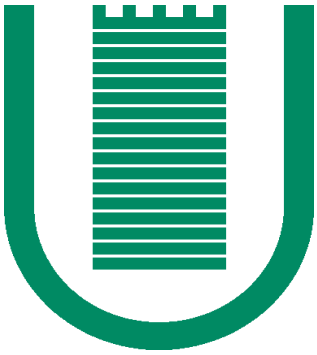
UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA



FACOLTÀ DI SCIENZE MM.FF.NN
Corso di Laurea Triennale in Informatica

Geolocalizzazione di tracce GPS

suddivisione dello spazio

Relatore
Prof. Gianluca Rossi

Candidato
Angelo Moroni

A/A 2012/2013

"Quand'è che il futuro è passato da essere una promessa a essere una minaccia?"

C. Palahniuk

Indice

1. Geolocalizzazione di una Traccia GPS

Si immagini di avere a disposizione una traccia GPS relativa ad un viaggio e di voler estrapolare da questa informazioni sulle strade utilizzate: per esempio, quanti chilometri sono stati percorsi su strade a pagamento, su tangenziali, su strade secondarie o di campagna.

Quindi, data una traccia e data una mappa che contiene informazioni sulle strade affronteremo il problema di geolocalizzare tracce GPS, ovvero stabilire quali strade sulla mappa corrispondono alla traccia GPS.

Nostro obiettivo è quindi creare un sistema acerbo (al quale si deve ovviamente lavorare per futuri miglioramenti) che permetta questo.

Prima di discutere formalmente il problema è necessario definire il significato di "geolocalizzazione"; per *Geolocalizzazione* si intende quel processo, il quale permette di identificare le coordinate geografiche di un qualsiasi oggetto, e quindi di poterlo rappresentare su una mappa.

Il problema così posto è molto vago e quindi cerchiamo di formalizzarlo.

Sia *tr* una traccia , ovvero una sequenza di punti GPS, $tr = \{p_1,p_2....p_n\}$ con p_i punti bidimensionali, che rappresentano le coordinate geografiche.

La mappa è rappresentata attraverso un grafo bidimensionale G=(V,E,W) non diretto e pesato, in modo tale che per ogni nodo della mappa esiste un v_i appartenente a V avente coordinate geografiche e per ogni coppia di vertici esiste un arco *e* appartenente ad E se e solo se questi due vertici sono adiacenti su una determinata strada della mappa. Ad ogni arco *e* è associato un peso che corrisponde alla distanza euclidea tra i due nodi.

Il nostro scopo dunque è trovare il cammino su G che corrisponde a *tr*.

Grazie alla diffusione di dispositivi GPS e di smartphone sempre più all'avanguardia, gli studi riguardanti le mappe e le tracce GPS sono aumentati notevolmente e l'uso che se ne fa di queste informazioni è veramente vario. Volendo citare qualche

esempio, si va dal più comune e diffuso Navigatore Satellitare, fino alle più recenti applicazioni gratuite per smartphone, le quali offrono anche la possibilità di conoscenza e condivisione della propria posizione GPS, da social network, come *4square* , a servizi come *Local* di Google. Si parla in effetti, di numerosissime applicazioni e tanto è diffuso l'utilizzo di tali informazioni che è sorto anche il bisogno di creare mappe accessibili da tutti, come *OpenStreetMap*.

Una utilizzazione pratica delle tracce GPS, per cui si investe notevolmente in programmi di ricerca, è relativa, per esempio, allo studio di fenomeni quali gli uragani: i meteorologi cercano di prevedere nella maniera più attendibile, la località che sarà interessata da tale fenomeno, con lo scopo di limitarne i danni, sia a cose che a persone.

Altra applicazione delle tracce GPS potrebbe essere nel campo turistico/commerciale: si potrebbero individuare gli itinerari più trafficati e popolari, e su questi, studiare un piano di realizzazione e diffusione di attività culturali ed economiche, in modo da ottenere ottimi risultati, grazie ad una buona progettazione ed al giusto indirizzo degli investimenti.

L'analisi di quest'ultima possibile applicazione è il tema degli studi presenti in [7] e [8], dove, inoltre, si riportano le modalità di creazione di mappe a partire da tracce GPS, mappe che saranno base di ulteriori studi. Problema principale quindi è proprio la costrizione di queste, poiché una volta che si ha a disposizione una mappa, risulta facile effettuare esperimenti.

Il nostro progetto potrebbe avere applicazioni del genere, poiché si preoccupa di risolvere il problema di geolocalizzazione delle tracce utilizzando mappe esistenti. Tuttavia in letteratura si evidenzia che oggi, è sempre più di moda costruire le mappe a partire da tracce GPS conosciute, piuttosto che utilizzare mappe già esistenti. Ciò è dipeso dal fatto che le uniche mappe (*OpenStreetMap*) a cui si ha un libero e completo accesso, sono ancora poco accurate e i loro aggiornamenti necessitano di tempi di lavoro notevoli.

Per risolvere tale problema, sono sempre di più gli studi relativi alla costruzione di

mappe con meno lavoro possibile da parte delle persone. In [9] l'obbiettivo illustrato è proprio questo, e i risultati conseguiti sono eccezionali: si possono disegnare mappe utilizzando le tracce ottenute da normali automobili, munite di un rilevatore GPS (navigatore proprio della vettura o smartphone di chi è a bordo), che giornalmente si muovo in città.

In generale quasi tutti i progetti citati sopra utilizzano un proprio metodo di clustering delle tracce: in [7] in particolare si usa il RICK (Route Inference framework based on Collective Knowledge), un metodo che calcola l'inferenza delle tracce basandosi sulla conoscenza collettiva. Invece in [10] e [11] si studia in particolare il problema del clustering.

L'ideale sarebbe utilizzare questi metodi per arricchire le mappe a disposizione della collettività.

Noi, invece, abbiamo deciso di geolocalizzare le tracce GPS su una mappa già tracciata. Le mappe da noi utilizzate sono fornite da *OpenStreetMap* (OSM). *OpenStreetMap*, citando la definizione presente sul sito ufficiale, “è un progetto che crea e fornisce dati cartografici, come ad esempio mappe stradali, liberi e gratuiti a chiunque ne abbia bisogno”.

Abbiamo già accennato a questo tipo di mappe e alla loro possibilità di poter accedere ai loro dati in maniera completa, e proprio ciò ha fatto ricadere la nostra scelta su queste, piuttosto che sulle quelle offerte, per esempio, da Google o Bing, i quali, pur mettendo a disposizione mappe molto accurate, ne limitano fortemente l'accesso.

È vero che le mappe di *OpenStreetMap*, sono al momento poco accurate e necessitano di ulteriore lavoro (come sopra accennato), ma presentano, in molte zone, una qualità di dati davvero elevata, qualità che continua a crescere giorno dopo giorno.

2. Algoritmo di Geolocalizzazione

Sia G=(V,E,W) il grafo che rappresenta la mappa, così come è stato presentato nel primo capitolo.

Sia $tr = \{p_1, p_2 ... p_n\}$ una traccia di punti GPS registrata attraverso un dispositivo che può essere o un dispositivo ad hoc o un semplice smartphone, in altre parole una sequenza di punti p_i aventi coordinate geografiche (latitudine, longitudine).

Sia una *way* una seguenza di strade, sentieri, piste ciclabili di una determinata mappa. Tale *way* è, inoltre, rappresentata per mezzo di un cammino sul grafo G.

Il nostro obbiettivo è quello di trovare una *way,* quindi un cammino sul grafo G, che approssima al meglio la traccia *tr*.

Il problema così posto nasconde al suo interno due sotto-problemi: il primo riguarda l'approssimazione dei punti della traccia ai punti reali della mappa; il secondo riguarda l'individuazione della *way* sulla mappa partendo dai punti trovati. Noi, nei nostri studi, ci siamo preoccupati in particolare del primo sotto-problema poiché lo abbiamo ritenuto più cruciale rispetto al secondo. In effetti, quest'ultimo, una volta a disposizione il grafo che rappresenta la mappa, è risolvibile implementando opportuni algoritmi di routing. Altrimenti esistono software che permettono di calcolare la *way* in base ad un traccia che viene passata in input. In tal caso, passeremo la traccia approssimata. *QlandkarteGT*[12] è un sotware Open Source GIS che presenta la possibilità di creare *way* partendo da una traccia GPS.

Prima di cominciare ad analizzare il problema, utilizzeremo punto e nodo di una mappa o di una traccia come sinonimi. Invece, se ci riferiamo a dei nodi del grafo utilizzeremo anche il termine vertice.

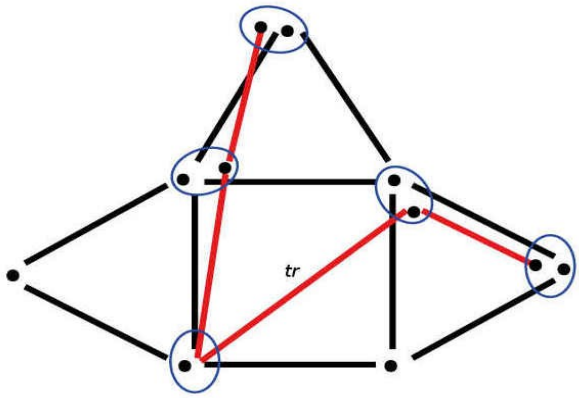
2.1 Approssimazione dei Punti

Per approssimazione di punti si intende: data una traccia $tr = \{p_1, p_2 ... p_n\}$ si deve trovare una sequenza $tm \subset V$, $tm = \{ v_1, v_2 ... v_n\}$ di punti sulla mappa che approssimano al meglio i punti della traccia.

L'approssimazione dei punti si rende necessaria per via degli errori che possono sorgere dalla rilevazione dei punti GPS. Infatti, i punti della traccia possono risultare esterni rispetto al percorso che realmente è stato compiuto. Tali errori di rilevazione, tuttavia, rientrano in range limitato e per questo ci pareva logico approssimare i punti della traccia ai punti più vicini sulla mappa. Quindi sia *dist(p,q)* la distanza euclidea tra due punti *p* e *q*, vogliamo approssimare i punti p_i della traccia in modo tale che p_i è approssimato con il punto v_i della mappa se e solo se $dist(p_i, v_i) < dist(p_i, v_k)$ con $k \neq i$.

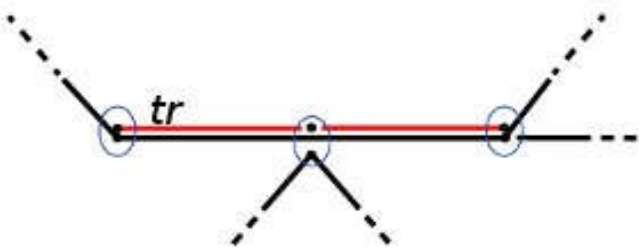
Oltre a questi problemi, l'approssimazione risulta necessaria anche per motivi applicativi. Per trovare i percorsi approssimati, infatti, bisogna utilizzare informazioni contenute nella mappa, ma non ottenibili dalla traccia.

Dunque, cerchiamo un metodo che ci faccia associare ogni punto della traccia al punto più vicino sulla mappa.

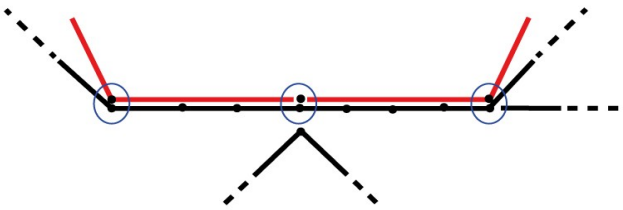


Abbiamo considerato la mappa come un grafo bidimensionale, nel quale ogni vertice ha le proprie coordinate. Con lo stesso principio, la traccia è stata considerata come una linea spezzata nella quale ogni punto di variazione fosse un punto derivato dalla rilevazione GPS e tale punto può o meno combaciare con un vertice del grafo. Inoltre, abbiamo preso in considerazione esclusivamente la distanza euclidea tra due punti, non considerando altri aspetti che potevano influenzare l'associazione. Quest'ultima semplificazione potrebbe portare problemi. Per esempio, immaginiamo

il caso in cui la nostra traccia contiene un punto p_i in mezzo ad un lungo rettilineo. Questo rettilineo sulla mappa sarà definito da due punti v_j e v_k che si troveranno agli estremi del rettilineo e molto probabilmente non avrà altri punti al suo interno e l'associazione avverrebbe in modo errato. Infatti come mostra la figura seguente possiamo notare come l'associazione basata esclusivamente sulla distanza euclidea potrebbe fallire.



Questo problema viene ovviato aggiungendo punti v_i fittizi tra due punti v_j e v_k, di una stessa *way*, se se $dist(v_j, v_k) > d$,cioé se la distanza tra questi due punti supera una determinata distanza *d*. In questo modo, l'associazione può basarsi su più punti ed risultare più accurata.



Calcolare, comunque, per ogni punto della traccia le distanze rispetto a tutti i punti della mappa risulta eccessivo e per questo sono stati utilizzati i digrammi di Voronoi e la struttura di ricerca del Point Location, con i quali abbiamo potuto compiere

l'approssimazione in tempi eccezionali.

I diagrammi di Voronoi, sono una versatile struttura dati geometrica, che rappresentano un partizionamento dello spazio in tante regioni. Ogni regione è costruita attorno ad un punto fisso *p* in modo tale che ogni punto all'interno della regione sia più vicino a *p* che ad ogni altro punto fisso dello spazio.

La struttura del Point Location, invece, permette la ricerca dei punti rispetto ad un partizionamento dello spazio. In altre parole, data una suddivisione dello spazio questa struttura determina in quale regione un determinato punto *q* giace.

Appare subito chiaro l'utilizzo che ne abbiamo fatto di queste due strutture: abbiamo generato il diagramma di Voronoi sulla mappa e per ogni punto della traccia abbiamo determinato, attraverso la struttura di ricerca del Point Location, costruita sul partizionamento di Voronoi, la regione nella quale giaceva. In questo modo potevamo approssimare i punti della traccia ai punti reali della mappa.

2.2 Estrapolare informazioni

Come già detto, individuare la *way* che descrive al meglio la traccia *tr* non è interesse di questa tesi. Comunque sia, può interessare, per vari motivi, estrapolare informazioni sulle strade che vengono toccate da *tr*. Da una strada *s* possiamo estrapolare un gran numero di informazioni come ad esempio la natura stessa di *s* o se è percorribile da pedoni o ciclisti, o se si stratta di una strada con pedaggio a pagamento o meno. Possiamo, addirittura, estrapolare il numero delle corsie o carreggiate di cui *s* è composta. *OpenStreetMap* fornisce un gran numero di informazioni estraibili e a noi interessa controllare se tra tutte le strade *s* toccate dalla traccia *tr,* ne esiste almeno una in cui il pedaggio è a pagamento.

Un'applicazione del genere è molto semplice e contrariamente quanto si possa pensare essa non richiede di calcolare la *way* intera. In effetti, possiamo facilmente risalire alle informazioni che ci interessano semplicemente utilizzando il grafo costruito nella prima parte del problema. Ogni elemento della lista di adiacenza, infatti, mantiene il riferimento alla strada dalla quale è stato creato. In questo modo si

può risalire a quest'ultima e alle sue informazioni molto facilmente.

