Geometria computazionale

Livio Colussi

17 maggio 2016

1 Introduzione

La Geometria Computazionale studia gli algoritmi per risolvere problemi geometrici. Essa trova applicazione in molti campi quali ad esempio la grafica computerizzata, la progettazione CAD, la robotica, l'intelligenza artificiale, la statistica, ecc.

L'input di un problema di geometria computazionale è di norma un insieme di oggetti geometrici: punti, segmenti, poligoni, ecc. L'output può essere una risposta a domande sugli oggetti in input; ad esempio se due segmenti si intersecano oppure se un dato punto è interno o esterno ad un poligono. Oppure l'output può essere un nuovo oggetto geometrico costruito a partire da quelli in input; ad esempio l'involucro convesso di un insieme di punti (che è il più piccolo poligono convesso che li contiene tutti).

Noi ci limiteremo a considerare oggetti geometrici piani ma le tecniche algoritmiche che studieremo saranno abbastanza generali da poter essere applicate anche al caso di oggetti geometrici in tre o più dimensioni.

2 Rappresentazione degli oggetti geometrici

Abbiamo detto che tratteremo soltanto oggetti geometrici piani. Dobbiamo intanto decidere come rappresentiamo tali oggetti nella memoria del computer. Cominciamo con i punti: un punto p verrà rappresentato con una coppia di coordinate (x_p, y_p). Dobbiamo però scegliere se usare per le coordinate la rappresentazione in virgola mobile oppure limitarci a considerare soltanto coordinate intere.

Usando coordinate in virgola mobile anche il semplice test se due punti p e q coincidono può dare risultati aleatori a causa degli errori di troncamento.

Scegliamo quindi di usare coordinate intere. Questo implica una limitazione al valore massimo CMAX delle coordinate x ed y i cui valori dovranno soddisfare la disuguaglianza

$$-\text{CMAX} \leq x, y \leq \text{CMAX}$$

Chiaramente CMAX deve essere minore del massimo intero rappresentabile MAXINT. Ma, come vedremo fra poco, noi vogliamo anche poter calcolare delle aree orientate (con segno) senza provocare errori di overflow. Dunque l'area A di ogni figura del piano deve risultare compresa tra $-$MAXINT e MAXINT. Siccome l'area massima è $4\,\text{CMAX}^2$ bisogna che $4\,\text{CMAX}^2 \leq \text{MAXINT}$ e quindi dovremo scegliere

$$\text{CMAX} \leq \sqrt{\text{MAXINT}}/2$$

Ad esempio, se MAXINT $\approx 2^{32}$ allora CMAX $\approx 2^{15}$ per cui gli assi x ed y contengono ciascuno 2^{16} punti; un risoluzione più che sufficiente nella maggior parte delle applicazioni.

Oltre a questo dobbiamo limitare l'insieme delle operazioni sulle coordinate soltanto a quelle che si possono eseguire in modo esatto. Dobbiamo perciò evitare sia la divisione sia l'uso delle funzioni trigonometriche, delle radici quadrate e dei logaritmi.

Le uniche operazioni permesse rimangono quindi somma, sottrazione, moltiplicazione e confronti; in altre parole dobbiamo lavorare con una mano legata dietro la schiena e per di più la destra.

Una volta deciso come rappresentiamo i punti la rappresentazione degli altri oggetti segue di conseguenza. Un segmento $\overline{pq}$ verrà rappresentato con la coppia di punti (p, q) che verrà considerata come coppia ordinata nel caso di segmenti orientati $\overrightarrow{pq}$. I poligoni verranno rappresentati con la sequenza $p_0, p_1, \ldots, p_{n-1}$ dei loro vertici presi in ordine antiorario sul perimetro. Un cerchio sarà rappresentato dalle coordinate del centro e dal raggio, ecc.

Rappresenteremo il vettore v di coordinate (v_x, v_y) con uno qualsiasi dei segmenti orientati $\overrightarrow{pq}$ tali che $v_x = x_q - x_p$ e $v_y = y_q - y_p$.

Talvolta indicheremo con lo stesso simbolo p sia il punto di coordinate (x_p, y_p) che il vettore di coordinate (x_p, y_p) rappresentato dal segmento orientato $\overrightarrow{op}$ (dove o è l'origine degli assi); dal contesto sarà sempre chiaro quando intendiamo riferirci al punto o al vettore.

3 I primi semplici problemi geometrici

Affronteremo ora i seguenti tre semplici problemi geometrici:

1. Dati due segmenti orientati $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_0p_2}$ aventi comune origine p_0 il segmento $\overrightarrow{p_0p_2}$ è ruotato in senso orario o antiorario rispetto al segmento $\overrightarrow{p_0p_1}$?

2. Se ci muoviamo lungo due segmenti orientati consecutivi $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_1p_2}$ partendo da p_0 verso p_2 quando siamo in p_1 si svolta a sinistra o a destra?

3. I due segmenti $\overline{p_1p_2}$ e $\overline{p_3p_4}$ si intersecano?

Dobbiamo risolvere questi problemi in tempo costante $O(1)$ usando soltanto le operazioni permesse. Ad esempio non possiamo risolvere il terzo problema calcolando dapprima le equazioni delle rette che contengono i due segmenti e quindi il loro punto di intersezione perché per trovare il punto di intersezione ci serve la divisione.

Useremo invece una operazione, il *prodotto vettoriale*, che si può esprimere soltanto in termini di somme, differenze e prodotti e che risulterà molto utile anche per risolvere molti altri problemi geometrici.

Dati due vettori v_1 e v_2 di coordinate (x_1, y_1) e (x_2, y_2) il loro prodotto vettoriale, che viene indicato con $v_1 \times v_2$, è definito come l'area orientata del parallelogramma di vertici $o \equiv (0, 0)$, $p_1 \equiv (x_1, y_1)$, $p_2 \equiv (x_2, y_2)$ e $q \equiv (x_1 + x_2, y_1 + y_2)$ (vedi Figura 1).

L'area è orientata nel senso che la si prende con segno positivo se l'angolo da v_1 a v_2 è orientato in senso antiorario con segno negativo altrimenti.

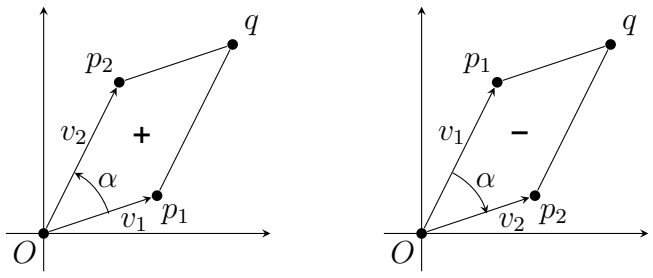


Figura 1: L'area orientata del parallelogramma individuato dai due vettori v_1 e v_2.

Una definizione operativa più utile è la seguente:

$$v_1 \times v_2 = \det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} = x_1 y_2 - x_2 y_1$$

l'Esercizio 2 chiede di dimostrare che le due definizioni sono equivalenti.

Vi è anche un'altra definizione equivalente

$$v_1 \times v_2 = |v_1||v_2| \sin \alpha$$

dove $|v_1|$ e $|v_2|$ sono le lunghezze dei due vettori e α è l'angolo orientato in senso antiorario da v_1 a v_2 ma non possiamo usarla perchè serve la funzione seno e la radice quadrata (per calcolare la lunghezza dei vettori).

Possiamo usare il prodotto vettoriale per risolvere il primo dei tre problemi: determinare se $\overrightarrow{p_0p_2}$ è ruotato in senso orario o antiorario rispetto a $\overrightarrow{p_0p_1}$.

Indichiamo con $p_1 - p_0$ il vettore di coordinate $(x_1 - x_0, y_1 - y_0)$ e analogamente con $p_2 - p_0$ il vettore di coordinate $(x_2 - x_0, y_2 - y_0)$. Calcoliamo quindi il prodotto vettoriale

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

Se il risultato è positivo $\overrightarrow{p_0p_2}$ è ruotato in senso antiorario rispetto a $\overrightarrow{p_0p_1}$, se è negativo $\overrightarrow{p_0p_2}$ è ruotato in senso orario. Il risultato può essere nullo soltanto se uno o entrambi i segmenti sono degeneri (p_1=p_2 o p_3=p_4) oppure i due segmenti sono collineari.

Il Programma 1 è lo pseudocodice di questo test.

```
ANGLE-LEFT(p0, p1, p2)    // p0, p1 e p2 punti di coordinate (x0, y0), (x1, y1) e (x2, y2).
1  d = (x1 − x0)(y2 − y0) − (x2 − x0)(y1 − y0)
   // Se d > 0 l'angolo tra p0p1 e p0p2 è orientato in senso antiorario.
   // Se d = 0 i due segmenti sono collineari (anche uno o entrambi degeneri).
   // Se d < 0 l'angolo è orientato in senso orario.
2  return d
```

Programma 1: Il test se $\overrightarrow{p_0p_2}$ è ruotato in senso antiorario rispetto a $\overrightarrow{p_0p_1}$.

In modo analogo possiamo risolvere anche il secondo problema: Percorrendo i due segmenti consecutivi $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_1p_2}$ quando ci si trova nel punto p_1 si svolta a sinistra o a destra?

```
TURN-LEFT(p_0, p_1, p_2)
1  d = ANGLE-LEFT(p_0, p_1, p_2)
   // Se d > 0 in p_1 si svolta a sinistra.
   // Se d = 0 in p_1 si prosegue nella stessa direzione o in direzione opposta.
   // Se d < 0 in p_1 si svolta a destra.
2  return d
```

Programma 2: Il test se percorrendo i due segmenti consecutivi $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_1p_2}$ in p_1 si svolta a sinistra.

