

Elementi di Geometria Computazionale



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

- **Preliminari**
- **Inviluppo convesso**
- **Intersezioni**
- **Triangolazioni**
- **Problemi di prossimità**
- **Ricerca geometrica**
- **Strutture dati geometriche**

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.

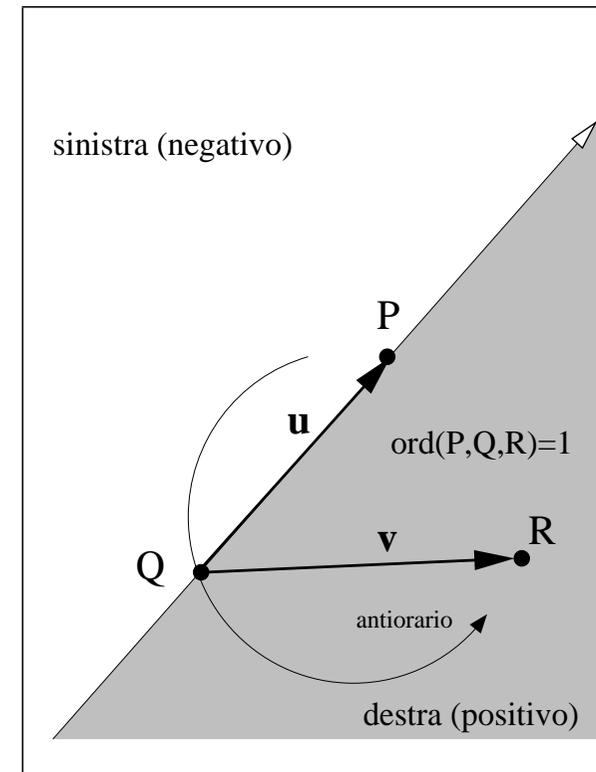
- Il fatto che si preferiscano algoritmi esatti esclude le divisioni e le operazioni trigonometriche, che sono affette da errori di arrotondamento (ed anche computazionalmente onerose). Idealmente vorremmo usare solo addizioni, sottrazioni, moltiplicazioni e confronti.
- L'input ad un problema di geometria 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.

Operazioni di base

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.
- dati due vettori nel piano $\mathbf{v} = (v_1, v_2)$ e $\mathbf{u} = (u_1, u_2)$, abbiamo definito $\mathbf{v} \times \mathbf{u} = \det(\mathbf{v}, \mathbf{u}) = v_1u_2 - v_2u_1$. Se $\mathbf{v} \times \mathbf{u}$ è positivo, allora \mathbf{u} deve ruotare in senso orario attorno all'origine per raggiungere \mathbf{v} (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 $\mathbf{v} \times \mathbf{u}$ e prenderne il segno.
- **Complessità**: costante ($O(1)$) poiché il numero di operazioni è fissato.



- $ord(P, Q, R)$ dice da che parte si trova R rispetto alla retta passante per P e Q , orientata da Q a P ; se R giace a destra della retta come nella figura, allora $ord(P, Q, R)$ è positivo.
- Il semipiano destro della rette prende anche il nome di semipiano positivo, in quanto sostituendo R nella equazione della retta si ottiene un valore positivo. In questo modo si può calcolare $ord(P, Q, R)$, arrivando alla stessa formula del determinante.
- $ord(P, Q, R)$ è pari al segno dell' area del triangolo P, Q, R (metà area del parallelogramma). Se, come nella figura, P, Q, R definiscono un percorso di senso antiorario, allora l'area del triangolo ha segno positivo.
- Si verifica (esercizio) che in modo equivalente si può scrivere l'area del triangolo (P, Q, R) come

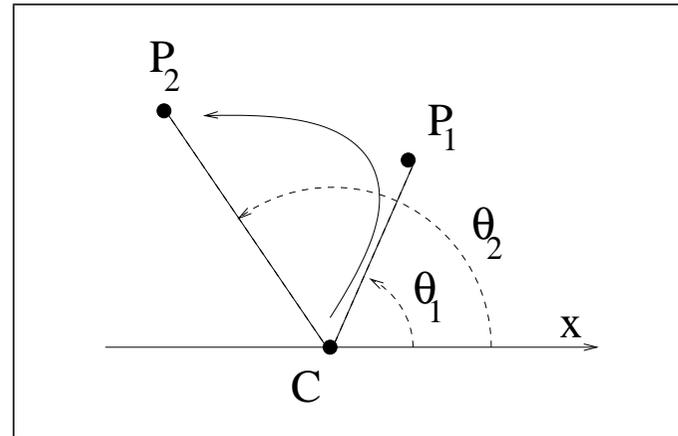
$$\frac{1}{2} \det \begin{bmatrix} P_x & Q_x & R_x \\ P_y & Q_y & R_y \\ 1 & 1 & 1 \end{bmatrix}$$

e questa forma si generalizza a n-dimensioni, per il calcolo di volumi con segno.

- Per esempio, in 3D, $ord(P, Q, R, S)$ stabilisce da che parte si trova S rispetto al piano orientato individuato da tre punti PQR (l'ordine dei punti determina l'orientazione del piano secondo la regola della mano destra).

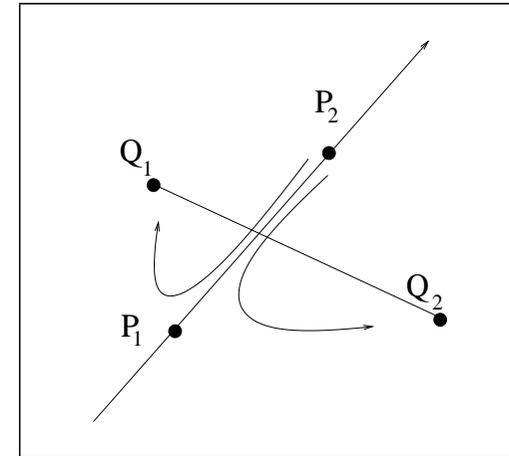
Confronto di angoli

- 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 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 $ord(C, P_1, P_2) = 1$)
- **Complessità**: costante ($O(1)$)



Test di intersezione tra due segmenti

- **Problema:** determinare se due segmenti in \mathbb{R}^2 si intersecano.
- Un segmento interseca una linea retta sse i suoi estremi giacciono da parti opposte rispetto alla retta.
- per esempio, il segmento (Q_1, Q_2) interseca la 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 interseca la 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) interseca la retta passante per P_1 e P_2 e contemporaneamente il segmento (P_1, P_2) interseca la retta passante per Q_1 e Q_2 ovvero se

$$ord(P_1, P_2, Q_1) \cdot ord(P_1, P_2, Q_2) < 0 \quad \text{e} \quad ord(Q_1, Q_2, P_1) \cdot ord(Q_1, Q_2, P_2) < 0$$

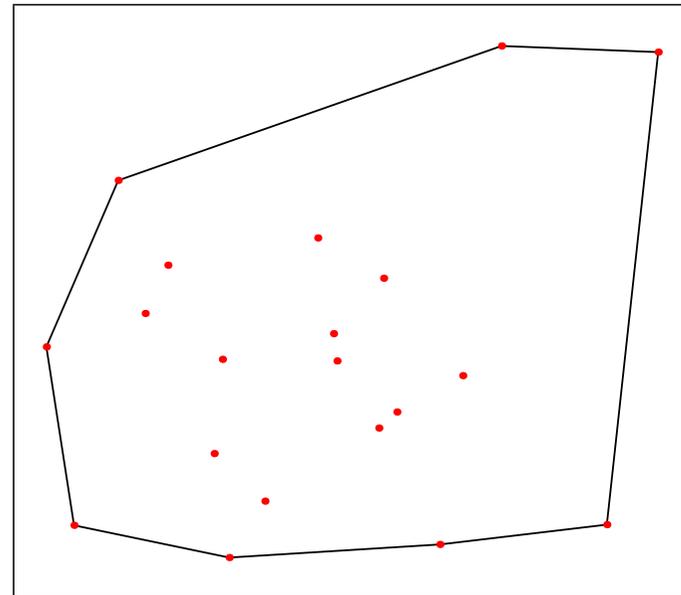
- **Complessità:** costante ($O(1)$) poiché il numero di operazioni è fissato.

Poligoni

- Informalmente, un **poligono** è una linea spezzata che si chiude su se stessa.
- Un poligono è una regione di piano delimitata da una successione ordinata e finita di punti $P_1 \dots P_n$ (vertici) e dai segmenti $(P_1, P_2) \dots (P_n, P_1)$ (lati) con la proprietà che due lati consecutivi si intersecano solo in corrispondenza del vertice comune.
- Un poligono è detto **semplice** se ogni coppia di lati non consecutivi ha intersezione vuota.
- Il **teorema di Jordan** ci dice che un poligono semplice partiziona il piano in due regioni distinte: interno ed esterno.
- Un poligono semplice divide il piano in due regioni, una limitata (detta **interno**) ed una illimitata (detta **esterno**). Percorrendo la frontiera del poligono in verso antiorario, l'interno rimane a sinistra.
- Per convenzione, un poligono viene rappresentato dalla sequenza dei suoi vertici $P_1 \dots P_n$ ordinata in senso **antiorario**.
- Un poligono semplice è **convesso** se la regione di piano che delimita è convessa (v. prossima slide).

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.
- Diremo che un punto appartiene all'inviluppo convesso, intendendo (se non specificato diversamente) al **bordo** di questo.



- **Problema:** (CONVEX HULL) dato un insieme \mathcal{V} di n punti nel piano, costruire la descrizione completa del suo guscio convesso.
- Il termine “descrizione completa” significa che bisogna fornire i punti che compongono il bordo del guscio convesso nell’ordine opportuno (in senso antiorario).
- Se si indebolisce questa richiesta e ci si accontenta dell’insieme dei punti che appartengono al guscio convesso, si ha il problema EXTREME POINTS.
- Descriveremo brevemente quattro algoritmi per la soluzione di CONVEX HULL.
- per semplicità presenteremo gli algoritmi per il caso bidimensionale.

