Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

IMPLEMENTAZIONE E VALIDAZIONE DI MOBILITÀ REATTIVA PER LA SIMULAZIONE A EVENTI DI SCENARI MOBILE CROWDSENSING

Relatore:
Chiar.mo Prof.
Luciano Bononi

Correlatore:
Dr.
Federico Montori

Presentata da:
Andrea Zapparoli

Sessione 1
Anno Accademico 2019/2020

Sono capaci tutti di non farcela, non serve nemmeno impegnarsi molto.

Sommario

Il Mobile Crowdsensing (MCS) è una ramificazione del paradigma Crowdsensing. Quest'ultimo indica una raccolta di dati con l'aiuto di utenti tramite i dispositivi personali o a loro distribuiti per studiare certi fenomeni. MCS precisa che i dispositivi sono mobili, e grazie alla loro crescente diffusione in questi decenni ne ha guadagnato in popolarità.
L'efficienza di MCS risiede in un elevato numero di partecipanti, spesso reclutati grazie ad incentivi. Per ragioni di costi e tempo, l'analisi in ambienti urbani reali è spesso impraticabile, la valida alternativa è rappresentata dai simulatori. La tesi propone nuove funzionalità incorporate in CrowdSenSim, simulatore stateful per lo sviluppo di sistemi MCS in ambienti urbani reali, mantenendo retro-compatibilità. L'obiettivo principale è gestire la copertura del territorio, grazie all'estensione dell'architettura, introducendo il cambio di percorso degli utenti soddisfacendo le richieste del server predisposto a questo controllo. La dimostrazione avviene adottando come caso di studio un algoritmo di raccolta dati, modificato con le nuove caratteristiche, testato su 3 città diverse per conformazione urbana e dimensione.

Introduzione

Mobile Crowdsensing (MCS) è uno dei più promettenti paradigmi per il rilevamento e la raccolta di dati in un ambiente urbano in ambito smart city, e negli ultimi anni ha guadagnato enorme popolarità. La particolarità della raccolta è che avviene per mezzo dei sensori disponibili nei dispositivi mobili che appartengono ai cittadini, come smartphone, tablet e orologi indossabili che contengono accelerometro, giroscopio, GPS, microfono, fotocamera e non solo. Oggi sono strumenti indispensabili per molte attività quotidiane, che vanno dal lavoro, la comunicazione e l'intrattenimento. Grazie alla loro crescente diffusione e tecnologia negli anni sono diventati un'ottima fonte di informazioni, congiuntamente a costi economici inferiori e di risorse per i settori che se ne servono per le proprie ricerche.
L'efficienza di MCS si basa sulla partecipazione e contribuzione di un elevato numero di utenti e maggior copertura territoriale. Questo permette di ottenere una quantità di dati difficilmente raggiungibile con metodologie classiche, di conseguenza consente di studiare più nel dettaglio certi fenomeni, in una moltitudine di scenari, ed offrire servizi efficaci anche a beneficio dei cittadini, ad esempio tramite la creazione di applicazioni mobile. I ricercatori hanno sviluppato diverse tecniche di gestione della privacy, di ottimizzazione della raccolta dati limitando il consumo di banda e batteria, e hanno pensato a tipologie di incentivi per contrastare la riluttanza alla partecipazione degli utenti. Nonostante ciò, spesso non è fattibile impiegarlo nel mondo reale a causa dei costi crescenti all'aumentare degli aderenti, ed al tempo necessario per l'intera organizzazione.

L'alternativa ottimale è rappresentata dai simulatori, i quali permettono di analizzare fenomeni di sistemi MCS in ambienti urbani, con libertà di scelta del numero di cittadini, in tempi ragionevoli.
CrowdSenSim, attualmente giunto alla seconda versione, è il simulatore stateful per lo sviluppo di sistemi MCS in ambienti urbani reali studiato e modificato in questo lavoro. Consente l'osservazione di aspetti quali consumo energetico e raccolta dati, di generare mobilità reale degli utenti e offre vari algoritmi di raccolta dati. Il limite principale è il percorso degli partecipanti assegnato, il quale rimane immutato. Nel caso di aree territoriali con poche letture, non esiste soluzione. Le modifiche aiutano a risolvere il problema, implementando il concetto di *participatory crowdsensing* per la gestione di dati sparsi. In particolare, i contributi di questo lavoro sono:

- creazione del server preposto al controllo dell'ammontare dei dati delle zone MGRS, il quale può effettuare una richiesta di deviazione agli utenti, che sono liberi di accettare o meno;
- ricalcolo dei percorsi per adattarli alle richieste del server;
- sviluppo di un caso d'uso, cioè l'applicazione finale per la validazione del lavoro, modificando un algoritmo distribuito opportunistico per la raccolta di dati, testata su 3 città, e di una classe per stimare le prestazioni. I risultati sono illustrati per mezzo di grafici e *heatmap*;
- possibilità di proseguire le simulazioni offline, e risoluzione di bug portati dalla versione precedente.

In conclusione, la tesi ha la seguente struttura: il capitolo 2 presenta il simulatore da cui è iniziato questo lavoro, ovvero nella sua seconda versione. Il capitolo 3 spiega la progettazione e l'implementazione delle *feature* aggiunte. Il capitolo 4 riguarda la validazione dell'applicazione, analizzando i risultati ottenuti dai test, e le prestazioni delle nuove caratteristiche. Infine, il capitolo 5 conclude la tesi.

Indice

Introduzione **iii**

1 Stato dell'arte **1**

1.1 Internet of Things 1

1.2 Crowdsensing 3

1.2.1 Risparmio energetico 6

1.2.2 Privacy 7

1.2.3 Coverage 8

1.3 Simulatori 9

1.4 Contributi 11

2 CrowdSenSim 2.0 **13**

2.1 Architettura 14

2.1.1 Event Generator 14

2.1.2 Simulator engine 19

2.2 File configurazione 21

2.3 Librerie 23

2.3.1 NetworkX 23

2.3.2 OSMnx 24

2.3.3 MGRS 24

2.3.4 Chart.js 25

2.4 Algoritmi di CrowdSenSim 25

2.4.1 Send Always 25

2.4.2 Deterministic Distributed Framework (DDF) 25

2.4.3 Piggyback CrowdSensing (PCS) 26
2.4.4 Probabilistic Distributed Algorithm (PDA) 26
2.4.5 Ricezione SI 29
2.5 Limiti del simulatore 29

3 Progettazione ed implementazione **31**
3.1 Applicazione finale 31
3.2 Server 36
3.3 Cambio direzione - C++ 38
3.4 Cambio direzione - Python 43
3.5 PROBA 47
3.6 Script e bug 49

4 Validazione **51**
4.1 Città 51
4.2 Performance 54
4.3 Validazione dell'applicazione 56

Conclusioni **63**

A Heatmap delle città **65**

Bibliografia **81**

Elenco delle figure

2.1 Architettura modulare CrowdSenSim 2.0 15
2.2 Grafo di Bologna semplificato 16

3.1 Architettura modulare nuova 32
3.2 Percorso originale di un utente nella città di Bologna e percorso fino a punto di deviazione 45
3.3 Percorso originale di un utente nella città di Bologna e nuovo percorso 46

4.1 Mappa OSMnx di Bologna con quadrati MGRS 52
4.2 Mappa OSMnx di Lussemburgo con quadrati MGRS 54
4.3 Mappa OSMnx di Melbourne con quadrati MGRS 55
4.4 Test Performance 56
4.5 Heatmap AO-JFS di Lussemburgo 60
4.6 Heatmap JFS-IBRIDO per destinazione di Lussemburgo . . . 60
4.7 Heatmap JFS-IBRIDO per probabilità di Lussemburgo 61
4.8 Heatmap differenza JFS-IBRIDO per destinazione e JFS di Lussemburgo 61
4.9 Heatmap differenza JFS-IBRIDO per probabilità e JFS di Lussemburgo 62

A.1 Heatmap AO-JFS di Bologna 65
A.2 Heatmap JFS-IBRIDO per destinazione di Bologna 66
A.3 Heatmap JFS-IBRIDO per probabilità di Bologna 67

A.4 Heatmap differenza JFS-IBRIDO per destinazione e JFS di Bologna 68
A.5 Heatmap differenza JFS-IBRIDO per probabilità e JFS di Bologna 69
A.6 Heatmap AO-JFS di Lussemburgo 70
A.7 Heatmap JFS-IBRIDO per destinazione di Lussemburgo . . . 71
A.8 Heatmap JFS-IBRIDO per probabilità di Lussemburgo 72
A.9 Heatmap differenza JFS-IBRIDO per destinazione e JFS di Lussemburgo 73
A.10 Heatmap differenza JFS-IBRIDO per probabilità e JFS di Lussemburgo 74
A.11 Heatmap AO-JFS di Melbourne 75
A.12 Heatmap JFS-IBRIDO per destinazione di Melbourne 76
A.13 Heatmap JFS-IBRIDO per probabilità di Melbourne 77
A.14 Heatmap differenza JFS-IBRIDO per destinazione e JFS di Melbourne 78
A.15 Heatmap differenza JFS-IBRIDO per probabilità e JFS di Melbourne 79

Capitolo 1

Stato dell'arte

1.1 Internet of Things

Internet delle cose (IoT, acronimo dell'inglese *Internet of Things*) è il termine con il quale ci si riferisce alla capacità di alcuni oggetti di essere connessi ad Internet permettendo loro di interagire fra di essi e con le persone. Il termine venne coniato nel 1999 dal ricercatore britannico Kevin Ashton all'epoca ricercatore del Massachusetts Institute of Technology di Boston.
Questa evoluzione consente agli oggetti di essere riconoscibili e di rendere disponibili i dati necessari per comprendere meglio il mondo reale, di accedere ai dati di altri ricavando così anche informazioni utili per processi decisionali. L'obiettivo dunque è creare una mappa del mondo reale con gli oggetti elettronici, chiamati *smart objects*, identificandoli e localizzandoli, grazie alla loro capacità di elaborazione dati e interazione con l'ambiente esterno.
Gli oggetti "intelligenti" sono molteplici, alcuni di uso comune, come elettrodomestici, termostati, veicoli, telecamere, dispositivi wearable (oggetti da indossare), schede come Arduino e Raspberry Pi, e molto altro, che grazie ai rispettivi sensori raccolgono dati e comunicano sfruttando tecnologie di comunicazione tipiche in ambito IoT. Tra le misurazioni tipiche

troviamo ad esempio temperatura, movimento (sensori di movimento), luminosità, umidità, rumore, onde elettromagnetiche ecc. Gli ambiti di applicazione sono vari: domotica (tecnologie per la casa), smart building (edifici intelligenti), industria, automotive, sanità, città (smart city), agricoltura ecc.
L'IoT nell'ambito della domotica può essere rappresentato da un frigorifero che si accorge della mancanza di un certo prodotto e lo segna nella lista della spesa. Oppure l'accensione o spegnimento di una lampadina in base alla presenza di persone nella stanza.
Nello smart city [22] può essere rilevata la congestione del traffico, informazione utile per i cittadini consentendo loro di pianificare in anticipo un diverso percorso verso la destinazione, ma anche per le autorità che possono intervenire gestendo la situazione. In alternativa, semafori che si autoregolano. Un altro esempio riguarda l'ottimizzazione dell'efficienza dell'illuminazione stradale in base all'ora del giorno, alle condizioni meteorologiche e alla presenza di persone.
La progressiva diminuzione del costo medio di sensori, attuatori e microcontrollori e delle loro dimensioni, insieme all'ampia copertura e disponibilità di internet, ha permesso un enorme crescita di questa tecnologia e della sua diffusione.
La presenza sempre più massiccia di queste "things" porta al concetto di IoT collaborativo (C-IoT) [13] nel quale esseri umani, imprese ed altre entità collaborano aumentando considerevolmente la quantità di dati raccolti. Si tratta di una soluzione ai problemi a cui può andare incontro un singolo soggetto che può trovare difficoltà a reperire i dati di cui necessita in maniera autonoma, ad esempio in caso di costi troppo alti per procurarsi i sensori.
In questa categoria è presente il concetto di Crowdsensing.

1.2 Crowdsensing

Con Crowdsensing si intende una raccolta di dati che avviene con l'intervento di utenti tramite i dispositivi personali o a loro distribuiti per studiare certi fenomeni.
Con Mobile Crowdsensing (MCS) [2] gli utenti fanno uso di dispositivi mobili (smartphone, tablet, weareable ecc.) e relativi sensori per generare dati. Il termine venne introdotto da Raghu Ganti, Fan Ye, and Hui Lei nel 2011 [7] facendo un'analisi completa del concetto e delle potenzialità offerte dall'uso degli smartphone in questo contesto. La tipica raccolta dati per mezzo di sensori fissi è sostituita dai contenuti generati dai cittadini tramite applicazioni installate sui dispositivi.
Attualmente l'interesse a monitorare fenomeni ambientali è alto, consentendo di prevedere eventi, e più in generale offrire servizi dedicati e migliori ai cittadini. Però se il territorio da analizzare è grande, basti pensare ad una città, è chiaro come il costo per avere un numero sufficiente di dispositivi e la loro collocazione possa essere elevato. Si può sostenere che sia proprio l'enorme crescita e diffusione degli smartphone e dei dispositivi indossabili con il loro sempre più crescente insieme di sensori integrati ad aver contribuito al successo di questo paradigma. Grazie alla presenza di giroscopio, accelerometro, microfono, GPS e fotocamera è stato possibile sviluppare applicazioni per diversi casi di studio. Ad esempio, con il sensore di luminosità si può misurare la luce dell'ambiente e con il microfono si riconosce il rumore. Il GPS permette la localizzazione del dispositivo con cui si può valutare lo spostamento e soppesare la presenza di persone in un'area. La bussola ed il giroscopio determinano la direzione e l'orientamento, integrabili anche con le misurazioni del GPS. Per capire se c'è movimento potrebbe essere sufficiente anche l'accelerometro. Con questi ultimi sensori ad esempio, è possibile distinguere con quale mezzo di trasporto gli utenti si stanno muovendo. Sebbene possa risultare strano anche la fotocamera può essere considerata un sensore, infatti con le foto e/o video e le tecniche di *machine learning* quali *detection* e *recognition* pos-

sono essere analizzate varie caratteristiche.
Sono inoltre equipaggiati con molteplici interfacce di comunicazione wireless, come Bluetooth e ZigBee che forniscono comunicazioni a corto raggio, WiFi per medio raggio, 3G/4G/GSM invece a lungo raggio, ed altre. Queste tecnologie permettono anche di collegare sensori esterni ai dispositivi, ad esempio per connettere un sensore per la misurazione della qualità dell'aria ad uno smartphone che tipicamente ne è sprovvisto grazie al Bluetooth. Tutto ciò consente, con l'aiuto delle persone, di ottenere più informazioni utili a decisamente un minor costo. Al crescere della presenza degli utenti, così come la loro mobilità, corrisponderà una maggiore copertura dell'area o territorio in esame. MCS è così una soluzione ideale per costruire città intelligenti, cercando di migliorare la vita quotidiana dei cittadini.
Basandosi sul tipo di coinvolgimento degli utenti, il crowdsensing può essere suddiviso in due categorie.

- *Participatory crowdsensing*, dove gli utenti volontariamente partecipano a fornire informazioni, per mezzo di applicazioni.

- *Opportunistic crowdsensing*, dove gli utenti sono quasi inconsapevoli, il loro intervento è praticamente nullo, quindi i dati vengono rilevati, raccolti e condivisi automaticamente. Si può sostenere che il loro unico compito sia di installare l'applicazione.

Un'ulteriore suddivisione si può applicare basandosi sulla tipologia di fenomeno rilevato. Sanitario, misura la salute dei soggetti (ad esempio, la temperatura del corpo). Infrastrutturale, misura le infrastrutture pubbliche (ad esempio, le condizioni stradali, traffico stradale grazie al GPS). Sociale, misura la vita sociale degli individui (ad esempio, dati di luoghi visitati). Ambientale, misura l'ambiente naturale (ad esempio, inquinamento, temperatura, qualità dell'aria, umidità). E se ne possono elencare altri.
La raccolta dei dati avviene seguendo i passi di collezione, dove le informazioni sono recuperate dai sensori dei dispositivi, salvataggio, dove le

informazioni sono inviate e condivise per la loro elaborazione, ed infine condivisione, dove i dati nati dal punto precedente sono a disposizione dell'utilizzatore.
Questo approccio porta sicuramente molti vantaggi, ma ci sono alcuni aspetti da tenere in conto, sia da parte degli sviluppatori, sia da parte degli utenti. Il numero di partecipanti di un'applicazione MCS può variare secondo diversi fattori, come l'incentivo offerto alle persone ed il loro interesse a raggiungere l'obiettivo comune, e questo ha un effetto sulla quantità di dati generati, i quali saranno *sparsi* o *densi* [16]. Nel primo caso i dati raccolti sono inferiori alla richiesta, in quanto le aree geografiche non sono totalmente coperte o ci sono periodi di tempo senza la loro produzione. Al contrario, nel secondo caso ne è generata una grande quantità. A tal proposito è chiaro come la loro distribuzione sia importante, nel senso di copertura geografica riuscendo ad ottenerli nelle zone desiderate, ma anche come numero di utenti per non avere riferimenti solo da un piccolo gruppo di essi.
I metodi di incentivo possono essere di varia natura, dal classico e forse più attrattivo incentivo monetario, a quello che trasforma i compiti in gioco, oppure metodi che forniscono servizi agli utenti in cambio dei loro dati.
Molte applicazioni sono state sviluppate che fanno uso di questa opportunità. Ad esempio, Nericell [12] monitora le condizioni del traffico. HazeWatch [20] e Third-Eye [9] esaminano l'inquinamento dell'aria.
Esistono anche *framework* di programmazione utili agli sviluppatori aiutandoli a dedicarsi esclusivamente alla creazione delle loro applicazioni senza dover implementare funzionalità comuni in questo ambito. Medusa [18], uno dei più famosi, è *multi-purpose* e fornisce un'astrazione ad alto livello per il *mobile sensing*. Un altro conosciuto è Hive [17], il quale permette diverse operazioni come la gestione dei dati, gestione dei partecipanti e analisi server o mobile.

1.2.1 Risparmio energetico

Per gli utenti è importante anche considerare il consumo energetico richiesto, quindi diminuzione della carica della batteria, per effettuare le letture dai sensori ed il loro conseguente invio.
Ormai non dovrebbe essere più un problema visti i piani telefonici offerti oggi, ma è giusto menzionare anche il consumo della connessione dati se non c'è presenza di rete WiFi.
Per contenere queste problematiche sono state studiate diverse tecniche, le quali ad esempio limitano le connessioni al server, o regolano la quantità di dati trasmessi dai partecipanti. Il controllo di questo flusso assicura una diminuzione dell'energia richiesta ai dispositivi evitando loro il compito dell'invio dei dati quando non sono necessari, inoltre questa tecnica limita la ridondanza dei dati soddisfacendo comunque la richiesta. Esistono diversi algoritmi di ottimizzazione. Tra le operazioni principali che permettono miglioramenti si trova l'invio dati solo in caso di cambiamento, l'utilizzo di connessioni già presenti, la scelta di solo le informazioni importanti ecc. Queste tecniche possono essere gestite in locale, cioè sui singoli dispositivi mobili, oppure in generale, modificando il comportamento dell'intera folla. Di seguito ne sono illustrati alcuni esempi.

- **Compressive sensing**: è basato sulla correlazione spazio-temporale dei dati in un'area, dove diventa meno importante osservare l'intero ambiente. L'idea è selezionare solo alcune zone da cui ricavare informazioni e dedurle nelle altre con i dati a disposizione. In questo modo si riduce la quantità di dati prodotta e si evita la loro densità. Un esempio semplice ma efficace riguarda la rilevazione della temperatura [10], in quanto è normale presupporre che fra zone non particolarmente distanti e con piccole differenze orarie non ci sia un cambiamento importante.

- **Piggyback crowdsensing**: introdotto in [8], il risparmio di energia è dovuto alla mancata attivazione dei sensori perché già svegli in

seguito ad azioni come telefonate o uso di applicazioni. Vale appunto anche per la trasmissione dei dati che se eseguita in maniera intermittente fa consumare più energia. Il risultato di tale risparmio si traduce in maggior partecipazione, che porta ad una migliore copertura evitando la generazione di dati sparsi.

- **Edge deduplication**: dal momento che uno dei maggiori punti critici per il consumo di energia riguarda la trasmissione dati, questa tecnica ne riduce la quantità da inviare. Per fare ciò, vengono elaborati a livello client evitando nel contempo ridondanza e densità degli stessi.