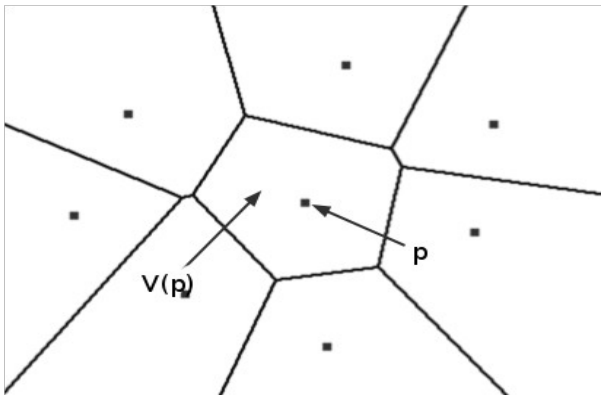
In pratica, in fase di costruzione del grafo costruiamo, anche, un insieme S di strade che hanno il pedaggio a pagamento e una volta generata *tm*, per ogni $v \in tm$ controlliamo se esso giace almeno su una strada $s \in S$.

3. Diagrammi di Voronoi

Il problema dell'agenzia pubblicitaria

Immaginiamo che un'agenzia pubblicitaria, con tanti sedi sparse per il territorio, debba diffondere una nuova pubblicità. L'agenzia deve fare in modo che ogni sede lavori il più possibile, affinché questa pubblicità copri più territorio possibile. Inoltre, per risparmiare, vuole spendere il meno possibile e sa che ciò può avvenire se ogni sede riesce a coprire più spazio possibile. In altre parole, il territorio deve essere diviso in regioni, una per ogni sede, in modo tale che ogni luogo della regione sia raggiungibile nel minor tempo possibile dai lavoratori della sede assegnata alla regione. Quindi, ogni luogo di una regione è più vicino alla sede associata alla regione che ad ogni altra sede.

Il problema posto dall'agenzia pubblicitaria però, è un problema geometrico già noto e molto studiato.
Formalmente, dato un insieme S di n punti, detti *siti*, si vuole partizionare lo spazio in *n* regioni in modo tale che qualsiasi punto *q*, non appartenente all'insieme S, cade in una regione se e solo se il sito associato alla regione è il più vicino a *q* rispetto ad ogni altro sito. Per ogni sito *p*, indichiamo *V(p)* la regione associata al sito *p*. L'insieme di tutte le regioni *V(p)* viene detto diagramma di Voronoi dei punti S.



Il nome del partizionamento viene da Georgy Voronoy nonostante Peter Gustav

Lejeune-Dirichlet fu il primo a parlarne nel 1850. Infatti un altro modo per chiamare i diagrammi di Voronoi è *Tassellatura di Dirichlet*. Tuttavia anche Cartesio utilizzava diagrammi molto simili quando descriveva la frammentazione dello spazio causata dal posizionamento delle stelle.

I diagrammi di Voronoi sono una struttura geometrica estremamente potente molto utilizzata in vari ambiti. Questa notevole diffusione è dovuta alla natura versatile della struttura.
In epidemiologia sono utilizzati per studiare la diffusione delle malattie, come ad esempio il medico britannico John Snow utilizzò un diagramma di Voronoi nel 1854 per illustrare come la maggioranza delle persone morte nell'epidemia di colera a Soho viveva più vicino ad una delle pompe infette di Broad Street che ad ogni altra pompa d'acqua.
Ad esempio in campo robotico sono utilizzati per il path planning: in pratica l'intero pavimento viene partizionato cosicché si possano localizzare velocemente gli ostacoli. I robot poi utilizzeranno gli archi del diagramma per muoversi nello spazio, così sono sicuri di non urtare nessun ostacolo durante il cammino.
In campo musicale i diagrammi di Voronoi sono stati utilizzati per rappresentare uno spazio artificiale bidimensionale di cui le dimensioni sono il tono e il tempo.
In ecologia vengono utilizzati per studiare la crescita degli alberi o il rischio di incendi.
Le grandi catene commerciali spesso utilizzano i diagrammi di Voronoi per capire meglio dove aprire un nuovo negozio.

3.1 Proprietà base dei Diagrammi di Voronoi

Prima di andare avanti bisogna ricordare che la distanza a cui si farà riferimento è una distanza euclidea. Siano p e q due punti del piano si definisce distanza euclidea:

$$d(p,q) := \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

Sia $P:=\{p_1, p_2, ..., p_n\}$ un insieme di n punti sul piano, detti *siti*. Si definisce un diagramma di Voronoi, *Vor(P)*, una suddivisione in *n* celle, una per ogni sito in *P*, in modo tale che un punto q , $q \in V(p_i)$ se e solo se $dist(q, p_i) < dist(q, p_j), \forall\, p_j \in P, i \neq j$ dove $V(p_i)$ è la cella che corrisponde all'i-esimo punto di S.

Adesso, poniamo attenzione sulla struttura di una cella di Voronoi.
Siano due punti del piano p_i,p_j definiamo *bisecante di* p_i *e* p_j la retta perpendicolare alla retta passante per p_i e p_j e passante per il punto medio del segmento $\overline{p_i p_j}$.
Questa bisecante quindi divide il piano in due semipiani: $h(p_i,p_j)$ conterrà p_i e viceversa, p_j sarà contenuto da $h(p_j,p_i)$. Da notare che, sia *r*, un qualsiasi altro punto

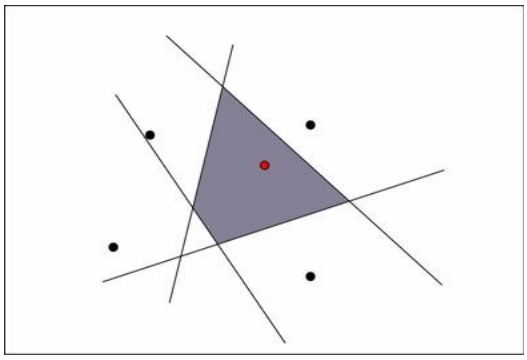
$$r \in h(p_i, p_j) \Leftrightarrow dist(r, p_i) < dist(r, p_j)$$

Da questa definizione deriva un'osservazione.

Osservazione 1

Una cella di Voronoi può essere vista come l'intersezione di n-1 semi piani.

$V(p_i) = \bigcap_{1 \le j \le n, j \neq i} h(p_i,p_j)$

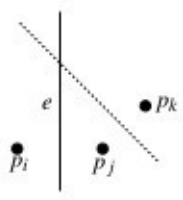


Grazie a tale definizione possiamo dire che ogni cella *V(pi)* del diagramma di Voronoi è convessa delimitata da al massimo *n-1* vertici e al massimo da *n-1* archi.

Inoltre definiamo *Vor(P)* il grafo che ha per vertici e per archi rispettivamente i vertici e gli archi del diagramma di Voronoi.
il diagramma di Voronoi con archi e vertici, cioè il grafo che memorizza il diagramma.
Definire il diagramma di Voronoi come un grafo, però, è un abuso In quanto questo presenta semirette che devono essere rappresentate da archi. Questa caratteristica, ci costringerà a delle riflessioni in fase di implementazione.

Teorema 1 *Se P è composta da n siti collineari allora Vor(P) consiste in n-1 linee parallele. Altrimenti, Vor(P) è un grafo connesso e i suoi archi o sono segmenti o semirette.*

Dimostrazione: La prima parte del teorema è facile da dimostrare, pertanto assumiamo che i siti non sono tutti collineari.
Quindi, deduciamo dall'osservazione 1 che il diagramma di Voronoi è composto da archi che possono essere o segmenti o semirette.
Ora, ci resta di dimostrare che il diagramma non contiene linee rette.



Supponiamo per assurdo che *Vor(p)* contiene un arco *e* che è una retta.
Questo arco, allora, dividerà due celle *Vor(p_i)* e *Vor(p_j).* Per quanto riguarda p_i e p_j possiamo anche pensare che essi siano collineari, ma esisterà, per ipotesi, almeno un punto p_k non collineare ai due.
Dunque la bisecante tra p_k e p_j non sarà parallela all'arco *e* per cui si intersecheranno in un punto.
Ed ecco la contraddizione.
Rimane da dimostrare che *Vor(P)* è connesso. Sempre per assurdo, se così non fosse, ci sarebbe una cella *V(p_i)* che divide in due parti il piano. Affinché sia possibile

questo scenario, la cella $V(p_i)$ deve essere delimitata da due linee parallele, ma questo è impossibile per la prima parte del teorema. □

Definita bene la struttura del diagramma di Voronoi, ora, bisogna discutere della complessità spaziale di *Vor(P)*, ovvero la memoria necessaria per memorizzare una struttura dati che rappresenti il diagramma.
Per quanto riguarda diagrammi su punti in d dimensioni (con $d \geq 2$) è dimostrabile che la complessità è di $\Omega(n^{d-1})$. Questa complessità rappresenta inoltre un lower bound[2].
In questo contesto ci interessano solo spazi a 2 dimensioni ($d = 2$) e il teorema seguente ci dimostra che la complessità è lineare.

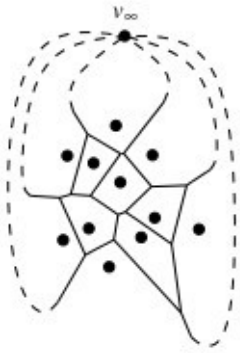
Teorema 2 *Sia n il numero dei siti, per* $n \geqslant 3$ *, il numero di vertici in Vor(P) sono al più 2n-5 e il numero degli archi è al massimo 3n-6.*

Dimostrazione: per la dimostrazione ci serviremo del Teorema di Eulero su grafi planari:

$$v - e + f = 2$$

di cui v è il numero di vertici, e è il numero di archi e f è il numero di facce del grafo.

Come già specificato *Vor(P)* non è veramente un grafo e quindi necessita di modifiche che non saranno influenti sul diagramma stesso.
A *Vor(P)* viene aggiunto un nodo fittizio v_∞ al quale vengono collegati tutte le semirette del diagramma, creando in questo modo un vero grafo planare sul quale vale la formula di Eulero.



Applicando quindi il teorema al nuovo grafo modificato otteniamo

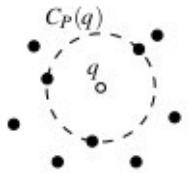
$$(n_v+1) - n_e + n = 2$$

dove n_v,n_e e n sono rispettivamente il numero dei vertici, il numero di archi e il numero delle facce di *Vor(P)*.

Inoltre, nel nuovo grafo, ogni arco ha due vertici, quindi se sommiamo il grado per ogni vertice otteniamo il doppio del numero degli archi. Ogni vertice ha anche un grado minimo pari a 3 e se quindi consideriamo insieme le due relazioni otteniamo che

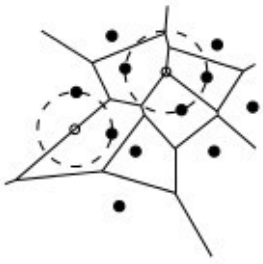
$$2n_e \geq 3\,(n_v + 1)$$

Dunque, infine, se mettiamo in relazione il teorema di Eulero e l'ultima espressione si dimostra il teorema. □



Ora ci rimane da caratterizzare i vertici e gli archi di *Vor(P)*. Innanzitutto definiamo per un punto *q il più largo cerchio vuoto di q che rispetta P*, denotato con $C_P(q)$, ovvero, il più largo cerchio che ha q come centro e che non contiene nessun sito in P.

Teorema 3 *Per un diagramma di Voronoi Vor(P) valgono le seguenti proprietà:*



I. *Un punto q è un vertice di Vor(P) se e solo se $C_P(q)$ contiene tre o più siti di P sulla sua circonferenza.*
II. *La bisecante del segmento p_ip_j definisce un arco di Vor(P) se e solo se esiste un punto q, che giace sulla bisettrice, tale che $C_P(q)$ contiene sia p_i che p_j e nessun altro sito sulla sua circonferenza*

Dimostrazione:

I. Supponiamo che esiste un punto q tale che $C_P(q)$ contiene tre o più siti sulla sua circonferenza. Siano, allora, p_i,p_j e p_k tre di questi punti. Dato che l'interno di $C_p(q)$ è vuoto q deve essere sul bordo di $V(p_i)$,$V(p_j)$ e $V(p_k)$ e quindi q deve essere un vertice di *Vor(P)*. Nell'altro verso, ogni vertice q di *Vor(P)* è incidente ad almeno tre archi e quindi ad almeno tre celle di Voronoi $V(p_i)$,$V(p_j)$ e $V(p_k)$. Pertanto q deve essere equidistante da p_i, p_j e p_k e nessun altro sito. Per questo $C_p(q)$ non contiene altri siti.

II. Sia q un punto che rispetta le proprietà definite sul teorema. Inoltre $C_p(q)$ non contiene nessun sito al suo interno, bensì contiene p_i e p_j sulla circonferenza e per questo

$dist(q,p_i) = dist(q,p_j) \leq dist(q,p_k)$ con $1 \leq k \leq n$. Per questo possiamo dire che q giace o su un arco o su un vertice di *Vor(P)*. Comunque sia, per il punto I, possiamo anche dire che q non giace su un vertice, ma giace su un arco definito dalla bisecante tra p_i e p_j. Nell'altro verso, sia una bisecante tra p_i e p_j un arco di *Vor(P)*. $C_p(q)$ deve contenere solamente p_i e p_j e nessun altro sito. □

3.2 Calcolare il Diagramma di Voronoi

Ora definita la struttura del diagramma di Voronoi passiamo a discutere degli algoritmi per calcolarlo.

L'Osservazione 1 del paragrafo precedente ci suggerisce un facile algoritmo con il quale possiamo calcolare il diagramma considerandolo intersezione di semipiani: per ogni sito p_i viene calcolata l'intersezione del semipiano $h(p_i,p_j)$ con $i \neq j$

```
Algoritmo IntersecaSemipiani(H)
      Input. Un insieme H di semipiani nel piano
      Output. La regione convessa C := h∈H h.
      1. if card(H) = 1
      2. then C ← l'unico semipiano h ∈ H
      3. else Dividi H in due insiemi H1 e H2 entrambi  di
      dimensione n/2  .
      4. C1 ←IntersecaSemipiani(H1 )
      5. C2 ←IntersecaSemipiani(H2 )
      6. C ←IntersecaRegioniConvesse(C1 ,C2 )
```

In particolare, questo algoritmo costa *O(nlogn)* perché genera una cella di *Vor(P)*, dunque per *n* celle e quindi per calcolare il diagramma di Voronoi costa $O(n^2 logn)$.

In altre parole, possiamo dimostrare che l'intersezione di due poligoni convessi, effettuata dall'algoritmo IntersecaRegioniConvesse(C1 ,C2) può essere svolta in tempo *O(n)* usando la tecnica della *sweep line* (in particolare, di questa tecnica ne parleremo successivamente). Inoltre, la natura delle regioni delle quali si vuole l'intersezione non condiziona il risultato finale, infatti, nel nostro caso C1 e C2 non sono necessariamente convessi, infatti possono essere illimitati, o degenerare in una retta o in punto. Per questo si ha:

$T(n) = O(1)$ *se* $n = 1$

$T(n) = O(n) + 2T(n/2)$ *se* $n > 1$

Risolvendo il sistema otteniamo che $T(n) = O(n\log^2 n) = O(n\log n)$. Questo algoritmo deve essere richiamato *n* volte, una per ogni sito e ciò implica che il tempo necessario per la costruzione del diagramma di Voronoi è $O(n^2 logn)$.

Il problema ha un lower bound di $\Omega(n\log n)$ ottenuto riducendo il problema di ordinamento al nostro problema. Non dimostriamo formalmente questo risultato, ma per i nostri scopi ci basta esserne a conoscenza. Per la dimostrazione completa si veda [4].