Applicazioni

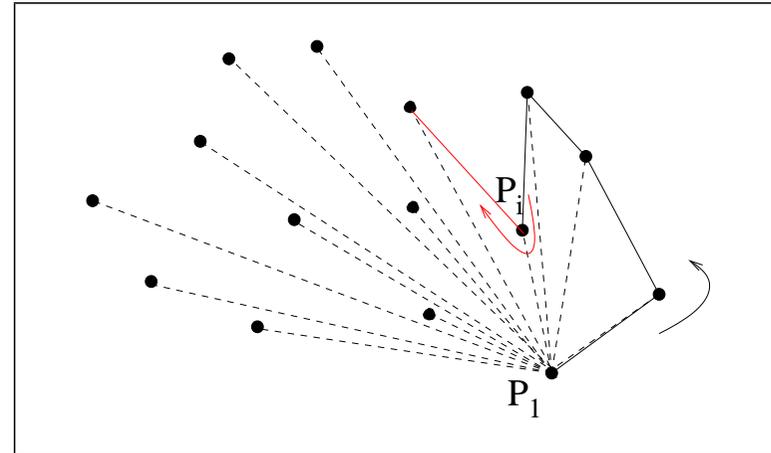
- I poligoni (o poliedri, in 3D) convessi sono le figure più semplici da trattare
- L’involuppo convesso è la più semplice “approssimazione” di un insieme di punti
- L’uso dell’involuppo convesso alle volte può aiutare ad eliminare operazioni inutili.
- **Esempio:** un problema tipico di applicazioni 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.

Algoritmo diretto

- Il modo più diretto per calcolare l'involuppo convesso è anche il più inefficiente.
- Idea: la retta che passa per due punti consecutivi del bordo lascia tutti gli altri dalla stessa parte
- 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
- Questo risolve EXTREME POINTS. Per ottenere una soluzione di CONVEX HULL bisogna ordinare i punti per coordinata polare rispetto ad qualunque punto interno.
- **Complessità:** si considerano $(n^2 - n)/2$ coppie di punti, e per ciascuna coppia si considerano nel test gli altri $n - 2$ punti, dunque la complessità è $O(n^3)$. Il costo dell'ordinamento è $O(n \log n)$ e viene assorbito.

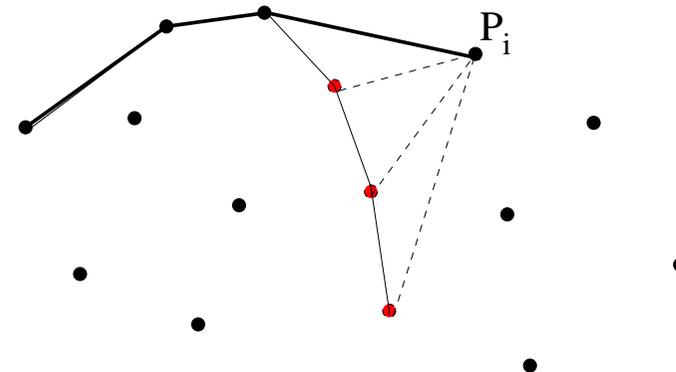
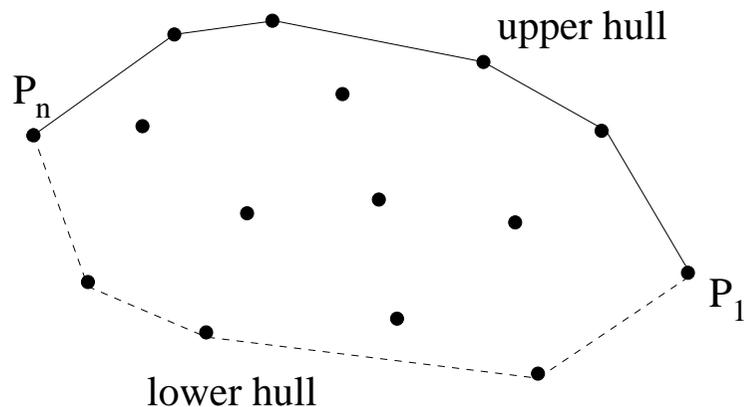
Graham's Scan

- 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 antiorario, 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 antiorario, si fanno solo svolte a sinistra. Ovvero: **l'ordine di tre vertici consecutivi di un poligono convesso è positivo.**
- Sia $P(i)$ l'ultimo elemento inserito nel guscio convesso.
- Se $P(i-1), P(i), P(i+1)$ definiscono una svolta a sinistra si avanza nella scansione e si passa a controllare la terna $P(i), P(i+1), P(i+2)$
- Se $P(i-1), P(i), P(i+1)$ definiscono una svolta a destra si elimina $P(i)$ dal guscio e si passa a controllare la terna $P(i-2), P(i-1), P(i+1)$
- Tutto è semplificato se si impiega uno stack per impilare i punti del convex hull.

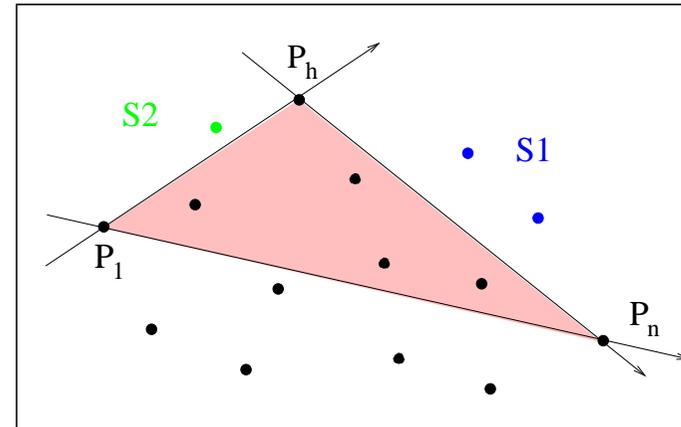
- **Variante:** ordino i punti per x decrescente, e li aggiungo da destra a sinistra.
- Inserendo i punti in quest'ordine ottengo solo la metà superiore del guscio (upper hull). Quella inferiore si ottiene analogamente con un'altra passata.
- L'algoritmo per il calcolo dell' upper hull è il seguente:
 - Si ordinano i punti per x crescente
 - Si prendono i primi due punti e li si mette nell'involucro convesso (una pila)
 - Aggiungo i punti successivi con la seguente regola
 - * Se l'ultimo punto aggiunto è tale che l'ordine degli ultimi tre punti dell'involucro convesso è positivo, allora si prosegue
 - * Altrimenti si eliminano via via gli ultimi punti aggiunti, a partire dal penultimo, fin tanto che la condizione di positività dell'ordine è soddisfatta o rimangono solo 3 punti nell'involucro convesso



- 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 $O(n \log n)$
- **Vantaggi e svantaggi**: 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 \mathcal{S} l'insieme dei punti che stanno sopra la retta orientata da P_1 a P_n .
- P_1 ed P_n appartengono all'involuppo 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 \mathcal{S}_1 ed \mathcal{S}_2 che sono esterni ai lati P_hP_n ed P_1P_h , rispettivamente, dell'attuale involuppo
- la procedura si attiva ricorsivamente prendendo \mathcal{S}_1 ed \mathcal{S}_2 separatamente come \mathcal{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 \mathcal{S}_1 ed \mathcal{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 \mathcal{S} è $O(m)$ dove m è la cardinalità dell'insieme \mathcal{S} corrente.
 - Sia $T(n)$ il costo del calcolo del quickhull su un insieme \mathcal{S} di n punti. $T(n)$ soddisfa la seguente ricorrenza:

$$T(0) = 1$$

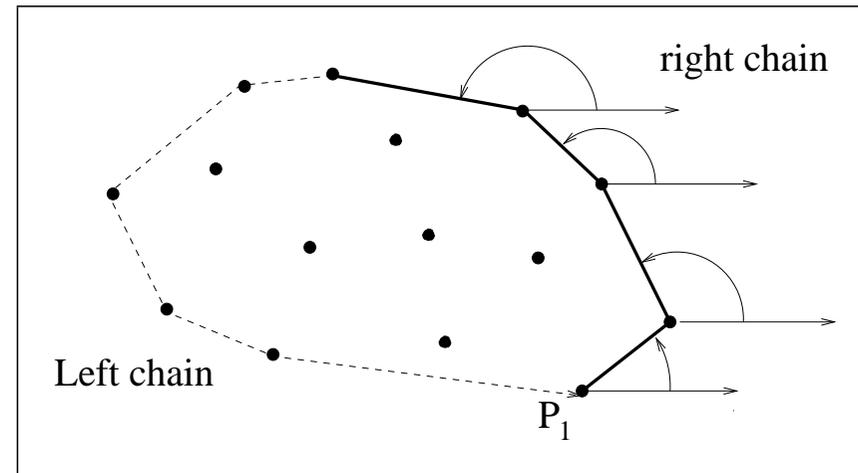
$$T(n) = T(n_1) + T(n_2) + n$$

dove n_1 ed n_2 sono le cardinalità di \mathcal{S}_1 ed \mathcal{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 da vedere se si prende $\alpha = 1/2$, come nel caso ottimo di Quicksort.)
- **Vantaggi e svantaggi:** 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.

Intersezioni

- Motivazione: due oggetti non possono occupare lo stesso posto allo stesso tempo.
- Applicazioni:
 - rimozione delle linee e delle superfici nascoste in Grafica (un oggetto ne oscura un'altro se le loro proiezioni si intersecano).
 - Separabilità lineare di due insiemi di punti in Pattern Recognition
 - Layout dei circuiti integrati.
- Problemi di **costruzione**: si richiede di restituire l'intersezione di oggetti geometrici.
- Problemi di **test**: si richiede di determinare se gli oggetti si intersecano oppure no.