- **Distributed heuristics**: viene fatto uso di euristiche per stimare valori di cui non si è a conoscenza. In [19] serve per stimare le traiettorie degli utenti secondo certe ipotesi. Ricade in questa categoria anche il modello basato su "indice di soddisfazione", presente nel simulatore analizzato in questa tesi, che indica quanto un server è soddisfatto dei dati ricevuti, ovvero la loro quantità rispetto a quella desiderata, ed inviata tale informazione agli utenti permette di regolare la loro frequenza di trasmissione, ottenendo un livello di densità adeguato. Il simulatore propone diverse versioni di questa tipologia di algoritmo.

- **Optimization-based**: vengono considerate delle metriche (copertura, consumo di energia ecc.) su cui costruire algoritmi di ottimizzazione selezionando alcuni vincoli come input.

- **Context and logical dependencies**: si cerca di ottenere una copertura accettabile selezionando solo una porzione minima di utenti idonei per svolgere una determinata attività.

1.2.2 Privacy

Un altro aspetto da segnalare riguarda una tematica molto importante e di cui oggi si sente parlare spesso: *privacy*. I dati raccolti infatti, è possibile

che contengano informazioni personali consentendo il tracciamento delle attività e abitudini dell'utente, ad esempio per mezzo della localizzazione GPS e dell'orario di invio dei dati. Non solo, nel caso l'applicazione faccia uso del microfono del dispositivo, ad esempio per monitorare il rumore, potrebbe registrare conversazioni private. Con opportunistic crowdsensing è un problema più rilevante poiché, come descritto sopra, l'utente è inconsapevole di quando e quali dati invia. Al contrario, con participatory crowdsensing è meno accentuato perché è l'utente a decidere quali azioni intraprendere. Gli sviluppatori delle applicazioni possono adottare diversi metodi per contrastare questo spiacevole effetto indesiderato. Ad esempio, in [1] vengono rese anonime le traiettorie, in [4] adottano una crittografia per nascondere i dati sensibili, oppure in [3] rimuovono le informazioni che rendono identificabile un utente.

1.2.3 Coverage

Per studiare certi fenomeni i dati sono estremamente importanti, in particolare le informazioni che si riescono a ricavare, quanto queste siano utili ed aggiornate. Un aspetto correlato è dato dalla *copertura*, ovvero da quanta parte del territorio studiato vengono generati i dati. Come descritto in precedenza, si possono incontrare due tipologie di scenari che differiscono per la quantità di dati, i quali sono *sparsi* (i dati raccolti sono inferiori a quelli richiesti) o *densi* (dati raccolti in abbondanza). Sono state illustrate tecniche per ridurne la trasmissione, ad esempio per evitarne la ridondanza e diminuire i costi, e gestirne la copertura stimando le informazioni di alcune aree. Questo indica che esistono studi in cui non è strettamente necessario avere un'ampia copertura, che può anche significare che i punti di osservazione possono essere distanti a livello spaziale e/o temporale. In molti altri, al contrario, la copertura totale del territorio è fondamentale per studiare fenomeni con maggior precisione. Qui nasce il problema di come reclutare le persone e incoraggiarle a recarsi nelle zone di cui si necessitano ulteriori letture, specialmente se lontane. Tipicamente avviene per mezzo

di incentivi.
In questa tesi è stata implementata la possibilità di far cambiare destinazione ed eventualmente anche traiettoria agli utenti simulati, opzioni non disponibili nella precedente versione. In un dato momento il server comunica a loro quali sono le zone prive di dati ed essi decidono se recarvisi. Questo nuovo comportamento risolve in parte il caso di generazione di dati sparsi aumentando la copertura del territorio.

1.3 Simulatori

Nonostante tutto questo sembri la soluzione ideale per studiare qualsiasi fenomeno, a volte è difficilmente applicabile. I motivi principali risiedono nel numero di partecipanti necessari per ottenere le informazioni desiderate e nel tempo. Sebbene esistano già delle applicazioni che ne sfruttano le potenzialità, ad esempio quelle per la rilevazione del traffico, a volte chi deve studiare certi fenomeni non può attendere l'intero processo di reclutamento (non facile in caso di molti cittadini) e raccolta dei dati di un certo periodo, tenendo anche presente la crescita dei costi all'aumentare della dimensione delle parti coinvolte. Inoltre la differenza tra dati "reali" e fittizi può non essere così importante. In aiuto a ciò vengono sviluppati dei simulatori, atti a valutare le prestazioni dei sistemi MCS. Grazie anche all'hardware disponibile oggi consentono di effettuare simulazioni con parametri personali (numero di utenti, il tempo di simulazione ecc.) in brevissimo tempo.
Ovviamente, MCS è un termine generale, ciò significa che per studi specifici esistono simulatori adatti. Per esempio, Network Simulator 3 (NS-3) [21] è ottimo per studi riguardanti la rete e dei suoi protocolli, può scendere ampiamente nei dettagli ed analizzarne qualsiasi aspetto, pacchetti compresi. Purtroppo è anche ciò che lo penalizza, infatti in scenari di simulazione con migliaia di utenti che agiscono per divero tempo, l'analisi diventa decisamente complessa.

CupCarbon [11] è un simulatore per wireless sensor network (WSN) nel quale, grazie alle mappe ottenute da OpenStreetMap (OSM)[1], può simulare sensori e postazioni in un ambiente urbano reale. I sensori possono essere resi mobili e fare seguire loro percorsi dedicati lungo le strade, caratteristica che lo rende sfruttabile anche per simulazioni MSC. L'utilizzatore può impostare manualmente le posizioni dei sensori, ma come prima, in scenari di una certa dimensione non è un'opzione praticabile, mitigata però dal suo limite delle dimensioni degli stessi che non lo rende ampiamente scalabile.
È risaputo che trovare parcheggio in città molto spesso è un compito arduo. Per analizzarne le dinamiche, in [5], viene presentato un simulatore sviluppato per l'analisi delle prestazioni di applicazioni MCS in questo scenario. Anche se prettamente limitato a questo caso di studio, può essere parzialmente generalizzato considerando i conducenti, che viaggiano tra un parcheggio e l'altro, come sensori che attivano eventi di parcheggio.
CrowdSenSim [6] è un simulatore per MCS decisamente più scalabile rispetto ai precedenti, sul quale si concentra questa tesi. Può infatti simulare un numero elevato di utenti (centinaia di migliaia). Questi posseggono smartphone predisposti con tre sensori per raccogliere informazioni dell'ambiente, il quale è rappresentato da una qualsiasi città del pianeta, una soluzione fornita grazie all'uso della libreria Python OSMnx[2] che ne recupera il grafo. Il suo studio principale riguarda il consumo energetico, analizzandone il comportamento per mezzo di diversi algoritmi che cercano di ridurlo. Grazie all'uso di file di configurazione si possono impostare i parametri di simulazione, ad esempio il numero di utenti ed il loro tempo di partecipazione. Al termine viene offerta una pagina web che raccoglie una serie di statistiche riguardanti i dati inviati, come la loro quantità e l'energia richiesta dai sensori per tale azione, una mappa del percorso del primo utente ed una heatmap con la distribuzione degli utenti nel territorio

[1] https://www.openstreetmap.org/
[2] https://geoffboeing.com/2016/11/osmnx-python-street-networks/

simulato.

1.4 Contributi

Il simulatore oggetto di studio in questa tesi nasce, nella sua prima versione, presso l'Università del Lussemburgo.
Successivamente, grazie alla tesi di Laurea di uno studente di questa facoltà è stato quasi completamente riscritto per migliorarne l'implementazione, e sono state aggiunte nuove funzionalità, in particolare algoritmi di simulazione, spiegati nel capitolo successivo.
I contributi principali apportati con questa proposta riguardano:

- facoltà di cambio di destinazione e traiettoria da parte degli utenti simulati. Ovviamente è disponibile per l'intero simulatore, quindi utilizzabile anche in prossimi casi di studio. È stato creato un modulo Python, responsabile della creazione dei nuovi eventi degli utenti che modificano il percorso originale, ed una nuova libreria in C++ che si occupa di farne l'interrogazione e l'inserimento ordinato degli eventi creati nell'apposita lista;

- per analizzare questa nuova funzionalità si è adottato un particolare caso di studio, cioè si è modificato l'algoritmo AO-JFS in una versione definita JFS-IBRIDO che la integra, trasformandolo in un algoritmo di raccolta dati che usa il cambio di traiettoria secondo una probabilità (distanza o probabilità) aggiungendo la modalità *participatory crowdsensing*;

- il punto precedente si lega con il nuovo modulo server che periodicamente controlla da quanto tempo le zone della città (rappresentate dai quadrati MGRS da 1 km) sono prive di letture dati, informazione usata dagli utenti per decidere se accettare la sua richiesta di deviazione;

- la validazione del lavoro è effettuata per mezzo di test basati su tre città (Bologna, Lussemburgo e Melbourne) con i tre algoritmi citati nel punto precedente e 10000 utenti simulati. I risultati sono facilmente visualizzabili e interpretabili grazie alle *heatmap* generate dal file di output prodotto dalla simulazione con script scritti in Python;
- implementazione della classe per la valutazione delle prestazioni della nuova funzionalità, nelle due modalità di esercizio, adottando diversi valori di probabilità, e producendo il grafico per l'analisi.

In aggiunta ai punti appena trattati, le attività minori svolte sono:

- possibilità di salvare la mappa delle città per effettuare simulazioni offline. In caso di nuova simulazione in una città della quale non è stata salvata la mappa precedentemente, il simulatore chiederà all'utente come deve procedere in merito;
- correzione di piccoli bug;

La tesi è strutturata nel seguente modo: nel capitolo 2 viene spiegato nel dettaglio il simulatore allo stato corrente, cioè alla sua seconda versione, quindi la sua architettura, le librerie usate, gli algoritmi implementati ed i suoi limiti. Nel capitolo 3 trovano spazio la progettazione ed implementazione delle nuove *feature* sviluppate. Il capitolo 4 è inerente alla validazione dell'applicazione sviluppata, analizzandone le *performance* ed i risultati dei test effettuate, e chiarendo le scelte adottate. Infine, il capitolo 5 è dedicato alle conclusioni.

Capitolo 2

CrowdSenSim 2.0

CrowdSenSim è un simulatore per analizzare sistemi MCS in ambienti urbani reali, i quali possono essere anche di notevoli dimensioni. È incentrato sullo studio dei dati generati e dell'energia richiesta per ottenerli, simulando spostamenti reali dei partecipanti.
CrowdSenSim 2.0 [15] è la naturale prosecuzione del lavoro svolto nella versione precedente, correggendolo e aggiungendo nuove caratteristiche. In particolare:

- aggiunta di:
  - simulazioni stateful, ovvero gli eventi vengono ordinati temporalmente e la simulazione si basa su questo;
  - algoritmi PDA;
  - parallelismo, cioè è possibile eseguire in contemporanea più algoritmi, uguali o diversi;
  - codifica spaziale MGRS;
  - nuovo algoritmo per la generazione delle traiettorie degli utenti;
- ottimizzazione del codice, conseguenza della sua quasi totale riscrittura, migliorando così il tempo di esecuzione e la memoria impiegata della simulazione.

La simulazione si svolge in questo modo. Dopo aver specificato la città da esaminare ed il numero di utenti da inserire, per ognuno di essi viene generato un percorso da seguire. All'interno del tragitto vengono creati degli eventi ogni 10 secondi di percorso, cioè la posizione o momento in cui l'utente effettua le letture dei dati e forse li invia. Questa scelta è determinata dall'algoritmo simulato, ad esempio nel caso di una qualche versione PDA dipenderà dal valore del *SI*, mentre con DDF la decisione si baserà sulla corrente utilizzata dal dispositivo.
A questo punto viene creata la lista ordinata degli eventi generati, definite ed inizializzate le variabili interne per gli smartphone, le antenne e gli algoritmi da elaborare, e la simulazione vera e propria ha inizio.
Conclusa la simulazione, viene generata una pagina web che contiene le informazioni dei risultati e delle statistiche degli algoritmi studiati sotto forma di grafici e mappe.
In figura 2.1 è illustrata l'architettura modulare del simulatore, la quale mostra i moduli da cui è composto e le loro sequenze nello svolgimento della simulazione, i quali vengono chiariti all'interno di questo capitolo.

2.1 Architettura

Si possono dunque distinguere due moduli separati, uno per la generazione degli eventi (Event Generator) ed il simulatore vero e proprio.

2.1.1 Event Generator

Questo modulo è scritto interamente in linguaggio Python. Per svolgere il suo compito necessita di conoscere il nome della città da simulare e alcuni parametri di configurazione presenti nel file `Event_Generation_config.txt`.

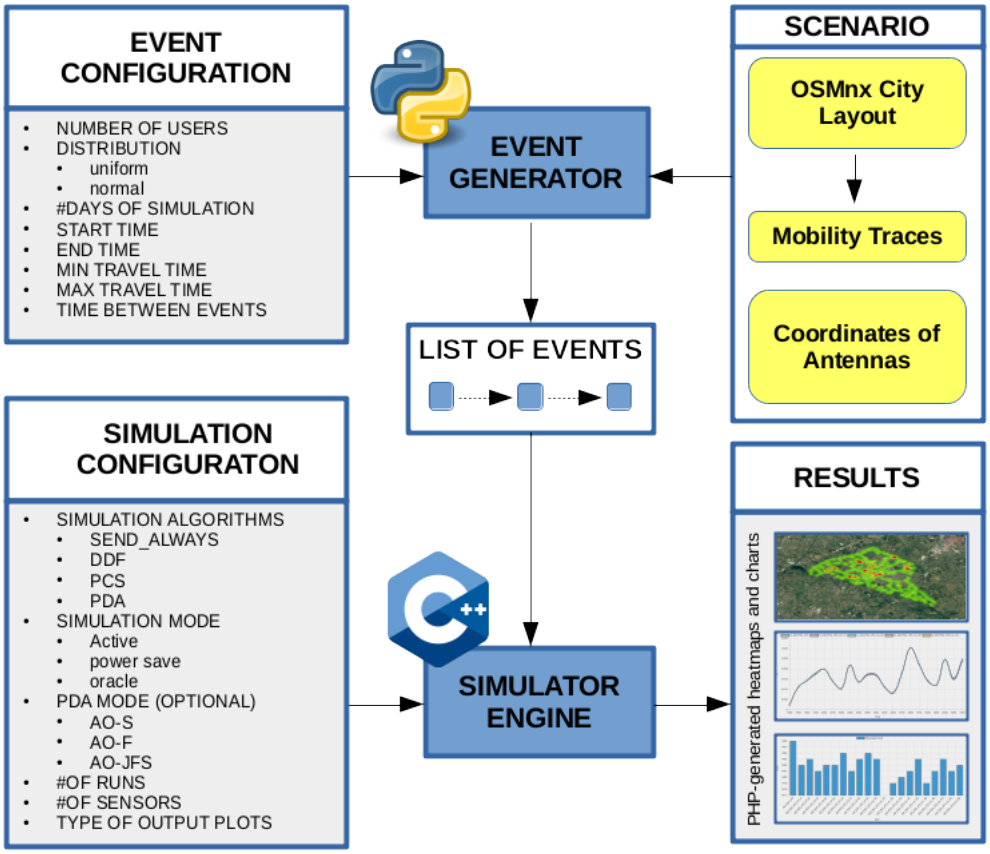

Figura 2.1: Architettura modulare CrowdSenSim 2.0

Creazione punti di partenza

Dopo aver scelto la città in cui eseguire la simulazione, la libreria OSMnx restituisce un grafo che presenta le strade e i relativi nodi. In figura 2.2 è rappresentata una porzione della città di Bologna semplificata, cioè con un minor numero di nodi. Il simulatore fa uso della mappa normale. Per ogni coppia di questi nodi vengono creati ulteriori nodi che vengono aggiunti all'interno della retta che congiunge la coppia. Questi nuovi nodi sono potenziali punti di partenza degli utenti, e hanno una distanza di 3 metri fra di essi. Il loro posizionamento avviene tramite la funzione inversa di

Vincenty con cui si ottiene l'angolo α di Azimut fra due punti, cioè l'angolo presente tra la retta che collega i due punti della coppia e il punto cardinale nord.
I nuovi punti contengono le informazioni riguardanti la coppia di nodi da cui sono stati generati, ovvero il loro ID e la distanza che li separa da essi, calcolata per mezzo della funzione presente nella libreria OSMnx **great_circle_vec**, la quale necessita delle coordinate latitudine e longitudine dei due nodi "genitori".

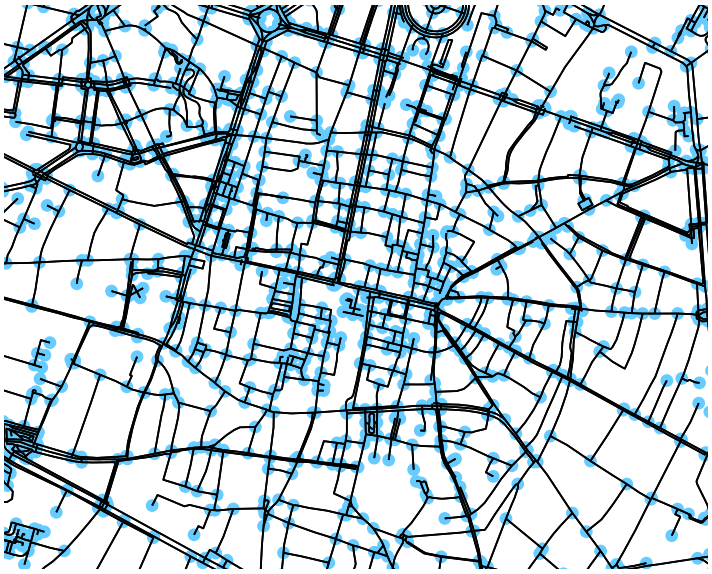

Figura 2.2: Grafo generato da OSMnx della città di Bologna con nodi semplificati

Creazione percorsi

Dato il grafo con tutti i nuovi nodi generati ed inseriti, le partenze dei partecipanti vengono definite scegliendo nodi casuali. Sono scelti casualmente all'interno degli intervalli specificati nel file di configurazione anche ora e giorno di partenza, così come durata e velocità di camminata. Proprio tramite questi ultimi due valori si può calcolare la distanza che deve percorrere l'utente. È però stato inserito un controllo per assicurarsi che questa sia accettabile, cioè viene incrementata di una costante, il cui valore identifica la lunghezza dell'arco del grafo più lungo.
La generazione del percorso da seguire avviene utilizzando la funzione della libreria NetworkX **single_source_dijkstra**. Conoscendo la distanza da percorrere ed il nodo di partenza trova tutti i possibili percorsi di lunghezza inferiore o uguale alla distanza. Come prima, la scelta di quale l'utente deve intraprendere è casuale. Dal nome si può intuire che tale funzione applica l'algoritmo di Dijkstra, il quale calcola i cammini minimi in un grafo, effettuando una visita in ampiezza per trovare i percorsi.
Dopo aver determinato quale percorso l'utente deve seguire avviene il processo di creazione degli eventi, cioè sono i punti dove avvengono le letture dei dati dai sensori dei dispositivi. Queste sono separate da un intervallo di tempo di 10 secondi (di default), che può essere impostato nel file di configurazione **Event_Generation_config.txt**. La procedura per la loro creazione è analoga a quella vista in precedenza per la creazione dei punti, cioè per ogni coppia di nodi collegati da un arco lungo il percorso si recuperano le loro coordinate latitudine e longitudine, e si calcola l'angolo α di Azimut tra il punto cardinale nord e la posizione del secondo punto della coppia. E ancora, per trovare la distanza dove inserire il punto di lettura, si moltiplica la velocità di camminata dell'utente per il tempo dell'intervallo. Con queste informazioni, cioè distanza tra il nodo di partenza ed il nuovo da inserire, angolo α di Azimut e nodo di partenza, la funzione **VincentyDistance** della libreria Python *geopy* calcola le coordinate latitudine e longitudine per gli eventi.

Il simulatore utilizza MGRS, perciò in questa fase converte le coordinate appena viste in stringhe MGRS, un compito reso semplice dalla funzione **toMGRS** presente nella libreria Python *MGRS* alla quale è possibile specificare la precisione desiderata, qui di default è uguale a 1 metro.
La creazione del percorso tiene ovviamente conto dei parametri impostati, perciò il tragitto di ogni utente non dovrà mai superare il proprio tempo di camminata, né superare il tempo previsto per lo svolgimento della simulazione.

