Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

CORREZIONE E SEMPLIFICAZIONE DI UN GRAFO STRADALE UTILIZZANDO RILEVAZIONI SATELLITARI GPS

Tesi di Laurea in Fisica dei Sistemi Complessi

Relatore:
Chiar.mo Prof.
Sandro Rambaldi

Presentata da:
Massimo Intelisano

I Sessione
Anno Accademico 2010-2011

Indice

Introduzione **5**

1 Rappresentazione dell'informazione territoriale **9**

1.1 Informatizzazione di una rete stradale 9

1.2 La rete stradale 12

1.2.1 Il modulo PolyArchiRef 17

1.3 Integrità della rete stradale 21

1.3.1 Incoerenze della rete stradale di Roma 23

1.4 La rete veicolare 27

1.4.1 Strutture dati 27

1.5 Associazione veicolo – arco 28

1.6 L'arco rispetto alla direzione del moto del veicolo 30

2 L'editor della rete stradale **33**

2.1 La struttura dell'editor 33

2.2 Le librerie grafiche 34

2.3 Il display dell'editor 36

2.4 Visualizzazione della rete stradale 37

2.5 La griglia della rete stradale 42

2.6 Rilevamento delle anomalie 49

2.7 Manipolazione del grafo 52

2.8 Gestione delle polilinee 54

2.9 Gestione degli archi 57

2.10 Esportazione di un sottografo 61

2.10.1 Esportazione circolare 61

2.10.2 Esportazione poligonale 64

2.11 Inserimento automatico delle polilinee 66

3 Semplificazione della rete stradale **75**
3.1 Cammini minimi 75
3.2 Algoritmo di Johnson 78
3.3 Realizzazione tramite un Time Stamp Heap 79
3.4 Algoritmo A* 82
3.5 Semplificazione a priori 84
3.5.1 Rimozione delle strade cieche 85
3.5.2 Rimozione dei nodi di rango 2 86
3.5.3 Rimozione delle polilinee 89
3.6 Semplificazione a posteriori 90
3.6.1 Determinazione dei frammenti del grafo 90
3.6.2 Connessione dei frammenti 92
3.7 Analisi conclusive sulle rimozioni 94

Conclusione e sviluppi futuri **105**

A Regressione lineare **107**

B IsGeom **109**

C Errore posizionale dei veicoli **113**

Introduzione

Questa tesi è la naturale continuazione di un precedente studio sulla correzione delle mappe stradali, affrontato in occasione della laurea triennale. Si lavora in via sperimentale a un sistema informatico per la realizzazione di un grafo, che ridefinisce la rete urbana della città di Roma in base alle informazioni veicolari GPS ad essa annesse. Il sistema determina la rete stradale con un grafo $G = (N, A)$ orientato e connesso e quella veicolare con un vettore contenente i dati relativi alla posizione, alla velocità e alla direzione del moto veicolare stesso. Con una particolare elaborazione comparativa, analizza i dati sulla posizione e la direzione del moto dei veicoli e li raffronta al grafo della rete stradale attinente al flusso. Esamina la corrispondenza delle differenti fonti, ne rileva le eventuali discordanze, fino a determinare una corretta visione della rete stradale. Basandosi su un modello logico relazionale realizza l'associazione fra le due reti aggiungendo alle caratteristiche dei veicoli un attributo, denominato *strada*, che contiene l'informazione dell'arco percorso dal veicolo. Il campo *strada* rappresenta la chiave esterna per associare il veicolo al corrispondente arco. A ogni arco identificato univocamente da un indice, sono associati più veicoli anch'essi identificati univocamente, mentre ogni veicolo è associato a uno e un solo arco. Così predisposte le strutture dati, si implementano gli algoritmi che determinano per ogni veicolo l'arco di percorrenza più attinente. I metodi, gli algoritmi e le strutture dati, acquisiti nella precedente esperienza, sono basilari per ridefinire l'informazione territoriale. In questo contesto, grande rilevanza assume il processo di sovrapposizione delle reti stradale-veicolare, utilizzato per il rilevamento dei flussi veicolari anomali e la rispettiva correzione del grafo.

Il grafo ottenuto, però, visualizza a video un'immagine statica, che non permette alcun tipo di manipolazione interattiva. Inoltre, il processo di correzione delle anomalie evidenziate dai flussi veicolari è incompleto, poiché non prevede l'inserimento di nuove polilinee che risultano mancanti nella rete stradale. Questo studio vuole migliorare la complessità del sistema apportando soluzioni algoritmiche sperimentali idonee a ottenere, in tempi brevi e con risultati soddisfacenti, una correzione interattiva delle anomalie della

rete stradale.

Il primo obiettivo di questa tesi è quello di realizzare un sistema informatico capace di operare e apportare agevolmente le modifiche nella rete stradale di Roma. Da qui nasce l'idea di creare un editor idoneo a visualizzare e interagire graficamente sulle informazioni. L'editor è costituito da una componente principale che permette di visualizzare a video l'immagine della sovrapposizione delle reti stradale-veicolare e consente di interagire su di essa. La complessità delle attività e dei servizi interattivi viene fornita da altre componenti che, ricevendo gli input esterni, apportano le modifiche al grafo. A tal proposito si realizzano i framework *OpGrafo* e *Disegno*. Il primo fornisce un'api che astrae le operazioni di manipolazione del grafo, per apportare le modifiche alle polilinee, agli archi e ai nodi estremi. Il secondo fornisce un'altra api che astrae le operazioni di basso livello, per creare gli oggetti grafici corrispondenti alle due reti. I servizi interattivi vengono gestiti con una serie di funzioni, che prendono come argomenti di input le coordinate cartesiane della porzione di rete stradale da modificare e operano gli opportuni cambiamenti ai corrispondenti elementi del grafo. Per individuare un elemento di coordinate cartesiane (x, y) si implementa un modulo che definisce una griglia, capace di ospitare la rete stradale e di determinare in tempo costante gli archi su cui cade un dato punto. Questo permette di rendere indipendenti le prestazioni del sistema dalle dimensioni del grafo stradale. Una volta realizzata la base dell'editor, si implementano le componenti per apportare possibili modifiche al grafo. A tal fine si creano *HwndPoly* e *HwndArco*, che dotano l'editor di una serie di azioni interattive guidate per inserire, spostare, dividere e eliminare gli elementi della rete stradale. A queste si aggiunge la componente opzionale *Esporta*, che consente di estrapolare, in qualsiasi forma voluta, il sottografo modificato e di salvare il lavoro finito. L'opera sin qui ottenuta è una buona base di partenza, che apre nuove prospettive di ricerca per risolvere il problema dell'inserimento automatico delle polilinee mancanti.

Il secondo obiettivo della tesi, infatti, è quello di automatizzare il sistema di correzione. Fissati con l'ausilio del mouse i punti di riferimento del flusso veicolare su cui operare, dove i nodi estremi fanno da limite di selezione e un qualsiasi punto interno da starter, si implementa l'algoritmo *rilevaFlusso*. Questo mediante una ricerca ricorsiva a largo spettro, marca tutti i veicoli con una colorazione diversa fino a raggiungere i nodi prefissati, evidenziando così l'intero flusso veicolare. Si studia un possibile metodo che crei automaticamente sul flusso evidenziato una nuova polilinea, per correggere l'anomalia della rete stradale. A tal fine si realizza il metodo *ricostruisciPoli* che, unendo con un unico arco i nodi estremi del flusso veicolare preso in esame, traccia una primordiale polilinea. Individua il punto del flusso più lontano dalla po-

lilinea generata e traccia due nuovi archi, che collegano il punto individuato ai nodi estremi della primordiale polilinea. Per fasi successive di affinamento si itera, con una buona approssimazione della realtà, questa procedura di sdoppiamento degli archi, fino a ricostruire automaticamente la polilinea mancante della rete stradale. L'attribuzione del senso di percorrenza della polilinea si realizza mediante una funzione, che compara la direzione del moto dei veicoli con la posizione dei rispettivi archi associati. Il senso di marcia sull'arco è dato dal coseno dell'angolo di percorrenza del veicolo rispetto a quello dell'arco. Se è positivo il veicolo percorre l'arco dal nodo di testa a quello di coda, invece, se è negativo lo percorre al contrario. Applicando il metodo su tutti i veicoli si stabilisce il senso di percorrenza della polilinea.

Terzo e ultimo obiettivo della tesi è quello di ricercare le possibili soluzioni di semplificazione del grafo stradale, per avere una visione dettagliata dell'andamento del flusso veicolare che transita in esso. Tale semplificazione consiste nel rimuovere dal grafo le polilinee con un flusso di traffico inferiore a un certo δ fissato, mantenendo i nodi in un'unica classe di equivalenza. La rimozione di una polilinea non solo può causare la sconnessione del grafo, ma anche può creare due classi di equivalenza distinte, dove i nodi appartenenti alla prima possono raggiungere quelli della seconda, ma non viceversa. Dato che in una rete stradale ogni nodo deve essere raggiungibile dagli altri, è necessario mantenere il grafo con un'unica classe di equivalenza. Si studiano gli algoritmi per eliminare le polilinee superflue, cioè quelle che non forniscono informazioni rilevanti per la correzione del grafo, mantenendo le proprietà di connessione tra i nodi. A tal fine, in via sperimentale, si individuano tre metodi di semplificazione: *a priori*, *a posteriori per distanze minime* e *a posteriori per flussi massimi*. Queste tecniche consentono di semplificare il grafo G in un sottografo $G' \subseteq G$ connesso che contiene solo le strade trafficate. Le soluzioni si rifanno allo studio del problema dei cammini minimi tra i nodi. Tale problema è noto in letteratura e si risolve con algoritmi efficienti, che operano su classi di grafi con pesi non negativi. Tuttavia, la realizzazione di una procedura che iteri molte volte la ricerca di un cammino minimo tra due nodi, causa forti rallentamenti soprattutto nella fase di inizializzazione delle strutture dati. Per cui si implementa un algoritmo in grado di effettuare un numero arbitrario di cammini minimi consecutivi, realizzati tramite un *time stamp heap*, in modo che la fase di inizializzazione delle strutture non infici il costo computazionale complessivo della ricerca stessa.

Si implementa il metodo *a priori* che opera la semplificazione accertandosi preventivamente che, a ogni fase iterativa di rimozione delle polilinee, il grafo resti costantemente connesso mantenendo i nodi in un'unica classe di equivalenza. L'opera di semplificazione, quindi, ha lo scopo di rimuovere agevolmente le polilinee, corrispondenti a strade cieche, senza sconnettere il

grafo. Perciò si implementano due funzioni: una spezza i circuiti della rete stradale, favorendo così la formazione di strade cieche e l'altra elimina i nodi di rango 2, per permettere l'unificazione di due polilinee connesse in una sola. Tale processo di rimozione, però, non sempre è speculare all'andamento del flusso veicolare. Ci sono tratti stradali, considerati indispensabili per il mantenimento delle proprietà stesse del grafo, che se rimossi ne causerebbero la sconnessione, per cui non possono essere cancellati.

Si implementa, allora, un algoritmo di rimozione *a posteriori*, che elimina tutte le polilinee con un flusso di traffico inferiore alla soglia stabilita, a prescindere dal mantenimento o meno delle proprietà di connessione del grafo. Tale modalità di rimozione, detta *naturale*, provoca la sconnessione del grafo in tanti frammenti di archi che devono essere necessariamente riconnessi.

Si testano due metodi per riconnettere tra loro i frammenti isolati in un'unica classe di equivalenza. Il primo metodo di riconnessione, denominato *per distanze minime*, ricerca il cammino più breve che connette in entrambi i sensi di marcia i frammenti del grafo. Il secondo metodo, per *flussi massimi*, sfrutta il più possibile gli archi già presenti nel grafo e evita il reinserimento di quelli poco trafficati. A conclusione del lavoro espletato si eseguono una serie di test sul grafo della rete stradale di Roma, per stabilire quale sia il processo più idoneo.

Nel primo capitolo viene data una panoramica generale del sistema informatico. Si descrivono le strutture dati utilizzate per rappresentare le reti stradale-veicolare e le tecniche basilari per realizzare la loro associazione. Si riportano alcune incoerenze trovate nei dati e la loro correzione.

Nel secondo capitolo sono trattatati i metodi, gli algoritmi e le strutture dati per creare un editor interattivo, capace di visualizzare e interagire graficamente sulle informazioni derivanti dal processo di sovrapposizione della rete stradale con quelle GPS veicolari annesse. Si approfondiscono le tecniche utilizzate per realizzare le componenti interattive che permettono di modificare le polilinee, i nodi estremi e gli archi del grafo. Si illustrano anche le soluzioni adottate per l'inserimento automatico di una polilinea mancante evidenziata dal corrispondente flusso veicolare anomalo. Si descrive, infine, la metodologia osservata per tracciare e determinare il senso di percorrenza della polilinea corretta.

Il terzo capitolo tratta le tecniche che realizzano la semplificazione del grafo stradale. Si illustrano i processi di rimozione e si riportano i risultati dei test effettuati sul grafo della rete stradale di Roma.

Capitolo 1

Rappresentazione dell'informazione territoriale

1.1 Informatizzazione di una rete stradale

Una rete stradale informatizzata è un grafo $G = (N, A)$ orientato e connesso. Dove N è un insieme non vuoto e A è una relazione binaria su N , $A \subseteq N \times N$, ossia un insieme di coppie ordinate di elementi di N. Gli elementi di N sono chiamati vertici, o nodi e quelli di A spigoli o archi. Dati due nodi di un grafo u e v, si definisce formalmente un arco di estremi (u, v) la connessione identificata da un segmento di nodi (u, v). Un grafo è orientato se l'insieme di archi che lo compongono sono orientati, ovvero hanno una proprietà direzionale che ne determina il senso di percorrenza. L'arco è costituito da un punto iniziale, detto testa, identificato dal primo valore della coppia del segmento, e da un punto finale, detto coda, identificato dal secondo valore. Un arco con nodi estremi (u, v), dove u è la testa dell'arco e v la sua coda, ha un significato diverso rispetto a quello identificato dalla coppia di nodi (v, u). Nella letteratura classica la notazione (u, v) sui grafi orientati viene indicata con v raggiungibile da u e non viceversa, invece, in questo studio la percorribilità è data dal valore $s_{(u,v)}$, che viene attribuito ad ogni arco. Il valore $s_{(u,v)}$, come dimostra la figura 1.1, determina il senso di percorrenza dell'arco (u, v). Se $s_{(u,v)}$ è uguale a:

- 0, si ha un doppio senso *(A)*;
- 1, si ha un senso unico, che va dal nodo di testa a quello di coda *(B)*;

- 2, si ha un senso unico opposto, che va dal nodo di coda a quello di testa *(C)*.

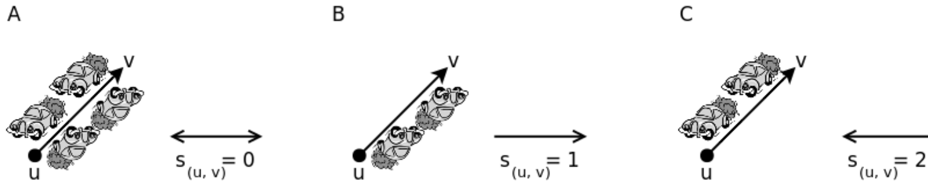

Figura 1.1: Senso di percorrenza degli archi

La proprietà di orientamento, così disaccoppiata da quella direzionale, non preclude comunque la possibilità di costruire un grafo $G = (N, A)$. Pertanto, preso in considerazione un grafo non orientato, se ne modifica la relazione binaria per escludere gli archi non percorribili in base alla proprietà direzionale attribuita. Sia $G' = (N', A')$ un grafo non orientato, dove A' è una relazione binaria simmetrica, per cui la presenza dell'arco (u, v) in A', implica quella dell'arco (v, u). Aggiungendo ad ogni arco la relativa proprietà inerente il senso di percorrenza, si ottiene il grafo $G = (N, A)$; dove A è una relazione binaria asimmetrica, derivante da A', che esclude tutti gli archi non idonei al senso di marcia stabilito. Pertanto, G è formato solo dagli archi che soddisfano una delle seguenti condizioni di esistenza:

- se $s_{(u,v)} = 0$, $(u, v) \in A \land (v, u) \in A$;
- se $s_{(u,v)} = 1$, $(u, v) \in A \land (v, u) \notin A$;
- se $s_{(u,v)} = 2$, $(u, v) \notin A$ e $(v, u) \in A$;
- se $s_{(u,v)} = 3$, $((u, v) \notin A \land (v, u) \notin A) \lor ((u, v) \in A \land (v, u) \in A)$.

L'ultima condizione permette di considerare o escludere gli archi di particolari zone del grafo G, come per esempio quelle a traffico limitato marcate "ZTL"; e se presi in esame, vengono trattati alla stregua degli archi a doppio senso. Ma il senso di marcia di un arco (u, v), così determinato dalle proprietà di orientamento e percorrenza, potrebbe prestarsi ad una non facile lettura. A tal fine si introduce la seguente notazione(*):

- (u, v) = arco con nodo di testa u e nodo di coda v;
- $(u, v)^{\rightarrow}$ = arco percorribile a senso unico dal nodo di testa u al nodo di coda v $[(u, v) \in A]$;

- $(u,v)^{\leftarrow}$ = arco percorribile a senso unico opposto dal nodo di coda v al nodo di testa u $[(v,u) \in A]$;

- $(u,v)^{\leftrightarrow}$ = arco percorribile a doppio senso di marcia dal nodo di testa u al nodo di coda v e viceversa $\{(u,v),(v,u)\} \in A$.

L'arco (u,v) è l'unità fondamentale di G, che determina un tratto lineare della rete stradale. Sia $P = \{u_0, u_1, \ldots, u_{n-1}\} \in N$ una catena percorribile ciclica o aciclica di nodi, che gode delle seguenti proprietà:

- il numero di nodi raggiungibili da u_0 e u_{n-1} può essere maggiore o uguale di uno;

- tutti gli archi hanno lo stesso senso di percorrenza;

- per ogni $i = 1, \ldots, n-1$, u_i è raggiungibile solo e esclusivamente da u_{i-1};

- da u_{n-2} si può raggiungere solo e esclusivamente u_{n-1}.

Allora P è una polilinea che definisce un tratto stradale complesso. È formata da una sequenza ordinata $\{u_0, u_1, \ldots, u_{n-1}\}$ di nodi appartenenti ad N, dove u_0 e u_{n-1} sono gli estremi di P, mentre $\{u_1, \ldots, u_{n-2}\}$ sono interni a P. Il nodo u_0 è chiamato estremo iniziale (o nodo di testa) e u_{n-1} è detto estremo finale (o nodo di coda). I nodi estremi servono a connettere le polilinee tra loro, dando origine agli incroci stradali. Per la proprietà di cui al sopra enunciato punto 2, la polilinea P ha lo stesso senso di marcia s uguale a quello degli archi. La figura 1.2 mostra una polilinea P di lunghezza sei, dove il nodo estremo iniziale è u_0, mentre quello finale è u_5.

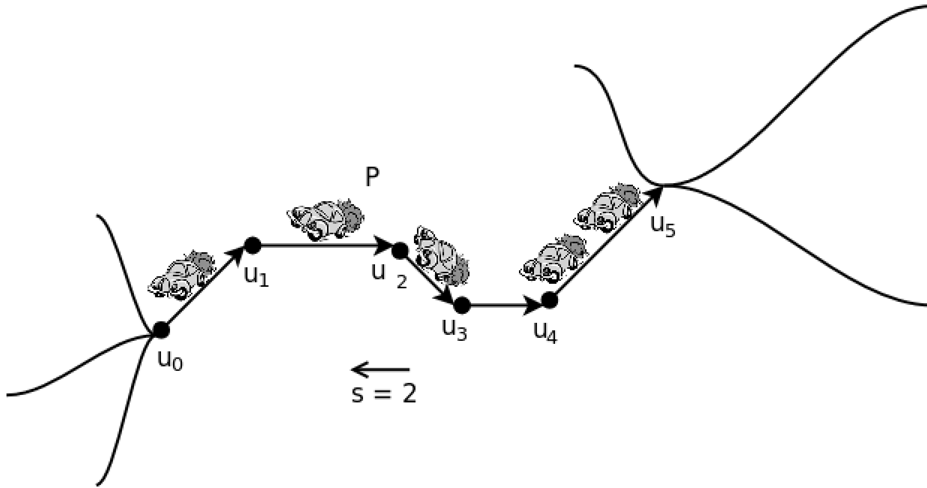

Figura 1.2: Polilinea percorribile dal nodo di coda a quello di testa

Si osserva che solo u_0 e u_5 sono raggiungibili da altre polilinee e i nodi interni $\{u_1, \ldots, u_4\}$ sono collegati consecutivamente a due a due e non sono rggiungibili da altre polilinee. Il senso di percorrenza $s = 2$, comune per tutti gli archi, indica che la polilinea ha un'unica direzione di marcia, dal nodo di coda u_5 a quello di testa u_0. Si nota,inoltre, che A contiene gli archi $\{(u_5, u_4),\ (u_4, u_3), \ldots, (u_1, u_0)\}$, e $\{(u_0, u_1), (u_1, u_2), ..., (u_4, u_5)\} \notin A$. Pertanto, applicando la sopra enunciata notazione (*), il senso di percorrenza di P si descrive con il simbolo $P^{\leftarrow}$ e quello degli archi che la compongono con $\{(u_0, u_1)^{\leftarrow}, (u_1, u_2)^{\leftarrow}, \ldots, (u_{n-1}, u_n)^{\leftarrow}\}$.

1.2 La rete stradale

Il grafo della rete stradale nazionale trae origine da un database, leggibile dagli attuali sistemi GIS, commercializzato dall'azienda Tele Atlas. Dall'enorme archivio, mediante una query di proiezione sui campi contenenti le proprietà desiderate ed un'altra di selezione sulle polilinee per ottenere quelle circoscritte da coordinate geografiche minime e massime, si ricava una tabella dati di una sotto-rete stradale, utili ai fini di questa tesi, di cui se ne elencano di seguito gli attributi:

- identificatore universale (*cidp*), che permette di distinguere univocamente ogni polilinea dalle altre del globo;

- posizione geografica (lon, lat), espressa in coordinate longitudinali e latitudinali;
- identificatore universale dei nodi estremi $(cidn_0, cidn_1)$, che determina l'inizio e la fine del tratto stradale;
- senso di percorrenza s, che determina il modo in cui il tratto stradale si percorre;
- lunghezza in metri m;
- nome della strada $nome$.

Tali dati, per una migliore leggibilità, vengono raggruppati e salvati su tre file (*poly_prop, poly_pnts e nodi*).

Poly_prop

Contiene le informazioni solo di alcune proprietà delle polilinee:

- l'identificatore univoco ($cidp$);
- l'identificatore univoco del nodo estremo iniziale ($cidp_0$);
- l'identificatore univoco del nodo estremo finale (cid_1);
- la lunghezza in metri;
- il senso di marcia rispetto all'orientamento degli archi;
- il nome della strada.

Il senso di marcia è una stringa di due caratteri, che determina la direzione dal nodo estremo iniziale a quello finale. Pertanto se la stringa è uguale a:

- __ la polilinea è percorribile nei due sensi di marcia;
- TF la polilinea è percorribile dal nodo estremo di testa a quello di coda;
- FT la polilinea è percorribile dal nodo estemo di coda a quello di testa;
- _N indica una zona a traffico limitato non percorribile.

Poly_pnts

Contiene le informazioni relative alla posizione geografico - terrestre delle polilinee. Ogni polilinea, identificata dal proprio *cidp*, ha la lista delle coordinate spaziali dei nodi che la compongono. La posizione è data da coordinate longitudinali e latitudinali espresse in gradi interi. Vi è inoltre un indice relativo, chiamato *id_poly*, che enumera in ordine crescente le polilinee. Data un'estrazione di n polilinee, la prima di queste ha come indice relativo 1, la seconda 2, e così via fino all'ultima con *id_poly* pari ad n. L'indice relativo, non solo, identifica univocamente le n polilinee presenti nel file, ma ne rappresenta anche il cursore di accesso al vettore che le contiene. Ad esempio, se la prima polilinea, con *id_poly* pari ad 1, è contenuta nella prima cella del vettore, conoscendo il suo indice relativo, si può accedere in tempo costante $O(1)$ alla corrispondente posizione del vettore, per leggerne o modificare le informazioni.

Nodi

Il terzo file contiene le informazioni dei nodi estremi delle polilinee. Ogni nodo estremo è identificato mediante il suo codice universale *cidn*. Come nel caso delle polilinee, vi è un indice relativo chiamato *id_nodo*, che è una enumerazione compresa tra 1 e n, dove n è uguale al numero di nodi presenti nel file. Da ciò si evince che l'indice relativo dei nodi inseriti in un vettore è pari al cursore corrispondente alla posizione degli stessi nodi nel vettore. Per ogni nodo è presente una lista che contiene una coppia di indici (k, h). L'indice k è l'*id_poly* che si collega al nodo specificato q. Se $k > 0$, la polilinea k ha come nodo di testa q e come nodo di coda h, mentre se $k < 0$, ha come nodo di testa h e come nodo di coda q. L'immagine seguente illustra il funzionamento degli indici relativi di una poliliea K, con i nodi di testa T e di coda C. In base al sopraccitato sistema degli indici relativi, si accede alle informazioni di K, T e C, utilizzando rispettivamente i cursori k, t e c.

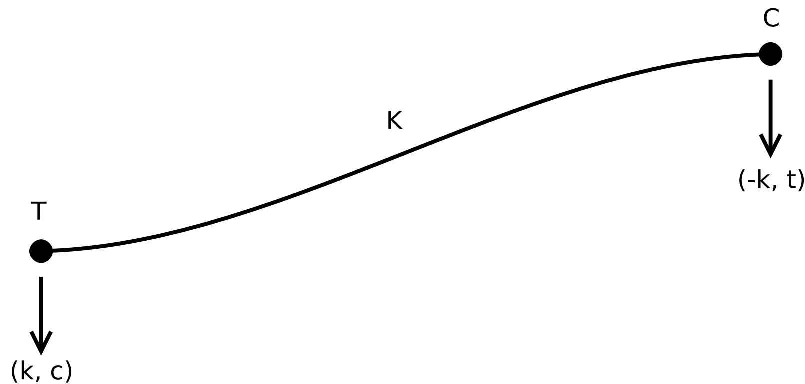

Figura 1.3: Cursori dei nodi estremi di una polilinea

Osservando la figura 1.3 si nota che T contiene i cursori (k, c) con i quali si può accedere in $O(1)$ alla posizione di K e C. Il valore $k > 0$ indica che la polilinea K ha il nodo estremo iniziale in T e quello estremo finale in C. Il nodo C, invece, ha la coppia di cursori $(-k, t)$, dove il valore $k < 0$ indica che K ha il nodo estremo finale in C e quello estremo iniziale in T. Questi dati vengono strutturati utilizzando le funzioni $leggiStradeElaborate$ e $leggiNodiElaborati$. La prima inizializza il vettore V_p che fa riferimento alla classe *PolyLine*, mentre la seconda inizializza il vettore V_n che fa riferimento alla classe *Nodo*. La funzione aggiuntiva *crea_archi*, che fa riferimento alla classe *Arco*, inizializza il vettore V_a trasformando la struttura di V_p in un vettore di archi. La classe *PolyLine* realizza una polilinea con un vettore di punti, inizializzato dalle coordinate espresse in gradi decimali di sei cifre contenute in *poly_pnts*, e le proprietà presenti in *poly_prop*.

Contiene anche gli attributi derivati:

- $(id_estremo0, id_estremo1)$, lega la polilinea ai relativi cursori dei nodi estremi che la connettono al grafo [*];
- *rimosso*, stabilisce l'appartenenza o meno della polilinea al grafo;
- *tipo*, distingue le polilinee provenienti dalla base di dati di TeleAtlas, da quelle inserite in altro modo;
- *n_car*, contiene il numero di veicoli che percorrono la polilinea [*];
- (vel_0, vel_1), contiene le velocità medie in entrambi i sensi di marci dei veicoli che percorrono la polilinea [*].

La classe *Nodo* determina un nodo con due vettori di numeri interi. Il primo (*id_link*) contiene i cursori delle polilinee collegate al nodo stesso. Il secondo (*id_nodi_vicini*) contiene i rispettivi cursori degli altri nodi estremi delle polilinee di *id_link*. Perciò, dato un nodo u con i vettori $id_link = \{p_0, p_1, \ldots, p_{n-1}\}$ e $id_vicino = \{u_0, u_1, \ldots, u_{n-1}\}$; per ogni $i = 0, \ldots, n-1$, $id_link[i]$ rappresenta il cursore della polilinea $|p_i|$ e $id_vicino[i]$ rappresenta l'altro nodo estremo di $|p_i|$.

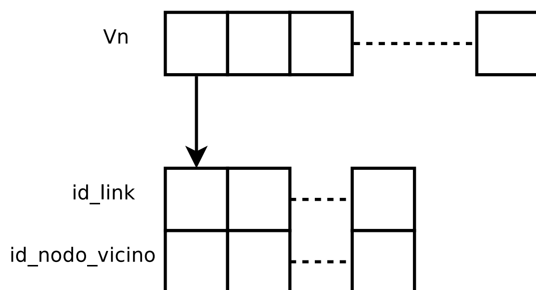

Figura 1.4: Struttura dati V_n

La classe $Nodo$ ha, anche, i seguenti attributi derivati:

- (lon, lat), contiene le coordinate geografiche del nodo [*];
- (x, y), contiene il risultato del mapping delle coordinate geografiche (lon, lat) in un sistema di riferimento cartesiano bidimensionale [*];
- *rimosso*, stabilisce l'appartenenza o meno del nodo al grafo;

La classe *Arco* determina un arco di coordinate geografiche $(lon_0, lat_0, lon_1, lat_1)$ e come la *PolyLine*, è dotata dei cursori di accesso alla struttura dati dei nodi estremi.

I suoi più significativi attributi sono:

- *id_arco*, identifica il cursore dell'arco in V_a;
- *id_poly*, identifica il cursore della polilinea appartenente all'arco;
- *id_punto*, identifica gli archi che hanno lo stesso *id_poly*;
- *id_nodo_inizio*, identifica il cursore del nodo estremo iniziale della polilinea;
- id_nodo_fine, identifica il cursore del nodo estremo finale della polilinea;

- (lon_0, lat_0), identifica le coordinate del nodo di testa dell'arco;
- (lon_1, lat_1), identifica le coordinate del nodo di coda dell'arco;
- (x_0, y_0, x_1, y_1), identifica l'arco con le coordinate cartesiane [*].

Pertanto, data una polilinea $P = \{a_0, a_1, \ldots, a_{h-1}\} \in V_n$, gli archi $\{(a_0, a_1), (a_1, a_2), \ldots, (a_{h-2}, a_{h-1})\} \in V_a$, i cui valori di *id_punto* sono un'enumerazione crescente da 0 ad $h - 2$. Se *id_punto* è uguale a zero, l'arco comprende il nodo di testa di P e *id_nodo_inizio* è pari al cursore $id_estremo_0$ di P. Se *id_punto* è uguale ad $h - 2$, comprende il nodo di coda di P e id_nodo_finale è pari al cursore $id_estremo_1$ di P. Se, invece, *id_punto* ha un valore compreso tra 1 e $h - 3$, gli attributi *id_nodo_inizio* e id_nodo_fine sono uguali a -1.

L'informazione espressa da V_a è equivalente a quella espressa da V_p. V_a permette l'accesso diretto agli archi mediante il cursore *id_arco*. Ma se si conosce solo il cursore della polilinea (*id_poly*), non si può accedere in $O(1)$ alla sequenza degli archi di V_a della polilinea stessa. In questo modo, si dovrebbe visitare tutto V_a, impiegando tempo $O(n)$, per trovare gli archi con *id_poly* uguale a quello del cursore della polilinea. Il problema si risolve creando un legame tra il cursore della polilinea e quello del primo arco che la identifica. Il modulo $poly_archi_ref$ permette di creare tale legame e di risalire in tempo costante al cursore del primo arco di un *id_poly* specifico.

1.2.1 Il modulo PolyArchiRef

Con la seguente interfaccia si risale in tempo costante $O(1)$ al cursore del primo arco, che ha come *id_poly* quello della polilinea:

- void $initPolyRef$ (int *size*), inizializza la struttura dati e crea l'associazione tra il cursore della polilinea e quello dell'arco;
- void $deallocPolyRef()$, dealloca la struttura dati;
- int $addPolyRef$ (int *id_poly*, int *id_arco*), crea il legame tra il cursore della polilinea e quello del primo arco;
- int $getPolyRef$ (int $* \ n$, int *id_poly*), restituisce il cursore dell'arco associato alla polilinea *id_poly*.

