ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

TECNICHE E ALGORITMI DI AGGREGATE COMPUTING A SUPPORTO DI CONTESTI DI SMART MOBILITY

Relazione finale in
INGEGNERIA DEI SISTEMI SOFTWARE ADATTATIVI COMPLESSI

Relatore
Prof. MIRKO VIROLI

Presentata da
FILIPPO BERLINI

Co-relatori
Prof. DARIO MAIO
Dott. ROBERTO CASADEI

Seconda Sessione di Laurea
Anno Accademico 2016 – 2017

PAROLE CHIAVE

Aggregate Programming

Smart Mobility

Distributed System

Anticipatory Vehicle Routing

Internet of Things

"Longe præstantius est præservare quam curare"
Bernardino Ramazzini

Indice

Introduzione

Il terzo millennio ha portato con sé importanti innovazioni tecnologiche nell'ambito dell'informatica che hanno causato modifiche sostanziali nel modo di vivere e di pensare delle persone. Al giorno d'oggi non vi è soggetto che esca di casa senza il proprio smartphone, questo ci porta ad essere sempre muniti di un dispositivo computazionale con un'elevata capacità di calcolo. La diffusione di questi device, in aggiunta alla crescita di dispositivi in rete, ha portato allo studio e lo sviluppo di sistemi basati sull'*Internet of Things* (IoT). Lo studio e lo sviluppo di tali sistemi offrono grandi potenzialità che non riescono però ad essere pienamente sfruttate a causa di un supporto tecnico non adeguato.

Un'altra conseguenza dello sviluppo e della diffusione dei device computazionali sono i sistemi di *Smart City* e gli studi ad essi associati. La *Smart City*, in tutti i suoi ambiti, è in fervente crescita e lo sviluppo e il progresso tecnologico le offrono sempre nuove potenzialità. Grazie ai finanziamenti delle istituzioni internazionali che supportano questo tipo di ricerca, il processo di evoluzione è continuo.

La progettazione di un sistema distribuito, come è di fatto l'IoT, è per sua natura complessa. Gli approcci utilizzati fin ora per lo sviluppo di questi sistemi si sono mostrati frequentemente inadeguati o poco efficaci. È proprio in questo ambito che si inserisce l'*Aggregate Programming*. Lo studio di questa tesi vuole mostrare infatti come l'*Aggregate Programming* offra un modo di interpretare e pensare in termini di sistemi distribuiti capace di sfruttare le potenzialità che questi sistemi offrono. Nella presente tesi si pone in particolare attenzione alla *Smart Mobility* con lo scopo di mostrare possibili soluzioni

algoritmiche nell'ambito della gestione del traffico e della viabilità utilizzando l'*Aggregate Programming*.

Il primo obiettivo di questo studio è quello di rendere evidente come l'*Aggregate Programming* permetta in primis di ragionare su tali sistemi con semplicità, spostando l'attenzione dal singolo dispositivo all'insieme di tutti i device visti come un unico sistema computazionale; in secondo luogo come lo sviluppo in questo paradigma sia caratterizzato da semplicità del codice e chiarezza, dovuta alla sua essenza dichiarativa; infine far comprendere come l'*Aggregate Computing* permetta alla sviluppatore di non occuparsi delle problematiche, ma demandarle al linguaggio stesso.

Il secondo obiettivo di questa tesi è quello di analizzare l'utilizzo di questo paradigma innovativo nell'ambito della *Smart Mobility*. Si vuole mostrare come all'interno di un sistema distribuito in contesti di mobilità intelligente l'*Aggregate Programming* offra nuove possibilità nella gestione del traffico. In particolare si vanno a studiare e proporre algoritmi anticipativi nella valutazione del *Shortest Path* sviluppati utilizzando l'*Aggregate Programming*. Lo scopo è quindi quello di mostrare soluzioni semplici e chiare, che permettano di gestire un problema annoso, come l'anticipazione e la prevenzione, che fino ad ora non ha trovato soluzioni solide da poter essere utilizzate da una larga utenza.

Questo studio inizierà da un'analisi del background: nel Capitolo 1 vi sarà lo studio e l'analisi dell'*Aggregate Programming*, partendo dal *Field Calculus* come base di questo paradigma, mostrando poi quali sono i costrutti principali e le API che l'*Aggregate Programming* può fornire. Si passerà poi allo studio del framework `scafi` per lo sviluppo di programmi aggregati, basato su Scala, mostrando come vengono implementati i *building blocks* del paradigma preso in considerazione. Infine, per quanto riguarda lo stato dell'arte, verrà analizzato l'ambito della *Smart Mobility*. Nel Capitolo 2 si proseguirà poi con uno studio specifico del problema dell'anticipazione nell'ambito della gestione del traffico e della navigazione per i veicoli, mostrando le funzionalità già fornite

dai sistemi attuali e le problematiche da affrontare per fornire servizi anticipativi. Dopo questa analisi si mostreranno gli algoritmi anticipativi studiati e proposti sfruttando l'*Aggregate Programming*: nel Capitolo 3 si mostrerà un'implementazione dell'algoritmo proposto in letteratura, ma mai implementato, denominato *Anticipative Gradient*; nel Capitolo 4 si analizzerà un nuovo algoritmo, il *Filtered Gradient*, andando a mostrare come meglio risponda al requisito dell'anticipazione rispetto all'*Anticipative Gradient* attraverso la sua implementazione in scafi. Infine nel Capitolo 5 verrà mostrato lo studio di un caso pratico, analizzandone requisiti e mostrandone l'architettura, mettendo in luce le potenzialità del *Filtered Gradient* in un caso di mobilità urbana.

Capitolo 1

Stato dell'Arte

1.1 Aggregate Programming

1.1.1 Sistemi Distribuiti

Negli ultimi decenni abbiamo potuto osservare una crescita esponenziale nella diffusione di device computazionali. Tanti sono i fattori che hanno portato a questo, a partire dall'abbassamento dei costi di produzione alla possibilità di produrre componenti di dimensioni sempre inferiori o ancora dalla diffusione e lo sviluppo di Internet alla diffusione del mobile computing (i soli device Android venduti nel 2013 hanno superato il miliardo, divenendo la piattaforma di computing più diffusa nel mondo).

Scenari

Queste condizioni hanno portato alla sviluppo e alla diffusione dei sistemi distribuiti con numeri e varietà di oggetti sempre più in aumento. Vari possono essere gli esempi di questi tipi di sistemi, per citarne alcuni:

- **Internet of Things (IoT):** fa riferimento all'estensione di Internet a oggetti di vario genere, come potrebbero essere oggetti di tutti i giorni, per esempio gli elettrodomestici, o ancora i computer di bordo nelle macchine, gli antifurto...

- **Pervasive Computing:** fa riferimento al concetto di computazione in ogni momento, ovunque, laddove sia necessaria, o possa creare vantaggio.
- **Wireless Sensor Network (WSN):** fa riferimento ai sensori e attuatori spazialmente distribuiti in rete tra loro per monitorare, ma anche processare le condizioni ambientali dove sono inseriti.
- **Smart Things, Ambient Intelligence:** fa riferimento all'incorporazione dell'*intelligence* nell'environment attraverso set di servizi in grado di percepire il contesto in cui sono inseriti.
- **Swarm Robotics:** sistemi composti da più robot che interagiscono e si coordinano tra loro e con l'ambiente.

Tratti comuni nei sistemi distribuiti

Caratteristiche comuni che devono avere questi sistemi:

- hanno a che fare con un numero elevato di device che interagiscono tra loro, e con l'ambiente circostante
- **Context sensitiveness/awerness:** i servizi devono fornire risposte che tengano conto del contesto in cui sono inseriti.
- **Global to local correlation:** i servizi devono essere in grado di fornire informazioni di sintesi sul sistema in particolari aree o a specifici device.
- **Collective behavior:** non deve essere possibile che pochi device facenti parte il sistema distribuito portino a termine un task, ovvero il grande numero di device, oltre poter servire per precisione ed efficienza, deve essere necessario per il servizio stesso.

Problematiche principali

La progettazione e costruzione di un sistema distribuito è complessa per sua natura, in quanto bisogna far fronte a tante problematiche: la consistenza, la rindondanza dei dati, possibili punti di *failure* e la conseguente *recovery*, la comunicazione e la sincronizzazione, la manutenibilità e la modularità, ma

anche i problemi relativi all'autonomia e all'eterogeneità. Infine un punto cardine è sicuramente quello della coordinazione.

Le tipologie sopra descritte portano con sé anche altre problematiche:

- **Unpredictability:** il sistema può dover far fronte alla randomicità e complessità di ciò, che non può essere in alcun modo predetto.
- **Network complexity:** il sistema può essere composto da un vasto numero di nodi, e può anche prevedere connessioni dinamiche tra essi.
- **Situatedness:** il sistema deve essere progettato tenendo in considerazione anche il luogo e l'ambiente in cui viene inserito.

1.1.2 Aggregate Programming

Tutte queste problematiche rendono l'approcio tradizionale inadeguato o non efficacie. L'Aggregate Programming fornisce un'alternativa che semplifica la progettazione, la creazione e la manutenzione di sistemi complessi. A differenza dell'approcio tradizionale, in cui l'unità fondamentale è il singolo device connesso con il mondo fisico e con gli altri device attraverso input ed output, nell'Aggregate Programming l'unità di base non è più il singolo device, ma una collezione di device che collaborano; in questo modo i dettagli relativi al singolo device sono superflui. Chiaramente non è possibile distaccarsi totalmente dal singolo device, il quale deve essere programmato, ma bisogna trovare un buon equilibrio tra un ragionamento bottom-up e top-down, tipico dell'Aggregate Computing [3].

Perciò l'idea di base è quella di poter dedurre il comportamento a livello locale, e quindi del device, da quello di più alto livello, ovvero quello globale. Questo è possibile attraverso un mapping da globale a locale. Per poter ottenere una tale astrazione è necessario che il programmatore, quando progetta o sviluppa un programma aggregato, deve tenere conto di due punti di vista [2]:

- **Local Viewpoint (device-centric):** fa riferimento alla computazione aggregata eseguita dal device singolo. Questa è la *vista* tradizionale,

ovvero dove il porgrammatore ragiona sulla computazione svolta da un device e come esso interagisce con gli altri elementi del sistema.

- **Global Viewpoint (aggregate view):** fa riferimento alla computazione svolta dal sistema concepito come un'unica unità. Quando il programmatore ragiona sul comportamento aggregato si concentra di più su cosa il sistema debba fare piuttosto che come debba farlo.

I limiti del punto di vista del singolo device nel dominio dei sistemi distribuiti hanno motivato gli studi e i lavori svolti sull'aggregate programming. In generale le strategie principali sono [3]:

- rendere le interazioni fra i device implicite (per esempio TOTA)
- comporre costruzioni geometriche e topologiche (per esempio Origami Shape Language)
- dividere la computazione in maniera automatica per le esecuzioni *cloud-style* (per esempio MapReduce)
- sintetizzare i dati proveninenti da regioni spazio temporali e inviarli come stream ad altre regioni (per esempio TinyDB)
- fornire costrutti generalizzabili per la computazione spazio temporale (per esempio Protelis)

Gli studi svolti su queste strategie, con i loro successi e i loro fallimenti, hanno portato ad alcune osservazioni sulla programmazione di sistemi situati su larga scala:

1 i meccanismi per la coordinazione devono essere nascosti *"under the hood"*, laddove non è richiesta al programmatore alcuna interazione

2 la composizione dei moduli deve essere semplice e trasparente

3 sottosistemi differenti necessitano di meccanismi diversi di coordinazione per regioni e tempi distinti.

L'Aggregate Programming mira ad affrontare queste problematiche con i seguenti 3 principi [3]:

- la *machine* che deve essere programmata è una regione dell'environment computazionale che astrae dai dettagli specifici
- il programma è specificato come manipolazione di *data constructs* con estensioni spaziali e temporali in tutta la regione
- queste manipolazioni sono eseguite dai device all'interno della regione, usando i meccanismi di coordinazione resilienti e le interazioni basate sulla prossimità

1.1.3 Field Calculus

Computational Field

Alla base dell'Aggregate Programming c'è il *Computational Field*. Generalizzando il concetto di campo in fisica, il campo computazionale (*field*) è una funzione che in un dato momento nel tempo, mappa ogni punto dello spazio, che nel caso dell'aggregate programming si tratta di un device o un nodo, con un oggetto computazionale che rappresenta il risultato della computazione su quel device. I *field* sono funzioni che possono, e tipicamente lo fanno, evolvere nel tempo.

Le operazioni sui campi computazionali generalmente prendono in input un *field* e in output restituiscono un *field*. Quindi l'idea è quella di esprimere il comportamento globale del sistema come una composizione funzionale di operatori che manipolano il campo computazionale [4], [3].

Operatori base del Field Calculus

I campi nel Field Calculus sono creati e manipolati attraverso i seguenti costrutti principali [4]:

- *Built-ins operators* - `b(e1,...,en)` applica la funzione `b` agli argomenti `e1,...,en`. Il *field* in output è quindi ottenuto dalla valutazione dell'operatore sul campo in ingresso; ovvero ogni device è mappato al risultato

ottenuto applicando l'operatore ai valori locali del campo in ingresso. Si tratta di funzioni *stateless* matematiche, logiche o algoritmiche, sensori o attuatori, o funzioni definite dall'utente o importate da librerie

- *Function definition and call* - `def f(x1,...,xn){eb}` la funzione `f` è definita con la lista di argomenti `x1,...,xn` e il suo body è `eb`.
  `f(e1,...,en)` la funzione `f` è applcata all'espressione che viene passata come input `e1,...,en`. La *function call expression* sarà equivalnte al body `eb` dopo la sostituzione dei parametri `x1,...,xn` con quelli passati in input, `e1,...,en`.

- *Time evolution* - `rep(x<-v){s1;..;sn}` definisce una variabile locale `x` con valore iniziale `v` che periodicamente aggiorna il valore sulla base del risultato del body `s1;...;sn`, definendo in questo modo un campo che evolve nel tempo.

- *Neighborhood values* - `nbr(s)` mappa ad ogni device un campo che consiste nei valori più recenti dei neighbor ottenuti come risultato di `s`. Questo implica una comunicazione tra ogni device e i suoi vicini ottenendo così come risultato un campo di campi.

- *Domain restriction* - `if(e){s1;...;sn} else {s1`$^{\texttt{i}}$`;...;sm`$^{\texttt{i}}$`}` divide la rete in due region: dove `e` è valutata a `true` viene eseguito `s1;...;sn` se no viene computato `s1`$^{\texttt{i}}$`;...;sm`$^{\texttt{i}}$. Le partizioni implicano che i *branch* creati siano incapsulati e non possano avere effetti fuori dal loro sottospazio.

Ognuno di questi costrutti può avere due tipi di interpretazioni: la prima è quella locale, *device-centric*, per cui ogni espressione rappresenta la computazione di un valore locale in un device in un dato istante di tempo; la seconda appartiene all'*aggregate-level*, ovvero un campo computazionale calcolato e valutato a livello globale.

Il field calculus è universale, supportando ogni computazione spazio-temporale causale e approssimabile. In più questi costrutti supportano la portabilità, l'indipendenza dall'infrastruttura e l'integrazione con servizi non aggregati.

Infatti l'aggregate programming può incorporare ogni device o infrastruttura che implementi i costrutti, includendo device eterogenei, con sensori, attuatori, capacità di connessione e di computazione differenti. E d'altro canto i servizi non aggregati possono essere integrati semplicemente importando le loro API nell'Aggregate Programming.

Building Blocks

Il livello successivo di astrazione nel framework di aggregate programming aggiunge *resilience*, identificando una collezione di *building-blocks operators*. Questo layer consiste di meccanismi di coordinazione *self-stabilizing*, è scalabile e conserva la proprietà di *resilience* nel momento in cui vengono composti tra loro i vari operatori.

Per *self-stabilisation* si intende la capacità, indipendentemente dallo stato attuale, di raggiungere uno stato stabile in un tempo finito. Ovvero il sistema deve essere in grado di ristabilire uno stato stabile, nel momento in cui ci sono perturbazioni di questo, prontamente. Nel caso dell'aggregate programming si fa riferimento alla capacità di reagire tempestivamente ai cambiamenti della *network structure* o dei valori di input. La *self-stabilisation* è provata per tutti i campi ottenuti dalla composizione funzionale di *fixed fields (sensors, values)* e da un processo di *spreading gradient-inspired* [4].