File degli eventi

Al termine delle due fasi precedenti viene generato un file contenente le informazioni degli eventi originati, `UserMovementsList_0.txt` (lo 0 indica il giorno in cui sono eseguite le letture, perciò può variare in base alle impostazioni scelte dall'utilizzatore del simulatore). Ogni riga rappresenta un evento di un utente, e i suoi campi sono:

- **ID-User**: è il numero identificativo dell'utente;
- **MGRS**: stringa MGRS che rappresenta la zona dove avviene la lettura, salvata con la precisione di 1 metro, quella massima;
- **Day, Hour, Minute, Second**: numero del giorno, l'ora, i minuti, e secondi in cui si dovrebbe verificare l'evento di lettura;
- **Lat, Long, Alt**: chiaramente latitudine, longitudine ed altezza dell'evento;
- **SecUsed**: numero di secondi trascorsi dall'inizio del percorso dell'utente considerato;
- **Bearing**: valore dell'angolo α di Azimut tra il punto cardinale nord e il nodo di arrivo della strada nella quale si trova l'utente;
- **Distance**: distanza percorsa dall'utente al momento corrente.

Questi ultimi tre valori sono calcolati durante la creazione del percorso, ciò significa che non sono presenti nel primo evento di ogni utente.

Salvataggio

Quando si genera una nuova simulazione all'utilizzatore viene chiesto se desidera salvarla. In caso affermativo, nel percorso `./Inputs/saved/` viene creata una nuova cartella, ad esempio `0list`. Lo 0 è un identificativo per le simulazioni, quindi sarà incrementato ad ogni salvataggio. Al suo interno finiscono il file `UserMovementsListEvents_0.txt` descritto precedentemente, il quale contiene gli eventi creati, `CoordinatesAntennas.txt` generato tra le due fasi di creazione di cui sopra, un elenco di identificativi e coordinate (latitudine e longitudine) delle antenne simulate, `Setup.txt` che è un piccolo file con solo i valori di giorno e numero utenti che fanno parte della simulazione, ed infine `route_usr_1day_0.html` che contiene il percorso del primo utente raffigurato all'interno di una mappa creata con la libreria *Folium*.

2.1.2 Simulator engine

Questo modulo è scritto interamente in C++.
Dopo aver raccolto i parametri impostati nei file di configurazione, inizia a creare gli smartphone dei partecipanti da simulare. Ad ognuno viene assegnata una capacità massima della batteria ed una percentuale di carica, così da simulare dispositivi differenti.
Il passo successivo riguarda la lettura degli eventi, recuperati dal file generato dal primo modulo `UserMovementsListEvents_x` (`x` è il giorno delle letture) e salvati in una lista. Ogni evento è composto dai valori di id_utente che lo effettua, id_evento, stringa MGRS, coordinate latitudine e longitudine ed orario in cui avviene.
A differenza del simulatore passato, la versione 2.0 ha introdotto l'ordinamento temporale degli eventi, eseguito in questa fase.

Per stabilire quali sono gli algoritmi da simulare viene analizzato il relativo file di configurazione da cui si ricava anche il numero di simulazioni da effettuare. Il file viene, come si dice in gergo, "parsato", costruendo così gli oggetti specifici per ognuno di essi.
Per salvare i dati generati dai dispositivi dei partecipanti viene creata una matrice a 3 dimensioni composta dai valori dei dati dei sensori, in una certa area, ad un determinato istante di tempo. Ne viene creata una per ogni algoritmo. Tra questi, quelli che devono calcolare il valore del *SI* la usano per poter conoscere i dati inviati negli istanti di tempo precedenti. Non solo, infatti viene sfruttata la sua conoscenza anche durante la creazione dei grafici tracciando l'evoluzione della simulazione.
La simulazione degli eventi avviene analizzandone uno alla volta e applicando tutti gli algoritmi specificati nel relativo file di configurazione.
Per questi compiti sono state create diverse classi di linguaggio. In particolare, le classi dei vari algoritmi ereditano dalla **DataCollectionFramework** che implementa le funzioni per il calcolo dei consumi energetici dei sensori, l'invio dei dati al server ed il calcolo della probabilità di invio. Le classi, e quindi i relativi algoritmi implementati, che ereditano da questa classe sono **PiggybackCrowdSensing**, **DeterministicDistributedAlgorithm**, e **ProbabilisticDistributedAlgorithm**. A sua volta, da quest'ultima, eredita la classe **PDA_LookAhead**, dalla quale ereditano tutte le classi degli algoritmi PDA.
Completata la simulazione vengono generati i grafici potendo ricavare le informazioni opportune dalle strutture che hanno salvato i dati. I grafici sono:

- dati inviati per ogni messaggio per ogni algoritmo;
- dati inviati da ogni algoritmo;
- messaggi inviati da ogni algoritmo;
- valore del *SI* per ogni algoritmo;

- invio dei dati sotto certe condizioni per ogni algoritmo;
- andamento del valore del *SI* al trascorrere del tempo;
- distribuzione temporale degli utenti.

I dati della simulazione che rappresentano la quantità di utenti attivi in ogni intervallo di tempo, la variazione del *SI*, il numero di letture effettuate da ogni dispositivo e ad ogni intervallo di tempo sono esportati in un file CSV, che uno script PHP legge ed elabora grazie a codice scritto in Javascript sfruttando la libreria *Chart.js*.
Per visualizzare questi grafici e le statistiche dell'intera simulazione viene automaticamente generata una pagina web che le contiene.

2.2 File configurazione

Nel simulatore è contenuta la cartella di nome `config`, al cui interno si trovano i file per la configurazione dello stesso. Sono 3 e permettono una totale personalizzazione di tutti gli aspetti che riguardano la simulazione da eseguire.
In particolare `Event_Generation_config.txt` consente di impostare i parametri relativi agli utenti, ad esempio il numero dei partecipanti. In `Simulation_config.txt` sono presenti i parametri per la configurazione della simulazione, ad esempio la lista di algoritmi da simulare. Infine, in `PDA_config.txt` si possono definire i dettagli inerenti all'algoritmo PDA.
Di seguito vengono presentati nel dettaglio, elencando i parametri gestiti al loro interno con annessa descrizione delle funzioni.
In `Event_Generation_config.txt` ci sono:

- **Numero utenti**: ossia i partecipanti alla simulazione;
- **Giorni di simulazione**: ossia il numero dei giorni di durata della simulazione;

- **Ora e minuto di inizio simulazione**: orario dell'inizio della simulazione, sarà sempre lo stesso per ogni giorno simulato;
- **Ora e minuto di fine simulazione**: orario della fine della simulazione, sarà sempre lo stesso per ogni giorno simulato;
- **Tempo minimo e massimo di percorrenza**: specifica il tempo utilizzato per muoversi all'interno della simulazione da ogni utente;
- **Distribuzione utenti**: la scelta è tra distribuzione uniforme e distribuzione normale;
- **Tempo tra eventi**: secondi che intercorrono tra gli eventi.

Per la configurazione della simulazione i valori da impostare sono contenuti all'interno del file `Simulation_config.txt`:

- **Tipologia di grafico**: scelta della tipologia di grafico che si visualizza al termine della simulazione. Ci sono 7 possibili scelte;
- **Somma dati sensori**: se si imposta il valore 1 all'interno del grafico verranno sommati i dati dei sensori dei dispositivi mobili;
- **Tipologia di Antenna**: rappresenta la tecnologia utilizzata dai dispositivi mobili che permette la trasmissione dei dati ricavati dai sensori per ottenere le informazioni riguardanti lo svolgimento dell'attività;
- **Raggio heatmap**: nella pagina web finale è visualizzata la heatmap che analizza la distribuzione delle letture nel territorio simulato. Questo valore rappresenta la dimensione in metri del raggio del cerchio raffigurato nella heatmap;
- **Numero simulazioni**: rappresenta il numero di simulazioni che verranno eseguite;
- **Numero di sensori**: il numero di sensori assegnati ad ogni smartphone;

- **Lista algoritmi da utilizzare**: è la lista di algoritmi da utilizzare nella simulazione.

Si precisa che:

- il valore dei giorni, non deve mai superare il numero 7;
- il valore delle ore di inizio e fine simulazione deve essere compreso tra 0 e 23;
- i minuti di simulazione devono essere compresi tra 0 e 59.

Nel file `PDA_config.txt` si possono impostare i parametri per la configurazione dell'algoritmo PDA:

- **Dimensione finestra**: per l'algoritmo PDA si specifica il numero di time slot a rappresentazione della dimensione della finestra;
- **Dimensione time slot**: la dimensione del time slot in secondi;
- **Limite inferiore e superiore del SI**: rappresenta i valori minimi e massimi nei quali collocare l'indice di soddisfazione (*SI*);
- **Dati richiesti**: sono elencati i sensori, con il loro nome ed i dati richiesti da ciascuno. Significa che se si aggiungono righe ogni dispositivo mobile avrà più sensori a disposizione.

2.3 Librerie

Il primo modulo, scritto interamente in Python, per svolgere i suoi compiti fa uso di alcune librerie, presentate di seguito.

2.3.1 NetworkX

La libreria NetworkX[1] permette di gestire i grafi restituiti dalla libreria OSMnx grazie ad una varietà di funzioni. Nel simulatore in particolare

[1] https://networkx.github.io/

viene utilizzata la funzione di libreria **single_source_dijkstra(graph, origin, cutoff=distance)** che implementa l'algoritmo di Dijkstra per la ricerca di cammini minimi all'interno di un grafo pesato. Dato il grafo, un nodo di partenza ed una distanza, è in grado di calcolare tutti i cammini di lunghezza minore o uguale alla distanza data che partono dal nodo specificato.

2.3.2 OSMnx

La libreria OSMnx[2] consente di scaricare mappe di aree geografiche. Si può scegliere il territorio desiderato, ad esempio specificando il nome di una città, oppure solo un'area partendo da un indirizzo, o una zona interna a certe coordinate ecc. Nel simulatore la scelta è per mezzo di una città, e viene restituito il grafo della libreria NetworkX. Contiene archi, i quali identificano le strade, e nodi, cioè gli incroci (e non solo). Ogni arco è pesato ed il suo valore determina la lunghezza della strada che rappresenta. Inoltre possiede anch'essa una varietà di funzioni di analisi ed elaborazione.
È stata usata la **graph_from_place(name city,..)** per scaricare la mappa della città scelta.
La **get_route_edge_attributes(Graph, route, attribute='length', minimize_key='length')** per ottenere una lista di attributi degli archi di un percorso, durante la creazione degli eventi.
great_circle_vec(lat1, lon1, lat2, lon2) calcola la distanza tra i due punti specificati dalle coordinate.

2.3.3 MGRS

La libreria MGRS[3] può convertire le coordinate latitudine e longitudine in stringa MGRS e viceversa. Nel simulatore esegue questa trasformazio-

[2]https://geoffboeing.com/2016/11/osmnx-python-street-networks/
[3]https://pypi.org/project/mgrs/

ne sulle coordinate dei punti di lettura degli utenti, grazie alla funzione **toMGRS(lat, lon)**. È possibile anche specificare la precisione desiderata, in questo caso quella massima, ovvero un metro quadrato.
MGRS (Military Grid Reference System) è uno standard utilizzato dai militari della NATO per localizzare punti sulla Terra per mezzo di quadrati con lati di lunghezza variabile per aumentarne o diminuirne la precisione di rilevamento, modificabile attraverso il numero di caratteri specificati nella stringa che li identifica.

2.3.4 Chart.js

La libreria Chart.js[4] è scritta in linguaggio Javascript. Il suo compito nel simulatore è creare i grafici finali. Infatti sono presenti grafici a linee per rappresentare il valore del *SI*, e istogrammi per raffigurare la quantità di dati inviati e numero di messaggi trasmessi dagli algoritmi simulati.

2.4 Algoritmi di CrowdSenSim

Il simulatore gestisce l'invio delle informazioni raccolte dagli utenti secondo alcuni algoritmi che sono esaminati di seguito.

2.4.1 Send Always

L'invio dei dati avviene ad ogni evento di lettura, sempre (da qui il nome), senza considerare letture precedenti o attuali. Permette di conoscere la quantità totale dei dati che possono essere generati nella simulazione.

2.4.2 Deterministic Distributed Framework (DDF)

È un algoritmo che si preoccupa di risparmiare l'energia e di conseguenza la banda del dispositivo. Per fare ciò, il dispositivo mantiene una

[4]https://www.chartjs.org/

cronologia delle sue passate azioni con la quale analizza l'energia necessaria per la lettura e l'invio dei dati, il proprio livello di batteria e l'ammontare di dati con cui ha già collaborato. In caso di "batteria quasi scarica" o di precedente invio dati consistente l'algoritmo non consente un ulteriore invio, garantendo l'equità tra i partecipanti.

2.4.3 Piggyback CrowdSensing (PCS)

Già introdotto nella precedente sezione, questo algoritmo ha come obiettivo la riduzione del consumo energetico richiesto per l'invio dei dati cercando di sfruttare azioni già avviate, ovvero durante una telefonata, o tramite applicazioni aperte. In particolare, controlla se il dispositivo è già connesso con il server. Si intuisce che possono verificarsi letture anche in mancanza di tale connessione, perciò il dispositivo manterrà in memoria i dati aspettando il momento giusto per l'invio. Si evince anche che questo algoritmo non è adatto in situazioni in cui sono richiesti dati in tempo reale proprio a causa dei periodi più o meno lunghi in cui possono verificarsi mancanze di connessione.

2.4.4 Probabilistic Distributed Algorithm (PDA)

Il server invia periodicamente osservazioni relative a una certa risorsa. Il suo obiettivo è di ottenere una determinata quantità di dati per essa in un certo periodo di tempo. Il server fa uso di un Indice di Soddisfazione (*SI*) per misurare l'andamento della raccolta dati, calcolata come rapporto tra osservazioni ricevute e osservazioni richieste all'interno di una finestra temporale. Lo scopo è ottenere un *SI* uguale a 1, che significa massima soddisfazione e di conseguenza i partecipanti non dovranno raccogliere ed inviare dati. Il secondo obiettivo è garantire equità dei dati inviati dagli utenti evitando così di fare affidamento solo su pochi dispositivi e lasciando inutilizzati i restanti. Questo modo di operare permette agli

utenti di non memorizzare i precedenti dati inviati.
Esistono delle varianti a questo algoritmo, presentate di seguito [14].

Asymptotically Optimal Backoff (AOB)

Il server è a conoscenza del valore del *SI* di un tempo t_i. Calcola la probabilità di inviare nello step successivo t_{i+1} come:

$$P_{i+1} = 1 - SI_i$$

Nel simulatore è stata rinominata Basic.

Asymptotic Opportunistic algorithm for SatisfactionIndex (AO-S)

A differenza dell'algoritmo precedente è stato introdotto un meccanismo *booster*, un valore b, che in caso di *SI* stabilizzato ad un livello basso consente di incrementare la probabilità di invio. Agisce anche al contrario, cioè quando *SI* è elevato riduce la probabilità. È così definita:

$$P_{i+1} = 1 - SI_i^b$$

L'incremento o diminuzione di b avviene nel seguente modo:

$$inc(b) = \begin{cases} \frac{1}{1/(b-1)} & \text{se } b < 1 \\ b + 1 & \text{altrimenti} \end{cases}$$

$$dec(b) = \begin{cases} b - 1 & \text{se } b > 1 \\ \frac{1}{1/(b+1)} & \text{altrimenti} \end{cases}$$

In particolare, periodicamente l'algoritmo calcola il nuovo valore del *SI*, e se questo supera il limite superiore applica il decremento, al contrario, se è minore del limite inferiore lo incrementa.

Asymptotic Opportunistic algorithm for Fairness (AO-F)

Con l'algoritmo precedente non è garantita l'equità del carico di trasmissioni tra i partecipanti. Viene perciò introdotto il fattore k che rappresenta il numero di intervalli di tempo consecutivi in cui il dispositivo non ha inviato dati, ed è calcolato come $k = \lfloor \log_2(j) \rfloor$ dove $j > 0$ è il numero di slot t_i trascorsi dall'ultima trasmissione. Maggiore sarà questo valore, più sarà alta la probabilità di invio, garantendo l'equità, così calcolata:

$$P_{i+1} = 1 - SI_i^{1+k}$$

Asymptotic Opportunistic for Joined Fairness and Satisfaction index(AO-JFS)

È una combinazione degli algoritmi AO-S e AO-F. La probabilità di trasmissione è data da:

$$P_{i+1} = 1 - SI_i^C$$

dove C è la combinazione tra b e k, ovvero $C = inc^k(b)$. Dopo vari esperimenti è stato calcolato il valore migliore che k può assumere, cioè:

$$k = \begin{cases} \lfloor \log_2(j) \rfloor & \text{se } b < 1 \\ j * b & \text{se b} \geq 1 \end{cases}$$

Ad ogni slot di tempo il server calcola il valore di b e lo invia ai partecipanti, i quali a loro volta, in ciascuna fascia oraria, si calcolano la propria probabilità di trasmissione basandosi sul valore b ricevuto ed il loro valore j.

PDA con finestra di lookahead

Ogni algoritmo PDA presentato può essere aggregato da questa ulteriore tecnica di risparmio energetico. L'invio dei dati avviene solo dopo aver atteso il tempo quantificato dalla dimensione della finestra di lookahead. In questo modo quelli raccolti fino a tale momento vengono riuniti in una

determinata trasmissione. Il *SI* è strettamente correlato a tale valore, infatti più è elevato e più il *SI* faticherà a raggiungere il valore ottimale in ogni finestra.

2.4.5 Ricezione SI

I partecipanti possono ottenere i valori di *SI* e di *b* secondo diverse strategie:

- **active mode**, in ciascuno slot t_i si mettono in ascolto e ricevono l'informazione dal server, di conseguenza i valori sono costantemente aggiornati;
- **power safe mode**, i valori sono ottenuti in seguito ad un loro invio come risposta, perciò a differenza del caso precedente non sono aggiornati in tempo reale, ma permette agli utenti di consumare meno energia;
- **oracle mode**, ha un consumo maggiore di energia permettendo però di mantenere il *SI* a valori ottimali poiché l'utente conosce *SI* e *b* in ogni momento.

2.5 Limiti del simulatore

È stato descritto CrowdSenSim 2.0, il quale ha portato nuove funzionalità e grossi miglioramenti rispetto alla versione precedente, presentati ad inizio capitolo. In particolare, l'aggiunta dell'ordinamento temporale rende più realistica l'intera simulazione. L'implementazione degli algoritmi PDA ha l'obiettivo di risolvere il problema della generazione di dati densi o sparsi in opportunistic crowdsensing, nel quale i partecipanti, in base a comunicazioni ricevute dal server, possono decidere se raccogliere ed inviare dati. Nel primo caso porta ad una limitazione di queste generazioni, al contrario, nel restante caso ne produce un aumento. Cerca di svolgere

al meglio il suo compito, ma se ad esempio i dati sono troppo sparsi perde in funzionalità.
Al momento attuale gli utenti seguono il percorso a loro assegnato ed eseguono i relativi eventi. Come è stato descritto, i dati fanno riferimento a determinate zone. Se in alcune di esse non vengono fatte letture a sufficienza è chiaro che la copertura totale del territorio viene meno, così come la completezza dei dati. Il server potrebbe averne bisogno.
In questa tesi viene implementata la possibilità di comunicare ai partecipanti in quale zona si desidera una maggiore affluenza. Gli utenti dunque, potranno decidere se proseguire lungo il loro percorso oppure effettuare una deviazione verso la zona richiesta. In questo modo si aggiunge il concetto di *participatory crowdsensing* simulando comportamenti degli utenti insiti nella definizione, cioè più simili a quelli che adotterebbero le persone reali in un contesto di MCS, risolvendo il problema dei dati sparsi.

Capitolo 3

Progettazione ed implementazione

In questo capitolo è illustrato il lavoro svolto in questa tesi. Viene spiegato il procedimento adottato per simulare le variazioni di traiettoria degli utenti e l'algoritmo modificato diventato il caso di studio per analizzare la nuova funzionalità. Per questa procedura avviene anche un cambio dell'architettura del simulatore. Mentre in precedenza era composto esclusivamente dai due moduli spiegati nel capitolo 2, ora se ne aggiunge un altro. Si tratta di un modulo Python, ed è il responsabile dei cambi. Tra questo ed il simulator engine ci sarà un continuo scambio di informazioni, con l'aiuto di una nuova libreria implementata che consente dialogo. In figura 3.1 è visibile la nuova architettura, e le linee rosse rappresentano le caratteristiche aggiunte.

