

Elementi di Geometria Computazionale



Dove si introduce la geometria computazionale e si studiano brevemente i suoi problemi più famosi ed interessanti e si delineano i collegamenti con la grafica al calcolatore

- *Preliminari*
- *Intersezioni di segmenti*
- *Inviluppo convesso*
- *Triangolazioni*
- *Ricerca geometrica*
- *Rappresentazione di regioni*

Preliminari

- Cosa è la *geometria computazionale*?
- Tentiamo la seguente definizione

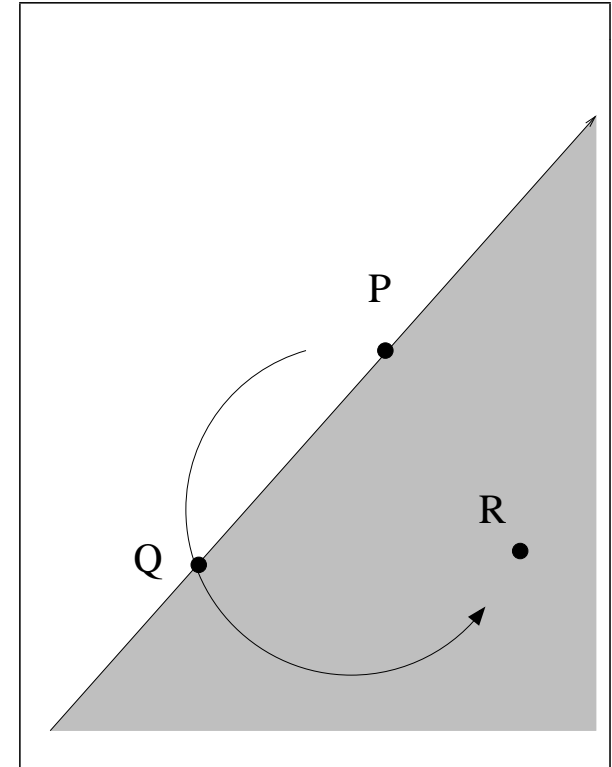
La geometria computazionale studia gli *algoritmi* e le *strutture dati* atti a risolvere problemi di natura *geometrica*, con attenzione ad algoritmi *esatti* e computazionalmente efficienti.

- Il fatto che si preferiscano algoritmi esatti esclude le divisioni e le operazioni trigonometriche, che sono affette da errori di arrotondamento (ed anche computazionalmente onerose). Idealmente vorremmo usare solo addizioni, sottrazioni, moltiplicazioni e confronti.
- L'input ad un problema di geometria computazionale è tipicamente una descrizione di un insieme di oggetti geometrici (punti, linee, poligoni...) e l'output è la risposta ad una domanda che coinvolge gli oggetti (es. intersezione) oppure un nuovo oggetto geometrico (es. guscio convesso),
- ha applicazioni in Grafica, robotica, CAD e GIS.

- Cosa c'entra con la grafica al calcolatore?
- La grafica 3D usa *strutture geometriche* per rappresentare il mondo ed il processo di generazione di una immagine (*rendering*) comprende algoritmi che operano su di essa.
- Noi cercheremo di
 1. Vedere alcuni esempi paradigmatici, tratti dai “classici” della geometria computazionale, per cogliere i meccanismi di base e le idee generali (si potrebbe fare un corso solo di Geometria Computazionale)
 2. Vedere alcuni esempi che useremo direttamente o che vengono usati tipicamente in applicazioni di grafica al calcolatore
- L'approccio sarà di tipo assolutamente non rigoroso. ci limiteremo ad una rassegna descrittiva.

Orientamento di una terna di punti

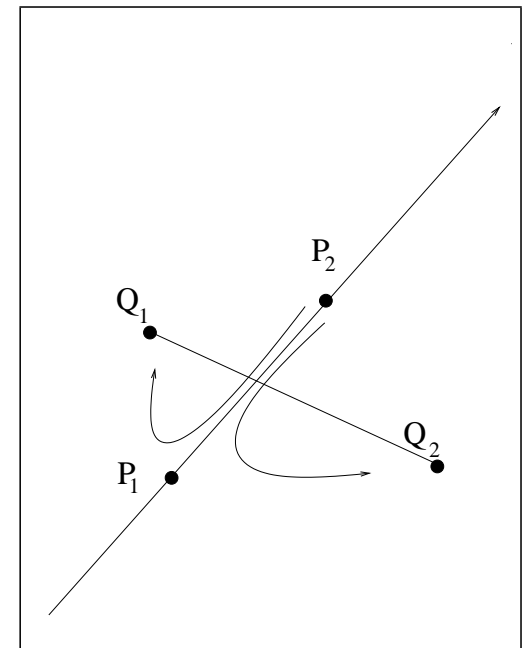
- Risulta utile introdurre una funzione di tre punti (utilizzata anche nel seguito) che chiamiamo *orientamento* o *ordine* di tre punti
- L'ordine di tre punti P , Q ed R – $ord(P, Q, R)$ – è pari a 1 se seguendo il percorso PQR si gira in senso antiorario, è pari a -1 se si gira in senso orario ed è 0 se i punti sono allineati
- $ord(P, Q, R)$ dice da che parte si trova R rispetto alla retta passante per P e Q ; per calcolarlo basta sostituire le coordinate di R nell'equazione della retta passante per P e Q , e guardare il segno
- L'espressione che si ottiene è un determinante, e si può ricavare anche nel seguente modo:
- dati due vettori nel piano $\mathbf{v} = (v_1, v_2)$ e $\mathbf{u} = (u_1, u_2)$, definiamo $\det(\mathbf{v}, \mathbf{u}) = v_1 u_2 - v_2 u_1$. Se $\det(\mathbf{v}, \mathbf{u})$ è positivo, allora \mathbf{u} deve ruotare in senso orario attorno all'origine per raggiungere \mathbf{u} (seguendo la via più breve)



- Consideriamo i due vettori $\mathbf{u} = P - Q$ e $\mathbf{v} = R - Q$: per calcolare $ord(P, Q, R)$ è sufficiente calcolare $\det(v, u)$ e prenderne il segno.
- **Complessità:** costante ($O(1)$) poiché il numero di operazioni è fissato.

Determinare se due segmenti in due dimensioni si intersecano

- Definizione: un segmento è a cavalcioni (straddles) di una linea se i suoi estremi giacciono da parti opposte rispetto alla linea
- per esempio, il segmento (Q_1, Q_2) è a cavalcioni della retta passante per P_1 e P_2 se Q_1 e Q_2 giacciono da parti opposte della retta, ovvero se $ord(P_1, P_2, Q_1)$ e $ord(P_1, P_2, Q_2)$ hanno segno opposto
- è facile verificare che due segmenti si intersecano se e solo se ciascuno è a cavalcioni della retta contenente l'altro



- dunque due segmenti (P_1, P_2) e (Q_1, Q_2) si intersecano se il segmento (Q_1, Q_2) è a cavalcioni della retta passante per P_1 e P_2 e contemporaneamente il segmento (P_1, P_2) è a cavalcioni della retta passante per Q_1 e Q_2 ovvero se le due quantità $(ord(P_1, P_2, Q_1) \cdot ord(P_1, P_2, Q_2))$ e $(ord(Q_1, Q_2, P_1) \cdot ord(Q_1, Q_2, P_2))$ sono entrambe negative.
- prima di procedere a questo test è opportuno effettuare una (rapida) verifica sulle intersezione delle rispettive bounding box.
- **Complessità:** costante ($O(1)$) poiché il numero di operazioni è fissato.
- E per trovare il punto di intersezione? Basta risolvere il seguente sistema di equazioni lineari:

$$(1 - \alpha)P_1 + \alpha P_2 = (1 - \gamma)Q_1 + \gamma Q_2$$

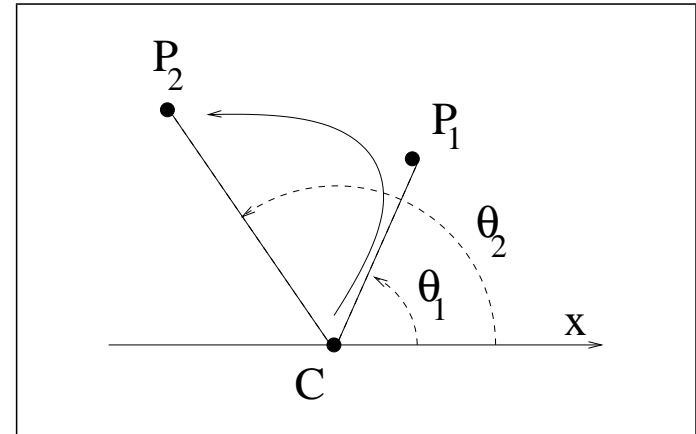
e trovare α e γ (entrambi devono appartenere a $[0, 1]$)

Determinare la posizione di un punto rispetto ad un poligono convesso

- Il teorema di Jordan ci assicura che un poligono (in generale una curva semplice e chiusa) partiziona il piano in due regioni distinte: interno ed esterno.
- Dato un poligono convesso rappresentato dalla sequenza dei suoi vertici $P_1 \dots P_n$ ordinata in senso *antiorario* e dato un punto Q , ci chiediamo se Q giace all'interno o all'esterno del poligono.
 - Q giace dentro il poligono se e solo se Q giace a sinistra di tutte le rette orientate passanti per i lati del poligono (la retta orientata da P_i a $P_{i+1} \forall i = 1 \dots n$)
 - Q giace sul contorno del poligono se Q è allineato con almeno una delle rette sopra menzionate, e giace a sinistra di tutte le altre
- basta dunque usare $ord(P_i, P_{i+1}, Q)$ per scoprire da che parte giace Q rispetto alla retta
- **Complessità:** nel caso peggiore (punto interno) devo controllare tutti i vertici. Il test ha costo costante, quindi $O(n)$

