UNIVERSITÀ DEGLI STUDI DI BARI ALDO MORO
DIPARTIMENTO DI INFORMATICA

Documentazione Caso di Studio
Ingegneria della Conoscenza
A.A. 2022/23
Semafori Intelligenti

Gruppo di lavoro

- Matteo Balice, 743479, m.balice15@studenti.uniba.it
- Antonio Giuseppe Doronzo, 736571, a.doronzo38@studenti.uniba.it
- Domenico Lacavalla, 738087, d.lacavalla2@studenti.uniba.it
- Andrei Alexandru Stefan, 735406, a.stefan1@studenti.uniba.it

Repository GitHub

- https://github.com/bralani/icon22-23

Sommario

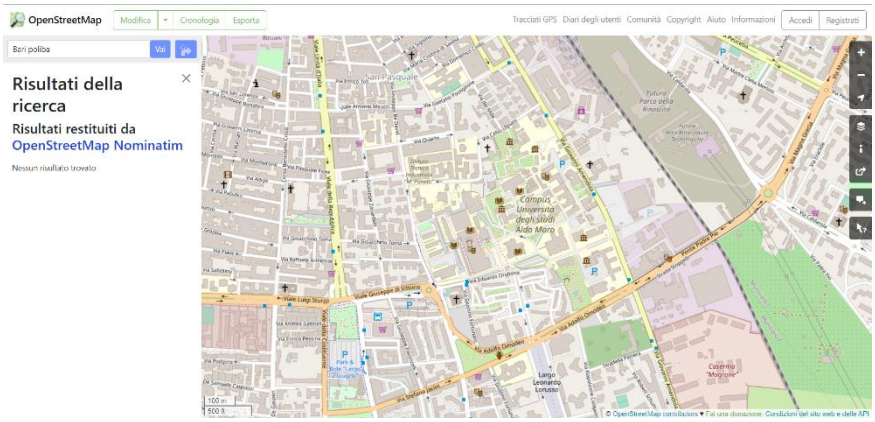
Introduzione

Il software per la gestione del traffico a incroci semaforici mira a migliorare l'efficienza del traffico attraverso l'utilizzo di tecnologie avanzate di intelligenza artificiale e analisi dei dati.

L'obiettivo principale del software è quello di aumentare la probabilità che i semafori diventino verdi per i veicoli, in modo da ridurre i tempi di attesa, migliorare la fluidità e ridurre l'inquinamento. Ciò può essere ottenuto utilizzando algoritmi di intelligenza artificiale per analizzare i dati del traffico in tempo reale e adattare il ciclo del semaforo in base alle esigenze del flusso di veicoli sfruttando una tecnica dell'onda verde.

Ontologie

OpenStreetMap (OSM) è un progetto open source che consente agli utenti di creare e condividere dati geografici, tra cui informazioni stradali, edifici e punti di interesse. I dati di OSM sono disponibili in diversi formati, tra cui XML, che è un formato di testo utilizzato per la trasmissione e la rappresentazione dei dati su Internet.



I dati di OpenStreetMap in formato XML sono stati usati per generare clausole in Prolog.

Prolog[10] è un linguaggio di programmazione logico che può essere utilizzato per rappresentare conoscenza e interrogare una base di conoscenza. I dati di OpenStreetMap in formato XML sono stati convertiti in un formato adatto per la rappresentazione in Prolog, come una serie di clausole e classi che descrivono le relazioni tra gli elementi della mappa. La struttura di un file XML di OpenStreetMap è composta da un insieme di elementi che descrivono gli oggetti presenti nella mappa, come strade, edifici, punti d'interesse e così via.

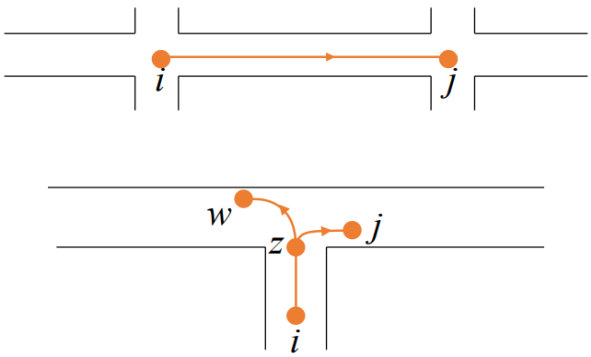
Il file XML inizia con un elemento radice <osm> che contiene tutti gli altri elementi del file.

Gli elementi principali sono:

- **<node>:** rappresenta un punto geografico sulla mappa, con un set di attributi come le coordinate latitudine e longitudine.
- **<way>:** rappresenta una strada o una via, come una via principale o un sentiero, composto da una serie di nodi.
- **<relation>:** rappresenta una relazione tra gli elementi della mappa, come una relazione tra una strada e un edificio.
- **<tag>:** contiene i metadati per un elemento, come il nome della strada, il tipo di edificio o il tipo di sagnale stradale.

```
<osm version="0.6" generator="JOSM">
  <node id="26111427" lat="41.1175778" lon="16.8736252" version="5"
timestamp="2008-09-21T16:06:51Z" changeset="145675" user="SomeoneElse" uid="87">
    <tag k="amenity" v="bar"/>
    <tag k="name" v="Bar Centrale"/>
    <tag k="address" v="via Sparano"/>
  </node>
  <way id="29881475" version="3" timestamp="2008-09-21T16:07:01Z"
changeset="145675" user="SomeoneElse" uid="87">
    <nd ref="26111427"/>
    <nd ref="29391217"/>
    <tag k="highway" v="residential"/>
    <tag k="name" v="via Sparano"/>
  </way>
</osm>
```

Struttura XML



Rappresentazione grafica dei dati in OSM

I dati in OSM sono organizzati come un grafo, dove ogni elemento è identificato da un identificativo univoco (id) e gli elementi sono collegati tra di loro mediante relazioni. Questo consente di navigare facilmente i dati e di estrarre informazioni rilevanti attraverso query sulla base di conoscenza rappresentata dai dati. La struttura a grafo rende possibile una facile navigazione e relazioni tra gli oggetti, permettendo di accedere facilmente alle informazioni richieste.

Knowledge Base (KB)

Una Knowledge Base (KB)[12] è un insieme di conoscenze organizzate in modo da poter essere utilizzate da un programma o sistema per rispondere a domande o prendere decisioni. La KB che abbiamo implementato descrive un modello per un sistema di navigazione stradale utilizzando la programmazione logica in Prolog. Il modello utilizza classi per rappresentare le strade e gli incroci, con proprietà specifiche per ogni classe. Le relazioni tra le classi sono definite utilizzando la funzione "prop".

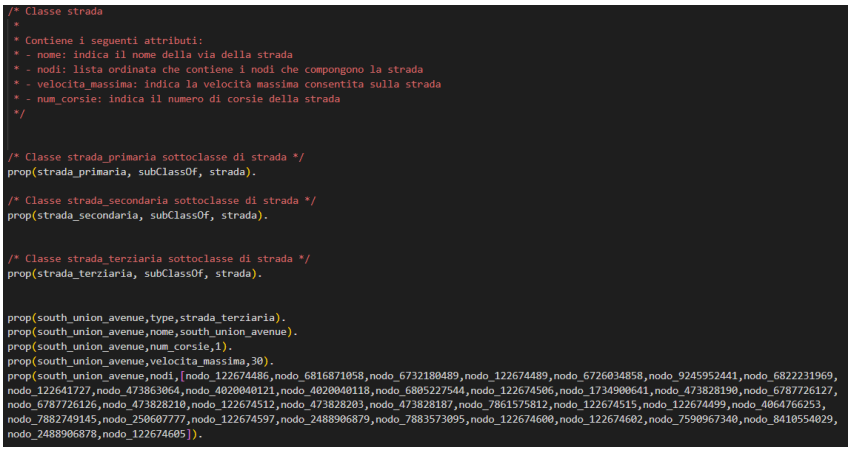
Fatti e regole della KB

Classe strada: La classe strada ha i seguenti attributi:

- **nome**: indica il nome della via della strada
- **nodi**: lista ordinata che contiene i nodi che compongono la strada (elementi di classe nodo)
- **velocita_massima**: indica la velocità massima consentita sulla strada
- **num_corsie**: indica il numero di corsie della strada

Sottoclassi di strada:

- Classe **strada_primaria** sottoclasse di strada
- Classe **strada_secondaria** sottoclasse di strada
- Classe **strada_terziaria** sottoclasse di strada



```
/* Classe strada
 *
 * Contiene i seguenti attributi:
 * - nome: indica il nome della via della strada
 * - nodi: lista ordinata che contiene i nodi che compongono la strada
 * - velocita_massima: indica la velocità massima consentita sulla strada
 * - num_corsie: indica il numero di corsie della strada
 */


/* Classe strada_primaria sottoclasse di strada */
prop(strada_primaria, subClassOf, strada).

/* Classe strada_secondaria sottoclasse di strada */
prop(strada_secondaria, subClassOf, strada).


/* Classe strada_terziaria sottoclasse di strada */
prop(strada_terziaria, subClassOf, strada).


prop(south_union_avenue,type,strada_terziaria).
prop(south_union_avenue,nome,south_union_avenue).
prop(south_union_avenue,num_corsie,1).
prop(south_union_avenue,velocita_massima,30).
prop(south_union_avenue,nodi,[nodo_122674486,nodo_6816871058,nodo_6732180489,nodo_122674489,nodo_6726034858,nodo_9245952441,nodo_6822231969,
nodo_122641727,nodo_473863064,nodo_4020040121,nodo_4020040118,nodo_6805227544,nodo_122674506,nodo_1734900641,nodo_473828190,nodo_6787726127,
nodo_6787726126,nodo_473828210,nodo_122674512,nodo_473828203,nodo_473828187,nodo_7861575812,nodo_122674515,nodo_122674499,nodo_4064766253,
nodo_7882749145,nodo_250607777,nodo_122674597,nodo_2488906879,nodo_7883573095,nodo_122674600,nodo_122674602,nodo_7590967340,nodo_8410554029,
nodo_2488906878,nodo_122674605]).
```

La classe strada descrive le proprietà comuni delle strade, come il nome, la velocità massima, il numero di corsie e i nodi che compongono la strada. Per descrivere una singola strada, viene utilizzata la funzione "prop" per specificare che un oggetto di tipo "strada" ha determinati valori per i suoi attributi. Le sottoclassi di strada (strada primaria, strada secondaria e strada terziaria) servono per descrivere le diverse tipologie di strade presenti nella KB. In generale, le sottoclassi vengono utilizzate per descrivere oggetti che condividono alcune proprietà comuni con una classe generale, ma che hanno anche proprietà uniche. In questo caso, le diverse sottoclassi di strada vengono diversificate per poter determinare l'indice di traffico medio in quella strada (vedere l'algoritmo di apprendimento supervisionato).