Alla luce della presenza di un lower bound ci spinge a considerare la prima soluzione eccessiva e perciò cerchiamo un algoritmo che sia ottimo.

Infatti nel prossimo paragrafo presenteremo un algoritmo il cui costo computazionale coincide con il lower bound di $\Omega(nlogn)$.

3.3 Algoritmo Fortune

Questo algoritmo prende il nome dal suo inventore, Steven Fortune, che pubblicò i risultati dei suoi studi nel 1986. Questo algoritmo sfrutta l'idea della *sweep line.*

Questi tipi di algoritmi sono stati introdotti per calcolare l'intersezione di segmenti su un piano. L'idea è quella di far scorrere da sinistra a destra una linea verticale, la *sweep line*. Tutto ciò che si trova a sinistra risulta già elaborato mentre quello che è a destra è ancora ignoto. L'output un'opportuna struttura dati. Tale struttura dati non deve cambiare mentre la linea viene spostata, a meno che questa non incontri un punto speciale detto *site event*. Col metodo della *sweep line* sono stati ottenuti algoritmi efficiente per diversi problemi di geometria computazionale.

Prima di continuare, c'è da precisare che l'algoritmo presentato fa scorrere questa linea dall'alto verso il basso. Nell'implementazione che presenteremo successivamente, questa linea si muoverà, addirittura dal basso verso l'alto. Comunque sia l'idea non cambia.

Sia l la *sweep line* e $P = \{p_1,p_2,....,p_n\}$ un insieme di punti del piano del quale si vuole calcolare $Vor(P)$.

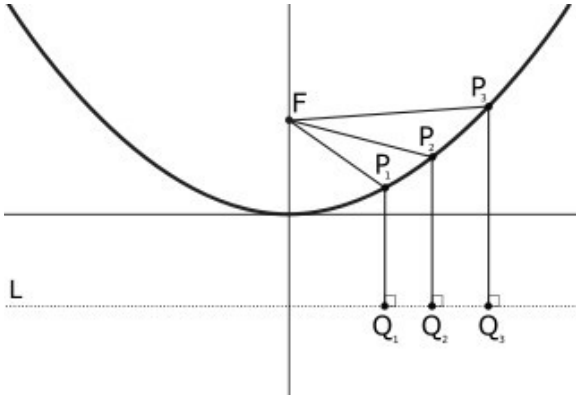
Sfortunatamente, non si può applicare la strategia generale in questo caso, perché ciò che è al di sopra di l può dipendere anche da ciò che è sotto. Questo fatto ci costringe a modificare l'algoritmo in modo tale che si possa tenere traccia di ciò di cui si è sicuri. Dovremmo, dunque, poter ricordare i *siti* sopra l che non potranno mai più essere condizionati dai *siti* che si trovano sotto l. Per ogni sito che si trova sopra l verrà costruita in modo dinamica una parabola la quale delimiterà la zona in cui i nuovi siti, che verranno scoperti dall'avanzamento della *sweep line*,non hanno alcuna influenza. L'unione di tutte queste parabole vanno a formare una nuova linea

denominata *beach line*, perché ricorda le onde che si infrangono sulle spiagge. Dunque, tutto ciò che sta sopra tale *beach line* è consolidato e non più modificabile, cioè la parte del diagramma di Voronoi disegnato è definitiva;

Per capire bene questa situazione bisogna capire come sono costruite queste parabole.

Per definizione sappiamo che quel luogo geometrico dei punti del piano equidistanti da un punto fisso F, detto *fuoco*, e da una retta fissa *d*, detta *direttrice.*

In questo caso, quindi avremo che il sito rappresenterà il fuoco della parabola e la direttrice è rappresentata dalla *sweep line l.* L'immagine seguente può aiutare a comprendere meglio.



Osservazione 2 *La beach line è monotona rispetto al piano delle ascisse, cioè, qualsiasi linea verticale la interseca in un solo punto.*

Inoltre la *beach line* presenta altre caratteristiche importati:

1. Ciascuna parabola associata a un punto può intervenire più di una volta a comporre diverse parti della spezzata
2. Per i *break-point* della *beach line* passano i lati del diagramma di Voronoi.

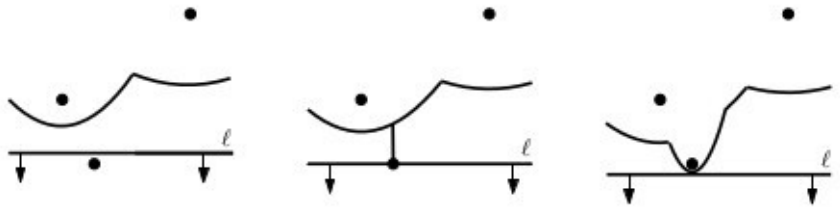
Tali *break-point* sono i punti in cui si incontrano parabole diverse.

Dimostrare le proprietà della *beach line* è semplice, basta utilizzare le proprietà delle parabole.

Quindi più che mantenere l'intersezione tra *Vor(P)* e la *sweep line* conviene mantenere la *beach line*, e dato che mantenere tutte le parabole sarebbe eccessivo, conviene tenere traccia dei suoi cambiamenti, cioè, tenere traccia di quando un arco di parabola appare o scompare.

Per prima cosa ci preoccupiamo di quando un arco compare. Ogni qual volta che la *l* incontra un nuovo sito siamo nella situazione di *site event*. In questo caso il nuovo sito è collegato alla *beach line* attraverso una parabola degenere di larghezza zero -corrisponde ad segmento verticale tra la parabola intersecata e il sito- e i due *break-point* combaciano in un punto solo. Mano a mano che la *sweep line* prosegue nella sua discesa la parabola si estende sempre più e quindi la *beach line* verrà modificata.

Lemma 1 *L'unico modo nel quale una nuova parabola può apparire è attraverso un site event.*



Dimostrazione Supponiamo per assurdo che una parabola B_j definita da un punto p_j appare nella *beach line*. Questa apparizione può avvenire in due modi.

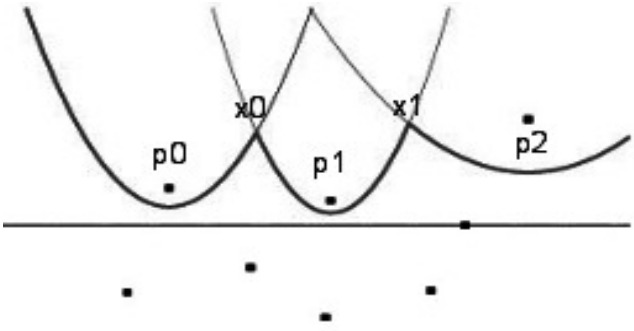
Il primo modo è che B_j appare nel mezzo dell'arco di una parabola B_i. Il momento in cui questo sta per accadere le tue parabole sono tangenti, cioè hanno un solo punto di intersezione, e sia l_y il punto di tangenza. Inoltre sia $p_j := (p_{j,x}, p_{j,y})$ allora la parabola B_j

è data da:

$$B_j := \frac{1}{2(p_{j,y} - l_y)}(x^2 - 2p_{j,x}\,x + p_{j,x}^2 + p_{j,y}^2 - l_y^2)$$

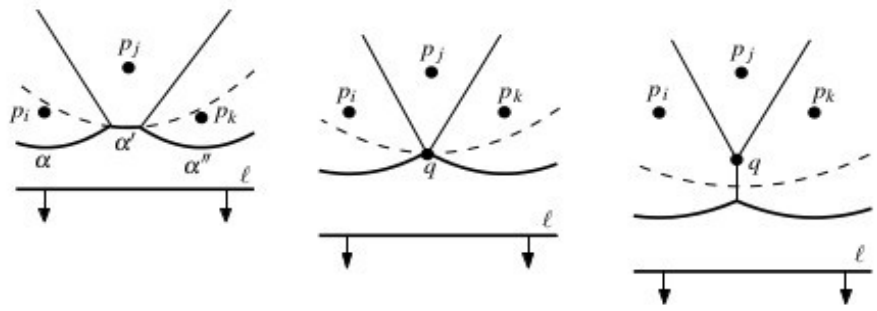
La formula per B_i è sicuramente simile. Sapendo che sia $p_{j,y}$ che $p_{i,y}$ sono più grandi di l_y allora è facile mostrare che è impossibile che B_j e B_i hanno un solo punto di intersezione e quindi è impossibile che B_j compaia attraverso il centro di un'altra parabola.

La seconda possibilità è che la parabola B_j compaia tra due parabole B_i e B_k. Sia q il punto di intersezione tra le due parabole e dal quale B_j sta per apparire, assumendo che B_i e B_k sono, rispettivamente, alla sinistra e alla destra del punto q. Quindi esiste un cerchio che contiene sulla sua circonferenza p_i, p_j e p_k -in questo senso orario- e inoltre è tangente alla *sweep line l*. In seguito a un piccolo spostamento di l questo cerchio ingloberà al suo interno almeno uno dei tre siti. Per tanto in uno spazio piccolo nelle vicinanze di q non può apparire nessuna parabola, mentre si muove la *sweep line*, perché o p_i o p_k saranno più vicini a l rispetto a p_j. □



Un'immediata conseguenza del lemma precedente è che la *beach line* può contenere fino a un massimo di *2n-1* archi di parabole. Questa conseguenza è banalmente dimostrabile, infatti, ogni sito può comparire nel mezzo di un arco e dividerlo in due creandone uno proprio.

L'algoritmo di Fortune gestisce anche un secondo tipo di evento: *circle event.* Tale evento permette di creare un vertice di *Vor(P)* e di poter eliminare un arco dalla *beach line*.



Per esempio ci troviamo nella situazione in cui nella *beach line* ci sono tre archi consecutivi a,a^* e a^{**}. Di questi tre archi a^* sta per sparire. Allora, in una maniera simile con la quale si è dimostrato il lemma 1, si può dimostrare che a e a^{**} non possono appartenere alla stessa parabola. Inoltre siano p_i,p_j e p_k i punti relativi rispettivamente ad a,a^* e a^{**},nel momento in cui a^* sparisce le tre parabole si incontrano in un punto q. Tale punto è equidistante sia dai tre siti che dalla *sweep line l*. Pertanto esiste un cerchio, con centro in q, che tocca p_i,p_j e p_k e è tangente ad *l.* Siamo sicuri che all'interno di questo cerchio non ci sia un altro sito, perché altrimenti si contraddirebbe il fatto che q si trovi sulla *beach line.* In questo modo q diventerà un vertice di *Vor(P)* e non c'è da meravigliarsi, poiché sapendo che i *break-point* rappresentano archi del diagramma, un loro incontro genera un vertice.

Il Lemma seguente ci formalizza quanto detto riguardo i *circle event.*

Lemma 2 *L'unico modo nel quale un arco può sparire dalla beach line è attraverso un circle event*

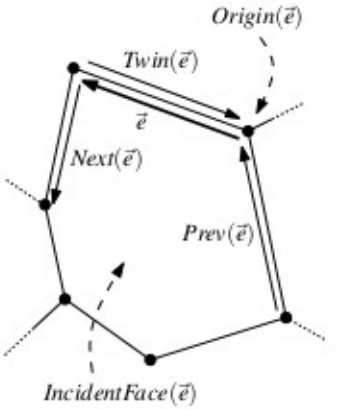
Non c'è una vera e propria dimostrazione al Lemma 2. Per spiegarlo e dimostrare la sua validità si possono usare osservazioni o punti già dimostrati precedentemente.

Ora siamo pronti per mostrare l'algoritmo. Il *site event* e il *circle event* sono

fondamentali per l'esecuzione dello stesso, poiché, in pratica, l'algoritmo di Fortune si preoccupa di gestire al meglio una lista di eventi. Naturalmente gli eventi da gestire sono semplicemente i due presentati dai lemmi.

Prima di mostrare e analizzare lo pseudo codice dobbiamo, però, presentare le strutture dati utilizzate. Risulta, infatti, chiaro che ci sia bisogno di strutture efficienti:

- *Vor(P)* viene mantenuta attraverso una struttura dati particolare e molto efficace: la *Doubly-Connected Edge List* (DCEL).



Questa struttura dati contiene le informazioni riguardanti i vertici, gli archi e le facce, cioè i poligoni che si vanno a creare all'interno di un ciclo. Più precisamente ogni vertice è caratterizzato dalle coordinate e contiene il riferimento ad un half-edge uscente da esso. Gli half-edge servono a rappresentare gli archi del grafo. Questa è una particolare rappresentazione poiché ognuno di essi rappresenta un “lato” dell'arco. In altre parole, ogni arco ha due “lati”, ognuno dei quali ha un'origine differente. Le due origini, comunque sia, sono i vertici collegati dall'arco. Ogni half-edge *e*, oltre a mantenere il riferimento all'altro “lato” dell'arco, cioè all'altro half-edge che costituisce l'arco, che chiameremo *twin(e)*, mantiene un riferimento alla faccia incidente, ovvero quella che si trova alla sinistra di *e*. Le facce sono costituite da una sequenza di half-edge. Comunque sia, all'interno di ogni faccia c'è il riferimento ad uno solo di essi, e le informazioni necessarie per scorrere la lista di half-edge sono contenute negli half-edge stessi, come se fossero una lista doppiamente linkata. In fase di costruzione risulta molto utile, purtroppo, il diagramma contiene anche archi infiniti che non possono essere rappresentati attraverso

la DCEL quindi dopo la costruzione si necessita di un'operazione ulteriore per la costruzione di un contorno, *bounding box*, in modo tale da poter mantenere il diagramma in maniera corretta.

- La *event-queue Q* serve a tenere traccia degli eventi che dovranno essere gestiti durante l'esecuzione. Questa struttura dati è una coda con priorità, nella quale la priorità è l'ordinata dell'evento. I *site-event* sono aggiunti tutti all'inizio, mentre i *circle-event* saranno aggiunti, mano a mano che la *sweep line* avanza. Questi ultimi vengono aggiunti con priorità data dall'ordinata del punto più basso del cerchio che andranno a costruire.

- La *beach line* è rappresentata da un albero binario bilanciato Γ. Tale albero mantiene gli archi di parabola nell'ordine corretto, così come compaiano nella *beach line*. Questo significa che ogni foglia μ dell'albero definisce un arco, e quindi la prima foglia definisce il primo arco, la seconda il secondo e così via. Comunque sia, le foglie non memorizzano la parabole, bensì i siti corrispondenti all'arco di parabola. I nodi interni, invece, memorizzano i *break-point* nella *beach line*. Tali nodi interni memorizzano una coppia ordinata $<p_i,p_j>$ dove p_i definisce la parabola alla sinistra e p_j la parabola alla destra del *break-point*. Grazie alla struttura bilanciata di Γ la ricerca in base alle parabole è possibile in tempo *O(logn)* ed è possibile semplicemente attraverso confronti sulla coordinata x dei siti che compaiono all'interno dei *break-point.* Infine Γ contiene puntatori alle altre strutture dati utilizzate dall'algortimo. Ad ogni foglia μ è associato un evento della *event-queue*, in particolare memorizza il *circle event* che farà sparire l'arco relativo ad μ. Naturalmente questo puntatore può essere settato a **null** se non esiste il *circle event* o se ancora non è stato trovato. Ogni nodo interno, invece, memorizza l'arco relativo al *break-point* e più precisamente uno dei due *half-edge* che rappresentano l'arco.

