UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Laurea Triennale in Ingegneria Informatica

Analisi ed evoluzione di un software per il routing vincolato e dinamico su dati di mobilità in una Smart City

Relatore:
Paolo Nesi

Candidato:
Andrea Delli

Correlatori:
Gianni Pantaleo
Stefano Bilotta

ANNO ACCADEMICO 2022/2023

Sommario

In una Smart City l'utilizzo di sensori per il monitoraggio ambientale porta un vantaggio significativo per il supporto decisionale. Una delle sfide più importanti all'interno di un grande centro urbano è rendere efficiente la mobilità. Il presente lavoro propone lo sviluppo di un software open source per la navigazione intelligente all'interno di una smart city in base agli obiettivi prefissati dall'utente e al mezzo di trasporto in uso.

Il sistema di routing implementato si pone l'obiettivo di fornire soluzioni di mobilità personalizzate, considerati diversi aspetti cruciali nella mobilità urbana.
L'utente ha la possibilità di scegliere l'obiettivo della navigazione col fine di trovare il percorso più veloce, il più breve, o il meno inquinato; il sistema consente inoltre di specificare dei vincoli personalizzati, come strade bloccate o zone da evitare, che rappresentano eventi come incidenti o manifestazioni. È infine disponibile l'approccio multimodale alla navigazione, in cui possono essere scelti diversi modi per raggiungere la destinazione: con automobili, biciclette, sedie a rotelle, e a piedi.

Queste caratteristiche permettono di fornire percorsi ottimizzati che tengono conto delle specifiche esigenze di ciascun utente, promuovendo una mobilità più efficiente e sostenibile all'interno della città.

Il software si integra nel contesto dell'applicativo di Snap4City, completamente open source e adattabile ad ogni esigenza, e per la navigazione utilizza ed estende strumenti open source come GraphHopper e OpenStreetMap. Nonostante sia facilmente adattabile per altri contesti urbani, è stato scelto come caso d'uso la città di Firenze, nella quale è diffusa un'ampia rete di sensori del traffico.

Indice

Capitolo 1

Introduzione

All'interno di una Smart City, la navigazione ha un ruolo fondamentale per garantire l'efficienza dei trasporti e la riduzione dei consumi. Un software per il routing deve garantire di poter trovare il percorso ottimo per muoversi all'interno della città, sfruttando dei dati in tempo reale forniti da dei sensori. Il programma implementato estende un progetto già esistente [1] che fornisce un sistema di navigazione in cui è possibile specificare delle aree bloccate (vincoli); questo progetto aggiunge al software l'utilizzo dei dati relativi al traffico urbano, forniti da vari sensori distribuiti per la città.

L'utilizzo dei dati del traffico permette di fornire una stima più accurata del percorso migliore. In caso ci fossero dei rallentamenti su determinate strade, i percorsi ottimi potrebbero variare: se l'obiettivo dell'utente è percorrere il tragitto nel minor tempo possibile, avere una stima del traffico lungo il percorso permette di evitare le zone più affollate della città in favore di percorsi meno trafficati, nell'eventualità che il tempo di percorrenza sia inferiore.

I dati del traffico possono basarsi sia su uno storico dei dati, in caso l'utente stia pianificando un itinerario da percorrere in futuro, sia su una stima del traffico in tempo reale, in caso la navigazione avvenga subito dopo il calcolo del percorso ottimo. In seguito (sezione 1.5.2) sono stati riportati i calcoli effettuati per stimare il traffico in tutta l'area urbana presa in considerazione.

1.1 Routing

I trasporti costituiscono una parte significativa dell'economia della maggior parte dei paesi sviluppati. L'instradamento dei veicoli è sempre stato oggetto di studio, e nella storia sono state presentate varie soluzioni a questo problema [2] che prende il nome di **Traveling Salesman Problem** (TSP), o problema del commesso viaggiatore [3].

Il TSP è uno dei casi di studio più comuni dell'informatica teorica: dato un insieme di città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una e una sola volta e per ritornare infine alla città di partenza.

La generalizzazione del TSP è il **Vehicle Routing Problem** (VRP) [4]. L'obiettivo del VRP è quello di trovare percorsi ottimali per più veicoli che visitano una serie di località; in genere per "percorsi ottimali" si intende l'insieme di percorsi con la distanza totale minima, tuttavia, senza vincoli aggiuntivi, la soluzione ottimale è assegnare un solo veicolo per assegnare tutte le località e trovare il percorso più breve per quel veicolo [4], come avviene nel TSP.
Un altro modo per definire i percorsi ottimali è ridurre al minimo la lunghezza del singolo percorso più lungo tra tutti i veicoli; determinare la soluzione ottimale per VRP è NP-hard, ossia la dimensione dei problemi che possono essere risolti in modo ottimale tramite la programmazione o l'ottimizzazione combinatoria è molto limitata.

Per poter effettuare la navigazione in tempo reale possono essere utilizzati diversi sensori comunemente presenti sui moderni dispositivi: GPS, odometri, accelerometri o giroscopi. I dati provenienti da questi sensori vengono combinati per ottenere una stima della posizione dell'utente. È inoltre fondamentale conoscere l'esatta conformazione della rete stradale: assumendo che il veicolo possa trovarsi solamente su una strada, le possibili posizioni in cui un utente si può trovare si riducono drasticamente, e la stima tramite sensori risulta essere molto più accurata.

1.2 Grafo

Come definito in "Introduzione agli algoritmi e strutture dati" [5], un **grafo** G è una coppia (V, E) dove V è un insieme finito ed E è una relazione binaria in V. L'insieme V è detto **insieme dei vertici** di G, e i suoi elementi sono detti **vertici**.

L'insieme E è detto **insieme degli archi** di G, e i suoi elementi sono detti **archi**. In Figura 1.1 è riportato un esempio di grafo con 4 vertici, rappresentati con dei cerchi, e 4 archi, rappresentati da linee.

Un grafo si dice **orientato** quando l'insieme degli archi E è composto da coppie di vertici **ordinate**. Un arco è un insieme $\{u, v\}$, dove $u, v \in V$ e $u \neq v$. Se l'arco è orientato, viene rappresentato con una freccia (come in Figura 1.2), la quale evidenza che l'arco $\{u, v\}$ può essere percorso partendo dal vertice u fino al vertice v, ma non viceversa.

In un grafo **non orientato** invece l'insieme degli archi E è composto da coppie di vertici **non ordinate**, pertanto l'arco può essere percorso in entrambe le direzioni. Gli archi non orientati sono rappresentati con una linea semplice, come mostrato in Figura 1.1.

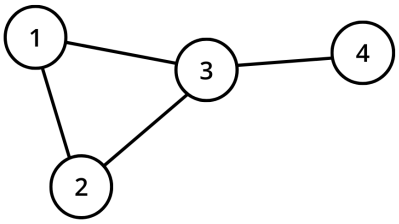

Figura 1.1: Grafo non orientato

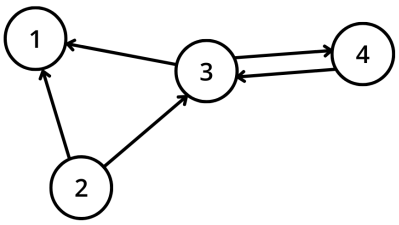

Figura 1.2: Grafo orientato

I grafi sono la struttura dati alla base di un software di navigazione, in quanto i vertici del grafo rappresentano punti d'interesse all'interno della mappa, e i vertici rappresentano le strade che possono essere percorse per andare da un vertice a un altro.

Si definisce un **cammino** di **lunghezza** k da un vertice u a un vertice u' in un grafo $G = (V, E)$, come una sequenza $\{v_0, v_1, v_2, ..., v_k\}$ di vertici tali che $u = v_0$ e $u' = v_k$, e $\{v_{i-1}, v_i\} \in E$ per $i = 1, 2, ..., k$. La lunghezza del cammino è il numero di archi nel cammino. Il cammino contiene i vertici $v_0, v_1, ..., v_k$ e gli archi $\{v_0, v_1\}, \{v_1, v_2\}, ..., \{v_{k-1}, v_k\}$. Se il cammino p unisce i nodi u e v, questo è indicato come $u \overset{p}{\mapsto} v$.

Un grafo si dice **pesato** se a ogni arco è associato un valore numerico, detto **peso**. La struttura dati del grafo pesato si adatta perfettamente alla modellazione di una mappa in quanto il peso potrebbe rappresentare diverse caratteristiche della strada,

come la lunghezza, la velocità, o l'intensità del traffico. La rappresentazione dei pesi all'interno di un grafo è mostrata nelle Figure 1.3 e 1.4.

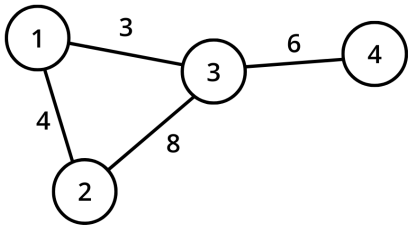

Figura 1.3: Grafo non orientato con pesi sugli archi

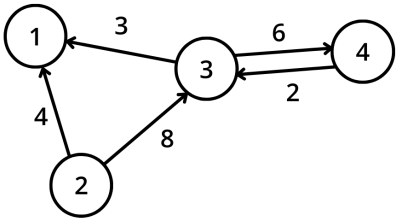

Figura 1.4: Grafo orientato con pesi sugli archi

Il **peso** $w(p)$ di un cammino $p = \{v_0, v_1, v_2, ..., v_k\}$ è la somma dei pesi degli archi che lo compongono [5]:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \tag{1.1}$$

Il **peso di un cammino minimo** $\delta(u, v)$ da u a v è definito nel seguente modo [5]:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \overset{p}{\mapsto} v\} & \text{se esiste un cammino da } u \text{ a } v \\ \infty & \text{negli altri casi} \end{cases} \tag{1.2}$$

Un **cammino minimo** dal vertice u al vertice v è definito come un cammino qualsiasi p con peso $w(p) = \delta(u, v)$.

1.3 What-if Analysis

La **What-if Analysis** è una tipologia di analisi che permette di simulare diversi scenari, col fine di stimarne le caratteristiche, ed è un tipo di analisi predittiva, ossia che fornisce previsioni sul futuro a partire da dei dati storici o real-time.
Serve quindi come supporto decisionale (DSS, Decision Support System) in quanto, grazie all'utilizzo di dati, è possibile simulare e prevedere cosa accadrebbe al sistema in esame a seguito di alcune modifiche. Un esempio di decisione da effettuare potrebbe essere l'apertura di un cantiere in una determinata area, oppure dove direzionare il traffico a seguito di un incidente stradale. Grazie all'utilizzo di dati, è possibile simulare diversi scenari in modo da fornire indicazioni accurate riguardo la scelta da effettuare. Per questo motivo la What-if Analysis è utilizzata in vari ambiti, come la business intelligence o la progettazione di sistemi critici.

Nel caso considerato in questa tesi, la decisione da effettuare riguarda il percorso da intraprendere durante la navigazione, in base all'ambiente circostante e ai vari vincoli del percorso. Lo strumento sviluppato può inoltre servire per capire come il traffico verrà ridiretto a seguito di una chiusura stradale.

1.4 Vincoli

Un vincolo è una qualsiasi limitazione alla libertà di movimento di un corpo [6]. Nel contesto del routing, un vincolo è una limitazione al transito di un determinato percorso stradale; è infatti possibile che certe strade o aree debbano essere chiuse a seguito di eventi stremi o calamità naturali (terremoti, alluvioni, attacchi terroristici, frane, ecc.), per manutenzione di strade ed edifici, per manifestazioni o per incidenti. In questi casi la rete stradale viene disconnessa e alterata, e i veicoli in transito dovranno seguire un percorso alternativo.

Nelle situazioni più estreme è cruciale prendere delle decisioni nel minor tempo possibile, per permettere alle autorità d'intervenire al più presto e limitare l'area coinvolta per redirigere altrove il traffico urbano. È necessario quindi disporre di strumenti che permettano rapidamente di fare delle stime per reindirizzare la mobilità nella corretta direzione; il software realizzato si pone esattamente quest'obiettivo.

1.5 Traffico

Il traffico urbano è una sfida che ogni grande città deve affrontare. Esso può rappresentare un grave disagio per i cittadini, e può influenzare negativamente la loro qualità di vita. Le lunghe code e i ritardi imprevisti possono generare stress, aumentare i tempi di percorrenza, ed esporre i cittadini ad inquinamento atmosferico, che se prolungato può portare a una significativa diminuzione dell'aspettativa di vita [7].

Considerare i dati sul traffico è diventato cruciale nella navigazione, in quanto considerare questi dati durante il routing aiuta a ridurre le emissioni e l'esposizione degli utenti all'inquinamento atmosferico urbano [8]. I dati sul traffico possono essere real-time, ottenuti tramite dei sensori sparsi per la città, oppure è possibile utilizzare uno storico dei dati in vari orari e periodi dell'anno, per stimare il traffico atteso in un determinato momento futuro.

1.5.1 Teoria dei flussi di traffico

La teoria del flusso del traffico consiste nello studiare le interazioni tra diversi tipi di viaggiatori (ad esempio pedoni, ciclisti, autisti, e i loro veicoli) e le infrastrutture (come autostrade, segnaletica, o piste ciclabili) con l'obiettivo di capire e sviluppare una rete di trasporti efficace e con la minor quantità di problemi di congestione.

Il modello di riferimento scelto per la ricostruzione del traffico è il **modello di Greenshield** [9], uno dei modelli cardine nel contesto della teoria dei flussi di traffico [10], tramite la quale è possibile calcolare il tempo di percorrenza di un segmento stradale data la ricostruzione del flusso di traffico [11].

Per ogni strada il modello mette in relazione la lunghezza d (in km), il tempo di percorrenza t (in h), la velocità v (in $\frac{km}{h}$) dei veicoli, il limite di velocità v_{max} (in $\frac{km}{h}$), la densità di traffico ρ (in $\frac{\#veicoli}{km}$), la densità massima di traffico ρ_{max} (in $\frac{\#veicoli}{km}$) e il flusso di veicoli f (in $\frac{\#veicoli}{h}$) [11].

$$v(\rho) = \begin{cases} 0 & \rho > \rho_{max} \\ v_F(1 - \frac{\rho}{\rho_{max}}) & \text{altrimenti} \end{cases} \tag{1.3}$$

$$f(\rho) = \rho \times v(\rho) \tag{1.4}$$

$$t = \frac{d}{v} \tag{1.5}$$

Nell'equazione 1.3 vengono considerate in particolare anche le casistiche in cui le misurazioni non sono state rilevate correttamente, ovvero in cui la densità di traffico rilevata è maggiore di quella massima per una determinata strada.

Le equazioni 1.3, 1.4, 1.5 sono state utilizzate nell'implementazione del programma, per calcolare il tempo totale di percorrenza di ogni tratto stradale coinvolto nel routing.

Nelle Figure 1.5, 1.6, 1.7 sono riportati dei grafici mostrano l'andamento del flusso $f(\rho)$ e della velocità $v(\rho)$ rispetto alla densità di traffico ρ, e la velocità $v(\rho)$ rispetto al flusso $f(\rho)$.

Osservando i grafici è possibile vedere che la velocità è massima (v_{max}) quando la densità di traffico è nulla; un valore di flusso nullo indica però che non ci sono veicoli nel tratto stradale considerato, quindi chiunque passi può procedere alla velocità massima. La velocità e il flusso inoltre si azzerano in un valore di densità

rappresentato nel grafico come ρ_{max}. Il valore massimo del flusso (f_{max}) si ottiene quando la densità è pari a $\rho_c = \frac{\rho_{max}}{2}$. Questo ci fornisce anche un'indicazione della velocità ottimale alla quale i veicoli dovrebbero percorrere la strada, indicata con v_{ottima}.