Classe nodo: La classe nodo ha i seguenti attributi:

- **id**: Identificativo del nodo
- **latitudine**: indica la latitudine del nodo
- **longitudine**: indica la longitudine del nodo

La classe nodo serve per descrivere gli elementi di connessione tra le strade nella KB. In particolare, in questa KB i nodi a cui si fa riferimento sono quelli presenti in **OpenStreetMap** (OSM). Come affermato precedentemente i dati sulle strade sono rappresentati come sequenza di nodi interconnessi. La classe nodo descrive le proprietà comuni dei nodi, come l'identificativo univoco, la posizione geografica (latitudine e longitudine) e altri eventuali attributi. In questo modo, gli oggetti della classe nodo possono essere utilizzati per descrivere le strade (come sequenza di nodi) e gli incroci tra le strade nella KB, permettendo di descrivere la topologia della rete stradale.

```
/* Classe nodo
 *
 * Contiene i seguenti attributi:
 * - id: Identificativo del nodo
 * - latitudine: indica la latitudine del nodo
 * - longitudine: indica la longitudine del nodo
 */
prop(nodo_18882133,type,nodo).
prop(nodo_18882133,id,nodo_18882133).
prop(nodo_18882133,latitudine,34.0571853).
prop(nodo_18882133,longitudine,-118.2759955).
```

Sottoclasse incrocio: L'incrocio è una sottoclasse del nodo e ha i seguenti attributi:

- **strade:** lista che indica le strade che si incrociano
- **semafori:** indica se è un incrocio semaforico o meno
- **latitudine:** indica la latitudine dell'incrocio
- **longitudine:** indica la longitudine dell'incrocio

La classe incrocio è una sottoclasse della classe nodo, che descrive gli elementi di connessione tra le strade nella KB. Un incrocio è un punto di intersezione tra due o più strade, dove le strade si incontrano. La classe incrocio descrive le proprietà specifiche degli incroci, come la lista delle strade che si incrociano in quel punto, se sono presenti semafori (1 si, 0 no), la posizione geografica (latitudine e longitudine). In questo modo, gli oggetti della classe incrocio possono essere utilizzati per descrivere in modo più dettagliato gli incroci tra le strade nella KB. Le strade che si intersecano hanno un nodo in comune, questo nodo viene usato come identificativo dell'incrocio.

```
/* Classe incrocio sottoclasse di nodo
 *
 * Contiene i seguenti attributi:
 *
 * - strade: lista che indica le strade che si incrociano
 * - semafori: lista che indica i semafori presenti nell'incrocio
 * - latitudine: indica la latitudine dell'incrocio
 * - longitudine: indica la longitudine dell'incrocio
 */
prop(incrocio, subClassOf, nodo).
prop(nodo_21300247,type,incrocio).
prop(nodo_21300247,strade,[wilshire_boulevard,south_alvarado_street]).
prop(nodo_21300247,semafori,1).
prop(nodo_21300247,latitudine,34.0579554).
prop(nodo_21300247,longitudine,-118.2758872).
```

Query Knowledge Base (KB)

Le query su una KB in Prolog servono per interrogare la KB e recuperare informazioni specifiche. In Prolog, le query sono espresse utilizzando la sintassi della programmazione logica e possono essere utilizzate per effettuare ricerche basate sui fatti e le regole presenti nella KB.

Calcola la distanza tra due nodi X e Y

```
/**
 * Calcola la distanza tra due nodi X e Y
 *
 * @param X: primo nodo
 * @param Y: secondo nodo
 * @param S: distanza tra i due nodi (viene restituito il risultato)
 */
distanza_nodi(X, Y, S) :- prop(X, latitudine, L1), prop(Y, latitudine, L2),
                          prop(X, longitudine, G1), prop(Y, longitudine, G2),
                          S is abs(L1 - L2 + G1 - G2).
```

La query distanza_nodi(X, Y, S) in Prolog calcola la distanza tra due nodi X e Y, e la assegna alla variabile S. Nello specifico, la query effettua le seguenti operazioni:

1. Il predicato **prop(X, latitudine, L1)** recupera la latitudine del nodo X e la assegna alla variabile L1.
2. Il predicato **prop(Y, latitudine, L2)** recupera la latitudine del nodo Y e la assegna alla variabile L2.
3. Il predicato **prop(X, longitudine, G1)** recupera la longitudine del nodo X e la assegna alla variabile G1.
4. Il predicato **prop(Y, longitudine, G2)** recupera la longitudine del nodo Y e la assegna alla variabile G2.
5. Il predicato **S is abs(L1 - L2 + G1 - G2)** calcola la distanza tra i due nodi come la somma della differenza tra le latitudini e la differenza tra le longitudini e la assegna al risultato S.

Incroci immediatamente vicini dell'incrocio passato in input

```
/**
 * Restituisce gli incroci immediatamente vicini dell'incrocio passato in input.
 * Due incroci sono immediatamente vicini se collegati da una stessa strada.
 *
 * @param Incrocio: Incrocio di cui si vogliono conoscere i vicini
 * @param Vicini: lista di incroci vicini (viene restituito il risultato)
 */
vicini_incrocio(Incrocio, Vicini) :- prop(Incrocio, type, incrocio),
                                     prop(Incrocio, strade, Strade),
                                     vicini_strade_incrocio(Incrocio, Strade, Vicini).

vicini_strade_incrocio(Incrocio, [], Vicini) :- prop(Incrocio, type, incrocio), Vicini = [].
vicini_strade_incrocio(Incrocio, [S1|S2], Vicini) :- prop(S1, nodi, N1),
                                                     suddividi_prefisso_suffisso(Incrocio, N1, Prefisso, Suffisso),
                                                     inverti(Prefisso, Prefisso1),
                                                     find_first(Prefisso1, Vicino1),
                                                     find_first(Suffisso, Vicino2),
                                                     vicini_strade_incrocio(Incrocio, S2, Vicini3),
                                                     append(Vicini3, [Vicino1|Vicino2], Vicini).
```

La query **vicini_incrocio(Incrocio, Vicini)** in Prolog serve per trovare gli incroci vicini ad un incrocio specifico.

Nello specifico, la query effettua le seguenti operazioni:

1. Il predicato **prop(Incrocio, type, incrocio)** verifica che l'oggetto passato come primo argomento sia un incrocio.
2. Il predicato **prop(Incrocio, strade, Strade)** recupera le strade che si incrociano nell'incrocio Incrocio e le assegna alla variabile Strade.
3. La query **vicini_strade_incrocio(Incrocio, Strade, Vicini)** utilizza la lista delle strade che si incrociano nell'incrocio restituisce una lista degli incroci vicini all'incrocio preso in input e li assegna alla variabile Vicini.

In questo modo la query principale **vicini_incrocio(Incrocio, Vicini)** utilizza le proprietà Incrocio per trovare gli incroci vicini ad esso utilizzando una query secondaria per effettuare la ricerca dei incroci vicini all'incrocio Incrocio.

Le query **vicini_strade_incrocio(Incrocio, [], Vicini)** e **vicini_strade_incrocio(Incrocio, [S1|S2], Vicini)** in Prolog servono per trovare gli incroci vicini ad un incrocio specifico, utilizzando la lista delle strade che si intersecano nell'incrocio.

La query **vicini_strade_incrocio(Incrocio, [], Vicini) :- prop(Incrocio, type, incrocio), Vicini = [].** è il passo base, in cui la lista delle strade è vuota, in questo caso la query restituisce una lista vuota per gli incroci vicini.

In particolare, la query effettua le seguenti operazioni:

1. Il predicato **prop(S1, nodi, N1)** recupera i nodi della strada S1 e li assegna alla variabile N1.
2. La query **suddividi_prefisso_suffisso(Incrocio, N1, Prefisso, Suffisso)** utilizza la lista dei nodi della strada S1 (N1) e l'incrocio Incrocio per suddividere la lista dei nodi in un prefisso e un suffisso. Il prefisso contiene i nodi che precedono l'incrocio Incrocio nella lista, mentre il suffisso contiene i nodi che seguono l'incrocio Incrocio nella lista.
3. La query **inverti(Prefisso, Prefisso1)** inverte l'ordine dei nodi nel prefisso.
4. La query **find_first(Prefisso1, Vicino1)** cerca il primo nodo presente nel prefisso invertito e lo assegna alla variabile Vicino1.
5. La query **find_first(Suffisso, Vicino2)** cerca il primo nodo presente nel suffisso e lo assegna alla variabile Vicino2.
6. La query **append(Vicini3, [Vicino1|Vicino2], Vicini)** utilizza la funzione append di Prolog per aggiungere gli incroci vicini trovati in questa chiamata (Vicino1 e Vicino2) alla lista degli incroci vicini trovati dalle chiamate precedenti (Vicini3) e assegnare il risultato alla variabile Vicini.

```
/**
 * Restituisce la latitudine e longitudine di un nodo passato in input
 *
 * @param X: nodo di cui si vogliono conoscere le coordinate
 * @param L: latitudine
 * @param G: latitudine
 *
 */
lat_lon(X, Latitudine, Longitudine) :- prop(X, latitudine, Latitudine),
                                       prop(X, longitudine, Longitudine).
```

La query **lat_lon(X, Latitudine, Longitudine)** in Prolog è utilizzata per recuperare la latitudine e la longitudine di un oggetto X specifico.

In particolare, la query effettua le seguenti operazioni:

1. Il predicato **prop(X, latitudine, Latitudine)** recupera la latitudine dell'oggetto X specificato e la assegna alla variabile Latitudine.
2. Il predicato **prop(X, longitudine, Longitudine)** recupera la longitudine dell'oggetto X specificato e la assegna alla variabile Longitudine.

Aggiornamento Knowledge Base (KB)

Inizialmente, la Knowledge Base non dispone di informazioni sulla gestione dell'impianto semaforico. Tuttavia, essa contiene informazioni sull'esistenza degli incroci. L'aggiornamento della KB avviene quando si utilizza il modello sviluppato in precedenza (Link) per determinare il traffico medio che passa per un incrocio specifico e il suo ciclo semaforico.

Per gestire la Knowledge Base, è stata creata una classe Python chiamata **KnowledgeBase** che utilizza la libreria pyswip per eseguire le query di interrogazione sulla KB viste precedentemente. Inoltre, la classe fornisce funzioni come **assegna_ciclo_semaforico(), modifica_ciclo_semaforico() e rimuovi_ciclo_semaforico()** per gestire i dati del ciclo semaforico nella KB di uno specifico incrocio.

```python
def assegna_ciclo_semaforico(self, incrocio, dict_strada):
    '''
    Metodo assegna_ciclo_semaforico
    -------------------
    Dati di input
    --------------
    incrocio: incrocio di cui si vuole assegnare il ciclo semaforico
    dict_strada: ciclo di ogni strada nell'incrocio
    '''

    for strada, ciclo in dict_strada.items():
        i = 0

        for item in ciclo:
            self.prolog.assertz("props(semaforo_"+incrocio+"_"+strada+", tempo_"+ str(i) +", " + str(item["tempo"]) + ")")
            self.prolog.assertz("props(semaforo_"+incrocio+"_"+strada+", colore_"+ str(i) +", " + str(item["colore"]) + ")")

            i += 1
```

```python
def modifica_ciclo_semaforico(self, assegnazione):
    '''
    Metodo modifica_ciclo_semaforico
    -------------------
    Modifica il ciclo semaforico di un incrocio

    Dati di input
    --------------
    assegnazione: dizionario contenente come chiave l'incrocio A
                  e come valore l'incrocio B sincronizzato con A
                  in modo che A dipende da B (A è slave di B).
                  Se il valore di A è A stesso significa che non
                  è sincronizzato con nessun altro incrocio.
    '''
    for slave, master in assegnazione.items():
        slave = slave.name
        if slave != master:
            nuovo_ciclo_slave = self.sincronizza_incroci(master, slave)
            self.rimuovi_ciclo_semaforico(slave)
            self.assegna_ciclo_semaforico(slave, nuovo_ciclo_slave)
```

```python
def rimuovi_ciclo_semaforico(self, incrocio):
    '''
    Metodo assegna_ciclo_semaforico
    -------------------
    Dati di input
    --------------
    incrocio: incrocio da rimuovere nella base di conoscenza
    '''
    strade = []
    query_incrocio = "prop("+str(incrocio)+", strade, Strada)"
    for atom in self.prolog.query(query_incrocio):
        strade = atom["Strada"]

    for strada in strade:
        individuo = "semaforo_"+incrocio+"_"+strada.value
        self.prolog.retractall("props("+individuo+", P, V)")
```

In particolare, le seguenti funzioni si occupano di:

- **assegna_ciclo_semaforico**: Assegna un ciclo semaforico a un determinato incrocio. In particolare, per ogni elemento del ciclo semaforico, la funzione inserisce due regole in Prolog: una per il tempo del semaforo e una per il colore del semaforo;
- **modifica_ciclo_semaforico**: Modifica il ciclo semaforico di un incrocio dipendente (slave) in base all'incrocio master a cui è sincronizzato, questo per migliorare la fluidità del traffico. Rimuove il vecchio ciclo semaforico ed assegna uno nuovo;
- **rimuovi_ciclo_semaforico**: Rimuove tutte le regole relative ad un incrocio specifico, eliminando così il suo ciclo semaforico dalla base di conoscenza, questo può essere utile per modificare o rimuovere un incrocio dal sistema.

Apprendimento Supervisionato

Il modello utilizzato ha il compito di predire l'indice di traffico per ciascuna strada che si interseca in un incrocio data una tipologia di strada (primaria/secondaria/terziaria), un giorno e un orario.