3.1 Applicazione finale

Il modello applicativo del cambio traiettoria per studiarne l'efficacia è basato sull'algoritmo PDA AO-JFS. La nuova classe implementa lo stesso algoritmo per la raccolta dati e successivamente, sotto certe condizioni avviene il cambio di traiettoria. Se è stato scelto tra gli algoritmi da simulare,

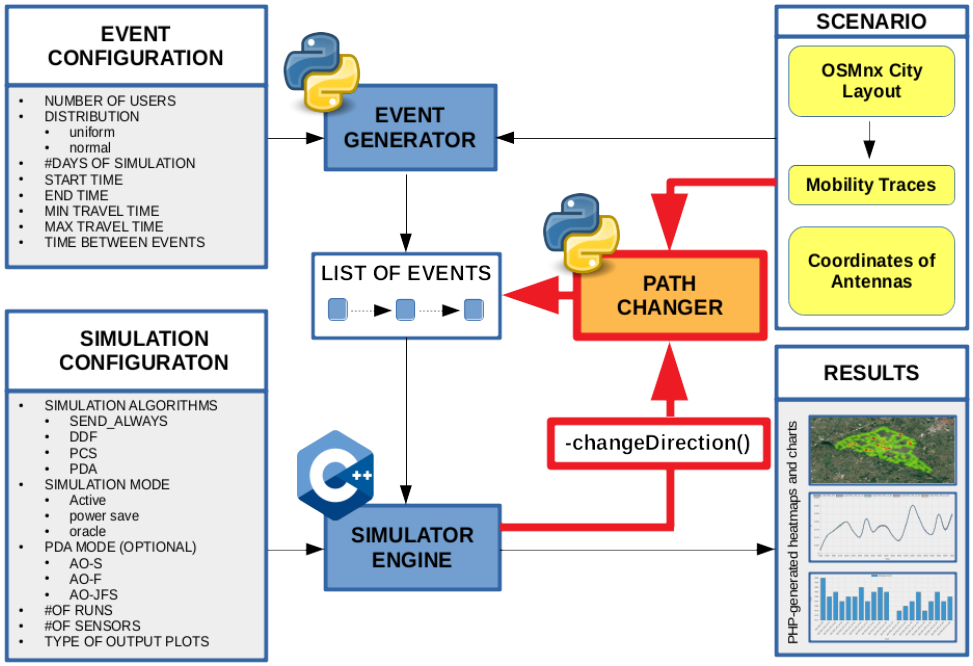

Figura 3.1: Architettura modulare nuova

ciascun evento lo eseguirà.

Il concetto che sta alla base del ragionamento è il seguente. Ogni 5 minuti il server controlla le zone MGRS del territorio. Se rileva mancanza di dati in quell'arco di tempo, invia una richiesta ai partecipanti di indirizzarsi verso tale luogo. Loro dovranno decidere se esaudire tale desiderio o meno. La scelta dipende da diversi fattori, ad esempio la distanza che intercorre tra la posizione corrente dell'utente e la zona da raggiungere, oppure dalla semplice volontà. Per capire meglio cosa si intende con l'ultimo esempio è possibile immaginare un utente ormai prossimo alla sua destinazione. In quel momento gli arriva la notizia di un'area che necessita di partecipanti per colmare l'assenza di dati. Potrebbe essere dall'altra parte della città, quindi abbastanza distante. Maggiore è la distanza e più tempo ci vuole per coprirla. Sotto queste ipotesi, l'utente può ragionevolmente declinare l'invito e proseguire sulla propria strada.

Data la lista delle coordinate MGRS dei quadrati della città simulata, vengono controllate quali non hanno avuto utenti che hanno generato dati negli ultimi 5 minuti con almeno un sensore. Viene interrogata la struttura (**mData**) che memorizza la quantità di dati inviati da un determinato sensore, in una certa zona MGRS, in un dato intervallo di tempo. Queste zone sono chiamate "critiche" e vanno a formare una propria lista. Ovviamente questa è una procedura che svolge un utente secondo il proprio tempo, quindi al passare dei suoi 5 minuti, con la funzione **calculateDataSentInRange(sensore, mgrs, *t* inizio, *t* fine)**.
La "lista critica" (nella pratica è una `map`, scelta anche per l'ordinamento automatico dei suoi elementi) contiene la distanza che separa l'utente dal centro della zona MGRS, la probabilità dell'utente di andare nel sito richiesto e le relative coordinate latitudine e longitudine.
La distanza, in metri, si calcola con la formula:

$$d = (R * acos((sin(la1) * sin(la2)) + (cos(la1) * cos(la2) * cos(lo1 - lo2)))) * 1000$$

dove $la1$ e $lo1$ sono le coordinate dell'utente, mentre $la2$ e $lo2$ sono le coordinate del centro dell'area MGRS. Devono essere trasformate in radianti. R è il raggio della terra in chilometri, il cui valore è 6371.
Per l'ordinamento della lista ed il calcolo della probabilità ci sono due modalità:

- per **distanza** (crescente), perciò saranno considerate prima le zone più vicine. La probabilità è calcolata come

  $$prob = (1 - (min(d, MAX\_DIST)/MAX\_DIST)) * userWill$$

  Più l'area è vicina, maggiore considerazione avrà da parte dell'utente, anche se poi entra in gioco la sua volontà.

- per **probabilità** (decrescente), perciò saranno considerate prima le zone con valore più alto. La probabilità è calcolata come

  $$prob = (((1 - (min(d, MAX\_DIST)/MAX\_DIST))/2) + ((min(vzone, MAX\_EMPTY)/MAX\_EMPTY)/2)) * userWill$$

Se una zona non ha misurazioni da tanto tempo, anche se è distante, l'utente potrebbe volerci andare ugualmente. *vzone* rappresenta da quanto tempo, in numero di slot da 5 minuti, una determinata zona non riceve rilevazioni.

Nelle formule sono presenti *userWill*, cioè la volontà dei vari utenti. È calcolata singolarmente e rimane fissa per tutta la simulazione, generata come valore random tra 0 e *MAX_WILL*, impostato a 0.25. Gli altri valori fissi sono *MAX_EMPTY*, uguale a 10, la quantità massima di time slot senza dati in un area da considerare, e *MAX_DIST* cioè la distanza oltre la quale gli utenti non gradiscono andare, ovvero 3 chilometri (da usare in metri). *vzone* è il vettore associato all'algoritmo correntemente simulato e recuperato dalla struttura più ampia *zoneNEmpty* che li contiene, gestita dalla classe *Server*. Per spiegare meglio questo concetto, si ricorda che è possibile eseguire una lista di algoritmi ad ogni simulazione, e hanno strutture e variabili separate. Ad ogni algoritmo è associato un indice numerico utile, in questo caso, a recuperare il vettore delle zone collegato alla propria esecuzione.
Per decidere se cambiare traiettoria, cioè fare una deviazione rispetto al percorso originale passando per l'area priva di dati identificata dalle sue coordinate, viene generato un nuovo numero casuale e lo si confronta con la probabilità calcolata in precedenza durante il controllo della "lista critica". Se questa è maggiore viene effettuata la deviazione con la chiamata alla funzione **changeDest()**. Tra i parametri, oltre alle coordinate della zona, lo smartphone ed il puntatore all'evento correnti e la lista degli eventi, sono da segnalare:

- il valore per il metodo di scelta del nodo sulla mappa date le coordinate (nodo vicino o massima precisione);
- il valore booleano se effettuare la deviazione o no, cioè solo il cambio di destinazione. In realtà, attualmente non avviene un vero cambio poiché non è così determinante in casi di studio. Le coordinate della

destinazione originale vengono salvate come variabili all'interno della classe **Smartphone** durante la fase di inizializzazione della simulazione. Quindi, in questo caso, viene comunque svolta la procedura di ricalcolo percorso ma mantenendo la medesima meta.

Implementazione

Creazione lista critica degli utenti

```
if ora_corrente_ev - controllo_prec_ev_utente >= 5min then
    for zona in lista_zone do
        for s in n_sensori do
            if calculateDataSentInRange(s, zona,
                controllo_prec, ora_corrente) == 0 then
                d := calcola_distanza(utente, zona)
                prob := calcola_prob()
                lista_critica := crea_lista_ordinata(
                    d, prob, zona.lat, zona.lon)
                break
            end if
        end for
    end for
end if
```

Decisione di variazione

```
scelta_nodo := MET1
deviaz := true
num_casuale := random(0, 1)
for area in lista_critica do
    prob := estrai_prob(area)
    if prob > num_casuale then
        coord := estrai_coordinate(area)
```

```
            deviazione(scelta_nodo, deviaz,
                coord.lat, coord.lon)
            break
        end if
    end for
```

3.2 Server

Il server si occupa di controllare da quanto tempo una zona MGRS non riceve dati da qualche sensore. Ogni 5 minuti di simulazione, per ogni zona esegue tale controllo con la funzione **calculateDataSentInRange(sensore, mgrs, *t* inizio, *t* fine)**, e se risulta vuota incrementa il valore di 1 nel vettore dedicato a questo, *vzone*. Al contrario, in caso di presenza di dati, il valore inerente all'area in esame sarà azzerato.
Al termine, il vettore viene salvato nella struttura *zoneNEmpty*, una `map` globale dedicata alla memorizzazione dei vettori di tutti gli algoritmi simulati ai quali è stato assegnato un identificativo numerico utile per distinguerli e recuperare il proprio vettore nell'applicazione.
La classe **Server** è a conoscenza delle zone MGRS e relative coordinate grazie alla funzione **getListZone** presente nella libreria **Utilities**, la quale consulta il file `MGRSzone.txt` creato dal primo modulo Python ed eventualmente salvato insieme agli altri file della simulazione già menzionati nella sezione "Salvataggio".
Per conoscere le zone MGRS vengono esaminati tutti i nodi del grafo, e convertite le loro coordinate latitudine e longitudine in MGRS secondo una certa precisione, indicata nel file di configurazione `PDA_config.txt`, per mezzo della funzione **toMGRS**, le quali vanno a popolare una lista se non già state inserite. Il grafo in questione è quello successivo alla creazione dei punti, procedura spiegata nel capitolo 2, che assieme ai nodi già presenti andranno a coprire il territorio in maniera completa con più facilità visto il loro numero elevato.

Viene poi esaminata questa lista, e con la funzione **mgrs_coord** viene scomposta la stringa sotto esame per costruire quella della zona confinante nell'angolo in alto a destra. Lo scopo di ciò è conoscere le coordinate del punto centrale della zona MGRS, in quanto con le funzioni **toLatLon** si ottengono quelle dell'angolo in basso a sinistra. Date queste due informazioni è possibile ricavare le coordinate del punto che si trova a metà della retta immaginaria che li collega, usate per calcolare la distanza di un partecipante da una zona. Al contrario, un utente potrebbe risultare più lontano di quanto non lo sia effettivamente. Infine, vengono scritte nel file `MGRSzone.txt`.

Implementazione

Controllo delle zone ogni 5 min e conta i time slot senza dati

```
if ora_corrente - controllo_prec >= 5min then
    for zona in lista_zone do
        for s in n_sensori do
            if calculateDataSentInRange(s, zona,
                controllo_prec, ora_corrente) == 0 then
                vzone[zona] += 1
                break
            end if
        if zona not vuota then
            vzone[zona] := 0
        end if
        end for
    end for
    controllo_prec := ora_corrente
    zoneNEmpty[dcf_index] := vzone
end if
```

Creazione file zone MGRS

```
G := grafo
precisione := leggi_precisione()

for nodo in G.nodi do
    mzone := converti_mgrs(nodo.lat, nodo.lon, precisione)
    if mzone not in lista_zone then
        lista_zone.aggiungi(mzone)
    end if
end for

for zona in lista_zone do
    #mgrs_coord() [
    updx := scomponi_&_calc_mgrs_updx(zona)
    zona_coord := mgrs_toLatLon(zona)
    updx_coord := mgrs_toLatLon(updx)
    centro := calc_coord_centrali(zona_coord, updx_coord)
    #mgrs_coord() ]
    write_file(zona, centro.lat, centro.lon)
end for
```

3.3 Cambio direzione - C++

Il cambio di direzione è gestito da una libreria scritta in linguaggio C++ responsabile di interfacciare il simulator engine con il modulo Python che effettua il vero lavoro, cioè produce i nuovi eventi, ed in seguito inserire quest'ultimi nell'apposita lista in maniera ordinata. Questa divisione permette di sfruttare le librerie descritte nel capitolo precedente per lavorare sul grafo generato, ed eventualmente salvato, successivamente alla scelta della città. Per riuscire ad effettuare questo collegamento sono utilizzate le Python/C API[1] che permettono di accedere all'interprete Python a vari

[1] https://docs.python.org/3/c-api/index.html

livelli.
La libreria è composta dalla funzione **changeDest** che nella prima chiamata, ad esempio da parte dell'applicazione, comincia inizializzando l'interprete Python e impostando i parametri utili per identificare il modulo all'interno della cartella del simulatore e le funzioni che contiene. Questo processo è racchiuso nella funzione **pyInit**. Di seguito è illustrata.

```
Py_Initialize();
PyObject* sysPath = PySys_GetObject("path");
PyList_Append(sysPath, PyUnicode_FromString("./"));
PyObject* pName = PyUnicode_FromString("ChangeDest");
PyObject* pModule = PyImport_Import(pName);
PyObject* pLoad =
    PyObject_GetAttrString(pModule, "loadGraph");
product = PyObject_CallFunction(pLoad, NULL);
pChange = PyObject_GetAttrString(pModule, "changeDest");
```

Con **Py_Initialize** viene inizializzato l'interprete, mentre con le due righe successive è impostato il percorso del file, con **PySys_GetObject("path")** che recupera `sys.path`, cioè una lista di stringhe che specifica i percorsi di ricerca dei moduli. A questo punto è indicato il nome del modulo (**ChangeDest**) e avviene la sua importazione. Infine, sono rese note le funzioni che contiene tramite **PyObject_GetAttrString**, per poterle chiamare all'interno della libreria. La **loadGraph**, dedicata alla generazione della mappa della città per il simulator engine, viene eseguita in questo punto con **PyObject_CallFunction**.
Dopo questa fase iniziale la **changeDest** si occupa di convertire i parametri da passare alla funzione del modulo in `PyObject*`, il tipo generale che può rappresentare qualunque oggetto Python, per mezzo della funzione **Py_BuildValue**, alla quale viene specificata una stringa che determina il tipo per la trasformazione. In questo lavoro le informazioni sono solo di due tipi, quindi `"f"` per `float` e `"i"` per `int`. Conclusi questi passaggi è

possibile invocare la funzione per il cambio di traiettoria o destinazione grazie a **PyObject_CallFunctionObjArgs** che accetta un numero variabile di parametri (l'ultimo deve essere `NULL`). Ad essa sono inoltrati gli oggetti creati. Di seguito è illustrato un estratto del codice appena descritto.

```
//First time
if(!pyIni) {
    pyInit(eventsL);
    pyIni = true;
}

PyObject* lat = Py_BuildValue("f", e->loc->lat);
PyObject* lon = Py_BuildValue("f", e->loc->lon);
PyObject* id_user = Py_BuildValue("i", e->id_user);
...

PyObject* deviat;
if(dev)
    deviat = Py_True;
else
    deviat = Py_False;

product = PyObject_CallFunctionObjArgs(pChange, lat, lon,
    id_user, day, timest, lat_dest, lon_dest, sec_event,
    speed, algo_choice, lat_dev, lon_dev, deviat, NULL);

if(!product) {
    PyErr_Print();
    exit(EXIT_FAILURE);
}
```

A questo punto entra in scena la funzione **plain** che si occupa di inserire nella lista i nuovi eventi creati, recuperati dal file generato dal modulo

Python sfruttando la funzione **readListOfEvents** già presente nella classe **Event**. Il principio di questa tecnica si basa sulla sostituzione, mediante posizione, dei vecchi eventi con quelli nuovi.
Inizialmente fa una ricerca nella lista per trovare la posizione dell'evento che ha scatenato il cambio di destinazione o traiettoria, memorizzandola nella variabile iteratore per non dover partire ogni volta dal principio, ricordando che viene esaminata una lista ordinata. Da questa posizione viene avviata un'altra ricerca all'interno della lista, incaricata di trovare le posizioni di tutti gli eventi successivi appartenenti all'utente corrente, salvate in un vettore.
Ora si presentano due casi:

- i nuovi eventi sono di numero inferiore rispetto a quelli vecchi, perciò si limita a fare la sostituzione per posizione e ad eliminare gli eccedenti;

- i nuovi eventi sono di numero maggiore rispetto a quelli vecchi, perciò procede con la sostituzione per posizione, ed i restanti vengono inseriti rispettando l'ordinamento.

Affinché questa libreria funzioni è necessario includere l'header `Python.h` ed in fase di compilazione aggiungere i flag `$(pkg-config --cflags --libs python3)`.
Oltre al metodo di inserimento appena descritto ne era stato ideato ed implementato anche un altro, denominato "a dizionario", ma poi abbandonato. Il funzionamento si basava sul mantenere una lista che ricordasse le posizioni degli eventi nella lista originale con cadenza di un minuto. Quando era da inserire quello nuovo veniva dapprima consultata la lista dizionario per conoscere in quale porzione si sarebbe collocato, ed in questa cercare la giusta posizione per rispettare l'ordinamento.
Nonostante fosse un'ottima idea, si sono riscontrate prestazioni decisamente peggiori rispetto al metodo usato, soprattutto in caso di un gran

numero di utenti simulati, dovute per lo più alla fase di cancellazione degli eventi vecchi. Ecco perché non viene spiegato nel dettaglio.

Implementazione

Inserimento nuovi eventi: funzione plain

```
Iteratore it
for it in lista_eventi do
    if it == evento then
        break
    end if
end for

Iteratore itt := it
for itt in lista_eventi do
    if itt.user == evento.user then
        inserisci_posizione(vec_posiz, n_posiz)
    end if
end for

if new_events < old_events then
    for i in new_events.size() do
        lista_eventi[vec_posiz[i]] := new_events[i]
    end for
    erase_events_remain(lista_eventi, evento.user)
else
    for i in vec_posiz.size() do
        lista_eventi[vec_posiz[i]] := new_events[i]
    end for
    n := vec_posiz[ultimo_valore]
    if n == ultima_posiz_lista_eventi then
        inserisci_coda(lista_eventi, new_events_remain)
```

```
    else
        inserisci_ordinata(lista_eventi, new_events_remain)
```

3.4 Cambio direzione - Python

Questo modulo è il vero responsabile del cambio di destinazione e/o traiettoria poiché è colui che produce nuovi eventi. Durante la simulazione viene consultato passando per la libreria C++ sopra descritta, grazie alle Python/C API.
Al suo interno sono presenti due funzioni, una per creare il grafo della città scelta, e l'altra per generare i nuovi eventi.
La prima, **loadGraph**, cerca di leggere dal file `Setup.txt`, prodotto dal primo modulo Python, il nome della città scelta. Se è presente, quindi si tratta di una simulazione realizzata con il simulatore modificato in questo lavoro, controlla se nella cartella `data` è stata salvata la mappa, ricordando che in caso negativo è un'opzione proposta all'utente. In caso affermativo, la "carica" in una variabile globale, modalità più veloce e che consente un'esecuzione offline, altrimenti la scarica come avviene normalmente avviando una nuova simulazione.
Se nel file `Setup.txt` non si trova il nome della città scarica la mappa della città Bagnacavallo, poiché è quella delle simulazioni archiviate nel simulatore precedente.
La funzione **changeDest** calcola il nuovo percorso da un punto di partenza ad uno di destinazione. Ciò vuol dire che in caso di deviazione prima origina gli eventi tra la posizione corrente dell'utente ed il punto di deviazione, e successivamente da tale punto alla destinazione.
Per questa attività si lavora con il grafo, ed è dunque necessario convertire le coordinate latitudine e longitudine in un nodo. Sono implementate 2 modalità:

- nodo vicino, cioè viene scelto il nodo più vicino alle coordinate, stabilito con la funzione **get_nearest_node** della libreria OSMnx;
- massima precisione, cioè viene aggiunto un nuovo nodo nel grafo, posizionato tramite le coordinate. In primo luogo viene cercata la strada (arco) più vicina, con la funzione **get_nearest_edge**. Dato che le coordinate appartengono a punti che certamente si trovano sulle strade, e non riguardano ad esempio palazzi, laghi, o altro, rintraccia quella giusta. In seguito vengono calcolate le due distanze tra il punto e le estremità dell'arco con **great_circle_vec**. Queste informazioni sono necessarie per l'inserimento del nodo e la sua valutazione durante la ricerca delle traiettorie. Infatti, con **add_node** viene inserito il nodo nel grafo, con le proprie coordinate ed un identificativo. Con le due **add_edge** si aggiungono gli archi che collegano il nuovo nodo con i nodi estremi dell'arco trovato in precedenza, assegnando loro le distanze e degli identificativi.
  Questo metodo è ovviamente più preciso, ma richiede più tempo. In questa tesi, i test sono stati effettuati adottando la tecnica precedente.