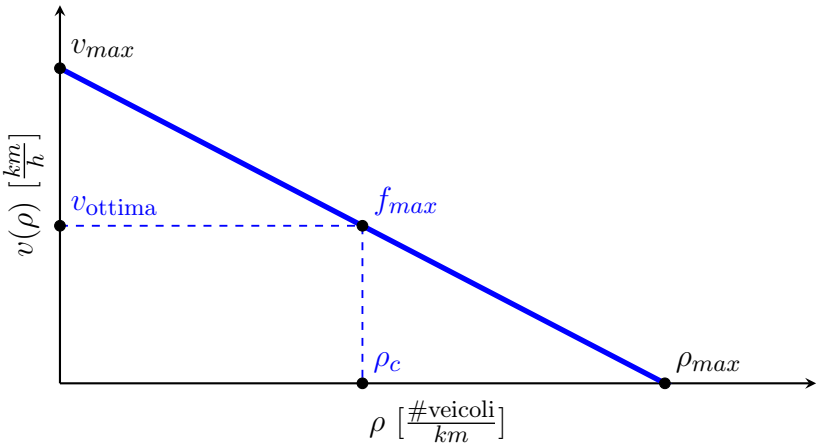

Figura 1.5: Velocità dei veicoli $v(\rho)$ rispetto alla densità di traffico ρ

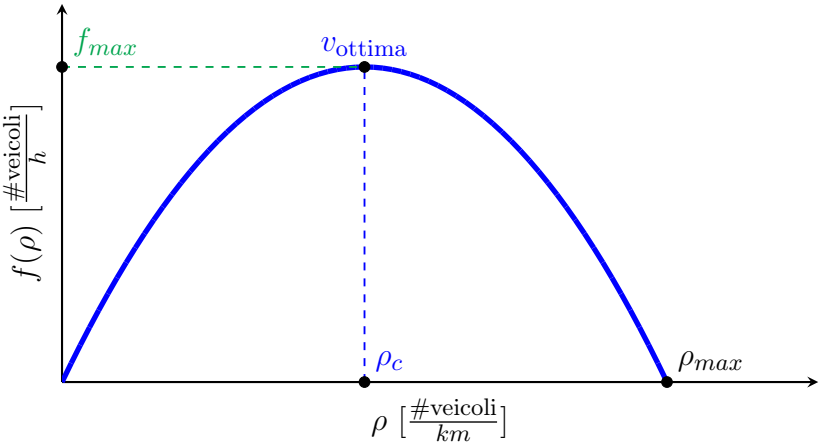

Figura 1.6: Flusso del traffico $f(\rho)$ rispetto alla densità di traffico ρ

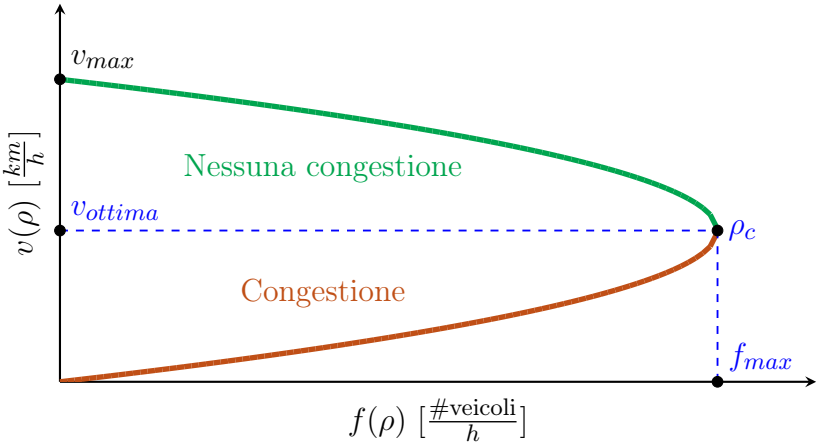

Figura 1.7: Flusso del traffico $f(\rho)$ rispetto alla velocità dei veicoli $v(\rho)$

1.5.2 Algoritmi per la stima del traffico

Una delle difficoltà principali della realizzazione di un software di navigazione è riuscire a estrapolare i dati del traffico in tutta la rete stradale avendo a disposizione i dati di un numero limitato di sensori sparsi per la città. Esistono numerosi approcci per la ricostruzione dei flussi di traffico [11], già integrati nel contesto di Snap4City [12], che forniscono al software proposto i dati sul traffico ricostruiti sui principali segmenti stradali di Firenze.

In Figura 1.8 è rappresentata la posizione dei sensori del traffico attualmente presenti nella città di Firenze: come si può notare, ci sono aree in cui i sensori sono distribuiti più densamente, come nel centro città e sui viali principali, e altre aree in cui i sensori sono più diradati, come nella periferia della città.

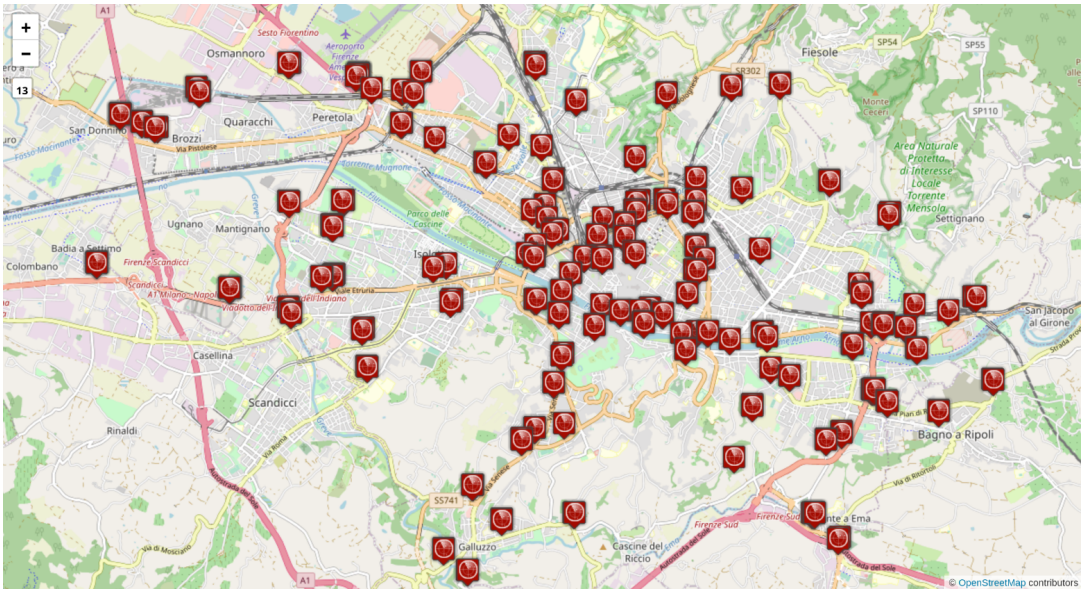

Figura 1.8: Posizione dei sensori del traffico nella città di Firenze

L'algoritmo utilizzato per calcolare l'andamento del traffico lungo un tratto stradale è il **Typical Time Trend** (TTT).

1.5.3 Typical Time Trend (TTT)

Il Typical Time Trend è un algoritmo che, date le misurazioni della densità del traffico in un certo intervallo di tempo in una determinata strada, calcola l'andamento tipico della densità del traffico nelle varie ore della giornata, per ogni giorno della settimana.

Nel caso specifico, sono state utilizzate le misurazioni effettuate ogni 10 minuti dai vari sensori sparsi per la città di Firenze (mostrati in Figura 1.8) dal 1 giugno 2023 al 14 giugno 2023, per avere un campione ampio (circa 1GB di dati, rappresentati con un file JSON generato ogni 10 minuti), condensate in un file JSON di circa 10MB.

Lo script utilizzato per il calcolo del TTT è stato scritto in Python e per ogni oggetto nei file JSON contenenti i dati di traffico, lavora nel seguente modo:

1. Legge il timestamp della misurazione
2. Calcola il giorno della settimana (valore da 0 a 6) e l'ora (da 0 a 23)
3. Inserisce in una matrice 7×24 di vettori la misurazione effettuata

Dopo aver effettuato questa elaborazione, viene calcolata la media di ogni vettore nelle 7×24 posizioni, che rappresenta il Typical Time Trend della densità di traffico del segmento stradale in preciso giorno della settimana a una determinata ora. Per ogni segmento stradale ci saranno quindi 7×24 valori di densità di traffico, oltre al valore che rappresenta la densità massima stradale (che dipende dalla conformazione della strada).

Rappresentazione grafica del TTT

In Figura 1.9 è rappresentata una mappa di calore (heatmap) che mostra visivamente l'andamento del TTT per un tratto stradale scelto come campione. L'asse verticale rappresenta i vari giorni della settimana (dal lunedì nella prima riga alla domenica nella settima riga), mentre quello orizzontale rappresenta l'orario (dalle 00:00 alle 01:00 nella prima colonna, ecc.).

Il colore verde indica che la strada non è soggetta a traffico, mentre il colore rosso rappresenta che la strada è più trafficata. Il valore massimo della scala di colori rappresenta la densità massima stradale.

Un altro grafico che rappresenta il TTT è il grafico a linee riportato in Figura 1.10, che mostra l'andamento del TTT rispetto all'orario. Ogni linea rappresenta l'andamento del TTT per uno specifico giorno della settimana, mentre la linea verde orizzontale rappresenta la densità massima stradale.

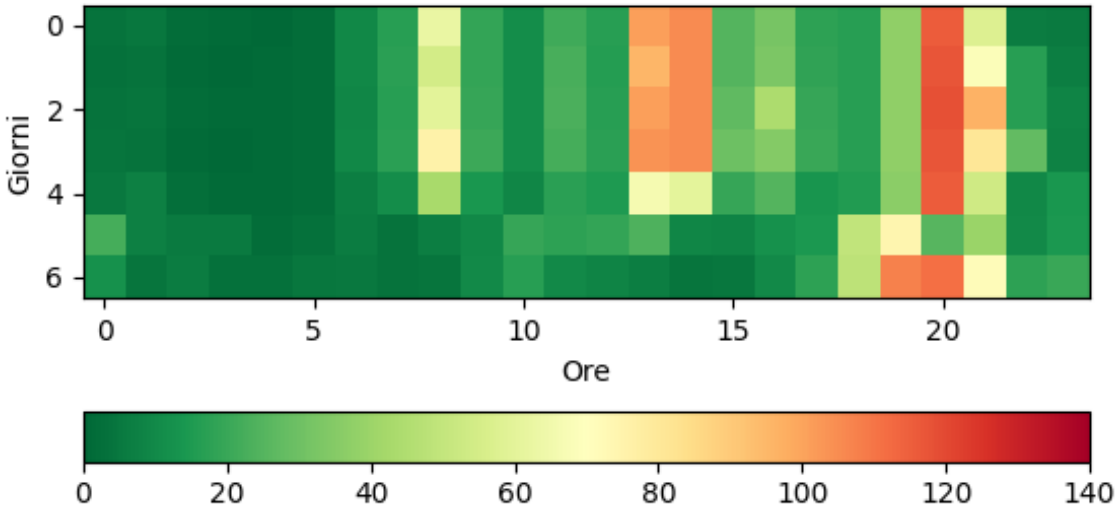

Figura 1.9: Heatmap che mostra il TTT della densità di traffico di un RoadElement, con i giorni sull'asse y e le ore sull'asse x

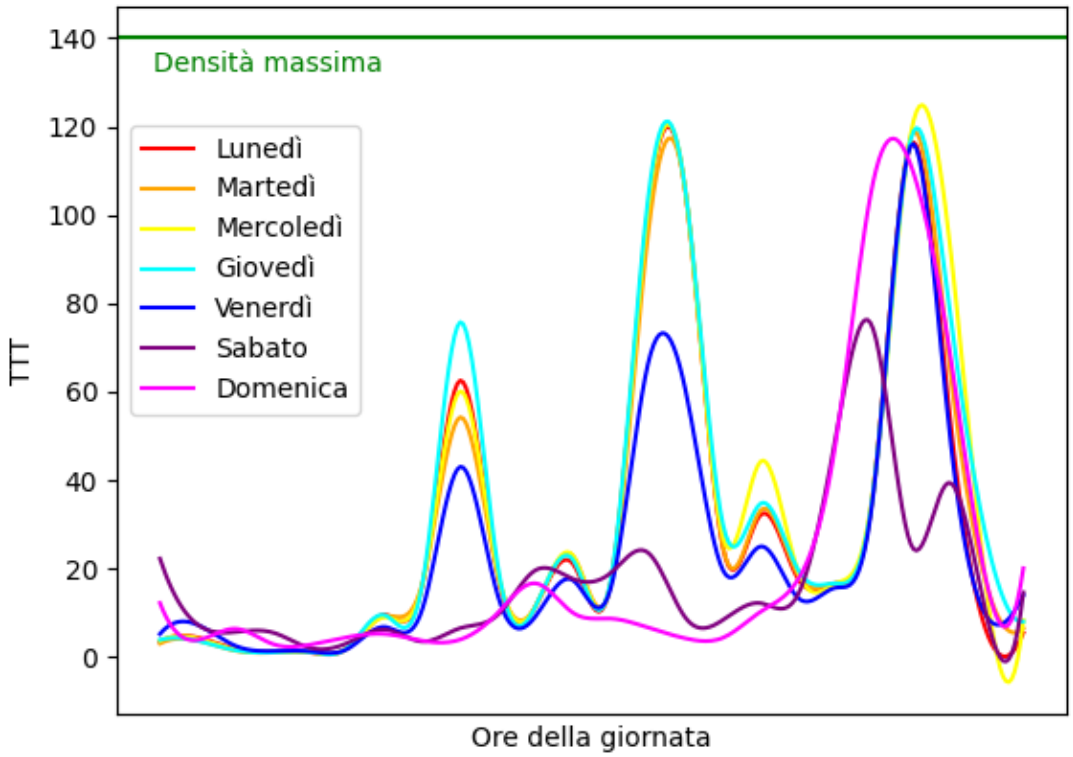

Figura 1.10: Andamento del TTT di un RoadElement nei vari giorni della settimana e densità massima stradale (in verde)

Come si può notare da entrambi i grafici, nel fine settimana il segmento stradale interessato è meno trafficato nelle ore di punta rispetto agli altri giorni della settimana, e questo mostra l'importanza di specificare il giorno e l'ora alla quale l'utente vuole iniziare la navigazione, al fine di fornire informazioni più attendibili sul traffico.

Applicando una colorazione simile su una mappa, si ottiene una rappresentazione del traffico molto esplicativa, come è possibile osservare da una dashboard di Snap4City, riportata in Figura 1.11.

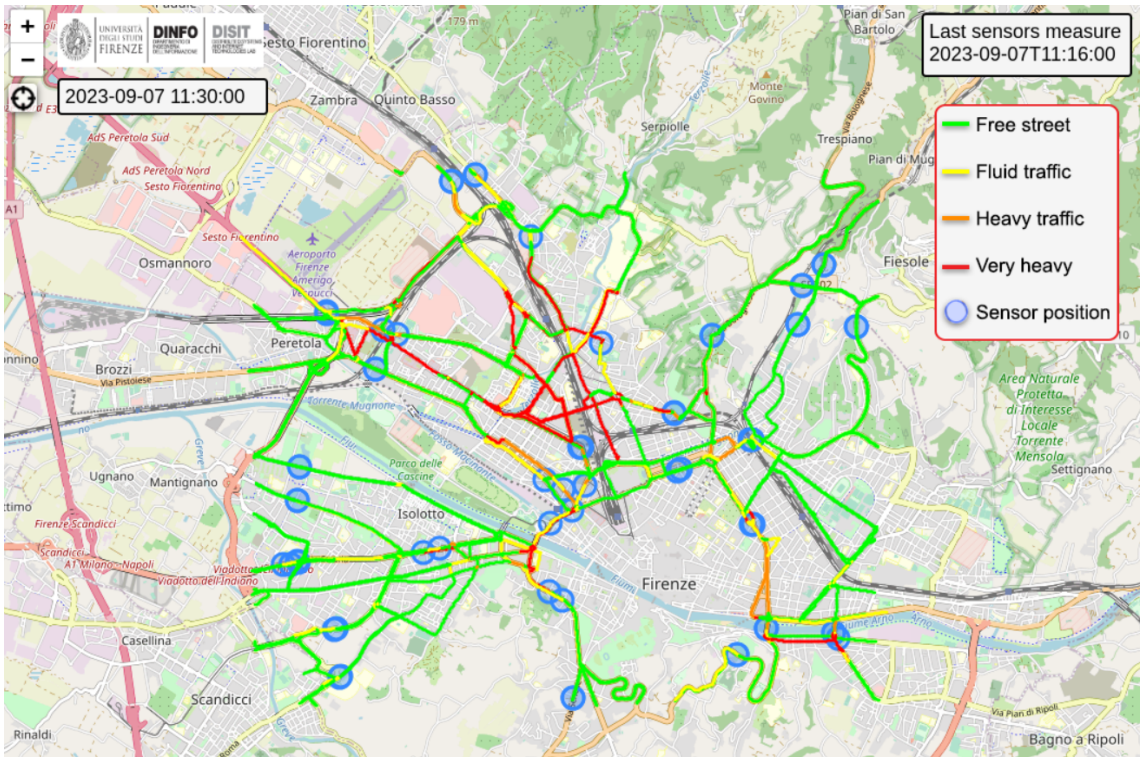

Figura 1.11: Ricostruzione del traffico in una dashboard di Snap4City

1.6 Algoritmi di routing

Individuare il cammino minimo su un grafo pesato, dato un nodo sorgente e un nodo di destinazione, è un noto problema nell'ingegneria dell'informazione ed esistono numerosi algoritmi per trovare una soluzione; esistono algoritmi che forniscono una soluzione esatta, ma richiedono un costo computazionale più elevato, e algoritmi euristici che tendono a fornire più percorsi come soluzione che non sono necessariamente il percorso ottimo, bensì una sua approssimazione. Le tipologie di algoritmo disponibili in GraphHopper e altri algoritmi per il calcolo del percorso ottimo sono stati analizzati nel lavoro precedente a questa tesi [1]. L'algoritmo scelto per il

routing con GraphHopper è l'**Algoritmo di Dijkstra bidirezionale** [13]. Questo algoritmo si basa sull'**algoritmo di Dijkstra**, mostrato in Figura 1.12, con la peculiarità che la ricerca bidirezionale esegue due ricerche simultanee: una in avanti dalla sorgente e una all'indietro dal target, fermandosi quando le due si incontrano a metà. Questa implementazione può essere utilizzata sia con un grafo diretto che con un grafo non orientato.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5       u = EXTRACT-MIN(Q)
6       S = S ∪ {u}
7       for each vertex v ∈ G.Adj[u]
8            RELAX(u, v, w)
```