Confronto di angoli

- dato un sistema di riferimento polare, formato da un asse x e da un centro C , e dati due punti P_1 e P_2 determinare quale dei due ha una coordinata angolare maggiore (*senza* calcolarla).
- si può vedere facilmente che $\theta_1 < \theta_2$ se e solo se C, P_1, P_2 è una terna antioraria (ovvero $ord(C, P_1, P_2) = 1$)



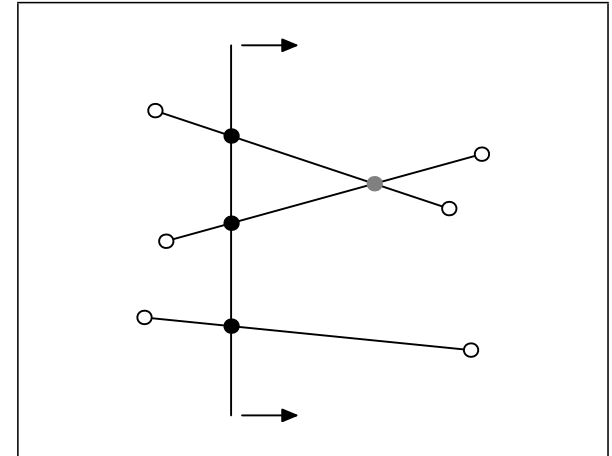
Intersezioni di segmenti

- Il problema che consideriamo è il seguente: dati n segmenti di retta nel piano, determinare tutti i punti in cui una coppia di segmenti si interseca.
- L'applicazione principale per quanto riguarda la grafica è la ricerca dell'intersezione tra poligoni, problema che compare, come vedremo, in un numero svariato di situazioni. Poiché i poligoni sono collezioni di segmenti, l'intersezione tra poligoni si riduce alla ricerca dell'intersezione tra un certo numero di segmenti
- l'algoritmo di forza bruta richiede $O(n^2)$ tempo, poiché considera ciascuna coppia di segmenti. Se tutti i segmenti si intersecano è ottimo. Nella pratica, ciascun segmento ne interseca pochi altri.
- Supponendo che il numero totale di intersezioni sia I , vediamo un algoritmo, denominato *plane sweep*, che risolve il problema in $O(n \log n + I \log n)$ (non lo dimostreremo)
- l'idea di fondo è di evitare di fare il test di intersezione per segmenti che sono lontani, e restringere il più possibile i candidati

Algoritmo plane sweep

- Per non complicare l'esposizione supporremo vere le seguenti condizioni semplificatrici:
 1. Non ci sono segmenti verticali
 2. I segmenti si possono intersecare in un solo punto
 3. In un punto si possono intersecare al più 2 segmenti
- **Idea 1:** Se le proiezioni di due segmenti sull'asse x non si sovrappongono, allora sicuramente i due segmenti non si intersecano.
- basterà allora controllare solo coppie di segmenti le cui proiezioni sull'asse x si sovrappongono, ovvero per i quali esiste una linea verticale che li interseca entrambe
- per trovare tali coppie simuleremo il passaggio di una linea verticale da sinistra a destra che "spazzerà" l'insieme dei segmenti che vogliamo analizzare la (*linea di sweep*)
- **Idea 2:** questo però non basta per ridurre a complessità: per includere anche la nozione di vicinanza nella direzione y , i segmenti che intersecano la sweep line vengono ordinati dall'alto al basso lungo la sweep line, e verranno controllati solo segmenti adiacenti secondo questo ordine.
- La metodologia introdotta da questo algoritmo ricorre più volte nell'ambito della grafica al calcolatore 3D; ne vedremo nel seguito alcuni esempi.

- Lo stato della linea di sweep è la sequenza dei segmenti che la intersecano ordinata secondo la coordinata y della intersezione.
- Lo stato cambia in corrispondenza di un evento, ovvero quando la sweep line raggiunge *un estremo di un segmento* (informazione inserita a priori), o quando incontra *un punto di intersezione tra due segmenti* (informazione inserita man mano che l'algoritmo procede)
- La sweep line non si muove con continuità ma su un insieme di posizioni discrete, corrispondenti agli eventi.
- In questi punti si calcolano le intersezioni, viene aggiornato lo stato della linea di sweep e la coda degli eventi.
- il calcolo delle intersezioni coinvolge solo i segmenti adiacenti nello stato della sweep line



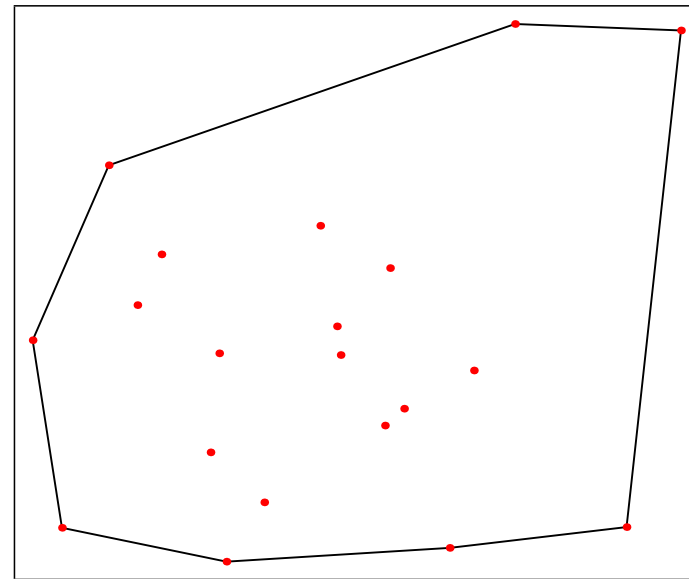
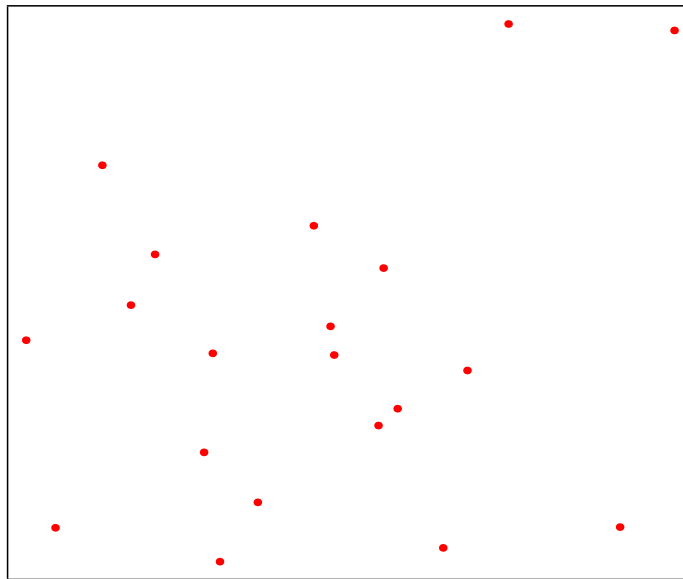
- **Aggiornamento dovuto all'evento:** quando la linea incontra un evento si devono aggiornare due opportune strutture dati
 - **La coda degli eventi:** si tratta di una lista di eventi “futuri” noti (cioè che la linea non ha ancora incontrato) ordinati per x crescente. Ogni elemento della coda deve specificare che tipo di evento sia e quali segmenti coinvolge. Tale struttura dati deve permettere l’inserzione di nuovi eventi (non ancora presenti nella coda) e l’estrazione dell’evento iniziale della coda. Non si può usare uno heap perché l’inserimento deve poter controllare se l’evento è già presente. Si può usare un albero binario bilanciato.
 - **Stato della linea di sweep:** in tale struttura dati si deve mantenere la lista di segmenti su cui la linea incide ordinati per y decrescente. Su tale struttura si deve poter eliminare un segmento dalla lista, inserirne uno nuovo, scambiare l’ordine di due segmenti consecutivi e determinare il precedente ed il successivo di ciascun elemento della lista. Inoltre gli elementi della lista dovrebbero mantenere l’informazione (che cambia con il muoversi della linea) della y di intersezione. Si può usare, di nuovo, un albero binario bilanciato.

Vediamo quindi l'algoritmo completo:

- Siano $\{s_1, \dots, s_n\}$ gli n segmenti da analizzare, rappresentati mediante gli estremi.
- Si parte con la linea tutta a sinistra, e si inseriscono nella coda degli eventi tutti gli estremi dei segmenti $\{s_1, \dots, s_n\}$; lo stato della linea di sweep è vuoto. Nel seguito ogni volta che si fa un test di intersezione, se il risultato è positivo allora si inserisce l'intersezione nella coda degli eventi.
- Fin tanto che la coda degli eventi è non vuota si estrae l'evento successivo e si guarda di che tipo è:
 - ***Estremo sinistro di un segmento:*** in tal caso si aggiunge il segmento allo stato della linea di sweep (guardando la sua y e posizionandolo di conseguenza) e si fa un test di intersezione con il segmento immediatamente sopra e con quello immediatamente sotto
 - ***Estremo destro di un segmento:*** in tal caso si elimina il segmento dallo stato della linea di sweep e si fa un test di intersezione tra il segmento immediatamente sopra e quello immediatamente sotto
 - ***Punto di intersezione:*** si scambiano i due segmenti coinvolti nell'intersezione e per quello che finisce sopra si fa un test di intersezione con il suo precedente, mentre per quello che finisce sotto si fa un test con il suo successivo.