La funzione continua creando gli eventi in maniera pressoché identica a come è svolta nel primo modulo. A differenza di quest'ultimo, ora con **single_source_dijkstra** si ottiene un singolo percorso, da un nodo di partenza ad uno di destinazione. Per ogni coppia di nodi collegati da un arco lungo tale percorso viene applicata la funzione **bearing_calculator** e **destination_calculator** implementate nel primo modulo per calcolare le coordinate dove inserire il nuovo evento, ricordando che per questo compito fanno uso dell'angolo α di Azimut tra il punto cardinale nord e la posizione del secondo punto della coppia, delle coordinate dei nodi, e della distanza tra il nodo di partenza ed il nuovo da inserire misurata moltiplicando il tempo dell'intervallo (10 secondi) per la velocità di camminata dell'utente, quest'ultima rimasta fissa per gli utenti poiché precedentemente salvata nel file degli eventi `UserMovementsListEvents_x` proprio per questo scopo. Chiaramente il tempo delle letture ed i secondi trascorsi dell'utente

sono progressivi con quelli dell'evento scatenante, così come rimane il controllo sul tempo dell'utente, infatti raggiunto il suo limite di percorrenza il processo termina.
Nella figura 3.2 è rappresentato un esempio di funzionamento del cambio di direzione. È stata eseguita una simulazione nella città di Bologna impostando manualmente le coordinate per il punto di deviazione e si è scelto casualmente il percorso di un utente da raffigurare. La linea rossa traccia il percorso originale creato dal primo modulo Python. La partenza è identificata dal marker in basso a sinistra della figura. Il marker vicino simboleggia il punto in cui l'utente decide di cambiare traiettoria e chiaramente coincide con l'inizio del nuovo percorso, cioè la linea verde. Il marker più a destra è il punto di deviazione.
La figura 3.3 mostra la conclusione del nuovo percorso, la destinazione è la stessa.

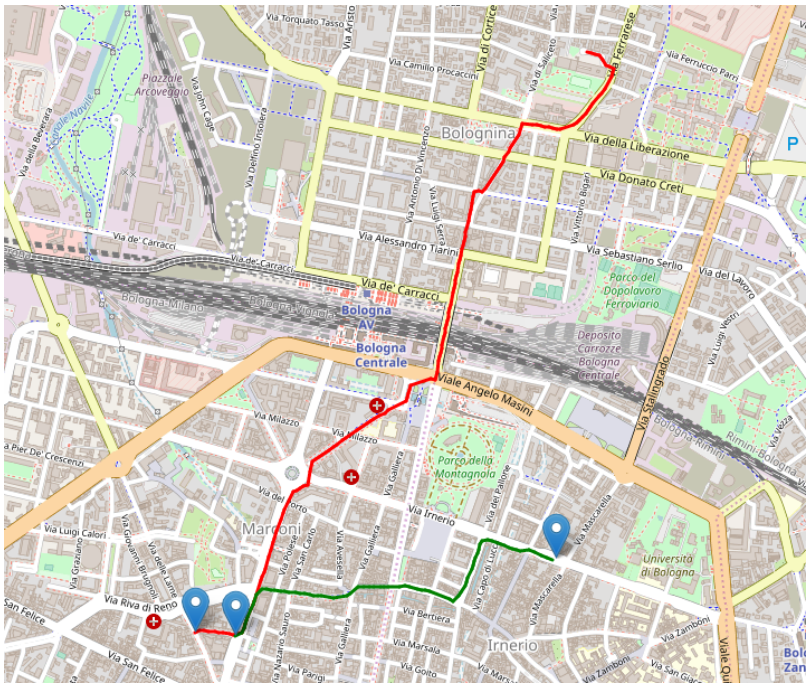

Figura 3.2: Percorso originale di un utente nella città di Bologna (linea rossa) e percorso fino a punto di deviazione (linea verde)

Figura 3.3: Percorso originale di un utente nella città di Bologna (linea rossa) e nuovo percorso (linea verde)

Implementazione

Cambio direzione: creazione nuovi eventi

```
G := grafo
if metodo == nodo_vicino then
    n_start := trova_nodo_vicino(G, lat_start, lon_start)
    n_end := trova_nodo_vicino(G, lat_end, lon_end)
else #massima precisione
    n_start := crea_nodo_preciso(G, lat_start, lon_start)
    n_end := crea_nodo_preciso(G, lat_end, lon_end)

percorso := calcola_percorso_ssd(G, n_start, n_end)
lista_archi := ottieni_lista_archi(G, percorso)
```

```
for arco in lista_archi do
    α := calcola_α(G, arco.inizio, arco.fine)
    for sec in tempo_percorrenza do
        distanza := sec * speed
        coord := new_event(G, arco.inizio, distanza, α)
        write_file(utente, mgrs, time, coord, sec, α,
            distanza, speed)
    end for
end for

if deviazione == True then
    n_start := nodo_deviazione
    n_end := nodo_destinazione
    ripeti_sopra()
end if
```

3.5 PROBA

La classe **PROBA** è stata implementata assieme alle librerie per il cambio di traiettoria per valutarne le prestazioni, andando a produrre un file `csv` con i tempi impiegati dalle simulazioni. L'importante è quindi effettuare nuovamente il calcolo per la generazione dei percorsi, mentre non è rilevante che il tragitto mutui effettivamente.
È composta, nella sua funzione principale, dalla **generateSample** come gli altri algoritmi, chiamata da ogni evento durante la simulazione.
Il cambio di traiettoria avviene secondo una certa probabilità, scelta dall'utilizzatore e specificata nel file di configurazione `Simulation_config.txt` al momento della scelta di adottare questo algoritmo. Le opzioni di chiamata della funzione **changeDest** sono sempre due, cioè effettuare esclusivamente il cambio di destinazione, oppure aggiungere la deviazione. Per quest'ultima preferenza viene scelto un evento a caso tra quelli rimasti

all'utente, che diventa il punto di passaggio. Infatti durante la fase di inizializzazione della simulazione vengono salvati negli smartphone degli utenti il loro numero di eventi, variabile decrementata ad ogni chiamata di questo algoritmo, quindi in seguito al loro avanzamento, e aggiornata dopo la creazione dei nuovi eventi.

Implementazione

```
num_event_remain -= 1
random := random()
Iteratore it

if random < probab then
    if deviaz == True then
        for it in lista_eventi do
            if it == evento then
                break
            end if
        end for

        if num_event_remain >= 1 then
            n_ev := random(num_event_remain)
            Iteratore itt := it
            for itt in lista_eventi and count < n_ev do
                if itt.user == evento.user then
                    count += 1
                end if
            end for
            dev_coord := itt.coord
        else
            dev_coord := dest.coord
    end if
```

```
    changeDest(sm, evento, lista_eventi, metodo,
        deviaz, dev_coord)
end if
```

3.6 Script e bug

In questa sezione vengono elencati gli script Python implementati per produrre i grafici visibili all'interno di questa tesi. Mettono in mostra i risultati ottenuti dalle simulazioni effettuate, nello specifico:

- grafo OSMnx delle città con quadrati MGRS;
- mappa che mostra il percorso originale di utente nella città di Bologna, e mappa dopo che ha effettuato la deviazione;
- grafico che illustra e confronta le prestazioni delle simulazioni al variare della probabilità di cambio, grazie alla classe PROBA, dell'algoritmo JFS-IBRIDO;
- heatmap delle città che visualizza il passaggio degli utenti nelle zone MGRS per gli algoritmi AO-JFS e JFS-IBRIDO;
- heatmap con le differenze di transito nelle zone MGRS delle città tra l'algoritmo JFS-IBRIDO, con le sue 2 varianti di deviazione, ed il AO-JFS.

Il simulatore presentava un bug al lancio di una nuova simulazione, ovvero terminava in errore senza eseguirla. Il problema era dato da una non corretta gestione dei percorsi dei file. Questi sono salvati nella libreria `Utilities.h` come variabili costanti, un modo semplice per richiamarli all'interno delle varie classi che compongono il simulatore. Nel primo modulo Python i percorsi indicati per la creazione dei nuovi file era diversa, portando in errore il simulatore durante l'inizializzazione delle sue strutture, perciò sono stati corretti in `crowdsensim2.py` e `menucl.py`, in

particolare per `UserMovementsListEvents_x.txt` e `Setup.txt`. Quest'ultimo serve anche per alcune informazioni nella pagina web finale.
In seguito all'aggiornamento della libreria OSMnx, nel primo modulo, sono stati aggiornati i nomi di alcuni parametri nelle funzioni **add_edge**.

Capitolo 4

Validazione

Il lavoro di questa tesi ha portato ad aggiungere al simulatore esistente una nuova funzionalità, cioè la deviazione di traiettoria di un percorso. Per la constatazione della sua correttezza e delle sue *performance* è stato implementato un caso di studio, definito precedentemente come applicazione, con sui svolgere le simulazioni. Per questa analisi sono state usate le mappe di tre città molto diverse tra loro per posizione geografica, tipologia di territorio, numero di abitanti e grandezza in kmq. Queste città sono: Bologna, Lussemburgo e Melbourne.

4.1 Città

Questa sezione descrive brevemente le città simulate ed i motivi delle loro scelte, ed illustra le mappe OSMnx dei territori con i quadrati MGRS.

Bologna

Bologna è capoluogo di provincia e della regione Emilia Romagna, è situata tra le montagne dell'Appennino tosco-emiliano ed il cuore della Pianura Padana. È la città più popolosa dell'Emilia Romagna e dal 1 gennaio 2015 è diventata città metropolitana. In totale, l'area metropolitana di Bologna conta una popolazione di circa 915.000 abitanti dei quali circa il

38% corrisponde alla città di Bologna (384.377). La densità di popolazione è pari a circa 264 ab/km^2. Il centro storico della città, anticamente racchiuso all'interno di una cerchia di mura medievali è oggi considerato il cuore pulsante della vita urbana. Un labirinto di portici e piazze.
Questa tesi è inerente ad un corso di studi all'interno dell'Università di questa città, è stata usata in lavori precedenti da parte di ricercatori e tesisti che hanno operato con versioni precedenti del simulatore, e come appena descritto è di dimensioni medio-grandi con un territorio abbastanza complesso. Sono i principali motivi che hanno favorito la sua selezione.
Il grafo OSMnx contiene 62271 nodi e 136488 archi racchiusi in 166 quadrati MGRS da 1 km.

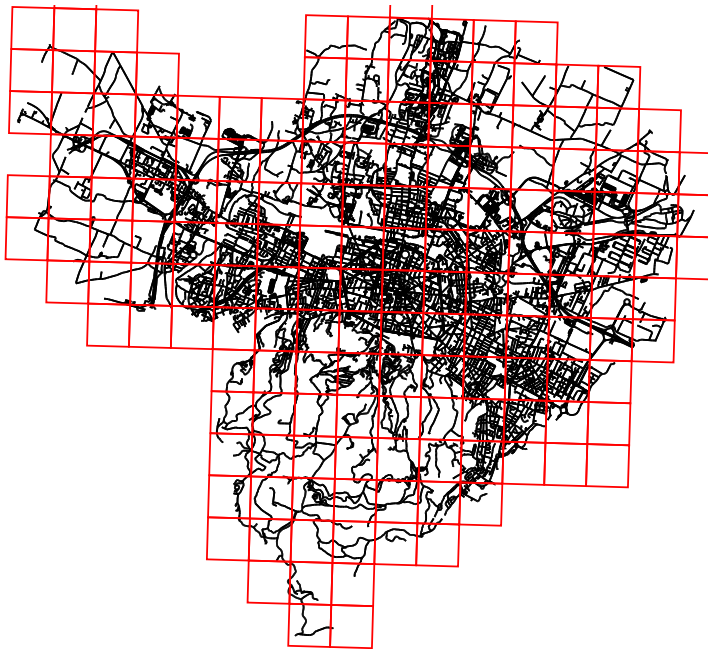

Figura 4.1: Bologna

Lussemburgo

Lussemburgo città capitale dello Stato omonimo (119.752 ab. Nel 2018) è nota per la città vecchia fortificata di origine medievale costruita su uno sperone roccioso alla confluenza dei fiumi Pétrusse con l'Alzette. Lo sperone su cui è stata costruita è per tre lati a strapiombo su dei canali naturali, per cui sin dal Medioevo la città è stata considerata una fortezza.
Il Granducato del Lussemburgo è una piccola Monarchia europea che confina con Belgio, Francia e Germania. Per la maggior parte presenta un paesaggio rurale.
La prima versione del simulatore è stata creata presso l'Università di questa città, la quale tutt'ora collabora, quindi è stata scelta per questioni di compatibilità con test precedenti.
Il grafo OSMnx contiene 36285 nodi e 80776 archi racchiusi in 72 quadrati MGRS da 1 km.

Melbourne

Melbourne Città dell'Australia sud-orientale, si trova nella parte più interna della Baia di Port Phillip, che si apre sullo Stretto di Bass. Il distretto urbano ha una superficie di 1850 km^2 e accentra circa il 75% della popolazione del suo stato. È la capitale dello Stato di Victoria ed è la seconda città più popolosa dell'Australia dopo Sydney.
Uno sguardo dall'alto sulla città mostra la sua pianta lineare e la griglia ordinata di strade che la rendono vivibile e facile da girare, presenta lunghi viali intersecati da una fitta rete di strade.
Delle tre città presenta meno zone, poiché viene scaricata esclusivamente la parte CBD (*Central Business District*) e le aree limitrofe, e come appena scritto una rete stradale atipica fatta di strade dritte. Questo è il motivo che ha portato alla sua scelta.
Il grafo OSMnx contiene 34969 nodi e 84026 archi racchiusi in 59 quadrati MGRS da 1 km.

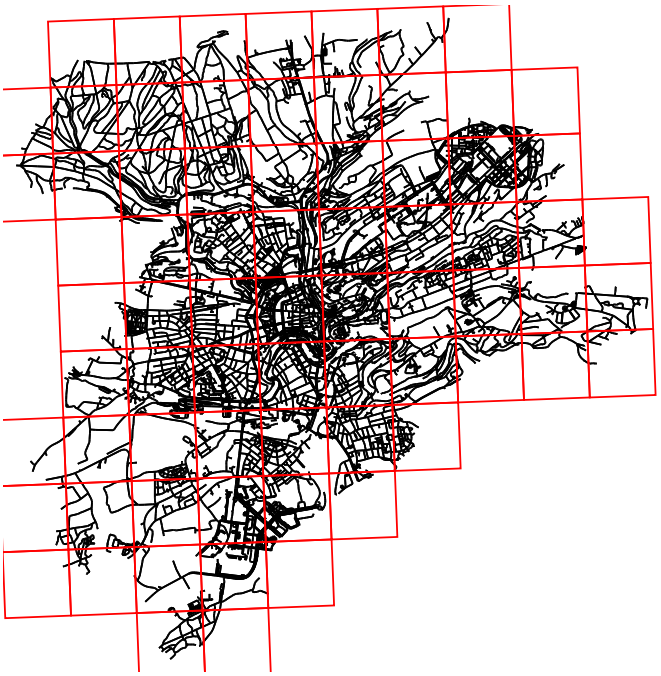

Figura 4.2: Lussemburgo

4.2 Performance

Prima di svolgere simulazioni test adottando un caso di studio, cioè l'applicazione finale, sono state valutate le *performance* della nuova funzionalità. Questo ha comportato l'esecuzione di simulazioni impiegando l'algoritmo PROBA, variando le probabilità. Nel capitolo 3 sono presenti le figure che mostrano la corretta generazione del nuovo percorso, però è importante anche valutare se l'implementazione è valida da un punto di vista tempistico.
Logicamente ad un valore di probabilità più alto corrisponde un tempo maggiore per la simulazione, dovuto ad un incremento nel numero di

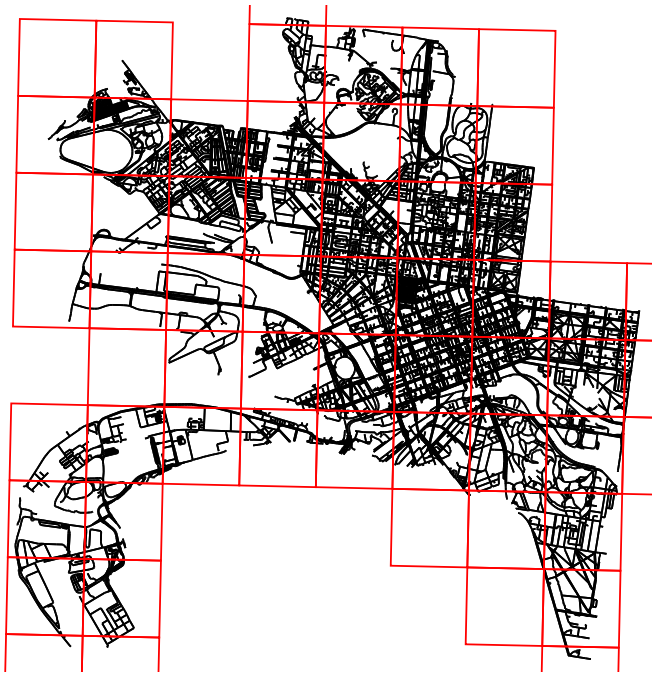

Figura 4.3: Melbourne

cambi di traiettoria. Il caso pessimo prevede una crescita esponenziale del tempo, mentre l'obiettivo è renderla il più lineare possibile. Questa valutazione è stata realizzata in due casi specifici: con e senza deviazione. Il test delle performance è stato effettuato con i seguenti valori di probabilità: 0, 0.25, 0.50, 0.75 e 1%. La città scelta è Bologna, con 1000 utenti. Il computer utilizzato è caratterizzato da un processore Intel Pentium Dual Core T3200 a 2,00 GHz, 3 GB di RAM e sistema operativo Ubuntu 18.04.4 LTS.

La figura 4.4 rappresenta il grafico che illustra i risultati ottenuti. È da sottolineare che essendo basati sulla probabilità i tempi delle simulazioni cambiano lievemente, a seconda di quanti cambi vengono svolti rispetto

ad un'esecuzione precedente o successiva. Detto questo, è l'esito aspettato.

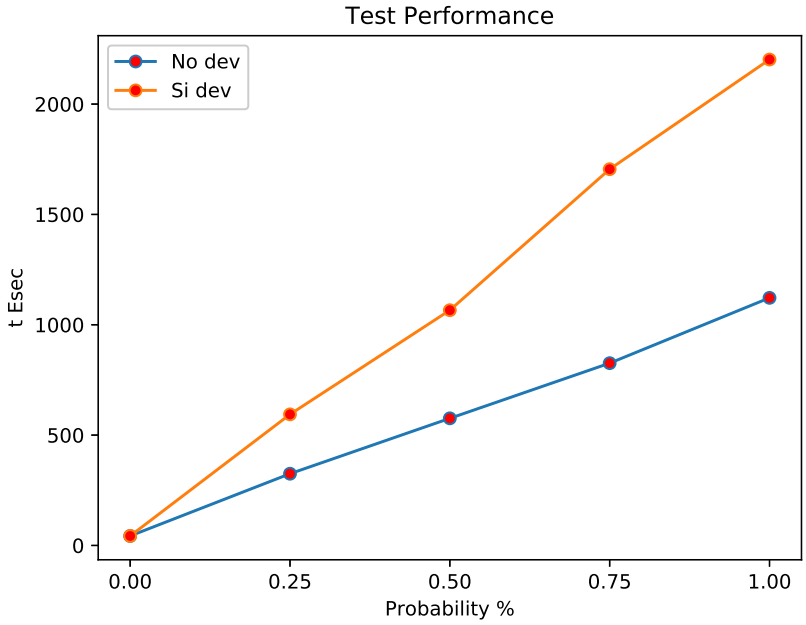

Figura 4.4: Performance secondo diverse probabilità

4.3 Validazione dell'applicazione

In questa sezione vengono presentate le simulazioni effettuate per verificare la validità dell'applicazione finale. Le città esaminate sono sempre le stesse, cioè Bologna, Lussemburgo e Melbourne. Rispetto al caso precedente, test delle prestazioni, il numero degli utenti simulati aumenta considerevolmente, ovvero 10000.
Per analizzare il corretto funzionamento dell'implementazione della nuova funzionalità attraverso l'applicazione viene prodotto un file contenente i movimenti degli utenti assieme ad altri dati, per ogni simulazione condotta. La sua creazione avviene prelevando informazioni da diverse

classi, cioè **Simulation**, le due degli algoritmi AO-JFS e JFS-IBRIDO, e **PDA_LookAhead**. Nello specifico, per ogni evento, comprende:

- identificatore dell'utente;
- coordinate latitudine e longitudine;
- coordinata MGRS;
- ora dell'evento trasformata in secondi;
- valori del *SI* per ogni sensore;
- valori 0 o 1 in base a se il sensore ha generato dati.