Si possono categorizzare i *building-blocks* in una serie di classi [4]:

- *Sensing* - questa *input-phase* consiste nell'acquisizione di informazioni, spesso associate alle strutture spazio-temporali del sistema. Questo tipo di *building-blocks* può essere modellato come funzioni che fanno query ai sensori per ottenere i valori.
- *Detection of situation of interest* - le informazioni raccolte devono essere analizzate per poter determinare il contesto e definire il goal del sistema. Tipicamente comprende la produzione di dati di sintesi (*aggregation*).
- *Moving information where needed* - l'informazione deve essere spostata o diffusa agli elementi responsabili dell'azione. Questo può implicare

l'identificazione di sottosistemi per limitare il flusso di informazioni a specifiche parti del sistema.

- *Acting based on context information* - in questa *output-phase* il sistema genera una risposta sulla base della situazione attuale. In particolare, la risposta tipicamente dipende sul risultato di qualche computazione, che potrebbe essere limitata in specifiche location del tessuto spazio temporale.

I *building-blocks* che sono stati individuati sono operatori di coordinazione che possono essere usati per tutti i pattern di coordinazione. In aggiunta ai built-ins e l'operatore `if` del field calculus ci sono i seguenti operatori [3], [4]:

- `G(source, init, metric, accumulate)` è un'operazione di *spreading* che svolge due task simultaneamente: il primo è quello di calcolare un campo di *shortest-path distances* da una regione sorgente `source`, la quale è un campo Boolean, usando la metrica `metric`. In secondo luogo propaga i valori sulla distanza del gradiente, iniziando dal valore `init` e accumulandolo sul gradiente attraverso la funzione `accumulate`.

- `C(potential, accumulate, local, null)` accumula i valori sulla sorgente `source` risalendo il gradiente del *field* `potential`. Il valore di partenza è `null` ed è combinato con gli altri valori utilizzando la funzione `accumulate`.

- `T(initial, floor, decay)` è un'operazione di *countdown* flessibile, con un rate temporale che può cambiare: la funzione decrementa il valore iniziale `initial` attraverso la funzione `decay` fino ad arrivare al valore `floor`.

Questi operatori sono già abbastanza generali per poter coprire, da soli o combinati tra loro, la maggior parte dei pattern di coordinazione usati nei sistemi a larga scala. Implementati in field calculus questi operatori forniscono un ambiente di programmazione espressivo che garantisce la *resilience* e la scalabilità.

Per rendere ancora più *user-friendly* la programmazione bisogna fornire API per un utilizzo più pragmatico, che mantengano, grazie all'utilizzo e alla composizione degli operatori di base, le stesse proprietà. Per esempio, molte azioni distribuite e funzioni di diffusione di informazione possono essere basate su G. Se si vuole calcolare la distanza da un campo sorgente basta prendere come valore iniziale 0, definire una metrica `nbrRange` delle distanze tra device e device si ha:

```
def distanceTo(source){
        G(source, 0, ()->{nbrRange}, (v)->{v+nbrRange})
}
```

1.2 Scafi

Scafi (Scala Fields) nasce come framework per l'aggregate programming. Prima di questo *toolchain*, a supporto dello sviluppo nell'ambito dell'aggregate programming, esisteva già Protelis [5], un DSL sviluppato su Xtext e *hosted* da Java, che si integra con il simulatore ispirato alla biochimica Alchemist. Scafi si pone la sfida di fornire un *toolchain* più integrato per lo sviluppo di *actual systems*.

Scafi è un framework che consiste di due parti principali [1]:

- *aggregate programming support*, un DSL *Scala-internal* che fornisce la sintassi e la semantica per i costrutti di base del *field calculus*. Attraverso questo supporto la computazione aggregata è espressa in maniera naturale e combinata con il codice Scala standard.
- *aggregate platform support* permette la configurazione e l'esecuzione di sistemi aggregati distribuiti.

La scelta di Scala come linguaggio *host* per la costruzione di una piattaforma di aggregate programming ha una ragione sia pragmatica che tecnica. Scala è un linguaggio moderno in grado di interoperare con Java che integra i paradigmi object-oriented e funzionale senza soluzione di continuità, ha un

type-system potente ed espressivo e offre features avanzate che supportano il design di librerie software di alto livello. In più Scala combina il vantaggio del type-checking statico con features che permettono di essere concisi a livello di produttività e di sintassi. Queste features permettono al *library designer* di progettare e implementare le API come se fossero dei DSL, e all'utente appaiano come *embedded-languages* [1].

1.2.1 Costrutti del Field Calculus

La sintassi e il *typing* delle primitive base del field calculus sono dichiarate nei metodi del *trait* `Constructs` come segue:

```
trait Constructs {
  def mid(): ID
  def nbr[A](expr: => A): A
  def rep[A](init: =>A)(fun: (A) => A): A
  def foldhood[A](init: => A)(aggr: (A, A) => A)
        (expr: => A): A
  def aggregate[A](f: => A): A
  def sense[A](name: LSNS): A
  def nbrvar[A](name: NSNS): A
}
```

L'interprete fornito da scafi implementa questo *trait*, che ha già al suo interno le API dell'aggregate programming.

Un programma aggregato in scafi consiste nella definizione di un set di funzioni e un body di dichiarazioni di variabili ed espressioni, tutte le quali possono essere un mix di costrutti core del *field calculus* e librerie esistenti Scala. Il risultato del programma è il *field* ottenuto dalla valutazione dell'ultima sua espressione. L'espressione più triviale è la valutazione di un valore costante, che può essere un booleano, un numero, una stringa. Per esempio il valore

```
"Hello␣World"
```

è interpretato come un campo costante che mantiene questo valore in ogni punto. Questo significa che il risultato della computazione locale per ogni device è la stringa `"Hello World"`.

Il costrutto `sense` è utilizzato per leggere il valore da un sensore locale. L'espressione

```
sense[Double]("temperature")
```

ottiene, in ogni sensore, un `Double` dal sensore di temperatura `temperature`, producendo un campo di letture di temperatura. Il *sensing* è importante perché è un meccanismo per estrarre input dall'ambiente e permette così comportamenti *context-sensitive*. L'accesso concreto al sensore è gestito a livello di piattaforma. Il programmatore può quindi assumere che l'esecuzione per un round includa la mappatura dal nome del sensore al valore di tale sensore, ottenuto dalla computazione del sensore stesso.

Il costrutto `rep`, come già descritto in 1.1.3, permette di costruire un campo che evolva nel tempo round by round. Un semplice esempio può essere quello di contare i round computazionali svolti da un device dall'inizio della computazione

```
rep(0){ _+1 }
```

Il primo argomento è il valore di partenza, che è 0, mentre il secondo è una funzione unaria, espressa con sintassi lambda, utilizzando l'*underscore* per denotare il parametro, che prende quindi in ingresso il valore calcolato il round precedente, incrementandolo di 1. La frequenza con cui ogni device computa un round può variare tra i vicini e tra agente e agente. Di conseguenza il *field* risultante cambia nel tempo da 0 in maniera eterogenea su spazio e tempo. Infatti il modello dell'aggregate computing assume generalmente una sincronia parziale, anche se in molti casi si può assumere pure la totale asincronia [1].

Il costrutto `nbr`, come già descritto in 1.1.3, permette l'interazione, ovvero la comunicazione bidirezionale, tra un device e i suoi vicini. Questi ultimi sono definiti da una relazione di vicinanza la quale spesso e volentieri è basata sulla

distanza euclidea. Il risultato dell'operazione `nbr` è una mappatura dai vicini alla valutazione dell'argomento da parte loro; sostanzialmente è la primitiva per l'osservazione dei vicini. Il costrutto `nbr` deve essere innestato dentro ad una operazione di `foldhood`, che permette sostanzialmente di ridurre la mappa ottenuta dalla valutazione di `expr` tra i vicini ad un singolo valore attraverso la funzione di aggregazione `aggr` che parte dal valore `init`. Operazioni che possono essere definite partendo da questa sono per esempio `minhood` o `sumhood`. Prendendo in considerazione i seguenti esempi:

```
//Counting number of neighbors
foldhood(0)(_+_){nbr(1)}

//Check if sensor "sns" is active in every neighbor
foldhood(false)(_&_){sense[Boolean]("sns")}
```

Il primo esempio produce un campo che mappa ad ogni device il numero di vicini presenti, mentre il secondo esempio è il campo che mappa ogni device ad un valore booleano che è true solo nel caso in cui tutti i vicini del device abbiano attivo il sensore `sns`.

Un altro costrutto di percezione e `nbrvar`. Questo operatore è utilizzato per interrogare un dato sensore dai vicini, restituendo una mappatura tra ogni vicino e il valore. Come `nbr` il risultato deve essere aggregato o ridotto attraverso una operazione di `foldhood`. Un tipico esempio di sensore ambientale è un sensore che stima la distanza per ogni nodo dai neighbors:

```
def nbrRange(): Double = nbrvar[Double](NBR_RANGE_NAME)

//Compute minimum distance of a neighbor
foldhood(Double.MaxValue)(min(_,_)){nbrRange()}
```

Spesso può essere utile regolare le interazioni possibili definendo *branch* differenti di computazione, dinamicamente assegnati a sottogruppi di device. Questa feature, che è stata introdotta in 1.1.3, è la *domain restriction* e in scafi è supportata dal costrutto `branch`, il quale divide il dominio dei device in due parti sulla base di una condizione booleana `cond`. Ognuna delle due parti

computa un differente *field* in completo isolamento. L'*execution engine* assicura che device appartenenti alla stessa partizione computino la stessa `branch` *expression*, e questo processo è detto *alignment*. Di conseguenza l'*engine* assicura anche che due device di partizioni separate computino senza interazioni tra essi. L'*alignment* tra due device è dinamico, e questo significa che ad ogni round può variare. Possiamo prendere come esempio un caso in cui vogliamo eseguire l'aggregate computation solo in un sotto gruppo della rete, mentre i restanti non partecipano.

```
branch(sense[Boolean]("flag")){
        Double.MaxValue
}{
        compute(..)
}
```

Scafi supporta una versione *higher-order* del field calculus la quale introduce la funzione first-class `aggregate`. Il costrutto `aggregate` è utilizzato per fare il *wrapping* del body di una funzione il quale necessita di lavorare su un campo a sé. Come esempio si può prendere in considerazione l'implementazione di `branch` che usa `aggregate`:

```
def branch[a](cond: => Boolean)(th: => A)(el: => A):A =
  mux(cond, () => aggregate{ th },
                () => aggregate{ el })()
```

dove `mux` è un multiplexer funzionale.

1.2.2 Dal Field Calculus ai Building Blocks

Le primitive del field calculus formano un set minimo per la computazione basata sui campi computazionali di sistemi distribuiti e situati. Sono operatori di basso livello, mentre sarebbe necessaria un'astrazione di più alto livello per rendere ancora più semplice il ragionamento e la progettazione di sistemi adattivi, collettivi e complessi. Questo è possibile grazie alla possibilità di comporre le primitive mantenendo le propietà del field calculus.

Building Blocks

Ora vediamo come sono stati implementati i building blocks discussi in 1.1.3:

- *Gradient-cast*

```
def G[V: Bounded](source: Boolean, field: V, acc: V => V,
    metric: => Double): V =
  rep((Double.MaxValue, field)) { case (dist, value) =>
    mux(source) { (0.0, field)  } {
      minHoodPlus {(nbr{dist}+metric, acc(nbr{value}))}
    }
  }._2
```

  Partendo dal nodo sorgente `source` in cui il valore è 0, propaga il valore della distanza minima, definita dallo *shortest path*, in ogni punto da esso alla sorgente. Questo viene calcolato ponendo a 0 la distanza da sé stesso, mentre per tutti gli altri nodi viene propagato un valore che mano a mano cresce, sulla base delle distanze tra nodi vicini. Ogni nodo somma la distanza minima che ha tra i suoi vicini garantendo così il percorso minimo fino a sé.
  Interesante è l'utilizzo del costrutto `case`, che mostra bene l'integrazione tra Scala e scafi, il quale in questo caso permette di usare i valori della tupla, che round dopo round si aggiornano, come variabili.

- *Converge-cast*

```
def C[V: Bounded](potential: V, acc: (V, V) => V, local: V,
    Null: V): V = {
  rep(local) { v =>
    acc(local, foldhood(Null)(acc) {
      mux(nbr(findParent(potential)) == mid()) {
        nbr(v)
      }{
        nbr(Null)
      }
    })
  }
}
```

Permette di accumulare sul nodo sorgente del gradiente `potential` un valore che parte da `local` e sulla base della funzione di aggregazione `acc` viene appunto accumulato. L'utilizzo della `findParent` permette sul gradiente di individuare chi sia il nodo che precede il device preso in esame sul potenziale propagato. `C` è l'opeartore duale di `G`.

- *Time-decay*

```
type Numeric[T] = scala.math.Numeric[T]
  val Numeric = scala.math.Numeric

def T[V](initial: V, floor: V, decay: V => V)
    (implicit ev: Numeric[V]): V = {
  rep(initial) { v => ev.min(initial, ev.max(floor, decay(v)))}
}
```

Partendo da un valore iniziale `initial` che ad ogni round decade in base alla funzione `decay`, valuta se il valore è arrivato al valore minimo `floor`, se è così restituisce quest'ultimo sennò il valore attuale dato dalla funzione `decay`. Interessante anche in questo caso è l'integrazione con Scala, nell'utilizzo di `implicit`, in modo tale che chi chiama `T` non necessiti di passare la seconda lista di parametri. Questo parametro in più attraverso il pattern TypeClass permette di fare la `min` sui valori che `T` deve restituire.

Developer Api

Partendo dai *building-blocks*, analizzati in 1.1.3, è possibile aumentare l'astrazione per dare la possibilità allo sviluppatore di poter ragionare più semplicemente su sistemi complessi ed avere API che permettano lo sviluppo di questi in maniera più semplice. Analizziamo ora come esempio il `channel`.

Da G a channel - Il goal di `channel` è quello di evidenziare il percorso da un'area sorgente ad un'area di destinazione. Cioè è un campo che mantiene a `true` i nodi del path che unisce l'area sorgente a quella di destinazione, e false in tutti gli altri. Possiamo ora vedere la sua implementazione in scafi che si

basa su operatori derivati di G. In primis l'operatore `broadcast` che diffonde un valore per tutto il gradiente.

```
def broadcast[V: Bounded](source: Boolean, field: V): V =
  G(source, field, v=>v,nbrRange())
```

Poi l'operatore `distanceTo` che calcola in ogni nodo la distanza dal nodo sorgente

```
def distanceTo(source: Boolean): Double =
  G(source, 0.0, _ + nbrRange(), nbrRange())
```

Per ultimo abbiamo bisogno di diffondere l'informazione di distanza tra una sorgente e una destinazione andando a definire l'operatore di `distanceBetween`

```
def distanceBetween(source:Boolean,target:Boolean):Double=
  broadcast(source, distanceTo(target))
```

In questo modo ci sono tutti gli operatori che servono per implementare l'operatore channel che quindi risulterà essere

```
def channel(source: Boolean, target: Boolean,
        width: Double): Boolean =
  distanceTo(source) + distanceTo(target) <=
        distanceBetween(source, target) + width
```

L'aggiunta di `width` definisce un margine di ampiezza al channel, aggiungendo al field anche quei nodi che fanno parte dei path che stanno dentro al margine `width`.

Sulla base delle Developer Api poi si va a scrivere il vero e proprio codice, e quindi si vanno a strutturare e a mettere in piedi sistemi distribuiti complessi nelle loro più svariate applicazioni.

1.3 Smart Mobility

1.3.1 Smart City

La *Smart City* è un ambito in grande sviluppo e fermento, fonte di spunti e di grande interesse per la ricerca. Questo è dovuto al fatto che l'obiettivo delle

Smart Cities è estremamente nobile e supportato anche dalle organizzazioni internazionali, per esempio l'UE [12], ma anche i governi statali e così via arrivando fino alle amministrazioni locali. Infatti l'obiettivo ultimo delle *Smart Cities* è quello di migliorare la qualità di vita dei cittadini [9], [6].

Il concetto di *Smart City* è un concetto in grande evoluzione, e questa evoluzione segue fortemente l'evoluzione anche della tecnologia. Non è semplice dargli una definizione univoca, ma possiamo vedere come sia evoluto e quali siano gli aspetti fondamentali di questa.

Ogni *Smart City* è a sé, con caratteristiche proprie, fortemente dipendenti dal luogo dov'è situata. Infatti in base alle necessità, alla cultura, alle politiche in atto, al tipo di tecnologie già in utilizzo e socialmente accettate, ogni città smart ha i suoi aspetti peculiari e singolari. Sì può però vedere come il concetto di *Smart City* venga da una serie di *topic* specifici in cui possiamo raggruppare anche il tipo di iniziativa presa nell'ambito delle città smart. Vari sono i siginificati che può assumere *Smart City* come in [12], ma analizziamo ora come vengano raggruppati in [6]:

- *Digital City* - fa riferimento ad un'utilizzo sostanzioso delle *Information and Comunication Tecnology* (ICT), mostrando come questo tipo di tecnologie forniscano un supporto alla progettazione e creazione delle città *intelligenti*.