Il vettore di tipo opt_ref (v. fig.1.5) ha due campi di numeri interi (*id_arcoen_archi*), che realizzano il collegamento tra il cursore della polilinea e quello dell'arco. Il campo *id_arco* contiene il cursore del primo arco della polilinea, mentre *n_archi* indica il numero di archi che la compongono.

Si riporta la struttura dati *opt_ref*:

```
typedef struct{
        int id_arco;
        int n_archi;
}opt_ref;
```

La funzione *initPolyRef* inizializza il vettore di tipo *opt_ref* , creando una sequenza di celle (size) corrispondenti al numero di polilinee di V_p. Previa l'attribuzione dei cursori *id_arco* e *id_poly* su tutti gli archi del grafo, si applica la funzione *addPolyRef* che compila i campi *id_arco* e *n_archi*. Se l'*opt_ref* è vuota, inserisce in *id_arco* il cursore dell'arco e in n archi il valore 1. Se invece è piena, inserisce in *id_arco* il cursore più basso, desunto dalla comparazione tra il valore preesistente con quello dato in input, e incrementa *n_archi* di una unità. Quest'ultimo caso è dovuto al fatto che V_a è sempre ordinato per *id_punto* (che varia da 0 a $n_archi - 1$) e gli *id_arco* sono un'enumerazione crescente. La funzione *getPolyRef*, restituendo le informazioni relative al legame, fornisce il cursore dell'arco più piccolo e il numero di archi appartenenti alla polilinea. In questo modo si può accedere in tempo costante a tutti gli archi di una polilinea, spaziando da quello con *id_arco* minore fino a quello con *id_arco* maggiore ($id_arco + n_archi - 1$).

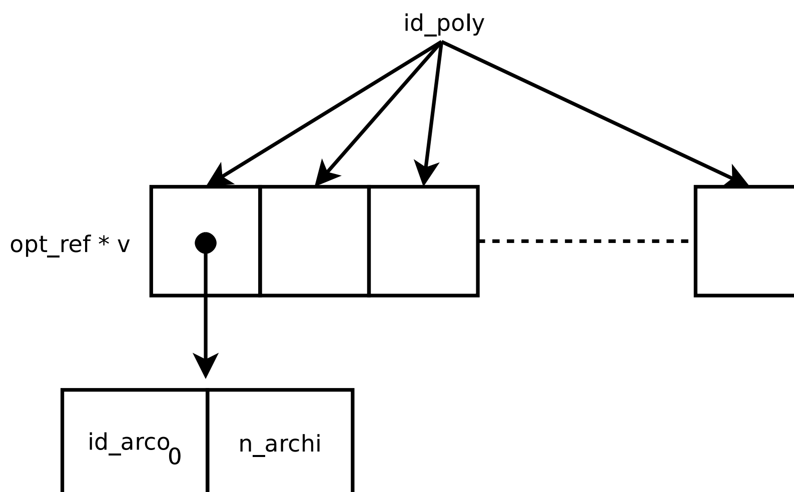

Figura 1.5: Struttura dati *opt_ref*

Analisi sulla complessità computazionale delle tre funzioni

Per verificare la buona rispondenza del modulo id_poly_ref, si effettua una prova sulla rete stradale di Roma. Dati 274674 nodi, 1032251 archi e 329249 polilinee, ipotizziamo di dover individuare il cursore del primo arco con *id_poly* uguale a quello del nodo. Ponendo n numero di archi e m numero di polilinee, si creano le funzioni $testVelocita$, $testVelocita_log$ e $testVelocita_ref$ che ricercano nel vettore *Arco* l'id_poly corrispondente alla polilinea di appartenenza. Si descrivono, di seguito, gli algoritmi implementati per effettuare i test e se ne analizza la complessità computazionale.

- **testVelocita** - La funzione effettua la ricerca dell'arco partendo dal primo elemento di V_a, fino a quando trova l'*id_poly* corrispondente a quello della polilinea. Nel caso pessimo, che ha come complessità computazionale $O(n)$, visita tutti gli archi del vettore. Applicandola su tutte le polilinee, la complessità computazionale diventa $O(nm)$.

- **testVelocita_log** - La funzione effettua una ricerca binaria sul vettore *Arco*, che sfrutta gli *id_poly* ordinati e si basa sull'importante tecnica divide et impera, usata per progettare algoritmi efficienti. La tecnica consiste nello scomporre il problema in sottoproblemi più piccoli dello stesso tipo, per risolvere e ottenere la soluzione di quello originale. Si analizza l'$id_poly(i)$, che occupa la posizione centrale del vettore e si compara con il valore (k) da cercare. Pertanto nel caso di:

  - $(i = k)$ l'algoritmo termina con successo;
  - $(i > k)$ l'algoritmo va in ricorsione sulla prima metà del vettore;
  - $(i < k)$ l'algoritmo va in ricorsione sulla seconda metà del vettore.

  Per calcolare la complessità computazionale degli algoritmi di tipo divide et impera, in cui il problema originario di dimensione n è diviso in a sottoproblemi di dimensione b ciascuno, si usa il teorema delle ricorrenze lineari con partizionamento bilanciato. Questo dice che, se si dividono i dati e/o si ricombinano i risultati in tempo polinomiale, la funzione di complessità $T(n)$ può essere espressa nei termini $aT(\frac{n}{b})+cn^\beta$. I parametri a e b sono costanti intere, tali che a $a \geq 1$ (si esegue sempre almeno una chiamata) e $b \geq 2$ (il problema è suddiviso sempre in almeno due sottoproblemi), mentre c e β sono due costanti reali, tali che $c > 0$ e $\beta \geq 0$.

Teorema 1. *siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$ e c, d e β costanti reali tali che $c > 0$, $d \geq 0$ e $\beta \geq 0$. Sia $T(n)$ data dalla relazione di ricorrenza:*

– $T(n) = d$, *per* $n = 1$

– $T(n) = aT(\frac{n}{b}) + cn^{\beta}$, *per* $n > 1$

Si pone $\alpha = \dfrac{\log a}{\log \beta}$, *allora vale che:*

– $T(n)$ *è* $O(n^{\alpha})$, *se* $\alpha > beta$;

– $T(n)$ *è* $O(n^{\alpha} \log n)$, *se* $\alpha = beta$;

– $T(n)$ *è* $O(n^{\beta})$, *se* $\alpha < beta$.

Per applicare il teorema nel caso dell'algoritmo di ricerca binaria, è necessario trovare i valori a, b e β. Il valore a si desume dal fatto che l'algoritmo va in ricorsione solo su una delle due metà del vettore ($a = 1$). Il valore b si desume dal fatto che ad ogni chiamata ricorsiva la lunghezza del vettore si dimezza ($b = 2$). Il valore β è posto a zero, perché si considera che l'algoritmo è eseguito solo per trovare un unico *id_arco* ($\beta = 0$).

Si ottiene dunque che:

– $T(n) = d$, per $n = 1$

– $T(n) = T(\frac{n}{2}) + 1$, per $n > 1$

Se $\alpha = \frac{\log 1}{\log 2}$ viene comparato con β, allora la complessità computazionale di questo algoritmo è $O(\log n)$. Quindi, la funzione di ricerca binariasi si può applicare a tutte le polilinee con complessità computazionale $O(m \log n)$).

- **TestVelocita_ref** - La funzione esegue la ricerca dell'*id_poly*, come nei test sopra descritti, utilizzando il modulo di supporto *id_poly_ref* anziché il vettore *Arco*. Il modulo restituisce, in tempo costante $O(1)$, l'id_arco_0 corrispondente all'*id_poly* cercato. Applicando la funzione su tutte le polilinee da ricercare, la complessità computazionale è uguale a $O(m)$. Se si considera anche il costo computazionale per inizializzare la struttura dati, la complessità diventa $O(m + n)$.

Conclusione

Dall'esecuzione separata dei test sopra descritti, si analizzano i tempi di esecuzione: $testVelocita = 6$ minuti circa; $testVelocita_log = 3$ secondi circa; $testVelocita_ref = 1$ secondo circa.

Dal $testVelocita$ risulta che, anche se non si impegna memoria centrale aggiuntiva (in quanto la ricerca avviene in modo iterativo), la sua complessità computazionale $O(mn)$ è molto elevata e i tempi di risposta sono troppo lunghi. Dal $testVelocita_log$ emerge che la ricerca dell'arco con id_arco_0 risulta spesso imprecisa, in quanto, con la probabilità di errore $1-(p)$ dove p è uguale al numero di archi della polilinea, restituisce un valore ide casuale compreso tra 0 e $p-1$ (estremi inclusi). Ciò è dovuto al fatto che l'id_poly, presente p volte in una polilinea, non è la chiave primaria di ricerca del vettore $Arco$. Per ottenere questa informazione l'algoritmo visita a ritroso gli archi della polilinea, fino a raggiungere quello con id_arco_0. E per restituire la quantità complessiva di archi presenti nella polilinea stessa, percorre i restanti archi non visitati, fino a raggiungere quello con $id_arco_p = -1$. D'altronde risulta che l'algoritmo non impegna memoria centrale aggiuntiva, in quanto la ricorsione avviene in modalità tail recursion. Dal $testVelocita_ref$, invece, risulta che la funzione $getPolyRef$ del modulo restituisce in tempo costante l'id_arco_0 e il numero di archi che compongono la polilinea. Ma il modulo, nell'allocare i dati inerenti il legame tra id_arco e id_poly nel vettore opt_ref , utilizza la memoria centrale aggiuntiva. Per stabilire il più conveniente utilizzo in alternativa all'algoritmo di ricerca binaria, occorre ulteriormente verificare l'incidenza del vettore opt_ref nella memoria centrale del sistema, che come detto è costituito da 329249 celle. Il calcolo dell'incidenza, utilizzando un sistema con architettura a 32 bit, evidenzia che la struttura dati richiede complessivamente 8 byte e il vettore circa $2.5MB$ da allocare nella memoria centrale. Si tratta di un irrisorio impiego di memoria,che non inficia l'indubbio vantaggio dato dalla velocità di esecuzione dell'algoritmo. Pertanto l'uso del modulo id poly ref risulta il più efficiente.

1.3 Integrità della rete stradale

Si implementa l'algoritmo $associaNodi$ per legare i nodi estremi di V_n agli estremi delle polilinee di V_p, determinandone nel contempo le coordinate geografiche. Per ogni nodo estremo $u \in V_n$ vengono considerati i cursori delle polilinee $P = \{p_0, p_1, \ldots, p_{k-1}\}$ e quelli inerenti i nodi vicini $V = \{v_0, v_1, \ldots, v_{k-1}\}$. Uno o entrambi i nodi estremi delle polilinee presenti in P hanno in comune le coordinate che li connette ad u. Per ogni

$i = 0, \ldots, k-1$, se il valore di p_i è positivo, il posizionamento di u è determinato dalle coordinate del nodo di testa della polilinea $|p_i|$ e se è negativo, da quelle di coda; ciò si applica a tutti gli elementi di P. Pertanto, sia $W = \{w_0, w_1, \ldots, w_{k-1}\}$ l'insieme di nodi che definisce il posizionamento di u; allora per ogni $i, j = 0, \ldots, k-1$, con $i \neq j$, se $w_i = w_j$ implica $u = w_0$, altrimenti si diagnostica un'anomalia di primo tipo.

Inoltre u contiene l'informazione dei due nodi estremi di p_i:

- se $p_i > 0$, il nodo estremo iniziale di p_i è uguale a u e quello finale è uguale a v_i;

- se $p_i < 0$, il nodo estremo iniziale di $|p_i|$ è uguale a v_i e quello finale è uguale u.

Per simmetria il nodo vi deve contenere l'informazione speculare presente in u:

- $p_i > 0$ in $u \Longrightarrow p_i < 0$ in v_i e *id_vicino* di $|p_i|$ è u;

- $p_i < 0$ in $u \Longrightarrow p_i > 0$ in v_i e *id_vicino* di p_i è u.

Se non è soddisfatta quest'ultima proprietà si verifica un'anomalia di secondo tipo, che impedisce l'attribuzione delle coordinate di u e la relativa determinazione dei nodi estremi delle polilinee ad esso connesse. Pertanto u e P sono marcati come *incoerenti*, rendendo così nulla l'associazione dei nodi di p_i. Ma, previa l'associazione dei nodi conformi alle proprietà del grafo, l'algoritmo può anche fornire una soluzione possibile per correggere dette incoerenze. A tal proposito ricerca le polilinee anomale e per ognuna di queste, confronta le coordinate geografiche dei nodi estremi non associati con quelle del grafo $\{u_0, u_1, \ldots, u_{n-1}\}$, in base al seguente criterio: Siano w e u due nodi il cui posizionamento geografico è mappato in un sistema di riferimento cartesiano (x, y). Allora w coincide con u se e solo se la distanza tra le singole componenti cartesiane è inferiore ad un certo $\epsilon > 0$ fissato.

$$|w_x - u_x| < \epsilon \quad |w_y - u_y| < \epsilon$$

La coincidenza tra u e w si verifica quando u cade nell'area quadrata centrata in w di perimetro pari a 8ϵ (figura 1.6).

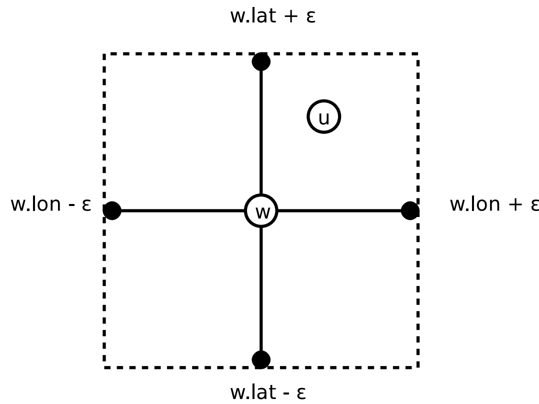

Figura 1.6: Area quadrata centrata in w su cui cade il nodo u

La scelta del paramentro epsilon influisce sulla precisione della soluzione restituita dall'algoritmo. Nel caso in cui ϵ sia un valore troppo grande, potrebbero esserci più nodi $C = \{c_0, c_1, \ldots, c_{h-1}\}$ coincidenti con w. Il migliore è il nodo $c_i \in C$, la cui distanza euclidea di ogni elemento di C con w è inferiore rispetto a quella degli altri nodi $\{c_0, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{h-1}\}$.

1.3.1 Incoerenze della rete stradale di Roma

Ciò premesso, si esamina la coerenza del grafo della rete stradale di Roma. Si attiva l'algoritmo *associaNodi*, che fornisce una lista di errori di primo e secondo tipo. Per comodità espositiva, si approfondisce di seguito un caso di errore di primo tipo, quello del nodo 14987, la cui correzione passa attraverso la rilevazione di un errore di secondo tipo. L'anomalia di primo tipo, infatti, evidenzia un'incoerenza del nodo nelle coordinate che risultano impredicibili, ossia la posizione dei nodi estremi delle polilinee, connesse tra loro, non combaciano. Si procede alla comparazione di quanto indicato dai cursori del nodo con le coordinate delle polilinee, per rilevare l'informazione non coerente. Si riporta, di seguito, la tabella coi valori dei cursori delle polilinee connesse al nodo 14987 e dei relativi nodi vicini:

id_poly	*id_vicino*
$-q = -329222$	1357
$p = 329223$	$u = 14987$
$-p = -329223$	$u = 14987$

Si definiscono con P e Q le rispettive polilinee identificate dai cursori $p = 329223$, $q = 329222$ e con U il nodo definito dal cursore $u = 14987$.

Dalla tabella sopraindicata si deduce che:

- U unisce le polilinee P e Q, dove la prima è connessa con entrambi gli estremi, mentre la seconda solo con quello di coda;

- P è una rotonda chiusa il cui unico sbocco è U;

- mentre Q è una strada che permette di accedere a P.

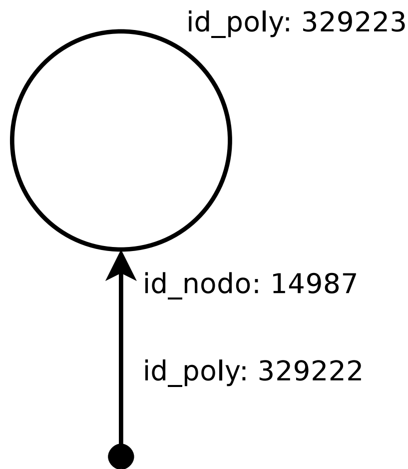

Figura 1.7: Anomalia riscontrata nel nodo 329223

U definisce P come una catena ciclica formata dai nodi $\{a_0, a_1, \ldots, a_{n-2}, a_0\}$. Ma ciò è in contrasto con le coordinate di P che, invece, la descrivono come una catena aciclica, con nodi estremi non coincidenti. L'informazione contenuta nei file nodi e *poly_pnts* è chiaramente contraddittoria. Per individuare l'errore, si esamina l'immagine satellitare della strada oggetto dell'incoerenza.

Figura 1.8: Comparazione del grafo con la corrispondente immagine satellitare

Dall'analisi delle traiettorie espresse da P e Q in corrispondenza delle coordinate dei nodi estremi, si osserva che:

- U connette Q con il nodo di coda;
- U connette P con il nodo di testa;
- la coda di P non si connette ad U (linea circolare tratteggiata)

- le coordinate di P e Q (indicate dalla linea continua) sono quelle attinenti alla realtà, in quanto coincidono perfettamente con

- l'immagine satellitare della strada.

Pertanto rimuovendo da U i cursori di P, si rettifica l'erronea informazione fornita dal file nodi.

Tabella dei cursori del nodo 14987

id_poly	*id_vicino*
$-q = -329222$	1357

La rimozione dei nodi estremi anomali rende P una polilinea fantasma, in quanto non è più contenuta nel file *nodi*. Si riapplica l'algoritmo $associaNodi$ che, rilevando la mancata associazione dei nodi estremi di P come errore di secondo tipo (v.*), ne fornisce le possibili soluzioni. Si stabilisce il parametro *epsilon*, che dà origine ai nodi estremi coincidenti con P, pari alla distanza minima tra le coordinate rappresentate nel formato di sei gradi decimali ($\epsilon = 0.000001$). In questo modo la posizione del nodo candidato deve corrispondere con quella di uno dei nodi estremi di P, a meno di un fattore di imprecisione trascurabile. L'algoritmo restituisce U, come prima soluzione, ma inaspettatamente non trova l'altro estremo con le stesse coordinate del nodo di coda. Perciò, si inserisce nel file *nodi* un nuovo nodo estremo V, di cursore pari a $v = 274667$, che connette la coda di P associandola al relativo nodo vicino U.

Tabella dei cursori del nodo 274667

id_poly	*id_vicino*
$-p = -329223$	$u = 14987$

Si aggiunge p ai cursori di U, il cui nodo vicino è V e si completa la correzione di U.

Tabella dei cursori del nodo 14987

id_poly	*id_vicino*
$-q = -329222$	1357
$p = 329223$	$v = 274667$

Riattivando nuovamente l'algoritmo si rileva un altro errore di primo tipo su V. Infatti, esaminando l'immagine satellitare (vedi figura 1.9), risulta che V connette P ad un'altra polilinea X, il cui cursore non è presente in V. Per procede alla correzione della nuova anomalia, si applica su V e X lo stesso metodo utilizzato per risolvere l' errore *diprimotipo* precedentemente riscontrati su U e P. Iterando il processo, fin qui descritto, a tutte le anomalie riscontrate nella rete stradale di Roma, si determina un grafo coerente, dove ogni polilinea è associata correttamente ai rispettivi nodi estremi.

1.4 La rete veicolare

La rete stradale fin qui analizzata è una buona base di partenza per poter informatizzare anche il flusso veicolare presente in essa. Per questo occorre associare alla rete stradale i dati relativi alle informazioni inerenti il movimento dei veicoli sul territorio. Tali informazioni, fornite da un apparato GPS, vengono trasmesse da un satellite a un centro di raccolta dati per, poi, essere informatizzate. Il segnale fornisce i dati relativi alla posizione geografica sul territorio, alla velocità e alla direzione del moto. I dati relativi alla posizione, però, sono soggetti a imprecisioni e di conseguenza determinano un'errata collocazione dei veicoli. Analizzando la loro distanza dalla strada di attinenza, si calcola lo scarto quadratico medio, pari a $\theta = 8m$ (vedi 1.5) e si approssima la loro distribuzione sul piano, con una funzione gaussiana la cui varianza è $(8m)^2 = 64m$. Dall'esame del grafico della funzione, si evince che il margine di errore posizionale di un veicolo dalla strada di attinenza, con la probabilità di 0.998, non è superiore a 5θ (vedi appendice C) ed è pari a un raggio variabile r compreso tra $0m \leq r \leq 40m$. Anche i dati legati alla direzione del moto contengono errori, che sono inversamente proporzionali alla velocità del veicolo. Se il veicolo è fermo o viaggia a velocità ridotta (inferiore a $5km$), l'informazione sulla direzione è dubbia, in certi casi completamente errata.

1.4.1 Strutture dati

Le informazioni satellitari vengono memorizzate in un file di estensione *xyv*, i cui dati sono letti e allocati in un vettore di tipo *str_trk*, mediante la funzione *getDatiTrk*. Questa, durante l'operazione di lettura dei dati, individua anche quelli relativi alla direzione del moto, i cui valori non essendo direttamente applicabili in trigonometria, li ricalcola effettuando su di essi l'operazione $90° - \alpha$, dove α è il dato GPS della direzione. Ciò è dovuto al

fatto che il segnale GPS misura la direzione del moto in senso orario a partire dal nord. Pertanto, se un veicolo viaggia parallelamente rispetto all'asse delle y (identificato dal nord), con angolo 0°, l'angolo ricalcolato è $90° - 0° = 90°$. Si illustra il codice che implementa la struttura *str_trk*:

```
typedef struct{
  double x;
  double y;
  int velocita;
  int angolo;
  int strada;
  char verso;
}str_trk;
str_trk * getDatiTrk(int * n_byte, char * filename);
```

Gli attributi della struttura sono:

- (x, y), indicano la posizione cartesiana del veicolo;
- *velocita*, indica la velocità del veicolo in km/h;
- *angolo*, indica il grado di angolazione assunto dal senso di marcia del veicolo rispetto al meridiano terrestre di riferimento;
- *strada*, indica il cursore dell'arco sul quale il veicolo viaggia;
- *verso*, indica il senso di marcia del veicolo, rispetto all'arco di appartenenza, con i simboli $>$ (per specificare il transito che va dal nodo di testa al nodo di coda) e $<$ (per specificare quello opposto).

Il campo relativo alla posizione (x, y) è ottenuto effettuando un mapping delle coordinate longitudinati e latitudinali in un sistema di riferimento cartesiano bidimensionale. Il campo inerente la direzione del moto è letto dal file *xyv*, mentre strada e verso sono campi derivati, inizialmente settati con i rispettivi valori di default -1 e e. L'attributo strada permette di rilevare l'errata posizione di un veicolo non attinente ad alcun arco, come nel caso di una polilinea non presente nella rete stradale, mentre verso permette di rilevare il senso di marcia del veicolo rispetto all'arco di percorrenza.

1.5 Associazione veicolo – arco

L'esistenza di un collegamento tra gli archi della rete stradale e gli ipotetici veicoli che vi transitano, permette di determinare i cursori degli archi

relativi al flusso veicolare. I valori ottenuti, inseriti nei rispettivi campi strada della struttura dati *str_trk*, formalizzano l'attinenza tra gli archi e i veicoli. Il grado di attinenza tra un veicolo v ad un arco a, è dato dalla funzione $f(v, a)$ che analizza la posizione e la direzione tra v e a. Si esamina la sovrapposizione delle immagini della rete stradale con quella veicolare. Si nota che la maggior parte dei veicoli si discosta dalla strada di percorrenza. Conseguentemente laddove c'è un'intensa densità di strade, risulta impossibile determinare l'appartenenza di un veicolo a una o un'altra strada. E ciò dipende dal non sempre preciso sistema di rilevamento GPS, che spesso fornisce imperfette coordinate geografiche. Infatti, dato un veicolo v percorrente l'arco a_1. Se, per errore, il GPS pone v equidistante da a_1 e da un altro arco a_2, con le sole coordinate non è, in alcun modo, possibile determinarne l'arco di percorrenza.

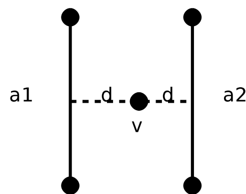

Figura 1.9: Veicolo v equidistante dagli archi a_1 e a_2

Per avere una stima dell'errore, si prendono in esame tutti i veicoli vicini a un arco isolato in modo che, non avendo altri archi vicini, si è certi che i veicoli adiacenti, anche se posizionati male, appartengono a quell'arco. Sia $V = \{v_0, v_1, \ldots, v_{n-1}\}$ l'insieme dei veicoli presi in esame appartenenti a un arco a. Allora, l'errore di posizionamento medio tra V e a è dato da $d_m = \frac{\sum_{i=1}^{n} d(v_i,a)}{n}$ pari a 8 metri. La migliore attinenza tra un veicolo e un arco non è determinato solo dalla posizione geografia, ma anche dalla direzione α del moto, pur essa soggetta a errori. Perciò si ricava l'errore medio $\alpha_m = \frac{\sum_{i=1}^{n} s(v_i,a)}{n}$ degli scarti degli angoli tra V e a, pari a 7°. Nell'associare i veicoli sotto i $4\frac{Km}{h}$ si tiene conto solo di d_m, in quanto la valutazione della direzione non è affidabile per piccole velocità. Gli errori medi consentiti sulle rilevazioni GPS permettono di implementare la funzione di affinità $f(v, a)$, che restituisce l'indice k di attinenza tra un veicolo e un arco. Siano $d = d(v, a)$ e $s = s(v, a)$ le funzioni che restituiscono rispettivamente la distanza e lo scarto di angolo tra v e a. Si dividono i valori d e s rispettivamente per gli errori medi sulla distanza d_m e sulla direzione α_m, al fine di stabilire l'attinenza tra v e a. Il valore 0 rappresenta il massimo grado di affinità raggiungibile. Si compara d con m dove il valore m, pari all'affinità minima

della distanza accettabile, è dato dal rapporto tra il raggio di errore GPS ($r = 40m$) e d_m. Se $d \geq m$, si determina la non affinità tra v e a con $k = -0.1$; invece, se $d < m$, si determina l'attinenza complessiva tra v e a, combinando d e s in un unico indice k:

$$k = \frac{\frac{1}{((1+d)(1+s))} - k_{min}}{1 - k_{min}}, \qquad k \leq 1$$

$$k_{min} = \frac{1}{1 + \frac{r}{d_m}}$$

Il valore $kmin$ indica il grado di affinità minimo per considerare un veicolo attinente all'arco. Più il valore k si avvicina a uno, più il livello di affinità tra v e a cresce.

1.6 L'arco rispetto alla direzione del moto del veicolo

L'indice di attinenza veicolo - arco, restituito dalla funzione di affinità, viene dato dalla comparazione tra la posizione nello spazio e il senso di marcia di entrambi. Per stabilire, invece, in che direzione un veicolo v percorre un arco a, si realizza un algoritmo che controlla la compatibilità del senso di marcia tra v e a.

Si ricava l'angolo in radianti α_s dell'arco, applicando l'arcotangente al rapporto tra la sua proiezione sulle ascisse con quella sulle ordinate, in quanto la classe Arco non contiene l'informazione sull'angolo dell'arco rispetto alle ascisse. Il dato del campo angolo di ogni veicolo, espresso in gradi, si converte in radianti α_v per allinearlo all'angolo α_s , anch'esso espresso in radianti. Dalla differenza tra α_v e $\alpha_s (\alpha_d = \alpha_v - \alpha_s)$, si ricava la direzione del moto del veicolo rispetto a quello dell'arco. Si calcola $p = \cos \alpha_d$, che restituisce i valori $-1 \leq p \leq 1$ per stabilire la direzione del veicolo.

Nei casi limite p assume i seguenti valori:

- 1, la direzione del moto del veicolo, che va dal nodo di testa a quello di coda ($\alpha_d = 0°$), è parallela all'arco;

- -1, la direzione del moto del veicolo, che va dal nodo di coda a quello di testa ($\alpha_d = 180°$), è parallela all'arco;

- 0, la direzione del moto del veicolo ($\alpha_d = 90°$) è perpendicolare all'arco, per cui non è possibile stabilire il senso di marcia;

Da ciò si evince che il senso di marcia del veicolo, rispetto a un arco di nodi (u, v), è dato dal segno di p; se è positivo, il veicolo percorre l'arco da u a v, altrimenti da v a u.

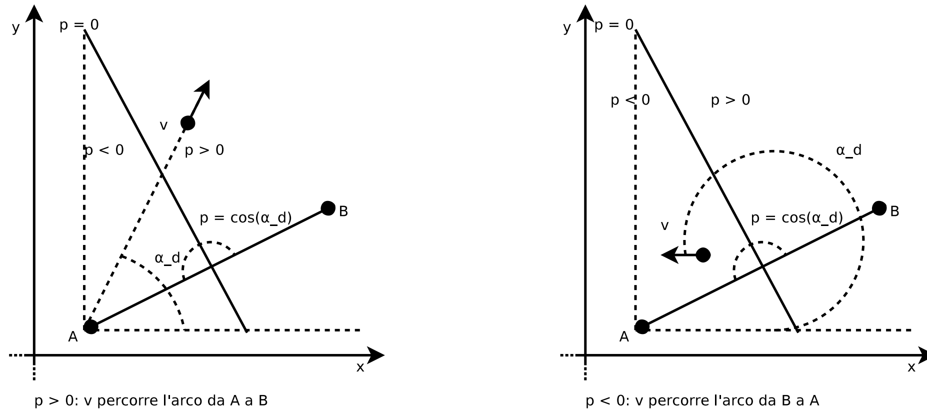

Figura 1.10: Determinazione dei sensi di percorrenza

La figura 1.10 mostra rispettivamente i casi in cui il veicolo v percorre l'arco da A a B e viceversa. Nella prima ipotesi si ha lo scarto di angolo α_d tra v e $AB(0° \leq \alpha_d < 90°) \vee (270° < \alpha_d \leq 360°)$ e conseguentemente il cos α_d è positivo, mentre nella seconda si ha $90° < \alpha_d < 270°$ e il $cos\alpha_d$ è negativo. Si compara p con l'informazione del senso di marcia s dell'arco:

- se $s = 0$ e $((p > 0) \vee (p < 0))$, il veicolo percorre l'arco $(u, v)^{longleftrightarrow}$ in modo corretto;

- se $s = 1$ e $p > 0$, il veicolo percorre l'arco $(u, v)^{\rightarrow}$ in modo corretto, altrimenti è in una posizione errata;

- se $s = 2$ e $p < 0$, il veicolo percorre l'arco $(u, v)^{\leftarrow}$ in modo corretto, altrimenti è in una posizione errata.

Nel caso la percorrenza del veicolo risulti compatibile con quella dell'arco, il campo verso è definito con i simbolo $>$ (se $p > 0$), oppure con $<$ (se $p < 0$); e nel caso contrario di errore è definito con il simbolo e.

Capitolo 2

L'editor della rete stradale

Gli algoritmi e le strutture dati, illustrati nel capitolo precedente, sono basilari per ridefinire l'informazione territoriale. In questo contesto, grande rilevanza assume il processo di sovrapposizione delle reti stradale-veicolare, utilizzato per il rilevamento dei flussi veicolari anomali e la rispettiva correzione del grafo. Uno degli obiettivi di questa tesi è quello di realizzare un sistema informatico che, partendo da tali presupposti, consenta di operare agevolmente sul grafo di una rete stradale, nella fattispecie quella urbana di Roma e di apportare in automatico l'inserimento delle polilinee mancanti, evidenziate da un flusso veicolare GPS anomalo. Da qui l'idea di creare un editor capace di visualizzare e interagire graficamente sulle informazioni. Si illustrano, di seguito, gli studi e le tecniche per realizzarlo.

