Prolog for Smart Tourism: KNN-Based Weather Inference and TSP-Based Route Optimization

Filippo di Gravina

Table of Contents

ABSTRACT

The project focuses on solving the Traveling Salesman Problem (TSP), a classic optimization problem, adding information related to the weather condition of some Italian cities and information about the state of the roads. For the nodes for which the weather is not available, a KNN is implemented in Prolog to infer the weather in those cities. We will then use the Branch and Bound algorithm in Prolog to find efficiently some solutions to this problem in two variants: considering the weather and state of the roads and without considering them. We will also implement visualization algorithms to graphically represent the results on a map, providing a clear understanding of the resulting solutions.

PROBLEM DEFINITION

The traveling salesman problem is an optimization problem, whose definition is: “Given a set of cities and the distances between each pair, the problem consists of finding the shortest route a traveling salesman must follow to visit all cities exactly once and return to the starting city”. In terms of graph theory, the traveling salesman problem can be defined as the search for a particular Hamiltonian cycle that minimizes a given parameter, such as the total distance traveled.

CONTEXT

The scenario we decided to work with is the following:

A tour operator wants to organize a tour of major Italian cities. The goal is to minimize such travel time. To do this, the tour operator must take into account a series of factors, such as:

- How bad is the road connecting two cities?
- What are the weather conditions?
- Are there any roadworks on the road from one city to the other?

These factors obviously contribute to increased travel time on extra-urban roads connecting cities.

ASSIGNING THE INFLUENCING VALUES

We assumed that the cities were connected to each other by an extra-urban road, with a travel speed of 90 km/h. This factor is obviously influenced by the conditions in which transportation takes place. The table below shows the influence each factor has on time:

	Coefficient			
Influencing factors	No	lieve	moderata	forte
Rain	1	1.1	1.2	1.3
Bumpy road	1	1.1	1.2	1.4

	entità	
Influencing factor	No	Si
Roadworks in progress	1	1.5

These influencing factors are assigned as follows:

- For the 90% of the nodes, the weather situation is randomly assigned.

- For the remaining nodes, we implemented a KNN in Prolog that, by studying the weather situation of the closest cities, assigns the most appropriate weather situation to the city.
- For edges, the worst-case weather situation of the two nodes involving it is assigned. For example, if one of the two cities experiences moderate rain and the other city experiences heavy rain, we consider heavy rain for the edge.
- At this point, the generated graph is used by our algorithms to search for a solution.

PROJECT IMPLEMENTATION

The logic of the project is divided into the following codes:

- Comuni.py
- Kb.pl
- ComuniGrafo.py
- Mappe.py
- Tsp_bb.pl

The first file, Comuni.py, generates a synthetic dataset to simulate travel and weather conditions between a subset of Italian cities. It begins by loading geographic data, i.e. latitude and longitude, for Italian municipalities from a CSV file and selects a random subset of cities based on a predefined list called cities.txt. For each pair of selected cities, the script calculates the distance using the Haversine formula, which measures the great-circle distance between two points on a sphere. These distances are written into a Prolog file, called predici.pl, as facts. Additionally, the script simulates weather conditions for 90% of the cities using a weighted random selection among four possible states: sunny, light rain, moderate rain, and heavy rain, while leaving the remaining 10% as unknown. Travel time between each city pair is computed assuming a constant speed limit of 90 km/h, and the resulting travel times are stored in the file dataset.csv. A separate file, called nomi.csv, is also generated to map city indices to their names.

The kb.pl code reads the facts from the “predici.pl” file. Specifically, the “predici.pl” file contains two types of facts:

- arco(Nodo1, Nodo2, Costo): this fact defines an edge. nodo1 represents the first node involved in the edge, nodo2 represents the second node involved in the edge, and cost represents the cost of the edge leading from nodo1 to nodo2.
- situazione(Nodo, Situazione_meteorologica): this fact defines the individual weather situation of a node.

