

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI MATEMATICA

*Corso di Laurea Magistrale in Informatica*

**PATHS: UN SERVIZIO LOCATION-BASED PER  
PERCORSI ALTERNATIVI**

*Laureando*

**Tobia Zorzan**

*Relatore*

**Prof. Claudio Palazzi**

*Matricola*

**606518**

MCCXXII

---

ANNO ACCADEMICO 2015/2016



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Contesto e lavori correlati</b>	<b>3</b>
2.1	Web <sup>2</sup> e Opportunistic Sensing . . . . .	3
2.2	Gestione dei dati di movimento . . . . .	8
2.2.1	Ricostruzione delle traiettorie . . . . .	10
2.3	Versione precedente e lavori correlati . . . . .	12
<b>3</b>	<b>PathS Client</b>	<b>15</b>
3.1	Requisiti . . . . .	15
3.2	Tecnologie . . . . .	17
3.3	Stato attuale . . . . .	19
<b>4</b>	<b>PathS Server</b>	<b>21</b>
4.1	Requisiti . . . . .	21
4.2	Componenti . . . . .	24
4.2.1	Ricezione dei campioni . . . . .	24
4.2.2	Elaborazione dei campioni . . . . .	24
4.2.3	Calcolo Percorsi . . . . .	25
4.2.4	Accesso da Browser . . . . .	25
4.3	Tecnologie . . . . .	25
<b>5</b>	<b>Ricezione dei campioni</b>	<b>27</b>
5.1	API . . . . .	27
5.2	Implementazione . . . . .	28
5.3	Esempi . . . . .	30
<b>6</b>	<b>Elaborazione dei campioni</b>	<b>33</b>
6.1	Servizi di Cartografia . . . . .	33
6.1.1	Google Maps . . . . .	33
6.1.2	OpenStreetMap . . . . .	35

6.2	Persistenza ed elaborazione dei dati GIS . . . . .	38
6.3	Algoritmo di Map Matching . . . . .	39
6.3.1	ST-Matching . . . . .	41
6.3.2	Pre-elaborazione del percorso - step 1 . . . . .	42
6.3.3	Identificazione dei candidati - step 2 . . . . .	43
6.3.4	Valutazione e selezione dei candidati - step 3 . . . . .	44
<b>7</b>	<b>Routing</b>	<b>49</b>
7.1	Implementazione . . . . .	49
7.2	Percorsi calcolati . . . . .	51
7.2.1	Shortest Path . . . . .	51
7.2.2	Percorsi con <i>label</i> . . . . .	53
7.2.3	Servizio Map Quest . . . . .	56
<b>8</b>	<b>Conclusione</b>	<b>59</b>
8.1	Risultati . . . . .	59
8.2	Miglioramenti ed Evoluzioni . . . . .	60
<b>Appendice</b>		<b>63</b>
<b>A</b>	<b>Manuale installazione</b>	<b>65</b>
A.1	Ambiente . . . . .	65
A.2	Dipendenze . . . . .	65
A.2.1	Database . . . . .	65
A.2.2	Java . . . . .	66
A.2.3	Repository . . . . .	66
A.3	Configurazione . . . . .	66
A.4	Avvio . . . . .	67
A.5	Note . . . . .	67
<b>Bibliografia</b>		<b>69</b>

## Sommario

I recenti sviluppi dei dispositivi *smartphone*, in particolare l'inclusione di sensori sempre più evoluti, ha portato alla definizione di un nuovo paradigma denominato *Web<sup>2</sup>* (*Web Squared*). La diffusione capillare di questi strumenti consente di creare una rete di sensori distribuiti in grado di raccogliere una ingente mole di dati. I dati stessi possono essere quindi elaborati e fornire il supporto a nuovi servizi che non si basano sulle sole informazioni *statiche* ma su un contesto *dinamico* modificato dall'interazione degli utenti. Il progetto *PathS* vuole essere un prototipo che segue questo paradigma. La sua componente *client* raccoglie informazioni aggiuntive sui percorsi pedonali percorsi dagli utenti. La parte *server* elabora le informazioni raccolte e mantiene una base di dati in grado di supportare servizi di interrogazione che adottano criteri aggiuntivi a quelli tradizionali. Questa tesi espone quali sono stati i principi e le valutazioni eseguite nella realizzazione del componente *PathS Server*, i risultati raggiunti e le possibili aree di miglioramento individuate.



# Capitolo 1

## Introduzione

La diffusione dei dispositivi *smartphone* e l'accesso ad internet in mobilità sono ormai un dato di fatto nel panorama delle telecomunicazioni. La base di utenza si è allargata a dismisura, coprendo quasi tutte le fasce d'età così come l'infrastruttura tecnologica consente ormai di utilizzare questi mezzi in svariate situazioni permettendo di restare sempre “connessi”.

L'utilizzo delle applicazioni mobili è ormai parte delle nostre attività quotidiane, sia ricreative che in ambito lavorativo. Le esigenze e le sfide per questo tipo di strumenti sono sempre più ambiziose: vogliamo applicazioni *smart*, che ci rendano facili e accessibili le informazioni più complesse e che offrano una esperienza d'uso personalizzata e tagliata su misura sulle nostre richieste.

Il progetto *PathS* si sviluppa in questo contesto, proponendo un'evoluzione dei sistemi di navigazione pedonale. L'idea di fondo è che la proposta del tragitto più breve verso una destinazione non sia più l'unica informazione da fornire agli utenti, ma grazie ai recenti sviluppi tecnologici può essere integrata con un'insieme di elementi di contorno che possono rendere più coinvolgente e mirata l'esperienza di utilizzo.

Il progetto è stato coordinato nelle attività dal Prof. Claudio Palazzi e dalla Prof.ssa Ombretta Gaggi dell'Università degli Studi di Padova. Nelle attività di analisi e approfondimento hanno collaborato i dottorandi Matteo Ciman e Armir Bujari, mentre le attività riguardanti l'applicazione *client* sono state svolte dallo studente Stefano Tombolini.

In questo elaborato saranno presentati inizialmente il contesto e un insieme di progetti che per problematica affrontata o approccio utilizzato risultano affini e di interesse. Saranno introdotti alcuni riferimenti di letteratura in particolare riguardo la tematica dei *Moving Object Databases*.

Il capitolo 3 fornirà un riassunto delle caratteristiche e delle funzioni offerte della componente *client*, le quali risultano fondamentali per una visione

complessiva del progetto.

I capitoli successivi documentano le attività di analisi e di implementazione del *server PathS* nelle sue due macro-funzioni: ricezione e analisi dei campioni e calcolo dei percorsi pedonali.

L'ultimo capitolo fornirà un esempio dei risultati raggiunti e proporrà alcune aree di miglioramento così come sono state individuate durante lo svolgimento delle attività.

# Capitolo 2

## Contesto e lavori correlati

In questo capitolo si presenta il contesto in cui si è sviluppato il progetto *PathS* ed alcuni concetti fondamentali collegati all’ambito di applicazione. Si citano inoltre progetti con finalità simili e approcci ritenuti interessanti punti di riferimento.

### 2.1 Web<sup>2</sup> e Opportunistic Sensing

Una rete di sensori o *Sensor Network* è composta di elementi autonomi e distribuiti atti al monitoraggio di determinati parametri ambientali. Ciascun nodo ha capacità autonome di calcolo, percezione e misurazione dell’ambiente circostante, comunicazione ed eventualmente di mobilità. Molte ricerche si sono focalizzate in questo ambito, in particolare con lo scopo di definire un sistema affidabile e decentralizzato per la comunicazione e il supporto a servizi in questo scenario. Raramente questo tipo di soluzioni sono andate oltre l’ambito della sperimentazione fino all’applicazione su larga scala.

Il recente sviluppo tecnologico e la diffusione capillare dei dispositivi *smartphone* ha radicalmente modificato la situazione, aprendo la possibilità ad importanti evoluzioni. Di fatto anche un dispositivo base ha tutte le caratteristiche necessarie, essendo dotato di un sistema GPS, moduli di connessione alla rete dati e numerosi sensori tra cui quello di prossimità, luminosità, microfono, fotocamera e accelerometri. Considerata la buona disponibilità di si possono coprire ampie zone geografiche con un buon numero di dispositivi, offrendo sia l’occasione di applicare le ricerche sviluppate nell’ambito delle *sensor networks*, sia di esplorare scenari del tutto nuovi. Ciò che rende ancora più interessante l’applicazione di una rete pervasiva di dispositivi, non è solo la possibilità di migliorare i servizi per gli utilizzatori

diretti, ma la possibilità di studiare scenari del tutto nuovi a giovamento dell'intera società.

Pensiamo a quella che è già stata una grande evoluzione del mondo di internet, ovvero il *Web 2.0*. Il suo avvento è fatto coincidere con la diffusione dei social networks, alla cui base c'è l'elevato livello di interazione tra il sito Web e gli utenti stessi. In ogni caso è richiesta una interazione diretta dell'utilizzatore, il quale consapevolmente contribuisce ad aumentare la base delle informazioni.

Lo sviluppo di alcune tecnologie specifiche (tra cui *AJAX*) ha permesso lo scambio di dati in background fra Web browser e server consentendo l'aggiornamento dinamico della pagina senza esplicito ricaricamento da parte dell'utente.

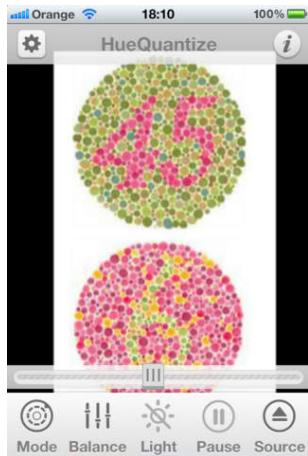
Fondamentalmente si è passati da un Web statico a uno di tipo dinamico e sociale in cui l'utente non è più solo un lettore e un consumatore passivo di contenuti, ma è il principale creatore di essi.

La diffusione capillare di questi dispositivi con sensori avanzati, apre la strada al paradigma *Web<sup>2</sup>* in cui la quantità di dati, raccolta anche senza l'esperienza utente diretta, non aumenta in modo lineare ma letteralmente esplode con andamento esponenziale. Diventa quindi di fondamentale importanza le modalità con cui si raccoglie questa mole di dati e i principi di design con cui si imposta lo sviluppo di applicazioni in questo contesto.

Una buona analisi tecnica e un'architettura di riferimento per lo sviluppo di servizi *Web<sup>2</sup>* basati su reti di sensori *smartphone* è fornita da Calma, Palazzi e Bujari in [1]. Nell'articolo è introdotto e sviluppato il concetto di *opportunistic sensing* il quale è ritornato molto utile anche nel progetto *PathS*. Gli autori espongono due criteri fondamentali secondo cui categorizzare le applicazioni in ambito *Web Squared*: il primo riguarda la scala di rilevamento, il secondo è il grado di coinvolgimento degli utenti. Considerando la scala di rilevamento possiamo distinguere una applicazione in:

- **personale**: progettata per utenti singoli e i risultati vengono visualizzati solo all'utente stesso, senza la loro condivisione. Un esempio di queste può essere *Dankam*, applicazione che aiuta nell'identificazione dei colori le persone affette da daltonismo;
- **di gruppo**: ad esempio *Nike+* o *Runtastic*, applicazioni in cui un gruppo di persone con interessi affini condivide informazioni spesso sfruttando la connessione ai popolari social network;
- **di comunità**: i dati sono raccolti da una vasta gamma di persone e i risultati condivisi pubblicamente per renderli disponibili a tutti gli utenti.

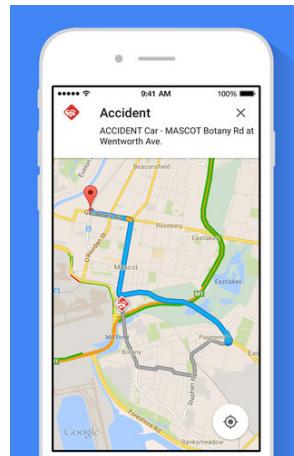
Un'esempio di applicazione è Google Maps e la relativa segnalazione dei tratti trafficati.



(a) Dankam



(b) Nike+



(c) Google Maps

Valutando invece il coinvolgimento con l'utente, le applicazioni di questa categoria si possono suddividere in:

- **partecipative**: ovvero la raccolta dei dati avviene tramite il coinvolgimento esplicito e diretto dell'utente. Ci si affida quindi all'entusiasmo e alla relazione diretta con gli utilizzatori i quali volontariamente eseguono l'applicazione e raccolgono i dati necessari. Solitamente le informazioni che derivano da questa tipologia sono molto accurate e ben distribuite anche per la possibilità di una opportuna pianificazione. La controparte è che spesso non è facile ottenere una sufficiente base di copertura se non si dispone del bacino di utenza e delle risorse necessarie.
- **opportunistiche**: in cui l'acquisizione delle informazioni avviene tramite i sensori del dispositivo in determinate situazioni di utilizzo, anche se in quel momento non è l'attività principale. La raccolta avviene quindi in *background* informando, ma non richiedendo necessariamente l'attenzione, dell'utente. In qualche modo si “distrae” l'utilizzatore con una attività primaria dall'alto livello di gradimento e nel frattempo si approfitta della situazione per catturare le informazioni necessarie. L'aspetto negativo di questo approccio è principalmente tecnico, dato che la sovrapposizione di più attività comporta un maggiore dispendio di risorse del dispositivo (*cpu*, batteria, rete, etc.) e al tempo stesso gli algoritmi che devono interpretare i dati provenienti dai sensori e

identificare le situazioni idonee sono particolarmente complessi. Questo tipo di approccio risulta particolarmente utile per le applicazioni *di comunità* che, pur raccogliendo pochi campioni, possono contare su un'ampio bacino di utenza.

Nello sviluppo del progetto *PathS* ed in particolare per la progettazione della componente *client* si è pensato di sfruttare entrambi questi aspetti. Tramite il coinvolgimento diretto di alcuni studenti e il coordinamento del Prof. Palazzi e dei suoi collaboratori, è stata organizzata una campagna di campionamento massivo della zona universitaria e dei dintorni. Questo ha consentito sia di raccogliere numerosi *feedback* riguardo il funzionamento del sistema che di raccogliere una consistente base di informazioni per l'area di riferimento. Per integrare e ampliare questa base di partenza, si è pensato ad un approccio di tipo opportunistico presentato in dettaglio nel capitolo successivo.

Un altro aspetto fondamentale proposto nello stesso articolo è il principio di architettura software secondo cui impostare un servizio basato sul paradigma *Web*<sup>2</sup>. Concepire una applicazione mobile di questo tipo non è banale, in quanto affiancare le due attività di raccolta intensiva di dati dai sensori e la condivisione di essi può portare a diversi problemi. Uno degli aspetti fondamentali riguarda la gestione delle risorse (spesso limitate) del dispositivo. Un utilizzo troppo intenso delle componenti relative ai sensori così come una trasmissione dati frequente può portare ad esaurire in breve tempo la batteria, o in alcuni casi a sovraccaricare l'unità di calcolo del dispositivo, causando un disservizio nelle funzionalità principali (come ricevere le chiamate). Una delle soluzioni proposte è prima di tutto individuare e separare le componenti sulla base del ruolo che devono svolgere per raggiungere il risultato atteso. Nel caso delle applicazioni *Web*<sup>2</sup> è individuato un sistema architettonico basato su tre *layer* di competenza che sono:

- **sensing layer:** che necessariamente risiede nel dispositivo e si occupa della raccolta dei dati grezzi dai sensori;
- **learning layer:** che svolge il ruolo di processare i dati raccolti e derivare da essi le informazioni necessarie;
- **release layer:** che si occupa di fornire i risultati ottenuti all'utente.

Le problematiche che riguardano il primo *layer* riguardano principalmente l'integrazione con gli ambienti di sviluppo (*SDK*) e le modalità di accesso alle componenti del dispositivo. Spesso gli strumenti messi a disposizione degli sviluppatori non sono così a basso livello da consentire un'utilizzo ottimale. E' necessario trovare dei *workaround* specifici, piattaforma per piattaforma,

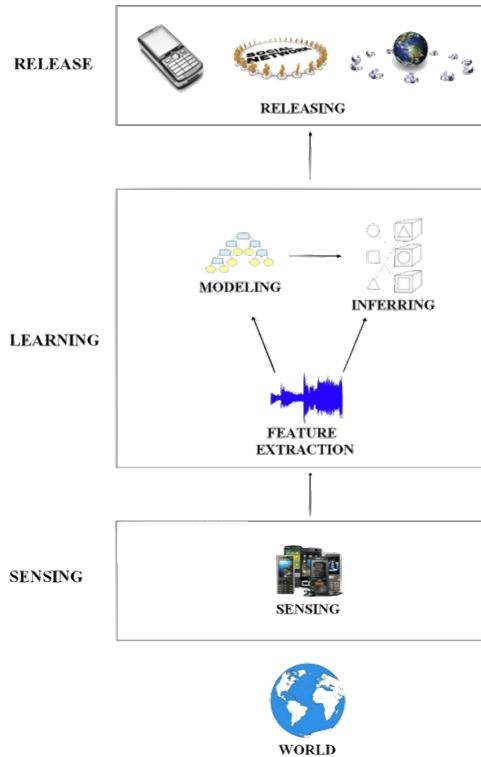


Figura 2.2: Architettura a *layer* di una applicazione *Web<sup>2</sup>*

che portino al giusto compromesso tra la frequenza di campionamento, la precisione dei dati raccolti e l'utilizzo delle risorse.