2.1 La struttura dell'editor

Esaminando il dominio del problema, s'implementa un primo prototipo, che si basa su una componente fondamentale denominata *Display*. *Display* visualizza a schermo l'immagine della sovrapposizione delle reti stradale-veicolare e consente di interagire su di essa. Per riprodurre fedelmente le immagini, utilizza un'apposita libreria (*OpenGL*), le cui funzioni permettono di creare in ambiente Linux e Windows le strade e i veicoli con apprezzabili risultati grafici. Le attività interattive di visualizzazione e manipolazione della scena grafica, però, si concretizzano mediante un'altra libreria (*Glut*), adatta a gestire gli input provenienti dall'esterno. La complessità delle attività e dei servizi che implementa, viene fornita da altre componenti esterne ad essa. Ciò dà alla struttura una dinamicità e duttilità tale da recepire, con irrisorie modifiche, gli eventuali cambiamenti del sevizio. Se la modifica di

un servizio interattivo si riferisce a una componente satellite già esistente, la variazione viene recepita da *Display* senza dover effettuare alcuna rettifica. Invece, quando nell'editor si aggiunge ex novo una componente, per renderla attiva occorre inserire i metodi, inerenti al nuovo sevizio, nella apposite chiamate callBack di *Display*. Queste ricevono l'input esterno e lo passano ai metodi che forniscono i servizi per modificare il grafo e l'immagine del display. A tal proposito si utilizzano due framework. Il primo fornisce un'api che astrae le operazioni di manipolazione, necessarie per apportare le modifiche alle polilinee e agli archi del grafo. Il secondo fornisce un'altra api che astrae le operazioni di basso livello, per creare gli oggetti grafici nel display. Non tutte le componenti satellite lavorano contemporaneamente. Solo quelle che acquisiscono gli input esterni e processano le relative informazioni sono in stato di *run*, mentre quelle dormienti si considerano *wait*. Le componenti che implementano un'interazione complessa formata da n fasi, hanno n stati di $run = (run_1, run_2, \dots, run_n)$, ognuno relativo alla corrispondente fase. Gli stati di *run* e *wait* sono settati dalla chiamata di callback *setAzione* mediante la componente *Menù*, che mette a disposizione di *Display* una serie di finestre di comando, utili ad attivare tutte le funzionalità interattive delle componenti satellite. La parte grafica di *Menù* utilizza la libreria *GTK+2*, la cui interfaccia è perfettamente compatibile con i sistemi operativi di Linux e Windows.

In conclusione i servizi interattivi sono forniti dalle componenti satellite. I loro metodi sono utilizzati da *Display*, che ha il compito di inizializzare la scena grafica e passare gli input esterni. Queste componenti includono, inoltre, i framework che consentono una facile manipolazione del grafo della rete stradale e un agevole utilizzo della grafica. *Menù* è mutualmente incluso nel *Display*, in modo da attivarne le funzionalità e riceverne eventuali risposte.

2.2 Le librerie grafiche

OpenGL (*Open Graphics Library*) è utilizzata per creare applicazioni grafiche bidimensionali o tridimensionali. Ha un'interfaccia che, servendosi di semplici primitive, rappresenta con punti e linee le coordinate vettoriali, e poi le rasterizza per convertirle in un'immagine bidimensionale, ossia in una griglia di pixel. Le coordinate subiscono una serie di trasformazioni, fino a essere visualizzate nel formato raster[7]. Dato un vertice $V = (x, y, z) \in \mathbb{R}^3$, espresso mediante un sistema di riferimento locale, si eseguono in pipeline le seguenti funzioni f_1, f_2, f_3, f_4 per trasformare V nel raster.

- f_1, *trasformazione di modellazione.* Trasforma V in un vertice $V' = (x', y', z')$, le cui coordinate, dette "del mondo", appartengono al sistema di riferimento globale comune a tutta la scena.

- f_2, *trasformazione di vista.* Trasforma V' in un vertice $V'' = (x'', y'', z'')$, le cui coordinate, dette "di vista", definiscono la scena vista dalla camera.

- f_3, *trasformazione di proiezione.* Trasforma V'' in un vertice $V''' = (x''', y''', z''')$ tale che $-1 \leq x''', y''', z''' \leq 1$, le cui coordinate, dette "di proiezione", definiscono la scena in base alla visione parallela o prospettica e al volume di vista.

- f_4, *trasformazione di viewport.* trasforma V''', nella corrispondente visualizzazione raster.

Le trasformazioni di modellazione e di vista sono espletate da *OpenGL* in base ai valori contenuti nella matrice *GL_MODELVIEW*, mentre quella di proiezione in base ai valori della matrice *GL_PROJECTON*.

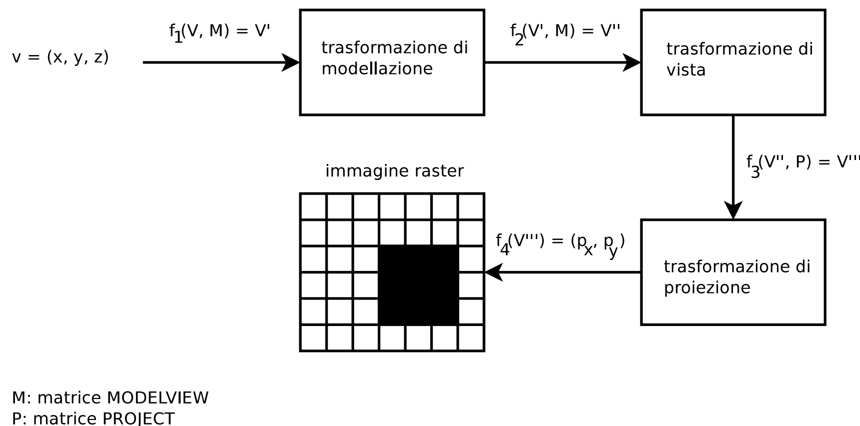

Figura 2.1: Trasformazioni vertici in *OpenGl*

Per visualizzare e gestire sullo schermo quanto elaborato da *OpenGL* è necessario usare un buffer video, che operi da interfaccia tra *OpenGL* e il windows manager del sistema operativo. A questo scopo si utilizza la libreria *GLUT* (*OpenGL Utilities Toolkit*), le cui funzioni consentono di realizzare delle finestre dove, mediante i meccanismi di callback, è possibile contenere e gestire le scene grafiche.

2.3 Il display dell'editor

La classe *Display* inizializza una finestra grafica vuota, predisposta per riprodurre e visualizzare graficamente l'elaborazione della sovrapposizione delle reti stradale-veicolare. Con essa interagiscono una complessità di attività e servizi, forniti da componenti esterne, che vengono avviati dai relativi metodi contenuti nelle apposite chiamate callBack.
A tal fine operano le seguenti funzioni *Glut*:

- *startdisplay*, carica i dati delle reti stradale-veicolare;
- *display*, resetta la scena grafica;
- *resize*, ridimensiona la finestra;
- *keypress*, determina la posizione della camera sulla scena;
- *mousemove*, determina gli input derivanti dal movimento del mouse;
- *mouseclick*, determina gli input derivanti dal click del mouse;
- *mousedrag*, determina l'azione di trascinamento del mouse;
- $setAzione$,simula una chiamata di callback da parte del menù.

La funzione *startdisplay* si attiva all'avvio dell'editor. Carica i dati delle reti stradale-veicolare nel sistema (vedi capitolo 1.2) e li mette a disposizione dei moduli che li usano per fornire i servizi. La funzione *display* resetta lo schermo cancellando ogni eventuale scena grafica preesistente. Ciò avviene mediante il richiamo della funzione *glutPostRedisplay.* Per non sovrapporre i nuovi inserimenti grafici a quelli preesistenti utilizza le funzioni *glClear* e *glLoadEntity*, che hanno il compito di pulire la scena. Così eliminando tutti gli oggetti grafici, la camera torna sull'asse di origine $(0, 0, 0)$. Mediante la funzione *glLookAt*, ripristina la camera nella posizione precedente al reset e si predispone nuovamente all'uso, attivando i metodi dei servizi per inserire nuovi oggetti grafici. La funzione *resize(w, h)* ridimensiona la scena grafica, a seguito di una variazione della larghezza w o dell'altezza h della finestra. Utilizzando la funzione $glViewport$ setta la viewport, corrispondente all'area della finestra, con l'angolo in basso a sinistra di coordinate $(0px, 0px)$ e quello in alto a destra di coordinate (w, h). Mediante la funzione *gluPerspective* determina la vista prospettica simmetrica del tronco di piramide, come il rapporto tra w ed h. In questo modo le dimensioni delle componenti grafiche si adattano automaticamente alla finestra, evitando eventuali distorsioni

dell'immagine. Infine richiama la funzione *display* per effettuare i nuovi inserimenti. La funzione *Keypress(k)* si attiva con il tasto k della tastiera del pc e serve a gestire il vettore (x, y, z), che determina la posizione della camera sulla scena. Dato un valore ϵ, per definire lo spostamento della camera, si modifica il vettore come segue:

- tasto freccia sinistra $(x - \epsilon, y, z)$;
- tasto freccia destra $(x + \epsilon, y, z)$;
- tasto freccia in basso $(x, y - \epsilon, z)$;
- tasto freccia in alto $(x, y + \epsilon, z)$;
- tasto "meno" $(x, y, z - \epsilon)$;
- tasto "più" $(x, y, z + \epsilon)$.

Con i tasti freccia si sposta l'occhio della camera nel grafo in direzione "su", "giù", "destra", "sinistra" e con i tasti +/- aumenta o diminuisce lo zoom della scena. La modifica del vettore implica la chiamata di *glutPostRedisplay* che ridisegna la scena grafica. Le funzioni *mousemove*, *mouseclick* e *mousedrag*, determinano gli input derivanti dalle operazioni di selezione tasto, movimento e trascinamento del puntatore del mouse nella finestra. Hanno come parametri di input le coordinate in pixel (x, y), relative alla posizione corrente del puntatore. Contengono i servizi interattivi per determinare l'esistenza di un tratto stradale vicino a (x, y), evidenziandolo nelle corrispondenti chiamate presenti in *display*. Se il tratto da evidenziare esiste, restituiscono il valore *true*, che attiva il ridisegno della scena grafica. La funzione *setAzione(id, param[])* simula una chiamata di callback derivante dal menù, specificando come argomenti di input il codice del servizio *id* con annesso il vettore *param* di parametri forniti dall'esterno. In base all'*id* chiama il metodo del servizio associato e se *param* non è vuoto, gli fornisce in input i dati contenuti in esso.

2.4 Visualizzazione della rete stradale

La posizione degli elementi delle reti stradale-veicolare è espressa da coordinate geografiche (lon, lat), che rappresentano le distanze angolari in gradi decimali, rispettivamente dal meridiano di Greenwich e dall'equatore. Per rappresentare tali coordinate su un piano cartesiano è necessario eseguire una proiezione cartografica che le trasformi in coordinate (x, y)[8].

Le coordinate geografiche (lon, lat) sono espresse mediante l'ellissoide standard di riferimento *WGS-84* [16] per la cartografia GPS, che dal punto di vista matematico è uno sferoide oblato[15] con i seguenti parametri:

- $a = 6378137m$, lunghezza in metri del semiasse maggiore;
- $b = 6356752,3142m$, lunghezza in metri del semiasse minore.

Una volta fissati questi valori è possibile ricavare le equazioni che forniscono una trasformazione di coordinate per i punti sul geoide da polari (λ, φ) a cartesiane:

$$\begin{cases} x = \dfrac{a \cos\varphi \cos\lambda}{R} \\ y = \dfrac{a \cos\varphi \sin\lambda}{R} \\ z = \dfrac{a(1{-}e^2) \sin\varphi}{R} \end{cases}$$

dove $R = \sqrt{1 - (e^2)\sin^2\varphi}$ e il valore e è definito come l'eccentricità, pari a $\frac{a^2 - c^2}{a^2}$. Si possono ulteriormente ricavare delle equazioni più semplici che, data una latitudine φ, forniscano la sua conversione in metri:

$$\begin{cases} x = (1.11324 - 0.003733 \sin(\varphi))10000 \\ y = (1.11324 - 0.003733 \sin(\varphi)) \cos(\varphi)10000 \end{cases}$$

Applicando le suddette equazioni alle coordinate geografiche centrali di Roma si ottengono rispettivamente le distanze in metri dal meridiano di Greenwich C_{lon} e dall'equatore C_{lat}[13]. La prima permette di ottenere la distanza lineare in metri tra due longitudini, mentre la seconda quella tra due latitudini. La conoscenza delle coordinate geografiche minime, centrali e massime del grafo, permettono di effettuare una proiezione su un piano cartesiano la cui origine degli assi è $(0, 0)$. Siano $O = (lon_c, lat_c)$, $Min = (lon_{min}, lat_{min})$, $Max = (lon_{max}, lat_{max})$ le coordinate geografiche rispettivamente del centro, del vertice minimo e di quello massimo del grafo. La trasformazione di un punto $P = (lon, lat)$ in coordinate cartesiane (x, y) si effettua calcolando il rapporto tra la distanza P–O con quella Max–Min. Il risultato può essere ulteriormente diviso per una costante k, che ne determina la risoluzione.

$$\begin{cases} x = \dfrac{C_{lon}(lon–lon_c)}{\dfrac{C_{lon}(lon_{max} - lon_{min})}{k}} \\ y = \dfrac{(C_{lat}(lat–lat_c))}{\dfrac{(C_{lat}(lat_{max} - lat_{min}))}{k}} \end{cases}$$

Analogamente si può passare dal sistema di riferimento cartesiano a quello geografico con le equazioni inverse:

$$\begin{cases} lon = \dfrac{\dfrac{C_{lon}(lon_{max} - lon_{min})}{k}x}{C_{lon}} + lon_c \\ lat = \dfrac{\dfrac{C_{lat}(lat_{max} - lat_{min})}{k}y}{C_{lat}} + lat_c \end{cases}$$

Nelle classi *Arco* e *Nodo* s'implementano i metodi *normalizza* e *denormalizza*, che, in accordo alle equazioni sopraddette, trasformano le rispettive coordinate della rete stradale da geografiche in cartesiane e viceversa. All'avvio dell'editor si attiva la funzione *startdisplay*, che carica i dati delle reti stradale-veicolare. Inizializza i vettori V_p, V_n e V_a rispettivamente con le funzioni *leggiStradeElaborate*, *leggiNodiElaborati* e *creaArchi*. Esegue la funzione *associaNodi*, che associa alle polilinee i rispettivi nodi estremi e ai nodi le coordinate geografiche. Per ogni nodo di V_n e arco di V_a esegue la trasformazione delle coordinate geografiche in quelle cartesiane. Con la funzione *leggiVeicoli* inizializza il vettore V_c, contenente le coordinate cartesiane dei veicoli direttamente normalizzate dal file in fase di lettura. La funzione termina con l'attivazione di *glutPostRedisply*, che invoca la chiamata callback di display. Questa contiene al suo interno le funzioni *disegnaStrade* e *disegnaAuto* del framework *Disegno*.

Void disegnaStrade(V_a) inserisce gli archi nella scena grafica mediante una scansione completa del vettore, ossia per ogni $i = 0, \ldots, n-1$, con n numero di archi di V_a, esegue il seguente comando OpenGL, che disegna un segmento di vertici $(x_0, y_0), (x_1, y_1)$.

```
glBegin(GL\_LINES);
          glVertex3f(Va[i].x0, Va[i].y0, 0.0f);
          glVertex3f(Va[i].x1, Va[i].y1, 0.0f);
glEnd();
```

Allo stesso modo opera la funzione *disegnaAuto(V_c)* che, eseguendo una

scansione completa del vettore V_c, inserisce nella scena i punti corrispondenti alle coordinate cartesiane dei veicoli.

```
glBegin(GL\_POINTS);
        glVertex3f(Vc[i].x, Vc[i].y, 0.0f);
glEnd();
```

Al termine del processo la funzione *display* dà a video una buona e fedele riproduzione delle reti stradale-veicolare, sovrapposte correttamente su un piano cartesiano bidimensionale.

Figura 2.2: Sovrapposizione reti stradale-veicolare di Roma

L'azione dell'algoritmo per la visualizzazione della scena grafica è insufficiente a operare in un sistema dinamico interattivo di grosse dimensioni. Si esegue un test di prova, sulla rete stradale di Roma composta da circa un milioni di archi e centomila veicoli. Le funzioni *disegnaAuto* e *disegnaStrade* impiegano circa quattro secondi per aggiornare la scena grafica delle

reti stradale-veicolare. Poiché a ogni azione interattiva di modifica segue sempre una ridefinizione della scena grafica, questa soluzione risulta inidonea. Si ripete la prova solo sulla rete veicolare e si osserva che la funzione *disegnaAuto* impiega circa due secondi per aggiornare la scena. Considerando che i veicoli sono rappresentati da punti, si prova a inserire quelli che cadono nell'area descritta dalla viewport. Per fare ciò si utilizzano le funzioni *getdimensionefinestra* e *getcoordinatebymouse* del framework disegno, che restituiscono rispettivamente la dimensione in pixel della viewport (w, h) e le coordinate cartesiane di un pixel della finestra. Attivando getcoordinatebymouse sui pixel $(0, h)$ e $(w, 0)$ si ottengono i vettori $m = (x_m, y_m)$ e $M = (x_M, y_M)$, che definiscono l'area visualizzata dalla vieweport in coordinate cartesiane. Si modifica l'algoritmo *disegnaAuto*, in modo che il veicolo sia inserito nella scena solo se cade nell'area descritta da m e M. Ossia per ogni $i = 0, \ldots, n-1$ con n numero di veicoli di V_c, allora $Vc[i]$ è disegnato se e solo se $x_m \leq Vc[i].x \leq x_M \wedge y_m \leq Vc[i].y \leq yM$. Tale tecnica d'inserimento dei veicoli visibili dalla viewport rende impercettibile all'operatore il tempo di esecuzione dell'algoritmo, nonostante la complessità computazionale sia rimasta $O(n)$. Pertanto il ritardo non è determinato dalla scansione lineare di V_c, ma dall'api di OpenGL che inserisce il punto nella scena. Il netto miglioramento delle prestazioni di *disegnaAuto*, porta a implementare la stessa tecnica sul vettore di archi. Il problema è stabilire quando un arco di coordinate (x_0, y_0, x_1, y_1) cade all'interno dell'area descritta da m e M. Per stabilirne l'inserimento nella scena non basta controllare l'appartenenza di uno dei vertici dell'arco.

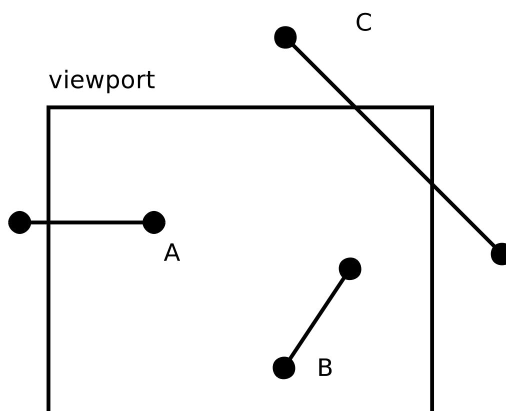

Figura 2.3: Archi visualizzati dalla viewport

Nell'esempio dato nella figura 2.3, gli archi A e B si considerano inclusi nella scena, in quanto uno dei loro vertici cade nell'area descritta dalla view-

port, mentre l'arco C, anche se una parte diversa dai vertici insiste nell'area, viene scartato. Il rilevamento di quest'ultima tipologia di archi si acquisisce previa verifica dell'esistenza di un loro punto d'intersezione con i lati dell'area della scena. In questo modo tutti gli archi che insistono nell'area si considerano nella scena. Aggiungendo all'algoritmo *disegnaStrade* dette modifiche, il processo d'inserimento degli archi nella scena risulta ancora lento (circa 1.4 secondi). Ciò è dovuto al fatto che per ogni arco di V_a si deve determinare l'intersezione sui quattro lati della viewport. Occorre creare una struttura, conforme a una griglia, capace di ospitare la rete stradale e di determinarne gli archi da inserire nella scena in un tempo inferiore a $O(n)$ e possibilmente costante $O(1)$.

2.5 La griglia della rete stradale

Si implementa il modulo *Griglia* che, suddivide il piano cartesiano in tante piccole celle disposte a reticolo, dove vengono memorizzati i cursori degli archi del grafo. Questo permette di trovare, in tempo costante $O(1)$, tutti gli archi appartenenti a una cella, su cui cade un punto p di coordinate cartesiane (x, y). Si definisce la struttura dati *str_griglia*, i cui campi sono:

- $(x_{min}, y_{min}) \in \mathbb{R}^2$, determinano le coordinate del punto in basso a sinistra della griglia;
- *risoluzione* $\in \mathbb{R}$, determina la lunghezza del lato di ogni cella della griglia;
- $w \in \mathbb{N}$, determina il numero di colonne della griglia;
- $h \in \mathbb{N}$, determina il numero di righe della griglia;
- *celle* $\in \mathbb{N}^{w \times h}$, definisce la griglia.

Si elencano di seguito le principali funzioni del modulo.

Void initgriglia(int w, int h, double x_min, double y_min, double r, str_griglia * g)

Inizializza il campo celle con un vettore di puntatori a intero di dimensione wh. Crea una griglia di celle vuote a cui assegna il valore di default *NULL*. Attribuisce al vertice in basso a sinistra le coordinate cartesiane (x_{min}, y_{min}) e al lato di ogni cella la lunghezza r. In questo modo determina, in funzione

di r, w e h, le coordinate degli altri tre vertici della griglia. Il punto in basso a destra è uguale a $(x_{min}+wr, y_{min})$, quello in alto a sinistra è $(x_{min}, y_{min}+hr)$ e quello in alto a destra è $(x_{min}+wr, y_{min}+hr)$.

int getidcella(double x, double y, str_griglia * g)

Restituisce in tempo costante $O(1)$ la posizione della cella del vettore celle su cui cade il punto $p = (x, y)$. Trasforma p in un altro punto $p' = (t_x, t_y) \in \mathbb{N}^2$, le cui coordinate rappresentano rispettivamente la riga e la colonna della cella di p.

$$t_x = \frac{x - x_{min}}{r}$$

$$t_y = \frac{y - y_{min}}{r}$$

Poiché la griglia è rappresentata con un vettore lineare di dimensione wh, commuta le coordinate bidimensionali (t_x, t_y) in quella corrispondente monodimensionale j, che identifica la cella p' per adeguarla al vettore *celle* della griglia.

$$j = (t_x - 1) + ((t_y - 1)w)$$

In questo modo si definisce una struttura che, dato un punto qualsiasi del piano, restituisce in tempo costante il contenuto della cella corrispondente su cui cade il punto stesso.

int getbordi(double x, double y, segment_2d segmento[4], str_griglia * g)

Restituisce in tempo costante le coordinate cartesiane degli archi che definiscono i lati della cella su cui cade un punto p. Con la funzione *getIdCella*, ricava l'indice j della cella su cui cade p e applica il sopradescritto metodo inverso, per determinare la riga e la colonna di j e il vertice (x, y) corrispondente.

$$\begin{cases} tx = j|w \\ ty = \dfrac{j}{w} \end{cases}$$

$$\begin{cases} x = t_x r + x_{min} \\ y = t_y r + y_{min} \end{cases}$$

Le coordinate (x, y) identificano il vertice v_0 in basso a sinistra della j-esima cella della griglia e conseguentemente gli archi s_0, s_1, s_2 e s_3 si ottengono in funzione di r (vedi figura 2.4).

$$s0 = (x, y, x, y{+}r), s1 = (x, y{+}r, x{+}r, y{+}r), s2 = (x{+}r, y{+}r, x{+}r, y), s3 = (x{+}r, y, x, y).$$

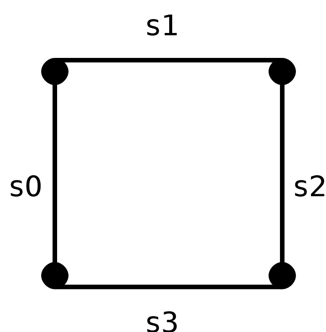

Figura 2.4: Segmenti dei bordi di una cella

void insptrincella(int j, int ptr, str_griglia * g)

Inserisce il cursore *ptr* nella cella j della griglia. Se la j-esima cella è vuota (*cella[j] == NULL*), inizializza un vettore di interi allocandovi due celle. Nella prima memorizza il valore 2, corrispondente alla dimensione del vettore e nella seconda il valore del cursore. Se la cella è piena (*cella[j] != NULL*), accede alla prima cella del vettore constatandone la dimensione. Verifica che il cursore non sia già presente nel vettore e in questo caso aggiunge in coda il valore del cursore. Infine aggiorna la prima cella del vettore con la sua nuova dimensione.

int inslinea(segment_2d segmento, int ptr, str_griglia * g)

Immette il cursore dell'arco $a = (u, v)$ nella griglia. Ricava l'indice j su cui cade u. Con la funzione *insptrincella* inserisce il cursore di a nella j-esima cella del vettore celle e con la funzione *getbordi* determina la posizione dei segmenti $S = s_0, s_1, s_2, s_3$ della cella j. Per ogni segmento di S controlla l'esistenza di un'intersezione con a. Se l'intersezione coincide esattamente con i vertici dei bordi, va in ricorsione sulle tre celle adiacenti al vertice interessato, altrimenti va sulla cella adiacente a j, dove si interseca a. Reitera tale operazione in tutte le celle intersecate, fino a quando il cursore non

raggiunge il nodo v. Applicando quanto detto su una immaginaria matrice di pixel, dove inserendo il cursore in un dato pixel si determina un cambio di colore, si osserva che l'algoritmo traccia esattamente un segmento che unisce il vertice u a v (figura 2.5).

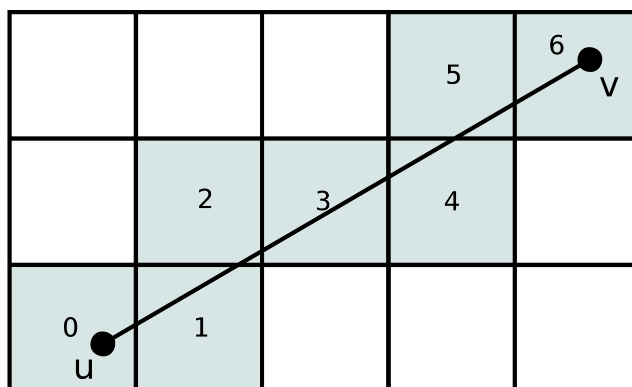

Figura 2.5: Inserimento di un arco nella griglia

int * getptrcellabyxy(double x, double y, str_griglia * g)

Richiama la funzione *getidcella* applicandola sul punto di coordinate (x, y), per restituire, in tempo costante, il vettore dei cursori della cella dove cade il punto stesso. L'inizializzazione della griglia avviene nella fase di caricamento dei dati delle reti stradale-veicolare, più precisamente dopo l'attivazione del vettore V_a e la conversione delle coordinate geografiche in cartesiane di ogni suo elemento. Si scandiscono tutti gli archi di V_a, al fine di determinare le coordinate minime $m = (x_{min}, y_{min})$ e massime $M = (x_{max}, y_{max})$, che descrivono l'area sulla quale è racchiuso il grafo. Si calcola la dimensione (w, h) della griglia, pari alla distanza tra m e M, divisa per la risoluzione r che si vuole dare.

$$\begin{cases} w = \left\lceil \dfrac{x_{max} - x_{min}}{r} \right\rceil \\ h = \left\lceil \dfrac{y_{max} - y_{min}}{r} \right\rceil \end{cases}$$

Con l'utilizzo dell'arrotondamento per eccesso si evita di lasciare scoperte porzioni di grafo. Ad esempio, se w è un numero compreso tra due interi $n_1 \leq w \leq n_2$, significa che esistono degli archi i cui vertici cadono nella cella est di n_1, che esiste solo se $w = n_2$.

Attivando la funzione *initGriglia* con i parametri di input (w, h), (x_{min}, y_{min}) e r si inizializza una struttura *str_Griglia* vuota di dimensione wh. Per ogni arco appartenente a V_a si attiva la funzione *inslinea*, che inserisce il suo cursore nelle celle della griglia su cui è posizionato. Tale procedura è lenta, ma ha ricadute vantaggiose di efficienza, in quanto permette agli algoritmi di effettuare la ricerca degli archi vicini ad un punto dato in input, in tempo costante $O(1)$. Si implementa, così, la funzione *disegnastradebygriglia* del framework disegno, che immette gli archi visibili nella viewport utilizzando la griglia. Come nel caso di *disegnastrade*, determina le coordinate $\{(v_{x_{min}}, v_{y_{min}}), (v_{x_{max}}, v_{y_{max}})\}$ e $\{(g_{x_{min}}, g_{y_{min}}), (g_{x_{max}}, g_{y_{max}})\}$ che definiscono i bordi dell'area V della viewport e di quella G della griglia. Effettua le seguenti comparazioni per determinare le coordinate $\{(v'_{x_{min}}, v'_{y_{min}}), (v'_{x_{max}}, v'_{y_{max}})\}$, corrispondenti all'area V' della griglia presente in V; se:

- $(v_{y_{min}} < g_{y_{max}} \wedge v_{y_{max}} > g_{y_{min}} \wedge v_{x_{min}} < g_{x_{max}} \wedge v_{x_{max}} > g_{x_{min}})$, pone $V' = V$, in quanto V si sovrappone a G. In caso contrario V e G sono due aree disgiunte;

- $v_{x_{min}} < g_{x_{min}}$, pone $v'_{x_{min}} = g_{x_{min}}$, in quanto il lato ovest di G non si sovrappone a quello di V;

- $v_{y_{min}} < g_{y_{min}}$, pone $v'_{y_{min}} = g_{x_{min}}$, in quanto il lato sud di G non si sovrappone a quello di V;

- $v_{x_{max}} > g_{x_{max}}$, pone $v'_{x_{max}} = g_{x_{max}}$, in quanto il lato est di G non si sovrappone a quello di V;

- $v_{y_{max}} > g_{y_{max}}$, pone $v'_{y_{max}} = g_{y_{max}}$, in quanto il lato nord di G non si sovrappone a quello di V.

Determina gli indici delle celle che definiscono la sotto-griglia presente in V, avviando la funzione *getidcella* su V'.

colonna_start = getidcella($v'_{x_{min}}$, $v'_{y_{min}}$)
colonna_end = getidcella($v'_{x_{max}}$, $v'_{y_{min}}$)
riga_end = getidcella($v'_{x_{min}}$, $v'_{y_{max}}$)

Pertanto il numero di colonne visibili c nella viewport è dato da *colonna_end* − *colonna_start*, mentre quello r di righe da *riga_end* − *colonna_start* . L'algoritmo scandisce le celle della griglia, partendo da *colonna_start* fino a

$colonna_start + c - 1$. Per ogni cella ricava la lista dei cursori degli archi e li inserisce nella scena grafica. Ripete la stessa operazione sulla riga $colonna_start + iw$ successiva della griglia, dove $i = 0, \dots, r - 1$ rappresenta l'indice della riga corrente presa in esame. Dato che lo stesso arco può essere presente in più celle, il suo ripetuto inserimento nella scena appesantirebbe il carico di lavoro dell'applicazione durante la gestione interattiva. Così determina una struttura dati per contenere i cursori degli archi attualmente inseriti e permettere di verificare quelli già presenti nella scena. Usa una tabella hash statica con liste di trabocco di dimensione cr, il cui costo computazionale per allocarla è uguale a quello di scandire le celle della griglia che cadono in V'. In tal modo le operazioni di inserimento nella tabella stessa e l'accertamento della presenza del cursore vengono espletate in tempo costante.