The kb.pl code contains several different predicates. The most important ones are explained in detail here:

k_nearest_neighbors/3:

```
% Predicato per trovare i k nodi più vicini
k_nearest_neighbors(Node, K, NeighborsNodes) :-
    findall(OtherNode-Cost, (arco(Node, OtherNode, Cost), situazione(OtherNode, _), OtherNode \= Node), NeighborsWithCost),
    sort(2, @=<, NeighborsWithCost, SortedNeighbors), % Ordina in base al costo
    take(K, SortedNeighbors, Neighbors),
    extract_nodes(Neighbors, NeighborsNodes),
    write('Per il nodo '), write(Node), write(' i vicini sono: '), write(NeighborsNodes), nl.
```

This predicate calculates the K nearest neighbors to a specified node in a graph. The predicate is structured as follows:

- k_nearest_neighbors(Node, K, NeighborsNodes):- defines a predicate with three arguments: Node (the starting node), K (the number of neighbors to find), and NeighborsNodes (an output variable containing the found neighbor nodes).
- findall(OtherNode-Cost, (arc(Node, OtherNode, Cost), situation(OtherNode, _), OtherNode \= Node), NeighborsWithCost), finds all OtherNode nodes that are connected to the starting node by an edge and that satisfy the condition situation(OtherNode, _). Creates a list of OtherNode-Cost pairs, where Cost is the cost of the arc between Node and OtherNode. This list is assigned to the variable NeighborsWithCost.
- sort(2, @=<, NeighborsWithCost, SortedNeighbors) sorts the NeighborsWithCost list by the second element of each pair (i.e. the cost), in ascending order. The sorted list is assigned to the variable SortedNeighbors.
- take(K, SortedNeighbors, Neighbors) takes the first K elements from the SortedNeighbors list and assigns them to the variable Neighbors.
- extract_nodes(Neighbors, NeighborsNodes) extracts nodes from the Neighbors list (which contains pairs of nodes and costs) and assigns them to the variable NeighborsNodes.
- write('For node '), write(Node), write('Neighbors are: '), write(NeighborsNodes), nl. prints a message indicating the neighbors found for the starting node, useful for debugging.

The take/3 predicate selects the first K elements of a list:

```
% Predicato per prendere i primi K elementi di una lista
take(0, _, []) :- !.
take(_, [], []) :- !.
take(K, [H|T], [H|Rest]) :-
    K1 is K - 1,
    take(K1, T, Rest).
```

- take(0, _, []) is the first base case of the recursion. When k is equal to 0, it returns an empty list.
- take(_, [], []) is the second base case of the recursion. It returns an empty list if the current list is empty.
- Take(K, [H|T], [H|Rest]) updates the value of K and updates the parameters of the recursion predicate removing the head from the list and adding the head to the output list.

The extract_nodes/2 predicate is used to extract node-cost pairs. It works as follows:

```
% Predicato per estrarre solo i nodi da una lista di coppie nodo-peso
extract_nodes([], []).
extract_nodes([Node-_ | Rest], [Node | Nodes]) :-
    extract_nodes(Rest, Nodes).
```

- extract_nodes([], []). This is the base case of recursion. When the input list is empty, the output list will also be empty.
- extract_nodes([Node-_ | Rest], [Node | Nodes]) :- extract_nodes(Rest, Nodes). This is the recursive case. This clause is called when the input list is not empty. The first element of the input list is a Node-_ pair, where Node is the node and _ is the cost, which is ignored. The rest of the input list is Rest. The first element of the output list is Node, and the rest of the output list is Nodes. Therefore, this clause takes the first node from the input list, adds it to the output list, and then calls itself recursively on the rest of the input list.

