Schema ricorsioni

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

```
def handleRicorsione(self, node_id):
  """Serve per calcolare il percorso a peso massimo a partire da un nodo source, ogni nodo puo esserci una sola
     vol ta,  il peso degli archi deve essere decrescente"""
  source = self._idMap[int(node_id
  parziale = [source]  # Inizializza il percorso parziale con il nodo sorgente
  for node in self._graph.successors(source):
    parziale.append(node
    self._ricorsione(parziale
    parziale.pop
  return self._solBest, self._pesoBest  # Restituisce il percorso e il peso migliori trovati

def _ricorsione(self, parziale):
  peso = self.peso(parziale)  # Calcola il peso del percorso parziale
  if peso > self._pesoBest:  # Se il peso corrente è maggiore del peso migliore trovato finora
    self._pesoBest = peso  # Aggiorna il peso migliore
    self._solBest = copy.deepcopy(parziale)  # Aggiorna il percorso migliore

  for n in self._graph.successors(parziale[-1]):  # Itera sui successori dell'ultimo nodo nel percorso parziale
    edge = self._graph[parziale[-1]][n]['weight']  # Ottiene il peso dell'arco corrente
    if edge < self._graph[parziale[2]][parziale[1]]['weight']
      and n not in parziale:  # Verifica le condizioni di peso decrescente e unicità del nodo

      parziale.append(n
      self._ricorsione(parziale)
      parziale.pop

def peso(self, lista):
  peso = 0  # Inizializza il peso a 0
  for i in range(len(lista) - 1):
    peso += self._graph[lista[i]][lista[i + 1]]['weight']
  return peso  # Restituisce il peso totale del percorso
```

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

```
def getCammino(self, sourceStr):

    source = self._idMap[int(sourceStr)]
    lp = []

    tree = nx.dfs_tree(self._graph, source)
    nodi = list(tree.nodes())

    for node in nodi:
      tmp = [node]

      while tmp != source:
        pred = nx.predecessor(tree, source, tmp)
        tmp.insert(0, pred)

      if len(tmp) > len(lp):
        lp = copy.deepcopy(tmp)

    return lp
```

- Per ogni nodo nell'albero DFS:
  - tmp = [node]: Inizia un percorso temporaneo con il nodo corrente.
  - **Ciclo while**: Continua a inserire i predecessori del nodo corrente fino a raggiungere il nodo sorgente.
  - if len(tmp) > len(lp): Se il percorso temporaneo è più lungo del percorso massimo attuale, aggiorna il percorso massimo.
- return lp: Restituisce il percorso più lungo trovato.

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)