Analisi sulla complessità computazionale

Si testa l'algoritmo su circa un milione di archi, per verificarne la buona rispondenza. Si esamina la sua complessità computazionale. Siano n, il numero di archi della rete stradale, wh, la dimensione della griglia e $k = cr$, il numero di celle presenti nell'area visualizzata dalla viewport. L'algoritmo effettua in tempo $O(k)$ un'allocazione della tabella hash, con i cursori inseriti nella scena. Esegue in $O(k)$ la scansione delle celle visibili nella viewport e per ognuna di esse effettua un'ulteriore scansione del vettore v con i cursori presenti. Sia p la dimensione massima di v, allora il tempo per scandire v può essere maggiorato da p. Tenendo conto che le operazioni eseguite sulla tabella hash sono in $O(1)$, l'algoritmo esamina k celle ognuna contenente al massimo p cursori in $O(kp)$. Si osserva che k decresce all'aumentare di p, per cui se p tende a n, allora k tende a uno, in quanto al diminuire della risoluzione della griglia, le celle contengono un numero sempre maggiore di archi. Da ciò si evince che minore è la risoluzione della griglia e maggiore è il numero di archi contenuti nelle celle.

$$\lim_{p \to n} kp = n$$

Se la griglia è formata da un'unica cella, si ottiene un costo computazionale $O(n)$ scadente, come quello ottenuto dalla funzione disegnatrade, discussa nella sezione 2.4. Se la viewport contiene una piccola porzione di grafo e un numero esiguo di celle, il costo computazionale cresce come O(n), ma è talmente inferiore ad n che si può considerare una costante. Si inserisce la rete stradale in una griglia di risoluzione $r = 1$, i cui archi determinano un reticolo di celle di dimensione pari a $1018 \cdot 1038 = 1057702$. La cella con

il maggior numero di archi ne contiene $p = 140$. Aumentando lo zoom al massimo ingrandimento, si visualizza la sotto-griglia della viewport di dimensione $12 \cdot 12$ celle, che determina un numero di iterazioni maggiorato da $12 \cdot 12 \cdot 140 = 20160 \ll n$. Diminuendo lo zoom al minimo, le celle visualizzate nella viewport sono $81 \cdot 81$ e la complessità teorica calcolata si avvicina ad n. Ciò è dovuto alla maggiorazione introdotta. Da prove sperimentali risulta che il numero di archi presenti nella zona urbana più densa, in regime di grandi porzioni di grafo visualizzate, ha un ordine di grandezza inferiore a quanto stimato (circa 100000 archi). Ciò farebbe pensare che la complessità computazionale dipenda dalla distribuzione degli archi sulla griglia. Sia X una variabile aleatoria che conta il numero di archi presenti in una cella. La distribuzione empirica[3] di X si può ricavare raggruppando le celle con la stessa quantità di archi e contandone il numero.

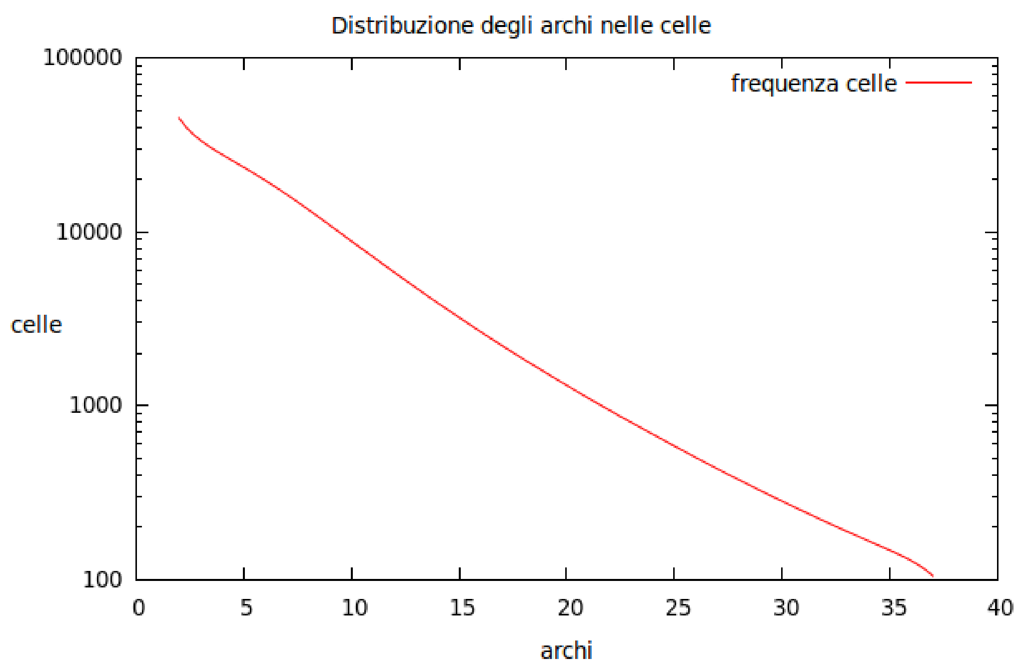

Figura 2.6: Distribuzione degli archi sulla griglia

Dal grafico ottenuto emerge che il 99.6% delle celle contiene un numero variabile di archi compreso tra 2 e 37. Tracciando una linea di tendenza su tale intervallo si determina l'andamento con la funzione esponenziale[19] $f(x) = Ae^{-0.18x}$. Si osserva che f non è una misura di probabilità, in quanto non è limitata in uno. Per renderla tale si pone $g(x)$ pari all'integrale di $f(x)$

definito sullo spazio dei valori $[0, \infty]$, che dà come risultato $\frac{k}{c}$. Eseguendo il rapporto tra $f(x)$ e $g(x)$ si ottiene la misura di probabilità μ, che ha legge esponenziale di parametro λ.

$$\mu(x) = \frac{Ae^{-\lambda x}}{\int_0^\infty Ae^{-\lambda x}\, dx} \Longrightarrow \int_0^\infty \mu(x)\, dx = 1$$

In questo specifico caso X ha legge empirica esponenziale di parametro $\lambda = 0,18$ sull'intervallo $[2; 40]$, la cui speranza matematica è data da $\nu = E[X] = \frac{1}{\lambda} \approx 6$. Ripetendo i test risulta che la complessità $O(k\nu)$ è una buona stima del reale numero di iterazioni effettuate dall'algoritmo. Pertanto la griglia della rete stradale si rivela una componente dell'editor indispensabile, per la realizzazione di algoritmi che devono ricavare velocemente tratti stradali vicini ad un determinato punto fornito in input.

2.6 Rilevamento delle anomalie

Si implementa un algoritmo che, associando le reti stradale-veicolare, assegni a ogni veicolo appartenente a V_c il cursore dell'arco di percorrenza più attinente o, diversamente, lo marchi come anomalo. Per ogni veicolo $v \in V_c$ determina l'indice i della cella, su cui cade v per la ricerca dell'arco. Considera anche gli indici delle otto celle adiacenti, solo nel caso in cui la loro distanza da v sia inferiore a $40m$. Siano w la larghezza della griglia, s_0, s_1, s_2, s_3 gli archi che delimitano rispettivamente i bordi ovest, nord, est, sud di i e $d(v, s)$ la funzione che restituisce la distanza euclidea tra v ed un arco. Allora l'insieme dei possibili archi attinenti $C = \{a_0, a_1, \ldots, a_{n-1}\}$ è dato dal contenuto delle celle:

- i in cui cade v;
- $i - 1$ se $d(v, s_0) \leq 40m$;
- $i + w$ se $d(v, s_1) \leq 40m$;
- $i + 1$ se $d(v, s_2) \leq 40m$;
- $i - w$ se $d(v, s_3) \leq 40m$;
- $i + w - 1$ se $d(v, s_0) \leq 40m \wedge d(v, s_1) \leq 40m$;
- $i + w + 1$ se $d(v, s_1) \leq 40m \wedge d(v, s_2) \leq 40m$;
- $i - w$–1 se $d(v, s_0) \leq 40m \wedge d(v, s_3) \leq 40m$;

- $i - w + 1$ se $d(v, s_2) \leq 40m \land d(v, s_3) \leq 40m$.

Assumendo per ipotesi che la risoluzione della griglia possa essere maggiore o uguale al raggio di errore posizionale GPS $r = 40m$, la ricerca dell'arco attinente si effettua sulle relative celle coperte da r centrate in v. La fig. 2.7 mostra un esempio di ricerca dell'arco di v nelle celle nord, nord-est ed est di i, in accordo alle proprietà discusse.

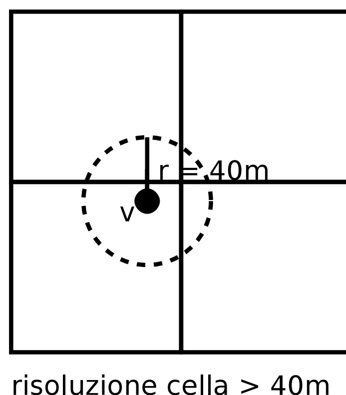

Figura 2.7: Ricerca dell'arco più attinente al veicolo v

L'algoritmo compara i parametri degli archi di C con quelli inerenti al veicolo (vedi cap 1.5). Mediante la funzione *affinità* restituisce, per ogni arco esaminato, un grado di attinenza il cui valore se è uguale a uno, indica la completa affinità dell'arco con il veicolo; se compreso tra zero e uno, è direttamente proporzionale al valore più vicino a uno; se minore o uguale a zero, non c'è alcun legame.

Dati n archi $a_1, \ldots, a_n$, un veicolo v e la funzione $affinita\ f(v, a)$.
Si applica $f_i(v, a_i) \quad \forall i = 1, \ldots, n$ e si considera $C = \{c_1, \ldots, c_n\}$ l'insieme dei valori restituiti da $f_1(v, a_1), \ldots, f_n(v, a_n)$. Sia $C' = \{c'_1, \ldots, c'_k\}$, $C' \subseteq C$, tale che $0 < c'_j \leq 1 \quad \forall j = 1, \ldots, k$ con $k \leq n$.
Se $C' = \varnothing$, allora il veicolo non ha nessun arco affine, altrimenti ($|C'| > 0$), l'arco più affine è dato dal $max\{c'_1, \ldots, c'_k\}$.

Inserisce nel campo *strada* di v il cursore dell'arco più affine, o il valore -1 nel caso di mancata associazione. Tale operazione, ripetuta su tutti i veicoli di V_c completa l'associazione tra le due reti prese in esame. Si apporta una modifica all'algoritmo *disegnaVeicoli*, in modo che colori in verde i veicoli associati correttamente ad un arco e in rosso quelli marcati come anomali.

L'immagine fornita dall'editor permette di comparare visivamente le due reti e di notare facilmente la loro congruenza, laddove la colorazione verde si sovrappone a quella bianca della strada. Mentre le discordanze, derivanti da situazioni di veicoli fermi in aree di sosta o in transito su una strada non presente nel grafo stradale, sono evidenziate in rosso. I veicoli parcheggiati certamente non forniscono informazioni utili ai fini della rilevazione di strade mancanti. Pertanto, dai dati di origine si filtrano solo quelli relativi ai veicoli in movimento con velocità superiore a $4\frac{km}{h}$.

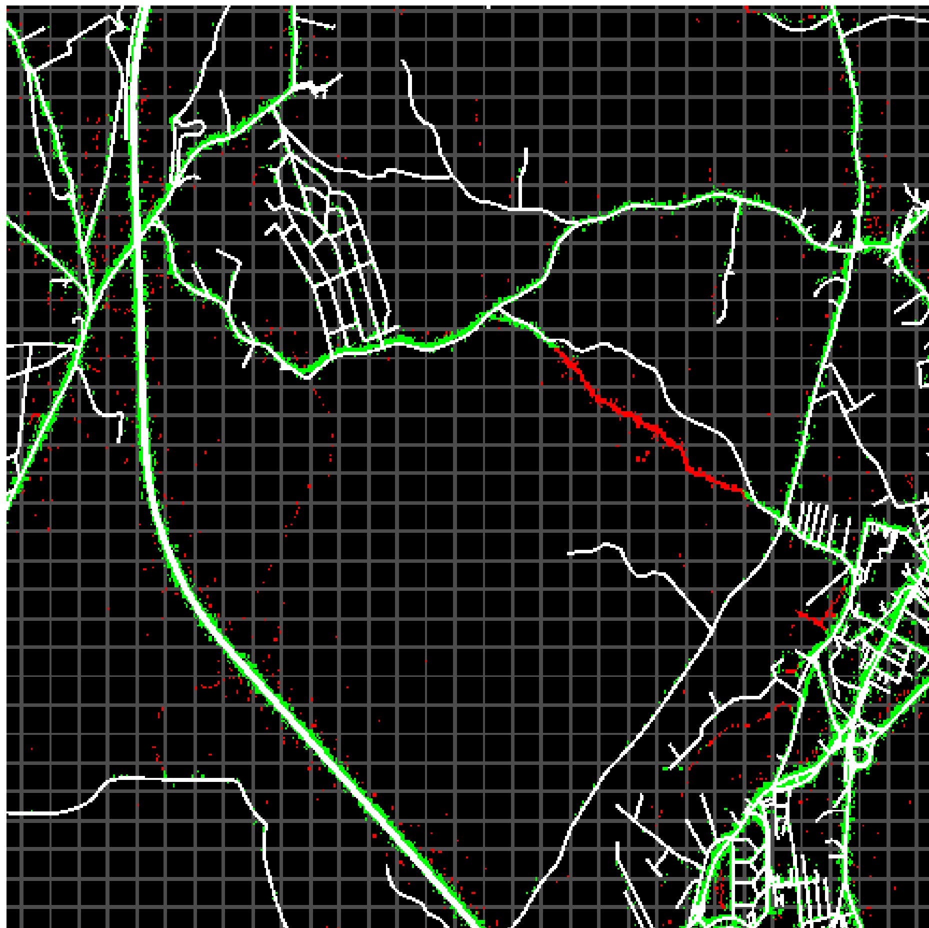

Figura 2.8: Associazione tre le reti stradale-veicolare

Osservando, qui sopra, l'immagine dell'associazione delle reti stradale-veicolare di Roma, si possono notare in rosso i flussi veicolari anomali inerenti a strade mancanti o posizionate male.

2.7 Manipolazione del grafo

Per correggere e trattare in modo interattivo le polilinee e gli archi del grafo, si implementano le componenti dell'editor che permettono di interagire graficamente sulla rete stradale. Questa, come detto, è composta dai vettori V_p, V_n e V_c, che contengono rispettivamente le informazioni sulle polilinee, i nodi estremi e gli archi del grafo (vedi cap. 1.2). V_p e V_c, pur strutturati diversamente, definiscono allo stesso modo l'informazione. V_p è organizzato su due livelli, dove il primo contiene le informazioni di ogni polilinea e il secondo la loro rappresentazione per punti. V_a, invece, ha una struttura piatta che descrive gli archi. Il collegamento tra V_a e V_p si realizza mediante il cursore *id_poly* presente in ogni arco. Ogni modifica effettuata sul grafo, deve mantenere sincronizzate le informazioni di V_p con quelle di V_a, in modo da descrivere sempre lo stesso grafo. Nel caso di inserimenti o rimozioni di tratti stradali è necessario aggiornare i nodi estremi di V_n, per tenerli sincronizzati con quelli di V_p e V_a. Inoltre si devono aggiornare le strutture *poly_archi_ref* e *griglia* in modo da consentire il corretto funzionamento dell'editor. Per questi motivi è impensabile che ogni componente interattiva possa modificare le strutture dati del grafo e provvedere ad aggiornare i moduli a basso livello dell'editor. A tale scopo si realizza il modulo *opgrafo*, che fornisce un'interfaccia per effettuare operazioni di rettifica al grafo, in modo trasparente e senza conoscere i dettagli implementativi delle strutture dati con le quali è realizzato. Le modifiche, oltre a mantenere la coerenza di V_p, V_a e V_n, vengono estese automaticamente anche a *griglia* ed a *poly_archi_ref*. Si descrivono di seguito le funzioni dell'interfaccia del framework di *opgrafo*.

void insPoly(Poly * nuova_poly, Vp, Va, Vn, str_griglia g)

Immette nel grafo una nuova polilinea. Come argomenti di input, oltre alle strutture dati V_p, V_a, V_n e *griglia*, si serve della variabile *nuova_poly*, che contiene le informazioni della nuova polilinea da immettere nel grafo. Attiva la funzione *polyToArchi(nuova_poly)*, che restituisce il vettore v di archi della rappresentazione per punti della nuova polilinea (vedi cap 1.2 per le relazioni di conversione da V_p a V_a). Il cursore dell'elemento i-esimo di v è dato da quello massimo presente in V_a incrementato di $i+1$ unità, in quanto $i = 0, \ldots, k-1$ con k pari al numero di archi di v. Esamina i cursori dei nodi estremi u_0 e u_1 di *nuova_poly*, per cui se:

- $u0 \in V_n$, inserisce il cursore di *nuova_poly* in $Vn[u_0]$ con il relativo nodo vicino u_1;

- $u1 \in Vn$, inserisce il cursore negato di *nuova_poly* in $Vn[u_1]$ con il relativo nodo vicino u_0;

- $u0 \notin Vn$, crea un nuovo nodo in posizione u_0 nel quale inserisce il cursore di *nuova_poly* con il relativo vicino u_1; assegna le coordinate di $Vn[u_0]$ uguali a quelle del nodo di testa di *nuova_poly*;

- $u1 \notin V_n$, crea un nuovo nodo in posizione u_1 nel quale inserisce il cursore negato di *nuova_poly* con il relativo vicino u_1; assegna le coordinate di $Vn[u_1]$ uguali a quelle del nodo di coda di *nuova_poly*.

Aggiunge una cella nella struttura dati *poly_archi_ref*, dove immette il cursore di *nuova_poly* associandolo a quello del primo arco di v. Aggiunge *nuova_poly* in V_p, v in V_a e attiva la funzione *inslinea* su ogni elemento di v, per inserire i nuovi archi nella griglia.

void rimuoviPoly(int id_poly, Vp, Va, Vn, str_griglia * g)

Rimuove dal grafo la polilinea *id_poly* di V_p, con i relativi archi di V_a, marcandoli come rimossi (*rimosso = true*, *tipo = -1*), per mantenere la coerenza tra i cursori delle strutture dati dell'editor. Ricava i nodi di testa u e di coda v della polilinea e rimuove da $V_n[u]$ e $V_n[v]$ il cursore *id_poly* e quello relativo al nodo vicino associato. Se poi il nodo resta senza cursori viene anch'esso rimosso. Elimina dalla griglia, con la funzione *rimuovilinea*, i cursori degli archi appena rimossi.

nascondiPoly e scopriPoly

nascondiPoly occulta dal grafo la polilinea *id_poly* di V_p con i relativi archi di V_a, marcandoli come nascosti (*rimosso = true*) e lasciando in V_n e nella griglia i relativi cursori. In tal modo la polilinea, anche se presente nel grafo, resta temporaneamente invisibile nella scena. *scopriPoly*, viceversa, reimmette in G la polilinea precedentemente nascosta, settando l'attributo rimosso a *true* di V_p e V_a. La due funzione vengono utilizzate per distinguere i grafi G e G', dove G', sottoinsieme di G, contiene solo gli archi con l'attributo rimosso pari a *false*, mentre G li contiene tutti inclusi quelli nascosti. Entrambe, si vedrà più avanti, sono particolarmente utili nella semplificazione a posteriori del grafo stradale (vedi cap 3).

void rimuoviNodo(int id_nodo, Vp, Va, Vn, str_griglia * g)

Rimuove il nodo *id_nodo* dal grafo attivando la funzione *rimuoviPoly* su tutti i cursori che connettono le polilinee al nodo stesso.

void nascondiNodo(int id_nodo, Vp, Va, Vn, str_griglia * g)

Nasconde il nodo *id_nodo* dal grafo attivando la funzione *nascondiPoly* su tutti i cursori che connettono le polilinee al nodo stesso.

void scopriNodo(int id_nodo, Vp, Va, Vn, str_griglia * g)

Scopre il nodo *id_nodo* attivando la funzione *scopriPoly* su tutti i cursori che connettono le polilinee al nodo stesso.

void dividiPoly(int id_poly, int h, point_2d p, Vp, Va, Vn, str_griglia * g)

Divide in due la polilinea *id_poly* in corrispondenza del nodo interno h. Data la polilinea $P = \{p_0, p_1, \ldots, p_h, \ldots, p_{n-1}\}$, crea le polilinee $P_0 = \{p_0, \ldots, p_h\}$ e $P1 = \{p_h, \ldots, p_{n-1}\}$. I nodi estremi di testa di P_0 e P_1 sono rispettivamente p_0 e p_h, mentre quelli di coda sono p_h e p_{n-1}. Il processo di divisione genera un nuovo nodo estremo nel punto p_h che connette P_0 a P_1 e compare in V_n.

2.8 Gestione delle polilinee

Si implementa la classe *HwndPoly* che permette di compiere operazioni di inserimento, divisione ed eliminazione delle polilinee della rete stradale. Nella fase iniziale di avvio, la componente *Display* dell'editor crea un'istanza della classe che si pone in uno stato di attesa *HPWAIT*. La chiamata di callback *setAzione* avvia i servizi interattivi. Questi, modificando lo stato della classe, abilitano i metodi *disegnaStato*, *azionaEvento* e *testNodo* che si innescano rispettivamente con il click del mouse, lo spostamento del puntatore e il ridisegno della scena grafica. Tali metodi eseguono operazioni diverse in base allo stato corrente del servizio e utilizzano le funzioni *clickonnodopoly* e *clickonnodoarco* del framework *Disegno*, per ricavare le informazioni inerenti alle polilinee e gli archi di un nodo u.

La funzione *clickonnodopoly* ha come argomenti di input le coordinate di tipo finestra $P_f = (x_f, y_f)$ puntate dal mouse e restituisce il cursore u

del nodo estremo, che cade nell'area quadrata descritta da $P_f + \epsilon$. Attiva la funzione *getcoordinatebymouse(P_f)* e ricava le coordinate di tipo mondo $P_m = (x, y)$ corrispondenti a quelle P_f puntate dal mouse. Con la funzione *getptrcellabyxy(P_m)* ottiene la lista dei cursori $C = \{a_0, a_1, \ldots, a_{h-1}\}$ degli archi contenuti nella cella su cui cade P_m. Per ogni $a \in C$ estrae le informazioni dei nodi estremi (u, v) della polilinea corrispondente e controlla la coincidenza delle coordinate dei nodi con P_m. Dato u_* un generico nodo di un arco (u o v), calcola la distanza euclidea d, tra P_m e u_*, e confronta d con un certo δ piccolo fissato per stabilire la coincidenza del mouse con il nodo. Se $d \leq \delta$, allora l'algoritmo termina restituendo il cursore di u_*, oppure continua la ricerca sino all'individuazione di un nodo coincidente con P_m; in caso contrario la ricerca termina con esito negativo. La funzione *clickonnodoarco* ha gli stessi argomenti di input di *clickonnodopoly*, ma restituisce il cursore della polilinea p e l'*id_punto* del nodo interno di p. Esegue una scansione analoga degli archi contenuti nella cella della griglia puntata dal mouse.

Si descrivono nelle sezioni successive i servizi gestiti da *HwndPoly*.

Inserimento di nuove polilinee

Il metodo *nuovaPoly* pone lo stato della classe su *HPNEWINS*, che attiva il servizio per permettere l'inserimento di una nuova polilinea $P = \{p_0, p_1, \ldots, p_{h-1}\}$ nel grafo di G. Il metodo *testNodo* attiva la funzione *clickonnodopoly* del framework *Disegno*, che restituisce un nodo estremo u vicino al punto selezionato dal mouse. Eseguendo un click del mouse in corrispondenza di u, si attiva la funzione *azionaEvento*, che trasforma le coordinate del mouse di tipo finestra in quelle di tipo mondo e pone $p_0 = u$ come nodo estremo iniziale di una ipotetica strada da definire. Stesso procedimento di trasformazione delle coordinate avviene a ogni click operato su qualsiasi punto della scena che interessa il tratto stradale, le cui coordinate vengono memorizzate nel vettore di appoggio v. Un ulteriore click operato sul nodo q pone $p_{h-1} = q$ come nodo estremo finale della strada e setta lo stato della classe pari ad *HPNEWBLOCK*, in modo da bloccare la ricezione di nuovi input esterni e permettere la sola visualizzazione della polilinea generata da v.

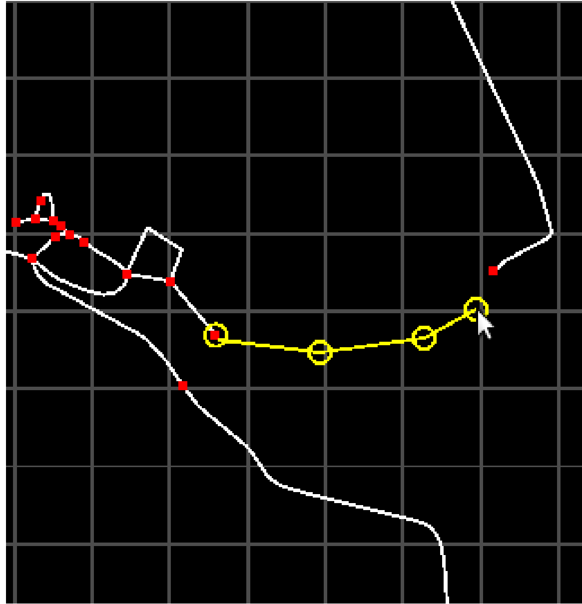

Figura 2.9: Inserimento di una polilinea

Con la chiamata di callback *setAzione* si attiva il metodo *salvaNuovaPoly* per inserire P in G. Questo inizializza una nuova variabile *new_poly* che contiene le informazioni di P. Inserisce le coordinate di v in *new_poly*, convertendole in quelle del sistema di riferimento geografico utilizzato. Aggiorna, poi, gli attributi che definiscono i nodi estremi di P. Se $p_0 = u$, il nodo estremo iniziale di P è pari ad u, altrimenti definisce un cursore $u' \notin V_n$ per creare un nuovo nodo estremo. $Sep_{h-1} = q$, il nodo estremo finale di P è pari a q, altrimenti definisce un cursore $q' \notin V_n$, per creare un nuovo nodo estremo. Attiva la funzione *insPoly* del framework *OpGrafo*, che inserisce la polilinea definita da *new_poly* in G. L'inserimento di P o l'annullamento della procedura sopradescritta avviano la funzione *fineNuovaPoly*, che resetta lo stato della classe ad *HPWAIT*.

Divisione e cancellazione delle polilinee

Il metodo *splitPoly* pone lo stato della classe su *HPSPLITSTART*, che attiva il servizio per permettere la divisione di una polilinea $P = \{p_0, p_1, \ldots, p_{h-1}\}$ in corrispondenza di un nodo interno $u \in \{p_1, \ldots, p_{h-2}\}$. Il metodo *testNodo* attiva la funzione *clickonnodoarco*, che restituisce il nodo u vicino al punto selezionato dal mouse. Con un click del mouse in corrispondenza di u si attiva la funzione *azionaEvento*, che memorizza il cursore *id_punto* di u. Per com-

pletare la procedura di divisione la chiamata di callback *setAzione* attiva il metodo *salvaSplitPoly*. Questo utilizza la funzione *dividipoly* del framework *OpGrafo* che, passandole come parametri il cursore di P e u, concretizza la divisione di P in due sottopolilinee P' e P''. La divisione di P o l'annullamento della procedura determina l'avvio della funzione *fineSplitPoly*, che resetta lo stato della classe in *HPWAIT*.

Allo stesso modo il metodo *eliminaPoly* pone lo stato della classe su *HPELIMINASTART*, che attiva il servizio per permettere la cancellazione di una polilinea $P = \{p_0, p1, \ldots, p_{h-1}\}$ in corrispondenza di un nodo interno $u \in \{p_1, \ldots, p_{h-2}\}$. Il metodo *testNodo* attiva la funzione *clickonnodoarco*, che restituisce il nodo u vicino al punto selezionato dal mouse. Eseguendo un click in corrispondenza di u si attiva la funzione *azionaEvento* che memorizza il cursore *id_punto* di u. Per completare la procedura di cancellazione la chiamata di callback *setAzione* permette di attivare il metodo *salvaEliminaPoly*. Questo utilizza la funzione *eliminapoly* del framework *OpGrafo* che, passandole come parametri il cursore di P, concretizza la cancellazione da G. La cancellazione di P o l'annullamento della procedura determina l'attivazione della funzione *fineEliminaPoly*, che resetta lo stato della classe in *HPWAIT*.

2.9 Gestione degli archi

Si implementa la classe *HwndArco* che permette di compiere operazioni di spostamento, divisione e rimozione degli archi della rete stradale. Nella fase iniziale di avvio, la componente *Display* dell'editor crea un'istanza della classe, che si pone in uno stato di attesa *HAWAIT*. La chiamata di callback *setAzione* avvia i servizi interattivi, che abilitano i seguenti metodi:

- *azionaEvento*, si attiva con un click del mouse;
- *trascinaEvento*, si attiva con il tasto sinistro del mouse premuto, per spostare un elemento nella scena grafica;
- *fineEvento*, si attiva con il rilascio del tasto del mouse, dopo la fase di trascinamento;
- *testSelezione*, si attiva con il semplice movimento del mouse;
- *disegnaStato*, si attiva quando si ridisegna la scena.

In base allo stato corrente della classe detti metodi possono eseguire diverse operazioni.

Spostamento degli archi

Il metodo *initSpostamento* pone lo stato della classe su *HASPOSTAINIT*, che attiva il servizio per permettere lo spostamento interattivo di un nodo u. Il metodo *testSelezione* determina il nodo u_* (interno u_i o estremo u_e) più vicino al punto di selezione del mouse. Con un clik del tasto sinistro del mouse su u_* si attiva la funzione *azionaEvento*, che se:

- $u_* = u_e$, ricava le coordinate di u_e;
- $u_* = u_i$, determina il cursore p della polilinea che contiene u_i e l'*id_punto* di u_i.

Gli archi così individuati vengono memorizzati in un vettore di appoggio v. Con il tasto sinistro premuto e trascinando il puntatore, si attiva il metodo *trascinaEvento*, che consente di spostare u_* e gli archi ad esso connessi nella scena grafica. Questo utilizza la funzione *getcoordinatebymouse* per convertire le coordinate di tipo finestra del mouse in quelle P_m di tipo mondo. Se $u_* = u_i$, allora per definizione gli archi da spostare sono due (cap. 1.2). Il metodo, poi, attiva la funzione *getpolyref(p)*, che restituisce il cursore a_0 del primo arco della polilinea p. Essendo la struttura dei cursori un'enumerazione crescente, allora u_i connette gli archi $a_x = a_0 + id_punto - 1$ e $a_{x+1} = a_0 + id_punto$. Lo spostamento diu_i si concretizza assegnando la coordinata P_f al nodo di coda di a_x e a quello di testa di a_{x+1}. Se invece $u_* = u_e$, allora gli archi connessi a u_e possono essere un numero variabile maggiore o uguale di uno. In questo caso *trascinaEvento* ricava la lista dei cursori $\{p_0, p_1, \ldots, p_{h-1}\}$ delle polilinee connesse a u_e. Per ogni p_i determina il cursore a_0 del primo arco di p_i e il numero k di archi che compongono p_i. Ricava, anche, l'arco a_i collegato a u_e e modifica le sue coordinate sfruttando le proprietà dei cursori:

- se $p_i < 0$, allora pone $a = a_0$ e assegna al nodo di testa di a le coordinate P_m;
- se $p_i > 0$, allora pone $a = a_0 + k - 1$ e assegna al nodo di coda di a le coordinate P_m.

Essendo u_e un nodo estremo dotato di proprie coordinate, *trascinaEvento* pone $u_e = P_f$. Una volta rilasciato il tasto del mouse si attiva il metodo *fineEvento*, che ratifica l'azione di spostamento di u_* con il conseguente posizionamento degli archi interessati. Questo avvia la funzione *rimuovilinea* sugli archi, che cancella gli archi da modificare per, poi, reinserli con le nuove

modifiche apportate. Ciò consente al sistema di mantenere sempre sincronizzata la griglia al grafo. Stessa opera di aggiornamento viene effettuata sulle coordinate geografiche degli elementi di V_p e V_n, per renderle coerenti con quelle degli archi modificati. Con la chiamata di callback *setAzione* infine si attiva il metodo *fineSpostamento*, che resetta lo stato della classe su *HAWAIT*.

Divisione degli archi

Il metodo *initSplit* pone lo stato della classe su *HASPLITINIT*, che attiva il servizio per permettere la divisione di un arco $a = (u, v)$ in corrispondenza di un punto w interno ad a. Il metodo *testSelezione* attiva la funzione *distanzaarcoclick*, che restituisce il cursore dell'arco a vicino al punto selezionato dal mouse. Con un click del mouse, in corrispondenza di a, si avvia la funzione *azionaEvento*, che memorizza a e setta lo stato della classe su *HASPLITSEPARA*. Alla pressione del tasto sinistro del mouse si riattiva nuovamente il metodo *azionaEvento*, che determina le coordinate del punto p selezionato dal mouse. Con il tasto sinistro premuto e trascinando il puntatore si avvia il metodo trascinaEvento, che calcola le coordinate del punto q selezionato dal mouse. La chiamata di callback disegna la scena grafica mediante la funzione *disegnaStato*, che inserisce il segmento formato dai nodi (p, q). La procedura si ripete fino al rilascio dell tasto del mouse, che richiama il metodo *fineEvento*. Questo calcola il punto di intersezione tra a e (p, q), che determina il punto w interno a a, per dividere l'arco in due sottoarchi (u, w) e (w, v) (figura 2.10).