Per quanto riguarda il *learning layer* si possono invece trovare le soluzioni più diverse. Non è indispensabile che questa logica applicativa risieda necessariamente nel dispositivo ma è possibile delegare l'attività ad un sistema esterno (es. server cloud) in cui la disponibilità di risorse è maggiore ed è possibile adottare algoritmi più complessi. In alcuni casi si suggerisce anche un approccio “misto” in cui i dati vengono pre-processati dal dispositivo cercando di estrarre delle *feature* che vengono quindi comunicate all'esterno ed elaborate. Il *release layer* si occupa di mostrare il risultato all'utente finale e, a seconda del tipo di applicazione, potrebbe avere come destinazione sia un singolo cliente così come una vasta comunità. Per questo motivo anche in questo caso le soluzioni proposte possono riguardare sistemi esterni o ad esempio il collegamento a vaste reti di distribuzione tipo *social-network*.

Nel progetto *PathS* ritroviamo tutte queste problematiche e le soluzioni adottate seguono in molti casi i principi generali qui presentati. Ad esempio le modalità di campionamento GPS eseguite dal *client* sono state tarate al fine di ottenere risultati accettabili minimizzando l'uso delle risorse. Dal

punto di vista dell'architettura del sistema si è deciso di elaborare tutti i dati nel sistema *server* esterno, dove i dati raccolti vengono processati e utilizzati per derivare le informazioni di sintesi. Infine agli utenti finali si comunicano i risultati sia in forma diretta (con opportuno protocollo di comunicazione con il client) che in forma più estesa tramite l'accesso a dei servizi web.

## 2.2 Gestione dei dati di movimento

Un'altra tematica affrontata in quanto affine all'ambito del progetto è quella dei *Mobility Data*. Alcuni esempi di *mobility data* sono forniti in [6] da Pelekis e Theodoridis e possono essere:

- i dati provenienti da una conversazione telefonica cellulare (la compagnia telefonica ha informazioni aggiuntive in merito al posizionamento del dispositivo);
- i dati provenienti da un dispositivo GPS durante una determinata attività (la combinazione tra la posizione e il momento in cui è rilevata);
- i dati scambiati tra i veicoli di una *vehicular ad hoc network* (*VANET*);
- i dati raccolti da un sistema di *radio-frequency identification* (*RFID*).

Come vediamo il concetto di *mobility data* è molto vario e può essere applicato a numerosi contesti. Più in generale si considera riguardante questa tematica tutti i casi in cui si combinano assieme i dati dell'asse temporale con i dati spaziali.

L'evoluzione del movimento di un oggetto nel tempo è diventato recentemente una importante tematica di ricerca. La gestione di questi dati in domini separati è ben consolidata: da una parte gli *Spatial Databases* sono in grado di gestire in modo efficiente i dati posizionali e le operazioni di interrogazione su di essi, così come dall'altra parte i *Temporal Databases* consentono di manipolare e accedere a dati temporali. Tuttavia la combinazione simultanea di questi due insiemi può aprire ad importanti prospettive di applicazione. Prendiamo ad esempio l'analisi del traffico in una rete di trasporto cittadina. È possibile (ed è già stato realizzato in diversi studi) dotare un numero consistente di veicoli circolanti di un dispositivo GPS. Il dispositivo registra le informazioni in merito alla posizione con una frequenza sufficientemente dettagliata (ad esempio 0.2 Hz, una campionamento ogni 5 secondi). La raccolta di queste informazioni può portare ad un *set* di dati che se interpretato correttamente può fornire un supporto a richieste del tipo:

- **analisi del traffico:** quanti veicoli affollano un determinato tratto in un momento specifico? Qual'è il tempo di attesa medio ad un semaforo oppure funziona correttamente l'effetto “onda verde”?
- **servizi *location-aware*:** qual è l'attività commerciale più vicina alla mia posizione attuale o al percorso che ho intenzione di seguire? Quali amici *facebook* sono in prossimità della città che sto visitando?

Quelli riportati sono solo alcuni esempi, ma più in generale lo studio di *mobility data* e di *database* di traiettorie può supportare lo sviluppo di *Location Based Services* ed applicazioni *Location-* e *Mobility-Aware*. Per ***Location Based Service*** si intendono tutti quei servizi che forniscono informazioni ai propri utenti sulla base della posizione geografica corrente. Se il servizio è caratterizzato da un altro grado di interazione tra gli utenti stessi e la formazione di una sorta di rete fra gli stessi, allora si può parlare di ***Location-based Service Networking***. Una tassonomia di questi servizi è proposta in tabella 2.1 e deriva dal grado di mobilità (stazionario o mobile) degli attori coinvolti ovvero l'utente che interroga il servizio e gli oggetti interrogati a database.

Reference Object		<i>Stationary</i>	<i>Mobile</i>
Database Objects			
<i>Stationary</i>		routing	guide-me
<i>Mobile</i>		find-me	get-together

Tabella 2.1: Tassonomia di applicazioni *location-aware*

Come vediamo l'ambito del progetto *PathS* ha molti aspetti in comune con quelli trattati nel *mobility data management*. Tuttavia le problematiche che si intendere risolvere (almeno nelle prime fasi del progetto) riguardano la prima classe di problemi (*<stationary, stationary>*) relativamente più conosciuta e trattata in letteratura.

Modellando la rete di trasporto (*transportation network*) come un grafo  $G = (N, E)$  composto da nodi  $N$  e archi  $E$ , l'operazione di *routing* consiste tipicamente nel cercare un percorso ottimale che porti dalla sorgente  $S$  alla destinazione  $T$  dove  $S, T \in N$ . Tecnicamente per queste tematiche sono disponibili diversi algoritmi, principalmente basati sulla soluzione del problema di flusso *Shortest Path*. Un adeguamento del calcolo dei pesi utilizzati negli archi della rete, consente di ottenere diverse modalità di navigazione ad esempio la più veloce, la più breve o, come è di recente interesse, la più ecologica.

Dato che l'attenzione si è concentrata nel trattare le questioni di *routing*, nello sviluppo del progetto *PathS* ci si è allontanati da quelle che sono le problematiche e gli strumenti specifici nell'ambito del *mobility data management*, seguendo piuttosto altri spunti da cui derivare l'implementazione. Ciò non toglie che l'ambito di applicazione è molto coerente con questo tema e l'adozione di queste tecniche può tornare molto utile nelle possibili evoluzioni del sistema, in particolare se si intende utilizzare i dati per successive analisi o interrogazioni che vadano oltre la tematica dei percorsi.

### 2.2.1 Ricostruzione delle traiettorie

L'ampia diffusione di dispositivi dotati di sistema GPS combinata con lo sviluppo di adeguate tecniche di memorizzazione, processamento ed interrogazione dei dati ha portato alla produzioni di immense quantità di dati *location-aware*. Tuttavia la derivazione di informazioni significative da questi dati grezzi non è per nulla banale. Nella trasformazione di questi dati si incontrano diverse problematiche, tra cui la gestione dei segnali di rumore o non accurati, la semplificazione dei *set* di dati ed in particolare l'identificazione delle traiettorie come sequenza ordinata di posizioni campionate. La ricostruzione di una traiettoria a partire da un insieme di dati grezzi è una operazione fondamentale che si rende necessaria prima di qualsiasi altra elaborazione o analisi dei dati.

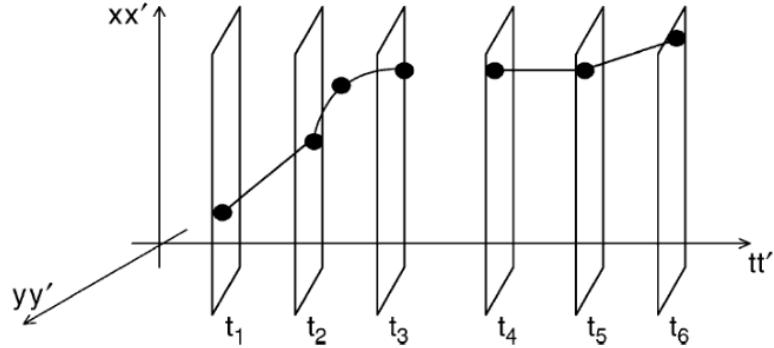


Figura 2.3: *Sliced representation* di un oggetto in movimento.

Come presentato in [6, capitolo 3.1], una definizione teorica del movimento di un oggetto può essere quella di una funzione dallo spazio temporale  $I \subseteq \mathbb{R}$  allo spazio geografico  $S \subseteq \mathbb{R}^2$ :

$$I \subseteq \mathbb{R} \rightarrow S \subseteq \mathbb{R}^2 : t \rightarrow l(t)$$

dove con  $l(t)$  si intende la posizione in cui si trova l'oggetto all'istante  $t$ . In questo modo la traiettoria effettiva seguita da un oggetto può essere definita come:

$$T_{act} = \{t, l(t) | t \in \mathbb{I}\} \subset \mathbb{R}^2 \times \mathbb{R}$$

Questo rappresenta una curva continua nella realtà, tuttavia la forma che ci si trova a rappresentare nei calcolatori è la sua versione finita, definita come sequenza di coppie di valori spazio-tempo:

$$T = \{< p_1, t_1 >, < p_2, t_2 >, \dots, < p_n, t_n >\}$$

dove  $p_i \in \mathbb{R}^2$ ,  $t_i \in \mathbb{R}$ ,  $1 \leq i \leq n$  e  $t_1 < t_2 < \dots < t_n$ . Il risultato è che una traiettoria può essere rappresentata con un modello che si definisce *sliced representation* come quello in figura 2.3. Questo modello decomponne lo sviluppo temporale in frammenti definiti *slice*, tali per cui questa evoluzione può essere descritta da una qualche funzione semplice, ad esempio l'interpolazione lineare.

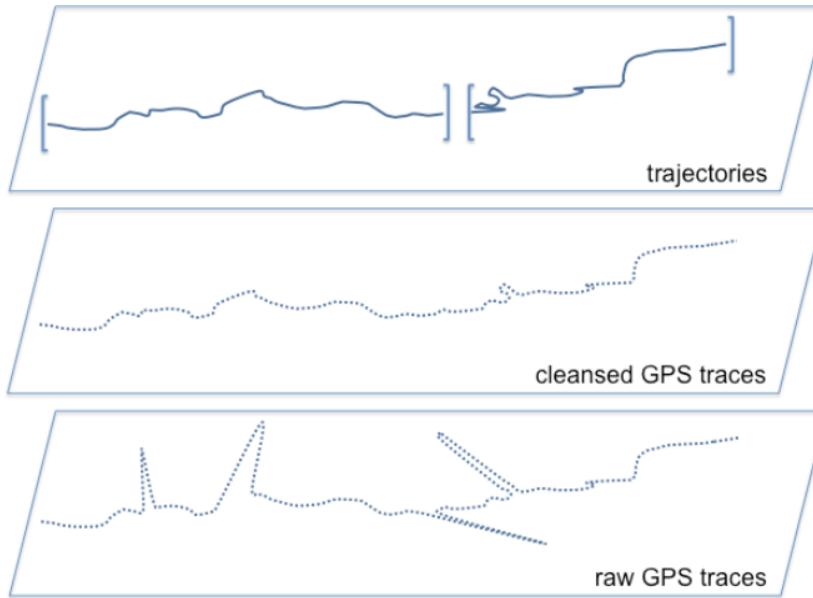


Figura 2.4: Il processo di *trajectory reconstruction* da dati grezzi GPS.

In generale il problema di *trajectory reconstruction* si decompone di due operazioni principali:

- *data cleansing*: è la fase in cui si ripuliscono i dati grezzi dai segnali di rumore o dagli eventuali *outlier*, è possibile eseguire operazioni di arrotondamento o altre elaborazioni;

- *trajectory identification*: ha l’obiettivo di interpolare la sequenza di posizioni raccolte al fine di approssimare il movimento continuo dell’oggetto.

In particolare per la seconda operazione le modalità sono tutt’altro che semplici e delineate. Sono disponibili diversi approcci, alcuni che si basano sulle caratteristiche geometriche della rappresentazione dei dati, altri si basano su sistemi probabilistici, altri ancora sono sistemi ibridi.

Anche in *PathS* si incontra la problematica fondamentale di ricostruire le traiettorie individuate dai campionamenti. L’approccio che si è valutato più opportuno (considerate le caratteristiche dei dati sorgente) è stato quello di utilizzare un algoritmo di *map matching*. Il caso di *trajectory reconstruction* affrontato è stato quello di ricondurre ciascun campionamento ad una posizione corrispondente nella rete di trasporto. Utilizzando la rappresentazione di grafo  $G = (V, E)$  composto di archi e vertici, per ciascuna posizione  $\langle p_i, t_i \rangle$  della traiettoria che non appartiene alla rete, si è individuata la corrispondente coppia  $\langle p'_i, t'_i \rangle$  in cui  $p'_i \in e_j$  e  $t'_i$  è simile ma non necessariamente corrisponde a  $t_i$ .

## 2.3 Versione precedente e lavori correlati

L’idea di sviluppo del progetto non è del tutto nuova, ma prende spunto da un lavoro precedente svolto sempre all’interno dell’Università di Padova dallo studente Lorenzo Teodori seguito dal Prof. Palazzi. Il progetto chiamato *Path2.0* voleva essere un sistema partecipativo per fornire indicazioni di navigazione accessibile per utenti con disabilità. L’idea di base è quella sviluppata in *PathS*, ovvero raccogliere informazioni da un insieme di utenti per poi riutilizzarle nel suggerire dei percorsi pedonali. Gli obiettivi in cui questo progetto estende e sviluppa l’idea iniziale sono i seguenti:

- estendere il concetto di informazione aggiuntiva sul percorso, non trattare solo l’accessibilità (valori 1, 0) ma delle *label* più generiche;
- implementare soluzioni che rispondessero con percorsi diversi dal semplice percorso accessibile e percorso breve;
- fornire una base di dati elaborati che consenta non solo il *routing*, ma anche l’analisi o l’applicazione ad altri scenari in chiave sociale.

Dal punto di vista tecnologico, si è preferito fare tesoro dell’esperienza precedente e strutturare un sistema completamente nuovo per le seguenti motivazioni:

- il sistema SDK utilizzato nello sviluppo dell'applicazione *mobile* era molto datato, richiedeva un profondo aggiornamento e la revisione di alcune modalità di sviluppo;
- tutta la soluzione era molto legata e del tutto dipendente dai servizi API di Google Maps (anch'essi da aggiornare);
- il progetto di partenza non presentava dei caratteri architetturali sufficientemente delineati da consentirne l'utilizzo come base, in particolare per le esigenze di espansione ed evoluzione ad altri contesti;

Un progetto analogo a *PathS* nel quale si utilizza però il microfono del dispositivo è Crowd++ [8]. Lo scopo dell'applicazione sviluppata è di individuare dati aggiuntive riguardo l'attività sociale dell'utente e l'interazione tra *speaker* e pubblico in un evento. I suoni registrati dal microfono sono utilizzati per individuare il numero di parole pronunciate o il numero di persone che stanno parlando, cercando quindi di derivare le informazioni necessarie. Altro esempio da cui si è tratto spunto è il progetto PartecipAct [2]. Sviluppato recentemente presso l'Università di Bologna, anch'esso ha come obiettivo quello di studiare il potenziale inesplorato della collaborazione tra utenti, sfruttando gli smartphone come strumento di interazione e interconnessione. Coinvolgendo studenti e volontari in un esperimento di raccolta dati, si sono raccolte informazioni sugli utenti e sul contesto in cui si trovano. Tutti i dati vengono memorizzati in un database online cercando di rispondere a domande del tipo: “dove passano la maggior parte del tempo i nostri utenti?”. Come vediamo l'approccio è molto simile a quello di *PathS*, nonostante le modalità di svolgimento e il fine ultimo siano diversi.



# Capitolo 3

## PathS Client

Tra le due componenti in cui si articola il progetto, la prima sviluppata è stata la componente *client*. Questa parte consiste sostanzialmente di una applicazione per dispositivi *smartphone* Android, completamente riscritta a partire da alcuni sviluppi precedenti per poter assolvere alle nuove funzioni specifiche di *PathS*. Si riassumono i requisiti che sono stati identificati e i criteri principali che hanno determinato lo sviluppo del software.

### 3.1 Requisiti

Il client mobile *PathS* presenta tutte le caratteristiche di una applicazione sviluppata secondo il paradigma *Web*<sup>2</sup>. Le funzioni principali che deve svolgere possono essere riassunte in:

- raccogliere dei valori riguardante l’ambiente circostante tramite i sensori di luminosità e microfono;
- tenere traccia degli spostamenti effettuati e abbinarli ai relativi campioni;
- offrire le indicazioni di percorso secondo la destinazione e le modalità richieste dall’utente;
- cercare di coinvolgere il più possibile l’utente incentivando l’uso dell’applicazione.

Alcuni di questi requisiti possono essere identificati come *funzionali* e sono indispensabili per gli scopi generali del progetto. Le operazioni di campionamento tramite i sensori ambientali e il tracciamento della posizione tramite GPS sono gli elementi fondamentali richiesti per poter disporre dei

dati necessari alle successive elaborazioni. Tuttavia nessun utente finale sarebbe motivato nell'eseguire queste operazioni se l'applicazione non fornisse una qualche altra utilità. Un gruppo di utenti volontari può essere selezionato per alcune campagne di raccolta, tuttavia per motivare un utilizzatore occasionale era necessario adottare altre strategie. Per questo sono stati identificati gli altri due requisiti, così da motivare l'utilizzo dell'applicazione stessa. Si è pensato che fornendo direttamente il servizio di *routing* dallo smartphone, si potevano sfruttare i risultati del servizio ma contemporaneamente contribuire allo stesso raccogliendo ulteriori informazioni durante il percorso. Per rendere ottimali i dati raccolti durante l'uso, era inoltre necessario ideare uno stratagemma affinchè il dispositivo fosse in una condizione ideale per il campionamento. La soluzione è venuta dalla proposta di adottare delle funzioni di *Augmented Reality* (Realtà Aumentata). In questo modo non solo si sarebbe ottenuto un maggiore coinvolgimento dell'utente finale, ma inconsapevolmente l'utilizzatore stesso avrebbe mantenuto lo *smartphone* fuori dalla tasca in condizioni perfette per le misurazioni tramite i sensori. Seguendo queste indicazioni quindi, si è strutturata una applicazione *Android* composta principalmente di 3 moduli:

- **Mappa:** E' la schermata con la quale l'utente può selezionare una destinazione e visualizzare il percorso suggerito. In questa sezione l'applicazione comunica con i servizi Google per la selezione dei luoghi e la ricerca della destinazione, mentre interroga la componente *server* di *PathS* per richiedere i percorsi da suggerire. Il risultato è visualizzato in una vista a cartina stile Google Maps.
- **Navigatore:** E' la modalità con la quale si guida l'utente durante il percorso pedonale: si forniscono le indicazioni di svolta ed altre informazioni aggiuntive presentandole in realtà aumentata. L'utente può quindi mantenere il dispositivo in posizione verticale inquadrando l'orizzonte con la fotocamera e visualizzare in modo "sovraposto" le informazioni di cui necessita. Durante questa modalità sono eseguiti in *background* i campionamenti di rumorosità e luminosità che saranno successivamente inviati al server.
- **Impostazioni:** In questa parte dell'applicazione è possibile configurare alcuni parametri di funzionamento e di visualizzazione.