Durante l'esecuzione dell'algoritmo la *beach line* può cambiare la sua composizione: un arco può sparire, grazie ad un *circle* event,o può essere aggiunto, attraverso un

site event. Già sappiamo tutto sui *site event* e, invece, per quanto riguarda i *circle event* bisogna fare delle ulteriori riflessioni.

Affinché si possa incontrare, durante l'esecuzione dell'algoritmo, un *cicle event* deve essere necessariamente che una tripla di archi ha quello interno che sta per sparire. Ciò è vero anche alla luce di quanto afferma il Lemma 2. Non sempre però una tripla genera tale evento. Può capitare, infatti, che due *break-point* non convergano, in altre parole può darsi che due archi non si incontrino in un punto come quando, per esempio, partono entrambi dallo stesso nodo. Detto ciò, risulta ovvio che durante l'esecuzione bisogna controllare se una tripla converge o meno. Il secondo caso, pertanto, è detto *false alarm*, falso allarme, poiché nonostante i *break-point* convergano il *circle event* non avrà mai luogo .

L'algoritmo di Fortune, durante la sua esecuzione, non fa niente altro che questo: per ogni evento di sito trovato controlla le nuove triple ed eventualmente aggiunge il *circle event.* In caso in cui debba processare un evento di cerchio, crea un vertice e ricontrolla le nuove triple.

Lemma 3 *Ogni vertice di Vor(P) viene rilevato per mezzo di un circle event.*

Dimostrazione: Sia q un vertice di *Vor(P)*, e siano p_i,p_j e p_k tre punti che giacono sulla circonferenza di $C_p(q)$. Per il Teorema 3 questi punti e il cerchio esistono. Per semplicità diciamo che quei siti sono gli unici che giacciono sulla circonferenza del cerchio e che nessuno si trova nel punto più basso di esso. Inoltre, diciamo che una visita in senso orario del cerchio di permette di incontrare p_i,p_j e p_k in questo ordine. Prima che la *sweep line* incontri il punto più basso di questo cerchio – scatenando un *circle event*- sulla *beach line* ci sono tre archi a,a^* e a^{**} definiti rispettivamente dai siti p_i,p_j e p_k (l'arco che sparirà sarà a^*). considerando il caso in cui siamo in modo infinitesimale al punto più basso di $C_p(q)$. In questo caso esiste un cerchio, diverso da $C_p(q)$, che contiene sulla circonferenza p_i e p_j ed è inoltre tangente alla *sweep line*. Allo stesso modo esiste un terzo cerchio differente dai precedenti che contiene sulla circonferenza p_j e p_k ed è anche esso tangente alla *sweep line.* E' facile, allora capire che i due archi definiti da p_j sono in realtà lo stesso arco e pertanto si è difronte ad

una tripla. Questo ci induce a dire che il *circle event* è già presente in *Q* prima che esso scaturisca, in modo tale da rilevare correttamente il vertice. □

Ora siamo pronti per descrive lo pseudo codice in dettaglio. Bisogna comunque osservare che idealmente, nonostante la coda degli eventi *Q* sia vuota, la *sweep line* continua a scendere e che i *break-point* rappresentano gli half-edge infinti. Pertanto è necessario definire un *bounding box* al quale si devono unire tali half-edge infiniti.

Algoritmo VORONOI DIAGRAM(P)

Input. insieme $P := \{p_1 , . . . , p_n \}$ di *siti* nel piano.
Output. Il diagramma di Voronoi Vor(P) con il bounding box, memorizzata in una DCEL *D*
1. Inizializzare *Q* con tutti i *site event*. Mentre Γ e D sono inizializzati vuoti
2. **while** Q is not empty
3. **do** Rimuovi l'evento con la y più alta da *Q*.
4. **if** l'evento è un *site event*, del sito p_i
5. **then** HANDLESITEEVENT(p_i)
6. **else** HANDLECIRCLEEVENT(γ), dove γ è la foglia di Γ che rappresenta l'arco che sparirà
7. Costruire il bounding box aggiornando *D*
8. Costruire correttamente le facce aggiornando D

Le procedure a cui si fa riferimento servono a gestire i due tipi di evento di cui si è discusso prima.

Procedura HANDLESITEEVENT(p_i)

1. se Γ è vuoto allora inserisci p_i e return. Altrimenti continua con l'esecuzione.

2. Cerca l'arco *a* verticale a p_i. Se la foglia che rappresenta questo arco ha un puntatore ad un vento di cerchio, esso va tolto da *Q*, perchè si tratta di un *false alarm*.

3. Sostituisci tale foglia con un sotto albero che ne ha tre di cui quella nel mezzo rappresenta p_i e le altre due rappresentano p_j, il sito che prima era memorizzato da *a*. Vengono quindi creati due nuovi *break-point* che memorizzano $<p_j,p_i>$ e $<p_i,p_j>$. Ribilanciare se necessario l'albero.

4.Creare un nuovo half-edge che divide $Vor(p_i)$ e $Vor(p_j)$ che sarà tracciato dai nuovi break-point

5. controllare le nuove triple generate. Comunque, il nuovo arco generato non si trova, in questo caso, nel centro di una tripla, per le considerazioni già fatte precedentemente. Per questo si controllano le triple alla destra e alla sinistra del nuovo arco.

Procedura HANDLECIRCLEVENT(γ)

1. cancella la foglia γ che rappresenta l'arco che scompare. Aggiornare opportunamente i nodi interni dell'albero e in caso ribilanciarlo. Cancellare tutti i *circle event* collegati all'arco che è stato tolto.

2. aggiungere in D il centro del cerchio come vertice di *Vor(P)*. Aggiornare opportunamente le strutture dati e creare due half-edge che rappresentano l'arco uscente dal vertice.

3. controllare le nuove triple di archi causate dalla scomparsa dell'arco. Eventualmente aggiungerli a *Q*.

Lemma 4 *L'algoritmo di Fortune ha costo computazionale O(nlogn) ed usa O(n) spazio.*

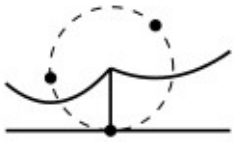
Dimostrazione: è facile vedere che le operazioni compiute dall'algoritmo nelle strutture dati hanno costo computazionale costante. Solamente la ricerca dell'arco superiore costa *O(logn)*. E' facile anche vedere che questa ricerca è fatta per *n site event*. I *circle event* sono al massimo *2n-5*. Inoltre neanche tutti gli eventi di cerchio sono processati. Quindi è facile vedere che tutto ciò implica il lemma. □

Infine esistono tre casi degeneri da discutere e controllare che l'algoritmo di Fortune li riesca a gestire in modo corretto.

Il primo caso particolare deriva da un normale problema di algoritmi che utilizzano la *sweep line*. In caso in cui due *siti* abbiano la stessa y essi dovranno essere scelti in base alla x, che sicuramente sarà diversa. Più attenzione richiede il caso in cui questa

situazione si verifica all'inizio dell'algoritmo, cioè, i primi due *siti* hanno la stessa ordinata: in questo caso bisogna aggiungere del codice aggiuntivo per gestire questa situazione cosicché l'algoritmo possa proseguire.

Il secondo caso riguarda la generazione di vertici. L'algoritmo prende in considerazione solamente triple, ovvero tre siti, ma può succedere che ce ne siano quattro o più. L'algoritmo gestisce questa situazione inserendo più vertici con le stesse coordinate, pur continuando a gestire triple. Ciò, comunque, richiede un controllo finale per togliere vertici in eccesso.



L'ultimo caso particolare, anche esso gestito bene dall'algoritmo si verifica quando un *site event* interseca la *beach line* in un *break-point*. In questo caso, l'algoritmo continua la normale esecuzione e riscontrerà un *circle event* e il caso degenere viene gestito correttamente.

Dopo queste osservazioni si può quindi concludere con il seguente teorema.

Teorema 4 *Il diagramma di Voronoi di un insieme di n siti su un piano può essere calcolato con un algoritmo di sweep line in tempo O(nlogn) usando O(n) di spazio.*

4. Implementazione Diagramma Voronoi

Qui di seguito e nel capitolo successivo ci preoccuperemo di analizzare l'aspetto implementativo del sistema. In particolare, prima discuteremo dell'implementazione dell'algoritmo di Fortune e nel capitolo successivo descriviamo l'implementazione del sistema intero.

Il linguaggio di programmazione scelto è Java, per questioni di diffusione e di comodità.

Per l'implementazione sono stati utilizzati codici sorgenti di alcuni progetti Open Source e quindi si è cercato di mantenere le loro politiche aperte e si è deciso di rendere pubblico tutto il codice sorgente sviluppato per questa tesi. Comunque sia, l'impegno nostro nei confronti degli sviluppatori, non è solo di renderlo pubblico, ma di mantenere inoltre le stesse licenze.

Di seguito verrà analizzata ogni parte del progetto di cui verrà discussa anche la relativa complessità.

4.1 Diagramma di Voronoi

Per quanto riguarda la realizzazione del diagramma ci si è basati sull'implementazione dell'algoritmo di Fortune presente nel codice di Geogebra.

Geogebra è un software Open Source di matematica dinamica libero e multi-piattaforma per tutti i livelli scolastici che comprende geometria, algebra, tabelle, grafici, statistica e analisi in un pacchetto semplice e intuitivo.

Questo software si è rivelato utile anche durante l'implementazione perché ci ha permesso di visualizzare gli esperimenti compiuti cosicché potessimo vedere ciò che non andava.

4.1.1 Analisi Codice Geogebra

L'algoritmo di Fortune implementato in Geogebra ricalca in linea di massima quello della descrizione teorica, tranne che per delle differenze di cui bisogna tener conto.

La *beach line* è rappresentata attraverso un albero binario di ricerca, non necessariamente bilanciato. Le foglie dell'albero, così come succedeva nella presentazione teorica dell'algoritmo, mantengono gli eventi e non gli archi delle parabole. Queste ultime vengono calcolate quando si vuole sapere il punto preciso in cui un nuovo sito modifica la *beach line.* Le foglie dell'albero sono mantenute in una lista doppiamente linkata cosicché si possa risalire alla precedente o alla successiva in tempo costante.

La coda degli eventi è stata implementata con un TreeMap [1]. Il TreeMap è un'implementazione dell'albero Red-Black con lo scopo di garantire le operazioni `get`, `put` e `remove` in tempo *O(logn)*.

Il codice di Geogebra presenta diverse strutture dati atte al mantenimento del diagramma, ma tra queste non compare la *Doubly-Connected Edge List* (DCEL). Ne esiste una simile, ma è si è deciso di non utilizzarla se non per prendere codice utile per l'implementazione della DCEL.

L'ultima differenza, non di notevole rilevanza, è il senso di spostamento della *sweep line* che nel caso di Geogebra si muove dal basso verso l'alto. Tale scelta influenza il codice aggiuntivo in caso in cui i primi due siti compaiono con lo stessa ordinata: in questo caso viene modificata l'ordinata in misura trascurabile. Per il resto dell'esecuzione, invece, la scelta di invertire l'ordine di scansione della *sweep line* non presenta particolari conseguenze.

4.1.2 Implementazione Finale

Partendo quindi da un'ottima base siamo arrivati ad implementare l'algoritmo di Fortune così come è stato studiato teoricamente.

Algoritmo VORONOI DIAGRAM(P)

Input. insieme $P := \{p_1 , . . . , p_n\}$ di *siti* nel piano.

Output. Il diagramma di Voronoi Vor(P) con il bounding box, memorizzata in una DCEL *D*

1. Inizializzare *Q* con tutti i *site event*. Mentre Γ e D sono inizializzati vuoti
2. **while** Q is not empty
3. **do** Rimuovi l'evento con la y più alta da *Q*.
4. **if** l'evento è un *site event*, del sito p_i
5. **then** HANDLESITEEVENT(p_i)
6. **else** HANDLECIRCLEEVENT(γ), dove γ è la foglia di Γ che rappresenta l'arco che sparirà
7. Costruire il bounding box aggiornando *D*
8. Costruire correttamente le facce aggiornando D

Attraverso lo pseudocodice andremo a discutere le parti salienti dell'implementazione.

Innanzitutto bisogna discutere della DCEL. La struttura implementata si discosta un po' dalla teoria. Essa, di per sé, mantiene una lista di vertici, half-edge e face, ma i metodi di inserimento sono dipendenti dall'algoritmo e inoltre si preoccupa della costruzione delle face, andando contro le consuete regole dei linguaggi di programmazione ad oggetti.

Altre modifiche riguardano i vertici: essi non mantengono soltanto un half-edge tra quelli da lui uscenti, bensì li ricorda tutti. In più ogni vertice è fornito di un tag che ci informa se sia o meno un vertice all'angolo, e in tal caso ci dice anche quale.

Le facce, oltre ai normali campi, presentano anche il sito a cui appartengono.

Modifiche sostanziali invece sono state fatte per quanto riguarda gli half-edge. Più propriamente sono state fatte delle sia aggiunte che delle modifiche. Sono stati aggiunti campi necessari per la rappresentazione della retta su cui giace. Sia

$$y = mx + b$$

la formula della retta, l'half-edge mantiene le informazioni riguardo *m* (il coefficiente angolare) e *b*. Inoltre mantiene informazioni su i due siti che divide e dai quali viene appunto calcolata la formula della retta: essendo quest'ultima la bisecante perpendicolare del segmento passante tra i due siti, facilmente si può trovare la formula sfruttando le proprietà geometriche. Oltre ai campi precedenti, vengono mantenute informazioni riguardante la direzione dell'half-edge, utili per la costruzione del Bounding Box di cui discuteremo in seguito.

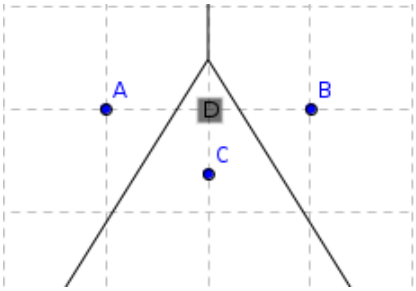
Una semplice modifica è stata necessaria anche nei nodi interni dell'albero binario: in pratica essi memorizzavano due archi, uno entrante e uno uscente . Questa informazione ci sembrava superflua in quanto i due archi sono “gemelli” ed in ognuno di essi c'è il riferimento all'altro. Perciò si è preferito che ne venga memorizzato uno solo.

I punti 1 – 6 dello pseudocodice erano già implementati nel codice di Geogebra, le uniche modifiche apportate sono state quelle relative ai giusti collegamenti nella DCEL oltre a quelle necessarie alla costruzione del Bounding Box.

Infatti i punti 7-8, essendo dipendenti dalla DCEL, non sono stati implementati nel codice di Geogebra, e ciò ha richiesto attenzione al riguardo.

I problemi riscontranti nella costruzione del Bounding Box riguardavo in particolare l'intersezione tra gli half-edge infiniti e i lati del Box. Infatti se osserviamo gli half-edge come delle semirette infinite, si vuole sapere precisamente la direzione di queste semirette. Questo problema sorge dal fatto che dell'half-edge si conosce la retta su cui giace, ma non su quale parte di essa.

Dunque necessitava un modo per riconoscere la direzione giusta. Inizialmente si pensava di tener conto del punto medio tra i due siti divisi dall'half-edge ma può capitare che esso non sia sulla semiretta corretta. Infatti potrebbe capitare che si trovi in una cella del diagramma.



Bisogna quindi scoprire quando ciò avviene. In pratica, il punto medio può comparire in una cella quando si è in prossimità di un vertice. Questo ci suggerisce di controllare questa possibilità quando si incontra un *circle event.* Infatti, la cella in cui potrebbe comparire il punto medio è proprio quella dell'evento che sta per sparire con l'evento.

