Dipartimento di Informatica

Corso di laurea in Informatica

Caso di Studio per l'esame di Ingegneria della Conoscenza

Trip Planner

Progetto di:
Davide Cirilli (760412) d.cirilli2@studenti.uniba.it

Link Repository:
https://github.com/Davy592/Trip-Planner

Anno Accademico **2023-2024**

Indice

Introduzione

Il presente caso di studio si pone l'obiettivo di sviluppare un software dedito alla risoluzione di due problematiche legate alla pianificazione di una visita turistica in una città:

1. Dati diversi punti di interesse presenti nella città che si sta visitando, ognuno con un tempo necessario a completare la visita e un eventuale costo per la visita, individuare il sott'insieme di punti di interesse da visitare in modo da massimizzare il grado di interesse rimanendo sotto una certa soglia di tempo impiegato e budget.
2. Una volta decisi i punti di interesse da visitare, individuare l'ordine e il percorso migliore per visitarli tutti e poi tornare al punto di partenza nonché l'alloggio del turista.

Il software utilizza un approccio basato su grafi, in cui la città viene rappresentata come un grafo e sia il punto di partenza (alloggio del turista) sia i vari punti di interesse sono identificati da dei nodi nel grafo.

Il dominio in analisi è quello dello spostamento all'interno di una città sia a piedi che in macchina, nello specifico infatti il software prende in considerazione diversi parametri per la scelta del percorso migliore come per esempio se una certa strada (arco sul grafo) sia percorribile con l'automobile, il suo eventuale senso di marcia o se l'utente decida si spostarsi a piedi potendo dunque attraversare qualsiasi strada in qualsiasi verso

L'obiettivo è dunque quello di garantire all'utente di visitare quanti più punti di interesse possibile e quanto più interessanti rimanendo sotto certi vincoli e muovendosi nel modo migliore possibile.

Elenco argomenti di interesse

- **Condivisione della Conoscenza:** utilizzo di ontologie per attribuire la semantica ai dataset, rappresentazione dei dati in RDF XML ed estrazione dei dati con un parser XML dopo aver inferito la semantica di tali dati da un'ontologia top level scritta in RDF Turtle tramite query **SPARQL**.
- **Rappresentazione e ragionamento relazionale:** popolazione di una base di conoscenza in **Prolog** con i dati presenti nel dataset, **ragionamento** sulla base di conoscenza, **inferenza** di informazioni necessarie alla ricerca del percorso ottimale nel grafo.
- **Risoluzione di Problemi Mediante Ricerca:** Utilizzo della tecnica della **Programmazione Dinamica** per risolvere un problema di ottimizzazione con vincoli rigidi, utilizzo della tecnica della **memoization** per conservare eventuali

soluzioni parziali già calcolate dall'algoritmo dp, utilizzo dell'algoritmo **A*** per la ricerca del percorso migliore nel grafo con tecnica del **Multiple Path Pruning** per la riduzione dello spazio di ricerca attraverso la potatura di percorsi multipli.

Condivisione della Conoscenza

Introduzione

L'ontologia è la specifica dei significati dei simboli in un sistema informativo, quest'ultimo potrebbe essere una Knowledge Base, un sensore, database o qualche altra fonte di informazioni.

Lo scopo primario di un'ontologia è di documentare cosa un significa un simbolo. Dato un simbolo, una persona è in grado di usare l'ontologia per determinare la semantica di esso.

Le ontologie sono tipicamente costruite dalle comunità, con l'intento di renderle indipendenti da specifiche applicazioni, è proprio questo vocabolario condiviso a permettere la comunicazione e l'interoperabilità dei dati da diverse sorgenti.

OpenStreetMap

L'ontologia usata nel progetto è quella di OpenStreetMap descritta in RDF Turtle nel file `ontology.ttl` nella cartella `resources\open_street_map` del progetto.

RDF Turtle è uno dei formati di serializzazione per rappresentazione di triple RDF. Turtle fornisce una sintassi facilmente leggibile e comprensibile e inoltre offre l'enorme vantaggio di essere compatibile con **SPARQL**, il più diffuso linguaggio di interrogazione per dati rappresentati tramite RDF.

Sono stati utilizzati diversi elementi dell'ontologia OpenStreetMap quali:

Node: il concetto di node è l'elemento principale del modello dei dati di OSM, esso consiste in un singolo punto nello spazio definito da **latitudine**, **longitudine** e da un **id**. Utilizzato all'interno del progetto per mappare la posizione geografica dell'inizio, della fine e di tutte le intersezioni presenti nelle strade della città.

```
osm:Node
        a                       rdfs:Class , owl:Class ;
        rdfs:comment            """A node is one of the core elements in the OpenStreetMap data model.
                                It consists of a single point in space defined by its latitude, longitude and node id."""@en ;
        rdfs:label              "Node"@en ;
        rdfs:subClassOf         geo:Point ;
        owl:disjointWith        osm:Way , osm:Relation ;
        rdfs:isDefinedBy        osm: ;
        prov:wasInfluencedBy    <https://wiki.openstreetmap.org/wiki/Node> .
```

Esempio di un elemento node:

```
<node id="32967865" visible="true" version="4" changeset="13175152" timestamp="2012-09-19T20:03:51Z" user="hotelsierra" uid="226150" lat="41.1949595" lon="16.6375891"/>
```

Way: il concetto di way è uno degli elementi fondamentali della mappa, corrisponde ad una linea passante per una lista ordinata di node. Utilizzato all'interno del progetto per creare gli archi tra i nodi.

```
osm:Way
        a                       rdfs:Class , owl:Class ;
        rdfs:comment            "A way is an ordered list of nodes which normally also has at least one tag."@en ;
        rdfs:label              "Way"@en ;
        owl:disjointWith        osm:Node , osm:Relation ;
        rdfs:isDefinedBy        osm: ;
        prov:wasInfluencedBy    <https://wiki.openstreetmap.org/wiki/Way> .
```

