Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

GEO PHOTO ROUTING:
Progettazione ed implementazione di una applicazione per routing pedonale

Relatore:
Chiar.mo Prof.
Bononi Luciano

Correlatore:
Dott. Luca Bedogni

Presentata da:
Bonezzi Mirko

II Sessione
Anno Accademico 2016/2017

Ringrazio il professor Luca Bedogni
per avermi seguito e consigliato
durante la stesura della tesi.

ABSTRACT

La maggior parte dei servizi di routing, sia web che mobile, quando fornisce un percorso per arrivare ad una determinata destinazione, suggerisce spesso il percorso più breve. L'obbiettivo di questo progetto di tesi è dimostrare che è possibile creare un servizio di routing che non indichi solamente il percorso più breve tra 2 punti, ma anche quello più emotivamente piacevole, rilassante, che aumenti il benessere di chi lo percorre.

Per rendere ciò possibile, ci si è basati sull'utilizzo di foto geotaggate presenti su un sistema di archiviazione e condivisione foto online: Flickr. Si ipotizza che un luogo geografico in cui sono state scattate molte fotografie, sia allora un luogo piacevole da visitare o da cui transitare. Questa ipotesi fondamentale guida quindi alla creazione di un percorso secondario, più piacevole, che passi da determinati punti di interesse.

In questo elaborato viene descritta l'implementazione e la progettazione di una piattaforma che abbia l'obbiettivo di mettere a disposizione degli utenti il servizio di routing appena descritto. È stato utilizzato un server Apache sul quale alcuni script PHP si occupano di gestire le richieste del Client e calcolare, tramite un opportuno algoritmo, il percorso adatto alle loro richieste. Inoltre è stata sviluppata un'applicazione Android, Client-side, su cui l'utente può effettuare richieste e visualizzare percorsi.

Indice

Elenco delle figure

Introduzione

Negli anni '50, è nata una disciplina chiamata **psicogeografia** [1], che viene definita come lo "studio degli effetti precisi dell'ambiente geografico, disposto coscientemente o meno, che agisce direttamente sul comportamento affettivo degli individui". La psicogeografia studia dunque le correlazioni tra psiche e ambiente, affermando quindi che l'ambiente esterno influenza le emozioni umane.

Partendo da questo presupposto, si può facilmente capire come talvolta, quando non si deve necessariamente prendere la strada più veloce per arrivare a destinazione, è piacevole scegliere strade alternative che offrono scenari meravigliosi. Si preferisce passare da un viale alberato o da una famosa piazza nel centro città piuttosto che da una strada trafficata. Tuttavia, molti servizi di routing, non offrono la possiblità di scegliere questa possibilità di viaggio. L'obbiettivo del progetto di tesi è, quindi, creare una piattaforma che permetta di calcolare e mostrare all'utente un percorso tra due punti che sia alternativo al percorso più breve e che offra un'esperienza piacevole, transitando da determinati punti di interesse. Ma come fare a capire quali zone, strade, luoghi sono emotivamente stimolanti?

Per rispondere a questa domanda si è deciso di fare affidamento alle reali esperienze delle persone. Grazie all'aumento di fotocamere digitali e all'inclusione di queste nei dispositivi mobili, è molto più facile scattare foto in qualunque occasione e solitamente si è portati a fotografare qualcosa che ci colpisce particolarmente, qualcosa di emotivamente stimolante e piacevole. Da questo presupposto si intuisce che là dove c'è un buon numero di foto scattate, allora esiste anche un luogo piacevole da visitare.
È necessario, però, avere la possibilità di recuperare una grande quantità di foto scattate da diverse persone riguardo una certa area geografica. Per rispondere a questo problema, si è deciso di utilizzare uno dei servizi online di condivisione di immagini, Flickr, che come altri servizi dello stesso tipo, viene sempre più utilizzato con l'avanzare degli anni. Flickr offre l'opportunità, tramite un'opportuna API[1], di accedere alle immagini presenti nel suo Database ed ai relativi *metadati*, che rappresentano informazioni riguardanti le immagini, tra cui, uno dei più importanti per il nostro scopo, il *geotag*. I geotags indicano l'esatto luogo geografico in sono state scattate le fotografie e vengono automaticamente acquisiti dai dispositivi mobili sopra citati o, in alternativa, possono essere specificati dall'utente.
Grazie all'utilizzo dei geotag si può facilmente capire quali sono i luoghi in cui sono state

[1] Application programming Interface

scattate più fotografie, cioè i luoghi che possono essere riconosciuti come interessanti e piacevoli da osservare.

Nasce così **Geo Photo Routing**, una piattaforma che permette di mostrare un itinerario di viaggio alternativo, più emotivamente piacevole, mediante l'utilizzo delle immagini geotaggate presenti su Flickr.
L'utente, attraverso una semplice interfaccia grafica, deve inserire il punto di partenza, quello di arrivo e un raggio d'azione, dopodiché la piattaforma si occuperà di calcolare e visualizzare sulla mappa sia il percorso più breve, calcolato da Google Maps, sia il percorso con maggiori punti d'attrazione, calcolato dall'algoritmo presente sul Server, sfruttando le immagini geotaggate.
Il presente elaborato si focalizza sulla descrizione di questo progetto, in particolare dello sviluppo di un'applicazione Android, della relativa applicazione sul Server e dell'algoritmo utilizzato per il calcolo del miglior itinerario alternativo.

Nel **primo capitolo** viene descritto il campo di ricerca interessato e vengono introdotte e descritte applicazioni attualmente esistenti da cui si è preso spunto per la creazione di **Geo Photo Routing**. In particolare verrà descritta la prima versione del progetto **Geo Photo Routing**, sviluppata da una studentessa bolognese come progetto di tesi. Essa era pensata esclusivamente per web, con caratteristiche che sono state riprese ed ampliate durante lo sviluppo del presente progetto.

Nel **secondo capitolo** si parla della progettazione del sistema, in particolare viene esposta la sua architettura, spiegate quali sono le tecnologie utilizzate, qual'è il motivo che ha spinto alla loro scelta e come utilizzare l'applicazione Android.

Nel **terzo capitolo** vengono spiegate le scelte implementative effettuate durante lo sviluppo della piattaforma.

Particolare enfasi viene data alla spiegazione dell'algoritmo utilizzato per il calcolo e la selezione del percorso da mostrar all'utente, in quanto rappresenta un'importante miglioria rispetto alla prima versione di **Geo Photo Routing**.

Capitolo 1

Stato dell'Arte

Da tempo l'uomo cerca di rappresentare tramite carte geografiche il globo nella sua interezza, disegnando la posizione di strade, città, fiumi e molto altro. I vari atlanti e carte geografiche disegnate sono state fatte in modo sempre migliore con l'avanzare degli anni, grazie anche alla possibilità di visualizzare tramite i satelliti, la reale posizione geografica dei vari elementi da rappresentare.
Con l'avvento del personal computer e soprattutto di internet, sono state messe a disposizione degli utenti mappe visualizzabili dalla propria postazione casalinga. Nel 2005, nasce soprattutto **Google Maps**, un servizio di mapping *web-based* che dà la possibilità di visualizzare sul web mappe interattive.
Dopo pochi anni fanno il loro ingresso gli smartphone ed i servizi di mapping iniziano ad essere alla portata di tutti, in qualsiasi luogo ci si trovi.
Tra le tante funzionalità offerte da **Google Maps**, una delle più importanti è rappresentata da **Google Directions**, un servizio di *routing* che dà la possibilità di visualizzare il percorso più breve da percorrere per andare da un qualsiasi punto A ad un qualsiasi punto B, considerando la rete stradale del luogo. Questo servizio, se integrato su smartphone, rende possibile arrivare in qualsiasi luogo conoscendo immediatamente il percorso più breve, ovunque ci si trovi.

Per permettere di visualizzare il percorso più breve tra due punti, Google Directions necessita di conoscere la conformazione della rete stradale ed utilizza un complesso algoritmo per la ricerca del *cammino minimo*. Gli algoritmi per la ricerca dei *cammini minimi* sono svariati, ma l'algoritmo usato da Google per il calcolo dell'itinerario stradale si basa molto probabilmente sull'Algoritmo di Dijkstra [2], in cui viene utilizzato un grafo che discretizza la rete stradale mondiale. Questo algoritmo fornisce il cammino più breve tra due nodi all'interno di un grafo pesato.

Google Maps ne usa però usata una versione modificata, in quanto Dijkstra è troppo semplice per lavorare su vasta scala; è un algoritmo che si può utilizzare in svariati ambiti, ma non può funzionare su una fitta rete stradale a scala mondiale. L'algoritmo di Dijkstra, non cerca direttamente il percorso breve verso un dato nodo, ma esplora prima tutti i nodi possibili e questo incrementa di gran lunga il tempo di calcolo, soprattutto in un grafo in cui sono presenti molti nodi.[3]

Col passare del tempo sono nati diversi servizi di *routing* stradale, oltre a Google Maps. Solitamente l'obbiettivo è quello di calcolare e visualizzare il percorso più breve tra due punti. Alcuni di questi servizi, però, permettono di specificare diverse opzioni di routing, prefiggendosi obbiettivi molto diversi dal "percorso più breve". Il focus viene posto su altre proprietà che il percorso deve avere, come per esempio: il percorso con il maggior numero di utilizzo di trasporti pubblici; il percorso ottimale che passi da stazioni di rifornimento, in modo da non rimanere senza benzina; o come nel caso del presente progetto di tesi, il percorso che passi da luoghi ritenuti piacevoli e d'interesse, per migliorare il benessere generale del percorso.

Questo è quindi l'obbiettivo che ci si è posti nella creazione di **Geo Photo Routing**: quello di creare un servizio di *routing*, che calcoli non solo il percorso più breve, come farebbe Google Maps, ma anche un percorso secondario, alternativo, che possa aumentare il benessere di chi lo percorre, utilizzando un opportuno algoritmo, ispirato all'algoritmo di Dijkstra, per il calcolo del percorso ottimale.
Come detto in precedenza, e come verrà spiegato meglio più avanti, per rendere l'algoritmo possibile, si è fatto affidamento all' API di Flickr, per poter ottenere una grossa quantità di foto e la locazione geografica in cui sono state scattate (**geotag**). Questo perchè si presuppone che in un punto in cui è presente un elevata quantità di foto, allora questo deve essere molto probabilmente un punto piacevole da cui passare. L'algoritmo utilizzato calcola quindi il percorso cercando di transitare il più possibile da questi punti ritenuti d'interesse, e al tempo stesso cercando di non allungare troppo il percorso.

Di seguito vengono descritti alcuni ambiti di applicazione dell'algoritmo di Dijkstra e del Routing Stradale, per inquadrare il campo, il settore in cui si sta lavorando. Verranno, poi, spiegate funzionalità e implementazione di un paio di applicazioni attualmente esistenti, che implementano servizi di routing alternativi, e che hanno ispirato fortemente il presente progetto di tesi: **Happy Maps** e la prima versione di **Geo Photo Routing**.

1.1 Ambiti di applicazione dell'algoritmo di Dijkstra

L'algoritmo di Dijkstra, come detto in precedenza, è un algoritmo generico, che può essere utilizzato nei più svariati ambiti. Per orientarsi nel campo, ne vengono elencati alcuni sotto, descrivendo com è stato utilizzato l'algoritmo e quali sono state eventuali modifiche e migliorie ad esso.

1.1.1 Pianificatore di percorso per Robot

Quattro ricercatori dell'università *Universiti Sultan Zainal Abidin*, a Terengganu, in Malaysia, hanno progettato un sistema che permette a robot con movimento autonomo di muoversi all'interno di spiazi chiusi, come in una casa, seguendo un percorso ottimale. Pianificare un percorso è l'atto di trovare il percorso più corto, più adatto per un veicolo autonomo che permette a questo di muoversi da un punto ad un altro. Una volta fornita una mappa con un punto di partenza ed uno di destinazione, la pianificazione di percorso consiste nel trovare la traiettoria che il robot deve seguire per raggiungere l'obbiettivo.
Per la pianificazione del percorso, la soluzione più famosa è relativa all'algoritmo di Dijkstra. Tuttavia la struttura di memorizzazione, la matrice di adiacenza, può essere considerata non del tutto efficiente: essa limita il movimento nell'ambiente in quanto si ha bisogno di più informazioni per ogni nodo che non solo la locazione. Molti ricercatori hanno proposto diversi metodi per migliorare la struttura dell'algoritmo [5]. Per esempio si è proposto l'utilizzo di una struttura Heap per memorizzare i nodi, il che permette di minimizzare lo spazio utilizzato ed il tempo di ricerca dei nodi.
Tuttavia in questo modo l'algoritmo si complica notevolmente, per cui i ricercatori sopra citati hanno deciso di utilizzare, per la memorizzazione, un dizionario a più livelli (*multi-layer dictionary*). Hanno anche considerato la difficoltà di passare dal collegamento tra due nodi come un parametro addizionale per calcolare il percorso ottimale del robot.
Il dizionario a più livelli consiste in due dizionari ed una lista di strutture dati organizzata in ordine gerarchico. Il primo dizionario mappa ogni nodo ai suoi nodi vicini. Il secondo dizionario memorizza l'informazione sul percorso di ogni cammino per arrivare ai nodi vicini. Dato una pianta di uno spazio interno, che chiamiamo G, con due nodi vicini che chiamiamo U e V, l'espressione G[U] restituirà tutti i nodi vicini di U, mentre G[U][V] restituirà l'informazione sul percorso tra i nodi U e V.
L'esperimento consiste quindi nel paragonare il percorso ottimizzato, calcolato in questo modo, con il percorso calcolato dall'algoritmo di Dijkstra tradizionale.
Per maggiori informazioni sul progetto, consultare [4].

1.1.2 Ottimizzazione dell'itinerario in un negozio di alimentari

Un gruppo di ricercatori della *School of EECE, Mapua Institute of Technology*, hanno applicato tre algoritmi di **ricerca del cammino minimo** all'interno di un negozio di alimentari per fornire agli utenti il cammino più breve da percorrere per ottenere tutti gli oggetti di cui essi necessitano, con l'obbiettivo di trovarli più facilmente e velocemente, senza vagare all'interno del negozio alla ricerca degli alimenti. Lo studio si focalizza sull'utilizzo di tre tecniche di ricerca del cammino minimo: l'algoritmo di **Dijkstra**, l'algoritmo di **Bellman-Ford** e l'algoritmo di **Floyd-Warshall**. Esso intende aiutare il consumatore a localizzare gli oggetti da comprare, in un preciso ordine di acquisto.
Si è creato un grafo basato su una tipica disposizione di un negozio di alimentari; il peso assegnato ad ogni arco del grafico è la distanza normalizzata tra gli scaffali vicini. I ricercatori hanno utilizzato un contapassi e camminato per il negozio mentre prendevano nota della locazioni degli scaffali e dei loro vicini.
Sono stati poi usati i 3 algoritmi per calcolare percorsi di prova che passassero da determinati scaffali e si è valutata la prestazione di questi algoritmi sulla base della distanza totale del percorso calcolato e del tempo di calcolo dell'algoritmo.
La lunghezza del percorso è risultata uguale tra tutti gli algoritmi, causa la semplicità della simulazione, ma il costo computazionale si è rivelato diverso, seppur di poco, tra i vari algoritmi. Nessun algoritmo si è però rivelato migliore degli altri: il tempo computazionale dipende molto dalla quantità di prodotti selezionati.
Per maggiori informazioni sul progetto, consultare [6].

1.1.3 Altri utilizzi

Tantissimi altri sono gli utilizzi dell'algoritmo di Dijkstra, e spesso in ambiti scientifici ben più complessi di quelli appena elencati. Ad esempio si può utilizzare:

- per effettuare una simulazione dell'onda frontale di uno Tsunami [7], tramite un metodo numerico.

- per il **Ray-Tracing**, cioè per calcolare il percorso effettuato dalla luce, come viene riflessa o rifratta dalle superfici di oggetti opportunamente collocati nello spazione, con una determinata forma. [8]

- per trovare il percorso più breve di una rete ottica, ma anche in un qualsiasi altro tipo di rete. [9]

Questi e molti altri gli ambiti di applicazione. Si nota che spesso, quando viene utilizzato, l'algoritmo di Dijkstra presenta sempre qualche variante o miglioria per adattarlo al meglio all'ambito di applicazione.

1.2 Routing Stradale

Abbiamo appena visto alcuni ambiti di applicazione dell'algoritmo di Dijkstra. Ci concentriamo, ora, sull'applicazione di esso nel routing stradale, quindi sulla possibilità di creare percorsi personalizzati all'interno di una rete stradale visualizzata su una mappa geografica. Di seguito si forniscono alcuni esempi di applicazioni o progetti esistenti nel campo del routing stradale.

1.2.1 Taxi Tracking and Routing

Un gruppo di ricercatori hanno ideato un'applicazione utile per capire la posizione in tempo reale dei taxi e per fornire uno di essi all'utente nel minor tempo possibile, utilizzando l'algoritmo di Dijkstra [10].
Il sistema può risultare utile, soprattutto per i passeggeri che aspettano molto tempo per un taxi che hanno prenotato, ma anche per i taxi che possono risparmiare carburante, in quanto verrà chiamato il tassista più vicino e non quello dall'altra parte della città.
Si riscontra spesso un problema quando si ordina un taxi e si deve aspettare a lungo il suo arrivo. Questo di solito avviene durante le ore di punta quando c'è molto traffico e il taxi rimane bloccato nel traffico. Inoltre accade spesso che il tassista deve andare molto lontano per prendere il passeggero, causando un aumento dei costi della benzina.
L'obbiettivo principale della ricerca è la costruzione di un sistema capace di fornire un servizio automatico al passeggero. La posizione e velocità di ogni taxi sarà monitorata grazie all'utilizzo di **GIS (Geographic Information System)**. Per fornire automaticamente la prenotazione di un taxi, che sia il più adatto per il passeggero chiamante, secondo i parametri di traffico e distanza del taxi, si utilizza l'algoritmo di Dijkstra. In questo modo il passeggero ordina un taxi che ci metterà il minor tempo possibile ad arrivare. Ovviamente anche per il tassista l'applicazione crea un benefit, dal momento che non deve utilizzare molta benzina per recarsi dal passeggero.

Il sistema funziona grazie all'utilizzo ai ricevitori GPS Android che sono installati su ogni veicolo. Questi vengono chiamati *OBU (OnBoard Unit)*. Inoltre è presente un server che riceve tutte le informazioni sulla posizione dei ricevitori GPS e le farà visualizzare all'u-

tente. I dati trasmessi dai GPS sui taxi vengono memorizzati sul database e quando i passeggeri effettuano l'accesso all'applicazione, il sistema mostrerà la posizione di ogni taxi disponibile, in base alla latitudine e longitudine dei dati memorizzati sul database. Viene quindi monitorata in tempo reale la posizione di ogni veicolo, ma anche il livello e la densità del traffico.
Quindi, perchè sia possibile trovare il veicolo che meglio risponde alle necessità del passeggero, il sistema utilizza l'algoritmo di Dijkstra. L'algoritmo utilizzerà come parametri la posizione del veicolo ed il traffico sulla strada per trovare il veicolo più adatto.
Una volta trovato il taxi, il sistema manderà automaticamente informazioni riguardo la posizione al futuro passeggero.

1.2.2 Naviki

Sviluppati dall'azienda tedesca "beemo GmbH", l'app e il sito web Naviki forniscono un servizio di routing che permette di pianificare un percorso, specificatamente per biciclette, in tutto il mondo e forniscono una documentazione completa dell'attività in bicicletta.
Naviki offre diverse tipologie di percorsi, tra cui si può scegliere a seconda del tipo di strada che si vuole percorrere:

- Per attività quotidiane: percorsi ciclabili confortevoli e veloci;
- Per il tempo libero: percorsi ufficiali per escursioni in bicicletta in ambienti piacevoli;
- Per le bici da corsa: superfici lisce per pedalate rapide, se possibile su strade secondarie;
- Per le mountain bike: percorsi offroad, boschi e itinerari di campagna, superfici sterrate;
- Percorso più breve: per arrivare velocemente sfruttando i collegamenti più brevi possibili;
- S-pedelec: percorsi personalizzati per bici elettriche veloci fino a 30 mph.

Una volta scelto il tipo di percorso, l'indirizzo di partenza e di destinazione, viene mostrato immediatamente l'itinerario su una mappa e le istruzioni di navigazione che permettono di navigare verso il punto di arrivo.

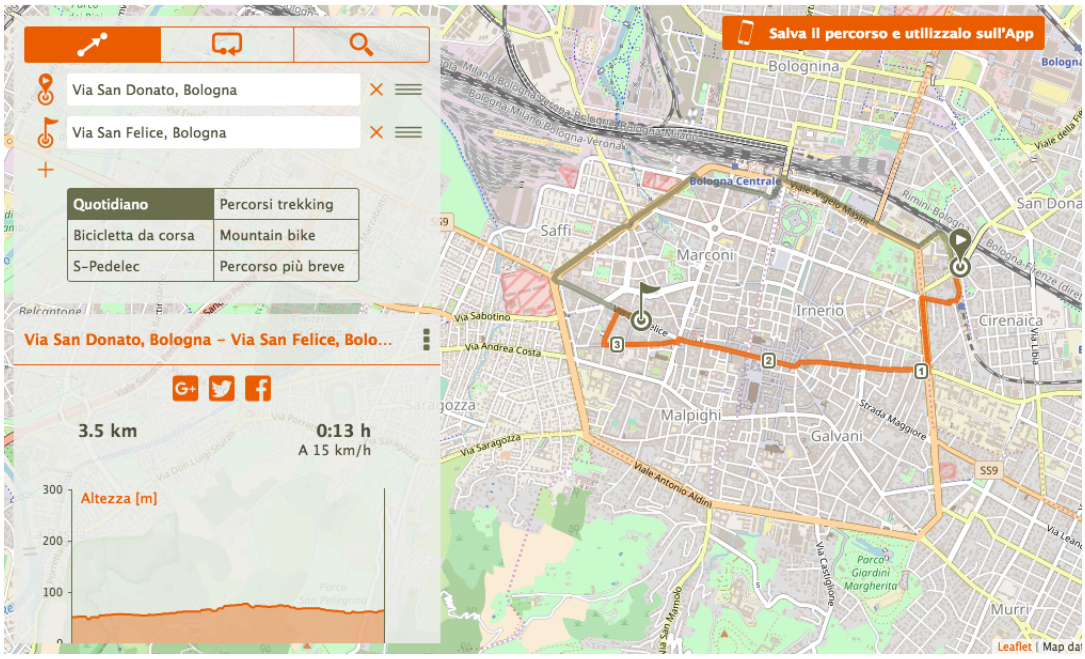

Figura 1.1: Esempio di pianificatore di percorso del sito `www.naviki.org`

In particolare, Naviki mette a disposizione le seguenti funzionalità:

- **Pianificatore di percorso**: è la funzionalità principale. Consiste nella possibilità di ricercare un percorso indicando il punto di partenza, di arrivo e la tipologia di strada da percorrere, come indicato precedentemente. Si ottiene il percorso personalizzato con le informazioni aggiuntive più importanti;
- **Round-trip**: rappresenta una seconda modalità di creazione di un itinerario. Se si vuole fare un giro in bici, tornando esattamente al punto di partenza, questa opzione permette di creare un percorso adatto, scegliendo il tipo di strada da percorrere, l'indirizzo di partenza e il numero di chilometri che si vuole percorrere.
- **Navigation turn-by-turn**: si intende le istruzioni vocali di navigazione e frecce direzionali sul display. Le deviazioni spontanee da un percorso pianificato non rappresentano un problema in quanto l'applicazione ricalcola automaticamente il nuovo percorso fino alla destinazione;
- **Mappe offline**: possibilità di scaricare sul proprio smartphone mappe Naviki di alta qualità per poi usarle indipendentemente dall'accesso a Internet;

- **Profilo altimetrico**: per ogni percorso viene fornito un profilo altimetrico, cioè i punti più alti e più bassi dell'intero percorso e l'altitudine generale in ogni punto del percorso, attraverso un grafico;
- **Cockpit**: come in una cabina di pilotaggio, l'applicazione mostra velocità, velocità media, distanza percorsa e distanza dal punto di arrivo. Durante la navigazione il contakilometri viene visualizzato automaticamente;
- **Collegamento dispositivi fitness**: collegando tramite Bluetooth i dispositivi per il fitness a Naviki, è possibile registrare il ritmo delle pulsazioni ed altri dati durante i percorsi e visualizzarli successivamente sotto forma di grafico;
- **Registrazione dei percorsi**: la registrazione e la memorizzazione di un percorso in bici sul cloud personale. In questo modo si possono pianificare comodamente dal sito web i percorsi e successivamente utilizzarli tramite l'app mobile.

