;quot;

                cursor.execute(query, (year,))

                # for row in cursor:

                # result.append(Edge2(**row))

                cursor.close()

                conn.close()

                return result

# Due piloti sono collegati da quello ha fatto più sorpassi verso chi ne ha subiti in
una

determinata stagione

# il peso è il numero di occorrenze

        @staticmethod

        def getAllEdges7():

        conn = DBConnect.get_connection()

        result = []

        cursor = conn.cursor(dictionary=True)
```

```
query = &quot;&quot;&quot;WITH sorpassi AS (
SELECT r1.driverId AS sorpassante, r2.driverId AS sorpassato
FROM races r, results r1
JOIN results r2 ON r1.raceId = r2.raceId
WHERE r1.grid &gt; r2.grid AND r1.position &lt; r2.position and r.raceId = r1.raceId and
r.year = 2000
)
SELECT sorpassante, sorpassato, COUNT(*) AS peso
FROM sorpassi
GROUP BY sorpassante, sorpassato&quot;&quot;&quot;
cursor.execute(query)
```

Due costruttori sono legati da chi ha avuto più punti verso chi ne ha avuti meno nella stessa

gara

 # il peso è il numero di occorrenze

```
@staticmethod
def getAllEdges10():
conn = DBConnect.get_connection()
result = []
cursor = conn.cursor(dictionary=True)
query = &quot;&quot;&quot;SELECT
cs1.constructorId AS vincente,
cs2.constructorId AS sconfitto,

COUNT(*) AS peso
FROM constructorstandings cs1
JOIN constructorstandings cs2
ON cs1.raceId = cs2.raceId AND cs1.constructorId &lt;&gt; cs2.constructorId
WHERE cs1.points &gt; cs2.points
```

```
        GROUP BY vincente, sconfitto&amp;quot;&amp;quot;&amp;quot;

        return result
```

TEMI DA SVOLGERE:
2024-07-18-A
2024-07-04-C
2024-07-04-B
2024-07-04-A – ok

ESERCIZIO: Un **arco** collega due geni *diversi* se e solo se i due geni hanno la **stessa Localizzazione** (tabella *classification*),
GeneID diverso, ed **esiste una interazione tra di loro** (ovvero sia la tabella *Interactions* contiene una interazione con i GeneID dei due nodi)
Il **peso** dell'arco corrisponde all'indice di correlazione dell'interazione fra i due geni (tabella *interactions*).
Il **verso** dell'arco sarà uscente dal gene con Cromosoma minore ed entrante nel gene con Cromosoma

maggiore. Il caso in cui due geni appartengono allo stesso cromosoma va gestito aggiungendo entrambi gli archi.

```
def build_graph(self, ch_min, ch_max):
    self._graph.clear()
    nodes = DAO.get_nodes(ch_min, ch_max)
    self._graph.add_nodes_from(nodes)

    for i in range(len(nodes)-1):
        for j in range(i+1, len(nodes)):
            if (self.get_localization_gene(nodes[i]) == self.get_localization_gene(nodes[j]) and
                nodes[i].GeneID != nodes[j].GeneID and
                (nodes[i].GeneID, nodes[j].GeneID) in self._correlations_map):
                peso = self._correlations_map[(nodes[i].GeneID, nodes[j].GeneID)]
                if nodes[i].Chromosome < nodes[j].Chromosome:
                    self._graph.add_edge(nodes[i], nodes[j], weight=peso)
                elif nodes[i].Chromosome > nodes[j].Chromosome:
                    self._graph.add_edge(nodes[j], nodes[i], weight=peso)
                else:
                    self._graph.add_edge(nodes[i], nodes[j], weight=peso)
                    self._graph.add_edge(nodes[j], nodes[i], weight=peso)
```

ESERCIZIO: Dato il grafo costruito al punto precedente, si vuole identificare un percorso su grafo tale per cui, dato un vertice di partenza (selezionato dall'apposita tendina), si identifichi il percorso più lungo in termini di numero di archi, composto da archi dal peso sempre crescente.