Quest'indice servirà per assegnare il ciclo semaforico a ciascun incrocio in base alla tipologia di strade che si intersecano (primaria/secondaria/terziaria), cioè in base al limite di velocità e all'indice di traffico per ciascuna strada.

Per poter assegnare il ciclo semaforico all'incrocio sono state usate le seguenti formule:

1. Tempo di verde = max(10, (IT * 20) * 3)
   Dove IT è l'indice di traffico per il ciclo semaforico che andiamo ad ottenere mediante un modello di apprendimento supervisionato il quale viene addestrato utilizzando un dataset che in seguito sarà modificato. Il dataset analizza per ciascun giorno e per ogni ora, il numero di veicoli che transitano.

   La costante 3 rappresenta il numero di secondi impiegati per il passaggio dei pedoni.

   20 è una costante di normalizzazione.

   10 è il limite minimo che viene assegnato al tempo di verde per evitare valori di verde troppo bassi durante le ore notturne.

2. Tempo di giallo = 4 secondi
3. Tempo di rosso = $\sum_{i \in S} T\_verde(i) + T\_giallo(i)$
   S indica le strade che si intersecano in un incrocio esclusa quella a cui si vuole prendire il tempo di rosso.

Data Cleaning

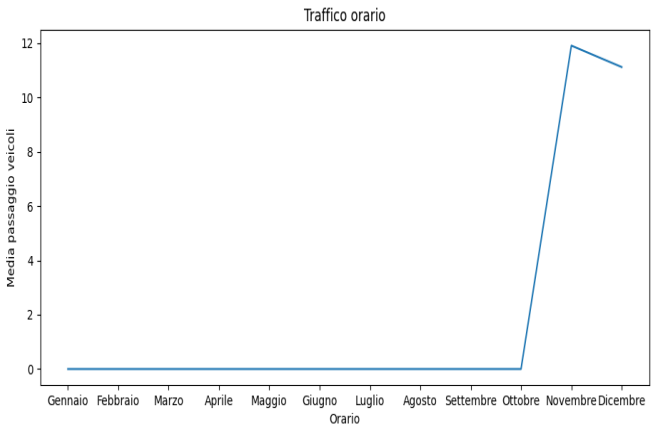
Il dataset iniziale trafficoutput.csv contiene i record relativi di ciascun giorno dell'anno in intervalli di 1 ora. Ad ogni data è associato l'incrocio e il numero di veicoli che passano.

	DateTime	Junction	Vehicles
0	2015-11-01 00:00:00	1	15
1	2015-11-01 01:00:00	1	13
2	2015-11-01 02:00:00	1	10
3	2015-11-01 03:00:00	1	7
4	2015-11-01 04:00:00	1	9

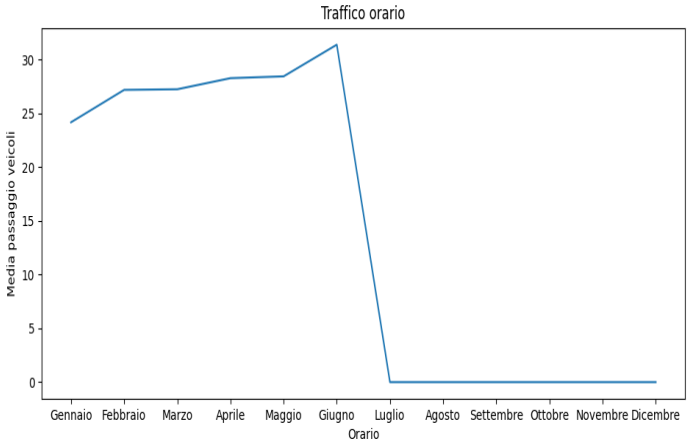
Il dataset originario contiene dati a partire dall'anno 2015 per finire all'anno 2017. Tuttavia, le registrazioni relative all'anno 2015 e 2017 non comprendono tutti i mesi (2015 solo 2 mesi e 2017 solo 6 mesi) perciò per avere un dataset più coerente, nella fase di preprocessing sono stati eliminati questi anni lasciando solamente il 2016, il quale ha tutti i mesi.

I seguenti grafici mostrano il passaggio dei veicoli relativi ai mesi per ogni anno:

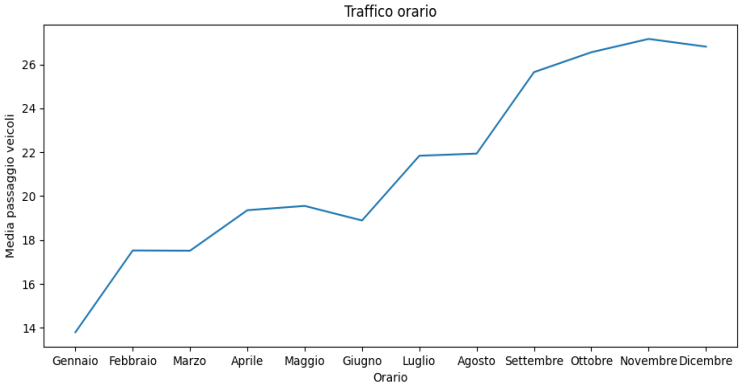
2015



2017



2016

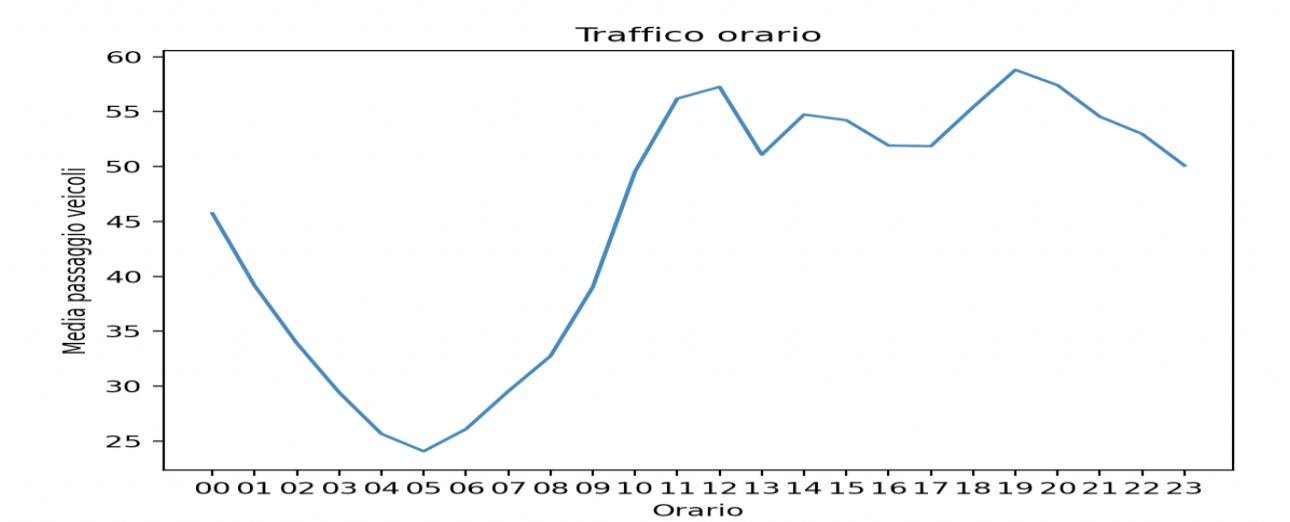


Come è possibile notare, ad eccezione dell'anno 2016, gli anni 2017 e 2015 non rappresentano tutti i mesi.

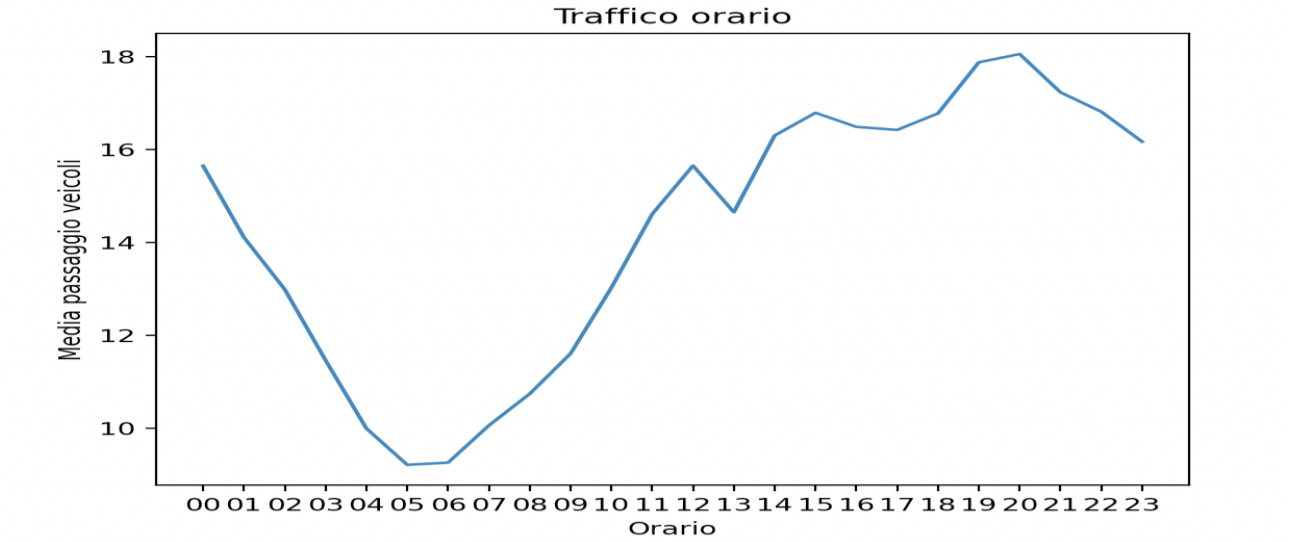
In seguito, sono state eliminate le colonne non ritenute utili al fine dell'addestramento del modello e ne sono state aggiunte altre derivandole dal dataset originario per poter avere più colonne differenti e dunque poter discriminare meglio gli esempi nel dataset. Di fondamentale importanza è stata l'introduzione del campo Traffic in quanto rappresenta il tipo di traffico della strada relativa alla registrazione, quindi se è poco trafficata, mediamente trafficata e molto trafficata.

	Unnamed: 0	Day	Month	Hour	Weekend	Week	Type	Traffic
0	1464	1	1	0	0	53	1	0.318182
1	1465	1	1	1	0	53	1	0.318182
2	1466	1	1	2	0	53	1	0.227273
3	1467	1	1	3	0	53	1	0.000000
4	1468	1	1	4	0	53	1	0.090909

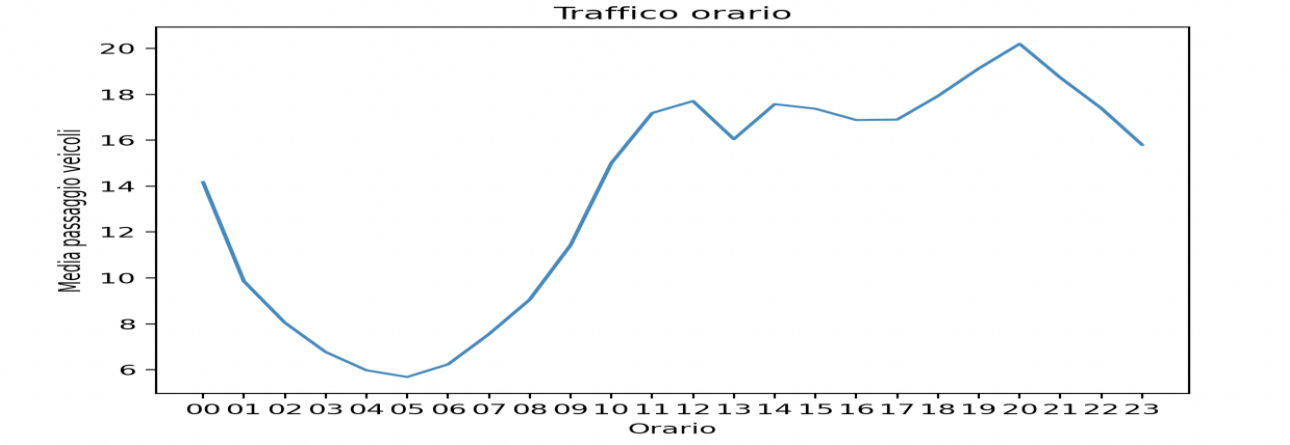
I seguenti grafici mostrano l'andamento orario in relazione dei veicoli selezionando solo il type1 cioè un traffico abbastanza elevato:



Type 2 cioè poco traffico:



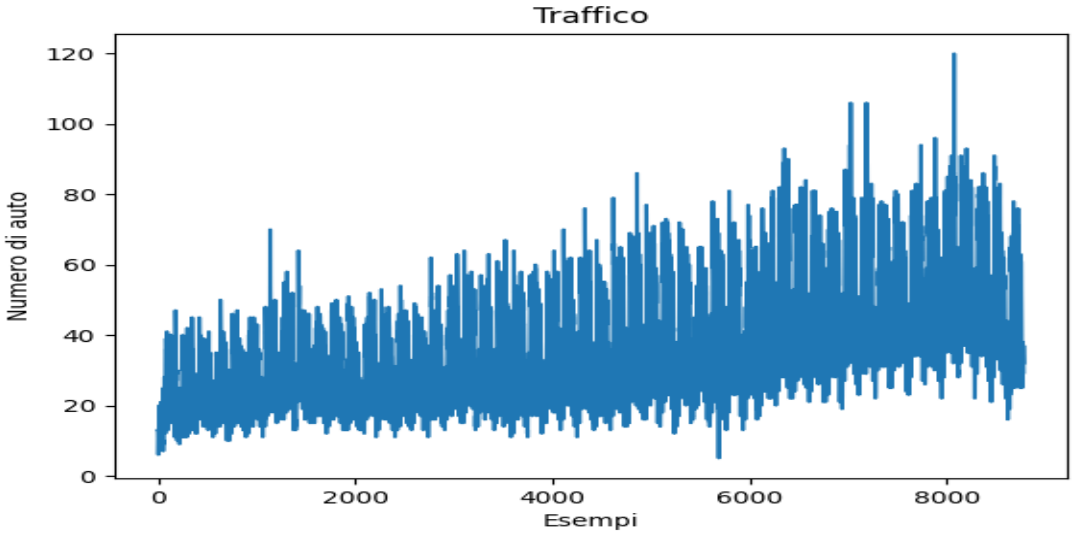
Type 3 cioè traffico medio:



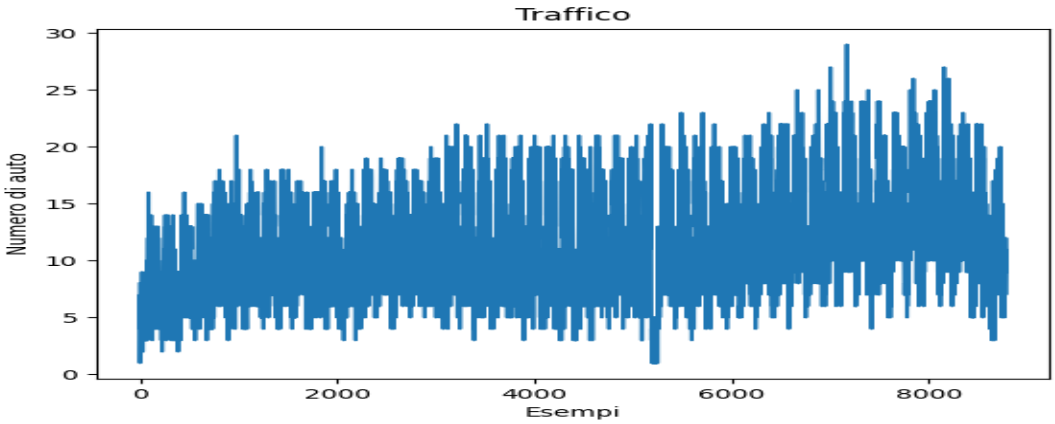
Dai seguenti grafici, tuttavia la situazione non appare ancora chiarissima per quanto riguarda la distribuzione dei veicoli in relazione al tempo.

Con questi grafici invece si può notare decisamente meglio la distribuzione non più andando a considerare l'orario ma tenendo conto di tutti gli esempi del dataset in relazione al numero di auto:

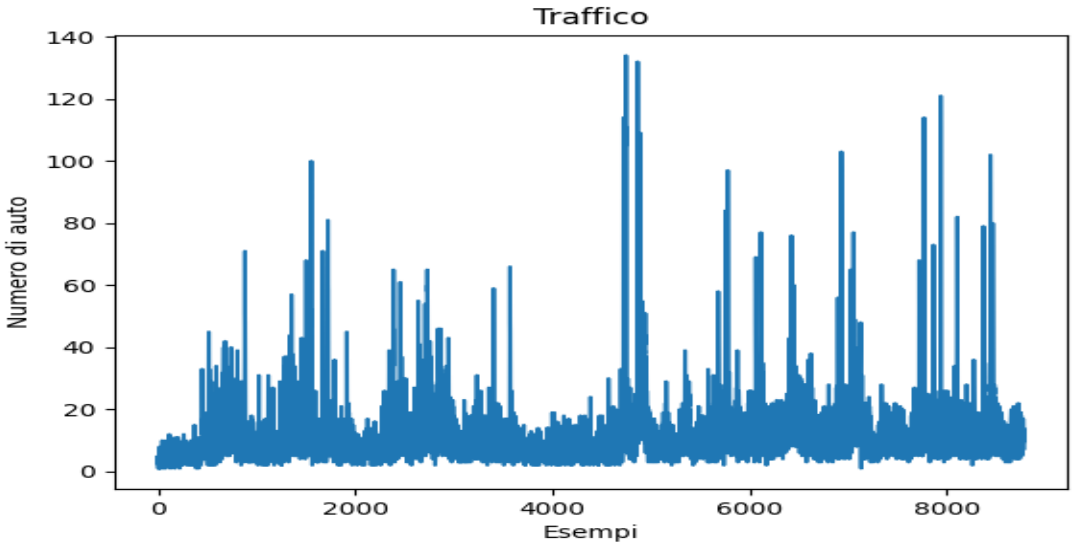
Type1



Type2

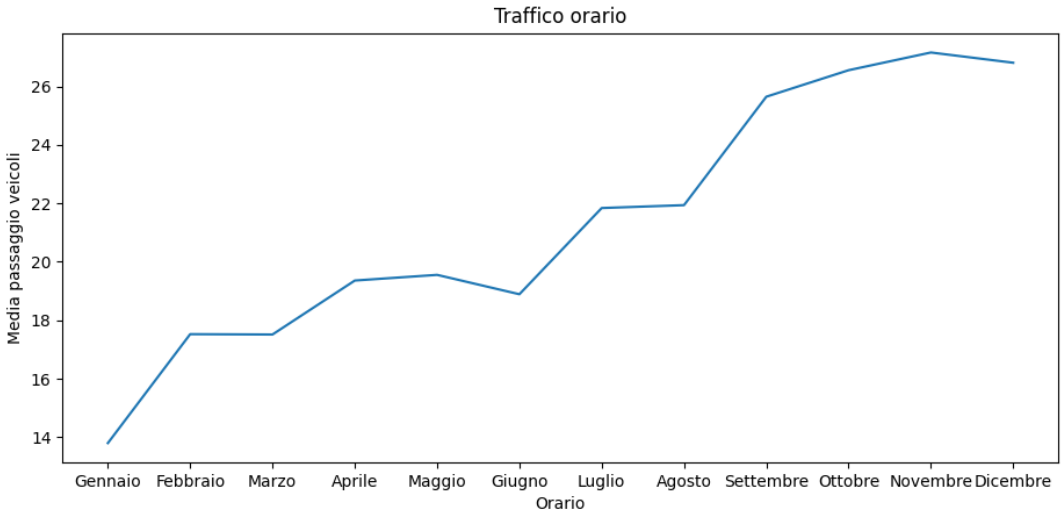


Type3:



Come si può vedere se il type2 mantiene una distribuzione abbastanza uniforme, il type1 è crescente e invece il type3 ha dei picchi in alcuni giorni che rendono la distribuzione completamente irregolare.

Quindi per poter risolvere questi problemi si è pensato di scalare il numero di auto per ogni settimana applicando il min max scaler e applicando un peso differente a seconda che si tratti un type1 o un type3 in modo tale da equilibrare queste distribuzioni. Questo perché si è notato che il traffico è crescente con il passare dei mesi:



Quindi a seguito di queste manipolazioni si è ottenuti un dataset più regolare.

Scelta del modello

Sommario

Per poter predire il traffico in relazione alla data e all'orario rispetto alle nostre strade, sono stati analizzate le prestazioni di 2 modelli in termini di accuratezza e tempi di risposta. Dunque, il nostro è un task di regressione.

Il primo modello analizzato è il Tree Regressor[8], quest'ultimo è un modello di apprendimento supervisionato che risolve un task di regressione; infatti, questi ultimi sono un tipo di albero di decisione che viene usato per la regressione; quindi, la nostra feature target è continua.

Il secondo modello utilizzato è il KNN[9], il quale è un modello Case-Based-Reasoning e si basa sull'immagazzinare i dati di training ed effettua la predizione secondo i casi più simili presenti nel training set. Dato che anche qui è un task di regressione è stato usato il KNN come regressore per poter predire la feature target che è continua.

Strumenti utilizzati

Per l'implementazione del modello relativo all'albero di regressione è stata utilizzata la libreria Scikit learn usando la classe DecisionTreeRegressor.

Per l'implementazione del modello relativo al KNN è stata utilizzata la libreria Scikit learn usando la classe KNeighborsRegressor.

Decisioni di Progetto

Di seguito vengono mostrati la scelta degli iperparametri dei modelli impostati "manualmente".

TreeRegressor:

Come primo iperaparametro si è utilizzata la profondità dell'albero pari a 15 in quanto a seguito di numerosi test si è notato come in termini di complessità e scoring, rappresenta una buona soluzione in quanto se aumentiamo la profondità, lo score non cresce più di tanto, ma il modello diventa più complesso, se invece riduciamo la profondità dell'albero, quest'ultimo non riesce a generalizzare correttamente e quindi lo score esce molto basso.

Come secondo iperparametro è stato usato il numero minimo di esempi per potersi trovare in un nodo foglia. Anche qui la scelta di questo iperparametro è stata ponderata in termini di complessità del modello e di varianza, dunque evitando che si vada in overfitting ed evitando che ci siano troppi pochi campioni per le foglie dell'albero.

KNN:

Come iperparametro si è utilizzato il numero dei vicini, è stato scelto 3 in quanto con tale valore si è notato a seguito di test che si riesce ad avere una corretta discriminazione tra i vari esempi in quanto aumentando tale valore non ci si discosta molto dallo score con k=3. Al contrario diminuendo tale valore c'è una riduzione marcata della predizione.

Ulteriori modelli:

I modelli quali Reti Neurali non sono stati scelti per il nostro task in quanto il numero di esempi e il numero di feature non è sufficientemente elevato da poter giustificare l'utilizzo di modelli più complessi e dunque introdurrebbe solamente un errore aggiuntivo dovuto alla varianza. Inoltre, dato che già abbiamo avuto un buon score con il KNN e con il tree regression che sono modelli di complessità minore, secondo il principio rasoio di Occam, è preferibile scegliere i modelli più semplici a parità di score.

Modelli di ensamble learning come la Random Forest non è stata designata per la scelta in quanto si è notato che non fornisce nessun miglioramento a livello di score e di riduzione degli errori e dunque si è evitato di scegliere tale modello data la sua complessità eccessiva per il nostro task.

Infine per la scelta del modello da adottare per risolvere il nostro task comparando KNN e albero di regressione, è stato scelto il KNN e nella parte di Valutazione viene mostrato il motivo comparando le prestazioni. Tuttavia, è bene precisare che nel nostro mondo viene utilizzato il KNN in quanto la predizione viene effettuata poche volte e dunque si preferisce avere un'accuratezza migliore in quanto l'efficienza in termini di tempo risulta essere quasi identica, ma questo in un contesto reale o più complesso non accadrebbe dato che ci sono state diverse semplificazioni. Infatti, in un modello "reale" dove questo task di regressione deve essere computato molte volte all'ora, magari con l'uso di più client, la scelta ricadrebbe sull'albero di regressione in quanto si è notato essere decisamente più veloce in fase di predizione rispetto al KNN e dunque si preferirebbe rinunciare leggermente all'accuratezza in favore di tempi di risposta che risultano essere decisamente minori rispetto al KNN.