Inviluppo convesso

- Veniamo ora ad un altro problema fondamentale della geometria computazionale che ha notevoli applicazioni nella grafica al calcolatore
- Ricordo che una regione \mathcal{O} dello spazio (affine) si dice **convessa** se per ogni coppia di punti P_1 e P_2 appartenenti a \mathcal{O} si ha che $P' = \alpha(P_1 - P_2) + P_2$ appartiene a \mathcal{O} per ogni $\alpha \in [0, 1]$ ovvero tutti i punti sul segmento che unisce P_1 con P_2 appartengono alla regione data.
- Definiamo **inviluppo convesso** di un insieme di punti $\{P_i\}$ nello spazio affine come la più piccola regione convessa che contiene tutti i punti dati



Applicazioni

- I politopi (poligoni in n dimensioni) convessi sono le figure più semplici da trattare
- L'involuppo convesso è la più semplice “approssimazione” di un insieme di punti
- L'uso dell'involuppo convesso alle volte può aiutare ad eliminare operazioni inutili.
- **Esempio:** un problema tipico di applicazioni interattive di grafica è il calcolo delle collisioni tra oggetti anche complessi. Se non si richiede una precisione elevata (caso frequente) si può usare l'involuppo convesso per calcolare le collisioni, semplificando molto il problema.

Algoritmi per il calcolo dell'involuppo convesso

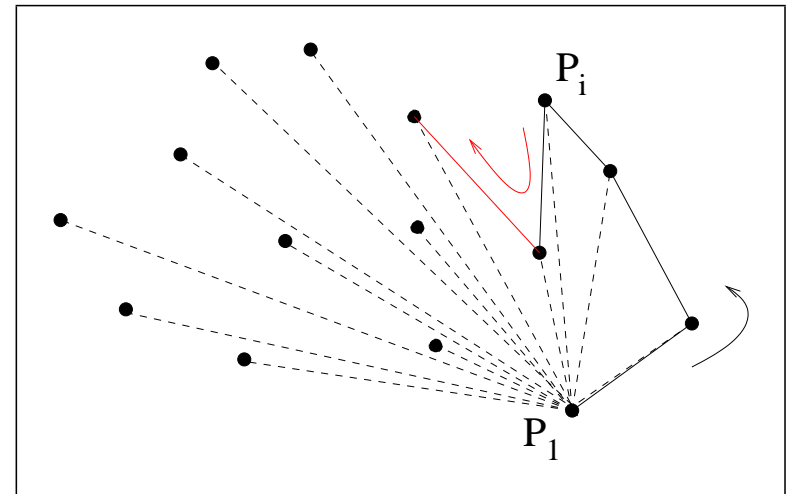
Descriviamo ora brevemente quattro algoritmi comuni per il calcolo dell'involuppo convesso di un insieme di punti; per semplicità presenteremo gli algoritmi per il caso bidimensionale, ma sono tutti estendibili in maniera indolore (o quasi) al caso delle tre dimensioni. Tutti gli algoritmi costruiscono l'involuppo convesso in modo progressivo. Quando dirò che un punto appartiene all'involuppo convesso, intenderò (se non specificato diversamente) al *bordo* di questo.

Algoritmo diretto

- Il modo più diretto per calcolare l'involuppo convesso è anche il più inefficiente.
- L'algoritmo è il seguente
 - Per ogni coppia di punti $P_i P_j$ dell'insieme si calcola $ord(P_i, P_j, P_k)$ per ogni $k \neq i, j$
 - Se tale valore è positivo $\forall k$ allora i due punti appartengono all'involuppo convesso
- **Complessità:** si considerano $n^2 - n$ coppie di punti, e per ciascuna coppia si considerano nel test gli altri $n - 2$ punti, dunque la complessità temporale è $O(n^3)$

Graham's Scan

- Vediamo ora un algoritmo più efficiente che permette di calcolare l'involucro convesso
- l'algoritmo è incrementale, ovvero aggiungiamo un punto alla volta ed aggiorniamo la soluzione dopo ogni aggiunta.
 - ordino i i punti per coordinate polari crescenti rispetto al punto di ordinata minima.
 - i punti vengono aggiunti uno alla volta, in senso orario, ed ogni volta il guscio viene aggiornato.

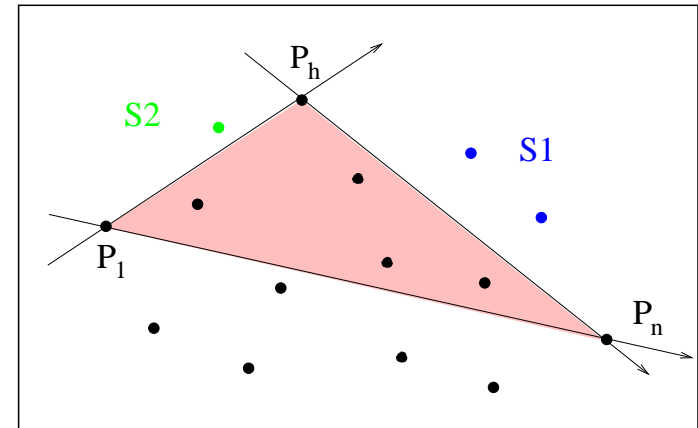


- L'osservazione chiave per processare i punti è che percorrendo i vertici di un poligono convesso in senso orario, si fanno solo svolte a destra. Ovvero: *l'ordine di tre vertici consecutivi di un poligono convesso è negativo.*
- **Variante:** ordino i punti per x crescente, e li aggiungo da sinistra a destra.
- Inserendo i punti in quest'ordine ottengo solo la metà superiore del guscio (upper hull). Quella inferiore si ottiene analogamente con un'altra passata.

- Se più punti hanno la stessa x si può perturbare leggermente la distribuzione di punti iniziale in modo da eliminare la degenerazione
- Oppure si può ordinare i punti per x crescente e per y crescente (lessicografico). Questo è un esempio di *perturbazione simbolica*, ovvero si evince l'effetto di una perturbazione *senza* applicare realmente la perturbazione.
- **Complessità:** dimostriamo che è $O(n \log n)$
 - Per ordinare gli n punti la complessità è $O(n \log n)$
 - Per ogni punto aggiunto durante la procedura la complessità è $O(A_i + 1)$, dove A_i sono i numeri di punti che devo eliminare durante quel passo
 - Sommando per tutti i punti si ha $\sum_i (A_i + 1) = n + \sum_i A_i$
 - Siccome ciascun punto può essere al più cancellato una volta sola, si ha $\sum_i A_i \leq n$ e dunque la complessità computazionale della fase di aggiunta dei punti è $O(n)$
 - Il discorso si ripete per la parte inferiore dell'involuppo convesso e quindi si ha infine $O(2n) = O(n)$
 - Il termine dominante è il primo (l'ordinamento), dunque la complessità computazionale totale del metodo è $O(n \log n)$
- **Vantaggio e svantaggio:** è semplice ed è ottimale, però non è generalizzabile al 3D

Quickhull

- Si considerano i punti di ascissa minima e massima, P_1 e P_n . La retta che li congiunge partiziona l'insieme dei punti in due sottoinsiemi, che verranno considerati uno alla volta.
- Sia dunque S l'insieme dei punti che stanno sopra la retta orientata da P_1 a P_n .
- P_1 ed P_n appartengono all'involucro convesso
- sia P_h il punto di massima distanza dalla retta passante per P_1 ed P_n . Si vede facilmente che P_h appartiene al guscio convesso.
- consideriamo le due rette orientate (P_1, P_h) e (P_h, P_n) .
 - non esistono punti a sinistra di entrambe (perché P_h appartiene al guscio convesso)
 - quelli che giacciono a destra di entrambe sono interni al triangolo $P_1P_nP_h$ e si possono eliminare dalla considerazione (perché non appartengono al guscio convesso)
 - quelli che giacciono a destra di una retta ed a sinistra dell'altra costituiscono i due insiemi $S1$ ed $S2$ che sono esterni ai lati P_hP_n ed P_1P_h , rispettivamente, dell'attuale involucro
- la procedura si attiva ricorsivamente prendendo $S1$ ed $S2$ separatamente come S .



- **Complessità:** L'analisi della complessità è analoga a quella che si fa per il Quicksort. Il caso ottimo si ha quando i due insiemi S_1 ed S_2 hanno circa la stessa cardinalità ed in tal caso la computazione richiede tempo $O(n \log n)$, ma nel caso peggiore richiede $O(n^2)$.
 - Il costo del calcolo del punto P_h ed il partizionamento dell'insieme S è $O(m)$ dove m è la cardinalità dell'insieme S corrente.
 - Sia $T(n)$ il costo del calcolo del quickhull su un insieme S di n punti. $T(n)$ soddisfa la seguente ricorrenza:

$$T(0) = 1$$

$$T(n) = T(n_1) + T(n_2) + n$$

dove n_1 ed n_2 sono le cardinalità di S_1 ed S_2 rispettivamente.