Al fine di interpretare al meglio i risultati delle simulazioni, in maniera rapida e precisa, viene successivamente sfruttato per la costruzione di *heatmap*, anche se utilizzando esclusivamente le coordinate MGRS, le quali vengono scomposte secondo la precisione scelta per rappresentare le zone della città e sulle quali vengono conteggiate le presenze degli utenti. Ricordiamo che l'applicazione cerca di rispecchiare la realtà introducendo il participatory crowdsensing, quindi i partecipanti decidono se accettare la richiesta del server e dirigersi verso la zona richiesta o meno, in base alle formule viste nel capitolo 3 durante la descrizione dell'applicazione finale. Le heatmap, dunque, raffigurano le città con i rispettivi quadrati MGRS con precisione di 1 km, i quali assumono colori diversi, in scala lineare, in base alla presenza dei partecipanti che li attraversano durante i propri percorsi, visibili con esattezza dal numero stampato all'interno. Per ogni città sono state svolte 3 tipologie di simulazioni, cioè con gli algoritmi AO-JFS e JFS-IBRIDO con le sue due modalità, per distanza e probabilità. Da queste vengono ricavate 5 heatmap: assieme alle 3 naturalmente sviluppate in seguito a queste simulazioni, vengono aggiunte le 2 che mostrano le differenze tra quelle di JFS-IBRIDO e AO-JFS. Chiaramente il file degli eventi originato dal primo modulo rimane il medesimo.

Per comprendere meglio l'andamento delle esecuzioni è opportuno esaminarne momenti interni e non solo l'esito finale. A tale scopo sono stati scomposti i file in step di 5 minuti. Le mappe visualizzate nel capitolo sono inerenti al periodo che intercorre dal al minuto 75 al 79 e 59 secondi. In questa sezione sono state inserite esclusivamente le heatmap del Lussemburgo per mantenere una migliore continuità del testo, essendo anche più facili da leggere i valori interni. Per ragioni di spazio, dunque, le restanti immagini (comprese quelle appena citate) sono state spostate in un'apposita appendice.
Nelle figure 4.5, A.1 e A.11, che rappresentano le heatmap relative all'algoritmo AO-JFS, si nota subito che le zone senza letture dati sono nella parte esterna del territorio, identificate dal quadrato scuro col numero 0. Analizzando i grafi OSM delle città illustrati precedentemente in questo capitolo (figure 4.1, 4.2 e 4.3), si osserva che i quadrati MGRS inerenti quelle zone contengono piccoli tratti di strade, a sottolineare questo alcuni addirittura sembrano vuoti, è necessario zoomare sul grafo per vederli. Ad esempio, nelle mappe generate si sono scoperte aree con probabilità di transito estremamente vicina allo 0, quindi si può considerare trascurabile. Per la città di Bologna sono 2: in alto a destra, sopra la frazione Primo Maggio, e in basso a destra, sopra la frazione Villanova di Castenaso. Stessa particolarità succede nella città di Melbourne con i settori in basso a destra.
Le figure riguardanti l'applicazione dell'algoritmo JFS-IBRIDO per destinazione, oltre a mostrare la copertura delle zone vuote incontrate prima, mettono in evidenza il limite imposto nella formula per il calcolo della volontà degli utenti di soddisfare le richieste del server, ovvero oltre i 3 km di lontananza le rifiutano. In figura 4.8 si vedono nella parte centrale e alta, ed in qualche zona in basso a destra, le zone con valori uguali a 0, a significare che non c'è stata differenza nel numero di transito in quanto troppo distanti da quelle vuote, di conseguenza i partecipanti non hanno modificato i propri percorsi. È un comportamento maggiormente visibile nel territorio di Bologna, essendo decisamente più esteso rispetto agli altri,

infatti in figura A.4 si trova in tutta la porzione centrale. Nella A.14 si può vedere in particolare nella parte sinistra. Questo algoritmo è in grado di offrire più copertura delle regioni carenti, specialmente nel breve periodo, ma con valori limitati.
Al contrario, con JFS-IBRIDO per probabilità c'è più movimento generale, infatti gli utenti sono meno intimoriti dalla distanza, accettando più volentieri lo spostamento, e suscettibili al tempo trascorso in assenza di dati da parte di un'area. Le zone attorno a quelle raggiunte registrano così presenze maggiori, ad esempio nella parte sinistra del Lussemburgo, in figura 4.7, così come quelle nei pressi delle aree con densità superiori. Una chiave di lettura è data dal tempo utile per raggiungere la zona, ovvero mentre un partecipante si dirige verso il punto designato, in particolare se distante, il periodo senza letture dell'area aumenta, richiamando così più gente. Inoltre, anche se un utente dovesse arrivare prima di un altro, quest'ultimo continuerebbe il proprio cammino verso la medesima destinazione in attesa di una nuova richiesta del server, la quale potrebbe essere rifiutata, proseguendo sullo stesso percorso, o accettata facendo modificare ulteriormente il tragitto. Ricordando che entrambi vengono a conoscenza della situazione territoriale con cadenza di 5 minuti. Questa peculiarità si accentua in caso di città grande, come Bologna, oppure con una conformazione particolare come Melbourne, che si può definire poco "connessa" e le persone, ad esempio, per giungere nelle zone a sud-ovest devono costeggiare la baia.
Nella tabella 4.1 sono elencati i tempi impiegati per svolgere le simulazioni che hanno prodotto le heatmap inserite, ed il numero di deviazioni effettuate. Come descritto, si nota una ragguardevole differenza tra le due tipologie di algoritmo JFS-IBRIDO, infatti Bologna ha valori 7 volte superiori con la modalità per probabilità rispetto a quella per destinazione, mentre Melbourne 4 volte più elevati con lo stesso criterio. È chiaro che il tempo di simulazione è strettamente correlato al numero di deviazioni.

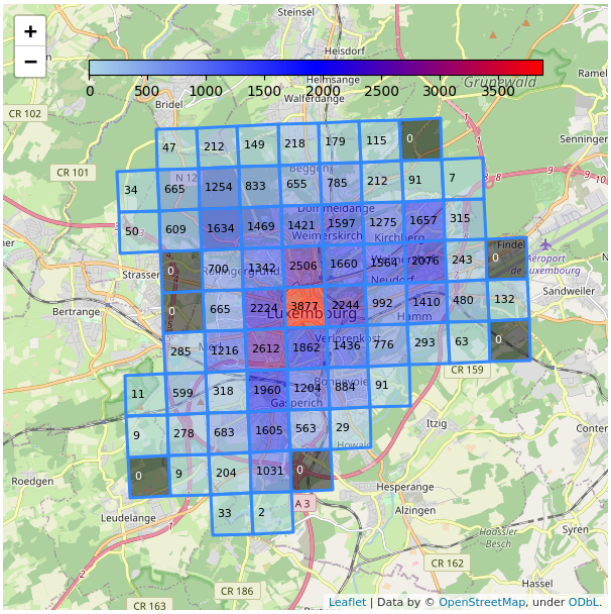

Figura 4.5: Heatmap AO-JFS di Lussemburgo

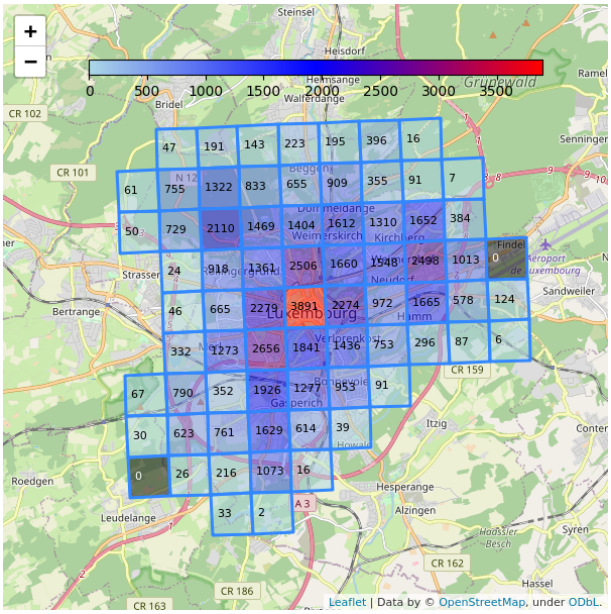

Figura 4.6: Heatmap JFS-IBRIDO per destinazione di Lussemburgo

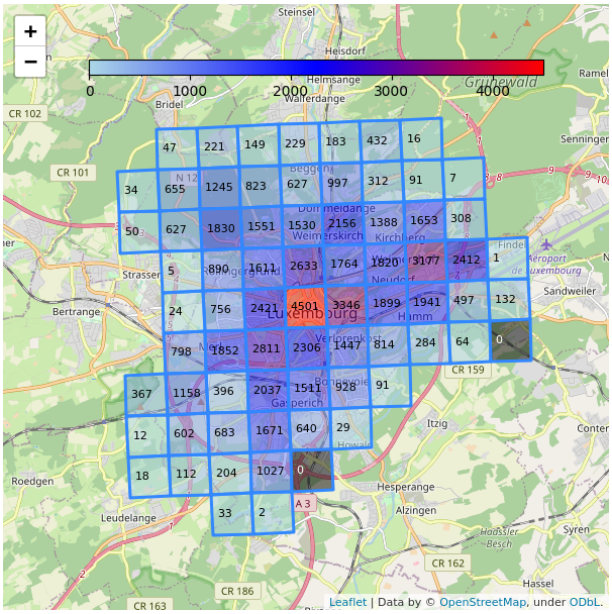

Figura 4.7: Heatmap JFS-IBRIDO per probabilità di Lussemburgo

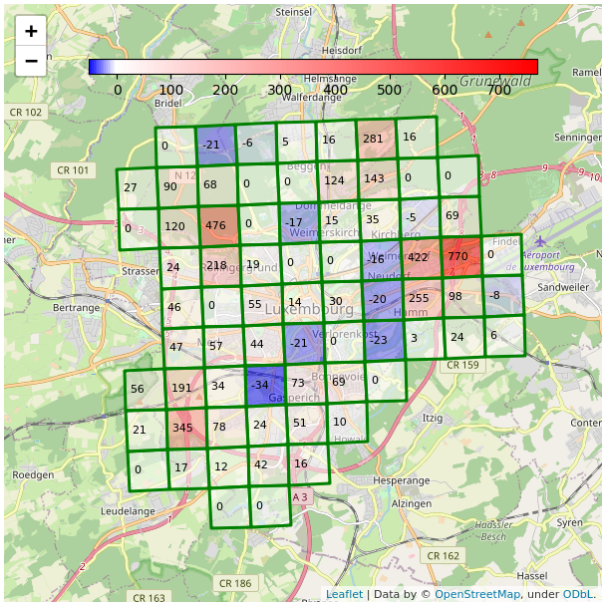

Figura 4.8: Heatmap che mostra la differenza tra JFS-IBRIDO per destinazione e JFS di Lussemburgo

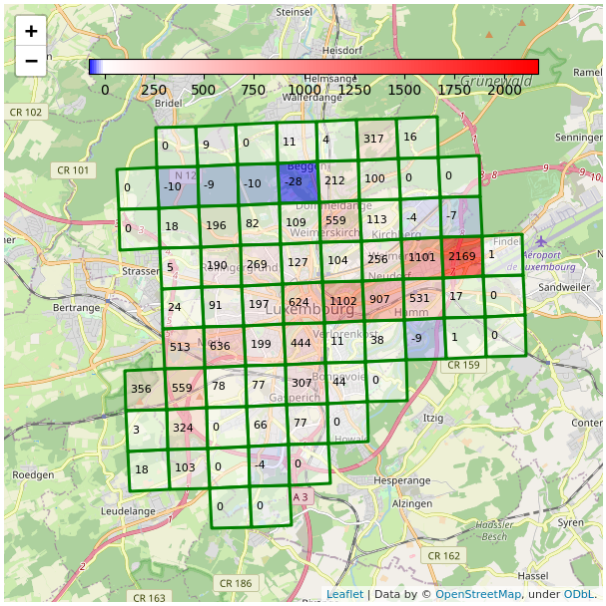

Figura 4.9: Heatmap che mostra la differenza tra JFS-IBRIDO per probabilità e JFS di Lussemburgo

	Bologna	Lussemburgo	Melbourne
AO-JFS			
SIMULATION TIME	852 sec	415 sec	610 sec
real	14m13s	6m55s	10m9s
user	9m27s	4m17s	5m24s
sys	3m41s	2m35s	3m39s
JFS-IBRIDO Dest.			
SIMULATION TIME	4566 sec	5297 sec	6950 sec
real	76m7s	88m16s	115m49s
user	63m4s	68m49s	94m29s
sys	12m37s	15m28s	18m41s
deviazioni	445	405	907
JFS-IBRIDO Prob.			
SIMULATION TIME	33710 sec	11183 sec	25161 sec
real	561m53s	186m24s	419m22s
user	476m32s	153m18s	347m42s
sys	83m8s	32m29s	70m7s
deviazioni	3395	870	3406

Tabella 4.1: Tempi delle simulazioni e numero di deviazioni

Conclusioni

Questa tesi presenta CrowdSenSim, un simulatore di attività MCS in ambienti urbani realistici, iniziando con la descrizione della seconda versione ed illustrando le differenze rispetto alla precedente, in particolare una nuova gestione spaziale, la trasformazione in simulazioni stateful ordinando temporalmente gli eventi, l'aggiunta di algoritmi per la gestione delle letture da parte degli utenti e loro esecuzione parallela, che assieme alla riscrittura del codice hanno portato al miglioramento delle performance, riducendo il tempo di esecuzione e utilizzazione della memoria. Espone la sua architettura, la quale è composta da due moduli scritti in linguaggi di programmazione distinti. Ne vengono spiegate le funzionalità e le implementazioni, come la creazione dei punti e dei percorsi, l'evoluzione della simulazione, e gli algoritmi presenti.
Il lavoro di questa tesi ha aumentato le funzionalità del simulatore mantenendo retro-compatibilità, in quanto rimane possibile effettuare le medesime simulazioni della versione precedente. Il limite principale del simulatore attuale è dato dal percorso che gli utenti seguono, il quale viene loro assegnato nella prima fase della simulazione. Se in alcune zone non c'è transito non vengono effettuate letture, e la copertura territoriale viene meno, come la completezza dei dati. La nuova funzionalità prevede la possibilità di modificare il tragitto degli utenti, portando all'aggiunta del concetto di participatory crowdsensing, in quanto i partecipanti hanno facoltà di scegliere se soddisfare le richieste di spostamento in una certa aerea territoriale da parte del server, il quale è stato introdotto in questa

versione. Grazie all'implementazione di una classe che permette di selezionare se cambiare solo destinazione oppure aggiungere la deviazione, secondo una probabilità stabilita in fase di configurazione, si è verificato che le prestazioni delle nuove simulazioni sono temporalmente ottimali. Come esempio di caso d'uso, è mostrata la valutazione dell'applicazione finale, cioè di un algoritmo distribuito opportunistico per la raccolta di dati modificato aggiungendo le capacità incluse, su tre città differenti per dimensione e tipologia di territorio urbano, la quale manifesta il raggiungimento dell'obiettivo di risolvere il problema dei dati sparsi. Inoltre, anche quest'ultima versione ha la potenzialità di esecuzione offline.

