UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

GuidaIA

PROGETTO CORSO DI INGEGNERIA DELLA CONOSCENZA

A.A. 2021-2022

Studente: Fabio Loguercio

(Matr. 706472 – Mail: f.loguercio1@studenti.uniba.it)

Repository GitHub:

https://github.com/logfabio/knowledge-engineering-project.git

Introduzione:

L'obiettivo di questo progetto è quello di creare una guida supporto per un turista che vuole visitare la città di Gravina in Puglia.
La guida offre la possibilità di:

- riconoscere un'attrazione turistica da una semplice foto, ricevendo così poi una piccola descrizione e l'indicazione di dove si trovi sulla mappa;
- elaborare il percorso più breve per raggiungere Gravina da qualsiasi località nei dintorni, con l'elenco dei nomi delle città da attraversare e dei km totali da percorrere;
- ricevere, effettuando delle query, varie informazioni come ad esempio nomi di ristoranti o b&b nelle vicinanze di un'attrazione turistica o viceversa, percorsi di itinerari turistici con prezzo e durata ecc…

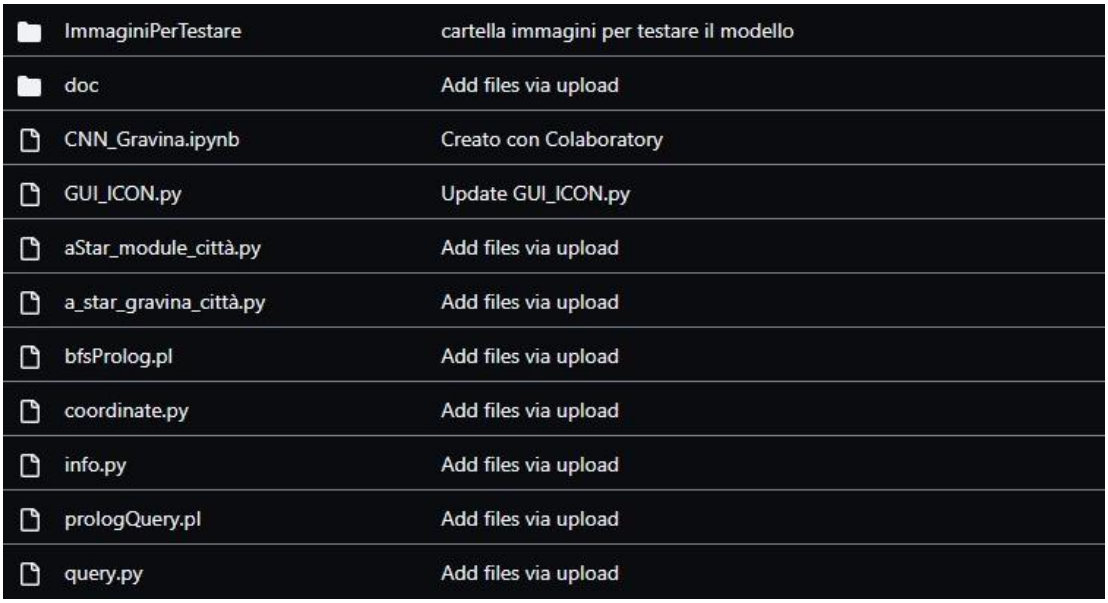
Strumenti utilizzati:

Il linguaggio utilizzato per sviluppare il progetto è stato Python, utilizzando *Spyder* come IDE e *Google Colab* per creare il modello. Le principali librerie utilizzate sono:

- *Keras*, libreria open source utile per l'apprendimento automatico e la costruzione di reti neurali. Supporta vari motori di back-en, quindi non esegue le proprie operazioni di basso livello ma si affida ad una libreria specializzata come ad esempio TensorFlow. Keras consente una prototipazione facile e veloce, supporta sia reti convoluzionali (CNN) che reti ricorrenti (RNN) o combinazioni di entrambi. Offre moduli utili per organizzare differenti livelli, il principale modello è quello sequenziale, ovvero una pila lineare di livelli.
- *PySwip*, che consente di interrogare SWI-Prolog nei programmi Python, necessaria per costruire e interrogare la Knowledge Base;

- *Requests* e *BeautifulSoup,* che consentono rispettivamente di effettuare in modo semplice richieste http e di compiere web scraping per la lettura di dati presenti nelle pagine web o documenti html;
- *TKinter,* utile per implementare interfacce grafiche (GUI) utilizzando il linguaggio python.