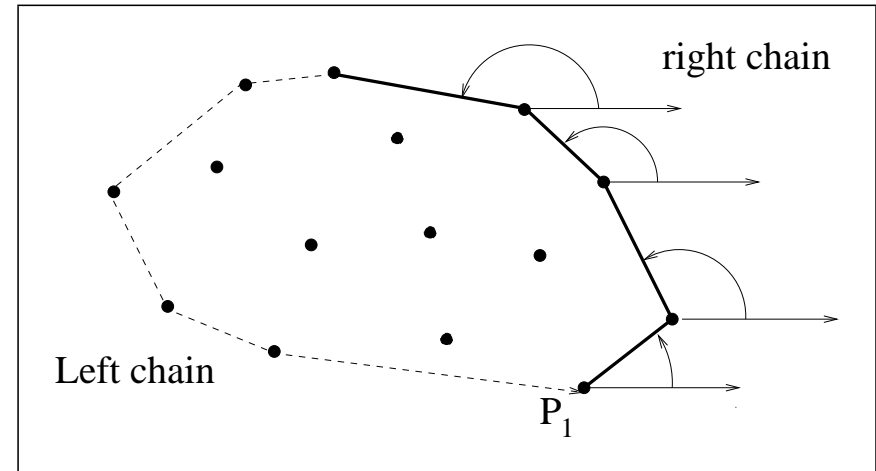
- Se assumiamo $\max(n_1, n_2) < \alpha n$ per un certo $\alpha < 1$, la soluzione è $O(n \log n)$, con una costante nascosta che dipende da α (è facile se si prende $\alpha = 1/2$, come nel caso ottimo di Quicksort.)
- **Vantaggio e svantaggio:** molto semplice da implementare, parallelizzabile, e nella pratica è veloce.

Jarvi's March

- L'ultimo algoritmo che vediamo prende anche il nome di *impacchettamento del regalo* (gift-wrapping) per ovvi motivi.

- Si tratta di partire da un punto che si sa appartenere all'involuppo convesso, per esempio il punto con y minima, P_1 .

- Quindi si traccia la retta che passa per tale punto e la si comincia a ruotare fino a quando non incontra un punto



- Si aggiunge tale punto all'involuppo convesso e si procede in modo analogo fino a quando non si è tornati al punto di partenza
- In pratica, basta prendere come punto successivo il punto che forma l'angolo più piccolo con l'asse delle x .
- Nota: non occorre calcolare l'angolo, si usa l'orientamento di tre punti.
- In questo modo arrivo fino al punto di y massima (right chain). Per calcolare la left chain riparto da P_1 ma considero l'asse x invertito.

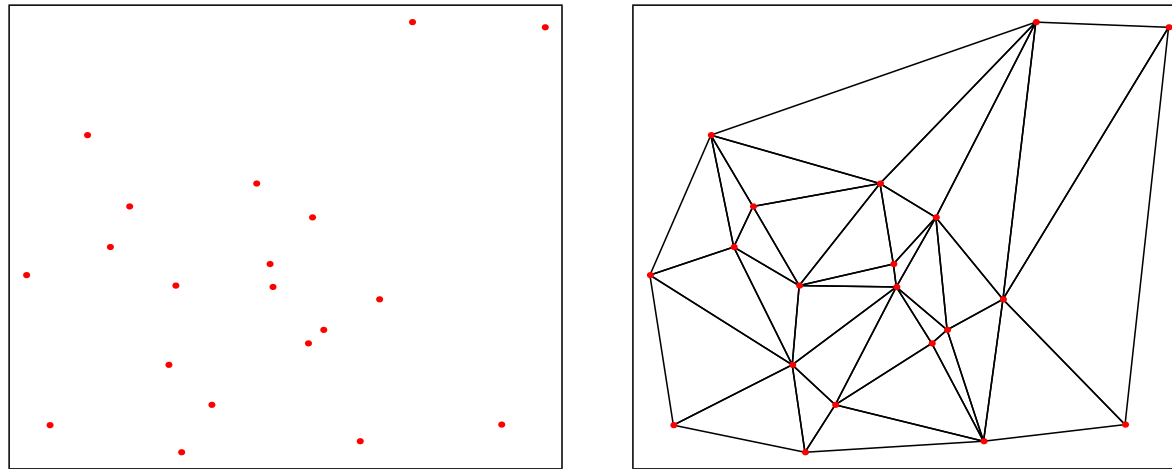
- **Complessità:** il costo della determinazione del punto successivo è $O(n)$, e questo viene fatto per ogni vertice del guscio convesso. Se h Sono i vertici dell'involuppo convesso, allora la complessità computazionale dell'algoritmo è $O(nh)$. Nel caso pessimo si arriva a $O(n^2)$ poiché si ha $h = O(n)$
- **Importante:** Il metodo si generalizza a più di 3 dimensioni.

Triangolazioni

Suddivisione poligonale del piano

- Una suddivisione poligonale del piano è un insieme \mathcal{P} (infinito) di poligoni convessi che soddisfanno le seguenti proprietà:
 - Dati due poligoni qualsiasi di \mathcal{P} , se la loro intersezione è non nulla allora è uguale ad un lato per entrambi, ovvero i poligoni si toccano al più lungo un lato a comune
 - L'unione di tutti i poligoni coincide con il piano (non ci sono buchi)
- Per non dover trattare con gli infiniti, cosa che non si fa mai nella pratica, conviene considerare una porzione finita di piano; una sua suddivisione poligonale è allora costituita da un insieme finito di poligoni convessi

- Un particolare (ed importante) tipo di suddivisione poligonale è costituito dalle cosiddette *triangolazioni* del piano, ovvero suddivisioni poligonali i cui elementi sono triangoli
- Una triangolazione di un insieme V di punti nel piano è una triangolazione i cui vertici sono i punti di V .

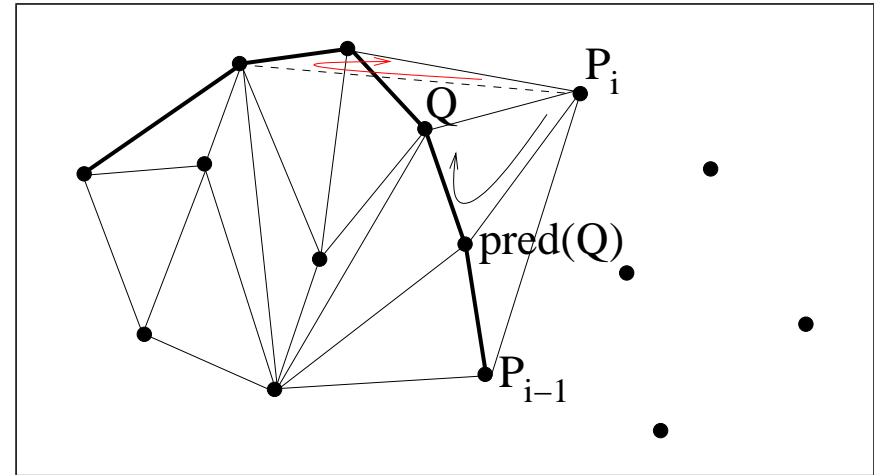


- Sono utili perché il triangolo è il poligono convesso più semplice
- Una suddivisione poligonale qualsiasi è sempre triangolabile aggiungendo opportunamente dei lati (questo deriva direttamente dal fatto di poter sempre decomporre un poligono convesso in un numero finito di triangoli)

Algoritmo di triangolazione plane sweep

- Si tratta di un algoritmo incrementale per il calcolo di una triangolazione, che impiega una tecnica di plane sweep e che ha qualche analogia con il Graham's scan.
- si ordinano i punti di V secondo l'ascissa crescente
- si costruisce il triangolo formato dai primi tre punti
- si aggiungono uno alla volta i punti e si aggiorna la triangolazione di conseguenza.
- il passo fondamentale è come unire P_i alla triangolazione formata dai punti $P_1 \dots P_{i-1}$, ovvero con quali punti della triangolazione corrente unirlo (formando un nuovo spigolo).
- P_i deve essere unito a ciascun punto P_j per $j < i$ che sia “visibile” da P_i , ovvero tale che il segmento (P_i, P_j) non interseca la triangolazione.
- per effettuare in modo efficiente questo passo, bisogna tenere presente che:
 - P_j deve appartenere al guscio convesso dei primi $i-1$ punti
 - P_{i-1} è sempre visibile da P_i

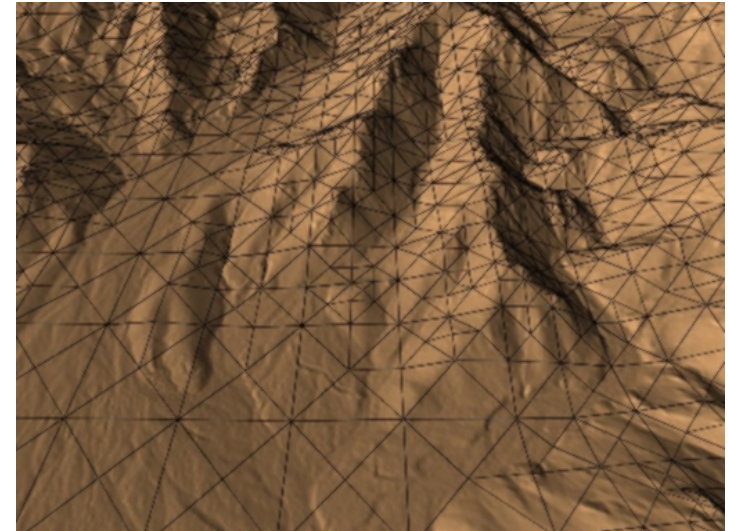
- **Soluzione:** muoversi lungo il guscio convesso a partire da P_{i-1} in senso orario ed antiorario ed unire i punti del guscio convesso a P_i finché non si incontra un punto non visibile. L'ultimo punto visibile incontrato prende il nome di punto di tangenza.