La risposta è che si svolta a sinistra se il vettore $\overrightarrow{p_0p_1}$ è ruotato in senso antiorario rispetto al vettore $\overrightarrow{p_0p_2}$. Possiamo quindi usare il test ANGLE-LEFT. Il Programma 2 è lo pseudocodice del test di svolta a sinistra.

Rimane da risolvere il terzo problema: Controllare se due segmenti $\overline{p_1p_2}$ e $\overline{p_3p_4}$ si intersecano.

Dobbiamo distinguere due casi:

1. Se i due segmenti sono collineari $\overline{p_3p_4}$ interseca $\overline{p_1p_2}$ se e solo se almeno uno dei due estremi p_3 e p_4 appartiene al segmento $\overline{p_1p_2}$ oppure il segmento $\overline{p_1p_2}$ è tutto contenuto nel segmento $\overline{p_3p_4}$.

2. Se i due segmenti non sono collineari $\overline{p_3p_4}$ interseca $\overline{p_1p_2}$ se e solo se il segmento $\overline{p_3p_4}$ interseca la retta contenente il segmento $\overline{p_1p_2}$ e il segmento $\overline{p_1p_2}$ interseca la retta contenente il segmento $\overline{p_3p_4}$.

Per controllare se i due segmenti sono collineari calcoliamo i due prodotti vettoriali $d_1 = (p_1 - p_3) \times (p_4 - p_3)$ e $d_2 = (p_2 - p_3) \times (p_4 - p_3)$ per controllare da che parte stanno i punti p_1 e p_2 rispetto al segmento orientato $\overrightarrow{p_3p_4}$.

Se $d_1 = d_2 = 0$ ci sono due possibilità: i due segmenti sono collineari (e lo sono anche se entrambi i segmenti sono degeneri) oppure $\overline{p_3p_4}$ è degenere ma $\overline{p_1p_2}$ non lo è. Per escludere la seconda possibilità calcoliamo anche i prodotti vettoriali $d_3 = (p_3 - p_1) \times (p_2 - p_1)$ e $d_4 = (p_4 - p_1) \times (p_2 - p_1)$ per controllare da che parte stanno i punti p_3 e p_4 rispetto al segmento orientato $\overrightarrow{p_1p_2}$.

Se anche $d_3 = d_4 = 0$ i due segmenti sono sicuramente collineari. Naturalmente, se almeno uno dei valori d_1, d_2, d_3 e d_4 è diverso da zero i due segmenti non sono collineari.

Supponiamo quindi che il controllo di collinearità ci dica che i due segmenti sono collineari, ossia che siamo nel primo caso. Dobbiamo controllare se almeno uno dei punti p_3 e p_4 appartiene al segmento $\overline{p_1p_2}$.

Per controllare se p_3 appartiene al segmento $\overline{p_1p_2}$ facciamo riferimento alle coordinate x e controlliamo se x_3 è compreso tra x_1 e x_2 e questo si può fare calcolando il prodotto $(x_3 - x_1)(x_3 - x_2)$. Se tale prodotto è negativo x_1 e x_2 stanno da parti opposte rispetto a x_3 e quindi x_3 è strettamente compreso tra x_1 e x_2, se il prodotto è nullo x_3 è uguale a x_1 o a x_2 o a entrambi e infine se il prodotto è positivo x_3 non è compreso tra x_1 e x_2.

Dunque se tale prodotto è negativo possiamo concludere che il punto p_3 è interno al segmento $\overline{p_1p_2}$, se è positivo possiamo concludere che il punto p_3 è esterno mentre se tale prodotto è nullo o p_3 coincide con p_1 e/o p_2 oppure i tre punti p_1, p_2 e p_3 sono allineati sulla stessa retta verticale. Quindi, se il prodotto è nullo dobbiamo fare anche il controllo sulle coordinate y calcolando il prodotto $(y_3 - y_1)(y_3 - y_2)$. Se il prodotto è negativo possiamo

concludere che p_3 è interno, se è positivo è esterno e se anche questo prodotto è nullo p_3 coincide con p_1 e/o p_2 e quindi p_3 appartiene al segmento $\overline{p_1p_2}$.

Naturalmente se p_3 risulta esterno dobbiamo fare lo stesso controllo anche per p_4 e se anche p_4 risulta esterno ci basta controllare se p_1 è interno al segmento $\overline{p_3p_4}$ per verificare il caso in cui $\overline{p_1p_2}$ è tutto contenuto in $\overline{p_3p_4}$.

Supponiamo infine che il controllo di collinearità dia risultato negativo e quindi non tutti e quattro i punti stiano sulla stessa retta. In questo caso possiamo usare i prodotti vettoriali d_1 e d_2 per controllare se p_1 e p_2 stanno da parti opposte rispetto alla retta passante per $\overline{p_3p_4}$ e d_3 e d_4 per controllare se p_3 e p_4 stanno da parti opposte rispetto alla retta passante per $\overline{p_1p_2}$.

Se d_1 e d_2 hanno lo stesso segno il segmento $\overline{p_1p_2}$ non interseca la retta di $\overline{p_3p_4}$ e quindi i due segmenti non si intersecano.

Se d_1 e d_2 hanno segno opposto p_1 e p_2 stanno da parti opposte rispetto alla retta passante per $\overline{p_3p_4}$ e quindi il segmento $\overline{p_1p_2}$ interseca internamente (non in uno dei due estremi p_1 o p_2) la retta di $\overline{p_3p_4}$.

Se d_1 o d_2 sono nulli allora o il segmento $\overline{p_3p_4}$ è degenere oppure p_1 o p_2 stanno sulla retta di $\overline{p_3p_4}$ ma non entrambi in quanto i quattro punti non sono allineati.

La stessa cosa vale per d_3 e d_4. Se d_1 e d_2 hanno lo stesso segno il segmento $\overline{p_3p_4}$ non interseca la retta di $\overline{p_1p_2}$ e quindi i due segmenti non si intersecano.

Se d_3 e d_4 hanno segno opposto p_3 e p_4 stanno da parti opposte rispetto alla retta passante per $\overline{p_1p_2}$ e quindi il segmento $\overline{p_3p_4}$ interseca internamente la retta di $\overline{p_1p_2}$.

Se d_3 o d_4 sono nulli allora o il segmento $\overline{p_1p_2}$ è degenere oppure p_3 o p_4 stanno sulla retta di $\overline{p_1p_2}$ ma non entrambi in quanto i quattro punti non sono allineati.

L'unico caso di incertezza che rimane è se entrambi i segmenti sono degeneri, ma siccome i quattro vertici non sono collineari i due segmenti non possono essere entrambi degeneri e quindi questo caso non si può presentare.