I tre moduli consentono all'applicazione di svolgere tutte le funzioni identificate dai requisiti.

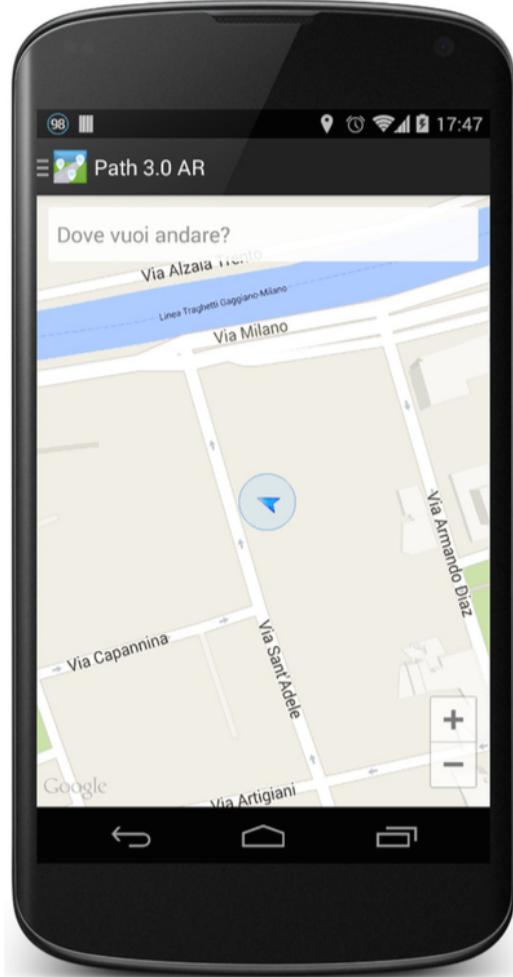


Figura 3.1: Modalità *Mappa* dell'applicazione mobile PathS.

## 3.2 Tecnologie

Considerando gli sviluppi precedenti e la disponibilità di dispositivi, la piattaforma scelta per lo sviluppo del client è stato il sistema *Android*. Questo sistema operativo è ormai largamente diffuso sia nei dispositivi tablet che smartphone anche di fascia economica, risultando un ottimo candidato per poter raggiungere una larga base di utenza. Le caratteristiche *Open Source* del sistema hanno reso inoltre più agevoli le modalità di sviluppo e installazione del software piuttosto che le alternative proprietarie (ad esempio *iOS*). Le tecnologia principale con cui è stata realizzata l'applicazione è stata quindi l'*Android Software Development Kit (SDK)*. Questo framework mette a disposizione tutte le librerie e i *tool* necessari allo sviluppo del modulo client.



Figura 3.2: Modalità *Navigatore* dell'applicazione mobile PathS.

Oltre alle funzioni di interfacciamento grafico e logica applicativa, l'*SDK* consente anche di sfruttare in modo agevolato i sensori hardware del dispositivo (luminosità e microfono) nonché il sotto-sistema *GPS*, tutte funzioni particolarmente critiche per la riuscita del progetto. Un'altra libreria fondamentale utilizzata nello sviluppo dell'applicazione mobile *PathS* è stata *Wikitude*. Questa libreria è stata selezionata tra altre analoghe, e il suo ruolo è stato quello di offrire le funzionalità di realtà aumentata richieste dalla soluzione. Per ulteriori dettagli riguardanti la selezione degli strumenti e l'implementazione dell'applicazione client si rimanda alla tesi specifica [7].

### 3.3 Stato attuale

In generale l'applicazione presenta un funzionamento completo e copre tutto il caso d'uso previsto. L'operazione di raccolta dati e trasmissione al server viene eseguita correttamente, così come l'interrogazione dei risultati disponibili e la presentazione degli stessi all'utente in modalità mappa e navigazione *turn by turn*. Tuttavia ci sono alcune aree in cui una evoluzione consentirebbe un utilizzo più pratico dell'applicazione e un miglioramento nei dati forniti alla componente server. Alcune di queste aree sono:

- Miglioramento della scala dei valori raccolti e normalizzazione degli stessi, in particolare per quanto riguarda la luminosità. Nei test eseguiti molti valori raggiungono il 100%, presentando pochi casi con valori intermedi nonostante le diverse situazioni reali di applicazione. Questa caratteristica coinvolge l'intero sistema e i servizi forniti, influenzando direttamente i risultati finali.
- Miglioramento delle modalità con cui si comunica all'utente la trasmissione dei dati al server, rendendo esplicito il momento in cui tale comunicazione avviene ed eventuali malfunzionamenti (rete non presente, connessione fallita, etc).
- Miglioramento della responsività della app, in particolare nell'accesso al modulo di navigazione che spesso non è “smooth” in particolare in dispositivi non performanti. In generale i rallentamenti nel feedback e il funzionamento scattoso peggiorano l'esperienza utente e disincentivano l'utilizzo della app.
- Verificare il consumo di risorse che si verifica durante l'esecuzione di tutte le operazioni in *background*, valutando una ottimizzazione delle stesse. In molti casi si è verificato un consumo abbondante della carica batteria e talvolta anche un surriscaldamento del dispositivo.

Al netto degli spunti di possibile miglioramento, lo stato attuale dell'applicazione mobile ha comunque consentito e supportato lo sviluppo della componente server e quindi dell'intera soluzione *PathS*. Le versioni aggiornate dell'applicazione possono essere scaricate da: <http://path-graphs.herokuapp.com/the-project/index.html>.



# Capitolo 4

## PathS Server

La parte cruciale del progetto *PathS* si basa sulla gestione delle informazioni aggiuntive relative ai percorsi pedonali e la possibilità di sfruttarle per fornire un servizio alternativo agli utenti finali. La persistenza, la gestione e l'elaborazione di questi dati avviene nella componente *server*. In questo capitolo saranno presentati i requisiti individuati per assolvere allo scopo del progetto e il modo in cui questi requisiti hanno guidato la definizione dell'architettura del software da sviluppare.

### 4.1 Requisiti

Come per l'altra componente, anche nel caso del server si è preso spunto dalla precedente versione del progetto (*Path2.0*) e si è cercato di identificarne i limiti per poter essere adattato al nuovo contesto e agli scopi più ampi che sono stati definiti. Le situazioni e le necessità a cui si intende applicare il progetto sono del tipo:

- qual è il percorso meno rumoroso in quest'ora del giorno?
- che percorso devo seguire per ottenere il maggior numero di tratti all'ombra?
- quali sono i percorsi preferiti dagli utenti che hanno visitato questa zona?

Le risposte fornite dovranno sempre tenere conto della soddisfabilità della domanda (quindi fornire sempre un percorso valido) ma valutare anche le preferenze espresse dall'utente.

Dal punto di vista tecnologico la soluzione precedente è stata valutata non idonea come base di partenza, in particolare presentava le seguenti caratteristiche che ne rendevano difficile una possibile evoluzione:

- il sistema era stato basato sul concetto dei percorsi accessibili e la ricerca del *routing* esclusivamente come riutilizzo dei tratti noti con questa caratteristica;
- vi è un forte accoppiamento e dipendenza dalle API Google Maps, i cui termini di utilizzo sono cambiati nel tempo e il modo in cui sono integrate è una forzatura inefficiente (interrogazioni multiple per soddisfare una singola richiesta di routing);
- il formato di comunicazione con il client era specifico e non adattabile al nuovo contesto senza una profonda rielaborazione;
- il modello di persistenza delle informazioni non era adatto a gestire il nuovo *set* di informazioni aggiuntive.

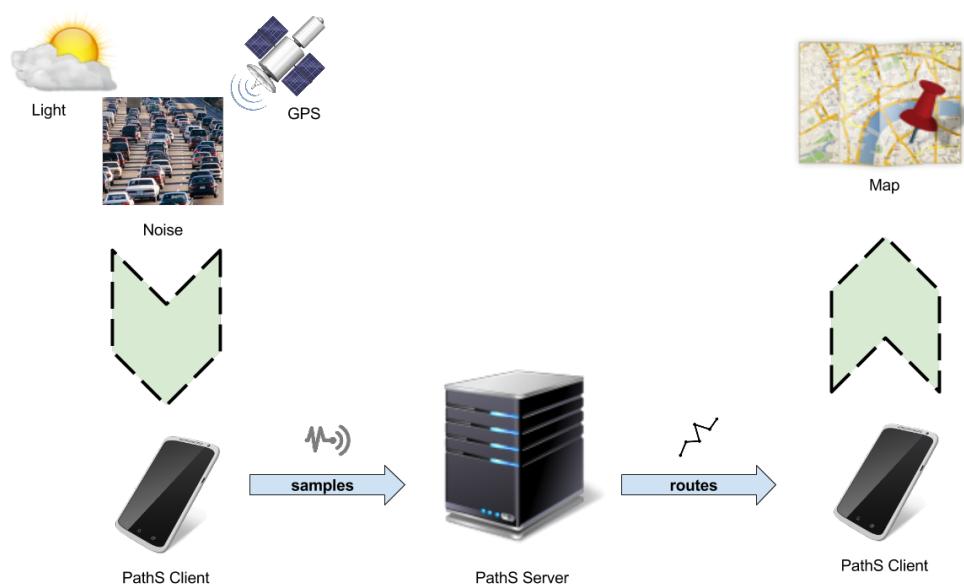


Figura 4.1: Schema generale del progetto PathS.

Considerati questi aspetti, ed in particolare le difficoltà nell'estendere il progetto esistente; con il gruppo di lavoro si è cercato di ridefinire i requisiti del sistema ponendo l'attenzione su alcuni punti principali, ovvero:

- **raccolta campioni:** il sistema deve ricevere i campioni dalle applicazioni *client* in un formato ben definito e dalle possibilità di estensione;
- **persistenza:** il sistema deve memorizzare in modo efficiente e coerente le informazioni ricevute relative a campioni di diversa tipologia;

- **associazione dei campioni ai percorsi:** il sistema deve eseguire l'operazione di associazione delle informazioni raccolte dai sistemi client (sensori) alle informazioni di geolocalizzazione e cartografia (tratti stradali);
- **routing:** il sistema deve implementare un servizio di calcolo dei percorsi pedonali complesso che rielabori le informazioni precedentemente raccolte;
- **interfacciamento con il client:** il sistema deve comunicare con le applicazioni *mobile* e i client *desktop* fornendo le informazioni necessarie alla navigazione.

Oltre a questi requisiti fondamentali per il funzionamento del sistema, sono stati individuati altri criteri preferibili di carattere qualitativo da considerare nella progettazione e implementazione del software. Si possono riassumere in:

- **suddivisione delle responsabilità:** definire in modo specifico le funzioni svolte da ciascun componente del sistema, evitando elementi che svolgono funzioni troppo complesse o che accoppiano troppi elementi. Questo approccio consente una migliore testabilità delle sotto-componenti e la possibilità di rivedere e riprogettare alcuni dettagli senza dover manipolare l'intero prodotto software;
- **astrazione:** identificare la funzione logica di ciascun componente concentrandosi sulla sua interfaccia prima di proseguire nell'implementazione. Questo consente di delineare con precisione il ruolo che dovrà svolgere, precondizioni, risultati attesi e i confini concettuali in cui opera. Rispettare questo criterio rende più facile lo sviluppo e il miglioramento dell'implementazione dei componenti senza doverne ridefinire la funzione logica e riducendo le ripercussioni sugli altri elementi dell'architettura;
- **estensibilità:** rendere agevole sia in termini architettonici che implementativi la possibilità di migliorare ed estendere il sistema. L'obiettivo del progetto è stato volutamente limitato ad un primo risultato tangibile, considerando però che vi sia la possibilità di migliorare ed estendere le sue parti in modo facile e coerente con la base che si andrà a sviluppare.

## 4.2 Componenti

Basandosi sulle funzioni che deve svolgere il server, sono state identificate le componenti logiche principali da sviluppare. Per ciascun caso d'uso, si presenta in dettaglio il ruolo delle componenti e il modo in cui assolvono al raggiungimento del risultato.

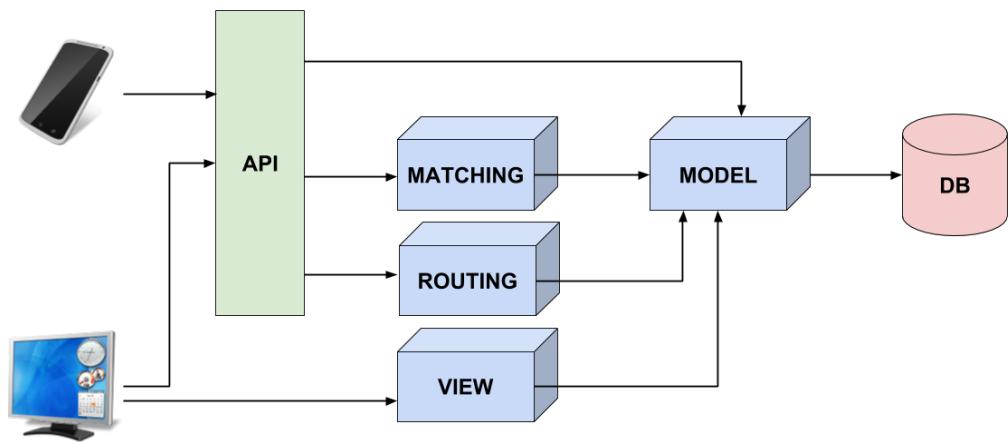


Figura 4.2: Macro componenti del del Server PathS.

### 4.2.1 Ricezione dei campioni

La prima componente che partecipa alla funzione di ricezione dei campioni deve dialogare con il client *mobile*. Si è pensato di definire una **API** su protocollo *HTTP* da utilizzare per tutte le chiamate di invio dati; in questo modo utilizzando un formato semplice che rispetta gli attuali *standard de-facto*, risulta facilmente implementabile con qualsiasi tecnologia *client* ed eventualmente da altre sorgenti dati future. Il componente **API** si occupa quindi di deserializzare i dati della richiesta e riorganizzarla per la persistenza. Il salvataggio avviene su apposito **Database** ma mediato da un componente intermedio di **Model**. Risultano quindi più agevoli le operazioni di interrogazione e persistenza, nonché la leggibilità e semplicità dell'implementazione.

### 4.2.2 Elaborazione dei campioni

I campioni così come sono ricevuti dai client sono in formato grezzo e non consentono di sfruttare le informazioni in esse contenute. Il passo principale

per l'utilizzo dei dati è quello di associare ciascun dato ad un segmento della rete di trasporto. Tutto questo procedimento è sintetizzato con l'espressione **Map Matching** che è di fatto l'algoritmo che esegue tale associazione. Il *matching* viene eseguito sia in modo autonomo dal server che tramite invocazioni dell'**API**. Il risultato delle esecuzioni dell'algoritmo viene comunque salvato a **database** tramite le apposite classi di **model**.

#### 4.2.3 Calcolo Percorsi

Lo scopo pensato per l'utilizzo delle informazioni raccolte è quello di fornire servizi *routing* alternativi, che tengano conto dei dati raccolti per suggerire percorsi che vanno oltre la semplice regola del percorso più breve. Per questo caso d'uso sono quindi coinvolti i componenti:

- **API** per definire un formato con cui i client (*app* o *browser*) richiedono un percorso;
- implementazione di un algoritmo di **routing** che esegue il calcolo effettivo;
- l'algoritmo utilizzi le classi di **model** per accedere alle informazioni aggiuntive utilizzate nel calcolo.

#### 4.2.4 Accesso da Browser

Per tutte le funzionalità principali del server si è pensato di dare un accesso di *monitoraggio* tramite *web browser*. In questo modo è possibile accedere facilmente alle informazioni gestite dal sistema, tramite visualizzazioni grafiche su mappa. La presentazione dei dati recuperati tramite le classi **model** è implementata con pagine *html* e linguaggio *Javascript* eseguito lato *client*. Queste componenti possono essere raggruppate con la definizione di **view**.

### 4.3 Tecnologie

Per la realizzazione del progetto *server* sono state selezionate alcune tecnologie e librerie utilizzate in modo trasversale per lo sviluppo dell'applicazione. I criteri adottati in questa scelta si riassumono in:

- preferenza per gli strumenti *Open Source* facilitando accesso al software, la gestione delle licenze e lo sviluppo di eventuali modifiche;

- preferenza per gli strumenti di larga adozione, per i quali sono disponibili *on-line* documentazione ed esperienza diretta degli utenti,
- preferenza per gli strumenti già adottati in altri progetti, in modo da ridurre i tempi di apprendimento e di configurazione.

