METODI UTILI TdP

- Salvare nei menù a tendina dropdown degli oggetti

```
def fillDDNodi(self, listOfNodes):
    for n in listOfNodes:
        self._view._ddNode.options.append(ft.dropdown.Option(key=n.order_id, data=n,
on_click=self.getSelectedNode))
    self._view.update_page()

def getSelectedNode(self, e):
    if e.control.data is None:
        print("error in reading DD Nodes")
        self._choiceDDNodi = None
    self._choiceDDNodi = e.control.data
```

- Costruito il grafo, visualizzare il cammino più lungo partendo da un nodo sorgente, indicato dall'utente.

```
def getCamminoMax(self, source):
    longest_path = []
    tree = nx.dfs_tree(self._graph, source)  # albero DFS a partire dal nodo source
    nodi = list(tree.nodes())  # tutti i nodi visitati nella DFS
    for node in nodi:
        tmp = [node]  # inizio un cammino che termina nel nodo attuale
        while tmp[0] != source:
            pred = nx.predecessor(tree, source, tmp[0])  # Trova il predecessore di tmp[0]
            tmp.insert(0, pred[0])  # Aggiungi il predecessore all'inizio del cammino
        if len(tmp) > len(longest_path):  # Se il cammino corrente è più lungo del precedente
massimo
            longest_path = copy.deepcopy(tmp)  # Salva il nuovo cammino più lungo
    return longest_path
```

- Metodo @staticmethod DAO

```
@staticmethod
def getNodes(anno, idMap):
    conn = DBConnect.get_connection()
    cursor = conn.cursor(dictionary=True)
    result = []
    query = """select distinct(rs.driverId)
               from results rs, (select *
                                 from races r
                                 where r.year=%s) t
               where t.raceId = rs.raceId and rs.`position` is not
null"""
    cursor.execute(query, (anno, ))
    for row in cursor:
        result.append(idMap[row["driverId"]])
    cursor.close()
    conn.close()
    return result
```

- la dimensione della componente connessa a cui appartiene a1 e la durata complessiva (in minuti) di tutti gli album appartenenti alla componente connessa di a1.

```
def getInfoConnessa(self, a1):
    cc = nx.node_connected_component(self._grafo, a1)
    return len(cc), self._getDurataTot(cc)

def _getDurataTot(self, listOfNodes):
    durataTot = 0
    for n in listOfNodes:
        durataTot += n.dTot
    return durataTot
    # return sum([n.dTot for n in listOfNodes])
```

oppure altri metodi

```
def getInfoConnessa(self, idInput):
    """
    Identifica la componente connessa che contiene idInput e ne restituisce la dimensione
    """
    # DFS è molto utile per trovare la componente connessa
    if not self.hasNode(idInput):  # controllo ridondante perchè già fatto nel controller
        return None

    source = self._idMap[idInput]  # identifico l'oggetto associato alla chiave passata, che diventa il nodo sorgente

    # Modo 1: conto i successori
    succ = nx.dfs_successors(self._graph, source)
    res = []
    for s in succ.values():  # itero sui valori associati alle chiavi
        res.extend(s)  # se la riga è un oggetto, mi aggiunge un oggetto, se il valore è una lista allora mi aggiunge tutti gli elementi della lista
    print("Size connessa con modo 1:", len(res))  # mi dà tutti i nodi della componente connessa. succ è un dizionario. Per i successori: per ogni nodo ho un lista di nodi a cui posso arrivare associata
    # dovrei aggiungere 1 alla return perchè devo aggiungere il nodo sorgente

    # Modo 2: conto i predecessori
    pred = nx.dfs_predecessors(self._graph, source)  # pred è un dizionario, per ogni nodo come chiave mi dà l'oggetto da cui arrivo
    print("Size connessa con modo 2:", len(pred.values()))  # dovrei aggiungere 1 al return, per includere il nodo source

    # Modo 3: conto i nodi nell'albero di visita
    dfsTree = nx.dfs_tree(self._graph, source)
```

```
    print("Size connessa con modo 3:", len(dfsTree.nodes()))  # funziona come return

    # Modo 4: uso il metodo node_connected_component di nx
    conn = nx.node_connected_component(self._graph, source)  # mi trova direttamente i
nodi della componente connessa partendo dal nodo source del grafo
    print("Size connessa con modo 4: ", len(conn))
    return len(conn)  # funziona come return
```