Teorema 1 *Il punto medio può capitare fuori dalla semiretta corretta solo in prossimità di un vertice. Se così fosse, la cella in cui compare è quella di mezzo nel circle event.*

Dimostrazione Per assurdo immaginiamo che così non fosse. Allora potremmo avere due casi in cui il punto medio esce dalla semiretta non in prossimità di un *circle event*:

1. Nel primo caso il punto medio potrebbe capitare fuori dall'half-edge in caso di *site event*.
2. Nel secondo caso, sempre difronte ad un *circle event*, il punto medio compare in un'altra cella e non in quella di mezzo.

É facile vede che entrambi i casi sono delle contraddizioni, perché altrimenti si dovrebbe dimostrare che un punto medio di un segmento sia esterno ad esso. □

Dunque, alla luce di questo risultato, basta semplicemente controllare la distanza del punto medio rispetto ad uno dei siti da cui è calcolato e il terzo sito, ovvero quello della cella di mezzo. In altri termini, sia d_1 la distanza tra il punto medio e uno dei due siti (i due siti sono equidistanti) e sia d_2 la distanza tra il punto medio e il terzo sito: se $d_1 > d_2$, quindi il punto medio compare nella cella di mezzo, allora come

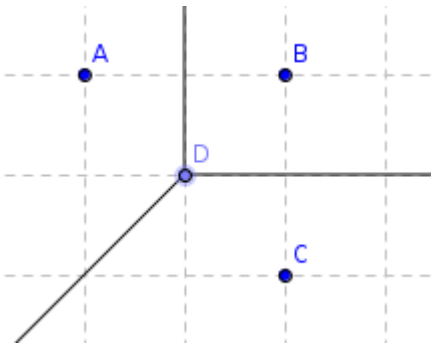
direzione si prende il punto opposto al punto medio rispetto all'origine, altrimenti viene considerato il punto medio.

Un attimo di attenzione necessita, invece, il caso in cui il punto medio cade sull'origine della semiretta. Dobbiamo notare che questo è un caso particolare difficile da presentarsi nelle applicazioni reali, ma comunque è una buona cosa gestirlo correttamente.

Anche qui ci sono due distinti scenari da distinguere:

a) il primo è lo scenario in cui bisogna gestire una tripla: in questo caso si esegue una soluzione simile per il caso precedente. Stavolta si utilizzano due punti differenti, uno opposto all'altro - rispetto all'origine - sulla retta. Se ne sceglierà dunque uno come direzione.

b) Il secondo scenario è più delicato. Come mostra l'immagine seguente, esso potrebbe assomigliare al precedente, ma non è così. L'esecuzione dell'algoritmo non gestisce correttamente la situazione della semiretta in basso a sinistra. In questo caso non si può conoscere la giusta direzione della semiretta, ciò succede comunque solamente con semirette che vengono da basso e per questo nella scelta della direzione, in caso di dubbio, viene scelta quella che tende verso il basso.



Un'ultima precisazione. Per quanto riguarda lo scenario b) tali considerazioni sono

ammissibili sapendo che la *sweep line* si muove dal basso verso l'alto. In caso in cui si avesse lo spostamento inverso, non si vede nessun ragionevole dubbio, per pensare che la scelta potrebbe essere invertita.

Il punto numero 8 dell'algoritmo riguarda la costruzione delle facce. L'idea è semplice. Si scorre la lista degli archi, se ne trova il primo, che chiameremo *e*, in cui la faccia incidente non è definita e partendo da *e* si scorrono tutti i successivi, fino a tornare, ad *e*. Questo è possibile per costruzione del diagramma. In questa operazione vengono anche associati i siti relativi alle facce. Tale associazione è po' delicata. Infatti i siti divisi da ogni half-edge possono essere messi in maniera disordinata e questo complica la ricerca del sito corretto. Sappiamo però che ogni half-edge memorizza il sito corretto per questo viene controllato quante volte un sito compare in una lista di appoggio: se un sito compare almeno due volte, allora, si può dire con certezza che esso è proprio il sito relativo alla faccia.

Dopo la costruzione delle facce è stata implementata un'altra funzione per completarle. Per completare le facce si intende assegnare tutte le informazioni agli archi che costituiscono il Bounding Box. Ciò è necessario poiché i siti di tali archi non sono definiti e in questo modo si completano le loro informazioni.

4.1.3 Analisi della Complessità

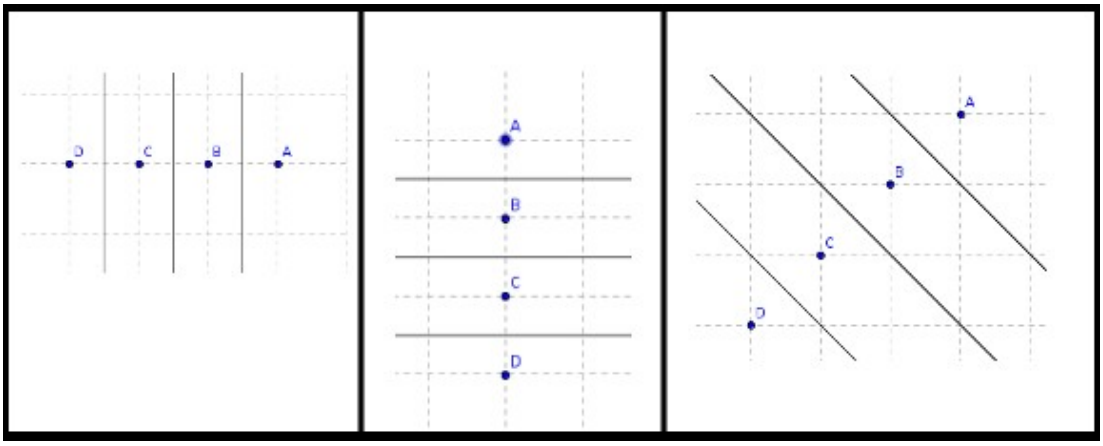
Si ritiene necessaria l'analisi della complessità del codice per via delle differenze che ci sono tra lo pseudocodice e l'implementazione finale. Verranno quindi analizzati anche i metodi aggiunti. L'obbiettivo, ovvio, dello sviluppo era di mantenere il tempo di *O(nlog)* anche nell'implementazione.

Il primo punto di cui discutere è l'albero binario di ricerca utilizzato per mantenere la *beach line*. Se fosse, per esempio, un AVL si avrebbero garanzie migliori per quanto riguarda i tempi di ricerca. In effetti un normale albero binario potrebbe essere altamente sbilanciato facendo degenerare il tempo di ricerca addirittura a *O(n)*. Per spiegare la scelta,dunque, bisogna ragionare sugli elementi memorizzati all'interno dell'albero durante l'esecuzione, e quindi su quanto possa essere sbilanciato.

Il ragionamento al riguardo non può non tenere conto del numero di nodi che causano lo sbilanciamento stesso. Tale numero deve essere considerevole affinché si verifichi una situazione tale da peggiorare le prestazioni in modo significativo. Sappiamo che un nuovo arco viene aggiunto sulla *beach* line attraverso un *site* event. Dunque anche il numero dei *site event* deve essere elevato. Inoltre, per ogni nuovo arco di parabola aggiunto si vanno ad analizzare due nuove triple che potranno avere o meno un *circle event.* Ciò ci induce a pensare che anche il numero dei *cirle event* sarà elevato e per questo molti degli archi di parabola presenti sulla *beach line* spariranno e, con essi, i nodi interni e le foglie dell'albero. In questo modo si mantiene approssimativamente limitato il bilanciamento dell'albero.

Un discorso diverso si avrebbe quando i *siti* sono collineari. Sia S un insieme di *siti* collineari, il diagramma di Voronoi su S è composto solamente da rette parallele. Esistono tre casi in cui questa situazione può verificarsi:

1. i *siti* presentano la stessa ordinata
2. i *siti* presenta la stessa ascissa
3. i *siti* sono collineari e obliqui



In tutti e tre i casi l'albero presenterà uno sbilanciamento che causerà degenerazione nei tempi di ricerca. Tuttavia, bisogna anche dire che questi scenari sono assai rari in condizioni normali.

Effettuate queste osservazioni si possono, allora, ritenere eccessive le operazioni di ristrutturazione degli AVL.

Altro punto di cui discutere è la costruzione del Bounding Box. L'idea semplice di tale costruzione, come già detto, è controllare ogni half-edge infinito su quale lato interseca il contorno. Vengono dunque ricercati tutti gli half-edge infiniti dalla *beach line* – con una visita nell'albero *O(n)* – e per ognuno, in base alla direzione, si aggiungono alla DCEL le intersezioni. Tutte queste operazioni sono costanti e quindi in tempo lineare il Bounding Box viene costruito.

Terzo punto interessante è la costruzione delle facce. Anche se il metodo presenta cicli annidati possiamo ritenere che la complessità sia lineare – in *m*, numero di half-edge , poiché si entra nel ciclo solo in una determinata condizione, altrimenti si prosegue. In caso si entri, vengono visitati tutti gli half-edge che costituiscono una faccia. Alla luce del fatto che ogni half-edge appartiene ad una sola faccia, il numero degli half-edge processati è proprio pari al numero degli half-edge – quindi ad *m*.

Per quanto riguarda la costruzione delle facce c'è un'altra parte in cui lo studio della complessità è delicato ed è quello relativo all'assegnazione del sito alla relativa faccia. Per ogni half-edge se tale sito non è definito si controlla che i due siti dell'half-edge compaiono nella lista di supporto. Questo controllo potrebbe costare parecchio, ma siamo sicuri che questa operazione viene fatta solamente due volte: la prima con la lista vuota e la seconda con solamente due siti all'interno. Pertanto possiamo assumere che tale operazione abbia costo costante.

Ultimo punto di cui discutere è il completamento delle facce. Anche in questo metodo sono presenti cicli annidati, ma si nota facilmente che ciò che viene realmente processato sono gli half-edge. Questa osservazione ci induce a fare un'analisi simile a quella relativa alla costruzione de facce e quindi si può dire che il costo di tale metodo è *O(m)* dove *m* è sempre il numero di half-edge.

Per quanto riguarda il resto del codice non c'è niente da aggiungere. Si rispetta il tempo teorico.

Infine precisando che $m = O(n)$ con n è il numero di vertici concludiamo che la nostra implementazione dell'algoritmo di Fortune calcola il diagramma di Voronoi in in tempo O(nlogn) mantenendo le informazioni in O(n) di spazio.

5. Implementazione del Sistema di Geolocalizzazione

Una volta presentata l'implementazione del diagramma di Voronoi, adesso presentiamo le implementazioni, e le relative scelte, del sistema. Allo stesso modo del capitolo precedente, presentiamo l'implementazione e la complessità computazionale che caratterizza l'implementazione.

5.1 Parser File OSM e Costruzione del Grafo

OpenStreetMap permette di gestire le informazioni riguardanti le mappe attraverso diversi tipi di file. Tra i più importanti ricordiamo

- OSM XML file -file xml fornito dalle API
- PBF file – file binario molto compresso simile alle API
- o5m file – utile per le esecuzioni veloci, usa codifica PBF, ha una struttura xml
- OSM JSON – una variante JSON del file xml

La nostra scelta è ricaduta sul file OSM XML. Essa è stata dettata dalla convenienza in quanto questo tipo di file è facile da processare – essendo un file xml – e perché è il più usato per mantenere le informazioni riguardanti le mappe di OpenStreetMap.

La struttura di tale file è semplice:

- tag **<?xml ?>** tipico del formato xml generale
- tag **<osm> </osm>** delimitatore dell'intero file. Ogni altro tag si trova al suo interno
- tag **<bounds />** definisce il confine dei dati contenuti
- tag **<node> </node>** definisce il nodo. Eventualmente può anche auto-chiudersi se non presenta tag annidati
- tag **<way> </way>** definisce una sequenza di nodi e per tanto presenta una lista di nodi annidati. Inoltre altri tag definiscono di che *way* si tratta

- tag **<relation> </relation>** definisce una relazione tra *nodi* o *way*. Esempio di relazione sono le linee del autobus.

Per maggiori informazioni si consiglia di andare a consultare la wiki ufficiale di *OpenStreetMap* [2].

Per processare il file comunque ci siamo serviti del lavoro di Marcio Aguiar e Willy Tiengo [3] i quali hanno implementato un Dom Parser per questo tipo di file. Sono state comunque apportate modifiche necessarie al nostro scopo. In particolare sono state aggiunte classi per la costruzione del grafo con le informazioni tratte dal file OSM. Altre modifiche riguardano le classi delle utility per la gestione delle coordinate geografiche.

Una volta estratte le informazioni dal file OSM. Possiamo procedere con la costruzione del grafo.

Sappiamo che il file OSM contiene sia la lista di nodi con tutte le informazioni che li riguardano sia la lista di tutte le *way*, le quali contengono a loro vola una lista di identificativi dei nodi di cui è composta. Per la costruzione del grafo, per prima cosa abbiamo estrapolato tutti i nodi. Successivamente abbiamo analizzato ogni *way* per scoprire i vicini di ogni nodo e inserire le informazioni ottenute all'interno della lista di adiacenza. Più precisamente ad ogni nodo è associata una lista di oggetti nei quali vengono memorizzati il vicino, la distanza che intercorre tra i due nodi e la *way* della quale fanno parte, memorizzata attraverso l'identificativo unico fornito nel file OSM.

Prima di proseguire dobbiamo fare una precisazione. Nel Cap. 2 abbiamo definito *way* come una sequenza di strade, sentieri, piste ciclabili e così via, ma in realtà OpenStreetMap considera *way* qualsiasi cosa che sia una sequenza di nodi. Per questo anche confini nazionali, confini geografici, rete ferroviaria e altre cose di questo genere sono considerate *way*. Comunque sia nel proseguo della tesi continueremo ad usare la definizione di *way* presentata nel Cap. 2. Comunque sia, per utilizzare questa definizione a livello applicativo, bisognerà filtrare il file OSM affinché mantenga solo le informazioni riguardanti la rete stradale.

Come già accennato nei capitoli precedenti, affinché l'esecuzione degli esperimenti vada a buon fine bisogna aggiungere dei nodi fittizi al grafo. Questa operazione, tuttavia viene svolta prima della costruzione stessa, cosicché l'algoritmo atto alla costruzione del grafo possa costruire il grafo completo subito, senza bisogno di modificarlo di nuovo.

Appare chiaro che il costo computazionale della costruzione del grafo è elevato: il Parsing del file xml è necessariamente legato alle dimensione del file, e le informazioni contenute in esso sono considerevoli anche se si tratta di una mappa limitata, come potrebbe essere un quartiere. Mentre, la costruzione del grafo (la lista di adiacenza), vero e proprio, è veramente dispendiosa poiché le *way* vengono processate due volte, e ogni volta per ogni *way* vengono estrapolate informazioni sui nodi di cui sono composte. Una prima volta, le *way* sono processate per permettere l'aggiunta dei nodi fittizi, la seconda volta vengono processate per estrapolare le informazioni affinché si possa costruire la lista di adiacenza. Quindi sia *m* il numero delle *way* e sia *n* il numero dei nodi: i nodi vengo processati in $O(n)$ di tempo; invece, entrambe le analisi delle *way* dipendono sia dal numero di *way* che da numero dei nodi di cui sono composte. Nel caso più eccessivo ogni *way* contiene al suo interno ogni nodo e quindi il grafo si costruirebbe nel caso peggiore in $O(n+nm)$ di tempo e le informazioni verrebbero mantenute in $O(nm)$ poiché il grafo non è diretto.