```
RELAX(u, v, w)
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4  s.d = 0
```

Figura 1.12: Algoritmo di Dijkstra [5]
$G = (V, E)$: Grafo pesato, con pesi non negativi
s: vertice d'origine, w: vertice di destinazione
RELAX(u, v, w): verifica se, passando per u, è possibile migliorare il cammino minimo per v precedentemente trovato
INITIALIZE-SINGLE-SOURCE(G, s): inizializza le stime dei cammini minimi e i predecessori di ogni vertice del grafo

Esistono inoltre altri algoritmi che hanno l'obiettivo specifico di trovare percorsi migliori, considerando la qualità dell'aria e il traffico lungo il tragitto, ad esempio il *Least Exposure to Air Pollution* (LEAP, [14]) o il *Clean Air Routing* (CAR, [15]).

Capitolo 2

Tecnologie utilizzate

Il presente capitolo è dedicato all'illustrazione delle tecnologie chiave impiegate nello sviluppo ed evoluzione del software proposto in questa tesi. Una combinazione sinergica delle piattaforme e librerie mostrate è essenziale per garantire un'implementazione efficace del sistema.

Un'altra peculiarità delle tecnologie scelte è che sono interamente open source, e ciò permette di utilizzare delle funzionalità ideate e realizzate da una vasta comunità di sviluppatori, che testano e migliorano continuamente il codice con l'obiettivo di aumentarne le performance e correggere eventuali bug.

2.1 OpenStreetMap

OpenStreetMap (OSM, [16]) è un progetto che mira a creare una mappa del mondo completamente libera e accessibile. I dati presenti in OpenStreetMap sono disponibili gratuitamente e possono essere utilizzati e modificati da chiunque, in conformità con la licenza Open Database License (ODbL).

OpenStreetMap fornisce una mappa che contiene tutte le informazioni necessarie per sviluppare un software di navigazione: strade, confini amministrativi, punti d'interesse, piste ciclabili, sentieri escursionistici, e molto altro.

Esistono diversi elementi fondamentali che fanno parte di una mappa di OpenStreetMap: nodi, percorsi, aree, relazioni.

2.1.1 Node

Un **nodo** (Node, [17]) è uno degli elementi principali del modello dati di OpenStreetMap. Consiste in un singolo punto nello spazio definito dalla sua *latitudine*, *longitudine* e *id*. Può essere inclusa anche una terza dimensione spaziale, che rappresenta l'*altitudine*, e può inoltre essere definito come parte di un particolare *layer* (o *level*), per distinguere elementi che passano sopra o sotto un altro elemento.

I nodi possono essere usati per indicare elementi puntuali staccati ma sono più spesso usati per definire la forma di un profilo o di una way e possono avere delle *etichette* che rappresentano informazioni aggiuntive per la descrizione del nodo.

2.1.2 Way

Un **percorso** (Way, [18]) è un insieme ordinato da un minimo di 2 a un massimo di 2000 nodi che descrivono una caratteristica in modo lineare tipo una strada, un sentiero, una linea ferroviaria, un fiume, delle aree o il perimetro di un edificio.

Un percorso è caratterizzato da proprietà uniformi; per esempio, la priorità (autostrada, strada statale, ...), il tipo di superficie, la velocità, ecc. I percorsi possono essere divisi in sezioni più piccole se sono presenti proprietà differenti; per esempio, se una strada ha una sezione a senso unico, quella sezione sarà un percorso differente da quella a doppio senso di marcia, anche se hanno in comune lo stesso nome.

Un **percorso chiuso** è un percorso in cui il primo e l'ultimo punto coincidono racchiudendo un'area. Le aree non sono dei dati primitivi di OSM ma semplicemente percorsi chiusi che sono etichettati a rappresentare un'area.

Un percorso è rappresentato da un *id* (intero ≥ 1), una *lista di nodi* che compongono il percorso, ed eventuali *etichette*.

2.1.3 Area

Un'**area** (Area, [19]) può essere definita come uno spazio racchiuso mediante un percorso chiuso usando le etichette appropriate oppure con una relazione multipoligono che, in pratica, crea un'area usando uno o più percorsi.

2.1.4 Relation

Una **relazione** (Relation, [20]) è uno degli elementi base di OpenStreetMap, e consiste di uno o più *Tag* (etichette) e anche un elenco ordinato di uno o più Nodi

e/o Percorsi in qualità di *membri della relazione*. Una relazione viene utilizzata per definire le relazioni logiche o geografiche tra vari elementi. Un membro di una relazione può avere un ruolo che descrive una parte che svolge una particolare funzione all'interno della relazione.

2.1.5 Esportazione della mappa

Per poter utilizzare una mappa di OpenStreetMap è possibile ottenere un file con estensione `.pbf` che rappresenta la mappa di una determinata area. È possibile scaricare sia la mappa globale che aree più limitate, come un continente, uno stato, una regione o una città.

Per poter acquisire una mappa è possibile utilizzare strumenti come **Overpass API** [21] o **BBBike** [22], in cui è possibile inoltre specificare un *Bounding Box*, ossia un rettangolo (delimitato da due coppie di latitudine e longitudine) che rappresenta l'area da esportare.

2.2 GraphHopper

GraphHopper è un software open source [23] per il routing e l'ottimizzazione che offre soluzioni per il calcolo dei percorsi e la pianificazione d'itinerari. Per questo motivo GraphHopper è stato scelto come motore di routing principale all'interno del sistema realizzato.

Grazie alla sua capacità di elaborare grafi di rete e calcolare percorsi ottimizzati in tempo reale, questa libreria si è dimostrata fondamentale per gestire le diverse modalità di trasporto in modo integrato. La flessibilità di GraphHopper nel supportare algoritmi di routing avanzati è stata cruciale per la realizzazione di un sistema capace di adattarsi in tempo reale alle condizioni variabili del contesto urbano, come la densità di traffico o le aree bloccate.

2.2.1 Navigazione multimodale

I sistemi di navigazione multimodale sono progettati per aiutare le persone a pianificare i loro spostamenti in modo più flessibile, tenendo conto di fattori come il traffico, le condizioni meteorologiche, i mezzi di trasporto, i tempi di attesa e le preferenze personali. La navigazione multimodale è particolarmente rilevante nelle aree urbane densamente popolate, dove esistono varie opzioni di trasporto e spesso

è necessario cambiare modalità durante il viaggio per raggiungere la destinazione desiderata. La navigazione multimodale può contribuire a ridurre il traffico stradale, migliorare l'accessibilità e promuovere la sostenibilità ambientale, incoraggiando l'uso del trasporto pubblico, della bicicletta o della camminata quando è possibile e conveniente.

Il software implementato permette all'utente di scegliere varie modalità con la quale poter percorrere il proprio percorso. La navigazione può inoltre avvenire sia con un singolo mezzo di trasporto che alternando diverse modalità.

GraphHopper [24] dispone di alcune modalità di trasporto predefinite:

- `foot` (camminata)
- `hike` (escursione)
- `wheelchair` (sedia a rotelle)
- `bike` (bici)
- `racingbike` (bici da corsa)
- `mtb` (mountain bike)
- `car` (macchina)
- `motorcycle` (motociclo)

La selezione del veicolo in uso da parte dell'utente permette al software di modificare la priorità di diversi tipi di strade durante il calcolo del percorso. Ad esempio, quando si percorrono lunghe distanze con un'auto, in genere si desidera utilizzare l'autostrada per ridurre al minimo la durata del viaggio. Se invece si utilizza una bicicletta, è preferibile evitare l'autostrada e piuttosto preferire il percorso più breve, favorendo l'utilizzo di piste ciclabili.

2.2.2 Navigazione multiobiettivo

Nel software realizzato, l'utente ha la possibilità di selezionare l'obiettivo della propria navigazione, scegliendo di voler cercare il percorso "più veloce", "più breve", o una delle altre modalità esistenti. Ciò che varia tra questi obiettivi durante la ricerca del percorso migliore è la modalità con la quale i pesi degli archi del grafo stradale vengono calcolati. Di conseguenza, anche il cammino ottimo per arrivare dalla sorgente alla destinazione potrebbe variare.

GraphHopper ha i seguenti obiettivi di navigazione predefiniti:

- `fastest` (percorso più veloce)
- `shortest` (percorso più breve)
- `short_fastest` (compromesso tra percorsi brevi e veloci)
- `curvature` (percorsi con molte curve, per piacevoli giri in moto)
- `custom` (profili personalizzati)

Uno degli obiettivi del programma è proprio quello di estendere GraphHopper per introdurre la modalità di navigazione `fastest_with_traffic`, che permette di trovare il percorso più veloce per la navigazione, considerando i dati sul traffico della rete stradale. Ciò è possibile sfruttando i sensori del traffico sparsi per la città.

In GraphHopper, la scelta del tipo di veicolo utilizzato e dell'obiettivo dell'instradamento prende il nome di **profilo** (Profile).

2.3 Snap4City

Le Smart City devono creare un ambiente flessibile per gestire l'**Internet delle Cose** (*Internet of Things*, IoT). Snap4City [12] fornisce un metodo e una soluzione flessibili per creare rapidamente una vasta gamma di applicazioni per città intelligenti, sfruttando dati eterogenei e abilitando servizi per gli stakeholder attraverso tecnologie IoT/IoE e big data. Le applicazioni Snap4City supportano diversi paradigmi come il data driven, l'elaborazione in streaming e batch, mentre i modelli IoT stanno rapidamente evolvendo e diventando i più prominenti.

Snap4City è una piattaforma completa per gestire, aggregare, visualizzare e analizzare big data e IoT; queste operazioni possono essere personalizzate per soddisfare qualsiasi esigenza aziendale.

Snap4City viene utilizzato in una vasta gamma di applicazioni per città intelligenti e Industria 4.0, sfruttando dati eterogenei e abilitando nuovi servizi, chiamati **Scenari**.

Snap4City permette la creazione di **Dashboard**, che sono strumenti usati da molte aziende per tenere traccia dei dati, analizzarli e visualizzarli in maniera efficiente.

Le **Micro Applications** sono applicazioni web indipendenti simili a delle dashboard, poiché visualizzano informazioni, ma nonostante siano molto più semplici e concentrate sui dati geospaziali, possono essere incorporate nelle dashboard. Sono sviluppate come soluzioni HTML5 in JavaScript e sono progettate per essere reattive, quindi possono essere utilizzate su tutte le dimensioni dello schermo.

Le Dashboard e le Micro Applications visualizzano dati che possono essere caricati da:

- Dati forniti da Snap4City, chiamati **Knowledge Base**
- Applicazioni IoT personalizzate volte a caricare e trasformare dati provenienti dai dispositivi IoT del cliente e da varie altre fonti
- Dati caricati dall'utente provenienti da varie altre fonti, che è possibile inserire grazie all'utilizzo dei vari **connettori** forniti da Snap4City
- Altre tipologie di strumenti d'ingestione di dati, ad esempio *MyKPI, table loading, open data*, ecc.