1.3 Lavori Correlati

Tra le applicazioni di routing stradale esistenti, alcuni lavori particolari hanno particolarmente ispirato il presente progetto di tesi. Vediamone le caratteristiche.

1.3.1 Happy Maps

Questo che si descrive ora è il progetto che ha ispirato la creazione di Geo Photo Routing, quindi molte caratteristiche della piattaforma sono state rielaborate a partire da Happy Maps.

Solitamente un servizio di routing, su web o su mobile, nel fornire indicazioni per un luogo, sono in grado di suggerire il percorso più breve tra i punti di partenza e di arrivo. L'obiettivo del progetto **Happy Maps**, idea di tre sviluppatori italiani, è quello di suggerire automaticamente i percorsi che sono non solo brevi ma anche **emotivamente piacevoli**.
A volte non vogliamo prendere il percorso più veloce, ma goderci quelli alternativi che offrono scenari meravigliosi, o un ambiente più tranquillo. Tuttavia, i servizi di mappatura Web e mobili attualmente non riescono a offrire questa esperienza. Happy Maps si propone allora di riuscire in questo intento.

Il progetto fa parte di un programma più grande, **Good City Life**[1], che è un gruppo globale di ricercatori e professionisti che pensa a risolvere problemi fondamentali degli spazi urbani che hanno sempre ricevuto poca attenzione. Cerca di rendere le città non più intelligenti, ma più *felici*. Altri progetti di Good City Life sono per esempio **Chatty Maps** o **Smelly Maps**, rispettivamente per ricavare percorsi cittadini che diminuiscano l'inquinamento acustico ed olfattivo.

Happy Maps utilizza metodi di *crowdsourcing*[2], immagini geotaggate e metadati associati per costruire una cartografia alternativa di una città pesata per le emozioni umane. Le persone hanno più probabilità di scattare fotografie di edifici storici, tratte distintive e vie piacevoli invece di strade principali infestate da auto.
Oltre a ciò, Happy Maps adotta un algoritmo di routing che suggerisce il percorso più breve tra due posizioni, massimizzando il guadagno emotivo. Quella piacevole deviazione, che potrebbe allungare di un paio di minuti il percorso più breve, potrebbe provocare un'esperienza di passeggiata completamente diversa.
Happy Maps cerca quindi di contribuire a cambiare il modo in cui i prodotti di ingegneria vengono disegnati: spesso viene fatto ricercando il concetto di *efficienza*, ma l'essere più efficienti non ci rende necessariamente più *felici*.

L'implementazione di un percorso breve e allo stesso tempo piacevole tra la posizione attuale dell'utente (S) e la destinazione prevista (D), può articolarsi in quattro fasi [11]:

1. **Creazione di un grafico i cui nodi sono tutti i luoghi della città in questione**: si divide il riquadro della città in questione in celle di dimensione 200x200 metri ciascuna, ognuna delle quali sarà considerata come nodo di un grafico di posizione. Ogni nodo è collegato con i suoi otto nodi geografici vicini.
2. **Percezione delle posizioni in tre dimensioni: *bellezza*, *tranquillità* e *felicità***: si utilizzano i dati raccolti da un sito web di *crowdsourcing* per valutare la misura in cui le diverse località della città sono percepite sotto le tre dimensioni. Disponibile all'indirizzo `UrbanGems.org`, il sito raccoglie due scene urbane casuali, ottenute tramite **Google Street View**, e chiede agli utenti quali delle due è più bella, silenziosa e felice. Ad ogni turno, l'utente deve selezionare una delle due scene o scegliere "Non riesco a dire", se è indeciso sull'immagine da cliccare. Dopo ogni selezione, viene chiesto all'utente di indovinare la percentuale di altre persone che hanno risposto allo stesso modo, accumulando punti per le risposte corrette.

[1] `www.goodcitylife.org`

[2] Il crowdsourcing (da crowd, "folla", e sourcing, da outsourcing, cioè esternalizzazione aziendale) in un'azienda, è l'affidare a persone esterne, tramite gli strumenti offerti dal web, l'ideazione o la realizzazione di progetti, la raccolta e l'analisi di dati e informazioni.

La scelta delle tre qualità è motivata dalla loro importanza nel contesto urbano. Essere in grado di trovare posti *tranquilli* potrebbe promuovere la salute sonica nelle varie città e offrire una guida pubblica a chi desidera un ritiro dalla folla. Per quanto riguarda la *bellezza*, non è la prima volta che si misurano le sue percezioni: un'analisi quantitativa delle percezioni pubbliche dell'apparenza visiva del quartiere ha scoperto che la bellezza e la sicurezza sono solitamente collegate. Infine, si è scelta la *felicità*, non da ultima, perchè gli studi urbani degli anni '60 tentavano di rapportare il benessere nell'ambiente urbano (cioè la felicità) al desiderio fondamentale dell'ordine visivo, della bellezza e dell'estetica.

3. **Assegnazione di punteggi alle posizioni per ciascuna delle tre caratteristiche**: per classificare un luogo, è necessario calcolare la probabilità che questo possa essere visitato perchè piacevole. Per esprimere ciò, prendendo come esempio la felicità, si è calcolata la probabilità $p(happiness — go)$, che rispecchia l'idea che gli individui visitino luoghi che li rendano felici. Per misurare la felicità di una posizione si ricorre ai punteggi di crowdsourcing del punto precedente. Più specificatamente, dato il punteggio di felicità h_i per la cella i, si calcola la corrispondente probabilità di felicità con una curva che ad esempio è cubica:

$$p(happiness|go) = k \cdot h_i^3, \text{ dove } k = \frac{1}{max\{h_i^3\}\forall i} \tag{1.1}$$

Si applica poi la stessa funzione per ricavare la probabilità relativa alle due dimensioni restanti, *tranquillità* e di *bellezza*, sostituendo h_i con q_i e successivamente con b_i, ovvero il punteggio di tranqullità e di bellezza della posizione i.

4. **Selezione del percorso tra i nodi S e D che soddisfa il giusto equilibrio tra l'essere breve e piacevole**. Avendo il grafico di posizione e la probabilità di visitare ciascuna località, è possibile scegliere il percorso migliore dall'origine S alla destinazione D, in quattro passi:

   - **Step 1**: Identificazione dei percorsi più brevi tra S e D. Applicando l'algoritmo di Eppstein, è possibile scoprire gli M percorsi più brevi che collegano ciascuna coppia di nodi S e D. Per essere sufficientemente "esaurienti", inizialmente M viene impostato ad un valore molto alto. Questa scelta rende possibile esplorare l'intera gamma di soluzioni.
   - **Step 2**: Calcolo del punteggio medio per tutte le posizioni in ciascuno dei primi m percorsi (con $m \leq$ M). Non si considerano tutti gli M percorsi contemporaneamente, ma vengono esplorati iterativamente i primi m percorsi. Ad ogni esplorazione, si registra il percorso con il punteggio medio minore, cioè il percorso migliore. Si può dimostrare che più percorsi esploriamo (più

alto è m), minore diventa la probabilità di trovare che il punteggio migliore: per questo non è necessario esplorare tutti gli M percorsi.

- **Step 3**: Si smette di calcolare i punteggi medi degli m percorsi quando il punteggio medio migliora meno di ϵ. Per selezionare ϵ, si applica il **Teorema del valore marginale (MVT)**. Si tratta di uno strumento analitico per ottimizzare il trade-off tra il beneficio (miglioramento del punteggio Δ punteggio) ed il costo (esplorazione dei primi m percorsi). Si può dimostrare, che per la funzione di punteggio rispetto a m, è meglio continuare ad aumentare m finchè $\frac{\Delta punteggio}{\Delta m}$ corrisponde a $\frac{punteggio}{m}$; dopo di che si dovrebbe terminare e scegliere il percorso, tra quelli considerati, con il miglior punteggio medio.
- **Step 4**: Selezione del percorso trovato per aver il miglior punteggio. Ripetendo i tre passaggi precedenti per ciascuna delle tre dimensioni, si ottengono tre percorsi tra i nodi S e D oltre a quello base, cioè quello più breve.

Per accertarsi che l'obiettivo è stato raggiunto, gli sviluppatori, attraverso uno studio che ha coinvolto più utenti, hanno dimostrato che le varianti di percorso proposte soddisfano appieno le loro aspettative (ad esempio, scegliere un percorso perchè il grado di felicità percepito risulta maggiore) aggiungendo solo pochi minuti di cammino in più rispetto agli itinerari più brevi.

Si possono sollevare alcune critiche al servizio: per esempio esso si basa su dati di crowdsourcing la cui raccolta richiede un forte coinvolgimento degli utenti, ed in secondo luogo, il progetto è stato testato solo in una città, Londra; espanderlo ad altre città potrebbe non risultare immediato.

1.3.2 Geo Photo Routing 1.0

Il presente progetto di tesi è la prosecuzione e il miglioramento di un progetto già esistente: **Geo Photo Routing** [12].
Come detto in precedenza, questo è un progetto nato per dare la possibilità di fornire un itinerario di viaggio, che non solo sia il più breve possibile, ma anche più emotivamente piacevole e rilassante, un po' come succede nel caso di Happy Maps.
Il progetto iniziale aveva come obbiettivo lo sviluppo di un' **applicazione web**, che utilizzasse, per suggerire l'itinerario, immagini geotaggate ottenute dai servizi di condivisione foto, in particolare Flickr. Le geolocazioni fotografate possono essere buone raccomandazioni per trovare luoghi interessanti, in quanto il caricamento di una fotografia sul social network può essere considerato come un voto positivo e, di conseguenza, come una località da cui vale la pena passare.

L'applicazione web prevedeva una semplice interfaccia grafica, dove poter inserire il punto di partenza, quello di arrivo e un raggio d'azione, dopodiché la piattaforma si sarebbe occupata di calcolare e visualizzare sulla mappa sia il percorso più breve, quello di base calcolato da Google Maps, sia il percorso con maggiori punti d'attrazione.

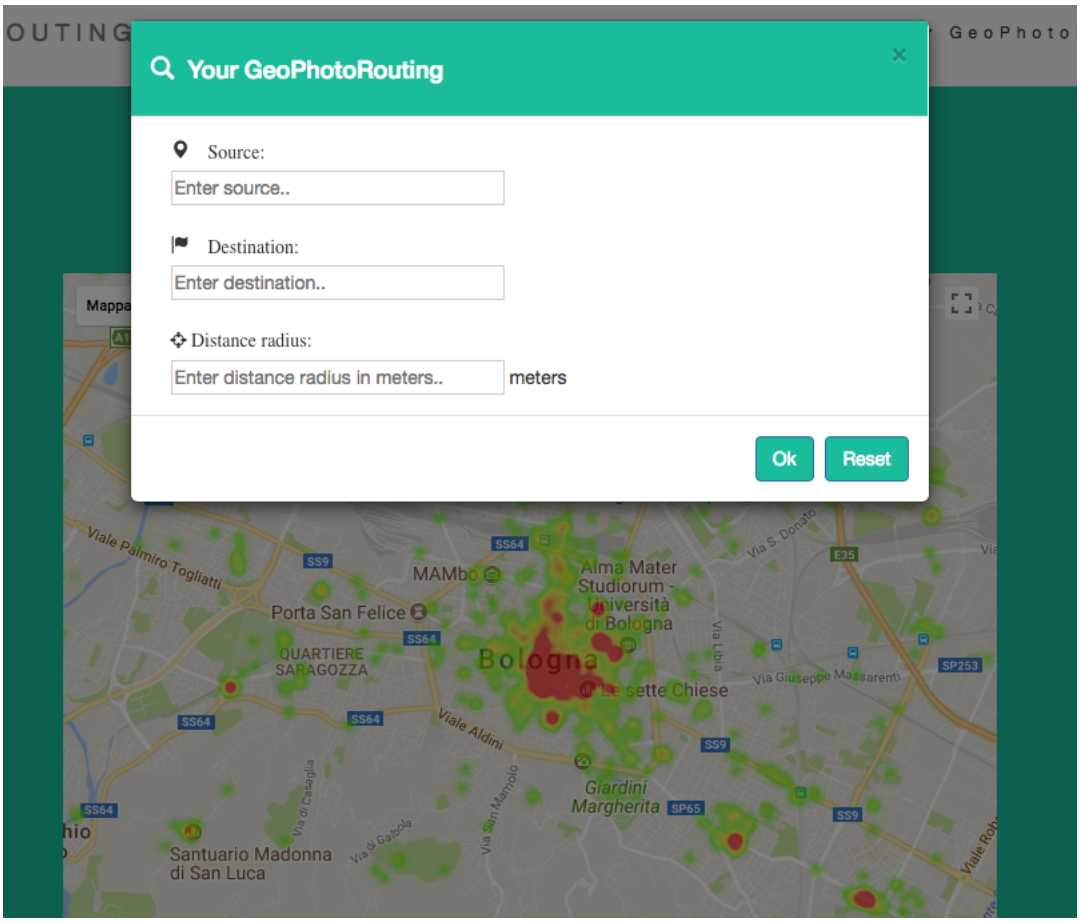

Figura 1.2: Schermata di inserimento percorso nella prima versione di Geo Photo Routing.

Le caratteristiche implementative della piattaforma erano:

- L'utilizzo di un'architettura *three-tiers*, cioè a tre livelli: **Client**, **Server**, **Database**

- L'utilizzo della piattaforma software **MAMP**, quindi l'utilizzo di un sistema operativo **MacOS**, un server **Apache**, un DBMS **MySQL** e un linguaggio Server-side, come **PHP**.
- Utilizzo di HTML, CSS, Javascript ed apposite librerie per implementare l'applicazione Web Client-side
- L'utilizzo di API di Google, per la creazione dei percorsi (**Google Directions**) e per i suggerimenti durante l'inserimento dell'indirizzo da parte dell'utente (**Google Place Autocomplete**)
- L'utilizzo di librerie di Google per la visualizzazione e modifica della mappa, tra cui la possibilità di aggiungere *Marker* ad essa, disegnare percorsi e visualizzare *Layer* alternativi tra cui il **Layer Heatmap**
- L'utilizzo dell'**API Flickr** per l'ottenimento delle foto geotaggate, della relativa posizione e dei relativi tag. Dati che sono stati poi salvati sul Database proprio di Geo Photo Routing.

Per il calcolo del percorso secondario, quello più emotivamente piacevole, si faceva affidamento ad un breve algoritmo sviluppato ad hoc per l'applicativo. La modalità con cui veniva calcolato il percorso consisteva in vari punti:

1. Inviare una richiesta a Google Directions per l'ottenimento del percorso più breve tra il punto di partenza ed il punto di fine;
2. Memorizzare poi i punti intermedi che componevano il percorso
3. Cercare i punti di interesse ad una certa distanza da ogni punto intermedio. I punti di interesse venivano riconosciuti tali in base alla quantità di foto (scaricare dall'API Flickr) scattate.
4. Per ogni punto intermedio, se esisteva un punto d'interesse nelle vicinanze, quest'ultimo lo sostituiva nella creazione del percorso
5. Veniva così generato un secondo percorso che passasse dai punti intermedi di Google Maps e dai punti di interesse appena ricercati.

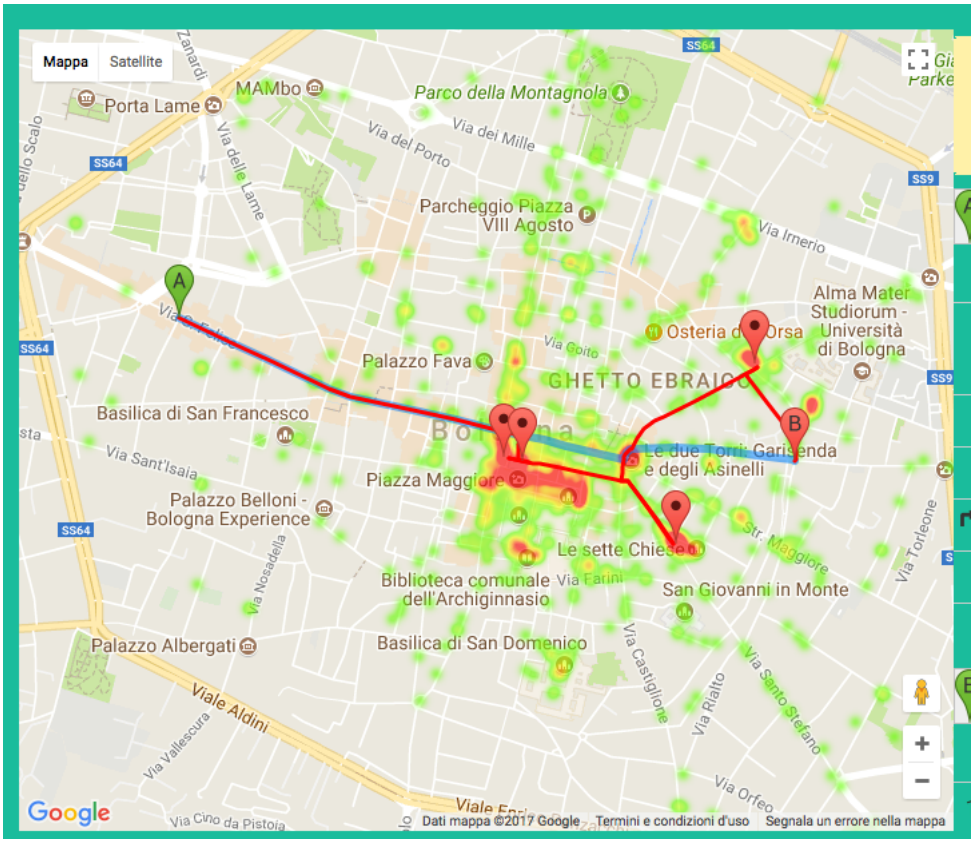

Figura 1.3: Schermata di visualizzazione del percorso nella prima versione di Geo Photo Routing.

In questo modo si otteneva un percorso secondario che allungava di un poco la lunghezza dell'itinerario ma che fornisse all'utente un'esperienza completamente diversa, più piacevole.
Perchè l'algoritmo funzionasse, era necessaria però la presenza sul Database di immagini geotaggate, la cui posizione ricadeva all'interno del percorso ricercato. Se sul Database non erano memorizzate foto nella zona ricercata dall'utente, quest'ultimo aveva il potere e la possibilità di aggiornare il Database in modo sicuro, tramite un'apposita pagina dell'applicazione Web, la quale avrebbe inserito automaticamente nel Database le foto presenti all'interno dell'area geografica desiderata. Spettava, quindi, agli utenti stessi aggiornare il Database per nuove aree geografiche.

Il presente progetto di tesi si basa fortemente sullo sviluppo dell'applicativo appena descritto, ma cerca di migliorarlo in molti punti, cambiando e rimuovendo funzionalità,

scrivendo nuovo codice, aggiungendo nuove piattaforme di visualizzazione. In particolare, ci si concentrerà su 3 principali punti di cambiamento:

- Creazione di un applicazione Android, Client-side, che abbia tutte le funzionalità dell'applicazione Web sopra descritta (possibilità di creare e visualizzare percorsi e layer Heatmap), più qualche funzionalità aggiuntiva utile all'utente.
- Modifica della modalità di salvataggio delle foto dall'API Flickr al Database di Geo Photo Routing. Come già spiegato, nella prima versione della piattaforma, l'aggiornamento del Database era interamente lasciato all'utente, ma nel presente progetto verrà eliminata questa possibilità, sostituendola con uno script PHP che automaticamente, ogni giorno, mantiene il Database aggiornato su determinate aree geografiche.
- Modifica dell'algoritmo di ricerca dei punti di interesse usati per la creazione del percorso secondario. L'algoritmo sopra spiegato presenta qualche piccolo problema e può essere notevolmente ampliato. Si è quindi cercato di dare una soluzione migliore al problema della creazione del percorso più emotivamente piacevole.

La descrizione di questi tre cambiamenti e dell'intero progetto, nei dettagli, avverrà nei prossimi capitoli.

Capitolo 2

Progettazione

In questo capitolo, dopo una breve descrizione del progetto di tesi, vengono elencate le funzionalità dell'applicazione mobile; descritta l'architettura del sistema; spiegate quali sono le tecnologie utilizzate e perchè è stata effettuata tale scelta e descritto il ragionamento che ha portato alla scelta dell'algoritmo di ricerca utilizzato.

2.1 Descrizione

Il progetto di tesi consiste nello sviluppo di un'applicazione Android che dà la possibilità di creare itinerari stradali, da un qualsiasi punto di partenza, ad uno qualsiasi di arrivo. L'applicazione mostra, oltre al percorso più breve tra i due punti, anche un percorso secondario, che è solitamente leggermente più lungo del primo, ma che contribuisce ad un maggior benessere nel percorrerlo, passando da punti "caldi", di interesse, ritenuti piacevoli da oltrepassare. Quella piacevole deviazione, che potrebbe allungare di un paio di minuti il percorso più breve, potrebbe provocare un'esperienza di passeggiata completamente diversa.
La ricerca di questo percorso secondario, più emotivamente piacevole, è possibile grazie allo sviluppo di un algoritmo server-side per la ricerca del cammino minimo, che si basa sull'algoritmo di Dijkstra. L'algoritmo sfrutta un grafo pesato ed orientato, in cui i nodi sono in realtà *cluster* (raggruppamenti) di immagini geotaggate memorizzate sul Database e raggruppate tramite uno standard di riferimento militare, che suddivide il mondo in una griglia (MGRS). Le suddette immagini sono scaricate quotidianamente da Flickr tramite uno script server-side, utilizzando l'apposita API Flickr.