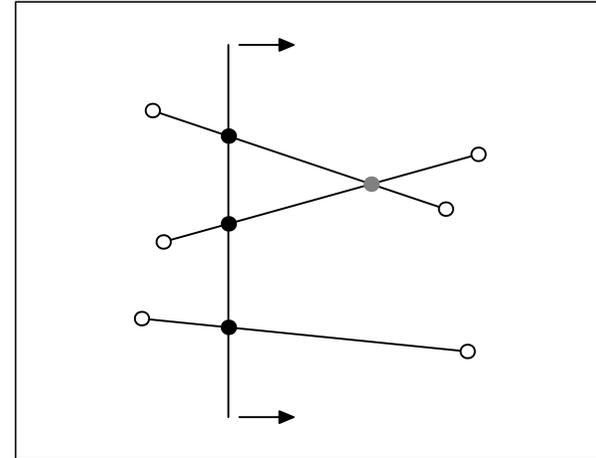
Intersezioni di segmenti

- **Problema:** (LINE-SEGMENT INTERSECTION) 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 (nel piano) 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 ℓ , vediamo un algoritmo, denominato **plane sweep**, che risolve il problema in $O(n \log n + \ell \log n)$ (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 sovrappongano, 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 geometria computazionale e nella grafica al calcolatore.

- 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 di ricerca (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 di ricerca (bilanciato).

Vediamo quindi l'algoritmo completo:

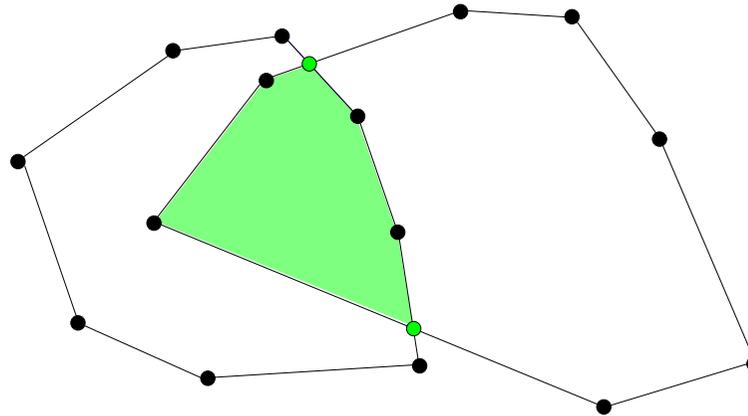
- Siano $\{s_1, \dots, s_n\}$ gli n segmenti da analizzare, rappresentati mediante gli estremi.
- Si parte con la linea tutta a sinistra, e si inseriscono nella coda degli eventi tutti gli estremi dei segmenti $\{s_1, \dots, s_n\}$; lo stato della linea di sweep è vuoto. Nel seguito ogni volta che si 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.

- **Problema:** (LINE-SEGMENT INTERSECTION TEST) dati n segmenti di retta nel piano, determinare se due qualunque si intersecano.
- È un problema più semplice del LINE-SEGMENT INTERSECTION, ed infatti si può esibire un algoritmo derivato dal plane sweep precedente che impiega $O(n \log n)$.
- Intuitivamente: dovendo solo rispondere sì o no, si risparmia sulla elencazione dell'output, cui era dovuto il termine con ℓ .

Intersezioni di poligoni

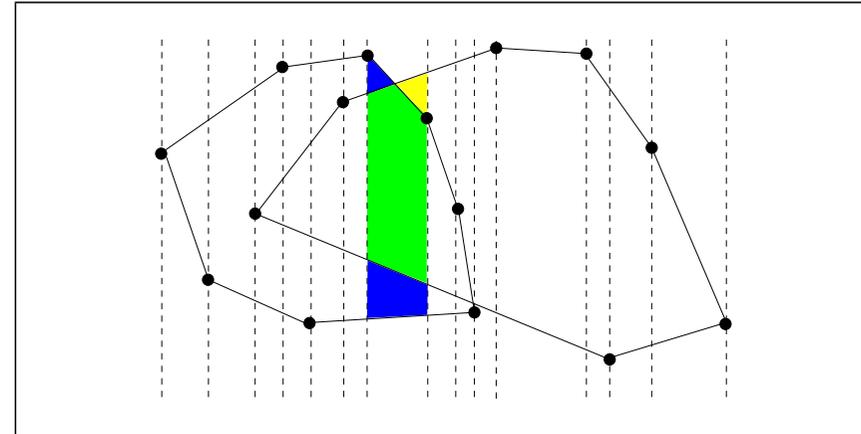
- **Problema:** (POLYGON INTERSECTION TEST) dati due poligoni semplici P con ℓ vertici e Q con m vertici, decidere se si intersecano.
- Osservazione: se P e Q si intersecano o i) uno dei due è contenuto nell'altro, oppure ii) qualche lato di P interseca qualche lato di Q .
- Si verifica prima la condizione ii) risolvendo un problema di LINE-SEGMENT INTERSECTION TEST, con un costo computazionale di $O(n \log n)$ con $n = \ell + m$.
- Se non si trova alcuna intersezione, bisogna controllare la condizione i).
- Perché P sia interno a Q è necessario che *tutti* i vertici di P siano interni a Q , dunque basta risolvere una istanza del problema di POLYGON INCLUSION (che vedremo) usando un qualunque vertice di P , con un costo lineare. La stessa cosa si fa per verificare se Q è interno a P .
- **Complessità:** $O(n \log n)$

- **Problema:** (CONVEX POLYGON INTERSECTION) dati due poligoni convessi P con ℓ vertici e Q con m vertici, determinare la loro intersezione.
- Si dimostra che l'intersezione di P e Q è un poligono convesso con al più $\ell + m$ vertici
- Il bordo di $P \cap Q$ consiste di sequenze alternate di vertici dei due poligoni inframezzate da punti in cui i bordi si intersecano.



Metodo delle Strisce (slab)

- Dati due poligoni convessi P e Q , suddividere il piano in strisce verticali passanti per i vertici ed ordinarle per ascissa crescente.
- Oss. Dato che i vertici di P e Q sono separatamente ordinati, per metterli assieme il costo è lineare (come in MergeSort).
- L'intersezione non vuota di una striscia con uno dei due poligoni è un trapezoide.
- All'interno di ciascuna striscia devo calcolare l'intersezione di due trapezoidi, che ha costo costante.
- Alla fine metto insieme i pezzi provenienti da ciascuna striscia con una passata a costo lineare.
- **Complessità**: sia l'ordinamento che la scansione delle strisce è lineare nel numero complessivo dei vertici, dunque $O(\ell + m)$.



Intersezioni di semipiani

- La costruzione della intersezione di n semipiani consiste nel determinare la regione che contiene le soluzioni del sistema di n equazioni lineari del tipo:

$$a_i x + b_i y + c_i \leq 0 \quad i = 1, 2, \dots, n$$

- si tratta di una regione poligonale convessa, non necessariamente limitata.
- **Problema:** (HALF-PLANE INTERSECTION) dati n semipiani $H_1 \dots H_n$, determinare la loro intersezione: $H_1 \cap H_2 \cap \dots \cap H_n$.
- Soluzione incrementale: assumiamo di avere già intersecato i primi i semipiani, abbiamo un poligono convesso con al più i lati. L'intersezione di questo con il prossimo semipiano si effettua in tempo $O(i)$. Dunque il lavoro totale richiesto è $O(n^2)$.
- Soluzione divide-et-impera: risolvo due sottoproblemi di dimensione $n/2$ e poi interseco i risultati (associatività della intersezione). Costo dato dalla ricorrenza:
 $T(n) = 2T(n/2) + O(n)$, la cui soluzione è $O(n \log n)$
- Nota: HALF-PLANE INTERSECTION è il duale di CONVEX HULL, secondo la mappa che porta linee in punti e viceversa: $L : y = 2ax - b \leftrightarrow p : (a, b)$

Intersezioni di semispazi

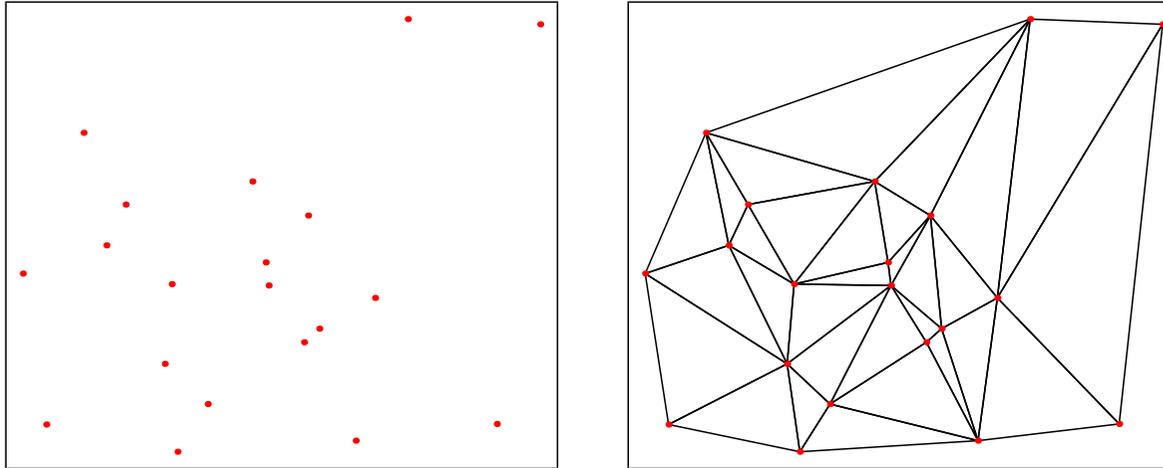
- **Problema:** (HALF-SPACE INTERSECTION) dati n semispazi $H_1 \dots H_n$ in \mathbb{R}^3 , determinare la loro intersezione: $H_1 \cap H_2 \cap \dots \cap H_n$.
- La soluzione è un poliedro convesso (oppure l'insieme vuoto).
- La generalizzazione del metodo divide-et-impera precedente costa $O(n^2 \log n)$ perché l'intersezione di poliedri convessi (fase di merge) non è lineare come nel caso dei poligoni ma richiede $O(n \log n)$, dove n è la somma del numero dei vertici dei due poliedri.
- Esiste però un algoritmo ottimo in $O(n \log n)$ dovuto a Preparata-Muller (1979).