Esempio di un elemento way:

```
<way id="27776786" visible="true" version="14" changeset="145728222" timestamp="2023-12-31T11:17:39Z" user="ross-map" uid="8215303">
 <nd ref="11324082776"/>
 <nd ref="1374182690"/>
 <nd ref="11324082775"/>
 <nd ref="314203326"/>
 <nd ref="1338435895"/>
 <nd ref="305594350"/>
 <nd ref="304993179"/>
 <tag k="highway" v="unclassified"/>
 <tag k="name" v="Corso Dante Alighieri"/>
 <tag k="oneway" v="yes"/>
 <tag k="surface" v="asphalt"/>
</way>
```

Come è possibile notare oltre che i node che la compongono (definiti dai tag XML **nd**) a una way possono essere associati anche dei **tag**, ognuno con attributi una chiave k e un valore v (seppur non esplicitato direttamente nell'ontologia).

Alcuni di questi tag sono stati utilizzati nel progetto durante il parsing dei dati XML per aggiungere maggiore conoscenza nella KB all'interno della quale è stato memorizzato il grafo. Tali tag sono:

highway: definisce la tipologia della way a cui è associato. Utilizzato all'interno del progetto per discriminare le strade che sono percorribili da automobili da quelle che non lo sono.

```
osm:highway
        a                       owl:ObjectProperty , rdf:Property ;
        owl:equivalentProperty  <http://www.wikidata.org/entity/Q57977870> ;
        rdfs:comment            "The kind of road, street or path."@en ;
        rdfs:label              "highway=*"@en ;
        rdfs:isDefinedBy        osm: ;
        prov:wasInfluencedBy    <https://wiki.openstreetmap.org/wiki/Key:highway> ;
        rdfs:domain             [ owl:unionOf    (osm:Node osm:Way) ] ;  _:o:7848  _:o:7866
        rdfs:range              osm:HighwayValue .
```

oneway: definisce in che modo interpretare gli elementi node che compongono la way. Utilizzato nel progetto per discriminare le strade che sono a senso unico da quelle che non lo sono oppure se i nodi per suddetta strada sono stati registrati in ordine inverso al senso di marcia.

```
osm:oneway
        a                       owl:ObjectProperty , rdf:Property ;
        owl:equivalentProperty  <http://www.wikidata.org/entity/Q786886> ;
        rdfs:comment            "The direction restrictions on highways."@en ;
        rdfs:label              "oneway=*"@en ;
        rdfs:isDefinedBy        osm: ;
        prov:wasInfluencedBy    <https://wiki.openstreetmap.org/wiki/Key:oneway> ;
        rdfs:domain             osm:Way ;
        rdfs:range              osm:OnewayValue .
```

Strumenti utilizzati

I dati con cui lavorare sono stati esportati direttamente dal sito di OSM tramite la funzione esporta. In particolare è stata esportata la struttura del centro storico e della zona villa comunale di Molfetta che è stata poi salvata all'interno di un file denominato `molf.osm` all'interno della cartella `resources\open_street_map` del progetto.

Prima di poter effettuare il parsing dei dati in RDF XML presenti nel file però, è stato necessario inferire dall'ontologia i possibili valori assumibili dai tag highway e oneway descritti in precedenza in modo da poter effettuare il parsing e la successiva rielaborazione dei dati al meglio. Per fare ciò sono state predisposti due metodi python `ask_range_of_prop(prop)` e `ask_label_for_meaning_in_range(mean, range)` all'interno della classe `sparql_client` che come è possibile intendere dal nome effettuano delle query sull'ontologia. In particolare è stata usata la classe **Graph** della libreria **rdflib** su cui è stato possibile eseguire le query grazie al metodo **query()**.

Le due query descritte sopra:

```
def ask_range_of_prop(self, prop):
    query = f"""
            PREFIX osm: <https://w3id.org/openstreetmap/terms#>
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

            SELECT ?domain ?range
            WHERE {{
              osm:{prop}
                rdfs:label "{prop}=*"@en;
                rdfs:domain ?domain;
                rdfs:range ?range .
            }}
            """
```

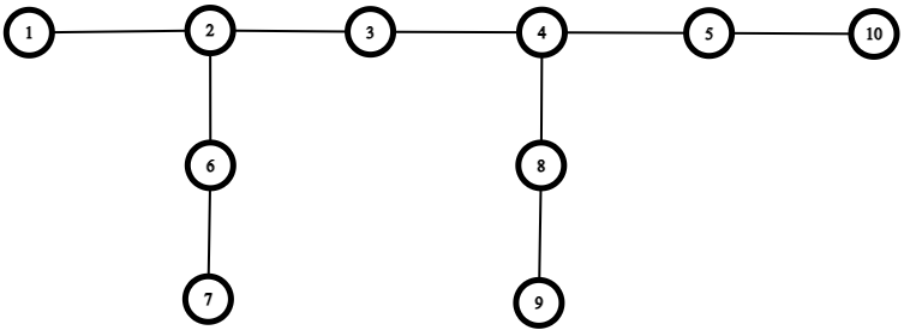
```
def ask_label_for_meaning_in_range(self, mean, range):
    query = f"""
            PREFIX osm: <https://w3id.org/openstreetmap/terms#>
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

            SELECT ?label
            WHERE {{
              osm:{mean}
                a osm:{range};
                rdfs:label ?label.
            }}
            """
```

A questo punto è stato possibile effettuare il parsing dai dati XML conoscendone il significato, in modo da rappresentare i dati in un formato più facilmente manipolabile dal linguaggio python. Per farlo è stata utilizzata la libreria **xml** default fornita dal linguaggio python.