Il risultato di questa selezione sono stati:

- **Framework Play!** (<https://www.playframework.com>): utilizzato per la struttura principale dell'applicazione Java. L'adozione di questo framework (già utilizzato anche in esperienze lavorative) ha permesso un rapido *setup* dell'architettura *Model-View-Controller* di base del server. Inoltre risultano semplificate alcune funzioni tra cui il *mapping* delle richieste *HTTP* previste dall'*API*, la gestione della persistenza e l'*Object Relational Mapping* tramite la specifica *JPA* e l'implementazione *Hibernate*. Il framework fornisce inoltre un facile motore di *templating* per lo sviluppo delle pagine di presentazione.
- **PostgresSQL** (<http://www.postgresql.org>): è il *RDBMS* selezionato in quanto tra i prodotti open source più validi e diffusi allo stato attuale. Risulta inoltre particolarmente adatto con le funzioni *GIS* aggiuntive fornite dalle librerie presentate in seguito e fondamentali per l'implementazione del progetto.
- **LeafletJS** (<http://leafletjs.com>): libreria Javascript utilizzata per la presentazione delle mappe interattive. È un prodotto open source altamente configurabile e di facile integrazione. È risultato lo strumento ideale per presentare i dati in questa forma visuale senza legarsi ad altri servizi esterni. Supporta la visualizzazione da *mobile* ed è particolarmente utile per la rappresentazione di *layer* sovrapposti contenenti informazioni diverse.
- **Bootstrap** (<http://getbootstrap.com>): Uno dei framework *HTML/CS-S/JavaScript* più diffusi in assoluto, utilizzato per semplificare la realizzazione delle pagine web. Ha permesso la realizzazione di pagine web con *design responsive* e graficamente gradevoli.

# Capitolo 5

## Ricezione dei campioni

La funzione di ricezione dei campioni ha richiesto la definizione di una nuova *API* di comunicazione con le applicazioni *client* ed in seguito lo sviluppo delle componenti di manipolazione dei dati ricevuti per la relativa persistenza. Si presentano le specifiche tecniche e di implementazione di questi elementi.

### 5.1 API

Nel definire il dettaglio delle chiamate necessarie all'invio dei dati, si è valutato di prediligere i criteri di semplicità e adeguamento alle tecnologie attuali. Si è quindi optato per una *API HTTP* e contenuto in formato JSON. La chiamata da utilizzare per l'invio dei dati risponde alla seguente specifica:

```
POST /api/data

BODY:
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [
          <longitude>,
          <latitude>
        ]
      },
      "id": <unique_id>,
      "properties": {
        "timestamp": <timestamp>,
        "accuracy": <accuracy_value>,
      }
    }
  ]
}
```

```

        "labels": {
            <label_name>: <label_value>,
            ...
        }
    },
    ...
],
"properties": {
    "vote": <vote_number>
}
}

```

Listing 5.1: Invocazione API di invio dati

Il formato definito consente di specificare una lista di elementi di campionamento e per ciascuno di esso le informazioni necessarie. Ogni campionamento è individuato da una chiave *id* generata in modo casuale dal *client* seguendo lo standard UUID versione 4<sup>1</sup>. Per ciascun campionamento sono indicati:

- l'*id* di riferimento;
- le coordinate espresse in *latitudine* e *longitudine*;
- il dettaglio *temporale* dell'avvenuto campionamento;
- l'*accuratezza* del rilevamento GPS così come fornita dal dispositivo;
- le *etichette* e i *valori* per il campione rilevato dai sensori.

Tutti i valori sono numeri e stringhe in formato JSON. Per la struttura del contenuto si è preferito non adottare una forma libera ma piuttosto aderire ad uno standard già definito, ovvero GeoJSON<sup>2</sup>. In questo modo, pur consentendo una struttura semplice ed estendibile (il numero e il tipo di rilevazioni non è limitato), si è favorità l'inter-operabilità con altre librerie e servizi. Risulta inoltre più agevole una validazione del formato dei dati ricevuti. Per le chiamata non è stato previsto alcun protocollo di autenticazione del *client*, ritenendolo non necessario in questa fase di sviluppo del progetto.

## 5.2 Implementazione

L'implementazione dei componenti server che gestiscono la chiamata è risultata agevolata dagli strumenti messi a disposizione dal framewrok selezionato. Per rispondere alla chiamata *HTTP* si è sviluppata la classe `Api.java`, la

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Universally\\_unique\\_identifier](https://en.wikipedia.org/wiki/Universally_unique_identifier)

<sup>2</sup><http://geojson.org>

quale estende la classe `play.Controller` e quindi mappa sul metodo `data()` la gestione della richiesta. Il metodo implementato si occupa di deserializzare i dati della richiesta JSON, ricavando quindi le informazioni specifiche di ciascun campione. La deserializzazione avviene tramite la libreria *Jackson* e le definizioni della struttura nella classi `Feature` e `FeatureCollection`. Una volta estratti i campi necessari, vengono utilizzati per ricreare gli oggetti di modello necessari alla persistenza. Per ciascun invio vengono quindi creati:

- un oggetto **Path** che rappresenta il percorso inviato, lo identifica con un *id* interno, tiene traccia della *data* in cui è stato ricevuto e della *valutazione* indicata dall'utente;
- **molteplici** oggetti **Sample**, uno per ciascun campionamento e quindi in relazione *many to one* con l'oggetto *Path*. Questo oggetto contiene tutte le informazioni relative al campione (*coordinate GPS*, *accuratezza*, *timestamp*, ...) più alcuni campi necessari alla successiva elaborazione (es. *roadsegment\_id*);
- **molteplici** oggetti **Label**, uno per ciascuna tipologia di rilevamento effettuata. Anche questi oggetti sono in relazione *many to one* con l'entità *Sample*. Il valore delle etichette è di tipo numerico decimale.

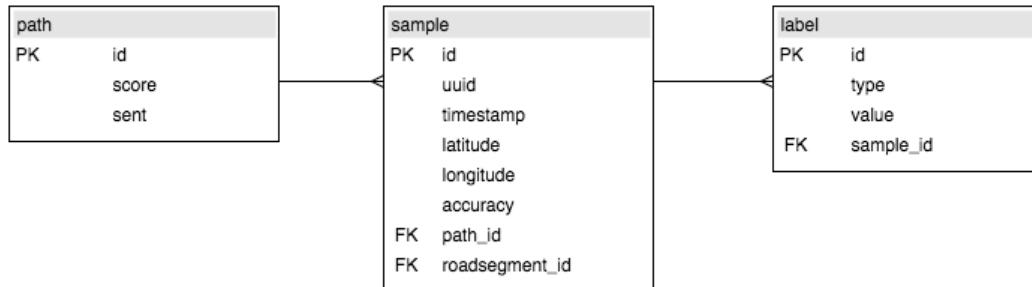


Figura 5.1: Diagramma ER modello dati campionamenti.

La persistenza degli oggetti di modello è agevolata ancora una volta dagli strumenti messi a disposizione dal framework *Play!*. Le classi estendono `play.Model` il quale fornisce i metodi di *default* per il salvataggio a *database* tramite il metodo `save()`. In questo modo non è necessario gestire manualmente né lo schema della base dati né l'implementazione specifica delle *query*. La funzione di ricerca degli oggetti di modello è fornita tramite il metodo di libreria `find()` e derivati.

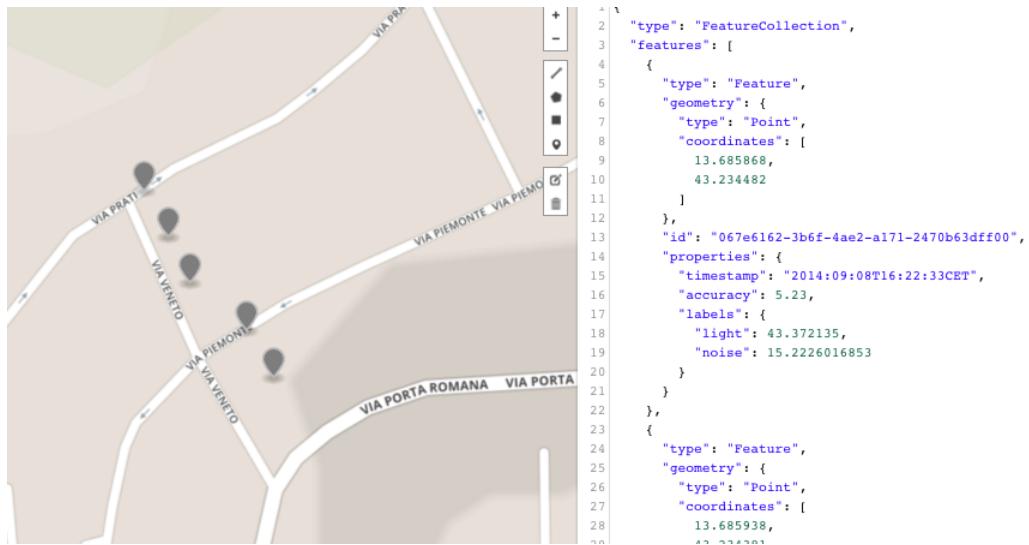


Figura 5.2: Esempio di visualizzazione GeoJSON relativa ad un invio dati.

### 5.3 Esempi

Un esempio di contenuto della chiamata è presente all'indirizzo <http://path-server.herokuapp.com/public/stub/path-samples.json>. Tale esempio è stato utilizzato anche come traccia condivisa per l'implementazione dell'applicazione *client*. La visualizzazione dei dati inviati può avvenire anche con strumenti esterni che interpretano il formato GeoJSON come presentato in figura 5.2.

Il risultato ottenuto dall'invio di alcuni campionamenti al server può essere osservato in figura 5.3. Il server mette inoltre a disposizione una pagina di amministrazione che presenta la lista dei tracciati ricevuti, le informazioni principali, e i link alla visualizzazione delle altre funzionalità (figura 5.4).

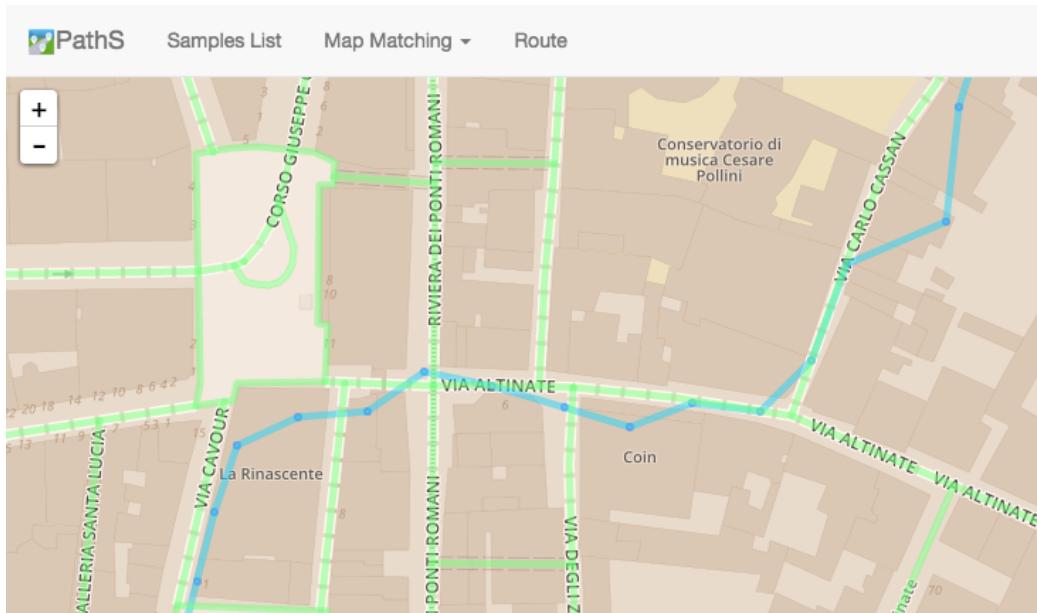
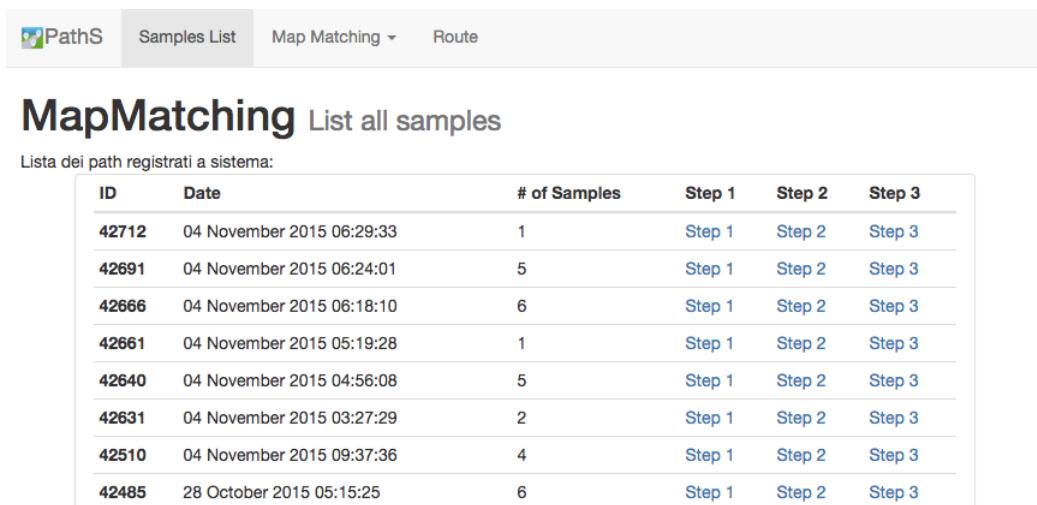


Figura 5.3: Esempio di campionamento ricevuto dal server.

Figura 5.4: Lista dei tracciati *Path* ricevuti.



# Capitolo 6

## Elaborazione dei campioni

### 6.1 Servizi di Cartografia

Lo scopo del progetto richiede necessariamente il recupero e la manipolazione di dati cartografici, in particolare le mappe stradali e i possibili percorsi pedonali. Già dalle prime fasi del progetto, si è quindi presentata la necessità di individuare un servizio di cartografia da cui recuperare questi dati ed eventualmente sfruttare per la successiva manipolazione. I criteri di preferenza in questa scelta sono stati ancora una volta la facilità di accesso al servizio in termini di licenza, la diffusione e la documentazione disponibile.

Le due soluzioni che si sono prospettate sono i maggiori *player* in questo ambito, lasciando poco spazio ad altre opzioni in termini di completezza dei dati e strumenti di integrazione. Si riassumono gli aspetti principali ed in particolare le differenze tra i due servizi che hanno determinato la scelta finale.

#### 6.1.1 Google Maps

E' stato uno dei primi servizi di mappe accessibile via web, fin dall'Ottobre 2005. Negli anni si è evoluto profondamente sia in termini di completezza e accuratezza dei dati disponibili, che in termini di funzionalità e strumenti di interrogazione. Tramite Google Maps oggi è possibile accedere alle mappe stradali di tutto il mondo, percorsi pedonali, siti naturalistici, edifici pubblici di interesse, viste in prima persona e molto altro. Sono inoltre disponibili diversi strumenti per gli sviluppatori che consentono di accedere ai servizi basati su questi dati. La valutazione si è quindi concentrata sulla tipologia degli strumenti disponibili e sulle operazioni che sarebbe stato possibile eseguire (<https://developers.google.com/maps>).

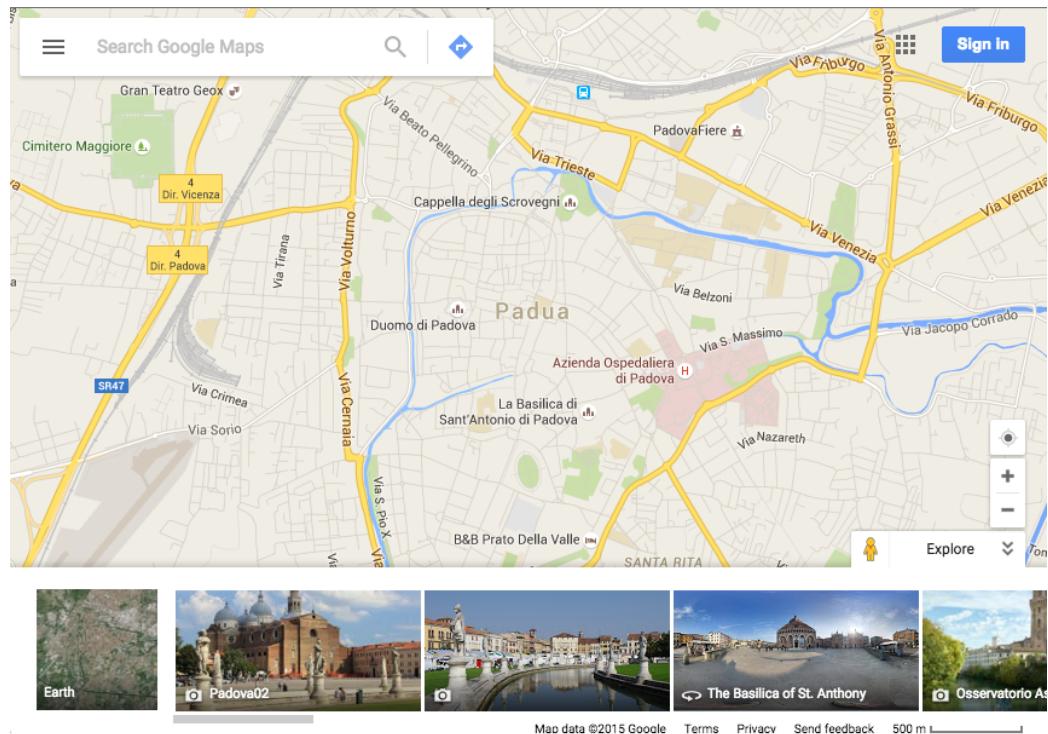


Figura 6.1: Schermata di accesso al servizio *Google Maps*.