5.2 Costruzione del diagramma di Voronoi

Una volta costruito il grafo, bisogna costruire il diagramma di Voronoi relativo al grafo. Abbiamo già discusso dell'implementazione dell'algoritmo di Fortune nel capitolo precedente, ma bisogna fare qualche considerazione per ottenere buoni risultati.

Come descritto precedentemente, la nostra implementazione dell'algoritmo di Fortune costruisce i diagrammi di Voronoi relativi ad un grafo che giace su di una superficie piana, come se giacesse su un piano cartesiano. Diversamente, le coordinate estratte dal file OSM sono geografiche e quindi si riferiscono a una

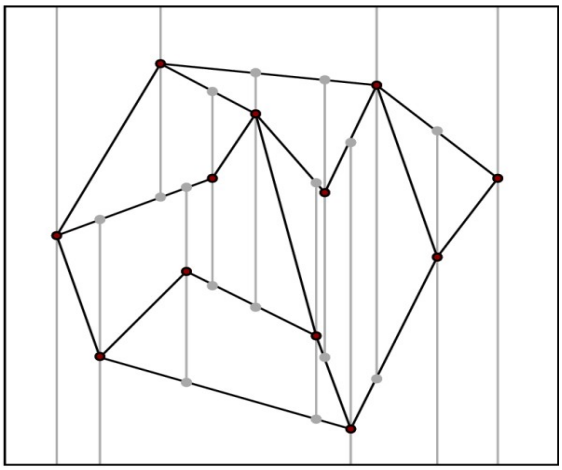
superficie non piana. Inoltre, c'è anche da dire che le distanze da noi prese in considerazione sono relativamente basse e ciò potrebbe indurci a considerare la superficie come piana. Allo scopo è stata implementata una funzione di traduzione tra coordinate geografiche e coordinate cartesiane . Nella nostra implementazione abbiamo modificato la funzione già implementata in [5], comunque sia una valida soluzione è presentata anche in [4]. In questo modo il diagramma di Voronoi può essere costruito in maniera corretta.

Dobbiamo inoltre specificare che questa funzione di traduzione considera la Terra come se fosse sferica, utilizzando il raggio medio terrestre.

Risulta ovvio che dobbiamo ottenere le coordinate cartesiane anche dei punti della traccia che si vuole approssimare.

5.3 Point Location

Abbiamo già accennato al Point location e alla struttura di ricerca nel Cap. 2. Nella nostra implementazione abbiamo deciso di costruire tale struttura per la partizione di Voronoi utilizzando il metodo della decomposizione trapezoidale. Questo tipo di decomposizione consiste nel suddividere ulteriormente le regioni in piccoli trapezi su cui costruire la struttura di ricerca.



La figura superiore mostra un esempio di decomposizione trapezoidale di un

generico partizionamento, nulla cambia se si ha a che fare con un partizionamento di Voronoi.

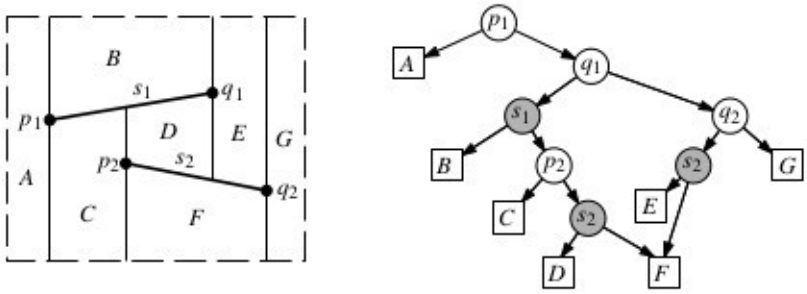
Come mostra la figura, l'ulteriore suddivisione è ottenuta creando per ogni vertice estensioni verticali fino ad incontrare un altro segmento o il bounding box.

Sia *T(S)* la decomposizione trapezoidale (detta anche mappa), con S l'insieme di segmenti, ci interessa costruire una struttura dati che può essere usata per rispondere alle domande di tipo geografico su *T(S)*. Queste domande sono del tipo "*In quale regione giace un determinato punto q*?".

La struttura, che chiameremo *search structure D*, è un albero binario con l'unica differenza che le foglie possono avere uno o più padri. Le foglie rappresentano univocamente i trapezi della mappa, cioè ad ogni foglia è associato un trapezio *t* appartenente a *T(S)*.

Altra particolarità della *search structure* sono i nodi interni. Essi possono essere di due tipi: *x-node* che memorizzano un end-point di un segmento in *S*; *y-node* che rappresentano i segmenti stessi. Per chiarezza: gli end-point corrispondono ai vertici del partizionamento e i segmenti corrispondono agli archi.

Una ricerca su un punto *q* comincia dalla radice dell'albero e procede per un solo ramo fino ad arrivare alla foglia del trapezio *t* che contiene *q*. Il ramo dell'albero è definito grazie alle proprietà geometriche del punto *q*. Infatti, durante il cammino, per ogni nodo interno in cui si giunge bisogna eseguire una domanda riguardante *q* per scegliere con quale figlio procedere. È chiaro che gli *x-node* abbiamo domande differenti rispetto agli *y-node* e infatti ad ogni *x-node* la domanda era del tipo "il punto *q* giace alla destra o alla sinistra della linea verticale passante attraverso l'end-point memorizzato in questo nodo?" e al contrario, la domanda di ogni *y-node* era del tipo "il punto *q* giace sopra o sotto al segmento memorizzato in questo nodo?".



La *search structure* e la mappa trapezoidale sono collegate tra loro: ogni trapezio *t* ha un puntatore ad una foglia della struttura e viceversa, ogni foglia della struttura di ricerca ha un puntatore al trapezio corrispondente.

L'algoritmo per la costruzione della *search structure* e la mappa trapezoidale da noi implementato è un algoritmo incrementale randomizzato. In altre parole le due strutture vengono costruite, passo dopo passo, aggiungendo di volta in volta un nuovo segmento. Tuttavia, l'ordine con il quale vengono selezionati i segmenti condiziona notevolmente sia la struttura di ricerca *D* sia l'esecuzione stessa dell'algoritmo e pertanto l'ordine è impostato casualmente cosicché si possano avere tempi di esecuzione soddisfacenti.

Non diamo una dimostrazione ne di quanto detto subito sopra ne del teorema che segue. A chi fosse interessato ad una dimostrazione completa rimandiamo a [12] di cui consigliamo la lettura.

Teorema 1 *L'algoritmo* TrapezoidalMap *calcola la mappa trapezoidale T(S) di un insieme di segmenti S e una search structure D per T(S) in tempo O(nlog). Lo spazio effettivamente occupato per memorizzare D è O(n) e per ogni domanda su un qualsiasi punto q si può rispondere in tempo O(logn).*

Qui di seguito viene mostrato lo pseudocodice dell'algoritmo TrapezoidalMap a cui si fa riferimento nel Teorema 1.

Algoritmo TrapezoidalMap(S)

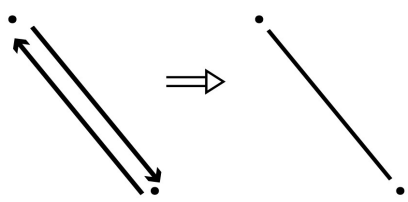
Input. Un insieme S di segmenti

Output. La mappa trapezoidale *T(S)* e la *search structure D*

1. Determina il bounding box e inizializza *T* e *D*
2. Calcola un ordine casuale dei segmenti di S
3. **for** $i \leftarrow 1$ **to** n:
4. **do** Trovare l'insieme $t_0 t_1 ... t_k$ trapezi di T che sono intersecati da s_i
5. rimuovere $t_0 t_1 ... t_k$ da *T* e rimpiazzarli con i nuovi trapezi che appaiono grazie all'inserimento di s_i
6. rimuovere le foglie in *D* che rappresentano $t_0 t_1 ... t_k$. Creare nuove foglie per i nuovi trapezi e collegarle adeguatamente all'interno della struttura.

L'algoritmo che è stato utilizzato nella nostra implementazione ricalca la struttura dello pseudocodice con l'unica differenza che in input, oltre all'insieme di segmenti, prende anche i vertici all'angolo del bounding box. Pertanto non c'è bisogno di ulteriori considerazioni rispetto al Teorema 1 per quanto riguarda la complessità spaziale e temporale.

Le uniche osservazioni necessarie riguardano la creazione dei dati in input da passare all'algoritmo. Ricordiamo che il diagramma di Voronoi è memorizzato all'interno di una DCEL e perciò è necessario un processo di traduzione affinché si possano passare gli input correttamente.



In pratica per ogni half-edge e il suo gemello viene creato un segmento, e per ogni vertice *v* del diagramma di Voronoi viene creato un end-point per i segmenti generati dagli half-edge che insistono *v*. Inoltre ogni segmento memorizza i *siti* che vengono separati dal segmento stesso. L'immagine seguente aiuta a capire la

traduzione degli oggetti.

5.4 Le Strade soggette al pagamento del pedaggio

Per estrapolare le informazioni sulle strade soggette al pagamento del pagamento, la nostra implementazione segue precisamente l'idea presentata nel Cap.2. L'insieme delle strade a pagamento è rappresentato attraverso un HashMap, nel quale ad ogni strada è associata una chiave di tipo String che corrisponde all'identificativo univoco con il quale è riconosciuta in *OpenStreetMap.*

In *OpenStreetMap* le strade a pagamento sono identificate da due *tag*: il tag *toll* e il tag *fee*. Nella nostra applicazione, allora, si controlla se almeno uno dei due tag esiste e, in caso affermativo, che esso abbia associato un valore diverso dalla negazione.

Quindi, per ogni nodo ottenuto dalla approssimazione si controlla che ogni strada su cui esso giace è contenuta o meno nell'insieme. Il metodo, alla fine, restituisce semplicemente un valore booleano.

6. Esperimenti

Qui di seguito discuteremo degli esperimenti compiuti e mostreremo alcuni risultati significativi con in quali sarà possibile fare delle considerazioni importanti per il nostro sistema.

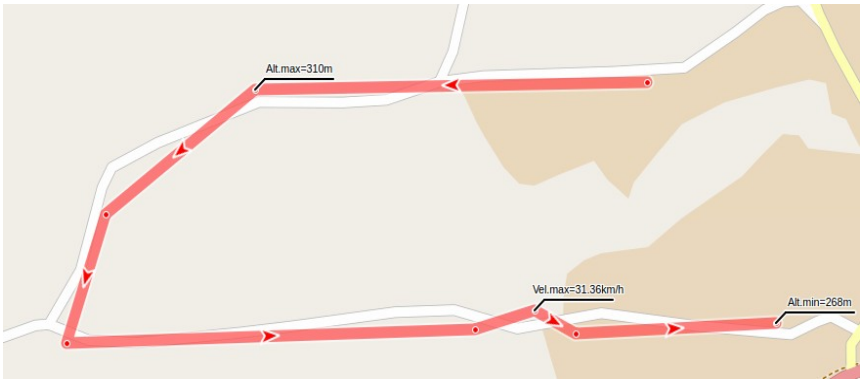
I dati forniti inizialmente all'algoritmo sono una mappa sulla quale compiere gli esperimenti e una traccia GPS dalla quale estrarre i punti che verranno poi approssimati. In particolare, le tracce fornite sono state costruite al computer ed esse sono state costruite in modo tale da presentare situazioni al limite cosi da mettere a dura prova il sistema.

Attraverso gli esperimenti vogliamo mostrare non solo i buoni risultati ottenuti, ma anche i limiti che il nostro sistema presenta. Dunque mostreremo i primi esperimenti in cui le mappe utilizzate sono molto semplici. Poi mostreremo esempi in cui mettiamo a dura prova il sistema: prima analizzando l'importanza di aggiungere nodi fittizi e successivamente mostrando la dipendenza, che il sistema presenta, dalle mappe che si utilizzano. Questi due aspetti sono molto collegati poiché in entrambi i casi bisogna manipolare le informazioni delle mappe. In particolare mostreremo che non sempre l'aggiunta di nodi fittizi aiuta a fornire i risultati corretti.

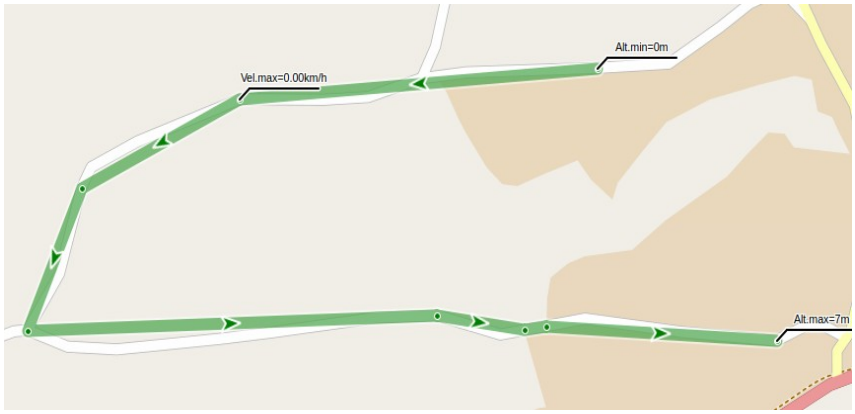
6.1 Primi esperimenti

I primi esperimenti che mostreremo non hanno considerazioni notevoli su cui discutere, ma servono a presentare il normale comportamento del sistema.

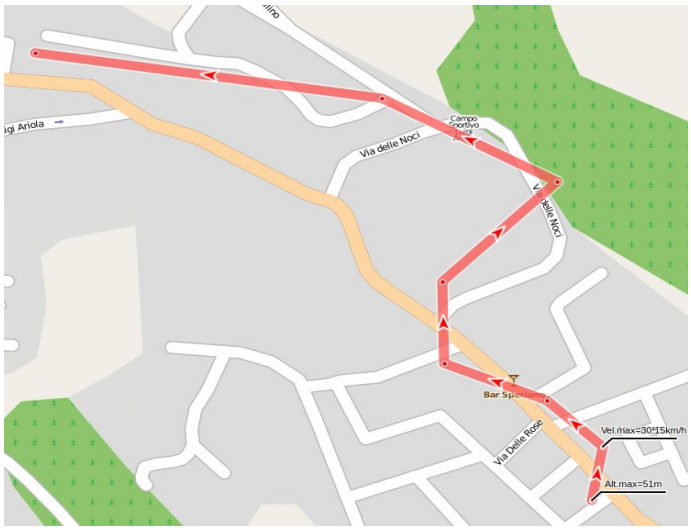
La figura seguente mostra un primo caso molto semplice.



Come si può vedere, la mappa su cui si è compiuto questo esperimento contiene strade di campagna. La mappa, dunque, presenta pochi nodi e perciò ci aspettiamo un risultato buono. Infatti, i risultati non ci deludono e la figura seguente ce lo mostra.



Nell'esempio seguente si vede come gli errori della traccia data in input siano eccessivi. Inoltre la mappa comincia a contenere un numero considerevole di punti ravvicinati.