Lo pseudo codice dell'algoritmo di decisione è riportato nel Programma 3.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)    // Controlla se p1p2 e p3p4 si intersecano.
1  d1 = ANGLE-LEFT(p3, p4, p1)
2  d2 = ANGLE-LEFT(p3, p4, p2)
3  d3 = ANGLE-LEFT(p1, p2, p3)
4  d4 = ANGLE-LEFT(p1, p2, p4)
5  if d1 == 0 and d2 == 0 and d3 == 0 and d4 == 0
6       // I due segmenti sono collineari
7       return ((x2 − x3)(x1 − x3) ≤ 0 and (y2 − y3)(y1 − y3) ≤ 0) or
                ((x2 − x4)(x1 − x4) ≤ 0 and (y2 − y4)(y1 − y4) ≤ 0) or
                ((x4 − x1)(x3 − x1) ≤ 0 and (y4 − y1)(y3 − y1) ≤ 0)
8  else // I due segmenti non sono collineari
9       return ((d1 ≤ 0 and d2 ≥ 0) or (d1 ≥ 0 and d2 ≤ 0)) and
                ((d3 ≤ 0 and d4 ≥ 0) or (d3 ≥ 0 and d4 ≤ 0))
```

Programma 3: Il test se due segmenti si intersecano.

Esercizio 1 Dimostrare che l'area orientata del triangolo di vertici p_0, p_1, p_2 è data dalla formula

$$A = \frac{1}{2}[(x_0 - x_1)(y_0 + y_1) + (x_1 - x_2)(y_1 + y_2) + (x_2 - x_0)(y_2 + y_0)]$$

Verificare che A è positiva se i vertici p_0, p_1, p_2 sono presi nel verso antiorario ed è negativa se vengono presi nel verso orario.

Esercizio 2 Usare il risultato dell'Esercizio 1 per dimostrare che le definizioni di prodotto vettoriale come area orientata di un parallelogramma e come determinante di una matrice sono equivalenti.

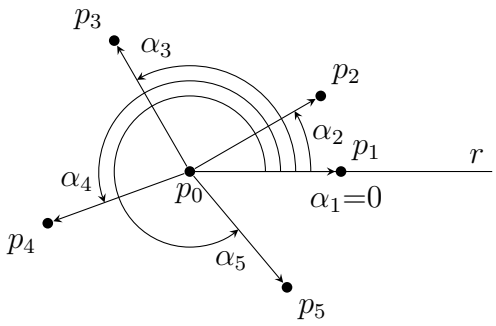
Esercizio 3 Usare il risultato dell'Esercizio 1 per dimostrare per induzione su n che l'area orientata di un poligono di n vertici $p_0, p_1, \ldots, p_{n-1}$ che sia semplice ma non necessariamente convesso si può calcolare in tempo lineare $O(n)$ mediante la formula

$$A = \frac{1}{2} \sum_{i=0}^{n-1} y_i (x_{i-1} - x_{i+1})$$

dove i vertici si intendono ordinati circolarmente in senso antiorario e quindi $x_{i-1} = x_{n-1}$ quando $i = 0$ e $x_{i+1} = x_0$ quando $i = n - 1$.

Esercizio 4 L'*angolo polare* di un punto p_i rispetto ad una origine p_0 è l'angolo formato dal vettore $\overrightarrow{p_0 p_i}$ con la semiretta orizzontale destra r con origine in p_0.

Ecco alcuni esempi di angoli polari:



Scrivere lo pseudo codice di un algoritmo per ordinare n punti $p_1, p_2, \ldots, p_n$ per angolo polare rispetto ad una origine p_0. L'algoritmo deve avere complessità $O(n \log n)$.

Esercizio 5 Usare il risultato dell'esercizio precedente per decidere in tempo $O(n^2 \log n)$ se un insieme di n punti contiene almeno tre punti collineari.

Esercizio 6 Un poligono è una linea chiusa costituita da una sequenza di segmenti, i *lati* del poligono, connessi agli estremi in corrispondenza dei *vertici* del poligono.

Normalmente un poligono è rappresentato con la sequenza $p_0, p_1, \ldots, p_{n-1}$ dei suoi vertici presi in senso antiorario sul *perimetro*; i lati sono quindi i segmenti $\overline{p_i p_{i+1}}$ che congiungono due vertici consecutivi più un ultimo lato $\overline{p_{n-1} p_0}$ che chiude il poligono.

Un poligono è *semplice* se non è intrecciato, ossia se non ci sono due lati non consecutivi che si intersecano.

Se un poligono è semplice possiamo suddividere i punti del piano in punti *interni* al poligono, punti *esterni* al poligono e punti di *frontiera* appartenenti al perimetro del poligono (ossia punti che stanno su uno dei lati).

Un poligono semplice è *convesso* se presi comunque due punti p_1 e p_2 interni o di frontiera anche tutti gli altri punti del segmento $\overline{p_1 p_2}$ sono o interni o di frontiera.

Si vuole determinare se una sequenza di punti $p_0, p_1, \ldots, p_{n-1}$ è la sequenza dei vertici di un poligono semplice convesso.

Mostrare che non è sufficiente percorrere il perimetro controllando che in nessun vertice si svolti a destra.

Trovare un algoritmo che risolva correttamente il problema in tempo lineare $O(n)$.

Esercizio 7 Dato un punto p ed un segmento $\overline{p_1p_2}$ spiegare come si possa determinare in tempo costante $O(1)$ se il segmento $\overline{p_1p_2}$ interseca la semiretta orizzontale destra con origine nel punto p. (Suggerimento: modificare opportunamente l'algoritmo SEGMENTS-INTERSECT).

Esercizio 8 Usare il risultato dell'esercizio precedente per trovare un algoritmo che decide in tempo $O(n)$ se un punto p è interno ad un poligono con n vertici che sia semplice ma non necessariamente convesso.

(Suggerimento: contare quanti lati sono intersecati dalla semiretta orizzontale destra con origine in p. Stare attenti a trattare correttamente anche i casi particolari in cui la semiretta passa per uno dei vertici o contiene un intero lato.)

4 La ricerca di due segmenti che si intersecano

Nel problema che ora affronteremo sono dati n segmenti e si vuole determinare se tra di essi ce ne sono almeno due che si intersecano.

Il nostro algoritmo deve soltanto dire se ci sono due segmenti che si intersecano, non deve cercare tutte le coppie di segmenti che si intersecano. Per trovare tutte le coppie di segmenti che si intersecano vale l'ovvio limite inferiore $\Omega(n^2)$ per il tempo calcolo nel caso pessimo: infatti nel caso pessimo in cui tutti i segmenti si intersecano a due a due ci sono $\Omega(n^2)$ coppie di segmenti da trovare.

Vedremo un algoritmo che risolve il problema in tempo $O(n \log n)$ usando una particolare tecnica detta *swepping* (spazzolatura).

Nella tecnica di swepping una immaginaria retta verticale, detta *spazzola*, si muove nel piano spazzolandolo da sinistra a destra in modo da incontrare ed elaborare tutti gli oggetti geometrici presenti nel piano.

Talvolta si può usare anche lo swepping polare in cui la spazzola è una semiretta che ruota attorno al suo punto di origine.

In ogni caso gli oggetti geometrici incontrati vengono inseriti in qualche tipo di struttura dati dinamica, che chiameremo *stato* della spazzola, in cui rimangono memorizzati fintanto che la spazzola li interseca e da cui vengono tolti non appena la spazzola passa oltre.

Chiameremo *evento* sia l'incontro della spazzola con un nuovo oggetto geometrico sia l'abbandono da parte della spazzola di un oggetto geometrico precedentemente incontrato. Tra un evento e l'evento immediatamente successivo lo stato della spazzola non cambia e quindi non serve che il movimento della spazzola sia continuo: basta che si muova passando direttamente da un evento a quello successivo.

Se gli eventi sono tutti noti all'inizio può essere opportuno ordinarli preventivamente da sinistra a destra per poi usare la loro sequenza ordinata per guidare lo spostamento della

spazzola. Occorre però tener conto che ci possono essere anche degli eventi contemporanei, ad esempio due figure geometriche che iniziano con la stessa coordinata x. Bisogna quindi fornire un criterio per ordinare i possibili eventi contemporanei.

Tornando al nostro problema dell'intersezione dei segmenti, gli oggetti geometrici sono gli n segmenti mentre gli eventi sono l'incontro della spazzola con gli estremi dei segmenti. Possiamo quindi identificare gli eventi con gli estremi dei segmenti. Ordineremo dunque per coordinata x crescente (da sinistra a destra) gli estremi dei segmenti e li useremo per guidare lo spostamento della spazzola.

Per dirimere il caso in cui più estremi di segmenti hanno la stessa coordinata x procediamo nel modo seguente:

1. Orientiamo ciascun segmento da sinistra a destra in modo da distinguere tra un estremo sinistro ed un estremo destro (nel caso in cui il segmento sia verticale chiameremo convenzionalmente estremo sinistro quello inferiore ed estremo destro quello superiore).

2. Conveniamo che gli estremi sinistri precedano sempre gli estremi destri aventi la stessa coordinata x. In altre parole, quando la spazzola raggiunge la coordinata x, considereremo prima i nuovi segmenti incontrati e poi i vecchi segmenti che vengono abbandonati. Questo ci assicura che per ogni coordinata x vi è sempre un'istante in cui nello stato della spazzola sono presenti tutti i segmenti che intersecano la retta verticale per x.

3. Gli estremi sinistri con la stessa coordinata x sono ordinati tra di loro per coordinata y crescente, e analogamente per gli estremi destri.

Durante l'esecuzione dell'algoritmo, lo stato della spazzola, ossia l'insieme dei segmenti incontrati e non ancora abbandonati viene mantenuto in una opportuna struttura dati dinamica T, in cui i segmenti sono ordinati per coordinata y crescente della loro intersezione con la spazzola (per gli eventuali segmenti verticali che stanno sulla spazzola si considera l'estremo inferiore come punto di intersezione).

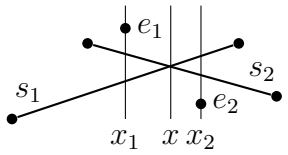


Figura 2: Tra i due eventi consecutivi e_1 ed e_2 che avvengono in due posizioni distinte $x_1 < x_2$ della spazzola l'ordine tra due segmenti s_1 ed s_2 cambia soltanto se i due segmenti si intersecano in un punto di coordinata x intermedia.

Osservazione 1 Tra un evento e il successivo l'ordine può cambiare soltanto se i due eventi hanno coordinate x_1 e x_2 distinte e vi sono due segmenti che si intersecano in un punto di coordinata x intermedia come illustrato in Figura 2.

Sulla struttura dati T vogliamo poter eseguire in modo efficiente le seguenti operazioni:

- INSERT(T, s) che aggiunge il segmento s alla struttura dati;

- DELETE(T, s) che toglie il segmento s dalla struttura dati;
- BELOW(T, s) che restituisce il segmento s' che precede s nell'ordine considerato, restituisce NIL quando s è il primo;
- ABOVE(T, s) che restituisce il segmento s'' che segue s nell'ordine considerato, restituisce NIL quando s è l'ultimo.

Usando un albero rosso-nero tutte e quattro le operazioni si eseguono in tempo $O(\log n)$.

L'algoritmo muove quindi la spazzola da sinistra a destra prendendo in considerazione gli estremi dei segmenti nell'ordine che abbiamo appena definito. Quando incontra un estremo sinistro aggiunge il segmento allo stato e quando incontra un estremo destro toglie il segmento dallo stato.