```
def trovaPercorso(self, partenza):
    parziale = [self._idMap[partenza]]
    self.sol_best = []
    self.best_value = 0

    viciniAmmissibili = self.viciniAmmissibili(parziale)
    for el in viciniAmmissibili:
        parziale.append(el)
        self._ricorsione(parziale)
        parziale.pop()

    return len(self.sol_best)-1


def _ricorsione(self,parziale):
    viciniAmmissibili = self.viciniAmmissibili(parziale)
    if (len(viciniAmmissibili) == 0):
        if len(parziale) > len(self.sol_best):
            self.sol_best = copy.deepcopy(parziale)
            return

    for el in viciniAmmissibili:
        parziale.append(el)
        self._ricorsione(parziale)
        parziale.pop()



def viciniAmmissibili(self, seq):
    vicini = self._graph.neighbors(seq[-1])
    res = []

    for vicino in vicini:
        if(len(seq)>=2):
            if (vicino not in seq and
                    self._graph[seq[-1]][vicino]['weight'] >= self._graph[seq[-2]][seq[-1]]['weight']):
                res.append(vicino)
        else:
            if vicino not in seq:
                res.append(vicino)

    return res
```

ESERCIZIO: Dato il grafo costruito al punto precedente, si vuole identificare un percorso semplice e chiuso a peso massimo
composto da esattamente N archi. Il valore di N deve essere inserito dall'utente tramite il campo apposito nell'interfaccia grafica. N deve essere almeno 2. A tal fine si identifichi la sequenza di vertici con le seguenti caratteristiche:

- Il primo e l'ultimo vertice della sequenza devono coincidere.
- I vertici intermedi non devono essere ripetuti
- La somma dei pesi degli archi percorsi deve essere massima.

```
    def best_path(self, n_max):
        parziale = []

        for node in self._graph.nodes():
            parziale.append(node)
            self._ricorsione(parziale, n_max)
            parziale.pop()

        return self._best, self._peso_best

    def _ricorsione(self, parziale, n_max):
```

```
    if len(parziale) == n_max+1 and parziale[0] == parziale[-1]:  #se sono alla max
lunghezza e il nodo iniziale e finale coincidono
        costo_attuale = self.calcola_costo(parziale)
        if costo_attuale>self._peso_best:
            self._best = copy.deepcopy(parziale)
            self._peso_best = costo_attuale
        return

    for el in self._graph.neighbors(parziale[-1]):
        if (len(parziale) == n_max and el == parziale[0]) or (len(parziale) < n_max and el
not in parziale):
            parziale.append(el)
            self._ricorsione(parziale, n_max) #entro direttamente nella cond di
terminazione
            parziale.pop()


def calcola_costo(self, seq):
    tot = 0
    for i in range(len(seq)-1):
        tot +=self._graph[seq[i]][seq[i+1]]['weight']