The predict_weather/3 predicate work as follows:

```
% Predicato per predire la situazione meteorologica di un nodo basata sui suoi k vicini
predict_weather(Node, K, Weather) :-
    k_nearest_neighbors(Node, K, Neighbors),
    find_most_common_weather(Neighbors, MostCommonWeather),
    secondo_elemento(MostCommonWeather, Weather).
    %write(MostCommonWeather), nl, write(Weather), nl.
```

This rule is used to predict the weather situation for a given node based on its k neighbors. Specifically, this node calls the k_nearest_neighbors predicate to extract the k closest nodes and derive the weather situation from them. As we can see, this method calls the find_most_common_weather predicate, which counts how many times each weather type occurs in the k elements retrieved. This operation is performed using the counting performed by the conteggio_meteo/2 and aggiorna_conteggio/3 predicates.

find_most_common_weather/2:

```
% Predicato per determinare la situazione meteorologica più comune tra i vicini
find_most_common_weather(Neighbors, MostCommonWeather) :-
    conteggio_meteo(Neighbors, ListaConteggiMeteo),
    sort(ListaConteggiMeteo, SortedCounts),
    reverse(SortedCounts, DescendingCounts),
    write('Meteo dei vicini e conteggio: '), write(DescendingCounts), nl,
    prioritize_weather(DescendingCounts, MostCommonWeather).
```

This rule works as follows:

- find_most_common_weather(Neighbors, MostCommonWeather): it defines a predicate with two arguments: Neighbors (the list of neighboring nodes) and MostCommonWeather (an output variable containing the most common weather type).
- conteggio_meteo (Neighbors, ListaConteggiMeteo): calls a weather_count/2 predicate that counts the frequency of each weather type among neighboring nodes. The result is assigned to the ListaConteggiMeteo variable.
- sort(ListaConteggiMeteo, SortedCounts): sorts the weather count list in ascending order and assigns the result to the SortedCounts variable.
- reverse(SortedCounts, DescendingCounts): reverses the SortedCounts list to obtain ordered in descending order, which is assigned to the DescendingCounts variable.
- write('Meteo dei vicini e conteggio: '), write(DescendingCounts), nl, prints a message indicating the frequency of each weather type among neighboring nodes.
- prioritize_weather(DescendingCounts, MostCommonWeather) calls the prioritize_weather/2 predicate, which selects the most common weather type from the DescendingCounts list. The result is assigned to the MostCommonWeather variable.

find_most_common_weather uses two auxiliar rules in order to work: conteggio_meteo/2 e aggiorna_conteggio/3, which work as follows:

```
% Predicato per contare il numero di occorrenze di ciascun tipo di meteo
conteggio_meteo([], []).
conteggio_meteo([Nodo|Resto], ListaConteggio) :-
    conteggio_meteo(Resto, ListaConteggioResto),
    (situazione(Nodo, Meteo) ->
        (aggiorna_conteggio(Meteo, ListaConteggioResto, ListaConteggioAggiornata),
        ListaConteggio = ListaConteggioAggiornata);
        ListaConteggio = ListaConteggioResto).

% Predicato ausiliario per aggiornare il conteggio
aggiorna_conteggio(Meteo, [], [(1, Meteo)]).
aggiorna_conteggio(Meteo, [(Conteggio, Meteo)|Resto], [(NuovoConteggio, Meteo)|Resto]) :-
    NuovoConteggio is Conteggio + 1.
aggiorna_conteggio(Meteo, [(Conteggio, AltroMeteo)|Resto], [(Conteggio, AltroMeteo)|RestoAggiornato]) :-
    Meteo \= AltroMeteo,
    aggiorna_conteggio(Meteo, Resto, RestoAggiornato).
```

prioritize_weather/2:

```
% Predicato per ordinare la lista in base alla priorità della situazione
prioritize_weather(Counts, Prioritized) :-
    max_count(Counts, MaxCount),
    filter_max(Counts, MaxCount, Filtered),
    %write(Filtered), nl,
    predsort(compare_priority, Filtered, Prioritized),
    write('Priorita ordinata e filtrata: '), write(Prioritized), nl.
```

prioritize_weather sorts the weather situations from worst to best, so that, if there are equal weather situation counts, the worst one is preferred. It works as follows:

- The prioritize_weather/2 predicate takes as input a Counts list containing pairs (count, weather situation) and returns the Prioritized list.
- The Prioritized list is sorted based on the priority of the weather situations. To do this, the maximum count among the weather situations is first identified, and then the list is sorted using a predicate specific to our situation, called compare_priority, which is used in predsort, that is a sorting that takes as first argument a custom predicate.

compare_priority/3:

```
% Predicato per confrontare le priorità degli elementi
compare_priority(Risultato, (_, Situazione1), (_, Situazione2)) :-
    priority(Situazione1, Priorita1),
    priority(Situazione2, Priorita2),
    %write(Situazione1), nl, write(Situazione2), nl,
    %write(Priorita1), nl, write(Priorita2), nl,
    (   Priorita1 > Priorita2 ->
        Risultato = '<'
    ;   Priorita1 < Priorita2 ->
        Risultato = '>'
    ).
```

This rule is used to perform a sorting on the weather situation with the same occurrence count. This rule works as follows:

- The predicate priority/2 is called to obtain the priority level of each weather situation (Situation1 and Situation2). This predicate associates a number representing its priority level with each weather situation.
- The obtained priority levels are compared: if the priority level of Situation1 is higher than that of Situation2, then Result is unified with the < character, indicating that Situation1 should be placed before Situation2 in the list. This is because, according to our specifications, we prefer a sorting based on the worst weather situation.

prioritize_weather/2 uses two auxiliar rules in order to work:

```
% Predicato per trovare il massimo conteggio nella lista di tuple (conteggio, situazione)
max_count([], 0).
max_count([(Count, _)|T], MaxCount) :-
    max_count(T, MaxCountT),
    MaxCount is max(Count, MaxCountT).

% Predicato per filtrare la lista mantenendo solo gli elementi con il massimo conteggio
filter_max([], _, []).
filter_max([(Count, Situation)|T], MaxCount, [(Count, Situation)|Filtered]) :-
    Count =:= MaxCount,
    filter_max(T, MaxCount, Filtered).
filter_max([(Count, _)|T], MaxCount, Filtered) :-
    Count \= MaxCount,
    filter_max(T, MaxCount, Filtered).
```

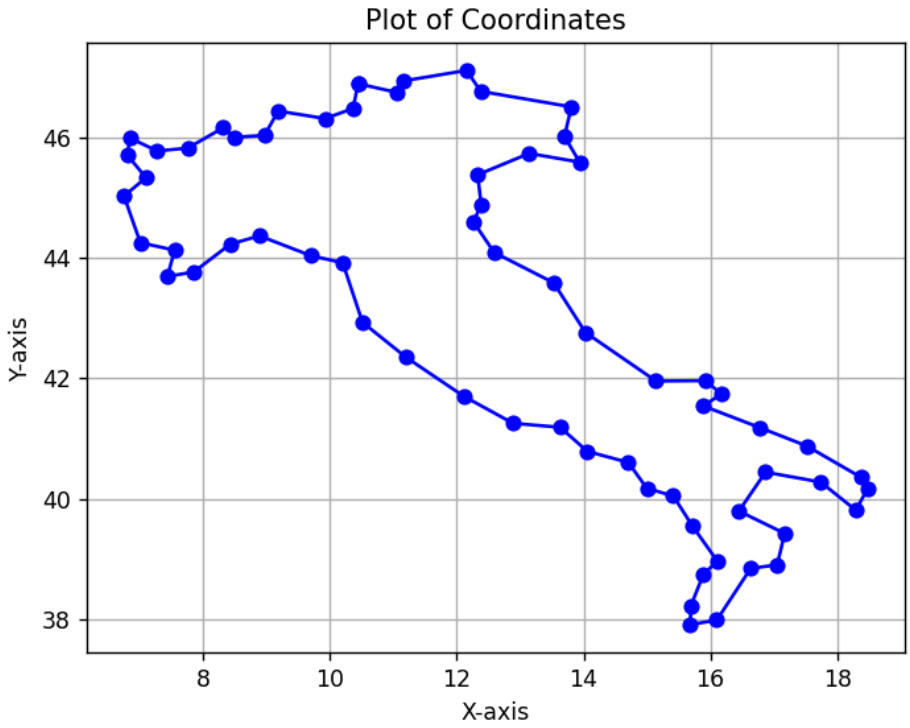
main/0:

```
main :-
    consult('code/predici.pl'),
    open('data/predizione.txt', write, Stream), % Apre il file in modalità di scrittura
    findall(Node, (arco(Node, _, _), \+ situazione(Node, _)), Nodes), % Trova tutti i nodi senza situazione
    list_to_set(Nodes, UniqueNodes), % Rimuove i duplicati
    process_nodes(UniqueNodes, Stream), % Passa lo stream del file come argomento
    close(Stream). % Chiude il file dopo aver scritto tutti i risultati
```