```
ANY-SEGMENTS-INTERSECT(S)    // S insieme di n segmenti s_1, ..., s_n.
 1  "Ordina gli estremi dei segmenti s_i = (p_i, q_i) in modo che p_i sia l'estremo
    sinistro e q_i sia quello destro "
 2  "Costruisci la sequenza degli eventi e_1, ..., e_2n"
    // Un evento e ha come attributi il segmento di appartenenza e.s ed un
    // valore booleano e.left che è TRUE se e è un estremo sinistro, FALSE
    // se è un estremo destro.
 3  "Ordina la sequenza degli eventi e_1, ..., e_2n per coordinata x crescente"
    // Gli eventi con la stessa coordinata x vengono ordinati mettendo prima
    // gli estremi sinistri e poi quelli destri e ordinando tra loro estremi sinistri
    // ed estremi destri per coordinata y crescente.
 4  for i = 1 to 2n
 5      s = e_i.s
 6      if e_i.left
 7          INSERT(T, s)
 8          s' = ABOVE(T, s)
 9          if s' ≠ NIL and SEGMENTS-INTERSECT(s, s')
10              return TRUE
11          s'' = BELOW(T, s)
12          if s'' ≠ NIL and SEGMENTS-INTERSECT(s, s'')
13              return TRUE
14      else s' = ABOVE(T, s), s'' = BELOW(T, s)
15          if s' ≠ NIL and s'' ≠ NIL and SEGMENTS-INTERSECT(s', s'')
16              return TRUE
17          DELETE(T, s)
18  return FALSE
```

Programma 4: Il test se un insieme di segmenti ne contiene almeno due che si intersecano.

Durante lo swepping l'algoritmo deve controllare se ci sono segmenti che si intersecano tra quelli presenti nello stato della spazzola (due segmenti che si intersecano devono prima o poi essere presenti contemporaneamente nello stato della spazzola).

In realtà l'algoritmo si limita a controllare che non ci siano intersezioni tra segmenti consecutivi (rispetto all'ordine prescelto) ma non controlla le intersezioni tra segmenti non consecutivi. Vedremo poi che questo è sufficiente per escludere *tutte* le intersezioni.

Per controllare che la spazzola T non contenga due segmenti consecutivi che si intersecano ci basta controllare ad ogni evento i segmenti che diventano consecutivi per effetto dell'evento stesso.

Un evento può essere l'inserimento di un nuovo segmento s quando se ne incontra l'estremo sinistro. In questo caso diventano consecutivi il segmento appena inserito s con quello immediatamente successivo s' e con quello immediatamente precedente s''. Basterà quindi controllare che non si intersechino s con s' ed s con s''.

Se invece l'evento è l'eliminazione di un segmento s quando se ne incontra l'estremo destro allora possono diventare consecutivi il segmento immediatamente successivo s' con il segmento immediatamente precedente s''. Basterà quindi controllare che s' ed s'' non si intersechino.

Il Programma 4 è lo pseudo codice dell'algoritmo.

È del tutto evidente che se l'algoritmo risponde TRUE ci sono sicuramente due segmenti che si intersecano. Per verificare che l'algoritmo è corretto occorre quindi mostrare che se ci sono due segmenti che si intersecano esso *deve* rispondere TRUE.

Supponiamo quindi che tra gli n segmenti ce ne siano almeno due che si itersecano e sia p il punto di intersezione tra due segmenti avente coordinata x_p minima (se ci sono più punti di intersezione sulla stessa retta verticale per x_p prendiamo quello con coordinata y_p minima).

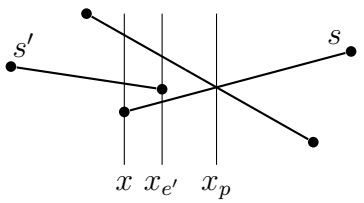
Dimostriamo che esiste un evento la cui coordinata x è minore o uguale di x_p ed in cui l'algoritmo scopre una intersezione.

Sia e l'evento che inserisce nella spazzola l'ultimo segmento s tra tutti quelli che si intersecano in p. Se prima dell'evento e l'algoritmo trova qualche intersezione siamo a posto. Altrimenti l'evento e viene elaborato e il segmento s viene inserito nella spazzola che a tal punto contiene tutti i segmenti che si intersecano in p.

Naturalmente e deve precedere p, ossia $x_e \leq x_p$, e questo ci assicura che l'ordine dei segmenti nella spazzola è rimasto corretto fino a quel momento.

Se quando s viene inserito in T esso risulta consecutivo con uno degli altri segmenti che si intersecano in p l'intersezione viene scoperta alla linea 7.

Altrimenti, al momento della sua inserzione s doveva essere separato dagli altri segmenti che si intersecano in p da qualche segmento intermedio s' che non passa per il punto p. Siccome p è il primo punto di intersezione, il segmento s' deve terminare prima di x_p. La situazione è illustrata nella figura seguente:



Deve quindi esistere un evento e' con coordinata $x_{e'} < x_p$ in cui s' viene tolto ed s diventa consecutivo con uno degli altri segmenti che si intersecano in p. Quindi l'intersezione viene scoperta alla linea 14.

Il tempo di esecuzione dell'algoritmo è $O(n)$ per costruire la sequenza degli eventi, $O(n \log n)$ per ordinarli e infine $O(n) \log n$ per esaminarli nel ciclo **for**. In totale tempo $O(n \log n)$.

Esercizio 9 Abbiamo detto che se lo stato della spazzola viene rappresentato con un albero rosso-nero le quattro operazioni INSERT, DELETE, BELOW e ABOVE si possono eseguire in tempo $O(\log n)$. Questo è vero soltanto se riusciamo ad effettuare in tempo costante il test su quale tra due segmenti ha l'itersezione più alta con la retta della spazzola.

Scrivere un algoritmo che dati due segmenti s ed s' e la coordinata x di una retta verticale che li interseca entrambi determina in tempo $O(1)$ quale dei due ha l'intersezione più alta. Ricordiamo che si possono usare soltanto somme, differenze, prodotti e prodotti vettoriali.

```
SEGMENTS-PRECEDE(p1, p2, p3, p4)    // Controlla se p1p2 precede p3p4.
1  if x1 == x3
2      if y1 == y3
3          return ANGLE-LEFT(p1, p2, p4) ≥ 0
4      else
5          return y1 < y3
6  if x1 > x3
7      return ANGLE-LEFT(p3, p4, p1) ≤ 0
8  else // x1 < x3
9      return ANGLE-LEFT(p1, p2, p3) ≥ 0
```

Programma 5: Il test di precedenza tra due segmenti nella spazzola.

Esercizio 10 Trovare un algoritmo per decidere in tempo $O(n \log n)$ se un poligono di n vertici è semplice.

Esercizio 11 Trovare un algoritmo per decidere in tempo $O(n \log n)$ se due poligoni semplici con un totale di n vertici si intersecano.

Esercizio 12 Sono dati n cerchi $c_1, c_2, \ldots, c_n$ nel piano. Di ciascun cerchio c_i sono date le coordinate (x_i, y_i) del centro ed il raggio r_i. Trovare un algoritmo per decidere in tempo $O(n \log n)$ se tra i cerchi dati ce ne sono almeno due che si intersecano. Suggerimento: modificare opportunamente l'algoritmo ANY-SEGMENTS-INTERSECT.

5 La ricerca dell'involucro convesso

In questo problema è dato un insieme Q di n punti nel piano e si vuole determinare il più piccolo poligono convesso che li contiene tutti.

Vedremo due algoritmi che risolvono questo problema. Il primo è dovuto a Graham e calcola l'involucro convesso in tempo $O(n \log n)$. Il secondo è dovuto a Jarvis e risolve il problema in tempo $O(nh)$ dove h è il numero di vertici dell'involucro convesso. Entrambi usano la tecnica dello swepping polare.

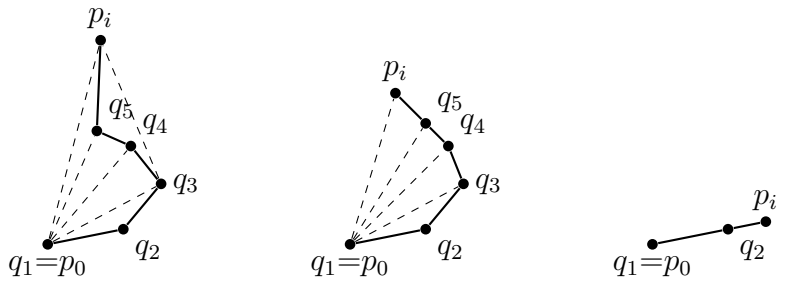


Figura 3: Alcuni esempi di aggiornamento della pila S quando si inserisce un nuovo punto p_i. (a) In q_5 e in q_4 si svolta a destra mentre in q_3 si svolta a sinistra; q_5 e q_4 vengono rimossi dalla pila prima di inserire p_i. (b) In q_5 si prosegue diritto mentre in q_4 si svolta a sinistra; viene rimosso soltanto q_5. (c) In q_2 si prosegue diritti e $q_1 = p_0$ non deve essere rimosso; viene rimosso soltanto q_2.

Algoritmo di Graham

L'algoritmo di Graham cerca dapprima il punto $p_0 \in Q$ più in basso a sinistra ossia quello con coordinata y minima e se ci sono più punti con la stessa coordinata y minima quello tra essi con coordinata x più piccola. Ordina quindi i rimanenti $n-1$ punti per angolo polare rispetto all'origine p_0 e se due punti hanno lo stesso angolo polare li ordina per distanza da p_0. Siano $p_0, p_1, p_2, \ldots, p_{n-1}$ i punti così ordinati.

L'algoritmo costruisce poi incrementalmente l'involucro convesso dei punti $p_0, \ldots, p_i$ per $i = 0, \ldots, n-1$. L'involucro convesso viene mantenuto in una struttura dati S organizzata come una pila al fondo della quale vi è il primo punto dell'involucro convesso (che è sempre p_0) e muovendosi verso la cima della pila ci sono gli altri vertici dell'involucro convesso presi in ordine antiorario. Le operazioni su S sono:

- PUSH(S, p) che aggiunge il punto p in cima alla pila;
- POP(S) che toglie il punto in cima alla pila;
- TOP(S) che restituisce il punto in cima alla pila;
- NEXT-TO-TOP(S) che restituisce il penultimo punto in cima alla pila.

L'algoritmo inizia inserendo p_0 nella pila. Per come è stato scelto p_0 i due vettori $\overrightarrow{p_0 p_1}$ e $\overrightarrow{p_0 p_{n-1}}$ non possono essere allineati e orientati in senso opposto. Inoltre tutti gli altri punti sono compresi nell'angolo individuato da tali due vettori. Dunque p_0 appartiene sicuramente all'involucro convesso.

Se $n = 1$ abbiamo finito, Altrimenti l'algoritmo aggiunge uno alla volta i rimanenti punti p_i per $i = 1, \ldots, n-1$ mantenendo la proprietà che S sia l'involucro convesso dei punti $p_0, \ldots, p_i$ considerati fino a quel momento.

Per mantenere vera tale proprietà quando aggiungiamo un nuovo punto p_i dobbiamo prima eliminare alcuni punti in cima alla pila; precisamente i punti in cui percorrendo l'ultimo lato di S per poi proseguire per p_i si effettua una svolta a destra oppure si prosegue

