Schema ricorsioni

1) Cammino a peso massimo a partire da un nodo sorgente con peso degli archi decrescente:

```
def handleRicorsione(self,node_id):
    """restituisce il cammino a peso massimo a partire da un nodo sorgente
    con peso degli archi decrescente"""

    source = self._idMap[int(node_id)]

    self._memo = {} #salvo il risultato del cammino migliore
    # (peso massimo) , a partire da ciascun nodo. (salvo con la chiave = nodo, peso ;

    return self._ricorsione2(source,float('inf'))

def _ricorsione2(self, u, pesoPrecedente):
    #la chiave è data da nodo + peso
    key = (u, pesoPrecedente)
    #se la chiave è presente la ritorno
    if key in self._memo:
        return self._memo[key]

    best_path = [u]
    best_w = self.peso(best_path)

    for v in self._graph.successors(u):
        w_uv = self._graph[u][v]["weight"]
        #se l'arco ha peso minore del precedente posso aggiungerlo
        if w_uv < pesoPrecedente:
            #calcolo percorso con peso a partire dal nodo v
            cand_path, cand_w = self._ricorsione2(v, w_uv)
            total_w = w_uv + cand_w
            #se il peso è migliore del best lo sostituisco
            if total_w > best_w:
                best_w = total_w
                best_path = [u] + cand_path

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

2) *Restituisce il cammino più lungo (in numero di nodi) a partire dal nodo sorgente. utilizzo algoritmo dfs*

```
def handleCerca(self,node_id):
    """ Restituisce il cammino più lungo (in numero di nodi)
    a partire dal nodo sorgente. utilizzo algoritmo dfs"""
    source = self._idMap[node_id]


    self._memo = {} #serve a salvare il risultato del cammino migliore
    # da ciascun nodo
    for node in self._graph.successors(source):
        print(str(node))
    return self._ricorsione(source)


def _ricorsione(self,u):
    #se gia calcolato, restituisco best cammino
    if u in self._memo:
        return self._memo[u]
    #cammino migliore che parte da u (inizialmente solo [u])
    parziale = [u]
    #Esploro i successori
    for v in self._graph.successors(u):
        cand = self._ricorsione(v)
        #se aggiungendo v ottengo un cammino piu lungo, aggiorno
        if len(cand) + 1 > len(parziale):
            parziale = [u] + cand


    #Memorizzo e restituisco
    self._memo[u] = parziale
    return parziale
```

3) Restituisce percorso con peso massimo data una lunghezza richiesta (numero di nodi):

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

    parziale = [source]

    for n in self._graph.neighbors(source):
      #cosa in più, voglio che gli oggetti abbiano la stessa caratteristica (“classificazione”)
      if parziale[-0].classification == n.classification:
        parziale.append(n)
        self._ricorsione(parziale, lun)
        parziale.pop()

    return self._bestPath, self._bestCost

  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[-0].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
```

4) Identifica la componente connessa che contiene idInput e ne restituisce la dimensione (per un nx.graph) :

```
def getInfoConnessa(self, idInput):
    """
    Identifica la componente connessa che
    contiene idInput e ne restituisce la dimensione
    """
    if not self.hasNode(idInput):
      return None

    source = self._idMap[idInput]

    # Modo1: conto i successori
    succ = nx.dfs_successors(self._graph, source).values()
    res = []
    for s in succ:
      res.extend(s)
    print("Size connessa con modo 1: ", len(res))

    # Modo2: conto i predecessori
    pred = nx.dfs_predecessors(self._graph, source)
    print("Size connessa con modo 2: ", len(pred.values()))

    #Modo3: conto i nodi dell'albero di visita
    dfsTree = nx.dfs_tree(self._graph, source)
    print("Size connessa con modo 3: ", len(dfsTree.nodes()))

    #Modo4: uso il metodo nodes_connected_components di networkx
    conn = nx.node_connected_component(self._graph, source)
    print("Size connessa con modo 4: ", len(conn))

    return len(conn)
```

Dropdown, una volta selezionato un elemento, starta la funzione:

→ self._ddAnno = ft.Dropdown(label="Anno", width=200, alignment=ft.alignment.top_left,

on_change= lambda e: self._controller.handleCreaSquadre(e))

Riempire il DropdownOption

#opzione 1 :

→ options = map(lambda x : ft.dropdown.Option(x), self._model.getAllYears())
self._view._ddAnno.options = options

#opzione 2 :

→ self._view._ddAlbum.options = [ft.dropdown.Option(text=str(x),key = x.Albumid) for x in self._model._nodes]

##Ordinare una lista di oggetti secondo una loro caratteristica, (es. peso)

#ordine crescente :

→elements.sort(key=lambda frutto: frutto.peso)

#ordine decrescente :

→elements.sort(key=lambda frutto: frutto.peso, reverse=True)

#creare una nuova lista senza modificare l'ordine della prima :

→ordinati = sorted(elements, key=lambda frutto: frutto.peso)

##Cammino minimo

→nx.shortest_path(G, source, target, weight=None)

##Componente connessa a partire da un nodo in un grafo orientato (BFS o DFS)

#opzione 1 :

→edges = list(nx.bfs_edges(G, sources)

raggiungibili = set([source] + [v for u,v in edges])

#opzione 2 (source escluso) :

→raggiungibili = nx.descendants(G, source)

##Componente fortemente connessa (grafo orientato; es. A →B→C→A, posso tornare ad A)

→nx.strongly_connected_components(G)

##Componente debolmente connessa (es. ho solo A→B→C)

→nx.weakly_connected_components(G)