- *Green City* - questa accezione si concentra invece di più sull'aspetto dell'ecologia, e quindi dello sviluppo sostenibile, dei bassi consumi, delle basse emissioni, e in generale fa riferimento a tutte quelle misure che servono per la salvaguardia dell'ambiente nell'ambito delle *Smart Cities*.

- *Knowledge City* - questo è l'ambito che più si avvicina alle istituzioni culturali, ovvero fa riferimento alle politiche di rafforzamento e valorizzazione delle informazioni e delle conoscenze disponibili all'interno della città.

L'ICT all'interno dell'ambito della *Smart City* in definitiva non è di certo un goal bensì una tecnologia importante, in alcuni casi anche fondamentale, utile al fine di elevare il livello qualitativo di vita delle città.

I 6 assi della Smart City

Abbiamo analizzato e visto quali sono i *topics* per quanto riguarda le *Smart Cities*. Gli ambiti di intervento in cui agisce si possono però raggruppare in 6 assi [12], [11]:

- *Smart Economy*
- *Smart Governance*
- *Smart Mobility*
- *Smart Environment*
- *Smart People*
- *Smart Living*

Come definito in [12] la *Smart City* è la città che è sviluppata e ben svolge tutti i 6 assi. Questi forniscono anche gli indicatori dello sviluppo della città per quanto riguarda l'ambito delle città intelligenti [11].

1.3.2 Smart Mobility

Definizione

La *Smart Mobility* è un'ambito della *Smart City* preponderante a livello di ricerca e di studi associati. Questo è dovuto al fatto che va ad impattare su vari aspetti che possono migliorare la qualità di vita nelle città e aumentare i benfit ai cittadini [6].

In letteratura si trovano due definizioni di *Smart Mobility*, come visto in [7] e in [6]:

- La prima definizione fa riferimento solo alla mobilità in quanto tale, indipendentemente dall'utilizzo dell'ICT. Infatti si parla di *Smart Mobility* con l'accezione di mobilità efficace ed efficiente. Si fa riferimento quindi a tutte quelle misure con basso utilizzo di tecnologie ICT che però vanno a migliorare la mobilità.

- La seconda definizione invece fa riferimento all'utilizzo delle tecnologie avanzate nell'ambito della mobilità, sia a livello di ICT, sia per quanto riguarda i mezzi di spostamento. Come si può notare questa seconda definizione ha come punto cardine l'utilizzo della tecnologia, prescindendo spesso anche da valutazioni sui costi e sulla sostenibilità.

È molto interessante la tassonomia che viene fornita in [6] per quanto riguarda i sistemi di *Smart Mobility*. Questa è indispensabile dal momento in cui il range di azioni in questo ambito è veramente vasto, e individuando degli aspetti chiave su cui classificare i vari interventi, si riesce ad organizzare in maniera sistematica le azioni possibili nell'ambito della mobilità.

Vengono individuati tre aspetti fondamentali:

- **gli attori della Smart Mobility** ovvero coloro che muovono il tipo di iniziativa che si vuole realizzare.

- **l'utilizzo dell'ICT** ovvero quanto sia fondamentale e in che misura vengano utilizzate questo tipo di tecnologie, come verrà approfondito poi in 1.3.2.

- **obiettivi e benifici** che l'azione di *Smart Mobility* prevede, come si analizzerà in 1.3.2

Per quanto riguarda gli attori che fanno parte di questo tipo di interventi, ne vengono definiti quattro:

- Organizzazioni e compagnie di trasporti pubblici

- Cittadini e compagnie private

- Le amministrazioni e le istituzioni pubbliche

- La combinazione di questi, andando a realizzare iniziative che integrino tutti questi attori

Con la definizione di questi attori si vanno così ad individuare anche gli ambiti di azione che saranno analizzati in 1.3.2

Obiettivi della Smart Mobilty

Come già accennato, la *Smart Mobility* comprende vari aspetti della *Smart City* e per questo comprende una gamma ampia di obiettivi, i quali sono però raggruppabili in 6 macro-obiettivi [6]:

- ridurre l'inquinamento
- ridurre le congestioni del traffico
- aumentare la sicurezza delle persone
- ridurre l'inquinamento del suono
- aumentare la velocità di trasferimento da un luogo all'altro
- ridurre i costi di trasferimento

Ambiti della Smart Mobilty

La *Smart Mobility* si sviluppa su quattro ambiti principali che ora si vanno ad analizzare [6]:

- **Mobilità privata e commerciale -** Comprende una gamma di interventi, realizzati da privati o compagnie, anche se spesso incentivati dal pubblico, che riguardano l'introduzione di veicoli con certe caratteristiche e tutte quelle azioni che hanno a che fare con le modalità di trasporto che influenzano il comportamento dei cittadini. Un esempio di questo tipo di interventi può essere il *Car/Bycicle Sharing*.
- **Infrastrutture e politiche a supporto della mobilità -** Comprende le iniziative che influenzano la mobilità urbana, come ad esempio l'aggiunta di piste ciclabili, ma fa anche riferimento a tutte quelle politiche

che possono cambiare il sistema: per esempio incentivi su macchine a basso impatto ambientale.

- **Mobilità pubblica -** Comprende tutte le iniziative che hanno a che fare con i trasporti pubblici. Dal migliorare i servizi ad esso legati, come possono essere il sistema di ticket, a migliorare l'impatto ambientale, come l'introduzione di veicoli elettrici.

- **Sistemi di trasporto intelligenti -** Per *Intelligent Transport System* (ITS) si intendono tutte quelle applicazioni che attraverso la raccolta, l'analisi e la computazione di dati hanno l'obiettivo di fornire servizi e funzionalità, nell'ambito della mobilità. Come si può facilmente comprendere è un ambito molto vasto, e soprattutto in estrema evoluzione grazie alle possibilità fornite dall'innovazione tecnologica.

Rapporto con l'ICT

Come anticipato all'inizio di 1.3.2, il rapporto tra *Smart Mobility* e ITC è molto forte. Ma nei primi studi fatti su questo ambito spesso il rapporto con la tecnologia sembrava quasi apparire come un obiettivo e non uno strumento fondamentale.

L'utilizzo di tecnologie più o meno avanzate ovviamente varia in base all'*aim* dell'iniziativa intrapresa. Per esempio se si riprendono gli ambiti discussi in 1.3.2, possiamo vedere come l'importanza dell'ITC sia assolutamente relativa alle azioni che si vogliono realizzare.

Per quanto riguarda la mobilità privata e commerciale l'impiego e la necessità di utilizzare tecnologie ICT è generalmente basso, in quanto la maggior parte delle iniziative non necessità di esse. Si prenda come esempio appunto il *Vehicle Sharing*.

Per quanto riguarda la mobilità pubblica invece l'utilizzo di questo tipo di tecnologie è molto variabile: si può andare da un basso utilizzo come può essere

l'impiego di motori elettrici, ad un elevato utilizzo come nel caso dei veicoli *driverless*.

Per quanto riguarda le infrastrutture e le politiche a supporto della mobilità si va dal basso al medio impiego: la chiusura del traffico in certe aree è a basso utilizzo, mentre l'introduzione di sensori per il rilevamento della velocità è a medio utilizzo.

Per quanto riguarda in ultimo l'ITS chiaramente l'utilizzo di tecnologie avanzate è assolutamente necessario, in quanto è ciò che permette lo sviluppo di tali interventi.

Analizzando il rapporto tra *Smart City* e ICT è interessante lo studio svolto in [9] riguardo alle *Virtual Social Network*. Già da tempo si parla di *Internet of Vehicles* (IoV) ovvero di quelle comunicazioni tra veicoli che fanno parte di una rete comune, e della raccolta di dati per fornire servizi che permettano sempre di più di raggiungere gli obiettivi della *Smart Mobility*. Questo è possibile sfruttando la capacità che ormai i veicoli hanno insita, per quanto riguarda sensoristica e computazione. L'idea delle VNS è quella di integrare l'IoV con i social network. Ovvero far sì che questi sistemi abilitino la comunicazione, per gli utenti che hanno *social relationship*, tra veicoli attraverso la *vehicle network*. In questo modo alla comunicazione *Vehicle to Vehicle* e a quella *Vehicle to Infrastracture* viene aggiunta quella diretta tra gli utenti.

Architettura per la Smart Mobility

Analizziamo ora un'ipotesi di architettura per le azioni di *Smart Mobility*. Hitachi nei suoi studi è arrivato alla definizione di 5 *layer* che vanno a formare l'architettura della *Smart Mobility* [10]. Questi *layer* vengono definiti come *five layers of transportation functions* e sono:

- *Transportation user experience layer* - questo è lo strato in cui gli utenti ricevono servizi di trasporto, di informazione, forniti dalle compagnie di trasporto. L'esempio banale è quello di essere portati da una certo luogo ad un altro.

- *Transportation services layer* - questo è lo strato in cui le compagnie forniscono servizi agli utenti.
- *Information collection layer* - questo è lo strato in cui vengono raccolte le informazioni; per esempio come gli utenti utilizzano i servizi.
- *Information and management layer* - questo è lo strato in cui vengono analizzate e gestite le informazioni raccolte, in modo tale che le compagnie possano fornire servizi sempre migliori.
- *Transportation company coordination layer* - questo è lo strato in cui vengono raccolte e analizzate le informazioni da tutte le compagnie, in modo tale da controllare e fornire un servizio di trasporto migliore, su tutto il sistema dei trasporti, e non della singola compagnia.

Smart Mobility e Cittadini

Come brevemente accennato già in precedenza, un aspetto fondamentale, che spesso è stato trascurato nelle progettazioni di nuovi sistemi, è il tessuto sociale. Fondamentale da questo punto di vista è prima di tutto lo studio degli utenti del proprio sistema, e di quali possano essere le loro attitudini nei confronti di questo [8]. Ma questo non va ad inficiare solo sulla *user friendliness* ma anche sui risultati attesi dal nostro sistema [7]. Di conseguenza non solo vanno valutate le reazioni degli utenti al sistema ma in più vanno valutate le possibili attività che nascono da questa reazione. Per esempio se si fa un sistema per la diminuzione delle congestioni, bisognerebbe tenere conto anche, dato un buon esito del proprio progetto, dell'aumentare dei veicoli vista la facilità di trasferimento. Questo implicherebbe nuove possibili congestioni.

Capitolo 2

Anticipazione nell'ambito del Vehicle Routing

2.1 Definizione e introduzione al problema

Per quanto riguarda la *Smart Mobility* un ambito che è fonte di grande interesse è la navigazione di un veicolo da un dato luogo ad un altro, ovvero il fornire ad un dato utente lo *shortest path* dati un punto di partenza e un punto di arrivo.

Il navigatore satellitare, *GPS navigation system*, è stato forse il primo strumento tecnologico che ha reso possibile fornire questo tipo di servizio ad una più vasta gamma di utenti. Ma con l'introduzione degli smartphone la navigazione GPS è divenuta una feature alla portata di tutti. Questo ha portato ad un raffinamento sempre maggiore sulla qualità del servizio. E questo processo è tutt'ora in grande fermento. L'utilizzo dello smartphone, oltre ad aver concesso a tutti di usufruire della navigazione GPS, ha dato la possibilità anche a coloro che forniscono il servizio di sfruttare le features che ha il device: sensori, connettività e interazioni con l'utente. L'utilizzo di questi supporti ha portato alla possibilità di registrare e campionare i tempi di percorrenza, di segnalare problematiche relative alla mappa o alla viabilità, di mantenere connessi tra loro i vari user e tanto altro ancora. Il risultato è stato di certo un'ottimizzazione del servizio, che ha portato gli utenti a fidarsi sempre di più

di queste tecnologie, e di conseguenza i fornitori hanno ottenuto un maggior bacino di utenza, il quale permette loro di poter migliorare ancora di più il campionamento di dati e di conseguenza la qualità generale del servizio stesso.

Lo scenario che è offerto dall'attuale diffusione degli smartphone è quello di un sistema distribuito di device con capacità computazionale elevata. Sfruttare questo sistema apre a possibilità ancora nuove.
La nuova frontiera nell'ambito della navigazione è l'anticipazione, ovvero sistemi che non solo reagiscano ad eventi, come incidenti, ma in più abbiano la possibilità di prevenire congestioni e anticipare eventi futuri nella valutazione dei percorsi più veloci. Non sono pochi gli studi a riguardo per esempio [13], [14], [15], [16].

Vi sono già applicazioni che in un qualche modo offrono servizi di prevenzione, ma non è una vera e propria anticipazione della situazione reale bensì una stima di quello che può accadere, ma questo verrà poi approfondito in 2.2.

2.2 Applicazioni Mainstream

Negli store per app dei vari sistemi operativi *mobile* sono presenti diverse applicazioni che svolgono la funzione di navigatore satellitare. Le due maggiormente diffuse e che prenderemo in esame sono *Google Maps* e *Waze*. La scelta è ricaduta su queste due app in quanto forniscono un servizio di prevenzione, una sorta di anticipazione, che si basa su un approccio che possiamo definire statistico e reattivo. Ma vediamo ora come svolgono questa azione:

- **Google Maps** è l'applicazione del *"gigante di MountainView"* per la navigazione satellitare. È stata forse una delle prime a diffondersi in quasi tutti gli smartphone, aiutata dal fatto che si trova già di base in quasi tutti gli smartphone Android. Questa applicazione per la acquisizione di dati rilevanti le condizioni del traffico utilizza una rete di sensori, in aggiunta ai dati degli smartphone, che grazie alla loro distribuzione permettono di avere una buona valutazione dello stato attuale delle congestioni. In questo modo fornisce la possibilità in *real-time* di vedere

quali siano nel proprio percorso i tratti di strada che hanno livelli di congestione media o elevata, differenziati dalla colorazione (Arancione e Rossa). In più i dati raccolti permettono di andare a modificare e accurare le stime dei tempi di percorrenza in quella data fascia oraria. Parliamo di fascia oraria in quanto, nel momento in cui viene fatta una richiesta di percorso, *Google Maps* sulla base della fascia oraria, e quindi sulle medie dei tempi di percorrenza delle macchine già passate su quel tratto in quel range temporale, restituisce il percorso stimato più breve [17]. Per questo motivo, come precedentemente accennato, non si può parlare di una vera e propria anticipazione, in quanto si basa su statistiche passate e non sulla situazione futura effettiva

- **Waze** è un esempio di applicazione nell'ambito delle VSN [9]. Infatti è la prima applicazione che si è diffusa, tra quelle che forniscono il servizio di navigazione, che integra i social network nella rete di veicoli. Ovvero l'utente può segnalare sul suo percorso eventi: postazioni di blocco, incidenti, code, oggetti sulla strada, autovelox e altro ancora. Le segnalazioni possono essere poi verificate e validate dagli altri utenti che passano su quel tratto, rendendo effettiva la segnalazione e provocando un cambiamento nella viabilità del sistema. Questo è il primo comportamento reattivo offerto dal sistema.
  In oltre, come nel caso di *Google Maps*, vengono registrati i dati di percorrenza di tutti coloro che hanno attiva l'applicazione, senza l'apporto di dati provenienti da sensori, e questi vengono registrati per aggiornare i tempi di percorrenza dei vari tratti di strada. Anche in questo caso sono presenti fasce orarie per la valutazione del tempo di percorrenza [18], [19].

2.3 Approccio basato sull'intenzione

Gli approcci che sono stati analizzati in 2.2, come già detto in precedenza, sono approcci statistici e reattivi. Ma un tipo di servizio con questo approccio non offre una stima che sia basata su dati reali attuali, ma si basa su dati storici.

L'idea che sta a fondamento dell'approccio basato sull'intenzione è quello che ogni utente che percorrerà un dato percorso dichiari la sua intenzione di farlo, e che quindi la stima delle possibili congestioni si basi sui veicoli che realmente passeranno in quel tratto. Questo tipo di approccio è per esempio in [16].

In generale l'idea è quella di predire eventi e valutarli nella stima dei tempi di percorrenza. Ad oggi gli studi fatti si basano su questa strategia chiaramente, ma variano nel modo in cui si predicono gli eventi e poi sul calcolo dei percorsi.

2.4 Problematiche relative all'anticipazione

Nei sistemi che vogliono offrire un servizio di navigazione che permetta l'anticipazione nella valutazione dei tempi di percorrenza le problematiche principali a cui bisogna far fronte sono sostanzialmente due: la stima del flusso di veicoli e gli utenti. Ora si vanno a esplicare nelle sezioni che seguono.