```
GRAHAM-SCAN(Q)    // Q insieme di n punti del piano.
 1  "Cerca il punto p_0 ∈ Q più in basso a sinistra"
    // p_0 è il punto con coordinata y minima e a parità di coordinata y quello con
    // coordinata x minima.
 2  "Ordina i rimanenti punti per angolo polare rispetto all'origine p_0"
    // Punti con lo stesso angolo polare vengono ordinati mettendo prima quelli
    // più vicini a p_0. Siano p_1, ..., p_{n-1} i punti ordinati in questo modo.
 3  PUSH(S, p_0)
 4  for i = 1 to n − 1
 5      p = TOP(S)
 6      q = NEXT-TO-TOP(S)
 7      while q ≠ NIL and TURN-LEFT(q, p, p_i) ≤ 0
 8          POP(S)
 9          p = q
10          q = NEXT-TO-TOP(S)
11      PUSH(S, p_i)
12  return S
```

Programma 6: L'algoritmo di Graham.

diritti. Naturalmente il primo punto p_0 non deve mai essere rimosso. La figura 3 illustra alcune situazioni possibili.

Il Programma 6 è lo pseudo codice dell'algoritmo.

Alla fine dell'algoritmo di Graham S contiene i vertici di un poligono semplice in quanto i suoi vertici sono presi in ordine di angolo polare crescente rispetto a p_0 (non si torna mai indietro) ed è convesso in quanto passando per ogni vertice si gira sempre a sinistra. I vertici di tale poligono sono inoltre punti dell'insieme Q.

Siccome ogni punto di Q viene inserito prima o poi in S, per dimostrare la correttezza dell'algoritmo di Graham è sufficiente mostrare che i punti eliminati non appartengono all'involucro convesso e questo non è difficile. Basta osservare che quando eliminiamo un punto q_k in cima alla pila perché in esso si gira a destra tale punto è sicuramente contenuto nel triangolo di vertici p_0, q_{k-1} e p_i mentre se si prosegue diritto esso è contenuto nel segmento $\overline{q_{k-1}p_i}$; in entrambi i casi esso non può appartenere all'involucro convesso.

Il tempo richiesto per eseguire l'algoritmo di Graham è $O(n)$ per eseguire la ricerca alla linea 1, tempo $O(n \log n)$ per eseguire l'ordinamento alla riga 2, Il ciclo **for**, se si esclude il tempo richiesto per eseguire il ciclo **while** interno, richiede tempo $O(n)$. Ogni volta che si esegue una iterazione del ciclo **while** viene rimosso un punto da S; siccome ciascun punto può essere rimosso una sola volta le iterazioni del ciclo **while** sono in totale al più $O(n)$ e ciascuna di esse richiede tempo costante. Dunque l'algoritmo di Graham si esegue in tempo $O(n \log n)$ e l'operazione più costosa è l'ordinamento; tutto il resto si esegue in tempo lineare.

Algoritmo di Jarvis

L'algoritmo di Jarvis calcola l'involucro convesso in tempo $O(nh)$ dove h è il numero di vertici dell'involucro convesso. L'algoritmo di Jarvis risulta quindi superiore a quello di

Graham quando il numero di vertici dell'involucro convesso è molto piccolo ossia quando $h = o(\log n)$.

L'algoritmo di Jarvis simula l'operazione di stendere una recinzione attorno ai punti di Q. Comincia ancorando la recinzione al punto p_0 più in basso a sinistra e poi procede in senso antiorario avvolgendo la recinzione attorno ai punti finché non si ritorna in p_0.

Più formalmente l'algoritmo di Jarvis costruisce la sequenza H dei vertici $q_1, \ldots, q_k$ dell'involucro convesso iniziando con $q_1 = p_0$ che sappiamo appartenere sicuramente all'involucro convesso. Il punto successivo q_2 è quello che forma il più piccolo angolo polare rispetto alla semiretta orizzontale destra con origine in p_0 (se ce ne sono più di uno si sceglie quello più lontano da p_0).

Ad ogni passo successivo si sceglie il punto con angolo polare minore rispetto all'ultimo punto in H e si procede in questo modo finchè si arriva ad inserire il punto p_t più in alto a destra (che appartiene sicuramente all'involucro convesso). In questo modo viene costruita la parte ascendente dell'involucro convesso.

A questo punto si procede allo stesso modo per costruire la parte discendente dell'involucro convesso ma questa volta cerchiamo il punto che forma il minimo ongolo polare rispetto alla semiretta *sinistra* con origine nell'ultimo punto in H. La costruzione della parte discendente termina non appena si ritorna al punto p_0 (che non deve essere aggiunto).