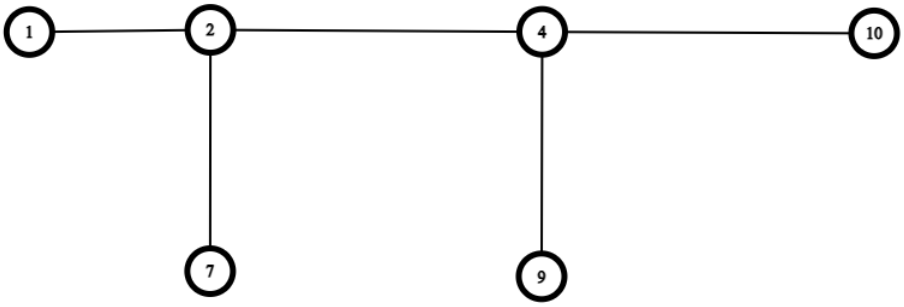
Decisioni di Progetto

Data l'enorme mole di dati estratta da OSM (in particolare la quantità di nodi) è stato deciso di effettuare un filtraggio per ridurre quanto più possibile le dimensioni del grafo (senza ovviamente intaccare la correttezza dei dati e dei percorsi generati). Questa decisione è stata presa in quanto l'implementazione della classe `Path` utilizzata per la risoluzione di problemi di ricerca tramite l'algoritmo A* fornita dalla libreria AIPython[1] fa uso della ricorsione, ma l'ambiente python non essendo ottimizzato per la ricorsione genera degli stack frame troppo grandi che potrebbero portare a consistenti rallentamenti o addirittura all'abort dell'intero programma nel caso di percorsi troppo lunghi (oltre le 1000 chiamate ricorsive data la maximum recursion depth in python). Per semplificare il grafo si è pensato dunque di mantenere al suo interno solo e soltanto i nodi che sono gli estremi di una way o che sono di intersezione tra 2 o più way ottenendo dunque le seguenti situazioni:

Esempio 1

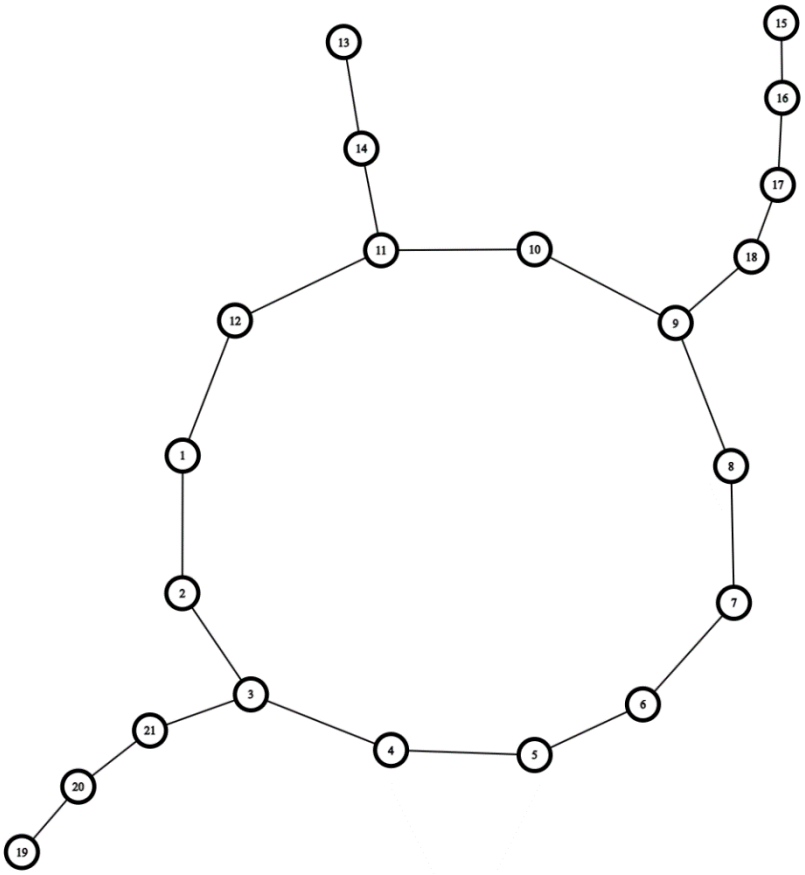


Una strada con una serie di incroci con altre strade prima della semplificazione

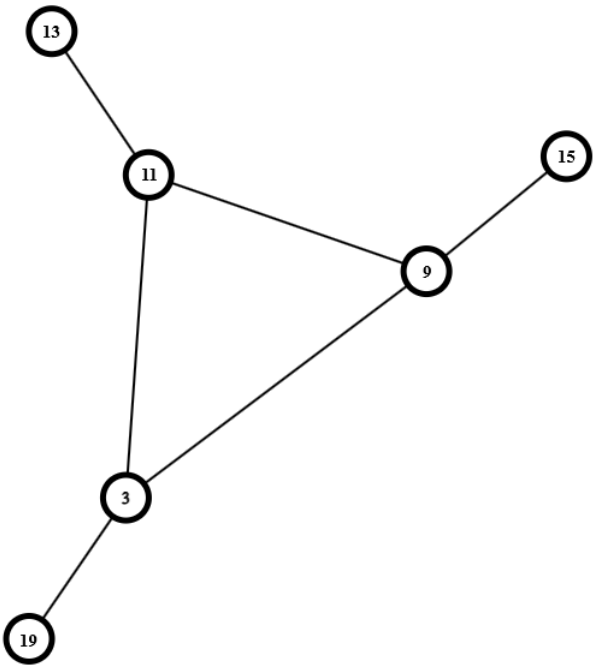


Una strada con una serie di incroci con altre strade dopo la semplificazione

Esempio 2



Una rotonda con le sue uscite prima della semplificazione

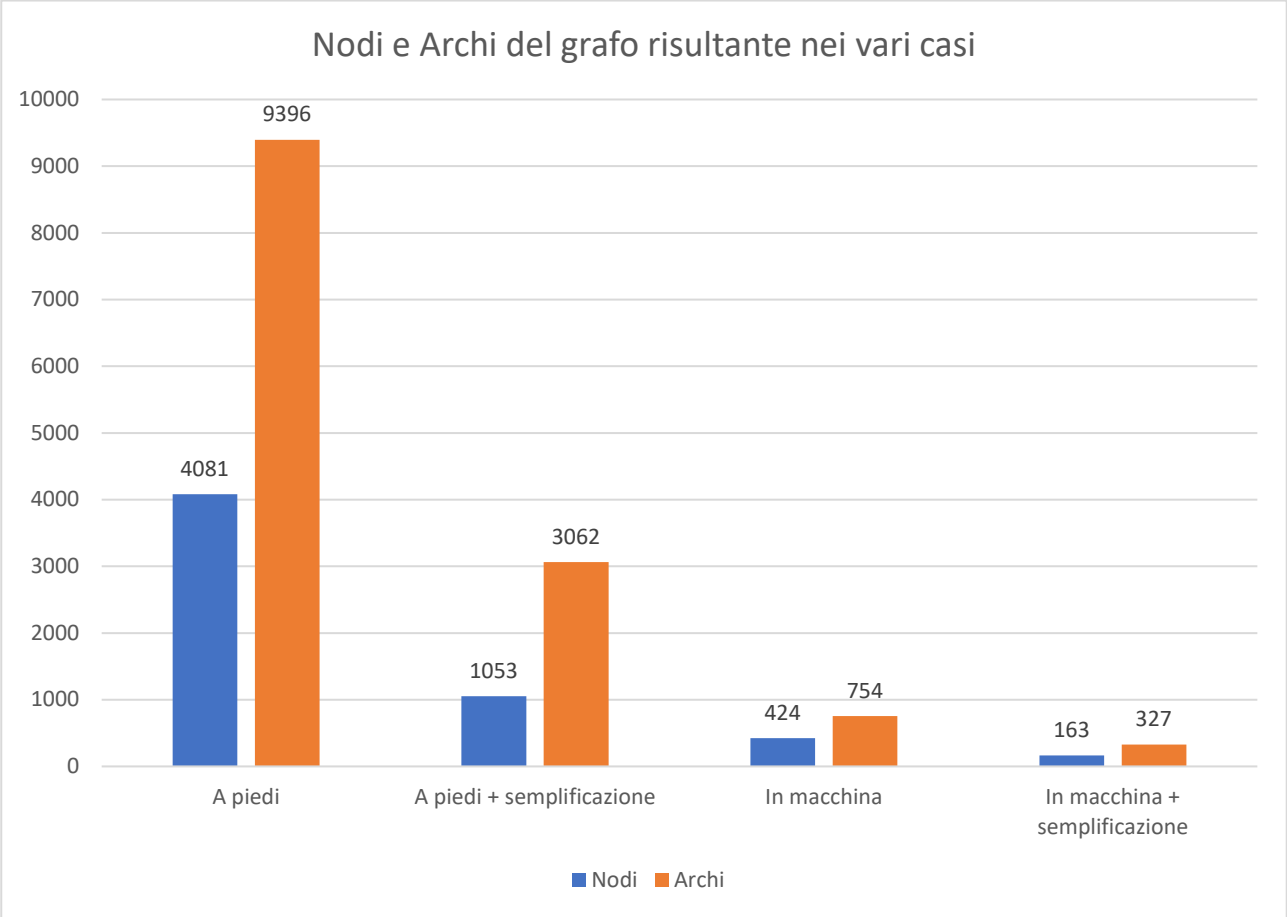


Una rotonda con le sue uscite prima dopo la semplificazione

Come è possibile notare in entrambi i casi il numero di nodi è diminuito sensibilmente dopo la semplificazione seppur la raggiungibilità e la topologia dei nodi rimasti sia rimasta invariata.

A questa semplificazione, come detto in precedenza, è stato aggiunto un eventuale filtraggio sulla base delle condizioni del turista: se il turista è a piedi potrà percorrere tutte le strade in tutte le direzioni, se il turista è in macchina invece potrà percorrere solo le strade che possono essere percorse in macchina e solo nel verso consentito.

Di seguito un grafico che indica l'effetto della semplificazione e del filtraggio sul grafo preso in esempio:



Rappresentazione e Ragionamento Relazionale

Introduzione

Una Knowledge Base è un sistema o un insieme di conoscenze organizzate in modo strutturato e accessibile, progettato per immagazzinare informazioni e facilitare l'elaborazione delle stesse da parte di un programma o da un agente intelligente.

La Knowledge Base rappresenta la memoria a lungo termine di un agente, dove viene conservata la conoscenza necessaria per agire in futuro. Essa tipicamente è costruita offline da una combinazione di esperti di conoscenza e di dati.