2.4.1 Stima Traffic Flow

Il flusso del traffico, per poter permettere di anticipare e prevenire congestioni, è un aspetto fondamentale. Valutare con una buona precisione quale sarà il flusso dei veicoli in un dato tratto di strada in una certa frazione temporale, permette di poter attivare azioni di prevenzione. Questa stima può essere fatta in vari modi, come per esempio la distribuzione di sensori che registrino la velocità dei veicoli che passano in quella data zona, oppure sfruttare i dati dei veicoli. Queste soluzioni restituiscono però solamente la situazione attuale e non danno informazioni su quale potrebbe essere invece la situazione futura. Un'alternativa può essere l'utilizzo di sistemi basati sull'intenzione, come quelli descritti in 2.3, i quali però necessitano di un'utenza quasi totale, ma questo verrà discusso in 2.4.2.

2.4.2 Utenti

Il rapporto con gli utenti, come visto in 1.3.2, è un aspetto cruciale. In questo caso sono due gli assi su cui influisce questo rapporto:

- il **numero di utenti** che utilizzano il sistema è cruciale. Come anticipato, per una buona predizione delle congestioni per esempio, è necessario che la totalità dei veicoli, o una percentuale particolarmente elevata di essi, utilizzi l'applicazione. Questo permette stime che siano più vicine alla realtà, in caso contrario l'approccio statistico visto in 2.2 può forse fornire risultati migliori.

- la **fiducia degli utenti** nei confronti del sistema è un altro aspetto basilare. Questo aspetto è fondamentale per l'efficacia dell'anticipazione. Prendiamo per esempio il caso in cui ad un utente venga cambiato il tragitto durante il viaggio. Se questo decide di non fidarsi del cambiamento, o di non avere voglia di cambiare il percorso, ma di seguire le prime indicazioni, le valutazioni del sistema rischiano di essere imprecise, e quindi di conseguenza il sistema può non essere efficace.

Nei prossimi capitoli verranno presentate due soluzioni algoritmiche per la gestione di scenari anticipativi attraverso l'utilizzo di Aggregate Computing in Scafi.

Capitolo 3

Anticipative Gradient

In questo capitolo si analizza una delle due soluzioni algoritmiche proposte in questa tesi utilizzando *Aggregate Programming*. In particolare l'*Anticipative Gradient* è una soluzione già studiata in letteratura, [1], ma che non è mai stata implementata prima.

3.1 Motivazioni e analisi delle problematiche

Un applicazione base utilizzando il gradiente è quella di guidare colui che fa la richiesta dalla propria posizione alla posizione della sorgente del gradiente. Questa richiesta può avvenire da parte di una macchina per quanto riguarda i sistemi di navigazione, di un agente autonomo all'interno di una rete per trovare informazioni, di una persona guidata in un ambiente complesso attraverso il proprio smartphone o di un informazione diretta ad un certo target.
L'adattamento del *gradient* ai cambienti è insito nel suo essere, per come è stato pensato e progettato, e tempestivo: per esempio nel caso ci sia un interruzione di una strada, potrebbe cambiare di conseguenza la topologia della network che la rappresenta, in modo tale che il gradiente sia in grado di ricalcolarsi in maniera automatica trovando un nuovo path che porti al target.

L'idea dell'anticipative gradient è invece qualcosa di diverso: ovvero è quella che il gradiente sia esteso aggiungendo informazioni e conoscenza su alcuni nodi riguardo a eventi che accadranno nel futuro e che modificheranno l'abilità di

un agente di passare per quel nodo. Si può in questo modo diffondere questa informazione e far si che i meccanismi di *self-organisation* facciano deviare il gradiente in alcune zone per anticipare l'effetto dell'evento futuro andando a percorrere una strada che è più lunga, ma in un tempo inferiore, perché si è evitato l'ostacolo [1].

L'obiettivo è quello di progettare una soluzione non specifica per un dato caso di studio, ma che possa essere generalizzata come il gradient. Per questo si assume che l'ambiente sia coperto da una rete di nodi computazionali i quali danno forma ad una topologia che riflette la struttura dell'ambiente. In seconda battuta, che in ogni nodo sia presente un modo per determinare la distanza dai vicini, che potrebbe essere anche espressa in maniera temporale, ovvero come tempo di percorrenza, o unità temporali che si distaccano dal tempo fisico. In alcuni nodi poi ci deve essere qualche componente che permetta di identificare gli eventi futuri, ovvero cosa succede e quando. Per come è stato inizialmente ideato l'*Anticipative Gradient* è ideato con ostacoli che non possono essere attraversati.

3.2 Strutture Spaziali

In questa sezione verranno enunciati quelli che sono i componenti fondamentali dell'*Anticipative Gradient*, che poi saranno spiegati e illustrati nel dettaglio in 3.3:

- *Horizon Wave:* Questo componente è colui che si occupa di propagare l'informazione di un evento futuro in una data area del sistema, in un certo intervallo temporale.

- *Gradient Shadow:* Questo componente identifica quale area sia interessata ad un certo evento futuro, a partire da un punto di destinazione.

- *Future Event Warning:* Questo componente identifica l'intersezione tra *Gradient Shadow* e *Horizon Wave*.

- *Anticipative Gradient:* Questo è il componente fondamentale, che sulla base dell'area identificata dal *Future Event Warning*, calcola un semplice gradiente al di fuori dell'area, mentre all'interno valuta se sia più conveniente evitare l'ostacolo e quindi deviare o attendere che l'evento termini.

3.3 Implementazione in scafi

Fino ad oggi, per quanto ne sappiamo, questo tipo di gradiente non è stato mai implementato, ma solo simulato per verificare la correttezza delle assunzioni fatte e del suo comportamento.

L'implementazione è stata fatta utilizzando scafi e di seguito vengono riportate le strutture discusse in 3.2 con la spiegazione della loro implementazione.

- *Horizon Wave* Il primo componente è l'avvertimento di un avvenimento futuro (FE). L'area di FE è il set di nodi che ospiteranno questo evento. Non c'è bisogno che questo tipo di avvertimento comprenda tutti i nodi, ma basta che raggiunga solo i nodi che sono compresi nel FE, ovvero quelli per cui passandoci attraverso ci si scontrerebbe effettivamente con l'evento futuro.
  L'*Horizon Wave* definisce una corona circolare che man mano che il tempo passa collassa verso l'area FE, che è l'area dell'ostacolo, fino a scomparire totalmente.
  Si può ottenere questa struttura utilizzando il gradiente. L'area FE è la sorgente e propaga informazioni sul tempo in cui inizierà questo evento futuro e il tempo in cui finirà. Un nodo della rete sarà dentro la Wave se la sua distanza, intesa come distanza temporale, è compresa nell'intervallo tra il tempo di inizio dell'evento futuro e il tempo di fine di questo.

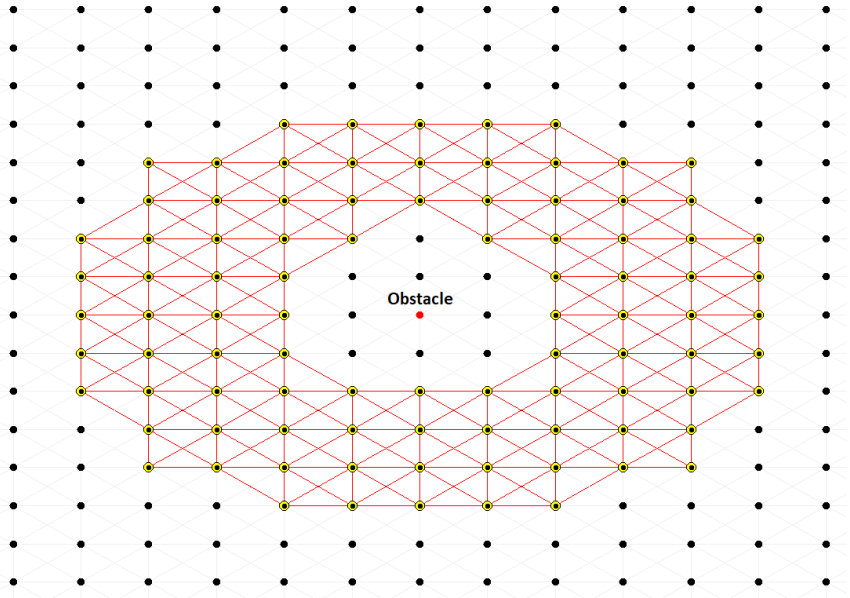

Figura 3.1: Horizon wave con tE 30s e tS 10s

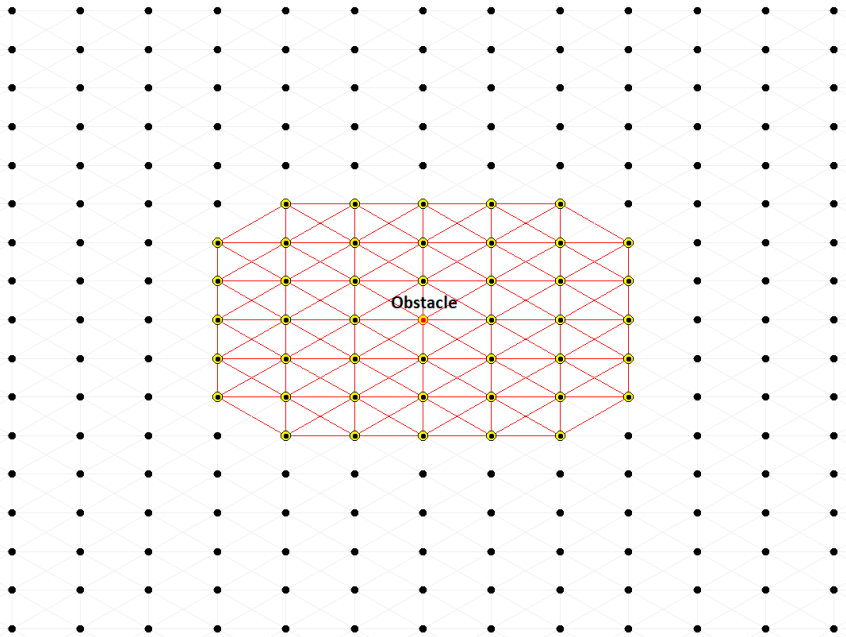

Figura 3.2: Horizon wave con tE 20s e tS 0s

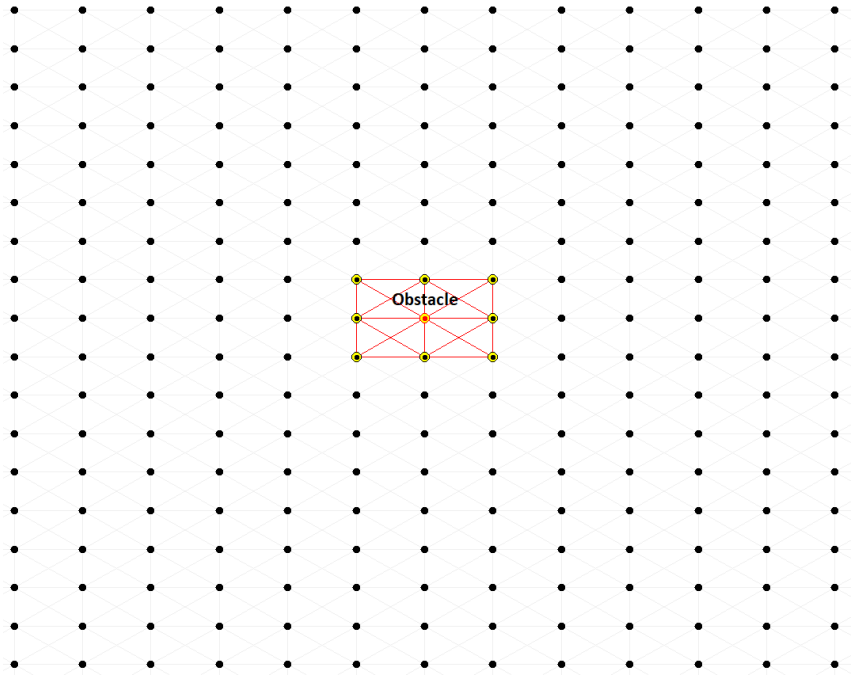

Figura 3.3: Horizon wave con tE 10s e tS 0s

```
def horizonWave (obstacle: Boolean, tS: Double, tE: Double):
    Boolean = {
  val distanceObst = distanceTo(obstacle)
  distanceObst >= tS & distanceObst <= tE
}
```

La struttura *Wave* è un campo booleano che localmente restituisce un valore dato dalla valutazione della propria distanza dall'ostacolo `obstacle`, e quindi vi è un gradiente propagato dall'ostacolo, rispetto ai tempi di inizio `tS` e di fine dell'evento `tE`. La valutazione restituisce true solo se il valore di distanza è nell'intervallo critico. Bisogna puntualizzare che la distanza viene considerata come distanza temporale, ovvero come tempo di percorrenza, andando a ridefinire la metrica `nbrRange()` sulla base di un sensore che sia in grado di stimare il tempo di percorrenza nei confronti dei vicini.

- *Gradient Shadow* Per sapere se è necessario ad un'agente tenere conto di un certo evento futuro è fondamentale anche sapere qual sia la dire-

zione e la destinazione del suo incedere, prendendo in considerazione il gradiente propagato dalla destinazione verso la quale l'utente si sta dirigendo. L'idea è quella di identificare la porzione di gradiente dalla quale l'utente muovendosi verso la destinazione passa attraverso l'area FE. Per questo viene chiamata Shadow, in quanto è sostanzialmente l'ombra che si genera nel gradiente nel momento in cui incontra l'evento futuro.
La strategia di base è quella di aggiungere dei tag al gradiente nel momento in cui attraversa l'area FE. Più specificatamente questo è ottenuto tenendo due annotazioni sullo stesso nodo: quella del gradiente propagato dalla destinazione e quella dell'evento futuro. L'effetto è quello di aggiungere l'annotazione dell'evento futuro.
La *Gradient Shadow* è una struttura statica nel tempo, a differenza dell'*Horizon Wave*, in quanto non dipende da nessun fattore temporale. Chiaramente mantiene la proprietà di *self-stabilisation* per cui si adatta ai cambiamenti della topologia, mantenendo quindi aspetti dinamici, ma dovuti all'adattività del sistema.

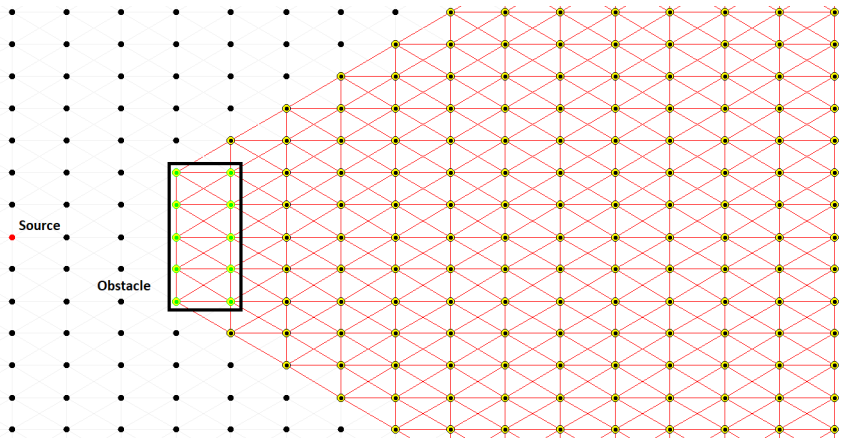

Figura 3.4: Gradient Shadow

```
def gradientShadow (target: Boolean, obstacle: Boolean): Boolean =
  G[Double](target, 0, x=> if(obstacle) x+1 else x , nbrRange()) > 0
```

La *Shadow* è un campo booleano valutato sulla base di un gradiente. Ha un implementazione quasi banale basata esclusivamente su `G`. Infatti

viene propagato un gradiente dalla sorgente `target`, che identifica la destinazione dell'utente che la richiede. L'opearazione che viene fatta è un operazione di *forecast*, ovvero un gradiente che mantiene un valore fintanto che non avviene una certa condizione. In questo caso la condizione è sostanzialmente se il nodo valutato in un dato momento è `obstacle`. Se così è, allora il gradiente da lì in poi cambia valore, in particolare qui viene incrementato, e si valuta se il valore locale sia ¿ 0. Dove è ¿ 0 il campo è `true`.

- *Future Event Warning* Si tratta sostanzialmente dell'intersezione delle due strutture appena descritte. Questa è la zona in cui il gradiente dovrebbe valutare se deviare per anticipare l'evento futuro. Dato quindi un punto di interesse, una destinazione, e almeno un area FE, l'intersezione delle due identifica il *Warning*.