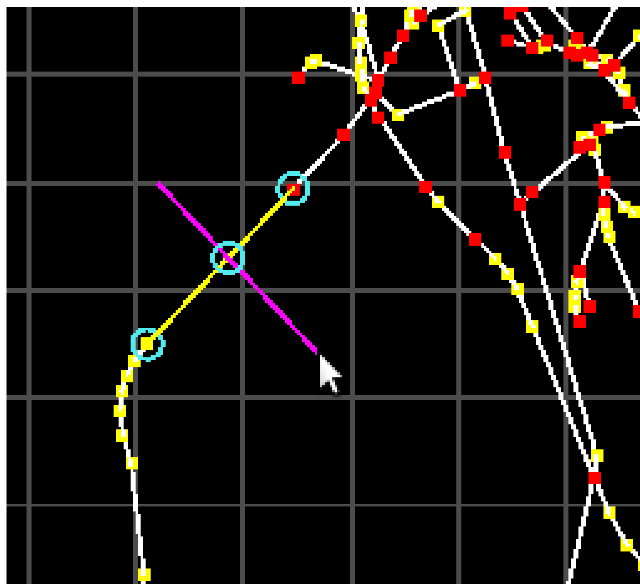

Figura 2.10: Divisione di un arco

La chiamata di callback *setAzione* attiva il metodo *salvaSplit*, che sostituisce la polilinea $P = \{p_0, p_1, \ldots, u, v, \ldots, p_{h-1}\}$ di a con una nuova $P' = \{p_0, p_1, \ldots, u, w, v, \ldots, p_h\}$. Aggiorna le strutture dati griglia V_p e V_n con i metodi *rimuovipoli(P)* e *inspoly(P')*. La divisione di a o l'annullamento della procedura determinano l'attivazione della funzione *fineSplit*, che resetta lo stato della classe su *HAWAIT*.

Rimozione degli archi

Il metodo *initRimuovi* pone lo stato della classe su *HADELINIT*, che attiva il servizio per permettere la cancellazione di un arco $a = (u, v)$. Il metodo *testSelezione* avvia la funzione *distanzaarcoclick*, che restituisce il cursore dell'arco a vicino al punto selezionato dal mouse. Con un click in corrispondenza di a si attiva la funzione *azionaEvento* che memorizza a. La chiamata di callback *setAzione* attiva il metodo *salvaRimuovi*, che sostituisce la polilinea $P = \{p_0, p_1, \ldots, p_i, u, v, p_j, \ldots, p_{h-1}\}$ di a con una nuova $P' = \{p_0, p_1, \ldots, p_i, p_j, \ldots, p_{h-1}\}$ e aggiorna le strutture dati *Griglia*, V_p e V_n con i metodi *rimuovipoli(P)* e *inspoly(P')*. La cancellazione di a o l'annullamento della procedura determinano l'attivazione della funzione *fineRimuovi*, che resetta lo stato della classe su *HAWAIT*.

2.10 Esportazione di un sottografo

Si implementa la classe *Esporta*, che permette di compiere operazioni di selezione ed esportazione di un sottografo della rete stradale. Nella fase iniziale di avvio, la componente *Display* dell'editor crea un'istanza della classe che si pone in uno stato di attesa *EXPWAIT*. La chiamata di callback *setAzione* attiva i servizi interattivi. Questi abilitano i metodi *disegnaStato*, *trascina* e *azionaEvento* che, si avviano, come detto, con un click del mouse o il tasto premuto e il trascinamento del puntatore. I servizi della classe permettono di esportare un sottografo a forma circolare o poligonale.

2.10.1 Esportazione circolare

Il metodo *initCerchio* pone lo stato della classe su *EXPINITCERCHIO*, che attiva il servizio per permettere la selezione circolare di un sottografo G'. Inizializza il vettore booleano *da_esportare* di dimensione n, pari al numero di polilinee della rete stradale e lo setta a *false*. Con un click del mouse, eseguito in un punto qualsiasi della mappa, si avvia la funzione *azionaEvento*. Questa, a sua volta, si serve della funzione *archiinraggio* del framework disegno, per ottenere la lista degli archi che cadono all'interno dell'area descritta da una circonferenza di raggio r.

void archiinraggio(double r, point_2d Pf, bool * da_esportare)

La funzione prende come argomenti di input la dimensione del raggio centrato nelle coordinate di tipo finestra P_f e il vettore booleano *da_esportare*. Al termine dell'algoritmo gli indici delle celle del vettore, corrispondenti alle polilinee che cadono nella circonferenza, sono settati a *true*. *archiinraggio*, così, converte le coordinate di tipo finestra in quelle cartesiane $C = (x, y)$. Ricava il numero k di celle della griglia coperte da r

$$k = \left\lceil \frac{rh}{s} \right\rceil$$

dove h è l'altezza della visuale e s è la risoluzione della cella della griglia. A tal proposito si osserva, che il prodotto rh permette di mantenere costante la grandezza della circonferenza visualizzata al variare dello zoom. La funzione esegue poi la scansione delle celle, partendo da quella su cui cade C e visitando quelle adiacenti con una traiettoria a spirale. Il numero di spire da compiere per maggiorare l'area coperta dalla circonferenza è pari a k (figura 2.11).

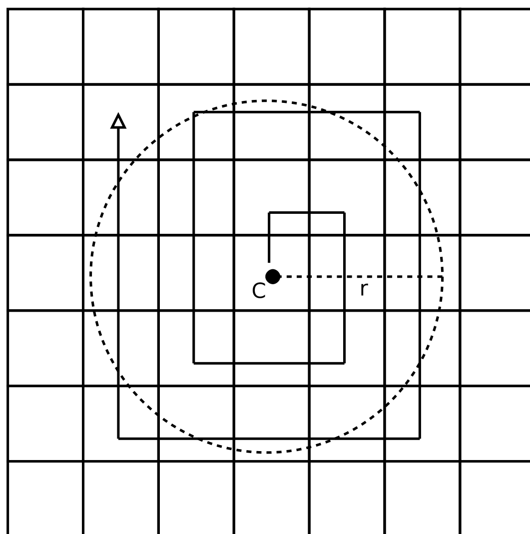

Figura 2.11: Scansione delle celle a spirale

L'algoritmo attiva, su ogni cella visitata, la funzione *archicellainraggio*, che ricava la lista degli archi $L = \{a_0, a_1, \ldots, a_{h-1}\}$ contenuti da z e determina la loro posizione rispetto alla circonferenza mediate il seguente teorema.

Teorema 2. *Sia s un segmento di nodi (u, v) tali che u ha coordinate cartesiane (x_1, y_1) e $v = (x_2, y_2)$. Sia C una circonferenza centrata nel punto $c = (x_c, y_c)$ di raggio r. Si pongono i valori a, b, c e δ tali che*

$$\begin{cases} a = (x_2 - x_1)^2 + (y_2 - y_1)^2 \\ b = 2(((x_2 - x_1)(x_1 - x_c)) + ((y_2 - y_1)(y_1 - y_c))) \\ c = {x_c}^2 + {y_c}^2 + {x_1}^2 + {y_1}^2 - (2(x_c x_1 + y_c y_1)) - r^2 \\ \delta = b^2 - 4ac \end{cases}$$

Allora vale che se $\delta < 0$ non vi è intersezione tra s e C, mentre se $\delta \geq 0$ si calcolano i valori

$$\begin{cases} u_1 = \dfrac{-b + \sqrt{\delta}}{2a} \\ u_2 = \dfrac{-b - \sqrt{\delta}}{2a} \end{cases}$$

che permettono di stabilire l'intersezione tra s e C come segue:

- *se* $\delta = 0 \wedge 0 \leq u1 \leq 1 \wedge 0 \leq u2 \leq 1$, *allora s è tangente a C, altrimenti interseca un punto di C;*

- *se* $\delta > 0 \wedge ((u_1 > 1 e u_2 > 1) \vee (u_1 < 0 e u_2 < 0))$, *allora s non interseca C;*

- *se* $\delta > 0 \wedge ((0 \leq u_1 \leq 1 \wedge (u_2 > 1 \vee u_2 < 0)) \vee (0 \leq u_2 \leq u_1 \wedge (u_1 > 1 \vee u_1 < 0)))$, *allora s interseca C;*

- *se* $\delta > 0 \wedge ((0 \leq u_1 \leq 1) \wedge (0 \leq u_2 \leq 1)) \wedge u_1 = u_2$, *allora s è tangente a C;*

- *se* $\delta > 0 \wedge ((0 \leq u_1 \leq 1) \wedge (0 \leq u_2 \leq 1)) \wedge u_1 \neq u_2$, *allora s interseca C in due punti;*

- *per tutti i delta* ≥ 0 *che non soddisfano i punti sopraelencati si determina che s è interno a C.*

Dato un arco $a \in L$ e il relativo cursore p della polilinea di appartenenza, si considera p appartenente al sottografo G' generato da C se e solo se a è interno, tangente o intersecante a C. L'appartenenza di p a G' è ratificata dalla funzione *archicellainraggio*, che pone *da_esportare[p] = true*. Pertanto, la funzione *archiinraggio* genera il sottografo G', che contiene le polilinee nelle quali almeno un arco è incluso nella circonferenza definita graficamente dall'utente. La funzione *disegnaStato* infine mette in evidenza G', colorando di giallo le polilinee appartenenti (fig. 2.12)

Figura 2.12: Selezione circolare di un sottografo

2.10.2 Esportazione poligonale

Il metodo *initPoligono* pone lo stato della classe su *EXPINITPOLIGONO*, che attiva il servizio per permettere la selezione poligonale di un sottografo G'. Inizializza il vettore booleano *da_esportare* di dimensione n, pari al numero di polilinee della rete stradale e lo setta a *false*. Con un primo click del mouse sulla mappa si attiva la funzione *azionaEvento*, che inizializza il primo vertice v_0 di un poligono P in corrispondenza delle coordinate cartesiane puntate dal mouse. I click successivi permettono di aggiungere a P gli altri vertici che determinano la forma. Tenendo premuto il tasto sinistro del mouse si attiva il metodo *trascinaEvento*, che aggiorna le coordinate cartesiane dell'ultimo vertice con quelle già puntate. In questo modo si può correggere in tempo reale il segmento che unisce gli ultimi due vertici di P. Ultimato il disegno del poligono nella scena, si procede alla selezione del sottografo G' che cade nell'area definita da P. Selezionando dal menù l'apposito comando di chiusura, si avvia con la chiamata *setAzione* il metodo *chiudiPoligono*, che chiude il perimetro di P per focalizzare le polilinee interne ad esso. Questo setta gli elementi del vettore *da_esportare* pari a *false* e attiva la procedura del framework *Disegno archiinpoligono*.

void archiinpoligono(point_2d * P, bool * da_esportare)

La funzione prende come argomenti di input il poligono P e il vettore *da_esportare*. Esegue una prima scansione dei vertici di P, per definire le coordinate minime (x_{min}, y_{min}) e massime (x_{max}, y_{max}). Determina i valori s, r e c e identifica l'area descritta da P (figura 2.13), dove:

- $s = getidcella(x_{min}, y_{min})$, corrisponde alla cella della griglia rettangolare situata in basso a sinistra;
- $c = getidcella(x_{max}, y_{min}) - s$, è il numero di colonne necessarie a contenere P sulle ascisse;
- $r = \frac{getidcella(x_{min}, y_{max}) - s}{w}$ (con w pari alla risoluzione delle celle), è il numero di righe necessarie a contenere P sulle ordinate.

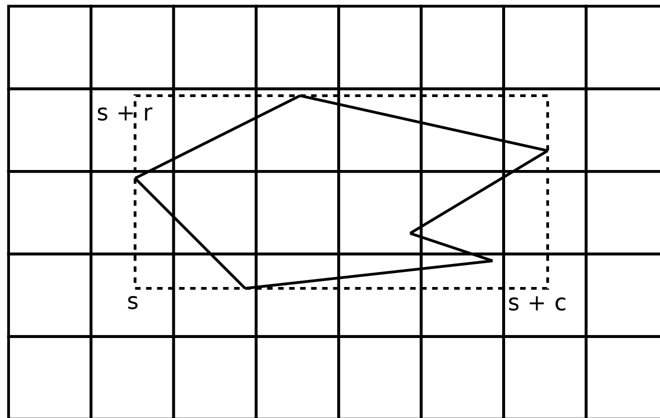

Figura 2.13: Sottogriglia che contiene il poligono

Esegue una scansione delle celle S per ricercare gli archi interni a P e inserirli nel sottografo G'. Un arco $a = (u, v)$ è interno a P se rispetta almeno una delle seguenti proprietà:

(1) il nodo di testa u è interno a P;

(2) il nodo di coda v è interno a P;

(3) i nodi u e v sono esterni a P e a interseca uno degli spigoli di P.

Per determinare l'appartenenza del nodo al poligono la funzione sfrutta il seguente teorema.

Teorema 3. *Sia u un punto di coordinate (x, y) e $P = \{p_0, p_1, \ldots, p_{h-1}\}$ un poligono concavo o convesso tale che per ogni $i = 0, \ldots, h-1$, p_i ha coordinate (x_i, y_i). Si assume per ipotesi che le coordinate di p_0 siano uguali a quelle di p_{h-1}, in modo da garantire la chiusura di P. Si considera r la semiretta parallela all'asse delle ordinate il cui punto iniziale è pari a u, mentre quello finale è (x, ∞). Allora u è interno a P se il numero di intersezioni tra r e P è dispari, altrimenti u è esterno a P.*

Nel caso in cui i nodi u e v sono esterni a P, la funzione ricerca il punto di intersezione con P stesso (figura 2.14).

Figura 2.14: Selezione poligonale di un sottografo

2.11 Inserimento automatico delle polilinee

La procedura di individuazione e correzione delle anomalie del grafo, fin qui trattata, impone un momento di riflessione. La realizzazione dell'editor che, fornisce a video l'immagine della rete stradale con la sovrapposizione dell'informazione veicolare annessa, permette di comparare visivamente le due reti e di vedere immediatamente una loro eventuale congruenza, con la colorazione verde e discordanza con quella rossa. Le componenti *HwndPoly* e *HwndArco* dotano l'editor di una serie di azioni interattive guidate, per apportare possibili modifiche al grafo mediante l'inserimento, lo spostamento,

la divisione e l'eliminazione degli elementi anomali, con la possibilità opzionale di poter salvare il lavoro di correzione. Ciò dà risultati soddisfacenti, ma non ottimali. Tale procedura, infatti, prevede un intervento manuale di ridisegno delle polilinee anomale, che dipende dalla precisione di chi opera nel sistema e quindi potrebbe determinare un'erronea correzione.

Per questo motivo si ricerca una nuova soluzione che permetta di inserire le strade mancanti nel grafo in modo automatico, sempre sfruttando le informazioni fornite dall'associazione delle reti stradale-veicolare (vedi cap. 2.8). L'associazione si realizza con l'inserimento del cursore dell'arco più affine nel campo *strada* dei veicoli di V_c; diversamente nel caso di mancata associazione il valore -1. Considerando che sarebbe impredicibile sapere a priori il flusso veicolare da correggere, si realizza uno strumento che ne permetta la selezione automatica. Si implementa la classe *AutoCorrect* che offre i servizi interattivi per selezionare i nodi estremi della polilinea mancante e il flusso veicolare anomalo da correggere. I servizi, allo stesso modo di quanto descritto per la gestione degli archi, in base allo stato corrente della classe abilitano le funzioni:

- *disegnaStato*, si attiva quando si ridisegna la scena grafica;
- *azionaEvento*, si attiva alla pressione del tasto del mouse;
- *trascinaEvento*, si attiva al trascinamento del mouse con il tasto premuto;
- *fineEvento*, si attiva al rilascio del tasto del mouse;
- *testSelezione*, si attiva ad ogni spostamento passivo del mouse.

Quando si seleziona dal menù il comando per inserire una nuova polilinea, si attiva la funzione *setAzione* che richiama il metodo *initNuovaPoly*. Questo pone lo stato del servizio su *ACINITNUOVAPOLY*, che permette di selezionare i nodi estremi della polilinea mancante.

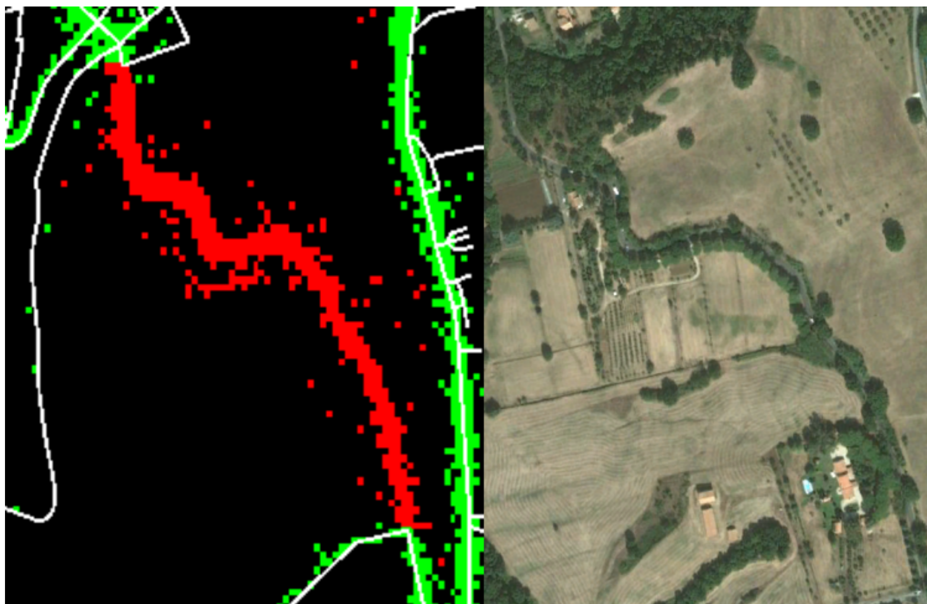

Figura 2.15: Flusso veicolare anomalo

La funzione *testSelezione* evidenzia il nodo estremo del grafo nelle vicinanze del cursore del mouse, utilizzando la stessa procedura descritta nella sezione 2.8 di questo capitolo. Con un click sul nodo desiderato si attiva la funzione *azionaEvento*, che memorizza il suo cursore. Al termine della selezione dei due nodi estremi, lo stato del servizio cambia automaticamente e si setta su *ACNUOVAPOLYRANGE*. Ciò consente di tracciare nella scena grafica il rettangolo che identifica la sottorete veicolare inerente alla zona anomala da correggere. Alla pressione del tasto del mouse si attiva *azionaEvento*, che registra la coordinata del primo vertice del rettangolo. Lo spostamento del puntatore avvia la funzione *trascinaEvento*, che aggiorna le coordinate degli altri tre vertici del rettangolo in base alle nuove coordinate del mouse. Al rilascio del tasto del mouse si attiva la funzione *fineEvento*, che scansiona la rete veicolare V_c e inserisce nella sottorete V_c' i veicoli che cadono nell'area descritta dal rettangolo (vedi figura 2.16).

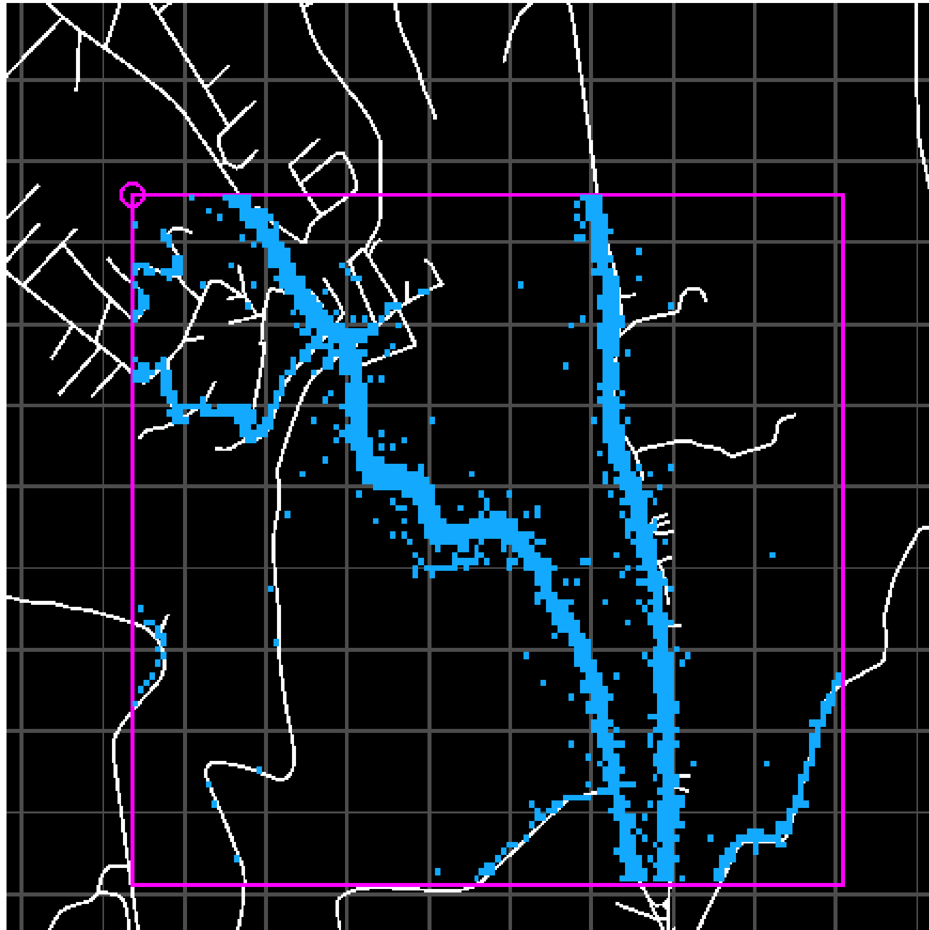

Figura 2.16: Selezione sottoflusso veicolare

Alla conferma dal menù della suddetta operazione, la chiamata *setAzione* attiva il metodo *confermaRange*, che pone lo stato del servizio su *ACNUOVAPOLYSEL*. La funzione *testSelezione* registra le coordinate correnti del mouse e le passa a *disegnaStato*, che traccia nella scena grafica una piccola circonferenza C di raggio r. Al click del mouse *azionaEvento* marca come selezionati i veicoli di V_c', le cui coordinate cadono all'interno dell'area descritta da C. Tale operazione deve essere eseguita tre volte sul flusso anomalo selezionato. Il primo click si deve effettuare in corrispondenza dell'inizio del flusso veicolare, il secondo in una zona a scelta, mentre il terzo alla fine del flusso stesso. Quindi i veicoli marcati vengono evidenziati in verde da *disegnaStato* (vedi figura 2.17).

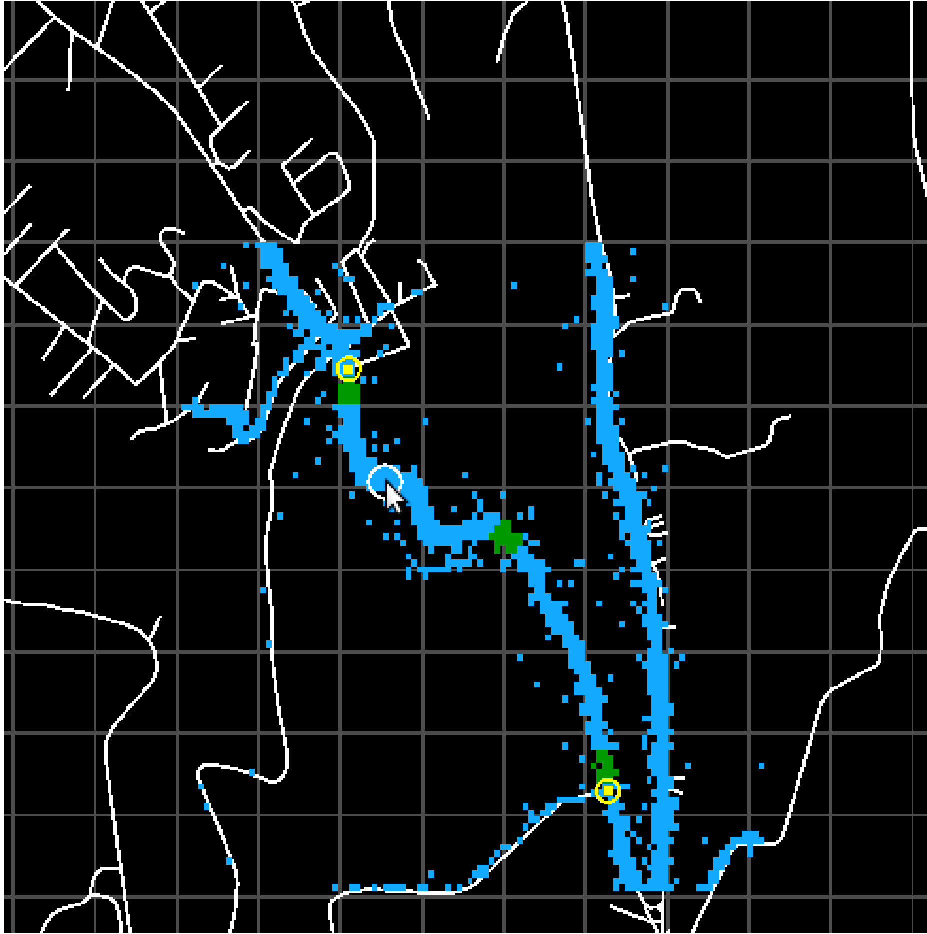

Figura 2.17: Selezione dei veicoli appartenenti al flusso veicolare anomalo

Il flusso veicolare anomalo è racchiuso tra le barriere formate dai veicoli marcati in corrispondenza dei nodi estremi della polilinea mancante. Pertanto il restante flusso anomalo si può evidenziare automaticamente a partire dai veicoli marcati al centro. Il terzo (ed ultimo click) attiva la procedura *selezionaAuto*, che seleziona l'intero flusso veicolare. A tal proposito l'algoritmo esegue una ricerca a largo spettro per individuare in $V_c{}'$ i veicoli vicini a quelli marcati. Siano V_i, V_f e V_s l'insieme dei veicoli marcati rispettivamente nella zona iniziale (1), finale (2) e centrale (3). Un possibile algoritmo di ricerca può essere realizzato sfruttando la ricorsione. Si prende un veicolo $v \in V_s$ e si ricercano tutti i veicoli $q \in V_c{}'$, tali che la loro distanza da v sia inferiore ad un certo ϵ fissato. Per ognuno di essi si attiva ricorsivamente la stessa procedura, fin quando non si toccano le barriere V_i e V_f che interrompono la ricorsione. Considerando che mediamente un flusso veicolare anomalo è

composto da molte centinaia di veicoli e la ricorsione è molto più lenta di una procedura iterativa, si decide di implementare l'algoritmo di ricerca simulando gli stack ricorsivi con l'utilizzo di una pila H. Nella fase iniziale la funzione *selezionaAuto* inserisce in H i veicoli di V_s. Estrae il primo $v \in H$, contrassegna tutti i veicoli di V_c' non marcati, la cui distanza è inferiore di ϵ e li inserisce in H. Dopodiché estrae dalla pila il veicolo successivo per ripete la stessa operazione. L'algoritmo termina quando $H = \emptyset$ (figura 2.18).

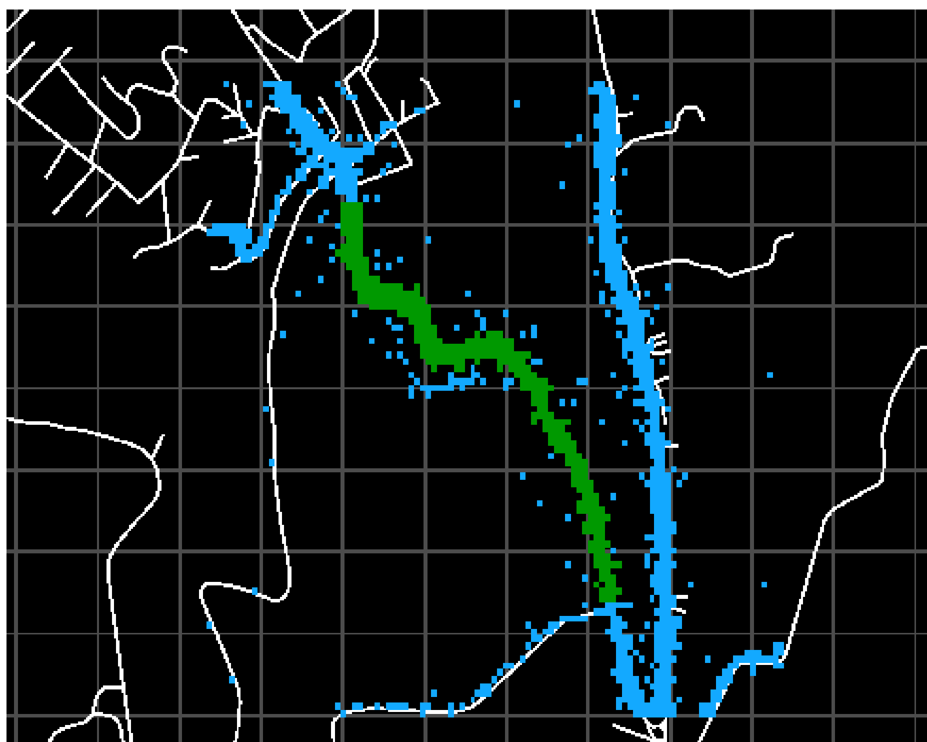

Figura 2.18: Risultato finale dopo l'attivazione di *selezionaAuto*, che marca in modo corretto i veicoli appartenenti al flusso veicolare anomalo e scarta quelli la cui distanza dal flusso stesso è troppo elevata

Il flusso di veicoli $F = \{f_0, f_1, \ldots, f_{h-1}\}$, così determinato, si presta bene per la ricerca di una soluzione adatta a tracciare in modo automatico una polilinea $P = \{p_0, p_1, \ldots, p_{k-1}\}$ passante per F. Un possibile metodo è quello di considerare una polilinea di partenza formata da un solo arco, che connette i nodi estremi (u, v) selezionati in precedenza. Si esegue l'associazione tra F e $P = (u, v)$, inserendo nel campo *strada* dei veicoli di F il cursore dell'unico arco di P. Essendo P formato da un solo arco, tutti i veicoli

avranno lo stesso cursore. Si determina il veicolo $f_l = (x, y)$ più distante da P, calcolando le distanze euclidee tra P e tutti gli elementi di F. Si spezza P in corrispondenza di f_l, aggiungendo un nuovo punto in $P = (u, f_l, v)$. Si osserva che f_l, il veicolo più lontano, è posizionato nei bordi del flusso, mentre P deve passare in mezzo a F. Si riesegue l'associazione tra F e P, associando i veicoli agli archi più vicini e si modificano le coordinate di tali archi, calcolando la retta di regressione lineare passante per il corrispondente sottoflusso a loro associato. Tale operazione si ripete fin quando le distanze euclidee massime dei veicoli dai rispettivi archi associati, risultano superiori di un certo δ fissato. Si realizza tale procedura nel metodo *eseguiInterpolazione*, che determina la polilinea passante per F. Il metodo salva in uno stack temporaneo i vari passaggi delle interpolazioni, eseguite con la sopradescritta tecnica di regressione lineare, in modo da poter fornire la registrazione di tutte le fasi della procedura. Nel contempo può continuare a eseguire nuove interpolazioni attivando il metodo *nuovaInterpolazione*.