Triangolazioni

Suddivisione poligonale del piano

- Una suddivisione poligonale del piano è un insieme \mathcal{P} (infinito) di poligoni 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.
- spesso si considerano 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 \mathcal{V} di punti nel piano è una triangolazione i cui vertici sono i punti di \mathcal{V} .

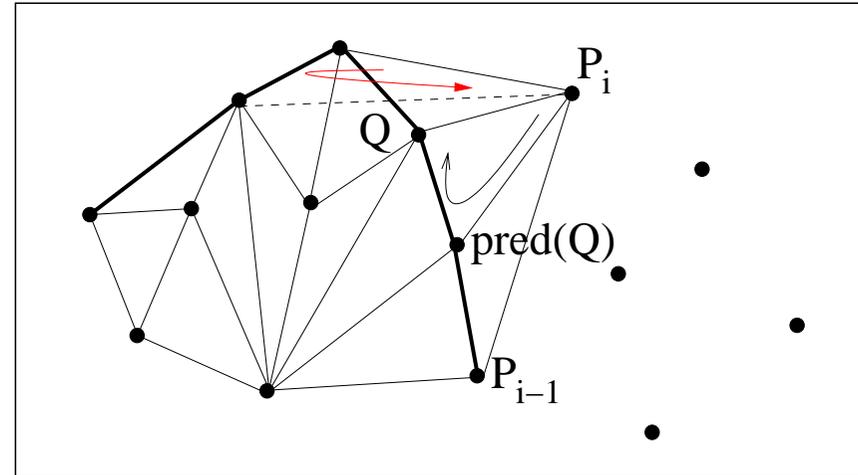


- Sono utili perché il triangolo è il poligono convesso più semplice
- Una suddivisione poligonale qualsiasi è sempre triangolabile aggiungendo opportunamente dei lati (questo deriva direttamente dal fatto di poter sempre decomporre un poligono convesso in un numero finito di triangoli)
- Si dimostra che: ogni poligono semplice con n lati può essere triangolato con $n - 2$ triangoli.
- **Problema:** (TRIANGULATION) dato un insieme \mathcal{V} di n punti nel piano, unirli tutti con segmenti non intersecantisi in modo che ogni regione interna al guscio convesso sia un triangolo.

Algoritmo di triangolazione plane sweep

- Risolve TRIANGULATION in $O(n \log n)$.
- Si tratta di un algoritmo incrementale, che impiega una tecnica di plane sweep e che ha qualche analogia con il Graham's scan.
- si ordinano i punti di \mathcal{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

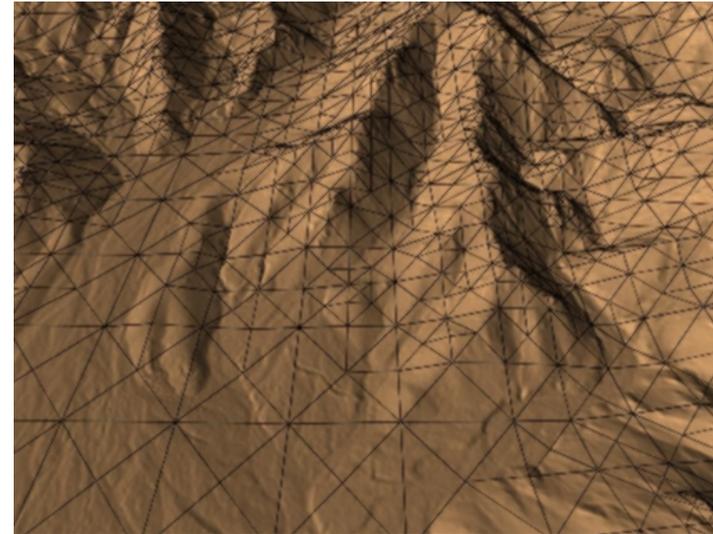
- **Soluzione:** muoversi lungo il guscio convesso a partire da P_{i-1} in senso orario ed antiorario ed unire i punti del guscio convesso a P_i finché non si incontra un punto non visibile. L'ultimo punto visibile incontrato prende il nome di punto di tangenza.



- Il test di visibilità si effettua controllando l'ordinamento della terna $P_i, \text{pred}(Q), Q$, dove Q è un punto sul guscio convesso e $\text{pred}(Q)$ 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.

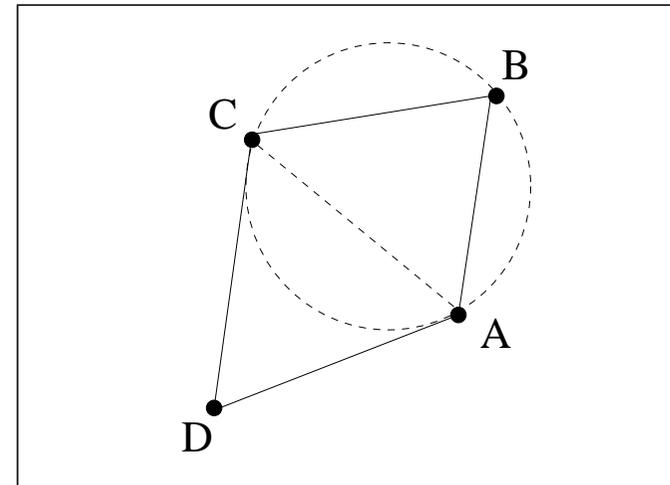


- Vorremmo allora che i vertici fossero tra loro il più possibile vicini, per evitare di interpolare tra valori distanti: questa richiesta si traduce nel cercare di avere angoli il meno acuti possibile.
- Possiamo allora pensare di ordinare le triangolazioni del piano in base all'angolo più piccolo, 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).

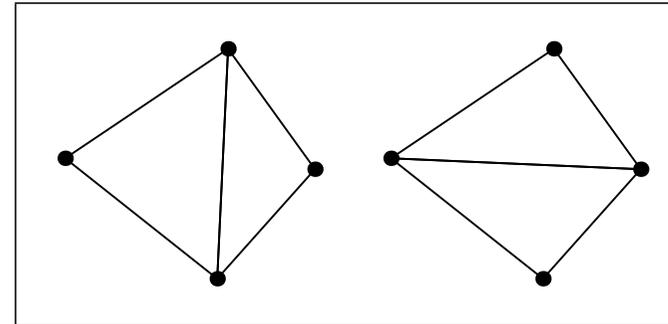
- Si può dare una caratterizzazione ulteriore della triangolazione di Delaunay (di solito è la definizione):
- una triangolazione di un insieme di punti \mathcal{V} è di Delaunay se e solo se ogni triangolo è inscritto in una circonferenza **libera**, ovvero tale che non contenga alcun punto di \mathcal{V} .
- 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.

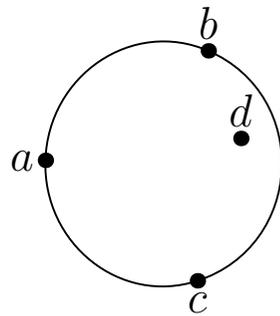


- **Osservazione:** se un triangolo non soddisfa la proprietà di circonferenza libera, è possibile effettuare una operazione locale (edge flip) la quale fa sì che la proprietà sia rispettata ed incrementa il rango della triangolazione in \mathcal{T} .
- **Operazione di edge flip:** dati due triangoli 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} .

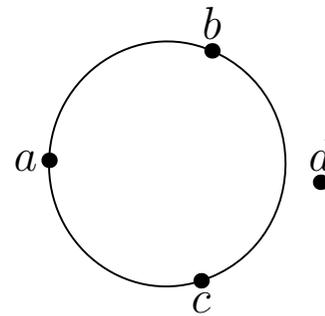


- Per calcolare il predicato $\text{InCircle}()$, sfruttiamo la seguente proprietà:
- Siano A, B, C e D quattro punti sul piano; si dimostra (v. appunti De Floriani) che D è interno al cerchio passante per $ABCD$ ($\text{InCircle}(A, B, C, D)$ è vero), se e solo se

$$\det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} < 0$$



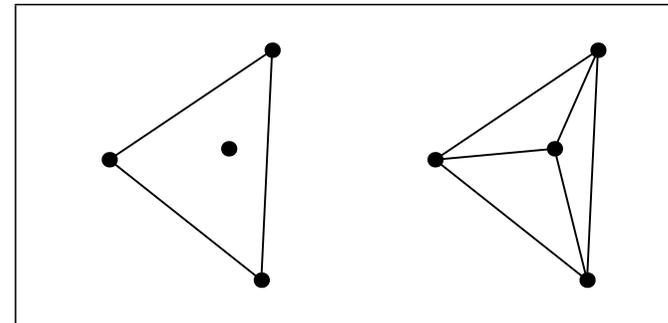
$$\text{In}(a, b, c, d) < 0$$



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

Algoritmo GKS

- Vediamo ora un metodo efficiente, **progressivo** e **casuale** per la costruzione della triangolazione di Delaunay di un insieme di punti \mathcal{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 \mathcal{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 , PBC 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 , PBC e PCA .
 - Per ciascuno dei nuovi triangoli si esegue il test $\text{InCircle}()$ 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 \mathcal{V}
- Come si parte? Per esempio con un triangolo opportuno abbastanza grande da contenere tutto \mathcal{V} .