Valutazione

Per poter testare le prestazioni dei modelli (KNN e albero di regressione) e per poter evitare l'overfitting, si è scelto di applicare la k fold cross validation[11] che consiste nella suddivisione del nostro set di dati totale in k partizioni/fold uguali, dove a rotazione tutte le partizioni saranno usate come training set tranne una che verrà usata come validazione; quindi, il modello viene addestrato k volte. Sulla base di questa suddivisione in K fold abbiamo applicato tutta una serie di metriche per poter valutare correttamente il modello.

La scelta della K non è casuale, nel nostro caso abbiamo scelto k=12 in quanto nel nostro set di dati abbiamo ogni giorno del mese registrato per ogni ora dalle 00 alle 23. Dunque, si è deciso di dividere il set in 12 fold in modo tale che possiamo utilizzare a rotazione un mese intero (su cui il modello non è stato addestrato) come validazione.

Di seguito è mostrata una tabella con vari risultati uno per ogni misura dell'errore di predizione compreso lo score, usando la k fold cross validation e mediando sui valori ottenuti da ciascuna predizione.

Albero di regressione:

Score	0.82275
L1	0.06592
MSE(mean squared error)	0.00776
Max Error	0.51248

Inoltre, è stato calcolato anche il BIC[7] score.

Il BIC è stato calcolato come:

$BIC = k \cdot \log(n) - 2 \cdot \log(L)$

Quindi si tiene conto del numero di parametri del modello(k) e n è il numero delle osservazioni (il dataset) e L che è la verosomiglianza del modello.

Come k si è scelta la profondità dell'albero, n è il numero di esempi, e la log likelihood calcolata come il log dell'MSE (perché più è grande il MSE nel nostro caso più si avvicinerà allo 0 perché l'errore è compreso tra 0 e 1, ma facendolo avvicinare allo 0, il logaritmo di un numero vicino allo 0 tende a -infinito).

BIC	-127854.26937

Il modello con il valore più basso di BIC è considerato come il migliore tenendo in considerazione la complessità del modello e la verosomiglianza.

KNN:

Score	0.86036
L1	0.05786
MSE(mean squared error)	0.00611
Max Error	0.47169

Analogamente a quanto detto in precedenza sul BIC, motiviamo come esce questo valore:

Come k si è scelto il numero di vicini, n è il numero di esempi, e la log likelihood calcolata come il log dell'MSE (perché più è grande il MSE nel nostro caso più si avvicinerà allo 0 perché l'errore è compreso tra 0 e 1, ma facendolo avvicinare allo 0, il logaritmo di un numero vicino allo 0 tende a -infinito).

BIC	-134274.06346

Valutazione efficienza in tempo di risposta della predizione tra albero di regressione e knn in base al numero di iterazioni. 10 è il numero di iterazioni che più o meno vengono fatte dal nostro task. In un ambiente reale sono 100/1000.

N°Iterazioni	KNN	Albero
10	0:00:01.881788	0:00:00.015627
100	0:00:23.6599950	0:00:02.210910
1000	0:03:40.833088	0:00:07.091335

Valutazione Iperparametri:

Mostriamo, usando le stesse metriche descritte in precedenza e sempre la k fold cross validation, la scelta di valori degli iperparametri differenti, i quali non sono stati utilizzati in modo da giustificare le scelte viste in precedenza.

Per quanto riguarda l'albero di regressione, l'errore sul test set risulta essere minore per profondità pari a 15 in quanto è determinato dal bias (quindi il nostro modello risulta essere sufficientemente in grado di risolvere il task con un ottimo score senza andare in overfitting) + la varianza + un errore irriducibile causato dal rumore dei dati.

Di seguito sono mostrati i risultati per un'altezza superiore e inferiore dell'albero.

Altezza=13

Score	0.80678
L1	0.06681
MSE(mean squared error)	0.00791
Max Error	0.51762

Altezza=16

Score	0.82325
L1	0.06581
MSE(mean squared error)	0.00774
Max Error	0.51383

Per quanto riguarda il modello KNN si è scelto di impostare il numero di vicini pari a 3 per le motivazioni descritte in precedenza e dunque sono mostrati i risultati per un numero di vicini superiore e inferiore:

K=4

Score	0.85783
L1	0.05822
MSE(mean squared error)	0.00623
Max Error	0.49722

K=2

Score	0.82870
L1	0.06410
MSE(mean squared error)	0.00750
Max Error	0.51806

Hidden Markov Model

Sommario

Sono state implementate utilizzando le HMM (Hidden Markov Model)[1] e il loro principale scopo è quello di massimizzare la probabilità di incontrare un semaforo verde all'incrocio successivo. Il grado di incertezza è dovuto dal fatto che non tutte le macchine seguono la velocità consigliata.

Strumenti utilizzati

Per la realizzazione e per l'utilizzo delle HMM sono state utilizzate le librerie AIPython[2], messe a disposizione dal libro di testo adottato.

Decisioni di progetto

Per fare ciò si è deciso di dividere la lunghezza di un ciclo semaforico in blocchi da cinque secondi ed impostarli come stati del modello probabilistico, mentre come osservazioni sono stati scelti i tre colori del semaforo. Per poter introdurre l'incertezza di passare allo stato successivo, si è preso in considerazione la deviazione standard cioè la velocità in cui l'auto può mediamente deviare dalla velocità consigliata, e si è osservato essere di 3.74 km/h (si rimanda alla Valutazione); perciò la nostra velocità consigliata, se ad esempio è di 30 km/h, è in realtà inclusa in un intervallo che va da 26.26 a 33.74. Perciò l'auto in base a questo intervallo di velocità avrà un'oscillazione di probabilità di passare allo stato successivo. Infatti, immaginiamo questo intervallo come una distribuzione gaussiana dove il picco è a 30 km/h quindi con probabilità 1 di passare allo stato successivo, e più "scendiamo" la curva, più la probabilità di passare allo stato successivo si abbasserà facendo in modo che la probabilità di rimanere nello stato attuale aumenti. Invece, la matrice in cui bisogna indicare la probabilità che sia presente una determinata osservazione noto lo stato ha necessitato di calcoli un po' più complessi.

Questa catena di Markov sarebbe stata periodica, (cioè che per far sì che la probabilità di tornare sullo stesso stato sia diversa da 0, sono necessari un numero di passi temporali pari al numero degli stati) se non avessimo introdotto questo grado di incertezza relativo alla velocità effettiva dell'auto.

Qui viene utilizzata una sequenza di colori del semaforo contenente anche il rispettivo tempo in cui questi permangono. Nel caso in cui uno stato dovesse rientrare completamente in un colore, la probabilità viene impostata ad 1 per questo colore e a 0 per i rimanenti. Nel caso in cui uno stato dovesse trovarsi a cavallo tra due colori, la probabilità è data dal rapporto dei secondi passati in ogni colore con il numero di secondi totali del blocco (5).

Esempio: Nell'immagine è presente un ciclo semaforico della durata di 35 secondi e diviso in 7 stati. La sequenza utilizzata in questo caso si adatta perfettamente al ciclo semaforico ed è la seguente: ***[{'tempo:' 16, 'colore': 'rosso'}, {'tempo:' 15, 'colore': 'verde'}, {'tempo:' 4, 'colore': 'giallo'}]*** *La probabilità che sia rosso mentre ci si trova nei primi tre stati è pari a 1, nel terzo stato è pari a* $\frac{1}{5}$.

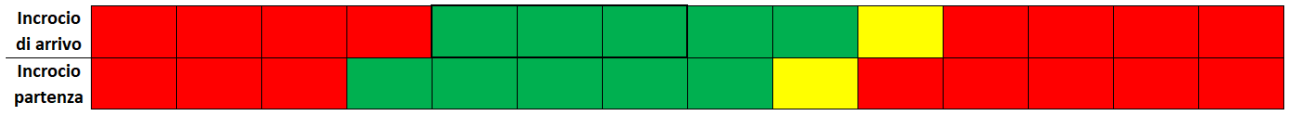
Per trovare la probabilità di incontrare il verde all'incrocio successivo è stata realizzata una funzione apposita (getprobverde). La funzione accetta tre argomenti: le due sequenze dei semafori su cui si sta effettuando il calcolo e la differenza in secondi tra i due incroci. La funzione crea le due HMM in base alle due sequenze fornite e avvia una simulazione. La funzione utilizza quindi le catene create per calcolare il numero di passaggi dallo stato rosso allo stato verde in entrambe le sequenze, tenendo conto della differenza di tempo tra le due. Infine, calcola la probabilità di trovare il semaforo verde nella seconda sequenza in corrispondenza dei

passaggi dallo stato rosso allo stato verde nella prima sequenza. La funzione restituisce infine questa probabilità. Per tener conto di eventuali imprevisti si è considerata la possibilità di inserire nel calcolo della probabilità anche l'eventualità in cui il veicolo dovesse raggiungere il secondo incrocio nello stato precedente o nello stato successivo rispetto a quello predetto. In questo caso, nel calcolo della probabilità viene dato più peso allo stato predetto e meno peso allo stato precedente e successivo a quello predetto.

Le catene di Markov nascoste sono state sfruttate per poter aumentare nella maniera più efficiente possibile la probabilità che da un incrocio di partenza si raggiunga quello successivo trovando il verde. Per fare ciò è stato necessario l'utilizzo di due tecniche diverse: una da utilizzare nel caso in cui in cui la lunghezza del ciclo semaforico dei due incroci è uguale e una nel caso in cui è diversa.

Nel primo caso, per provvedere ad un'efficace sincronizzazione è sufficiente effettuare lo shift della sequenza dell'incrocio di arrivo finché non si avrà probabilità certa di incontrare il verde o finché non si sarà effettuato un numero di shift pari al numero degli stati del ciclo semaforico per poi prendere la sequenza con probabilità più alta di incontrare il verde.

Esempio: Prendendo in considerazione le due sequenze di questa immagine, gli stati con un bordo più marcato dell'incrocio di arrivo rappresentano quelli in cui il veicolo è previsto che lo raggiunga una volta partito il verde nell'incrocio di partenza. Avendo questi due cicli una lunghezza uguale, è possibile applicare lo shift. L'immagine seguente mostra il risultato:



Come è possibile vedere, gli stati in cui è previsto che il veicolo raggiunga il secondo incrocio sono verdi.

Nel secondo caso non è possibile effettuare lo shift visto che la lunghezza dei due cicli è diversa. Allora si è scelto di aumentare di un secondo, per ogni iterazione, il tempo in cui permane il verde, accorciando della stessa quantità il rosso (e modificando di conseguenza anche il ciclo semaforico delle altre strade presenti nell'incrocio corrente). In questo caso le iterazioni termineranno quando si avrà probabilità certa o quando il numero di iterazioni effettuate avrà raggiunto il valore di **cycle2 / (seconds * 10)** arrotondato per eccesso al successivo valore intero e poi moltiplicato per **seconds** dove cycle2 è la lunghezza del ciclo semaforico di arrivo e seconds è il numero di secondi che ogni stato occupa, quindi 5.

Valutazione

Prima di tutto si è valutata l'efficacia avuta grazie alla sincronizzazione di due incroci vicini, cioè di quanto effettivamente la probabilità di incontrare il verde è aumentata prima che fossero sincronizzati.

Per effettuare questa valutazione sono stati utilizzati i cicli semaforici della mappa d'esempio precaricata nel sistema, richiamando il metodo getprobverde (come spiegato in precedenza) su due incroci vicini nelle condizioni ottimali, cioè supponendo che tutte le automobili non ritardino o non anticipino l'arrivo all'incrocio successivo per più di un blocco, quindi non più e non meno di 5 secondi rispetto alla velocità consigliata. La valutazione dell'accuratezza è stata effettuata su tutti gli incroci semaforici sincronizzati per poi prendere la media aritmetica.