Contenuto del repository:

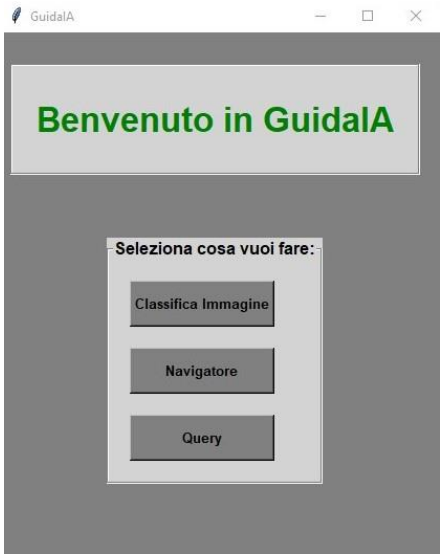


ImmaginiPerTestare	cartella immagini per testare il modello
doc	Add files via upload
CNN_Gravina.ipynb	Creato con Colaboratory
GUI_ICON.py	Update GUI_ICON.py
aStar_module_città.py	Add files via upload
a_star_gravina_città.py	Add files via upload
bfsProlog.pl	Add files via upload
coordinate.py	Add files via upload
info.py	Add files via upload
prologQuery.pl	Add files via upload
query.py	Add files via upload

- La cartella “ImmaginiPerTestare” contiene le immagini da utilizzare per testare il modello nel programma;
- La cartella “doc” contiene la relazione del progetto;
- Il file “CNN_Gravina” è il notebook colab utilizzato per creare il modello;
- Il file “GUI_ICON” implementa l’interfaccia utente, quindi svolge il compito di main;
- Il file “aStar_module_città” implementa l’algoritmo A*;
- Il file “aStar_gravina_città” implementa il grafo delle città, necessario per l’algoritmo A*;
- Il file “bfsProlog” è il file prolog in cui è implementato l’algoritmo BFS;
- Nel file “coordinate” ci sono le coordinate delle città utilizzate per visualizzarle sulla mappa;
- Il file “info” implementa le richieste http per ricevere la descrizione delle attrazioni turistiche;
- Il file “prologQuery” è il file prolog in cui è implementata la KB che viene interrogata per le varie query;
- Il file “query” implementa le query per interrogare la KB.

Dettagli implementativi:

All'avvio del programma la finestra principale si presenterà in questo modo, con i tre bottoni dei task principali:



1. **Classificatore di immagini**:

Il modello creato è una *Rete Neurale Convoluzionale (CNN)* utilizzata nel riconoscere le immagini e restituire la categoria di appartenenza. Le CNN sono formate da tre layer principali:

- Convolutional Layer: è il componente più importante di una CNN perché è dove avviene la maggior parte del calcolo. Richiede dati di input, un filtro e una feature map. Il filtro, detto anche kernel, viene applicato sull'immagine cioè moltiplicando i valori del filtro con i pixel, poi questi prodotti vengono sommati, andando così a creare una feature map. Dopo ogni operazione di convoluzione viene applicata una funzione non lineare, la più utilizzata è la ReLU che consiste nel sostituire tutti valori negativi della feature map con zero.
- Pooling Layer: operazione utilizzata per ridurre la dimensionalità di ogni feature map. I più utilizzati sono il max pooling, dove viene preso il massimo valore di un sottoinsieme nella feature map, e average pooling dove invece viene effettuata la media. (di solito ci sono diversi round di convoluzione e pooling).
- Fully-Connected Layer: il risultato dei livelli precedenti alimentano la struttura della rete completamente connessa che guida la decisione finale di classificazione. Sono formate da diversi layer composti da un certo numero di neuroni, tutti connessi con quelli del layer successivo. L'ultimo layer restituisce la classe di appartenenza con la relativa probabilità.

In questo progetto non avendo molti dati su cui addestrare il modello, ho utilizzato il processo di *fine tuning* di un modello di rete convoluzionale pre addestrato (il modello utilizzato è VGG16, addestrato sul dataset ImageNet). Con il fine tuning viene sostituito soltanto l'ultimo layer di classificazione, lasciando intatti i layer in grado di estrarre informazioni.