2.2 Funzionalità

In questa sezione vengono descritte tutte le funzionalità offerte all'utente dall'applicazione Android.
All'avvio dell'applicazione viene mostrata una Heatmap del territorio Italiano, come mostrato in Figura 2.1. Al momento sono state scaricate sul Database solo immagini relative a quattro importanti città (Torino, Milano, Bologna e Roma), ma l'obbiettivo è quello di ottenere le immagini relative a tutto il territorio Italiano.

Figura 2.1: Heatmap del territorio italiano.

Una Heatmap è un layer offerto da Google Maps, cioè uno strato da applicare alla mappa di Google, che rappresenta visualmente l'intensità geografica dei dati nella mappa di riferimento. Nel nostro caso i dati sono rappresentati dai geotag delle immagini presenti sul Database.
Il layer heatmap quindi, mostra il luogo geografico in sono state scattate le foto presenti sul Database, utilizzando diverse gradazioni di colore nella mappa: le aree di maggiore intensità di foto sono colorate di rosso mentre quelle di intensità inferiore appaiono in verde.

Dalla schermata principale è possibile accedere ad un menù laterale, che reindirizza a tutte le funzionalità dell'applicazione:

- **Calculate your GeoPhoto Route**, per accedere alla funzione principale di routing, cioè la possibilità di ricercare un percorso tra 2 punti che ottimizzi la funzione di soddisfazione;
- **Search by Keyword**, per visualizzare una heatmap, riguardante solo le foto che abbiano un determinato tag, inserito da utente;
- **Show Heatmap**, per visualizzare la heatmap iniziale, riguardante tutte le foto sul Database;
- **My Routes**, per visualizzare la lista degli itinerari precedentemente salvati.

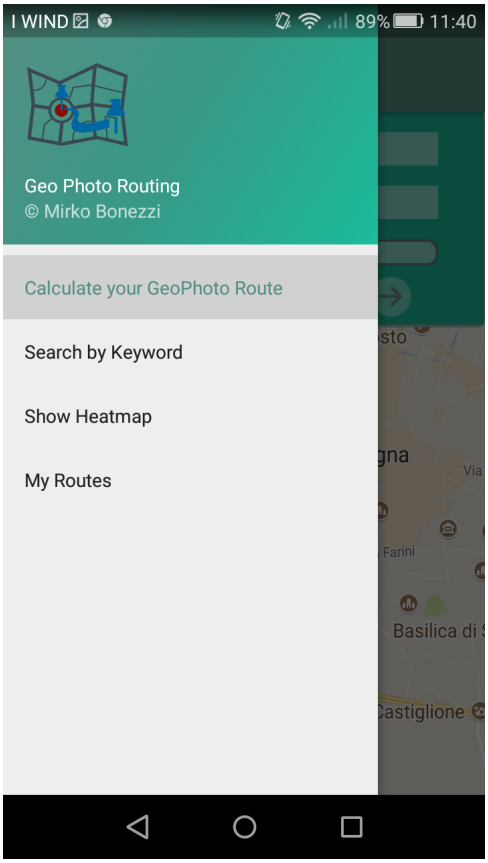

Figura 2.2: Menù principale.

Selezionando la prima voce del menù, "**Calculate your GeoPhoto Route**", appare un nuovo layer di inserimento dati, come in figura 2.3, in cui l'utente ha la possibilità di inserire un indirizzo di partenza ed uno di arrivo, scegliendolo tramite il servizio di **Google Place Autocomplete**, un servizio che suggerisce una lista di indirizzi, variabile durante l'inserimento, mostrando quelli che l'utente sta più probabilmente cercando.
Nel form di inserimento, si può inoltre specificare la volontà di muoversi in macchina o a piedi e scegliere il raggio di ricerca.
Quest'ultima opzione è un parametro utilizzato nell'algoritmo di ricerca del percorso secondario: indica il raggio di ricerca di punti di interesse, specificato in metri, ovvero la massima deviazione del percorso secondario da quello originale.

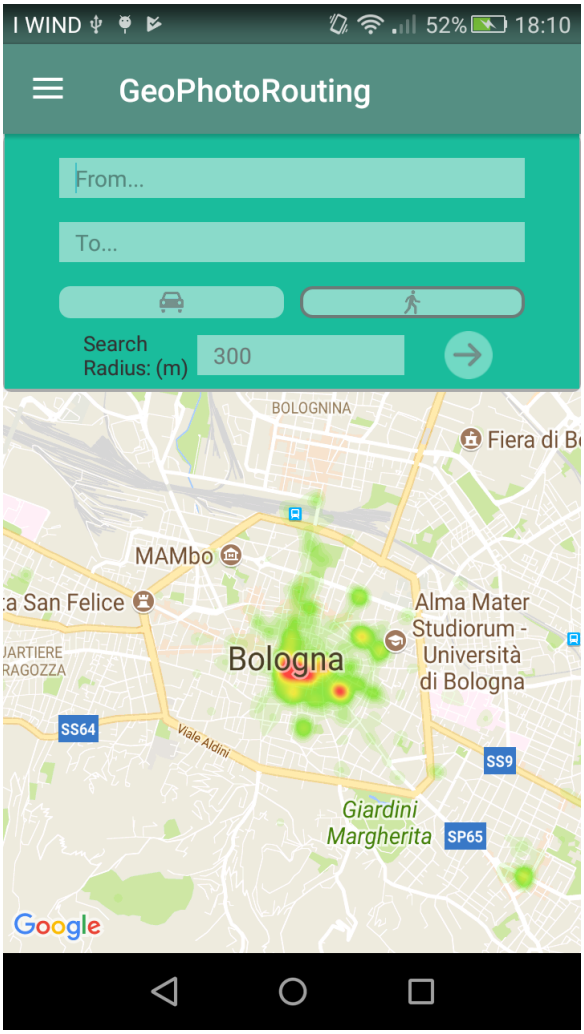

Figura 2.3: Form per ricercare un percorso.

Una volta ricercato un itinerario, compaiono sulla mappa due diversi percorsi: il percorso in blu indica il percorso più breve, quello suggerito da Google Maps; mentre il percorso in rosso indica il percorso secondario prodotto dall'algoritmo di ricerca, quel percorso che propone una piccola deviazione per poter rendere migliore l'esperienza del viaggio.
Come si nota dalla figura 2.4, altre opzioni sono comparse nella schermata.
Innanzitutto, sotto il form di inserimento dati, è rappresentato un semplice confronto tra i due percorsi: la distanza da percorrere ed il tempo previsto di percorrenza del percorso primario, sulla sinistra; e gli stessi dati relativi al percorso secondario, sulla destra. Inoltre, due pulsanti circolari portano a nuove funzionalità: il pulsante in basso

a sinistra permette di visualizzare testualmente le indicazioni per il tragitto del percorso secondario; mentre il pulsante in basso a destra permette il salvataggio del percorso appena ricercato, per visualizzarlo più facilmente in futuro.

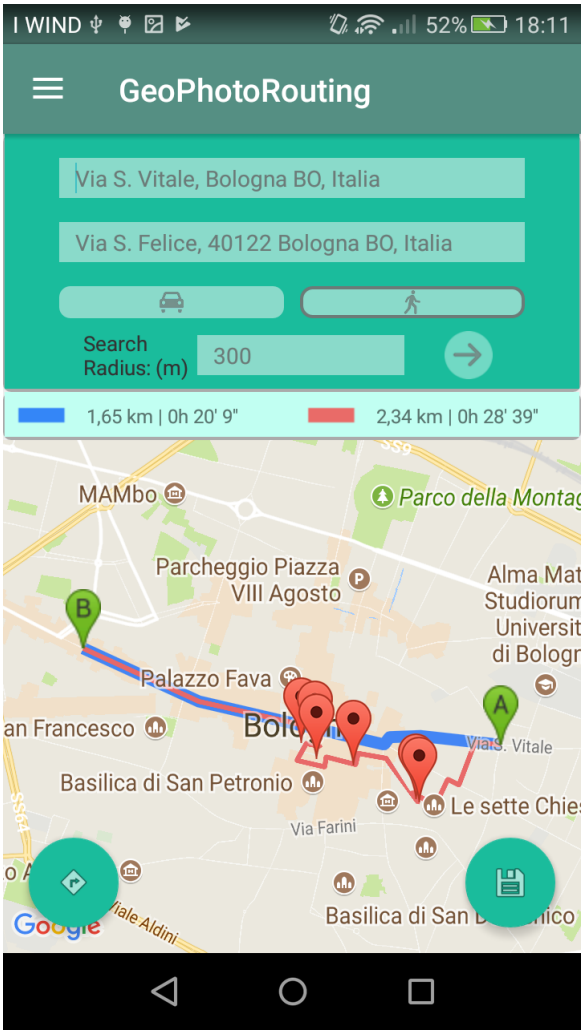

Figura 2.4: Visualizzazione dell'itinerario calcolato.

Tornando al menù principale e selezionando la seconda voce, "**Search by Keyword**", appare una heatmap corredata da un campo di inserimento testo, in cui si può ricercare una qualsiasi keyword. Questo permette di visualizzare una heatmap riguardante solo le foto che abbiano un tag contenente quella determinata keyword.

Figura 2.5: Heatmap di Bologna riguardante la keyword "Nettuno".

La voce "**Show Heatmap**" riporta semplicemente alla prima schermata, visualizzando una heatmap riguardante tutte le foto presenti sul Database.

L'ultima voce, "**My Routes**", permette invece di visualizzare una lista di itinerari salvati precedentemente, in modo da poterli consultare più rapidamente.
Se si vuole cancellare un itinerario, è sufficiente tenere premuto uno dei percorsi ed apparirà una finestra di dialogo che chiederà conferma sull'eliminazione.

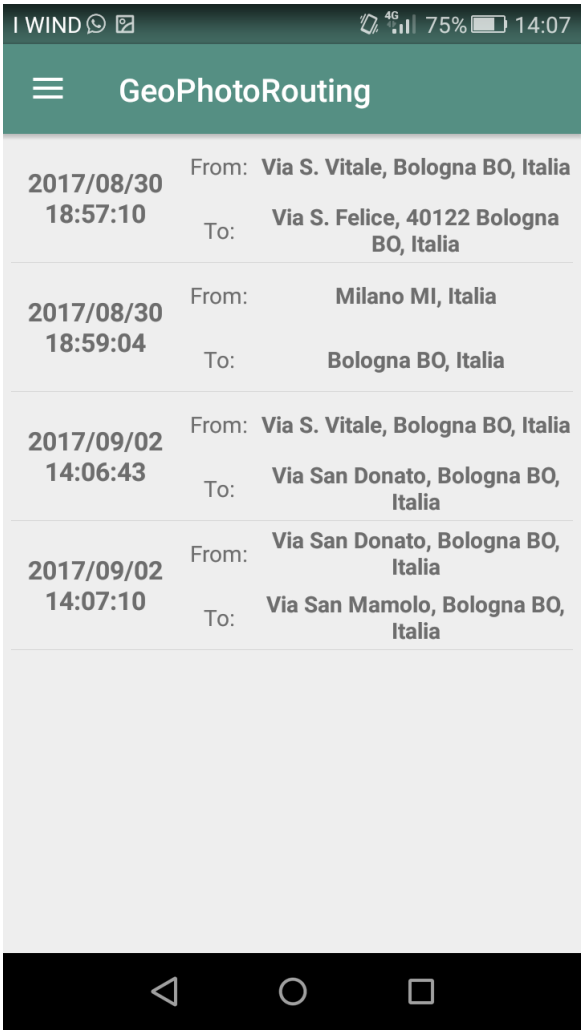

Figura 2.6: Lista di itinerari salvati.

2.3 Architettura di Sistema

Il progetto di tesi è stato sviluppato con un'*Architettura a Tre Livelli* (o Three-Tier) [13], le cui fondamenta sono costituite da un importante modello: il modello **Client-Server** [14].

La linfa vitale di quest'ultimo è il dialogo tra terminali Client ed un Server, un terminale

solitamente molto potente in termini di capacità d'entrata-uscita, che fornisce loro dei servizi. Il client effettua una richiesta ad un server e questo, attraverso un linguaggio di scripting (come il PHP) interpreta la richiesta ed invia una risposta al client, spesso sotto forma di pagina HTML, JSON o XML. In questo modo il client è in grado di interpretare la risposta ricevuta e fornirla all'utente.

L'architettura a tre livelli è strutturata, appunto, su tre diversi livelli logici:

- **Livello di presentazione (Client)**: A questo livello è legata la responsabilità per la presentazione grafica e l'interazione dell'utente. Si tratta di un qualsiasi tipo di sistema front-end, quale può essere un applicazione Web, una GUI (Graphic User Interface), o come nel nostro caso, un'applicazione Android. Il Client si occupa, quindi, di ricevere degli input dall'utente e di inoltrarli al secondo livello, cioè alla logica dell'applicazione, e non parlerà mai direttamente con il livello dei dati.

- **Logica dell'applicazione (Server)**: Il secondo livello si occupa di gestire la logica dell'applicazione, smistando le richieste ricevute dal livello Client, eseguendo opportune elaborazioni e richiamando i corretti servizi del terzo livello. Il livello logico dell'applicazione è il punto cruciale dell'architettura perchè funge da collo di bottiglia incanalando tutte le richieste dei diversi Client, ed esegue l'elaborazione necessaria ad ogni richiesta, eventualmente accedendo al Database, in modo da poter restituire una risposta al Client.
  I Client possono effettuare le proprie richieste anche contemporaneamente, quindi il livello logico dell'applicazione deve gestire correttamente anche la concorrenza, per evitare di creare inconsistenza sul Database.
  Nel presente progetto di tesi, questo livello è costituito da una serie di pagine PHP presenti sul Server.

- **Livello dei dati (Database)**: Questo livello è costituito da una o più basi di dati, gestori risorse e applicazioni mainframe. La base di dati è progettata per memorizzare nel migliore dei modi tutti i dati per la realizzazione dell'applicazione. Questo livello si trova all'interno di una rete protetta, in modo che solo i processi del secondo livello possano accedervi.
  Nel progetto di tesi, questo livello è formato da un database relazionale realizzato utilizzando MySQL.

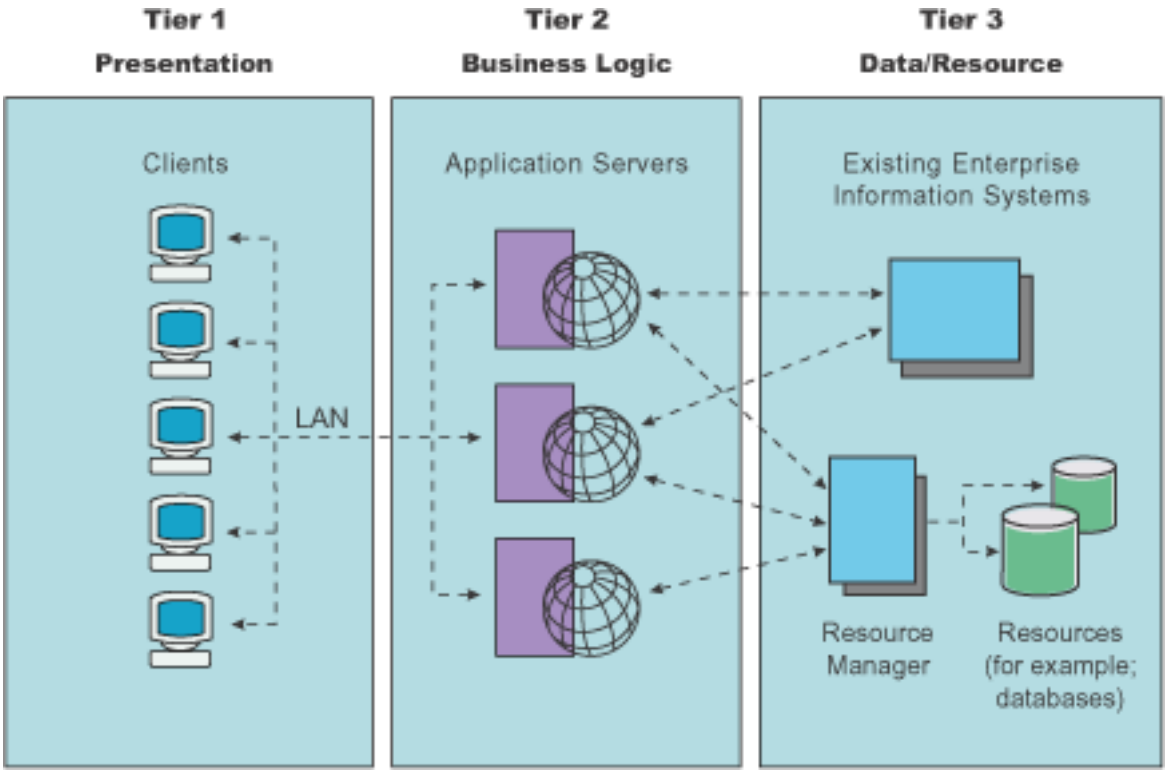

Figura 2.7: Architettura a tre livelli.

2.4 Tecnologie Utilizzate

In questa sezione vengono descritte nel dettaglio le diverse tecnologie utilizzate per la realizzazione del progetto di tesi, su tutti i livelli dell'architettura: Client, Server e Database e qual'è la motivazione che ha spinto alla scelta di tali tecnologie.

2.4.1 Client

Il livello di presentazione è costituito da un'applicazione **Android**, realizzata tramite l'utilizzo dell'IDE **Android Studio**. Il linguaggio di programmazione utilizzato per la logica dell'applicazione è ovviamente **Java**, mentre per la parte presentazionale è stato utilizzato il linguaggio di markup **XML**.

Nello sviluppo dell'applicazione sono state anche utilizzate le librerie di Google Maps per poter visualizzare e interagire con le mappe Google su Android. Si è visto inoltre necessario l'utilizzo di un paio di **API Google**, per l'esattezza l'API **Google Directions** per aver la possibilità di creare un itinerario e mostrarlo sulla mappa e **Place Autocomplete** per sfruttare i suggerimenti forniti da Google durante l'inserimento degli indirizzi stradali da parte dell'utente.

Android Studio

Android Studio è l'**IDE (Integrated Development Enviroment** integrato ufficiale per lo sviluppo di applicazioni Android, basato sul software **IntelliJ IDEA**[1].

Rilasciato ufficialmente nel 2014, Android Studio offre molte caratteristiche che aumentano la produttività e la fruibilità del codice durante lo sviluppo di applicazioni Android. Per esempio, un ottimo sistema di *build automation*, **Gradle** [15], per ottimizzare la fase di costruzione del progetto ed aiutare gli sviluppatori in ogni fase del processo di sviluppo del software: collaudo, pubblicazione e la distribuzione finale del software.
Altre caratteristiche importanti sono: un ricco emulatore, decorato di molte features; un ambiente di sviluppo unificato in cui puoi facilmente sviluppare applicazioni per tutti i dispositivi Android; possibilità di integrare facilmente codice da GitHub ed integrare framework e librerie esterne, e molte altre.

È stato quindi scelto di utilizzare questo ambiente di sviluppo per la sua facilità di utilizzo e per la presenza di svariate features che facilitano il lavoro del programmatore.

Java

Il linguaggio di programmazione utilizzato per la logica dell'applicazione Android, è **Java**, supportato da tutte le librerie necessarie per poter creare applicazioni su smartphone Android.

Per effettuare le chiamate asincrone alle API di Google o alla stessa API di Geo Photo Routing, è stata estesa la classe **AsyncTask**, utilizzando i relativi metodi ed una *interface* per restituire il risultato, tramite una callback, non appena esso viene ottenuto.

[1] `www.jetbrains.com/idea/`

Si è scelto di utilizzare questa tecnica di comunicazione asincrona per permettere all'utente di navigare all'interno dell'applicazione senza interruzioni. Se non fosse stata utilizzata, l'esperienza utente sarebbe risultata negativa, essendo egli costretto ad aspettare la risposta dal server senza la possibilità di utilizzare in alcun modo l'applicazione.

XML

La parte presentazionale dell'applicazione, è stata sviluppata con il linguaggio di Markup **XML (eXtensible Markup Language)**. Un linguaggio utilizzato in vari ambiti informatici, solitamente come linguaggio utilizzato dai servizi web per restituire informazioni ai richiedenti. La principale peculiarità del linguaggio è che, come suggerisce il nome, è un linguaggio estensibile (eXtensible), in quanto permette di creare tag personalizzati.

In Android, il suddetto linguaggio è importante perchè permette di creare e visualizzare Layout personalizzati, Widget (Button, Label, EditText), animazioni, forme geometriche, colori, stili, risorse e qualunque altra caratteristica presentazionale (e non solo) di un'applicazione che può essere ritenuta statica, non variabile con l'esecuzione dell'applicazione.

In ogni applicazione, esiste un file particolare, chiamato *AndroidManifest.xml* [16], che fornisce le informazioni essenziali riguardo l'applicazione al sistema operativo Android dello smartphone. Se non è presente o non è corretto, il codice dell'applicazione non viene eseguito dal sistema operativo. Esso contiene informazioni riguardo:

- Il nome del package dell'applicazione;
- i componenti dell'applicazione, tra cui activity, service, broadcast receiver e content provider;
- i permessi che l'applicazione deve avere per poter accedere a parti protette del sistema operativo o per interagire con altre applicazioni;
- i permessi che regolano come una seconda applicazione può utilizzare le componenti della prima;
- la versione SDK (Software Development Kit) minima da utilizzare.

API e librerie Google

Le principiali funzioni del progetto di tesi si basano sull'utilizzo di mappe stradali. Di conseguenza si è scelto di utilizzare tra le tante librerie di Android adatte a tale scopo, quella più comune e più documentata: la libreria di Google per l'utilizzo delle mappe, **Google Play Services**, corredata con un'altra libreria di utility per le mappe, **Android Maps Utils**.
Tramite queste librerie è possibile visualizzare mappe stradali, spostare la visuale tra determinate zone, aggiungere e personalizzare i **Markers**, aggiungere vari Layer alla mappa, tra cui il **Layer Heatmap** ed aggiungere molte altre funzionalità alla mappa.