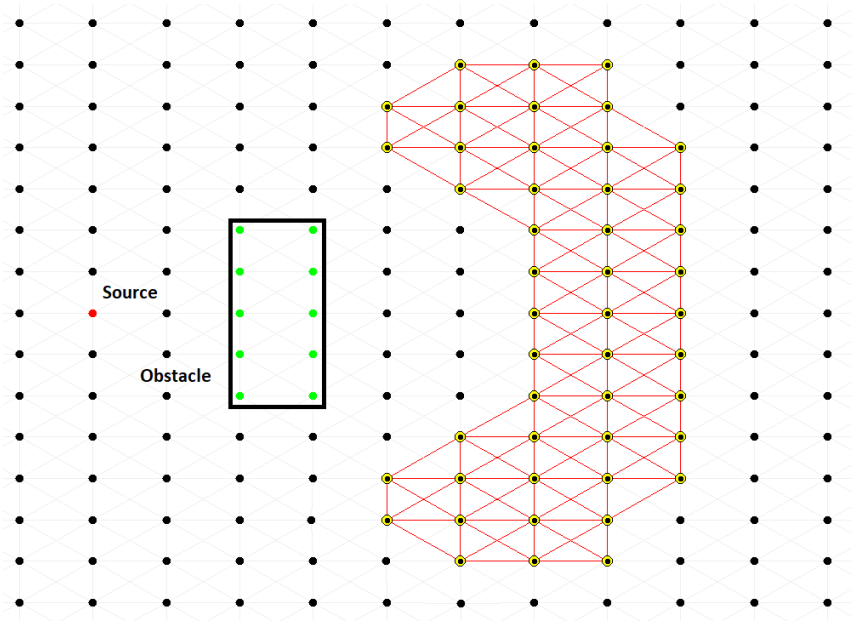

Figura 3.5: Future Event Warning

```
def futureEventWarning (target: Boolean, obs: Boolean,
    tS: Double, tE: Double): Boolean =
  rep(false)(_ =>
    horizonWave(obs, tS, tE) & gradientShadow(target, obs))
```

Il *Warning* è anche esso un campo booleano che è ottenuto dalla valutazione sia della *Wave* che della *Shadow*. E localmente restituisce `true` solo dove entrambi i campi computazionali valutati restituiscono `true`, ovvero fa l'intersezione tra i due campi.

- *Anticipative Gradient* Ora possiamo definire il pattern *self-organising* che si ottiene dalla composizione di quelli appena descritti che è l'*Anticipative Gradient*. Partendo da un punto di interesse da cui si propaga un gradiente e almeno un evento futuro, questo è identico ad un gradiente standard fuori dal *Warning*. All'interno del Warning invece la distanza e la direzone per ogni nodo sono il risultato della valutazione di due differenti alternative: passare attraverso l'evento futuro o circumnavigarlo. Questa valutazione si basa tutta sul pre-calcolo dei tempi di percorrenza delle due possibilità. Nel caso in cui si valuti il passaggio attraverso l'area dell'evento futuro si tiene conto del tempo di attesa prima che finisca questo evento. Per esempio si potrebbe attendere nel punto di partenza, fino a quando non si finisce fuori dall'area di *Warning* e a quel punto allora si può procedere tranquillamente dritto. O sennò si può procedere fino all'area dell'evento futuro e una volta arrivati lì attendere che finisca. Nel caso il tempo di percorrenza ottenuto tenendo conto dell'attesa, o prima o dopo, della fine dell'evento futuro sia inferiore al tempo di percorrenza ottenuto circumnavigando l'ostacolo, allora il gradiente non si deforma, viceversa il gradiente cambia direzione.
  Quindi possiamo dire che l'*Anticipative Gradient* restituisca fuori dal *Warning* il valore del gradiente tradizionale, mentre all'interno dell'area critica restituisca un valore aumentato, o a causa dell'attesa o a causa della circumnavigazione, ma sempre garantendo il minimo tempo di percorrenza.

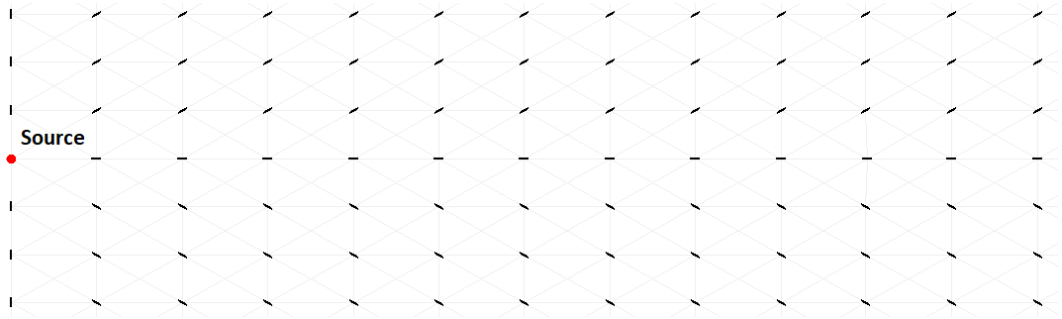

Figura 3.6: Standard Gradient orientation

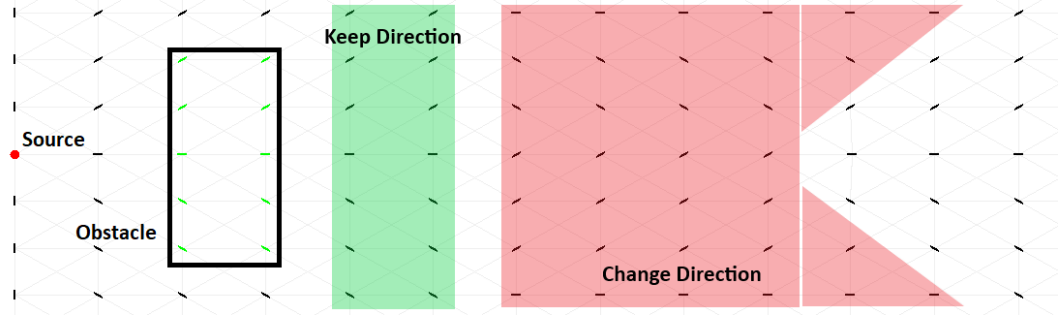

Figura 3.7: Anticipative Gradient orientation

Per potere definire la struttura fondamentale, ottenuta dalla composizione delle precedenti, bisonga prima definire altre due funzioni:

```
def functionStay (target: Boolean, obstacle: Boolean,
    tE: Double): Double =
  distanceTo(target)+ tE - distanceTo(obstacle)

def functionChange (target: Boolean, obstacle: Boolean): Double =
  branch(obstacle) {Double.MaxValue}{distanceTo(target)}
```

La `functionStay` è la funzione che valuta la distanza dal `target` attendendo la fine dell'evento, che è calcolata a partire dalla propria distanza dall'ostacolo `obstacle` sottratta al tempo di fine dell'evento `tE`. Questo valore è il tempo di attesa necessario per poter proseguire dritto.
La `functionChange` è la funzione che valuta la distanza dal `target` circumnavigando l'ostacolo `obstacle`. Questo è ottenuto separando le aree di eventi futuri dal resto della rete attraverso l'operatore `branch`.

A questo punto ci sono tutti gli ingredienti necessari per l'implementazione dell'*Anticipative Gradient*:

```
def anticipativeGradient (target: Boolean, obs: Boolean,
    tS: Double, tE: Double): Double = {
  mux(futureEventWarning(target, obs, tS, tE)) {
    functionStay(target, obs, tE) min
        functionChange(target, obs)
  }{
    (distanceTo(target))
  }
}
```

L'*Anticipative Gradient* sulla base della valutazione del *Warning* definisce che cosa restituiscono localmente i nodi. Nel caso un nodo non sia dentro al *Warning*, viene calcolato il gradiente tradizionale propagato da `target`. Altrimenti viene valutato quale delle due funzioni, `functionStay` e `functionChange` restituisca un valore inferiore, e viene valutato attraverso l'operatore `min` di Scala.

Capitolo 4

Filtered Gradient

In questo capitolo si analizza la seconda soluzione algoritmica proposta utilizzando sempre l'*Aggregate Programming*.

4.1 Motivazioni e analisi delle problematiche

Le motivazioni dietro all'ideazione e studio di questo nuovo approcio sono le stesse analizzate in 3.1 in aggiunta alle problematiche relative all'*Anticipative Gradient*, che verranno analizzate poi in 4.4.

L'idea di base, come per l'*Anticipative Gradient*, è quella di estendere il gradiente aggiungendo informazioni e conoscenza sui nodi che sono dentro all'area dell'ostacolo che avverrà nel futuro e che modificheranno la possibilità di passare attraverso di essi. In questo modo si vuole creare un gradiente che in base alla valutazione sugli ostacoli si modifichi anticipandoli. Il *gradient* in questo modo devia fornendo il path con tempo di percorrenza inferiore, anche se probabilmente più lungo a livello spaziale.

Va sottolineato il fatto che le distanze tra i vari punti devono essere sempre valutate come tempi di percorrenza e non distanze fisiche.

4.2 Componenti

In questa sezione vengono enunciati i componenti del *Filtered Gradient*, che poi saranno spiegati in dettaglio nella sezione 4.3:

- *Filter:* Questo componente identifica le aree dove saranno presenti ostacoli futuri, fornendo una mappatura che dato un nodo restituisca un range temporale.

- *Filtered Gradient:* Questo è il componente fondamentale, che sulla base degli ostacoli che si possono evincere dal filtro `filter`, definisce l'andamento del gradiente. In particolare se viene valutato che l'ostacolo è attivo il *gradient* cambia direzione.

4.3 Implementazione in scafi

L'implementazione del *Filtered Gradient* è stata fatta utilizzando il framework *scafi* per *Aggregate Programming*. Qui di seguito vengono mostrate le implementazioni e le descrizioni dei componenti anticipati in 4.2.

- *Filter* Questo componente è la segnalazione di un evento futuro. L'area critica su cui il filtro restituisce dei valori è data dai nodi su cui la viabilità sarà compromessa per la presenza di tale ostacolo.
  Il filtro è una struttura statica, che non varia nel tempo e nello spazio. Per ottenere questo tipo di componente sì può utilizzare una struttura dati che mappi da un determinato identificativo al range di tempo interessato. Questo significa che se su un nodo viene fatto un controllo per verificare se sia o meno all'interno dell'area dell'evento futuro, questo restituirà localmente o il range di tempo, se effetivamente è dentro quest'area, altrimenti nulla.
  Nell'immagine 4.1 si può notare come i tempi vengano considerati come assoluti, ovvero ad ogni ostacolo viene dato un range di ore minuti e secondi. In questo modo non è necessario nessuno strumento per garantire la sincronia temporale, come sarebbe necessario invece se fosse un range del tipo *"tra N minuti"*.

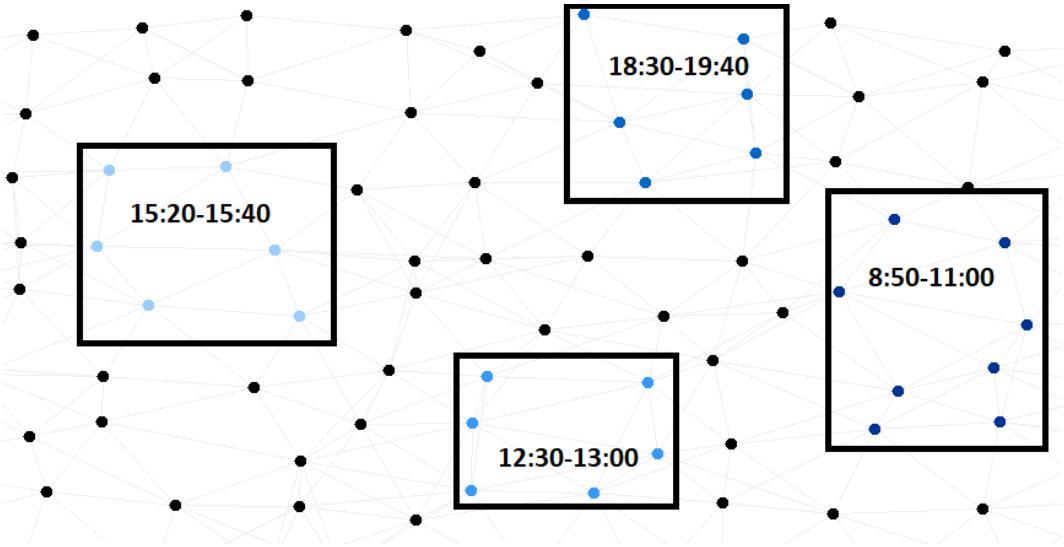

Figura 4.1: Ostacoli con differenti range orari

```
var filter = Map[Boolean,(Int,Int)]()
```

Il filtro in scafi non è altro che una `Map` di Scala, che dato un booleano, il quale identifica il sensore attivo per segnalare l'ostacolo, restituisca una tupla di interi, i quali identificano l'orario in secondi (`H*60*60 + M*60 + S`). Ciò che ne risulta è un *field* statico che permette di risalire alle frazioni temporali critiche per determinati ostacoli.
Nel caso il filtro mappi da ogni nodo ai range, come vedremo nel capitolo 5.4.3, allora l'implementazione sarebbe la seguente:

```
var filter = Map[ID,(Int,Int)]()
```

Da questo tipo di implementazione ogni nodo, localmente, potrebbe verificare se appartiene o meno al campo. `ID` è un valore di tipo `Int` ma identificativo e univoco per ogni nodo.

- *Filtered Gradient* Partendo dal filtro che permette di identificare dove e quando saranno gli ostacoli, si può allora strutturare il componente fondamentale, ovvero il *Filtered Gradient*. Questo tipo di gradiente viene propagato da una sorgente, in particolare il punto di partenza del tragitto richiesto. Ad ogni nodo, sulla base della propria distanza da esso, la quale

è espressa in termini di tempi di percorrenza, il gradiente valuta se il proprio passaggio è nel range orario restituito dal filtro su quel nodo. Se così è il gradiente aumenta in quel nodo il valore cambiando la viabilità attraverso quel nodo. Questo processo implica che tutti i nodi che erano nel gradiente dopo il nodo critico cambiano il loro tempo di percorrenza sulla base del nuovo tragitto più breve.

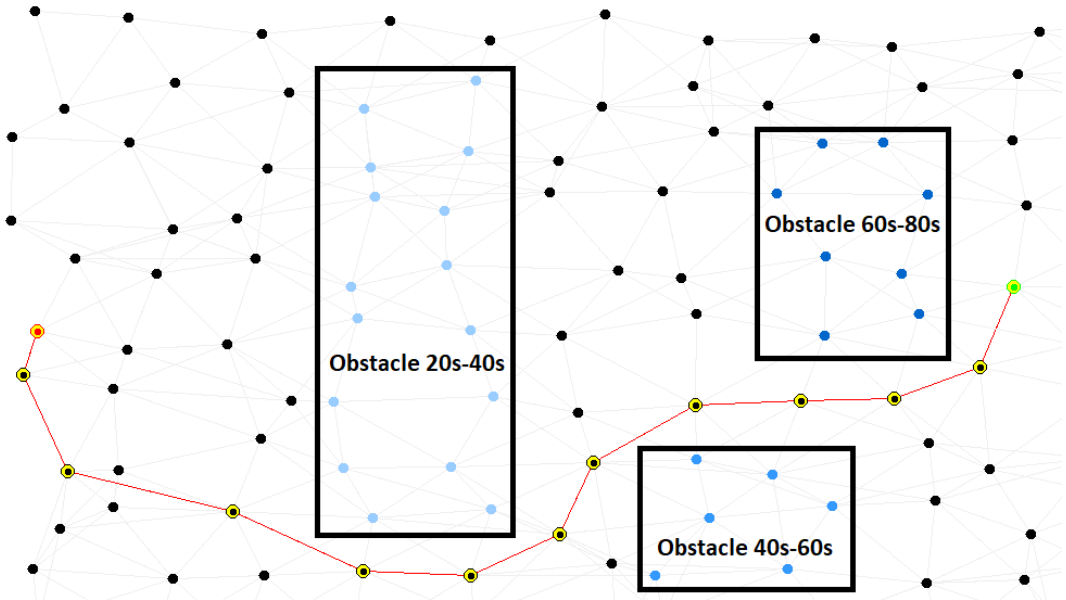

Figura 4.2: Esempio di shortest path con Filtered Gradient

- Per poter arrivare all'implementazione del *Filtered Gradient* bisogna prima definire dei componenti necessari ad esso:

```
def range(passingTimeRange: (Int,Int), distance: Int): Boolean  = {
  val currentTime = Calendar.getInstance().getTime()
  val minuteFormat = new SimpleDateFormat("mm")
  val hourFormat = new SimpleDateFormat("hh")
  val currentHour = hourFormat.format(currentTime).toInt
  val currentMinute = minuteFormat.format(currentTime).toInt
  val myPassingTime = distance + (currentMinute * 60)
        + (currentHour * 3600)
  return myPassingTime < passingTimeRange._1
        | myPassingTime > passingTimeRange._2
}
```

In primis è fondamentale definire un metodo che permetta di valutare se in base alla mia distanza dall'ostacolo io debba tenere conto di questo o meno. `range` ha proprio questa funzione, ed è un metodo di Scala puro. Prendendo in ingresso una tupla `passingTimeRange` che definisce il range temporale, la distanza `distance`, che altro non è il tempo di percorrenza fino al nodo esaminato, sommando la propria distanza all'ora attuale verifica se il tempo stimato è all'interno del range critico.

```
def filterDistanceTo(source: Boolean, filter:
        Map[Boolean, (Int,Int)], default: Double): Double =
  rep(Double.MaxValue) { case (dist) =>
    mux(source) {  0.0  } {
      val distance = minHoodPlus { nbr{dist} + nbrRange() }
      mux(range(filterEvaluation(filter),
        distance.toInt)){
        distance
      }{default}
    }
  }
```

Ora il codice per definire un comportamento che sia adattivo e anticipativo in Aggregate Computing risulta così estremamente semplice. In sostanza quello che è stato creato è una vera e propria rivisitazione di `G` visto in 1.1.3. Infatti oltre alla sorgente `source` da cui viene propagato il gradiente viene passato come parametro il filtro `filter` e un valore di default. Quello che cambia da un gradiente tradizionale è il controllo che viene fatto dentro al `mux`, ovvero il secondo `mux` innestato. Questo calcola il valore di distanza tra il nodo e il nodo precedente sul gradiente, e avendo come condizione il controllo se tale nodo sia in questo modo valutato come ostacolo, definisce come il gradiente si propaga, se cambiando direzione o meno. Ma questo senza la necessità di calcolare due possibili gradienti, bensì semplicemente variando il valore dei nodi nelle aree degli ostacoli ponendo il loro valore a `default`.
Due aspetti sono fondamentali da sottolineare: il primo è l'utilizzo di `filterEvaluation`, che è un metodo, che sia di Java o di Scala o anche in Aggregate Programming, che dato il filtro restituisce il range temporale.