```
JARVIS-MARCH(Q)    // Q insieme di n punti del piano.
 1  "Cerca il punto p_0 ∈ Q più in basso a sinistra"
    // p_0 è il punto con coordinata y minima e a parità di coordinata y quello con
    // coordinata x minima.
 2  "Cerca il punto p_t ∈ Q più in alto a destra"
    // p_t è il punto con coordinata y massima e a parità di coordinata y quello con
    // coordinata x massima.
 3  H[1] = p_0
 4  k = 1
 5  while H[k] ≠ p_t
 6       q = MIN-POLAR-RIGHT(H[k], Q)
 7       k = k + 1
 8       H[k] = q
 9  q = MIN-POLAR-LEFT(H[k], Q)
10  while q ≠ p_0
11       k = k + 1
12       H[k] = q
13       q = MIN-POLAR-LEFT(H[k], Q)
14  return H
```

Programma 7: L'algoritmo di Jarvis.

Il Programma 7 è lo pseudo codice dell'algoritmo.

L'algoritmo di Jarvis richiede tempo $O(n)$ per eseguire le linee 1 e 2 dopo di che vengono eseguite in totale h iterazioni dei cicli **while** e ciascuna iterazione richiede tempo $O(n)$ per la ricerca del minimo. In totale l'elgoritmo di Jarvis richiede quindi tempo $O(nh)$.

Esercizio 13 Nell'algoritmo di Graham quando si deve ordinare un insieme di punti per angolo polare sappiamo che l'angolo polare è sempre minore di 180^o (per come è stata scelta l'origine). Nell'ordinare tali punti occorre inoltre dirimere l'ambiguità tra punti che hanno lo stesso angolo polare ponendo prima quello dei due più vicino all'origine.

Scrivere un algoritmo che data l'origine o e due punti p e q il cui angolo polare rispetto ad o è minore di 180^o determina in tempo $O(1)$ quale dei due precede l'altro nell'ordine richiesto. Ricordare che le operazioni permesse sono soltanto somme, sottrazioni, prodotti e confronti.

Esercizio 14 Vi sono altri modi per calcolare l'involucro convesso in tempo $O(n \log n)$. Uno di questi utilizza la tecnica *incrementale*. I punti vengono dapprima ordinati per coordinata x crescente ottenendo una sequenza ordinata $p_1, p_2, \ldots, p_n$. Poi viene costruito in modo incrementale l'involucro convesso dei primi $i+1$ punti $p_1, \ldots, p_i, p_{i+1}$ conoscendo l'involucro convesso dei primi i punti $p_1, \ldots, p_i$. Scrivere lo pseudo codice di un algoritmo che opera in questo modo. Assicurarsi che il tempo calcolo risulti $O(n \log n)$.

Esercizio 15 Per calcolare l'involucro convesso possiamo anche usare la tecnica *divide-et-impera*. Con questa tecnica l'insieme di punti viene suddiviso in due sottoinsiemi, uno contenente gli $\lceil n/2 \rceil$ punti più a sinistra (con coordinata x minore) e l'altro contenente i rimanenti $\lfloor n/2 \rfloor$ punti più a destra. Vengono calcolati quindi ricorsivamente gli involucri convessi dei due sottoinsiemi e poi essi vengono riuniti per ottenere l'involucro convesso dell'intero insieme. Scrivere lo pseudo codice di un algoritmo che opera in questo modo. Assicurarsi che il tempo calcolo risulti $O(n \log n)$.

6 La localizzazione dei punti nel piano

Nel problema della localizzazione dei punti nel piano è data una suddivisione del piano in regioni. Si vuole rappresentare tale suddivisione con una opportuna struttura dati che permetta di trovare rapidamente a quale regione appartiene un punto p.

Assumiamo che le regioni siano dei poligoni (non necessariamente convessi ed eventualmente illimitati) e che tali poligoni siano rappresentati dalla successione di lati e vertici che formano il perimetro percorso in senso antiorario.

Assumiamo inoltre che per ogni lato siano memorizzate le due regioni che esso separa.

In questo caso il problema della localizzazione dei punti si può ridurre al seguente problema:

Problema 1 (Segmento dominante) *Dati n segmenti $s_1, \ldots, s_n$ ed un punto p trovare il segmento s_i che si trova immediatamente sopra il punto p.*

Se riusciamo a risolvere il problema del segmento dominante sull'insieme dei lati delle regioni poligonali in cui è suddiviso il piano allora la regione a cui appartiene un punto p è quella limitata superiormente dal lato trovato.[1]

Affrontiamo quindi il problema del segmento dominante. Assumiamo che gli n segmenti $s_1, \ldots, s_n$ non si intersechino e che non ci siano segmenti verticali. Questo non è

[1] Nel successivo paragrafo sulle note implementative vedremo come trattare il caso in cui la regione sia superiormente illimitata.

vero per i lati delle regioni di una suddivisione del piano che possono essere verticali e possono intersecarsi agli estremi; l'Esercizio 16 mostra che segmenti verticali e intersezioni agli estremi possono essere facilmente trattati.

Dati gli n segmenti $s_1, \ldots, s_n$ costruiamo una struttura dati detta *partizione trapeziodale* del piano nel modo seguente:

1. Partiamo con la regione R_0 che comprende tutto il piano.

2. Usiamo il primo segmento s_1 per suddividere R_0 nelle quattro regioni R_1, R_2, R_3 ed R_4 come illustrato in Figura 4a.

3. Usiamo il secondo segmento s_2 per suddividere alcune delle regioni precedenti come illustrato in Figura 4b e ripetiamo quindi la stessa operazione per ciascuno dei segmenti successivi $s_3, \ldots, s_n$.

Quando suddividiamo una regione (in al più quattro parti) non eliminiamo tale regione dalla nostra struttura dati ma aggiungiamo dei puntatori alle regioni che si ottengono dalla suddivisione.

In questo modo otteniamo un grafo orientato aciclico in cui le regioni costituiscono i vertici e gli archi sono i puntatori da una regione alle regioni che si ottengono suddividendola (Figura 4c). Osserviamo che aggiungendo un segmento può accadere che una regione della suddivisione indotta da tale segmento sia costituita dall'unione di parti di una o più regioni precedenti. Nell'esempio di Figura 4 questo avviene per la regione R_7 che è formata da una parte della regione R_3 unita ad una parte della regione R_4. In questo caso in R_7 entrano due puntatori, uno che parte da R_3 ed uno che parte da R_4.

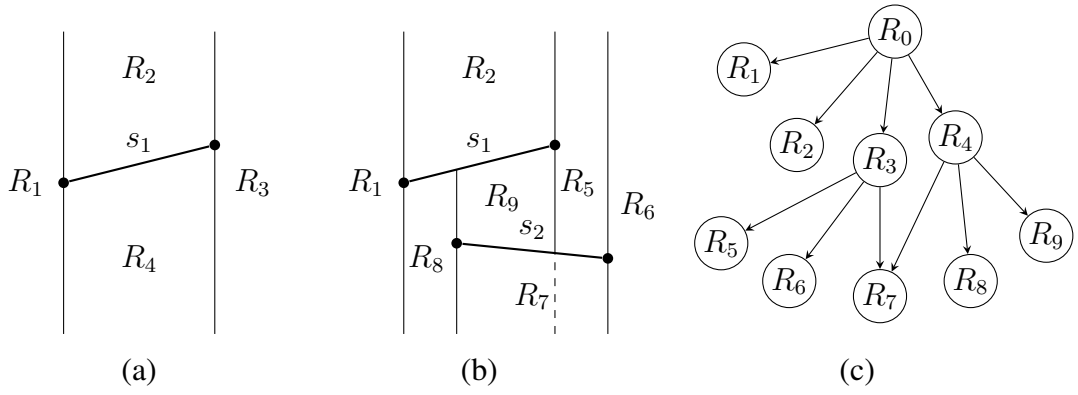


Figura 4: a) La partizione trapezoidale del piano mediante il primo segmento s_1. b) La partizione trapezoidale che si ottiene aggiungendo il secondo segmento. c) Il grafo orientato aciclico delle regioni. Come si vede non è una struttura ad albero: la regione R_7 si ottiene dalla suddivisione delle due regioni R_3 ed R_4.

Una volta costruita la partizione trapezoidale relativa ai segmenti $s_1, \ldots, s_n$ possiamo decidere a quale regione appartiene un punto p seguendo la storia della costruzione della partizione nel seguente modo:

1. Il punto p appartiene sicuramente alla regione iniziale R_0. Partiamo quindi con con la regione $R = R_0$;

2. Supponiamo di conoscere che $p \in R$. Se R non è stata suddivisa ulteriormente R è la regione cercata.

3. Altrimenti la regione R è stata suddivisa in al più quattro regioni. Possiamo trovare tali regioni usando i puntatori creati durante la costruzione della struttura dati. Tra queste regioni cerchiamo la regione R' che contiene il punto p, poniamo $R = R'$ e ripetiamo il punto 2.

Una volta trovata la regione trapezoidale a cui appartiene il punto p il segmento dominante è quello a cui appartiene il lato superiore del trapezoide oppure non esiste se il trapezoide è illimitato superiormente. Questa informazione può facilmente essere calcolata e memorizzata per ogni regione trapezoidale durante la costruzione della struttura dati.

Per verificare se un punto p appartiene ad una regione trapezoidale basta verificare se la sua coordinata x è compresa tra i due lati verticali e controllare la posizione di p rispetto al lato superiore ed inferiore del trapezoide. Tutto questo si può fare in tempo costante.

Siccome al Passo 3 dobbiamo verificare al più quattro regioni il tempo calcolo per eseguire la ricerca è proporzionale alla lunghezza della storia della regione di appartenenza, ossia al numero di volte che la regione contenente il punto p è cambiata durante la costruzione della partizione trapezoidale a causa di una sua suddivisione.

Assumiamo che l'ordine di inserzione dei segmenti $s_1, \ldots, s_n$ sia casuale e ci chiediamo quale sia la lunghezza media della storia della regione R che contiene il punto p. Procediamo a ritroso togliendo un segmento alla volta dalla struttura dati e contiamo quante volte cambia la regione a cui appartiene il punto p. Supponiamo che dopo aver rimosso i segmenti $s_{i+1}, \ldots, s_n$ il punto p appartenga alla regione R e ci chiediamo quale sia la probabilità che la regione R cambi quando viene rimosso il segmento s_i.

Una regione cambia soltanto quando viene tolto o il segmento che la delimita superiormente o il segmento che la delimita inferiormente o il segmento il cui estremo determina il lato sinistro o il segmento il cui estremo determina il lato destro: in totale al più quattro segmenti. Siccome il segmento tolto è uno qualsiasi degli i segmenti che non abbiamo ancora rimosso la probabilità che la regione cambi è minore o uguale di $4/i$. Il valore atteso del numero di volte che la regione cambia è quindi

$$\sum_{i=1}^{n} \frac{4}{i} = O(\log n).$$

e pertanto la localizzazione di un punto p nella partizione trapezoidale richiede tempo medio $O(\log n)$.

Rimane da analizzare il costo della costruzione della partizione trapezoidale.

L'inserzione di un nuovo segmento s_i richiede due passi

1. Cerca le regioni intersecate dal segmento s_i.

2. Percorri il segmento da sinistra a destra suddividendo opportunamente le regioni intersecate da s_i. Se dalla suddivisione di due regioni consecutive si ottengono due regioni adiacenti limitate superiormente e inferiormente dagli stessi segmenti riuniscile in un'unica regione.

Come vedremo nelle note implementative l'esecuzione del primo passo richiede tempo medio $O(\log i)$ usando un algoritmo analogo all'algoritmo di localizzazione di un punto. Dunque l'esecuzione dei primi passi richiede in totale tempo medio $O(n \log n)$.

Per i secondi passi osserviamo che una regione può essere suddivisa in al più quattro parti con un costo costante O(1). Possiamo quindi effettuare una analisi ammortizzata attribuendo ad ogni regione che viene creata e successivamente suddivisa un costo ammortizzato che comprende anche il costo della sua successiva suddivisione. Otteniamo quindi un costo ammortizzato costante per ogni regione creata e poi suddivisa mentre il costo delle regioni finali risulta nullo in quanto prepagato con il costo ammortizzato delle regioni suddivise.

Dobbiamo quindi calcolare il numero di regioni che vengono suddivise durante la costruzione della partizione trapezoidale.

In una partizione trapezoidale i lati verticali sono in corrispondenza dei vertici dei segmenti inseriti. Per ogni vertice di un segmento inserito ci sono al più due lati sulla sua verticale: quello che congiunge il vertice stesso con il segmento immediatamente soprastante e quello che lo congiunge con il segmento immediatamente sottostante. Siccome i vertici dei segmenti sono $2n$ il numero totale di vertici della partizione trapezoidale è minore o uguale di $6n$.

Possiamo vedere la partizione trapezoidale come un grafo planare connesso con al più $6n + 1$ vertici (facendo convergere tutti i lati illimitati ad un unico nuovo vertice z).

Per la formula di Eulero in un grafo planare connesso il numero di regioni f, il numero di lati e ed il numero di vertici v sono legati dalla relazione $v - e + f = 2$. Inoltre in un grafo planare con almeno tre vertici vale la diseguaglianza $e \leq 3v - 6$ (che diventa uguaglianza quando aggiungiamo gli archi che servono per suddividere tutte le facce in triangoli). Sostituendo nella formula di Eulero otteniamo $f = 2 + e - v \leq 2v - 4 \leq 12n - 2 = O(n)$.

Dunque il numero di regioni della partizione trapezoidale finale è $O(n)$ e quindi anche il numero di regioni suddivise durante la costruzione è $O(n)$.

Siccome abbiamo attribuito un costo ammortizzato costante ad ogni regione che viene suddivisa il costo totale dell'esecuzione dei secondi passi è $O(n)$ e l'intera costruzione della partizione trapezoidale ha costo $O(n \log n)$.

Note implementative

Uno dei problemi per l'implementazione dell'algoritmo della localizzazione di un punto è dovuto al fatto che ci sono regioni illimitate. Per poter trattare in modo uniforme regioni illimitate e regioni limitate si deve aggiungere la retta all'infinito, ossia la retta dell'orizzonte.

In geometria proiettiva questo viene fatto usando coordinate proiettive che sono terne (x, y, z) con x, y e z non tutte e tre nulle. Le coordinate proiettive sono definite a meno di una costante di proporzionalità, ossia, per ogni $t \neq 0$ le coordinate (x, y, z) e (tx, ty, tz) rappresentano lo stesso punto.

Se $z \neq 0$ la terna (x, y, z) rappresenta il punto di coordinate cartesiane $(x/z, y/z)$. Se $z = 0$ la terna $(x, y, 0)$ rappresenta il punto all'orizzonte nella direzione del vettore di coordinate (x, y).

Anche per i vettori si possono usare le coordinate proiettive: se $z \neq 0$ il vettore di coordinate proiettive (x, y, z) rappresenta il vettore di coordinate cartesiane $(x/z, y/z)$ mentre il vettore $(x, y, 0)$ rappresenta il vettore di lunghezza infinita in direzione (x, y).

Per calcolare un vettore come differenza $q - p$ tra due punti p e q dobbiamo prima rendere uguali le coordinate z_p e z_q sfruttando la costante di proporzionalità. Per fare questo basta moltiplicare le coordinate di p per z_q e le coordinate di q per z_p. Le coordinate del vettore differenza sono quindi

$$q - p \equiv (x_q z_p - x_p z_q, y_q z_p - y_p z_q, z_p z_q)$$

Noi però dobbiamo lavorare senza la divisione. Assumeremo quindi che z possa assumere soltanto i valori 0 e 1 e interpreteremo il punto di coordinate proiettive $(x, y, 1)$ come il punto p di coordinate cartesiane (x, y) e il punto di coordinate proiettive $(x, y, 0)$ come il punto sulla linea dell'orizzonte nella direzione del vettore di coordinate (x, y).

Inoltre, siccome noi useremo i vettori soltanto per determinare come sono ruotati l'uno rispetto all'altro possiamo limitarci a considerare soltanto la loro direzione che è data dalle coordinate x e y e trascurare la coordinata z che determina la loro lunghezza. Calcoleremo quindi la differenza di due punti come

$$q - p \equiv (x_q z_p - x_p z_q, y_q z_p - y_p z_q)$$

Chiameremo questo sistema di coordinate *coordinate intere estese*.

Osserviamo che le coordinate proiettive (x, y, z) e $(-x, -y, -z)$ rappresentano lo stesso punto a causa del coefficiente di proporzionalità che può essere anche negativo. In particolare rimangono identificati i due punti all'orizzonte nelle due direzioni opposte (x, y) e $(-x, -y)$ in quanto $(x, y, 0)$ e $(-x, -y, 0)$ differiscono per il coefficiente di proporzionalità $t = -1$. Questo ha senso in quanto proiettando il piano su di un altro piano non parallelo tali due punti finiscono in un unico punto.

Noi non dovremo proiettare il piano ma soltanto descrivere una sua partizione poligonale. Abbiamo inoltre assunto che z possa assumere soltanto i valori 1 e 0 e quindi non sia mai negativa. Possiamo quindi considerare distinti i due punti all'orizzonte in direzioni opposte di coordinate $(x, y, 0)$ e $(-x, -y, 0)$.

Dati due punti p e q di coordinate estese (x_p, y_p, z_p) e (x_q, y_q, z_q) il segmento di estremi p e q viene interpretato nel modo seguente:

- Se $z_p = z_q = 1$ il segmento $\overline{pq}$ è un normale segmento di estremi p e q e $\overrightarrow{pq}$ è il segmento orientato da p a q;
- Se $z_p = 1$ e $z_q = 0$ il segmento $\overline{pq}$ è la semiretta di origine p in direzione del punto q all'orizzonte e $\overrightarrow{pq}$ è tale semiretta orientata nel verso uscente dall'origine p;
- Se $z_p = 0$ e $z_q = 1$ il segmento $\overline{pq}$ è la semiretta di origine q in direzione del punto p all'orizzonte e $\overrightarrow{pq}$ è tale semiretta orientata nel verso entrante nell'origine q;
- Se $z_p = 0$ e $z_q = 0$ il segmento $\overline{pq}$ è il segmento di orizzonte sotteso dall'angolo tra le due direzioni (x_p, y_p) e (x_q, y_q). Siccome l'orizzonte è una linea chiusa questa definizione risulta ambigua: ci sono due angoli tra le direzioni (x_p, y_p) e (x_q, y_q). Converremo quindi che $\overline{pq}$ rappresenti ambiguamente entrambi i segmenti di orizzonte. L'ambiguità si risolve considerando il segmento orientato $\overrightarrow{pq}$ che è il segmento di orizzonte sotteso dall'angolo orientato in senso antiorario tra la direzione del punto p e quella del punto q.

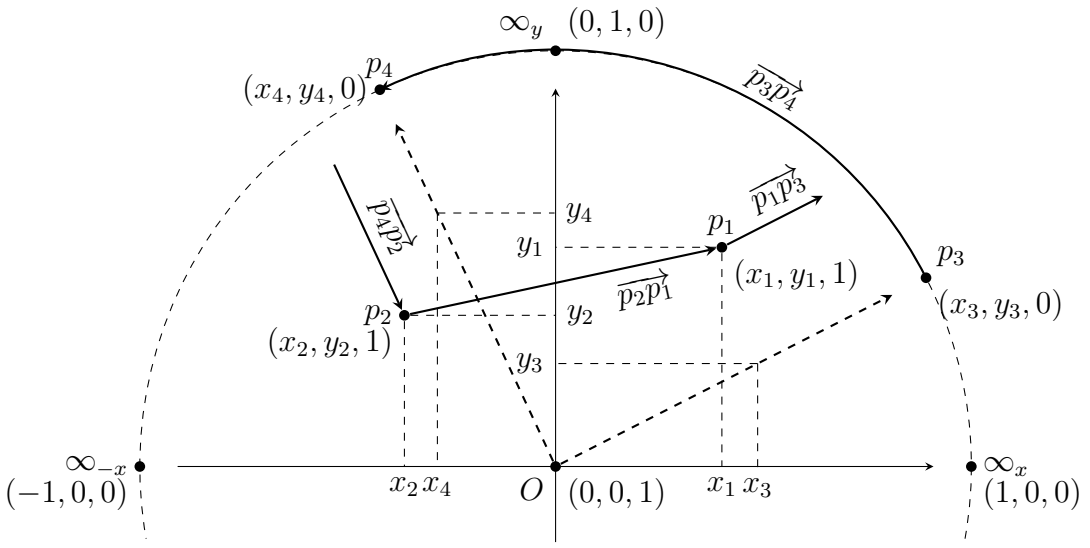


Figura 5: Coordinate intere estese e loro interpretazione. All'asse delle x è stato aggiunto a sinistra un primo punto all'infinito $\infty_{-x} \equiv (-1,0,0)$ e a destra un ultimo punto all'infinito $\infty_x \equiv (1,0,0)$. La stessa cosa vale anche per l'asse delle y. I punti $p_1 \equiv (x_1,y_1,1)$ e $p_2 \equiv (x_2,y_2,1)$ sono i due punti di coordinate cartesiane (x_1,y_1) e (x_2,y_2). I punti $p_3 \equiv (x_3,y_3,0)$ e $p_4 \equiv (x_4,y_4,0)$ sono i due punti all'infinito nelle direzioni (x_3,y_3) e (x_4,y_4). Il segmento orientato $\overrightarrow{p_2p_1}$ è un normale segmento orientato finito. Il segmento orientato $\overrightarrow{p_4p_2}$ rappresenta la semiretta con origine in p_2, che si estende in direzione di p_4 ed è orientata nel verso entrante nell'origine p_2. Il segmento orientato $\overrightarrow{p_1p_3}$ rappresenta la semiretta con origine in p_1, che si estende in direzione di p_3 ed è orientata nel verso uscente dall'origine p_1. Infine il segmento orientato $\overrightarrow{p_3p_4}$ appartiene alla retta all'infinito e rappresenta l'angolo orientato in senso antiorario tra la direzione di p_3 e quella di p_4.

La Figura 5 illustra l'interpretazione di punti e segmenti orientati in coordinate intere estese.

Dati due punti p e q di coordinate intere estese (x_p,y_p,z_p) e (x_q,y_q,z_q) la loro differenza $q-p$ ha la seguente interpretazione:

- Se $z_p = z_q = 1$ il vettore $q-p$ ha coordinate (x_q-x_p, y_q-y_p) ed è quindi l'usuale vettore differenza tra due punti di coordinate cartesiane (x_p,y_p) e (x_q,y_q);
- Se $z_p = 1$ e $z_q = 0$ il vettore $q-p$ ha coordinate (x_q,y_q) ed è un vettore che ha la stessa direzione e verso della semiretta orientata $\overrightarrow{pq}$;
- Se $z_p = 0$ e $z_q = 1$ il vettore $q-p$ ha coordinate $(-x_p,-y_p)$ ed è un vettore che ha la stessa direzione e verso della semiretta orientata $\overrightarrow{qp}$;
- Se $z_p = 0$ e $z_q = 0$ il vettore $q-p$ ha coordinate $(0,0)$ ed è quindi il vettore nullo la cui direzione è indeterminata come è giusto che sia visto che i due punti p e q stanno entrambi sulla linea dell'orizzonte.

A questo punto dobbiamo rivedere il test ANGLE-LEFT in termini di coordinate intere estese. Per questo basta sostituire, nel calcolo del prodotto vettoriale $(p_1-p_0)\times(p_2-p_0)$

la differenza tra vettori calcolata in termini di coordinate estese:

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 z_0 - x_0 z_1)(y_2 z_0 - y_0 z_2) - (x_2 z_0 - x_0 z_2)(y_1 z_0 - y_0 z_1)$$

Il problema è che se p_0 è un punto all'infinito il prodotto è sempre nullo. Possiamo risolvere questo problema osservando che l'angolo da p_1 a p_2 con vertice in p_0 ha lo stesso orientamento dell'angolo da p_2 a p_0 con vertice in p_1 e dell'angolo da p_0 a p_1 con vertice in p_2. Tali tre angoli sono infatti i tre angoli di un triangolo orientati nello stesso verso di percorrenza del perimetro del triangolo.

Se tutti e tre i vertici sono all'infinito essi appartengono alla linea dell'orizzonte e quindi sono allineati, altrimenti, per fare il test possiamo sempre scegliere un vertice al finito. Il Programma 8 è lo pseudo codice della procedura ANGLE-LEFT modificata.