L'importazione del modello VGG16 viene fatto semplicemente grazie alla libreria keras:

```
[ ]  vgg16 = VGG16(weights="imagenet", include_top=False, input_shape=(IMAGE_SIZE, IMAGE_SIZE, 3))
```

Poi si aggiungono i layer fully-connected che si occuperanno appunto della classificazione:

```
[ ]  model = Sequential()

     model.add(vgg16)

     model.add(Flatten())
     model.add(Dense(1024, activation="relu"))
     model.add(Dropout(0.5))
     model.add(Dense(17, activation="softmax"))

     vgg16.summary()
     model.summary()
```

Viene creato quindi un nuovo modello di tipo Sequential, a cui aggiungiamo tutti i layer di VGG16 e i nuovi layer Flatten (utile per appiattire tutti i valori in un unico vettore), Dense (che sono i fully connected) e Dropout (che in modo casuale, in questo caso il 50%, annulla le connessioni tra il primo livello di Dense e il secondo, utile per aumentare la generalizzazione del modello).

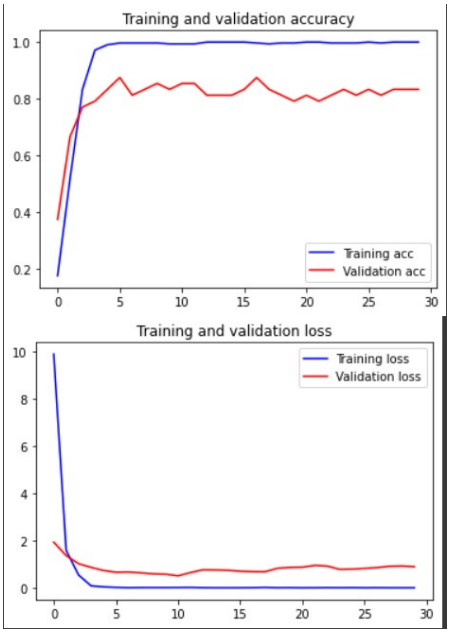
Poi vengono creati i generator del train e della validation (15% del dataset di train), che servono per caricare le immagini dal dataset. Il dataset è stato creato da me, dove per ogni attrazione turistica ci sono in media 20 immagini:

```
[ ]  train_batchsize = 34
     val_batchsize = 8

     train_generator = train_datagen.flow_from_directory(
             train_folder,
             target_size=(IMAGE_SIZE, IMAGE_SIZE),
             batch_size=train_batchsize,
             class_mode="categorical",
             subset='training'
     )

     val_generator = train_datagen.flow_from_directory(
             train_folder,
             target_size=(IMAGE_SIZE, IMAGE_SIZE),
             batch_size=val_batchsize,
             class_mode="categorical",
             shuffle=False,
             subset='validation'
     )
```

Quindi ora possiamo far partire il training del modello:

```
[ ]  history = model.fit(
            train_generator,
            epochs=num_epochs,
            validation_data=val_generator,
            verbose=1,
            callbacks=callbacks_list
        )
```

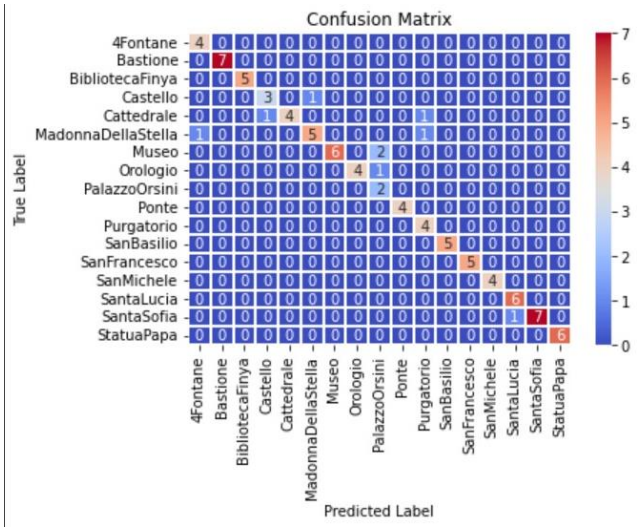
Ricevendo come risultati dell'accuracy e loss i seguenti valori:



Dopo aver creato il modello della rete, esso è stato testato con le immagini presenti nel testset del Dataset, dove per ogni attrazione ci sono in media 5 immagini:

```
[ ]  predictions = model.predict_generator(test_generator,
                                steps=test_generator.samples / test_generator.batch_size,
                                verbose=1)
     predicted_classes = np.argmax(predictions, axis=1)
```

Ricevendo come risultati questa matrice di confusione e i corrispettivi valori di precision, recall e f1:

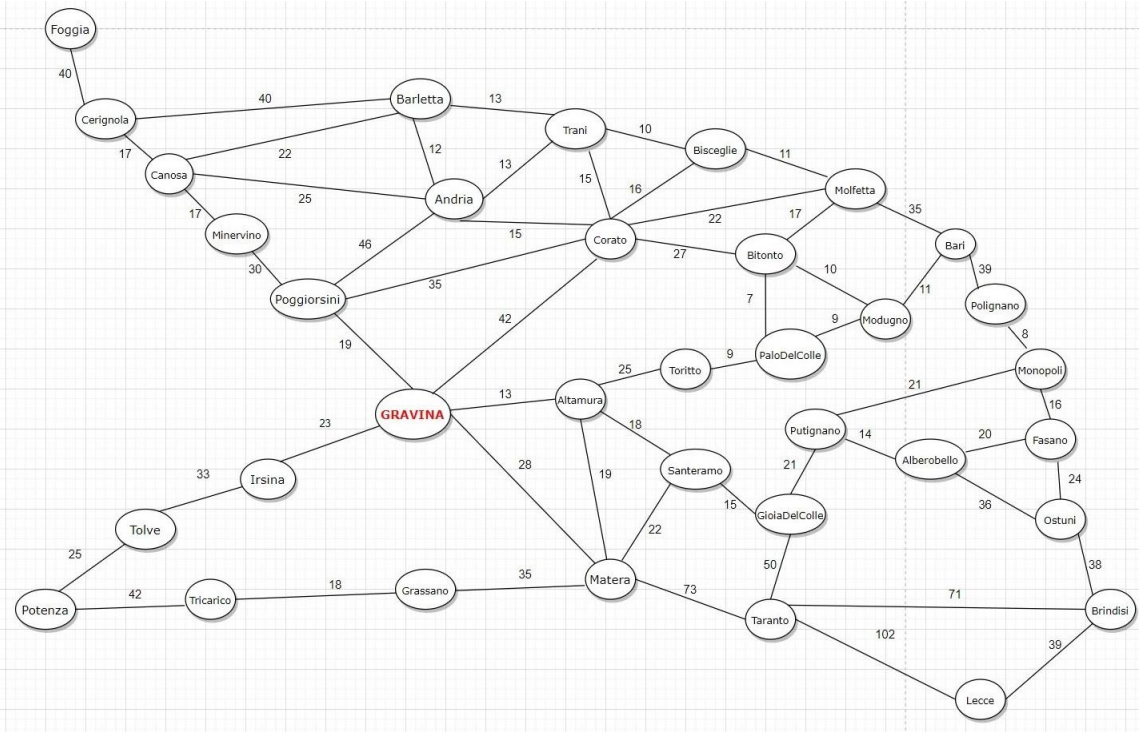


	precision	recall	f1-score	support
0	0.80	1.00	0.89	4
1	1.00	1.00	1.00	7
2	1.00	1.00	1.00	5
3	0.75	0.75	0.75	4
4	1.00	0.67	0.80	6
5	0.83	0.71	0.77	7
6	1.00	0.75	0.86	8
7	1.00	0.80	0.89	5
8	0.40	1.00	0.57	2
9	1.00	1.00	1.00	4
10	0.67	1.00	0.80	4
11	1.00	1.00	1.00	5
12	1.00	1.00	1.00	5
13	1.00	1.00	1.00	4
14	0.86	1.00	0.92	6
15	1.00	0.88	0.93	8
16	1.00	1.00	1.00	6
accuracy			0.90	90
macro avg	0.90	0.92	0.89	90
weighted avg	0.93	0.90	0.90	90