- La correttezza dipende dal fatto che:
 - è sufficiente testare triangoli che contengono P , perché P è l'unica perturbazione apportata 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**

Problemi di prossimità

- Si tratta di problemi che coinvolgono il concetto di prossimità o vicinanza tra punti.
- **Problema 1:** (NEAREST NEIGHBOUR SEARCH) Dato un insieme \mathcal{P} di n punti nello spazio e dato un nuovo punto Q (query point) si chiede di trovare il punto dell'insieme \mathcal{P} che è più vicino a Q .
- **Problema 2:** (CLOSEST PAIR) Dato un insieme \mathcal{P} di n punti, trovare i due più vicini.
- Vedremo due tecniche: una basata su un algoritmo *divide-et-impera* (di M. I. Shamos), l'altra basata su una struttura dati notevole: i **diagrammi di Voronoi**.
- Come al solito ci limiteremo ad analizzare il caso bidimensionale, tenendo conto che è possibile l'estensione di questa analisi a dimensioni maggiori

Algoritmo di Shamos

- Risolve il problema di CLOSEST PAIR in $O(n \log n)$.
- L'approccio di forza bruta richiederebbe di controllare tutte le possibili coppie di punti, ovvero $O(n^2)$.
- Si parte con due array X e Y , che contengono entrambe i punti di \mathcal{P} , ordinati per x crescente e per y crescente, rispettivamente.
- L'algoritmo ricorsivo segue lo schema classico del divide-et-impera. Si articola quindi in tre fasi (cfr. QuickSort o MergeSort).

Dividi: con una linea verticale biseca l'insieme di punti \mathcal{P} in due sottoinsiemi \mathcal{P}_L e \mathcal{P}_R , in modo che i punti di \mathcal{P}_L stiano alla sinistra della linea e quelli di \mathcal{P}_R stiano alla destra della linea. I due array ordinati X e Y sono divisi ciascuno in due nuovi array ordinati X_L , X_R e Y_L , Y_R rispettivamente (in tempo lineare)

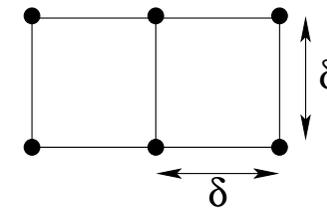
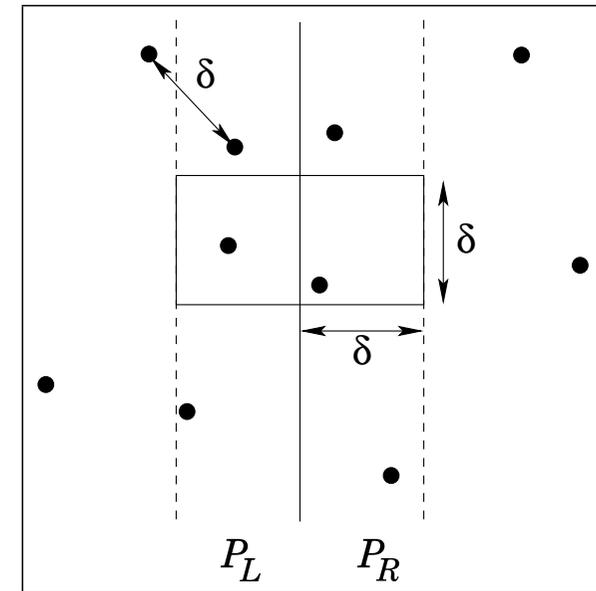
Conquista: chiama ricorsivamente la procedura per trovare in \mathcal{P}_L e \mathcal{P}_R i due punti più vicini. Il caso base è quando si hanno 3 o meno punti.

Combina: la coppia di punti più vicini in \mathcal{P} è una delle due coppie trovate in \mathcal{P}_L e \mathcal{P}_R oppure una coppia formata da un punto di \mathcal{P}_L ed uno di \mathcal{P}_R . Detta δ la minima distanza tra le coppie ritornate dalla chiamata ricorsiva, bisogna determinare se esiste una coppia “mista” con distanza inferiore a δ . La ricerca si limita quindi ad una striscia larga 2δ attorno alla linea di separazione. Per ogni punto P in tale striscia, si devono considerare tutti i punti che distano da P meno di δ .

L’osservazione chiave è che tali punti sono in numero **costante**, infatti, in un rettangolo $2\delta \times \delta$ possono esserci al più 6 punti separati almeno da una distanza δ .

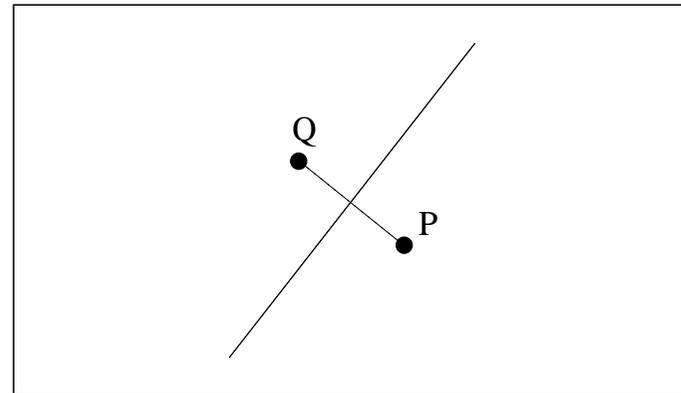
Per ogni punto della striscia, solo i 5 punti che lo seguono nell’array ordinato per y crescente devono essere considerati.

- **Complessità:** essendo i passi “dividi” e “combina” entrambe di costo lineare, la complessità è data dalla soluzione della (solita) ricorrenza $T(n) = 2T(n/2) + O(n)$, ovvero $O(n \log n)$.

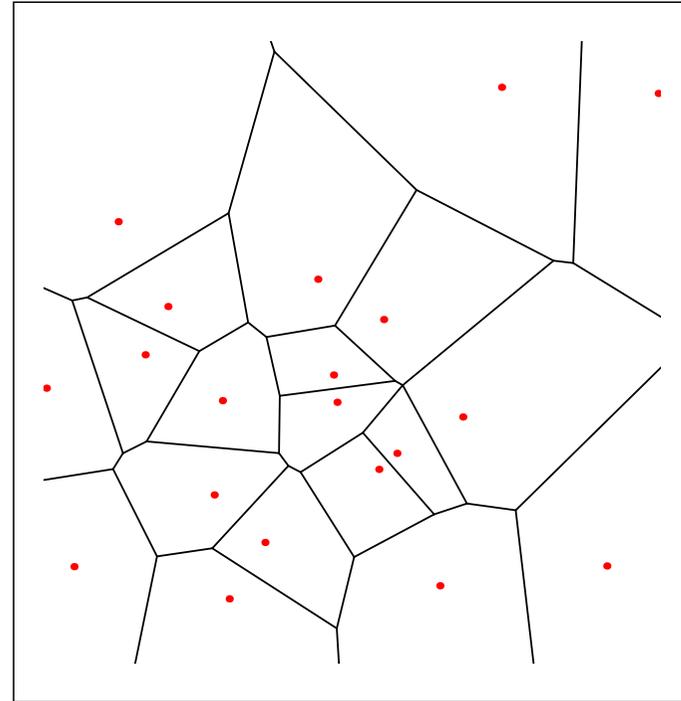


Diagrammi di Voronoi

- Dato un insieme \mathcal{P} di n punti sul piano, la **cella di Voronoi** $\mathcal{V}(P)$ di un punto $P \in \mathcal{P}$ è definita come l'insieme dei punti del piano Q tali che $\forall R \in \mathcal{P}$ con $R \neq P$ si abbia $\|Q - P\| < \|Q - R\|$
- Una cella di Voronoi si può definire alternatively come intersezione dei semispazi definiti dalle rette bisettrici il punto dato con tutti gli altri punti.
- Dati due punti P e Q , la loro bisettrice è la retta perpendicolare a \overline{PQ} nel punto medio.
- Da questo segue immediatamente che le celle di Voronoi sono poligoni convessi (eventualmente illimitati).
- Si dimostra inoltre due celle di voronoi hanno al più un lato a comune e che tale lato giace sulla retta equidistante dai due punti corrispondenti alle celle.



- L'insieme delle celle di Voronoi di \mathcal{P} (private del loro interno) costituisce il **diagramma di Voronoi** di \mathcal{P} .
- Il diagramma di Voronoi è un grafo planare i cui vertici sono i punti nei quali tre celle si intersecano ed i lati sono i lati delle celle.



- Un vertice del diagramma di Voronoi dove si intersecano le tre celle $\mathcal{V}(P_1)$, $\mathcal{V}(P_2)$ e $\mathcal{V}(P_3)$ è equidistante dai 3 punti P_1, P_2, P_3 . Dunque esiste un cerchio centrato in tale vertice che passa per P_1, P_2 e P_3 e che non contiene nessun altro punto di \mathcal{P} (per la proprietà delle celle).
- Questo ricorda la triangolazione di Delaunay, infatti ...

- **Il duale del diagramma di Voronoi è la triangolazione di Delaunay.**
- Il grafo duale ha per vertici le celle (prendiamo i punti originali di \mathcal{P} come rappresentanti) ed ha un lato che connette due vertici se le corrispondenti celle sono adiacenti.
- Poiché i vertici del diagramma di Voronoi hanno grado 3, le facce del duale sono triangoli.
- Si può calcolare il diagramma di Voronoi usando la definizione in termini di intersezione di semispazi (in $O(n^2 \log n)$, inefficiente),
- oppure si può calcolare a partire dalla triangolazione di Delaunay (in $O(n \log n)$),
- oppure mediante algoritmi ad hoc, come quello di Fortune, basato su una sweep line modificata (in $O(n \log n)$).