Durata dei due cicli semaforici	Probabilità di incontrare verde prima della sincronizzazione	Probabilità di incontrare verde dopo la sincronizzazione	Guadagno
Stessa durata	0.57063	0.94092	0.37029
Durata differente	0.43521	0.64318	0.20797
Globale	0.51413	0.85792	0.34379

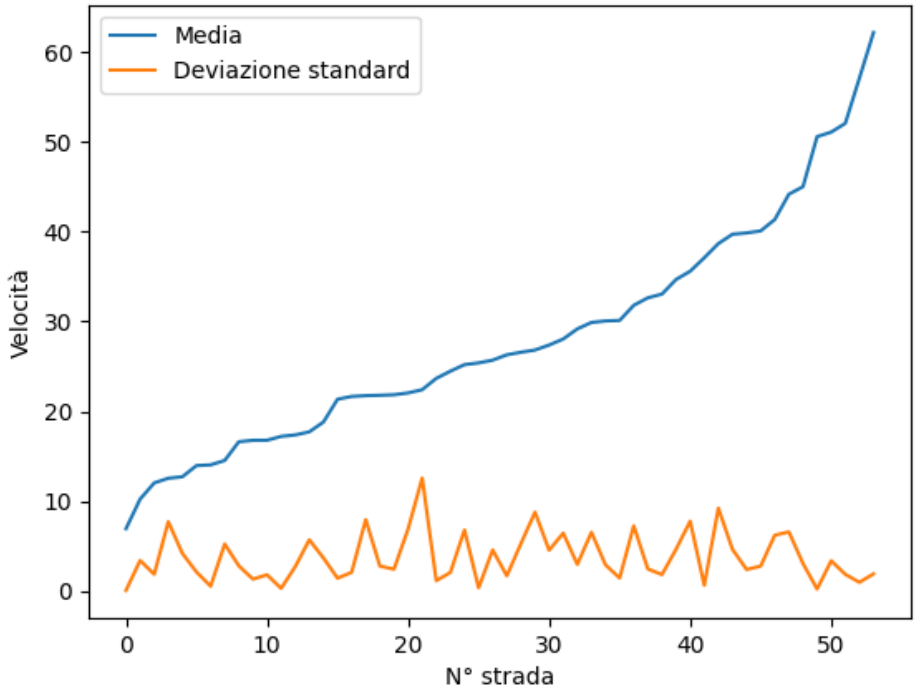
Come è possibile evincere dalla tabella, la sincronizzazione di due incroci semaforici di stessa durata ha un guadagno migliore rispetto alla sincronizzazione di incroci di durata differente proprio come ci si aspettava sin dall'inizio.

In entrambi i casi quindi il guadagno è presente partendo dal presupposto però che il modello HMM costruito sia effettivamente valido, ovvero:

1) le distribuzioni di probabilità degli stati inferite siano corrette (il che è vero in quanto le catene sono periodiche, quindi non c'è incertezza da questo punto di vista);
2) effettuare una previsione corretta di quanto tempo in media un'automobile impiega per percorrere i metri di distanza tra i due incroci: questo introduce un grado di incertezza perché non tutte le automobili seguono la velocità consigliata e quindi non tutte arrivano all'incrocio successivo nello stesso istante.

In merito a questo secondo punto, sono state analizzate di conseguenza le velocità medie di diverse strade urbane in base ad un dataset reperibile su internet[3].

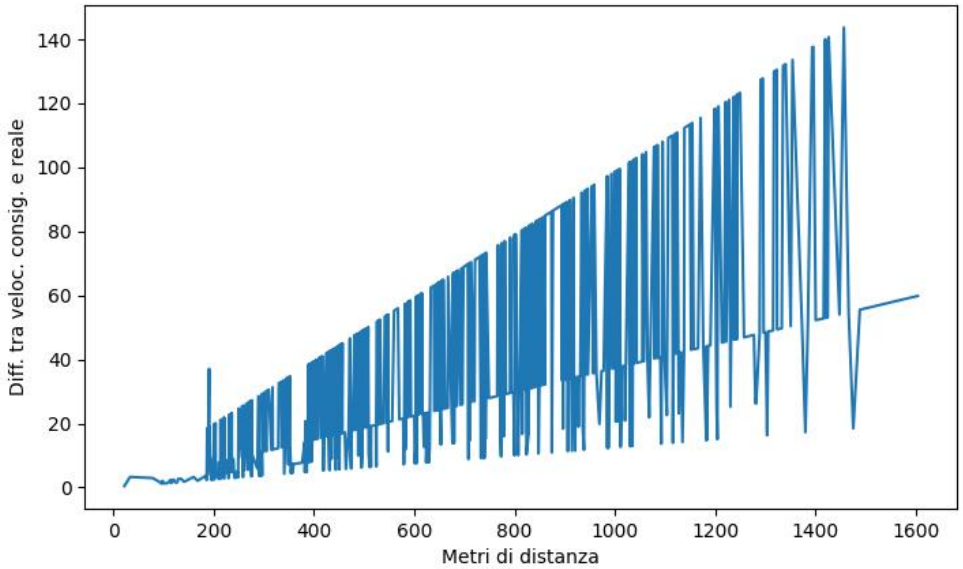
I risultati vengono di seguito mostrati nel grafico.



Quello che interessa analizzare è la deviazione standard[4] dei dati (linea arancione) per avere una stima numerica della dispersione della velocità, ovvero di quanto le automobili si discostano dalla media della velocità di quella strada. Escludendo l'outlier (il picco alla 20° strada circa), si può affermare che la dispersione

della velocità non supera i 7 km/h. Precisamente la media aritmetica della deviazione standard è di **3.76 km/h.**

Utilizzando come esempio questo valore di 3.76 km/h, si è calcolato lo scarto tra tempo di tragitto tra i due incroci alla velocità consigliata e tempo reale (aumentando sempre di più la distanza tra gli incroci).



Ovviamente, come è possibile notare, con l'aumentare della distanza tra i due incroci, aumenta di conseguenza anche lo scarto tra i due tempi, andando sempre di più a peggiorare l'effettiva sincronizzazione tra i due incroci.

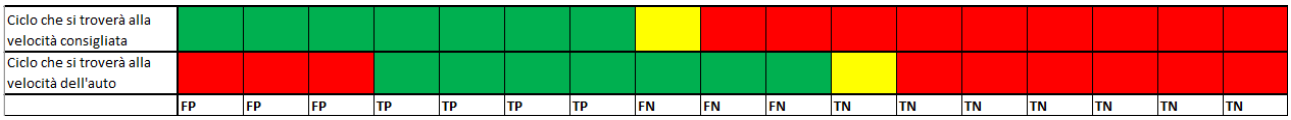
Per questo motivo, si vuole valutare fino a quale distanza e fino a quale valore di deviazione di velocità il modello HMM è in grado di offrire ancora una buona sincronizzazione (per questa valutazione si è riutilizzato ancora una volta il dataset precedente[3]).

Si è definita innanzitutto la matrice di confusione come:

	Attuali positivi (verdi nel ciclo alla velocità reale dell'auto)	**Attuali negativi (rossi e gialli nel ciclo alla velocità reale dell'auto)**
Positivi predetti (verdi nel ciclo alla velocità consigliata)	**TP (verdi trovati nello stesso blocco in entrambi i cicli)**	**FP (verde alla velocità consigliata e rosso/giallo in quella reale)**
Negativi predetti (rossi e gialli nel ciclo alla velocità consigliata)	**FN (rosso/giallo alla velocità consigliata e verde in quella reale)**	**TN (rossi e gialli trovati nello stesso blocco in entrambi i cicli)**

Sicuramente **FP** è ben più grave rispetto a **FN**, in quanto nel primo caso (**FP**) significa che la diversa velocità dell'auto ha influito negativamente con la sincronizzazione dei due semafori, facendogli trovare un rosso su un verde se avesse seguito la velocità consigliata. Nel secondo caso (**FN**), la diversa velocità dell'auto non ha influito negativamente con la sincronizzazione dei due semafori, perché ha trovato un verde piuttosto che un rosso se avesse seguito la velocità consigliata.

Ecco un esempio:

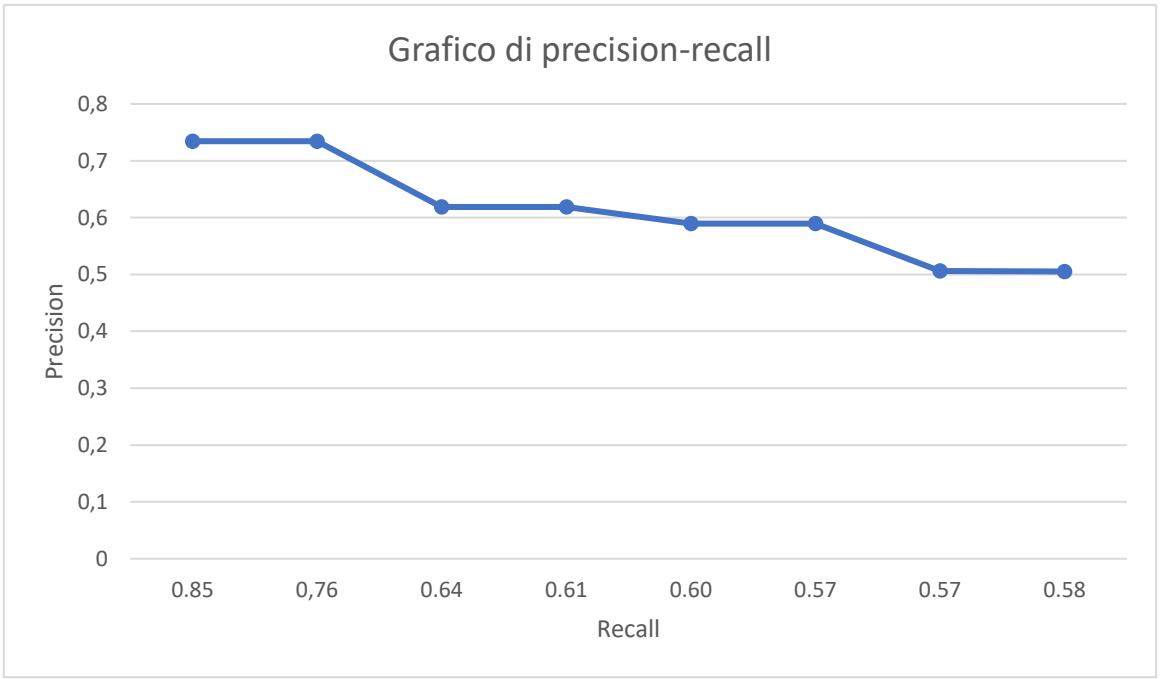


Di seguito la tabella della valutazione con diverse metriche utilizzate:

Distanza	Deviazione di velocità (m/s)	Accuratezza	Precision	Recall	F1
Fino a 200 m	<= a 2	0.83892	0.73433	0.76033	0.74583
	<= a 4	0.78720	0.71819	0.84013	0.77439
	<= a 6	0.67029	0.66760	0.85800	0.75092
Fino a 400 m	<= a 2	0.77005	0.61876	0.61811	0.61843
	<= a 4	0.64653	0.49740	0.60445	0.54573
	<= a 6	0.53242	0.47478	0.64186	0.54582
Fino a 600 m	<= a 2	0.68567	0.58920	0.57708	0.58308
	<= a 4	0.60539	0.50602	0.57941	0.54023
	<= a 6	0.48339	0.50517	0.58987	0.54425

Nel caso di studio in esame, la precision è una metrica da preferire rispetto alla recall, questo perché:

- la precision indica il numero di verdi trovati sul totale di volte che la macchina arriva al semaforo;
- la recall indica il numero di verdi trovati rispetto al numero totale di verdi;



-

Dopo aver utilizzato diverse metriche per la valutazione, si può confermare che in un sistema reale, dove la deviazione di velocità delle automobili non supera in genere i 2 m/s (cioè 7,2 km/h) o anche meno (tranne che su particolari strade principali), il modello HMM costruito è sempre stato in grado di garantire una buona probabilità di incontrare il verde fino ai 200 metri di distanza per poi peggiorare (come ci si aspettava, naturalmente) per valori di distanza superiori ai 400/500 metri.

Constraint Satisfaction Problems (CSP)

Sommario

È stato scelto di ricorrere all'utilizzo di CSP[6] per far fronte ad una problematica riscontrata: con l'utilizzo delle HMM è possibile sincronizzare due incroci in maniera tale da far sì che al secondo incrocio sia più probabile trovare il verde. Però l'obbiettivo di questo caso di studio è minimizzare i tempi di attesa globali, quindi non tra soli due incroci.