Per quanto riguarda l'interfaccia grafica, essa ti da la possibilità di caricare un'immagine da riconoscere e una volta fatto ciò, facendo click su appositi bottoni, si può visualizzare la posizione su mappa dell'attrazione turistica classificata (possibile grazie alla libreria TkinterMapView) e una piccola descrizione (ottenuta effettuando una richiesta http e utilizzando il web scraping per la lettura delle informazioni, possibile grazie alle librerie Requests e Bs4):

2. **Navigatore**:

Per questo task del progetto ho creato un grafo, dove ogni nodo rappresenta una città nei dintorni di Gravina e il peso degli archi rappresenta le rispettive distanze in km tra di loro:



Poi per effettuare la ricerca del percorso più breve con le rispettive città attraversate, ho deciso di utilizzare l'algoritmo *A*,* essendo un algoritmo di pathfinding, cioè in una rete di nodi consente di trovare il percorso ottimale che congiunge un nodo di partenza a un nodo di arrivo.

Alla base di questo algoritmo abbiamo due insiemi che contengono nodi:

- openSet: che contiene i nodi terminali dei tracciati parziali esaminati fino al momento corrente;
- closeSet: contiene i nodi intermedi dei tracciati già esaminati e quindi dai quali non si deve più passare;

Per ognuno dei nodi dell'openSet, al momento del loro inserimento in tale insieme, deve essere valutata una funzione di costo *f(n)* così definita:

$$f(n)=g(n)+h(n),\ \ n \in \text{openSet}$$

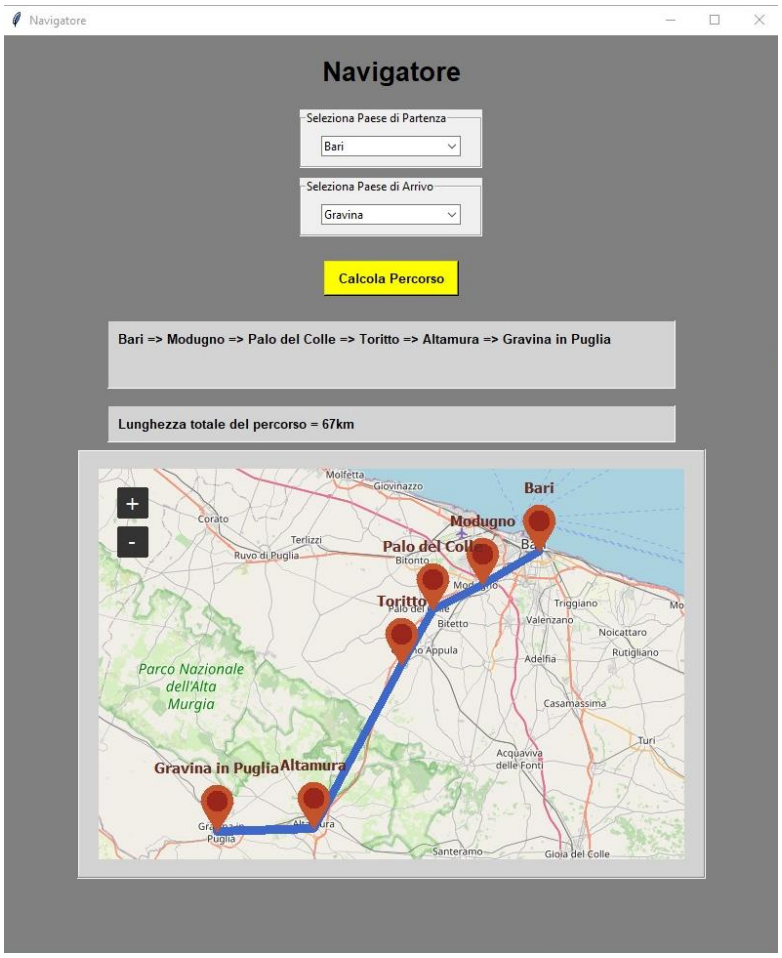
dove:

- *g(n)*: è una funzione che fornisce il costo necessario (nel mio caso la somma dei km sugli archi) per raggiungere il nodo *n* dal nodo scelto come partenza. Dipende dal percorso compiuto per raggiungere il nodo *n* e si presenta nella forma: *g(n)=g(n-1)+d(n-1,n)*, dove *n-1* rappresenta il nodo da cui si proviene, mentre *d(n-1,n)* è la distanza tra il nodo precedente ed *n*.