Classi

Per modellare il dominio sono state identificate le seguenti classi di individui:

Node	
Proprietà	**Descrizione**
lat	Latitudine del nodo
lon	Longitudine del nodo
Significato	
La classe Node modella un singolo punto nello spazio avente le coordinate identificate dalla latitudine e dalla longitudine, esso svolge il compito di congiunzione tra le Way (che sono gli archi). I nodi permettono inoltre di identificare la collocazione del turista e dei punti di interesse da visitare.	

Un esempio di definizione di node:

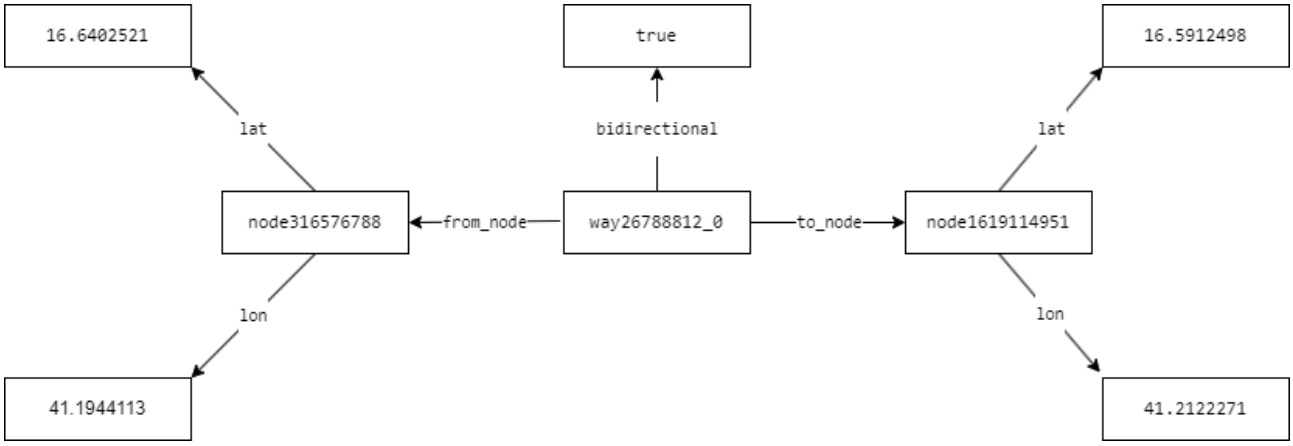
```
prop(node922748928, lat, 16.600514).
prop(node922748928, lon, 41.2057967).
prop(node922748928, type, node).
```

Way	
Proprietà	**Descrizione**
from_node	Indica il nodo di partenza dell'arco
to_node	Indica il nodo di arrivo dell'arco
bidirectional	Indica se l'arco è bidirezionale
Significato	
La classe Way modella un arco dal nodo from_node al nodo to_node. Tale arco è caratterizzato dalla proprietà bidirectional che indica se l'arco può essere percorso in entrambe le direzioni oppure se deve essere percorso necessariamente per andare dal nodo from_node al nodo to_node	

Un esempio di definizione di way:

```
prop(way26788812_0, from_node, node316576788).
prop(way26788812_0, to_node, node1619114951).
prop(way26788812_0, bidirectional, true).
prop(way26788812_0, type, way).
```

La rete semantica risultate è la seguente:



Fatti

I fatti sono affermazioni vere sul dominio. Essi sono costituiti da predicati, che sono relazioni tra termini. Un fatto è una clausola di tipo *Predicato(arg1, arg2, … , argN).* dove il predicato rappresenta la relazione e gli argomenti specificano gli elementi coinvolti nella relazione.

Come è possibile notare anche dagli esempi precedenti per strutturare la conoscenza in maniera più omogenea ho utilizzato la rappresentazione **individuo-proprietà-valore** avvicinandomi così alla struttura di triple del framework RDF.

Si ottengono una serie di clausole del tipo

prop(Ind, Prop, Val)

che indicano che l'individuo *Ind* ha il valore *Val* per la proprietà *Prop.*
Questa rappresentazione è chiamate anche **triple representation** in quanto tutte le relazioni sono rappresentate come triple.
Il primo elemento di questa tripla è chiamato **subject**, il secondo **verb** e il terzo **object**, usando l'analogia nella quale una tripla è una frase semplice di tre parole.

All'interno è presente una sola relazione con i 3 parametri chiamata '*prop*' che permette di definire i fatti. Nel caso il turista decida di andare a piedi (definito dal flag on_foot_main) verrà anche asserita la verità dell'atomo on_foot all'interno della KB.

Tutti questi fatti vengono generati automaticamente dalla classe `FactsWriter` dopo il parsing e scritti all'interno del file `facts.pl` memorizzato nella cartella `resources\prolog` del progetto.

Forma finale della KB:

```
on_foot.
prop(node922748928, lat, 16.600514).
prop(node922748928, lon, 41.2057967).
prop(node922748928, type, node).
prop(node2484019202, lat, 16.6019449).
prop(node2484019202, lon, 41.2032214).
prop(node2484019202, type, node).
prop(node2484019204, lat, 16.6019709).
prop(node2484019204, lon, 41.203268).
prop(node2484019204, type, node).
prop(node2742083589, lat, 16.5973088).
prop(node2742083589, lon, 41.2066822).
prop(node2742083589, type, node).
prop(node922748934, lat, 16.5995411).
prop(node922748934, lon, 41.2063517).
prop(node922748934, type, node).
prop(node4124397575, lat, 16.599779).
prop(node4124397575, lon, 41.2051672).
prop(node4124397575, type, node).
```

```
prop(way26788812_0, from_node, node316576788).
prop(way26788812_0, to_node, node1619114951).
prop(way26788812_0, bidirectional, true).
prop(way26788812_0, type, way).
prop(way26788812_1, from_node, node1619114951).
prop(way26788812_1, to_node, node316576788).
prop(way26788812_1, bidirectional, true).
prop(way26788812_1, type, way).
prop(way26788812_2, from_node, node1619114951).
prop(way26788812_2, to_node, node1619114947).
prop(way26788812_2, bidirectional, true).
prop(way26788812_2, type, way).
prop(way26788812_3, from_node, node1619114947).
prop(way26788812_3, to_node, node1619114951).
prop(way26788812_3, bidirectional, true).
prop(way26788812_3, type, way).
prop(way26788812_4, from_node, node1619114947).
prop(way26788812_4, to_node, node565447288).
prop(way26788812_4, bidirectional, true).
prop(way26788812_4, type, way).
prop(way26788812_5, from_node, node565447288).
prop(way26788812_5, to_node, node1619114947).
prop(way26788812_5, bidirectional, true).
prop(way26788812_5, type, way).
```