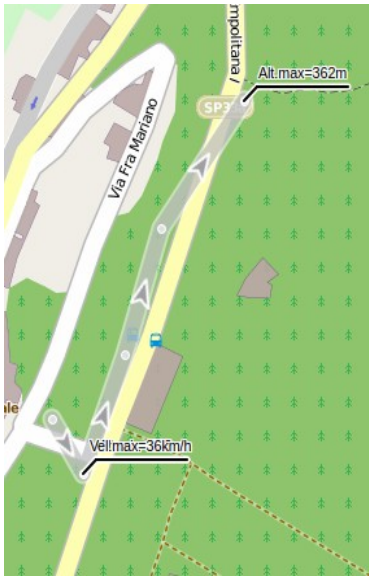
Ci aspettiamo in questo caso qualche imprecisione e, invece, il sistema si comporta adeguatamente.

In questi due esempi si possono vede come i punti delle tracce vengono approssimati

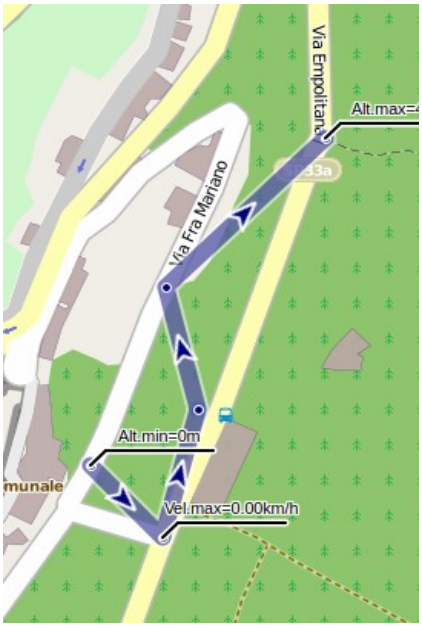
nel miglior modo possibile. I punti sulla mappa che sono stati trovati, facilitano notevolmente l'eventuale costruzione di *way* che corrisponde alla traccia.

6.2 Nodi Fittizi

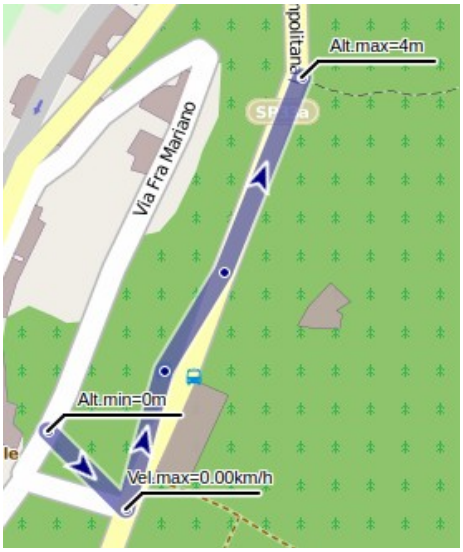
Abbiamo già accennato nei capitoli precedenti l'importanza di aggiungere nodi fittizi tra due nodi della stessa *way* che superano una determinata distanza. Molti degli esperimenti che abbiamo compiuto hanno dimostrato quanto abbiamo detto.



L'immagine precedente mostra la traccia, che è stata fornita all'inizio, come, nonostante non sia precisa, segui logicamente la strada. Ci troviamo in un caso in cui un punto della traccia si trova in mezzo ad un rettilineo. Questa situazione è l'applicazione reale dell'esempio citato nel Cap. 2, quando si introduceva il problema di inserire i nodi fittizi. Infatti, come ci aspettiamo, tale punto viene associato ad un nodo non corretto.



Allora, proviamo a risolvere la situazione aggiungendo i nodi fittizi. In effetti, una volta compiuta questa operazione, otteniamo risultati più che buoni e l'immagine seguente ce li mostra.



6.3 L'importanza delle Mappe

Abbiamo discusso quanto siano importanti l'inserimento dei nodi fittizi all'interno della mappa. Questa soluzione, però, non risolve tutti i problemi riguardanti le approssimazioni sbagliate. Le mappe, infatti, contengono altri tipi di nodi che non appartengono alle strade ma che vengono, comunque, considerati in fase di

costruzione del diagramma di Voronoi e della relativa struttura di ricerca.

Ciò di cui abbiamo bisogno è uno strumento esterno che ci permetta di filtrare le mappe in modo tale da mantenere solamente le strade ed eliminare tutti quei nodi in eccesso. In *OpenStreetMap*, infatti, i nodi e le *way* servono anche a rappresentare, ad esmepio, i confini di un parco, oppure linee ferroviarie o confini comunali. Questi punti possono, ovviamente, interferire con la corretta approssimazione.

Lo strumento che abbiamo utilizzato per il filtraggio delle mappe è *osmfilter* ed è fornito direttamente da *OpenStreetMap*.

L'immagine seguente mostra una traccia che è stata data in input al sistema. Notiamo che ci troviamo in un centro cittadino e il numero dei punti presi in considerazione è molto elevato.

Quindi abbiamo calcolato due risultati con due mappe differenti: la prima senza essere filtrata e la seconda, invece, contiene solamente le strade cittadine.

Il primo risultato è già molto buono, ma notiamo come il punto in basso a sinistra viene associato ad un confine e non alla strada come vorremmo che sia. Vogliamo precisare che questo non è un errore algoritmico, anzi, il nostro sistema si comporta nella maniera più corretta possibile associando tale punto al nodo più vicino sulla mappa.

Il secondo risultato, ottenuto processando una mappa filtrata, è ottimo come possiamo vedere nell'immagine che segue.

Vogliamo notare che in questo caso i punti intermedi non avrebbero aiutato: in tal caso, infatti, il punto in basso a sinistra sarebbe approssimato sempre al nodo appartenente al confine.

Tuttavia, il processo di filtraggio è delicato e non sempre migliora la situazione, soprattutto nel caso in cui si hanno strade molto ravvicinate.

Tali considerazioni riguardanti le mappe inducono ragionevolmente a pensare che anche la qualità delle mappe influenzi l'esito dell'esecuzione. Ad esempio, potremmo avere approssimazioni inconsistenti nel caso in cui si utilizzi una mappa incompleta.

Ore presentiamo un esempio i questo problema.

L'immagine superiore mostra due punti della traccia che non corrispondono a nessuna strada sulla mappa, ma che realtà esiste. In effetti l'approssimazione presente errori notevoli.

6.4 “Snellire” le Tracce

Risulta chiaro come il sistema dipenda moltissimo dai dati che manipola. Molti nodi sulla mappa possono creare problemi come quelli che abbiamo visto nel paragrafo precedente. Inoltre, non è assurdo pensare che un numero notevole di nodi possa portare problemi di tipo matematico. Affermare ciò non è del tutto corretto, visto che si tratta di problemi causati dalla precisione di macchina e non prettamente matematici. Infatti, in questo sistema si ha a che fare con valori a molte cifre decimali e le operazioni possono risultare non del tutto corrette. Inoltre, i nodi possono essere molto ravvicinati e calcolarne il diagramma di Voronoi preciso può risultare difficile.

Per questo è facile intuire che meno dati vengono manipolati e meglio risulta l'approssimazione dei punti della traccia.

Abbiamo già parlato di filtrare la mappa quindi parliamo dell'importanza di filtrare le tracce GPS. Sappiamo che le tracce GPS non contano nulla nella costruzione del Diagramma di Voronoi e nella struttura di ricerca, ma una diminuzione dei punti della traccia può portare benefici e ridurre la possibilità che avvengano errori.

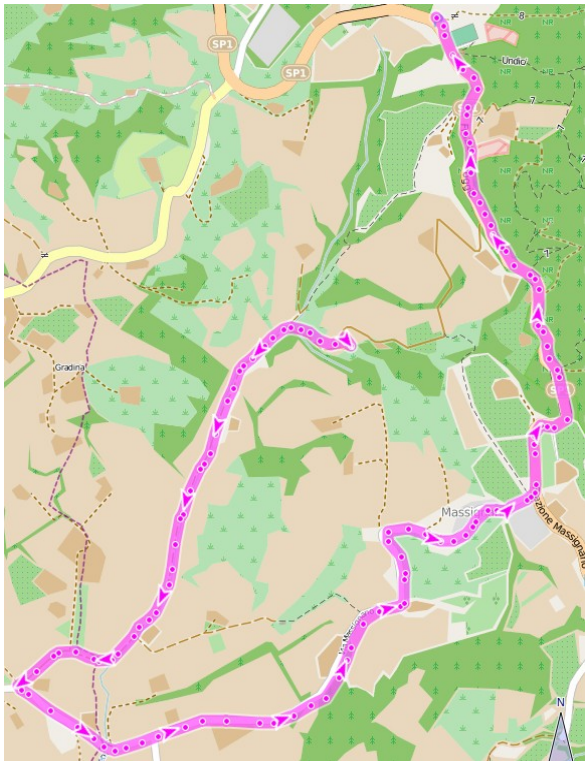
“Snellire” le tracce significa proprio questo, cioè dalla traccia originale vengono tolti molti nodi inutili ottenendo, così, un traccia costituita da soli nodi necessari.

Un primo vantaggio è la quantità di dati da processare e in questo modo si può ridurre il tempo di esecuzione. Immaginate, ora, di avere una traccia con tanti nodi e un gran numero di essi risultato essere poco precisi. Questi punti possono causare errori simili a quelli discussi quando abbiamo introdotto l'importanza dei punti fittizi. Il secondo vantaggio quindi viene da sé: eliminare molti nodi in eccesso riduce ragionevolmente questo problema.

Tali ragionamenti sulle tracce GPS sono validi perché l'eliminazione dei punti non necessari non condiziona gli algoritmi di routing ne l'estrazione le informazioni riguardanti le strade.

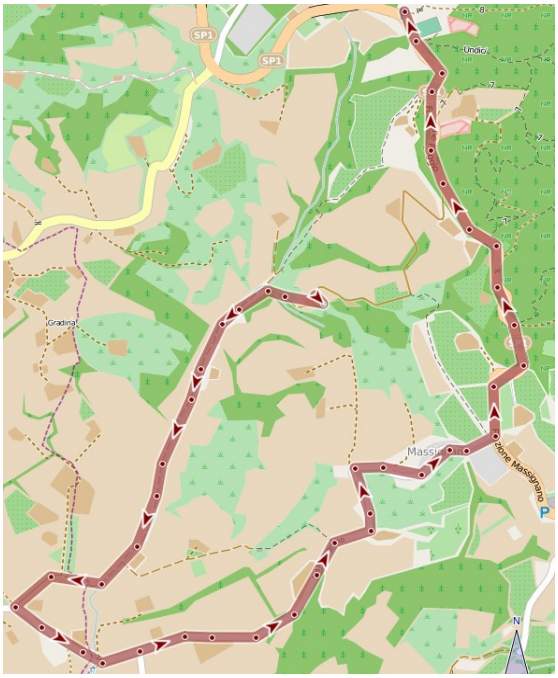
Nei nostri esperimenti abbiamo complicato la situazione cominciando a processare tracce reali le quali erano composte da moltissimi punti. Abbiamo, quindi, ridotto tale numero e poi avviato l'operazione di approssimazione. La riduzione è stata compiuta attraverso un script fornito direttamente da *OpenStreetMap* descritto nella wiki.

Nell'esempio seguente possiamo vedere il numero elevato di nodi che compongono la traccia passata in input.

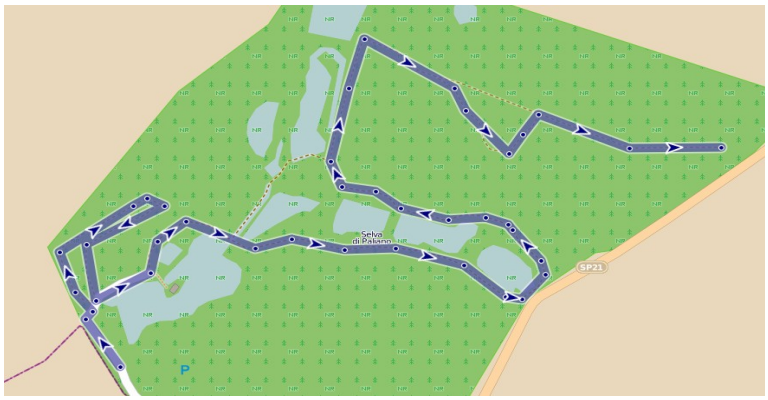


Una volta eliminati i nodi in eccesso e avviato il sistema, il risultato ottenuto è ottimo e l'immagine seguente lo mostra.

È facile notare ciò che abbiamo detto sopra: anche con un numero minore di nodi si può accedere correttamente alle strade interessate.

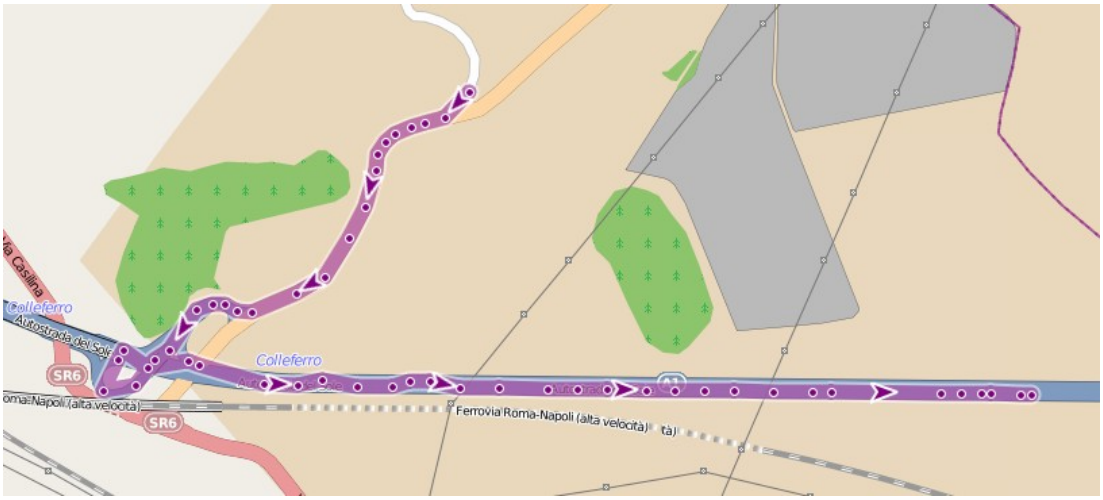


Allo stesso modo il seguente esempio mostra una soluzione ottenuta filtrando la traccia GPS.



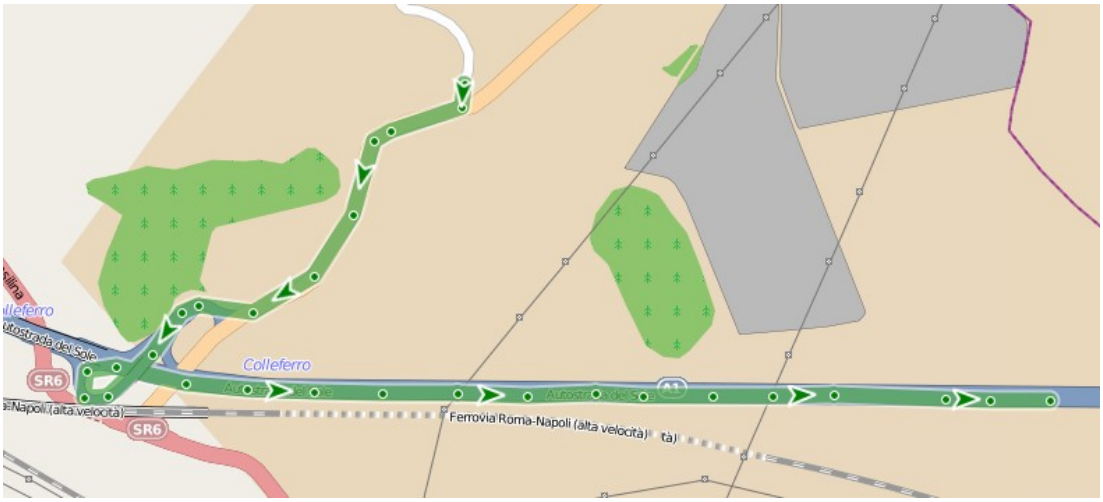
Con il seguente esempio, invece, vogliamo mostrare come avere molti nodi e poco precisi possa creare approssimazioni errate. Infatti qui di seguito mostreremo prima il risultato dell'esecuzione ottenuto dando in input una traccia completa e successivamente mostriamo il risultato ottenuto da una traccia dal numero dei nodi ridotti.