- Il test di visibilità si effettua controllando l'ordinamento della terna P_i , $pred(Q)$, Q , dove Q è un punto sul guscio convesso e $pred(Q)$ è il punto che lo precede (in senso antiorario)
- Quando si aggiorna la triangolazione in seguito alla introduzione di un punto, bisogna anche aggiornare il guscio convesso, rimuovendo tutti i punti visibili da P_i esclusi i due punti di tangenza, e inserendo P_i tra i due
- **Complessità:** costo dominato dall'ordinamento dei punti: $O(n \log n)$
- La triangolazione che si ottiene è una delle possibili triangolazioni. Vedremo che non tutte le triangolazioni sono ugualmente buone, per certi scopi.

Triangolazione di Delaunay

- Se usiamo la triangolazione per rappresentare il dominio di una certa funzione $z(x, y)$ (es. superfici topografiche), i cui valori sono noti solo in corrispondenza dei punti della triangolazione, il valore nei punti interni ai triangoli si ottiene per interpolazione dei valori dei vertici.

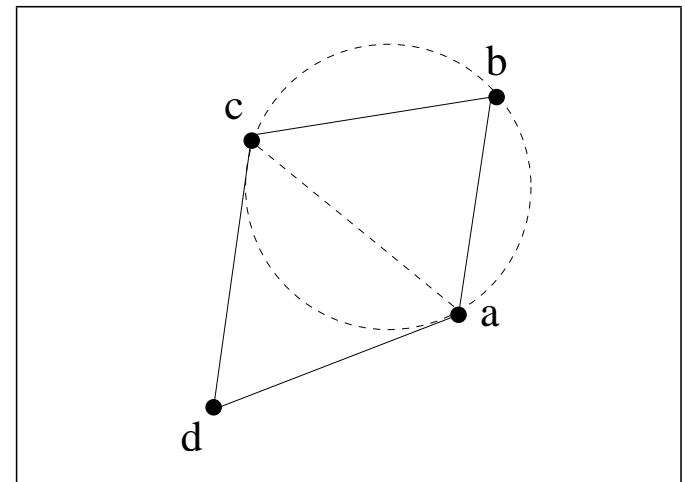


- Vorremmo allora che i vertici fossero tra loro il più possibile vicini, per evitare di interpolare tra valori distanti: questa richiesta si traduce nel cercare di avere angoli il meno acuti possibile.
- Possiamo allora pensare di classificare le triangolazioni del piano in base all'angolo più piccolo, ed a parità di questo guardare il secondo più piccolo, e così via come in un ordinamento lessicografico, definendo in effetti un ordine \mathcal{T} .
- Poiché le triangolazioni sono in numero finito, e le abbiamo ordinate, deve esserci una triangolazione \mathcal{T} -massimale: questa è la triangolazione di Delaunay (francesizzazione di Boris Nikolaevich Delone).

- Si può dare una caratterizzazione ulteriore della triangolazione di Delaunay (di solito è la definizione):
- una triangolazione di un insieme di punti V è di Delaunay se e solo se ogni triangolo è inscritto in una circonferenza *libera*, ovvero tale che non contenga alcun punto di V .
- Formalizziamo il concetto di circonferenza libera definendo un predicato booleano che si applica a quattro punti distinti del piano: $\text{InCircle}(a, b, c, d)$ è vero se e solo se il punto d è interno alla circonferenza orientata abc .
- Si dimostra che dato un quadrilatero $abcd$, $\text{InCircle}(a, b, c, d)$ è equivalente a:

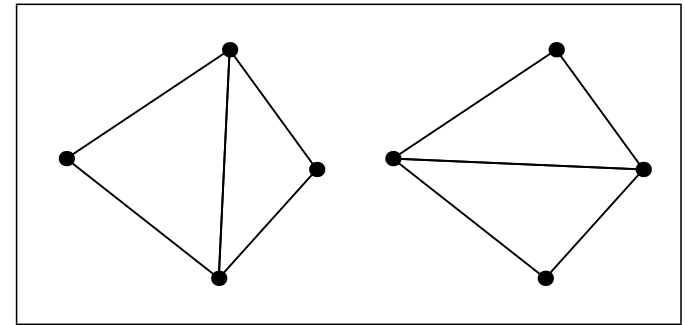
$$\hat{b} + \hat{d} < \hat{c} + \hat{a}$$

- il quadrilatero ammette due possibili triangolazioni: una con la diagonale bd , l'altra con la diagonale ac . Si dimostra che le circonferenze circoscritte ai due triangoli che si ottengono sono libere se e solo se si sceglie la diagonale che connette i due angoli opposti la cui somma è massima, ovvero ac , in questo caso.



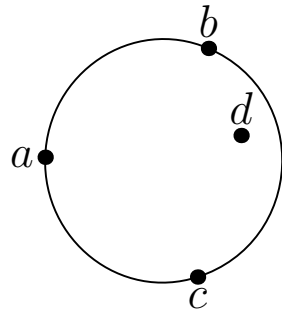
- si dimostra, inoltre, che la scelta di ac massimizza il più piccolo angolo interno dei due triangoli risultanti.

- **Osservazione:** se un triangolo non soddisfa la proprietà di circonferenza libera, è possibile effettuare una operazione locale (edge flip) la quale fa sì che la proprietà sia rispettata ed incrementa il rango della triangolazione in \mathcal{T} .
- **Operazione di edge flip:** dati due triangoli abc e cda che hanno in comune il lato ac , in modo che formano il quadrilatero convesso $abcd$ l'operazione di edge flip elimina il lato ac e crea il nuovo lato bd ; si ottengono così i triangoli abd e bcd .
- Questo suggerisce anche un **algoritmo immediato** (ma inefficiente) per trovare una triangolazione di Delaunay:
 - si parte da una triangolazione qualunque
 - finché esistono triangoli che non soddisfano la proprietà di circonferenza libera si effettuano operazioni di edge flip
 - l'algoritmo termina, perché le triangolazioni sono finite, e la triangolazione finale è \mathcal{T} -massimale poiché ad ogni passo si incrementa il rango della triangolazione in \mathcal{T} .

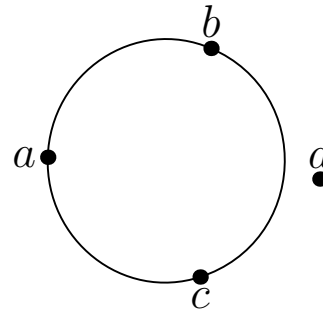


- Per calcolare il predicato $\text{InCircle}()$, sfruttiamo la seguente proprietà:
- Siano a , b , c e d quattro punti sul piano; si dimostra che d è interno al cerchio passante per abc ($\text{InCircle}(a, b, c, d)$ è vero), se e solo se

$$\text{in}(a, b, c, d) = \det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} < 0$$



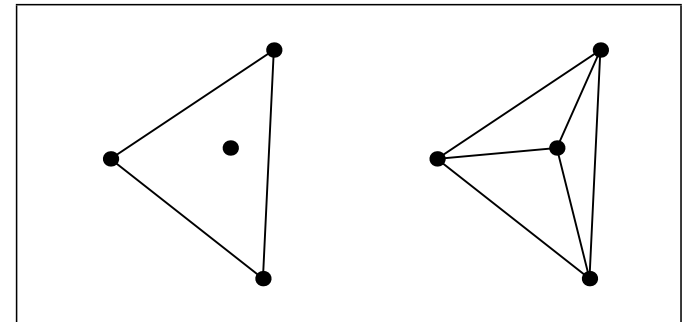
$$\text{in}(a, b, c, d) > 0$$



$$\text{in}(a, b, c, d) < 0$$

Algoritmo GKS

- Vediamo ora un metodo efficiente, *progressivo* e *casuale* per la costruzione della triangolazione di Delaunay di un insieme di punti V , dovuto a Guibas, Knuth, e Sharir (GKS)
- L'algoritmo per la costruzione della triangolazione di Delaunay è progressivo.
- Ad ogni passo si suppone di avere una triangolazione di Delaunay di una parte di V .
- Si aggiunge un vertice a caso non ancora considerato e si modifica la triangolazione usando le due operazioni di inserzione e di edge flip in modo da accomodare tale vertice e da essere ancora una triangolazione di Delaunay
- ***Inserzione di un vertice:*** data una triangolazione aggiungiamo un vertice p ; se tale vertice cade nel triangolo abc aggiungiamo inoltre i lati pa , pb e pc in modo tale da formare tre nuovi triangoli pab , pcb e pca .