Nel presente progetto di tesi, le funzionalità delle suddette librerie, che sono state utilizzate, riguardano soprattutto l'aggiunta di Marker per segnare il punto di origine e di fine percorso e segnare i punti intermedi di interesse; la possibilità di disegnare il percorso tramite una *polyline* e la visualizzazione del layer Heatmap. Quest'ultimo, come detto in precedenza, è uno strato da applicare alla mappa di Google, che rappresenta visualmente l'intensità geografica dei dati nella mappa di riferimento. In questo caso i dati sono rappresentati dai geotag delle immagini presenti sul Database.

Oltre alle librerie Java di Google, sono state utilizzate anche due API di Google:

- **Google Place Autocomplete:** [17] Un servizio per Android utilizzato durante l'inserimento da parte dell'utente di un indirizzo stradale. Place Autocomplete fornisce dinamicamente previsioni di completamento dell'indirizzo, in risposta alla query dall'utente ricercata. Mentre l'utente digita, il servizio ritorna suggerimenti sui possibili indirizzi o punti di interesse ricercati.
  Il servizio può essere aggiunto all'applicazione in diverse modalità, come l'aggiunta di un widget Autocomplete o la richiesta dei suggerimenti a livello di programmazione.
  L'utilizzo di Place Autocomplete nel progetto di tesi si limita al richiamo, tramite una *intent* di un'activity Autocomplete, durante l'inserimento dell'indirizzo di partenza e di arrivo del percorso scelto dall'utente, in modo che egli riesca a trovare più velocemente l'indirizzo che sta cercando.

- **Google Directions:** Un servizio, fruibile non soltanto da dispositivi Android, utile per ricevere indicazioni sul percorso ottimale tra due posizioni, secondo Google Maps.
  Il servizio richiede semplicemente di effettuare una **HTTP Request** con metodo GET all'indirizzo `https://maps.googleapis.com/maps/api/directions/outputFormat?parameters`, dove *outputFormat* è il formato dell'output deside-

rato, e può essere scelto tra i valori "*json*" o "*xml*"; e *parameters* indica tutti i parametri richiesti dalla API per poter restituire un risultato. I parametri obbligatori sono: *origin*, *destination* e *key*, che rappresentano rispettivamente il luogo di partenza e di arrivo, sotto forma di coordinate geografiche e la chiave personale di ogni sviluppatore, utilizzata per controllare il numero di utilizzi giornalieri. Esistono ovviamente svariati parametri opzionali per richiedere varie funzioni, come l'inserimento di punti intermedi da cui transitare (*waypoints*), le strade da evitare, la modalità di spostamento e molte altre.
La risposta viene fruita nel formato desiderato e rappresenta il percorso da effettuare, considerando tutte le tappe da percorrere, ed indicando tramite opportuna codifica, la *polyline*[2] generale del percorso.

Nella presente applicazione, questo servizio viene richiamato ogni qualvolta l'utente ricerca un nuovo percorso o ne carica uno salvato in precedenza. La chiamata viene effettuata asincronamente tramite *AsyncTask* e viene effettuata sempre in coppia con una seconda chiamata allo stesso servizio.
La prima si riferisce alla richiesta del percorso standard, quello restituito usualmente da Google Maps; la seconda si riferisce, invece, al percorso secondario, quel percorso che passa dai punti di interesse aumentando il grado di felicità. Dopodichè, le polyline relative ai due percorsi vengono disegnate sulla mappa rispettivamente in blu ed in rosso.
Tramite gli opportuni parametri, durante la chiamata a Google Directions API, viene specificato sempre se ci si vuole spostare a piedi o in macchina e, nel caso della chiamata per l'itinerario secondario, anche da quali *waypoints* transitare.

2.4.2 Server

Per lo sviluppo del progetto di tesi è stato utilizzato un server esterno su cui è presente una piattaforma **LAMP**. Quest'ultima è una raccolta di software liberamente distribuiti per il funzionamento di un server web per siti dinamici ed è l'acronimo di:

- **L**inux, il sistema operativo per il quale è stato progettato;
- **A**pache, il server web;
- **M**ySQL, il database management system;

[2]La polyline è una stringa che rappresenta l'intero percorso. Più precisamente rappresenta le coordinate geografiche, opportunamente codificate, di tutti i punti necessari alla ricostruzione del percorso sulla mappa.

- **P**HP, **P**erl o **P**ython che sono linguaggi di sviluppo per la programmazione web.

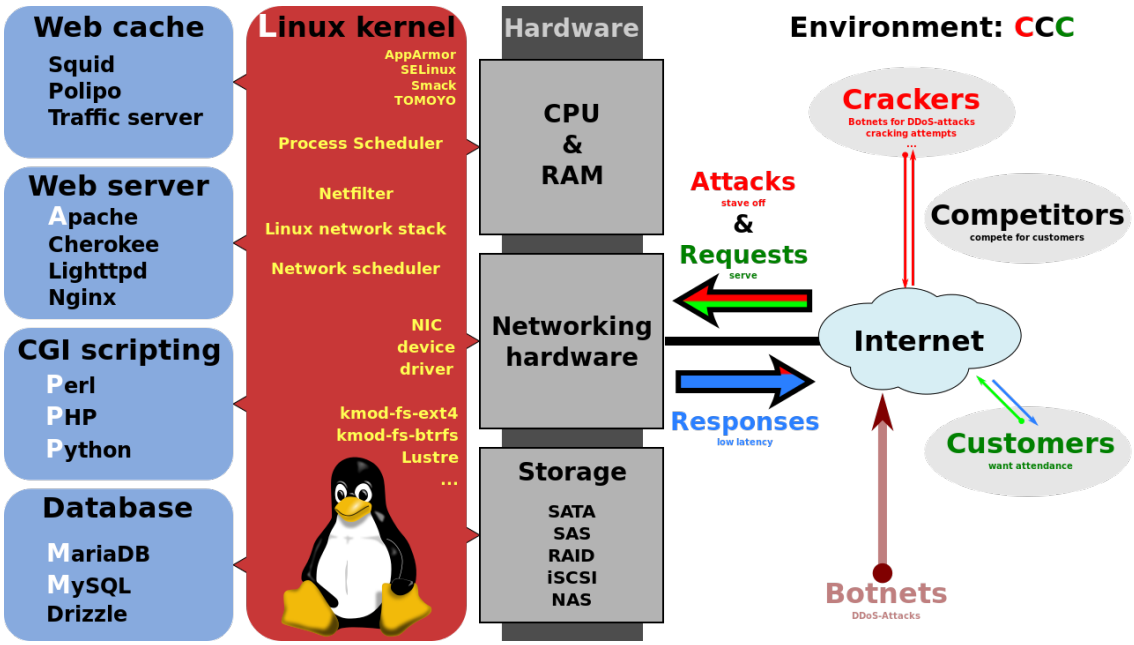

Figura 2.8: Schema della piattaforma LAMP

Si è scelto di sfruttare le potenzialità di questa architettura per la sua accessibilità, facilità di utilizzo e per la presenza di un'ampia documentazione.

PHP

PHP (acronimo ricorsivo per PHP: Hypertext Preprocessor) è un linguaggio di scripting general-purpose open source, creato nel 1994 [18]. Esso ha come obiettivo principale lo scripting server-side per lo sviluppo web, ma può essere usato anche per scrivere script a riga di comando o applicazioni desktop stand-alone con interfaccia grafica.
Con riferimento al suo principale utilizzo, PHP può fare tutto ciò che fa un qualunque programma **CGI (Common Gateway Interface)**, come raccogliere dati da un form, generare pagine dai contenuti dinamici, oppure mandare e ricevere cookies. In fase di esecuzione, PHP interpreta le informazioni ricevute dal client grazie al web server, le elabora e restituisce un risultato HTML al client che ha formulato la richiesta.

Nel progetto di tesi, PHP è stato utilizzato per ricevere le richieste dall'applicazione Android e fornire le risposte corrette, tramite opportuni calcoli, riguardo la ricerca del percorso secondario o le coordinate delle foto, usate per costruire la Heatmap.
Per rendere ciò possibile, si è dovuta creare una connessione al Database, utilizzando un *abstraction layer*: **PDO**. Questo è una classe PHP che rappresenta un'interfaccia, un livello di astrazione per l'accesso ai dati: indipendentemente dal database in uso puoi usare le stesse funzioni per eseguire query e ottenere dati.
Un'altra funzione che ha PHP nel progetto di tesi, senza interfacciarsi con il Client, è l'utilizzo di uno script per caricare le immagini sul Database, ottenendole tramite l'API Flickr.

PHP si pone quindi in mezzo alla comunicazione tra Client e dati presenti sul Database, indirizzando correttamente le richieste del Client, eseguendo opportuni calcoli e procurandosi i giusti dati dal Database.

API Flickr

Flickr è una delle più importanti piattaforme per la gestione e condivisione di foto. Su Flickr, gli utenti possono caricare foto, condividerle in modo sicuro, integrarle con metadati, come informazioni sulla licenza, posizione geografica, persone, tag e così via.
Dal 2005, gli sviluppatori collaborano anche sulle API di Flickr [19], cioè la modalità di accesso a tutti i dati riguardanti le foto di cui Flickr dispone. I dati ottenibili tramite l'API sono quasi tutti quelli di cui Flickr è in possesso, tra cui informazioni riguardo albumo, foto, persone, luoghi, tag, url e molti altri.
Le informazioni che si sono rivelate necessarie per realizzare il progetto di tesi riguardano soprattutto le immagini geotaggate, le relative coordinate, i tag ad esse associati e gli url di queste immagini.

Come molte altri API presenti sul web, l'API Flickr consiste in un insieme di metodi chiamabili e alcuni endpoint API. Per effettuare un'operazione con API Flickr, è necessario inviare una richiesta al suo endpoint, con un formato di richiesta che può essere *REST*, *XML-RPC*, o *SOAP*, specificando un metodo e alcuni argomenti e verrà restituita una risposta formattata con il formato desiderato, per esempio *JSON* o *XML*. I parametri obbligatori sono "method", usato per specificare il metodo chiamato, e "api_key", usato per specificare la chiave API in possesso di ogni sviluppatore che vuole utilizzare il servizio.

Nel presente progetto di tesi, l'API Flickr viene richiamata quotidianamente da uno script PHP, per aggiornare il Database, inserendo in esso i dati riguardanti le ultime foto

caricate su Flickr.
Si è scelto di utilizzare l'API Flickr per la sua facilità di utilizzo e per la presenza di geotag come informazione associata alle immagini. È vero, però, che esiste una varietà di differenti piattaforme di gestione foto che mettono a disposizione API altrettanto valide, come per esempio **Instagram** o **Tumblr**. La scelta sarebbe potuta ricadere anche su alcune di queste ed è infatti una possibile futura implementazione del presente progetto di tesi, quella di ampliare il Database, recuperando immagini provenienti da altre piattaforme di condivisione foto.

MGRS

Le coordinate geografiche delle foto scaricate da Flickr sono state clusterizzate (raggruppate) in piccoli quadrati di 10 metri, in cui è stato suddiviso tutto il territorio italiano, come una griglia, in modo da poter riconoscere più facilmente quali luoghi sono ritenuti "interessanti".
Questa suddivisione, però, non è stata fatta *ad hoc* per il progetto di tesi, ma è stato utilizzato un sistema di griglia di riferimento militare, chiamato **MGRS (Military Grid Reference System)**, che nasce da uno standard molto simile, relativo agli Stati Uniti, **USNG (United States National Grid)** e che si basa sul più noto sistema di coordinate **UTM (Universal Transverse Mercator)**.
MGRS si basa sull'utilizzo di un sistema di griglie, sempre più piccole e dettagliate, che ricoprono virtualmente l'intero pianeta. L'unica vera differenza tra MGRS è USNG è che USNG utilizza una spaziatura per suddividere le tre parti del proprio "indirizzo".
Questi due sistemi di coordinate, si sovrappongono alla griglia proposta da **UTM** [20]. Quest'ultimo si basa su di una proiezione cilindrica del *Mercatore Trasversale* sul pianeta, suddividendolo in 60 grandi zone verticali della larghezza di 6° di longitudine ognuna. Queste zone sono denominato con i numeri da 1 a 60.

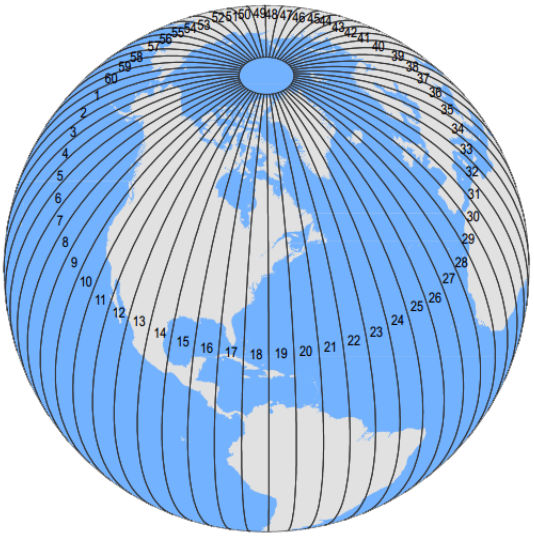

Figura 2.9: Proiezione del Mercatore Trasversale nel sistema di coordinate UTM

Ognuna di queste zone è poi suddivisa in bande orizzontali di 8° di latitudine ognuna, per un totale di 20 bande, denominate con le lettere da C a X, rispettivamente da sud a nord. Il risultato è una griglia che suddivide interamente il pianeta.

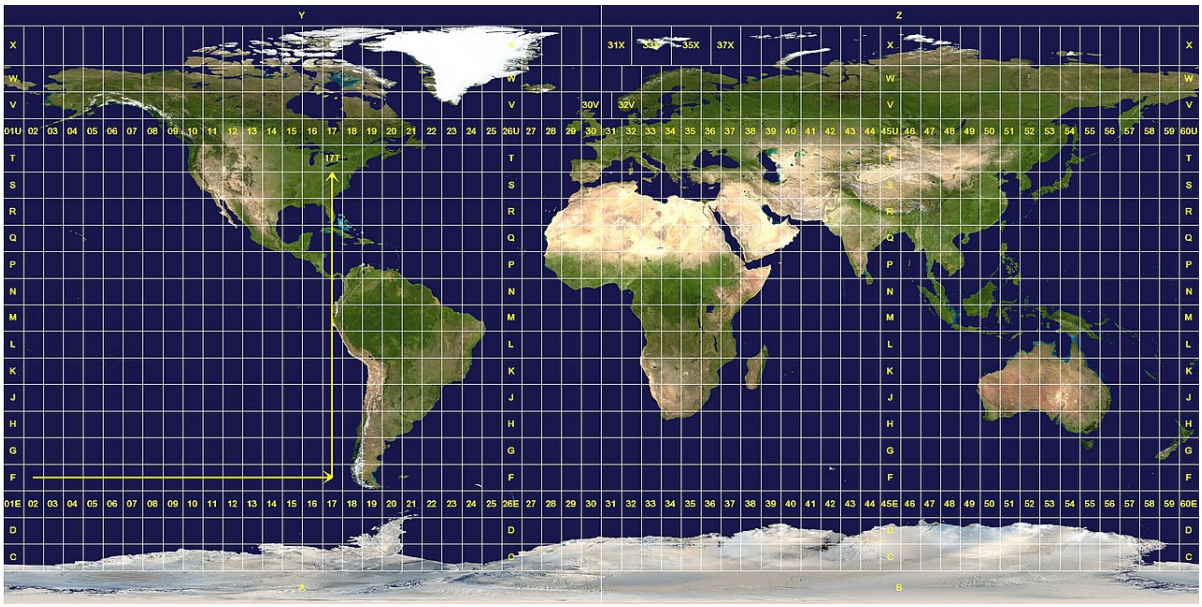

Figura 2.10: Zone della griglia UTM proiettata sulla mappa del mondo

La griglia così definita, diventa la griglia di partenza per il sistema di coordinate utilizzato, **MGRS**. L'indirizzo di questo, è alfa-numerico e si suddivide in tre parti [21]:

- Un codice che designa la zona della griglia UTM, definita a livello mondiale (Es.: 32T).
- Un codice che definisce un quadrato specifico all'interno di un ulteriore griglia interna, identificata a livello di aree regionali, non univoca a livello mondiale. Ogni quadrato identifica uno spazio di 100 000 metri quadrati. (Es.: PQ)
- Le coordinate interne all'ultimo quadrato definito. Queste coordinate sono fornite col sistema est-nord, come ogni altro sistema di coordinate. La particolarità, però è che queste coordinate finali possono essere date con diversi livelli di precisione, di granularità.
  Se non viene fornita nessuna cifra, si considera un livello di precisione 100km. (Es.: 32T PQ)
  Se vengono fornite solo 4 cifre (2 est e 2 nord), il livello di precisione scende a 1000 metri. (Es.: 32T PQ 87 29)
  Se vengono fornite 6 cifre (3 est e 3 nord), il livello di precisione scende a 100 metri. (Es.: 32T PQ 873 297)
  Se vengono fornite 8 cifre (4 est e 4 nord), il livello di precisione diventa di 10 metri. (Es.: 32T PQ 8732 2978)
  Se infine vengono fornite 10 cifre (5 est e 5 nord), il livello di precisione è di 1 metro, la precisione massima raggiungibile da MGRS. (Es.: 32T PQ 87321 29783)

L'indirizzo così creato: 32T PQ 87321 29793 indica il Dipartimento di Informatica in Mura Anteo Zamboni, a Bologna.

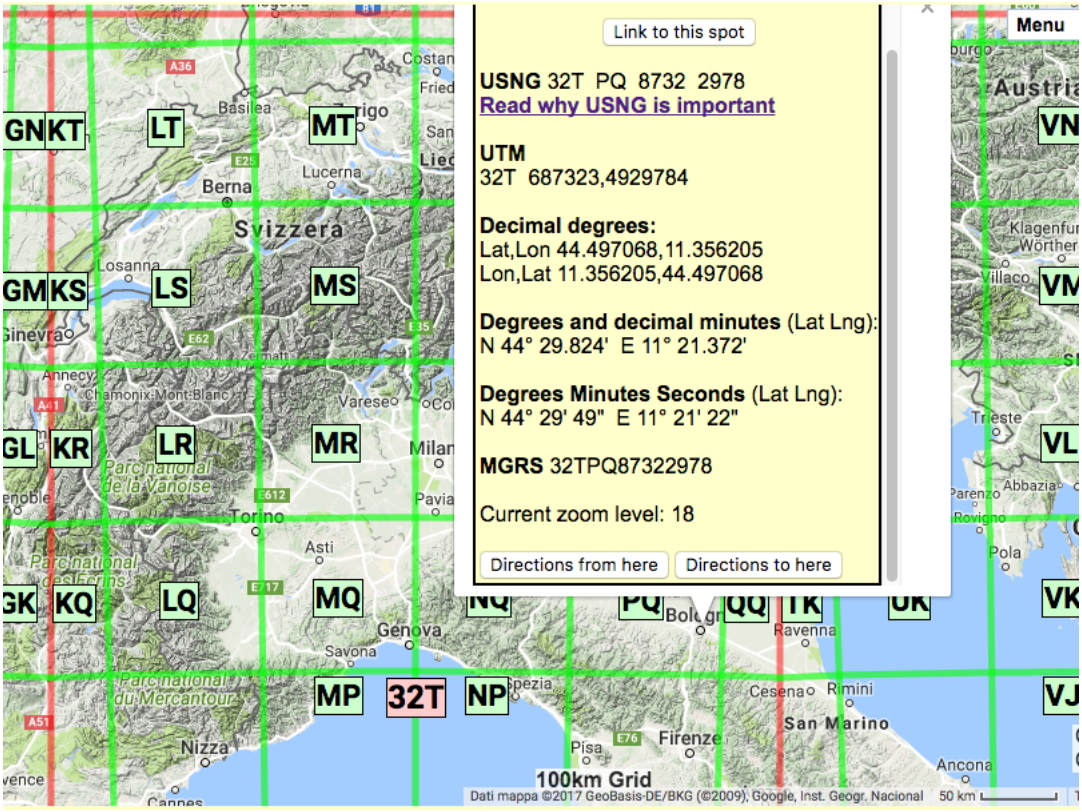

Figura 2.11: Esempio di utilizzo delle coordinate MGRS [22].

Nel presente progetto di tesi, si è deciso di fermarsi a 10 metri di precisione, raggruppando tutte le foto le cui coordinate si trovano all'interno di un determinato quadrato. In questo modo, se foto scattate allo stesso monumento hanno una posizione leggermente diversa, verranno considerate come rappresentanti lo stesso soggetto.
Per l'utilizzo pratico di questo sistema di coordinate, è stata trovata online, sul repository **GitHub** una classe PHP che converte le coordinate nel sistema *longitudine-latitudine*, in coordinate *MGRS* e viceversa [23]. Si ringrazia Julian Aceves per lo script.

2.4.3 Comunicazione Client-Server

Nelle architetture Client-Server esistono diversi modi per mettere in comunicazione i due agenti. Solitamente le comunicazioni avvengono tramite *pull / get*, cioè solitamente è il Client a richiedere le informazioni al Server quando ne ha bisogno. Ma è possibile avere

anche una comunicazione di tipo *push*, in cui è il Server ad iniziare la comunicazione, inviando dati al Client, senza che egli li abbia specificatamente richiesti (es.: servizi di posta elettronica). A volte la comunicazione *push*, può essere emulata tramite *polling*: il Client esegue diverse richieste *pull* ad intervalli regolari, controllando se sul server ci sono nuove informazioni.

Siccome nel progetto di tesi è sempre e solo il Client a richiedere informazioni di cui il Server è in possesso, la comunicazione avviene tramite *GET*, più specificatamente tramite l'utilizzo di un'**API REST** che il Client deve richiamare e che restituisce una risposta in formato **JSON**.

API REST

REST è l'acronimo di **REpresentional State Transfer** [24], ed è il modello architetturale che sta dietro alla progettazione di molte applicazioni web. Un'applicazione REST si basa fondamentalmente sull'uso dei protocolli di trasporto (**HTTP**) e sugli identificatori univoci di risorse (**URI**) per generare interfacce generiche di interazione con l'applicazione. Nel modello REST, ogni operazione è espressa attraverso i metodi di HTTP, quali **GET, POST, PUT, DELETE** e vengono previsti parametri per rappresentare al meglio lo stato della risorsa richiesta.
Un' **API (Application program Interface) REST**, definisce quindi le caratteristiche che deve avere l'interfaccia dell'applicazione sul server, per essere utilizzata dai Client.

Come ogni API REST, anche quella creata per il progetto di tesi fornisce un end-point: *function.php* a cui si possono passare parametri. Il parametro principale è *action* per indicare quale funzione l'applicazione PHP deve svolgere. I valori del parametro a cui risponde una funzione in risposta sono: "*async_returnGeoPhotoPointsCoord*", "*async_returnCordForTag*" e "*async_returnCord*"; rispettivamente per ottenere: le coordinate dei punti di interesse, le coordinate delle foto corrispondenti ad un determinato tag e le coordinate di tutte le foto sul database.