Google fornisce *API* e *SDK* per le principali piattaforme applicative, sia *desktop* che *mobile*. Considerato però l'ambito specifico di utilizzo nel server *PathS*, le funzionalità ispezionate sono quelle denominate **Web Services** ovvero interrogazioni *HTTP* con risposte in formato *JSON*.

La prima caratteristica per cui si contraddistingue il servizio Google Maps è che non consente l'accesso diretto alle informazioni cartografiche. Non è possibile estrarre parte di questi dati, siano essi in forma grezza o finita. Da qui la necessità di utilizzare uno dei servizi a disposizione per ottenere le informazioni necessarie, in particolare il servizio **Google Maps Direction API**. Con tale servizio è possibile ottenere un percorso impostando un punto di partenza ed una destinazione. Lo stesso servizio era utilizzato in precedenza dal progetto *Path 2.0* e veniva impiegato direttamente per il calcolo della soluzione finale da proporre agli utenti. Tuttavia questo tipo di accesso non consente di ottenere dati per una determinata zona se non con una precisa interrogazione. Inoltre l'utilizzo delle *API* presenta i seguenti limiti:

- il numero di interrogazioni è limitato dal tipo di piano utilizzato. Il servizio libero e gratuito denominato *Standard* consente 2,500 chiamate al giorno, applicando una tariffa per le successive. Questa situazione

sarebbe stata sostenibile per le prime fasi prototipali del progetto, ma non per una sua applicazione su larga scala.

- il piano di utilizzo *Standard* limita anche l'utilizzo della funzione “way-point” nel calcolo del percorso ad 8 tappe intermedie. Questo risulterebbe un ulteriore limite nel calcolo dei percorsi complessi;
- il servizio Maps opera direttamente sui dati memorizzati da Google ma non è possibile aumentare o integrare direttamente questo insieme di dati. Non è possibile quindi sfruttare gli algoritmi forniti dal servizio (es. calcolo percorso) su un insieme di informazioni *custom* (es. i campioni di luminosità), se non tramite un uso indiretto e improprio delle *API*.

Alla data in cui si è eseguita l'analisi comparativa, i servizi *Google Maps* non disponevano ancora delle *Roads API* introdotte in forma sperimentale nel Marzo 2015. La funzione di interpolazione e mapping dei punti GPS sarebbe risultata molto utile all'applicazione del progetto ma per questo motivo non è stata considerata.

### 6.1.2 OpenStreetMap

Le mappe *OpenStreetMap* (<http://www.openstreetmap.org>) sono il risultato di un progetto collaborativo nato nel 2004. Le informazioni cartografiche sono raccolte e corrette liberamente dai contributori la cui direzione e coordinamento sono seguiti dalla fondazione non a scopo di lucro *OpenStreetMap Foundation*. Il progetto ha ormai una copertura a livello mondiale alla quale hanno contribuito sia organizzazioni governative che private tra cui ricordiamo Yahoo (2006) con le proprie ortofoto aree, l'Automotive Navigation Data (2007) con il database stradale di Paesi bassi, India e Cina ed infine l'inclusione del database stradale TIGER (2007) con le informazioni stradali degli Stati Uniti. Nel 2009 il progetto ha superato la quota dei 100.000 utenti iscritti. La caratteristica più importante dei dati *OpenStreetMap* è la modalità con cui sono diffusi, ovvero coperti dalla licenza *Open Database License* la quale consente un uso libero dei dati anche per prodotti derivati, purchè ne sia sempre citata la fonte originale. Considerate le caratteristiche di libero accesso e riutilizzo, nonchè la completezza e precisione dei dati stessi, la soluzione OpenStreetMap si è candidata da subito come la soluzione ideale per l'implementazione del server *PathS*. Tuttavia le modalità tecniche con cui accedere e manipolare i dati non erano immediate. Le opzioni principali offerte agli sviluppatori per consultare i dati *OSM* sono (<http://wiki.openstreetmap.org/wiki/Develop>):

- **API** ([http://wiki.openstreetmap.org/wiki/API\\_v0.6](http://wiki.openstreetmap.org/wiki/API_v0.6)): consentono di interrogare i dati *OSM* tramite chiamate *HTTP REST* e formato delle risposte *XML*. Sono chiamate riservate agli *editor* e non consentono di eseguire query per ampie porzioni (superiori a 0.25 deg quadrati).
- ***dump* dei dati:** ovvero estrazioni dei dati *OSM* in formato *XML* o come database, aggiornate in modo costante. Le estrazioni complete tuttavia hanno un volume difficilmente gestibile (45*GByte* in formato compresso) e risulta comunque difficile accedere ad estrazioni parziali con strumenti alternativi come le API estese (<http://wiki.openstreetmap.org/wiki/Xapi>).

Entrambe le opzioni non si presentano particolarmente agevoli per le esigenze del progetto, tuttavia grazie alle caratteristiche degli *open data*, è stato possibile trovare delle valide alternative che, pur basandosi sui dati *OSM*, fornisco modalità di integrazione più idonee al contesto.

### Overpass API

Il servizio di interrogazione *Overpass API* è uno strumento di introduzione recente che si è aggiunto come alternativa di consultazione in sola lettura dei dati *OpenStreetMap*. Il suo scopo è quello di agire come un “*database over the web*” che può essere interrogato in modo efficiente tramite un linguaggio specifico ([http://wiki.openstreetmap.org/wiki/Overpass\\_API](http://wiki.openstreetmap.org/wiki/Overpass_API)). Un client può quindi eseguire una chiamata API comunicando la *query* da eseguire e in breve ottenere una risposta che raccoglie anche migliaia di elementi geografici. Il servizio è erogato liberamente e senza limitazione da infrastrutture di servizi terzi (es. <http://overpass-api.de>). Sono presenti inoltre strumenti on-line (ad esempio: <http://overpass-turbo.eu>) i quali assittono lo sviluppo e testano l'esecuzione delle interrogazioni scritte nel formato specifico *Overpass Query Language*.

Le funzionalità offerte da questa soluzione sono state valutate come ottimali nell'ambito del progetto *PathS* server in quanto:

- sono uno strumento specificatamente sviluppato per l'integrazione lato server (*webservice*);
- dispongono di sufficiente documentazione e strumenti di supporto allo sviluppo;
- consentono di interrogare in modo parziale i dati cartografici *OpenStreetMap* per la sola porzione di interesse (ad esempio il singolo percorso analizzato);

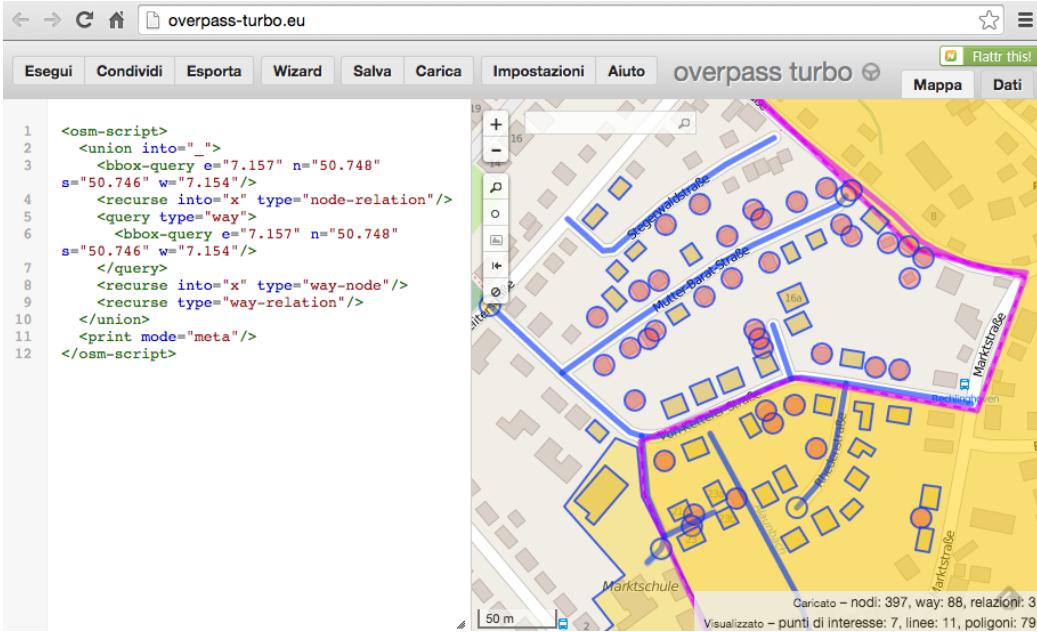


Figura 6.2: Esempio di interrogaione tramite *Overpass API*.

- non impongono nessun tipo di limitazione in termini di numero o estensione delle interrogazioni.

Si è quindi studiata una *query* specifica per interrogaire il servizio ed ottenere i dati necessari al contesto delle elaborazioni *PathS*. Per un determinato percorso o insieme di campioni ricevuti dal *client*, era necessario ricavare tutti i possibili percorsi pedonali che potevano essere coinvolti. La soluzione ottenuta esegue quindi queste due operazioni:

- limita con una *bounding box* la zona in cui eseguire l'interrogaione, ricavandola dai punti più estremi del campionamento;
- estrae tutti i possibili elementi geografici che possono essere considerati percorsi pedonali (strade, passerelle, ponti, sentieri, etc.)

Nell'implementazione del server sono stati sviluppate alcune classi di utilità che agevolano l'interrogaione e la gestione delle risposte al servizio Overpass e sono rispettivamente: `app.utils.OverpassQuery.java` e `app.models.-boundaries.OverpassResponse.java`.

## Mapquest

Oltre al servizio di interrogaione dei dati *OpenStreetMap* si è ritenuto opportuno identificare anche un servizio di *routing* che operasse sugli stessi dati,

interrogabile tramite *webservices* dalla componente server. Sebbene questa funzione non sia indispensabile, è risultata molto utile in fase di sviluppo e come verifica del funzionamento degli algoritmi. Il servizio è inoltre utilizzata come risposta di *fallback* nel caso non vi siano dati disponibili da fornire ai *client*.

La soluzione individuata è quella fornita da *MapQuest* (<https://developer.mapquest.com>). Come per gli altri servizi, le interrogazioni avvengono tramite *API HTTP* in formato *JSON* e quindi gestite da alcune classi *Java* di supporto per l'interrogazione (*app.utils.MapQuestQuery.java*) e l'interpretazione (*app.models.boundaries.MapQuestResponse.java*).

## 6.2 Persistenza ed elaborazione dei dati GIS

Una delle esigenze che si presenta nello sviluppo del server *PathS* è la persistenza e l'interrogazione di dati di tipo geografico. Ad esempio può essere necessario salvare le coordinate di cui si compone un percorso pedonale o ricercare tutti gli elementi salvati entro una certa distanza da una posizione nota. Per eseguire questo tipo di operazioni in modo agevole ed efficiente, è stata adottata la libreria *PostGIS* (<http://postgis.net>).

Questa libreria si propone come estensione del database *PostgreSQL* aggiungendo alcuni elementi fondamentali, tra cui:

- tipi di dato geometrico e geografico (ad esempio punti, linee spezzate, poligoni);
- funzioni ed operatori di manipolazione dei dati geometrici (intersezione, unione, calcolo distanza, proiezione, etc.);
- indici ed ottimizzazioni per la gestione dei dati spaziali.

Le operazioni sui dati spaziali sono accessibili come funzioni e primitive del linguaggio *SQL* utilizzato per interrogare il *database*. L'utilizzo di questa libreria ha consentito di implementare agevolmente le operazioni richieste dagli algoritmi e di garantire già una prima forma di ottimizzazione e accesso intelligente ai dati salvati, caratteristica non banale nel caso si fosse dovuta ri-sviluppare da zero.

Di pari passo all'adozione della libreria per il *database*, si è valutata anche uno strumento per l'ambiente *Java* il quale fornisce la rappresentazione base dei tipi geometrici le operazioni basilari su di essi. La soluzione individuata è stata la libreria *JTS* (<http://www.vividsolutions.com/jts/JTSHome.htm>) la quale tramite il *package* `com.vividsolutions.jts` fornisce una gerarchia

di classi e operazioni per la rappresentazione e manipolazione di dati spaziali 2D. L'implementazione è in puro linguaggio *Java* e quindi si integra facilmente nel contesto software del server.

## 6.3 Algoritmo di Map Matching

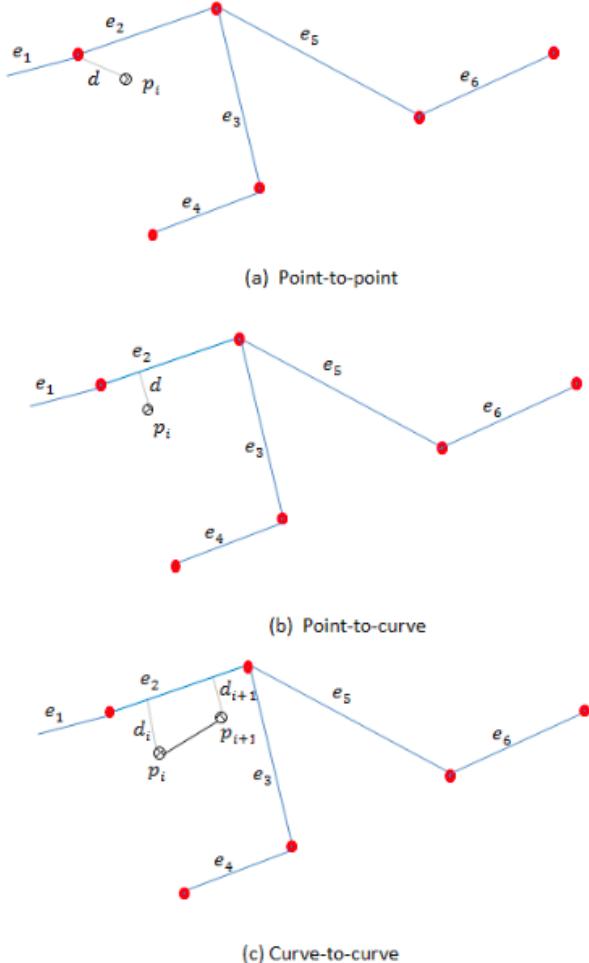
L'ambito a cui si applica il progetto *PathS* prevede la raccolta di campioni da dispositivi che non si muovono liberamente nello spazio ma piuttosto che seguono dei percorsi predefiniti ovvero la rete stradale e pedonale. Ottenuto un insieme di rilevamenti e le relative coordinate GPS è necessario quindi associare queste informazioni a quella che viene definita rete di trasporto. Questa associazione consente, oltre ad aumentare il contenuto informativo della base dati, ad un miglioramento e correzione dei dati relativi ai campioni, dato che o per errori nella rilevazione o per differenza di rappresentazione le coordinate potrebbero non corrispondere.

L'operazione può essere eseguita tramite l'utilizzo di una famiglia di algoritmi denominati di *Map Matching* [6, capitolo 3.3.1]. Tali algoritmi sono una funzione che associa a ciascuna rilevazione di un oggetto in movimento la corrispondente posizione nella rete di trasporto. Le tecniche di questo tipo si suddividono in due approcci:

- ***online***: quando i rilevamenti sono disponibili progressivamente con la loro acquisizione e spesso si applicano a contesti *real-time* in cui risulta critica la velocità di risposta e i dati che si elaborano riguardano solo la situazione corrente e passata;
- ***offline***: quando l'esecuzione dell'algoritmo può essere rimandata ad un momento successivo in cui tutte le rilevazioni sono disponibili e i vincoli temporali non sono così forti. Solitamente queste situazioni di post-processamento conducono a risultati qualitativamente migliori.

Gli algoritmi di entrambe le categorie possono essere ulteriormente categorizzati sulla base della strategia che implementano in:

- **geometrici**: quando si considerano unicamente le caratteristiche geometriche dei segmenti della rete di trasporto. Alcuni esempi di questa tecnica sono gli algoritmi *point-to-point*, *point-to-segment* o *curve-to-curve* in cui l'oggetto in movimento viene associato calcolando diversi tipi di distanza (Euclidea, perpendicolare, Fréchet).
- **topologici**: quando oltre alle caratteristiche geometriche si considera anche la topologia della rete di trasporto con le possibili adiacenze e

Figura 6.3: Esempio d algoritmi di *Map Matching* geometrico.

connessioni tra i vari segmenti. Generalmente questo tipo di approccio fornisce risultati migliori rispetto a quello esclusivamente geometrico ma è suscettibile di errori soprattutto in presenza di *outlier*. In questa categoria non si considerano le informazioni aggiuntive come velocità e direzione del movimento.

- **probabilistici:** sono algoritmi che sfruttano una regione di errore, tipicamente di forma ellittica, la cui dimensione dipende dalla velocità di spostamento. Quest'area viene utilizzata per una selezione dei possibili *match* in particolare nelle zone di congiunzione. La scelta finale viene eseguita considerando direzione e velocità dello spostamento, nonché le caratteristiche di connessione degli elementi della rete.

- **ibridi:** sono gli algoritmi che uniscono gli approcci precedenti in soluzioni particolarmente intelligenti in cui la combinazione dei vari criteri consente di eseguire scelte più accurate. Questi algoritmi inoltre possono fare uso di informazioni da sistemi terzi come ad esempio l'*Assisted GPS*.

Indipendentemente dall'algoritmo utilizzato, i risultati dipendono molto dall'accuratezza della risoluzione della mappa: più alta è la definizione, migliori saranno i risultati ottenuti. Spesso gli algoritmi presuppongono che le informazioni sulla rete di trasporto siano corrette e complete. Così è anche nel progetto *PathS* in cui si assume come insieme completo i dati estratti da *OpenStreetMap* e i campioni vengono associati a quei percorsi. In altre situazioni tuttavia è anche possibile utilizzare gli algoritmi di *Map Matching* per l'operazione inversa, ovvero migliorare la risoluzione o estendere la rete di trasporto a partire dai rilevamenti di un oggetto in movimento. Questo aspetto non è stato considerato in questa fase di progetto ma solo come possibile evoluzione.