The main/0 rule is the main part of the program that coordinates the overall execution. Here's how it works:

- consult('predici.pl'): this statement loads the predici.pl file, which contains the facts needed for the program to run. This file contains the initial information about the graph and the weather situations of the nodes.
- open('predizione.txt', write, Stream): this statement opens the predizione.txt file in write mode and returns an output stream that will be used to write the forecast results.
- findall(Node, (arco(Node, _, _), \+ situazione(Node, _)), Nodes): this statement finds all nodes in the graph that have an outgoing edge but do not yet have a defined weather situation. The found nodes are stored in the Nodes list.
- list_to_set(Nodes, UniqueNodes): this statement removes duplicates from the Nodes list, creating a new UniqueNodes list containing only unique nodes.
- process_nodes(UniqueNodes, Stream): this statement calls the process_nodes/2 predicate, passing the list of unique nodes (UniqueNodes) and the output stream (Stream) as arguments. This will start the process of forecasting the weather for each node and writing the results to the file.
- close(Stream): this statement closes the file stream after all results have been successfully written.

Once the Prolog program had finished running, we checked to make sure the roads didn't extend beyond the boundaries. To do this, we used the geopandas library, which allowed us to extract the coordinates needed to define the Italian borders within a polygon:



For each pair of nodes, we drew the edge between the two nodes. If the edge connecting the two nodes intersects the perimeter of the polygon defining the Italy borders, then the cost of that edge is modified, because we need a slower means of transport to reach the destination.

Once this computation is completed, we calculated how the influencing factors affected the travel time on the roads. Specifically, we used the formula to calculate the time a car would take to travel the distance s at an average speed of 90 km/h: t=s/v. That is, the travel time is given by the distance between two cities divided by the speed. At this point, we consider the influencing factors to modify the travel times of the involved edges. The distance between two cities was calculated with the Haversine distance between the two points, to take into account the curvature of the Earth and make distance calculations more accurate. After calculating the distances and converting them to the appropriate time, we multiplied these values by the influencing factor that affects the travel durations.

The tsp_bb.pl file is then used to compute the best Hamiltonian path, i.e. the one that requires the minimum amount of time to be followed and to return to the starting point.

This code uses some rules, and the most important ones are explained here.

main/0:

```
main :-

    retractall(bound(_)),
    retractall(best_path(_, _)), % Pulizia dei fatti precedenti

    consult('results/input.txt'),
    setof(Node, X^Y^edge(Node, X, Y), Nodes),

    write('Nodi: '), write(Nodes), nl,

    bound(Bound),
    write('bound: '), write(Bound), nl, nl,

    Start = 1,
    exclude(==(Start), Nodes, RestNodes),
    Path = [Start],

    % Ricerca del percorso migliore
    (tsp(Start, RestNodes, Path, 0, Start) ; true ),

    best_path(BestCost, BestPath),

    write('Best Tour: '), write(BestPath), nl,
    write('Costo: '), write(BestCost), nl,
    write_results_to_file(BestPath, BestCost).
```