```
ANGLE-LEFT(p_0, p_1, p_2)
     // p_0, p_1 e p_2 punti di coordinate intere estese (x_0, y_0, z_0), (x_1, y_1, z_1) e (x_2, y_2, z_2).
 1   if z_0 ≠ 0
 2       // p_0 è un punto al finito. Calcolo d = (p_1 − p_0) × (p_2 − p_0).
 3       d = (x_1 z_0 − x_0 z_1)(y_2 z_0 − y_0 z_2) − (x_2 z_0 − x_0 z_2)(y_1 z_0 − y_0 z_1)
 4   elseif z_1 ≠ 0
 5       // p_1 è un punto al finito. Calcolo d = (p_2 − p_1) × (p_0 − p_1).
 6       d = (x_2 z_1 − x_1 z_2)(y_0 z_1 − y_1 z_0) − (x_0 z_1 − x_1 z_0)(y_2 z_1 − y_1 z_2)
 7   elseif z_2 ≠ 0
 8       // p_2 è un punto al finito. Calcolo d = (p_0 − p_2) × (p_1 − p_2).
 9       d = (x_0 z_2 − x_2 z_0)(y_1 z_2 − y_2 z_1) − (x_1 z_2 − x_2 z_1)(y_0 z_2 − y_2 z_0)
10   else // Tutti e tre i punti sono sulla retta dell'orizzonte.
11       d = 0
     // Se d > 0 l'angolo tra p_0p_1 e p_0p_2 è orientato in senso antiorario.
     // Se d < 0 l'angolo è orientato in senso orario.
     // Se d = 0 i due segmenti sono collineari oppure uno dei due è degenere.
12   return d
```

Programma 8: La procedura ANGLE-LEFT in coordinate intere estese.

I due test TURN-LEFT e SEGMENTS-INTERSECT usano il test ANGLE-LEFT. Lasciamo come esercizio verificare che essi funzionano anche in coordinate estese purché venga richiamata la nuova versione del test ANGLE-LEFT.

Usando coordinate estese tutte le regioni in cui è suddiviso il piano risultano poligoni chiusi con eventualmente qualche vertice sulla linea dell'orizzonte. Quando vogliamo localizzare un punto siamo quindi certi che vi è sempre un lato dominante.

Supponiamo quindi che le regioni in input siano state elaborate per collezionare e memorizzare tutti i lati associando ad ogni lato le due regioni che esso separa (una sola se si tratta di lati all'infinito). Il risultato è un insieme $s_1, \ldots, s_n$ di segmenti che non si intersecano internamente ma eventualmente soltanto nei vertici. Il Programma 9 è lo pseudo codice della procedura COLLEZIONA-SEGMENTI per collezionare i segmenti. La procedura COLLEZIONA-SEGMENTI richiama la procedura INSERT che aggiunge un nuovo segmento alla collezione S dopo essersi assicurati che il segmento non sia già presente.

La struttura dati più adatta per rappresentare la collezione S è un albero di intervalli che ci garantisce una complessità $O(\log n)$ di INSERT nel caso pessimo. Possiamo anche usare una tavola hash con complessità media $O(n)$ per INSERT (ma $O(n^2)$ nel caso pessimo).