Questo metodo non è stato definito a priori in quanto è proprio del caso specifico, in base al tipo di ostacoli che si vogliono modellare. In secondo luogo il valore di default, il quale permette di modellare non solo ostacoli insuperabili ma anche rallentamenti dilatando i tempi di percorrenza e vie preferenziali restringendoli.
Un altra operazione che rende effettivamente efficacie il *Filtered Gradient* è lo *Shortest Path*, che prima di questo studio non era mai stato implementato in scafi. Questa permette di creare un campo che è a `true` nello *Shortest Path* sul gradiente che gli viene passato come parametro.

```
def shortestPath(source: Boolean, gradient: Double): Boolean =
  rep(false)(path =>
        mux(source){
      true
    } {
      foldhood(false)(_|_) {
        nbr(path) & gradient == nbr(minHood(nbr(gradient)))
      }
        }
  )
```

4.4 Scenari

In questa sezione si andranno ad analizzare quelli che sono gli scenari critici ed interessanti per poter comprendere il funzionamento e validare il *Filtered Gradient* andando a vedere come si propone come risposta alle problematiche che emergono dall'utilizzo dell'*Anticipative Gradient*.

Gli ostacoli verranno rappresentati con una distanza temporale dal momento attuale, per poter meglio scorgere il variare delle valutazioni da parte del sistema.

Valutazione ostacolo

Il sistema ha come obiettivo quello di valutare se l'ostacolo sia o meno da considerare dato il proprio punto di partenza in un preciso istante temporale.

Gli scenari che possono avvenire sono i seguenti:

- **Singolo ostacolo impedente**

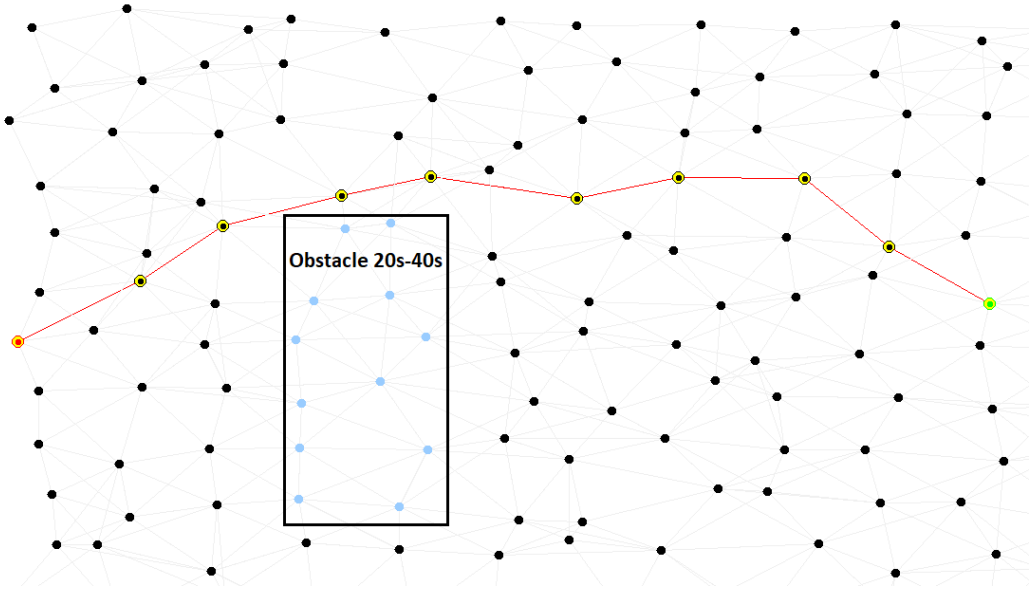

Figura 4.3: Ostacolo che impedisce il passaggio

In questo caso il sistema valuta, in base alla propria distanza, che l'ostacolo sarà presente al momento del passaggio variando così la direzione del gradiente standard.

- **Singolo ostacolo non impedente**

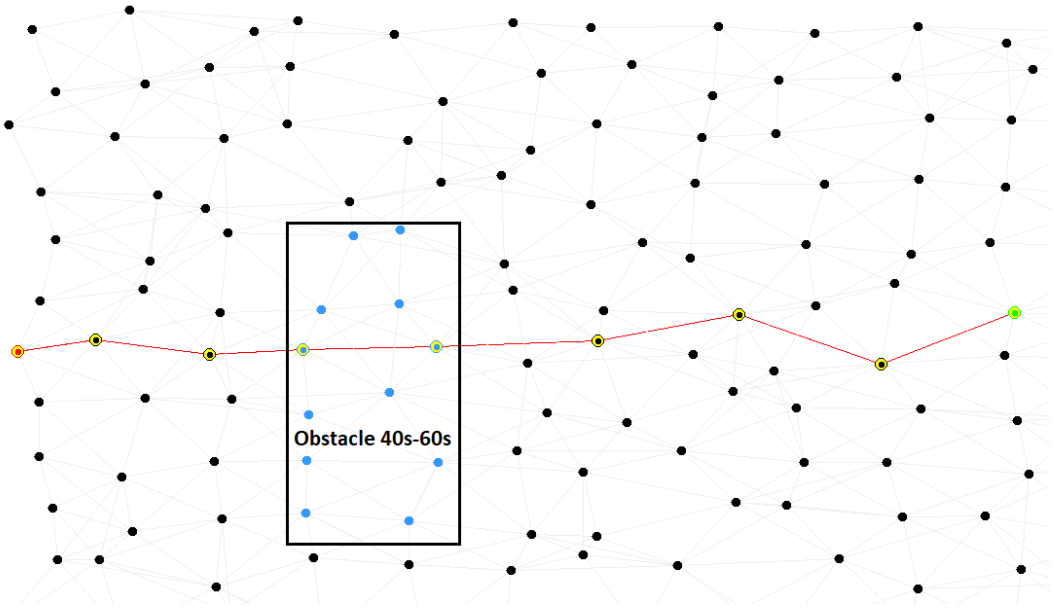

Figura 4.4: Ostacolo che non impedisce il passaggio

In questo caso, aumentando il range temporale, ovvero *"allontanandolo"* nel tempo, l'ostacolo al passaggio del veicolo non sarà ancora presente, per cui il sistema valuta di non cambiare direzione.

- **Doppio ostacolo**

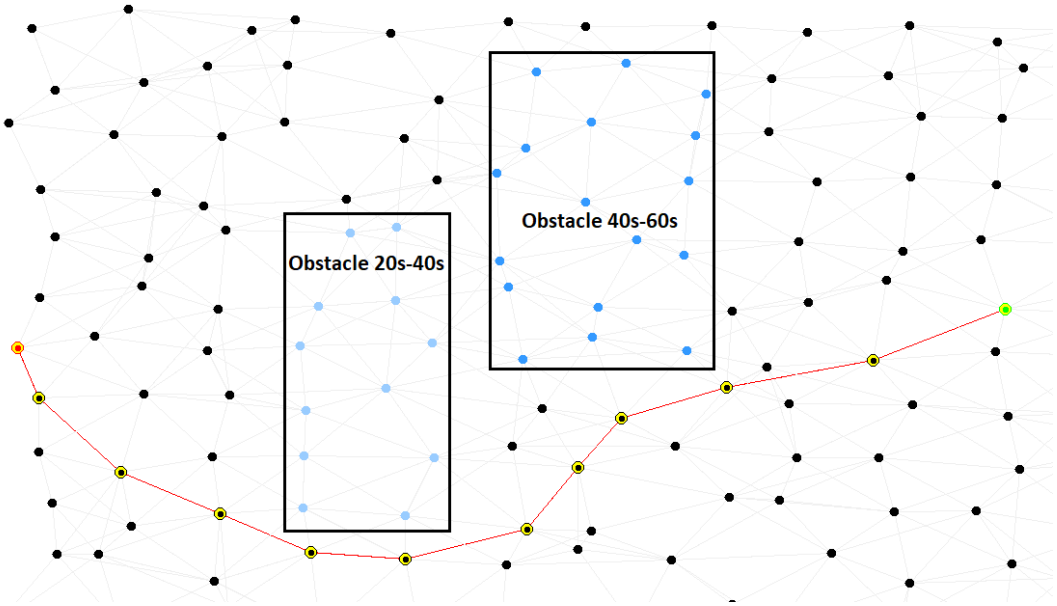

Figura 4.5: Doppio ostacolo

In questo scenario entrambi gli ostacoli vengono valutati come veri nel momento del passaggio del veicolo, quindi il sistema sceglie di cambiare direzione, trovando il nuovo path più breve.

Valutazione del medesimo ostacolo

In questo caso andiamo ad analizzare come ogni ostacolo venga valutato in base alla distanza minima che ha in quel preciso momento dalla sorgente. Questo può portare alla valutazione di uno stesso ostacolo, in uno stesso punto, in maniera differente. I due scenari possibili sono i seguenti:

- **Valutazione ostacolo impedente**

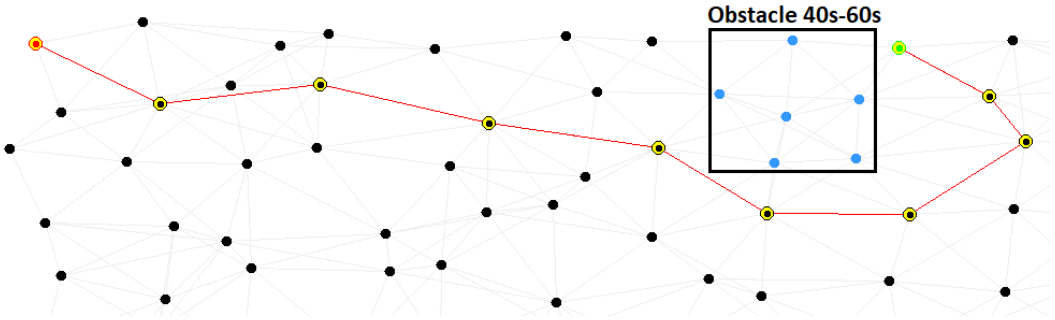

Figura 4.6: Valutazione ostacolo impedente

In questo caso lo scenario è quello di un ostacolo ad una certa distanza che viene valutato come vero nel momento del passaggio dell'utente.

- **Valutazione stesso ostacolo non impedente**

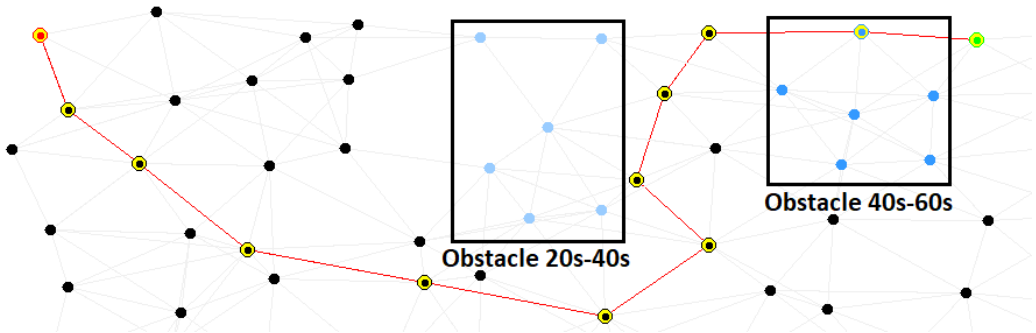

Figura 4.7: Valutazione stesso ostacolo non impedente

In questo scenario, avendo un altro ostacolo davanti, i tempi di percorrenza minimi per raggiungere tale ostacolo sono aumentati, e questo ha portato alla valutazione di questo come non impedente.

Valutazione ostacoli multipli

In questo scenario è forte il distacco dall'*Anticipative Gradient*. Quest'ultimo infatti ha una crescita esponenziale dei percorsi valutati all'aumentare degli ostacoli. Prendiamo per esempio il caso di un solo ostacolo valutato. Per fare questo i gradienti utilizzati sono due, e la valutazione avviene di conseguenza su due percorsi. Ma se aggiungiamo altri due ostacoli, e quindi altri

4 gradienti, per poter valutare il percorso è necessario valutare ogni abbinamento, e questo porta ad un numero di percorsi valutati pari a 2^n dove `n` è il numero degli ostacoli.

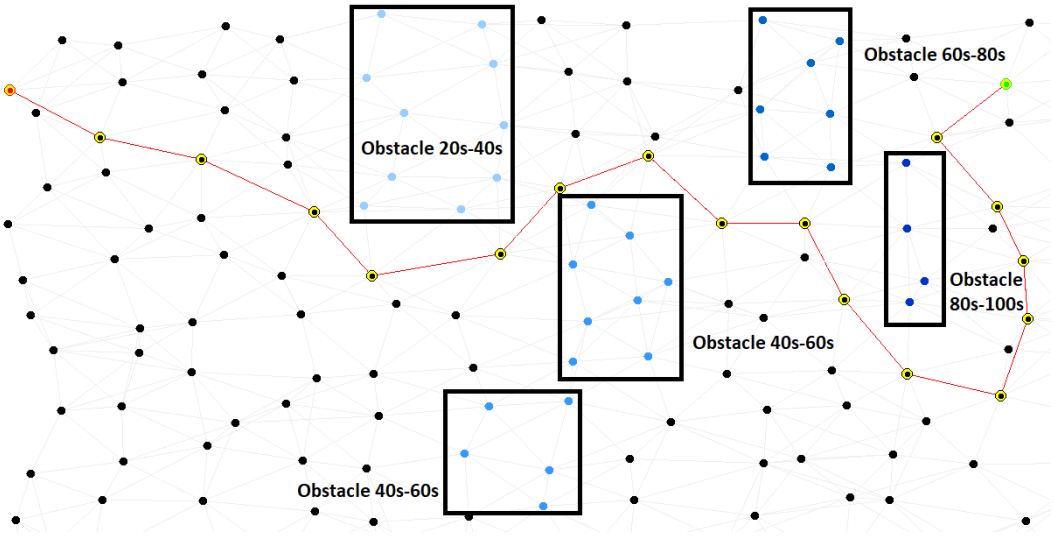

Figura 4.8: Multipli ostacoli

Il *Filtered Gradient* invece utilizzando un unico gradiente su un filtro, ha una crescita lineare sulla base del numero degli ostacoli, ovvero vengono valutati `n` range temporali su ogni nodo, dove `n` è il numero degli ostacoli. Questo vuol dire che per ogni nodo verrà valutato se la distanza dalla sorgente ad esso sia tale perché il passaggio avvenga nel momento di uno degli `n` ostacoli futuri.