### 6.3.1 ST-Matching

Per l'applicazione al progetto *PathS* è stato necessario individuare un algoritmo di *Map Matching* che consentisse l'associazione dei rilevamenti di rumore e luminosità ai tratti di percorso delle mappe *OpenStreetMap*.

L'utilizzo di un algoritmo di tipo *offline* è lecito, dato che l'elaborazione dei percorsi può avvenire dopo il loro invio alla componente *server* e in quel caso tutti i rilevamenti sono stati eseguiti. Inoltre l'operazione non è direttamente collegata ad una interrogazione dell'utente ma piuttosto ad un processo *batch* di elaborazione del server, il quale può tollerare anche una alta complessità computazionale e tempi di elaborazione lunghi.

Tra le alternative disponibili in letteratura, si è optato per una opzione che fosse relativamente semplice da implementare e al tempo stesso garantisse dei risultati qualitativamente accettabili. Un algoritmo di tipo esclusivamente geometrico ad esempio non sarebbe stato utile, dato che non tiene conto delle caratteristiche di sequenzialità dei percorsi pedonali che andiamo a trattare e potrebbe portare ad associazioni non corrette.

Un buon compromesso individuato è l'algoritmo ***Spatio Temporal Map Matching (ST-Matching)*** [5]. Nella soluzione si considerano sia le caratteristiche spaziali e topologiche della rete di trasporto che i vincoli temporali e di velocità della traiettoria. L'algoritmo si compone principalmente di tre passaggi [6.4] in cui:

- si analizzano i dati GPS e la rete di trasporto per l'identificazione dei possibili candidati;
- si valutano singolarmente i candidati tramite considerazioni spazio-temporali;
- si determina il *matching* ottimale con il supporto di un grafo dei candidati.

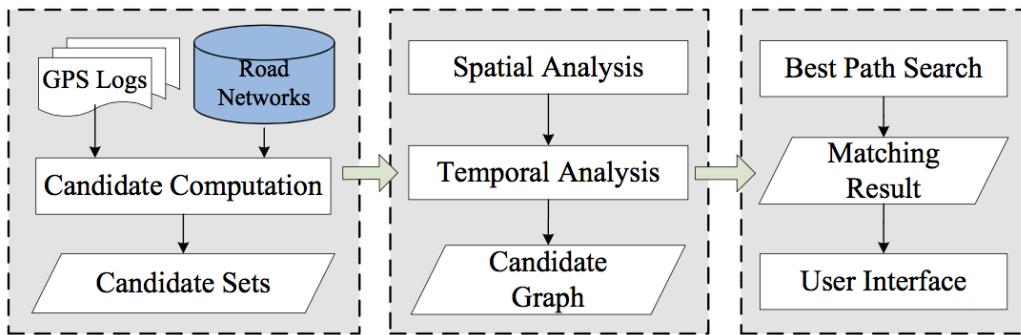


Figura 6.4: Schema dei passi previsti dall'algoritmo *ST Matching*.

Le caratteristiche dell'algoritmo fanno sì che possano ottenere dei risultati in linea con le aspettative in termini di errore di precisione e al tempo stesso delle *performance* accettabili.

Si presenta l'implementazione di dettaglio così come è stata eseguita per ciascun passo previsto dall'algoritmo.

### 6.3.2 Pre-elaborazione del percorso - step 1

Il primo passo consiste nell'identificare tutti i possibili elementi della rete di trasporto che possono essere coinvolti dal percorso che si sta elaborando.

La prima operazione è quindi quella di calcolare una *bounding box* in grado di contenere tutti i campioni del percorso. Il calcolo avviene sfruttando le librerie *PostGIS* e una *query* opportunamente sviluppata per selezionare le coordinate massime e minime di latitudine e longitudine per tutti i campioni GPS del percorso. I valori vengono ulteriormente allargati di un valore fisso (20 metri) per assicurarsi di coprire un'area sufficientemente estesa al netto degli errori di accuratezza.

La stessa *bounding box* viene quindi utilizzata per l'interrogazione dei dati tramite *OverpassAPI* e quindi si ottengono tutti i tratti della rete di trasporto possibilmente coinvolti nel percorso. Le informazioni vengono quindi

inserite a database nella tabella `RoadSegment` e gestite tramite la classe di modello `app.models.RoadSegment.java`. I segmenti già presenti a sistema non vengono duplicati.

Le operazioni citate sono implementate nel metodo `doStep1()` della classe `app.controllers.MaMatching.java` e il risultato della pre-elaborazione può essere visualizzato tramite l'apposita interfaccia del server (figura 6.5). La *bounding box* è rappresentata in arancione, mentre tutti i segmenti verdi sono gli elementi elaborati e aggiunti a sistema.

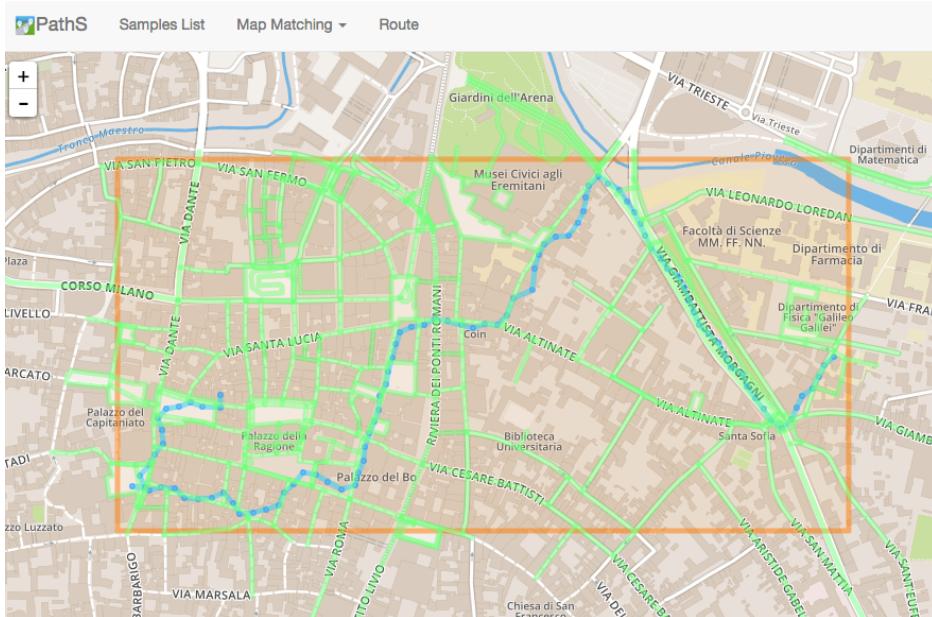


Figura 6.5: Risultato dell'esecuzione *step 1* dell'algoritmo *ST Matching*.

### 6.3.3 Identificazione dei candidati - step 2

Il passo successivo dell'algoritmo è l'individuazione, per ciascun campione, dei possibili candidati. Per ciascuna posizione GPS, la quale potrebbe non appartenere alla rete di trasporto, si ricercano le possibili corrispondenze. Data quindi la traiettoria  $T = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$  si ricava per ciascun punto  $p_i, 1 \leq i \leq n$  i possibili segmenti candidati in un raggio  $r$ . Tale operazione è implementata con una *query GIS* e la funzione `ST_DWithin`. La distanza utilizzata per la selezione dei segmenti da considerare è stata impostata sperimentalmente alla misura di 20 metri.

Per ciascun segmento vengono quindi calcolati i **punti candidati** calcolando la *Line Segment Projection*. La *Line Segment Projection* di un punto  $p$

rispetto ad un segmento di rete di trasporti  $e$  è definita come il punto  $c$  appartenente ad  $e$  tale per cui  $c = \arg \min_{\forall c_i \in e} dist(c_i, p)$  dove  $dist(c_i, p)$  ritorna la distanza di ciascun punto  $c_i$  su  $e$ . [6.6]

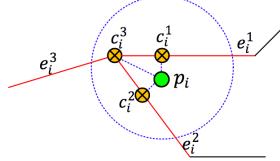


Figura 6.6: Calcolo dei candidati di un punto con il metodo *Line Segment Projection*.

L'implementazione del calcolo è risultato relativamente agevole tramite la funzione `ST_ClosestPoint(geometry, geometry)` fornita dalla libreria *PostGIS*. In questo modo per ciascun punto sono state ricavate le coordinate delle proiezioni sui segmenti adiacenti e quindi possibili candidati per il *matching*. I punti ottenuti sono salvati a database tramite la classe di modello `CandidatePoint`. Il risultato di questo calcolo è visualizzabile per ciascun elemento nell'interfaccia di amministrazione del server tramite i collegamenti allo *step 2* (figura 6.7). Il punto del campionamento è indicato in colore blu mentre sono segnalate con cerchi arancioni le posizioni dei punti candidati.

Una volta individuato l'insieme dei candidati per ciascuna posizione di campionamento della traiettoria  $T$ , il problema si riduce a scegliere da ciascun insieme un solo elemento tale per cui:  $P : c_1^j 1 \rightarrow c_2^j 2 \rightarrow \dots c_n^j n$  è l'assegnazione ottima per  $T = p_1 \rightarrow p_2 \rightarrow \dots p_n$ .

### 6.3.4 Valutazione e selezione dei candidati - step 3

Ciascun candidato viene quindi valutato considerando due aspetti:

- analisi spaziale
- analisi temporale

Nella valutazione spaziale sono considerate sia le caratteristiche geometriche che quelle topologiche della rete di trasporto. Per ciascun punto ricavato al passo precedente vengono calcolati due valori, rispettivamente probabilità di osservazione e probabilità di trasmissione.

La prima misura indica la possibilità per cui un campione GPS  $p_i$  possa coincidere con il candidato  $c_i^j$  basandosi sulla distanza tra i due punti  $dist(p_i, c_i^j)$ . Il calcolo della probabilità si esegue assumendo che l'errore del campionamento GPS abbia una distribuzione normale con deviazione standard di 20 metri (misurazione empirica riportata in [5, paragrafo 5.2]). Il

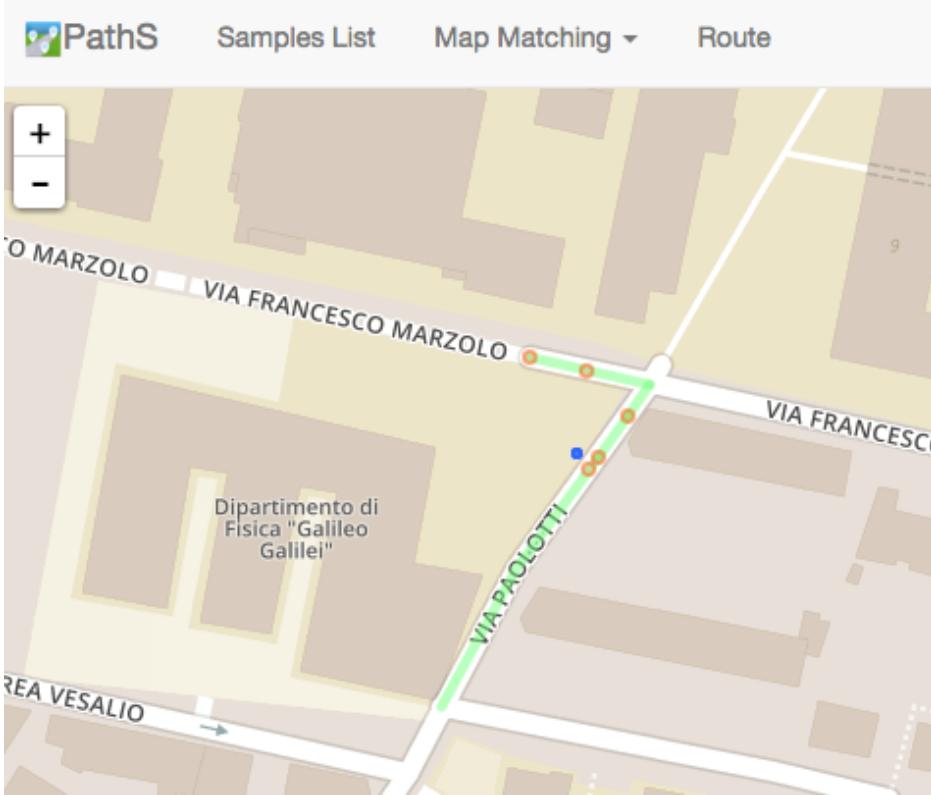


Figura 6.7: Risultato dell'esecuzione step 2 dell'algoritmo *ST Matching*.

valore della probabilità di osservazione sarà quindi tanto più elevato quanto il candidato è vicino alla posizione del campionamento.

Tuttavia la probabilità di osservazione non tiene conto del contesto in cui si verifica la rilevazione, né l'intorno degli altri punti. Questo può portare in diverse situazioni ad un *matching* errato come quello presentato in figura 6.8.

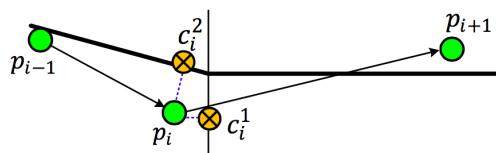


Figura 6.8: Esempio di situazione in cui è necessaria la valutazione della probabilità di trasmissione.

La linea ingrossata rappresenta una strada principale, mentre la linea più fine indica una strada secondaria. Nonostante il candidato  $c_i - 1$  sia più vicino

al punto  $p_i$  dovremmo assegnare il *match* al candidato  $c_i^2$  in virtù del fatto che sia  $p_i - 1$  che  $p_i + 1$  sono assegnati alla strada principale. Per esprimere questa intuizione, il calcolo della probabilità di trasmissione viene eseguito rapportando la distanza euclidea tra i candidati del rilevamento precedente  $c_i - 1^j$  con il candidato corrente  $c_i^j$  rispetto al percorso effettivo tra le due stesse posizioni seguendo il percorso più breve nella rete di trasporto (*shortest path*). Tanto più questo percorso rappresenta la traiettoria ideale, tanto più alto sarà il valore della probabilità di trasmissione.

I due valori calcolati sono quindi combinati per ottenere la valutazione totale secondo l'analisi spaziale come funzione  $F_s$ :

$$F_s(c_i - 1^t \rightarrow c_i^s) = N(c_i^s) * V(c_i - 1^t \rightarrow c_i^s), 2 \leq i \leq n$$

dove  $N$  è la funzione per il calcolo della probabilità di osservazione e  $V$  è la funzione di calcolo della probabilità di trasmissione tra il candidato  $c_i - 1^t$  per il punto  $p_i - 1$  e il candidato  $c_i^s$  per il punto  $p_i$ .

La valutazione secondo l'analisi temporale risulta utile per i casi in cui l'analisi spaziale non sia determinante come l'esempio riportato in figura: 6.9. La sola analisi geometrica non fornisce indicazioni rilevanti come invece può essere il fatto di considerare la velocità con cui si muove l'oggetto rispetto al tratto di rete di trasporto. La comparazione tra la velocità media delle rilevazioni GPS e la velocità indicativa del tratto è definita in dettaglio nel [5, paragrafo 5.3].

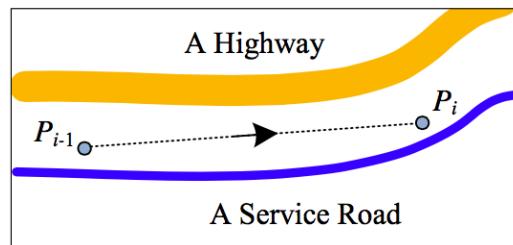


Figura 6.9: Esempio di situazione in cui è necessaria l'analisi temporale.

Tutte le operazioni di calcolo sono implementate nella classe `app.utils.-STMapMatching.java` con l'ausilio della libreria matematica *Apache Commons Math3*.

Eseguite l'analisi temporale e spaziale per tutti i candidati, si è in grado di generare un grafo dei candidati  $G'_T(V'_T, E'_T)$  per la traiettoria  $T = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$  in cui  $V'_T$  è l'insieme dei candidati per ciascun punto di campionamento GPS ed  $E'_T$  è l'insieme di tutte le connessioni tra due candidati vicini come rappresentato in figura 6.10.

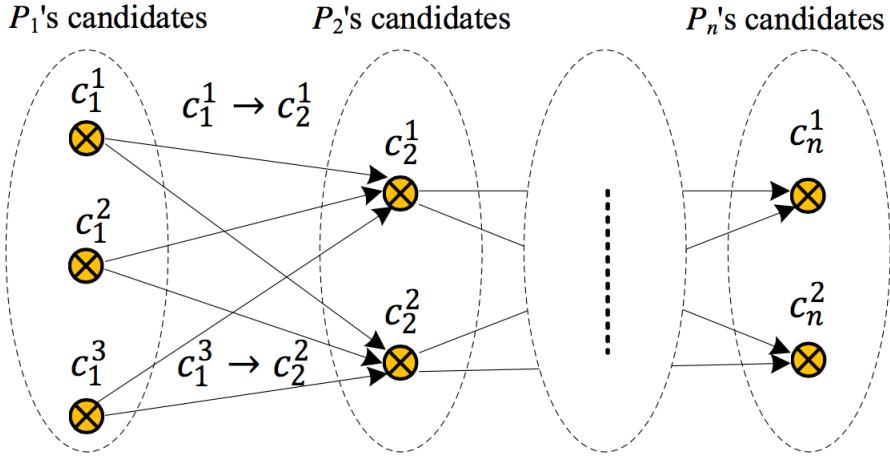


Figura 6.10: Grafo dei candidati.