L'API REST di Geo Photo Routing permette quindi alle varie applicazioni Client di comunicare col server, richiamando, tramite l'opportuno parametro, uno dei metodi messi a disposizione dal'end-point, in modo da poter accedere ai servizi che il server mette a disposizione, come la ricerca del percorso migliore tra due punti o la restituzione delle coordinate per visualizzare una Heatmap.

JSON

JSON (Javascript Object notation) [25], è un tipo di formato tipicamente utilizzato per lo scambio dati in applicazioni client server. Permette la descrizione e soprattutto lo scambio di dati ed è comodo, ordinato, facilmente leggibile.
JSON è una valida alternativa al formato **XML-XSLT** e sempre più servizi di Web Services mettono a disposizione entrambe le possibilità di integrazione. Leggere ed interpretare uno stream in Json è semplice in tutti i linguaggi, soprattutto in PHP ed altri linguaggi server-side.

Nel progetto di tesi viene utilizzato come formato dei dati che le funzioni PHP restituiscono ai Client. In particolare, vengono usati soprattutto gli array JSON, per contenere i gruppi di coordinate da restituire alle richieste Client.
È stato considerato più funzionale utilizzare JSON rispetto ad XML o altri formati di interscambio dati, perchè ha una struttura molto leggera, permettendo quindi un maggiore velocità di trasmissione. Inoltre, sia **Java**(Client-side) che **PHP**(Server-side) dispongono di opportune classi per facilitare la lettura di strutture JSON.

2.4.4 Database

Nell'architettura a tre livelli, il Database rappresenta il livello in cui sono contenuti i dati sensibili dell'applicazioni. Essi sono separati dagli altri livelli per mantenere consistenza, ordine, autonomia.
Le funzioni del secondo livello non accedono direttamente al Database, ma dialogano con un software che lo gestisce, il **DBMS**, per esempio **MySQL**.

MySQL

Come suggerisce la metodologia **LAMP**, per la memorizzazione dei dati si è scelto di utilizzare **MySQL**. Esso è un **RDBMS (Relational Database Management System)** open-source e multi-piattaforma ed è il più diffuso a livello mondiale grazie alle sue performance collaudate, alla sua affidabilità, flessibilità e facilità d'uso.
MySQL è un *Relational* Database Management System (RDBMS), e come suggerisce il nome, è basato sul modello *relazionale*. Esso, come ogni altri DBMS, si occupa quindi di creare, la manipolare e interrogare un Database, ovvero una collezione di dati strut-

turati; si occupa di tutte le politiche di accesso, gestione, sicurezza ed ottimizzazione del Database ed è l'unico ad accedere fisicamente alle informazioni contenute in esso.

Struttura Database

Si passa ora a spiegare nel dettaglio come è organizzato il database utilizzato nel progetto di tesi. Le tabelle utilizzate sono 3: "*photos*", "*tags*" e "*mgrs_squares*".

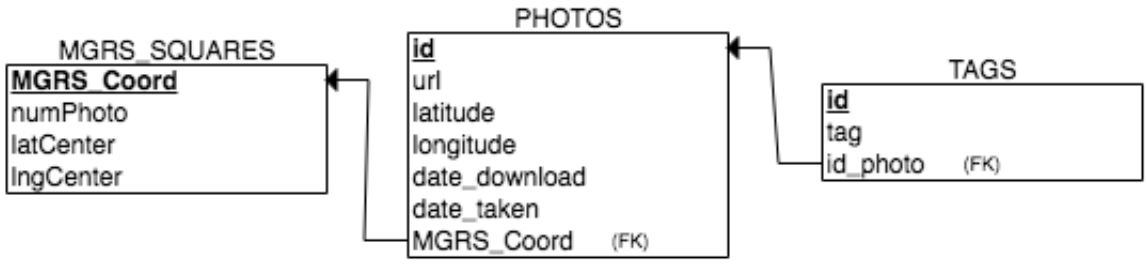

Figura 2.12: Schema relazionale del Database. Disegnata con `https://erdplus.com/`

La tabella "*photos*" contiene i dati delle fotografie scaricate tramite l'ausilio dell'API Flickr. Nello specifico gli attributi della tabella sono:

- **id** : costituisce la chiave primaria della tabella. Rappresenta l'id univoco della fotografia.
- **url**: rappresenta l'URL (Uniform Resource Locator) della fotografia.
- **latitude**: latitudine geografica della posizione in cui la foto è stata scattata.
- **longitude**: longitudine geografica della posizione in cui la foto è stata scattata.
- **date_download**: data (nel formato aaaa-mm-gg) in cui la fotografia è stata salvata sul database.
- **date_taken**: data (nel formato aaaa-mm-gg) in cui la fotografia è stata scattata.

- **MGRS_Coord**: il cluster relativo al sistema di coordinate MGRS a cui la fotografia appartiene.

La tabella "*tags*" rappresenta i tags di ogni singola fotografia caricata nel database:

- **id**: è la primary key della tabella. Rappresenta l'identificativo di ogni singolo tag ed è generato in modo automatico da MySQL, incrementando sempre di uno il suo valore ogni volta che viene aggiunto un nuovo record.
- **tag**: è il contenuto del tag, inteso come testo.
- **id_photo**: rappresenta il valore dell'id della fotografia a cui appartiene il tag in questione. Chiave esterna della tabella "*tags*".

Infine la tabella "*mgrs_squares*" rappresenta i cluster del sistema di coordinate MGRS con alcune loro informazioni.

- **MGRS_Coord**: è la primary key della tabella. Rappresenta l'indirizzo geografico scritto nel sistema di coordinate MGRS, con precisione di 10 metri.
- **numPhoto**: indica il numero di foto presenti nella tabella *photos* che sono contenute in questo cluster. È un dato duplicato, ma è di grande utilità, in quanto permette di risparmiare un elevato numero di computazioni ogni volta che l'utente ricerca un nuovo percorso.
- **latCenter**: latitudine geografica del centro del quadrato. Questo dato è utilizzato insieme alla longitudine per indicare un punto che rappresenti tutto il quadrato.
- **lngCenter**: longitudine geografica della posizione in cui la foto è stata scattata.

2.5 Algoritmo di ricerca

Nella versione iniziale di Geo Photo Routing, il cammino da percorrere consigliato dall'applicazione, quello che avrebbe dovuto portare ad un maggior benessere, veniva calcolato in un modo molto superficiale, senza un vero e proprio algoritmo che lo valorizzasse.

Il lavoro svolto in questo progetto di tesi è volto anche alla ricerca di un migliore algoritmo, che possa risolvere i problemi del precedente e possa valorizzare il percorso secondario consigliato all'utente, in modo che rispecchi più fedelmente i valori richiesti di *benessere* e *felicità*, senza stravolgerlo completamente.

Ma partiamo con ordine, descrivendo ora l'algoritmo iniziale.
Per calcolare il percorso secondario, l'algoritmo ricercava una serie di punti di interesse da cui transitare, per poi inviare una richiesta all'**API Directions** di Google Maps fornendogli i punti di interesse trovati, grazie al parametro *waypoints*. Dopodichè Google Maps calcolava il percorso che passava da tutti quei punti di interesse, nel modo che riteneva più opportuno (ottimizzando i tempi di spostamento, scegliendo la strada più breve). Si otteneva in questo modo il percorso secondario finale.
Entrando più nel dettaglio, i passi per la selezione dei punti di interesse da cui dover transitare erano:

1. Si calcola l'itinerario standard tra i punti di partenza e di arrivo indicati, utilizzando l'**API Directions** di Google Maps
2. Si ricava quali sono i punti intermedi che definiscono il percorso, analizzando la *polyline* resituita dal servizio appena citato.
3. Si considera ognuno dei punti intermedi come un ipotetico centro di un cerchio con raggio, misurato in metri, definito dall'utente.
4. Si prende in considerazione tutte le immagini che cadono all'interno dei cerchi che si sono così venuti a creare.
5. Per ogni immagine, si controlla se nella stessa esatta posizione (con le stesse coordinate), esistono almeno 10 immagini
6. Se in tale posizione esistono effettivamente almeno 10 immagini, allora il punto è da considerare rilevante, perchè si suppone che se un punto molto fotografato, allora è di conseguenza molto interessante e piacevole.
7. Il punto così trovato viene giudicato d'interesse e viene aggiunto come tappa del percorso.

Ad una prima lettura forse l'algoritmo non sembra essere poi così male, ma nasconde qualche problema che può assumere molta rilevanza, per esempio:

- Se lo stesso soggetto è stato fotografato più di 10 volte, ma da locazioni geografiche leggermente diverse, non viene riconosciuto come luogo d'interesse, perchè le

coordinate non sono *esattamente* uguali.
Un esempio reale: le due torri di Bologna sono state ovviamente fotografate più di 10 volte, ma da luoghi, seppur vicini, non esattamente identici. Questo ha portato, nella vecchia implementazione, a *non* considerare le torri degli Asinelli e della Garisenda come luogo d'interesse.
Questo avviene ovviamente in molte altre situazioni, portando l'algoritmo a considerare importanti pochissimi luoghi che potrebbero effettivamente aumentare il benessere di un percorso.

- Se un punto di interesse si trova all'interno del raggio di ricerca, viene *sempre* selezionato, indipendentemente dal percorso globale ed indipendentemente dagli altri punti di interesse. La selezione avviene quindi senza tenere in considerazione il percorso nel suo insieme. Ciò porta alla creazione di percorsi molto improbabili, difficili da percorrere, o che aumentino di molto la strada da percorrere rispetto al percorso originario. Per esempio, può accadere che la lunghezza del percorso raddoppi, consigliando in alcuni casi, di incamminarsi dalla parte opposta rispetto alla destinazione.
  Questa situazione non è da considerarsi un errore implementativo, ma si scontra con quelle che sono le specifiche iniziali, cioè di trovare un percorso non solo più piacevole da percorrere, ma anche il più *breve* possibile.

Di seguito due immagini per mostrare un esempio di quanto appena spiegato:

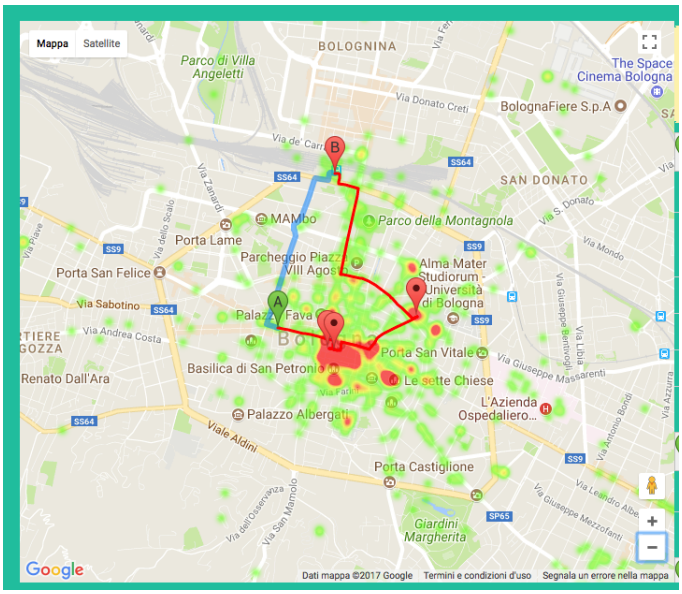

Figura 2.13: Esempio di percorso non ottimale.

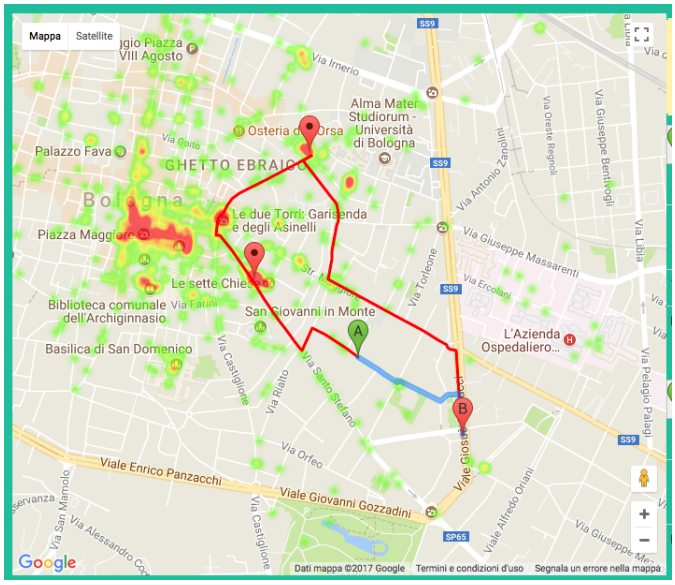

Figura 2.14: Esempio di percorso non ottimale.

Tramite le due immagini sopra riportate possiamo notare come il percorso tende ad allungarsi, passando da punti di interesse che potrebbero essere evitati, in quanto aggiungono una rilevante quantità di strada da percorrere rispetto al percorso originale.

Per risolvere la prima di queste problematiche, cioè l'incapacità di considerare interessanti luoghi che effettivamente lo sono, si è deciso di *clusterizzare* il territorio, quindi suddividerlo in piccoli quadrati di 10 metri ed inglobare tutte le immagini che ricadono all'interno di ogni singolo quadrato, nel quadrato stesso. In questo modo, tutte le foto che sono state scattate ad una distanza minore di 10 metri l'una dall'altra, vengono considerate come raffiguranti lo stesso soggetto.
Come spiegato in precedenza, per la clusterizzazione si è utilizzato il sistema di coordinate **MGRS**.

Per risolvere la seconda di queste problematiche, cioè la restituzione di percorsi non ottimizzati, sono state pensate diverse soluzioni.
Inizialmente si è pensato di considerare tutta l'area geografica compresa tra il punto di partenza e il punto di arrivo e di considerare i quadrati di suddivisione proposti da MGRS.
Assegnare, poi, ad ogni quadrato un valore, in base a vari criteri, come la quantità di foto presenti all'interno del quadrato. Infine utilizzare tecniche di **Programmazione Dinamica** per trovare il percorso con valore maggiore che porta dal punto di partenza al

punto di arrivo, scegliendo passo passo quale sarà il prossimo quadratino da percorrere. Questa soluzione permette di trovare percorsi di gran lunga migliori rispetto all'algoritmo originale, ma purtroppo difetta nella complessità: è facile arrivare ad un numero di computazioni elevatissimo, se i punti di partenza e di arrivo sono distanti tra loro.

Si è quindi deciso di utilizzare una tecnica diversa: l'algoritmo di **Dijkstra** applicato ad un grafo *pesato* ed *orientato*. Questa soluzione implementativa verrà spiegata nel dettaglio nel prossimo capitolo.

Capitolo 3

Implementazione

Questo capitolo ha lo scopo di fornire una descrizione pratica della fase realizzativa del progetto di tesi descrivendo le più importanti scelte implementative compiute durante lo sviluppo e talvolta illustrandole con pezzi di codice sorgente. La descrizione è suddivisa nei vari livelli che costituiscono l'architettura adottata, introdotti nel paragrafo 2.3.

3.1 Client-side

Si procede, innanzitutto, a descrivere tutte le funzionalità e le scelte implementative presenti nell'applicazione Android.

3.1.1 Google Maps

Tutte le funzioni dell'applicazione si basano sull'utilizzo di mappe. Si è scelto quindi di utilizzare il servizio più famoso e facilmente integrabile con i dispositivi android: **Google Maps**.
Per poter iniziare ad utilizzare le mappe di Google durante la programmazione Android, è stato necessario, dopo aver ottenuto la **google maps key**, creare un opportuno *fragment* con la proprietà `android:name="com.google.android.gms.maps.SupportMapFragment"`.
All'interno dell'activity Java, sono state incluse le opportune librerie, implementata la

classe **OnMapReadyCallback** e creato un oggetto di tipo **GoogleMap**, che è stato chiamato *mMap*. Quest'ultimo sarà l'oggetto principale che utilizzeremo per ogni funzionalità dell'applicazione, richiamando i suoi metodi e attributi.
La callback principale fornita da questa classe è *onMapReady*. Essa permette di eseguire codice non appena la mappa è pronta all'utilizzo.

3.1.2 Chiamate alle API

Le principali funzioni dell'applicazione richiedono di richiamare l'**API REST** di **Google Directions**, per ottenere i percorsi da percorrere, o l'API REST che mette a disposizione il Server di **Geo Photo Routing**, per ottenere coordinate geografiche riguardo le foto che compongono le heatmap o riguardo i punti di interesse del percorso secondario.
Si è deciso di unificare tutte queste richieste esterne all'interno di un unica classe, chiamata **APICaller**. Essa estende la classe **AsyncTask**, che permette di eseguire codice in modo asincrono rispetto al thread principale. Questo passaggio è fondamentale per fornite all'utente un'esperienza fluida, senza dover aspettare l'esecuzione di codice che può avvenire in background.
Questa classe ha due costruttori: il primo permette di attribuire i parametri giusti all'URL dell'endpoint dell'API REST di Geo Photo Routing. In pratica con questo costruttore si indica quale funzione dovrà essere invocata dall'applicazione sul server di Geo Photo Routing, passando gli opportuni parametri.
Il secondo costruttore permette di fornire alla classe un URL già preimpostato, da richiamare, che potrà avere un endpoint diverso da quello dell' API di GeoPhotoRouting.

Una volta che viene chiamato il metodo *execute* della classe *AsyncTask*, la classe in questione si occupa di accedere all'URL fornito nel costruttore, attendere e memorizzare la risposta del server.
Siccome tutto il procedimento è avvenuto asincronamente rispetto al *main thread*, per fornire la risposta al chiamante, è stata creata una **interface**, che contiene un metodo che verrà richiamato ad operazione conclusa. Questo metodo, si chiama *onResponseReady* e restituisce, a chiunque implementi l'interfaccia, la risposta del server in formato stringa.

Riassumendo, questa classe si occupa solo di recuperare la risposta dell'API REST indicata, tramite un URL, dal chiamante e restituirla tramite un'opportuna callback.

3.1.3 Heatmap

Ad ogni accesso all'applicazione, la prima schermata che si presenta all'utente è una Heatmap generale di tutte le foto presenti sul database. Per poterla visualizzare, viene invocato un metodo chiamato *getPhotoCoordForHeatmap*, che si occupa di istanziare la classe di tipo *APICaller* sopra descritta, con un opportuno parametro ("*async_returnCord*") e chiamarne il metodo *execute*.

È stato poi implementato il metodo astratto "*onResponseReady*", che viene richiamato una volta che la risposta da APICaller è pronta.

```
@Override
public void onResponseReady(String response, String action){

    DirectionParser parser;

    switch (action)
    {
        case "async_returnCord":
            showHeatMap(parseJSON(response));
            break;
        case "async_returnCordForTag":
            showHeatMap(parseJSON(response));

            pbKeywordLoading.setVisibility(View.GONE);
            btnKeywordDone.setVisibility(View.VISIBLE);
            break;
        case "async_returnGeoPhotoPointsCoord":

            GPRPoints = new ArrayList<>();
            GPRPoints = parseJSON(response);
            getGeoPhotoRoutingDirections();

            break;
        case "google_maps_standard_directions":

            parser = new DirectionParser(this,response,"standardDirections");
            parser.execute();
            break;
        case "google_maps_GPR_directions":

            parser = new DirectionParser(this,response,"GPRDirections");
            parser.execute();
            break;
        default:
            break;
    }

}
```

Figura 3.1: Metodo di callback di APICaller implementato nell'activity principale.

In base alla variabile *action*, che rappresenta quale funzione ha richiesto i dati, vengono eseguiti metodi differenti.
Si osservino per adesso i primi due casi, che sono relativi alla creazione di una Heatmap. In essi viene richiamato un metodo per eseguire il parsing della struttura JSON, in modo da ottenere una lista di *LatLng* contenente le coordinate delle immagini. Questa lista viene poi passata ad una funzione che si occupa di visualizzare effettivamente la Heatmap, grazie ad una classe di Google chiamata *HeatmapTileProvider*.

```
private List<LatLng> parseJSON(String jsonString)  {

    List<LatLng> list = new ArrayList<>();

    try
    {
        JSONArray responseArray = new JSONArray(jsonString);


        for (int i = 0; i < responseArray.length(); i++) {
            JSONArray latlngArr = responseArray.getJSONArray(i);
            Double lat = latlngArr.getDouble(0);
            Double lng = latlngArr.getDouble(1);
            list.add(new LatLng(lat,lng));

        }


    } catch (JSONException e) {
        e.printStackTrace();
    }

    return list;
}
```

Figura 3.2: Metodo che trasforma la stringa JSON in ingresso, in un array di coordinate.

3.1.4 Calcolo del percorso

Si passa ora a descrivere come è stata implementata la funzionalità principale, cioè la ricerca dei percorsi: quello consigliato da Google Maps e quello consigliato da Geo Photo Routing.
Gli step per arrivare alla rappresentazione sulla mappa dei due percorsi, sono 4:

1. Ottenere tramite l'apposito form, come si vede in figura 2.3, i dati relativi al punto di partenza, punto di arrivo, modalità di viaggio e raggio di ricerca.

2. Chiamare l'API REST di Geo Photo Routing, passando come parametri i dati appena recuperati. L'API restituisce le coordinate dei punti di interesse da cui passare.

3. Chiamare una prima volta l'**API Directions** di Google, ottenendo il percorso più breve che passa solo dai punti di partenza e di arrivo.

4. Chiamare una seconda volta l'**API Directions** di Google, passando come parametri i punti intermedi ottenuti al punto 2. In questo modo si ottiene il percorso secondario, quello che aumenta il *benessere* dell'utente.

Per chiamare l'API REST, come indicato al passo 2, si utilizza le stessa modalità spiegata nella sezione precedente: si crea un istanza di *APICaller* passando gli opportuni parametri al costruttore; si chiama la funzione di esecuzione; si fa il parsing del JSON di risposta, dentro alla funzione callback *onResponseReady* (vedi Fig. 3.1); ed una volta ottenuti i punti intermedi, si può invocare la funzione *getGeoPhotoRoutingDirections*, che si occupa di richiamare l'API Directions di Google, specificando i punti intermedi da cui passare.

I passi 3 e 4 sono per lo più identici: le due funzioni *getDirectionsFromGoogleMaps* e *getGeoPhotoRoutingDirections*, si occupano di istanziare la classe *API Caller*, passando come parametro l'endpoint dell'**API Directions** di Google.
La prima delle due funzioni fornisce all'APICaller un URL preconfezionato, che indica a google di restituire il percorso più breve tra due punti. Es.: `https://maps.googleapis.com/maps/api/directions/json?origin=44.4941904,11.3520296&destination=44.4974584,11.33222&key=API_KEY&mode=walking`.
La seconda delle due funzioni, fornisce all'APICaller lo stesso URL preconfezionato, ma con l'aggiunta del parametro *waypoints*, che indica da quali punti intermedi deve passare il percorso restituito.
Il risultato, sottoforma di JSON, viene analizzato come al solito nella callback *onResponseReady* (vedi Fig. 3.1).