Applicazioni

- Per risolvere il problema del NEAREST NEIGHBOUR SEARCH usando il diagramma di Voronoi basta cercare la cella in cui il punto cade (point location, $O(n \log n)$) per ottenere il risultato (in $O(\log n)$).
- Per risolvere invece il problema di CLOSEST PAIR, si ritorna alla triangolazione di Delaunay. In particolare, grazie alla dualità con il diagramma di Voronoi, è facile rendersi conto della seguente proprietà:
- **le coppie di punti più vicini in \mathcal{P} sono connessi da un lato nella triangolazione di Delaunay.**
- Quindi bisogna costruire la triangolazione di Delaunay (in $O(n \log n)$) e quindi esaminare tutti i suoi lati, che sono $O(n)$; totale $O(n \log n)$.

Ricerca Geometrica

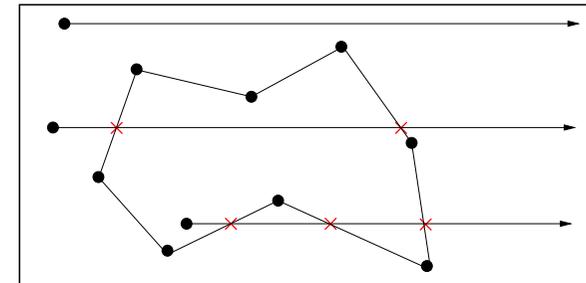
- Supponiamo di avere dei dati accumulati (“archivio”) ed un dato nuovo (“campione”). La ricerca geometrica consiste nel collegare il campione con l’archivio.
- Vi sono fondamentalmente due paradigmi di ricerca geometrica:
 1. **Problemi di localizzazione** (location), dove l’archivio rappresenta una partizione dello spazio in regioni ed il campione è un punto. Si vuole identificare la regione a cui appartiene il punto.
 2. **Range search**, dove l’archivio è una collezione di punti ed il campione è un regione dello spazio (tipicamente un rettangolo o una sfera). Il range search consiste nell’elencare (report problem) o contare (census or count problem) tutti i punti contenuti nella regione data.
- I due problemi sono in qualche senso duali.

Localizzazione di un punto

- Un tipico problema interessante è quello della localizzazione di un punto (point-location) rispetto a:
 - un poligono
 - una suddivisione poligonale del piano
- **Problema 1:** (POLYGON INCLUSION) Dato un poligono semplice – 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.
- **Problema 2:** (CONVEX POLYGON INCLUSION) Dato un poligono convesso e dato un punto Q , ci chiediamo se Q giace all'interno o all'esterno del poligono.
- **Problema 3:** (PLANAR POINT LOCATION) Data una **suddivisione poligonale** (di una regione finita) del piano e dato un punto P , trovare il poligono della suddivisione che contiene P .

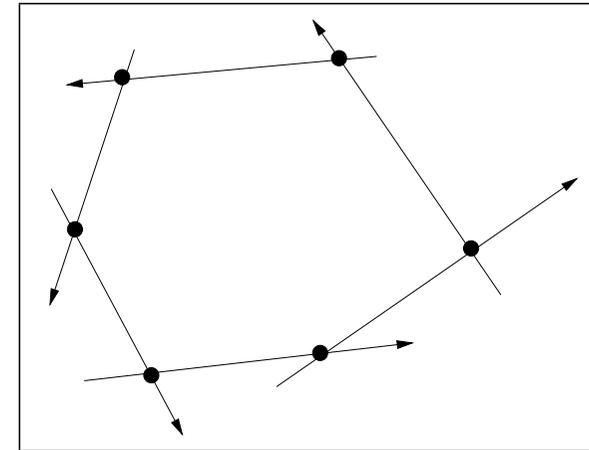
Metodo di Jordan

- **Problema:** POLYGON INCLUSION
- L'algoritmo ricalca la dimostrazione di Jordan.
- Consideriamo la semiretta orizzontale con origine in Q e diretta verso destra. Il punto all'infinito è per definizione all'esterno del poligono.
- Per scoprire se Q è interno o esterno, contiamo quante volte la semiretta attraversa il poligono.
 - se il numero di attraversamenti è pari Q , giace all'esterno del poligono;
 - se il numero di attraversamenti è dispari, Q giace all'interno del poligono;
- Casi degeneri vanno gestiti a parte.
- **Complessità:** devo effettuare il controllo di intersezione tra la semiretta e ciascun lato del poligono. Il test ha costo costante, quindi $O(n)$.



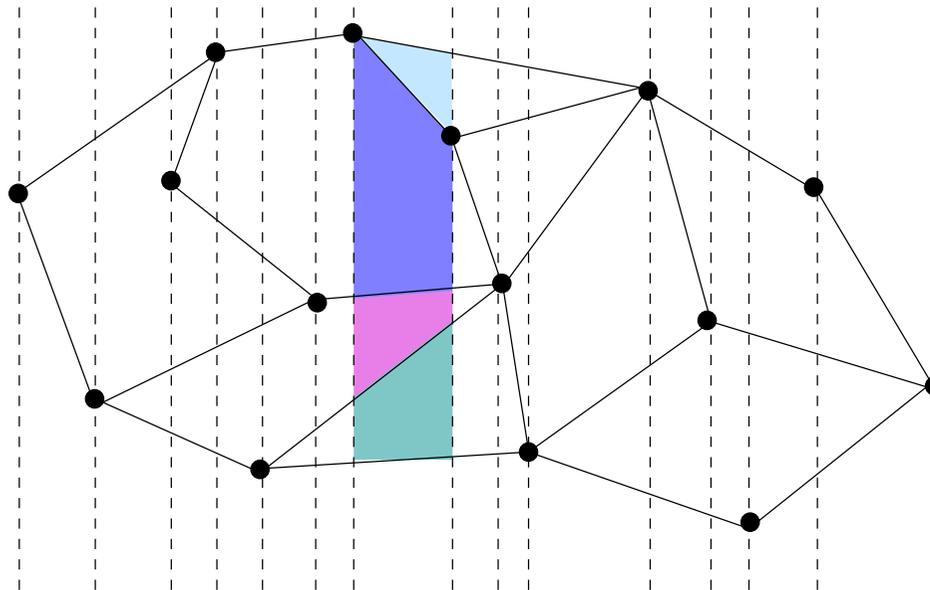
Metodo dei semipiani

- **Problema:** CONVEX POLYGON INCLUSION
- Una caratterizzazione equivalente di un poligono convesso, che sfruttiamo qui, dice che un poligono semplice convesso è l'intersezione dei semipiani sinistri che contengono i suoi lati sulla frontiera. Più precisamente:
 - Q giace all'interno del 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 $ord(P_{i+1}, P_i, Q)$ 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 $O(n)$



Metodo delle strisce

- **Problema:** PLANAR POINT LOCATION
- **Slab method** di Dobkin e Lipton, 1976.
- **Idea:** suddividere il piano in strisce verticali passanti per i vertici. L'intersezione non vuota di una striscia con un poligono della suddivisione è un trapezoide. Quindi i lati non verticali dei trapezoidi possono essere ordinati, per esempio dal basso verso l'alto.



- Non si assume che i poligoni della suddivisione siano convessi.

- Per localizzare un punto $P = (x, y)$ si compie una ricerca sulle ascisse per individuare la striscia contenente la x , seguita da una ricerca sulle ordinate per individuare il trapeziode che contiene la y .
- Ovvero si cercano due segmenti consecutivi nella striscia tra i quali $P = (x, y)$ sia compreso. Il poligono della suddivisione che contiene P è quello a cui appartengono i due segmenti individuati.
- La struttura dati spaziale che rappresenta il contenuto delle strisce può essere efficientemente costruita usando l'algoritmo plane sweep (per intersezione di segmenti) visto precedentemente. In corrispondenza di una striscia, l'ordine dei segmenti al suo interno è ottenuto leggendo lo stato della linea di sweep (un albero) quando questa è all'interno della striscia. Questo costa $O(n^2)$ tempo ed occupa $O(n^2)$ spazio.
- **Complessità:** Le strisce sono al più $n - 1$, quindi la ricerca (dicotomica) sulle x costa $O(\log n)$. I lati che intersecano una striscia sono $O(n)$, quindi anche la ricerca sulle y costa $O(\log n)$.

Localizzazione di un punto: caso 3D

- **Problema:** (CONVEX POLYHEDRON INCLUSION)
- Si usa la definizione di poliedro convesso come intersezione dei semispazi negativi individuati dalle facce, e quindi si usa $ord(P, Q, R, S)$ per controllare se il punto da localizzare si trova a sinistra di tutte le facce del poliedro.
- **Problema:** (POLYHEDRON INCLUSION)
- Si estende il metodo di Jordan visto in precedenza; si tratta quindi di contare le intersezioni di una semiretta di direzione arbitraria con le facce del poliedro.
- L'intersezione retta-poligono (faccia) si effettua in due passi:
 1. Si controlla se la retta interseca il piano contenente il poligono ed eventualmente si calcola il punto di intersezione P .
 2. Si localizza P rispetto al poligono.