- Nel dettaglio
 - Si suppone di avere una triangolazione valida
 - Si aggiunge il vertice p alla triangolazione e sia abc il triangolo in cui cade p
 - Si esegue l'operazione vista sopra, ottenendo 3 nuovi triangoli al posto di abc , ovvero pab , $pbac$ e pca .
 - Per ciascuno dei nuovi triangoli si esegue il test *in* con il vertice d esterno al triangolo e opposto a p
 - Se il test fallisce si esegue un edge flip e si esegue ricorsivamente il test sui due nuovi triangoli
 - Si va avanti ad inserire punti fino a quando non si ottiene la triangolazione di Delaunay di tutto V
- Come si parte? Per esempio con un triangolo opportuno abbastanza grande da contenere tutto V .
- La correttezza dipende dal fatto che:
 - è sufficiente testare triangoli che contengono p , perché p è l'unica perturbazione apporata a una triangolazione già di Delaunay
 - è sufficiente testare il punto d esterno al triangolo e opposto a p perché si dimostra che se esso è esterno alla circonferenza passante per i vertici del triangolo, allora tutti gli altri punti sono esterni

- Quando inserisco un punto devo poter localizzare il triangolo a cui appartiene, tramite una struttura dati opportuna.
- **Complessità:** si dimostra (non facilmente) che la complessità dell'algoritmo è $O(n \log n)$.
- La triangolazione di Delaunay ammette estensioni *multidimensionali*

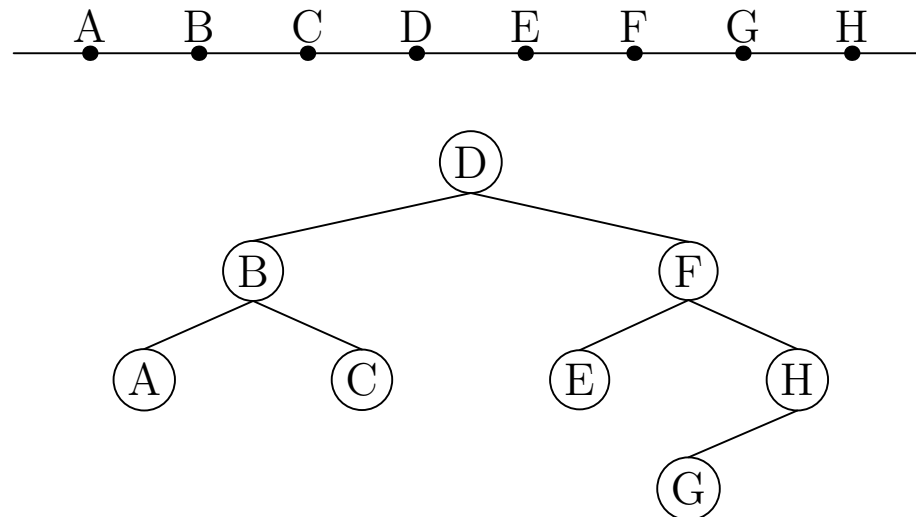
Ricerca Geometrica

- Spostiamo ora la nostra attenzione dagli algoritmi alle strutture dati.
- Studieremo ora un problema che richiede l'impiego di *strutture dati geometriche*.
- I due fattori concorrenti che caratterizzano tali strutture dati sono lo *spazio* che esse occupano (in termini di memoria) ed il *tempo* che richiedono per operare una ricerca.
- Il problema è il seguente: dato un insieme di punti \mathcal{P} , determinare quelli che appartengono ad una data regione dello spazio
- Sebbene il problema sia generalizzabile ad n dimensioni, come vedremo, ci concentreremo sul caso bidimensionale per motivi didattici
- Ci occuperemo inoltre del caso in cui la regione di ricerca sia un rettangolo con i lati paralleli agli assi cartesiani, semplificando così i termini del problema; tale ricerca prende anche il nome di *ricerca ortogonale*.

Ricerca unidimensionale con alberi binari

- Per inquadrare il problema partiamo da un caso molto semplice; la ricerca geometrica in una dimensione
- Sia quindi $\mathcal{P} = \{P_1, \dots, P_n\}$ un insieme di n punti disposti sulla retta
- Vogliamo cercare una struttura dati ideale per effettuare una ricerca geometrica con un intervallo, ovvero la ricerca di quei punti di \mathcal{P} che cadono in un dato intervallo
- Si potrebbe semplicemente ordinare i punti da sinistra a destra e poi cercare il primo punto che giace a destra dell'estremo sinistro dell'intervallo ed il primo punto che giace a sinistra dell'estremo destro; tutti i punti in mezzo sono compresi nell'intervallo
- Tale soluzione è però specifica del caso unidimensionale e non si generalizza a dimensioni maggiori
- Vediamo un altro approccio che sfrutta gli alberi binari

- Si costruisce un albero di ricerca binario con i punti dell'insieme
 - Giusto per ricordare ..
 - L'albero viene costruito scegliendo uno dei punti e ponendolo come *radice*
 - Tutti i punti che cadono a *sinistra* del punto andranno a popolare il *sottoalbero sinistro* della radice, quelli che cadono a *destra* il *sottoalbero destro*
 - Si itera fino a quando tutti i punti sono stati inseriti nell'albero
 - Convieni costruire alberi bilanciati, dove cioè i sottoalberi sinistro e destro di un nodo qualsiasi sono egualmente popolati (o comunque sono poco sbilanciati)
- Una osservazione che servirà nella generalizzazione n-D:
 - ciascun punto è associato ad un intervallo
 - l'inserimento di un punto risulta nella divisione in due di un intervallo preesistente.



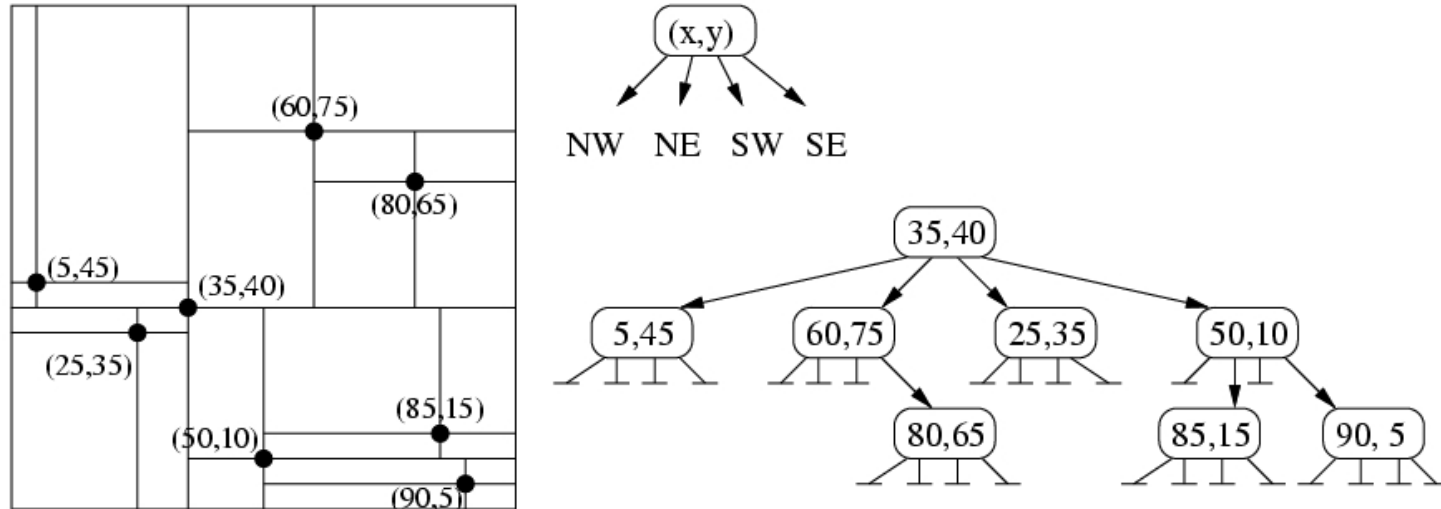
- Supponiamo di voler ricercare i punti che cadono nell'intervallo $\mathcal{I} = [x : x']$
 - Si esegue un test di appartenenza a \mathcal{I} della radice
 - Se il test è *positivo*:
 - * Si mette la radice nella lista dei punti appartenenti a \mathcal{I}
 - * Si itera sul sottoalbero sinistro
 - * Si itera sul sottoalbero destro
 - Se il test è *negativo*:
 - * Se la radice cade a sinistra di \mathcal{I} si itera sul sottoalbero destro
 - * Se la radice cade a destra di \mathcal{I} si itera sul sottoalbero sinistro
 - Le precedenti iterazioni vanno avanti fino a raggiungere i nodi foglia dell'albero

Point Quadtree

- La ricerca geometrica tramite alberi binari si estende in modo naturale al caso di dimensione 2 con i cosiddetti *point quadtree*.
- sono alberi quaternari: ciascun nodo contiene un punto ed ha 4 figli, etichettati NW, NE, SW e SE. Il sottoalbero NW (p.es) contiene tutti i punti che sono a Nord-Ovest di quello contenuto nel nodo radice.
- oltre che contenere un punto, ogni nodo è associato ad una regione rettangolare (come nell'albero binario ciascun nodo era associato ad un intervallo).
- I punti vengono inseriti uno ad uno. L'inserimento di ciascun punto risulta in una suddivisione di una regione rettangolare in 4 rettangoli più piccoli, mediante linee di divisione orizzontali e verticali che passano per il punto inserito.
- Si consideri l'inserimento dei punti:

$(35, 40), (50, 10), (60, 75), (80, 65), (85, 15), (5, 45), (25, 35), (90, 5)$

il risultato è mostrato nella figura seguente:

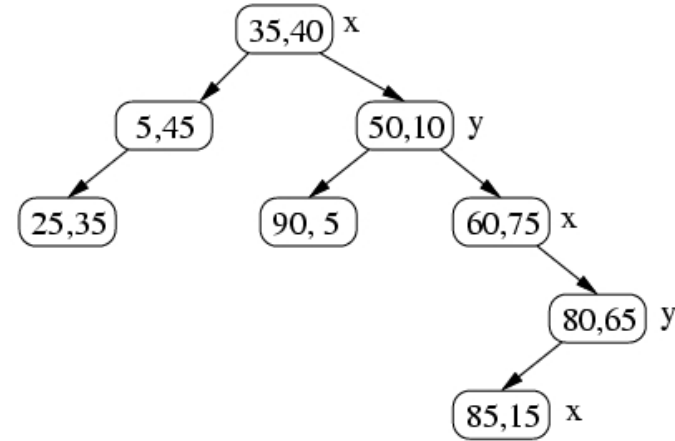
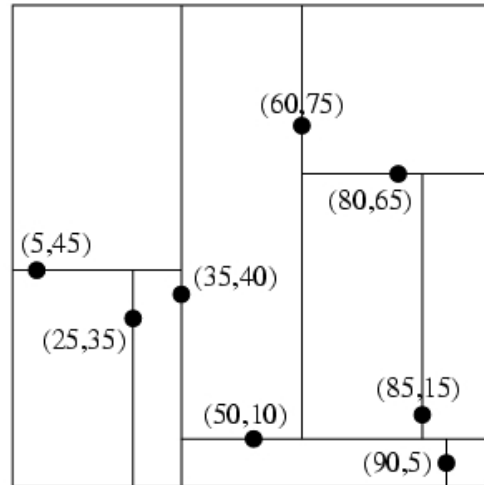


©D. Mount

- La complessità della ricerca in un point quadtree è $O(\sqrt{n})$
- il difetto dei quadtrees è che la “arietà” dell’albero (quanti figli ha ciascun nodo) cresce esponenzialmente con la dimensione. In d dimensioni, ogni nodo ha 2^d figli.
- In 3D ogni nodo ha otto figli (si chiamano *point octrees*).
- Un modo diverso di generalizzare gli alberi binari, che non soffre di questo problema, sono i k-D trees.