Il problema del *matching* si riconduce alla ricerca del *path* di candidati  $P_C$  per l'intera traiettoria  $T$  definito come:  $P_C : c_1^{S1} \rightarrow c_1^{S2} \dots c_1^{Sn}$ . Il punteggio totale per una sequenza di candidati si calcola come sommatoria delle funzioni precedentemente esposte ovvero:  $F(P_C) = \sum_{i=2}^n F(c_i - 1^{s_{i-1}} \rightarrow c_i^{s_i})$ . Per tutti le possibili sequenze di candidati, si vuole trovare l'assegnazione ottima che copre l'intera traiettoria, espresso in termini formali:

$$P = \operatorname{argmax}_{P_C} F(P_C), \quad \forall P_C \in G'_T(V'_T, E'_T)$$

In generale, il problema di ricerca del percorso più lungo in un grafo è *NP* completo, tuttavia il fatto che il grafo sia diretto e aciclico (*DAG*) fa sì che possa essere calcolato in modo efficiente sfruttando l'ordine topologico del grafico ottenuto per costruzione.

Il procedimento implementato per il calcolo è quello presentato al [5, paragrafo 5.4]) con il nome di *FindMatchedSequence* e se ne riporta lo pseudo codice.

```

Input: Candidate graph  $G'_T(V'_T, E'_T)$ 
Output: The longest sequence  $P : c_1^{s1} \rightarrow c_2^{s2} \rightarrow \dots c_n^{sn}$  in  $G'_T$ 

Let  $f[]$  denote the highest score computed so far;
Let  $pre[]$  denote the parent of current candidate;

for each  $c_1^s$  do
   $f[c_1^s] = N(c_1^s)$ 
for  $i = 2$  to  $n$  do
  for each  $c_i^s$  do
     $max = -\infty$ 
    for each  $c_j^{s-1}$  do
      if  $f[c_j^{s-1}] + N(c_j^{s-1} \rightarrow c_i^s) > max$ 
         $max = f[c_j^{s-1}] + N(c_j^{s-1} \rightarrow c_i^s)$ 
         $pre[c_i^s] = c_j^{s-1}$ 
    end
     $f[c_i^s] = max$ 
  end
end

```

```

for each  $c_i - 1^t$  do
     $alt = f[c_i - 1^t] + F(c_i - 1^t \rightarrow c_i^s);$ 
    if ( $alt > max$ ) then
         $max = alt$ 
         $pre[c_i^s] = c_i - 1^t;$ 
         $f[c_i^s] = max;$ 
Initialize \emph{rList} as an empty list;
 $c = argmax_{c_n^s}(f[c_n^s])$ 
for  $i = n$  to 2 do
    rList.add( $c$ );
     $c = pre[c]$ ;
rList.add( $c$ );
return rList.reverse();

```

Il risultato dell'esecuzione dell'algoritmo produce in *output* il *matching* ottimale per tutti i candidati. Il risultato dell'esecuzione applicato ai percorsi *PathS* è consultabile da interfaccia di amministrazione tramite i collegamenti allo *step 3*.

Al termine della procedura di *Map Matching* il sistema interpreta i risultati ottenuti assegnando i valori dei campioni di luminosità e rumore al segmento della rete di trasporto associato. Questa informazione è considerata nelle procedure di calcolo dei percorsi presentate nel prossimo capitolo.

# Capitolo 7

## Routing

### 7.1 Implementazione

Uno degli obiettivi principali del progetto *PathS* è di fornire indicazioni di navigazione ai suoi utenti. Si è quindi presentata la necessità di dover rispondere a quesiti di *routing* utilizzando le informazioni raccolte tramite le operazioni di campionamento e l’interrogazione del servizio di cartografia.

Allo scopo di agevolare l’implementazione del componente e di ottimizzare le operazioni di calcolo è stata selezionata per l’utilizzo la libreria *PGRouting* la quale fornisce un grande supporto in questo senso. Questa estensione si appoggia alle funzioni *GIS* del database, introducendo le capacità di calcolo del percorso tramite l’utilizzo di algoritmi generici già implementati e configurabili ad esempio Floyd-Warshall, Shortest Path A\*, Dijkstra e *Traveling Sales Person*.

Tuttavia per poter operare correttamente la libreria necessita che siano verificate alcune precondizioni sulla base dati da utilizzare, in particolare:

- la rete di trasporto deve contenere informazioni corrette in corrispondenza delle intersezioni e degli estremi dei tratti adiacenti;
- la base dati deve contenere le informazioni sulla topologia della rete affinché si possa costruire un grafo utilizzabile dagli algoritmi.

Per affrontare il primo problema, la libreria mette a disposizione una funzione di “*noding*”. Tale procedura riceve in input la tabella in cui sono contenuti i dati geografici e un valore di tolleranza, quindi rielabora le informazioni cercando di rimuovere situazioni non coerenti ad esempio:

- intersezioni non gestite come il caso (1) in figura 7.1. In alcuni casi i segmenti presenti nel database potrebbero intersecarsi senza però che

sia presente il corrispondente punto di intersezione. In questa situazione le successive operazioni di *routing* non sarebbero in grado di sfruttare l’intersezione come punto di svolta. La procedura quindi analizza i dati *GIS* per individuare le possibili sovrapposizioni e in corrispondenza delle stesse crea dei vertici aggiuntivi spezzando i segmenti. La procedura mantiene traccia del numero e dell’ordine in cui vengono decomposti gli elementi originali.

- segmenti adiacenti con estremi che non coincidono come il caso (2) in figura 7.1. In questa situazione la funzione cerca di semplificare e riunire in un unico vertice gli elementi che sono all’interno della distanza di tolleranza impostata. Senza questa elaborazione i due segmenti non potrebbero essere utilizzati dall’algoritmo di *routing* dato che non sono comunicanti.

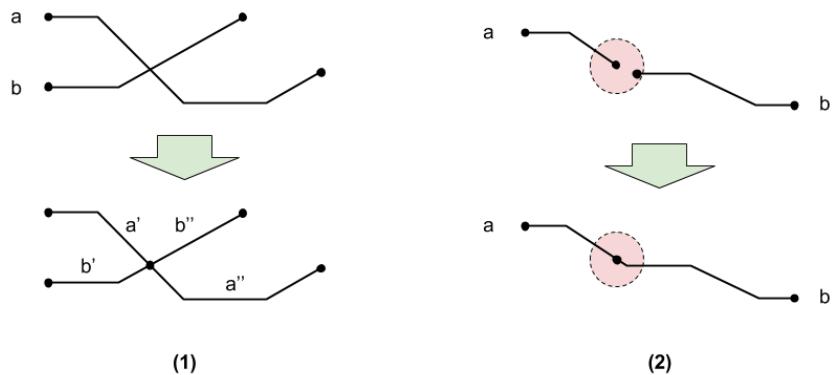


Figura 7.1: Casi in cui si applica la procedura di *noding*.

Nel caso del database di *PathS* la procedura di *noding* viene eseguita sulla tabella `roadsegment` e il risultato è salvato nella tabella `roadsegment_noded`. La procedura non altera i dati originali, ma è necessario rilanciarla ogni volta che si aggiungono nuovi elementi all’insieme dei segmenti importati tramite il servizio di cartografia.

Le informazioni dei segmenti rielaborate tramite la procedura di *noding* possono essere quindi utilizzate per generare il grafo della rete di trasporto, sul quale saranno lanciati gli algoritmi di *routing*. L’operazione di generazione del grafo è implementata da un’altra funzione messa a disposizione dalla libreria, ovvero: `pgr_createTopology`. In modo simile alla precedente, la funzione riceve come argomenti il nome della tabella da cui leggere le informazioni geografiche dei segmenti e un valore di tolleranza entro il quale eseguire l’operazione di unificazione dei tratti non con-

nessi. Il risultato, nel caso specifico, porta alla generazione della tabella `roadsegment_noded_vertices_pgr`.

La struttura della tabella utilizzata per la persistenza delle informazioni dei segmenti e le conseguenti elaborazioni è presentata in figura 7.2.

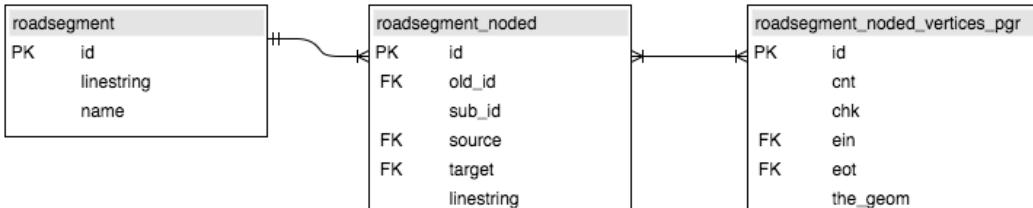


Figura 7.2: Schema tabelle relative ai segmenti della rete di trasporto.

Eseguite le operazioni di pre-processamento, i dati a sistema sono nella forma corretta per poter essere utilizzati dagli algoritmi di routing della libreria *PGrouting*. A supporto dell'utilizzo e interpretazione del risultato è stata implementata la classe `app.utils.RouteResult` e le operazioni da eseguire sono state riunite nel metodo `getRouteResultFromQueryResult()`.

## 7.2 Percorsi calcolati

### 7.2.1 Shortest Path

La prima tipologia di percorso che si è deciso di calcolare è stato il caso semplice del percorso più breve (*Shortest Path*). In questo modo è stato possibile valutare l'effettiva funzionalità della libreria e correggere eventuali errori nella procedura. L'algoritmo adottato per il calcolo è quello di *Dijkstra* implementato dalla funzione `pgr_dijkstra`. Le modalità richieste per l'invocazione dell'algoritmo richiedono come argomenti:

- il nodo sorgente da cui inizia il percorso, specificato tramite `id` così come è presente nella tabella dei vertici;
- il nodo destinazione del tragitto, anch'esso individuato tramite `id`;
- una *query* con la quale ricavare i costi per ciascun arco del grafo della rete di trasporto.

Per identificare i nodi sorgente e destinazione, il server esegue una ricerca a database calcolando i vertici più vicini rispetto alle coordinate di latitudine e longitudine fornite.

Per quanto riguarda il costo da utilizzare per ciascun arco, trattandosi del semplice caso *Shortest Path*, si è sviluppata una *query* che ritorna come valore la lunghezza in metri del segmento stesso. L'algoritmo di *routing* cercherà quindi di minimizzare il peso degli archi selezionati e quindi il tragitto più breve.

Il risultato dell'interrogazione è manipolato tramite la classe di utilità `RouteResult` per ricavare alcune informazioni aggiuntive come ad esempio la lunghezza totale del percorso o le indicazioni di svolta tra un tratto di strada e il successivo.

Il modo in cui si presentano i dati all'applicazione client ma anche al browser web è stato scelto coerentemente con le altre *API*; anche in questo caso si è optato per una risposta in formato *GeoJSON* ed un esempio di invocazione è riportato in 7.1.

```
GET /api/route?

BODY:
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [
            11.8882108,
            45.4107728
          ],
          ...
          [
            11.8917144,
            45.409708
          ]
        ]
      },
      "properties": {
        "comment": "Shortest\u2022Path\u2022\u2022generato\u2022con\u2022PGRouting",
        "color": "red",
        "distance": 388.26767042966577,
        "maneuvers": [
          {
            "narrative": "Parti\u2022da\u2022Passeggiata\u2022Arturo\u2022Miolati",
            "iconUrl": "/public/images/start.gif",
            "streets": [
              "Passeggiata\u2022Arturo\u2022Miolati"
            ]
          }
        ]
      }
    }
  ]
}
```

```

        },
        ...
    {
        "narrative": "Svolta\u00e1 sinistra in \u2022 Via\u2022 Leonardo\u2022 Loredan",
        "iconUrl": "/public/images/left.gif",
        "streets": [
            "Via\u2022 Leonardo\u2022 Loredan"
        ]
    },
    {
        "narrative": "Sei arrivato\u00e1 alla destinazione",
        "iconUrl": "/public/images/end.gif",
        "streets": [
            "Via\u2022 Leonardo\u2022 Loredan"
        ]
    }
],
"maneuverIndexes": [
    0,
    ...
    9
]
}
}
]
```

Listing 7.1: Invocazione API di routing

L'interpretazione della stessa risposta avviene con modalità diverse nei sistemi client, ovvero:

- nella applicazione mobile viene attivata la modalità di navigazione e le indicazioni *turn-by-turn*;
- nel browser si presenta il percorso completo sulla vista mappa segnalando la lunghezza totale e un riassunto delle indicazioni di svolta.

### 7.2.2 Percorsi con *label*

Uno degli obiettivi principali del progetto *PathS* è il calcolo del percorso sfruttando le informazioni ricevute dai campionamenti di luce e rumorosità. L'implementazione dell'algoritmo di *routing* per questo caso utilizza la stessa procedura presentata al punto precedente, manipolando però le strutture dati in modo da utilizzare le informazioni aggiuntive relative alle *label*.

Le problematiche da risolvere in questo contesto sono state principalmente due:

- introdurre la gestione del contesto **temporale** in cui viene eseguita l’interrogazione così come per i dati da interrogare. Mentre per il caso semplice *Shortest Path* si fa riferimento ad informazioni immutabili nel tempo, in questo caso una interrogazioni in momenti diversi della giornata può comportare la consultazion di dati diversi e quindi risultati diversi;
- modificare la logica di **calcolo dei pesi** introducendo la valutazione dei campioni a sistema, in un modo coerente e funzionale con gli obiettivi di *routing*.

Per gestire del primo problema, si è proceduto con il definire il concetto di **classe temporale**. Tutti i campioni ricevuti sono stati suddivisi, sulla base del *timestamp* della rilevazione stessa, in insiemi distinti che rappresentano una fascia oraria delle giornata. Come compromesso tra semplicità e accuratezza dei risultati, si è scelto di definire quattro classi temporali uniformi, ciascuna delle quali copre un intervallo di 6 ore. Rispettivamente:

- classe 0 dalle 00:00 alle 05:59: rappresenta il periodo notturno in cui i campioni di luminosità dovrebbero essere praticamente nulli così come quelli di rumorosità dovrebbero presentare valori molto bassi;
- classe 1 dalle 06:00 alle 11:59: rappresenta il periodo influenzato delle condizioni di luce mattutine così come dalle attività (es. traffico pendolare);
- classe 2 dalle 12:00 alle 17:59: è la fase in cui i campionamenti di luce dovrebbero essere determinanti data la maggiore esposizione;
- classe 3 dalle 18:00 alle 23:59: è l’intervallo temporale serale, in cui la luce rilevata torna ad essere meno influente mentre i campionamenti audio potrebbero essere influenzati da attività particolari (es. traffico, attività di intrattenimento)

Per agevolare l’attribuzione dei campioni alla relativa classe temporale in tutte le operazioni a database, è stata definita una funzione di utilità così come consentito dal database *PostgreSQL*. La definizione della funzione è `time_class(timestamp without timezone) → integer` la quale riceve in input il timestamp del campione e ritorna l’intero delle classe temporale di appartenenza. Accentrandando in un unico punto l’operazione di assegnazione, risulterà più agevole, se fosse necessario, cambiare le regole da utilizzare.

Un’altra operazione a supporto della manipolazione dei dati per questo caso di *routing* è stata la definizione di alcune viste a database le quali riasumono le informazioni relative ai campionamenti e le presentano in una

forma più agevole per lo sviluppo delle query. Le viste implementate sono `light_sample` e `noise_sample` la cui struttura è presentata in figura 7.3. In questo modo si riassumono in un unico punto le informazioni complete relative ad un campinamento, ovvero:

- l'**id** di riferimento;
- il **segmento** della rete di trasporto a cui è stato attribuito;
- la **classe temporale** a cui appartiene;
- il **valore** rilevato per la *label* in oggetto.

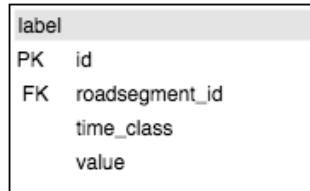


Figura 7.3: Schema viste di supporto campionamenti.

L'ultima operazione necessaria per implementare il calcolo dei percorsi influenzato dalle *label* è stata la definizione delle modalità di **calcolo dei pesi** nella valutazione degli archi della rete di trasporto.

La formula da utilizzare deve rispettare le seguenti caratteristiche:

- deve considerare nel peso degli archi i campioni ricevuti e attribuiti a quel percorso;
- deve considerare solo i campionamenti che riguardano la classe temporale richiesta (momento della richiesta o impostata manualmente);
- deve comunque considerare che si vuole ottenere un percorso breve per cui è permesso un “allungamento” del tragitto a fronte di caratteristiche più vantaggiose (es. meno rumoroso) ma entro una certa soglia;
- alcuni tratti potrebbero non avere campionamenti attribuiti, quindi è necessario pesare in modo adeguato anche l'assenza di valori *label*.

Il risultato finale a cui si è giunti è il seguente:

$$c = l + (\alpha * l(\frac{\sum_{i=1}^{i=n} v_{ti}}{n} - \beta))$$

dove:

- $c$  è il costo utilizzato come peso per l'arco;
- $l$  è la lunghezza del segmento espressa in metri;
- $\alpha$  è un parametro compreso tra 0 e 1 che indica il rapporto tra la componente di peso dipendente dalla lunghezza e la componente relativa alle *label*;
- $v_{ti}$  sono tutti i valori dei campioni per la classe temporale  $t$  normalizzati  $\geq 0$  e  $\leq 1$ ;
- $\beta$  è un parametro di soglia  $\geq 0$  e  $\leq 1$  utilizzato per scalare i segmenti senza rilevazioni assegnate.

Come si può vedere, il calcolo così formulato rispetta le condizioni per i casi limite, ovvero:

- per valori di  $v_{ti} \approx 1$  e quindi molto alti, il peso totale  $c$  è maggiore della lunghezza, al massimo aumentato del fattore  $\alpha$  impostato di default a 0.25;
- per valori di  $v_{ti} \approx 0$  e quindi molto bassi, il peso attribuito è inferiore e quindi l'algoritmo tende a favorire l'uso di quel segmento (meno luminoso, meno rumoroso, etc.);
- nel caso non siano presenti campioni  $v_{ti}$ , il segmento è valutato solo per la sua lunghezza, mantenendo le proprietà dell'algoritmo per la ricerca del percorso più breve. Il segmento sarà preferito ai tratti con campionamenti dai valori alti sulla base del parametro  $\beta$  di base impostato a 0.5.