Range search

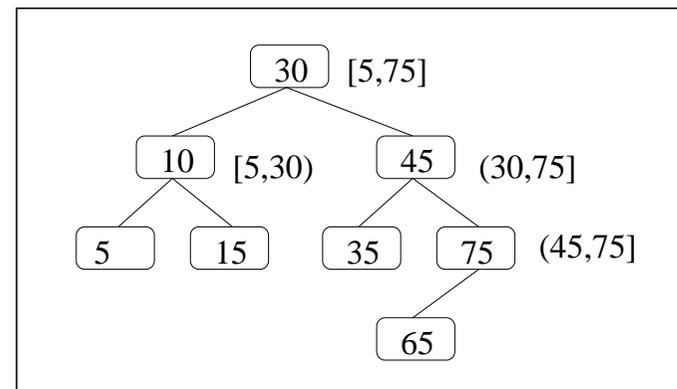
- **Problema:** (RANGE SEARCH - REPORT) Dato un insieme di punti \mathcal{P} in uno spazio Cartesiano d -dimensionale, e data una regione dello spazio (query), riportare i punti di \mathcal{P} che appartengono alla regione.
- Il più semplice problema RANGE SEARCH - COUNT prevede solo di contare il numero di punti che appartengono alla regione di query.
- Ci occuperemo inoltre del caso in cui la regione di ricerca sia un rettangolo con i lati paralleli agli assi cartesiani; tale ricerca prende anche il nome di **ricerca ortogonale**.
- Se la ricerca non viene eseguita una volta sola (single-shot), ma si pensa (come spesso accade), che si presenteranno query diverse sullo stesso insieme di punti (repetitive-mode), allora vale la pena di investire nella costruzione di una **struttura dati** opportuna per rappresentare \mathcal{P} .
- Introduremo strutture dati per la **rappresentazione gerarchica di insiemi di punti nello spazio**
- 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 costo della struttura dati verrà ripagato dalla maggiore velocità di ricerca.

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 $R = [a, b]$.
- Si potrebbe semplicemente ordinare i punti in modo crescente, e poi cercare, con ricerca binaria, il più piccolo punto $> a$ e quindi scorrere il vettore (scrivendo i punti) fino al primo punto $> b$
- Il costo è $O(\log n)$ per la ricerca binaria e $O(\ell)$, dove ℓ è il numero di punti che cadono in R , per la scansione lineare, in totale: $O(\log n + \ell)$.
- Questa 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 \mathcal{P}
 - Tutti i punti che giacciono a sinistra di un certo punto popolano il suo sottoalbero sinistro, quelli che giacciono a destra, il suo sottoalbero destro.
 - Assumiamo di avere alberi **bilanciati**, dove cioè i sottoalberi sinistro e destro di un nodo qualsiasi sono egualmente popolati.
- A ciascun nodo (interno) v è associato, oltre che il punto $\text{point}(v)$, anche un intervallo $\text{reg}(v)$ che contiene i punti che popolano il suo sottoalbero.
- Il punto $\text{point}(v)$ divide in due l'intervallo $\text{reg}(v)$.
- **Importante:** queste ultime due osservazioni, ovvero che ad ogni nodo è associato un intervallo e che il punto contenuto nel nodo divide l'intervallo, sono generalizzabili in n-D.

- Si consideri l'inserimento dei punti: 30, 10, 45, 5, 15, 75, 35, 65. Il risultato è mostrato nella figura accanto:



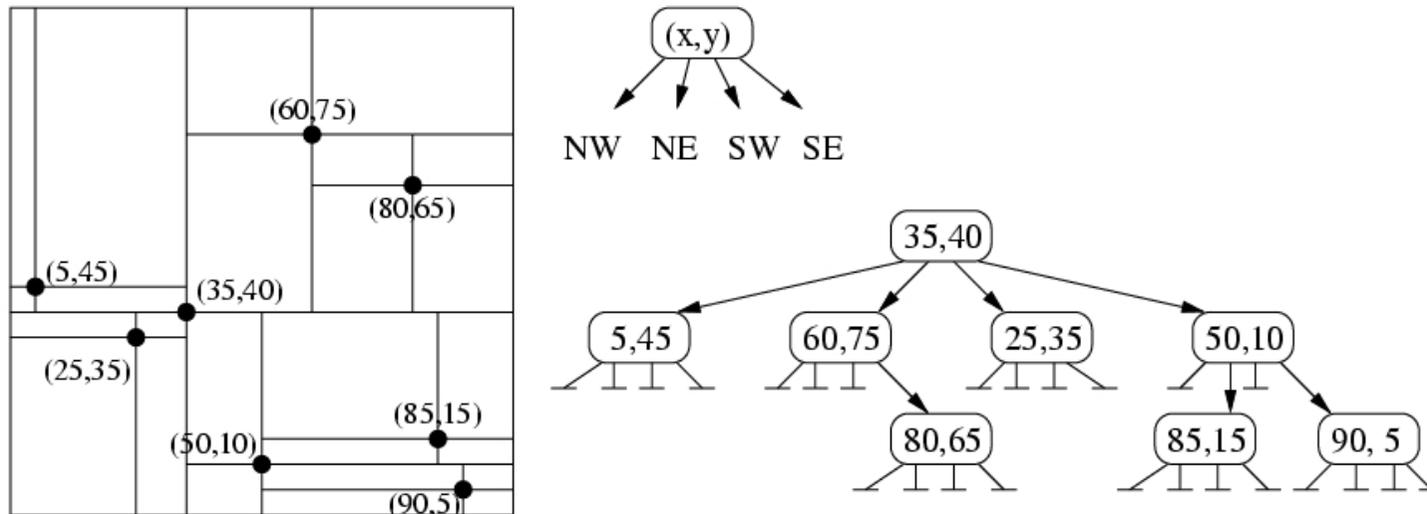
- Supponiamo di voler ricercare i punti che cadono nell'intervallo $R = [a, b]$. Si procede ricorsivamente, attraversando l'albero a partire dalla radice. La visita del nodo v consiste delle seguenti operazioni:
 - se v è una foglia e $\text{point}(v) \in R$, allora ritorna $\text{point}(v)$
 - se v è un nodo interno allora
 - * se $\text{reg}(v) \subset R$ ritorna tutte le foglie del sottoalbero di v
 - * altrimenti, se $\text{point}(v) \in R$ scrivi $\text{point}(v)$, e visita i figli che intersecano R .
- Si vede che questo algoritmo genera due percorsi di ricerca per gli estremi a e b di R . I due percorsi possono avere un tratto in comune e poi si dividono.
- Dopo la divisione, ogni volta che nel percorso di ricerca di a si procede verso sinistra allora tutto il sottoalbero destro del nodo corrente è contenuto interamente in R . Analogamente per b .
- **Complessità:** la costruzione dell'albero richiede $O(n \log n)$ tempo e richiede $O(n)$ spazio. La ricerca dei punti dell'intervallo richiede $O(\log n + \ell)$ tempo, dove ℓ è il numero di punti che cadono nell'intervallo. Infatti;
 - l'elencazione delle foglie di un albero è lineare nel numero delle foglie (perché il numero di nodi interni di un albero binario è minore del numero delle foglie);
 - gli altri nodi che vengono visitati sono i nodi sui percorsi di ricerca di a e b , i quali sono lunghi $O(\log n)$, se l'albero è bilanciato.

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:

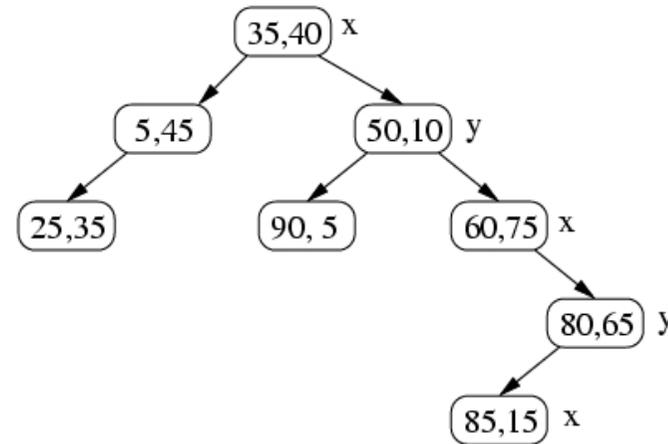
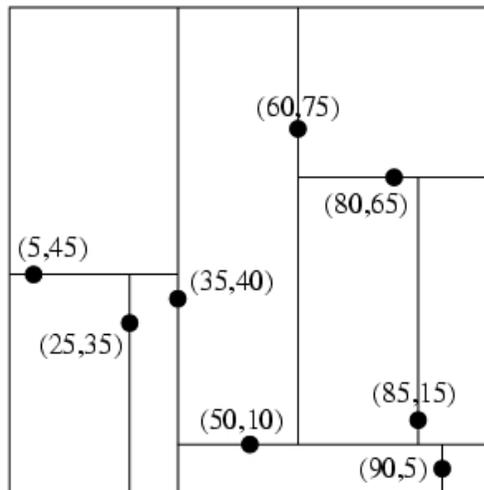


©D. Mount

- La complessità della ricerca (range query) in un point quadtree è $O(\ell + \sqrt{n})$
- 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 verticale 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

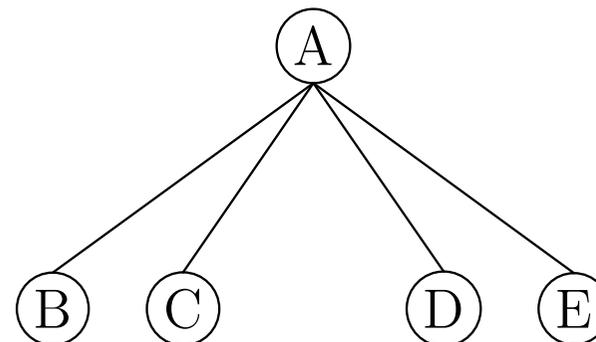
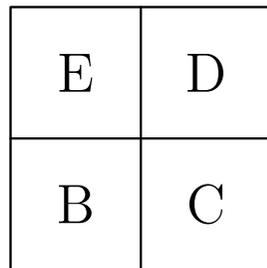
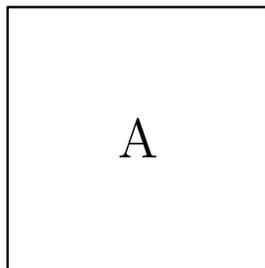
- La ricerca tramite kd-tree dei punti che cadono in un intervallo rettangolare R segue lo stesso schema di quella unidimensionale con alberi binari già vista.
- Come per un albero binario di ricerca, se gli n elementi vengono inseriti a caso, il valore atteso dell'altezza dell'albero è $O(\log n)$. Il costo temporale per costruire un k-D tree è $O(n \log n)$. La complessità computazionale della ricerca (range query) è data da $O(\ell + \sqrt{n})$, dove ℓ è il numero di punti che cadono in R (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.