Snap4City è fornito con un set di dati geografici provenienti da molte fonti, pronto per essere incluso nelle dashboard personalizzate. Snap4City è inoltre associato a diverse organizzazioni locali che hanno aperto i loro servizi e dati agli utenti della piattaforma.

Il software implementato per questa tesi è stato integrato in Snap4City e può essere utilizzato e visualizzato tramite una Dashboard.

2.3.1 Dashboard di Snap4City

Una **dashboard** può visualizzare, interagire e agire con/su qualsiasi tipo di dati al fine di semplificarne la comprensione e la gestione.

Le principali caratteristiche delle dashboard sono le seguenti:

- Possono essere realizzate su misura per Smart City e Industrie 4.0
- Utilizzano di mappe, dati storici e in tempo reale, KPI
- Possono contenere elementi *Event Driven*, per poter mostrare i dati e agire di conseguenza
- Possono essere create semplicemente anche da persone senza conoscenze di programmazione, grazie alla procedura guidata

- Sono adattabili a qualsiasi dimensione dello schermo
- Forniscono un controllo degli accessi integrato, flessibile per ogni esigenza

In Figura 2.1 è mostrato il sito web di Snap4City [12] in cui vengono mostrate le dashboard con la quale l'utente può interagire.

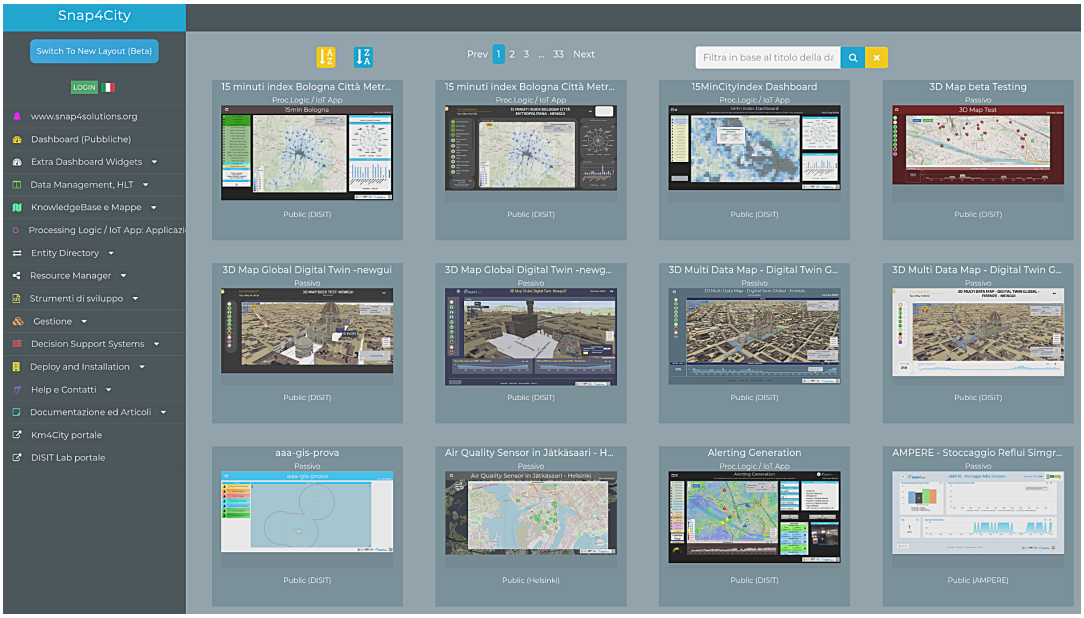

Figura 2.1: Elenco di dashboard presenti in Snap4City

2.3.2 Servlet

Una **Servlet** è un programma Java che ha il compito di estendere le funzionalità offerte da un server per realizzare pagine web dinamiche.

Per ospitare la servlet all'interno di un web server è stato utilizzato *Apache Tomcat* [25], mentre per la realizzazione è stato utilizzato *Jersey* [26].

Tomcat è un web server open source sviluppato dalla Apache Software Foundation, che fornisce una piattaforma software per l'esecuzione di applicazioni web sviluppate in Java, dette **servlet**.

Jersey è un framework open source sviluppato dalla Eclipse Foundation, che serve a sviluppare servizi web RESTful in Java che implementa la specifica JAX-RS, ossia permette di definire endpoint, parametri, e URL tramite l'utilizzo di *annotazioni*. Grazie a questo framework è possibile creare un'API alla quale la dashboard può effettuare delle richieste per generare i percorsi di navigazione più efficienti.

2.4 Stato dell'arte

La presente tesi è un'evoluzione di un software di What-If Analysis [1] già integrato in Snap4City. L'attuale sistema di navigazione utilizza la versione di GraphHopper 0.13 e permette di creare degli scenari multipli in cui possono essere bloccate diverse aree per un intervallo di tempo a scelta, il cui unico obiettivo è trovare il percorso più veloce nonostante questi vincoli.

Il primo passo per evolvere il programma è stato aggiornare la libreria all'ultima versione (GraphHopper 7.0) e adattare il codice del progetto per mantenerne il funzionamento, nonostante le numerose modifiche alla libreria.

In seguito, è stata implementata la possibilità per l'utente di scegliere l'obiettivo della navigazione (ad esempio `shortest`), in quanto in precedenza l'unico obiettivo possibile in questo sistema di navigazione era la ricerca del percorso più veloce (modalità `fastest`), come spiegato in dettaglio nella sezione 4.1. È stata introdotta la possibilità di utilizzare le altre modalità predefinite di GraphHopper, mantenendo comunque la possibilità di bloccare alcune aree nella navigazione.
Infine, è stata integrata la modalità `fastest_with_traffic`, in cui vengono considerati anche i dati storici (come spiegato in Sezione 1.5.2) per capire quale percorso fosse il più veloce nonostante il traffico urbano.

In Figura 2.6 è presente un esempio del funzionamento del programma: una volta scelto il punto di partenza e la destinazione (Figura 2.2), è possibile trovare il percorso più veloce che li unisce (Figura 2.3). Se sono impostate delle aree bloccate (Figura 2.4), queste devono essere considerate durante la ricerca del percorso. Se sono inoltre presenti dei dati sul traffico (Figura 2.5), alcune strade potrebbero essere penalizzate durante la scelta del percorso, a favore di altre strade meno congestionate.

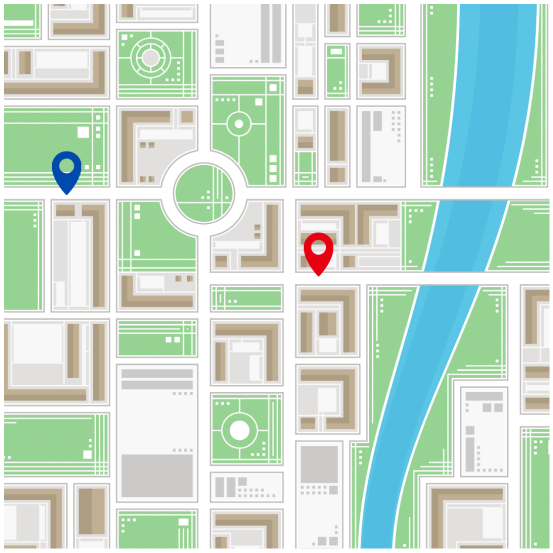

Figura 2.2: Punto di partenza (blu) e di arrivo (rosso)

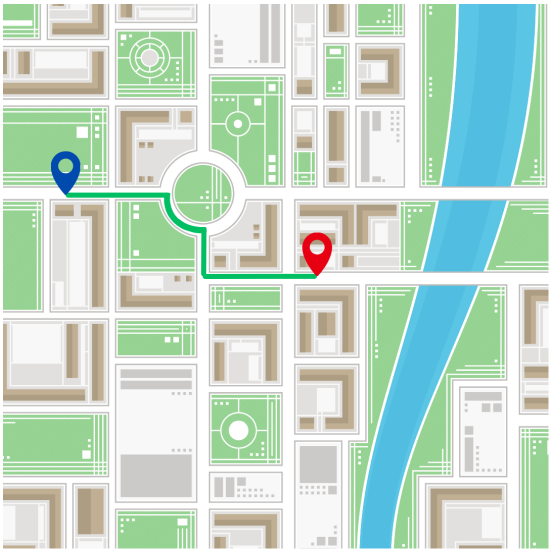

Figura 2.3: Percorso più veloce (verde) per arrivare da sorgente a destinazione

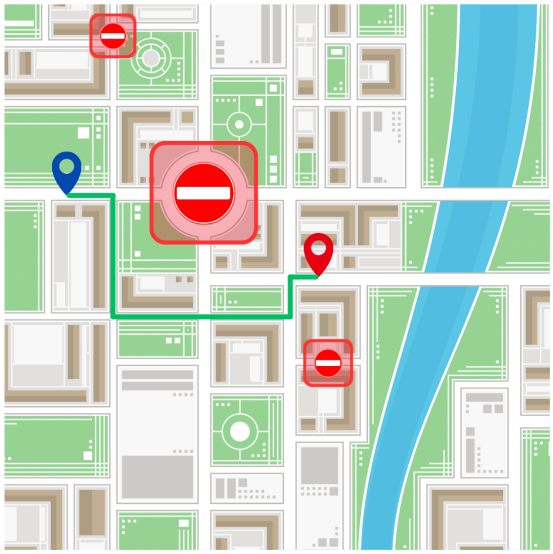

Figura 2.4: Percorso migliore con aree bloccate (zone rosse)

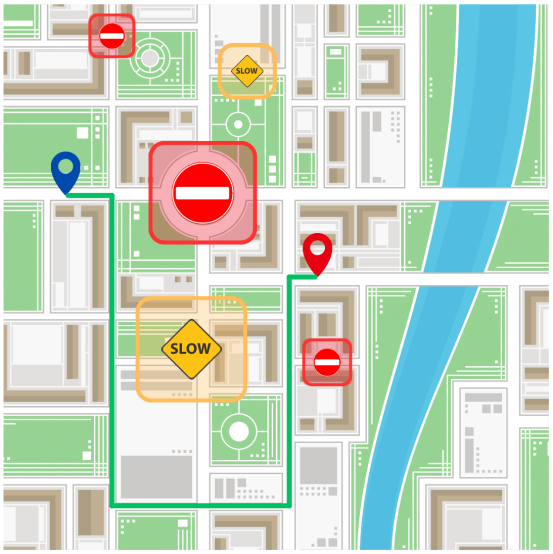

Figura 2.5: Percorso migliore con aree bloccate e traffico (zone gialle)

Figura 2.6: Evoluzione del sistema di navigazione proposto

2.4.1 Google Maps

Uno dei più famosi software per la navigazione è sicuramente **Google Maps** [27], un servizio internet geografico sviluppato da Google che consente la ricerca e la visualizzazione di carte geografiche. Il servizio è fruibile sia tramite il sito web che da app per dispositivi mobili, permette di cercare servizi in particolari luoghi, tra cui ristoranti, monumenti e negozi e consente di visualizzare un possibile percorso stradale tra due punti, oltre a foto satellitari di molte zone con diversi gradi di dettaglio (per le zone che sono state coperte dal servizio si riescono a distinguere in molti casi le case, i giardini, le strade e così via).

Uno dei principali svantaggi di Google Maps è che è closed source, non è quindi noto con esattezza come vengono raccolti e analizzati i dati, e per poter utilizzare le *Google Maps API* per progetti professionali è necessario pagare una cifra per ogni richiesta.

Similarmente al software proposto in questa tesi, anche Google Maps considera i blocchi stradali attualmente presenti sul percorso. Non consente però di effettuare una What-If analysis, bensì soltanto la navigazione real-time evitando le aree bloccate. Google Maps permette la navigazione multimodale, in quanto consente di scegliere il mezzo di trasporto desiderato, ma non permette la navigazione multiobiettivo (viene cercato sempre il percorso più veloce). Inoltre, anche Google Maps considera dei dati sul traffico per il calcolo del percorso più rapido, come mostrato in Figura 2.7. I dati di traffico utilizzati non vengono però da dei sensori sul territorio, bensì dalla velocità degli utenti che percorrono determinate strade utilizzando l'app (Figura 2.8). Per la pianificazione dei viaggi invece viene utilizzato uno storico dei dati del traffico (Figura 2.9) in modo simile a quanto implementato in questo progetto.

La navigazione in Google Maps è poco personalizzabile, ad esempio è possibile scegliere se evitare traghetti, strade a pedaggio o autostrade, mentre utilizzando una libreria come GraphHopper si ha la piena libertà di personalizzazione.

Google Maps non fornisce le informazioni sulla qualità dell'aria, in modo tale da poter offrire percorsi ottimizzati per la salute dell'utente. Ha però le informazioni sul trasporto pubblico locale in cui è possibile vedere la posizione real-time dei mezzi pubblici.
Sia la qualità dell'aria che le informazioni sul trasporto pubblico sono dati che potrebbero essere integrati in una futura estensione del progetto realizzato.

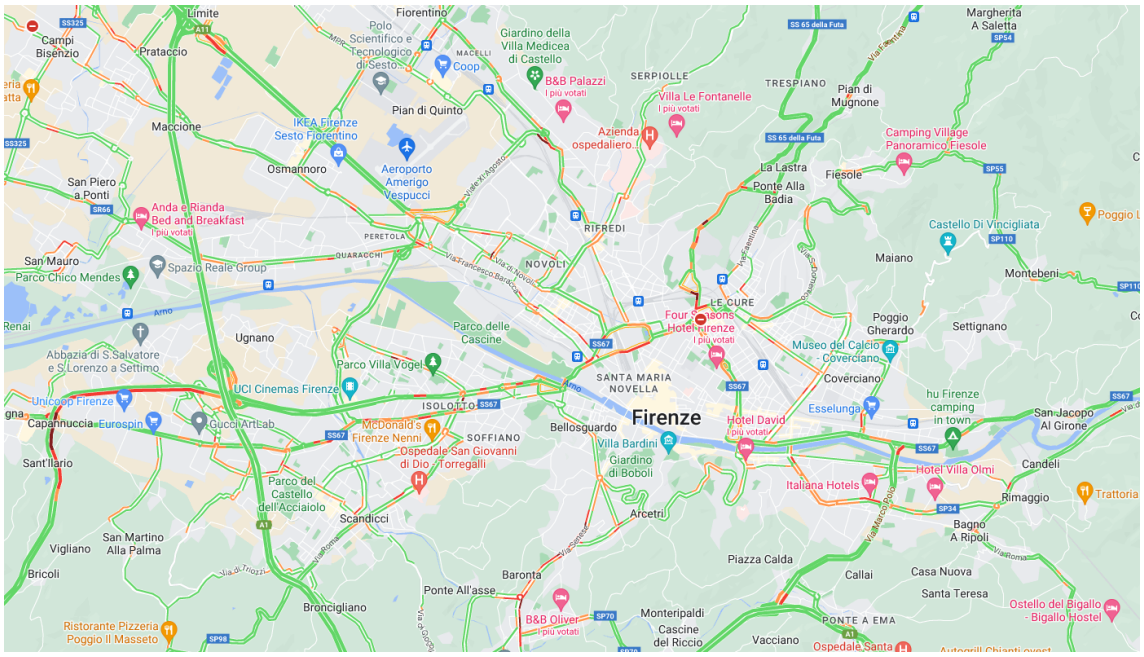

Figura 2.7: Esempio di mappa del traffico nella città di Firenze [28]

Figura 2.8: Legenda del traffico in tempo reale di Google Maps [28]

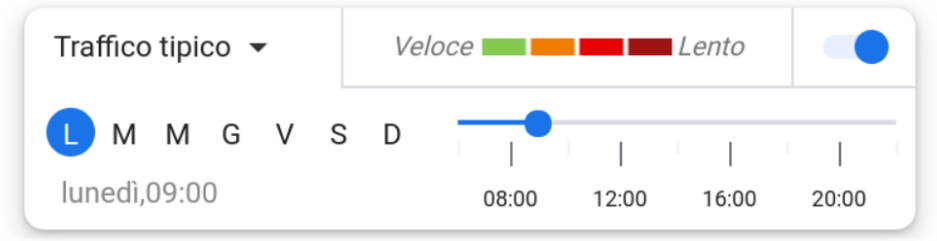

Figura 2.9: Legenda interattiva del traffico storico di Google Maps [28]

Capitolo 3

Progettazione

Nel seguente capitolo sono riportate le fasi dell'analisi effettuata per la realizzazione ed evoluzione del software.

Sono state utilizzate delle tecniche classiche della progettazione del software. Oltre all'**analisi dei requisiti**, è stata scelta l'**architettura di rete** di riferimento ed è stato tracciato un **sequence diagram** per mostrare come il sistema lavora. Per la realizzazione del programma è stato realizzato un diagramma delle classi da utilizzare nella fase di scrittura del codice.

3.1 Analisi dei requisiti

L'**analisi dei requisiti** è un'attività preliminare allo sviluppo e ha lo scopo di definire le funzionalità che il prodotto deve offrire, ossia i *requisiti* che devono essere soddisfatti dal software sviluppato.

I requisiti del software estendono i requisiti già esistenti [1] prima dell'evoluzione introdotta in questa tesi, e possono essere classificati in requisiti generali, funzionali e non funzionali.

Requisiti generali

- L'applicazione deve interfacciarsi e interconnettersi con il sistema **Snap4City** [12], ed essere integrato con il sistema di dashboard del Dashboard Builder

- Il software per il calcolo del percorso utilizzato dev'essere non proprietario e open source, mantenuto da una comunità di sviluppatori che ne garantiscano l'affidabilità nel tempo [24] [23]
- Il sistema dev'essere compatibile con i dati open source provenienti da OpenStreetMap [16]

Requisiti funzionali

- Il sistema deve consentire l'introduzione di uno o più **scenari**, ovvero particolari situazioni identificate dalla creazione di una o più barriere
- Il sistema deve consentire la creazione delle **barriere** attraverso uno strumento grafico che consenta di posizionare i vari tipi di barriere sulla mappa
- Il sistema deve consentire il salvataggio degli scenari assegnando loro un nome identificativo univoco, per garantire che due scenari non possano avere lo stesso nome
- Il sistema deve consentire la scelta di uno scenario su cui effettuare un routing dinamico, cioè basato sui vincoli del contesto
- Il sistema deve consentire che lo scenario sezionato venga mostrato sulla mappa
- Il sistema deve consentire la selezione della **sorgente e destinazione** desiderate tramite l'interfaccia grafica
- Il sistema deve consentire la scelta del **mezzo di trasporto** per il calcolo del routing
- Il sistema deve mostrare su mappa il percorso calcolato dal routing
- Il sistema deve mostrare anche delle **possibili alternative** per il tragitto, se disponibili
- Il sistema deve permettere il salvataggio di uno studio, ovvero un'analisi di percorso effettuata a partire da uno scenario precedentemente definito
- Il sistema deve consentire il salvataggio di scenari/studi di tipo pubblico o privato: un elemento di tipo privato sarà visibile solo all'utente che lo ha creato (fatta eccezione per un utente di tipo *RootAdmin* che ha un livello di visibilità globale), mentre un elemento di tipo pubblico può essere visto da chiunque, anche da un utente non loggato (*guest*)

- Il sistema deve consentire di poter specificare l'**obiettivo della navigazione** (ad esempio "*percorso più breve*")
- Il sistema deve consentire la scelta della data e ora di partenza
- Il sistema deve mostrare i **dati del traffico** sulla mappa

Requisiti non funzionali

- **Prestazioni**: l'obiettivo del sistema è di offrire un supporto efficace per determinare come agire in situazioni critiche, e ciò deve avvenire il più tempestivamente possibile. È perciò richiesta la velocità di calcolo dei percorsi ogni volta che si presenta una nuova richiesta