L'immagine superiore mostra la traccia passata in input e quella seguente il relativo risultato ottenuto.



È facile notare gli errori di approssimazione. Questo caso sottolinea quanto sono inutili e a volte deleteri i nodi in eccesso.

L'immagine seguente mostra, invece, la traccia filtrata. Notiamo che i nodi sul rettilineo sono quasi tutti equidistanti e che i nodi che causano problemi sono spariti.



Ci aspettiamo allora che il risultato sia soddisfacente e l'immagine seguente mostra ciò che vogliamo.

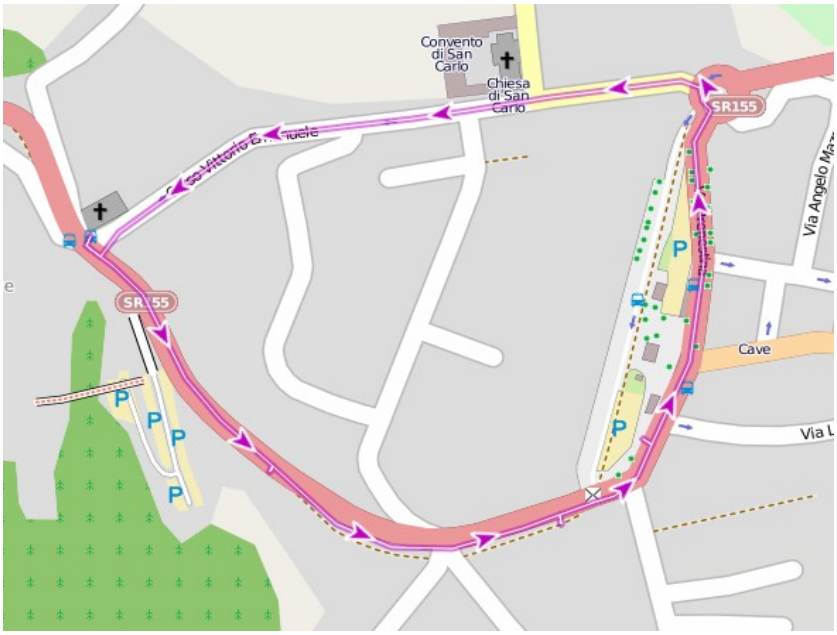
6.4 Individuazione delle way

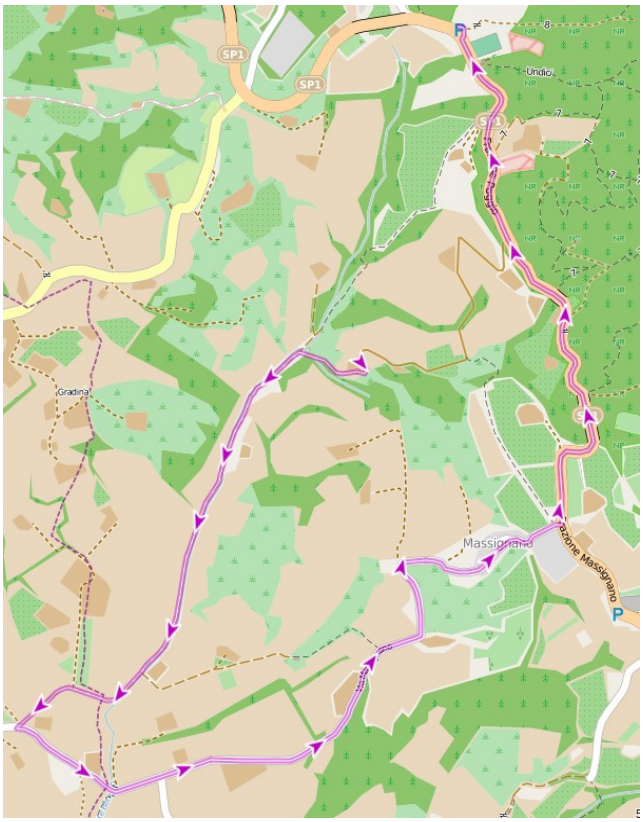
Per quanto riguarda l'individuazione delle *way* abbiamo utilizzato *QlandkarteGT*. Come già abbiamo detto precedentemente, questo software è un Open Source GIS che include anche questa funzionalità, cioè permette la costruzione delle *way* partendo da una traccia di punti GPS.

QlandkarteGT permette la costruzione di *way* più veloci o più corte se percorse in automobile o bicicletta, percorribili a piedi. Ne abbiamo fatto un largo uso di quasi tutte le opzione poiché si ha a che fare con tracce che tocca strade con pedaggio a pagamento che non sono percorribili da pedoni.

Qui di seguito mostreremo qualche esempio di *way* ottenuti dalle nostre approssimazioni. Tuttavia, in alcuni esempi, abbiamo riscontrato problemi, poiché, molto problabilmente, il server utilizzato da *QlandkarteGT* non contiene tutte le informazioni necessarie per la costruzione.

Nelle prime due immagini che seguono vediamo come viene trovata bene la *way*.





Nell'esempio seguente invece vogliamo mostrare il caso in cui ci possono essere molti punti imprecisi e la costruzione della *way* potrebbe avere dei problemi.

Ricordiamo che l'esempio in questione è stato presentato nel paragrafo in cui si mostrava la necessità di “snellire” la traccia. Infatti, una volta effettuata tale operazione possiamo notare che la *way* ottenuta risulta decisamente migliore. L'immagine seguente evidenzia, appunto, tale risultato.

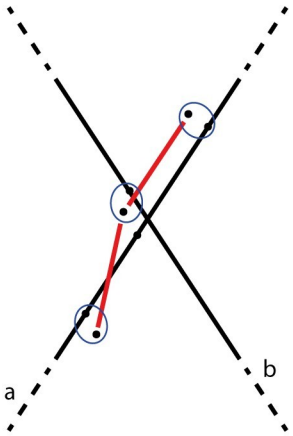
6.6 Informazioni sulle Strade

Per quanto riguarda l'estrazione delle informazioni riguardanti le strade non sono stati compiuti esperimenti differenti,bensì sono stati utilizzati gli stessi di cui si è già discusso. Come mostrano le immagini, solamente nell'ultimo caso si ha avuto una risposta affermativa, poiché viene percorso un tratto di autostrada.

La semplicità dell'applicazione non richiede molta attenzione, poiché si tratta di leggere le informazioni contenute nella mappa e ciò sottolinea la dipendenza dalla qualità della stessa.

7. Conclusioni

Alla luce dei risultati ottenuti dagli esperimenti, notiamo come questo modo di affrontare il problema sia dipendente per gran parte dalla mappa da cui si acquisiscono le informazioni.



Come mostra l'immagine, la traccia *tr*, segue la strada *a;* potrebbe capitare, come nella figura, che la strada *a* passi sopra un ponte e sotto il ponte passi un'altra strada, *b* nell'esempio. Un punto di *tr* potrebbe essere quindi associato alla strada *b*, creando cosi degli inconvenienti che non dipendono dall'algoritmo.

In questo caso è necessaria una rifinitura della traccia, ad esempio, riducendo il numero dei nodi da cui è composta, così come abbiamo parlato nel capitolo precedente.

Un'altra situazione in cui ci si può trovare è che la mappa su cui si lavora presenta degli errori e delle imprecisioni, ma in questo caso non ci si può fare niente.

Dai risultati è emerso come il nostro sistema dipenda dalla quantità di dati che vengono manipolati. Tuttavia, i problemi causati dalla precisione di macchina non sono del tutto risolvibili grazie a mappe filtrate o tracce semplificate. È chiaro come questo problema potrebbe essere ovviato prendendo delle precauzioni implementative, utilizzando variabili con maggiore precisione(come per esempio nel linguaggio java possiamo utilizzare il *BigDecimal*), oppure utilizzando altri linguaggi di programmazione che permettono di manipolare le variabili in modo migliore.

Risulta chiaro che, effettuate le opportune considerazioni, mappare le tracce GPS utilizzando le informazioni contenute nelle mappe è un metodo che offre molte opportunità di applicazione.

Applicazioni Possibili

Una diretta applicazione è l'individuazione della *way* sulla mappa (la definizione di *way* è riportata nel Cap. 2).

Una volta creato il Grafo dalla mappa e ottenuti tutti i punti p_{mi} sulla mappa che approssimano i punti p_{tj} della traccia, è possibile ottenere moltissime informazioni riguardanti i cammini che descrivono al meglio la traccia stessa. Si può implementare l'algoritmo di Dijkstra per trovare il cammino più breve, oppure utilizzando la ricerca in ampiezza, qualora non interessino informazioni particolari. Utilizzando l'algoritmo di Bellman-Ford si può trovare il cammino tenendo conto del dislivello tra i punti.

Formalmente possiamo dire che sia $tm = \{ v_1, v_2 ... v_n\}$ un insieme di punti sulla mappa trovati approssimando la traccia $tr = \{p_1, p_2 ... p_n\}$. La *way* finale non è niente altro che la concatenazione dei cammini, trovati attraverso algoritmi di routing, che uniscono i punti di *tm*. In altre parole sia c_{ij} il cammino che unisce i punti p_{mi} e p_{mj} di *tm* con i=1,2,...,n-1 e j = 2,3,...,n , la *way* è definita come $way = \{c_{12}, c_{23}, ..., c_{n.1n}\}$.

Il metodo da noi studiato potrebbe essere sfruttato per fini turistico/commerciali, come accennato nel Cap.1; studiare le tracce, e quindi i percorsi più popolari, può favorire la nascita di attività commerciali in zone maggiormente redditizie.

Altre applicazioni potrebbero essere relative all'ambito della mobilità urbana: con uno studio sulle strade più trafficate, si può ipotizzare l'eventualità di un incidente, e le ripercussioni che questo può avere sul traffico (relative certo, anche all'entità dello stesso). Un'agenzia di trasporti pubblici potrebbe utilizzare quindi, i dati raccolti, per perfezionare il servizio offerto, studiando per esempio percorsi alternativi nei casi di emergenza, favorendone il normale svolgimento.

8. Riferimenti

[1] http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html

[2] http://wiki.openstreetmap.org/wiki/Main_Page

[3] http://code.google.com/p/osm-parser/

[4] http://williams.best.vwh.net/avform.htm#flat

[5] https://josm.openstreetmap.de/browser/josm/trunk/src/org/openstreetmap/josm/data/projection/Ellipsoid.java?rev=4382

[6] http://www.qlandkarte.org/

[7] Z. Chen, H. T. Shen, X. Zhou, *Discovering Popular Routes from Trajectories*, School of Information Technology & Electrical Engineering , The University of Queensland, Australia

[8] L.-Y. Wei, Y. Zheng, W.-C. Peng, *Constructing Popular Routes from Uncertain Trajectories*, Pechino, China, 2012

[9] L. Cao, J. Krumm , *From GPS Traces to a Routable Road Map*, Seattle, USA, 2009

[10] J.-G. Lee, J. Han, K.-Y. Whang, *Trajectory Clustering: A Partition-and-Group Framework*, Pechino, China, 2009

[11] J.-G. Lee, J. Han, X. Li, H. Gonzalez, *TraClass: Trajectory Classification Using Hierarchical Region-Based and Trajectory-Based Clustering*, Auckland, Nuova Zelanda, 2008

[12]L. Spignoli, *Geolocalizzazione di Tracce GPS: localizzazione di punti*, Roma 2013 (Lavoro di Tesi)

9. Bibliografia

M. de Berg,O. Cheong, M. van Kreveld, M. Overmars, *Computational Geometry, Algorithms and Applications Third Edition*, Berlin Heidelberg 2008

A. McLean, *Voronoi Diagrams of Music*, 2006

B. Aronov, *A lower bound on Voronoi diagram complexity*, Brooklyn, NY, USA, 2002

I. Selimi, *Analisi ed Implementazione in ambiente web di un sistema per Voronoi Game*,Perugia, 2003

F. Aurenhammer, R. Klein, *Voronoi Diagrams*, In J. Sack, G. Urrutia, *Handbook of Computational Geometry*, Graz 2000, pp 201-290

K. Keating, *Line Segment Intersection Using a Sweep Line*, Tufts University, 2005

F. Malucelli, G*eometria Computazionale*, Milano 2005 (http://home.deib.polimi.it/malucell/didattica/PAA/materialepaa/geocomp.pdf)

I. Cervesato, *Elementi di Matematica*, Milano, pp 18-26

D. De Sanctis, *Voronoi Diagram*, Roma (http://twiki.di.uniroma1.it/twiki/view --DIAGRAMMIVORONOI.odt)

T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Cambridge, Massachusetts, 2002

L. Spignoli, *Geolocalizzazione di Tracce GPS: localizzazione di punti*, Roma 2013 (Lavoro di Tesi)

http://www.openstreetmap.org/

http://www.geogebra.org/cms/it/

http://www.topografix.com/gpx.asp 8

Appendice A

Il codice che abbiamo sviluppato è reperibile all'indirizzo https://github.com/chemickypes/searchgpspath

Oltre al file eseguibile è disponibile il codice sorgente e la licenza che copre il codice. In particolare, si tratta di una licenza GPLv3.

Modalità di utilizzo

Di seguito mostriamo il comando per avviare il sistema. Il comando va eseguito da linea di comando.

```
# java -jar searchgpspath.jar [map] [track] [distance or null]
```

- [map] è il percorso al file che contiene la mappa e deve essere un file di tipo OSM.
- [track] è il percorso al file che contiene la traccia e deve essere un file di tipo GPX.
- [distance or null] sta a identificare la distanza limite sotto la quale vogliamo costruire i nodi fittizi. Bisogna passargli un valore o intero o nullo (quindi non viene passato alcun valore). Nel caso in cui non viene passato alcun valore, il sistema interpreta la scelta come se non c'è la richiesta di creare punti fittizi.

```
# java -jar searchgpspath.jar –help [or -h]
```

Il comando superiore serve per mostrare le semplici opzioni del sistema.

Ringraziamenti

Al Professor Gianluca Rossi, per la gentilezza e la pazienza dimostratami

All'amico e collega Lorenzo Spignoli, con il quale ho condiviso gioie e dolori di questa tesi

A mio padre e mia madre, che mi hanno trasmesso il valore della curiosità

A mia sorella, Agnese, per il suo aiuto costante in ogni cosa

A mia zia, Marietta, e al suo affetto incondizionato

A Sara, che mi supporta e sopporta ogni giorno

Agli amici e alle serate passate insieme

Alla Grammatica Italiana, per tutti i bocconi amari mandati giù durante la stesura della tesi

"Mille volte al giorno penso al fatto che la mia vita interiore ed esteriore si basa sul lavoro di altri uomini, vivi o morti, e che io devo sforzarmi a offrire in misura eguale a quanto ho ricevuto e tuttora ricevo."

A. Einstein