DESCRIZIONE CONTENUTI LAIB
Programmazione Avanzata

- **Laib 06:** non presente neanche il DAO, utilizzo della funzione cursor.fetchone( ) che restituisce la prima riga disponibile

- **Laib 07:** centrato sul popolamento di due dropdown i cui dati sono ottenuti da un DAO basilare(SELECT … FROM …).
  Popolamento dropdown nel controller:

```
self._view.dropdown.options.clear()
self._view.dropdown.options.append(
                                    ft.dropdown.Option(None,"Nessun
                                    Filtro"))
lista_valori = self.model.get_valori()
if lista_valori:
    for valore in lista_valori:
        self._view.dropdown_museo.options.append(
                                    ft.dropdown.Option(museo.nome))
else:
    self._view.show_alert("Errore nel caricamento dei musei.")
```

  Database utilizzato: *musei_torino.sql*

- **Laib 08:** argomento principale è la ricorsione.
  Interessante perché ti chiede di analizzare i primi 7 giorni del mese e utilizza anche una data estraendo il mese così: *c.data.month == mese.*
  Calcola il consumo medio giornaliero:
  sum(consumi_mese)/len(consumi_mese)
  La ricorsione ha come parametri: parziale, giorno, ultimo_impianto(perché se si cambia impianto si impone una tassa di 5$), costo_corrente e consumi_settimana (calcolati nella funzione in basso con un dizionario con id → valori per i primi 7 giorni del mese).
  Caratteristiche da segnalare:
  consumi_mese.sort(key=lambda c: c.data) → ordinamento per DATA
  consumi_settimana[impianto.id] = [c.kwh for c in consumi_mese[:7]] → estraggo solo i primi 7 *id→kwh.*
  Database utilizzato: *gestione_energia.sql*

- **Laib 09:** secondo laib sulla *ricorsione*.

Dao sempre basilare, a parte l'uso di dizionari per mappare tour e attrazioni (query sempre e solo SELECT … FROM …).
La ricorsione si basa sulla ricerca del valore culturale massimo associato alle attrazioni del tour.
In questo caso, i vincoli della ricorsione sono che:

1. Tour tutti diversi tra loro;
2. Assenti attrazioni duplicate;
3. Durata_pacchetto < Durata_massima indicata in giorni dall'utente;
4. Costo_pacchetto < Budget indicato dall'utente.

Database utilizzato: *archivio_tour.sql*

- **Laib 10:** grafo semplice PESATO,
  utilizzo di GREATEST(origine, destinazione) e LEAST() che prendono il più grande e il più piccolo dei parametri inseriti tra parentesi, SUM per calcolare il totale, COUNT(*) per calcolare il numero di spedizioni,
  GROUP BY (hub1, hub2) per raggruppare a tupla.
  Utilizzo di un set( ) per creare i nodi del grafo e un if tratta.media >= soglia per aggiungere gli archi e il peso sopra una soglia.
  Tramite le funzione self.G.number_of_nodes() e self.G.number_of_edges() restituisce il numero di nodi e di archi.

```
tratte = []
for u, v, data in self.G.edges(data=True):
   tratte.append((u, v, data["peso"]))
```

  NB al self.G.edges(data = True):
  In questo modo restituisco tutte le tratte con i pesi nel model in una lista (tratte_superiori) che poi confronto nel controller così:

```
for u, v, peso in tratte_superiori:
   self._view.lista_visualizzazione.controls.append(
       ft.Text(f"Tratta {u} - {v} ha una media =
{peso:.2f} €")
   )

self._view.update()
```

  Con questo ciclo for u, v, peso in tratte_superiori riesco a stampare ogni tratta presente come arco

  Database utilizzato: logistics_network.sql

- **Laib 11:** grafo SEMPLICE, NO orientato, NO pesato
  Utilizzo del metodo lista che poi inserisco all'interno di un dizionario.
  Che insieme alla lista di connessioni formo gli archi tra id_rifugio1 e id_rifugio2. Connessione presente già nei database

```
self._rifugi_list = DAO.readRifugi(year)
self.connessioni = DAO.readConnessioni(year)
self._rifugi_dict = {r.id: r for r in self._rifugi_list}
for c in self.connessioni:
   self.G.add_edge(c.id_rifugio1, c.id_rifugio2, id =
c.id)
```

  Utilizzo del dizionario per poter utilizzare gli ID come chiave così posso arrivare facilmente ai valori.
  Restituisco il numero di vicini utilizzando la funzione

```
node_id = node.id
if self.G.has_node(node_id): #se self.G ha come nodo
(node_id)
   return self.G.degree(node_id)
```

  Raccolgo l'id dal nodo (un rifugio) e controllo faccia parte, poi con la funzione self.G.degree(node_id) restituisco il numero di vicini al nodo all'interno del grafo.