In questo caso, però, il JSON risultante è complesso e non si può analizzare tramite la funzione in Fig. 3.2, per cui viene istanziata un'ulteriore classe che estende *AsyncTask*: la classe **DirectionParser**.
Questa classe, che viene eseguita in modo asincrono e restituisce il risultato tramite una callback esattamente come succede nel caso di APICaller, si occupa di prendere in input il JSON restituito dalla API Directions di Google, analizzarne il contenuto, e restituirlo in un formato utilizzabile, tramite una classe appositamente creata, chiamata **GoogleRoute**.
Il JSON risultante dalla **API Directions** di Google è formato da:

- un array chiamato ***routes***, che comprende tutti gli itinerari dal punto di partenza a quello di arrivo trovati (solitamente è solo uno).
- per ogni *routes*, un array chiamato ***legs***, che rappresenta una sezione del percorso, descrivendo alcune proprietà della sezione, come lunghezza, durata in minuti per percorrerla, indirizzo di partenza, indirizzo di arrivo
- per ogni *legs*, un array chiamato ***steps***, che rappresenta tutti i passi che bisogna fare per percorrere quella determinata *leg*, descrivendo per ogni passo il punto di partenza, quello di fine, la distanza, la durata in minuti per percorrerlo, le istruzioni scritte, interpretabili facilmente da chiunque, da seguire per percorrere quel pezzo di itinerario e soprattutto la *polyline* che rappresenta tutti i punti geografici, codificati con una stringa, che formano quel determinato *step*.


```
"routes":[ ⊖
   { ⊖
      "bounds":{ ⊖
         "northeast":{ ⊖
            "lat":44.4974584,
            "lng":11.3520296
         },
         "southwest":{ ⊖
            "lat":44.49419049999999,
            "lng":11.3322225
         }
      },
      "copyrights":"Dati mappa ©2017 Google",
      "legs":[ ⊖
         { ⊖
            "distance":{ ⊖
               "text":"1,7 km",
               "value":1652
            },
            "duration":{ ⊖
               "text":"20 min",
               "value":1209
            },
            "end_address":"Via Pietralata, 8, 40122 Bologna BO, Italia",
            "end_location":{ ⊖
               "lat":44.4974584,
               "lng":11.3322225
            },
            "start_address":"Piazza Aldrovandi, 23, 40125 Bologna BO, Italia",
            "start_location":{ ⊖
               "lat":44.49419049999999,
               "lng":11.3520296
            },
            "steps":[ ⊖
               { ⊖
```


Figura 3.3: Un esempio del JSON restituito dall'API Directions.

La classe **DirectionParser** prende quindi in input il JSON sopra descritto e, per ogni *step*, decodifica la *polyline* associata, tramite un'opportuna funzione, memorizzando tutti i punti geografici necessari a disegnare l'itinerario sulla mappa di Google. La funzione utilizzata per decodificare la *polyline* ed ottenere latitudine e longitudine dei punti geografici, non è stata creata appositamente per questo progetto, ma è stata utilizzata una funziona precedentemente esistente, ottenuta sul sito `http://jeffreysambells.com/2010/05/27/decoding-polylines-from-google-maps-direction-api-with-java`.

Per restituire il risultato, DirectionParser, istanzia una classe di tipo **GoogleRoute**, creata appositamente per contenere il risultato della decodifica di un itinerario. All'oggetto appena creato vengono attribuiti dati come:

- la lista di punti decodificati a partire dalla suddetta polyline.
- la lunghezza totale del percorso
- la durata totale, in minuti, per percorrere l'intero percorso
- la lista di istruzioni che verranno fornite all'utente per orientarsi nella città e seguire il percorso.

In conclusione, l'oggetto **GoogleRoute** appena creato, viene restituito al chiamante tramite una *callback*, denominata **didParsingFinish**, nello stesso modo che avviene per la classe APICaller.

Si torni ora al thread principale e si analizzi la callback **didParsingFinish**. Questa funzione si occupa di :

- Disegnare i due percorsi sulla mappa, basandosi sui punti decodificati dalla *polyline*;
- Visualizzare a schermo la durata e la lunghezza dei due percorsi, per un veloce confronto;
- Preparare la lista di istruzioni da seguire per percorrere l'itinerario, in formato visualizzabile facilmente dall'utente;
- Agguingere appositi *marker* alla mappa, posizionati sui punti di interesse.

3.1.5 My Routes

Una volta ricercato un itinerario, l'utente ha la possibilità si salvarlo, in modo da poterlo facilmente recuperare in un secondo momento.

Per la memorizzazione degli itinerari si è scelto di utilizzare l'interfaccia che Android mette a disposizione, chiamata **SharedPreferences**. Essa permette di memorizzare vari tipi di dati, indicizzati da una stringa scelta, sul dispositivo in cui l'applicazione viene installata. I dati sono associati all'applicazione, quindi una volta che questa viene disinstallata, anche i dati spariscono.
Questa modalità di memorizzazione non è completamente sicura, perchè chiunque con i privilegi *superuser* o privilegi a livello di *root*, può accedere ai dati memorizzati. Ma siccome non vengono salvati dati sensibili, come una password, si è deciso che **SharedPreferences** fosse lo strumento adatto alla memorizzazione.

Quando l'utente preme sul pulsante adibito al salvataggio dell'itinerario, invoca una funzione chiamata *saveRoute()*, nella quale:

- Viene caricata la lista di itinerari già presenti come salvataggio, se ne esiste almeno uno, e convertita in formato *JSONArray*.
- Viene codificato l'itinerario da salvare, in formato JSON.
- L'itinerario appena codificato, viene aggiunto alla lista di itinerari.
- La nuova lista, completa del nuovo itinerario, viene quindi convertita in stringa e salvata sul dispositivo, grazie a *SharedPreferences*.

Per la visualizzazione della lista di itinerari salvati, si è invece scelto di utilizzare una **ListView**. Questa, necessita di un *ArrayAdapter* per essere riempita, inizializzato con il tipo dati corretto.
Siccome i dati che devono essere visualizzati su ogni elemento della lista sono più di uno, si è vista la necessità di creare un'apposita classe, chiamata **MyRoute**, che contenesse tutti i dati dell'itinerario necessari (punto di partenza, di arrivo, punti intermedi, modalità di viaggio, raggio di ricerca e data di salvataggio). Questa classe è diventata poi il tipo dati dell'ArrayAdapter.
Quando l'utente preme sul pulsante *MyRoutes*, per visualizzare la lista di itinerari salvati, viene prima caricata la lista salvata sul dispositivo, sotto forma di stringa JSON; poi, viene eseguito il *parsing* del JSON tramite un'apposita funzione chiamata *getRoutesFromJSON*, ottenendo così una lista di tipo **MyRoute**. Infine questa lista viene data in

pasto all'ArrayAdapter che si occuperà di far visualizzare correttamente ogni elemento sulla lista.

È stata prevista anche la possibilità, da parte dell'utente, di cancellare un itinerario salvato. Egli dopo aver tenuto premuto l'itinerario da cancellare, dovrà rispondere affermativamente ad un messaggio di conferma. In seguito l'itinerario viene rimosso sia dall'Array Adapter, utilizzando il metodo *notifyDataSetChanged* per aggiornarne la visualizzazione; sia dalla struttura JSON memorizzata sul dispositivo, utilizzando un apposita funzione che ricerca e rimuove l'itinerario selezionato, lasciando intatto il resto del file JSON.

3.2 Server-side

In questa sezione si spiegherà nel dettaglio come sono state implementate le funzionalità del server:

- Il server si occupa di gestire la connessione al Database e di inviare query ad esso.
- Il principale servizio fornito all'esterno dal Server è un API REST che permette ai Client di accedere a diverse tipologie di dati, a seconda della funzione richiesta.
- Ogni giorno, il server avvia uno script PHP che ottiene, tramite l'API Flickr, le foto scattate quello stesso giorno e le carica sul Database, in modo da avere un Database costantemente aggiornato.

3.2.1 Connessione al Database

All'interno del file *connection.php* avviene la connessione al database utilizzando l'*abstraction layer* **PDO**. Come avviene con qualsiasi altra libreria per l'interazione tra PHP e MySQL, anche con PDO è richiesta la connessione al DBMS prima di poter svolgere qualsiasi tipo di operazione sui dati.
Quando si parla di "connessione" tra un'applicazione e un DBMS si sta descrivendo una procedura di autenticazione e per portarla a termine occorrono alcuni parametri, necessari per fare in modo che il DBMS riconosca l'applicazione web. Questi parametri sono:

- **il tipo di database**: dato che PDO supporta più tecnologie DBMS, si può specificare quella che si intende utilizzare;
- **host**: per indicare l'indirizzo del sistema di riferimento che ospita il DBMS utilizzato;
- **il nome del database**: ovvero il nome del database sul quale si desidera operare;
- **user**: è il nome di un utente al quale sono stati associati dei privilegi per eseguire istruzioni sul database e in base a questi saranno consentite o meno determinate operazioni;
- **password**: è la chiave utilizzata dall'utente per concludere la procedura di autenticazione.

```php
$servername = "89.46.111.30";
$username = "Sql1025268";
$password = "xxxxxx";

try {
    header('Content-Type: text/html; charset=utf-8');
    $conn = new PDO("mysql:host=$servername;dbname=Sql1025268_3", $username, $password);
    // set the PDO error mode to exception
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $conn->exec("set names utf8");

}
catch(PDOException $e)
{
    echo "Connection failed: " . $e->getMessage();
}
```

Figura 3.4: Script per la connessione al Database

Subito dopo aver stabilito la connessione, si indica che la codifica dei caratteri durante la trasmissione deve essere di tipo UTF-8. Il blocco try/catch per la gestione delle eccezioni viene utilizzato per gestire eventuali errori che si possono verificare nel tentativo di stabilire la connessione.

3.2.2 Salvataggio Foto da Flickr

Perchè Geo Photo Routing possa fornire il proprio servizio, è necessario che siano memorizzate sul Database foto geotaggate, a cui far riferimento per la creazione dei percorsi che aumentino il benessere. Per memorizzare sul database tali foto, si è utilizzata l'**API**

Flickr come spiegato nella sezione 2.4.2.
La memorizzazione è avvenuta in due differenti step:

- Inizialmente si sono memorizzate tramite un apposito script, una buona percentuale delle immagini presenti su Flickr, il cui geotag ricade all'interno delle quattro città italiane scelte (Torino, Milano, Bologna, Roma).
  Questa operazione ha richiesto un po' di tempo, non è stata immediata, ma è stato necessario farla solo una volta.

- Dopo la prima fase di memorizzazione, si è iniziata una procedura quotidiana, in cui ogni giorno vengono memorizzate sul Database le foto scattate e caricate su Flickr il giorno stesso, il cui geotag ricade sempre all'interno di queste 4 città italiane.
  In questo modo, il Database rimarrà sempre aggiornato, permettendo allo script che si occupa di caricare le foto, di eseguire velocemente ed in modo pulito.
  Inoltre, in questo modo, si va a risolvere un problema che era presente nella precedente versione di Geo Photo Routing. In essa, spettava all'utente il compito di aggiornare il Database. Ad egli era data la possibilità di caricare foto sul Database concernenti una determinata area geografica, nel caso in cui l'area in questione ne fosse priva ed egli volesse cercarne un percorso all'interno. Questa soluzione scarica sugli utenti tutta la responsabilità di aggiornare il Database e può essere un sollievo dal punto di vista del programmatore o di chi gestisce il sistema, ma in questo modo la *user experience* cala di qualità perchè l'utente deve aspettare anche parecchi secondi che le immagini vengano caricate sul Database prima di poter ricercare il percorso desiderato.
  Eseguendo, invece, uno script giornaliero sul Server, la user experience è più fluida: l'utente può ricercare direttamente il percorso sul territorio definito.

Per richiamare questa API, ci si è riferiti all'endpoint di Flickr, `https://api.flickr.com/services`, specificando, tramite un apposito URL, il metodo utilizzato, diversi parametri ed i loro valori, come illustrato di seguito in Figura 3.5.

```php
//Imposto i PARAMETRI per chiamata flickr
$method = "flickr.photos.search";
$api_key = "e1e8eea598XXXXXXXXX2122148a3e1";
$sort = "interestingness-desc";
$accuracy = 16;
$has_geo= 1;
$extras = "geo, url_l, date_taken, date_upload, tags";
$date = date('Y-m-d');


$bboxParams =  "7.053223,36.517362,18.391113,46.459620"; // ITALIA

$url = "https://api.flickr.com/services/rest/?method=".$method."&api_key=".$api_key.
"&sort=".$sort."&bbox=".$bboxParams."&accuracy=".$accuracy."&has_geo=".$has_geo.
"&extras=".urlencode($extras)."&per_page=500&page=1&format=json&nojsoncallback=1&min_upload_date=".$date;
```

Figura 3.5: I parametri della chiamata all'API Flickr.

La prima parte dell'URL https://api.flickr.com/services/**rest**/? specifica il formato di richiesta dei dati: in questo caso è REST.
Nella restante parte dell'URL vengono specificati i parametri necessari per il filtraggio dei dati richiesti ed i corrispondenti valori:

- **api_key** il valore di questo parametro contiene la stringa corrispondente alla chiave che è stata fornita da Flickr per poter utilizzare il loro servizio.
- **method** è il metodo API di Flickr utilizzato per inviare la richiesta, ovvero flickr.photos.search, che restituisce semplicemente le informazioni relative ad una lista di foto corrispondente ad alcuni criteri.
- **sort** indica l'ordinamento delle foto restituite Il valore attribuito parametro è *interestingness-desc* che ordina le fotografie dalla più alla meno rilevante sulla piattaforma Flickr.
- **bboxParams** è un elenco di quattro valori che definiscono il riquadro che delimita l'area geografica dentro la quale verranno cercate le foto. I quattro valori rappresentano l'angolo inferiore sinistro dell'area, cioè la longitudine minima e la latitudine minima, e l'angolo superiore destro, nello specifico la longitudine e la latitudine massima.
- **accuracy** specifica il livello di precisione desiderato per le informazioni sulla posizione. Il range dei valori che può assumere è [1, 16]. Nel caso specifico il valore del parametro è impostato a 16, ovvero la massima accuratezza corrispondente alla precisione a "livello di via".

- **has_geo** serve per specificare se si vogliono scaricare fotografie geotaggate o meno. In caso positivo, come quello in questione, il parametro assume valore "1". Se invece non si vogliono dati con geo-tag il valore deve essere impostato a "0".
- **extras** è formato da un elenco, i cui elementi sono suddivisi da virgole, di informazioni aggiuntive da recuperare per ogni foto restituita. Nel caso in questione, l'elenco prevede i parametri: "geo", per prelevare latitudine e longitudine di ogni fotografia; "url_l" per recuperare l'URL; "date_taken", corrispondente alla data in cui la foto è stata scattata e "tags", necessario per avere tutti i tag collegati a ciascuna foto.
- **per_page** rappresenta il numero di foto restituite ad ogni richiesta.
- **format** specifica il formato dei dati restituiti dall'API Flickr, nel nostro caso JSON.
- **min_upload_date** indica qual'è la data di riferimento da cui partire per cercare le foto. Verranno restituite solo foto caricate su Flickr dopo la data indicata. Questo parametro è utilizzato durante l'upload quotidiano delle foto sul Database di Geo Photo Routing, ed assume ogni volta il valore del giorno in questione, in modo che vengano caricate solo le foto relative all'ultima giornata disponibile.

Per collegarsi al servizio di Flickr, si è utilizzata una libreria di PHP, **cURL** [26]. Questa è una libreria, creata da Daniel Stenberg, che permette di connettersi e comunicare a diversi tipi di server utilizzando svariati protocolli, tra cui appunto HTTP GET, il protocollo utilizzato nel progetto.

La risposta dell'**API Flickr** è semplicemente un file in formato **JSON** in cui è presente un array chiamato *photo*, formato da elementi che rappresentano ognuno una foto di Flickr corredata da tutte le informazioni necessarie.

```
{
  "photos":{
    "page":4,
    "pages":16540,
    "perpage":250,
    "total":"4134968",
    "photo":[
      {
        "id":"27876456501",
        "owner":"46267286@N07",
        "secret":"11d183b68f",
        "server":"7324",
        "farm":8,
        "title":"The Great Conservatory Syon Gardens London - Through The Looking Glass b
        "ispublic":1,
        "isfriend":0,
        "isfamily":0,
        "dateupload":"1467096186",
        "datetaken":"2016-06-26 10:32:40",
        "datetakengranularity":"0",
        "datetakenunknown":"0",
        "tags":"crystalball sphere orb glass reflection water pond statue building islewo
        "latitude":"51.478102",
        "longitude":"-0.312724",
        "accuracy":"16",
        "context":0,
        "place_id":"COziR6VVUbk8Fw",
        "woeid":"43349",
        "geo_is_family":0,
        "geo_is_friend":0,
        "geo_is_contact":0,
        "geo_is_public":1,
        "url_l":"https:\/\/farm8.staticflickr.com\/7324\/27876456501_11d183b68f_b.jpg",
        "height_l":"664",
        "width_l":"1024"
```


Figura 3.6: Esempio di file JSON restituito dall'API Flickr.

Lo script presente sul server si occupa quindi di scorrere l'array in questione e, per ogni foto, chiamare due funzioni: *savePhotos* e *saveTags*. Queste si occupano rispettivamente di salvare sul database i dati della foto in oggetto e di salvare i tag relativi a tale foto.

3.2.3 API REST

Il Client, per visualizzare le Heatmap ed il percorso consigliato da Geo Photo Routing, ha bisogno di informazioni presenti sul Database, ma non può accederci direttamente. Il Server deve quindi mettere a disposizione queste informazioni al Client, e per far ciò si è scelto di creare un API con architettura REST.
Come ogni API REST, si necessita di un endpoint che fornisce il servizio. Esso è rappresentato dal file *function.php*, che include tutti i calcoli e le query necessarie per poter rispondere correttamente al Client. Il parametro principale da fornire all applicazione endpoint è il parametro *action*.
Questo parametro indica quale delle 3 possibili funzioni, l'applicazione deve eseguire. I 3 valori possibili sono:

- *async_returnCord*
- *async_returnCordForTag*
- *async_returnGeoPhotoPointsCoord*

Il primo di questi valori, indirizza l'applicazione endpoint ad una funzione che interroga il Database, selezionando le coordinate di TUTTE le foto presenti, e restituisce il risultato sotto forma di Array JSON.
La query di interrogazione al Database viene mandata in esecuzione con il metodo *query()* di **PDO**. Le informazioni restituite sono poi salvate in variabili e queste vengono convertite in un array. Prima di inviarlo al Client, l'array viene dato in pasto alla funzione *json_encode()* per trasformarlo in formato JSON.

```php
if($_REQUEST['action']=='async_returnCord'){
    $cordinate = array();
    foreach ($conn->query("SELECT latitude, longitude FROM photos") as $risultato)
    {
        $latitudine = $risultato['latitude'];
        $longitudine = $risultato['longitude'];
        $cordinate[] = array($latitudine, $longitudine);
    }
    $cordinate = json_encode($cordinate);
    //printiamo il nostro risultato
    echo $cordinate;
}
```

Figura 3.7: Script innescato dal valore async_returnCord del parametro action

Il secondo dei valori elencati accettati dal parametro *action*, indirizza ad una funzione che esegue gli stessi comandi di quella appena spiegata, ma con la differenza che accetta un parametro in più: *tag*. Questo parametro serve a scremare i risultati, selezionando dal Database solo le foto che hanno un determinato tag. Il funzionamento del resto dello script è esattamente identico al funzionamento dello script sopracitato.

Il terzo ed ultimo valore accettato dal parametro *action*, indirizza alla funzione principale di Geo Photo Routing: quella che calcola il percorso secondario, più piacevole da percorrere e la restituisce al Client sotto forma di Array di coordinate dei punti di interesse da cui passare.
L'API, in questo caso, ha bisogno di più parametri oltre ad *action*. Questi rappresentano:

- latitudine e longitudine del punto di partenza del percorso
- latitudine e longitudine del punto di arrivo del percorso
- raggio di ricerca scelto dall'utente

I passi per arrivare all'ottenimento delle coordinate dei punti di interesse da cui dover transitare sono:

1. Si chiama l'**API Directions** di Google per ottenere i punti che compongono il percorso standard tra i punti di partenza e di fine.
   Per completare questo passo non si fa niente di diverso rispetto a quanto fa il Client chiamando l'**API Directions** di Google: si accede tramite l'opportuno URL all'API, utilizzando la libreria **cURL**, e si ottiene il JSON contentente il percorso codificato.

2. Si decodifica il JSON così ottenuto, ed in particolare la *polyline* rappresentante l'intero percorso, esattamente come si è fatto nel Client. Si memorizzano quindi i punti geografici che compongono il percorso in un array.

3. Si considera ogni punto così trovato come il centro di un cerchio con raggio indicato dall'apposito parametro.

4. Per ognuno di questi cerchi che si sono venuti a creare, si memorizzano tutti i cluster ritenuti "interessanti", il cui centro ricade all'interno del cerchio. Un cluster viene ritenuto interessante se al suo interno ricadono più di 10 foto.
   La funzione utilizzata per memorizzare i cluster interessanti per un dato punto, è di seguito illustrata.

```php
// ** FORNISCE LE COORDINATE DEI PUNTI DI INTERESSE INTORNO AD UN CERTO RAGGIO
function getGPRPointsCoord($conn,$lat, $lng, $radius)
{
    $res = array();
    $radius = $radius/1000;

    $sql = "
    SELECT numPhoto, latCenter, lngCenter,  111.111 * DEGREES(ACOS(COS(RADIANS($lat))
                                                    * COS(RADIANS(latCenter))
                                                    * COS(RADIANS($lng - lngCenter))
                                                    + SIN(RADIANS($lat))
                                                    * SIN(RADIANS(latCenter)))) AS distance_in_km
    FROM mgrs_squares
    GROUP BY latCenter, lngCenter
    HAVING distance_in_km<'$radius' AND numPhoto>=10
    ORDER BY distance_in_km
    LIMIT 0, 300 ";

    foreach ($conn->query($sql) as $risultato)
    {
        $numPhoto = $risultato['numPhoto'];
        $latRet = $risultato['latCenter'];
        $longRet = $risultato['lngCenter'];

        $res[] = array($latRet, $longRet, $numPhoto);
    }

    return $res;
}
```

Figura 3.8: Script per cercare sul Database i cluster "interessanti" in un cerchio, con raggio e centro indicati dai parametri.