- **Accessi concorrenti**: Il sistema deve permettere l'accesso a più utenti in contemporanea, che possono creare scenari e utilizzare lo strumento contemporaneamente
- **Database**: si richiede un database che abbia le caratteristiche idonee per supportare il funzionamento di un'applicazione web
- **Sicurezza**: non sono previsti particolari meccanismi per la sicurezza del sistema, in quanto dopo l'integrazione in una dashboard erediterà direttamente i meccanismi di accesso e autorizzazione presenti in Snap4City
- **Portabilità**: il sistema può essere utilizzato su ogni tipo di dispositivo, ad esempio un PC o uno smartphone

3.2 Architettura

La scelta dell'architettura di riferimento è una fase fondamentale nella progettazione di un sistema software efficiente. Questa fase include la scelta dell'hardware, dell'infrastruttura di rete e dei componenti software che costituiranno il sistema, con il fine di trovare la combinazione di componenti ottimale, ossia in grado di soddisfare i requisiti applicativi e rispettare i vincoli tecnologici.

L'architettura scelta deve assicurare delle buone prestazioni per garantire la qualità del servizio, deve essere scalabile, mantenibile, e sicura. Le scelte effettuate in questa fase influenzano anche i costi, la complessità e gli standard da utilizzare.

L'architettura scelta [1], mostrata in Figura 3.1, è quella di un applicativo web, generalmente chiamata **architettura a tre strati** (*three tier*). Quest'architettura pone un livello intermedio tra i *client* e i dati, con il fine di contenere la logica applicativa del programma.

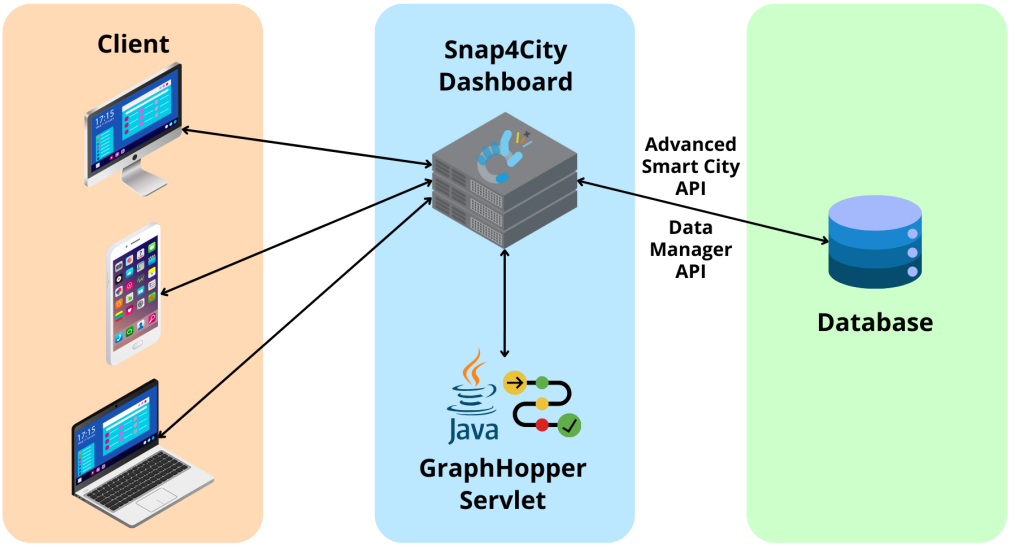

Figura 3.1: Architettura di rete

Le parti che compongono l'architettura, come rappresentato in Figura 3.1 sono:

- **Client**: dispositivi che accedono ai servizi e alle risorse messe a disposizione dal server, che è anch'esso un sistema informatico. Nel caso specifico, il client comunica con il server tramite la rete Internet. I dispositivi del client possono essere di diversa natura, sia dal punto di vista software che hardware. Ad esempio, nella Figura 3.1 sono riportati tre diversi dispositivi: un PC fisso, un dispositivo mobile e un PC portatile. Il software realizzato consente l'interazione tra dispositivi diversi, ed è stato quindi progettato per funzionare correttamente su diversi dispositivi.

- **Server**: dispositivo hardware o software che esegue il programma principale, e offre un servizio per il software eseguito sui client. Esso fornisce ai client dei servizi attraverso la rete. In particolare, il server dell'architettura di riferimento è un **web server**, ossia in grado di gestire le richieste di trasferimento di pagine web da parte di un client, tipicamente rappresentato come un web browser. La comunicazione tra client e server avviene attraverso il protocollo **HTTP** sulla porta TCP 80 del server, oppure utilizzando il protocollo **HTTPS** sulla porta TCP 443. In particolare, il server in Figura 3.1 è stato chiamato **Snap4City**

Dashboard in quanto è il server sulla quale gira l'applicativo per la gestione delle dashboard di Snap4City, tramite la quale è possibile utilizzare l'applicativo realizzato.

- **GraphHopper Servlet**: servizio di routing implementato che estende le funzionalità di GraphHopper, alla quale la dashboard di Snap4City effettua le richieste per il calcolo del percorso in base ai parametri scelti dal client.
- **Database**: collezione di dati strutturati sotto forma tabellare che permettono operazioni di ricerca, inserimento, modifica e cancellazione. In questo database vengono salvati gli **scenari** inseriti dai Client, che identificano delle chiusure stradali in alcune aree della città. Per leggere e salvare i dati sul database, il Server utilizza le **Data Manager API** e le **Advanced Smart City API**, che permettono l'accesso alla *knowledge base* di Snap4City e ai dati necessari al funzionamento di alcuni servizi. Queste API permettono di lavorare con la base di dati senza conoscerne in modo approfondito la struttura, in quanto l'interfaccia fornita è ad alto livello.

3.3 Sequence diagram

Un **sequence diagram** (diagramma di sequenza) è un tipo di diagramma UML utilizzato per descrivere uno scenario, ossia una determinata sequenza di azioni in cui tutte le scelte sono già state effettuate. Il tempo scorre dall'alto verso il basso, e gli attori coinvolti sono rappresentati come linee verticali. Una freccia che parte da un attore rappresenta un messaggio o richiesta, mentre una freccia tratteggiata rappresenta la risposta a un messaggio. In Figura 3.2 è riportato il sequence diagram che illustra cosa accade quando un client effettua una richiesta di navigazione da una sorgente a una destinazione.

Il sequence diagram in Figura 3.2 rappresenta la sequenza delle azioni svolte alla richiesta di routing, ed è composto dalle seguenti fasi:

1. Il Client effettua una richiesta di navigazione alla Dashboard, dopo aver scelto il punto di partenza e di arrivo, lo scenario (che indica le aree bloccate sulla mappa), l'obiettivo della navigazione (ad esempio `fastest` o `shortest`), e l'orario di partenza, richiede alla dashboard di mostrare il percorso da effettuare
2. La Dashboard richiede al Database i dati dello scenario, per mostrare le aree bloccate sulla mappa

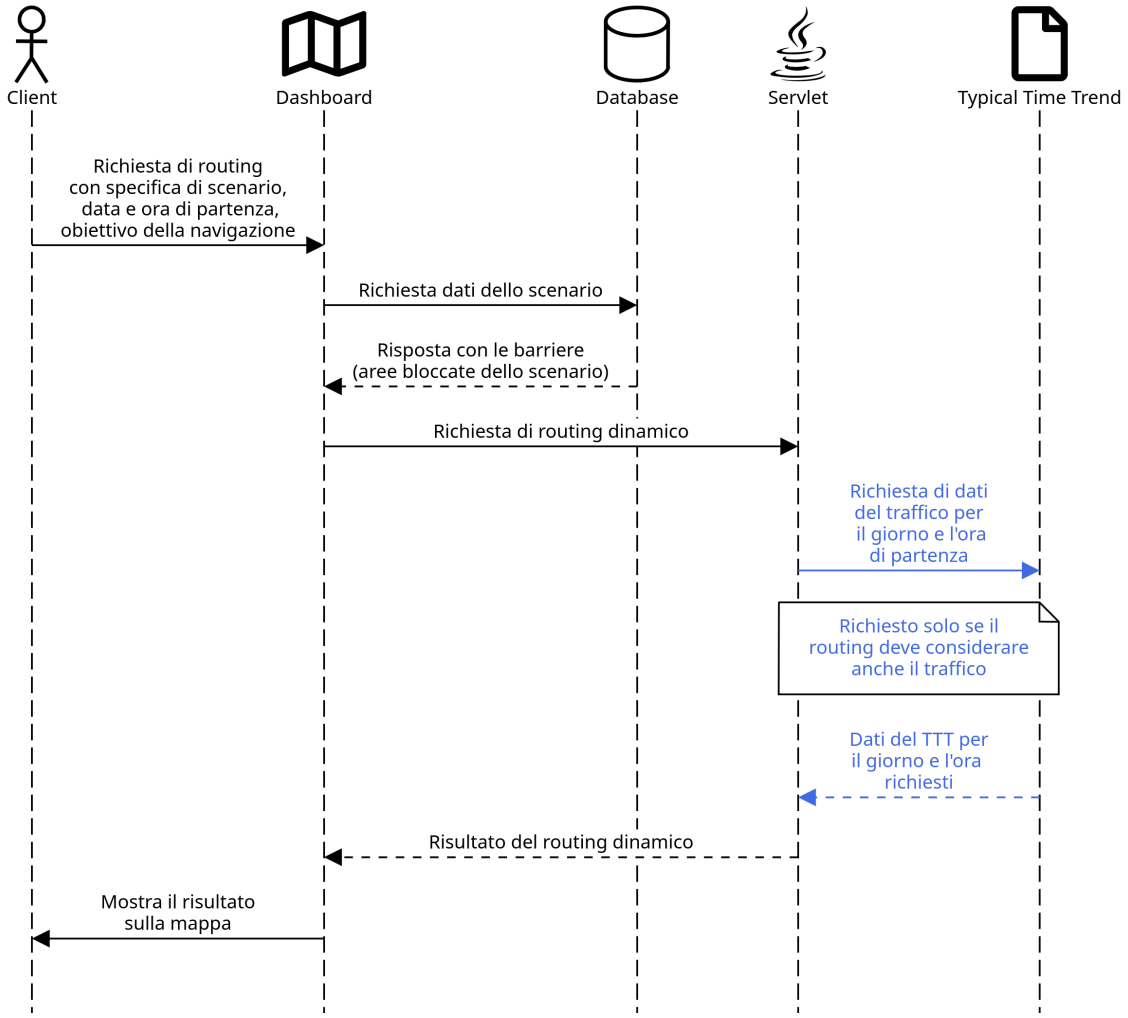

Figura 3.2: Sequence diagram che rappresenta ciò che avviene quando un client effettua la richiesta di navigazione

3. Il Database restituisce le aree bloccate alla Dashboard
4. La Dashboard effettua una richiesta di routing alla Servlet, per ottenere le informazioni sulla navigazione
5. Se la richiesta di routing include i dati sul traffico, la Servlet legge i dati del *Typical Time Trend* relativi al giorno e l'ora di partenza
6. La Servlet elabora la richiesta e calcola il percorso migliore in base ai parametri
7. La Servlet risponde alla Dashboard inviandole le informazioni di navigazione
8. La Dashboard mostra al Cliente il percorso risultante su una mappa

3.4 Class diagram

Un **diagramma delle classi** serve a descrivere le varie entità coinvolte nel software, tramite i loro attributi e metodi, e a mostrare come queste classi collaborano tra di loro. Il paradigma utilizzato è quello della programmazione a oggetti, in quanto il software implementato e la libreria utilizzata (GraphHopper, [23]) sono scritte in Java.

In Figura 3.3 è riportato il diagramma delle classi del software implementato, in cui sono presenti due package: `servlet`, in cui è presente il software implementato per il progetto, e `graphhopper`, che rappresenta la libreria utilizzata per il routing.

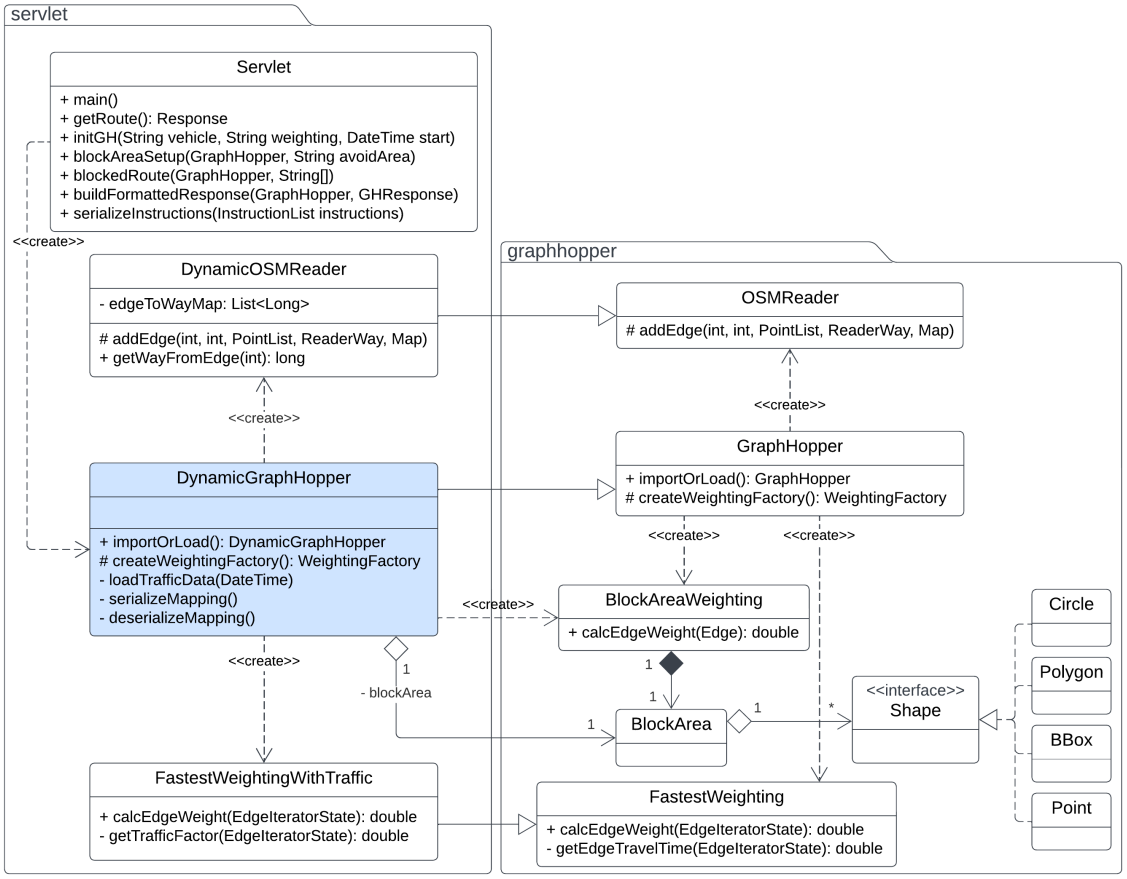

Figura 3.3: Diagramma delle classi

La classe `Servlet` rappresenta l'endpoint della servlet Jersey, che riceve le richieste HTTP, inizializza un oggetto di tipo `DynamicGraphHopper`, e delega ad esso la ricerca del percorso ottimo. Ha inoltre vari metodi utili per il routing e per il debugging.

La classe `DynamicGraphHopper`, riportata in celeste in Figura 3.3, è la classe principale. Eredita direttamente dalla classe `GraphHopper`, della libreria `graphhopper`,

della quale sovrascrive i metodi `importOrLoad()` e `createWeightingFactory()`. Essa utilizza la classe `DynamicOSMReader`, che estende la classe `OSMReader` di GraphHopper, per ottenere il mapping tra *Way* e *Edge* (come descritto in Sezione 4.2). Il mapping è inoltre salvato e riletto nel file system, tramite i metodi `serializeMapping()` e `deserializeMapping()`. Queste informazioni sono utilizzate in caso di routing che considera i dati di traffico, e vengono caricate tramite il metodo `loadTrafficData()`. Il metodo `createWeightingFactory()` si occupa di creare un oggetto di tipo `WeightingFactory`, che serve a creare i pesi per ogni segmento stradale secondo il pattern **Factory** [29]. Un tipo di peso (`Weighting`) utilizzato è il `FastestWeightingWithTraffic`, che si occupa di creare dei pesi in base ai dati sul traffico e alla velocità di percorrenza di un arco stradale. In caso non si utilizzi la modalità di navigazione `fastest_with_traffic`, viene utilizzato il metodo `createWeightingFactory()` della classe base `GraphHopper`, che crea la Factory corretta tra quelle predefinite nella libreria.

La tipologia di peso `BlockAreaWeighting` si occupa di fare da "wrapper" di un'altra tipologia di peso, ed è sempre utilizzata quando c'è un'area bloccata. Essa restituisce un peso infinito per gli archi non attraversabili, e restituisce il peso creato dalla `Weighting` "wrappata" altrimenti. Questa classe utilizza un oggetto di tipo `BlockArea` per capire quali zone della mappa non possono essere attraversate, che a sua volta contiene una lista di `Shape`, ossia forme geometriche che rappresentano i luoghi non visitabili (di forme come `Circle`, `Polygon`, `BBox` e `Point`).

Capitolo 4

Implementazione

La servlet implementata è stata scritta in Java, utilizzando l'ambiente di sviluppo integrato (IDE) Intellij IDEA 2023.2. L'ambiente di sviluppo sulla quale è stato testato il programma è un laptop Aspire 7, con un processore Intel Core i7-9750H 2.6GHz, 24GB di RAM, con sistema operativo Fedora Linux 38.

Il codice è stato testato tramite l'utilizzo di container Docker, spiegato in dettaglio nella Sezione 4.4, in cui sono inclusi tutti i servizi base di Snap4City. Questi container forniscono inoltre la possibilità di usufruire del servizio offerto dalla servlet usando delle dashboard come interfaccia grafica, che permettono la visualizzazione di una mappa e l'inserimento dei parametri necessari alla navigazione.

4.1 Evoluzione del programma esistente

Il primo passaggio per poter evolvere il software è stato aggiornare le dipendenze delle librerie utilizzate, in particolare GraphHopper, alla versione più recente. Il software di partenza infatti utilizzava la versione `0.13` della libreria, mentre la versione più recente è la `7.0`.

Per permettere la retrocompatibilità del programma è stato necessario fare riferimento direttamente al codice e ai changelog nella repository di GraphHopper [23], in quanto la documentazione della libreria si concentra principalmente sull'utilizzo dell'API, e non sull'estensione del codice Java. Alcune classi della libreria sono state rinominate, pur mantenendo un funzionamento simile al precedente (ad esem-

pio `QueryResult` ⟶ `Snap`), altre sono state rimosse (come `PathWrapper`), e altre invece sono state introdotte (come `ResponsePath` e `InstructionListSerializer`).

La classe `GraphHopper`, estesa da `DynamicGraphHopper`, non ha più il metodo `createWeighting()`, bensì viene utilizzato il design pattern **Factory** [29] per creare una classe che rappresenta il peso di un arco della mappa tramite il metodo `createWeightingFactory()`, come mostrato in Figura 4.1.