- Permettere all'utente di selezionare, attraverso un menu a tendina, una delle squadre esistenti nel grafo, ed alla pressione del bottone "Dettagli" stampare per tale squadra l'elenco delle squadre adiacenti, ed il peso degli archi corrispondenti, in ordine decrescente di peso.

```
def getNeighborsSorted(self, source):
    vicini = nx.neighbors(self._graph, source)  # lista di nodi [v0,
v1, v2, ...]
    # vicini = self._graph.neighbors(source)
    viciniTuple = []  # lista di tuple [(v0, p0), (v1, p1), ...]
    for v in vicini:
        viciniTuple.append((v, self._graph[source][v]['weight']))
    viciniTuple.sort(key=lambda x: x[1], reverse=True)  # ordino in
ordine decrescente i pesi
    return viciniTuple
```

- Stampare il numero di componenti connesse nel grafo.

```
def getCompConnesse(self):
    conn = list(nx.connected_components(self._grafo))
    return len(conn)
```

- Partendo dal grafo ottenuto nel punto precedente, alla pressione del bottone "Ricorsione", si implementi una procedura ricorsiva che calcoli un percorso di peso massimo. Il vertice di partenza è quello selezionato nel punto 1.c e il peso degli archi nel percorso deve essere strettamente decrescente. → METODO RICORSIVO ESEMPIO

```
def getBestPathPesoMax(self, source):
    self.bestPath = []
    self.pesoMax = 0
    parziale = [source]
    esplorabili = list(self._graph.successors(source))
    self._ricorsione(parziale, esplorabili)
    return self.bestPath, self.pesoMax

def _ricorsione(self, parziale, esplorabili):
    if len(esplorabili) == 0:
        if self.getPeso(parziale) > self.pesoMax:
            self.bestPath = copy.deepcopy(parziale)
            self.pesoMax = self.getPeso(parziale)
    else:
        for n in esplorabili:
            if n not in parziale:
                parziale.append(n)
                nuovi_esplorabili = self.getEsplorabili(n, parziale)
```

```
                self._ricorsione(parziale, nuovi_esplorabili)
                parziale.pop()

def getPeso(self, parziale):
    pesoTot = 0
    for i in range(0, len(parziale) - 1):
        pesoTot += self._graph[parziale[i]][parziale[i +
1]]['weight']
    return pesoTot

def getEsplorabili(self, n, parziale):
    esplorabili = []
    for node in self._graph.successors(n):
        if node not in parziale and self._graph[parziale[-
1]][node]['weight'] < self._graph[parziale[-2]][parziale[-
1]]['weight']:
            esplorabili.append(node)
    return esplorabili
```

- Visitare il grafo partendo da tale stazione. Provare sia con la visita in ampiezza, che in profondità:

```
# visita del grafo
def getBFSNodesFromTree(self, source):  # visita in ampiezza, source è la fermata di partenza che inserisce l'utente
    tree = nx.bfs_tree(self._grafo, source)  # ritorna un albero orientato costruito a partire da source
    archi = tree.edges
    nodi = list(tree.nodes)
    return nodi[1:]  # escludo la source

def getDFSNodesFromTree(self, source):  # visita in profondità
    tree = nx.dfs_tree(self._grafo, source)
    nodi = list(tree.nodes)
    return nodi[1:]  # escludo la source

def getBFSNodesFromEdges(self, source):
    archi = nx.bfs_edges(self._grafo, source)  # mi restituisce direttamente gli archi ottenuto attraverso il metodo bfs
    res = []
    for u, v in archi:
        res.append(
            v)  # aggiungo il secondo elemento della tupla per prendere solo i nodi di arrivo dei vari archi, questo mi genera il cammino della ricerca BFS
    return res

def getDFSNodesFromEdges(self, source):
    archi = nx.dfs_edges(self._grafo,
                         source)  # mi restituisce direttamente gli archi ottenuto attraverso il metodo dfs
```

```
    res = []
    for u, v in archi:
        res.append(v)
    return res
```