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

  Restituisce il numero di componenti connesse al grafo
  componente_connessa= list(nx.node_connected_component(self.G, nodo))

  tree = nx.dfs_tree(self.G, start.id)
  Restituisce un albero che contiene il nodo di partenza e tutti i nodi raggiungibili da esso

```
id_vicini = list(tree.nodes)
id_vicini.remove(start.id)
```

  Converto gli ID con i nomi
  Prende la chiave del dizionario tramite i e aggiunge ad una lista tutti i nomi dei rifugi raggiungibili

```
for i in id_vicini:
   raggiungibili.append(self._rifugi_dict[i]) #NB
utilizzo del dizionario fondamentale così id chiave e
faccio uscire il nome
return raggiungibili #lista con tutti i rifugi vicini
```

Nel controller invece è presente una fill_dropdown() che popola i dropdown con i rifugi presenti nel grafo. NB ricordarsi di updatare la pagina!!
E un callback chiamato quando si seleziona un'opzione nel dropdown chiamata read_dd_rifugio
Presente una soglia in anni che taglia fino all'anno dato nella listview

Database utilizzato: mountain paths

- **Laib 12:** Stesso corpo del laib 11, grafo SEMPLICE PESATO
Come per il laib 11 presente la soglia in anni, ma anche presente una soglia legata al peso degli archi.

Viene eseguito un calcolo tramite un ciclo for c in connessioni: in cui si eseguono operazioni sul valore che in seguito viene aggiunto come peso (gia modificato) in aggiunta anche dell'attributo id

```
self.G.add_edge(c.id_rifugio1, c.id_rifugio2, id = c.id, weight = peso)
```

Tramite questo ciclo for e con il metodo data.get('weight') raccolgo il peso e ne calcolo il massimo e minimo come una variabile normale

```
for u,v,data in self.G.edges(data=True):#cosi prendo il dato diretto per ogni edge
    peso = data.get('weight')
```

Da questo si calcolano numero archi sopra e sotto la soglia.

```
if peso < soglia:
    min_soglia = min_soglia + 1
if peso > soglia:
    max_soglia = max_soglia +1
```

E per la ricerca del cammino minimo utilizzo

```
percorso = nx.shortest_path(SubG, source=nodo_start, target=nodo_end, weight="weight")
```

nx.shortest_path(grafo, nodo di inizio, nodo di arrivo, peso) trova il percorso minimo tra i due nodi.

```
for k in range(len(percorso) - 1):
    u = percorso[k]
    v = percorso[k + 1]
    peso += SubG[u][v]["weight"]
```

Con questo ciclo for prendo il percorso minimo e ne sommo i pesi
Nel controller utilizzo lo stesso ciclo for per far uscire poi i nomi e le località e stampare

Database utilizzato: mountain paths

- **Laib 13 SE:** grafo semplice, PESATO e ORIENTATO
  richiama query del DAO direttamente nel model in _init_ e carica tutti i dati all inizializzazione.
  crea dizionario: self.id_map = {}
  for g in self._lista_geni:
  self.id_map[g.id] = g.cromosoma
  implementa funzione che restituisce (nodo1, nodo2, attributi):
  def get_edges(self):
  return list(self.**G.edges(data=True)**)

  implementa funzione per calcolare pesi massimi/minimi:
  def get_min_weight(self):
  return min([x[2]['weight'] for x in self.get_edges()])

  def get_max_weight(self):
  return max([x[2]['weight'] for x in self.get_edges()])

  io avevo implementato la precedente funzione nel seguente modo:

```
def get_edges_weight_min_max(self):
 peso_min = float('inf')
 peso_max = float('-inf')
 for u, v, data in self.G.edges(data=True):
     weight = data.get('weight', 1)
     if weight > peso_max:
         peso_max = weight
     if weight < peso_min:
         peso_min = weight
 return peso_min, peso_max
```

  nella query sql per ricavare connessione tra nodi del grafo sdoppia stessa tabella e ottiene direttamente dalla query le coppie tra i nodi:
  SELECT g1.id AS gene1, g2.id AS gene2, i.correlazione
  FROM gene g1, gene g2, interazione

  la ricorsione cerca percorso dal costo massimo

  Database utilizzato: genes_small.sql

- **SE_BikeStore:** grafo DIREZIONALE, PESATO
  Query non semplice per tirare fuori le vendite con l'utilizzo del COUNT per calcolare le vendite totali e l'utilizzo del comando BETWEEN per cercare tra due date. Inoltre è stato implementato dai borsisti dei datepickers, con presenza delle date da utilizzare.
  Dropdown da riempire, però semplice senza implicare altro.

Funzione set_dates implementata da loro utile per lavorare con le date
Creazione del grafo utilizzando il doppio for, essendo un grafo direzionale bisogna capire la differenza tra nodo entrante e uscente. Inoltre presente il calcolo del peso tra i due nodi dell'arco

Nell'handler del grafo presente come stampare una data, numero nodi e num archi. Per prendere i 5 prodotti più venduti utilizzo la funzione calcola_peso_nodo() che utilizza le funzioni

```
for arco_out in self.G.out_edges(product_id, data=True):
    valore = valore + arco_out[2]["weight"]

for arco_in in self.G.in_edges(product_id, data=True):
    valore = valore - arco_in[2]["weight"]

return valore
```

Che equivale a ovvero i nodi la cui somma dei pesi degli archi uscenti meno la somma dei pesi degli archi entranti è massima.