Regole

Le regole permettono di definire nuove relazioni o di inferire informazioni a partire dai fatti esistenti. La struttura di una regola è costituita da una testa e un corpo separati dall'operatore :-. La testa di una regola specifica il predicato che si desidera dedurre o ottenere come risultato. La testa è costituita da un unico predicato o da una congiunzione di predicati. Il corpo di una regola contiene una serie di predicati che devono essere soddisfatti affinché la regola possa essere applicata.

Per questo caso di studio le regole sono state scritte manualmente nel file `rules.pl` all'interno della cartella `resources\prolog` del progetto in base alle necessità del dominio. Di seguito le regole più importanti/complesse:

is_node

```
% tells if a variable it's a node
is_node(X):-
    prop(X, type, node).
```

Dato un individuo restituisce true se tale individuo appartiene alla classe node

get_node_coord

```
% tells the coord of a node
get_node_coord(X, Lat, Lon):-
    is_node(X),
    prop(X, lat, Lat),
    prop(X, lon, Lon).
```

Dato un nodo ne restituisce latitudine e longitudine.

is_way

```
% tells if a variable it's a way
is_way(X):-
    prop(X, type, way).
```

Dato un individuo restituisce true se tale individuo appartiene alla classe way

get_ways

```
% gets all the ways from a given node From_node to a given node To_node
get_ways(Way, From_node, To_node):-
    is_way(Way),
    prop(Way, from_node, From_node),
    prop(Way, to_node, To_node).
```

Dati 2 nodi restituisce l'arco che li collega (se esiste).

get_neighbours

```
% gets all the info of all the neighbours of a given From_node (without limitations)
get_neighbours(From_node, To_node, To_node_lat, To_node_lon):-
    get_ways(_, From_node, To_node),
    get_node_coord(To_node, To_node_lat, To_node_lon).
```

Dato un nodo ritorna tutti i suoi nodi vicini con tutte le loro informazioni quali latitudine e longitudine. La ricerca viene effettuata senza considerare eventuali limitazioni data dal fatto che il turista sia a piedi o in macchina oppure se l'arco che collega i 2 nodi sia bidirezionale o meno. Corrisponde al cercare i vicini considerando tutti gli archi monodirezionali.

get_available_neighbours

```
% gets all the info of all the neighbours of a given From_node (with limitations)
% true if exist a way from From_node to To_node
% or if exist a way from To_node to From_node and the way is bidirectional
% or if exist a way between the nodes (besides the direction) and im on foot
get_available_neighbours(From_node, To_node, To_node_lat, To_node_lon):-
    get_ways(_, From_node, To_node),
    get_node_coord(To_node, To_node_lat, To_node_lon);
    get_ways(Way, To_node, From_node),
    get_node_coord(To_node, To_node_lat, To_node_lon),
    prop(Way, bidirectional, true);
    on_foot,
    (get_ways(_, From_node, To_node),
    get_node_coord(To_node, To_node_lat, To_node_lon);
    get_ways(_, To_node, From_node),
    get_node_coord(To_node, To_node_lat, To_node_lon)).
```

Dato un nodo ritorna tutti i suoi nodi vicini con tutte le loro informazioni quali latitudine e longitudine. La ricerca viene effettuata considerando sia il fatto che il turista sia a piedi o in macchina sia il fatto che a collegare i due nodi ci potrebbe essere un arco posto nel senso inverso ma bidirezionale

get_all_node

```
% get all the ids presents for the node
get_all_node(Node, Lat, Lon):-
    is_node(Node),
    prop(Node, lat, Lat),
    prop(Node, lon, Lon).
```

Restituisce tutti i nodi della KB compresi delle loro informazioni

get_all_ways

```
% get all the ways in KB
get_all_ways(Way, From_node, To_node, bidi):-
    is_way(Way),
    prop(Way, from_node, From_node),
    prop(Way, to_node, To_node),
    prop(Way, bidirectional, bidi).
```

Restituisce tutti gli archi della KB compresi delle loro informazioni

get_distance

```
% gets the distance between two nodes
get_distance(From_node, To_node, Distance):-
    get_node_coord(From_node, From_node_lat, From_node_lon),
    get_node_coord(To_node, To_node_lat, To_node_lon),
    Distance is sqrt((From_node_lat - To_node_lat) ** 2 + (From_node_lon - To_node_lon) ** 2).
```

Dati due nodi ritorna la loro distanza calcolata col teorema di pitagora sulla base di latitudine e longitudine

haversine_distance

```
% haversine distance
% given two nodes returns the distance between them in km
haversine_distance(From_node, To_node):-
    get_node_coord(From_node, Lat1, Lon1),
    get_node_coord(To_node, Lat2, Lon2),
    % Convert latitude and longitude from decimal degree to radians
    Radians1Lat is Lat1 * pi / 180,
    Radians1Lon is Lon1 * pi / 180,
    Radians2Lat is Lat2 * pi / 180,
    Radians2Lon is Lon2 * pi / 180,
    % Calculate the difference between the two longitudes and latitudes
    Dlon is Radians2Lon - Radians1Lon,
    Dlat is Radians2Lat - Radians1Lat,
    % Calculate the half-verse of angular distance
    A is sin(Dlat / 2) ** 2 + cos(Radians1Lat) * cos(Radians2Lat) * sin(Dlon / 2) ** 2,
    % Calculate the distance
    C is 2 * asin(sqrt(A)),
    % Return the distance in km
    Distance is 6371 * C.
```

Dati due nodi ritorna la loro distanza in km calcolata attraverso la formula dell' haversine o formula dell'emisenoverso.