    return tot

def get_peso_arco(self, u,v):
    return self._graph[u][v]['weight']
```

RICERCHE SU GRAFI:

!! PRIMA DI CERCARE LA COMPONENTE CONNESSA ASSICURATI CHE IL GRAFO POSSEGGA QUEL NODO

Nx.MultiGrap() se non ho grafo semplice

Nx.MultiDiGrap() se ho un multigrafo diretto

https://networkx.org/documentation/stable/reference/classes/index.html

funzioni predefinite

PRENDO ARCHI INCIDENTI A UN NODO

(caso grafo non orientato)

- **archi = self._graph.edges(node, data=True)** serve a **ottenere tutti gli archi (edges)** connessi a un nodo specifico in un grafo, **includendo anche i dati associati a quegli archi**.

(caso grafo orientato)

- **out = self._graph.out_edges(node, data=True)** prendo gli archi uscenti da un grafo orientato
- **en = self._graph.in_edges(node, data=True)** prendo gli archi entranti da un grafo orientato

PRENDO I VICINI DI UN NODO

(caso grafo non orientato)

- **graph.neighbors(node):** tutti i nodi adiacenti

(caso grafo orientato)

- **graph.successors(node):** nodi raggiunti da node tramite un arco uscente
- **graph.predecessors(node):** nodi da cui posso arrivare a node

ORDINAMENTI

- **volumi.sort(key=lambda x:x[1], reverse=True)** ordino una lista di tuple in maniera decrescente sulla base del secondo valore

FUNZIONI "RICERCA"

- **tree = nx.dfs_tree(self._graph, p)** restituisce un **albero orientato** costruito seguendo una **visita in profondità** partendo da p. Il grafo di partenza può essere orientato o meno, ma il ritorno sarà sempre orientato.

CAMMINO MINIMO

(valide per ENTRAMBI I CASI)

- **path = nx.shortest_path(G, source=source, target=target)**: Se vuoi il cammino più breve IN TERMINI DI NUMERO DI ARCHI **da un nodo source a un nodo target specifico**. E' una lista contentente i nodi nell'ordine in cui devi visitarli per andare da source a target lungo il cammino più breve.

- **path = nx.shortest_path(G, source=source, target=target, weight='weight')**: Se vuoi il cammino più breve USANDO I PESI **da un nodo source a un nodo target specifico**. E' una lista contentente i nodi nell'ordine in cui devi visitarli per andare da source a target lungo il cammino più breve.

- **paths = nx.single_source_shortest_path(G, source)**: Se vuoi i cammini più brevi breve IN TERMINI DI NUMERO DI ARCHI da source a **tutti i nodi raggiungibili**. paths sarà un dizionario: con chiave = nodo raggiunto, valore = lista di nodi che costituisce il percorso più breve da source a quel nodo
- **paths = nx.single_source_shortest_path(G, source, weight='weight')**: Se vuoi i cammini più brevi breve IN TERMINI DI PESI da source a **tutti i nodi raggiungibili**. paths sarà un dizionario: chiave = nodo raggiunto,
  valore = lista di nodi che costituisce il percorso più breve da source a quel nodo

(caso

-

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

Camm minimo da tutti i nodi a tutti i nodi: dijkstra ripetuto

(Dijkstra ripetuto)

```
distanze = {}

for nodo in G.nodes:
    distanze[nodo] = nx.single_source_dijkstra_path_length(G, nodo)

# Stampa le distanze minime da ogni nodo
for sorgente, mappa in distanze.items():
    print(f"Da {sorgente}: {mappa}")
```

(caso grafo non orientato)

RENDI IL GRAFO EULERIANO AGGIUNGENDO IL MINIMO NUMERO DI ARCHI: eulerize(graph)

algoritmi utili

TROVA COMPONENTE CONNESSA:

(caso grafo non orientato)

```
def getNumCompConnesse(self):

    # S = [self._graph.subgraph(c).copy() for c in nx.connected_components(self._graph)]
#this is to get all the components

    return nx.number_connected_components(self._graph)
```

- **nx.connected_components** restituisce tutte le componenti connesse di un grafo non orientato, ogni componente è un **insieme di nodi** che sono **mutuamente raggiungibili** tra loro ma **non sono connessi** a nodi di altre componenti

Operazione	Risultato	Cosa contiene	Utilità
`list(nx.connected_components(G))`	Lista di `set`	Solo nodi	Leggero, ma poco usabile
`[G.subgraph(c).copy() for c in ...]`	Lista di `Graph`	Nodi + archi + attributi	Pronto per analisi, disegno, modifica

-

(caso grafo orientato)

- **nx.number_weakly_connected_components(self._graph)** Numero di componenti **debolmente connesse** (ignora direzione archi)

- **nx.number_strongly_connected_components(self._graph)** Numero di componenti **fortemente connesse** (seguendo la direzione)

- **nx.weakly_connected_components(G)** Restituisce un **generatore** di insiemi di nodi, dove ogni insieme rappresenta una **componente connessa** del grafo.

- **nx.strongly_connected_components(G)**

TROVA LA COMPONENTE CONNESSA PIU GRANDE

- **max_component = max(nx.connected_components(G), key=len)** PER TROVARE TUTTI I NODI DELLA COMPONENTE CONNESSA PIU GRANDE

- **S = [self._graph.subgraph(c).copy() for c in nx.connected_components(self._graph)]**
  +
  max_subgraph = max(S, key=lambda g: g.number_of_nodes()) TROVARE TUTTO IL SOTTOGRAFO DELLA COMPONENTE CONNESSA PIU GRANDE

SOMMA DEI PESI DI TUTTI GLI ARCHI INCIDENTI A UN NODO – ordinato per valore decrescente