Appendice A

Heatmap delle città

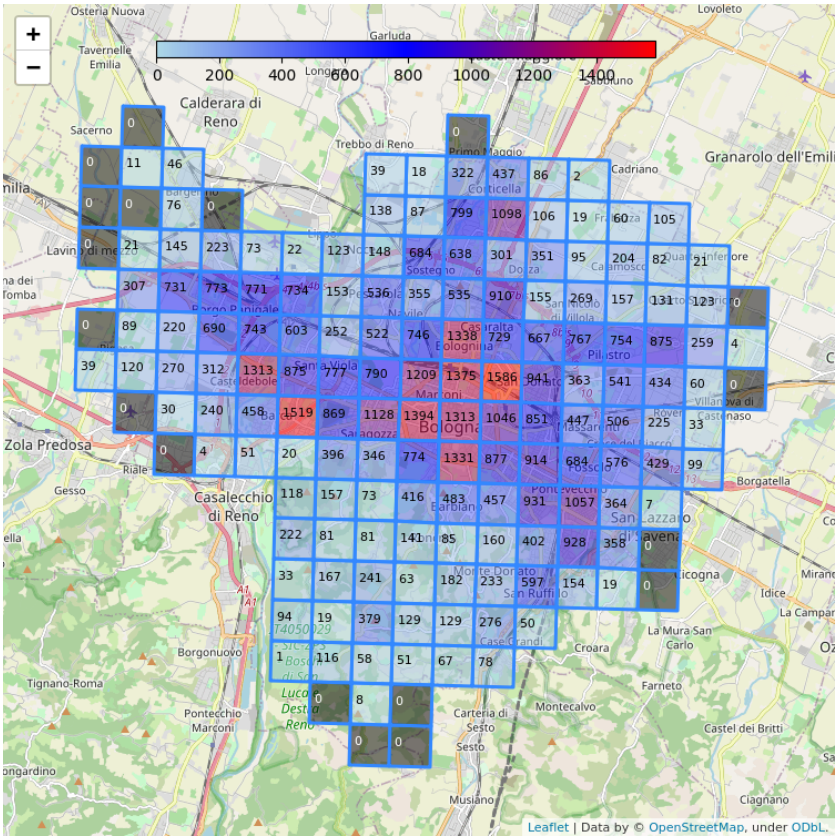

Figura A.1: Heatmap AO-JFS di Bologna

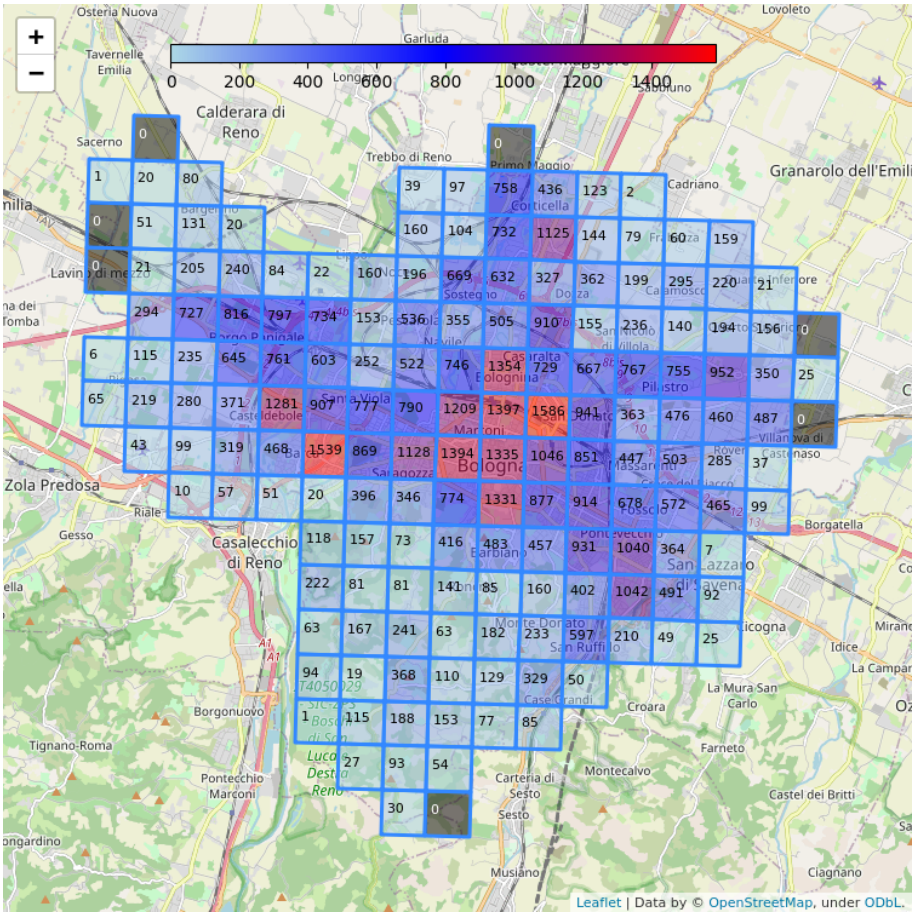

Figura A.2: Heatmap JFS-IBRIDO per destinazione di Bologna

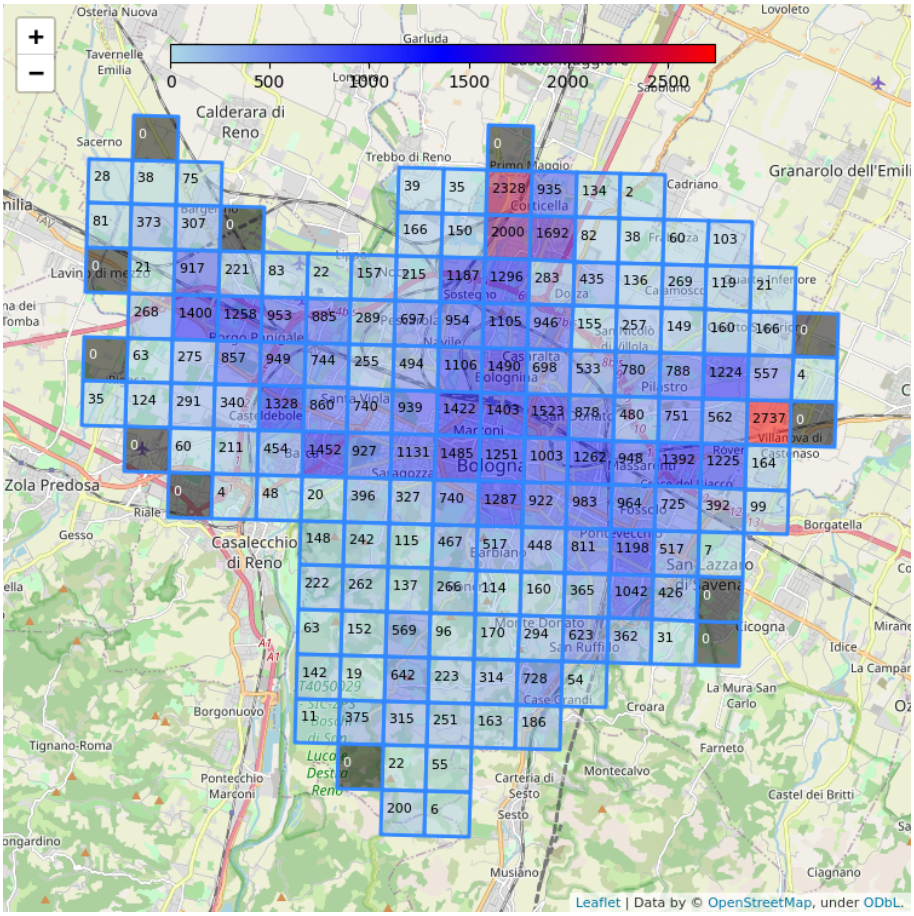

Figura A.3: Heatmap JFS-IBRIDO per probabilità di Bologna

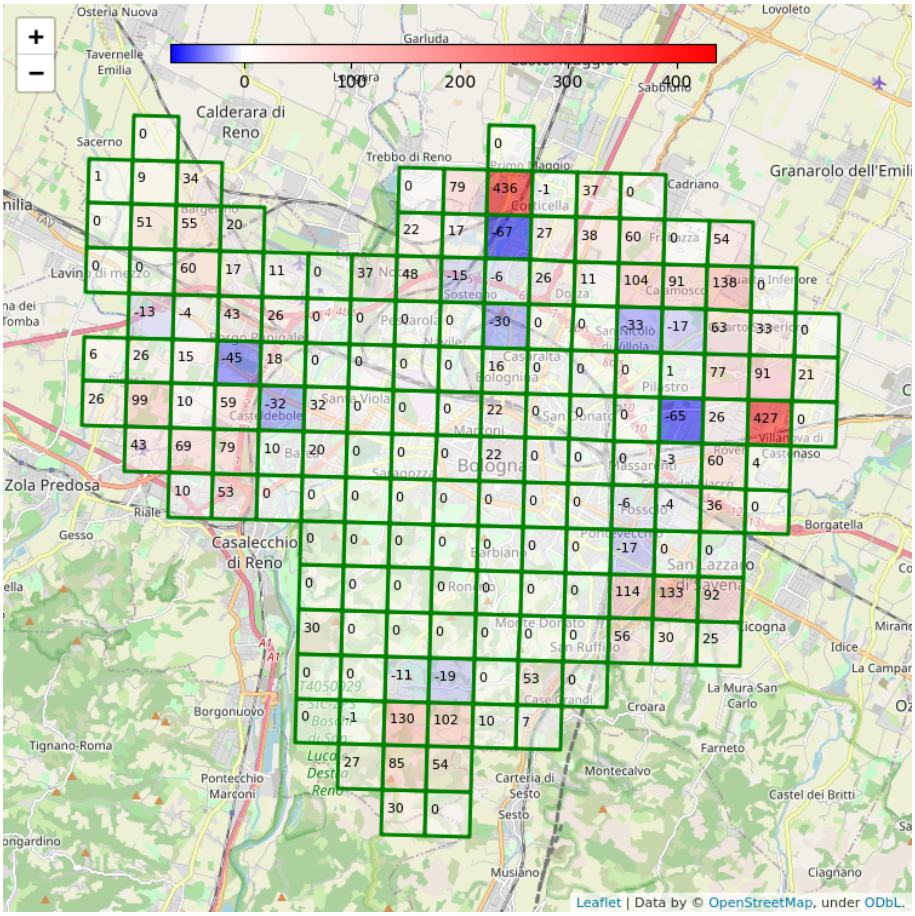

Figura A.4: Heatmap che mostra la differenza tra JFS-IBRIDO per destinazione e JFS di Bologna

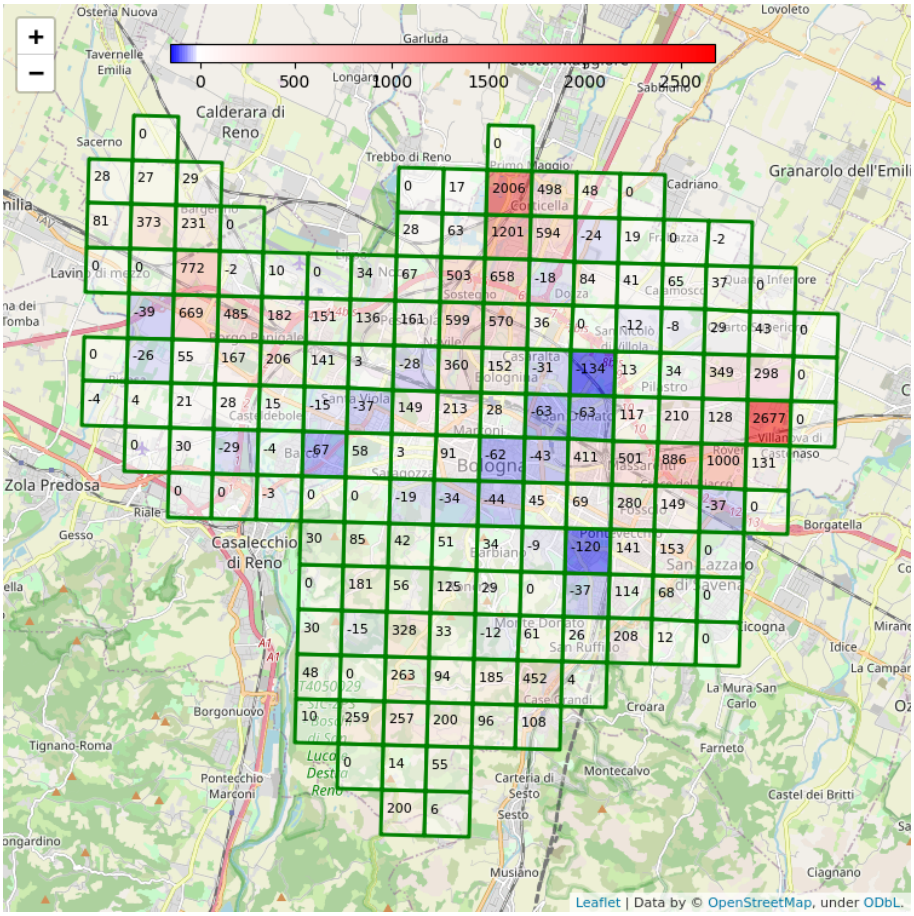

Figura A.5: Heatmap che mostra la differenza tra JFS-IBRIDO per probabilità e JFS di Bologna

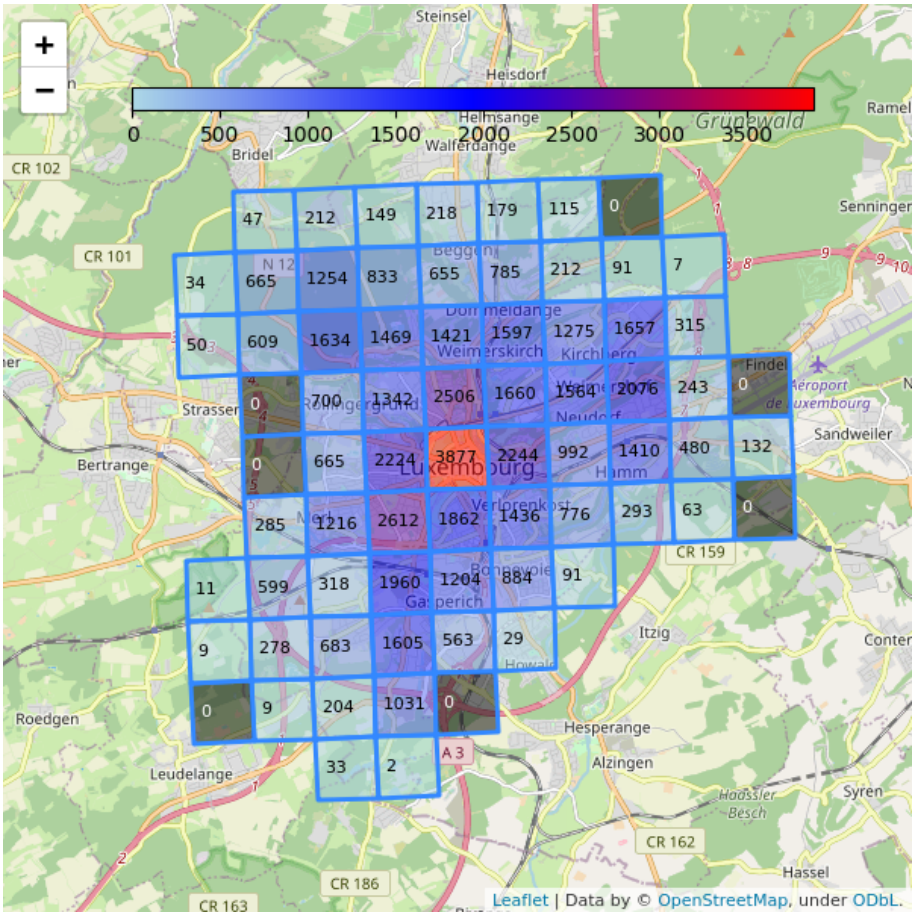

Figura A.6: Heatmap AO-JFS di Lussemburgo

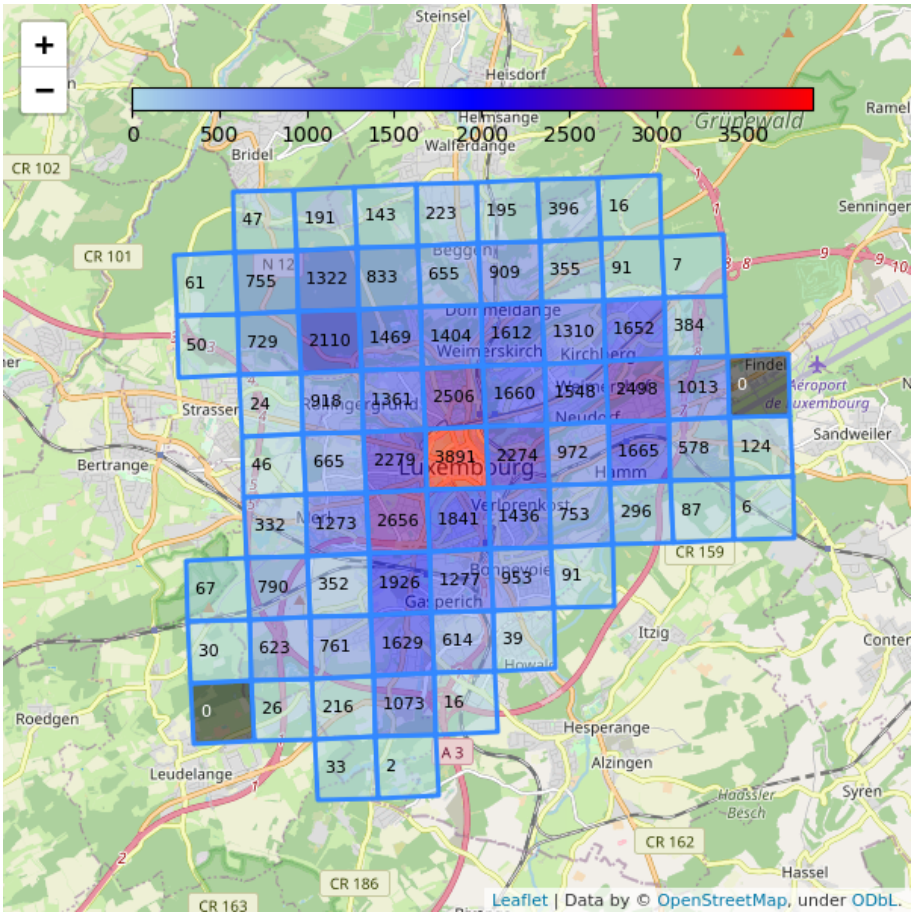

Figura A.7: Heatmap JFS-IBRIDO per destinazione di Lussemburgo

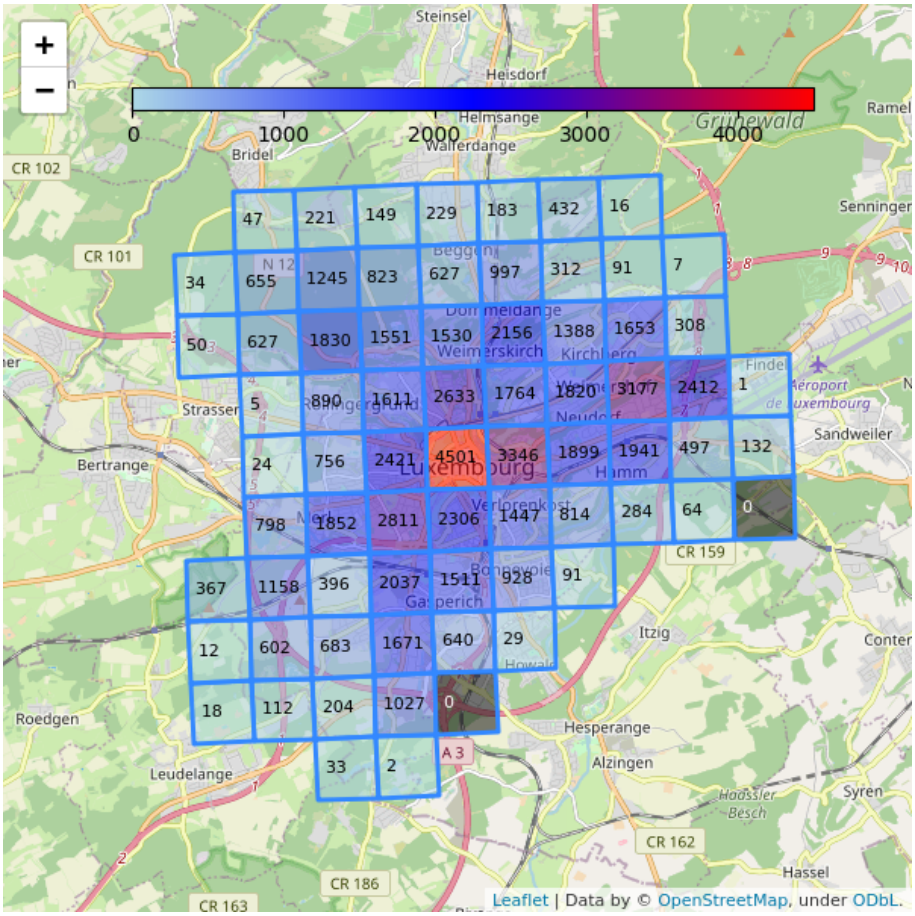

Figura A.8: Heatmap JFS-IBRIDO per probabilità di Lussemburgo

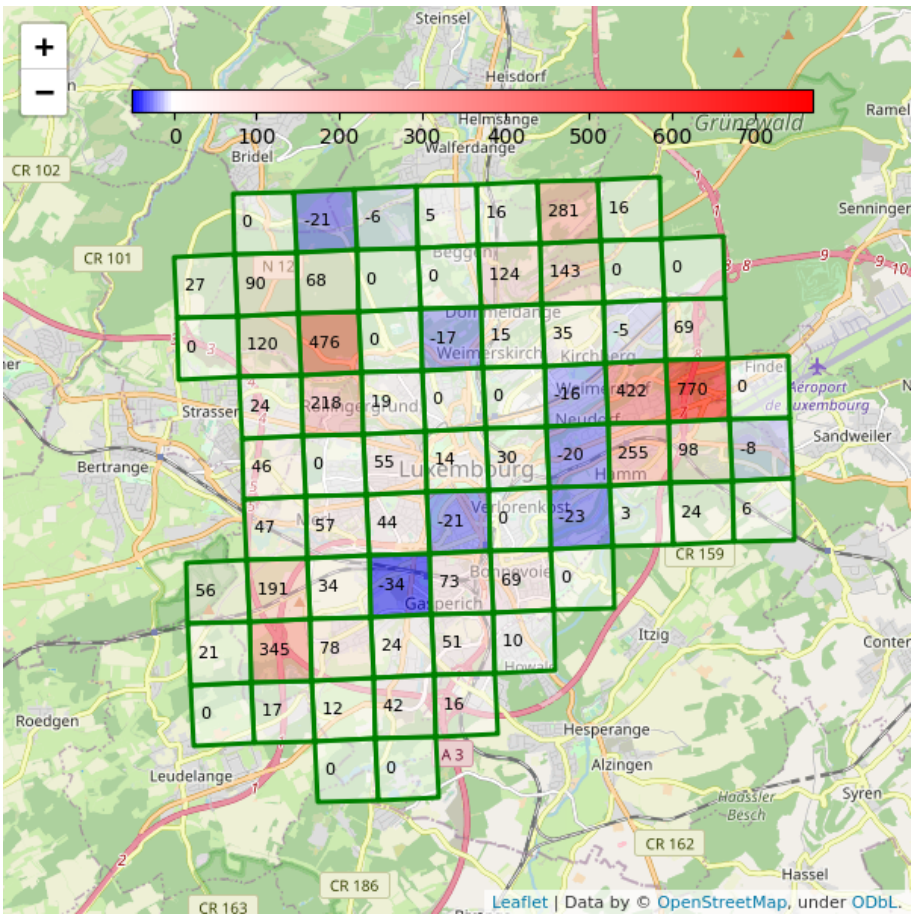

Figura A.9: Heatmap che mostra la differenza tra JFS-IBRIDO per destinazione e JFS di Lussemburgo

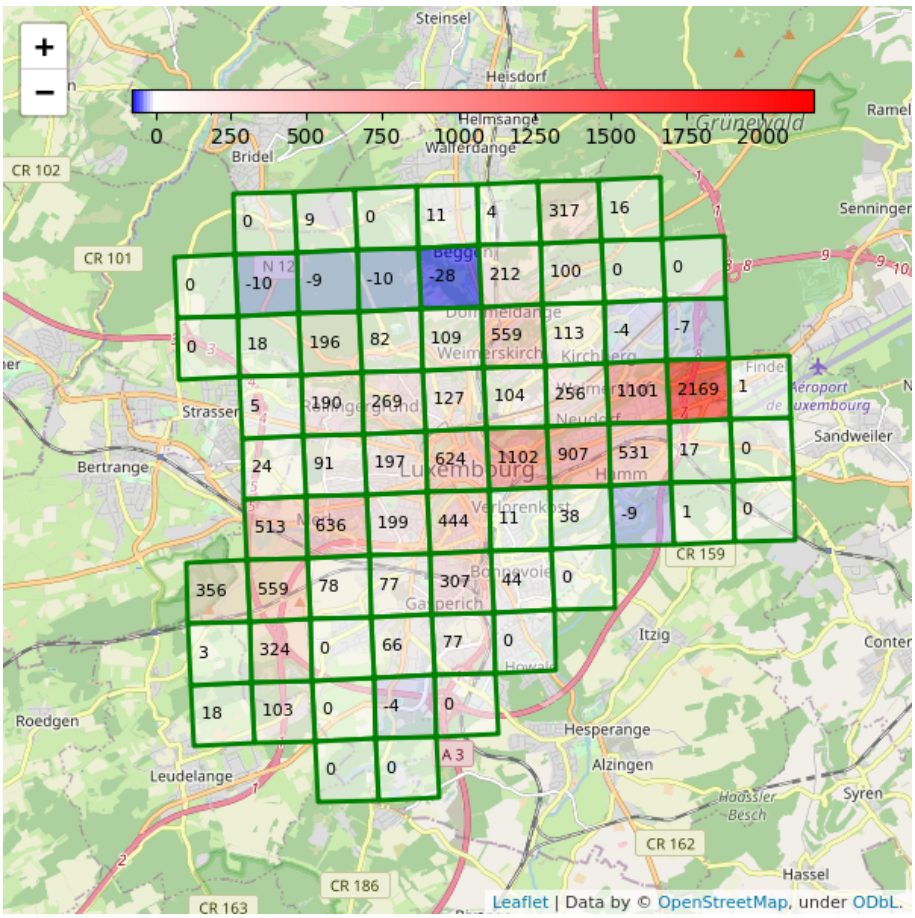

Figura A.10: Heatmap che mostra la differenza tra JFS-IBRIDO per probabilità e JFS di Lussemburgo