Strutture dati geometriche

- Nella sezione precedente abbiamo introdotto alcune strutture dati utili nei problemi di ricerca ortogonale.
- Questo ha spostato la nostra attenzione sulle strutture dati geometriche, delle quali continuiamo ad occuparci.
- Illustreremo alcune strutture dati basate sulla suddivisione ricorsiva dello spazio:
Quadtrees, Octrees e Binary Space Partition (BSP) tree.
- Servono, in molti contesti, per dare una organizzazione spaziale ad elementi geometrici (punti, segmenti, poligoni, oggetti tridimensionali).

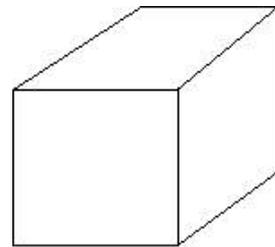
Quadtrees

- Il quadtree è un albero quaternario, simile al point-quadtree, che si costruisce nel seguente modo
 - Si considera un quadrato iniziale grande abbastanza da contenere gli oggetti 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.
 - Si esegue ricorsivamente la suddivisione di ogni nodo fino a quando un quadrante contiene un numero di oggetti (o frammenti di essi) inferiore ad un numero fissato.
- Nelle foglie dell'albero è quindi contenuto il puntatore ad una struttura dati per gli oggetti (o parti di essi) della scena contenuti nel quadrante associato alla foglia.

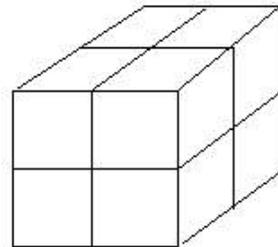


Octrees

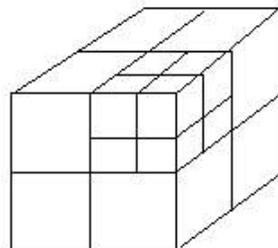
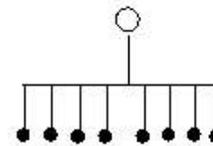
- È immediato estendere il quadtree alle tre dimensioni
- Si ottiene il cosiddetto **octree**
- Ogni cubo viene suddiviso in 8 (da qui il nome), per il resto è identico al quadtree



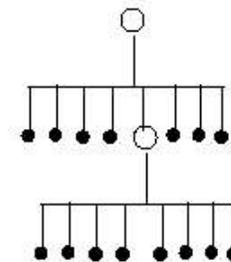
(root)



(1 level)



(2 levels)

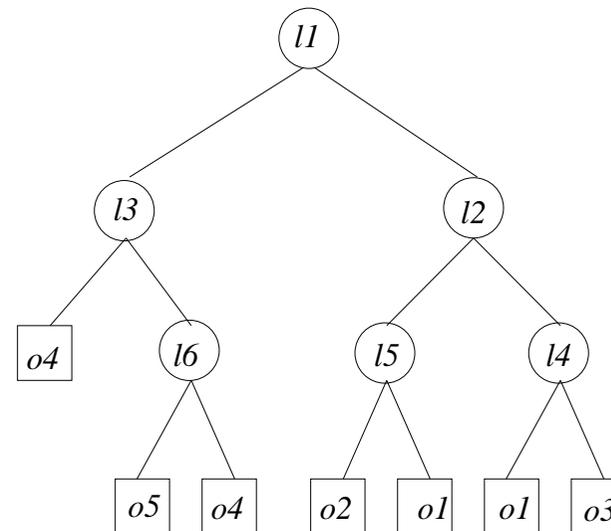
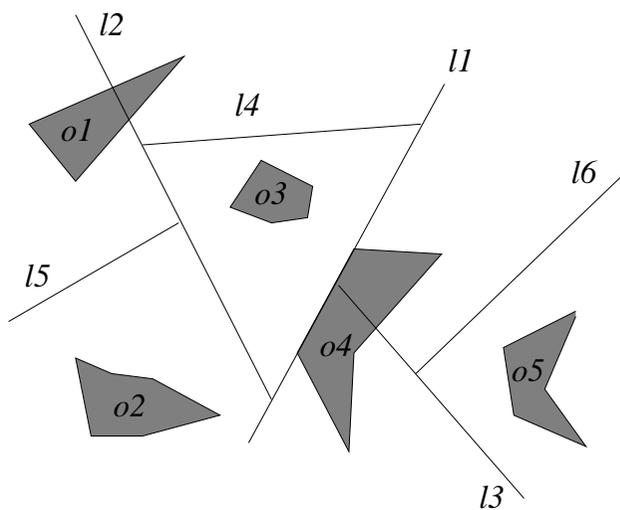


- Nella procedura di suddivisione entrano in gioco due fattori:
 1. **Il numero di oggetti a cui punta una foglia;** più è basso (idealmente uno) e più è alto il beneficio portato dalla struttura ad albero.
 2. **La profondità dell'albero;** più è alta e più ottanti ci sono (più piccoli)
- Bisogna trovare un bilanciamento tra i due fattori; il primo velocizza la ricerca (meno test), ma una profondità troppo alta significa una scarsa efficienza nell'attraversamento dell'albero
- La pratica ha mostrato che gli octree sono efficienti solo quando gli oggetti sono distribuiti uniformemente nella scena
- Se la scena è composta da molti spazi vuoti tra gli oggetti, allora l'octree diventa inefficiente (alta profondità per suddividere essenzialmente lo spazio vuoto)

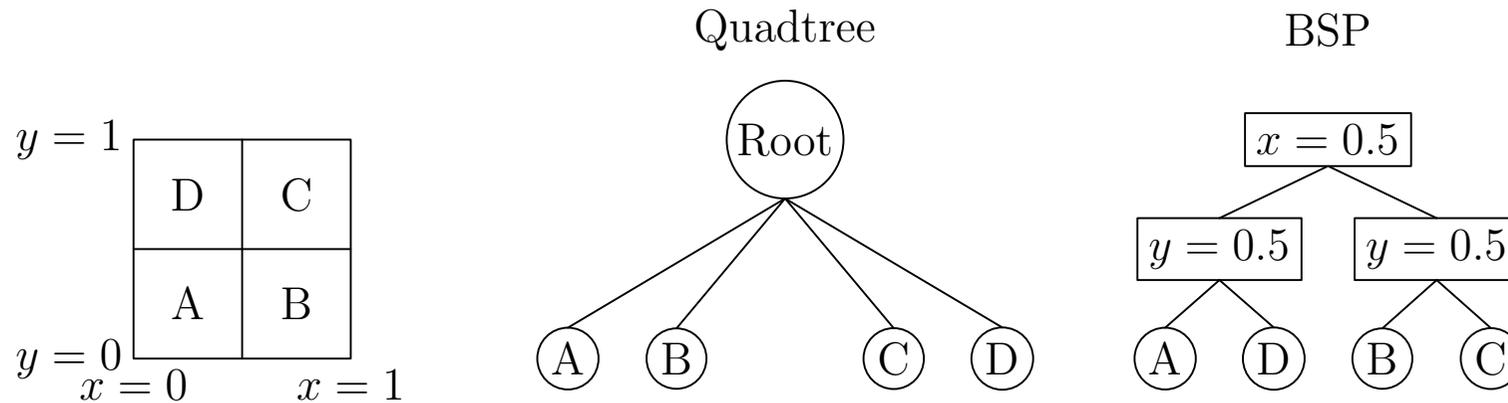
BSP tree

- 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 punto di vista sono potenziali occlusori 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.

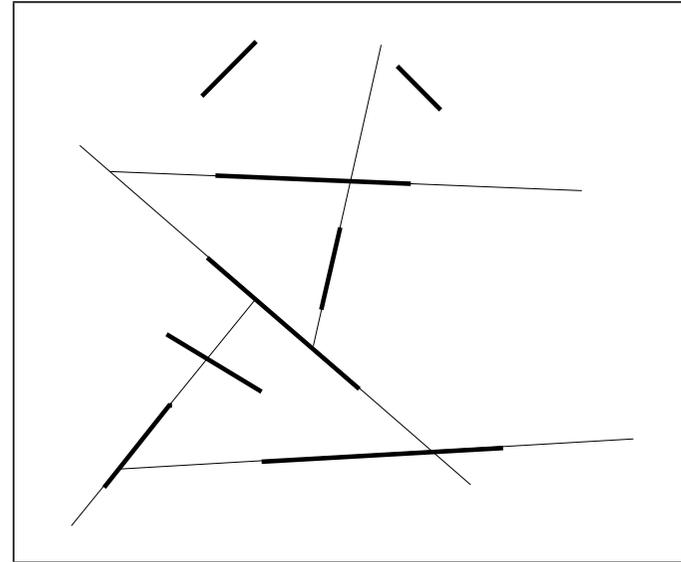


- Vediamo un esempio bidimensionale semplice in modo da compararlo con un quadtree
- Si supponga di avere la suddivisione in figura
- I due alberi, il quadtree ed il BSP, sono semplici da trovare



- 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.