```
volumi = []

for node in self._graph.nodes():
    volume = 0
    vicini = list(nx.neighbors(self._graph, node))

    for vicino in vicini:
        volume += self._graph[node][vicino]['weight']

    volumi.append((node, volume))
```

```
    volumi.sort(key=lambda x:x[1], reverse=True)
return volumi
```

CAMINO PIU' LUNGO PARTENDO DA UN NODO

```
def cercaPercorso(self, partenza):
        p = self._idMap[(int(partenza))]
        tree = nx.dfs_tree(self._graph, p)
        nodes = list(tree.nodes())
        sol = []
        for node in nodes:
            temp = [node]

            while (temp[0]!= p):
                pred = nx.predecessor(tree, p, temp[0])
                temp.insert(0, pred[0])

            if len(temp) > len(sol):
                sol = copy.deepcopy(temp)

        return sol
```

I CINQUE NODI CON IL MAGGIORE NUMERO DI ARCHI USCENTI ED IL PESO COMPLESSIVO DI QUESTI ARCHI:

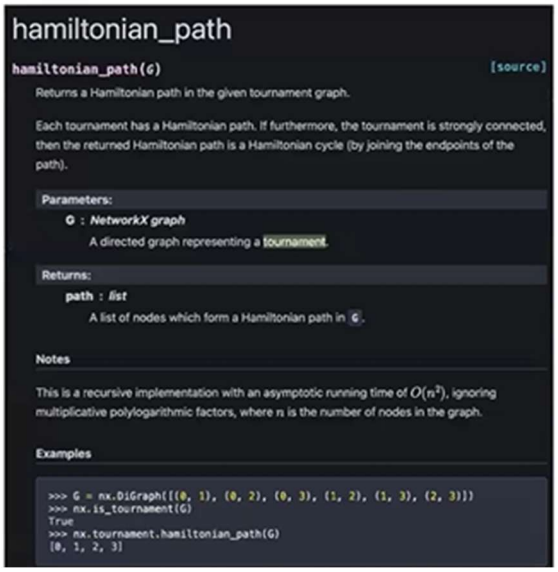
```
def get_node_max_uscenti(self):
        sorted_nodes = sorted(self._graph.nodes(), key=lambda n: self._graph.out_degree(n),
    reverse=True)
        result = []
        for i in range(min(len(sorted_nodes), 5)):
            peso_tot = 0.0
            for e in self._graph.out_edges(sorted_nodes[i], data=True):
                peso_tot += float(e[2].get("weight"))
            result.append((sorted_nodes[i], self._graph.out_degree(sorted_nodes[i]), peso_tot))
```

return result

---

uso tutti I vertici -> ciclo hamiltoniano (travelling salesman problem)

uso tutti i nodi -> ciclo euleriano, uso l'algoritm odi hierholzer SE TUTTI I NODI HANNO UN NUMERO PARI DI ARCHI



hamiltonian_path

hamiltonian_path(G) [source]

Returns a Hamiltonian path in the given tournament graph.

Each tournament has a Hamiltonian path. If furthermore, the tournament is strongly connected, then the returned Hamiltonian path is a Hamiltonian cycle (by joining the endpoints of the path).

Parameters:

G : *NetworkX graph*

A directed graph representing a tournament.

Returns:

path : *list*

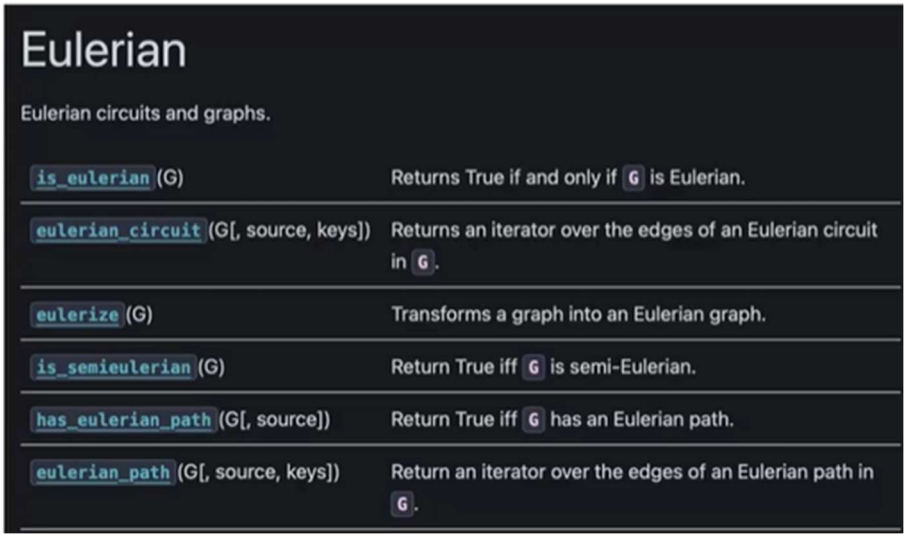
A list of nodes which form a Hamiltonian path in `G`.

Notes

This is a recursive implementation with an asymptotic running time of $O(n^2)$, ignoring multiplicative polylogarithmic factors, where n is the number of nodes in the graph.

Examples