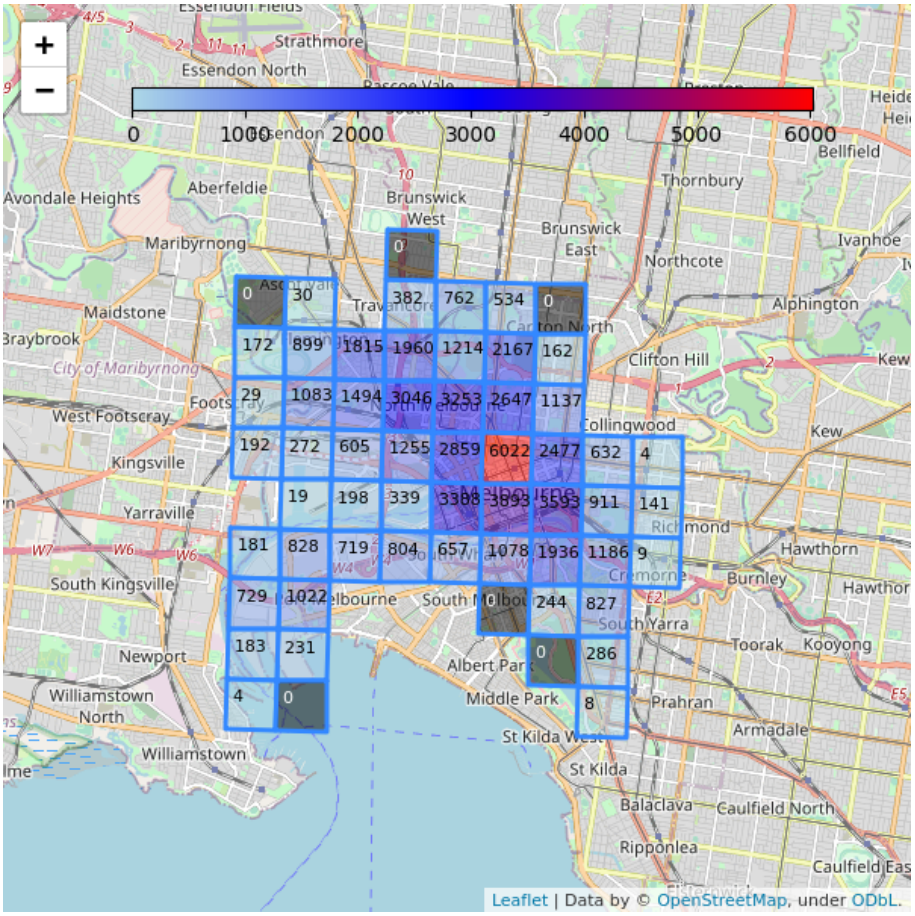

Figura A.11: Heatmap AO-JFS di Melbourne

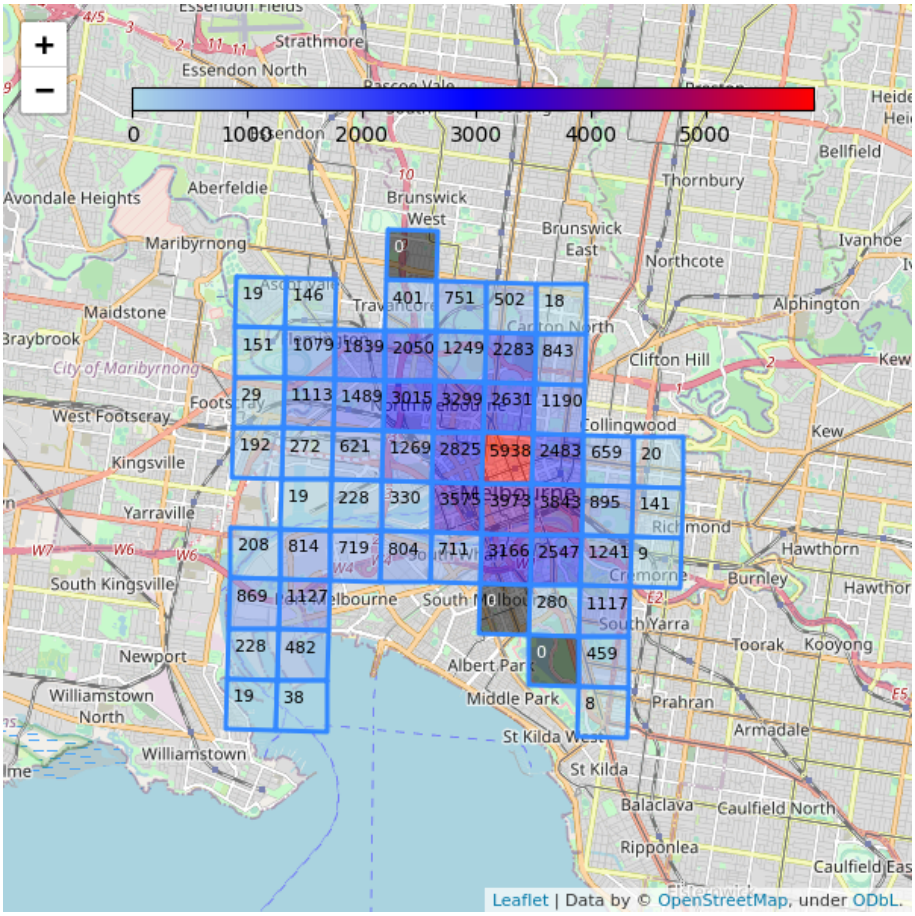

Figura A.12: Heatmap JFS-IBRIDO per destinazione di Melbourne

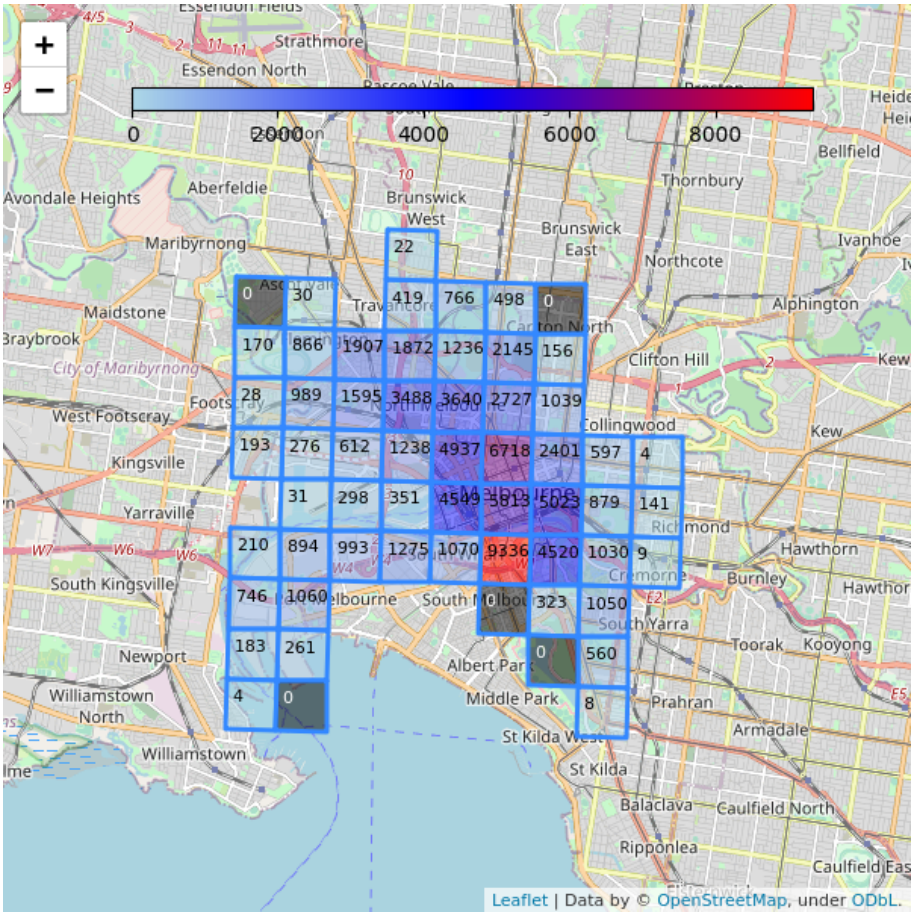

Figura A.13: Heatmap JFS-IBRIDO per probabilità di Melbourne

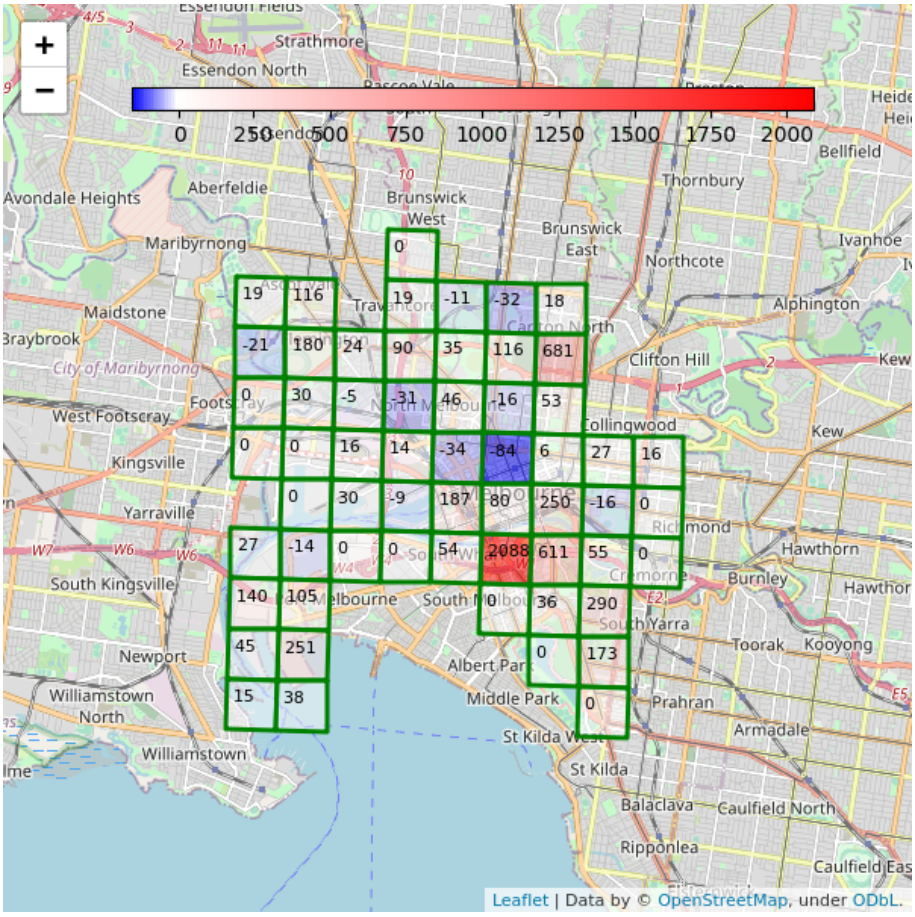

Figura A.14: Heatmap che mostra la differenza tra JFS-IBRIDO per destinazione e JFS di Melbourne

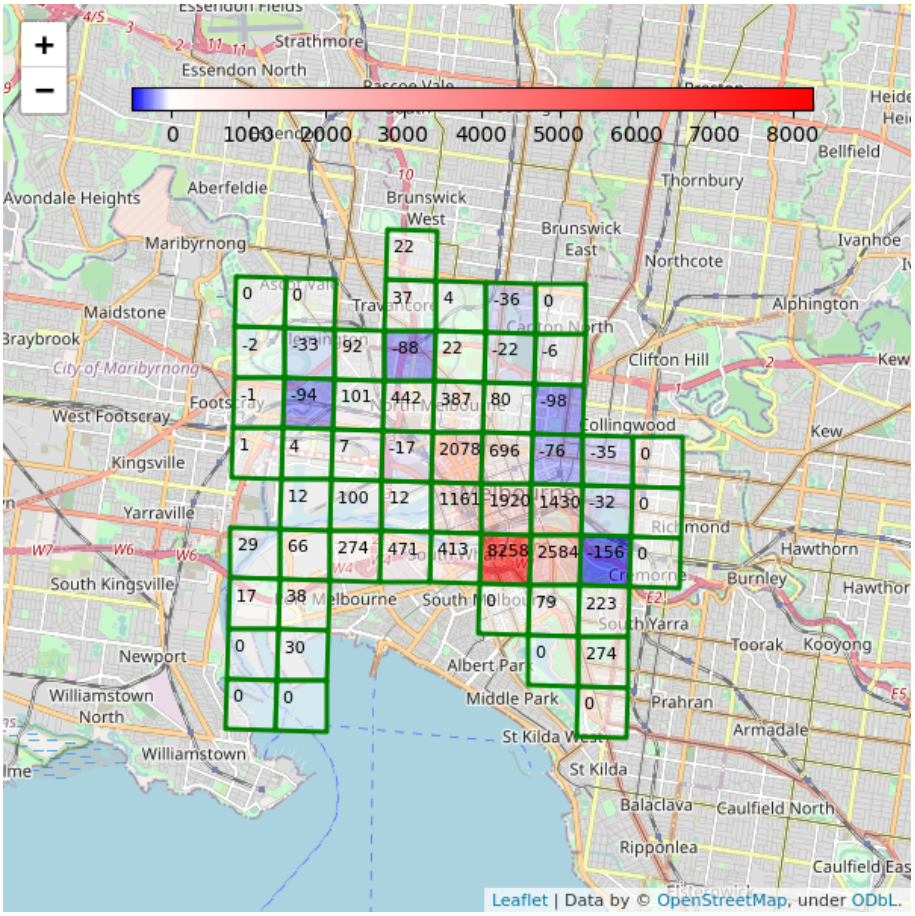

Figura A.15: Heatmap che mostra la differenza tra JFS-IBRIDO per probabilità e JFS di Melbourne

Bibliografia

[1] Ioannis Boutsis and Vana Kalogeraki. Privacy preservation for participatory sensing data. In *2013 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 103–113. IEEE, 2013.

[2] Andrea Capponi, Claudio Fiandrino, Burak Kantarci, Luca Foschini, Dzmitry Kliazovich, and Pascal Bouvry. A survey on mobile crowdsensing systems: Challenges, solutions, and opportunities. *IEEE Communications Surveys & Tutorials*, 21(3):2419–2465, 2019.

[3] Cory Cornelius, Apu Kapadia, David Kotz, Dan Peebles, Minho Shin, and Nikos Triandopoulos. Anonysense: privacy-aware people-centric sensing. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, pages 211–224, 2008.

[4] Emiliano De Cristofaro and Claudio Soriente. Participatory privacy: Enabling privacy in participatory sensing. *IEEE network*, 27(1):32–36, 2013.

[5] Károly Farkas and Imre Lendák. Simulation environment for investigating crowd-sensing based urban parking. In *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, pages 320–327. IEEE, 2015.

[6] Claudio Fiandrino, Andrea Capponi, Giuseppe Cacciatore, Dzmitry Kliazovich, Ulrich Sorger, Pascal Bouvry, Burak Kantarci, Fabrizio Granelli, and Stefano Giordano. Crowdsensim: a simulation platform

for mobile crowdsensing in realistic urban environments. *IEEE Access*, 5:3490–3503, 2017.

[7] Raghu K Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE communications Magazine*, 49(11):32–39, 2011.

[8] Nicholas D Lane, Yohan Chon, Lin Zhou, Yongzhe Zhang, Fan Li, Dongwon Kim, Guanzhong Ding, Feng Zhao, and Hojung Cha. Piggyback crowdsensing (pcs) energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2013.

[9] Liang Liu, Wu Liu, Yu Zheng, Huadong Ma, and Cheng Zhang. Third-eye: A mobilephone-enabled crowdsensing system for air quality monitoring. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(1):1–26, 2018.

[10] Chong Luo, Feng Wu, Jun Sun, and Chang Wen Chen. Compressive data gathering for large-scale wireless sensor networks. In *Proceedings of the 15th annual international conference on Mobile computing and networking*, pages 145–156, 2009.

[11] Kamal Mehdi, Massinissa Lounis, Ahcène Bounceur, and Tahar Kechadi. Cupcarbon: A multi-agent and discrete event wireless sensor network design and simulation tool. In *7th International ICST Conference on Simulation Tools and Techniques, Lisbon, Portugal, 17-19 March 2014*, pages 126–131. Institute for Computer Science, Social Informatics and Telecommunications . . . , 2014.

[12] Prashanth Mohan, Venkata N Padmanabhan, and Ramachandran Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 323–336, 2008.

[13] Federico Montori, Luca Bedogni, and Luciano Bononi. A collaborative internet of things architecture for smart cities and environmental monitoring. *IEEE Internet of Things Journal*, 5(2):592–605, 2017.

[14] Federico Montori, Luca Bedogni, and Luciano Bononi. Distributed data collection control in opportunistic mobile crowdsensing. In *Proceedings of the 3rd Workshop on Experiences with the Design and Implementation of Smart Objects*, pages 19–24, 2017.

[15] Federico Montori, Emanuele Cortesi, Luca Bedogni, Andrea Capponi, Claudio Fiandrino, and Luciano Bononi. Crowdsensim 2.0: a stateful simulation platform for mobile crowdsensing in smart cities. In *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 289–296, 2019.

[16] Federico Montori, Prem Prakash Jayaraman, Ali Yavari, Alireza Hassani, and Dimitrios Georgakopoulos. The curse of sensing: Survey of techniques and challenges to cope with sparse and dense data in mobile crowd sensing for internet of things. *Pervasive and Mobile Computing*, 49:111–125, 2018.

[17] Dimitar Valentinov Pavlov. Hive: An extensible and scalable framework for mobile crowdsourcing. *Diss. Imperial College London*, 2013.

[18] Moo-Ryong Ra, Bin Liu, Tom F La Porta, and Ramesh Govindan. Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 337–350, 2012.

[19] Xiang Sheng, Jian Tang, and Weiyi Zhang. Energy-efficient collaborative sensing with mobile phones. In *2012 Proceedings IEEE INFOCOM*, pages 1916–1924. IEEE, 2012.

[20] Vijay Sivaraman, James Carrapetta, Ke Hu, and Blanca Gallego Luxan. Hazewatch: A participatory sensor system for monitoring air pollution in sydney. In *38th Annual IEEE Conference on Local Computer Networks-Workshops*, pages 56–64. IEEE, 2013.

[21] Cristian Tanas and Jordi Herrera-Joancomartí. Crowdsensing simulation using ns-3. In *International Workshop on Citizen in Sensor Networks*, pages 47–58. Springer, 2013.

[22] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *IEEE Internet of Things journal*, 1(1):22–32, 2014.

Ringraziamenti

Le prime persone a cui devo dire grazie sono i miei genitori, che mi hanno permesso di raggiungere questo traguardo. Senza mia madre e mio padre, non avrei avuto la possibilità di studiare fino ad arrivare a scrivere questo elaborato.
Un grazie a mia sorella e alle mie nonne, per avermi sempre incoraggiato fin dall'inizio del percorso universitario, e la mia gattina per la compagnia nei mesi di lavoro.
Un sentito ringraziamento al relatore Prof. Luciano Bononi, ed in modo particolare al mio correlatore Dott. Federico Montori per la sua infinita disponibilità e tempestività, per i suoi suggerimenti e la guida per la realizzazione della tesi.