Per poterli gestire e farli uscire in modo corretto nella GUI ho dovuto raccogliere

```
grafo = self._model.crea_grafo(categoria_selezionata,
data_inizio, data_fine)
for prodotto in grafo:
    valore = self._model.calcola_peso_nodo(prodotto)
    valori[prodotto] = valore
sorted_valori = sorted(valori.items(), key=lambda x: x[1],
reverse=True)
```

Ho creato un dizionario per i valori con chiave id e peso il valore, poi le ho messe in ordine con il metodo sorted.

Ho raccolto gli id in una lista e ho creato un dizionario per i prodotti, sennò non si poteva stampare i nomi in output essendo di una tabella diversa

```
sorted_valori_id = []
for i in range (len(sorted_valori)):
    sorted_valori_id.append(sorted_valori[i][0])

prodotti_dict = {}
for i in range(len(prodotti)):
    prodotti_dict[prodotti[i].id] = prodotti[i].product_name
```

Infine ho uninto nel print il tutto e utilizzato un count per prendere le prime 5

```
for prodotto in sorted_valori_id:
    if prodotto in prodotti_dict and count < 5:
        count += 1
        self._view.txt_risultato.controls.append(
```

```
            ft.Text(f"{prodotti_dict[prodotto]} with score
{valori[prodotto]}")
        )
self._view.update()
```

Nella ricorsione si chiede di implementare cammino da nodo1 a nodo2 con lunghezza L (ovvero len(nodi_cammino) == L):
if len(parziale) == lungh:
    if parziale[-1] == end and self._get_score(parziale) > self.best_score:

database utilizzato: bike_store_full.sql

- **SE_iTunes:** grafo semplice, non orientato e non pesato
  Query complessa, utilizzo degli IN per creare gli archi tra due album. Utilizzo di due track e due playlist track perché dovevano essere uguali playlist ma diverse le track.
  Grafo semplice se fatta la query difficile.
  Caricamento di una dropdown album che però deve aspettare la soglia dei minuti, quindi caricamento gia nel grafo. (NB a questa cosa, dropdown implicata da un textfield)
  Utilizzo della funzione componente connessa con la funzione:
  list(nx.node_connected_component(self.G, album))

  ```
  componente = list(nx.node_connected_component(self.G,
  int(album)))
  ```

  che crea una lista di id delle componenti connesse
  Per poi unirle nel controller ho dovuto creare un dizionario e trovare la durata totale
  `album_dict = {a.id: a for a in album_list}`
  Programma semplice ma fondamentale fare la query annidata bene sennò molto complicato

  La ricorsione ottiene una lista che includa il maggior numero possibile di album la cui durata complessiva è minore di DTot:
  new_duration = current_duration + album.duration
      if new_duration <= max_duration:
          current_set.append(album)
          self._ricorsione(albums, current_set, new_duration, max_duration)
          current_set.pop()

  Database utilizzato: iTunes.sql

- **SE_Baseball:** grafo semplice, non orientato e pesato.
  Query nel DAO abbastanza basiche, evidenzio l'uso di query parametrizzata e SUM(salary) per definire il salario totale.
  Nel model carica dal DAO tutti i vari dati utili con funzioni del tipo *load_dato().*
  Usa la funzione ***enumerate(lista)*** per creare archi tra squadre diverse (enumerate restituisce l'indice e il valore associato a tale indice nella lista di squadre).
Calcola i pesi delle squadre adiacenti ad una squadra selezionata dall'utente e ordina la lista di tuple [(squadra, peso)] per peso decrescente in questo modo:

```
sorted(vicini, key=lambda x: x[1], reverse=True)
```

!!! Il calcolo di questi pesi decrescenti sarà utile nel punto 2.

La ricorsione si basa sul calcolo di un percorso dal peso massimo dove

1. il punto di partenza è la squadra selezionata dall'utente per la ricerca delle squadre adiacenti;
2. ogni vertice compare solo una volta;
3. il peso degli archi è decrescente;
4. posso esplorare solo i K archi collegati al vertice per limitare il tempo di esecuzione.

Database utilizzato: lamhansbaseballdb_small.sql

- **SE_Ufo:** grafo semplice, pesato e non orientato
  peso degli archi si ottiene come somma avvistamenti nodo1 + nodo2
  nel primo punto si chiede di popolare due dropdown con 2 query

  nella query per ricavare peso archi si utilizza LEAST e GREATEST
  LEAST(n.state1, n.state2) AS st1,
  GREATEST(n.state1, n.state2) AS st2

  e si ricava l anno con YEAR(s_datetime) in sql
  mentre in python un datetime si tratta come s_datetime.year / month / day

  NB: per verificare su sql che un valore non sia nullo si usa IS NOT NULL, mentre in python diventa is not None

  nel DAO popola la classe usando:
  for row in cursor:
  result.append(Sighting(**row))

  ricorsione richiede percorso semplice che massimizza la distanza tra stati con archi di peso sempre crescente. utilizza la distanza geodesica

  Database utilizzato: ufo_sightings