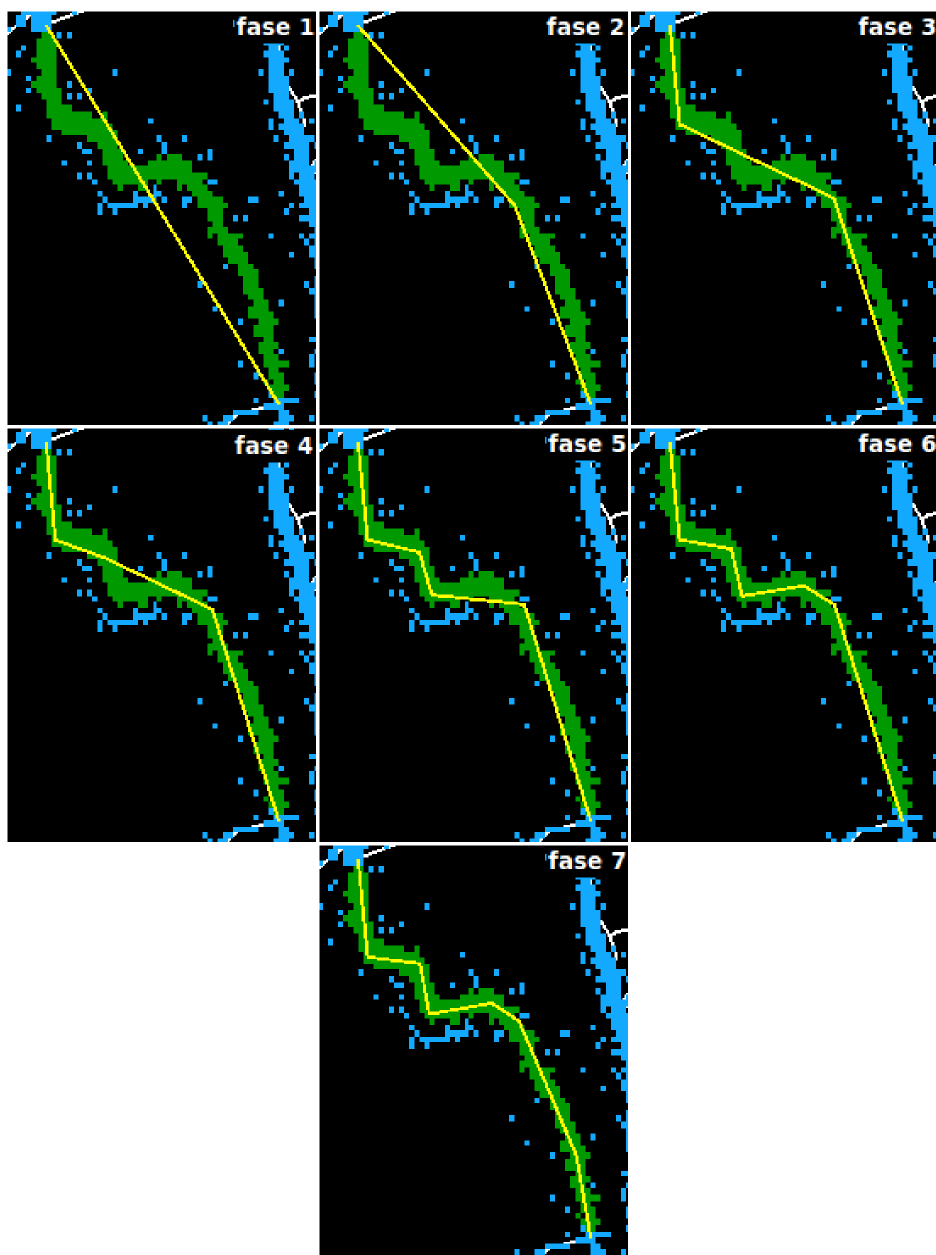

Figura 2.19: Fasi di ricostruzione automatica di una polilinea

Esaminando l'immagine satellitare (figura 2.15) della strada appena inserita, si nota che la polilinea ricostruita automaticamente si adatta con una buona approssimazione alla realtà. La classe fornisce anche un metodo che permette di ricavare automaticamente il senso di percorrenza della polilinea

creata. Per fare ciò effettua una comparazione tra la direzione del moto veicolare del flusso con la posizione nello spazio dei rispettivi archi associati (vedi cap. 1.6). Se tutti i veicoli percorrono gli archi dal nodo di testa a quello di coda, allora P è una polilinea con nodi estremi u e v della forma $(u, v)^{\rightarrow}$. Se i veicoli percorrono gli archi dal nodo di coda a quello di testa, allora $P = (u, v)^{\leftarrow}$. Altrimenti la strada è a doppio senso di percorrenza, ossia $P = (u, v)^{\leftrightarrow}$.

Capitolo 3

Semplificazione della rete stradale

In questo capitolo si illustrano i metodi e gli algoritmi per eliminare le polilinee del grafo che, pur mantenendo le proprietà di connessione tra i nodi, non forniscono informazioni rilevanti alla correzione del grafo stesso e appesantiscono inutilmente il sistema informatico in termini di sovraccarico di memoria e di tempi di esecuzione. A tal fine, in via sperimentale, si individuano due metodi per determinare un sottografo $G' \subseteq G$ connesso, che contiene solo le strade trafficate. Entrambe le soluzioni si rifanno allo studio del problema che ricerca i cammini minimi tra due nodi. Tale problema è noto in letteratura e si risolve con algoritmi efficienti, che operano su classi di grafi con pesi non negativi. Tuttavia, la realizzazione di una procedura che iteri molte volte la ricerca di un cammino minimo tra due nodi, causa forti rallentamenti soprattutto nella fase di inizializzazione delle strutture dati. Per cui si implementa un algoritmo in grado di effettuare un numero arbitrario di cammini minimi consecutivi, in modo che la fase di inizializzazione non infici il costo computazionale complessivo della ricerca stessa.

3.1 Cammini minimi

Sia $G = (N, A)$ un grafo orientato, con pesi $c_{u,v} \in \mathbb{R}$ per ogni arco $(u, v) \in A$. Dato un nodo iniziale di partenza $sappN$, si vuole trovare un percorso T, da s a u per ogni $u \in N$, tale che la somma dei pesi degli archi del cammino sia la più piccola possibile [1] (problema dei cammini minimi). In generale non è detto che il cammino da s a u esista. Se u non è

raggiungibile da s allora non vi è alcun percorso che connette i due nodi e la somma dei pesi per arrivare a u sarebbe infinita. Potrebbe anche esistere in G una catena ciclica C, la cui somma dei pesi degli archi sia un valore negativo c. In questa classe di grafi non si possono determinare i cammini minimi tra i nodi, in quanto il problema non è limitato inferiormente. L'inclusione di C nella soluzione permetterebbe di decrementare la somma dei pesi t con il valore c, ma essendo un ciclo, si dovrebbe inserire infinite volte in T in modo da minimizzare t. Così risulta un cammino infinito che non arriverebbe mai al nodo u. Una soluzione ammissibile del problema è data da un insieme di $n-1$ cammini, ognuno dei quali ha come nodo di partenza s e come nodo di destinazione u, per ogni $u \in N$. In questo caso si osserva che due cammini possono condividere un percorso fino ad un certo nodo q, per poi proseguire la traiettoria verso le rispettive destinazioni. Non può accadere la situazione inversa, ossia i cammini dopo aver percorso due traiettorie distinte, non possono convergere in q e se ciò accadesse vi arriverebbero con due percorsi distinti. La formulazione del problema richiede, invece, una soluzione con un unico percorso da s a tutti i nodi di G. Da ciò si evince che T, per essere una soluzione ammissibile, deve essere un albero di copertura radicato in s, che include un cammino da s a ogni altro nodo raggiungibile $u \in N$. Sia T una soluzione ammissibile. Ogni nodo u ha una distanza d_u pari alla somma dei pesi degli archi, che si trovano nell'unico cammino tra s e u in T. Per verificare che T sia una soluzione ottima si usa il seguente teorema.

Teorema 4. *Sono equivalenti i seguenti fatti:*

(a) T *è una soluzione ottima;*

(b) $d_u + c_{u,v} = d_v, \forall (u,v) \in T,$ *e* $d_u + c_{u,v} \geq d_v, \forall (u,v) \in A.$

Dimostrazione.
$(a) \Rightarrow (b)$
Essendo T una soluzione ottima, vale che se $(u,v) \in T$ la distanza d_u sommata al peso dell'arco $c_{u,v}$ deve essere pari a d_v, ossia $d_u + c_{u,v} = d_v$. Se invece $(u,v) \notin T$, la distanza tra s e u sommata a $c_{u,v}$, deve essere maggiore o uguale alla distanza tra s e v. In alternativa a ciò, esiste un cammino da s a v in G, con distanza minore rispetto a quello presente in T, rappresentato proprio dal percorso da s a v seguito dall'arco (u,v).

$(b) \Rightarrow (a)$

Si ipotizza per assurdo che il cammino da s a u in T non sia ottimo. Allora esiste un cammino C da s a u in G, tale che la distanza d'_u da s sia inferiore a quella d_u di T. Si indica con d'_q la distanza da s al nodo intermedio q appartenente a C. Pertanto $d'_s = d_s = 0$, in quanto la distanza da s a s è zero, e d'_u ¡ d_u. In questo scenario c'è anche un arco (q, w) tale che $d'_q \geq d_q$ e $d'_w < d_w$. Ma $d'_q + c_{q,w} = d'_w$ e per ipotesi $d_q + c_{q,w} \geq d_w$. Combinando le due relazioni si ottiene che $d'_w = d'_q + c_{q,w} \geq d_q + c_{u,v} \geq d_w$. Pertanto l'affermazione $d'_w \geq d_w$ contraddice l'ipotesi, in quanto il cammino tra s e w in C, che dovrebbe essere la soluzione ottima, ha una distanza superiore a quella del corrispondente cammino in T.

□

La soluzione T può essere realizzata con un vettore dei padri che, inizialmente, è un albero radicato in s, i cui figli sono i nodi $u \in N$ tali che $u \neq r$. Pertanto $d_s = 0$, mentre per ogni $u \neq s, ds = \infty$.

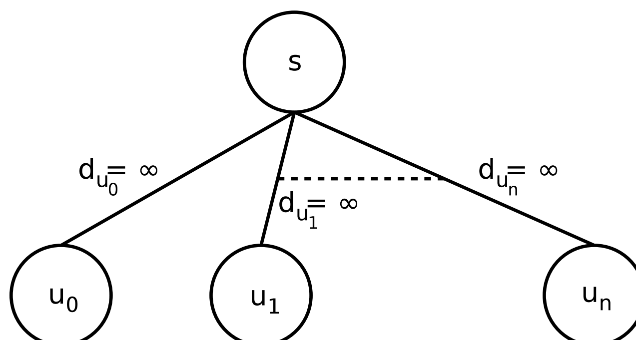

Figura 3.1: Inizializzazione del minimo albero di copertura T

La ricerca del cammino minimo tra s e q inizia inserendo s in una struttura dati S, che contiene i nodi da esaminare. Per ogni $u \in S$ si verificano le condizioni di Bellman sui nodi adiacenti a u, che formano gli archi $(u, v) \in A$. Pertanto si cancella u da S e nel caso esista un nodo v adiacente ad u tale che $d_u + c_{u,v} < d_v$, allora l'inclusione del nodo v in T, il cui padre è u, rappresenta il percorso parziale minimo trovato. Si aggiorna, poi, d_v sommando alla distanza tra s e u, il peso dell'arco (u, v). Qualora v non sia presente in S si effettua il suo inserimento nella struttura dati. Si itera tale procedura fino a quando $S = \emptyset$. Per ottenere il cammino minimo da un qualsiasi nodo u ad

s, si accede alla cella u-esima di T che contiene il nodo padre v al quale u è connesso. Iterando tale procedura si ottengono i nodi intermedi dell'albero di copertura, che rappresentano il cammino minimo da u ad s. Si osserva che se il padre di u è ∞ non esiste alcun cammino possibile che da u porta in s.

3.2 Algoritmo di Johnson

Nel caso in cui la struttura dati S sia un heap H si ottiene l'algoritmo di Johnson. Le regole di priorità prevedono che il nodo u estratto da H deve essere quello con distanza da s minore rispetto a tutti gli altri nodi presenti in H. Se i pesi degli archi sono positivi, ossia per ogni $(u, v) \in A$ $c_{u,v} > 0$, ogni nodo viene estratto da H una e una sola volta e la sua distanza da s è minima. La dimostrazione di ciò si ottiene per induzione sul numero j di iterazioni. Nella prima iterazione ($j = 1$) si ha $d_u = d_s = 0$ che è proprio la distanza minima per andare da s a s. Si assume per ipotesi la tesi vera fino all'iterazione (j–1)-esima e si prova per j. Si estrae da H il nodo u tale che $d_u = \min\{d1_, \ldots, d_k\}$. La distanza d_u, essendo la più piccola presente in H, non dipende dai nodi presenti in H, che hanno distanze superiori a d_u. Pertanto d_u dipende solo dai $k - 1$ nodi estratti nelle iterazioni precedenti che, per ipotesi induttiva, hanno distanza da s minima. Dato che non ci sono pesi negativi e d_u è la distanza più piccola tra quelle di H, si evince che d_u è minima e per le condizioni di Bellman, u non è più reinserito in H. Quindi qualora i pesi degli archi siano non negativi ogni nodo viene estratto da H una e una sola volta. Le operazioni di inserimento, estrazione e aggiornamento dei pesi del nodo in H hanno complessità $O(\log n)$, mentre la verifica dell'appartenenza di un nodo può essere realizzata in $O(1)$ (vedi cap 3.3). Nel caso pessimo l'operazione di aggiornamento dei pesi può essere richiamata su tutti i nodi adiacenti a quello considerato. Pertanto la complessità computazionale complessiva è $O(m \log n)$, con m numero di archi e n numero di nodi di G.

Si osserva che per grafi sparsi, dove il numero di archi m cresce come $O(n)$, la complessità è $O(n \log n)$ e l'algoritmo risulta più efficiente di quello di Dijkstra, che ha complessità $O(n^2)$. Considerando che le reti stradali sono definite da grafi sparsi, si giustifica la scelta di utilizzare l'algoritmo di Johnson, anche se quello di Pape-D'esopo risulta più veloce. Non si è scelto quest'ultimo algoritmo perché la sua struttura S combina assieme una pila e una coda, dove i nodi vengono inseriti in testa o in coda se la loro distanza da s sia rispettivamente infinita o finita. In tal modo non è detto che l'estrazione di un nodo abbia già la distanza minima da s. Quindi per

conoscere il cammino minimo da s a q, fissato anche il nodo di destinazione q, si dovrebbe generare l'intero albero di copertura T e poi prendere il cammino da q fino a s. Invece, con l'algoritmo di Johnson si può sfruttare la proprietà, descritta precedentemente, dove l'estrazione di u da H implica che u abbia già una distanza minima da s. Nel caso in cui $u = q$, allora l'algoritmo termina la ricerca, in quanto trova nell'albero parziale T il cammino minimo da s a q.

3.3 Realizzazione tramite un Time Stamp Heap

Si implementa la struttura dati H da utilizzare nell'algoritmo di Johnson. Gli elementi della coda con priorità possono essere disposti in un vettore, che si può raffigurare come un albero binario addossato a sinistra, dove l'elemento di indice 1 è la radice, che ha come figli gli elementi posizionati negli indici 2 e 3. A sua volta l'elemento di indice 2 ha due figli in corrispondenza degli indici 4 e 5. In generale l'elemento in posizione k del vettore ha il figlio destro nella cella $2k$, quello sinistro nella cella $2k+1$ e il padre in $\lfloor \frac{k}{2} \rfloor$ per ogni $k > 1$. Gli elementi sono disposti seguendo il criterio di ordinamento, dove il valore del padre deve essere più piccolo di quello dei suoi figli. Nel caso delle reti stradali il valore che determina la posizione del nodo u all'interno dell'heap è dato da d_u. Inoltre, per verificare in $O(1)$ la presenza di un nodo in H, è necessario utilizzare un ulteriore vettore di appoggio che, in corrispondenza della u-esima cella, contiene il cursore nella posizione dell'elemento ordinato della struttura ad albero. Pertanto H è implementato da una struttura dati *str_heap*, che contiene i seguenti campi:

- *heap*, vettore che definisce l'albero binario addossato a sinistra;
- *posizione*, vettore che permette di accedere in tempo costante alla posizione del nodo nel vettore *heap*;
- *peso*, vettore associato a *heap* che contiene le distanze di ogni nodo u dell'heap da s;
- k, numero di elementi presenti nell'heap;
- *max_lung*, numero di elementi massimo che l'heap può contenere.

La funzione *initHeap* permette di inizializzare H impostando la dimensione massima pari al numero di nodi n del grafo. Alloca in memoria i vettori *heap*, *posizione* e *peso*, i cui indici delle celle (da 1 a n) corrispondono ai

cursori dei nodi di G, che si ricorda sono un'enumerazione crescente fino a n. Setta il campo *max_lung* a n e k pari a 0, per indicare che l'heap è vuoto. Infine setta gli elementi dei tre vettori con il valore -1. La funzione *insHeap* permette di inserire nell'heap il cursore del nodo u con la relativa distanza da s associata. Incrementa k di un'unità per indicare che l'heap contiene un nodo in più. Inserisce il cursore di u nella prima cella libera di *heap*, setta $peso[u] = d_u$ e $posizione[u] = k$. Effettua la comparazione tra d_u e la distanza d_v associata al nodo padre v di u. Nel caso in cui d_u è minore di d_v scambia d_u con d_v. Aggiorna il vettore *posizione* in modo che $posizione[u]$ contenga il nuovo indice di *heap* dove si trova u ossia $\lfloor \frac{(h->n}{2} \rfloor$ e setta $posizione[v]$ con il valore $h->n$. Esegue iterativamente tale procedura per ogni nodo v, padre di u, fino a quando $d_u < d_v$. Se, nel caso pessimo, d_u è minore di tutte le distanze presenti in H, u diventa la nuova radice in $O(logn)$ iterazioni, in quanto l'heap è una struttura che simula un albero binario.

La funzione *readHeap* permette di leggere la radice di H. Tale operazione si esegue in tempo costante $O(1)$, in quanto la radice corrisponde alla cella di indice 1 del vettore *heap*. Anche l'accesso al peso si può eseguire in tempo costante. Infatti $heap[1]$ contiene il cursore del nodo radice u che, usato come indice di accesso del vettore peso, restituisce in $O(1)$ il valore d_u.

La funzione *popHeap* permette di rimuovere il nodo radice da H. La cancellazione della radice spezza H in due sottoalberi separati. È necessario, quindi, spostare gli elementi in modo da avere un nuovo nodo radice, il cui peso sia il più piccolo degli altri nodi. Per fare ciò la funzione sposta l'ultimo elemento dell'heap in testa e applica la procedura opposta a quella vista per l'inserimento. Pertato, dato u il nodo spostato alla radice di peso d_u, effettua la comparazione tra i due figli v e q, le cui distanze sono rispettivamente d_v e d_q. Considera d_* la distanza minima tra d_u e d_v, e v_* il corrispondente nodo associato. Compara d_u con d_* e nel caso in cui $d_* < d_u$, porta v_* alla radice per porre u come figlio di v_*. Essendo v_* il valore più basso tra i pesi dei fratelli del padre, allora v_* risulta il più piccolo nodo la cui distanza è inferiore a quella di tutti gli altri. Tale procedura si applica sul sottoalbero di u fino a quando d_u è maggiore di d_*.

La funzione *aggiornaPeso* permette di modificare il peso ${d_u}'$ di un nodo u già presente in H. Aggiorna la cella $peso[u]$ con il valore ${d_u}'$. Poichè le condizioni di Bellman assicurano che ${d_u}'$ sia sempre minore di d_u, la funzione esegue un eventuale spostamento di u verso la radice, eseguendo una procedura del tutto analoga a quella vista per *insHeap*.

Si esegue un test di velocità realizzato mediante un heap H, per verificare la buona rispondenza dell'algoritmo di ricerca dei cammini minimi. Si prova l'algoritmo che genera l'intero albero dei cammini minimi di alcuni nodi

$\{u_0, u_1, \ldots, u_{h-1}\}$ del grafo di Roma, composto da circa $n = 200000$ nodi. Analizzando il tempo di esecuzione si stima che, se h tende a n, impiegherebbe circa 22 ore per trovare tutti i cammini minimi dei nodi del grafo. Tale risultato, esageratamente pessimistico, non boccia l'algoritmo perché in realtà per la semplificazione del grafo non effettua il calcolo di tutte le possibili soluzioni dei cammini tra tutti i nodi, ma solo brevi tratti, dove i nodi di partenza e di destinazione sono fissati. Pertanto si esegue un nuovo test dove si calcolano percorsi medio-lunghi tra due nodi u e v. I risultati indicano un notevole abbassamento del tempo computazionale che, per h che tende ad n, è circa di 8 ore. Reimpostando il test su percorsi molto corti, che contengono non più una decina di nodi intermedi, il tempo impiegato dal test, per h che tende a n, è di circa di 46 minuti. Il risultato ottenuto in quest'ultimo test è ancora lontano dalla sufficienza. Si rivede l'algoritmo di Johnson per individuare l'eventuale esistenza di punti nevralgici, che fanno notevolmente rallentare il sistema e se del caso, implementare le possibili modifiche. Si individua un degrado delle prestazioni nella fase di inizializzazione delle strutture dati con i valori di default pari a -1. Si disattiva la ricerca del cammino minimo e si misura, per h che tende a n, il tempo che la procedura impiega per inizializzare ad ogni iterazione i vettori dell'heap. Con grande sorpresa emerge che il tempo impiegato da questa sola fase è di circa 46 minuti, di poco inferiore a quello complessivo dell'intero test. Si ricerca una soluzione che permetta di inizializzare una e una sola volta le strutture dati, per poi essere utilizzate nella ricerca di un numero arbitrario di cammini minimi consecutivi. Per risolvere il problema si incapsula l'heap in una struttura dati, che contiene un vettore di dimensione n denominato *orologio* e un campo intero chiamato *giro*. Nella fase di inizializzazione oltre all'heap H si alloca in memoria il vettore *orologio*, settando le celle con il valore di default -1 e si pone $giro = -1$. Alla chiamata della funzione, che ricerca un cammino minimo, si incrementa il campo *giro* di un'unità. Ogni nodo q esaminato durante l'esecuzione dell'algoritmo è marcato nella corrispondente cella q-esima del vettore *orologio* con il valore di *giro*. Pertanto, siano $Q = \{q_0, q_1, \ldots, q_{h-1}\}$ i nodi inseriti in H durante l'algoritmo e $S = s_0, s_1, \ldots, s_{n-h}$ quelli non visitati. Si osserva che $Q \cup S = N$, ossia l'unione dei due insiemi corrisponde all'insieme complessivo dei nodi del grafo. Allora si ha che per ogni $q \in Q$, $orologio[q] = giro$ e per ogni $s \in S$ $orologio[s] < giro$. Data questa proprietà si può attivare una nuova ricerca di cammini minimi senza dover reinizializzare ogni volta H e *orologio*. Infatti, all'inizio dell'algoritmo tutti i nodi di N appartengono a S, ossia hanno un valore in *orologio* inferiore a *giro*. Il primo nodo a essere inserito in H è quello u di partenza. Ciò determina che $Q = \{u\}$ e $S = N \setminus \{u\}$. La ricerca del cammino minimo prevede l'estrazione di u da H e la verifica delle condizioni

di Bellman sui nodi adiacenti. L'inserimento di tali nodi in H implica il loro spostamento da S a Q. Tale procedura si ripete fino al termine della ricerca. Poiché in H sono presenti solo ed esclusivamente i nodi appartenenti a Q, per svuotare H basta settare il numero di elementi di H a zero e incrementare il campo *giro* di un'unità. In tal modo si simula un reset virtuale dei vettori di H, dove gli elementi presenti sono marcati come obsoleti, in quanto hanno un valore *giro* inferiore a quello corrente.

Ripetendo il test di velocità sulla ricerca di cammini minimi tra nodi vicini, si abbatte completamente il tempo computazionale impiegato nell'inizializzazione ripetuta dei vettori di H. Infatti la ricerca di 200000 nodi, pari al numero di quelli presenti nella rete stradale, viene eseguita in circa 13 secondi contro i 46 minuti rilevati precedentemente. Al nuovo heap si dà il nome *Time Stamp Heap*, in quanto si basa essenzialmente sul riconoscimento dei nodi obsoleti, trattati in precedenti ricerche di cammini.

3.4 Algoritmo A*

La ricerca del cammino minimo con l'algoritmo di Johnson garantisce la correttezza della soluzione sui grafi che non hanno cicli negativi. Evidenziando i nodi intermedi esaminati durante la ricerca di un cammino, si può notare che la visita dei nodi si espande in modo circolare dal nodo di partenza. Ciò accade perché, in generale, i grafi non permettono di ricavare informazioni aggiuntive sulla collocazione spaziale del nodo di destinazione rispetto a quello considerato. Nel caso dei grafi stradali, invece, tali informazioni sono disponibili. L'algoritmo $A*$ è uno dei più efficienti e trova un buon impiego nella ricerca euristica dei cammini minimi su grafi stradali. La determinazione dell'ordine di visita dei nodi è basata sulla funzuione $f(u) = d_u + h(u)$ per ogni $u \in N$, dove d_u è la somma dei pesi degli archi del cammino intermedio che connette il nodo iniziale ad u, mentre $h(u)$, detta funzione euristica, stima la distanza tra u e il nodo di arrivo. La funzione $h(u)$ deve soddisfare il requisito di ammissibilità, ossia non deve sovrastimare la distanza di ogni nodo dall'arrivo. Se h è anche monotona si ottiene una versione più efficiente di $A*$, in quanto l'inserimento dei nodi nell'heap è eseguito una ed una sola volta e al momento dell'estrazione, il nodo ha distanza minima dalla destinazione. La funzione h è monotona, se soddisfa la condizione $h(u) \leq c_{u,v} + h(v)$ per ogni coppia di nodi adiacenti u e v. Si osserva che il criterio di monotonia altro non è che una forma di disuguaglianza triangolare. Tale proprietà è valida negli spazi metrici piani, dove la distanza euclidea tra i nodi u e v è sempre minore o uguale a quella tra u e un certo z, sommata a quella tra z e v. Pertanto, sia $h(u)$ la funzione euristica che restituisce la distanza in

linea d'aria tra il nodo u e quello di arrivo. La validità della disuguaglianza triangolare permette di dire che h è monotona. In queste condizioni $A*$ restituisce il cammino minimo tra il nodo di partenza e quello di destinazione, effettuando la ricerca dei nodi situati nella traiettoria in linea d'aria (vedi figura 3.2). Dai test di velocità appositamente eseguiti, però, emerge un risultato interessante. L'algoritmo $A*$ dà risposte nettamente superiori a quelle di Johnson, se utilizzato nell'ipotesi in cui esista almeno un cammino tra il nodo di partenza e quello di destinazione. Con l'algoritmo di Johnson i cammini tra due nodi distanti sono calcolati in circa 14 secondi, mentre con $A*$ in appena 4. Ciò è possibile in quanto $A*$, visitando i nodi sulla traiettoria del collegamento diretto tra partenza e arrivo, ne esamina mediamente un terzo rispetto al Johnson. Tuttavia, se il cammino tra due nodi non esiste, le prestazioni di $A*$ risultano più scadenti rispetto a quelle date dal Johnson. Poiché il codice dei due algoritmi è pressoché identico, risulta evidente che la causa del degrado delle prestazioni in questione è data proprio dalla funzione h. A ogni iterazione, infatti, h calcola la distanza euclidea tra i nodi visitati e quello di destinazione. Il calcolo della radice quadrata è un'operazione molto pesante, che inficia in modo negativo la velocità di $A*$. Pertanto nell'ipotesi in cui il cammino minimo non esista, anche se entrambi gli algoritmi effettuano il calcolo dell'intero albero di copertura su tutto N, si preferisce l'algoritmo di Johnson che, non avendo la funzione euristica da valutare, impiega meno tempo rispetto ad $A*$.

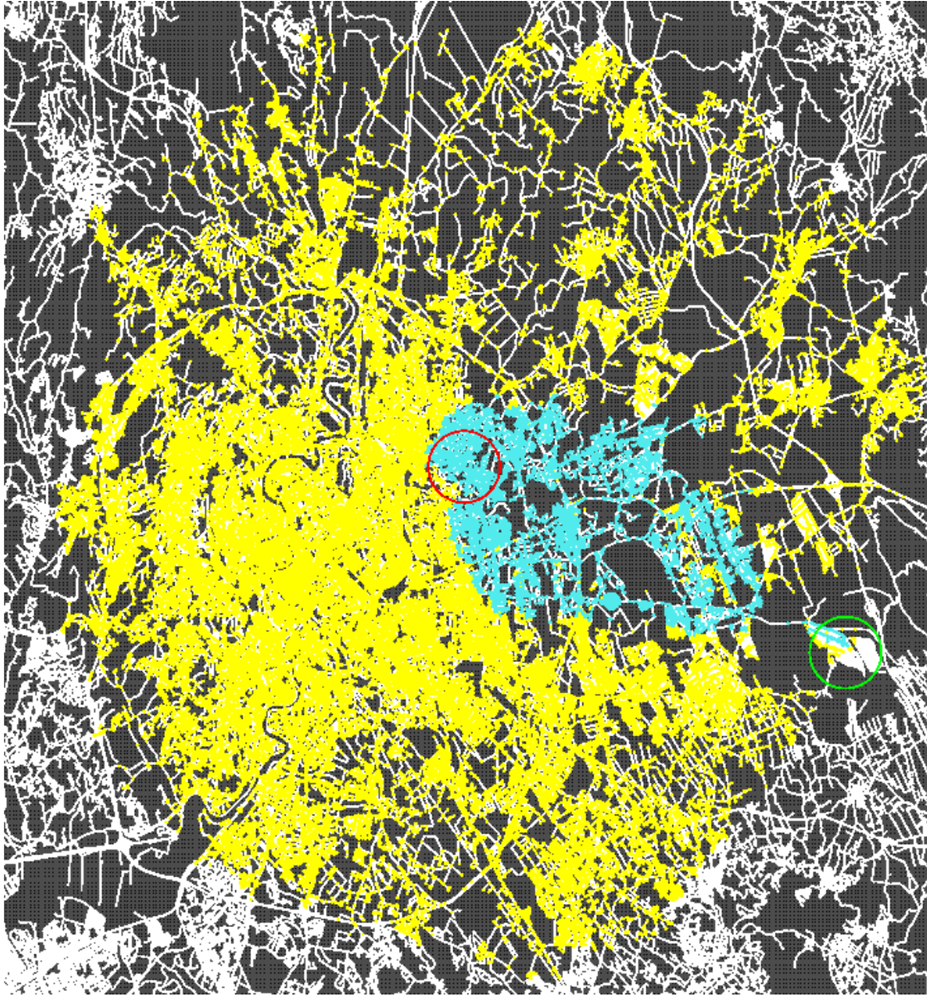

Figura 3.2: ricerca del cammino minimo da un nodo di partenza (cerchio in rosso) ad un nodo di destinazione (cerchio in verde) con l'algoritmo di Johnson (in giallo) e con A* (in azzurro)

3.5 Semplificazione a priori

Si implementa un algoritmo che determina un sottografo $G' = (N', A')$ derivante da $G = (N, A)$, tale che G' sia connesso e ogni nodo di N raggiungibile da tutti gli altri. Quest'ultima proprietà indica che i nodi possono comunicare con tutti quelli che appartengono a N. Formalmente si dice che i nodi fanno parte di un'unica classe di equivalenza[2] formata dagli stessi.

Qualora un nodo comunichi solo con un sottogruppo di N, si forma una classe di equivalenza composta da nodi mutualmente comunicanti tra loro. Dato che in una rete stradale ogni nodo deve comunicare con gli altri, è necessario mantenere il grafo con un'unica classe di equivalenza. La funzione prevede proprio ciò, ossia accerta "a priori" (da qui ne deriva il nome), che durante l'intero processo di rimozione rimangano sempre mantenute le proprietà sopraindicate, per cui a ogni fase iterativa il grafo risultante resta costantemente connesso, mantenendo i nodi in un'unica classe di equivalenza. La rimozione delle strade cieche è banale, in quanto la loro eliminazione dal grafo non compromette le proprietà da mantenere.

Si indica con il termine rango n del nodo u il numero di polilinee connesse al nodo estremo $u \in V_n$. Se il rango di u è pari a 2, le polilinee con lo stesso senso di percorrenza si possono unire in una sola. Così facendo si può rimuovere u dal grafo, lasciando inalterate le proprietà di connessione. La semplificazione a priori è formata dai seguenti algoritmi:

- *rimuoviVicoli*, per la rimozione dei vicoli;
- *unisciPolyRango_2*, per la rimozione dei nodi estremi di rango 2;
- *rimuoviPolyRango_n*, per la rimozione delle polilinee connesse a nodi di rango n.