Capitolo 5

Caso di studio

In questo capitolo si va a presentare il caso di studio sul quale è stato studiato e ipotizzato l'utilizzo dell'*Aggregate Computing* nell'ambito della *Smart Mobility*. Questo approfondimento è stato svolto insieme al Dott. Matteo Aldini, il quale ha analizzato nel dettaglio questo caso nella tesi "Progettazione e realizzazione di un sistema sicuro a supporto della mobilità dì veicoli in contesti urbani".

5.1 Descrizione caso di studio

Il caso che si vuole studiare è quello di un sistema di Smart Mobility per centri urbani prendendo in considerazione un particolare quartiere di Roma, in figura 5.1. Il sistema deve fornire servizi di navigazione all'interno del quartiere, mediante l'utilizzo di un sottosistema di sensori per il controllo del traffico, attraverso l'utilizzo di un sottosistema che si occupi del calcolo dello shortest path sfruttando le possibilità offerte dai sistemi distribuiti, il tutto appoggiato su un Server su Cloud.

5.2 Requisiti

Si vanno ora ad enunciare quali sono i requisiti fondamentali di questo sistema:

Figura 5.1: Quartiere preso come riferimento nello studio del caso

- **Navigazione:** il sistema deve fornire un sistema di navigazione per gli utenti, fornendo il percorso più breve dato un punto di partenza e un punto di arrivo.

- **Raccolta di dati:** i sensori devono essere in grado di raccogliere informazioni riguardanti al passaggio dei mezzi, ed in particolare ai tempi di percorrenza, per poter aggiornarli nei vari tratti del grafo, rendendo così il sistema adattivo.

- **Veicoli prioritari:** ci devono essere dei veicoli che devono poter ottenere

la priorità nel passaggio, ovvero devono poter percorrere i loro tragitti senza intralci. Questo però deve essere possibile mantenendo segreto il tratto di strada percorso agli altri utenti.

5.3 Architettura del sistema

In questa sezione si descrive la soluzione architetturale ideata per il sistema di Smart Mobility. La soluzione sfrutta i vantaggi sia delle architetture centralizzate sia di quelle distribuite, essendo di fatto una soluzione "ibrida" tra le due.

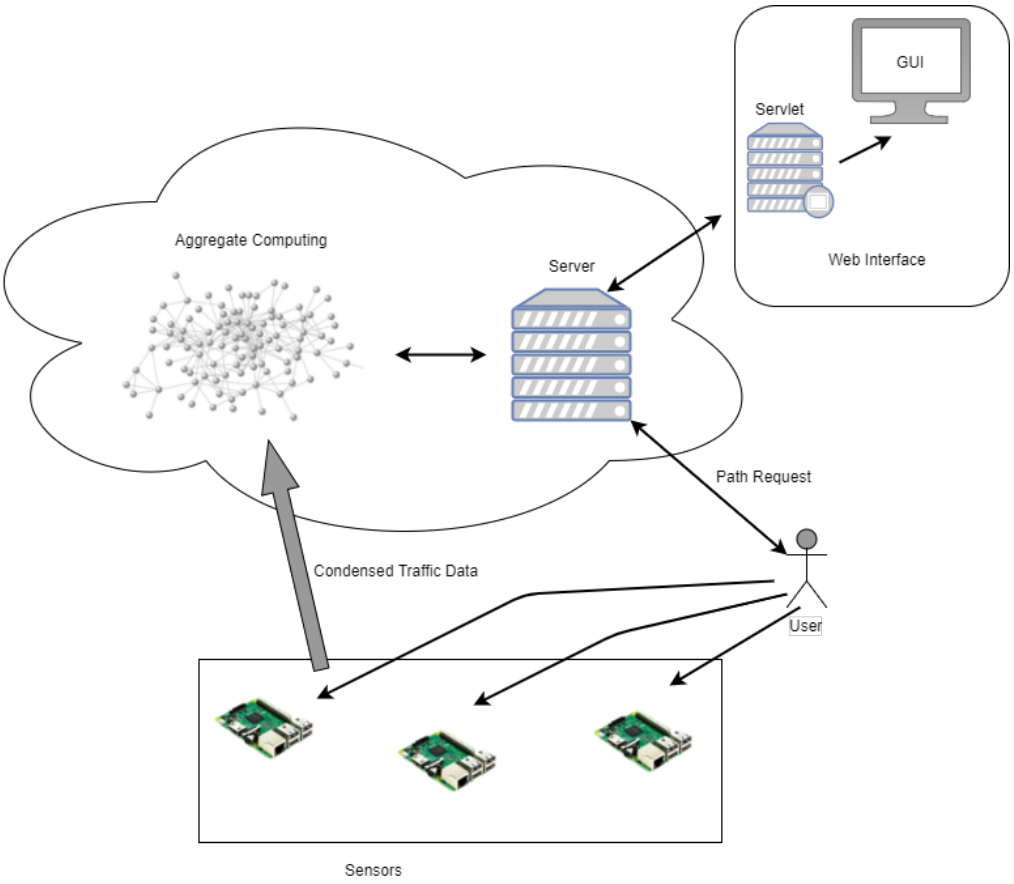

Figura 5.2: Architettura del sistema

Di seguito si descrivono in breve i componenti facenti parte dell'architettura proposta.

5.3.1 User

È l'utente del sistema, dotato di dispositivo mobile. Il suo obiettivo è quello di richiedere il percorso più breve dati un punto di partenza e uno di arrivo. Inoltre informa il sistema sui propri tempi di percorrenza.

Interazioni:

- **Server:** richiede un percorso e riceve una risposta. Questa interazione può essere ripetuta a intervalli regolari
- **Sensors:** invia i propri tempi di percorrenza ai device di competenza. Questo serve per l'aggiornamento dei tempi medi e per tenere traccia di quanti veicoli siano sul manto stradale

5.3.2 Server

Il suo obiettivo è quello di rispondere alle richieste di percorso da parte dell'utente, interagendo con il sottosistema aggregato. Inoltre fornisce a un'interfaccia Web i dati relativi alle navigazioni.

Interazioni:

- **User:** riceve una richiesta e restituisce il percorso più veloce
- **Aggregate Computing Subsystem:** invia il punto di arrivo e di partenza e riceve il percorso più veloce
- **Web Interface:** invia i percorsi che vengono restituiti agli utenti in modo da poterli visualizzare

5.3.3 Sottosistema di Aggregate computing

Si occupa del calcolo del percorso più veloce, propagando il gradiente a ogni richiesta. L'idea è che non venga mantenuto il gradiente attivo in continuazione, ma che sia attivato solamente alla richiesta del server. Per questo motivo ha senso che l'utente periodicamente rieffettui la richiesta. Riceve inoltre i dati sensoriali dall'infrastruttura fisica riguardanti le congestioni future.

Interazioni:

- **Server:** riceve un punto di partenza e uno di arrivo e restituisce il percorso più veloce, calcolato in base agli ostacoli presenti.
- **Sensors:** riceve i dati sul traffico, andando ad aggiornare i tempi di percorrenza

5.3.4 Sensors

I sensori si occupano di immagazzinare dati sul traffico in tempo reale, eventualmente salvandoli su database. Nel momento in cui avvengono variazioni sul traffico, vengono inviati i dati necessari al sottosistema di Aggregate Computing.

Interazioni:

- **User:** i sensori ricevono i tempi di percorrenza e il percorso scelto dagli utenti
- **Aggregate computing system:** i sensori inviano i dati sul traffico nel momento in cui ci sono variazioni

5.3.5 Interfaccia Web

Permette di visualizzare in tempo reale i percorsi restituiti dal sistema ed eventualmente la creazione di nuovi utenti.

Interazioni:

- **Server:** riceve i dati sui percorsi

5.3.6 Sequenza delle interazioni e funzionamento del sistema

L'utente richiede al Server un percorso attraverso il suo device dati punto di partenza e di arrivo. Il Server inoltra la richiesta al sottosistema di Aggregate

Computing, il quale si occupa della propagazione del gradiente e del calcolo del percorso. Il Server riceve il percorso e lo invia come risposta all'utente. Lo invia, inoltre, all'interfaccia Web per poterlo visualizzare. Il device utente, non appena ricevuto il percorso, notifica i sensori associati ai nodi di tale percorso. La procedura di richiesta del percorso viene ripetuta ciclicamente, in modo da poter moficare in tempo reale la navigazione. Durante il tragitto, il device utente invia i propri tempi di percorrenza ai sensori. Il sottosistema composto dai sensori invia dati sul traffico ogniqualvolta un nuovo veicolo si inserisca nella rete.

Chiarimenti

- I componenti Server e Aggregate Computing Subsystem sono su Cloud
- Si è scelto di separare la parte "fisica" dell'Aggregate Computing, ovvero la sensoristica, dalla computazione vera e propria, in modo da alleggerire il carico su Cloud e effettuare un preprocessing dei dati sul traffico, riducendo al minimo le interazioni.
- I sensori sono in realtà single-board computer dotati di scheda di rete. Tali dispositivi sono considerati concentratori, ovvero possono gestire le interazioni e le informazioni di più sensori "logici"
- Gli utenti che vengono creati tramite interfaccia Web sono in realtà creati dal Server come attori la cui computazione avviene sul Cloud
- Un'architettura di questo tipo è realizzabile attraverso l'utilizzo di microservizi, in modo tale da poter scalare orizzontalmente. Si identificano tre principali servizi: Server, Aggregate Computing e Web Interface.

5.4 Aggregate Computing

In questa sezione si va ad analizzare come viene sfruttato l'*Aggregate Programming* in questo sistema, come avvengono le interazioni con il sistema e di conseguenza il calcolo dei percorsi.

5.4.1 Ruolo nell'architettura

Come visto nella sezione 5.3 il ruolo dell'aggregate computing all'interno di questo sistema è sostanzialmente quello del calcolo del percorso più breve, tutto ciò inserito in un architettura Cloud distribuita. L'utilizzo dell'*Aggregate Computing* permette di sfruttare il sistema distribuito con una gestione semplificata, basandoci su un codice semplice e intuitivo alla lettura, grazie all'essenza dichiarativa di questo paradigma.

L'idea di base è che venga mantenuta una rappresentazione della rete stradale nel model su cui il programma aggregato computa. Attraverso poi i rilevamenti del sottosistema di sensori questo model può essere aggiornato. Questi aggiornamenti a livello di programma aggregato vengono valutati alla prima occorrenza, rendendo la valutazione del percorso adattiva e reattiva ai cambiamenti.

L'altro servizio che deve offrire l'*Aggregate Computing* è quello di gestioni dei veicoli prioritari, ma verrà approfondito nella sezione 5.4.3. Analizziamo anche nel dettaglio come avvengono le computazioni e ogni quanto ci sia necessità che avvengano le interazioni nella sezione 5.4.2

5.4.2 Bolle spazio-temporali

Le decisioni prese sulle interazioni tra il sottosistema di *Aggregate Computing* e il Server per il calcolo del percorso e sulla diffusione del calcolo di questi a livello spaziale tra i nodi, sono fondamentali per quanto riguarda il livello di performance.

Se il gradiente per il calcolo dei percorsi fosse sempre attivo, e fosse attivo su tutti i device, o nodi, i costi computazionali sarebbero elevati, e andrebbero a rallentare il sistema in maniera importante e sensibile, andando a aggravare il servizio fornito. Il concetto di *bolla spazio temporale* è basato sul circoscrivere sia temporalmente che spazialmente le computazioni, come in figura 5.3.

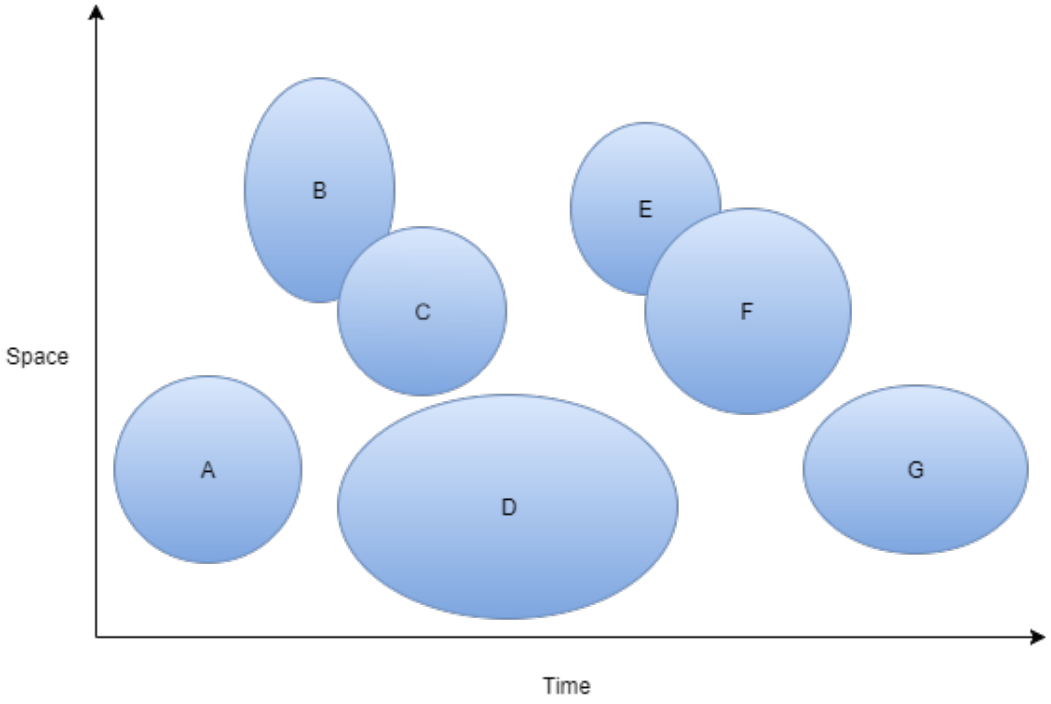

Figura 5.3: Bolle spazio-temporali

La computazione di una singola istanza non è continua nel tempo ma ha un inizio e una fine. E per ridurre il numero dei device che computano questi calcoli l'idea è quella di limitare l'area spaziale sulla quale viene calcolato.

Nel caso qui studiato tutto questo si può ottenere attraverso due accorgimenti:

- il primo è quello di terminare il calcolo del percorso una volta che esso è stato identificato. Questo vuol dire che, affinché il sistema sia adattivo e reattivo ai cambiamenti, la richiesta del percorso avvenga periodicamente. L'idea è che la possibilità che il percorso cambi nel breve tempo è molto bassa, quindi si possono tenere cicli temporali abbastanza ampi, nell'ordine dei secondi, se non delle decine di secondi.

- il secondo è quello di limitare il calcolo del gradiente in un area ragionevolmente circoscritta. Cosa significa questo: l'idea parte dal presupposto che conoscendo il punto di arrivo e il punto di partenza il sistema possa

ottenere la direzione del proprio tragitto. Ora limitando spazialmente i calcoli entro una certa latitudine e una certa longitudine, conoscendo le coordinate GPS dei vari nodi, si può ridurre sensibilmente il carico sul sistema, a vantaggio delle performance.

5.4.3 Filtered Gradient

Come visto nei requisiti in 5.2 il sistema prevede la presenza di veicoli prioritari. Questi possono essere modellati come una serie di ostacoli che hanno un inizio e una fine nota, come in figura 5.4.

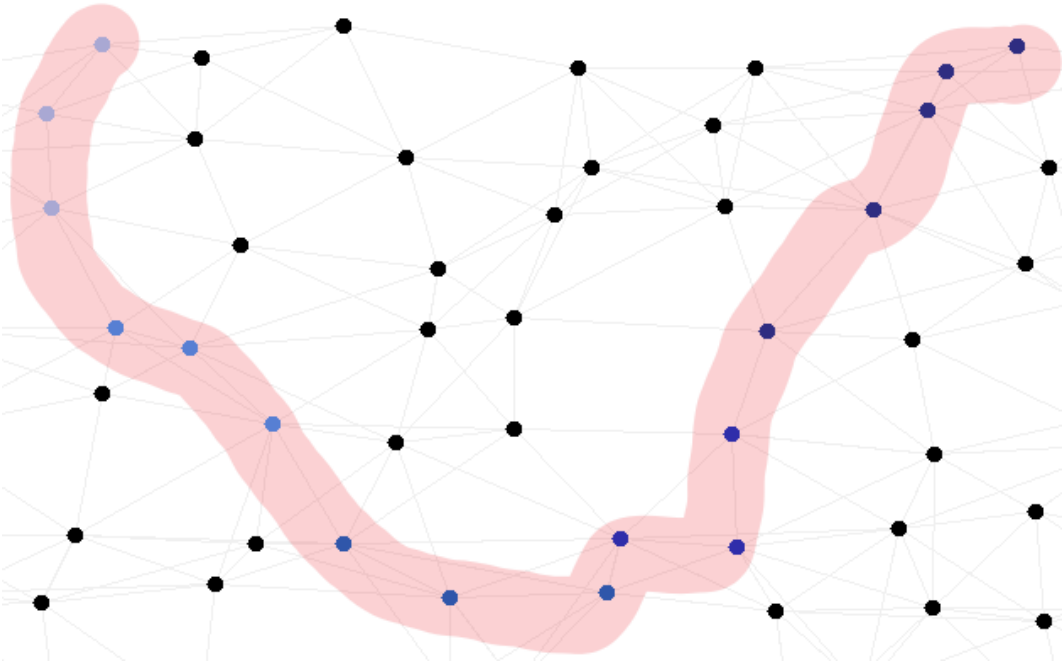

Figura 5.4: Veicolo prioritario

Nel caso in figura 5.4 il percorso è stato mappato come aree di ostacoli con un certo tempo noto, ovvero più nodi hanno lo stesso range temporale, per esemplificare la rappresentazione. Questo tipo di rappresentazione verrà mantenuta anche nei prossimi esempi presi in considerazione.