Quindi sincronizzando l'incrocio di partenza con il successivo, non sarà possibile sincronizzarlo anche con tutti gli altri incroci vicini perché in tal caso sarebbe necessario desincronizzare il successivo individuato in precedenza. Quindi la maniera scelta per riuscire ad individuare il miglior incrocio tra quelli presenti nelle immediate vicinanze è quella di usare il CSP.

Strumenti utilizzati

Per la realizzazione e per l'utilizzo del CSP sono state utilizzate le librerie AIPython[2], messe a disposizione dal libro di testo adottato.

È stato deciso di optare per una variante dell'algoritmo che implementa il CSP che prevede una via di mezzo tra la ricerca locale e l'any conflict. I parametri che questa variante richiede sono tre:

- **max_steps:** È il numero massimo di passi che eseguirà prima di arrendersi;
- **prob_best:** La probabilità che la variabile con più conflitti venga scelta per minimizzare i suoi conflitti;
- **prob_anycon**: La probabilità che una variabile random venga scelta (con almeno un conflitto).

A seguito di diversi test sono stati scelti i seguenti valori: max_steps = 500000; prob_best = 1; prob_anycon = 0.

Il motivo di questa scelta di parametri è da rimandare alla valutazione.

Decisioni di progetto

Sono stati scelti come variabile il nodo (l'incrocio) e come dominio i suoi nodi vicini (incroci vicini).

Il CSP esegue quindi tante possibili assegnazioni, quindi per ciascuna variabile sceglie un solo valore del suo dominio rispettando però determinati vincoli. Assegnando un valore ad una variabile si intende che il ciclo semaforico di quella determinata variabile è stato modificato (sincronizzato) con il ciclo semaforico del valore. Facendo in questo modo, sarà resa possibile la sincronizzazione di un incrocio solo con un altro, tra quelli vicini.

Come Hard Constraint è stato deciso che sincronizzando un incrocio, non sarà possibile sincronizzarlo nuovamente. Quindi un incrocio può essere sincronizzato solo una volta. Per definire l'Hard Constraint si è deciso di utilizzare due notazioni: Master e Slave. In una coppia di incroci, con Slave si intende l'incrocio che, durante la sincronizzazione, modifica la sequenza del suo ciclo semaforico e corrisponde all'incrocio di arrivo. In una coppia di incroci, con Master, invece, si intende l'incrocio che non viene modificato in e corrisponde all'incrocio di partenza.

Quindi uno stesso incrocio può essere Slave una sola volta tra tutte le coppie nella quale compare, però può essere Master un numero illimitato di volte.

Invece, quello che massimizza la probabilità di trovare il tempo di verde in un qualsiasi incrocio è il Soft Constraint. Il CSP andrà prima a verificare l'Hard Constraint e successivamente i Soft Constraint. I Soft Constraint sono dei vincoli in cui si cerca di ottimizzare il più possibile, seppur non bloccanti. Questi restituiscono un valore da mantenere più alto possibile. Quando il CSP richiama il Soft Constraint, passerà

l'assegnazione attuale che rispetta già l'Hard Constraint e richiamerà una funzione scritta apposita allo scopo di sincronizzare e valutare l'efficacia.

Questa funzione, data in input l'assegnazione globale richiamerà al suo interno l'HMM per procedere con la sincronizzazione e restituire la valutazione di efficacia globale.

```
def valutazione_efficacia(self, incroci_sincronizzati):
    '''
    Metodo valutazione_efficacia
    ----------------------
    Dati di input
    --------------
    incroci_sincronizzati: dizionario contenente come chiave l'incrocio A
                           e come valore l'incrocio B sincronizzato con A
                           in modo che A dipende da B (A è slave di B).
                           Se il valore di A è A stesso significa che non
                           è sincronizzato con nessun altro incrocio.
    Dati di output
    --------------
    efficacia: efficacia totale
    '''
    efficacia = 0

    cicli_aggiornati = copy.deepcopy(self.dict_strade)

    for slave, master in incroci_sincronizzati.items():
        if(isinstance(slave, str) == False):
            slave = slave.name

        if master != slave:
            nuovo_ciclo_slave = self.sincronizza_incroci(master, slave)
            cicli_aggiornati[slave] = nuovo_ciclo_slave
```

Dopo la sincronizzazione, avvenuta secondo i valori restituiti dal CSP, è necessario procedere con la valutazione calcolando l'efficacia. Si recupera l'assegnazione con i cicli sincronizzati e per ciascun incrocio avviene un'iterazione recuperando gli incroci vicini. Quindi viene richiamata una funzione per calcolare la probabilità di incontrare il verde passando come parametri il Master e lo Slave.

In questo caso il ruolo del Master è ricoperto dall'incrocio fisso e il ruolo dello Slave da uno degli incroci vicini, modificando lo Slave per ogni iterazione finché non saranno terminati gli incroci vicini.

Quest'operazione sarà ripetuta usando come Master ogni incrocio e all'interno verrà richiamata la funzione getprobverde.

```
count_strade = 0
for master, vicini in self.incrocio_vicini.items():
    for vicino in vicini:
        common_strade = self.incrocio_strade_comuni(master, vicino)

        strada = ""
        if len(common_strade) > 0:
            strada = common_strade[0]

            distanza_incroci = self.distanza_nodi_secondi(master, vicino, 0, False)
            ciclo_vicino = cicli_aggiornati[vicino][strada]
            ciclo_master = cicli_aggiornati[master][strada]
            prob_verde = getprobverde(ciclo_vicino, ciclo_master, distanza_incroci)
            efficacia += prob_verde
            count_strade += 1

if count_strade > 0:
    return efficacia / count_strade
else:
    return 0
```

Valutazione

Si è valutata per prima cosa l'efficacia ricavata dal CSP, cioè se la probabilità di sincronizzazione globale all'interno della mappa è aumentata. Partendo quindi da una situazione iniziale in cui tutti gli incroci semaforici non sono sincronizzati, si è ricavata una probabilità di incontrare il verde di 0.51792 (che coincide con la situazione di partenza globale delle HMM).

Questa valutazione iniziale è stata fatta richiamando il metodo valutazione_efficacia (che a sua volta utilizza il modello delle HMM per determinare la probabilità di incontrare il verde tra due incroci vicini) su tutti gli incroci nella mappa e per ogni vicino.

Ci si potrebbe aspettare quindi che, la probabilità d'arrivo sia esattamente come nelle HMM, cioè che la probabilità di incontrare il verde dopo la sincronizzazione globale sia circa del 0.85792, tuttavia non è così. Questo perché:

- La valutazione effettuata dalle HMM valuta ogni incrocio semaforico nella mappa verso il suo unico vicino sincronizzato.
- La valutazione effettuata dal CSP, invece, valuta ogni incrocio semaforico nella mappa verso ogni suo vicino (sia quando i due incroci sono sincronizzati e sia quando non lo sono).

Inevitabilmente, quindi, ci si aspetta uno score **inferiore** nel CSP rispetto alle HMM dovuto allo scopo differente dei due algoritmi. Lo scopo richiesto all'algoritmo del CSP è quello di avvicinarsi il più possibile allo score ottimale globale senza andare a discapito dell'efficienza, cioè esplorare lo spazio degli stati in un tempo accettabile in modo tale da mediare efficienza (complessità temporale) ed efficacia (avvicinarsi il più possibile allo score ottimale).

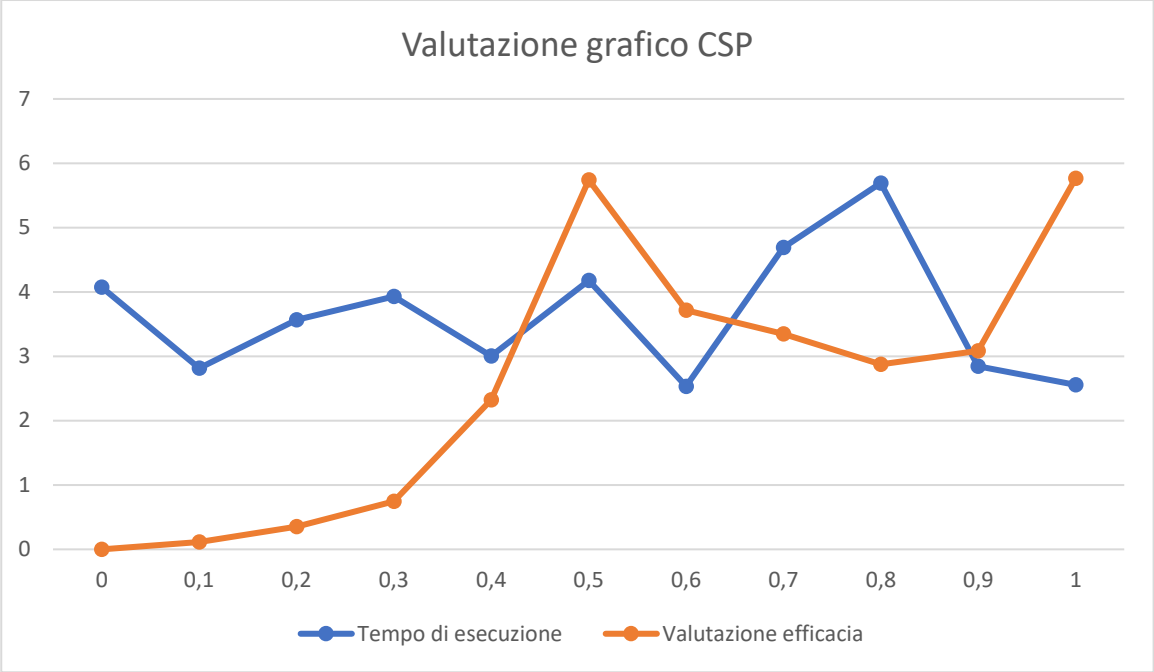
Innanzitutto, si è voluto capire quanto fosse il valore dello **score ottimale globale** (nella mappa in esame), quindi per poterlo ricavare si è eseguito un algoritmo generate-and-test (e salvando ogni modello che soddisfa tutti gli hard constraint), utilizzando come misura di qualità proprio valutazione_efficacia. Il valore dello **score ottimale** ricavato è stato di **0.68607.**

Un valore vicino al 100% non è possibile perché così facendo ci sarebbero solo semafori verdi. Un'efficienza del 68% è sufficiente anche perché il tempo di rosso globale è sempre maggiore del tempo di verde globale. Questo perché, in una sincronizzazione tra due incroci, aumentando il tempo di verde di un semaforo, il tempo di verde di tutti gli altri semafori dello stesso incrocio diminuirà di conseguenza, visto che in ogni incrocio sono presenti minimo 2 semafori. Quindi globalmente, per ogni incrocio, i tempi di rosso e verde non cambiano. Ciò che sarà aumentata sarà però la probabilità di incontrare il verde.

Quindi, a seguito di diverse prove, si è scelto la prob_best (greedy) pari a 1.0 (quindi totalmente greedy). Di seguito la valutazione delle diverse prove:

Greedy	**Any-conflict**	**Tempo**	**Valutazione_efficacia**
1.0	0.0	2.55600 secondi	0.67610
0.9	0.1	2.84350 secondi	0.65324
0.8	0.2	5.69301 secondi	0.65145
0.7	0.3	4.69227 secondi	0.65550
0.6	0.4	2.53400 secondi	0.65861
0.5	0.5	4.18100 secondi	0.67591
0.4	0.6	3.00299 secondi	0.64676
0.3	0.7	3.92900 secondi	0.63331
0.2	0.8	3.56694 secondi	0.62995
0.1	0.9	2.81500 secondi	0.62793
0.0	1.0	4.07403 secondi	0.626952

Di seguito, il grafico risultante:



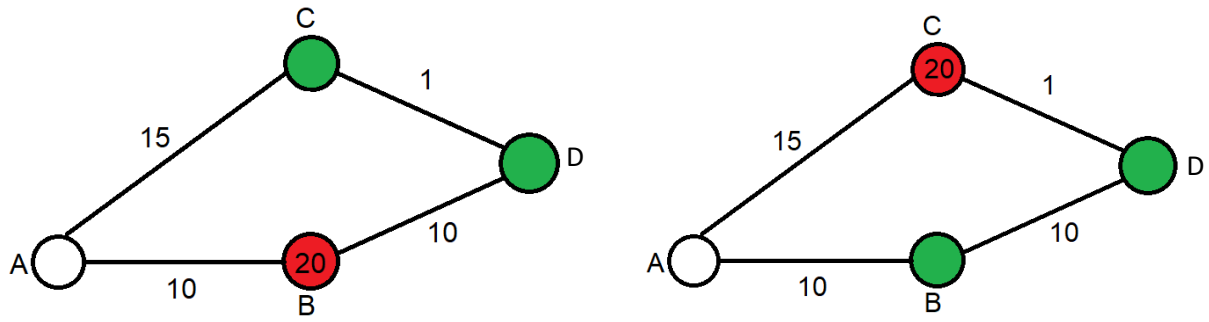
Come è possibile notare dal grafico, il tempo è generalmente costante, tranne nel picco a 0.8. La situazione migliore è proprio in greedy 1.0, perché in questo punto la differenza tra efficacia (linea arancione) e efficienza (tempo di esecuzione, linea blu) è maggiore. Nel punto 1.0 si ha tra l'altro anche il punto di massimo raggiungibile dall'efficacia, ovvero 67%.