```
>>> G = nx.DiGraph([(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)])
>>> nx.is_tournament(G)
True
>>> nx.tournament.hamiltonian_path(G)
[0, 1, 2, 3]
```



Eulerian

Eulerian circuits and graphs.

`is_eulerian`(G)	Returns True if and only if `G` is Eulerian.
`eulerian_circuit`(G[, source, keys])	Returns an iterator over the edges of an Eulerian circuit in `G`.
`eulerize`(G)	Transforms a graph into an Eulerian graph.
`is_semieulerian`(G)	Return True iff `G` is semi-Eulerian.
`has_eulerian_path`(G[, source])	Return True iff `G` has an Eulerian path.
`eulerian_path`(G[, source, keys])	Return an iterator over the edges of an Eulerian path in `G`.

RICORSIONE

ESERCIZIO: Qui ho un grafo non pesato, voglio che i vertici che aggiungo in parziale abbiano duration sempre crescente. Il cammino può contenere al massimo 3 avvistamenti dello stesso mese. un arco può essere percorso solo nella sua direzione, ovvero un arco diretto da A verso B non può essere percorso da B ad A. punteggio composto dai seguenti termini:

- +100 punti per ogni avvistamento nel cammino
- +200 punti per ogni avvistamento del cammino che è occorso nello stesso mese dell'avvistamento precedente (ovviamente non applicabile al primo avvistamento del cammino, dato che non ha un avvistamento che lo precede).

```
def _ricorsione(self, parziale):

    if(len(self._viciniAmmissibili(parziale)) == 0):
        if self._costo_best < self.calcolaCosto(parziale):
            self._best = copy.deepcopy(parziale)
            self._costo_best = self.calcolaCosto(parziale)
        return


    for node in self._viciniAmmissibili(parziale):
        parziale.append(node)
        self._ricorsione(parziale)
        parziale.pop()


def _viciniAmmissibili(self, parziale):
    amm = []
    vicini = self._graph.successors(parziale[-1])

    for vicino in vicini:
        if vicino not in parziale:
            cnt = 0
            for i,el in enumerate(parziale):
                if el.datetime.month == vicino.datetime.month:
                    cnt += 1

            if cnt < 3:
                if len(parziale) > 0:
                    if parziale[-1].duration < vicino.duration:
                        amm.append(vicino)

                else:
                    amm.append(vicino)

    print(f"Vicini di {parziale[-1]}: {amm}")
    return amm
```

```
def calcolaCosto(self, parziale):
    costo = 0
    for i,el in enumerate(parziale):

        if i > 0:
            if parziale[i-1].datetime.month==parziale[i].datetime.month:
                costo += 300
            else:
                costo += 100
        else:
            costo += 100
    return costo
```