In realtà in questo caso ha più senso usare una versione di `filteredGradient` basata sugli `ID`.

Il filtro per rappresentare il veicolo viene modellato in questo modo:

```
var vehicle = Map[ID,(Int,Int)]()
```

In questo modo ad ogni nodo è possibile associare un preciso range temporale in cui sarà impedito dall'ostacolo.

Scenari

Gli scenari principali che si possono analizzare riguardo all'utilizzo del *Filtered Gradient* per veicoli prioritari sono sostanzialmente tre:

- **Singolo veicolo impedente** In questo caso la valutazione del percorso

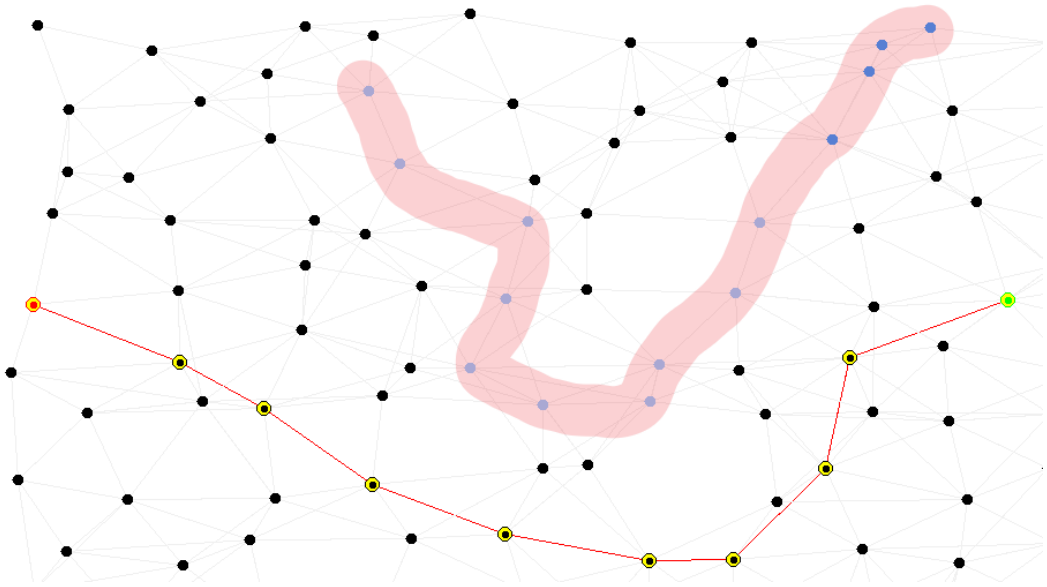

Figura 5.5: Shortest path con singolo veicolo impedente

più breve considera il veicolo come impedimento in quanto il suo passaggio avverrebbe in concomitanza con il passaggio dell'utente. Per lasciare sgombro il tragitto del veicolo prioritario quindi il sistema considera come ostacolo il veicolo su tutti i nodi interessati.

- **Singolo veicolo non impedente** In questo scenario il passaggio del veicolo prioritario avviene o prima o dopo il passaggio dell'utente. Per

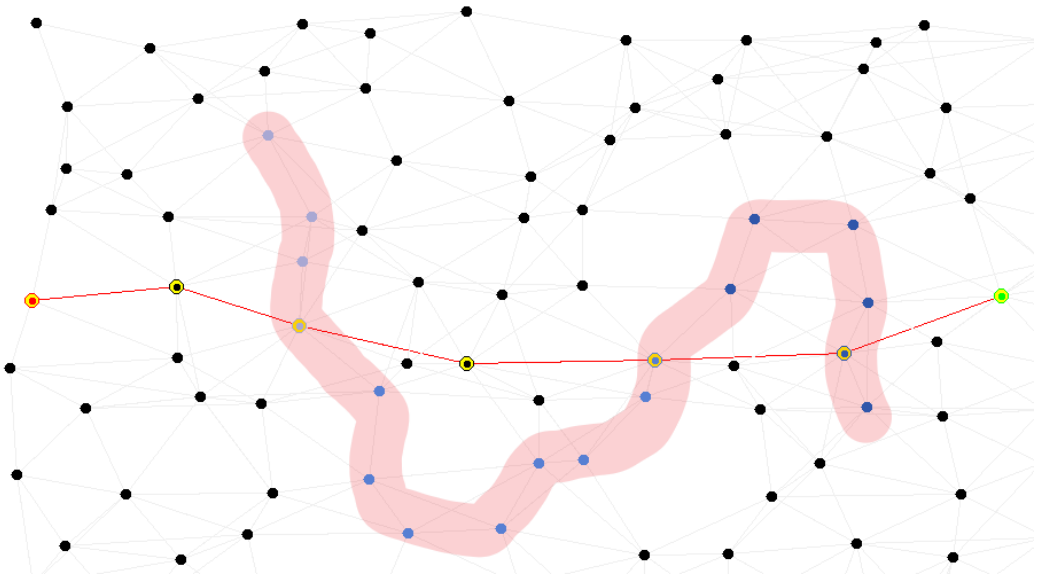

Figura 5.6: Shortest path con singolo veicolo non impedente

questo il sistema non rileva il veicolo prioritario come un impedimento, in quanto gli è già garantita la libertà di passaggio.

- **Più veicoli** In questo caso vi sono più veicoli prioritari e ognuno di essi per avere libero il passaggio viene valutato come ostacolo. Questo avviene perché per entrambi il tempo di percorrenza che intercorrerebbe tra il punto di partenza dell'utente e i nodi interessati al passaggio di tali veicoli fa sì che il passaggio dell'utente avvenga in concomitanza con quello dei mezzi prioritari.

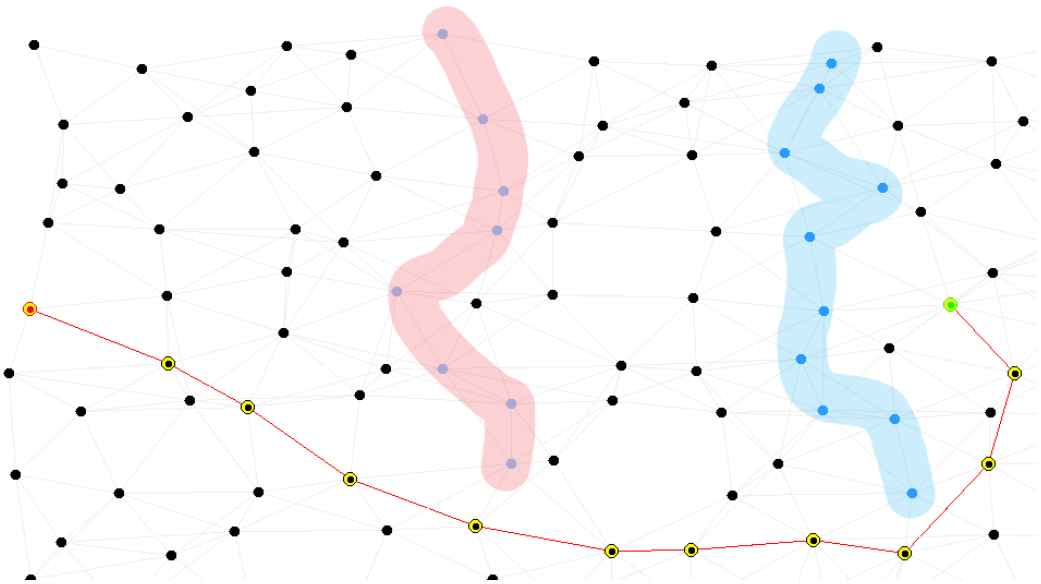

Figura 5.7: Shortest path con multipli veicoli impedente

Conclusioni

Giunti alla fine di questo studio il bilancio è assolutamente positivo. Per prima cosa l'*Aggregate Programming* si è dimostrato una soluzione molto interessante nello sviluppo di sistemi distribuiti. Il codice sviluppato in `scafi` si è dimostrato semplice e molto chiaro, ovvero, conoscendo i costrutti di base di questo paradigma, il codice si sviluppa in poche righe e molto espressive, questo grazie al fatto che è totalmente dichiarativo. È chiaro che lo scoglio grande nell'approcciarsi a tali tecnologie è dato dal modo di pensare *"aggregato"*, in quanto l'attenzione è data al *field* e non al singolo nodo o device; ne consegue che lo sviluppo di codice che si basi su questo concetto non è assolutamente banale. Si mostra banale invece il codice risultante, in quanto in poche righe si riescono a gestire un numero di device elevato senza doversi preoccupare della coordinazione, le comunicazioni tra essi e in generale le problematiche relative ai sistemi distribuiti.

Anche nell'ambito della *Smart Mobility* i risultati ottenuti sono del tutto positivi. Gli algoritmi analizzati hanno mostrato grande potenzialità nella prevenzione e nell'anticipazione. Già l'avere dato un'implementazione all'*Anticipative Gradient* è stato un risultato importante. Ma molto di più l'aver trovato un algoritmo come il *Filtered Gradient* che non solo risponde alle esigenze a cui risponde l'*Anticipative Gradient* ma in più migliora in maniera imponente la qualità del codice e le performance. Questo è vero già dall'anticipazione di un unico ostacolo, e cresce in maniera esponenziale all'aumentare degli ostacoli.

Un algoritmo di anticipazione, come appunto il *Filtered Gradient* apre scenari estremamente interessanti nell'ambito della *Smart Mobility*. Le applicazioni

possibili in cui può essere utilizzato sono parecchie e tutte innovative: in primis la garanzia a veicoli prioritari della strada di loro interesse sgombra, come visto e analizzato nel caso di studio. Questo tipo di soluzione può essere utile per vari scenari: veicoli di emergenza, come ambulanze, pompieri, polizia e in generale tutte le forze dell'ordine. Un'altra applicazione può essere quella delle *road construction*; queste infatti hanno sempre momenti precisi e predeterminati in cui avvengono. Un'altra applicazione simile sono i ponti levatoi e i passaggi a livello; anche qui siamo di fronte ad eventi futuri programmati, che permettono di etichettare i nodi interessati identificando il momento dell'ostacolo. E ancora le gare che hanno tratti urbani, come le gare ciclistiche. I momenti in cui la strada sarà occupata sono conosciuti a priori. Infine forse la più importante applicazione è nella gestione delle congestioni; questo è possibile andando ad *"appesantire"* il tratto di strada e non rendendo il passaggio impossibile. In questo modo si possono modellare scenari di congestione. E per la stessa ragione si possono pure rendere delle vie preferenziali andando ad abbassare il tempo di percorrenza di un dato tratto: questo tipo di applicazione può essere utile ad esempio nella selezione di preferenze come strade senza pedaggio nel proprio tragitto, e in questi casi si può diminuire il tempo di percorrenza di queste.

Tra gli sviluppi futuri il più imminente ed importante è quello riguardante l'analisi delle performance dell'approccio proposto. Prove specifiche sul numero di messaggi scambiati tra i vari device, sui tempi di risposta del sistema e in generale sui costi computazionali dell'approccio preso nella sua completezza saranno le prime analisi da svolgere in seguito a questa tesi. Il valore dei risultati degli studi fatti in questa attività non sono però invalidati dalla mancanza delle analisi delle performance; alla luce di ciò che è stato ottenuto in questa tesi l'*Aggregate Programming* si è dimostrato già un approccio valido, in virtù della sua semplicità e della sua efficacia.

In conclusione l'*Aggregate Programming* si è dimostrato un paradigma di tutto interesse per la realizzazione di sistemi distribuiti e che apre scenari

molto interessanti, non solo nell'ambito della *Smart Mobility* come studiato e approfondito in questa tesi, ma più in generale per tutti i sistemi distribuiti.

Ringraziamenti

Devo prima di tutto porre un grande ringraziamento alla mia famiglia che mi ha sostenuto nello studio di questi anni e soprattutto nell'ultimo periodo in cui il mio lavoro era diretto all'ideazione di questa tesi. Un'altro ringraziamento particolare va a Roberto Casadei e il Prof. Mirko Viroli che mi hanno seguito con pazienza dandomi preziosi consigli e puntualizzazioni sul mio operato e al supporto alla mia tesi del Prof. Dario Maio. Infine un ringraziamento sentito va ai miei compagni di studio di questi cinque anni Matteo Aldini, Brando Mordenti, Andrea De Castri, Davide Foschi, Christian Paolucci, Alex Collini coi quali ho affrontato tutti i progetti ed esami che l'università ci ha posto di fronte.

Bibliografia

[1] Sara Montagna, Danilo Pianini, Mirko Viroli *Gradient-based Self-organisation Patterns of Anticipative Adaptation. SASO 2012: 169-174*

[2] Roberto Casadei, Mirko Viroli *Towards Aggregate Programming in Scala. PMLDC@ECOOP 2016: 5*

[3] Jacob Beal, Mirko Viroli, Danilo Pianini *Aggregate Programming for the Internet of Things. IEEE Computer 48(9): 22-30 (2015)*

[4] Jacob Beal, Mirko Viroli *Building blocks for aggregate programming of self-organising applications. SASO Workshops 2014: 8-13*

[5] Danilo Pianini, Mirko Viroli *Protelis: Practical Aggregate Programming. SAC 2015: 1846-1853*

[6] Clara Benevolo, Renata Dameri, Beatrice D'Auria *Smart Mobility in Smart City. Action taxonomy, ICT intensity and public benefits. Systems and Organisation 11, DOI 10.1007/978-3-319-23784-8-2: 13-28. January 2016*

[7] Luca Staricco *Smart Mobility Opportunità e Condizioni. DOI: 10.6092/1970-9870/1933: 341-354. 2013*

[8] Barbara Lenz, Dirk Heinrichs *What Can We Learn from Smart Urban Mobility Technologies? IEEE Pervasive Computing 1536-1268/17. DOI:10.1109/MPRV.2017.27. 2017*

[9] Zhaolong Ning, Feng Xia, Noor Ullah, Xiangjie Kong, Xiping Hu *Vehicular Social Networks: Enabling Smart Mobility. IEEE Communications Magazine. DOI 10.1109/MCOM.2017.1600263 April 2017*

[10] Tatsuo Okuda, Shigeki Hirasawa, Nobuhiko Matsukuma, Takashi Fukumoto, Akitoshi Shimura *Smart Mobility for Smart Cities. Hitachi Review Vol. 61 (2012), No. 3*

[11] European Parliament *Mapping Smart Cities in the EU. IP/A/ITRE/ST/2013-02 January 2014*

[12] *European Smart Cities* `http://www.smart-cities.eu/`

[13] Marlin Wolf Ulmer *Approximate Dynamic Programming for Dynamic Vehicle Routing. Springer International Publishing AG 2017*

[14] Xiangjie Konga, Zhenzhen Xua, Guojiang Shenb, Jinzhong Wanga, Qiuyuan Yanga, Benshi Zhanga *Urban traffic congestion estimation and prediction based on floating car trajectory data. 0167-739X/© 2015 Elsevier B.V.*

[15] Xiao Zhang, Enrique Onieva, Asier Perallos, Eneko Osaba, Victor C.S. Lee *Hierarchical fuzzy rule-based system optimized with genetic algorithms for short term traffic congestion prediction. 0968-090X 2014 Elsevier Ltd.*

[16] Rutger Claes, Tom Holvoet, and Danny Weyns *A Decentralized Approach for Anticipatory Vehicle Routing Using Delegate Multiagent Systems. Digital Object Identifier 10.1109/TITS.2011.2105867. 2011*

[17] *How does Google Maps predict traffic?* `http://electronics.howstuffworks.com/how-does-google-maps-predict-traffic.htm`

[18] *How does Waze work?* `https://support.google.com/waze/answer/6078702?hl=en`

[19] *Waze* `https://www.waze.com`

Elenco delle figure