La funzione prende in input le coordinate di un punto geografico, il raggio di ricerca ed esegue una query sul Database che restituisce direttamente la lista di *cluster* ritenuti "interessanti".
In particolare la query scorre tutti i cluster presenti sul Database e controlla la loro distanza dal punto considerato come centro del cerchio e controlla il numero di foto presenti all'interno del cluster. Se la distanza è minore del raggio di ricerca dato e se il numero di foto è maggiore di 10, allora il punto può essere ritenuto interessante.

Per calcolare la distanza tra i due punti, è stata utilizzata la **legge dei coseni sferici**[27], che è un teorema della trigonometria sferica, che si può dimostrare tramite un opportuno procedimento, che specifica qual'è la relazione tra i coseni di un triangolo disegnato su di una sfera.
Il teorema esplica che: $cos(a) = cos(A)sin(b)sin(c) + cos(b)cos(c)$.
Questo teorema viene però utilizzato nel nostro caso con una sorta di formula inversa e ci permette di trovare la distanza di uno dei lati del triangolo disegnato sulla sfera, che nel nostro caso è la distanza tra i due punti geografici sulla Terra.

Tutto ciò di cui si ha bisogno sono le latitudini e longitudini dei due punti, che sono in pratica gli altri due lati del triangolo disegnato sulla sfera, la Terra.

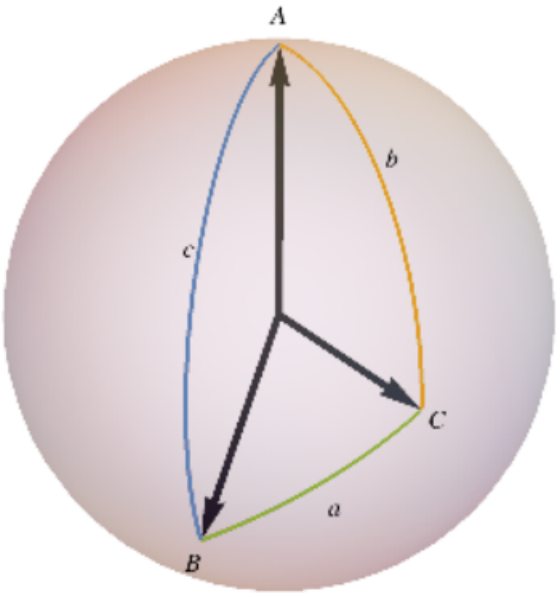

Figura 3.9: Esempio di utilizzo della legge dei coseni sferici.

Si noti che la costante "111.111" è il numero di chilometri per ogni grado di latitudine, basato sulla vecchia definizione Napoleonica del **metro**. Questo è definito come 1/10 000 000 del quarto del meridiano terrestre, compreso fra il polo nord e l'equatore.

5. Si considerano quindi tutti i punti di interesse che sono stati trovati in questo modo e si crea un opportuno **grafo orientato e pesato**, con l'ausilio di una **matrice di adiacenza**, in cui ogni nodo è rappresentato da uno dei cluster d'interesse.

6. Si calcola grazie all'**algoritmo di Dijkstra** il percorso minimo del grafo appena creato tra i punti di partenza e di arrivo, considerando le interconnessioni e i pesi stabiliti.

In questo modo si otterrà una lista di nodi del grafo da seguire per arrivare dal punto di partenza al punto di arrivo. I nodi in questione rappresentano i cluster che sono ritenuti "d'interesse", il cui centro geografico verrà ritornato al Client in risposta alla

sua richiesta. Quest'ultimo, una volta ottenuta la lista dei punti di interesse da cui passare, provvederà a creare un percorso che transiti da essi.
Il funzionamento specifico dell'algoritmo di Dijsktra e della creazione della matrice di adiacenza verrà spiegato nella prossima sezione.

3.3 Algoritmo di Dijkstra

Come spiegato nel capitolo 2.5, dopo un accurato ragionamento, si è deciso di utilizzare l'**algoritmo di Dijkstra** per la ricerca del percorso secondario: un percorso che deve essere il più breve possibile ma al contempo passare da eventuali punti di interesse situati lungo il percorso, prendendo piccole deviazioni rispetto al percorso principale. Questo algoritmo è spesso usato per la realizzazioni di reti idriche, stradali, circuitali, e in molti altri campi.

L'algoritmo di Dijkstra rientra tra gli algoritmi di *pathfinding*, il cui obbiettivo è ricercare il percorso più breve tra due nodi all'interno di un grafo pesato. In particolare, permette di trovare il **cammino minimo** (cioè il percorso più breve) tra due nodi all'interno di un grafo, che PUO' essere orientato[1] o ciclico[2], caratterizzato da archi con pesi NON negativi. Con "*cammino minimo*" si intende il percorso tra 2 nodi di un grafo, che minimizza la somma dei pesi degli archi attraversati.

[1] in un grafo orientato, gli archi sono coppie ordinate di nodi.

[2] un grafo è ciclico quando, per almeno un nodo, partendo da esso, esiste almeno un cammino che ritorna sullo stesso nodo

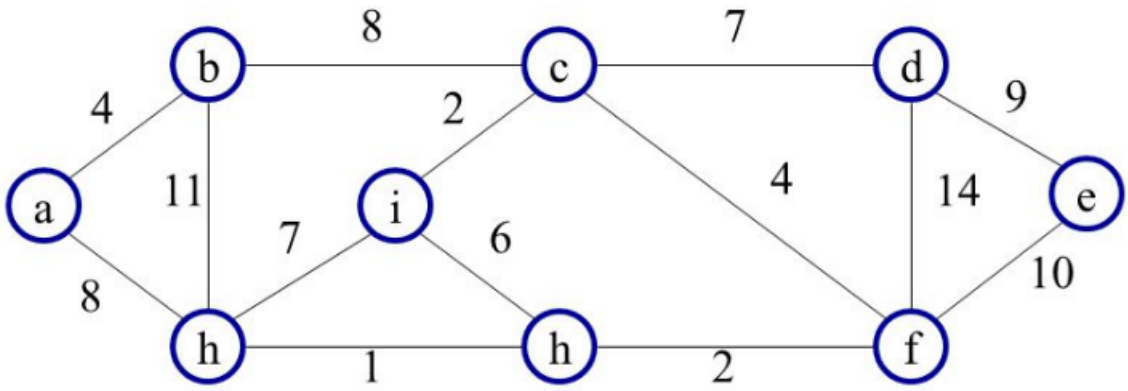

Figura 3.10: Esempio di grafo pesato, ciclico e non orientato.

Analizziamo ora brevemente il funzionamento generale dell'algoritmo di Dijkstra:
In ogni istante, l'insieme N dei nodi del grafo è diviso in tre parti:

- l'insieme dei nodi visitati **V**;
- l'insieme dei nodi di frontiera **F**, che sono successori[3] dei nodi visitati;
- i nodi **sconosciuti**, che sono ancora da esaminare.

Per ogni nodo z appartenente al grafo, l'algoritmo tiene traccia di un valore d_z, che rappresenta la distanza dal nodo in questione rispetto al nodo iniziale, da cui si parte per calcolare l'algoritmo. Inizialmente il valore d_z di ogni nodo è posto pari ad *infinito*, per indicare che il nodo è ancora nell'insieme dei **nodi sconosciuti**, tranne il valore d_z del nodo iniziale che viene posto pari a 0.
Inoltre, si tiene anche traccia, per ogni nodo z, del nodo u che lo precede, in modo che sia possibile ricostruire il cammino a partire dal nodo iniziale, fino ad arrivare a z.

[3]Un successore di z è un nodo raggiungibile lungo un arco uscente da z

Prima di iniziare effettivamente a calcolare l'algoritmo, si considera l'insieme **V** vuoto e si considera il nodo da cui si intende partire come l'unico appartenente all'insieme **F**. Tutti gli altri nodi appartengono per ora all'insieme dei nodi **sconosciuti**.
L'algoritmo consiste semplicemente nel ripetere i seguenti passi [28]:

1. Si prende dall'insieme **F** dei nodi di frontiera, il nodo z che ha d_z minimo;
2. Si sposta z da F in V;
3. Si spostano i successori di z **sconosciuti** in **F**
4. Si aggiornano i valori d_w di TUTTI i successori di z, cioè si aggiorna la distanza dei successori di z dal nodo origine. L'aggiornamento viene effettuato con la regola $d_w = \min \{ d_w , d_z + p_a \}$, dove a è l'arco che collega z a w.
   Quello che succede, cioè, è controllare, per ogni successore di z, se la sua distanza d_w dal nodo origine è minore della distanza d_z dal nodo origine più la lunghezza dell'arco p_a che porta da z a w. Se lo è, allora la distanza rimane invariata, altrimenti, se la nuova distanza trovata $d_z + p_a$ è minore di d_w, la distanza d_w del successore viene aggiornata a tale valore.
5. Per ogni successore di z, inoltre, si aggiorna il nodo predecessore u_w. Seguendo la regola di prima, se $d_z + p_a$ è minore di d_w, allora il nodo predecessore u_w viene cambiato e posto uguale a z. Altrimenti, se il valore di d_w rimane invariato, lo rimane anche il valore del nodo predecessore u_w.

L'algoritmo termina quando l'insieme **F** dei nodi di frontiera è vuoto, cioè quando non ci sono più nodi raggiungibili da visitare. Quando l'algoritmo è concluso, conosciamo con certezza, per ogni nodo z, qual'è la sua distanza d_z dal nodo origine e qual'è il suo predecessore u_z. Si può dimostrare tramite apposite proprietà e dimostrazioni che l'algoritmo funziona correttamente.
Tramite la successione dei nodi predecessori u_z, che vengono memorizzati dentro un array chiamato **array dei padri**[4], si può ricavare il **cammino minimo** dal nodo di partenza ad ogni altro nodo z, cioè si può capire quali sono i nodi intermedi da visitare per arrivare ad un certo nodo x, partendo dal nodo iniziale.

[4] in un array dei padri ogni posizione identifica un nodo ed il valore all'interno della posizione identifica il nodo padre del nodo in questione

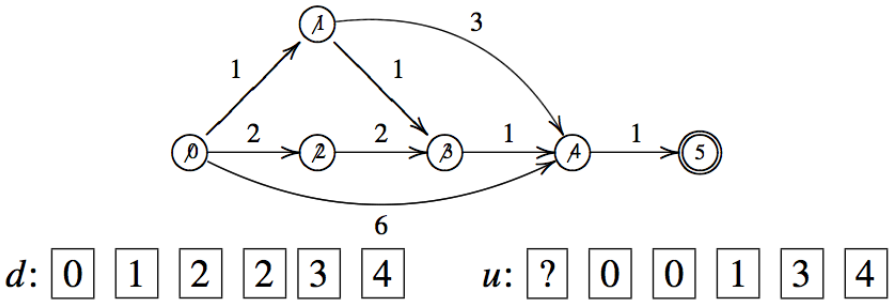

Figura 3.11: Esempio di fase finale dell'algoritmo di Dijkstra. Gli array "d" e "u" sono rispettivamente array delle distanze e dei padri.

Si può dimostrare che la complessità di quest'algoritmo è di **O(m * log(n))**, dove m rappresenta il numero di archi e n il numero di nodi.

Si spiega ora come si è implementato l'**Algoritmo di Dijkstra** all'interno del progetto di tesi.

Innanzitutto è necessario trovare un modo di rappresentare il grafo in linguaggio informatico. Le possibilità di scelta erano: **lista di adiacenza** o **matrice di adiacenza**. Si è scelto per comodità di utilizzare la matrice di adiacenza, che consiste un una matrice in cui ogni riga ed ogni colonna corrispondono ad un nodo ed ogni casella corrisponde al peso dell'arco che collega il nodo rappresentato dalla riga col nodo rappresentato dalla colonna. Se l'arco non esiste, il peso viene posto pari a 0.
In particolare si è scelto di utilizzare un grafo pesato e **orientato**. Nella matrice di adiacenza, quindi, le righe corrispondono ai nodi da cui parte l'arco e le colonne rappresentano i nodi a cui arriva l'arco.

Per la costruzione del grafo si sono considerati come nodi i punti geografici di partenza e di fine percorso, più tutti i punti geografici ritenuti "interessanti" trovati con l'algoritmo descritto nella sezione precedente. Si è creata quindi una serie di **archi orientati e pesati** che partissero da ogni nodo ed arrivassero ad ogni altro nodo, con peso pari a:

$$Peso = \frac{(\text{Distanza in metri tra i due punti geografici})^2}{\text{Numero immagini presenti nel nodo puntato dall'arco}} \tag{3.1}$$

Questo passaggio è importante e necessario perchè se il peso fosse solo uguale alla distanza tra i due punti, allora l'algoritmo troverebbe solo il percorso più breve; mentre se fosse solo uguale al numero di immagini del nodo puntato, allora l'algoritmo passerebbe da ogni punto ritenuto "interessante" senza curarsi della lunghezza del percorso.
Rapportando, invece, le due grandezze, il peso degli archi diventa proporzionato tra lunghezza del percorso ed interesse del nodo:

- maggiore è il numero di immagini nel nodo di arrivo, minore è il valore del peso, e quindi maggiore la "appetibilità" dell'arco che porta a quel nodo. E viceversa.
- maggiore è la distanza tra i due punti, maggiore è il peso risultante, e quindi minore la "appetibilità" dell'arco. E viceversa.

Si è impostato il numero di immagini presenti nel nodo di arrivo pari a **1**, data la sua neutralità rispetto alla divisione. In questo l'algoritmo considera il punto d'arrivo come privo di punti di interesse.
Inizialmente il peso era calcolato senza elevare al quadrato la distanza tra i due punti, ma si è notato che in questo modo l'altro fattore diveniva troppo rilevante e spesso non risultava la strada ottimale.

```php
function creaMatriceAdiacenza($GPRPoints){

    $matriceAdiacenza=[];

    // Tutti i nodi hanno archi uscenti tranne il nodo di arrivo
    for($i = 0; $i<sizeof($GPRPoints)-1;$i++){
        for($j = 0; $j<sizeof($GPRPoints); $j++){
            if($i!=$j){

                $distanceMeters = distanceBetweenPoints($GPRPoints[$i][0],
                    $GPRPoints[$i][1],$GPRPoints[$j][0],$GPRPoints[$j][1]);

                // PESO = (Distanza)^2 / n° immagini
                $matriceAdiacenza[$i][$j] = pow($distanceMeters,2)/$GPRPoints[$j][2];

            }
            else{
                $matriceAdiacenza[$i][$j] = 0;
            }
        }
    }

    // Il nodo di arrivo non ha archi uscenti
    for($j = sizeof($GPRPoints)-1; $j<sizeof($GPRPoints); $j++){
        if($i!=$j){
            $matriceAdiacenza[$i][$j] = 0;
        }
    }

    return $matriceAdiacenza;

}
```

Figura 3.12: Funzione per creare la matrice di adiacenza.

Lo script sopra illustrato serve per creare la matrice di adiacenza nella modalità spiegata: si crea un arco che esce da ogni nodo e va verso ogni altro nodo, con il peso indicato sopra. Eccetto per l'ultimo nodo che non ha archi uscenti, essendo il nodo di arrivo.
La funzione *distanceBetweenPoints* calcola la distanza in metri tra due punti geografici di cui si conosce latitudine e longitudine, utilizzando uno script trovato online, che sfrutta la **formula dell'emisenoverso**, una formula della trigonometria sferica utilizzata nella navigazione. Il funzionamento è simile alla formula spiegata nel capitolo 3.2.3.

Una volta calcolata la matrice di adiacenza, viene eseguito l'algoritmo vero e proprio, implementato in un file PHP a sè stante, chiamato *algDijkstra.php*. Per l'esecuzione dell'algoritmo è sufficiente conoscere la matrice di adiacenza, ed il nodo di partenza.
L'algoritmo implementato non ha nessuna variante rispetto al tipico algoritmo di Dijkstra. Utilizza una **coda di priorità**[5] per tenere in memoria i nodi appartenenti al-

[5]Una coda di priorità è una struttura dati astratta, in cui ogni elemento inserito all'interno della

l'insieme dei nodi di frontiera **F** ed una classe *ArcoDijkstra* per tenere in memoria ogni arco e le sue proprietà. L'algoritmo esegue e trova passo per passo i percorsi minimi per arrivare ad ogni nodo, fino a che la coda di priorità diventa vuota. Questo indica che la ricerca dei cammini minimi è finita.

Dopo che l'algoritmo è stato eseguito, viene restituito come risultato un array, chiamato **array dei padri**, in cui ogni posizione dell'array rappresenta un nodo ed il valore all'interno della posizione rappresenta il padre del nodo in questione.
Si scorre quindi a ritroso l'array, partendo dal nodo di arrivo, controllando qual'è il nodo padre, spostandosi sulla posizione del nodo padre e ripetendo l'operazione, finchè non si arriva ad analizzare il nodo di partenza. Questo sta ad indicare che la ricerca è finita e tutti i nodi che sono stati analizzati in questo procedimento sono i nodi da cui è necessario passare per minimizzare il percorso e di conseguenza sono i nodi che possono essere ritenuti *interessanti* per il percorso secondario.
Tali nodi vengono allora restituiti in risposta al Client in formato JSON.

3.4 Analisi della performance

In questa sezione si analizza empiricamente l'algoritmo per la ricerca del cammino col miglior rapporto brevità/interesse, analizzando la sua efficienza e la differenza del percorso ottenuto in questo modo rispetto al percorso ottenuto dall'algoritmo della precedente versione di Geo Photo Routing.
Sono state effettuate quindi diverse prove, mediante il calcolo di più percorsi con diverse caratteristiche, prendendo come riferimento le città di Torino, Milano, Bologna e Roma.
L'obbiettivo dell'analisi è dimostrare che il nuovo algoritmo, rispetto a quello vecchio:

- Non prenda in considerazione percorsi alternativi che aumentino di gran lunga la lunghezza generale dell'itinerario;
- Sappia riconoscere luoghi d'interesse che prima venivano tralasciati;
- Ottimizzi meglio il percorso alternativo, scegliendo in modo più intelligente i punti di interesse da cui passare, cosicchè la lunghezza del percorso finale sia più breve, senza tralasciare punti di interesse importanti

coda possiede una sua "priorità". Ogni elemento avente priorità più alta, viene inserito prima rispetto ad un elemento avente priorità più bassa. L'elemento con priorità più alta è quindi in testa e quello con priorità più bassa, in coda.

Per effettuare la valutazione della performance si è scelto di effettuare il calcolo di 20 percorsi, con lunghezze diverse, nelle varie città, sia con la vecchia versione di Geo Photo Routing, sia con la nuova versione. Il raggio di ricerca dei punti di interesse è fissato a 1000 metri.
Ad ogni ricerca sono stati analizzati 2 parametri:

- La differenza tra la lunghezza di ognuno dei 2 percorsi alternativi della vecchia e nuova versione di Geo Photo Routing, e la lunghezza del percorso proposto da Google Maps.
- Il numero di luoghi di interesse trovati dagli algoritmi di entrambe le versioni di Geo Photo Routing.

È chiaro che un algoritmo è tanto migliore quanto più breve è il percorso e quanti più punti di interesse trova. Di seguito si mostrano due grafici che espongono i risultati ottenuti:

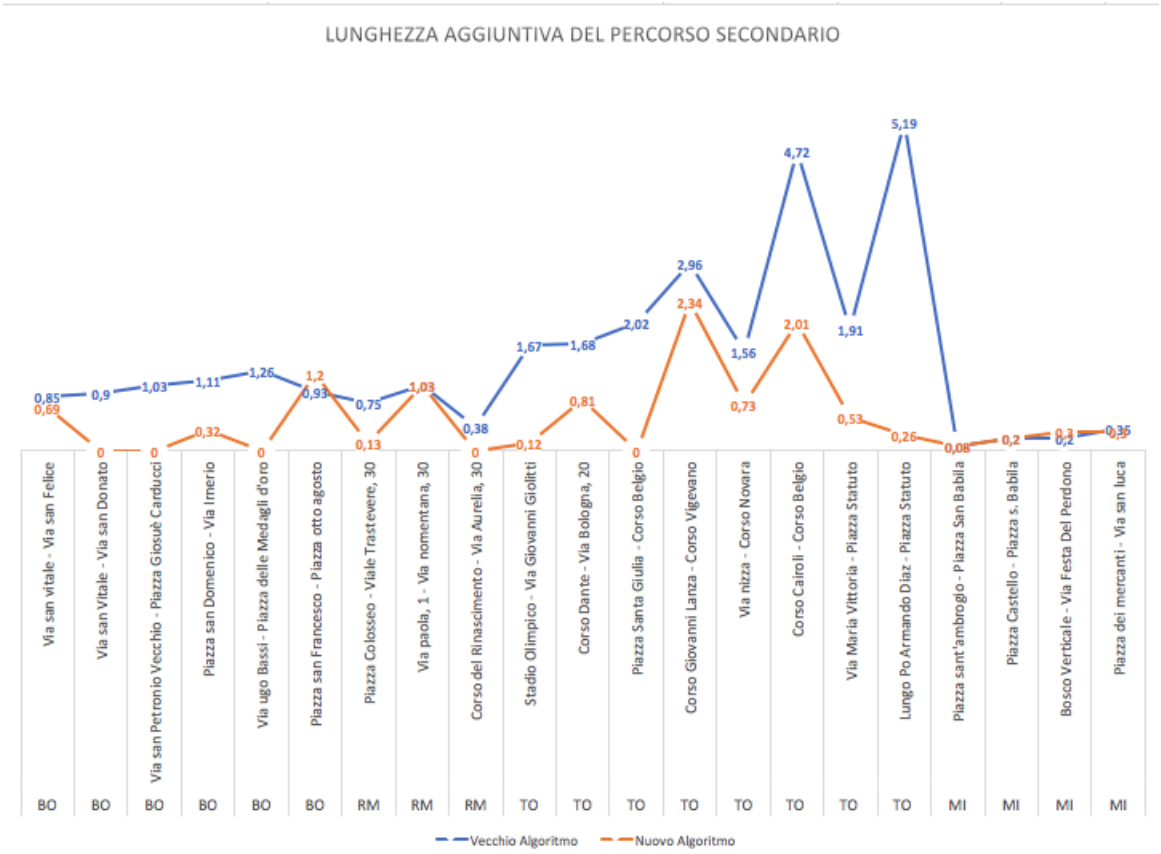

Figura 3.13: Grafico che mostra la differenza tra la lunghezza di ognuno dei due percorsi e la lunghezza del percorso proposto da Google Maps.