La rimozione di una polilinea P è data dalla comparazione tra il flusso ϕ_P, a lei associato, con un valore soglia δ fissato.

$$\phi_P = \frac{v_P}{t}$$

Dove v_P è il numero di veicoli che percorrono P e t l'intervallo di tempo delle rilevazioni (in questo caso $t = 1$ mese). Nel caso in cui $\phi_P < \delta$ allora si può rimuovere P, in quanto le informazioni sul traffico a lei associato sono troppo scarne. Il processo rimuove a priori P, se e solo se, $\phi_P < \delta$ e i nodi del grafo, dopo tale rimozione, restano connessi in un'unica classe di equivalenza. La procedura viene eseguita iterativamente fino alla rimozione completa degli elementi superflui dal grafo.

3.5.1 Rimozione delle strade cieche

Si Implementa l'algoritmo rimuoviVicoli, che effettua la rimozione delle strade cieche. Una polilinea P rappresenta una strada cieca se uno dei due nodi estremi non connette altre polilinee, ossia ha come unico cursore quello di P. L'algoritmo esegue una scansione completa delle polilinee $P \in V_p$. Per

ognuna di esse accede ai cursori dei nodi estremi di testa u_t e di coda u_c. Se almeno uno dei nodi, tra u_t e u_c, ha come unico cursore quello di P e d_P è inferiore a δ, allora l'algoritmo attiva la funzione *rimuoviPoly(P)* del framework *OpGrafo* (vedi cap 2.7) che elimina P. Ponendo $\delta = \infty$, si effettua un test sul grafo, per constatare gli effetti prodotti dalla semplificazione. Alla fine emerge un grafo semplificato G' che, a seguito della rimozione di tutte le strade cieche, risulta formato solo da nodi estremi $u \in V_n$ di rango minimo pari a 2.

3.5.2 Rimozione dei nodi di rango 2

Per snellire ulteriormente il grafo si pensa di unire in una unica polilinea quelle connesse tra loro da un nodo estremo di rango 2, in quanto risulta superfluo tenerle distinte da un nodo, quando si potrebbero considerare nella loro interezza. Si implementa l'algoritmo *unisciPoly*, per unire le polilinee P^a e P^b, aventi lo stesso senso di percorrenza, che sono connesse tra loro dal nodo estremo u di rango 2. La funzione scansiona l'intero vettore V_n alla ricerca di nodi di rango 2, che connettono polilinee con lo stesso senso di marcia. Siano $P^A = \{p^a{}_0, p^a{}_1, \ldots, p^a{}_{k-1}\}$ e $P^b = \{p^b{}_0, p^b{}_1, \ldots, p^b{}_{h-1}\}$ due polilinee di nodi estremi (u, v) e (v, z). Allora, P^a e P^b si possono unire in un'unica polilinea P, se v ha rango pari a 2 e valgono una delle seguenti condizioni:

- $(u, v)^{\leftrightarrow} \wedge (v, z)^{\leftrightarrow}$;
- $(u, v)^{\rightarrow} \wedge (v, z)^{\leftarrow}$;
- $(u, v)^{\leftarrow} \wedge (v, z)^{\leftarrow}$;
- $(u, v)^{\rightarrow} \wedge (z, v)^{\leftarrow}$;
- $(v, u)^{\leftarrow} \wedge (v, z)^{\rightarrow}$.

Effettua la fusione tra P^a e P^b in base al segno del cursore memorizzato in v. Nel caso in cui v contiene i cursori positivi di P^a e di P^b, allora entrambe si connettono a v con il rispettivo nodo di testa. Inserisce in ordine i nodi di P^a dalla coda fino alla testa e quelli di P^b dalla testa fino alla coda, ossia $P = \{p^a{}_{k-1}, p^a{}_{k-2}, \ldots, p^a{}_0, p^a{}_1, p^b{}_2, \ldots, p^b{}_{h-1}\}$. Se, invece, v contiene i cursori di P^a e P^b negativi, allora inserisce i nodi di P^a dalla testa alla coda e di P^b dalla coda alla testa; ovvero $P = \{p^a{}_0, p^a{}_1, \ldots, p^a{}_{k-1}, p^b{}_{h-2}, p^b{}_{h-3}, \ldots, p^b{}_0\}$, in quanto entrambe le polilinee si connettono a v con i rispettivi nodi di coda. Così facendo P ha sempre il nodo di testa pari a u, quello di coda pari a z e varia solo l'attribuzione del suo senso di marcia, dato dalle seguenti relazioni:

- se P^a e P^b sono a doppio senso di percorrenza, allora $P^{a\leftrightarrow} \Rightarrow P^{b\leftrightarrow} \Rightarrow P^{\leftrightarrow}$;
- se P^a è formato dai nodi estremi $(v,u)^{\rightarrow}$, allora P^b deve essere necessariamente composto da quelli $(v,z)^{\leftarrow}$ e P ha nodi estremi $(u,z)^{\leftarrow}$, ossia $P^{a\rightarrow} \Rightarrow P^{b\leftarrow} \Rightarrow P^{\leftarrow}$;
- se P^a è formato dai nodi estremi $(u,v)^{\rightarrow}$, allora P^b deve essere necessariamente composto da quelli $(z,v) \leftarrow$ e P ha nodi estremi $(u,z) \rightarrow$, ossia $P^{a\rightarrow} \Rightarrow P^{b\leftarrow} \Rightarrow P^{\rightarrow}$;
- se P^a è formato dai nodi estremi $(u,v)^{\leftarrow}$, allora P^b deve essere necessariamente composto da quelli $(z,v)^{\rightarrow}$ e P ha nodi estremi $(u,z)^{\leftarrow}$, ossia $P^{a\leftarrow} \Rightarrow P^{b\rightarrow} \Rightarrow P^{\leftarrow}$.

Dalle relazioni dei sensi di percorrenza, si evince che se v ha entrambi i cursori di P^a e P^b positivi, allora P ha senso di marcia pari a quello di P^b, altrimenti è pari a quello di P^a.

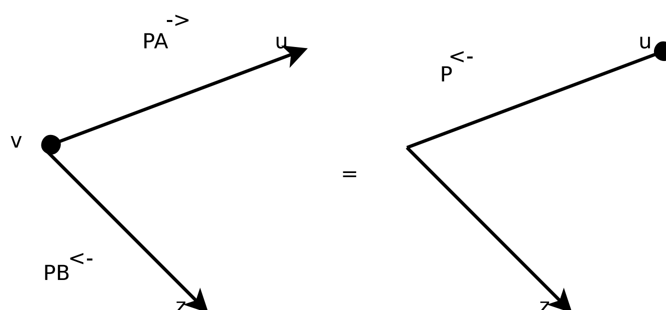

Figura 3.3: Unione di due polilinee che si connettono con il nodo di testa

La figura 3.3 mostra un esempio di polilinee a senso unico P^a e P^b, che si connettono a u con il nodo di testa. Queste, per essere percorribili, devono avere sensi unici di marcia opposti, ossia P^a ha $s_{v,u} = 1$ e P^b ha $s_{v,z} = 2$, o viceversa. Dato che P ha il nodo di testa pari a u e quello di coda pari a z, eredita il senso di percorrenza da quello di P^b.

Nel caso in cui v contiene i cursori di P^a e P^b con segni opposti, la polilinea che si connette a v con il nodo di testa, implica la connessione dell'altra con quello di coda o viceversa. Se P^a si connette a v con il nodo di testa e P^b con quello di coda, allora P è formata dai nodi $\{{p^a}_{k-1}, {p^a}_{k-2}, \ldots, {p^a}_0, {p^b}_{h-2}, {p^b}_{k-3}, \ldots, {p^b}_0\}$, altrimenti $P = \{{p^a}_0, {p^a}_1, \ldots, {p^a}_{k-1}, {p^b}_2, {p^b}_3, \ldots, {p^b}_{h-1}\}$. Anche in quest'ultimo caso l'algoritmo determina P con il nodo di testa pari a u e quello di coda pari a v. Il senso di percorrenza lo determina in base alle seguenti relazioni:

- se P^a e P^b sono a doppio senso di percorrenza, allora $P^{a\leftrightarrow} \Rightarrow P^{b\leftrightarrow} \Rightarrow P^{\leftrightarrow}$;S

- se P^a è formato dai nodi estremi $(v,u)^{\rightarrow}$, allora P^b deve essere necessariamente composto da quelli $(z,v)^{\rightarrow}$ e P ha nodi estremi $(u,z) \leftarrow$, ossia $P^{a\rightarrow} \Rightarrow P^{b\rightarrow} \Rightarrow P^{\leftarrow}$;

- se P^a è formato dai nodi estremi $(v,u)^{\leftarrow}$, allora P^b deve essere necessariamente composto da quelli $(z,v)^{\leftarrow}$ e P ha nodi estremi $(u,z)^{\leftarrow}$, ossia $P^{a\leftarrow} \Rightarrow P^{b\leftarrow} \Rightarrow P^{\leftarrow}$;

- se P^a è formato dai nodi estremi $(u,v)^{\rightarrow}$, allora P^b deve essere necessariamente composto da quelli $(v,z)^{\rightarrow}$ e P ha nodi estremi $(u,z)^{\rightarrow}$, ossia $P^{a\rightarrow} \Rightarrow P^{b\rightarrow} \Rightarrow P^{\rightarrow}$;

- se P^a è formato dai nodi estremi $(u,v)^{\leftarrow}$, allora P^b deve essere necessariamente composto da quelli $(v,z)^{\leftarrow}$ e P ha nodi estremi $(u,z)^{\leftarrow}$, ossia $P^{a\leftarrow} \Rightarrow P^{b\leftarrow} \Rightarrow P^{\leftarrow}$.

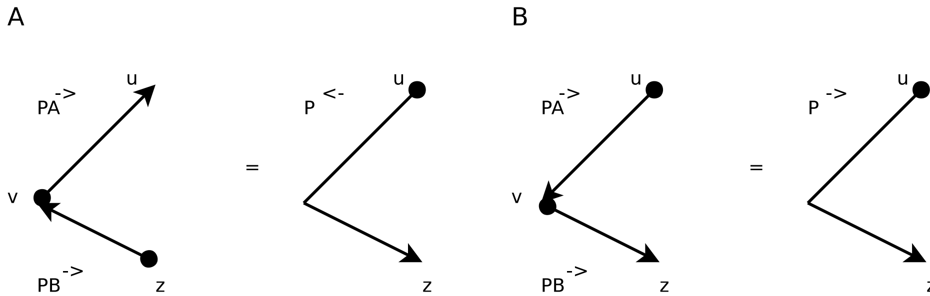

Figura 3.4: Unione di due polilinee che si connettono con nodi estremi diversi

La figura 3.4 *(A)* mostra un esempio di polilinee a senso unico P^a e P^b che si connettono a u rispettivamente con il nodo di testa e con quello di coda. Per essere percorribili devono avere sensi unici di marcia uguale, ossia P^a ha $s_{v,u} = 1$ e P^b ha $s_{z,v} = 1$ o viceversa. Dato che P ha il nodo di testa pari a u e quello di coda pari a z, il suo senso di percorrenza è inverso a quello di P^a e P^b. La figura 3.4 *(B)*, invece, raffigura un senso unico di percorrenza, dove P^a è connessa a v con il nodo estremo finale, mentre P^b è connessa a v con quello iniziale. In questo caso P eredita il loro senso di marcia, in quanto ha il nodo estremo iniziale pari a u e quello finale pari a v.

L'algoritmo concretizza l'unione di P^a e P^b in P, attivando le funzioni *rimuoviPoly(P^a)*, *rimuoviPoly(P^b)* e *insPoly(P)* che provvedono automaticamente a eliminare il nodo u da V_n, non appena u resta senza cursori associati. Si itera la procedura su tutti i nodi estremi di rango 2 fino alla totale rimozione degli elementi superflui, con la quale si conclude l'operazione di semplificazione.

3.5.3 Rimozione delle polilinee

Si realizza l'algoritmo che rimuove una generica polilinea P di nodi estremi (u, v), tale che ϕ_P è minore di δ, ossia il flusso di traffico presente in P è inferiore alla soglia fissata. La rimozione di P deve mantenere il grafo connesso con una sola classe di equivalenza, alla quale appartengono tutti i nodi di N. La connessione si mantiene se esiste un cammino alternativo, non passante per P, che connette u con v. Questa non è, però, una condizione sufficiente a garantire il mantenimento di una sola classe di equivalenza. Nel caso in cui l'unico cammino che connette u con v è a senso unico, si creano due distinte classi di equivalenza, composte rispettivamente da tutti i nodi mutualmente raggiungibili da u e v. Perciò le condizioni necessarie e sufficienti, che permettono la rimozione di P, sono date dall'esistenza di due cammini C_1 e C_2, non passanti per P, che connettono rispettivamente u con v e viceversa. Si procede alla rimozione in base alla seguente tipologia di casi; se:

- C_1 e C_2 sono a senso unico, allora elimina P in quanto è l'unica rimovibile;
- C_1 e C_2 sono a doppio senso, rimuove la polilinea con flusso di traffico minore tra P e quelle presenti in C_1 e C_2;
- P e uno dei cammini C_* è a senso unico, rimuove la polilinea meno trafficata tra P e C_*.

Iterando la procedura su tutte le polilinee presenti, si eliminano i cicli dal grafo. Ciò provoca la formazione di nuove strade cieche e nodi di rango 2, che avviano nuovamente le funzioni *rimuoviVicoli* e *unisciPoly* per eliminare gli ulteriori elementi superflui. L'opera di semplificazione termina quando restano nel grafo solo polilinee con flusso maggiore di δ, che con quelle strutturali con flusso minore, contribuiscono a mantenere il grafo connesso con una sola classe di equivalenza.

3.6 Semplificazione a posteriori

Nella semplificazione a priori la rimozione delle polilinee si effettua se e solo se il grafo resta connesso e ha una sola classe di equivalenza. Il processo di rimozione, però, non sempre è speculare all'andamento del flusso veicolare. Ci sono tratti stradali, considerati indispensabili per il mantenimento delle proprietà stesse del grafo, che se rimossi ne causerebbero la sconnessione, per cui non possono essere cancellati. Si implementa, allora, un algoritmo di rimozione a posteriori, per cancellare tutte polilinee con flusso di traffico inferiore alla soglia stabilita, a prescindere dal mantenimento o meno delle proprietà di connessione del grafo (rimozione naturale). La rimozione è detta a posteriori in quanto la connessione dei nodi in un'unica classe di equivalenza del grafo ottenuto, avviene dopo di essa. Si avvia la funzione per rimuovere dal grafo le polilinee con densità di traffico nulla. Per ogni polilinea $P \in G$ se $\phi_P < \delta$ allora P viene rimossa da G, altrimenti P fa parte del grafo G' semplificato.

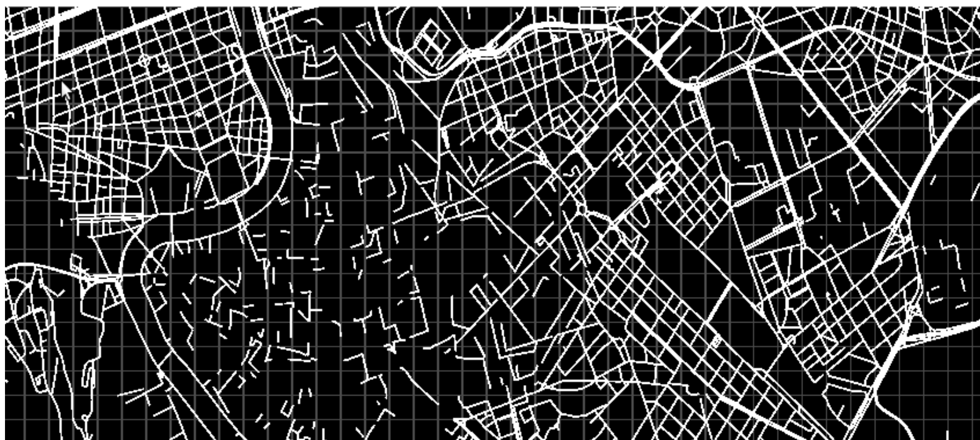

Figura 3.5: Sconnessione del grafo mediante il processo di rimozione naturale

La rimozione naturale delle strade scompone G in tanti sottografi connessi separati. Per ottenere un grafo G' semplificato correttamente è necessario connettere i sottografi tra loro, in modo che i nodi risultanti formino un'unica classe di equivalenza.

3.6.1 Determinazione dei frammenti del grafo

Si implementa un algoritmo per connette i frammenti del grafo determinati dal processo di rimozione naturale delle polilinee. Sia $G = (N, A)$

un grafo sconnesso. Si definisce frammento di G l'insieme dei nodi $F = \{u_0, u_1, \ldots, u_{k-1}\} \in N$ che formano una classe di equivalenza, ossia per ogni $u \in F$ esiste un cammino verso tutti di altri nodi di F. Essendo G sconnesso, il numero di frammenti è maggiore o uguale a zero. Infatti, se G è connesso e i nodi formano un'unica classe di equivalenza, il numero di frammenti è pari a 1, altrimenti è sempre maggiore di 1. Si definisce come connessione transiente dal nodo u verso v l'esistenza di un cammino che connette u con v, ma non permette di ritornare in u da v. In questo caso u e v non possono appartenere allo stesso frammento, in quanto v non comunica con u. Si realizza l'algoritmo *marcaFrammenti*, che marca ogni nodo $uappV_n$ con l'indice del frammento al quale appartiene, detto *id_frammento*. Pertanto i nodi di uno stesso frammento condividono l'*id_frammento* equivalente.

L'algoritmo effettua la scansione dei nodi di V_n. Per ogni $uappV_n$ ricerca un circuito C che parte da u e torna in u. L'esistenza di C per definizione è una classe di equivalenza, in quanto da ogni nodo di C si possono raggiungere gli altri. Tale procedura deve, poi, essere attivata ricorsivamente su tutti i nodi di $C \setminus u$. In questo modo l'esistenza di altri circuiti connessi a C, garantiscono che tutti i nodi interessati formino un'unica classe di equivalenza e quindi un frammento di G. Dato che la procedura ricorsiva è più lenta della corrispondente iterativa, si decide di implementare l'algoritmo in modo che simuli, con una coda Q, le chiamate ricorsive applicate ai nodi. La funzione procede alla scansione di V_n. Per ogni nodo u non marcato, inserisce u in Q, gli assegna l'indice x del frammento F del quale fa parte e lo marca come leader del frammento stesso. Essendo u il leader, la funzione procede nella ricerca dei cicli che tornano a u stesso. Per fare ciò ricava la lista dei nodi $V = \{v_0, v_1, \ldots, v_{k-1}\}$ adiacenti a u e determina la percorribilità da u verso gli elementi di V. Per ogni $v_i \in V$ raggiungibile da u, candida v_i a far parte del frammento F, inserendo la coppia (v_i, u) in Q. Nelle iterazioni successive estrae i nodi v_i con il relativo nodo u. Determina l'esistenza di un cammino C tra v_i e u, per verificare se v_i comunica con u. In caso positivo v_i è marcato con x e entra a far parte di F. Anche gli eventuali nodi intermedi, che determinano il percorso per connettere v_i a u, vengono inclusi in F e inseriti in Q, con associato un valore di default *ok*. Le coppie (v_i, ok), facendo già parte del ciclo che connette v_i a u, sono automaticamente incluse in F e nel caso formino un circuito con v_i, vengono inserite in Q solo per proseguire l'espansione di F ai loro nodi vicini. La funzione, infine, ricava i nodi percorribili adiacenti a v_i per includerli in Q, associarli a v_i e se esiste un percorso che li connette a esso, entrano a far parte di F. Tale procedura si esegue finché da ogni nodo intermedio è possibile costruire un circuito.

Se invece v_i non si connette a u allora viene scartato, in quanto non è possibile costruire un circuito che da u torna in u passando per v_i. Tale

nodo sarà visitato nelle successive fasi dell'algoritmo e incluso in un altro frammento.

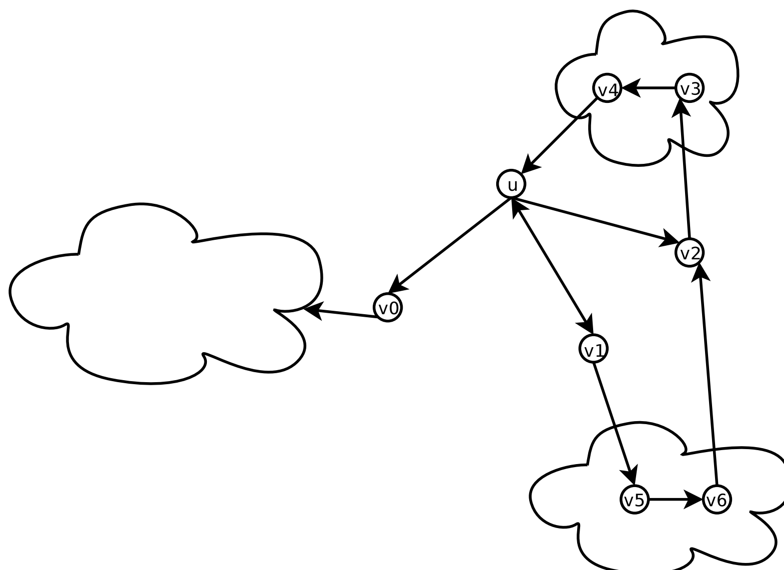

Figura 3.6: Rilevamento del frammento x del nodo leader u

Nell'esempio dato in fig. 3.6 i nodi vicini v_0, v_1, v_2 sono inseriti in Q. Il primo ad essere estratto, v_0 non ha un cammino che permette di raggiungere u e viene scartato. Si osserva, infatti, che u comunica con v_0, ma non viceversa. L'arco che connette u a v_0 è una connessione transiente, dove u e v_0 appartengono a due classi di equivalenza diverse. Il nodo v_1 comunica direttamente con u ed è incluso nel frammento. Il nodo v_2, invece, si connette ad u attraverso un ciclo che passa per i nodi intermedi v_3 e v_4. Pertanto v_2, v_3 e v_4 sono nella stessa classe di equivalenza di u e appartengono al frammento di u. In Q viene estratto v_5, in quanto è un nodo adiacente di un altro appartenente al frammento. Al momento dell'estrazione di v_5, l'algoritmo deve controllare l'esistenza di un circuito che torna a v_1. Il circuito esiste e passa per i nodi v_6, v_2, eu. Quindi anche v_5 e v_6, che fanno parte del circuito di v_1 e di u, sono inseriti nello stesso frammento di u.

3.6.2 Connessione dei frammenti

La generazione di frammenti nel grafo impone l'implementazione di un algoritmo che li connetta tra loro. Per connettere almeno un nodo di un

frammento con quello appartenente ad un altro frammento, si reimmettono nel grafo dei tratti stradali rimossi in precedenza.

Dati due frammenti F_1 e F_2, appartenenti a due classi di equivalenza diverse, la loro unione in un unico frammento F si realizza connettendo i nodi $u \in F_1$ con $q \in F_2$, in modo che da u si raggiunga q e viceversa. In generale $F = F1 \cup F2$ si ottiene con l'inserimento delle polilinee del grafo originario G che permettono di formare un ciclo tra F_1 e F_2.

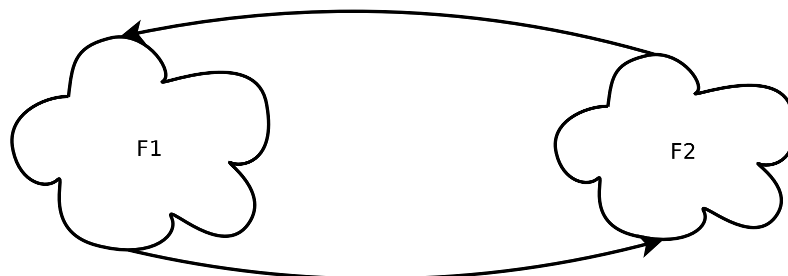

Figura 3.7: Fusione di due frammenti collegati mediante due polilinee che permettono di creare un circuito tra F_1 e F_2

Si realizza l'algoritmo *fondiFrammenti* per eseguire l'unione dei frammenti di G' in un'unica classe di equivalenza tra i nodi. Questo esamina il grafo $G = (N, A)$ di partenza e quello $G' = (N', A')$ ottenuto dopo la rimozione delle polilinee. Ricava il frammento con il maggior numero di nodi e ne considera uno u a esso appartenente. Calcola il minimo albero di copertura radicato in u, per trovare i cammini minimi verso i nodi di G'. Con un movimento circolare visita i nodi per pesi minimi crescenti, partendo da quello con distanza più piccola da u (rif. cap 3.2 – v. algoritmo di Johnson). Quando estrae dall'heap un nodo v, appartenente a un frammento diverso, avvia una nuova procedura di cammini minimi da v a u, per trovare il percorso più breve a u, o ad un altro nodo appartenente alla sua classe di equivalenza. Determinati i cammini tra i due frammenti, ne esegue la fusione marcando i nodi del frammento più piccolo con l'identificativo di quello più grande e inserisce in G' le polilinee mancanti di G per connettere i frammenti. Iterado tale operazione fino al completamento dell'albero di copertura radicato in u, termina la procedura.

Essendo G per definizione connesso e con una classe di equivalenza, anche G' acquisisce tale proprietà grazie all'ausilio delle strade di G a basso flusso veicolare in precedenza scartate. Si implementano gli algoritmi *per distanze minime* e *per flussi massimi*, che in diverso modo connettono i frammenti del grafo. La prima procedura connette i due frammenti con la ricerca del

cammino più breve, dove la verifica delle condizioni di Bellman, eseguita solo sulla lunghezza in metri della polilinea, determina il percorso con la minore quantità di metri possibile. La seconda procedura ricerca, invece, tra i cammini più brevi quello maggiormente trafficato. Il flusso di una polilinea P, come detto, è una quantità ϕ_P, che esprime il rapporto tra il numero di veicoli rilevati v_P e il tempo dell'osservazione t, ossia $\phi_P = \frac{v_P}{t}$. Il peso f_P delle polilinee è determinato dalla combinazione tra la lunghezza in metri l_P di P e la quantità di flusso veicolare ϕ_P presente.

$$f_P(x) = \frac{l_P}{1 + \left(\frac{\phi_P}{\phi_0}\right)^x}$$

Dove ϕ_0 è la quantità minima di flusso richiesta e x un parametro positivo, che determina l'influenza di ϕ_P su l_P. Da prove sperimentali posto $x = 1$ si ottiene un cammino, che privilegia le strade maggiormente trafficate tra l'insieme dei possibili percorsi più brevi.

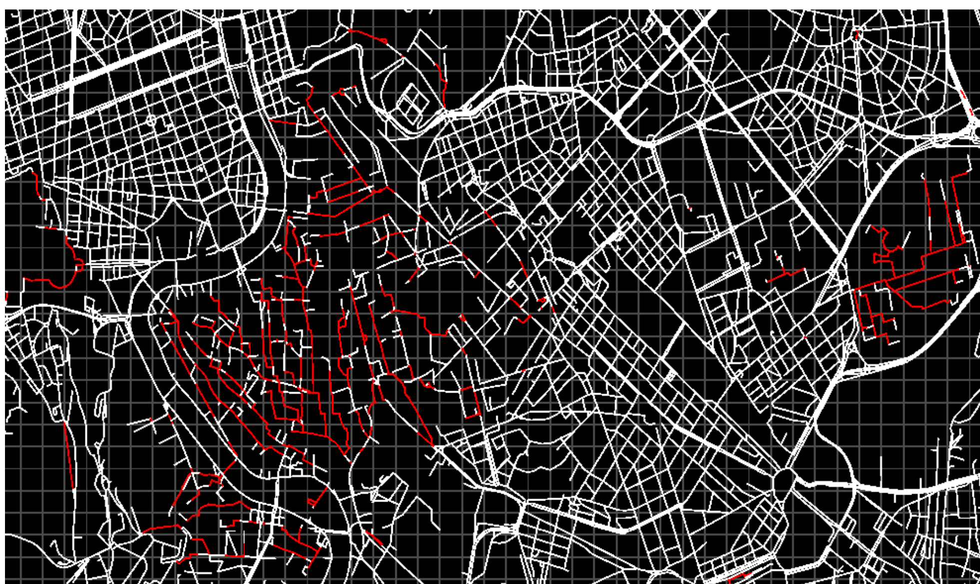

Figura 3.8: Riconnessione dei frammenti di G' in un'unica classe di equivalenza

3.7 Analisi conclusive sulle rimozioni

Si testano le semplificazioni a priori e a posteriori sul grafo della rete stradale di Roma, per determinare la percentuale di elementi rimanenti del

grafo G' rispetto a G. Si riportano nella tabella 3.1 i risultati ottenuti dopo le semplificazioni a priori e a posteriori con flusso δ pari a $1000\frac{v}{t}$ (veicoli rilevati per mese).

Tipo semplificazione	*Polilinee*	*Archi*	*Nodi*
a priori	61%	58%	68%
a posteriori per distanze minime	18%	12%	20%
a posteriori per flussi massimi	17%	11%	19%

Tabella 3.1: Elementi rimanenti in G' dopo il processo di semplificazione

Dai risultati emerge che la semplificazione a posteriori è nettamente più efficace di quella a priori, in quanto si basa sulla rimozione naturale delle polilinee. La conseguente riconnessione dei frammenti del grafo, secondo la ricerca dei cammini minimi, permette di mantenere limitato il numero di polilinee da reimmettere nel grafo stesso. La procedura della semplificazione a priori, invece, non consente l'eliminazione di molti elementi superflui, in quanto la rimozione delle polilinee si concretizza se e solo se il grafo resta connesso. La connessione a posteriori dei frammenti *per flussi massimi*, che ricercano tra i cammini più brevi quello maggiormente trafficato, risulta più efficace della semplice connessione *per distanze minime*. Ciò avviene perché la ricerca privilegia le strade ad alto flusso veicolare presenti in G', anziché quelle già rimosse. La connessione dei frammenti *per distanze minime* ricerca soltanto il percorso più breve, non curandosi del flusso veicolare. Le performances migliori, però, si ottengono da una semplificazione *ibrida*, tra la tecnica a posteriori e la rimozione dei nodi di rango 2, che determina un grafo G' con appena il 12% delle polilinee di G. Lo stesso risultato si rileva per i nodi estremi rimanenti in G', ma non per gli archi, in quanto l'unione di due polilinee ne mantiene invariato il numero (vedi tabella 3.2).

Tipo semplificazione	*Polilinee*	*Archi*	*Nodi*
ibrida per distanze minime	13%	12%	15%
ibrida per flussi massimi	12%	11%	14%

Tabella 3.2: Elementi rimanenti in G' dopo il processo di semplificazione ibrida

Inoltre il grafico della figura 3.9 mostra come nella semplificazione a priori, per δ grande crescente, corrisponde una piccola quantità di metri rimossi, mentre in quelle a posteriori una più consistente. Sull'asse delle ordinate si riporta la percentuale dei metri rimanenti m_δ, a seguito della rimozione naturale di livello δ, sui metri totali m_{tot} del grafo originario.

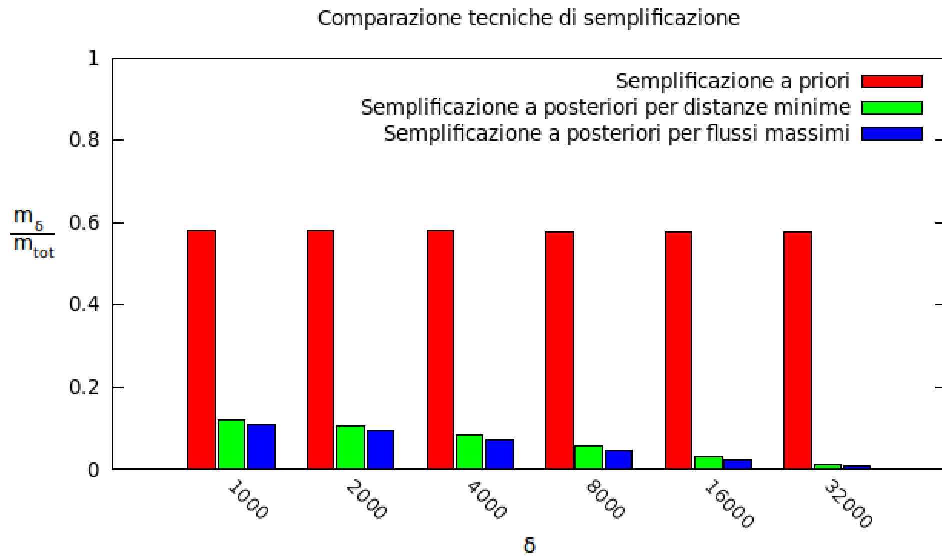

Figura 3.9: Comparazione degli elementi rimanenti in G' delle semplificazioni a priori e posteriori

Si focalizza l'attenzione sulla semplificazione a posteriori e si illustra il comportamento dell'algoritmo nelle fasi di rimozione naturale e connessione dei frammenti. La semplificazione naturale permette di analizzare il comportamento del grafo al variare della soglia δ minima consentita di flusso di traffico. Si eseguono numerosi test di semplificazione a posteriori sul grafo,

impostando, con diverse gradazioni crescenti, il valore δ da una soglia minima (δ piccolo) fino a quella massima (δ grande), il cui risultato di pochissime polilinee rimanenti in G' aumenta col decrescere di δ. Il grafico (fig. 3.10) rappresenta l'andamento delle funzioni, che varia in base al valore δ indicato sull'asse delle ascisse, per la percentuale di metri rimasti $\frac{m_\delta}{m_{tot}}$ indicati sull'asse delle ordinate.