```
// Override the createWeighting method of the GraphHopper class to enable BlockAreaWeighting
@Override  3 usages
protected WeightingFactory createWeightingFactory() {
  // Get encoded values for the vehicle
  EncodingManager em = this.getEncodingManager();
  BooleanEncodedValue accessEnc = em.getBooleanEncodedValue(VehicleAccess.key(this.getProfiles().get(0).getVehicle()));
  DecimalEncodedValue speedEnc = em.getDecimalEncodedValue(VehicleSpeed.key(this.getProfiles().get(0).getVehicle()));

  // Get the weighting in use
  String weighting = this.getProfiles().get(0).getWeighting();
  WeightingFactory result;

  if (weighting.equals("fastest_with_traffic")) {
    if (edgeToWayMap.isEmpty()) {
      result = (Profile profile, PMap hints, boolean disableTurnCosts) -> new FastestWeighting(accessEnc, speedEnc);
    }
    else {
      result = (Profile profile, PMap hints, boolean disableTurnCosts) -> new FastestWeightingWithTraffic(trafficData, accessEnc, speedEnc, edgeToWayMap);
    }
  }
  // Other default weightings, like "shortest", "short_fastest", etc. See https://github.com/graphhopper/graphhopper/blob/master/docs/core/profiles.md
  else result = super.createWeightingFactory();

  // Add the blockArea to the weighting
  if (blockArea != null) {
    // Create a new WeightingFactory
    // with the createWeighting method that returns a BlockAreaWeighting if a BlockArea is set, and uses the "result" weighting otherwise
    return (Profile profile, PMap hints, boolean disableTurnCosts) -> {
      Weighting w = result.createWeighting(profile, hints, disableTurnCosts);
      return new BlockAreaWeighting(w, blockArea);
    };
  }
  else return result;
}
```

Figura 4.1: Metodo createWeightingFactory() della classe *DynamicGraphHopper*

L'estensione del programma si è poi concentrata sull'introdurre la modalità di navigazione `fastest_with_traffic`, in cui viene utilizzato il Typical Time Trend per stimare la densità del traffico stradale al momento della partenza. Per fare ciò, è stata modificata la classe `DynamicGraphHopper` per far sì che essa considerasse sempre un'area bloccata, a prescindere dall'obiettivo della navigazione. Prima dell'evoluzione del software, l'unica modalità disponibile era la `block_area`, che cercava il percorso più veloce nonostante le aree bloccate. Questa modalità è stata rimossa in quanto l'area bloccata (`BlockArea`) adesso può essere specificata a prescindere dal modo con cui vengono calcolati i pesi degli archi (`Weighting`), come mostrato in Figura 4.2, e ciò permette all'utente di poter specificare una diversa modalità di navigazione, ad esempio `shortest`, che cerca il percorso più breve nonostante le aree bloccate.

La modalità `fastest_with_traffic` si comporta come la modalità `fastest`, ossia cerca il percorso più veloce per arrivare da sorgente a destinazione, ma effettua il calcolo in base ai dati storici sulla densità di traffico per ogni arco. Per poter fare ciò è stata implementata la classe `FastestWeightingWithTraffic`, che estende

```java
@GET
@Produces(MediaType.TEXT_PLAIN)
public static Response getRoute(@DefaultValue("car") @QueryParam("vehicle") String vehicle,
                                @DefaultValue("") @QueryParam("avoid_area") String avoidArea,
                                @DefaultValue("") @QueryParam("waypoints") String waypoints,
                                @DefaultValue("") @QueryParam("startDatetime") String startTimestamp,
                                @DefaultValue("fastest") @QueryParam("weighting") String weighting) {
    _vehicle = vehicle;

    // If the startDatetime is not specified, use the current datetime
    LocalDateTime startDatetime;
    if (startTimestamp.isEmpty()) startDatetime = LocalDateTime.now();
    else startDatetime = LocalDateTime.parse(startTimestamp);

    // 1: init GH
    DynamicGraphHopper hopper = initGH(_vehicle, weighting, startDatetime);

    // 2. If there's an avoidArea, apply the avoidArea
    if (!avoidArea.isEmpty()) {
        blockAreaSetup(hopper, avoidArea);  // extract barriers and apply them
    }

    // 3: extract waypoints
    String[] waypointsArray = waypoints.split( regex: ";");

    // 4: perform blocked routing
    GHResponse response = blockedRoute(hopper, waypointsArray);

    // 5: build response
    JSONObject jsonResponse = buildFormattedResponse(hopper, response);
    return Response.ok(jsonResponse.toString()).header( s: "Access-Control-Allow-Origin",  o: "*").build();
}
```

Figura 4.2: Endpoint Jersey nella classe *Servlet*

la classe `FastestWeighting` di GraphHopper, per fare in modo che la durata del percorso venisse calcolata anche in base alla congestione stradale, come mostrato in Figura 4.3.

```java
/**
 * Use the traffic data to calculate the travel time of the edge.
 * If the edge is not in the map, it means that I can't get its traffic data, so I calculate the travel time at max speed.
 *
 * @param edgeState edge to calculate the travel time of
 * @return the travel time of the edge
 */
private double getEdgeTravelTime(EdgeIteratorState edgeState) {  1 usage

    // Get the traffic data for each road element that belongs to the way (the RoadElement id contains the Way id)
    long wayId = edgeToWayMap.get(edgeState.getEdge());
    List<Float> averageDensityList = new ArrayList<>();
    float maxDensity = 0;

    List<String> roadElementsOfWay = new ArrayList<>();
    trafficData.keySet().forEach(roadElementId -> {
        if (roadElementId.contains(Long.toString(wayId))) roadElementsOfWay.add(roadElementId);
    });

    for (String roadElementId : roadElementsOfWay) {
        Pair<Float, Float> trafficData = this.trafficData.get(roadElementId);
        averageDensityList.add(trafficData.first);
        maxDensity = trafficData.second;
    }

    // Calculate the average traffic density
    float averageDensity = averageDensityList.stream().reduce( identity: 0f, Float::sum) / averageDensityList.size();

    // If the way has no traffic data, return the travel time of the edge without considering the traffic
    if (roadElementsOfWay.isEmpty()) return super.calcEdgeWeight(edgeState,  reverse: false);
    // If the average traffic density is greater than the maximum traffic density, return infinity
    // (critical condition, the road is blocked from the traffic)
    if (averageDensity > maxDensity) return Double.POSITIVE_INFINITY;

    // Calculate the speed
    double speed = maxSpeed * (1 - averageDensity / maxDensity);    // As the density increases, the speed decreases (Greenshield's model)
    return edgeState.getDistance() / speed * SPEED_CONV;
}
```

Figura 4.3: Metodo della classe *FastestWeightingWithTraffic* per calcolare il tempo di percorrenza di un arco secondo il modello di Greenshield [9]

I dati sul traffico sono forniti rispetto ai *RoadElement* di Snap4City, che sono legati alle *Way* di OpenStreetMap tramite un codice identificativo. È stato quindi necessario estendere anche la classe `OSMReader`, usata da GraphHopper per leggere il file della mappa, per tenere traccia del legame tra *Way* di OpenStreetMap e *Edge* di GraphHopper, come spiegato in Sezione 4.2 e mostrato in Figura 4.4.

```
@Override
protected void addEdge(int fromIndex, int toIndex, PointList pointList, ReaderWay way, Map<String, Object> nodeTags) {
    // This method might be called multiple times for each way.
    // Every time this method is called, it means that a new edge is being added to the graph, with an increasing edgeId.
    super.addEdge(fromIndex, toIndex, pointList, way, nodeTags);

    // Update the mapping between the edge and the ways that it belongs to
    edgeToWayMap.add(way.getId());
}
```

Figura 4.4: Salvataggio del mapping tra Way e Edge nella classe *DynamicOSMReader*

4.1.1 Gestione del progetto

Apache Maven

Il software è stato realizzato tramite **Apache Maven** [30], uno strumento per la gestione di progetti Java che fornisce funzionalità di *build automation* e download automatico delle librerie necessarie sotto forma di file JAR. Maven usa un costrutto conosciuto come **Project Object Model** (POM), che è un file XML nella quale vengono scritte le dipendenze fra il progetto e le varie versioni delle librerie utilizzate.

In particolare, le dipendenze specificate nel file `pom.xml` sono le seguenti:

- `jersey-bundle`: per utilizzare le annotazioni di Jersey per la creazione dell'endpoint
- `graphhopper-core`: libreria principale di GraphHopper
- `json`: per poter fare *marshalling* e *unmarshalling* di dati in formato JSON
- `gson`: per la serializzazione/deserializzazione di oggetti Java in JSON

Repository su GitHub

Per poter gestire al meglio il progetto è stata creata una repository su **GitHub** [31], uno dei più famosi *Version Control System* (VCS), che permette di salvare le modifiche fatte al codice durante lo sviluppo, in modo da tenere traccia dei cambiamenti e fornire flessibilità nella progettazione.

4.2 Gestione dei dati sul traffico

Nonostante la scarsa documentazione di GraphHopper, navigando le classi GraphHopper, OSMReader, Weighting e WaySegmentParser è stato possibile capire come il file della mappa di OpenStreetMap (con estensione `.pbf`) viene letto, e come viene creata la struttura dati interna usata per il calcolo del percorso. Questa struttura dati è rappresentata dalla classe BaseGraph, che è una struttura a grafo i cui archi sono chiamati *Edge*.

Per formare questi Edge, GraphHopper legge il file `.pbf` della mappa e itera le strade presenti, chiamate *Way* da OpenStreetMap [16]. Ogni Way è composta da una serie di nodi, chiamati *Node* [17], come spiegato anche nella sezione 2.1. I Nodes vengono raggruppati da GraphHopper con dei criteri interni alla libreria, ad esempio rimuovendo i nodi sovrapposti (confrontandone le coordinate), e compone quelli che in GraphHopper sono chiamati *Edge* (struttura dati non presente in OpenStreetMap).

I percorsi salvati nel grafo di GraphHopper sono questi Edge, il cui peso dipende dalla funzione obiettivo scelta. Ad esempio, se l'obiettivo del percorso è `shortest`, il peso di ogni arco è la propria lunghezza. È stato quindi necessario trovare un modo per mantenere la mappatura tra le *Way* di OpenStreetMap e gli *Edge* di GraphHopper, in quanto gli Edge sono utilizzati da GraphHopper per il routing, e le Way sono utilizzate per assegnare i pesi tramite i dati del traffico (Figura 4.4).

Anche Snap4City effettua una propria suddivisione delle strade (Way) di OpenStreetMap in segmenti chiamati *RoadElement* [32]. Una Way (chiamata *Road* in Km4City) può essere scomposta in più *RoadElement*, ad esempio per tenere conto dei tratti stradali separati da un incrocio o della direzione di percorrenza della strada, come rappresentato in Figura 4.5.

4.2.1 Richiesta dei dati sul traffico

I dati sul traffico sono calcolati a partire dai dati dei sensori tramite il Typical Time Trend (TTT), come spiegato in Sezione 1.5.3.

Al momento del calcolo dei pesi per gli archi del grafo, non è però necessario sapere il TTT di ogni ora e di tutti i giorni della settimana per ogni tratto stradale, bensì è sufficiente poter conoscere le informazioni sul traffico *al momento della partenza* su ogni segmento stradale. Il programma permette infatti di pianificare l'orario di partenza, in modo tale da poter lavorare soltanto con i dati di traffico di uno specifico

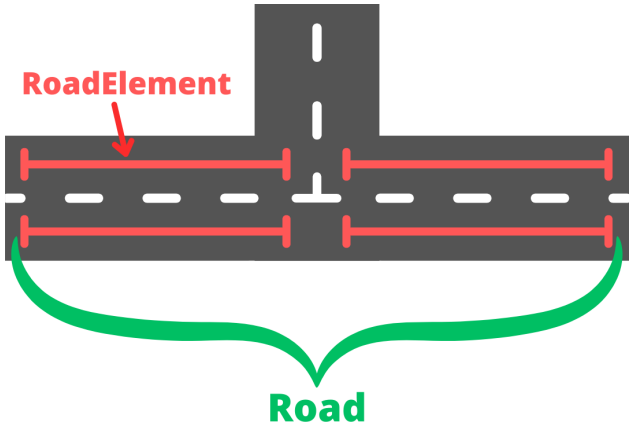

Figura 4.5: Esempio di suddivisione di una strada in RoadElement, a seconda della direzione stradale e degli incroci

giorno e orario della settimana. Per permettere ciò è stato implementato uno script Python che convertisse il file `ttt.json`, contenente una matrice 7×24 con il TTT per ogni RoadElement, in 7×24 file distinti, ognuno che rappresenta il TTT di un preciso giorno e orario della settimana (come mostrato in Figura 4.6).

Ad esempio, il file `4_12.json` rappresenta il valore del Typical Time Trend per ogni tratto stradale durante le ore 12 del venerdì (5° giorno della settimana, quindi n°4 contando da 0).

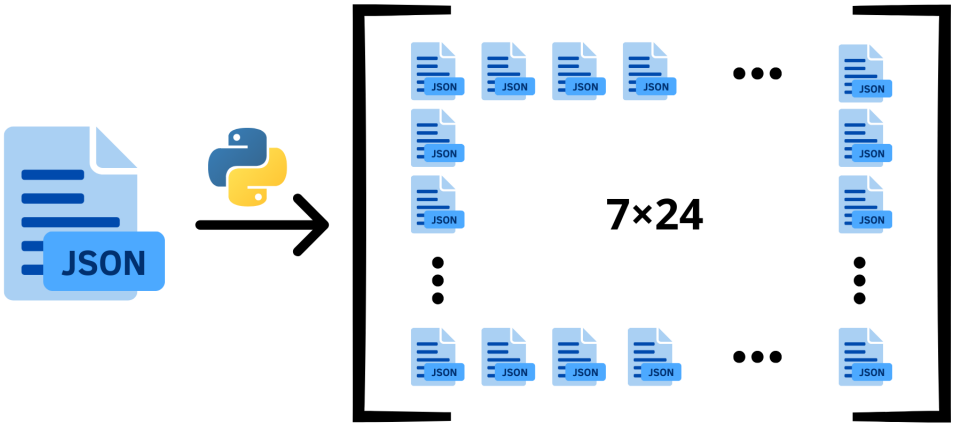

Figura 4.6: Scomposizione dei file JSON contenente i TTT settimanali

4.3 Interfaccia grafica

Per poter utilizzare il programma è stata creata una dashboard pubblica all'interno di Snap4City, utilizzando il **Dashboard Builder** [33], che permette la completa configurazione di ciò che viene visualizzato nell'interfaccia grafica.
In Figura 4.7 è riportata la configurazione dei selettori, in cui è specificato l'elenco di ciò che è possibile aggiungere alla mappa. In particolare è stato aggiunto il livello per la creazione degli scenari (`Scenario`), quello per il routing (`WhatIf`) e quello per mostrare il traffico (`Traffic TTT`).

Figura 4.7: Configurazione dei selettori della dashboard

Per poter gestire i parametri aggiuntivi richiesti dalla servlet, sono stati modificati alcuni file che controllano il funzionamento dell'interfaccia grafica. In particolare, sono stati aggiunti due campi per selezionare l'obiettivo della navigazione (`Weighting`) e l'orario della partenza (mostrati in Figura 4.8), e sono stati aggiunti i parametri alla query effettuata alla servlet.

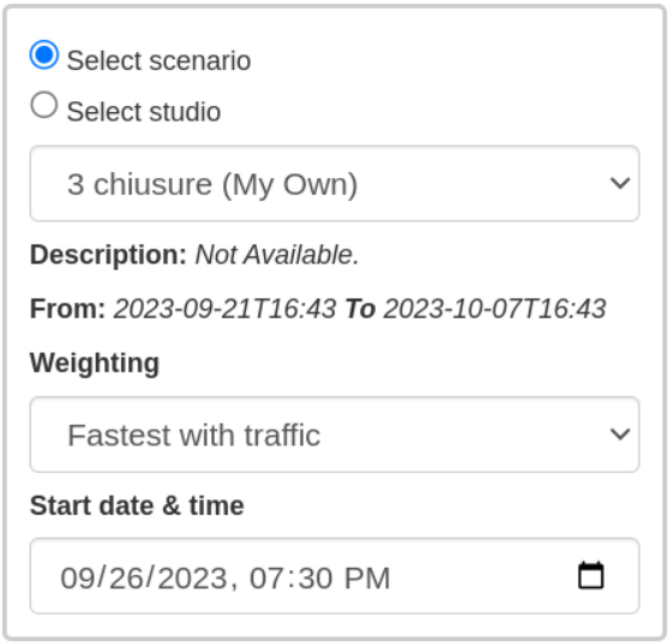

Figura 4.8: Parametri di configurazione della navigazione

La dashboard risultante è mostrata in Figura 4.9, composta dai seguenti elementi:

- **Selettori**: servono a selezionare cosa visualizzare sulla mappa
- **Creazione degli scenari**: strumento per definire e salvare degli scenari contenenti dei blocchi stradali
- **Selezione dei parametri**: strumento per scegliere le impostazioni per il calcolo della navigazione
- **Mappa**: mostra le aree bloccate, i punti da attraversare e il percorso risultante
- **Selezione del veicolo**: permette di scegliere quale veicolo usare per la navigazione
- **Descrizione della navigazione**: elenco d'indicazioni da seguire per arrivare dalla sorgente alla destinazione

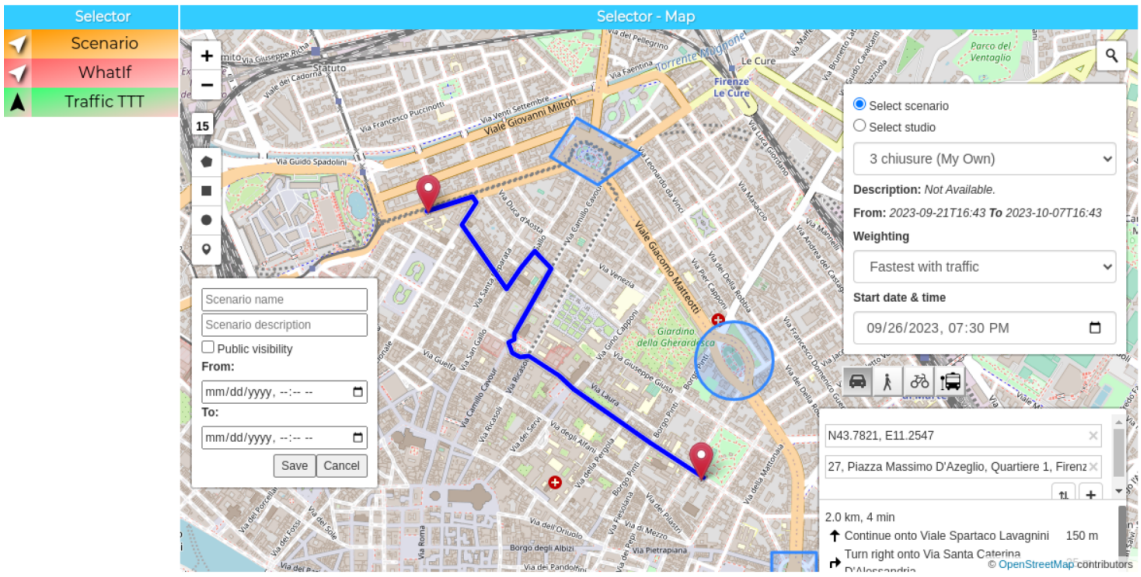

Figura 4.9: Dashboard per la What-If Analysis

4.4 Ambiente Docker

Per poter testare al meglio il software è stato configurato un ambiente Docker fornito da Snap4City, che prende il nome di **MicroX**. Questo ambiente Docker ha integrati al suo interno vari servizi e utilizza diverse immagini con lo scopo di far funzionare Snap4City. In particolare, fornisce la possibilità di poter accedere alle dashboard e di aggiungere altre immagini utili allo sviluppo. Lo schema delle immagini presenti in una installazione MicroX è rappresentato in Figura 4.10.

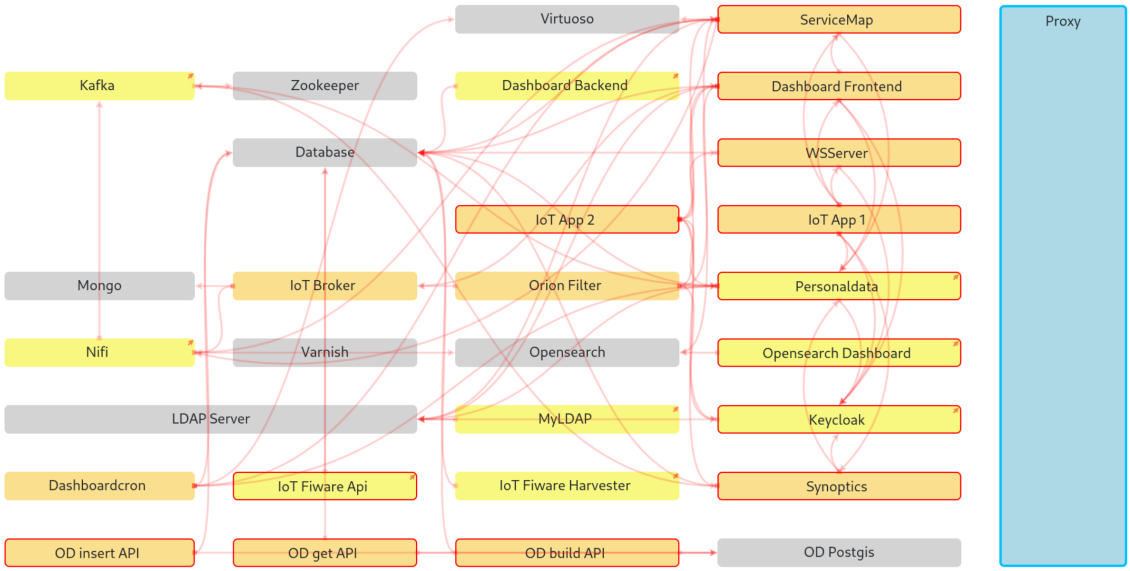

Figura 4.10: Struttura delle immagini Docker nella configurazione MicroX di Snap4City

Per poter utilizzare la servlet è stato aggiunto un ulteriore container al file `docker-compose.yml`, con il nome `graph-hopper`, come mostrato in Figura 4.11. Questo container ha lo scopo di ospitare la servlet e renderla accessibile dall'esterno, garantendo così che le dashboard possano inviargli delle richieste di routing.