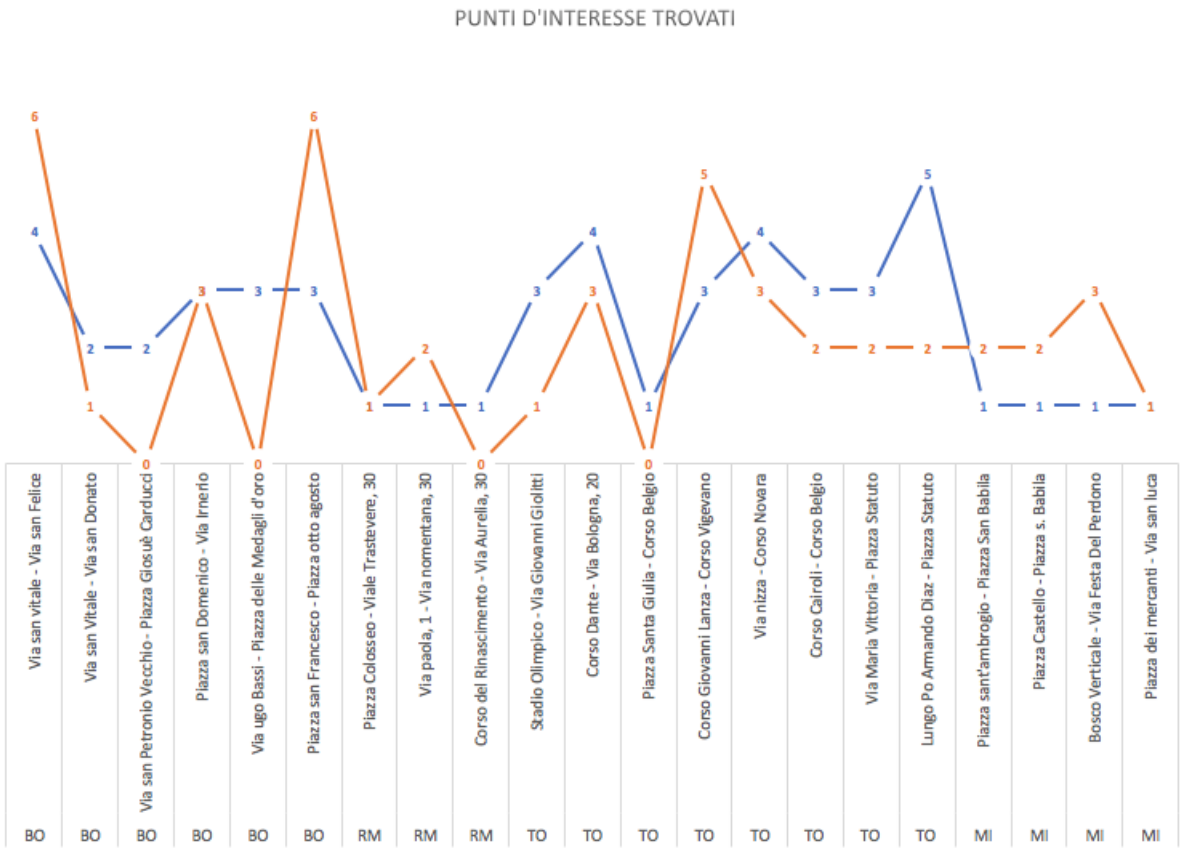

Figura 3.14: Grafico che mostra il numero di punti di interesse trovato da entrambi gli algoritmi.

Si può notare subito come il nuovo algoritmo rende quasi sempre il percorso alternativo più breve del vecchio algoritmo, talvolta anche di svariati chilometri. Allo stesso tempo, però, spesso il nuovo algoritmo rinuncia alla visita di luoghi interessanti.
Tra i due parametri, viene considerata di maggiore importanza la lunghezza del percorso: si cerca quindi di trovare un percorso alternativo che sia il più breve possibile, e che, se possibile, passi per qualche punto di interesse senza allungare esageratamente la lunghezza del percorso.
Si può ritenere l'algoritmo migliorato rispetto a quello precedente, in quanto l'obbiettivo iniziale non era fornire un percorso che passasse da più punti di interesse possibile, ma fornire un itinerario che potesse aumentare il benessere del tragitto, eventualmente allungando di poco la lunghezza del percorso.

Si mostra ora qualche esempio visuale dei miglioramenti avvenuti nell'algoritmo, partendo con percorsi brevi già mostrati nelle figure 2.13 e 2.14.
Vicino questi percorsi non sono presenti punti di interesse, ma nella prima versione di Geo Photo Routing, la lunghezza del percorso veniva trascurata e tratte di pochi metri venivano trasformate in tratte lunghe centinaia di metri, o più.
Utilizzando però la nuova tecnica di calcolo del percorso, viene dato un peso maggiore alla distanza dei nodi del grafo, in modo che l'algoritmo riesca a capire quanto sia infruttuoso, in alcuni casi, passare dai luoghi d'interesse. Di seguito due immagini che contengono lo stesso itinerario delle figure 2.13 e 2.14 ricalcolato con il nuovo algoritmo.

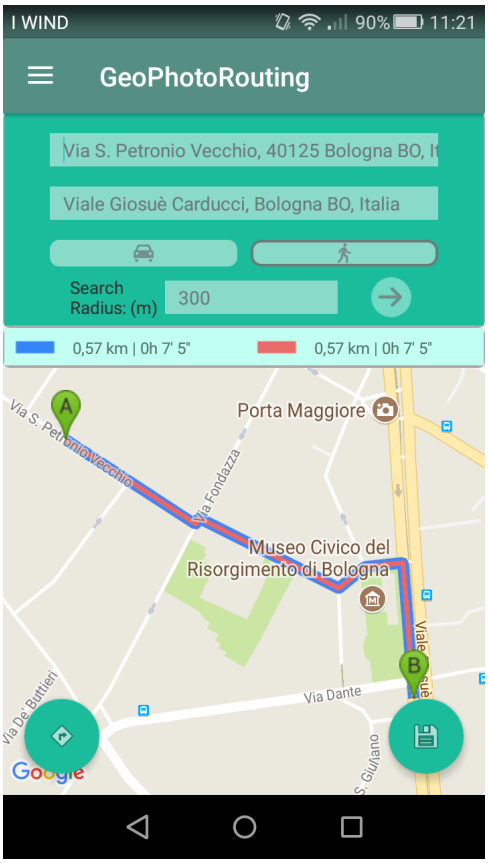

Figura 3.15: Esempio di percorso che non passa da punti di interesse calcolato dall'algoritmo rinnovato.

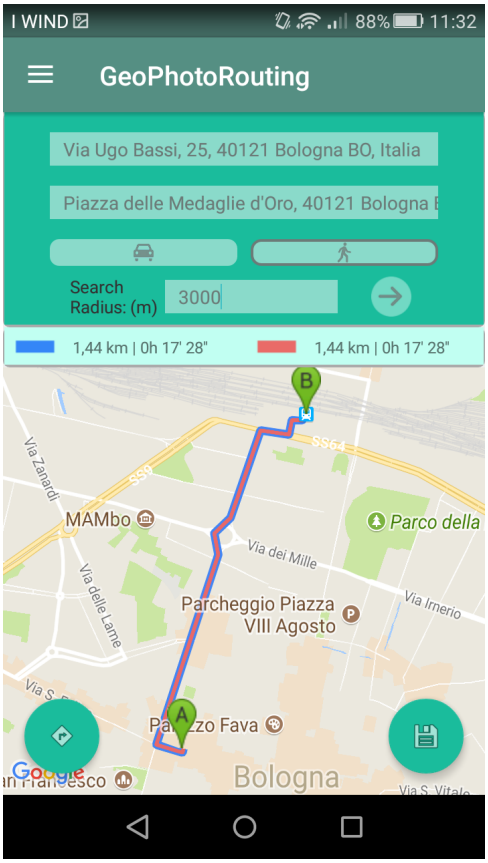

Figura 3.16: Esempio di percorso che non passa da punti di interesse calcolato dall'algoritmo rinnovato.

Si nota come l'itinerario secondario, a differenza della precedente di versione di Geo Photo Routing, è esattamente identico all'itinerario proposto da Google Maps, in quanto l'algoritmo non ha ritenuto necessario o conveniente passare dai punti di interesse, in quanto troppo lontani.

Si vuole far notare ora, tramite un esempio concreto, come l'utilizzo di *cluster* per il calcolo dell'algoritmo, aiuti a riconoscere luoghi di interesse come tali. In questo particolare esempio, si vuole far notare come le due torri di Bologna, vengano riconosciute dalla nuova versione di Geo Photo Routing, ma non dalla vecchia.

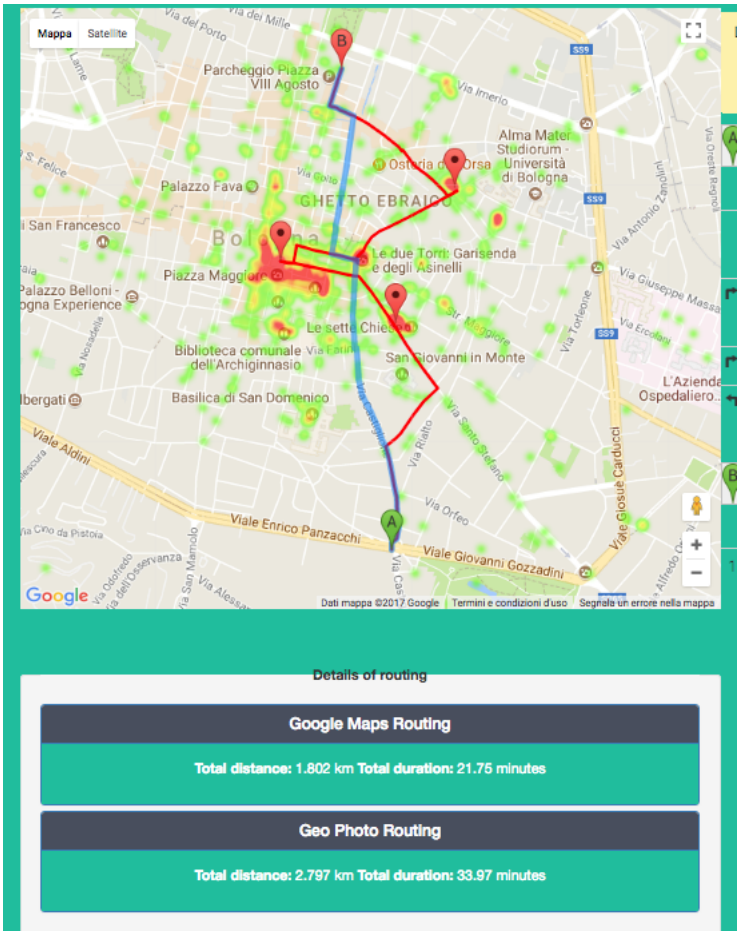

Figura 3.17: Esempio di percorso che NON identifica i giusti luoghi di interesse.

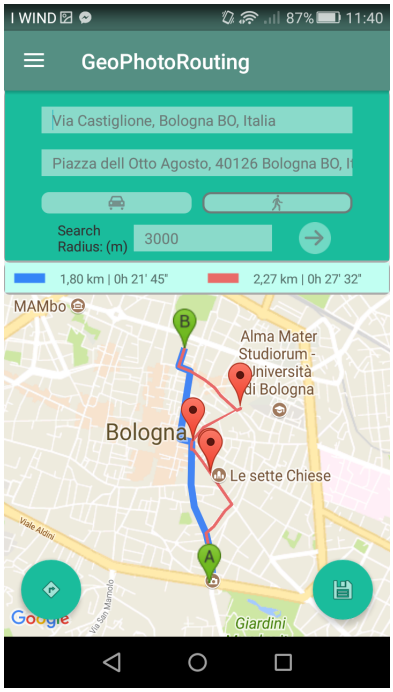

Figura 3.18: Esempio di percorso che identifica i giusti luoghi di interesse.

Si è scelto, infine, di misurare la rapidità di esecuzione dell'algoritmo del presente progetto di tesi. Facendo ciò si vuole dimostrare che, anche aumentando la distanza tra i punti di partenza e di arrivo ed aumentando il raggio di ricerca in modo da comprendere più punti di interesse possibile, l'algoritmo restituirà il risultato in un tempo del tutto ragionevole. Si vuole dimostrare quindi, che la velocità dell'algoritmo non è influenzata dal cambiamento dei parametri di ricerca.
Per dimostrare questo, è stato eseguito un test in cui sono state scelte 3 tratte di diversa lunghezza, e 3 diversi raggi di ricerca a cui corrisponde una diversa quantità di punti di interesse. Si è ripetuto il calcolo del percorso più volte e si è fatta una media di tempo impiegato nel calcolo. Di seguito una tabella che mostra i risultati del test:

Come è possibile notare dalla tabella 3.1, calcolare un percorso di lunghezza maggiore aumenta leggermente la durata del calcolo, ma si parla di un aumento non percepibile dall'utente. Lo stesso discorso si applica sull'aumento del raggio di ricerca. Ciò dimostra che la velocità dell'algoritmo non è influenzata in modo negativo dai parametri di ricerca.

Non è stato possibile confrontare l'algoritmo della prima versione con il nuovo algoritmo poichè la struttura dell'implementazione dell'algoritmo è completamente diversa nelle

Tabella 3.1: Risultati test di efficienza dell'algoritmo

	Radius: 100m	Radius: 1000m	Radius: 10000m
P: Università Bocconi, Via Roberto Sarfatti (MI) A: Alcatraz, Via Valtellina (MI)	$\sim$ 0.18s	$\sim$ 0.19s	$\sim$ 0.20s
P: Via S. Vitale, Bologna A: Via S. Felice, Bologna	$\sim$ 0.14s	$\sim$ 0.15s	$\sim$ 0.155s
P: Milano MI A: Roma RM	$\sim$ 0.34s	$\sim$ 0.35s	$\sim$ 0.36s

due versioni ed un confronto sarebbe poco significativo ed efficace.

Capitolo 4

Conclusioni

L'obbiettivo del presente progetto di tesi è stato la creazione di un'applicazione Android, e di una relativa applicazione server-side, che potesse suggerire un itinerario da un punto di partenza ad un punto di arrivo, che non fosse solo il più breve possibile, ma che fosse più piacevole da percorrere, che aumentasse il benessere generale.
Per poter capire quali sono gli itinerari alternativi che aumentano il benessere di chi li percorre, si è deciso di fare affidamento alle fotografie con geolocalizzazione provenienti dalla piattaforma Flickr.com, in quanto lo scatto e la condivisione di siffatte foto, può essere considerata come un'opinione positiva del luogo a cui fanno riferimento, il quale, quindi, viene considerato piacevole da visitare.
Si è quindi costruito un'algoritmo, basato sull'algoritmo di Dijkstra, che valutasse quali sono i luoghi geografici, in cui sono presenti foto geotaggate di Flickr, da cui convenisse transitare per migliorare il percorso standard tra due punti.
L'architettura della piattaforma si compone di un'applicazione Client-side, caratterizzata più precisamente da un'applicazione Android, che ha l'obbiettivo di ricevere gli input e le richieste dell'utente; di effettuare opportune chiamate all'applicazione Server-side, tramite chiamate ad un'API REST; e di visualizzare in modo opportuno le risposte del server, per esempio visualizzando il percorso ottimale calcolato.
L'applicazione Server-side si occupa quindi di mettere a disposizione dei Client un'API REST, che riceve le richieste ed indirizza verso la corretta funzione da eseguire. Il ruolo più importante del server è quello di calcolare il percorso alternativo ottimale dato un punto di partenza ed uno di arrivo. Per fare ciò, esso dialoga con il Database per ottenere informazione relative alle immagini geotaggate di Flickr ed esegue un algoritmo opportunamente ottimizzato per capire quali sono i luoghi geografici da cui conviene passare per aumentare il benessere del percorso, senza aumentare troppo la lunghezza complessiva di esso.

Il progetto di tesi è in realtà la prosecuzione di un'altro progetto di tesi, con gli stessi obbiettivi, ma con molte differenze a livello implementativo. Il progetto iniziale consisteva nella creazione di un'applicazione web che consigliasse lo stesso tipo di percorso alternativo tra due punti: un percorso più piacevole da percorrere, utilizzando le immagini geotaggate di Flickr. Il progetto era, però, quasi un prototipo, al quale potevano essere fatte molte migliore.
Nell'attuale progetto, tra le tante migliorie possibili:

- è stata creata un'applicazione Android, che avesse all'incirca le stesse funzionalità dell'applicazione web precedente;
- è stato migliorato l'algoritmo di calcolo del percorso alternativo, utilizzando l'algoritmo di Dijkstra, in modo ottimizzare il rapporto $\frac{\text{Distanza dei punti di interesse}}{\text{Valore emotivo che essi hanno}}$;
- è stata migliorata la memorizzazione delle immagini da Flickr sul Database, rendendola automatica, senza un'interazione utente, in modo da rendere fluida la *User experience*.

Come la prima versione di Geo Photo Routing, anche l'attuale piattoforma non può essere considerata definitiva. Seppur vari miglioramenti sono stati apportati, al progetto si possono applicare ancora diverse migliorie ed estensioni, quali:

- Permettere all'utente di inserire una durata e una distanza massime e calcolare il percorso Geo Photo Route tenendo conto anche di queste proprietà;
- Permettere all'utente di scegliere una preferenza tra brevità del percorso e benessere dato dal cammino.
  Il percorso alternativo potrebbe passare da molti più punti di "interesse" con lo svantaggio di allungare il percorso. Spetta, però, all'utente scegliere cosa preferire tra brevità del percorso ed importanza data ai punti di interesse;
- Utilizzare i tags delle fotografie come filtri, cioè consentire all'utente di inserire una parola chiave e calcolare il Geo Photo Route considerando solo le geolocazioni delle fotografie che possiedono un tag corrispondente alla parola fornita;
- Includere nel Database le immagini relative a tutto il territorio italiano. Senza nessuna grossa difficoltà, si potrà anche espandere la geolocazione delle immagini a tutto il territorio Europeo ed anche oltre;
- Aumentare il numero di immagini geotaggate presenti sul Database, sfruttando altre API di servizi di condivisione immagini, oltre a Flickr. Per esempio, si potrebbe integrare il Database con le immagini proveniente dall'API di Intsagram;

- Migliorare ulteriormente l'algoritmo di ricerca del percorso alternativo ottimale, riuscendo a trovare percorsi "felici" anche là dove non esistono punti di interesse riconosciuti grazie alle immagini geotaggate. Permettere, quindi, una seconda modalità di riconoscimento di percorsi alternativi.

Queste ed altre sono le possibilità di ampliamento del progetto di tesi.

Bibliografia

[1] M. Coverley. *Psychogeography*. Pocket Essentials, 2006

[2] A Note on Two Problems in Connexion with Graphs. *E. W. Dijkstra*. http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf

[3] Does Google Maps use Dijkstra's algorithm? If so, can you explain how they discretize the world? https://www.quora.com/Does-Google-Maps-use-Dijkstras-algorithm-If-so-can-you-explain-how-they-discretize-the-world

[4] Syed Abdullah Fadzli, Sani Iyal Abdulkadir, Mokhairi Makhtar, Azrul Amri Jamal. "*Robotic Indoor Path Planning using Dijkstra's Algorithm with Multi-Layer Dictionaries*". Published in "Information Science and Security (ICISS), 2015 2nd International Conference on".

[5] Y.Q. Liu, Q.L. Huang, Y.X. Zhang, G.F. Luan M.H. Xu, "*An improved Dijkstra's shortest path algorithm for sparse network*". Applied Mathematics and Computation, vol. 185, no. 1, pp. 247-254, February 2007.

[6] Jennifer C. Dela Cruz, Glenn V. Magwili, Juan Pocholo E. Mundo, Giann Paul B. Gregorio, Monique Lorraine L. Lamoca, Jasmin A. Villaseñor. *Items-mapping and Route Optimization in a Grocery Store using Dijkstra's, Bellman-Ford and FloydWarshall Algorithms* . Published in: "Region 10 Conference (TENCON), 2016 IEEE.

[7] Kazunori Uchida, Shinsuke Nogami, Masafumi Takematsu, and Junichi Honda. *Tsunami Simulation Based on Dijkstra Algorithm*. Published in: "2014 17th International Conference on Network-Based Information Systems".

[8] Kazunori Uchida e Leonard Barolli. *Dijkstra-Algorithm Based Ray-Tracing by Controlling Proximity Node Mapping*. Published in: "2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)".

[9] Ireneusz Szczesniak e Bozena Wozna-Szczesniak. *Adapted and Constrained Dijkstra for Elastic Optical Networks*. Published in "2016 International Conference on Optical Network Design and Modeling (ONDM)".

[10] Muh. Ansto Indrajaya, Achmad Affandi, Istas Pratomo. *Design of geographic information system for tracking and routing using Dijkstra algorithm for public transportation*. Published on: "2015 1st International Conference on Wireless and Telematics (ICWT)".

[11] Daniele Quercia, Rossano Schifanella, Luca Maria Aiello. "*The Shortest Path to Happiness: Recommending Beautiful, Quiet, and Happy Routes in the City*". In Proc. of Conference on Hypertext and Social Media (HyperText), 2014.

[12] Cecilia Toccaceli. *GEO PHOTO ROUTING: raccomandazione dell'itinerario di viaggio utilizzando i geotag dei siti di condivisione foto*. Alma Mater Studiorum - Università di Bologna.

[13] Architettura a tre livelli.
`http://it.ccm.net/contents/136-sistema-client-server`

[14] Modello Client-Server.
`https://www.ibm.com/support/knowledgecenter/it/SSAW57_7.0.0/com.ibm.websphere.nd.doc/info/ae/ae/covr_3-tier.html`

[15] Gradle.
`http://www.androidos-lab.it/2015/02/01/gradle-che-cose/`

[16] Android Manifest.
`https://developer.android.com/guide/topics/manifest/manifest-intro.html`

[17] Google Place Autocomplete.
`https://developers.google.com/places/android-api/autocomplete`

[18] PHP.
`http://php.net/manual/it/index.php`

[19] API Flickr.
`https://www.flickr.com/services/api/`

[20] UTM - Universal Transverse Mercator.
`http://geokov.com/education/utm.aspx`

[21] How to Read a United States National Grid (USNG) Spatial Address.
`https://www.fgdc.gov/usng/how-to-read-usng/index_html`

[22] Google Maps with Military Grid Reference System (MGRS) Coordinates.
`https://mappingsupport.com/p/coordinates-mgrs-google-maps.html`

[23] PHP Standalone MGRS - LatLng Converter. Copyright (C) 2014 J42 (Julian Aceves). `https://gist.github.com/j42/2aae5a360624ae2a43d7`

[24] I principi dell'architettura RESTful.
`http://www.html.it/pag/19596/i-principi-dellarchitettura-restful/`

[25] JSON.
`http://www.html.it/faq/cose-json/`

[26] cURL.
http://php.net/manual/en/book.curl.php

[27] Legge dei coseni sferici
`https://staff.imsa.edu/~fogel/ModGeo/PDF/04%20Spherical%20Trigonometry.pdf`

[28] Algoritmo di Dijkstra.
`http://ioi.di.unimi.it/dijkstra.pdf`