La porzione di codice in cui è stato implementato il calcolo e definite le *query* da fornire all'algoritmo di *routing* è nella classe `app.implementations.-SPDLightRouter` la quale, come gli altri servizi di calcolo del percorso, implementa l'interfaccia `app.interfaces.Router`.

Con modalità del tutto analoghe, è stato implementato lo stesso algoritmo di *routing* per il caso dei percorsi meno rumorosi valutando le *label* di tipo `NOISE`.

### 7.2.3 Servizio Map Quest

Come sistema di verifica dei risultati è stato implementato un quarto servizio di routing, il quale non opera sui dati a sistema ma ottiene la risposta da un servizio esterno. Questa implementazione è stata utilizzata come controprova

dei percorsi proposti dagli algoritmi di *routing* nonchè come alternativa di *fallback* nel caso in cui le informazioni inserite a database non siano sufficienti a coprire la zona interessata dalla richiesta.

Il servizio utilizzato per l’interrogazione è offerto liberamente da *MapQuest* e anch’esso si basa sui dati della cartografia *OpenStreetMap*. L’*API* fornita agli sviluppatori è di tipo *HTTP* con formato di risposta *JSON*. Il codice sviluppato a supporto dell’esecuzione della chiamata *webservice* e interpretazione della risposta è presente nella classe di utilità `app.utils.-MapQuestQuery` e nella classe di modello `app.models.MapQuestResponse`. L’operazione eseguita è sostanzialmente quella di richiedere il percorso più breve tra la posizione di partenza e destinazione indicate tramite le coordinate di latitudine e longitudine.

Il componente sviluppato per questa funzionalità implementa anch’esso l’interfaccia `app.interfaces.Router` rendendo quindi del tutto uniforme la modalità di presentazione dei dati ai client.



# Capitolo 8

## Conclusione

### 8.1 Risultati

Il risultato finale ottenuto dal progetto *PathS* può essere consultato effettuando una richiesta di *routing* per una zona coperta dai campionamenti di test come quella presentata in figura 8.1.

In questa situazione tutte le operazioni di pre-processamento ed elaborazione dei dati si sintetizzano nel tipo di risposta che viene fornita. Nell'esempio presentato possiamo notare che:

- i diversi tipi di calcolo si discostano nel risultato suggerendo percorsi diversi;
- i percorsi suggeriti hanno lunghezze totali analoghe, aumentando leggermente per rispettare la preferenza accordata (valutazione del rumore o della luminosità);
- il percorso più breve calcolato (in rosso) è praticamente coincidente con quello fornito dai servizi esterni (in blu);
- il percorso meno luminoso (indicato in grigio) cerca di evitare per quanto possibile le zone con campionamenti di luminosità dai valori alti;
- il percorso meno rumoroso (indicato in verde) non suggerisce il percorso più breve ma comprende tratti con campionamenti di rumore bassi;

Una conferma delle caratteristiche spazio-temporali della soluzione si può ottenere effettuando lo stesso tipo di interrogazione per una classe temporale diversa. Come si può vedere in figura 8.2 i percorsi proposti sono diversi e

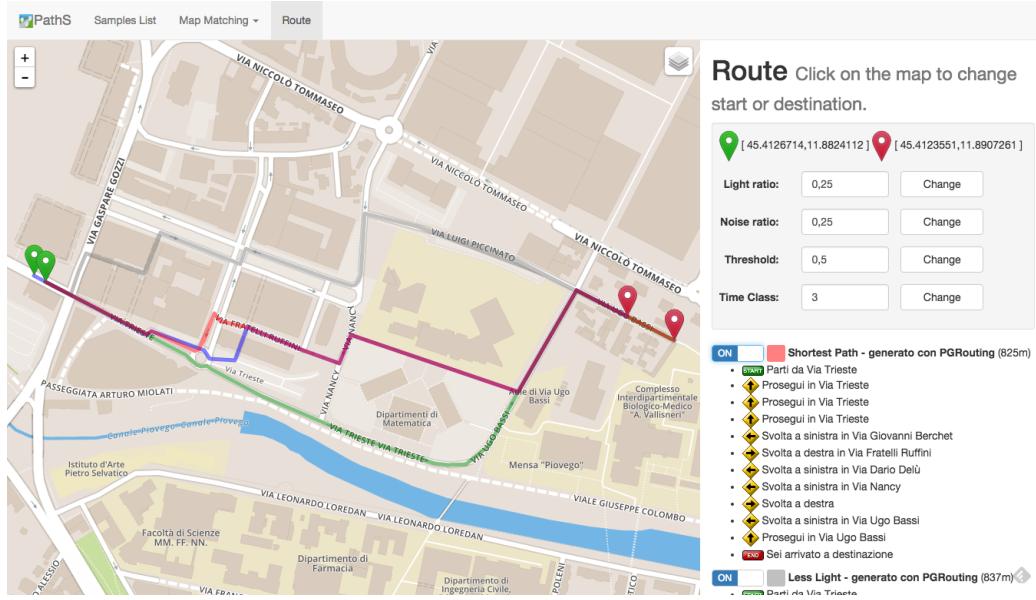


Figura 8.1: Esempio di richiesta di *routing* per la classe temporale 3.

tendono ad uniformarsi. Nella fascia oraria specificata infatti (dalle 17:59 alle 24:00) i campioni di luminosità non sono più molto influenti e quindi l'algoritmo propone un percorso molto più simile a quello più breve.

Nei test effettuati il sistema ha presentato dei tempi di risposta accettabili per tutte le richieste di *routing*. Per questi casi si ritiene che le prestazioni generali possano essere adeguate ad una interazione utente sia tramite browser che da terminale mobile. Per quanto riguarda le operazioni di processamento dei dati, pur essendo eseguite *offline* e quindi non influenzando direttamente le attività degli utenti finali, potrebbero essere analizzate con maggiore dettaglio e probabilmente ottimizzate ulteriormente.

Nello sviluppo di tutte le componenti si è cercato di seguire tutti i criteri di progettazione richiesti, utili ad un risultato facilmente modificabile ed estendibile.

## 8.2 Miglioramenti ed Evoluzioni

Già nella fase di sviluppo sono stati individuate alcune possibili aree di miglioramento, vengono presentate in seguito come spunto per possibili attività in futuro.

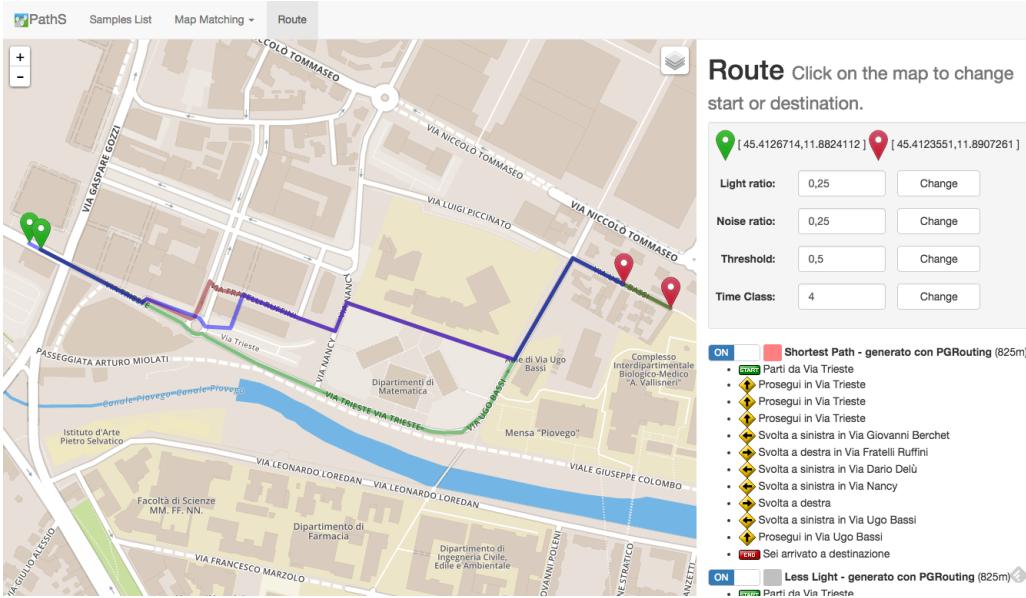


Figura 8.2: Esempio di richiesta di *routing* per la classe temporale 4.

### Sistema di generazione di dati di test e procedura automatica di verifica

Così come presentato nel [6, capitolo 3.4] può risultare utile l’adozione di un sistema di generazione di dati sintesi per le traiettorie. Questo approccio consentirebbe di accedere ad un’ampia base di dati controllati sulla quale valutare gli algoritmi sia in termini di *performance* che in termini di correttezza e qualità dei risultati.

### Strategia di suddivisione segmenti

Nell’implementazione attuale non viene applicato nessun intervento in termini di suddivisione dei segmenti. Questo semplifica le operazioni di assegnazione dei campioni e facilita la corrispondenza tra gli elementi salvati a sistema con quelli presenti nel servizio *OpenStreetMap*. In alcuni casi limite tuttavia i segmenti forniti dalla cartografia possono risultare troppo estesi, in particolare se si presentano misurazioni con valori molto diversi. Un possibile miglioramento può riguardare quindi la procedura di analisi dei campioni introducendo l’operazione di suddivisione in sotto-segmenti per le situazioni in cui può migliorare la qualità dei dati a sistema.

### Algoritmi di map-matching

Le attività del progetto si sono focalizzate nell’implementazione di un singolo algoritmo di *Map matching* allo scopo di ottenere una prima soluzione funzionante. Può essere interessante riprendere l’analisi sulle alternative presenti in letteratura e implementare degli algoritmi alternativi. L’applicazione degli algoritmi ad un sistema automatico di generazione dei dati e di test potrebbe quindi portare ad una comparativa tra le varie soluzioni e ad una valutazione su quale sia l’implementazione che meglio si adatta al contesto e alla natura dei campioni raccolti.

### Algoritmi di routing

Un ragionamento analogo al precedente può essere applicato alla selezione ed implementazione degli algoritmi di *routing*.

In una fase del progetto si è pensato di implementare questa componente da zero, applicando una soluzione di programmazione dinamica per *Shortest Path Problem with Resource Constraints (SPPRC)*, sulla base dei lavori [3, Desrochers e Soumis] e successivi [4, capitolo 4.4].

Per difficoltà implementative, si è poi optato per la soluzione tramite l’utilizzo della libreria *PGRouting*. La realizzazione di alcune alternative, probabilmente basate su algoritmi diversi, anche in questo caso potrebbe portare alle selezione di un prodotto che presenti caratteristiche migliori in termini di qualità dei risultati forniti o di complessità.

### Altri servizi

I servizi offerti da *PathS* potrebbero non riguardare solo le interrogazioni relative al *routing*. I dati raccolti e il relativo processamento potrebbe essere una base di partenza per altri tipi di analisi, ad esempio uno studio sulle zone più frequentate nei diversi momenti della giornata o sui flussi più probabili. Analogamente si potrebbero sviluppare applicazioni del tipo “*get-together*” che favoriscono l’incontro tra gli utenti e lo scambio di informazioni in chiave *social*.

# Appendice



# Appendice A

## Manuale installazione

Si riassumono i requisiti necessari all’installazione del sistema e le modalità di configurazione.

### A.1 Ambiente

Il sistema è stato sviluppato in ambiente *Linux* e per la sua installazione ed esecuzione si consiglia il sistema operativo *Ubuntu Linux ver. 14.04 LTS*. Il requisito non è vincolante, ma semplifica l’installazione e la gestione dei pacchetti necessari per le dipendenze. In termini di risorse il sistema richiede almeno 2 unità di calcolo a 64 bit e 2GB di memoria RAM disponibile.

### A.2 Dipendenze

L’ambiente di esecuzione si compone di due elementi principali, il **database** contenente tutte le informazioni persistenti del sistema e la **logica applicativa** sviluppata con tecnologia *Java*.

#### A.2.1 Database

E’ necessario utilizzare un database *PostgreSQL ver 9.3* o superiore. L’installazione può avvenire sia tramite il gestore dei pacchetti del sistema operativo che in modalità manuale. Oltre al pacchetto standard, sarà necessario installare due estensioni del database, in particolare:

- **PostGIS**: una estensione che consente la memorizzazione di dati spaziali e fornisce altre funzioni di manipolazione di dati geografici (<http://postgis.net>);

- **PGRouting:** una libreria che estende ulteriormente le funzioni del database geospaziale introducendo algoritmi di *routing* e procedure di supporto (<http://pgrouting.org>).

Nei sistemi *Ubuntu Linux* l'installazione del servizio di database e di tutte le dipendenze può avvenire tramite l'esecuzione dei comandi:

```
sudo add-apt-repository ppa:georepublic/pgrouting-unstable
sudo apt-get update
sudo apt-get install postgresql-9.3-pgrouting
```

Seguire le indicazioni presentate a riga di comando, in particolare per le prime operazioni in cui si aggiunge un *repository* non ufficiale al gestore dei pacchetti.

### A.2.2 Java

Per eseguire il codice con cui è stata sviluppata la logica applicativa del server, è necessario disporre di un ambiente *Java Virtual Machine* compatibile. Si consiglia di installare il pacchetto *Oracle Sun JDK ver. 1.7.0* secondo le modalità previste dal sistema operativo usato. Non si garantisce l'esecuzione del software in ambiente *Java 8* né con altre *JVM* diverse da quella Oracle SUN.

Nello sviluppo è stato utilizzato il framework di supporto *Play! Framework ver. 1.2.7*. Installare il pacchetto seguendo le indicazioni sul sito ufficiale (<https://www.playframework.com/documentation/1.2.7/install>). La modalità standard consiste nello scaricare l'archivio e scompattarlo in una posizione nota. Le librerie necessarie all'esecuzione del codice sono incluse nel *repository* del codice e saranno recuperate tramite i successivi passi di installazione.

### A.2.3 Repository

Il codice sviluppato è disponibile tramite *repository* pubblico *GIT* all'indirizzo <https://github.com/tzorzan/path-server.git>. Clonare il *branch master* in locale per ottenere una copia aggiornata del codice da eseguire.

## A.3 Configurazione

Eseguire il *restore* del database iniziale. Il file da cui eseguire il ripristino è presente nella posizione *install/paths.backup* ed è in formato archivio. Eseguendo questa operazione si crea un database con la struttura corretta ed

un set di dati di partenza. Il ripristino può essere eseguito tramite tool grafici o da linea di comando utilizzando le modalità previste da *PostgreSQL* (<http://www.postgresql.org/docs/9.3/static/backup.html>).

A questo punto è necessario istruire l'applicazione fornendo il puntamento al database da utilizzare. Questo può essere fatto impostando la variabile d'ambiente *DATABASE\_URL* prima di avviare l'applicazione. Impostare il valore sostituendo i parametri necessari:

```
export \
DATABASE_URL=postgres://<user>:<pwd>@<address>:<port>/<db_name>
```

Impostare le credenziali di un utente che possiede diritti completi sullo schema, dato che alcune procedure richiedono la cancellazione e la creazione di alcune tabelle.

Per installare le dipendenze (librerie *Java*) necessarie all'esecuzione è necessario posizionarsi nella directory base del progetto e lanciare il comando:

```
play deps
```

Saranno scaricate tutte le dipendenze necessarie e salvate nella cartella */lib* usata durante l'esecuzione del software.

## A.4 Avvio

Avviare l'applicazione tramite il comando:

```
play run
```

In questo modo si avvia il server e resta collegato al terminale in uso.

Per avviare in *background* il servizio e quindi lasciare il server attivo usare i comandi di avvio e spegnimento:

```
play start
play stop
```

Il software utilizzerà come *log* di sistema il file: */logs/system.out*.

## A.5 Note

E' necessario riconfigurare i client mobile affinchè puntino al nuovo servizio server avviato. Questa operazione allo stato attuale può essere effettuata modificando il file sorgente *com.path3ar.modelUtilities.java* e rieseguendo packaging ed installazione dell'applicazione.



# Bibliografia

- [1] Giampiero Calma, Claudio E. Palazzi e Armir Bujari. «Web Squared: Paradigms and Opportunities». In: *Proc. of the International Workshop on DIstributed SImulation and Online gaming (DISIO 2012) - ICST SIMUTools 2012*. Desenzano, Italy, 2012.
- [2] Giuseppe Cardone et al. «The ParticipAct Mobile Crowd Sensing Living Lab: The Testbed for Smart Cities». In: *IEEE Communications Magazine* (ott. 2014).
- [3] Martin Desrochers e Francois Soumis. «A Generalized permanent labelling Algorithm for the Shortest Path Problem with Time Windows.» In: (ott. 1986).
- [4] Jacques Desrosiers et al. «Handbooks in OR and MS». In: Elsevier Scienc B. V., 1995. Cap. 2.
- [5] Yin Lou et al. «Map-Matching for Low-Sampling-Rate GPS Trajectories». In: *ACM SIGSPATIAL GIS 2009*. Association for Computing Machinery, Inc., nov. 2009. URL: <http://research.microsoft.com/apps/pubs/default.aspx?id=105051>.
- [6] Nikos Pelekis e Yannis Theodoridis. *Mobility Data Management and Exploration*. Springer-Verlag New York, 2014.
- [7] Stefano Tombolini. «PathS: un'applicazione basata sul paradigma Web Squared». Tesi di laurea mag. Università degli Studi di Padova, dic. 2014.
- [8] Chenren Xu et al. «Crowdsensing the Speaker Count in the Wild: Implications and Applications». In: *IEEE Communications Magazine* (ott. 2014).