Kd-Tree

- L'idea dietro il *k-D tree* è di estendere la nozione di albero binario di ricerca unidimensionale alternando la coordinata lungo cui effettuare il confronto (splitting dimension).
- Come al solito vediamo l'implementazione nel caso di dimensione 2; l'estensione a dimensioni più alte risulta semplice.
- Ciascun nodo è associato ad una regione rettangolare. Quando un nuovo punto viene inserito, la regione si divide in due, tramite una linea verticale oppure orizzontale passante per il punto.
- In principio, oltre alle coordinate del punto, un nodo dovrebbe contenere anche la splitting dimension. Se però alterniamo tra verticale ed orizzontale non serve.
- Il sottoalbero sinistro di un nodo contiene tutti i punti la cui coordinata verticale o orizzontale è minore di quella della radice. Il sottoalbero destro quelli la cui coordinata è maggiore.
- Si consideri l'inserimento degli stessi punti dell'esempio precedente. Il risultato è mostrato nella figura seguente:



©D. Mount

- Si abbia quindi un insieme \mathcal{P} di punti in due dimensioni su cui si vogliono operare ricerche di appartenenza in regioni rettangolari (orthogonal range query)
- Il 2d-tree si costruisce nel seguente modo
 - Si seleziona un punto appartenente a \mathcal{P} e lo si pone come radice di un albero binario
 - Si traccia una retta orizzontale che passa per tale punto
 - Tutti i punti che cadono sopra tale retta vengono posti nel sottoalbero sinistro, quelli che cadono sotto nel sottoalbero destro
 - Si itera, ma al passo successivo si usano rette orizzontali al posto di rette verticali e così via, alternando suddivisioni con rette orizzontali e rette verticali

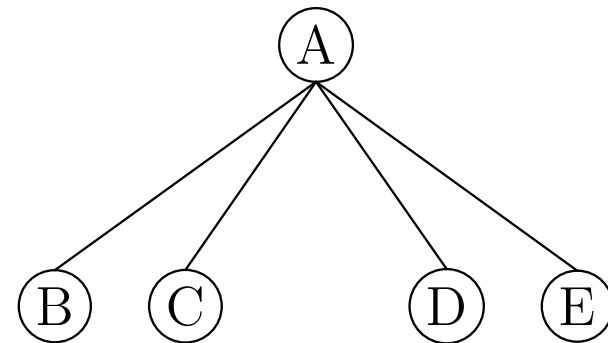
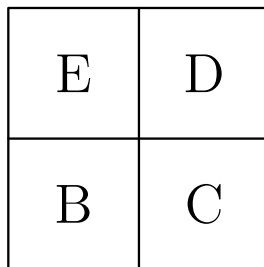
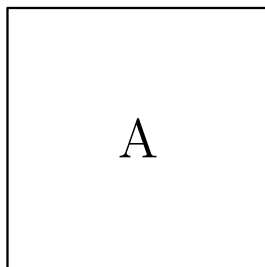
- La ricerca tramite kd-tree dei punti che cadono in un intervallo rettangolare \mathcal{I} segue a questo punto lo stesso modello di quella unidimensionale già vista.
- Si parte dal nodo radice e si esegue un test per vedere se \mathcal{I} interseca la retta associata al nodo
 - Nel caso *positivo* allora si vede se il nodo radice appartiene a \mathcal{I} e si scende successivamente lungo entrambi i sottoalberi destro e sinistro
 - Nel caso *negativo* allora si scende nel sottoalbero dalla parte in cui giace \mathcal{I}
- Come per un albero binario di ricerca, se gli n elementi vengono inseriti a caso, il valore atteso dell'altezza dell'albero è $O(\log n)$. Il costo temporale per costruire un k-D tree è $O(n \log n)$. La complessità computazionale della ricerca è data da $O(k + \sqrt{n})$, dove k è il numero di punti che cadono in \mathcal{I} (non è immediato da dimostrare).
- Una possibile generalizzazione dei k-D trees rimuove il vincolo che il partizionamento dello spazio avvenga lungo rette ortogonali. Nei BSP trees, le linee di suddivisione sono generici iperpiani (che dividono lo spazio in due). Questi però non sono utili a risolvere problemi di ricerca ortogonale, naturalmente.

Rappresentazione di regioni

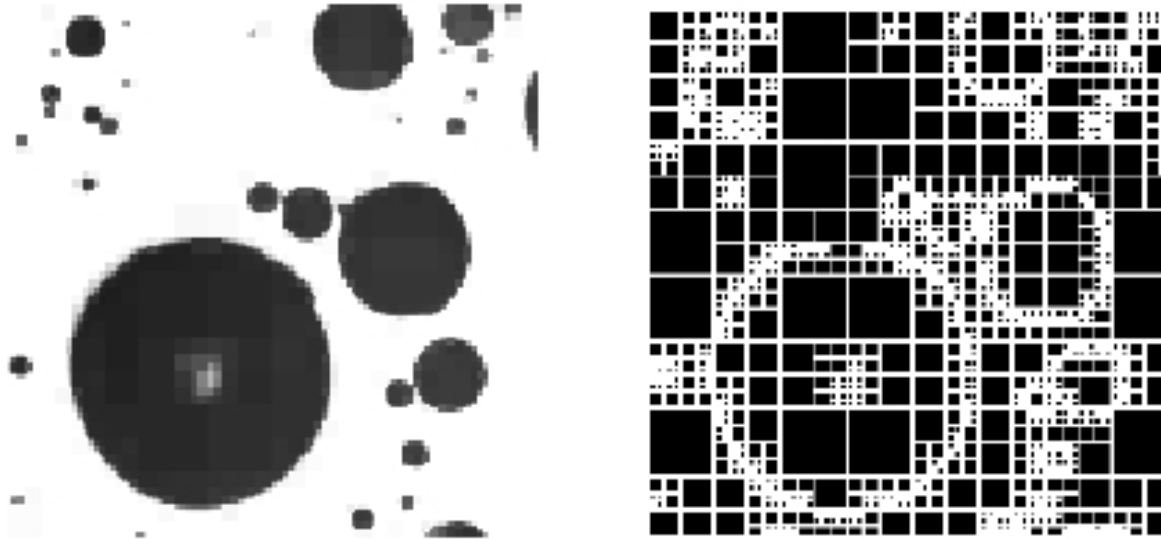
- Continuiamo a parlare di strutture dati geometriche.
- Quelle viste nei lucidi precedenti riguardavano la rappresentazione di insiemi di punti (point data).
- Vediamo ora suddivisioni ricorsive dello spazio possano essere utilmente impiegati per rappresentare regioni e volumi (region data)
- In particolare, parleremo di *region quadtrees*, *region octrees* e *Binary Space Partition tree*.
- Consideriamo regioni rappresentate in base allo spazio occupato (si potrebbero anche rappresentare mediante il contorno).
- Lo spazio è suddiviso in celle dette pixel (o voxel);
- una cella è “piena” se ha intersezione non vuota con la regione, è detta vuota in caso contrario.

Region quadtree

- Il region quadtree è un albero quaternario, simile al point-quadtree, che si costruisce nel seguente modo
 - Si considera un quadrato iniziale grande abbastanza da contenere la regione in questione e lo si pone come radice del quadtree
 - Si suddivide quindi tale quadrato in quattro parti uguali, ottenendo quattro quadrati di lato metà rispetto alla radice (quadranti); ciascuno di essi è un figlio per la radice del quadtree.
 - Quando un quadrante contiene solo celle piene o solo celle vuote non viene ulteriormente suddiviso, ed il nodo corrispondente è una foglia; altrimenti si ripete la divisione in quattro.



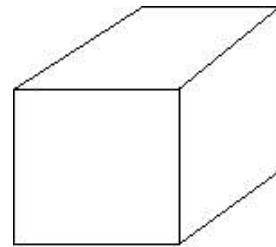
- la rappresentazione della regione che si ottiene è più economica rispetto alla rappresentazione delle singole celle, poiché grandi aree uniformi (piene o vuote) vengono rappresentate con una sola foglia (anche se nel caso peggiore il numero delle foglie è pari a quello delle celle)



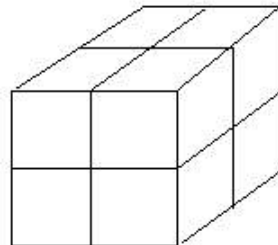
- In questa figure si vede una immagine (a livelli di grigio) e le regioni determinate dall'octree che la rappresenta.