```
COLLEZIONA-SEGMENTI(R_1, ..., R_m)
    // R_1, ..., R_m regioni poligonali di una partizione del piano rappresentate dai
    // vertici in coordinate intere estese presi in ordine antiorario sul perimetro.
 1  S = ∅
 2  for i = 1 to m
 3      R = R_i
        // R regione poligonale di vertici p_0, ..., p_{n-1}
 4      for j = 0 to n − 1
 5          p = p_j, q = p_{(j+1) mod n}
            // Oriento il segmento pq da sinistra a destra;
            // dal basso all'alto se è verticale.
 6          if p_x q_z < q_x p_z or (p_x q_z == q_x p_z and p_y q_z ≤ q_y p_z)
 7              s = pq→
 8          else s = qp→
 9          INSERT(S, s, R)
            // INSERT aggiunge ad S il segmento s con un riferimento ad R.
            // Se S contiene già s aggiunge soltanto un riferimento ad R.
10  return S
```

Programma 9: La procedura COLLEZIONA-SEGMENTI.

Per la costruzione della suddivisione trapezoidale rappresentiamo ogni regione trapezoidale con le due rette verticali che la delimitano a sinistra e a destra e i due segmenti che la delimitano superiormente e inferiormente.

Per rappresentare le rette verticali usiamo le coordinate intere estese del punto di intersezione $p_x \equiv (x, 0, 1)$ con l'asse delle x. Se la regione trapezoidale non è limitata a sinistra useremo il punto $\infty_{-x} \equiv (-1, 0, 0)$ che è il punto in cui l'asse delle x interseca a sinistra la retta dell'orizzonte. Analogamente se una regione trapezoidale è illimitata a destra useremo il punto $\infty_x \equiv (1, 0, 0)$ che è il punto in cui l'asse delle x interseca a destra la retta dell'orizzonte.

Per memorizzare i segmenti superiore e inferiore della regione useremo due puntatori ai segmenti $s_1, \ldots, s_n$ collezionati.

Ogni regione trapezoidale R avrà quindi quattro attributi: $R.left$ ed $R.right$ che sono le coordinate estese delle intersezioni con l'asse delle x dei due lati verticali e $R.up$ ed $R.down$ che sono i puntatori ai segmenti che la delimitano superiormente e inferiormente.

Ogni regione trapezoidale R avrà inoltre quattro puntatori $R.R_h$, con $h = 1, \ldots, 4$, per memorizzare le regioni in cui essa verrà eventualmente suddivisa.

Se R è l'intero piano allora $R.left = \infty_{-x}$, $R.right = \infty_x$, $R.up = \overline{\infty_x \infty_{-x}}$ e $R.down = \overline{\infty_{-x} \infty_x}$. Dovremo quindi aggiungere anche i due segmenti $\overline{\infty_x \infty_{-x}}$ e $\overline{\infty_{-x} \infty_x}$ a quelli collezionati.

La costruzione della partizione trapezoidale parte quindi dalla regione R costituita dall'intero piano e poi procede nella suddivisione prendendo in considerazione un segmento alla volta.

Per ogni segmento s_i in input vengono effettuate le seguenti operazioni:

1. Usando la partizione trapezoidale costruita fino a quel momento trova le regioni intersecate dal segmento s_i. Supponiamo che s_i intersechi $t \geq 1$ regioni $R_1, \ldots, R_t$

e che tali regioni siano ordinate da sinistra a destra.

2. Se $t = 1$ il segmento s_i è interno alla regione R_1. Costruiamo quindi le quattro regioni in cui rimane suddivisa R_1 e facciamo puntare i quattro puntatori di R_1 alle regioni così ottenute. Se il segmento è verticale la suddivisione è soltanto in due parti.

3. Se $t > 1$ dividiamo R_1 in tre parti, le regioni intermedie in due parti e infine la regione finale in tre parti. Se suddividendo due regioni consecutive si osserva che esse sono delimitate dallo stesso segmento inferiore si riuniscono le due parti inferiori e se sono delimitate dallo stesso segmento superiore si riuniscono le due parti superiori. Facciamo inoltre puntare i puntatori delle regioni suddivise alle regioni ottenute con la suddivisione.

```
BELOW(s_1, s_2)
    // s_1 ed s_2 segmenti orientati che non si intersecano internamente e che sono
    // comparabili, ossia itersecano una stessa retta verticale.
    // s_1 = p_1q_1 è sotto s_2 = p_2q_2 se e solo se p_2 e q_2 stanno entrambi nel semipiano
    // positivo rispetto alla retta per s_1 oppure p_1 e q_1 stanno entrambi nel semipiano
    // negativo rispetto alla retta per s_2.
1   d_1 = ANGLE-LEFT(p_1, q_1, p_2)
2   d_2 = ANGLE-LEFT(p_1, q_1, q_2)
3   d_3 = ANGLE-LEFT(p_2, q_2, p_1)
4   d_4 = ANGLE-LEFT(p_2, q_2, q_1)
5   return (d_1 ≥ 0 and d_2 ≥ 0) or (d_3 ≤ 0 and d_4 ≤ 0)
```

Programma 10: Il test se un segmento si trova sotto ad un altro segmento.

```
REGION-INTERSECTS(s, R)
    // s = p̄q segmento orientato ed R regione trapezoidale.
1   r = R.left, t = R.right
    // r e t punti di intersezione con l'asse delle x delle rette verticali che delimitano R.
2   if p_x t_z ≥ t_x p_z or q_x r_z ≤ r_x q_z return FALSE
    // Altrimenti s interseca la striscia verticale delimitata dalle rette verticali per r e t.
3   return BELOW(s, R.up) and BELOW(R.down, s)
```

Programma 11: La procedura REGION-INTERSECTS.

La realizzazione del primo punto è analoga alla ricerca della regione trapezoidale contenente un punto p e procede nel modo seguente:

1. Si parte con la lista $[R]$ dove R è l'intero piano.

2. Finché la lista $[R_1, \ldots, R_t]$ contiene qualche regione R che è stata ulteriormente suddivisa seleziona, tra le regioni in cui essa è stata suddivisa, quelle che intersecano il segmento s_i e inseriscile nella lista al posto di R eliminando possibili duplicazioni dovute alle riunioni di parti contigue.

Il controllo se una regione trapezoidale è intersecata da un segmento si può fare in tempo costante usando i prodotti vettoriali.

Il Programma 11 è lo pseudo codice della procedura REGION-INTERSECTS che controlla se un segmento interseca una regione trapezoidale. Esso usa la procedura BELOW il cui pseudocodice è riportato nel Programma 10 per controllare se un segmento stà sotto un altro segmento.

Esercizio 16 Cosa succede se tra i segmenti usati per la costruzione di una partizione trapezoidale ci sono anche segmenti che si intersecano agli estremi (ma non internamente)? Cosa succede se ci sono segmenti verticali? Dire se la procedura funziona anche in questi casi e se non funziona suggerire le modifiche da apportare.