```
1  graph-hopper:
2    image: tomcat:9.0.26-jdk8-openjdk-slim
3    ports:
4      - 8180:8080
5    volumes:
6      - ./gh/GHServlet.war:/usr/local/tomcat/webapps/GHServlet.war
7      - ./gh/toscana.osm.pbf:/usr/local/tomcat/toscana.osm.pbf
8      - ./gh/typical_ttt:/usr/local/tomcat/typical_ttt
```

Figura 4.11: Configurazione del container contenente la Servlet

Capitolo 5

Risultati sperimentali

In questo capitolo sono mostrati i risultati di alcuni casi d'uso che sono stati creati per mettere alla prova il programma, volti a mostrare la differenza nel calcolo del percorso ottimo nelle varie modalità di navigazione.

Gli esempi mostrati validano il software da un punto di vista qualitativo, in quanto permette di mostrare intuitivamente la correttezza del software; la correttezza dal punto di vista quantitativo è data per certa in quanto il framework GraphHopper garantisce la ricerca del percorso minimo all'interno del grafo stradale, che sarà quindi la soluzione ottima al problema del *Vehicle Routing Problem* [4]. Questa caratteristica può essere confermata osservando la distanza e il tempo di percorrenza del tragitto risultante, al variare dei parametri della navigazione.

5.1 Casi d'uso

I casi d'uso mostrati evidenziano come il percorso ottimo può variare in base ai parametri inseriti dall'utente: i luoghi di partenza e di destinazione sono gli stessi per ogni esempio riportato, ciò che varia sono le aree bloccate e l'obiettivo di navigazione scelto.
In ogni esempio è anche riportato il traffico nella data e ora considerate, per mostrare come questo possa influenzare la ricerca del percorso ottimo.

Il primo caso d'uso, riportato in Figura 5.1, mostra il percorso trovato quando non sono presenti aree bloccate, non viene tenuto in considerazione il traffico e l'obiettivo della navigazione è `fastest`: il tempo di percorrenza è 3 minuti e la lunghezza del tragitto è 2,6km.

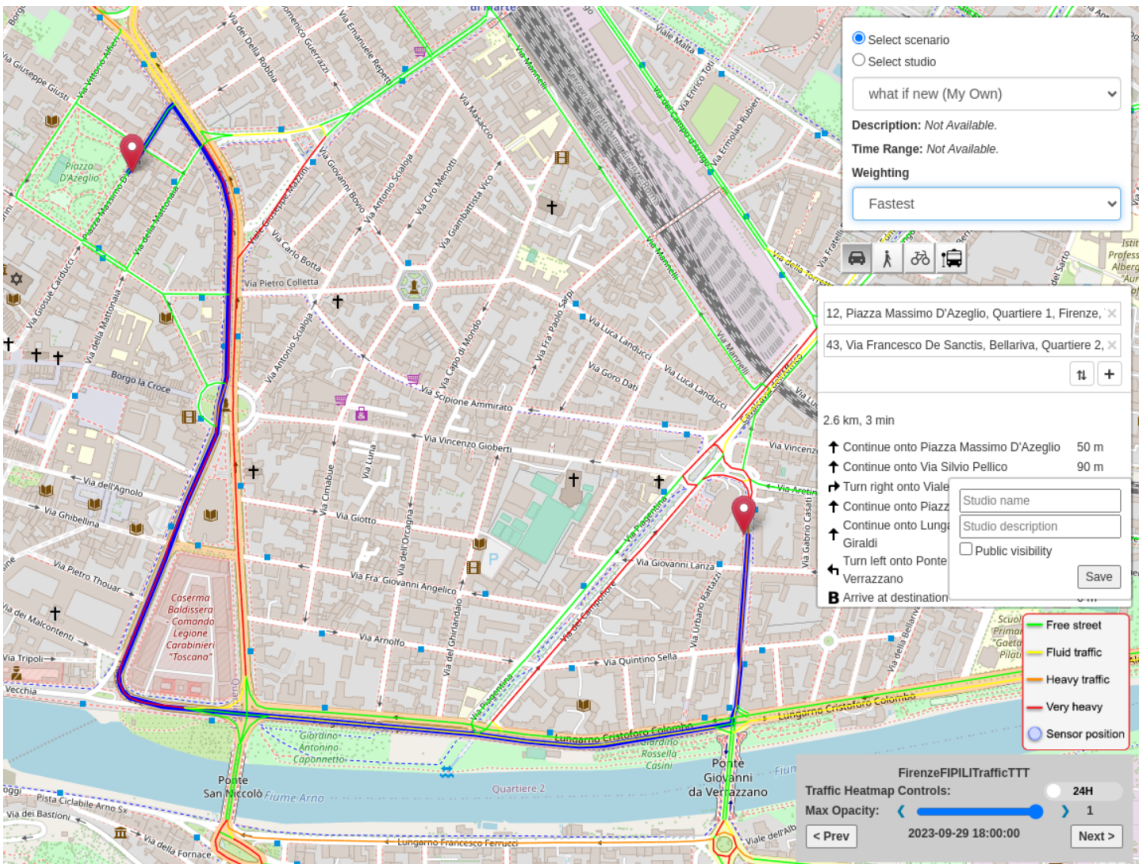

Figura 5.1: Esempio di navigazione semplice
in cui si cerca il percorso più veloce tra sorgente e destinazione

Modificando l'obiettivo della navigazione in `shortest`, come mostrato in Figura 5.2, si osserva che il percorso ottenuto ha un tempo di percorrenza di 3 minuti e 30 secondi, maggiore rispetto al caso precedente, e una lunghezza di 2,1km, inferiore alla modalità `fastest`.

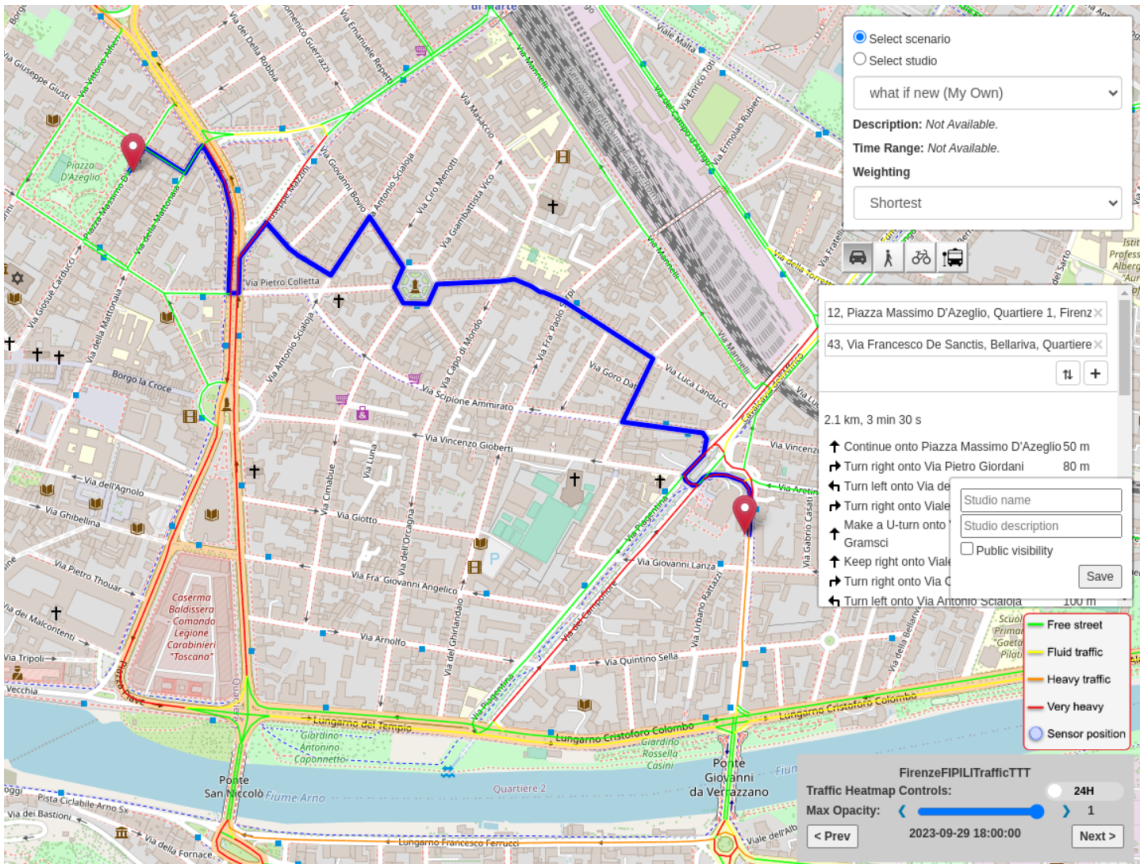

Figura 5.2: Navigazione in cui si cerca il percorso più breve

Tenendo in considerazione il traffico (quindi cambiando l'obiettivo della navigazione in `fastest_with_traffic`), il percorso ottimo trovato in Figura 5.1 si rivela non essere più il percorso più veloce, in quanto il traffico rallenta il flusso stradale durante il tragitto. Il percorso trovato, riportato in Figura 5.3, evita le strade più trafficate (riportate in rosso), ha una lunghezza di 2,2km e un tempo di percorrenza di 4 minuti: questo evidenzia come tenere in considerazione il traffico permette un'ottimizzazione durante il calcolo della navigazione.

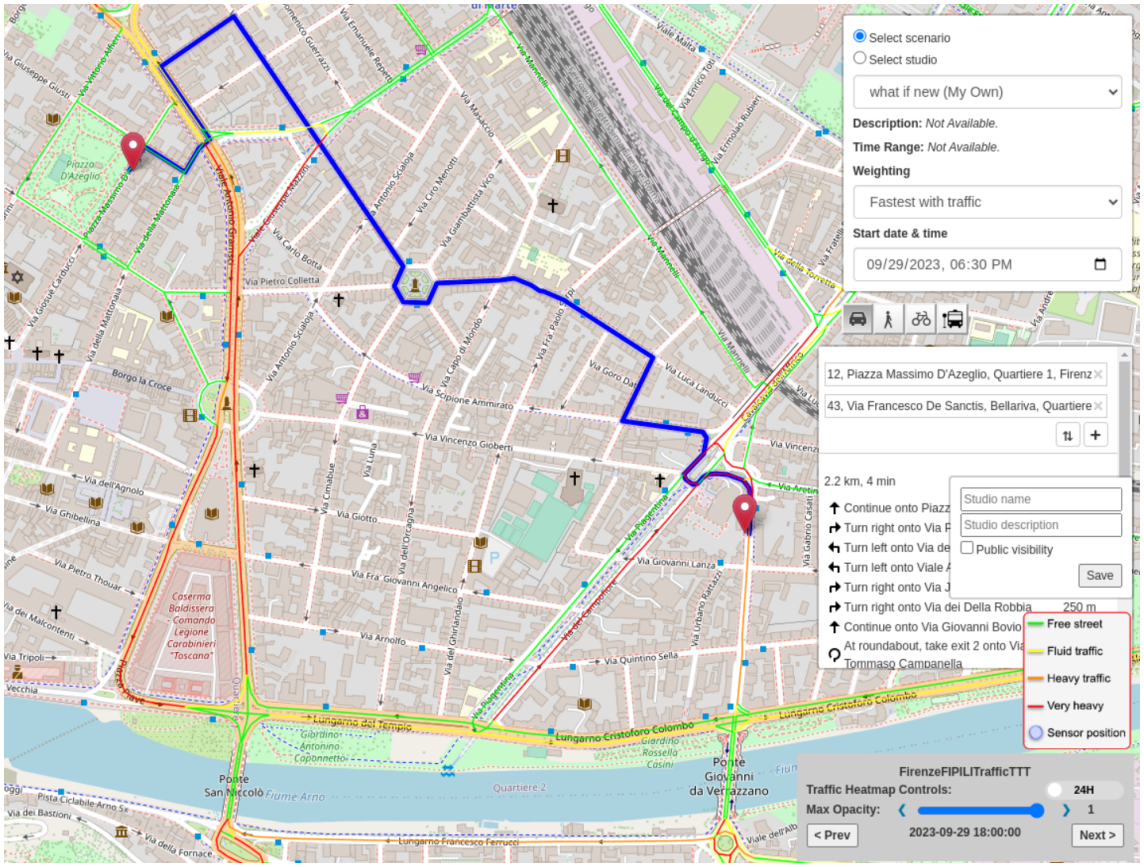

Figura 5.3: Navigazione in cui viene cercato il percorso più veloce considerando il traffico lungo il percorso

Introducendo infine delle aree bloccate, come riportato in Figura 5.4, il percorso più veloce subisce un'ulteriore variazione, in quanto parte del tragitto percorso non è più accessibile. Il percorso risultante ha una lunghezza di 2,4km e un tempo di percorrenza di 4 minuti, e come nel caso precedente evita le zone più trafficate della città.

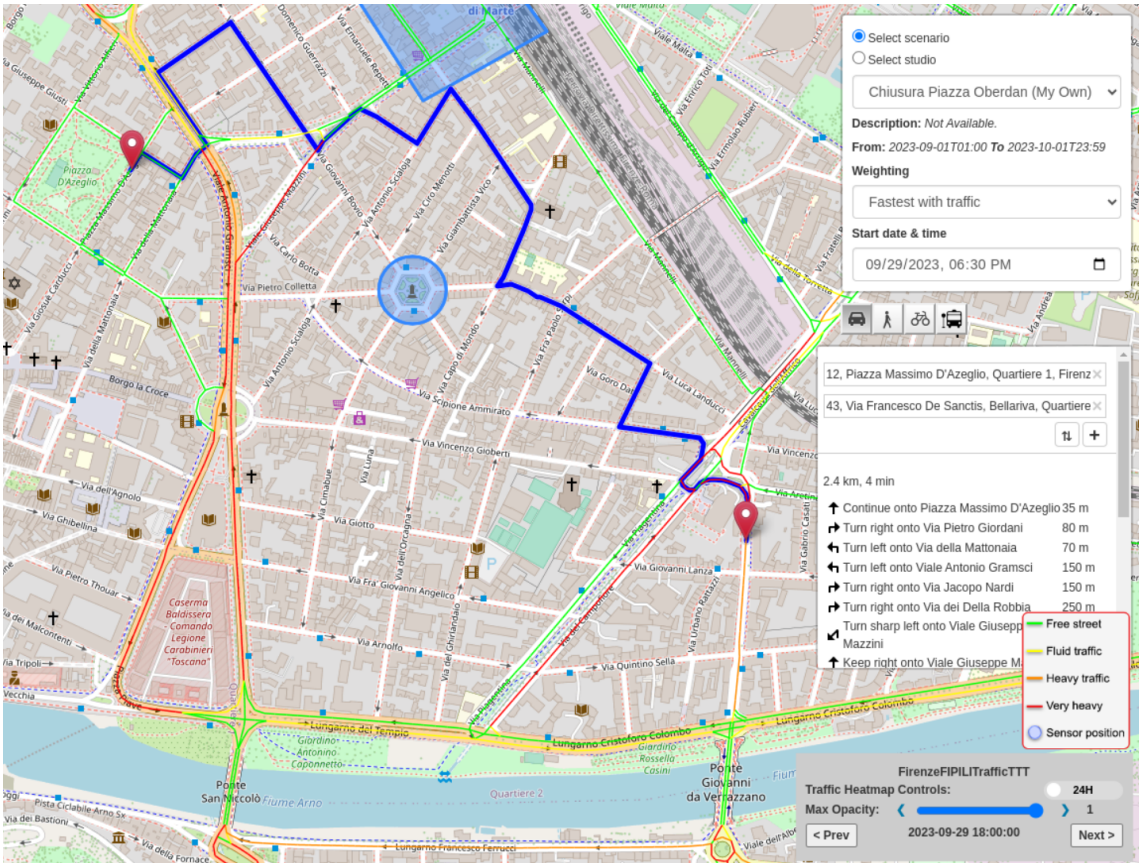

Figura 5.4: Navigazione in cui viene cercato il percorso più veloce considerando le aree bloccate e il traffico lungo il percorso

5.2 Performance

Il calcolo del percorso ottimo in base ai parametri è un'operazione che richiede una quantità di tempo variabile, che dipende sia dai parametri di navigazione che dal server che ospita la servlet. I test sono stati effettuati sull'ambiente di sviluppo descritto nel Capitolo 4, quindi su un calcolatore con performance limitate, destinato all'uso personale.

L'esecuzione del programma con parametri standard, quindi con un percorso urbano in cui ci sono delle aree bloccate e un obiettivo di navigazione predefinito, impiega circa 200ms. In caso si consideri anche il traffico la chiamata HTTP alla servlet avrà una durata di circa 400ms, in quanto sarà necessario leggere su un file i dati del Typical Time Trend relativi alla densità di traffico. Aumentando la distanza tra la sorgente e la destinazione, la chiamata potrebbe impiegare fino ad alcuni secondi in quanto l'elaborazione deve effettuare più calcoli per ogni segmento stradale. Infine, alla prima esecuzione GraphHopper non è a conoscenza della mappa sulla quale lavorerà, quindi la prima chiamata impiegherà fino a 30 secondi; ciò permette alla libreria di leggere il file della mappa, elaborarlo, e salvare in memoria una propria rappresentazione del grafo stradale, utilizzata nelle chiamate seguenti. Questa operazione è necessaria alla prima esecuzione per ogni profilo gestito da GraphHopper, ossia per diverse combinazioni di veicolo e di obiettivo della navigazione.

Il software garantisce inoltre l'accesso concorrente da parte di più utenti, in quanto gli utenti possono accedere in contemporanea alla dashboard di Snap4City, creare i propri scenari contenenti le aree bloccate e calcolare il percorso migliore in base alle personalizzazioni scelte.

Capitolo 6

Conclusioni

Nel corso di questo lavoro è stata affrontata una delle sfide più importanti nelle moderne Smart City: ottimizzare la mobilità urbana attraverso l'impiego di sensori per il monitoraggio del traffico. L'obiettivo principale è stato lo sviluppo di un software open source per la navigazione intelligente, focalizzato sulle esigenze e le preferenze dell'utente, nonché sul mezzo di trasporto utilizzato.

Il sistema di routing implementato non si limita a fornire semplici indicazioni stradali, ma si pone come obiettivo quello di creare soluzioni di mobilità personalizzate, considerando una molteplicità di fattori critici nell'ambito della mobilità urbana. L'utente ha la possibilità di scegliere tra differenti obiettivi di navigazione, come la ricerca del percorso più veloce o più breve. Inoltre, il sistema consente la definizione di vincoli personalizzati, come strade bloccate o zone da evitare, considerando eventi quali incidenti o manifestazioni. Queste funzionalità si traducono in percorsi ottimizzati che rispondono alle specifiche esigenze di ciascun utente, promuovendo una mobilità più efficiente e sostenibile all'interno della città.