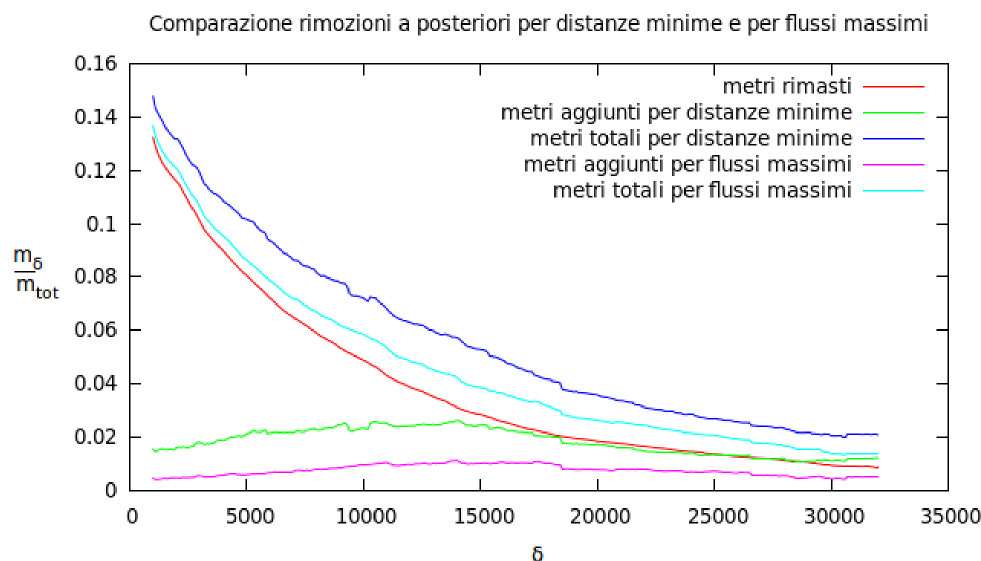

Figura 3.10: Percentuali di metri rimasti, aggiunti e totali con le semplificazioni a posteriori per distanze minime e per flussi massimi

La linea rossa mostra come la rimozione naturale, segua un andamento esponenziale sull'intervallo $\delta \in [1000; 32000]$. Ciò indica che a una variazione lineare di δ, ne corrisponde una esponenziale di metri rimanenti nel grafo. A parità di rimozione naturale, si confronta la tecnica di connessione dei frammenti con le distanze minime (linea verde) e i flussi massimi (linea viola). Si nota che i metri di strade inseriti dalla prima, sono sempre superiori a quelli aggiunti dalla seconda. Anche se potrebbe sembrare un risultato inatteso, si trova una valida giustificazione nel comportamento intrinseco di entrambe le procedure adottate. La prima, infatti, cerca semplicemente la distanza più breve che connette i frammenti tra loro, senza considerare il flusso di traffico delle stesse strade reimmesse. La seconda, invece, calcola il cammino privilegiando le strade con maggiore flusso di traffico, che molto probabilmente, non sono state tolte dalla rimozione naturale. Pertanto, rispetto alla tecnica di connessione per distanze minime, riesce a connettere i frammenti

del grafo, sfruttando più strade presenti e immettendone una minima quantità di quelle rimosse. Negli andamenti di entrambe le procedure si nota che per mantenere il grafo connesso, fino a un $\delta = 15000\frac{v}{t}$, vengono immesse un numero sempre crescente di strade, che dopo tale soglia va a decrescere. Quando δ è una soglia piccola, la rimozione naturale elimina solo le polilinee con flusso di traffico molto basso, lasciando le altre in G'. Si creano, così, molti frammenti vicini tra loro, che per essere connessi hanno bisogno del reinserimento in G' di poche polilinee di G. Aumentando la soglia di rimozione, con il conseguente decremento delle polilinee di G', diminuisce anche il numero di frammenti e aumenta la distanza tra loro, che giustifica l'aumento delle polilinee inserite per completare la connessione. Per δ grande, invece, l'algoritmo riesce a connettere dei frammenti di grafo con un numero sempre minore di polilinee. Anche ciò trova una ragionevole giustificazione, in quanto, venendo eliminate quasi tutte le polilinee, restano solo alcuni tratti nevralgici di Roma vicini tra loro con elevate concentrazioni di traffico, per cui la loro connessione richiede un numero esiguo di polilinee da immettere nel grafo.

Partendo dall'idea che la semplificazione a posteriori per flussi massimi connette i frammenti sfruttando le strade già presenti in G', si pensa alla possibilità di creare una gerarchia di grafi G_δ, dove $\forall \delta \epsilon > 0, G_\delta \supseteq G_{\delta+\epsilon}$. Ossia gli elementi di G_δ contengono quelli di $G_{\delta+\epsilon}$ più altri rimanenti dalla rimozione naturale e dalla fase di connessione dei frammenti. Si implementa un algoritmo che, fissato un δ grande, reitera la semplificazione a posteriori al decrescere di δ, mantenendo le polilinee dell'iterazione precedente. Si esamina il grafico 3.11 dove con la linea rossa è rappresentata la percentuale di metri rimasti dopo la rimozione naturale, con la verde i metri aggiunti, con la blu i metri totali del grafo semplificato e con il viola i metri totali ottenuti da una singola semplificazione del grafo originario.

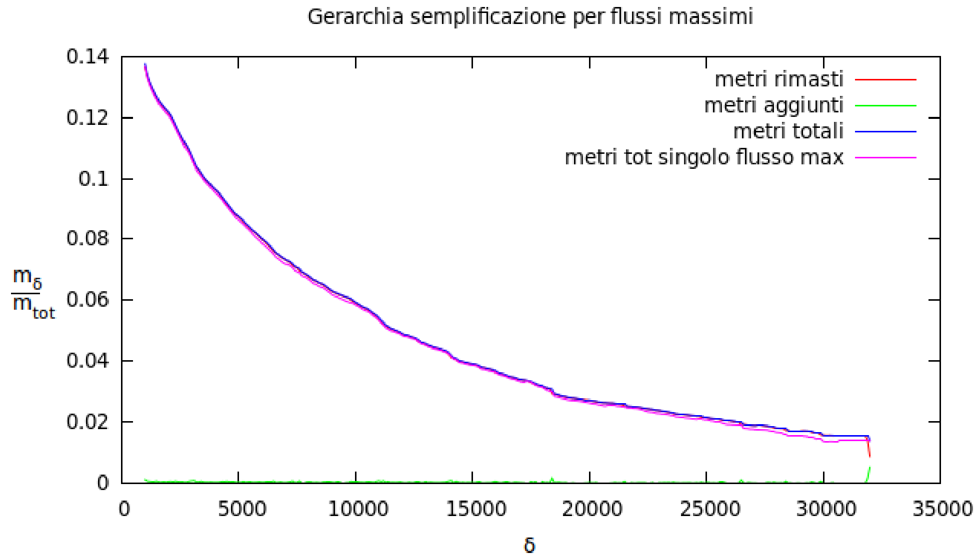

Figura 3.11: Andamento della gerarchia di grafi con la tecnica di rimozione a posteriori per flussi massimi

Si osserva che la rimozione a posteriori si comporta alla stregua di una gerarchia. L'opera di mantenimento delle polilinee espletata in ordine gerarchico dai grafi generati in precedenza, fa si che se ne inseriscano un numero esiguo per connettere i nuovi frammenti creati nella generazione corrente. Da ciò risulta che, per ogni δ fissato, la quantità di metri di strade presenti nel grafo è di poco superiore, rispetto a quella ottenuta dalla corrispondente semplificazione a posteriori del grafo originario.

Si sposta l'attenzione sulla parte centrale della rete urbana, interna al raccordo anulare, dove le strade sono rappresentate con colorazioni diverse in base alla normale velocità del flusso veicolare in assenza di traffico. Le polilinee con velocità al di sotto dei $30\frac{km}{h}$ sono colorante in rosso, quelle comprese tra 30 e $60\frac{km}{h}$ in giallo, quelle tra 60 e $90\frac{km}{h}$ in verde e quelle superiori a $90\frac{km}{h}$ in blu. Applicando la semplificazione a posteriori gerarchica per flussi massimi, con δ pari rispettivamente a $8000\frac{v}{t}$, $4000\frac{v}{t}$ e $1000\frac{v}{t}$, si ottiene un grafo colorato, dove le tonalità accesse rappresentano le strade realizzate in fase di rimozione naturale e le spente quelle inserite in fase di connessione dei frammenti. Dalle immagini si osserva che le polilinee inserite in fase di connessione sono un numero esiguo, mentre la maggior parte sono quelle restanti, che derivano da una rimozione naturale corrente o precedente. Dato che la generazione corrente eredita tutte le strade delle generazioni

precedenti, capita che molti frammenti generati a seguito della rimozione naturale siano già connessi al frammento padre. Si osserva, inoltre, un legame tra il flusso di traffico e la velocità con strada libera. Infatti, per $\delta = 8000\frac{v}{t}$, si nota che restano le strade con un'elevata velocità (azzurre e verdi) e poche con bassa velocità. Quest'ultime rappresentano un intenso flusso veicolare lento, dovuto probabilmente a ingorghi. Ponendo il $\delta = 4000\frac{v}{t}$, si assiste a un aumento delle strade a bassa velocità percorse tuttavia da meno veicoli. Per $\delta = 1000\frac{v}{t}$, si ottengono tutte le strade che hanno un flusso veicolare abbastanza consistente, utile per uno degli obiettivi della tesi.

Figura 3.12: Grafo gerarchico G_δ con $\delta = 8000\frac{v}{t}$

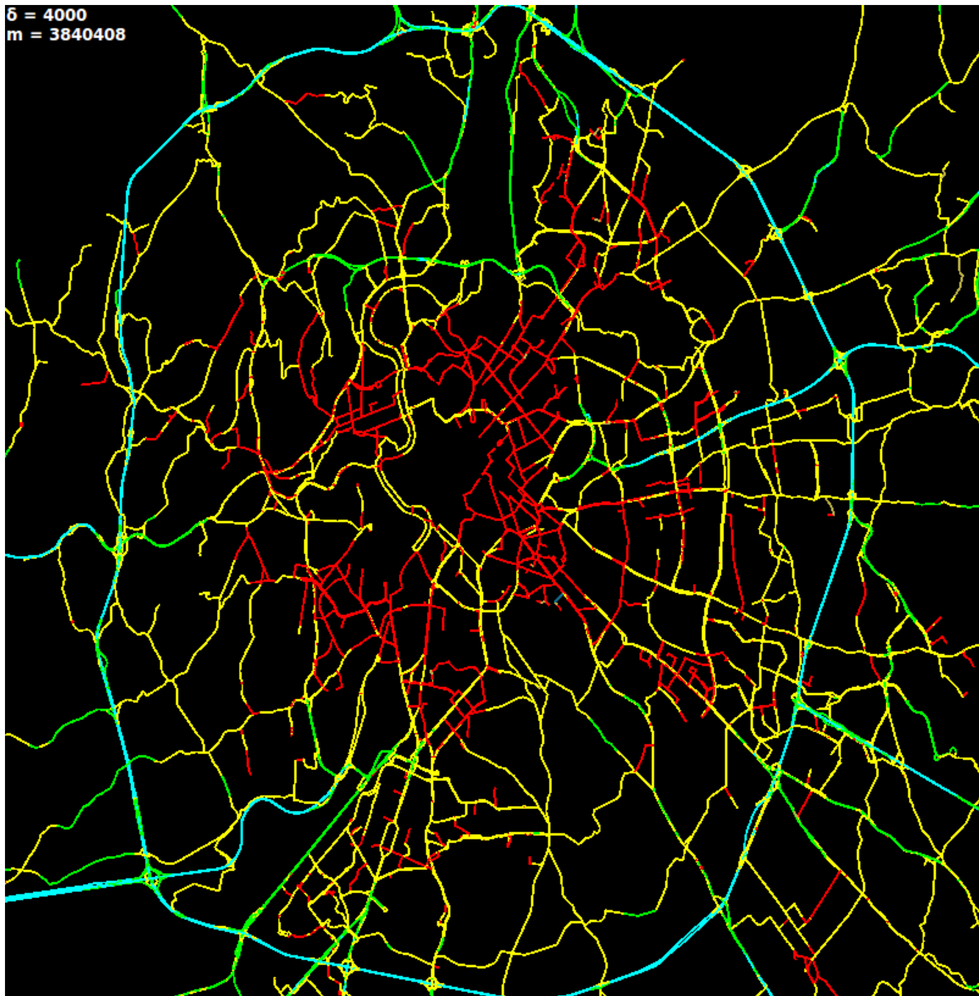

Figura 3.13: Grafo gerarchico G_δ con $\delta = 4000\frac{v}{t}$

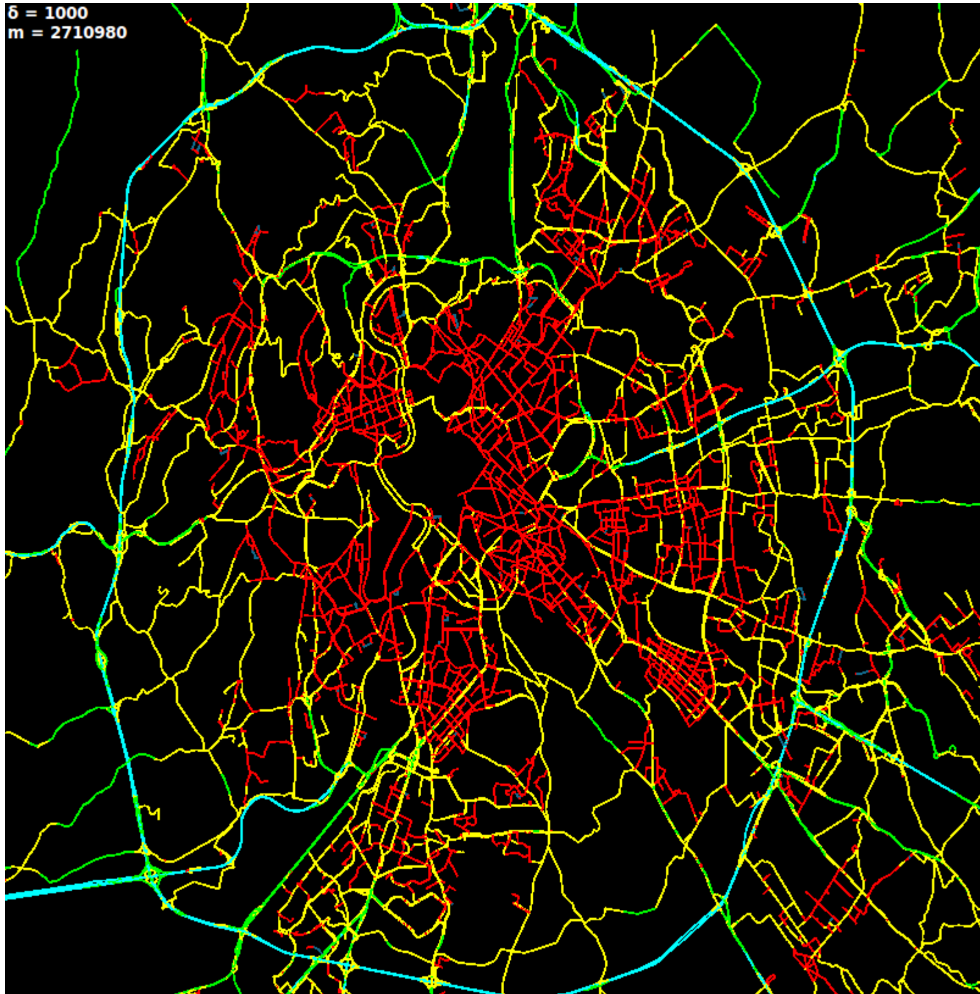

Figura 3.14: Grafo gerarchico G_δ con $\delta = 1000\frac{v}{t}$

Si analizza il comportamento del flusso veicolare della rete stradale, a seguito del procedimento di semplificazione a posteriori per flussi massimi, rilevato nei diversi livelli gerarchici di grafi. Sia G_δ il grafo gerarchico semplificato al livello δ, composto da $\{P_0, P_1, \ldots, P_{n-1}\}$ polilinee, tale che ogni $P_i \in \{P_0, P_1, \ldots, P_{n-1}\}$ ha associati il flusso veicolare ϕ_{P_i} e la lunghezza in metri L_{P_i}. Allora si può definire

$$\phi_\delta L_\delta = \sum_{i \in \{0,\ldots,n-1\}} \phi_{Pi} L_{P_i}$$

che rappresenta l'"estensione del flusso veicolare" presente in G_δ. Si calcola $\phi_G L_G$, che rappresenta l'"estensione del flusso veicolare" nel grafo di

origine e si compara con quello dei livelli gerarchici per $\delta \in [1000; 32000]$.

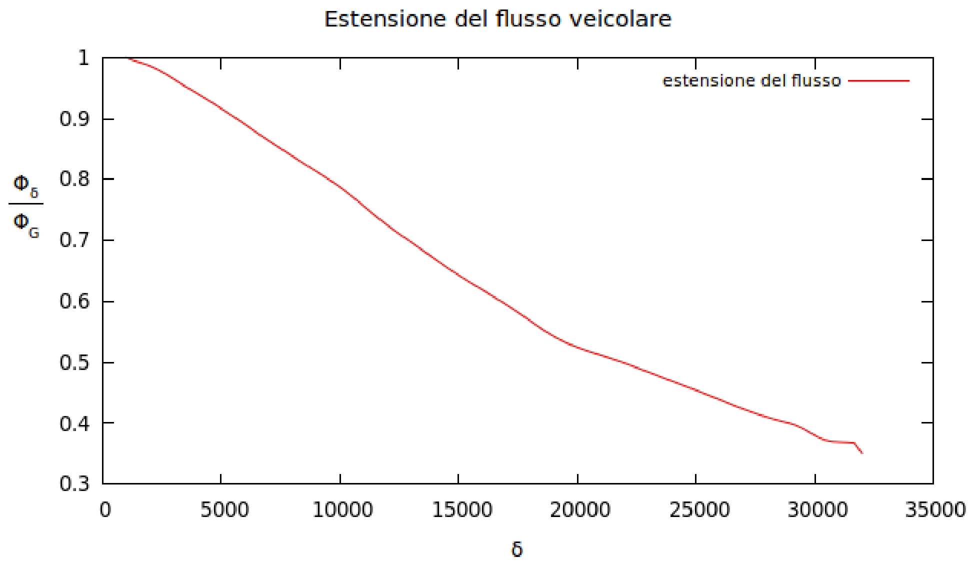

Figura 3.15: Percentuale dell'estensione di flusso rimanente di G_δ

Dal grafico si evince che per $\delta = 1000\frac{v}{t}$ è presente la totalità del flusso veicolare, che percorre il grafo semplificato G' con appena il 12% degli archi di G. Al crescere di δ si osserva una decrescita lineare che arriva al 35% del flusso veicolare, per $\delta = 32000\frac{v}{t}$, con lo $0,8\%$ degli archi di G.

Conclusione e sviluppi futuri

Lo scopo di questa tesi è stato quello di realizzare un sistema informatico, che ha consentito di ridefinire il grafo della rete stradale di Roma, mediante l'automatizzazione del processo di inserimento delle polilinee mancanti e la ricerca delle possibili soluzioni di semplificazione di esso in base al flusso di traffico considerato. È stato creato un editor che ha agevolato notevolmente le operazioni di modifica della rete stradale. L'editor è stato dotato di una componente principale per visualizzare a video l'immagine della sovrapposizione delle reti stradale-veicolare e di una serie di servizi idonei a effettuare operazioni interattive guidate di inserimento, spostamento, divisione e eliminazione degli elementi della rete stradale. Il risultato anche se soddisfacente non era però ottimale, in quanto non prevedeva la ricostruzione automatica delle polilinee mancanti e quindi non escludeva la possibilità di errore, derivante da un eventuale intervento manuale. Il problema è stato adeguatamente superato con l'implementazione di una serie di algoritmi, che hanno permesso con buoni risultati l'inserimento automatico delle strade mancanti nel grafo.

In questa fase è emerso che le strade senza veicoli associati, cioè prive di un flusso veicolare consistente, non fornivano informazioni utili ai fini della correzione del grafo. Da qui è nata l'idea di rimuovere dal grafo gli elementi superflui per constatarne il comportamento e gli effetti. A tal fine in via sperimentale sono state elaborate tre possibili soluzioni, che hanno consentito, con risultati diversi, di semplificare il grafo G in un sottografo $G' \subseteq G$ connesso in un'unica classe di equivalenza. Dai numerosi test effettuati sul grafo di Roma è risultata più performante una tecnica ibrida di semplificazione, che combina la rimozione *a posteriori per flussi massimi* con *l'eliminazione dei nodi di rango* 2. Il grafo così semplificato ha dato risultati soddisfacenti. Ha fornito una migliore visione dell'andamento del flusso veicolare, che ha permesso di focalizzare le principali strade ad alta densità di traffico e di individuare le zone maggiormente congestionate.

La semplificazione *a posteriori* ha risposto secondo le aspettative, in quanto ha consentito di generare una gerarchia crescente di grafi che, al diminuire

della soglia di rimozione δ, ha fornito informazioni sempre più dettagliate della rete stradale

Il progetto potrebbe essere ampliato per la realizzazione di cartografie aggiornate e l'implementazione di un database del traffico della rete stradale, costantemente aggiornato dal flusso continuo di dati GPS. Sarebbe interessante approfondire il lavoro sin qui espletato con la ricerca di nuove soluzioni, che permetterebbero di rilevare gli incroci stradali di un flusso veicolare anomalo, per ricostruire automaticamente porzioni di rete stradale formate da più polilinee connesse tra loro. La combinazione delle tecniche di ricostruzione automatica delle polilinee e di rilevamento degli incroci stradali, potrebbe aprire nuovi scenari di ricerca dei metodi, che permetterebbero di costruire interamente un grafo stradale partendo dalla sola informazione veicolare GPS. Sarebbe altrettanto interessante poter parametrizzare le soluzioni trovate nel campo della semplificazione stradale, in modo da ottenere un grafo ad hoc contenente le polilinee che rispettano le proprietà definite.

Appendice A

Regressione lineare

La regressione lineare è un metodo per trovare l'equazione lineare del tipo $y = \beta x + \alpha$ che meglio approssima un insieme di punti dato $(x_1, y_1), \ldots, (x_n, y_n)$ nell'asse cartesiano[14]. La retta di regressione lineare è calcolata utilizzando il metodo dei minimi quadrati ed è data dal valore minimo della somma dei quadrati delle differenze tra le coordinate dei punti delle y e le coordinate dei corrispondenti punti della retta di regressione lineare. β prende il nome di coefficente di regressione e misura la variazione della y conseguente ad una variazione unitaria della x, mentre α prende il nome di 'intercetta' che corrisponde al valore della retta sull'asse delle y quando x vale zero. Dal punto di vista geometrico, se si considera un asse cartesiano (x, y), β è il coefficente angolare della retta e α è l'ordinata per $x = 0$.

Di seguito è riportata la formula per calcolare la retta di regressione lineare su un insieme di n punti su un piano cartesiano:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}, \ \overline{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

$$\beta = \frac{\sum_{i=1}^{n}((x_i - \overline{x})(y_i - \overline{y}))}{\sum_{i=0}^{n}(x_i - \overline{x})^2}$$

$$\alpha = \overline{y} - \overline{x}\beta$$

Appendice B

IsGeom

IsGeom è una libreria, scritta in c, implementata appositamente per risolvere alcuni problemi geometrici derivati dallo svolgimento di questa tesi. Mette a disposizione delle funzioni che permettono di creare e gestire punti, segmenti e rette. Le strutture dati definiscono:

- un punto di coordinate cartesiane (x, y);

```
typedef struct{

        double x;
        double y;

}point_2d;
```

- un arco di nodi (u, v) con coordinate cartesiane (x_0, y_0, x_1, y_1);

```
typedef struct{

        point_2d xy0;
        point_2d xy1;

}segment_2d;
```

- l'equazione di una retta in forma implicita ($ax + by + c = 0$);

```
typedef struct{

        double x;
        double y;
        double c;

}implicit_2d;
```

- l'equazione di una retta in forma esplicita ($y = mx + q$ o $x = my + q$).

```
typedef struct{

        double m;
        double q;

}explicit_2d;
```

Dei numerosi metodi implementati durante lo svolgimento della tesi, si riportano solo quelli effettivamente utilizzati:

- point_2d *initPoint*(double x, double y), crea un punto di coordinate cartesiane (x, y);
- segment_2d *initSegment*(double x_0, double y_0, double x_1, double y_1), crea un arco di coordinate cartesiane (x_0, y_0, x_1, y_1);
- implicit_2d *segmentToImplicit*(int * *cod*, segment_2d s), restituisce l'equazione in forma parametrica passante per il segmento s;
- implicit_2d *explicitToImplicit_y*(explicit_2d r), restituisce l'equazione della retta r, espressa nella forma esplicita $y = mx + q$, in forma implicita;
- implicit_2d *explicitToImplicit_x*(explicit_2d r), restituisce l'equazione della retta r, espressa nella forma esplicita $x = my + q$, in forma implicita;
- explicit_2d *implicitToExplicit_y*(implicit_2d r), restituisce l'equazione della retta r, espressa in forma implicita, nella forma esplicita $y = mx + q$;

- explicit_2d $implicitToExplicit_x$(implicit_2d r), restituisce l'equazione della retta r, espressa in forma implicita, nella forma esplicita $x = my + q$;

- double $segmentLen$(segment_2d s), restituisce la lunghezza di un segmento;

- point_2d $segmentIntersect$(int * cod, implicit_2d $r1$, implicit_2d $r2$), restituisce il punto di intersezione tra $r1$ e $r2$;

- int $segmentAngle$(segment_2d s), restituisce l'angolo in gradi di un segmento rispetto le ascisse;

- point_2d $awayPoint$(int $coeff$, point_2d p, segment_2d s), restituisce le coordinate cartesiane del punto spostato in base all'angolo di s per un coefficiente dato;

- point_2d $translatePoint$(int * pos, double * $teta_point$, double * $teta_seg$, point_2d $point$, segment_2d s), restituisce le coordinate cartesiane del punto traslato rispetto il segmento s;

- explicit_2d $linearRegression_y$(int * cod, int n, point_2d * p), restituisce l'equazione, in forma esplicita ($y = mx + q$), della retta di regressione lineare calcolata sulla nube di punti p;

- explicit_2d $linearRegression_x$(int * cod, int n, point_2d * p), restituisce l'equazione, in forma esplicita ($x = my + q$), della retta di regressione lineare calcolata sulla nube di punti p;

- int $radiantToGrade$(double $angle$), restituisce il valore dell'angolo $angle$ espresso in gradi;

- double $gradeToRadiant$(int $angle$), restituisce il valore dell'angolo $angle$ espresso in radianti;

- int $normalizeAngle$(int $sign$, int $angle$), restituisce il valore di $angle$, espresso in gradi, normalizzato. Se $sign = MODULE$ è $0° \leq angle \leq 90°$; altrimenti se $sign = NOMODULE$ è $\pm 0° \leq angle \leq \pm 90°$

- void $getPointAttack$(int * s_a, int * s_b, segment_2d $s1$, segment_2d $s2$), restituisce per i segmenti $s1$ e $s2$ i loro rispettivi nodi di congiunzione.

Appendice C

Errore posizionale dei veicoli

Dati un arco a, n veicoli $v_1, \ldots, v_n$, distribuiti sul piano con una funzione gaussiana[18] $f(x)$ il cui scarto quadratico medio è $\vartheta = 8m$ (vedi capitolo 1.4), si calcola la probabilità che un v_i è collocato ad una distanza massima di $40m$ da a. Si considera come sistema di riferimento il piano cartesiano, dove l'ordinata di origine rappresenta a e l'ascissa la distanza da a.

L'integrale di $f(x)$, con estremi $(-\infty, +\infty)$ corrisponde alla funzione di ripartizione $F(x)$ della distribuzione gaussiana, che considera tutti i veicoli di qualsiasi distanza da a con probabilità pari a 1.

$$\int_{-\infty}^{+\infty} \frac{1}{\vartheta\sqrt{2\pi}} e^{-\frac{x^2}{2\vartheta^2}}\, dx = 1$$

Eseguendo $\frac{\partial^2}{\partial x^2} f(x)$ si ricava la variazione della pendenza di $f(x)$, corrispondente all'inizio della sua coda, pari a ϑ. Limitando gli estremi dell'integrale di $f(x)$ con $(-\vartheta, +\vartheta)$ si ottiene la probabilità, pari a 0.667, che un v_i è collocato nella relativa area descritta.

$$\int_{-\vartheta}^{+\vartheta} \frac{1}{\vartheta\sqrt{2\pi}} e^{-\frac{x^2}{2\vartheta^2}}\, dx \approx 0.667$$

Analogamente, ponendo $\delta = 5\vartheta$, si ottiene la probabilità che un veicolo si trova nell'area, delimitata dall'integrale di $f(x)$, con una distanza massima pari a $40m$.

$$\int_{-\delta}^{+\delta} \frac{1}{\vartheta\sqrt{2\pi}} e^{-\frac{x^2}{2\vartheta^2}}\, dx \approx 0.998$$

La probabilità che un veicolo si trova fuori dall'area di $f(x)$, delimitata da $(-\delta, +\delta)$, è quasi nulla e per questo motivo si ricerca la sua strada attinente entro e non oltre una circonferenza di raggio pari a $40m$ dalla sua posizione sul piano.

Bibliografia

[1] Bertossi A., *Algoritmi e Strutture di Dati*, UTET, 2004

[2] Biagini F., Campanino M., *Elementi di Probabilità e Statistica*, Springer, 2006

[3] Baldi P., *Calcolo delle probabilità e Statistica*, McGraw-Hill, 1998

[4] Bramanti, Pagani, Salsa, *Matematica Calcolo Infinitesimale e Algebra Lineare*, Zanichelli, 2004

[5] Resnick Robert, Halliday David, Krane Kenneth, *Fisica1*, Ambrosiana, 2003

[6] Kernighan Brian, Ritchie Dennis, *Il linguaggio C*, Pearson, 2006

[7] Woo M., Neider J., Davis T, Shreiner D., *OpenGL Programming Guide: The Official Guide to Learning OpenGL*, Addison-Wesley, 1999

[8] Campbell J., Introduzione alla Cartografia, Zanichelli, 1989

[9] Tinaglia Calogero, *Matematica Discreta per Informatici*, Pitagora, 2003

[10] Capper Derek, *Introducing C++*, Springer, 2001

[11] Edwards A., *An Introduction to Linear Regression and Correlation*, Freeman, 1976

[12] Intelisano M., *Correzione Mappe Stradali con Dati Veicolari GPS*, Tesi di laurea in Informatica - Università degli Studi di Bologna, 2008

[13] Peli G., *Analisi dei Dati Provenienti da GPS su un Network Stradale*, Tesi di laurea in Fisica - Università degli Studi di Bologna, 2006

[14] Wikipedia, *Linear Regression*, `http://en.wikipedia.org/wiki/Linear_regression`

[15] Wikipedia, *Ellissoide*, http://it.wikipedia.org/wiki/Ellissoide

[16] Wikipedia, *WGS-84*, http://it.wikipedia.org/wiki/Ellissoide_di_riferimento

[17] Weisstein E., *Trigonometry*, http://mathworld.wolfram.com/Trigonometry.html

[18] Weisstein E., *Gaussian Function*, http://mathworld.wolfram.com/GaussianFunction.html

[19] Weisstein E., *Exponential Distribution*, http://mathworld.wolfram.com/ExponentialDistribution.html

[20] Weisstein E., *Graph*, http://mathworld.wolfram.com/Graph.html