- *h(n)*: è la funzione euristica che fornisce una stima ottimale della distanza che separa il nodo *n* dal nodo di arrivo. Nel progetto è stata scelta come funzione euristica la distanza in linea d'aria.
- *f(n)*: è quindi una stima del costo associato al percorso che congiunge il nodo di partenza al nodo di arrivo passando per il nodo *n*.

Quindi ora si sceglie il nodo dell'openSet che ha il costo *f* minore, se l'openSet è vuoto il problema non ha soluzione. Il nodo che viene scelto si sposta nel closeSet e a questo punto vengono presi tutti i nodi raggiungibili dal nodo appena inserito nel closeSet, che non sono già presenti in esso, e li si aggiungono nell'openSet, salvando anche il nodo da cui proviene. Ora di questi nodi viene calcolata la funzione di costo e si ripete il tutto. Quando si ottiene un nodo che è già presente nell'openSet ma a cui si è giunti da un nodo differente, si valuta comunque la funzione di costo e se essa è maggiore del nodo già presente il nodo duplicato viene ignorato, altrimenti il nodo duplicato va a sostituire nell'openSet lo stesso nodo già presente ma raggiunto dal nodo precedente.
Il ciclo si arresta nel momento in cui si aggiunge all'openSet il nodo di arrivo che stavamo cercando e viene quindi ricostruito il percorso.

Per quanto riguarda l'interfaccia grafica, essa ti dà la possibilità di scegliere la città di partenza e di arrivo, quindi poi pigiando sul bottone viene calcolato il percorso, la distanza totale in km e viene data anche la possibilità di visualizzare il percorso sulla mappa:



3. **Query**:

Per quest'ultimo task sono state create due Knowledge Base (prologQuery.pl e bfsProlog.pl) formate da un insieme di clausole sottoforma di fatti, che dichiarano un certo stato di cose, e regole, che definiscono le relazioni fra i fatti. Per fare ciò è stato utilizzato il linguaggio di programmazione logica Prolog, usando il toolkit *SWI-Prolog.* Poi grazie alla libreria *PySwip* di Python è stato possibile interrogare le KB formulando delle query.

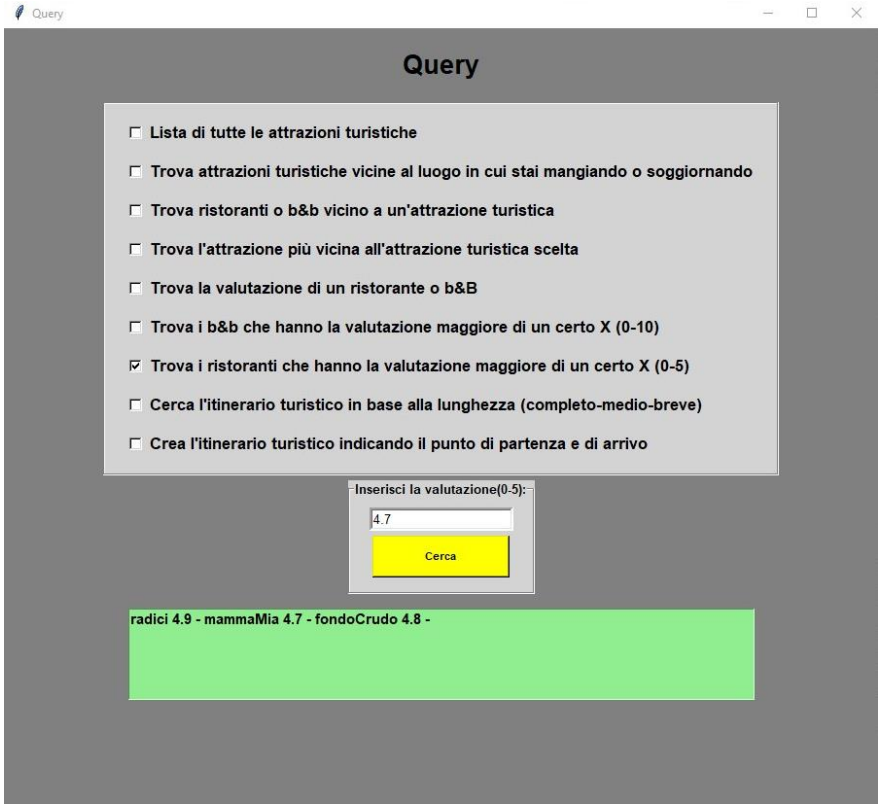
- Interrogando la prima KB "prologQuery.pl" è possibile ricavare varie informazioni, come ad esempio trovare i ristornati che hanno una valutazione maggiore di un certo valore fornito dall'utente:
  - La query eseguita grazie a PySwip è la seguente:

```
for val in prolog.query("searchTopR(Y,Z,"+rate+")"):
    answer=answer+str(val["Y"])+' '+str(val["Z"])+' - '
```