- Date due fermate, calcolare il cammino più breve (utilizzando l'algoritmo di Dijkstra) e visualizzarlo.

```
def getShortestPath(self, u, v):
    return nx.single_source_dijkstra(self._grafo, u, v)  # restituisce distanza e percorso, come lista di nodi
```

- I 5 nodi col maggiore numero di archi uscenti col numero di archi uscenti ed il peso complessivo di questi archi (la somma dei loro pesi). I nodi devono essere stampati in ordine decrescente per numero di archi uscenti.

```
def getBestNodiArchiUscenti(self):
    diz = {}
    for node in self._graph.nodes:
        gradoUscente = self._graph.out_degree(node)
        diz[node] = gradoUscente
    dizSortato = sorted(diz.items(), key=lambda x: x[1], reverse=True)
    result = []
    for node, gradoUscente in dizSortato[:5]:
        pesoTot = 0.0
        for e in self._graph.out_edges(node, data=True):
            pesoTot += float(e[2]["weight"])
        result.append((node, gradoUscente, pesoTot))
    return result
```

- Stampare i 5 archi di peso maggiore, ordinati in ordine decrescente. Per ognuno di questi 5 archi stampare l'id del nodo di origine, l'id del nodo di destinazione, ed il peso.

```
def getPesiMaggiori(self):
    archi = list(self._graph.edges)
    archiConPeso = []
    for a in archi:
        archiConPeso.append((a[0].id, a[1].id, self._graph[a[0]][a[1]]["weight"]))  # aggiungo una tupla con (nodo1.id, nodo2.id, peso)
    archiConPeso.sort(key=lambda x: x[2], reverse=True)
    return archiConPeso[:5]
```

oppure

```
def get_top_edges(self):
    sorted_edges = sorted(self._grafo.edges(data=True), key=lambda edge: edge[2].get('weight'), reverse=True)
    return sorted_edges[0:5]
```

- Stampare il numero di componenti connesse. Inoltre, identificare la componente connessa di dimensione maggiore, e stamparne i nodi.

```
def getInfoComponentiConnesse(self):
    cc = list(nx.connected_components(self._graph))
    cc.sort(key=lambda x: len(x), reverse=True)  # il primo elemento è la componente connessa più grande
    return len(cc), list(cc[0])
```

- Stampare il numero di componenti debolmente connesse. Inoltre, identificare la componente connessa di dimensione maggiore, e stamparne i nodi.

```
def getInfoConnesseDeboli(self):
    cc = nx.number_weakly_connected_components(self._graph)
    return cc

def getLargestConnessaDebole(self):
    cc = list(nx.weakly_connected_components(self._graph))
    cc.sort(key=lambda x: len(x), reverse=True)
    return cc[0]
```

- Costruito il grafo, l'applicazione visualizza il pilota che ha totalizzato il miglior risultato, definito come differenza tra il numero di vittorie (archi uscenti) e di sconfitte (entranti).

```
def getBestDriver(self):
    bestDriver = (None, 0)
    degree_nodes = {}
    for n in self._graph.nodes():
        vittorie = 0
        for node in self._graph.successors(n):
            if self._graph.has_edge(n, node):
                vittorie += self._graph[n][node]["weight"]
            if self._graph.has_edge(node, n):
                vittorie -= self._graph[node][n]["weight"]
        degree_nodes[n] = vittorie
    for key, value in degree_nodes.items():
        # print(key, value)
        if value > bestDriver[1]:
            bestDriver = (key, value)
    return bestDriver
```