Algoritmo di path finding con A*

Sommario

Lo scopo dell'algoritmo di ricerca su grafo implementato è quella di trovare il percorso ottimale da un incrocio di partenza ad un incrocio d'arrivo che minimizzi il tempo totale del percorso tenendo in considerazione il costo dell'arco (strada che congiunge due incroci) e il costo del nodo (incrocio semaforico).

In particolare, se l'incrocio è semaforico, il suo costo dipende dal tempo di arrivo della macchina all'incrocio e in particolare se il colore è verde il costo è 0, altrimenti è pari ai secondi di tempo rosso che la macchina dovrà aspettare.



Da notare come il costo del percorso da A a D possa variare in base al ciclo semaforico degli incroci.

Strumenti utilizzati

L'algoritmo di ricerca utilizzato per lo scopo è **A***[5] ma con alcune varianti. La funzione per la stima del costo (tempo dal nodo di partenza al nodo obiettivo) introduce una variabile **t** ed è la seguente:

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

dove p è il percorso e t è il tempo trascorso dal nodo di partenza. Questo significa che, nel nostro algoritmo, uno stesso percorso può avere costo reale differente a seconda del valore di t.

Per valutare quindi il costo di un percorso bisogna sommare il peso di ciascun arco (secondi che ci si impiega per attraversare una strada) che è indipendente dalla variabile t più il costo associato a ciascun nodo (incrocio) che invece dipende dal valore di t: quest'ultimo costo è infatti il tempo d'attesa al semaforo rosso di quell'incrocio se si arriva al tempo t.

La libreria di partenza utilizzata per implementare A* è AIPython[2].

Decisioni di Progetto

L'algoritmo comunica direttamente solo con la base di conoscenza, quindi per determinare il costo dell'incrocio si suppone che la macchina segua esattamente la velocità consigliata in quella specifica strada e che quindi arrivi esattamente all'incrocio al tempo t (secondi trascorsi dalla partenza). Il grado di incertezza relativo all'arrivo della macchina all'incrocio è invece demandato alle HMM e CSP.

Prima di tutto si è definita la funzione euristica *h(n)* che è indipendente dalla variabile t: questa restituisce il tempo che si prevede di impiegare se la strada fosse in linea retta tra il nodo n attuale e il nodo d'arrivo incontrando tutti semafori verdi. In questo modo h(p) è sicuramente ammissibile perché non sovrastima il costo reale, per tre motivi:

1. La distanza reale tra il nodo attuale e il nodo obiettivo non può essere più corta della distanza in linea retta tra i due nodi, ma al massimo uguale. Per calcolare la distanza prevista, si interroga la base di conoscenza per prelevare la latitudine e longitudine di entrambi i nodi per poi convertirli in metri tramite la formula della distanza euclidea.
2. Per convertire la distanza a tempo previsto si usa la formula $t_{previsto} = distanza_{prevista} / velocità_{prevista}$. Il tempo previsto non sarà mai inferiore (al massimo sarà uguale) al tempo reale perché come velocità si impone sempre il minimo tra tutte le velocità delle strade presenti all'interno della base di conoscenza.
3. Nel tragitto reale tra il nodo attuale e il nodo obiettivo al minimo ci saranno tutti verdi (proprio come nell'euristica).

Quindi, al massimo l'euristica utilizzata è uguale al costo reale ma mai minore, il che la rende ammissibile.

Un'ulteriore caratteristica implementata nell'algoritmo è la rimozione dalla frontiera dei percorsi che presentano cicli. Di per sé, l'algoritmo **A*** è in grado di trovare la soluzione ottimale anche in presenza di cicli, tuttavia per una questione di efficienza in complessità computazionale si è deciso di effettuare questa scelta anche perché nella ricerca di un percorso (come in un navigatore) non si può far passare la macchina due volte dallo stesso incrocio.

La costruzione del grafo si è scelto di farla in modo dinamico, richiamando ogni volta dalla base di conoscenza la lista di nodi vicini al nodo attuale, ovvero l'ultimo presente nel percorso selezionato dalla frontiera (passando anche i secondi trascorsi dall'inizio del percorso). Si è deciso di far questo perché sarebbe stato impossibile avere i costi di ciascun nodo e per ciascun valore di t prima ancora di far partire la ricerca.

Valutazione

Innanzitutto, si è scelto di utilizzare A* piuttosto che IDA* per una questione di complessità computazionale in tempo; infatti, anche se A* ha una complessità in spazio esponenziale rispetto alla lineare dell'IDA*, questa è sicuramente trascurabile (nel nostro caso che non si hanno migliaia di nodi nella base di conoscenza) rispetto allo svantaggio di IDA* di dover ricalcolare ad ogni passo il percorso dall'inizio (dovendo quindi richiamare la base di conoscenza numerose volte). Si è preferito quindi la complessità in tempo dell'A* piuttosto che in spazio dell'IDA*.

La valutazione è stata effettuata sulla mappa di esempio precaricata nel sistema (che ha un totale di 92 nodi).

Si è valutata l'efficienza avuta dell'algoritmo A* grazie all'euristica h(p):

Nodi nel percorso finale	**Tempo di esecuzione con l'euristica h(p)**	**Tempo di esecuzione senza euristica h(p)**
2 nodi incontrati	0.0025 secondi	0.001 secondi
4 nodi incontrati	0.008 secondi	0.014 secondi
7 nodi incontrati	0.025 secondi	1.33 secondi
9 nodi incontrati	0.09 secondi	1.41 secondi

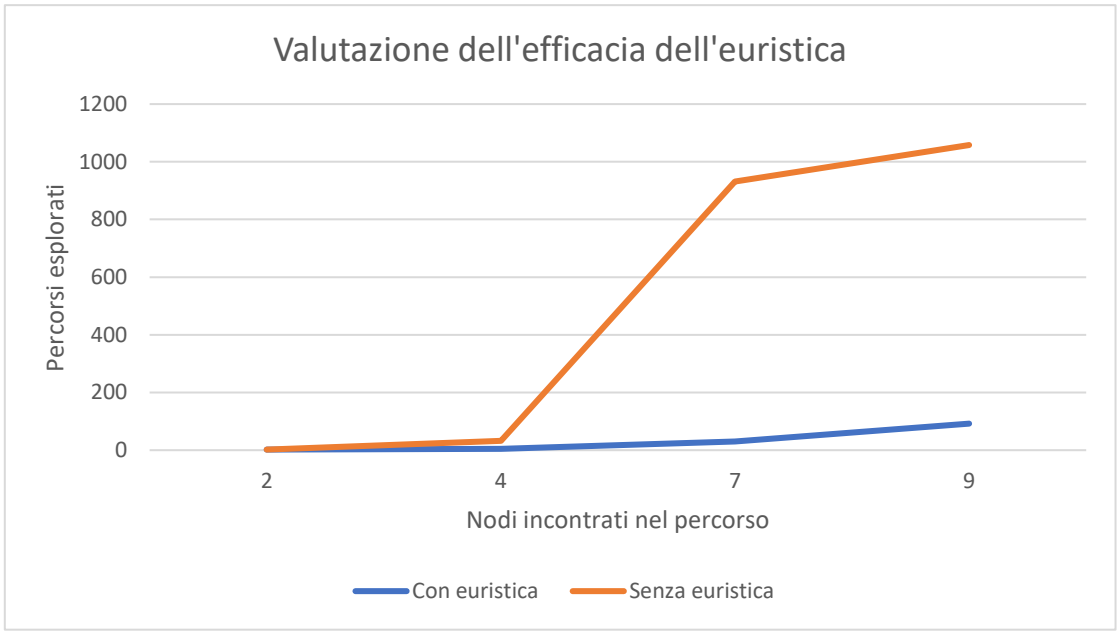
Come è possibile notare, l'euristica introduce un leggero overhead (infatti si nota dalla prima riga dove nodo partenza e obiettivo sono immediatamente vicini). Tuttavia, aumentando sempre di più la lunghezza del percorso e quindi di conseguenza il numero di nodi incontrati, l'euristica comporta un notevole miglioramento dell'efficienza dal punto di vista della complessità temporale.

Infatti, l'efficienza avuta grazie all'euristica è misurabile anche e soprattutto dal numero di percorsi che permette di non esplorare, cioè massimizzare la differenza tra:

$$|\{p|cost(p,t) < c\}| - |\{p|cost(p,t) + h(p) < c\}|$$

dove **c** è il costo del percorso ottimale. L'obiettivo è quindi quello di ridurre al minimo il numero di percorsi da esplorare dopo aver introdotto l'euristica (cioè minimizzare l'addendo evidenziato in giallo).

Di seguito il grafico che mostra la differenza tra il numero dei percorsi esplorati nell'utilizzo dell'euristica e senza euristica:



Nel caso peggiore analizzato, cioè nel percorso più lungo (9 nodi), A* con l'utilizzo dell'euristica ha esplorato circa 100 percorsi, al contrario di quello senza euristica che ne ha esplorati circa 1050 (risparmiando l'esplorazione di ben 950 percorsi).

Conclusione

Il software ha inoltre un ampio margine di espansione, rendendo possibile un'eventuale integrazione futura con altre tecnologie come il GPS e le comunicazioni wireless per fornire una visualizzazione in tempo reale del traffico e aiutare gli automobilisti a prendere decisioni informate sul percorso da seguire. Ciò potrebbe anche permettere di offrire informazioni al conducente in anticipo sulla situazione traffico e incoraggiare loro a prendere un percorso alternativo per evitare code.

Il software potrebbe anche essere espanso in diversi altri modi per migliorare ulteriormente la gestione del traffico. Ad esempio, può essere integrato con sistemi di navigazione per veicoli, utilizzare tecnologie di sensori, sincronizzato con altri sistemi di traffico, e altro ancora.

Tuttavia, è importante che l'integrazione con questi sistemi sia fatta con attenzione per garantire che non causi problemi di sicurezza e privacy.

Riferimenti Bibliografici

[1] https://artint.info/2e/html/ArtInt2e.Ch8.S5.SS2.html

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

[3] https://www.kaggle.com/datasets/vietexob/pgh-traffic-prediction?select=pgh_train.csv

[4] https://it.wikipedia.org/wiki/Scarto_quadratico_medio

[5] https://artint.info/2e/html/ArtInt2e.Ch3.S6.SS1.html

[6] https://artint.info/2e/html/ArtInt2e.Ch4.S1.SS3.html

[7] https://artint.info/2e/html/ArtInt2e.Ch10.S1.SS4.html

[8] https://artint.info/2e/html/ArtInt2e.Ch7.S6.html#:~:text=representation%20is%20a-,regression%20tree,-%2C%20which%20is%20a

[9] https://artint.info/2e/html/ArtInt2e.Ch7.S7.html#:~:text=k,%2Dnearest%20neighbors

[10] https://it.wikipedia.org/wiki/Prolog

[11] https://artint.info/2e/html/ArtInt2e.Ch7.S4.SS3.html

[12] https://artint.info/2e/html/ArtInt2e.Ch2.S4.SS2.html