This rule reads the nodes of the graph from the list of the edges, it initializes the bound with a starting value written in the input.txt file and calls the tsp/5 rule, that computes the best path, i.e. the one having the best (smallest) cost.

tsp/5:

```
% Caso base: tutti i nodi visitati, ritorna al nodo iniziale e calcola il costo
tsp(Current, [], Path, Cost, Start) :-
    edge(Current, Start, CostToStart),
    append(Path, [Start], PathToStart),
    CostTot is Cost + CostToStart,
    update_best_path(PathToStart, CostTot).


% Caso ricorsivo: trova il percorso migliore tra i nodi rimanenti
tsp(Current, Nodes, Path, CostP, Start) :-
    %write('Current: '), write(Current), nl,
    %write('Nodes: '), write(Nodes), nl,
    %write('Path: '), write(Path), nl,
    %write('Cost: '), write(CostP), nl, nl, nl,
    select(Next, Nodes, RestNodes),
    edge(Current, Next, Cost),
    bound(Bound),
    CostTot is Cost + CostP,
    CostTot =< Bound,
    append(Path, [Next], NewPath),
    tsp(Next, RestNodes, NewPath, CostTot, Start),
    fail. % Forza il backtracking per esplorare tutte le possibilità

```

The parameters of tsp/5 are:

- Current: the current city
- Nodes: the list of unvisited nodes

- Path: current path
- Cost: accumulated cost so far
- Start: starting city to return to.

The base case handles the case when all nodes have been visited, so the only thing left is to return to the starting node. edge(Current, Start, CostToStart) finds the cost to return from the current city to the start city using the edge/3 predicate; append(Path, [Start], PathToStart) appends the Start city to the current path to complete the tour; costTot is updated and update_best_path/2 is called to verify if the current path is the best one.

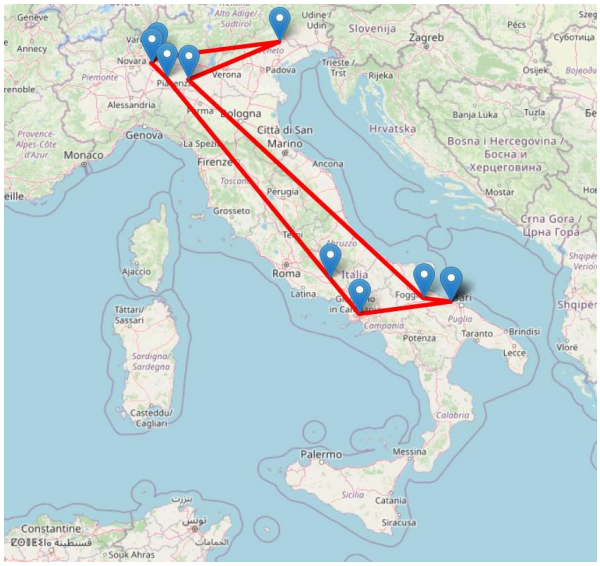
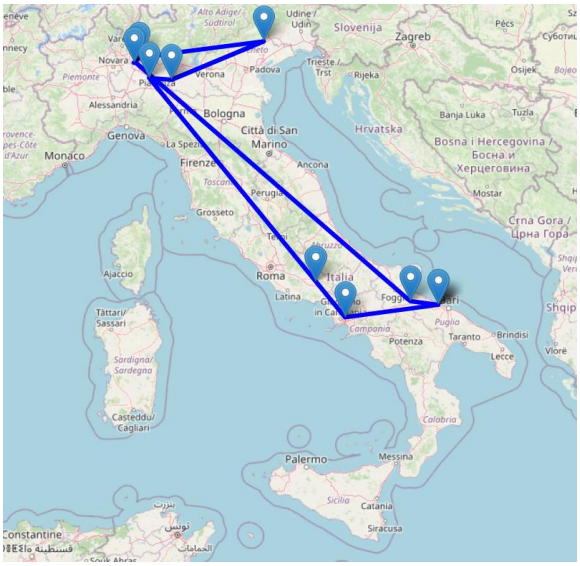
The recursive case works in the following way:

- select(Next, Nodes, RestNodes) chooses a Next node to visit from Nodes, and RestNodes will be the remaining ones after removing Next
- edge(Current, Next, Cost) looks up the cost of the edge from Current to Next
- bound(Bound) retrieves the current cost bound. If the current path exceeds this bound, the path will be pruned
- CostTot is Cost + CostP calculates the new total cost by adding the cost of going to Next
- CostTot =< Bound prunes the path if it exceeds the current best bound
- append(Path, [Next], NewPath) adds the Next node to the current path
- tsp(Next, RestNodes, NewPath, CostTot, Start) recursively continues the search from Next, with the remaining nodes
- fail forces the backtracking to explore all the permutations that are possible solutions. This forced fail is managed appropriately in the main/0 predicate.

This rule uses the auxiliar rule update_best_path/2:

```
54  % Predicato per aggiornare il miglior percorso trovato
55  update_best_path(NewPath, NewCost) :-
56      bound(CurrentBound),
57      NewCost < CurrentBound,
58
59      retractall(best_path(_, _)),
60      assertz(best_path(NewCost, NewPath)),
61
62      retractall(bound(_)),
63      assertz(bound(NewCost)).  % Aggiorno il bound
64
```

To improve the visualization of the results, we opted to use the Python library "Folium." This library allows you to represent points on the map using their geographic coordinates. It also offers the ability to connect these points with lines, which we used to trace the traveling salesman's route. Below is the result obtained at the end of our processing:

The solution on the left is the one identified without applying changes related to the conditions that alter the passability of the road sections, while the one on the right represents the one with the influencing factors included. We can see that the route doesn't differ significantly, but there are still some variations. We've attached the files city_route_map_with_weight (file with influencing factors) and city_route_map_pure (file without influencing factors). This files are created after every new execution depending on the current selected cities, and they can be viewed in any web browser.

CONCLUSIONS

To run the code, the following requirements must be met:

- folium==0.16.0
- geopandas==0.14.3
- geopy==2.4.1
- matplotlib==3.7.2
- networkx==3.5
- numpy==2.3.1
- pandas==2.3.1
- Shapely==2.1.1
- swipl

A requirements.txt file is already created in the project folder to permit a faster installation of the required python modules.

The order of execution of the program is:

- Run comuni.py to create some initial utility files and to create the graph instance
- Consult kb.pl and query the main/0 predicate to infer the weather situation of the chosen set of networks
- Run comuniGrafo.py to add the information related to the bumpy roads, the roadworks in progress and the Italian borders
- Run mappe.py and it will internally use the files tsp_bb.pl and tsp_bb_pure.pl to create the two different planning routes, the first considering all the conditions and the second one without considering them

The required plots and prolog files will be created during the execution.

In conclusion, this project presents a context-aware system for optimizing tourism in Italy by modeling a graph of cities enriched with environmental and infrastructural factors such as weather conditions and road quality. Using Prolog, a KNN algorithm was implemented to infer unknown weather conditions for cities based on available data. This was followed by solving a Traveling Salesman Problem in Prolog to find the most time-efficient route that considers real-world constraints. The integration of logic programming demonstrates a powerful approach for enhancing travel planning. The system is able to suggest efficient and adaptive tourist routes across the country.

This work contributes to the development of smart tourism tools that could support sustainable travel and reduce unnecessary transit time. Future improvements may include integrating real-time data, like live weather and traffic conditions, or expanding the model to include user preferences, for example permitting to force the presence of a particular set of cities in the planning trip.