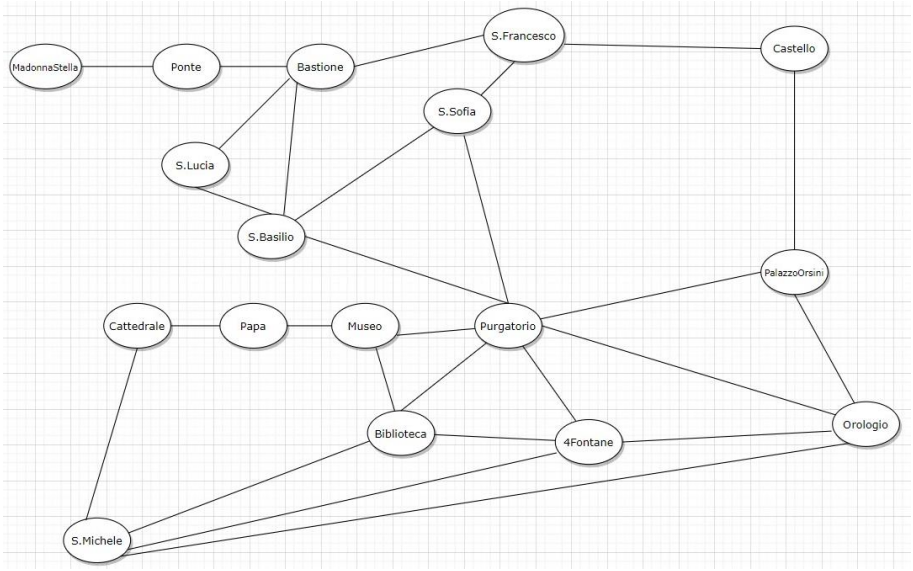
  - Dove la regola corrispondete in linguaggio Prolog è la seguente:

```
searchTopR(Y,Z,X):-
    restaurant(Y),
    rate(Y,Z),
    Z>=X.
```

  - Mentre l'interfaccia utente si presenterà in questo modo:



- Interrogando invece la seconda KB “bfsProlog.pl” è possibile ricavare l’itinerario turistico costruito in base al punto di partenza e di arrivo scelto dall’utente.
  Per fare ciò ho creato un grafo di tutte le attrazioni turistiche:

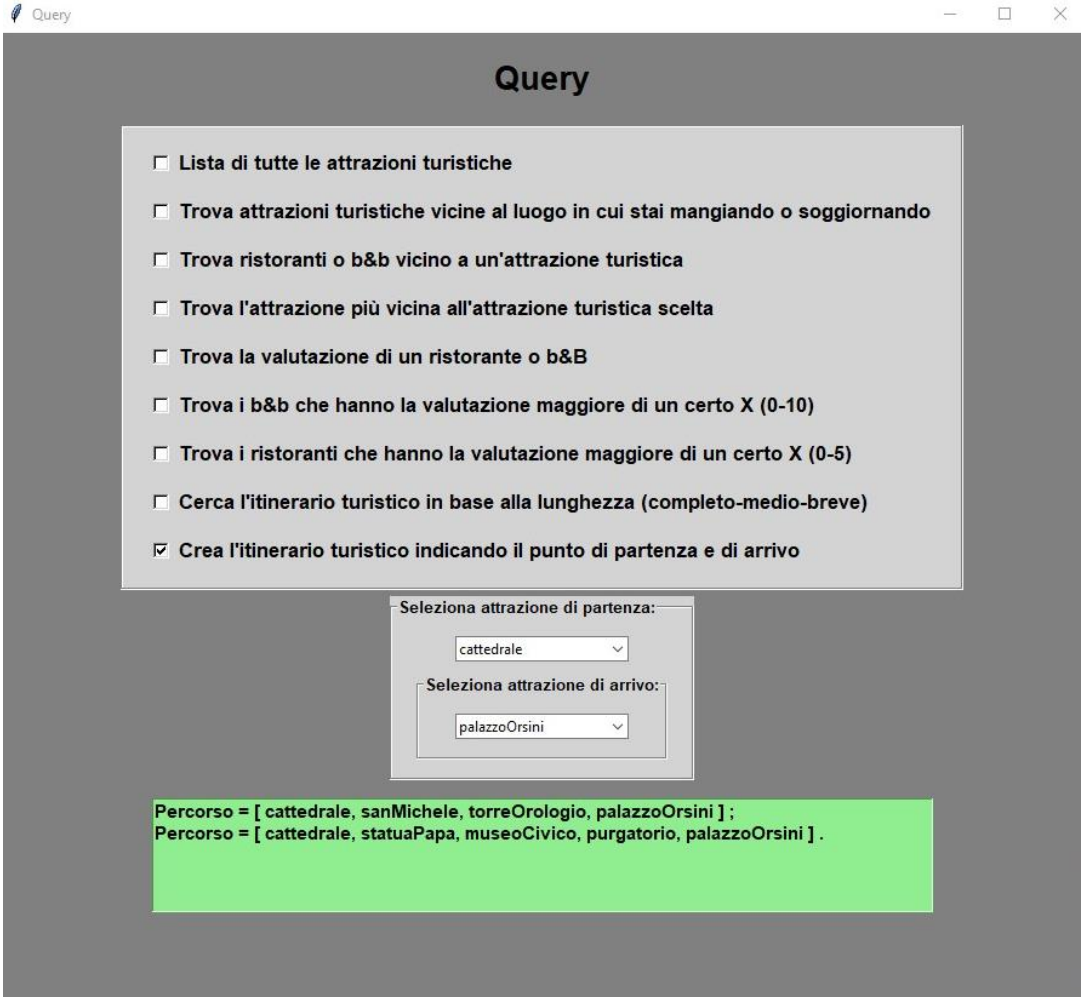


  Che ho poi costruito con dei fatti in Prolog e per effettuare la ricerca ho scelto l’algoritmo di ricerca in ampiezza Breadth-First Search sempre nel medesimo linguaggio.
  Il BFS è un algoritmo di attraversamento del grafo in cui parte da un nodo iniziale e attraversando tutto il grafo a strati, esplorando i nodi vicini, cerca di arrivare al nodo obiettivo.
  In questo algoritmo abbiamo una lista frontiera, implementata come una coda, dove vengono inseriti i nodi vicini al nodo che stiamo esaminando. Dalla frontiera viene esaminato il nodo in testa alla coda e se non è il nodo obiettivo, lo si espande prendendo i nodi vicini che vengono inseriti in fondo alla coda della lista frontiera.

- In questo caso per ricevere il percorso, viene prima inserito all’interno della KB il luogo di arrivo e poi viene eseguita la query:

```
prolog2.assertz('destinazione('+luogo_arrivo+')')

result=list(prolog2.query("cercaPercorso("+luogo_partenza+", Percorso)"))
```

- Mentre l’interfaccia utente permette di selezionare il punto di partenza e di arrivo e poi restituisce i primi due percorsi trovati:


Query
Query
Lista di tutte le attrazioni turistiche
Trova attrazioni turistiche vicine al luogo in cui stai mangiando o soggiornando
Trova ristoranti o b&b vicino a un'attrazione turistica
Trova l'attrazione più vicina all'attrazione turistica scelta
Trova la valutazione di un ristorante o b&B
Trova i b&b che hanno la valutazione maggiore di un certo X (0-10)
Trova i ristoranti che hanno la valutazione maggiore di un certo X (0-5)
Cerca l'itinerario turistico in base alla lunghezza (completo-medio-breve)
Crea l'itinerario turistico indicando il punto di partenza e di arrivo
Seleziona attrazione di partenza:
cattedrale
Seleziona attrazione di arrivo:
palazzoOrsini
Percorso = [cattedrale, sanMichele, torreOrologio, palazzoOrsini] ;
Percorso = [cattedrale, statuaPapa, museoCivico, purgatorio, palazzoOrsini] .