Il software si integra perfettamente nell'ecosistema di Snap4City ed è utilizzabile in tutti i contesti urbani in cui sono presenti dei sensori del traffico, come nel caso d'uso della città di Firenze analizzato nei capitoli precedenti.

6.1 Sviluppi futuri

Le prospettive future di questo progetto comprendono varie migliorie che potrebbero essere apportate per ottimizzare ulteriormente la mobilità urbana in una Smart City.

Il progetto potrebbe essere esteso introducendo la navigazione con altre tipologie di veicolo, ad esempio veicoli emergenziali come le ambulanze, che hanno la facoltà di passare in zone chiuse al traffico urbano, come in ZTL (zone a traffico limitato), oppure i camion dei pompieri, che devono tenere conto della larghezza stradale durante la navigazione per via della dimensione del veicolo. Questo può essere fatto estendendo i *Custom Model* [34] di GraphHopper, che consente di definire condizioni personalizzate per i vari veicoli.

Potrebbero essere inoltre introdotte le informazioni sul trasporto pubblico, tramite l'utilizzo dei file GTFS (***General Transit Feed Specification***). Questo favorirebbe principalmente i cittadini nei propri spostamenti quotidiani e aiuterebbe a ridurre l'inquinamento atmosferico in città densamente abitate.

Oltre alla scelta del giorno e l'ora di partenza, potrebbe essere aggiunta la scelta dell'orario entro la quale l'utente vuole arrivare a destinazione, così da fornire una stima accurata dell'orario di partenza e della strada da percorrere.

Un'ulteriore estensione naturale del software realizzato è l'introduzione della modalità di navigazione `avoid_pollution`, che permette di trovare il percorso con meno inquinamento dell'aria. Ciò è possibile sfruttando i sensori della qualità dell'aria sparsi per la città. La modalità `shortest` permette di trovare il percorso più breve per raggiungere la destinazione scelta, ma non tiene conto di altri fattori come il traffico o il consumo di carburante della specifica tipologia di veicolo in uso. Il percorso a minor consumo di carburante può essere calcolato tenendo conto anche di questi aspetti, e questo permetterebbe d'introdurre un sistema per poter stimare con maggiore precisione il consumo di carburante di un tragitto [35], che dovrebbe tenere in considerazione la conformazione della strada, il traffico lungo il percorso, e la tipologia di veicolo in uso.

Bibliografia

[1] Cristiano Gelli. «Studio e realizzazione di un sistema per l'analisi What-If in ambito Smart City». Tesi magistrale. Università degli studi di Firenze, 2019.

[2] Marshall Fisher. «Vehicle routing». In: *Network Routing*. Handbooks in Operations Research and Management Science. Elsevier, 1995.

[3] Google. *Traveling Salesperson Problem*. URL: https://developers.google.com/optimization/routing/tsp.

[4] Google. *Vehicle Routing Problem*. URL: https://developers.google.com/optimization/routing/vrp.

[5] Thomas H. Cormen e Livio Colussi. *Introduzione Agli Algoritmi e Strutture Dati*. McGraw-Hill, 2010.

[6] *Treccani*. URL: https://www.treccani.it/vocabolario/vincolo/.

[7] Rashmi Choudhary, Siftee Ratra e Amit Agarwal. «Multimodal routing framework for urban environments considering real-time air quality and congestion». In: *Atmospheric Pollution Research* (2022).

[8] Engelmann Martin, Schulze Paul e Wittmann Jochen. «Emission-Based Routing Using the GraphHopper API and OpenStreetMap». In: *Advances and New Trends in Environmental Informatics*. Springer International Publishing, 2020.

[9] Oregon State University, Portland State University e University of Idaho. *Greenshield's Model*. URL: https://www.webpages.uidaho.edu/niatt_labmanual/chapters/trafficflowtheory/theoryandconcepts/GreenshieldsModel.htm.

[10] LH Immers e S Logghe. «Traffic flow theory». In: *Faculty of Engineering, Department of Civil Engineering, Section Traffic and Infrastructure, Kasteelpark Arenberg* (2002).

[11] Stefano Bilotta e Paolo Nesi. «Traffic flow reconstruction by solving indeterminacy on traffic distribution at junctions». In: *Future Generation Computer Systems* (2021). URL: https://www.sciencedirect.com/science/article/pii/S0167739X20308359.

[12] Snap4City. *Snap4City Website*. URL: https://www.snap4city.org/.

[13] Matthew Towers. *Bidirectional Dijkstra*. URL: https://www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html.

[14] Rashmi Choudhary, Siftee Ratra e Amit Agarwal. «Fusing real-time congestion and air pollution in a multi-modal routing engine». In: 2022.

[15] Sachit Mahajan et al. «CAR: The Clean Air Routing Algorithm for Path Navigation With Minimal PM2.5 Exposure on the Move». In: *IEEE Access* (2019).

[16] OpenStreetMap. *OpenStreetMap Website*. URL: https://www.openstreetmap.org/.

[17] OpenStreetMap. *Node - OpenStreetMap Wiki*. URL: https://wiki.openstreetmap.org/wiki/Node.

[18] OpenStreetMap. *Way - OpenStreetMap Wiki*. URL: https://wiki.openstreetmap.org/wiki/Way.

[19] OpenStreetMap. *Area - OpenStreetMap Wiki*. URL: https://wiki.openstreetmap.org/wiki/Area.

[20] OpenStreetMap. *Relation - OpenStreetMap Wiki*. URL: https://wiki.openstreetmap.org/wiki/Relation.

[21] Overpass API. *Export OSM maps*. URL: https://overpass-api.de/.

[22] BBBike. *BBBike Extract service*. URL: https://extract.bbbike.org/.

[23] GraphHopper. *GraphHopper GitHub*. URL: https://github.com/graphhopper/graphhopper/.

[24] GraphHopper. *GraphHopper Website*. URL: https://www.graphhopper.com/.

[25] Apache Software Foundation. *Apache Tomcat*. URL: https://tomcat.apache.org/.

[26] Eclipse Foundation. *Jersey*. URL: https://jersey.github.io/.

[27] Google. *Maps*. URL: https://www.google.com/maps.

[28] Google. *Google Maps Traffic map*. URL: https://www.google.co.uk/maps/@43.78,11.23,13z/data=!5m1!1e1?entry=ttu.

[29] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[30] Apache Software Foundation. *Apache Maven*. URL: https://maven.apache.org/.

[31] *Project repository*. URL: https://github.com/RonPlusSign/whatif-router.

[32] Pierfrancesco Bellini et al. «Km4City ontology building vs data harvesting and cleaning for smart-city services». In: *Journal of Visual Languages & Computing* (2014). URL: https://www.sciencedirect.com/science/article/pii/S1045926X14001165.

[33] Snap4City. *Dashboard Builder*. URL: https://github.com/disit/dashboard-builder/.

[34] GraphHopper. *GraphHopper Custom Models*. URL: https://github.com/graphhopper/graphhopper/blob/master/docs/core/custom-models.md.

[35] Kanok Boriboonsomsin et al. «Eco-Routing Navigation System Based on Multisource Historical and Real-Time Traffic Information». In: *IEEE Transactions on Intelligent Transportation Systems* (2012).