Region octree

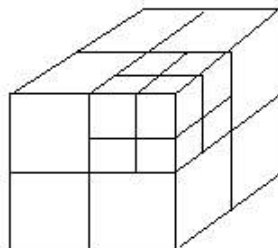
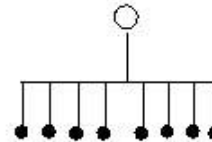
- È immediato estendere il region quadtree alle tre dimensioni
- Si ottiene il cosiddetto *region octree*
- Ogni cubo viene suddiviso in 8 (da qui il nome), per il resto è identico al quadtree



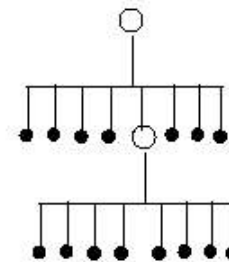
(root)



(1 level)



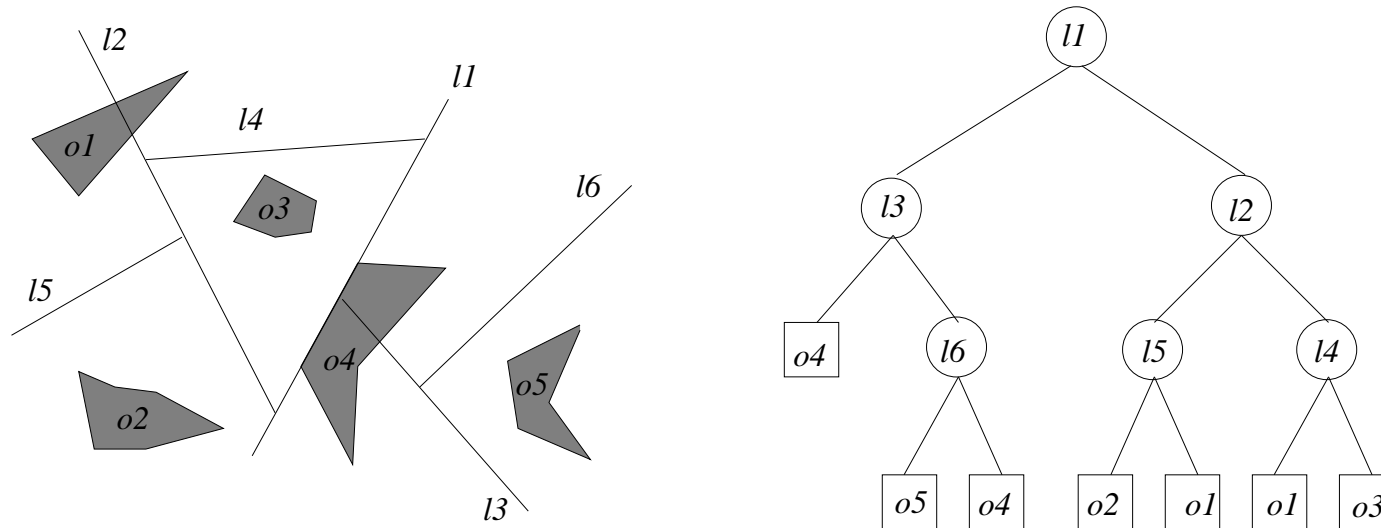
(2 levels)



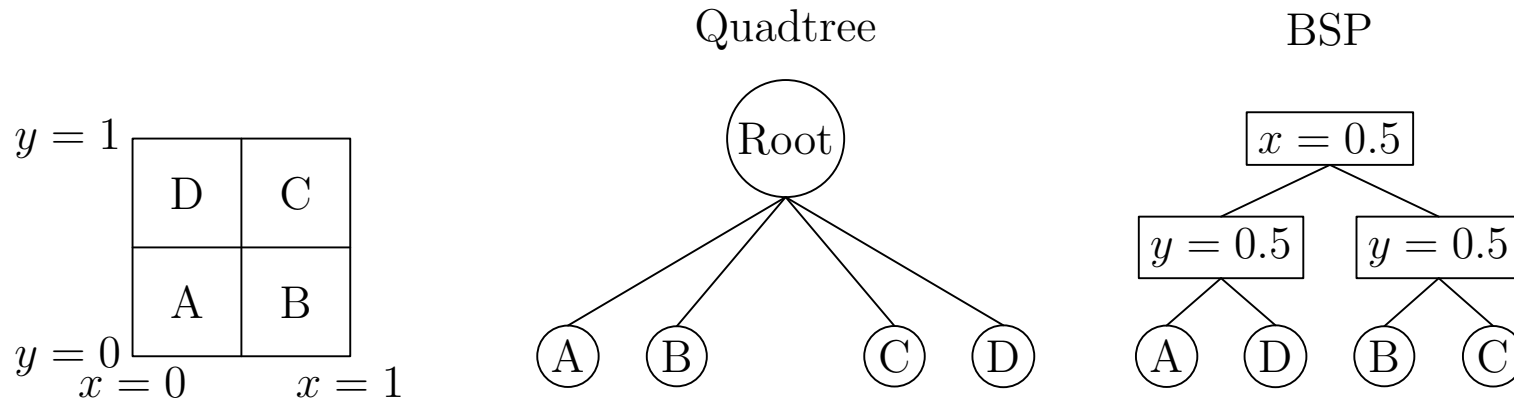
BSP tree

- Il *Binary Space Partition tree* è una struttura dati basata sulla suddivisione ricorsiva dello spazio lungo iperpiani arbitrari.
- Sebbene i BSP possano essere impiegati per organizzare rappresentazioni volumetriche, non offrono in questo caso alcun vantaggio sugli octrees (v. esempio);
- essi sono piuttosto impiegati per rappresentare collezioni di oggetti geometrici (segmenti, poligoni, ...)
- Gli iperpiani oltre a partizionare lo spazio, possono anche dividere gli oggetti
- Le proprietà che vengono sfruttate in grafica sono:
 - un oggetto (o parte di esso) che si trova una parte di un piano non interseca gli oggetti che si trovano dall'altra parte
 - dato un punto di vista, ed un piano, gli oggetti che stanno dalla stessa parte del punti di vista sono ad esso più vicini di quelli che stanno dalla parte opposta.

- Il processo di suddivisione ricorsiva continua finché un solo frammento di oggetto è contenuto in ogni regione.
- Questo processo si modella naturalmente con un albero binario, in cui
 - le foglie sono le regioni in cui lo spazio è suddiviso e contengono i frammenti di oggetto.
 - I nodi interni rappresentano gli iperpiani.
 - Le foglie del sottoalbero destro contengono i frammenti di oggetti che stanno alla destra dell'iperpiano.
 - Le foglie del sottoalbero sinistro contengono i frammenti di oggetti che stanno alla sinistra dell'iperpiano.

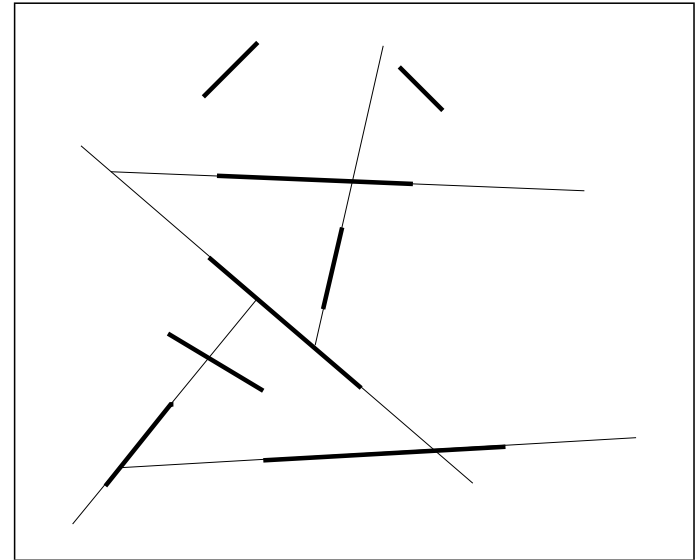


- Vediamo un esempio bidimensionale semplice in modo da compararlo con un quadtree
- Si supponga di avere la suddivisione in figura
- I due alberi, il quadtree ed il BSP, sono semplici da trovare



- Da notare che per il BSP si sono scelte le due rette di suddivisione $x = 0.5$ e $y = 0.5$; in particolare la seconda retta si è usata due volte
- Per suddivisioni di questo tipo (ortogonali), non vi è nessun vantaggio a scegliere il BSP rispetto al Quadtree

- Vediamo ora come costruire un BSP tree.
- Consideriamo il caso di un insieme di segmenti nel piano (che non si intersecano).
- Le linee di suddivisione sono arbitrarie.
- Per ragioni computazionali restringiamo le possibili linee a quelle contenenti i segmenti dati: un BSP che usa solo queste linee si dice *auto-partizionante*.



- Una buona scelta delle linee dovrebbe mantenere minima la frammentazione dei segmenti: poiché la scelta è difficile, si tira a caso.
- Algoritmo casuale:
 - si ordinano i segmenti in modo casuale e si procede pescando un segmento alla volta
 - se il segmento è l'ultimo della lista, si crea una foglia
 - altrimenti si usa la retta contenente il segmento come linea di suddivisione e si creano due liste di segmenti (eventualmente spezzando quelli originali) appartenenti ai due semipiani
 - si crea un nodo nell'albero e si considerano ricorsivamente le due liste di segmenti

- **Complessità:** La dimensione di un BSP tree è pari al numero di frammenti che vengono generati. Se n è il numero di segmenti originali, si può dimostrare che l'algoritmo casuale produce un numero di frammenti pari a $O(n \log n)$ (valore atteso). Il tempo necessario per costruire il BSP tree è $O(n^2 \log n)$.
- L'algoritmo si generalizza facilmente al caso di poligoni nello spazio 3D.