Strumenti utilizzati

Per la definizione di fatti e regole è stato utilizzato il linguaggio per la programmazione logica **Prolog** attraverso il software **SWI-Prolog** che fornisce un bridge a cui è possibile agganciarsi con la libreria **pyswip**.

Decisioni di Progetto

Un limite della libreria pyswip è proprio quello della definizione a runtime di fatti. Sarebbe possibile infatti definire i vari fatti della KB a runtime attraverso la funzione *assertz* di pyswip ma per questo caso di studio si è deciso di scrivere suddetti fatti nel file `fact.pl` e solo successivamente di caricarli dal file attraverso la funzione *consult*. Tale decisione è stata presa poiché la libreria memorizza temporaneamente i fatti passati alla funzione *assertz* in una cache che si è rivelata ben presto troppo piccola per contenere la mole di fatti necessari allo svolgimento di questo caso di studio.

Risoluzione di Problemi Mediante Ricerca

Introduzione

Risolvere i problemi mediante ricerca è una tecnica fondamentale nella ingegneria della conoscenza.

Questa tecnica prevede la formulazione di uno spazio di ricerca, dove vengono esplorate diverse soluzioni per trovare l'ottimale.

Per applicarla è necessario definire uno stato iniziale, una serie di azioni che cambiano il nostro stato e lo stato finale.

Durante il processo di ricerca si possono applicare diversi algoritmi che si distinguono in ricerca informata e ricerca non informata. La ricerca non informata non ha conoscenza sulla posizione degli obbiettivi e non possono essere guidati. La ricerca informata ha conoscenza sulla posizione degli obbiettivi e possono essere guidati.

Programmazione Dinamica

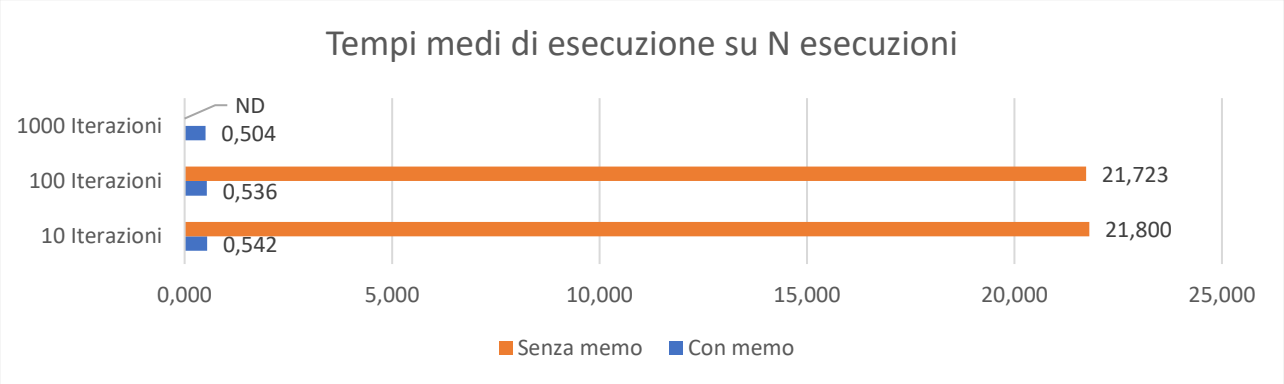
Una volta memorizzate tutte le informazioni necessarie ad inferire la struttura del grafo nella KB si passa alla fase di risoluzione problemi. Come già anticipato nell'introduzione il primo problema che è necessario risolvere è il seguente: dati diversi punti di interesse, ognuno caratterizzato da un certo grado di interesse, il tempo di visita e il costo di visita, individuare un subset di punti di interesse (anche tutti) che massimizzi l'interesse complessivo mantenendo però il tempo totale di visita e il costo complessivo delle visite sotto delle determinate soglie di tempo e budget.

Per questo caso di studio sono stati generati randomicamente i valori di interesse, tempo e costo (in quanto soggettivi o non asseribili) per un totale di 100 punti di interesse (associati a 100 nodi diversi) e su questi si è proceduto a trovare il subset ottimale. Si tratta dunque di un problema di ottimizzazione, in quanto bisogna massimizzare un certo parametro, con la presenza di vincoli rigidi, in quanto non bisogna superare le soglie di tempo e costo.

Per risolverlo si è ricorso alla tecnica della programmazione dinamica, una tecnica che esplora lo spazio di ricerca **memorizzando le soluzioni già calcolate** in modo da non doverle ricalcolare. Quello del memorizzare le soluzioni parziali è il vero punto forte della DP che le permette di abbassare di molto la complessità computazionale e dunque ridurre i tempi di esecuzione per problemi che altrimenti potrebbero avere complessità esponenziale.

Per questo caso di studio si è deciso di memorizzare le soluzioni parziali con la tecnica della **memoization**, implementata all'atto pratico come un dizionario con chiave una tupla [i, t, b], rappresentanti i punti di interesse attualmente analizzato, il tempo rimanente e il budget rimanente, e valore memo[i, t, b] corrispondente alle soluzioni parziali con questi valori. La complessità è stata ridotta da esponenziale $O(2^n)$ dove n è il numero totale di punti di interesse (nel nostro caso 100) a O(ntb) dove t e b sono rispettivamente il tempo e il budget totale.

Di seguito riportati i tempi di esecuzione per l'algoritmo DP con e senza l'utilizzo della memoization:



Con media (μ), dev. standard (o scarto quadratico medio, σ) e varianza (σ^2):

Modalità	**μ**	**σ**	**σ^2**
Con memo	0,527	0.017	0,00029
Senza memo	21,762	0,039	0,001521

Non è presente il dato per le 1000 esecuzioni senza memo perché tale dato richiederebbe troppo tempo per essere calcolato

Ovviamente per eseguire la valutazione delle prestazioni dell'algoritmo tutte le esecuzioni sono state eseguite sugli stessi valori di interesse, tempo e costo per i 100 punti di interesse.

Floyd-Warshall

Una volta individuato il miglior subset di punti di interesse che vanno visitati il passo successivo è quello di capire l'ordine in cui questi devono essere visitati.

Per computare tale ordine si sono considerate tutte le permutazioni dei punti di interesse, e si è scelta quella in cui il costo del percorso potenziale passante per i nodi dei punti di interesse nell'ordine della permutazione in analisi (e per poi tornare al punto di partenza ossia l'alloggio) sia minimo.

Per poter ottenere la lunghezza del percorso ottimale da un nodo *A* ad un nodo *B* è stato utilizzato l'algoritmo di Floyd-Warshall che permette di computare una matrice contenente tutte le distanze da ogni nodo ad ogni altro nodo.

A questo punto per individuare l'ordine corretto di visita dei punti di interesse, ossia la permutazione corretta, è stata individuata la permutazione perm che minimizza la seguente formula:

$$dist(perm_{|perm|}, perm_0) + \sum_{i=1}^{|perm|} dist(perm_{i-1}, perm_i)$$

Con $perm_0$ = $nodo_{start}$ e perm vettore dei punti di interesse.
Il primo membro della formula corrisponde alla distanza da percorrere per tornare all'alloggio partendo dall'ultimo punto di interesse della permutazione.

A* per il percorso minimo

Una volta individuato l'ordine corretto di visita dei punti di interesse il problema si riduce al trovare il percorso migliore tra ogni coppia di nodi consecutivi nella permutazione selezionata e infine dall'ultimo nodo della permutazione selezionata al primo (percorso per tornare all'alloggio).

Per fare questo si è deciso di applicare |perm| volte l'algoritmo A*.
Nell'algoritmo A* per ogni percorso *p* nella frontiera possiamo definire

$$f(p) = cost(p) + h(p)$$

dove *h(p)* è una funzione euristica che stima la lunghezza del percorso dall'ultimo nodo di *p* al nodo goal.
Utilizzando proprio la matrice *dist* computata dall'algoritmo di Floyd-Warshall utilizzato in precedenza come euristica otterremo una euristica perfetta, che sarà ammissibile, consistente e che per ogni *p* restituirà esattamente la distanza che bisogna ancora percorrere per andare dall'ultimo nodo del percorso *p* al nodo goal.

In particolare risulta che:

$$h(p) = dist(p, goal)$$

considerando la distanza partendo dall'ultimo nodo del percorso *p*

Strumenti utilizzati

Per la rappresentazione di un problema di ricerca e della sua soluzione sono state utilizzate le classi provenienti da AIPython[1]. Tali classi sono situate nella cartella

`src/external_libs` del progetto. I file sono stati modificati dagli originali per modificare la stampa a video.

Decisioni di Progetto

Dato che si ha a disposizione un'euristica ammissibile e soprattutto consistente si è deciso di aggiungere all'algoritmo A* la tecnica dell'**MPP** (**M**ultiple **P**ath **P**runing). Questa tecnica ha permesso di potare dallo spazio di ricerca eventuali percorsi multipli ritrovati durante la ricerca del percorso migliore in modo da ottimizzare ancor di più l'algoritmo A*.

Conclusioni

I due obiettivi preposti di individuare il subset ottimale di punti di interesse da visitare e successivamente di trovare il miglior percorso per visitarli sono stati raggiunti.

La maggiore difficoltà è stata lavorare con reali che nel caso delle distanze sono risultati molto piccoli (sia nel caso della distanza normale che per quella di haversine) e che a causa di errori di approssimazione hanno portato in alcuni casi al considerare nodi distinti come se posti nello stesso punto geografico. È stato inoltre difficile avere a che fare con una tale mole di dati estratti da OpenStreetMap prima di effettuare la semplificazione che si è rivelata necessaria e provvidenziale

Possibili sviluppi futuri

- Si potrebbe considerare anche l'elemento relation fornito da OpenStreetMap in modo da permettere anche spostamenti con i mezzi pubblici.
- Si potrebbe considerare nel calcolo del percorso migliore, qual ora si sia in macchina, anche la presenza di eventuali semafori aggiungendo un costo supplementare agli archi che li contengono.
- Si potrebbe tramite altre integrazioni o ontologie riuscire ad inferire posizione, tempo di visita e costo di visita dei vari punti di interesse in modo da non doverli generare casualmente. In una versione completa dell'applicazione l'interesse del turista verso un certo punto di interesse potrebbe essere inserito in input dal turista stesso oppure inferito da un sistema di Information Filtering.

Riferimenti Bibliografici

[1] https://artint.info/AIPython/